MapReduce 服务

应用开发指南

文档版本 01

发布日期 2024-08-16

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

目 录

1 MRS 应用开发简介

MRS 应用开发概述

MRS是企业级大数据存储、查询、分析的统一平台，能够帮助企业快速构建海量数据信息处理系统，通过对海量信息数据的分析挖掘，发现全新价值点和企业商机。

MRS提供了各组件的常见业务场景样例程序，开发者用户可基于样例工程进行相关数据应用的开发与编译，样例工程依赖的jar包直接通过华为云开源镜像站下载，其他社区开源jar包可从各Maven公共仓库下载。

开发者能力要求

- 您已经对大数据各组件具备一定的认识。
- 您已经对Java语法具备一定的认识。
- 您已经对弹性云服务器的使用方式和MapReduce服务开发组件有一定的了解。
- 您已经对Maven构建方式具备一定的认识和使用方法有一定了解。

MRS 应用开发流程说明

通常MRS应用开发流程如下图所示，各组件应用的开发编译操作可参考组件开发指南对应章节。

图 1-1 MRS 应用开发流程

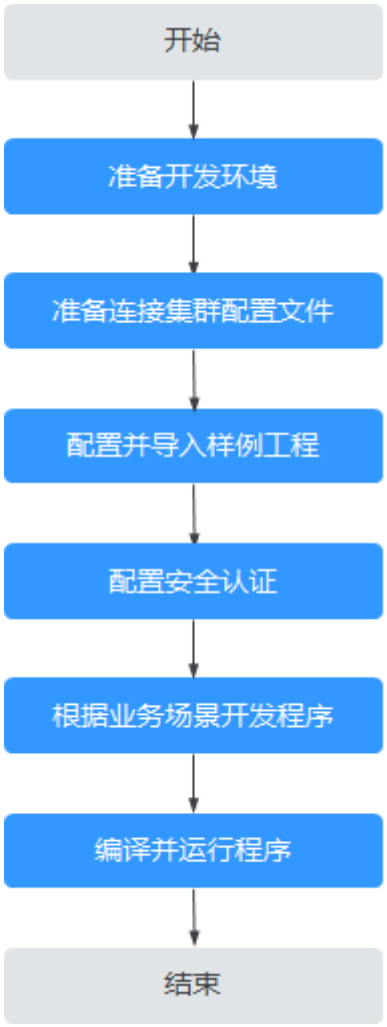


表 1-1 MRS 应用开发流程说明

阶段	说明
准备开发环境	在进行应用开发前，需首先准备开发环境，推荐使用IntelliJ IDEA工具，同时本地需完成JDK、Maven等初始配置。
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通。
配置并导入样例工程	MRS提供了不同组件场景下的多种样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。
配置安全认证	连接开启了Kerberos认证的MRS集群时，应用程序中需配置具有相关资源访问权限的用户进行安全认证。
根据业务场景开发程序	根据实际业务场景开发程序，调用组件接口实现对应功能。

阶段	说明
编译并运行程序	将开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。

2 获取 MRS 应用开发样例工程

MRS 样例工程构建流程

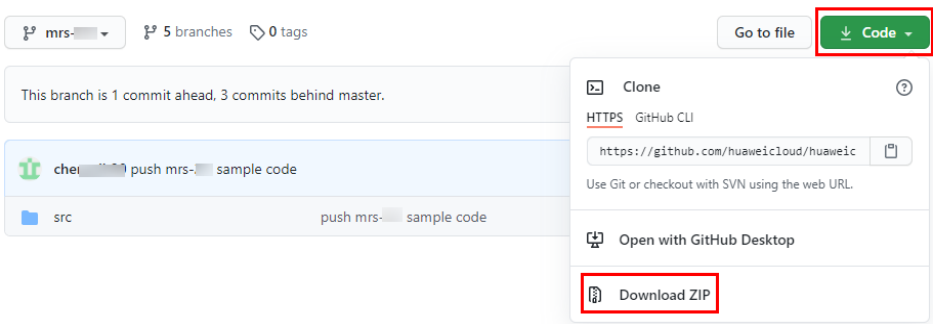
MRS样例工程构建流程包括三个主要步骤：

1. 下载样例工程的Maven工程源码和配置文件，请参见**样例工程获取地址**。
2. 配置华为镜像站中SDK的Maven镜像仓库，请参见**配置华为开源镜像仓**。
3. 根据用户自身需求，构建完整的Maven工程并进行编译开发。

样例工程获取地址

- MRS服务1.8.x版本的样例工程下载地址为：**https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-1.8**。
- MRS服务1.9.x版本的样例工程下载地址为：**https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-1.9**。
- MRS服务2.0.x版本和2.1.x版本的样例工程下载地址为：**https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-2.0**。
- MRS服务3.0.2版本的样例工程下载地址为：**https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-3.0.2**。
- MRS服务3.1.0版本的样例工程下载地址为：**https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-3.1.0**。
- MRS服务3.1.5版本的样例工程下载地址为：**https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-3.1.5**。

图 2-1 样例代码下载



下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

说明

MRS服务3.1.2-LTS版本获取样例工程请参考**通过开源镜像站获取样例工程**。

配置华为开源镜像仓

华为提供开源镜像站，各服务样例工程依赖的jar包都可在华为开源镜像站下载，剩余所依赖的开源jar包请直接从Maven中央库或者其他用户自定义的仓库地址下载。

说明

本地环境使用开发工具下载依赖的jar包前，需要确认以下信息。

- 确认本地环境网络正常。

  打开浏览器访问：华为提供开源镜像站，查看网站是否能正常访问。如果访问异常，请先开通本地网络。

- 确认当前开发工具是否开启代理。下载jar包前需要确保开发工具代理关闭。

  比如以2020.2版本的IntelliJ IDEA开发工具为例，单击“File > Settings > Appearance & Behavior > System Settings > HTTP Proxy”，选择“No proxy”，单击“OK”保存配置。

开源镜像配置方式如下所示：

步骤1 使用前请确保您已安装JDK 1.8及以上版本和Maven 3.0及以上版本。

步骤2 配置Maven配置文件。

- 如果想要覆盖Maven配置文件，在华为开源镜像站（https://mirrors.huaweicloud.com/），选择“华为SDK > HuaweiCloud SDK”，下载华为开源镜像站提供的“settings.xml”文件，覆盖至“*<本地Maven安装目录>*/conf/settings.xml”文件即可。
- 如果不想覆盖Maven配置文件，可以参考以下方法手动修改“settings.xml”配置文件或者组件样例工程中的“pom.xml”文件，配置镜像仓地址。
  - **配置方法一**：修改“settings.xml”配置文件。

    手动在“settings.xml”配置文件的“mirrors”节点中添加以下开源镜像仓地址：

    ```
    <mirror>
      <id>repo2</id>
      <mirrorOf>central</mirrorOf>
      <url>https://repo1.maven.org/maven2/</url>
    </mirror>
    ```

    在“settings.xml”配置文件的“profiles”节点中添加以下镜像仓地址：

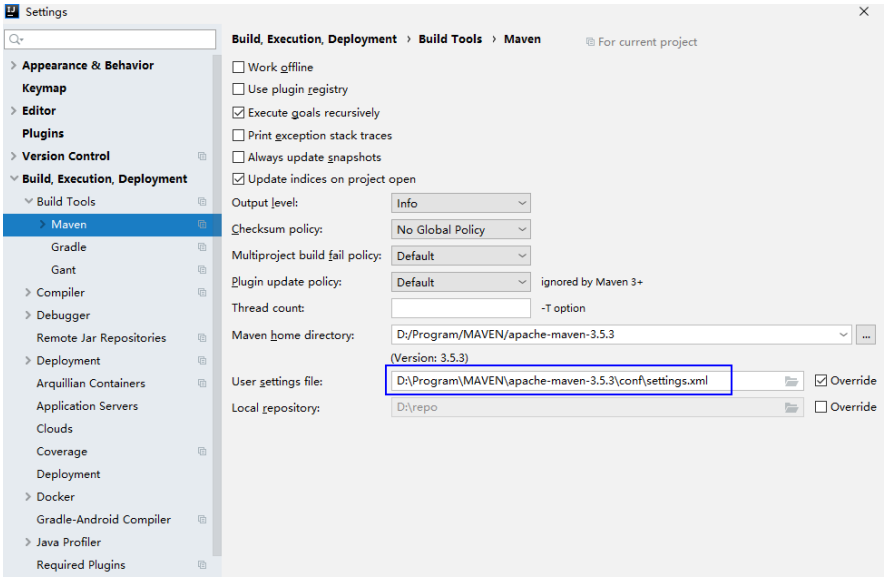
    ```
    <profile>
      <id>huaweicloudsdk</id>
      <repositories>
        <repository>
          <id>huaweicloudsdk</id>
          <url>https://repo.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </repository>
      </repositories>
    </profile>
    ```

    在“settings.xml”配置文件的“activeProfiles”节点中添加如下profile地址：

    ```
    <activeProfile>huaweicloudsdk</activeProfile>
    ```

说明

- 华为开源镜像站不提供第三方开源jar包下载，请配置开源镜像后，额外配置第三方Maven镜像仓库地址。
- 使用IntelliJ IDEA开发工具时，可单击“File > Settings > Build, Execution, Deployment > Build Tools > Maven”查看当前“settings.xml”文件放置目录。



- **配置方法二**：修改“pom.xml”配置文件。

  请直接在二次开发工程样例工程中的“pom.xml”文件添加如下镜像仓地址：

```
<repositories>

    <repository>
        <id>huaweicloudsdk</id>
        <url>https://mirrors.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
        <releases><enabled>true</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
    </repository>

    <repository>
        <id>central</id>
        <name>Maven Central</name>
        <url>https://repo1.maven.org/maven2/</url>
    </repository>

</repositories>
```

步骤3 配置Maven默认编码和JDK。在“settings.xml”配置文件的“profiles”节点中添加以下内容：

```
<profile>
<id>JDK1.8</id>
<activation>
<activeByDefault>true</activeByDefault>
<jdk>1.8</jdk>
</activation>
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<maven.compiler.encoding>UTF-8</maven.compiler.encoding>
<maven.compiler.source>1.8</maven.compiler.source>
<maven.compiler.target>1.8</maven.compiler.target>
<maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
```



```
</properties>
</profile>
```

----结束

3 MRS 各组件样例工程汇总

样例工程获取地址参见**获取MRS应用开发样例工程**，**切换分支为与MRS集群相匹配的版本分支**，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

MRS样例代码库提供了各组件的基本功能样例工程供用户使用，当前版本各组件提供的样例工程汇总参见**表3-1**。

表 3-1 各组件样例工程汇总

组件	样例工程位置	描述
ClickHouse	clickhouse-examples	指导用户基于Java语言，实现MRS集群中的ClickHouse的数据表创建、删除以及数据的插入、查询等操作。 本工程中包含了建立服务端连接、创建数据库、创建数据表、插入数据、查询数据及删除数据表等操作示例。

组件	样例工程位置		描述
Flink	• 开启Kerberos认证集群的样例工程目录“flink-examples/flink-examples-security”。 • 未开启Kerberos认证集群的样例工程目录为“flink-examples/flink-examples-normal”。	FlinkCheckpointJavaExample FlinkCheckpointScalaExample	Flink异步Checkpoint机制的Java/Scala示例程序。 本工程中，程序使用自定义算子持续产生数据，产生的数据为一个四元组（Long，String，String，Integer）。数据经统计后，将统计结果打印到终端输出。每隔6秒钟触发一次checkpoint，然后将checkpoint的结果保存到HDFS中。
		FlinkKafkaJavaExample FlinkKafkaScalaExample	Flink向Kafka生产并消费数据的Java/Sacla示例程序。 在本工程中，假定某个Flink业务每秒就会收到1个消息记录，启动Producer应用向Kafka发送数据，然后启动Consumer应用从Kafka接收数据，对数据内容进行处理后并打印输出。

组件	样例工程位置		描述
		FlinkPipelineJavaExample	Flink Job Pipeline的Java/Sacla示例程序。 本样例中一个发布者Job自己每秒钟产生10000条数据，另外两个Job作为订阅者，分别订阅一份数据。订阅者收到数据之后将其转化格式，并抽样打印输出。
		FlinkPipelineScalaExample	
		FlinkSqlJavaExample	使用客户端通过jar作业提交SQL作业的应用开发示例。
		FlinkStreamJavaExample	Flink构造DataStream的Java/Sacla示例程序。 本工程示例为基于业务要求分析用户日志数据，读取文本数据后生成相应的DataStream，然后筛选指定条件的数据，并获取结果。
		FlinkStreamScalaExample	

组件	样例工程位置		描述
		FlinkStreamSqlJoinExample	Flink SQL Join示例程序。 本工程示例调用flink-connector-kafka模块的接口，生产并消费数据。生成Table1和Table2，并使用Flink SQL对Table1和Table2进行联合查询，打印输出结果。
HBase	hbase-examples	hbase-example	HBase数据读写操作的应用开发示例。 通过调用HBase接口可实现创建用户表、导入用户数据、增加用户信息、查询用户信息及为用户表创建二级索引等功能。
		hbase-rest-example	HBase Rest接口应用开发示例。 使用Rest接口实现查询HBase集群信息、获取表、操作NameSpace、操作表等功能。

组件	样例工程位置		描述
		hbase-thrift-example	访问HBase ThriftServer应用开发示例。 访问ThriftServer操作表、向表中写数据、从表中读数据。
		hbase-zk-example	HBase访问ZooKeeper应用开发示例。 在同一个客户端进程内同时访问MRS ZooKeeper和第三方的ZooKeeper，其中HBase客户端访问MRS ZooKeeper，客户应用访问第三方ZooKeeper。
HDFS	• 开启Kerberos认证集群的样例工程目录“hdfs-example-security”。 • 未开启Kerberos认证集群的样例工程目录为“hdfs-example-normal”。		HDFS文件操作的Java示例程序。 本工程主要给出了创建HDFS文件夹、写文件、追加文件内容、读文件和删除文件/文件夹等相关接口操作示例。

组件	样例工程位置	描述
	hdfs-c-example	HDFS C语言开发代码样例。 本示例提供了基于C语言的HDFS文件系统连接、文件操作如创建文件、读写文件、追加文件、删除文件等。
Hive	hive-jdbc-example	Hive JDBC处理数据Java示例程序。 本工程使用JDBC接口连接Hive，在Hive中执行相关数据操作。使用JDBC接口实现创建表、加载数据、查询数据等功能，还可实现在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper。
	hive-jdbc-example-multizk	
	hcatalog-example	Hive HCatalog处理数据Java示例程序。 使用HCatalog接口实现通过Hive命令行方式对MRS Hive元数据进行数据定义和查询操作。

组件	样例工程位置	描述
	python3-examples	使用Python3连接Hive执行SQL样例。 可实现使用Python3对接Hive并提交数据分析任务。
Kafka	kafka-examples	Kafka流式数据的处理Java示例程序。 本工程基于Kafka Streams完成单词统计功能，通过读取输入Topic中的消息，统计每条消息中的单词个数，从输出Topic消费数据，然后将统计结果以Key-Value的形式输出。
Manager	manager-examples	FusionInsight Manager API接口调用示例。 本工程调用Manager API接口实现集群用户的创建、修改及删除等操作。

组件	样例工程位置		描述
MapReduce	• 开启Kerberos认证集群的样例工程目录“mapreduce-example-security”。 • 未开启Kerberos认证集群的样例工程目录为“mapreduce-example-normal”。		MapReduce任务提交Java示例程序。 本工程提供了一个MapReduce统计数据的应用开发示例，实现数据分析、处理，并输出满足用户需要的数据信息。 另外以MapReduce访问HDFS、HBase、Hive为例，介绍如何编写MapReduce作业访问多个服务组件。帮助用户理解认证、配置加载等关键使用方式。
Oozie	• 开启Kerberos认证集群的样例工程目录“oozie-examples/ooziesecurity-examples”。 • 未开启Kerberos认证集群的样例工程目录为“oozie-examples/oozienormal-examples”。	OozieMapReduceExample	Oozie提交MapReduce任务示例程序。 本示例演示了如何通过Java API提交MapReduce作业和查询作业状态，对网站的日志文件进行离线分析。
		OozieSparkHBaseExample	使用Oozie调度Spark访问HBase的示例程序。
		OozieSparkHiveExample	使用Oozie调度Spark访问Hive的示例程序。

组件	样例工程位置		描述
Spark	• 开启Kerberos认证集群的样例工程目录“spark-examples/sparksecurity-examples”。 • 未开启Kerberos认证集群的样例工程目录为“spark-examples/sparknormal-examples”。	SparkHbasetoCarbonJavaExample	Spark同步HBase数据到CarbonData的Java示例程序。 本示例工程中，应用将数据实时写入HBase，用于点查业务。数据每隔一段时间批量同步到CarbonData表中，用于分析型查询业务。
		SparkHbasetoHbaseJavaExample	Spark从HBase读取数据再写入HBase的Java/Scala/Python示例程序。 本示例工程中，Spark应用程序实现两个HBase表数据的分析汇总。
		SparkHbasetoHbasePythonExample	
		SparkHbasetoHbaseScalaExample	
		SparkHivetoHbaseJavaExample	Spark从Hive读取数据再写入到HBase的Java/Scala/Python示例程序。 本示例工程中，Spark应用程序实现分析处理Hive表中的数据，并将结果写入HBase表。
		SparkHivetoHbasePythonExample	
		SparkHivetoHbaseScalaExample	

组件	样例工程位置		描述
		SparkJavaExample	Spark Core任务的Java/Python/Scala/R示例程序。 本工程应用程序实现从HDFS上读取文本数据并计算分析。 SparkRExample示例不支持未开启Kerberos认证的集群。
		SparkPythonExample	
		SparkScalaExample	
		SparkRExample	
		SparkLauncherJavaExample	使用Spark Launcher提交作业的Java/Scala示例程序。 本工程应用程序通过org.apache.spark.launcher.SparkLauncher类采用Java/Scala命令方式提交Spark应用。
		SparkLauncherScalaExample	
		SparkOnHbaseJavaExample	Spark on HBase场景的Java/Scala/Python示例程序。 本工程应用程序以数据源的方式去使用HBase，将数据以Avro格式存储在HBase中，并从中读取数据以及对读取的数据进行过滤等操作。
		SparkOnHbasePythonExample	
		SparkOnHbaseScalaExample	

组件	样例工程位置		描述
		SparkOnHudiJavaExample	Spark on Hudi场景的Java/Scala/Python示例程序。 本工程应用程序使用Spark操作Hudi执行插入数据、查询数据、更新数据、增量查询、特定时间点查询、删除数据等操作。
		SparkOnHudiPythonExample	
		SparkOnHudiScalaExample	
		SparkOnMultiHbaseScalaExample	Spark同时访问两个集群中的HBase的Scala示例程序。 本示例不支持未开启Kerberos认证的集群。
		SparkSQLJavaExample	Spark SQL任务的Java/Python/Scala示例程序。 本工程应用程序实现从HDFS上读取文本数据并计算分析。
		SparkSQLPythonExample	
		SparkSQLScalaExample	
		SparkStreamingKafka010JavaExample	Spark Streaming从Kafka接收数据并进行统计分析的Java/Scala示例程序。 本工程应用程序实时累加计算Kafka中的流数据，统计每个单词的记录总数。
		SparkStreamingKafka010ScalaExample	

组件	样例工程位置		描述
		SparkStreamingtoHbaseJavaExample010	Spark Streaming读取Kafka数据并写入HBase的Java/Scala/Python示例程序。 本工程应用程序每5秒启动一次任务，读取Kafka中的数据并更新到指定的HBase表中。
		SparkStreamingtoHbasePythonExample010	
		SparkStreamingtoHbaseScalaExample010	
		SparkStructuredStreamingJavaExample	在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。
		SparkStructuredStreamingPythonExample	
		SparkStructuredStreamingScalaExample	
		SparkThriftServerJavaExample	通过JDBC访问Spark SQL的Java/Scala示例程序。 本示例中，用户自定义JDBCServer的客户端，使用JDBC连接来进行表的创建、数据加载、查询和删除。
		SparkThriftServerScalaExample	

组件	样例工程位置		描述
		StructuredStreamingADScalaExample	使用Structured Streaming，从kafka中读取广告请求数据、广告展示数据、广告点击数据，实时获取广告有效展示统计数据和广告有效点击统计数据，将统计结果写入kafka中。
		StructuredStreamingStateScalaExample	在Spark结构流应用中，跨批次统计每个session期间发生了多少次event以及本session的开始和结束timestamp；同时输出本批次被更新状态的session。

4 MRS 应用开发开源 jar 包冲突列表说明

4.1 HBase

HBase jar 包冲突列表

Jar包名称	描述
hbase-client-2.2.3-*.jar	连接HBase服务必须的jar包。
zookeeper-*.jar	连接ZooKeeper服务必须的jar包。

解决方案

- 使用MRS集群的ZooKeeper包“zookeeper*.jar”。
- 使用exclusions排除掉hbase-client里面的zookeeper。

4.2 HDFS

HDFS jar 包冲突列表

Jar包名称	描述	处理方案
hadoop-plugins-*.jar	HDFS可以直接使用开源同版本的hadoop包运行样例代码，但是MRS 3.x之后的版本默认的主备倒换类是dfs.client.failover.proxy.provider.hacluster=org.apache.hadoop.hdfs.server.namenode.ha.AdaptiveFailoverProxyProvider，默认HDFS的LZC压缩格式类为io.compression.codec.lzc.class=com.huawei.hadoop.datasight.io.compress.lzc.ZCodec。	• 方式一：参考样例代码里面的pom.xml文件，增加配置： <properties> <hadoop.ext.version>*8.0.2-302002*</hadoop.ext.version> </properties> ... <dependency> <groupId>com.huawei.mrs</groupId> <artifactId>hadoop-plugins</artifactId> <version>${hadoop.ext.version}</version> </dependency> • 方式二： 1. 将“hdfs-site.xml”配置文件里面的参数dfs.client.failover.proxy.provider.hacluster改为与开源一致的值“org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider”。 2. 不使用LZC压缩格式。

4.3 Kafka

Kafka jar 包冲突列表

Jar包名称	描述
kafka_2.11-*.jar	连接Kafka服务必须的jar包。
kafka-clients-*.jar	连接Kafka服务必须的jar包。

解决方案

Kafka不建议使用开源版本的包。

4.4 Spark2x

Spark2x jar 包冲突列表

Jar包名称	描述	处理方案
spark-core_2.1.1-*.jar	Spark任务的核心jar包。	Spark可以直接使用开源同版本的spark包运行样例代码，但是不同版本的spark-core包在使用的时候可能互相序列化ID不一样，建议使用集群自带jar包。
jackson-*.jar	执行Spark程序时报错： com.fasterxml.jackson.databind.JsonMappingException: Scala module 2.11.4 requires Jackson Databind version >= 2.11.0 and < 2.12.0	执行程序时引入的jackson相关包与集群自带的包版本不一致，导致报错，建议使用集群自带的jackson相关jar包。 集群jar包路径：*客户端安装目录*/Spark2x/spark/jars”或者“客户端安装目录/Spark/spark/jars”。

说明

Spark jar包冲突也可以参考**常见jar包冲突处理方式**。

5 MRS 组件 jar 包版本与集群对应关系说明

MRS 3.1.5

表 5-1 MRS 3.1.5 版本集群 jar 版本

组件	组件版本	jar版本
Flink	1.12.2	1.12.2-hw-ei-315008
Hive	3.1.0	3.1.0-hw-ei-315008
Tez	0.9.2	0.9.1.0101-hw-ei-315008
Spark2x	3.1.1	3.1.1-hw-ei-315008
CarbonData	2.2.0	-
Hadoop	3.1.1	3.1.1-hw-ei-315008
HBase	2.2.3	2.2.3-hw-ei-315008
ZooKeeper	3.6.3	3.6.3-hw-ei-315008
Hue	4.7.0	-
Oozie	5.1.0	5.1.0-hw-ei-315008
Flume	1.9.0	-
Kafka	2.4.0	2.4.0-hw-ei-315008
Ranger	2.0.0	2.0.0-hw-ei-315008
Phoenix	5.0.0	5.0.0-HBase-2.0-hw-ei-315008

组件	组件版本	jar版本
ClickHouse	21.3.4.25	0.3.1-hw-ei-315008
Presto	333	333-hw-ei-315008
scala	2.11	-

MRS 3.1.0

表 5-2 MRS 3.1.0 版本集群 jar 版本

组件	组件版本	jar版本
Flink	1.12.0	1.12.0-hw-ei-310003
Hive	3.1.0	3.1.0-hw-ei-310003
Tez	0.9.2	0.9.1.0101-hw-ei-12
Spark2x	2.4.5	2.4.5-hw-ei-310003
CarbonData	2.0.1	-
Hadoop	3.1.1	3.1.1-hw-ei-310003
HBase	2.2.3	2.2.3-hw-ei-310003
ZooKeeper	3.5.6	3.5.6-hw-ei-310003
Hue	4.7.0	-
Oozie	5.1.0	5.1.0-hw-ei-310003
Flume	1.9.0	-
Kafka	2.4.0	2.4.0-hw-ei-310003
Ranger	2.0.0	-
Phoenix	5.0.0	5.0.0-HBase-2.0-hw-ei-310003
ClickHouse	21.3.4.25	0.3.0
Presto	316	316-hw-ei-310003
scala	2.11	-

MRS 3.0.x

表 5-3 MRS 3.0.x 版本集群 jar 版本

组件	组件版本	jar版本
Flink	1.10.0	1.10.0-hw-ei-302002
Hive	3.1.0	3.1.0-hw-ei-302002
Tez	0.9.2	0.9.1.0101-hw-ei-12
Spark	2.4.5	2.4.5-hw-ei-302002
CarbonData	2.0.0	-
Hadoop	3.1.1	3.1.1-hw-ei-302002
HBase	2.2.3	2.2.3-hw-ei-302002
ZooKeeper	3.5.6	3.5.6-hw-ei-302002
Hue	4.7.0	-
Oozie	5.1.0	5.1.0-hw-ei-302002
Flume	1.9.0	-
Kafka	2.4.0	2.4.0-hw-ei-302002
Ranger	2.0.0	-
Storm	1.2.0	1.2.1-hw-ei-302002
Phoenix	5.0.0	5.0.0-HBase-2.0-hw-ei-302002
Presto	316	316-hw-ei-302002
scala	2.11	-

MRS 2.1.x

表 5-4 MRS 2.1.x 版本集群 jar 版本

组件	组件版本	jar版本
Zookeeper	3.5.1	3.5.1-mrs-2.1
Hadoop	3.1.1	3.1.1-mrs-2.1
HBase	2.1.1	2.1.1-mrs-2.1
Tez	0.9.1	0.9.1.0101-hw-ei-12
Hive	3.1.0	3.1.0-mrs-2.1

组件	组件版本	jar版本
Hive_Spark	1.2.1	1.2.1.spark_2.3.2-mrs-2.1
Spark	2.3.2	2.3.2-mrs-2.1
Carbon	1.5.1	-
Presto	308	-
Kafka	1.1.0	1.1.0-mrs-2.1
KafkaManager	1.3.3.18	-
Flink	1.7.0	1.7.0-mrs-2.1
Storm	1.2.1	1.2.1-mrs-2.1
Flume	1.6.0	1.6.0-mrs-2.1
Hue	3.11.0	-
Loader(Sqoop)	1.99.7	1.99.7-mrs-2.1
scala	2.11	NA

MRS 2.0.x

表 5-5 2.0.x 版本集群 jar 版本

组件	组件版本	jar版本
Zookeeper	3.5.1	3.5.1-mrs-2.0
Hadoop	3.1.1	3.1.1-mrs-2.0
HBase	2.1.1	2.1.1-mrs-2.0
Tez	0.9.1	0.9.1.0101-hw-ei-12
Hive	3.1.0	3.1.0-mrs-2.0
Hive_Spark	1.2.1	1.2.1.spark_2.3.2-mrs-2.0
Spark	2.3.2	2.3.2-mrs-2.0
Carbon	1.5.1	-
Presto	308	-
Kafka	1.1.0	1.1.0-mrs-2.0
KafkaManager	1.3.3.18	-
Storm	1.2.1	1.2.1-mrs-2.0
Flume	1.6.0	1.6.0-mrs-2.0

组件	组件版本	jar版本
Hue	3.11.0	-
Loader(Sqoop)	1.99.7	1.99.7-mrs-2.0
scala	2.11	-

MRS 1.9.x

表 5-6 MRS 1.9.x 版本集群 jar 版本

组件	组件版本	jar版本
Zookeeper	3.5.1	3.5.1-mrs-1.9.0
Hadoop	2.8.3	2.8.3-mrs-1.9.0
HBase	1.3.1	1.3.1-mrs-1.9.0
OpenTSDB	2.3.0	-
Tez	0.9.1	0.9.1.0101-hw-ei-12
Hive	2.3.3	2.3.3-mrs-1.9.0
Hive_Spark	1.2.1	1.2.1.spark_2.2.1-mrs-1.9.0
Spark	2.2.2	2.2.2-mrs-1.9.0
Carbon	1.6.1	-
Presto	0.216	0.216-mrs-1.9
Kafka	1.1.0	1.1.0-mrs-1.9.0
KafkaManager	1.3.3.1	-
Flink	1.7.0	1.7.0-mrs-1.9.0
Storm	1.2.1	1.2.1-mrs-1.9.0
Flume	1.6.0	1.6.0-mrs-1.9.0
Hue	3.11.0	-
Loader(Sqoop)	1.99.7	1.99.7-mrs-1.9.0
scala	2.11	-

MRS 1.8.x

表 5-7 MRS 1.8.x 版本集群 jar 版本

组件	组件版本	jar版本
Zookeeper	3.5.1	3.5.1-mrs-1.8.0
Hadoop	2.8.3	2.8.3-mrs-1.8.0
HBase	1.3.1	1.3.1-mrs-1.8.0
OpenTSDB	2.3.0	-
Hive	1.3.0	1.3.0-mrs-1.8.0
Hive_Spark	1.2.1	1.2.1.spark_2.2.1-mrs-1.8.0
Spark	2.2.1	2.2.1-mrs-1.8.0
Carbon	1.6.1	-
Presto	0.215	0.215-mrs-1.8.0
Kafka	1.1.0	1.1.0-mrs-1.8.0
KafkaManager	1.3.3.18	-
Flink	1.7.0	1.7.0-mrs-1.8.0
Storm	1.2.1	1.2.1-mrs-1.8.0
Flume	1.6.0	1.6.0-mrs-1.8.0
Hue	3.11.0	-
Loader(Sqoop)	1.99.7	1.99.7-mrs-1.8.0
scala	2.11	-

6 MRS 应用开发安全认证说明

6.1 MRS 安全认证原理和认证机制

功能

开启了Kerberos认证的安全模式集群，进行应用开发时需要进行安全认证。

使用Kerberos的系统在设计上采用“客户端/服务器”结构与AES等加密技术，并且能够进行相互认证（即客户端和服务器端均可对对方进行身份认证）。可以用于防止窃听、防止replay攻击、保护数据完整性等场合，是一种应用对称密钥体制进行密钥管理的系统。

结构

Kerberos的原理架构如图6-1所示，各模块的说明如表6-1所示。

图 6-1 原理架构

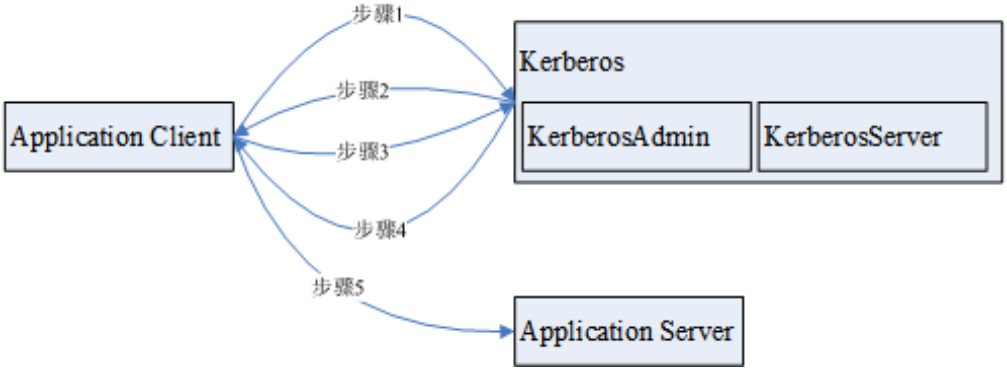


表 6-1 模块说明

模块	说明
Application Client	应用客户端，通常是需要提交任务（或者作业）的应用程序。
Application Server	应用服务端，通常是应用客户端需要访问的应用程序。
Kerberos	提供安全认证的服务。
KerberosAdmin	提供认证用户管理的进程。
KerberosServer	提供认证票据分发的进程。

步骤原理说明：

应用客户端（Application Client）可以是集群内某个服务，也可以是客户二次开发的一个应用程序，应用程序可以向应用服务提交任务或者作业。

1. 应用程序在提交任务或者作业前，需要向Kerberos服务申请TGT（Ticket-Granting Ticket），用于建立和Kerberos服务器的安全会话。
2. Kerberos服务在收到TGT请求后，会解析其中的参数来生成对应的TGT，使用客户端指定的用户名的密钥进行加密响应消息。
3. 应用客户端收到TGT响应消息后，解析获取TGT，此时，再由应用客户端（通常是rpc底层）向Kerberos服务获取应用服务端的ST（Server Ticket）。
4. Kerberos服务在收到ST请求后，校验其中的TGT合法后，生成对应的应用服务的ST，再使用应用服务密钥将响应消息进行加密处理。
5. 应用客户端收到ST响应消息后，将ST打包到发给应用服务的消息里面传输给对应的应用服务端（Application Server）。
6. 应用服务端收到请求后，使用本端应用服务对应的密钥解析其中的ST，并校验成功后，本次请求合法通过。

基本概念

以下为常见的基本概念，可以帮助用户减少学习Kerberos框架所花费的时间，有助于更好的理解Kerberos业务。以HDFS安全认证为例：

TGT

票据授权票据（Ticket-Granting Ticket），由Kerberos服务生成，提供给应用程序与Kerberos服务器建立认证安全会话，该票据的默认有效期为24小时，24小时后该票据自动过期。

TGT申请方式(以HDFS为例)：

1. 通过HDFS提供的接口获取。

```
/**
 * login Kerberos to get TGT, if the cluster is in security mode
 * @throws IOException if login is failed
 */
private void login() throws IOException {
// not security mode, just return
  if (! "kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
```

```
        return;
    }

    //security mode
    System.setProperty("java.security.krb5.conf", PATH_TO_KRB5_CONF);

    UserGroupInformation.setConfiguration(conf);
    UserGroupInformation.loginUserFromKeytab(PRNCIPAL_NAME, PATH_TO_KEYTAB);
  }
```

2. 通过客户端shell命令以**kinit**方式获取。

ST

服务票据（Server Ticket），由Kerberos服务生成，提供给应用程序与应用服务建立安全会话，该票据一次性有效。

ST的生成在FusionInsight产品中，基于hadoop-rpc通信，由rpc底层自动向Kerberos服务端提交请求，由Kerberos服务端生成。

认证代码实例讲解

```
package com.huawei.bigdata.hdfs.examples;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.security.UserGroupInformation;

public class KerberosTest {
    private static String PATH_TO_HDFS_SITE_XML = KerberosTest.class.getClassLoader().getResource("hdfs-site.xml")
            .getPath();
    private static String PATH_TO_CORE_SITE_XML = KerberosTest.class.getClassLoader().getResource("core-site.xml")
            .getPath();
    private static String PATH_TO_KEYTAB = KerberosTest.class.getClassLoader().getResource("user.keytab").getPath();
    private static String PATH_TO_KRB5_CONF = KerberosTest.class.getClassLoader().getResource("krb5.conf").getPath();
    private static String PRNCIPAL_NAME = "develop";
    private FileSystem fs;
    private Configuration conf;

    /**
     * initialize Configuration
     */
    private void initConf() {
        conf = new Configuration();

        // add configuration files
        conf.addResource(new Path(PATH_TO_HDFS_SITE_XML));
        conf.addResource(new Path(PATH_TO_CORE_SITE_XML));
    }

    /**
     * login Kerberos to get TGT, if the cluster is in security mode
     * @throws IOException if login is failed
     */
    private void login() throws IOException {
        // not security mode, just return
        if (! "kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
            return;
        }

        //security mode
```

```
        System.setProperty("java.security.krb5.conf", PATH_TO_KRB5_CONF);

        UserGroupInformation.setConfiguration(conf);
        UserGroupInformation.loginUserFromKeytab(PRNCIPAL_NAME, PATH_TO_KEYTAB);
    }

    /**
     * initialize FileSystem, and get ST from Kerberos
     * @throws IOException
     */
    private void initFileSystem() throws IOException {
        fs = FileSystem.get(conf);
    }

    /**
     * An example to access the HDFS
     * @throws IOException
     */
    private void doSth() throws IOException {
        Path path = new Path("/tmp");
        FileStatus fStatus = fs.getFileStatus(path);
        System.out.println("Status of " + path + " is " + fStatus);
        //other thing
    }


    public static void main(String[] args) throws Exception {
        KerberosTest test = new KerberosTest();
        test.initConf();
        test.login();
        test.initFileSystem();
        test.doSth();
    }
}
```

说明

- Kerberos认证时需要配置Kerberos认证所需要的文件参数，主要包含keytab路径，Kerberos认证的用户名称，Kerberos认证所需要的客户端配置krb5.conf文件。
- 方法login()为调用hadoop的接口执行Kerberos认证，生成TGT票据。
- 方法doSth()调用hadoop的接口访问文件系统，此时底层RPC会自动携带TGT去Kerberos认证，生成ST票据。
- 以上代码可在安全模式下的HDFS二次开发样例工程中创建KerberosTest.java，运行并查看调测结果，具体操作过程请参考**HDFS开发指南（安全模式）**。

6.2 准备 MRS 应用开发用户

操作场景

开发用户用于运行样例工程。进行不同服务的组件开发时，需要赋予不同的用户权限。

操作步骤

步骤1 登录FusionInsight Manager。

步骤2 在FusionInsight Manager界面选择“系统 > 权限 > 角色 > 添加角色”。

1. 填写角色的名称，例如**developrole**，单击“确定”保存角色。
2. 参考**如何判断某个服务是否使用了Ranger鉴权**，确认服务是否启用了Ranger鉴权?

- 是，执行**步骤3**。
- 否，编辑角色，根据服务的权限控制类别添加业务开发时需要的权限，参见**表6-2**。

表 6-2 权限列表

服务	所需添加权限
HDFS	在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统”，勾选“hdfs://hacluster”的“读”、“写”和“执行”，单击“确定”保存。
Mapreduce/ Yarn	1. 在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/”，勾选“user”的“读”、“写”和“执行”。继续选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > user”，勾选“mapred”的“读”、“写”、“执行”； 如果要执行多组件用例，还需： 选择“*待操作集群的名称* > HBase > HBase Scope > global”勾选“default”的“创建”。 选择“待操作集群的名称 > HBase > HBase Scope > global > hbase”，勾选“hbase:meta”的“执行”。 选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > user”，勾选“hive”的“读”、“写”、“执行”。 选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > user > hive”，勾选“warehouse”的“读”、“写”、“执行”。 选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > tmp”，勾选“hive-scratch”的“读”、“写”、“执行”，若存在“examples”，勾选“examples”的“读”、“写”、“执行”和“递归”。 选择“*待操作集群的名称* > Hive > Hive读写权限”，勾选“default”的“查询”、“插入”、“建表”、“递归”。单击“确定”保存。 2. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选default的“提交”，单击“确定”保存。
HBase	在“配置资源权限”的表格中选择“*待操作集群的名称* > HBase > HBase Scope ”，勾选“global”的“管理”、“创建”、“读”、“写”和“执行”，单击“确定”保存。

服务	所需添加权限
Spark2x	1. （若安装了HBase，则配置）在“配置资源权限”表格中选择“*待操作集群的名称* > HBase > HBase Scope > global”，勾选“default”的“创建”，单击“确定”保存。 2. （若安装了HBase，则配置）编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > HBase > HBase Scope > global > hbase”，勾选“hbase:meta”的“执行”，单击“确定”保存。 3. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > user”，勾选“hive”的“执行”，单击“确定”保存。 4. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ >user >hive”，勾选“warehouse”的“读”、“写”和“执行”，单击“确定”保存。 5. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > Hive > Hive读写权限”，勾选“default”的“建表”，单击“确定”保存。 6. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选“default”的“提交”，单击“确定”保存。
Hive	在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选“default”的“提交”和“管理”，单击“确定”保存。 **说明** Hive应用开发需要到的额外的操作权限需要从系统管理员处获取。
ClickHouse	在“配置资源权限”的表格中选择“*待操作集群的名称* > ClickHouse > Clickhouse Scope”，勾选对应数据库的创建权限。单击对应的数据库名称，根据不同任务场景，勾选对应表的“读”、“写”权限，单击“确定”保存。
Flink	1. 在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > flink ”，勾选“读”、“写”和“执行”，单击“配置资源权限”表格中“服务”返回。 2. 在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选default的“提交”，单击“确定”保存。 **说明** 若指定state backend为HDFS，例如指定state backend目录为"hdfs://hacluster/flink-checkpoint"，需同样为"hdfs://hacluster/flink-checkpoint"目录配置“读”、“写”和“执行”权限。
GraphBase	-
Kafka	-

服务	所需添加权限
Impala	-
Storm/CQL	-
Oozie	1. 在“配置资源权限”的表格中选择“*待操作集群的名称* > Oozie > 普通用户权限”，单击“确定”保存。 2. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统”，勾选“hdfs://hacluster”的“读”、“写”和“执行”，单击“确定”保存。 3. 编辑角色，在“权限”表格中选择“*待操作集群的名称* > Yarn”，勾选“集群管理操作权限”，单击“确定”保存。

步骤3 单击“系统 > 权限 > 用户组 > 添加用户组”，为样例工程创建一个用户组，例如**developgroup**。

步骤4 单击“系统 > 权限 > 用户 > 添加用户”，为样例工程创建一个用户。

步骤5 填写用户名，例如*developuser*，按照**表6-3**选择相应的用户类型和需加入的用户组，并绑定角色**developrole**取得权限，单击“确定”。

表 6-3 用户类型和用户组列表

服务	用户类型	需加入的用户组
HDFS	机机	加入**developgroup**和**supergroup**组，设置其“主组”为**supergroup**。
Mapreduce/Yarn	机机	加入**developgroup**组。
HBase	机机	加入**hadoop**组。
Spark2x	机机/人机	加入**developgroup**组。 若用户需要对接Kafka，则需要添加kafkaadmin用户组。
Hive	机机/人机	加入**hive**组。
Kafka	机机	加入**kafkaadmin**组。

服务	用户类型	需加入的用户组
Impala	机机	加入**impala**和**supergroup**组，设置其“主组”为**supergroup**。
Storm/CQL	人机	加入**storm**组。
ClickHouse	人机	加入**developgroup**和**supergroup**组，设置其“主组”为**supergroup**，并添加具有ClickHouse权限的角色。
Oozie	人机	加入**hadoop**、**supergroup**、**hive**组。若使用Hive多实例，该用户还需要从属于具体的Hive实例组，如hive3。
GraphBase	人机	加入**graphbaseadmin**或**graphbasedeveloper**或**graphbaseoperator**组。
Flink	人机	加入**developgroup**和**hadoop**组。设置其“主组”为**developgroup**。 **说明** 若用户需要对接Kafka，则需创建具有Flink和Kafka组件的混合集群，或者为拥有Flink组件的集群和拥有Kafka组件的集群配置跨集群互信，并将创建的Flink用户加入“kafkaadmin”用户组。

步骤6 如果服务启用了Ranger鉴权，创建完成用户后，除了系统默认用户组及默认角色的权限外，需通过Ranger WebUI为用户或者用户所关联的角色/用户组进行赋权，具体操作可参考**配置组件权限策略**相关访问权限策略设置章节。

步骤7 在FusionInsight Manager界面选择“系统 > 权限 > 用户”，在用户名中选择**developuser**，单击操作“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到用户的“user.keytab”文件与“krb5.conf”文件。Keytab文件用于在样例工程中进行安全认证，具体使用请参考各服务的开发指南指导。

说明

如果用户类型是人机，需要先修改初始密码后再下载认证凭据文件，否则在使用时会提示“Password has expired - change password to reset”，导致安全认证失败。

----结束

6.3 MRS 应用开发认证失败常见问题

现象描述

MRS样例工程调试运行过程发现认证失败。

处理流程

出现认证失败的原因很多，在不同场景中建议参考以下步骤来排查：

步骤1 确认本应用所运行设备和集群网络上是否通畅，Kerberos认证所需的各类端口（TCP/UDP）是否可正常访问。

步骤2 确认各个配置文件是否被正确读取到，路径是否保存正确。

步骤3 确认用户名和keytab文件是按操作指导得到的。

步骤4 确认各类配置信息是否已经先设置好了，再发起认证。

步骤5 确认没有在同一个进程中发起多次认证，即重复调用login()方法。

步骤6 若还有问题，需联系技术支持人员做进一步分析。

----结束

认证失败样例

解决认证出现如下关键字：clock skew too great的问题

步骤1 检查集群时间。

步骤2 检查开发环境所在机器的时间，与集群时间的偏差应小于5分钟。

----结束

解决认证出现如下关键字：(Receive time out) can not connect to kdc server的问题

步骤1 要检查“krb5.conf”文件内容是否正确，即是否与集群中的KerberoServer的业务IP配置相同。

步骤2 检查Kerberos服务是否正常。

步骤3 检查防火墙是否关闭。

----结束

解决客户端应用提交任务到hadoop集群报错，提示Failed to find any Kerberos tgt或者No valid credentials provided的问题

步骤1 检查是否执行了kinit，若未执行，则先执行kinit认证操作，再提交任务。

步骤2 多线程场景下，需要在进程的开始处调用hadoop提供的loginfromkeytab函数登录KDC，得到TGT，后续提交任务之前，调用reloginFromKeytab函数刷新该TGT。

```
//进程入口首次登录，登录成功设置userGroupInformation
UserGroupInformation.loginUserFromKeytab(this.userPrincipal,this.keytabFile);
//线程提交任务之前：
UserGroupInformation.getLoginUser().reloginFromKeytab();
```

步骤3 多个脚本同时使用kinit命令认证同一个用户的场景下，需要在各个脚本中执行kinit命令之前，先执行export KRB5CCNAME=keytab_path命令，确保每个脚本进程中KRB5CCNAME指向的路径不一致。

----结束

7 ClickHouse 开发指南（安全模式）

7.1 ClickHouse 应用开发简介

7.1.1 ClickHouse 简介

ClickHouse 简介

ClickHouse是面向联机分析处理的列式数据库，支持SQL查询，且查询性能好，特别是基于大宽表的聚合分析查询性能非常优异，比其他分析型数据库速度快一个数量级。

ClickHouse的设计优点：

- 数据压缩比高
- 多核并行计算
- 向量化计算引擎
- 支持嵌套数据结构
- 支持稀疏索引
- 支持数据Insert和Update

ClickHouse的应用场景：

- 实时数仓场景

  使用流式计算引擎（如Flink）把实时数据写入ClickHouse，借助ClickHouse的优异查询性能，在亚秒级内响应多维度、多模式的实时查询分析请求。

- 离线查询场景

  把规模庞大的业务数据导入到ClickHouse，构造数亿至数百亿记录规模、数百以上的维度的大宽表，随时进行个性化统计和持续探索式查询分析，辅助商业决策，具有非常好的查询体验。

ClickHouse 开发接口简介

ClickHouse由C++语言开发，定位为DBMS，支持HTTP和Native TCP两种网络接口协议，支持JDBC、ODBC等多种驱动方式，推荐使用社区版本的**clickhouse-jdbc**来进行应用程序开发。

7.1.2 ClickHouse 应用开发常用概念

基本概念

- **cluster**

  cluster（集群）在ClickHouse里是一种逻辑的概念，它可以由用户根据需要自由的定义，与通常理解的集群有一定的差异。多个ClickHouse节点之间是一种松耦合的关系，各自独立存在。

- **shards**

  shard（分片）是对cluster的横向切分，1个cluster可以由多个shard组成。

- **replicas**

  replica（副本），1个shard可以有多个replica组成。

- **partition**

  partition（分区），针对的是本地replica而言的，可以理解为是一种纵向切分。

- **MergeTree**

  ClickHouse拥有非常庞大的表引擎体系，MergeTree作为家族系统最基础的表引擎，提供了数据分区、一级索引和二级索引等功能。在创建表的时候需要指定表引擎，不同的表引擎会决定一张数据表的最终“性格”，比如数据表拥有何种特性、数据以何种形式被存储以及如何被加载。

7.1.3 ClickHouse 应用开发流程介绍

开发流程中各阶段的说明如**ClickHouse应用程序开发流程**和**表7-1**所示。

图 7-1 ClickHouse 应用程序开发流程

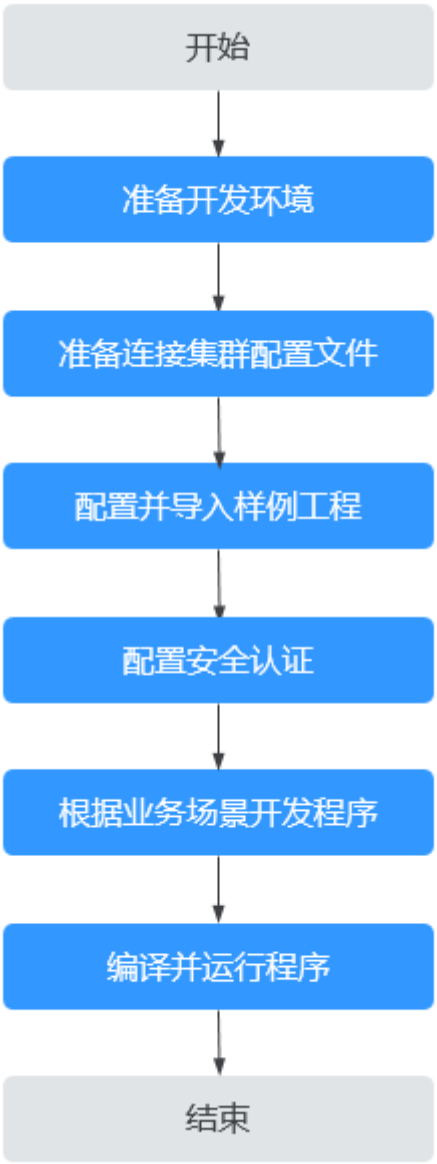


表 7-1 ClickHouse 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，ClickHouse的应用程序支持多种语言开发，推荐使用Java语言，使用IntelliJ IDEA工具，同时完成JDK、Maven等初始配置。	准备ClickHouse应用开发环境

阶段	说明	参考文档
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	准备ClickHouse应用运行环境
配置并导入样例工程	ClickHouse提供了不同场景下的样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。	导入并配置ClickHouse样例工程
根据业务场景开发程序	提供样例工程，帮助用户快速了解ClickHouse各部件的编程接口。	开发ClickHouse应用
编译并运行程序	将开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。	在本地Windows环境中调测ClickHouse应用 在Linux环境中调测ClickHouse应用

7.1.4 ClickHouse 样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下ClickHouse相关样例工程：

表 7-2 ClickHouse 相关样例工程

样例工程位置	描述
clickhouse-examples	指导用户基于Java语言，实现MRS集群中的ClickHouse的数据表创建、删除以及数据的插入、查询等操作。 本工程中包含了建立服务端连接、创建数据库、创建数据表、插入数据、查询数据及删除数据表等操作示例。

7.2 准备 ClickHouse 应用开发环境

7.2.1 准备 ClickHouse 应用开发环境

在进行应用开发时，要准备的开发和运行环境如表7-3所示。

表 7-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境：Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。 华为提供开源镜像站，各服务样例工程依赖的Jar包通过华为开源镜像站下载，剩余所依赖的开源Jar包请直接从Maven中央库或者其他用户自定义的仓库地址下载，详情请参考配置华为开源镜像仓。

准备项	说明
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

7.2.2 准备 ClickHouse 应用运行环境

准备集群认证用户信息

对于开启Kerberos认证的MRS集群，需提前准备具有相关组件操作权限的用户用于程序认证。

以下ClickHouse权限配置示例供参考，在实际业务场景中可根据业务需求灵活调整。

步骤1 登录FusionInsight Manager。

步骤2 在FusionInsight Manager界面选择“系统 > 权限 > 角色 > 添加角色”。

1. 填写角色的名称，例如**developrole**，单击“确定”保存角色。
2. 在“配置资源权限”的表格中选择“*待操作集群的名称* > ClickHouse > Clickhouse Scope”，勾选对应数据库的创建权限。单击对应的数据库名称，根据不同任务场景，勾选对应表的“读”、“写”权限，单击“确定”保存。

步骤3 单击“系统 > 权限 > 用户组 > 添加用户组”，创建一个用户组，例如**developgroup**。

步骤4 单击“系统 > 权限 > 用户 > 添加用户”，创建一个人机用户，例如**developuser**。

1. “用户组”需加入“**developgroup**”和“**supergroup**”组，设置其“主组”为“**supergroup**”。
2. “角色”加入**developrole**。

步骤5 使用新建的**developuser**用户登录FusionInsight Manager，根据界面提示修改初始密码。

步骤6 使用**admin**用户登录FusionInsight Manager，选择“系统 > 权限 > 用户”，在用户名为**developuser**的操作列选择“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到该用户的“user.keytab”文件与“krb5.conf”文件。

----结束

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。

1. **登录FusionInsight Manager页面**，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。

2. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境程序所需配置文件。

1. 在节点中安装客户端。例如客户端安装目录为“/opt/client”。
2. 检查客户端节点网络连接。

   在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

7.2.3 导入并配置 ClickHouse 样例工程

背景信息

获取ClickHouse开发样例工程，将工程导入到IntelliJ IDEA开始样例学习。

前提条件

- 确保本地环境的时间与MRS集群的时间差要小于5分钟，若无法确定，请联系系统管理员。集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备ClickHouse应用运行环境**。

操作场景

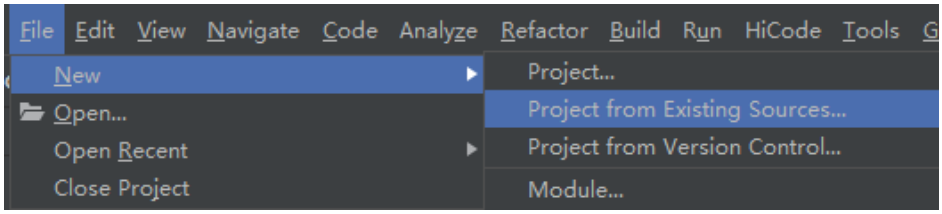
ClickHouse针对多个场景提供样例工程，帮助客户快速学习ClickHouse工程。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程，可根据实际业务场景选择对应的样例，相关样例介绍请参见**ClickHouse样例工程介绍**。

步骤2 在应用开发环境中，导入样例工程到IntelliJ IDEA开发环境。

1. 在IDEA界面选择“File > New > Project from Existing Sources”。



2. 在显示的“Select File or Directory to Import”对话框中，选择“clickhouse-examples”文件夹中的“pom.xml”文件，单击“OK”。
3. 确认后续配置，单击“Next”，如无特殊需求，相关配置使用默认值即可。
4. 选择推荐的JDK版本，单击“Finish”完成样例工程导入。

步骤3 工程导入完成后，修改样例工程的“conf”目录下的“clickhouse-example.properties”文件，根据实际环境信息修改相关参数。

MRS 3.1.5之前版本：

```
loadBalancerIPList=
sslUsed=false
loadBalancerHttpPort=21425
loadBalancerHttpsPort=21426
CLICKHOUSE_SECURITY_ENABLED=true
user=
#密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全
password=
clusterName=default_cluster
databaseName=testdb
tableName=testtb
batchRows=10000
batchNum=10
```

MRS 3.1.5及之后版本：

```
loadBalancerIPList=
sslUsed=false
loadBalancerHttpPort=21425
loadBalancerHttpsPort=21426
CLICKHOUSE_SECURITY_ENABLED=true
user=
#密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全
password=
isMachineUser=false
isSupportMachineUser=true
clusterName=default_cluster
databaseName=testdb
tableName=testtb
batchRows=10000
batchNum=10
clickhouse_dataSource_ip_list=
native_dataSource_ip_list=
```

表 7-4 配置说明表

配置名称	默认值	含义
loadBalancerIPList	-	必填参数，配置为LoadBalance的IP列表。 登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，查看所有ClickHouseBalancer实例对应的业务IP地址。 多个IP地址使用逗号分割，例如配置为“10.10.10.100,10.10.10.101”。
sslUsed	false	是否启用ssl加密，安全模式集群建议配置为“true”。

配置名称	默认值	含义
loadBalancerHttpPort	21425	LoadBalance的HTTP端口。若sslUsed配置为false，则此参数不允许为空。 MRS 3.2.0之前版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，单击对应的ClickHouseBalancer实例，选择“实例配置 > 全部配置”，搜索“lb_http_port”并获取其参数值，默认为21425。 MRS 3.2.0及之后版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的HTTP Balancer端口号中的“非加密端口”。
loadBalancerHttpsPort	21426	LoadBalance的HTTPS端口。若sslUsed配置为true，则此参数不允许为空。 MRS 3.2.0之前版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，单击对应的ClickHouseBalancer实例，选择“实例配置 > 全部配置”，搜索“lb_https_port”并获取其参数值，默认为21426。 MRS 3.2.0及之后版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的HTTP Balancer端口号中的“加密端口”。
CLICKHOUSE_SECURITY_ENABLED	true	ClickHouse安全模式开关，安全模式集群时该参数固定为true。
user	无默认值	**准备集群认证用户信息**中已准备好的开发用户的用户名。
password	无默认值	开发用户对应的密码。 密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。 **说明** 如果是在Manager上新建的用户，首次使用前需要修改初始密码。
isMachineUser	false	使用机机用户认证时，参数值修改为true。 如果配置为true，则user和password参数为机机用户的用户名和密码。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。

配置名称	默认值	含义
isSupportMachineUser	true	是否支持机机用户认证的功能。 true：支持机机用户认证的功能。 false：不支持机机用户认证的功能。 该参数配置请通过以下操作确认： 登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 配置 > 全部配置”，在“ClickHouseServer（角色）”下选择“安全”，查看“SUPPORT_MACHINE_USER”参数配置。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。
clusterName	default_cluster	ClickHouse逻辑集群名称，保持默认值。
databaseName	testdb	样例代码工程中需要创建的数据库名称，可以根据实际情况修改。
tableName	testtb	样例代码工程中需要创建的表名称，可以根据实际情况修改。
batchRows	10000	一个批次写入数据的条数。
batchNum	10	写入数据的总批次。
clickhouse_dataSource_ip_list	-	必填参数，配置为ClickHouseBalancer实例的IP列表和http端口。 多个IP地址使用逗号分割，具体格式为：ip:port,ip:port,ip:port IP和端口获取： IP：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，查看所有ClickHouseBalancer实例对应的业务IP地址。 端口：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的HTTP Balancer端口号中的“加密端口”。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。

配置名称	默认值	含义
native_dataSource_ip_list	-	必填参数，配置为ClickHouseBalancer实例的IP列表和tcp端口。 多个IP地址使用逗号分割，具体格式为：ip:port,ip:port,ip:port IP和端口获取： IP：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，查看所有ClickHouseBalancer实例对应的业务IP地址。 端口：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的TCP Balancer端口号中的“非加密端口”。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。

说明

ClickHouse提供了基于Loadbalance部署架构，可以将用户访问流量自动分发到多台后端节点，扩展系统对外的服务能力，实现更高水平的应用容错。客户端应用请求集群时，使用基于Nginx的ClickHouseBalancer控制节点来进行流量分发，无论集群写入的负载、读的负载以及应用接入的高可用性都具备了有力的保障。

----结束

7.3 开发 ClickHouse 应用

7.3.1 ClickHouse 应用程序开发思路

通过典型场景，用户可以快速学习和掌握ClickHouse的开发过程，并且对关键的接口函数有所了解。

场景说明

ClickHouse可以使用SQL进行常见的业务操作，代码样例中所涉及的SQL操作主要包括创建数据库、创建表、插入表数据、查询表数据以及删除表操作。

本代码样例讲解顺序为：

1. 设置属性
2. 建立连接
3. 创建库
4. 创建表
5. 插入数据
6. 查询数据
7. 删除表

开发思路

ClickHouse作为一款独立的DBMS系统，使用SQL语言就可以进行常见的操作。开发程序示例中，全部通过clickhouse-jdbc API接口来进行描述，开发流程主要分为以下几部分：

- 设置属性：设置连接ClickHouse服务实例的参数属性。
- 建立连接：建立和ClickHouse服务实例的连接。
- 创建库：创建ClickHouse数据库。
- 创建表：创建ClickHouse数据库下的表。
- 插入数据：插入数据到ClickHouse表中。
- 查询数据：查询ClickHouse表数据。
- 删除表：删除已创建的ClickHouse表。

7.3.2 配置 ClickHouse 连接属性

在ClickhouseJDBCHaDemo、Demo、NativeJDBCHaDemo和Util文件创建connection的样例中设置连接属性，如下样例代码设置socket超时时间为60s。

```
ClickHouseProperties clickHouseProperties = new ClickHouseProperties();
clickHouseProperties.setSocketTimeout(60000);
```

如果**导入并配置ClickHouse样例工程**中的“clickhouse-example.properties”配置文件中“sslUsed”参数配置为“true”时，则需要在ClickhouseJDBCHaDemo、Demo、NativeJDBCHaDemo和Util文件创建connection的样例中设置如下连接属性：

```
clickHouseProperties.setSsl(true);
clickHouseProperties.setSslMode("none");
```

7.3.3 建立 ClickHouse 连接

以下代码片段在“ClickhouseJDBCHaDemo”类的initConnection方法中。在创建连接时传入**表7-4**中配置的user和password作为认证凭据，ClickHouse会带着用户名和密码在服务端进行安全认证。

```
clickHouseProperties.setPassword(userPass);
clickHouseProperties.setUser(userName);
BalancedClickhouseDataSource balancedClickhouseDataSource = new
BalancedClickhouseDataSource(JDBC_PREFIX + UriList, clickHouseProperties);
```

7.3.4 创建 ClickHouse 数据库

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的createDatabase方法中。通过on cluster语句在集群中创建**表7-4**中以databaseName参数值为数据库名的数据库。

```
private void createDatabase(String databaseName, String clusterName) throws Exception  {
    String createDbSql = "create database if not exists " + databaseName + " on cluster " + clusterName;
    util.exeSql(createDbSql);
}
```

7.3.5 创建 ClickHouse 表

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的createTable方法中。通过on cluster语句在集群中创建**表7-4**中tableName参数值为表名的ReplicatedMerge表和Distributed表。

```
private void createTable(String databaseName, String tableName, String clusterName) throws Exception {
  String createSql = "create table " + databaseName + "." + tableName + " on cluster " + clusterName  +
```

```
" (name String, age UInt8, date Date)engine=ReplicatedMergeTree('/clickhouse/tables/{shard}/" +
databaseName + "." + tableName + "'," + "'{replica}') partition by toYYYYMM(date) order by age";
   String createDisSql = "create table " + databaseName + "." + tableName + "_all" + " on cluster " +
clusterName + " as " + databaseName + "." + tableName + " ENGINE = Distributed(default_cluster," +
databaseName + "," + tableName + ", rand());";    ArrayList<String> sqlList = new ArrayList<String>();
   sqlList.add(createSql);
   sqlList.add(createDisSql);
   util.exeSql(sqlList);
}
```

7.3.6 插入 ClickHouse 数据

创建ClickHouse表创建的表具有三个字段，分别是String、UInt8和Date类型。

```
String insertSql = "insert into " + databaseName + "." + tableName + " values (?,?,?)";
PreparedStatement preparedStatement = connection.prepareStatement(insertSql);
long allBatchBegin = System.currentTimeMillis();
for (int j = 0; j < batchNum; j++) {
    for (int i = 0; i < batchRows; i++) {
       preparedStatement.setString(1, "huawei_" + (i + j * 10));
       preparedStatement.setInt(2, ((int) (Math.random() * 100)));
       preparedStatement.setDate(3, generateRandomDate("2018-01-01", "2021-12-31"));
       preparedStatement.addBatch();
    }
   long begin = System.currentTimeMillis();
   preparedStatement.executeBatch();
   long end = System.currentTimeMillis();
   log.info("Inert batch time is {} ms", end - begin);
}
long allBatchEnd = System.currentTimeMillis();
log.info("Inert all batch time is {} ms", allBatchEnd - allBatchBegin);
```

7.3.7 查询 ClickHouse 数据

查询语句1：**querySql1**查询**创建ClickHouse表**创建的tableName表中任意10条数据；查询语句2：**querySql2**通过内置函数对**创建ClickHouse表**创建的tableName表中的日期字段取年月后进行聚合。

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的queryData方法中。

```
private void queryData(String databaseName, String tableName) throws Exception {
   String querySql1 = "select * from " + databaseName + "." + tableName + "_all" + " order by age limit
10";
   String querySql2 = "select toYYYYMM(date),count(1) from " + databaseName + "." + tableName + "_all"
+ " group by toYYYYMM(date) order by count(1) DESC limit 10";
   ArrayList<String> sqlList = new ArrayList<String>();
   sqlList.add(querySql1);
   sqlList.add(querySql2);
   ArrayList<ArrayList<ArrayList<String>>> result = util.exeSql(sqlList);
   for (ArrayList<ArrayList<String>> singleResult : result) {
      for (ArrayList<String> strings : singleResult) {
         StringBuilder stringBuilder = new StringBuilder();
         for (String string : strings) {
            stringBuilder.append(string).append("\t");
         }
         log.info(stringBuilder.toString());
      }
   }
}
```

7.3.8 删除 ClickHouse 表

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的dropTable方法中，用于删除在**创建ClickHouse表**中创建的副本表和分布式表。

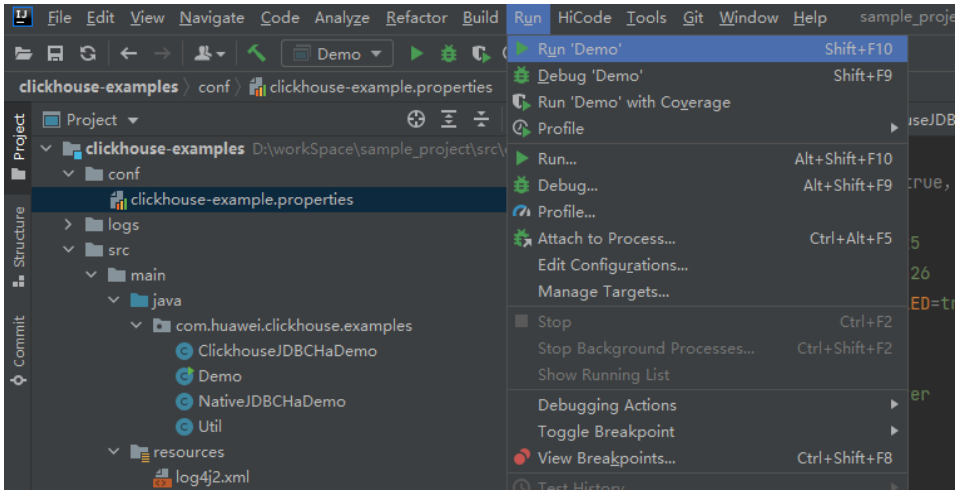
```
private void dropTable(String databaseName, String tableName, String clusterName) throws Exception {
    String dropLocalTableSql = "drop table if exists " + databaseName + "." + tableName + " on cluster " +
clusterName;
    String dropDisTableSql = "drop table if exists " + databaseName + "." + tableName + "_all" + " on cluster
" + clusterName;
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(dropLocalTableSql);
    sqlList.add(dropDisTableSql);
    util.exeSql(sqlList);
}
```

7.4 调测 ClickHouse 应用

7.4.1 在本地 Windows 环境中调测 ClickHouse 应用

编译并运行程序

在程序代码完成开发后，您可以在Windows环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。在开发环境IntelliJ IDEA工程“clickhouse-examples”中单击“Run 'Demo'”运行应用程序工程。



查看调测结果

ClickHouse应用程序运行完成后，可通过以下方式查看程序运行情况：

- 通过运行结果查看程序运行情况。
- 通过ClickHouse日志获取应用运行情况，即“logs”目录下的日志文件：clickhouse-example.log。

运行clickhouse-examples的完整样例后，控制台显示部分运行结果如下：

```
Connected to the target VM, address: '127.0.0.1:53321', transport: 'socket'
2021-04-21 21:02:16,900 | INFO  | main | loadBalancerIPList is 10.120.147.36, loadBalancerHttpPort is
21425, user is xxx, clusterName is default_cluster, isSec is true, password is xxx. |
com.huawei.clickhouse.examples.Demo.main(Demo.java:40)
2021-04-21 21:02:16,904 | INFO  | main | ckLbServerList current member is 0, ClickhouseBalancer is
10.120.147.36:21425 | com.huawei.clickhouse.examples.Demo.getCkLbServerList(Demo.java:96)
2021-04-21 21:02:16,914 | INFO  | main | Driver registered |
ru.yandex.clickhouse.ClickHouseDriver.<clinit>(ClickHouseDriver.java:49)
2021-04-21 21:02:16,918 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:17,517 | INFO  | main | Execute query:drop table if exists testdb.testtb on cluster
```

```
default_cluster | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:17,656 | INFO  | main | Execute time is 139 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:17,657 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:17,701 | INFO  | main | Execute query:drop table if exists testdb.testtb_all on cluster
default_cluster | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:17,851 | INFO  | main | Execute time is 148 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:17,851 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:17,895 | INFO  | main | Execute query:create database if not exists testdb on cluster
default_cluster | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:18,043 | INFO  | main | Execute time is 148 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:18,044 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:18,088 | INFO  | main | Execute query:create table testdb.testtb on cluster default_cluster
(name String, age UInt8, date Date)engine=ReplicatedMergeTree('/clickhouse/tables/{shard}/
testdb.testtb','{replica}') partition by toYYYYMM(date) order by age |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:18,233 | INFO  | main | Execute time is 144 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:18,233 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:18,278 | INFO  | main | Execute query:create table testdb.testtb_all on cluster
default_cluster as testdb.testtb ENGINE = Distributed(default_cluster,testdb,testtb, rand()); |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:18,422 | INFO  | main | Execute time is 144 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:18,423 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.insertData(Util.java:128)
2021-04-21 21:02:19,380 | INFO  | main | Inert batch time is 720 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:19,927 | INFO  | main | Inert batch time is 492 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:20,456 | INFO  | main | Inert batch time is 504 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:20,894 | INFO  | main | Inert batch time is 410 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:21,348 | INFO  | main | Inert batch time is 431 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:21,813 | INFO  | main | Inert batch time is 442 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:22,273 | INFO  | main | Inert batch time is 434 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:22,730 | INFO  | main | Inert batch time is 435 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:23,212 | INFO  | main | Inert batch time is 459 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:23,689 | INFO  | main | Inert batch time is 452 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:23,689 | INFO  | main | Inert all batch time is 5216 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:148)
2021-04-21 21:02:23,689 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:23,732 | INFO  | main | Execute query:select * from testdb.testtb_all order by age limit 10 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:23,803 | INFO  | main | Execute time is 71 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:23,804 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:23,848 | INFO  | main | Execute query:select toYYYYMM(date),count(1) from
testdb.testtb_all group by toYYYYMM(date) order by count(1) DESC limit 10 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:23,895 | INFO  | main | Execute time is 47 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:23,896 | INFO  | main | name    age    date     |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
```

```
2021-04-21 21:02:23,896 | INFO  | main | huawei_4077    0    2021-12-21    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_183    0    2021-12-10    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_5407    0    2021-12-13    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_1072    0    2021-12-03    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_4667    0    2021-12-22    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_1767    0    2021-12-03    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_8001    0    2021-12-22    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_1822    0    2021-12-04    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_5095    0    2021-12-23    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_7133    0    2021-12-26    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | toYYYYMM(date)    count()    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202101    2184    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202105    2176    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201810    2173    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201907    2162    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201803    2159    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201805    2153    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202110    2145    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201801    2144    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201908    2143    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202005    2133    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
Disconnected from the target VM, address: '127.0.0.1:53321', transport: 'socket'
Process finished with exit code 0
```

7.4.2 在 Linux 环境中调测 ClickHouse 应用

ClickHouse应用程序也支持在Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。

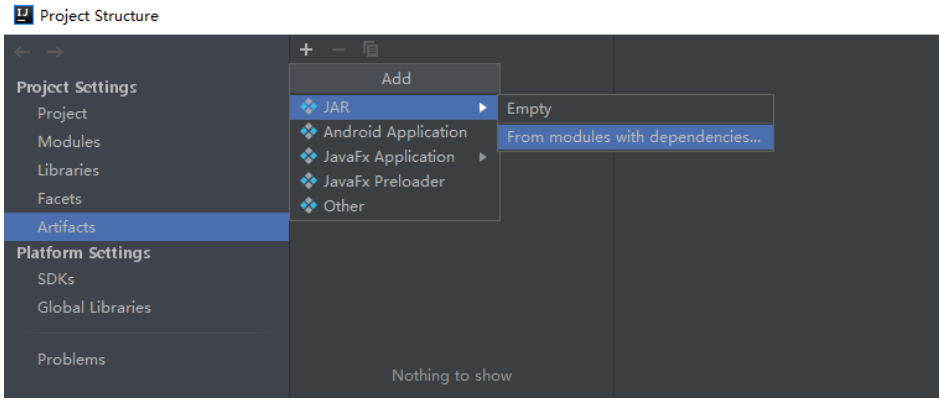
前提条件

Linux环境已安装JDK，版本号需要和IntelliJ IDEA导出Jar包使用的JDK版本一致，并设置好Java环境变量。

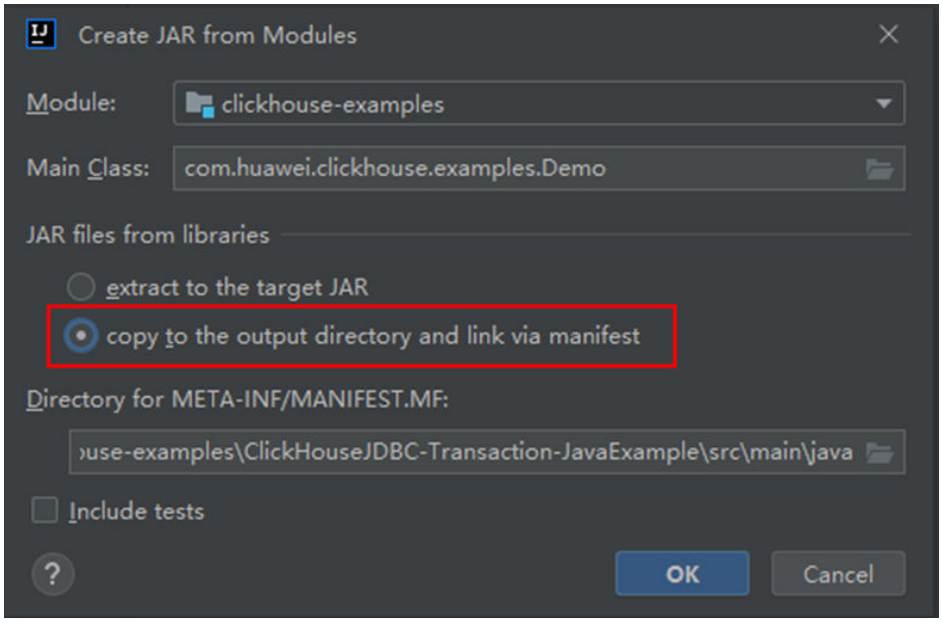
编译并运行程序

步骤1 导出jar包。

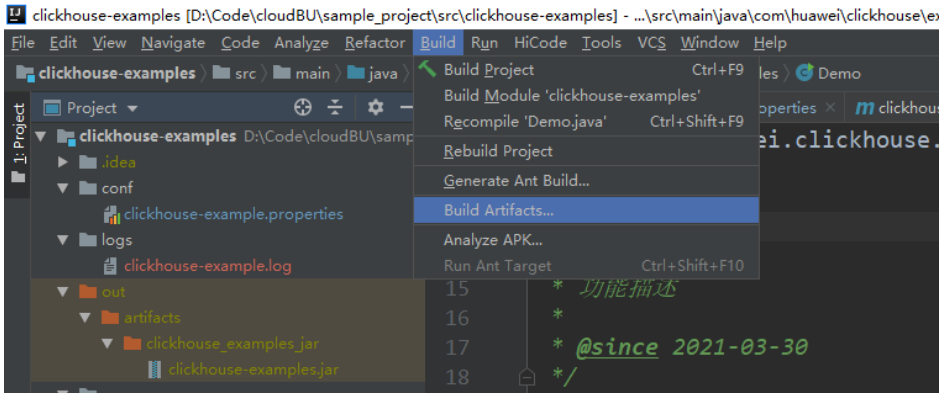
1. 进入IntelliJ IDEA，选择“File > Project Structure > Artifacts”。
2. 单击“加号”，选择“JAR > from modules with dependencies”。



3. “Main Class”选择“com.huawei.clickhouse.examples.Demo”，单击“OK”。



4. 选择“Build> Build Artifacts”，编译成功后在“clickhouse-examples\out\artifacts\clickhouse_examples_jar”目录下查看并获取当前目录的所有jar文件。



步骤2 将“clickhouse-examples\out\artifacts\clickhouse_examples.jar”目录下的所有jar文件和“clickhouse-examples”目录下的“conf”文件夹复制到ClickHouse客户端安装目录下，例如“*客户端安装目录*/JDBC”目录下。

步骤3 登录客户端节点，进入jar文件上传目录下，修改文件权限为700。

cd /opt/client

chmod 700 clickhouse-examples.jar

步骤4 在“clickhouse_examples.jar”所在客户端目录下执行如下命令运行jar包：

source *客户端安装目录*/**bigdata_env**

java -cp ./*:conf/clickhouse-example.properties com.huawei.clickhouse.examples.Demo

----结束

查看调测结果

ClickHouse应用程序运行完成后，可通过以下方式查看程序运行情况：

- 通过运行结果查看程序运行情况。
- 通过ClickHouse日志获取应用运行情况。

  即查看当前jar文件所在目录的“logs/clickhouse-example.log”日志文件，例如“*客户端安装目录*/**JDBC/logs/clickhouse-example.log**”。

jar包运行结果如下：

```
2021-06-10 20:53:56,028 | INFO  | main | Current load balancer is 10.112.17.150:21426 |
com.huawei.clickhouse.examples.Util.insertData(Util.java:128)
2021-06-10 20:53:58,247 | INFO  | main | Inert batch time is 1442 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:53:59,649 | INFO  | main | Inert batch time is 1313 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:05,872 | INFO  | main | Inert batch time is 6132 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:10,223 | INFO  | main | Inert batch time is 4272 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:11,614 | INFO  | main | Inert batch time is 1300 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:12,871 | INFO  | main | Inert batch time is 1200 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:14,589 | INFO  | main | Inert batch time is 1663 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:16,141 | INFO  | main | Inert batch time is 1500 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:17,690 | INFO  | main | Inert batch time is 1498 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:19,206 | INFO  | main | Inert batch time is 1468 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:19,207 | INFO  | main | Inert all batch time is 22626 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:148)
2021-06-10 20:54:19,208 | INFO  | main | Current load balancer is 10.112.17.150:21426 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-06-10 20:54:20,231 | INFO  | main | Execute query:select * from mutong1.testtb_all order by age limit
10 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-06-10 20:54:21,266 | INFO  | main | Execute time is 1035 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-06-10 20:54:21,267 | INFO  | main | Current load balancer is 10.112.17.150:21426 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-06-10 20:54:21,815 | INFO  | main | Execute query:select toYYYYMM(date),count(1) from
mutong1.testtb_all group by toYYYYMM(date) order by count(1) DESC limit 10 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-06-10 20:54:22,897 | INFO  | main | Execute time is 1082 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-06-10 20:54:22,898 | INFO  | main | name    age    date     |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,898 | INFO  | main | huawei_266    0    2021-12-19     |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
```



```
2021-06-10 20:54:22,899 | INFO  | main | huawei_2500    0    2021-12-29    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_8980    0    2021-12-16    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_671    0    2021-12-29    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_2225    0    2021-12-12    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_6040    0    2021-12-14    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_7294    0    2021-12-10    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_1133    0    2021-12-25    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | huawei_3161    0    2021-12-21    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | huawei_3992    0    2021-11-25    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | toYYYYMM(date)    count()    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 201910    2247    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 202105    2213    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 201801    2208    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 201803    2204    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201810    2167    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201805    2166    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201901    2164    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201908    2145    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201912    2143    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 202107    2137    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
```

8 ClickHouse 开发指南（普通模式）

8.1 ClickHouse 应用开发简介

8.1.1 ClickHouse 简介

ClickHouse 简介

ClickHouse是面向联机分析处理的列式数据库，支持SQL查询，且查询性能好，特别是基于大宽表的聚合分析查询性能非常优异，比其他分析型数据库速度快一个数量级。

ClickHouse的设计优点：

- 数据压缩比高
- 多核并行计算
- 向量化计算引擎
- 支持嵌套数据结构
- 支持稀疏索引
- 支持数据Insert和Update

ClickHouse的应用场景：

- 实时数仓场景

  使用流式计算引擎（如Flink）把实时数据写入ClickHouse，借助ClickHouse的优异查询性能，在亚秒级内响应多维度、多模式的实时查询分析请求。

- 离线查询场景

  把规模庞大的业务数据导入到ClickHouse，构造数亿至数百亿记录规模、数百以上的维度的大宽表，随时进行个性化统计和持续探索式查询分析，辅助商业决策，具有非常好的查询体验。

ClickHouse 开发接口简介

ClickHouse由C++语言开发，定位为DBMS，支持HTTP和Native TCP两种网络接口协议，支持JDBC、ODBC等多种驱动方式，推荐使用社区版本的**clickhouse-jdbc**来进行应用程序开发。

8.1.2 ClickHouse 应用开发常用概念

基本概念

- **cluster**

  cluster（集群）在ClickHouse里是一种逻辑的概念，它可以由用户根据需要自由的定义，与通常理解的集群有一定的差异。多个ClickHouse节点之间是一种松耦合的关系，各自独立存在。

- **shards**

  shard（分片）是对cluster的横向切分，1个cluster可以由多个shard组成。

- **replicas**

  replica（副本），1个shard可以有多个replica组成。

- **partition**

  partition（分区），针对的是本地replica而言的，可以理解为是一种纵向切分。

- **MergeTree**

  ClickHouse拥有非常庞大的表引擎体系，MergeTree作为家族系统最基础的表引擎，提供了数据分区、一级索引和二级索引等功能。在创建表的时候需要指定表引擎，不同的表引擎会决定一张数据表的最终“性格”，比如数据表拥有何种特性、数据以何种形式被存储以及如何被加载。

8.1.3 ClickHouse 应用开发流程介绍

开发流程中各阶段的说明如**图8-1**和**表8-1**所示。

图 8-1 ClickHouse 应用程序开发流程

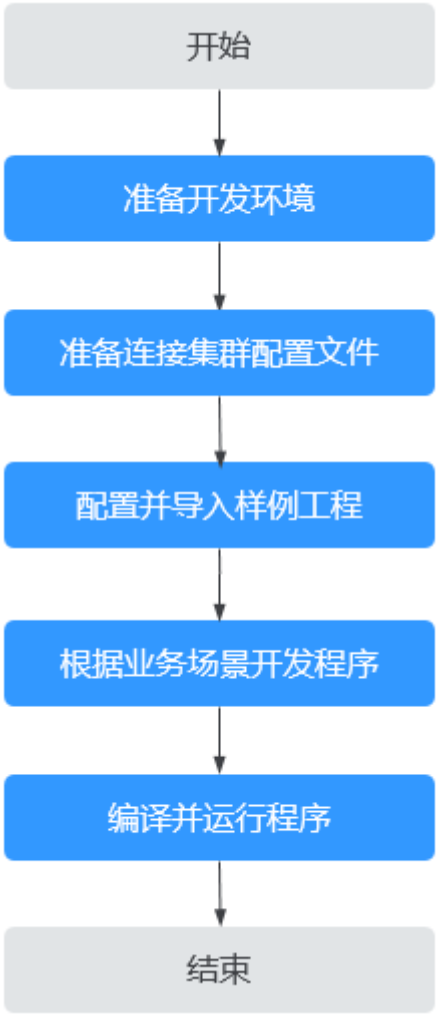


表 8-1 ClickHouse 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，ClickHouse的应用程序支持多种语言开发，推荐使用Java语言，使用IntelliJ IDEA工具，同时完成JDK、Maven等初始配置。	**准备ClickHouse应用开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群信息文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备ClickHouse应用运行环境**

阶段	说明	参考文档
配置并导入样例工程	ClickHouse提供了不同场景下的样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。	导入并配置ClickHouse样例工程
根据业务场景开发程序	提供样例工程，帮助用户快速了解ClickHouse各部件的编程接口。	开发ClickHouse应用
编译并运行程序	将开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。	在本地Windows环境中调测ClickHouse应用 在Linux环境中调测ClickHouse应用

8.1.4 ClickHouse 样例工程介绍

MRS样例工程获取地址为https://github.com/huaweicloud/huaweicloud-mrs-example，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下ClickHouse相关样例工程：

表 8-2 ClickHouse 相关样例工程

样例工程位置	描述
clickhouse-examples	指导用户基于Java语言，实现MRS集群中的ClickHouse的数据表创建、删除以及数据的插入、查询等操作。 本工程中包含了建立服务端连接、创建数据库、创建数据表、插入数据、查询数据及删除数据表等操作示例。

8.2 准备 ClickHouse 应用开发环境

8.2.1 准备 ClickHouse 应用开发环境

在进行应用开发时，要准备的开发和运行环境如表8-3所示。

表 8-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境：Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

8.2.2 准备 ClickHouse 应用运行环境

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。

1. **登录FusionInsight Manager页面**，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

   例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。
2. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

   说明

   - 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
   - Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境程序所需配置文件。

1. 在节点中安装客户端。例如客户端安装目录为“/opt/client”。
2. 检查客户端节点网络连接。

   在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

8.2.3 导入并配置 ClickHouse 样例工程

背景信息

获取ClickHouse开发样例工程，将工程导入到IntelliJ IDEA开始样例学习。

前提条件

- 确保本地环境的时间与MRS集群的时间差要小于5分钟，若无法确定，请联系系统管理员。集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备ClickHouse应用运行环境**。

操作场景

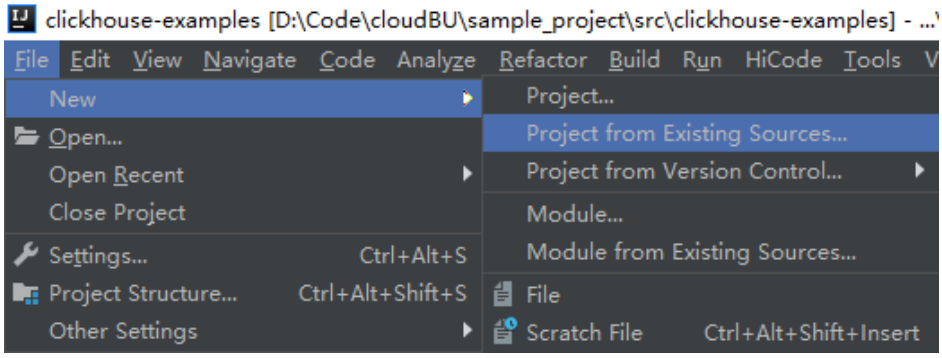
ClickHouse针对多个场景提供样例工程，帮助客户快速学习ClickHouse工程。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程，可根据实际业务场景选择对应的样例，相关样例介绍请参见**ClickHouse样例工程介绍**。

步骤2 在应用开发环境中，导入样例工程到IntelliJ IDEA开发环境。

1. 在IDEA界面选择“File > New > Project from Existing Sources”。



2. 在显示的“Select File or Directory to Import”对话框中，选择“clickhouse-examples”文件夹中的“pom.xml”文件，单击“OK”。
3. 确认后续配置，单击“Next”；如无特殊需求，使用默认值即可。
4. 选择推荐的JDK版本，单击“Finish”完成导入。

步骤3 工程导入完成后，修改样例工程的“conf”目录下的“clickhouse-example.properties”文件，根据实际环境信息修改相关参数。

MRS 3.1.5之前版本：

```
loadBalancerIPList=
sslUsed=false
loadBalancerHttpPort=21425
loadBalancerHttpsPort=21426
CLICKHOUSE_SECURITY_ENABLED=true
user=
#密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全
password=
clusterName=default_cluster
databaseName=testdb
tableName=testtb
batchRows=10000
batchNum=10
```

MRS 3.1.5及之后版本：

```
loadBalancerIPList=
sslUsed=false
loadBalancerHttpPort=21425
loadBalancerHttpsPort=21426
CLICKHOUSE_SECURITY_ENABLED=true
user=
#密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全
password=
isMachineUser=false
isSupportMachineUser=true
clusterName=default_cluster
databaseName=testdb
tableName=testtb
batchRows=10000
batchNum=10
clickhouse_dataSource_ip_list=
native_dataSource_ip_list=
```


表 8-4 配置说明表

配置名称	默认值	含义
loadBalancerIPList	-	必填参数，配置为LoadBalance的IP列表。 登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，查看所有ClickHouseBalancer实例对应的业务IP地址。 多个IP地址使用逗号分割，例如配置为“10.10.10.100,10.10.10.101”。
sslUsed	false	是否启用ssl加密，普通模式集群建议配置为“false”。
loadBalancerHttpPort	21425	LoadBalance的HTTP端口。若sslUsed配置为false，则此参数不允许为空。 MRS 3.2.0之前版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，单击对应的ClickHouseBalancer实例，选择“实例配置 > 全部配置”，搜索“lb_http_port”并获取其参数值，默认为21425。 MRS 3.2.0及之后版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的HTTP Balancer端口号中的“非加密端口”。
loadBalancerHttpsPort	21426	LoadBalance的HTTPS端口，若sslUsed配置为true，则此参数不允许为空。 MRS 3.2.0之前版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，单击对应的ClickHouseBalancer实例，选择“实例配置 > 全部配置”，搜索“lb_https_port”并获取其参数值，默认为21426。 MRS 3.2.0及之后版本，登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的HTTP Balancer端口号中的“加密端口”。
CLICKHOUSE_SECURITY_ENABLED	true	ClickHouse安全模式开关，普通模式集群时该参数填写为false。
user	无默认值	表8-3中已准备好的开发用户。
password	无默认值	开发用户对应的密码。 密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全 **说明** 如果是在Manager上新建的用户，首次使用前需要修改初始密码。

配置名称	默认值	含义
isMachineUser	false	使用机机用户认证时，参数值修改为true。 如果配置为true，则user和password参数为机机用户的用户名和密码。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。
isSupportMachineUser	true	是否支持机机用户认证的功能。 true：支持机机用户认证的功能。 false：不支持机机用户认证的功能。 该参数配置请通过以下操作确认： 登录Manager，选择“集群 > 服务 > ClickHouse > 配置 > 全部配置”，在“ClickHouseServer（角色）”下选择“安全”，查看“SUPPORT_MACHINE_USER”参数配置。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。
clusterName	default_cluster	ClickHouse逻辑集群名称，保持默认值。
databaseName	testdb	样例代码工程中需要创建的数据库名称，可以根据实际情况修改。
tableName	testtb	样例代码工程中需要创建的表名称，可以根据实际情况修改。
batchRows	10000	一个批次写入数据的条数。
batchNum	10	写入数据的总批次。
clickhouse_dataSource_ip_list	-	必填参数，配置为ClickHouseBalancer实例的IP列表和http端口。 多个IP地址使用逗号分割，具体格式为：ip:port,ip:port,ip:port IP和端口获取： IP：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，查看所有ClickHouseBalancer实例对应的业务IP地址。 端口：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的HTTP Balancer端口号中的“非加密端口”。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。

配置名称	默认值	含义
native_dataSource_ip_list	-	必填参数，配置为ClickHouseBalancer实例的IP列表和tcp端口。 多个IP地址使用逗号分割，具体格式为：ip:port,ip:port,ip:port IP和端口获取： IP：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 实例”，查看所有ClickHouseBalancer实例对应的业务IP地址。 端口：登录FusionInsight Manager，选择“集群 > 服务 > ClickHouse > 逻辑集群”，查看对应逻辑集群的TCP Balancer端口号中的“非加密端口”。 **说明** 该参数仅适用于MRS 3.1.5及之后版本。

说明

ClickHouse提供了基于Loadbalance部署架构，可以将用户访问流量自动分发到多台后端节点，扩展系统对外的服务能力，实现更高水平的应用容错。客户端应用请求集群时，使用基于Nginx的ClickHouseBalancer控制节点来进行流量分发，无论集群写入的负载、读的负载以及应用接入的高可用性都具备了有力的保障。

----结束

8.3 开发 ClickHouse 应用

8.3.1 ClickHouse 应用程序开发思路

通过典型场景，用户可以快速学习和掌握ClickHouse的开发过程，并且对关键的接口函数有所了解。

场景说明

ClickHouse可以使用SQL进行常见的业务操作，代码样例中所涉及的SQL操作主要包括创建数据库、创建表、插入表数据、查询表数据以及删除表操作。

本代码样例讲解顺序为：

1. 设置属性
2. 建立连接
3. 创建库
4. 创建表
5. 插入数据
6. 查询数据

7. 删除表

开发思路

ClickHouse作为一款独立的DBMS系统，使用SQL语言就可以进行常见的操作。开发程序示例中，全部通过clickhouse-jdbc API接口来进行描述，开发流程主要分为以下几部分：

- 设置属性：设置连接ClickHouse服务实例的参数属性。
- 建立连接：建立和ClickHouse服务实例的连接。
- 创建库：创建ClickHouse数据库。
- 创建表：创建ClickHouse数据库下的表。
- 插入数据：插入数据到ClickHouse表中。
- 查询数据：查询ClickHouse表数据。
- 删除表：删除已创建的ClickHouse表。

8.3.2 配置 ClickHouse 连接属性

在ClickhouseJDBCHaDemo、Demo、NativeJDBCHaDemo和Util文件创建connection的样例中设置连接属性，如下样例代码设置socket超时时间为60s。

```
ClickHouseProperties clickHouseProperties = new ClickHouseProperties();
clickHouseProperties.setSocketTimeout(60000);
```

如果**导入并配置ClickHouse样例工程**中的“clickhouse-example.properties”配置文件中“sslUsed”参数配置为“true”时，则需要在ClickhouseJDBCHaDemo、Demo、NativeJDBCHaDemo和Util文件创建connection的样例中设置如下连接属性：

```
clickHouseProperties.setSsl(true);
clickHouseProperties.setSslMode("none");
```

8.3.3 建立 ClickHouse 连接

以下代码片段在“ClickhouseJDBCHaDemo”类的initConnection方法中。在创建连接时传入**表8-4**中配置的user和password作为认证凭据，ClickHouse会带着用户名和密码在服务端进行安全认证。

```
clickHouseProperties.setPassword(userPass);
clickHouseProperties.setUser(userName);
BalancedClickhouseDataSource balancedClickhouseDataSource = new
BalancedClickhouseDataSource(JDBC_PREFIX + UriList, clickHouseProperties);
```

8.3.4 创建 ClickHouse 库

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的createDatabase方法中。通过on cluster语句在集群中创建**表8-4**中以databaseName参数值为数据库名的数据库。

```
private void createDatabase(String databaseName, String clusterName) throws Exception  {
    String createDbSql = "create database if not exists " + databaseName + " on cluster " + clusterName;
    util.exeSql(createDbSql);
}
```

8.3.5 创建 ClickHouse 表

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的createTable方法中。通过on cluster语句在集群中创建**表8-4**中tableName参数值为表名的ReplicatedMerge表和Distributed表。

```
private void createTable(String databaseName, String tableName, String clusterName) throws Exception {
   String createSql = "create table " + databaseName + "." + tableName + " on cluster " + clusterName  +
" (name String, age UInt8, date Date)engine=ReplicatedMergeTree('/clickhouse/tables/{shard}/" +
databaseName + "." + tableName + "'," + "'{replica}') partition by toYYYYMM(date) order by age";
   String createDisSql = "create table " + databaseName + "." + tableName + "_all" + " on cluster " +
clusterName + " as " + databaseName + "." + tableName + " ENGINE = Distributed(default_cluster," +
databaseName + "," + tableName + ", rand());";    ArrayList<String> sqlList = new ArrayList<String>();
   sqlList.add(createSql);
   sqlList.add(createDisSql);
   util.exeSql(sqlList);
}
```

8.3.6 插入 ClickHouse 数据

创建ClickHouse表创建的表具有三个字段，分别是String、UInt8和Date类型。

```
String insertSql = "insert into " + databaseName + "." + tableName + " values (?,?,?)";
PreparedStatement preparedStatement = connection.prepareStatement(insertSql);
long allBatchBegin = System.currentTimeMillis();
for (int j = 0; j < batchNum; j++) {
    for (int i = 0; i < batchRows; i++) {
       preparedStatement.setString(1, "huawei_" + (i + j * 10));
       preparedStatement.setInt(2, ((int) (Math.random() * 100)));
       preparedStatement.setDate(3, generateRandomDate("2018-01-01", "2021-12-31"));
       preparedStatement.addBatch();
    }
   long begin = System.currentTimeMillis();
   preparedStatement.executeBatch();
   long end = System.currentTimeMillis();
   log.info("Inert batch time is {} ms", end - begin);
}
long allBatchEnd = System.currentTimeMillis();
log.info("Inert all batch time is {} ms", allBatchEnd - allBatchBegin);
```

8.3.7 查询 ClickHouse 数据

查询语句1：**querySql1**查询**创建ClickHouse表**创建的tableName表中任意10条数据；查询语句2：**querySql2**通过内置函数对**创建ClickHouse表**创建的tableName表中的日期字段取年月后进行聚合。

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的queryData方法中。

```
private void queryData(String databaseName, String tableName) throws Exception {
   String querySql1 = "select * from " + databaseName + "." + tableName + "_all" + " order by age limit
10";
   String querySql2 = "select toYYYYMM(date),count(1) from " + databaseName + "." + tableName + "_all"
+ " group by toYYYYMM(date) order by count(1) DESC limit 10";
   ArrayList<String> sqlList = new ArrayList<String>();
   sqlList.add(querySql1);
   sqlList.add(querySql2);
   ArrayList<ArrayList<ArrayList<String>>> result = util.exeSql(sqlList);
   for (ArrayList<ArrayList<String>> singleResult : result) {
      for (ArrayList<String> strings : singleResult) {
         StringBuilder stringBuilder = new StringBuilder();
         for (String string : strings) {
            stringBuilder.append(string).append("\t");
         }
         log.info(stringBuilder.toString());
      }
   }
}
```


8.3.8 删除 ClickHouse 表

以下代码片段在com.huawei.clickhouse.examples包的“Demo”类的dropTable方法中，用于删除在**创建ClickHouse表**中创建的副本表和分布式表。

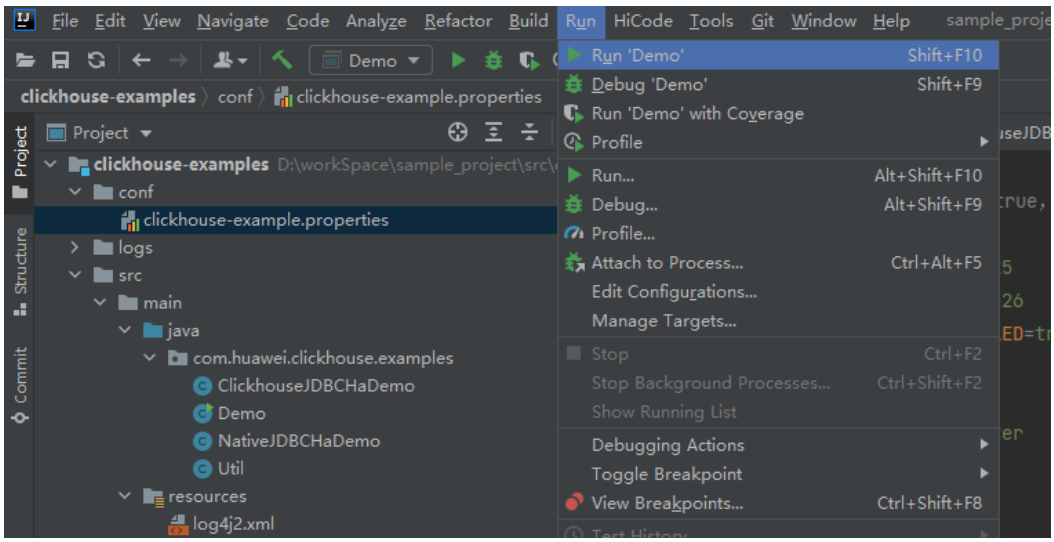
```
private void dropTable(String databaseName, String tableName, String clusterName) throws Exception {
    String dropLocalTableSql = "drop table if exists " + databaseName + "." + tableName + " on cluster " + clusterName;
    String dropDisTableSql = "drop table if exists " + databaseName + "." + tableName + "_all" + " on cluster " + clusterName;
    ArrayList<String> sqlList = new ArrayList<String>();
    sqlList.add(dropLocalTableSql);
    sqlList.add(dropDisTableSql);
    util.exeSql(sqlList);
}
```

8.4 调测 ClickHouse 应用

8.4.1 在本地 Windows 环境中调测 ClickHouse 应用

编译并运行程序

在程序代码完成开发后，您可以在Windows环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。在开发环境IntelliJ IDEA工程“clickhouse-examples”中单击“Run 'Demo'”运行应用程序工程。



查看调测结果

ClickHouse应用程序运行完成后，可通过以下方式查看程序运行情况：

- 通过运行结果查看程序运行情况。
- 通过ClickHouse日志获取应用运行情况，即**logs**目录下的日志文件：clickhouse-example.log。

运行clickhouse-examples的完整样例后，控制台显示部分运行结果如下：

```
Connected to the target VM, address: '127.0.0.1:53321', transport: 'socket'
2021-04-21 21:02:16,914 | INFO  | main | Driver registered |
ru.yandex.clickhouse.ClickHouseDriver.<clinit>(ClickHouseDriver.java:49)
```

```
2021-04-21 21:02:16,918 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:17,517 | INFO  | main | Execute query:drop table if exists testdb.testtb on cluster
default_cluster | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:17,656 | INFO  | main | Execute time is 139 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:17,657 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:17,701 | INFO  | main | Execute query:drop table if exists testdb.testtb_all on cluster
default_cluster | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:17,851 | INFO  | main | Execute time is 148 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:17,851 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:17,895 | INFO  | main | Execute query:create database if not exists testdb on cluster
default_cluster | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:18,043 | INFO  | main | Execute time is 148 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:18,044 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:18,088 | INFO  | main | Execute query:create table testdb.testtb on cluster default_cluster
(name String, age UInt8, date Date)engine=ReplicatedMergeTree('/clickhouse/tables/{shard}/
testdb.testtb','{replica}') partition by toYYYYMM(date) order by age |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:18,233 | INFO  | main | Execute time is 144 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:18,233 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:18,278 | INFO  | main | Execute query:create table testdb.testtb_all on cluster
default_cluster as testdb.testtb ENGINE = Distributed(default_cluster,testdb,testtb, rand()); |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:18,422 | INFO  | main | Execute time is 144 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:18,423 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.insertData(Util.java:128)
2021-04-21 21:02:19,380 | INFO  | main | Inert batch time is 720 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:19,927 | INFO  | main | Inert batch time is 492 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:20,456 | INFO  | main | Inert batch time is 504 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:20,894 | INFO  | main | Inert batch time is 410 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:21,348 | INFO  | main | Inert batch time is 431 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:21,813 | INFO  | main | Inert batch time is 442 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:22,273 | INFO  | main | Inert batch time is 434 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:22,730 | INFO  | main | Inert batch time is 435 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:23,212 | INFO  | main | Inert batch time is 459 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:23,689 | INFO  | main | Inert batch time is 452 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-04-21 21:02:23,689 | INFO  | main | Inert all batch time is 5216 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:148)
2021-04-21 21:02:23,689 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:23,732 | INFO  | main | Execute query:select * from testdb.testtb_all order by age limit 10 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:23,803 | INFO  | main | Execute time is 71 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:23,804 | INFO  | main | Current load balancer is 10.120.147.36:21425 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-04-21 21:02:23,848 | INFO  | main | Execute query:select toYYYYMM(date),count(1) from
testdb.testtb_all group by toYYYYMM(date) order by count(1) DESC limit 10 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-04-21 21:02:23,895 | INFO  | main | Execute time is 47 ms |
```

```
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-04-21 21:02:23,896 | INFO  | main | name    age    date    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_4077    0    2021-12-21    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_183    0    2021-12-10    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_5407    0    2021-12-13    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_1072    0    2021-12-03    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_4667    0    2021-12-22    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_1767    0    2021-12-03    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_8001    0    2021-12-22    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_1822    0    2021-12-04    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_5095    0    2021-12-23    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,896 | INFO  | main | huawei_7133    0    2021-12-26    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | toYYYYMM(date)    count()    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202101    2184    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202105    2176    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201810    2173    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201907    2162    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201803    2159    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201805    2153    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202110    2145    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201801    2144    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 201908    2143    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-04-21 21:02:23,897 | INFO  | main | 202005    2133    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
Disconnected from the target VM, address: '127.0.0.1:53321', transport: 'socket'
Process finished with exit code 0
```

8.4.2 在 Linux 环境中调测 ClickHouse 应用

ClickHouse应用程序支持在Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。

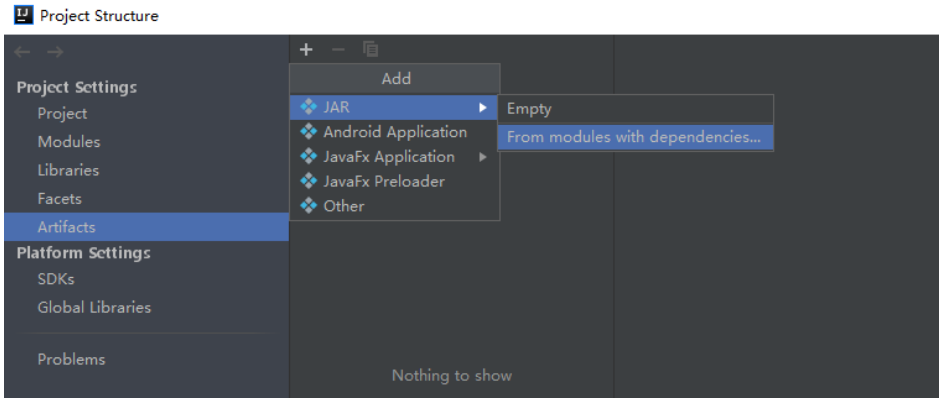
前提条件

Linux环境已安装JDK，版本号需要和IntelliJ IDEA导出Jar包使用的JDK版本一致，并设置好Java环境变量。

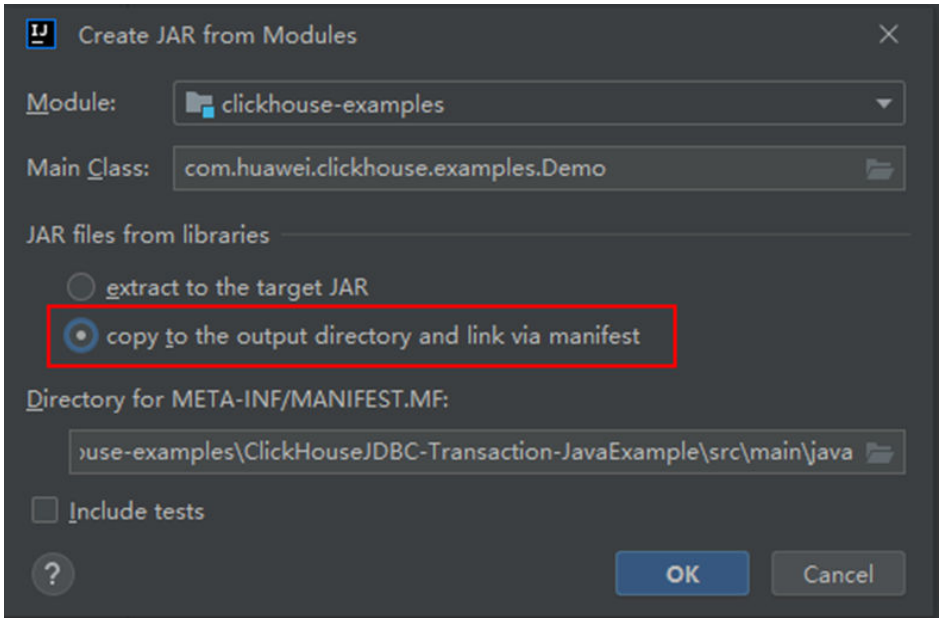
编译并运行程序

步骤1 导出jar包

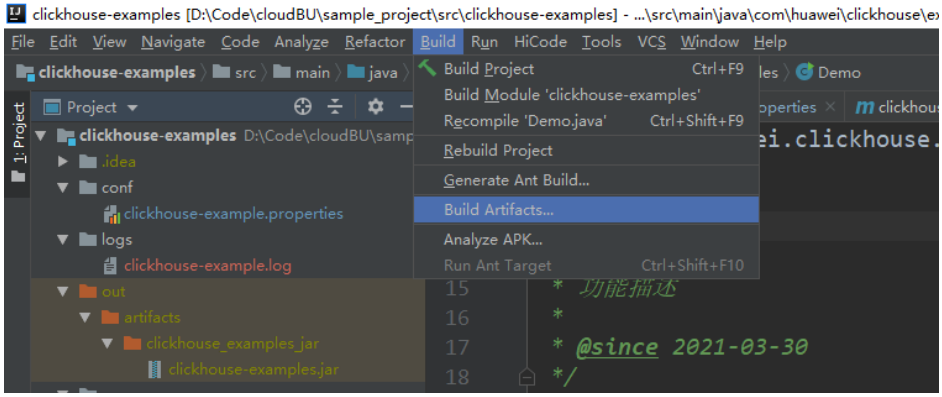
1. 进入IntelliJ IDEA，选择“File > Project Structure > Artifacts”。
2. 单击“加号”，选择“JAR > From modules with dependencies”。



3. “Main Class”选择“com.huawei.clickhouse.examples.Demo”，单击OK。



4. 选择“Build> Build Artifacts”。编译成功后在“clickhouse-examples\out\artifacts\”目录下查看并获取当前目录的所有jar文件。



步骤2 将“clickhouse-examples\out\artifacts\clickhouse_examples.jar”目录下的所有jar文件和“clickhouse-examples”目录下的“conf”文件夹复制到ClickHouse客户端安装目录下，例如“*客户端安装目录*/JDBC”目录下。

步骤3 登录客户端节点，进入jar文件上传目录下，修改文件权限为700。

cd /opt/client

chmod 700 clickhouse-examples.jar

步骤4 在“clickhouse_examples.jar”所在客户端目录下执行如下命令运行jar包：

source *客户端安装目录*/**bigdata_env**

java -cp ./*:conf/clickhouse-example.properties com.huawei.clickhouse.examples.Demo

----结束

查看调测结果

ClickHouse应用程序运行完成后，可通过以下方式查看程序运行情况：

- 通过运行结果查看程序运行情况。
- 通过ClickHouse日志获取应用运行情况。

  即查看当前jar文件所在目录的“logs/clickhouse-example.log”日志文件，例如“*客户端安装目录*/**JDBC/logs/clickhouse-example.log**”。

  jar包运行结果如下：

```
2021-06-10 20:53:56,028 | INFO  | main | Current load balancer is 10.112.17.150:21426 |
com.huawei.clickhouse.examples.Util.insertData(Util.java:128)
2021-06-10 20:53:58,247 | INFO  | main | Inert batch time is 1442 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:53:59,649 | INFO  | main | Inert batch time is 1313 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:05,872 | INFO  | main | Inert batch time is 6132 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:10,223 | INFO  | main | Inert batch time is 4272 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:11,614 | INFO  | main | Inert batch time is 1300 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:12,871 | INFO  | main | Inert batch time is 1200 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:14,589 | INFO  | main | Inert batch time is 1663 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:16,141 | INFO  | main | Inert batch time is 1500 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:17,690 | INFO  | main | Inert batch time is 1498 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:19,206 | INFO  | main | Inert batch time is 1468 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:145)
2021-06-10 20:54:19,207 | INFO  | main | Inert all batch time is 22626 ms |
com.huawei.clickhouse.examples.Util.insertData(Util.java:148)
2021-06-10 20:54:19,208 | INFO  | main | Current load balancer is 10.112.17.150:21426 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-06-10 20:54:20,231 | INFO  | main | Execute query:select * from mutong1.testtb_all order by age
limit 10 | com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-06-10 20:54:21,266 | INFO  | main | Execute time is 1035 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-06-10 20:54:21,267 | INFO  | main | Current load balancer is 10.112.17.150:21426 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:58)
2021-06-10 20:54:21,815 | INFO  | main | Execute query:select toYYYYMM(date),count(1) from
mutong1.testtb_all group by toYYYYMM(date) order by count(1) DESC limit 10 |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:63)
2021-06-10 20:54:22,897 | INFO  | main | Execute time is 1082 ms |
com.huawei.clickhouse.examples.Util.exeSql(Util.java:67)
2021-06-10 20:54:22,898 | INFO  | main | name    age    date    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,898 | INFO  | main | huawei_266    0    2021-12-19    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_2500    0    2021-12-29    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_8980    0    2021-12-16    |
```



```
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_671    0    2021-12-29    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_2225    0    2021-12-12    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_6040    0    2021-12-14    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_7294    0    2021-12-10    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,899 | INFO  | main | huawei_1133    0    2021-12-25    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | huawei_3161    0    2021-12-21    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | huawei_3992    0    2021-11-25    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | toYYYYMM(date)    count()    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 201910    2247    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 202105    2213    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 201801    2208    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,900 | INFO  | main | 201803    2204    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201810    2167    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201805    2166    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201901    2164    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201908    2145    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 201912    2143    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
2021-06-10 20:54:22,901 | INFO  | main | 202107    2137    |
com.huawei.clickhouse.examples.Demo.queryData(Demo.java:144)
```

9 Flink 开发指南（安全模式）

9.1 Flink 应用开发简介

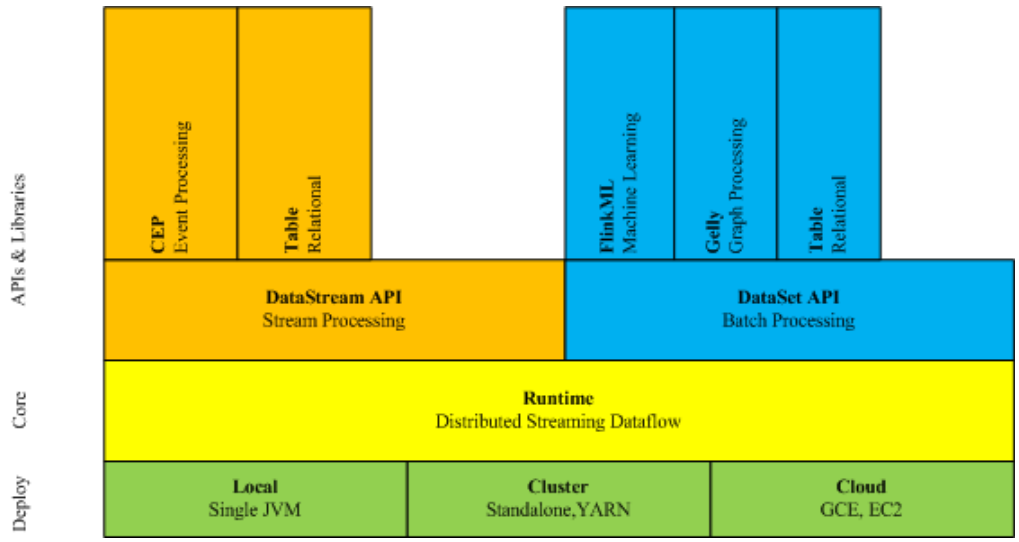
简介

Flink是一个批处理和流处理结合的统一计算框架，其核心是一个提供了数据分发以及并行化计算的流数据处理引擎。它的最大亮点是流处理，是业界最顶级的开源流处理引擎。

Flink最适合的应用场景是低时延的数据处理（Data Processing）场景：高并发pipeline处理数据，时延毫秒级，且兼具可靠性。

Flink技术栈如图9-1所示。

图 9-1 Flink 技术栈



Flink在当前版本中重点构建如下特性，其他特性继承开源社区，不做增强，具体请参考：https://ci.apache.org/projects/flink/flink-docs-release-1.15。

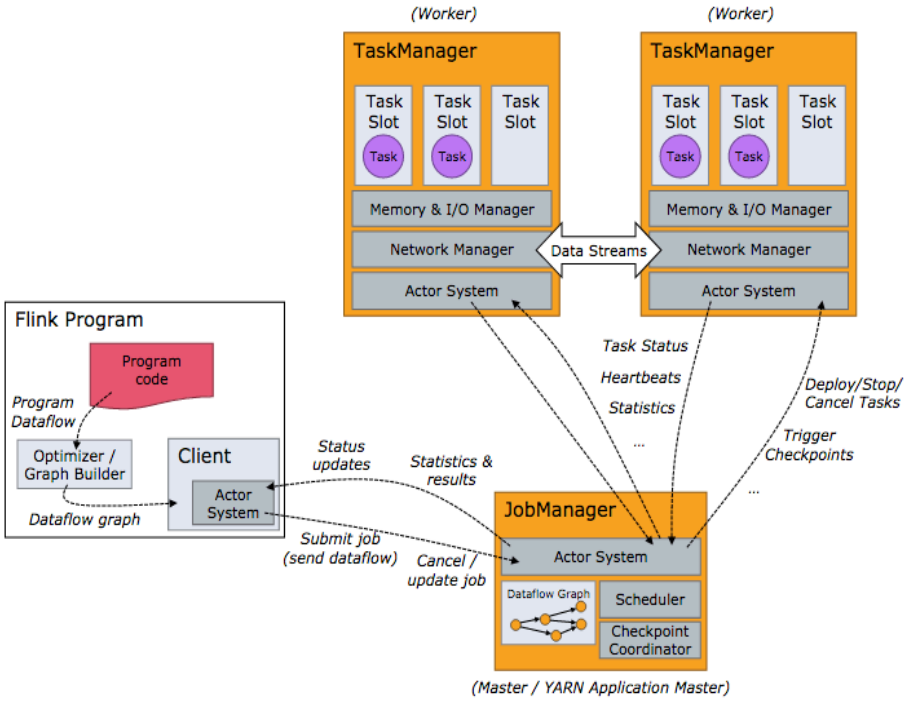
- DataStream

- Checkpoint
- 窗口
- Job Pipeline
- 配置表

结构

Flink结构如图9-2所示。

图 9-2 Flink 结构



Flink整个系统包含三个部分：

- Client

  Flink Client主要给用户提供向Flink系统提交用户任务（流式作业）的能力。

- TaskManager

  Flink系统的业务执行节点，执行具体的用户任务。TaskManager可以有多个，各个TaskManager都平等。

- JobManager

  Flink系统的管理节点，管理所有的TaskManager，并决策用户任务在哪些Taskmanager执行。JobManager在HA模式下可以有多个，但只有一个主JobManager。

Flink系统提供的关键能力：

- 低时延

  提供ms级时延的处理能力。

- ExactlyOnce

  提供异步快照机制，保证所有数据真正只处理一次。
- HA

  JobManager支持主备模式，保证无单点故障。
- 水平扩展能力

  TaskManager支持手动水平扩展。

Flink 开发接口简介

Flink DataStream API提供Scala和Java两种语言的开发方式，如表9-1所示。

表 9-1 Flink DataStream API 接口

功能	说明
Scala API	提供Scala语言的API，提供过滤、join、窗口、聚合等数据处理能力。
Java API	提供Java语言的API，提供过滤、join、窗口、聚合等数据处理能力。

基本概念

- **DataStream**

  数据流，是指Flink系统处理的最小数据单元。该数据单元最初由外部系统导入，可以通过Socket、Kafka和文件等形式导入，在Flink系统处理后，在通过Socket、Kafka和文件等输出到外部系统，这是Flink的核心概念。
- **Data Transformation**

  数据处理单元，会将一或多个DataStream转换成一个新的DataStream。

  具体可以细分如下几类：
  - 一对一的转换：如Map。
  - 一对0、1或多个的转换：如FlatMap。
  - 一对0或1的转换，如Filter。
  - 多对1转换，如Union。
  - 多个聚合的转换，如window、keyby。
- **CheckPoint**

  CheckPoint是Flink数据处理高可靠、最重要的机制。该机制可以保证应用在运行过程中出现失败时，应用的所有状态能够从某一个检查点恢复，保证数据仅被处理一次（Exactly Once）。
- **SavePoint**

  Savepoint是指允许用户在持久化存储中保存某个checkpoint，以便用户可以暂停自己的任务进行升级。升级完后将任务状态设置为savepoint存储的状态开始恢复运行，保证数据处理的延续性。

样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Flink相关样例工程，安全模式路径为“flink-examples/flink-examples-security”，普通模式路径为“flink-examples/flink-examples-normal”：

表 9-2 Flink 相关样例工程

样例工程	描述
FlinkCheckpointJavaExample FlinkCheckpointScalaExample	异步Checkpoint机制程序的应用开发示例。 假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性，即：当应用出现异常并恢复后，各个算子的状态能够处于统一的状态，相关业务场景介绍请参见**Flink开启Checkpoint样例程序**。
FlinkKafkaJavaExample FlinkKafkaScalaExample	向Kafka生产并消费数据程序的应用开发示例。 通过调用flink-connector-kafka模块的接口，生产并消费数据，相关业务场景介绍请参见**Flink Kafka样例程序**。
FlinkPipelineJavaExample FlinkPipelineScalaExample	Job Pipeline程序的应用开发示例。 相关业务场景介绍请参见**Flink Job Pipeline样例程序**。 发布者Job自己每秒钟产生10000条数据，然后经由该job的NettySink算子向下游发送。另外两个Job作为订阅者，分别订阅一份数据并打印输出。
FlinkSqlJavaExample	使用客户端通过jar作业提交SQL作业的应用开发示例。
FlinkStreamJavaExample FlinkStreamScalaExample	DataStream程序的应用开发示例。 相关业务场景介绍请参见**Flink DataStream样例程序**。 假定用户有某个网站周末网民网购停留时间的日志文本，另有一张网民个人信息的csv格式表，可通过Flink应用程序实现例如实时统计总计网购时间超过2个小时的女性网民信息，包含对应的个人详细信息的功能。
FlinkStreamSqlJoinExample	Stream SQL Join程序的应用开发示例。 相关业务场景介绍请参见**Flink Join样例程序**。 假定某个Flink业务1每秒就会收到1条消息记录，消息记录某个用户的基本信息，包括名字、性别、年龄。另有一个Flink业务2会不定时收到1条消息记录，消息记录该用户的名字、职业信息。实现实时的以根据业务2中消息记录的用户名字作为关键字，对两个业务数据进行联合查询的功能。

9.2 Flink 应用开发流程介绍

Flink 应用程序开发流程

Flink开发流程参考如下步骤：

图 9-3 Flink 应用程序开发流程

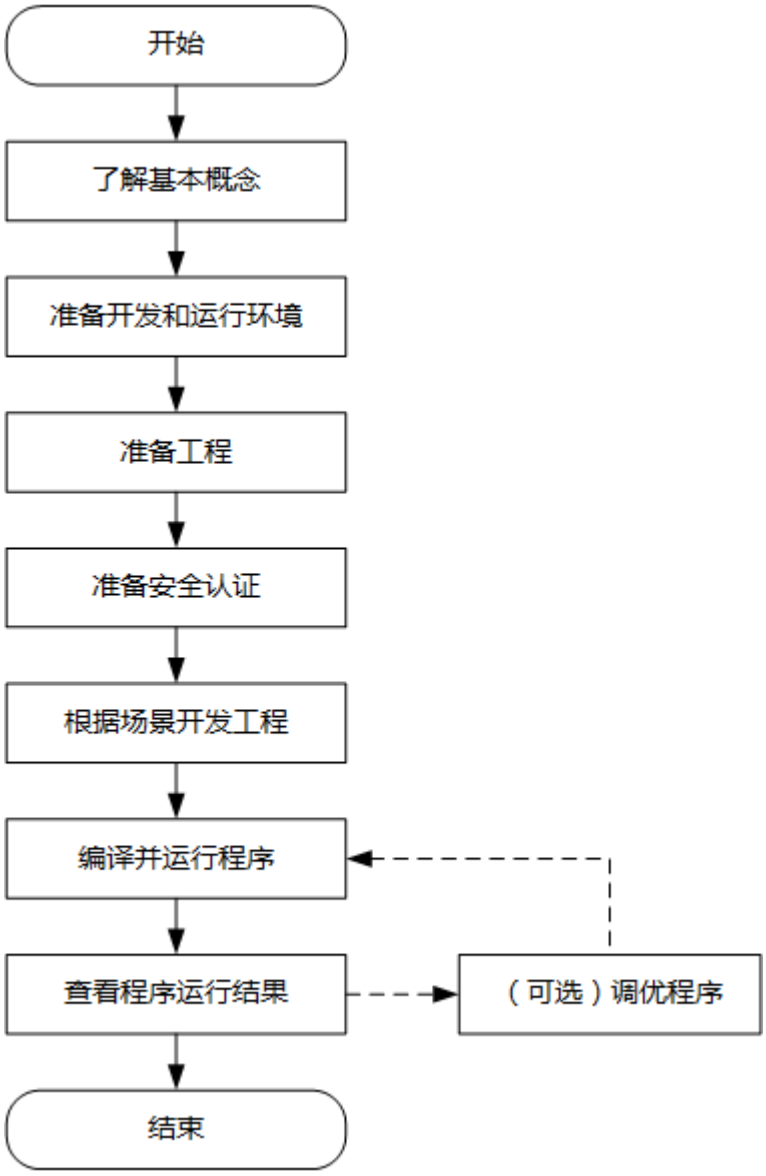


表 9-3 Flink 应用开发的流程说明

阶段	说明	参考章节
了解基本概念	在开始开发应用前，需要了解Flink的基本概念。	基本概念

阶段	说明	参考章节
准备开发和运行环境	Flink的应用程序支持使用Scala、Java两种语言进行开发。推荐使用IDEA工具，请根据指导完成不同语言的开发环境配置。Flink的运行环境即Flink客户端，请根据指导完成客户端的安装和配置。	**准备本地应用开发环境** **准备连接集群配置文件**
准备工程	Flink提供了样例程序，您可以导入样例工程进行程序学习。或者您可以根据指导，新建一个Flink工程。	**导入并配置Flink样例工程**
准备安全认证	如果您使用的是安全集群，需要进行安全认证。	**准备Flink安全认证**
根据场景开发工程	提供了Scala、Java两种不同语言的样例工程，帮助用户快速了解Flink各部件的编程接口。	**开发Flink应用**
编译并运行程序	指导用户将开发好的程序编译并提交运行。	**编译并调测Flink应用**
查看程序运行结果	程序运行结果会写在用户指定的路径下，用户还可以通过UI查看应用运行情况。	**查看Flink应用调测结果**
调优程序	您可以根据程序运行情况，对程序进行调优，使其性能满足业务场景需求。 调优完成后，请重新进行编译和运行。	组件操作指南中的“Flink性能调优”

9.3 准备 Flink 应用开发环境

9.3.1 准备本地应用开发环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如**表9-4**所示。

表 9-4 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。

准备项	说明
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 ● X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 ● TaiShan客户端：OpenJDK：支持1.8.0_272版本。
安装和配置IDEA	用于开发Flink应用程序的工具。版本要求：2019.1或其他兼容版本。
安装Scala	Scala开发环境的基本配置。版本要求：2.11.7。
安装Scala插件	Scala开发环境的基本配置。版本要求：1.5.4。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

9.3.2 准备连接集群配置文件

准备集群认证用户信息

对于开启Kerberos认证的MRS集群，需提前准备具有相关组件操作权限的用户用于程序认证。

以下Flink权限配置示例供参考，在实际业务场景中可根据业务需求灵活调整。

步骤1 登录FusionInsight Manager。

步骤2 选择“系统 > 权限 > 角色 > 添加角色”，根据服务的权限控制类别添加业务开发时需要的权限。

1. 填写角色的名称，例如**developrole**。
2. 在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > flink ”，勾选“读”、“写”和“执行”，单击“配置资源权限”表格中“服务”返回。
3. 在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选default的“提交”，单击“确定”保存。

   说明

   若指定state backend为HDFS，例如指定state backend目录为"hdfs://hacluster/flink-checkpoint"，需同样为"hdfs://hacluster/flink-checkpoint"目录配置“读”、“写”和“执行”权限。

步骤3 选择“系统 > 权限 > 用户组 > 添加用户组”，为样例工程创建一个用户组，例如**developgroup**。

步骤4 选择“系统 > 权限 > 用户 > 添加用户”，在新增用户界面，创建一个人机用户，例如**developuser**。

- “用户组”：加入“**developgroup**”和“**hadoop**”用户组，设置主组为“**developgroup**”。

  说明

  若用户需要对接Kafka，则需创建具有Flink和Kafka组件的混合集群，或者为拥有Flink组件的集群和拥有Kafka组件的集群配置跨集群互信，并将创建的Flink用户加入“kafkaadmin”用户组。

- “角色”：加入**步骤2**新增的**developrole**等角色。

步骤5 使用**developuser**用户登录FusionInsight Manager，首次登录需根据界面提示修改用户密码，修改成功后再次登录FusionInsight Manager。

步骤6 选择“系统 > 权限 > 用户”，在用户名为**developuser**的操作列选择“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到该用户的“user.keytab”文件与“krb5.conf”文件。

----结束

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

准备Linux环境运行程序所需配置文件。

1. 在节点中安装MRS集群客户端。例如客户端安装目录为“/opt/client”。

   说明

   - 客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。
   - 确保Flink客户端的“flink-conf.yaml”配置文件中的认证相关配置项已经配置正确，请参考**准备Flink安全认证**。
   - 安全模式下需要将客户端安装节点的业务IP地址以及Manager的浮动IP地址追加到“flink-conf.yaml”文件中的“jobmanager.web.allow-access-address”配置项中，IP地址之间使用英文逗号分隔。

2. 获取配置文件：

   a. 登录FusionInsight Manager下载客户端，“选择客户端类型”设置为“仅配置文件”，勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。

   b. 以**root**登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client”），解压软件包后获取“FusionInsight_Cluster_*x*_Services_ClientConfig/Flink/config”路径下的所有配置文件，放置到客户端节点中与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

   例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

   cd /tmp/FusionInsight-Client

   tar -xvf FusionInsight_Cluster_1_Services_Client.tar

   tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp Flink/config/* root@*客户端节点IP地址*:/opt/client/conf

准备集群认证用户信息获取的keytab文件也放置于该目录下，主要配置文件说明如**表9-5**所示。

表 9-5 配置文件

文件名称	作用
core-site.xml	配置Flink详细参数。
hdfs-site.xml	配置HDFS详细参数。
yarn-site.xml	配置Yarn详细参数。
flink-conf.yaml	Flink客户端配置文件。
user.keytab	对于Kerberos安全认证提供用户信息。
krb5.conf	Kerberos Server配置信息。

3. 检查客户端节点网络连接。

   在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

准备安全认证（安全模式集群）

参考**准备Flink安全认证**。

9.3.3 导入并配置 Flink 样例工程

操作场景

Flink针对多个场景提供样例工程，包含Java样例工程和Scala样例工程等，帮助客户快速学习Flink工程。

针对Java和Scala不同语言的工程，其导入方式相同。

以下操作步骤以导入Java样例代码为例。操作流程如**图9-4**所示。

图 9-4 导入样例工程流程

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\flink-examples”目录下的样例工程文件夹“flink-examples-security”，可根据实际业务场景选择对应的样例。

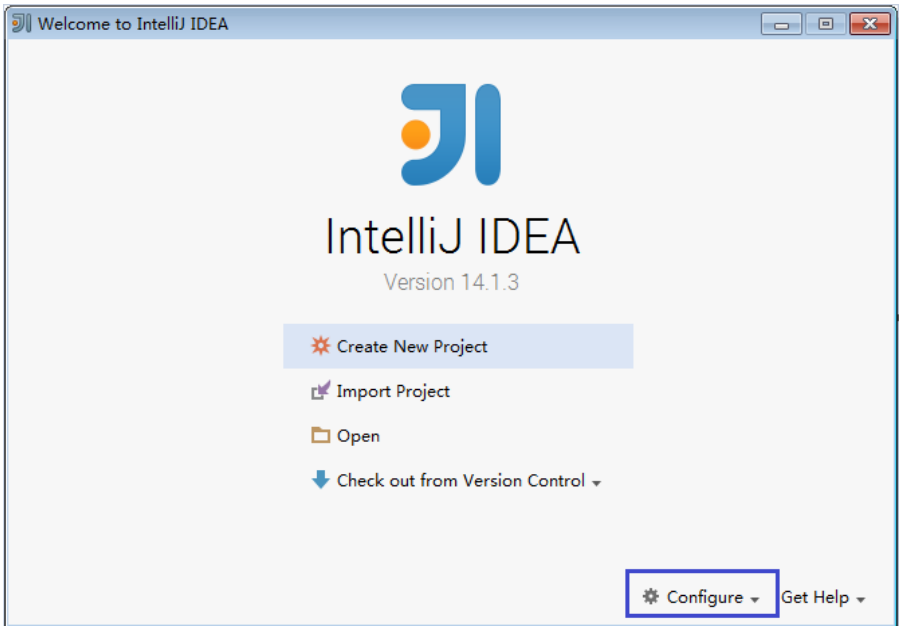
说明

- 在安全模式下，获取“src\flink-examples”下的样例工程flink-examples-security。
- 在普通模式下，获取“src\flink-examples”下的样例工程flink-examples-normal。

步骤2 在导入样例工程之前，IntelliJ IDEA需要进行配置JDK。

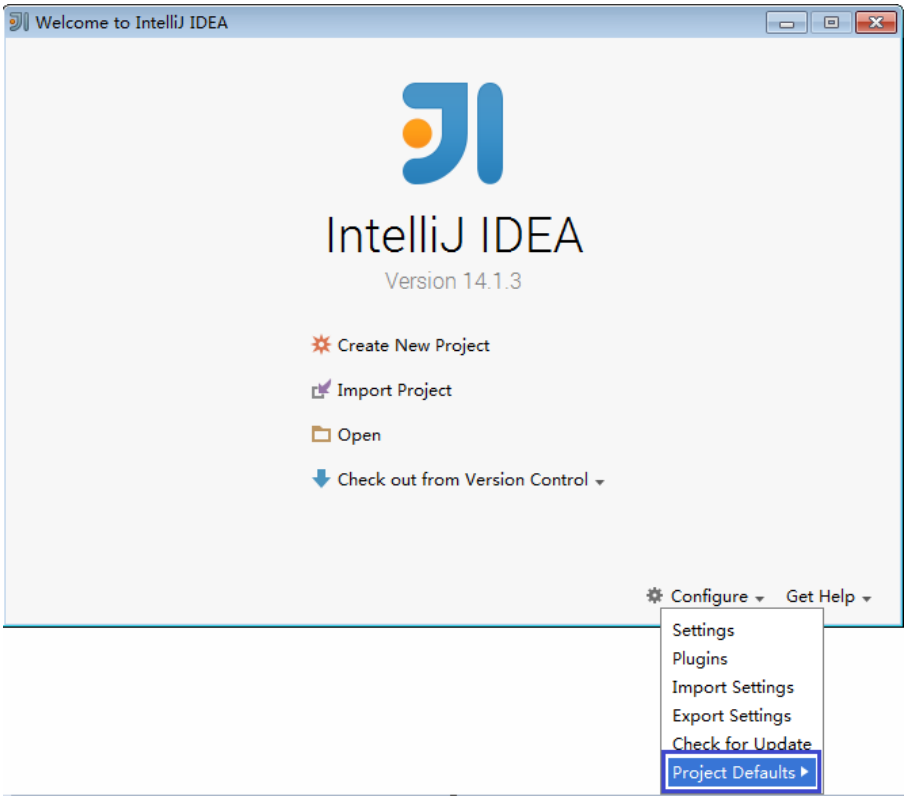
1. 打开IntelliJ IDEA，单击“Configure”下拉按钮。

图 9-5 Choosing Configure



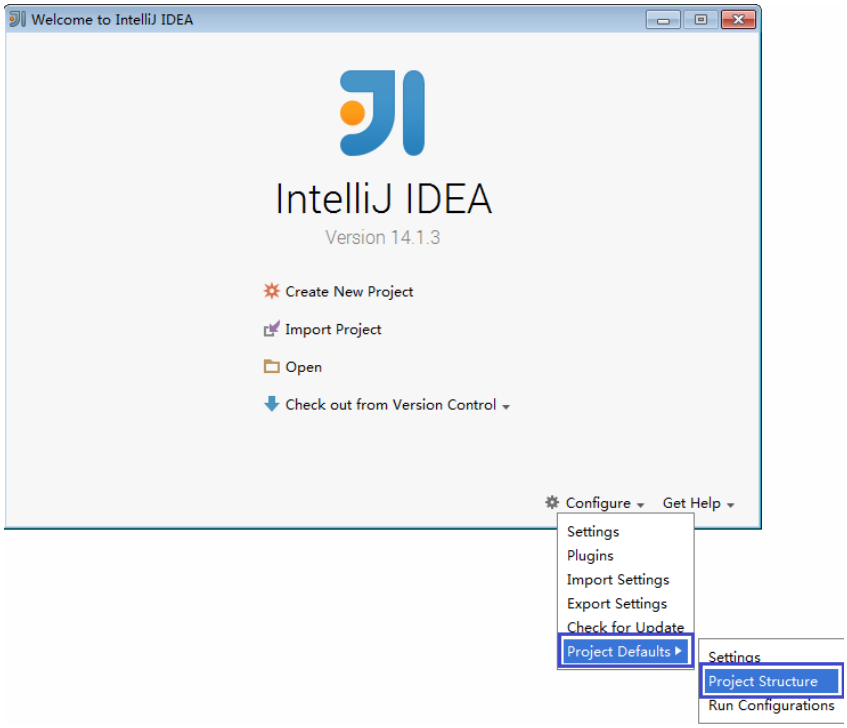
2. 在“Configure”下拉菜单中单击“Project Defaults”。

图 9-6 Choosing Project Defaults



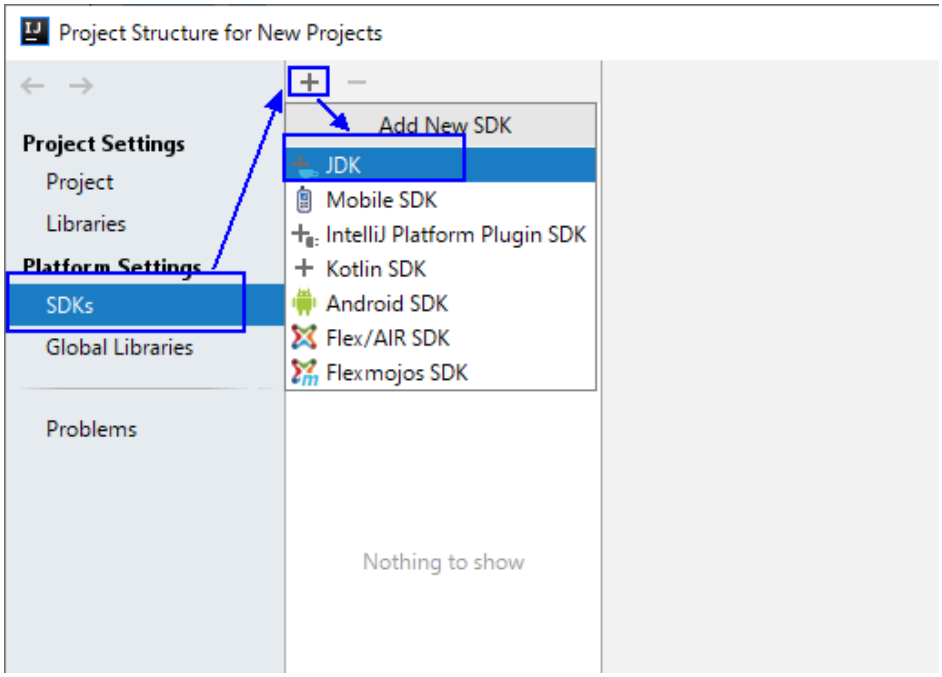
3. 在“Project Defaults”菜单中选择“Project Structure”。

图 9-7 Project Defaults



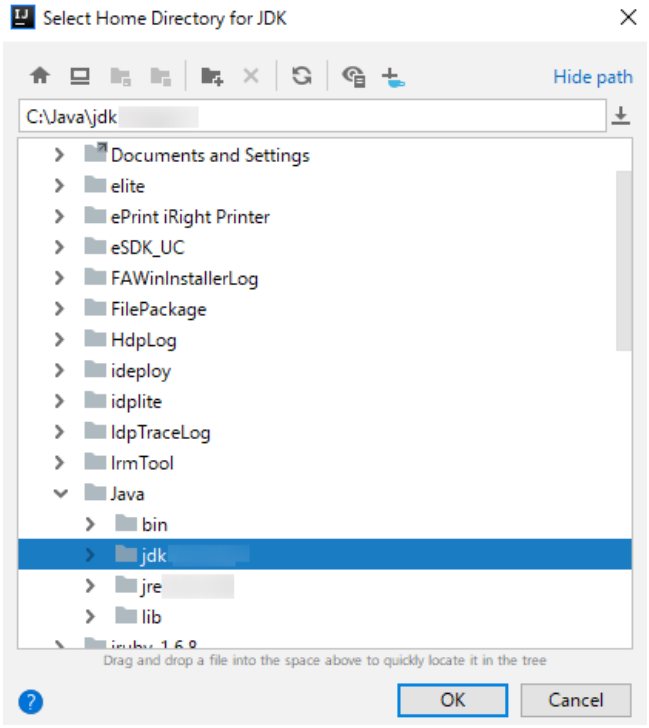
4. 在打开的“Project Structure”页面中，选择“SDKs”，单击绿色加号添加JDK。

图 9-8 添加 JDK



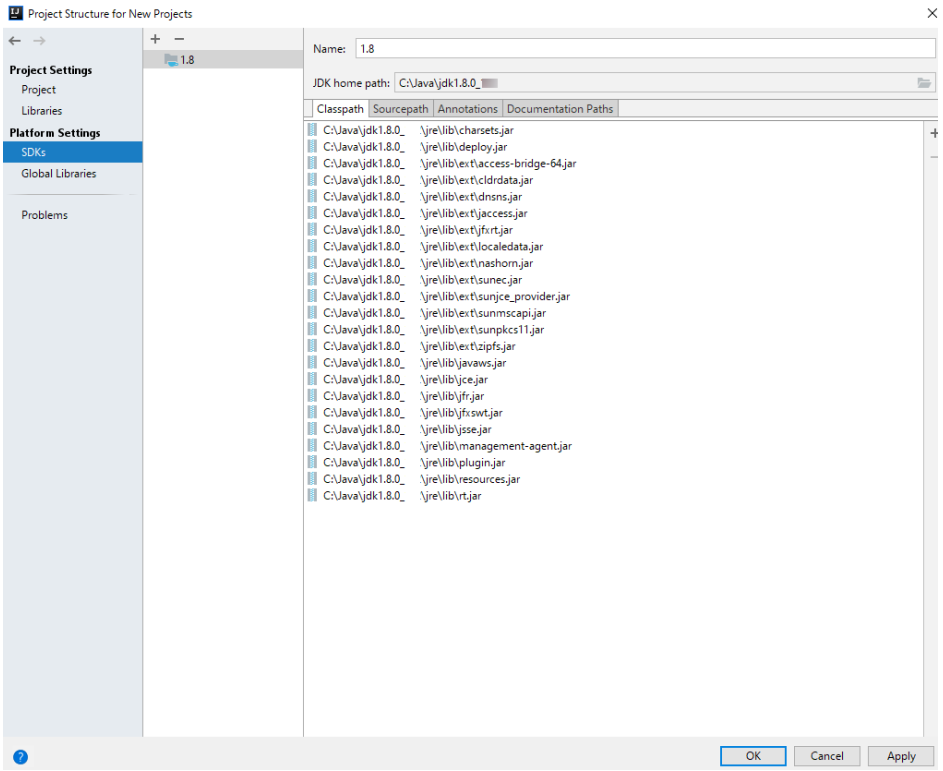
5. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 9-9 选择 JDK 目录



6. 完成JDK选择后，单击“OK”完成配置。

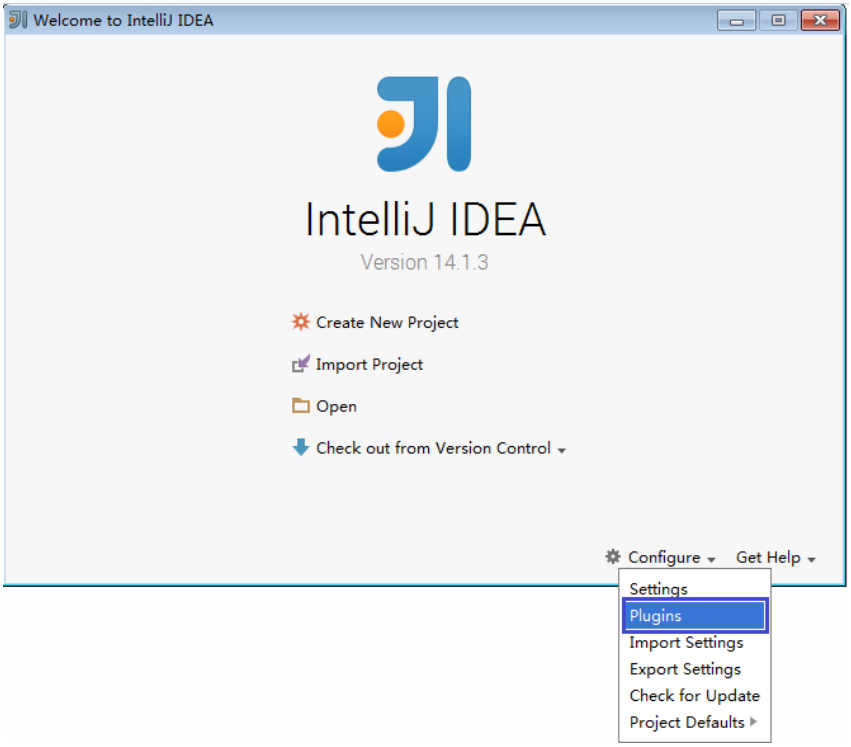
图 9-10 完成 JDK 配置



步骤3 （可选）如果导入Scala语言开发样例工程，还需要在IntelliJ IDEA中安装Scala插件。

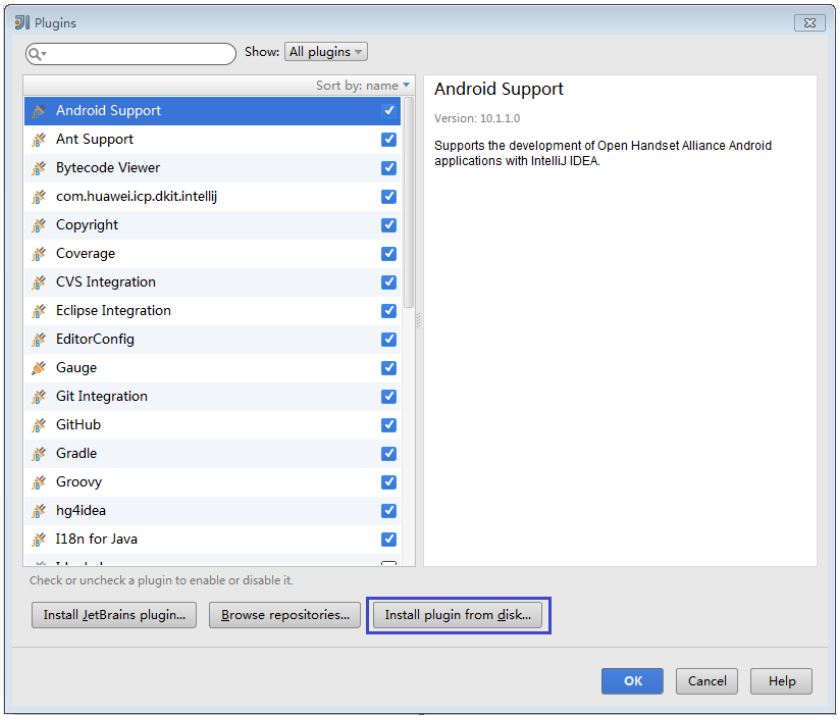
1. 在“Configure”下拉菜单中，单击“Plugins”。

图 9-11 Plugins



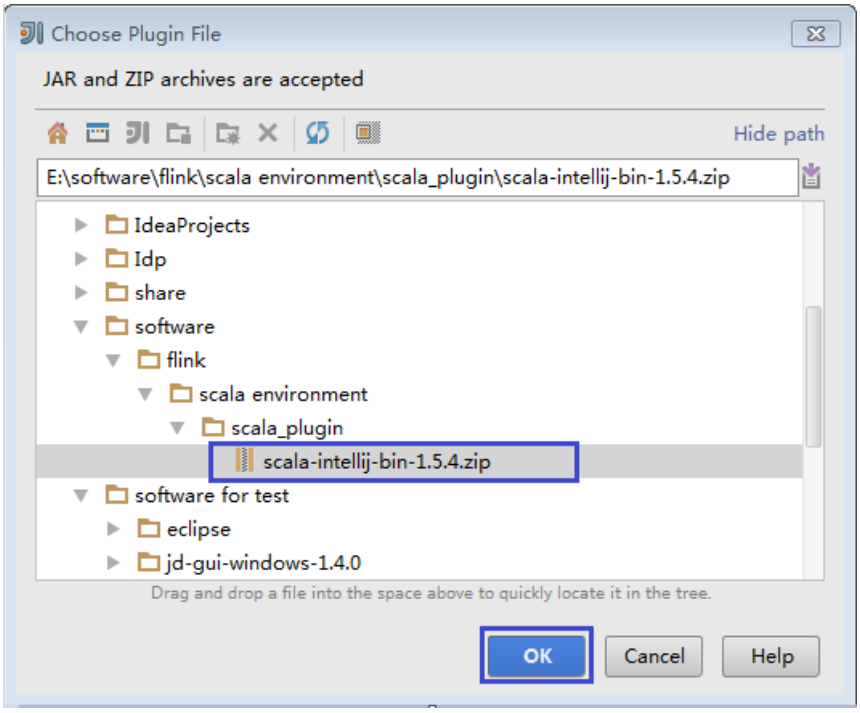
2. 在“Plugins”页面，选择“Install plugin from disk”。

图 9-12 Install plugin from disk



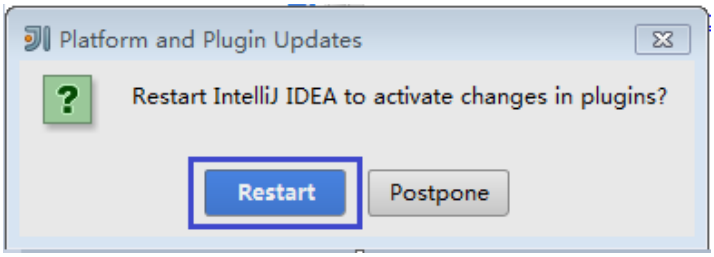
3. 在“Choose Plugin File”页面，选择对应版本的Scala插件包，单击“OK”。

图 9-13 choose plugin File



4. 在“Plugins”页面，单击“Apply”安装Scala插件。
5. 在弹出的“Platform and Plugin Updates”页面，单击“Restart”，使配置生效。

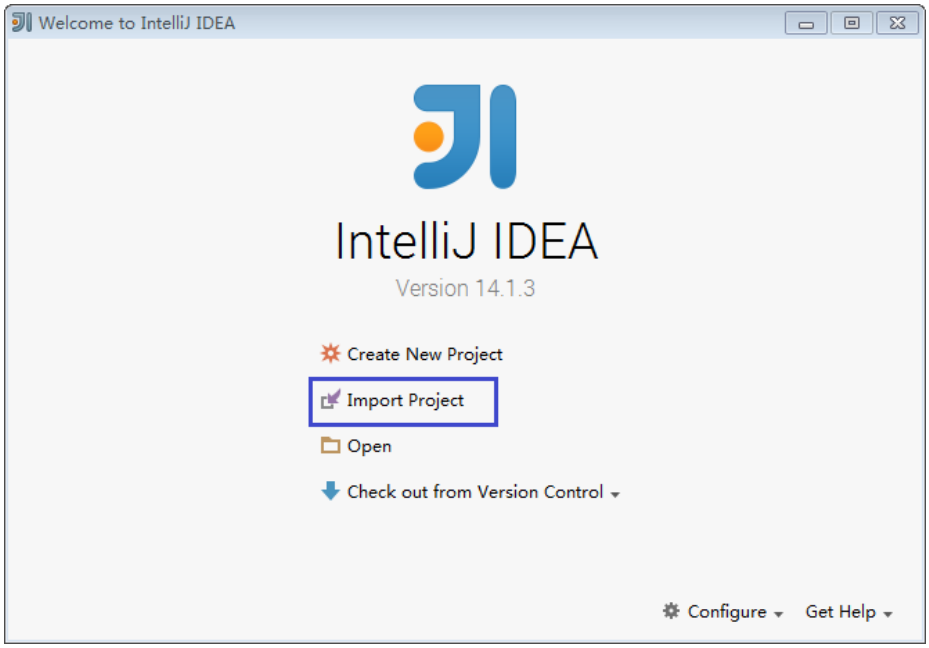
图 9-14 Platform and Plugin Updates



步骤4 将Java样例工程导入到IDEA中。

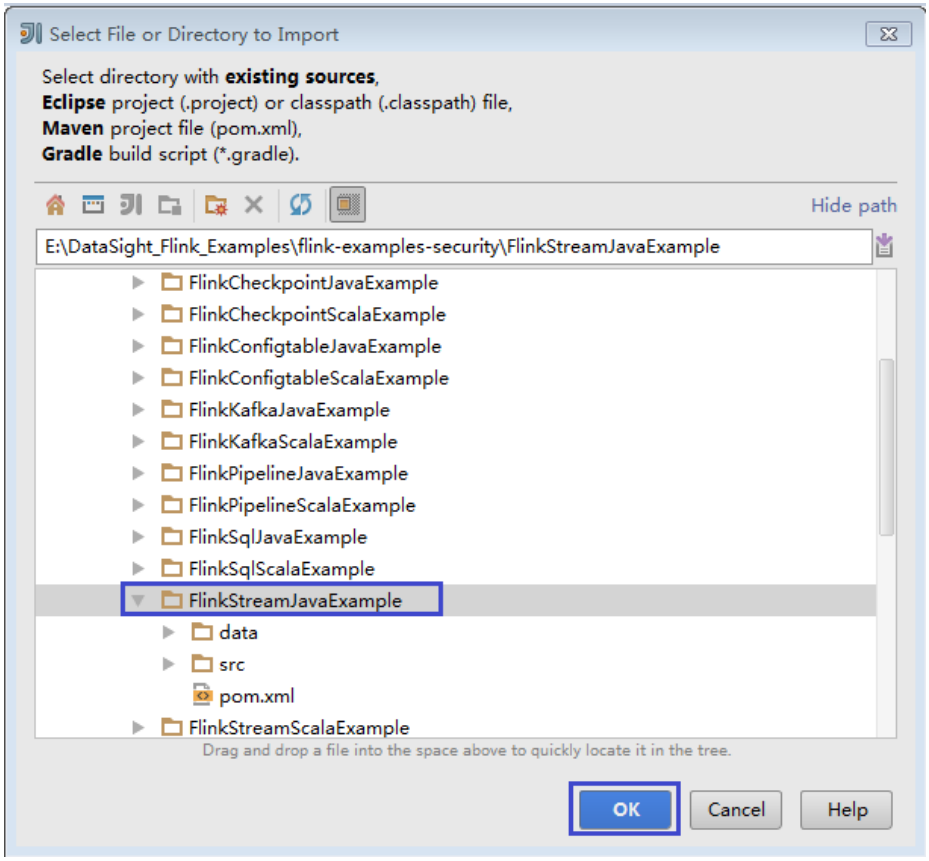
1. 打开IntelliJ IDEA。在“Quick Start”页面选择“Import Project”。
   或者，针对已使用过的IDEA工具，您可以从IDEA主界面直接添加。选择“File > Import project...”导入工程。

图 9-15 Import Project（Quick Start 页面）



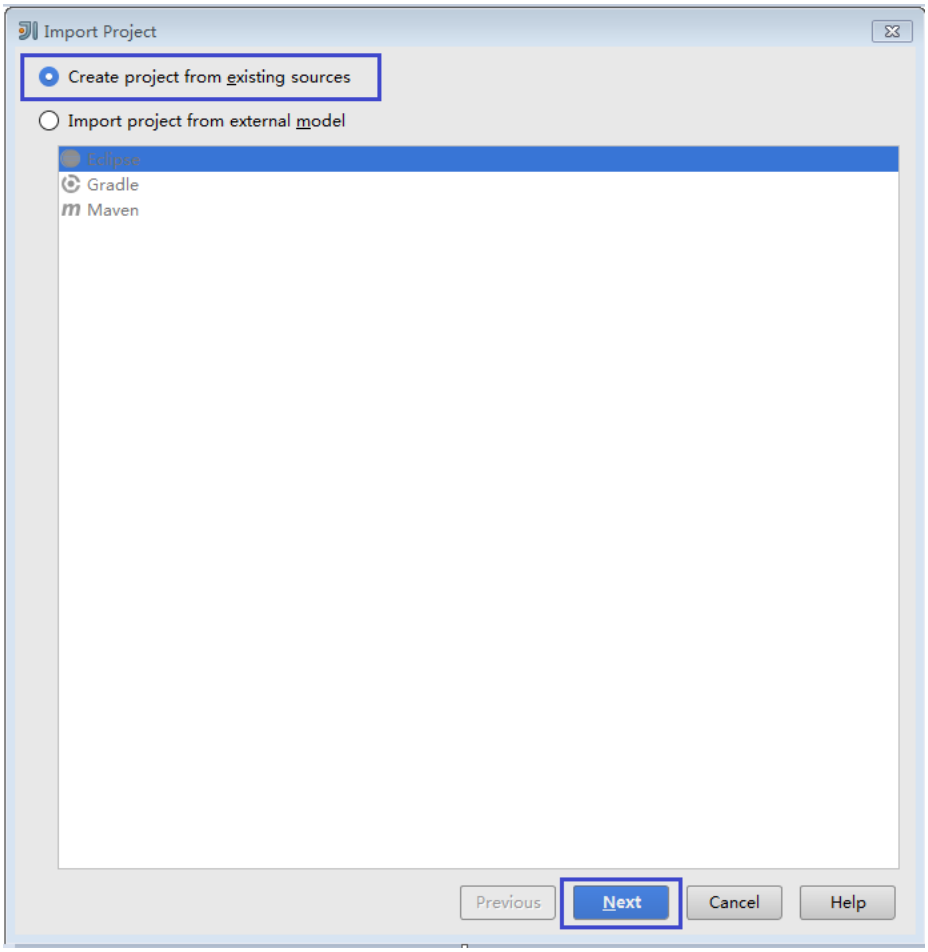
2. 选择需导入的样例工程路径，然后单击“OK”。

图 9-16 Select File or Directory to Import



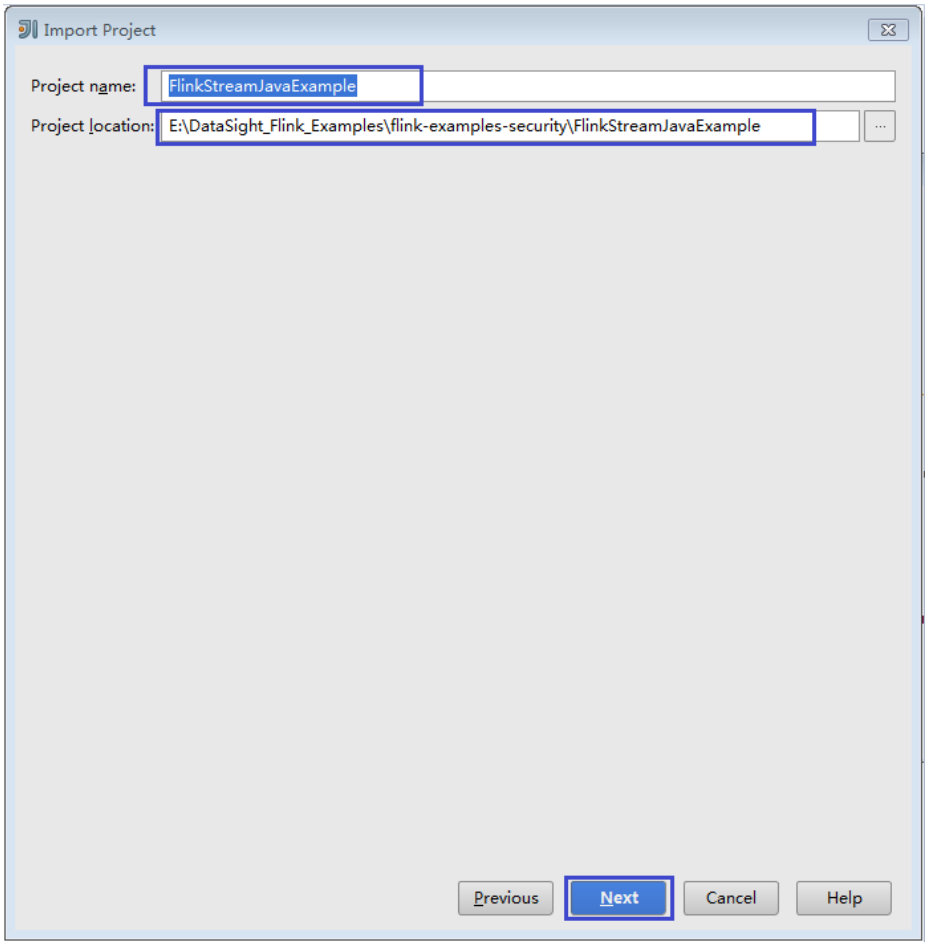
3. 选择从已存在的源码创建工程，然后单击“Next”。

图 9-17 Create project from existing sources



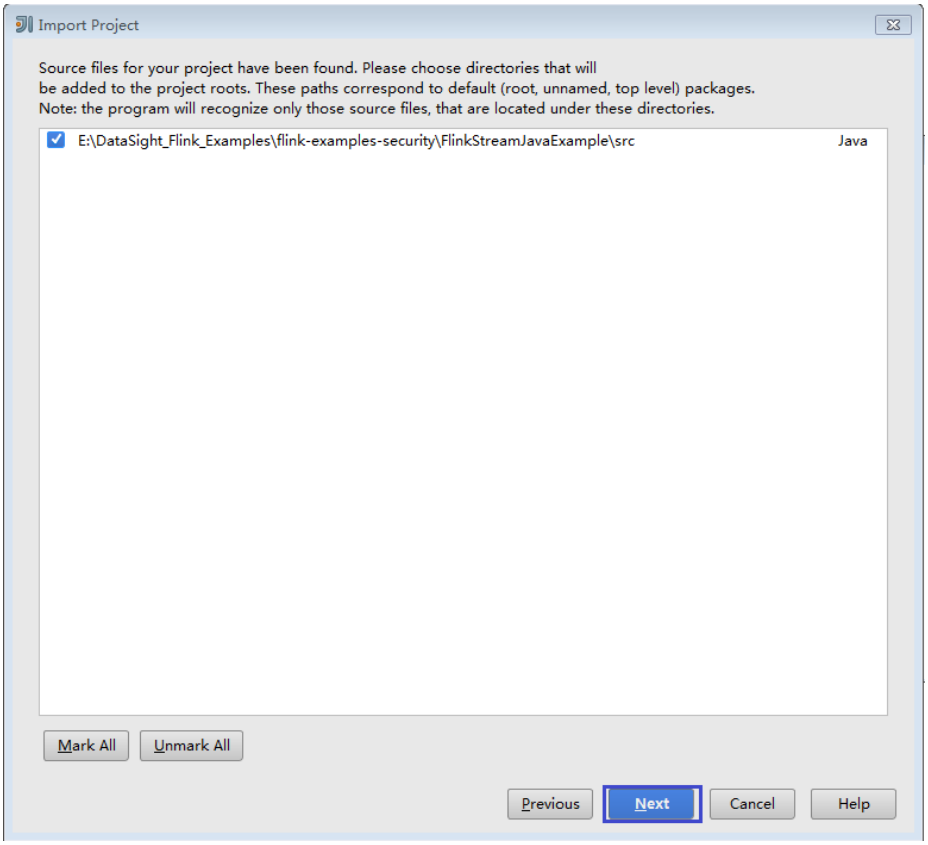
4. 确认导入路径和工程名称，单击“Next”。

图 9-18 Import Project



5. 确认导入工程的root目录，默认即可，单击“Next”。

图 9-19 Import Project



6. 确认IDEA自动识别的依赖库以及建议的模块结构，默认即可，单击“Next”。
7. 确认工程所用JDK，然后单击“Next”。
8. 导入结束，单击“Finish”，IDEA主页显示导入的样例工程。

图 9-20 导入结束

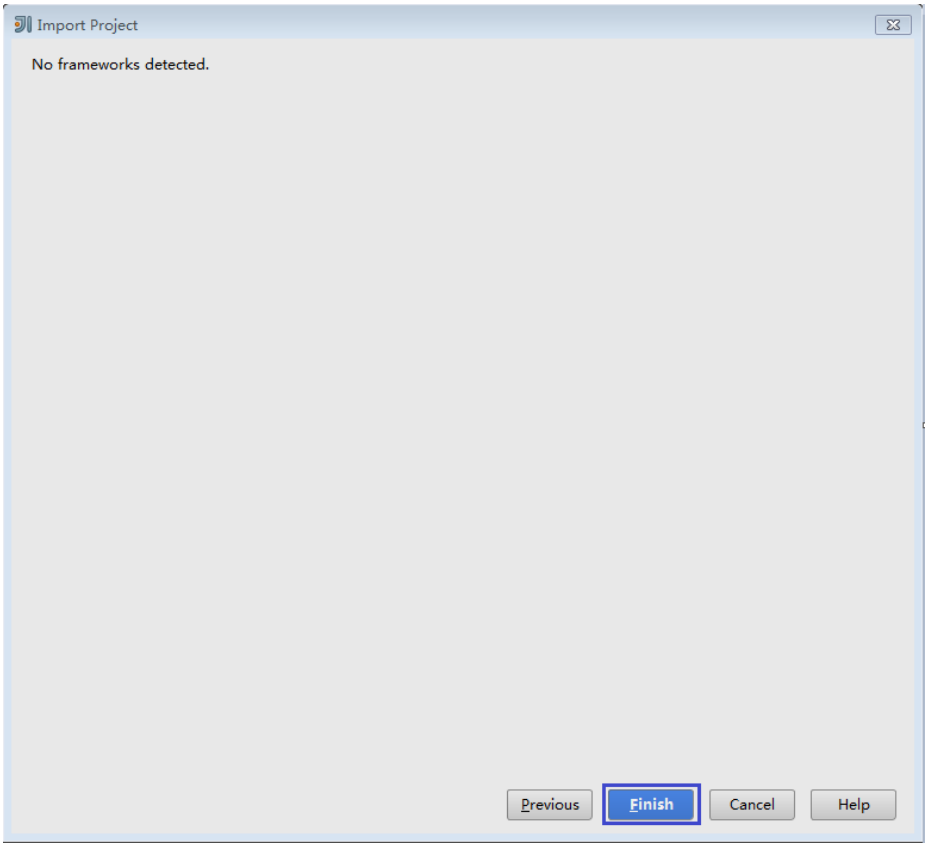
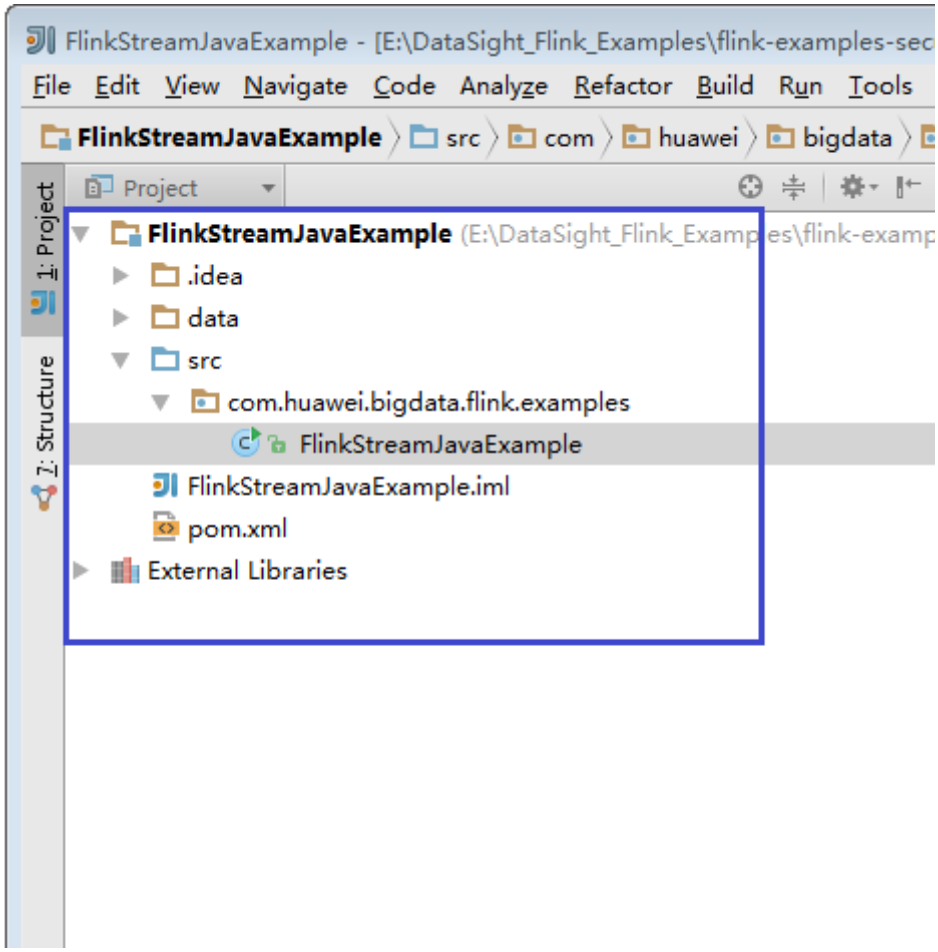


图 9-21 已导入工程



步骤5 导入样例工程依赖的Jar包。

如果通过开源镜像站方式获取的样例工程代码，在配置好Maven后， 相关依赖jar包将自动下载，不需手动添加。

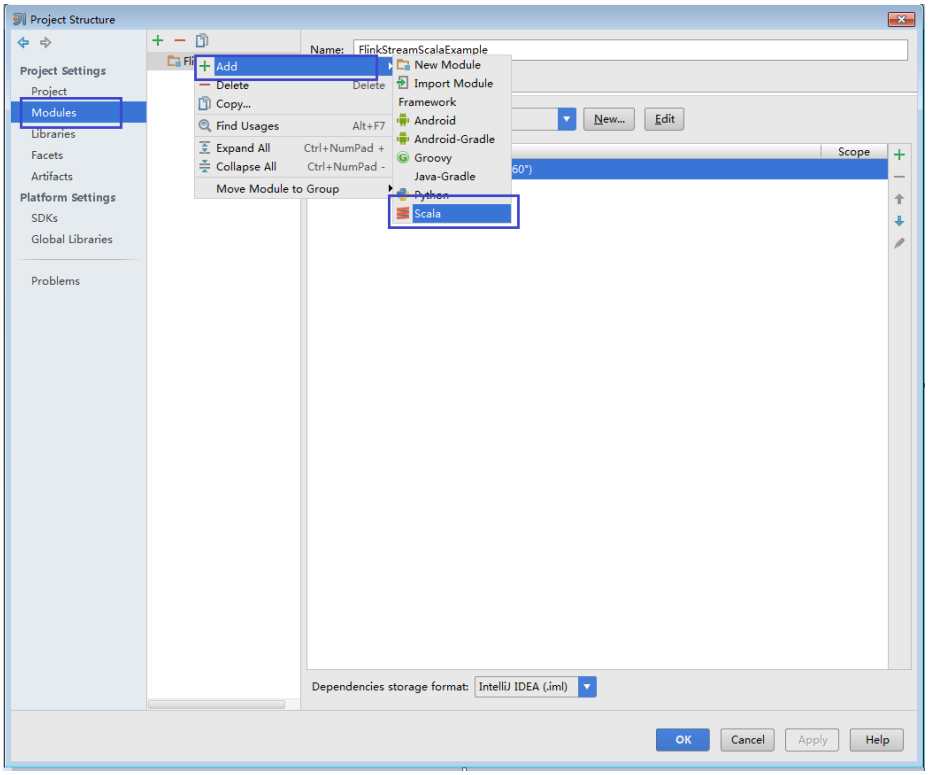
说明

当样例代码使用其他FusionInsight组件时，例如Kafka等，请去对应FusionInsight组件的服务端安装目录查找并添加依赖包。样例工程对应的依赖包详情，请参见**参考信息**。

步骤6 （可选）如果导入Scala语言开发的样例程序，还需要为工程设置语言。

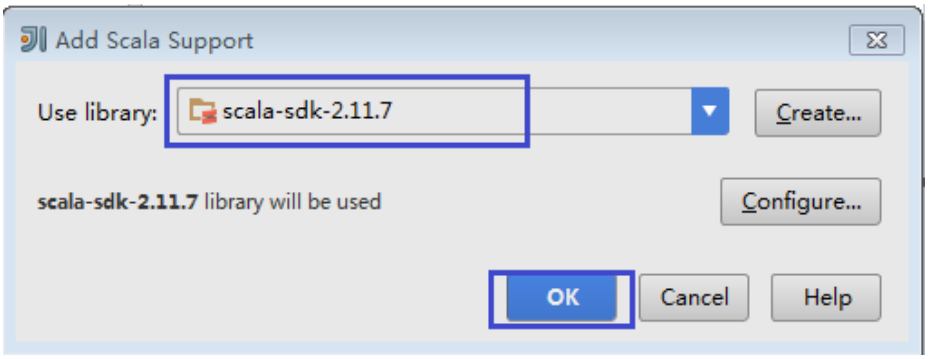
1. 在IDEA主页，选择“File>Project Structures...”进入“Project Structure”页面。
2. 选择“Modules”，选中工程名称，然后右键选择“Add > Scala”。

图 9-22 选择 Scala 语言



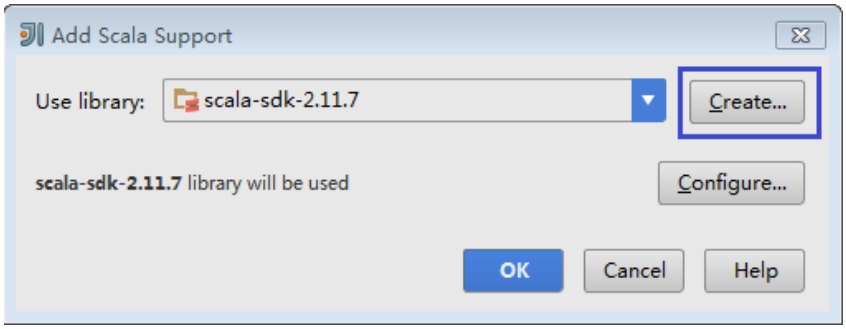
3. 当IDEA可以识别出Scala SDK时，在设置界面，选择编译的依赖jar包，然后单击“OK”应用设置

图 9-23 Add Scala Support



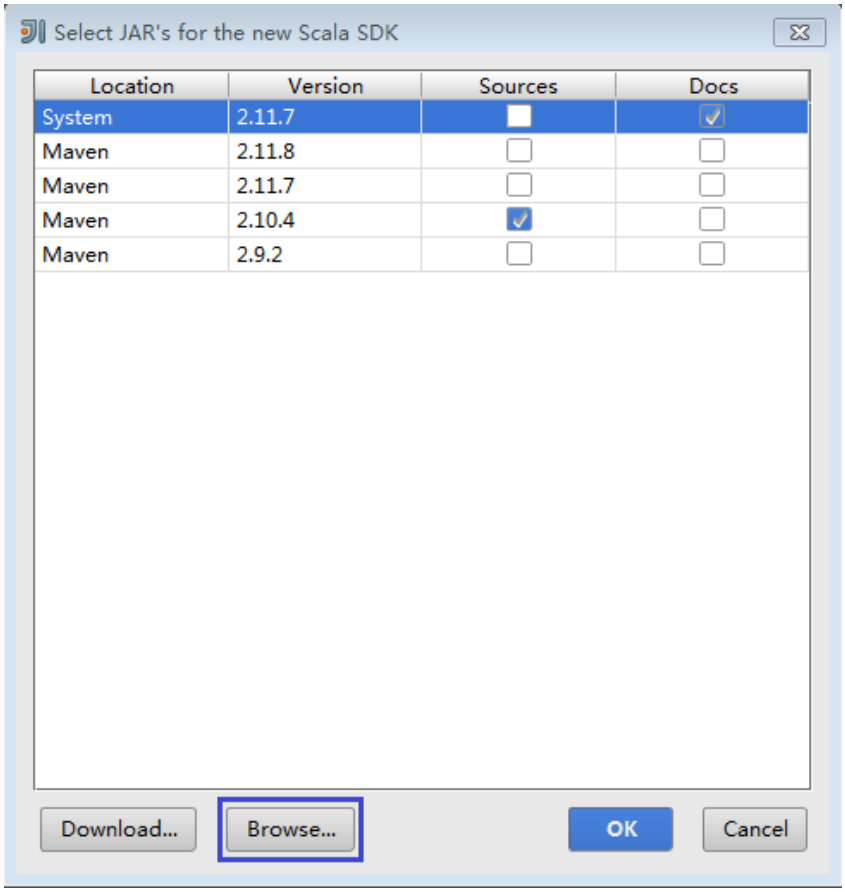
4. 当系统无法识别出Scala SDK时，需要自行创建。
   a. 单击“Create...”。

图 9-24 Create...



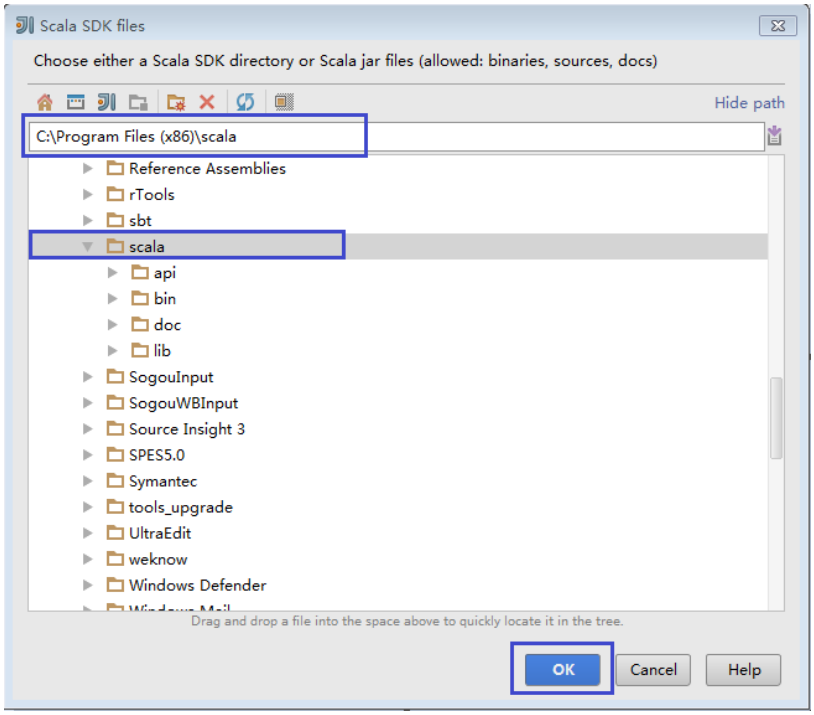
b. 在“Select JAR's for the new Scala SDK”页面单击“Browse...”。

图 9-25 Select JAR's for the new Scala SDK



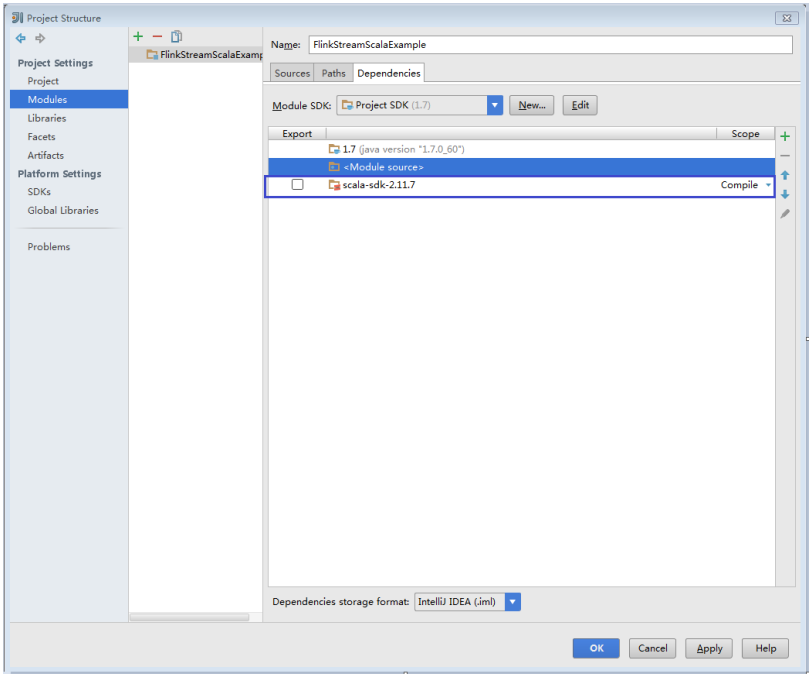
c. 在“Scala SDK files”页面选择scala sdk目录，单击“OK”。

图 9-26 Scala SDK files



5. 设置成功，单击“OK”保存设置。

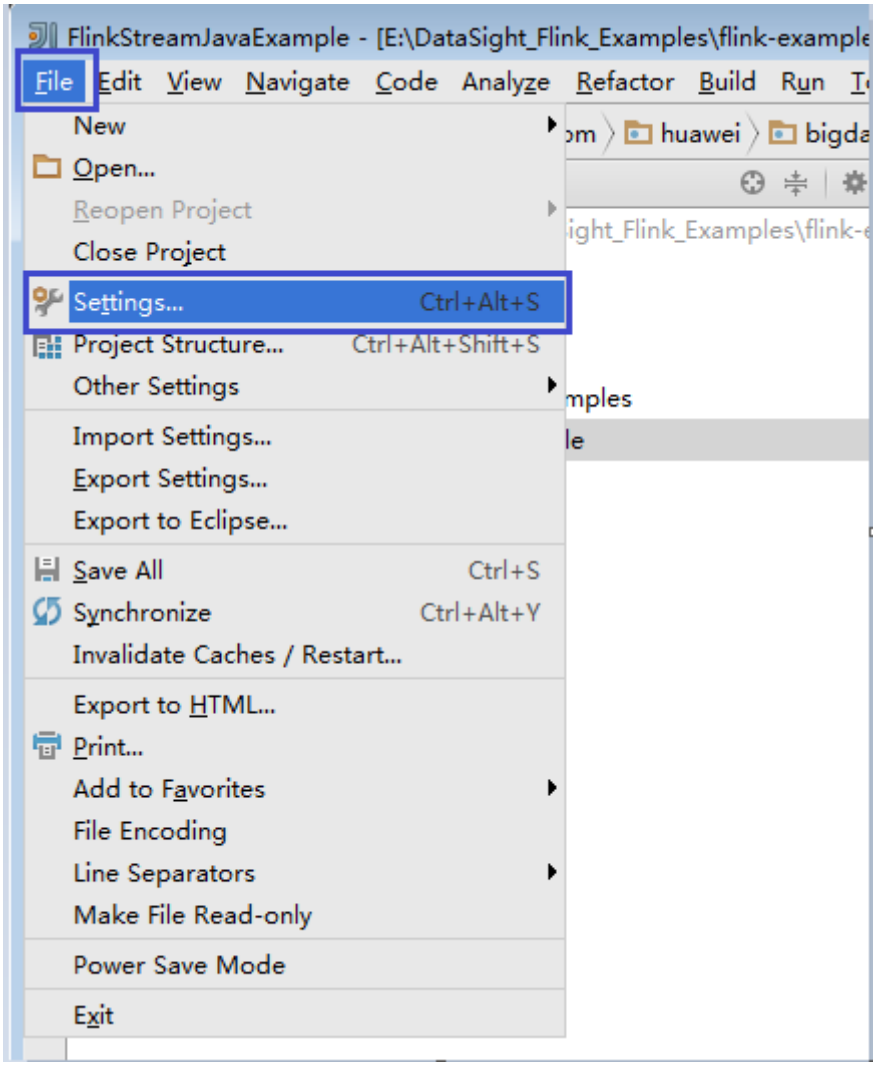
图 9-27 设置成功



步骤7 设置IDEA的文本文件编码格式，解决乱码显示问题。

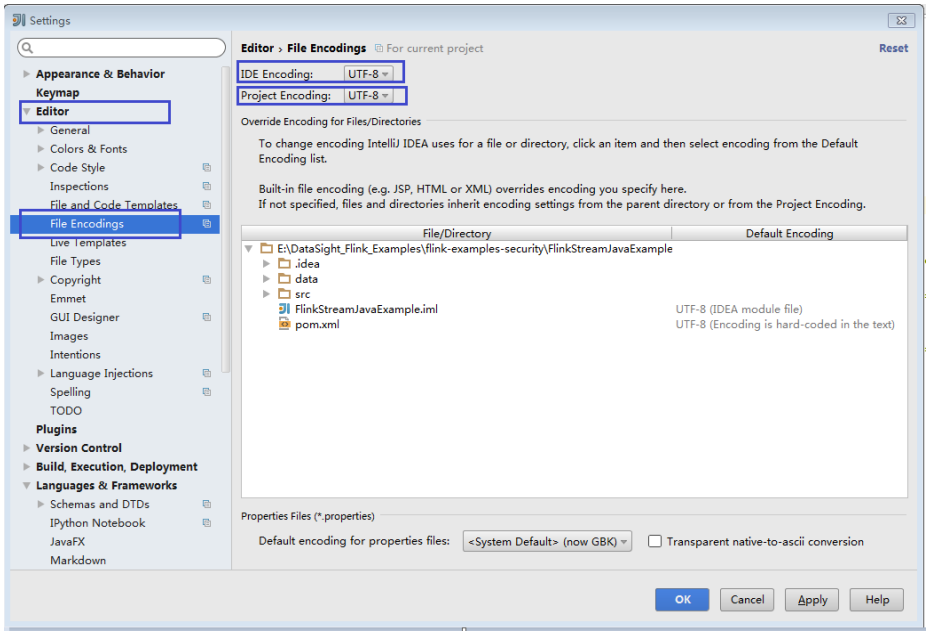
1. 在IDEA首页，选择“File > Settings...”。

图 9-28 选择 Settings



2. 编码配置。
   a. 在“Settings”页面，展开“Editor”，选择“File Encodings”。
   b. 分别在右侧的“IDE Encoding”和“Project Encoding”的下拉框中，选择“UTF-8”。
   c. 单击“Apply”应用配置。
   d. 单击“OK”完成编码配置。

图 9-29 Settings



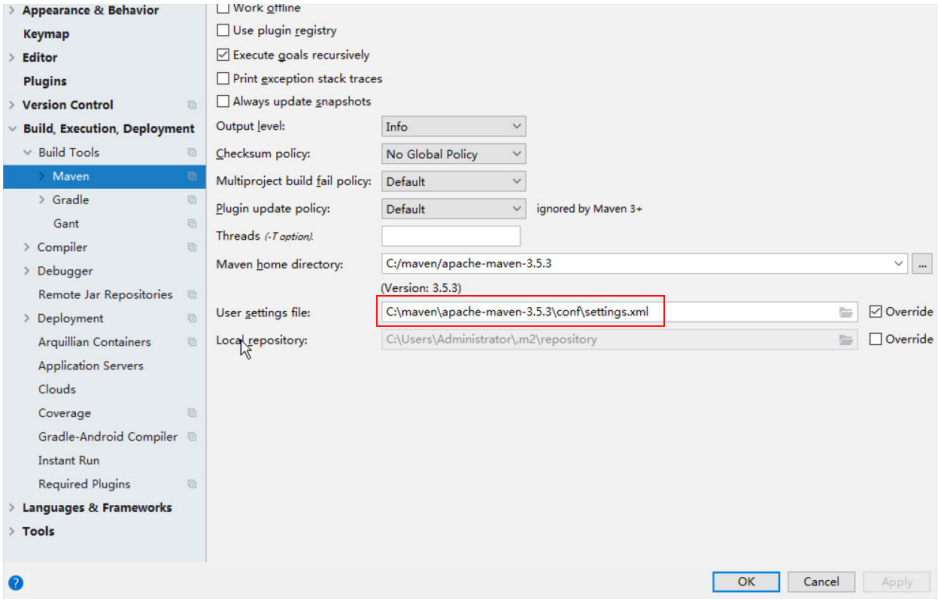
说明

- 请从Flink服务端安装目录获取相关的依赖包。
- 请从Kafka环境中获取Kafka依赖包。
- 具体依赖包请查看**参考信息**。

步骤8 配置Maven。

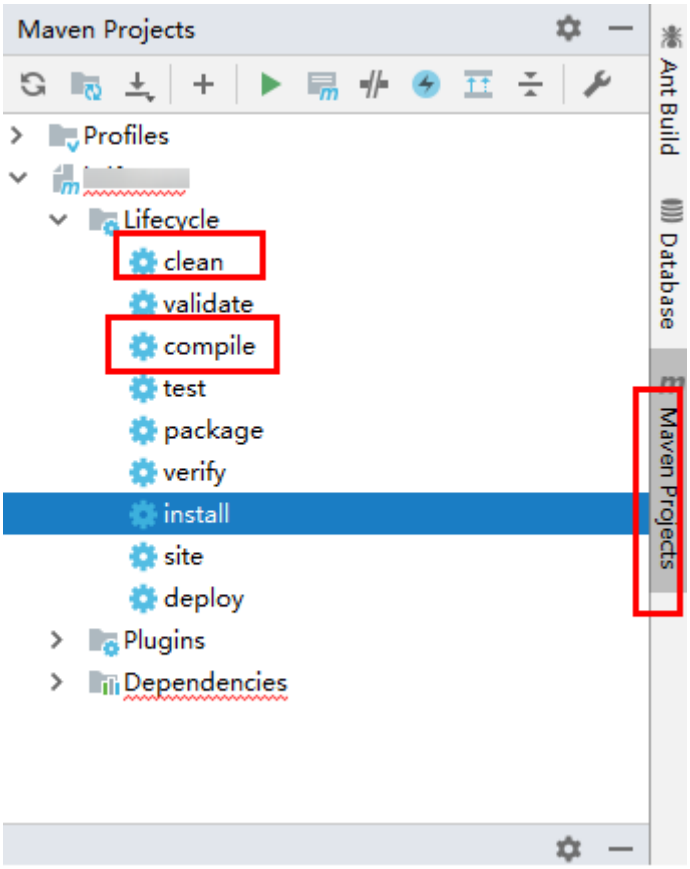
1. 参考**配置华为开源镜像仓**章节描述，增加开源镜像仓地址等配置信息到本地Maven的“setting.xml”配置文件。
2. 修改完成后，在IntelliJ IDEA选择“File > Settings > Build, Execution, Deployment > Build Tools > Maven”，勾选“User settings file”右侧的“Override”，并修改“User settings file”的值为当前“settings.xml”文件放置目录，确保该目录为“*<本地Maven安装目录>*\conf\settings.xml”。

图 9-30 “settings.xml”文件放置目录



3. 单击“Maven home directory”右侧的下拉菜单，选择Maven的安装路径。
4. 单击“Apply”并单击“OK”。
5. 在IntelliJ IDEA主界面右侧，单击“Maven Projects”，在“Maven Projects”界面执行“*项目名称* > Lifecycle”目录下的“clean”和“compile”脚本。

图 9-31 Maven Projects 界面



----结束

参考信息

Flink客户端lib目录、opt目录中都有flink jar包，其中lib目录中默认是flink核心jar包，opt目录中是对接外部组件的jar包（例如flink-connector-kafka*.jar），若应用开发中需要请手动复制相关jar包到lib目录中。

针对Flink提供的几个样例工程，其对应的运行依赖包如下：

表 9-6 样例工程依赖包

样例工程	依赖包	说明
DataStream程序样例工程（Java/Scala）	flink-dist_*.jar	在Flink的客户端或者服务端安装路径的lib目录下获取。
异步Checkpoint机制程序样例工程（Java/Scala）		
向Kafka生产并消费数据程序样例工程（Java/Scala）	kafka-clients-*.jar	由Kafka组件发布提供，可在Kafka组件客户端或者服务端安装路径下的lib目录下获取。

样例工程	依赖包	说明
	flink-connector-kafka_*.jar	在Flink客户端或者服务端安装路径的opt目录下获取。
pipeline程序样例工程（Java/Scala）	flink-dist_*.jar	在Flink的客户端或者服务端安装路径的lib目录下获取。
	flink-connector-netty_*.jar	在二次开发样例代码编译后产生的lib文件夹下获取。
	curator-client-2.12.0.jar	
	curator-framework-2.12.0.jar	
Stream SQL Join样例工程（Java）	flink-dist_2.11*.jar	在Flink的客户端或者服务端安装路径的lib目录下获取。
	kafka-clients-*.jar	由Kafka组件发布提供，可在Kafka组件客户端或者服务端安装路径下的lib目录下获取。
	flink-connector-kafka_2.11*.jar	在Flink客户端或者服务端安装路径的opt目录下获取。
	flink-table_2.11*.jar	-

9.3.4 准备 Flink 安全认证

场景说明

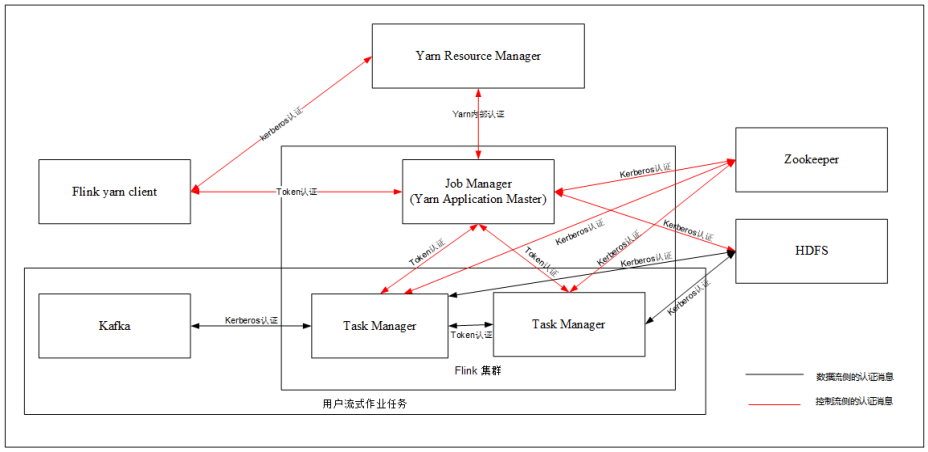
在安全集群环境下，各个组件之间的相互通信不能够简单的互通，而需要在通信之前进行相互认证，以确保通信的安全性。

用户在提交Flink应用程序时，需要与Yarn、HDFS等之间进行通信。那么提交Flink的应用程序中需要设置安全认证，确保Flink程序能够正常运行。

当前Flink系统支持认证和加密传输，要使用认证和加密传输，用户需要做如下准备：

安全认证

图 9-32 Flink 系统认证方式



Flink整个系统存在三种认证方式，需参考下表进行配置：

- 使用kerberos认证：Flink yarn client与Yarn Resource Manager、JobManager与Zookeeper、JobManager与HDFS、TaskManager与HDFS、Kafka与TaskManager、TaskManager和Zookeeper。
- 使用security cookie进行认证：Flink yarn client与Job Manager、JobManager与TaskManager、TaskManager与TaskManager。
- 使用YARN内部的认证机制：Yarn Resource Manager与Application Master（简称AM）。

  说明

  - Flink的JobManager与YARN的AM是在同一个进程下。
  - 如果用户安装安全模式需要使用kerberos认证和security cookie认证。

表 9-7 安全认证方式

安全认证方式	配置方法
Kerberos认证（当前只支持keytab认证方式）	1. 从FusionInsight Manager上下载**准备集群认证用户信息**创建的用户keytab文件，并放置到Flink客户端所在节点的某个目录下。 2. 在“*客户端安装路径*/Flink/flink/conf/flink-conf.yaml”上配置： a. 配置客户端安装节点的业务IP和Master节点IP到“jobmanager.web.access-control-allow-origin”和“jobmanager.web.allow-access-address”配置项中。 jobmanager.web.access-control-allow-origin: *xx.xx.xxx.xxx,xx.xx.xxx.xxx,xx.xx.xxx.xxx* jobmanager.web.allow-access-address: *xx.xx.xxx.xxx,xx.xx.xxx.xxx,xx.xx.xxx.xxx* **说明** 集群外节点业务IP为安装客户端所在的弹性云服务器的IP。集群内节点业务IP获取方式如下： 登录MapReduce服务管理控制台，选择“现有集群”，选中当前的集群并单击集群名，进入集群信息页面。在“节点管理”中查看安装客户端所在的节点IP。 b. keytab路径。 security.kerberos.login.keytab: /home/flinkuser/keytab/flinkuser.keytab **说明** “/home/flinkuser/keytab/”表示的是用户保存keytab文件的目录。 c. principal名为用于运行作业的用户名。 security.kerberos.login.principal: flinkuser d. 对于HA模式，如果配置了ZooKeeper，还需要设置ZK kerberos认证相关的配置。 zookeeper.sasl.disable: false security.kerberos.login.contexts: Client e. 如果用户对于Kafka client和Kafka broker之间也需要做kerberos认证，配置如下： security.kerberos.login.contexts: Client,KafkaClient

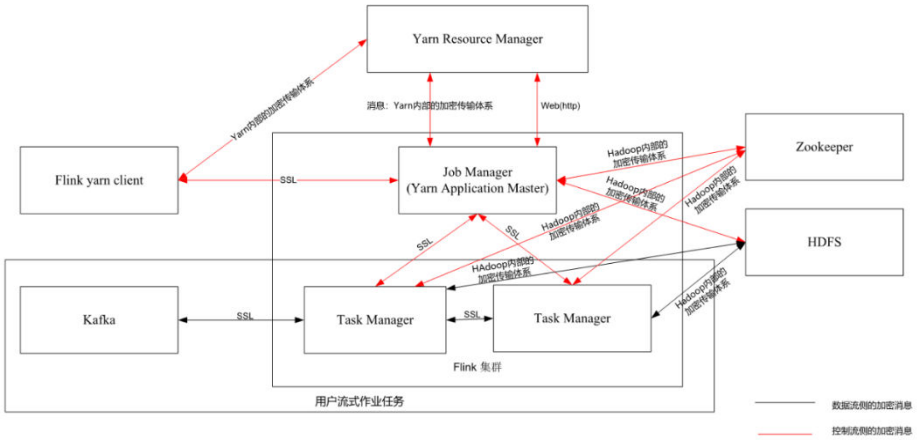
安全认证方式	配置方法
Security Cookie认证	1. 将“generate_keystore.sh”脚本放置到Flink客户端“bin”目录中并调用“generate_keystore.sh”脚本，生成“Security Cookie”、“flink.keystore”文件和“flink.truststore”文件。具体操作可参考**认证和加密**。执行**sh generate_keystore.sh**，输入用户自定义密码。密码不允许包含#。 **说明** 执行脚本后，在Flink客户端的“conf”目录下生成“flink.keystore”和“flink.truststore”文件，并且在客户端配置文件“flink-conf.yaml”中将以下配置项进行了默认赋值。 • 将配置项“security.ssl.keystore”设置为“flink.keystore”文件所在绝对路径。 • 将配置项“security.ssl.truststore”设置为“flink.truststore”文件所在的绝对路径。 • 将配置项“security.cookie”设置为“generate_keystore.sh”脚本自动生成的一串随机规则密码。 • 默认“flink-conf.yaml”中“security.ssl.encrypt.enabled: false”，“generate_keystore.sh”脚本将配置项“security.ssl.key-password”、“security.ssl.keystore-password”和“security.ssl.truststore-password”的值设置为调用“generate_keystore.sh”脚本时输入的密码。 • MRS 3.x及之后版本，如果需要使用密文时，设置“flink-conf.yaml”中“security.ssl.encrypt.enabled: true”，“generate_keystore.sh”脚本不会配置“security.ssl.key-password”、“security.ssl.keystore-password”和“security.ssl.truststore-password”的值，需要使用Manager明文加密API进行获取，执行**curl -k -i -u** *user name*:*password* **-X POST -HContent-type:application/json -d '{"plainText":"***password***"}' 'https://***x.x.x.x***:28443/web/api/v2/tools/encrypt'** 其中*user name*:password分别为当前系统登录用户名和密码；**"plainText"**的password为调用“generate_keystore.sh”脚本时的密码；x.x.x.x为集群Manager的浮动IP。 2. 打开“Security Cookie”开关，配置“security.enable: true”，查看“security cookie”是否已配置成功，例如： `security.cookie: ae70acc9-9795-4c48-ad35-8b5adc8071744f605d1d-2726-432e-88ae-dd39bfec40a9`
YARN内部认证方式	该方式是YARN内部的认证方式，不需要用户配置。

说明

当前一个Flink集群只支持一个用户，一个用户可以创建多个Flink集群。

加密传输

图 9-33 Flink 系统加密传输方式



Flink整个系统存在三种加密传输方式：

- 使用Yarn内部的加密传输方式：Flink yarn client与Yarn Resource Manager、Yarn Resource Manager与Job Manager。
- SSL：Flink yarn client与JobManager、JobManager与TaskManager、TaskManager与TaskManager。
- 使用Hadoop内部的加密传输方式：JobManager和HDFS、TaskManager和HDFS、JobManager与ZooKeeper、TaskManager与ZooKeeper。

说明

Yarn内部和Hadoop内部都不需要用户配置加密，用户只需要配置SSL加密传输方式。

配置SSL传输，用户主要在客户端的“flink-conf.yaml”文件中做如下配置：

1. 打开SSL开关和设置SSL加密算法，配置参数如表9-8所示，请根据实际情况修改对应参数值。

表 9-8 参数描述

参数	参数值示例	描述
security.ssl.enabled	true	打开SSL总开关
akka.ssl.enabled	true	打开akka SSL开关
blob.service.ssl.enabled	true	打开blob通道SSL开关
taskmanager.data.ssl.enabled	true	打开taskmanager之间通信的SSL开关

参数	参数值示例	描述
security.ssl.algorithms	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_DHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	设置SSL加密的算法

说明

如果打开Task Manager之间data传输通道的SSL，对性能会有较大影响，需要用户从安全和性能综合考虑。

2. 在Flink客户端的bin目录下，执行命令***sh generate_keystore.sh <password>***，具体操作可参考**认证和加密**，**表9-9**中的配置项会被默认赋值，用户也可以手动配置。

表 9-9 参数描述

参数	参数值示例	描述
security.ssl.keystore	${path}/flink.keystore	keystore的存放路径，“flink.keystore”表示用户通过generate_keystore.sh*工具生成的keystore文件名称。
security.ssl.keystore-password	-	keystore的password，表示需要用户输入自定义设置的密码值。
security.ssl.key-password	-	ssl key的password，表示需要用户输入自定义设置的密码值。
security.ssl.truststore	${path}/flink.truststore	truststore存放路径，“flink.truststore”表示用户通过generate_keystore.sh*工具生成的truststore文件名称。
security.ssl.truststore-password	-	truststore的password，表示需要用户输入自定义设置的密码值。

说明

“path”目录是用来存放SSL keystore、truststore相关配置文件，该目录是由用户自定义创建。相对路径和绝对路径的不同导致执行命令存在差异，在3和4详细说明。

3. 配置keystore或truststore文件路径为相对路径时，Flink Client执行命令的目录需要可以直接访问该相对路径。Flink有两种执行方式来传输keystore和truststore文件。
   - 在Flink的CLI yarn-session.sh命令中增加“-t”选项来传输keystore和truststore文件到各个执行节点。例如：

     cd /opt/client/Flink/flink

     ./bin/yarn-session.sh -t ssl/
   - 在Flink run命令中增加“-yt”选项来传输keystore和truststore文件到各个执行节点。例如：

     ./bin/flink run -yt ssl/ -ys 3 -m yarn-cluster -c com.huawei.SocketWindowWordCount ../lib/flink-eg-1.0.jar --hostname r3-d3 --port 9000

     说明

     - 示例中的“ssl/”是Flink客户端目录下自定义的子目录，用来存放SSL keystore、truststore相关配置文件。
     - Flink客户端执行命令的当前路径需要能访问到“ssl/”相对路径。
4. 配置keystore或truststore文件路径为绝对路径时，需要在Flink Client以及Yarn各个节点的该绝对路径上放置keystore或truststore文件。

   Flink有两种方式执行应用程序，且执行命令中不需要使用“-t”或“-yt”来传输keystore和truststore文件。
   - 使用Flink的CLI yarn-session.sh命令执行应用程序。如：

     ./bin/yarn-session.sh
   - 使用Flink run命令执行应用程序。如：

     ./bin/flink run -ys 3 -m yarn-cluster -c com.huawei.SocketWindowWordCount ../lib/flink-eg-1.0.jar --hostname r3-d3 --port 9000

9.4 开发 Flink 应用

9.4.1 Flink DataStream 样例程序

9.4.1.1 Flink DataStream 样例程序开发思路

场景说明

假定用户有某个网站周末网民网购停留时间的日志文本，基于某些业务要求，要求开发Flink的DataStream应用程序实现如下功能：

说明

DataStream应用程序可以在Windows环境和Linux环境中运行。

- 实时统计总计网购时间超过2个小时的女性网民信息。
- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

  log1.txt：周六网民停留日志。该日志文件在该样例程序中的data目录下获取。

```
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

  log2.txt：周日网民停留日志。该日志文件在该样例程序中的data目录下获取。

```
LiuYang,female,20
YuanJing,male,10
CaiXuyu,female,50
FangBo,female,50
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
CaiXuyu,female,50
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
FangBo,female,50
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

数据规划

DataStream样例工程的数据存储在文本中。

将log1.txt和log2.txt放置在指定路径下，例如"/opt/log1.txt"和"/opt/log2.txt"。

说明

- 数据文件若存放在本地文件系统，需在所有部署Yarn NodeManager的节点指定目录放置，并设置运行用户访问权限。
- 若将数据文件放置于HDFS，需指定程序中读取文件路径HDFS路径，例如"hdfs://hacluster/path/to/file"。

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

1. 读取文本数据，生成相应DataStream，解析数据生成UserRecord信息。
2. 筛选女性网民上网时间数据信息。
3. 按照姓名、性别进行keyby操作，并汇总在一个时间窗口内每个女性上网时间。
4. 筛选连续上网时间超过阈值的用户，并获取结果。

9.4.1.2 Flink DataStream 样例程序（Java）

功能介绍

统计连续网购时间超过2个小时的女性网民信息，将统计结果直接打印。

代码样例

下面代码片段仅为演示，完整代码参见FlinkStreamJavaExample样例工程下的com.huawei.bigdata.flink.examples.FlinkStreamJavaExample：

```
    // 参数解析:
    // <filePath>为文本读取路径，用逗号分隔。
    // <windowTime>为统计数据的窗口跨度,时间单位都是分。
public class FlinkStreamJavaExample {
    public static void main(String[] args) throws Exception {
        // 打印出执行flink run的参考命令
        System.out.println("use command as: ");
        System.out.println("./bin/flink run --class
com.huawei.bigdata.flink.examples.FlinkStreamJavaExample /opt/test.jar --filePath /opt/log1.txt,/opt/
log2.txt --windowTime 2");
        System.out.println("*******************************************************************************************");
        System.out.println("<filePath> is for text file to read data, use comma to separate");
        System.out.println("<windowTime> is the width of the window, time as minutes");
        System.out.println("*******************************************************************************************");

        // 读取文本路径信息，并使用逗号分隔
        final String[] filePaths = ParameterTool.fromArgs(args).get("filePath", "/opt/log1.txt,/opt/
log2.txt").split(",");
        assert filePaths.length > 0;

        // windowTime设置窗口时间大小，默认2分钟一个窗口足够读取文本内的所有数据了
        final int windowTime = ParameterTool.fromArgs(args).getInt("windowTime", 2);

        // 构造执行环境，使用eventTime处理窗口数据
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        env.setParallelism(1);

        // 读取文本数据流
        DataStream<String> unionStream = env.readTextFile(filePaths[0]);
        if (filePaths.length > 1) {
            for (int i = 1; i < filePaths.length; i++) {
                unionStream = unionStream.union(env.readTextFile(filePaths[i]));
            }
        }

        // 数据转换，构造整个数据处理的逻辑，计算并得出结果打印出来
        unionStream.map(new MapFunction<String, UserRecord>() {
            @Override
            public UserRecord map(String value) throws Exception {
                return getRecord(value);
            }
        }).assignTimestampsAndWatermarks(
                new Record2TimestampExtractor()
        ).filter(new FilterFunction<UserRecord>() {
            @Override
            public boolean filter(UserRecord value) throws Exception {
                return value.sexy.equals("female");
            }
        }).keyBy(
            new UserRecordSelector()
        ).window(
            TumblingEventTimeWindows.of(Time.minutes(windowTime))
        ).reduce(new ReduceFunction<UserRecord>() {
            @Override
            public UserRecord reduce(UserRecord value1, UserRecord value2)
```

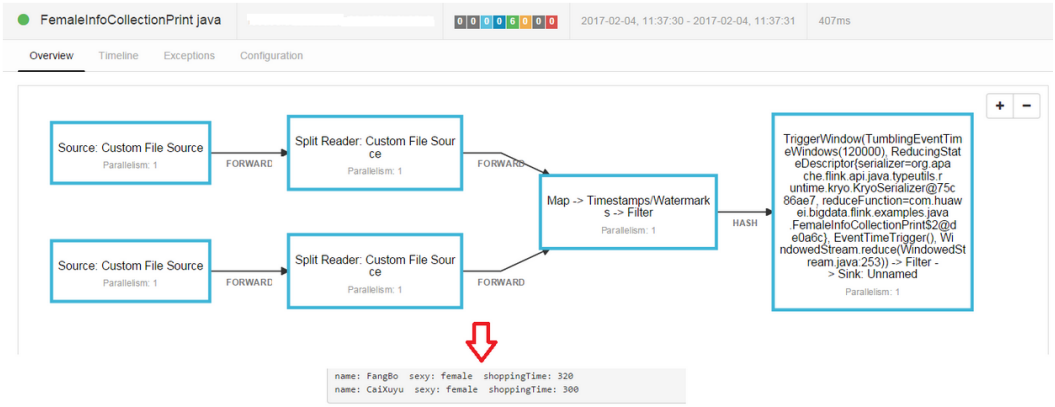
```
                throws Exception {
            value1.shoppingTime += value2.shoppingTime;
            return value1;
        }
    }).filter(new FilterFunction<UserRecord>() {
        @Override
        public boolean filter(UserRecord value) throws Exception {
            return value.shoppingTime > 120;
        }
    }).print();

    // 调用execute触发执行
    env.execute("FemaleInfoCollectionPrint java");
}

// 构造keyBy的关键字作为分组依据
private static class UserRecordSelector implements KeySelector<UserRecord, Tuple2<String, String>> {
    @Override
    public Tuple2<String, String> getKey(UserRecord value) throws Exception {
        return Tuple2.of(value.name, value.sexy);
    }
}

// 解析文本行数据，构造UserRecord数据结构
private static UserRecord getRecord(String line) {
    String[] elems = line.split(",");
    assert elems.length == 3;
    return new UserRecord(elems[0], elems[1], Integer.parseInt(elems[2]));
}

// UserRecord数据结构的定义，并重写了toString打印方法
public static class UserRecord {
    private String name;
    private String sexy;
    private int shoppingTime;

    public UserRecord(String n, String s, int t) {
        name = n;
        sexy = s;
        shoppingTime = t;
    }

    public String toString() {
        return "name: " + name + "  sexy: " + sexy + "  shoppingTime: " + shoppingTime;
    }
}

// 构造继承AssignerWithPunctuatedWatermarks的类，用于设置eventTime以及waterMark
private static class Record2TimestampExtractor implements
AssignerWithPunctuatedWatermarks<UserRecord> {

    // add tag in the data of datastream elements
    @Override
    public long extractTimestamp(UserRecord element, long previousTimestamp) {
        return System.currentTimeMillis();
    }

    // give the watermark to trigger the window to execute, and use the value to check if the window
elements is ready
    @Override
    public Watermark checkAndGetNextWatermark(UserRecord element, long extractedTimestamp) {
        return new Watermark(extractedTimestamp - 1);
    }
}
}
```

执行之后打印结果如下所示：

```
name: FangBo  sexy: female  shoppingTime: 320
name: CaiXuyu  sexy: female  shoppingTime: 300
```

执行如图9-34所示。

图 9-34 显示图



9.4.1.3 Flink DataStream 样例程序（Scala）

功能介绍

实时统计连续网购时间超过2个小时的女性网民信息，将统计结果直接打印出来。

代码样例

下面代码片段仅为演示，完整代码参见FlinkStreamScalaExample样例工程下的com.huawei.bigdata.flink.examples.FlinkStreamScalaExample：

```
  // 参数解析:
  // filePath为文本读取路径，用逗号分隔。
  // windowTime;为统计数据的窗口跨度,时间单位都是分。
  object FlinkStreamScalaExample {
 def main(args: Array[String]) {
  // 打印出执行flink run的参考命令
  System.out.println("use command as: ")
  System.out.println("./bin/flink run --class
com.huawei.bigdata.flink.examples.FlinkStreamScalaExample /opt/test.jar --filePath /opt/log1.txt,/opt/
log2.txt --windowTime 2")
  System.out.println("*******************************************************************************************")
  System.out.println("<filePath> is for text file to read data, use comma to separate")
  System.out.println("<windowTime> is the width of the window, time as minutes")
  System.out.println("*******************************************************************************************")

  // 读取文本路径信息，并使用逗号分隔
  val filePaths = ParameterTool.fromArgs(args).get("filePath",
    "/opt/log1.txt,/opt/log2.txt").split(",").map(_.trim)
  assert(filePaths.length > 0)

  // windowTime设置窗口时间大小，默认2分钟一个窗口足够读取文本内的所有数据了
  val windowTime = ParameterTool.fromArgs(args).getInt("windowTime", 2)

  // 构造执行环境，使用eventTime处理窗口数据
  val env = StreamExecutionEnvironment.getExecutionEnvironment
  env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
  env.setParallelism(1)

  // 读取文本数据流
  val unionStream = if (filePaths.length > 1) {
    val firstStream = env.readTextFile(filePaths.apply(0))
    firstStream.union(filePaths.drop(1).map(it => env.readTextFile(it)): _*)
  } else {
```

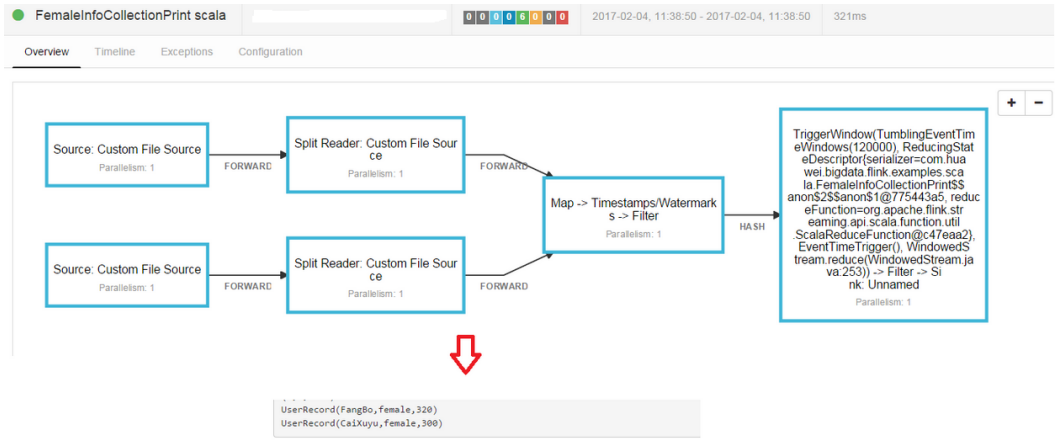
```
    env.readTextFile(filePaths.apply(0))
  }

  // 数据转换，构造整个数据处理的逻辑，计算并得出结果打印出来
  unionStream.map(getRecord(_))
    .assignTimestampsAndWatermarks(new Record2TimestampExtractor)
    .filter(_.sexy == "female")
    .keyBy("name", "sexy")
    .window(TumblingEventTimeWindows.of(Time.minutes(windowTime)))
    .reduce((e1, e2) => UserRecord(e1.name, e1.sexy, e1.shoppingTime + e2.shoppingTime))
    .filter(_.shoppingTime > 120).print()

  // 调用execute触发执行
  env.execute("FemaleInfoCollectionPrint scala")
}

// 解析文本行数据，构造UserRecord数据结构
def getRecord(line: String): UserRecord = {
  val elems = line.split(",")
  assert(elems.length == 3)
  val name = elems(0)
  val sexy = elems(1)
  val time = elems(2).toInt
  UserRecord(name, sexy, time)
}

// UserRecord数据结构的定义
case class UserRecord(name: String, sexy: String, shoppingTime: Int)


// 构造继承AssignerWithPunctuatedWatermarks的类，用于设置eventTime以及waterMark
private class Record2TimestampExtractor extends AssignerWithPunctuatedWatermarks[UserRecord] {

  // add tag in the data of datastream elements
  override def extractTimestamp(element: UserRecord, previousTimestamp: Long): Long = {
    System.currentTimeMillis()
  }

  // give the watermark to trigger the window to execute, and use the value to check if the window
elements is ready
  def checkAndGetNextWatermark(lastElement: UserRecord,
                               extractedTimestamp: Long): Watermark = {
    new Watermark(extractedTimestamp - 1)
  }
}
}
```

执行之后打印结果如下所示：

```
UserRecord(FangBo,female,320)
UserRecord(CaiXuyu,female,300)
```

执行如图9-35所示。

图 9-35 显示图



9.4.2 Flink Kafka 样例程序

9.4.2.1 Flink Kafka 样例程序开发思路

场景说明

假定某个Flink业务每秒就会收到1个消息记录。

基于某些业务要求，开发的Flink应用程序实现功能：实时输出带有前缀的消息内容。

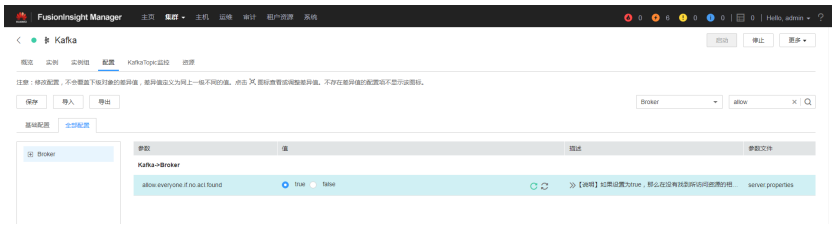
数据规划

Flink样例工程的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户），并从Kafka组件接收数据。

1. 确保集群安装完成，包括HDFS、Yarn、Flink和Kafka。
2. 创建Topic。
   a. 在服务端配置用户创建topic的权限。

      将Kafka的Broker配置参数“allow.everyone.if.no.acl.found”的值修改为“true”，如图9-36所示。配置完后重启Kafka服务。

      图 9-36 配置用户创建 topic 的权限

   b. 用户使用Linux命令行创建topic，执行命令前需要使用kinit命令进行人机认证，如：**kinit** *flinkuser*。

      说明

      flinkuser需要用户自己创建，并拥有创建Kafka的topic权限。具体操作请参考**准备MRS应用开发用户**章节。

创建topic的命令格式：

bin/kafka-topics.sh --create --zookeeper {*zkQuorum*}/kafka --partitions {*partitionNum*} --replication-factor {*replicationNum*} --topic {*Topic*}

表 9-10 参数说明

参数名	说明
{zkQuorum}	ZooKeeper集群信息，格式为IP:port。
{partitionNum}	topic的分区数。
{replicationNum}	topic中每个partition数据的副本数。
{Topic}	Topic名称。

示例：在Kafka的客户端路径下执行命令，此处以ZooKeeper集群的IP:port是10.96.101.32:2181,10.96.101.251:2181,10.96.101.177:2181,10.91.8.160:2181，Topic名称为topic1的数据为例。

```
bin/kafka-topics.sh --create --zookeeper 10.96.101.32:2181,10.96.101.251:2181,10.96.101.177:2181,10.91.8.160:2181/kafka --partitions 5 --replication-factor 1 --topic topic1
```

3. 安全认证。

   安全认证的方式有三种：Kerberos认证、SSL加密认证和Kerberos+SSL模式认证，用户在使用的时候可任选其中一种方式进行认证。

   - **Kerberos认证配置**

     i. 客户端配置。

     在Flink配置文件“flink-conf.yaml”中，增加kerberos认证相关配置（主要在“contexts”项中增加“KafkaClient”），示例如下：

     ```
     security.kerberos.login.keytab: /home/demo/flink/release/flink-x.x.x/keytab/user.keytab
     security.kerberos.login.principal: flinkuser
     security.kerberos.login.contexts: Client,KafkaClient
     security.kerberos.login.use-ticket-cache: false
     ```

     ii. 运行参数。

     关于“SASL_PLAINTEXT”协议的运行参数示例如下：

     ```
     --topic topic1 --bootstrap.servers 10.96.101.32:21007 --security.protocol SASL_PLAINTEXT --sasl.kerberos.service.name kafka --kerberos.domain.name hadoop.系统域名.com
     ```

     说明

     - 10.96.101.32:21007：kafka服务器的IP和端口。
     - 系统域名：用户可登录FusionInsight Manager，单击“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。

   - **SSL加密配置**

     - 服务端配置。

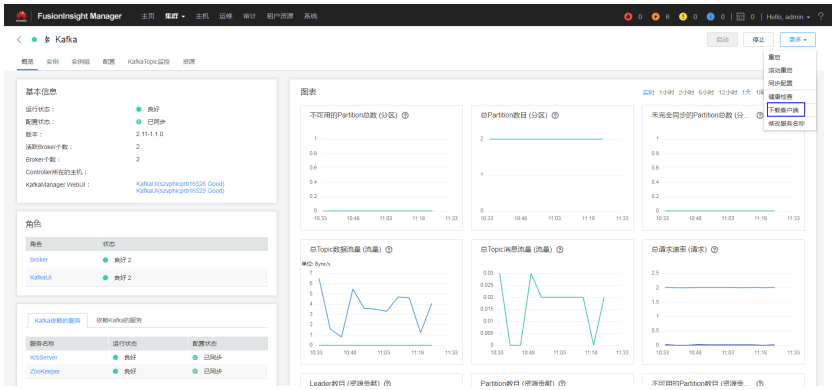
       配置“ssl.mode.enable”为“true”，如图9-37所示：

图 9-37 服务端配置

- 客户端配置。

  1) 登录FusionInsight Manager系统，选择“集群 > *待操作集群的名称* > 服务 > Kafka > 更多 > 下载客户端”，下载客户端压缩文件到本地机器。如图9-38所示：

  图 9-38 客户端配置

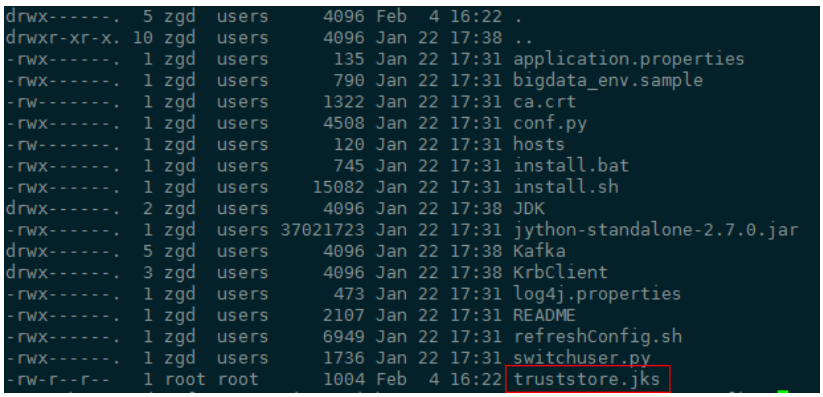
  2) 使用客户端根目录中的“ca.crt”证书文件生成客户端的“truststore”。

  执行命令如下：

  ```
  keytool -noprompt -import -alias myservercert -file ca.crt -keystore truststore.jks
  ```

  命令执行结果查看：

  图 9-39 执行结果

  3) 运行参数。

  “ssl.truststore.password”参数内容需要跟创建“truststore”时输入的密码保持一致，执行以下命令运行参数。

  ```
  --topic topic1 --bootstrap.servers 10.96.101.32:9093 --security.protocol SSL --
  ssl.truststore.location /home/zgd/software/FusionInsight_XXX_Kafka_ClientConfig/
  ```

truststore.jks --ssl.truststore.password *xxx* //10.96.101.32:9093表示kafka服务器的IP:port，XXX表示FusionInsight相应的版本号，xxx表示密码。

- **Kerberos+SSL模式配置**

  完成上文中Kerberos和SSL各自的服务端和客户端配置后，只需要修改运行参数中的端口号和协议类型即可启动Kerberos+SSL模式。

  --topic *topic1* --bootstrap.servers *10.96.101.32:21009* --security.protocol SASL_SSL --sasl.kerberos.service.name kafka --ssl.truststore.location --kerberos.domain.name hadoop.*系统域名*.com /home/zgd/software/FusionInsight_*XXX*_Kafka_ClientConfig/truststore.jks --ssl.truststore.password *xxx* //10.96.101.32:21009表示kafka服务器的IP:port，XXX表示FusionInsight相应的版本号，xxx表示密码。

开发思路

1. 启动Flink Kafka Producer应用向Kafka发送数据。
2. 启动Flink Kafka Consumer应用从Kafka接收数据，保证topic与producer一致。
3. 在数据内容中增加前缀并进行打印。

9.4.2.2 Flink Kafka 样例程序（Java）

功能介绍

在Flink应用中，调用flink-connector-kafka模块的接口，生产并消费数据。

代码样例

用户在开发前需要使用对接安全模式的Kafka，则需要引入FusionInsight的kafka-clients-*.jar，该jar包可在Kafka的客户端目录下获取。

下面代码片段仅为演示，完整代码参见FlinkKafkaJavaExample样例工程下的com.huawei.bigdata.flink.examples.WriteIntoKafka和com.huawei.bigdata.flink.examples.ReadFromKafka。

```
//producer代码
public class WriteIntoKafka {

    public static void main(String[] args) throws Exception {
    // 打印出执行flink run的参考命令

      System.out.println("use command as: ");

      System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +

          " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092");

      System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +

          " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21007 --security.protocol SASL_PLAINTEXT --sasl.kerberos.service.name kafka");

      System.out.println("******************************************************************************************");

      System.out.println("<topic> is the kafka topic name");

      System.out.println("<bootstrap.servers> is the ip:port list of brokers");

      System.out.println("******************************************************************************************");

      // 构造执行环境
      StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
      // 设置并发度
```



```
    env.setParallelism(1);
    // 解析运行参数
    ParameterTool paraTool = ParameterTool.fromArgs(args);
    // 构造流图，将自定义Source生成的数据写入Kafka
    DataStream<String> messageStream = env.addSource(new SimpleStringGenerator());

    messageStream.addSink(new FlinkKafkaProducer<>(paraTool.get("topic"),

        new SimpleStringSchema(),

        paraTool.getProperties()));
      // 调用execute触发执行
    env.execute();

  }


 // 自定义Source，每隔1s持续产生消息
  public static class SimpleStringGenerator implements SourceFunction<String> {

    private static final long serialVersionUID = 2174904787118597072L;

    boolean running = true;

    long i = 0;



    @Override

    public void run(SourceContext<String> ctx) throws Exception {

      while (running) {

        ctx.collect("element-" + (i++));

        Thread.sleep(1000);

      }

    }



    @Override

    public void cancel() {

      running = false;

    }

  }

}

//consumer代码
public class ReadFromKafka {

    public static void main(String[] args) throws Exception {
    // 打印出执行flink run的参考命令
      System.out.println("use command as: ");

      System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka" +

          " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092");

      System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka" +
```

```
        " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21007 --security.protocol
SASL_PLAINTEXT --sasl.kerberos.service.name kafka");


      System.out.println
("*************************************************************************************************");

      System.out.println("<topic> is the kafka topic name");

      System.out.println("<bootstrap.servers> is the ip:port list of brokers");

      System.out.println
("*************************************************************************************************");
      // 构造执行环境
      StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
      // 设置并发度
      env.setParallelism(1);
      // 解析运行参数
      ParameterTool paraTool = ParameterTool.fromArgs(args);
      // 构造流图，从Kafka读取数据并换行打印
      DataStream<String> messageStream = env.addSource(new
FlinkKafkaConsumer<>(paraTool.get("topic"),

          new SimpleStringSchema(),

          paraTool.getProperties()));

      messageStream.rebalance().map(new MapFunction<String, String>() {

        @Override

        public String map(String s) throws Exception {

          return "Flink says " + s + System.getProperty("line.separator");

        }

      }).print();
      // 调用execute触发执行
      env.execute();

    }

  }
```

9.4.2.3 Flink Kafka 样例程序（Scala）

功能介绍

在Flink应用中，调用flink-connector-kafka模块的接口，生产并消费数据。

代码样例

用户在开发前需要使用对接安全模式的Kafka，则需要引入FusionInsight的kafka-clients-*.jar，该jar包可在kafka客户端目录下获取。

下面代码片段仅为演示，完整代码参见FlinkKafkaScalaExample样例工程下的com.huawei.bigdata.flink.examples.WriteIntoKafka和com.huawei.bigdata.flink.examples.ReadFromKafka。

```
//producer代码
object WriteIntoKafka {

    def main(args: Array[String]) {
    // 打印出执行flink run的参考命令
```

```
    System.out.println("use command as: ")

    System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +

      " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092")

    System.out.println
("*****************************************************************************************")

    System.out.println("<topic> is the kafka topic name")

    System.out.println("<bootstrap.servers> is the ip:port list of brokers")

    System.out.println
("*****************************************************************************************")
    // 构造执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 设置并发度
    env.setParallelism(1)
    // 解析运行参数
    val paraTool = ParameterTool.fromArgs(args)
    // 构造流图，将自定义Source生成的数据写入Kafka
    val messageStream: DataStream[String] = env.addSource(new SimpleStringGenerator)

    messageStream.addSink(new FlinkKafkaProducer(

      paraTool.get("topic"), new SimpleStringSchema, paraTool.getProperties))
    // 调用execute触发执行
    env.execute

  }

 }


 // 自定义Source，每隔1s持续产生消息
 class SimpleStringGenerator extends SourceFunction[String] {

  var running = true

  var i = 0



  override def run(ctx: SourceContext[String]) {

   while (running) {

    ctx.collect("element-" + i)

    i += 1

    Thread.sleep(1000)

   }

  }



  override def cancel() {

   running = false

  }

 }
//consumer代码
```

```
object ReadFromKafka {

    def main(args: Array[String]) {
    // 打印出执行flink run的参考命令
      System.out.println("use command as: ")

      System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka" +

        " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092")

      System.out.println
("*****************************************************************************************")

      System.out.println("<topic> is the kafka topic name")

      System.out.println("<bootstrap.servers> is the ip:port list of brokers")

      System.out.println
("*****************************************************************************************")


     // 构造执行环境
      val env = StreamExecutionEnvironment.getExecutionEnvironment
     // 设置并发度
      env.setParallelism(1)
     // 解析运行参数
      val paraTool = ParameterTool.fromArgs(args)
     // 构造流图，从Kafka读取数据并换行打印
      val messageStream = env.addSource(new FlinkKafkaConsumer(

        paraTool.get("topic"), new SimpleStringSchema, paraTool.getProperties))

      messageStream

        .map(s => "Flink says " + s + System.getProperty("line.separator")).print()
      // 调用execute触发执行
      env.execute()

    }

  }
```

9.4.3 Flink 开启 Checkpoint 样例程序

9.4.3.1 Flink 开启 Checkpoint 样例程序开发思路

场景说明

假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性，即：当应用出现异常并恢复后，各个算子的状态能够处于统一的状态。

数据规划

1. 使用自定义算子每秒钟产生大约10000条数据。
2. 产生的数据为一个四元组（Long，String，String，Integer）。
3. 数据经统计后，统计结果打印到终端输出。
4. 打印输出的结果为Long类型的数据。

开发思路

1. source算子每隔1秒钟发送10000条数据，并注入到Window算子中。
2. window算子每隔1秒钟统计一次最近4秒钟内数据数量。
3. 每隔1秒钟将统计结果打印到终端。具体查看方式请参考**查看Flink应用调测结果**。
4. 每隔6秒钟触发一次checkpoint，然后将checkpoint的结果保存到HDFS中。

9.4.3.2 Flink 开启 Checkpoint 样例程序（Java）

功能介绍

假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性。

代码样例

1. 快照数据

   该数据在算子制作快照时用于保存到目前为止算子记录的数据条数。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointJavaExample样例工程下的com.huawei.bigdata.flink.examples.UDFState：

```
import java.io.Seriablizale;

// 该类作为快照的一部分，保存用户自定义状态
public class UDFState implements Serializable {
    private long count;

    // 初始化用户自定义状态
    public UDFState() {
        count = 0L;
    }

    // 设置用户自定义状态
    public void setState(long count) {
       this.count = count;
    }

    // 获取用户自定义状态
    public long geState() {
        return this.count;
    }
}
```

2. 带checkpoint的数据源

   source算子的代码，该段代码每发送10000条数据休息1秒钟，制作快照时将到目前为止已经发送的数据的条数保存在UDFState中；从快照中状态恢复时，读取UDFState中的数据条数并重新赋值给count变量。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointJavaExample样例工程下的com.huawei.bigdata.flink.examples.SEventSourceWithChk：

```
import org.apache.flink.api.java.tuple.Tuple4;
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed;
import org.apache.flink.streaming.api.functions.source.RichSourceFunction;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

// 该类是带checkpoint的source算子
public class SEventSourceWithChk extends RichSourceFunction<Tuple4<Long, String, String, Integer>>
implements ListCheckpointed<UDFState> {
```



```
    private Long count = 0L;
    private boolean isRunning = true;
    private String alphabet =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuv
wxyzABCDEFGHIJKLMNOPQRSTUVWZYX0987654321";

    // 算子的主要逻辑，每秒钟向流图中注入10000个元组
    public void run(SourceContext<Tuple4<Long, String, String, Integer>> ctx) throws Exception {
        Random random = new Random();
        while(isRunning) {
            for (int i = 0; i < 10000; i++) {
                ctx.collect(Tuple4.of(random.nextLong(), "hello-" + count, alphabet, 1))
                count++;
            }
            Thread.sleep(1000);
        }
    }

    // 任务取消时调用
    public void cancel() {
        isRunning = false;
    }

    // 制作自定义快照
    public List<UDFState> snapshotState(long l, long ll) throws Exception {
        UDFState udfState = new UDFState();
        List<UDFState> listState = new ArrayList<UDFState>();
        udfState.setState(count);
        listState.add(udfState);
        return listState;
    }

    // 从自定义快照中恢复数据
    public void restoreState(List<UDFState> list) throws Exception {
        UDFState udfState = list.get(0);
        count = udfState.getState();
    }
}
```

3. 带checkpoint的窗口定义

   该段代码是window算子的代码，每当触发计算时统计窗口中元组数量。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointJavaExample样例工程下的com.huawei.bigdata.flink.examples.WindowStatisticWithChk：

```
import org.apache.flink.api.java.tuple.Tuple;
import org.apache.flink.api.java.tuple.Tuple4;
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed;
import org.apache.flink.streaming.api.functions.windowing.WindowFunction;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

import java.util.ArrayList;
import java.util.List;

// 该类是带checkpoint的window算子
public class WindowStatisticWithChk implements WindowFunction<Tuple4<Long, String, String,
Integer>, Long, Tuple, TimeWindow>, ListCheckpointed<UDFState> {
    private Long total = 0L;

    // window算子实现逻辑，统计window中元组的个数
    void apply(Tuple key, TimeWindow window, Iterable<Tuple4<Long, String, String, Integer>> input,
               Collector<Long> out) throws Exception {
        long count = 0L;
        for (Tuple4<Long, String, String, Integer> event : input) {
            count++;
        }
        total += count;
        out.collect(count);
    }
```

```
    // 制作自定义快照
    public List<UDFState> snapshotState(Long l, Long ll) {
        List<UDFState> listState = new ArrayList<UDFState>();
        UDFState udfState = new UDFState();
        udfState.setState(total);
        listState.add(udfState);
        return listState;
     }

     // 从自定义快照中恢复状态
     public void restoreState(List<UDFState> list) throws Exception {
         UDFState udfState = list.get(0);
         total = udfState.getState();
     }
}
```

4. 应用代码

   该段代码是流图定义代码，具体实现业务流程，另外，代码中窗口的触发时间使用了processing time。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointJavaExample样例工程下的com.huawei.bigdata.flink.examples.FlinkProcessingTimeAPIMain：

```
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

public class FlinkProcessingTimeAPIChkMain {
    public static void main(String[] args) throws Exception{

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 设置相关配置，并开启checkpoint功能
        env.setStateBackend(new FsStateBackend("hdfs://hacluster/flink/checkpoint/"));
        env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
        env.getCheckpointConfig.setCheckpointInterval(6000);

        // 应用逻辑
        env.addSource(new SEventSourceWithChk())
           .keyBy(0)
           .window(SlidingProcessingTimeWindows.of(Time.seconds(4), Time.seconds(1)))
           .apply(new WindowStatisticWithChk())
           .print()

        env.execute();
    }
}
```

9.4.3.3 Flink 开启 Checkpoint 样例程序（Scala）

功能介绍

假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性。

代码样例

1. 发送数据形式。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointScalaExample样例工程下的com.huawei.bigdata.flink.examples.SEvent：

```
case class SEvent(id: Long, name: String, info: String, count: Int)
```

2. 快照数据

   该数据在算子制作快照时用于保存到目前为止算子记录的数据条数。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointScalaExample样例工程下的com.huawei.bigdata.flink.examples.UDFState：

```
// 用户自定义状态
class UDFState extends Serializable{
    private var count = 0L

    // 设置用户自定义状态
    def setState(s: Long) = count = s

    // 获取用户自定状态
    def getState = count
}
```

3. 带checkpoint的数据源

   source算子的代码，该段代码每发送10000条数据休息1秒钟，制作快照时将到目前为止已经发送的数据的条数保存在UDFState中；从快照中状态恢复时，读取UDFState中的数据条数并重新赋值给count变量。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointScalaExample样例工程下的com.huawei.bigdata.flink.examples.SEventSourceWithChk：

```
import java.util
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed
import org.apache.flink.streaming.api.functions.source.RichSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext

// 该类是带有checkpoint的source算子
class SEventSourceWithChk extends RichSourceFunction[SEvent] with ListCheckpointed[UDFState]{
    private var count = 0L
    private var isRunning = true
    private val alphabet =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuv
wxyzABCDEFGHIJKLMNOPQRSTUVWZYX0987654321"

    // source算子的逻辑，即：每秒钟向流图中注入10000个元组
    override def run(sourceContext: SourceContext[SEvent]): Unit = {
        while(isRunning) {
            for (i <- 0 until 10000) {
                sourceContext.collect(SEvent(1, "hello-"+count, alphabet,1))
                count += 1L
            }
            Thread.sleep(1000)
        }
    }

    // 任务取消时调用
    override def cancel(): Unit = {
        isRunning = false;
    }

     override def close(): Unit = super.close()

     // 制作快照
     override def snapshotState(l: Long, l1: Long): util.List[UDFState] = {
         val udfList: util.ArrayList[UDFState] = new util.ArrayList[UDFState]
         val udfState = new UDFState
         udfState.setState(count)
         udfList.add(udfState)
         udfList
     }

     // 从快照中获取状态
     override def restoreState(list: util.List[UDFState]): Unit = {
         val udfState = list.get(0)
         count = udfState.getState
```

```
    }
}
```

4. 带checkpoint的窗口定义

   该段代码是window算子的代码，每当触发计算时统计窗口中元组数量。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointScalaExample样例工程下的com.huawei.bigdata.flink.examples.WindowStatisticWithChk：

```
import java.util
import org.apache.flink.api.java.tuple.Tuple
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed
import org.apache.flink.streaming.api.scala.function.WindowFunction
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

// 该类是带checkpoint的window算子
class WindowStatisticWithChk extends WindowFunction[SEvent, Long, Tuple, TimeWindow] with
ListCheckpointed[UDFState]{
    private var total = 0L

    // window算子的实现逻辑，即：统计window中元组的数量
    override def apply(key: Tuple, window: TimeWindow, input: Iterable[SEvent], out: Collector[Long]):
Unit = {
        var count = 0L
        for (event <- input) {
            count += 1L
        }
        total += count
        out.collect(count)
    }

    // 制作自定义状态快照
    override def snapshotState(l: Long, l1: Long): util.List[UDFState] = {
        val udfList: util.ArrayList[UDFState] = new util.ArrayList[UDFState]
        val udfState = new UDFState
        udfState.setState(total)
        udfList.add(udfState)
        udfList
    }

    // 从自定义快照中恢复状态
    override def restoreState(list: util.List[UDFState]): Unit = {
        val udfState = list.get(0)
        total = udfState.getState
    }
}
```

5. 应用代码

   该段代码是流图定义代码，具体实现业务流程，另外，代码中窗口的触发时间使用了event time。

   下面代码片段仅为演示，完整代码参见FlinkCheckpointScalaExample样例工程下的com.huawei.bigdata.flink.examples.FlinkEventTimeAPIChkMain：

```
import com.huawei.rt.flink.core.{SEvent, SEventSourceWithChk, WindowStatisticWithChk}
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend
import org.apache.flink.streaming.api.functions.AssignerWithPeriodicWatermarks
import org.apache.flink.streaming.api.{CheckpointingMode, TimeCharacteristic}
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.watermark.Watermark
import org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.api.scala._
import org.apache.flink.runtime.state.filesystem.FsStateBackend
import org.apache.flink.streaming.api.environment.CheckpointConfig.ExternalizedCheckpointCleanup

object FlinkEventTimeAPIChkMain {
    def main(args: Array[String]): Unit ={
        val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
    env.setStateBackend(new FsStateBackend("hdfs://hacluster/flink/checkpoint/"))
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.getConfig.setAutoWatermarkInterval(2000)
    env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
    env.getCheckpointConfig.setCheckpointInterval(6000)

    // 应用逻辑
    env.addSource(new SEventSourceWithChk)
      .assignTimestampsAndWatermarks(new AssignerWithPeriodicWatermarks[SEvent] {
        // 设置watermark
        override def getCurrentWatermark: Watermark = {
          new Watermark(System.currentTimeMillis())
        }
        // 给每个元组打上时间戳
        override def extractTimestamp(t: SEvent, l: Long): Long = {
          System.currentTimeMillis()
        }
      })
      .keyBy(0)
      .window(SlidingEventTimeWindows.of(Time.seconds(4), Time.seconds(1)))
      .apply(new WindowStatisticWithChk)
      .print()
    env.execute()
  }
}
```

9.4.4 Flink Job Pipeline 样例程序

9.4.4.1 Flink Job Pipeline 样例程序开发思路

场景说明

本样例中发布者Job自己每秒钟产生10000条数据，然后经由该job的NettySink算子向下游发送。另外两个Job作为订阅者，分别订阅一份数据。

数据规划

1. 发布者Job使用自定义算子每秒钟产生10000条数据。
2. 数据包含两个属性：分别是Int和String类型。
3. 配置文件。
   - nettyconnector.registerserver.topic.storage：设置NettySink的IP、端口及并发度信息在第三方注册服务器上的路径（必填），例如：
     ```
     nettyconnector.registerserver.topic.storage: /flink/nettyconnector
     ```
   - nettyconnector.sinkserver.port.range：设置NettySink的端口范围（必填），例如：
     ```
     nettyconnector.sinkserver.port.range: 28444-28943
     ```
   - nettyconnector.ssl.enabled：设置NettySink与NettySource之间通信是否SSL加密（默认为false），例如：
     ```
     nettyconnector.ssl.enabled: true
     ```
   - nettyconnector.sinkserver.subnet：设置网络所属域，例如：
     ```
     nettyconnector.sinkserver.subnet: 10.162.0.0/16
     ```
4. 安全认证配置：
   - Zookeeper的SASL认证，依赖“flink-conf.yaml”中有关HA的相关配置，具体配置请参见**配置管理Flink**。
   - SSL的keystore、truststore、keystore password、truststore password以及password等也使用“flink-conf.yaml”的相关配置，具体配置请参见**加密传输**。

5. 接口说明。
   - 注册服务器接口

     注册服务器用来保存NettySink的IP、端口以及并发度信息，以便NettySource连接使用。为用户提供以下接口：

```
public interface RegisterServerHandler {

/**
   * 启动注册服务器
    * @param configuration Flink的Configuration类型
   */
void start(Configuration configuration) throws Exception;
/**
       *注册服务器上创建Topic节点（目录）
       * @param topic topic节点名称
       */
void createTopicNode(String topic) throw Exception;
/**
*将信息注册到某个topic节点（目录）下
* @param topic 需要注册到的目录
* @param registerRecord 需要注册的信息
*/
void register(String topic, RegisterRecord registerRecord) throws Exception;
/**
      *删除topic节点
      * @param topic 待删除topic
      */
    void deleteTopicNode(String topic) throws Exception;
/**
   *注销注册信息
   *@param topic 注册信息所在的topic
   *@param recordId 待注销注册信息ID
   */
void unregister(String topic, int recordId) throws Exception;
/**
    * 查寻信息
* @param 查询信息所在的topic
*@recordId 查询信息的ID
*/
RegisterRecord query(String topic, int recordId) throws Exception;
/**
    * 查询某个Topic是否存在
    * @param topic
    */
Boolean isExist(String topic) throws Exception;
/**
    *关闭注册服务器句柄
   */
void shutdown() throws Exception;
```

     工程基于以上接口提供了ZookeeperRegisterHandler供用户使用。

   - NettySink算子

```
Class NettySink(String name,
String topic,
RegisterServerHandler registerServerHandler,
int numberOfSubscribedJobs)
```

     - name：为本NettySink的名称。
     - topic：为本NettySink产生数据的Topic，每个不同的NettySink（并发度除外）必须使用不同的TOPIC，否则会引起订阅混乱，数据无法正常分发。
     - registerServerHandler：为注册服务器的句柄。

- numberOfSubscribedJobs：为订阅本NettySink的作业数量，该数量必须是明确的，只有当所有订阅者都连接上NettySink，NettySink才发送数据。

– NettySource算子

```
Class  NettySource(String name,
String topic,
RegisterServerHandler registerServerHandler)
```

- name：为本NettySource的名称，该NettySource必须是唯一的（并发度除外），否则，连接NettySink时会出现冲突，导致无法连接。
- topic：订阅的NettySink的topic。
- registerServerHandler：为注册服务器的句柄。

说明

NettySource的并发度必须与NettySink的并发度相同，否则无法正常创建连接。

开发思路

1. 一个Job作为发布者Job，其余两个作为订阅者Job。
2. 发布者Job自己产生数据将其转化成byte[]，分别向订阅者发送。
3. 订阅者收到byte[]之后将其转化成String类型，并抽样打印输出。

9.4.4.2 Flink Job Pipeline 样例程序（Java）

1. 发布Job自定义Source算子产生数据

   下面代码片段仅为演示，完整代码参见FlinkPipelineJavaExample样例工程下的com.huawei.bigdata.flink.examples.UserSource：

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction;

import java.io.Serializable;

public class UserSource extends RichParallelSourceFunction<Tuple2<Integer, String>> implements
Serializable {

    private boolean isRunning = true;

    public void open(Configuration configuration) throws Exception {
        super.open(configuration);

    }
/**
    * 数据产生函数，每秒钟产生10000条数据
   */
    public void run(SourceContext<Tuple2<Integer, String>> ctx) throws Exception {

        while(isRunning) {
            for (int i = 0; i < 10000; i++) {
                ctx.collect(Tuple2.of(i, "hello-" + i));
            }
            Thread.sleep(1000);
        }
    }
```

```
    public void close() {
        isRunning = false;
    }

    public void cancel() {
        isRunning = false;
    }
}
```

2. 发布者代码

   下面代码片段仅为演示，完整代码参见FlinkPipelineJavaExample样例工程下的com.huawei.bigdata.flink.examples.TestPipeline_NettySink：

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.netty.sink.NettySink;
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler;

public class TestPipeline_NettySink {

    public static void main(String[] args) throws Exception{

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
//设置job的并发度为2
        env.setBufferTimeout(2);

// 创建Zookeeper的注册服务器handler
        ZookeeperRegisterServerHandler zkRegisterServerHandler = new
ZookeeperRegisterServerHandler();
// 添加自定义Source算子
        env.addSource(new UserSource())
                .keyBy(0)
                .map(new MapFunction<Tuple2<Integer,String>, byte[]>() {
                    //将发送信息转化成字节数组
@Override
                    public byte[] map(Tuple2<Integer, String> integerStringTuple2) throws Exception {
                        return integerStringTuple2.f1.getBytes();
                    }
                }).addSink(new NettySink("NettySink-1", "TOPIC-2", zkRegisterServerHandler, 2));//通过
NettySink发送出去。

        env.execute();

    }
}
```

3. 第一个订阅者

   下面代码片段仅为演示，完整代码参见FlinkPipelineJavaExample样例工程下的com.huawei.bigdata.flink.examples.TestPipeline_NettySource1：

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.netty.source.NettySource;
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler;

public class TestPipeline_NettySource1 {

    public static void main(String[] args) throws Exception{

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
// 设置job的并发度为2
env.setParallelism(2);
```

```
// 创建Zookeeper的注册服务器句柄
        ZookeeperRegisterServerHandler zkRegisterServerHandler = new
ZookeeperRegisterServerHandler();
//添加NettySource算子，接收来自发布者的消息
        env.addSource(new NettySource("NettySource-1", "TOPIC-2", zkRegisterServerHandler))
                .map(new MapFunction<byte[], String>() {
                  // 将接收到的字节流转化成字符串
    @Override
                    public String map(byte[] b) {
                        return new String(b);
                    }
                }).print();

        env.execute();
    }
}
```

4. 第二个订阅者

   下面代码片段仅为演示，完整代码参见FlinkPipelineJavaExample样例工程下的com.huawei.bigdata.flink.examples.TestPipeline_NettySource2：

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.netty.source.NettySource;
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler;

public class TestPipeline_NettySource2 {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
// 设置作业的并发度为2
 env.setParallelism(2);

//创建Zookeeper的注册服务器句柄
        ZookeeperRegisterServerHandler zkRegisterServerHandler = new
ZookeeperRegisterServerHandler();
//添加NettySource算子，接收来自发布者的数据
        env.addSource(new NettySource("NettySource-2", "TOPIC-2", zkRegisterServerHandler))
                .map(new MapFunction<byte[], String>() {
          //将接收到的字节数组转化成字符串
                    @Override
                    public String map(byte[] b) {
                        return new String(b);
                    }
                }).print();

        env.execute();
    }
}
```

9.4.4.3 Flink Job Pipeline 样例程序（Scala）

1. 发送消息

   下面代码片段仅为演示，完整代码参见FlinkPipelineScalaExample样例工程下的com.huawei.bigdata.flink.examples.Information：

```
package com.huawei.bigdata.flink.examples

case class Inforamtion(index: Int, content: String) {

  def this() = this(0, "")
}
```

2. 发布者job自定义source算子产生数据

下面代码片段仅为演示，完整代码参见FlinkPipelineScalaExample样例工程下的com.huawei.bigdata.flink.examples.UserSource：

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext

class UserSource extends RichParallelSourceFunction[Inforamtion] with Serializable{

 var isRunning = true

 override def open(parameters: Configuration): Unit = {
  super.open(parameters)

 }

// 每秒钟产生10000条数据
 override def run(sourceContext: SourceContext[Inforamtion]) = {

  while (isRunning) {
   for (i <- 0 until 10000) {
    sourceContext.collect(Inforamtion(i, "hello-" + i));

   }
   Thread.sleep(1000)
  }
 }

 override def close(): Unit = super.close()

 override def cancel() = {
  isRunning = false
 }
}
```

3. 发布者代码

   下面代码片段仅为演示，完整代码参见FlinkPipelineScalaExample样例工程下的com.huawei.bigdata.flink.examples.TestPipeline_NettySink：

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.netty.sink.NettySink
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler
import org.apache.flink.streaming.api.scala._

object TestPipeline_NettySink {

 def main(args: Array[String]): Unit = {

  val env = StreamExecutionEnvironment.getExecutionEnvironment
// 设置job的并发度为2
env.setParallelism(2)
//设置Zookeeper为注册服务器
  val zkRegisterServerHandler = new ZookeeperRegisterServerHandler
//添加用户自定义算子产生数据
env.addSource(new UserSource)
    .keyBy(0).map(x=>x.content.getBytes)//将发送数据转化成字节数组
    .addSink(new NettySink("NettySink-1", "TOPIC-2", zkRegisterServerHandler, 2))//添加NettySink算子发送数据

  env.execute()
 }
}
```

4. 第一个订阅者

下面代码片段仅为演示，完整代码参见FlinkPipelineScalaExample样例工程下的com.huawei.bigdata.flink.examples.TestPipeline_NettySource1：

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.netty.source.NettySource
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler
import org.apache.flink.streaming.api.scala._

import scala.util.Random


object TestPipeline_NettySource1 {

  def main(args: Array[String]): Unit = {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
// 设置Job的并发度为2
  env.setParallelism(2)
//设置Zookeeper作为注册服务器
val zkRegisterServerHandler = new ZookeeperRegisterServerHandler
//添加NettySource算子，接收来自发布者的数据
    env.addSource(new NettySource("NettySource-1", "TOPIC-2", zkRegisterServerHandler))
      .map(x => (1, new String(x)))//将接收到的字节流转化成字符串
      .filter(x => {
        Random.nextInt(50000) == 10
      })
      .print

    env.execute()
  }
}
```

5. 第二个订阅者

   下面代码片段仅为演示，完整代码参见FlinkPipelineScalaExample样例工程下的com.huawei.bigdata.flink.examples.TestPipeline_NettySource2：

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.netty.source.NettySource
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler
import org.apache.flink.streaming.api.scala._

import scala.util.Random


object TestPipeline_NettySource2 {

  def main(args: Array[String]): Unit = {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
//设置job的并发度为2
 env.setParallelism(2)
//创建Zookeeper作为注册服务器
    val zkRegisterServerHandler = new ZookeeperRegisterServerHandler
//添加NettySource算子，接收数据
env.addSource(new NettySource("NettySource-2", "TOPIC-2", zkRegisterServerHandler))
      .map(x=>(2, new String(x)))//将接收到的字节数组转化成字符串
      .filter(x=>{
        Random.nextInt(50000) == 10
      })
      .print()

    env.execute()
  }
}
```


9.4.5 Flink Join 样例程序

9.4.5.1 Flink Join 样例程序开发思路

场景说明

假定某个Flink业务1每秒就会收到1条消息记录，消息记录某个用户的基本信息，包括名字、性别、年龄。另有一个Flink业务2会不定时收到1条消息记录，消息记录该用户的名字、职业信息。

基于某些业务要求，开发的Flink应用程序实现功能：实时的以根据业务2中消息记录的用户名字作为关键字，对两个业务数据进行联合查询。

数据规划

- 业务1的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户），并从Kafka组件接收数据。Kafka配置参见样例**数据规划**章节。
- 业务2的数据通过socket接收消息记录，可使用netcat命令用户输入模拟数据源。
  - 使用Linux命令netcat -l -p <port>，启动一个简易的文本服务器。
  - 启动应用程序连接netcat监测的port成功后，向netcat终端输入数据信息。

开发思路

1. 启动Flink Kafka Producer应用向Kafka发送数据。
2. 启动Flink Kafka Consumer应用从Kafka接收数据，构造Table1，保证topic与producer一致。
3. 从soket中读取数据，构造Table2。
4. 使用Flink SQL对Table1和Table2进行联合查询，并进行打印。

9.4.5.2 Flink Join 样例程序（Java）

功能介绍

在Flink应用中，调用flink-connector-kafka模块的接口，生产并消费数据。

代码样例

用户在开发前需要使用对接安全模式的Kafka，则需要引入FusionInsight的kafka-clients-*.jar，该jar包可在Kafka客户端目录下获取。下面列出producer和consumer，以及Flink Stream SQL Join使用主要逻辑代码作为演示。

1. 每秒钟往Kafka中生产一条用户信息，用户信息有姓名、年龄、性别组成。

   下面代码片段仅为演示，完整代码参见FlinkStreamSqlJoinExample样例工程下的com.huawei.bigdata.flink.examples.WriteIntoKafka。

```
//producer代码
public class WriteIntoKafka {

    public static void main(String[] args) throws Exception {

    // 打印出执行flink run的参考命令
      System.out.println("use command as: ");
```

```
        System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +

            " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092");

        System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +

            " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21007 --security.protocol
SASL_PLAINTEXT --sasl.kerberos.service.name kafka");


        System.out.println("******************************************************************************************");

        System.out.println("<topic> is the kafka topic name");

        System.out.println("<bootstrap.servers> is the ip:port list of brokers");

        System.out.println("******************************************************************************************");

        // 构造执行环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 设置并发度
        env.setParallelism(1);
        // 解析运行参数
        ParameterTool paraTool = ParameterTool.fromArgs(args);
        // 构造流图，将自定义Source生成的数据写入Kafka
        DataStream<String> messageStream = env.addSource(new SimpleStringGenerator());

        FlinkKafkaProducer<String> producer = new FlinkKafkaProducer<>(paraTool.get("topic"),

            new SimpleStringSchema(),

            paraTool.getProperties());

        producer.setWriteTimestampToKafka(true);

        messageStream.addSink(producer);

        // 调用execute触发执行
        env.execute();
    }

// 自定义Source，每隔1s持续产生消息
public static class SimpleStringGenerator implements SourceFunction<String> {
        static final String[] NAME = {"Carry", "Alen", "Mike", "Ian", "John", "Kobe", "James"};

        static final String[] SEX = {"MALE", "FEMALE"};

        static final int COUNT = NAME.length;

        boolean running = true;

        Random rand = new Random(47);

       @Override
        //rand随机产生名字，性别，年龄的组合信息
         public void run(SourceContext<String> ctx) throws Exception {

            while (running) {

                int i = rand.nextInt(COUNT);

                int age = rand.nextInt(70);

                String sexy = SEX[rand.nextInt(2)];

                ctx.collect(NAME[i] + "," + age + "," + sexy);

                thread.sleep(1000);
```

```
        }
    }
    @Override
    public void cancel() {
      running = false;
    }
  }
}
```

2. 生成Table1和Table2，并使用Join对Table1和Table2进行联合查询，打印输出结果。

   下面代码片段仅为演示，完整代码参见FlinkStreamSqlJoinExample样例工程下的com.huawei.bigdata.flink.examples.SqlJoinWithSocket。

```
public class SqlJoinWithSocket {
   public static void main(String[] args) throws Exception{

      final String hostname;

      final int port;

      System.out.println("use command as: ");

      System.out.println("flink run --class com.huawei.bigdata.flink.examples.SqlJoinWithSocket" +
            " /opt/test.jar --topic topic-test -bootstrap.servers xxxx.xxx.xxx.xxx:9092 --hostname
xxx.xxx.xxx.xxx --port xxx");

      System.out.println("flink run --class com.huawei.bigdata.flink.examples.SqlJoinWithSocket" +
            " /opt/test.jar --topic topic-test -bootstrap.servers xxxx.xxx.xxx.xxx:21007 --security.protocol
SASL_PLAINTEXT --sasl.kerberos.service.name kafka"
            + "--hostname xxx.xxx.xxx.xxx --port xxx");


      System.out.println("*******************************************************************************************");
      System.out.println("<topic> is the kafka topic name");
      System.out.println("<bootstrap.servers> is the ip:port list of brokers");
      System.out.println("*******************************************************************************************");

      try {
         final ParameterTool params = ParameterTool.fromArgs(args);

         hostname = params.has("hostname") ? params.get("hostname") : "localhost";

         port = params.getInt("port");

      } catch (Exception e) {
         System.err.println("No port specified. Please run 'FlinkStreamSqlJoinExample " +
               "--hostname <hostname> --port <port>', where hostname (localhost by default) " +
               "and port is the address of the text server");

         System.err.println("To start a simple text server, run 'netcat -l -p <port>' and " +
               "type the input text into the command line");

         return;
      }
      EnvironmentSettings fsSettings =
EnvironmentSettings.newInstance().useOldPlanner().inStreamingMode().build();
      StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
      StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, fsSettings);

      //基于EventTime进行处理
```

```
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        env.setParallelism(1);

        ParameterTool paraTool = ParameterTool.fromArgs(args);

        //Stream1，从Kafka中读取数据
        DataStream<Tuple3<String, String, String>> kafkaStream = env.addSource(new
FlinkKafkaConsumer<>(paraTool.get("topic"),
                new SimpleStringSchema(),
                paraTool.getProperties())).map(new MapFunction<String, Tuple3<String, String, String>>()
{
            @Override
            public Tuple3<String, String, String> map(String s) throws Exception {
                String[] word = s.split(",");

                return new Tuple3<>(word[0], word[1], word[2]);
            }
        });

        //将Stream1注册为Table1
        tableEnv.registerDataStream("Table1", kafkaStream, "name, age, sexy, proctime.proctime");

        //Stream2，从Socket中读取数据
        DataStream<Tuple2<String, String>> socketStream = env.socketTextStream(hostname, port,
"\n").
                map(new MapFunction<String, Tuple2<String, String>>() {
                    @Override
                    public Tuple2<String, String> map(String s) throws Exception {
                        String[] words = s.split("\\s");
                        if (words.length < 2) {
                            return new Tuple2<>();
                        }

                        return new Tuple2<>(words[0], words[1]);
                    }
                });

        //将Stream2注册为Table2
        tableEnv.registerDataStream("Table2", socketStream, "name, job, proctime.proctime");

        //执行SQL Join进行联合查询
        Table result = tableEnv.sqlQuery("SELECT t1.name, t1.age, t1.sexy, t2.job, t2.proctime as shiptime
\n" +
                "FROM Table1 AS t1\n" +
                "JOIN Table2 AS t2\n" +
                "ON t1.name = t2.name\n" +
                "AND t1.proctime BETWEEN t2.proctime - INTERVAL '1' SECOND AND t2.proctime +
INTERVAL '1' SECOND");

        //将查询结果转换为Stream，并打印输出
        tableEnv.toAppendStream(result, Row.class).print();

        env.execute();
    }
}
```

9.5 调测 Flink 应用

9.5.1 编译并调测 Flink 应用

操作场景

在程序代码完成开发后，编译jar包并上传至Linux客户端环境中运行应用。使用Scala或Java语言开发的应用程序在Flink客户端的运行步骤是相同的。

说明

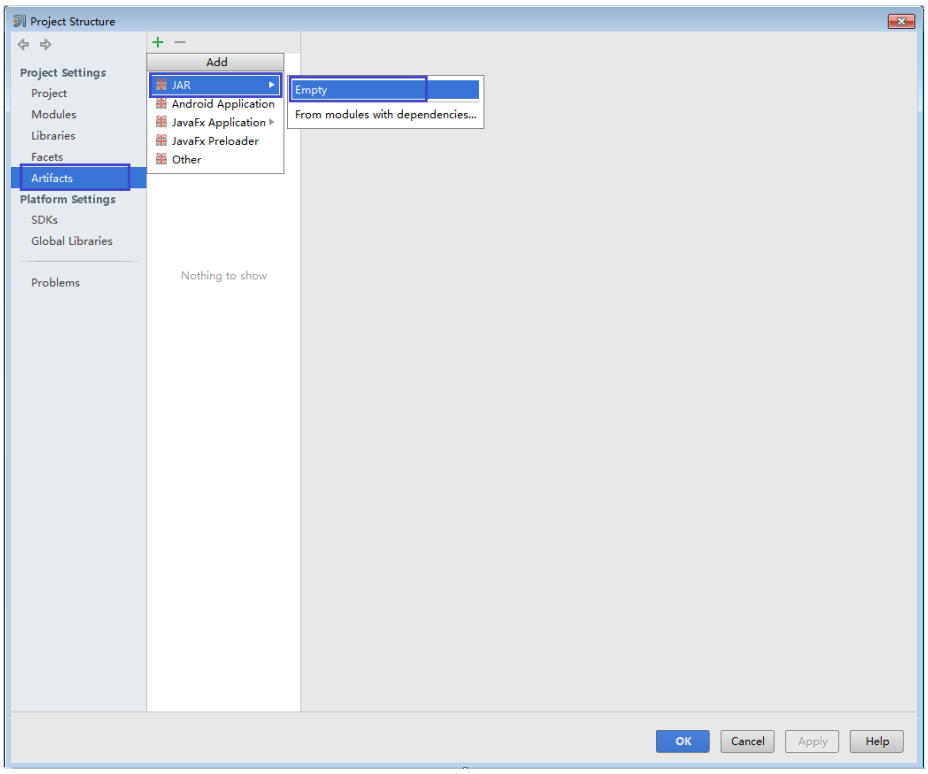
基于YARN集群的Flink应用程序不支持在Windows环境下运行，只支持在Linux环境下运行。

操作步骤

步骤1 在IntelliJ IDEA中，在生成Jar包之前配置工程的Artifacts信息。

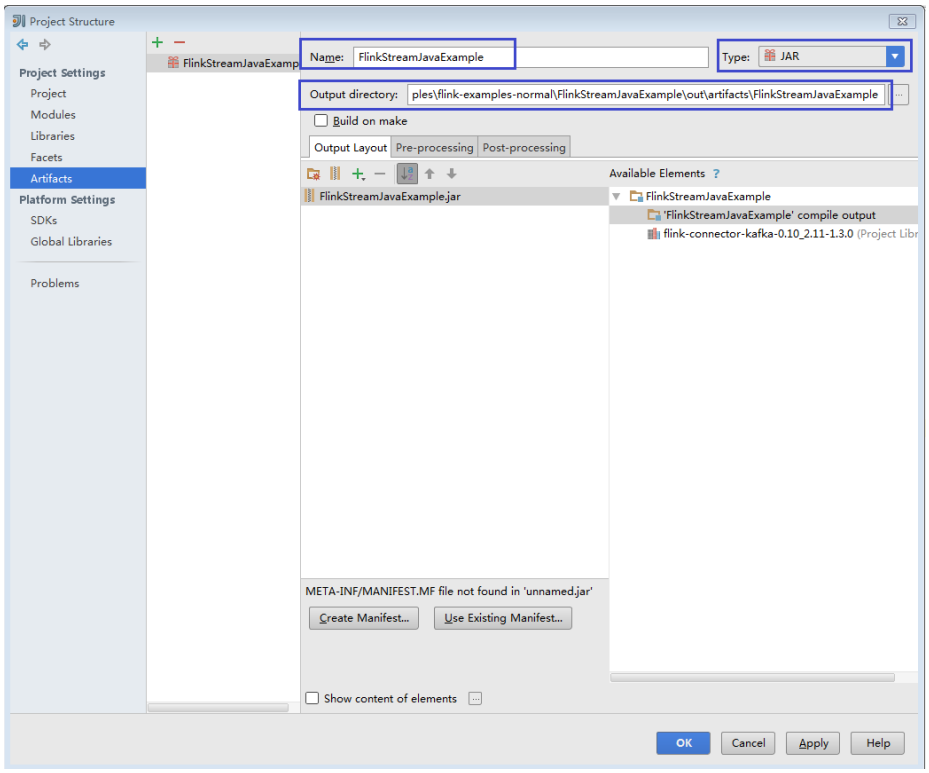
1. 在IDEA主页面，选择“File > Project Structures...”进入“Project Structure”页面。
2. 在“Project Structure”页面，选择“Artifacts”，单击“+”并选择“JAR > Empty”。

图 9-40 添加 Artifacts



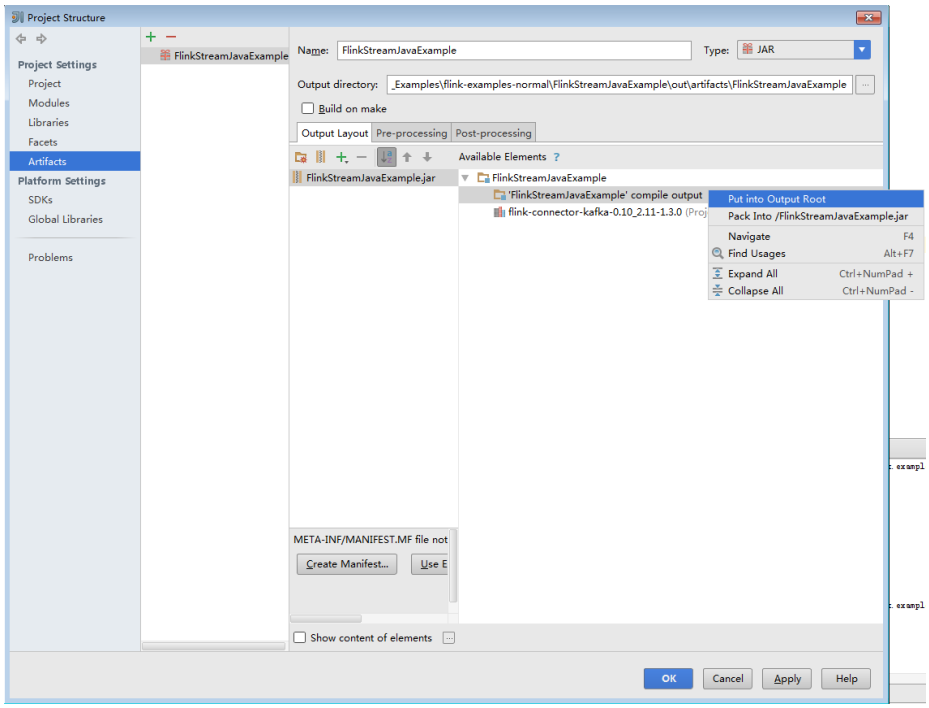
3. 您可以根据实际情况设置Jar包的名称、类型以及输出路径。

图 9-41 设置基本信息



4. 选中“'FlinkStreamJavaExample' compile output”，右键选择“Put into Output Root”。然后单击“Apply”。

图 9-42 Put into Output Root

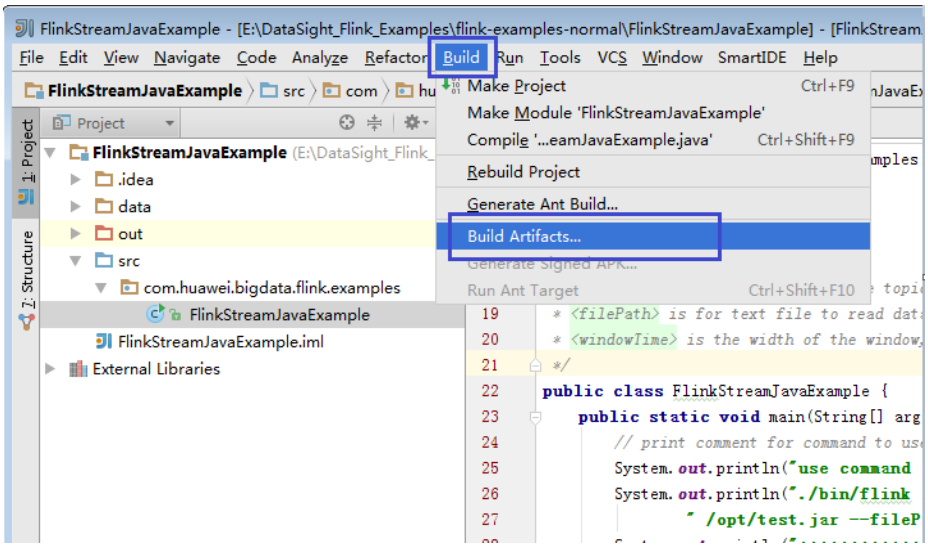


5. 最后单击“OK”完成配置。

步骤2 生成Jar包。

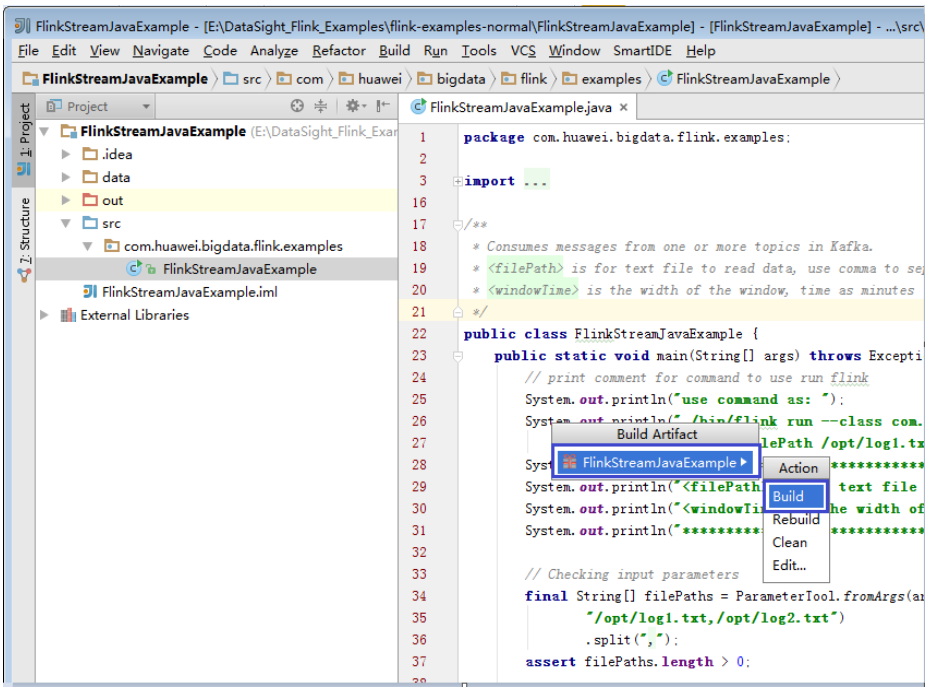
1. 在IDEA主页面，选择“Build > Build Artifacts...”。

图 9-43 Build Artifacts



2. 在弹出的菜单中，选择“FlinkStreamJavaExample > Build”开始生成Jar包。

图 9-44 Build



3. 当Event log中出现如下类似日志时，表示Jar包生成成功，您可以从**步骤1.3**中配置的路径下获取到Jar包。

```
21:25:43 Compilation completed successfully in 36 sec
```

步骤3 将**步骤2**中生成的Jar包（如FlinkStreamJavaExample.jar）复制到Flink客户端节点相关目录下，例如“/opt/client”。然后在该目录下创建“conf”目录，将需要的配置文件复制至“conf”目录，具体操作请参考**准备运行环境配置文件**，运行Flink应用程序。

在Linux环境中运行Flink应用程序，需要先启动Flink集群。在Flink客户端下通过yarn session命令启动flink集群。

例如：

cd /opt/client/Flink/flink

bin/yarn-session.sh -jm 1024 -tm 1024 -t conf/ssl/

说明

- 执行yarn-session.sh之前，应预先将Flink应用程序的运行依赖包复制到客户端目录“#{客户端安装目录}/Flink/flink/lib”下，应用程序运行依赖包请参考**参考信息**。
- 在Flink任务运行过程中禁止重启HDFS服务或者重启所有DataNode实例，否则可能会导致任务失败，并可能导致应用部分临时数据无法清空。
- 示例中的“ssl/”是Flink客户端目录下自定义的子目录，用来存放SSL keystore、truststore相关配置文件。

- 运行DataStream（Scala和Java）样例程序。

  在终端另开一个窗口，进入Flink客户端目录，调用bin/flink run脚本运行代码。

  – Java

    bin/flink run --class com.huawei.bigdata.flink.examples.FlinkStreamJavaExample /opt/client/FlinkStreamJavaExample.jar --filePath /opt/log1.txt,/opt/log2.txt --windowTime 2

  – Scala

    bin/flink run --class com.huawei.bigdata.flink.examples.FlinkStreamScalaExample /opt/client/FlinkStreamScalaExample.jar --filePath /opt/log1.txt,/opt/log2.txt --windowTime 2

  说明

  “log1.txt”、“log2.txt”放置在本地时，需放在每个部署了Yarn NodeManager实例的节点上，权限为755。

  表 9-11 参数说明

参数名称	说明
<filePath>	指本地文件系统中文件路径，每个节点都需要放一份/opt/log1.txt和/opt/log2.txt。可以默认，也可以设置。
<windowTime>	指窗口时间大小，以分钟为单位。可以默认，也可以设置。

- 运行向Kafka生产并消费数据样例程序（Scala和Java语言）。

  生产数据的执行命令启动程序。

  ```
  bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka /opt/client/FlinkKafkaJavaExample.jar <topic> <bootstrap.servers> [security.protocol] [sasl.kerberos.service.name] [ssl.truststore.location] [ssl.truststore.password] [kerberos.domain.name]
  ```

  消费数据的执行命令启动程序。

  ```
  bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/FlinkKafkaJavaExample.jar <topic> <bootstrap.servers> [security.protocol] [sasl.kerberos.service.name] [ssl.truststore.location] [ssl.truststore.password]
  ```

表 9-12 参数说明

参数名称	说明	是否必须配置
topic	表示Kafka主题名。	是
bootstrap.server	表示broker集群ip/port列表。	是
security.protocol	运行参数可以配置为PLAINTEXT（可不配置）/SASL_PLAINTEXT/SSL/SASL_SSL四种协议，分别对应FusionInsight Kafka集群的21005/21007/21008/21009端口。 – 如果配置了SASL，则必须配置sasl.kerberos.service.name为kafka，配置kerberos.domain.name为hadoop.*系统域名*，并在conf/flink-conf.yaml中配置security.kerberos.login相关配置项。 **说明** 登录FusionInsight Manager页面，选择“系统 > 权限 > 域和互信 > 本端域”，即可查看系统域名，系统域名所有字母需转换为小写。 – 如果配置了SSL，则必须配置ssl.truststore.location和ssl.truststore.password，前者表示truststore的位置，后者表示truststore密码。	否 **说明** – 该参数未配置时为非安全Kafka。 – 如果需要配置SSL，truststore.jks文件生成方式可参考“Kafka开发指南 > 客户端SSL加密功能使用说明”章节。

说明

- 执行本样例工程，需配置“allow.everyone.if.no.acl.found”为“true”，详情请参考**配置对接Kafka**。
- Kafka应用需要添加如下所示的jar文件：
  - Flink服务端安装路径的lib目录下“flink-dist_*.jar”。
  - Flink服务端安装路径的opt目录下的“flink-connector-kafka_*.jar”。
  - Kafka客户端或Kafka服务端安装路径中的lib目录下“kafka-clients-*.jar”。
  - truststore.jks配置若配置为绝对路径，应将该truststore.jks文件放置于每一个Yarn nodemanager指定目录；若配置为相对路径，应在yarn-session.sh脚本执行时添加上传目录，例如在执行**▪命令4：**之前，应首先执行yarn-session.sh -t config/***。

四种类型实际命令示例，以ReadFromKafka为例，集群域名为“HADOOP.COM”：

– 命令1：

```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/
FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:9092
```

- 命令2：

```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/
FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:21007 --
security.protocol SASL_PLAINTEXT --sasl.kerberos.service.name kafka --kerberos.domain.name
hadoop.hadoop.com
```

- 命令3：

```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/conf/
FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:9093 --
security.protocol SSL --ssl.truststore.location /home/truststore.jks --ssl.truststore.password xxx
```

- 命令4：

```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/
FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:21009 --
security.protocol SASL_SSL --sasl.kerberos.service.name kafka --ssl.truststore.location config/
truststore.jks --ssl.truststore.password xxx --kerberos.domain.name hadoop.hadoop.com
```

- 运行异步Checkpoint机制样例程序（Scala和Java语言）。

  为了丰富样例代码，Java版本使用了Processing Time作为数据流的时间戳，而Scala版本使用Event Time作为数据流的时间戳。具体执行命令参考如下：

  - 将Checkpoint的快照信息保存到HDFS。

    - Java

      ```
      bin/flink run --class com.huawei.bigdata.flink.examples.FlinkProcessingTimeAPIMain /opt/
      client/FlinkCheckpointJavaExample.jar --chkPath hdfs://hacluster/flink/checkpoint/
      ```

    - Scala

      ```
      bin/flink run --class com.huawei.bigdata.flink.examples.FlinkEventTimeAPIChkMain /opt/
      client/conf/FlinkCheckpointScalaExample.jar --chkPath hdfs://hacluster/flink/checkpoint/
      ```

  - 将Checkpoint的快照信息保存到本地文件。

    - Java

      ```
      bin/flink run --class com.huawei.bigdata.flink.examples.FlinkProcessingTimeAPIMain /opt/
      client/FlinkCheckpointJavaExample.jar --chkPath file:///home/zzz/flink-checkpoint/
      ```

    - Scala

      ```
      bin/flink run --class com.huawei.bigdata.flink.examples.FlinkEventTimeAPIChkMain
      /opt/client/FlinkCheckpointScalaExample.jar --ckkPath file:///home/zzz/flink-checkpoint/
      ```

  说明

  - Checkpoint源文件路径：flink/checkpoint/fd5f5b3d08628d83038a30302b611/chk-X/4f854bf4-ea54-4595-a9d9-9b9080779ffe

    flink/checkpoint //指定的根目录。

    fd5f5b3d08628d83038a30302b611 //以jobID命名的第二层目录。

    chk-X // "X"为checkpoint编号，第三层目录。

    4f854bf4-ea54-4595-a9d9-9b9080779ffe //checkpoint源文件。
  - Flink在集群模式下checkpoint将文件放到HDFS，本地路径只支持Flink的local模式，便于调测。

- 运行Pipeline样例程序。

  - Java

    i. 启动发布者Job

    ```
    bin/flink run -p 2 --class com.huawei.bigdata.flink.examples.TestPipeline_NettySink /opt/
    client/FlinkPipelineJavaExample.jar
    ```

    ii. 启动订阅者Job1

    ```
    bin/flink run --class com.huawei.bigdata.flink.examples.TestPipeline_NettySource1 /opt/
    client/FlinkPipelineJavaExample.jar
    ```


iii. 启动订阅者Job2
```
bin/flink run --class com.huawei.bigdata.flink.examples.TestPipeline_NettySource2 /opt/client/FlinkPipelineJavaExample.jar
```

- Scala

i. 启动发布者Job
```
bin/flink run -p 2 --class com.huawei.bigdata.flink.examples.TestPipeline_NettySink /opt/client/FlinkPipelineScalaExample.jar
```

ii. 启动订阅者Job1
```
bin/flink run --class com.huawei.bigdata.flink.examples.TestPipeline_NettySource1 /opt/client/FlinkPipelineScalaExample.jar
```

iii. 启动订阅者Job2
```
bin/flink run --class com.huawei.bigdata.flink.examples.TestPipeline_NettySource2 /opt/client/FlinkPipelineScalaExample.jar
```

- 运行Stream SQL Join样例程序

a. 启动程序向Kafka生产。Kafka配置可参考**运行向Kafka生产并消费数据样例程序（Scal...**
```
bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka /opt/client/FlinkStreamSqlJoinExample.jar --topic topic-test --bootstrap.servers xxx.xxx.xxx.xxx:9092
```

b. 在集群内任一节点启动netcat命令，等待应用程序连接。
```
netcat -l -p 9000
```

c. 启动程序接受Socket数据，并执行联合查询。
```
bin/flink run --class com.huawei.bigdata.flink.examples.SqlJoinWithSocket /opt/client/FlinkStreamSqlJoinExample.jar --topic topic-test --bootstrap.servers xxx.xxx.xxx.xxx:9092 --hostname xxx.xxx.xxx.xxx --port 9000
```

----结束

说明

针对Flink提供的几个样例工程，其对应的运行依赖包请参考**参考信息**。

9.5.2 查看 Flink 应用调测结果

操作场景

Flink应用程序运行完成后，您可以查看运行结果数据，也可以通过Flink WebUI查看应用程序运行情况。

操作步骤

- **查看Flink应用运行结果数据。**

  当用户查看执行结果时，需要在Flink的web页面上查看Task Manager的Stdout日志。

  当执行结果输出到文件或者其他，由Flink应用程序指定，您可以通过指定文件或其他获取到运行结果数据。以下用Checkpoint、Pipeline和配置表与流JOIN为例：

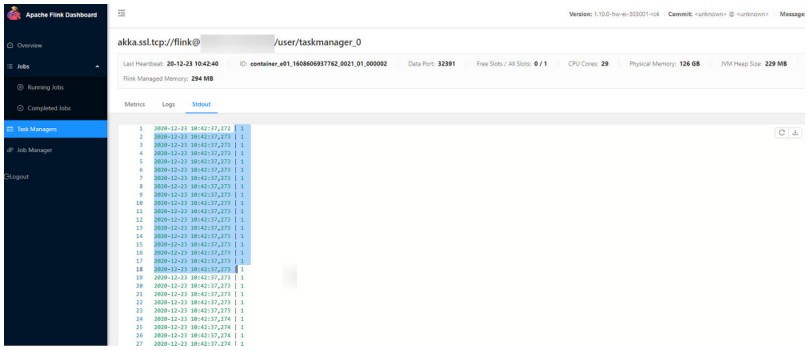
  - **查看Checkpoint结果和文件**

    - 结果在flink的“taskmanager.out”文件中。用户可以进入Yarn的WebUI页面，选择“Jobs > Checkpoints”查看提交的作业如**图9-45**。选择“Task Managers > Stdout”查看运行结果如**图9-46**。

      图 9-45 提交的作业

图 9-46 运行结果



- 有两种方式查看Checkpoint文件。
  - 若将checkpoint的快照信息保存到HDFS，则通过执行**hdfs dfs -ls hdfs://hacluster/flink/checkpoint/**命令查看。
  - 若将checkpoint的快照信息保存到本地文件，则可直接登录到各个节点查看。
- **查看Pipeline结果**
  - 结果在flink的“taskmanager.out”文件中。用户可以进入Yarn的WebUI页面，选择“Jobs > Running Jobs”查看提交的作业如图9-47。选择“Task Managers ”可以看到有两个任务如图9-48。分别单击任意Task，选择“Stdout”查看该任务的输出结果如图9-49和图9-50。

图 9-47 提交的作业

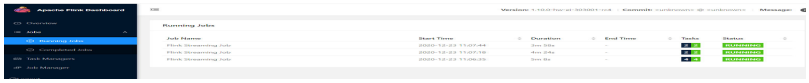


图 9-48 提交的任务

图 9-49 Task1 输出结果

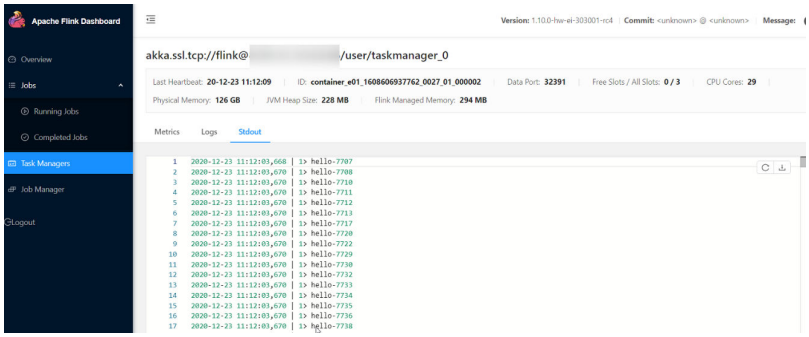
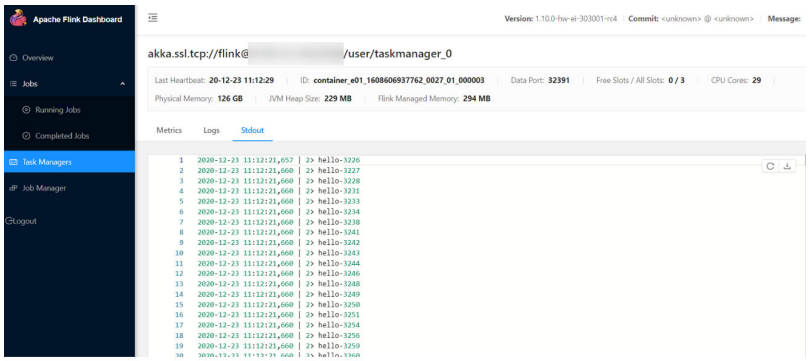


图 9-50 Task2 输出结果



- **查看配置表与流JOIN结果**
  - 结果在flink的“taskmanager.out”文件中，用户可以进入Yarn的WebUI页面，选择“Jobs > Completed Jobs”查看完成作业如图9-51。选择“Task Managers ”查看提交的任务如图9-52。单击该任务进入该任务详细信息页面，单击“Stdout”查看该任务的输出结果如图9-53。

图 9-51 运行完成的作业

图 9-52 提交的任务

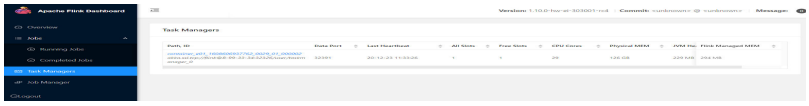
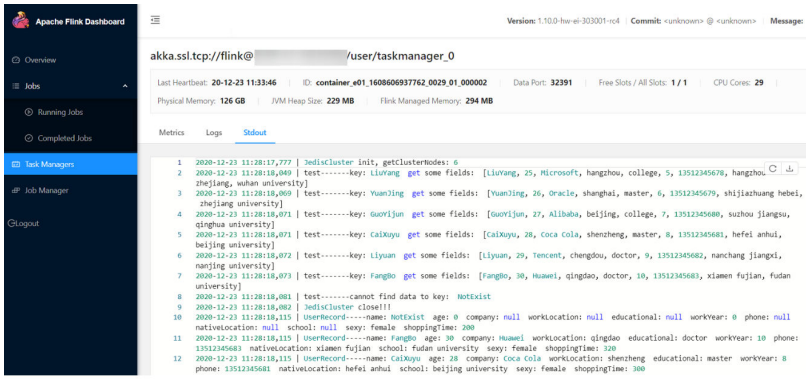


图 9-53 输出结果



- **查看DataStream结果**
  - 结果在flink的“taskmanager.out”文件中，用户可以进入Yarn的WebUI页面，选择“Jobs > Completed Jobs”查看完成作业如图9-54。选择“Task Managers ”查看提交的任务如图9-55。单击该任务进入该任务详细信息页面，单击“Stdout”查看该任务的输出结果如图9-56。

图 9-54 运行完成的作业

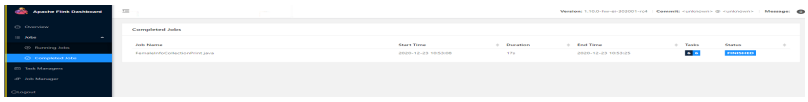
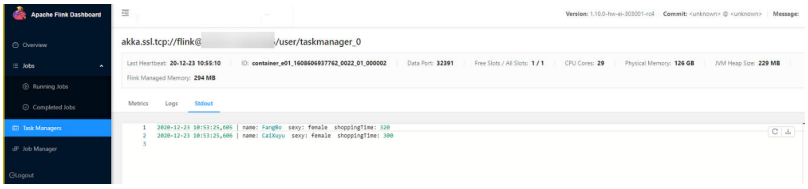


图 9-55 提交的任务

图 9-56 运行结果



- **查看Stream SQL Join结果**
  - 结果在flink的“taskmanager.out”文件中，用户可以进入Yarn的WebUI页面，选择“Jobs > Running Jobs”查看运行的作业如图9-57。选择“Task Managers ”查看提交的任务如图9-58。单击该任务进入该任务详细信息页面，单击“Stdout”查看该任务的输出结果如图9-59。

图 9-57 运行的作业

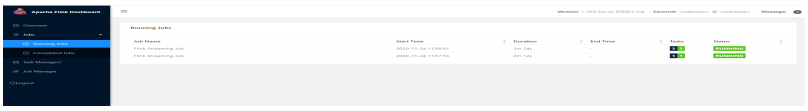


图 9-58 提交的任务

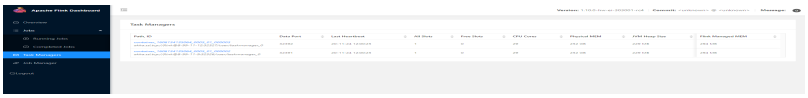
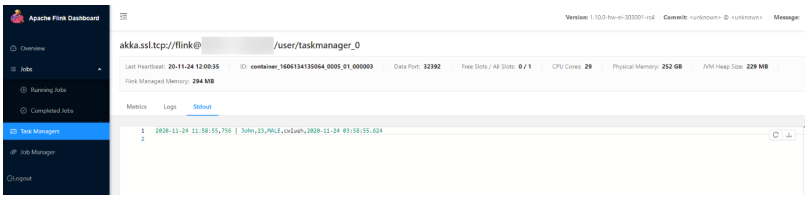


图 9-59 运行结果



- **查看向Kafka生产并消费数据结果**
  - 结果在flink的“taskmanager.out”文件中，用户可以进入Yarn的WebUI页面，选择“Jobs > Running Jobs”查看运行的作业如图9-60。选择“Task Managers ”查看提交的任务如图9-61。单击该任务进入该任务详细信息页面，单击“Stdout”查看该任务的输出结果如图9-62。

图 9-60 运行的作业

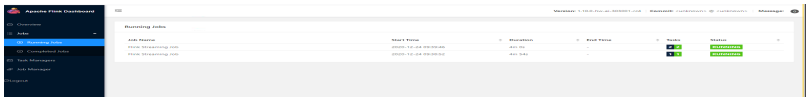


图 9-61 提交的任务

图 9-62 运行结果

- **使用Flink Web页面查看Flink应用程序运行情况。**

  Flink Web页面主要包括了Overview、Running Jobs、Completed Jobs、Task Managers、Job Manager和Logout等部分。

  在YARN的Web UI界面，查找到对应的Flink应用程序。单击应用信息的最后一列“ApplicationMaster”，即可进入Flink Web页面。

  查看程序执行的打印结果：找到对应的Task Manager，查看对应的Stdout标签日志信息。

- **查看Flink日志获取应用运行情况。**

  有三种方式获取Flink日志，分别为通过Flink Web页面或者Yarn的日志。

  - Flink Web页面可以查看Task Managers、Job Manager部分的日志。
  - Yarn页面主要包括了Job Manager日志以及GC日志等。

    页面入口：在YARN的Web UI界面，查找到对应的Flink应用程序。单击应用信息的第一列ID，然后选择Logs列单击进去即可打开。
  - 使用Yarn客户端获取或查看Task Managers、Job Manager的日志，具体操作如下：

    i. 下载并安装Yarn客户端（例安装目录：/opt/client）。

    ii. 以客户端安装用户，登录安装客户端的节点。

    iii. 执行以下命令，切换到客户端安装目录。

    cd /opt/client

    iv. 执行以下命令配置环境变量。

    source bigdata_env

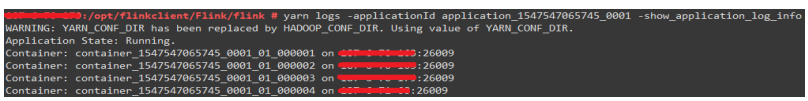
    v. 如果集群为安全模式，执行以下命令进行用户认证。普通模式集群无需执行用户认证。

    kinit *组件业务用户*

    vi. 执行以下命令，获取Flink集群container信息。

    yarn logs -applicationId application_* -show_application_log_info

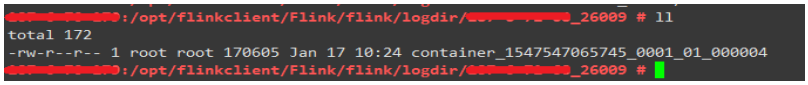
    图 9-63 获取 Flink 集群 container 信息



vii. 执行以下命令，获取指定container运行日志，通常container_*_000001为JobManager运行所在container。

yarn logs -applicationId application_* --containerId container_1547547065745_0001_01_000004 -out logdir/

图 9-64 获取指定 container 运行日志



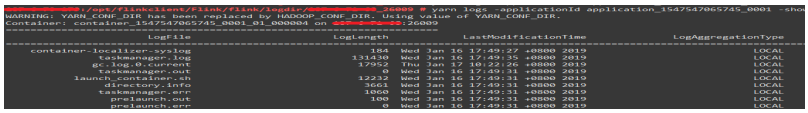
上述命令会将container运行日志下载至本地，该日志包含了TaskManager/JobManager的运行日志，GC日志等信息。

viii. 还可以使用如下命令获取指定名称日志。

获取container日志列表：

yarn logs -applicationId application_* -show_container_log_info --containerId container_1547547065745_0001_01_000004

图 9-65 获取 container 日志列表



下载指定日志taskmanager.log至本地：

yarn logs -applicationId application_* --containerId container_1547547065745_0001_01_000004 -log_files taskmanager.log -out localpath

9.6 Flink 应用开发常见问题

9.6.1 Flink 常用 API 介绍

9.6.1.1 Flink Java API 接口介绍

由于Flink开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

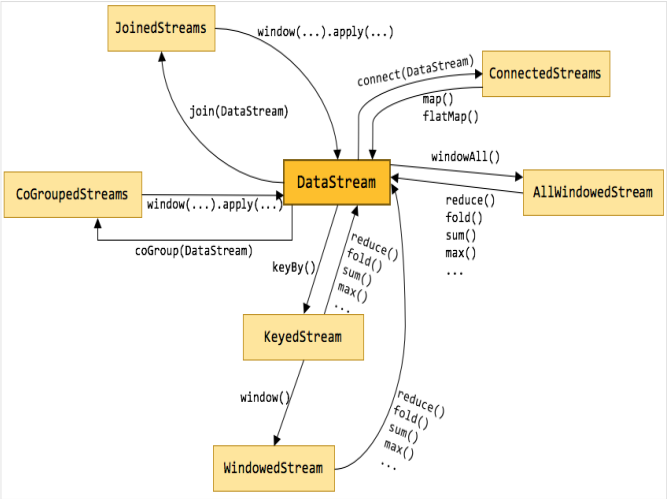
Flink 常用接口

Flink主要使用到如下这几个类：

- StreamExecutionEnvironment：是Flink流处理的基础，提供了程序的执行环境。
- DataStream：Flink用类DataStream来表示程序中的流式数据。用户可以认为它们是含有重复数据的不可修改的集合(collection)，DataStream中元素的数量是无限的。
- KeyedStream：DataStream通过keyBy分组操作生成流，通过设置的key值对数据进行分组。

- WindowedStream：KeyedStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- AllWindowedStream：DataStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- ConnectedStreams：将两条DataStream流连接起来并且保持原有流数据的类型，然后进行map或者flatMap操作。
- JoinedStreams：在窗口上对数据进行等值join操作（等值就是判断两个值相同的join，比如a.id = b.id），join操作是coGroup操作的一种特殊场景。
- CoGroupedStreams：在窗口上对数据进行coGroup操作，可以实现流的各种join类型。

图 9-66 Flink Stream 的各种流类型转换



流数据输入

表 9-13 流数据输入的相关接口

API	说明
public final <OUT> DataStreamSource<OUT> fromElements(OUT... data)	获取用户定义的多个元素的数据，作为输入流数据。 • type为元素的数据类型。 • data为多个元素的具体数据。
public final <OUT> DataStreamSource<OUT> fromElements(Class<OUT> type, OUT... data)	
public <OUT> DataStreamSource<OUT> fromCollection(Collection<OUT> data)	获取用户定义的集合数据，作为输入流数据。 • type为集合中元素的数据类型。 • typeInfo为集合中根据元素数据类型获取的类型信息。 • data为集合数据或者可迭代的数据体。
public <OUT> DataStreamSource<OUT> fromCollection(Collection<OUT> data, TypeInformation<OUT> typeInfo)	

API	说明
public <OUT> DataStreamSource<OUT> fromCollection(Iterator<OUT> data, Class<OUT> type)	
public <OUT> DataStreamSource<OUT> fromCollection(Iterator<OUT> data, TypeInformation<OUT> typeInfo)	
public <OUT> DataStreamSource<OUT> fromParallelCollection(SplittableIterator<OUT> iterator, Class<OUT> type)	获取用户定义的集合数据，作为输入并行流数据。 ● type为集合中元素的数据类型。 ● typeInfo为集合中根据元素数据类型获取的类型信息。 ● iterator为可被分割成多个partition的迭代数据体。
public <OUT> DataStreamSource<OUT> fromParallelCollection(SplittableIterator<OUT> iterator, TypeInformation<OUT> typeInfo)	
private <OUT> DataStreamSource<OUT> fromParallelCollection(SplittableIterator<OUT> iterator, TypeInformation<OUT> typeInfo, String operatorName)	
public DataStreamSource<Long> generateSequence(long from, long to)	获取用户定义的一串序列数据，作为输入流数据。 ● from是指数值串的起点。 ● to是指数值串的终点。
public DataStreamSource<String> readTextFile(String filePath)	获取用户定义的某路径下的文本文件数据，作为输入流数据。 ● filePath是指文本文件的路径。 ● charsetName是编码格式的名字。
public DataStreamSource<String> readTextFile(String filePath, String charsetName)	
public <OUT> DataStreamSource<OUT> readFile(FileInputFormat<OUT> inputFormat, String filePath)	获取用户定义的某路径下的文件数据，作为输入流数据。 ● filePath是指文件的路径。 ● inputFormat是指文件的格式。 ● watchType指的是文件的处理模式“PROCESS_ONCE”或者“PROCESS_CONTINUOUSLY”。 ● interval指的是多长时间判断目录或文件变化进行处理。
public <OUT> DataStreamSource<OUT> readFile(FileInputFormat<OUT> inputFormat, String filePath, FileProcessingMode watchType, long interval)	

API	说明
public <OUT> DataStreamSource<OUT> readFile(FileInputFormat<OUT> inputFormat, String filePath, FileProcessingMode watchType, long interval, TypeInformation<OUT> typeInformation)	
public DataStreamSource<String> socketTextStream(String hostname, int port, String delimiter, long maxRetry)	获取用户定义的Socket数据，作为输入流数据。 • hostname是指Socket的服务器端的主机名称。 • port指的是服务器的监测端口。 • delimiter指的是消息之间的分隔符。 • maxRetry指的是由于连接异常可以触发的最大重试次数。
public DataStreamSource<String> socketTextStream(String hostname, int port, String delimiter)	
public DataStreamSource<String> socketTextStream(String hostname, int port)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function)	用户自定义SourceFunction，addSource方法可以添加Kafka等数据源，主要实现方法为SourceFunction的run。 • function指的是用户自定义的SourceFunction函数。 • sourceName指的是定义该数据源的名称。 • typeInfo则是根据元素数据类型获取的类型信息。
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, String sourceName)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, TypeInformation<OUT> typeInfo)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, String sourceName, TypeInformation<OUT> typeInfo)	

数据输出

表 9-14 数据输出的相关接口

API	说明
public DataStreamSink<T> print()	数据输出以标准输出流打印出来。
public DataStreamSink<T> printToErr()	数据输出以标准error输出流打印出来。

API	说明
public DataStreamSink<T> writeAsText(String path)	数据输出写入到某个文本文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。
public DataStreamSink<T> writeAsText(String path, WriteMode writeMode)	
public DataStreamSink<T> writeAsCsv(String path)	数据输出写入到某个csv格式的文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。 ● rowDelimiter为行分隔符。 ● fieldDelimiter为列分隔符。
public DataStreamSink<T> writeAsCsv(String path, WriteMode writeMode)	
public <X extends Tuple> DataStreamSink<T> writeAsCsv(String path, WriteMode writeMode, String rowDelimiter, String fieldDelimiter)	
public DataStreamSink<T> writeToSocket(String hostName, int port, SerializationSchema<T> schema)	数据输出写入到Socket连接中。 ● hostName为主机名称。 ● port为端口。
public DataStreamSink<T> writeUsingOutputFormat(OutputFormat<T> format)	数据输出到普通文件中，例如二进制文件。
public DataStreamSink<T> addSink(SinkFunction<T> sinkFunction)	用户自定义的数据输出，addSink方法可以添加Kafka等数据输出，主要实现方法为SinkFunction的invoke方法。

过滤和映射能力

表 9-15 过滤和映射能力的相关接口

API	说明
public <R> SingleOutputStreamOperator<R> map(MapFunction<T, R> mapper)	输入一个元素，生成另一个元素，元素类型不变。
public <R> SingleOutputStreamOperator<R> flatMap(FlatMapFunction<T, R> flatMapper)	输入一个元素，生成零个、一个或者多个元素。
public SingleOutputStreamOperator<T> filter(FilterFunction<T> filter)	对每个元素执行一个布尔函数，只保留返回true的元素。

聚合能力

表 9-16 聚合能力的相关接口

API	说明
public KeyedStream<T, Tuple> keyBy(int... fields)	将流逻辑分区成不相交的分区，每个分区包含相同key的元素。内部是用hash分区来实现的。这个转换返回了一个KeyedStream。 KeyBy操作之后返回KeyedStream，然后再调用KeyedStream的函数（例如reduce/fold/min/minby/max/maxby/sum/sumby等）进行相应操作。 ● fields为数据某几列的序号或者成员变量的名称。 ● key则为用户自定义的指定分区依据的方法。
public KeyedStream<T, Tuple> keyBy(String... fields)	
public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key)	
public SingleOutputStreamOperator<T> reduce(ReduceFunction<T> reducer)	在一个KeyedStream上“滚动”reduce。合并当前元素与上一个被reduce的值，然后输出新的值。注意三者的类型是一致的。
public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> folder)	在一个KeyedStream上基于一个初始值“滚动”折叠。合并当前元素和上一个被折叠的值，然后输出新值。注意Fold的输入值与返回值类型可以不一致。
public SingleOutputStreamOperator<T> sum(int positionToSum)	在一个KeyedStream上滚动求和操作。 positionToSum和field代表对某一列求和。
public SingleOutputStreamOperator<T> sum(String field)	
public SingleOutputStreamOperator<T> min(int positionToMin)	在一个KeyedStream上滚动求最小值。min返回了最小值，不保证非最小值列的准确性。 positionToMin和field代表对某一列求最小值。
public SingleOutputStreamOperator<T> min(String field)	
public SingleOutputStreamOperator<T> max(int positionToMax)	在一个KeyedStream上滚动求最大值。max返回了最大值，不保证非最大值列的准确性。 positionToMax和field代表对某一列求最大值。
public SingleOutputStreamOperator<T> max(String field)	

API	说明
public SingleOutputStreamOperator<T> minBy(int positionToMinBy)	在一个KeyedStream上求某一列最小值所在的该行数据，minBy返回了该行数据的所有元素。 ● positionToMinBy代表对哪一列做minBy操作。 ● first表示是否按最先遇到的最小值输出还是最后遇到的最小值输出。
public SingleOutputStreamOperator<T> minBy(String positionToMinBy)	
public SingleOutputStreamOperator<T> minBy(int positionToMinBy, boolean first)	
public SingleOutputStreamOperator<T> minBy(String field, boolean first)	
public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy)	在一个KeyedStream上求某一列最大值所在的该行数据，maxBy返回了该行数据的所有元素。 ● positionToMaxBy代表对哪一列做maxBy操作。 ● first表示是否按最先遇到的最大值输出还是最后遇到的最大值输出。
public SingleOutputStreamOperator<T> maxBy(String positionToMaxBy)	
public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy, boolean first)	
public SingleOutputStreamOperator<T> maxBy(String field, boolean first)	

数据流分发能力

表 9-17 数据流分发能力的相关接口

API	说明
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, int field)	使用一个用户自定义的Partitioner对每一个元素选择目标task。 ● partitioner指的是用户自定义的分区类重写partition方法。 ● field指的是partitioner的输入参数。 ● keySelector指的是用户自定义的partitioner的输入参数。
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, String field)	

API	说明
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, KeySelector<T, K> keySelector)	
public DataStream<T> shuffle()	以均匀分布的形式将元素随机地进行分区。
public DataStream<T> rebalance()	基于round-robin对元素进行分区，使得每个分区负责均衡。对于存在数据倾斜的性能优化是很有用的。
public DataStream<T> rescale()	以round-robin的形式将元素分区到下游操作的子集中。
public DataStream<T> broadcast()	广播每个元素到所有分区。

提供 project 的能力

表 9-18 提供 project 的能力的相关接口

API	说明
public <R extends Tuple> SingleOutputStreamOperator<R> project(int... fieldIndexes)	从元组中选择了一部分字段子集。 fieldIndexes指的是需要选择的元组中的某几个序列。 **说明** 只支持Tuple数据类型的project投影。

提供设置 eventtime 属性的能力

表 9-19 提供设置 eventtime 属性的能力的相关接口

API	说明
public SingleOutputStreamOperator<T> assignTimestampsAndWatermarks(AssignerWithPeriodicWatermarks<T> timestampAndWatermarkAssigner)	为了能让event time窗口可以正常触发窗口计算操作，需要从记录中提取时间戳。

API	说明
public SingleOutputStreamOperator<T> assignTimestampsAndWatermarks(AssignerWithPunctuatedWatermarks<T> timestampAndWatermarkAssigner)	

根据接口参数不同可以分为以上两种，AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks的区别如表9-20所示。

表 9-20 接口参数区别

参数	说明
AssignerWithPeriodicWatermarks	根据StreamExecutionEnvironment类的getConfig().setAutoWatermarkInterval(200L)时间戳生成Watermark。
AssignerWithPunctuatedWatermarks	每接收到一个元素，都会生成一个Watermark，而且可以根据接收到的元素生成不同的Watermark。

提供迭代的能力

表 9-21 提供迭代的能力的相关接口

API	说明
public IterativeStream<T> iterate()	在流(flow)中创建一个带反馈的循环，通过重定向一个operator的输出到之前的operator。 **说明** • 对于定义一些需要不断更新模型的算法是非常有帮助的。 • long maxWaitTimeMillis：该超时时间指的是每一轮迭代体执行的超时时间。
public IterativeStream<T> iterate(long maxWaitTimeMillis)	

提供分流能力

表 9-22 提供分流能力的相关接口

API	说明
public SplitStream<T> split(OutputSelector< T> outputSelector)	传入OutputSelector，重写select方法确定分流的依据(即打标记)，构建SplitStream流。即对每个元素做一个字符串的标记，作为选择的依据，打好标记之后就可以通过标记选出并新建某个标记的流。
public DataStream<OUT> select(String... outputNames)	从一个SplitStream中选出一个或多个流。 outputNames指的是使用split方法对每个元素做的字符串标记的序列。

窗口能力

窗口分为跳跃窗口和滑动窗口。

- 支持Window、TimeWindow、CountWindow以及WindowAll、TimeWindowAll、CountWindowAll API窗口生成。
- 支持Window Apply、Window Reduce、Window Fold、Aggregations on windows API窗口操作。
- 支持多种Window Assigner（TumblingEventTimeWindows、TumblingProcessingTimeWindows、SlidingEventTimeWindows、SlidingProcessingTimeWindows、EventTimeSessionWindows、ProcessingTimeSessionWindows、GlobalWindows）。
- 支持三种时间ProcessingTime、EventTime和IngestionTime。
- 支持两种EventTime时间戳方式：AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks。

窗口生成类API如表9-23所示。

表 9-23 窗口生成类的相关接口

API	说明
public <W extends Window> WindowedStream<T, KEY, W> window(WindowAssigner<? super T, W> assigner)	窗口可以被定义在已经被分区的KeyedStreams上。窗口会对数据的每一个key根据一些特征（例如在最近5秒钟内到达的数据）进行分组。

API	说明
public <W extends Window> AllWindowedStream<T, W> windowAll(WindowAssigner<? super T, W> assigner)	窗口可以被定义在DataStream上。
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size)	时间窗口定义在已经被分区的KeyedStreams上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。 **说明** ● WindowedStream和AllWindowedStream代表不同的两种流。 ● 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size, Time slide)	
public AllWindowedStream<T, TimeWindow> timeWindowAll(Time size)	时间窗口定义在DataStream上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。
public AllWindowedStream<T, TimeWindow> timeWindowAll(Time size, Time slide)	
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size)	按照元素个数区分窗口，定义在已经被分区的KeyedStreams上。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。 **说明** ● WindowedStream和AllWindowedStream代表不同的两种流。 ● 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size, long slide)	
public AllWindowedStream<T, GlobalWindow> countWindowAll(Time size)	按照元素个数区分窗口，定义在DataStream上。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。

API	说明
public AllWindowedStream<T, GlobalWindow> countWindowAll(Time size, Time slide)	

窗口操作类API如表9-24所示。

表 9-24 窗口操作类的相关接口

方法	API	说明
Window	public <R> SingleOutputStreamOperator<R> apply(WindowFunction<T, R, K, W> function)	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。 ● function指的是执行的窗口函数。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> apply(WindowFunction<T, R, K, W> function, TypeInformation<R> resultType)	
	public SingleOutputStreamOperator<T> reduce(ReduceFunction<T> function)	应用一个reduce函数到窗口上，返回reduce后的值。 ● reduceFunction指的是执行的reduce函数。 ● WindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, WindowFunction<T, R, K, W> function)	
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, WindowFunction<T, R, K, W> function, TypeInformation<R> resultType)	

方法	API	说明
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function)	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function, TypeInformation<R> resultType)	
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, WindowFunction<ACC, R, K, W> function)	
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, WindowFunction<ACC, R, K, W> function, TypeInformation<ACC> foldAccumulatorType, TypeInformation<R> resultType)	
WindowAll	public <R> SingleOutputStreamOperator<R> apply(AllWindowFunction<T, R, W> function)	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。
	public <R> SingleOutputStreamOperator<R> apply(AllWindowFunction<T, R, W> function, TypeInformation<R> resultType)	

方法	API	说明
	public SingleOutputStreamOperator<T> reduce(ReduceFunction<T> function)	应用一个reduce函数到窗口上，返回reduce后的值。 ● reduceFunction指的是执行的reduce函数。 ● AllWindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, AllWindowFunction<T, R, W> function)	
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, AllWindowFunction<T, R, W> function, TypeInformation<R> resultType)	
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function)	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function, TypeInformation<R> resultType)	
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, AllWindowFunction<ACC, R, W> function)	
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, AllWindowFunction<ACC, R, W> function, TypeInformation<ACC> foldAccumulatorType, TypeInformation<R> resultType)	

方法	API	说明
Window 和 Window All	public SingleOutputStreamOperator< T> sum(int positionToSum)	对窗口数据的某一列求和。 positionToSum和field代表数据的某一列。
	public SingleOutputStreamOperator< T> sum(String field)	
	public SingleOutputStreamOperator< T> min(int positionToMin)	对窗口数据的某一列求最小值。min返回了最小值，不保证非最小值列的准确性。 positionToMin和field代表对某一列求最小值。
	public SingleOutputStreamOperator< T> min(String field)	
	public SingleOutputStreamOperator< T> minBy(int positionToMinBy)	对窗口数据的某一列求最小值所在的该行数据，minBy返回了该行数据的所有元素。 ● positionToMinBy代表对哪一列做minBy操作。 ● first表示是否按最先遇到的最小值输出还是最后遇到的最小值输出。
	public SingleOutputStreamOperator< T> minBy(String positionToMinBy)	
	public SingleOutputStreamOperator< T> minBy(int positionToMinBy, boolean first)	
	public SingleOutputStreamOperator< T> minBy(String field, boolean first)	
	public SingleOutputStreamOperator< T> max(int positionToMax)	对窗口数据的某一列求最大值。max返回了最大值，不保证非最大值列的准确性。 positionToMax和field代表对某一列求最大值。
	public SingleOutputStreamOperator< T> max(String field)	
	public SingleOutputStreamOperator< T> maxBy(int positionToMaxBy)//默认true	对窗口数据的某一列求最大值所在的该行数据，maxBy返回了在这个字段上是最大值的所有元素。 ● positionToMaxBy代表对哪一列做maxBy操作。 ● first表示是否按最先遇到的最大值输出还是最后遇到的最大值输出。
	public SingleOutputStreamOperator< T> maxBy(String positionToMaxBy)//默认true	

方法	API	说明
	public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy, boolean first)	
	public SingleOutputStreamOperator<T> maxBy(String field, boolean first)	

提供多流合并的能力

表 9-25 提供多流合并的能力的相关接口

API	说明
public final DataStream<T> union(DataStream<T>... streams)	Union两个或多个数据流，生成一个新的包含了来自所有流的所有数据的数据流。 **说明** 如果你将一个数据流与其自身进行了合并，在结果流中对于每个元素你都会拿到两份。
public <R> ConnectedStreams<T, R> connect(DataStream<R> dataStream)	“连接”两个数据流并保持原先的类型。Connect可以让两条流之间共享状态。产生ConnectedStreams之后，调用map或者flatmap进行操作计算。
public <R> SingleOutputStreamOperator<R> map(CoMapFunction<IN1, IN2, R> coMapper)	在ConnectedStreams上做元素映射，类似DataStream的map操作，元素映射之后流数据类型统一。
public <R> SingleOutputStreamOperator<R> flatMap(CoFlatMapFunction<IN1, IN2, R> coFlatMapper)	在ConnectedStreams上做元素映射，类似DataStream的flatMap操作，元素映射之后流数据类型统一。

提供 Join 能力

表 9-26 提供 Join 能力的相关接口

API	说明
public <T2> JoinedStreams<T, T2> join(DataStream<T2> otherStream)	通过给定的key在一个窗口范围内join两条数据流。
public <T2> CoGroupedStreams<T, T2> coGroup(DataStream< T2> otherStream)	通过给定的key在一个窗口范围内co-group两条数据流。

9.6.1.2 Flink Scala API 接口介绍

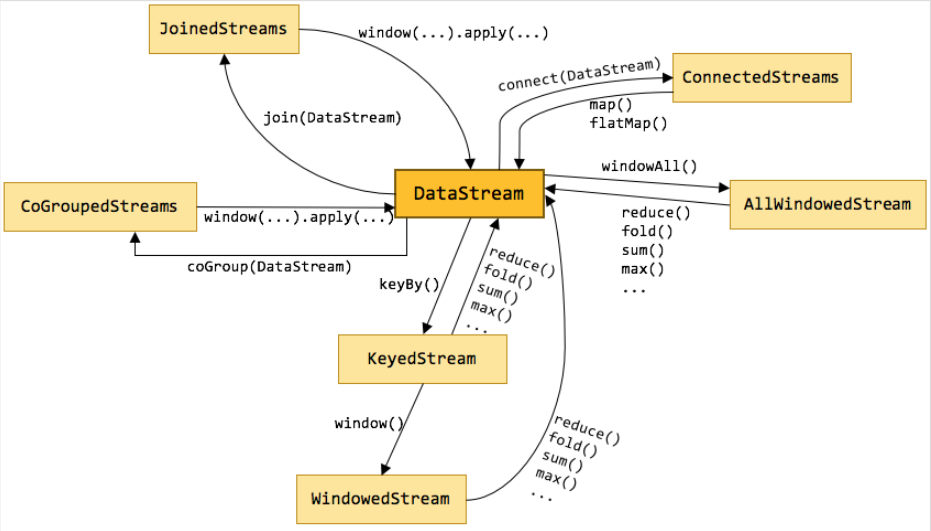
由于Flink开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

Flink 常用接口

Flink主要使用到如下这几个类：

- StreamExecutionEnvironment：是Flink流处理的基础，提供了程序的执行环境。
- DataStream：Flink用特别的类DataStream来表示程序中的流式数据。用户可以认为它们是含有重复数据的不可修改的集合(collection)，DataStream中元素的数量是无限的。
- KeyedStream：DataStream通过keyBy分组操作生成流，数据经过对设置的key值进行分组。
- WindowedStream：KeyedStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- AllWindowedStream：DataStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- ConnectedStreams：将两条DataStream流连接起来并且保持原有流数据的类型，然后进行map或者flatMap操作。
- JoinedStreams：在窗口上对数据进行等值join操作，join操作是coGroup操作的一种特殊场景。
- CoGroupedStreams：在窗口上对数据进行coGroup操作，可以实现流的各种join类型。

图 9-67 Flink Stream 的各种流类型转换



流数据输入

表 9-27 流数据输入的相关接口

API	说明
def fromElements[T: TypeInformation](data: T*): DataStream[T]	获取用户定义的多个元素的数据，作为输入流数据。 data是多个元素的具体数据。
def fromCollection[T: TypeInformation](data: Seq[T]): DataStream[T]	获取用户定义的集合数据，作为输入流数据。 data可以是集合数据或者可迭代的数据体。
def fromCollection[T: TypeInformation](data: Iterator[T]): DataStream[T]	
def fromParallelCollection[T: TypeInformation] (data: SplittableIterator[T]): DataStream[T]	获取用户定义的集合数据，作为输入并行流数据。 data是可被分割成多个partition的迭代数据体。
def generateSequence(from: Long, to: Long): DataStream[Long]	获取用户定义的一串序列数据，作为输入流数据。 • from是指数值串的起点。 • to是指数值串的终点。
def readTextFile(filePath: String): DataStream[String]	获取用户定义的某路径下的文本文件数据，作为输入流数据。 • filePath是指文本文件的路径。 • charsetName指的是编码格式的名字。
def readTextFile(filePath: String, charsetName: String): DataStream[String]	

API	说明
def readFile[T: TypeInformation] (inputFormat: FileInputFormat[T], filePath: String)	获取用户定义的某路径下的文件数据，作为输入流数据。 ● filePath是指文件的路径。 ● inputFormat是指文件的格式。 ● watchType指的是文件的处理模式“PROCESS_ONCE”或者“PROCESS_CONTINUOUSLY”。 ● interval指的是多长时间判断目录或文件变化进行处理。
def readFile[T: TypeInformation] (inputFormat: FileInputFormat[T], filePath: String, watchType: FileProcessingMode, interval: Long): DataStream[T]	
def socketTextStream(hostname: String, port: Int, delimiter: Char = '\n', maxRetry: Long = 0): DataStream[String]	获取用户定义的Socket数据，作为输入流数据。 ● hostname是指Socket的服务器端的主机名称。 ● port指的是服务器的监测端口。 ● delimiter和maxRetry两个参数scala接口暂时不支持设置。
def addSource[T: TypeInformation] (function: SourceFunction[T]): DataStream[T]	用户自定义SourceFunction，addSource方法可以添加Kafka等数据源，主要实现方法为SourceFunction的run。 ● function指的是用户自定义的SourceFunction函数。 ● scala支持简化写法。
def addSource[T: TypeInformation] (function: SourceContext[T] => Unit): DataStream[T]	

数据输出

表 9-28 数据输出的相关接口

API	说明
def print(): DataStreamSink[T]	数据输出以标准输出流打印出来。
def printToErr()	数据输出以标准error输出流打印出来。
def writeAsText(path: String): DataStreamSink[T]	数据输出写入到某个文本文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。
def writeAsText(path: String, writeMode: FileSystem.WriteMode): DataStreamSink[T]	

API	说明
def writeAsCsv(path: String): DataStreamSink[T]	数据输出写入到某个csv格式的文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。 ● rowDelimiter为行分隔符。 ● fieldDelimiter为列分隔符。
def writeAsCsv(path: String, writeMode: FileSystem.WriteMode): DataStreamSink[T]	
def writeAsCsv(path: String, writeMode: FileSystem.WriteMode, rowDelimiter: String, fieldDelimiter: String): DataStreamSink[T]	
def writeUsingOutputFormat(format: OutputFormat[T]): DataStreamSink[T]	数据输出到普通文件中，例如二进制文件。
def writeToSocket(hostname: String, port: Integer, schema: SerializationSchema[T]): DataStreamSink[T]	数据输出写入到Socket连接中。 ● hostName为主机名称。 ● port为端口。
def addSink(sinkFunction: SinkFunction[T]): DataStreamSink[T]	用户自定义的数据输出，addSink方法通过flink-connectors支持数据输出到Kafka，主要实现方法为SinkFunction的invoke方法。
def addSink(fun: T => Unit): DataStreamSink[T]	

过滤和映射能力

表 9-29 过滤和映射能力的相关接口

API	说明
def map[R: TypeInformation](fun: T => R): DataStream[R]	输入一个元素，生成另一个元素，元素类型不变。
def map[R: TypeInformation](mapper: MapFunction[T, R]): DataStream[R]	
def flatMap[R: TypeInformation](flatMapper: FlatMapFunction[T, R]): DataStream[R]	输入一个元素，生成零个、一个或者多个元素。
def flatMap[R: TypeInformation](fun: (T, Collector[R]) => Unit): DataStream[R]	
def flatMap[R: TypeInformation](fun: T => TraversableOnce[R]): DataStream[R]	
def filter(filter: FilterFunction[T]): DataStream[T]	对每个元素执行一个布尔函数，只保留返回true的元素。

API	说明
def filter(fun: T => Boolean): DataStream[T]	

聚合能力

表 9-30 聚合能力的相关接口

API	说明
def keyBy(fields: Int*): KeyedStream[T, JavaTuple]	将流逻辑分区成不相交的分区，每个分区包含相同key的元素。内部是用hash分区来实现的。这个转换返回了一个KeyedStream。 KeyBy操作之后返回KeyedStream，然后再调用KeyedStream的函数（例如reduce/fold/min/minby/max/maxby/sum/sumby等）进行相应操作。 ● fields为数据某几列的序号。 ● firstField和otherFields为数据结构的成员变量的名称。 ● key为用户自定义的指定分区依据的方法。
def keyBy(firstField: String, otherFields: String*): KeyedStream[T, JavaTuple]	
def keyBy[K: TypeInformation](fun: T => K): KeyedStream[T, K]	
def reduce(fun: (T, T) => T): DataStream[T]	在一个KeyedStream上“滚动”reduce。合并当前元素与上一个被reduce的值，然后输出新的值。注意三者的类型是一致的。
def reduce(reducer: ReduceFunction[T]): DataStream[T]	
def fold[R: TypeInformation](initialValue: R)(fun: (R,T) => R): DataStream[R]	在一个KeyedStream上基于一个初始值“滚动”折叠。合并当前元素和上一个被折叠的值，然后输出新值。注意Fold的输入值与返回值类型可以不一致。
def fold[R: TypeInformation](initialValue: R, folder: FoldFunction[T,R]): DataStream[R]	
def sum(position: Int): DataStream[T]	在一个KeyedStream上滚动求和操作。 position和field代表对某一列求和。
def sum(field: String): DataStream[T]	
def min(position: Int): DataStream[T]	在一个KeyedStream上滚动求最小值。min返回了最小值，不保证非最小值列的准确性。 position和field代表对某一列求最小值。
def min(field: String): DataStream[T]	

API	说明
def max(position: Int): DataStream[T]	在一个KeyedStream上滚动求最大值。max返回了最大值，不保证非最大值列的准确性。 position和field代表对某一列求最大值。
def max(field: String): DataStream[T]	
def minBy(position: Int): DataStream[T]	在一个KeyedStream上求某一列最小值所在的该行数据，minBy返回了该行数据的所有元素。 position和field代表对某一列做minBy操作。
def minBy(field: String): DataStream[T]	
def maxBy(position: Int): DataStream[T]	在一个KeyedStream上求某一列最大值所在的该行数据，maxBy返回了该行数据的所有元素。 position和field代表对某一列做maxBy操作。
def maxBy(field: String): DataStream[T]	

数据流分发能力

表 9-31 数据流分发能力的相关接口

API	说明
def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], field: Int) : DataStream[T]	使用一个用户自定义的Partitioner对每一个元素选择目标task。 • partitioner指的是用户自定义的分区类重写partition方法。 • field指的是partitioner的输入参数。 • keySelector指的是用户自定义的partitioner的输入参数。
def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], field: String):DataStream[T]	
def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], fun: T => K): DataStream[T]	
def shuffle: DataStream[T]	以均匀分布的形式将元素随机地进行分区。
def rebalance: DataStream[T]	基于round-robin对元素进行分区，使得每个分区负责均衡。对于存在数据倾斜的性能优化是很有用的。
def rescale: DataStream[T]	以round-robin的形式将元素分区到下游操作的子集中。 **说明** 查看代码和rebalance的方式是一样的。

API	说明
def broadcast: DataStream[T]	广播每个元素到所有分区。

提供设置 eventtime 属性的能力

表 9-32 提供设置 eventtime 属性的能力的相关接口

API	说明
def assignTimestampsAndWatermarks(assigner: AssignerWithPeriodicWatermarks[T]): DataStream[T]	为了能让event time窗口可以正常触发窗口计算操作，需要从记录中提取时间戳。
def assignTimestampsAndWatermarks(assigner: AssignerWithPunctuatedWatermarks[T]): DataStream[T]	

提供迭代的能力

表 9-33 提供迭代的能力的相关接口

API	说明
def iterate[R](stepFunction: DataStream[T] => (DataStream[T], DataStream[R]),maxWaitTimeMillis:Long = 0,keepPartitioning: Boolean = false) : DataStream[R]	在流(flow)中创建一个带反馈的循环，通过重定向一个operator的输出到之前的operator。 **说明** • 对于定义一些需要不断更新模型的算法是非常有帮助的。 • long maxWaitTimeMillis：该超时时间指的是每一轮迭代体执行的超时时间。
def iterate[R, F: TypeInformation](stepFunction: ConnectedStreams[T, F] => (DataStream[F], DataStream[R]),maxWaitTimeMillis:Long): DataStream[R]	

提供分流能力

表 9-34 提供分流能力的相关接口

API	说明
def split(selector: OutputSelector[T]): SplitStream[T]	传入OutputSelector，重写select方法确定分流的依据（即打标记），构建SplitStream流。即对每个元素做一个字符串的标记，作为选择的依据，打好标记之后就可以通过标记选出并新建某个标记的流。
def select(outputNames: String*): DataStream[T]	从一个SplitStream中选出一个或多个流。 outputNames指的是使用split方法对每个元素做的字符串标记的序列。

窗口能力

窗口分为跳跃窗口和滑动窗口。

- 支持Window、TimeWindow、CountWindow以及WindowAll、TimeWindowAll、CountWindowAll API窗口生成。
- 支持Window Apply、Window Reduce、Window Fold、Aggregations on windows API窗口操作。
- 支持多种Window Assigner（TumblingEventTimeWindows、TumblingProcessingTimeWindows、SlidingEventTimeWindows、SlidingProcessingTimeWindows、EventTimeSessionWindows、ProcessingTimeSessionWindows、GlobalWindows）。
- 支持三种时间ProcessingTime、EventTime和IngestionTime。
- 支持两种EventTime时间戳方式：AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks。

窗口生成类API如表9-35所示。

表 9-35 窗口生成类的相关接口

API	说明
def window[W <: Window](assigner: WindowAssigner[_ >: T, W]): WindowedStream[T, K, W]	窗口可以被定义在已经被分区的KeyedStreams上。窗口会对数据的每一个key根据一些特征（例如在最近5秒钟内到达的数据）进行分组。
def windowAll[W <: Window](assigner: WindowAssigner[_ >: T, W]): AllWindowedStream[T, W]	窗口可以被定义在DataStream上。

API	说明
def timeWindow(size: Time): WindowedStream[T, K, TimeWindow]	时间窗口定义在已经被分区的KeyedStreams上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 • size指的是窗口时间的大小。 • slide指的是窗口的滑动时间。 **说明** • WindowedStream和AllWindowedStream代表不同的两种流。 • 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
def countWindow(size: Long, slide: Long): WindowedStream[T, K, GlobalWindow]	
def timeWindowAll(size: Time): AllWindowedStream[T, TimeWindow]	时间窗口定义在DataStream上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 • size指的是窗口时间的大小。 • slide指的是窗口的滑动时间。
def timeWindowAll(size: Time, slide: Time): AllWindowedStream[T, TimeWindow]	
def countWindow(size: Long, slide: Long): WindowedStream[T, K, GlobalWindow]	按照元素个数区分窗口，定义在已经被分区的KeyedStreams上。 • size指的是窗口时间的大小。 • slide指的是窗口的滑动时间。 **说明** • WindowedStream和AllWindowedStream代表不同的两种流。 • 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
def countWindow(size: Long): WindowedStream[T, K, GlobalWindow]	
def countWindowAll(size: Long, slide: Long): AllWindowedStream[T, GlobalWindow]	按照元素个数区分窗口，定义在DataStream上。 • size指的是窗口时间的大小。 • slide指的是窗口的滑动时间。
def countWindowAll(size: Long): AllWindowedStream[T, GlobalWindow]	

窗口操作类API如**表9-36**所示。

表 9-36 窗口操作类的相关接口

方法	API	说明
Window	def apply[R: TypeInformation](function: WindowFunction[T, R, K, W]): DataStream[R]	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。 function指的是执行的窗口函数。
	def apply[R: TypeInformation](function: (K, W, Iterable[T], Collector[R]) => Unit): DataStream[R]	
	def reduce(function: ReduceFunction[T]): DataStream[T]	应用一个reduce函数到窗口上，返回reduce后的值。 ● reduceFunction指的是执行的reduce函数。 ● WindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。
	def reduce(function: (T, T) => T): DataStream[T]	
	def reduce[R: TypeInformation](preAggregator: ReduceFunction[T], function: WindowFunction[T, R, K, W]): DataStream[R]	
	def reduce[R: TypeInformation](preAggregator: (T, T) => T, windowFunction: (K, W, Iterable[T], Collector[R]) => Unit): DataStream[R]	
	def fold[R: TypeInformation](initialValue: R, function: FoldFunction[T,R]): DataStream[R]	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。
	def fold[R: TypeInformation](initialValue: R)(function: (R, T) => R): DataStream[R]	
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, foldFunction: FoldFunction[T, ACC], function: WindowFunction[ACC, R, K, W]): DataStream[R]	

方法	API	说明
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, foldFunction: (ACC, T) => ACC, windowFunction: (K, W, Iterable[ACC], Collector[R]) => Unit): DataStream[R]	
WindowAll	def apply[R: TypeInformation](function: AllWindowFunction[T, R, W]): DataStream[R]	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。
	def apply[R: TypeInformation](function: (W, Iterable[T], Collector[R]) => Unit): DataStream[R]	
	def reduce(function: ReduceFunction[T]): DataStream[T]	应用一个reduce函数到窗口上，返回reduce后的值。 ● reduceFunction指的是执行的reduce函数。 ● AllWindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。
	def reduce(function: (T, T) => T): DataStream[T]	
	def reduce[R: TypeInformation](preAggregator: ReduceFunction[T], windowFunction: AllWindowFunction[T, R, W]): DataStream[R]	
	def reduce[R: TypeInformation](preAggregator: (T, T) => T, windowFunction: (W, Iterable[T], Collector[R]) => Unit): DataStream[R]	
	def fold[R: TypeInformation](initialValue: R, function: FoldFunction[T,R]): DataStream[R]	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。
	def fold[R: TypeInformation](initialValue: R)(function: (R, T) => R): DataStream[R]	

方法	API	说明
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, preAggregator: FoldFunction[T, ACC], windowFunction: AllWindowFunction[ACC, R, W]): DataStream[R]	
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, preAggregator: (ACC, T) => ACC, windowFunction: (W, Iterable[ACC], Collector[R]) => Unit): DataStream[R]	
Window 和 Window All	def sum(position: Int): DataStream[T]	对窗口数据的某一列求和。 position和field代表数据的某一列。
	def sum(field: String): DataStream[T]	
	def min(position: Int): DataStream[T]	对窗口数据的某一列求最小值。min返回了最小值，不保证非最小值列的准确性。 position和field代表对某一列求最小值。
	def min(field: String): DataStream[T]	
	def max(position: Int): DataStream[T]	对窗口数据的某一列求最大值。max返回了最大值，不保证非最大值列的准确性。 position和field代表对某一列求最大值。
	def max(field: String): DataStream[T]	
	def minBy(position: Int): DataStream[T]	对窗口数据的某一列求最小值所在的该行数据，minBy返回了该行数据的所有元素。 position和field代表对某一列做minBy操作。
	def minBy(field: String): DataStream[T]	
	def maxBy(position: Int): DataStream[T]	对窗口数据的某一列求最大值所在的该行数据，maxBy返回了该行数据的所有元素。 position和field代表对某一列做maxBy操作。
	def maxBy(field: String): DataStream[T]	

提供多流合并的能力

表 9-37 提供多流合并的能力的相关接口

API	说明
def union(dataStreams: DataStream[T]*): DataStream[T]	Union两个或多个数据流，生成一个新的包含了来自所有流的所有数据的数据流。 **说明** 如果你将一个数据流与其自身进行了合并，在结果流中对于每个元素你都会拿到两份。
def connect[T2](dataStream: DataStream[T2]): ConnectedStreams[T, T2]	“连接”两个数据流并保持原先的类型。Connect可以让两条流之间共享状态。产生ConnectedStreams之后，调用map或者flatmap进行操作计算。
def map[R: TypeInformation](coMapper: CoMapFunction[IN1, IN2, R]): DataStream[R]	在ConnectedStreams上做元素映射，类似DataStream的map操作，元素映射之后流数据类型统一。
def map[R: TypeInformation](fun1: IN1 => R, fun2: IN2 => R): DataStream[R]	
def flatMap[R: TypeInformation](coFlatMapper: CoFlatMapFunction[IN1, IN2, R]): DataStream[R]	在ConnectedStreams上做元素映射，类似DataStream的flatMap操作，元素映射之后流数据类型统一。
def flatMap[R: TypeInformation](fun1: (IN1, Collector[R]) => Unit, fun2: (IN2, Collector[R]) => Unit): DataStream[R]	
def flatMap[R: TypeInformation](fun1·IN1 => TraversableOnce[R], fun2: IN2 => TraversableOnce[R]): DataStream[R]	

提供 Join 能力

表 9-38 提供 Join 能力的相关接口

API	说明
def join[T2] (otherStream: DataStream[T2]): JoinedStreams[T, T2]	通过给定的key在一个窗口范围内join两条数据流。 join操作的key值通过where和eaualTo方法进行指定，代表两条流过滤出包含等值条件的数据。
def coGroup[T2] (otherStream: DataStream[T2]): CoGroupedStreams[T, T2]	通过给定的key在一个窗口范围内co-group两条数据流。 coGroup操作的key值通过where和eaualTo方法进行指定，代表两条流通过该等值条件进行分区处理。

9.6.1.3 Flink REST API 接口介绍

Flink具有可用于查询正在运行的作业的状态和统计信息以及最近完成作业的监视API。该监视API由Flink自己的WEB UI使用。

监视API是REST API，可接受HTTP GET请求并使用JSON数据进行响应。REST API是访问Web服务器的一套API。当前在Flink中，Web服务器是JobManager的一个模块，和JobManager共进程。默认情况下，web服务器监测的端口是8081，用户可以在配置文件“flink-conf.yaml”中配置“jobmanager.web.port”来修改监测端口。

使用Netty和Netty路由器库来处理REST请求和解析URL。

REST API接口的执行方式是通过HTTP请求进行。

HTTP请求的格式为：http://<JobManager_IP>:<JobManager_Port><Path>，

其中JobManager_IP是指JobManager进程所在服务器节点的IP地址，JobManager_Port是指JobManager进程的监测端口，Path为路径的部分，参见**表9-39**。例如：http://10.162.181.57:32261/config。

说明

需要修改Flink Client的配置文件“flink-conf.yaml”，在“jobmanager.web.allow-access-address”和“jobmanager.web.access-control-allow-origin”中添加访问主机的IP地址，可使用逗号分隔。

Flink支持的所有REST API的URL中的Path信息如**表9-39**所示。

表 9-39 Path 介绍

Path	说明
/config	有关监控API和服务器设置的一些信息。
/logout	注销的重要信息。
/overview	Flink集群状态的简单概要。

Path	说明
/jobs	Job的ID，按运行，完成，失败和取消等状态进行分组。
/jobmanager/config	JobManager的配置。
/joboverview	业务按状态进行分组，每个业务组都有一个小状态。
/joboverview/running	与“/ joboverview”相同，Job按状态进行分组，每个Job组都有一个小状态，但只包含当前运行的Job。
/joboverview/completed	Job按状态进行分组，每个都有一个小状态的摘要。 与“/ joboverview”相同，但仅包含已完成，已取消或失败的Job。
/jobs/<jobid>	一个Job主要信息包含列出数据流计划，状态，状态转换的时间戳，每个顶点（运算符）的聚合信息。
/jobs/<jobid>/vertices	目前与“/jobs/<jobid>”相同。
/jobs/<jobid>/config	Job使用用户定义的执行配置。
/jobs/<jobid>/exceptions	Job探察到不可恢复的异常。截取的标识提示是否存在更多异常，但不列出这些异常，否则回复会太大。
/jobs/<jobid>/accumulators	聚合用户累加器加上Job累加器。
/jobs/<jobid>/checkpoints	Job的checkpoint的统计信息。
/jobs/<jobid>/metrics	一个Job的所有可用指标。
/jobs/<jobid>/vertices/<vertexid>	关于流图的顶点下每个子任务的信息。
/jobs/<jobid>/vertices/<vertexid>/subtasktimes	请求返回流图的顶点的所有子任务状态转换的时间戳。 这些可以用于在子任务之间创建时间线的比较。
/jobs/<jobid>/vertices/<vertexid>/taskmanagers	一个流图顶点的TaskManager统计信息。 这是“/ jobs / <jobid> / vertices / <vertexid>”返回的子任务统计信息的聚合。
/jobs/<jobid>/vertices/<vertexid>/accumulators	聚合的用户定义的累加器，用于流图顶点。
/jobs/<jobid>/vertices/<vertexid>/checkpoints	单个Job顶点的检查点统计信息。
/jobs/<jobid>/vertices/<vertexid>/backpressure	单个Job顶点的背压统计数据及其所有子任务。
/jobs/<jobid>/vertices/<vertexid>/metrics	一组指标值的给定任务。

Path	说明
/jobs/<jobid>/vertices/<vertexid>/subtasks/accumulators	获取流图顶点的所有子任务的所有用户定义的累加器。 这些是通过请求“/ jobs / <jobid> / vertices / <vertexid> / accumulators”以聚合形式返回的各个累加器。
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>	特定子任务的当前或最近执行尝试的摘要。
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>/attempts/<attempt>	特定子任务的具体执行尝试的摘要。 发生故障/恢复时会发生多次执行尝试。
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>/attempts/<attempt>/accumulators	在一次特定的执行尝试期间，为一个特定子任务收集累加器（在发生故障/恢复时会发生多次尝试）。
/jobs/<jobid>/plan	Job的数据流计划。 该计划也包括在Job摘要（“/ jobs / <jobid>”）中。
/taskmanagers	任务管理员信息。
/taskmanagers/<taskmanagerid>/metrics	任务管理员的度量信息。
/taskmanagers/<taskmanagerid>/log	任务管理员的日志信息。
/taskmanagers/<taskmanagerid>/stdout	一个任务管理员的标准。
/jobmanager/log	JobManager的日志信息。
/jobmanager/stdout	JobManager的标准。
/jobmanager/metrics	JobManager的指标。
/:*	对Web前端的静态文件（如HTML，CSS或JS文件）的请求。

表9-39中变量的介绍请参见表9-40。

表 9-40 变量说明

变量	说明
jobid	job的id。
vertexid	流图的顶点id。
subtasknum	子任务的总和。
attempt	尝试。

变量	说明
taskmanagerid	任务管理的id。

9.6.1.4 Flink Savepoints CLI 介绍

概述

Savepoints在持久化存储中保存某个checkpoint，以便用户可以暂停自己的应用进行升级，并将状态设置为savepoint的状态，并继续运行。该机制利用了Flink的checkpoint机制创建流应用的快照，并将快照的元数据（meta-data）写入到一个额外的持久化文件系统中。

如果需要使用savepoints的功能，强烈推荐用户为每个算子通过uid(String)分配一个固定的ID，以便将来升级恢复使用，示例代码如下：

```
DataStream<String> stream = env
// Stateful source (e.g. Kafka) with ID
.addSource(new StatefulSource())
.uid("source-id") // ID for the source operator
.shuffle()
 // Stateful mapper with ID
.map(new StateFulMapper())
.uid("mapper-id") // ID for the mapper
// Stateless printing sink
.print(); //Auto-generated ID
```

savepoint 恢复

如果用户不手动设置ID，系统将自动给每个算子分配一个ID。只要该算子的ID不改变，即可从savepoint恢复，ID的产生取决于用户的应用代码，并且对应用代码的结构十分敏感。因此，强烈推荐用户手动为每个算子设置ID。Savepoint产生的数据将被保存到配置的文件系统中，如FsStateBackend或者RocksDBStateBackend。

1. 触发一个savepoint

   ```
   $ bin/flink savepoint <jobId> [targetDirectory]
   ```

   以上命令将触发ID为jobId的作业产生一个savepoint，另外，用户可以通过targetDirectory指定savepoint的存储路径，该路径必须是jobManager可以访问的，由于targetDirectory是可选的，如果用户没有配置targetDirectory，则是使用配置文件中“state.savepoints.dir”配置的目录来存放savepoint。

   用户可以在“flink-conf.yaml”中通过“state.savepoints.dir”选项设置默认的savepoint路径。

   ```
   # Default savepoint target directory
   ```

   说明

   建议用户将targetDirectory路径设置为HDFS路径 ，例如：

   ```
   bin/flink savepoint  405af8c02cf6dc069a0f9b7a1f7be088 hdfs://savepoint
   ```

2. 删除一个作业并进行savepoint

   ```
   $ bin/flink cancel -s [targetDirectory] jobId
   ```

   以上命令将删除一个作业，同时，在删除前将对该作业的状态进行保存。另外，用户可以通过targetDirectory指定savepoint的存储路径，该路径必须是jobManager可以访问的。

3. 恢复作业方式
   - 从savepoint恢复作业。
     ```
     $ bin/flink run -s savepointPath [runArgs]
     ```
     以上命令将提交一个作业，并将该作业的初始状态置为savepointPath指定的状态。

     说明

     runArgs是指用户应用中自定义的参数，每个用户自定义的参数形式、名称都不一样。
   - 允许不恢复某个算子的状态
     ```
     $ bin/flink run -s savepointPath -n [runArgs]
     ```
     默认情况下，系统将尝试将savepoint的状态全部映射到用户的流应用中，如果用户升级的流应用删除了某个算子，可以通过--allowNonRestoredState(简写-n)恢复状态。
4. 清除savepoints
   ```
   $ bin/flink savepoint -d savepointPath
   ```
   以上命令将删除保存在savepointPath的savepoint。

注意事项

- 如果一个task中有算子链（Chained operators），将会将算子链上第一个算子的ID分配给该task。给算子链上的中间算子手动分配ID是不可能的。例如：在链（Chain）[a->b->c]中，只能给a手动分配ID，b和c不能分配。如果用户想给b和c分配ID，用户必须手动建链。手动建链时需要使用disableChaining()接口。举例如下：
  ```
  env.addSource(new GetDataSource())
  .keyBy(0)
  .timeWindow(Time.seconds(2)).uid("window-id")
  .reduce(_+_).uid("reduce-id")
  .map(f=>(f,1)).disableChaining().uid("map-id")
  .print().disableChaining().uid("print-id")
  ```
- 用户升级job时不允许更改算子的数据类型。

9.6.1.5 Flink Client CLI 介绍

常用 CLI

Flink常用的CLI如下所示：

1. yarn-session.sh
   - 可以使用yarn-session.sh启动一个常驻的Flink集群，接受来自客户端提交的任务。启动一个有3个TaskManager实例的Flink集群示例如下：
     ```
     bin/yarn-session.sh
     ```
   - yarn-session.sh的其他参数可以通过以下命令获取：
     ```
     bin/yarn-session.sh -help
     ```
2. Flink
   - 使用flink命令可以提交Flink作业，作业既可以被提交到一个常驻的Flink集群上，也可以使用单机模式运行。
     - 提交到常驻Flink集群上的一个示例如下：
       ```
       bin/flink run ../examples/streaming/WindowJoin.jar
       ```

说明

用户在用该命令提交任务前需要先用yarn-session启动Flink集群。

- 以yarn-cluster模式运行作业的一个示例如下：
  ```
  bin/flink run -m yarn-cluster ../examples/streaming/WindowJoin.jar
  ```

  说明

  通过参数-m yarn-cluster使作业以yarn-cluster模式运行，该模式为指定作业单独启动一个Flink集群来执行。

- 列出所有的作业（包含JobID）：
  ```
  bin/flink list
  ```
- 取消作业：
  ```
  bin/flink cancel <JobID>
  ```
- 停止作业（仅流式作业）：
  ```
  bin/flink stop <JobID>
  ```

  说明

  取消和停止作业的区别如下：

  - 取消作业：执行“cancel”命令时，指定作业会立即收到cancel()方法调用ASAP。如果调用结束后作业仍然没有停止，Flink会定期开始中断执行线程直至作业停止。
  - 停止作业：“stop”命令仅适用于Flink源（source）实现了StoppableFunction接口的作业。“stop”命令会等待所有资源都正确关闭。相比“cancel”命令，“stop”停止作业的方式更为优雅，但可能导致停止作业失败。

– flink脚本的其他参数可以通过以下命令获取：
  ```
  bin/flink --help
  ```

注意事项

- 如果yarn-session.sh使用-z配置特定的zookeeper的namespace，则在使用flink run时必须使用-yid指出applicationID，使用-yz指出zookeeper的namespace，前后namespace保持一致。

  举例：

  ```
  bin/yarn-session.sh -z YARN101
  bin/flink run -yid application_****_**** -yz YARN101 examples/streaming/WindowJoin.jar
  ```

- 如果yarn-session.sh不使用-z配置特定的zookeeper的namespace，则在使用flink run时不要使用-yz指定特定的zookeeper的namespace。

  举例：

  ```
  bin/yarn-session.sh
  bin/flink run examples/streaming/WindowJoin.jar
  ```

- 如果使用flink run -m yarn-cluster时启动集群则可以使用-yz指定一个zookeeper的namespace。
- 不能同时启动两个或两个以上的集群来共享一个namespace。
- 用户在启动集群或提交作业时如果使用了-z配置项，则在删除、停止及查询作业、触发savepoint时也要使用-z配置项指明namespace。

9.6.2 如何处理用户在使用 chrome 浏览器时无法显示任务状态的 title

问题

用户在使用chrome浏览器浏览Flink Web UI页面时无法显示title。此处以Tasks为例进行分析，用户将鼠标置于Tasks的彩色小方框上，无法显示彩色小框的title说明，如图9-68所示。正常的显示界面如图9-69所示。

图 9-68 界面无法显示 title

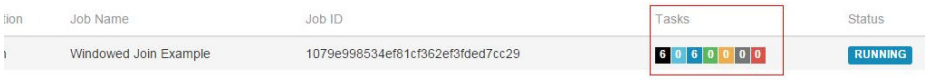
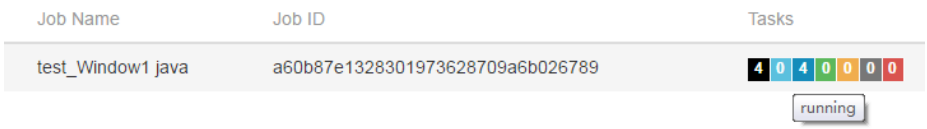


图 9-69 界面正常显示 title



回答

如果用户遇到chrome浏览器无法显示title，步骤如下：

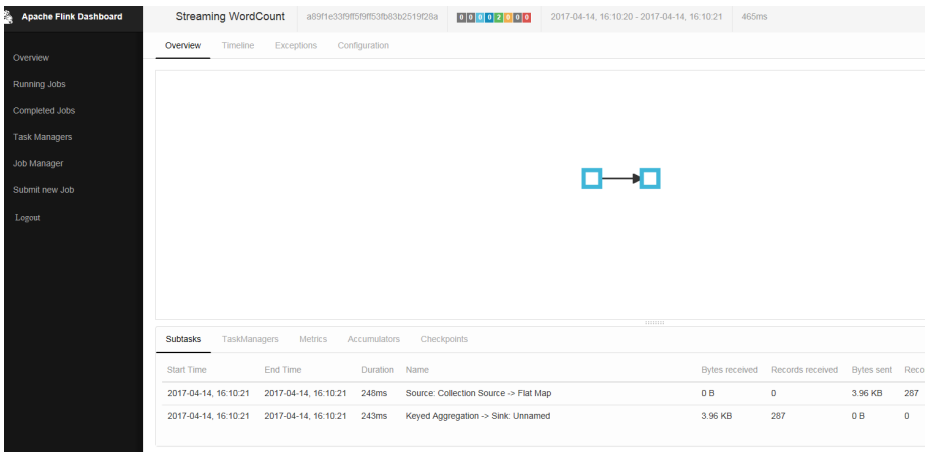
请检查本机是否同时运行会影响chrome浏览器冒泡提示的工具，如果运行了，需要将该工具关闭。

9.6.3 如何处理 IE10/11 页面算子的文字部分显示异常

问题

如何处理IE10/11页面显示异常，每个算子的文字部分没有显示出来的问题？如图9-70所示，Overview显示为空白。

图 9-70 页面显示异常



回答

Flink中用了foreignObject元素来代理绘制svg矢量图，但是IE 10/11不支持foreignObject导致算子显示异常。推荐使用chrome浏览器。

9.6.4 如何处理 Checkpoint 设置 RocksDBStateBackend 方式时 Checkpoint 慢

问题

如何处理checkpoint设置RocksDBStateBackend方式，且当数据量大时，执行checkpoint会很慢的问题?

原因分析

由于窗口使用自定义窗口，这时窗口的状态使用ListState，且同一个key值下，value的值非常多，每次新的value值到来都要使用RocksDB的merge()操作；触发计算时需要将该key值下所有的value值读出。

- RocksDB的方式为merge()->merge()....->merge()->read()，该方式读取数据时非常耗时，如**图9-71**所示。
- source算子在瞬间发送了大量数据，所有数据的key值均相等，导致window算子处理速度过慢，使barrier在缓存中积压，快照的制作时间过长，导致window算子在规定时间内没有向CheckpointCoordinator报告快照制作完成，CheckpointCoordinator认为快照制作失败，如**图9-72**所示。

图 9-71 时间监控信息

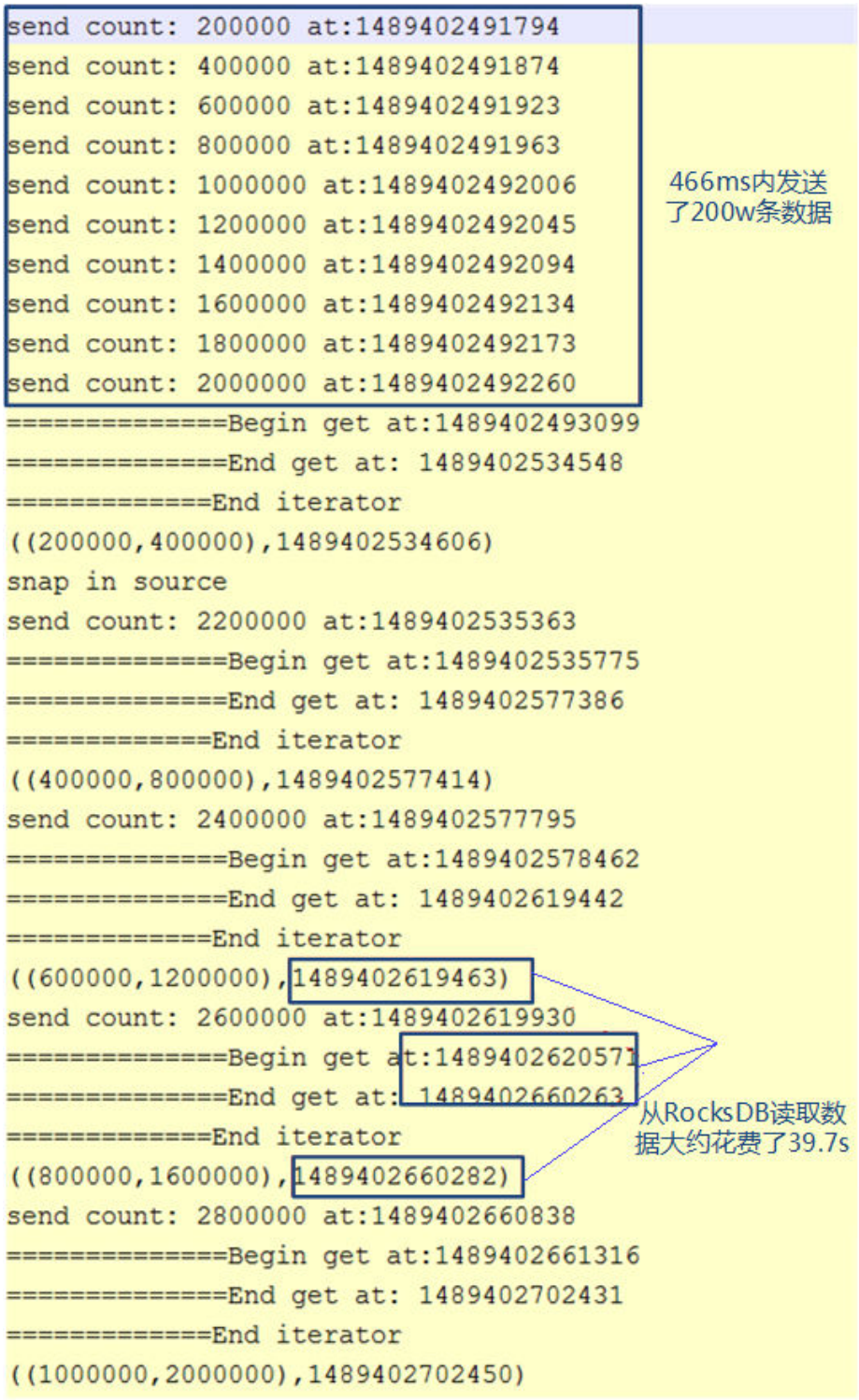
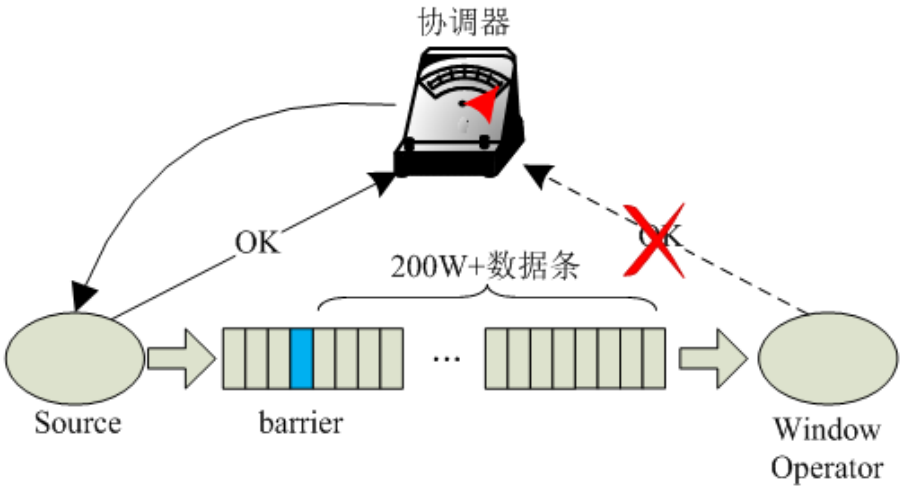


图 9-72 关系图



回答

Flink引入了第三方软件包RocksDB的缺陷问题导致该现象的发生。建议用户将checkpoint设置为FsStateBackend方式。

用户需要在应用代码中将checkpoint设置为FsStateBackend。例如：

```
env.setStateBackend(new FsStateBackend("hdfs://hacluster/flink/checkpoint/"));
```

9.6.5 如何处理 blob.storage.directory 配置/home 目录时启动 yarn-session 失败

问题

当用户设置“blob.storage.directory”为“/home”时，用户没有权限在“/home”下创建“blobStore-UUID”的文件，导致yarn-session启动失败。

回答

步骤1 建议将"blob.storage.directory"配置选项设置成“/tmp”或者“/opt/huawei/Bigdata/tmp”。

步骤2 当用户将"blob.storage.directory"配置选项设置成自定义目录时，需要手动赋予用户该目录的owner权限。以下以FusionInsight的admin用户为例。

1. 修改Flink客户端配置文件conf/flink-conf.yaml，配置blob.storage.directory: /home/testdir/testdirdir/xxx。
2. 创建目录/home/testdir（创建一层目录即可），设置该目录为admin用户所属。

图 9-73 创建目录

```
SZV1000064084:/home # id admin
uid=20000(admin) gid=9998(ficommon) groups=9998(ficommon),8003(System_administrator_186)
SZV1000064084:/home # chown admin:ficommon testdir/ -R
```

说明

/home/testdir/下的testdirdir/xxx目录在启动Flink集群时会在每个节点下自动创建。

3. 进入客户端路径，执行命令**./bin/yarn-session.sh -jm 2048 -tm 3072**，可以看到yarn-session正常启动并且成功创建目录。

图 9-74 执行命令

----结束

9.6.6 如何处理非 static 的 KafkaPartitioner 类对象构造 FlinkKafkaProducer010 运行时报错

问题

Flink内核升级到1.3.0之后，当Kafka调用带有非static的KafkaPartitioner类对象为参数的FlinkKafkaProducer010去构造函数时，运行时会报错。

报错内容如下：

```
org.apache.flink.api.common.InvalidProgramException: The implementation of the FlinkKafkaPartitioner is not serializable. The object probably contains or references non serializable fields.
```

回答

Flink的1.3.0版本，为了兼容原有那些使用KafkaPartitioner的API接口，如FlinkKafkaProducer010带KafkaPartitioner对象的构造函数，增加了FlinkKafkaDelegatePartitioner类。

该类定义了一个成员变量，即kafkaPartitioner：

```
private final KafkaPartitioner<T> kafkaPartitioner;
```

当Flink传入参数是KafkaPartitioner去构造FlinkKafkaProducer010时，调用栈如下：

```
FlinkKafkaProducer010(String topicId, KeyedSerializationSchema<T> serializationSchema, Properties producerConfig, KafkaPartitioner<T> customPartitioner)
-> FlinkKafkaProducer09(String topicId, KeyedSerializationSchema<IN> serializationSchema, Properties producerConfig, FlinkKafkaPartitioner<IN> customPartitioner)
----> FlinkKafkaProducerBase(String defaultTopicId, KeyedSerializationSchema<IN> serializationSchema, Properties producerConfig, FlinkKafkaPartitioner<IN> customPartitioner)
------> ClosureCleaner::clean(Object func, boolean checkSerializable)
```

首先使用KafkaPartitioner对象去构造一个FlinkKafkaDelegatePartitioner对象，然后再检查该对象是否可序列化。由于ClosureCleaner::clean函数是static函数，当用例中的KafkaPartitioner对象是非static时，ClosureCleaner::clean函数无法访问KafkaDelegatePartitioner类内的非static成员变量kafkaPartitioner，导致报错。

解决方法如下，两者任选其一：

- 将KafkaPartitioner类改成static类。

- 改用以FlinkKafkaPartitioner为参数的FlinkKafkaProducer010构造函数，内部实现不会去构造FlinkKafkaDelegatePartitioner，也就不会存在成员变量的问题。

9.6.7 如何处理新创建的 Flink 用户提交任务报 ZooKeeper 文件目录权限不足

问题

创建一个新的Flink用户，提交任务，ZooKeeper目录无权限导致提交Flink任务失败，日志中报如下错误：

```
NoAuth for /flink_base/flink/application_1499222480199_0013
```

回答

1. 首先查看ZooKeeper中/flink_base的目录权限是否为：'world,'anyone: cdrwa；如果不是，请修改/flink_base的目录权限为：'world,'anyone: cdrwa，然后继续根据**步骤二**排查；如果是，请根据**步骤二**排查。
2. 由于在Flink配置文件中“high-availability.zookeeper.client.acl”默认为“creator”，即谁创建谁有权限，由于原有用户已经使用ZooKeeper上的/flink_base/flink目录，导致新创建的用户访问不了ZooKeeper上的/flink_base/flink目录。

   新用户可以通过以下操作来解决问题。

   a. 查看客户端的配置文件“conf/flink-conf.yaml”。
   b. 修改配置项“high-availability.zookeeper.path.root”对应的ZooKeeper目录，例如：/flink2。
   c. 重新提交任务。

9.6.8 如何处理无法直接通过 URL 访问 Flink Web

问题

无法通过“http://JobManager IP:JobManager的端口”访问Web页面。

回答

由于浏览器所在的计算机IP地址未加到Web访问白名单导致。用户可以通过修改客户端的配置文件“conf/flink-conf.yaml”来解决问题。

1. 确认配置项“jobmanager.web.ssl.enabled”的值是否是“false”，若不是，请修改为“false”。
2. 确认配置项“jobmanager.web.access-control-allow-origin”和“jobmanager.web.allow-access-address”中是否已经添加浏览器所在的计算机IP地址。如果没有添加，可以通过这两项配置项进行添加。例如：

   ```
   jobmanager.web.access-control-allow-origin: 浏览器所在的计算机IP地址
   jobmanager.web.allow-access-address: 浏览器所在的计算机IP地址
   ```

9.6.9 如何查看 System.out.println 打印的调试信息或将调试信息输出至指定文件

问题

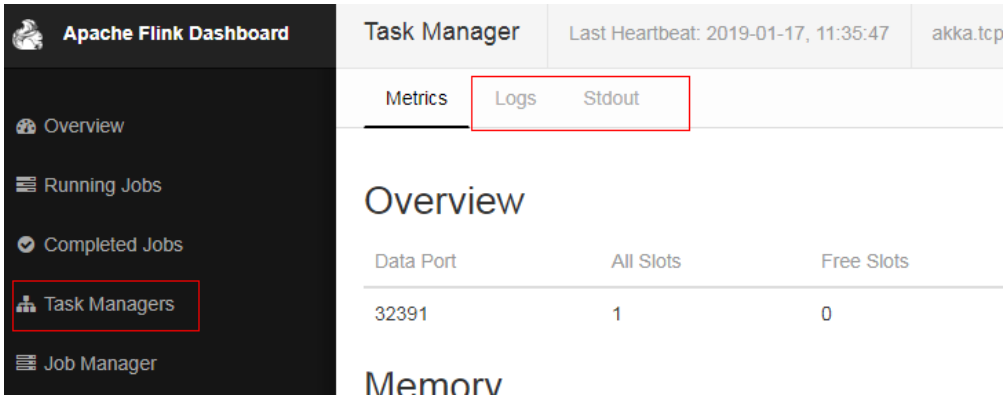
Flink业务代码中添加了System.out.println调试信息打印，该如何查看此调试日志？避免与运行日志混在一起打印，如何将业务日志单独输出至指定文件？

回答

Flink所有的运行日志打印都会打印至Yarn的本地目录下，默认所有Log都会输出至Yarn container本地目录下taskmanager.log，所有调用System.out打印都会输出值taskmanager.out文件。查看方式如下：

1. 进入Flink原生Web页面。
2. 在左侧导航栏单击“Task Managers”，可在“Logs”或“Stdout”页签查看日志信息。

图 9-75 查看日志信息



配置业务日志与TaskManager运行日志独立打印：

说明

若配置业务日志与TaskManager运行日志分开打印后，业务日志不输出至taskmanager.log，无法使用Web页面进行查看相应日志信息。

1. 修改客户端的配置文件“conf/logback.xml”，在文件中添加如下日志配置信息，加粗标注部分根据需要进行修改。

```
<appender name="TEST" class="ch.qos.logback.core.rolling.RollingFileAppender">
   <file>/path/test.log</file>
   <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
      <fileNamePattern>/path/test.log.%i</fileNamePattern>
      <minIndex>1</minIndex>
      <maxIndex>20</maxIndex>
   </rollingPolicy>
   <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
      <maxFileSize>20MB</maxFileSize>
   </triggeringPolicy>
   <encoder>
      <pattern>%d{"yyyy-MM-dd HH:mm:ss,SSS"} | %m %n</pattern>
   </encoder>
</appender>

<logger name="com.huawei.bigdata.flink.examples" additivity="false">
```

```
    <level value="INFO"/>
    <appender-ref ref="TEST"/>
  </logger>
```

2. 重新启动yarn-session.sh，提交任务。

说明

自定义日志若指定了路径**<file>/path/test.log</file>**，需确保任务运行所使用的用户（**flink-conf.yaml**配置用户）有权限对该目录进行读写操作。

9.6.10 如何处理 Flink 任务配置 State Backend 为 RocksDB 时报错 GLIBC 版本问题

问题

Flink任务配置State Backend为RocksDB时，运行报如下错误：

```
Caused by: java.lang.UnsatisfiedLinkError: /srv/BigData/hadoop/data1/nm/usercache/***/appcache/
application_****/rocksdb-lib-****/librocksdbjni-linux64.so: /lib64/libpthread.so.0: version `GLIBC_2.12` not
found (required by /srv/BigData/hadoop/***/librocksdbjni-linux64.so)
at java.lang.ClassLoader$NativeLibrary.load(Native Method)
at java.lang.ClassLoader.loadLibrary0(ClassLoader.java:1965)
at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1890)
at java.lang.Runtime.load0(Runtime.java:795)
at java.lang.System.load(System.java:1062)
at org.rocksdb.NativeLibraryLoader.loadLibraryFromJar(NativeLibraryLoader.java:78)
at org.rocksdb.NativeLibraryLoader.loadLibrary(NativeLibraryLoader.java:56)
at
org.apache.flink.contrib.streaming.state.RocksDBStateBackend.ensureRocksDBIsLoaded(RocksDBStateBacken
d.java:734)
... 11 more
```

可能原因

运行的系统和编译环境所在的系统版本不同，造成GLIBC的版本不兼容。

定位思路

使用**strings /lib64/libpthread.so.0 | grep GLIBC**命令查看GLIBC是否版本低于2.12。

处理步骤

如果GLIBC版本太低，则需要使用含有较高版本的（此处为2.12）的文件替换掉"libpthread-*.so"（注意，这是一个链接文件，执行时只需要替换掉它所指向的文件即可）。

参考信息

无

10 Flink 开发指南（普通模式）

10.1 Flink 应用开发简介

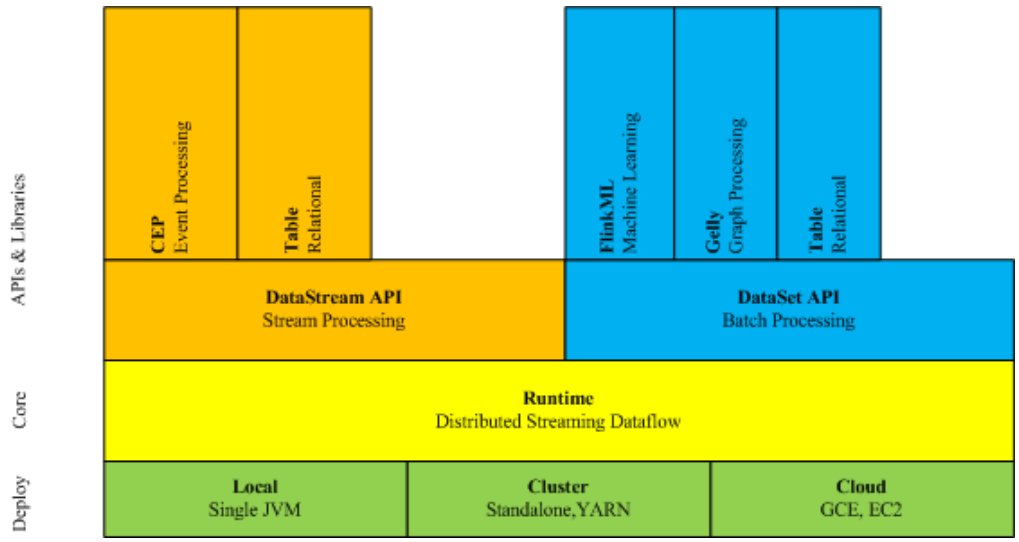
简介

Flink是一个批处理和流处理结合的统一计算框架，其核心是一个提供了数据分发以及并行化计算的流数据处理引擎。它的最大亮点是流处理，是业界最顶级的开源流处理引擎。

Flink最适合的应用场景是低时延的数据处理（Data Processing）场景：高并发pipeline处理数据，时延毫秒级，且兼具可靠性。

Flink技术栈如图10-1所示。

图 10-1 Flink 技术栈



Flink在当前版本中重点构建如下特性，其他特性继承开源社区，不做增强。

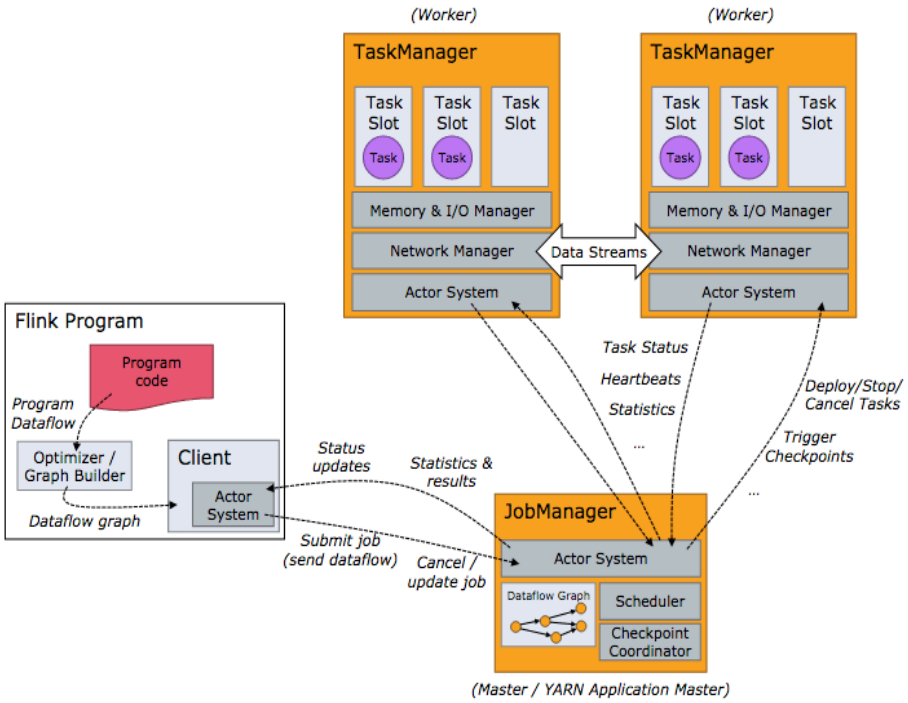
- DataStream
- Checkpoint

- 窗口
- Job Pipeline
- 配置表

架构

Flink架构如图10-2所示。

图 10-2 Flink 架构



Flink整个系统包含三个部分：

- Client

  Flink Client主要给用户提供向Flink系统提交用户任务（流式作业）的能力。

- TaskManager

  Flink系统的业务执行节点，执行具体的用户任务。TaskManager可以有多个，各个TaskManager都平等。

- JobManager

  Flink系统的管理节点，管理所有的TaskManager，并决策用户任务在哪些Taskmanager执行。JobManager在HA模式下可以有多个，但只有一个主JobManager。

Flink系统提供的关键能力：

- 低时延

  提供ms级时延的处理能力。

- Exactly Once

提供异步快照机制，保证所有数据真正只处理一次。

- HA

  JobManager支持主备模式，保证无单点故障。

- 水平扩展能力

  TaskManager支持手动水平扩展。

Flink 开发接口简介

Flink DataStream API提供Scala和Java两种语言的开发方式，如表10-1所示。

表 10-1 Flink DataStream API 接口

功能	说明
Scala API	提供Scala语言的API，提供过滤、join、窗口、聚合等数据处理能力。由于Scala语言的简洁易懂，推荐用户使用Scala接口进行程序开发。
Java API	提供Java语言的API，提供过滤、join、窗口、聚合等数据处理能力。

基本概念

- **DataStream**

  数据流，是指Flink系统处理的最小数据单元。该数据单元最初由外部系统导入，可以通过socket、Kafka和文件等形式导入，在Flink系统处理后，在通过Socket、Kafka和文件等输出到外部系统，这是Flink的核心概念。

- **Data Transformation**

  数据处理单元，会将一或多个DataStream转换成一个新的DataStream。

  具体可以细分如下几类：

  - 一对一的转换：如Map。
  - 一对0、1或多个的转换：如FlatMap。
  - 一对0或1的转换，如Filter。
  - 多对1转换，如Union。
  - 多个聚合的转换，如window、keyby。

- **CheckPoint**

  CheckPoint是Flink数据处理高可靠、最重要的机制。该机制可以保证应用在运行过程中出现失败时，应用的所有状态能够从某一个检查点恢复，保证数据仅被处理一次（Exactly Once）。

- **SavePoint**

  Savepoint是指允许用户在持久化存储中保存某个checkpoint，以便用户可以暂停自己的任务进行升级。升级完后将任务状态设置为savepoint存储的状态开始恢复运行，保证数据处理的延续性。

样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Flink相关样例工程，安全模式路径为“flink-examples/flink-examples-security”，普通模式路径为“flink-examples/flink-examples-normal”：

表 10-2 Flink 相关样例工程

样例工程	描述
FlinkCheckpointJavaExample FlinkCheckpointScalaExample	异步Checkpoint机制程序的应用开发示例。 假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性，即：当应用出现异常并恢复后，各个算子的状态能够处于统一的状态，相关业务场景介绍请参见**Flink开启Checkpoint样例程序**。
FlinkKafkaJavaExample FlinkKafkaScalaExample	向Kafka生产并消费数据程序的应用开发示例。 通过调用flink-connector-kafka模块的接口，生产并消费数据，相关业务场景介绍请参见**Flink Kafka样例程序**。
FlinkPipelineJavaExample FlinkPipelineScalaExample	Job Pipeline程序的应用开发示例。 相关业务场景介绍请参见**Flink Job Pipeline样例程序**。 发布者Job自己每秒钟产生10000条数据，然后经由该job的NettySink算子向下游发送。另外两个Job作为订阅者，分别订阅一份数据并打印输出。
FlinkSqlJavaExample	使用客户端通过jar作业提交SQL作业的应用开发示例。
FlinkStreamJavaExample FlinkStreamScalaExample	DataStream程序的应用开发示例。 相关业务场景介绍请参见**Flink DataStream样例程序**。 假定用户有某个网站周末网民网购停留时间的日志文本，另有一张网民个人信息的csv格式表，可通过Flink应用程序实现例如实时统计总计网购时间超过2个小时的女性网民信息，包含对应的个人详细信息的功能。
FlinkStreamSqlJoinExample	Stream SQL Join程序的应用开发示例。 相关业务场景介绍请参见**Flink Join样例程序**。 假定某个Flink业务1每秒就会收到1条消息记录，消息记录某个用户的基本信息，包括名字、性别、年龄。另有一个Flink业务2会不定时收到1条消息记录，消息记录该用户的名字、职业信息。实现实时的以根据业务2中消息记录的用户名字作为关键字，对两个业务数据进行联合查询的功能。

10.2 Flink 应用开发流程介绍

Flink 应用程序开发流程

Flink开发流程参考如下步骤：

图 10-3 Flink 应用程序开发流程

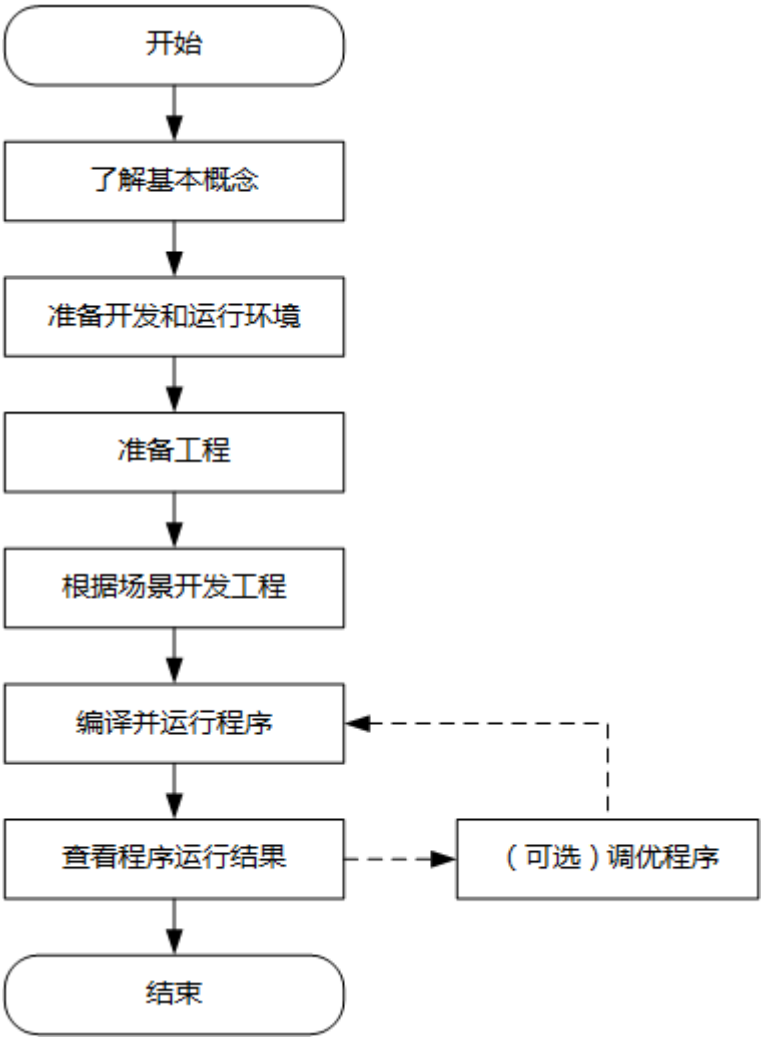


表 10-3 Flink 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Flink的基本概念。	基本概念
准备开发和运行环境	Flink的应用程序支持使用Scala、Java两种语言进行开发。推荐使用IDEA工具，请根据指导完成不同语言的开发环境配置。Flink的运行环境即Flink客户端，请根据指导完成客户端的安装和配置。	准备本地应用开发环境

阶段	说明	参考文档
准备工程	Flink提供了样例程序，您可以导入样例工程进行程序学习。或者您可以根据指导，新建一个Flink工程。	**导入并配置Flink样例工程**
根据场景开发工程	提供了Scala、Java两种不同语言的样例工程，帮助用户快速了解Flink各部件的编程接口。	**开发Flink应用**
编译并运行程序	指导用户将开发好的程序编译并提交运行。	**编译并调测Flink应用**
查看程序运行结果	程序运行结果会写在用户指定的路径下，用户还可以通过UI查看应用运行情况。	**查看Flink应用调测结果**
调优程序	您可以根据程序运行情况，对程序进行调优，使其性能满足业务场景需求。 调优完成后，请重新进行编译和运行。	组件操作指南中的“Flink性能调优”

10.3 准备 Flink 应用开发环境

10.3.1 准备本地应用开发环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如**表10-4**所示。

表 10-4 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统。 • 运行环境：Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。
安装和配置IDEA	用于开发Flink应用程序的工具。版本要求：14.1.7。
安装Scala	Scala开发环境的基本配置。版本要求：2.11.7。
安装Scala插件	Scala开发环境的基本配置。版本要求：1.5.4。

准备项	说明
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果使用Linux环境调测程序，需在准备安装集群客户端的Linux节点并获取相关配置文件。
  a. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。
     客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。
     集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。
  b. 登录FusionInsight Manager页面，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig\Flink\config”路径下的所有配置文件至客户端节点，放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。
     例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：
     cd /tmp/FusionInsight-Client
     tar -xvf FusionInsight_Cluster_1_Services_Client.tar
     tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar
     cd FusionInsight_Cluster_1_Services_ClientConfig
     scp Flink/config/* root@*客户端节点IP地址*:/opt/client/conf
     主要配置文件说明如**表10-5**所示。

     表 10-5 配置文件

文件名称	作用
core-site.xml	配置Flink详细参数。
hdfs-site.xml	配置HDFS详细参数。
yarn-site.xml	配置Yarn详细参数。
flink-conf.yaml	Flink客户端配置文件。

  c. 检查客户端节点网络连接。
     在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

10.3.2 导入并配置 Flink 样例工程

操作场景

Flink针对多个场景提供样例工程，包含Java样例工程和Scala样例工程等，帮助客户快速学习Flink工程。

针对Java和Scala不同语言的工程，其导入方式相同。

以下操作步骤以导入Java样例代码为例。操作流程如图10-4所示。

图 10-4 导入样例工程流程

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\flink-examples”目录下的样例工程文件夹“flink-examples-normal”。

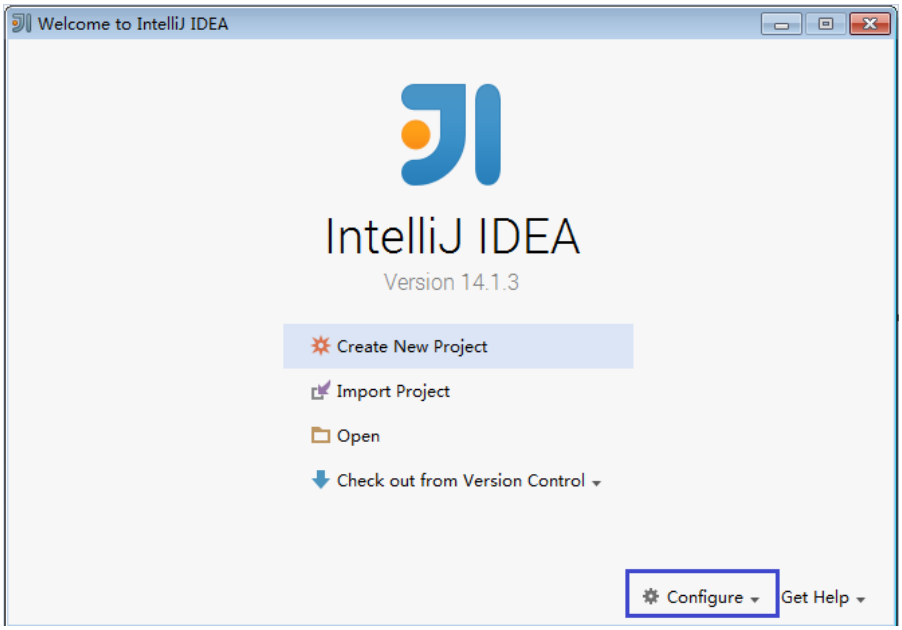
说明

- 在安全模式下，获取“src\flink-examples”下的样例工程flink-examples-security。
- 在普通模式下，获取“src\flink-examples”下的样例工程flink-examples-normal。

步骤2 在导入样例工程之前，IntelliJ IDEA需要进行配置JDK。

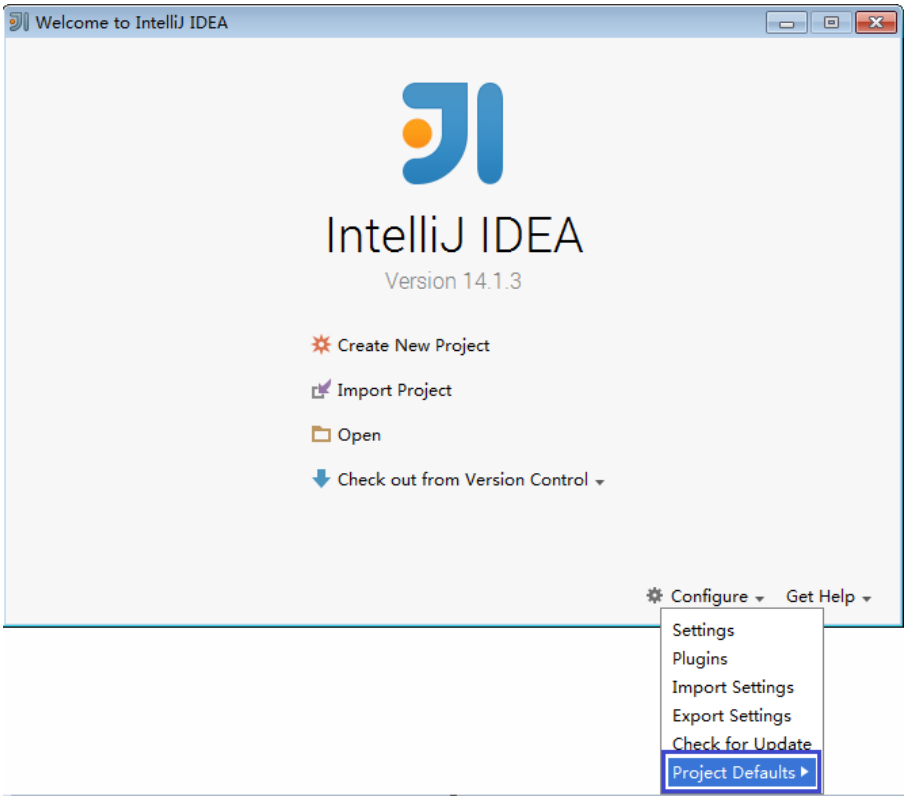
1. 打开IntelliJ IDEA，单击“Configure”下拉按钮。

图 10-5 Choosing Configure



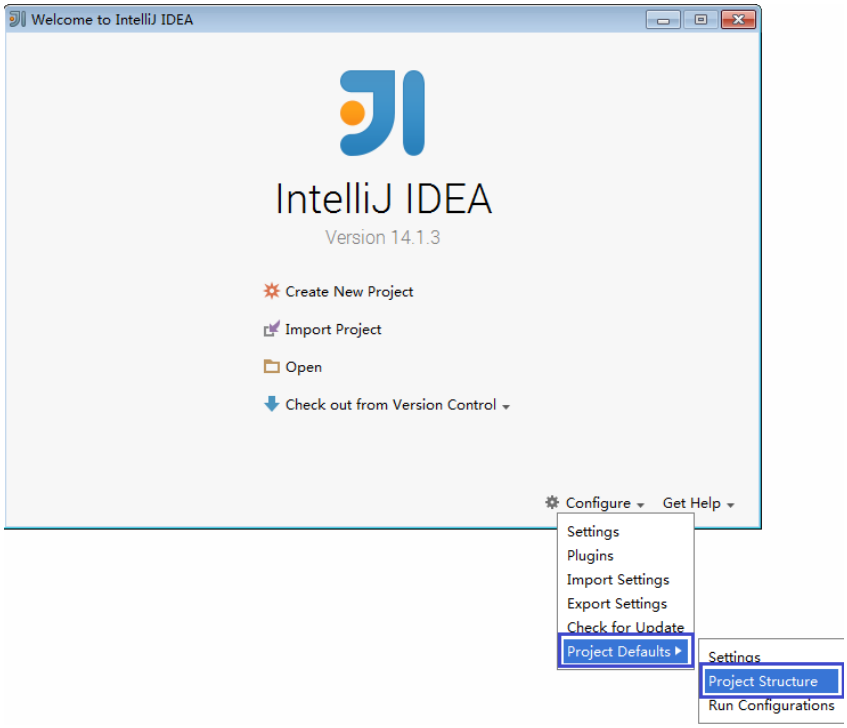
2. 在“Configure”下拉菜单中单击“Project Defaults”。

图 10-6 Choosing Project Defaults



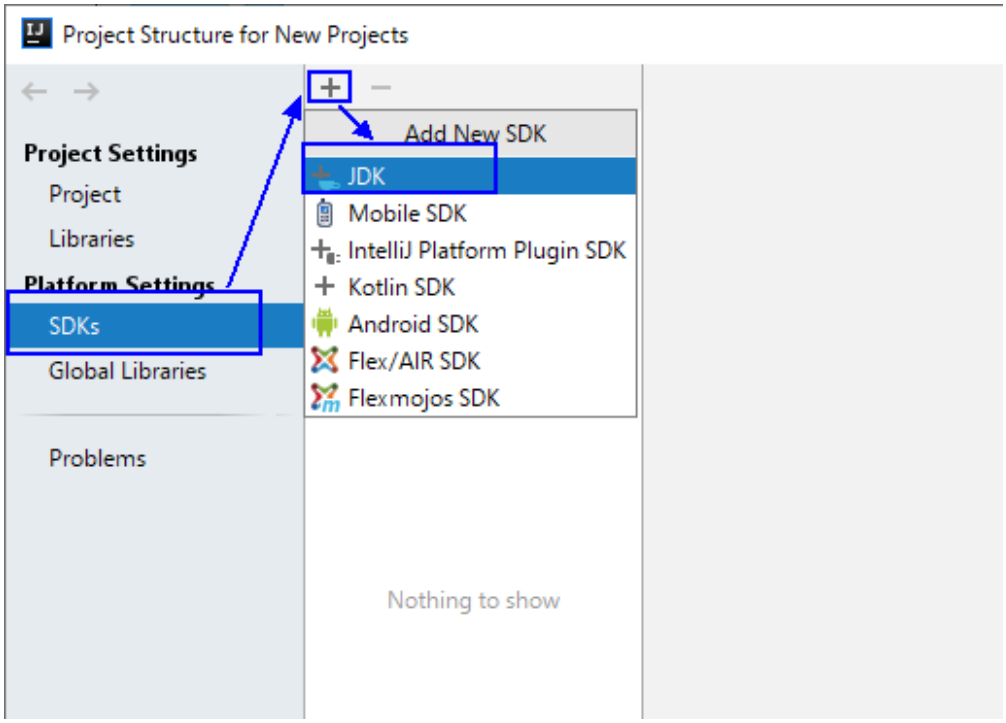
3. 在“Project Defaults”菜单中选择“Project Structure”。

图 10-7 Project Defaults



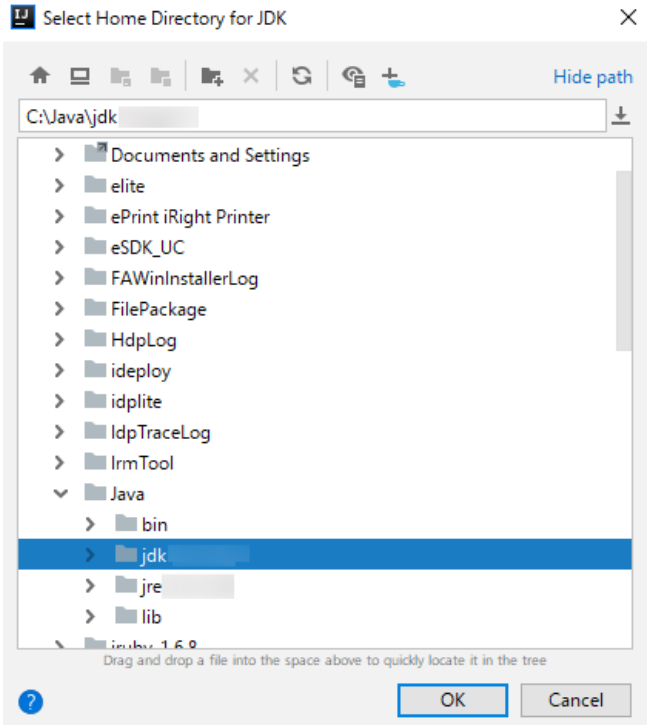
4. 在打开的“Project Structure”页面中，选择“SDKs”，单击绿色加号添加JDK。

图 10-8 添加 JDK



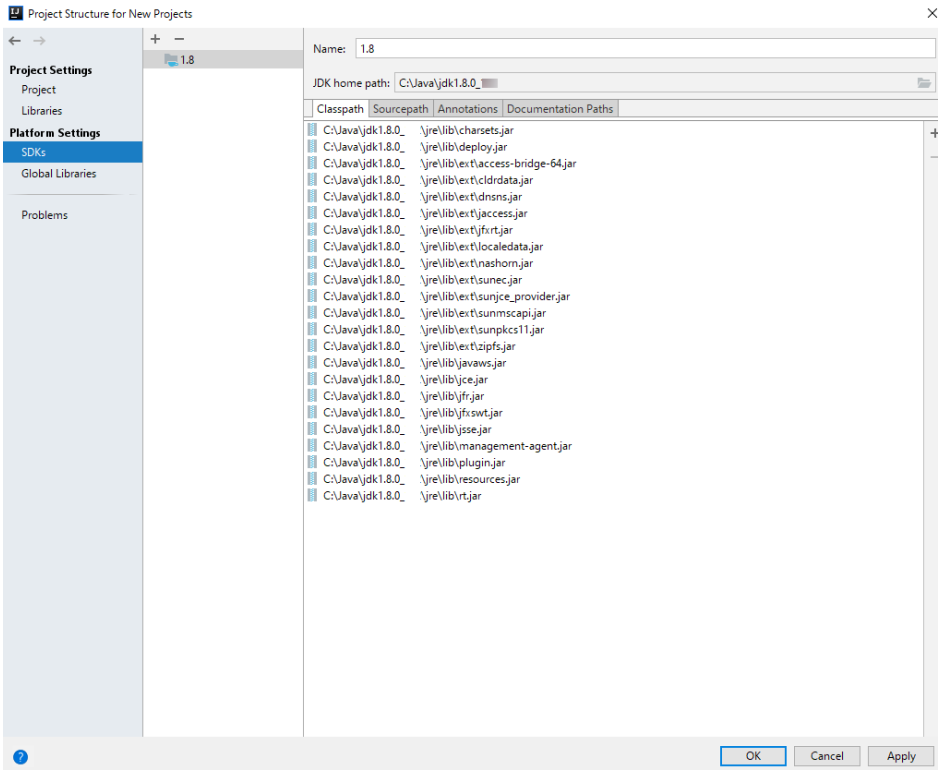
5. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 10-9 选择 JDK 目录



6. 完成JDK选择后，单击“OK”完成配置。

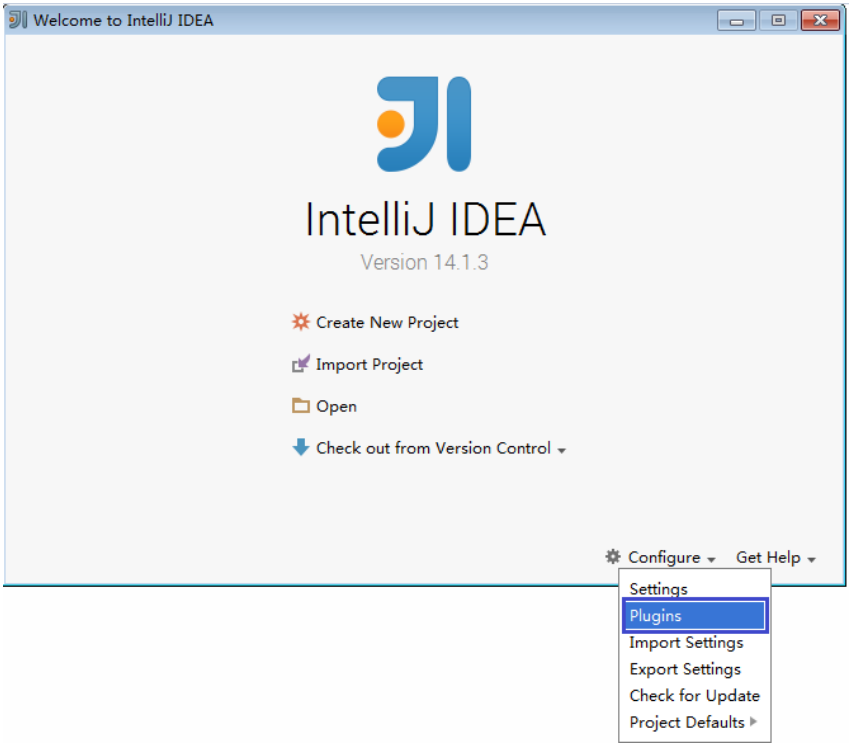
图 10-10 完成 JDK 配置



步骤3 （可选）如果导入Scala语言开发样例工程，还需要在IntelliJ IDEA中安装Scala插件。

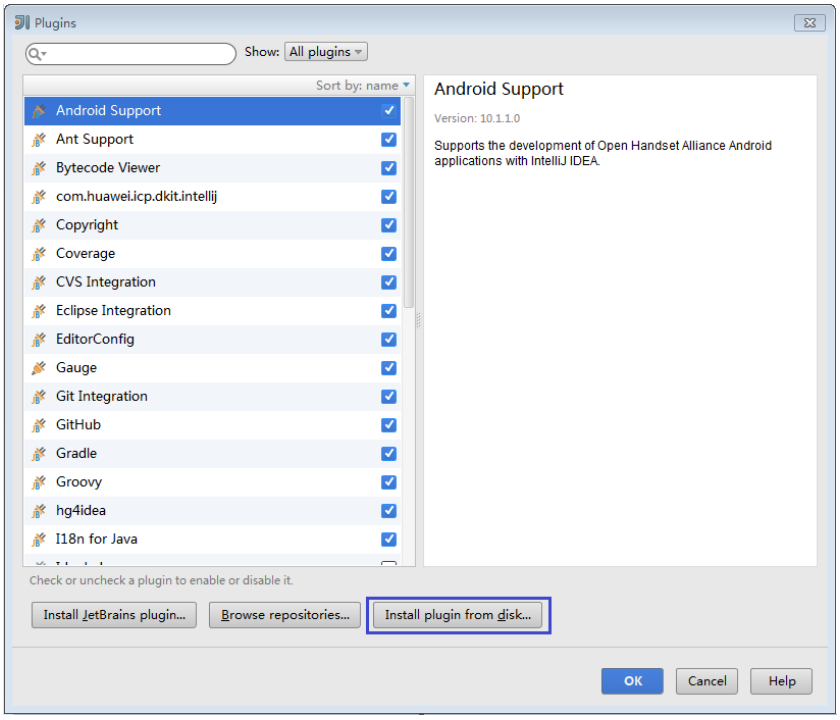
1. 在“Configure”下拉菜单中，单击“Plugins”。

图 10-11 Plugins



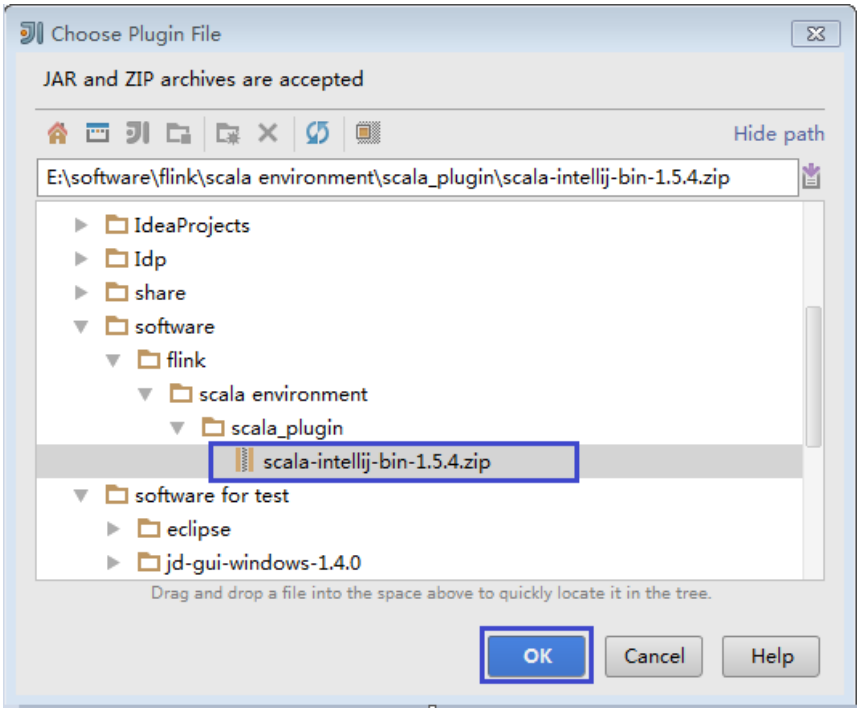
2. 在“Plugins”页面，选择“Install plugin from disk”。

图 10-12 Install plugin from disk



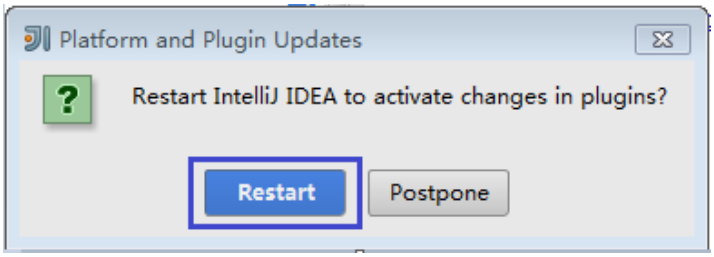
3. 在“Choose Plugin File”页面，选择对应版本的Scala插件包，单击“OK”。

图 10-13 choose plugin File



4. 在“Plugins”页面，单击“Apply”安装Scala插件。
5. 在弹出的“Platform and Plugin Updates”页面，单击“Restart”，使配置生效。

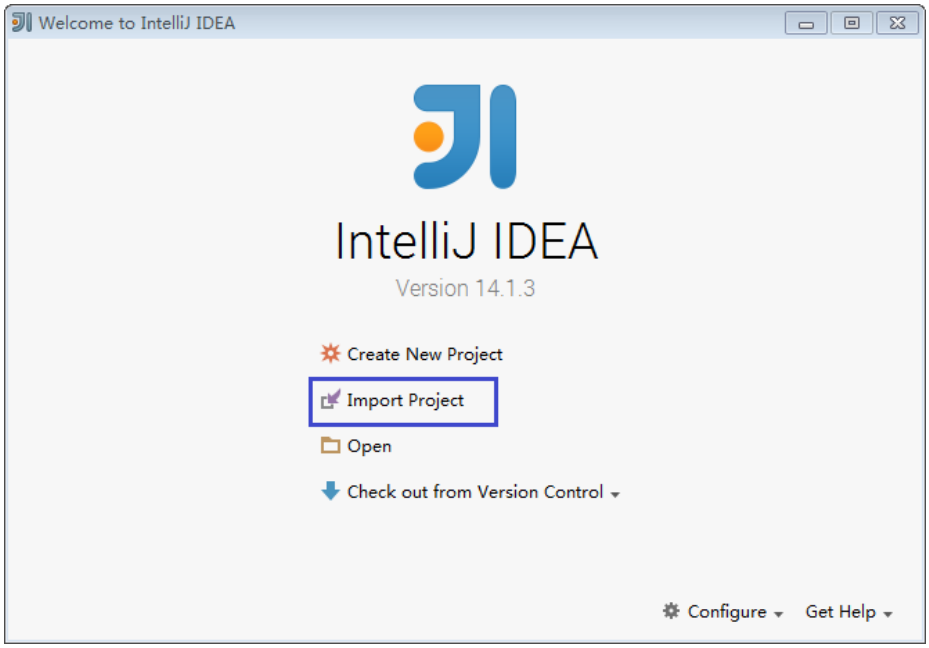
图 10-14 Platform and Plugin Updates



步骤4 将Java样例工程导入到IDEA中。

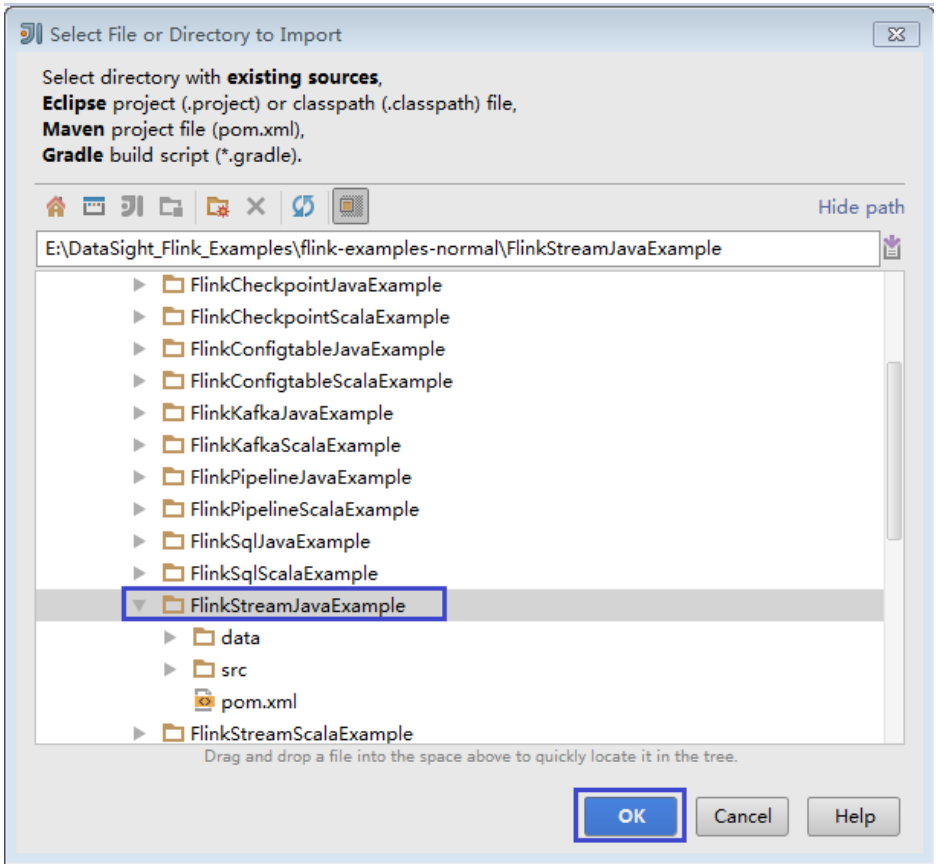
1. 打开IntelliJ IDEA。在“Quick Start”页面选择“Import Project”。
   或者，针对已使用过的IDEA工具，您可以从IDEA主界面直接添加。选择“File > Import project...”导入工程。

图 10-15 Import Project（Quick Start 页面）



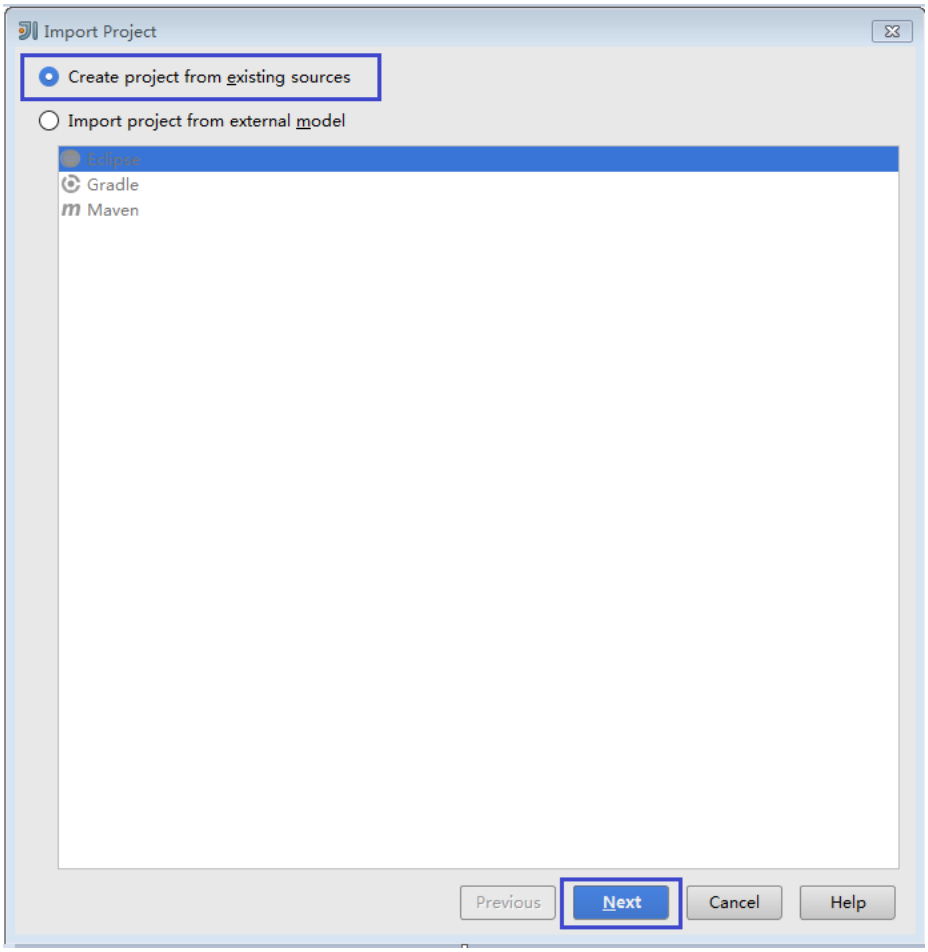
2. 选择需导入的样例工程路径，然后单击“OK”。

图 10-16 Select File or Directory to Import



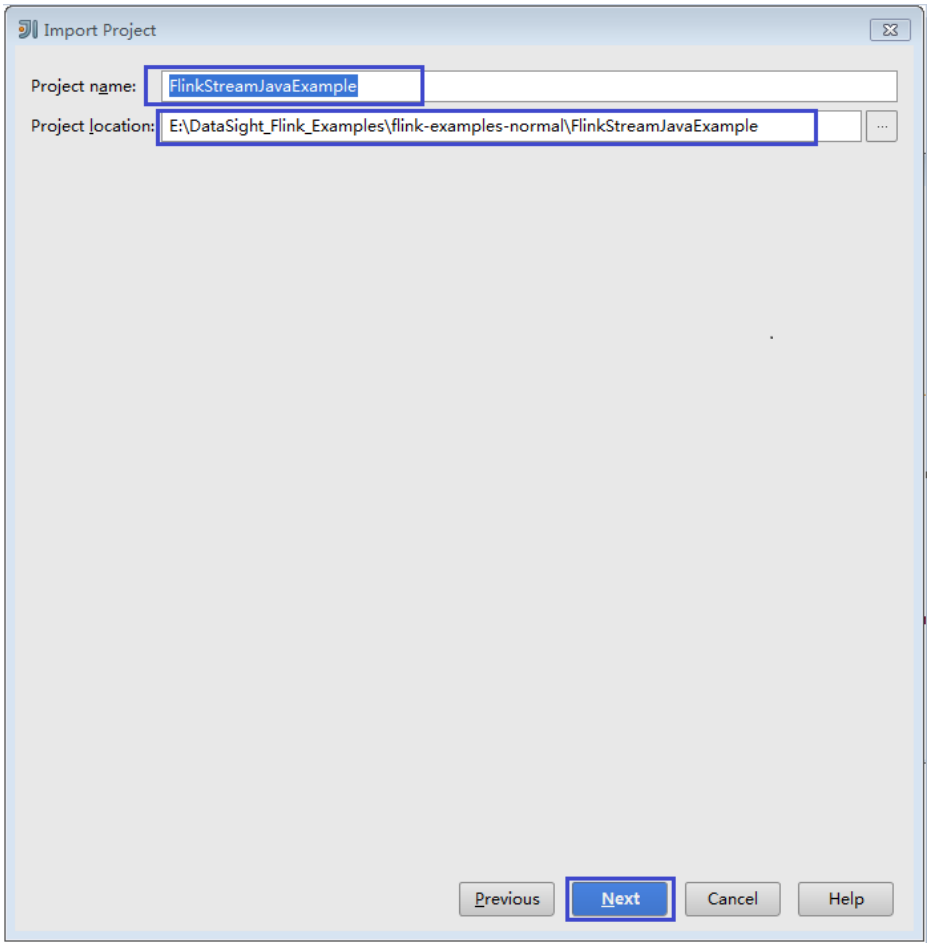
3. 选择从已存在的源码创建工程，然后单击“Next”。

图 10-17 Create project from existing sources



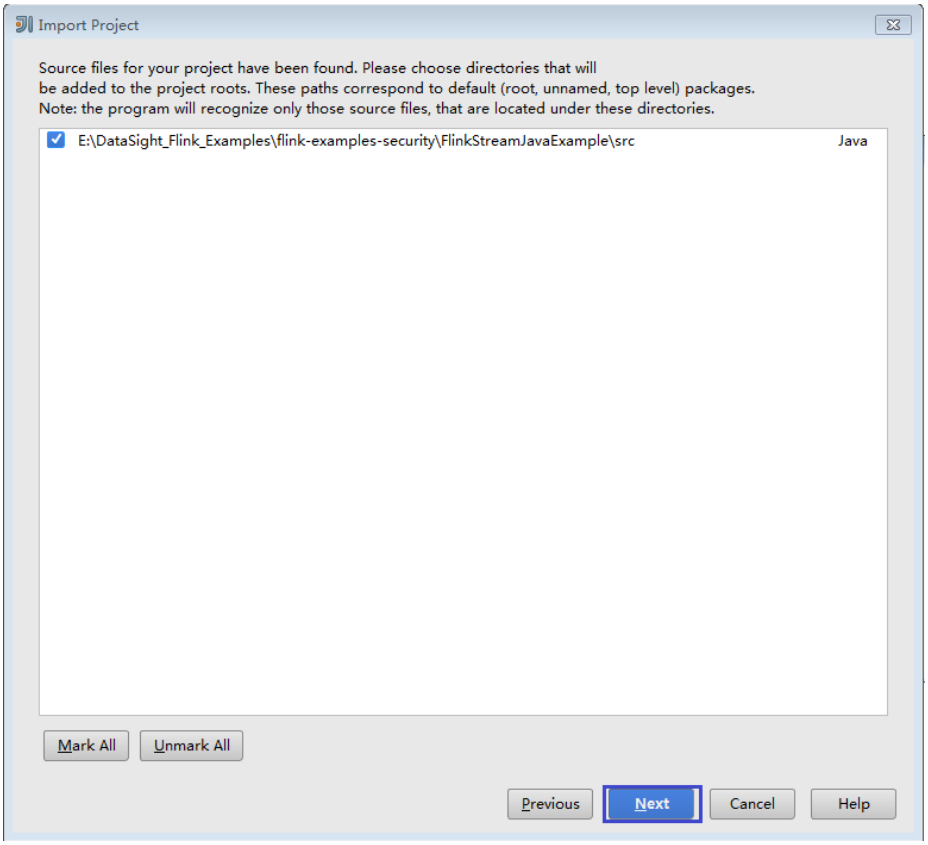
4. 确认导入路径和工程名称，单击“Next”。

图 10-18 Import Project



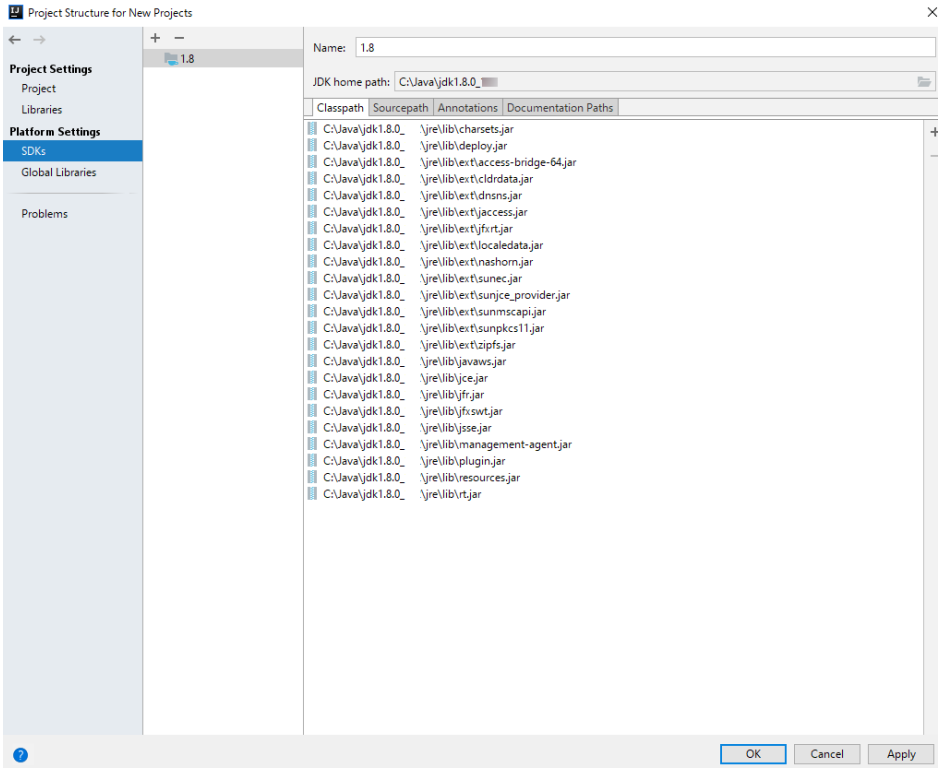
5. 确认导入工程的root目录，默认即可，单击“Next”。

图 10-19 Import Project



6. 确认IDEA自动识别的依赖库以及建议的模块结构，默认即可，单击“Next”。
7. 确认工程所用JDK，然后单击“Next”。

图 10-20 Select project SDK



8. 导入结束，单击“Finish”，IDEA主页显示导入的样例工程。

图 10-21 导入结束

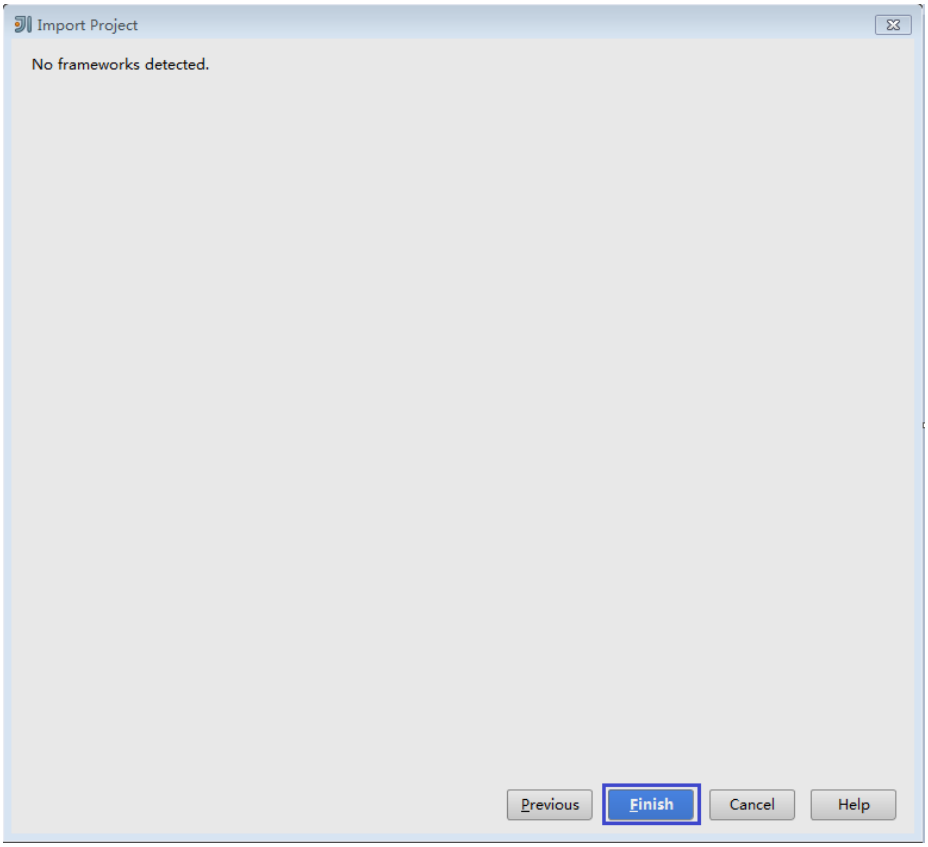
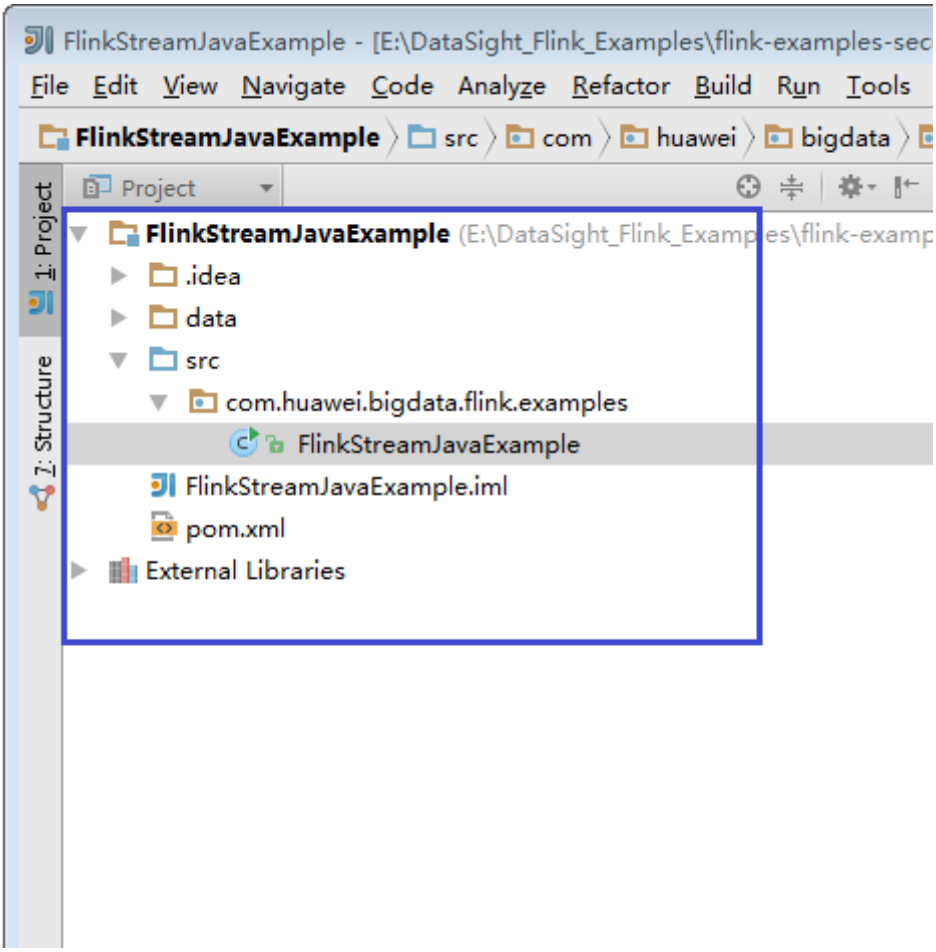


图 10-22 已导入工程



步骤5 导入样例工程依赖的Jar包。

如果通过开源镜像站方式获取的样例工程代码，在配置好Maven后， 相关依赖jar包将自动下载，不需手动添加。

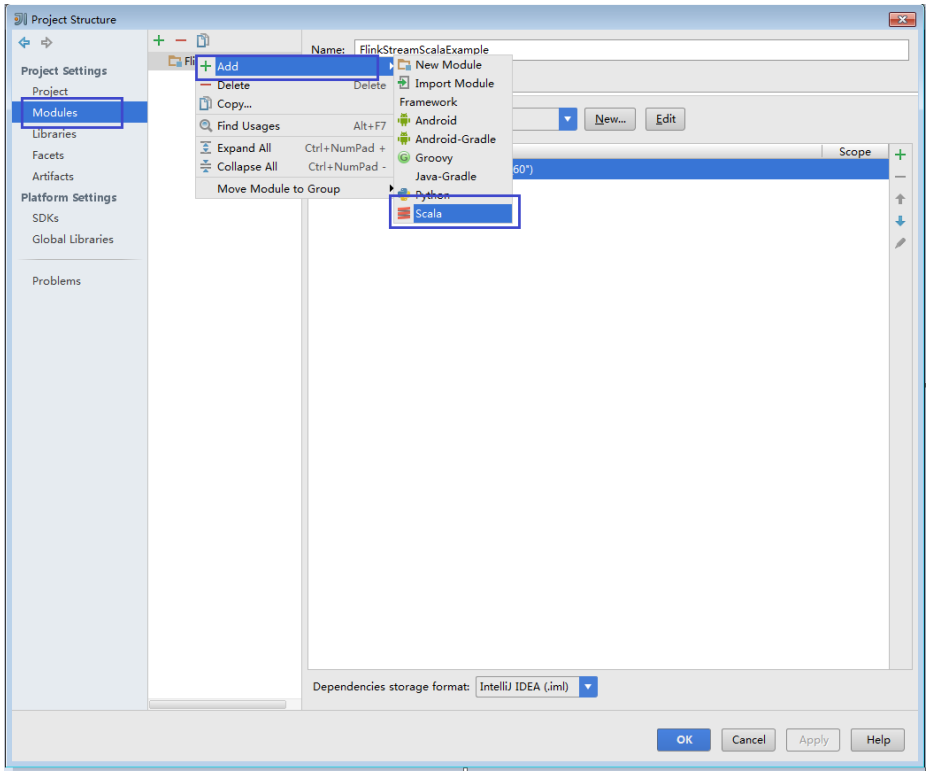
说明

当样例代码使用其他FusionInsight组件时，例如Kafka等，请去对应FusionInsight组件的服务端安装目录查找并添加依赖包。样例工程对应的依赖包详情，请参见**参考信息**。

步骤6 （可选）如果导入Scala语言开发的样例程序，还需要为工程设置语言。

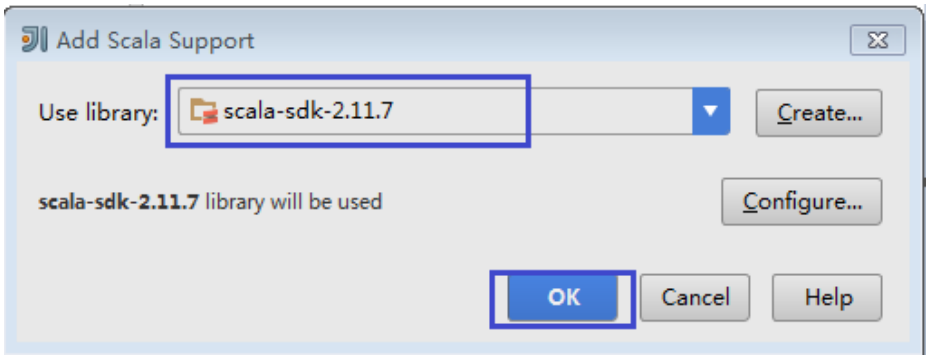
1. 在IDEA主页，选择“File>Project Structures...”进入“Project Structure”页面。
2. 选择“Modules”，选中工程名称，然后右键选择“Add > Scala”。

图 10-23 选择 Scala 语言



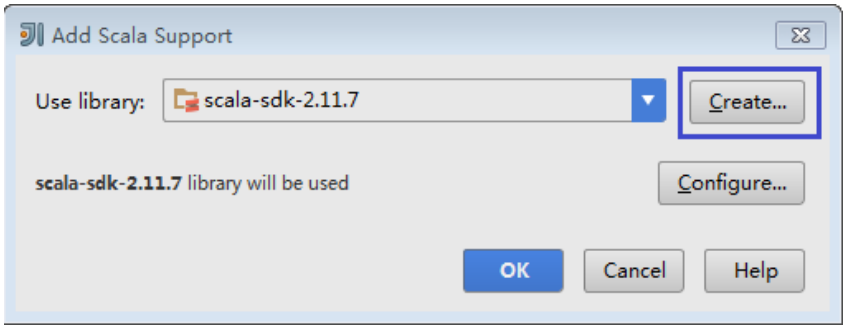
3. 当IDEA可以识别出Scala SDK时，在设置界面，选择编译的依赖jar包，然后单击“OK”应用设置

图 10-24 Add Scala Support



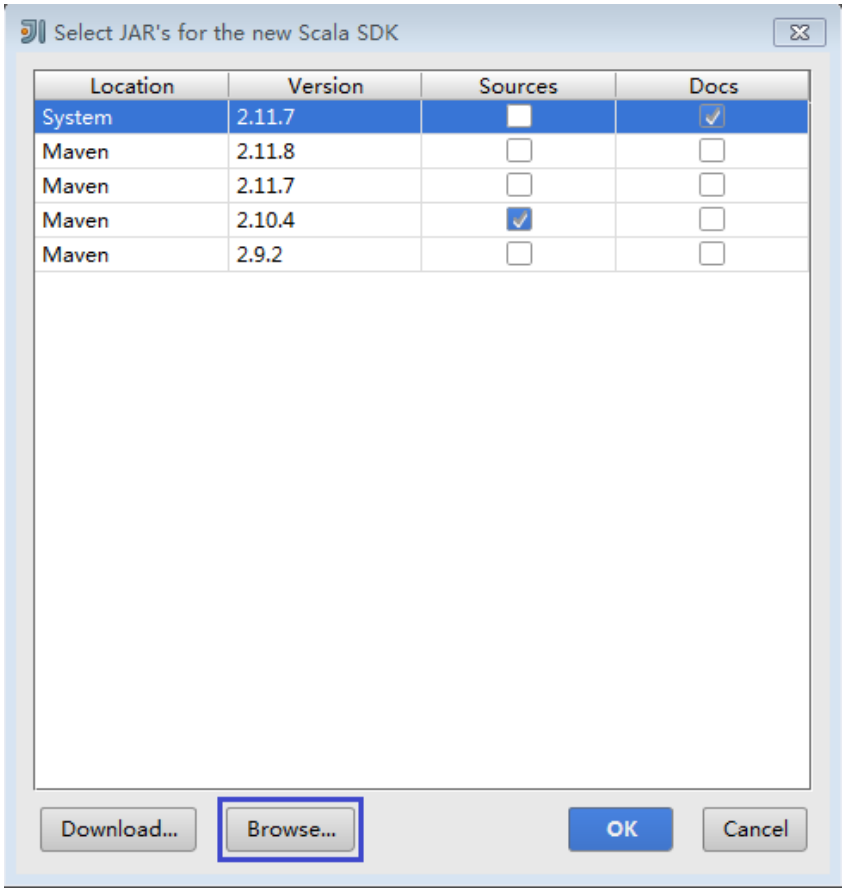
4. 当系统无法识别出Scala SDK时，需要自行创建。
   a. 单击“Create...”。

图 10-25 Create...



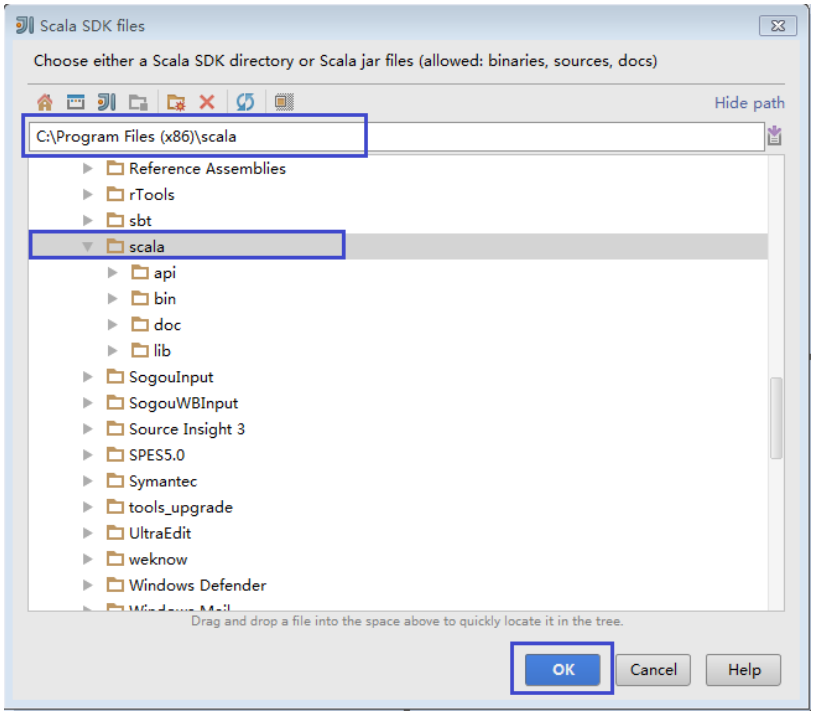
b. 在“Select JAR's for the new Scala SDK”页面单击“Browse...”。

图 10-26 Select JAR's for the new Scala SDK



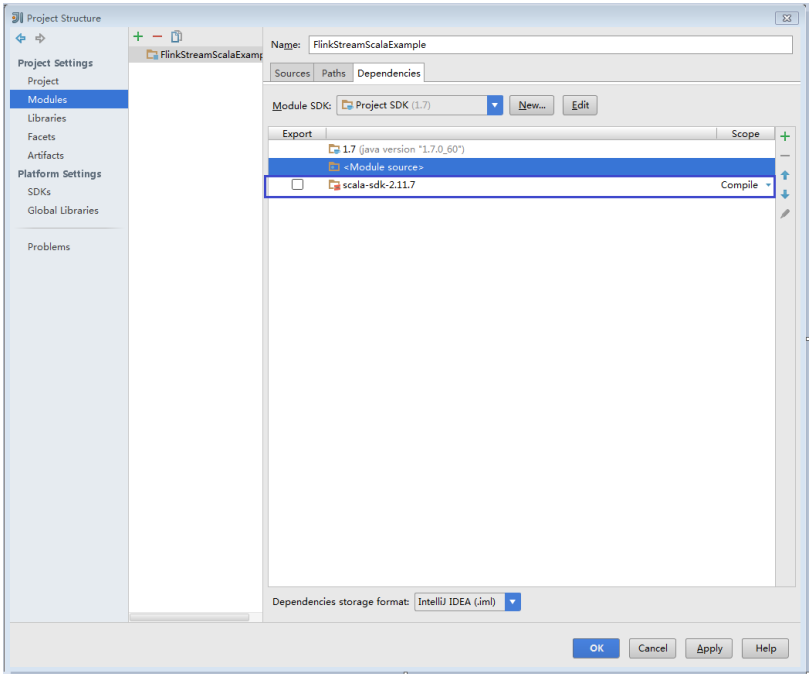
c. 在“Scala SDK files”页面选择scala sdk目录，单击“OK”。

图 10-27 Scala SDK files



5. 设置成功，单击“OK”保存设置。

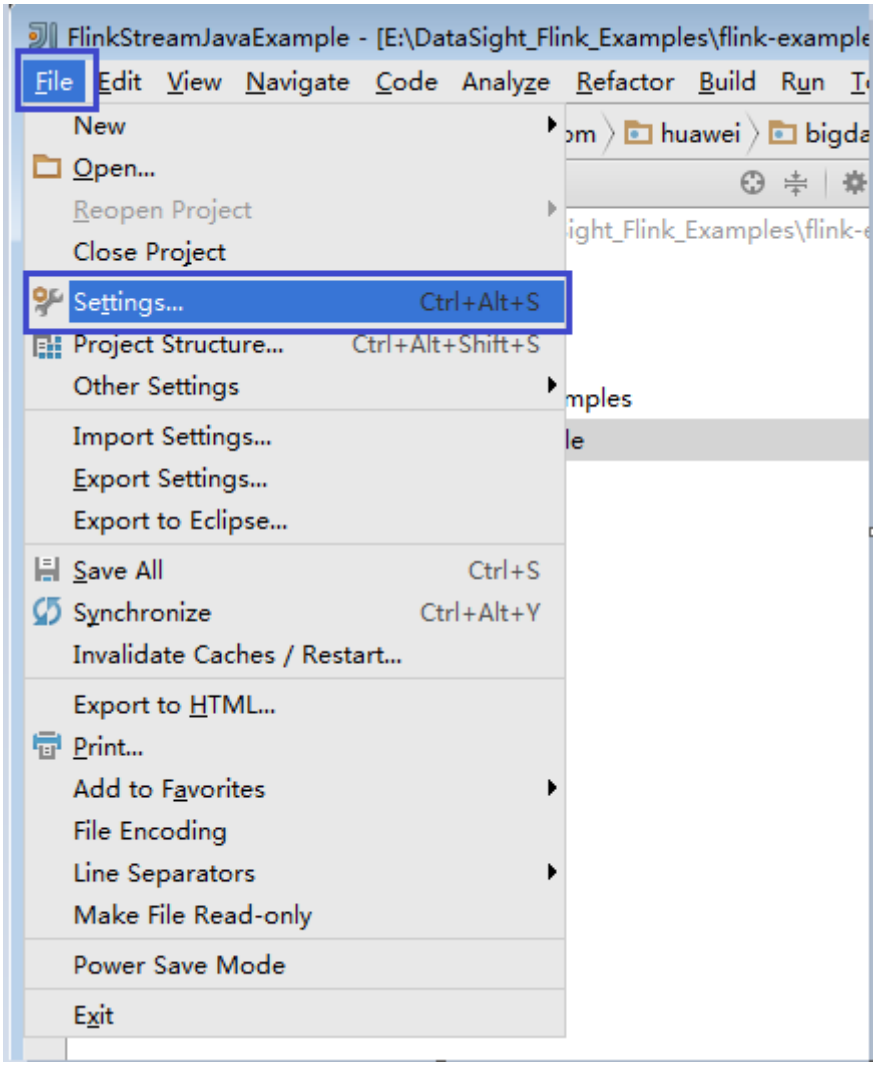
图 10-28 设置成功



步骤7 设置IDEA的文本文件编码格式，解决乱码显示问题。

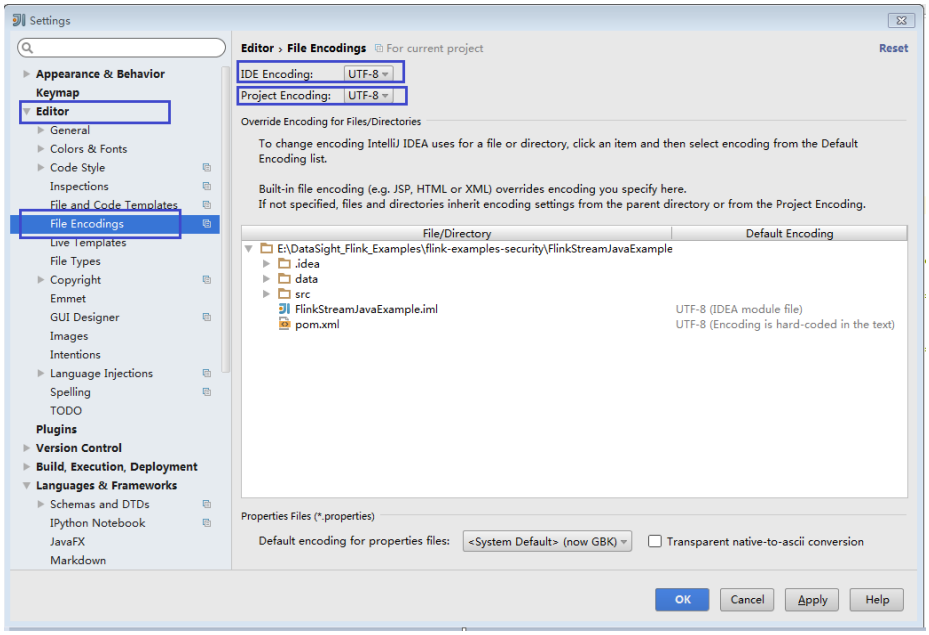
1. 在IDEA首页，选择“File > Settings...”。

图 10-29 选择 Settings



2. 编码配置。
   a. 在“Settings”页面，展开“Editor”，选择“File Encodings”。
   b. 分别在右侧的“IDE Encoding”和“Project Encoding”的下拉框中，选择“UTF-8”。
   c. 单击“Apply”应用配置。
   d. 单击“OK”完成编码配置。

图 10-30 Settings



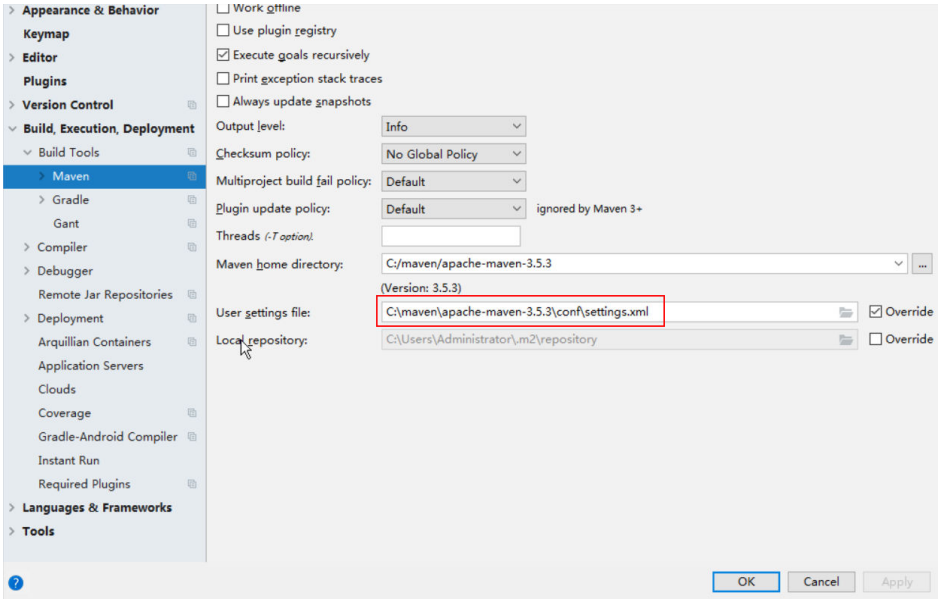
说明

- 请从Flink服务端安装目录获取相关的依赖包。
- 请从Kafka环境中获取Kafka依赖包。
- 具体依赖包请查看**参考信息**。

步骤8 配置Maven。

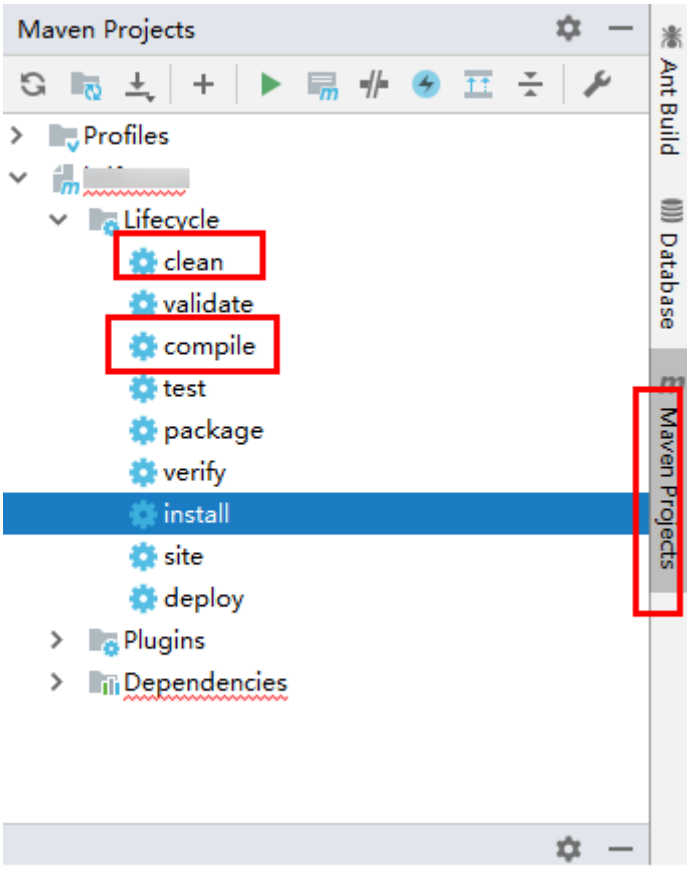
1. 参考**配置华为开源镜像仓**章节描述，增加开源镜像仓地址等配置信息到本地Maven的“setting.xml”配置文件。
2. 修改完成后，在IntelliJ IDEA选择“File > Settings > Build, Execution, Deployment > Build Tools > Maven”，勾选“User settings file”右侧的“Override”，并修改“User settings file”的值为当前“settings.xml”文件放置目录，确保该目录为“*<本地Maven安装目录>*\conf\settings.xml”。

图 10-31 “settings.xml”文件放置目录



3. 单击“Maven home directory”右侧的下拉菜单，选择Maven的安装路径。
4. 单击“Apply”并单击“OK”。
5. 在IntelliJ IDEA主界面右侧，单击“Maven Projects”，在“Maven Projects”界面执行“*项目名称* > Lifecycle”目录下的“clean”和“compile”脚本。

图 10-32 Maven Projects 界面



----结束

参考信息

Flink客户端lib目录、opt目录中都有flink jar包，其中lib目录中默认是flink核心jar包，opt目录中是对接外部组件的jar包（例如flink-connector-kafka*.jar）,若应用开发中需要请手动复制相关jar包到lib目录中。

针对Flink提供的几个样例工程，其对应的运行依赖包如下：

- DataStream程序样例工程（Java/Scala）
  - flink-dist_*.jar

    说明

    flink-dist_*.jar可在Flink的客户端或者服务端安装路径的lib目录下获取。
- 向Kafka生产并消费数据程序样例工程（Java/Scala）
  - kafka-clients-*.jar
  - flink-connector-kafka_*.jar

    说明

    - flink-connector-kafka_*.jar可在Flink客户端或者服务端安装路径的opt目录下获取。
    - kafka-clients-*.jar由Kafka组件发布提供，可在Kafka组件客户端或者服务端安装路径下的lib目录下获取。

- 异步Checkpoint机制程序样例工程（Java/Scala）
  - flink-dist_*.jar
- pipeline程序样例工程（Java/Scala）
  - flink-connector-netty_*.jar
  - flink-dist_*.jar
  - flink-shaded-curator-*.jar
  - curator-client-2.12.0.jar
  - curator-framework-2.12.0.jar

  说明

  - flink-shaded-curator-*.jar可在Flink客户端或者服务端安装路径的opt目录下获取。
  - flink-connector-netty_*.jar、curator-client-2.12.0.jar、curator-framework-2.12.0.jar可在二次开发样例代码编译后产生的lib文件夹下获取。
  - flink-shaded-curator-*.jar仅适用于MRS 3.0.X集群。
  - curator-client-2.12.0.jar、curator-framework-2.12.0.jar仅适用于MRS 3.1.X集群。
- Stream SQL Join样例工程（Java）
  - kafka-clients-*.jar
  - flink-connector-kafka_2.11*.jar
  - flink-connector-kafka-base_*.jar
  - flink-connector-kafka_*.jar
  - flink-dist_2.11*.jar
  - flink-table_2.11*.jar

  说明

  - flink-connector-kafka-base_*.jar、flink-connector-kafka_*.jar可在Flink客户端或者服务端安装路径的“opt”目录下获取。
  - flink-connector-kafka-base_*.jar、flink-connector-kafka_*.jar仅适用于MRS 3.0.X集群。
  - flink-connector-kafka_2.11*.jar、flink-dist_2.11*.jar和flink-table_2.11*.jar仅适用于MRS 3.1.X集群。

10.4 开发 Flink 应用

10.4.1 Flink DataStream 样例程序

10.4.1.1 Flink DataStream 样例程序开发思路

场景说明

假定用户有某个网站周末网民网购停留时间的日志文本，基于某些业务要求，要求开发Flink的DataStream应用程序实现如下功能：

说明

DataStream应用程序可以在Windows环境和Linux环境中运行。

- 实时统计总计网购时间超过2个小时的女性网民信息。
- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

  log1.txt：周六网民停留日志。该日志文件在该样例程序中的data目录下获取。

  ```
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  FangBo,female,50
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,50
  CaiXuyu,female,50
  FangBo,female,60
  ```

  log2.txt：周日网民停留日志。该日志文件在该样例程序中的data目录下获取。

  ```
  LiuYang,female,20
  YuanJing,male,10
  CaiXuyu,female,50
  FangBo,female,50
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  CaiXuyu,female,50
  FangBo,female,50
  LiuYang,female,20
  YuanJing,male,10
  FangBo,female,50
  GuoYijun,male,50
  CaiXuyu,female,50
  FangBo,female,60
  ```

数据规划

DataStream样例工程的数据存储在文本中。

将log1.txt和log2.txt放置在指定路径下，例如"/opt/log1.txt"和"/opt/log2.txt"。

说明

- 数据文件若存放在本地文件系统，需在所有部署Yarn NodeManager的节点指定目录放置，并设置运行用户访问权限。
- 或将数据文件放置于HDFS，并指定程序中读取文件路径HDFS路径，例如"hdfs://hacluster/path/to/file"。

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

1. 读取文本数据，生成相应DataStream，解析数据生成UserRecord信息。
2. 筛选女性网民上网时间数据信息。
3. 按照姓名、性别进行keyby操作，并汇总在一个时间窗口内每个女性上网时间。
4. 筛选连续上网时间超过阈值的用户，并获取结果。

10.4.1.2 Flink DataStream 样例程序（Java）

功能介绍

统计连续网购时间超过2个小时的女性网民信息，将统计结果直接打印。

DataStream FlinkStreamJavaExample 代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.flink.examples.FlinkStreamJavaExample：

```
    // 参数解析:
    // <filePath>为文本读取路径，用逗号分隔。
    // <windowTime>为统计数据的窗口跨度,时间单位都是分。
public class FlinkStreamJavaExample {
    public static void main(String[] args) throws Exception {
        // 打印出执行flink run的参考命令
        System.out.println("use command as: ");
        System.out.println("./bin/flink run --class
com.huawei.bigdata.flink.examples.FlinkStreamJavaExample /opt/test.jar --filePath /opt/log1.txt,/opt/
log2.txt --windowTime 2");
        System.out.println("*******************************************************************************************");
        System.out.println("<filePath> is for text file to read data, use comma to separate");
        System.out.println("<windowTime> is the width of the window, time as minutes");
        System.out.println("*******************************************************************************************");

        // 读取文本路径信息，并使用逗号分隔
        final String[] filePaths = ParameterTool.fromArgs(args).get("filePath", "/opt/log1.txt,/opt/
log2.txt").split(",");
        assert filePaths.length > 0;

        // windowTime设置窗口时间大小，默认2分钟一个窗口足够读取文本内的所有数据了
        final int windowTime = ParameterTool.fromArgs(args).getInt("windowTime", 2);

        // 构造执行环境，使用eventTime处理窗口数据
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        env.setParallelism(1);

        // 读取文本数据流
        DataStream<String> unionStream = env.readTextFile(filePaths[0]);
        if (filePaths.length > 1) {
            for (int i = 1; i < filePaths.length; i++) {
                unionStream = unionStream.union(env.readTextFile(filePaths[i]));
            }
        }

        // 数据转换，构造整个数据处理的逻辑，计算并得出结果打印出来
        unionStream.map(new MapFunction<String, UserRecord>() {
            @Override
            public UserRecord map(String value) throws Exception {
                return getRecord(value);
            }
        }).assignTimestampsAndWatermarks(
                new Record2TimestampExtractor()
        ).filter(new FilterFunction<UserRecord>() {
            @Override
            public boolean filter(UserRecord value) throws Exception {
                return value.sexy.equals("female");
            }
        }).keyBy(
            new UserRecordSelector()
        ).window(
            TumblingEventTimeWindows.of(Time.minutes(windowTime))
        ).reduce(new ReduceFunction<UserRecord>() {
            @Override
            public UserRecord reduce(UserRecord value1, UserRecord value2)
```

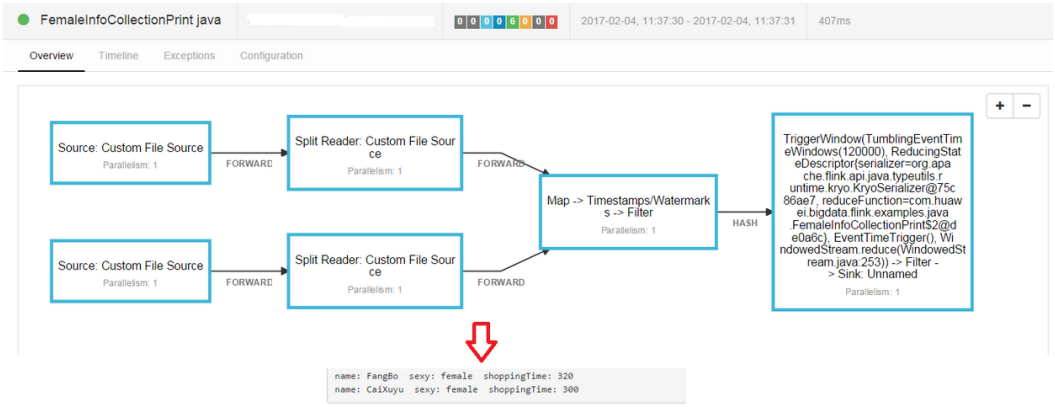
```
                throws Exception {
            value1.shoppingTime += value2.shoppingTime;
            return value1;
        }
    }).filter(new FilterFunction<UserRecord>() {
        @Override
        public boolean filter(UserRecord value) throws Exception {
            return value.shoppingTime > 120;
        }
    }).print();

    // 调用execute触发执行
    env.execute("FemaleInfoCollectionPrint java");
}

// 构造keyBy的关键字作为分组依据
private static class UserRecordSelector implements KeySelector<UserRecord, Tuple2<String, String>> {
    @Override
    public Tuple2<String, String> getKey(UserRecord value) throws Exception {
        return Tuple2.of(value.name, value.sexy);
    }
}

// 解析文本行数据，构造UserRecord数据结构
private static UserRecord getRecord(String line) {
    String[] elems = line.split(",");
    assert elems.length == 3;
    return new UserRecord(elems[0], elems[1], Integer.parseInt(elems[2]));
}

// UserRecord数据结构的定义，并重写了toString打印方法
public static class UserRecord {
    private String name;
    private String sexy;
    private int shoppingTime;

    public UserRecord(String n, String s, int t) {
        name = n;
        sexy = s;
        shoppingTime = t;
    }

    public String toString() {
        return "name: " + name + "  sexy: " + sexy + "  shoppingTime: " + shoppingTime;
    }
}

// 构造继承AssignerWithPunctuatedWatermarks的类，用于设置eventTime以及waterMark
private static class Record2TimestampExtractor implements
AssignerWithPunctuatedWatermarks<UserRecord> {

    // add tag in the data of datastream elements
    @Override
    public long extractTimestamp(UserRecord element, long previousTimestamp) {
        return System.currentTimeMillis();
    }

    // give the watermark to trigger the window to execute, and use the value to check if the window
elements is ready
    @Override
    public Watermark checkAndGetNextWatermark(UserRecord element, long extractedTimestamp) {
        return new Watermark(extractedTimestamp - 1);
    }
}
}
```

执行之后打印结果如下所示：

```
name: FangBo  sexy: female  shoppingTime: 320
name: CaiXuyu  sexy: female  shoppingTime: 300
```

执行如图10-33所示。

图 10-33 显示图



10.4.1.3 Flink DataStream 样例程序（Scala）

功能介绍

实时统计连续网购时间超过2个小时的女性网民信息，将统计结果直接打印出来。

DataStream FlinkStreamScalaExample 代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.flink.examples.FlinkStreamScalaExample：

```
  // 参数解析:
  // filePath为文本读取路径，用逗号分隔。
  // windowTime;为统计数据的窗口跨度,时间单位都是分。
  object FlinkStreamScalaExample {
 def main(args: Array[String]) {
  // 打印出执行flink run的参考命令
  System.out.println("use command as: ")
  System.out.println("./bin/flink run --class
com.huawei.bigdata.flink.examples.FlinkStreamScalaExample /opt/test.jar --filePath /opt/log1.txt,/opt/
log2.txt --windowTime 2")
  System.out.println("*******************************************************************************************")
  System.out.println("<filePath> is for text file to read data, use comma to separate")
  System.out.println("<windowTime> is the width of the window, time as minutes")
  System.out.println("*******************************************************************************************")

  // 读取文本路径信息，并使用逗号分隔
  val filePaths = ParameterTool.fromArgs(args).get("filePath",
    "/opt/log1.txt,/opt/log2.txt").split(",").map(_.trim)
  assert(filePaths.length > 0)

  // windowTime设置窗口时间大小，默认2分钟一个窗口足够读取文本内的所有数据了
  val windowTime = ParameterTool.fromArgs(args).getInt("windowTime", 2)

  // 构造执行环境，使用eventTime处理窗口数据
  val env = StreamExecutionEnvironment.getExecutionEnvironment
  env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
  env.setParallelism(1)

  // 读取文本数据流
  val unionStream = if (filePaths.length > 1) {
    val firstStream = env.readTextFile(filePaths.apply(0))
    firstStream.union(filePaths.drop(1).map(it => env.readTextFile(it)): _*)
  } else {
```

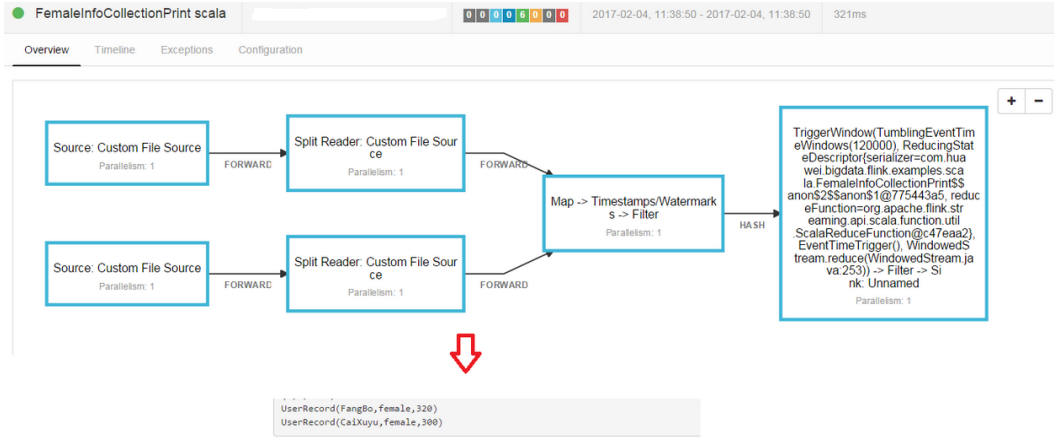
```
    env.readTextFile(filePaths.apply(0))
  }

  // 数据转换，构造整个数据处理的逻辑，计算并得出结果打印出来
  unionStream.map(getRecord(_))
    .assignTimestampsAndWatermarks(new Record2TimestampExtractor)
    .filter(_.sexy == "female")
    .keyBy("name", "sexy")
    .window(TumblingEventTimeWindows.of(Time.minutes(windowTime)))
    .reduce((e1, e2) => UserRecord(e1.name, e1.sexy, e1.shoppingTime + e2.shoppingTime))
    .filter(_.shoppingTime > 120).print()

  // 调用execute触发执行
  env.execute("FemaleInfoCollectionPrint scala")
}

// 解析文本行数据，构造UserRecord数据结构
def getRecord(line: String): UserRecord = {
  val elems = line.split(",")
  assert(elems.length == 3)
  val name = elems(0)
  val sexy = elems(1)
  val time = elems(2).toInt
  UserRecord(name, sexy, time)
}

// UserRecord数据结构的定义
case class UserRecord(name: String, sexy: String, shoppingTime: Int)


// 构造继承AssignerWithPunctuatedWatermarks的类，用于设置eventTime以及waterMark
private class Record2TimestampExtractor extends AssignerWithPunctuatedWatermarks[UserRecord] {

  // add tag in the data of datastream elements
  override def extractTimestamp(element: UserRecord, previousTimestamp: Long): Long = {
    System.currentTimeMillis()
  }

  // give the watermark to trigger the window to execute, and use the value to check if the window
elements is ready
  def checkAndGetNextWatermark(lastElement: UserRecord,
                               extractedTimestamp: Long): Watermark = {
    new Watermark(extractedTimestamp - 1)
  }
 }
}
```

执行之后打印结果如下所示：

```
UserRecord(FangBo,female,320)
UserRecord(CaiXuyu,female,300)
```

执行如**图10-34**所示。

图 10-34 显示图



10.4.2 Flink Kafka 样例程序

10.4.2.1 Flink Kafka 样例程序开发思路

场景说明

假定某个Flink业务每秒就会收到1个消息记录。

基于某些业务要求，开发的Flink应用程序实现功能：实时输出带有前缀的消息内容。

数据规划

Flink样例工程的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户），并从Kafka组件接收数据。

1. 确保集群安装完成，包括HDFS、Yarn、Flink和Kafka。
2. 创建Topic。

   创建topic的命令格式：

   bin/kafka-topics.sh --create --zookeeper {zkQuorum}/kafka --partitions {partitionNum} --replication-factor {replicationNum} --topic {Topic}

   表 10-6 参数说明

参数名	说明
{zkQuorum}	ZooKeeper集群信息，格式为IP:port。
{partitionNum}	topic的分区数。
{replicationNum}	topic中每个partition数据的副本数。
{Topic}	topic名称。

示例：在Kafka的客户端路径下执行命令，此处以ZooKeeper集群的IP:port是10.96.101.32:2181,10.96.101.251:2181,10.96.101.177:2181,10.91.8.160:2181，Topic名称为topic1的数据为例。

```
bin/kafka-topics.sh --create --zookeeper 10.96.101.32:2181,10.96.101.251:2181,10.96.101.177:2181,10.91.8.160:2181/kafka --partitions 5 --replication-factor 1 --topic topic1
```

开发思路

1. 启动Flink Kafka Producer应用向Kafka发送数据。
2. 启动Flink Kafka Consumer应用从Kafka接收数据，保证topic与producer一致。
3. 在数据内容中增加前缀并进行打印。

10.4.2.2 Flink Kafka 样例程序（Java）

功能介绍

在Flink应用中，调用flink-connector-kafka模块的接口，生产并消费数据。

代码样例

下面列出producer和consumer主要逻辑代码作为演示。

完整代码参见com.huawei.bigdata.flink.examples.WriteIntoKafka和com.huawei.bigdata.flink.examples.ReadFromKafka

```
//producer代码
public class WriteIntoKafka {
  public static void main(String[] args) throws Exception {
    // 打印出执行flink run的参考命令
    System.out.println("use command as: ");
    System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +
        " /opt/test.jar --topic topic-test --bootstrap.servers 10.91.8.218:9092");
    System.out.println("******************************************************************************************");
    System.out.println("<topic> is the kafka topic name");
    System.out.println("<bootstrap.servers> is the ip:port list of brokers");
    System.out.println("******************************************************************************************");

    // 构造执行环境
    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    // 设置并发度
    env.setParallelism(1);
    // 解析运行参数
    ParameterTool paraTool = ParameterTool.fromArgs(args);
    // 构造流图，将自定义Source生成的数据写入Kafka
    DataStream<String> messageStream = env.addSource(new SimpleStringGenerator());
    messageStream.addSink(new FlinkKafkaProducer<>(paraTool.get("topic"),
        new SimpleStringSchema(),
        paraTool.getProperties()));
    // 调用execute触发执行
    env.execute();
  }

  // 自定义Source，每隔1s持续产生消息
  public static class SimpleStringGenerator implements SourceFunction<String> {
    private static final long serialVersionUID = 2174904787118597072L;
    boolean running = true;
    long i = 0;

    @Override
    public void run(SourceContext<String> ctx) throws Exception {
      while (running) {
```

```
      ctx.collect("element-" + (i++));
      Thread.sleep(1000);
    }
  }

  @Override
  public void cancel() {
    running = false;
  }
 }
}

//consumer代码
public class ReadFromKafka {
 public static void main(String[] args) throws Exception {
  // 打印出执行flink run的参考命令
  System.out.println("use command as: ");
  System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka" +
      " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092");
  System.out.println("******************************************************************************************");
  System.out.println("<topic> is the kafka topic name");
  System.out.println("<bootstrap.servers> is the ip:port list of brokers");
  System.out.println("******************************************************************************************");

  // 构造执行环境
  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
  // 设置并发度
  env.setParallelism(1);
  // 解析运行参数
  ParameterTool paraTool = ParameterTool.fromArgs(args);
  // 构造流图，从Kafka读取数据并换行打印
  DataStream<String> messageStream = env.addSource(new FlinkKafkaConsumer<>(paraTool.get("topic"),
      new SimpleStringSchema(),
      paraTool.getProperties()));
  messageStream.rebalance().map(new MapFunction<String, String>() {
    @Override
    public String map(String s) throws Exception {
      return "Flink says " + s + System.getProperty("line.separator");
    }
  }).print();
  // 调用execute触发执行
  env.execute();
 }
}
```

10.4.2.3 Flink Kafka 样例程序（Scala）

功能介绍

在Flink应用中，调用flink-connector-kafka模块的接口，生产并消费数据。

代码样例

下面列出producer和consumer主要逻辑代码作为演示。

完整代码参见com.huawei.bigdata.flink.examples.WriteIntoKafka和com.huawei.bigdata.flink.examples.ReadFromKafka

```
//producer代码
object WriteIntoKafka {
 def main(args: Array[String]) {
  // 打印出执行flink run的参考命令
  System.out.println("use command as: ")
  System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +
    " /opt/test.jar --topic topic-test --bootstrap.servers 10.91.8.218:9092")
  System.out.println("******************************************************************************************")
  System.out.println("<topic> is the kafka topic name")
```

```
    System.out.println("<bootstrap.servers> is the ip:port list of brokers")
    System.out.println("************************************************************************************************")

    // 构造执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 设置并发度
    env.setParallelism(1)
    // 解析运行参数
    val paraTool = ParameterTool.fromArgs(args)
    // 构造流图，将自定义Source生成的数据写入Kafka
    val messageStream: DataStream[String] = env.addSource(new SimpleStringGenerator)
    messageStream.addSink(new FlinkKafkaProducer(
      paraTool.get("topic"), new SimpleStringSchema, paraTool.getProperties))
    // 调用execute触发执行
    env.execute
  }
}

// 自定义Source，每隔1s持续产生消息
class SimpleStringGenerator extends SourceFunction[String] {
  var running = true
  var i = 0

  override def run(ctx: SourceContext[String]) {
    while (running) {
      ctx.collect("element-" + i)
      i += 1
      Thread.sleep(1000)
    }
  }

  override def cancel() {
    running = false
  }
}

//consumer代码
object ReadFromKafka {
  def main(args: Array[String]) {
    // 打印出执行flink run的参考命令
    System.out.println("use command as: ")
    System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka" +
      " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092")
    System.out.println("************************************************************************************************")
    System.out.println("<topic> is the kafka topic name")
    System.out.println("<bootstrap.servers> is the ip:port list of brokers")
    System.out.println("************************************************************************************************")

    // 构造执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 设置并发度
    env.setParallelism(1)
    // 解析运行参数
    val paraTool = ParameterTool.fromArgs(args)
    // 构造流图，从Kafka读取数据并换行打印
    val messageStream = env.addSource(new FlinkKafkaConsumer(
      paraTool.get("topic"), new SimpleStringSchema, paraTool.getProperties))
    messageStream
      .map(s => "Flink says " + s + System.getProperty("line.separator")).print()
    // 调用execute触发执行
    env.execute()
  }
}
```

10.4.3 Flink 开启 Checkpoint 样例程序

10.4.3.1 Flink 开启 Checkpoint 样例程序开发思路

场景说明

假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性，即：当应用出现异常并恢复后，各个算子的状态能够处于统一的状态。

数据规划

1. 使用自定义算子每秒钟产生大约10000条数据。
2. 产生的数据为一个四元组（Long，String，String，Integer）。
3. 数据经统计后，统计结果打印到终端输出。
4. 打印输出的结果为Long类型的数据。

开发思路

1. source算子每隔1秒钟发送10000条数据，并注入到Window算子中。
2. window算子每隔1秒钟统计一次最近4秒钟内数据数量。
3. 每隔1秒钟将统计结果打印到终端。具体查看方式请参考**查看Flink应用调测结果**。
4. 每隔6秒钟触发一次checkpoint，然后将checkpoint的结果保存到HDFS中。

10.4.3.2 Flink 开启 Checkpoint 样例程序（Java）

功能介绍

假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性。

代码样例

1. 快照数据

   该数据在算子制作快照时用于保存到目前为止算子记录的数据条数。

```
import java.io.Seriablizale;

// 该类作为快照的一部分，保存用户自定义状态
public class UDFState implements Serializable {
    private long count;

    // 初始化用户自定义状态
    public UDFState() {
        count = 0L;
    }

    // 设置用户自定义状态
    public void setState(long count) {
       this.count = count;
    }

    // 获取用户自定义状态
    public long geState() {
        return this.count;
    }
}
```

2. 带checkpoint的数据源

source算子的代码，该段代码每发送10000条数据休息1秒钟，制作快照时将到目前为止已经发送的数据的条数保存在UDFState中；从快照中状态恢复时，读取UDFState中的数据条数并重新赋值给count变量。

```
import org.apache.flink.api.java.tuple.Tuple4;
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed;
import org.apache.flink.streaming.api.functions.source.RichSourceFunction;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

// 该类是带checkpoint的source算子
public class SEventSourceWithChk extends RichSourceFunction<Tuple4<Long, String, String, Integer>>
implements ListCheckpointed<UDFState> {
    private Long count = 0L;
    private boolean isRunning = true;
    private String alphabet =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuv
wxyzABCDEFGHIJKLMNOPQRSTUVWZYX0987654321";

    // 算子的主要逻辑，每秒钟向流图中注入10000个元组
    public void run(SourceContext<Tuple4<Long, String, String, Integer>> ctx) throws Exception {
        Random random = new Random();
        while(isRunning) {
            for (int i = 0; i < 10000; i++) {
                ctx.collect(Tuple4.of(random.nextLong(), "hello-" + count, alphabet, 1))
                count++;
            }
            Thread.sleep(1000);
        }
    }

    // 任务取消时调用
    public void cancel() {
        isRunning = false;
    }

    // 制作自定义快照
    public List<UDFState> snapshotState(long l, long ll) throws Exception {
        UDFState udfState = new UDFState();
        List<UDFState> listState = new ArrayList<UDFState>();
        udfState.setState(count);
        listState.add(udfState);
        return listState;
    }

    // 从自定义快照中恢复数据
    public void restoreState(List<UDFState> list) throws Exception {
        UDFState udfState = list.get(0);
        count = udfState.getState();
    }
}
```

3. 带checkpoint的窗口定义

   该段代码是window算子的代码，每当触发计算时统计窗口中元组数量。

```
import org.apache.flink.api.java.tuple.Tuple;
import org.apache.flink.api.java.tuple.Tuple4;
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed;
import org.apache.flink.streaming.api.functions.windowing.WindowFunction;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

import java.util.ArrayList;
import java.util.List;

// 该类是带checkpoint的window算子
public class WindowStatisticWithChk implements WindowFunction<Tuple4<Long, String, String,
Integer>, Long, Tuple, TimeWindow>, ListCheckpointed<UDFState> {
```

```
    private Long total = 0L;

    // window算子实现逻辑，统计window中元组的个数
    void apply(Tuple key, TimeWindow window, Iterable<Tuple4<Long, String, String, Integer>> input,
               Collector<Long> out) throws Exception {
        long count = 0L;
        for (Tuple4<Long, String, String, Integer> event : input) {
            count++;
        }
        total += count;
        out.collect(count);
    }

    // 制作自定义快照
    public List<UDFState> snapshotState(Long l, Long ll) {
        List<UDFState> listState = new ArrayList<UDFState>();
        UDFState udfState = new UDFState();
        udfState.setState(total);
        listState.add(udfState);
        return listState;
     }

     // 从自定义快照中恢复状态
     public void restoreState(List<UDFState> list) throws Exception {
         UDFState udfState = list.get(0);
         total = udfState.getState();
     }
}
```

4. 应用代码

   该段代码是流图定义代码，具体实现业务流程，另外，代码中窗口的触发时间使用了processing time。

```
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

public class FlinkProcessingTimeAPIChkMain {
    public static void main(String[] args) throws Exception{

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 设置相关配置，并开启checkpoint功能
        env.setStateBackend(new FsStateBackend("hdfs://hacluster/flink/checkpoint/"));
        env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
        env.getCheckpointConfig.setCheckpointInterval(6000);

        // 应用逻辑
        env.addSource(new SEventSourceWithChk())
           .keyBy(0)
           .window(SlidingProcessingTimeWindows.of(Time.seconds(4), Time.seconds(1)))
           .apply(new WindowStatisticWithChk())
           .print()

        env.execute();
    }
}
```

10.4.3.3 Flink 开启 Checkpoint 样例程序（Scala）

功能介绍

假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，并做到状态严格一致性。

代码样例

1. 发送数据形式

```
case class SEvent(id: Long, name: String, info: String, count: Int)
```

2. 快照数据

   该数据在算子制作快照时用于保存到目前为止算子记录的数据条数。

```
// 用户自定义状态
class UDFState extends Serializable{
    private var count = 0L

    // 设置用户自定义状态
    def setState(s: Long) = count = s

    // 获取用户自定状态
    def getState = count
}
```

3. 带checkpoint的数据源

   source算子的代码，该段代码每发送10000条数据休息1秒钟，制作快照时将到目前为止已经发送的数据的条数保存在UDFState中；从快照中状态恢复时，读取UDFState中的数据条数并重新赋值给count变量。

```
import java.util
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed
import org.apache.flink.streaming.api.functions.source.RichSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext

// 该类是带有checkpoint的source算子
class SEventSourceWithChk extends RichSourceFunction[SEvent] with ListCheckpointed[UDFState]{
    private var count = 0L
    private var isRunning = true
    private val alphabet =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuv
wxyzABCDEFGHIJKLMNOPQRSTUVWZYX0987654321"

    // source算子的逻辑，即：每秒钟向流图中注入10000个元组
    override def run(sourceContext: SourceContext[SEvent]): Unit = {
        while(isRunning) {
            for (i <- 0 until 10000) {
                sourceContext.collect(SEvent(1, "hello-"+count, alphabet,1))
                count += 1L
            }
            Thread.sleep(1000)
        }
    }

    // 任务取消时调用
    override def cancel(): Unit = {
        isRunning = false;
    }

     override def close(): Unit = super.close()

     // 制作快照
     override def snapshotState(l: Long, l1: Long): util.List[UDFState] = {
         val udfList: util.ArrayList[UDFState] = new util.ArrayList[UDFState]
         val udfState = new UDFState
         udfState.setState(count)
         udfList.add(udfState)
         udfList
     }

    // 从快照中获取状态
    override def restoreState(list: util.List[UDFState]): Unit = {
        val udfState = list.get(0)
        count = udfState.getState
```

```
    }
}
```

4. 带checkpoint的窗口定义

   该段代码是window算子的代码，每当触发计算时统计窗口中元组数量。

```
import java.util
import org.apache.flink.api.java.tuple.Tuple
import org.apache.flink.streaming.api.checkpoint.ListCheckpointed
import org.apache.flink.streaming.api.scala.function.WindowFunction
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

// 该类是带checkpoint的window算子
class WindowStatisticWithChk extends WindowFunction[SEvent, Long, Tuple, TimeWindow] with
ListCheckpointed[UDFState]{
    private var total = 0L

    // window算子的实现逻辑，即：统计window中元组的数量
    override def apply(key: Tuple, window: TimeWindow, input: Iterable[SEvent], out: Collector[Long]):
Unit = {
        var count = 0L
        for (event <- input) {
            count += 1L
        }
        total += count
        out.collect(count)
    }

    // 制作自定义状态快照
    override def snapshotState(l: Long, l1: Long): util.List[UDFState] = {
        val udfList: util.ArrayList[UDFState] = new util.ArrayList[UDFState]
        val udfState = new UDFState
        udfState.setState(total)
        udfList.add(udfState)
        udfList
    }

    // 从自定义快照中恢复状态
    override def restoreState(list: util.List[UDFState]): Unit = {
        val udfState = list.get(0)
        total = udfState.getState
    }
}
```

5. 应用代码

   该段代码是流图定义代码，具体实现业务流程，另外，代码中窗口的触发时间使用了event time。

```
import com.huawei.rt.flink.core.{SEvent, SEventSourceWithChk, WindowStatisticWithChk}
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend
import org.apache.flink.streaming.api.functions.AssignerWithPeriodicWatermarks
import org.apache.flink.streaming.api.{CheckpointingMode, TimeCharacteristic}
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.watermark.Watermark
import org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.api.scala._
import org.apache.flink.runtime.state.filesystem.FsStateBackend
import org.apache.flink.streaming.api.environment.CheckpointConfig.ExternalizedCheckpointCleanup

object FlinkEventTimeAPIChkMain {
    def main(args: Array[String]): Unit ={
        val env = StreamExecutionEnvironment.getExecutionEnvironment
        env.setStateBackend(new FsStateBackend("hdfs://hacluster/flink/checkpoint/"))
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
        env.getConfig.setAutoWatermarkInterval(2000)
        env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
        env.getCheckpointConfig.setCheckpointInterval(6000)
```

```
    // 应用逻辑
    env.addSource(new SEventSourceWithChk)
      .assignTimestampsAndWatermarks(new AssignerWithPeriodicWatermarks[SEvent] {
        // 设置watermark
        override def getCurrentWatermark: Watermark = {
          new Watermark(System.currentTimeMillis())
        }
        // 给每个元组打上时间戳
        override def extractTimestamp(t: SEvent, l: Long): Long = {
          System.currentTimeMillis()
        }
      })
      .keyBy(0)
      .window(SlidingEventTimeWindows.of(Time.seconds(4), Time.seconds(1)))
      .apply(new WindowStatisticWithChk)
      .print()
    env.execute()
  }
}
```

10.4.4 Flink Job Pipeline 样例程序

10.4.4.1 Flink Job Pipeline 样例程序开发思路

场景说明

本样例中发布者Job自己每秒钟产生10000条数据，然后经由该job的NettySink算子向下游发送。另外两个Job作为订阅者，分别订阅一份数据。

数据规划

1. 发布者Job使用自定义算子每秒钟产生10000条数据
2. 数据包含两个属性：分别是Int和String类型
3. 配置文件
   - nettyconnector.registerserver.topic.storage：设置NettySink的IP、端口及并发度信息在第三方注册服务器上的路径（必填），例如：
     ```
     nettyconnector.registerserver.topic.storage: /flink/nettyconnector
     ```
   - nettyconnector.sinkserver.port.range：设置NettySink的端口范围（必填），例如：
     ```
     nettyconnector.sinkserver.port.range: 28444-28943
     ```
   - nettyconnector.sinkserver.subnet：设置网络所属域，例如：
     ```
     nettyconnector.sinkserver.subnet: 10.162.0.0/16
     ```
4. 接口说明
   - 注册服务器接口

     注册服务器用来保存NettySink的IP、端口以及并发度信息，以便NettySource连接使用。为用户提供以下接口：

     ```
     public interface RegisterServerHandler {

     /**
       * 启动注册服务器
       * @param configuration Flink的Configuration类型
       */
     void start(Configuration configuration) throws Exception;
     /**
           *注册服务器上创建Topic节点（目录）
           * @param topic topic节点名称
           */
     ```

```
void createTopicNode(String topic) throw Exception;
/**
*将信息注册到某个topic节点（目录）下
* @param topic 需要注册到的目录
* @param registerRecord 需要注册的信息
*/
void register(String topic, RegisterRecord registerRecord) throws Exception;
/**
      *删除topic节点
      * @param topic 待删除topic
      */
   void deleteTopicNode(String topic) throws Exception;
/**
  *注销注册信息
  *@param topic 注册信息所在的topic
  *@param recordId 待注销注册信息ID
  */
void unregister(String topic, int recordId) throws Exception;
/**
   * 查寻信息
* @param 查询信息所在的topic
*@recordId 查询信息的ID
*/
RegisterRecord query(String topic, int recordId) throws Exception;
/**
   * 查询某个Topic是否存在
   * @param topic
   */
Boolean isExist(String topic) throws Exception;
/**
   *关闭注册服务器句柄
  */
void shutdown() throws Exception;
```

工程基于以上接口提供了ZookeeperRegisterHandler供用户使用。

- NettySink算子

```
Class NettySink(String name,
String topic,
RegisterServerHandler registerServerHandler,
int numberOfSubscribedJobs)
```

  - name：为本NettySink的名称。
  - topic：为本NettySink产生数据的Topic，每个不同的NettySink（并发度除外）必须使用不同的TOPIC，否则会引起订阅混乱，数据无法正常分发。
  - registerServerHandler：为注册服务器的句柄。
  - numberOfSubscribedJobs：为订阅本NettySink的作业数量，该数量必须是明确的，只有当所有订阅者都连接上NettySink，NettySink才发送数据。

- NettySource算子

```
Class  NettySource(String name,
String topic,
RegisterServerHandler registerServerHandler)
```

  - name：为本NettySource的名称，该NettySource必须是唯一的（并发度除外），否则，连接NettySink时会出现冲突，导致无法连接。
  - topic：订阅的NettySink的topic。
  - registerServerHandler：为注册服务器的句柄。

说明

NettySource的并发度必须与NettySource的并发度相同，否则无法正常创建连接。

开发思路

1. 一个Job作为发布者Job，其余两个作为订阅者Job。
2. 发布者Job自己产生数据将其转化成byte[],分别向订阅者发送。
3. 订阅者收到byte[]之后将其转化成String类型，并抽样打印输出。

10.4.4.2 Flink Job Pipeline 样例程序（Java）

下面列出的主要逻辑代码作为演示。

完整代码请参阅：

- com.huawei.bigdata.flink.examples.UserSource。
- com.huawei.bigdata.flink.examples.TestPipelineNettySink。
- com.huawei.bigdata.flink.examples.TestPipelineNettySource1。
- com.huawei.bigdata.flink.examples.TestPipelineNettySource2。

1. 发布Job自定义Source算子产生数据

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction;

import java.io.Serializable;

public class UserSource extends RichParallelSourceFunction<Tuple2<Integer, String>> implements
Serializable {

    private boolean isRunning = true;

    public void open(Configuration configuration) throws Exception {
        super.open(configuration);

    }
/**
    * 数据产生函数，每秒钟产生10000条数据
   */
    public void run(SourceContext<Tuple2<Integer, String>> ctx) throws Exception {

        while(isRunning) {
            for (int i = 0; i < 10000; i++) {
                ctx.collect(Tuple2.of(i, "hello-" + i));
            }
            Thread.sleep(1000);
        }
    }

    public void close() {
        isRunning = false;
    }

    public void cancel() {
        isRunning = false;
    }
}
```

2. 发布者代码

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.netty.sink.NettySink;
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler;

public class TestPipelineNettySink {

    public static void main(String[] args) throws Exception{

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
//设置job的并发度为2
        env.setBufferTimeout(2);

// 创建Zookeeper的注册服务器handler
        ZookeeperRegisterServerHandler zkRegisterServerHandler = new ZookeeperRegisterServerHandler();
// 添加自定义Source算子
        env.addSource(new UserSource())
                .keyBy(0)
                .map(new MapFunction<Tuple2<Integer,String>, byte[]>() {
                    //将发送信息转化成字节数组
@Override
                    public byte[] map(Tuple2<Integer, String> integerStringTuple2) throws Exception {
                        return integerStringTuple2.f1.getBytes();
                    }
                }).addSink(new NettySink("NettySink-1", "TOPIC-2", zkRegisterServerHandler, 2));//通过NettySink
发送出去。

        env.execute();

    }
}
```

3. 第一个订阅者

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.netty.source.NettySource;
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler;

import java.nio.charset.Charset;

public class TestPipelineNettySource1 {

    public static void main(String[] args) throws Exception{

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
// 设置job的并发度为2
env.setParallelism(2);

// 创建Zookeeper的注册服务器句柄
        ZookeeperRegisterServerHandler zkRegisterServerHandler = new ZookeeperRegisterServerHandler();
//添加NettySource算子，接收来自发布者的消息
        env.addSource(new NettySource("NettySource-1", "TOPIC-2", zkRegisterServerHandler))
                .map(new MapFunction<byte[], String>() {
                  // 将接收到的字节流转化成字符串
    @Override
                    public String map(byte[] bytes) {
                        return new String(bytes, Charset.forName("UTF-8"));
                    }
                }).print();

        env.execute();
    }
}
```

4. 第二个订阅者

```
package com.huawei.bigdata.flink.examples;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.netty.source.NettySource;
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler;

import java.nio.charset.Charset;

public class TestPipelineNettySource2 {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
// 设置作业的并发度为2
 env.setParallelism(2);

//创建Zookeeper的注册服务器句柄
        ZookeeperRegisterServerHandler zkRegisterServerHandler = new ZookeeperRegisterServerHandler();
//添加NettySource算子，接收来自发布者的数据
        env.addSource(new NettySource("NettySource-2", "TOPIC-2", zkRegisterServerHandler))
                .map(new MapFunction<byte[], String>() {
          //将接收到的字节数组转化成字符串
                    @Override
                    public String map(byte[] bytes) {
                        return new String(bytes, Charset.forName("UTF-8"));
                    }
                }).print();

        env.execute();
    }
}
```

10.4.4.3 Flink Job Pipeline 样例程序（Scala）

下面列出的主要逻辑代码作为演示。

完整代码请参阅：

- com.huawei.bigdata.flink.examples.UserSource。
- com.huawei.bigdata.flink.examples.TestPipeline_NettySink。
- com.huawei.bigdata.flink.examples.TestPipeline_NettySource1。
- com.huawei.bigdata.flink.examples.TestPipeline_NettySource2。

1. 发送消息

```
package com.huawei.bigdata.flink.examples

case class Inforamtion(index: Int, content: String) {

  def this() = this(0, "")
}
```

2. 发布者job自定义source算子产生数据

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext

class UserSource extends RichParallelSourceFunction[Inforamtion] with Serializable{

  var isRunning = true
```

```
  override def open(parameters: Configuration): Unit = {
    super.open(parameters)

  }

// 每秒钟产生10000条数据
  override def run(sourceContext: SourceContext[Inforamtion]) = {

    while (isRunning) {
      for (i <- 0 until 10000) {
        sourceContext.collect(Inforamtion(i, "hello-" + i));

      }
      Thread.sleep(1000)
    }
  }

  override def close(): Unit = super.close()

  override def cancel() = {
    isRunning = false
  }
}
```

3. 发布者代码

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.netty.sink.NettySink
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler
import org.apache.flink.streaming.api.scala._

object TestPipeline_NettySink {

  def main(args: Array[String]): Unit = {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
// 设置job的并发度为2
env.setParallelism(2)
//设置Zookeeper为注册服务器
    val zkRegisterServerHandler = new ZookeeperRegisterServerHandler
//添加用户自定义算子产生数据
env.addSource(new UserSource)
      .keyBy(0).map(x=>x.content.getBytes)//将发送数据转化成字节数组
      .addSink(new NettySink("NettySink-1", "TOPIC-2", zkRegisterServerHandler, 2))//添加NettySink算子发送
数据

    env.execute()
  }
}
```

4. 第一个订阅者

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.netty.source.NettySource
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler
import org.apache.flink.streaming.api.scala._

import scala.util.Random


object TestPipeline_NettySource1 {

  def main(args: Array[String]): Unit = {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
// 设置Job的并发度为2
  env.setParallelism(2)
```

```
//设置Zookeeper作为注册服务器
val zkRegisterServerHandler = new ZookeeperRegisterServerHandler
//添加NettySource算子，接收来自发布者的数据
    env.addSource(new NettySource("NettySource-1", "TOPIC-2", zkRegisterServerHandler))
      .map(x => (1, new String(x)))//将接收到的字节流转化成字符串
      .filter(x => {
        Random.nextInt(50000) == 10
      })
      .print

    env.execute()
  }
}
```

5. 第二个订阅者

```
package com.huawei.bigdata.flink.examples

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.netty.source.NettySource
import org.apache.flink.streaming.connectors.netty.utils.ZookeeperRegisterServerHandler
import org.apache.flink.streaming.api.scala._

import scala.util.Random


object TestPipeline_NettySource2 {

  def main(args: Array[String]): Unit = {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
//设置job的并发度为2
 env.setParallelism(2)
//创建Zookeeper作为注册服务器
    val zkRegisterServerHandler = new ZookeeperRegisterServerHandler
//添加NettySource算子，接收数据
env.addSource(new NettySource("NettySource-2", "TOPIC-2", zkRegisterServerHandler))
      .map(x=>(2, new String(x)))//将接收到的字节数组转化成字符串
      .filter(x=>{
        Random.nextInt(50000) == 10
      })
      .print()

    env.execute()
  }
}
```

10.4.5 Flink Join 样例程序

10.4.5.1 Flink Join 样例程序开发思路

场景说明

假定某个Flink业务1每秒就会收到1条消息记录，消息记录某个用户的基本信息，包括名字、性别、年龄。另有一个Flink业务2会不定时收到1条消息记录，消息记录该用户的名字、职业信息。

基于某些业务要求，开发的Flink应用程序实现功能：实时的以根据业务2中消息记录的用户名字作为关键字，对两个业务数据进行联合查询。

数据规划

- 业务1的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户），并从Kafka组件接收数据。Kafka配置参见样例**数据规划**章节。

- 业务2的数据通过socket接收消息记录，可使用netcat命令用户输入模拟数据源。
  - 使用Linux命令netcat -l -p <port>，启动一个简易的文本服务器。
  - 启动应用程序连接netcat监测的port成功后，向netcat终端输入数据信息。

开发思路

1. 启动Flink Kafka Producer应用向Kafka发送数据。
2. 启动Flink Kafka Consumer应用从Kafka接收数据，构造Table1，保证topic与producer一致。
3. 从soket中读取数据，构造Table2。
4. 使用Flink SQL对Table1和Table2进行联合查询，并进行打印。

10.4.5.2 Flink Join 样例程序（Java）

功能介绍

在Flink应用中，调用flink-connector-kafka模块的接口，生产并消费数据。

代码样例

用户在开发前需要使用对接安全模式的Kafka，则需要引入kafka-clients-*.jar，该jar包可在client目录下获取。

下面列出producer和consumer，以及Flink Stream SQL Join使用主要逻辑代码作为演示。

完整代码参见com.huawei.bigdata.flink.examples.WriteIntoKafka和com.huawei.bigdata.flink.examples.SqlJoinWithSocket

1. 每秒钟往Kafka中生产一条用户信息，用户信息有姓名、年龄、性别组成。

```
//producer代码
public class WriteIntoKafka {

    public static void main(String[] args) throws Exception {

    // 打印出执行flink run的参考命令
      System.out.println("use command as: ");

      System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +

        " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:9092");

      System.out.println("./bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka" +

        " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21007 --security.protocol SASL_PLAINTEXT --sasl.kerberos.service.name kafka");

      System.out.println("******************************************************************************************");

      System.out.println("<topic> is the kafka topic name");

      System.out.println("<bootstrap.servers> is the ip:port list of brokers");

      System.out.println("******************************************************************************************");

      // 构造执行环境
      StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
      // 设置并发度
      env.setParallelism(1);
```

```
        // 解析运行参数
        ParameterTool paraTool = ParameterTool.fromArgs(args);
        // 构造流图，将自定义Source生成的数据写入Kafka
        DataStream<String> messageStream = env.addSource(new SimpleStringGenerator());

        FlinkKafkaProducer<String> producer = new FlinkKafkaProducer<>(paraTool.get("topic"),

          new SimpleStringSchema(),

          paraTool.getProperties());

        producer.setWriteTimestampToKafka(true);

        messageStream.addSink(producer);

        // 调用execute触发执行
        env.execute();
    }

// 自定义Source，每隔1s持续产生消息
public static class SimpleStringGenerator implements SourceFunction<String> {
        static final String[] NAME = {"Carry", "Alen", "Mike", "Ian", "John", "Kobe", "James"};

        static final String[] SEX = {"MALE", "FEMALE"};

        static final int COUNT = NAME.length;

        boolean running = true;

        Random rand = new Random(47);

        @Override
        //rand随机产生名字，性别，年龄的组合信息
         public void run(SourceContext<String> ctx) throws Exception {

            while (running) {

                int i = rand.nextInt(COUNT);

                int age = rand.nextInt(70);

                String sexy = SEX[rand.nextInt(2)];

                ctx.collect(NAME[i] + "," + age + "," + sexy);

                thread.sleep(1000);

            }

    }

      @Override

      public void cancel() {

        running = false;

      }

    }

  }
```

2. 生成Table1和Table2，并使用Join对Table1和Table2进行联合查询，打印输出结果。

```
public class SqlJoinWithSocket {
    public static void main(String[] args) throws Exception{

        final String hostname;
```

```
        final int port;

        System.out.println("use command as: ");

        System.out.println("flink run --class com.huawei.bigdata.flink.examples.SqlJoinWithSocket" +
                " /opt/test.jar --topic topic-test -bootstrap.servers xxxx.xxx.xxx.xxx:9092 --hostname
xxx.xxx.xxx.xxx --port xxx");

        System.out.println("flink run --class com.huawei.bigdata.flink.examples.SqlJoinWithSocket" +
                " /opt/test.jar --topic topic-test -bootstrap.servers xxxx.xxx.xxx.xxx:21007 --security.protocol
SASL_PLAINTEXT --sasl.kerberos.service.name kafka"
                + "--hostname xxx.xxx.xxx.xxx --port xxx");

        System.out.println("*******************************************************************************************");
        System.out.println("<topic> is the kafka topic name");
        System.out.println("<bootstrap.servers> is the ip:port list of brokers");
        System.out.println("*******************************************************************************************");

        try {
            final ParameterTool params = ParameterTool.fromArgs(args);

            hostname = params.has("hostname") ? params.get("hostname") : "localhost";

            port = params.getInt("port");

        } catch (Exception e) {
            System.err.println("No port specified. Please run 'FlinkStreamSqlJoinExample " +
                    "--hostname <hostname> --port <port>', where hostname (localhost by default) " +
                    "and port is the address of the text server");

            System.err.println("To start a simple text server, run 'netcat -l -p <port>' and " +
                    "type the input text into the command line");

            return;
        }

        EnvironmentSettings fsSettings =
EnvironmentSettings.newInstance().useOldPlanner().inStreamingMode().build();
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, fsSettings);

        //基于EventTime进行处理
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        env.setParallelism(1);

        ParameterTool paraTool = ParameterTool.fromArgs(args);

        //Stream1，从Kafka中读取数据
        DataStream<Tuple3<String, String, String>> kafkaStream = env.addSource(new
FlinkKafkaConsumer<>(paraTool.get("topic"),
                new SimpleStringSchema(),
                paraTool.getProperties())).map(new MapFunction<String, Tuple3<String, String, String>>()
{
            @Override
            public Tuple3<String, String, String> map(String s) throws Exception {
                String[] word = s.split(",");

                return new Tuple3<>(word[0], word[1], word[2]);
            }
        });

        //将Stream1注册为Table1
        tableEnv.registerDataStream("Table1", kafkaStream, "name, age, sexy, proctime.proctime");

        //Stream2，从Socket中读取数据
        DataStream<Tuple2<String, String>> socketStream = env.socketTextStream(hostname, port,
"\n").
                map(new MapFunction<String, Tuple2<String, String>>() {
```

```
                @Override
                public Tuple2<String, String> map(String s) throws Exception {
                    String[] words = s.split("\\s");
                    if (words.length < 2) {
                        return new Tuple2<>();
                    }

                    return new Tuple2<>(words[0], words[1]);
                }
            });

        //将Stream2注册为Table2
        tableEnv.registerDataStream("Table2", socketStream, "name, job, proctime.proctime");

        //执行SQL Join进行联合查询
        Table result = tableEnv.sqlQuery("SELECT t1.name, t1.age, t1.sexy, t2.job, t2.proctime as shiptime
\n" +
                "FROM Table1 AS t1\n" +
                "JOIN Table2 AS t2\n" +
                "ON t1.name = t2.name\n" +
                "AND t1.proctime BETWEEN t2.proctime - INTERVAL '1' SECOND AND t2.proctime +
INTERVAL '1' SECOND");

        //将查询结果转换为Stream，并打印输出
        tableEnv.toAppendStream(result, Row.class).print();

        env.execute();
    }
}
```

10.4.6 Flink 对接云搜索服务（CSS）样例程序

10.4.6.1 Flink 对接云搜索服务（CSS）样例程序开发思路

场景说明

本样例实现了Flink消费一个自定义数据源，并将消费的数据写入Elasticsearch或云搜索服务CSS的功能。

主要提供了Elasticsearch Sink的构建及参数设置方法，实现通过Flink将数据写入Elasticsearch的功能。

Flink支持1.12.0及以后版本，Elasticsearch支持7.x及以后版本但不支持HTTPS的Elasticsearch集群。

数据规划

- 如果使用自定义数据源，需保证源端集群和目标端之间网络端口通信正常。
- 如果使用Kafka，MySQL等外源数据，需要确保对应用户具备数据操作的权限。

开发思路

1. 导入Flink相关依赖包，版本需与集群Flink版本一致。
2. 构建源端数据源。
3. 构建目标端Elasticsearch数据源（可以在构建数据源的时候，通过setRestClientFactory方法，配置自定义实现的UserRestClientFactory）。
4. 构建Flink执行环境。

10.4.6.2 Flink 对接云搜索服务（CSS）样例程序（Java）

功能介绍

当前基于随机数生成器实现了一个持续产生长度为4字符串的数据源用于写入数据。

样例代码

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.flink.examples。

```
public class FlinkEsSinkExample {
  public static void main(String[] args) throws Exception {

    System.out.println("use command as:");
    System.out.println(
        "flink run -m yarn-cluster --class com.huawei.bigdata.flink.examples.FlinkEsSinkExample /<JAR_PATH>/
FlinkEsSinkExample-1.0.jar");
    System.out.println(
        "******************************************************************************************");
    System.out.println("<JAR_PATH> is the path of jar file");
    System.out.println(
        "******************************************************************************************");
    //init flink execution environment
    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    //create a dataSource according to RandomDataGenerator
    //of course, you can create any datasource from kafka, mysql, text file and so on
    DataGeneratorSource<String> dataGeneratorSource =
        new DataGeneratorSource<>(new DataGenerator<String>() {
          RandomDataGenerator generator;

          @Override
          public void open(String name, FunctionInitializationContext context,
              RuntimeContext runtimeContext) throws Exception {
            generator = new RandomDataGenerator();
          }

          @Override
          public boolean hasNext() {
            return true;
          }

          @Override
          public String next() {
            return generator.nextHexString(4);
          }
        });

    //create HtppHost list which are needed by elasticsearch sink
    List<HttpHost> httpHosts = new ArrayList<>();
    httpHosts.add(new HttpHost("172.16.0.117", 9200, "http"));
    //httpHosts.add(new HttpHost("172.16.0.xxx", 9200, "http"));
    ElasticsearchSink.Builder<String> esSinkBuilder =
        new ElasticsearchSink.Builder<>(httpHosts, new ElasticsearchSinkFunction<String>() {
          public IndexRequest createIndexRequest(String element) {
            Map<String, String> json = new HashMap<>();
            json.put("data", element);
            //init index request,put add real data as index request source
            return Requests.indexRequest().index("my-index").id(element).source(json);
          }

          @Override
          public void process(String s, RuntimeContext runtimeContext,
              RequestIndexer requestIndexer) {
            requestIndexer.add(createIndexRequest(s));
          }
        });
```

```
    // configuration for the bulk requests; this instructs the sink to emit after every element, otherwise they
would be buffered
    esSinkBuilder.setBulkFlushMaxActions(1);

    //add data source
    DataStream<String> stringDataStream =
        env.addSource(dataGeneratorSource, "DataGeneratorSource").returns(Types.STRING);
    //add elstic search sink
    stringDataStream.addSink(esSinkBuilder.build());
    env.execute();
  }
}
```

10.5 调测 Flink 应用

10.5.1 编译并调测 Flink 应用

操作场景

在程序代码完成开发后，建议您上传至Linux客户端环境中运行应用。使用Scala或Java语言开发的应用程序在Flink客户端的运行步骤是一样的。

说明

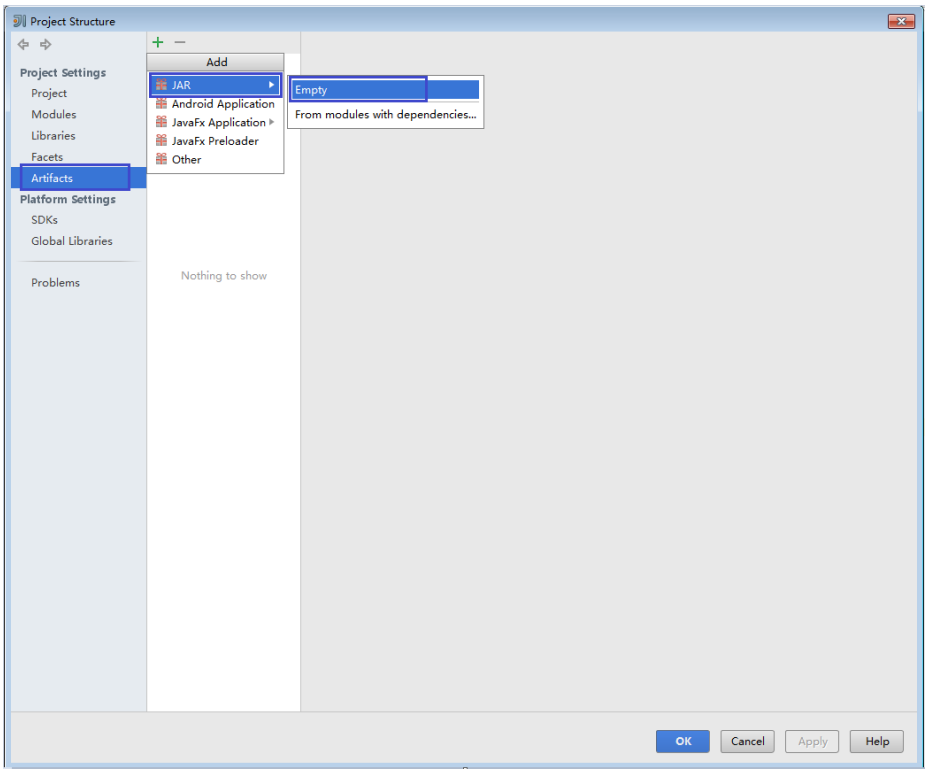
基于YARN集群的Flink应用程序不支持在Windows环境下运行，只支持在Linux环境下运行。

操作步骤

步骤1 在IntelliJ IDEA中，在生成Jar包之前配置工程的Artifacts信息。

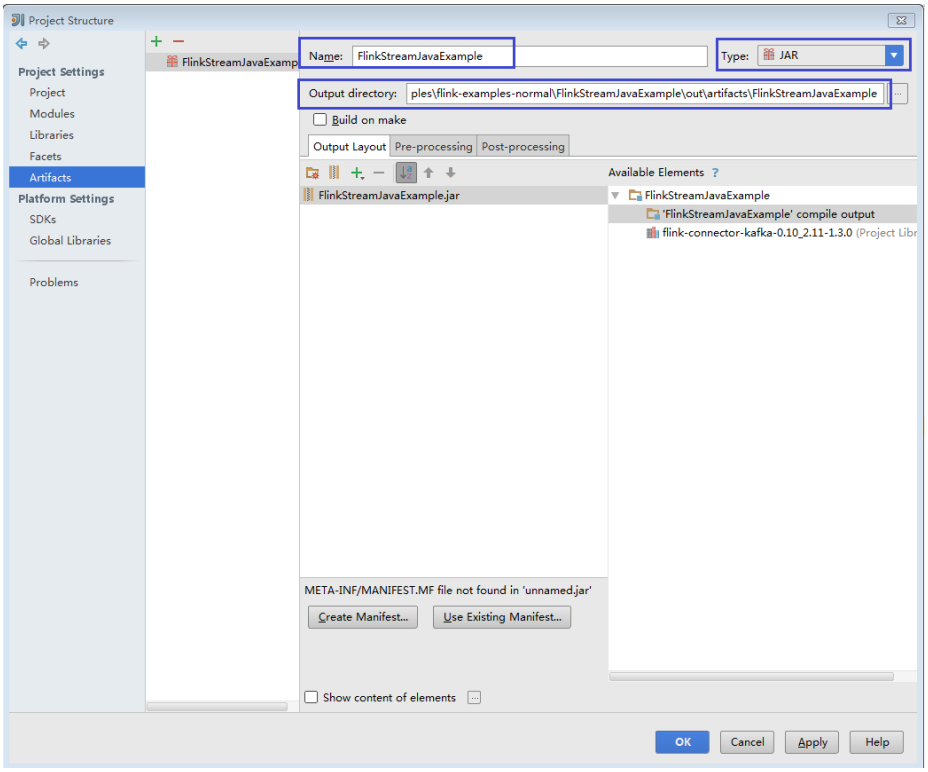
1. 在IDEA主页面，选择“File > Project Structures...”进入“Project Structure”页面。
2. 在“Project Structure”页面，选择“Artifacts”，单击“+”并选择“JAR > Empty”。

图 10-35 添加 Artifacts



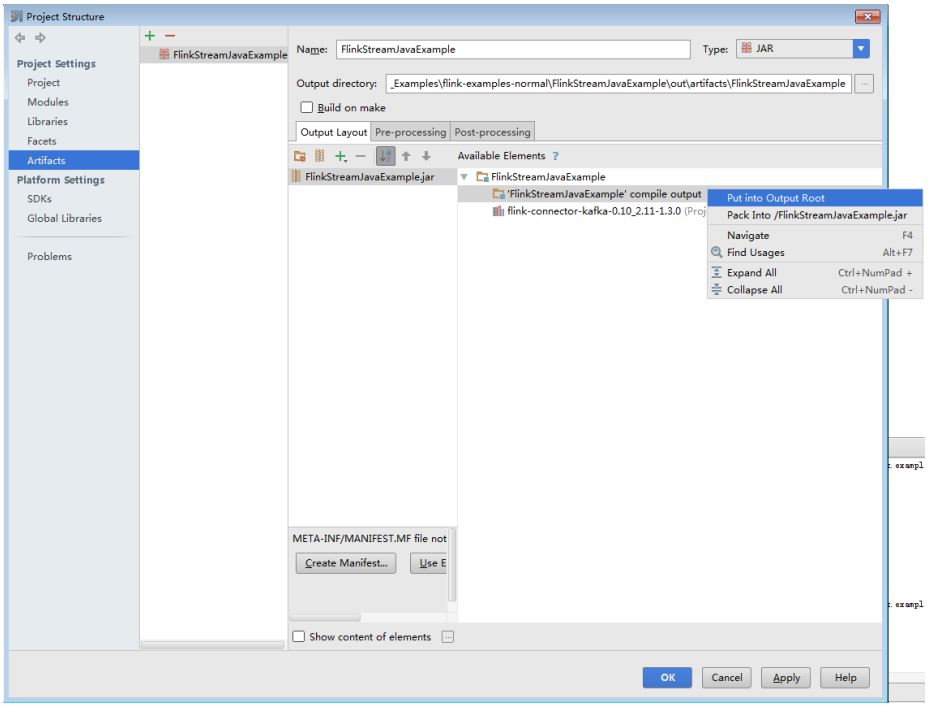
3. 您可以根据实际情况设置Jar包的名称、类型以及输出路径。

图 10-36 设置基本信息



4. 选中“'FlinkStreamJavaExample' compile output”，右键选择“Put into Output Root”。然后单击“Apply”。

图 10-37 Put into Output Root

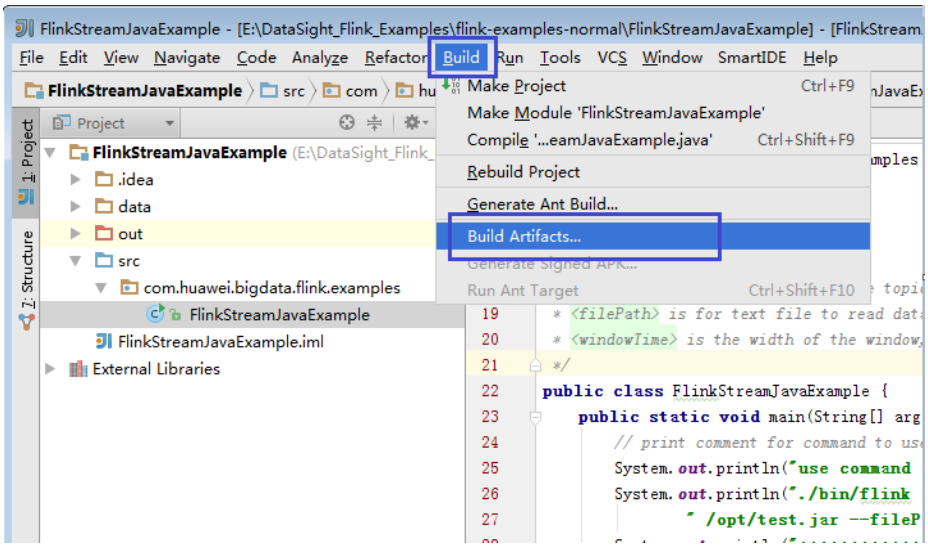


5. 最后单击“OK”完成配置。

步骤2 生成Jar包。

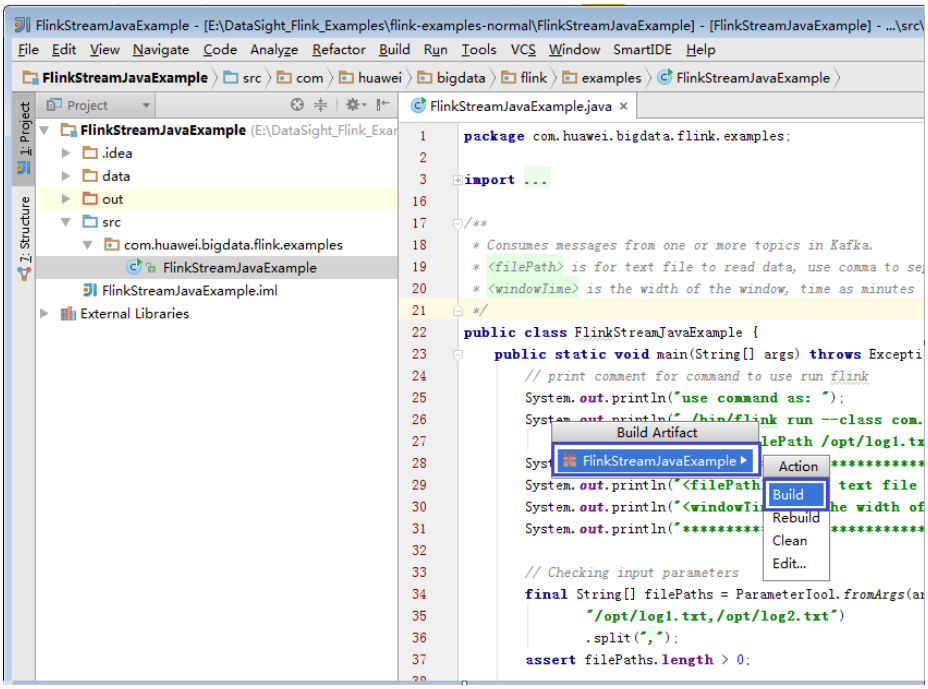
1. 在IDEA主页面，选择“Build > Build Artifacts...”。

图 10-38 Build Artifacts



2. 在弹出的菜单中，选择“FlinkStreamJavaExample > Build”开始生成Jar包。

图 10-39 Build



3. 当Event log中出现如下类似日志时，表示Jar包生成成功。您可以从**步骤1.3**中配置的路径下获取到Jar包。

```
21:25:43 Compilation completed successfully in 36 sec
```

步骤3 将**步骤2**中生成的Jar包（如FlinkStreamJavaExample.jar）复制到Linux环境的Flink运行环境下（即Flink客户端），如“/opt/client”。然后在该目录下创建“conf”目录，将需要的配置文件复制至“conf”目录，具体操作请参考**准备运行环境**，运行Flink应用程序。

在Linux环境中运行Flink应用程序，需要先启动Flink集群。在Flink客户端下执行yarn session命令，启动flink集群。执行命令例如：

```
bin/yarn-session.sh -jm 1024 -tm 1024
```

说明

- 执行yarn-session.sh之前，应预先将Flink应用程序的运行依赖包复制到客户端目录{client_install_home}/Flink/flink/lib下，应用程序运行依赖包请参考**参考信息**。
- 不同的样例工程使用的依赖包可能会有冲突，在运行新的样例工程时需删除旧的样例工程复制至户端目录{client_install_home}/Flink/flink/lib下的依赖包。
- 执行yarn-session.sh之前，请在客户端安装目录执行**source bigdata_env**命令。
- yarn-session.sh命令需进入“/*Flink客户端安装目录*/Flink/flink”目录执行，例如“/opt/client/Flink/flink”。
- 在Flink任务运行过程中禁止重启HDFS服务或者重启所有DataNode实例，否则可能会导致任务失败，并可能导致应用部分临时数据无法清空。
- 请确保Jar包和配置文件的用户权限与Flink客户端一致，例如都是**omm**用户，且权限为755。

- 运行DataStream（Scala和Java）样例程序。

  在终端另开一个窗口，进入Flink客户端目录，调用bin/flink run脚本运行代码。

  – Java

  ```
  bin/flink run --class com.huawei.bigdata.flink.examples.FlinkStreamJavaExample /opt/client/FlinkStreamJavaExample.jar --filePath /opt/log1.txt,/opt/log2.txt --windowTime 2
  ```

- Scala

```
bin/flink run --class com.huawei.bigdata.flink.examples.FlinkStreamScalaExample /opt/client/FlinkStreamScalaExample.jar --filePath /opt/log1.txt,/opt/log2.txt --windowTime 2
```

说明

“log1.txt”、“log2.txt”需放在每个部署了Yarn NodeManager实例的节点上，权限为755。

表 10-7 参数说明

参数名称	说明
<filePath>	指本地文件系统中文件路径，每个节点都需要放一份/opt/log1.txt和/opt/log2.txt。可以默认，也可以设置。
<windowTime>	指窗口时间大小，以分钟为单位。可以默认，也可以设置。

- 运行向Kafka生产并消费数据样例程序（Scala和Java语言）。

  生产数据的执行命令启动程序。

```
bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka /opt/client/FlinkKafkaJavaExample.jar <topic> <bootstrap.servers> [security.protocol] [sasl.kerberos.service.name] [ssl.truststore.location] [ssl.truststore.password]
```

  消费数据的执行命令启动程序。

```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/FlinkKafkaJavaExample.jar <topic> <bootstrap.servers> [security.protocol] [sasl.kerberos.service.name] [ssl.truststore.location] [ssl.truststore.password]
```

表 10-8 参数说明

参数名称	说明	是否必须配置
topic	表示Kafka主题名。	是
bootstrap.server	表示broker集群ip/port列表。	是

参数名称	说明	是否必须配置
security.protocol	运行参数可以配置为PLAINTEXT（可不配置）/SASL_PLAINTEXT/SSL/SASL_SSL四种协议，分别对应FusionInsight Kafka集群的21005/21007/21008/21009端口。 – 如果配置了SASL，则必须配置sasl.kerberos.service.name为kafka，并在conf/flink-conf.yaml中配置security.kerberos.login相关配置项。 – 如果配置了SSL，则必须配置ssl.truststore.location和ssl.truststore.password，前者表示truststore的位置，后者表示truststore密码。	否 **说明** – 该参数未配置时为非安全Kafka。 – 如果需要配置SSL，truststore.jks文件生成方式可参考“Kafka开发指南 > 客户端SSL加密功能使用说明”章节。

四种类型实际命令示，以ReadFromKafka为例，集群域名为“HADOOP.COM”：

– 命令1：
```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:9092
```
– 命令2：
```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:21005 --security.protocol PLAINTEXT --sasl.kerberos.service.name kafka --kerberos.domain.name hadoop.hadoop.com
```
– 命令3：
```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:9093 --security.protocol SSL --ssl.truststore.location /home/truststore.jks --ssl.truststore.password xxx
```
– 命令4：
```
bin/flink run --class com.huawei.bigdata.flink.examples.ReadFromKafka /opt/client/FlinkKafkaJavaExample.jar --topic topic1 --bootstrap.servers 10.96.101.32:21005 --security.protocol PLAINTEXT --sasl.kerberos.service.name kafka --ssl.truststore.location config/truststore.jks --ssl.truststore.password xxx --kerberos.domain.name hadoop.hadoop.com
```

- 运行异步Checkpoint机制样例程序（Scala和Java语言）。

  为了丰富样例代码，Java版本使用了Processing Time作为数据流的时间戳，而Scala版本使用Event Time作为数据流的时间戳。具体执行命令参考如下：

  将Checkpoint的快照信息保存到HDFS。

- Java
  ```
  bin/flink run --class com.huawei.bigdata.flink.examples.FlinkProcessingTimeAPIMain /opt/client/FlinkCheckpointJavaExample.jar --chkPath hdfs://hacluster/flink/checkpoint/
  ```
- Scala
  ```
  bin/flink run --class com.huawei.bigdata.flink.examples.FlinkEventTimeAPIChkMain /opt/client/FlinkCheckpointScalaExample.jar --chkPath hdfs://hacluster/flink/checkpoint/
  ```

说明

- Checkpoint源文件路径：flink/checkpoint/fd5f5b3d08628d83038a30302b611/chk-X/4f854bf4-ea54-4595-a9d9-9b9080779ffe

  flink/checkpoint //指定的根目录。

  fd5f5b3d08628d83038a30302b611 //以jobID命名的第二层目录。

  chk-X // "X"为checkpoint编号，第三层目录。

  4f854bf4-ea54-4595-a9d9-9b9080779ffe //checkpoint源文件。
- Flink在集群模式下checkpoint将文件放到HDFS，本地路径只支持Flink的local模式，便于调测。

- 运行Pipeline样例程序。
  - Java
    i. 启动发布者Job
    ```
    bin/flink run -p 2 --class com.huawei.bigdata.flink.examples.TestPipelineNettySink /opt/client/FlinkPipelineJavaExample.jar
    ```
    ii. 启动订阅者Job1
    ```
    bin/flink run --class com.huawei.bigdata.flink.examples.TestPipelineNettySource1 /opt/client/FlinkPipelineJavaExample.jar
    ```
    iii. 启动订阅者Job2
    ```
    bin/flink run --class com.huawei.bigdata.flink.examples.TestPipelineNettySource2 /opt/client/FlinkPipelineJavaExample.jar
    ```
  - Scala
    i. 启动发布者Job
    ```
    bin/flink run -p 2 --class com.huawei.bigdata.flink.examples.TestPipeline_NettySink /opt/client/FlinkPipelineScalaExample.jar
    ```
    ii. 启动订阅者Job1
    ```
    bin/flink run --class com.huawei.bigdata.flink.examples.TestPipeline_NettySource1 /opt/client/FlinkPipelineScalaExample.jar
    ```
    iii. 启动订阅者Job2
    ```
    bin/flink run --class com.huawei.bigdata.flink.examples.TestPipeline_NettySource2 /opt/client/FlinkPipelineScalaExample.jar
    ```
- 运行Stream SQL Join样例程序。
  a. 启动程序向Kafka生产，Kafka配置可参考**运行向Kafka生产并消费数据...**
  ```
  bin/flink run --class com.huawei.bigdata.flink.examples.WriteIntoKafka /opt/client/FlinkStreamSqlJoinExample.jar --topic topic-test --bootstrap.servers xxx.xxx.xxx.xxx:9092
  ```
  b. 在集群内任一节点启动netcat命令，等待应用程序连接。
  ```
  netcat -l -p 9000
  ```

  说明

  若回显提示“command not found”，请用户自行安装netcat工具后再次执行。

  c. 启动程序接受Socket数据，并执行联合查询。
  ```
  bin/flink run --class com.huawei.bigdata.flink.examples.SqlJoinWithSocket /opt/client/FlinkStreamSqlJoinExample.jar --topic topic-test --bootstrap.servers xxx.xxx.xxx.xxx:9092 --hostname xxx.xxx.xxx.xxx --port 9000
  ```

----结束

10.5.2 查看 Flink 应用调测结果

操作场景

Flink应用程序运行完成后，您可以查看运行结果数据，也可以通过Flink WebUI查看应用程序运行情况。

操作步骤

- **查看Flink应用运行结果数据。**

  当用户查看执行结果时，需要在Flink的web页面上查看Task Manager的Stdout日志。

  当执行结果输出到文件或者其他，由Flink应用程序指定，您可以通过指定文件或其他获取到运行结果数据。以下用Checkpoint、Pipeline和配置表与流JOIN为例。

- **查看Checkpoint结果和文件**
  - 结果在flink的“taskmanager.out”文件中。用户可以进入Yarn的WebUI页面，选择“Jobs > Checkpoints”查看提交的作业如图10-40。选择“Task Managers > Stdout”查看运行结果如图10-41。

    图 10-40 提交的作业

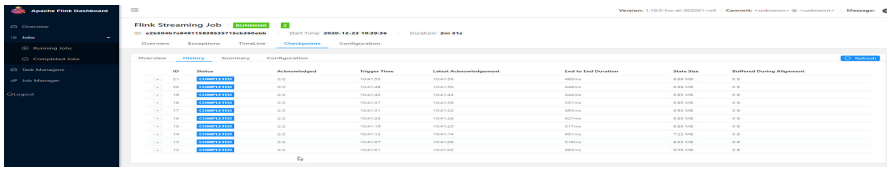
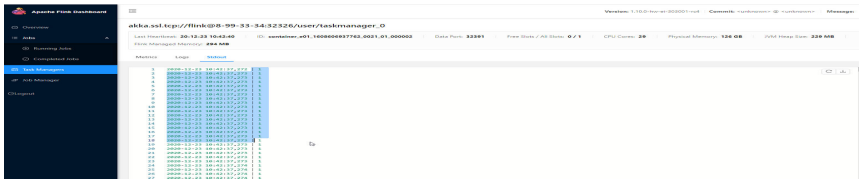
    

    图 10-41 运行结果

  - 通过执行**hdfs dfs -ls hdfs://hacluster/flink/checkpoint/**命令查看HDFS上的checkpoint的快照信息。
  - **查看Pipeline结果**
    - 结果在flink的“taskmanager.out”文件中。用户可以进入Yarn的WebUI页面，选择“Jobs > Running Jobs”查看提交的作业如图10-42。选择“Task Managers ”可以看到有两个任务如图10-43。分别单击任意Task，选择“Stdout”查看该任务的输出结果如图10-44和图10-45。

      图 10-42 提交的作业

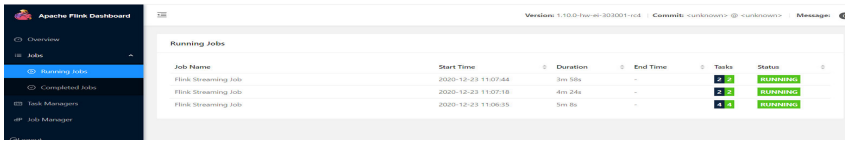
      

图 10-43 提交的任务

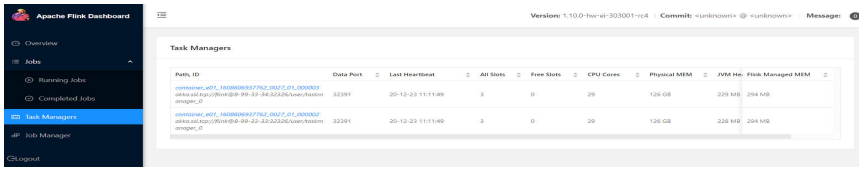


图 10-44 Task1 输出结果

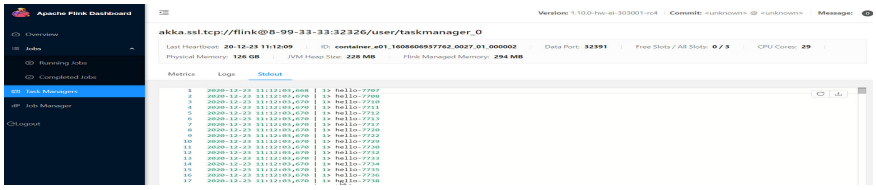
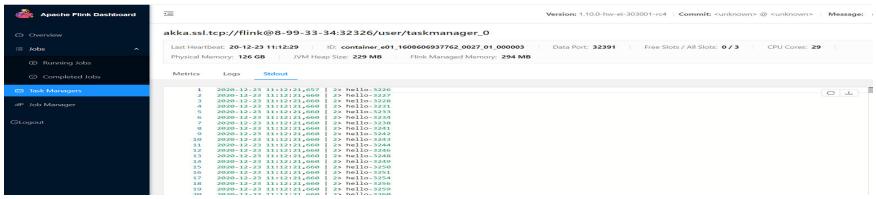


图 10-45 Task2 输出结果



- **查看DataStream结果**
  - 结果在flink的“taskmanager.out”文件中，用户可以进入Yarn的WebUI页面，选择“Jobs > Completed Jobs”查看完成作业如图10-46。选择“Task Managers ”查看提交的任务如图10-47。单击该任务进入该任务详细信息页面，单击“Stdout”查看该任务的输出结果如图10-48。

图 10-46 运行完成的作业

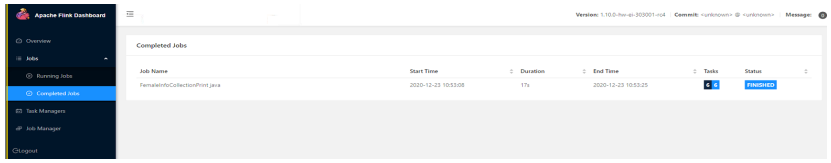


图 10-47 提交的任务

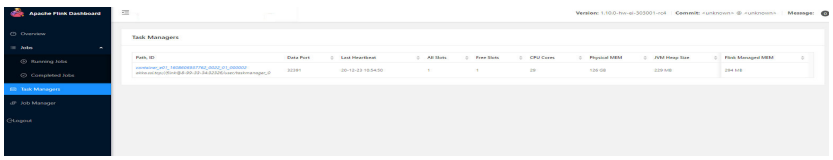
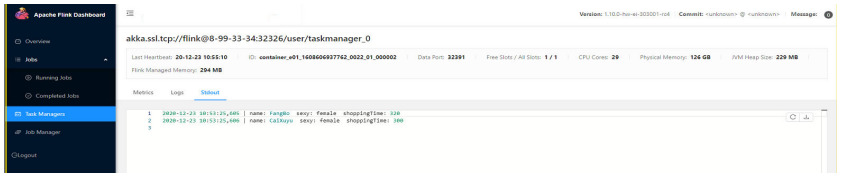


图 10-48 运行结果



- **查看Stream SQL Join结果**

  - 结果在flink的“taskmanager.out”文件中，用户可以进入Yarn的WebUI页面，选择“Jobs > Running Jobs”查看运行的作业如图10-49。选择“Task Managers ”查看提交的任务如图10-50。单击该任务进入该任务详细信息页面，单击“Stdout”查看该任务的输出结果如图10-51。

    图 10-49 运行的作业

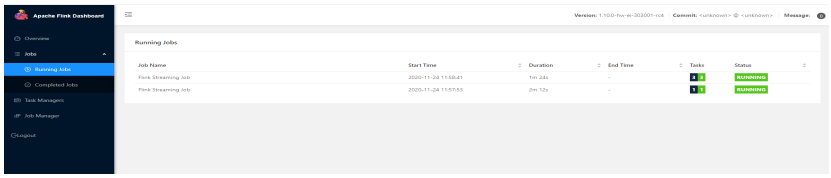
    

    图 10-50 提交的任务

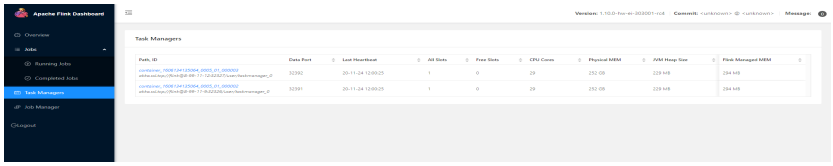
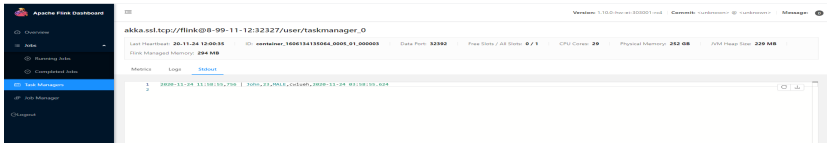
    

    图 10-51 运行结果

- **查看向Kafka生产并消费数据结果**

  - 结果在flink的“taskmanager.out”文件中，用户可以进入Yarn的WebUI页面，选择“Jobs > Running Jobs”查看运行的作业如图10-52。选择“Task Managers ”查看提交的任务如图10-53。单击该任务进入该任务详细信息页面，单击“Stdout”查看该任务的输出结果如图10-54。

    图 10-52 运行的作业

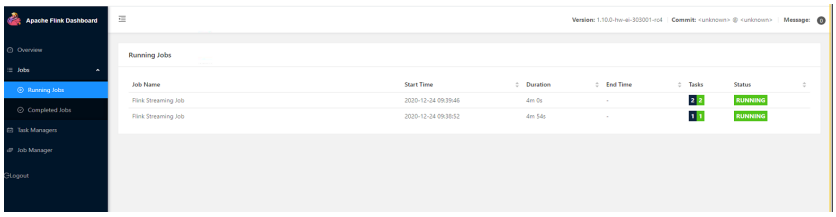
    

    图 10-53 提交的任务

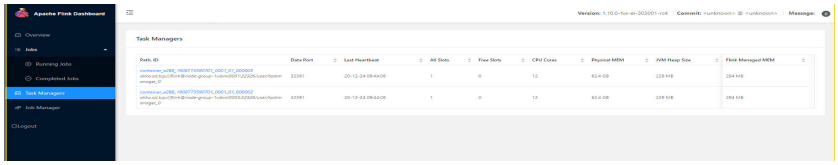
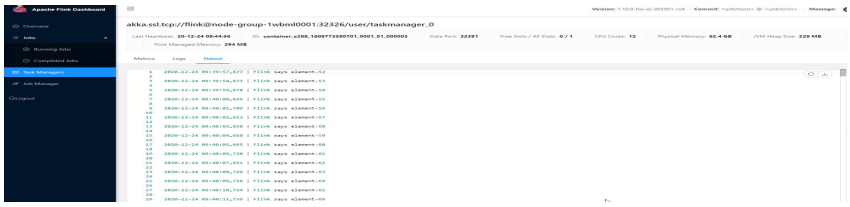
    

图 10-54 运行结果



- **使用Flink Web页面查看Flink应用程序运行情况。**

  Flink Web页面主要包括了Overview、Running Jobs、Completed Jobs、Task Managers、Job Manager和Logout等部分。

  在YARN的Web UI界面，查找到对应的Flink应用程序。单击应用信息的最后一列“ApplicationMaster”，即可进入Flink Web页面。

  查看程序执行的打印结果：找到对应的Task Manager，查看对应的Stdout标签日志信息。

- **查看Flink日志获取应用运行情况。**

  有三种方式获取Flink日志，分别为通过Flink Web页面或者Yarn的日志

  - Flink Web页面可以查看Task Managers、Job Manager部分的日志。
  - Yarn页面主要包括了Job Manager日志以及GC日志等。

    页面入口：在YARN的Web UI界面，查找到对应的Flink应用程序。单击应用信息的第一列ID，然后选择Logs列单击进去即可打开。
  - 使用Yarn客户端获取或查看Task Managers、Job Manager的日志，具体操作如下：

    i. 下载并安装Yarn客户端（例安装目录：/opt/client）。

    ii. 以客户端安装用户，登录安装客户端的节点。

    iii. 执行以下命令，切换到客户端安装目录。

    cd /opt/client

    iv. 执行以下命令配置环境变量。

    source bigdata_env

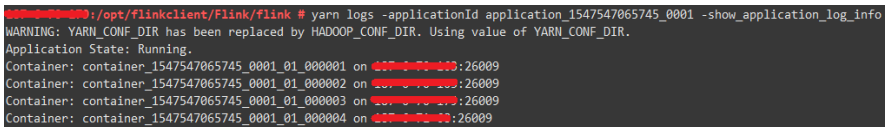
    v. 如果集群为安全模式，执行以下命令进行用户认证。普通模式集群无需执行用户认证。

    kinit *组件业务用户*

    vi. 执行以下命令，获取Flink集群container信息。

    yarn logs -applicationId application_* -show_application_log_info

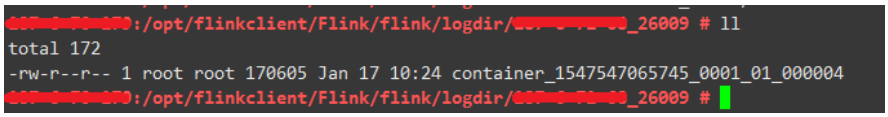
    图 10-55 获取 Flink 集群 container 信息

    vii. 执行以下命令，获取指定container运行日志，通常container_*_000001为JobManager运行所在container。

    yarn logs -applicationId application_* --containerId container_1547547065745_0001_01_000004 -out logdir/

图 10-56 获取指定 container 运行日志



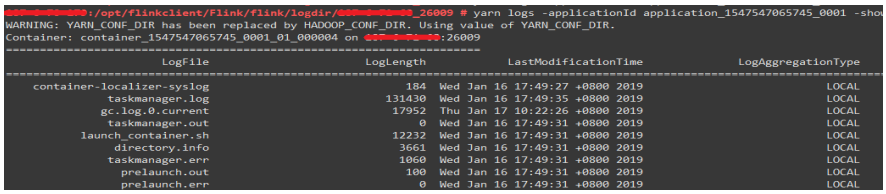
上述命令会将container运行日志下载至本地，该日志包含了TaskManager/JobManager的运行日志，GC日志等信息。

viii. 还可以使用如下命令获取指定名称日志。

获取container日志列表：

yarn logs -applicationId application_* -show_container_log_info --containerId container_1547547065745_0001_01_000004

图 10-57 获取 container 日志列表



下载指定日志taskmanager.log至本地：

yarn logs -applicationId application_* --containerId container_1547547065745_0001_01_000004 -log_files taskmanager.log -out localpath

10.6 Flink 应用开发常见问题

10.6.1 Flink 常用 API 介绍

10.6.1.1 Flink Java API 接口介绍

由于Flink开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

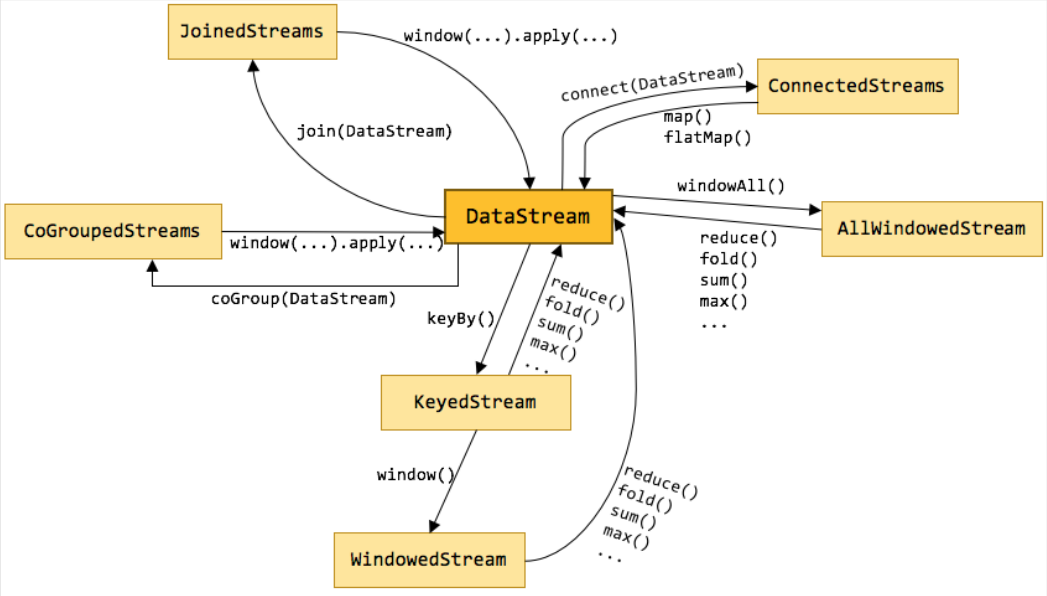
Flink 常用接口

Flink主要使用到如下这几个类：

- StreamExecutionEnvironment：是Flink流处理的基础，提供了程序的执行环境。
- DataStream：Flink用类DataStream来表示程序中的流式数据。用户可以认为它们是含有重复数据的不可修改的集合(collection)，DataStream中元素的数量是无限的。
- KeyedStream：DataStream通过keyBy分组操作生成流，通过设置的key值对数据进行分组。
- WindowedStream：KeyedStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。

- AllWindowedStream：DataStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- ConnectedStreams：将两条DataStream流连接起来并且保持原有流数据的类型，然后进行map或者flatMap操作。
- JoinedStreams：在窗口上对数据进行等值join操作（等值就是判断两个值相同的join，比如a.id = b.id），join操作是coGroup操作的一种特殊场景。
- CoGroupedStreams：在窗口上对数据进行coGroup操作，可以实现流的各种join类型。

图 10-58 Flink Stream 的各种流类型转换



流数据输入

表 10-9 流数据输入的相关接口

API	说明
public final <OUT> DataStreamSource<OUT> fromElements(OUT... data)	获取用户定义的多个元素的数据，作为输入流数据。 • type为元素的数据类型。 • data为多个元素的具体数据。
public final <OUT> DataStreamSource<OUT> fromElements(Class<OUT> type, OUT... data)	

API	说明
public <OUT> DataStreamSource<OUT> fromCollection(Collection<OUT> data)	获取用户定义的集合数据，作为输入流数据。 ● type为集合中元素的数据类型。 ● typeInfo为集合中根据元素数据类型获取的类型信息。 ● data为集合数据或者可迭代的数据体。
public <OUT> DataStreamSource<OUT> fromCollection(Collection<OUT> data, TypeInformation<OUT> typeInfo)	
public <OUT> DataStreamSource<OUT> fromCollection(Iterator<OUT> data, Class<OUT> type)	
public <OUT> DataStreamSource<OUT> fromCollection(Iterator<OUT> data, TypeInformation<OUT> typeInfo)	
public <OUT> DataStreamSource<OUT> fromParallelCollection(SplittableIterator<OUT> iterator, Class<OUT> type)	获取用户定义的集合数据，作为输入并行流数据。 ● type为集合中元素的数据类型。 ● typeInfo为集合中根据元素数据类型获取的类型信息。 ● iterator为可被分割成多个partition的迭代数据体。
public <OUT> DataStreamSource<OUT> fromParallelCollection(SplittableIterator<OUT> iterator, TypeInformation<OUT> typeInfo)	
private <OUT> DataStreamSource<OUT> fromParallelCollection(SplittableIterator<OUT> iterator, TypeInformation<OUT> typeInfo, String operatorName)	
public DataStreamSource<Long> generateSequence(long from, long to)	获取用户定义的一串序列数据，作为输入流数据。 ● from是指数值串的起点。 ● to是指数值串的终点。
public DataStreamSource<String> readTextFile(String filePath)	获取用户定义的某路径下的文本文件数据，作为输入流数据。 ● filePath是指文本文件的路径。 ● charsetName是编码格式的名字。
public DataStreamSource<String> readTextFile(String filePath, String charsetName)	

API	说明
public <OUT> DataStreamSource<OUT> readFile(FileInputFormat<OUT> inputFormat, String filePath)	获取用户定义的某路径下的文件数据，作为输入流数据。 ● filePath是指文件的路径。 ● inputFormat是指文件的格式。 ● watchType指的是文件的处理模式“PROCESS_ONCE”或者“PROCESS_CONTINUOUSLY”。 ● interval指的是多长时间判断目录或文件变化进行处理。
public <OUT> DataStreamSource<OUT> readFile(FileInputFormat<OUT> inputFormat, String filePath, FileProcessingMode watchType, long interval)	
public <OUT> DataStreamSource<OUT> readFile(FileInputFormat<OUT> inputFormat, String filePath, FileProcessingMode watchType, long interval, TypeInformation<OUT> typeInformation)	
public DataStreamSource<String> socketTextStream(String hostname, int port, String delimiter, long maxRetry)	获取用户定义的Socket数据，作为输入流数据。 ● hostname是指Socket的服务器端的主机名称。 ● port指的是服务器的监测端口。 ● delimiter指的是消息之间的分隔符。 ● maxRetry指的是由于连接异常可以触发的最大重试次数。
public DataStreamSource<String> socketTextStream(String hostname, int port, String delimiter)	
public DataStreamSource<String> socketTextStream(String hostname, int port)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function)	用户自定义SourceFunction，addSource方法可以添加Kafka等数据源，主要实现方法为SourceFunction的run。 ● function指的是用户自定义的SourceFunction函数。 ● sourceName指的是定义该数据源的名称。 ● typeInfo则是根据元素数据类型获取的类型信息。
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, String sourceName)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, TypeInformation<OUT> typeInfo)	
public <OUT> DataStreamSource<OUT> addSource(SourceFunction<OUT> function, String sourceName, TypeInformation<OUT> typeInfo)	

数据输出

表 10-10 数据输出的相关接口

API	说明
public DataStreamSink<T> print()	数据输出以标准输出流打印出来。
public DataStreamSink<T> printToErr()	数据输出以标准error输出流打印出来。
public DataStreamSink<T> writeAsText(String path)	数据输出写入到某个文本文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。
public DataStreamSink<T> writeAsText(String path, WriteMode writeMode)	
public DataStreamSink<T> writeAsCsv(String path)	数据输出写入到某个csv格式的文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。 ● rowDelimiter为行分隔符。 ● fieldDelimiter为列分隔符。
public DataStreamSink<T> writeAsCsv(String path, WriteMode writeMode)	
public <X extends Tuple> DataStreamSink<T> writeAsCsv(String path, WriteMode writeMode, String rowDelimiter, String fieldDelimiter)	
public DataStreamSink<T> writeToSocket(String hostName, int port, SerializationSchema<T> schema)	数据输出写入到Socket连接中。 ● hostName为主机名称。 ● port为端口。
public DataStreamSink<T> writeUsingOutputFormat(OutputFormat<T> format)	数据输出到普通文件中，例如二进制文件。
public DataStreamSink<T> addSink(SinkFunction<T> sinkFunction)	用户自定义的数据输出，addSink方法可以添加Kafka等数据输出，主要实现方法为SinkFunction的invoke方法。

过滤和映射能力

表 10-11 过滤和映射能力的相关接口

API	说明
public <R> SingleOutputStreamOperator<R> map(MapFunction<T, R> mapper)	输入一个元素，生成另一个元素，元素类型不变。

API	说明
public <R> SingleOutputStreamOperator<R> flatMap(FlatMapFunction<T, R> flatMapper)	输入一个元素，生成零个、一个或者多个元素。
public SingleOutputStreamOperator<T> filter(FilterFunction<T> filter)	对每个元素执行一个布尔函数，只保留返回true的元素。

聚合能力

表 10-12 聚合能力的相关接口

API	说明
public KeyedStream<T, Tuple> keyBy(int... fields)	将流逻辑分区成不相交的分区，每个分区包含相同key的元素。内部是用hash分区来实现的。这个转换返回了一个KeyedStream。 KeyBy操作之后返回KeyedStream，然后再调用KeyedStream的函数（例如reduce/fold/min/minby/max/maxby/sum/sumby等）进行相应操作。 ● fields为数据某几列的序号或者成员变量的名称。 ● key则为用户自定义的指定分区依据的方法。
public KeyedStream<T, Tuple> keyBy(String... fields)	
public <K> KeyedStream<T, K> keyBy(KeySelector<T, K> key)	
public SingleOutputStreamOperator<T> reduce(ReduceFunction<T> reducer)	在一个KeyedStream上“滚动”reduce。合并当前元素与上一个被reduce的值，然后输出新的值。注意三者的类型是一致的。
public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> folder)	在一个KeyedStream上基于一个初始值“滚动”折叠。合并当前元素和上一个被折叠的值，然后输出新值。注意Fold的输入值与返回值类型可以不一致。
public SingleOutputStreamOperator<T> sum(int positionToSum)	在一个KeyedStream上滚动求和操作。 positionToSum和field代表对某一列求和。
public SingleOutputStreamOperator<T> sum(String field)	

API	说明
public SingleOutputStreamOperator<T> min(int positionToMin)	在一个KeyedStream上滚动求最小值。min返回了最小值，不保证非最小值列的准确性。 positionToMin和field代表对某一列求最小值。
public SingleOutputStreamOperator<T> min(String field)	
public SingleOutputStreamOperator<T> max(int positionToMax)	在一个KeyedStream上滚动求最大值。max返回了最大值，不保证非最大值列的准确性。 positionToMax和field代表对某一列求最大值。
public SingleOutputStreamOperator<T> max(String field)	
public SingleOutputStreamOperator<T> minBy(int positionToMinBy)	在一个KeyedStream上求某一列最小值所在的该行数据，minBy返回了该行数据的所有元素。 • positionToMinBy代表对哪一列做minBy操作。 • first表示是否按最先遇到的最小值输出还是最后遇到的最小值输出。
public SingleOutputStreamOperator<T> minBy(String positionToMinBy)	
public SingleOutputStreamOperator<T> minBy(int positionToMinBy, boolean first)	
public SingleOutputStreamOperator<T> minBy(String field, boolean first)	
public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy)	在一个KeyedStream上求某一列最大值所在的该行数据，maxBy返回了该行数据的所有元素。 • positionToMaxBy代表对哪一列做maxBy操作。 • first表示是否按最先遇到的最大值输出还是最后遇到的最大值输出。
public SingleOutputStreamOperator<T> maxBy(String positionToMaxBy)	
public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy, boolean first)	
public SingleOutputStreamOperator<T> maxBy(String field, boolean first)	

数据流分发能力

表 10-13 数据流分发能力的相关接口

API	说明
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, int field)	使用一个用户自定义的Partitioner对每一个元素选择目标task。 • partitioner指的是用户自定义的分区类重写partition方法。 • field指的是partitioner的输入参数。 • keySelector指的是用户自定义的partitioner的输入参数。
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, String field)	
public <K> DataStream<T> partitionCustom(Partitioner<K> partitioner, KeySelector<T, K> keySelector)	
public DataStream<T> shuffle()	以均匀分布的形式将元素随机地进行分区。
public DataStream<T> rebalance()	基于round-robin对元素进行分区，使得每个分区负责均衡。对于存在数据倾斜的性能优化是很有用的。
public DataStream<T> rescale()	以round-robin的形式将元素分区到下游操作的子集中。
public DataStream<T> broadcast()	广播每个元素到所有分区。

提供 project 的能力

表 10-14 提供 project 的能力的相关接口

API	说明
public <R extends Tuple> SingleOutputStreamOperator<R> project(int... fieldIndexes)	从元组中选择了一部分字段子集。 fieldIndexes指的是需要选择的元组中的某几个序列。 **说明** 只支持Tuple数据类型的project投影。

提供设置 eventtime 属性的能力

表 10-15 提供设置 eventtime 属性的能力的相关接口

API	说明
public SingleOutputStreamOperator<T> assignTimestampsAndWatermarks(AssignerWithPeriodicWatermarks<T> timestampAndWatermarkAssigner)	为了能让event time窗口可以正常触发窗口计算操作，需要从记录中提取时间戳。
public SingleOutputStreamOperator<T> assignTimestampsAndWatermarks(AssignerWithPunctuatedWatermarks<T> timestampAndWatermarkAssigner)	

根据接口参数不同可以分为以上两种，AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks的区别如**表10-16**所示。

表 10-16 接口参数区别

参数	说明
AssignerWithPeriodicWatermarks	根据StreamExecutionEnvironment类的getConfig().setAutoWatermarkInterval(200L)时间戳生成Watermark。
AssignerWithPunctuatedWatermarks	每接收到一个元素，都会生成一个Watermark，而且可以根据接收到的元素生成不同的Watermark。

提供迭代的能力

表 10-17 提供迭代的能力的相关接口

API	说明
public IterativeStream<T> iterate()	在流(flow)中创建一个带反馈的循环，通过重定向一个operator的输出到之前的operator。 **说明** • 对于定义一些需要不断更新模型的算法是非常有帮助的。 • long maxWaitTimeMillis：该超时时间指的是每一轮迭代体执行的超时时间。
public IterativeStream<T> iterate(long maxWaitTimeMillis)	

提供分流能力

表 10-18 提供分流能力的相关接口

API	说明
public SplitStream<T> split(OutputSelector< T> outputSelector)	传入OutputSelector，重写select方法确定分流的依据(即打标记)，构建SplitStream流。即对每个元素做一个字符串的标记，作为选择的依据，打好标记之后就可以通过标记选出并新建某个标记的流。
public DataStream<OUT> select(String... outputNames)	从一个SplitStream中选出一个或多个流。 outputNames指的是使用split方法对每个元素做的字符串标记的序列。

窗口能力

窗口分为跳跃窗口和滑动窗口。

- 支持Window、TimeWindow、CountWindow以及WindowAll、TimeWindowAll、CountWindowAll API窗口生成。
- 支持Window Apply、Window Reduce、Window Fold、Aggregations on windows API窗口操作。
- 支持多种Window Assigner（TumblingEventTimeWindows、TumblingProcessingTimeWindows、SlidingEventTimeWindows、SlidingProcessingTimeWindows、EventTimeSessionWindows、ProcessingTimeSessionWindows、GlobalWindows）。
- 支持三种时间ProcessingTime、EventTime和IngestionTime。
- 支持两种EventTime时间戳方式：AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks。

窗口生成类API如**表10-19**所示。

表 10-19 窗口生成类的相关接口

API	说明
public <W extends Window> WindowedStream<T, KEY, W> window(WindowAssigner<? super T, W> assigner)	窗口可以被定义在已经被分区的KeyedStreams上。窗口会对数据的每一个key根据一些特征（例如在最近5秒钟内到达的数据）进行分组。
public <W extends Window> AllWindowedStream<T, W> windowAll(WindowAssigner<? super T, W> assigner)	窗口可以被定义在DataStream上。
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size)	时间窗口定义在已经被分区的KeyedStreams上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。 **说明** ● WindowedStream和AllWindowedStream代表不同的两种流。 ● 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size, Time slide)	
public AllWindowedStream<T, TimeWindow> timeWindowAll(Time size)	时间窗口定义在DataStream上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。
public AllWindowedStream<T, TimeWindow> timeWindowAll(Time size, Time slide)	
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size)	按照元素个数区分窗口，定义在已经被分区的KeyedStreams上。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。 **说明** ● WindowedStream和AllWindowedStream代表不同的两种流。 ● 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size, long slide)	

API	说明
public AllWindowedStream<T, GlobalWindow> countWindowAll(Time size)	按照元素个数区分窗口，定义在DataStream上。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。
public AllWindowedStream<T, GlobalWindow> countWindowAll(Time size, Time slide)	

窗口操作类API如表10-20所示。

表 10-20 窗口操作类的相关接口

方法	API	说明
Window	public <R> SingleOutputStreamOperator<R> apply(WindowFunction<T, R, K, W> function)	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。 ● function指的是执行的窗口函数。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> apply(WindowFunction<T, R, K, W> function, TypeInformation<R> resultType)	
	public SingleOutputStreamOperator<T> reduce(ReduceFunction<T> function)	应用一个reduce函数到窗口上，返回reduce后的值。 ● reduceFunction指的是执行的reduce函数。 ● WindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, WindowFunction<T, R, K, W> function)	

方法	API	说明
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, WindowFunction<T, R, K, W> function, TypeInformation<R> resultType)	
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function)	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function, TypeInformation<R> resultType)	
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, WindowFunction<ACC, R, K, W> function)	
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, WindowFunction<ACC, R, K, W> function, TypeInformation<ACC> foldAccumulatorType, TypeInformation<R> resultType)	
WindowAll	public <R> SingleOutputStreamOperator<R> apply(AllWindowFunction<T, R, W> function)	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。

方法	API	说明
	public <R> SingleOutputStreamOperator<R> apply(AllWindowFunction<T, R, W> function, TypeInformation<R> resultType)	
	public SingleOutputStreamOperator<T> reduce(ReduceFunction<T> function)	应用一个reduce函数到窗口上，返回reduce后的值。 ● reduceFunction指的是执行的reduce函数。 ● AllWindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, AllWindowFunction<T, R, W> function)	
	public <R> SingleOutputStreamOperator<R> reduce(ReduceFunction<T> reduceFunction, AllWindowFunction<T, R, W> function, TypeInformation<R> resultType)	
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function)	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。 ● resultType为返回的数据的类型信息。
	public <R> SingleOutputStreamOperator<R> fold(R initialValue, FoldFunction<T, R> function, TypeInformation<R> resultType)	
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, AllWindowFunction<ACC, R, W> function)	

方法	API	说明
	public <ACC, R> SingleOutputStreamOperator<R> fold(ACC initialValue, FoldFunction<T, ACC> foldFunction, AllWindowFunction<ACC, R, W> function, TypeInformation<ACC> foldAccumulatorType, TypeInformation<R> resultType)	
Window 和 Window All	public SingleOutputStreamOperator<T> sum(int positionToSum)	对窗口数据的某一列求和。 positionToSum和field代表数据的某一列。
	public SingleOutputStreamOperator<T> sum(String field)	
	public SingleOutputStreamOperator<T> min(int positionToMin)	对窗口数据的某一列求最小值。min返回了最小值，不保证非最小值列的准确性。 positionToMin和field代表对某一列求最小值。
	public SingleOutputStreamOperator<T> min(String field)	
	public SingleOutputStreamOperator<T> minBy(int positionToMinBy)	对窗口数据的某一列求最小值所在的该行数据，minBy返回了该行数据的所有元素。 ● positionToMinBy代表对哪一列做minBy操作。 ● first表示是否按最先遇到的最小值输出还是最后遇到的最小值输出。
	public SingleOutputStreamOperator<T> minBy(String positionToMinBy)	
	public SingleOutputStreamOperator<T> minBy(int positionToMinBy, boolean first)	
	public SingleOutputStreamOperator<T> minBy(String field, boolean first)	

方法	API	说明
	public SingleOutputStreamOperator<T> max(int positionToMax)	对窗口数据的某一列求最大值。max返回了最大值，不保证非最大值列的准确性。 positionToMax和field代表对某一列求最大值。
	public SingleOutputStreamOperator<T> max(String field)	
	public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy)//默认true	对窗口数据的某一列求最大值所在的该行数据，maxBy返回了在这个字段上是最大值的所有元素。 ● positionToMaxBy代表对哪一列做maxBy操作。 ● first表示是否按最先遇到的最大值输出还是最后遇到的最大值输出。
	public SingleOutputStreamOperator<T> maxBy(String positionToMaxBy)//默认true	
	public SingleOutputStreamOperator<T> maxBy(int positionToMaxBy, boolean first)	
	public SingleOutputStreamOperator<T> maxBy(String field, boolean first)	

提供多流合并的能力

表 10-21 提供多流合并的能力的相关接口

API	说明
public final DataStream<T> union(DataStream<T>... streams)	Union两个或多个数据流，生成一个新的包含了来自所有流的所有数据的数据流。 **说明** 如果你将一个数据流与其自身进行了合并，在结果流中对于每个元素你都会拿到两份。
public <R> ConnectedStreams<T, R> connect(DataStream<R> dataStream)	“连接”两个数据流并保持原先的类型。Connect可以让两条流之间共享状态。产生ConnectedStreams之后，调用map或者flatmap进行操作计算。

API	说明
public <R> SingleOutputStreamOperator<R> map(CoMapFunction<IN1, IN2, R> coMapper)	在ConnectedStreams上做元素映射，类似DataStream的map操作，元素映射之后流数据类型统一。
public <R> SingleOutputStreamOperator<R> flatMap(CoFlatMapFunction<IN1, IN2, R> coFlatMapper)	在ConnectedStreams上做元素映射，类似DataStream的flatMap操作，元素映射之后流数据类型统一。

提供 Join 能力

表 10-22 提供 Join 能力的相关接口

API	说明
public <T2> JoinedStreams<T, T2> join(DataStream<T2> otherStream)	通过给定的key在一个窗口范围内join两条数据流。
public <T2> CoGroupedStreams<T, T2> coGroup(DataStream<T2> otherStream)	通过给定的key在一个窗口范围内co-group两条数据流。

10.6.1.2 Flink Scala API 接口介绍

由于Flink开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

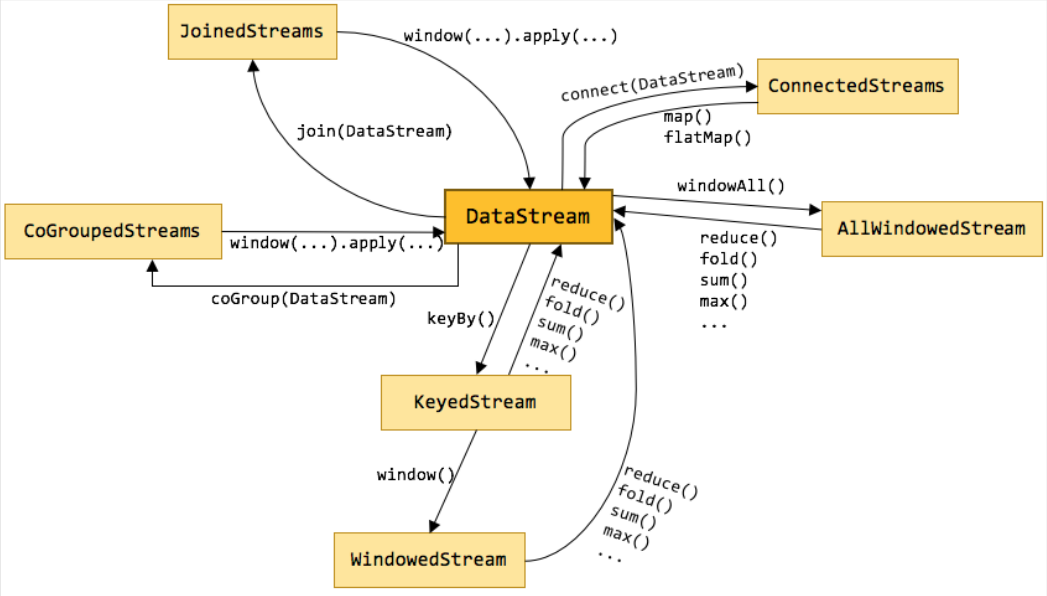
Flink 常用接口

Flink主要使用到如下这几个类：

- StreamExecutionEnvironment：是Flink流处理的基础，提供了程序的执行环境。
- DataStream：Flink用特别的类DataStream来表示程序中的流式数据。用户可以认为它们是含有重复数据的不可修改的集合(collection)，DataStream中元素的数量是无限的。
- KeyedStream：DataStream通过keyBy分组操作生成流，数据经过对设置的key值进行分组。
- WindowedStream：KeyedStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。

- AllWindowedStream：DataStream通过window窗口函数生成的流，设置窗口类型并且定义窗口触发条件，然后在窗口数据上进行一些操作。
- ConnectedStreams：将两条DataStream流连接起来并且保持原有流数据的类型，然后进行map或者flatMap操作。
- JoinedStreams：在窗口上对数据进行等值join操作，join操作是coGroup操作的一种特殊场景。
- CoGroupedStreams：在窗口上对数据进行coGroup操作，可以实现流的各种join类型。

图 10-59 Flink Stream 的各种流类型转换



流数据输入

表 10-23 流数据输入的相关接口

API	说明
def fromElements[T: TypeInformation](data: T*): DataStream[T]	获取用户定义的多个元素的数据，作为输入流数据。 data是多个元素的具体数据。
def fromCollection[T: TypeInformation](data: Seq[T]): DataStream[T]	获取用户定义的集合数据，作为输入流数据。 data可以是集合数据或者可迭代的数据体。
def fromCollection[T: TypeInformation](data: Iterator[T]): DataStream[T]	
def fromParallelCollection[T: TypeInformation] (data: SplittableIterator[T]): DataStream[T]	获取用户定义的集合数据，作为输入并行流数据。 data是可被分割成多个partition的迭代数据体。

API	说明
def generateSequence(from: Long, to: Long): DataStream[Long]	获取用户定义的一串序列数据，作为输入流数据。 ● from是指数值串的起点。 ● to是指数值串的终点。
def readTextFile(filePath: String): DataStream[String]	获取用户定义的某路径下的文本文件数据，作为输入流数据。 ● filePath是指文本文件的路径。 ● charsetName指的是编码格式的名字。
def readTextFile(filePath: String, charsetName: String): DataStream[String]	
def readFile[T: TypeInformation] (inputFormat: FileInputFormat[T], filePath: String)	获取用户定义的某路径下的文件数据，作为输入流数据。 ● filePath是指文件的路径。 ● inputFormat是指文件的格式。 ● watchType指的是文件的处理模式“PROCESS_ONCE”或者“PROCESS_CONTINUOUSLY”。 ● interval指的是多长时间判断目录或文件变化进行处理。
def readFile[T: TypeInformation] (inputFormat: FileInputFormat[T], filePath: String, watchType: FileProcessingMode, interval: Long): DataStream[T]	
def socketTextStream(hostname: String, port: Int, delimiter: Char = '\n', maxRetry: Long = 0): DataStream[String]	获取用户定义的Socket数据，作为输入流数据。 ● hostname是指Socket的服务器端的主机名称。 ● port指的是服务器的监测端口。 ● delimiter和maxRetry两个参数scala接口暂时不支持设置。
def addSource[T: TypeInformation] (function: SourceFunction[T]): DataStream[T]	用户自定义SourceFunction，addSource方法可以添加Kafka等数据源，主要实现方法为SourceFunction的run。 ● function指的是用户自定义的SourceFunction函数。 ● scala支持简化写法。
def addSource[T: TypeInformation] (function: SourceContext[T] => Unit): DataStream[T]	

数据输出

表 10-24 数据输出的相关接口

API	说明
def print(): DataStreamSink[T]	数据输出以标准输出流打印出来。
def printToErr()	数据输出以标准error输出流打印出来。

API	说明
def writeAsText(path: String): DataStreamSink[T]	数据输出写入到某个文本文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。
def writeAsText(path: String, writeMode: FileSystem.WriteMode): DataStreamSink[T]	
def writeAsCsv(path: String): DataStreamSink[T]	数据输出写入到某个csv格式的文件中。 ● path指的是文本文件的路径。 ● writeMode为文本文件写入模式“OVERWRITE”或者“NO_OVERWRITE”。 ● rowDelimiter为行分隔符。 ● fieldDelimiter为列分隔符。
def writeAsCsv(path: String, writeMode: FileSystem.WriteMode): DataStreamSink[T]	
def writeAsCsv(path: String, writeMode: FileSystem.WriteMode, rowDelimiter: String, fieldDelimiter: String): DataStreamSink[T]	
def writeUsingOutputFormat(format: OutputFormat[T]): DataStreamSink[T]	数据输出到普通文件中，例如二进制文件。
def writeToSocket(hostname: String, port: Integer, schema: SerializationSchema[T]): DataStreamSink[T]	数据输出写入到Socket连接中。 ● hostName为主机名称。 ● port为端口。
def addSink(sinkFunction: SinkFunction[T]): DataStreamSink[T]	用户自定义的数据输出，addSink方法通过flink-connectors支持数据输出到Kafka，主要实现方法为SinkFunction的invoke方法。
def addSink(fun: T => Unit): DataStreamSink[T]	

过滤和映射能力

表 10-25 过滤和映射能力的相关接口

API	说明
def map[R: TypeInformation](fun: T => R): DataStream[R]	输入一个元素，生成另一个元素，元素类型不变。
def map[R: TypeInformation](mapper: MapFunction[T, R]): DataStream[R]	
def flatMap[R: TypeInformation](flatMapper: FlatMapFunction[T, R]): DataStream[R]	输入一个元素，生成零个、一个或者多个元素。
def flatMap[R: TypeInformation](fun: (T, Collector[R]) => Unit): DataStream[R]	

API	说明
def flatMap[R: TypeInformation](fun: T => TraversableOnce[R]): DataStream[R]	
def filter(filter: FilterFunction[T]): DataStream[T]	对每个元素执行一个布尔函数，只保留返回true的元素。
def filter(fun: T => Boolean): DataStream[T]	

聚合能力

表 10-26 聚合能力的相关接口

API	说明
def keyBy(fields: Int*): KeyedStream[T, JavaTuple]	将流逻辑分区成不相交的分区，每个分区包含相同key的元素。内部是用hash分区来实现的。这个转换返回了一个KeyedStream。 KeyBy操作之后返回KeyedStream，然后再调用KeyedStream的函数（例如reduce/fold/min/minby/max/maxby/sum/sumby等）进行相应操作。 ● fields为数据某几列的序号。 ● firstField和otherFields为数据结构的成员变量的名称。 ● key为用户自定义的指定分区依据的方法。
def keyBy(firstField: String, otherFields: String*): KeyedStream[T, JavaTuple]	
def keyBy[K: TypeInformation](fun: T => K): KeyedStream[T, K]	
def reduce(fun: (T, T) => T): DataStream[T]	在一个KeyedStream上“滚动”reduce。合并当前元素与上一个被reduce的值，然后输出新的值。注意三者的类型是一致的。
def reduce(reducer: ReduceFunction[T]): DataStream[T]	
def fold[R: TypeInformation](initialValue: R)(fun: (R,T) => R): DataStream[R]	在一个KeyedStream上基于一个初始值“滚动”折叠。合并当前元素和上一个被折叠的值，然后输出新值。注意Fold的输入值与返回值类型可以不一致。
def fold[R: TypeInformation](initialValue: R, folder: FoldFunction[T,R]): DataStream[R]	
def sum(position: Int): DataStream[T]	在一个KeyedStream上滚动求和操作。 position和field代表对某一列求和。
def sum(field: String): DataStream[T]	

API	说明
def min(position: Int): DataStream[T]	在一个KeyedStream上滚动求最小值。min返回了最小值，不保证非最小值列的准确性。 position和field代表对某一列求最小值。
def min(field: String): DataStream[T]	
def max(position: Int): DataStream[T]	在一个KeyedStream上滚动求最大值。max返回了最大值，不保证非最大值列的准确性。 position和field代表对某一列求最大值。
def max(field: String): DataStream[T]	
def minBy(position: Int): DataStream[T]	在一个KeyedStream上求某一列最小值所在的该行数据，minBy返回了该行数据的所有元素。 position和field代表对某一列做minBy操作。
def minBy(field: String): DataStream[T]	
def maxBy(position: Int): DataStream[T]	在一个KeyedStream上求某一列最大值所在的该行数据，maxBy返回了该行数据的所有元素。 position和field代表对某一列做maxBy操作。
def maxBy(field: String): DataStream[T]	

数据流分发能力

表 10-27 数据流分发能力的相关接口

API	说明
def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], field: Int) : DataStream[T]	使用一个用户自定义的Partitioner对每一个元素选择目标task。 • partitioner指的是用户自定义的分区类重写partition方法。 • field指的是partitioner的输入参数。 • keySelector指的是用户自定义的partitioner的输入参数。
def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], field: String):DataStream[T]	
def partitionCustom[K: TypeInformation](partitioner: Partitioner[K], fun: T => K): DataStream[T]	
def shuffle: DataStream[T]	以均匀分布的形式将元素随机地进行分区。
def rebalance: DataStream[T]	基于round-robin对元素进行分区，使得每个分区负责均衡。对于存在数据倾斜的性能优化是很有用的。

API	说明
def rescale: DataStream[T]	以round-robin的形式将元素分区到下游操作的子集中。 **说明** 查看代码和rebalance的方式是一样的。
def broadcast: DataStream[T]	广播每个元素到所有分区。

提供设置 eventtime 属性的能力

表 10-28 提供设置 eventtime 属性的能力的相关接口

API	说明
def assignTimestampsAndWatermarks(assigner: AssignerWithPeriodicWatermarks[T]): DataStream[T]	为了能让event time窗口可以正常触发窗口计算操作，需要从记录中提取时间戳。
def assignTimestampsAndWatermarks(assigner: AssignerWithPunctuatedWatermarks[T]): DataStream[T]	

提供迭代的能力

表 10-29 提供迭代的能力的相关接口

API	说明
def iterate[R](stepFunction: DataStream[T] => (DataStream[T], DataStream[R]),maxWaitTimeMillis:Long = 0,keepPartitioning: Boolean = false) : DataStream[R]	在流(flow)中创建一个带反馈的循环，通过重定向一个operator的输出到之前的operator。 **说明** • 对于定义一些需要不断更新模型的算法是非常有帮助的。 • long maxWaitTimeMillis：该超时时间指的是每一轮迭代体执行的超时时间。

API	说明
def iterate[R, F: TypeInformation] (stepFunction: ConnectedStreams[T, F] => (DataStream[F], DataStream[R]),maxWaitTimeMillis:Long): DataStream[R]	

提供分流能力

表 10-30 提供分流能力的相关接口

API	说明
def split(selector: OutputSelector[T]): SplitStream[T]	传入OutputSelector，重写select方法确定分流的依据（即打标记），构建SplitStream流。即对每个元素做一个字符串的标记，作为选择的依据，打好标记之后就可以通过标记选出并新建某个标记的流。
def select(outputNames: String*): DataStream[T]	从一个SplitStream中选出一个或多个流。 outputNames指的是使用split方法对每个元素做的字符串标记的序列。

窗口能力

窗口分为跳跃窗口和滑动窗口。

- 支持Window、TimeWindow、CountWindow以及WindowAll、TimeWindowAll、CountWindowAll API窗口生成。
- 支持Window Apply、Window Reduce、Window Fold、Aggregations on windows API窗口操作。
- 支持多种Window Assigner（TumblingEventTimeWindows、TumblingProcessingTimeWindows、SlidingEventTimeWindows、SlidingProcessingTimeWindows、EventTimeSessionWindows、ProcessingTimeSessionWindows、GlobalWindows）。
- 支持三种时间ProcessingTime、EventTime和IngestionTime。
- 支持两种EventTime时间戳方式：AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks。

窗口生成类API如表10-31所示。

表 10-31 窗口生成类的相关接口

API	说明
def window[W <: Window](assigner: WindowAssigner[_ >: T, W]): WindowedStream[T, K, W]	窗口可以被定义在已经被分区的KeyedStreams上。窗口会对数据的每一个key根据一些特征（例如在最近5秒钟内到达的数据）进行分组。
def windowAll[W <: Window](assigner: WindowAssigner[_ >: T, W]): AllWindowedStream[T, W]	窗口可以被定义在DataStream上。
def timeWindow(size: Time): WindowedStream[T, K, TimeWindow]	时间窗口定义在已经被分区的KeyedStreams上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。 **说明** ● WindowedStream和AllWindowedStream代表不同的两种流。 ● 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
def countWindow(size: Long, slide: Long): WindowedStream[T, K, GlobalWindow]	
def timeWindowAll(size: Time): AllWindowedStream[T, TimeWindow]	时间窗口定义在DataStream上，根据environment.getStreamTimeCharacteristic()参数选择是ProcessingTime还是EventTime，根据参数个数确定是TumblingWindow还是SlidingWindow。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。
def timeWindowAll(size: Time, slide: Time): AllWindowedStream[T, TimeWindow]	
def countWindow(size: Long, slide: Long): WindowedStream[T, K, GlobalWindow]	按照元素个数区分窗口，定义在已经被分区的KeyedStreams上。 ● size指的是窗口时间的大小。 ● slide指的是窗口的滑动时间。 **说明** ● WindowedStream和AllWindowedStream代表不同的两种流。 ● 接口中只有一个参数则是TumblingWindow；有两个或两个以上的参数则是SlidingWindow。
def countWindow(size: Long): WindowedStream[T, K, GlobalWindow]	

API	说明
def countWindowAll(size: Long, slide: Long): AllWindowedStream[T, GlobalWindow]	按照元素个数区分窗口，定义在DataStream上。 • size指的是窗口时间的大小。 • slide指的是窗口的滑动时间。
def countWindowAll(size: Long): AllWindowedStream[T, GlobalWindow]	

窗口操作类API如表10-32所示。

表 10-32 窗口操作类的相关接口

方法	API	说明
Window	def apply[R: TypeInformation] (function: WindowFunction[T, R, K, W]): DataStream[R]	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。 function指的是执行的窗口函数。
	def apply[R: TypeInformation] (function: (K, W, Iterable[T], Collector[R]) => Unit): DataStream[R]	
	def reduce(function: ReduceFunction[T]): DataStream[T]	应用一个reduce函数到窗口上，返回reduce后的值。 • reduceFunction指的是执行的reduce函数。 • WindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。
	def reduce(function: (T, T) => T): DataStream[T]	
	def reduce[R: TypeInformation] (preAggregator: ReduceFunction[T], function: WindowFunction[T, R, K, W]): DataStream[R]	
	def reduce[R: TypeInformation] (preAggregator: (T, T) => T, windowFunction: (K, W, Iterable[T], Collector[R]) => Unit): DataStream[R]	

方法	API	说明
	def fold[R: TypeInformation](initialValue: R, function: FoldFunction[T,R]): DataStream[R]	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。
	def fold[R: TypeInformation](initialValue: R)(function: (R, T) => R): DataStream[R]	
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, foldFunction: FoldFunction[T, ACC], function: WindowFunction[ACC, R, K, W]): DataStream[R]	
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, foldFunction: (ACC, T) => ACC, windowFunction: (K, W, Iterable[ACC], Collector[R]) => Unit): DataStream[R]	
WindowAll	def apply[R: TypeInformation](function: AllWindowFunction[T, R, W]): DataStream[R]	应用一个一般的函数到窗口上，窗口中的数据会作为一个整体被计算。
	def apply[R: TypeInformation](function: (W, Iterable[T], Collector[R]) => Unit): DataStream[R]	
	def reduce(function: ReduceFunction[T]): DataStream[T]	应用一个reduce函数到窗口上，返回reduce后的值。 ● reduceFunction指的是执行的reduce函数。 ● AllWindowFunction的function指的是在经过reduce操作之后再触发一次窗口操作。
	def reduce(function: (T, T) => T): DataStream[T]	
	def reduce[R: TypeInformation](preAggregator: ReduceFunction[T], windowFunction: AllWindowFunction[T, R, W]): DataStream[R]	

方法	API	说明
	def reduce[R: TypeInformation] (preAggregator: (T, T) => T, windowFunction: (W, Iterable[T], Collector[R]) => Unit): DataStream[R]	
	def fold[R: TypeInformation] (initialValue: R, function: FoldFunction[T,R]): DataStream[R]	应用一个fold函数到窗口上，然后返回折叠后的值。 ● initialValue指的是初始值。 ● foldFunction指的是折叠函数。 ● WindowFunction的function指的是在经过fold操作之后再触发一次窗口操作。
	def fold[R: TypeInformation] (initialValue: R)(function: (R, T) => R): DataStream[R]	
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, preAggregator: FoldFunction[T, ACC], windowFunction: AllWindowFunction[ACC, R, W]): DataStream[R]	
	def fold[ACC: TypeInformation, R: TypeInformation](initialValue: ACC, preAggregator: (ACC, T) => ACC, windowFunction: (W, Iterable[ACC], Collector[R]) => Unit): DataStream[R]	
Window 和 Window All	def sum(position: Int): DataStream[T]	对窗口数据的某一列求和。 position和field代表数据的某一列。
	def sum(field: String): DataStream[T]	
	def min(position: Int): DataStream[T]	对窗口数据的某一列求最小值。min返回了最小值，不保证非最小值列的准确性。 position和field代表对某一列求最小值。
	def min(field: String): DataStream[T]	
	def max(position: Int): DataStream[T]	对窗口数据的某一列求最大值。max返回了最大值，不保证非最大值列的准确性。 position和field代表对某一列求最大值。
	def max(field: String): DataStream[T]	

方法	API	说明
	def minBy(position: Int): DataStream[T]	对窗口数据的某一列求最小值所在的该行数据，minBy返回了该行数据的所有元素。 position和field代表对某一列做minBy操作。
	def minBy(field: String): DataStream[T]	
	def maxBy(position: Int): DataStream[T]	对窗口数据的某一列求最大值所在的该行数据，maxBy返回了该行数据的所有元素。 position和field代表对某一列做maxBy操作。
	def maxBy(field: String): DataStream[T]	

提供多流合并的能力

表 10-33 提供多流合并的能力的相关接口

API	说明
def union(dataStreams: DataStream[T]*): DataStream[T]	Union两个或多个数据流，生成一个新的包含了来自所有流的所有数据的数据流。 **说明** 如果你将一个数据流与其自身进行了合并，在结果流中对于每个元素你都会拿到两份。
def connect[T2](dataStream: DataStream[T2]): ConnectedStreams[T, T2]	“连接”两个数据流并保持原先的类型。Connect可以让两条流之间共享状态。产生ConnectedStreams之后，调用map或者flatmap进行操作计算。
def map[R: TypeInformation](coMapper: CoMapFunction[IN1, IN2, R]): DataStream[R]	在ConnectedStreams上做元素映射，类似DataStream的map操作，元素映射之后流数据类型统一。
def map[R: TypeInformation](fun1: IN1 => R, fun2: IN2 => R): DataStream[R]	
def flatMap[R: TypeInformation](coFlatMapper: CoFlatMapFunction[IN1, IN2, R]): DataStream[R]	在ConnectedStreams上做元素映射，类似DataStream的flatMap操作，元素映射之后流数据类型统一。

API	说明
def flatMap[R: TypeInformation](fun1: (IN1, Collector[R]) => Unit, fun2: (IN2, Collector[R]) => Unit): DataStream[R]	
def flatMap[R: TypeInformation](fun1·IN1 => TraversableOnce[R], fun2: IN2 => TraversableOnce[R]): DataStream[R]	

提供 Join 能力

表 10-34 提供 Join 能力的相关接口

API	说明
def join[T2](otherStream: DataStream[T2]): JoinedStreams[T, T2]	通过给定的key在一个窗口范围内join两条数据流。 join操作的key值通过where和eaualTo方法进行指定，代表两条流过滤出包含等值条件的数据。
def coGroup[T2](otherStream: DataStream[T2]): CoGroupedStreams[T, T2]	通过给定的key在一个窗口范围内co-group两条数据流。 coGroup操作的key值通过where和eaualTo方法进行指定，代表两条流通过该等值条件进行分区处理。

10.6.1.3 Flink REST API 接口介绍

Flink具有可用于查询正在运行的作业的状态和统计信息以及最近完成作业的监视API。该监视API由Flink自己的WEB UI使用。

监视API是REST API，可接受HTTP GET请求并使用JSON数据进行响应。REST API是访问Web服务器的一套API。当前在Flink中，Web服务器是JobManager的一个模块，和JobManager共进程。默认情况下，web服务器监测的端口是8081，用户可以在配置文件“flink-conf.yaml”中配置“jobmanager.web.port”来修改监测端口。

使用Netty和Netty路由器库来处理REST请求和解析URL。

REST API接口的执行方式是通过HTTP请求进行。

HTTP请求的格式为：http://<JobManager_IP>:<JobManager_Port><Path>，

其中JobManager_IP是指JobManager进程所在服务器节点的IP地址，JobManager_Port是指JobManager进程的监测端口，Path为路径的部分，参见**表10-35**。例如：http://10.162.181.57:32261/config。

说明

需要修改Flink Client的配置文件“flink-conf.yaml”，在“jobmanager.web.allow-access-address”和“jobmanager.web.access-control-allow-origin”中添加访问主机的IP地址，可使用逗号分隔。

Flink支持的所有REST API的URL中的Path信息如**表10-35**所示。

表 10-35 Path 介绍

Path	说明
/config	有关监控API和服务器设置的一些信息。
/logout	注销的重要信息。
/overview	Flink集群状态的简单概要。
/jobs	Job的ID，按运行，完成，失败和取消等状态进行分组。
/jobmanager/config	JobManager的配置。
/joboverview	业务按状态进行分组，每个业务组都有一个小状态。
/joboverview/running	与“/ joboverview”相同，Job按状态进行分组，每个Job组都有一个小状态，但只包含当前运行的Job。
/joboverview/completed	Job按状态进行分组，每个都有一个小状态的摘要。 与“/ joboverview”相同，但仅包含已完成，已取消或失败的Job。
/jobs/<jobid>	一个Job主要信息包含列出数据流计划，状态，状态转换的时间戳，每个顶点（运算符）的聚合信息。
/jobs/<jobid>/vertices	目前与“/jobs/<jobid>”相同。
/jobs/<jobid>/config	Job使用用户定义的执行配置。
/jobs/<jobid>/exceptions	Job探察到不可恢复的异常。截取的标识提示是否存在更多异常，但不列出这些异常，否则回复会太大。
/jobs/<jobid>/accumulators	聚合用户累加器加上Job累加器。
/jobs/<jobid>/checkpoints	Job的checkpoint的统计信息。
/jobs/<jobid>/metrics	一个Job的所有可用指标。
/jobs/<jobid>/vertices/<vertexid>	关于流图的顶点下每个子任务的信息。

Path	说明
/jobs/<jobid>/vertices/<vertexid>/subtasktimes	请求返回流图的顶点的所有子任务状态转换的时间戳。 这些可以用于在子任务之间创建时间线的比较。
/jobs/<jobid>/vertices/<vertexid>/taskmanagers	一个流图顶点的TaskManager统计信息。 这是“/ jobs / <jobid> / vertices / <vertexid>”返回的子任务统计信息的聚合。
/jobs/<jobid>/vertices/<vertexid>/accumulators	聚合的用户定义的累加器，用于流图顶点。
/jobs/<jobid>/vertices/<vertexid>/checkpoints	单个Job顶点的检查点统计信息。
/jobs/<jobid>/vertices/<vertexid>/backpressure	单个Job顶点的背压统计数据及其所有子任务。
/jobs/<jobid>/vertices/<vertexid>/metrics	一组指标值的给定任务。
/jobs/<jobid>/vertices/<vertexid>/subtasks/accumulators	获取流图顶点的所有子任务的所有用户定义的累加器。 这些是通过请求“/ jobs / <jobid> / vertices / <vertexid> / accumulators”以聚合形式返回的各个累加器。
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>	特定子任务的当前或最近执行尝试的摘要。
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>/attempts/<attempt>	特定子任务的具体执行尝试的摘要。 发生故障/恢复时会发生多次执行尝试。
/jobs/<jobid>/vertices/<vertexid>/subtasks/<subtasknum>/attempts/<attempt>/accumulators	在一次特定的执行尝试期间，为一个特定子任务收集累加器（在发生故障/恢复时会发生多次尝试）。
/jobs/<jobid>/plan	Job的数据流计划。 该计划也包括在Job摘要（“/ jobs / <jobid>”）中。
/taskmanagers	任务管理员信息。
/taskmanagers/<taskmanagerid>/metrics	任务管理员的度量信息。
/taskmanagers/<taskmanagerid>/log	任务管理员的日志信息。
/taskmanagers/<taskmanagerid>/stdout	一个任务管理员的标准。
/jobmanager/log	JobManager的日志信息。
/jobmanager/stdout	JobManager的标准。
/jobmanager/metrics	JobManager的指标。

Path	说明
/:*	对Web前端的静态文件（如HTML，CSS或JS文件）的请求。

表10-35中变量的介绍请参见表10-36。

表 10-36 变量说明

变量	说明
jobid	job的id。
vertexid	流图的顶点id。
subtasknum	子任务的总和。
attempt	尝试。
taskmanagerid	任务管理的id。

10.6.1.4 Flink Savepoints CLI 介绍

概述

Savepoints在持久化存储中保存某个checkpoint，以便用户可以暂停自己的应用进行升级，并将状态设置为savepoint的状态，并继续运行。该机制利用了Flink的checkpoint机制创建流应用的快照，并将快照的元数据（meta-data）写入到一个额外的持久化文件系统中。

如果需要使用savepoints的功能，强烈推荐用户为每个算子通过uid(String)分配一个固定的ID，以便将来升级恢复使用，示例代码如下：

```
DataStream<String> stream = env
// Stateful source (e.g. Kafka) with ID
.addSource(new StatefulSource())
.uid("source-id") // ID for the source operator
.shuffle()
 // Stateful mapper with ID
.map(new StateFulMapper())
.uid("mapper-id") // ID for the mapper
// Stateless printing sink
.print(); //Auto-generated ID
```

savepoint 恢复

如果用户不手动设置ID，系统将自动给每个算子分配一个ID。只要该算子的ID不改变，即可从savepoint恢复，ID的产生取决于用户的应用代码，并且对应用代码的结构十分敏感。因此，强烈推荐用户手动为每个算子设置ID。Savepoint产生的数据将被保存到配置的文件系统中，如FsStateBackend或者RocksDBStateBackend。

1. 触发一个savepoint
   ```
   $ bin/flink savepoint <jobId> [targetDirectory]
   ```
   以上命令将触发ID为jobId的作业产生一个savepoint，另外，用户可以通过targetDirectory指定savepoint的存储路径，该路径必须是jobManager可以访问

的，由于targetDirectory是可选的，如果用户没有配置targetDirectory，则是使用配置文件中“state.savepoints.dir”配置的目录来存放savepoint。

用户可以在“flink-conf.yaml”中通过“state.savepoints.dir”选项设置默认的savepoint路径。

```
# Default savepoint target directory
```

说明

建议用户将targetDirectory路径设置为HDFS路径 ，例如：

```
bin/flink savepoint  405af8c02cf6dc069a0f9b7a1f7be088 hdfs://savepoint
```

2. 删除一个作业并进行savepoint

```
$ bin/flink cancel -s [targetDirectory] jobId
```

以上命令将删除一个作业，同时，在删除前将对该作业的状态进行保存。另外，用户可以通过targetDirectory指定savepoint的存储路径，该路径必须是jobManager可以访问的。

3. 恢复作业方式
   - 从savepoint恢复作业。

     ```
     $ bin/flink run -s savepointPath [runArgs]
     ```

     以上命令将提交一个作业，并将该作业的初始状态置为savepointPath指定的状态。

     说明

     runArgs是指用户应用中自定义的参数，每个用户自定义的参数形式、名称都不一样。

   - 允许不恢复某个算子的状态

     ```
     $ bin/flink run -s savepointPath -n [runArgs]
     ```

     默认情况下，系统将尝试将savepoint的状态全部映射到用户的流应用中，如果用户升级的流应用删除了某个算子，可以通过--allowNonRestoredState(简写-n)恢复状态。

4. 清除savepoints

```
$ bin/flink savepoint -d savepointPath
```

以上命令将删除保存在savepointPath的savepoint。

注意事项

- 如果一个task中有算子链（Chained operators），将会将算子链上第一个算子的ID分配给该task。给算子链上的中间算子手动分配ID是不可能的。例如：在链（Chain）[a->b->c]中，只能给a手动分配ID，b和c不能分配。如果用户想给b和c分配ID，用户必须手动建链。手动建链时需要使用disableChaining()接口。举例如下：

```
env.addSource(new GetDataSource())
.keyBy(0)
.timeWindow(Time.seconds(2)).uid("window-id")
.reduce(_+_).uid("reduce-id")
.map(f=>(f,1)).disableChaining().uid("map-id")
.print().disableChaining().uid("print-id")
```

- 用户升级job时不允许更改算子的数据类型。

10.6.1.5 Flink Client CLI 介绍

常用 CLI

Flink常用的CLI如下所示：

1. yarn-session.sh
   - 可以使用yarn-session.sh启动一个常驻的Flink集群，接受来自客户端提交的任务。启动一个有3个TaskManager实例的Flink集群示例如下：
     ```
     bin/yarn-session.sh
     ```
   - yarn-session.sh的其他参数可以通过以下命令获取：
     ```
     bin/yarn-session.sh -help
     ```
2. Flink
   - 使用flink命令可以提交Flink作业，作业既可以被提交到一个常驻的Flink集群上，也可以使用单机模式运行。
     - 提交到常驻Flink集群上的一个示例如下：
       ```
       bin/flink run ../examples/streaming/WindowJoin.jar
       ```

       说明

       用户在用该命令提交任务前需要先用yarn-session启动Flink集群。

     - 以yarn-cluster模式运行作业的一个示例如下：
       ```
       bin/flink run -m yarn-cluster ../examples/streaming/WindowJoin.jar
       ```

       说明

       通过参数-m yarn-cluster使作业以yarn-cluster模式运行，该模式为指定作业单独启动一个Flink集群来执行。

     - 列出所有的作业（包含JobID）：
       ```
       bin/flink list
       ```
     - 取消作业：
       ```
       bin/flink cancel <JobID>
       ```
     - 停止作业（仅流式作业）：
       ```
       bin/flink stop <JobID>
       ```

       说明

       取消和停止作业的区别如下：

       - 取消作业：执行“cancel”命令时，指定作业会立即收到cancel()方法调用ASAP。如果调用结束后作业仍然没有停止，Flink会定期开始中断执行线程直至作业停止。
       - 停止作业：“stop”命令仅适用于Flink源（source）实现了StoppableFunction接口的作业。“stop”命令会等待所有资源都正确关闭。相比“cancel”命令，“stop”停止作业的方式更为优雅，但可能导致停止作业失败。
   - flink脚本的其他参数可以通过以下命令获取：
     ```
     bin/flink --help
     ```

注意事项

- 如果yarn-session.sh使用-z配置特定的zookeeper的namespace，则在使用flink run时必须使用-yid指出applicationID，使用-yz指出zookeeper的namespace，前后namespace保持一致。

  举例：

  ```
  bin/yarn-session.sh -z YARN101
  bin/flink run -yid application_****_**** -yz YARN101 examples/streaming/WindowJoin.jar
  ```

- 如果yarn-session.sh不使用-z配置特定的zookeeper的namespace，则在使用flink run时不要使用-yz指定特定的zookeeper的namespace。

举例：

```
bin/yarn-session.sh
bin/flink run examples/streaming/WindowJoin.jar
```

- 如果使用flink run -m yarn-cluster时启动集群则可以使用-yz指定一个zookeeper的namespace。
- 不能同时启动两个或两个以上的集群来共享一个namespace。
- 用户在启动集群或提交作业时如果使用了-z配置项，则在删除、停止及查询作业、触发savepoint时也要使用-z配置项指明namespace。

10.6.2 如何处理用户在使用 chrome 浏览器时无法显示任务状态的 title

问题

用户在使用chrome浏览器浏览Flink Web UI页面时无法显示title。此处以Tasks为例进行分析，用户将鼠标置于Tasks的彩色小方框上，无法显示彩色小框的title说明，如图10-60所示。正常的显示界面如图10-61所示。

图 10-60 界面无法显示 title

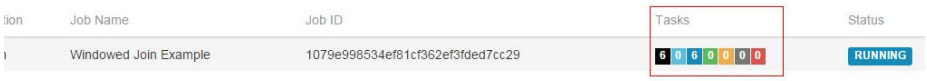
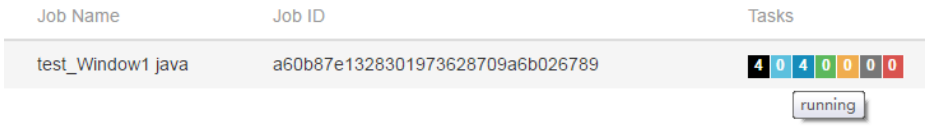


图 10-61 界面正常显示 title



回答

如果用户遇到chrome浏览器无法显示title，步骤如下：

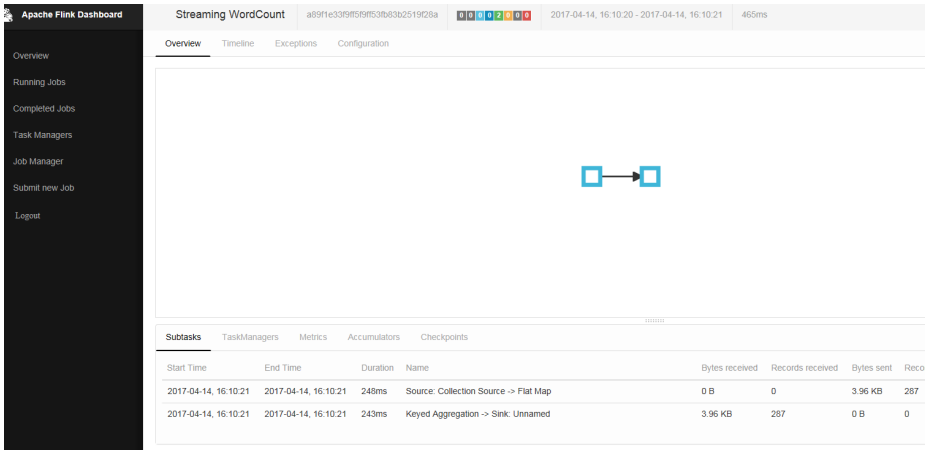
请检查本机是否同时运行会影响chrome浏览器冒泡提示的工具，如果运行了，需要将该工具关闭。

10.6.3 如何处理 IE10/11 页面算子的文字部分显示异常

问题

如何处理IE10/11页面显示异常，每个算子的文字部分没有显示出来的问题？如图10-62所示，Overview显示为空白。

图 10-62 页面显示异常



回答

Flink中用了foreignObject元素来代理绘制svg矢量图，但是IE 10/11不支持foreignObject导致算子显示异常。支持使用chrome浏览器。

10.6.4 如何处理 Checkpoint 设置 RocksDBStateBackend 方式时 Checkpoint 慢

问题

如何处理checkpoint设置RocksDBStateBackend方式，且当数据量大时，执行checkpoint会很慢的问题?

原因分析

由于窗口使用自定义窗口，这时窗口的状态使用ListState，且同一个key值下，value的值非常多，每次新的value值到来都要使用RocksDB的merge()操作；触发计算时需要将该key值下所有的value值读出。

- RocksDB的方式为merge()->merge()....->merge()->read()，该方式读取数据时非常耗时，如图10-63所示。
- source算子在瞬间发送了大量数据，所有数据的key值均相等，导致window算子处理速度过慢，使barrier在缓存中积压，快照的制作时间过长，导致window算子在规定时间内没有向CheckpointCoordinator报告快照制作完成，CheckpointCoordinator认为快照制作失败，如图10-64所示。

图 10-63 时间监控信息

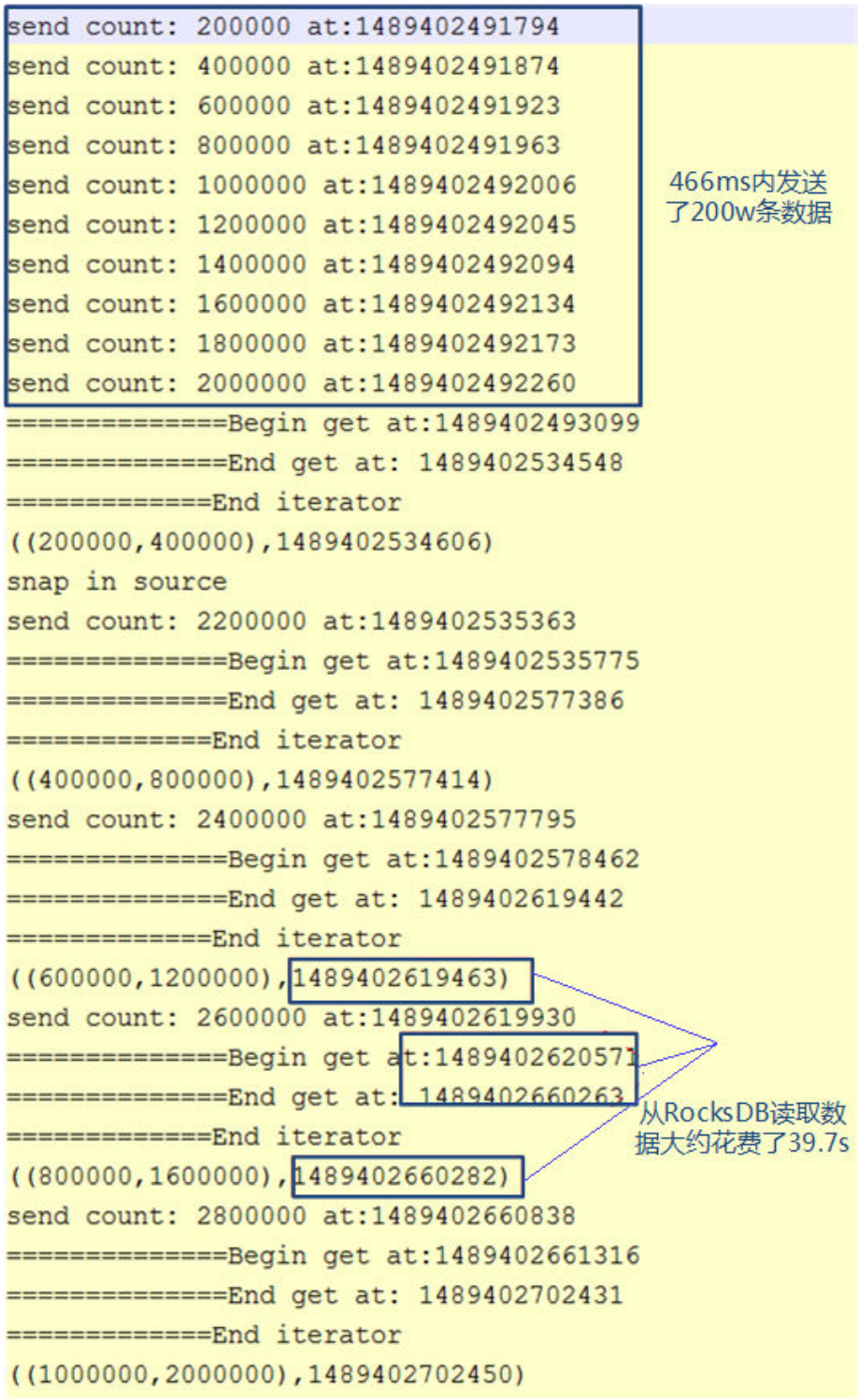
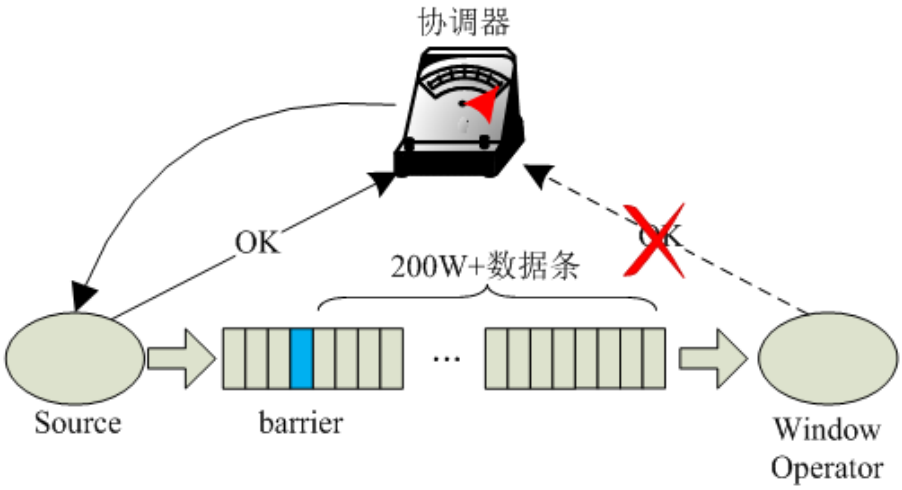


图 10-64 关系图



回答

Flink引入了第三方软件包RocksDB的缺陷问题导致该现象的发生。建议用户将checkpoint设置为FsStateBackend方式。

用户需要在应用代码中将checkpoint设置为FsStateBackend。例如：

```
env.setStateBackend(new FsStateBackend("hdfs://hacluster/flink/checkpoint/"));
```

10.6.5 如何处理 blob.storage.directory 配置/home 目录时启动 yarn-session 失败

问题

当用户设置“blob.storage.directory”为“/home”时，用户没有权限在“/home”下创建“blobStore-UUID”的文件，导致yarn-session启动失败。

回答

步骤1 建议将"blob.storage.directory"配置选项设置成“/tmp”或者“/opt/huawei/Bigdata/tmp”。

步骤2 当用户将"blob.storage.directory"配置选项设置成自定义目录时，需要手动赋予用户该目录的owner权限。以下以FusionInsight的admin用户为例。

1. 修改Flink客户端配置文件conf/flink-conf.yaml，配置blob.storage.directory: /home/testdir/testdirdir/xxx。
2. 创建目录/home/testdir（创建一层目录即可），设置该目录为admin用户所属。

图 10-65 创建目录

```
SZV1000064084:/home # id admin
uid=20000(admin) gid=9998(ficommon) groups=9998(ficommon),8003(System_administrator_186)
SZV1000064084:/home # chown admin:ficommon testdir/ -R
```

说明

/home/testdir/下的testdirdir/xxx目录在启动Flink集群时会在每个节点下自动创建。

3. 进入客户端路径，执行命令**./bin/yarn-session.sh -jm 2048 -tm 3072**，可以看到yarn-session正常启动并且成功创建目录。

图 10-66 执行命令

----结束

10.6.6 如何处理非 static 的 KafkaPartitioner 类对象构造 FlinkKafkaProducer010 运行时报错

问题

Flink内核升级到1.3.0之后，当Kafka调用带有非static的KafkaPartitioner类对象为参数的FlinkKafkaProducer010去构造函数时，运行时会报错。

报错内容如下：

```
org.apache.flink.api.common.InvalidProgramException: The implementation of the FlinkKafkaPartitioner is not serializable. The object probably contains or references non serializable fields.
```

回答

Flink的1.3.0版本，为了兼容原有那些使用KafkaPartitioner的API接口，如FlinkKafkaProducer010带KafkaPartitioner对象的构造函数，增加了FlinkKafkaDelegatePartitioner类。

该类定义了一个成员变量，即kafkaPartitioner：

```
private final KafkaPartitioner<T> kafkaPartitioner;
```

当Flink传入参数是KafkaPartitioner去构造FlinkKafkaProducer010时，调用栈如下：

```
FlinkKafkaProducer010(String topicId, KeyedSerializationSchema<T> serializationSchema, Properties producerConfig, KafkaPartitioner<T> customPartitioner)
-> FlinkKafkaProducer09(String topicId, KeyedSerializationSchema<IN> serializationSchema, Properties producerConfig, FlinkKafkaPartitioner<IN> customPartitioner)
----> FlinkKafkaProducerBase(String defaultTopicId, KeyedSerializationSchema<IN> serializationSchema, Properties producerConfig, FlinkKafkaPartitioner<IN> customPartitioner)
------> ClosureCleaner::clean(Object func, boolean checkSerializable)
```

首先使用KafkaPartitioner对象去构造一个FlinkKafkaDelegatePartitioner对象，然后再检查该对象是否可序列化。由于ClosureCleaner::clean函数是static函数，当用例中的KafkaPartitioner对象是非static时，ClosureCleaner::clean函数无法访问KafkaDelegatePartitioner类内的非static成员变量kafkaPartitioner，导致报错。

解决方法如下，两者任选其一：

- 将KafkaPartitioner类改成static类。

- 改用以FlinkKafkaPartitioner为参数的FlinkKafkaProducer010构造函数，内部实现不会去构造FlinkKafkaDelegatePartitioner，也就不会存在成员变量的问题。

10.6.7 如何处理新创建的 Flink 用户提交任务报 ZooKeeper 文件目录权限不足

问题

创建一个新的Flink用户，提交任务，ZooKeeper目录无权限导致提交Flink任务失败，日志中报如下错误：

```
NoAuth for /flink_base/flink/application_1499222480199_0013
```

回答

1. 首先查看ZooKeeper中/flink_base的目录权限是否为：'world,'anyone: cdrwa；如果不是，请修改/flink_base的目录权限为：'world,'anyone: cdrwa，然后继续根据**步骤二**排查；如果是，请根据**步骤二**排查。
2. 由于在Flink配置文件中“high-availability.zookeeper.client.acl”默认为“creator”，即谁创建谁有权限，由于原有用户已经使用ZooKeeper上的/flink_base/flink目录，导致新创建的用户访问不了ZooKeeper上的/flink_base/flink目录。

   新用户可以通过以下操作来解决问题。

   a. 查看客户端的配置文件“conf/flink-conf.yaml”。
   b. 修改配置项“high-availability.zookeeper.path.root”对应的ZooKeeper目录，例如：/flink2。
   c. 重新提交任务。

10.6.8 如何处理无法直接通过 URL 访问 Flink Web

问题

无法通过“http://JobManager IP:JobManager的端口”访问Web页面。

回答

由于浏览器所在的计算机IP地址未加到Web访问白名单导致。用户可以通过修改客户端的配置文件“conf/flink-conf.yaml”来解决问题。

1. 确认配置项“jobmanager.web.ssl.enabled”的值是否是“false”，若不是，请修改为“false”。
2. 确认配置项“jobmanager.web.access-control-allow-origin”和“jobmanager.web.allow-access-address”中是否已经添加浏览器所在的计算机IP地址。如果没有添加，可以通过这两项配置项进行添加。例如：

   ```
   jobmanager.web.access-control-allow-origin: 浏览器所在的计算机IP地址
   jobmanager.web.allow-access-address: 浏览器所在的计算机IP地址
   ```

10.6.9 如何查看 System.out.println 打印的调试信息或将调试信息输出至指定文件

问题

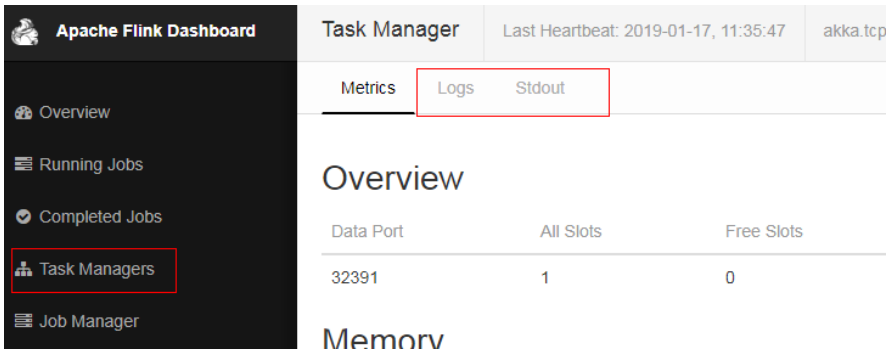
Flink业务代码中添加了System.out.println调试信息打印，该如何查看此调试日志？避免与运行日志混在一起打印，如何将业务日志单独输出至指定文件？

回答

Flink所有的运行日志打印都会打印至Yarn的本地目录下，默认所有Log都会输出至Yarn container本地目录下taskmanager.log，所有调用System.out打印都会输出值taskmanager.out文件。查看方式如下：

1. 进入Flink原生Web页面。
2. 在左侧导航栏单击“Task Managers”，可在“Logs”或“Stdout”页签查看日志信息。

图 10-67 查看日志信息



配置业务日志与TaskManager运行日志独立打印：

说明

若配置业务日志与TaskManager运行日志分开打印后，业务日志不输出至taskmanager.log，无法使用Web页面进行查看相应日志信息。

1. 修改客户端的配置文件“conf/logback.xml”，在文件中添加如下日志配置信息，加粗标注部分根据需要进行修改。

```
<appender name="TEST" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>/path/test.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
        <fileNamePattern>/path/test.log.%i</fileNamePattern>
        <minIndex>1</minIndex>
        <maxIndex>20</maxIndex>
    </rollingPolicy>
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
        <maxFileSize>20MB</maxFileSize>
    </triggeringPolicy>
    <encoder>
        <pattern>%d{"yyyy-MM-dd HH:mm:ss,SSS"} | %m %n</pattern>
    </encoder>
</appender>

<logger name="com.huawei.bigdata.flink.examples" additivity="false">
    <level value="INFO"/>
```

```
    <appender-ref ref="TEST"/>
  </logger>
```

2. 重新启动yarn-session.sh，提交任务。

说明

自定义日志若指定了路径**<file>/path/test.log</file>**，需确保任务运行所使用的用户（**flink-conf.yaml**配置用户）有权限对该目录进行读写操作。

10.6.10 如何处理 Flink 任务配置 State Backend 为 RocksDB 时报错 GLIBC 版本问题

问题

Flink任务配置State Backend为RocksDB时，运行报如下错误：

```
Caused by: java.lang.UnsatisfiedLinkError: /srv/BigData/hadoop/data1/nm/usercache/***/appcache/
application_****/rocksdb-lib-****/librocksdbjni-linux64.so: /lib64/libpthread.so.0: version `GLIBC_2.12` not
found (required by /srv/BigData/hadoop/***/librocksdbjni-linux64.so)
at java.lang.ClassLoader$NativeLibrary.load(Native Method)
at java.lang.ClassLoader.loadLibrary0(ClassLoader.java:1965)
at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1890)
at java.lang.Runtime.load0(Runtime.java:795)
at java.lang.System.load(System.java:1062)
at org.rocksdb.NativeLibraryLoader.loadLibraryFromJar(NativeLibraryLoader.java:78)
at org.rocksdb.NativeLibraryLoader.loadLibrary(NativeLibraryLoader.java:56)
at
org.apache.flink.contrib.streaming.state.RocksDBStateBackend.ensureRocksDBIsLoaded(RocksDBStateBacken
d.java:734)
... 11 more
```

可能原因

运行的系统和编译环境所在的系统版本不同，造成GLIBC的版本不兼容。

定位思路

使用**strings /lib64/libpthread.so.0 | grep GLIBC**命令查看GLIBC是否版本低于2.12。

处理步骤

如果GLIBC版本太低，则需要使用含有较高版本的（此处为2.12）的文件替换掉"libpthread-*.so"（注意，这是一个链接文件，执行时只需要替换掉它所指向的文件即可）。

参考信息

无

11 HBase 开发指南（安全模式）

11.1 HBase 应用开发概述

11.1.1 HBase 应用开发简介

HBase 简介

HBase是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统。HBase设计目标是解决关系型数据库在处理海量数据时的局限性。

HBase使用场景有如下几个特点：

- 处理海量数据（TB或PB级别以上）。
- 具有高吞吐量。
- 在海量数据中实现高效的随机读取。
- 具有很好的伸缩能力。
- 能够同时处理结构化和非结构化的数据。
- 不需要完全拥有传统关系型数据库所具备的ACID特性。ACID特性指原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。
- HBase中的表具有如下特点：
  - 大：一个表可以有上亿行，上百万列。
  - 面向列：面向列（族）的存储和权限控制，列（族）独立检索。
  - 稀疏：对于为空（null）的列，并不占用存储空间，因此，表可以设计的非常稀疏。

接口类型简介

由于HBase本身是由java语言开发出来的，且java语言具有简洁通用易懂的特性，推荐用户使用java语言进行HBase应用程序开发。

HBase采用的接口与Apache HBase保持一致。

HBase通过接口调用，可提供的功能如**表11-1**所示。

表 11-1 HBase 接口提供的功能

功能	说明
CRUD数据读写功能	增查改删
高级特性	过滤器、二级索引，协处理器
管理功能	表管理、集群管理

11.1.2 HBase 应用开发常用概念

- **过滤器**

  过滤器提供了非常强大的特性来帮助用户提高HBase处理表中数据的效率。用户不仅可以使用HBase中预定义好的过滤器，而且可以实现自定义的过滤器。
- **协处理器**

  允许用户执行region级的操作，并且可以使用与RDBMS中触发器类似的功能。
- **keytab文件**

  存放用户信息的密钥文件。在安全模式下，应用程序采用此密钥文件进行API方式认证。
- **Client**

  客户端直接面向用户，可通过Java API、HBase Shell或者Web UI访问服务端，对HBase的表进行读写操作。本文中的HBase客户端特指HBase client的安装包，可参考**HBase对外接口介绍**。

11.1.3 HBase 应用开发流程介绍

本文档主要基于Java API对HBase进行应用开发。

开发流程中各阶段的说明如**图11-1**和**表11-2**所示。

图 11-1 HBase 应用程序开发流程

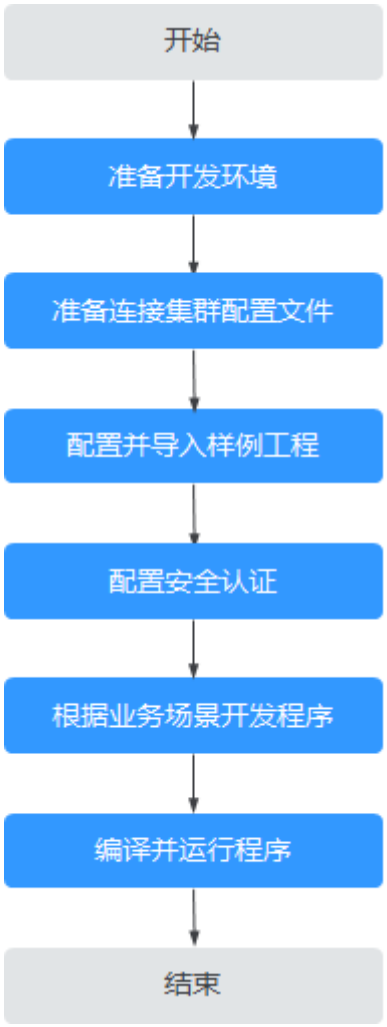


表 11-2 HBase 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，推荐使用Java语言进行开发，使用IntelliJ IDEA工具，同时完成JDK、Maven等初始配置。	准备本地应用开发环境

阶段	说明	参考文档
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备连接HBase集群配置文件**
配置并导入样例工程	HBase提供了不同场景下的多种样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。	**导入并配置HBase样例工程**
配置安全认证	如果您使用的是开启了Kerberos认证的MRS集群，需要进行安全认证。	**配置HBase应用安全认证**
根据业务场景开发程序	根据实际业务场景开发程序，调用组件接口实现对应功能。	**开发HBase应用**
编译并运行程序	将开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。	**调测HBase应用**

11.1.4 HBase 应用开发样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下HBase相关样例工程：

表 11-3 HBase 相关样例工程

样例工程位置	描述
hbase-examples/hbase-example	HBase数据读写操作的应用开发示例。 通过调用HBase接口可实现创建用户表、导入用户数据、增加用户信息、查询用户信息及为用户表创建二级索引等功能，相关业务场景介绍请参见**HBase样例程序开发思路**。
hbase-examples/hbase-rest-example	HBase Rest接口应用开发示例。 使用Rest接口实现查询HBase集群信息、获取表、操作NameSpace、操作表等功能，相关样例介绍请参见**HBase Rest接口调用样例程序**。
hbase-examples/hbase-thrift-example	访问HBase ThriftServer应用开发示例。 访问ThriftServer操作表、向表中写数据、从表中读数据，相关样例介绍请参见**访问HBase ThriftServer连接样例程序**。
hbase-examples/hbase-zk-example	HBase访问ZooKeeper应用开发示例。 在同一个客户端进程内同时访问MRS ZooKeeper和第三方的ZooKeeper，其中HBase客户端访问MRS ZooKeeper，客户应用访问第三方ZooKeeper。

11.2 准备 HBase 应用开发环境

11.2.1 准备本地应用开发环境

在进行二次开发时，要准备的开发和运行环境如**表11-4**所示。

表 11-4 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。

准备项	说明
安装JDK	开发和运行环境的基本配置，版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的： ● X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 ● TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装和配置IntelliJ IDEA	用于开发HBase应用程序的工具，版本要求：2019.1或其他兼容版本。 **说明** ● 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 ● 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 ● 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 ● 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Junit插件	开发环境的基本配置。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。 华为提供开源镜像站，各服务样例工程依赖的Jar包通过华为开源镜像站下载，剩余所依赖的开源Jar包请直接从Maven中央库或者其他用户自定义的仓库地址下载，详情请参考**配置华为开源镜像仓**。
7-zip	用于解压“*.zip”和“*.rar”文件。 支持7-Zip 16.04版本。

11.2.2 准备连接 HBase 集群配置文件

准备集群认证用户信息

对于开启Kerberos认证的MRS集群，需提前准备具有相关组件操作权限的用户用于程序认证。

以下HBase权限配置示例供参考，在实际业务场景中可根据业务需求灵活调整。

步骤1 登录FusionInsight Manager。

步骤2 选择“集群 > 服务 > HBase > 更多 > 启用Ranger鉴权”，查看该参数是否置灰。

- 是，创建用户并在Ranger中赋予该用户相关操作权限：
  a. 选择“系统 > 权限 > 用户 > 添加用户”，在新增用户界面创建一个机机用户，例如**developuser**。
     “用户组”需加入“hadoop”用户组。
  b. 使用Ranger管理员用户**rangeradmin**登录Ranger管理页面。
     rangeradmin用户默认密码为“Rangeradmin@123”，详细内容请参见**用户账号一览表**。
  c. 在首页中单击“HBASE”区域的组件插件名称如“HBase”。
  d. 单击“Policy Name”名称为“all - table, column-family, column”操作列的 。
  e. 在“Allow Conditions”区域新增策略允许条件，“Select User”列勾选**步骤2.a**新建的用户名称，“Permissions”列勾选“Select/Deselect All”。
  f. 单击“Save”。
- 否，创建用户并在Manager赋予用户相关操作权限：
  a. 选择“系统 > 权限 > 角色 > 添加角色”。
     i. 填写角色的名称，例如**developrole**。
     ii. 在“配置资源权限”的表格中选择“*待操作集群的名称* > HBase > HBase Scope ”，勾选“global”的“管理”、“创建”、“读”、“写”和“执行”，单击“确定”保存。
  b. 选择“用户 > 添加用户”，在新增用户界面，创建一个机机用户，例如**developuser**。
     - “用户组”需加入“hadoop”用户组。
     - “角色”加入**步骤2.a**新增的角色。

步骤3 使用**admin**用户登录FusionInsight Manager，选择“系统 > 权限 > 用户”，在用户名为**developuser**的操作列选择“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到该用户的“user.keytab”文件与“krb5.conf”文件。

----结束

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。
  a. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**），单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。
     例如，客户端配置文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到

“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。

b. 进入客户端配置文件解压路径的“HBase\config”，获取**表11-5**中相关配置文件。

表 11-5 配置文件

配置文件	作用
core-site.xml	配置Hadoop Core详细参数。
hbase-site.xml	配置HBase详细参数。
hdfs-site.xml	配置HDFS详细参数。

说明

访问HBase ThriftServer应用开发示例工程所需的配置文件还需参考**准备ThriftServer实例配置文件**获取。

c. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群，具体操作请参考**配置Windows通过EIP访问安全模式集群HBase**。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。

  a. 在节点中安装MRS集群客户端。

  例如客户端安装目录为“/opt/client”。

  b. 获取配置文件：

  i. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**），勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。

  ii. 以**root**登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“HBase/config”路径下的**表11-5**中相关配置文件。

  例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

  cd /tmp/FusionInsight-Client

  tar -xvf FusionInsight_Cluster_1_Services_Client.tar

  tar -xvf FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar

  cd FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles

说明

访问HBase ThriftServer应用开发示例工程所需的配置文件还需参考**准备ThriftServer实例配置文件**获取。

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

准备 ThriftServer 实例配置文件

若需实现访问HBase ThriftServer并进行表相关操作，则需执行以下步骤获取相关配置文件。

步骤1 登录FusionInsight Manager，选择“集群 > 服务 > HBase > 配置 > 全部配置”，搜索并修改ThriftServer实例的配置参数“hbase.thrift.security.qop”。该参数值需与“hbase.rpc.protection”的值一一对应。保存配置，重启配置过期节点服务使更改的配置生效。

说明

“hbase.rpc.protection”与“hbase.thrift.security.qop”参数值的对应关系为：

- **"privacy" - "auth-conf"**
- **"authentication" - "auth"**
- **"integrity" - "auth-int"**

步骤2 获取ThriftServer实例配置文件：

- 方法一：选择“集群 > 服务 > HBase > 实例”，单击待操作的ThriftServer实例，在“概览”界面的“配置文件”区域分别单击配置文件“hdfs-site.xml”、“core-site.xml”、“hbase-site.xml”名称，获取配置文件。
- 方法二：通过**准备运行环境配置文件**中解压客户端文件的方法获取配置文件，需要在获取的“hbase-site.xml”中手动添加以下配置，其中“hbase.thrift.security.qop”的参数值与**步骤1**保持一致。

```
<property>
<name>hbase.thrift.security.qop</name>
<value>auth</value>
</property>
<property>
<name>hbase.thrift.kerberos.principal</name>
<value>thrift/hadoop.hadoop.com@HADOOP.COM</value>
</property>
<property>
<name>hbase.thrift.keytab.file</name><value>/opt/huawei/Bigdata/FusionInsight_HD_8.1.0.1/install/FusionInsight-HBase-2.2.3/keytabs/HBase/thrift.keytab</value>
```

----结束

11.2.3 导入并配置 HBase 样例工程

背景信息

获取HBase开发样例工程，将工程导入到IntelliJ IDEA开始样例学习。

前提条件

- 确保本地PC的时间与MRS集群的时间差要小于5分钟，若无法确定，请联系系统管理员。MRS集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备连接HBase集群配置文件**。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程，可根据实际业务场景选择对应的样例，相关样例介绍请参见**HBase应用开发样例工程介绍**。

步骤2 若需要在本地Windows调测HBase样例代码，需参考**表11-6**放置各样例项目所需的配置文件、认证文件：

表 11-6 放置各样例项目所需的配置文件/认证文件

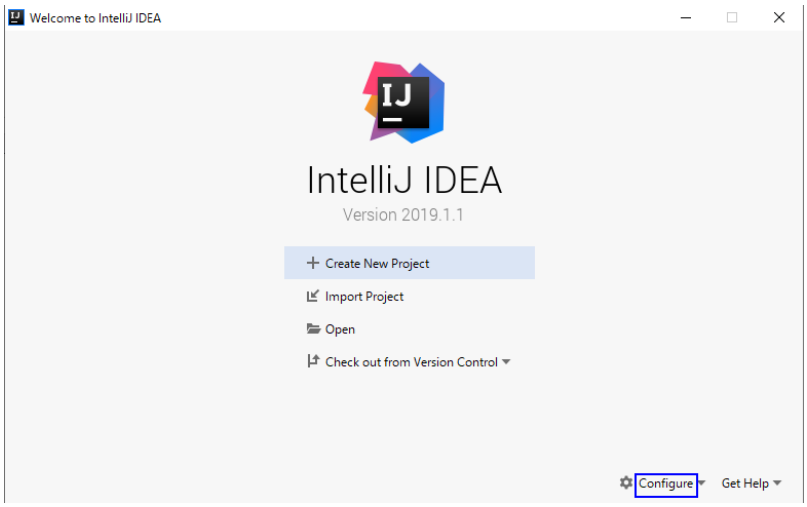
样例工程位置	需放置的配置/认证文件
hbase-examples/hbase-example（单集群场景）	需将以下文件放置在样例工程的“../src/main/resources/conf”目录下： ● 配置文件为**准备运行环境配置文件**获取的“core-site.xml”、“hbase-site.xml”和“hdfs-site.xml”。 ● 认证文件为**准备集群认证用户信息**获取的keytab认证文件“user.keytab”和“krb5.conf”。
hbase-examples/hbase-example（多集群互信场景）	将互信场景下的同名用户其中一个集群的认证凭据及其配置文件放入“../src/main/resources/hadoopDomain”目录下，将另一集群的配置文件放入“../src/main/resources/hadoop1Domain”目录下。其中： ● 配置文件为**准备运行环境配置文件**获取的“core-site.xml”、“hbase-site.xml”和“hdfs-site.xml”。 ● 认证文件为**准备集群认证用户信息**获取的“user.keytab”和“krb5.conf”。
hbase-examples/hbase-rest-example	需将在**准备集群认证用户信息**时获取的keytab认证文件“user.keytab”和“krb5.conf”放置到“../src/main/resources/conf”（若不存在conf目录，请自行创建）中。

样例工程位置	需放置的配置/认证文件
hbase-examples/hbase-thrift-example	需将以下文件放置在样例工程的“../src/main/resources/conf”目录下： • 配置文件为**准备ThriftServer实例配置文件**获取的“hdfs-site.xml”、“core-site.xml”、“hbase-site.xml”。 • 认证文件为**准备集群认证用户信息**获取的keytab认证文件“user.keytab”和“krb5.conf”。
hbase-examples/hbase-zk-example	需将以下文件放置在样例工程的“../src/main/resources”目录下： • 配置文件为**准备运行环境配置文件**获取的“core-site.xml”、“hbase-site.xml”和“hdfs-site.xml”。 • 认证文件为**准备集群认证用户信息**获取的keytab认证文件“user.keytab”、“krb5.conf”。 • 还需确保该目录下已存在**HBase访问多ZooKeeper场景安全认证**所需的“zoo.cfg”和“jaas.conf”文件。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

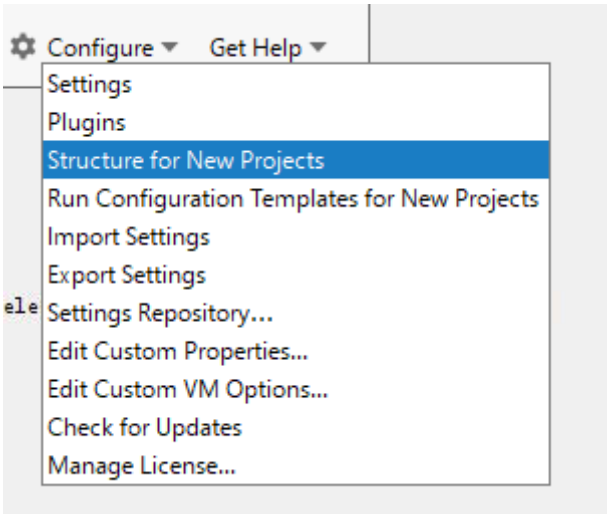
1. 打开IntelliJ IDEA，选择“Configure”。

图 11-2 Quick Start



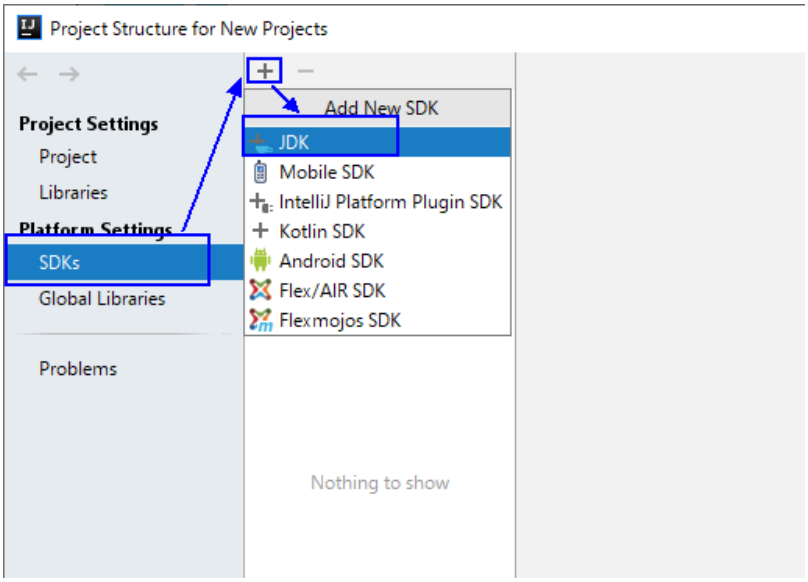
2. 在下拉框中选择“Structure for New Projects”。

图 11-3 Configure



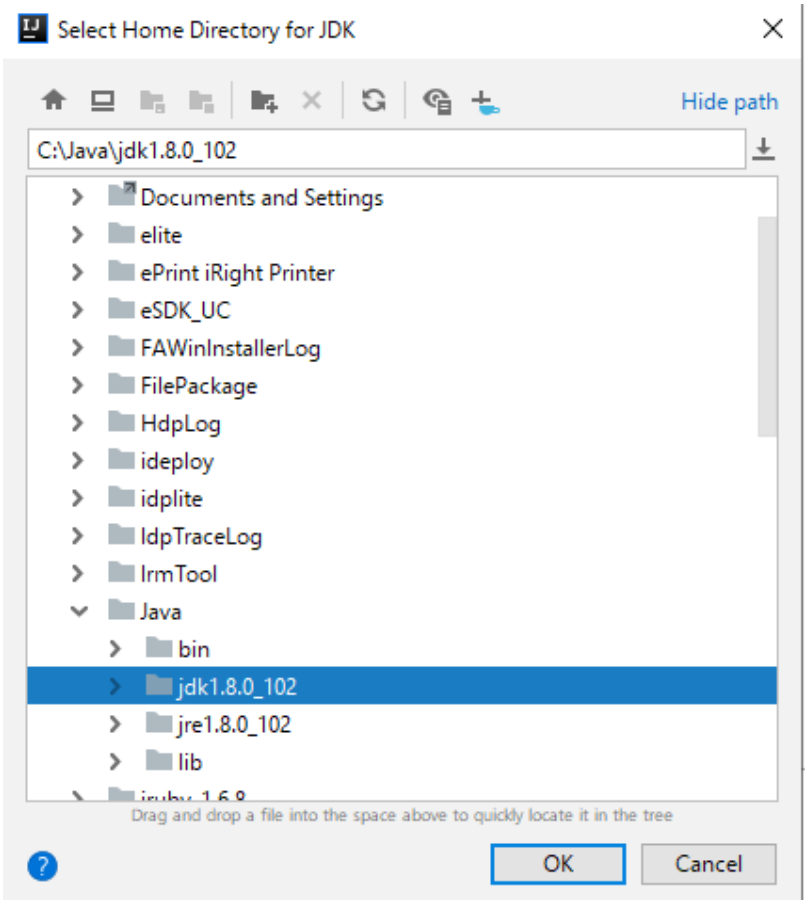
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 11-4 Project Structure for New Projects



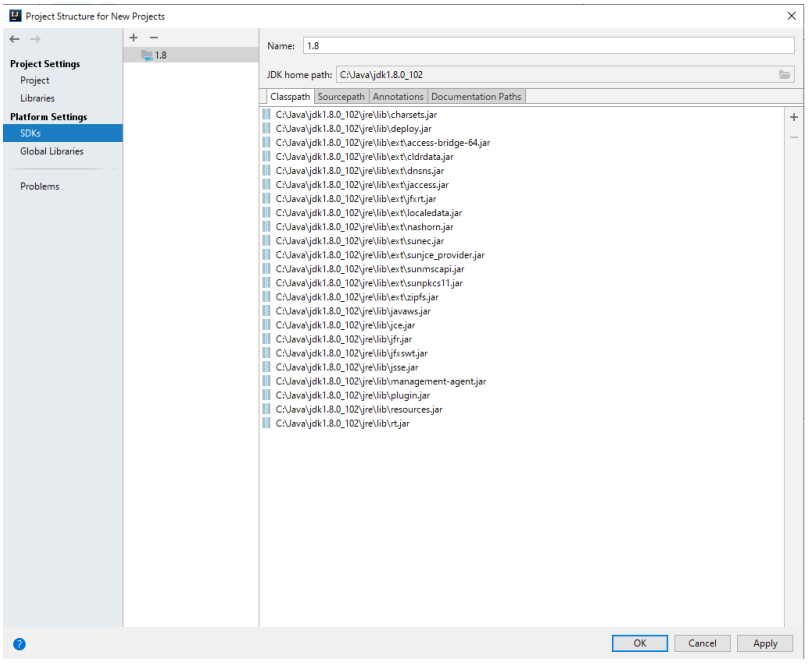
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 11-5 Select Home Directory for JDK



5. 完成JDK选择后，单击“OK”完成配置。

图 11-6 完成 JDK 配置



说明

不同的IDEA版本的操作步骤可能存在差异，以实际版本的界面操作为准。

步骤4 导入样例工程到IntelliJ IDEA开发环境。

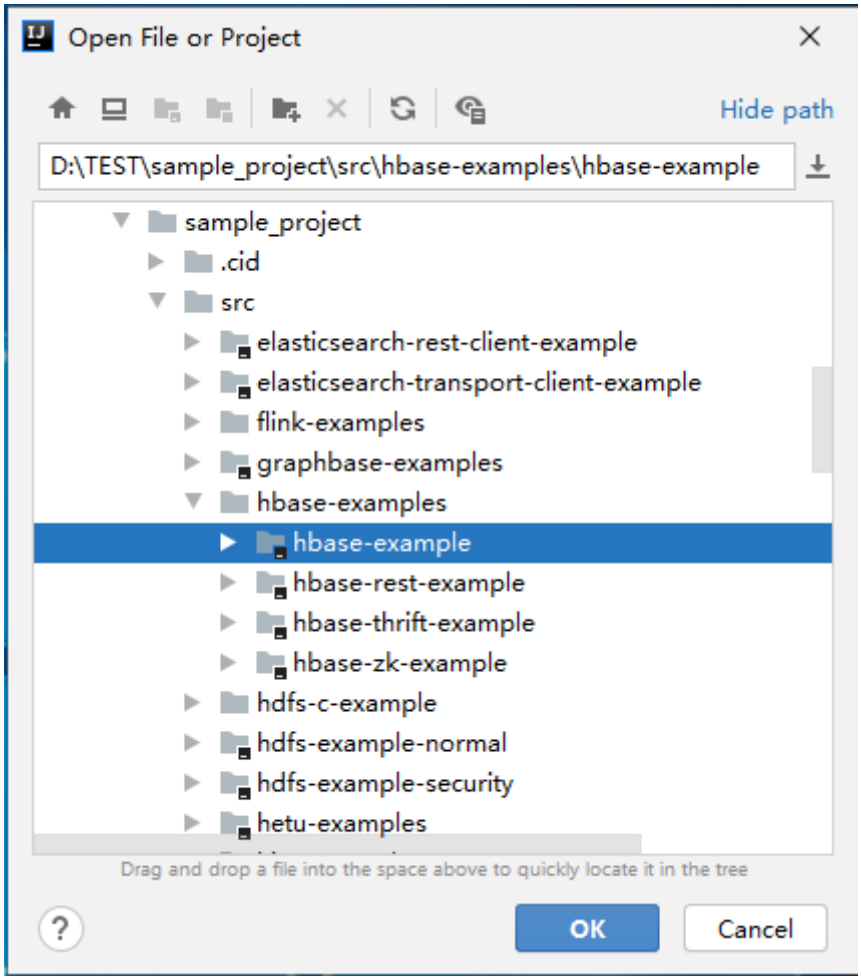
1. 打开IntelliJ IDEA，在“Quick Start”页面选择“Open or Import”。
   另外，针对已使用过的IDEA工具，可以从主界面选择“File > Import project...”导入样例工程。

   图 11-7 Open or Import（Quick Start 页面）

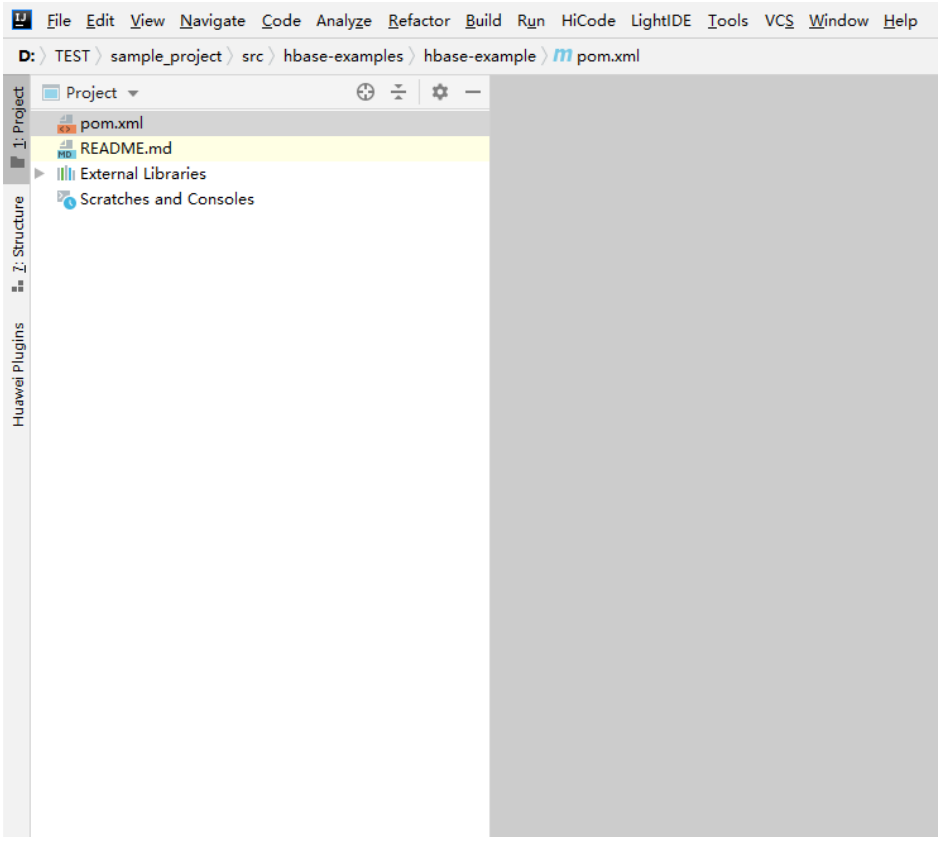
2. 选择样例工程文件夹“hbase-example”，然后单击“OK”。

图 11-8 Select File or Directory to Import



3. 导入结束，IDEA主页显示导入的样例工程。

图 11-9 导入样例工程成功



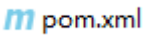
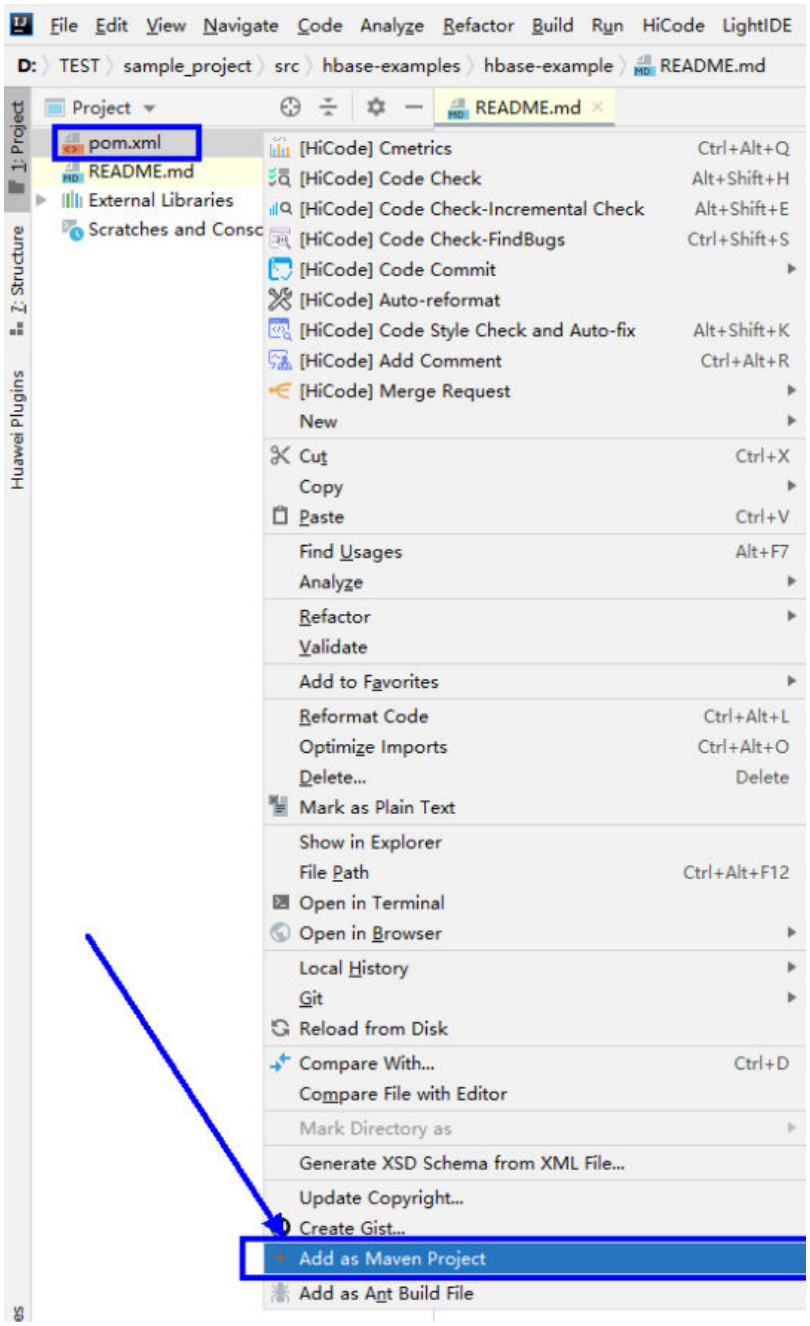
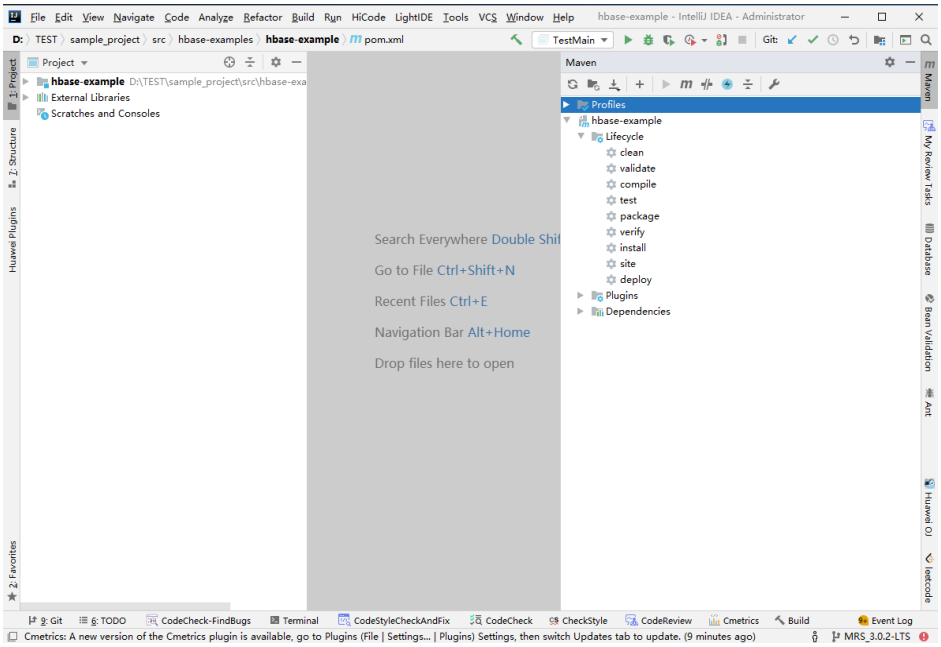
4. 右键单击“pom.xml”，选择“Add as Maven Project”，将该项目添加为Maven Project。若"pom.xml"图标如  所示，可直接进行下一步骤操作。

图 11-10 Add as Maven Project



此时IDEA可将该项目识别为Maven项目。

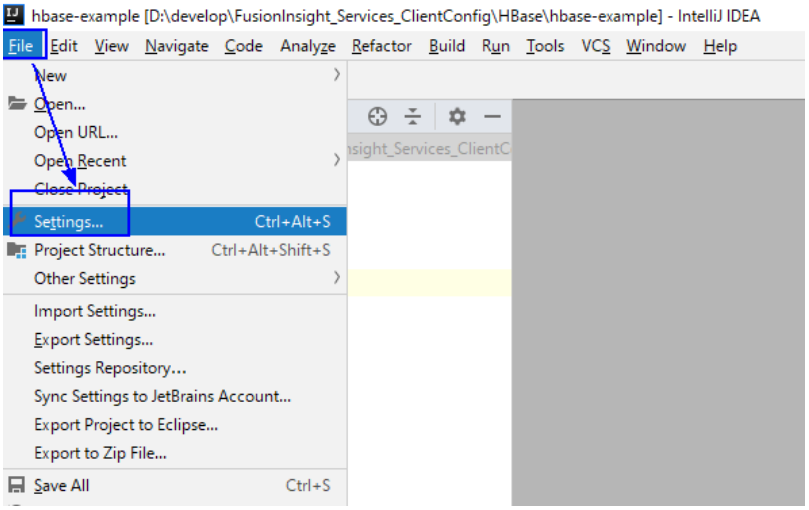
图 11-11 样例项目作为 maven 项目在 IDEA 中显示



步骤5 设置项目使用的Maven版本。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings...”。

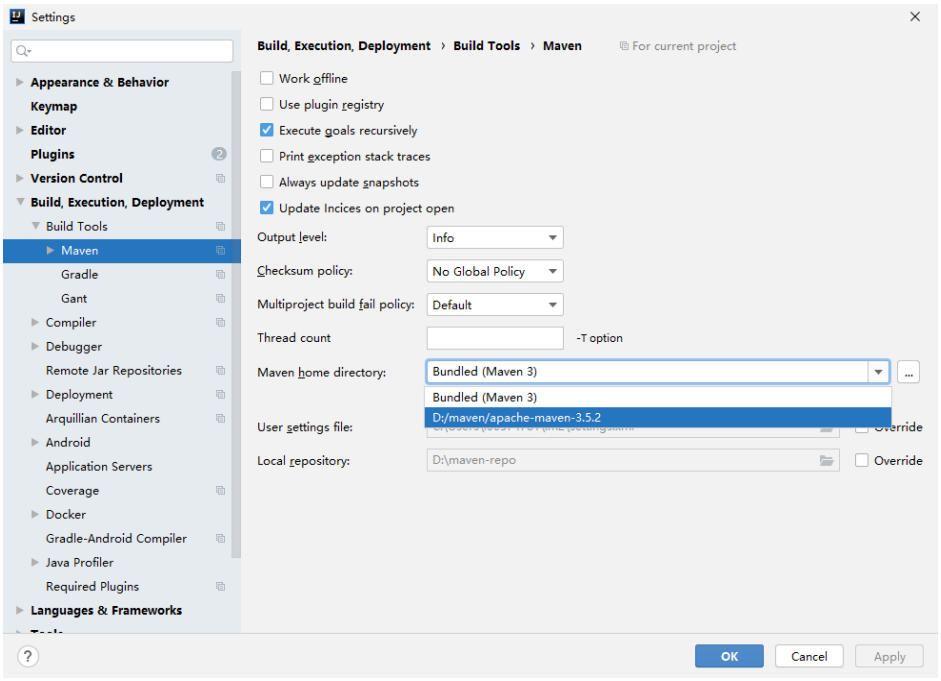
图 11-12 Settings



2. 选择“Build,Execution,Deployment > Maven”，选择“Maven home directory”为本地安装的Maven版本。

   然后根据实际情况设置好“User settings file”和“Local repository”参数，依次单击“Apply > OK”。

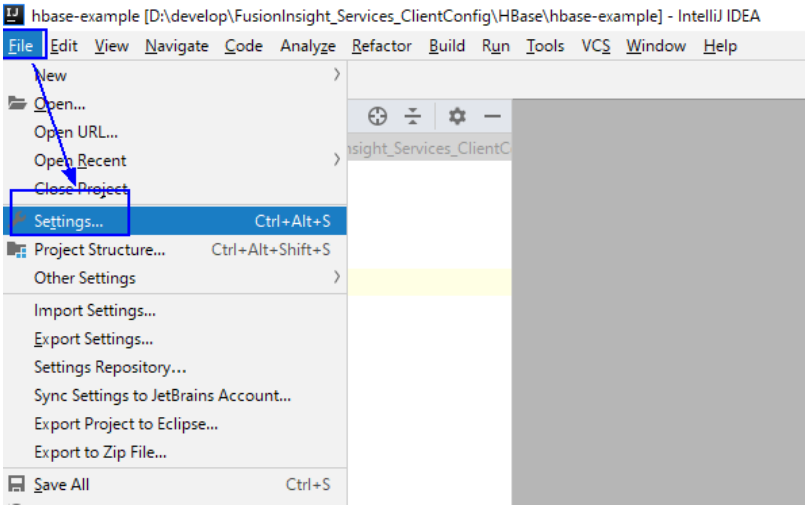
图 11-13 选择本地 Maven 安装目录



步骤6 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

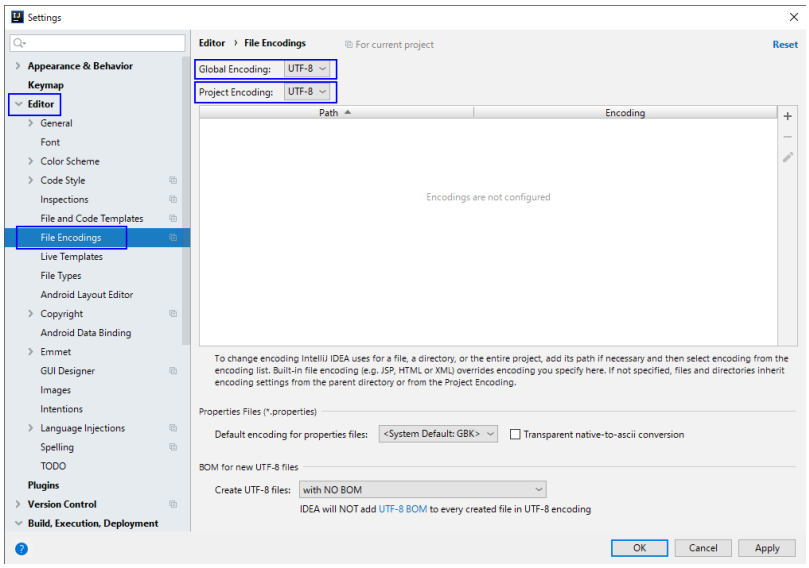
1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings...”。

图 11-14 Settings



2. 在弹出的“Settings”页面左边导航上选择“Editor > File Encodings”，分别在右侧的“Global Encoding”和“Project Encoding”的下拉框中，选择“UTF-8”。

图 11-15 File Encodings



3. 然后单击“Apply”和“OK”，完成编码配置。

----结束

11.2.4 配置 HBase 应用安全认证

11.2.4.1 HBase 数据读写示例安全认证（单集群场景）

场景说明

在安全集群环境下，各个组件之间的相互通信不能够简单的互通，而需要在通信之前进行相互认证，以确保通信的安全性。HBase应用开发需要进行ZooKeeper和Kerberos安全认证。用于ZooKeeper认证的文件为“jaas.conf”，用于Kerberos安全认证文件为keytab文件和krb5.conf文件。具体使用方法在样例代码的“README.md”中会有详细说明。

安全认证主要采用代码认证方式。支持Oracle JAVA平台和IBM JAVA平台。

以下代码在“com.huawei.bigdata.hbase.examples”包的“TestMain”类中。

- 代码认证

```
try {
   init();
   login();
   }
catch (IOException e) {
   LOG.error("Failed to login because ", e);
   return;
}
```

- 初始化配置

```
private static void init() throws IOException {
        // Default load from conf directory
        conf = HBaseConfiguration.create();
        //In Windows environment
        String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
[1]
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
```

```
        conf.addResource(new Path(userdir + "core-site.xml"), false);
        conf.addResource(new Path(userdir + "hdfs-site.xml"), false);
        conf.addResource(new Path(userdir + "hbase-site.xml"), false);
    }
```

[1]**userdir**获取的是编译后资源路径下conf目录的路径。

前提条件

已获取样例工程运行所需的配置文件及认证文件，详细操作请参见**准备连接HBase集群配置文件**。

配置安全登录

请根据实际情况，在“com.huawei.bigdata.hbase.examples”包的“TestMain”类中修改“userName”为实际用户名，例如“developuser”。

```
private static void login() throws IOException {
    if (User.isHBaseSecurityEnabled(conf)) {
        userName = "developuser";

        //In Windows environment
        String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;

        /*
         * if need to connect zk, please provide jaas info about zk. of course,
         * you can do it as below:
         * System.setProperty("java.security.auth.login.config", confDirPath +
         * "jaas.conf"); but the demo can help you more : Note: if this process
         * will connect more than one zk cluster, the demo may be not proper. you
         * can contact us for more help
         */
        LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, userName, userKeytabFile);
        LoginUtil.login(userName, userKeytabFile, krb5File, conf);
    }
}
```

11.2.4.2 HBase 服务数据读写示例安全认证（多集群互信场景）

场景说明

当不同的多个Manager系统下安全模式的集群需要互相访问对方的资源时，管理员可以设置互信的系统，使外部系统的用户可以在本系统中使用。每个系统用户安全使用的范围定义为“域”，不同的Manager系统需要定义唯一的域名。跨Manager访问实际上就是用户跨域使用。集群配置互信具体操作步骤请参考**集群互信管理**章节。

多集群互信场景下，以符合跨域访问的用户身份，使用从其中一个manager系统中获取到的用于Kerberos安全认证的keytab文件和principal文件，以及多个Manager系统各自的客户端配置文件，可实现一次认证登录后访问调用多集群的HBase服务。

以下代码在hbase-example样例工程的“com.huawei.bigdata.hbase.examples”包的“TestMultipleLogin”类中。

- 代码认证

```
    List<String> confDirectorys = new ArrayList<>();
    List<Configuration> confs = new LinkedList<>();
    try {
        // conf directory
        confDirectorys.add("hadoopDomain");[1]
        confDirectorys.add("hadoop1Domain");[2]
```



```
            for (String confDir : confDirectorys) {
                confs.add(init(confDir));[3]
            }

            login(confs.get(0), confDirectorys.get(0));[4]
        } catch (IOException e) {
            LOG.error("Failed to login because ", e);
            return;
        }
```

[1]此处hadoopDomain为保存用户凭证和其中一个集群的配置文件目录名称，该目录相对路径为hbase-example/src/main/resources/hadoopDomain，可根据需要进行变更。

[2]此处hadoop1Domain为保存另一个集群配置文件的目录名称，该目录相对路径为hbase-example/src/main/resources/hadoop1Domain，可根据需要进行变更。

[3]依次初始化conf对象。

[4]进行登录认证。

- 初始化配置

```
private static Configuration init(String confDirectoryName) throws IOException {
        // Default load from conf directory
        Configuration conf = HBaseConfiguration.create();
        //In Windows environment
        String userdir = TestMain.class.getClassLoader().getResource(confDirectoryName).getPath() +
File.separator;
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + confDirectoryName +
File.separator;
        conf.addResource(new Path(userdir + "core-site.xml"), false);
        conf.addResource(new Path(userdir + "hdfs-site.xml"), false);
        conf.addResource(new Path(userdir + "hbase-site.xml"), false);
        return conf;
}
```

前提条件

已获取样例工程运行所需的配置文件及认证文件，详细操作请参见**准备连接HBase集群配置文件**。

配置安全登录

请根据实际情况，在hbase-example样例工程的“com.huawei.bigdata.hbase.examples”包的“TestMultipleLogin”类中修改“userName”为实际用户名，例如“developuser”。

```
private static void login(Configuration conf, String confDir) throwsIOException {

    if (User.isHBaseSecurityEnabled(conf)) {
      userName = " developuser ";

      //In Windows environment
      String userdir = TestMain.class.getClassLoader().getResource(confDir).getPath() + File.separator;
      //In Linux environment
      //String userdir = System.getProperty("user.dir") + File.separator + confDir + File.separator;

      userKeytabFile = userdir + "user.keytab";
      krb5File = userdir + "krb5.conf";

      /*

       * if need to connect zk, please provide jaas info about zk. of course,
       * you can do it as below:
```



```
         * System.setProperty("java.security.auth.login.config",confDirPath +
         * "jaas.conf"); but the demo can help you more : Note: if this process
         * will connect more than one zk cluster, the demo may be not proper. you
         * can contact us for more help
         */

        LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, userName,userKeytabFile);
        LoginUtil.login(userName, userKeytabFile, krb5File, conf);
    }
}
```

11.2.4.3 调用 REST 接口访问 HBase 应用安全认证

场景说明

HBase服务安装时可选部署RESTServer实例，可通过访问HBase REST服务的形式调用HBase相关操作，包括对Namespace、table的操作等。访问HBase REST服务同样需要进行Kerberos认证。

前提条件

已获取样例工程运行所需的配置文件及认证文件，详细操作请参见**准备连接HBase集群配置文件**。

配置安全登录

该场景下不需要进行初始化配置，仅需要用于Kerberos安全认证的keytab文件和krb5.conf文件。

以下代码在hbase-rest-example样例工程的“com.huawei.bigdata.hbase.examples”包的“HBaseRestTest”类中。

- 代码认证

  请根据实际情况，修改“principal”为实际用户名，例如“developuser”。

```
        //In Windows environment
        String userdir = HBaseRestTest.class.getClassLoader().getResource("conf").getPath() +
File.separator;[1]
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
        String principal = "developuser";
        login(principal, userKeytabFile, krb5File);
        // RESTServer's hostname.
        String restHostName = "10.120.16.170";[2]
        String securityModeUrl = new
StringBuilder("https://").append(restHostName).append(":21309").toString();
        String nonSecurityModeUrl = new
StringBuilder("http://").append(restHostName).append(":21309").toString();
        HBaseRestTest test = new HBaseRestTest();

        //If cluster is non-security mode，use nonSecurityModeUrl as  parameter.
        test.test(securityModeUrl);[3]
```

  [1]**userdir**获取的是编译后资源路径下conf目录的路径。

  [2]修改**restHostName**为待访问的RestServer实例所在节点IP地址，并将访问节点IP配置到运行样例代码的本机hosts文件中。

  RestServer实例IP地址可登录FusionInsight Manager，选择“集群 > 服务 > HBase > 实例”获取。

  [3]安全模式采用https模式进行访问HBase REST服务，传入“securityModeUrl”作为test.test()参数。

- 安全登录

```
    private static void login(String principal, String userKeytabFile, String krb5File) throws
LoginException {
        Map<String, String> options = new HashMap<>();
        options.put("useTicketCache", "false");
        options.put("useKeyTab", "true");
        options.put("keyTab", userKeytabFile);

        /**
         * Krb5 in GSS API needs to be refreshed so it does not throw the error
         * Specified version of key is not available
         */

        options.put("refreshKrb5Config", "true");
        options.put("principal", principal);
        options.put("storeKey", "true");
        options.put("doNotPrompt", "true");
        options.put("isInitiator", "true");
        options.put("debug", "true");
        System.setProperty("java.security.krb5.conf", krb5File);
        Configuration config = new Configuration() {
            @Override
            public AppConfigurationEntry[] getAppConfigurationEntry(String name) {
                return new AppConfigurationEntry[] {
                    new AppConfigurationEntry("com.sun.security.auth.module.Krb5LoginModule",
                        AppConfigurationEntry.LoginModuleControlFlag.REQUIRED, options)
                };
            }
        };
        subject = new Subject(false, Collections.singleton(new KerberosPrincipal(principal)),
Collections.EMPTY_SET,
            Collections.EMPTY_SET);
        LoginContext loginContext = new LoginContext("Krb5Login", subject, null, config);
        loginContext.login();
}
```

11.2.4.4 访问 HBase ThriftServer 安全认证

操作场景

HBase把Thrift结合起来可以向外部应用提供HBase服务。在HBase服务安装时可选部署ThriftServer实例，ThriftServer系统可访问HBase的用户，拥有HBase所有NameSpace和表的读、写、执行、创建和管理的权限。访问ThriftServer服务同样需要进行Kerberos认证。HBase实现了两套Thrift Server服务，此处“hbase-thrift-example”为ThriftServer实例服务的调用实现。

前提条件

已获取样例工程运行所需的配置文件及认证文件，详细操作请参见**准备连接HBase集群配置文件**。

配置样例代码

- 代码认证

  以下代码在“hbase-thrift-example”样例工程的“com.huawei.bigdata.hbase.examples”包的“TestMain”类中。

```
    private static void init() throws IOException {
        // Default load from conf directory
        conf = HBaseConfiguration.create();

        String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
[1]
```

```
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
        conf.addResource(new Path(userdir + "core-site.xml"), false);
        conf.addResource(new Path(userdir + "hdfs-site.xml"), false);
        conf.addResource(new Path(userdir + "hbase-site.xml"), false);
}
```

[1]**userdir**获取的是编译后资源路径下conf目录的路径。

- 安全登录

  请根据实际情况，修改“userName”为实际用户名，例如“developuser”。

```
    private static void login() throws IOException {
        if (User.isHBaseSecurityEnabled(conf)) {
            userName = " developuser ";

            //In Windows environment
            String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() +
File.separator;
            //In Linux environment
            //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;

            userKeytabFile = userdir + "user.keytab";
            krb5File = userdir + "krb5.conf";

            /*
             * if need to connect zk, please provide jaas info about zk. of course,
             * you can do it as below:
             * System.setProperty("java.security.auth.login.config", confDirPath +
             * "jaas.conf"); but the demo can help you more : Note: if this process
             * will connect more than one zk cluster, the demo may be not proper. you
             * can contact us for more help
             */
            LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, userName,
userKeytabFile);
            LoginUtil.login(userName, userKeytabFile, krb5File, conf);
        }
}
```

- 连接ThriftServer实例

```
    try {
        test = new ThriftSample();
        test.test("10.120.16.170", THRIFT_PORT, conf);[2]
    } catch (TException | IOException e) {
        LOG.error("Test thrift error", e);
    }
```

[2]test.test()传入参数为待访问的ThriftServer实例所在节点ip地址，需根据实际运行集群情况进行修改，且该节点ip需要配置到运行样例代码的本机hosts文件中。

“THRIFT_PORT”为ThriftServer实例的配置参数"hbase.regionserver.thrift.port"对应的值。ThriftServer实例所在节点IP地址可通过登录FusionInsight Manager，选择“集群 > 服务 > HBase > 实例”获取。

11.2.4.5 HBase 访问多 ZooKeeper 场景安全认证

场景说明

在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper时，为了避免访问连接ZooKeeper认证冲突，提供了样例代码使HBase客户端访问FusionInsight ZooKeeper和客户应用访问第三方ZooKeeper。

前提条件

已获取样例工程运行所需的配置文件及认证文件，详细操作请参见**准备连接HBase集群配置文件**。

配置样例代码

以下为“src/main/resources”目录下提供的与认证相关的配置文件。

- zoo.cfg

```
# The configuration in jaas.conf used to connect fi
zookeeper.zookeeper.sasl.clientconfig=Client_new[1]
# Principal of fi zookeeper server side.
zookeeper.server.principal=zookeeper/hadoop.hadoop.com[2]
# Set true if the fi cluster is security mode.
# The other two parameters doesn't work if the value is false.
zookeeper.sasl.client=true[3]
```

[1] zookeeper.sasl.clientconfig：指定使用jaas.conf文件中的对应配置访问FusionInsight ZooKeeper；

[2] zookeeper.server.principal：指定ZooKeeper服务端使用principal，格式为“zookeeper/hadoop.*系统域名*”，例如：zookeeper/hadoop.HADOOP.COM。系统域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数值获取。；

[3] zookeeper.sasl.client：如果MRS集群是安全模式，该值设置为“true”，否则设置为“false”，设置为“false”的情况下，“zookeeper.sasl.clientconfig”和“zookeeper.server.principal”参数不生效。

- jaas.conf

```
Client_new {  [4]
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="D:\\work\\sample_project\\src\\hbase-examples\\hbase-zk-example\\target\\classes\\conf\
\user.keytab" [5]
  principal="hbaseuser1"
  useTicketCache=false
  storeKey=true
  debug=true;
};
Client {  [6]
  org.apache.zookeeper.server.auth.DigestLoginModule required
  username="bob"
  password="xxxxxx"; [7]
};
```

[4] Client_new：zoo.cfg中指定的读取配置，当该名称修改时，需要同步修改zoo.cfg中对应配置。

[5] keyTab ： 指明工程使用的“user.keytab”在运行样例的主机上的保存路径，使用绝对路径便于更好定位文件位置。在Windows环境和Linux环境下配置时需注意区分不同操作系统路径书写方式，即“\\”与“\”差异。

[6] Client：第三方ZooKeeper使用该配置进行访问连接，具体连接认证配置由第三方ZooKeeper版本决定。

[7] password：密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

11.3 开发 HBase 应用

11.3.1 HBase 数据读写样例程序

11.3.1.1 HBase 样例程序开发思路

通过典型场景，您可以快速学习和掌握HBase的开发过程，并且对关键的接口函数有所了解。

场景说明

假定用户开发一个应用程序，用于管理企业中的使用A业务的用户信息，如表11-7所示，A业务操作流程如下：

- 创建用户信息表。
- 在用户信息中新增用户的学历、职称等信息。
- 根据用户编号查询用户姓名和地址。
- 根据用户姓名进行查询。
- 查询年龄段在[20-29]之间的用户信息。
- 数据统计，统计用户信息表的人员数、年龄最大值、年龄最小值、平均年龄。
- 用户销户，删除用户信息表中该用户的数据。
- A业务结束后，删除用户信息表。

表 11-7 用户信息

编号	姓名	性别	年龄	地址
12005000201	张三	男	19	广东省深圳市
12005000202	李婉婷	女	23	河北省石家庄市
12005000203	王明	男	26	浙江省宁波市
12005000204	李刚	男	18	湖北省襄阳市
12005000205	赵恩如	女	21	江西省上饶市
12005000206	陈龙	男	32	湖南省株洲市
12005000207	周微	女	29	河南省南阳市
12005000208	杨艺文	女	30	重庆市开县
12005000209	徐兵	男	26	陕西省渭南市

编号	姓名	性别	年龄	地址
12005000210	肖凯	男	25	辽宁省大连市

数据规划

合理地设计表结构、行键、列名能充分利用HBase的优势。本样例工程以唯一编号作为RowKey，列都存储在info列族中。

注意

HBase表以“*命名空间:表名*”格式进行存储，若在创建表时不指定命名空间，则默认存储在“default”中。其中，“hbase”命名空间为系统表命名空间，请不要对该系统表命名空间进行业务建表或数据读写等操作。

功能分解

根据上述的业务场景进行功能分解，需要开发的功能点如表11-8所示。

表 11-8 在 HBase 中开发的功能

序号	步骤	代码实现
1	根据表11-7中的信息创建表。	请参见创建HBase表。
2	导入用户数据。	请参见向HBase表中插入数据。
3	增加“教育信息”列族，在用户信息中新增用户的学历、职称等信息。	请参见修改HBase表。
4	根据用户编号查询用户姓名和地址。	请参见使用Get API读取HBase表数据。
5	根据用户姓名进行查询。	请参见使用Filter过滤器读取HBase表数据。
6	为提升查询性能，创建二级索引或者删除二级索引。	请参见创建HBase表二级索引和基于二级索引查询HBase表数据。
7	用户销户，删除用户信息表中该用户的数据。	请参见删除HBase表数据。
8	A业务结束后，删除用户信息表。	请参见删除HBase表。

关键设计原则

HBase是以RowKey为字典排序的分布式数据库系统，RowKey的设计对性能影响很大，具体的RowKey设计请考虑与业务结合。

11.3.1.2 初始化 HBase 配置

功能介绍

HBase通过login方法来获取配置项。包括用户登录信息、安全认证信息等配置项。

代码样例

下面代码片段在com.huawei.bigdata.hbase.examples包的“TestMain”类的init方法中。

```
private static void init() throws IOException {
    // Default load from conf directory
    conf = HBaseConfiguration.create();
    //In Windows environment
    String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
    //In Linux environment
    //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
    conf.addResource(new Path(userdir + "core-site.xml"), false);
    conf.addResource(new Path(userdir + "hdfs-site.xml"), false);
    conf.addResource(new Path(userdir + "hbase-site.xml"), false);
  }
```

11.3.1.3 创建 HBase 客户端连接

功能介绍

HBase通过ConnectionFactory.createConnection(configuration)方法创建Connection对象。传递的参数为上一步创建的Configuration。

Connection封装了底层与各实际服务器的连接以及与ZooKeeper的连接。Connection通过ConnectionFactory类实例化。创建Connection是重量级操作，Connection是线程安全的，因此，多个客户端线程可以共享一个Connection。

典型的用法，一个客户端程序共享一个单独的Connection，每一个线程获取自己的Admin或Table实例，然后调用Admin对象或Table对象提供的操作接口。不建议缓存或者池化Table、Admin。Connection的生命周期由调用者维护，调用者通过调用close()，释放资源。

代码样例

以下代码片段是登录，创建Connection并创建表的示例，在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的HBaseSample方法中。

```
    private TableName tableName = null;
    private Connection conn = null;

    public HBaseSample(Configuration conf) throws IOException {
        this.tableName = TableName.valueOf("hbase_sample_table");
        this.conn = ConnectionFactory.createConnection(conf);
}
```

说明

登录代码要避免重复调用。

11.3.1.4 创建 HBase 表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin对象的createTable方法来创建表，并指定表名、列族名。创建表有两种方式（强烈建议采用预分Region建表方式）：

- 快速建表，即创建表后整张表只有一个Region，随着数据量的增加会自动分裂成多个Region。
- 预分Region建表，即创建表时预先分配多个Region，此种方法建表可以提高写入大量数据初期的数据写入速度。

说明

表的列名以及列族名不能包含特殊字符，可以由字母、数字以及下划线组成。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testCreateTable方法中。

```
public void testCreateTable() {
        LOG.info("Entering testCreateTable.");
        // Specify the table descriptor.
        TableDescriptorBuilder htd = TableDescriptorBuilder.newBuilder(tableName); ( 1 )

        // Set the column family name to info.
        ColumnFamilyDescriptorBuilder hcd =
ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes("info")); ( 2 )

        // Set data encoding methods, HBase provides DIFF,FAST_DIFF,PREFIX

        hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
        // Set compression methods, HBase provides two default compression
        // methods:GZ and SNAPPY
        // GZ has the highest compression rate,but low compression and
        // decompression effeciency,fit for cold data
        // SNAPPY has low compression rate, but high compression and
        // decompression effeciency,fit for hot data.
        // it is advised to use SNAANPPY
        hcd.setCompressionType(Compression.Algorithm.SNAPPY);//注[1]
        htd.setColumnFamily(hcd.build());  ( 3 )
        Admin admin = null;
        try {
          // Instantiate an Admin object.
          admin = conn.getAdmin();  ( 4 )
          if (!admin.tableExists(tableName)) {
            LOG.info("Creating table...");
            admin.createTable(htd.build());//注[2]  ( 5 )
            LOG.info(admin.getClusterMetrics().toString());
            LOG.info(admin.listNamespaceDescriptors().toString());
            LOG.info("Table created successfully.");
          } else {
            LOG.warn("table already exists");
          }
        } catch (IOException e) {
            LOG.error("Create table failed " ,e);
        } finally {
          if (admin != null) {
            try {
              // Close the Admin object.
              admin.close();
            } catch (IOException e) {
              LOG.error("Failed to close admin " ,e);
```



```
            }
          }
        }
        LOG.info("Exiting testCreateTable.");
}
```

解释

（1）创建表描述符

（2）创建列族描述符

（3）添加列族描述符到表描述符中

（4）获取Admin对象，Admin提供了建表、创建列族、检查表是否存在、修改表结构和列族结构以及删除表等功能。

（5）调用Admin的建表方法。

注意事项

- 注[1] 可以设置列族的压缩方式，代码片段如下：

```
//设置编码算法，HBase提供了DIFF，FAST_DIFF，PREFIX三种编码算法。
 hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);

 //设置文件压缩方式，HBase默认提供了GZ和SNAPPY两种压缩算法
 //其中GZ的压缩率高，但压缩和解压性能低，适用于冷数据
 //SNAPPY压缩率低，但压缩解压性能高，适用于热数据
 //建议默认开启SNAPPY压缩
 hcd.setCompressionType(Compression.Algorithm.SNAPPY);
```

- 注[2] 可以通过指定起始和结束RowKey，或者通过RowKey数组预分Region两种方式建表，代码片段如下：

```
// 创建一个预划分region的表
 byte[][] splits = new byte[4][];
 splits[0] = Bytes.toBytes("A");
 splits[1] = Bytes.toBytes("H");
 splits[2] = Bytes.toBytes("O");
 splits[3] = Bytes.toBytes("U");
 admin.createTable(htd, splits);
```

11.3.1.5 创建 HBase 表 Region

功能简介

一般通过org.apache.hadoop.hbase.client.HBaseAdmin进行多点分割。注意：分割操作只对空Region起作用。

本例使用multiSplit进行多点分割将HBase表按照“-∞~A”、“A~D”、“D~F”、“F~H”、“H~+∞”分为五个Region。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testMultiSplit方法中。

```
public void testMultiSplit() {
   LOG.info("Entering testMultiSplit.");

   Table table = null;
   Admin admin = null;
   try {
```



```
    admin = conn.getAdmin();

    // initilize a HTable object
    table = conn.getTable(tableName);
    Set<HRegionInfo> regionSet = new HashSet<HRegionInfo>();
    List<HRegionLocation> regionList = conn.getRegionLocator(tableName).getAllRegionLocations();
    for(HRegionLocation hrl : regionList){
        regionSet.add(hrl.getRegionInfo());
    }
    byte[][] sk = new byte[4][];
    sk[0] = "A".getBytes();
    sk[1] = "D".getBytes();
    sk[2] = "F".getBytes();
    sk[3] = "H".getBytes();
    for (RegionInfo regionInfo : regionSet) {
      admin.multiSplitSync(regionInfo.getRegionName(), sk);
    }
    LOG.info("MultiSplit successfully.");
  } catch (Exception e) {
    LOG.error("MultiSplit failed.");
  } finally {
    if (table != null) {
      try {
        // Close table object
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
    if (admin != null) {
        try {
          // Close the Admin object.
          admin.close();
        } catch (IOException e) {
            LOG.error("Close admin failed " ,e);
        }
    }
  }
  LOG.info("Exiting testMultiSplit.");
}
```

注意：分割操作只对空Region起作用。

11.3.1.6 向 HBase 表中插入数据

功能简介

HBase是一个面向列的数据库，一行数据，可能对应多个列族，而一个列族又可以对应多个列。通常，写入数据的时候，需要指定要写入的列（含列族名称和列名称）。HBase通过HTable的put方法来Put数据，可以是一行数据也可以是数据集。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testPut方法中。

```
public void testPut() {
        LOG.info("Entering testPut.");

        // Specify the column family name.
        byte[] familyName = Bytes.toBytes("info");
        // Specify the column name.
        byte[][] qualifiers = {
            Bytes.toBytes("name"), Bytes.toBytes("gender"), Bytes.toBytes("age"), Bytes.toBytes("address")
        };
```

```
    Table table = null;
    try {
        // Instantiate an HTable object.
        table = conn.getTable(tableName);
        List<Put> puts = new ArrayList<Put>();

        // Instantiate a Put object.
        Put put = putData(familyName, qualifiers,
            Arrays.asList("012005000201", "Zhang San", "Male", "19", "Shenzhen, Guangdong"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000202", "Li Wanting", "Female", "23", "Shijiazhuang, Hebei"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000203", "Wang Ming", "Male", "26", "Ningbo, Zhejiang"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000204", "Li Gang", "Male", "18", "Xiangyang, Hubei"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000205", "Zhao Enru", "Female", "21", "Shangrao, Jiangxi"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000206", "Chen Long", "Male", "32", "Zhuzhou, Hunan"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000207", "Zhou Wei", "Female", "29", "Nanyang, Henan"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000208", "Yang Yiwen", "Female", "30", "Kaixian, Chongqing"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000209", "Xu Bing", "Male", "26", "Weinan, Shaanxi"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000210", "Xiao Kai", "Male", "25", "Dalian, Liaoning"));
        puts.add(put);

        // Submit a put request.
        table.put(puts);

        LOG.info("Put successfully.");
    } catch (IOException e) {
        LOG.error("Put failed ", e);
    } finally {
        if (table != null) {
            try {
                // Close the HTable object.
                table.close();
            } catch (IOException e) {
                LOG.error("Close table failed ", e);
            }
        }
    }
    LOG.info("Exiting testPut.");
}
```


注意事项

不允许多个线程在同一时间共用同一个HTable实例。HTable是一个非线程安全类，因此，同一个HTable实例，不应该被多个线程同时使用，否则可能会带来并发问题。

11.3.1.7 创建 HBase 表二级索引

功能简介

一般都通过调用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中方法进行HBase二级索引的管理，该类中提供了创建索引的方法。

说明

二级索引不支持修改，如果需要修改，请先删除旧的然后重新创建。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的createIndex方法中。

```
public void createIndex() {
   LOG.info("Entering createIndex.");

   String indexName = "index_name";
   // Create hindex instance
   TableIndices tableIndices = new TableIndices();
   IndexSpecification iSpec = new IndexSpecification(indexName);
   iSpec.addIndexColumn(ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes("info")).build(),
         "name", ValueType.STRING);//注[1]
   tableIndices.addIndex(iSpec);

   HIndexAdmin iAdmin = null;
   Admin admin = null;
   try {

     admin = conn.getAdmin();
     iAdmin = HIndexClient.newHIndexAdmin(admin);

     // add index to the table
     iAdmin.addIndices(tableName, tableIndices);

     LOG.info("Create index successfully.");
   } catch (IOException e) {
     LOG.error("Create index failed " ,e);
   } finally {
     if (admin != null) {
        try {
          admin.close();
      } catch (IOException e) {
          LOG.error("Close admin failed " ,e);
      }
     }
     if (iAdmin != null) {
       try {
         // Close IndexAdmin Object
         iAdmin.close();
       } catch (IOException e) {
         LOG.error("Close admin failed " ,e);
       }
     }
   }
   LOG.info("Exiting createIndex.");
 }
```


新创建的二级索引默认是不启用的，如果需要启用指定的二级索引，可以参考如下代码片段。该代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的enableIndex方法中。

```
public void enableIndex() {
    LOG.info("Entering createIndex.");

    // Name of the index to be enabled
    String indexName = "index_name";

    List<String> indexNameList = new ArrayList<String>();
    indexNameList.add(indexName);

    HIndexAdmin iAdmin = null;
    Admin admin = null;
    try {
        admin = conn.getAdmin();
        iAdmin = HIndexClient.newHIndexAdmin(admin);

        // Alternately, enable the specified indices
        iAdmin.enableIndices(tableName, indexNameList);
        LOG.info("Successfully enable indices {}  of the table {}", indexNameList, tableName);
    } catch (IOException e) {
        LOG.error("Failed to enable indices {}  of the table {} . {}", indexNameList, tableName, e);
    } finally {
        if (admin != null) {
            try {
                admin.close();
            } catch (IOException e) {
                LOG.error("Close admin failed ", e);
            }
        }
        if (iAdmin != null) {
            try {
                iAdmin.close();
            } catch (IOException e) {
                LOG.error("Close admin failed ", e);
            }
        }
    }
}
```

注意事项

注[1]：创建联合索引

HBase支持在多个字段上创建二级索引，例如在列name和age上。

```
HIndexSpecification iSpecUnite = new HIndexSpecification(indexName);
 iSpecUnite.addIndexColumn(new HColumnDescriptor("info"), "name", ValueType.String);
 iSpecUnite.addIndexColumn(new HColumnDescriptor("info"), "age", ValueType.String);
```

相关操作

使用命令创建索引表。

您还可以通过TableIndexer工具在已有用户表中创建索引。

说明

<table_name>用户表必须存在。

```
hbase org.apache.hadoop.hbase.hindex.mapreduce.TableIndexer -Dtablename.to.index=<table_name> -
Dindexspecs.to.add='IDX1=>cf1:[q1->datatype];cf2:[q1->datatype],[q2->datatype],[q3-
>datatype]#IDX2=>cf1:[q5->datatype]' -Dindexnames.to.build='IDX1'
```

“#”用于区分不同的索引，“;”用于区分不同的列族，“,”用于区分不同的列。

tablename.to.index：创建索引的用户表表名。

indexspecs.to.add：创建索引对应的用户表列。

其中命令中各参数的含义如下：

- IDX1：索引名称
- cf1：列族名称。
- q1：列名。
- datatype：数据类型。数据类型仅支持Integer、String、Double、Float、Long、Short、Byte、Char类型。

11.3.1.8 基于二级索引查询 HBase 表数据

功能介绍

针对添加了二级索引的用户表，您可以通过Filter来查询数据。其数据查询性能高于针对无二级索引用户表的数据查询。

说明

- HIndex支持的Filter类型为“SingleColumnValueFilter”，“SingleColumnValueExcludeFilter”以及“SingleColumnValuePartitionFilter”。
- HIndex支持的Comparator为“BinaryComparator”，“BitComparator”，“LongComparator”，“DecimalComparator”，“DoubleComparator”，“FloatComparator”，“IntComparator”，“NullComparator”。

二级索引的使用规则如下：

- 针对某一列或者多列创建了单索引的场景下：
  - 当查询时使用此列进行过滤时，不管是AND还是OR操作，该索引都会被利用来提升查询性能。

    例如：Filter_Condition(IndexCol1) AND/OR Filter_Condition(IndexCol2)
  - 当查询时使用“索引列AND非索引列”过滤时，此索引会被利用来提升查询性能。

    例如：Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2) AND Filter_Condition(NonIndexCol1)
  - 当查询时使用“索引列OR非索引列”过滤时，此索引将不会被使用，查询性能不会因为索引得到提升。

    例如：Filter_Condition(IndexCol1) AND/OR Filter_Condition(IndexCol2) OR Filter_Condition(NonIndexCol1)
- 针对多个列创建的联合索引场景下：
  - 当查询时使用的列（多个），是联合索引所有对应列的一部分或者全部，且列的顺序与联合索引一致时，此索引会被利用来提升查询性能。

    例如，针对C1、C2、C3列创建了联合索引，生效的场景包括：

    Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2) AND Filter_Condition(IndexCol3)

    Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2)

    Filter_Condition(IndexCol1)

    不生效的场景包括：

Filter_Condition(IndexCol2) AND Filter_Condition(IndexCol3)

Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol3)

Filter_Condition(IndexCol2)

Filter_Condition(IndexCol3)

- 当查询时使用“索引列AND非索引列”过滤时，此索引会被利用来提升查询性能。

  例如：

  Filter_Condition(IndexCol1) AND Filter_Condition(NonIndexCol1)

  Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2) AND Filter_Condition(NonIndexCol1)

- 当查询时使用“索引列OR非索引列”过滤时，此索引不会被使用，查询性能不会因为索引得到提升。

  例如：

  Filter_Condition(IndexCol1) OR Filter_Condition(NonIndexCol1)

  (Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2))OR (Filter_Condition(NonIndexCol1))

- 当查询时使用多个列进行范围查询时，只有联合索引中最后一个列可指定取值范围，前面的列只能设置为“=”。

  例如：针对C1、C2、C3列创建了联合索引，需要进行范围查询时，只能针对C3设置取值范围，过滤条件为“C1=XXX，C2=XXX，C3=取值范围”。

- 针对添加了二级索引的用户表，可以通过Filter来查询数据，在单列索引和复合列索引上进行过滤查询，查询结果都与无索引结果相同，且其数据查询性能高于无二级索引用户表的数据查询性能。

代码样例

下面代码片段在com.huawei.hadoop.hbase.example包的“HBaseSample”类的testScanDataByIndex方法中：

样例：使用二级索引查找数据

```
public void testScanDataByIndex() {
  LOG.info("Entering testScanDataByIndex.");
  Table table = null;
  ResultScanner scanner = null;
  try {
    table = conn.getTable(tableName);

    // Create a filter for indexed column.
    Filter filter = new SingleColumnValueFilter(Bytes.toBytes("info"), Bytes.toBytes("name"),
        CompareOperator.EQUAL, "Li Gang".getBytes());
    Scan scan = new Scan();
    scan.setFilter(filter);
    scanner = table.getScanner(scan);
    LOG.info("Scan indexed data.");

    for (Result result : scanner) {
      for (Cell cell : result.rawCells()) {
        LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
          Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
          Bytes.toString(CellUtil.cloneValue(cell)));
      }
    }
    LOG.info("Scan data by index successfully.");
  } catch (IOException e) {
```



```
    LOG.error("Scan data by index failed.");
  } finally {
    if (scanner != null) {
      // Close the scanner object.
      scanner.close();
    }
    try {
      if (table != null) {
        table.close();
      }
    } catch (IOException e) {
      LOG.error("Close table failed.");
    }
  }

  LOG.info("Exiting testScanDataByIndex.");
}
```

注意事项

需要预先对字段name创建二级索引。

相关操作

基于二级索引表查询。

查询样例如下：

用户在hbase_sample_table的info列族的name列添加一个索引，在客户端执行，

```
hbase org.apache.hadoop.hbase.hindex.mapreduce.TableIndexer -Dtablename.to.index=hbase_sample_table -Dindexspecs.to.add='IDX1=>info:[name->String]' -Dindexnames.to.build='IDX1'
```

然后用户需要查询“info:name”，在**hbase shell**执行如下命令：

scan 'hbase_sample_table',
{FILTER=>"SingleColumnValueFilter(*family*,*qualifier*,*compareOp*,*comparator*,*filterIfMissing*,*latestVersionOnly*)"}

说明

hbase shell下面做复杂的查询请使用API进行处理。

参数说明：

- family：需要查询的列所在的列族，例如info；
- qualifier：需要查询的列，例如name；
- compareOp：比较符，例如=、>等；
- comparator：需要查找的目标值，例如binary:Zhang San；
- filterIfMissing：如果某一行不存在该列，是否过滤，默认值为false；
- latestVersionOnly：是否仅查询最新版本的值，默认值为false。

例如：

```
scan 'hbase_sample_table',{FILTER=>"SingleColumnValueFilter('info','name',=,'binary:Zhang San',true,true)"}
```

11.3.1.9 修改 HBase 表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin的modifyTable方法修改表信息。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testModifyTable方法中。

```
public void testModifyTable() {
  LOG.info("Entering testModifyTable.");

  // Specify the column family name.
  byte[] familyName = Bytes.toBytes("education");
  Admin admin = null;
  try {
    // Instantiate an Admin object.
    admin = conn.getAdmin();
    // Obtain the table descriptor.
    TableDescriptor htd = admin.getTableDescriptor(tableName);

    // Check whether the column family is specified before modification.
    if (!htd.hasColumnFamily(familyName)) {
      // Create the column descriptor.
      TableDescriptor tableBuilder = TableDescriptorBuilder.newBuilder(htd)
          .setColumnFamily(ColumnFamilyDescriptorBuilder.newBuilder(familyName).build()).build();


      // Disable the table to get the table offline before modifying
      // the table.
      admin.disableTable(tableName);//注[1]
      // Submit a modifyTable request.
      admin.modifyTable(tableBuilder);
      // Enable the table to get the table online after modifying the
      // table.
      admin.enableTable(tableName);
    }
    LOG.info("Modify table successfully.");
  } catch (IOException e) {
    LOG.error("Modify table failed " ,e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Close admin failed " ,e);
      }
    }
  }
  LOG.info("Exiting testModifyTable.");
}
```

注意事项

注[1] modifyTable只有表被disable时，才能生效。

11.3.1.10 使用 Get API 读取 HBase 表数据

功能简介

要从表中读取一条数据，首先需要实例化该表对应的Table实例，然后创建一个Get对象。也可以为Get对象设定参数值，如列族的名称和列的名称。查询到的行数据存储在Result对象中，Result中可以存储多个Cell。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testGet方法中。

```
public void testGet() {
    LOG.info("Entering testGet.");
    // Specify the column family name.
    byte[] familyName = Bytes.toBytes("info");
    // Specify the column name.
    byte[][] qualifier = { Bytes.toBytes("name"), Bytes.toBytes("address") };
    // Specify RowKey.
    byte[] rowKey = Bytes.toBytes("012005000201");
    Table table = null;
    try {
      // Create the Table instance.
      table = conn.getTable(tableName);
      // Instantiate a Get object.
      Get get = new Get(rowKey);
      // Set the column family name and column name.
      get.addColumn(familyName, qualifier[0]);
      get.addColumn(familyName, qualifier[1]);
      // Submit a get request.
      Result result = table.get(get);
      // Print query results.
      for (Cell cell : result.rawCells()) {
        LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
                    Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
                    Bytes.toString(CellUtil.cloneValue(cell)));
      }
      LOG.info("Get data successfully.");
    } catch (IOException e) {
      LOG.error("Get data failed " ,e);
    } finally {
      if (table != null) {
        try {
          // Close the HTable object.
          table.close();
        } catch (IOException e) {
          LOG.error("Close table failed " ,e);
        }
      }
    }
    LOG.info("Exiting testGet.");
  }
```

11.3.1.11 使用 Scan API 读取 HBase 表数据

功能简介

要从表中读取数据，首先需要实例化该表对应的Table实例，然后创建一个Scan对象，并针对查询条件设置Scan对象的参数值，为了提高查询效率，建议指定StartRow和StopRow。查询结果的多行数据保存在ResultScanner对象中，每行数据以Result对象形式存储，Result中存储了多个Cell。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testScanData方法中。

```
public void testScanData() {
    LOG.info("Entering testScanData.");
    Table table = null;
    // Instantiate a ResultScanner object.
```



```
  ResultScanner rScanner = null;
  try {
    // Create the Configuration instance.
    table = conn.getTable(tableName);
    // Instantiate a Get object.
    Scan scan = new Scan();
    scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
    // Set the cache size.
    scan.setCaching(1000);

    // Submit a scan request.
    rScanner = table.getScanner(scan);
    // Print query results.
     for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
       for (Cell cell : r.rawCells()) {
         LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
             Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
             Bytes.toString(CellUtil.cloneValue(cell)));
       }
    }
    LOG.info("Scan data successfully.");
  } catch (IOException e) {
    LOG.error("Scan data failed " ,e);
  } finally {
    if (rScanner != null) {
      // Close the scanner object.
      rScanner.close();
    }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testScanData.");
}
```

注意事项

- 建议Scan时指定StartRow和StopRow，一个有确切范围的Scan，性能会更好些。
- 可以设置Batch和Caching关键参数。
  - Batch
  - 使用Scan调用next接口每次最大返回的记录数，与一次读取的列数有关。
  - Caching
  - RPC请求返回next记录的最大数量，该参数与一次RPC获取的行数有关。

11.3.1.12 使用 Filter 过滤器读取 HBase 表数据

功能简介

HBase Filter主要在Scan和Get过程中进行数据过滤，通过设置一些过滤条件来实现，如设置RowKey、列名或者列值的过滤条件。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testSingleColumnValueFilter方法中。


```
public void testSingleColumnValueFilter() {
   LOG.info("Entering testSingleColumnValueFilter.");
   Table table = null;

   ResultScanner rScanner = null;

   try {

    table = conn.getTable(tableName);

    Scan scan = new Scan();
    scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
    // Set the filter criteria.
    SingleColumnValueFilter filter = new SingleColumnValueFilter(
        Bytes.toBytes("info"), Bytes.toBytes("name"), CompareOperator.EQUAL,
        Bytes.toBytes("Xu Bing"));
    scan.setFilter(filter);
    // Submit a scan request.
    rScanner = table.getScanner(scan);
    // Print query results.
     for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
       for (Cell cell : r.rawCells()) {
         LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
             Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
             Bytes.toString(CellUtil.cloneValue(cell)));
       }
     }
    LOG.info("Single column value filter successfully.");
   } catch (IOException e) {
    LOG.error("Single column value filter failed " ,e);
   } finally {
       if (rScanner != null) {
          // Close the scanner object.
          rScanner.close();
        }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
   }
   LOG.info("Exiting testSingleColumnValueFilter.");
 }
```

注意事项

当前二级索引不支持使用SubstringComparator类定义的对象作为Filter的比较器。

例如，如下示例中的用法当前不支持：

```
Scan scan = new Scan();
filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL);
filterList.addFilter(new SingleColumnValueFilter(Bytes
.toBytes(columnFamily), Bytes.toBytes(qualifier),
CompareOperator.EQUAL, new SubstringComparator(substring)));
scan.setFilter(filterList);
```

11.3.1.13 删除 HBase 表数据

功能简介

HBase通过Table实例的delete方法来Delete数据，可以是一行数据也可以是数据集。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testDelete方法中。

```
public void testDelete() {
  LOG.info("Entering testDelete.");

  byte[] rowKey = Bytes.toBytes("012005000201");

  Table table = null;
  try {
    // Instantiate an HTable object.
    table = conn.getTable(tableName);

    // Instantiate an Delete object.
    Delete delete = new Delete(rowKey);

    // Submit a delete request.
    table.delete(delete);

    LOG.info("Delete table successfully.");
  } catch (IOException e) {
    LOG.error("Delete table failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testDelete.");
 }
```

说明

如果被删除的cell所在的列族上设置了二级索引，也会同步删除索引数据。

11.3.1.14 删除 HBase 二级索引

功能简介

一般都通过调用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中方法进行HBase二级索引的管理，该类中提供了索引的查询和删除等方法。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的dropIndex方法中。

```
public void dropIndex() {
  LOG.info("Entering dropIndex.");
  String indexName = "index_name";
  List<String> indexNameList = new ArrayList<String>();
  indexNameList.add(indexName);

  IndexAdmin iAdmin = null;
  try {
    // Instantiate HIndexAdmin Object
    iAdmin = HIndexClient.newHIndexAdmin(conn.getAdmin());
    // Delete Secondary Index
```

```
    iAdmin.dropIndex(tableName, indexNameList);

    LOG.info("Drop index successfully.");
  } catch (IOException e) {
    LOG.error("Drop index failed.");
  } finally {
    if (iAdmin != null) {
      try {
        // Close Secondary Index
        iAdmin.close();
      } catch (IOException e) {
        LOG.error("Close admin failed.");
      }
    }
  }
  LOG.info("Exiting dropIndex.");
}
```

11.3.1.15 删除 HBase 表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin的deleteTable方法来删除表。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的dropTable方法中。

```
public void dropTable() {
  LOG.info("Entering dropTable.");
  Admin admin = null;
  try {
    admin = conn.getAdmin();
    if (admin.tableExists(tableName)) {
      // Disable the table before deleting it.
      admin.disableTable(tableName);
      // Delete table.
      admin.deleteTable(tableName);//注[1]
    }
    LOG.info("Drop table successfully.");
  } catch (IOException e) {
    LOG.error("Drop table failed " ,e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Close admin failed " ,e);
      }
    }
  }
  LOG.info("Exiting dropTable.");
}
```

注意事项

注[1] 只有表被disable时，才能被删除掉，所以deleteTable常与disableTable，enableTable，tableExists，isTableEnabled，isTableDisabled结合在一起使用。

11.3.1.16 创建 Phoenix 表

功能简介

Phoenix依赖HBase作为其后备存储，支持标准SQL和JDBC API的强大功能，使得SQL用户可以访问HBase集群。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“PhoenixSample”类的testCreateTable方法中。

```
/**
 * Create Table
 */
public void testCreateTable() {
  LOG.info("Entering testCreateTable.");
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Create table
  String createTableSQL =
      "CREATE TABLE IF NOT EXISTS TEST (id integer not null primary key, name varchar, "
          + "account char(6), birth date)";
  try (Connection conn = DriverManager.getConnection(url, props);
      Statement stat = conn.createStatement()) {
    // Execute Create SQL
    stat.executeUpdate(createTableSQL);
    LOG.info("Create table successfully.");
  } catch (Exception e) {
    LOG.error("Create table failed.", e);
  }
  LOG.info("Exiting testCreateTable.");
}
/**
 * Drop Table
 */
public void testDrop() {
  LOG.info("Entering testDrop.");
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Delete table
  String dropTableSQL = "DROP TABLE TEST";

  try (Connection conn = DriverManager.getConnection(url, props);
      Statement stat = conn.createStatement()) {
    stat.executeUpdate(dropTableSQL);
    LOG.info("Drop successfully.");
  } catch (Exception e) {
    LOG.error("Drop failed.", e);
  }
  LOG.info("Exiting testDrop.");
}
```

11.3.1.17 向 Phoenix 表中插入数据

功能简介

使用Phoenix实现写数据。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“PhoenixSample”类的testPut方法中。


```
/**
 * Put data
 */
public void testPut() {
  LOG.info("Entering testPut.");
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Insert
  String upsertSQL =
      "UPSERT INTO TEST VALUES(1,'John','100000', TO_DATE('1980-01-01','yyyy-MM-dd'))";
  try (Connection conn = DriverManager.getConnection(url, props);
      Statement stat = conn.createStatement()){
    // Execute Update SQL
    stat.executeUpdate(upsertSQL);
    conn.commit();
    LOG.info("Put successfully.");
  } catch (Exception e) {
    LOG.error("Put failed.", e);
  }
  LOG.info("Exiting testPut.");
}
```

11.3.1.18 读取 Phoenix 表数据

功能简介

使用Phoenix实现读数据。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“PhoenixSample”类的testSelect方法中。

```
/**
 * Select Data
 */
public void testSelect() {
  LOG.info("Entering testSelect.");
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Query
  String querySQL = "SELECT * FROM TEST WHERE id = ?";
  Connection conn = null;
  PreparedStatement preStat = null;
  Statement stat = null;
  ResultSet result = null;
  try {
    // Create Connection
    conn = DriverManager.getConnection(url, props);
    // Create Statement
    stat = conn.createStatement();
    // Create PrepareStatement
    preStat = conn.prepareStatement(querySQL);
    // Execute query
    preStat.setInt(1, 1);
    result = preStat.executeQuery();
    // Get result
    while (result.next()) {
      int id = result.getInt("id");
      String name = result.getString(1);
      System.out.println("id: " + id);
      System.out.println("name: " + name);
    }
    LOG.info("Select successfully.");
  } catch (Exception e) {
    LOG.error("Select failed.", e);
  } finally {
    if (null != result) {
```



```
      try {
        result.close();
      } catch (Exception e2) {
        LOG.error("Result close failed.", e2);
      }
    }
    if (null != stat) {
      try {
        stat.close();
      } catch (Exception e2) {
        LOG.error("Stat close failed.", e2);
      }
    }
    if (null != conn) {
      try {
        conn.close();
      } catch (Exception e2) {
        LOG.error("Connection close failed.", e2);
      }
    }
  }
  LOG.info("Exiting testSelect.");
}
```

11.3.1.19 配置 HBase 应用输出运行日志

功能介绍

将HBase客户端的日志单独输出到指定日志文件，与业务日志分开，方便分析定位HBase的问题。

如果进程中已经有log4j的配置，需要将“hbase-example\src\main\resources\log4j.properties”中RFA与RFAS相关的配置复制到已有的log4j配置中。

代码样例

```
hbase.root.logger=INFO,console,RFA              //hbase客户端日志输出配置，console：输出到控制台；RFA：输出到日志文件
hbase.security.logger=DEBUG,console,RFAS        //hbase客户端安全相关的日志输出配置，console：输出到控制台；RFAS：输出到日志文件
hbase.log.dir=/var/log/Bigdata/hbase/client/    //日志路径，根据实际路径修改，但目录要有写入权限
hbase.log.file=hbase-client.log                 //日志文件名
hbase.log.level=INFO                            //日志级别，如果需要更详细的日志定位问题，需要修改为DEBUG，修改完需要重启进程才能生效
hbase.log.maxbackupindex=20                     //最多保存的日志文件数目
# Security audit appender
hbase.security.log.file=hbase-client-audit.log  //审计日志文件命令
```

11.3.2 HBase Rest 接口调用样例程序

11.3.2.1 使用 REST 接口查询 HBase 集群信息

功能简介

使用REST服务，传入对应host与port组成的url，通过HTTPS协议，获取集群版本与状态信息。

代码样例

- 获取集群版本信息

以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getClusterVersion方法中。

```
private void getClusterVersion(String url) {
    String endpoint = "/version/cluster";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult((Optional<ResultModel>) result);
}
```

- 获取集群状态信息

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getClusterStatus方法中。

```
private void getClusterStatus(String url) {
    String endpoint = "/status/cluster";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

11.3.2.2 使用 REST 接口获取所有 HBase 表

功能简介

使用REST服务，传入对应host与port组成的URL，通过HTTPS协议，获取得到所有table。

代码样例

以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getAllUserTables方法中。

```
private void getAllUserTables(String url) {
    String endpoint = "/";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

11.3.2.3 使用 REST 接口操作 Namespace

功能简介

使用REST服务，传入对应host与port组成的url以及指定的Namespace，通过HTTPS协议，对Namespace进行创建、查询、删除，获取指定Namespace中表的操作。

注意

HBase表以“*命名空间:表名*”格式进行存储，若在创建表时不指定命名空间，则默认存储在“default”中。其中，“hbase”命名空间为系统表命名空间，请不要对该系统表命名空间进行业务建表或数据读写等操作。

代码样例

- 方法调用

```
// Namespace operations.
createNamespace(url, "testNs");
getAllNamespace(url);
deleteNamespace(url, "testNs");
getAllNamespaceTables(url, "default");
```

- 创建namespace

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的createNamespace方法中。

```
private void createNamespace(String url, String namespace) {
    String endpoint = "/namespaces/" + namespace;
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.POST, null);
    if (result.orElse(new ResultModel()).getStatusCode() == HttpStatus.SC_CREATED) {
        LOG.info("Create namespace '{}' success.", namespace);
    } else {
        LOG.error("Create namespace '{}' failed.", namespace);
    }
}
```

- 查询所有namespace

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getAllNamespace方法中。

```
private void getAllNamespace(String url) {
    String endpoint = "/namespaces";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

- 删除指定namespace

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的deleteNamespace方法中。

```
private void deleteNamespace(String url, String namespace) {
    String endpoint = "/namespaces/" + namespace;
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.DELETE, null);
    if (result.orElse(new ResultModel()).getStatusCode() == HttpStatus.SC_OK) {
        LOG.info("Delete namespace '{}' success.", namespace);
    } else {
        LOG.error("Delete namespace '{}' failed.", namespace);
    }
}
```

- 根据指定namespace获取其中的表

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getAllNamespaceTables方法中。

```
private void getAllNamespaceTables(String url, String namespace) {
    String endpoint = "/namespaces/" + namespace + "/tables";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

11.3.2.4 使用 REST 接口操作 HBase 表

功能简介

使用REST服务，传入对应host与port组成的url以及指定的tableName和jsonHTD，通过HTTPS协议，进行查询表信息，修改表，创建表以及删除表的操作。

代码样例

- 方法调用

```
// Add a table with specified info.
createTable(url, "testRest",
    "{\"name\":\"default:testRest\",\"ColumnSchema\":[{\"name\":\"cf1\"}," + "{\"name\":\"cf2\"}]}");

// Add column family 'testCF1' if not exist, else update the 'VERSIONS' to 3.
```

```
 // Notes: The unspecified property of this column family will be updated to default value.
modifyTable(url, "testRest",
    "{\"name\":\"testRest\",\"ColumnSchema\":[{\"name\":\"testCF1\"," + "\"VERSIONS\":\"3" +
"\"}]}");

// Describe specific Table.
descTable(url, "default:testRest");

// delete a table with specified info.
deleteTable(url, "default:testRest",
    "{\"name\":\"default:testRest\",\"ColumnSchema\":[{\"name\":\"testCF\"," + "\"VERSIONS
\":\"3\"}]}");
```

- 查询表信息

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的descTable方法中。

```
private void descTable(String url, String tableName) {
    String endpoint = "/" + tableName + "/schema";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult((Optional<ResultModel>) result);
}
```

- 修改表信息

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的modifyTable方法中。

```
private void modifyTable(String url, String tableName, String jsonHTD) {
    LOG.info("Start modify table.");
    String endpoint = "/" + tableName + "/schema";
    JsonElement tableDesc = new JsonParser().parse(jsonHTD);

    // Add a new column family or modify it.
    handleNormalResult(sendAction(url + endpoint, MethodType.POST, tableDesc));
}
```

- 创建表

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的createTable方法中。

```
private void createTable(String url, String tableName, String jsonHTD) {
    LOG.info("Start create table.");
    String endpoint = "/" + tableName + "/schema";
    JsonElement tableDesc = new JsonParser().parse(jsonHTD);

    // Add a table.
    handleCreateTableResult(sendAction(url + endpoint, MethodType.PUT, tableDesc));
}
```

- 删除表

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的deleteTable方法中。

```
private void deleteTable(String url, String tableName, String jsonHTD) {
    LOG.info("Start delete table.");
    String endpoint = "/" + tableName + "/schema";
    JsonElement tableDesc = new JsonParser().parse(jsonHTD);

    // delete a table.
    handleNormalResult(sendAction(url + endpoint, MethodType.DELETE, tableDesc));
}
```

11.3.3 访问 HBase ThriftServer 连接样例程序

11.3.3.1 通过 ThriftServer 实例操作 HBase 表

功能简介

传入ThriftServer实例所在host和提供服务的port，根据认证凭据及配置文件新建Thrift客户端，访问ThriftServer，进行根据指定namespace获取tablename以及创建表、删除表的操作。

代码样例

- 方法调用

```
// Get table of specified namespace.
getTableNamesByNamespace(client, "default");
// Create table.
createTable(client, TABLE_NAME);
// Delete specified table.
 deleteTable(client, TABLE_NAME);
```

- 根据指定namespace获取tablename

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的getTableNamesByNamespace方法中。

```
private void getTableNamesByNamespace(THBaseService.Iface client, String namespace) throws
TException {
    client.getTableNamesByNamespace(namespace)
        .forEach(
            tTableName -> LOGGER.info("{}", TableName.valueOf(tTableName.getNs(),
tTableName.getQualifier())));
 }
```

- 创建表

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的createTable方法中。

```
private void createTable(THBaseService.Iface client, String tableName) throws TException,
IOException {
    TTableName table = getTableName(tableName);
    TTableDescriptor descriptor = new TTableDescriptor(table);
    descriptor.setColumns(
        Collections.singletonList(new
TColumnFamilyDescriptor().setName(COLUMN_FAMILY.getBytes())));
    if (client.tableExists(table)) {
        LOGGER.warn("Table {} is exists, delete it.", tableName);
        client.disableTable(table);
        client.deleteTable(table);
    }
    client.createTable(descriptor, null);
    if (client.tableExists(table)) {
        LOGGER.info("Created {}.", tableName);
    } else {
        LOGGER.error("Create {} failed.", tableName);
    }
 }
```

- 删除表

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的deleteTable方法中。

```
private void deleteTable(THBaseService.Iface client, String tableName) throws TException,
IOException {
    TTableName table = getTableName(tableName);
    if (client.tableExists(table)) {
        client.disableTable(table);
        client.deleteTable(table);
```



```
        LOGGER.info("Deleted {}.", tableName);
    } else {
        LOGGER.warn("{} not exist.", tableName);
    }
}
```

11.3.3.2 通过 ThriftServer 实例向 HBase 表中写入数据

功能简介

传入ThriftServer实例所在host和提供服务的port，根据认证凭据及配置文件新建Thrift客户端，访问ThriftServer，分别使用put和putMultiple进行写数据操作。

代码样例

- 方法调用

```
// Write data with put.
putData(client, TABLE_NAME);

// Write data with putlist.
 putDataList(client, TABLE_NAME);
```

- 使用put进行写数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的putData方法中。

```
private void putData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test putData.");
    TPut put = new TPut();
    put.setRow("row1".getBytes());

    TColumnValue columnValue = new TColumnValue();
    columnValue.setFamily(COLUMN_FAMILY.getBytes());
    columnValue.setQualifier("q1".getBytes());
    columnValue.setValue("test value".getBytes());
    List<TColumnValue> columnValues = new ArrayList<>(1);
    columnValues.add(columnValue);
    put.setColumnValues(columnValues);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    client.put(table, put);
    LOGGER.info("Test putData done.");
}
```

- 使用putMultiple进行写数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的putDataList方法中。

```
private void putDataList(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test putDataList.");
    TPut put1 = new TPut();
    put1.setRow("row2".getBytes());
    List<TPut> putList = new ArrayList<>();

    TColumnValue q1Value = new TColumnValue(ByteBuffer.wrap(COLUMN_FAMILY.getBytes()),
        ByteBuffer.wrap("q1".getBytes()), ByteBuffer.wrap("test value".getBytes()));
    TColumnValue q2Value = new TColumnValue(ByteBuffer.wrap(COLUMN_FAMILY.getBytes()),
        ByteBuffer.wrap("q2".getBytes()), ByteBuffer.wrap("test q2 value".getBytes()));
    List<TColumnValue> columnValues = new ArrayList<>(2);
    columnValues.add(q1Value);
    columnValues.add(q2Value);
    put1.setColumnValues(columnValues);
    putList.add(put1);

    TPut put2 = new TPut();
    put2.setRow("row3".getBytes());
```



```
    TColumnValue columnValue = new TColumnValue();
    columnValue.setFamily(COLUMN_FAMILY.getBytes());
    columnValue.setQualifier("q1".getBytes());
    columnValue.setValue("test q1 value".getBytes());
    put2.setColumnValues(Collections.singletonList(columnValue));
    putList.add(put2);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    client.putMultiple(table, putList);
    LOGGER.info("Test putDataList done.");
}
```

11.3.3.3 通过 ThriftServer 实例读 HBase 表数据

功能简介

传入ThriftServer实例所在host和提供服务的port，根据认证凭据及配置文件新建Thrift客户端，访问ThriftServer，分别使用get和scan进行读数据操作。

代码样例

- 方法调用

```
// Get data with single get.
getData(client, TABLE_NAME);

// Get data with getlist.
getDataList(client, TABLE_NAME);

// Scan data.
 scanData(client, TABLE_NAME);
```

- 使用get进行读数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的getData方法中。

```
private void getData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test getData.");
    TGet get = new TGet();
    get.setRow("row1".getBytes());

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    TResult result = client.get(table, get);
    printResult(result);
    LOGGER.info("Test getData done.");
}
```

- 使用getlist进行读数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的getDataList方法中。

```
private void getDataList(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test getDataList.");
    List<TGet> getList = new ArrayList<>();
    TGet get1 = new TGet();
    get1.setRow("row1".getBytes());
    getList.add(get1);

    TGet get2 = new TGet();
    get2.setRow("row2".getBytes());
    getList.add(get2);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    List<TResult> resultList = client.getMultiple(table, getList);
    for (TResult tResult : resultList) {
        printResult(tResult);
```



```
    }
    LOGGER.info("Test getDataList done.");
}
```

- 使用scan进行读数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的scanData方法中。

```
private void scanData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test scanData.");
    int scannerId = -1;
    try {
        ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
        TScan scan = new TScan();
        scan.setLimit(500);
        scannerId = client.openScanner(table, scan);
        List<TResult> resultList = client.getScannerRows(scannerId, 100);
        while (resultList != null && !resultList.isEmpty()) {
            for (TResult tResult : resultList) {
                printResult(tResult);
            }
            resultList = client.getScannerRows(scannerId, 100);
        }
    } finally {
        if (scannerId != -1) {
            client.closeScanner(scannerId);
            LOGGER.info("Closed scanner {}.", scannerId);
        }
    }
    LOGGER.info("Test scanData done.");
}
```

11.3.4 HBase 访问多个 ZooKeeper 样例程序

功能简介

在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper，其中HBase客户端访问FusionInsight ZooKeeper，客户应用访问第三方ZooKeeper。

代码样例

以下代码片段在“hbase-zk-example\src\main\java\com\huawei\hadoop\hbase\example”包的“TestZKSample”类中，用户主要需要关注“login”和“connectApacheZK”这两个方法。

```
  private static void login(String keytabFile, String principal) throws IOException {
    conf = HBaseConfiguration.create();
    //In Windows environment
    String confDirPath = TestZKSample.class.getClassLoader().getResource("").getPath() + File.separator;
[1]
    //In Linux environment
    //String confDirPath = System.getProperty("user.dir") + File.separator + "conf" + File.separator;

    // Set zoo.cfg for hbase to connect to fi zookeeper.
    conf.set("hbase.client.zookeeper.config.path", confDirPath + "zoo.cfg");
    if (User.isHBaseSecurityEnabled(conf)) {
        // jaas.conf file, it is included in the client pakcage file
        System.setProperty("java.security.auth.login.config", confDirPath + "jaas.conf");
        // set the kerberos server info,point to the kerberosclient
        System.setProperty("java.security.krb5.conf", confDirPath + "krb5.conf");
        // set the keytab file name
        conf.set("username.client.keytab.file", confDirPath + keytabFile);
        // set the user's principal
        try {
            conf.set("username.client.kerberos.principal", principal);
```

```
            User.login(conf, "username.client.keytab.file", "username.client.kerberos.principal",
                InetAddress.getLocalHost().getCanonicalHostName());
        } catch (IOException e) {
            throw new IOException("Login failed.", e);
        }
    }
}
private void connectApacheZK() throws IOException, org.apache.zookeeper.KeeperException {
    try {
        // Create apache zookeeper connection.
        ZooKeeper digestZk = new ZooKeeper("127.0.0.1:2181", 60000, null);
        LOG.info("digest directory：{}", digestZk.getChildren("/", null));
        LOG.info("Successfully connect to apache zookeeper.");
    } catch (InterruptedException e) {
        LOG.error("Found error when connect apache zookeeper ", e);
    }
}
```

- [1]**userdir**获取的是编译后资源目录的路径。将初始化需要的配置文件“core-site.xml”、“hdfs-site.xml”、“hbase-site.xml”和用于安全认证的用户凭证文件放置到“src/main/resources”的目录下。
- “login”方法中的参数“java.security.auth.login.config”设置的jaas.conf文件用来设置访问ZooKeeper相关认证信息，样例代码中包含Client_new和Client两部分，Client_new的配置用来访问FusionInsight ZooKeeper，Client用来访问Apache ZooKeeper。
- “login”方法中的参数“hbase.client.zookeeper.config.path”用来设置访问FusionInsight ZooKeeper客户端的配置，主要涉及如下三个参数：
  - zookeeper.sasl.clientconfig：指定使用jaas.conf文件中的对应配置访问FusionInsight ZooKeeper；
  - zookeeper.server.principal：指定ZooKeeper服务端使用principal，格式为“zookeeper/hadoop.*系统域名*”，例如：zookeeper/hadoop.HADOOP.COM。系统域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数值获取。
  - zookeeper.sasl.client：如果MRS集群是安全模式，该值设置为“true”，否则设置为“false”，设置为“false”的情况下，“zookeeper.sasl.clientconfig”和“zookeeper.server.principal”参数不生效。

11.4 调测 HBase 应用

11.4.1 在本地 Windows 环境中调测 HBase 应用

操作场景

在程序代码完成开发后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

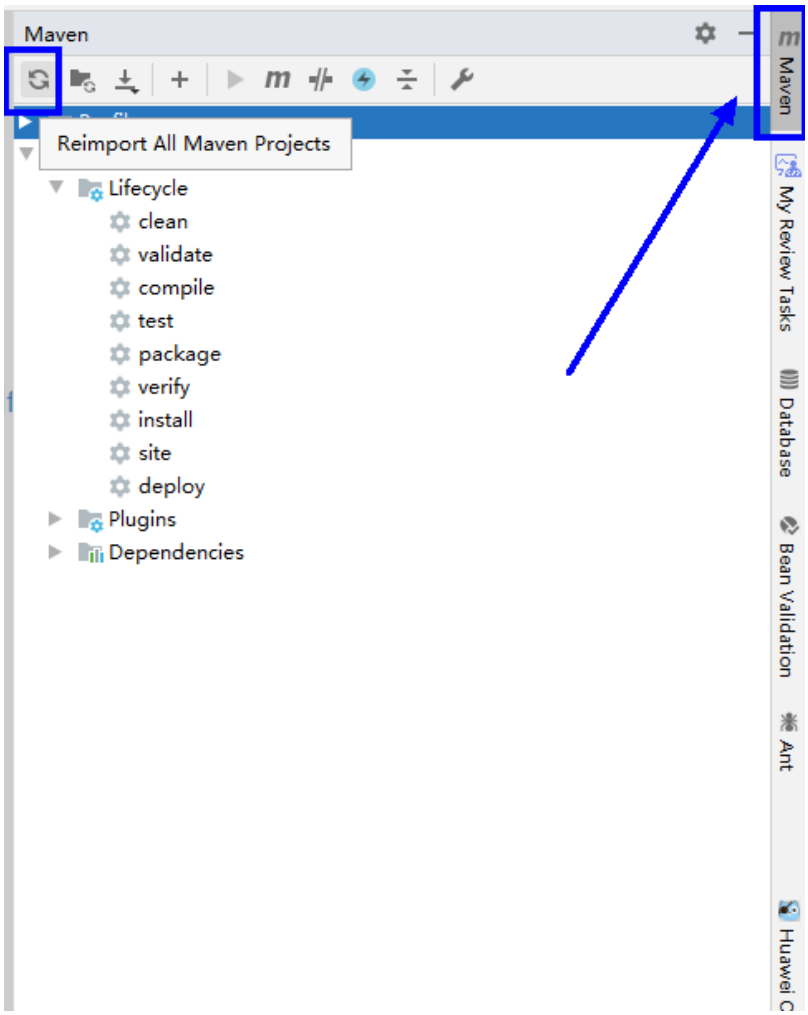
说明

- 如果Windows开发环境中使用IBM JDK，不支持在Windows环境中直接运行应用程序。
- 需要在运行样例代码的本机**hosts**文件中设置访问节点的主机名和公网IP地址映射，主机名和公网IP地址请保持一一对应。

在本地 Windows 环境中调测 HBase 应用

步骤1 单击IDEA右边Maven窗口的“Reimport All Maven Projects”，进行maven项目依赖import。

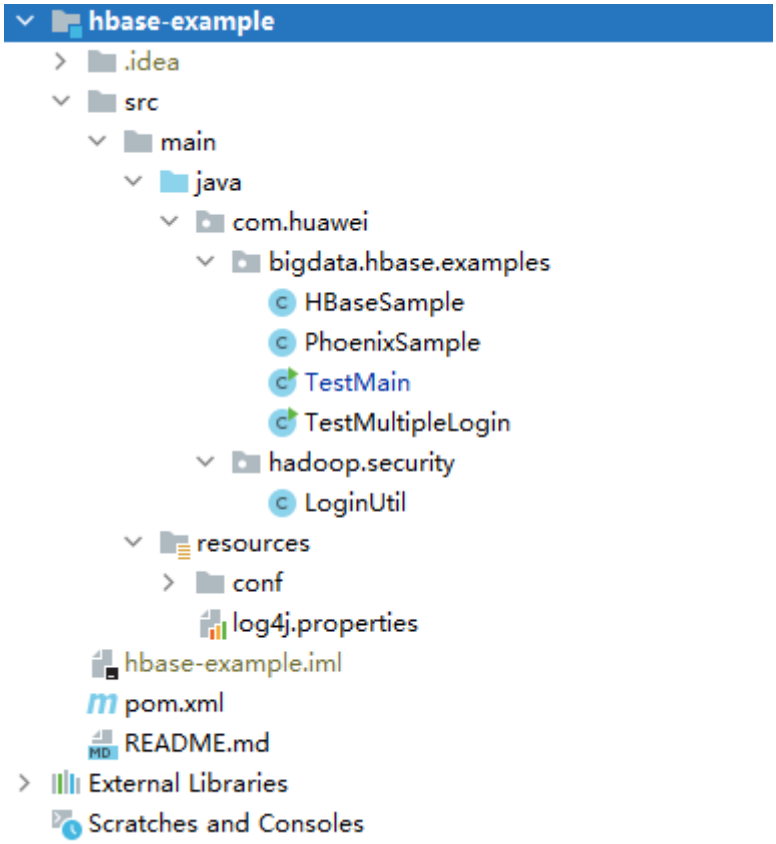
图 11-16 reimport projects



步骤2 编译运行程序。

放置好配置文件，并修改代码匹配登录用户后，文件列表如图11-17所示。

图 11-17 hbase-example 待编译目录列表

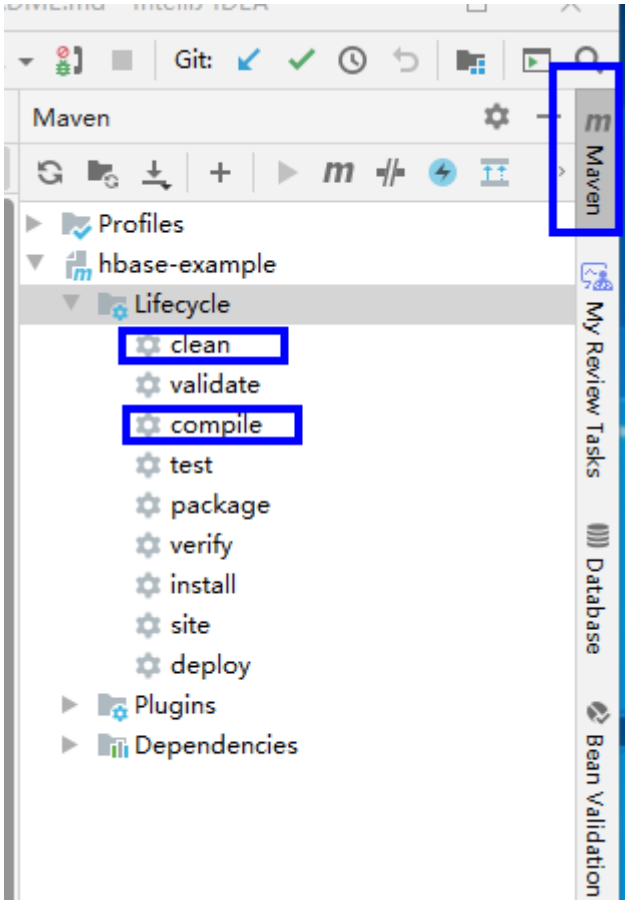


1. 编译方式有以下两种：
   - 方法一

     选择“Maven > *样例工程名称* > Lifecycle > clean”，双击“clean”运行maven的clean命令。

     选择“Maven > *样例工程名称* > Lifecycle > compile”，双击“compile”运行maven的compile命令。

图 11-18 mavne 工具 clean 和 compile



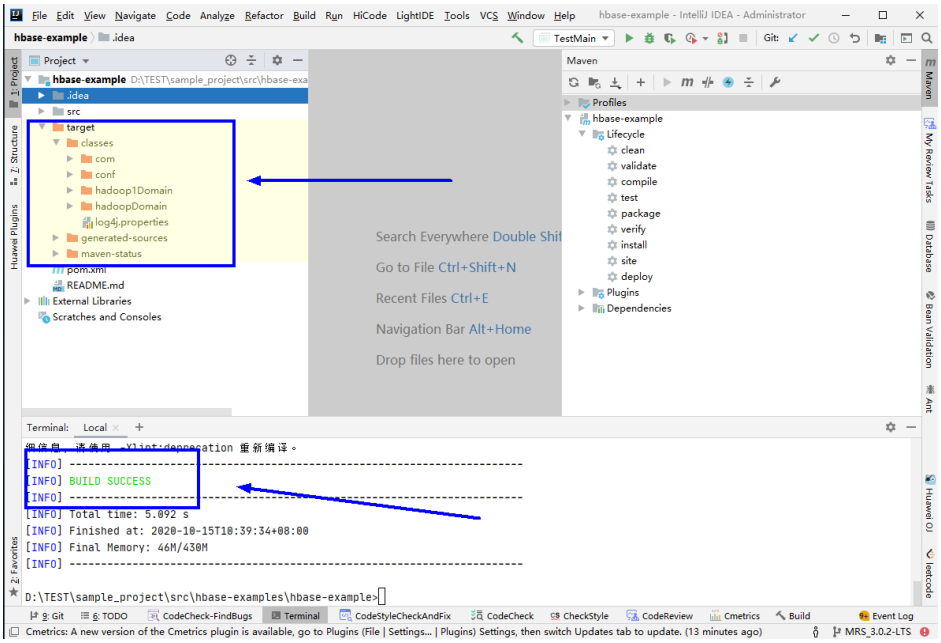
- 方法二

  在IDEA的下方Terminal窗口进入“pom.xml”所在目录，手动输入**mvn clean compile**命令进行编译。

图 11-19 idea termial 输入“mvn clean compile”

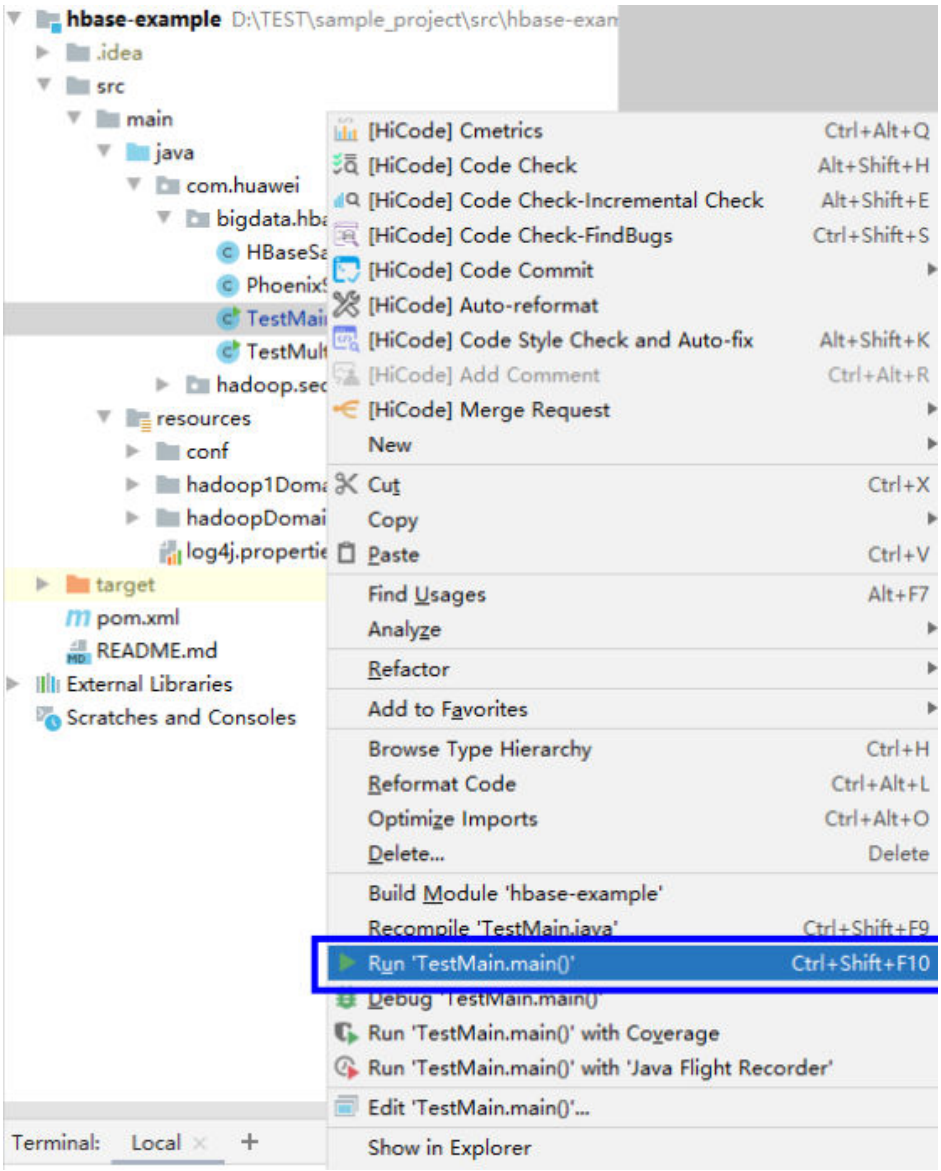
编译完成，打印“Build Success”，生成target目录。

图 11-20 编译完成



2. 运行程序。
   右键“TestMain.java”文件，选择“Run 'TestMain.main()”

图 11-21 运行程序



----结束

查看 Windows 调测结果

HBase应用程序运行完成后，可通过如下方式查看运行情况。

- 通过IntelliJ IDEA运行结果查看应用程序运行情况。
- 通过HBase日志获取应用程序运行情况。
- 登录HBase WebUI查看应用程序运行情况。可参见“更多信息 > 对外接口 > Web UI ”。
- 通过HBase shell命令查看应用程序运行情况。可参见“更多信息 > 对外接口 > Shell ”。

各样例程序运结果如下：

- hbase-example样例运行成功后，显示信息如下：


```
...
2020-09-09 22:11:48,496 INFO  [main] example.TestMain: Entering testCreateTable.
2020-09-09 22:11:48,894 INFO  [main] example.TestMain: Creating table...
2020-09-09 22:11:50,545 INFO  [main] example.TestMain: Master:
10-1-131-140,2130016000,1441784082485
Number of backup masters: 1
  10-1-131-130,2130016000,1441784098969
Number of live region servers: 3
  10-1-131-150,2130216020,1441784158435
  10-1-131-130,2130216020,1441784126506
  10-1-131-140,2130216020,1441784118303
Number of dead region servers: 0
Average load: 1.0
Number of requests: 0
Number of regions: 3
Number of regions in transition: 0
2020-09-09 22:11:50,562 INFO  [main] example.TestMain:
Lorg.apache.hadoop.hbase.NamespaceDescriptor;@11c6af6
2020-09-09 22:11:50,562 INFO  [main] example.TestMain: Table created successfully.
2020-09-09 22:11:50,563 INFO  [main] example.TestMain: Exiting testCreateTable.
2020-09-09 22:11:50,563 INFO  [main] example.TestMain: Entering testMultiSplit.
2020-09-09 22:11:50,630 INFO  [main] example.TestMain: MultiSplit successfully.
2020-09-09 22:11:50,630 INFO  [main] example.TestMain: Exiting testMultiSplit.
2020-09-09 22:11:50,630 INFO  [main] example.TestMain: Entering testPut.
2020-09-09 22:11:51,148 INFO  [main] example.TestMain: Put successfully.
2020-09-09 22:11:51,148 INFO  [main] example.TestMain: Exiting testPut.
2020-09-09 22:11:51,148 INFO  [main] example.TestMain: Entering createIndex.
...
```

说明

在Windows环境运行样例代码时会出现下面的异常，但是不影响业务：

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

- 日志说明

  日志级别默认为INFO，可以通过调整日志打印级别（DEBUG，INFO，WARN，ERROR，FATL）来显示更详细的信息。可以通过修改“log4j.properties”文件来实现，如：

```
hbase.root.logger=INFO,console
...
log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=INFO
# Make these two classes DEBUG-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
...
```

11.4.2 在 Linux 环境中调测 HBase 应用

操作场景

HBase应用程序支持在已安装或未安装HBase客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。

前提条件

- 已安装客户端时：
  - 已安装HBase客户端。
  - 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

- 未安装HBase客户端时：
  - Linux环境已安装JDK，版本号需要和IntelliJ IDEA导出Jar包使用的JDK版本一致。
  - 当Linux环境所在主机不是集群中的节点时，需要在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

已安装客户端时编译并运行程序

步骤1 导出Jar包。

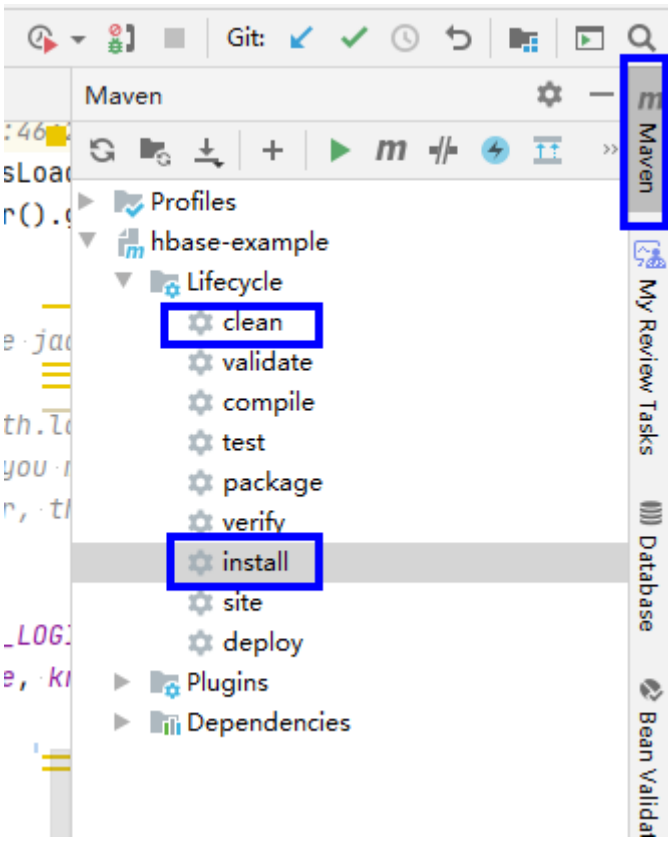
构建jar包方式有以下两种：

- 方法一：

  选择“Maven > *样例工程名称* > Lifecycle > clean”，双击“clean”运行maven的clean命令。

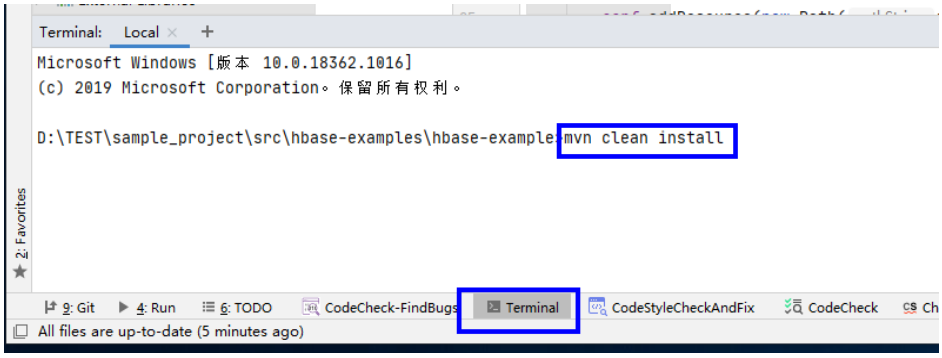
  选择“Maven > *样例工程名称* > Lifecycle > install”，双击“install”运行maven的install命令。

  图 11-22 maven 工具 clean 和 install

- 方法二：在IDEA的下方Terminal窗口进入“pom.xml”所在目录，手动输入**mvn clean install**命令进行编译。

图 11-23 idea termial 输入“mvn clean install”



编译完成，打印“BUILD SUCCESS”，生成target目录，生成jar包在target目录中。

步骤2 导出样例项目依赖的jar包。

在IDEA的下方Terminal窗口或其他命令行工具进入“pom.xml”所在目录。

执行命令**mvn dependency:copy-dependencies -DoutputDirectory=lib**。

在“pom.xml”所在目录将生成lib文件夹，其中包含样例项目所依赖的jar包。

步骤3 执行Jar包。

1. 使用客户端安装用户登录客户端所在节点，切换到客户端目录：
   cd *客户端安装目录*
2. 执行以下命令加载环境变量：
   source bigdata_env

   说明

   启用多实例功能后，为其他HBase服务实例进行应用程序开发时还需执行以下命令，切换指定服务实例的客户端。

   例如HBase2：**source /opt/client/HBase2/component_env**。
3. 将应用开发环境中生成的样例项目Jar包（非依赖jar包）上传至客户端运行环境的“*客户端安装目录*/HBase/hbase/lib”目录，还需将**准备连接HBase集群配置文件**获取的配置文件和认证文件复制到“*客户端安装目录*/HBase/hbase/conf”目录。
4. 进入目录“*客户端安装目录*/HBase/hbase”，执行以下命令运行Jar包。
   hbase *com.huawei.bigdata.hbase.examples.TestMain*
   其中，*hbase com.huawei.bigdata.hbase.examples.TestMain*为举例，具体以实际样例代码为准。

----结束

未安装客户端时编译并运行程序

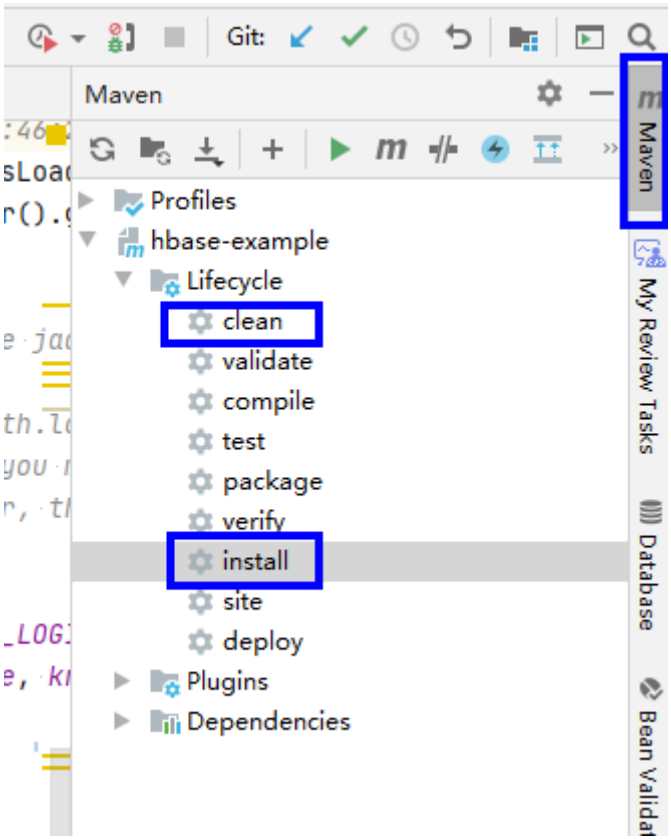
步骤1 导出Jar包。

构建jar包方式有以下两种：

- 方法一：
  选择“Maven > *样例工程名称* > Lifecycle > clean”，双击“clean”运行maven的clean命令。

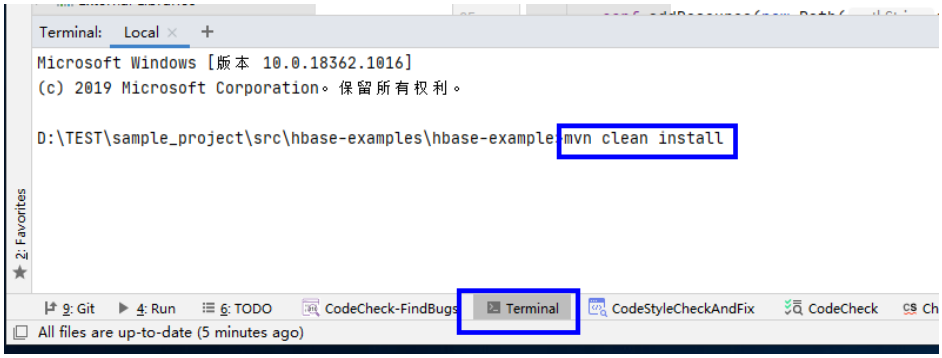
选择“Maven > *样例工程名称* > Lifecycle > install”，双击“install”运行maven的install命令。

图 11-24 maven 工具 clean 和 install



- 方法二：在IDEA的下方Terminal窗口进入“pom.xml”所在目录，手动输入**mvn clean install**命令进行编译。

图 11-25 idea termial 输入“mvn clean install”



编译完成，打印“BUILD SUCCESS”，生成target目录，生成jar包在target目录中。

步骤2 准备依赖的Jar包和配置文件。

1. 在Linux环境新建目录，例如“/opt/test”，并创建子目录“lib”和“conf”。将样例工程依赖的Jar包导出，导出步骤请参考**在Linux环境中调测HBase应用**章节

的**步骤2**，以及**步骤1**导出的Jar包，上传到Linux的“lib”目录。将**准备连接HBase集群配置文件**获取的配置文件及认证文件上传到Linux中“conf”目录。

2. 在“/opt/test”根目录新建脚本“run.sh”，修改内容如下并保存：

```
#!/bin/sh
BASEDIR=`cd $(dirname $0);pwd`
cd ${BASEDIR}
for file in ${BASEDIR}/lib/*.jar
do
i_cp=$i_cp:$file
echo "$file"
done
for file in ${BASEDIR}/conf/*
do
i_cp=$i_cp:$file
done

java -cp .${i_cp} com.huawei.bigdata.hbase.examples.TestMain
```

其中，*com.huawei.bigdata.hbase.examples.TestMain*为举例，具体以实际样例代码为准。

步骤3 切换到“/opt/test”，执行以下命令，运行Jar包。

sh run.sh

----结束

查看 Linux 调测结果

HBase应用程序运行完成后可通过如下方式查看应用程序的运行情况。

- 通过运行结果查看应用程序运行情况。
- 通过HBase日志获取应用程序运行情况。
- 登录HBase WebUI查看应用程序运行情况。可参见“更多信息 > 对外接口 > Web UI ”。
- 通过HBase shell命令查看应用程序运行情况。可参见“更多信息 > 对外接口 > Shell ”。

通过运行日志可查看应用提交后的执行详情，例如，hbase-example样例运行成功后，显示信息如下：

```
2280 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Entering testCreateTable.
3091 [main] WARNcom.huawei.hadoop.hbase.example.HBaseSample- table already exists
3091 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Exiting testCreateTable.
3091 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Entering testPut.
3264 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Put successfully.
3264 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Exiting testPut.
3264 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Entering testGet.
3283 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- 012005000201:info,address,Shenzhen,
Guangdong
3283 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- 012005000201:info,name,yugeZhang
San
3283 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Get data successfully.
3283 [main] INFOcom.huawei.hadoop.hbase.example.HBaseSample- Exiting testGet.
3283 [main] INFOorg.apache.hadoop.hbase.client.ConnectionManager$HConnectionImplementation-
Closing zookeeper sessionid=0xd000035eba278e9
3297 [main] INFOorg.apache.zookeeper.ZooKeeper- Session: 0xd000035eba278e9 closed
3297 [main-EventThread] INFOorg.apache.zookeeper.ClientCnxn- EventThread shut down
----------finish HBase ----------
```

11.5 HBase 应用开发常见问题

11.5.1 Phoenix SQL 查询样例介绍

功能简介

Phoenix是构建在HBase之上的一个SQL中间层，提供一个客户端可嵌入的JDBC驱动，Phoenix查询引擎将SQL输入转换为一个或多个HBase scan，编译并执行扫描任务以产生一个标准的JDBC结果集。

代码样例

- 客户端“hbase-example/conf/hbase-site.xml”中配置存放查询中间结果的临时目录，如果客户端程序在Linux上执行临时目录就配置Linux上的路径，如果客户端程序在Windows上执行临时目录则配Windows上的路径。

```
<property>
    <name>phoenix.spool.directory</name>
    <value>[1]查询中间结果的临时目录</value>
</property>
```

- JAVA样例：使用JDBC接口访问HBase

```
        public String getURL(Configuration conf)
        {
           String phoenix_jdbc = "jdbc:phoenix";
           String zkQuorum = conf.get("hbase.zookeeper.quorum");
           return phoenix_jdbc + ":" + zkQuorum;
        }

        public void testSQL()
        {
           String tableName = "TEST";
           // Create table
           String createTableSQL = "CREATE TABLE IF NOT EXISTS TEST(id integer not null primary key,
name varchar, account char(6), birth date)";

           // Delete table
           String dropTableSQL = "DROP TABLE TEST";

           // Insert
           String upsertSQL = "UPSERT INTO TEST VALUES(1,'John','100000',
TO_DATE('1980-01-01','yyyy-MM-dd'))";

           // Query
           String querySQL = "SELECT * FROM TEST WHERE id = ?";

           // Create the Configuration instance
           Configuration conf = getConfiguration();

           // Get URL
           String URL = getURL(conf);

           Connection conn = null;
           PreparedStatement preStat = null;
           Statement stat = null;
           ResultSet result = null;

           try
           {
              // Create Connection
              conn = DriverManager.getConnection(URL);
              // Create Statement
              stat = conn.createStatement();
              // Execute Create SQL
              stat.executeUpdate(createTableSQL);
              // Execute Update SQL
              stat.executeUpdate(upsertSQL);
              // Create PrepareStatement
              preStat = conn.prepareStatement(querySQL);
```

```
            conn.commit();
            // Execute query
            preStat.setInt(1,1);
            result = preStat.executeQuery();
            // Get result
            while (result.next())
            {
                int id = result.getInt("id");
                String name = result.getString(1);
            }
        }
        catch (Exception e)
        {
            // handler exception
        }
        finally
        {
            if(null != result){
                try {
                    result.close();
                    } catch (Exception e2) {
                        // handler exception
                        }
                }
            if(null != stat){
                try {
                    stat.close();
                } catch (Exception e2) {
                    // handler exception
                }
            }
            if(null != conn){
                try {
                    conn.close();
                } catch (Exception e2) {
                    // handler exception
                }
            }
        }
    }
```

注意事项

- 需要在“hbase-site.xml”中配置用于存放中间查询结果的临时目录路径，该目录大小限制可查询结果集大小；
- Phoenix实现了大部分java.sql接口，SQL紧跟ANSI SQL标准。

11.5.2 HBase 对外接口介绍

11.5.2.1 HBase Shell 接口介绍

您可以使用Shell在服务端直接对HBase进行操作。HBase的Shell接口同开源社区版本保持一致，请参见**http://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/**

Shell命令执行方法：

进入HBase客户端任意目录，执行以下命令。

hbase shell

进入HBase命令行运行模式（也称为CLI客户端连接），如下所示。

```
hbase(main):001:0>
```

您可以在命令行运行模式中运行***help***命令获取HBase的命令参数的帮助信息。

注意事项

count命令不支持条件统计，仅支持全表统计。

获取 HBase replication 指标的命令

通过Shell命令“status”可以获取到所有需要的指标。

- 查看replication source指标的命令。

  hbase(main):019:0> status 'replication', 'source'

  输出结果如下：

  ```
  version 2.2.3
  1 live servers
  BLR1000006595:
  SOURCE: PeerID=1, SizeOfLogQueue=0, ShippedBatches=0, ShippedOps=0, ShippedBytes=0,
  LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4, SizeOfLogToReplicate=0,
  TimeForLogToReplicate=0, ShippedHFiles=0,
  SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
  20:44:42 CST 2016, Replication Lag=0 PeerID=3, SizeOfLogQueue=0, ShippedBatches=0,
  ShippedOps=0, ShippedBytes=0, LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4,
  SizeOfLogToReplicate=0, TimeForLogToReplicate=0, ShippedHFiles=0,
  SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
  20:44:42 CST 2016, Replication Lag=0 FailedReplicationAttempts=0
  ```

- 查看replication sink指标的命令。

  hbase(main):020:0> status 'replication', 'sink'

  输出结果如下：

  ```
  version 2.2.3
  1 live servers
  BLR1000006595:
  SINK : AppliedBatches=0, AppliedOps=0, AppliedHFiles=0, AgeOfLastAppliedOp=0,
  TimeStampsOfLastAppliedOp=Wed May 25 17:55:21 CST 2016
  ```

- 同时查看replication source和replication sink指标的命令。

  hbase(main):018:0> status 'replication'

  输出结果如下：

  ```
  version 2.2.3
  1 live servers
  BLR1000006595:
  SOURCE: PeerID=1, SizeOfLogQueue=0, ShippedBatches=0, ShippedOps=0, ShippedBytes=0,
  LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4, SizeOfLogToReplicate=0,
  TimeForLogToReplicate=0, ShippedHFiles=0,
  SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
  20:43:24 CST 2016, Replication Lag=0 PeerID=3, SizeOfLogQueue=0, ShippedBatches=0,
  ShippedOps=0, ShippedBytes=0, LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4,
  SizeOfLogToReplicate=0, TimeForLogToReplicate=0, ShippedHFiles=0,
  SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
  20:43:24 CST 2016, Replication Lag=0 FailedReplicationAttempts=0
  SINK  : AppliedBatches=0, AppliedOps=0, AppliedHFiles=0, AgeOfLastAppliedOp=0,
  TimeStampsOfLastAppliedOp=Wed May 25 17:55:21 CST 2016
  ```

11.5.2.2 HBase Java API 接口介绍

接口使用建议

- 建议使用org.apache.hadoop.hbase.Cell作为KV数据对象，而不是org.apache.hadoop.hbase.KeyValue。

- 建议使用Connection connection = ConnectionFactory.createConnection(conf)来创建连接，废弃HTablePool。
- 建议使用org.apache.hadoop.hbase.mapreduce，不建议使用org.apache.hadoop.hbase.mapred。
- 建议通过构造出来的Connection对象的getAdmin()方法来获取HBase的客户端操作对象。

HBase 常用接口介绍

HBase常用的Java类有以下几个：

- 接口类Admin，HBase客户端应用的核心类，主要封装了HBase管理类操作的API，例如建表，删表等操作，部分常见接口参见表**表11-9**。
- 接口类Table，HBase读写操作类，主要封装了HBase表的读写操作的API，部分常见接口参见表**表11-10**。

表 11-9 org.apache.hadoop.hbase.client.Admin

方法	描述
boolean tableExists(final TableName tableName)	通过该方法可以判断指定的表是否存在。如果hbase:meta表中存在该表则返回true，否则返回false。
HTableDescriptor[] listTables(String regex)	查看匹配指定正则表达式格式的用户表。该方法还有另外两个重载的方法，一个入参类型为Pattern；一个入参为空，默认查看所有用户表。
HTableDescriptor[] listTables(final Pattern pattern, final boolean includeSysTables)	作用与上一个方法类似，用户可以通过该方法指定返回的结果是否包含系统表，上一个接口只返回用户表。
TableName[] listTableNames(String regex)	查看匹配指定正则表达式格式的用户表。该方法还有另外两个重载的方法，一个入参类型为Pattern；一个入参为空，默认查看所有用户表。该方法的作用与listTables类似，只是该方法返回类型为TableName[]。
TableName[] listTableNames(final Pattern pattern, final boolean includeSysTables)	作用与上一个方法类似，用户可以通过该方法指定返回的结果是否包含系统表，上一个方法只返回用户表。
void createTable(HTableDescriptor desc)	创建一个只有一个region的表。
void createTable(HTableDescriptor desc, byte[] startKey, byte[] endKey, int numRegions)	创建一个有指定数量region的表，其中第一个region的endKey为参数中的startKey，最后一个region的StartKey为参数中的endKey。如果region的数量过多，该方法可能调用超时。

方法	描述
void createTable(final HTableDescriptor desc, byte[][] splitKeys)	创建一个表，该表的region数量以及每个region的startKey由splitKeys决定。如果region的数量过多，该方法可能调用超时。
void createTable(final HTableDescriptor desc, final byte[][] splitKeys)	创建一个表，该表的region数量以及每个region的startKey由splitKeys决定。该方法为异步调用，不会等待创建的表上线。
void deleteTable(final TableName tableName)	删除指定的表。
public void truncateTable(final TableName tableName, final boolean preserveSplits)	重建指定的表，如果第二个参数为true，重建后的表region与之前保持一致，否则只有一个region。
void enableTable(final TableName tableName)	启用指定的表。如果表的region数量过多，该方法可能调用超时。
void enableTableAsync(final TableName tableName)	启用指定的表。该方法为异步调用，不会等待所有region上线后才返回。
void disableTable(final TableName tableName)	禁用指定的表。如果表的region数量过多，该方法可能调用超时。
void disableTableAsync(final TableName tableName)	禁用指定的表。该方法为异步调用，不会等待所有region下线后才返回。
boolean isTableEnabled(TableName tableName)	判断表是否已启用。该方法可以配合enableTableAsync方法使用来判断一个启用表的操作是否完成。
boolean isTableDisabled(TableName tableName)	判断表是否已禁用。该方法可以配合disableTableAsync方法使用来判断一个禁用表的操作是否完成。
void addColumn(final TableName tableName, final HColumnDescriptor column)	添加一个列簇到指定的表。
void deleteColumn(final TableName tableName, final HColumnDescriptor column)	从指定的表删除指定的列簇。
void modifyColumn(final TableName tableName, final HColumnDescriptor column)	修改指定的列簇。

表 11-10 org.apache.hadoop.hbase.client.Table

方法	描述
boolean exists(Get get)	判断指定的rowkey在表中是否存在。
boolean[] existsAll(List<Get> gets)	判断这批指定的rowkey在表中是否存在，返回的boolean数组结果与入参位置一一对应。
Result get(Get get)	通过指定的rowkey读取数据。
Result[] get(List<Get> gets)	通过指定一批rowkey的方式批量读取数据。
ResultScanner getScanner(Scan scan)	获取该表的一个Scanner对象，查询相关的参数可以通过入参scan指定，包括StartRow，StopRow，caching大小等。
void put(Put put)	往该表写入一条数据。
void put(List<Put> puts)	往该表写入一批数据。
void close()	释放该Table对象持有的资源。

说明

表11-9和表11-10只列举了部分常用的方法。

11.5.2.3 Sqlline 接口介绍

你可以直接使用sqlline.py在服务端对HBase进行SQL操作。Phoenix的sqlline接口与开源社区保持一致，请参见**http://phoenix.apache.org/**。

Sqlline常用语法见**表11-11**，常用函数见**表11-12**，命令行使用可以参考**Phoenix命令行操作介绍**章节。

表 11-11 Sqlline 常用语法

命令	描述	示例
CREATE TABLE	创建表。	CREATE TABLE MY_TABLE(id BIGINT not null primary key, name VARCHAR);
ALTER	修改表/视图。	ALTER TABLE MY_TABLE DROP COLUMN name;
DROP TABLE	删除表。	DROP TABLE MY_TABLE;
UPSERT VALUES	插入/修改数据。	UPSERT INTO MY_TABLE VALUES(1,'abc');
SELECT	查询数据。	SELECT * FROM MY_TABLE;

命令	描述	示例
CREATE INDEX	创建全局索引。	CREATE INDEX MY_IDX ON MY_TABLE(name);
CREATE LOCAL INDEX	创建局部索引。	CREATE LOCAL INDEX MY_LOCAL_IDX ON MY_TABLE(id,name);
ALTER INDEX	修改索引状态。	ALTER INDEX MY_IDX ON MY_TABLE DISABLE;
DROP INDEX	删除索引。	DROP INDEX MY_IDX ON MY_TABLE;
EXPLAIN	显示执行计划。	EXPLAIN SELECT name FROM MY_TABLE;
CREATE SEQUENCE	创建序列。	CREATE SEQUENCE MY_SEQUENCE;
DROP SEQUENCE	删除序列。	DROP SEQUENCE MY_SEQUENCE;
CREATE VIEW	创建视图。	CREATE VIEW MY_VIEW AS SELECT * FROM MY_TABLE;
DROP VIEW	删除视图。	DROP VIEW MY_VIEW;
CREATE SCHEMA	创建SCHEMA。	CREATE SCHEMA MY_SCHEMA;
USE	修改默认SCHEMA。	USE MY_SCHEMA;
DROP SCHEMA	删除SCHEMA。	DROP SCHEMA MY_SCHEMA;

表 11-12 Sqlline 常用函数

函数类型	函数	描述	示例
字符串	**SUBSTR**	字符串截取。	SUBSTR('[Hello]', 2, 5)
	INSTR	查询子串位置。	INSTR('Hello World', 'World')
	TRIM	去除前后空格。	TRIM(' Hello ')
	LTRIM	去除左空格。	LTRIM(' Hello')
	RTRIM	去除右空格。	RTRIM('Hello ')
	LPAD	字符串左侧填充。	LPAD('John',30,'*')
	LENGTH	获取字符串长度。	LENGTH('Hello')
	UPPER	字符串转大写。	UPPER('Hello')
	LOWER	字符串转小写。	LOWER('HELLO')

函数类型	函数	描述	示例
	REVERSE	字符串反转。	REVERSE('Hello')
	REGEXP_SPLIT	字符串分割。	REGEXP_SPLIT('ONE,TWO,THREE', ',')
	REGEXP_REPLACE	字符串替换。	REGEXP_REPLACE('abc123ABC', '[0-9]+', '#')
	REGEXP_SUBSTR	获取正则子串。	REGEXP_SUBSTR('na1-appsrv35-sj35', '[^-]+')
聚合	**AVG**	获取平均数。	AVG(X)
	COUNT	获取数据条数。	COUNT(*)
	MAX	获取最大值。	MAX(NAME)
	MIN	获取最小值。	MIN(NAME)
	SUM	数字求合。	SUM(X)
	STDDEV_POP	标准差。	STDDEV_POP(X)
	STDDEV_SAMP	样板标准差。	STDDEV_SAMP(X)
	NTH_VALUE	分组后的第几个值。	NTH_VALUE(name, 2) WITHIN GROUP (ORDER BY salary DESC)
时间	**NOW**	获取当前时间(DATE类型)。	NOW()
	CURRENT_TIME	获取当前时间(TIME类型)。	CURRENT_TIME()
	CURRENT_DATE	获取当前时间(DATE类型)。	CURRENT_DATE()
	TO_DATE	字符串转DATE类型。	TO_DATE('1970-01-01', 'yyyy-MM-dd', 'GMT+1')
	TO_TIME	字符串转TIME类型。	TO_TIME('1970-01-01', 'yyyy-MM-dd', 'GMT+1')
	TO_TIMESTAMP	字符串转TIMESTAMP类型。	TO_TIMESTAMP('1970-01-01', 'yyyy-MM-dd', 'GMT+1')
	YEAR	获取年。	YEAR(TO_DATE('2015-6-05'))
	MONTH	获取月。	MONTH(TO_TIMESTAMP('2015-6-05'))
	WEEK	获取星期。	WEEK(TO_TIME('2010-6-15'))
	HOUR	获取小时。	HOUR(TO_TIMESTAMP('2015-6-05'))

函数类型	函数	描述	示例
	MINUTE	获取分钟。	MINUTE(TO_TIME('2015-6-05'))
	SECOND	获取秒。	SECOND(TO_DATE('2015-6-05'))
数字	**ROUND**	四舍五入（也可用于时间）。	ROUND(2.56)
	CEIL	向上取整。	CEIL(2.34)
	FLOOR	向下取整。	FLOOR(2.34)
	TRUNC	向下取整（与**FLOOR**相同）。	TRUNC(2.34)
	TO_NUMBER	字符串转数字。	TO_NUMBER('-123.33')
	RAND	获取随机数。	RAND()
数学	**ABS**	求绝对值。	ABS(-1)
	SQRT	$\sqrt{x}$	SQRT(1.1)
	EXP	e^x	EXP(-1)
	POWER	x^y	POWER(2, 3)
	LN	$ln(x)$	LN(3)
	LOG	$log_x(y)$	LOG(2, 3)
数组	**ARRAY_ELEM**	通过下标访问数组。	ARRAY_ELEM(ARRAY[1,2,3], 1)
	ARRAY_PREPEND	指定位置插入数据到数组。	ARRAY_APPEND(ARRAY[1,2,3], 4)
	ARRAY_CAT	连接数组。	ARRAY_CAT(ARRAY[1,2], ARRAY[3,4])
	ARRAY_FILL	数组填充。	ARRAY_FILL(1, 3)
	ARRAY_TO_STRING	数组转字符串。	ARRAY_TO_STRING(ARRAY['a','b','c'], ',')
	ANY	数组任意值满足条件。	10 > ANY(my_array)

函数类型	函数	描述	示例
	ALL	数组所有值满足条件。	10 > ALL(my_array)
其他	**MD5**	获取MD5码。	MD5(my_column)
	ENCODE	编码字符串。	ENCODE(myNumber, 'BASE62')
	DECODE	解码字符串。	DECODE('000000008512af277ffffff8', 'HEX')

11.5.2.4 HBase JDBC API 接口介绍

Phoenix实现了大部分的java.sql接口，SQL语法紧跟ANSI SQL标准。

其支持处理函数可参见：

http://phoenix.apache.org/language/functions.html

其支持语法可参见：

http://phoenix.apache.org/language/index.html

11.5.2.5 HBase Web UI 接口介绍

操作场景

Web UI展示了HBase集群的状态，其中包括整个集群概况信息、RegionServer和Master的信息、快照、运行进程等信息。通过Web UI提供的信息可以对整个HBase集群的状况有一定的了解。

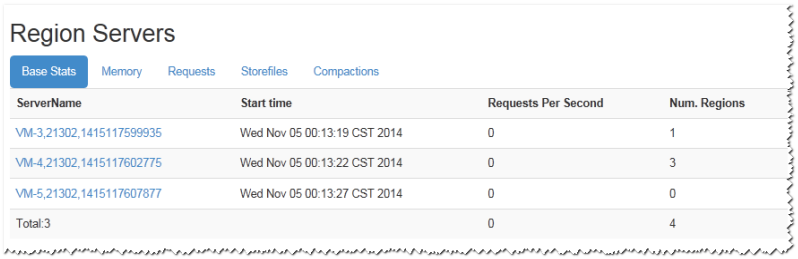
说明

请联系管理员获取具有访问Web UI权限的业务账号及其密码。

操作步骤

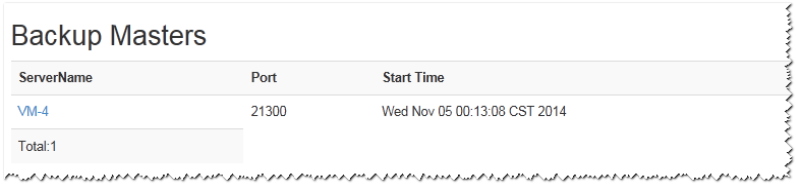
1. **登录FusionInsight Manager页面**。选择“集群 > *待操作集群的名称* > 服务 > HBase > HMaster(主)”打开HBase的Web UI。
2. 在HBase的Web UI页面中，Home页面展示的是HBase的一些概况信息，具体包括以下信息：
   a. Region Servers页面展示了RegionServer的一些基本信息，如**图11-26**所示。

图 11-26 Region Servers 基本信息



b. Backup Master页面展示了Backup Master的信息，如图11-27所示。

图 11-27 Backup Masters 基本信息

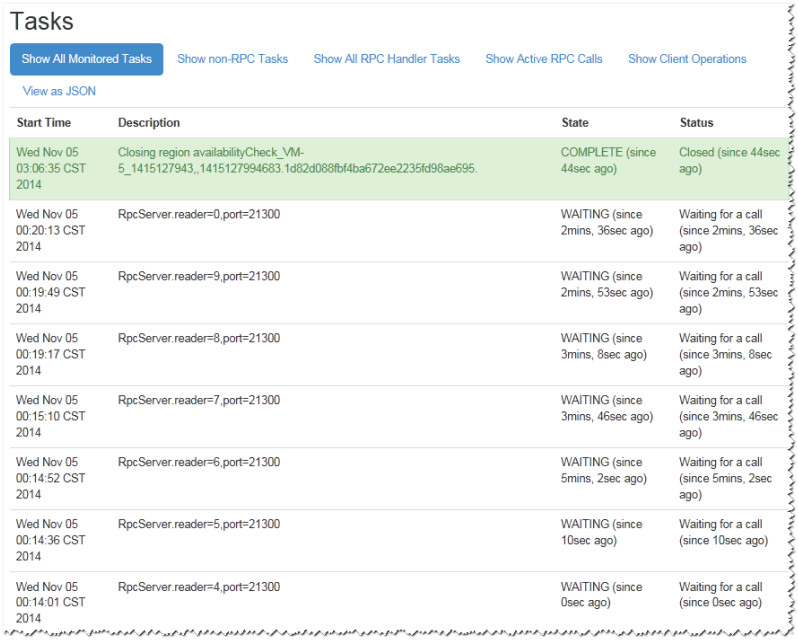


c. Tables页面显示了HBase中表的信息，包括User Tables、Catalog Tables、Snapshots，如图11-28所示。

图 11-28 Tables 基本信息

d. Tasks页面显示了运行在HBase上的任务信息，包括开始时间，状态等信息，如图11-29所示。

图 11-29 Tasks 基本信息



3. 在HBase的Web UI页面中，Table Details页面展示的是HBase存储表的概要信息，如图11-30所示。

图 11-30 TableDetails

4. 在HBase的Web UI页面中，Debug dump页面展示的是HBase的Debug信息，如图11-31所示。

图 11-31 Debug dump

5. 在HBase的Web UI页面中，HBaseConfiguration页面展示的是HBase的控制信息，如图11-32所示。

图 11-32 HBase Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
- <configuration>
  - <property>
      <name>dfs.journalnode.rpc-address</name>
      <value>0.0.0.0:8485</value>
      <source>hdfs-default.xml</source>
    </property>
  - <property>
      <name>io.storefile.bloom.block.size</name>
      <value>131072</value>
      <source>hbase-default.xml</source>
    </property>
  - <property>
      <name>hbase.sessioncontrol.maxSessions</name>
      <value>65535</value>
      <source>hbase-site.xml</source>
    </property>
  - <property>
      <name>yarn.ipc.rpc.class</name>
      <value>org.apache.hadoop.yarn.ipc.HadoopYarnProtoRPC</value>
      <source>yarn-default.xml</source>
    </property>
  - <property>
      <name>mapreduce.job.maxtaskfailures.per.tracker</name>
      <value>3</value>
      <source>mapred-default.xml</source>
    </property>
  - <property>
      <name>hbase.rest.threads.min</name>
      <value>2</value>
      <source>hbase-default.xml</source>
    </property>
  - <property>
      <name>hbase.rs.cacheblocksonwrite</name>
      <value>false</value>
      <source>hbase-default.xml</source>
    </property>
  - <property>
      <name>ha.health-monitor.connect-retry-interval.ms</name>
      <value>1000</value>
      <source>core-default.xml</source>
    </property>
  - <property>
      <name>yarn.resourcemanager.work-preserving-recovery.enabled</name>
      <value>false</value>
      <source>yarn-default.xml</source>
    </property>
  - <property>
      <name>dfs.client.mmap.cache.size</name>
      <value>256</value>
      <source>hdfs-default.xml</source>
    </property>
```

11.5.3 Phoenix 命令行操作介绍

Phoenix支持SQL的方式来操作HBase，以下简单介绍使用SQL语句建表/插入数据/查询数据/删表等操作。

前提条件

已安装HBase客户端，例如安装目录为“/opt/client”。以下操作的客户端目录只是举例，请根据实际安装目录修改。在使用客户端前，需要先下载并更新客户端配置文件，确认Manager的主管理节点后才能使用客户端。

操作步骤

步骤1 以客户端安装用户，登录安装HBase客户端的节点。

进入HBase客户端安装目录：

例如：**cd /opt/client**

步骤2 执行以下命令配置环境变量。

source bigdata_env

步骤3 如果当前集群已启用Kerberos认证，执行以下命令认证当前用户，当前用户需要具有创建HBase表的权限，具体请参见**创建角色**配置拥有对应权限的角色，参考**创建用户**为用户绑定对应角色。如果当前集群未启用Kerberos认证，则无需执行此命令。

kinit MRS 集群用户

例如，**kinit hbaseuser**。

步骤4 直接执行Phoenix客户端命令。

sqlline.py

步骤5 建表：

```
CREATE TABLE TEST (id VARCHAR PRIMARY KEY, name VARCHAR);
```

步骤6 插入数据：

```
UPSERT INTO TEST(id,name) VALUES ('1','jamee');
```

步骤7 查询数据：

```
SELECT * FROM TEST;
```

步骤8 删表：

```
DROP TABLE TEST;
```

步骤9 退出Phoenix命令行。

!quit

----结束

11.5.4 如何配置 HBase 双读功能

操作场景

HBase客户端应用通过自定义加载主备集群配置项，实现了双读能力。HBase双读作为提高HBase集群系统高可用性的一个关键特性，适用于四个查询场景：使用**Get**读取数据、使用批量**Get**读取数据、使用**Scan**读取数据，以及基于二级索引查询。它能够同时读取主备集群数据，减少查询毛刺，具体表现为：

- 高成功率：双并发读机制，保证每一次读请求的成功率。
- 可用性：单集群故障时，查询业务不中断。短暂的网络抖动也不会导致查询时间变长。
- 通用性：双读特性不支持双写，但不影响原有的实时写场景。
- 易用性：客户端封装处理，业务侧不感知。

说明

HBase双读使用约束：

- HBase双读特性基于Replication实现，备集群读取的数据可能和主集群存在差异，因此只能实现最终一致性。
- 目前HBase双读功能仅用于查询。主集群宕机时，最新数据无法同步，备集群可能查询不到最新数据。
- HBase的**Scan**操作可能分解为多次RPC。由于相关session信息在不同集群间不同步，数据不能保证完全一致，因此双读只在第一次RPC时生效，ResultScanner close之前的请求会固定访问第一次RPC时使用的集群。
- HBase Admin接口、实时写入接口只会访问主集群。所以主集群宕机后，不能提供Admin接口功能和实时写入接口功能，只能提供**Get**、**Scan**查询服务。

操作步骤

步骤1 将在**准备集群认证用户信息**时获取的主集群keytab认证文件“user.keytab”与“krb5.conf”放置到二次样例“src/main/resources/conf”目录下。

步骤2 参考**准备连接HBase集群配置文件**章节，获取HBase主集群客户端配置文件“core-site.xml”、“hbase-site.xml”、“hdfs-site.xml”，并将其放置到“src/main/resources/conf/active”目录下，该目录需要自己创建。

步骤3 参考**准备连接HBase集群配置文件**章节，获取备集群客户端配置文件“core-site.xml”、“hbase-site.xml”、“hdfs-site.xml”，并将其放置到“src/main/resources/conf/standby”目录下，该目录需要自己创建。

步骤4 创建“hbase-dual.xml”配置文件，放置到“src/main/resources/conf/”目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<!--主集群配置文件目录-->
   <property>
      <name>hbase.dualclient.active.cluster.configuration.path</name>
      <value>{样例代码目录}\\src\\main\\resources\\conf\\active</value>
      </property>
<!--备集群配置文件目录-->
   <property>
      <name>hbase.dualclient.standby.cluster.configuration.path</name>
      <value>{样例代码目录}\\src\\main\\resources\\conf\\standby</value>
   </property>
<!--双读模式的Connection实现-->
   <property>
      <name>hbase.client.connection.impl</name>
      <value>org.apache.hadoop.hbase.client.HBaseMultiClusterConnectionImpl</value>
   </property>
<!--安全模式-->
   <property>
      <name>hbase.security.authentication</name>
      <value>kerberos</value>
   </property>
<!--安全模式-->
   <property>
      <name>hadoop.security.authentication</name>
      <value>kerberos</value>
   </property>
```

----结束

代码样例

- 创建双读Configuration,下面代码片段在“com.huawei.bigdata.hbase.examples”包的“TestMain”类的**init**方法中添加。

```
private static void init() throws IOException {
    // Default load from conf directory
    conf = HBaseConfiguration.create();
    //In Windows environment
    String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
    //In Linux environment
    //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
    conf.addResource(new Path(userdir + "hbase-dual.xml"), false);
  }
```

- 确定数据来源的集群
  - GET请求，以下代码片段在“com.huawei.bigdata.hbase.examples”包的“HBaseSample”类的**testGet**方法中添加。

    ```
    Result result = table.get(get);
    if (result instanceof DualResult) {
        LOG.info(((DualResult)result).getClusterId());
    }
    ```

  - Scan请求，以下代码片段在“com.huawei.bigdata.hbase.examples”包的“HBaseSample”类的**testScanData**方法中添加。

    ```
    ResultScanner rScanner = table.getScanner(scan);
    if (rScanner instanceof HBaseMultiScanner) {
        LOG.info(((HBaseMultiScanner)rScanner).getClusterId());
    }
    ```

- 客户端支持打印metric信息

  “log4j.properties”文件中增加如下内容，客户端将metric信息输出到指定文件。

```
log4j.logger.DUAL=debug,DUAL
log4j.appender.DUAL=org.apache.log4j.RollingFileAppender
log4j.appender.DUAL.File=/var/log/dual.log //客户端本地双读日志路径，根据实际路径修改，但目录要有写入权限
log4j.additivity.DUAL=false
log4j.appender.DUAL.MaxFileSize=${hbase.log.maxfilesize}
log4j.appender.DUAL.MaxBackupIndex=${hbase.log.maxbackupindex}
log4j.appender.DUAL.layout=org.apache.log4j.PatternLayout
log4j.appender.DUAL.layout.ConversionPattern=%d{ISO8601} %-5p [%t] %c{2}: %m%n
```

HBase 双读操作相关配置项说明

表 11-13 hbase-dual.xml 配置项

配置项名称	配置项详解	默认值	级别
hbase.dualclient.active.cluster.configuration.path	主集群HBase客户端配置目录	无	必选配置
hbase.dualclient.standby.cluster.configuration.path	备集群HBase客户端配置目录	无	必选配置
dual.client.schedule.update.table.delay.second	更新开启容灾表列表的周期时间	5	可选配置

配置项名称	配置项详解	默认值	级别
hbase.dualclient.glitchtimeout.ms	可以容忍主集群的最大毛刺时间	50	可选配置
hbase.dualclient.slow.query.timeout.ms	慢查询告警日志	180000	可选配置
hbase.dualclient.active.cluster.id	主集群id	ACTIVE	可选配置
hbase.dualclient.standby.cluster.id	备集群id	STANDBY	可选配置
hbase.dualclient.active.executor.thread.max	请求主集群的线程池max大小	100	可选配置
hbase.dualclient.active.executor.thread.core	请求主集群的线程池core大小	100	可选配置
hbase.dualclient.active.executor.queue	请求主集群的线程池queue大小	256	可选配置
hbase.dualclient.standby.executor.thread.max	请求备集群的线程池max大小	100	可选配置
hbase.dualclient.standby.executor.thread.core	请求备集群的线程池core大小	100	可选配置
hbase.dualclient.standby.executor.queue	请求备集群的线程池queue大小	256	可选配置
hbase.dualclient.clear.executor.thread.max	清理资源线程池max大小	30	可选配置
hbase.dualclient.clear.executor.thread.core	清理资源线程池core大小	30	可选配置
hbase.dualclient.clear.executor.queue	清理资源线程池queue大小	Integer.MAX_VALUE	可选配置
dual.client.metrics.enable	客户端metric信息是否打印	true	可选配置
dual.client.schedule.metrics.second	客户端metric信息打印周期	300	可选配置

配置项名称	配置项详解	默认值	级别
dual.client.asynchronous.enable	是否异步请求主备集群	false	可选配置

打印 metric 信息

表 11-14 基本指标项

Metric名称	描述	日志级别
total_request_count	周期时间内查询总次数	INFO
active_success_count	周期时间内主集群查询成功次数	INFO
active_error_count	周期时间内主集群查询失败次数	INFO
active_timeout_count	周期时间内主集群查询超时次数	INFO
standby_success_count	周期时间内备集群查询成功次数	INFO
standby_error_count	周期时间内备集群查询失败次数	INFO
Active Thread pool	周期打印请求主集群的执行线程池信息	DEBUG
Standby Thread pool	周期打印请求备集群的执行线程池信息	DEBUG
Clear Thread pool	周期打印释放资源的执行线程池信息	DEBUG

表 11-15 针对 GET、BatchGET、SCAN 请求，分别打印 Histogram 指标项

Metric名称	描述	日志级别
averageLatency(ms)	平均时延	INFO
minLatency(ms)	最小时延	INFO
maxLatency(ms)	最大时延	INFO
95thPercentileLatency(ms)	95%请求的最大时延	INFO
99thPercentileLatency(ms)	99%请求的最大时延	INFO

Metric名称	描述	日志级别
99.9PercentileLatency(ms)	99.9%请求的最大时延	INFO
99.99PercentileLatency(ms)	99.99%请求的最大时延	INFO

11.5.5 配置 Windows 通过 EIP 访问安全模式集群 HBase

操作场景

该章节通过指导用户配置集群绑定EIP，并配置HBase文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行hbase-example中的样例为例进行说明。

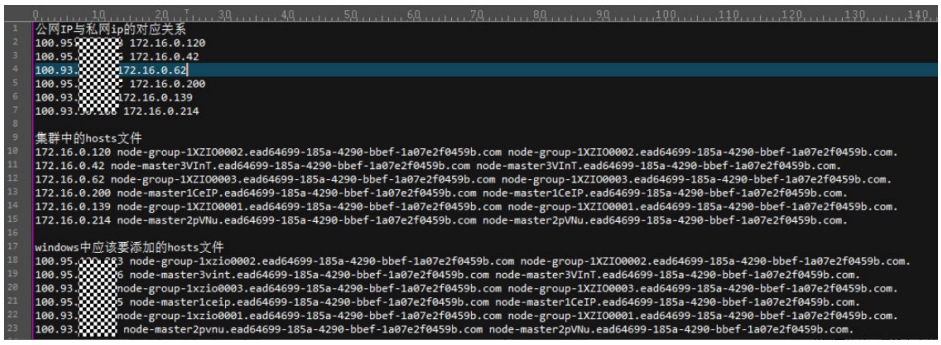
操作步骤

步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。



步骤2 将krb5.conf文件中的IP地址修改为对应IP的主机名称。

步骤3 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和21730TCP、21731TCP/UDP、21732TCP/UDP端口。

步骤4 在Manager界面选择“集群 > 服务 > HBase > 更多 > 下载客户端”，将客户端中的“core-site.xml”、“hdfs-site.xml”和“hbase-site.xml”复制到样例工程的resources目录下。

步骤5 在运行样例代码前，需要将样例代码中的userName改为安全认证的用户名。

----结束

11.5.6 运行 HBase 应用开发程序产生 ServerRpcControllerFactory 异常

步骤1 检查应用开发工程的配置文件hbase-site.xml中是否包含配置项hbase.rpc.controllerfactory.class。

```
<name>hbase.rpc.controllerfactory.class</name>
<value>org.apache.hadoop.hbase.ipc.controller.ServerRpcControllerFactory</value>
```

步骤2 如果当前的应用开发工程配置项中包含该配置项，则应用开发程序还需要引入Jar包“phoenix-core-*.jar”。此Jar包可以从HBase客户端安装目录下的“HBase/hbase/lib”获取。

步骤3 如果不想引入该Jar包，请将应用开发工程的配置文件“hbase-site.xml”中的配置“hbase.rpc.controllerfactory.class”删除掉。

----结束

11.5.7 BulkLoad 和 Put 应用场景有哪些

问题

HBase支持使用bulkload和put方式加载数据，在大部分场景下bulkload提供了更快的数据加载速度，但bulkload并不是没有缺点的，在使用时需要关注bulkload和put适合在哪些场景使用。

回答

bulkload是通过启动MapReduce任务直接生成HFile文件，再将HFile文件注册到HBase，因此错误的使用bulkload会因为启动MapReduce任务而占用更多的集群内存和CPU资源，也可能会生成大量很小的HFile文件频繁的触发Compaction，导致查询速度急剧下降。

错误的使用put，会造成数据加载慢，当分配给RegionServer内存不足时会造成RegionServer内存溢出从而导致进程退出。

下面给出bulkload和put适合的场景：

- bulkload适合的场景：
  - 大量数据一次性加载到HBase。
  - 对数据加载到HBase可靠性要求不高，不需要生成WAL文件。
  - 使用put加载大量数据到HBase速度变慢，且查询速度变慢时。
  - 加载到HBase新生成的单个HFile文件大小接近HDFS block大小。
- put适合的场景：
  - 每次加载到单个Region的数据大小小于HDFS block大小的一半。
  - 数据需要实时加载。
  - 加载数据过程不会造成用户查询速度急剧下降。

11.5.8 install 编译构建 HBase Jar 包失败报错 Could not transfer artifact 如何处理

问题

样例代码在进行maven编译构建jar包时，Build Failed，提示错误信息：Could not transfer artifact org.apache.commons:commons-crypto:pom:${commons-crypto.version}

回答

hbase-common模块依赖commons-crypto，在hbase-common的pom.xml文件中，对于commons-crypto的引入，<version>使用了${commons-crypto.version}变量。该变量的解析逻辑为，os为aarch64时值为“**1.0.0-hw-aarch64**”，os为x86_64时值为“**1.0.0**”。编译环境因为一些配置原因导致maven未能通过os正确解析该变量时，可采用手动修改pom.xml方式进行规避正确编译。

在pom.xml中手动修改直接或间接依赖hbase-common模块的dependency，修改为如下形式，排除commons-crypto依赖。

```
<dependency>
   <groupId>org.apache.hbase</groupId>
   <artifactId>hbase-common</artifactId>
   <version>${hbase.version}</version>
   <exclusions>
      <exclusion>
         <groupId>org.apache.commons</groupId>
         <artifactId>commons-crypto</artifactId>
      </exclusion>
   </exclusions>
</dependency>
```

再手动添加指定版本的commons-crypto依赖。根据os架构为x86_64或aarch64填写正确version。

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-crypto</artifactId>
    <version>1.0.0</version>
</dependency>
```

12 HBase 开发指南（普通模式）

12.1 HBase 应用开发概述

12.1.1 HBase 应用开发简介

HBase 简介

HBase是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统。HBase设计目标是解决关系型数据库在处理海量数据时的局限性。

HBase使用场景有如下几个特点：

- 处理海量数据（TB或PB级别以上）。
- 具有高吞吐量。
- 在海量数据中实现高效的随机读取。
- 具有很好的伸缩能力。
- 能够同时处理结构化和非结构化的数据。
- 不需要完全拥有传统关系型数据库所具备的ACID特性。ACID特性指原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。
- HBase中的表具有如下特点：
  - 大：一个表可以有上亿行，上百万列。
  - 面向列：面向列（族）的存储和权限控制，列（族）独立检索。
  - 稀疏：对于为空（null）的列，并不占用存储空间，因此，表可以设计的非常稀疏。

接口类型简介

由于HBase本身是由java语言开发出来的，且java语言具有简洁通用易懂的特性，推荐用户使用java语言进行HBase应用程序开发。

HBase采用的接口与Apache HBase保持一致。

HBase通过接口调用，可提供的功能如**表12-1**所示。

表 12-1 HBase 接口提供的功能

功能	说明
CRUD数据读写功能	增查改删。
高级特性	过滤器、二级索引，协处理器。
管理功能	表管理、集群管理。

12.1.2 HBase 应用开发常用概念

- **过滤器**

  过滤器用于帮助用户提高HBase处理表中数据的效率。用户不仅可以使用HBase中预定义好的过滤器，而且可以实现自定义的过滤器。

- **协处理器**

  允许用户执行region级的操作，并且可以使用与RDBMS中触发器类似的功能。

- **Client**

  客户端直接面向用户，可通过Java API、HBase Shell或者Web UI访问服务端，对HBase的表进行读写操作。本文中的HBase客户端特指HBase client的安装包，可参考**HBase对外接口介绍**。

12.1.3 HBase 应用开发流程

本文档主要基于Java API对HBase进行应用开发。

开发流程中各阶段的说明如**图12-1**和**表12-2**所示。

图 12-1 HBase 应用程序开发流程

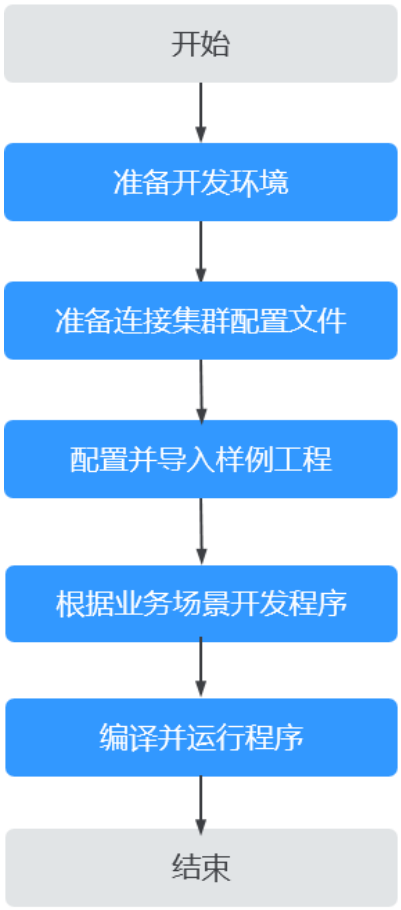


表 12-2 HBase 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，推荐使用Java语言进行开发，使用IntelliJ IDEA工具，同时完成JDK、Maven等初始配置。	**准备本地应用开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备连接HBase集群配置文件**

阶段	说明	参考文档
配置并导入样例工程	HBase提供了不同场景下的多种样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。	导入并配置HBase样例工程
根据业务场景开发程序	根据实际业务场景开发程序，调用组件接口实现对应功能。	开发HBase应用
编译并运行程序	开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。	调测HBase应用

12.1.4 HBase 应用开发样例工程介绍

MRS样例工程获取地址为https://github.com/huaweicloud/huaweicloud-mrs-example，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下HBase相关样例工程：

表 12-3 HBase 相关样例工程

样例工程位置	描述
hbase-examples/hbase-example	HBase数据读写操作的应用开发示例。 通过调用HBase接口可实现创建用户表、导入用户数据、增加用户信息、查询用户信息及为用户表创建二级索引等功能，相关业务场景介绍请参见HBase样例程序开发思路。
hbase-examples/hbase-rest-example	HBase Rest接口应用开发示例。 使用Rest接口实现查询HBase集群信息、获取表、操作NameSpace、操作表等功能，相关样例介绍请参见HBase Rest接口调用样例程序。
hbase-examples/hbase-thrift-example	访问HBase ThriftServer应用开发示例。 访问ThriftServer操作表、向表中写数据、从表中读数据，相关样例介绍请参见HBase ThriftServer连接样例程序。
hbase-examples/hbase-zk-example	HBase访问ZooKeeper应用开发示例。 在同一个客户端进程内同时访问MRS ZooKeeper和第三方的ZooKeeper，其中HBase客户端访问MRS ZooKeeper，客户应用访问第三方ZooKeeper。

12.2 准备 HBase 应用开发环境

12.2.1 准备本地应用开发环境

在进行二次开发时，要准备的开发和运行环境如表12-4所示。

表 12-4 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置，版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls。
安装和配置IntelliJ IDEA	用于开发HBase应用程序的工具。版本要求：2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。
安装Junit插件	开发环境的基本配置。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。 华为提供开源镜像站，各服务样例工程依赖的Jar包通过华为开源镜像站下载，剩余所依赖的开源Jar包请直接从Maven中央库或者其他用户自定义的仓库地址下载，详情请参考配置华为开源镜像仓。

准备项	说明
7-zip	用于解压“*.zip”和“*.rar”文件。 支持7-Zip 16.04版本。

12.2.2 准备连接 HBase 集群配置文件

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。
  a. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**），单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

     例如，客户端配置文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。
  b. 进入客户端配置文件解压路径的“HBase\config”，获取HBase**表12-5**中相关配置文件。

     表 12-5 配置文件

配置文件	作用
core-site.xml	配置Hadoop Core详细参数。
hbase-site.xml	配置HBase详细参数。
hdfs-site.xml	配置HDFS详细参数。

     说明

     访问HBase ThriftServer应用开发示例工程所需的配置文件还需参考**准备ThriftServer实例配置文件**获取。
  c. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群，具体操作请参考**配置Windows通过EIP访问普通模式集群HBase**。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。

  a. 在节点中安装MRS集群客户端。

  例如客户端安装目录为“/opt/client”。

  b. 获取配置文件：

  i. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**），单击“确定”，下载客户端配置文件至集群主OMS点。

  ii. 以**root**登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“HBase/config”路径下的**表12-5**中相关配置文件。

  例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

  cd /tmp/FusionInsight-Client

  tar -xvf FusionInsight_Cluster_1_Services_Client.tar

  tar -xvf FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar

  cd FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles

  说明

  访问HBase ThriftServer应用开发示例工程所需的配置文件还需参考**准备ThriftServer实例配置文件**获取。

  c. 检查客户端节点网络连接。

  在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

准备 ThriftServer 实例配置文件

若需实现访问HBase ThriftServer并进行表相关操作，则需执行以下步骤获取相关配置文件。

步骤1 登录FusionInsight Manager，选择“集群 > 服务 > HBase > 配置 > 全部配置”，搜索并修改ThriftServer实例的配置参数“hbase.thrift.security.qop”。该参数值需与“hbase.rpc.protection”的值一一对应。保存配置，重启配置过期节点服务使更改的配置生效。

说明

“hbase.rpc.protection”与“hbase.thrift.security.qop”参数值的对应关系为：

- **"privacy" - "auth-conf"**
- **"authentication" - "auth"**
- **"integrity" - "auth-int"**

步骤2 获取ThriftServer实例配置文件：

- 方法一：选择“集群 > 服务 > HBase > 实例”，单击待操作的ThriftServer实例，在“概览”界面的“配置文件”区域分别单击配置文件“hdfs-site.xml”、“core-site.xml”、“hbase-site.xml”名称，获取配置文件。
- 方法二：通过**准备运行环境配置文件**中解压客户端文件的方法获取配置文件，需要在获取的“hbase-site.xml”中手动添加以下配置，其中“hbase.thrift.security.qop”的参数值与**步骤1**保持一致。

```
<property>
<name>hbase.thrift.security.qop</name>
<value>auth</value>
</property>
<property>
<name>hbase.thrift.kerberos.principal</name>
<value>thrift/hadoop.hadoop.com@HADOOP.COM</value>
</property>
<property>
<name>hbase.thrift.keytab.file</name><value>/opt/huawei/Bigdata/FusionInsight_HD_8.1.0.1/install/FusionInsight-HBase-2.2.3/keytabs/HBase/thrift.keytab</value>
```

----结束

12.2.3 导入并配置 HBase 样例工程

背景信息

获取HBase开发样例工程，将工程导入到IntelliJ IDEA开始样例学习。

前提条件

- 确保本地PC的时间与集群的时间差要小于5分钟，若无法确定，请联系系统管理员。集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备连接HBase集群配置文件**。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src/hbase-examples”目录下的样例工程文件夹“hbase-example”，

可根据实际业务场景选择对应的样例，相关样例介绍请参见**HBase应用开发样例工程介绍**。

步骤2 若需要在本地Windows调测HBase样例代码，需参考**表12-6**放置各样例项目所需的配置文件：

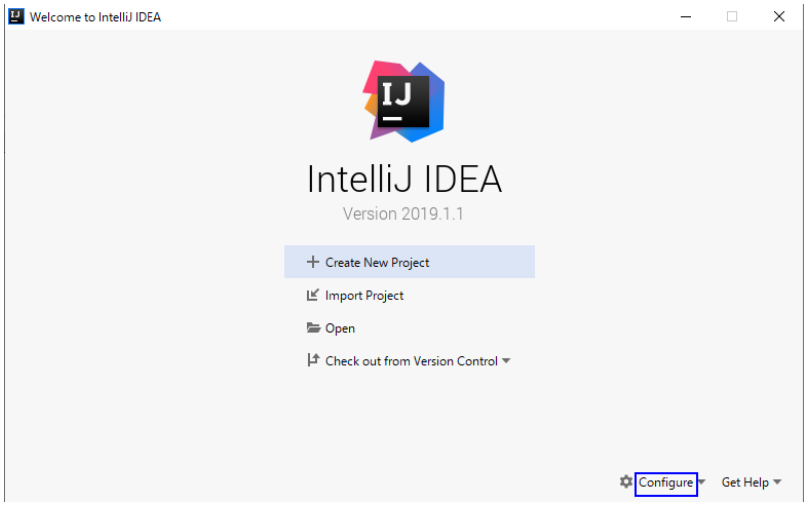
表 12-6 放置各样例项目所需的配置文件

样例工程位置	需放置的配置/认证文件
hbase-examples/hbase-example（单集群场景）	需将**准备运行环境配置文件**获取的“core-site.xml”、“hbase-site.xml”和“hdfs-site.xml”放置在样例工程的“../src/main/resources/conf”目录下。
hbase-examples/hbase-example（多集群互信场景）	将互信场景下的同名用户其中一个集群的配置文件放入“../src/main/resources/hadoopDomain”目录下，将另一集群的配置文件放入“../src/main/resources/hadoop1Domain”目录下。 其中，配置文件为**准备运行环境配置文件**获取的“core-site.xml”、“hbase-site.xml”和“hdfs-site.xml”。
hbase-examples/hbase-rest-example	-
hbase-examples/hbase-thrift-example	需将**准备ThriftServer实例配置文件**获取的“hdfs-site.xml”、“core-site.xml”、“hbase-site.xml”文件放置在样例工程的“../src/main/resources/conf”目录下。
hbase-examples/hbase-zk-example	需将以下文件放置在样例工程的“../src/main/resources”目录下： • 配置文件为**准备运行环境配置文件**获取的“core-site.xml”、“hbase-site.xml”和“hdfs-site.xml”。 • 还需确保该目录下已存在**HBase访问多个ZooKeeper样例程序**所需的“zoo.cfg”和“jaas.conf”文件。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

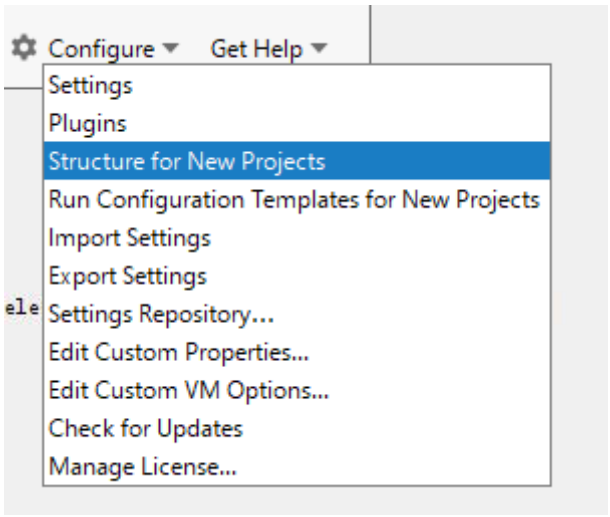
1. 打开IntelliJ IDEA，选择“Configure”。

图 12-2 Quick Start



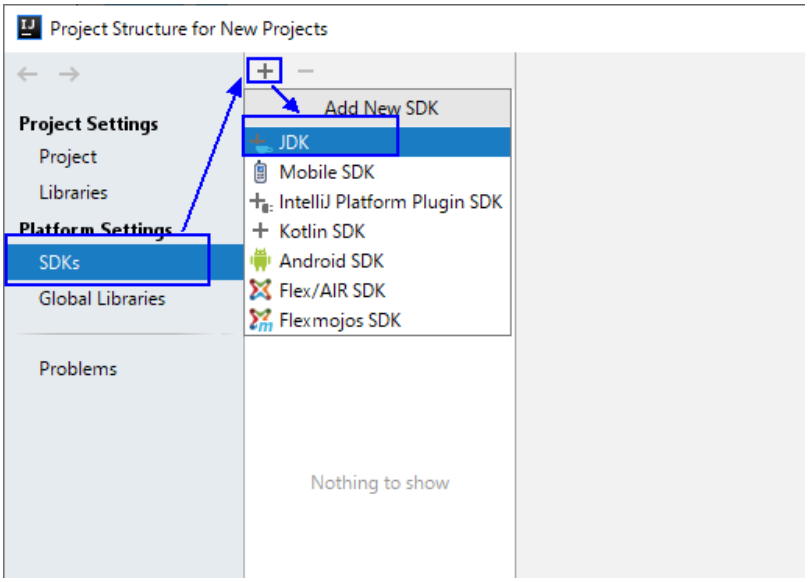
2. 在下拉框中选择“Structure for New Projects”。

图 12-3 Configure



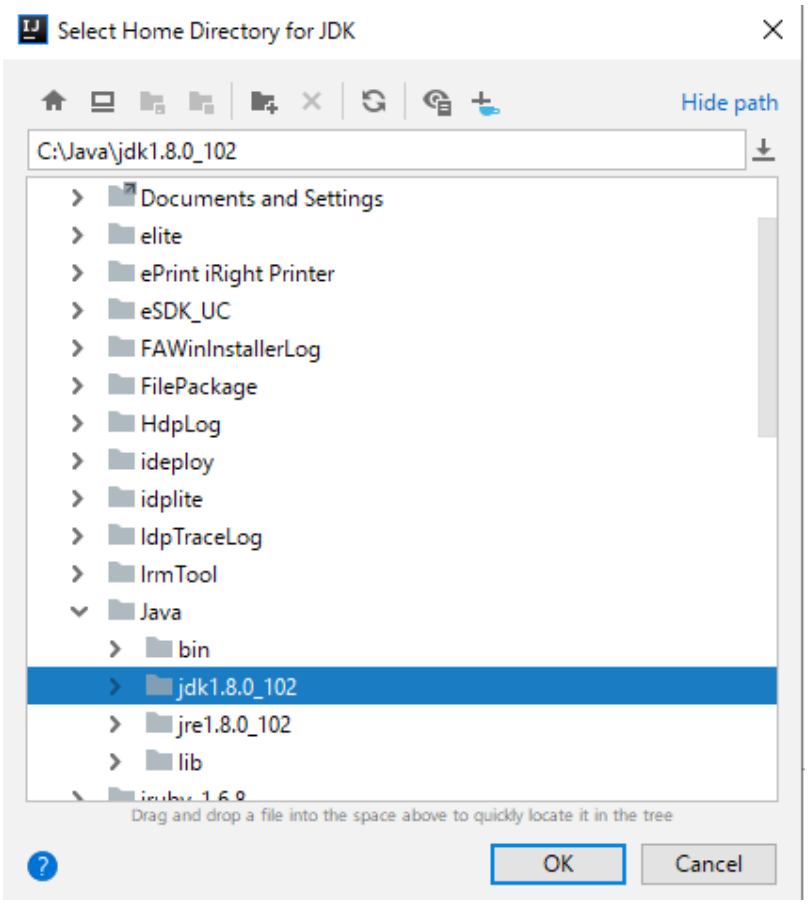
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 12-4 Project Structure for New Projects



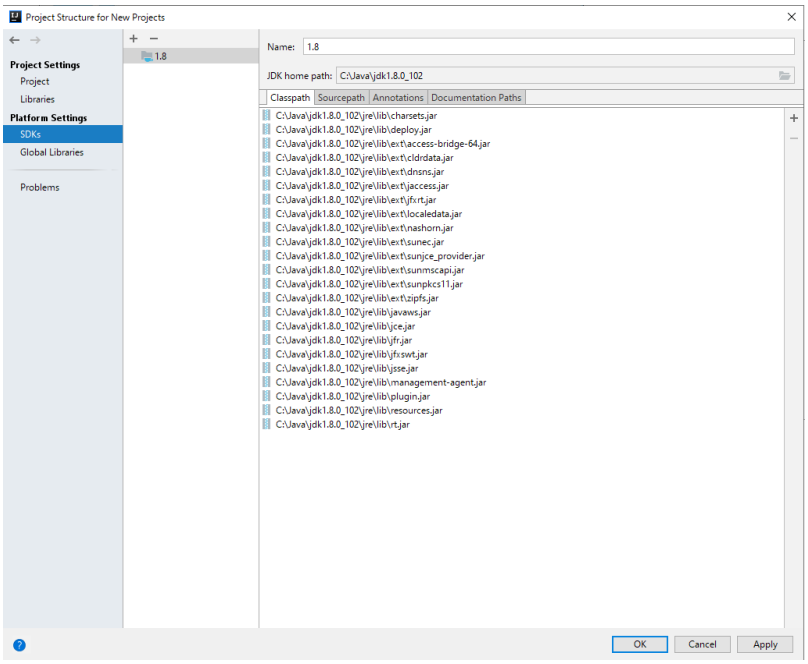
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 12-5 Select Home Directory for JDK



5. 完成JDK选择后，单击“OK”完成配置。

图 12-6 完成 JDK 配置



说明

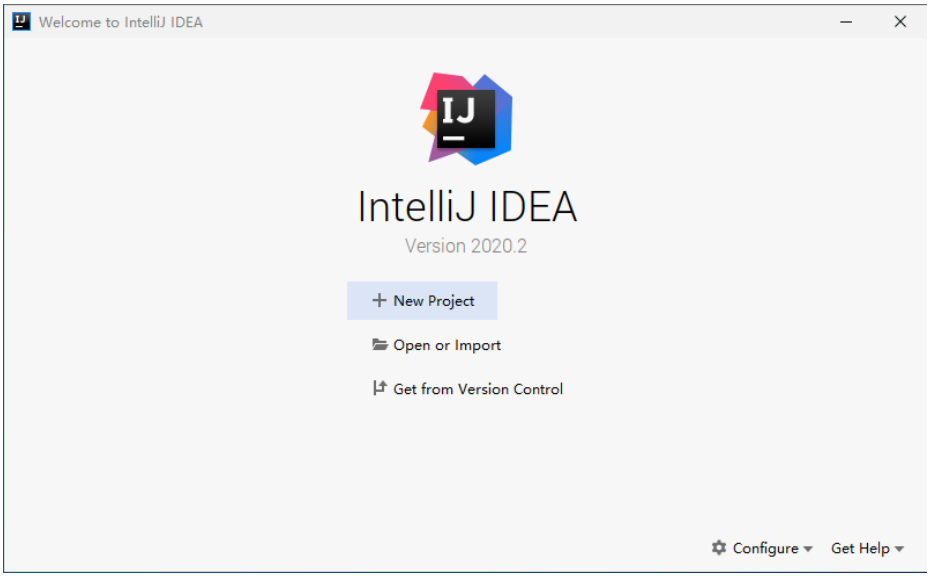
不同的IDEA版本的操作步骤可能存在差异，以实际版本的界面操作为准。

步骤4 导入样例工程到IntelliJ IDEA开发环境。

1. 打开IntelliJ IDEA，在“Quick Start”页面选择“Open or Import”。

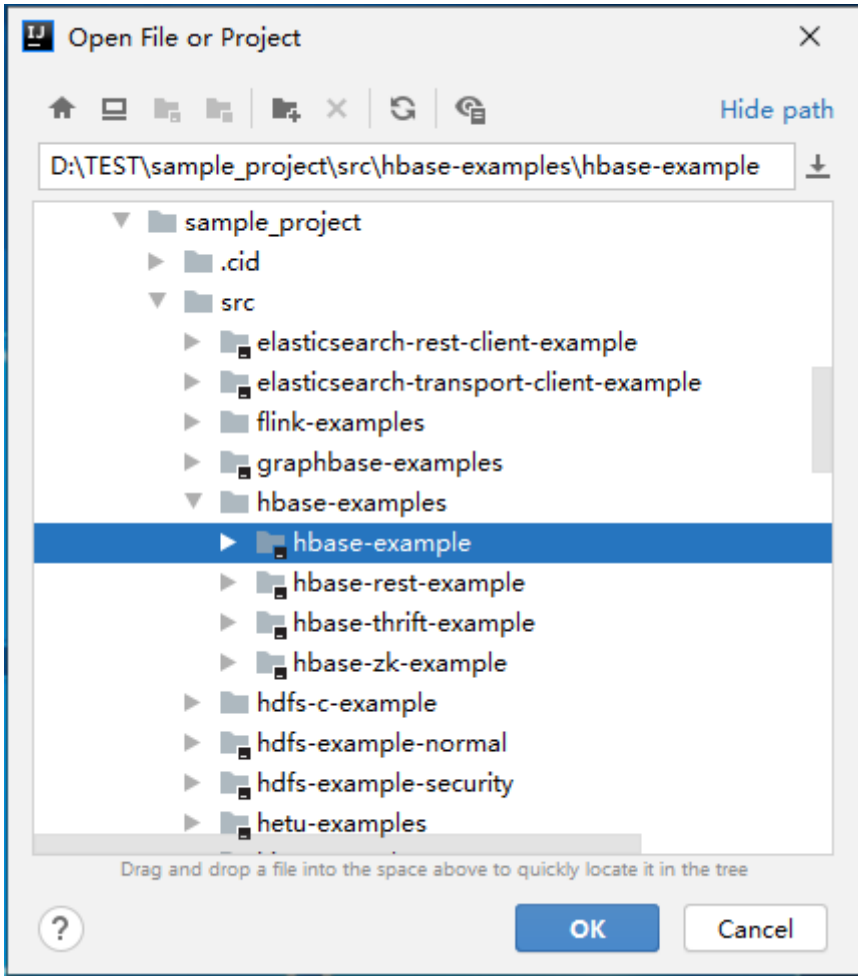
   另外，针对已使用过的IDEA工具，可以从主界面选择“File > Import project...”导入样例工程。

图 12-7 Open or Import（Quick Start 页面）



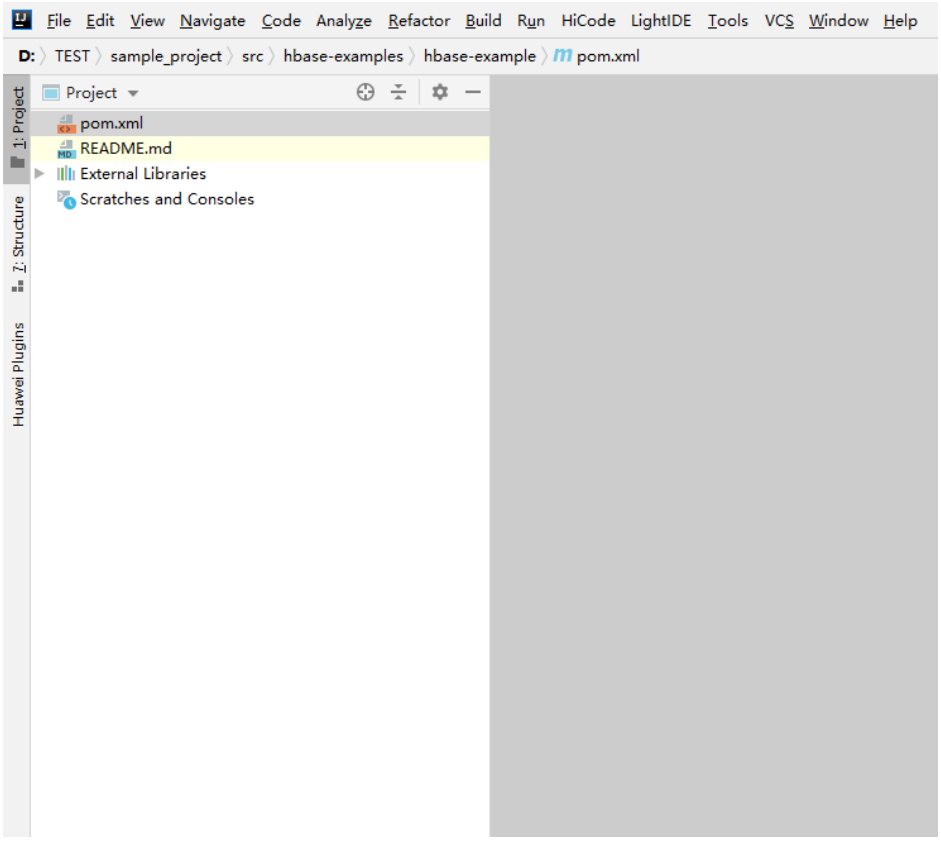
2. 选择样例工程文件夹“hbase-example”，然后单击“OK”。

图 12-8 Select File or Directory to Import



3. 导入结束，IDEA主页显示导入的样例工程。

图 12-9 导入样例工程成功



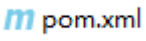
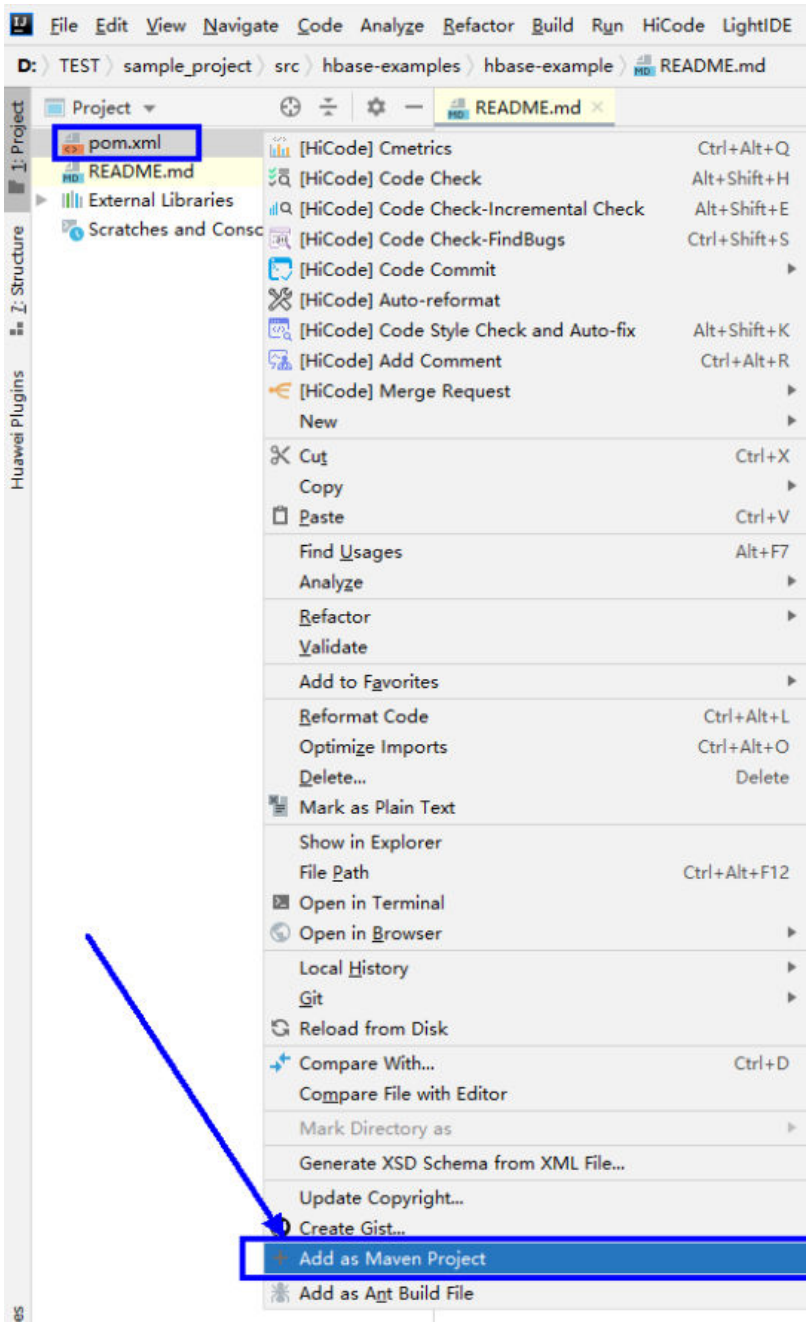
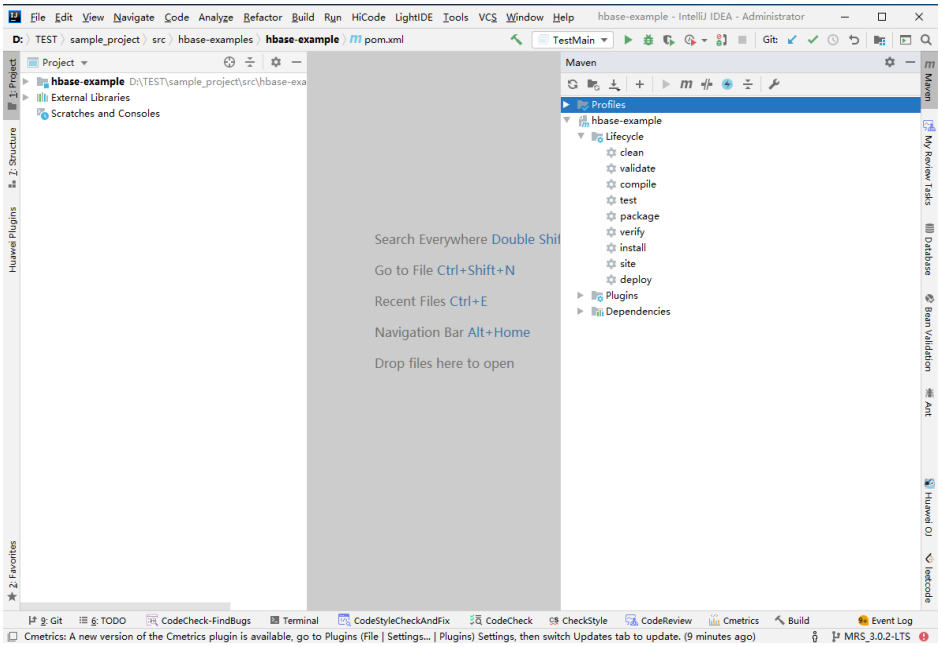
4. 右键单击“pom.xml”，选择“Add as Maven Project”，将该项目添加为Maven Project。若"pom.xml"图标如  所示，可直接进行下一步骤操作。

图 12-10 Add as Maven Project



此时IDEA可将该项目识别为Maven项目。

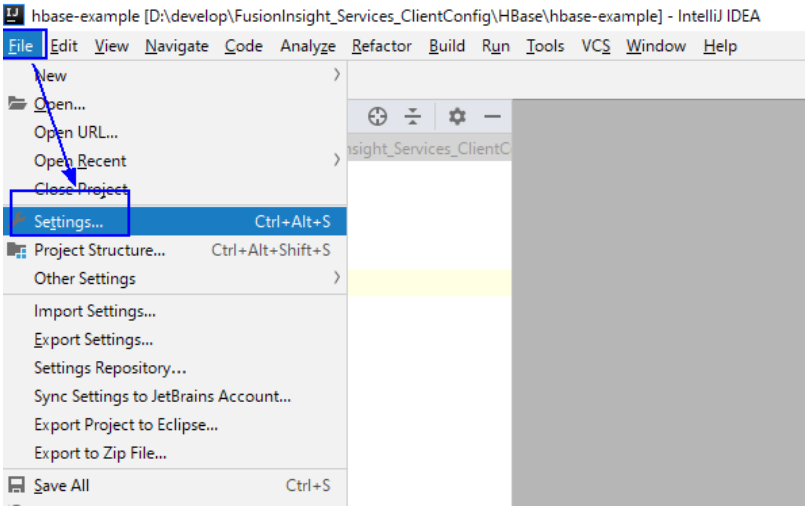
图 12-11 样例项目作为 Maven 项目在 IDEA 中显示



步骤5 设置项目使用的Maven版本。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings...”。

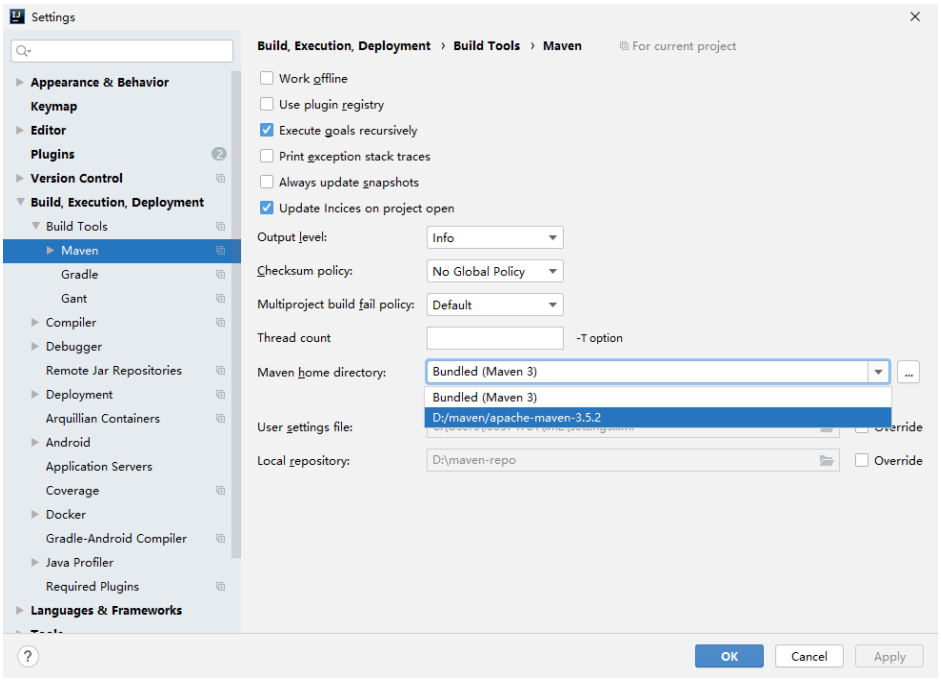
图 12-12 Settings



2. 选择“Build,Execution,Deployment > Maven”，选择Maven home directory为本地安装的Maven版本。

   然后根据实际情况设置好“User settings file”和“Local repository”参数，依次单击“Apply > OK”。

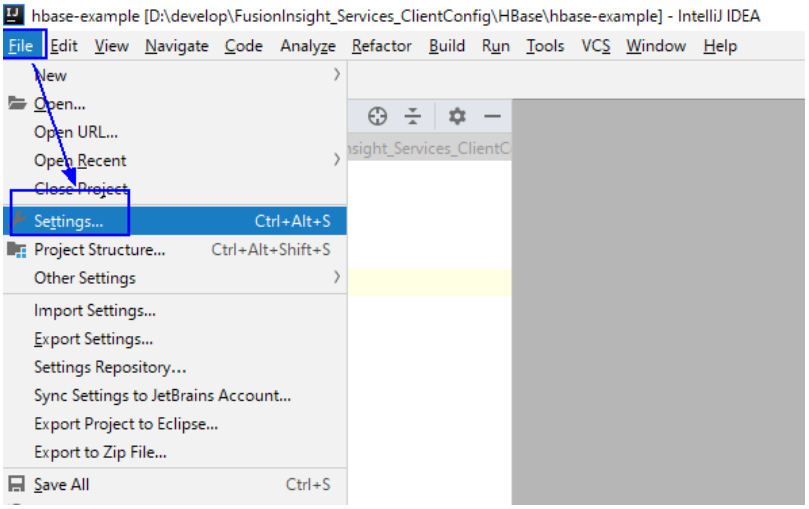
图 12-13 选择本地 Maven 安装目录



步骤6 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

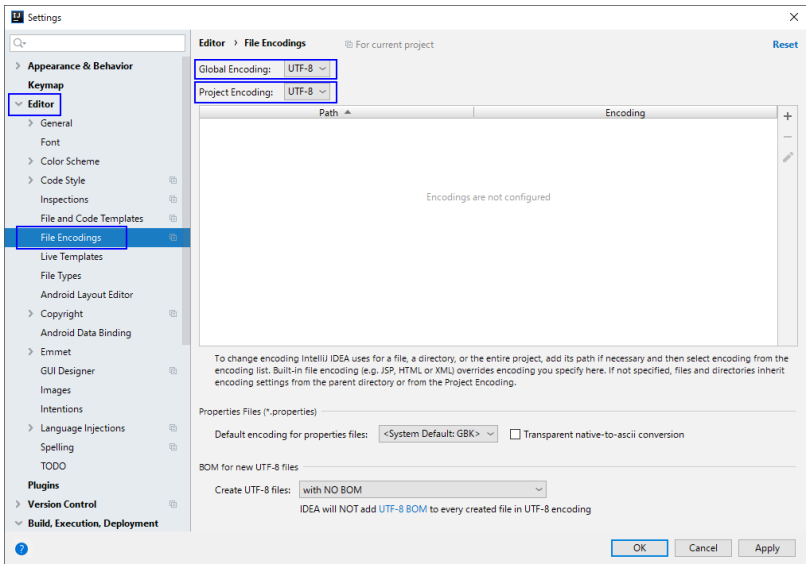
1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings...”。

图 12-14 Settings



2. 在弹出的“Settings”页面左边导航上选择“Editor > File Encodings”，分别在右侧的“Global Encoding”和“Project Encoding”的下拉框中，选择“UTF-8”。

图 12-15 File Encodings



3. 然后单击“Apply”和“OK”，完成编码配置。

----结束

12.3 开发 HBase 应用

12.3.1 HBase 数据读写示例程序

12.3.1.1 HBase 样例程序开发思路

通过典型场景，您可以快速学习和掌握HBase的开发过程，并且对关键的接口函数有所了解。

场景说明

假定用户开发一个应用程序，用于管理企业中的使用A业务的用户信息，如**表12-7**所示，A业务操作流程如下：

- 创建用户信息表。
- 在用户信息中新增用户的学历、职称等信息。
- 根据用户编号查询用户姓名和地址。
- 根据用户姓名进行查询。
- 查询年龄段在[20-29]之间的用户信息。
- 数据统计，统计用户信息表的人员数、年龄最大值、年龄最小值、平均年龄。
- 用户销户，删除用户信息表中该用户的数据。
- A业务结束后，删除用户信息表。

表 12-7 用户信息

编号	姓名	性别	年龄	地址
1200500020 1	张三	男	19	广东省深圳市
1200500020 2	李婉婷	女	23	河北省石家庄市
1200500020 3	王明	男	26	浙江省宁波市
1200500020 4	李刚	男	18	湖北省襄阳市
1200500020 5	赵恩如	女	21	江西省上饶市
1200500020 6	陈龙	男	32	湖南省株洲市
1200500020 7	周微	女	29	河南省南阳市
1200500020 8	杨艺文	女	30	重庆市开县
1200500020 9	徐兵	男	26	陕西省渭南市
1200500021 0	肖凯	男	25	辽宁省大连市

数据规划

合理地设计表结构、行键、列名能充分利用HBase的优势。本样例工程以唯一编号作为RowKey，列都存储在info列族中。

注意

HBase表以“*命名空间:表名*”格式进行存储，若在创建表时不指定命名空间，则默认存储在“default”中。其中，“hbase”命名空间为系统表命名空间，请不要对该系统表命名空间进行业务建表或数据读写等操作。

功能分解

根据上述的业务场景进行功能分解，需要开发的功能点如表12-8所示。

表 12-8 在 HBase 中开发的功能

序号	步骤	代码实现
1	根据**表12-7**中的信息创建表。	请参见**创建HBase表**。
2	导入用户数据。	请参见**向HBase表中插入数据**。
3	增加“教育信息”列族，在用户信息中新增用户的学历、职称等信息。	请参见**修改HBase表**。
4	根据用户编号查询用户姓名和地址。	请参见**使用Get API读取HBase表数据**。
5	根据用户姓名进行查询。	请参见**使用Filter过滤器读取HBase表数据**。
6	为提升查询性能，创建二级索引或者删除二级索引。	请参见**创建HBase表二级索引**和**基于二级索引查询HBase表数据**。
7	用户销户，删除用户信息表中该用户的数据。	请参见**删除HBase表数据**。
8	A业务结束后，删除用户信息表。	请参见**删除HBase表**。

关键设计原则

HBase是以RowKey为字典排序的分布式数据库系统，RowKey的设计对性能影响很大，具体的RowKey设计请考虑与业务结合。

12.3.1.2 初始化 HBase 配置

功能介绍

HBase通过login方法来获取配置项。包括用户登录信息、安全认证信息等配置项。

代码样例

下面代码片段在com.huawei.bigdata.hbase.examples包的“TestMain”类的init方法中。

```
private static void init() throws IOException {
    // Default load from conf directory
    conf = HBaseConfiguration.create();
    //In Windows environment
    String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;[1]
    //In Linux environment
    //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
    conf.addResource(new Path(userdir + "core-site.xml"), false);
    conf.addResource(new Path(userdir + "hdfs-site.xml"), false);
    conf.addResource(new Path(userdir + "hbase-site.xml"), false);
  }
```

[1]**userdir**获取的是编译后资源路径下conf目录的路径。初始化配置用到的core-site.xml、hdfs-site.xml、hbase-site.xml文件，需要放置到"src/main/resources/conf"的目录下。

12.3.1.3 创建 HBase 客户端连接

功能介绍

HBase通过ConnectionFactory.createConnection(configuration)方法创建Connection对象。传递的参数为上一步创建的Configuration。

Connection封装了底层与各实际服务器的连接以及与ZooKeeper的连接。Connection通过ConnectionFactory类实例化。创建Connection是重量级操作，Connection是线程安全的，因此，多个客户端线程可以共享一个Connection。

典型的用法，一个客户端程序共享一个单独的Connection，每一个线程获取自己的Admin或Table实例，然后调用Admin对象或Table对象提供的操作接口。不建议缓存或者池化Table、Admin。Connection的生命周期由调用者维护，调用者通过调用close()，释放资源。

代码样例

以下代码片段是登录，创建Connection并创建表的示例，在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的HBaseSample方法中。

```
    private TableName tableName = null;
    private Connection conn = null;

    public HBaseSample(Configuration conf) throws IOException {
        this.tableName = TableName.valueOf("hbase_sample_table");
        this.conn = ConnectionFactory.createConnection(conf);
}
```

说明

登录代码要避免重复调用。

12.3.1.4 创建 HBase 表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin对象的createTable方法来创建表，并指定表名、列族名。创建表有两种方式（强烈建议采用预分Region建表方式）：

- 快速建表，即创建表后整张表只有一个Region，随着数据量的增加会自动分裂成多个Region。
- 预分Region建表，即创建表时预先分配多个Region，此种方法建表可以提高写入大量数据初期的数据写入速度。

说明

表的列名以及列族名不能包含特殊字符，可以由字母、数字以及下划线组成。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testCreateTable方法中。

```
public void testCreateTable() {
        LOG.info("Entering testCreateTable.");
        // Specify the table descriptor.
        TableDescriptorBuilder htd = TableDescriptorBuilder.newBuilder(tableName);（1）

        // Set the column family name to info.
        ColumnFamilyDescriptorBuilder hcd =
ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes("info"));（2）

        // Set data encoding methods, HBase provides DIFF,FAST_DIFF,PREFIX

        hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);
        // Set compression methods, HBase provides two default compression
        // methods:GZ and SNAPPY
        // GZ has the highest compression rate,but low compression and
        // decompression effeciency,fit for cold data
        // SNAPPY has low compression rate, but high compression and
        // decompression effeciency,fit for hot data.
        // it is advised to use SNAANPPY
        hcd.setCompressionType(Compression.Algorithm.SNAPPY);//注[1]
        htd.setColumnFamily(hcd.build()); （3）
        Admin admin = null;
        try {
          // Instantiate an Admin object.
          admin = conn.getAdmin(); （4）
          if (!admin.tableExists(tableName)) {
            LOG.info("Creating table...");
            admin.createTable(htd.build());//注[2] （5）
            LOG.info(admin.getClusterMetrics().toString());
            LOG.info(admin.listNamespaceDescriptors().toString());
            LOG.info("Table created successfully.");
          } else {
            LOG.warn("table already exists");
          }
        } catch (IOException e) {
            LOG.error("Create table failed " ,e);
        } finally {
          if (admin != null) {
            try {
              // Close the Admin object.
              admin.close();
            } catch (IOException e) {
              LOG.error("Failed to close admin " ,e);
            }
          }
        }
        LOG.info("Exiting testCreateTable.");
 }
```

参数说明

（1）创建表描述符

（2）创建列族描述符

（3）添加列族描述符到表描述符中

（4）获取Admin对象，Admin提供了建表、创建列族、检查表是否存在、修改表结构和列族结构以及删除表等功能。

（5）调用Admin的建表方法。

注意事项

- 注[1] 可以设置列族的压缩方式，代码片段如下：

```
//设置编码算法，HBase提供了DIFF，FAST_DIFF，PREFIX三种编码算法
 hcd.setDataBlockEncoding(DataBlockEncoding.FAST_DIFF);

 //设置文件压缩方式，HBase默认提供了GZ和SNAPPY两种压缩算法
 //其中GZ的压缩率高，但压缩和解压性能低，适用于冷数据
 //SNAPPY压缩率低，但压缩解压性能高，适用于热数据
 //建议默认开启SNAPPY压缩
 hcd.setCompressionType(Compression.Algorithm.SNAPPY);
```

- 注[2] 可以通过指定起始和结束RowKey，或者通过RowKey数组预分Region两种方式建表，代码片段如下：

```
// 创建一个预划分region的表
 byte[][] splits = new byte[4][];
 splits[0] = Bytes.toBytes("A");
 splits[1] = Bytes.toBytes("H");
 splits[2] = Bytes.toBytes("O");
 splits[3] = Bytes.toBytes("U");
 admin.createTable(htd, splits);
```

12.3.1.5 创建 HBase 表 Region

功能简介

一般通过org.apache.hadoop.hbase.client.HBaseAdmin进行多点分割。注意：分割操作只对空Region起作用。

本例使用multiSplit进行多点分割将HBase表按照“-∞~A”“A~D”、“D~F”、“F~H”、“H~+∞”分为五个Region。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testMultiSplit方法中。

```
public void testMultiSplit() {
   LOG.info("Entering testMultiSplit.");

   Table table = null;
   Admin admin = null;
   try {
     admin = conn.getAdmin();

     // initilize a HTable object
     table = conn.getTable(tableName);
     Set<HRegionInfo> regionSet = new HashSet<HRegionInfo>();
     List<HRegionLocation> regionList = conn.getRegionLocator(tableName).getAllRegionLocations();
     for(HRegionLocation hrl : regionList){
         regionSet.add(hrl.getRegionInfo());
     }
     byte[][] sk = new byte[4][];
     sk[0] = "A".getBytes();
     sk[1] = "D".getBytes();
     sk[2] = "F".getBytes();
     sk[3] = "H".getBytes();
     for (RegionInfo regionInfo : regionSet) {
       admin.multiSplitSync(regionInfo.getRegionName(), sk);
     }
     LOG.info("MultiSplit successfully.");
   } catch (Exception e) {
     LOG.error("MultiSplit failed.");
   } finally {
     if (table != null) {
```

```
      try {
        // Close table object
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Close admin failed " ,e);
      }
    }
  }
  LOG.info("Exiting testMultiSplit.");
}
```

注意：分割操作只对空Region起作用。

12.3.1.6 向 HBase 表中插入数据

功能简介

HBase是一个面向列的数据库，一行数据，可能对应多个列族，而一个列族又可以对应多个列。通常，写入数据的时候，需要指定要写入的列（含列族名称和列名称）。HBase通过HTable的put方法来Put数据，可以是一行数据也可以是数据集。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testPut方法中

```
public void testPut() {
    LOG.info("Entering testPut.");

    // Specify the column family name.
    byte[] familyName = Bytes.toBytes("info");
    // Specify the column name.
    byte[][] qualifiers = {
        Bytes.toBytes("name"), Bytes.toBytes("gender"), Bytes.toBytes("age"), Bytes.toBytes("address")
    };

    Table table = null;
    try {
        // Instantiate an HTable object.
        table = conn.getTable(tableName);
        List<Put> puts = new ArrayList<Put>();

        // Instantiate a Put object.
        Put put = putData(familyName, qualifiers,
            Arrays.asList("012005000201", "Zhang San", "Male", "19", "Shenzhen, Guangdong"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000202", "Li Wanting", "Female", "23", "Shijiazhuang, Hebei"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000203", "Wang Ming", "Male", "26", "Ningbo, Zhejiang"));
        puts.add(put);

        put = putData(familyName, qualifiers,
```

```
            Arrays.asList("012005000204", "Li Gang", "Male", "18", "Xiangyang, Hubei"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000205", "Zhao Enru", "Female", "21", "Shangrao, Jiangxi"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000206", "Chen Long", "Male", "32", "Zhuzhou, Hunan"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000207", "Zhou Wei", "Female", "29", "Nanyang, Henan"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000208", "Yang Yiwen", "Female", "30", "Kaixian, Chongqing"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000209", "Xu Bing", "Male", "26", "Weinan, Shaanxi"));
        puts.add(put);

        put = putData(familyName, qualifiers,
            Arrays.asList("012005000210", "Xiao Kai", "Male", "25", "Dalian, Liaoning"));
        puts.add(put);

        // Submit a put request.
        table.put(puts);

        LOG.info("Put successfully.");
    } catch (IOException e) {
        LOG.error("Put failed ", e);
    } finally {
        if (table != null) {
            try {
                // Close the HTable object.
                table.close();
            } catch (IOException e) {
                LOG.error("Close table failed ", e);
            }
        }
    }
    LOG.info("Exiting testPut.");
}
```

注意事项

不允许多个线程在同一时间共用同一个HTable实例。HTable是一个非线程安全类，因此，同一个HTable实例，不应该被多个线程同时使用，否则可能会带来并发问题。

12.3.1.7 创建 HBase 表二级索引

功能简介

一般都通过调用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中方法进行HBase二级索引的管理，该类中提供了创建索引的方法。

说明

二级索引不支持修改，如果需要修改，请先删除旧的然后重新创建。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的createIndex方法中。

```
public void createIndex() {
   LOG.info("Entering createIndex.");

   String indexName = "index_name";
   // Create hindex instance
   TableIndices tableIndices = new TableIndices();
   IndexSpecification iSpec = new IndexSpecification(indexName);
   iSpec.addIndexColumn(ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes("info")).build(),
         "name", ValueType.STRING);//注[1]
   tableIndices.addIndex(iSpec);

   HIndexAdmin iAdmin = null;
   Admin admin = null;
   try {

     admin = conn.getAdmin();
     iAdmin = HIndexClient.newHIndexAdmin(admin);

     // add index to the table
     iAdmin.addIndices(tableName, tableIndices);

     LOG.info("Create index successfully.");
   } catch (IOException e) {
     LOG.error("Create index failed " ,e);
   } finally {
     if (admin != null) {
        try {
          admin.close();
     } catch (IOException e) {
        LOG.error("Close admin failed " ,e);
     }
     }
     if (iAdmin != null) {
       try {
         // Close IndexAdmin Object
         iAdmin.close();
       } catch (IOException e) {
         LOG.error("Close admin failed " ,e);
       }
     }
   }
   LOG.info("Exiting createIndex.");
 }
```

新创建的二级索引默认是不启用的，如果需要启用指定的二级索引，可以参考如下代码片段。该代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的enableIndex方法中。

```
   public void enableIndex() {
      LOG.info("Entering createIndex.");

      // Name of the index to be enabled
      String indexName = "index_name";

      List<String> indexNameList = new ArrayList<String>();
      indexNameList.add(indexName);

      HIndexAdmin iAdmin = null;
      Admin admin = null;
      try {
         admin = conn.getAdmin();
         iAdmin = HIndexClient.newHIndexAdmin(admin);
```



```
            // Alternately, enable the specified indices
            iAdmin.enableIndices(tableName, indexNameList);
            LOG.info("Successfully enable indices {}  of the table {}", indexNameList, tableName);
        } catch (IOException e) {
            LOG.error("Failed to enable indices {}  of the table {} . {}", indexNameList, tableName, e);
        } finally {
            if (admin != null) {
                try {
                    admin.close();
                } catch (IOException e) {
                    LOG.error("Close admin failed ", e);
                }
            }
            if (iAdmin != null) {
                try {
                    iAdmin.close();
                } catch (IOException e) {
                    LOG.error("Close admin failed ", e);
                }
            }
        }
    }
```

注意事项

注[1]：创建联合索引

HBase支持在多个字段上创建二级索引，例如在列name和age上。

```
HIndexSpecification iSpecUnite = new HIndexSpecification(indexName);
 iSpecUnite.addIndexColumn(new HColumnDescriptor("info"), "name", ValueType.String);
 iSpecUnite.addIndexColumn(new HColumnDescriptor("info"), "age", ValueType.String);
```

相关操作

使用命令创建索引表。

您还可以通过TableIndexer工具在已有用户表中创建索引。

说明

<table_name>用户表必须存在。

```
hbase org.apache.hadoop.hbase.hindex.mapreduce.TableIndexer -Dtablename.to.index=<table_name> -
Dindexspecs.to.add='IDX1=>cf1:[q1->datatype];cf2:[q1->datatype],[q2->datatype],[q3-
>datatype]#IDX2=>cf1:[q5->datatype]' -Dindexnames.to.build='IDX1'
```

“#”用于区分不同的索引，“;”用于区分不同的列族，“,”用于区分不同的列。

tablename.to.index：创建索引的用户表表名。

indexspecs.to.add：创建索引对应的用户表列。

其中命令中各参数的含义如下：

- IDX1：索引名称
- cf1：列族名称。
- q1：列名。
- datatype：数据类型。数据类型仅支持Integer、String、Double、Float、Long、Short、Byte、Char类型。

12.3.1.8 基于二级索引查询 HBase 表数据

功能介绍

针对添加了二级索引的用户表，您可以通过Filter来查询数据。其数据查询性能高于针对无二级索引用户表的数据查询。

说明

- HIndex支持的Filter类型为“SingleColumnValueFilter”，“SingleColumnValueExcludeFilter”以及“SingleColumnValuePartitionFilter”。
- HIndex支持的Comparator为“BinaryComparator”，“BitComparator”，“LongComparator”，“DecimalComparator”，“DoubleComparator”，“FloatComparator”，“IntComparator”，“NullComparator”。

二级索引的使用规则如下：

- 针对某一列或者多列创建了单索引的场景下：
  - 当查询时使用此列进行过滤时，不管是AND还是OR操作，该索引都会被利用来提升查询性能。

    例如：Filter_Condition(IndexCol1) AND/OR Filter_Condition(IndexCol2)
  - 当查询时使用“索引列AND非索引列”过滤时，此索引会被利用来提升查询性能。

    例如：Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2) AND Filter_Condition(NonIndexCol1)
  - 当查询时使用“索引列OR非索引列”过滤时，此索引将不会被使用，查询性能不会因为索引得到提升。

    例如：Filter_Condition(IndexCol1) AND/OR Filter_Condition(IndexCol2) OR Filter_Condition(NonIndexCol1)
- 针对多个列创建的联合索引场景下：
  - 当查询时使用的列（多个），是联合索引所有对应列的一部分或者全部，且列的顺序与联合索引一致时，此索引会被利用来提升查询性能。

    例如，针对C1、C2、C3列创建了联合索引，生效的场景包括：

    Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2) AND Filter_Condition(IndexCol3)

    Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2)

    Filter_Condition(IndexCol1)

    不生效的场景包括：

    Filter_Condition(IndexCol2) AND Filter_Condition(IndexCol3)

    Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol3)

    Filter_Condition(IndexCol2)

    Filter_Condition(IndexCol3)
  - 当查询时使用“索引列AND非索引列”过滤时，此索引会被利用来提升查询性能。

    例如：

    Filter_Condition(IndexCol1) AND Filter_Condition(NonIndexCol1)

    Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2) AND Filter_Condition(NonIndexCol1)

  - 当查询时使用“索引列OR非索引列”过滤时，此索引不会被使用，查询性能不会因为索引得到提升。

    例如：

    Filter_Condition(IndexCol1) OR Filter_Condition(NonIndexCol1)

    (Filter_Condition(IndexCol1) AND Filter_Condition(IndexCol2))OR (Filter_Condition(NonIndexCol1))
  - 当查询时使用多个列进行范围查询时，只有联合索引中最后一个列可指定取值范围，前面的列只能设置为“=”。

    例如：针对C1、C2、C3列创建了联合索引，需要进行范围查询时，只能针对C3设置取值范围，过滤条件为“C1=XXX，C2=XXX，C3=取值范围”。
- 针对添加了二级索引的用户表，可以通过Filter来查询数据，在单列索引和复合列索引上进行过滤查询，查询结果都与无索引结果相同，且其数据查询性能高于无二级索引用户表的数据查询性能。

代码样例

下面代码片段在com.huawei.hadoop.hbase.example包的“HBaseSample”类的testScanDataByIndex方法中：

样例：使用二级索引查找数据

```
public void testScanDataByIndex() {
  LOG.info("Entering testScanDataByIndex.");
  Table table = null;
  ResultScanner scanner = null;
  try {
    table = conn.getTable(tableName);

    // Create a filter for indexed column.
    Filter filter = new SingleColumnValueFilter(Bytes.toBytes("info"), Bytes.toBytes("name"),
        CompareOperator.EQUAL, "Li Gang".getBytes());
    Scan scan = new Scan();
    scan.setFilter(filter);
    scanner = table.getScanner(scan);
    LOG.info("Scan indexed data.");

    for (Result result : scanner) {
      for (Cell cell : result.rawCells()) {
        LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
          Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
          Bytes.toString(CellUtil.cloneValue(cell)));
      }
    }
    LOG.info("Scan data by index successfully.");
  } catch (IOException e) {
    LOG.error("Scan data by index failed.");
  } finally {
    if (scanner != null) {
      // Close the scanner object.
      scanner.close();
    }
    try {
      if (table != null) {
        table.close();
      }
    } catch (IOException e) {
      LOG.error("Close table failed.");
    }
  }

  LOG.info("Exiting testScanDataByIndex.");
}
```


注意事项

需要预先对字段name创建二级索引。

相关操作

基于二级索引表查询。

查询样例如下：

用户在hbase_sample_table的info列族的name列添加一个索引，在客户端执行，

```
hbase org.apache.hadoop.hbase.hindex.mapreduce.TableIndexer -Dtablename.to.index=hbase_sample_table -Dindexspecs.to.add='IDX1=>info:[name->String]' -Dindexnames.to.build='IDX1'
```

然后用户需要查询“info:name”，在**hbase shell**执行如下命令：

>scan 'hbase_sample_table',
{FILTER=>"SingleColumnValueFilter(*family*,*qualifier*,*compareOp*,*comparator*,*filterIfMissing*,*latestVersionOnly*)"}

说明

hbase shell下面做复杂的查询请使用API进行处理。

参数说明：

- family：需要查询的列所在的列族，例如info；
- qualifier：需要查询的列，例如name；
- compareOp：比较符，例如=、>等；
- comparator：需要查找的目标值，例如binary:Zhang San；
- filterIfMissing：如果某一行不存在该列，是否过滤，默认值为false；
- latestVersionOnly：是否仅查询最新版本的值，默认值为false。

例如：

```
>scan 'hbase_sample_table',{FILTER=>"SingleColumnValueFilter('info','name',=,'binary:Zhang San',true,true)"}
```

12.3.1.9 修改 HBase 表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin的modifyTable方法修改表信息。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testModifyTable方法中

```
public void testModifyTable() {
  LOG.info("Entering testModifyTable.");

  // Specify the column family name.
  byte[] familyName = Bytes.toBytes("education");
  Admin admin = null;
  try {
    // Instantiate an Admin object.
    admin = conn.getAdmin();
    // Obtain the table descriptor.
    TableDescriptor htd = admin.getDescriptor(tableName);
```

```
    // Check whether the column family is specified before modification.
    if (!htd.hasColumnFamily(familyName)) {
      // Create the column descriptor.
      TableDescriptor tableBuilder = TableDescriptorBuilder.newBuilder(htd)
          .setColumnFamily(ColumnFamilyDescriptorBuilder.newBuilder(familyName).build()).build();


      // Disable the table to get the table offline before modifying
      // the table.
      admin.disableTable(tableName);//注[1]
      // Submit a modifyTable request.
      admin.modifyTable(tableBuilder);
      // Enable the table to get the table online after modifying the
      // table.
      admin.enableTable(tableName);
    }
    LOG.info("Modify table successfully.");
  } catch (IOException e) {
    LOG.error("Modify table failed " ,e);
  } finally {
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Close admin failed " ,e);
      }
    }
  }
  LOG.info("Exiting testModifyTable.");
}
```

注意事项

注[1] modifyTable只有表被disable时，才能生效。

12.3.1.10 使用 Get API 读取 HBase 表数据

功能简介

要从表中读取一条数据，首先需要实例化该表对应的Table实例，然后创建一个Get对象。也可以为Get对象设定参数值，如列族的名称和列的名称。查询到的行数据存储在Result对象中，Result中可以存储多个Cell。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testGet方法中

```
public void testGet() {
    LOG.info("Entering testGet.");
    // Specify the column family name.
    byte[] familyName = Bytes.toBytes("info");
    // Specify the column name.
    byte[][] qualifier = { Bytes.toBytes("name"), Bytes.toBytes("address") };
    // Specify RowKey.
    byte[] rowKey = Bytes.toBytes("012005000201");
    Table table = null;
    try {
      // Create the Table instance.
      table = conn.getTable(tableName);
      // Instantiate a Get object.
      Get get = new Get(rowKey);
      // Set the column family name and column name.
```

```
    get.addColumn(familyName, qualifier[0]);
    get.addColumn(familyName, qualifier[1]);
    // Submit a get request.
    Result result = table.get(get);
    // Print query results.
    for (Cell cell : result.rawCells()) {
      LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
                Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
                Bytes.toString(CellUtil.cloneValue(cell)));
    }
    LOG.info("Get data successfully.");
  } catch (IOException e) {
    LOG.error("Get data failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testGet.");
}
```

12.3.1.11 使用 Scan API 读取 HBase 表数据

功能简介

要从表中读取数据，首先需要实例化该表对应的Table实例，然后创建一个Scan对象，并针对查询条件设置Scan对象的参数值，为了提高查询效率，建议指定StartRow和StopRow。查询结果的多行数据保存在ResultScanner对象中，每行数据以Result对象形式存储，Result中存储了多个Cell。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testScanData方法中

```
public void testScanData() {
    LOG.info("Entering testScanData.");
    Table table = null;
    // Instantiate a ResultScanner object.
    ResultScanner rScanner = null;
    try {
      // Create the Configuration instance.
      table = conn.getTable(tableName);
      // Instantiate a Get object.
      Scan scan = new Scan();
      scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
      // Set the cache size.
      scan.setCaching(1000);

      // Submit a scan request.
      rScanner = table.getScanner(scan);
      // Print query results.
       for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
        for (Cell cell : r.rawCells()) {
          LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
              Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
              Bytes.toString(CellUtil.cloneValue(cell)));
        }
      }
      LOG.info("Scan data successfully.");
```

```
  } catch (IOException e) {
    LOG.error("Scan data failed " ,e);
  } finally {
    if (rScanner != null) {
      // Close the scanner object.
      rScanner.close();
    }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testScanData.");
}
```

注意事项

1. 建议Scan时指定StartRow和StopRow，一个有确切范围的Scan，性能会更好些。
2. 可以设置Batch和Caching关键参数。
   - Batch
     使用Scan调用next接口每次最大返回的记录数，与一次读取的列数有关。
   - Caching
     RPC请求返回next记录的最大数量，该参数与一次RPC获取的行数有关。

12.3.1.12 使用 Filter 过滤器读取 HBase 表数据

功能简介

HBase Filter主要在Scan和Get过程中进行数据过滤，通过设置一些过滤条件来实现，如设置RowKey、列名或者列值的过滤条件。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testSingleColumnValueFilter方法中。

```
public void testSingleColumnValueFilter() {
    LOG.info("Entering testSingleColumnValueFilter.");
    Table table = null;

    ResultScanner rScanner = null;

    try {

      table = conn.getTable(tableName);

      Scan scan = new Scan();
      scan.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"));
      // Set the filter criteria.
      SingleColumnValueFilter filter = new SingleColumnValueFilter(
          Bytes.toBytes("info"), Bytes.toBytes("name"), CompareOperator.EQUAL,
          Bytes.toBytes("Xu Bing"));
      scan.setFilter(filter);
      // Submit a scan request.
      rScanner = table.getScanner(scan);
      // Print query results.
       for (Result r = rScanner.next(); r != null; r = rScanner.next()) {
```

```
      for (Cell cell : r.rawCells()) {
        LOG.info("{}:{},{},{}", Bytes.toString(CellUtil.cloneRow(cell)),
            Bytes.toString(CellUtil.cloneFamily(cell)), Bytes.toString(CellUtil.cloneQualifier(cell)),
            Bytes.toString(CellUtil.cloneValue(cell)));
      }
    }
    LOG.info("Single column value filter successfully.");
  } catch (IOException e) {
    LOG.error("Single column value filter failed " ,e);
  } finally {
      if (rScanner != null) {
        // Close the scanner object.
        rScanner.close();
      }
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testSingleColumnValueFilter.");
}
```

注意事项

当前二级索引不支持使用SubstringComparator类定义的对象作为Filter的比较器。

例如，如下示例中的用法当前不支持：

```
Scan scan = new Scan();
filterList = new FilterList(FilterList.Operator.MUST_PASS_ALL);
filterList.addFilter(new SingleColumnValueFilter(Bytes
.toBytes(columnFamily), Bytes.toBytes(qualifier),
CompareOperator.EQUAL, new SubstringComparator(substring)));
scan.setFilter(filterList);
```

12.3.1.13 删除 HBase 表数据

功能简介

HBase通过Table实例的delete方法来Delete数据，可以是一行数据也可以是数据集。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的testDelete方法中

```
public void testDelete() {
  LOG.info("Entering testDelete.");

  byte[] rowKey = Bytes.toBytes("012005000201");

  Table table = null;
  try {
    // Instantiate an HTable object.
    table = conn.getTable(tableName);

    // Instantiate an Delete object.
    Delete delete = new Delete(rowKey);

    // Submit a delete request.
    table.delete(delete);
```

```
    LOG.info("Delete table successfully.");
  } catch (IOException e) {
    LOG.error("Delete table failed " ,e);
  } finally {
    if (table != null) {
      try {
        // Close the HTable object.
        table.close();
      } catch (IOException e) {
        LOG.error("Close table failed " ,e);
      }
    }
  }
  LOG.info("Exiting testDelete.");
}
```

说明

如果被删除的cell所在的列族上设置了二级索引，也会同步删除索引数据。

12.3.1.14 删除 HBase 二级索引

功能简介

一般都通过调用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中方法进行HBase二级索引的管理，该类中提供了索引的查询和删除等方法。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的dropIndex方法中。

```
public void dropIndex() {
    LOG.info("Entering dropIndex.");
    String indexName = "index_name";
    List<String> indexNameList = new ArrayList<String>();
    indexNameList.add(indexName);

    IndexAdmin iAdmin = null;
    try {
      // Instantiate HIndexAdmin Object
      iAdmin = HIndexClient.newHIndexAdmin(conn.getAdmin());
      // Delete Secondary Index
      iAdmin.dropIndex(tableName, indexNameList);

      LOG.info("Drop index successfully.");
    } catch (IOException e) {
      LOG.error("Drop index failed.");
    } finally {
      if (iAdmin != null) {
        try {
          // Close Secondary Index
          iAdmin.close();
        } catch (IOException e) {
          LOG.error("Close admin failed.");
        }
      }
    }
    LOG.info("Exiting dropIndex.");
  }
```

12.3.1.15 删除 HBase 表

功能简介

HBase通过org.apache.hadoop.hbase.client.Admin的deleteTable方法来删除表。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseSample”类的dropTable方法中

```
public void dropTable() {
    LOG.info("Entering dropTable.");
    Admin admin = null;
    try {
      admin = conn.getAdmin();
      if (admin.tableExists(tableName)) {
        // Disable the table before deleting it.
        admin.disableTable(tableName);
        // Delete table.
        admin.deleteTable(tableName);//注[1]
      }
      LOG.info("Drop table successfully.");
    } catch (IOException e) {
      LOG.error("Drop table failed " ,e);
    } finally {
      if (admin != null) {
        try {
          // Close the Admin object.
          admin.close();
        } catch (IOException e) {
          LOG.error("Close admin failed " ,e);
        }
      }
    }
    LOG.info("Exiting dropTable.");
  }
```

注意事项

注[1] 只有表被disable时，才能被删除掉，所以deleteTable常与disableTable，enableTable，tableExists，isTableEnabled，isTableDisabled结合在一起使用。

12.3.1.16 创建 Phoenix 表

功能简介

Phoenix依赖HBase作为其后备存储，支持标准SQL和JDBC API的强大功能，使得SQL用户可以访问HBase集群。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“PhoenixSample”类的testCreateTable方法中。

```
/**
 * Create Table
 */
public void testCreateTable() {
  LOG.info("Entering testCreateTable.");
```

```
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Create table
  String createTableSQL =
      "CREATE TABLE IF NOT EXISTS TEST (id integer not null primary key, name varchar, "
          + "account char(6), birth date)";
  try (Connection conn = DriverManager.getConnection(url, props);
      Statement stat = conn.createStatement()) {
    // Execute Create SQL
    stat.executeUpdate(createTableSQL);
    LOG.info("Create table successfully.");
  } catch (Exception e) {
    LOG.error("Create table failed.", e);
  }
  LOG.info("Exiting testCreateTable.");
}
/**
 * Drop Table
 */
public void testDrop() {
  LOG.info("Entering testDrop.");
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Delete table
  String dropTableSQL = "DROP TABLE TEST";

  try (Connection conn = DriverManager.getConnection(url, props);
      Statement stat = conn.createStatement()) {
    stat.executeUpdate(dropTableSQL);
    LOG.info("Drop successfully.");
  } catch (Exception e) {
    LOG.error("Drop failed.", e);
  }
  LOG.info("Exiting testDrop.");
}
```

12.3.1.17 向 Phoenix 表中写入数据

功能简介

使用Phoenix实现写数据。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“PhoenixSample”类的testPut方法中。

```
/**
 * Put data
 */
public void testPut() {
  LOG.info("Entering testPut.");
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Insert
  String upsertSQL =
      "UPSERT INTO TEST VALUES(1,'John','100000', TO_DATE('1980-01-01','yyyy-MM-dd'))";
  try (Connection conn = DriverManager.getConnection(url, props);
      Statement stat = conn.createStatement()){
    // Execute Update SQL
    stat.executeUpdate(upsertSQL);
    conn.commit();
    LOG.info("Put successfully.");
  } catch (Exception e) {
    LOG.error("Put failed.", e);
  }
  LOG.info("Exiting testPut.");
}
```

12.3.1.18 读取 Phoenix 表数据

功能简介

使用Phoenix实现读数据。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“PhoenixSample”类的testSelect方法中。

```
/**
 * Select Data
 */
public void testSelect() {
  LOG.info("Entering testSelect.");
  String URL = "jdbc:phoenix:" + conf.get("hbase.zookeeper.quorum");
  // Query
  String querySQL = "SELECT * FROM TEST WHERE id = ?";
  Connection conn = null;
  PreparedStatement preStat = null;
  Statement stat = null;
  ResultSet result = null;
  try {
    // Create Connection
    conn = DriverManager.getConnection(url, props);
    // Create Statement
    stat = conn.createStatement();
    // Create PrepareStatement
    preStat = conn.prepareStatement(querySQL);
    // Execute query
    preStat.setInt(1, 1);
    result = preStat.executeQuery();
    // Get result
    while (result.next()) {
      int id = result.getInt("id");
      String name = result.getString(1);
      System.out.println("id: " + id);
      System.out.println("name: " + name);
    }
    LOG.info("Select successfully.");
  } catch (Exception e) {
    LOG.error("Select failed.", e);
  } finally {
    if (null != result) {
      try {
        result.close();
      } catch (Exception e2) {
        LOG.error("Result close failed.", e2);
      }
    }
    if (null != stat) {
      try {
        stat.close();
      } catch (Exception e2) {
        LOG.error("Stat close failed.", e2);
      }
    }
    if (null != conn) {
      try {
        conn.close();
      } catch (Exception e2) {
        LOG.error("Connection close failed.", e2);
      }
    }
  }
```

```
  LOG.info("Exiting testSelect.");
 }
```

12.3.1.19 配置 HBase 应用输出运行日志

功能介绍

将hbase client的日志单独输出到指定日志文件，与业务日志分开，方便分析定位hbase的问题。如果进程中已经有log4j的配置，需要将hbase-example\src\main\resources\log4j.properties中RFA与RFAS相关的配置复制到已有的log4j配置中。

代码样例

```
hbase.root.logger=INFO,console,RFA              //hbase客户端日志输出配置，console：输出到控制台；RFA：输出到日志文件
hbase.security.logger=DEBUG,console,RFAS        //hbase客户端安全相关的日志输出配置，console：输出到控制台；RFAS：输出到日志文件
hbase.log.dir=/var/log/Bigdata/hbase/client/    //日志路径，根据实际路径修改，但目录要有写入权限
hbase.log.file=hbase-client.log                 //日志文件名
hbase.log.level=INFO                            //日志级别，如果需要更详细的日志定位问题，需要修改为DEBUG，修改完需要重启进程才能生效
hbase.log.maxbackupindex=20                     //最多保存的日志文件数目
# Security audit appender
hbase.security.log.file=hbase-client-audit.log  //审计日志文件命令
```

12.3.2 HBase Rest 接口调用样例程序

12.3.2.1 使用 REST 接口查询 HBase 集群信息

功能简介

使用REST服务，传入对应host与port组成的url，通过HTTP协议，获取集群版本与状态信息。

代码样例

- 连接RestServer服务

  普通模式下，用户不需要登录即可连接RestServer服务。所以请将“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的main方法中与登录相关代码语句如下所示进行注释：

  ```
        //In Windows environment
        //String userdir = HBaseRestTest.class.getClassLoader().getResource("conf").getPath() + File.separator;
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
        //userKeytabFile = userdir + "user.keytab";
        //krb5File = userdir + "krb5.conf";
        //String principal = "hbaseuser1";
        //login(principal, userKeytabFile, krb5File);
  ```

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的main方法中。

  ```
  // RESTServer's hostname.
        String restHostName = "xxx.xxx.xxx.xxx";[1]
        String securityModeUrl = new StringBuilder("https://").append(restHostName).append(":21309").toString();
        String nonSecurityModeUrl = new StringBuilder("http://").append(restHostName).append(":21309").toString();
  ```

```
    HBaseRestTest test = new HBaseRestTest();

    //If cluster is non-security mode，use nonSecurityModeUrl as  parameter.
    test.test(nonSecurityModeUrl);[2]
```

[1]修改**restHostName**为待访问的RestServer实例所在节点的IP地址，并将节点IP配置到运行样例代码的本机hosts文件中。

[2]非安全模式采用http模式进行访问HBase REST服务，传入“nonSecurityModeUrl作为test.test()”参数。

- 获取集群版本信息

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getClusterVersion方法中。

```
private void getClusterVersion(String url) {
    String endpoint = "/version/cluster";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult((Optional<ResultModel>) result);
}
```

- 获取集群状态信息

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getClusterStatus方法中。

```
private void getClusterStatus(String url) {
    String endpoint = "/status/cluster";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

12.3.2.2 使用 REST 接口获取所有 HBase 表

功能简介

使用REST服务，传入对应host与port组成的url，通过HTTP协议，获取得到所有table。

代码样例

以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getAllUserTables方法中。

```
private void getAllUserTables(String url) {
    String endpoint = "/";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

12.3.2.3 使用 REST 接口操作 Namespace

功能简介

使用REST服务，传入对应host与port组成的url以及指定的Namespace，通过HTTP协议，对Namespace进行创建、查询、删除，获取指定Namespace中表的操作。

注意

HBase表以“*命名空间:表名*”格式进行存储，若在创建表时不指定命名空间，则默认存储在“default”中。其中，“hbase”命名空间为系统表命名空间，请不要对该系统表命名空间进行业务建表或数据读写等操作。

代码样例

- 方法调用

```
// Namespace operations.
createNamespace(url, "testNs");
getAllNamespace(url);
deleteNamespace(url, "testNs");
getAllNamespaceTables(url, "default");
```

- 创建namespace

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的createNamespace方法中。

```
private void createNamespace(String url, String namespace) {
    String endpoint = "/namespaces/" + namespace;
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.POST, null);
    if (result.orElse(new ResultModel()).getStatusCode() == HttpStatus.SC_CREATED) {
        LOG.info("Create namespace '{}' success.", namespace);
    } else {
        LOG.error("Create namespace '{}' failed.", namespace);
    }
}
```

- 查询所有namespace

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getAllNamespace方法中。

```
private void getAllNamespace(String url) {
    String endpoint = "/namespaces";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

- 删除指定namespace

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的deleteNamespace方法中。

```
private void deleteNamespace(String url, String namespace) {
    String endpoint = "/namespaces/" + namespace;
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.DELETE, null);
    if (result.orElse(new ResultModel()).getStatusCode() == HttpStatus.SC_OK) {
        LOG.info("Delete namespace '{}' success.", namespace);
    } else {
        LOG.error("Delete namespace '{}' failed.", namespace);
    }
}
```

- 根据指定namespace获取其中的表

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的getAllNamespaceTables方法中。

```
private void getAllNamespaceTables(String url, String namespace) {
    String endpoint = "/namespaces/" + namespace + "/tables";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult(result);
}
```

12.3.2.4 使用 REST 接口操作 HBase 表

功能简介

使用REST服务，传入对应host与port组成的url以及指定的tableName和jsonHTD，通过HTTP协议，进行查询表信息，修改表，创建表以及删除表的操作。

代码样例

- 方法调用

```
// Add a table with specified info.
createTable(url, "testRest",
    "{\"name\":\"default:testRest\",\"ColumnSchema\":[{\"name\":\"cf1\"}," + "{\"name\":\"cf2\"}]}");

// Add column family 'testCF1' if not exist, else update the 'VERSIONS' to 3.
 // Notes: The unspecified property of this column family will be updated to default value.
modifyTable(url, "testRest",
    "{\"name\":\"testRest\",\"ColumnSchema\":[{\"name\":\"testCF1\"," + "\"VERSIONS\":\"3" +
"\"}]}");

// Describe specific Table.
descTable(url, "default:testRest");

// delete a table with specified info.
deleteTable(url, "default:testRest",
    "{\"name\":\"default:testRest\",\"ColumnSchema\":[{\"name\":\"testCF\"," + "\"VERSIONS
\":\"3\"}]}");
```

- 查询表信息

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的descTable方法中。

```
private void descTable(String url, String tableName) {
    String endpoint = "/" + tableName + "/schema";
    Optional<ResultModel> result = sendAction(url + endpoint, MethodType.GET, null);
    handleNormalResult((Optional<ResultModel>) result);
}
```

- 修改表信息

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的modifyTable方法中。

```
private void modifyTable(String url, String tableName, String jsonHTD) {
    LOG.info("Start modify table.");
    String endpoint = "/" + tableName + "/schema";
    JsonElement tableDesc = new JsonParser().parse(jsonHTD);

    // Add a new column family or modify it.
    handleNormalResult(sendAction(url + endpoint, MethodType.POST, tableDesc));
}
```

- 创建表

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的createTable方法中。

```
private void createTable(String url, String tableName, String jsonHTD) {
    LOG.info("Start create table.");
    String endpoint = "/" + tableName + "/schema";
    JsonElement tableDesc = new JsonParser().parse(jsonHTD);

    // Add a table.
    handleCreateTableResult(sendAction(url + endpoint, MethodType.PUT, tableDesc));
}
```

- 删除表

  以下代码片段在“hbase-rest-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“HBaseRestTest”类的deleteTable方法中。

```
private void deleteTable(String url, String tableName, String jsonHTD) {
    LOG.info("Start delete table.");
    String endpoint = "/" + tableName + "/schema";
    JsonElement tableDesc = new JsonParser().parse(jsonHTD);

    // delete a table.
    handleNormalResult(sendAction(url + endpoint, MethodType.DELETE, tableDesc));
}
```

12.3.3 HBase ThriftServer 连接样例程序

12.3.3.1 通过 ThriftServer 实例操作 HBase 表

操作场景

传入ThriftServer实例所在host和提供服务的port，根据认证凭据及配置文件新建Thrift客户端，访问ThriftServer，进行根据指定namespace获取tablename以及创建表、删除表的操作。

操作步骤

步骤1 登录FusionInsight Manager，选择“集群 > 服务 > HBase > 配置 > 全部配置”，搜索并修改ThriftServer实例的配置参数“hbase.thrift.security.qop”。该参数值需与“hbase.rpc.protection”的值一一对应。保存配置，重启配置过期节点服务使更改的配置生效。

说明

“hbase.rpc.protection”与“hbase.thrift.security.qop”参数值的对应关系为：

- **"privacy" - "auth-conf"**
- **"authentication" - "auth"**
- **"integrity" - "auth-int"**

步骤2 获取集群中安装ThriftServer对应实例的配置文件。

- 方法一：选择“集群 > 服务 > HBase > 实例”，单击待操作的实例ThriftServer进入详情界面，获取配置文件“hdfs-site.xml”、“core-site.xml”、“hbase-site.xml”。
- 方法二：通过**准备本地应用开发环境**中解压客户端文件的方法获取配置文件，需要在获取的“hbase-site.xml”中手动添加以下配置，其中“hbase.thrift.security.qop”的参数值与**步骤1**保持一致。

```
<property>
<name>hbase.thrift.security.qop</name>
<value>auth</value>
</property>
<property>
<name>hbase.thrift.kerberos.principal</name>
<value>thrift/hadoop.hadoop.com@HADOOP.COM</value>
</property>
<property>
<name>hbase.thrift.keytab.file</name>
<value>/opt/huawei/Bigdata/FusionInsight_HD_8.1.0.1/install/FusionInsight-HBase-2.2.3/keytabs/HBase/thrift.keytab</value>
</property>
```

----结束

样例代码

- 初始化配置

  以下代码在“hbase-thrift-example”样例工程的“com.huawei.bigdata.hbase.examples”包的“TestMain”类中。

```
private static void init() throws IOException {
    // Default load from conf directory
    conf = HBaseConfiguration.create();
```

```
        String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
[1]
        //In Linux environment
        //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
        conf.addResource(new Path(userdir + "core-site.xml"), false);
        conf.addResource(new Path(userdir + "hdfs-site.xml"), false);
        conf.addResource(new Path(userdir + "hbase-site.xml"), false);
}
```

[1]**userdir**获取的是编译后资源路径下conf目录的路径。初始化配置用到的core-site.xml、hdfs-site.xml、hbase-site.xml文件，需要放置到"src/main/resources/conf"的目录下。

- 连接ThriftServer实例

  以下代码在“hbase-thrift-example”样例工程的“com.huawei.bigdata.hbase.examples”包的“TestMain”类中。

  ```
  try {
      test = new ThriftSample();
      test.test("xxx.xxx.xxx.xxx", THRIFT_PORT, conf);[2]
  } catch (TException | IOException e) {
      LOG.error("Test thrift error", e);
  }
  ```

  [2]test.test()传入参数为待访问的ThriftServer实例所在节点ip地址，需根据实际运行集群情况进行修改，且该节点ip需要配置到运行样例代码的本机hosts文件中。

  “THRIFT_PORT”为ThriftServer实例的配置参数"hbase.regionserver.thrift.port"对应的值。

- 方法调用

  ```
  // Get table of specified namespace.
  getTableNamesByNamespace(client, "default");
  // Create table.
  createTable(client, TABLE_NAME);
  // Delete specified table.
   deleteTable(client, TABLE_NAME);
  ```

- 根据指定namespace获取tablename

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的getTableNamesByNamespace方法中。

  ```
  private void getTableNamesByNamespace(THBaseService.Iface client, String namespace) throws
  TException {
      client.getTableNamesByNamespace(namespace)
          .forEach(
              tTableName -> LOGGER.info("{}", TableName.valueOf(tTableName.getNs(),
  tTableName.getQualifier())));
   }
  ```

- 创建表

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的createTable方法中。

  ```
  private void createTable(THBaseService.Iface client, String tableName) throws TException,
  IOException {
      TTableName table = getTableName(tableName);
      TTableDescriptor descriptor = new TTableDescriptor(table);
      descriptor.setColumns(
          Collections.singletonList(new
  TColumnFamilyDescriptor().setName(COLUMN_FAMILY.getBytes())));
      if (client.tableExists(table)) {
          LOGGER.warn("Table {} is exists, delete it.", tableName);
          client.disableTable(table);
          client.deleteTable(table);
      }
  ```

```
    client.createTable(descriptor, null);
    if (client.tableExists(table)) {
        LOGGER.info("Created {}.", tableName);
    } else {
        LOGGER.error("Create {} failed.", tableName);
    }
}
```

- 删除表

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的deleteTable方法中。

```
private void deleteTable(THBaseService.Iface client, String tableName) throws TException,
IOException {
    TTableName table = getTableName(tableName);
    if (client.tableExists(table)) {
        client.disableTable(table);
        client.deleteTable(table);
        LOGGER.info("Deleted {}.", tableName);
    } else {
        LOGGER.warn("{} not exist.", tableName);
    }
}
```

12.3.3.2 通过 ThriftServer 实例向 HBase 表中写入数据

功能简介

传入ThriftServer实例所在host和提供服务的port，根据认证凭据及配置文件新建Thrift客户端，访问ThriftServer，分别使用put和putMultiple进行写数据操作。

代码样例

- 方法调用

```
// Write data with put.
putData(client, TABLE_NAME);

// Write data with putlist.
 putDataList(client, TABLE_NAME);
```

- 使用put进行写数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的putData方法中。

```
private void putData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test putData.");
    TPut put = new TPut();
    put.setRow("row1".getBytes());

    TColumnValue columnValue = new TColumnValue();
    columnValue.setFamily(COLUMN_FAMILY.getBytes());
    columnValue.setQualifier("q1".getBytes());
    columnValue.setValue("test value".getBytes());
    List<TColumnValue> columnValues = new ArrayList<>(1);
    columnValues.add(columnValue);
    put.setColumnValues(columnValues);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    client.put(table, put);
    LOGGER.info("Test putData done.");
}
```

- 使用putMultiple进行写数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的putDataList方法中。

```
private void putDataList(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test putDataList.");
    TPut put1 = new TPut();
    put1.setRow("row2".getBytes());
    List<TPut> putList = new ArrayList<>();

    TColumnValue q1Value = new TColumnValue(ByteBuffer.wrap(COLUMN_FAMILY.getBytes()),
        ByteBuffer.wrap("q1".getBytes()), ByteBuffer.wrap("test value".getBytes()));
    TColumnValue q2Value = new TColumnValue(ByteBuffer.wrap(COLUMN_FAMILY.getBytes()),
        ByteBuffer.wrap("q2".getBytes()), ByteBuffer.wrap("test q2 value".getBytes()));
    List<TColumnValue> columnValues = new ArrayList<>(2);
    columnValues.add(q1Value);
    columnValues.add(q2Value);
    put1.setColumnValues(columnValues);
    putList.add(put1);

    TPut put2 = new TPut();
    put2.setRow("row3".getBytes());

    TColumnValue columnValue = new TColumnValue();
    columnValue.setFamily(COLUMN_FAMILY.getBytes());
    columnValue.setQualifier("q1".getBytes());
    columnValue.setValue("test q1 value".getBytes());
    put2.setColumnValues(Collections.singletonList(columnValue));
    putList.add(put2);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    client.putMultiple(table, putList);
    LOGGER.info("Test putDataList done.");
}
```

12.3.3.3 通过 ThriftServer 实例读 HBase 表数据

功能简介

传入ThriftServer实例所在host和提供服务的port，根据认证凭据及配置文件新建Thrift客户端，访问ThriftServer，分别使用get和scan进行读数据操作。

代码样例

- 方法调用

```
// Get data with single get.
getData(client, TABLE_NAME);

// Get data with getlist.
getDataList(client, TABLE_NAME);

// Scan data.
 scanData(client, TABLE_NAME);
```

- 使用get进行读数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的getData方法中。

```
private void getData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test getData.");
    TGet get = new TGet();
    get.setRow("row1".getBytes());

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    TResult result = client.get(table, get);
    printResult(result);
    LOGGER.info("Test getData done.");
}
```

- 使用getlist进行读数据

以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的getDataList方法中。

```
private void getDataList(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test getDataList.");
    List<TGet> getList = new ArrayList<>();
    TGet get1 = new TGet();
    get1.setRow("row1".getBytes());
    getList.add(get1);

    TGet get2 = new TGet();
    get2.setRow("row2".getBytes());
    getList.add(get2);

    ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
    List<TResult> resultList = client.getMultiple(table, getList);
    for (TResult tResult : resultList) {
        printResult(tResult);
    }
    LOGGER.info("Test getDataList done.");
}
```

- 使用scan进行读数据

  以下代码片段在“hbase-thrift-example\src\main\java\com\huawei\hadoop\hbase\examples”包的“ThriftSample”类的scanData方法中。

```
private void scanData(THBaseService.Iface client, String tableName) throws TException {
    LOGGER.info("Test scanData.");
    int scannerId = -1;
    try {
        ByteBuffer table = ByteBuffer.wrap(tableName.getBytes());
        TScan scan = new TScan();
        scan.setLimit(500);
        scannerId = client.openScanner(table, scan);
        List<TResult> resultList = client.getScannerRows(scannerId, 100);
        while (resultList != null && !resultList.isEmpty()) {
            for (TResult tResult : resultList) {
                printResult(tResult);
            }
            resultList = client.getScannerRows(scannerId, 100);
        }
    } finally {
        if (scannerId != -1) {
            client.closeScanner(scannerId);
            LOGGER.info("Closed scanner {}.", scannerId);
        }
    }
    LOGGER.info("Test scanData done.");
}
```

12.3.4 HBase 访问多个 ZooKeeper 样例程序

功能简介

在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper，其中HBase客户端访问FusionInsight ZooKeeper，客户应用访问第三方ZooKeeper。

代码样例

以下代码片段在“hbase-zk-example\src\main\java\com\huawei\hadoop\hbase\example”包的“TestZKSample”类中，用户主要需要关注“login”和“connectApacheZK”这两个方法。

```
private static void login(String keytabFile, String principal) throws IOException {
    conf = HBaseConfiguration.create();
```

```
    //In Windows environment
    String confDirPath = TestZKSample.class.getClassLoader().getResource("").getPath() + File.separator;
[1]
    //In Linux environment
    //String confDirPath = System.getProperty("user.dir") + File.separator + "conf" + File.separator;

    // Set zoo.cfg for hbase to connect to fi zookeeper.
    conf.set("hbase.client.zookeeper.config.path", confDirPath + "zoo.cfg");
    if (User.isHBaseSecurityEnabled(conf)) {
        // jaas.conf file, it is included in the client pakcage file
        System.setProperty("java.security.auth.login.config", confDirPath + "jaas.conf");
        // set the kerberos server info,point to the kerberosclient
        System.setProperty("java.security.krb5.conf", confDirPath + "krb5.conf");
        // set the keytab file name
        conf.set("username.client.keytab.file", confDirPath + keytabFile);
        // set the user's principal
        try {
            conf.set("username.client.kerberos.principal", principal);
            User.login(conf, "username.client.keytab.file", "username.client.kerberos.principal",
                InetAddress.getLocalHost().getCanonicalHostName());
        } catch (IOException e) {
            throw new IOException("Login failed.", e);
        }
    }
  }
  private void connectApacheZK() throws IOException, org.apache.zookeeper.KeeperException {
    try {
        // Create apache zookeeper connection.
        ZooKeeper digestZk = new ZooKeeper("127.0.0.1:2181", 60000, null);
        LOG.info("digest directory：{}", digestZk.getChildren("/", null));
        LOG.info("Successfully connect to apache zookeeper.");
    } catch (InterruptedException e) {
        LOG.error("Found error when connect apache zookeeper ", e);
    }
  }
```

- [1]**userdir**获取的是编译后资源目录的路径。将初始化需要的配置文件“core-site.xml”、“hdfs-site.xml”、“hbase-site.xml”放置到"src/main/resources"的目录下。
- “login”方法中的参数“java.security.auth.login.config”设置的jaas.conf文件用来设置访问ZooKeeper相关认证信息，样例代码中包含Client_new和Client两部分，Client_new的配置用来访问FusionInsight ZooKeeper，Client用来访问Apache ZooKeeper。
- “login”方法中的参数“hbase.client.zookeeper.config.path”用来设置访问FusionInsight ZooKeeper客户端的配置，主要涉及如下三个参数：
  - zookeeper.sasl.clientconfig：指定使用jaas.conf文件中的对应配置访问FusionInsight ZooKeeper；
  - zookeeper.server.principal：指定ZooKeeper服务端使用principal，格式为“zookeeper/hadoop.*系统域名*”，例如：zookeeper/hadoop.HADOOP.COM。系统域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数值获取。
  - zookeeper.sasl.client：如果集群是安全模式，该值设置为“true”，否则设置为“false”，设置为“false”的情况下，“zookeeper.sasl.clientconfig”和“zookeeper.server.principal”参数不生效。

12.4 调测 HBase 应用

12.4.1 在本地 Windows 环境中调测 HBase 应用

操作场景

在程序代码完成开发后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

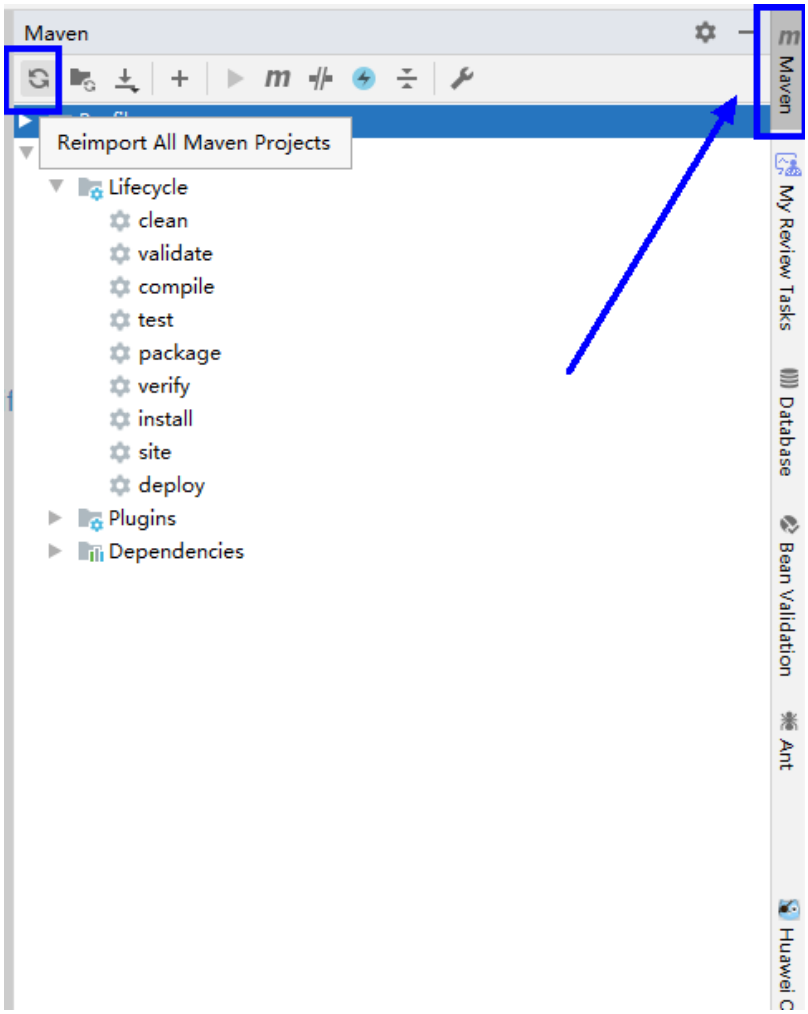
说明

- 如果Windows开发环境中使用IBM JDK，不支持在Windows环境中直接运行应用程序。
- 需要在运行样例代码的本机**hosts**文件中设置访问节点的主机名和公网IP地址映射，主机名和公网IP地址请保持一一对应。

在本地 Windows 环境中调测 HBase 应用

步骤1 单击IDEA右边Maven窗口的“Reimport All Maven Projects”，进行maven项目依赖import。

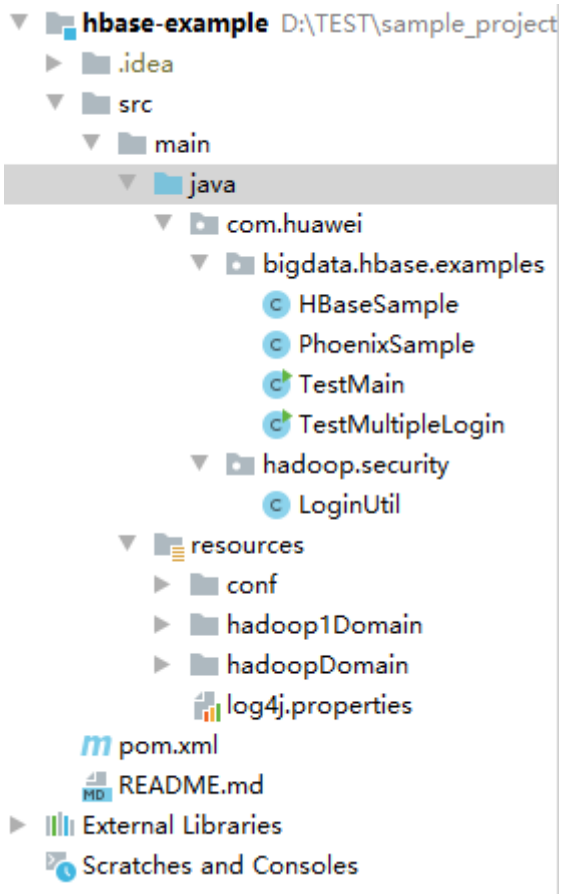
图 12-16 reimport projects



步骤2 编译运行程序。

放置好配置文件，并修改代码匹配登录用户后，文件列表如图12-17所示。

图 12-17 hbase-example 待编译目录列表

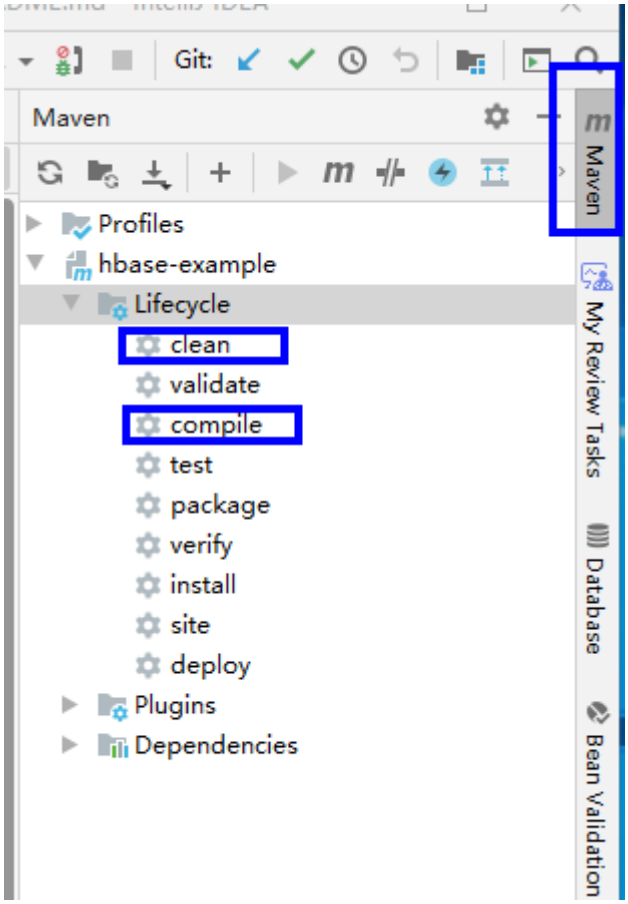


1. 编译方式有以下两种。
   - 方法一

     选择“Maven > *样例工程名称* > Lifecycle > clean”，双击“clean”运行maven的clean命令。

     选择“Maven > *样例工程名称* > Lifecycle > compile”，双击“compile”运行maven的compile命令。

图 12-18 mavne 工具 clean 和 compile



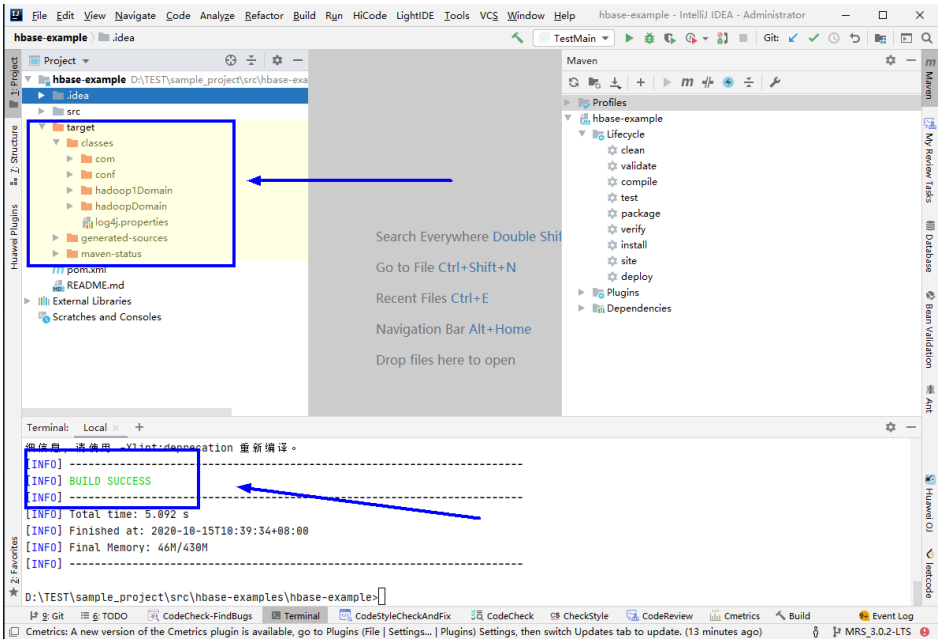
- 方法二

  在IDEA的下方Terminal窗口进入“pom.xml”所在目录，手动输入**mvn clean compile**命令进行编译。

图 12-19 idea termial 输入“mvn clean compile”

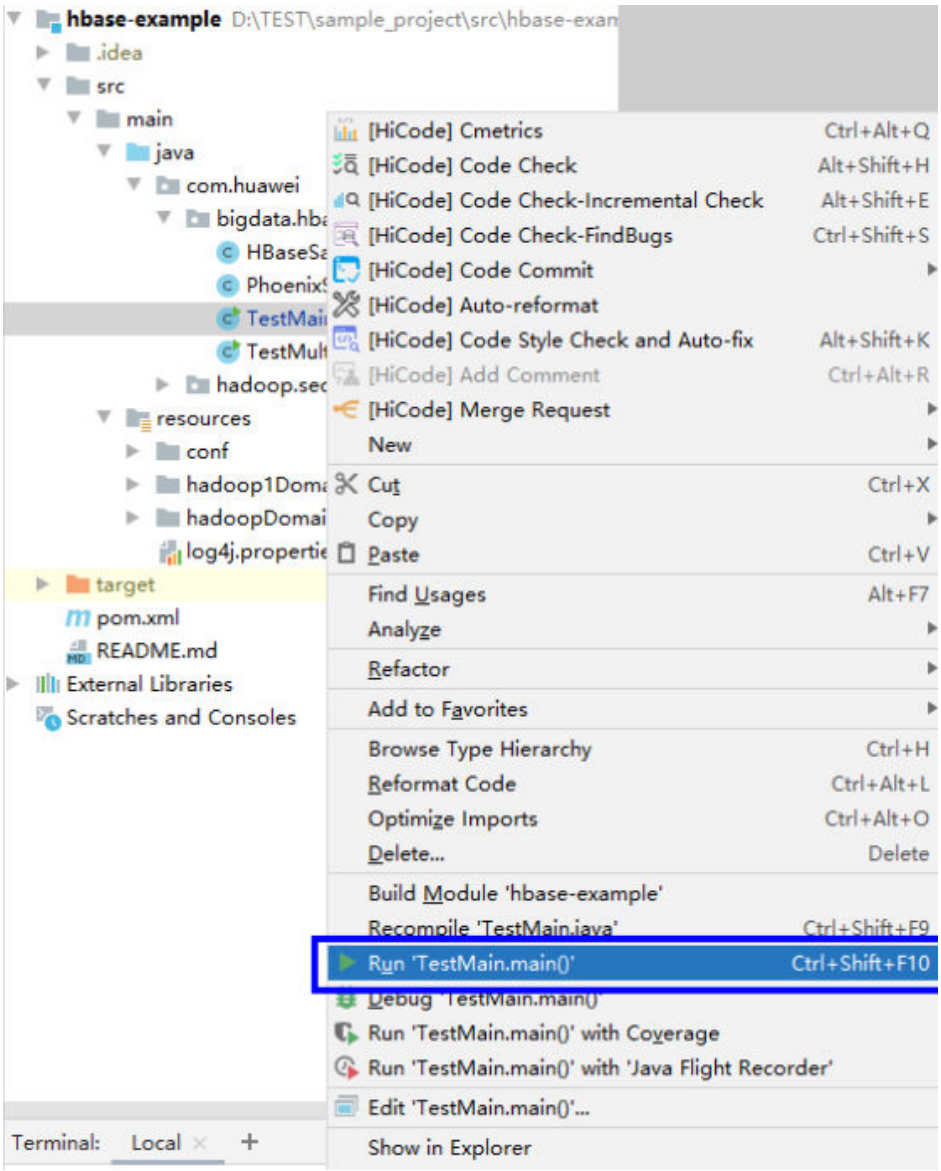
编译完成，打印“Build Success”，生成target目录。

图 12-20 编译完成



2. 运行程序。
   右键“TestMain.java”文件，选择“Run 'TestMain.main()”

图 12-21 运行程序



----结束

查看 Windows 调测结果

HBase应用程序运行完成后，可通过如下方式查看运行情况。

- 通过IntelliJ IDEA运行结果查看应用程序运行情况。
- 通过HBase日志获取应用程序运行情况。
- 登录HBase WebUI查看应用程序运行情况。可参见“更多信息 > 对外接口 > Web UI ”。
- 通过HBase shell命令查看应用程序运行情况。可参见“更多信息 > 对外接口 > Shell ”。

各样例程序运结果如下：

- HBase数据读写样例运行成功会有如下信息：


```
2016-07-13 14:36:12,736 INFO  [main] basic.CreateTableSample: Create table
sampleNameSpace:sampleTable successful!
2016-07-13 14:36:15,426 INFO  [main] basic.ModifyTableSample: Modify table
sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:16,708 INFO  [main] basic.MultiSplitSample: Mmulti split table
sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:17,299 INFO  [main] basic.PutDataSample: Successfully put 9 items data into
sampleNameSpace:sampleTable.
2016-07-13 14:36:18,992 INFO  [main] basic.ScanSample: Scan data successfully.
2016-07-13 14:36:20,532 INFO  [main] basic.DeletaDataSample: Successfully delete data from table
sampleNameSpace:sampleTable.
2016-07-13 14:36:21,006 INFO  [main] acl.AclSample: Grant ACL for table
sampleNameSpace:sampleTable successfully.
2016-07-13 14:36:27,836 INFO  [main] index.CreateIndexSample: Successfully add index for table
sampleNameSpace:sampleTable.
```

说明

在Windows环境运行样例代码时会出现下面的异常，但是不影响业务：

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

- 日志说明

  日志级别默认为INFO，可以通过调整日志打印级别（DEBUG，INFO，WARN，ERROR，FATL）来显示更详细的信息。可以通过修改log4j.properties文件来实现，如：

```
hbase.root.logger=INFO,console
...
log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=INFO
# Make these two classes DEBUG-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
...
```

12.4.2 在 Linux 环境中调测 HBase 应用

操作场景

HBase应用程序支持在已安装或未安装HBase客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至Linux环境中运行应用。

前提条件

- 已安装客户端时：
  - 已安装HBase客户端。
  - 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。
- 未安装HBase客户端时：
  - Linux环境已安装JDK，版本号需要和IntelliJ IDEA导出Jar包使用的JDK版本一致。
  - 当Linux环境所在主机不是集群中的节点时，需要在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

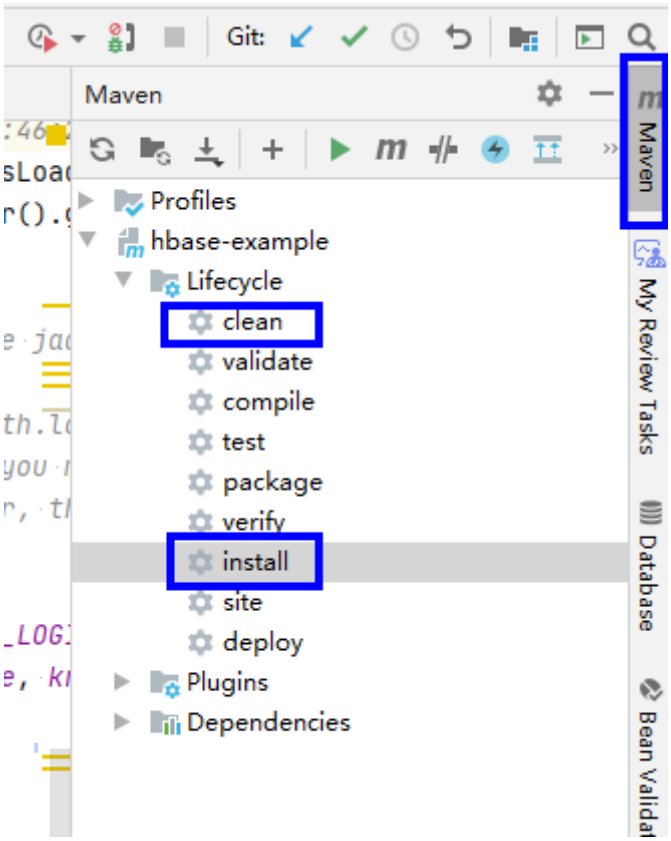
已安装客户端时编译并运行程序

步骤1 导出Jar包。

构建jar包方式有以下两种：

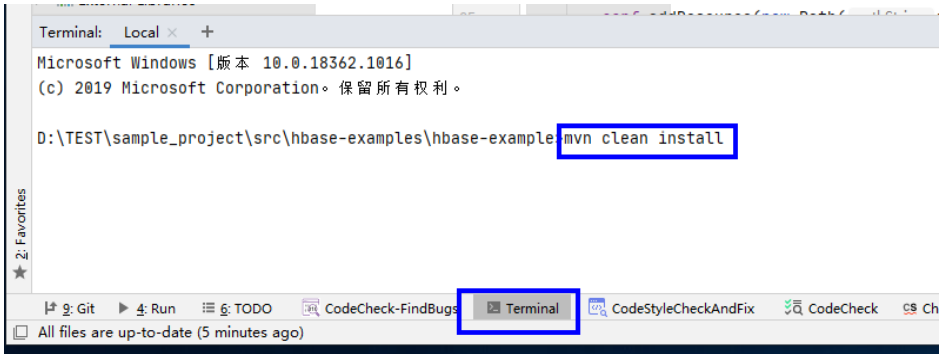
- 方法一：
  选择“Maven > *样例工程名称* > Lifecycle > clean”，双击“clean”运行maven的clean命令。
  选择“Maven > *样例工程名称* > Lifecycle > install”，双击“install”运行maven的install命令。

  图 12-22 maven 工具 clean 和 install

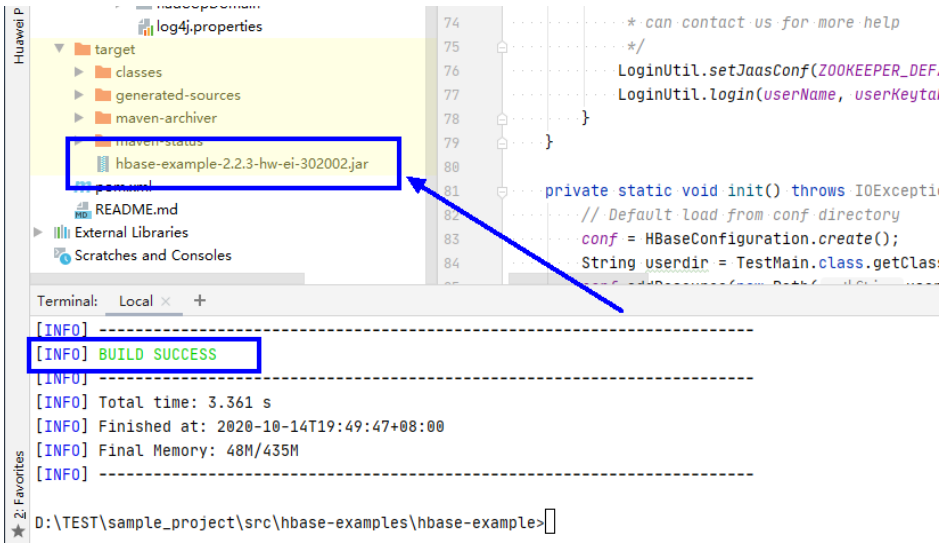
- 方法二：在IDEA的下方Terminal窗口进入“pom.xml”所在目录，手动输入**mvn clean install**命令进行编译。

  图 12-23 idea termial 输入“mvn clean install”

编译完成，打印“Build Success”，生成target目录，生成jar包在target目录中。

图 12-24 编译完成，生成 jar 包



步骤2 导出样例项目依赖的jar包。

在IDEA的下方Terminal窗口或其他命令行工具进入“pom.xml”所在目录。

执行命令**mvn dependency:copy-dependencies -DoutputDirectory=lib**。

在“pom.xml”所在目录将生成lib文件夹，其中包含样例项目所依赖的jar包。

步骤3 执行Jar包。

1. 在Linux客户端下执行Jar包的时候，需要用安装用户切换到客户端目录：

   cd *$BIGDATA_CLIENT_HOME*

   说明

   “$BIGDATA_CLIENT_HOME”为HBase客户端安装目录，例如“/opt/client”。

2. 然后执行：

   source bigdata_env

   说明

   启用多实例功能后，为其他HBase服务实例进行应用程序开发时还需执行以下命令，切换指定服务实例的客户端。

   例如HBase2：**source /opt/client/HBase2/component_env**。

3. 将应用开发环境中生成的样例项目Jar包（非依赖jar包）上传至客户端运行环境的“*客户端安装目录*/HBase/hbase/lib”目录，还需将**表12-5**获取的例工程所需的配置文件复制到“*客户端安装目录*/HBase/hbase/conf”目录。

4. 进入目录“$BIGDATA_CLIENT_HOME/HBase/hbase”，执行以下命令运行Jar包。

   hbase *com.huawei.bigdata.hbase.examples.TestMain*

   其中，*hbase com.huawei.bigdata.hbase.examples.TestMain*为举例，具体以实际样例代码为准。

----结束

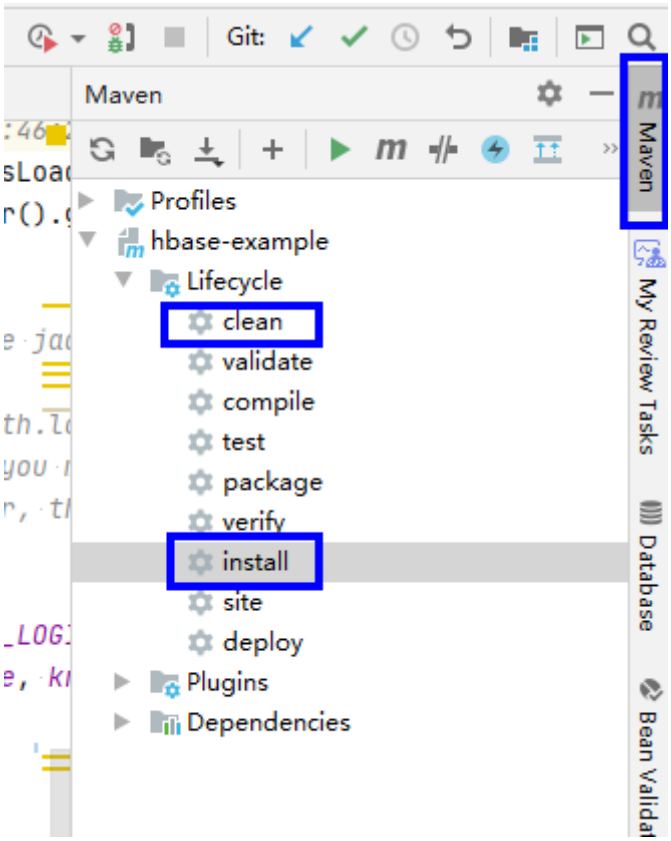
未安装客户端时编译并运行程序

步骤1 导出Jar包。

构建jar包方式有以下两种：

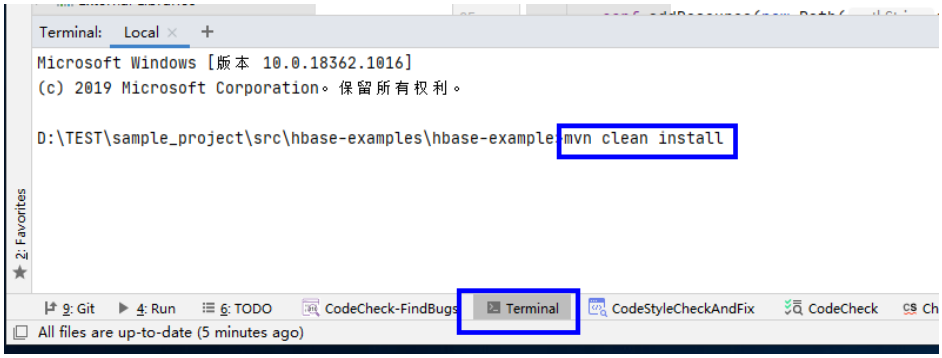
- 方法一：
  选择“Maven > *样例工程名称* > Lifecycle > clean”，双击“clean”运行maven的clean命令。
  选择“Maven > *样例工程名称* > Lifecycle > install”，双击“install”运行maven的install命令。

  图 12-25 maven 工具 clean 和 install

- 方法二：在IDEA的下方Terminal窗口进入“pom.xml”所在目录，手动输入**mvn clean install**命令进行编译。

  图 12-26 idea termial 输入“mvn clean install”

编译完成，打印“Build Success”，生成target目录，生成jar包在target目录中。

图 12-27 编译完成，生成 jar 包

步骤2 准备依赖的Jar包和配置文件。

1. 在Linux环境新建目录，例如“/opt/test”，并创建子目录“lib”和“conf”。将样例工程中“lib”的Jar包导出，导出步骤请参考**步骤2**，以及**步骤1**导出的Jar包，上传到Linux的“lib”目录。将样例工程中“conf”的配置文件上传到Linux中“conf”目录。
2. 在“/opt/test”根目录新建脚本“run.sh”，修改内容如下并保存：

```
#!/bin/sh
BASEDIR=`cd $(dirname $0);pwd`
cd ${BASEDIR}
for file in ${BASEDIR}/lib/*.jar
do
i_cp=$i_cp:$file
echo "$file"
done
for file in ${BASEDIR}/conf/*
do
i_cp=$i_cp:$file
done

java -cp .${i_cp} com.huawei.bigdata.hbase.examples.TestMain
```

步骤3 切换到“/opt/test”，执行以下命令，运行Jar包。

sh run.sh

----结束

查看 Linux 调测结果

HBase应用程序运行完成后，可通过如下方式查看应用程序的运行情况。

- 通过运行结果查看应用程序运行情况。
- 通过HBase日志获取应用程序运行情况。
- 登录HBase WebUI查看应用程序运行情况。可参见“更多信息 > 对外接口 > Web UI ”。

- 通过HBase shell命令查看应用程序运行情况。可参见“更多信息 > 对外接口 > Shell ”。

通过运行日志可查看应用提交后的执行详情，例如，hbase-example样例运行成功后，显示信息如下：

```
2020-07-13 14:36:12,736 INFO  [main] basic.CreateTableSample: Create table
sampleNameSpace:sampleTable successful!
2020-07-13 14:36:15,426 INFO  [main] basic.ModifyTableSample: Modify table
sampleNameSpace:sampleTable successfully.
2020-07-13 14:36:16,708 INFO  [main] basic.MultiSplitSample: Mmulti split table
sampleNameSpace:sampleTable successfully.
2020-07-13 14:36:17,299 INFO  [main] basic.PutDataSample: Successfully put 9 items data into
sampleNameSpace:sampleTable.
2020-07-13 14:36:18,992 INFO  [main] basic.ScanSample: Scan data successfully.
2020-07-13 14:36:20,532 INFO  [main] basic.DeletaDataSample: Successfully delete data from table
sampleNameSpace:sampleTable.
2020-07-13 14:36:21,006 INFO  [main] acl.AclSample: Grant ACL for table sampleNameSpace:sampleTable
successfully.
2020-07-13 14:36:27,836 INFO  [main] index.CreateIndexSample: Successfully add index for table
sampleNameSpace:sampleTable.
```

12.5 HBase 应用开发常见问题

12.5.1 Phoenix SQL 查询样例介绍

功能简介

Phoenix是构建在HBase之上的一个SQL中间层，提供一个客户端可嵌入的JDBC驱动，Phoenix查询引擎将SQL输入转换为一个或多个HBase scan，编译并执行扫描任务以产生一个标准的JDBC结果集。

代码样例

- 客户端“hbase-example/conf/hbase-site.xml”中配置存放查询中间结果的临时目录，如果客户端程序在Linux上执行临时目录就配置Linux上的路径，如果客户端程序在Windows上执行临时目录则配Windows上的路径。

```
<property>
    <name>phoenix.spool.directory</name>
    <value>[1]查询中间结果的临时目录</value>
</property>
```

- JAVA样例：使用JDBC接口访问HBase

```
        public String getURL(Configuration conf)
        {
           String phoenix_jdbc = "jdbc:phoenix";
           String zkQuorum = conf.get("hbase.zookeeper.quorum");
           return phoenix_jdbc + ":" + zkQuorum;
        }

        public void testSQL()
        {
           String tableName = "TEST";
           // Create table
           String createTableSQL = "CREATE TABLE IF NOT EXISTS TEST(id integer not null primary key,
name varchar, account char(6), birth date)";

           // Delete table
           String dropTableSQL = "DROP TABLE TEST";

           // Insert
```

```
        String upsertSQL = "UPSERT INTO TEST VALUES(1,'John','100000',
TO_DATE('1980-01-01','yyyy-MM-dd'))";

        // Query
        String querySQL = "SELECT * FROM TEST WHERE id = ?";

        // Create the Configuration instance
        Configuration conf = getConfiguration();

        // Get URL
        String URL = getURL(conf);

        Connection conn = null;
        PreparedStatement preStat = null;
        Statement stat = null;
        ResultSet result = null;

        try
        {
            // Create Connection
            conn = DriverManager.getConnection(URL);
            // Create Statement
            stat = conn.createStatement();
            // Execute Create SQL
            stat.executeUpdate(createTableSQL);
            // Execute Update SQL
            stat.executeUpdate(upsertSQL);
            // Create PrepareStatement
            preStat = conn.prepareStatement(querySQL);
            conn.commit();
            // Execute query
            preStat.setInt(1,1);
            result = preStat.executeQuery();
            // Get result
            while (result.next())
            {
                int id = result.getInt("id");
                String name = result.getString(1);
            }
        }
        catch (Exception e)
        {
            // handler exception
        }
        finally
        {
            if(null != result){
                try {
                    result.close();
                    } catch (Exception e2) {
                        // handler exception
                        }
                }
            if(null != stat){
                try {
                    stat.close();
                } catch (Exception e2) {
                    // handler exception
                }
            }
            if(null != conn){
                try {
                    conn.close();
                } catch (Exception e2) {
                    // handler exception
                }
            }
        }
    }
```

注意事项

- 需要在“hbase-site.xml”中配置用于存放中间查询结果的临时目录路径，该目录大小限制可查询结果集大小。
- Phoenix实现了大部分java.sql接口，SQL紧跟ANSI SQL标准。

12.5.2 HBase 对外接口介绍

12.5.2.1 HBase Shell 接口介绍

您可以使用Shell在服务端直接对HBase进行操作。HBase的Shell接口同开源社区版本保持一致，请参见**http://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/**

Shell命令执行方法：

进入HBase客户端任意目录，执行以下命令。

hbase shell

进入HBase命令行运行模式（也称为CLI客户端连接），如下所示。

```
hbase(main):001:0>
```

您可以在命令行运行模式中运行***help***命令获取HBase的命令参数的帮助信息。

注意事项

count命令不支持条件统计，仅支持全表统计。

获取 HBase replication 指标的命令

通过Shell命令“status”可以获取到所有需要的指标。

- 查看replication source指标的命令。

  hbase(main):019:0> status 'replication', 'source'

  输出结果如下：

  ```
  version 2.2.3
  1 live servers
  BLR1000006595:
  SOURCE: PeerID=1, SizeOfLogQueue=0, ShippedBatches=0, ShippedOps=0, ShippedBytes=0,
  LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4, SizeOfLogToReplicate=0,
  TimeForLogToReplicate=0, ShippedHFiles=0,
  SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
  20:44:42 CST 2016, Replication Lag=0 PeerID=3, SizeOfLogQueue=0, ShippedBatches=0,
  ShippedOps=0, ShippedBytes=0, LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4,
  SizeOfLogToReplicate=0, TimeForLogToReplicate=0, ShippedHFiles=0,
  SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
  20:44:42 CST 2016, Replication Lag=0 FailedReplicationAttempts=0
  ```

- 查看replication sink指标的命令。

  hbase(main):020:0> status 'replication', 'sink'

  输出结果如下：

  ```
  version 2.2.3
  1 live servers
  BLR1000006595:
  SINK : AppliedBatches=0, AppliedOps=0, AppliedHFiles=0, AgeOfLastAppliedOp=0,
  TimeStampsOfLastAppliedOp=Wed May 25 17:55:21 CST 2016
  ```

- 同时查看replication source和replication sink指标的命令。

  hbase(main):018:0> status 'replication'

  输出结果如下：

```
version 2.2.3
1 live servers
BLR1000006595:
SOURCE: PeerID=1, SizeOfLogQueue=0, ShippedBatches=0, ShippedOps=0, ShippedBytes=0,
LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4, SizeOfLogToReplicate=0,
TimeForLogToReplicate=0, ShippedHFiles=0,
SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
20:43:24 CST 2016, Replication Lag=0 PeerID=3, SizeOfLogQueue=0, ShippedBatches=0,
ShippedOps=0, ShippedBytes=0, LogReadInBytes=1389, LogEditsRead=4, LogEditsFiltered=4,
SizeOfLogToReplicate=0, TimeForLogToReplicate=0, ShippedHFiles=0,
SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0, TimeStampsOfLastShippedOp=Wed May 25
20:43:24 CST 2016, Replication Lag=0 FailedReplicationAttempts=0
SINK  : AppliedBatches=0, AppliedOps=0, AppliedHFiles=0, AgeOfLastAppliedOp=0,
TimeStampsOfLastAppliedOp=Wed May 25 17:55:21 CST 2016
```

12.5.2.2 HBase Java API 接口介绍

接口使用建议

- 建议使用org.apache.hadoop.hbase.Cell作为KV数据对象，而不是org.apache.hadoop.hbase.KeyValue。
- 建议使用Connection connection = ConnectionFactory.createConnection(conf)来创建连接，废弃HTablePool。
- 建议使用org.apache.hadoop.hbase.mapreduce，不建议使用org.apache.hadoop.hbase.mapred。
- 建议通过构造出来的Connection对象的getAdmin()方法来获取HBase的客户端操作对象。

HBase 常用接口介绍

HBase常用的Java类有以下几个：

- 接口类Admin，HBase客户端应用的核心类，主要封装了HBase管理类操作的API，例如建表，删表等操作，部分常见接口参见表**表12-9**。
- 接口类Table，HBase读写操作类，主要封装了HBase表的读写操作的API，部分常见接口参见表**表12-10**。

表 12-9 org.apache.hadoop.hbase.client.Admin

方法	描述
boolean tableExists(final TableName tableName)	通过该方法可以判断指定的表是否存在。如果hbase:meta表中存在该表则返回true，否则返回false。
HTableDescriptor[] listTables(String regex)	查看匹配指定正则表达式格式的用户表。该方法还有另外两个重载的方法，一个入参类型为Pattern；一个入参为空，默认查看所有用户表。

方法	描述
HTableDescriptor[] listTables(final Pattern pattern, final boolean includeSysTables)	作用与上一个方法类似，用户可以通过该方法指定返回的结果是否包含系统表，上一个接口只返回用户表。
TableName[] listTableNames(String regex)	查看匹配指定正则表达式格式的用户表。该方法还有另外两个重载的方法，一个入参类型为Pattern；一个入参为空，默认查看所有用户表。该方法的作用与listTables类似，只是该方法返回类型为TableName[]。
TableName[] listTableNames(final Pattern pattern, final boolean includeSysTables)	作用与上一个方法类似，用户可以通过该方法指定返回的结果是否包含系统表，上一个方法只返回用户表。
void createTable(HTableDescriptor desc)	创建一个只有一个region的表。
void createTable(HTableDescriptor desc, byte[] startKey, byte[] endKey, int numRegions)	创建一个有指定数量region的表，其中第一个region的endKey为参数中的startKey，最后一个region的StartKey为参数中的endKey。如果region的数量过多，该方法可能调用超时。
void createTable(final HTableDescriptor desc, byte[][] splitKeys)	创建一个表，该表的region数量以及每个region的startKey由splitKeys决定。如果region的数量过多，该方法可能调用超时。
void createTable(final HTableDescriptor desc, final byte[][] splitKeys)	创建一个表，该表的region数量以及每个region的startKey由splitKeys决定。该方法为异步调用，不会等待创建的表上线。
void deleteTable(final TableName tableName)	删除指定的表。
public void truncateTable(final TableName tableName, final boolean preserveSplits)	重建指定的表，如果第二个参数为true，重建后的表region与之前保持一致，否则只有一个region。
void enableTable(final TableName tableName)	启用指定的表。如果表的region数量过多，该方法可能调用超时。
void enableTableAsync(final TableName tableName)	启用指定的表。该方法为异步调用，不会等待所有region上线后才返回。
void disableTable(final TableName tableName)	禁用指定的表。如果表的region数量过多，该方法可能调用超时。
void disableTableAsync(final TableName tableName)	禁用指定的表。该方法为异步调用，不会等待所有region下线后才返回。

方法	描述
boolean isTableEnabled(TableName tableName)	判断表是否已启用。该方法可以配合enableTableAsync方法使用来判断一个启用表的操作是否完成。
boolean isTableDisabled(TableName tableName)	判断表是否已禁用。该方法可以配合disableTableAsync方法使用来判断一个禁用表的操作是否完成。
void addColumn(final TableName tableName, final HColumnDescriptor column)	添加一个列簇到指定的表。
void deleteColumn(final TableName tableName, final HColumnDescriptor column)	从指定的表删除指定的列簇。
void modifyColumn(final TableName tableName, final HColumnDescriptor column)	修改指定的列簇。

表 12-10 org.apache.hadoop.hbase.client.Table

方法	描述
boolean exists(Get get)	判断指定的rowkey在表中是否存在。
boolean[] existsAll(List<Get> gets)	判断这批指定的rowkey在表中是否存在，返回的boolean数组结果与入参位置一一对应。
Result get(Get get)	通过指定的rowkey读取数据。
Result[] get(List<Get> gets)	通过指定一批rowkey的方式批量读取数据。
ResultScanner getScanner(Scan scan)	获取该表的一个Scanner对象，查询相关的参数可以通过入参scan指定，包括StartRow，StopRow，caching大小等。
void put(Put put)	往该表写入一条数据。
void put(List<Put> puts)	往该表写入一批数据。
void close()	释放该Table对象持有的资源。

说明

表12-9和表12-10只列举了部分常用的方法。

12.5.2.3 Sqlline 接口介绍

你可以直接使用sqlline.py在服务端对HBase进行SQL操作。Phoenix的sqlline接口与开源社区保持一致，请参见**http://phoenix.apache.org/**。

Sqlline常用语法见**表12-11**，常用函数见**表12-12**，命令行使用可以参考**Phoenix命令行操作介绍**章节。

表 12-11 Sqlline 常用语法

命令	描述	示例
CREATE TABLE	创建表。	CREATE TABLE MY_TABLE(id BIGINT not null primary key, name VARCHAR);
ALTER	修改表/视图。	ALTER TABLE MY_TABLE DROP COLUMN name;
DROP TABLE	删除表。	DROP TABLE MY_TABLE;
UPSERT VALUES	插入/修改数据。	UPSERT INTO MY_TABLE VALUES(1,'abc');
SELECT	查询数据。	SELECT * FROM MY_TABLE;
CREATE INDEX	创建全局索引。	CREATE INDEX MY_IDX ON MY_TABLE(name);
CREATE LOCAL INDEX	创建局部索引。	CREATE LOCAL INDEX MY_LOCAL_IDX ON MY_TABLE(id,name);
ALTER INDEX	修改索引状态。	ALTER INDEX MY_IDX ON MY_TABLE DISABLE;
DROP INDEX	删除索引。	DROP INDEX MY_IDX ON MY_TABLE;
EXPLAIN	显示执行计划。	EXPLAIN SELECT name FROM MY_TABLE;
CREATE SEQUENCE	创建序列。	CREATE SEQUENCE MY_SEQUENCE;
DROP SEQUENCE	删除序列。	DROP SEQUENCE MY_SEQUENCE;
CREATE VIEW	创建视图。	CREATE VIEW MY_VIEW AS SELECT * FROM MY_TABLE;
DROP VIEW	删除视图。	DROP VIEW MY_VIEW;
CREATE SCHEMA	创建SCHEMA。	CREATE SCHEMA MY_SCHEMA;
USE	修改默认SCHEMA。	USE MY_SCHEMA;
DROP SCHEMA	删除SCHEMA。	DROP SCHEMA MY_SCHEMA;

表 12-12 Sqlline 常用函数

函数类型	函数	描述	示例
字符串	**SUBSTR**	字符串截取。	SUBSTR('[Hello]', 2, 5)
	INSTR	查询子串位置。	INSTR('Hello World', 'World')
	TRIM	去除前后空格。	TRIM(' Hello ')
	LTRIM	去除左空格。	LTRIM(' Hello')
	RTRIM	去除右空格。	RTRIM('Hello ')
	LPAD	字符串左侧填充。	LPAD('John',30,'*')
	LENGTH	获取字符串长度。	LENGTH('Hello')
	UPPER	字符串转大写。	UPPER('Hello')
	LOWER	字符串转小写。	LOWER('HELLO')
	REVERSE	字符串反转。	REVERSE('Hello')
	REGEXP_SPLIT	字符串分割。	REGEXP_SPLIT('ONE,TWO,THREE', ',')
	REGEXP_REPLACE	字符串替换。	REGEXP_REPLACE('abc123ABC', '[0-9]+', '#')
	REGEXP_SUBSTR	获取正则子串。	REGEXP_SUBSTR('na1-appsrv35-sj35', '[^-]+')
聚合	**AVG**	获取平均数。	AVG(X)
	COUNT	获取数据条数。	COUNT(*)
	MAX	获取最大值。	MAX(NAME)
	MIN	获取最小值。	MIN(NAME)
	SUM	数字求合。	SUM(X)
	STDDEV_POP	标准差。	STDDEV_POP(X)
	STDDEV_SAMP	样板标准差。	STDDEV_SAMP(X)
	NTH_VALUE	分组后的第几个值。	NTH_VALUE(name, 2) WITHIN GROUP (ORDER BY salary DESC)
时间	**NOW**	获取当前时间(DATE类型)。	NOW()
	CURRENT_TIME	获取当前时间(TIME类型)。	CURRENT_TIME()
	CURRENT_DATE	获取当前时间(DATE类型)。	CURRENT_DATE()

函数类型	函数	描述	示例
	TO_DATE	字符串转DATE类型。	TO_DATE('1970-01-01', 'yyyy-MM-dd', 'GMT+1')
	TO_TIME	字符串转TIME类型。	TO_TIME('1970-01-01', 'yyyy-MM-dd', 'GMT+1')
	TO_TIMESTAMP	字符串转TIMESTAMP类型。	TO_TIMESTAMP('1970-01-01', 'yyyy-MM-dd', 'GMT+1')
	YEAR	获取年。	YEAR(TO_DATE('2015-6-05'))
	MONTH	获取月。	MONTH(TO_TIMESTAMP('2015-6-05'))
	WEEK	获取星期。	WEEK(TO_TIME('2010-6-15'))
	HOUR	获取小时。	HOUR(TO_TIMESTAMP('2015-6-05'))
	MINUTE	获取分钟。	MINUTE(TO_TIME('2015-6-05'))
	SECOND	获取秒。	SECOND(TO_DATE('2015-6-05'))
数字	**ROUND**	四舍五入（也可用于时间）。	ROUND(2.56)
	CEIL	向上取整。	CEIL(2.34)
	FLOOR	向下取整。	FLOOR(2.34)
	TRUNC	向下取整（与**FLOOR**相同）。	TRUNC(2.34)
	TO_NUMBER	字符串转数字。	TO_NUMBER('-123.33')
	RAND	获取随机数。	RAND()
数学	**ABS**	求绝对值。	ABS(-1)
	SQRT	$\sqrt{x}$	SQRT(1.1)
	EXP	e^x	EXP(-1)
	POWER	x^y	POWER(2, 3)
	LN	$ln(x)$	LN(3)

函数类型	函数	描述	示例
	LOG	$log_x(y)$	LOG(2, 3)
数组	**ARRAY_ELEM**	通过下标访问数组。	ARRAY_ELEM(ARRAY[1,2,3], 1)
	ARRAY_PREPEND	指定位置插入数据到数组。	ARRAY_APPEND(ARRAY[1,2,3], 4)
	ARRAY_CAT	连接数组。	ARRAY_CAT(ARRAY[1,2], ARRAY[3,4])
	ARRAY_FILL	数组填充。	ARRAY_FILL(1, 3)
	ARRAY_TO_STRING	数组转字符串。	ARRAY_TO_STRING(ARRAY['a','b','c'], ',')
	ANY	数组任意值满足条件。	10 > ANY(my_array)
	ALL	数组所有值满足条件。	10 > ALL(my_array)
其他	**MD5**	获取MD5码。	MD5(my_column)
	ENCODE	编码字符串。	ENCODE(myNumber, 'BASE62')
	DECODE	解码字符串。	DECODE('000000008512af277ffffff8', 'HEX')

12.5.2.4 HBase JDBC API 接口介绍

Phoenix实现了大部分的java.sql接口，SQL语法紧跟ANSI SQL标准。

其支持处理函数可参见：

http://phoenix.apache.org/language/functions.html

其支持语法可参见：

http://phoenix.apache.org/language/index.html

12.5.2.5 HBase Web UI 接口介绍

操作场景

Web UI展示了HBase集群的状态，其中包括整个集群概况信息、RegionServer和Master的信息、快照、运行进程等信息。通过Web UI提供的信息可以对整个HBase集群的状况有一定的了解。

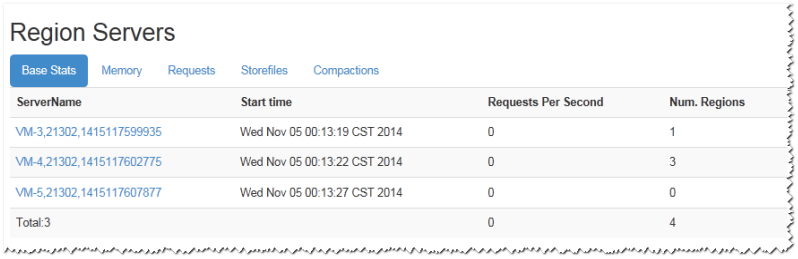
说明

请联系管理员获取具有访问Web UI权限的业务账号及其密码。

操作步骤

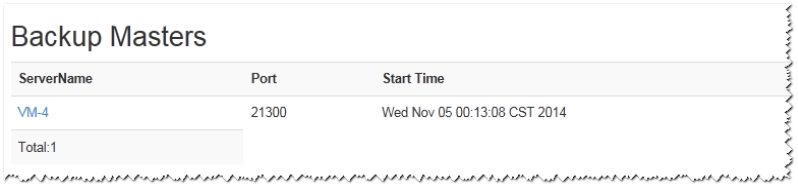
1. 登录FusionInsight Manager页面。选择“集群 > *待操作集群的名称* > 服务 > HBase > HMaster(主)”打开HBase的Web UI。
2. 在HBase的Web UI页面中，Home页面展示的是HBase的一些概况信息，具体包括以下信息：
   a. Region Servers页面展示了RegionServer的一些基本信息，如图12-28所示。

   图 12-28 Region Servers 基本信息

   b. Backup Master页面展示了Backup Master的信息，如图12-29所示。

   图 12-29 Backup Masters 基本信息

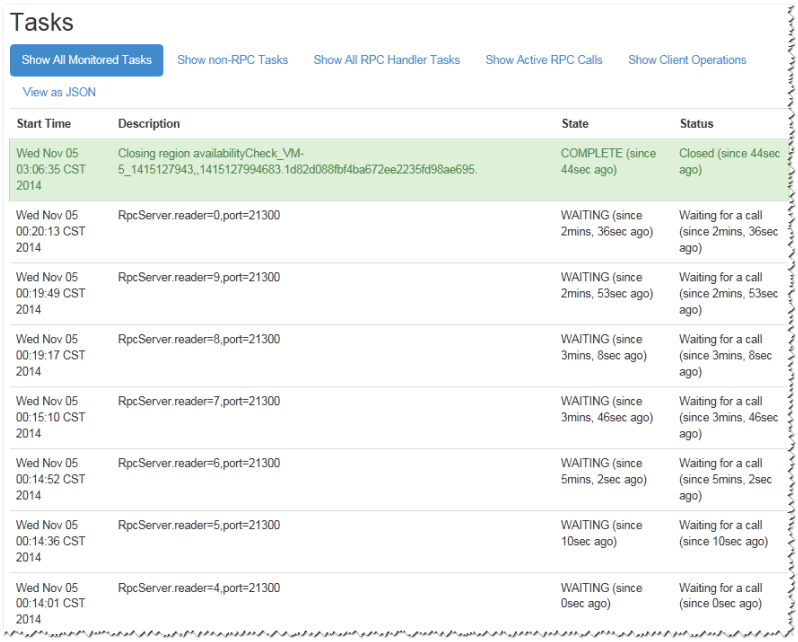
   c. Tables页面显示了HBase中表的信息，包括User Tables、Catalog Tables、Snapshots，如图12-30所示。

   图 12-30 Tables 基本信息

   d. Tasks页面显示了运行在HBase上的任务信息，包括开始时间，状态等信息，如图12-31所示。

图 12-31 Tasks 基本信息



Tasks

Show All Monitored Tasks | Show non-RPC Tasks | Show All RPC Handler Tasks | Show Active RPC Calls | Show Client Operations

View as JSON

Start Time	Description	State	Status
Wed Nov 05 03:06:35 CST 2014	Closing region availabilityCheck_VM-5_1415127943,,1415127994683.1d82d088fbf4ba672ee2235fd98ae695.	COMPLETE (since 44sec ago)	Closed (since 44sec ago)
Wed Nov 05 00:20:13 CST 2014	RpcServer.reader=0,port=21300	WAITING (since 2mins, 36sec ago)	Waiting for a call (since 2mins, 36sec ago)
Wed Nov 05 00:19:49 CST 2014	RpcServer.reader=9,port=21300	WAITING (since 2mins, 53sec ago)	Waiting for a call (since 2mins, 53sec ago)
Wed Nov 05 00:19:17 CST 2014	RpcServer.reader=8,port=21300	WAITING (since 3mins, 8sec ago)	Waiting for a call (since 3mins, 8sec ago)
Wed Nov 05 00:15:10 CST 2014	RpcServer.reader=7,port=21300	WAITING (since 3mins, 46sec ago)	Waiting for a call (since 3mins, 46sec ago)
Wed Nov 05 00:14:52 CST 2014	RpcServer.reader=6,port=21300	WAITING (since 5mins, 2sec ago)	Waiting for a call (since 5mins, 2sec ago)
Wed Nov 05 00:14:36 CST 2014	RpcServer.reader=5,port=21300	WAITING (since 10sec ago)	Waiting for a call (since 10sec ago)
Wed Nov 05 00:14:01 CST 2014	RpcServer.reader=4,port=21300	WAITING (since 0sec ago)	Waiting for a call (since 0sec ago)

3. 在HBase的Web UI页面中，Table Details页面展示的是HBase存储表的概要信息，如图12-32所示。

图 12-32 TableDetails

User Tables

1 table(s) in set.

Table	Description
t	't', {NAME => 'd', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}

4. 在HBase的Web UI页面中，Debug dump页面展示的是HBase的Debug信息，如图12-33所示。

图 12-33 Debug dump

5. 在HBase的Web UI页面中，HBaseConfiguration页面展示的是HBase的控制信息，如图12-34所示。

图 12-34 HBase Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
- <configuration>
  - <property>
      <name>dfs.journalnode.rpc-address</name>
      <value>0.0.0.0:8485</value>
      <source>hdfs-default.xml</source>
    </property>
  - <property>
      <name>io.storefile.bloom.block.size</name>
      <value>131072</value>
      <source>hbase-default.xml</source>
    </property>
  - <property>
      <name>hbase.sessioncontrol.maxSessions</name>
      <value>65535</value>
      <source>hbase-site.xml</source>
    </property>
  - <property>
      <name>yarn.ipc.rpc.class</name>
      <value>org.apache.hadoop.yarn.ipc.HadoopYarnProtoRPC</value>
      <source>yarn-default.xml</source>
    </property>
  - <property>
      <name>mapreduce.job.maxtaskfailures.per.tracker</name>
      <value>3</value>
      <source>mapred-default.xml</source>
    </property>
  - <property>
      <name>hbase.rest.threads.min</name>
      <value>2</value>
      <source>hbase-default.xml</source>
    </property>
  - <property>
      <name>hbase.rs.cacheblocksonwrite</name>
      <value>false</value>
      <source>hbase-default.xml</source>
    </property>
  - <property>
      <name>ha.health-monitor.connect-retry-interval.ms</name>
      <value>1000</value>
      <source>core-default.xml</source>
    </property>
  - <property>
      <name>yarn.resourcemanager.work-preserving-recovery.enabled</name>
      <value>false</value>
      <source>yarn-default.xml</source>
    </property>
  - <property>
      <name>dfs.client.mmap.cache.size</name>
      <value>256</value>
      <source>hdfs-default.xml</source>
    </property>
  - <property>
```

12.5.3 如何配置 HBase 双读能力

操作场景

HBase客户端应用通过自定义加载主备集群配置项，实现了双读能力。HBase双读作为提高HBase集群系统高可用性的一个关键特性，适用于四个查询场景：使用**Get**读取数据、使用批量**Get**读取数据、使用**Scan**读取数据，以及基于二级索引查询。它能够同时读取主备集群数据，减少查询毛刺，具体表现为：

- 高成功率：双并发读机制，保证每一次读请求的成功率。
- 可用性：单集群故障时，查询业务不中断。短暂的网络抖动也不会导致查询时间变长。
- 通用性：双读特性不支持双写，但不影响原有的实时写场景。
- 易用性：客户端封装处理，业务侧不感知。

说明

HBase双读使用约束：

- HBase双读特性基于Replication实现，备集群读取的数据可能和主集群存在差异，因此只能实现最终一致性。
- 目前HBase双读功能仅用于查询。主集群宕机时，最新数据无法同步，备集群可能查询不到最新数据。
- HBase的**Scan**操作可能分解为多次RPC。由于相关session信息在不同集群间不同步，数据不能保证完全一致，因此双读只在第一次RPC时生效，ResultScanner close之前的请求会固定访问第一次RPC时使用的集群。
- HBase Admin接口、实时写入接口只会访问主集群。所以主集群宕机后，不能提供Admin接口功能和实时写入接口功能，只能提供**Get**、**Scan**查询服务。

操作步骤

步骤1 参考**准备连接HBase集群配置文件**章节，获取HBase主集群客户端配置文件“core-site.xml”、“hbase-site.xml”、“hdfs-site.xml”，并将其放置到“src/main/resources/conf/active”目录下，该目录需要自己创建。

步骤2 参考**准备连接HBase集群配置文件**章节，获取备集群客户端配置文件“core-site.xml”、“hbase-site.xml”、“hdfs-site.xml”，并将其放置到“src/main/resources/conf/standby”目录下，该目录需要自己创建。

步骤3 创建“hbase-dual.xml”配置文件，放置到“src/main/resources/conf/”目录下。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<!--主集群配置文件目录-->
   <property>
      <name>hbase.dualclient.active.cluster.configuration.path</name>
      <value>{样例代码目录}\\src\\main\\resources\\conf\\active</value>
      </property>
<!--备集群配置文件目录-->
   <property>
      <name>hbase.dualclient.standby.cluster.configuration.path</name>
      <value>{样例代码目录}\\src\\main\\resources\\conf\\standby</value>
   </property>
<!--双读模式的Connection实现-->
   <property>
      <name>hbase.client.connection.impl</name>
      <value>org.apache.hadoop.hbase.client.HBaseMultiClusterConnectionImpl</value>
   </property>
<!--普通模式-->
   <property>
      <name>hbase.security.authentication</name>
      <value>Simple</value>
   </property>
<!--普通模式-->
   <property>
      <name>hadoop.security.authentication</name>
      <value>Simple</value>
   </property>
```

----结束

代码样例

- 创建双读Configuration，下面代码片段在“com.huawei.bigdata.hbase.examples”包的“TestMain”类的**init**方法中添加。

```
private static void init() throws IOException {
   // Default load from conf directory
```



```
    conf = HBaseConfiguration.create();
    //In Windows environment
    String userdir = TestMain.class.getClassLoader().getResource("conf").getPath() + File.separator;
    //In Linux environment
    //String userdir = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
    conf.addResource(new Path(userdir + "hbase-dual.xml"), false);
  }
```

- 确定数据来源的集群
  - GET请求，以下代码片段在“com.huawei.bigdata.hbase.examples”包的“HBaseSample”类的**testGet**方法中添加。

    ```
    Result result = table.get(get);
    if (result instanceof DualResult) {
        LOG.info(((DualResult)result).getClusterId());
    }
    ```

  - Scan请求，以下代码片段在“com.huawei.bigdata.hbase.examples”包的“HBaseSample”类的**testScanData**方法中添加。

    ```
    ResultScanner rScanner = table.getScanner(scan);
    if (rScanner instanceof HBaseMultiScanner) {
        LOG.info(((HBaseMultiScanner)rScanner).getClusterId());
    }
    ```

- 客户端支持打印metric信息

  “log4j.properties”文件中增加如下内容，客户端将metric信息输出到指定文件。

  ```
  log4j.logger.DUAL=debug,DUAL
  log4j.appender.DUAL=org.apache.log4j.RollingFileAppender
  log4j.appender.DUAL.File=/var/log/dual.log //客户端本地双读日志路径，根据实际路径修改，但目录要有写入权限
  log4j.additivity.DUAL=false
  log4j.appender.DUAL.MaxFileSize=${hbase.log.maxfilesize}
  log4j.appender.DUAL.MaxBackupIndex=${hbase.log.maxbackupindex}
  log4j.appender.DUAL.layout=org.apache.log4j.PatternLayout
  log4j.appender.DUAL.layout.ConversionPattern=%d{ISO8601} %-5p [%t] %c{2}: %m%n
  ```

HBase 双读操作相关配置项说明

表 12-13 hbase-dual.xml 配置项

配置项名称	配置项详解	默认值	级别
hbase.dualclient.active.cluster.configuration.path	主集群HBase客户端配置目录	无	必选配置
hbase.dualclient.standby.cluster.configuration.path	备集群HBase客户端配置目录	无	必选配置
dual.client.schedule.update.table.delay.second	更新开启容灾表列表的周期时间	5	可选配置
hbase.dualclient.glitchtimeout.ms	可以容忍主集群的最大毛刺时间	50	可选配置
hbase.dualclient.slow.query.timeout.ms	慢查询告警日志	180000	可选配置

配置项名称	配置项详解	默认值	级别
hbase.dualclient.active.cluster.id	主集群id	ACTIVE	可选配置
hbase.dualclient.standby.cluster.id	备集群id	STANDBY	可选配置
hbase.dualclient.active.executor.thread.max	请求主集群的线程池max大小	100	可选配置
hbase.dualclient.active.executor.thread.core	请求主集群的线程池core大小	100	可选配置
hbase.dualclient.active.executor.queue	请求主集群的线程池queue大小	256	可选配置
hbase.dualclient.standby.executor.thread.max	请求备集群的线程池max大小	100	可选配置
hbase.dualclient.standby.executor.thread.core	请求备集群的线程池core大小	100	可选配置
hbase.dualclient.standby.executor.queue	请求备集群的线程池queue大小	256	可选配置
hbase.dualclient.clear.executor.thread.max	清理资源线程池max大小	30	可选配置
hbase.dualclient.clear.executor.thread.core	清理资源线程池core大小	30	可选配置
hbase.dualclient.clear.executor.queue	清理资源线程池queue大小	Integer.MAX_VALUE	可选配置
dual.client.metrics.enable	客户端metric信息是否打印	true	可选配置
dual.client.schedule.metrics.second	客户端metric信息打印周期	300	可选配置
dual.client.asynchronous.enable	是否异步请求主备集群	false	可选配置

打印 metric 信息

表 12-14 基本指标项

Metric名称	描述	日志级别
total_request_count	周期时间内查询总次数	INFO
active_success_count	周期时间内主集群查询成功次数	INFO
active_error_count	周期时间内主集群查询失败次数	INFO
active_timeout_count	周期时间内主集群查询超时次数	INFO
standby_success_count	周期时间内备集群查询成功次数	INFO
standby_error_count	周期时间内备集群查询失败次数	INFO
Active Thread pool	周期打印请求主集群的执行线程池信息	DEBUG
Standby Thread pool	周期打印请求备集群的执行线程池信息	DEBUG
Clear Thread pool	周期打印释放资源的执行线程池信息	DEBUG

表 12-15 针对 GET、BatchGET、SCAN 请求，分别打印 Histogram 指标项

Metric名称	描述	日志级别
averageLatency(ms)	平均时延	INFO
minLatency(ms)	最小时延	INFO
maxLatency(ms)	最大时延	INFO
95thPercentileLatency(ms)	95%请求的最大时延	INFO
99thPercentileLatency(ms)	99%请求的最大时延	INFO
99.9PercentileLatency(ms)	99.9%请求的最大时延	INFO
99.99PercentileLatency(ms)	99.99%请求的最大时延	INFO

12.5.4 配置 Windows 通过 EIP 访问普通模式集群 HBase

操作场景

该章节通过指导用户配置集群绑定EIP，并配置HBase文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行hbase-example中的样例为例进行说明。

操作步骤

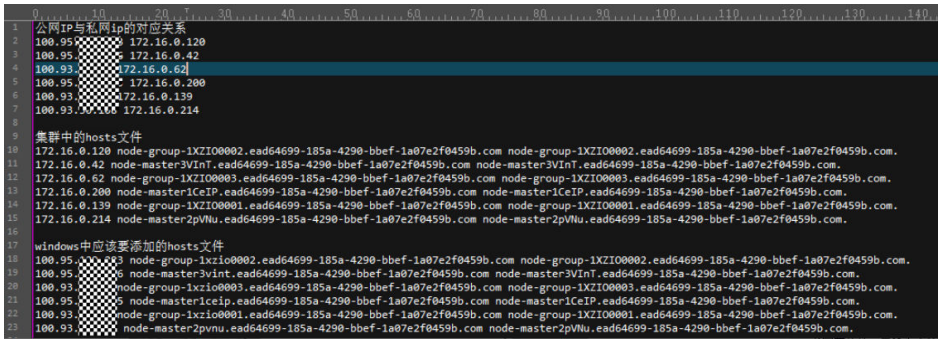
步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。

图 12-35 配置 host 文件



步骤2 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。
2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和21730TCP、21731TCP/UDP、21732TCP/UDP端口。

图 12-36 添加入方向规则

步骤3 在Manager界面选择“集群 > 服务 > HBase > 更多 > 下载客户端”，将客户端中的“core-site.xml”、“hdfs-site.xml”和“hbase-site.xml”复制到样例工程的resources目录下。

----结束

12.5.5 Phoenix 命令行操作介绍

Phoenix支持SQL的方式来操作HBase，以下简单介绍使用SQL语句建表/插入数据/查询数据/删表等操作。

前提条件

已安装HBase客户端，例如安装目录为“/opt/client”。以下操作的客户端目录只是举例，请根据实际安装目录修改。在使用客户端前，需要先下载并更新客户端配置文件，确认Manager的主管理节点后才能使用客户端。

操作步骤

步骤1 以客户端安装用户，登录安装HBase客户端的节点。

进入HBase客户端安装目录：

例如：**cd /opt/client**

步骤2 执行以下命令配置环境变量。

source bigdata_env

步骤3 如果当前集群已启用Kerberos认证，执行以下命令认证当前用户，当前用户需要具有创建HBase表的权限，具体请参见**创建角色**配置拥有对应权限的角色，参考**创建用户**为用户绑定对应角色。如果当前集群未启用Kerberos认证，则无需执行此命令。

kinit MRS 集群用户

例如，**kinit hbaseuser**。

步骤4 直接执行Phoenix客户端命令。

sqlline.py

步骤5 建表：

```
CREATE TABLE TEST (id VARCHAR PRIMARY KEY, name VARCHAR);
```

步骤6 插入数据：

```
UPSERT INTO TEST(id,name) VALUES ('1','jamee');
```

步骤7 查询数据：

```
SELECT * FROM TEST;
```

步骤8 删表：

```
DROP TABLE TEST;
```

步骤9 退出Phoenix命令行。

!quit

----结束

12.5.6 运行 HBase 应用开发程序产生 ServerRpcControllerFactory 异常如何处理

步骤1 检查应用开发工程的配置文件hbase-site.xml中是否包含配置项hbase.rpc.controllerfactory.class。

```
<name>hbase.rpc.controllerfactory.class</name>
<value>org.apache.hadoop.hbase.ipc.controller.ServerRpcControllerFactory</value>
```

步骤2 如果当前的应用开发工程配置项中包含该配置项，则应用开发程序还需要引入Jar包“phoenix-core-*.jar”。此Jar包可以从HBase客户端安装目录下的“HBase/hbase/lib”获取。

步骤3 如果不想引入该Jar包，请将应用开发工程的配置文件“hbase-site.xml”中的配置“hbase.rpc.controllerfactory.class”删除掉。

----结束

12.5.7 BulkLoad 和 Put 应用场景有哪些

问题

HBase支持使用bulkload和put方式加载数据，在大部分场景下bulkload提供了更快的数据加载速度，但bulkload并不是没有缺点的，在使用时需要关注bulkload和put适合在哪些场景使用。

回答

bulkload是通过启动MapReduce任务直接生成HFile文件，再将HFile文件注册到HBase，因此错误的使用bulkload会因为启动MapReduce任务而占用更多的集群内存和CPU资源，也可能会生成大量很小的HFile文件频繁的触发Compaction，导致查询速度急剧下降。

错误的使用put，会造成数据加载慢，当分配给RegionServer内存不足时会造成RegionServer内存溢出从而导致进程退出。

下面给出bulkload和put适合的场景：

- bulkload适合的场景：
  - 大量数据一次性加载到HBase。
  - 对数据加载到HBase可靠性要求不高，不需要生成WAL文件。
  - 使用put加载大量数据到HBase速度变慢，且查询速度变慢时。
  - 加载到HBase新生成的单个HFile文件大小接近HDFS block大小。
- put适合的场景：
  - 每次加载到单个Region的数据大小小于HDFS block大小的一半。
  - 数据需要实时加载。
  - 加载数据过程不会造成用户查询速度急剧下降。

12.5.8 install 编译构建 HBase Jar 包报错 Could not transfer artifact 如何处理

问题

样例代码在进行maven编译构建jar包时，Build Failed，提示错误信息：Could not transfer artifact org.apache.commons:commons-crypto:pom:${commons-crypto.version}

回答

hbase-common模块依赖commons-crypto，在hbase-common的pom.xml文件中，对于commons-crypto的引入，<version>使用了${commons-crypto.version}变量。该变量的解析逻辑为，os为aarch64时值为“**1.0.0-hw-aarch64**”，os为x86_64时值为“**1.0.0**”。编译环境因为一些配置原因导致maven未能通过os正确解析该变量时，可采用手动修改pom.xml方式进行规避正确编译。

在pom.xml中手动修改直接或间接依赖hbase-common模块的dependency，修改为如下形式，排除commons-crypto依赖。

```
<dependency>
   <groupId>org.apache.hbase</groupId>
   <artifactId>hbase-common</artifactId>
   <version>${hbase.version}</version>
   <exclusions>
      <exclusion>
         <groupId>org.apache.commons</groupId>
         <artifactId>commons-crypto</artifactId>
      </exclusion>
   </exclusions>
</dependency>
```

再手动添加指定版本的commons-crypto依赖。根据os架构为x86_64或aarch64填写正确version。

```
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-crypto</artifactId>
   <version>1.0.0</version>
</dependency>
```

13 HDFS 开发指南（安全模式）

13.1 HDFS 应用开发简介

HDFS 简介

HDFS（Hadoop Distribute FileSystem）是一个适合运行在通用硬件之上，具备高度容错特性，支持高吞吐量数据访问的分布式文件系统，非常适合大规模数据集应用。

HDFS适用于如下场景：

- 处理海量数据（TB或PB级别以上）
- 需要很高的吞吐量
- 需要高可靠性
- 需要很好的可扩展能力

HDFS 开发接口简介

HDFS支持使用Java语言进行程序开发，具体的API接口内容请参考**HDFS Java API接口介绍**。

常用概念

- Colocation

  同分布（Colocation）功能是将存在关联关系的数据或可能要进行关联操作的数据存储在相同的存储节点上。HDFS文件同分布的特性是，将那些需进行关联操作的文件存放在相同的数据节点上，在进行关联操作计算时，避免了到别的数据节点上获取数据的动作，大大降低了网络带宽的占用。

- Client

  HDFS Client主要包括五种方式：JAVA API、C API、Shell、HTTP REST API、WEB UI五种方式，可参考**常用API介绍**、**HDFS Shell命令介绍**。

  - JAVA API

    提供HDFS文件系统的应用接口，本开发指南主要介绍如何使用Java API进行HDFS文件系统的应用开发。

  - C API

提供HDFS文件系统的应用接口，使用C语言开发的用户可参考C接口的描述进行应用开发。

- Shell

  提供shell命令完成HDFS文件系统的基本操作。

- HTTP REST API

  提供除Shell、Java API和C API以外的其他接口，可通过此接口监控HDFS状态等信息。

- WEB UI

  提供Web可视化组件管理界面。

- keytab文件

  存放用户信息的密钥文件，应用程序采用此密钥文件在组件中进行API方式认证。

13.2 HDFS 应用开发流程介绍

开发流程中各阶段的说明如**图13-1**和**表13-1**所示。

图 13-1 HDFS 应用程序开发流程

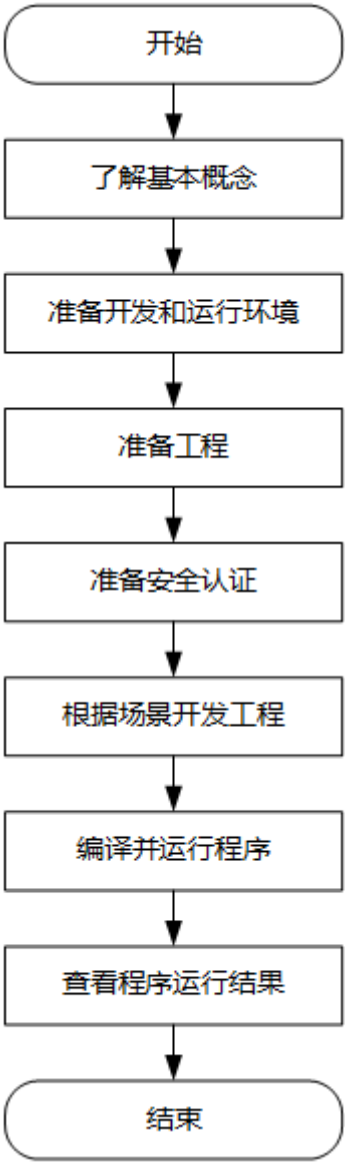


表 13-1 HDFS 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解HDFS的基本概念。	**HDFS应用开发简介**
准备开发和运行环境	使用IntelliJ IDEA工具，请根据指导完成开发环境配置。 HDFS的运行环境即HDFS客户端，请根据指导完成客户端的安装和配置。	**准备HDFS应用开发和运行环境**
准备工程	HDFS提供了不同场景下的样例程序，可以导入样例工程进行程序学习。	**导入并配置HDFS样例工程**

阶段	说明	参考文档
准备安全认证	如果使用的是安全集群，需要进行安全认证。	**配置HDFS应用安全认证**
根据场景开发工程	提供样例工程，帮助用户快速了解HDFS各部件的编程接口。	**开发HDFS应用**
编译并运行程序	指导用户将开发好的程序编译并提交运行。	**调测HDFS应用**
查看程序运行结果	程序运行结果会写在用户指定的路径下。用户还可以通过UI查看应用运行情况。	**调测HDFS应用**

13.3 HDFS 样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下HDFS相关样例工程：

表 13-2 HDFS 相关样例工程

样例工程位置	描述
hdfs-example-security	HDFS文件操作的Java示例程序。 本工程主要给出了创建HDFS文件夹、写文件、追加文件内容、读文件和删除文件/文件夹等相关接口操作示例。
hdfs-c-example	HDFS C语言开发代码样例。 本示例提供了基于C语言的HDFS文件系统连接、文件操作如创建文件、读写文件、追加文件、删除文件等。相关业务场景介绍请参见**HDFS C API接口介绍**。

13.4 准备 HDFS 应用开发环境

13.4.1 准备 HDFS 应用开发和运行环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如**表13-3**所示。

表 13-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境：Windows或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置，版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
准备开发用户	参考**准备MRS应用开发用户**进行操作，准备用于应用开发的集群用户并授予相应权限。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果本地Windows开发环境和集群业务平面网络互通，可下载集群客户端到本地，获取相关调测程序所需的集群配置文件及配置网络连通后，然后直接在Windows中进行程序调测。
  a. **登录FusionInsight Manager页面**，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“完整客户端”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig.tar”，继续解压该文件。解压到本地PC的“D:\FusionInsight_Cluster_1_Services_ClientConfig”目录下（路径中不能有空格）。

b. 进入客户端解压路径“FusionInsight_Cluster_1_Services_ClientConfig\HDFS\config”，手动将配置文件导入到HDFS样例工程的配置文件目录中（通常为“conf”文件夹）。

准备MRS应用开发用户时获取的keytab文件也放置于该目录下。

c. 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与解压目录下“hosts”文件中所列出的各主机在网络上互通。

说明

- 当客户端所在主机不是集群中的节点时，配置客户端网络连接，可避免执行客户端命令时出现错误。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 如果使用Linux环境调测程序，需准备安装集群客户端的Linux节点并获取相关配置文件。

a. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。

客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

b. **登录FusionInsight Manager页面**，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig/HDFS/config”路径下的所有配置文件至客户端节点，放置到工程代码的conf文件夹下。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp HDFS/config/* root@*客户端节点IP地址*:/opt/client/conf

准备MRS应用开发用户时获取的keytab文件也需放置于该目录下，主要配置文件说明如**表13-4**所示。

表 13-4 配置文件

文件名称	作用
core-site.xml	配置HDFS详细参数。
hdfs-site.xml	配置HDFS详细参数。
user.keytab	对于Kerberos安全认证提供用户信息。
krb5.conf	Kerberos Server配置信息。

说明

- 不同集群的“user.keytab”、“krb5.conf”不能共用。
- “conf”目录下的“log4j.properties”文件客户根据自己的需要进行配置。

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

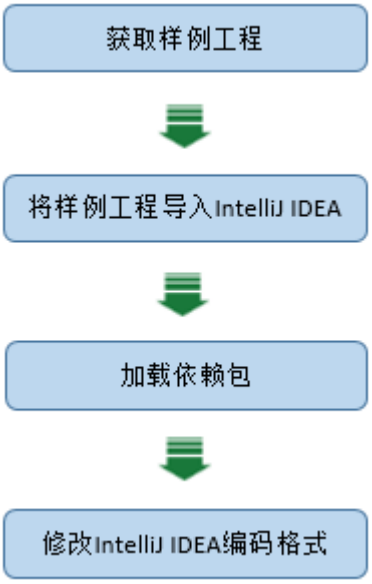
13.4.2 导入并配置 HDFS 样例工程

操作场景

HDFS针对多个场景提供样例工程，帮助客户快速学习HDFS工程。

以下操作步骤以导入HDFS样例代码为例。操作流程如图13-2所示。

图 13-2 导入样例工程流程



操作步骤

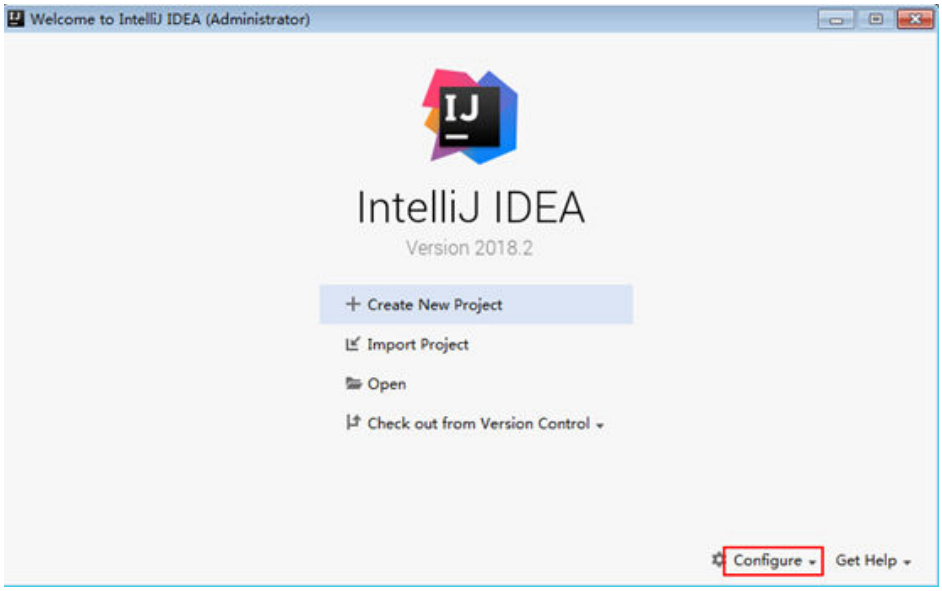
步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程“hdfs-example-security”。

步骤2 将**准备MRS应用开发用户**时得到的keytab文件“user.keytab”和“krb5.conf”文件放到样例工程的“conf”目录下。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

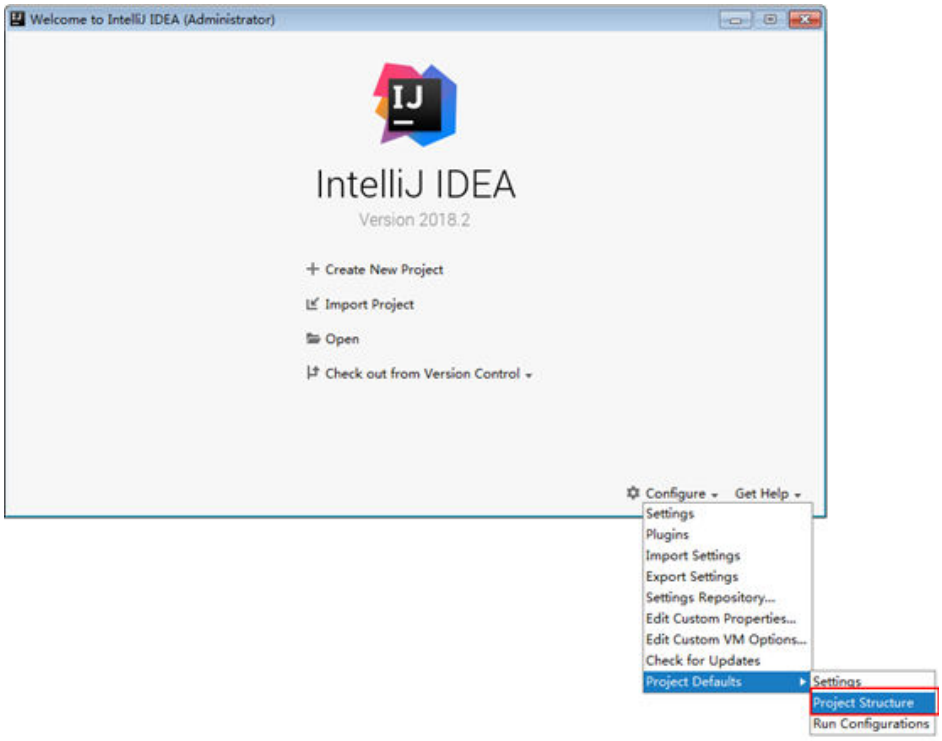
1. 打开IntelliJ IDEA，选择“Configure”。

图 13-3 Quick Start



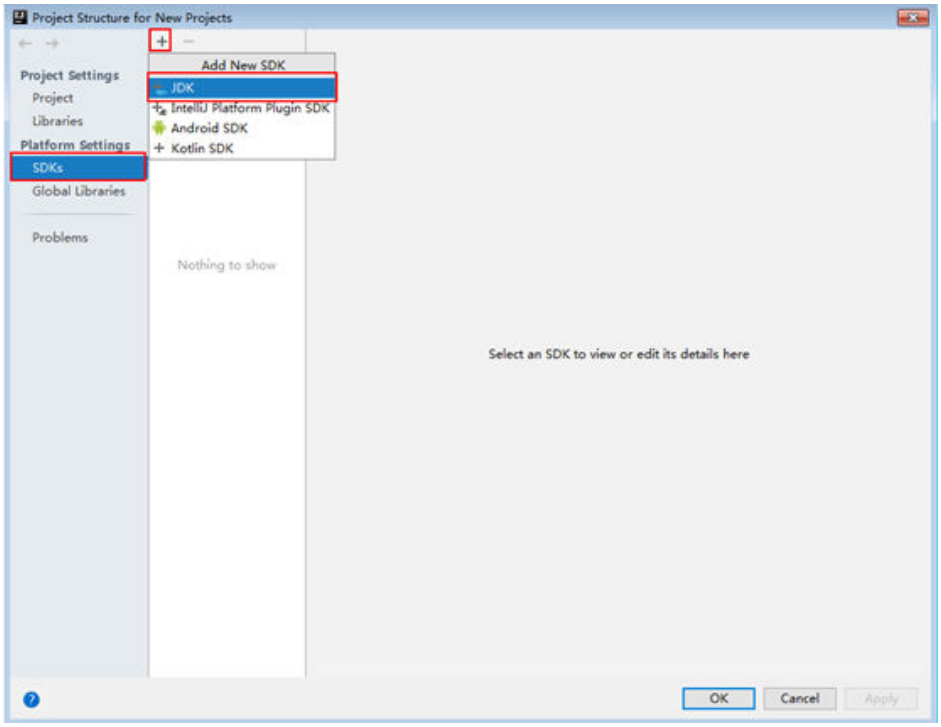
2. 在下拉框中选择“Project Defaults > Project Structure”。

图 13-4 Configure



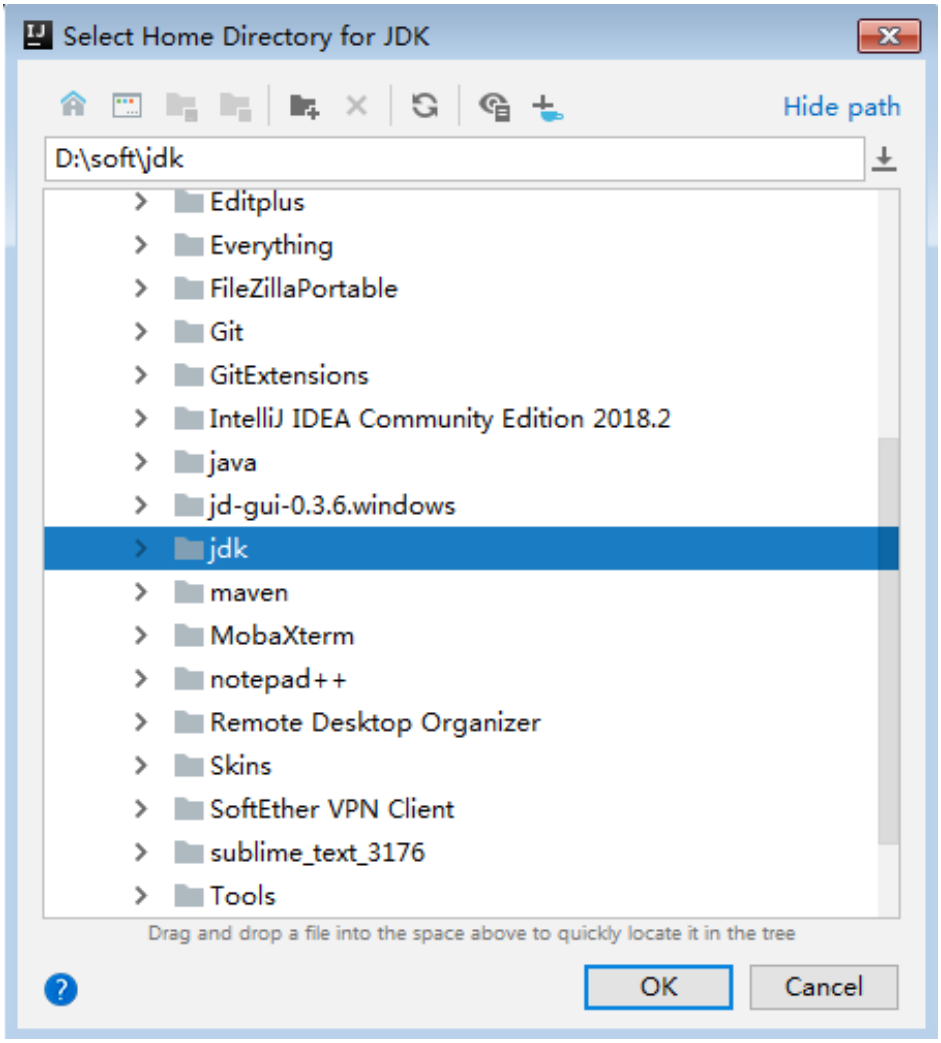
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 13-5 Project Structure for New Projects



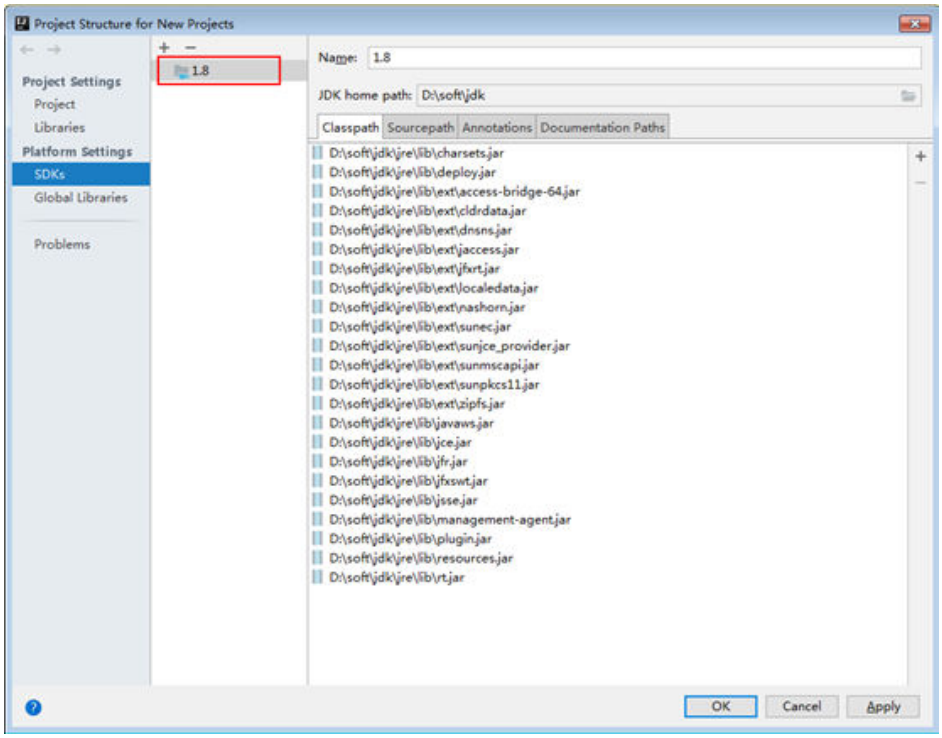
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 13-6 Select Home Directory for JDK



5. 完成JDK选择后，单击“OK”完成配置。

图 13-7 完成 JDK 配置



步骤4 导入样例工程到IntelliJ IDEA开发环境。

1. 打开IntelliJ IDEA，依次选择“File > Open”。
2. 在弹出的Open File or Project对话框中选择样例工程文件夹“hdfs-example-security”，单击“OK”。

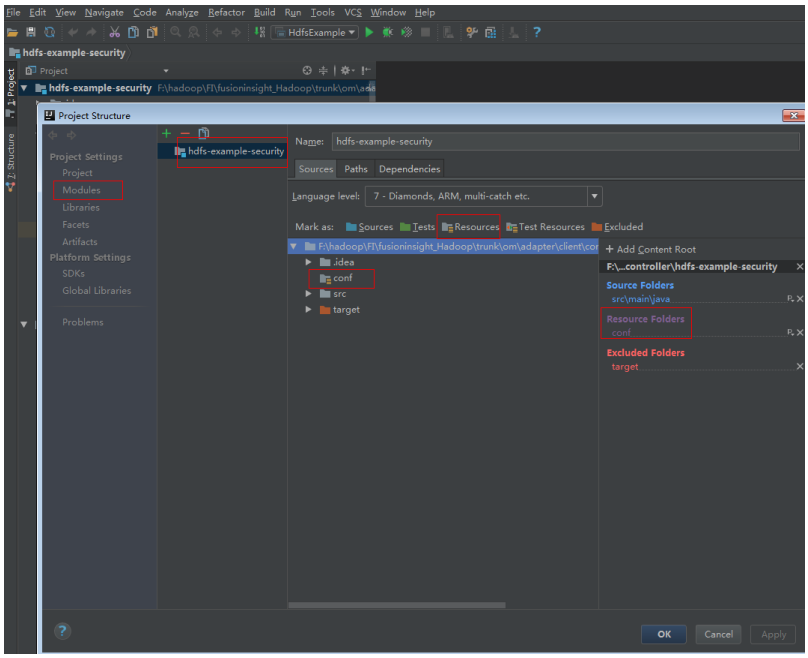
步骤5 将工程依赖的jar包添加到类路径。

如果通过开源镜像站方式获取的样例工程代码，在配置好Maven后（配置方式参考**配置华为开源镜像仓**）， 相关依赖jar包将自动下载，不需手动添加。

步骤6 将工程中的conf目录添加到资源路径。

在Intellij IDEA的菜单栏选择“File > Project Structure”。在弹出的会话框中，单击“Modules”，选中当前工程，并依次单击“Resources > conf > OK”，从而完成资源目录的设置，如“**图13-8**”所示。

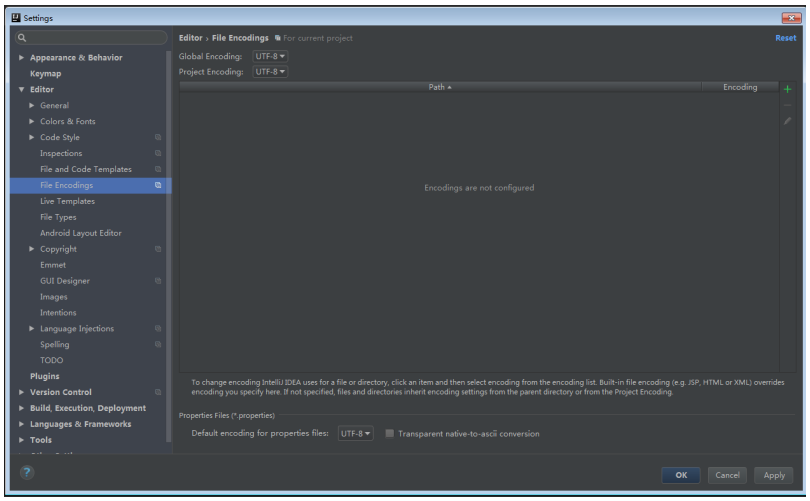
图 13-8 设置工程资源目录



步骤7 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”。
2. 在弹出的“Settings”窗口左边导航上选择“Editor > File Encodings”，在“Global Encoding”和“Project Encodings”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图13-9所示。

图 13-9 设置 IntelliJ IDEA 的编码格式



----结束

13.4.3 配置 HDFS 应用安全认证

场景说明

访问安全集群环境中的服务，需要先通过Kerberos安全认证。所以HDFS应用程序中需要写入安全认证代码，确保HDFS程序能够正常运行。

安全认证有两种方式：

- 命令行认证：
  提交HDFS应用程序运行前，在HDFS客户端执行如下命令进行认证。
  kinit *组件业务用户*

  说明

  该方式仅适用于Linux操作系统，且安装了HDFS的客户端。

- 代码认证：
  通过获取客户端的principal和keytab文件进行认证。
  注意修改代码中的***PRINCIPAL_NAME***变量为实际使用的值。

```
private static final String PRNCIPAL_NAME = "hdfsDeveloper";
```

安全认证代码

目前样例代码统一调用LoginUtil类进行安全认证。

在HDFS样例工程代码中，不同的样例工程，使用的认证代码不同，包括基本安全认证和带ZooKeeper认证。

- 基本安全认证：
  com.huawei.bigdata.hdfs.examples包的HdfsExample类样例程序不需要访问HBase或ZooKeeper，所以使用基本的安全认证代码即可。示例代码如下：

```
...
    private static final String PATH_TO_HDFS_SITE_XML =
HdfsExample.class.getClassLoader().getResource("hdfs-site.xml").getPath();
    private static final String PATH_TO_CORE_SITE_XML =
HdfsExample.class.getClassLoader().getResource("core-site.xml").getPath();
    private static final String PRNCIPAL_NAME = "hdfsDeveloper";
    private static final String PATH_TO_KEYTAB =
HdfsExample.class.getClassLoader().getResource("user.keytab").getPath();
    private static final String PATH_TO_KRB5_CONF =
HdfsExample.class.getClassLoader().getResource("krb5.conf").getPath();
    private static Configuration conf = null;
    //private static String PATH_TO_SMALL_SITE_XML =
HdfsExample.class.getClassLoader().getResource("smallfs-site.xml").getPath();
...
    private static void confLoad() throws IOException {
        System.setProperty("java.security.krb5.conf", PATH_TO_KRB5_CONF);
        conf = new Configuration();
        // conf file
        conf.addResource(new Path(PATH_TO_HDFS_SITE_XML));
        conf.addResource(new Path(PATH_TO_CORE_SITE_XML));
        // conf.addResource(new Path(PATH_TO_SMALL_SITE_XML));
    }
...
    private static void authentication() throws IOException {
        // security mode
        if ("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
            System.setProperty("java.security.krb5.conf", PATH_TO_KRB5_CONF);
            LoginUtil.login(PRNCIPAL_NAME, PATH_TO_KEYTAB, PATH_TO_KRB5_CONF, conf);
        }
    }
```

- 带ZooKeeper认证：
  com.huawei.bigdata.hdfs.examples包的“ColocationExample”类样例程序不仅需要基础安全认证，还需要添加ZooKeeper服务端Principal才能完成安全认证。示例代码如下：

```
...
    private static final String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
```

```
    private static final String PRINCIPAL = "username.client.kerberos.principal";
    private static final String KEYTAB = "username.client.keytab.file";
    private static final String PRNCIPAL_NAME = "hdfsDeveloper";
    private static final String LOGIN_CONTEXT_NAME = "Client";
    private static final String PATH_TO_KEYTAB = System.getProperty("user.dir") + File.separator +
"conf" + File.separator + "user.keytab";
    private static final String PATH_TO_KRB5_CONF =
ColocationExample.class.getClassLoader().getResource("krb5.conf") .getPath();
    private static String zookeeperDefaultServerPrincipal = null;
    private static Configuration conf = new Configuration();
    private static DFSColocationAdmin dfsAdmin;
    private static DFSColocationClient dfs;
    private static void init() throws IOException {
        LoginUtil.login(PRNCIPAL_NAME, PATH_TO_KEYTAB, PATH_TO_KRB5_CONF, conf);
        LoginUtil.setJaasConf(LOGIN_CONTEXT_NAME, PRNCIPAL_NAME, PATH_TO_KEYTAB);
        zookeeperDefaultServerPrincipal = "zookeeper/hadoop." +
KerberosUtil.getKrb5DomainRealm().toLowerCase();
        LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
zookeeperDefaultServerPrincipal);
    }
...
```

说明

- 以上安全认证代码中的hdfsDeveloper用户及该用户的user.keytab、krb5.conf为示例，实际操作时请联系管理员获取相应权限的账号以及对应该账号的keytab文件和krb5文件。
- 用户可登录FusionInsight Manager，单击“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。
- “zookeeper/hadoop.*<系统域名>*”为用户名，用户的用户名所包含的系统域名所有字母为小写。例如“本端域”参数为“9427068F-6EFA-4833-B43E-60CB641E5B6C.COM”，用户名为“zookeeper/hadoop.9427068f-6efa-4833-b43e-60cb641e5b6c.com”。

13.5 开发 HDFS 应用

13.5.1 HDFS 样例程序开发思路

场景说明

HDFS的业务操作对象是文件，代码样例中所涉及的文件操作主要包括创建文件夹、写文件、追加文件内容、读文件和删除文件/文件夹；HDFS还有其他的业务处理，例如设置文件权限等，其他操作可以在掌握本代码样例之后，再扩展学习。

本代码样例讲解顺序为：

1. HDFS初始化
2. 创建目录
3. 写文件
4. 追加文件内容
5. 读文件
6. 删除文件
7. 删除目录
8. 多线程
9. 设置存储策略
10. Colocation

开发思路

根据前述场景说明进行功能分解，以“/user/hdfs-examples/test.txt”文件的读写删除等操作为例，说明HDFS文件的基本操作流程，可分为以下八部分：

1. 通过安全认证。
2. 创建FileSystem对象：fSystem。
3. 调用fSystem的mkdir接口创建目录。
4. 调用fSystem的create接口创建FSDataOutputStream对象：out，使用out的write方法写入数据。
5. 调用fSystem的append接口创建FSDataOutputStream对象：out，使用out的write方法追加写入数据。
6. 调用fSystem的open接口创建FSDataInputStream对象：in，使用in的read方法读取文件。
7. 调用fSystem中的delete接口删除文件。
8. 调用fSystem中的delete接口删除文件夹。

13.5.2 初始化 HDFS

功能简介

在使用HDFS提供的API之前，需要先进行HDFS初始化操作。过程为：

1. 加载HDFS服务配置文件，并进行kerberos安全认证。
2. 认证通过后，实例化Filesystem。

说明

此处kerberos安全认证需要使用到的keytab文件，请提前准备。

配置文件介绍

登录HDFS时会使用到如表13-5所示的配置文件。这些文件均已导入到“hdfs-example-security”工程的“conf”目录。

表 13-5 配置文件

文件名称	作用
core-site.xml	配置HDFS详细参数。
hdfs-site.xml	配置HDFS详细参数。
user.keytab	对于Kerberos安全认证提供HDFS用户信息。
krb5.conf	Kerberos server配置信息。

说明

- 不同集群的“user.keytab”、“krb5.conf”不能共用。
- “conf”目录下的“log4j.properties”文件客户根据自己的需要进行配置。

代码样例

如下是代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples的HdfsExample类。

在Linux客户端运行应用和在Windows环境下运行应用的初始化代码相同，代码样例如下所示。

```
// 完成初始化和认证
confLoad();
authentication();
// 创建一个用例
HdfsExample hdfs_examples = new HdfsExample("/user/hdfs-examples", "test.txt");


/**
 *
 * 如果程序运行在Linux上，则需要core-site.xml、hdfs-site.xml的路径修改
 * 为在Linux下客户端文件的绝对路径
 *
 *
 */
private static void confLoad() throws IOException {
  conf = new Configuration();
  // conf file
  conf.addResource(new Path(PATH_TO_HDFS_SITE_XML));
  conf.addResource(new Path(PATH_TO_CORE_SITE_XML));
  // conf.addResource(new Path(PATH_TO_SMALL_SITE_XML));
}

/**
 *安全认证
 *
 */
private static void authentication() throws IOException {
  // security mode
  if ("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
    System.setProperty("java.security.krb5.conf", PATH_TO_KRB5_CONF);
    LoginUtil.login(PRNCIPAL_NAME, PATH_TO_KEYTAB, PATH_TO_KRB5_CONF, conf);
  }
}
/**
 *创建用例
 */
public HdfsExample(String path, String fileName) throws IOException  {
  this.DEST_PATH = path;
  this.FILE_NAME = fileName;
  instanceBuild();
}

private void instanceBuild() throws IOException {
  fSystem = FileSystem.get(conf);
}
```

在Windows环境和Linux环境下都需要运行login的代码样例，用于第一次登录使用，详细代码请参考com.huawei.hadoop.security中的LoginUtil类。

```
public synchronized static void login(String userPrincipal, String userKeytabPath, String krb5ConfPath,
Configuration conf)
        throws IOException
   {
        // 1.检查输入参数
        if ((userPrincipal == null) || (userPrincipal.length() <= 0))
```

```
    {
        LOG.error("input userPrincipal is invalid.");
        throw new IOException("input userPrincipal is invalid.");
    }

    if ((userKeytabPath == null) || (userKeytabPath.length() <= 0))
    {
        LOG.error("input userKeytabPath is invalid.");
        throw new IOException("input userKeytabPath is invalid.");
    }

    if ((krb5ConfPath == null) || (krb5ConfPath.length() <= 0))
    {
        LOG.error("input krb5ConfPath is invalid.");
        throw new IOException("input krb5ConfPath is invalid.");
    }

    if ((conf == null))
    {
        LOG.error("input conf is invalid.");
        throw new IOException("input conf is invalid.");
    }

    // 2.检查文件是否存在
    File userKeytabFile = new File(userKeytabPath);
    if (!userKeytabFile.exists())
    {
        LOG.error("userKeytabFile(" + userKeytabFile.getAbsolutePath() + ") does not exsit.");
        throw new IOException("userKeytabFile(" + userKeytabFile.getAbsolutePath() + ") does not exsit.");
    }
    if (!userKeytabFile.isFile())
    {
        LOG.error("userKeytabFile(" + userKeytabFile.getAbsolutePath() + ") is not a file.");
        throw new IOException("userKeytabFile(" + userKeytabFile.getAbsolutePath() + ") is not a file.");
    }

    File krb5ConfFile = new File(krb5ConfPath);
    if (!krb5ConfFile.exists())
    {
        LOG.error("krb5ConfFile(" + krb5ConfFile.getAbsolutePath() + ") does not exsit.");
        throw new IOException("krb5ConfFile(" + krb5ConfFile.getAbsolutePath() + ") does not exsit.");
    }
    if (!krb5ConfFile.isFile())
    {
        LOG.error("krb5ConfFile(" + krb5ConfFile.getAbsolutePath() + ") is not a file.");
        throw new IOException("krb5ConfFile(" + krb5ConfFile.getAbsolutePath() + ") is not a file.");
    }

    // 3.设置并检查krb5config
    setKrb5Config(krb5ConfFile.getAbsolutePath());
    setConfiguration(conf);

    // 4.登录hadoop并检查
    loginHadoop(userPrincipal, userKeytabFile.getAbsolutePath());
    LOG.info("Login success!!!!!!!!!!!!!!");
}
```

13.5.3 创建 HDFS 目录

功能简介

创建目录过程为：

1. 调用FileSystem实例的exists方法查看该目录是否存在。
2. 如果存在，则直接返回。
3. 如果不存在，则调用FileSystem实例的mkdirs方法创建该目录。

代码样例

如下是写文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 创建目录
 *
 * @throws java.io.IOException
 */
private void mkdir() throws IOException {
  Path destPath = new Path(DEST_PATH);
  if (!createPath(destPath)) {
      LOG.error("failed to create destPath " + DEST_PATH);
  return;
  }

  LOG.info("success to create path " + DEST_PATH);
}

/**
 * create file path
 *
 * @param filePath
 * @return
 * @throws java.io.IOException
 */
private boolean createPath(final Path filePath) throws IOException {
    if (!fSystem.exists(filePath)) {
        fSystem.mkdirs(filePath);
    }
    return true;
}
```

13.5.4 创建 HDFS 文件并写入内容

功能简介

写文件过程为：

1. 使用FileSystem实例的create方法获取写文件的输出流。
2. 使用该输出流将内容写入到HDFS的指定文件中。

说明

在写完文件后，需关闭所申请资源。

代码样例

如下是写文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 创建文件，写文件
 *
 * @throws java.io.IOException
 * @throws com.huawei.bigdata.hdfs.examples.ParameterException
 */
private void write() throws IOException {
  final String content = "hi, I am bigdata. It is successful if you can see me.";
  FSDataOutputStream out = null;
  try {
      out = fSystem.create(new Path(DEST_PATH + File.separator + FILE_NAME));
```

```
      out.write(content.getBytes());
      out.hsync();
      LOG.info("success to write.");
    } finally {
     // make sure the stream is closed finally.
     IOUtils.closeStream(out);
   }
}
```

13.5.5 追加信息到 HDFS 指定文件

功能简介

追加文件内容，是指在HDFS的某个指定文件后面，追加指定的内容。过程为：

1. 使用FileSystem实例的append方法获取追加写入的输出流。
2. 使用该输出流将待追加内容添加到HDFS的指定文件后面。

说明

在完成后，需关闭所申请资源。

代码样例

如下是代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 追加文件内容
 *
 * @throws java.io.IOException
 */
private void append() throws IOException {
    final String content = "I append this content.";
    FSDataOutputStream out = null;
    try {
        out = fSystem.append(new Path(DEST_PATH + File.separator + FILE_NAME));
        out.write(content.getBytes());
        out.hsync();
        LOG.info("success to append.");
    } finally {
        // make sure the stream is closed finally.
        IOUtils.closeStream(out);
    }
}
```

13.5.6 读取 HDFS 指定文件内容

功能简介

获取HDFS上某个指定文件的内容。过程为：

1. 使用FileSystem实例的open方法获取读取文件的输入流。
2. 使用该输入流读取HDFS的指定文件的内容。

说明

在完成后，需关闭所申请资源。

代码样例

如下是读文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 读文件
 *
 * @throws java.io.IOException
 */
private void read() throws IOException {
    String strPath = DEST_PATH + File.separator + FILE_NAME;
    Path path = new Path(strPath);
    FSDataInputStream in = null;
    BufferedReader reader = null;
    StringBuffer strBuffer = new StringBuffer();
    try {
        in = fSystem.open(path);
        reader = new BufferedReader(new InputStreamReader(in));
        String sTempOneLine;
        // write file
        while ((sTempOneLine = reader.readLine()) != null) {
            strBuffer.append(sTempOneLine);
        }
        LOG.info("result is : " + strBuffer.toString());
        LOG.info("success to read.");
    } finally {
        // make sure the streams are closed finally.
        IOUtils.closeStream(reader);
        IOUtils.closeStream(in);
    }
}
```

13.5.7 删除 HDFS 指定文件

功能简介

删除HDFS上某个指定文件。

说明

被删除的文件会被直接删除，且无法恢复。所以，执行删除操作需谨慎。

代码样例

如下是删除文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 删除文件
 *
 * @throws java.io.IOException
 */
private void delete() throws IOException {
    Path beDeletedPath = new Path(DEST_PATH + File.separator + FILE_NAME);
    if (fSystem.delete(beDeletedPath, true)) {
        LOG.info("success to delete the file " + DEST_PATH + File.separator + FILE_NAME);
    } else {
        LOG.warn("failed to delete the file " + DEST_PATH + File.separator + FILE_NAME);
    }
}
```

13.5.8 删除 HDFS 指定目录

功能简介

删除HDFS上某个指定目录。

说明

被删除的目录会被直接删除，且无法恢复。所以，执行删除操作需谨慎。

代码样例

如下是删除文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 删除目录
 *
 * @throws java.io.IOException
 */
private void rmdir() throws IOException {
    Path destPath = new Path(DEST_PATH);
    if (!deletePath(destPath)) {
        LOG.error("failed to delete destPath " + DEST_PATH);
        return;
    }
    LOG.info("success to delete path " + DEST_PATH);
}

/**
 * delete file path
 *
 * @param filePath
 * @return
 * @throws java.io.IOException
 */
private boolean deletePath(final Path filePath) throws IOException {
    if (!fSystem.exists(filePath)) {
        return false;
    }
    // fSystem.delete(filePath, true);
    return fSystem.delete(filePath, true);
}
```

13.5.9 创建 HDFS 多线程任务

功能简介

建立多线程任务，同时启动多个实例执行文件操作。

代码样例

如下是删除文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
 // 业务示例2：多线程
final int THREAD_COUNT = 2;
for (int threadNum = 0; threadNum < THREAD_COUNT; threadNum++) {
    HdfsExampleThread example_thread = new HdfsExampleThread("hdfs_example_" + threadNum);
    example_thread.start();
```

```
}


class HdfsExampleThread extends Thread {
    private final static Log LOG = LogFactory.getLog(HdfsExampleThread.class.getName());
    /**
     *
     * @param threadName
     */
    public HdfsExampleThread(String threadName) {
        super(threadName);
    }
    public void run() {
        HdfsExample example;
        try {
            example = new HdfsExample("/user/hdfs-examples/" + getName(), "test.txt");
            example.test();
        } catch (IOException e) {
            LOG.error(e);
        }
    }
}
```

example.test()方法即为对文件的操作，代码如下：

```
/**
 * HDFS 操作实例
 *
 * @throws IOException
 * @throws ParameterException
 *
 * @throws Exception
 *
 */
public void test() throws IOException {
    // 创建目录
    mkdir();

    // 写文件
    write();

    // 追加文件内容
    append();

    // 读文件
    read();

    // 删除文件
    delete();

    // 删除目录
    rmdir();
}
```

13.5.10 配置 HDFS 存储策略

功能简介

为HDFS上某个文件或文件夹指定存储策略。

代码样例

1. **登录FusionInsight Manager页面**，选择“集群 > *待操作集群的名称* > 服务 > HDFS > 配置 > 全部配置”。

2. 搜索并查看“dfs.storage.policy.enabled”的参数值是否为“true”，如果不是，修改为“true”，并单击“保存”，重启HDFS。
3. 查看代码。

   如下是代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
  /**
  * 设置存储策略
  * @param policyName
  * 策略名称能够被接受:
  * <li>HOT
  * <li>WARM
  * <li>COLD
  * <li>LAZY_PERSIST
  * <li>ALL_SSD
  * <li>ONE_SSD
  * @throws IOException
  */
 private void setStoragePolicy(String policyName) throws IOException {
    if (fSystem instanceof DistributedFileSystem) {
        DistributedFileSystem dfs = (DistributedFileSystem) fSystem;
        Path destPath = new Path(DEST_PATH);
        Boolean flag = false;
        mkdir();
        BlockStoragePolicySpi[] storage = dfs.getStoragePolicies();
        for (BlockStoragePolicySpi bs : storage) {
            if (bs.getName().equals(policyName)) {
                flag = true;
            }
            LOG.info("StoragePolicy:" + bs.getName());
        }
        if (!flag) {
            policyName = storage[0].getName();
        }
        dfs.setStoragePolicy(destPath, policyName);
        LOG.info("success to set Storage Policy path " + DEST_PATH);
        rmdir();
    } else {
        LOG.info("SmallFile not support to set Storage Policy !!!");
    }
 }
```

13.5.11 配置 HDFS 同分布策略（Colocation）

功能简介

同分布（Colocation）功能是将存在关联关系的数据或可能要进行关联操作的数据存储在相同的存储节点上。HDFS文件同分布的特性，将那些需进行关联操作的文件存放在相同数据节点上，在进行关联操作计算时避免了到别的数据节点上获取数据，大大降低网络带宽的占用。

在使用Colocation功能之前，建议用户对Colocation的内部机制有一定了解，包括：

Colocation分配节点原理

扩容与Colocation分配

Colocation与数据节点容量

- **Colocation分配节点原理**

  Colocation为locator分配数据节点的时候，locator的分配算法会根据已分配的情况，进行均衡的分配数据节点。

说明

locator分配算法的原理是，查询目前存在的所有locators，读取所有locators所分配的数据节点，并记录其使用次数。根据使用次数，对数据节点进行排序，使用次数少的排在前面，优先选择排在前面的节点。每次选择一个节点后，计数加1，并重新排序，选择后续的节点。

- **扩容与Colocation分配**

集群扩容之后，为了平衡地使用所有的数据节点，使新的数据节点的分配频率与旧的数据节点趋于一致，有如下两种策略可以选择，如表13-6所示。

表 13-6 分配策略

编号	策略	说明
1	删除旧的locators，为集群中所有数据节点重新创建locators。	1. 在未扩容之前分配的locators，平衡的使用了所有数据节点。当扩容后，新加入的数据节点并未分配到已经创建的locators中，所以使用Colocation来存储数据的时候，只会往旧的数据节点存储数据。 2. 由于locators与特定数据节点相关，所以当集群进行扩容的时候，就需要对Colocation的locators分配进行重新规划。
2	创建一批新的locators，并重新规划数据存放方式。	旧的locators使用的是旧的数据节点，而新创建的locators偏重使用新的数据节点，所以需要根据实际业务对数据的使用需求，重新规划locators的使用。

说明

一般的，建议用户在进行集群扩容之后采用策略一来重新分配locators，可以避免数据偏重使用新的数据节点。

- **Colocation与数据节点容量**

由于使用Colocation进行存储数据的时候，会固定存储在指定的locator所对应的数据节点上面，所以如果不对locator进行规划，会造成数据节点容量不均衡。下面总结了保证数据节点容量均衡的两个主要的使用原则，如表13-7所示。

表 13-7 使用原则

编号	使用原则	说明
1	所有的数据节点在locators中出现的频率一样。	如何保证频率一样：假如数据节点有N个，则创建locators的数量应为N的整数倍（N个、2N个……）。
2	对于所有locators的使用需要进行合理的数据存放规划，让数据均匀的分布在这些locators中。	无

HDFS的二次开发过程中，可以获取DFSColocationAdmin和DFSColocationClient实例，进行从location创建group、删除group、写文件和删除文件的操作。

说明

- 使用Colocation功能，用户指定了DataNode，会造成某些节点上数据量很大。数据倾斜严重，导致HDFS写任务失败。
- 由于数据倾斜，导致MapReduce只会在某几个节点访问，造成这些节点上负载很大，而其他节点闲置。
- 针对单个应用程序任务，只能使用一次DFSColocationAdmin和DFSColocationClient实例。如果每次对文件系统操作都获取此实例，会创建过多HDFS链接，消耗HDFS资源。
- Colocation提供了文件同分布的功能，执行集群Balancer或Mover操作时，会移动数据块，使Colocation功能失效。因此，使用Colocation功能时，建议将HDFS配置项dfs.datanode.block-pinning.enabled设置为true，此时执行集群Balancer或Mover操作时，使用Colocation写入的文件将不会被移动，从而保证了文件同分布。

代码样例

完整样例代码可参考com.huawei.bigdata.hdfs.examples.ColocationExample。

说明

- 在运行Colocation工程时，需要将HDFS用户绑定supergroup用户组。
- 在运行Colocation工程时，HDFS的配置项fs.defaultFS不能配置为viewfs://ClusterX。

1. 初始化

   使用Colocation前需要进行kerberos安全认证。

```
private static void init() throws IOException {

    conf.set(KEYTAB, PATH_TO_KEYTAB);
    conf.set(PRINCIPAL, PRNCIPAL_NAME);

    LoginUtil.setJaasConf(LOGIN_CONTEXT_NAME, PRNCIPAL_NAME, PATH_TO_KEYTAB);
    LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);
    LoginUtil.login(PRNCIPAL_NAME, PATH_TO_KEYTAB, PATH_TO_KRB5_CONF, conf);
}
```

2. 获取实例

   样例：Colocation的操作使用DFSColocationAdmin和DFSColocationClient实例，在进行创建group等操作前需获取实例。

```
    dfsAdmin = new DFSColocationAdmin(conf);
    dfs = new DFSColocationClient();
    dfs.initialize(URI.create(conf.get("fs.defaultFS")), conf);
```

3. 创建group

   样例：创建一个gid01组，组中包含3个locator。

```
/**
 * 创建group
 *
 * @throws java.io.IOException
 */
private static void createGroup() throws IOException {
    dfsAdmin.createColocationGroup(COLOCATION_GROUP_GROUP01,
            Arrays.asList(new String[] { "lid01", "lid02", "lid03" }));
}
```

4. 写文件，写文件前必须创建对应的group

   样例：写入testfile.txt文件。

```
/**
 * 创建并写入文件
 *
 * @throws java.io.IOException
 */
private static void put() throws IOException {
    FSDataOutputStream out = dfs.create(new Path(TESTFILE_TXT), true,
COLOCATION_GROUP_GROUP01, "lid01");
    // 待写入HDFS的数据
    byte[] readBuf = "Hello World".getBytes("UTF-8");
    out.write(readBuf, 0, readBuf.length);
    out.close();
}
```

5. 删除文件

   样例：删除testfile.txt文件。

```
/**
 * 删除文件
 *
 * @throws java.io.IOException
 */
@SuppressWarnings("deprecation")
private static void delete() throws IOException {
    dfs.delete(new Path(TESTFILE_TXT));
}
```

6. 删除group

   样例：删除gid01。

```
/**
 * 删除group
 *
 * @throws java.io.IOException
 */
private static void deleteGroup() throws IOException {
    dfsAdmin.deleteColocationGroup(COLOCATION_GROUP_GROUP01);
}
```

13.6 调测 HDFS 应用

13.6.1 在本地 Windows 环境中调测 HDFS 程序

操作场景

在代码完成开发后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

HDFS应用程序运行完成后，可直接通过运行结果查看应用程序运行情况，也可以通过HDFS日志获取应用运行情况。

在本地 Windows 环境中调测 HDFS 程序

步骤1 在开发环境中（例如IntelliJ IDEA中），分别选中以下两个工程运行程序：

- 选中HdfsExample.java，右键工程，选择“Run 'HdfsExample.main()'”运行应用工程。
- 选中ColocationExample.java，右键工程，选择“Run 'ColocationExample.main()'”运行应用工程。

说明

- 在HDFS任务运行过程中禁止重启HDFS服务，否则可能会导致任务失败。
- 在运行Colocation工程时，HDFS的配置项fs.defaultFS不能配置为viewfs://ClusterX。

----结束

设置系统的环境变量说明

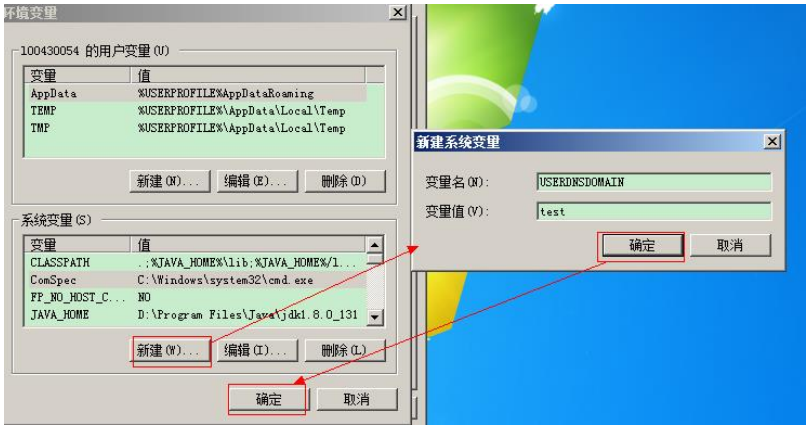
Hadoop在安全认证的时候，需要获取客户端所在主机的域名（Default Realm，从环境变量USERDNSDOMAIN中获取）。如果该主机没有域名，则运行样例程序会有如下报错：

```
Exception in thread "main" java.lang.IllegalArgumentException: Can't get Kerberos realm
        at org.apache.hadoop.security.HadoopKerberosName.setConfiguration(HadoopKerberosName.java:65)
        at org.apache.hadoop.security.UserGroupInformation.initialize(UserGroupInformation.java:288)
        at org.apache.hadoop.security.UserGroupInformation.ensureInitialized(UserGroupInformation.java:273)
        at org.apache.hadoop.security.UserGroupInformation.loginUserFromSubject(UserGroupInformation.java:832)
        at org.apache.hadoop.security.UserGroupInformation.getLoginUser(UserGroupInformation.java:802)
        at org.apache.hadoop.security.UserGroupInformation.getCurrentUser(UserGroupInformation.java:675)
        at org.apache.hadoop.conf.Configuration$Resource.getRestrictParserDefault(Configuration.java:243)
        at org.apache.hadoop.conf.Configuration$Resource.<init>(Configuration.java:211)
        at org.apache.hadoop.conf.Configuration$Resource.<init>(Configuration.java:203)
        at org.apache.hadoop.conf.Configuration.addResource(Configuration.java:851)
        at com.huawei.bigdata.hdfs.examples.HdfsExample.confLoad(HdfsExample.java:307)
        at com.huawei.bigdata.hdfs.examples.HdfsExample.main(HdfsExample.java:277)
Caused by: java.lang.reflect.InvocationTargetException
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at org.apache.hadoop.security.authentication.util.KerberosUtil.getDefaultRealm(KerberosUtil.java:88)
        at org.apache.hadoop.security.HadoopKerberosName.setConfiguration(HadoopKerberosName.java:63)
        ... 11 more
Caused by: KrbException: Cannot locate default realm
        at sun.security.krb5.Config.getDefaultRealm(Unknown Source)
        ... 17 more
```

此时需要用户设置系统的环境变量USERDNSDOMAIN以规避该问题，具体如下：

1. 单击“计算机”右键，选择“属性”，然后选择“高级系统设置 > 高级 > 环境变量”。
2. 设置系统环境变量，在系统变量下单击“新建”，弹出新建系统变量框，变量名中输入“USERDNSDOMAIN”，变量值设为非空字符串，图中以“test”为例。

   连续单击“确定”，完成系统环境变量的设置。



3. 关闭样例工程，重新打开，运行。

查看调测结果

- **查看运行结果获取应用运行情况**
  - HdfsExample Windows样例程序运行结果如下所示。

    ```
    ...
    1308 [main] INFO  org.apache.hadoop.security.UserGroupInformation  - Login successful for user hdfsDevelop using keytab file
    1308 [main] INFO  com.huawei.hadoop.security.LoginUtil  - Login success!!!!!!!!!!!!!
    ```

```
2040 [main] WARN  org.apache.hadoop.hdfs.shortcircuit.DomainSocketFactory  - The short-circuit local reads feature cannot be used because UNIX Domain sockets are not available on Windows.
3006 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to create path /user/hdfs-examples
3131 [main] WARN  org.apache.hadoop.util.NativeCodeLoader  - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
3598 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to write.
4408 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to append.
5015 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I am bigdata. It is successful if you can see me.I append this content.
5015 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to read.
5077 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete the file /user/hdfs-examples\test.txt
5186 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete path /user/hdfs-examples
5311 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to create path /user/hdfs-examples/hdfs_example_0
5311 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to create path /user/hdfs-examples/hdfs_example_1
5669 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to write.
5669 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to write.
7258 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to append.
7741 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to append.
7896 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I am bigdata. It is successful if you can see me.I append this content.
7896 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to read.
7959 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete the file /user/hdfs-examples/hdfs_example_1\test.txt
8068 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete path /user/hdfs-examples/hdfs_example_1
8364 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I am bigdata. It is successful if you can see me.I append this content.
8364 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to read.
8426 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete the file /user/hdfs-examples/hdfs_example_0\test.txt
8535 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete path /user/hdfs-examples/hdfs_example_0
...
```

说明

在Windows环境运行样例代码时可能会出现下面的异常，但是不影响业务：

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

- ColocationExample Windows样例程序运行结果如下所示。

```
...
945  [main] INFO  org.apache.hadoop.security.UserGroupInformation  - Login successful for user hdfsDeveloper using keytab file user.keytab
945  [main] INFO  com.huawei.hadoop.security.LoginUtil  - Login success!!!!!!!!!!!!!!
945  [main] INFO  com.huawei.hadoop.security.LoginUtil  - JaasConfiguration loginContextName=Client principal=hdfsDeveloper useTicketCache=false keytabFile=XXX\sample_project\src\hdfs-example-security\conf\user.keytab
946  [main] INFO  com.huawei.hadoop.security.KerberosUtil  - Get default realm successfully, the realm is : HADOOP.COM
...
Create Group has finished.
Put file is running...
Put file has finished.
Delete file is running...
Delete file has finished.
Delete Group is running...
```



```
Delete Group has finished.
4946 [main-EventThread] INFO  org.apache.zookeeper.ClientCnxn  - EventThread shut down for connection: 0x440751cb41a4d415
4946 [main] INFO  org.apache.zookeeper.ZooKeeper  - Connection: 0x440751cb41a4d415 closed
...
```

- **查看HDFS日志获取应用运行情况**

  您可以查看HDFS的NameNode日志了解应用运行情况，并根据日志信息调整应用程序。

13.6.2 在 Linux 环境中调测 HDFS 应用

操作场景

HDFS应用程序支持在Linux环境中运行。在程序代码完成开发后，可以上传Jar包至准备好的Linux环境中运行。

HDFS应用程序运行完成后，可直接通过运行结果查看应用程序运行情况，也可以通过HDFS日志获取应用运行情况。

前提条件

- 已安装客户端时：
  - 已安装HDFS客户端。
  - 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的hosts文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。
- 未安装客户端时：
  - Linux环境已安装JDK，版本号需要和IDEA导出Jar包使用的JDK版本一致。
  - 当Linux环境所在主机不是集群中的节点时，需要在Linux环境所在节点的hosts文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

已安装客户端时编译并运行程序

步骤1 进入样例工程本地根目录，在Windows命令提示符窗口中执行下面命令进行打包。

mvn -s "*{maven_setting_path}*" clean package

说明

- “{maven_setting_path}”为本地Maven的“settings.xml”文件路径，例如“C:\Users\Developer\settings.xml”。
- 打包成功之后，在工程根目录的“target”子目录下获取打好的jar包，例如“*HDFSTest-XXX.jar*”，jar包名称以实际打包结果为准。

步骤2 将导出的Jar包上传至集群客户端运行环境的任意目录下，例如“/opt/client”，然后在该目录下创建“conf”目录，将需要的配置文件复制至“conf”目录，具体操作请参考**准备运行环境**。

步骤3 配置环境变量：

cd /opt/client

source bigdata_env

步骤4 执行如下命令，运行Jar包。

hadoop jar *HDFSTest-XXX.jar* **com.huawei.bigdata.hdfs.examples.HdfsExample**

hadoop jar *HDFSTest-XXX.jar* **com.huawei.bigdata.hdfs.examples.ColocationExample**

说明

在运行**com.huawei.bigdata.hdfs.examples.ColocationExample**时，HDFS的配置项“fs.defaultFS”不能配置为“viewfs://ClusterX”。

----结束

未安装客户端时编译并运行程序

步骤1 进入工程本地根目录，在Windows命令提示符窗口中执行下面命令进行打包。

mvn -s "*{maven_setting_path}***" clean package**

说明

- 上述打包命令中的{maven_setting_path}为本地Maven的“settings.xml”文件路径。
- 打包成功之后，在工程根目录的target子目录下获取打好的jar包。

步骤2 将导出的Jar包上传至Linux运行环境的任意目录下，例如“/opt/client”。

步骤3 将工程中的“lib”文件夹和“conf”文件夹上传至和Jar包相同的Linux运行环境目录下（其中“lib”目录汇总包含了工程中依赖的所有的Jar包，“conf”目录包含运行jar包所需的集群相关配置文件，请参考**准备运行环境**）。

步骤4 执行如下命令运行Jar包。

java -cp *HDFSTest-XXX.jar***:conf/:lib/*** **com.huawei.bigdata.hdfs.examples.HdfsExample**

java -cp *HDFSTest-XXX.jar***:conf/:lib/*** **com.huawei.bigdata.hdfs.examples.ColocationExample**

说明

在运行“com.huawei.bigdata.hdfs.examples.ColocationExample”时，HDFS的配置项“fs.defaultFS”不能配置为“viewfs://ClusterX”。

----结束

查看调测结果

- **查看运行结果获取应用运行情况**
  - HdfsExample Linux样例程序运行结果如下所示。

```
0    [main] INFO  org.apache.hadoop.security.UserGroupInformation  - Login successful for user hdfsDevelop using keytab file user.keytab
1    [main] INFO  com.huawei.hadoop.security.LoginUtil  - Login success!!!!!!!!!!!!!
568  [main] WARN  org.apache.hadoop.util.NativeCodeLoader  - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
582  [main] WARN  org.apache.hadoop.hdfs.shortcircuit.DomainSocketFactory  - The short-circuit local reads feature cannot be used because libhadoop cannot be loaded.
793  [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to create path /user/hdfs-examples
969  [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to write.
1068 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to append.
1191 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I am bigdata. It is successful if you can see me.I append this content.
1191 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to read.
1202 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete the file /
```



```
user/hdfs-examples/test.txt
1210 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete path /user/hdfs-examples
1223 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to create path /user/hdfs-examples/hdfs_example_0
1224 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to create path /user/hdfs-examples/hdfs_example_1
1261 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to write.
1264 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to write.
2807 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to append.
2810 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to append.
2861 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I am bigdata. It is successful if you can see me.I append this content.
2861 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to read.
2866 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete the file /user/hdfs-examples/hdfs_example_0/test.txt
2874 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete path /user/hdfs-examples/hdfs_example_0
2874 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I am bigdata. It is successful if you can see me.I append this content.
2874 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to read.
2879 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete the file /user/hdfs-examples/hdfs_example_1/test.txt
2885 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete path /user/hdfs-examples/hdfs_example_1
```

- ColocationExample Linux样例程序运行结果如下所示。

```
0    [main] INFO  com.huawei.hadoop.security.LoginUtil  - JaasConfiguration loginContextName=Client principal=hdfsDevelop useTicketCache=false keytabFile=/opt/hdfsDemo/conf/user.keytab
817  [main] INFO  org.apache.hadoop.security.UserGroupInformation  - Login successful for user hdfsDevelop using keytab file user.keytab
817  [main] INFO  com.huawei.hadoop.security.LoginUtil  - Login success!!!!!!!!!!!!!!
1380 [main] WARN  org.apache.hadoop.util.NativeCodeLoader  - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
...
Create Group has finished.
Put file is running...
Put file has finished.
Delete file is running...
Delete file has finished.
Delete Group is running...
Delete Group has finished.
...
```

- **查看HDFS日志获取应用运行情况**

  您可以查看HDFS的namenode日志了解应用运行情况，并根据日志信息调整应用程序。

13.7 HDFS 应用开发常见问题

13.7.1 常用 API 介绍

13.7.1.1 HDFS Java API 接口介绍

HDFS完整和详细的接口可以直接参考官方网站描述：

http://hadoop.apache.org/docs/r3.1.1/api/index.html

HDFS 常用接口

HDFS常用的Java类有以下几个：

- FileSystem：是客户端应用的核心类。常用接口参见表13-8。
- FileStatus：记录文件和目录的状态信息。常用接口参见表13-9。
- DFSColocationAdmin：管理colocation组信息的接口。常用接口参见表13-10。
- DFSColocationClient：操作colocation文件的接口。常用接口参见表13-11。

说明

- 系统中不保留文件与LocatorId的映射关系，只保留节点与LocatorId的映射关系。当文件使用Colocation接口创建时，系统会将文件创建在LocatorId所对应的节点上。文件创建和写入要求使用Colocation相关接口。
- 文件写入完毕后，后续对该文件的相关操作不限制使用Colocation接口，也可以使用开源接口进行操作。
- DFSColocationClient类继承于开源的DistributedFileSystem类，包含其常用接口。建议使用DFSColocationClient进行Colocation相关文件操作。

表 13-8 类 FileSystem 常用接口说明

接口	说明
public static FileSystem get(Configuration conf)	Hadoop类库中最终面向用户提供的接口类是FileSystem，该类是个抽象类，只能通过该类的get方法得到具体类。get方法存在几个重载版本，常用的是这个。
public FSDataOutputStream create(Path f)	通过该接口可在HDFS上创建文件，其中f为文件的完整路径。
public void copyFromLocalFile(Path src, Path dst)	通过该接口可将本地文件上传到HDFS的指定位置上，其中src和dst均为文件的完整路径。
public boolean mkdirs(Path f)	通过该接口可在HDFS上创建文件夹，其中f为文件夹的完整路径。
public abstract boolean rename(Path src, Path dst)	通过该接口可为指定的HDFS文件重命名，其中src和dst均为文件的完整路径。
public abstract boolean delete(Path f, boolean recursive)	通过该接口可删除指定的HDFS文件，其中f为需要删除文件的完整路径，recuresive用来确定是否进行递归删除。
public boolean exists(Path f)	通过该接口可查看指定HDFS文件是否存在，其中f为文件的完整路径。
public FileStatus getFileStatus(Path f)	通过该接口可以获取文件或目录的FileStatus对象，该对象记录着该文件或目录的各种状态信息，其中包括修改时间、文件目录等等。

接口	说明
public BlockLocation[] getFileBlockLocations(FileStatus file, long start, long len)	通过该接口可查找指定文件在HDFS集群上块的位置，其中file为文件的完整路径，start和len来标识查找文件的块的范围。
public FSDataInputStream open(Path f)	通过该接口可以打开HDFS上指定文件的输出流，并可通过FSDataInputStream类提供接口进行文件的读出，其中f为文件的完整路径。
public FSDataOutputStream create(Path f, boolean overwrite)	通过该接口可以在HDFS上创建指定文件的输入流，并可通过FSDataOutputStream类提供的接口进行文件的写入，其中f为文件的完整路径，overwrite为true时表示如果文件已经存在，则重写文件；如果为false，当文件已经存在时，则抛出异常。
public FSDataOutputStream append(Path f)	通过该接口可以打开HDFS上已经存在的指定文件的输入流，并可通过FSDataOutputStream类提供的接口进行文件的写入，其中f为文件的完整路径。

表 13-9 类 FileStatus 常用接口说明

接口	说明
public long getModificationTime()	通过该接口可查看指定HDFS文件的修改时间。
public Path getPath()	通过该接口可查看指定HDFS中某个目录下所有文件。

表 13-10 类 DFSColocationAdmin 常用接口说明

接口	说明
public Map<String, List<DatanodeInfo>> createColocationGroup(String groupId,String file)	根据文件file中的locatorIds信息，创建group。file为文件路径。
public Map<String, List<DatanodeInfo>> createColocationGroup(String groupId,List<String> locators)	使用内存中List的locatorIds信息，创建group。
public void deleteColocationGroup(String groupId)	删除group。

接口	说明
public List<String> listColocationGroups()	返回colocation所有组信息，返回的组Id数组按创建时间排序。
public List<DatanodeInfo> getNodesForLocator(String groupId, String locatorId)	获取该locator中所有节点列表。

表 13-11 类 DFSColocationClient 常用接口说明

接口	说明
public FSDataOutputStream create(Path f, boolean overwrite, String groupId,String locatorId)	用colocation模式，创建一个FSDataOutputStream，从而允许用户在f路径写文件。 f为HDFS路径。 overwrite表示如果文件已存在是否允许覆盖。 用户指定文件所属的groupId和locatorId必须已经存在。
public FSDataOutputStream create(final Path f, final FsPermission permission, final EnumSet<CreateFlag> cflags, final int bufferSize, final short replication, final long blockSize, final Progressable progress, final ChecksumOpt checksumOpt, final String groupId, final String locatorId)	功能与FSDataOutputStream create(Path f, boolean overwrite, String groupId,String locatorId)相同，只是允许用户自定义checksum选项。
public void close()	使用完毕后关闭连接。

表 13-12 HDFS 客户端 WebHdfsFileSystem 接口说明

接口	说明
public RemoteIterator<FileStatus> listStatusIterator(final Path)	该API有助于通过使用远程迭代的多个请求获取子文件和文件夹信息，从而避免在获取大量子文件和文件夹信息时，用户界面变慢。

基于 API 的 Glob 路径模式以获取 LocatedFileStatus 和从 FileStatus 打开文件

在DistributedFileSystem中添加了以下API，以获取具有块位置的FileStatus，并从FileStatus对象打开文件。这些API将减少从客户端到Namenode的RPC调用的数量。

表 13-13 FileSystem API 接口说明

Interface接口	Description说明
public LocatedFileStatus[] globLocatedStatus(Path, PathFilter, boolean) throws IOException	返回一个LocatedFileStatus对象数组，其对应文件路径符合路径过滤规则。
public FSDataInputStream open(FileStatus stat) throws IOException	如果stat对象是LocatedFileStatusHdfs的实例，该实例已具有位置信息，则直接创建InputStream而不联系Namenode。

13.7.1.2 HDFS C API 接口介绍

功能简介

C语言应用开发代码样例中所涉及的文件操作主要包括创建文件、读写文件、追加文件、删除文件。完整和详细的接口请直接参考官网上的描述以了解其使用方法：

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/LibHdfs.html。

代码样例

下面代码片段仅为演示，具体代码请参见获取样例代码解压目录中“hdfs-c-example/hdfs_test.c”文件。

1. 设置HDFS NameNode参数，建立HDFS文件系统连接。

```
hdfsFS fs = hdfsConnect("default", 0);
fprintf(stderr, "hdfsConnect- SUCCESS!\n");
```

2. 创建HDFS目录。

```
const char* dir = "/tmp/nativeTest";
int exitCode = hdfsCreateDirectory(fs, dir);
if( exitCode == -1 ){
    fprintf(stderr, "Failed to create directory %s \n", dir );
    exit(-1);
}
fprintf(stderr, "hdfsCreateDirectory- SUCCESS! : %s\n", dir);
```

3. 写文件。

```
const char* file = "/tmp/nativeTest/testfile.txt";
hdfsFile writeFile = openFile(fs, (char*)file, O_WRONLY |O_CREAT, 0, 0, 0);
fprintf(stderr, "hdfsOpenFile- SUCCESS! for write : %s\n", file);

if(!hdfsFileIsOpenForWrite(writeFile)){
   fprintf(stderr, "Failed to open %s for writing.\n", file);
   exit(-1);
}

char* buffer = "Hadoop HDFS Native file write!";

hdfsWrite(fs, writeFile, (void*)buffer, strlen(buffer)+1);
```

```
fprintf(stderr, "hdfsWrite- SUCCESS! : %s\n", file);

printf("Flushing file data ....\n");
if (hdfsFlush(fs, writeFile)) {
    fprintf(stderr, "Failed to 'flush' %s\n", file);
    exit(-1);
}
hdfsCloseFile(fs, writeFile);
fprintf(stderr, "hdfsCloseFile- SUCCESS! : %s\n", file);
```

4. 读文件。

```
hdfsFile readFile = openFile(fs, (char*)file, O_RDONLY, 100, 0, 0);
fprintf(stderr, "hdfsOpenFile- SUCCESS! for read : %s\n", file);

if(!hdfsFileIsOpenForRead(readFile)){
   fprintf(stderr, "Failed to open %s for reading.\n", file);
   exit(-1);
}

buffer = (char *) malloc(100);
tSize num_read = hdfsRead(fs, readFile, (void*)buffer, 100);
fprintf(stderr, "hdfsRead- SUCCESS!, Byte read : %d, File contant : %s \n", num_read ,buffer);
hdfsCloseFile(fs, readFile);
```

5. 指定位置开始读文件。

```
buffer = (char *) malloc(100);
readFile = openFile(fs, file, O_RDONLY, 100, 0, 0);
if (hdfsSeek(fs, readFile, 10)) {
    fprintf(stderr, "Failed to 'seek' %s\n", file);
    exit(-1);
}
num_read = hdfsRead(fs, readFile, (void*)buffer, 100);
fprintf(stderr, "hdfsSeek- SUCCESS!, Byte read : %d, File seek contant : %s \n", num_read ,buffer);
hdfsCloseFile(fs, readFile);
```

6. 复制文件。

```
const char* destfile = "/tmp/nativeTest/testfile1.txt";
if (hdfsCopy(fs, file, fs, destfile)) {
   fprintf(stderr, "File copy failed, src : %s, des : %s \n", file, destfile);
   exit(-1);
}
fprintf(stderr, "hdfsCopy- SUCCESS!, File copied, src : %s, des : %s \n", file, destfile);
```

7. 移动文件。

```
const char* mvfile = "/tmp/nativeTest/testfile2.txt";
if (hdfsMove(fs, destfile, fs, mvfile )) {
   fprintf(stderr, "File move failed, src : %s, des : %s \n", destfile , mvfile);
   exit(-1);
}
fprintf(stderr, "hdfsMove- SUCCESS!, File moved, src : %s, des : %s \n", destfile , mvfile);
```

8. 重命名文件。

```
const char* renamefile = "/tmp/nativeTest/testfile3.txt";
if (hdfsRename(fs, mvfile, renamefile)) {
    fprintf(stderr, "File rename failed, Old name : %s, New name : %s \n", mvfile, renamefile);
    exit(-1);
}
fprintf(stderr, "hdfsRename- SUCCESS!, File renamed, Old name : %s, New name : %s \n", mvfile,
renamefile);
```

9. 删除文件。

```
if (hdfsDelete(fs, renamefile, 0)) {
   fprintf(stderr, "File delete failed : %s \n", renamefile);
   exit(-1);
}
fprintf(stderr, "hdfsDelete- SUCCESS!, File deleted : %s\n",renamefile);
```

10. 设置副本数。

```
if (hdfsSetReplication(fs, file, 10)) {
   fprintf(stderr, "Failed to set replication : %s \n", file );
   exit(-1);
```



```
}
fprintf(stderr, "hdfsSetReplication- SUCCESS!, Set replication 10 for %s\n",file);
```

11. 设置用户、用户组。

```
if (hdfsChown(fs, file, "root", "root")) {
   fprintf(stderr, "Failed to set chown : %s \n", file );
   exit(-1);
}
fprintf(stderr, "hdfsChown- SUCCESS!, Chown success for %s\n",file);
```

12. 设置权限。

```
if (hdfsChmod(fs, file, S_IRWXU | S_IRWXG | S_IRWXO)) {
   fprintf(stderr, "Failed to set chmod: %s \n", file );
   exit(-1);
}
fprintf(stderr, "hdfsChmod- SUCCESS!, Chmod success for %s\n",file);
```

13. 设置文件时间。

```
struct timeval now;
gettimeofday(&now, NULL);
if (hdfsUtime(fs, file, now.tv_sec, now.tv_sec)) {
   fprintf(stderr, "Failed to set time: %s \n", file );
   exit(-1);
}
fprintf(stderr, "hdfsUtime- SUCCESS!, Set time success for %s\n",file);
```

14. 获取文件信息。

```
hdfsFileInfo *fileInfo = NULL;
if((fileInfo = hdfsGetPathInfo(fs, file)) != NULL) {
   printFileInfo(fileInfo);
   hdfsFreeFileInfo(fileInfo, 1);
   fprintf(stderr, "hdfsGetPathInfo - SUCCESS!\n");
}
```

15. 遍历目录。

```
hdfsFileInfo *fileList = 0;
int numEntries = 0;
if((fileList = hdfsListDirectory(fs, dir, &numEntries)) != NULL) {
   int i = 0;
   for(i=0; i < numEntries; ++i) {
      printFileInfo(fileList+i);
   }
   hdfsFreeFileInfo(fileList, numEntries);
}
fprintf(stderr, "hdfsListDirectory- SUCCESS!, %s\n", dir);
```

16. stream builder接口。

```
buffer = (char *) malloc(100);
struct hdfsStreamBuilder *builder= hdfsStreamBuilderAlloc(fs, (char*)file, O_RDONLY);
hdfsStreamBuilderSetBufferSize(builder,100);
hdfsStreamBuilderSetReplication(builder,20);
hdfsStreamBuilderSetDefaultBlockSize(builder,10485760);
readFile = hdfsStreamBuilderBuild(builder);
num_read = hdfsRead(fs, readFile, (void*)buffer, 100);
fprintf(stderr, "hdfsStreamBuilderBuild- SUCCESS! File read success. Byte read : %d, File contant : %s
\n", num_read ,buffer);
struct hdfsReadStatistics *stats = NULL;
hdfsFileGetReadStatistics(readFile, &stats);
fprintf(stderr, "hdfsFileGetReadStatistics- SUCCESS! totalBytesRead : %" PRId64 ",
totalLocalBytesRead : %" PRId64 ", totalShortCircuitBytesRead : %" PRId64 ",
totalZeroCopyBytesRead : %" PRId64 "\n", stats->totalBytesRead , stats->totalLocalBytesRead, stats-
>totalShortCircuitBytesRead,  stats->totalZeroCopyBytesRead);
hdfsFileFreeReadStatistics(stats);
free(buffer);
```

17. 断开HDFS文件系统连接。

```
hdfsDisconnect(fs);
```

准备运行环境

在节点上安装客户端，例如安装到“/opt/client”目录。

Linux 中编译并运行程序

1. 进入Linux客户端目录，运行如下命令导入公共环境变量：

   cd/opt/client

   source*bigdata_env*

2. 在该目录下用hdfs用户进行命令行认证，用户密码请咨询集群管理员。

   kinit*hdfs*

   说明

   kinit一次票据时效24小时。24小时后再次运行样例，需要重新**kinit**命令。

3. 进入“/opt/client/HDFS/hadoop/hdfs-c-example”目录下，运行如下命令导入客户端环境变量。

   cd/opt/client/HDFS/hadoop/hdfs-c-example

   source*component_env_C_example*

4. 清除之前运行生成的目标文件和可执行文件，运行如下命令。

   make clean

   执行结果如下：

   ```
   [root@10-120-85-2 hdfs-c-example]# make clean
   rm -f hdfs_test.o
   rm -f hdfs_test
   ```

5. 编译生成新的目标和可执行文件，运行如下命令。

   make（或***make all***）

   执行结果如下：

   ```
   [root@10-120-85-2 hdfs-c-example]# make all
   cc -c -I/opt/client/HDFS/hadoop/include -Wall -o hdfs_test.o hdfs_test.c
   cc -o hdfs_test hdfs_test.o -lhdfs
   ```

6. 运行文件以实现创建文件、读写追加文件和删除文件的功能，运行如下命令。

   make run

   执行结果如下：

   ```
   [root@10-120-85-2 hdfs-c-example]# make run
   ./hdfs_test
   hdfsConnect- SUCCESS!
   hdfsCreateDirectory- SUCCESS! : /tmp/nativeTest
   hdfsOpenFile- SUCCESS! for write : /tmp/nativeTest/testfile.txt
   hdfsWrite- SUCCESS! : /tmp/nativeTest/testfile.txt
   Flushing file data ....
   hdfsCloseFile- SUCCESS! : /tmp/nativeTest/testfile.txt
   hdfsOpenFile- SUCCESS! for read : /tmp/nativeTest/testfile.txt
   hdfsRead- SUCCESS!, Byte read : 31, File contant : Hadoop HDFS Native file write!
   hdfsSeek- SUCCESS!, Byte read : 21, File seek contant : S Native file write!
   hdfsPread- SUCCESS!, Byte read : 10, File pead contant : S Native f
   hdfsCopy- SUCCESS!, File copied, src : /tmp/nativeTest/testfile.txt, des : /tmp/nativeTest/testfile1.txt
   hdfsMove- SUCCESS!, File moved, src : /tmp/nativeTest/testfile1.txt, des : /tmp/nativeTest/testfile2.txt
   hdfsRename- SUCCESS!, File renamed, Old name : /tmp/nativeTest/testfile2.txt, New name : /tmp/
   nativeTest/testfile3.txt
   hdfsDelete- SUCCESS!, File deleted : /tmp/nativeTest/testfile3.txt
   hdfsSetReplication- SUCCESS!, Set replication 10 for /tmp/nativeTest/testfile.txt
   hdfsChown- SUCCESS!, Chown success for /tmp/nativeTest/testfile.txt
   hdfsChmod- SUCCESS!, Chmod success for /tmp/nativeTest/testfile.txt
   hdfsUtime- SUCCESS!, Set time success for /tmp/nativeTest/testfile.txt

   Name: hdfs://hacluster/tmp/nativeTest/testfile.txt, Type: F, Replication: 10, BlockSize: 134217728, Size:
   31, LastMod: 1500345260, Owner: root, Group: root, Permissions: 511 (rwxrwxrwx)
   hdfsGetPathInfo - SUCCESS!
   ```

```
Name: hdfs://hacluster/tmp/nativeTest/testfile.txt, Type: F, Replication: 10, BlockSize: 134217728, Size:
31, LastMod: 1500345260, Owner: root, Group: root, Permissions: 511 (rwxrwxrwx)
hdfsListDirectory- SUCCESS!, /tmp/nativeTest
hdfsTruncateFile- SUCCESS!, /tmp/nativeTest/testfile.txt
Block Size : 134217728
hdfsGetDefaultBlockSize- SUCCESS!
Block Size : 134217728 for file /tmp/nativeTest/testfile.txt
hdfsGetDefaultBlockSizeAtPath- SUCCESS!
HDFS Capacity : 102726873909
hdfsGetCapacity- SUCCESS!
HDFS Used : 4767076324
hdfsGetCapacity- SUCCESS!
hdfsExists- SUCCESS! /tmp/nativeTest/testfile.txt
hdfsConfGetStr- SUCCESS : hdfs://hacluster
hdfsStreamBuilderBuild- SUCCESS! File read success. Byte read : 31, File contant : Hadoop HDFS
Native file write!
hdfsFileGetReadStatistics- SUCCESS! totalBytesRead : 31, totalLocalBytesRead : 0,
totalShortCircuitBytesRead : 0, totalZeroCopyBytesRead : 0
```

7. 进入debug模式（可选）

   make gdb

   执行结果如下：

```
[root@10-120-85-2 hdfs-c-example]# make gdb
gdb hdfs_test
GNU gdb (GDB) SUSE (7.5.1-0.7.29)
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-suse-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /opt/client/HDFS/hadoop/hdfs-c-example/hdfs_test...done.
(gdb)
```

13.7.1.3 HDFS HTTP REST API 接口介绍

功能简介

REST应用开发代码样例中所涉及的文件操作主要包括创建文件、读写文件、追加文件、删除文件。完整和详细的接口请参考官网上的描述以了解其使用：

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/WebHDFS.html

准备运行环境

步骤1 安装集群客户端，例如客户端安装目录为“/opt/client”。

1. 执行下列命令进行用户认证，这里以**hdfs**为例，用户可根据实际用户名修改。

   kinit hdfs

   说明

   kinit认证的默认时效为24小时，到期后再次运行样例，需要重新执行**kinit**。

2. 在客户端目录创建文件“testFile”和“testFileAppend”，文件内容分别为“Hello, webhdfs user!”和“Welcome back to webhdfs!”。

   touch testFile

   vi testFile


```
Hello, webhdfs user!
touch testFileAppend
vi testFileAppend
Welcome back to webhdfs!
```

步骤2 MRS集群默认只支持HTTPS服务访问，若使用HTTPS服务访问，执行**步骤3**；若使用HTTP服务访问，执行**步骤4**。

步骤3 与HTTP服务访问相比，以HTTPS方式访问HDFS时，由于使用了SSL安全加密，需要确保Curl命令所支持的SSL协议在集群中已添加支持。若不支持，可对应修改集群中SSL协议。例如，若Curl仅支持TLSv1协议，修改方法如下：

登录FusionInsight Manager页面，选择“集群 > *待操作集群的名称* > 服务 > HDFS > 配置 > 全部配置”，在“搜索”框里搜索“hadoop.ssl.enabled.protocols”，查看参数值是否包含“TLSv1”，若不包含，则在配置项“hadoop.ssl.enabled.protocols”中追加“,TLSv1”。清空“ssl.server.exclude.cipher.list”配置项的值，否则以HTTPS访问不了HDFS。单击“保存 > 确定”，保存完成后重启HDFS服务。

说明

TLSv1协议存在安全漏洞，请谨慎使用。

步骤4 **登录FusionInsight Manager页面**，单击“集群 > *待操作集群的名称* > 服务 > HDFS > 配置 >全部配置”，在“搜索”框里搜索“dfs.http.policy”，然后勾选“HTTP_AND_HTTPS”，单击“保存”，单击“更多 > 重启服务”重启HDFS服务。

----结束

操作步骤

步骤1 **登录FusionInsight Manager页面**，单击“集群 > *待操作集群的名称* > 服务”，选择“HDFS”，单击进入HDFS服务状态页面。

说明

由于webhdfs是http/https访问的，需要主NameNode的IP和http/https端口。

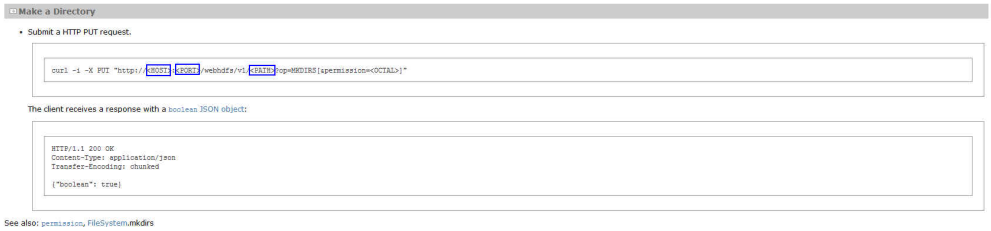
1. 单击“实例”，进入HDFS实例界面，找到“NameNode(hacluster,主)”的主机名（host）和对应的IP。
2. 单击“配置”，进入HDFS服务配置界面，找到“namenode.http.port”（9870）和“namenode.https.port”（9871）。

步骤2 参考如下链接，创建目录。

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/WebHDFS.html#Make_a_Directory

单击链接，如**图13-10**所示。

图 13-10 创建目录样例命令



进入到客户端的安装目录下，此处为“/opt/client”，创建名为“huawei”的目录。

1. 执行下列命令，查看当前是否存在名为“huawei”的目录。

   hdfs dfs -ls /

   执行结果如下：

   ```
   linux1:/opt/client # hdfs dfs -ls /
   16/04/22 16:10:02 INFO hdfs.PeerCache: SocketCache disabled.
   Found 7 items
   -rw-r--r--   3 hdfs   supergroup          0 2016-04-20 18:03 /PRE_CREATE_DIR.SUCCESS
   drwxr-x---   - flume  hadoop              0 2016-04-20 18:02 /flume
   drwx------   - hbase  hadoop              0 2016-04-22 15:19 /hbase
   drwxrwxrwx   - mapred hadoop              0 2016-04-20 18:02 /mr-history
   drwxrwxrwx   - spark  supergroup          0 2016-04-22 15:19 /sparkJobHistory
   drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 14:51 /tmp
   drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 14:50 /user
   ```

   当前路径下不存在“huawei”目录。

2. 执行**图13-10**中的命令创建以“huawei”为名的目录。其中，用**步骤1**中查找到的主机名或IP和端口分别替代命令中的<HOST>和<PORT>，在<PATH>中输入想要创建的目录“huawei”。

   说明

   用主机名或IP代替<HOST>都是可以的，要注意HTTP和HTTPS的端口不同。

   - 执行下列命令访问HTTP：

     ```
     linux1:/opt/client # curl -i -X PUT --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei?op=MKDIRS"
     ```

     其中用linux1代替<HOST>，用9870代替<PORT>。

   - 运行结果：

     ```
     HTTP/1.1 401 Authentication required
     Date: Thu, 05 May 2016 03:10:09 GMT
     Pragma: no-cache
     Date: Thu, 05 May 2016 03:10:09 GMT
     Pragma: no-cache
     X-Frame-Options: SAMEORIGIN
     WWW-Authenticate: Negotiate
     Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; HttpOnly
     Content-Length: 0
     HTTP/1.1 200 OK
     Cache-Control: no-cache
     Expires: Thu, 05 May 2016 03:10:09 GMT
     Date: Thu, 05 May 2016 03:10:09 GMT
     Pragma: no-cache
     Expires: Thu, 05 May 2016 03:10:09 GMT
     Date: Thu, 05 May 2016 03:10:09 GMT
     Pragma: no-cache
     Content-Type: application/json
     X-Frame-Options: SAMEORIGIN
     WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
     +iTTBLoAMCARKiRARCArhuv39Ttp6lhBlG3B0JAmFjv9weLp+SGFI+t2HSEHN6p4UVWKKy/
     kd9dKEgNMlyDu/o7ytzs0cqMxNsI69WbN5H
     Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名
     >&t=kerberos&e=1462453809395&s=wiRF4rdTWpm3tDST+a/Sy0lwgA4="; Path=/; Expires=Thu,
     05-May-2016 13:10:09 GMT; HttpOnly
     Transfer-Encoding: chunked
     {"boolean":true}linux1:/opt/client #
     ```

     返回值{"boolean":true}说明创建成功。

   - 执行下列命令访问HTTPS：

     ```
     linux1:/opt/client # curl -i -k -X PUT --negotiate -u: "https://10.120.172.109:9871/webhdfs/v1/huawei?op=MKDIRS"
     ```

     其中用IP10.120.172.109代替<HOST>，用9871代替<PORT>。

- 运行结果：

```
HTTP/1.1 401 Authentication required
Date: Fri, 22 Apr 2016 08:13:37 GMT
Pragma: no-cache
Date: Fri, 22 Apr 2016 08:13:37 GMT
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate
Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; Secure; HttpOnly
Content-Length: 0
HTTP/1.1 200 OK
Cache-Control: no-cache
Expires: Fri, 22 Apr 2016 08:13:37 GMT
Date: Fri, 22 Apr 2016 08:13:37 GMT
Pragma: no-cache
Expires: Fri, 22 Apr 2016 08:13:37 GMT
Date: Fri, 22 Apr 2016 08:13:37 GMT
Pragma: no-cache
Content-Type: application/json
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
+iTTBLoAMCARKiRARCugB+yT3Y+z8YCRMYJHXF84o1cyCfJq157+NZN1gu7D7yhMULnjr
+7BuUdEcZKewFR7uD+DRiMY3akg3OgU45xQ9R
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名
>&t=kerberos&e=1461348817963&s=sh57G7iVccX/Aknoz410yJPTLHg="; Path=/; Expires=Fri, 22-
Apr-2016 18:13:37 GMT; Secure; HttpOnly
Transfer-Encoding: chunked

{"boolean":true}linux1:/opt/client #
```

返回值{"boolean":true}说明创建成功。

3. 再执行下列命令进行查看，可以看到路径下出现“huawei”目录。

```
linux1:/opt/client # hdfs dfs -ls /
16/04/22 16:14:25 INFO hdfs.PeerCache: SocketCache disabled.
Found 8 items
-rw-r--r--   3 hdfs   supergroup          0 2016-04-20 18:03 /PRE_CREATE_DIR.SUCCESS
drwxr-x---   - flume  hadoop              0 2016-04-20 18:02 /flume
drwx------   - hbase  hadoop              0 2016-04-22 15:19 /hbase
drwxr-xr-x   - hdfs   supergroup          0 2016-04-22 16:13 /huawei
drwxrwxrwx   - mapred hadoop              0 2016-04-20 18:02 /mr-history
drwxrwxrwx   - spark  supergroup          0 2016-04-22 16:12 /sparkJobHistory
drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 14:51 /tmp
drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 16:10 /user
```

步骤3 创建请求上传命令，获取集群分配的可写入DataNode节点地址的信息Location。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -i -X PUT --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?
op=CREATE"
```

- 运行结果：

```
HTTP/1.1 401 Authentication required
Date: Thu, 05 May 2016 06:09:48 GMT
Pragma: no-cache
Date: Thu, 05 May 2016 06:09:48 GMT
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate
Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; HttpOnly
Content-Length: 0

HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Thu, 05 May 2016 06:09:48 GMT
Date: Thu, 05 May 2016 06:09:48 GMT
Pragma: no-cache
Expires: Thu, 05 May 2016 06:09:48 GMT
Date: Thu, 05 May 2016 06:09:48 GMT
Pragma: no-cache
Content-Type: application/octet-stream
```



```
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
+iTTBLoAMCARKiRARCzQ6w
+9pNzWCTJEdoU3z9xKEyg1JQNka0nYaB9TndvrL5S0neAoK2usnictTFnqIincAjwB6SnTtht8Q16WDlHJX/
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名
>&t=kerberos&e=1462464588403&s=qry87vAyYzSn9VsS6Rm6vKLhKeU="; Path=/; Expires=Thu, 05-
May-2016 16:09:48 GMT; HttpOnly
Location: http://linux1:9864/webhdfs/v1/huawei/testHdfs?
op=CREATE&delegation=HgAFYWRtaW4FYWRtaW4AigFUf4lZdIoBVKOV3XQOCBSyXvFAp92alcRs4j-
KNulnN6wUoBJXRUJIREZTIGRlbGVnYXRpb24UMTAuMTIwLjE3Mi4xMDk6MjUwMDA&namenoderpcaddr
ess=hacluster&overwrite=false
Content-Length: 0
```

- 执行如下命令访问HTTPS：

```
linux1:/opt/client # curl -i -k -X PUT --negotiate -u: "https://linux1:9871/webhdfs/v1/huawei/testHdfs?
op=CREATE"
```

- 运行结果：

```
HTTP/1.1 401 Authentication required
Date: Thu, 05 May 2016 03:46:18 GMT
Pragma: no-cache
Date: Thu, 05 May 2016 03:46:18 GMT
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate
Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; Secure; HttpOnly
Content-Length: 0

HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Thu, 05 May 2016 03:46:18 GMT
Date: Thu, 05 May 2016 03:46:18 GMT
Pragma: no-cache
Expires: Thu, 05 May 2016 03:46:18 GMT
Date: Thu, 05 May 2016 03:46:18 GMT
Pragma: no-cache
Content-Type: application/octet-stream
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
+iTTBLoAMCARKiRARCZMYR8GGUkn7pPZaoOYZD5HxzLTRZ71angUHKubW2wC/18m9/
OOZstGQ6M1wH2pGriipuCNsKIfwP93eO2Co0fQF3
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名
>&t=kerberos&e=1462455978166&s=F4rXUwEevHZze3PR8TxkzcV7RQQ="; Path=/; Expires=Thu, 05-
May-2016 13:46:18 GMT; Secure; HttpOnly
Location: https://linux1:9865/webhdfs/v1/huawei/testHdfs?
op=CREATE&delegation=HgAFYWRtaW4FYWRtaW4AigFUfwX3t4oBVKMSe7cCCBSFJTi9j7X64QwnSz59T
GFPKFf7GhNTV0VCSERGUyBkZWxlZ2F0aW9uFDEwLjEyMC4xNzIuMTA5OjI1MDAw&namenoderpcaddr
ess=hacluster&overwrite=false
Content-Length: 0
```

步骤4 根据获取的Location地址信息，可在HDFS文件系统上创建“/huawei/testHdfs”文件，并将本地“testFile”中的内容上传至“testHdfs”文件。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -i -X PUT -T testFile --negotiate -u: "http://linux1:9864/webhdfs/v1/huawei/
testHdfs?
op=CREATE&delegation=HgAFYWRtaW4FYWRtaW4AigFUf4lZdIoBVKOV3XQOCBSyXvFAp92alcRs4j-
KNulnN6wUoBJXRUJIREZTIGRlbGVnYXRpb24UMTAuMTIwLjE3Mi4xMDk6MjUwMDA&namenoderpcaddr
ess=hacluster&overwrite=false"
```

- 运行结果：

```
HTTP/1.1 100 Continue
HTTP/1.1 201 Created
Location: hdfs://hacluster/huawei/testHdfs
Content-Length: 0
Connection: close
```

- 执行如下命令访问HTTPS：

```
linux1:/opt/client # curl -i -k -X PUT -T testFile --negotiate -u: "https://linux1:9865/webhdfs/v1/
huawei/testHdfs?
op=CREATE&delegation=HgAFYWRtaW4FYWRtaW4AigFUfwX3t4oBVKMSe7cCCBSFJTi9j7X64QwnSz59T
```



```
GFPKFf7GhNTV0VCSERGUyBkZWxlZ2F0aW9uFDEwLjEyMC4xNzIuMTA5OjI1MDAw&namenoderpcaddr
ess=hacluster&overwrite=false"
```

- 运行结果：

```
HTTP/1.1 100 Continue
HTTP/1.1 201 Created
Location: hdfs://hacluster/huawei/testHdfs
Content-Length: 0
Connection: close
```

步骤5 打开“/huawei/testHdfs”文件，并读取文件中上传写入的内容。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -L --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?op=OPEN"
```

- 运行结果：

```
Hello, webhdfs user!
```

- 执行如下命令访问HTTPS：

```
linux1:/opt/client # curl -k -L --negotiate -u: "https://linux1:9871/webhdfs/v1/huawei/testHdfs?
op=OPEN"
```

- 运行结果：

```
Hello, webhdfs user!
```

步骤6 创建请求追加文件的命令，获取集群为已存在“/huawei/testHdfs”文件分配的可写入DataNode节点地址信息Location。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -i -X POST --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?
op=APPEND"
```

- 运行结果：

```
HTTP/1.1 401 Authentication required
Cache-Control: must-revalidate,no-cache,no-store
Date: Thu, 05 May 2016 05:35:02 GMT
Pragma: no-cache
Date: Thu, 05 May 2016 05:35:02 GMT
Pragma: no-cache
Content-Type: text/html; charset=iso-8859-1
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate
Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; HttpOnly
Content-Length: 1349

HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Thu, 05 May 2016 05:35:02 GMT
Date: Thu, 05 May 2016 05:35:02 GMT
Pragma: no-cache
Expires: Thu, 05 May 2016 05:35:02 GMT
Date: Thu, 05 May 2016 05:35:02 GMT
Pragma: no-cache
Content-Type: application/octet-stream
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
+iTTBLoAMCARKiRARCTYvNX/2JMXhzsVPTw3Sluox6s/gEroHH980xMBkkYlCnO3W+0fM32c4/
F98U5bl5dzgoolQoBvqq/EYXivvR12WX
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名
>&t=kerberos&e=1462462502626&s=et1okVIOd7DWJ/LdhzNeS2wQEEY="; Path=/; Expires=Thu, 05-
May-2016 15:35:02 GMT; HttpOnly
Location: http://linux1:9864/webhdfs/v1/huawei/testHdfs?
op=APPEND&delegation=HgAFYWRtaW4FYWRtaW4AigFUf2mGHooBVKN2Ch4KCBRzjM3jwSMlAowXb
4dhqfKB5rT-8hJXRUJIREZTIGRlbGVnYXRpb24UMTAuMTIwLjE3Mi4xMDk6MjUwMDA&namenoderpcadd
ress=hacluster
Content-Length: 0
```

- 执行如下命令访问HTTPS：

```
linux1:/opt/client # curl -i -k -X POST --negotiate -u: "https://linux1:9871/webhdfs/v1/huawei/
testHdfs?op=APPEND"
```

- 运行结果：

```
HTTP/1.1 401 Authentication required
Cache-Control: must-revalidate,no-cache,no-store
Date: Thu, 05 May 2016 05:20:41 GMT
Pragma: no-cache
Date: Thu, 05 May 2016 05:20:41 GMT
Pragma: no-cache
Content-Type: text/html; charset=iso-8859-1
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate
Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; Secure; HttpOnly
Content-Length: 1349

HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Thu, 05 May 2016 05:20:41 GMT
Date: Thu, 05 May 2016 05:20:41 GMT
Pragma: no-cache
Expires: Thu, 05 May 2016 05:20:41 GMT
Date: Thu, 05 May 2016 05:20:41 GMT
Pragma: no-cache
Content-Type: application/octet-stream
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
+iTTBLoAMCARKiRARCXgdjZuoxLHGtM1oyrPcXk95/
Y869eMfXIQV5UdEwBZ0iQiYaOdf5+Vk7a7FezhmzCABOWYXPxEQPNugbZ/yD5VLT
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名
>&t=kerberos&e=1462461641713&s=tGwwOH9scmnNtxPjlnu28SFtex0="; Path=/; Expires=Thu, 05-
May-2016 15:20:41 GMT; Secure; HttpOnly
Location: https://linux1:9865/webhdfs/v1/huawei/testHdfs?
op=APPEND&delegation=HgAFYWRtaW4FYWRtaW4AigFUf1xi_4oBVKNo5v8HCBSE3Fg0f_EwtFKKlODK
QSM2t32CjhNTV0VCSERGUyBkZWxlZ2F0aW9uFDEwLjEyMC4xNzIuMTA5OjI1MDAw&namenoderpcadd
ress=hacluster
```

步骤7 根据获取的Location地址信息，可将本地“testFileAppend”文件中的内容追加到HDFS文件系统上的“/huawei/testHdfs”文件。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -i -X POST -T testFileAppend --negotiate -u: "http://linux1:9864/webhdfs/v1/
huawei/testHdfs?
op=APPEND&delegation=HgAFYWRtaW4FYWRtaW4AigFUf2mGHooBVKN2Ch4KCBRzjM3jwSMlAowXb
4dhqfKB5rT-8hJXRUJIREZTIGRlbGVnYXRpb24UMTAuMTIwLjE3Mi4xMDk6MjUwMDA&namenoderpcadd
ress=hacluster"
```

- 运行结果：

```
HTTP/1.1 100 Continue
HTTP/1.1 200 OK
Content-Length: 0
Connection: close
```

- 执行如下命令访问HTTPS：

```
linux1:/opt/client # curl -i -k -X POST -T testFileAppend --negotiate -u: "https://linux1:9865/
webhdfs/v1/huawei/testHdfs?
op=APPEND&delegation=HgAFYWRtaW4FYWRtaW4AigFUf1xi_4oBVKNo5v8HCBSE3Fg0f_EwtFKKlODK
QSM2t32CjhNTV0VCSERGUyBkZWxlZ2F0aW9uFDEwLjEyMC4xNzIuMTA5OjI1MDAw&namenoderpcadd
ress=hacluster"
```

- 运行结果：

```
HTTP/1.1 100 Continue
HTTP/1.1 200 OK
Content-Length: 0
Connection: close
```

步骤8 打开“/huawei/testHdfs”文件，并读取文件中全部的内容。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -L --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?op=OPEN"
```

- 运行结果：

```
Hello, webhdfs user!
Welcome back to webhdfs!
```


- 执行如下命令访问HTTPS：
```
linux1:/opt/client # curl -k -L --negotiate -u: "https://linux1:9871/webhdfs/v1/huawei/testHdfs?op=OPEN"
```
- 运行结果：
```
Hello, webhdfs user!
Welcome back to webhdfs!
```

步骤9 可列出文件系统上“huawei”目录下所有目录和文件的详细信息。

LISTSTATUS将在一个请求中返回所有子文件和文件夹的信息。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?op=LISTSTATUS"
```
- 运行结果：
```
{"FileStatuses":{"FileStatus":[
{"accessTime":1462425245595,"blockSize":134217728,"childrenNum":0,"fileId":17680,"group":"supergroup","length":70,"modificationTime":1462426678379,"owner":"hdfs","pathSuffix":"","permission":"755","replication":3,"storagePolicy":0,"type":"FILE"}
]}}
```
- 执行如下命令访问HTTPS：
```
linux1:/opt/client # curl -k --negotiate -u: "https://linux1:9871/webhdfs/v1/huawei/testHdfs?op=LISTSTATUS"
```
- 运行结果：
```
{"FileStatuses":{"FileStatus":[
{"accessTime":1462425245595,"blockSize":134217728,"childrenNum":0,"fileId":17680,"group":"supergroup","length":70,"modificationTime":1462426678379,"owner":"hdfs","pathSuffix":"","permission":"755","replication":3,"storagePolicy":0,"type":"FILE"}
]}}
```

带有大小参数和startafter参数的LISTSTATUS将有助于通过多个请求获取子文件和文件夹信息，从而避免获取大量子文件和文件夹信息时，用户界面变慢。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/?op=LISTSTATUS&startafter=sparkJobHistory&size=1"
```
- 运行结果：
```
{"FileStatuses":{"FileStatus":[
{"accessTime":1462425245595,"blockSize":134217728,"childrenNum":0,"fileId":17680,"group":"supergroup","length":70,"modificationTime":1462426678379,"owner":"hdfs","pathSuffix":"testHdfs","permission":"755","replication":3,"storagePolicy":0,"type":"FILE"}
]}}
```
- 执行如下命令访问HTTPS：
```
linux1:/opt/client # curl -k --negotiate -u: "https://linux1:9871/webhdfs/v1/huawei/?op=LISTSTATUS&startafter=sparkJobHistory&size=1"
```
- 运行结果：
```
{"FileStatuses":{"FileStatus":[
{"accessTime":1462425245595,"blockSize":134217728,"childrenNum":0,"fileId":17680,"group":"supergroup","length":70,"modificationTime":1462426678379,"owner":"hdfs","pathSuffix":"testHdfs","permission":"755","replication":3,"storagePolicy":0,"type":"FILE"}
]}}
```

步骤10 删除HDFS上的文件“/huawei/testHdfs”。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl -i -X DELETE  --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?op=DELETE"
```
- 运行结果：
```
HTTP/1.1 401 Authentication required
Date: Thu, 05 May 2016 05:54:37 GMT
Pragma: no-cache
Date: Thu, 05 May 2016 05:54:37 GMT
```



```
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate
Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; HttpOnly
Content-Length: 0
HTTP/1.1 200 OK
Cache-Control: no-cache
Expires: Thu, 05 May 2016 05:54:37 GMT
Date: Thu, 05 May 2016 05:54:37 GMT
Pragma: no-cache
Expires: Thu, 05 May 2016 05:54:37 GMT
Date: Thu, 05 May 2016 05:54:37 GMT
Pragma: no-cache
Content-Type: application/json
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
+iTTBLoAMCARKiRARC9k0/v6Ed8VlUBy3kuT0b4RkqkNMCrDevsLGQOUQRORkzWI3Wu
+XLJUMKlmZaWpP+bPzpx8O2Od81mLBgdi8sOkLw
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名
>&t=kerberos&e=1462463677153&s=Pwxe5UIqaULjFb9R6ZwlSX85GoI="; Path=/; Expires=Thu, 05-
May-2016 15:54:37 GMT; HttpOnly
Transfer-Encoding: chunked
{"boolean":true}linux1:/opt/client #
```

- 执行如下命令访问HTTPS：

```
linux1:/opt/client # curl -i -k -X DELETE --negotiate -u: "https://linux1:9871/webhdfs/v1/huawei/
testHdfs?op=DELETE"
```

- 运行结果：

```
HTTP/1.1 401 Authentication required
Date: Thu, 05 May 2016 06:20:10 GMT
Pragma: no-cache
Date: Thu, 05 May 2016 06:20:10 GMT
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate
Set-Cookie: hadoop.auth=; Path=/; Expires=Thu, 01-Jan-1970 00:00:00 GMT; Secure; HttpOnly
Content-Length: 0
HTTP/1.1 200 OK
Cache-Control: no-cache
Expires: Thu, 05 May 2016 06:20:10 GMT
Date: Thu, 05 May 2016 06:20:10 GMT
Pragma: no-cache
Expires: Thu, 05 May 2016 06:20:10 GMT
Date: Thu, 05 May 2016 06:20:10 GMT
Pragma: no-cache
Content-Type: application/json
X-Frame-Options: SAMEORIGIN
WWW-Authenticate: Negotiate YGoGCSqGSIb3EgECAgIAb1swWaADAgEFoQMCAQ
+iTTBLoAMCARKiRARCLY5vrVmgsiH2VWRypc30iZGffRUf4nXNaHCWni3TIDUOTl+S+hfjatSbo/+uayQI/
6k9jAfaJrvFIfxqppFtofpp
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs@<系统域名>&t=kerberos&e=1462465210180&s=KGd2SbH/
EUSaaeVKCb5zPzGBRKo="; Path=/; Expires=Thu, 05-May-2016 16:20:10 GMT; Secure; HttpOnly
Transfer-Encoding: chunked
{"boolean":true}linux1:/opt/client #
```

----结束

密钥管理系统通过HTTP REST API对外提供密钥管理服务，接口请参考官网：

http://hadoop.apache.org/docs/r3.1.1/hadoop-kms/index.html

说明

由于REST API接口做了安全加固，防止脚本注入攻击。通过REST API的接口，无法创建包含"<script ", "<iframe", "<frame", "javascript:" 这些关键字的目录和文件名。

13.7.2 HDFS Shell 命令介绍

HDFS Shell

您可以使用HDFS Shell命令对HDFS文件系统进行操作，例如读文件、写文件等操作。

执行HDFS Shell的方法：

进入HDFS客户端如下目录，直接输入命令即可。例如：

cd /opt/client/HDFS/hadoop/bin

./hdfs dfs -mkdir /tmp/input

执行如下命令查询HDFS命令的帮助。

./hdfs --help

HDFS命令行参考请参见官网：

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-common/FileSystemShell.html

表 13-14 透明加密相关命令

场景	操作	命令	描述
hadoop shell命令管理密钥	创建密钥	***hadoop key create*** *<keyname> [-cipher <cipher>] [-size <size>] [-description <description>] [-attr <attribute=value>] [-provider <provider>] [-help]*	create子命令为provider中<keyname>参数指定的name创建一个新的密钥，provider是由-provider参数指定。用户可以使用参数-cipher定义一个密码。目前默认的密码为"AES/CTR/NoPadding"。 默认密钥的长度为128。用户可以使用参数-size定义需要的密钥的长度。任意的attribute=value类型属性可以用参数-attr定义。每一个属性，-attr可以被定义很多次。
	回滚操作	***hadoop key roll*** *<keyname> [-provider <provider>] [-help]*	roll子命令为provider中指定的key创建一个新的版本，provider是由-provider参数指定。
	删除密钥	***hadoop key delete*** *<keyname> [-provider <provider>] [-f] [-help]*	delete子命令删除key的所有版本，key是由provider中的<keyname>参数指定，provider是由-provider参数指定。除非-f被指定否则该命令需要用户确认。
	查看密钥	***hadoop key list*** *[-provider <provider>] [-metadata] [-help]*	list子命令显示provider中所有的密钥名，这个provider由用户在core-site.xml中配置或者由-provider参数指定。-metadata参数显示的是元数据。

表 13-15 Colocation 客户端 shell 命令

操作	命令	描述
创建组	hdfs colocationadmin -createGroup -groupId <groupID> -locatorIds <comma separated locatorIDs> or -file <path of the file contains all of locatorIDs>	创建组，groupID为组名，locatorID为locator名，locatorID可通过命令行输入，多个locatorID之间用逗号分隔；也可将locatorID写入文件，通过读文件获取。
删除组	hdfs colocationadmin -deleteGroup <groupID>	删除指定组。
查询组	hdfs colocationadmin -queryGroup <groupID>	查询指定组的详细信息，包括该group包含的locators以及每个locator及其对应的DataNode。
查看所有组	hdfs colocationadmin -listGroups	列出所有组及其创建时间。
设置colocation根目录的acl权限	hdfs colocationadmin -setAcl	设置zookeeper中colocation根目录的acl权限。 colocation在zookeeper中的根目录默认为/hadoop/colocationDetails。

13.7.3 配置 Windows 通过 EIP 访问安全模式集群 HDFS

操作场景

该章节通过指导用户配置集群绑定EIP，并配置HDFS文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行HdfsExample样例为例进行说明。

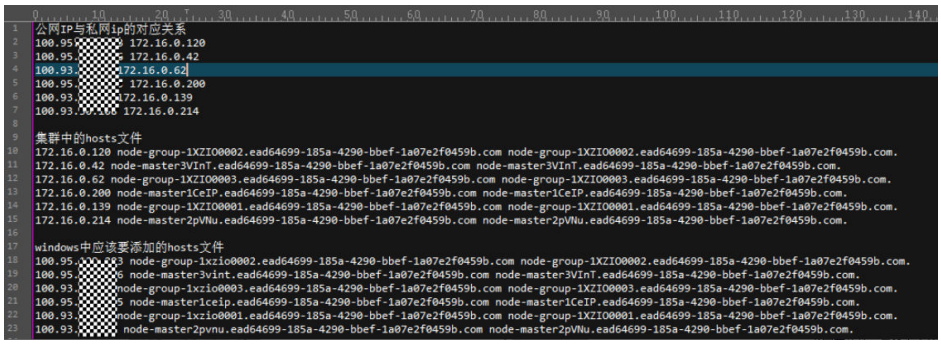
操作步骤

步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。



步骤2 将krb5.conf文件中的IP地址修改为对应IP的主机名称。

步骤3 配置集群安全组规则。

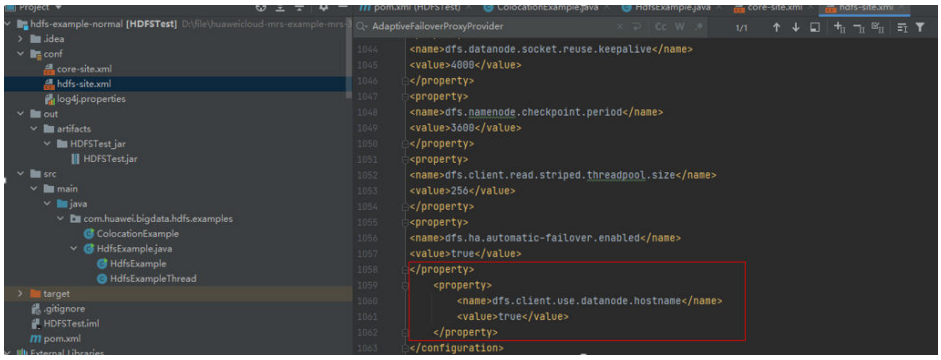
1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”配置Windows的IP和21730TCP、21731TCP/UDP、21732TCP/UDP端口。

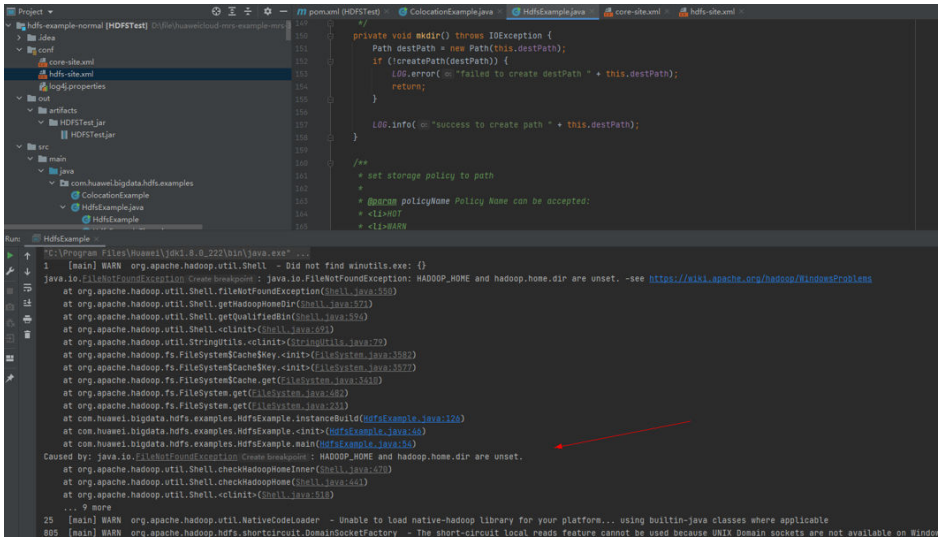
步骤4 在Manager界面选择“集群 > 服务 > HDFS > 更多 > 下载客户端”，将客户端中的core-site.xml和hdfs-site.xml复制到样例工程的conf目录下，并对hdfs-site.xml添加以下内容：

```
<property>
 <name>dfs.client.use.datanode.hostname</name>
 <value>true</value>
 </property>
```

（将datanode的通信改成通过hostname）



修改该操作后，在运行样例工程时，可能会报一个没有hadoop_home的异常，这个不影响使用，可以忽略。



步骤5 在运行样例代码前，需要将样例代码中的PRNCIPAL_NAME改为安全认证的用户名。

----结束

14 HDFS 开发指南（普通模式）

14.1 HDFS 应用开发简介

HDFS 简介

HDFS（Hadoop Distribute File System）是一个适合运行在通用硬件之上，具备高度容错特性，支持高吞吐量数据访问的分布式文件系统，非常适合大规模数据集应用。

HDFS适用于如下场景：

- 处理海量数据（TB或PB级别以上）
- 需要很高的吞吐量
- 需要高可靠性
- 需要很好的可扩展能力

HDFS 开发接口简介

HDFS支持使用Java语言进行程序开发，具体的API接口内容请参考**HDFS Java API接口介绍**。

常用概念

- Colocation

  同分布（Colocation）功能是将存在关联关系的数据或可能要进行关联操作的数据存储在相同的存储节点上。HDFS文件同分布的特性是，将那些需进行关联操作的文件存放在相同的数据节点上，在进行关联操作计算时，避免了到别的数据节点上获取数据的动作，大大降低了网络带宽的占用。

- Client

  HDFS Client主要包括五种方式：JAVA API、C API、Shell、HTTP REST API、WEB UI五种方式，可参考**HDFS常用API介绍**、**HDFS Shell命令介绍**。

  - JAVA API

    提供HDFS文件系统的应用接口，本开发指南主要介绍如何使用Java API进行HDFS文件系统的应用开发。

  - C API

提供HDFS文件系统的应用接口，使用C语言开发的用户可参考C接口的描述进行应用开发。

- Shell

  提供shell命令完成HDFS文件系统的基本操作。

- HTTP REST API

  提供除Shell、Java API和C API以外的其他接口，可通过此接口监控HDFS状态等信息。

- WEB UI

  提供Web可视化组件管理界面。

14.2 HDFS 应用开发流程介绍

开发流程中各阶段的说明如图14-1和表14-1所示。

图 14-1 HDFS 应用程序开发流程

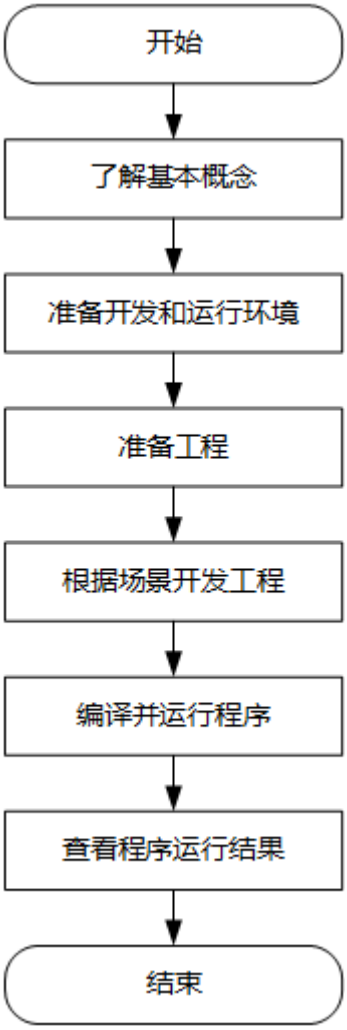


表 14-1 HDFS 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解HDFS的基本概念。	**常用概念**
准备开发和运行环境	使用IntelliJ IDEA工具，请根据指导完成开发环境配置。 HDFS的运行环境即HDFS客户端，请根据指导完成客户端的安装和配置。	**准备HDFS应用开发和运行环境**
准备工程	HDFS提供了不同场景下的样例程序，可以导入样例工程进行程序学习。	**导入并配置HDFS样例工程**
根据场景开发工程	提供样例工程，帮助用户快速了解HDFS各部件的编程接口。	**开发HDFS应用**
编译并运行程序	指导用户将开发好的程序编译并提交运行。	**调测HDFS应用**
查看程序运行结果	程序运行结果会写在用户指定的路径下。用户还可以通过UI查看应用运行情况。	**调测HDFS应用**

14.3 HDFS 样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下HDFS相关样例工程：

表 14-2 HDFS 相关样例工程

样例工程位置	描述
hdfs-example-normal	HDFS文件操作的Java示例程序。 本工程主要给出了创建HDFS文件夹、写文件、追加文件内容、读文件和删除文件/文件夹等相关接口操作示例。
hdfs-c-example	HDFS C语言开发代码样例。 本示例提供了基于C语言的HDFS文件系统连接、文件操作如创建文件、读写文件、追加文件、删除文件等。相关业务场景介绍请参见**HDFS C API接口介绍**。

14.4 准备 HDFS 应用开发环境

14.4.1 准备 HDFS 应用开发和运行环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如表14-3所示。

表 14-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境：Windows或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置，版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件。 支持7-Zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果本地Windows开发环境和集群业务平面网络互通，可下载集群客户端到本地，获取相关调测程序所需的集群配置文件及配置网络连通后，然后直接在Windows中进行程序调测。

a. **登录FusionInsight Manager页面**，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“完整客户端”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig.tar”，继续解压该文件。解压到本地PC的“D:\FusionInsight_Cluster_1_Services_ClientConfig”目录下（路径中不能有空格）。

b. 进入客户端解压路径“FusionInsight_Cluster_1_Services_ClientConfig\HDFS\config”，手动将配置文件导入到HDFS样例工程的配置文件目录中（通常为“conf”文件夹）。

c. 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与解压目录下“hosts”文件中所列出的各主机在网络上互通。

说明

- 当客户端所在主机不是集群中的节点时，配置客户端网络连接，可避免执行客户端命令时出现错误。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 如果使用Linux环境调测程序，需准备安装集群客户端的Linux节点并获取相关配置文件。

a. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。

客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

b. **登录FusionInsight Manager页面**，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig/HDFS/config”路径下的所有配置文件至客户端节点，放置到工程代码的conf文件夹下。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp HDFS/config/* root@*客户端节点IP地址*:/opt/client/conf

表 14-4 配置文件

文件名称	作用
core-site.xml	配置HDFS详细参数。
hdfs-site.xml	配置HDFS详细参数。

说明

“conf”目录下的“log4j.properties”文件客户根据自己的需要进行配置。

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

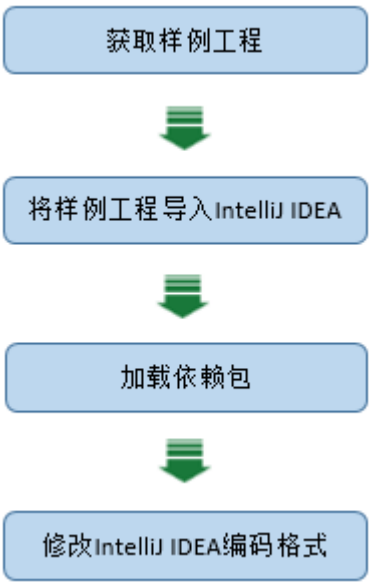
14.4.2 导入并配置 HDFS 样例工程

操作场景

HDFS针对多个场景提供样例工程，帮助客户快速学习HDFS工程。

以下操作步骤以导入HDFS样例代码为例。操作流程如图14-2所示。

图 14-2 导入样例工程流程



操作步骤

步骤1 参考获取MRS应用开发样例工程，获取样例代码解压目录中“src”目录下的样例工程文件夹“hdfs-example-normal”。

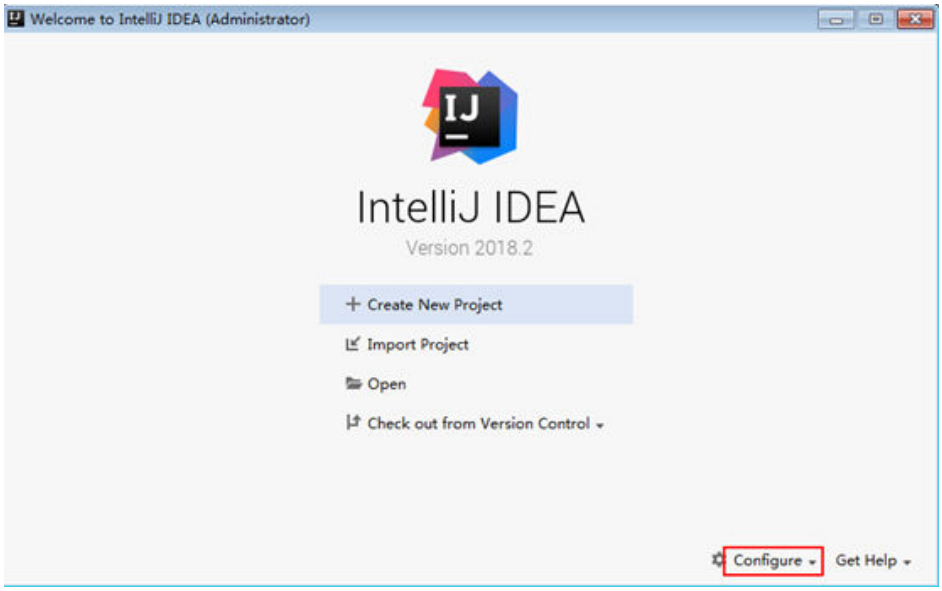
步骤2 导入样例工程到IntelliJ IDEA开发环境。

1. 打开IntelliJ IDEA，依次选择“File > Open”。
2. 在弹出的Open File or Project对话框中选择样例工程文件夹“hdfs-example-normal”，单击“OK”。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

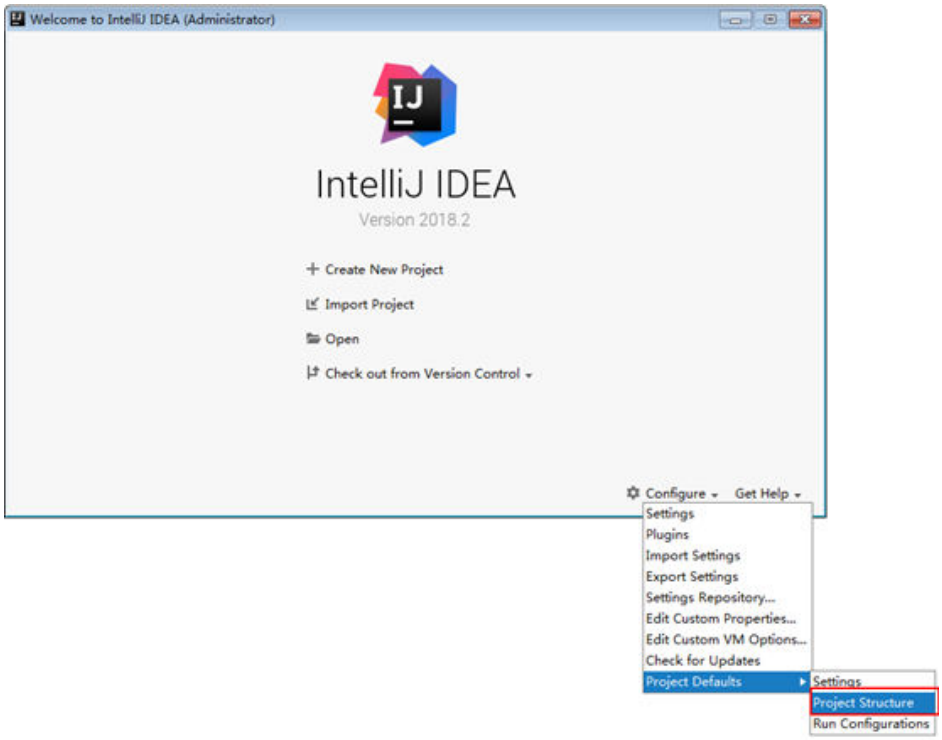
1. 打开IntelliJ IDEA，选择“Configure”。

图 14-3 Quick Start



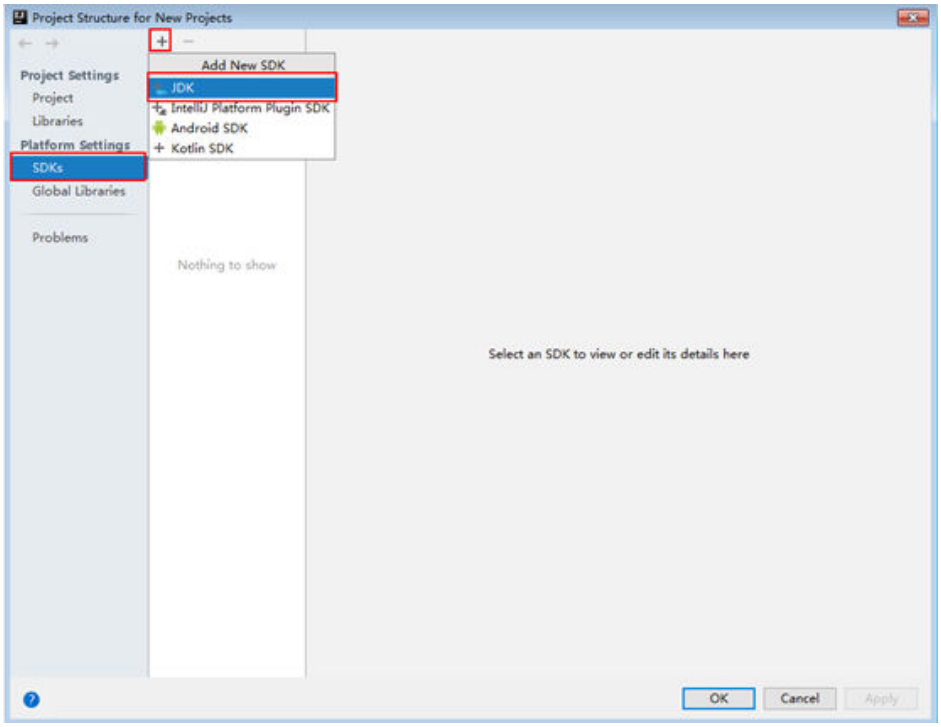
2. 在下拉框中选择“Project Defaults > Project Structure”。

图 14-4 Configure



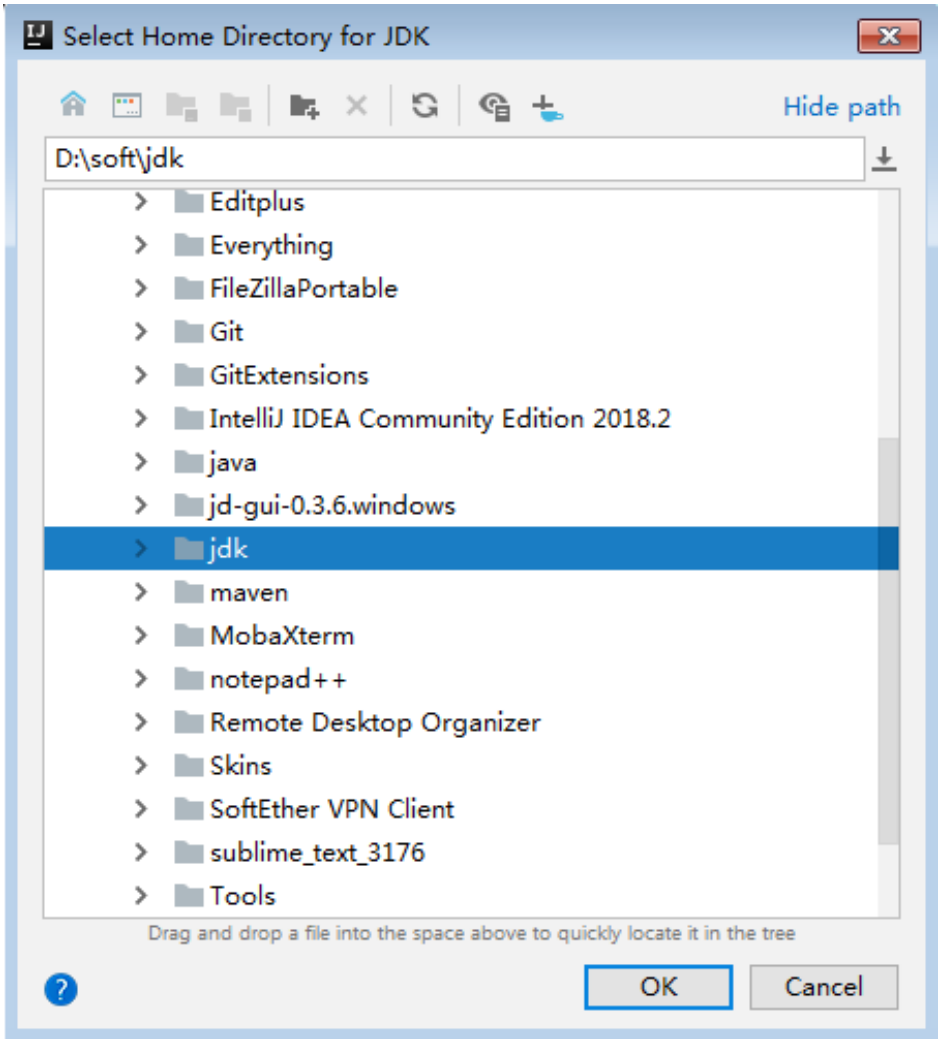
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 14-5 Project Structure for New Projects



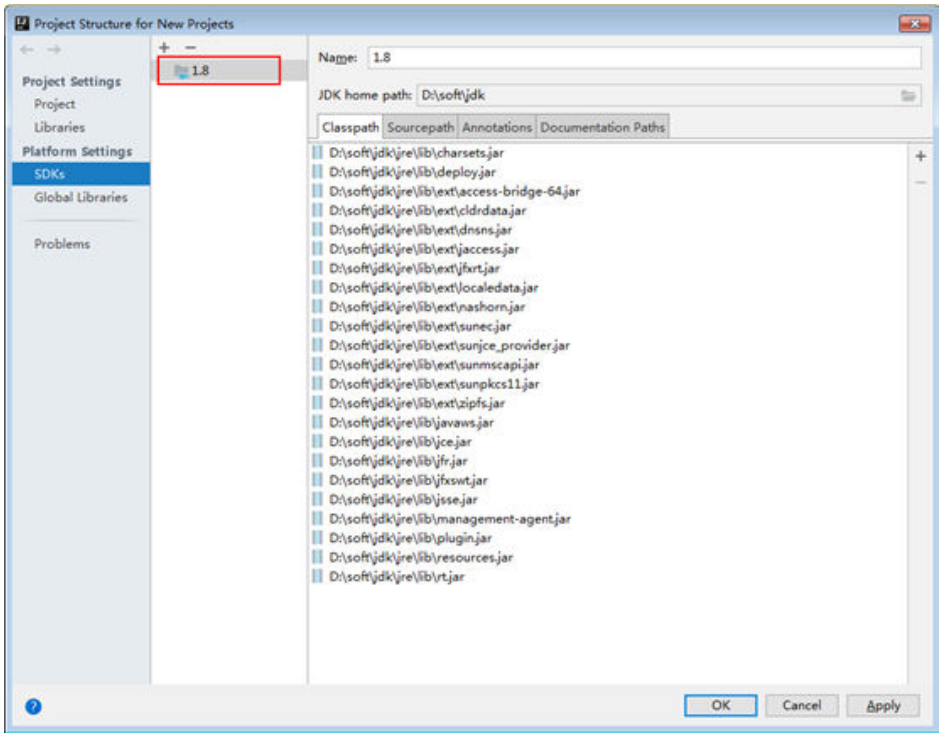
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 14-6 Select Home Directory for JDK



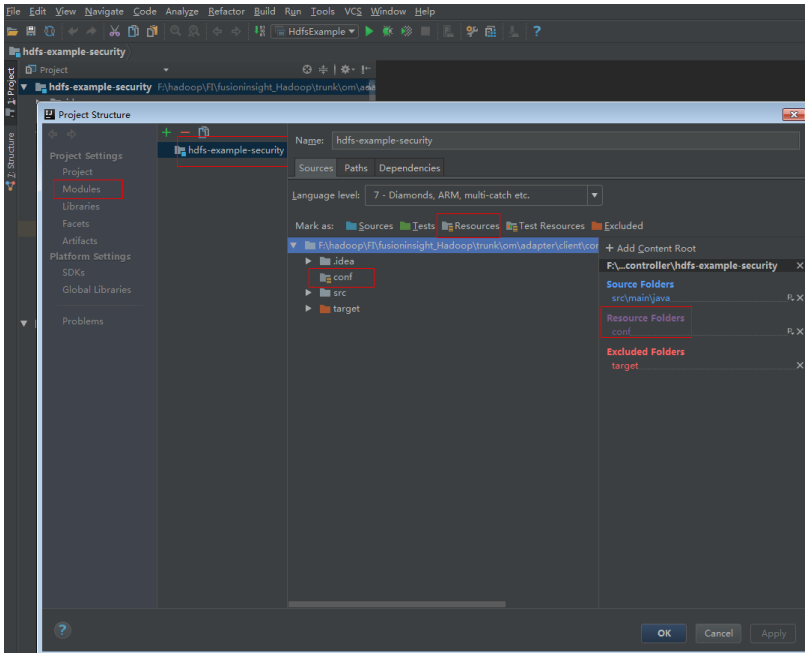
5. 完成JDK选择后，单击“OK”完成配置。

图 14-7 完成 JDK 配置



步骤4 将工程中的conf目录添加到资源路径。在Intellij IDEA的菜单栏中依次选择“File > Project Structure”。在弹出的会话框中选中当前工程，并选择“Resources > conf > OK”，从而完成资源目录的设置。如图“图14-8”所示。

图 14-8 设置工程资源目录



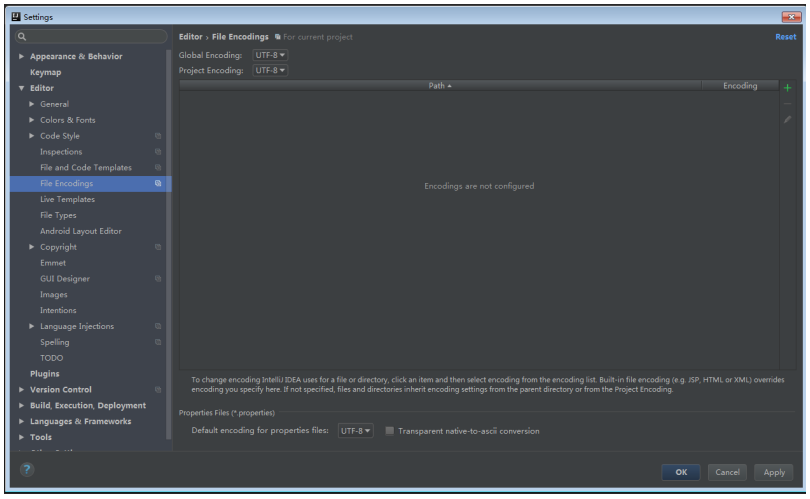
步骤5 将工程依赖的jar包添加到类路径。

如果通过开源镜像站方式获取的样例工程代码，在配置好Maven后（配置方式参考**配置华为开源镜像仓**）， 相关依赖jar包将自动下载，不需手动添加。

步骤6 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File> Settings”。
2. 在弹出的“Settings”窗口左边导航上选择“Editor > FileEncodings”，在“Global Encoding”和“Project Encodings”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图14-9所示。

图 14-9 设置 IntelliJ IDEA 的编码格式



----结束

14.5 开发 HDFS 应用

14.5.1 HDFS 样例程序开发思路

场景说明

HDFS的业务操作对象是文件，代码样例中所涉及的文件操作主要包括创建文件夹、写文件、追加文件内容、读文件和删除文件/文件夹；HDFS还有其他的业务处理，例如设置文件权限等，其他操作可以在掌握本代码样例之后，再扩展学习。

本代码样例讲解顺序为：

1. HDFS初始化
2. 创建目录
3. 写文件
4. 追加文件内容
5. 读文件
6. 删除文件
7. 删除目录
8. 多线程
9. 设置存储策略

10. Colocation

开发思路

根据前述场景说明进行功能分解，以“/user/hdfs-examples/test.txt”文件的读写删除等操作为例，说明HDFS文件的基本操作流程，可分为以下八部分：

1. 创建FileSystem对象：fSystem。
2. 调用fSystem的mkdir接口创建目录。
3. 调用fSystem的create接口创建FSDataOutputStream对象：out，使用out的write方法写入数据。
4. 调用fSystem的append接口创建FSDataOutputStream对象：out，使用out的write方法追加写入数据。
5. 调用fSystem的open接口创建FSDataInputStream对象：in，使用in的read方法读取文件。
6. 调用fSystem中的delete接口删除文件。
7. 调用fSystem中的delete接口删除文件夹。

14.5.2 初始化 HDFS

功能简介

在使用HDFS提供的API之前，需要先进行HDFS初始化操作。过程为：

1. 加载HDFS服务配置文件。
2. 实例化Filesystem。

配置文件介绍

登录HDFS时会使用到如表14-5所示的配置文件。这些文件均已导入到“hadoop-examples”工程的“conf”目录。

表 14-5 配置文件

文件名称	作用
core-site.xml	配置HDFS详细参数。
hdfs-site.xml	配置HDFS详细参数。

说明

“conf”目录下的“log4j.properties”文件客户根据自己的需要进行配置。

代码样例

如下是代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

在Linux客户端运行应用和在Windows环境下运行应用的初始化代码相同，代码样例如下所示。

```
//初始化
confLoad();

// 创建一个用例
HdfsExample hdfs_examples = new HdfsExample("/user/hdfs-examples", "test.txt");
/**
 *
 * 如果程序运行在Linux上，则需要core-site.xml、hdfs-site.xml的路径修改
 * 为在Linux下客户端文件的绝对路径
 *
 *
 */
private static void confLoad() throws IOException {
  conf = new Configuration();
  // conf file
  conf.addResource(new Path(PATH_TO_HDFS_SITE_XML));
  conf.addResource(new Path(PATH_TO_CORE_SITE_XML));
  // conf.addResource(new Path(PATH_TO_SMALL_SITE_XML));
}

/**
 *创建用例
 */
public HdfsExample(String path, String fileName) throws IOException  {
  this.DEST_PATH = path;
  this.FILE_NAME = fileName;
  instanceBuild();
}
private void instanceBuild() throws IOException {
  fSystem = FileSystem.get(conf);
}
```

说明

- （可选）运行此样例代码需要设置运行用户，若需运行Colocation相关操作的样例代码，则此用户需属supergroup用户组。设置运行用户有两种方式，添加环境变量HADOOP_USER_NAME或者修改代码。

  添加环境变量HADOOP_USER_NAME：参考**调测HDFS应用**章节。

  修改代码：在没有设置HADOOP_USER_NAME的场景下，直接修改代码中的USER。如下所示。

  ```
  System.setProperty("HADOOP_USER_NAME", USER);
  ```

14.5.3 创建 HDFS 目录

功能简介

创建目录过程为：

1. 调用FileSystem实例的exists方法查看该目录是否存在。
2. 如果存在，则直接返回。
3. 如果不存在，则调用FileSystem实例的mkdirs方法创建该目录。

代码样例

如下是写文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 创建目录
 *
 * @throws java.io.IOException
 */
private void mkdir() throws IOException {
```

```
  Path destPath = new Path(DEST_PATH);
  if (!createPath(destPath)) {
      LOG.error("failed to create destPath " + DEST_PATH);
  return;
  }

  LOG.info("success to create path " + DEST_PATH);
}

/**
 * create file path
 *
 * @param filePath
 * @return
 * @throws java.io.IOException
 */
private boolean createPath(final Path filePath) throws IOException {
    if (!fSystem.exists(filePath)) {
        fSystem.mkdirs(filePath);
    }
    return true;
}
```

14.5.4 创建 HDFS 文件并写入内容

功能简介

写文件过程为：

1. 使用FileSystem实例的create方法获取写文件的输出流。
2. 使用该数据流将内容写入到HDFS的指定文件中。

说明

在写完文件后，需关闭所申请资源。

代码样例

如下是写文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 创建文件，写文件
 *
 * @throws java.io.IOException
 * @throws com.huawei.bigdata.hdfs.examples.ParameterException
 */
private void write() throws IOException {
  final String content = "hi, I am bigdata. It is successful if you can see me.";
  FSDataOutputStream out = null;
  try {
      out = fSystem.create(new Path(DEST_PATH + File.separator + FILE_NAME));
      out.write(content.getBytes());
      out.hsync();
      LOG.info("success to write.");
  } finally {
    // make sure the stream is closed finally.
    IOUtils.closeStream(out);
  }
}
```

14.5.5 追加信息到 HDFS 指定文件

功能简介

追加文件内容，是指在HDFS的某个指定文件后面，追加指定的内容。过程为：

1. 使用FileSystem实例的append方法获取追加写入的输出流。
2. 使用该输出流将待追加内容添加到HDFS的指定文件后面。

说明

在完成后，需关闭所申请资源。

代码样例

如下是代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 追加文件内容
 *
 * @throws java.io.IOException
 */
private void append() throws IOException {
    final String content = "I append this content.";
    FSDataOutputStream out = null;
    try {
        out = fSystem.append(new Path(DEST_PATH + File.separator + FILE_NAME));
        out.write(content.getBytes());
        out.hsync();
        LOG.info("success to append.");
    } finally {
        // make sure the stream is closed finally.
        IOUtils.closeStream(out);
    }
}
```

14.5.6 读取 HDFS 指定文件内容

功能简介

获取HDFS上某个指定文件的内容。过程为：

1. 使用FileSystem实例的open方法获取读取文件的输入流。
2. 使用该输入流读取HDFS的指定文件的内容。

说明

在完成后，需关闭所申请资源。

代码样例

如下是读文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 读文件
 *
 * @throws java.io.IOException
```

```
 */
private void read() throws IOException {
    String strPath = DEST_PATH + File.separator + FILE_NAME;
    Path path = new Path(strPath);
    FSDataInputStream in = null;
    BufferedReader reader = null;
    StringBuffer strBuffer = new StringBuffer();
    try {
        in = fSystem.open(path);
        reader = new BufferedReader(new InputStreamReader(in));
        String sTempOneLine;
        // write file
        while ((sTempOneLine = reader.readLine()) != null) {
            strBuffer.append(sTempOneLine);
        }
        LOG.info("result is : " + strBuffer.toString());
        LOG.info("success to read.");
    } finally {
        // make sure the streams are closed finally.
        IOUtils.closeStream(reader);
        IOUtils.closeStream(in);
    }
}
```

14.5.7 删除 HDFS 指定文件

功能简介

删除HDFS上某个指定文件。

说明

被删除的文件会被直接删除，且无法恢复。所以，执行删除操作需谨慎。

代码样例

如下是删除文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
* 删除文件
*
* @throws java.io.IOException
*/
private void delete() throws IOException {
    Path beDeletedPath = new Path(DEST_PATH + File.separator + FILE_NAME);
    if (fSystem.delete(beDeletedPath, true)) {
        LOG.info("success to delete the file " + DEST_PATH + File.separator + FILE_NAME);
    } else {
        LOG.warn("failed to delete the file " + DEST_PATH + File.separator + FILE_NAME);
    }
}
```

14.5.8 删除 HDFS 指定目录

功能简介

删除HDFS上某个指定目录。

说明

被删除的目录会被直接删除，且无法恢复。所以，执行删除操作需谨慎。

代码样例

如下是删除文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
/**
 * 删除目录
 *
 * @throws java.io.IOException
 */
private void rmdir() throws IOException {
    Path destPath = new Path(DEST_PATH);
    if (!deletePath(destPath)) {
        LOG.error("failed to delete destPath " + DEST_PATH);
        return;
    }
    LOG.info("success to delete path " + DEST_PATH);
}

/**
 *
 * @param filePath
 * @return
 * @throws java.io.IOException
 */
private boolean deletePath(final Path filePath) throws IOException {
    if (!fSystem.exists(filePath)) {
        return false;
    }
    // fSystem.delete(filePath, true);
    return fSystem.delete(filePath, true);
}
```

14.5.9 创建 HDFS 多线程任务

功能简介

建立多线程任务，同时启动多个实例执行文件操作。

代码样例

如下是删除文件的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

```
 // 业务示例2：多线程
final int THREAD_COUNT = 2;
for (int threadNum = 0; threadNum < THREAD_COUNT; threadNum++) {
    HdfsExampleThread example_thread = new HdfsExampleThread("hdfs_example_" + threadNum);
    example_thread.start();
}


class HdfsExampleThread extends Thread {
    private final static Log LOG = LogFactory.getLog(HdfsExampleThread.class.getName());
    /**
     *
     * @param threadName
     */
    public HdfsExampleThread(String threadName) {
        super(threadName);
    }
    public void run() {
        HdfsExample example;
        try {
```

```
            example = new HdfsExample("/user/hdfs-examples/" + getName(), "test.txt");
            example.test();
        } catch (IOException e) {
            LOG.error(e);
        }
    }
}
```

example.test()方法即为对文件的操作，代码如下：

```
/**
 * HDFS 操作实例
 *
 * @throws IOException
 * @throws ParameterException
 *
 * @throws Exception
 *
 */
public void test() throws IOException {
    // 创建目录
    mkdir();

    // 写文件
    write();

    // 追加文件内容
    append();

    // 读文件
    read();

    // 删除文件
    delete();

    // 删除目录
    rmdir();
}
```

14.5.10 配置 HDFS 存储策略

功能简介

为HDFS上某个文件或文件夹指定存储策略。

代码样例

1. **登录FusionInsight Manager页面**，选择“集群 > *待操作集群的名称* > 服务 > HDFS > 配置 > 全部配置”。
2. 搜索并查看“dfs.storage.policy.enabled”的参数值是否为“true”，如果不是，修改为“true”，并单击“保存”，重启HDFS。
3. 查看代码。

   如下是代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsExample类。

   ```
   /**
   * 设置存储策略
   * @param policyName
   * 策略名称能够被接受:
   * <li>HOT
   * <li>WARM
   * <li>COLD
   * <li>LAZY_PERSIST
   * <li>ALL_SSD
   ```

```
   * <li>ONE_SSD
   * @throws IOException
   */
  private void setStoragePolicy(String policyName) throws IOException {
    if (fSystem instanceof DistributedFileSystem) {
        DistributedFileSystem dfs = (DistributedFileSystem) fSystem;
        Path destPath = new Path(DEST_PATH);
        Boolean flag = false;
        mkdir();
        BlockStoragePolicySpi[] storage = dfs.getStoragePolicies();
        for (BlockStoragePolicySpi bs : storage) {
            if (bs.getName().equals(policyName)) {
                flag = true;
            }
            LOG.info("StoragePolicy:" + bs.getName());
        }
        if (!flag) {
            policyName = storage[0].getName();
        }
        dfs.setStoragePolicy(destPath, policyName);
        LOG.info("success to set Storage Policy path " + DEST_PATH);
        rmdir();
    } else {
        LOG.info("SmallFile not support to set Storage Policy !!!");
    }
  }
```

14.5.11 配置 HDFS 同分布策略（Colocation）

功能简介

同分布（Colocation）功能是将存在关联关系的数据或可能要进行关联操作的数据存储在相同的存储节点上。HDFS文件同分布的特性，将那些需进行关联操作的文件存放在相同数据节点上，在进行关联操作计算时避免了到别的数据节点上获取数据，大大降低网络带宽的占用。

在使用Colocation功能之前，建议用户对Colocation的内部机制有一定了解，包括：

- **Colocation分配节点原理**
- **扩容与Colocation分配**
- **Colocation与数据节点容量**

- **Colocation分配节点原理**

  Colocation为locator分配数据节点的时候，locator的分配算法会根据已分配的情况，进行均衡的分配数据节点。

  说明

  locator分配算法的原理是，查询目前存在的所有locators，读取所有locators所分配的数据节点，并记录其使用次数。根据使用次数，对数据节点进行排序，使用次数少的排在前面，优先选择排在前面的节点。每次选择一个节点后，计数加1，并重新排序，选择后续的节点。

- **扩容与Colocation分配**

  集群扩容之后，为了平衡地使用所有的数据节点，使新的数据节点的分配频率与旧的数据节点趋于一致，有如下两种策略可以选择，如**表14-6**所示。

表 14-6 分配策略

编号	策略	说明
1	删除旧的locators，为集群中所有数据节点重新创建locators。	1. 在未扩容之前分配的locators，平衡的使用了所有数据节点。当扩容后，新加入的数据节点并未分配到已经创建的locators中，所以使用Colocation来存储数据的时候，只会往旧的数据节点存储数据。 2. 由于locators与特定数据节点相关，所以当集群进行扩容的时候，就需要对Colocation的locators分配进行重新规划。
2	创建一批新的locators，并重新规划数据存放方式。	旧的locators使用的是旧的数据节点，而新创建的locators偏重使用新的数据节点，所以需要根据实际业务对数据的使用需求，重新规划locators的使用。

说明

一般的，建议用户在进行集群扩容之后采用策略一来重新分配locators，可以避免数据偏重使用新的数据节点。

- **Colocation与数据节点容量**

  由于使用Colocation进行存储数据的时候，会固定存储在指定的locator所对应的数据节点上面，所以如果不对locator进行规划，会造成数据节点容量不均衡。下面总结了保证数据节点容量均衡的两个主要的使用原则，如表14-7所示。

表 14-7 使用原则

编号	使用原则	说明
1	所有的数据节点在locators中出现的频率一样。	如何保证频率一样：假如数据节点有N个，则创建locators的数量应为N的整数倍（N个、2N个……）。
2	对于所有locators的使用需要进行合理的数据存放规划，让数据均匀的分布在这些locators中。	无

HDFS的二次开发过程中，可以获取DFSColocationAdmin和DFSColocationClient实例，进行从location创建group、删除group、写文件和删除文件的操作。

说明

- 使用Colocation功能，用户指定了DataNode，会造成某些节点上数据量很大。数据倾斜严重，导致HDFS写任务失败。
- 由于数据倾斜，导致MapReduce只会在某几个节点访问，造成这些节点上负载很大，而其他节点闲置。
- 针对单个应用程序任务，只能使用一次DFSColocationAdmin和DFSColocationClient实例。如果每次对文件系统操作都获取此实例，会创建过多HDFS链接，消耗HDFS资源。
- Colocation提供了文件同分布的功能，执行集群balancer或mover操作时，会移动数据块，使Colocation功能失效。因此，使用Colocation功能时，建议将HDFS配置项dfs.datanode.block-pinning.enabled设置为true，此时执行集群Balancer或Mover操作时，使用Colocation写入的文件将不会被移动，从而保证了文件同分布。

代码样例

完整样例代码可参考com.huawei.bigdata.hdfs.examples.ColocationExample。

说明

- 在运行Colocation工程时，需要设置运行用户，此用户需绑定supergroup用户组。
- 在运行Colocation工程时，HDFS的配置项fs.defaultFS不能配置为viewfs://ClusterX。

1. 初始化

   使用Colocation前需要设置运行用户。

```
private static void init() throws IOException {
    // 设置用户，若用户没有设置HADOOP_USER_NAME，则使用USER
    if (System.getenv("HADOOP_USER_NAME") == null &&
System.getProperty("HADOOP_USER_NAME") == null) {
        System.setProperty("HADOOP_USER_NAME", USER);
    }
}
```

2. 获取实例

   样例：Colocation的操作使用DFSColocationAdmin和DFSColocationClient实例，在进行创建group等操作前需获取实例。

```
 dfsAdmin = new DFSColocationAdmin(conf);
 dfs = new DFSColocationClient();
 dfs.initialize(URI.create(conf.get("fs.defaultFS")), conf);
```

3. 创建group

   样例：创建一个gid01组，组中包含3个locator。

```
/**
 * 创建group
 *
 * @throws java.io.IOException
 */
private static void createGroup() throws IOException {
    dfsAdmin.createColocationGroup(COLOCATION_GROUP_GROUP01,
            Arrays.asList(new String[] { "lid01", "lid02", "lid03" }));
}
```

4. 写文件，写文件前必须创建对应的group

   样例：写入testfile.txt文件。

```
 /**
 * 创建并写入文件
 *
 * @throws java.io.IOException
 */
private static void put() throws IOException {
    FSDataOutputStream out = dfs.create(new Path(TESTFILE_TXT), true,
```

```
COLOCATION_GROUP_GROUP01, "lid01");
    // 代写入HDFS的数据
    byte[] readBuf = "Hello World".getBytes("UTF-8");
    out.write(readBuf, 0, readBuf.length);
    out.close();
}
```

5. 删除文件

   样例：删除testfile.txt文件。

```
/**
* 删除文件
*
* @throws java.io.IOException
*/
@SuppressWarnings("deprecation")
private static void delete() throws IOException {
    dfs.delete(new Path(TESTFILE_TXT));
}
```

6. 删除group

   样例：删除gid01。

```
/**
* 删除group
*
* @throws java.io.IOException
*/
private static void deleteGroup() throws IOException {
    dfsAdmin.deleteColocationGroup(COLOCATION_GROUP_GROUP01);
}
```

14.6 调测 HDFS 应用

14.6.1 在本地 Windows 中调测 HDFS 程序

操作场景

在代码完成开发后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

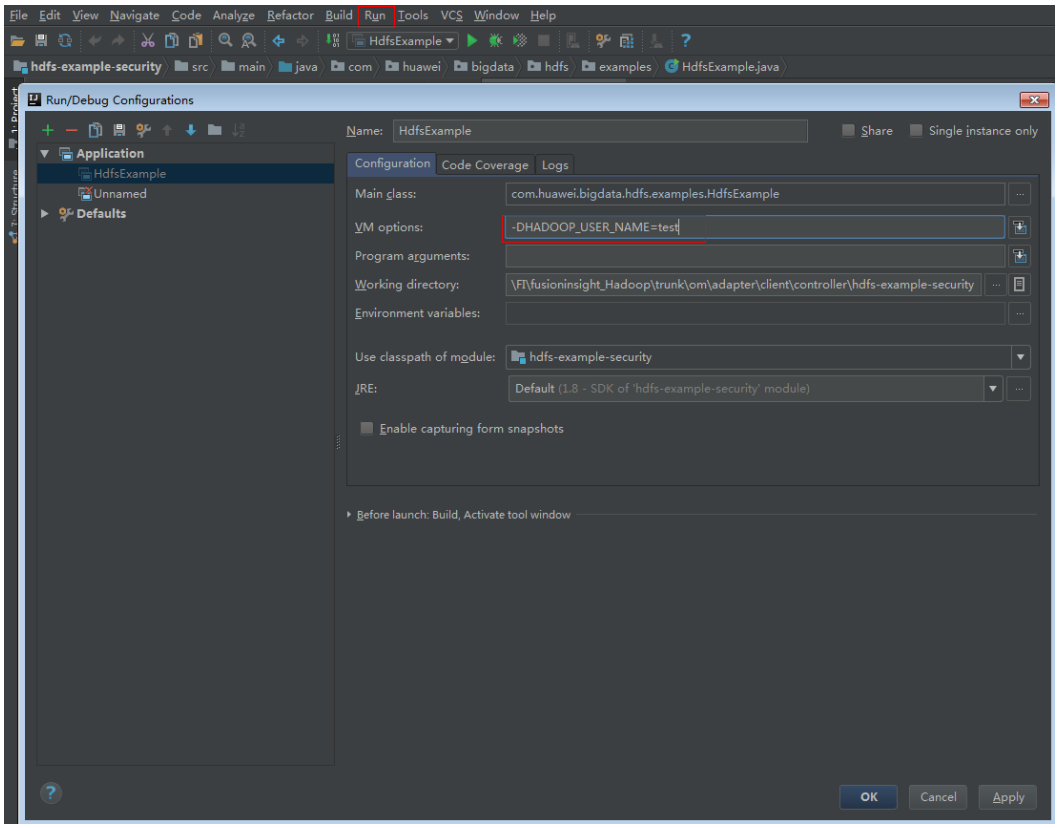
HDFS应用程序运行完成后，可直接通过运行结果查看应用程序运行情况，也可以通过HDFS日志获取应用运行情况。

在本地 Windows 中调测 HDFS 程序

步骤1 （可选）在开发环境中（例如IntelliJ IDEA中），运行此样例代码需要设置运行用户，设置运行用户有两种方式，添加环境变量HADOOP_USER_NAME或者修改代码。设置环境变量方法具体如下：

选中需要运行的样例程序HdfsExample.java或者ColocationExample.java，右键工程，选择“Run Configurations”，在对话框中选择“JavaApplication > HdfsExample”进行运行参数设置。在IntelliJ IDEA的菜单栏依次选择“Run > Edit Configurations”，在弹出的会话框中设置运行用户。

```
-DHADOOP_USER_NAME=test
```



说明

用户可向管理员咨询运行用户。test在这里只是举例，若需运行Colocation相关操作的样例代码，则此用户需属于supergroup用户组。

步骤2 若已按照步骤1设置环境变量，则直接单击Run，运行应用工程。否则分别选中以下两个工程运行程序：

- 选中HdfsExample.java，右键工程，选择“Run 'HdfsExample.main()'"”运行应用工程。
- 选中ColocationExample.java，右键工程，选择“Run 'ColocationExample.main()'”运行应用工程。

说明

- 在HDFS任务运行过程中禁止重启HDFS服务，否则可能会导致任务失败。
- 在运行Colocation工程时，HDFS的配置项fs.defaultFS不能配置为viewfs://ClusterX。

----结束

查看调测结果

- **查看运行结果获取应用运行情况**
  - HdfsExample Windows样例程序运行结果如下所示。

```
1654 [main] WARN  org.apache.hadoop.hdfs.shortcircuit.DomainSocketFactory  - The short-circuit local reads feature cannot be used because UNIX Domain sockets are not available on Windows.
2013 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to create path /user/hdfs-examples
2137 [main] WARN  org.apache.hadoop.util.NativeCodeLoader  - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
```



```
2590 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to write.
3245 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to append.
4447 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I am
bigdata. It is successful if you can see me.I append this content.
4447 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to read.
4509 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete the file /
user/hdfs-examples\test.txt
4618 [main] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to delete path /
user/hdfs-examples
4743 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
create path /user/hdfs-examples/hdfs_example_1
4743 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
create path /user/hdfs-examples/hdfs_example_0
5087 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
write.
5087 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
write.
6507 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
append.
6553 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
append.
7505 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I
am bigdata. It is successful if you can see me.I append this content.
7505 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
read.
7568 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
delete the file /user/hdfs-examples/hdfs_example_1\test.txt
7583 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - result is : hi, I
am bigdata. It is successful if you can see me.I append this content.
7583 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
read.
7630 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
delete the file /user/hdfs-examples/hdfs_example_0\test.txt
7677 [hdfs_example_1] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
delete path /user/hdfs-examples/hdfs_example_1
7739 [hdfs_example_0] INFO  com.huawei.bigdata.hdfs.examples.HdfsExample  - success to
delete path /user/hdfs-examples/hdfs_example_0
```

说明

在Windows环境运行样例代码时会出现下面的异常，但是不影响业务：

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

- ColocationExample Windows样例程序运行结果如下所示。

```
1623 [main] WARN  org.apache.hadoop.hdfs.shortcircuit.DomainSocketFactory  - The short-
circuit local reads feature cannot be used because UNIX Domain sockets are not available on
Windows.
1670 [main] INFO  org.apache.zookeeper.ZooKeeper  - Client
environment:zookeeper.version=***, built on 10/19/2017 04:21 GMT
1670 [main] INFO  org.apache.zookeeper.ZooKeeper  - Client
environment:host.name=siay7user1.china.huawei.com
1670 [main] INFO  org.apache.zookeeper.ZooKeeper  - Client environment:java.version=***
1670 [main] INFO  org.apache.zookeeper.ZooKeeper  - Client environment:java.vendor=Oracle
Corporation
1670 [main] INFO  org.apache.zookeeper.ZooKeeper  - Client
environment:java.home=D:\Program Files\Java\jre1.8.0_131
......
Create Group has finished.
Put file is running...
5930 [main] WARN  org.apache.hadoop.util.NativeCodeLoader  - Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Put file has finished.
Delete file is running...
Delete file has finished.
Delete Group is running...
Delete Group has finished.
6866 [main] INFO  org.apache.zookeeper.ZooKeeper  - Session: 0x13000074b7e464b7 closed
6866 [main-EventThread] INFO  org.apache.zookeeper.ClientCnxn  - EventThread shut down for
```



```
session: 0x13000074b7e464b7
6928 [main-EventThread] INFO  org.apache.zookeeper.ClientCnxn  - EventThread shut down for
session: 0x14000073f13b657b
6928 [main] INFO  org.apache.zookeeper.ZooKeeper  - Session: 0x14000073f13b657b closed
```

- **查看HDFS日志获取应用运行情况**

  可以查看HDFS的NameNode日志了解应用运行情况，并根据日志信息调整应用程序。

14.6.2 在 Linux 环境中调测 HDFS 应用

操作场景

HDFS应用程序支持在Linux环境中运行。在程序代码完成开发后，可以上传Jar包至准备好的Linux环境中运行。

HDFS应用程序运行完成后，可直接通过运行结果查看应用程序运行情况，也可以通过HDFS日志获取应用运行情况。

前提条件

- 已安装客户端时：
  - 已安装HDFS客户端。
  - 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的hosts文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。
- 未安装客户端时：
  - Linux环境已安装JDK，版本号需要和IDEA导出Jar包使用的JDK版本一致。
  - 当Linux环境所在主机不是集群中的节点时，需要在Linux环境所在节点的hosts文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

已安装客户端时编译并运行程序

步骤1 进入样例工程本地根目录，在Windows命令提示符窗口中执行下面命令进行打包。

mvn -s "*{maven_setting_path}*" clean package

说明

- 上述打包命令中的{maven_setting_path}为本地Maven的“settings.xml”文件路径。
- 打包成功之后，在工程根目录的target子目录下获取打好的jar包，例如“*HDFSTest-XXX.jar*”，jar包名称以实际打包结果为准。

步骤2 将导出的Jar包上传至Linux客户端运行环境的任意目录下，例如“/opt/client”。

步骤3 配置环境变量：

cd /opt/client

source bigdata_env

步骤4 运行此样例代码需要设置运行用户，设置运行用户有两种方式，添加环境变量HADOOP_USER_NAME或者修改代码设置运行用户。若在没有修改代码的场景下，执行以下语句添加环境变量：

```
export HADOOP_USER_NAME=test
```

说明

用户可向管理员咨询运行用户。test在这里只是举例，若需运行Colocation相关操作的样例代码，则此用户需属supergroup用户组。

步骤5 执行如下命令，运行Jar包。

hadoop jar *HDFSTest-XXX.jar com.huawei.bigdata.hdfs.examples.HdfsExample*

hadoop jar *HDFSTest-XXX.jar com.huawei.bigdata.hdfs.examples.ColocationExample*

说明

在运行**com.huawei.bigdata.hdfs.examples.ColocationExample**时，HDFS的配置项“fs.defaultFS”不能配置为“viewfs://ClusterX”。

----结束

未安装客户端时编译并运行程序

步骤1 进入工程本地根目录，在Windows命令提示符窗口中执行下面命令进行打包。

mvn -s "*{maven_setting_path}***" clean package**

说明

- 上述打包命令中的{maven_setting_path}为本地Maven的“settings.xml”文件路径。
- 打包成功之后，在工程根目录的target子目录下获取打好的jar包。

步骤2 将导出的Jar包上传至Linux运行环境的任意目录下，例如“/optclient”。

步骤3 将工程中的“lib”文件夹和“conf”文件夹上传至和Jar包相同的Linux运行环境目录下，例如“/opt/client”（其中“lib”目录汇总包含了工程中依赖的所有的Jar包，“conf”目录包含运行jar包所需的集群相关配置文件，请参考**准备运行环境**）。

步骤4 运行此样例代码需要设置运行用户，设置运行用户有两种方式，添加环境变量HADOOP_USER_NAME或者修改代码设置运行用户。若在没有修改代码的场景下，执行以下语句添加环境变量：

```
export HADOOP_USER_NAME=test
```

说明

用户可向管理员咨询运行用户。test在这里只是举例，若需运行Colocation相关操作的样例代码，则此用户需属supergroup用户组。

步骤5 执行如下命令运行Jar包。

java -cp *HDFSTest-XXX.jar*:conf/:lib/* *com.huawei.bigdata.hdfs.examples.HdfsExample*

java -cp *HDFSTest-XXX.jar*:conf/:lib/* *com.huawei.bigdata.hdfs.examples.ColocationExample*

说明

在运行*com.huawei.bigdata.hdfs.examples.ColocationExample*时，HDFS的配置项“fs.defaultFS”不能配置为“viewfs://ClusterX”。

----结束

查看调测结果

- **查看运行结果获取应用运行情况**
  - HdfsExample Linux样例程序运行结果如下所示。

```
[root@192-168-32-144 client]#hadoop jar HDFSTest-XXX.jar
com.huawei.bigdata.hdfs.examples.HdfsExample
WARNING: Use "yarn jar" to launch YARN applications.
17/10/26 19:11:44 INFO examples.HdfsExample: success to create path /user/hdfs-examples
17/10/26 19:11:44 INFO examples.HdfsExample: success to write.
17/10/26 19:11:45 INFO examples.HdfsExample: success to append.
17/10/26 19:11:45 INFO examples.HdfsExample: result is : hi, I am bigdata. It is successful if you
can see me.I append this content.
17/10/26 19:11:45 INFO examples.HdfsExample: success to read.
17/10/26 19:11:45 INFO examples.HdfsExample: success to delete the file /user/hdfs-examples/
test.txt
17/10/26 19:11:45 INFO examples.HdfsExample: success to delete path /user/hdfs-examples
17/10/26 19:11:45 INFO examples.HdfsExample: success to create path /user/hdfs-examples/
hdfs_example_1
17/10/26 19:11:45 INFO examples.HdfsExample: success to create path /user/hdfs-examples/
hdfs_example_0
17/10/26 19:11:45 INFO examples.HdfsExample: success to write.
17/10/26 19:11:45 INFO examples.HdfsExample: success to write.
17/10/26 19:11:46 INFO examples.HdfsExample: success to append.
17/10/26 19:11:46 INFO examples.HdfsExample: result is : hi, I am bigdata. It is successful if you
can see me.I append this content.
17/10/26 19:11:46 INFO examples.HdfsExample: success to read.
17/10/26 19:11:46 INFO examples.HdfsExample: success to delete the file /user/hdfs-examples/
hdfs_example_1/test.txt
17/10/26 19:11:46 INFO examples.HdfsExample: success to delete path /user/hdfs-examples/
hdfs_example_1
17/10/26 19:11:46 INFO examples.HdfsExample: success to append.
17/10/26 19:11:46 INFO examples.HdfsExample: result is : hi, I am bigdata. It is successful if you
can see me.I append this content.
17/10/26 19:11:46 INFO examples.HdfsExample: success to read.
17/10/26 19:11:46 INFO examples.HdfsExample: success to delete the file /user/hdfs-examples/
hdfs_example_0/test.txt
17/10/26 19:11:46 INFO examples.HdfsExample: success to delete path /user/hdfs-examples/
hdfs_example_0
```

  - ColocationExample Linux样例程序运行结果如下所示。

```
[root@192-168-32-144 client]#hadoop jar HDFSTest-XXX.jar
com.huawei.bigdata.hdfs.examples.ColocationExample
WARNING: Use "yarn jar" to launch YARN applications.
17/10/26 19:12:38 INFO zookeeper.ZooKeeper: Client environment:zookeeper.version=xxx, built
on 10/19/2017 04:21 GMT
17/10/26 19:12:38 INFO zookeeper.ZooKeeper: Client environment:host.name=192-168-32-144
17/10/26 19:12:38 INFO zookeeper.ZooKeeper: Client environment:java.version=1.8.0_144
17/10/26 19:12:38 INFO zookeeper.ZooKeeper: Client environment:java.vendor=Oracle
Corporation
17/10/26 19:12:38 INFO zookeeper.ZooKeeper: Client environment:java.home=/opt/client/JDK/
jdk1.8.0_144/jre
.....
Create Group has finished.
Put file is running...
Put file has finished.
Delete file is running...
Delete file has finished.
Delete Group is running...
Delete Group has finished.
17/10/26 19:12:39 INFO zookeeper.ZooKeeper: Session: 0x13000074b7e4687f closed
17/10/26 19:12:39 INFO zookeeper.ClientCnxn: EventThread shut down for session:
0x13000074b7e4687f
17/10/26 19:12:39 INFO zookeeper.ZooKeeper: Session: 0x12000059699f69e1 closed
17/10/26 19:12:39 INFO zookeeper.ClientCnxn: EventThread shut down for session:
0x12000059699f69e1
```

- **查看HDFS日志获取应用运行情况**

可以查看HDFS的namenode日志了解应用运行情况，并根据日志信息调整应用程序。

14.7 HDFS 应用开发常见问题

14.7.1 HDFS 常用 API 介绍

14.7.1.1 HDFS Java API 接口介绍

HDFS完整和详细的接口可以直接参考官方网站上的描述：

http://hadoop.apache.org/docs/r3.1.1/api/index.html

HDFS 常用接口

HDFS常用的Java类有以下几个：

- FileSystem：是客户端应用的核心类。常用接口参见**表14-8**。
- FileStatus：记录文件和目录的状态信息。常用接口参见**表14-9**。
- DFSColocationAdmin：管理colocation组信息的接口。常用接口参见**表14-10**。
- DFSColocationClient：操作colocation文件的接口。常用接口参见**表14-11**。

  说明

  - 系统中不保留文件与LocatorId的映射关系，只保留节点与LocatorId的映射关系。当文件使用Colocation接口创建时，系统会将文件创建在LocatorId所对应的节点上。文件创建和写入要求使用Colocation相关接口。
  - 文件写入完毕后，后续对该文件的相关操作不限制使用Colocation接口，也可以使用开源接口进行操作。
  - DFSColocationClient类继承于开源的DistributedFileSystem类，包含其常用接口。建议使用DFSColocationClient进行Colocation相关文件操作。

表 14-8 类 FileSystem 常用接口说明

接口	说明
public static FileSystem get(Configuration conf)	Hadoop类库中最终面向用户提供的接口类是FileSystem，该类是个抽象类，只能通过该类的get方法得到具体类。get方法存在几个重载版本，常用的是这个。
public FSDataOutputStream create(Path f)	通过该接口可在HDFS上创建文件，其中f为文件的完整路径。
public void copyFromLocalFile(Path src, Path dst)	通过该接口可将本地文件上传到HDFS的指定位置上，其中src和dst均为文件的完整路径。
public boolean mkdirs(Path f)	通过该接口可在HDFS上创建文件夹，其中f为文件夹的完整路径。

接口	说明
public abstract boolean rename(Path src, Path dst)	通过该接口可为指定的HDFS文件重命名，其中src和dst均为文件的完整路径。
public abstract boolean delete(Path f, boolean recursive)	通过该接口可删除指定的HDFS文件，其中f为需要删除文件的完整路径，recuresive用来确定是否进行递归删除。
public boolean exists(Path f)	通过该接口可查看指定HDFS文件是否存在，其中f为文件的完整路径。
public FileStatus getFileStatus(Path f)	通过该接口可以获取文件或目录的FileStatus对象，该对象记录着该文件或目录的各种状态信息，其中包括修改时间、文件目录等等。
public BlockLocation[] getFileBlockLocations(FileStatus file, long start, long len)	通过该接口可查找指定文件在HDFS集群上块的位置，其中file为文件的完整路径，start和len来标识查找文件的块的范围。
public FSDataInputStream open(Path f)	通过该接口可以打开HDFS上指定文件的输出流，并可通过FSDataInputStream类提供接口进行文件的读出，其中f为文件的完整路径。
public FSDataOutputStream create(Path f, boolean overwrite)	通过该接口可以在HDFS上创建指定文件的输入流，并可通过FSDataOutputStream类提供的接口进行文件的写入，其中f为文件的完整路径，overwrite为true时表示如果文件已经存在，则重写文件；如果为false，当文件已经存在时，则抛出异常。
public FSDataOutputStream append(Path f)	通过该接口可以打开HDFS上已经存在的指定文件的输入流，并可通过FSDataOutputStream类提供的接口进行文件的写入，其中f为文件的完整路径。

表 14-9 类 FileStatus 常用接口说明

接口	说明
public long getModificationTime()	通过该接口可查看指定HDFS文件的修改时间。
public Path getPath()	通过该接口可查看指定HDFS中某个目录下所有文件。

表 14-10 类 DFSColocationAdmin 常用接口说明

接口	说明
public Map<String, List<DatanodeInfo>> createColocationGroup(String groupId,String file)	根据文件file中的locatorIds信息，创建group。file为文件路径。
public Map<String, List<DatanodeInfo>> createColocationGroup(String groupId,List<String> locators)	使用内存中List的locatorIds信息，创建group。
public void deleteColocationGroup(String groupId)	删除group。
public List<String> listColocationGroups()	返回colocation所有组信息，返回的组Id数组按创建时间排序。
public List<DatanodeInfo> getNodesForLocator(String groupId, String locatorId)	获取该locator中所有节点列表。

表 14-11 类 DFSColocationClient 常用接口说明

接口	说明
public FSDataOutputStream create(Path f, boolean overwrite, String groupId,String locatorId)	用colocation模式，创建一个FSDataOutputStream，从而允许用户在f路径写文件。 f为HDFS路径。 overwrite表示如果文件已存在是否允许覆盖。 用户指定文件所属的groupId和locatorId必须已经存在。

接口	说明
public FSDataOutputStream create(final Path f, final FsPermission permission, final EnumSet<CreateFlag> cflags, final int bufferSize, final short replication, final long blockSize, final Progressable progress, final ChecksumOpt checksumOpt, final String groupId, final String locatorId)	功能与FSDataOutputStream create(Path f, boolean overwrite, String groupId,String locatorId)相同，只是允许用户自定义checksum选项。
public void close()	使用完毕后关闭连接。

表 14-12 HDFS 客户端 WebHdfsFileSystem 接口说明

接口	说明
public RemoteIterator<FileStatus> listStatusIterator(final Path)	该API有助于通过使用远程迭代的多个请求获取子文件和文件夹信息，从而避免在获取大量子文件和文件夹信息时，用户界面变慢。

基于 API 的 Glob 路径模式以获取 LocatedFileStatus 和从 FileStatus 打开文件

在DistributedFileSystem中添加了以下API，以获取具有块位置的FileStatus，并从FileStatus对象打开文件。这些API将减少从客户端到Namenode的RPC调用的数量。

表 14-13 FileSystem API 接口说明

Interface接口	Description说明
public LocatedFileStatus[] globLocatedStatus(Path, PathFilter, boolean) throws IOException	返回一个LocatedFileStatus对象数组，其对应文件路径符合路径过滤规则。
public FSDataInputStream open(FileStatus stat) throws IOException	如果stat对象是LocatedFileStatusHdfs的实例，该实例已具有位置信息，则直接创建InputStream而不联系Namenode。

14.7.1.2 HDFS C API 接口介绍

功能简介

C语言应用开发代码样例中所涉及的文件操作主要包括创建文件、读写文件、追加文件、删除文件。完整和详细的接口请直接参考官网上的描述以了解其使用方法：

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/LibHdfs.html

代码样例

下面代码片段仅为演示，具体代码请参见样例代码解压目录中HDFS的C样例代码“hdfs-c-example/hdfs_test.c”文件。

1. 设置HDFS NameNode参数，建立HDFS文件系统连接。

```
hdfsFS fs = hdfsConnect("default", 0);
fprintf(stderr, "hdfsConnect- SUCCESS!\n");
```

2. 创建HDFS目录。

```
const char* dir = "/tmp/nativeTest";
int exitCode = hdfsCreateDirectory(fs, dir);
if( exitCode == -1 ){
    fprintf(stderr, "Failed to create directory %s \n", dir );
    exit(-1);
}
fprintf(stderr, "hdfsCreateDirectory- SUCCESS! : %s\n", dir);
```

3. 写文件。

```
const char* file = "/tmp/nativeTest/testfile.txt";
hdfsFile writeFile = openFile(fs, (char*)file, O_WRONLY |O_CREAT, 0, 0, 0);
fprintf(stderr, "hdfsOpenFile- SUCCESS! for write : %s\n", file);

if(!hdfsFileIsOpenForWrite(writeFile)){
    fprintf(stderr, "Failed to open %s for writing.\n", file);
    exit(-1);
}

char* buffer = "Hadoop HDFS Native file write!";

hdfsWrite(fs, writeFile, (void*)buffer, strlen(buffer)+1);
fprintf(stderr, "hdfsWrite- SUCCESS! : %s\n", file);

printf("Flushing file data ....\n");
if (hdfsFlush(fs, writeFile)) {
    fprintf(stderr, "Failed to 'flush' %s\n", file);
    exit(-1);
}
hdfsCloseFile(fs, writeFile);
fprintf(stderr, "hdfsCloseFile- SUCCESS! : %s\n", file);
```

4. 读文件。

```
hdfsFile readFile = openFile(fs, (char*)file, O_RDONLY, 100, 0, 0);
fprintf(stderr, "hdfsOpenFile- SUCCESS! for read : %s\n", file);

if(!hdfsFileIsOpenForRead(readFile)){
    fprintf(stderr, "Failed to open %s for reading.\n", file);
    exit(-1);
}

buffer = (char *) malloc(100);
tSize num_read = hdfsRead(fs, readFile, (void*)buffer, 100);
fprintf(stderr, "hdfsRead- SUCCESS!, Byte read : %d, File contant : %s \n", num_read ,buffer);
hdfsCloseFile(fs, readFile);
```

5. 指定位置开始读文件。

```
buffer = (char *) malloc(100);
readFile = openFile(fs, file, O_RDONLY, 100, 0, 0);
if (hdfsSeek(fs, readFile, 10)) {
     fprintf(stderr, "Failed to 'seek' %s\n", file);
     exit(-1);
}
num_read = hdfsRead(fs, readFile, (void*)buffer, 100);
fprintf(stderr, "hdfsSeek- SUCCESS!, Byte read : %d, File seek contant : %s \n", num_read ,buffer);
hdfsCloseFile(fs, readFile);
```

6. 复制文件。

```
const char* destfile = "/tmp/nativeTest/testfile1.txt";
if (hdfsCopy(fs, file, fs, destfile)) {
    fprintf(stderr, "File copy failed, src : %s, des : %s \n", file, destfile);
    exit(-1);
}
fprintf(stderr, "hdfsCopy- SUCCESS!, File copied, src : %s, des : %s \n", file, destfile);
```

7. 移动文件。

```
const char* mvfile = "/tmp/nativeTest/testfile2.txt";
if (hdfsMove(fs, destfile, fs, mvfile )) {
    fprintf(stderr, "File move failed, src : %s, des : %s \n", destfile , mvfile);
    exit(-1);
}
fprintf(stderr, "hdfsMove- SUCCESS!, File moved, src : %s, des : %s \n", destfile , mvfile);
```

8. 重命名文件。

```
const char* renamefile = "/tmp/nativeTest/testfile3.txt";
if (hdfsRename(fs, mvfile, renamefile)) {
     fprintf(stderr, "File rename failed, Old name : %s, New name : %s \n", mvfile, renamefile);
     exit(-1);
}
fprintf(stderr, "hdfsRename- SUCCESS!, File renamed, Old name : %s, New name : %s \n", mvfile,
renamefile);
```

9. 删除文件。

```
if (hdfsDelete(fs, renamefile, 0)) {
    fprintf(stderr, "File delete failed : %s \n", renamefile);
    exit(-1);
}
fprintf(stderr, "hdfsDelete- SUCCESS!, File deleted : %s\n",renamefile);
```

10. 设置副本数。

```
if (hdfsSetReplication(fs, file, 10)) {
    fprintf(stderr, "Failed to set replication : %s \n", file );
    exit(-1);
}
fprintf(stderr, "hdfsSetReplication- SUCCESS!, Set replication 10 for %s\n",file);
```

11. 设置用户、用户组。

```
if (hdfsChown(fs, file, "root", "root")) {
   fprintf(stderr, "Failed to set chown : %s \n", file );
   exit(-1);
}
fprintf(stderr, "hdfsChown- SUCCESS!, Chown success for %s\n",file);
```

12. 设置权限。

```
if (hdfsChmod(fs, file, S_IRWXU | S_IRWXG | S_IRWXO)) {
   fprintf(stderr, "Failed to set chmod: %s \n", file );
   exit(-1);
}
fprintf(stderr, "hdfsChmod- SUCCESS!, Chmod success for %s\n",file);
```

13. 设置文件时间。

```
struct timeval now;
gettimeofday(&now, NULL);
if (hdfsUtime(fs, file, now.tv_sec, now.tv_sec)) {
   fprintf(stderr, "Failed to set time: %s \n", file );
   exit(-1);
}
fprintf(stderr, "hdfsUtime- SUCCESS!, Set time success for %s\n",file);
```

14. 获取文件信息。

```
hdfsFileInfo *fileInfo = NULL;
if((fileInfo = hdfsGetPathInfo(fs, file)) != NULL) {
  printFileInfo(fileInfo);
  hdfsFreeFileInfo(fileInfo, 1);
  fprintf(stderr, "hdfsGetPathInfo - SUCCESS!\n");
}
```

15. 遍历目录。

```
hdfsFileInfo *fileList = 0;
int numEntries = 0;
if((fileList = hdfsListDirectory(fs, dir, &numEntries)) != NULL) {
  int i = 0;
  for(i=0; i < numEntries; ++i) {
      printFileInfo(fileList+i);
  }
  hdfsFreeFileInfo(fileList, numEntries);
}
fprintf(stderr, "hdfsListDirectory- SUCCESS!, %s\n", dir);
```

16. stream builder接口。

```
buffer = (char *) malloc(100);
struct hdfsStreamBuilder *builder= hdfsStreamBuilderAlloc(fs, (char*)file, O_RDONLY);
hdfsStreamBuilderSetBufferSize(builder,100);
hdfsStreamBuilderSetReplication(builder,20);
hdfsStreamBuilderSetDefaultBlockSize(builder,10485760);
readFile = hdfsStreamBuilderBuild(builder);
num_read = hdfsRead(fs, readFile, (void*)buffer, 100);
fprintf(stderr, "hdfsStreamBuilderBuild- SUCCESS! File read success. Byte read : %d, File contant : %s
\n", num_read ,buffer);
free(buffer);

struct hdfsReadStatistics *stats = NULL;
hdfsFileGetReadStatistics(readFile, &stats);
fprintf(stderr, "hdfsFileGetReadStatistics- SUCCESS! totalBytesRead : %" PRId64 ",
totalLocalBytesRead : %" PRId64 ", totalShortCircuitBytesRead : %" PRId64 ",
totalZeroCopyBytesRead : %" PRId64 "\n", stats->totalBytesRead , stats->totalLocalBytesRead, stats-
>totalShortCircuitBytesRead,  stats->totalZeroCopyBytesRead);
hdfsFileFreeReadStatistics(stats);
```

17. 断开HDFS文件系统连接。

```
hdfsDisconnect(fs);
```

准备运行环境

在节点上安装客户端，例如安装到“/opt/client”目录。

Linux 中编译并运行程序

1. 进入Linux客户端目录，运行如下命令导入公共环境变量：

   cd/opt/client

   source*bigdata_env*

2. 进入“/opt/client/HDFS/hadoop/hdfs-c-example”目录下，运行如下命令导入C客户端环境变量。

   cd/opt/client/HDFS/hadoop/hdfs-c-example

   source*component_env_C_example*

3. 清除之前运行生成的目标文件和可执行文件，运行如下命令。

   make clean

   执行结果如下：

```
[root@10-120-85-2 hdfs-c-example]# make clean
rm -f hdfs_test.o
rm -f hdfs_test
```


4. 编译生成新的目标和可执行文件，运行如下命令。

   make（或***make all***）

   执行结果如下：

   ```
   [root@10-120-85-2 hdfs-c-example]# make
   cc -c -I/opt/client/HDFS/hadoop/include -Wall -o hdfs_test.o hdfs_test.c
   cc -o hdfs_test hdfs_test.o -lhdfs
   ```

5. 运行文件以实现创建文件、读写追加文件和删除文件的功能，运行如下命令。

   make run

   执行结果如下：

   ```
   [root@10-120-85-2 hdfs-c-example]# make run
   ./hdfs_test
   hdfsConnect- SUCCESS!
   hdfsCreateDirectory- SUCCESS! : /tmp/nativeTest
   hdfsOpenFile- SUCCESS! for write : /tmp/nativeTest/testfile.txt
   hdfsWrite- SUCCESS! : /tmp/nativeTest/testfile.txt
   Flushing file data ....
   hdfsCloseFile- SUCCESS! : /tmp/nativeTest/testfile.txt
   hdfsOpenFile- SUCCESS! for read : /tmp/nativeTest/testfile.txt
   hdfsRead- SUCCESS!, Byte read : 31, File contant : Hadoop HDFS Native file write!
   hdfsSeek- SUCCESS!, Byte read : 21, File seek contant : S Native file write!
   hdfsPread- SUCCESS!, Byte read : 10, File pead contant : S Native f
   hdfsCopy- SUCCESS!, File copied, src : /tmp/nativeTest/testfile.txt, des : /tmp/nativeTest/testfile1.txt
   hdfsMove- SUCCESS!, File moved, src : /tmp/nativeTest/testfile1.txt, des : /tmp/nativeTest/testfile2.txt
   hdfsRename- SUCCESS!, File renamed, Old name : /tmp/nativeTest/testfile2.txt, New name : /tmp/
   nativeTest/testfile3.txt
   hdfsDelete- SUCCESS!, File deleted : /tmp/nativeTest/testfile3.txt
   hdfsSetReplication- SUCCESS!, Set replication 10 for /tmp/nativeTest/testfile.txt
   hdfsChown- SUCCESS!, Chown success for /tmp/nativeTest/testfile.txt
   hdfsChmod- SUCCESS!, Chmod success for /tmp/nativeTest/testfile.txt
   hdfsUtime- SUCCESS!, Set time success for /tmp/nativeTest/testfile.txt

   Name: hdfs://hacluster/tmp/nativeTest/testfile.txt, Type: F, Replication: 10, BlockSize: 134217728, Size:
   31, LastMod: 1500345260, Owner: root, Group: root, Permissions: 511 (rwxrwxrwx)
   hdfsGetPathInfo - SUCCESS!

   Name: hdfs://hacluster/tmp/nativeTest/testfile.txt, Type: F, Replication: 10, BlockSize: 134217728, Size:
   31, LastMod: 1500345260, Owner: root, Group: root, Permissions: 511 (rwxrwxrwx)
   hdfsListDirectory- SUCCESS!, /tmp/nativeTest
   hdfsTruncateFile- SUCCESS!, /tmp/nativeTest/testfile.txt
   Block Size : 134217728
   hdfsGetDefaultBlockSize- SUCCESS!
   Block Size : 134217728 for file /tmp/nativeTest/testfile.txt
   hdfsGetDefaultBlockSizeAtPath- SUCCESS!
   HDFS Capacity : 102726873909
   hdfsGetCapacity- SUCCESS!
   HDFS Used : 4767076324
   hdfsGetCapacity- SUCCESS!
   hdfsExists- SUCCESS! /tmp/nativeTest/testfile.txt
   hdfsConfGetStr- SUCCESS : hdfs://hacluster
   hdfsStreamBuilderBuild- SUCCESS! File read success. Byte read : 31, File contant : Hadoop HDFS
   Native file write!
   hdfsFileGetReadStatistics- SUCCESS! totalBytesRead : 31, totalLocalBytesRead : 0,
   totalShortCircuitBytesRead : 0, totalZeroCopyBytesRead : 0
   ```

6. 进入debug模式(可选)

   make gdb

   执行结果如下：

   ```
   [root@10-120-85-2 hdfs-c-example]# make gdb
   gdb hdfs_test
   GNU gdb (GDB) SUSE (7.5.1-0.7.29)
   Copyright (C) 2012 Free Software Foundation, Inc.
   License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
   This is free software: you are free to change and redistribute it.
   There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
   ```



```
and "show warranty" for details.
This GDB was configured as "x86_64-suse-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /opt/client/HDFS/hadoop/hdfs-c-example/hdfs_test...done.
(gdb)
```

14.7.1.3 HDFS HTTP REST API 接口介绍

功能简介

REST应用开发代码样例中所涉及的文件操作主要包括创建文件、读写文件、追加文件、删除文件。完整和详细的接口请参考官网上的描述以了解其使用：

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/WebHDFS.html

准备运行环境

步骤1 安装客户端。在节点上安装客户端，如安装到“/opt/client”目录，可参考“安装客户端”。

1. 在客户端目录准备文件“testFile”和“testFileAppend”，文件内容分别“Hello, webhdfs user!”和“Welcome back to webhdfs!”，执行如下命令准备文件。

   touch *testFile*

   vi *testFile*

   写入“Hello, webhdfs user!”保存退出。

   touch *testFileAppend*

   vi *testFileAppend*

   写入“Welcome back to webhdfs!”保存退出。

步骤2 在普通模式下，只支持使用HTTP服务访问。登录FusionInsight Manager页面，选择“集群 > *待操作集群的名称* > 服务 > HDFS > 配置 > 全部配置”，在“搜索”框里搜索“dfs.http.policy”，然后勾选“HTTP_ONLY”，单击“保存”，单击“确定”，重启HDFS服务。

说明

“HTTP_ONLY”默认是勾选的。

----结束

操作步骤

步骤1 登录FusionInsight Manager页面，单击“集群 > *待操作集群的名称* > 服务”，选择“HDFS”，单击进入HDFS服务状态页面。

说明

由于webhdfs是http访问的，需要主NameNode的IP和http端口。

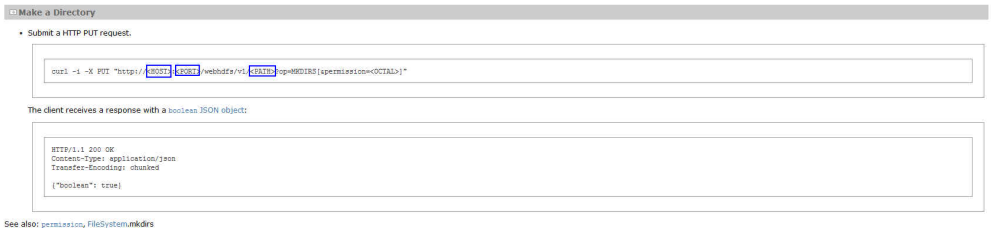
1. 单击“实例”，找到“NameNode(hacluster,主)”的主机名（host）和对应的IP。
2. 单击“配置”，在搜索框搜索“namenode.http.port”（9870）。

步骤2 参考如下链接，创建目录。

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/WebHDFS.html#Make_a_Directory

单击链接，如图14-10所示。

图 14-10 创建目录样例命令



进入到客户端的安装目录下，此处为“/opt/client”，创建名为“huawei”的目录。

1. 执行下列命令，查看当前是否存在名为“huawei”的目录。

   hdfs dfs -ls /

   执行结果如下：

   ```
   linux1:/opt/client # hdfs dfs -ls /
   16/04/22 16:10:02 INFO hdfs.PeerCache: SocketCache disabled.
   Found 7 items
   -rw-r--r--   3 hdfs   supergroup          0 2016-04-20 18:03 /PRE_CREATE_DIR.SUCCESS
   drwxr-x---   - flume  hadoop              0 2016-04-20 18:02 /flume
   drwx------   - hbase  hadoop              0 2016-04-22 15:19 /hbase
   drwxrwxrwx   - mapred hadoop              0 2016-04-20 18:02 /mr-history
   drwxrwxrwx   - spark  supergroup          0 2016-04-22 15:19 /sparkJobHistory
   drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 14:51 /tmp
   drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 14:50 /user
   ```

   当前路径下不存在“huawei”目录。

2. 执行图14-10中的命令创建以“huawei”为名的目录。其中，用步骤1中查找到的主机名或IP和端口分别替代命令中的<HOST>和<PORT>，在<PATH>中输入想要创建的目录“huawei”。

   说明

   用主机名或IP代替<HOST>都是可以的，要注意HTTP和HTTPS的端口不同。

   - 执行下列命令访问HTTP：

     ```
     curl -i -X PUT --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei?user.name=test&op=MKDIRS"
     ```

     其中用linux1代替<HOST>，用9870代替<PORT>，test为执行操作的用户，此用户需与管理员确认是否有权限进行操作。

   - 运行结果：

     ```
     HTTP/1.1 200 OK
     Cache-Control: no-cache
     Expires: Thu, 14 Jul 2016 08:04:39 GMT
     Date: Thu, 14 Jul 2016 08:04:39 GMT
     Pragma: no-cache
     Expires: Thu, 14 Jul 2016 08:04:39 GMT
     Date: Thu, 14 Jul 2016 08:04:39 GMT
     Pragma: no-cache
     Content-Type: application/json
     X-FRAME-OPTIONS: SAMEORIGIN
     Set-Cookie: hadoop.auth="u=hdfs&p=hdfs&t=simple&e=1468519479514&s=/j/J+ZnVrN7NSz1yKnB2JVIwkj0="; Path=/; Expires=Thu, 14-Jul-2016 18:04:39 GMT; HttpOnly
     ```



```
Transfer-Encoding: chunked
{"boolean":true}
```

返回值{"boolean":true}说明创建成功。

```
linux1:/opt/client # curl -i -k -X PUT --negotiate -u: "https://10.120.172.109:25003/webhdfs/v1/
huawei?op=MKDIRS"
```

3. 再执行下列命令进行查看，可以看到路径下出现“huawei”目录。

```
linux1:/opt/client # hdfs dfs -ls /
16/04/22 16:14:25 INFO hdfs.PeerCache: SocketCache disabled.
Found 8 items
-rw-r--r--   3 hdfs   supergroup          0 2016-04-20 18:03 /PRE_CREATE_DIR.SUCCESS
drwxr-x---   - flume  hadoop              0 2016-04-20 18:02 /flume
drwx------   - hbase  hadoop              0 2016-04-22 15:19 /hbase
drwxr-xr-x   - hdfs   supergroup          0 2016-04-22 16:13 /huawei
drwxrwxrwx   - mapred hadoop              0 2016-04-20 18:02 /mr-history
drwxrwxrwx   - spark  supergroup          0 2016-04-22 16:12 /sparkJobHistory
drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 14:51 /tmp
drwxrwxrwx   - hdfs   hadoop              0 2016-04-22 16:10 /user
```

步骤3 创建请求上传命令，获取集群分配的可写入DataNode节点地址的信息Location。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -i -X PUT --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?
user.name=test&op=CREATE"
```

- 运行结果：

```
HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Thu, 14 Jul 2016 08:53:07 GMT
Date: Thu, 14 Jul 2016 08:53:07 GMT
Pragma: no-cache
Expires: Thu, 14 Jul 2016 08:53:07 GMT
Date: Thu, 14 Jul 2016 08:53:07 GMT
Pragma: no-cache
Content-Type: application/octet-stream
X-FRAME-OPTIONS: SAMEORIGIN
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs&t=simple&e=1468522387880&s=OIksfRJvEkh/
Out9y2Ot2FvrxWk="; Path=/; Expires=Thu, 14-Jul-2016 18:53:07 GMT; HttpOnly
Location: http://10-120-180-170:25010/webhdfs/v1/testHdfs?
op=CREATE&user.name=hdfs&namenoderpcaddress=hacluster&createflag=&createparent=true&overwr
ite=false
Content-Length: 0
```

步骤4 根据获取的Location地址信息，可在HDFS文件系统上创建“/huawei/testHdfs”文件，并将本地“testFile”中的内容上传至“testHdfs”文件。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -i -X PUT -T testFile --negotiate -u: "http://10-120-180-170:25010/webhdfs/v1/
testHdfs?
op=CREATE&user.name=test&namenoderpcaddress=hacluster&createflag=&createparent=true&overwri
te=false"
```

- 运行结果：

```
HTTP/1.1 100 Continue

HTTP/1.1 201 Created
Location: hdfs://hacluster/testHdfs
Content-Length: 0
Connection: close
```

步骤5 打开“/huawei/testHdfs”文件，并读取文件中上传写入的内容。

- 执行如下命令访问HTTP：

```
curl -L --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?user.name=test&op=OPEN"
```

- 运行结果：

```
Hello, webhdfs user!
```

步骤6 创建请求追加文件的命令，获取集群为已存在“/huawei/testHdfs”文件分配的可写入DataNode节点地址信息Location。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl -i -X POST --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?user.name=test&op=APPEND"
```
- 运行结果：
```
HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Thu, 14 Jul 2016 09:18:30 GMT
Date: Thu, 14 Jul 2016 09:18:30 GMT
Pragma: no-cache
Expires: Thu, 14 Jul 2016 09:18:30 GMT
Date: Thu, 14 Jul 2016 09:18:30 GMT
Pragma: no-cache
Content-Type: application/octet-stream
X-FRAME-OPTIONS: SAMEORIGIN
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs&t=simple&e=1468523910234&s=JGK+6M6PsVMFdAw2cgIHaKU1kBM="; Path=/; Expires=Thu, 14-Jul-2016 19:18:30 GMT; HttpOnly
Location: http://10-120-180-170:25010/webhdfs/v1/testHdfs?op=APPEND&user.name=hdfs&namenoderpcaddress=hacluster
Content-Length: 0
```

步骤7 根据获取的Location地址信息，可将本地“testFileAppend”文件中的内容追加到HDFS文件系统上的“/huawei/testHdfs”文件。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl -i -X POST -T testFileAppend --negotiate -u: "http://10-120-180-170:25010/webhdfs/v1/huawei/testHdfs?op=APPEND&user.name=hdfs&namenoderpcaddress=hacluster"
```
- 运行结果：
```
HTTP/1.1 100 Continue
HTTP/1.1 200 OK
Content-Length: 0
Connection: close
```

步骤8 打开“/huawei/testHdfs”文件，并读取文件中全部的内容。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl -L --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?user.name=test&op=OPEN"
```
- 运行结果：
```
Hello, webhdfs user!
Welcome back to webhdfs!
```

步骤9 可列出文件系统上“huawei”目录下所有目录和文件的详细信息。

LISTSTATUS将在一个请求中返回所有子文件和文件夹的信息。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/testHdfs?user.name=test&op=LISTSTATUS"
```
- 运行结果：
```
{"FileStatuses":{"FileStatus":[
{"accessTime":1462425245595,"blockSize":134217728,"childrenNum":0,"fileId":17680,"group":"supergroup","length":70,"modificationTime":1462426678379,"owner":"test","pathSuffix":"","permission":"755","replication":3,"storagePolicy":0,"type":"FILE"}
]}}
```

带有大小参数和startafter参数的LISTSTATUS将有助于通过多个请求获取子文件和文件夹信息，从而避免获取大量子文件和文件夹信息时，用户界面变慢。

- 执行如下命令访问HTTP：
```
linux1:/opt/client # curl --negotiate -u: "http://linux1:9870/webhdfs/v1/huawei/?user.name=test&op=LISTSTATUS&startafter=sparkJobHistory&size=1"
```
- 运行结果：
```
{"FileStatuses":{"FileStatus":[
{"accessTime":1462425245595,"blockSize":134217728,"childrenNum":0,"fileId":17680,"group":"supergroup","length":70,"modificationTime":1462426678379,"owner":"test","pathSuffix":"testHdfs","permissio
```



```
n":"755","replication":3,"storagePolicy":0,"type":"FILE"}
]}}
```

步骤10 删除HDFS上的文件“/huawei/testHdfs”。

- 执行如下命令访问HTTP：

```
linux1:/opt/client # curl -i -X DELETE  --negotiate -u: "http://linux1:25002/webhdfs/v1/huawei/testHdfs?user.name=test&op=DELETE"
```

- 运行结果：

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Expires: Thu, 14 Jul 2016 10:27:44 GMT
Date: Thu, 14 Jul 2016 10:27:44 GMT
Pragma: no-cache
Expires: Thu, 14 Jul 2016 10:27:44 GMT
Date: Thu, 14 Jul 2016 10:27:44 GMT
Pragma: no-cache
Content-Type: application/json
X-FRAME-OPTIONS: SAMEORIGIN
Set-Cookie: hadoop.auth="u=hdfs&p=hdfs&t=simple&e=1468528064220&s=HrvUEd72+V5L4GwCLC/sG3xTI0o="; Path=/; Expires=Thu, 14-Jul-2016 20:27:44 GMT; HttpOnly
Transfer-Encoding: chunked
{"boolean":true}
```

----结束

密钥管理系统通过HTTP REST API对外提供密钥管理服务，接口请参考官网：

http://hadoop.apache.org/docs/r3.1.1/hadoop-kms/index.html

说明

由于REST API接口做了安全加固，防止脚本注入攻击。通过REST API的接口，无法创建包含"<script ", "<iframe", "<frame", "javascript:" 这些关键字的目录和文件名。

14.7.2 HDFS Shell 命令介绍

HDFS Shell

您可以使用HDFS Shell命令对HDFS文件系统进行操作，例如读文件、写文件等操作。

执行HDFS Shell的方法：

进入HDFS客户端如下目录，直接输入命令即可。例如：

cd /opt/client/HDFS/hadoop/bin

./hdfs dfs -mkdir /tmp/input

执行如下命令查询HDFS命令的帮助。

./hdfs --help

HDFS命令行参考请参见官网：

http://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-common/FileSystemShell.html

表 14-14 透明加密相关命令

场景	操作	命令	描述
hadoop shell命令管理密钥	创建密钥	***hadoop key create*** *<keyname> [-cipher <cipher>] [-size <size>] [-description <description>] [-attr <attribute=value>] [-provider <provider>] [-help]*	create子命令为provider中<keyname>参数指定的name创建一个新的密钥，provider是由-provider参数指定。用户可以使用参数-cipher定义一个密码。目前默认的密码为"AES/CTR/NoPadding"。 默认密钥的长度为128。用户可以使用参数-size定义需要的密钥的长度。任意的attribute=value类型属性可以用参数-attr定义。每一个属性，-attr可以被定义很多次。
	回滚操作	***hadoop key roll*** *<keyname> [-provider <provider>] [-help]*	roll子命令为provider中指定的key创建一个新的版本，provider是由-provider参数指定。
	删除密钥	***hadoop key delete*** *<keyname> [-provider <provider>] [-f] [-help]*	delete子命令删除key的所有版本，key是由provider中的<keyname>参数指定，provider是由-provider参数指定。除非-f被指定否则该命令需要用户确认。
	查看密钥	***hadoop key list*** *[-provider <provider>] [-metadata] [-help]*	list子命令显示provider中所有的密钥名，这个provider由用户在core-site.xml中配置或者由-provider参数指定。-metadata参数显示的是元数据。

表 14-15 Colocation 客户端 shell 命令

操作	命令	描述
创建组	hdfs colocationadmin -createGroup -groupId <groupID> -locatorIds <comma separated locatorIDs> or -file <path of the file contains all of locatorIDs>	创建组，groupID为组名，locatorID为locator名，locatorID可通过命令行输入，多个locatorID之间用逗号分隔；也可将locatorID写入文件，通过读文件获取。
删除组	hdfs colocationadmin -deleteGroup <groupID>	删除指定组。
查询组	hdfs colocationadmin -queryGroup <groupID>	查询指定组的详细信息，包括该group包含的locators以及每个locator及其对应的DataNode。

操作	命令	描述
查看所有组	hdfs colocationadmin -listGroups	列出所有组及其创建时间。
设置colocation根目录的acl权限	hdfs colocationadmin -setAcl	设置zookeeper中colocation根目录的acl权限。 colocation在zookeeper中的根目录默认为/hadoop/colocationDetails。

14.7.3 配置 Windows 通过 EIP 访问普通模式集群 HDFS

操作场景

该章节通过指导用户配置集群绑定EIP，并配置HDFS文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行HdfsExample样例为例进行说明。

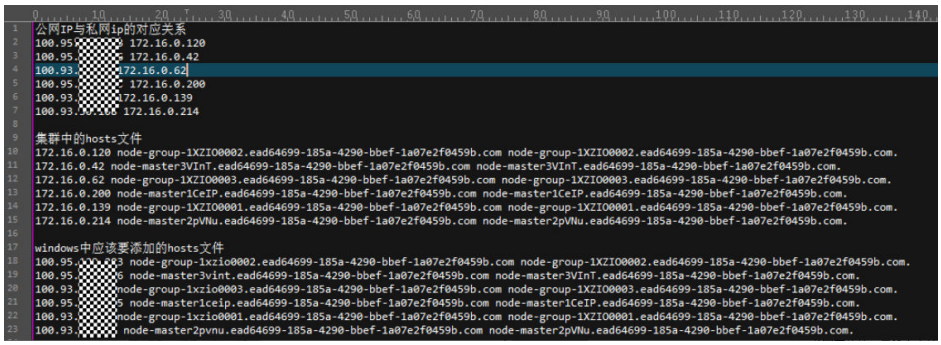
操作步骤

步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。



步骤2 配置集群安全组规则。

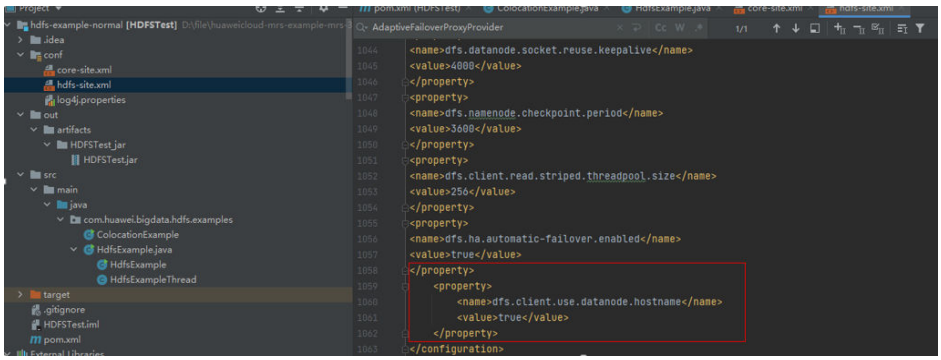
1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和8020、9866端口。

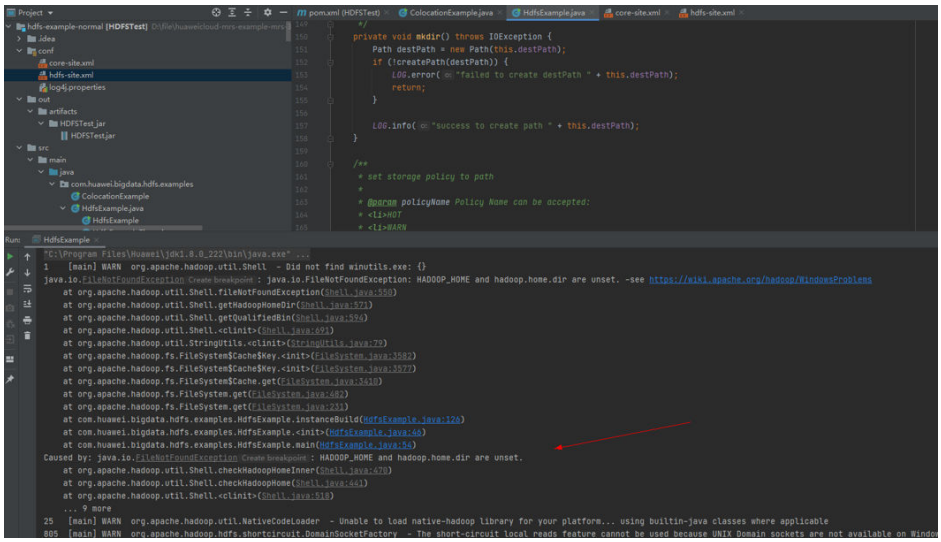
步骤3 在Manager界面选择“集群 > 服务 > HDFS > 更多 > 下载客户端”，将客户端中的core-site.xml和hdfs-site.xml复制到样例工程的conf目录下，并对hdfs-site.xml添加以下内容：

```
<property>
 <name>dfs.client.use.datanode.hostname</name>
 <value>true</value>
 </property>
```

（将datanode的通信改成通过hostname）



在运行样例工程时，可能会报一个没有hadoop_home的异常，这个异常可以忽略。



----结束

15 Hive 开发指南（安全模式）

15.1 Hive 应用开发概述

15.1.1 Hive 应用开发简介

Hive 简介

Hive是一个开源的，建立在Hadoop上的数据仓库框架，提供类似SQL的HQL语言操作结构化数据，其基本原理是将HQL语言自动转换成Mapreduce任务或Spark任务，从而完成对Hadoop集群中存储的海量数据进行查询和分析。

Hive主要特点如下：

- 通过HQL语言非常容易的完成数据提取、转换和加载（ETL）。
- 通过HQL完成海量结构化数据分析。
- 灵活的数据存储格式，支持JSON、CSV、TEXTFILE、RCFILE、ORCFILE、SEQUENCEFILE等存储格式，并支持自定义扩展。
- 多种客户端连接方式，支持JDBC接口。

Hive的主要应用于海量数据的离线分析（如日志分析，集群状态分析）、大规模的数据挖掘（用户行为分析，兴趣分区，区域展示）等场景下。

为保证Hive服务的高可用性、用户数据的安全及访问服务的可控制，在开源社区的Hive-3.1.0版本基础上，Hive新增如下特性：

- 基于Kerberos技术的安全认证机制。
- 数据文件加密机制。
- 完善的权限管理。

开源社区的Hive特性，请参见**https://cwiki.apache.org/confluence/display/hive/designdocs**。

15.1.2 Hive 应用开发常用概念

- **keytab文件**

存放用户信息的密钥文件。应用程序采用此密钥文件在MRS产品中进行API方式认证。

- **客户端**

  客户端直接面向用户，可通过Java API、Thrift API访问服务端进行Hive的相关操作。

- **HQL语言**

  Hive Query Language，类SQL语句。

- **HCatalog**

  HCatalog是建立在Hive元数据之上的一个表信息管理层，吸收了Hive的DDL命令。为Mapreduce提供读写接口，提供Hive命令行接口来进行数据定义和元数据查询。基于MRS的HCatalog功能，Hive、Mapreduce开发人员能够共享元数据信息，避免中间转换和调整，能够提升数据处理的效率。

- **WebHCat**

  WebHCat运行用户通过Rest API来执行Hive DDL，提交Mapreduce任务，查询Mapreduce任务执行结果等操作。

15.1.3 Hive 应用开发开发流程

开发流程中各阶段的说明如图15-1和表15-1所示。

图 15-1 Hive 应用程序开发流程

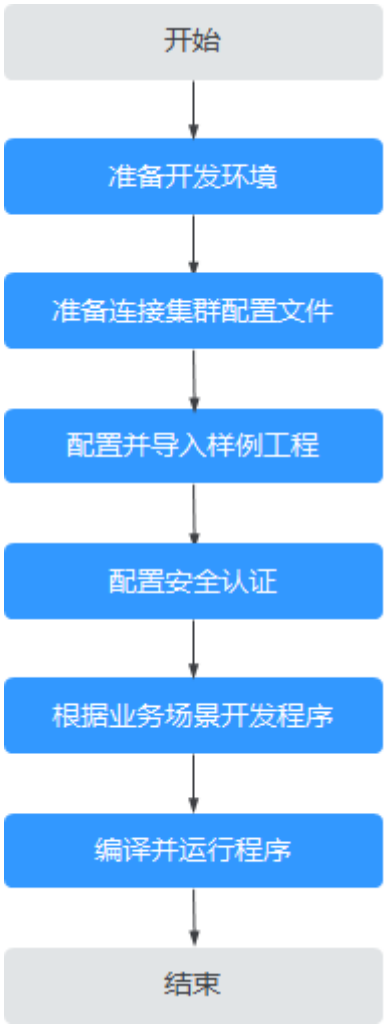


表 15-1 Hive 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，推荐使用Java语言进行开发，使用IntelliJ IDEA工具，同时完成JDK、Maven等初始配置。	准备本地应用开发环境

阶段	说明	参考文档
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备连接Hive集群配置文件**
配置并导入样例工程	HIve提供了不同场景下的多种样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。	**导入并配置Hive样例工程**
配置安全认证	如果您使用的是JDBC访问开启了Kerberos认证的MRS集群，需要进行安全认证。	**配置Hive JDBC接口访问Hive安全认证**
根据业务场景开发程序	根据实际业务场景开发程序，调用组件接口实现对应功能。	**开发Hive应用**
编译并运行程序	指导用户将开发好的程序编译提交运行并查看结果。	**调测Hive应用**

15.1.4 Hive 应用开发样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Hive相关样例工程：

表 15-2 Hive 相关样例工程

样例工程位置	描述
• hive-examples/hive-jdbc-example • hive-examples/hive-jdbc-example-multizk	Hive JDBC处理数据Java示例程序。 本工程使用JDBC接口连接Hive，在Hive中执行相关数据操作。使用JDBC接口实现创建表、加载数据、查询数据等功能，还可实现在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper，相关样例介绍请参见**Hive JDBC访问样例程序**。

样例工程位置	描述
hive-examples/hcatalog-example	Hive HCatalog处理数据Java示例程序。 使用HCatalog接口实现通过Hive命令行方式对MRS Hive元数据进行数据定义和查询操作，相关样例介绍请参见**HCatalog访问Hive样例程序**。
hive-examples/python-examples	使用Python连接Hive执行SQL样例。 可实现使用Python对接Hive并提交数据分析任务，相关样例介绍请参见**基于Python的Hive样例程序**。
hive-examples/python3-examples	使用Python3连接Hive执行SQL样例。 可实现使用Python3对接Hive并提交数据分析任务，相关样例介绍请参见**基于Python3的Hive样例程序**。

15.2 准备 Hive 应用开发环境

15.2.1 准备本地应用开发环境

Hive组件可以使用JDBC、HCatalog、Python、Python3接口进行应用开发。

准备 JDBC/HCatalog 开发环境

表 15-3 JDBC/HCatalog 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。

准备项	说明
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 ● X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 ● TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装和配置IntelliJ IDEA	用于开发Hive应用程序的工具。版本要求如下： JDK使用1.8版本，IntelliJ IDEA使用2019.1或其他兼容版本。 **说明** ● 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 ● 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 ● 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。 华为提供开源镜像站，各服务样例工程依赖的Jar包通过华为开源镜像站下载，剩余所依赖的开源Jar包请直接从Maven中央库或者其他用户自定义的仓库地址下载，详情请参考**配置华为开源镜像仓**。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

准备 Python 开发环境

表 15-4 Python 开发环境

准备项	说明
操作系统	开发环境和运行环境：Linux系统。
安装Python	用于开发Hive应用程序的工具，版本要求不低于2.6.6，最高不超过2.7.13。

准备项	说明
安装setuptools	Python开发环境的基本配置，要求5.0之后版本。

说明

Python开发工具的详细安装配置可参见**配置Hive Python样例工程**。

准备 Python3 开发环境

表 15-5 Python3 开发环境

准备项	说明
操作系统	开发环境和运行环境：Linux系统。
安装Python3	用于开发Hive应用程序的工具，版本要求不低于3.6，最高不超过3.8。
安装setuptools	Python3开发环境的基本配置，要求为47.3.1版本。

说明

Python3开发工具的详细安装配置可参见**配置Hive Python3样例工程**。

15.2.2 准备连接 Hive 集群配置文件

准备集群认证用户信息

对于开启Kerberos认证的MRS集群，需提前准备具有相关组件操作权限的用户用于程序认证。

以下Hive权限配置示例供参考，在实际业务场景中可根据业务需求灵活调整。

步骤1 登录FusionInsight Manager。

步骤2 选择“集群 > 服务 > Hive > 更多 > 启用Ranger鉴权”，查看该参数是否置灰。

- 是，创建用户并在Ranger中赋予该用户相关操作权限：
  a. 选择“系统 > 权限 > 用户 > 添加用户”，在新增用户界面创建一个机机用户，例如**developuser**。
     “用户组”需加入“hive”用户组。
  b. 使用Ranger管理员用户**rangeradmin**登录Ranger管理页面。
     rangeradmin用户默认密码为“Rangeradmin@123”，详细内容请参见**用户账号一览表**。
  c. 在首页中单击“Settings”，选择“Roles”。
  d. 单击Role Name为admin的角色，在“Users”区域，单击“Select User”，选择**步骤2.a**创建的用户名。

e. 单击Add Users按钮，在对应用户名所在行勾选“Is Role Admin”，单击“Save”保存配置。

说明

若在对数据库/表进行相关操作时指定了HDFS路径，需参考**表15-7**在Ranger中给用户添加HDFS路径的相关权限。

- 否，创建用户并在Manager赋予用户相关操作权限：

a. 选择“系统 > 权限 > 角色 > 添加角色”。

i. 填写角色的名称，例如**developrole**。

ii. 在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选“default”的“提交”和“管理”，单击“确定”保存。

说明

若在对数据库/表进行相关操作时指定了HDFS路径，需参考**表15-7**在Manager中给用户添加HDFS路径的相关权限。

b. 选择“用户 > 添加用户”，在新增用户界面，创建一个机机用户，例如**developuser**。

- “用户组”需加入“hive”用户组。
- “角色”加入**步骤2.a**新增的角色。

步骤3 使用**admin**用户登录FusionInsight Manager，选择“系统 > 权限 > 用户”，在用户名为**developuser**的操作列选择“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到该用户的“user.keytab”文件与“krb5.conf”文件。

----结束

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。

a. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。

b. 进入客户端解压路径“Hive\config”，获取**表15-6**中相关配置文件。

表 15-6 配置文件

文件名称	作用
hiveclient.properties	Hive客户端连接相关配置参数。
core-site.xml	Hadoop客户端相关配置参数。

c. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群，具体操作请参考**配置Windows通过EIP访问安全模式集群Hive**。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。

a. 在节点中安装客户端。

例如客户端安装目录为“/opt/client”。

客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。

b. 获取配置文件：

i. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**），勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。

ii. 以**root**用户登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“Hive/config”路径下的**表15-6**中相关配置文件。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar

cd FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles/Hive/config

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

Hive 操作权限

在进行应用程序开发之前，使用的用户的基本权限要求是属于Hive组，额外操作权限需从系统管理员处获取，具体权限要求请参见**表15-7**。用户运行样例程序，需要在default数据库中有CREATE权限。

表 15-7 操作权限要求

操作类型/作用对象	操作	权限要求
DATABASE	CREATE DATABASE *dbname* [LOCATION "hdfs_path"]	如果指定了HDFS路径hdfs_path，需要是路径hdfs_path的所有者和具有RWX权限。
	DROP DATABASE *dbname*	拥有数据库dbname的所有权。
	ALTER DATABASE *dbname*SET OWNER*user_or_role*	具有admin权限。
TABLE	CREATE TABLE *table_a*	拥有数据库的CREATE权限。
	CREATE TABLE *table_a*AS SELECT*table_b*	拥有数据库的CREATE权限，对表table_b拥有SELECT权限。
	CREATE TABLE *table_a* LIKE*table_b*	拥有数据库的CREATE权限。
	CREATE [EXTERNAL] TABLE *table_a* LOCATION "hdfs_path"	拥有数据库的CREATE权限，是HDFS上的数据路径hdfs_path的所有者和具有RWX权限。
	DROP TABLE *table_a*	是表table_a的所有者。
	ALTER TABLE *table_a* SET LOCATION "hdfs_path"	是表table_a的所有者，是HDFS上的数据路径hdfs_path的所有者和具有RWX权限。
	ALTER TABLE *table_a*SET *FILEFORMAT*	是表table_a的所有者。
	TRUNCATE TABLE *table_a*	是表table_a的所有者。
	ANALYZE TABLE *table_a* COMPUTE STATISTICS	对表table_a拥有SELECT和INSERT权限。
	SHOW TBLPROPERTIES *table_a*	对表table_a拥有SELECT权限。
	SHOW CREATE TABLE *table_a*	对表table_a拥有SELECT且带有WITH GRANT OPTION的权限。
Alter	ALTER TABLE *table_a* ADD COLUMN	是表table_a的所有者。

操作类型/作用对象	操作	权限要求
	ALTER TABLE *table_a* REPLACE COLUMN	是表table_a的所有者。
	ALTER TABLE *table_a* RENAME	是表table_a的所有者。
	ALTER TABLE *table_a* SET SERDE	是表table_a的所有者。
	ALTER TABLE *table_a* CLUSTER BY	是表table_a的所有者。
PARTITION	ALTER TABLE *table_a* ADD PARTITION*partition_spec* LOCATION "hdfs_path"	对表table_a拥有INSERT权限，是HDFS上的数据路径hdfs_path的所有者和具有RWX权限。
	ALTER TABLE *table_a* DROP PARTITION*partition_spec*	对表table_a拥有DELETE权限。
	ALTER TABLE *table_a* PARTITION*partition_spec* SET LOCATION "hdfs_path"	是表table_a的所有者，是HDFS上的数据路径hdfs_path的所有者和具有RWX权限。
	ALTER TABLE *table_a*PARTITION*partition_spec* SET FILEFORMAT	是表table_a的所有者。
LOAD	LOAD INPATH 'hdfs_path' INTO TABLE *table_a*	对表table_a拥有INSERT权限，是HDFS上的数据路径hdfs_path的所有者和具有RWX权限。
INSERT	INSERT TABLE *table_a* SELECT FROM*table_b*	对表table_a拥有INSERT权限，对表table_b拥有SELECT权限。拥有Yarn的default队列的Submit权限。
SELECT	SELECT * FROM *table_a*	对表table_a拥有SELECT权限。
	SELECT FROM *table_a*JOIN*table_b*	对表table_a、表table_b拥有SELECT权限，拥有Yarn的default队列的Submit权限。
	SELECT FROM (SELECT FROM *table_a*UNION ALL SELECT FROM*table_b*)	对表table_a、表table_b拥有SELECT权限。拥有Yarn的default队列的Submit权限。

操作类型/作用对象	操作	权限要求
EXPLAIN	EXPLAIN [EXTENDED| DEPENDENCY] *query*	对相关表目录具有RX权限。
VIEW	CREATE VIEW *view_name* AS SELECT ...	对相关表拥有SELECT且带有WITH GRANT OPTION的权限。
	ALTER VIEW *view_name* RENAME TO*new_view_name*	是视图view_name的所有者。
	DROP VIEW *view_name*	是视图view_name的所有者。
FUNCTION	CREATE [TEMPORARY] FUNCTION *function_name* AS 'class_name'	具有admin权限。
	DROP [TEMPORARY] *function_name*	具有admin权限。
MACRO	CREATE TEMPORARY MACRO *macro_name* ...	具有admin权限。
	DROP TEMPORARY MACRO *macro_name*	具有admin权限。

说明

- 以上所有的操作只要拥有Hive的admin权限以及对应的HDFS目录权限就能做相应的操作。
- 如果当前组件使用了Ranger进行权限控制，需基于Ranger配置相关策略进行权限管理。

15.2.3 导入并配置 Hive 样例工程

15.2.3.1 导入并配置 Hive JDBC/HCatalog 样例工程

操作场景

为了运行MRS产品Hive组件的JDBC/HCatalog接口样例代码，需要完成下面的操作。

说明

- 以在Windows环境下开发JDBC/HCatalog方式连接Hive服务的应用程序为例。
- HCatalog样例仅支持在Linux节点上运行。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“hive-jdbc-example”或“hcatalog-example”。

步骤2 对于“hive-jdbc-example”样例工程，还需将以下文件放置到“hive-jdbc-example\src\main\resources”目录下：

- **准备集群认证用户信息**获取的认证文件“user.keytab”和“krb5.conf”。
- **准备运行环境配置文件**获取的配置文件“core-site.xml”、“hiveclient.properties”。

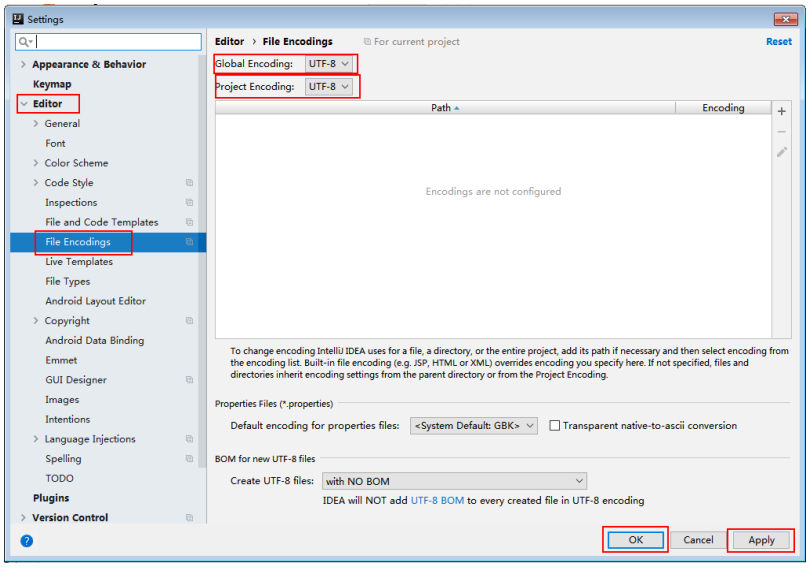
步骤3 导入样例工程到IntelliJ IDEA开发环境中。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Open...”，显示“Open File or Project”对话框。
2. 在弹出窗口选择文件夹“hive-jdbc-example”，单击“OK”。Windows下要求该文件夹的完整路径不包含空格。

步骤4 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

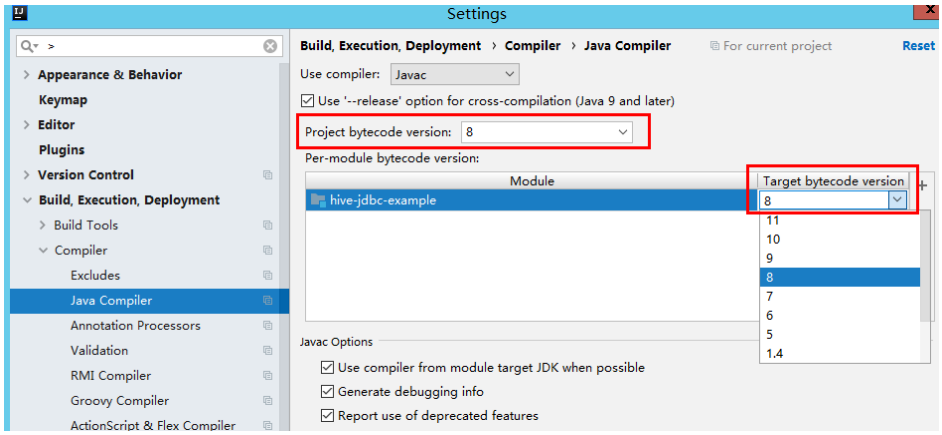
1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”。
   弹出“Settings”窗口。
2. 在左边导航上选择“Editor > File Encodings”，在“Project Encoding”和“Global Encoding”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图15-2所示。

图 15-2 设置 IntelliJ IDEA 的编码格式

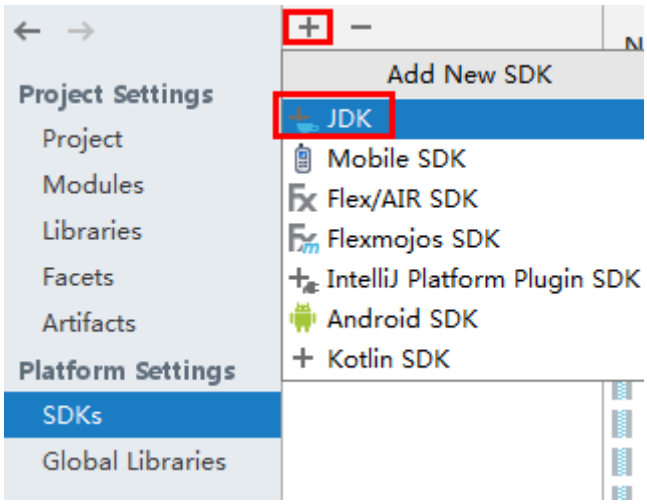


步骤5 设置工程JDK。

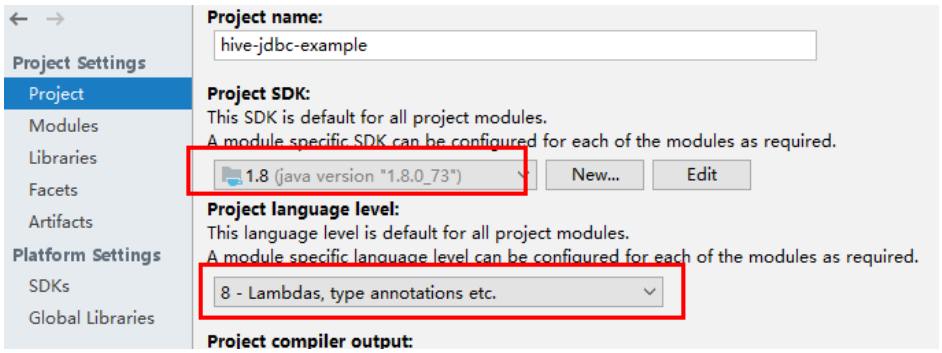
1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”，弹出“Settings”窗口。
2. 选择“Build, Execution, Deployment > Compiler > Java Compiler”，在“Project bytecode version”右侧的下拉菜单中，选择“8”。修改“hive-jdbc-example”的“Target bytecode version”为“8”。



3. 单击“Apply”后单击“OK”。
4. 在IntelliJ IDEA的菜单栏中，选择“File > Project Structure...”，弹出“Project Structure”窗口。
5. 选择“SDKs”，单击加号选择“JDK”。

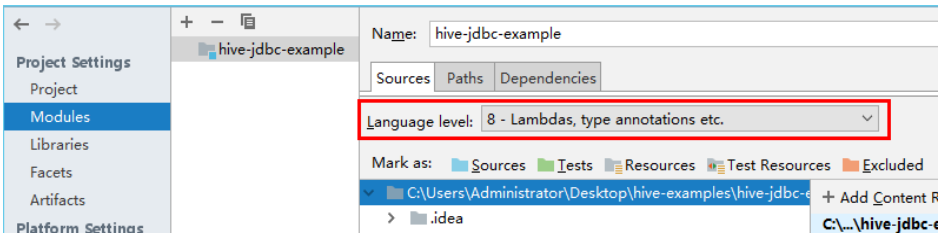


6. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。
7. 完成JDK选择后，单击“Apply”。
8. 选择“Project”，在“Project SDK”下的下拉菜单中选择在“SDKs”中添加的JDK，在“Project language level”下的下拉菜单中选择“8 - Lambdas, type annotations etc.”。

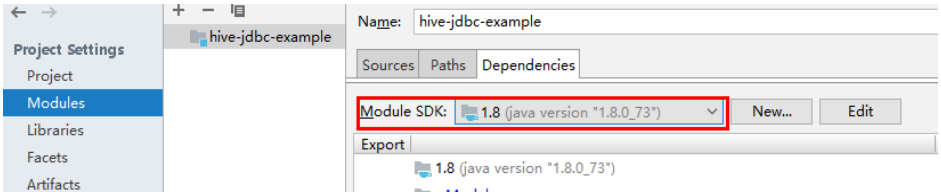


9. 单击“Apply”。

10. 选择“Modules”，在“Source”页面，修改“Language level”为“8 - Lambdas, type annotations etc.”。



在“Dependencies”页面，修改“Module SDK”为“SDKs”中添加的JDK。

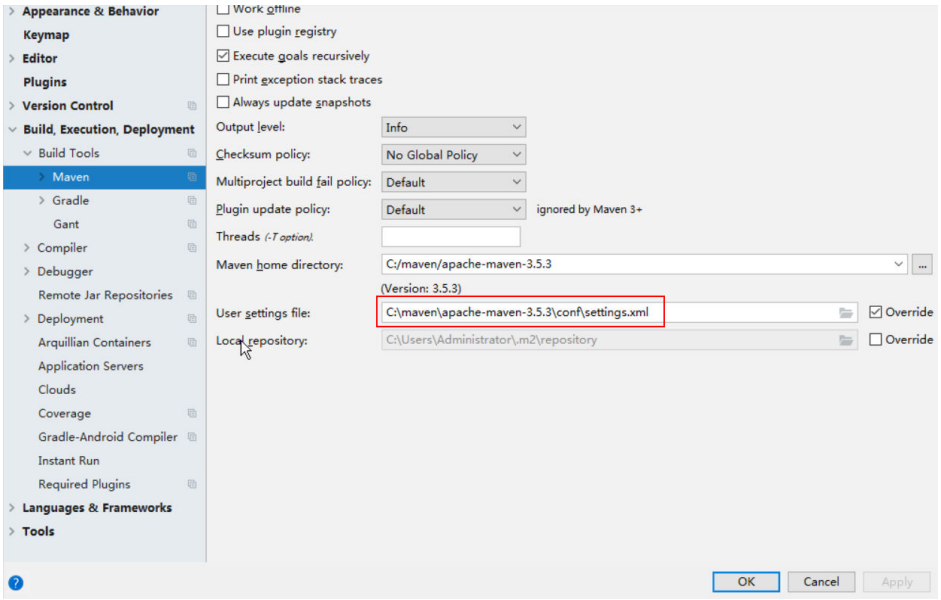


11. 单击“Apply”，单击“OK”。

步骤6 配置Maven。

1. 参考**配置华为开源镜像仓**章节描述，增加开源镜像仓地址等配置信息到本地Maven的“setting.xml”配置文件。
2. 修改完成后，在IntelliJ IDEA选择“File > Settings > Build, Execution, Deployment > Build Tools > Maven”，勾选“User settings file”右侧的“Override”，并修改“User settings file”的值为当前“settings.xml”文件放置目录，确保该目录为“*<本地Maven安装目录>*\conf\settings.xml”。

图 15-3 “settings.xml”文件放置目录



3. 单击“Maven home directory”右侧的下拉菜单，选择Maven的安装路径。
4. 单击“Apply”并单击“OK”。

----结束

15.2.3.2 配置 Hive Python 样例工程

操作场景

为了运行MRS产品Hive组件的Python接口样例代码，需要完成下面的操作。

说明

- MRS 3.1.2及之后版本默认仅支持Python3。
- 该样例仅支持在Linux节点上运行。

操作步骤

步骤1 客户端机器必须安装有Python，其版本不低于2.6.6，最高不能超过2.7.13。

在客户端机器的命令行终端输入**python**可查看Python版本号。如下显示Python版本为2.6.6。

```
Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)
[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

步骤2 客户端机器必须安装有setuptools，其版本不低于5.0，最高不能超过36.8.0。可在**https://pypi.org/project/setuptools/#files**获取相应的安装包。

将下载的setuptools压缩文件复制到客户端机器上，解压后进入解压目录，在客户端机器的命令行终端执行以下命令：

python setup.py install

如下内容表示安装setuptools的5.7版本成功：

```
Finished processing dependencies for setuptools==5.7
```

步骤3 安装Python客户端到客户端机器。

1. 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“python-examples”。
2. 进入“python-examples”文件夹。
3. 在命令行终端执行以下命令

   python setup.py install

   输出以下关键内容表示安装Python客户端成功：

   ```
   Finished processing dependencies for pyhs2==0.5.0
   ```

步骤4 安装成功后，“python-examples/pyCLI_sec.py”为Python客户端样例代码，“python-examples/pyhs2/haconnection.py”为Python客户端接口API。

“python-examples/hive_python_client”脚本提供了直接执行SQL的功能，例如：**sh hive_python_client 'show tables'**。

说明

该功能只适用于常规简单的SQL，并且需要依赖ZooKeeper的客户端。

----结束

15.2.3.3 配置 Hive Python3 样例工程

操作场景

为了运行MRS产品Hive组件的Python3接口样例代码，需要完成下面的操作。

该样例仅支持在Linux节点上运行。

操作步骤

步骤1 客户端机器必须安装有Python3，其版本不低于3.6，最高不能超过3.8。

在客户端机器的命令行终端输入**python3**可查看Python版本号。如下显示Python版本为3.8.2。

```
Python 3.8.2 (default, Jun 23 2020, 10:26:03)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

步骤2 客户端机器必须安装有setuptools，版本为47.3.1。可在**https://pypi.org/project/setuptools/#files**下载相应的安装包。

将下载的setuptools压缩文件复制到客户端机器上，解压后进入解压目录，在客户端机器的命令行终端执行以下命令：

python3 setup.py install

如下内容表示安装setuptools的47.3.1版本成功。

```
Finished processing dependencies for setuptools==47.3.1
```

说明

若提示setuptools的47.3.1版本安装不成功，则需要检查环境是否有问题或是Python自身原因导致的。

步骤3 安装Python客户端到客户端机器。

1. 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“python3-examples”。
2. 进入“python3-examples”文件夹。
3. 根据python3的版本，选择进入“dependency_python3.6”或“dependency_python3.7”或“dependency_python3.8”文件夹。
4. 执行**whereis easy_install**命令，找到easy_install程序路径。如果有多个路径，使用**easy_install --version**确认选择setuptools对应版本的**easy_install**，例如：/usr/local/bin/easy_install。
5. 使用对应的easy_install命令，安装dependency_python3.x文件夹下的egg文件，egg文件存在依赖关系，可使用通配符安装，如：
   - “dependency_python3.6”目录：

     /usr/local/bin/easy_install future*egg six*egg python*egg sasl-*linux-$(uname -p).egg thrift-*egg thrift_sasl*egg
   - “dependency_python3.7”目录：

     /usr/local/bin/easy_install future*egg six*egg sasl-*linux-$(uname -p).egg thrift-*egg thrift_sasl*egg
   - “dependency_python3.8”目录：

/usr/local/bin/easy_install future*egg six*egg python*egg sasl-*linux-$(uname -p).egg thrift-*linux-$(uname -p).egg thrift_sasl*egg

每个egg文件安装输出以下关键内容表示安装成功。

```
Finished processing dependencies for ***
```

步骤4 安装成功后，“python3-examples/pyCLI_sec.py”为Python客户端样例代码，“python3-examples/pyhive/hive.py”为Python客户端接口API。

----结束

15.2.4 配置 Hive JDBC 接口访问 Hive 安全认证

在开启了Kerberos认证的集群中，客户端连接组件之前需要进行安全认证，以确保通信的安全性，Hive应用开发需要进行ZooKeeper和Kerberos安全认证。

JDBC样例工程包含安全认证代码，支持在Windows与Linux环境运行，不依赖集群客户端。HCatalog、Python、Python3样例工程仅提供Linux环境运行示例，依赖Linux环境已安装的集群客户端进行认证，样例工程代码不涉及安全认证。

前提条件

已获取样例工程代码以及运行所需的配置文件及认证文件，详细操作请参见**准备连接Hive集群配置文件**。

配置安全登录

安全认证主要采用代码认证方式，支持Oracle JAVA平台和IBM JAVA平台。

以下代码在“hive-examples/hive-jdbc-example”样例工程的“com.huawei.bigdata.hive.example”包中，该包包括JDBCExample和JDBCExamplePreLogin类，实现的功能相同，只是认证方式有区别。JDBCExample使用JDBC连接中拼接keytab的方式进行认证；JDBCExamplePreLogin的JDBC连接中不包含认证信息，使用Hadoop通用接口UserGroupInformation认证。

根据实际情况，在JDBCExample或JDBCExamplePreLogin类中修改“USER_NAME”为实际用户名，例如“developuser”，样例代码如下：

```
// 设置新建用户的USER_NAME，其中"xxx"为已创建的用于认证的用户名，例如创建的用户为developuser，则USER_NAME为developuser
USER_NAME = "xxx";
if ("KERBEROS".equalsIgnoreCase(auth)) {
    // 设置客户端的keytab和zookeeper认证principal
    USER_KEYTAB_FILE = "src/main/resources/user.keytab";
    ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/" + getUserRealm();
    System.setProperty(ZOOKEEPER_SERVER_PRINCIPAL_KEY, ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);
}
```

15.3 开发 Hive 应用

15.3.1 Hive JDBC 访问样例程序

15.3.1.1 Hive JDBC 样例程序开发思路

场景说明

假定用户开发一个Hive数据分析应用，用于管理企业雇员信息，如表15-8、表15-9所示。

开发思路

步骤1 数据准备。

1. 创建三张表，雇员信息表“employees_info”、雇员联络信息表“employees_contact”、雇员信息扩展表“employees_info_extended”。
   - 雇员信息表“employees_info”的字段为雇员编号、姓名、支付薪水币种、薪水金额、缴税税种、工作地、入职时间，其中支付薪水币种“R”代表人民币，“D”代表美元。
   - 雇员联络信息表“employees_contact”的字段为雇员编号、电话号码、e-mail。
   - 雇员信息扩展表“employees_info_extended”的字段为雇员编号、姓名、电话号码、e-mail、支付薪水币种、薪水金额、缴税税种、工作地，分区字段为入职时间。

     创建表代码实现请见**创建Hive表**。
2. 加载雇员信息数据到雇员信息表“employees_info”中。

   加载数据代码实现请见**加载数据到Hive表中**。

   雇员信息数据如**表15-8**所示：

表 15-8 雇员信息数据

编号	姓名	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
1	Wang	R	8000.01	personal income tax&0.05	Country 1:City1	2014
3	Tom	D	12000.02	personal income tax&0.09	Country 2:City2	2014
4	Jack	D	24000.03	personal income tax&0.09	Country 3:City3	2014
6	Linda	D	36000.04	personal income tax&0.09	Country 4:City4	2014
8	Zhang	R	9000.05	personal income tax&0.05	Country 5:City5	2014

3. 加载雇员联络信息数据到雇员联络信息表“employees_contact”中。
雇员联络信息数据如表15-9所示：

表 15-9 雇员联络信息数据

编号	电话号码	e-mail
1	135 XXXX XXXX	xxxx@xx.com
3	159 XXXX XXXX	xxxxx@xx.com.cn
4	186 XXXX XXXX	xxxx@xx.org
6	189 XXXX XXXX	xxxx@xxx.cn
8	134 XXXX XXXX	xxxx@xxxx.cn

4. 加载雇员扩展信息数据到雇员联络信息表“employees_info_extended”中。
雇员扩展信息数据如表15-10所示：

表 15-10 雇员扩展信息数据

编号	姓名	电话号码	e-mail	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
1	Wang	135 XXXX XXXX	xxxx@xx.com	R	8000.01	personal income tax&0.05	Country1:City1	2014
3	Tom	159 XXXX XXXX	xxxxx@xx.com.cn	D	12000.02	personal income tax&0.09	Country2:City2	2014
4	Jack	186 XXXX XXXX	xxxx@xx.org	D	24000.03	personal income tax&0.09	Country3:City3	2014
6	Linda	189 XXXX XXXX	xxxx@xxx.cn	D	36000.04	personal income tax&0.09	Country4:City4	2014
8	Zhang	134 XXXX XXXX	xxxx@xxx.cn	R	9000.05	personal income tax&0.05	Country5:City5	2014

步骤2 数据分析。

数据分析代码实现，请见**查询Hive表数据**。

- 查看薪水支付币种为美元的雇员联系方式。
- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中。
- 统计表employees_info中有多少条记录。
- 查询使用以“cn”结尾的邮箱的员工信息。

步骤3 提交数据分析任务，统计表employees_info中有多少条记录。实现请参见**使用JDBC接口提交数据分析任务**。

----结束

15.3.1.2 创建 Hive 表

功能介绍

本小节介绍了如何使用HQL创建内部表、外部表的基本操作。创建表主要有以下三种方式：

- 自定义表结构，以关键字EXTERNAL区分创建内部表和外部表。
  - 内部表，如果对数据的处理都由Hive完成，则应该使用内部表。在删除内部表时，元数据和数据一起被删除。
  - 外部表，如果数据要被多种工具共同处理，则应该使用外部表，可避免对该数据的误操作。删除外部表时，只删除掉元数据。
- 根据已有表创建新表，使用CREATE LIKE句式，完全复制原有的表结构，包括表的存储格式。
- 根据查询结果创建新表，使用CREATE AS SELECT句式。

  这种方式比较灵活，可以在复制原表表结构的同时指定要复制哪些字段，不包括表的存储格式。

说明

- 在启用了安全服务的集群中执行如下操作，需要在数据库中CREATE权限，使用CREATE AS SELECT句式创建表，需要对SELECT查询的表具有SELECT权限。详情请参见**Hive应用开发概述**。
- 目前表名长度最长为128，字段名长度最长为128，字段注解长度最长为4000，WITH SERDEPROPERTIES 中key长度最长为256，value长度最长为4000。以上的长度均表示字节长度。

样例代码

```
-- 创建外部表employees_info.
CREATE EXTERNAL TABLE IF NOT EXISTS employees_info
(
   id INT,
   name STRING,
   usd_flag STRING,
   salary DOUBLE,
   deductions MAP<STRING, DOUBLE>,
   address STRING,
   entrytime STRING
)
-- 指定行中各字段分隔符.
-- "delimited fields terminated by"指定列与列之间的分隔符为','，"MAP KEYS TERMINATED BY"指定MAP中键值的分隔符为'&'.
```

```
ROW FORMAT delimited fields terminated by ',' MAP KEYS TERMINATED BY '&'
-- 指定表的存储格式为TEXTFILE.
STORED AS TEXTFILE;

-- 创建外部表employees_contact
CREATE EXTERNAL TABLE IF NOT EXISTS employees_contact
(
   id INT,
   tel_phone STRING,
   email STRING
)
ROW FORMAT delimited fields terminated by ','
STORED AS TEXTFILE;

-- 创建外部表employees_info_extended
CREATE EXTERNAL TABLE IF NOT EXISTS employees_info_extended(id INT, name STRING, usd_flag STRING,
salary DOUBLE, deductions MAP<STRING, DOUBLE>, address STRING)
-- 一个表可以拥有一个或多个分区，每个分区以文件夹的形式单独存在表文件夹的目录下。对分区内数据进行查询，可缩小查询范围，加快数据的检索速度,还可对数据按照一定的条件进行管理。
-- 使用关键字PARTITIONED BY指定分区列名及数据类型
PARTITIONED BY(entrytime STRING)
ROW FORMAT delimited fields terminated by ',' MAP KEYS TERMINATED BY '&'
STORED AS TEXTFILE;
-- 一个表在创建完成后，还可以使用ALTER TABLE执行增/删字段、修改表属性、添加分区等操作。
-- 为表employees_info_extended增加“tel_phone”和“email”字段。
ALTER TABLE employees_info_extended ADD COLUMNS (tel_phone STRING, email STRING);
```

15.3.1.3 加载数据到 Hive 表中

功能介绍

本小节介绍了如何使用HQL向已有的表employees_info中加载数据。从本节中可以掌握如何从本地文件系统、MRS集群中加载数据。以关键字LOCAL区分数据源是否来自本地。

说明

在启用了安全服务的集群中执行如下操作，需要在数据库中具有UPDATE权限及对加载数据文件具有owner权限和读写权限，详情请参见**Hive应用开发概述**。

如果加载数据语句中有关键字LOCAL，表明从本地加载数据，除要求对相应表的UPDATE权限外，还要求该数据在当前连接的HiveServer节点上，加载用户对数据路径“PATH”具有读权限，且以omm用户能够访问该数据文件。

如果加载数据语句中有关键字OVERWRITE，表示加载的数据会覆盖表中原有的数据，否则加载的数据会追加到表中。

样例代码

```
-- 从本地文件系统/opt/hive_examples_data/目录下将employee_info.txt加载进employees_info表中.
---- 用新数据覆盖原有数据
LOAD DATA LOCAL INPATH '/opt/hive_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;
---- 保留原有数据，将新数据追加到表中
LOAD DATA LOCAL INPATH '/opt/hive_examples_data/employee_info.txt' INTO TABLE employees_info;

-- 从HDFS上/user/hive_examples_data/employee_info.txt加载进employees_info表中.
---- 用新数据覆盖原有数据
LOAD DATA INPATH '/user/hive_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;
---- 保留原有数据，将新数据追加到表中
LOAD DATA INPATH '/user/hive_examples_data/employee_info.txt' INTO TABLE employees_info;
```

说明

加载数据的实质是将数据复制到HDFS上指定表的目录下。

样例数据

表**employees_info**的数据如下：

```
1,Wang,R,8000.01,person&personal^Btype&income^Btax&0.05,Country1:City1,2014
3,Tom,D,12000.02,person&personal^Btype&income^Btax&0.09,Country2:City2,2014
4,Jack,D,24000.03,person&personal^Btype&income^Btax&0.05,Country3:City3,2014
6,Linda,D,36000.04,person&personal^Btype&income^Btax&0.05,Country4:City4,2014
8,Zhang,R,9000.05,person&personal^Btype&income^Btax&0.05,Country5:City5,2014
```

表**employees_contact**的数据如下：

```
1,135 XXXX XXXX,xxxx@xx.com
3,159 XXXX XXXX,xxxxx@xx.com.cn
4,186 XXXX XXXX,xxxx@xx.org
6,189 XXXX XXXX,xxxx@xxx.cn
8,134 XXXX XXXX,xxxx@xxxx.cn
```

表**employees_info_extended**的数据如下：

```
1,Wang,135 XXXX
XXXX,xxxx@xx.com,R,8000.01,person&personal^Btype&income^Btax&0.05,Country1:City1,2014
3,Tom,159 XXXX
XXXX,xxxxx@xx.com.cn,D,12000.02,person&personal^Btype&income^Btax&0.09,Country2:City2,2014
4,Jack,186 XXXX
XXXX,xxxx@xx.org,D,24000.03,person&personal^Btype&income^Btax&0.05,Country3:City3,2014
6,Linda,189 XXXX
XXXX,xxxx@xxx.cn,D,36000.04,person&personal^Btype&income^Btax&0.05,Country4:City4,2014
8,Zhang,134 XXXX
XXXX,xxxx@xxxx.cn,R,9000.05,person&personal^Btype&income^Btax&0.05,Country5:City5,2014
```

15.3.1.4 查询 Hive 表数据

功能介绍

本小节介绍了如何使用HQL对数据进行查询分析。从本节中可以掌握如下查询分析方法：

- SELECT查询的常用特性，如JOIN等。
- 加载数据进指定分区。
- 如何使用Hive自带函数。
- 如何使用自定义函数进行查询分析，如何创建、定义自定义函数请见**创建Hive用户自定义函数**。

说明

在启用了安全服务的集群中执行如下操作，需要对涉及的表具有与操作对应的权限。详情请参见**Hive应用开发概述**。

样例代码

```
-- 查看薪水支付币种为美元的雇员联系方式.
SELECT
a.name,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b  ON(a.id = b.id) WHERE usd_flag='D';

-- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中.
INSERT OVERWRITE TABLE employees_info_extended PARTITION (entrytime = '2014')
SELECT
a.id,
```

```
a.name,
a.usd_flag,
a.salary,
a.deductions,
a.address,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE a.entrytime = '2014';

-- 使用Hive中已有的函数COUNT()，统计表employees_info中有多少条记录.
SELECT COUNT(*) FROM employees_info;

-- 查询使用以“cn”结尾的邮箱的员工信息.
SELECT a.name, b.tel_phone FROM  employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE
b.email like '%cn';
```

扩展使用

- 配置Hive中间过程的数据加密

  指定表的格式为RCFile(推荐使用)或SequenceFile，加密算法为ARC4Codec。SequenceFile是Hadoop特有的文件格式，RCFile是Hive优化的文件格式。RCFile优化了列存储，在对大表进行查询时，综合性能表现比SequenceFile更优。

  ```
  set hive.exec.compress.output=true;
  set hive.exec.compress.intermediate=true;
  set hive.intermediate.compression.codec=org.apache.hadoop.io.encryption.arc4.ARC4Codec;
  ```

- 自定义函数，具体内容请参见**创建Hive用户自定义函数**。

15.3.1.5 实现 Hive 进程访问多 ZooKeeper

功能简介

FusionInsight支持在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper，分别通过“testConnectHive”和“testConnectApacheZK”方法实现。

在“hive-jdbc-example-multizk”包中的“JDBCExample”类中，main方法的代码结构如下：

```
public static void main(String[] args) throws InstantiationException,IllegalAccessException,
ClassNotFoundException, SQLException, IOException{
    testConnectHive();//访问FusionInsight ZooKeeper的方法
    testConnectApacheZk();//访问开源ZooKeeper的方法
}
```

访问 FusionInsight ZooKeeper

如果仅需运行访问FusionInsight Zookeeper方法，需注释掉main函数中的“testConnectApacheZk”方法。

使用“testConnectHive”方法访问FusionInsight ZooKeeper前需执行如下操作：

步骤1 修改JDBCExample中“init”方法中的“USER_NAME”参数的值。“USER_NAME”对应的用户用于访问FusionInsight ZooKeeper，需拥有FusionInsight Hive、Hadoop普通用户组权限。

步骤2 进入客户端解压路径“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles\Hive\config”，手动将“core-site.xml”、“hiveclient.properties”文件放到样例工程的“hive-jdbc-example-multizk\src\main\resources”目录下。

步骤3 下载导入该用户的krb5.conf和user.keytab文件到hive-jdbc-example-multizk包中的resources目录下。

步骤4 检查并修改resources目录下hiveclient.properties文件中“zk.port”和“zk.quorum”参数的值：

- zk.port：为访问FusionInsight ZooKeeper的端口，通常保持默认，根据实际使用情况修改。
- zk.quorum：为访问ZooKeeper quorumpeer的地址，请修改为集群部署有FusionInsight ZooKeeper服务的IP地址。

----结束

访问开源 ZooKeeper

使用“testConnectApacheZk”连接开源ZooKeeper的代码，只需要将以下代码中的“xxx.xxx.xxx.xxx”修改为需要连接的开源的ZooKeeper的IP，端口号按照实际情况修改。如果仅需运行访问第三方Zookeeper的样例，需注释掉main函数中的“testConnectHive”方法。

```
digestZK = new org.apache.zookeeper.ZooKeeper("xxx.xxx.xxx.xxx:端口号", 60000, null);
```

说明

ZooKeeper连接使用完后需要关闭连接，否则可能导致连接泄露。可根据业务实际情况进行处理，代码如下：

```
//使用try-with-resources方式，try语句执行完后会自动关闭ZooKeeper连接。
try (org.apache.zookeeper.ZooKeeper digestZk =
    new org.apache.zookeeper.ZooKeeper("xxx.xxx.xxx.xxx:端口号", 600000, null)) {
    ...
}
```

15.3.1.6 使用 JDBC 接口提交数据分析任务

功能介绍

本章节介绍如何使用JDBC样例程序完成数据分析任务。

样例代码

使用Hive JDBC接口提交数据分析任务，该样例程序在“hive-examples/hive-jdbc-example”的“JDBCExample.java”中，实现该功能的模块如下：

1. 读取HiveServer客户端property文件，其中“hiveclient.properties”文件在“hive-jdbc-example/src/main/resources”目录下。

```
Properties clientInfo = null;
String userdir = System.getProperty("user.dir") + File.separator
+ "conf" + File.separator;
InputStream fileInputStream = null;
try{
clientInfo = new Properties();
//"hiveclient.properties"为客户端配置文件
//"hiveclient.properties"文件可从对应实例客户端安装包解压目录下的config目录下获取，并上传到JDBC样例工程的“hive-jdbc-example/src/main/resources”目录下
String hiveclientProp = userdir + "hiveclient.properties" ;
File propertiesFile = new File(hiveclientProp);
fileInputStream = new FileInputStream(propertiesFile);
clientInfo.load(fileInputStream);
}catch (Exception e) {
throw new IOException(e);
}finally{
if(fileInputStream != null){
fileInputStream.close();
fileInputStream = null;
```

```
}
}
```

2. 获取ZooKeeper的IP列表和端口、集群的认证模式、HiveServer的SASL配置、HiveServer在ZooKeeper中节点名称、客户端对服务端的发现模式、以及服务端进程认证的principal。这些配置样例代码会自动从“hiveclient.properties中”读取。

```
//zkQuorum获取后的格式为"xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181";
//"xxx.xxx.xxx.xxx"为集群中ZooKeeper所在节点的业务IP，端口默认是2181
   zkQuorum = clientInfo.getProperty("zk.quorum");
   auth = clientInfo.getProperty("auth");
   sasl_qop = clientInfo.getProperty("sasl.qop");
   zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
   serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
   principal = clientInfo.getProperty("principal");
```

3. 在安全模式下，需要Kerberos用户以及keytab文件路径等信息进行登录认证。

```
   // 设置新建用户的userName，其中"xxx"指代之前创建的用户名，例如创建的用户为developuser，则USER_NAME为developuser
   USER_NAME = "xxx";
   // 设置客户端的keytab和krb5文件路径，即“hive-jdbc-example\src\main\resources”
   String userdir = System.getProperty("user.dir") + File.separator
         + "conf" + File.separator;
   USER_KEYTAB_FILE = userdir + "user.keytab";
   KRB5_FILE = userdir + "krb5.conf";
```

4. 定义HQL。HQL必须为单条语句，注意HQL不能包含“;”。

```
  // 定义HQL，不能包含“;”
  String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
         "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};
```

5. 拼接JDBC URL。

   说明

   拼接JDBC URL也可以不提供账户和keytab路径，采用提前认证的方式。如果使用IBM JDK运行Hive应用程序，则必须使用“JDBC代码样例二”提供的预认证方式才能访问。

   以下代码片段，拼接完成后的JDBC URL示例为：

   ```
   jdbc:hive2://
   xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2;sasl.qop=auth-conf;auth=KERBEROS;principal=hive/hadoop.<系统域名>@<系统域名>;
   ```

   系统域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数获取。

```
   // 拼接JDBC URL
   StringBuilder sBuilder = new StringBuilder(
       "jdbc:hive2://").append(zkQuorum).append("/");

   if ("KERBEROS".equalsIgnoreCase(auth)) {
    sBuilder.append(";serviceDiscoveryMode=")
          .append(serviceDiscoveryMode)
          .append(";zooKeeperNamespace=")
          .append(zooKeeperNamespace)
          .append(";sasl.qop=")
          .append(sasl_qop)
          .append(";auth=")
          .append(auth)
          .append(";principal=")
          .append(principal)
                .append(";user.principal=")
          .append(USER_NAME)
          .append(";user.keytab=")
          .append(USER_KEYTAB_FILE)
          .append(";");

   }
```

```
  String url = sBuilder.toString();
```

6. 加载Hive JDBC驱动。

```
// 加载Hive JDBC驱动
Class.forName(HIVE_DRIVER);
```

7. 获取JDBC连接，确认HQL的类型（DDL/DML），调用对应的接口执行HQL，输出查询的列名和结果到控制台，关闭JDBC连接。

```
Connection connection = null;
  try {
    // 获取JDBC连接

    connection = DriverManager.getConnection(url, "", "");

    // 建表
    // 表建完之后，如果要往表中导数据，可以使用LOAD语句将数据导入表中，比如从HDFS上将数据导入表:
    //load data inpath '/tmp/employees.txt' overwrite into table employees_info;
    execDDL(connection,sqls[0]);
    System.out.println("Create table success!");

    // 查询
    execDML(connection,sqls[1]);

    // 删表
    execDDL(connection,sqls[2]);
    System.out.println("Delete table success!");
  }
  finally {
    // 关闭JDBC连接
    if (null != connection) {
      connection.close();
    }

public static void execDDL(Connection connection, String sql)
  throws SQLException {
  PreparedStatement statement = null;
  try {
    statement = connection.prepareStatement(sql);
    statement.execute();
  }
  finally {
    if (null != statement) {
      statement.close();
    }
  }
}

public static void execDML(Connection connection, String sql) throws SQLException {
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  ResultSetMetaData resultMetaData = null;

  try {
    // 执行HQL
    statement = connection.prepareStatement(sql);
    resultSet = statement.executeQuery();

    // 输出查询的列名到控制台
    resultMetaData = resultSet.getMetaData();
    int columnCount = resultMetaData.getColumnCount();
    for (int i = 1; i <= columnCount; i++) {
      System.out.print(resultMetaData.getColumnLabel(i) + '\t');
    }
    System.out.println();

    // 输出查询结果到控制台
    while (resultSet.next()) {
      for (int i = 1; i <= columnCount; i++) {
```

```
          System.out.print(resultSet.getString(i) + '\t');
        }
        System.out.println();
      }
    }
    finally {
      if (null != resultSet) {
        resultSet.close();
      }

      if (null != statement) {
        statement.close();
      }
    }
  }
```

15.3.2 HCatalog 访问 Hive 样例程序

功能介绍

本章节介绍如何在MapReduce任务中使用HCatalog分析Hive表数据，读取输入表第一列**int**类型数据执行**count(distinct XX)**操作，将结果写入输出表。

样例代码

该样例程序在“hive-examples/hcatalog-example”的“HCatalogExample.java”中，实现该功能的模块如下：

1. 实现Mapper类，通过HCatRecord获取第一列int类型数据，计数1并输出；

```
public static class Map extends
        Mapper<LongWritable, HCatRecord, IntWritable, IntWritable> {
    int age;
    @Override
    protected void map(
            LongWritable key,
            HCatRecord value,
            Mapper<LongWritable, HCatRecord,
                    IntWritable, IntWritable>.Context context)
            throws IOException, InterruptedException {
        if ( value.get(0) instanceof Integer ) {
            age = (Integer) value.get(0);
        }
        context.write(new IntWritable(age), new IntWritable(1));
    }
}
```

2. 实现Reducer类，将map输出结果合并计数，统计不重复的值出现次数，使用HCatRecord输出结果；

```
public static class Reduce extends Reducer<IntWritable, IntWritable,
        IntWritable, HCatRecord> {
    @Override
    protected void reduce(
            IntWritable key,
            Iterable<IntWritable> values,
            Reducer<IntWritable, IntWritable,
                    IntWritable, HCatRecord>.Context context)
            throws IOException, InterruptedException {
        int sum = 0;
        Iterator<IntWritable> iter = values.iterator();
        while (iter.hasNext()) {
            sum++;
            iter.next();
        }
        HCatRecord record = new DefaultHCatRecord(2);
        record.set(0, key.get());
```

```
        record.set(1, sum);
        context.write(null, record);
    }
}
```

3. MapReduce任务定义，指定输入/输出类，Mapper/Recducer类，输入输出键值对格式；

```
Job job = new Job(conf, "GroupByDemo");
HCatInputFormat.setInput(job, dbName, inputTableName);
job.setInputFormatClass(HCatInputFormat.class);
job.setJarByClass(HCatalogExample.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setMapOutputKeyClass(IntWritable.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(WritableComparable.class);
job.setOutputValueClass(DefaultHCatRecord.class);
String outputTableName = otherArgs[1];
OutputJobInfo outputjobInfo = OutputJobInfo.create(dbName, outputTableName, null);
HCatOutputFormat.setOutput(job, outputjobInfo);
HCatSchema schema = outputjobInfo.getOutputSchema();
HCatOutputFormat.setSchema(job, schema);
job.setOutputFormatClass(HCatOutputFormat.class);
```

15.3.3 基于 Python 的 Hive 样例程序

功能介绍

本章节介绍如何使用Python连接Hive执行数据分析任务。

样例代码

使用Python方式提交数据分析任务，参考样例程序中的“hive-examples/python-examples/pyCLI_sec.py”。该样例程序连接的集群的认证模式是安全模式，运行样例程序之前需要使用**kinit**命令认证相应权限的Kerberos用户。

1. 导入HAConnection类。

```
from pyhs2.haconnection import HAConnection
```

2. 声明HiveServer的IP地址列表。本例中hosts代表HiveServer的节点，xxx.xxx.xxx.xxx代表业务IP地址。

```
hosts = ["xxx.xxx.xxx.xxx", "xxx.xxx.xxx.xxx"]
```

> **说明**
>
> 如果HiveServer实例被迁移，原始的示例程序会失效。在HiveServer实例迁移之后，用户需要更新示例程序中使用的HiveServer的IP地址。

3. 配置Kerberos主机名和服务名。本例中“krb_host”参数值为“hadoop.*实际域名*”，实际域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信 > 本端域” 查看；主机名为hadoop，服务名为hive。

```
conf = {"krb_host":"hadoop.<系统域名>", "krb_service":"hive"}
```

4. 创建连接，执行HQL，样例代码中仅执行查询所有表功能，可根据实际情况修改HQL内容，输出查询的列名和结果到控制台。

```
try:
    with HAConnection(hosts = hosts,
                      port = 21066,
                      authMechanism = "KERBEROS",
                      configuration = conf) as haConn:
        with haConn.getConnection() as conn:
            with conn.cursor() as cur:
                # show databases
                print cur.getdatabases()
```

```
            # execute query
            cur.execute("show tables")

            # return column info from query
            print cur.getschema()

            # fetch table results
            for i in cur.fetch():
                print i

except exception, e:
    print e
```

15.3.4 基于 Python3 的 Hive 样例程序

功能介绍

本章节介绍如何使用Python3连接Hive执行数据分析任务。

样例代码

安全模式连接Hive前需要使用集群客户端进行认证，使用**kinit**命令认证相应权限的Kerberos用户，认证后执行分析任务示例在“hive-examples/python3-examples/pyCLI_sec.py”文件中。

1. 导入hive类
   ```
   from pyhive import hive
   ```
2. 创建JDBC连接。
   ```
   connection = hive.Connection(host='hiveserverIp', port=hiveserverPort, username='hive',
   database='default', auth='KERBEROS', kerberos_service_name="hive", krbhost='hadoop.hadoop.com')
   ```
   需按照实际环境修改以下参数：
   - hiveserverIp：替换为实际需要连接的HiveServer节点IP地址，可登录FusionInsight Manager，选择“集群 > 服务 > Hive > 实例”查看。
   - hiveserverPort：需要替换为Hive服务的端口，可在FusionInsight Manager界面，选择“集群 > 服务 > Hive > 配置”，在搜索框中搜索“hive.server2.thrift.port”查看，默认值为“10000”。
   - username：参数值为实际使用的用户名，即**准备集群认证用户信息**创建的用户名。
   - kerberos_service_name：参数值为实际连接的实例，以连接Hive为例，修改为：kerberos_service_name="hive"。
   - krbhost：参数值为“hadoop.*实际域名*”，实际域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信 > 本端域” 查看。
3. 执行SQL语句，样例代码中仅执行查询所有表功能，可根据实际情况修改HQL内容。
   ```
   cursor = connection.cursor()
   cursor.execute('show tables')
   ```
4. 获取结果并输出
   ```
   for result in cursor.fetchall():
       print(result)
   ```

15.4 调测 Hive 应用

15.4.1 在本地 Windows 环境中调测 Hive JDBC 样例程序

在程序代码完成开发后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

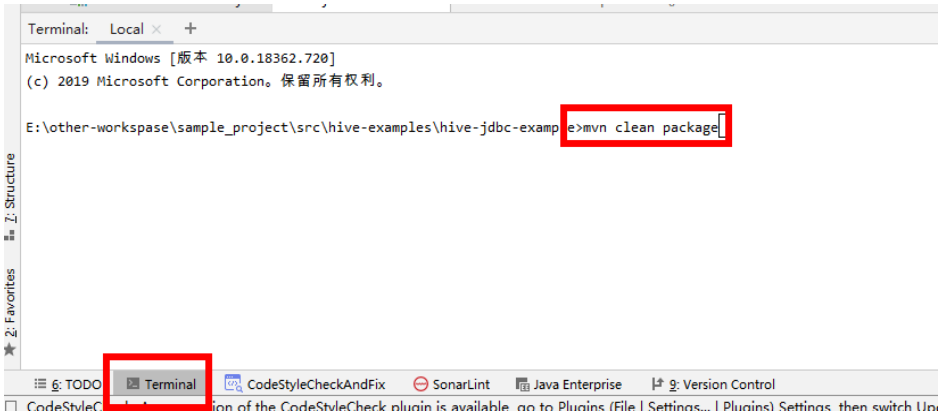
说明

- 如果Windows开发环境中使用IBM JDK，不支持在Windows环境中直接运行应用程序。
- 需要在运行样例代码的本机**hosts**文件中设置访问节点的主机名和公网IP地址映射，主机名和公网IP地址请保持一一对应。
- 仅JDBC样例程序支持在本地Windows中运行。

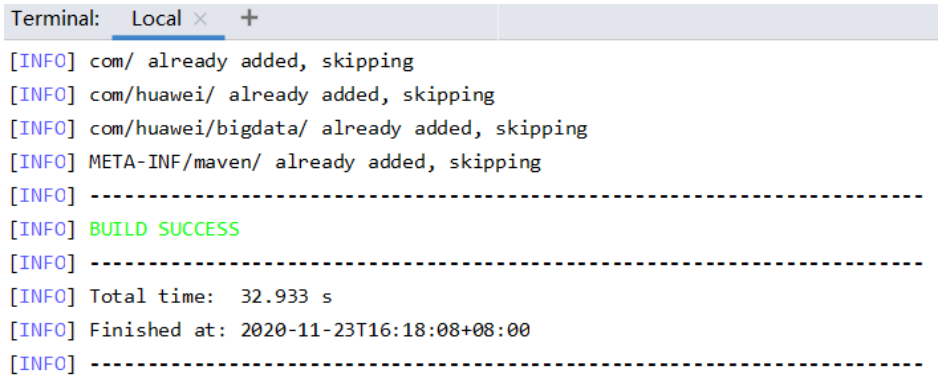
操作步骤

步骤1 编译JDBC样例程序：

在IDEA界面左下方单击“Terminal”进入终端，执行命令**mvn clean package**进行编译。



当输出“BUILD SUCCESS”，表示编译成功，如下图所示。编译成功后将会在样例工程的target下生成含有“-with-dependencies”字段的jar包。



步骤2 运行JDBC样例程序：

- 使用Windows命令行形式运行JDBC样例工程：
  a. 在Windows上创建一个目录作为运行目录，如“D:\jdbc_example”，将步骤1中生成的“target”目录下包名中含有“-with-dependencies”字段的Jar包放到该路径下，并在该目录下创建子目录“src/main/resources”，将已获取

的“hive-jdbc-example\src\main\resources”目录下的所有文件复制到“resources”下。

b. 执行以下命令运行Jar包：

cd /d d:\jdbc_example

java -jar *hive-jdbc-example-1.0-SNAPSHOT-jar-with-dependencies.jar*

说明

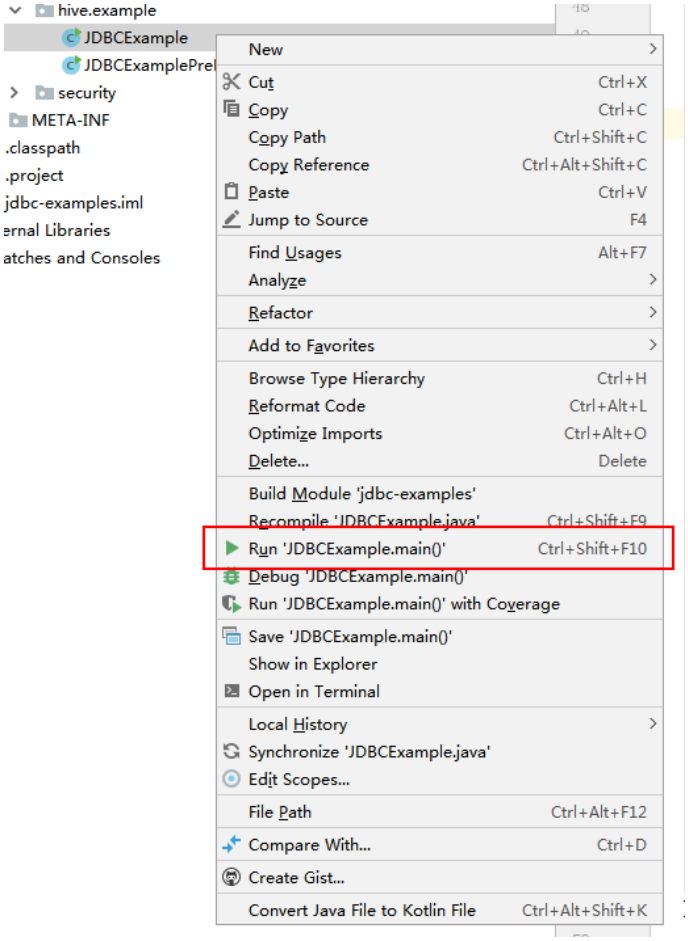
以上Jar包名称仅供参考，具体名称以实际生成为主。

c. 在命令行终端查看样例代码中的HQL所查询出的结果，运行成功结果会有如下信息：

```
Create table success!
_c0
0
Delete table success!
```

- 使用IntelliJ IDEA形式运行JDBC样例工程：

a. 在IntelliJ IDEA的jdbc-examples工程的“JDBCExample”类上单击右键，在弹出菜单中选择“Run JDBCExample.main()”，如下图所示：



b. 在IntelliJ IDEA输出窗口查看样例代码中的HQL所查询出的结果，会有如下信息：

```
Create table success!
_c0
0
Delete table success!
```

----结束

15.4.2 在 Linux 环境中调测 Hive JDBC 样例程序

Hive JDBC应用程序支持在安装Hive客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。

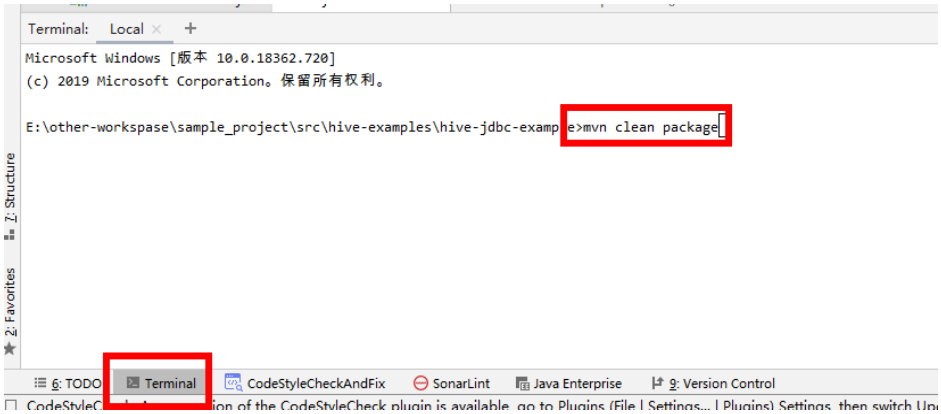
前提条件

- 已安装Hive客户端。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

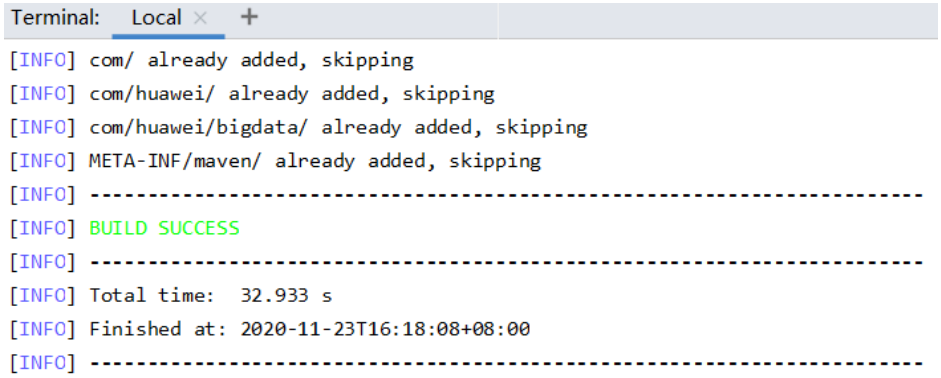
操作步骤

步骤1 编译JDBC样例程序：

在IDEA界面左下方单击“Terminal”进入终端，执行命令**mvn clean package**进行编译。



当输出“BUILD SUCCESS”，表示编译成功，如下图所示。编译成功后将会在样例工程的target下生成含有“-with-dependencies”字段的jar包。



步骤2 运行JDBC样例程序：

1. 在Linux上创建一个目录作为运行目录，如“/opt/jdbc_example”，将**步骤1**中生成的“target”目录下包名中含有“-with-dependencies”字段的Jar包放进该路径下，并在该目录下创建子目录“src/main/resources”，将已获取的“hive-jdbc-example\src\main\resources”目录下的所有文件复制到“resources”下。
2. 执行以下命令运行Jar包：

chmod +x /opt/jdbc_example -R

cd /opt/jdbc_example

java -jar *hive-jdbc-example-1.0-SNAPSHOT-jar-with-dependencies.jar*

说明

以上Jar包名称仅供参考，具体名称以实际生成为主。

3. 在命令行终端查看样例代码中的HQL所查询出的结果，运行成功会显示如下信息：

```
Create table success!
_c0
0
Delete table success!
```

----结束

15.4.3 调测 Hive HCatalog 样例程序

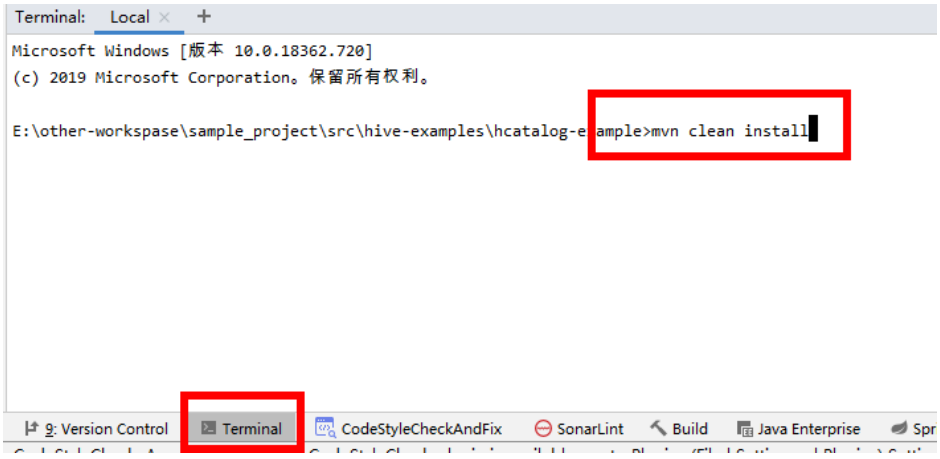
Hive HCatalog应用程序支持在安装Hive和Yarn客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。

前提条件

- 已安装Hive和Yarn客户端。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

操作步骤

步骤1 在IDEA界面左下方单击“Terminal”进入终端，执行命令**mvn clean install**进行编译。



当输出“BUILD SUCCESS”，表示编译成功，如下图所示。编译成功后将会在样例工程的target下生成“hcatalog-example-*.jar”包。

```
Terminal:  Local  +
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hcatalog-example ---
[INFO] Building jar: E:\other-workspase\sample_project\src\hive-examples\hca
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ hcatalog-exa
[INFO] Installing E:\other-workspase\sample_project\src\hive-examples\hcata
[INFO] Installing E:\other-workspase\sample_project\src\hive-examples\hcata
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  4.916 s
[INFO] Finished at: 2020-11-23T16:25:57+08:00
[INFO] ------------------------------------------------------------------------
```

说明

以上Jar包名称仅供参考，具体名称以实际生成为主。

步骤2 将步骤1中在“target”目录下生成的“hcatalog-example-*.jar”上传至Linux的指定路径，例如“/opt/hive_client”，记作“$HCAT_CLIENT”，并确保已经安装好Hive和Yarn客户端。运行环境变量使“HCAT_CLIENT”生效。

```
export HCAT_CLIENT=/opt/hive_client
```

步骤3 执行以下命令用于配置环境变量信息（以客户端安装路径为“/opt/client”为例）：

```
export HADOOP_HOME=/opt/client/HDFS/hadoop
export HIVE_HOME=/opt/client/Hive/Beeline
export HCAT_HOME=$HIVE_HOME/../HCatalog
export LIB_JARS=$HCAT_HOME/lib/hive-hcatalog-core-xxx.jar,$HCAT_HOME/lib/hive-metastore-xxx.jar,$HCAT_HOME/lib/hive-standalone-metastore-xxx.jar,$HIVE_HOME/lib/hive-exec-xxx.jar,$HCAT_HOME/lib/libfb303-xxx.jar,$HCAT_HOME/lib/slf4j-api-xxx.jar,$HCAT_HOME/lib/jdo-api-xxx.jar,$HCAT_HOME/lib/antlr-runtime-xxx.jar,$HCAT_HOME/lib/datanucleus-api-jdo-xxx.jar,$HCAT_HOME/lib/datanucleus-core-xxx.jar,$HCAT_HOME/lib/datanucleus-rdbms-fi-xxx.jar,$HCAT_HOME/lib/log4j-api-xxx.jar,$HCAT_HOME/lib/log4j-core-xxx.jar,$HIVE_HOME/lib/commons-lang-xxx.jar
export HADOOP_CLASSPATH=$HCAT_HOME/lib/hive-hcatalog-core-xxx.jar:$HCAT_HOME/lib/hive-metastore-xxx.jar:$HCAT_HOME/lib/hive-standalone-metastore-xxx.jar:$HIVE_HOME/lib/hive-exec-xxx.jar:$HCAT_HOME/lib/libfb303-xxx.jar:$HADOOP_HOME/etc/hadoop:$HCAT_HOME/conf:$HCAT_HOME/lib/slf4j-api-xxx.jar:$HCAT_HOME/lib/jdo-api-xxx.jar:$HCAT_HOME/lib/antlr-runtime-xxx.jar:$HCAT_HOME/lib/datanucleus-api-jdo-xxx.jar:$HCAT_HOME/lib/datanucleus-core-xxx.jar:$HCAT_HOME/lib/datanucleus-rdbms-fi-xxx.jar:$HCAT_HOME/lib/log4j-api-xxx.jar:$HCAT_HOME/lib/log4j-core-xxx.jar:$HIVE_HOME/lib/commons-lang-xxx.jar
```

说明

xxx：表示Jar包的版本号。“LIB_JARS”和“HADOOP_CLASSPATH”中指定的Jar包的版本号需要根据实际环境的版本号进行修改。

步骤4 运行前准备：

1. 使用Hive客户端，在beeline中执行以下命令创建源表t1：

   create table t1(col1 int);

   向t1中插入如下数据：

```
+----------+--+
| t1.col1  |
+----------+--+
| 1        |
| 1        |
| 1        |
| 2        |
```



```
| 2       |
| 3       |
```

2. 执行以下命令创建目的表t2：

   create table t2(col1 int,col2 int);

   说明

   本样例工程中创建的表使用Hive默认的存储格式，暂不支持指定存储格式为ORC的表。

步骤5 使用Yarn客户端提交任务。

yarn --config $HADOOP_HOME/etc/hadoop jar $HCAT_CLIENT/hcatalog-example-1.0-SNAPSHOT.jar com.huawei.bigdata.HCatalogExample -libjars $LIB_JARS t1 t2

步骤6 运行结果查看，运行后t2表数据如下所示：

```
0: jdbc:hive2://192.168.1.18:2181,192.168.1.> select * from t2;
+----------+----------+--+
| t2.col1  | t2.col2  |
+----------+----------+--+
| 1        | 3        |
| 2        | 2        |
| 3        | 1        |
+----------+----------+--+
```

----结束

15.4.4 调测 Hive Python 样例程序

Python 样例工程的命令行形式运行

步骤1 赋予“python-examples”文件夹中脚本的可执行权限。在命令行终端执行以下命令：

chmod +x python-examples -R。

步骤2 在“python-examples/pyCLI_sec.py”中的hosts数组中填写安装HiveServer的节点的业务平面IP地址。HiveServer业务平面IP地址可登录FusionInsight Manager，选择“集群 > 服务 > Hive > 实例”查看。

步骤3 将“python-examples/pyCLI_sec.py”和“python-examples/pyline.py”的conf数组中的“hadoop.hadoop.com”修改为hadoop.*实际域名*。实际域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信 > 本端域” 查看。

步骤4 先使用**kinit**命令获取Kerberos认证的缓存。

使用**准备集群认证用户信息**中创建的开发用户执行以下命令运行客户端程序：

kinit -kt *keytab的存储路径 username*

cd python-examples

python pyCLI_sec.py

步骤5 在命令行终端查看样例代码中的HQL所查询出的结果。例如：

```
[['default', '']]
 [{'comment': 'from deserializer', 'columnName': 'tab_name', 'type': 'STRING_TYPE'}]
 ['xx']
```

说明

如果出现如下异常：

```
importError: libsasl2.so.2: cannot open shared object file: No such file or directory
```

请按照以下方式处理：

1. 首先执行如下命令，查询所装操作系统中LibSASL的版本
   ```
   ldconfig -p|grep sasl
   ```
   结果如下则表示当前操作系统仅存在3.x版本
   ```
   libsasl2.so.3 (libc6,x86-64) => /usr/lib64/libsasl2.so.3
   libsasl2.so.3 (libc6) => /usr/lib/libsasl2.so.3
   ```
2. 如果仅存在3.x版本，需要执行如下命令创建软链接
   ```
   ln -s /usr/lib64/libsasl2.so.3.0.0 /usr/lib64/libsasl2.so.2
   ```

----结束

15.4.5 调测 Hive Python3 样例程序

Python3 样例工程的命令行形式运行

步骤1 赋予“python3-examples”文件夹中脚本的可执行权限。在命令行终端执行以下命令：

chmod +x python3-examples -R。

步骤2 将“python3-examples/pyCLI_sec.py”中的host的值修改为安装HiveServer的节点的业务平面IP，port的值修改为Hive提供Thrift服务的端口。

说明

- HiveServer业务平面IP地址可登录FusionInsight Manager，选择“集群 > 服务 > Hive > 实例”查看。
- port可在FusionInsight Manager界面，选择“集群 > 服务 > Hive > 配置”，在搜索框中搜索“hive.server2.thrift.port”查看，默认值为“10000”。

步骤3 在“python3-examples/pyCLI_sec.py”中，修改“hadoop.hadoop.com”为“hadoop.*实际域名*”。实际域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信 > 本端域” 查看。

步骤4 先使用**kinit**命令获取kerberos认证的缓存。

使用**准备集群认证用户信息**中创建的开发用户执行以下命令运行客户端程序：

kinit -kt *keytab的存储路径 username*

cd python3-examples

python3 pyCLI_sec.py

步骤5 在命令行终端查看样例代码中的HQL所查询出的结果。例如：

```
('table_name1',)
('table_name2',)
('table_name3',)
('table_name4',)
('table_name5',)
```

其中，“table_name*X*”表示实际的表名。

----结束

15.5 Hive 应用开发常见问题

15.5.1 Hive 对外接口介绍

15.5.1.1 Hive JDBC 接口介绍

Hive JDBC接口遵循标准的JAVA JDBC驱动标准。

说明

Hive作为数据仓库类型数据库，其并不能支持所有的JDBC标准API。例如事务类型的操作：rollback、setAutoCommit等，执行该类操作会获得“Method not supported”的SQLException异常。

15.5.1.2 Hive WebHCat 接口介绍

说明

- 以下示例的IP为WebHCat的业务IP，端口为安装时设置的WebHCat HTTP端口。
- 需要在安装客户端的机器上进行kinit认证操作后才可执行示例操作。
- 以下示例均为https协议的示例，若要使用http协议，需要执行以下操作：
  1. 将REST接口切换成HTTP协议方式，请参见**配置基于HTTPS/HTTP协议的REST接口**。
  2. 将示例中的“--insecure”去掉，将https替换成http，例如
     curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/status'
     更改为
     curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/status'
- 执行操作前需确保使用的curl版本在7.34.0以上。
  可以使用以下命令查看curl版本：
  curl -V

1. :version(GET)
   - 描述
     查询WebHCat支持的返回类型列表。
   - URL
     https://www.myserver.com/templeton/:version
   - 参数

参数	描述
:version	WebHCat版本号（当前必须是v1）。

   - 返回结果

参数	描述
responseTypes	WebHCat支持的返回类型列表。

- 例子
  ```
  curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1'
  ```

2. status（GET）
   - 描述
     获取当前服务器的状态
   - URL
     https://www.myserver.com/templeton/v1/status
   - 参数
     无
   - 返回结果

参数	描述
status	WebHCat连接正常，返回OK。
version	字符串，包含版本号，比如v1。

   - 例子
     ```
     curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/status'
     ```
3. version（GET）
   - 描述
     获取服务器WebHCat的版本
   - URL
     https://www.myserver.com/templeton/v1/version
   - 参数
     无
   - 返回结果

参数	描述
supportedVersions	所有支持的版本
version	当前服务器WebHCat的版本

   - 例子
     ```
     curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/version'
     ```
4. version/hive（GET）
   - 描述
     获取服务器Hive的版本
   - URL
     https://www.myserver.com/templeton/v1/version/hive
   - 参数
     无
   - 返回结果

参数	描述
module	hive
version	Hive的版本

- 例子
  ```
  curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/version/hive'
  ```

5. version/hadoop (GET)
   - 描述

     获取服务器Hadoop的版本
   - URL

     https://www.myserver.com/templeton/v1/version/hadoop
   - 参数

     无
   - 返回结果

参数	描述
module	hadoop
version	Hadoop的版本

   - 例子
     ```
     curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/version/hadoop'
     ```

6. ddl (POST)
   - 描述

     执行DDL语句
   - URL

     https://www.myserver.com/templeton/v1/ddl
   - 参数

参数	描述
exec	需要执行的HCatalog DDL语句。
group	当DDL是创建表时，创建表使用的用户组。
permissions	当DDL是创建表时，创建表使用的权限，格式为rwxr-xr-x。

   - 返回结果

参数	描述
stdout	HCatalog执行时的标准输出值，可能为空。

参数	描述
stderr	HCatalog执行时的错误输出，可能为空。
exitcode	HCatalog的返回值。

- 例子
  ```
  curl -i -u : --insecure --negotiate -d exec="show tables" 'https://10.64.35.144:9111/templeton/v1/ddl'
  ```

7. ddl/database (GET)
   - 描述

     列出所有的数据库
   - URL

     https://www.myserver.com/templeton/v1/ddl/database
   - 参数

参数	描述
like	用来匹配数据库名的正则表达式。

   - 返回结果

参数	描述
databases	数据库名

   - 例子
     ```
     curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/ddl/database'
     ```
8. ddl/database/:db (GET)
   - 描述

     获取指定数据库的详细信息
   - URL

     https://www.myserver.com/templeton/v1/ddl/database/:db
   - 参数

参数	描述
:db	数据库名

   - 返回结果

参数	描述
location	数据库位置
comment	数据库的备注，如果没有备注则不存在

参数	描述
database	数据库名
owner	数据库的所有者
owertype	数据库所有者的类型

- 例子
```
curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/ddl/database/default'
```

9. ddl/database/:db (PUT)
   - 描述

     创建数据库
   - URL

     https://www.myserver.com/templeton/v1/ddl/database/:db
   - 参数

参数	描述
:db	数据库名
group	创建数据库时使用的用户组
permission	创建数据库时使用的权限
location	数据库的位置
comment	数据库的备注，比如描述
properties	数据库属性

   - 返回结果

参数	描述
database	新创建的数据库的名字

   - 例子
```
curl -i -u : --insecure --negotiate -X PUT -HContent-type:application/json -d '{"location": "/tmp/a", "comment": "my db", "properties": {"a": "b"}}' 'https://10.64.35.144:9111/templeton/v1/ddl/database/db2'
```

10. ddl/database/:db (DELETE)
    - 描述

      删除数据库
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db
    - 参数

参数	描述
:db	数据库名

参数	描述
ifExists	如果指定数据库不存在，Hive会返回错误，除非设置了ifExists为true。
option	将参数设置成cascade或者restrict。如果选择cascade，将清除一切，包括数据和定义。如果选择restrict，表格内容为空，模式也将不存在。

- 返回结果

参数	描述
database	删除的数据库名字

- 例子

```
curl -i -u : --insecure --negotiate -X DELETE 'https://10.64.35.144:9111/templeton/v1/ddl/database/db3?ifExists=true'
```

11. ddl/database/:db/table (GET)
    - 描述

      列出数据库下的所有表
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table
    - 参数

参数	描述
:db	数据库名
like	用来匹配表名的正则表达式

    - 返回结果

参数	描述
database	数据库名字
tables	数据库下表名列表

    - 例子

```
curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/ddl/database/default/table'
```

12. ddl/database/:db/table/:table (GET)
    - 描述

      获取表的详细信息
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table

- 参数

参数	描述
:db	数据库名
:table	表名
format	格式：format=extended，参考额外信息（“table extended like”）。

- 返回结果

参数	描述
columns	列名和类型
database	数据库名
table	表名
partitioned	是否分区表，只有extended下才会显示。
location	表的位置，只有extended下才会显示。
outputformat	输出形式，只有extended下才会显示。
inputformat	输入形式，只有extended下才会显示。
owner	表的属主，只有extended下才会显示。
partitionColumns	分区的列，只有extended下才会显示。

- 例子

```
curl -i -u : --insecure --negotiate 'https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1?format=extended'
```

13. ddl/database/:db/table/:table (PUT)
    - 描述

      创建表
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table
    - 参数

参数	描述
:db	数据库名
:table	新建表名

参数	描述
group	创建表时使用的用户组
permissions	创建表时使用的权限
external	指定位置，hive不使用表的默认位置。
ifNotExists	设置为true，当表存在时不会报错。
comment	备注
columns	列描述，包括列名，类型和可选备注。
partitionedBy	分区列描述，用于划分表格。参数columns列出了列名，类型和可选备注。
clusteredBy	分桶列描述，参数包括columnNames、sortedBy、和numberOfBuckets。参数columnNames包括columnName和排列顺序（ASC为升序，DESC为降序）。
format	存储格式，参数包括rowFormat，storedAs，和storedBy。
location	HDFS路径
tableProperties	表属性和属性值（name-value对）

- 返回结果

参数	描述
database	数据库名
table	表名

- 例子

```
curl -i -u : --insecure --negotiate -X PUT -HContent-type:application/json -d '{"columns": [{"name": "id", "type": "int"}, {"name": "name","type": "string"}], "comment": "hello","format": {"storedAs": "orc"} }' 'https://10.64.35.144:9111/templeton/v1/ddl/database/db3/table/tbl1'
```

14. ddl/database/:db/table/:table (POST)
    - 描述

      重命名表
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table
    - 参数

参数	描述
:db	数据库名
:table	已有表名
rename	新表表名

- 返回结果

参数	描述
database	数据库名
table	新表表名

- 例子

```
curl -i -u : --insecure --negotiate -d rename=table1 'https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/tbl1'
```

15. ddl/database/:db/table/:table (DELETE)
    - 描述

      删除表
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table
    - 参数

参数	描述
:db	数据库名
:table	表名
ifExists	当设置为true时，不报错。

    - 返回结果

参数	描述
database	数据库名
table	表名

    - 例子

```
curl -i -u : --insecure --negotiate -X DELETE 'https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/table2?ifExists=true'
```

16. ddl/database/:db/table/:existingtable/like/:newtable (PUT)
    - 描述

      创建一张和已经存在的表一样的表
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:existingtable/like/:newtable

- 参数

参数	描述
:db	数据库名
:existingtable	已有表名
:newtable	新表名
group	创建表时使用的用户组。
permissions	创建表时使用的权限。
external	指定位置，hive不使用表的默认位置。
ifNotExists	当设置为true时，如果表已经存在，Hive不报错。
location	HDFS路径

- 返回结果

参数	描述
database	数据库名
table	表名

- 例子

```
curl -i -u : --insecure --negotiate -X PUT -HContent-type:application/json -d '{"ifNotExists": "true"}' 'https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/like/tt1'
```

17. ddl/database/:db/table/:table/partition(GET)
    - 描述

      列出表的分区信息
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition
    - 参数

参数	描述
:db	数据库名
:table	表名

    - 返回结果

参数	描述
database	数据库名
table	表名

参数	描述
partitions	分区属性值和分区名

- 例子

```
curl -i -u : --insecure --negotiate https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition
```

18. ddl/database/:db/table/:table/partition/:partition(GET)
    - 描述

      列出表的某个具体分区的信息
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition/:partition
    - 参数

参数	描述
:db	数据库名
:table	表名
:partition	分区名，解码http引用时，需当心。比如country=%27algeria%27。

    - 返回结果

参数	描述
database	数据库名
table	表名
partition	分区名
partitioned	如果设置为true，为分区表
location	表的存储路径
outputFormat	输出格式
columns	列名，类型，备注
owner	所有者
partitionColumns	分区的列
inputFormat	输入格式
totalNumberFiles	分区下文件个数
totalFileSize	分区下文件总大小
maxFileSize	最大文件大小

参数	描述
minFileSize	最小文件大小
lastAccessTime	最后访问时间
lastUpdateTime	最后更新时间

- 例子

```
curl -i -u : --insecure --negotiate https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition/dt=1
```

19. ddl/database/:db/table/:table/partition/:partition(PUT)
    - 描述

      增加一个表分区
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition/:partition
    - 参数

参数	描述
:db	数据库名。
:table	表名。
group	创建新分区时使用的用户组。
permissions	创建新分区时用户的权限。
location	新分区的存放位置。
ifNotExists	如果设置为true， 当分区已经存在，系统报错。

    - 返回结果

参数	描述
database	数据库名
table	表名
partitions	分区名

    - 例子

```
curl -i -u : --insecure --negotiate -X PUT -HContent-type:application/json -d '{}' https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition/dt=10
```

20. ddl/database/:db/table/:table/partition/:partition(DELETE)
    - 描述

      删除一个表分区
    - URL

https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition/:partition

- 参数

参数	描述
:db	数据库名。
:table	表名。
group	删除新分区时使用的用户组。
permissions	删除新分区时用户的权限， 格式为rwxrw-r-x。
ifExists	如果指定分区不存在，Hive报错。参数值设置为true除外。

- 返回结果

参数	描述
database	数据库名
table	表名
partitions	分区名

- 例子

```
curl -i -u : --insecure --negotiate -X DELETE -HContent-type:application/json -d '{}' https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition/dt=10
```

21. ddl/database/:db/table/:table/column(GET)
    - 描述

      获取表的column列表

    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/column

    - 参数

参数	描述
:db	数据库名
:table	表名

    - 返回结果

参数	描述
database	数据库名
table	表名

参数	描述
columns	列名字和类型

- 例子

```
curl -i -u : --insecure --negotiate https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column
```

22. ddl/database/:db/table/:table/column/:column(GET)
    - 描述

      获取表的某个具体的column的信息
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/column/:column
    - 参数

参数	描述
:db	数据库名
:table	表名
:column	列名

    - 返回结果

参数	描述
database	数据库名
table	表名
column	列名字和类型

    - 例子

```
curl -i -u : --insecure --negotiate https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column/id
```

23. ddl/database/:db/table/:table/column/:column(PUT)
    - 描述

      增加表的一列
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/column/:column
    - 参数

参数	描述
:db	数据库名
:table	表名

参数	描述
:column	列名
type	列类型，比如string和int
comment	列备注，比如描述

- 返回结果

参数	描述
database	数据库名
table	表名
column	列名

- 例子

```
curl -i -u : --insecure --negotiate -X PUT -HContent-type:application/json -d '{"type": "string", "comment": "new column"}'  https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column/name
```

24. ddl/database/:db/table/:table/property(GET)
    - 描述

      获取表的property
    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/property
    - 参数

参数	描述
:db	数据库名
:table	表名

    - 返回结果

参数	描述
database	数据库名
table	表名
properties	属性列表

    - 例子

```
curl -i -u : --insecure --negotiate https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property
```

25. ddl/database/:db/table/:table/property/:property(GET)
    - 描述

      获取表的某个具体的property的值

- URL

  https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/property/:property

- 参数

参数	描述
:db	数据库名
:table	表名
:property	属性名

- 返回结果

参数	描述
database	数据库名
table	表名
property	属性列表

- 例子

```
curl -i -u : --insecure --negotiate https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property/last_modified_by
```

26. ddl/database/:db/table/:table/property/:property(PUT)
    - 描述

      增加表的property的值

    - URL

      https://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/property/:property

    - 参数

参数	描述
:db	数据库名
:table	表名
:property	属性名
value	属性值

    - 返回结果

参数	描述
database	数据库名
table	表名
property	属性名

- 例子

```
curl -i -u : --insecure --negotiate -X PUT -HContent-type:application/json -d '{"value": "my value"}' https://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property/mykey
```

27. mapreduce/jar(POST)
    - 描述

      执行MR任务，在执行之前，需要将MR的jar包上传到HDFS中
    - URL

      https://www.myserver.com/templeton/v1/mapreduce/jar
    - 参数

参数	描述
jar	需要执行的MR的jar包。
class	需要执行的MR的分类。
libjars	需要加入的classpath的jar包名，以逗号分隔。
files	需要复制到集群的文件名，以逗号分隔。
arg	Main类接受的输入参数。
define	设置hadoop的配置，格式为：define=NAME=VALUE。
statusdir	WebHCat会将执行的MR任务的状态写入到statusdir中。如果设置了这个值，那么需要用户手动进行删除。
enablelog	如果statusdir设置，enablelog设置为true，收集Hadoop任务配置和日志到$statusdir/logs。此后，成功和失败的尝试，都将记录进日志。$statusdir/logs下，子目录布局为： logs/$job_id (directory for $job_id) logs/$job_id/job.xml.html logs/$job_id/$attempt_id (directory for $attempt_id) logs/$job_id/$attempt_id/stderr logs/$job_id/$attempt_id/stdout logs/$job_id/$attempt_id/syslog 仅支持Hadoop 1.X。
callback	在MR任务执行完的回调地址，使用$jobId，将任务ID嵌入回调地址。在回调地址中，任务ID替换该$jobId。

- 返回结果

参数	描述
id	任务ID，类似“job_201110132141_0001”

- 例子

```
curl -i -u : --insecure --negotiate -d jar="/tmp/word.count-0.0.1-SNAPSHOT.jar" -d
class=com.huawei.word.count.WD -d statusdir="/output" -d enablelog=true "https://
10.64.35.144:9111/templeton/v1/mapreduce/jar"
```

28. mapreduce/streaming(POST)
    - 描述

      以Streaming方式提交MR任务
    - URL

      https://www.myserver.com/templeton/v1/mapreduce/streaming
    - 参数

参数	描述
input	Hadoop中input的路径。
output	存储output的路径。如没有规定，WebHCat将output储存在使用队列资源可以发现到的路径。
mapper	mapper程序位置。
reducer	reducer程序位置。
files	HDFS文件添加到分布式缓存中。
arg	设置argument。
define	设置hadoop的配置变量，格式：define=NAME=VALUE
cmdenv	设置环境变量，格式：cmdenv=NAME=VALUE
statusdir	WebHCat会将执行的MR任务的状态写入到statusdir中。如果设置了这个值，那么需要用户手动进行删除。

参数	描述
enablelog	如果statusdir设置，enablelog设置为true，收集Hadoop任务配置和日志到$statusdir/logs。此后，成功和失败的尝试，都将记录进日志。$statusdir/logs下，子目录布局为： logs/$job_id (directory for $job_id) logs/$job_id/job.xml.html logs/$job_id/$attempt_id (directory for $attempt_id) logs/$job_id/$attempt_id/stderr logs/$job_id/$attempt_id/stdout logs/$job_id/$attempt_id/syslog 仅支持Hadoop 1.X。
callback	在MR任务执行完的回调地址，使用$jobId，将任务ID嵌入回调地址。在回调地址中，任务ID将替换该$jobId。

- 返回结果

参数	描述
id	任务ID，类似job_201110132141_0001

- 例子

```
curl -i -u : --insecure --negotiate -d input=/input -d output=/oooo -d mapper=/bin/cat -d reducer="/usr/bin/wc -w" -d statusdir="/output" 'https://10.64.35.144:9111/templeton/v1/mapreduce/streaming'
```

29. /hive(POST)
    - 描述

      执行Hive命令
    - URL

      https://www.myserver.com/templeton/v1/hive
    - 参数

参数	描述
execute	hive命令，包含整个和短的Hive命令。
file	包含hive命令的HDFS文件。
files	需要复制到集群的文件名，以逗号分隔。
arg	设置argument。

参数	描述
define	设置hive配置，格式：define=key=value。使用post语句时需要配置实例的scratch dir。WebHCat实例使用define=hive.exec.scratchdir=**/tmp/hive-scratch**，WebHCat1实例使用define=hive.exec.scratchdir=**/tmp/hive1-scratch**，以此类推。
statusdir	WebHCat会将执行的MR任务的状态写入到statusdir中。如果设置了这个值，那么需要用户手动进行删除。
enablelog	如果statusdir设置，enablelog设置为true，收集Hadoop任务配置和日志到$statusdir/logs。此后，成功和失败的尝试，都将记录进日志。$statusdir/logs下，子目录布局为： logs/$job_id (directory for $job_id) logs/$job_id/job.xml.html logs/$job_id/$attempt_id (directory for $attempt_id) logs/$job_id/$attempt_id/stderr logs/$job_id/$attempt_id/stdout logs/$job_id/$attempt_id/syslog
callback	在MR任务执行完的回调地址，使用$jobId，将任务ID嵌入回调地址。在回调地址中，任务ID将替换该$jobId。

- 返回结果

参数	描述
id	任务ID，类似job_201110132141_0001

- 例子

```
curl -i -u : --insecure --negotiate -d execute="select count(*) from t1" -d define=hive.exec.scratchdir=/tmp/hive-scratch -d statusdir="/output" "https://10.64.35.144:9111/templeton/v1/hive"
```

30. jobs(GET)
    - 描述

      获取所有的job id
    - URL

      https://www.myserver.com/templeton/v1/jobs

- 参数

参数	描述
fields	如果设置成*，那么会返回每个job的详细信息。如果没设置，只返回任务ID。现在只能设置成*，如设置成其他值，将出现异常。
jobid	如果设置了jobid，那么只有字典顺序比jobid大的job才会返回。比如，如果jobid为"job_201312091733_0001"，只有大于该值的job才能返回。返回的job的个数，取决于numrecords。
numrecords	如果设置了numrecords和jobid，jobid列表按字典顺序排列，待jobid返回后，可以得到numrecords的最大值。如果jobid没有设置， 而numrecords设置了参数值，jobid按字典顺序排列后，可以得到numrecords的最大值。相反，如果numrecords没有设置，而jobid设置了参数值，所有大于jobid的job都将返回。
showall	如果设置为true，用户可以浏览的所有job都将返回。不仅仅是用户所拥有的job。

- 返回结果

参数	描述
id	Job id
detail	如果showall为true，那么显示detail信息，否则为null。

- 例子

```
curl -i -u : --insecure --negotiate  "https://10.64.35.144:9111/templeton/v1/jobs"
```

31. jobs/:jobid(GET)
    - 描述

      获取指定job的信息
    - URL

      https://www.myserver.com/templeton/v1/jobs/:jobid
    - 参数

参数	描述
jobid	Job创建后的Jobid

- 返回结果

参数	描述
status	包含job状态信息的json对象。
profile	包含job状态的json对象。WebHCat解析JobProfile对象中的信息，该对象因Hadoop版本不同而不同。
id	Job的id。
percentComplete	完成百分比，比如75% complete，如果完成后则为null。
user	创建job的用户。
callback	回调URL（如果有）。
userargs	用户提交job时的argument参数和参数值。
exitValue	job退出值。

- 例子

```
curl -i -u : --insecure --negotiate  "https://10.64.35.144:9111/templeton/v1/jobs/job_1440386556001_0255"
```

32. jobs/:jobid(DELETE)
    - 描述

      kill任务
    - URL

      https://www.myserver.com/templeton/v1/jobs/:jobid
    - 参数

参数	描述
:jobid	删除的Job的ID

    - 返回结果

参数	描述
user	提交Job的用户。
status	包含Job状态信息的JSON对象。
profile	包含job信息的json对象。WebHCat解析JobProfile对象中的信息，该对象因Hadoop版本不同而不同。
id	Job的id。
callback	回调的URL（如果有）。

- 例子
  curl -i -u : --insecure --negotiate -X DELETE "https://10.64.35.143:9111/templeton/v1/jobs/job_1440386556001_0265"

15.5.2 配置 Windows 通过 EIP 访问安全模式集群 Hive

操作场景

该章节通过指导用户配置集群绑定EIP，并配置Hive文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行hive-jdbc-example样例为例进行说明。

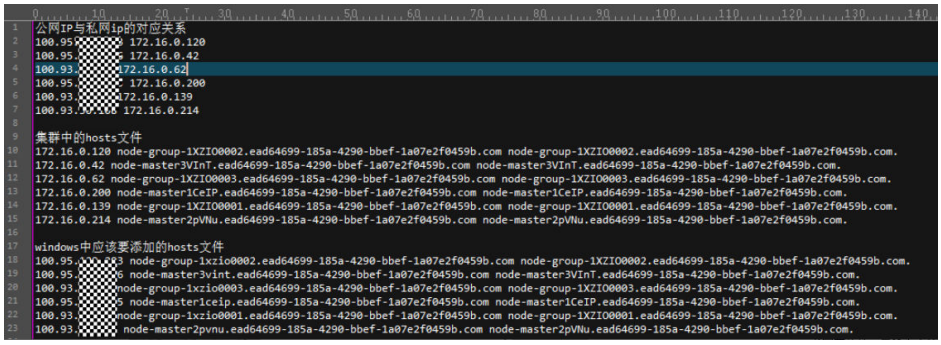
操作步骤

步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。



步骤2 将krb5.conf文件中的IP地址修改为对应IP的主机名称。

步骤3 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和21730TCP、21731TCP/UDP、21732TCP/UDP端口。

步骤4 在Manager界面选择“集群 > 服务 > Hive > 更多 > 下载客户端”，将客户端中的core-site.xml和hiveclient.properties复制到样例工程的resources目录下。

步骤5 修改样例代码中的JDBC URL中使用zookeeper的连接改为直接使用hiveserver2的地址连接。将url改为jdbc:hive2:// *hiveserver主机名*:10000/

说明

- 由于使用zookeeper连接会访问zookeeper的“/hiveserver2”目录下的IP，但是里面存储的是私有IP，本地Windows无法连通，所以需要替换为hiveserver2的地址连接。
- hiveserver2服务的主机名可以在Manager界面选择“集群 > 服务 > Hive > 实例”，在“实例”界面查看“HiveServer”的“主机名称”获取。

```
// 拼接JDBC URL
//   StringBuilder strBuilder = new StringBuilder("jdbc:hive2://").append(zkQuorum).append("/");
StringBuilder strBuilder = new StringBuilder("jdbc:hive2://node-master1lnks.c0c17d76-481f-4b24-83af-159ed415ad95.com:10000/");

if ("KERBEROS".equalsIgnoreCase(auth)) {
    strBuilder
//              .append(";serviceDiscoveryMode=")
//              .append(serviceDiscoveryMode)
//              .append(";zooKeeperNamespace=")
//              .append(zooKeeperNamespace)
//              .append(";sasl.qop=")
//              .append(sasl_qop)
//              .append(";auth=")
//              .append(auth)
            .append(";principal=")
            .append(principal)
            .append(";user.principal=")
            .append(USER_NAME)
            .append(";user.keytab=")
            .append(USER_KEYTAB_FILE)
            .append(";");
} else {
    /* 普通模式 */
//    strBuilder
            .append(";serviceDiscoveryMode=")
//          .append(serviceDiscoveryMode)
//          .append(";zooKeeperNamespace=")
//          .append(zooKeeperNamespace)
//          .append(";auth=none");
}
```

步骤6 在运行样例代码前，需要将样例代码中的PRNCIPAL_NAME改为安全认证的用户名。

----结束

15.5.3 使用二次开发程序产生 Unable to read HiveServer2 异常如何处理

问题

使用二次开发程序产生异常，提示“Unable to read HiveServer2 configs from ZooKeeper”信息。

回答

- 问题原因
  - 使用的krb5.conf、user.keytab文件不是最新的，或者文件与示例代码里填写登录用户不匹配。
  - 使用客户端环境的时间与连接的集群时间差大于5分钟。
- 解决措施
  - 检查代码下载最新的用户的认证凭据文件。
  - 查看集群环境和客户端环境的时间是否相差在5分钟之内，若超过5分钟，请修改客户端环境时间。

15.5.4 使用 IBM JDK 产生异常“Problem performing GSS wrap”如何处理

问题

使用IBM JDK产生异常，提示“Problem performing GSS wrap”信息。

回答

问题原因：

在IBM JDK下建立的Hive connection时间超过登录用户的认证超时时间（默认一天），导致认证失败。

说明

IBM JDK的机制跟Oracle JDK的机制不同，IBM JDK在认证登录后的使用过程中做了时间检查却没有检测外部的时间更新，导致即使显式调用Hive relogin也无法得到刷新。

解决措施：

通常情况下，在发现Hive connection不可用的时候，可以关闭该connection，重新创建一个connection继续执行。

15.5.5 Hive SQL 与 SQL2003 标准有哪些兼容性问题

本文列举目前已发现的Hive SQL与SQL2003标准兼容性问题。

- **不支持在having中写视图。**

  举例如下：

```
select c_last_name
    ,c_first_name
    ,s_store_name
```

```
    ,sum(netpaid) paid
from ssales
where i_color = 'chiffon'
group by c_last_name
      ,c_first_name
      ,s_store_name
having sum(netpaid) > (select 0.05*avg(netpaid) from ssales);
```

报错：

```
Error: Error while compiling statement: FAILED:ParseException line 46:23 cannot recognize input near 'select' '0.05' '*' in expression specification (state=42000,code=40000)
```

- **Having不支持子查询。**

  举例如下：

```
select
      ps_partkey,
      sum(ps_supplycost * ps_availqty) as value
from
      partsupp,
      supplier,
      nation
where
      ps_suppkey = s_suppkey
      and s_nationkey = n_nationkey
      and n_name = 'SAUDI ARABIA'
group by
      ps_partkey having
            sum(ps_supplycost * ps_availqty) > (
                  select
                        sum(ps_supplycost * ps_availqty) * 0.0001000000
                  from
                        partsupp,
                        supplier,
                        nation
                  where
                        ps_suppkey = s_suppkey
                        and s_nationkey = n_nationkey
                        and n_name = 'SAUDI ARABIA'
            )
order by
      value desc;
```

  报错：

```
Error: Error while compiling statement: FAILED: SemanticException Line 0:-1 Unsupported SubQuery Expression ''SAUDI ARABIA'': Only SubQuery expressions that are top level conjuncts are allowed (state=42000,code=40000)
```

- **不支持多查询结果当成多个字段输出。**

  举例如下：

```
select
      c_count,
      count(*) as custdist
from
      (
            select
                  c_custkey,
                  count(o_orderkey)
            from
                  customer left outer join orders on
                        c_custkey = o_custkey
                        and o_comment not like '%pending%requests%'
            group by
                  c_custkey
      ) as c_orders (c_custkey, c_count)
group by
      c_count
```

```
order by
    custdist desc,
    c_count desc;
```

报错：

```
Error: Error while compiling statement: FAILED: ParseException line 1:213 missing EOF at '(' near
'c_orders' (state=42000,code=40000)
```

- **不支持把查询的结果当成字段比较。**

  举例如下：

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#25'
    and p_container = 'MED JAR'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

  报错：

```
Error: Error while compiling statement: FAILED: SemanticException [Error 10249]: Line 14:4
Unsupported SubQuery Expression 'ps_suppkey': Correlating expression contains ambiguous column
references. (state=42000,code=10249)
```

- **多表关联的过滤条件中，不支持按not in或in的子查询过滤**

  举例如下：

```
select
    p_brand,
    p_type,
    p_size,
    count(distinct ps_suppkey) as supplier_cnt
from
    partsupp,
    part
where
    p_partkey = ps_partkey
    and p_brand <> 'Brand#12'
    and p_type not like 'PROMO PLATED%'
    and p_size in (25, 2, 43, 9, 35, 36, 48, 24)
    and ps_suppkey in (
        select
            s_suppkey as ps_suppkey
        from
            supplier
        where
            s_comment like '%Customer%Complaints%'
    )
group by
    p_brand,
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size;
```

  报错：

```
Error: Error while compiling statement: FAILED: SemanticException [Error 10249]: Line 14:4
Unsupported SubQuery Expression 'ps_suppkey': Correlating expression contains ambiguous column
references. (state=42000,code=10249)
```

- **多表关联不支持not in和in在子查询中过滤。**

  举例如下：

```
select
      c_name,
      c_custkey,
      o_orderkey,
      o_orderdate,
      o_totalprice,
      sum(l_quantity)
from
      customer,
      orders,
      lineitem
where
      o_orderkey in (
            select
                  l_orderkey
            from
                  lineitem
            group by
                  l_orderkey having
                        sum(l_quantity) > 315
      )
      and c_custkey = o_custkey
      and o_orderkey = l_orderkey
group by
      c_name,
      c_custkey,
      o_orderkey,
      o_orderdate,
      o_totalprice
order by
      o_totalprice desc,
      o_orderdate
limit 100;
```

  报错：

```
Error: Error while compiling statement: FAILED: SemanticException [Error 10249]: Line 13:0
Unsupported SubQuery Expression 'o_orderkey': Correlating expression contains ambiguous column
references. (state=42000,code=10249)
```

- **关联条件中不支持多个exists查询体。**

  举例如下：

```
select
      s_name,
      count(*) as numwait
from
      supplier,
      lineitem l1,
      orders,
      nation
where
      s_suppkey = l1.l_suppkey
      and o_orderkey = l1.l_orderkey
      and o_orderstatus = 'F'
      and l1.l_receiptdate > l1.l_commitdate
      and exists (
            select
                  *
            from
                  lineitem l2
            where
                  l2.l_orderkey = l1.l_orderkey
                  and l2.l_suppkey <> l1.l_suppkey
```

```
    )
    and not exists (
        select
            *
        from
            lineitem l3
        where
            l3.l_orderkey = l1.l_orderkey
            and l3.l_suppkey <> l1.l_suppkey
            and l3.l_receiptdate > l3.l_commitdate
    )
    and s_nationkey = n_nationkey
    and n_name = 'VIETNAM'
group by
    s_name
order by
    numwait desc,
    s_name
limit 100;
```

报错：

```
Error: Error while compiling statement: FAILED: SemanticException [Error 10249]: Line 23:8
Unsupported SubQuery Expression 'l_commitdate': Only 1 SubQuery expression is supported.
(state=42000,code=10249)
```

- **不支持多层级in嵌套子查询。**

  举例如下：

```
select i_item_id item_id,
       sum(sr_return_quantity) sr_item_qty
 from store_returns,
     item,
     date_dim
 where sr_item_sk = i_item_sk
 and   d_date    in
       (select d_date
       from date_dim
       where d_week_seq in
             (select d_week_seq
             from date_dim
         where d_date in ('1998-01-02','1998-10-15','1998-11-10')))
 and   sr_returned_date_sk   = d_date_sk
 group by i_item_id),
 cr_items as
 (select i_item_id item_id,
       sum(cr_return_quantity) cr_item_qty
 from catalog_returns,
     item,
     date_dim
 where cr_item_sk = i_item_sk);
```

报错：

```
Unsupported SubQuery Expression 'd_week_seq': SubQuery cannot use the table alias: date_dim; this
is also an alias in the Outer Query and SubQuery contains a unqualified column reference
(state=42000,code=10249)
```

16 Hive 开发指南（普通模式）

16.1 Hive 应用开发概述

16.1.1 Hive 应用开发简介

Hive 简介

Hive是一个开源的，建立在Hadoop上的数据仓库框架，提供类似SQL的HQL语言操作结构化数据，其基本原理是将HQL语言自动转换成MapReduce任务或Spark任务，从而完成对Hadoop集群中存储的海量数据进行查询和分析。

Hive主要特点如下：

- 通过HQL语言非常容易的完成数据提取、转换和加载（ETL）。
- 通过HQL完成海量结构化数据分析。
- 灵活的数据存储格式，支持JSON、CSV、TEXTFILE、RCFILE、ORCFILE、SEQUENCEFILE等存储格式，并支持自定义扩展。
- 多种客户端连接方式，支持JDBC接口。

Hive的主要应用于海量数据的离线分析（如日志分析，集群状态分析）、大规模的数据挖掘（用户行为分析，兴趣分区，区域展示）等场景下。

为保证Hive服务的高可用性、用户数据的安全及访问服务的可控制，在开源社区的Hive-3.1.0版本基础上，Hive新增如下特性：

数据文件加密机制：开源社区的Hive特性，请参见**https://cwiki.apache.org/confluence/display/hive/designdocs**。

16.1.2 Hive 应用开发常用概念

- **客户端**

  客户端直接面向用户，可通过Java API、Thrift API访问服务端进行Hive的相关操作。
- **HQL语言**

  Hive Query Language，类SQL语句。

- **HCatalog**

  HCatalog是建立在Hive元数据之上的一个表信息管理层，吸收了Hive的DDL命令。为MapReduce提供读写接口，提供Hive命令行接口来进行数据定义和元数据查询。基于MRS的HCatalog功能，Hive、MapReduce开发人员能够共享元数据信息，避免中间转换和调整，能够提升数据处理的效率。

- **WebHCat**

  WebHCat运行用户通过Rest API来执行Hive DDL，提交MapReduce任务，查询MapReduce任务执行结果等操作。

16.1.3 Hive 应用开发流程

开发流程中各阶段的说明如图16-1和表16-1所示。

图 16-1 Hive 应用程序开发流程

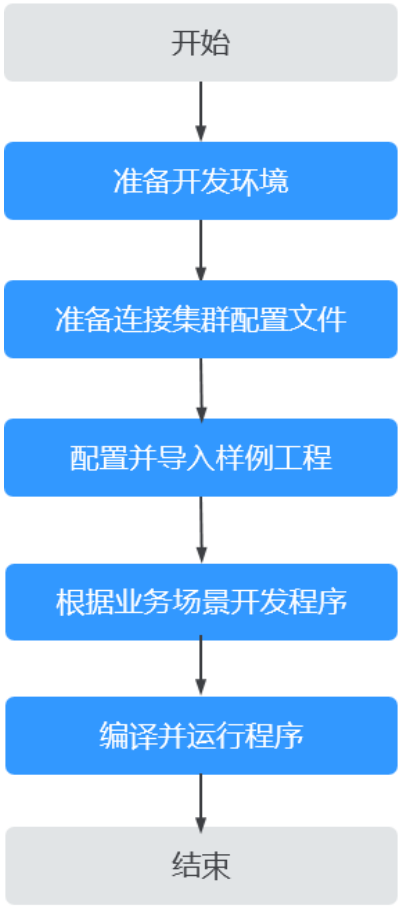


表 16-1 Hive 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，推荐使用Java语言进行开发，使用IntelliJ IDEA工具，同时完成JDK、Maven等初始配置。	**准备本地应用开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备连接Hive集群配置文件**
配置并导入样例工程	Hive提供了不同场景下的多种样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。	**导入并配置Hive样例工程**
根据业务场景开发程序	根据实际业务场景开发程序，调用组件接口实现对应功能。	**开发Hive应用**
编译并运行程序	开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。	**调测Hive应用**

16.1.4 Hive 应用开发样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Hive相关样例工程：

表 16-2 Hive 相关样例工程

样例工程位置	描述
• hive-examples/hive-jdbc-example • hive-examples/hive-jdbc-example-multizk	Hive JDBC处理数据Java示例程序。 本工程使用JDBC接口连接Hive，在Hive中执行相关数据操作。使用JDBC接口实现创建表、加载数据、查询数据等功能，还可实现在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper，相关样例介绍请参见**Hive JDBC访问样例程序**。
hive-examples/hcatalog-example	Hive HCatalog处理数据Java示例程序。 使用HCatalog接口实现通过Hive命令行方式对MRS Hive元数据进行数据定义和查询操作，相关样例介绍请参见**HCatalog访问Hive样例程序**。
hive-examples/python-examples	使用Python连接Hive执行SQL样例。 可实现使用Python对接Hive并提交数据分析任务，相关样例介绍请参见**基于Python的Hive样例程序**。
hive-examples/python3-examples	使用Python3连接Hive执行SQL样例。 可实现使用Python3对接Hive并提交数据分析任务，相关样例介绍请参见**基于Python3的Hive样例程序**。

16.2 准备 Hive 应用开发环境

16.2.1 准备本地应用开发环境

Hive组件可以使用JDBC/HCatalog/Python/Python3接口进行应用开发。

准备 JDBC/HCatalog 开发环境

表 16-3 JDBC/HCatalog 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。

准备项	说明
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 ● X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 ● TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装和配置IntelliJ IDEA	用于开发Hive应用程序的工具。版本要求如下： JDK使用1.8版本，IntelliJ IDEA使用2019.1或其他兼容版本。 **说明** ● 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 ● 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 ● 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。 华为提供开源镜像站，各服务样例工程依赖的Jar包通过华为开源镜像站下载，剩余所依赖的开源Jar包请直接从Maven中央库或者其他用户自定义的仓库地址下载，详情请参考**配置华为开源镜像仓**。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

准备 Python 开发环境

表 16-4 Python 开发环境

准备项	说明
操作系统	开发环境和运行环境：Linux系统。
安装Python	用于开发Hive应用程序的工具，版本要求不低于2.6.6，最高不超过2.7.13。
安装setuptools	Python开发环境的基本配置，版本要求5.0以上。

说明

Python开发工具的详细安装配置可参见**配置Hive Python样例工程**。

准备 Python3 开发环境

表 16-5 Python3 开发环境

准备项	说明
操作系统	开发环境和运行环境：Linux系统。
安装Python3	用于开发Hive应用程序的工具，版本要求不低于3.6，最高不超过3.8。
安装setuptools	Python3开发环境的基本配置，版本要求为47.3.1。

说明

Python3开发工具的详细安装配置可参见**配置Hive Python3样例工程**。

16.2.2 准备连接 Hive 集群配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。
  a. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

     例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。
  b. 进入客户端解压路径“Hive\config”，获取**表16-6**中相关配置文件。

     表 16-6 配置文件

文件名称	作用
hiveclient.properties	Hive客户端连接相关配置参数。
core-site.xml	Hadoop客户端相关配置参数。

  c. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群，具体操作请参考**配置Windows通过EIP访问普通模式集群Hive**。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。
  a. 在节点中安装客户端。
     例如客户端安装目录为“/opt/client”。
     客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。
  b. 获取配置文件：
     i. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**），勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。
     ii. 以**root**用户登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“Hive/config”路径下的**表16-6**中相关配置文件。
     例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：
     cd /tmp/FusionInsight-Client
     tar -xvf FusionInsight_Cluster_1_Services_Client.tar
     tar -xvf FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar
     cd FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles
  c. 检查客户端节点网络连接。
     在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

16.2.3 导入并配置 Hive 样例工程

16.2.3.1 导入并配置 Hive JDBC/HCatalog 样例工程

操作场景

为了运行MRS产品Hive组件的JDBC接口样例代码，需要完成下面的操作。

说明

- 本章节以在Windows环境下开发JDBC/HCatalog方式连接Hive服务的应用程序为例。
- HCatalog样例仅支持在Linux节点上运行。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“hive-jdbc-example”或“hcatalog-example”。

步骤2 对于“hive-jdbc-example”样例工程，还需将**准备连接Hive集群配置文件**获取的配置文件“core-site.xml”、“hiveclient.properties”放置到“hive-jdbc-example\src\main\resources”目录下。

步骤3 导入样例工程到IntelliJ IDEA开发环境中。

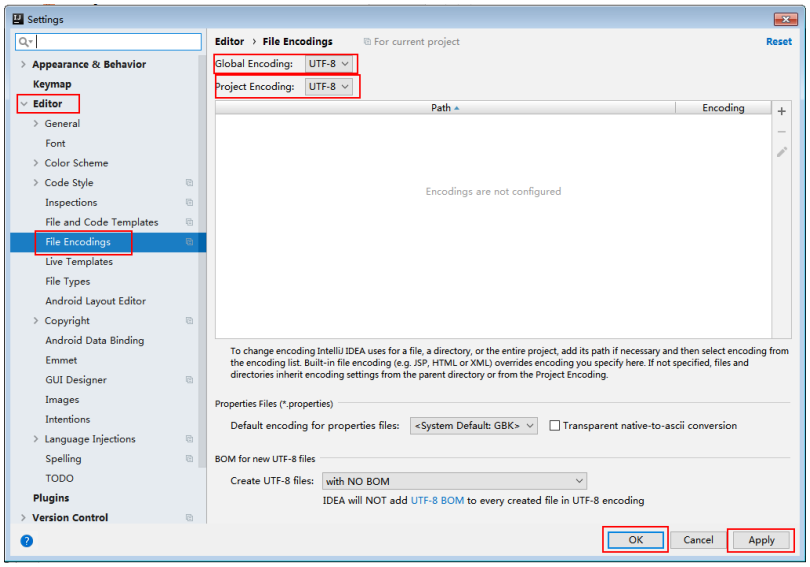
1. 在IntelliJ IDEA的菜单栏中，选择“File > Open...”，显示“Open File or Project”对话框。
2. 在弹出窗口选择文件夹“hive-jdbc-example”，单击“OK”。Windows下要求该文件夹的完整路径不包含空格。

步骤4 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”。

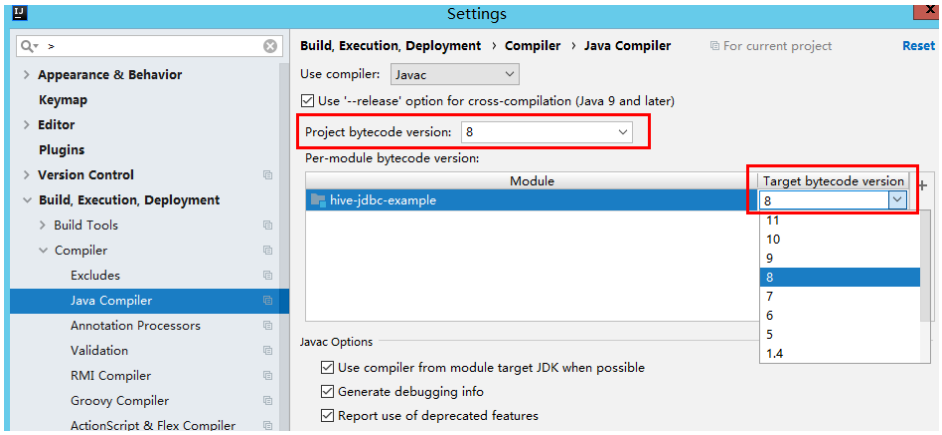
   弹出“Settings”窗口。
2. 在左边导航上选择“Editor > File Encodings”，在“Project Encoding”和“Global Encoding”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如**图16-2**所示。

图 16-2 设置 IntelliJ IDEA 的编码格式

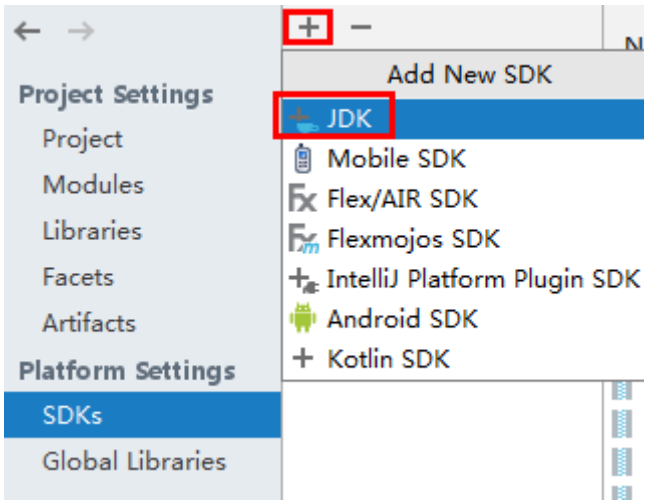


步骤5 设置工程JDK。

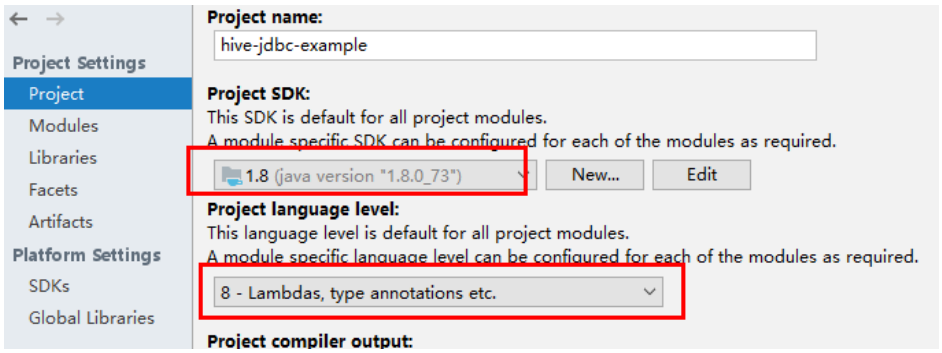
1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”，弹出“Settings”窗口。
2. 选择“Build, Execution, Deployment > Compiler > Java Compiler”，在“Project bytecode version”右侧的下拉菜单中，选择“8”。修改“hive-jdbc-example”的“Target bytecode version”为“8”。



3. 单击“Apply”后单击“OK”。
4. 在IntelliJ IDEA的菜单栏中，选择“File > Project Structure...”，弹出“Project Structure”窗口。
5. 选择“SDKs”，单击加号选择“JDK”。

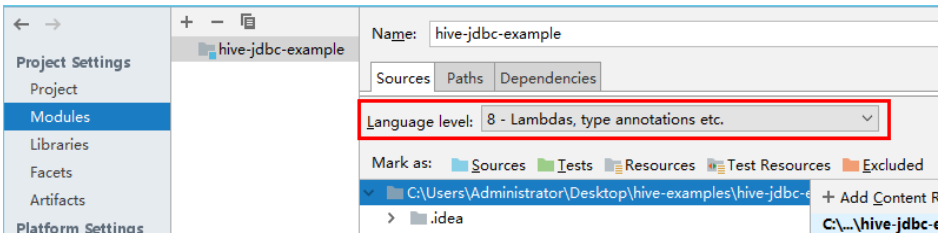


6. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。
7. 完成JDK选择后，单击“Apply”。
8. 选择“Project”，在“Project SDK”下的下拉菜单中选择在“SDKs”中添加的JDK，在“Project language level”下的下拉菜单中选择“8 - Lambdas, type annotations etc.”。

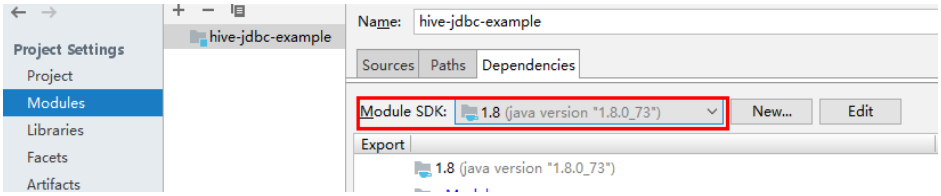


9. 单击“Apply”。

10. 选择“Modules”，在“Source”页面，修改“Language level”为“8 - Lambdas, type annotations etc.”。



在“Dependencies”页面，修改“Module SDK”为“SDKs”中添加的JDK。

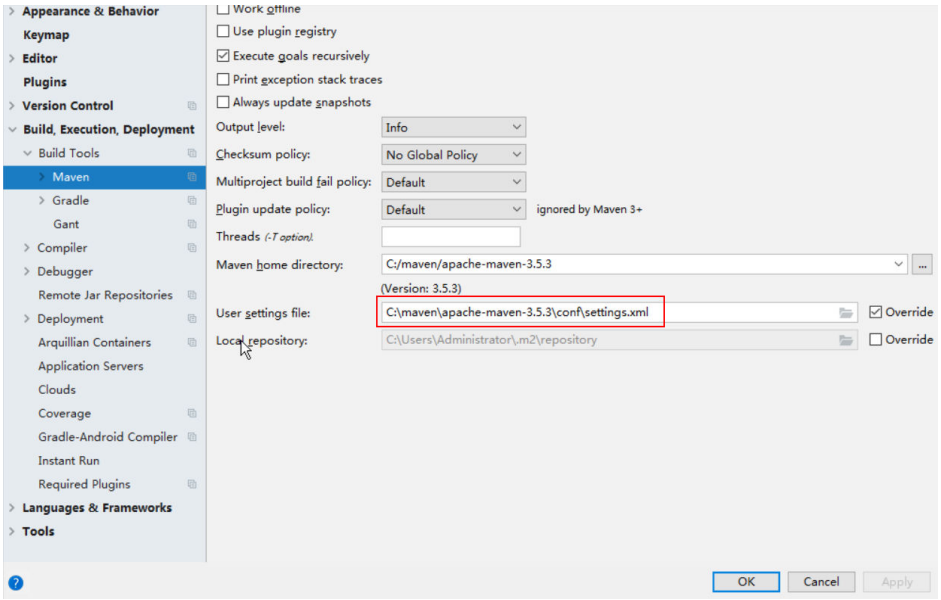


11. 单击“Apply”，单击“OK”。

步骤6 配置Maven。

1. 参考**配置华为开源镜像仓**章节描述，增加开源镜像仓地址等配置信息到本地Maven的“setting.xml”配置文件。
2. 修改完成后，在IntelliJ IDEA选择“File > Settings > Build, Execution, Deployment > Build Tools > Maven”，勾选“User settings file”右侧的“Override”，并修改“User settings file”的值为当前“settings.xml”文件放置目录，确保该目录为“*<本地Maven安装目录>*\conf\settings.xml”。

图 16-3 “settings.xml”文件放置目录



3. 单击“Maven home directory”右侧的下拉菜单，选择Maven的安装路径。
4. 单击“Apply”并单击“OK”。

----结束

16.2.3.2 配置 Hive Python 样例工程

操作场景

为了运行MRS产品Hive组件的Python接口样例代码，需要完成下面的操作。

说明

- MRS 3.1.2及之后版本默认仅支持Python3。
- 该样例仅支持在Linux节点上运行。

操作步骤

步骤1 客户端机器必须安装有Python，其版本不低于2.6.6，最高不能超过2.7.13。

在客户端机器的命令行终端输入**python**可查看Python版本号。如下显示Python版本为2.6.6。

```
Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)
[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

步骤2 客户端机器必须安装有setuptools，其版本不低于5.0，最高不能超过36.8.0。可在**https://pypi.org/project/setuptools/#files**获取相应的安装包。

将下载的setuptools压缩文件复制到客户端机器上，解压后进入解压目录，在客户端机器的命令行终端执行以下命令：

python setup.py install

如下内容表示安装setuptools的5.7版本成功。

```
Finished processing dependencies for setuptools==5.7
```

步骤3 安装Python客户端到客户端机器。

1. 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“python-examples”。
2. 进入“python-examples”文件夹。
3. 在命令行终端执行以下命令：

   python setup.py install

   输出以下关键内容表示安装Python客户端成功。

   ```
   Finished processing dependencies for pyhs2==0.5.0
   ```

步骤4 安装成功后，“python-examples/pyCLI_nosec.py”为Python客户端样例代码，“python-examples/pyhs2/haconnection.py”为Python客户端接口API。“hive_python_client”脚本提供了直接执行SQL的功能，如：**sh hive_python_client 'show tables'**。该功能只适用于常规简单的SQL，并且需要依赖ZooKeeper的客户端。

----结束

16.2.3.3 配置 Hive Python3 样例工程

操作场景

为了运行MRS产品Hive组件的Python3接口样例代码，需要完成下面的操作。

该样例仅支持在Linux节点上运行。

操作步骤

步骤1 客户端机器必须安装有Python3，其版本不低于3.6，最高不能超过3.8。

在客户端机器的命令行终端输入**python3**可查看Python版本号。如下显示Python版本为3.8.2。

```
Python 3.8.2 (default, Jun 23 2020, 10:26:03)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

步骤2 客户端机器必须安装有setuptools，版本可取47.3.1。可在**https://pypi.org/project/setuptools/#files**下载相应的安装包。

将下载的setuptools压缩文件复制到客户端机器上，解压后进入解压目录，在客户端机器的命令行终端执行以下命令：

python3 setup.py install

如下内容表示安装setuptools的47.3.1版本成功。

```
Finished processing dependencies for setuptools==47.3.1
```

说明

若提示setuptools的47.3.1版本安装不成功，则需要检查环境是否有问题或是Python自身原因导致的。

步骤3 安装Python客户端到客户端机器。

1. 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“python3-examples”。
2. 进入“python3-examples”文件夹。
3. 根据python3的版本，选择进入“dependency_python3.6”或“dependency_python3.7”或“dependency_python3.8”文件夹。
4. 执行**whereis easy_install**命令，找到easy_install程序路径。如果有多个路径，使用**easy_install --version**确认选择setuptools对应版本的**easy_install**，例如/usr/local/bin/easy_install。
5. 使用对应的easy_install命令，安装dependency_python3.x文件夹下的egg文件，egg文件存在依赖关系，可使用通配符安装，如：
   - “dependency_python3.6”目录：

     /usr/local/bin/easy_install future*egg six*egg python*egg sasl-*linux-$(uname -p).egg thrift-*egg thrift_sasl*egg
   - “dependency_python3.7”目录：

     /usr/local/bin/easy_install future*egg six*egg sasl-*linux-$(uname -p).egg thrift-*egg thrift_sasl*egg
   - “dependency_python3.8”目录：

     /usr/local/bin/easy_install future*egg six*egg python*egg sasl-*linux-$(uname -p).egg thrift-*linux-$(uname -p).egg thrift_sasl*egg

   每个egg文件安装输出以下关键内容表示安装成功。

   ```
   Finished processing dependencies for ***
   ```


步骤4 安装成功后，“python3-examples/pyCLI_nosec.py”为Python客户端样例代码，“python3-examples/pyhive/hive.py”为Python客户端接口API。

----结束

16.3 开发 Hive 应用

16.3.1 Hive JDBC 访问样例程序

16.3.1.1 Hive JDBC 样例程序开发思路

场景说明

假定用户开发一个Hive数据分析应用，用于管理企业雇员信息，如**表16-7**、**表16-8**所示。

开发思路

步骤1 数据准备。

1. 创建三张表，雇员信息表“employees_info”、雇员联络信息表“employees_contact”、雇员信息扩展表“employees_info_extended”。
   - 雇员信息表“employees_info”的字段为雇员编号、姓名、支付薪水币种、薪水金额、缴税税种、工作地、入职时间，其中支付薪水币种“R”代表人民币，“D”代表美元。
   - 雇员联络信息表“employees_contact”的字段为雇员编号、电话号码、e-mail。
   - 雇员信息扩展表“employees_info_extended”的字段为雇员编号、姓名、电话号码、e-mail、支付薪水币种、薪水金额、缴税税种、工作地，分区字段为入职时间。

     创建表代码实现请见**创建Hive表**。
2. 加载雇员信息数据到雇员信息表“employees_info”中。

   加载数据代码实现请见**加载数据到Hive表中**。

   雇员信息数据如**表16-7**所示：

表 16-7 雇员信息数据

编号	姓名	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
1	Wang	R	8000.01	personal income tax&0.05	Country 1:City1	2014
3	Tom	D	12000.02	personal income tax&0.09	Country 2:City2	2014

编号	姓名	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
4	Jack	D	24000.03	personal income tax&0.09	Country 3:City3	2014
6	Linda	D	36000.04	personal income tax&0.09	Country 4:City4	2014
8	Zhang	R	9000.05	personal income tax&0.05	Country 5:City5	2014

3. 加载雇员联络信息数据到雇员联络信息表“employees_contact”中。
雇员联络信息数据如**表16-8**所示：

表 16-8 雇员联络信息数据

编号	电话号码	e-mail
1	135 XXXX XXXX	xxxx@xx.com
3	159 XXXX XXXX	xxxxx@xx.com.cn
4	186 XXXX XXXX	xxxx@xx.org
6	189 XXXX XXXX	xxxx@xxx.cn
8	134 XXXX XXXX	xxxx@xxxx.cn

4. 加载雇员扩展信息数据到雇员联络信息表“employees_info_extended”中。
雇员扩展信息数据如**表16-9**所示：

表 16-9 雇员扩展信息数据

编号	姓名	电话号码	e-mail	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
1	Wang	135 XXXX XXXX	xxxx@xx.com	R	8000.01	personal income tax&0.05	Country1:City1	2014
3	Tom	159 XXXX XXXX	xxxxx@xx.com.cn	D	12000.02	personal income tax&0.09	Country2:City2	2014

编号	姓名	电话号码	e-mail	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
4	Jack	186 XXXX XXXX	xxxx@xx.org	D	24000.03	personal income tax&0.09	Country3:City3	2014
6	Linda	189 XXXX XXXX	xxxx@xxx.cn	D	36000.04	personal income tax&0.09	Country4:City4	2014
8	Zhang	134 XXXX XXXX	xxxx@xxxx.cn	R	9000.05	personal income tax&0.05	Country5:City5	2014

步骤2 数据分析。

数据分析代码实现，请见**查询Hive表数据**。

- 查看薪水支付币种为美元的雇员联系方式。
- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中。
- 统计表employees_info中有多少条记录。
- 查询使用以“cn”结尾的邮箱的员工信息。

步骤3 提交数据分析任务，统计表employees_info中有多少条记录。实现请参见**使用JDBC接口提交数据分析任务**。

----结束

16.3.1.2 创建 Hive 表

功能介绍

本小节介绍了如何使用HQL创建内部表、外部表的基本操作。创建表主要有以下三种方式：

- 自定义表结构，以关键字EXTERNAL区分创建内部表和外部表。
  - 内部表，如果对数据的处理都由Hive完成，则应该使用内部表。在删除内部表时，元数据和数据一起被删除。
  - 外部表，如果数据要被多种工具共同处理，则应该使用外部表，可避免对该数据的误操作。删除外部表时，只删除掉元数据。
- 根据已有表创建新表，使用CREATE LIKE句式，完全复制原有的表结构，包括表的存储格式。
- 根据查询结果创建新表，使用CREATE AS SELECT句式。

这种方式比较灵活，可以在复制原表表结构的同时指定要复制哪些字段，不包括表的存储格式。

说明

目前表名长度最长为128，字段名长度最长为128，字段注解长度最长为4000，WITH SERDEPROPERTIES 中key长度最长为256，value长度最长为4000。以上的长度均表示字节长度。

样例代码

```
-- 创建外部表employees_info.
CREATE EXTERNAL TABLE IF NOT EXISTS employees_info
(
id INT,
name STRING,
usd_flag STRING,
salary DOUBLE,
deductions MAP<STRING, DOUBLE>,
address STRING,
entrytime STRING
)
-- 指定行中各字段分隔符.
-- "delimited fields terminated by"指定列与列之间的分隔符为','，"MAP KEYS TERMINATED BY"指定MAP中键值的分隔符为'&'.
ROW FORMAT delimited fields terminated by ',' MAP KEYS TERMINATED BY '&'
-- 指定表的存储格式为TEXTFILE.
STORED AS TEXTFILE;

-- 创建外部表employees_contact
CREATE EXTERNAL TABLE IF NOT EXISTS employees_contact
(
   id INT,
   tel_phone STRING,
   email STRING
)
ROW FORMAT delimited fields terminated by ','
STORED AS TEXTFILE;

-- 创建外部表employees_info_extended
CREATE EXTERNAL TABLE IF NOT EXISTS employees_info_extended(id INT, name STRING, usd_flag STRING, salary DOUBLE, deductions MAP<STRING, DOUBLE>, address STRING)
-- 一个表可以拥有一个或多个分区，每个分区以文件夹的形式单独存在表文件夹的目录下。对分区内数据进行查询，可缩小查询范围，加快数据的检索速度，还可对数据按照一定的条件进行管理。
-- 使用关键字PARTITIONED BY指定分区列名及数据类型
PARTITIONED BY(entrytime STRING)
ROW FORMAT delimited fields terminated by ',' MAP KEYS TERMINATED BY '&'
STORED AS TEXTFILE;
-- 一个表在创建完成后，还可以使用ALTER TABLE执行增/删字段、修改表属性、添加分区等操作。
-- 为表employees_info_extended增加“tel_phone”和“email”字段。
ALTER TABLE employees_info_extended ADD COLUMNS (tel_phone STRING, email STRING);
```

16.3.1.3 加载数据到 Hive 表中

功能介绍

本小节介绍了如何使用HQL向已有的表employees_info中加载数据。从本节中可以掌握如何从本地文件系统、MRS集群中加载数据。以关键字LOCAL区分数据源是否来自本地。

样例代码

```
-- 从本地文件系统/opt/hive_examples_data/目录下将employee_info.txt加载进employees_info表中.
---- 用新数据覆盖原有数据
```

```
LOAD DATA LOCAL INPATH '/opt/hive_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;
---- 保留原有数据，将新数据追加到表中
LOAD DATA LOCAL INPATH '/opt/hive_examples_data/employee_info.txt' INTO TABLE employees_info;

-- 从HDFS上/user/hive_examples_data/employee_info.txt加载进employees_info表中.
---- 用新数据覆盖原有数据
LOAD DATA INPATH '/user/hive_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;
---- 保留原有数据，将新数据追加到表中
LOAD DATA INPATH '/user/hive_examples_data/employee_info.txt' INTO TABLE employees_info;
```

说明

加载数据的实质是将数据复制到HDFS上指定表的目录下。

样例数据

表**employees_info**的数据如下：

```
1,Wang,R,8000.01,person&personal^Btype&income^Btax&0.05,Country1:City1,2014
3,Tom,D,12000.02,person&personal^Btype&income^Btax&0.09,Country2:City2,2014
4,Jack,D,24000.03,person&personal^Btype&income^Btax&0.05,Country3:City3,2014
6,Linda,D,36000.04,person&personal^Btype&income^Btax&0.05,Country4:City4,2014
8,Zhang,R,9000.05,person&personal^Btype&income^Btax&0.05,Country5:City5,2014
```

表**employees_contact**的数据如下：

```
1,135 XXXX XXXX,xxxx@xx.com
3,159 XXXX XXXX,xxxxx@xx.com.cn
4,186 XXXX XXXX,xxxx@xx.org
6,189 XXXX XXXX,xxxx@xxx.cn
8,134 XXXX XXXX,xxxx@xxxx.cn
```

表**employees_info_extended**的数据如下：

```
1,Wang,135 XXXX
XXXX,xxxx@xx.com,R,8000.01,person&personal^Btype&income^Btax&0.05,Country1:City1,2014
3,Tom,159 XXXX
XXXX,xxxxx@xx.com.cn,D,12000.02,person&personal^Btype&income^Btax&0.09,Country2:City2,2014
4,Jack,186 XXXX
XXXX,xxxx@xx.org,D,24000.03,person&personal^Btype&income^Btax&0.05,Country3:City3,2014
6,Linda,189 XXXX
XXXX,xxxx@xxx.cn,D,36000.04,person&personal^Btype&income^Btax&0.05,Country4:City4,2014
8,Zhang,134 XXXX
XXXX,xxxx@xxxx.cn,R,9000.05,person&personal^Btype&income^Btax&0.05,Country5:City5,2014
```

16.3.1.4 查询 Hive 表数据

功能介绍

本小节介绍了如何使用HQL对数据进行查询分析。从本节中可以掌握如下查询分析方法：

- SELECT查询的常用特性，如JOIN等。
- 加载数据进指定分区。
- 如何使用Hive自带函数。
- 如何使用自定义函数进行查询分析，如何创建、定义自定义函数请见**创建Hive用户自定义函数**。

样例代码

```
-- 查看薪水支付币种为美元的雇员联系方式.
SELECT
a.name,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b  ON(a.id = b.id) WHERE usd_flag='D';

-- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中.
INSERT OVERWRITE TABLE employees_info_extended PARTITION (entrytime = '2014')
SELECT
a.id,
a.name,
a.usd_flag,
a.salary,
a.deductions,
a.address,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE a.entrytime = '2014';

-- 使用Hive中已有的函数COUNT()，统计表employees_info中有多少条记录.
SELECT COUNT(*) FROM employees_info;

-- 查询使用以“cn”结尾的邮箱的员工信息.
SELECT a.name, b.tel_phone FROM  employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE b.email like '%cn';
```

扩展使用

- 配置Hive中间过程的数据加密

  指定表的格式为RCFile(推荐使用)或SequenceFile，加密算法为ARC4Codec。SequenceFile是Hadoop特有的文件格式，RCFile是Hive优化的文件格式。RCFile优化了列存储，在对大表进行查询时，综合性能表现比SequenceFile更优。

  ```
  set hive.exec.compress.output=true;
  set hive.exec.compress.intermediate=true;
  set hive.intermediate.compression.codec=org.apache.hadoop.io.encryption.arc4.ARC4Codec;
  ```

- 自定义函数，具体内容请参见**创建Hive用户自定义函数**。

16.3.1.5 实现 Hive 进程访问多 ZooKeeper

功能简介

FusionInsight支持在同一个客户端进程内同时访问FusionInsight ZooKeeper和第三方的ZooKeeper，分别通过“testConnectHive”和“testConnectApacheZK”方法实现。

在“hive-jdbc-example-multizk”包中的“JDBCExample”类中，main方法的代码结构如下：

```
public static void main(String[] args) throws InstantiationException,IllegalAccessException,
ClassNotFoundException, SQLException, IOException{
    testConnectHive();//访问FusionInsight ZooKeeper的方法
    testConnectApacheZk();//访问开源ZooKeeper的方法
}
```

访问 FusionInsight ZooKeeper

如果仅需运行访问FusionInsight Zookeeper方法，需注释掉main函数中的“testConnectApacheZk”方法。

使用“testConnectHive”方法访问FusionInsight ZooKeeper前需执行如下操作：

步骤1 修改JDBCExample中“init”方法中的“USER_NAME”参数的值。“USER_NAME”对应的用户用于访问FusionInsight ZooKeeper，需拥有FusionInsight Hive、Hadoop普通用户组权限。

步骤2 进入客户端解压路径“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles\Hive\config”，手动将“core-site.xml”、“hiveclient.properties”文件放到样例工程的“hive-jdbc-example-multizk\src\main\resources”目录下。

步骤3 检查并修改resources目录下hiveclient.properties文件中“zk.port”和“zk.quorum”参数的值：

- zk.port：为访问FusionInsight ZooKeeper的端口，通常保持默认，根据实际使用情况修改。
- zk.quorum：为访问ZooKeeper quorumpeer的地址，请修改为集群部署有FusionInsight ZooKeeper服务的IP地址。

----结束

访问开源 ZooKeeper

使用“testConnectApacheZk”连接开源ZooKeeper的代码，只需要将以下代码中的“xxx.xxx.xxx.xxx”修改为需要连接的开源的ZooKeeper的IP，端口号按照实际情况修改。如果仅需运行访问第三方Zookeeper的样例，需注释掉main函数中的“testConnectHive”方法。

```
digestZK = new org.apache.zookeeper.ZooKeeper("xxx.xxx.xxx.xxx:端口号", 60000, null);
```

说明

ZooKeeper连接使用完后需要关闭连接，否则可能导致连接泄露。可根据业务实际情况进行处理，代码如下：

```
//使用try-with-resources方式，try语句执行完后会自动关闭ZooKeeper连接。
try (org.apache.zookeeper.ZooKeeper digestZk =
    new org.apache.zookeeper.ZooKeeper("xxx.xxx.xxx.xxx:端口号", 600000, null)) {
    ...
}
```

16.3.1.6 使用 JDBC 接口提交数据分析任务

功能简介

本章节介绍如何使用JDBC样例程序完成数据分析任务。

样例代码

使用Hive JDBC接口提交数据分析任务，该样例程序在“hive-examples/hive-jdbc-example”的“JDBCExample.java”中，实现该功能的模块如下：

1. 读取HiveServer客户端property文件，其中“hiveclient.properties”文件在“hive-jdbc-example\src\main\resources”目录下。

```
Properties clientInfo = null;
String userdir = System.getProperty("user.dir") + File.separator
+ "conf" + File.separator;
InputStream fileInputStream = null;
try{
clientInfo = new Properties();
//"hiveclient.properties"为客户端配置文件
```

```
//"hiveclient.properties"文件可从对应实例客户端安装包解压目录下的config目录下获取，并上传到JDBC样例工程的“hive-jdbc-example\src\main\resources”目录下
String hiveclientProp = userdir + "hiveclient.properties" ;
File propertiesFile = new File(hiveclientProp);
fileInputStream = new FileInputStream(propertiesFile);
clientInfo.load(fileInputStream);
}catch (Exception e) {
throw new IOException(e);
}finally{
if(fileInputStream != null){
fileInputStream.close();
fileInputStream = null;
}
}
```

2. 获取ZooKeeper的IP列表和端口、集群的认证模式、HiveServer的SASL配置、HiveServer在ZooKeeper中节点名称、客户端对服务端的发现模式、以及服务端进程认证的principal。这些配置样例代码会自动从“hiveclient.properties中”读取。

```
//zkQuorum获取后的格式为"xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181";
//"xxx.xxx.xxx.xxx"为集群中ZooKeeper所在节点的业务IP，端口默认是2181
   zkQuorum = clientInfo.getProperty("zk.quorum");
   auth = clientInfo.getProperty("auth");
   sasl_qop = clientInfo.getProperty("sasl.qop");
   zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
   serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
   principal = clientInfo.getProperty("principal");
```

3. 定义HQL。HQL必须为单条语句，注意HQL不能包含“;”。

```
  // 定义HQL，不能包含“;”
  String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
        "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};
```

4. 拼接JDBC URL。

   说明

   拼接JDBC URL也可以不提供账户和keytab路径，采用提前认证的方式。如果使用IBM JDK运行Hive应用程序，则必须使用“JDBC代码样例二”提供的预认证方式才能访问。

   以下代码片段，拼接完成后的JDBC URL示例为：

   ```
   jdbc:hive2://
   xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2;sasl.qop=auth-conf;auth=KERBEROS;principal=hive/hadoop.<系统域名>@<系统域名>;
   ```

   系统域名可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数获取。

   “hive/hadoop.*<系统域名>*”为用户名，用户的用户名所包含的系统域名所有字母为小写。例如“本端域”参数为“9427068F-6EFA-4833-B43E-60CB641E5B6C.COM”，则用户名为“hive/hadoop.9427068f-6efa-4833-b43e-60cb641e5b6c.com”。

```
  // 拼接JDBC URL
    // 普通模式
    sBuilder.append(";serviceDiscoveryMode=")
          .append(serviceDiscoveryMode)
          .append(";zooKeeperNamespace=")
          .append(zooKeeperNamespace)
          .append(";auth=none;");
```

5. 加载Hive JDBC驱动。

```
  // 加载Hive JDBC驱动
  Class.forName(HIVE_DRIVER);
```

6. 获取JDBC连接，确认HQL的类型（DDL/DML），调用对应的接口执行HQL，输出查询的列名和结果到控制台，关闭JDBC连接。

```
  Connection connection = null;
    try {
      // 获取JDBC连接
```

```
      connection = DriverManager.getConnection(url, "", "");

      // 建表
      // 表建完之后，如果要往表中导数据，可以使用LOAD语句将数据导入表中，比如从HDFS上将数据导入表:
      //load data inpath '/tmp/employees.txt' overwrite into table employees_info;
      execDDL(connection,sqls[0]);
      System.out.println("Create table success!");

      // 查询
      execDML(connection,sqls[1]);

      // 删表
      execDDL(connection,sqls[2]);
      System.out.println("Delete table success!");
    }
    finally {
      // 关闭JDBC连接
      if (null != connection) {
        connection.close();
      }

public static void execDDL(Connection connection, String sql)
  throws SQLException {
    PreparedStatement statement = null;
    try {
      statement = connection.prepareStatement(sql);
      statement.execute();
    }
    finally {
      if (null != statement) {
        statement.close();
      }
    }
  }

  public static void execDML(Connection connection, String sql) throws SQLException {
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    ResultSetMetaData resultMetaData = null;

    try {
      // 执行HQL
      statement = connection.prepareStatement(sql);
      resultSet = statement.executeQuery();

      // 输出查询的列名到控制台
      resultMetaData = resultSet.getMetaData();
      int columnCount = resultMetaData.getColumnCount();
      for (int i = 1; i <= columnCount; i++) {
        System.out.print(resultMetaData.getColumnLabel(i) + '\t');
      }
      System.out.println();

      // 输出查询结果到控制台
      while (resultSet.next()) {
        for (int i = 1; i <= columnCount; i++) {
          System.out.print(resultSet.getString(i) + '\t');
        }
        System.out.println();
      }
    }
    finally {
      if (null != resultSet) {
        resultSet.close();
      }

      if (null != statement) {
```

```
        statement.close();
      }
    }
  }
```

16.3.2 HCatalog 访问 Hive 样例程序

功能介绍

本章节介绍如何在MapReduce任务中使用HCatalog分析Hive表数据，读取输入表第一列**int**类型数据执行**count(distinct XX)**操作，将结果写入输出表。

样例代码

该样例程序在“hive-examples/hcatalog-example”的“HCatalogExample.java”中，实现该功能的模块如下：

1. 实现Mapper类，通过HCatRecord获取第一列int类型数据，计数1并输出；

```
public static class Map extends
        Mapper<LongWritable, HCatRecord, IntWritable, IntWritable> {
    int age;
    @Override
    protected void map(
            LongWritable key,
            HCatRecord value,
            Mapper<LongWritable, HCatRecord,
                    IntWritable, IntWritable>.Context context)
            throws IOException, InterruptedException {
        if ( value.get(0) instanceof Integer ) {
            age = (Integer) value.get(0);
        }
        context.write(new IntWritable(age), new IntWritable(1));
    }
}
```

2. 实现Reducer类，将map输出结果合并计数，统计不重复的值出现次数，使用HCatRecord输出结果；

```
public static class Reduce extends Reducer<IntWritable, IntWritable,
        IntWritable, HCatRecord> {
    @Override
    protected void reduce(
            IntWritable key,
            Iterable<IntWritable> values,
            Reducer<IntWritable, IntWritable,
                    IntWritable, HCatRecord>.Context context)
            throws IOException, InterruptedException {
        int sum = 0;
        Iterator<IntWritable> iter = values.iterator();
        while (iter.hasNext()) {
            sum++;
            iter.next();
        }
        HCatRecord record = new DefaultHCatRecord(2);
        record.set(0, key.get());
        record.set(1, sum);
        context.write(null, record);
    }
}
```

3. MapReduce任务定义，指定输入/输出类，Mapper/Recducer类，输入输出键值对格式；

```
Job job = new Job(conf, "GroupByDemo");
HCatInputFormat.setInput(job, dbName, inputTableName);
job.setInputFormatClass(HCatInputFormat.class);
job.setJarByClass(HCatalogExample.class);
```

```
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setMapOutputKeyClass(IntWritable.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(WritableComparable.class);
job.setOutputValueClass(DefaultHCatRecord.class);
String outputTableName = otherArgs[1];
OutputJobInfo outputjobInfo = OutputJobInfo.create(dbName, outputTableName, null);
HCatOutputFormat.setOutput(job, outputjobInfo);
HCatSchema schema = outputjobInfo.getOutputSchema();
HCatOutputFormat.setSchema(job, schema);
job.setOutputFormatClass(HCatOutputFormat.class);
```

16.3.3 基于 Python 的 Hive 样例程序

功能介绍

本章节介绍如何使用Python连接Hive执行数据分析任务。

样例代码

使用Python方式提交数据分析任务，参考样例程序中的“hive-examples/python-examples/pyCLI_sec.py”。

1. 导入HAConnection类。
   ```
   from pyhs2.haconnection import HAConnection
   ```
2. 声明HiveServer的IP地址列表。本例中hosts代表HiveServer的节点，xxx.xxx.xxx.xxx代表业务IP地址。
   ```
   hosts = ["xxx.xxx.xxx.xxx", "xxx.xxx.xxx.xxx"]
   ```

   说明

   如果HiveServer实例被迁移，原始的示例程序会失效。在HiveServer实例迁移之后，用户需要更新示例程序中使用的HiveServer的IP地址。
3. 在HAConnection的第三个参数填写正确的用户名，密码可以不填写。创建连接，执行HQL，样例代码中仅执行查询所有表功能，可根据实际情况修改HQL内容，输出查询的列名和结果到控制台。
   ```
   try:
     with HAConnection(hosts = hosts,
                       port = 21066,
                       authMechanism = "PLAIN",
                       user='root',
                       password='******') as haConn:
       with haConn.getConnection() as conn:
         with conn.cursor() as cur:
           # Show databases
           print cur.getDatabases()
           # Execute query
           cur.execute("show tables")
           # Return column info from query
           print cur.getSchema()
           # Fetch table results
           for i in cur.fetch():
             print i
   except Exception, e:
     print e
   ```

16.3.4 基于 Python3 的 Hive 样例程序

功能介绍

本章节介绍如何使用Python3连接Hive执行数据分析任务。

样例代码

以下分析任务示例在“hive-examples/python3-examples/pyCLI_nosec.py”文件中。

1. 导入hive类
```
from pyhive import hive
```
2. 创建JDBC连接：
```
connection = hive.Connection(host='hiveserverIp', port=hiveserverPort, username='hive',
database='default', auth=None, kerberos_service_name=None, krbhost=None)
```
需按照实际环境修改以下参数：
   - hiveserverIp：替换为实际需要连接的HiveServer节点IP地址，可登录FusionInsight Manager，选择“集群 > 服务 > Hive > 实例”查看。
   - hiveserverPort：需要替换为Hive服务的端口，可在FusionInsight Manager界面，选择“集群 > 服务 > Hive > 配置”，在搜索框中搜索“hive.server2.thrift.port”查看，默认值为“10000”。
3. 执行SQL语句，样例代码中仅执行查询所有表功能，可根据实际情况修改HQL内容。
```
cursor = connection.cursor()
cursor.execute('show tables')
```
4. 获取结果并输出
```
for result in cursor.fetchall():
    print(result)
```

16.4 调测 Hive 应用

16.4.1 在 Windows 环境中调测 Hive JDBC 样例程序

在程序代码完成开发后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

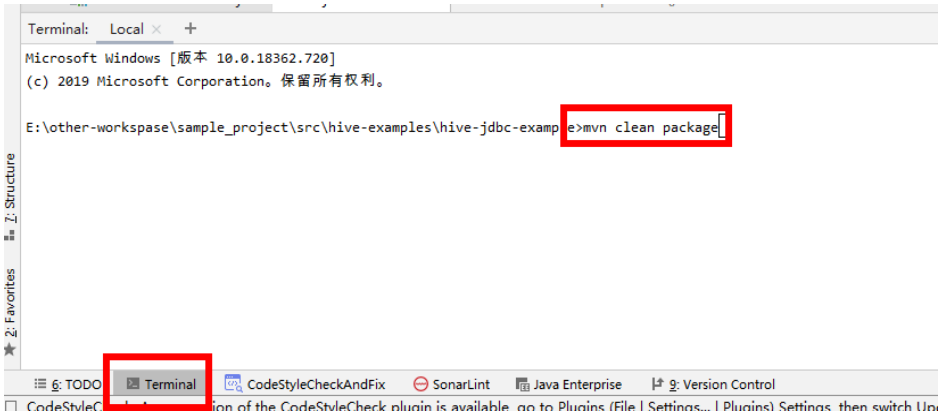
说明

- 如果Windows开发环境中使用IBM JDK，不支持在Windows环境中直接运行应用程序。
- 需要在运行样例代码的本机**hosts**文件中设置访问节点的主机名和公网IP地址映射，主机名和公网IP地址请保持一一对应。
- 仅JDBC样例程序支持在本地Windows中运行。

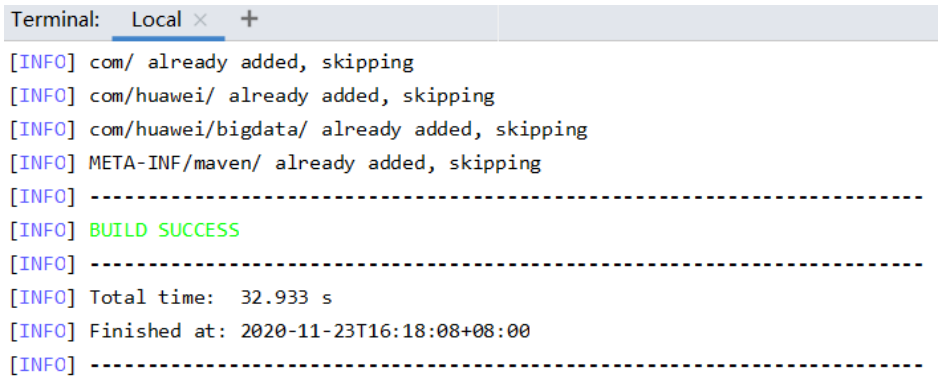
操作步骤

步骤1 编译JDBC样例程序：

在IDEA界面左下方单击“Terminal”进入终端，执行命令**mvn clean package**进行编译。



当输出“BUILD SUCCESS”，表示编译成功，如下图所示。编译成功后将会在样例工程的target下生成含有“-with-dependencies”字段的jar包。



步骤2 运行JDBC样例程序：

- 使用Windows命令行形式运行JDBC样例工程：
  a. 在Windows上创建一个目录作为运行目录，如“D:\jdbc_example”，将**步骤1**中生成的“target”目录下包名中含有“-with-dependencies”字段的Jar包放到该路径下，并在该目录下创建子目录“src/main/resources”，将已获取的“hive-jdbc-example\src\main\resources”目录下的所有文件复制到“resources”下。
  b. 执行以下命令运行Jar包：

     cd /d d:\jdbc_example

     java -jar *hive-jdbc-example-1.0-SNAPSHOT-jar-with-dependencies.jar*

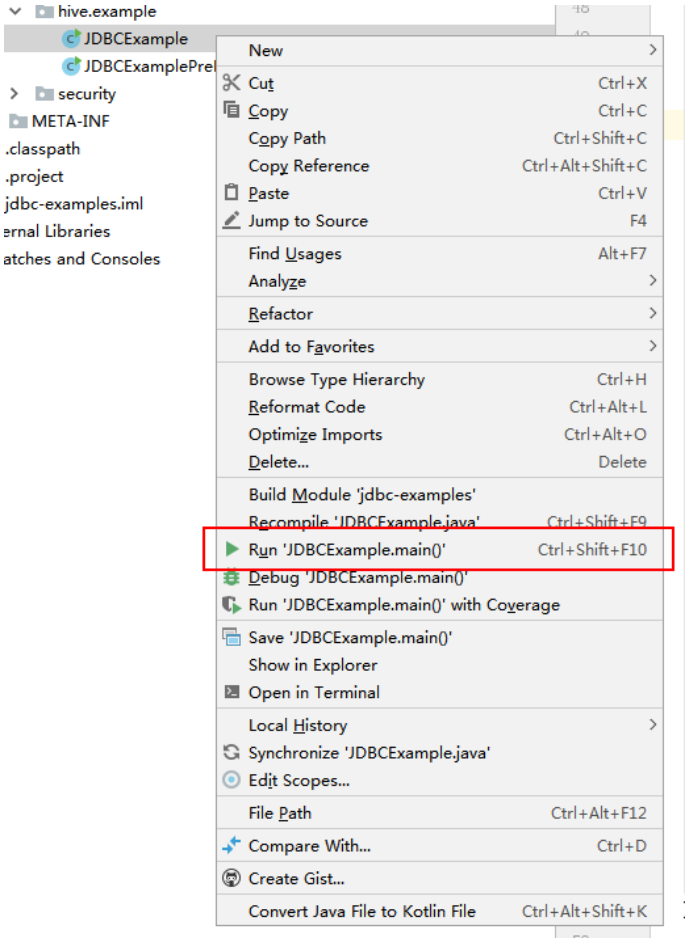
     说明

     以上Jar包名称仅供参考，具体名称以实际生成为主。

  c. 在命令行终端查看样例代码中的HQL所查询出的结果，运行成功结果会有如下信息：

     ```
     Create table success!
     _c0
     0
     Delete table success!
     ```

- 使用IntelliJ IDEA形式运行JDBC样例工程：
  a. 在IntelliJ IDEA的jdbc-examples工程的“JDBCExample”类上单击右键，在弹出菜单中选择“Run JDBCExample.main()”，如下图所示：



b. 在IntelliJ IDEA输出窗口查看样例代码中的HQL所查询出的结果，会有如下信息：

```
Create table success!
_c0
0
Delete table success!
```

----结束

16.4.2 在 Linux 环境中调测 Hive JDBC 样例程序

Hive JDBC应用程序支持在安装Hive客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。

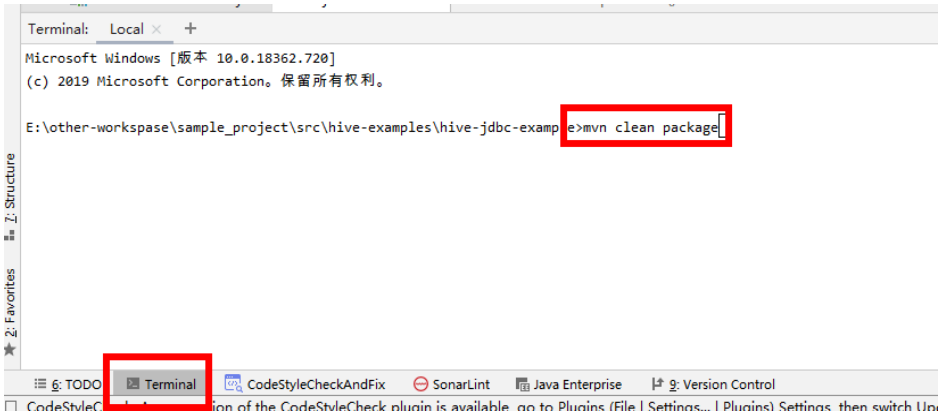
前提条件

- 已安装Hive客户端。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

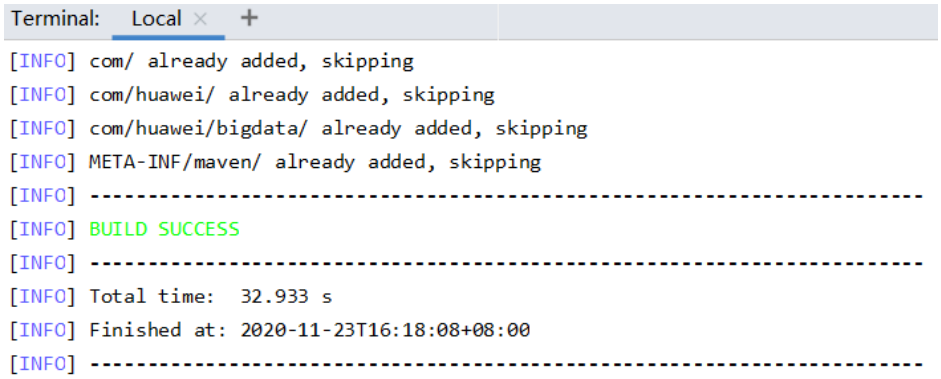
操作步骤

步骤1 编译JDBC样例程序：

在IDEA界面左下方单击“Terminal”进入终端，执行命令**mvn clean package**进行编译。



当输出“BUILD SUCCESS”，表示编译成功，如下图所示。编译成功后将会在样例工程的target下生成含有“-with-dependencies”字段的jar包。



步骤2 运行JDBC样例程序：

1. 在Linux上创建一个目录作为运行目录，如“/opt/jdbc_example”，将**步骤1**中生成的“target”目录下包名中含有“-with-dependencies”字段的Jar包放进该路径下，并在该目录下创建子目录“src/main/resources”，将已获取的“hive-jdbc-example\src\main\resources”目录下的所有文件复制到“resources”下。
2. 执行以下命令运行Jar包：

   chmod +x /opt/jdbc_example -R

   cd /opt/jdbc_example

   java -jar *hive-jdbc-example-1.0-SNAPSHOT-jar-with-dependencies.jar*

   说明

   以上Jar包名称仅供参考，具体名称以实际生成为主。
3. 在命令行终端查看样例代码中的HQL所查询出的结果，运行成功会显示如下信息：

   ```
   Create table success!
   _c0
   0
   Delete table success!
   ```

----结束

16.4.3 调测 Hive HCatalog 样例程序

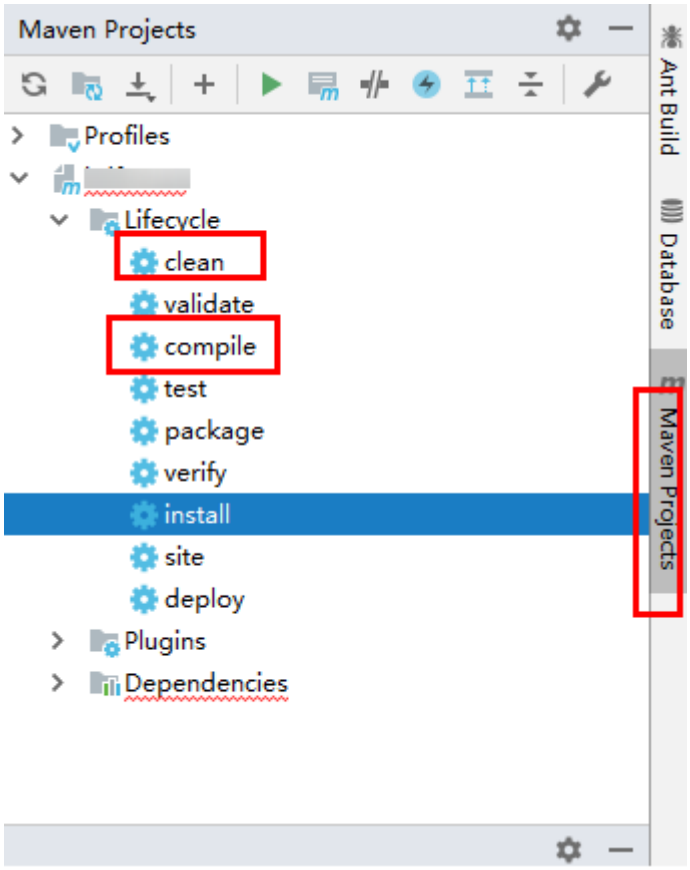
Hive HCatalog应用程序支持在安装Hive和Yarn客户端的Linux环境中运行。在程序代码完成开发后，您可以上传Jar包至准备好的Linux运行环境中运行。

前提条件

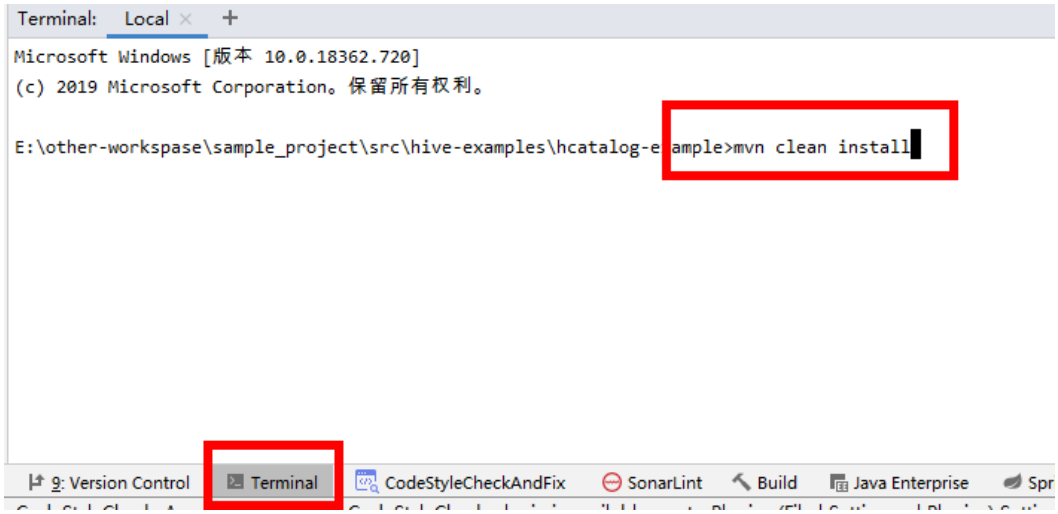
- 已安装Hive和Yarn客户端。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

操作步骤

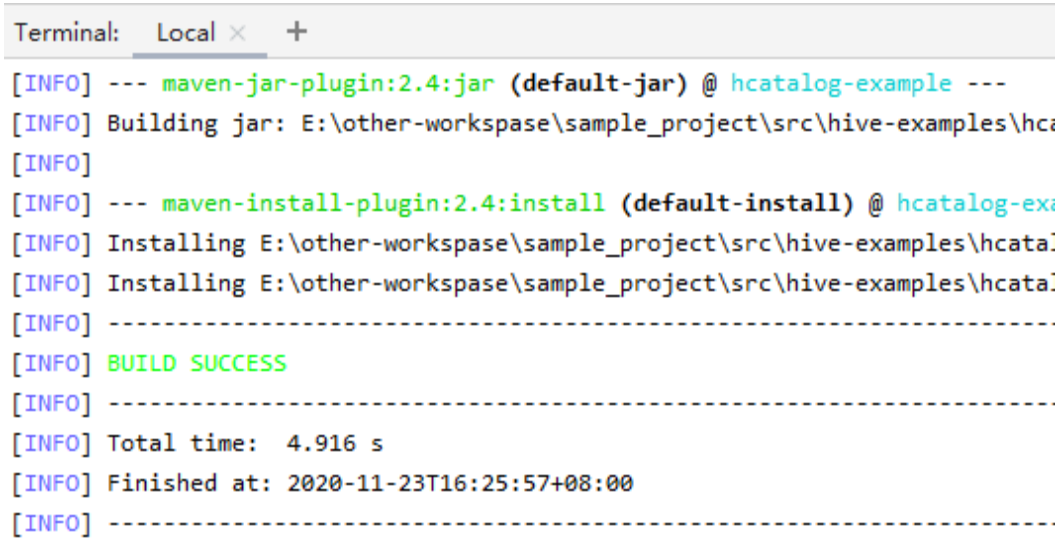
步骤1 在IntelliJ IDEA主界面右侧，单击“Maven Projects”，在“Maven Projects”界面执行“*项目名称* > Lifecycle”目录下的“clean”和“compile”脚本。



步骤2 在IDEA界面左下方找到Terminal，单击进入终端，执行**mvn clean install**进行编译



当输出“BUILD SUCCESS”，表示编译成功，如下图所示。编译成功后将会在样例工程的target下生成含有“hcatalog-example-*.jar”包。



说明

以上jar包名称仅供参考，具体名称以实际生成为主。

步骤3 将步骤2中在样例工程下的target下生成的“hcatalog-example-*.jar”上传至Linux的指定路径，例如“/opt/hive_client”，记作“$HCAT_CLIENT”，并确保已经安装好Hive和Yarn客户端。运行环境变量使HCAT_CLIENT生效。

```
export HCAT_CLIENT=/opt/hive_client
```

步骤4 执行以下命令用于配置环境变量信息（以客户端安装路径为“/opt/client”为例）：

```
export HADOOP_HOME=/opt/client/HDFS/hadoop
export HIVE_HOME=/opt/client/Hive/Beeline
export HCAT_HOME=$HIVE_HOME/../HCatalog
export LIB_JARS=$HCAT_HOME/lib/hive-hcatalog-core-xxx.jar,$HCAT_HOME/lib/hive-metastore-xxx.jar,$HCAT_HOME/lib/hive-standalone-metastore-xxx.jar,$HIVE_HOME/lib/hive-exec-xxx.jar,$HCAT_HOME/lib/libfb303-xxx.jar,$HCAT_HOME/lib/slf4j-api-xxx.jar,$HCAT_HOME/lib/jdo-api-xxx.jar,$HCAT_HOME/lib/antlr-runtime-xxx.jar,$HCAT_HOME/lib/datanucleus-api-jdo-xxx.jar,$HCAT_HOME/lib/datanucleus-core-xxx.jar,$HCAT_HOME/lib/datanucleus-rdbms-fi-xxx.jar,$HCAT_HOME/lib/log4j-api-xxx.jar,$HCAT_HOME/lib/log4j-core-xxx.jar,$HIVE_HOME/lib/commons-lang-xxx.jar
export HADOOP_CLASSPATH=$HCAT_HOME/lib/hive-hcatalog-core-xxx.jar:$HCAT_HOME/lib/hive-metastore-xxx.jar:$HCAT_HOME/lib/hive-standalone-metastore-xxx.jar:$HIVE_HOME/lib/hive-exec-xxx.jar:$HCAT_HOME/lib/libfb303-xxx.jar:$HADOOP_HOME/etc/hadoop:$HCAT_HOME/conf:$HCAT_HOME/lib/slf4j-api-xxx.jar:$HCAT_HOME/lib/jdo-api-xxx.jar:$HCAT_HOME/lib/antlr-runtime-xxx.jar:$HCAT_HOME/lib/datanucleus-api-jdo-xxx.jar:$HCAT_HOME/lib/datanucleus-core-xxx.jar:$HCAT_HOME/lib/datanucleus-rdbms-fi-xxx.jar:$HCAT_HOME/lib/log4j-api-xxx.jar:$HCAT_HOME/lib/log4j-core-xxx.jar:$HIVE_HOME/lib/commons-lang-xxx.jar
```

说明

xxx：表示Jar包的版本号。**LIB_JARS和HADOOP_CLASSPATH中指定的jar包的版本号需要根据实际环境的版本号进行修改**。

步骤5 运行前准备：

1. 使用Hive客户端，在beeline中执行以下命令创建源表t1：

   create table t1(col1 int);

   向t1中插入如下数据：


```
+----------+--+
| t1.col1  |
+----------+--+
| 1        |
| 1        |
| 1        |
| 2        |
| 2        |
| 3        |
```

2. 执行以下命令创建目的表t2：

   create table t2(col1 int,col2 int);

   说明

   本样例工程中创建的表使用Hive默认的存储格式，暂不支持指定存储格式为ORC的表。

步骤6 执行以下命令使用Yarn客户端提交任务：

yarn --config $HADOOP_HOME/etc/hadoop jar $HCAT_CLIENT/hcatalog-example-1.0-SNAPSHOT.jar com.huawei.bigdata.HCatalogExample -libjars $LIB_JARS t1 t2

步骤7 运行结果查看，运行后t2表数据如下所示：

```
0: jdbc:hive2://192.168.1.18:2181,192.168.1.> select * from t2;
+----------+----------+--+
| t2.col1  | t2.col2  |
+----------+----------+--+
| 1        | 3        |
| 2        | 2        |
| 3        | 1        |
+----------+----------+--+
```

----结束

16.4.4 调测 Hive Python 样例程序

Python 样例工程的命令行形式运行

步骤1 赋予“python-examples”文件夹中脚本的可执行权限。在命令行终端执行以下命令：

chmod +x python-examples -R

步骤2 在“python-examples/pyCLI_nosec.py”中的hosts数组中填写安装HiveServer的节点的业务平面IP地址。HiveServer业务平面IP地址可登录FusionInsight Manager，选择“集群 > 服务 > Hive > 实例”查看。

步骤3 执行以下命令运行Python客户端：

cd python-examples

python pyCLI_nosec.py

步骤4 在命令行终端查看样例代码中的HQL所查询出的结果。

例如：

```
[['default', '']]
 [{'comment': 'from deserializer', 'columnName': 'tab_name', 'type': 'STRING_TYPE'}]
 ['xx']
```

说明

如果出现如下异常：

```
importError: libsasl2.so.2: cannot open shared object file: No such file or directory
```

请按照以下方式处理：

1. 首先执行如下命令，查询所装操作系统中LibSASL的版本
   ```
   ldconfig -p|grep sasl
   ```
   结果如下则表示当前操作系统仅存在3.x版本
   ```
   libsasl2.so.3 (libc6,x86-64) => /usr/lib64/libsasl2.so.3
   libsasl2.so.3 (libc6) => /usr/lib/libsasl2.so.3
   ```
2. 如果仅存在3.x版本，需要执行如下命令创建软链接
   ```
   ln -s /usr/lib64/libsasl2.so.3.0.0 /usr/lib64/libsasl2.so.2
   ```

----结束

16.4.5 调测 Hive Python3 样例程序

Python3 样例工程的命令行形式运行

步骤1 赋予“python3-examples”文件夹中脚本的可执行权限。在命令行终端执行以下命令：

chmod +x python3-examples -R。

步骤2 将“python3-examples/pyCLI_nosec.py”中的host的值修改为安装HiveServer的节点的业务平面IP地址，port的值修改为Hive提供Thrift服务的端口。

说明

- HiveServer业务平面IP地址可登录FusionInsight Manager，选择“集群 > 服务 > Hive > 实例”查看。
- port可在FusionInsight Manager界面，选择“集群 > 服务 > Hive > 配置”，在搜索框中搜索“hive.server2.thrift.port”查看，默认值为“10000”。

步骤3 执行以下命令运行Python3客户端：

cd python3-examples

python pyCLI_nosec.py

步骤4 在命令行终端查看样例代码中的HQL所查询出的结果。例如：

```
('table_name1',)
('table_name2',)
('table_name3',)
('table_name4',)
('table_name5',)
```

其中，“table_name*X*”表示实际的表名。

----结束

16.5 Hive 应用开发常见问题

16.5.1 Hive 对外接口介绍

16.5.1.1 Hive JDBC 接口介绍

Hive JDBC接口遵循标准的JAVA JDBC驱动标准。

说明

Hive作为数据仓库类型数据库，其并不能支持所有的JDBC标准API。例如事务类型的操作：rollback、setAutoCommit等，执行该类操作会获得“Method not supported”的SQLException异常。

16.5.1.2 Hive WebHCat 接口介绍

说明

- 以下示例的IP为WebHCat的业务IP，端口为安装时设置的WebHCat HTTP端口。
- 除“:version”、“status”、“version”、“version/hive”、“version/hadoop”以外，其他API都需要添加user.name参数。

1. :version(GET)
   - 描述

     查询WebHCat支持的返回类型列表。
   - URL

     http://www.myserver.com/templeton/:version
   - 参数

参数	描述
:version	WebHCat版本号（当前必须是v1）。

   - 返回结果

参数	描述
responseTypes	WebHCat支持的返回类型列表。

   - 例子

     ```
     curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1'
     ```
2. status (GET)
   - 描述

     获取当前服务器的状态
   - URL

     http://www.myserver.com/templeton/v1/status
   - 参数

     无
   - 返回结果

参数	描述
status	WebHCat连接正常，返回OK。

参数	描述
version	字符串，包含版本号，比如v1。

- 例子
  ```
  curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/status'
  ```

3. version (GET)
   - 描述

     获取服务器WebHCat的版本
   - URL

     http://www.myserver.com/templeton/v1/version
   - 参数

     无
   - 返回结果

参数	描述
supportedVersions	所有支持的版本
version	当前服务器WebHCat的版本

   - 例子
     ```
     curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/version'
     ```

4. version/hive (GET)
   - 描述

     获取服务器Hive的版本
   - URL

     http://www.myserver.com/templeton/v1/version/hive
   - 参数

     无
   - 返回结果

参数	描述
module	hive
version	Hive的版本

   - 例子
     ```
     curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/version/hive'
     ```

5. version/hadoop (GET)
   - 描述

     获取服务器Hadoop的版本
   - URL

     http://www.myserver.com/templeton/v1/version/hadoop

- 参数

  无

- 返回结果

参数	描述
module	hadoop
version	Hadoop的版本

- 例子

```
curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/version/hadoop'
```

6. ddl (POST)
   - 描述

     执行DDL语句

   - URL

     http://www.myserver.com/templeton/v1/ddl

   - 参数

参数	描述
exec	需要执行的HCatalog DDL语句。
group	当DDL是创建表时，创建表使用的用户组。
permissions	当DDL是创建表时，创建表使用的权限，格式为rwxr-xr-x。

   - 返回结果

参数	描述
stdout	HCatalog执行时的标准输出值，可能为空。
stderr	HCatalog执行时的错误输出，可能为空。
exitcode	HCatalog的返回值。

   - 例子

```
curl -i -u : --negotiate -d exec="show tables" 'http://10.64.35.144:9111/templeton/v1/ddl?user.name=user1'
```

7. ddl/database (GET)
   - 描述

     列出所有的数据库

   - URL

     http://www.myserver.com/templeton/v1/ddl/database

   - 参数

参数	描述
like	用来匹配数据库名的正则表达式。

- 返回结果

参数	描述
databases	数据库名

- 例子

```
curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database?user.name=user1'
```

8. ddl/database/:db (GET)
   - 描述

     获取指定数据库的详细信息
   - URL

     http://www.myserver.com/templeton/v1/ddl/database/:db
   - 参数

参数	描述
:db	数据库名

   - 返回结果

参数	描述
location	数据库位置
comment	数据库的备注，如果没有备注则不存在
database	数据库名
owner	数据库的所有者
owertype	数据库所有者的类型

   - 例子

```
curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database/default?
user.name=user1'
```

9. ddl/database/:db (PUT)
   - 描述

     创建数据库
   - URL

     http://www.myserver.com/templeton/v1/ddl/database/:db
   - 参数

参数	描述
:db	数据库名
group	创建数据库时使用的用户组
permission	创建数据库时使用的权限
location	数据库的位置
comment	数据库的备注，比如描述
properties	数据库属性

- 返回结果

参数	描述
database	新创建的数据库的名字

- 例子

```
curl -i -u : --negotiate -X PUT -HContent-type:application/json -d '{"location": "/tmp/a", "comment": "my db", "properties": {"a": "b"}}' 'http://10.64.35.144:9111/templeton/v1/ddl/database/db2?user.name=user1'
```

10. ddl/database/:db (DELETE)
    - 描述

      删除数据库
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db
    - 参数

参数	描述
:db	数据库名
ifExists	如果指定数据库不存在，Hive会返回错误，除非设置了ifExists为true。
option	将参数设置成cascade或者restrict。如果选择cascade，将清除一切，包括数据和定义。如果选择restrict，表格内容为空，模式也将不存在。

    - 返回结果

参数	描述
database	删除的数据库名字

    - 例子

```
curl -i -u : --negotiate -X DELETE 'http://10.64.35.144:9111/templeton/v1/ddl/database/db3?ifExists=true&user.name=user1'
```

11. ddl/database/:db/table (GET)
    - 描述

      列出数据库下的所有表
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table
    - 参数

参数	描述
:db	数据库名
like	用来匹配表名的正则表达式

    - 返回结果

参数	描述
database	数据库名字
tables	数据库下表名列表

    - 例子

      ```
      curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/table?user.name=user1'
      ```

12. ddl/database/:db/table/:table (GET)
    - 描述

      获取表的详细信息
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table
    - 参数

参数	描述
:db	数据库名
:table	表名
format	格式：format=extended，参考额外信息（“table extended like”）。

    - 返回结果

参数	描述
columns	列名和类型
database	数据库名
table	表名

参数	描述
partitioned	是否分区表，只有extended下才会显示。
location	表的位置，只有extended下才会显示。
outputformat	输出形式，只有extended下才会显示。
inputformat	输入形式，只有extended下才会显示。
owner	表的属主，只有extended下才会显示。
partitionColumns	分区的列，只有extended下才会显示。

- 例子

```
curl -i -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1?format=extended&user.name=user1'
```

13. ddl/database/:db/table/:table (PUT)
    - 描述

      创建表
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table
    - 参数

参数	描述
:db	数据库名
:table	新建表名
group	创建表时使用的用户组
permissions	创建表时使用的权限
external	指定位置，hive不使用表的默认位置。
ifNotExists	设置为true，当表存在时不会报错。
comment	备注
columns	列描述，包括列名，类型和可选备注。
partitionedBy	分区列描述，用于划分表格。参数columns列出了列名，类型和可选备注。

参数	描述
clusteredBy	分桶列描述，参数包括columnNames、sortedBy、和numberOfBuckets。参数columnNames包括columnName和排列顺序（ASC为升序，DESC为降序）。
format	存储格式，参数包括rowFormat，storedAs，和storedBy。
location	HDFS路径
tableProperties	表属性和属性值（name-value对）

- 返回结果

参数	描述
database	数据库名
table	表名

- 例子

```
curl -i -u : --negotiate -X PUT -HContent-type:application/json -d '{"columns": [{"name": "id", "type": "int"}, {"name": "name","type": "string"}], "comment": "hello","format": {"storedAs": "orc"} }' 'http://10.64.35.144:9111/templeton/v1/ddl/database/db3/table/tbl1?user.name=user1'
```

14. ddl/database/:db/table/:table (POST)
    - 描述

      重命名表
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table
    - 参数

参数	描述
:db	数据库名
:table	已有表名
rename	新表表名

    - 返回结果

参数	描述
database	数据库名
table	新表表名

    - 例子


```
curl -i -u : --negotiate -d rename=table1 'http://10.64.35.144:9111/templeton/v1/ddl/database/
default/table/tbl1?user.name=user1'
```

15. ddl/database/:db/table/:table (DELETE)
    - 描述

      删除表
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table
    - 参数

参数	描述
:db	数据库名
:table	表名
ifExists	当设置为true时，不报错。

    - 返回结果

参数	描述
database	数据库名
table	表名

    - 例子

```
curl -i -u : --negotiate -X DELETE 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/
table/table2?ifExists=true&user.name=user1'
```

16. ddl/database/:db/table/:existingtable/like/:newtable (PUT)
    - 描述

      创建一张和已经存在的表一样的表
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/
      table/:existingtable/like/:newtable
    - 参数

参数	描述
:db	数据库名
:existingtable	已有表名
:newtable	新表名
group	创建表时使用的用户组。
permissions	创建表时使用的权限。
external	指定位置，hive不使用表的默认位置。

参数	描述
ifNotExists	当设置为true时，如果表已经存在，Hive不报错。
location	HDFS路径

- 返回结果

参数	描述
database	数据库名
table	表名

- 例子

```
curl -i -u : --negotiate -X PUT -HContent-type:application/json -d '{"ifNotExists": "true"}' 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/like/tt1?user.name=user1'
```

17. ddl/database/:db/table/:table/partition(GET)
    - 描述

      列出表的分区信息
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition
    - 参数

参数	描述
:db	数据库名
:table	表名

    - 返回结果

参数	描述
database	数据库名
table	表名
partitions	分区属性值和分区名

    - 例子

```
curl -i -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition?user.name=user1
```

18. ddl/database/:db/table/:table/partition/:partition(GET)
    - 描述

      列出表的某个具体分区的信息
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition/:partition

- 参数

参数	描述
:db	数据库名
:table	表名
:partition	分区名，解码http引用时，需当心。比如country=%27algeria%27。

- 返回结果

参数	描述
database	数据库名
table	表名
partition	分区名
partitioned	如果设置为true，为分区表
location	表的存储路径
outputFormat	输出格式
columns	列名，类型，备注
owner	所有者
partitionColumns	分区的列
inputFormat	输入格式
totalNumberFiles	分区下文件个数
totalFileSize	分区下文件总大小
maxFileSize	最大文件大小
minFileSize	最小文件大小
lastAccessTime	最后访问时间
lastUpdateTime	最后更新时间

- 例子

```
curl -i -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition/dt=1?user.name=user1
```

19. ddl/database/:db/table/:table/partition/:partition(PUT)
    - 描述

      增加一个表分区
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition/:partition

- 参数

参数	描述
:db	数据库名。
:table	表名。
group	创建新分区时使用的用户组。
permissions	创建新分区时用户的权限。
location	新分区的存放位置。
ifNotExists	如果设置为true， 当分区已经存在，系统报错。

- 返回结果

参数	描述
database	数据库名
table	表名
partitions	分区名

- 例子

```
curl -i -u : --negotiate -X PUT -HContent-type:application/json -d '{}' http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition/dt=10?user.name=user1
```

20. ddl/database/:db/table/:table/partition/:partition(DELETE)
    - 描述

      删除一个表分区
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/partition/:partition
    - 参数

参数	描述
:db	数据库名。
:table	表名。
group	删除新分区时使用的用户组。
permissions	删除新分区时用户的权限， 格式为rwxrw-r-x。
ifExists	如果指定分区不存在，Hive报错。参数值设置为true除外。

    - 返回结果

参数	描述
database	数据库名
table	表名
partitions	分区名

- 例子
  ```
  curl -i -u : --negotiate -X DELETE -HContent-type:application/json -d '{}' http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition/dt=10?user.name=user1
  ```

21. ddl/database/:db/table/:table/column(GET)
    - 描述

      获取表的column列表
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/column
    - 参数

参数	描述
:db	数据库名
:table	表名

    - 返回结果

参数	描述
database	数据库名
table	表名
columns	列名字和类型

    - 例子
      ```
      curl -i -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column?user.name=user1
      ```

22. ddl/database/:db/table/:table/column/:column(GET)
    - 描述

      获取表的某个具体的column的信息
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/column/:column
    - 参数

参数	描述
:db	数据库名

参数	描述
:table	表名
:column	列名

- 返回结果

参数	描述
database	数据库名
table	表名
column	列名字和类型

- 例子

```
curl -i -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column/id?user.name=user1
```

23. ddl/database/:db/table/:table/column/:column(PUT)
    - 描述

      增加表的一列
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/column/:column
    - 参数

参数	描述
:db	数据库名
:table	表名
:column	列名
type	列类型，比如string和int
comment	列备注，比如描述

    - 返回结果

参数	描述
database	数据库名
table	表名
column	列名

    - 例子

```
curl -i -u : --negotiate -X PUT -HContent-type:application/json -d '{"type": "string", "comment": "new column"}'  http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column/name?user.name=user1
```

24. ddl/database/:db/table/:table/property(GET)
    - 描述

      获取表的property
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/property
    - 参数

参数	描述
:db	数据库名
:table	表名

    - 返回结果

参数	描述
database	数据库名
table	表名
properties	属性列表

    - 例子

      ```
      curl -i -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property?user.name=user1
      ```

25. ddl/database/:db/table/:table/property/:property(GET)
    - 描述

      获取表的某个具体的property的值
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/property/:property
    - 参数

参数	描述
:db	数据库名
:table	表名
:property	属性名

    - 返回结果

参数	描述
database	数据库名
table	表名

参数	描述
property	属性列表

- 例子

```
curl -i -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property/last_modified_by?user.name=user1
```

26. ddl/database/:db/table/:table/property/:property(PUT)
    - 描述

      增加表的property的值
    - URL

      http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table/property/:property
    - 参数

参数	描述
:db	数据库名
:table	表名
:property	属性名
value	属性值

    - 返回结果

参数	描述
database	数据库名
table	表名
property	属性名

    - 例子

```
curl -i -u : --negotiate -X PUT -HContent-type:application/json -d '{"value": "my value"}' http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property/mykey?user.name=user1
```

27. mapreduce/jar(POST)
    - 描述

      执行MR任务，在执行之前，需要将MR的jar包上传到HDFS中
    - URL

      http://www.myserver.com/templeton/v1/mapreduce/jar
    - 参数

参数	描述
jar	需要执行的MR的jar包。
class	需要执行的MR的分类。

参数	描述
libjars	需要加入的classpath的jar包名，以逗号分隔。
files	需要复制到集群的文件名，以逗号分隔。
arg	Main类接受的输入参数。
define	设置hadoop的配置，格式为：define=NAME=VALUE。
statusdir	WebHCat会将执行的MR任务的状态写入到statusdir中。如果设置了这个值，那么需要用户手动进行删除。
enablelog	如果statusdir设置，enablelog设置为true，收集Hadoop任务配置和日志到$statusdir/logs。此后，成功和失败的尝试，都将记录进日志。$statusdir/logs下，子目录布局为： logs/$job_id (directory for $job_id) logs/$job_id/job.xml.html logs/$job_id/$attempt_id (directory for $attempt_id) logs/$job_id/$attempt_id/stderr logs/$job_id/$attempt_id/stdout logs/$job_id/$attempt_id/syslog 仅支持Hadoop 1.X。
callback	在MR任务执行完的回调地址，使用$jobId，将任务ID嵌入回调地址。在回调地址中，任务ID替换该$jobId。

- 返回结果

参数	描述
id	任务ID，类似“job_201110132141_0001”

- 例子

```
curl -i -u : --negotiate -d jar="/tmp/word.count-0.0.1-SNAPSHOT.jar" -d
class=com.huawei.word.count.WD -d statusdir="/output" "http://10.64.35.144:9111/
templeton/v1/mapreduce/jar?user.name=user1"
```

28. mapreduce/streaming(POST)
    - 描述

      以Streaming方式提交MR任务
    - URL

http://www.myserver.com/templeton/v1/mapreduce/streaming

- 参数

参数	描述
input	Hadoop中input的路径。
output	存储output的路径。如没有规定，WebHCat将output储存在使用队列资源可以发现到的路径。
mapper	mapper程序位置。
reducer	reducer程序位置。
files	HDFS文件添加到分布式缓存中。
arg	设置argument。
define	设置hadoop的配置变量，格式：define=NAME=VALUE
cmdenv	设置环境变量，格式：cmdenv=NAME=VALUE
statusdir	WebHCat会将执行的MR任务的状态写入到statusdir中。如果设置了这个值，那么需要用户手动进行删除。
enablelog	如果statusdir设置，enablelog设置为true，收集Hadoop任务配置和日志到$statusdir/logs。此后，成功和失败的尝试，都将记录进日志。$statusdir/logs下，子目录布局为： logs/$job_id (directory for $job_id) logs/$job_id/job.xml.html logs/$job_id/$attempt_id (directory for $attempt_id) logs/$job_id/$attempt_id/stderr logs/$job_id/$attempt_id/stdout logs/$job_id/$attempt_id/syslog 仅支持Hadoop 1.X。
callback	在MR任务执行完的回调地址，使用$jobId，将任务ID嵌入回调地址。在回调地址中，任务ID将替换该$jobId。

- 返回结果

参数	描述
id	任务ID，类似 job_201110132141_0001

- 例子

```
curl -i -u : --negotiate -d input=/input -d output=/oooo -d mapper=/bin/cat -d
reducer="/usr/bin/wc -w" -d statusdir="/output" 'http://10.64.35.144:9111/templeton/v1/
mapreduce/streaming?user.name=user1'
```

29. /hive(POST)
    - 描述

      执行Hive命令
    - URL

      http://www.myserver.com/templeton/v1/hive
    - 参数

参数	描述
execute	hive命令，包含整个和短的Hive命令。
file	包含hive命令的HDFS文件。
files	需要复制到集群的文件名，以逗号分隔。
arg	设置argument。
define	设置Hive的配置，格式：define=key=value，如果使用多实例，需要配置实例的scratch dir，如WebHCat实例使用define=hive.exec.scratchdir=**/tmp/hive-scratch**，WebHCat1实例使用define=hive.exec.scratchdir=**/tmp/hive1-scratch**，以此类推。
statusdir	WebHCat会将执行的MR任务的状态写入到statusdir中。如果设置了这个值，那么需要用户手动进行删除。

参数	描述
enablelog	如果statusdir设置，enablelog设置为true，收集Hadoop任务配置和日志到$statusdir/logs。此后，成功和失败的尝试，都将记录进日志。$statusdir/logs下，子目录布局为： logs/$job_id (directory for $job_id) logs/$job_id/job.xml.html logs/$job_id/$attempt_id (directory for $attempt_id) logs/$job_id/$attempt_id/stderr logs/$job_id/$attempt_id/stdout logs/$job_id/$attempt_id/syslog
callback	在MR任务执行完的回调地址，使用$jobId，将任务ID嵌入回调地址。在回调地址中，任务ID将替换该$jobId。

- 返回结果

参数	描述
id	任务ID，类似job_201110132141_0001

- 例子

```
curl -i -u : --negotiate -d execute="select count(*) from t1" -d statusdir="/output" -d define=hive.exec.scratchdir=/tmp/hive-scratch "http://10.64.35.144:9111/templeton/v1/hive?user.name=user1"
```

30. jobs(GET)
    - 描述

      获取所有的job id
    - URL

      http://www.myserver.com/templeton/v1/jobs
    - 参数

参数	描述
fields	如果设置成*，那么会返回每个job的详细信息。如果没设置，只返回任务ID。现在只能设置成*，如设置成其他值，将出现异常。

参数	描述
jobid	如果设置了jobid，那么只有字典顺序比jobid大的job才会返回。比如，如果jobid为"job_201312091733_0001"，只有大于该值的job才能返回。返回的job的个数，取决于numrecords。
numrecords	如果设置了numrecords和jobid，jobid列表按字典顺序排列，待jobid返回后，可以得到numrecords的最大值。如果jobid没有设置， 而numrecords设置了参数值，jobid按字典顺序排列后，可以得到numrecords的最大值。相反，如果numrecords没有设置，而jobid设置了参数值，所有大于jobid的job都将返回。
showall	如果设置为true，用户可以获取所有job，如果设置为false，则只获取当前用户提交的job。默认为false。

- 返回结果

参数	描述
id	Job id
detail	如果showall为true，那么显示detail信息，否则为null。

- 例子

```
curl -i -u : --negotiate  "http://10.64.35.144:9111/templeton/v1/jobs?user.name=user1"
```

31. jobs/:jobid(GET)
    - 描述

      获取指定job的信息
    - URL

      http://www.myserver.com/templeton/v1/jobs/:jobid
    - 参数

参数	描述
jobid	Job创建后的Jobid

    - 返回结果

参数	描述
status	包含job状态信息的json对象。

参数	描述
profile	包含job状态的json对象。WebHCat解析JobProfile对象中的信息，该对象因Hadoop版本不同而不同。
id	Job的id。
percentComplete	完成百分比，比如75% complete，如果完成后则为null。
user	创建job的用户。
callback	回调URL（如果有）。
userargs	用户提交job时的argument参数和参数值。
exitValue	job退出值。

- 例子

```
curl -i -u : --negotiate  "http://10.64.35.144:9111/templeton/v1/jobs/job_1440386556001_0255?user.name=user1"
```

32. jobs/:jobid(DELETE)
    - 描述

      kill任务
    - URL

      http://www.myserver.com/templeton/v1/jobs/:jobid
    - 参数

参数	描述
:jobid	删除的Job的ID

    - 返回结果

参数	描述
user	提交Job的用户。
status	包含Job状态信息的JSON对象。
profile	包含job信息的json对象。WebHCat解析JobProfile对象中的信息，该对象因Hadoop版本不同而不同。
id	Job的id。
callback	回调的URL（如果有）。

    - 例子

```
curl -i -u : --negotiate -X DELETE  "http://10.64.35.143:9111/templeton/v1/jobs/job_1440386556001_0265?user.name=user1"
```

16.5.2 配置 Windows 通过 EIP 访问普通模式集群 Hive

操作场景

该章节通过指导用户配置集群绑定EIP，并配置Hive文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行hive-jdbc-example样例为例进行说明。

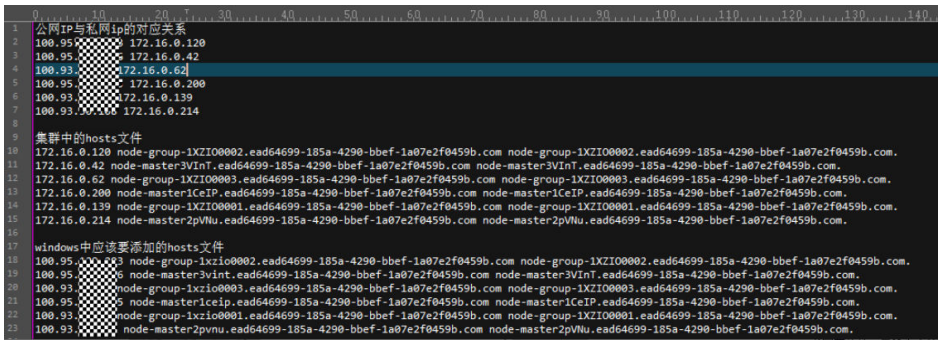
操作步骤

步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。



步骤2 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和10000端口。

步骤3 在Manager界面选择“集群 > 服务 > Hive > 更多 > 下载客户端”，将客户端中的core-site.xml和hiveclient.properties复制到样例工程的resources目录下。

步骤4 修改样例代码中的JDBC URL中使用zookeeper的连接改为直接使用hiveserver2的地址连接。将url改为jdbc:hive2:// *hiveserver主机名*:10000/

说明

- 由于使用zookeeper连接会访问zookeeper的“/hiveserver2”目录下的IP，但是里面存储的是私有IP，本地Windows无法连通，所以需要替换为hiveserver2的地址连接。
- hiveserver2服务的主机名可以在Manager界面选择“集群 > 服务 > Hive > 实例”，在“实例”界面查看“HiveServer”的“主机名称”获取。

```
// 拼接JDBC URL
//  StringBuilder strBuilder = new StringBuilder("jdbc:hive2://").append(zkQuorum).append("/");
StringBuilder strBuilder = new StringBuilder("jdbc:hive2://node-master1lnks.c0c17d76-481f-4b24-83af-159ed415ad95.com:10000/");

if ("KERBEROS".equalsIgnoreCase(auth)) {
    strBuilder
//            .append(";serviceDiscoveryMode=")
//            .append(serviceDiscoveryMode)
//            .append(";zooKeeperNamespace=")
//            .append(zooKeeperNamespace)
//            .append(";sasl.qop=")
//            .append(sasl_qop)
//            .append(";auth=")
//            .append(auth)
            .append(";principal=")
            .append(principal)
            .append(";user.principal=")
            .append(USER_NAME)
            .append(";user.keytab=")
            .append(USER_KEYTAB_FILE)
            .append(";");
} else {
    /* 普通模式 */
    strBuilder
//            .append(";serviceDiscoveryMode=")
//            .append(serviceDiscoveryMode)
//            .append(";zooKeeperNamespace=")
//            .append(zooKeeperNamespace)
//            .append(";auth=none");
}
```

----结束

16.5.3 使用 IBM JDK 产生异常“Problem performing GSS wrap”如何处理

问题

使用IBM JDK产生异常，提示“Problem performing GSS wrap”信息。

回答

问题原因：

在IBM JDK下建立的Hive connection时间超过登录用户的认证超时时间（默认一天），导致认证失败。

说明

IBM JDK的机制跟Oracle JDK的机制不同，IBM JDK在认证登录后的使用过程中做了时间检查却没有检测外部的时间更新，导致即使显式调用Hive relogin也无法得到刷新。

解决措施：

通常情况下，在发现Hive connection不可用的时候，可以关闭该connection，重新创建一个connection继续执行。

17 Impala 开发指南（安全模式）

17.1 Impala 应用开发概述

17.1.1 Impala 应用开发简介

Impala直接对存储在HDFS、HBase或对象存储服务（OBS）中的Hadoop数据提供快速、交互式SQL查询。除了使用相同的统一存储平台之外，Impala还使用与Apache Hive相同的元数据、SQL语法（Hive SQL）、ODBC驱动程序和用户界面（Hue中的Impala查询UI）。这为实时或面向批处理的查询提供了一个熟悉且统一的平台。作为查询大数据的工具补充，Impala不会替代基于MapReduce构建的批处理框架，例如Hive。基于MapReduce构建的Hive和其他框架最适合长时间运行的批处理作业。

Impala主要特点如下：

- 支持Hive查询语言（HiveQL）中大多数的SQL-92功能，包括SELECT，JOIN和聚合函数。
- HDFS，HBase 和对象存储服务（OBS）存储，包括：
  - HDFS文件格式：基于分隔符的text file，Parquet，Avro，SequenceFile和RCFile。
  - 压缩编解码器：Snappy，GZIP，Deflate，BZIP。
- 常见的数据访问接口包括：
  - JDBC驱动程序。
  - ODBC驱动程序。
  - Hue beeswax和Impala查询UI。
- impala-shell命令行接口。
- 支持Kerberos身份认证。

Impala主要应用于实时查询数据的离线分析（如日志分析，集群状态分析）、大规模的数据挖掘（用户行为分析，兴趣分区，区域展示）等场景。

17.1.2 Impala 应用开发常用概念

- **客户端**

客户端直接面向用户，可通过Java API、Thrift API访问服务端进行Impala的相关操作。本文中的Impala客户端特指Impala client的安装目录，里面包含通过Java API访问Impala的样例代码。

- **HiveQL语言**

  Hive Query Language，类SQL语句，与Hive类似。

- **Statestore**

  Statestore管理Impala集群中所有的Impalad实例的健康状态，并将实例健康信息广播到所有实例上。当某一个Impalad实例发生故障，比如节点异常、网络异常等，Statestore将通知其他Impalad实例，后续的查询请求等将不会向该实例分发。

- **Catalog**

  Catalog实例服务将每个Impalad实例上发生的元数据变动同步到集群内其他Impalad实例，从而避免在一个Impalad实例中更改元数据，其他各个实例需要执行REFRESH操作来更新。但是，在Hive中建表、修改表等，则需要执行REFRESH或者INVALIDATE METADATA操作。

17.1.3 Impala 应用开发流程

开发流程中各阶段的说明如图17-1和表17-1所示。

图 17-1 Impala 应用程序开发流程

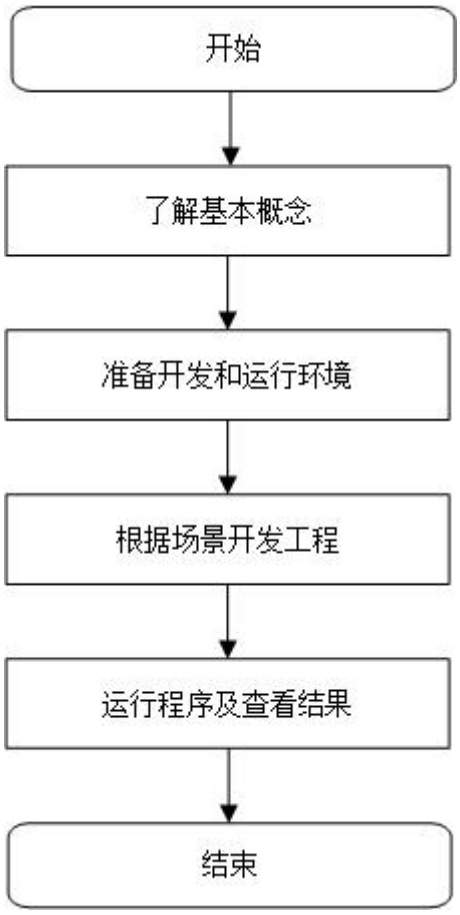


表 17-1 Impala 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Impala的基本概念。	**Impala应用开发常用概念**
准备开发和运行环境	Impala的应用程序支持使用Java、Python两种语言进行开发。推荐使用IntelliJ IDEA工具，请根据指导完成不同语言的开发环境配置。	**准备Impala开发和运行环境**
根据场景开发工程	提供了Java、Python两种不同语言的样例工程，还提供了从建表、数据加载到数据查询的样例工程。	**Impala样例程序开发思路**
运行程序及查看结果	指导用户将开发好的程序编译提交运行并查看结果。	**调测Impala应用**

17.2 准备 Impala 应用开发环境

17.2.1 准备 Impala 开发和运行环境

准备开发环境

在进行应用开发时，需要准备的本地开发环境如**表17-2**所示。

表 17-2 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，推荐Windows7以上版本。 • 运行环境：Windows或Linux系统。如需在本地调测程序，运行环境需要和集群业务平面网络互通。

准备项	说明
安装JDK	开发和运行环境的基本配置。版本要求如下： ● MRS集群的服务端和客户端仅支持自带的Oracle JDK（版本为1.8），不允许替换。 ● 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的，支持Oracle JDK和IBM JDK。 – Oracle JDK：支持1.7和1.8版本。 – IBM JDK：推荐1.7.8.10、1.7.9.40和1.8.3.0版本。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** ● 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 ● 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 ● 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
准备开发用户	参考**准备MRS应用开发用户**进行操作，准备用于应用开发的集群用户并授予相应权限。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果本地Windows开发环境和集群业务平面网络互通，然后直接在Windows中进行程序调测。
- 如果使用Linux环境调测程序，需准备Linux节点。

17.3 开发 Impala 应用

17.3.1 Impala 样例程序开发思路

场景说明

假定用户开发一个Impala数据分析应用，用于管理企业雇员信息，如表17-3、表17-4所示。

开发思路

步骤1 数据准备。

1. 创建三张表，雇员信息表“employees_info”、雇员联络信息表“employees_contact”、雇员信息扩展表“employees_info_extended”。
   - 雇员信息表“employees_info”的字段为雇员编号、姓名、支付薪水币种、薪水金额、缴税税种、工作地、入职时间，其中支付薪水币种“R”代表人民币，“D”代表美元。
   - 雇员联络信息表“employees_contact”的字段为雇员编号、电话号码、e-mail。
   - 雇员信息扩展表“employees_info_extended”的字段为雇员编号、姓名、电话号码、e-mail、支付薪水币种、薪水金额、缴税税种、工作地，分区字段为入职时间。

     创建表代码实现请见**创建Impala表**。
2. 加载雇员信息数据到雇员信息表“employees_info”中。

   加载数据代码实现请见**加载Impala数据**。

   雇员信息数据如表17-3所示。

表 17-3 雇员信息数据

编号	姓名	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
1	Wang	R	8000.01	personal income tax&0.05	China:Shenzhen	2014
3	Tom	D	12000.02	personal income tax&0.09	America:NewYork	2014
4	Jack	D	24000.03	personal income tax&0.09	America:Manhattan	2014
6	Linda	D	36000.04	personal income tax&0.09	America:NewYork	2014
8	Zhang	R	9000.05	personal income tax&0.05	China:Shanghai	2014

3. 加载雇员联络信息数据到雇员联络信息表“employees_contact”中。
   雇员联络信息数据如**表17-4**所示。

   表 17-4 雇员联络信息数据

编号	电话号码	e-mail
1	135 XXXX XXXX	xxxx@xx.com
3	159 XXXX XXXX	xxxxx@xx.com.cn
4	186 XXXX XXXX	xxxx@xx.org
6	189 XXXX XXXX	xxxx@xxx.cn
8	134 XXXX XXXX	xxxx@xxxx.cn

步骤2 数据分析。

数据分析代码实现，请见**查询Impala数据**。

- 查看薪水支付币种为美元的雇员联系方式。
- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中。
- 统计表employees_info中有多少条记录。
- 查询使用以“cn”结尾的邮箱的员工信息。

步骤3 提交数据分析任务，统计表employees_info中有多少条记录。实现请见**Impala样例程序指导**。

----结束

17.3.2 创建 Impala 表

功能简介

本小节介绍了如何使用Impala SQL建内部表、外部表的基本操作。创建表主要有以下三种方式。

- 自定义表结构，以关键字EXTERNAL区分创建内部表和外部表。
  - 内部表，如果对数据的处理都由Impala完成，则应该使用内部表。在删除内部表时，元数据和数据一起被删除。
  - 外部表，如果数据要被多种工具共同处理，则应该使用外部表，可避免对该数据的误操作。删除外部表时，只删除掉元数据。
- 根据已有表创建新表，使用CREATE LIKE句式，完全复制原有的表结构，包括表的存储格式。
- 根据查询结果创建新表，使用CREATE AS SELECT句式。
  这种方式比较灵活，可以在复制原表表结构的同时指定要复制哪些字段，不包括表的存储格式。

样例代码

```
-- 创建外部表employees_info.
CREATE EXTERNAL TABLE IF NOT EXISTS employees_info
(
id INT,
name STRING,
usd_flag STRING,
salary DOUBLE,
deductions MAP<STRING, DOUBLE>,
address STRING,
entrytime STRING
)
-- 指定行中各字段分隔符.
-- "delimited fields terminated by"指定列与列之间的分隔符为','.
ROW FORMAT delimited fields terminated by ','
-- 指定表的存储格式为TEXTFILE.
STORED AS TEXTFILE;

-- 使用CREATE Like创建表.
CREATE TABLE employees_like LIKE employees_info;

-- 使用DESCRIBE查看employees_info、employees_like表结构.
DESCRIBE employees_info;
DESCRIBE employees_like;

-- 创建内部表employees_info.
CREATE TABLE IF NOT EXISTS employees_info
(
id INT,
name STRING,
usd_flag STRING,
salary DOUBLE,
deductions MAP<STRING, DOUBLE>,
address STRING,
entrytime STRING
)
-- 指定行中各字段分隔符.
-- "delimited fields terminated by"指定列与列之间的分隔符为','.
ROW FORMAT delimited fields terminated by ','
-- 指定表的存储格式为TEXTFILE.
STORED AS TEXTFILE;
```

扩展应用

- 创建分区表

  一个表可以拥有一个或者多个分区，每个分区以文件夹的形式单独存在表文件夹的目录下。对分区内数据进行查询，可缩小查询范围，加快数据的检索速度和可对数据按照一定的条件进行管理。

  分区是在创建表的时候用PARTITIONED BY子句定义的。

  ```
  CREATE EXTERNAL TABLE IF NOT EXISTS employees_info_extended
  (
  id INT,
  name STRING,
  usd_flag STRING,
  salary DOUBLE,
  deductions MAP<STRING, DOUBLE>,
  address STRING
  )
  -- 使用关键字PARTITIONED BY指定分区列名及数据类型 .
  PARTITIONED BY (entrytime STRING)
  STORED AS TEXTFILE;
  ```

- 更新表的结构

  一个表在创建完成后，还可以使用ALTER TABLE执行增、删字段，修改表属性，添加分区等操作。


```
-- 为表employees_info_extended增加tel_phone、email字段.
ALTER TABLE employees_info_extended ADD COLUMNS (tel_phone STRING, email STRING);
```

- Impala使用OBS存储。

  需要在集群管理页面Manager里面设置指定的参数到“core-site.xml”，AK/SK可登录“OBS控制台”，进入“我的凭证”页面获取。

  ```
  fs.obs.access.key=AK;
  fs.obs.secret.key=SK;
  fs.obs.endpoint=endpoint;
  ```

  新建表的存储类型为OBS。

  ```
  create table obs(c1 string, c2 string) stored as parquet location 'obs://obs-lmm/hive/orctest'
  tblproperties('orc.compress'='SNAPPY');
  ```

  说明

  当前Impala使用OBS存储时，同一张表中，不支持分区和表存储位置处于不同的桶中。

  例如：创建分区表指定存储位置为OBS桶1下的文件夹，此时修改表分区存储位置的操作将不会生效，在实际插入数据时以表存储位置为准。

  1. 创建分区表并指定表存储路径。
     ```
     create table table_name(id int,name string,company string) partitioned by(dt date) row format delimited fields terminated by ',' stored as textfile location "obs://OBS桶1/桶下文件夹";
     ```
  2. 修改此表分区位置到另外一个桶下，此时该修改不会生效。
     ```
     alter table table_name partition(dt date) set location "obs://OBS桶2/桶下文件夹";
     ```

17.3.3 加载 Impala 数据

功能简介

本小节介绍了如何使用Impala SQL向已有的表employees_info中加载数据。从本节中可以掌握如何从集群中加载数据。

样例代码

```
-- 从本地文件系统/opt/impala_examples_data/目录下将employee_info.txt加载进employees_info表中.
LOAD DATA LOCAL INPATH '/opt/impala_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;

-- 从HDFS上/user/impala_examples_data/employee_info.txt加载进employees_info表中.
LOAD DATA INPATH '/user/impala_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;
```

说明

加载数据的实质是将数据复制到HDFS上指定表的目录下。

“LOAD DATA LOCAL INPATH”命令可以完成从本地文件系统加载文件到Impala的需求，但是当指定“LOCAL”时，这里的路径指的是当前连接的“Impalad”的本地文件系统的路径。

17.3.4 查询 Impala 数据

功能简介

本小节介绍了如何使用Impala SQL对数据进行查询分析。从本节中可以掌握如下查询分析方法。

- SELECT查询的常用特性，如JOIN等。
- 加载数据进指定分区。

- 如何使用Impala自带函数。
- 如何使用自定义函数进行查询分析，如何创建、定义自定义函数请见**开发Impala用户自定义函数**。

样例代码

```
-- 查看薪水支付币种为美元的雇员联系方式.
SELECT
a.name,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b  ON(a.id = b.id) WHERE usd_flag='D';

-- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中.
INSERT OVERWRITE TABLE employees_info_extended PARTITION (entrytime = '2014')
SELECT
a.id,
a.name,
a.usd_flag,
a.salary,
a.deductions,
a.address,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE a.entrytime = '2014';

-- 使用Impala中已有的函数COUNT()，统计表employees_info中有多少条记录.
SELECT COUNT(*) FROM employees_info;

-- 查询使用以“cn”结尾的邮箱的员工信息.
SELECT a.name, b.tel_phone FROM  employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE b.email like '%cn';
```

扩展使用

自定义函数，具体内容请参见**开发Impala用户自定义函数**。

17.3.5 开发 Impala 用户自定义函数

当Impala的内置函数不能满足需要时，可以通过编写用户自定义函数UDF（User-Defined Functions）插入自己的处理代码并在查询中使用它们。

按实现方式，UDF有如下分类：

- 普通的UDF，用于操作单个数据行，且产生一个数据行作为输出。
- 用户定义聚集函数UDAF（User-Defined Aggregating Functions），用于接受多个输入数据行，并产生一个输出数据行。
- 用户定义表生成函数UDTF(User-Defined Table-Generating Functions)，用于操作单个输入行，产生多个输出行。**Impala不支持该类UDF**。

按使用方法，UDF有如下分类：

- 临时函数，只能在当前会话使用，重启会话后需要重新创建。
- 永久函数，可以在多个会话中使用，不需要每次创建。

Impala支持开发Java UDF，也可以复用Hive开发的UDFs，前提是使用Impala支持的数据类型。Impala支持的数据类型请参考**http://impala.apache.org/docs/build3x/html/topics/impala_datatypes.html**。

此外，Impala还支持C++编写的UDF，性能上比Java UDF更好。

使用示例

以下为复用lower()函数的示例。

```
[localhost:21000] > create database udfs;
[localhost:21000] > use udfs;
[localhost:21000] > create function my_lower(string)
returns string location '/user/hive/udfs/hive.jar'
symbol='org.apache.hadoop.hive.ql.udf.UDFLower';
[localhost:21000] > select my_lower('Some String NOT ALREADY LOWERCASE');
+----------------------------------------------------+
| udfs.my_lower('some string not already lowercase') |
+----------------------------------------------------+
| some string not already lowercase |
+----------------------------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create table t2 (s string);
[localhost:21000] > insert into t2 values ('lower'),('UPPER'),('Init cap'),('CamelCase');
Inserted 4 rows in 2.28s
[localhost:21000] > select * from t2;
+-----------+
| s |
+-----------+
| lower |
| UPPER |
| Init cap |
| CamelCase |
+-----------+
Returned 4 row(s) in 0.47s
[localhost:21000] > select my_lower(s) from t2;
+------------------+
| udfs.my_lower(s) |
+------------------+
| lower |
| upper |
| init cap |
| camelcase |
+------------------+
Returned 4 row(s) in 0.54s
```

17.3.6 Impala 样例程序指导

功能简介

本小节介绍了如何使用样例程序完成分析任务。本章节以使用JDBC接口提交数据分析任务为例。

样例代码

使用Impala JDBC接口提交数据分析任务，参考样例程序中的JDBCExample.java。

1. 修改以下变量为false，标识连接集群的认证模式为普通模式。

   ```
   // 所连接集群的认证模式是否在安全模式
     boolean isSecureVer = false;
   ```

2. 定义Impala SQL。Impala SQL必须为单条语句，注意不能包含“;”。

   ```
     // 定义HQL，不能包含“;”
     String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
           "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};
   ```

3. 拼接JDBC URL。

   ```
   // 拼接JDBC URL
   StringBuilder sBuilder = new StringBuilder(
     "jdbc:hive2://").append("impalad_ip").append("/");

   if (isSecurityMode) {
   ```



```
    // 安全模式
    sBuilder.append(";auth=")
            .append(clientInfo.getAuth())
            .append(";principal=")
            .append(clientInfo.getPrincipal())
            .append(";");
} else {
    // 普通模式
    sBuilder.append(";auth=noSasl");
}
String url = sBuilder.toString();
```

说明

直连Impalad实例时，若当前连接的Impalad实例故障则会导致访问Impala失败。

4. 加载Hive JDBC驱动。

```
// 加载Hive JDBC驱动
Class.forName(HIVE_DRIVER);
```

5. 填写正确的用户名，获取JDBC连接，确认Impala SQL的类型（DDL/DML），调用对应的接口执行Impala SQL，输出查询的列名和结果到控制台，关闭JDBC连接。

```
Connection connection = null;
  try {
    // 获取JDBC连接
    // 第二个参数需要填写正确的用户名，否则会以匿名用户(anonymous)登录
    connection = DriverManager.getConnection(url, "userName", "");

    // 建表
    // 表建完之后，如果要往表中导数据，可以使用LOAD语句将数据导入表中，比如从HDFS上将数据导入表:
    //load data inpath '/tmp/employees.txt' overwrite into table employees_info;
    execDDL(connection,sqls[0]);
    System.out.println("Create table success!");

    // 查询
    execDML(connection,sqls[1]);

    // 删表
    execDDL(connection,sqls[2]);
    System.out.println("Delete table success!");
  }
  finally {
    // 关闭JDBC连接
    if (null != connection) {
      connection.close();
    }

public static void execDDL(Connection connection, String sql)
  throws SQLException {
    PreparedStatement statement = null;
    try {
      statement = connection.prepareStatement(sql);
      statement.execute();
    }
    finally {
      if (null != statement) {
        statement.close();
      }
    }
  }

  public static void execDML(Connection connection, String sql) throws SQLException {
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    ResultSetMetaData resultMetaData = null;

    try {
```



```
    // 执行Impala SQL
    statement = connection.prepareStatement(sql);
    resultSet = statement.executeQuery();

    // 输出查询的列名到控制台
    resultMetaData = resultSet.getMetaData();
    int columnCount = resultMetaData.getColumnCount();
    for (int i = 1; i <= columnCount; i++) {
      System.out.print(resultMetaData.getColumnLabel(i) + '\t');
    }
    System.out.println();

    // 输出查询结果到控制台
    while (resultSet.next()) {
      for (int i = 1; i <= columnCount; i++) {
        System.out.print(resultSet.getString(i) + '\t');
      }
      System.out.println();
    }
  }
  finally {
    if (null != resultSet) {
      resultSet.close();
    }

    if (null != statement) {
      statement.close();
    }
  }
}
```

17.4 调测 Impala 应用

17.4.1 在 Windows 中调测 Impala JDBC 应用

步骤1 运行样例。

导入和修改样例后，即可在开发环境中，右击“JDBCExample.java”，选择“Run 'JDBCExample.main()'”运行对应的应用程序工程。

说明

使用Windows访问MRS集群来操作Impala，有如下两种方式。

方法一：申请一台Windows的ECS访问MRS集群操作Impala，在安装开发环境后可直接运行样例代码。

1. 在“现有集群”列表中，单击已创建的集群名称。

   记录集群的“可用分区”、“虚拟私有云”，以及Master节点的“默认安全组”。

2. 在弹性云服务管理控制台，创建一个新的弹性云服务器。

   弹性云服务器的“可用分区”、“虚拟私有云”、“安全组”，需要和待访问集群的配置相同。

   选择一个Windows系统的公共镜像。

   其他配置参数详细信息，请参见**购买弹性云服务器**。

方法二：使用本机访问MRS集群操作Impala，在安装开发环境后并完成以下步骤后再运行样例代码。

1. 为任意一个Core节点绑定弹性公网IP，完成后将该IP地址配置在开发样例的client.properties下的impala-server配置项中，用于访问Impala服务、提交SQL语句。
   1. 在弹性公网IP页面申请一个弹性公网IP，并为集群的任一Core节点绑定该弹性公网IP，具体请参考**申请并绑定弹性公网IP**。
   2. 为MRS集群开放安全组规则。在集群Master节点和Core节点的安全组添加安全组规则使弹性云服务器可以访问集群，具体请参考**配置安全组规则**。
2. 修改导入样例的“krb5.conf”中"kdc"、"admin_server"和“kpasswd_server”三个参数的ip，使其对应于KrbServer服务中对应的弹性公网IP（Kerberos服务默认在Master节点上，此处取Master节点的公网IP）。

样例中的client.properties配置如下：

```
auth = KERBEROS    ##kerberos模式
principal = impala/node-ana-corexphm@10530B19_8446_4846_97BD_87880A2535DF.COM   ##所要连接的impalad实例使用的principal
impala-server = XX.XX.XX.XX:21050   ##指定要连接的impalad实例所在Core节点绑定的服务地址，方式二需要填写步骤1中绑定的弹性公网IP
```

步骤2 查看结果。

查看样例代码中的Impala SQL所查询出的结果，运行成功结果会有如下信息。

JDBC客户端运行及结果查看。

```
Create table success!
_c0
0
Delete table success!
```

----结束

17.4.2 在 Linux 中调测 Impala JDBC 应用

步骤1 在运行调测环境上创建一个目录作为运行目录，如“/opt/impala_examples”，并在该目录下创建子目录“conf”。

步骤2 执行**mvn package**，在样例工程target目录下获取jar包，比如: impala-examples-mrs-2.1-jar-with-dependencies.jar ，复制到“/opt/impala_examples”下。

步骤3 开启Kerberos认证的安全集群下把从**准备MRS应用开发用户**获取的“user.keytab”和“krb5.conf”复制到“/opt/impala_examples/conf”下。

步骤4 在Linux环境下执行如下命令运行样例程序。

chmod +x /opt/impala_examples -R

cd /opt/impala_examples

java -cp impala-examples-mrs-2.1-jar-with-dependencies.jar com.huawei.bigdata.impala.example.ExampleMain

步骤5 在命令行终端查看样例代码中的Impala SQL所查询出的结果。

Linux环境运行成功结果会有如下信息。

```
Create table success!
_c0
0
Delete table success!
```

----结束

17.5 Impala 应用开发常见问题

17.5.1 Impala JDBC 接口介绍

Impala使用Hive的JDBC接口，Hive JDBC接口遵循标准的JAVA JDBC驱动标准。

说明

Impala并不能支持所有的Hive JDBC标准API。执行某些操作会产生“Method not supported”的SQLException异常。

17.5.2 Impala SQL 接口介绍

Impala SQL提供对HiveQL的高度兼容性，详情请参见**https://impala.apache.org/docs/build/html/topics/impala_langref.html**。

17.6 Impala 开发规范

17.6.1 Impala 开发规则

Hive JDBC 驱动的加载

客户端程序以JDBC的形式连接Impalad时，需要首先加载Hive的JDBC驱动类org.apache.hive.jdbc.HiveDriver。

所以在客户端程序开始前，必须先使用当前类加载器加载该驱动类。

如果classpath下没有相应的jar包，则客户端程序抛出Class Not Found异常并退出。

如下：

```
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
```

获取数据库连接

使用JDK的驱动管理类java.sql.DriverManager来获取一个Impalad的数据库连接。

Impalad的数据库URL为url="jdbc:hive2://xxx.xxx.xxx.xxx:21050;auth=KERBEROS;principal=impala/

hadoop.hadoop.com@HADOOP.COM;user.principal=impala/hadoop.hadoop.com;user.keytab=conf/impala.keytab";

以上已经经过安全认证，所以用户名和密码为null或者空。

// 建立连接

```
connection = DriverManager.getConnection(url, "", "");
```

执行 Impala SQL

执行Impala SQL，注意Impala SQL不能以“;”结尾。

正确示例：

```
String sql = "SELECT COUNT(*) FROM employees_info";
Connection connection = DriverManager.getConnection(url, "", "");
PreparedStatement statement = connection.prepareStatement(sql);
resultSet = statement.executeQuery();
```

错误示例：

```
String sql = "SELECT COUNT(*) FROM employees_info;";
Connection connection = DriverManager.getConnection(url, "", "");
PreparedStatement statement = connection.prepareStatement(sql);
resultSet = statement.executeQuery();
```

Impala SQL 语法规则之判空

判断字段是否为“空”，即没有值，使用“is null”；判断不为空，即有值，使用“is not null”。

要注意的是，在Impala SQL中String类型的字段若是空字符串， 即长度为0，那么对它进行is null的判断结果是False。此时应该使用“col = '' ”来判断空字符串；使用“col != '' ”来判断非空字符串。

正确示例：

```
select * from default.tbl_src where id is null;
select * from default.tbl_src where id is not null;
select * from default.tbl_src where name = '';
select * from default.tbl_src where name != '';
```

错误示例：

```
select * from default.tbl_src where id = null;
select * from default.tbl_src where id != null;
select * from default.tbl_src where name is null;
select * from default.tbl_src where name is not null;注：表tbl_src的id字段为Int类型，name字段为String类型。
```

多线程安全登录方式

如果有多线程进行login的操作，当应用程序第一次登录成功后，所有线程再次登录时应该使用relogin的方式。

login的代码样例：

```
private Boolean login(Configuration conf){
  boolean flag = false;
  UserGroupInformation.setConfiguration(conf);

  try {
    UserGroupInformation.loginUserFromKeytab(conf.get(PRINCIPAL), conf.get(KEYTAB));
```



```
    System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
     flag = true;
   } catch (IOException e) {
     e.printStackTrace();
   }
   return flag;

 }
```

relogin的代码样例：

```
public Boolean relogin(){
       boolean flag = false;
       try {

         UserGroupInformation.getLoginUser().reloginFromKeytab();
         System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
         flag = true;
       } catch (IOException e) {
          e.printStackTrace();
       }
       return flag;
   }
```

避免对同一张表同时进行读写操作

目前的版本中，Hive不支持并发操作，需要避免对同一张表同时进行读写操作，否则会出现查询结果不准确，甚至任务失败的情况。

17.6.2 Impala 开发建议

Impala SQL 编写之不支持隐式类型转换

查询语句使用字段的值做过滤时，不支持使用Hive类似的隐式类型转换来编写Impala SQL：

Impala示例：

```
select * from default.tbl_src where id = 10001;
select * from default.tbl_src where name = 'TestName';
```

Hive示例(支持隐式类型转换)：

```
select * from default.tbl_src where id = '10001';
select * from default.tbl_src where name = TestName;
```

说明

表tbl_src的id字段为Int类型，name字段为String类型。

JDBC 超时限制

Impala使用Hive提供的JDBC，Hive提供的JDBC实现有超时限制，默认是5分钟，用户可以通过java.sql.DriverManager.setLoginTimeout(int seconds)设置，seconds的单位为秒。

17.6.3 Impala 开发示例

JDBC 二次开发示例代码

以下示例代码主要功能如下。

1. 普通(非Kerberos)模式下，使用用户名和密码进行登录，如不指定用户，则匿名登录；
2. 在JDBC URL地址中提供登录Kerberos用户的principal，程序自动完成安全登录、建立Impala连接。
3. 执行创建表、查询和删除三类Impala SQL语句。

```
package com.huawei.bigdata.impala.example;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

/**
 * Simple example for hive jdbc.
 */
public class JDBCExample {
    private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";
    private ClientInfo clientInfo;
    private boolean isSecurityMode;
    public JDBCExample(ClientInfo clientInfo, boolean isSecurityMode){
        this.clientInfo = clientInfo;
        this.isSecurityMode = isSecurityMode;
    }

    /**
     *
     * @throws ClassNotFoundException
     * @throws SQLException
     */
    public void run() throws ClassNotFoundException, SQLException {

        //Define hive sql, the sql can not include ";"
        String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
                "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};

        StringBuilder sBuilder = new StringBuilder(
                "jdbc:hive2://").append(clientInfo.getImpalaServer()).append("/");

        if (isSecurityMode) {
            sBuilder.append(";auth=")
                    .append(clientInfo.getAuth())
                    .append(";principal=")
                    .append(clientInfo.getPrincipal())
                    .append(";");
        } else {
            sBuilder.append(";auth=noSasl");
        }
        String url = sBuilder.toString();
        Class.forName(HIVE_DRIVER);
        Connection connection = null;
        try {
            /**
             * Get JDBC connection, If not use security mode, need input correct username,
             * otherwise, wil login as "anonymous" user
             */
            //connection = DriverManager.getConnection(url, "", "");
```

```
        connection = DriverManager.getConnection(url);
        /**
         * Run the create table sql, then can load the data if needed. eg.
         * "load data inpath '/tmp/employees.txt' overwrite into table employees_info;"
         */
        execDDL(connection,sqls[0]);
        System.out.println("Create table success!");

        execDML(connection,sqls[1]);

        execDDL(connection,sqls[2]);
        System.out.println("Delete table success!");
    }
    finally {
        if (null != connection) {
            connection.close();
        }
    }
}

public static void execDDL(Connection connection, String sql)
        throws SQLException {
    PreparedStatement statement = null;
    try {
        statement = connection.prepareStatement(sql);
        statement.execute();
    }
    finally {
        if (null != statement) {
            statement.close();
        }
    }
}

public static void execDML(Connection connection, String sql) throws SQLException {
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    ResultSetMetaData resultMetaData = null;

    try {
        statement = connection.prepareStatement(sql);
        resultSet = statement.executeQuery();

        /**
         * Print the column name to console
         */
        resultMetaData = resultSet.getMetaData();
        int columnCount = resultMetaData.getColumnCount();
        for (int i = 1; i <= columnCount; i++) {
            System.out.print(resultMetaData.getColumnLabel(i) + '\t');
        }
        System.out.println();

        /**
         * Print the query result to console
         */
        while (resultSet.next()) {
            for (int i = 1; i <= columnCount; i++) {
                System.out.print(resultSet.getString(i) + '\t');
            }
            System.out.println();
        }
    }
    finally {
        if (null != resultSet) {
            resultSet.close();
        }

        if (null != statement) {
```



```
            statement.close();
        }
    }
  }
}
```


18 Impala 开发指南（普通模式）

18.1 Impala 应用开发概述

18.1.1 Impala 应用开发简介

Impala直接对存储在HDFS，HBase 或对象存储服务（OBS）中的Hadoop数据提供快速，交互式SQL查询。除了使用相同的统一存储平台之外，Impala还使用与Apache Hive相同的元数据，SQL语法（Hive SQL），ODBC驱动程序和用户界面（Hue中的Impala查询UI）。这为实时或面向批处理的查询提供了一个熟悉且统一的平台。作为查询大数据的工具补充，Impala不会替代基于MapReduce构建的批处理框架，例如Hive。基于MapReduce构建的Hive和其他框架最适合长时间运行的批处理作业。

Impala主要特点如下：

- 支持Hive查询语言（HiveQL）中大多数的SQL-92功能，包括 SELECT，JOIN和聚合函数。
- HDFS，HBase 和对象存储服务（OBS）存储，包括：
  - HDFS文件格式：基于分隔符的text file，Parquet，Avro，SequenceFile和RCFile。
  - 压缩编解码器：Snappy，GZIP，Deflate，BZIP。
- 常见的数据访问接口包括：
  - JDBC驱动程序。
  - ODBC驱动程序。
  - HUE beeswax和Impala查询UI。
- impala-shell命令行接口。
- 支持Kerberos身份认证。

Impala主要应用于实时查询数据的离线分析（如日志分析，集群状态分析）、大规模的数据挖掘（用户行为分析，兴趣分区，区域展示）等场景下。

18.1.2 Impala 应用开发常用概念

- **客户端**

客户端直接面向用户，可通过Java API、Thrift API访问服务端进行Impala的相关操作。本文中的Impala客户端特指Impala client的安装目录，里面包含通过Java API访问Impala的样例代码。

- **HiveQL语言**

  Hive Query Language，类SQL语句，与Hive类似。

- **Statestore**

  Statestore管理Impala集群中所有的Impalad实例的健康状态，并将实例健康信息广播到所有实例上。当某一个Impalad实例发生故障，比如节点异常、网络异常等，Statestore将通知其他Impalad实例，后续的查询请求等将不会向该实例分发。

- **Catalog**

  Catalog实例服务将每个Impalad实例上发生的元数据变动同步到集群内其他Impalad实例，从而避免在一个Impalad实例中更改元数据，其他各个实例需要执行REFRESH操作来更新。但是，在Hive中建表，修改表等，则需要执行REFRESH或者INVALIDATE METADATA操作。

18.1.3 Impala 应用开发流程

开发流程中各阶段的说明如图18-1和表18-1所示。

图 18-1 Impala 应用程序开发流程

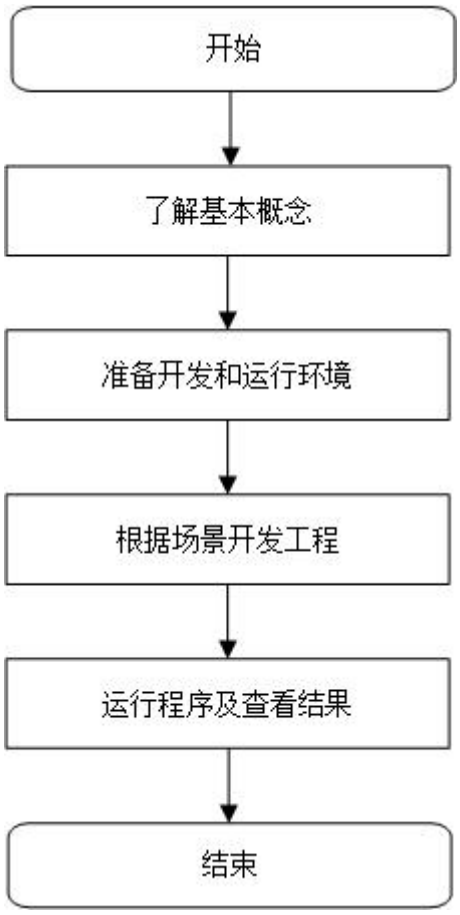


表 18-1 Impala 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Impala的基本概念。	**Impala应用开发常用概念**
准备开发和运行环境	Impala的应用程序支持使用Java、Python两种语言进行开发。推荐使用IntelliJ IDEA工具，请根据指导完成不同语言的开发环境配置。	**准备Impala开发和运行环境**
根据场景开发工程	提供了Java、Python两种不同语言的样例工程，还提供了从建表、数据加载到数据查询的样例工程。	**Impala样例程序开发思路**
运行程序及查看结果	指导用户将开发好的程序编译提交运行并查看结果。	**调测Impala应用**

18.2 准备 Impala 应用开发环境

18.2.1 准备 Impala 开发和运行环境

准备开发环境

在进行应用开发时，需要准备的本地开发环境如**表18-2**所示。

表 18-2 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，推荐Windows7以上版本。 • 运行环境：Windows或Linux系统。如需在本地调测程序，运行环境需要和集群业务平面网络互通。

准备项	说明
安装JDK	开发和运行环境的基本配置。版本要求如下： ● MRS集群的服务端和客户端仅支持自带的Oracle JDK（版本为1.8），不允许替换。 ● 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的，支持Oracle JDK和IBM JDK。 – Oracle JDK：支持1.7和1.8版本。 – IBM JDK：推荐1.7.8.10、1.7.9.40和1.8.3.0版本。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** ● 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 ● 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 ● 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果本地Windows开发环境和集群业务平面网络互通，然后直接在Windows中进行程序调测。
- 如果使用Linux环境调测程序，需准备安装集群客户端的Linux节点并获取相关配置文件。

  集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

18.2.2 导入并配置 Impala 样例工程

操作场景

为了运行Impala组件的JDBC接口样例代码，需要完成下面的配置并导入样例工程操作。

说明

以在Windows环境下开发JDBC方式连接Impala服务的应用程序为例。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取Impala示例工程。

步骤2 在Impala示例工程根目录impala-examples-normal文件夹下，打开cmd窗口，执行**mvn install**进行编译。

```
D:\SuccessRepo2\huaweicloud-mrs-example-mrs-3.1.0\src\impala-examples\impala-examples-normal 的目录

2022/09/06  16:28    <DIR>          .
2022/09/06  16:28    <DIR>          ..
2022/09/06  16:31    <DIR>          ${env.MAVEN_CACHE_DIR}
2022/07/30  21:44    <DIR>          conf
2022/07/30  21:44             6,512 pom.xml
2022/07/30  21:44    <DIR>          src
2022/09/06  16:30    <DIR>          target
               1 个文件          6,512 字节
               6 个目录 573,197,250,560 可用字节

D:\SuccessRepo2\huaweicloud-mrs-example-mrs-3.1.0\src\impala-examples\impala-examples-normal>mvn install
```

步骤3 在Impala示例工程根目录impala-examples-normal文件夹下，打开cmd窗口，执行**mvn idea:idea**创建IntelliJ IDEA工程。

步骤4 导入样例工程到IntelliJ IDEA开发环境。

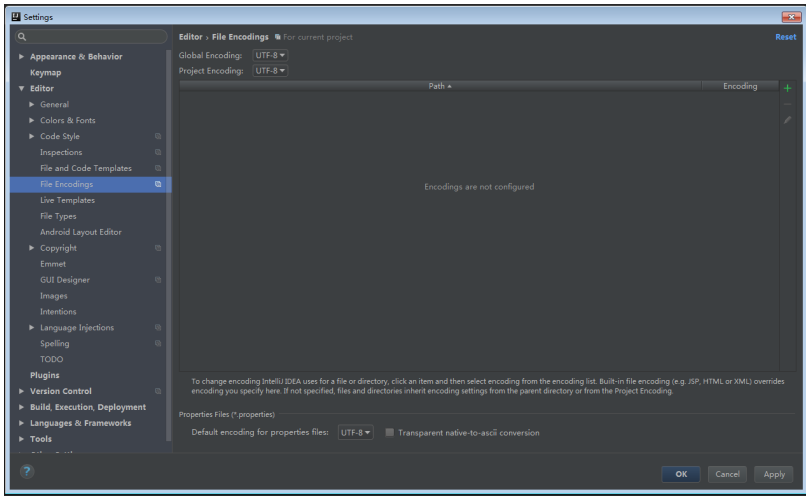
1. 打开IntelliJ IDEA，依次选择“File > Open”。
2. 在弹出的“Open File or Project”对话框中选择Impala样例工程文件夹，单击“OK”。

   导入成功后，com.huawei.bigdata.impala.example包下的JDBCExample类为JDBC接口样例代码。

步骤5 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”。
2. 在弹出的“Settings”窗口左边导航上选择“Editor > File Encodings”，在“Global Encoding”和“Project Encodings”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如**图4 设置IntelliJ IDEA的编码格式**所示。

图 18-2 设置 IntelliJ IDEA 的编码格式



----结束

18.3 开发 Impala 应用

18.3.1 Impala 样例程序开发思路

场景说明

假定用户开发一个Impala数据分析应用，用于管理企业雇员信息，如**表18-3**、**表18-4**所示。

开发思路

步骤1 数据准备。

1. 创建三张表，雇员信息表“employees_info”、雇员联络信息表“employees_contact”、雇员信息扩展表“employees_info_extended”。
   - 雇员信息表“employees_info”的字段为雇员编号、姓名、支付薪水币种、薪水金额、缴税税种、工作地、入职时间，其中支付薪水币种“R”代表人民币，“D”代表美元。
   - 雇员联络信息表“employees_contact”的字段为雇员编号、电话号码、e-mail。
   - 雇员信息扩展表“employees_info_extended”的字段为雇员编号、姓名、电话号码、e-mail、支付薪水币种、薪水金额、缴税税种、工作地，分区字段为入职时间。

     创建表代码实现请见**创建Impala表**。
2. 加载雇员信息数据到雇员信息表“employees_info”中。

   加载数据代码实现请见**加载Impala数据**。

   雇员信息数据如**表18-3**所示。

表 18-3 雇员信息数据

编号	姓名	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
1	Wang	R	8000.01	personal income tax&0.05	China:Shenzhen	2014
3	Tom	D	12000.02	personal income tax&0.09	America:NewYork	2014
4	Jack	D	24000.03	personal income tax&0.09	America:Manhattan	2014
6	Linda	D	36000.04	personal income tax&0.09	America:NewYork	2014

编号	姓名	支付薪水币种	薪水金额	缴税税种	工作地	入职时间
8	Zhang	R	9000.05	personal income tax&0.05	China:Shanghai	2014

3. 加载雇员联络信息数据到雇员联络信息表“employees_contact”中。
   雇员联络信息数据如**表18-4**所示。

   表 18-4 雇员联络信息数据

编号	电话号码	e-mail
1	135 XXXX XXXX	xxxx@xx.com
3	159 XXXX XXXX	xxxxx@xx.com.cn
4	186 XXXX XXXX	xxxx@xx.org
6	189 XXXX XXXX	xxxx@xxx.cn
8	134 XXXX XXXX	xxxx@xxxx.cn

步骤2 数据分析。

数据分析代码实现，请见**查询Impala数据**。

- 查看薪水支付币种为美元的雇员联系方式。
- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中。
- 统计表employees_info中有多少条记录。
- 查询使用以“cn”结尾的邮箱的员工信息。

步骤3 提交数据分析任务，统计表employees_info中有多少条记录。实现请见**Impala样例程序指导**。

----结束

18.3.2 创建 Impala 表

功能简介

本小节介绍了如何使用Impala SQL建内部表、外部表的基本操作。创建表主要有以下三种方式。

- 自定义表结构，以关键字EXTERNAL区分创建内部表和外部表。
  - 内部表，如果对数据的处理都由Impala完成，则应该使用内部表。在删除内部表时，元数据和数据一起被删除。
  - 外部表，如果数据要被多种工具共同处理，则应该使用外部表，可避免对该数据的误操作。删除外部表时，只删除掉元数据。

- 根据已有表创建新表，使用CREATE LIKE句式，完全复制原有的表结构，包括表的存储格式。
- 根据查询结果创建新表，使用CREATE AS SELECT句式。

  这种方式比较灵活，可以在复制原表表结构的同时指定要复制哪些字段，不包括表的存储格式。

样例代码

```
-- 创建外部表employees_info.
CREATE EXTERNAL TABLE IF NOT EXISTS employees_info
(
id INT,
name STRING,
usd_flag STRING,
salary DOUBLE,
deductions MAP<STRING, DOUBLE>,
address STRING,
entrytime STRING
)
-- 指定行中各字段分隔符.
-- "delimited fields terminated by"指定列与列之间的分隔符为','.
ROW FORMAT delimited fields terminated by ','
-- 指定表的存储格式为TEXTFILE.
STORED AS TEXTFILE;

-- 使用CREATE Like创建表.
CREATE TABLE employees_like LIKE employees_info;

-- 使用DESCRIBE查看employees_info、employees_like表结构.
DESCRIBE employees_info;
DESCRIBE employees_like;

-- 创建内部表employees_info.
CREATE TABLE IF NOT EXISTS employees_info
(
id INT,
name STRING,
usd_flag STRING,
salary DOUBLE,
deductions MAP<STRING, DOUBLE>,
address STRING,
entrytime STRING
)
-- 指定行中各字段分隔符.
-- "delimited fields terminated by"指定列与列之间的分隔符为','.
ROW FORMAT delimited fields terminated by ','
-- 指定表的存储格式为TEXTFILE.
STORED AS TEXTFILE;
```

扩展应用

- 创建分区表

  一个表可以拥有一个或者多个分区，每个分区以文件夹的形式单独存在表文件夹的目录下。对分区内数据进行查询，可缩小查询范围，加快数据的检索速度和可对数据按照一定的条件进行管理。

  分区是在创建表的时候用PARTITIONED BY子句定义的。

  ```
  CREATE EXTERNAL TABLE IF NOT EXISTS employees_info_extended
  (
  id INT,
  name STRING,
  usd_flag STRING,
  salary DOUBLE,
  deductions MAP<STRING, DOUBLE>,
  ```



```
address STRING
)
-- 使用关键字PARTITIONED BY指定分区列名及数据类型 .
PARTITIONED BY (entrytime STRING)
STORED AS TEXTFILE;
```

- 更新表的结构

  一个表在创建完成后，还可以使用ALTER TABLE执行增、删字段，修改表属性，添加分区等操作。

  ```
  -- 为表employees_info_extended增加tel_phone、email字段.
  ALTER TABLE employees_info_extended ADD COLUMNS (tel_phone STRING, email STRING);
  ```

- Impala使用OBS存储。

  需要在集群管理页面Manager里面设置指定的参数到“core-site.xml”，AK/SK可登录“OBS控制台”，进入“我的凭证”页面获取。

  ```
  fs.obs.access.key=AK;
  fs.obs.secret.key=SK;
  fs.obs.endpoint=endpoint;
  ```

  新建表的存储类型为obs。

  ```
  create table obs(c1 string, c2 string) stored as parquet location 'obs://obs-lmm/hive/orctest'
  tblproperties('orc.compress'='SNAPPY');
  ```

  说明

  当前Impala使用OBS存储时，同一张表中，不支持分区和表存储位置处于不同的桶中。

  例如：创建分区表指定存储位置为OBS桶1下的文件夹，此时修改表分区存储位置的操作将不会生效，在实际插入数据时以表存储位置为准。

  1. 创建分区表并指定表存储路径。
     ```
     create table table_name(id int,name string,company string) partitioned by(dt date) row
     format delimited fields terminated by ',' stored as textfile location "obs://OBS桶1/桶下文件夹";
     ```
  2. 修改此表分区位置到另外一个桶下，此时该修改不会生效。
     ```
     alter table table_name partition(dt date) set location "obs://OBS桶2/桶下文件夹";
     ```

18.3.3 加载 Impala 数据

功能简介

本小节介绍了如何使用Impala SQL向已有的表employees_info中加载数据。从本节中可以掌握如何从集群中加载数据。

样例代码

```
-- 从本地文件系统/opt/impala_examples_data/目录下将employee_info.txt加载进employees_info表中.
LOAD DATA LOCAL INPATH '/opt/impala_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;

-- 从HDFS上/user/impala_examples_data/employee_info.txt加载进employees_info表中.
LOAD DATA INPATH '/user/impala_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;
```

说明

加载数据的实质是将数据复制到HDFS上指定表的目录下。

“LOAD DATA LOCAL INPATH”命令可以完成从本地文件系统加载文件到Impala的需求，但是当指定“LOCAL”时，这里的路径指的是当前连接的“Impalad”的本地文件系统的路径。

18.3.4 查询 Impala 数据

功能简介

本小节介绍了如何使用Impala SQL对数据进行查询分析。从本节中可以掌握如下查询分析方法。

- SELECT查询的常用特性，如JOIN等。
- 加载数据进指定分区。
- 如何使用Impala自带函数。
- 如何使用自定义函数进行查询分析，如何创建、定义自定义函数请见**Impala样例程序指导**。

样例代码

```
-- 查看薪水支付币种为美元的雇员联系方式.
SELECT
a.name,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b  ON(a.id = b.id) WHERE usd_flag='D';

-- 查询入职时间为2014年的雇员编号、姓名等字段，并将查询结果加载进表employees_info_extended中的入职时间为2014的分区中.
INSERT OVERWRITE TABLE employees_info_extended PARTITION (entrytime = '2014')
SELECT
a.id,
a.name,
a.usd_flag,
a.salary,
a.deductions,
a.address,
b.tel_phone,
b.email
FROM employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE a.entrytime = '2014';

-- 使用Impala中已有的函数COUNT()，统计表employees_info中有多少条记录.
SELECT COUNT(*) FROM employees_info;

-- 查询使用以“cn”结尾的邮箱的员工信息.
SELECT a.name, b.tel_phone FROM  employees_info a JOIN employees_contact b ON (a.id = b.id) WHERE b.email like '%cn';
```

扩展使用

自定义函数，具体内容请参见**开发Impala用户自定义函数**。

18.3.5 开发 Impala 用户自定义函数

当Impala的内置函数不能满足需要时，可以通过编写用户自定义函数UDF（User-Defined Functions）插入自己的处理代码并在查询中使用它们。

按实现方式，UDF有如下分类：

- 普通的UDF，用于操作单个数据行，且产生一个数据行作为输出。
- 用户定义聚集函数UDAF（User-Defined Aggregating Functions），用于接受多个输入数据行，并产生一个输出数据行。
- 用户定义表生成函数UDTF(User-Defined Table-Generating Functions)，用于操作单个输入行，产生多个输出行。**Impala不支持该类UDF**。

按使用方法，UDF有如下分类：

- 临时函数，只能在当前会话使用，重启会话后需要重新创建。
- 永久函数，可以在多个会话中使用，不需要每次创建。

Impala支持开发Java UDF，也可以复用Hive开发的UDFs，前提是使用Impala支持的数据类型。Impala支持的数据类型请参考**http://impala.apache.org/docs/build3x/html/topics/impala_datatypes.html**。

此外，Impala还支持C++编写的UDF，性能上比Java UDF更好。

使用示例

以下为复用lower()函数的示例。

```
[localhost:21000] > create database udfs;
[localhost:21000] > use udfs;
[localhost:21000] > create function my_lower(string)
returns string location '/user/hive/udfs/hive.jar'
symbol='org.apache.hadoop.hive.ql.udf.UDFLower';
[localhost:21000] > select my_lower('Some String NOT ALREADY LOWERCASE');
+----------------------------------------------------+
| udfs.my_lower('some string not already lowercase') |
+----------------------------------------------------+
| some string not already lowercase |
+----------------------------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create table t2 (s string);
[localhost:21000] > insert into t2 values ('lower'),('UPPER'),('Init cap'),('CamelCase');
Inserted 4 rows in 2.28s
[localhost:21000] > select * from t2;
+-----------+
| s |
+-----------+
| lower |
| UPPER |
| Init cap |
| CamelCase |
+-----------+
Returned 4 row(s) in 0.47s
[localhost:21000] > select my_lower(s) from t2;
+------------------+
| udfs.my_lower(s) |
+------------------+
| lower |
| upper |
| init cap |
| camelcase |
+------------------+
Returned 4 row(s) in 0.54s
```

18.3.6 Impala 样例程序指导

功能简介

本小节介绍了如何使用样例程序完成分析任务。本章节以使用JDBC接口提交数据分析任务为例。

样例代码

使用Impala JDBC接口提交数据分析任务，参考样例程序中的JDBCExample.java。

1. 修改以下变量为false，标识连接集群的认证模式为普通模式。

```
// 所连接集群的认证模式是否在安全模式
  boolean isSecureVer = false;
```

2. 定义Impala SQL。Impala SQL必须为单条语句，注意不能包含“;”。

```
  // 定义HQL，不能包含“;”
  String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
          "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};
```

3. 拼接JDBC URL。

```
// 拼接JDBC URL
StringBuilder sBuilder = new StringBuilder(
  "jdbc:hive2://").append("impalad_ip").append("/");

if (isSecurityMode) {
    // 安全模式
    sBuilder.append(";auth=")
            .append(clientInfo.getAuth())
            .append(";principal=")
            .append(clientInfo.getPrincipal())
            .append(";");
} else {
    // 普通模式
    sBuilder.append(";auth=noSasl");
}
String url = sBuilder.toString();
```

说明

直连Impalad实例时，若当前连接的Impalad实例故障则会导致访问Impala失败。

4. 加载Hive JDBC驱动。

```
  // 加载Hive JDBC驱动
  Class.forName(HIVE_DRIVER);
```

5. 填写正确的用户名，获取JDBC连接，确认Impala SQL的类型（DDL/DML），调用对应的接口执行Impala SQL，输出查询的列名和结果到控制台，关闭JDBC连接。

```
  Connection connection = null;
    try {
      // 获取JDBC连接
      // 第二个参数需要填写正确的用户名，否则会以匿名用户(anonymous)登录
      connection = DriverManager.getConnection(url, "userName", "");

      // 建表
      // 表建完之后，如果要往表中导数据，可以使用LOAD语句将数据导入表中，比如从HDFS上将数据导
入表:
      //load data inpath '/tmp/employees.txt' overwrite into table employees_info;
      execDDL(connection,sqls[0]);
      System.out.println("Create table success!");

      // 查询
      execDML(connection,sqls[1]);

      // 删表
      execDDL(connection,sqls[2]);
      System.out.println("Delete table success!");
    }
    finally {
      // 关闭JDBC连接
      if (null != connection) {
        connection.close();
      }

 public static void execDDL(Connection connection, String sql)
   throws SQLException {
     PreparedStatement statement = null;
     try {
       statement = connection.prepareStatement(sql);
```



```
    statement.execute();
  }
  finally {
    if (null != statement) {
      statement.close();
    }
  }
}

public static void execDML(Connection connection, String sql) throws SQLException {
  PreparedStatement statement = null;
  ResultSet resultSet = null;
  ResultSetMetaData resultMetaData = null;

  try {
    // 执行Impala SQL
    statement = connection.prepareStatement(sql);
    resultSet = statement.executeQuery();

    // 输出查询的列名到控制台
    resultMetaData = resultSet.getMetaData();
    int columnCount = resultMetaData.getColumnCount();
    for (int i = 1; i <= columnCount; i++) {
      System.out.print(resultMetaData.getColumnLabel(i) + '\t');
    }
    System.out.println();

    // 输出查询结果到控制台
    while (resultSet.next()) {
      for (int i = 1; i <= columnCount; i++) {
        System.out.print(resultSet.getString(i) + '\t');
      }
      System.out.println();
    }
  }
  finally {
    if (null != resultSet) {
      resultSet.close();
    }

    if (null != statement) {
      statement.close();
    }
  }
}
```

18.4 调测 Impala 应用

18.4.1 在 Windows 中调测 Impala JDBC 应用

步骤1 运行样例。

导入和修改样例后，即可在开发环境中，右击“ExampleMain.java”，选择“ExampleMain.main()”运行对应的应用程序工程。

说明

使用Windows访问MRS集群来操作Impala，有如下两种方式。

方法一：申请一台Windows的ECS访问MRS集群操作Impala，在安装开发环境后可直接运行样例代码。

1. 在“现有集群”列表中，单击已创建的集群名称。

   记录集群的“可用分区”、“虚拟私有云”，以及Master节点的“默认安全组”。

2. 在弹性云服务管理控制台，创建一个新的弹性云服务器。

   弹性云服务器的“可用分区”、“虚拟私有云”、“安全组”，需要和待访问集群的配置相同。

   选择一个Windows系统的公共镜像。

   其他配置参数详细信息，请参见**购买弹性云服务器**。

方法二：使用本机访问MRS集群操作Impala，在安装开发环境后并完成以下步骤后再运行样例代码。

为任意一个Core节点绑定弹性公网IP，完成后将该IP地址配置在开发样例的client.properties下的impala-server配置项中，用于访问Impala服务、提交SQL语句。

1. 在弹性公网IP页面申请一个弹性公网IP，并为集群的任一Core节点绑定该弹性公网IP，具体请参考**申请并绑定弹性公网IP**。
2. 为MRS集群开放安全组规则。在集群Master节点和Core节点的安全组添加安全组规则使弹性云服务器可以访问集群，具体请参考**配置安全组规则**。

样例中的client.properties配置如下：

```
impala-server = XX.XX.XX.XX:21050  ##指定要连接的impalad实例所在Core节点绑定的服务地址，方式二需要填写步骤1中绑定的弹性公网IP
```

步骤2 查看结果。

查看样例代码中的Impala SQL所查询出的结果，运行成功结果会有如下信息。

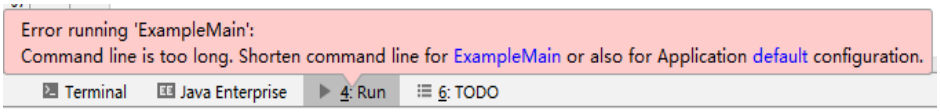
ExampleMain运行及结果查看。

```
Create table success!
_c0
0
Delete table success!
```


说明

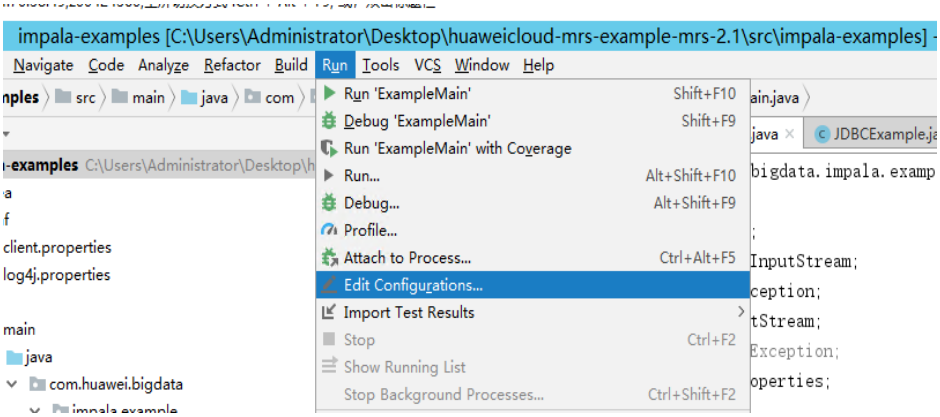
运行样例出错，出现如下提示：

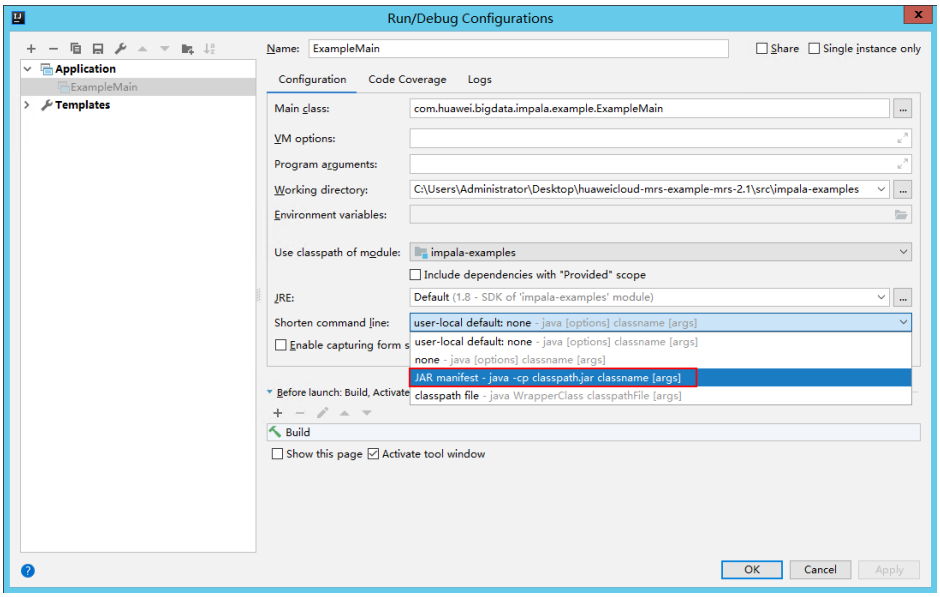
Error running 'ExampleMain': Command line is too long. Shorten command line for ServiceStarter or also for Application default configuration.



解决办法：

在Intellij中的配置Edit Configurations 中设置shorten command line 即可。





----结束

18.4.2 在 Linux 中调测 Impala JDBC 应用

前提条件

已安装MRS客户端，具体请参见：

- 集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**。
- MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

操作步骤

步骤1 在运行调测环境上创建一个目录作为运行目录，如“/opt/impala_examples”，并在该目录下创建子目录“conf”，将样例代码文件夹impala-examples-normal中的client.properties上传到linux系统上的样例代码文件夹中的/opt/impala_examples/conf文件夹，并在client.properties中填入impalad的ip地址。

步骤2 在cmd或Intellij中执行**mvn package**，在工程target目录下获取jar包，比如“impala-examples-mrs-2.1-jar-with-dependencies.jar”，复制到“/opt/impala_examples”下。

步骤3 **source**客户端中的bigdata_env，在Linux环境下执行如下命令运行样例程序。

```
chmod +x /opt/impala_examples -R
cd /opt/impala_examples
java -cp impala-examples-mrs-2.1-jar-with-dependencies.jar
com.huawei.bigdata.impala.example.ExampleMain
```

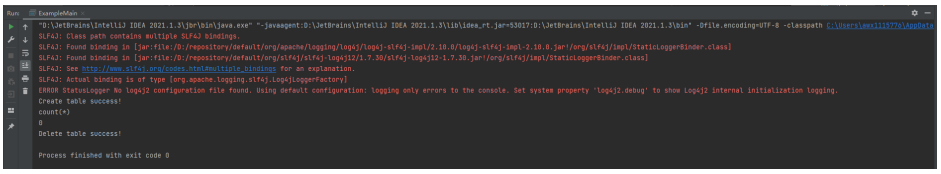
步骤4 在命令行终端查看样例代码中的Impala SQL所查询出的结果。

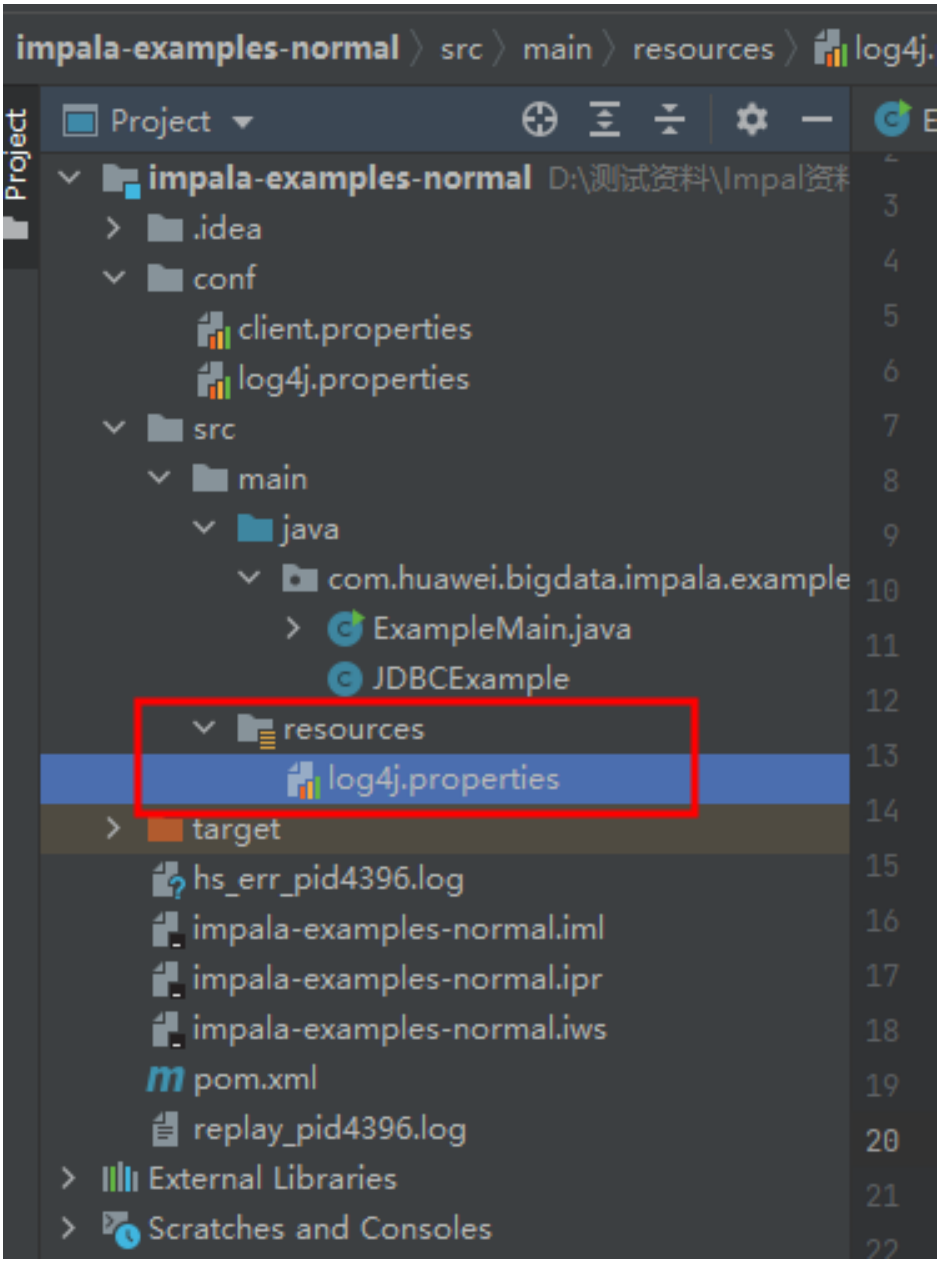
Linux环境运行成功结果会有如下信息。

```
Create table success!
_c0
0
Delete table success!
```


说明

如果出现如下报错提示，请客户根据log报错信息自行配置log4j2信息。





----结束

18.5 Impala 应用开发常见问题

18.5.1 Impala JDBC 接口介绍

Impala使用Hive的JDBC接口，Hive JDBC接口遵循标准的JAVA JDBC驱动标准，详情请参见JDK1.7 API。

说明

Impala并不能支持所有的Hive JDBC标准API。执行某些操作会产生“Method not supported”的SQLException异常。

18.5.2 Impala SQL 接口介绍

Impala SQL提供对HiveQL的高度兼容性，详情请参见**https://impala.apache.org/docs/build/html/topics/impala_langref.html**。

18.6 Impala 开发规范

18.6.1 Impala 开发规则

Hive JDBC 驱动的加载

客户端程序以JDBC的形式连接Impalad时，需要首先加载Hive的JDBC驱动类org.apache.hive.jdbc.HiveDriver。

所以在客户端程序开始前，必须先使用当前类加载器加载该驱动类。

如果classpath下没有相应的jar包，则客户端程序抛出Class Not Found异常并退出。

如下：

```
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
```

获取数据库连接

使用JDK的驱动管理类java.sql.DriverManager来获取一个Impalad的数据库连接。

Impalad的数据库URL为url="jdbc:hive2://xxx.xxx.xxx.xxx:21050;auth=noSasl;principal=impala/hadoop.hadoop.com@HADOOP.COM;user.principal=impala/hadoop.hadoop.com;user.keytab=conf/impala.keytab";

以上已经经过安全认证，所以用户名和密码为null或者空。

// 建立连接

```
connection = DriverManager.getConnection(url, "", "");
```

执行 Impala SQL

执行Impala SQL，注意Impala SQL不能以";"结尾。

正确示例：

```
String sql = "SELECT COUNT(*) FROM employees_info";
Connection connection = DriverManager.getConnection(url, "", "");
```

```
PreparedStatement statement = connection.prepareStatement(sql);
resultSet = statement.executeQuery();
```

错误示例：

```
String sql = "SELECT COUNT(*) FROM employees_info;";
Connection connection = DriverManager.getConnection(url, "", "");
PreparedStatement statement = connection.prepareStatement(sql);
resultSet = statement.executeQuery();
```

Impala SQL 语法规则之判空

判断字段是否为“空”，即没有值，使用“is null”；判断不为空，即有值，使用“is not null”。

要注意的是，在Impala SQL中String类型的字段若是空字符串， 即长度为0，那么对它进行is null的判断结果是False。此时应该使用“col = '' ”来判断空字符串；使用“col != '' ”来判断非空字符串。

正确示例：

```
select * from default.tbl_src where id is null;
select * from default.tbl_src where id is not null;
select * from default.tbl_src where name = '';
select * from default.tbl_src where name != '';
```

错误示例：

```
select * from default.tbl_src where id = null;
select * from default.tbl_src where id != null;
select * from default.tbl_src where name is null;
select * from default.tbl_src where name is not null;注：表tbl_src的id字段为Int类型，name字段为String类型。
```

多线程安全登录方式

如果有多线程进行login的操作，当应用程序第一次登录成功后，所有线程再次登录时应该使用relogin的方式。

login的代码样例：

```
 private Boolean login(Configuration conf){
   boolean flag = false;
   UserGroupInformation.setConfiguration(conf);

   try {
     UserGroupInformation.loginUserFromKeytab(conf.get(PRINCIPAL), conf.get(KEYTAB));
     System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
     flag = true;
   } catch (IOException e) {
     e.printStackTrace();
   }
   return flag;

 }
```

relogin的代码样例：

```
public Boolean relogin(){
       boolean flag = false;
       try {

         UserGroupInformation.getLoginUser().reloginFromKeytab();
         System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
```

```
            flag = true;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return flag;
    }
```

避免对同一张表同时进行读写操作

目前的版本中，Hive不支持并发操作，需要避免对同一张表同时进行读写操作，否则会出现查询结果不准确，甚至任务失败的情况。

18.6.2 Impala 开发建议

Impala SQL 编写之不支持隐式类型转换

查询语句使用字段的值做过滤时，不支持使用Hive类似的隐式类型转换来编写Impala SQL：

Impala示例：

```
select * from default.tbl_src where id = 10001;
select * from default.tbl_src where name = 'TestName';
```

Hive示例(支持隐式类型转换)：

```
select * from default.tbl_src where id = '10001';
select * from default.tbl_src where name = TestName;
```

说明

表tbl_src的id字段为Int类型，name字段为String类型。

JDBC 超时限制

Impala使用Hive提供的JDBC，Hive提供的JDBC实现有超时限制，默认是5分钟，用户可以通过java.sql.DriverManager.setLoginTimeout(int seconds)设置，seconds的单位为秒。

18.6.3 Impala 开发示例

JDBC 二次开发示例代码

以下示例代码主要功能如下。

1. 普通(非Kerberos)模式下，使用用户名和密码进行登录，如不指定用户，则匿名登录；
2. 在JDBC URL地址中提供登录Kerberos用户的principal，程序自动完成安全登录、建立Impala连接。
3. 执行创建表、查询和删除三类Impala SQL语句。

```
package com.huawei.bigdata.impala.example;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
```

```
import java.sql.SQLException;

/**
 * Simple example for hive jdbc.
 */
public class JDBCExample {
    private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";
    private ClientInfo clientInfo;
    private boolean isSecurityMode;
    public JDBCExample(ClientInfo clientInfo, boolean isSecurityMode){
        this.clientInfo = clientInfo;
        this.isSecurityMode = isSecurityMode;
    }

    /**
     *
     * @throws ClassNotFoundException
     * @throws SQLException
     */
    public void run() throws ClassNotFoundException, SQLException {

        //Define hive sql, the sql can not include ";"
        String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
                "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};

        StringBuilder sBuilder = new StringBuilder(
                "jdbc:hive2://").append(clientInfo.getImpalaServer()).append("/");

        if (isSecurityMode) {
            sBuilder.append(";auth=")
                    .append(clientInfo.getAuth())
                    .append(";principal=")
                    .append(clientInfo.getPrincipal())
                    .append(";");
        } else {
            sBuilder.append(";auth=noSasl");
        }
        String url = sBuilder.toString();
        Class.forName(HIVE_DRIVER);
        Connection connection = null;
        try {
            /**
             * Get JDBC connection, If not use security mode, need input correct username,
             * otherwise, wil login as "anonymous" user
             */
            //connection = DriverManager.getConnection(url, "", "");
            connection = DriverManager.getConnection(url);
            /**
             * Run the create table sql, then can load the data if needed. eg.
             * "load data inpath '/tmp/employees.txt' overwrite into table employees_info;"
             */
            execDDL(connection,sqls[0]);
            System.out.println("Create table success!");

            execDML(connection,sqls[1]);

            execDDL(connection,sqls[2]);
            System.out.println("Delete table success!");
        }
        finally {
            if (null != connection) {
                connection.close();
            }
        }
    }

    public static void execDDL(Connection connection, String sql)
            throws SQLException {
        PreparedStatement statement = null;
```



```
        try {
            statement = connection.prepareStatement(sql);
            statement.execute();
        }
        finally {
            if (null != statement) {
                statement.close();
            }
        }
    }

    public static void execDML(Connection connection, String sql) throws SQLException {
        PreparedStatement statement = null;
        ResultSet resultSet = null;
        ResultSetMetaData resultMetaData = null;

        try {
            statement = connection.prepareStatement(sql);
            resultSet = statement.executeQuery();

            /**
             * Print the column name to console
             */
            resultMetaData = resultSet.getMetaData();
            int columnCount = resultMetaData.getColumnCount();
            for (int i = 1; i <= columnCount; i++) {
                System.out.print(resultMetaData.getColumnLabel(i) + '\t');
            }
            System.out.println();

            /**
             * Print the query result to console
             */
            while (resultSet.next()) {
                for (int i = 1; i <= columnCount; i++) {
                    System.out.print(resultSet.getString(i) + '\t');
                }
                System.out.println();
            }
        }
        finally {
            if (null != resultSet) {
                resultSet.close();
            }

            if (null != statement) {
                statement.close();
            }
        }
    }
}
```


19 Kafka 开发指南（安全模式）

19.1 Kafka 应用开发简介

Kafka 简介

Kafka是一个分布式的消息发布-订阅系统。它采用独特的设计提供了类似JMS的特性，主要用于处理活跃的流式数据。

Kafka有很多适用的场景：消息队列、行为跟踪、运维数据监控、日志收集、流处理、事件溯源、持久化日志等。

Kafka有如下几个特点：

- 高吞吐量
- 消息持久化到磁盘
- 分布式系统易扩展
- 容错性好
- 支持online和offline场景

接口类型简介

Kafka主要提供了的API主要可分Producer API和Consumer API两大类，均提供有Java API，使用的具体接口说明请参考**Kafka Java API接口介绍**。

常用概念

- **Topic**

  Kafka维护的同一类的消息称为一个Topic。

- **Partition**

  每一个Topic可以被分为多个Partition，每个Partition对应一个可持续追加的、有序不可变的log文件。

- **Producer**

  将消息发往Kafka Topic中的角色称为Producer。

- **Consumer**
  从Kafka topic中获取消息的角色称为Consumer。
- **Broker**
  Kafka集群中的每一个节点服务器称为Broker。
- **keytab file**
  存放用户信息的密钥文件。应用程序采用此密钥文件在集群中进行API方式认证。

19.2 Kafka 应用开发流程介绍

Kafka客户端角色包括Producer和Consumer两个角色，其应用开发流程是相同的。

开发流程中各个阶段的说明如图19-1和表19-1所示。

图 19-1 Kafka 客户端程序开发流程

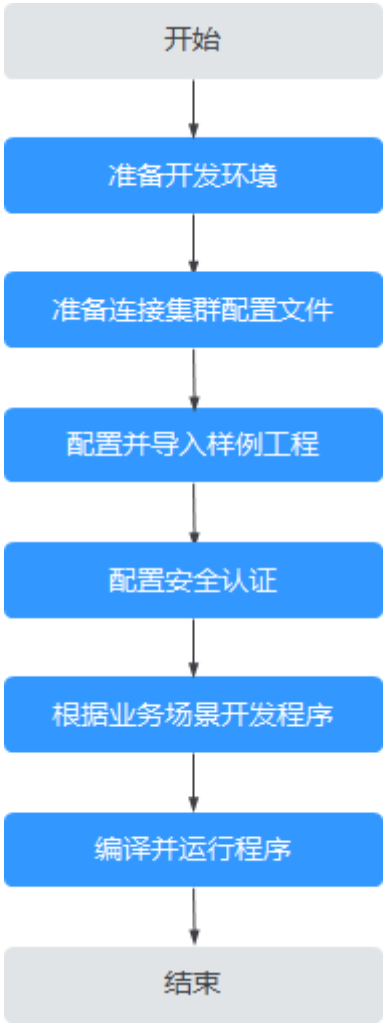


表 19-1 Kafka 客户端开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，推荐使用Java语言进行开发，使用IntelliJ IDEA工具。同时完成JDK、Maven等初始配置。	**准备本地应用开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备连接Kafka集群配置文件**
配置并导入样例工程	Kafka提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。	**导入并配置Kafka样例工程**
配置安全认证	如果您使用的是开启了Kerberos认证的MRS集群，需要进行安全认证。	**配置Kafka应用安全认证**
根据业务场景开发程序	提供了Producer和Consumer相关API的使用样例，包含了API和多线程的使用场景，帮助用户快速熟悉Kafka接口。 将开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。	**开发Kafka应用**
编译与运行程序	指导用户将开发好的程序编译并提交运行并查看结果。	**调测Kafka应用**

19.3 Kafka 样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Kafka相关样例工程：

表 19-2 Kafka 相关样例工程

样例工程位置	描述
kafka-examples	1. 单线程生产数据，相关样例请参考**使用Producer API向安全Topic生产消息**。 2. 单线程消费数据，相关样例请参考**使用Consumer API订阅安全Topic并消费**。 3. 多线程生产数据，相关样例请参考**使用多线程Producer发送消息**。 4. 多线程消费数据，相关样例请参考**使用多线程Consumer消费消息**。 5. 基于KafkaStreams实现WordCount，相关样例请参考**使用KafkaStreams统计数据**

19.4 准备 Kafka 应用开发环境

19.4.1 准备本地应用开发环境

Kafka开发应用时，需要准备的开发和运行环境如**表19-3**所示：

表 19-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装和配置IntelliJ IDEA	开发环境的基本配置。版本要求：JDK使用1.8版本，IntelliJ IDEA使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。

准备项	说明
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 ● X86客户端： – Oracle JDK：支持1.8版本 – IBM JDK：支持1.8.5.11版本 ● TaiShan客户端： OpenJDK：支持1.8.0_272版本 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。 华为提供开源镜像站，各服务样例工程依赖的Jar包通过华为开源镜像站下载，剩余所依赖的开源Jar包请直接从Maven中央库或者其他用户自定义的仓库地址下载，详情请参考**配置华为开源镜像仓**。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

19.4.2 准备连接 Kafka 集群配置文件

准备集群认证用户信息

对于开启Kerberos认证的MRS集群，需提前准备具有相关组件操作权限的用户用于程序认证。

以下Kafka权限配置示例供参考，在实际业务场景中可根据业务需求灵活调整。

步骤1 登录FusionInsight Manager。

步骤2 选择“集群 > 服务 > Kafka > 更多 > 启用Ranger鉴权”，查看该参数是否置灰。

- 是，创建用户并在Ranger中赋予该用户相关操作权限：
  a. 选择“系统 > 权限 > 用户 > 添加用户”，在新增用户界面创建一个机机用户，例如**developuser**。
     “用户组”需加入“kafkaadmin”用户组。
  b. 使用Ranger管理员用户**rangeradmin**登录Ranger管理页面。
  c. 在首页中单击“KAFKA”区域的组件插件名称如“Kafka”。

d. 单击“Policy Name”名称为“all - topic”操作列的 。

e. 在“Allow Conditions”区域新增策略允许条件，“Select User”列勾选**步骤2.a**新建的用户名称，“Permissions”列勾选“Select/Deselect All”。

f. 单击“Save”。

- 否，创建用户并在Manager赋予用户相关操作权限：

a. 选择“系统 > 权限 > 用户 > 添加用户”，在新增用户界面，创建一个机机用户，例如**developuser**，“用户组”需加入“kafkaadmin”用户组。

b. 单击“确定”。

步骤3 使用**admin**用户登录FusionInsight Manager，选择“系统 > 权限 > 用户”，在用户名为**developuser**的操作列选择“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到该用户的“user.keytab”文件与“krb5.conf”文件。

----结束

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。

a. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端配置文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。

b. 进入客户端配置文件解压路径的“Kafka\config”，获取Kafka**表19-4**中相关配置文件。

表 19-4 配置文件

配置文件	作用
client.properties	Kafka的客户端的配置信息。
consumer.properties	Kafka的consumer端配置信息。
kafkaSecurityMode	Kafka服务是否开启安全模式标记文件。
producer.properties	Kafka的producer端配置信息。
server.properties	Kafka的服务端的配置信息。

c. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群，具体操作请参考**配置Windows通过EIP访问安全模式集群Kafka**。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。

  a. 在节点中安装MRS集群客户端。

  例如客户端安装目录为“/opt/client”。

  b. 获取配置文件：

  i. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。

  ii. 以**root**登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“Kafka/config”路径下的**表19-4**中相关配置文件。

  例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

  cd /tmp/FusionInsight-Client

  tar -xvf FusionInsight_Cluster_1_Services_Client.tar

  tar -xvf FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar

  cd FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles

  c. 检查客户端节点网络连接。

  在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

19.4.3 导入并配置 Kafka 样例工程

背景信息

获取Kafka开发样例工程，将工程导入到IntelliJ IDEA开始样例学习及应用程序开发。

前提条件

- 确保本地环境的时间与MRS集群的时间差要小于5分钟，若无法确定，请联系系统管理员。MRS集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备连接Kafka集群配置文件**。

操作步骤

步骤1 获取样例工程文件夹。

参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程文件夹“kafka-examples”。

步骤2 获取配置文件。

若需要在本地Windows调测Kafka样例代码，将**准备集群认证用户信息**时得到的keytab文件“user.keytab”和“krb5.conf”文件以及**准备运行环境配置文件**时获取的所有配置文件放置在样例工程的“kafka-examples\src\main\resources”目录下。

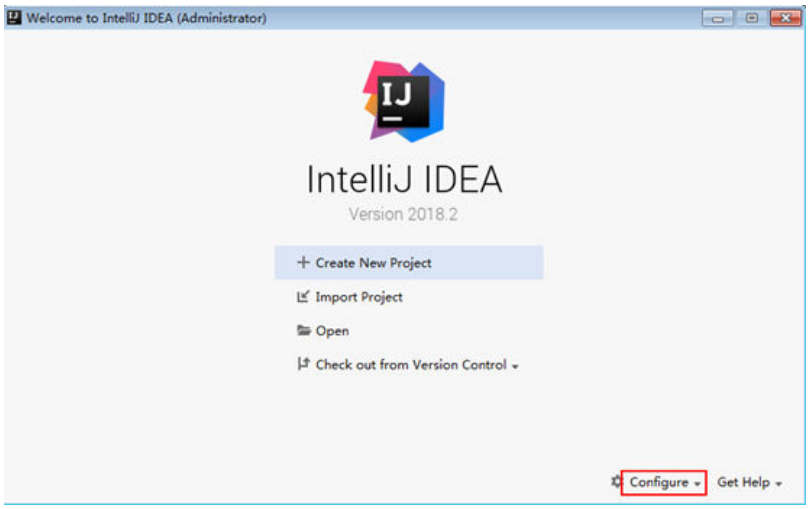
步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

说明

不同的IDEA版本的操作步骤可能存在差异，以实际版本的界面操作为准。

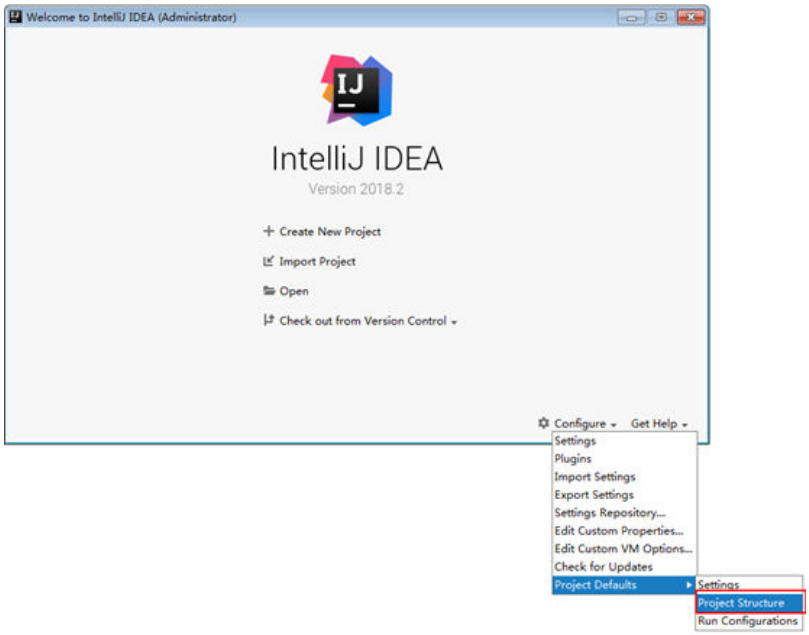
1. 打开IntelliJ IDEA，选择“Configure”。

图 19-2 Quick Start



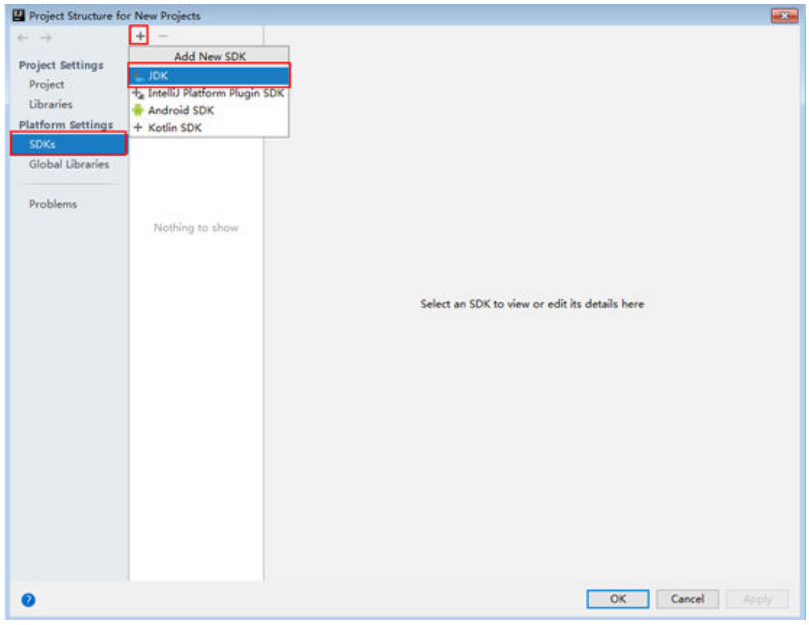
2. 在下拉框中选择“Project Defaults > Project Structure”。

图 19-3 Configure



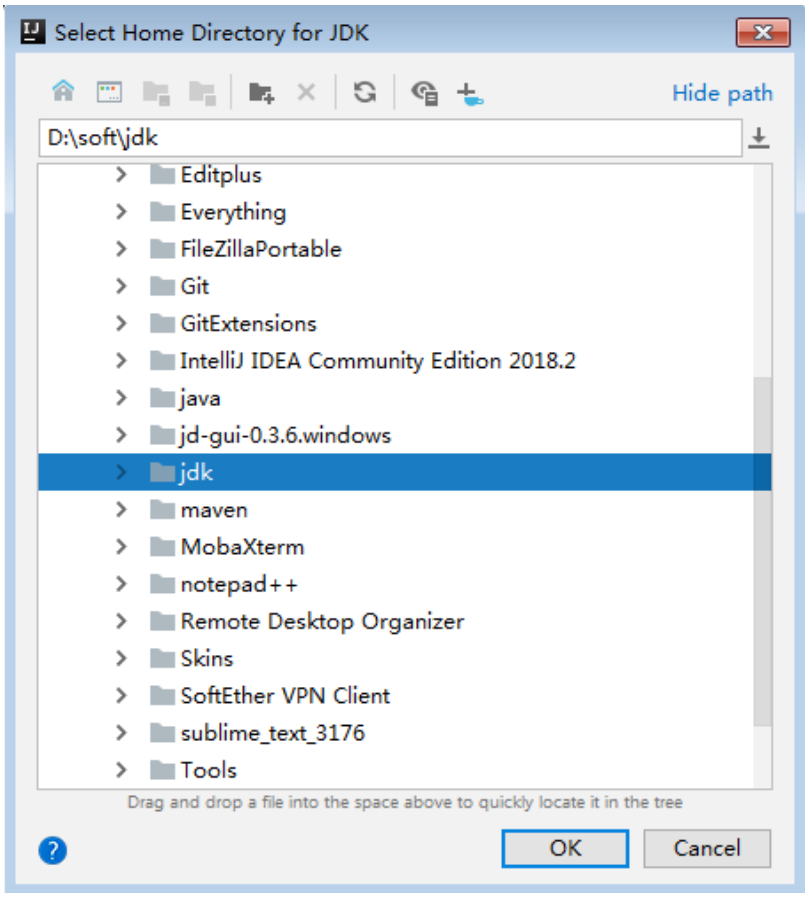
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 19-4 Project Structure for New Projects



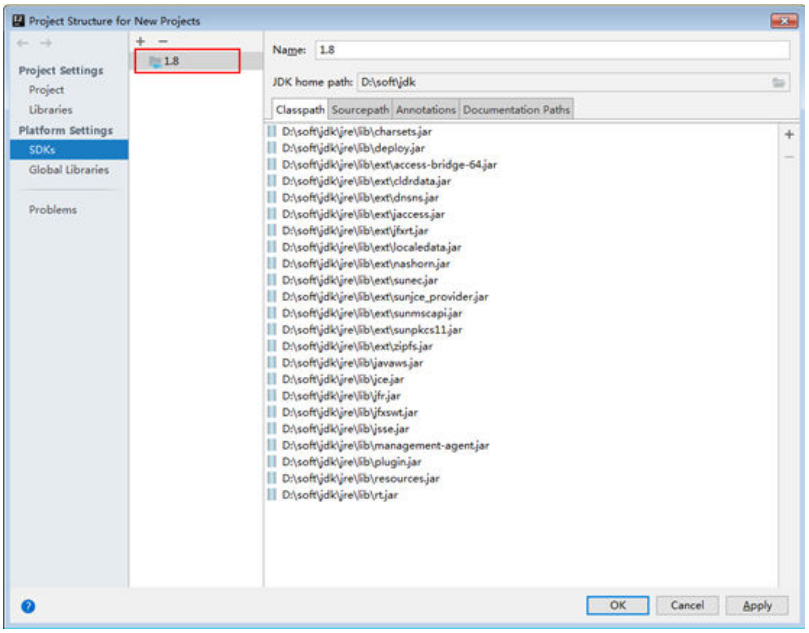
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 19-5 Select Home Directory for JDK



5. 完成JDK选择后，单击“OK”完成配置。

图 19-6 完成 JDK 配置



步骤4 将样例工程导入到IntelliJ IDEA开发环境。

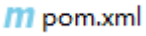
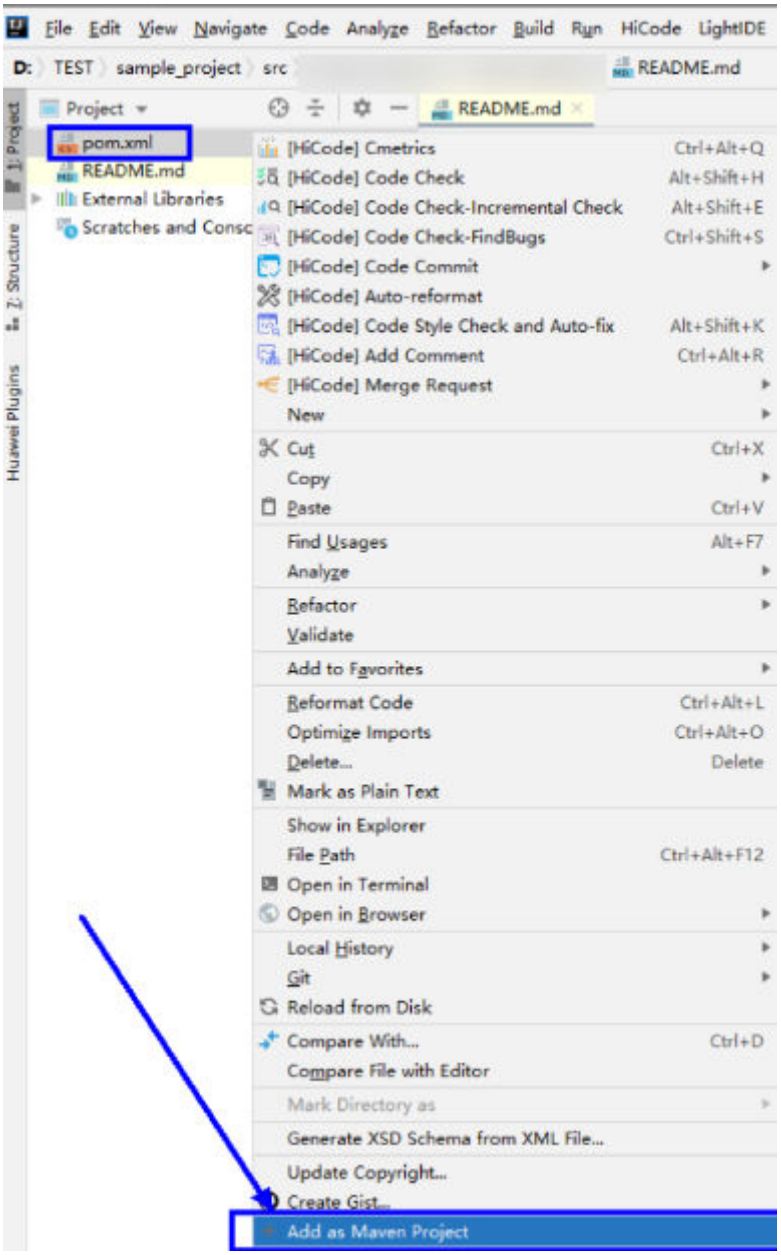
1. 打开IntelliJ IDEA，选择“Open”。
   显示“浏览文件夹”对话框。
2. 选择样例工程文件夹，单击“OK”。
3. 导入结束，IDEA主页显示导入的样例工程。
4. 右键单击“pom.xml”，选择“Add as Maven Project”，将该项目添加为Maven Project。若"pom.xml"图标如  所示，可直接进行下一步骤操作。

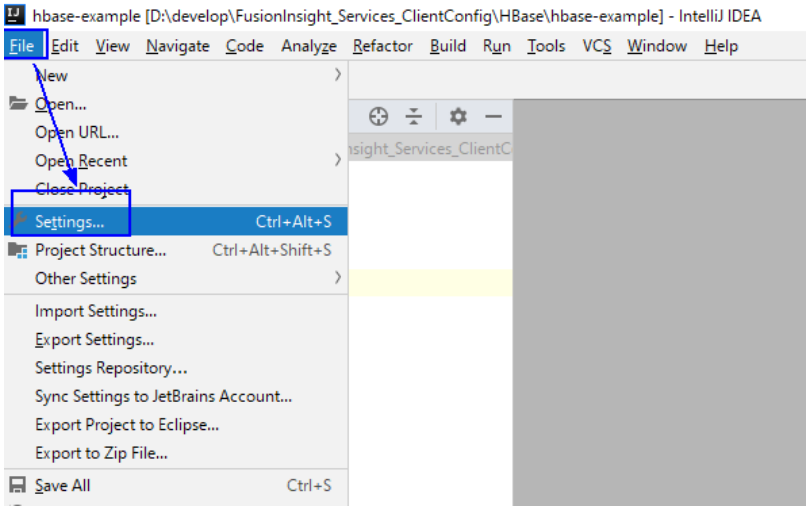
图 19-7 Add as Maven Project



步骤5 设置项目使用的Maven版本。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings...”。

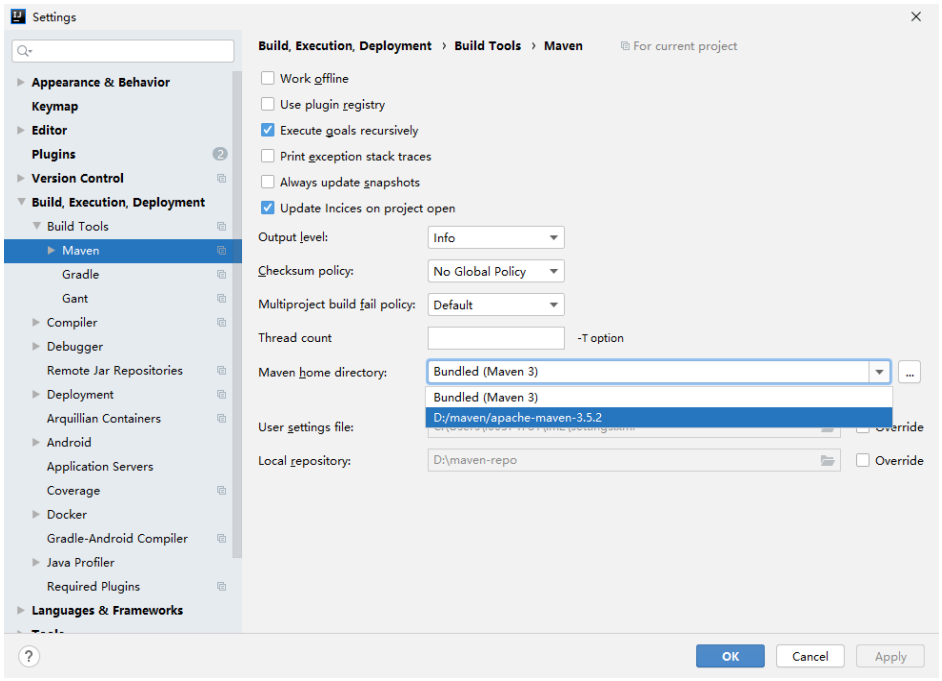
图 19-8 Settings



2. 选择“Build,Execution,Deployment > Maven”，选择“Maven home directory”为本地安装的Maven版本。

   然后根据实际情况设置好“User settings file”和“Local repository”参数，依次单击“Apply”、“OK”。

图 19-9 选择本地 Maven 安装目录



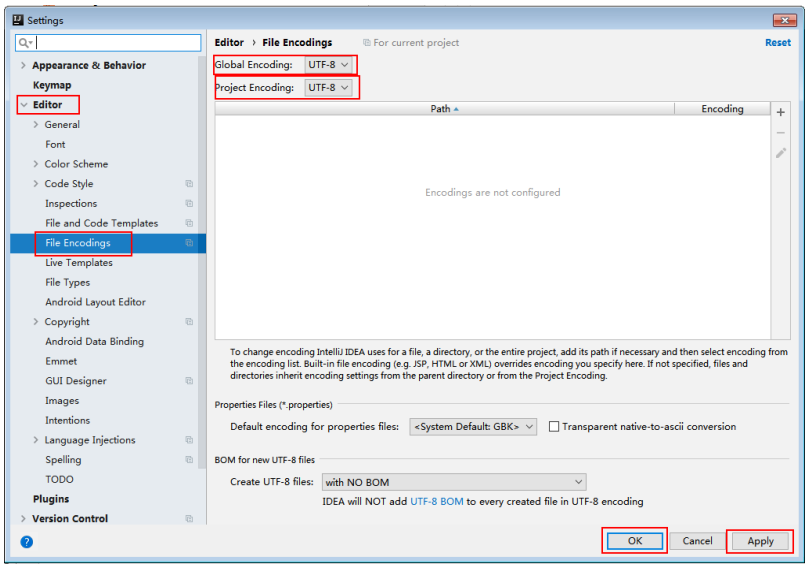
步骤6 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”

   弹出“Settings”窗口。

2. 在左边导航上选择“Editor > File Encodings”，在“Project Encoding”和“Global Encoding”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图19-10所示。

图 19-10 设置 IntelliJ IDEA 的编码格式



----结束

19.4.4 配置 Kafka 应用安全认证

19.4.4.1 使用 Sasl Kerberos 认证

在安全集群环境下，各个组件之间不能够简单的相互通信，而需要在通信之前进行相互认证，以确保通信的安全性。Kafka应用开发需要进行Kafka、ZooKeeper、Kerberos的安全认证，这些安全认证只需要生成一个jaas文件并设置相关环境变量即可。LoginUtil相关接口可以完成这些配置。

代码样例

此代码片段在com.huawei.bigdata.kafka.example.security包的LoginUtil类中。

```
   /**
    * 用户自己申请的机机账号keytab文件名称
    */
   private static final String USER_KEYTAB_FILE = "用户自己申请的机机账号keytab文件名称，例如user.keytab";

   /**
    * 用户自己申请的机机账号名称
    */
   private static final String USER_PRINCIPAL = "用户自己申请的机机账号名称";

public static void securityPrepare() throws IOException
   {
       String filePath = System.getProperty("user.dir") + File.separator + "src" + File.separator + "main" + File.separator + "resources" + File.separator;
       String krbFile = filePath + "krb5.conf";
       String userKeyTableFile = filePath + USER_KEYTAB_FILE;

       //windows路径下分隔符替换
       userKeyTableFile = userKeyTableFile.replace("\\", "\\\\");
       krbFile = krbFile.replace("\\", "\\\\");

       LoginUtil.setKrb5Config(krbFile);
       LoginUtil.setZookeeperServerPrincipal("zookeeper/hadoop.<系统域名>");
```

```
    LoginUtil.setJaasFile(USER_PRINCIPAL, userKeyTableFile);
}
```

说明

用户可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。

19.4.4.2 使用 Kafka Token 认证

场景说明

Token认证机制是一种轻量级身份认证机制，无需访问Kerberos认证，可在API中使用。

代码样例

Token认证机制支持API，用户可在二次开发样例的**Producer()**和**Consumer()**中对其进行配置。

- **Producer()**配置的样例代码如下：

```
public static Properties initProperties() {
    Properties props = new Properties();
    KafkaProperties kafkaProc = KafkaProperties.getInstance();

    // Broker地址列表
    props.put(BOOTSTRAP_SERVER, kafkaProc.getValues(BOOTSTRAP_SERVER, "localhost:21007"));
    // 客户端ID
    props.put(CLIENT_ID, kafkaProc.getValues(CLIENT_ID, "DemoProducer"));
    // Key序列化类
    props.put(KEY_SERIALIZER,
            kafkaProc.getValues(KEY_SERIALIZER,
"org.apache.kafka.common.serialization.StringSerializer"));
    // Value序列化类
    props.put(VALUE_SERIALIZER,
            kafkaProc.getValues(VALUE_SERIALIZER,
"org.apache.kafka.common.serialization.StringSerializer"));
    // 协议类型:当前支持配置为SASL_PLAINTEXT或者PLAINTEXT
    props.put(SECURITY_PROTOCOL, kafkaProc.getValues(SECURITY_PROTOCOL, "SASL_PLAINTEXT"));
    // 服务名
    props.put(SASL_KERBEROS_SERVICE_NAME, "kafka");
    // 域名
    props.put(KERBEROS_DOMAIN_NAME, kafkaProc.getValues(KERBEROS_DOMAIN_NAME,
"hadoop.hadoop.com"));
    // 分区类名
    props.put(PARTITIONER_NAME,
            kafkaProc.getValues(PARTITIONER_NAME,
"com.huawei.bigdata.kafka.example.SimplePartitioner"));
    // 生成Token配置
    StringBuilder token = new StringBuilder();
    String LINE_SEPARATOR = System.getProperty("line.separator");
    token.append("org.apache.kafka.common.security.scram.ScramLoginModule
required").append(LINE_SEPARATOR);
    /**
     * 用户自己生成的Token的TOKENID
     */
    token.append("username=\"PPVz2cxuQC-okwJVZnFKFg\"").append(LINE_SEPARATOR);
    /**
     * 用户自己生成的Token的HMAC
     */
    token.append("password=\"pL5nHsIUODg5u0dRM+o62cOIf/j6yATSt6uaPBYfIb29dj/
jbpiAnRGSWDJ6tL4KXo89dot0axcRIDsMagyN4g==\"").append(LINE_SEPARATOR);
    token.append("tokenauth=true;");
    // 用户使用的SASL机制，配置为SCRAM-SHA-512
    props.put("sasl.mechanism", "SCRAM-SHA-512");
```

```
    props.put("sasl.jaas.config", token.toString());

    return props;
}
```

- **Consumer()**配置的样例代码如下：

```
public static Properties initProperties() {
    Properties props = new Properties();
    KafkaProperties kafkaProc = KafkaProperties.getInstance();
        // Broker连接地址
        props.put(BOOTSTRAP_SERVER, kafkaProc.getValues(BOOTSTRAP_SERVER, "localhost:21007"));
        // Group id
        props.put(GROUP_ID, kafkaProc.getValues(GROUP_ID, "DemoConsumer"));
        // 是否自动提交offset
        props.put(ENABLE_AUTO_COMMIT, kafkaProc.getValues(ENABLE_AUTO_COMMIT, "true"));
        // 自动提交offset的时间间隔
        props.put(AUTO_COMMIT_INTERVAL_MS,
kafkaProc.getValues(AUTO_COMMIT_INTERVAL_MS,"1000"));
        // 会话超时时间
        props.put(SESSION_TIMEOUT_MS, kafkaProc.getValues(SESSION_TIMEOUT_MS, "30000"));
        // 消息Key值使用的反序列化类
        props.put(KEY_DESERIALIZER,
            kafkaProc.getValues(KEY_DESERIALIZER,
"org.apache.kafka.common.serialization.StringDeserializer"));
        // 消息内容使用的反序列化类
        props.put(VALUE_DESERIALIZER,
            kafkaProc.getValues(VALUE_DESERIALIZER,
"org.apache.kafka.common.serialization.StringDeserializer"));
        // 安全协议类型
        props.put(SECURITY_PROTOCOL, kafkaProc.getValues(SECURITY_PROTOCOL,
"SASL_PLAINTEXT"));
        // 服务名
        props.put(SASL_KERBEROS_SERVICE_NAME, "kafka");
        // 域名
        props.put(KERBEROS_DOMAIN_NAME, kafkaProc.getValues(KERBEROS_DOMAIN_NAME,
"hadoop.hadoop.com"));
    // 生成Token配置
    StringBuilder token = new StringBuilder();
    String LINE_SEPARATOR = System.getProperty("line.separator");
    token.append("org.apache.kafka.common.security.scram.ScramLoginModule
required").append(LINE_SEPARATOR);
    /**
     * 用户自己生成的Token的TOKENID
     */
    token.append("username=\"PPVz2cxuQC-okwJVZnFKFg\"").append(LINE_SEPARATOR);
    /**
     * 用户自己生成的Token的HMAC
     */
    token.append("password=\"pL5nHsIUODg5u0dRM+o62cOIf/j6yATSt6uaPBYfIb29dj/
jbpiAnRGSWDJ6tL4KXo89dot0axcRIDsMagyN4g==\"").append(LINE_SEPARATOR);
    token.append("tokenauth=true;");
    // 用户使用的SASL机制，配置为SCRAM-SHA-512
    props.put("sasl.mechanism", "SCRAM-SHA-512");
    props.put("sasl.jaas.config", token.toString());

    return props;
}
```

说明

- BOOTSTRAP_SERVERS需根据集群实际情况，配置为Kafka Broker节点的主机名及端口，可通过集群FusionInsight Manager界面中选择“集群 > 服务 > Kafka > 实例”查看。
- SECURITY_PROTOCOL为连接Kafka的协议类型，在本示例中，配置为“SASL_PLAINTEXT”。
- “TOKENID”和“HMAC”参考**Kafka Token认证机制工具使用说明**为用户生成Token时产生。
- 在使用Token认证机制时，需要把Kerberos认证机制注释掉，保证代码运行过程中只使用一个认证机制，如下所示：

```
public static void main(String[] args)
    {
        if (isSecurityModel())
        {
//            try
//            {
//                LOG.info("Securitymode start.");
//
//                //!!注意，安全认证时，需要用户手动修改为自己申请的机机账号
//                securityPrepare();
//            }
//            catch (IOException e)
//            {
//                LOG.error("Security prepare failure.");
//                LOG.error("The IOException occured.", e);
//                return;
//            }
            LOG.info("Security prepare success.");
        }

        // 是否使用异步发送模式
        final boolean asyncEnable = false;
        Producer producerThread = new Producer(KafkaProperties.TOPIC, asyncEnable);
        producerThread.start();
    }
```

19.5 开发 Kafka 应用

19.5.1 Kafka 样例程序开发思路

场景说明

Kafka是一个分布式消息系统，在此系统上您可以做一些消息的发布和订阅操作，假定用户要开发一个Producer，让其每秒向Kafka集群某Topic发送一条消息，另外还需要实现一个Consumer，订阅该Topic，实时消费该类消息。

开发思路

1. 使用Linux客户端创建一个Topic。可参考**Kafka Shell命令介绍**。
2. 开发一个Producer向该Topic生产数据。
3. 开发一个Consumer消费该Topic的数据。

性能调优建议

1. 建议预先创建Topic，根据业务需求合理规划Partition数目，Partition数目限制了消费者的并发数。

2. 消息key值选取一定是可变的，防止由于消息key值不变导致消息分布不均匀。
3. 消费者尽量使用主动提交offset的方式，避免重复消费。

19.5.2 使用 Producer API 向安全 Topic 生产消息

功能简介

下面代码片段在**com.huawei.bigdata.kafka.example.Producer**类的run方法中，用于实现Producer API向安全Topic生产消息。

代码样例

```
/**
 * 生产者线程执行函数,循环发送消息。
 */
public void run() {
    LOG.info("New Producer: start.");
    int messageNo = 1;

    while (messageNo <= MESSAGE_NUM) {
        String messageStr = "Message_" + messageNo;
        long startTime = System.currentTimeMillis();

        // 构造消息记录
        ProducerRecord<Integer, String> record = new ProducerRecord<Integer, String>(topic, messageNo,
messageStr);

        if (isAsync) {
            // 异步发送
            producer.send(record, new DemoCallBack(startTime, messageNo, messageStr));
        } else {
            try {
                // 同步发送
                producer.send(record).get();
                long elapsedTime = System.currentTimeMillis() - startTime;
                LOG.info("message(" + messageNo + ", " + messageStr + ") sent to topic(" + topic + ") in " +
elapsedTime + " ms.");
            } catch (InterruptedException ie) {
                LOG.info("The InterruptedException occured : {}.", ie);
            } catch (ExecutionException ee) {
                LOG.info("The ExecutionException occured : {}.", ee);
            }
        }
        messageNo++;
    }
}
```

19.5.3 使用 Consumer API 订阅安全 Topic 并消费

功能简介

下面代码片段在**com.huawei.bigdata.kafka.example.Consumer**类中，用于实现使用Consumer API订阅安全Topic，并进行消息消费。

代码样例

```
/**
  * Consumer构造函数。
  * @param topic 订阅的Topic名称。
  */

public Consumer(String topic) {
```

```
    super("KafkaConsumerExample", false);
    // 初始化consumer启动所需的配置参数，详见代码。
    Properties props = initProperties();
    consumer = new KafkaConsumer<Integer, String>(props);
    this.topic = topic;
}

public void doWork() {
    // 订阅
    consumer.subscribe(Collections.singletonList(this.topic));
    // 消息消费请求
    ConsumerRecords<Integer, String> records = consumer.poll(waitTime);
    // 消息处理
    for (ConsumerRecord<Integer, String> record : records) {
        LOG.info("[ConsumerExample], Received message: (" + record.key() + ", " + record.value() + ") at
offset " + record.offset());
    }
}
```

19.5.4 使用多线程 Producer 发送消息

功能简介

在**使用Producer API向安全Topic生产消息**基础上，实现了多线程Producer，可启动多个Producer线程，并通过指定相同key值的方式，使每个线程对应向特定Partition发送消息。

下面代码片段在**com.huawei.bigdata.kafka.example.ProducerMultThread**类的run方法中，用于实现多线程生产数据。

代码样例

```
/**
 * 指定Key值为当前ThreadId，发送数据。
 */
public void run()
{
LOG.info("Producer: start.");

    // 用于记录消息条数。
int messageCount = 1;

    // 每个线程发送的消息条数。
int messagesPerThread = 5;
while (messageCount <= messagesPerThread)
    {

        // 待发送的消息内容。
        String messageStr = new String("Message_" + sendThreadId + "_" + messageCount);

        // 此处对于同一线程指定相同Key值，确保每个线程只向同一个Partition生产消息。
        String key = String.valueOf(sendThreadId);

        // 消息发送。
        producer.send(new KeyedMessage<String, String>(sendTopic, key, messageStr));
LOG.info("Producer: send " + messageStr + " to " + sendTopic + " with key: " + key);
        messageCount++;
     }
}
```

19.5.5 使用多线程 Consumer 消费消息

功能简介

在**使用Consumer API订阅安全Topic并消费**基础上，实现了多线程并发消费，可根据Topic的Partition数目启动相应个数的Consumer线程来对应消费每个Partition上的消息。

下面代码片段在**com.huawei.bigdata.kafka.example.ConsumerMultThread**类的run方法中，用于实现对指定Topic的并发消费。

代码样例

```
/**
 * 启动多线程并发消费Consumer。
 */
public void run() {
    LOG.info("Consumer: start.");
    Properties props = Consumer.initProperties();
    // 启动指定个数Consuemr线程来消费
    // 注意：当该参数大于待消费Topic的Partition个数时，多出的线程将无法消费到数据
    for (int threadNum = 0; threadNum < CONCURRENCY_THREAD_NUM; threadNum++) {
        new ConsumerThread(threadNum, topic, props).start();
        LOG.info("Consumer Thread " + threadNum + " Start.");
    }
}

private class ConsumerThread extends ShutdownableThread {
    private int threadNum = 0;
    private String topic;
    private Properties props;
    private KafkaConsumer<String, String> consumer = null;

    /**
     * 消费者线程类构造方法
     *
     * @param threadNum 线程号
     * @param topic     topic
     */
    public ConsumerThread(int threadNum, String topic, Properties props) {
        super("ConsumerThread" + threadNum, true);
        this.threadNum = threadNum;
        this.topic = topic;
        this.props = props;
        this.consumer = new KafkaConsumer<String, String>(props);
    }

    public void doWork() {
        consumer.subscribe(Collections.singleton(this.topic));
        ConsumerRecords<String, String> records = consumer.poll(waitTime);
        for (ConsumerRecord<String, String> record : records) {
            LOG.info("Consumer Thread-" + this.threadNum + " partitions:" + record.partition() + " record: "
                + record.value() + " offsets: " + record.offset());
        }
    }
}
```

19.5.6 使用 KafkaStreams 统计数据

功能简介

以下提供High level KafkaStreams API代码样例及Low level KafkaStreams API代码样例，通过Kafka Streams读取输入Topic中的消息，统计每条消息中的单词个数，从输出Topic消费数据，将统计结果以Key-Value的形式输出，完成单词统计功能。

High level KafkaStreams API 代码样例

下面代码片段在**com.huawei.bigdata.kafka.example.WordCountDemo**类的createWordCountStream方法中。

```
static void createWordCountStream(final StreamsBuilder builder) {
    // 从 input-topic 接收输入记录
    final KStream<String, String> source = builder.stream(INPUT_TOPIC_NAME);

    // 聚合 key-value 键值对的计算结果
    final KTable<String, Long> counts = source
            // 处理接收的记录，根据正则表达式REGEX_STRING进行分割
            .flatMapValues(value ->
Arrays.asList(value.toLowerCase(Locale.getDefault()).split(REGEX_STRING)))
            // 聚合key-value键值对的计算结果
            .groupBy((key, value) -> value)
            // 最终结果计数
            .count();

    // 将计算结果的 key-value 键值对从 output topic 输出
    counts.toStream().to(OUTPUT_TOPIC_NAME, Produced.with(Serdes.String(), Serdes.Long()));
}
// 用户自己申请的机机账号keytab文件名称
private static final String USER_KEYTAB_FILE = "请修改为真实keytab文件名";
// 用户自己申请的机机账号名称
private static final String USER_PRINCIPAL = "请修改为真实用户名称"
```

Low level KafkaStreams API 代码样例

下面代码片段在**com.huawei.bigdata.kafka.example.WordCountProcessorDemo**类中。

```
private static class MyProcessorSupplier implements ProcessorSupplier<String, String> {
    @Override
    public Processor<String, String> get() {
        return new Processor<String, String>() {
            // ProcessorContext实例，它提供对当前正在处理的记录的元数据的访问
            private ProcessorContext context;
            private KeyValueStore<String, Integer> kvStore;

            @Override
            @SuppressWarnings("unchecked")
            public void init(ProcessorContext context) {
                // 在本地保留processor context，因为在punctuate()和commit()时会用到
                this.context = context;
                // 每1秒执行一次punctuate()
                this.context.schedule(Duration.ofSeconds(1), PunctuationType.STREAM_TIME, timestamp -> {
                    try (final KeyValueIterator<String, Integer> iter = kvStore.all()) {
                        System.out.println("----------- " + timestamp + " ----------- ");
                        while (iter.hasNext()) {
                            final KeyValue<String, Integer> entry = iter.next();
                            System.out.println("[" + entry.key + ", " + entry.value + "]");
                            // 将新纪录作为键值对发送到下游处理器
                            context.forward(entry.key, entry.value.toString());
                        }
                    }
                });
                // 检索名称为KEY_VALUE_STATE_STORE_NAME的key-value状态存储区，可用于记忆最近收到的输
入记录等
                this.kvStore = (KeyValueStore<String, Integer>)
context.getStateStore(KEY_VALUE_STATE_STORE_NAME);
            }

            // 对input topic的接收记录进行处理，将记录拆分为单词并计数
            @Override
            public void process(String dummy, String line) {
                String[] words = line.toLowerCase(Locale.getDefault()).split(REGEX_STRING);

                for (String word : words) {
```

```
                Integer oldValue = this.kvStore.get(word);

                if (oldValue == null) {
                    this.kvStore.put(word, 1);
                } else {
                    this.kvStore.put(word, oldValue + 1);
                }
            }
        }

        @Override
        public void close() {
        }
    };
  }
}

// 用户自己申请的机机账号keytab文件名称
private static final String USER_KEYTAB_FILE = "请修改为真实keytab文件名";
// 用户自己申请的机机账号名称
private static final String USER_PRINCIPAL = "请修改为真实用户名称";
```

19.6 调测 Kafka 应用

19.6.1 调测 Kafka Producer 样例程序

前提条件

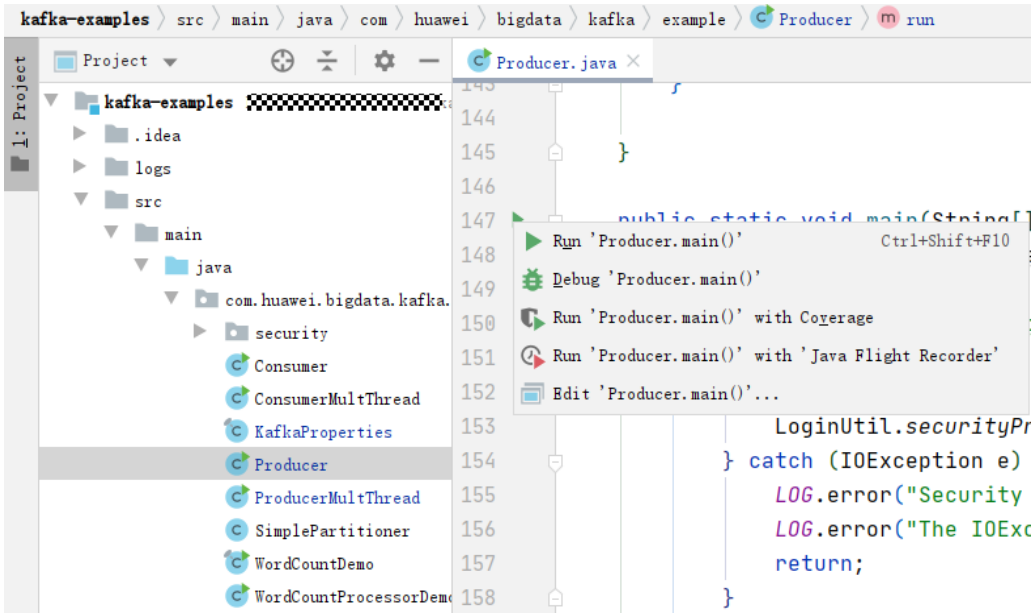
- 如需在Windows调测程序，需要配置Windows通过EIP访问集群Kafka，详情请参见**配置Windows通过EIP访问安全模式集群Kafka**。
- 如需在Linux调测程序，需要确保当前用户对“src/main/resources”目录下和依赖库文件目录下的所有文件，均具有可读权限。同时保证已安装JDK并已设置java相关环境变量。

在 Windows 中调测程序

步骤1 确保本地的hosts文件中配置了远程集群所有主机的主机名和业务IP映射关系。

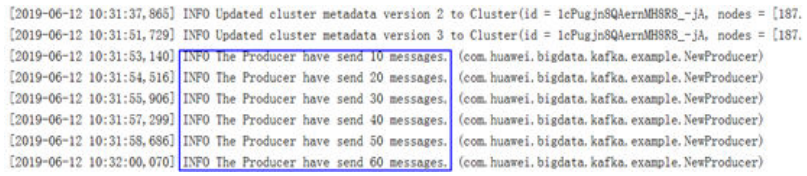
步骤2 通过IntelliJ IDEA可直接运行Producer.java，如**图19-11**所示：

图 19-11 运行 Producer.java



步骤3 运行后弹出控制台窗口，可以查看到Producer正在向默认Topic（example-metric1）发送消息，每发送10条，打印一条日志。

图 19-12 Producer 运行窗口



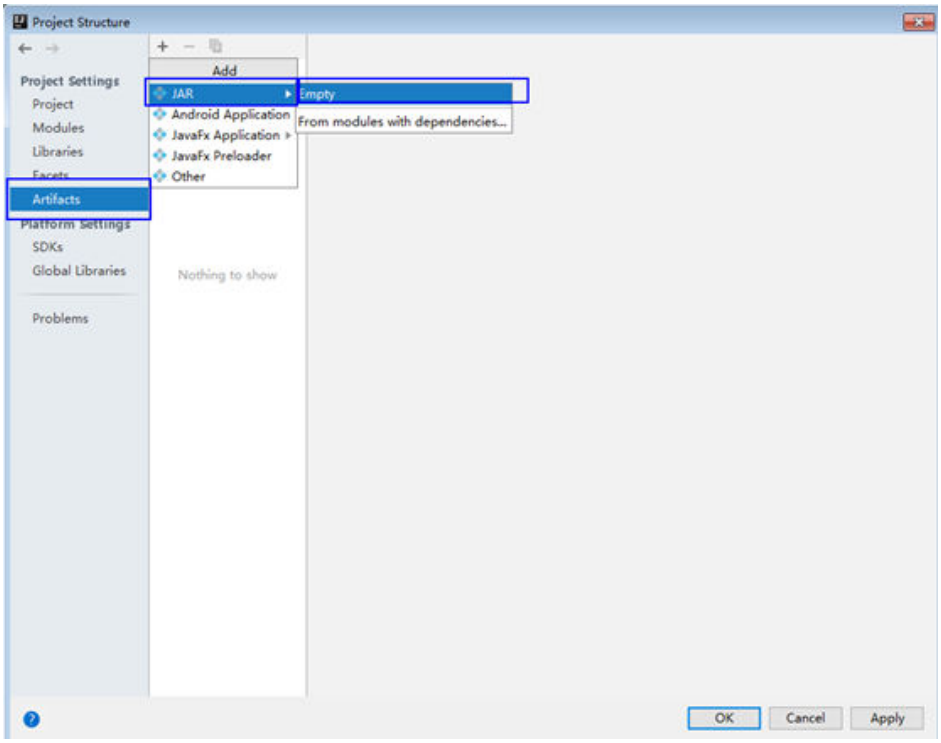
----结束

在 Linux 调测程序

步骤1 在IntelliJ IDEA中，在生成jar包之前配置工程的“Artifacts”信息。

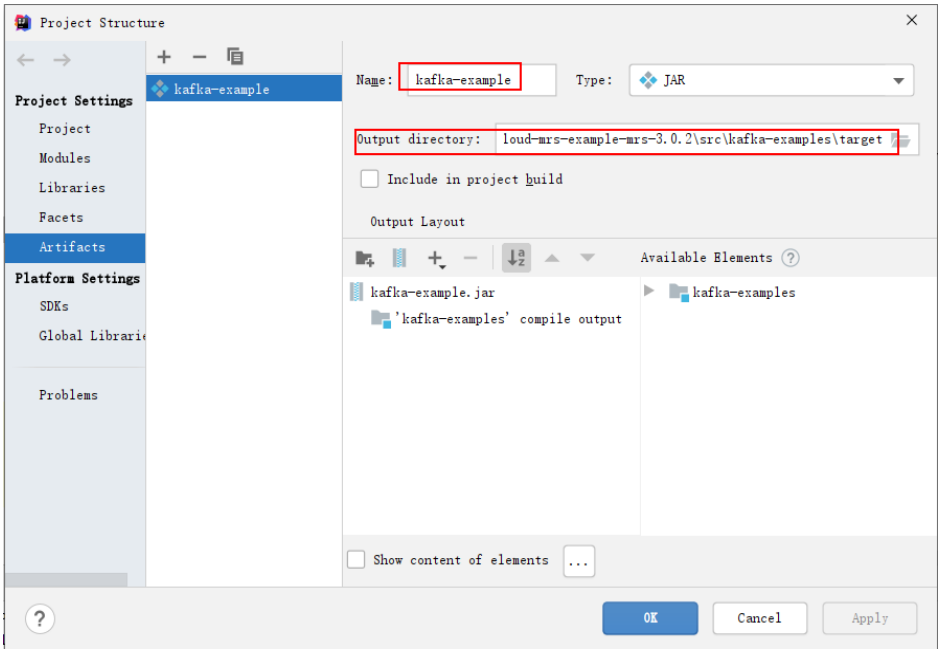
1. 打开IntelliJ IDEA，选择“File > Project Structures...”进入“Project Structure”页面。
2. 在“Project Structure”页面，选择“Artifacts”，单击“+”并选择“JAR > Empty”。

图 19-13 添加 Artifacts



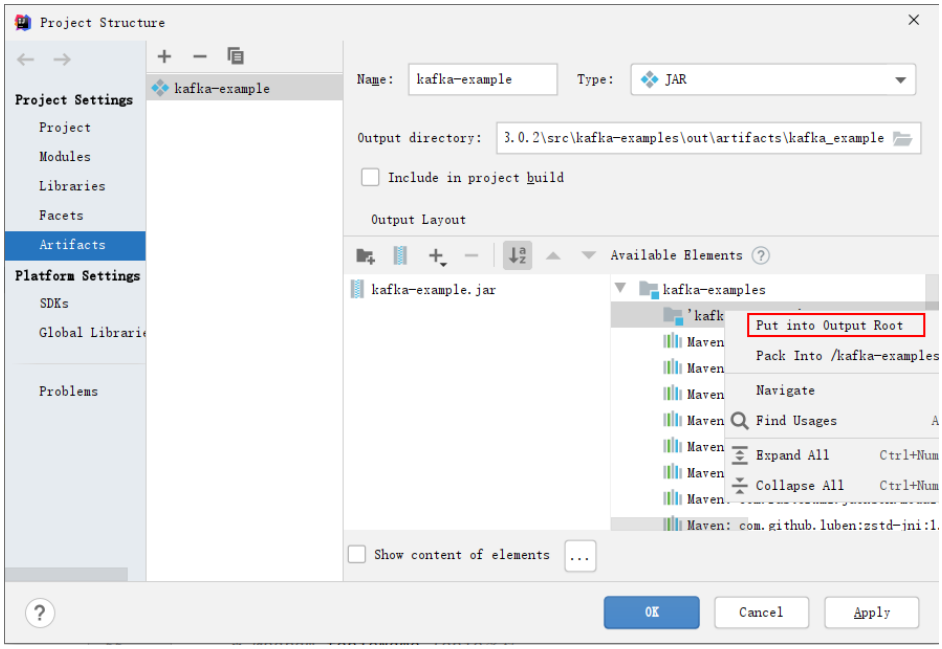
3. 根据实际情况设置jar包的名称、类型以及输出路径。

图 19-14 设置基本信息



4. 选中“'kafka-examples' compile output”，右键选择“Put into Output Root”。然后单击“Apply”。

图 19-15 Put into Output Root

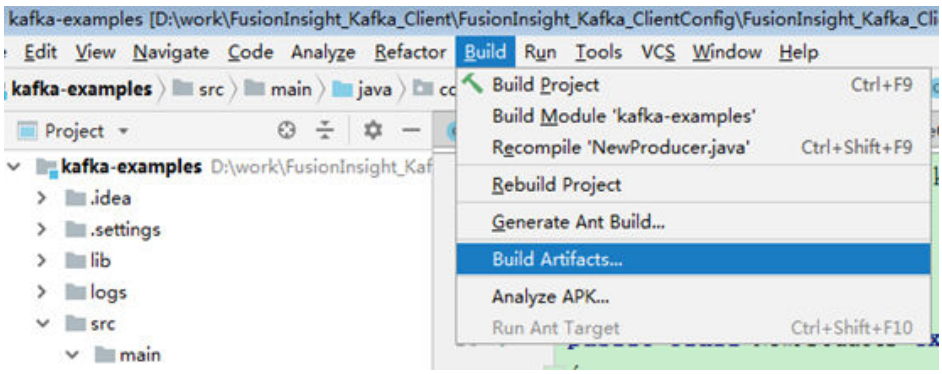


5. 最后单击“OK”完成配置。

步骤2 生成jar包。

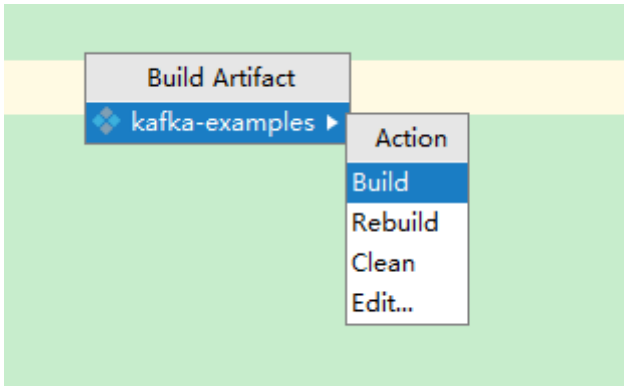
1. 打开IntelliJ IDEA，选择“Build > Build Artifacts...”。

图 19-16 Build Artifacts



2. 在弹出的菜单中，选择“kafka-example:jar > Build”开始生成jar包。

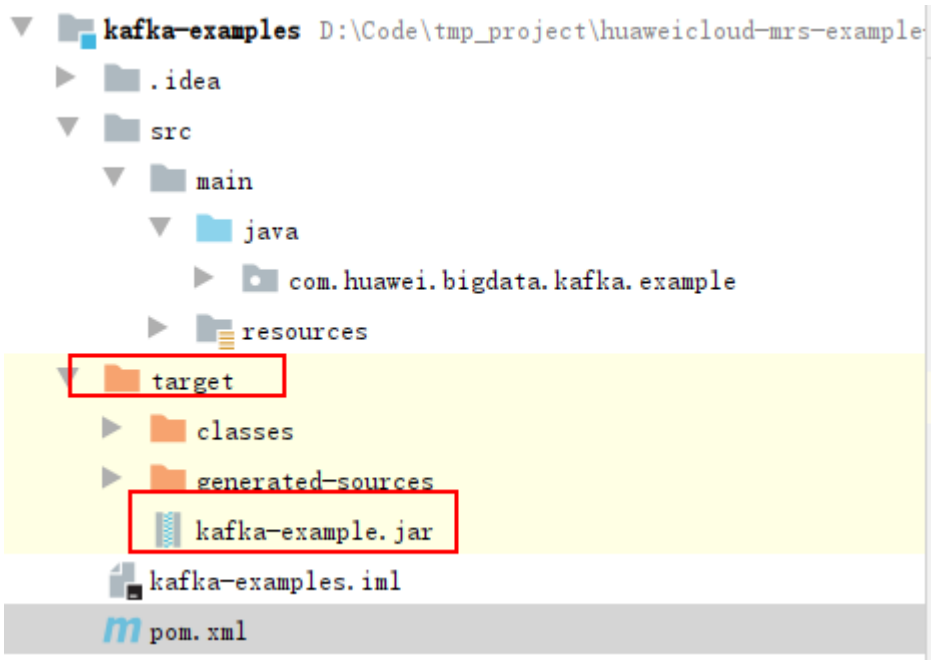
图 19-17 Build



3. 当Event log中出现如下类似日志时，表示Jar包生成成功。您可以在1.c中配置的路径下获取到jar包。
   14:37 Compilation completed successfully in 25 s 991 ms

步骤3 将工程编译生成的jar包复制到“/opt/client/lib”目录下。

图 19-18 文件位置



步骤4 将IntelliJ IDEA工程“src/main/resources”目录下的所有文件复制到与依赖库文件夹同级的目录“src/main/resources”下，即“/opt/client/src/main/resources”。“/opt/client”为客户端安装路径，具体以实际为准。

步骤5 进入目录“/opt/client”，执行如下命令，运行样例工程。

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.Producer

----结束

19.6.2 调测 Kafka Consumer 样例程序

前提条件

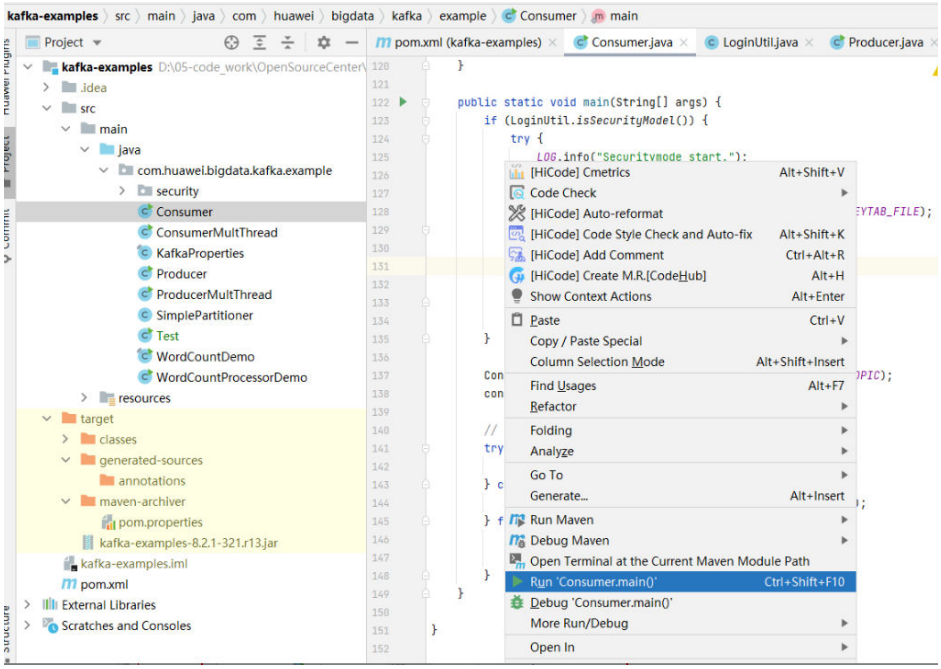
- 如需在Windows调测程序，需要配置Windows通过EIP访问集群Kafka，详情请参见**配置Windows通过EIP访问安全模式集群Kafka**。
- 如需在Linux调测程序，需要确保当前用户对“src/main/resources”目录下和依赖库文件目录下的所有文件，均具有可读权限。同时保证已安装JDK并已设置java相关环境变量。

在 Windows 中调测程序

步骤1 确保本地的hosts文件中配置了远程集群所有主机的主机名和业务IP映射关系。

步骤2 通过IntelliJ IDEA可直接运行Consumer.java，如图19-19所示：

图 19-19 运行 Consumer.java



步骤3 单击运行后弹出控制台窗口，可以看到Consumer启动成功后，再启动Producer，即可看到实时接收消息：

图 19-20 Consumer.java 运行窗口

```
[2019-06-23 17:49:48,609] INFO [Consumer clientId=consumer-1, groupId=DemoConsumer] (Re-)joining group (org.apache.kafka.clients.consumer.internals.AbstractCoordinator)
[2019-06-23 17:49:51,865] INFO [Consumer clientId=consumer-1, groupId=DemoConsumer] Successfully joined group with generation 5 (org.apache.kafka.clients.consumer.inter
[2019-06-23 17:49:51,866] INFO [Consumer clientId=consumer-1, groupId=DemoConsumer] Setting newly assigned partitions [example-metric1-0, example-metric1-1] (org.apache.
[2019-06-23 17:49:56,670] INFO [NewConsumerExample], Received message: (1, Message_1) at offset 188 (com.huawei.bigdata.kafka.example.NewConsumer)
[2019-06-23 17:49:56,670] INFO [NewConsumerExample], Received message: (5, Message_5) at offset 189 (com.huawei.bigdata.kafka.example.NewConsumer)
[2019-06-23 17:49:56,670] INFO [NewConsumerExample], Received message: (8, Message_8) at offset 190 (com.huawei.bigdata.kafka.example.NewConsumer)
[2019-06-23 17:49:56,670] INFO [NewConsumerExample], Received message: (9, Message_9) at offset 191 (com.huawei.bigdata.kafka.example.NewConsumer)
[2019-06-23 17:49:56,670] INFO [NewConsumerExample], Received message: (15, Message_15) at offset 192 (com.huawei.bigdata.kafka.example.NewConsumer)
[2019-06-23 17:49:56,670] INFO [NewConsumerExample], Received message: (18, Message 18) at offset 193 (com.huawei.bigdata.kafka.example.NewConsumer)
```

----结束

在 Linux 调测程序

步骤1 编译并生成Jar包，并将Jar包复制到与依赖库文件夹同级的目录“src/main/resources”下，具体步骤请参考**在Linux调测程序**。

步骤2 运行Consumer样例工程的命令如下。

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.Consumer

----结束

19.6.3 调测 Kafka High level Streams 样例程序

在 Windows 中调测程序

在Windows环境调测程序步骤请参考**在Windows中调测程序**。

在 Linux 环境调测程序

步骤1 编译并生成Jar包，并将Jar包复制到与依赖库文件夹同级的目录“src/main/resources”下，具体步骤请参考**在Linux调测程序**。

步骤2 使用集群安装用户登录集群客户端节点。

cd /opt/client

source bigdata_env

kinit *组件操作用户（例如developuser）*

步骤3 创建输入Topic和输出Topic，与样例代码中指定的Topic名称保持一致，输出Topic的清理策略设置为compact。

kafka-topics.sh --create --zookeeper *quorumpeer实例IP地址:ZooKeeper客户端连接端口*/**kafka --replication-factor 1 --partitions 1 --topic** *Topic名称*

quorumpeer实例IP地址可登录集群的FusionInsight Manager界面，在“集群 > 服务 > ZooKeeper > 实例”界面中查询，多个地址可用“,”分隔。ZooKeeper客户端连接端口可通过ZooKeeper服务配置参数“clientPort”查询，例如端口号为2181。

例如执行以下命令：

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-input

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-output --config cleanup.policy=compact

步骤4 Topic创建成功后，执行以下命令运行程序。

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.WordCountDemo

步骤5 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-producer.sh”向输入Topic中写入消息：

cd /opt/client

source bigdata_env

kinit *组件操作用户（例如developuser）*

kafka-console-producer.sh --broker-list *Broker实例IP地址:Kafka连接端口* **--topic streams-wordcount-input --producer.config /opt/client/Kafka/kafka/config/producer.properties**

说明

- *Broker实例IP地址*：登录FusionInsight Manager，选择“集群 > 服务 > Kafka > 实例”，在实例列表页面中查看并记录任意一个Broker实例业务IP地址。
- *Kafka连接端口*：集群已启用Kerberos认证（安全模式）时Broker端口为“sasl.port”参数的值。集群未启用Kerberos认证（普通模式）时Broker端口为“port”的值。

步骤6 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-consumer.sh”从输出Topic消费数据，查看统计结果。

cd /opt/client

source bigdata_env

kinit *组件操作用户（例如developuser）*

kafka-console-consumer.sh --topic streams-wordcount-output --bootstrap-server *Broker实例IP地址:Kafka连接端口* **--consumer.config /opt/client/Kafka/kafka/config/consumer.properties --from-beginning --property print.key=true --property print.value=true --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer --formatter kafka.tools.DefaultMessageFormatter**

向输入Topic中写入消息：

```
>This is Kafka Streams test
>test starting
>now Kafka Streams is running
>test end
```

消息输出：

```
this    1
is      1
kafka   1
streams 1
test    1
test    2
starting 1
now     1
kafka   2
streams 2
is      2
running 1
test    3
end     1
```

----结束

19.6.4 调测 Kafka Low level Streams 样例程序

在 Windows 中调测程序

在Windows环境调测程序步骤请参考**在Windows中调测程序**。

在 Linux 环境调测程序

步骤1 编译并生成Jar包，并将Jar包复制到与依赖库文件夹同级的目录“src/main/resources”下，具体步骤请参考**在Linux调测程序**。

步骤2 使用**root**用户登录安装了集群客户端的节点。

cd /opt/client

source bigdata_env

kinit *组件操作用户（例如developuser）*

步骤3 创建输入Topic和输出Topic，与样例代码中指定的Topic名称保持一致，输出Topic的清理策略设置为compact。

kafka-topics.sh --create --zookeeper *quorumpeer实例IP地址:ZooKeeper客户端连接端口*/**kafka --replication-factor 1 --partitions 1 --topic** *Topic名称*

quorumpeer实例IP地址可登录集群的FusionInsight Manager界面，在“集群 > 服务 > ZooKeeper > 实例”界面中查询，多个地址可用“,”分隔。ZooKeeper客户端连接端口可通过ZooKeeper服务配置参数“clientPort”查询，例如端口号为2181。

例如执行以下命令：

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-processor-input

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-processor-output --config cleanup.policy=compact

步骤4 Topic创建成功后，执行以下命令运行程序。

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.WordCountDemo

步骤5 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-producer.sh”向输入Topic中写入消息：

cd /opt/client

source bigdata_env

kinit *组件操作用户（例如developuser）*

kafka-console-producer.sh --broker-list *Broker实例IP地址:Kafka连接端口* **--topic streams-wordcount-processor-input --producer.config /opt/client/Kafka/kafka/config/producer.properties**

说明

- *Broker实例IP地址*：登录FusionInsight Manager，选择“集群 > 服务 > Kafka > 实例”，在实例列表页面中查看并记录任意一个Broker实例业务IP地址。
- *Kafka连接端口*：集群已启用Kerberos认证（安全模式）时Broker端口为“sasl.port”参数的值。集群未启用Kerberos认证（普通模式）时Broker端口为“port”的值。

步骤6 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-consumer.sh”从输出Topic消费数据，查看统计结果。

cd /opt/client

source bigdata_env

kinit *组件操作用户（例如developuser）*

kafka-console-consumer.sh --topic streams-wordcount-processor-output --bootstrap-server *Broker实例IP地址:Kafka连接端口* **--consumer.config /opt/client/Kafka/kafka/config/consumer.properties --from-beginning --property print.key=true --property print.value=true**

向输入Topic中写入消息：

```
>This is Kafka Streams test
>test starting
>now Kafka Streams is running
>test end
```


消息输出：

```
this    1
is      1
kafka   1
streams 1
test    1
test    2
starting 1
now     1
kafka   2
streams 2
is      2
running 1
test    3
end     1
```

----结束

19.6.5 调测 Kafka Token 认证机制样例程序

步骤1 Kafka服务端配置Kafka Token认证。

1. 登录FusionInsight Manager管理界面，选择“集群 > 服务 > Kafka > 配置”，打开Kafka服务配置页面。
2. 开启Token认证机制。

   查找配置项“delegation.token.master.key”，该配置指定用于生成和验证Token的主密钥。先查看是否已经配置，如果已配置且不为null，则表示Token认证机制是开启的，不用重新配置（重新配置会导致之前生产的Token无法使用）。

   说明

   参数“delegation.token.master.key”的值为自定义配置，例如配置参数值为“Tokentest”。
3. 指定服务使用的SASL认证机制。

   查找配置项“sasl.enabled.mechanisms”，配置为“GSSAPI,SCRAM-SHA-256,SCRAM-SHA-512”（使用英文逗号将这三项分隔）。
4. 使用Scram登录组件。

   查找自定义配置项“kafka.config.expandor”，配置名称为**listener.name.sasl_plaintext.scram-sha-512.sasl.jaas.config**，值为“org.apache.kafka.common.security.scram.ScramLoginModule required;”。
5. 登录FusionInsight Manager管理界面，重启Kafka服务所有Broker实例。

步骤2 Kafka客户端配置Kafka Token认证。

参考**Kafka Token认证机制工具使用说明**为用户生成Token。

步骤3 二次开发样例工程配置。

在二次开发样例的**Producer()**和**Consumer()**中对其进行配置。配置的样例代码可参考**使用Kafka Token认证**。

步骤4 运行样例代码。

- 在Windows环境调测程序请参考**在Windows中调测程序**。
- 在Linux环境调测程序请参考**在Linux调测程序**。

----结束

19.7 Kafka 应用开发常见问题

19.7.1 Kafka 常用 API 介绍

19.7.1.1 Kafka Shell 命令介绍

- 查看当前集群Topic列表。

  shkafka-topics.sh --list --zookeeper <*ZooKeeper集群IP:2181***/kafka>**

  shkafka-topics.sh --list --bootstrap-server *<Kafka集群IP:21007>* **--command-config config/client.properties**

- 查看单个Topic详细信息。

  shkafka-topics.sh --describe --zookeeper <*ZooKeeper集群IP:2181***/kafka> --topic** *<Topic名称>*

  shkafka-topics.sh --describe --bootstrap-server *<Kafka集群IP:21007>* **--command-config config/client.properties --topic** *<Topic名称>*

- 删除Topic，由管理员用户操作。

  shkafka-topics.sh --delete --zookeeper <*ZooKeeper集群IP:2181***/kafka> --topic** *<Topic名称>*

  sh**kafka-topics.sh --delete --bootstrap-server** *<Kafka集群IP:21007>***--command-config config/client.properties --topic** *<Topic名称>*

- 创建Topic，由管理员用户操作。

  shkafka-topics.sh --create --zookeeper <*ZooKeeper集群IP:2181***/kafka> --partitions 6 --replication-factor 2 --topic** *<Topic名称>*

  shkafka-topics.sh --create --bootstrap-server *<Kafka集群IP:21007>* **--command-config config/client.properties --partitions 6 --replication-factor 2 --topic** *<Topic名称>*

- 赋Consumer权限命令，由管理员用户操作。

  shkafka-acls.sh --authorizer-properties zookeeper.connect=<ZooKeeper集群IP:2181**/kafka > --add --allow-principal User:***<用户名>* **--consumer --topic** *<Topic名称>* **--group** *<消费者组名称>*

  shkafka-acls.sh --bootstrap-server <*Kafka集群IP***:21007> --command-config config/client.properties --add --allow-principal User:***<用户名>* **--consumer --topic** *<Topic名称>* **--group** *<消费者组名称>*

- 赋Producer权限命令，由管理员用户操作。

  shkafka-acls.sh --authorizer-properties zookeeper.connect=<ZooKeeper集群IP:2181**/kafka > --add --allow-principal User:***<用户名>* **--producer --topic** *<Topic名称>*

  shkafka-acls.sh --bootstrap-server <*Kafka集群IP***:21007> --command-config config/client.properties --add --allow-principal User:***<用户名>* **--producer --topic <***Topic名称***>**

- 生产消息，需要拥有该Topic生产者权限。

  shkafka-console-producer.sh --broker-list <*Kafka集群IP***:21007> --topic** *<Topic名称>* **--producer.config config/producer.properties**

- 消费数据，需要拥有该Topic的消费者权限。

  shkafka-console-consumer.sh --topic *<Topic名称>* **--bootstrap-server** *<Kafka集群IP:21007>* **--consumer.config config/consumer.properties**

  说明

  - Shell命令需要在目录“*客户端安装目录*/Kafka/kafka/bin”下执行。
  - 凡可指定“ * ”值以代表all value，且格式为“*--参数 参数值*”，例如： --group *，必须通过单引号或者等于号赋值，例如：--group '*' 或者 --group=*。

19.7.1.2 Kafka Java API 接口介绍

Kafka相关接口同开源社区保持一致，详情请参见**https://kafka.apache.org/24/documentation.html**。

Producer 重要接口

表 19-5 Producer 重要参数

参数	描述	备注
bootstrap.servers	Broker地址列表。	生产者通过此参数值，创建与Broker之间的连接。
security.protocol	安全协议类型。	生产者使用的安全协议类型，当前安全模式下仅支持SASL协议，需要配置为SASL_PLAINTEXT。
sasl.kerberos.service.name	服务名。	Kafka集群运行，所使用的Kerberos用户名（需配置为kafka）。
key.serializer	消息Key值序列化类。	指定消息Key值序列化方式。
value.serializer	消息序列化类。	指定所发送消息的序列化方式。

表 19-6 Producer 重要接口函数

返回值类型	接口函数	描述
java.util.concurrent.Future<RecordMetadata>	send(ProducerRecord<K, V> record)	不带回调函数的发送接口，通常使用Future的get()函数阻塞发送，实现同步发送。
java.util.concurrent.Future<RecordMetadata>	send(ProducerRecord<K, V> record, Callback callback)	带回调函数的发送接口，通常用于异步发送后，通过回调函数实现对发送结果的处理。

返回值类型	接口函数	描述
void	onCompletion(RecordMetadata metadata, Exception exception);	回调函数接口方法，通过实现Callback中的此方法来进行异步发送结果的处理。

Consumer 重要接口

表 19-7 Consumer 重要参数

参数	描述	备注
bootstrap.servers	Broker地址列表。	消费者通过此参数值，创建与Broker之间的连接。
security.protocol	安全协议类型。	消费者使用的安全协议类型，当前安全模式下仅支持SASL协议，需要配置为SASL_PLAINTEXT。
sasl.kerberos.service.name	服务名。	Kafka集群运行，所使用的Kerberos用户名（需配置为kafka）。
key.deserializer	消息Key值反序列化类。	反序列化消息Key值。
value.deserializer	消息反序列化类。	反序列化所接收的消息。

表 19-8 Consumer 重要接口函数

返回值类型	接口函数	描述
void	close()	关闭Consumer接口方法。
void	subscribe(java.util.Collection<java.lang.String> topics)	Topic订阅接口方法。
ConsumerRecords<K,V>	poll(final Duration timeout)	请求获取消息接口方法。

19.7.2 使用 Kafka 客户端 SSL 加密

前提说明

- 客户端使用SSL功能前，必须要保证服务端SSL对应服务功能已经开启（服务端参数“ssl.mode.enable”设置为“true”）。

- SSL功能需要配合API进行使用，可参考**Kafka安全使用说明**章节。

使用说明

- **Linux客户端使用SSL功能**
  a. 修改“客户端安装目录/Kafka/kafka/config/producer.properties”和“客户端安装目录/Kafka/kafka/config/consumer.properties”中“security.protocol”的值为“SASL_SSL”或者“SSL”。
  b. 进入“*客户端安装目录*/Kafka/kafka/bin”使用shell命令时，根据上一步中配置的协议填写对应的端口，例如使用配置的“security.protocol”为“SASL_SSL”，则需要填写SASL_SSL协议端口，默认为21009：

     shkafka-console-producer.sh --broker-list *<Kafka集群IP:21009>* **--topic** *<Topic名称>* **--producer.config config/producer.properties**

     shkafka-console-consumer.sh --topic *<Topic名称>* **--bootstrap-server** *<Kafka集群IP:21009>* **--consumer.config config/consumer.properties**
- **Windows客户端代码使用SSL功能**
  a. 下载Kafka客户端，解压后在根目录中找到ca.crt证书文件。
  b. 使用ca.crt证书生成客户端的truststore。

     在安装了Java的环境下执行命令：

     keytool -noprompt -import -alias myservercert -file ca.crt -keystore truststore.jks。
  c. 将生成的truststore.jks复制至IntelliJ IDEA工程的conf目录下，并在客户端代码中（Producer.java或者Consumer.java的构造方法）添加如下代码：

```
//truststore文件地址
props.put("ssl.truststore.location", System.getProperty("user.dir") + File.separator + "conf" + File.separator + "truststore.jks");
//truststore文件密码（生成时输入的密码）
props.put("ssl.truststore.password", "XXXXX");
```

  d. 按需修改客户端样例工程的“src/main/resources”目录下的“producer.properties”和“consumer.properties”中的“security.protocol”的值，同时修改“producer.properties”中的“bootstrap.servers”的值，确保security.protocol协议类型和bootstrap.servers中的端口号匹配。

19.7.3 配置 Windows 通过 EIP 访问安全模式集群 Kafka

操作场景

该章节通过指导用户配置集群绑定EIP，并配置Kafka文件的方法，方便用户可以在本地对样例文件进行编译。

操作步骤

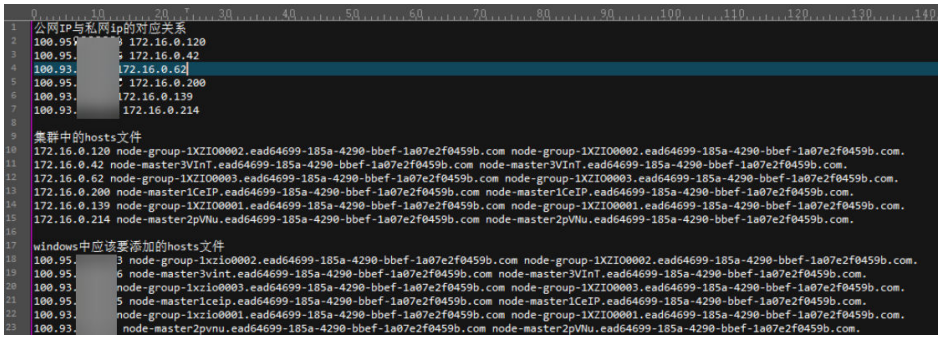
步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。

图 19-21 配置 hosts 文件



步骤2 将krb5.conf文件中的IP地址修改为对应IP的主机名称。

步骤3 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

图 19-22 管理安全组规则

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和21007、21730TCP、21731TCP/UDP、21732TCP/UDP端口。

图 19-23 添加入方向规则

步骤4 在Manager界面选择“集群 > 服务 > Kafka > 配置 > 全部配置”，搜索“kafka.config.expandor”，并添加参数名为“advertised.listeners”，值为“PLAINTEXT://:9092,SSL://:9093,TRACE://:21013”，修改配置保存后重启Kafka集群。

说明

如果当前集群为MRS 3.2.0-LTS.1，执行该步骤无法通过EIP访问Kafka时，可以参考如下操作进行处理：

1. 登录到FusionInsight Manager页面，选择“集群 > 服务 > Kafka > 实例”，勾选所有Broker实例，选择“更多 > 停止实例”验证管理员密码后停止所有Broker实例。 (该操作对业务有影响，请在业务低峰期操作)
2. 以**root**用户**分别**登录到Broker节点后台，修改“server.properties”配置。

   vi ${BIGDATA_HOME}/FusionInsight_HD_*/*_*_Broker/etc/server.properties

   将“host.name”修改为当前登录Broker节点的主机名称，将“listeners”及“advertised.listeners”参数值修改为“EXTERNAL_PLAINTEXT://*{主机名称}:{port}*”。
3. 登录到FusionInsight Manager页面，选择“集群 > 服务 > Kafka > 实例”，勾选所有Broker实例，单击“启动实例”。
4. 给MRS服务集群的Broker节点分别绑定EIP。
5. 在Windows通过已配置的Broker节点EIP地址和端口连接到Kafka集群并调试代码。

步骤5 运行样例代码前，修改样例代码中Kafka的连接串为hostname1:21007, hostname2:21007, hostname3:21007；修改代码中的域名；修改“用户自己申请的机机账号名称、keytab文件名称”。

说明

用户可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。

----结束

19.7.4 运行样例时提示 Topic 鉴权失败“TOPIC_AUTHORIZATION_FAILED”

解决步骤

步骤1 向管理员申请该Topic的访问权限。

步骤2 如果拥有权限后还是无法访问该Topic，使用管理员用户登录FusionInsight Manager，选择“集群 > 服务 > Kafka > 配置 > 全部配置”进入Kafka服务配置页面，搜索“allow.everyone.if.no.acl.found”配置项，将该值修改为“true”后，重新运行程序。

----结束

19.7.5 运行 Producer.java 样例报错“ERROR fetching topic metadata...”

解决步骤

步骤1 检查工程conf目录下“producer.properties”中配置的“bootstrap.servers”配置值中访问的IP和端口是否正确：

- 如果IP与Kafka集群部署的业务IP不一致，那么需要修改为当前集群正确的IP地址。
- 如果配置中的端口为21007（Kafka安全模式端口），那么修改该端口为9092（Kafka普通模式端口）。

步骤2 检查网络是否正常，确保当前机器能够正常访问Kafka集群。

----结束

20 Kafka 开发指南（普通模式）

20.1 Kafka 应用开发简介

Kafka 简介

Kafka是一个分布式的消息发布-订阅系统。它采用独特的设计提供了类似JMS的特性，主要用于处理活跃的流式数据。

Kafka有很多适用的场景：消息队列、行为跟踪、运维数据监控、日志收集、流处理、事件溯源、持久化日志等。

Kafka有如下几个特点：

- 高吞吐量
- 消息持久化到磁盘
- 分布式系统易扩展
- 容错性好
- 支持online和offline场景

接口类型简介

Kafka主要提供了的API主要可分Producer API和Consumer API两大类，均提供有Java API，使用的具体接口说明请参考**Kafka Java API介绍**。

常用概念

- **Topic**

  Kafka维护的同一类的消息称为一个Topic。

- **Partition**

  每一个Topic可以被分为多个Partition，每个Partition对应一个可持续追加的、有序不可变的log文件。

- **Producer**

  将消息发往Kafka topic中的角色称为Producer。

- **Consumer**
  从Kafka Topic中获取消息的角色称为Consumer。
- **Broker**
  Kafka集群中的每一个节点服务器称为Broker。

20.2 Kafka 应用开发流程介绍

Kafka客户端角色包括Producer和Consumer两个角色，其应用开发流程是相同的。

开发流程中各个阶段的说明如图20-1和表20-1所示。

图 20-1 Kafka 客户端程序开发流程

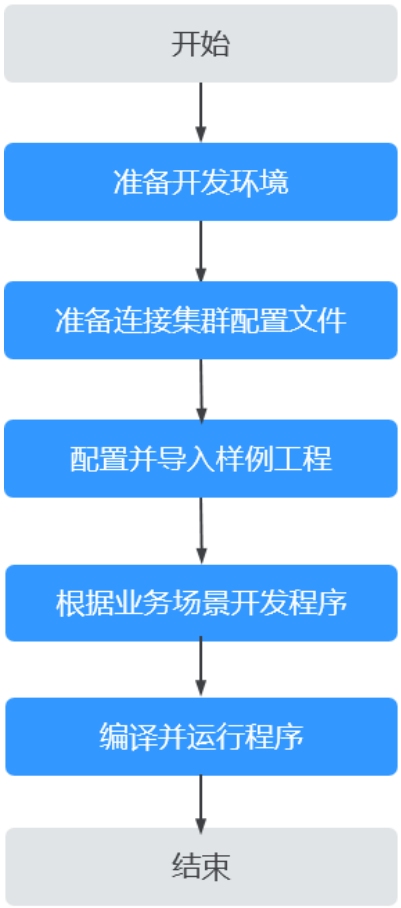


表 20-1 Kafka 客户端开发的流程说明

阶段	说明	参考文档
准备开发和运行环境	Kafka的客户端程序当前推荐使用java语言进行开发，可使用IntelliJ IDEA工具开发。 Kafka的运行环境即Kafka客户端，请根据指导完成客户端的安装和配置。	**准备本地应用开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备连接Kafka集群配置文件**
配置并导入样例工程	Kafka提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。	**导入并配置Kafka样例工程**
根据业务场景开发工程	提供了Producer和Consumer相关API的使用样例，包含了API和多线程的使用场景，帮助用户快速熟悉Kafka接口。	**开发Kafka应用**
编译与运行程序	指导用户将开发好的程序编译并提交运行。	**调测Kafka应用**

20.3 Kafka 样例工程简介

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Kafka相关样例工程：

表 20-2 Kafka 相关样例工程

样例工程位置	描述
kafka-examples	1. 单线程生产数据，相关样例请参考**使用Producer API向安全Topic生产消息**。 2. 单线程消费数据，相关样例请参考**使用Consumer API订阅安全Topic并消费**。 3. 多线程生产数据，相关样例请参考**使用多线程Producer发送消息**。 4. 多线程消费数据，相关样例请参考**使用Consumer API订阅安全Topic并消费**。 5. 基于KafkaStreams实现WordCount，相关样例请参考**使用多线程Consumer消费消息**

20.4 准备 Kafka 应用开发环境

20.4.1 准备本地应用开发环境

Kafka开发应用时，需要准备的开发和运行环境如**表20-3**所示：

表 20-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装和配置IntelliJ IDEA	开发环境的基本配置。版本要求：JDK使用1.8版本，IntelliJ IDEA使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。

准备项	说明
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 ● X86客户端： – Oracle JDK：支持1.8版本 – IBM JDK：支持1.8.5.11版本 ● TaiShan客户端： OpenJDK：支持1.8.0_272版本 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件,支持7-Zip 16.04版本。

20.4.2 准备连接 Kafka 集群配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，用于验证应用程序运行。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。
  a. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。
     例如，客户端配置文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。
  b. 进入客户端配置文件解压路径的“Kafka\config”，获取Kafka**表20-4**中相关配置文件。

表 20-4 配置文件

配置文件	作用
client.properties	Kafka的客户端的配置信息。
consumer.properties	Kafka的consumer端配置信息。
producer.properties	Kafka的producer端配置信息。
server.properties	Kafka的服务端的配置信息。

c. 复制解压目录下的“hosts”文件中的内容到本地hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群，具体操作请参考**配置Windows通过EIP访问普通模式集群Kafka**。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。

  a. 在节点中安装MRS集群客户端。

  例如客户端安装目录为“/opt/client”。

  b. 获取配置文件：

  i. 登录FusionInsight Manager，选择“集群 > 概览 > 更多 > 下载客户端”，，“选择客户端类型”设置为“仅配置文件”，勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。

  ii. 以**root**登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“Kafka/config”路径下的**表20-4**中相关配置文件。

  例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

  cd /tmp/FusionInsight-Client

  tar -xvf FusionInsight_Cluster_1_Services_Client.tar

  tar -xvf FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar

  cd FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles

  scp Kafka/install_files/kafka/libs/* root@*客户端节点IP地址*:/opt/client/lib

  c. 检查客户端节点网络连接。

  在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

20.4.3 导入并配置 Kafka 样例工程

步骤1 获取样例工程文件夹。

参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程文件夹“kafka-examples”。

步骤2 获取配置文件。

若需要在本地Windows调测Kafka样例代码，将**准备连接Kafka集群配置文件**时获取的所有配置文件放置在样例工程的“kafka-examples\src\main\resources”目录下。

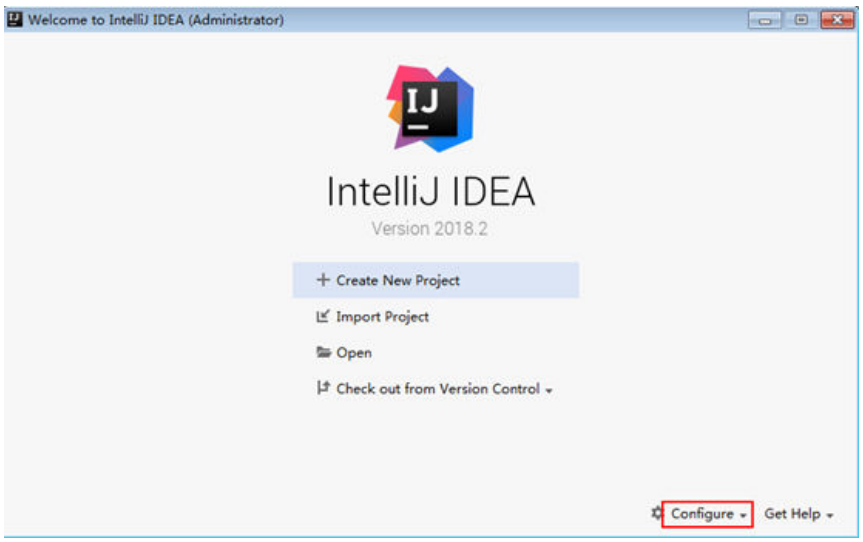
步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

说明

不同的IDEA版本的操作步骤可能存在差异，以实际版本的界面操作为准。

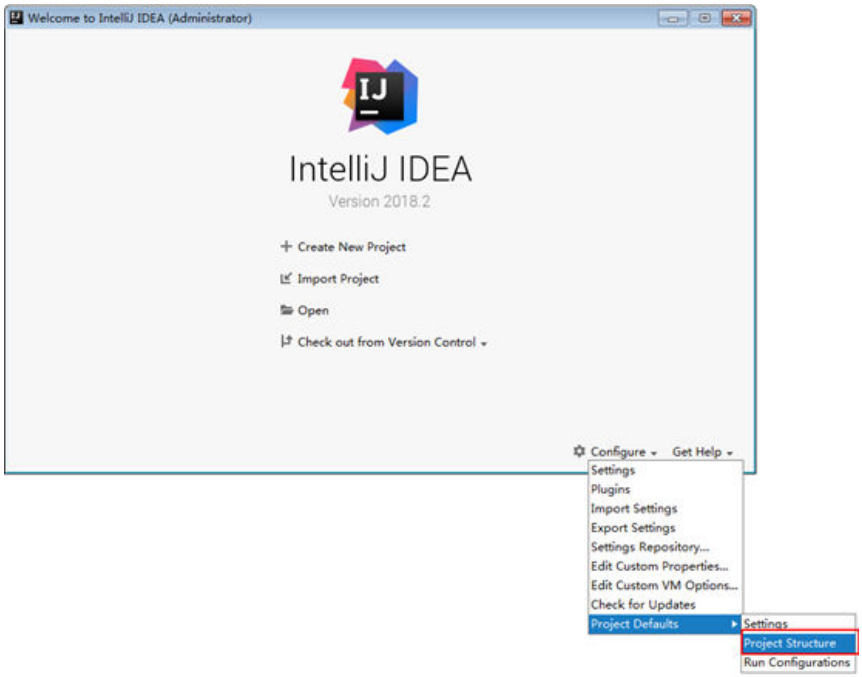
1. 打开IntelliJ IDEA，选择“Configure”。

图 20-2 Quick Start



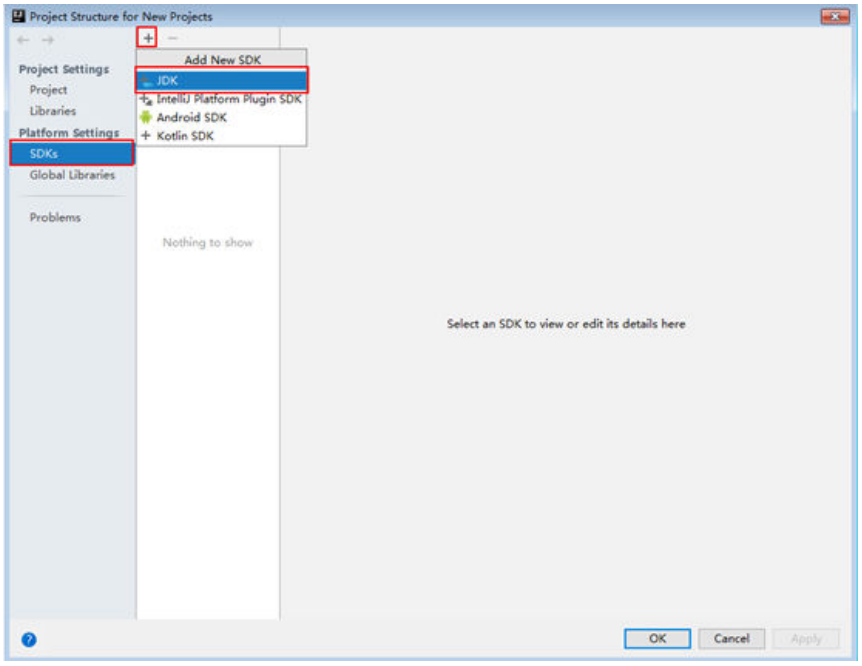
2. 在下拉框中选择“Project Defaults > Project Structure”。

图 20-3 Configure



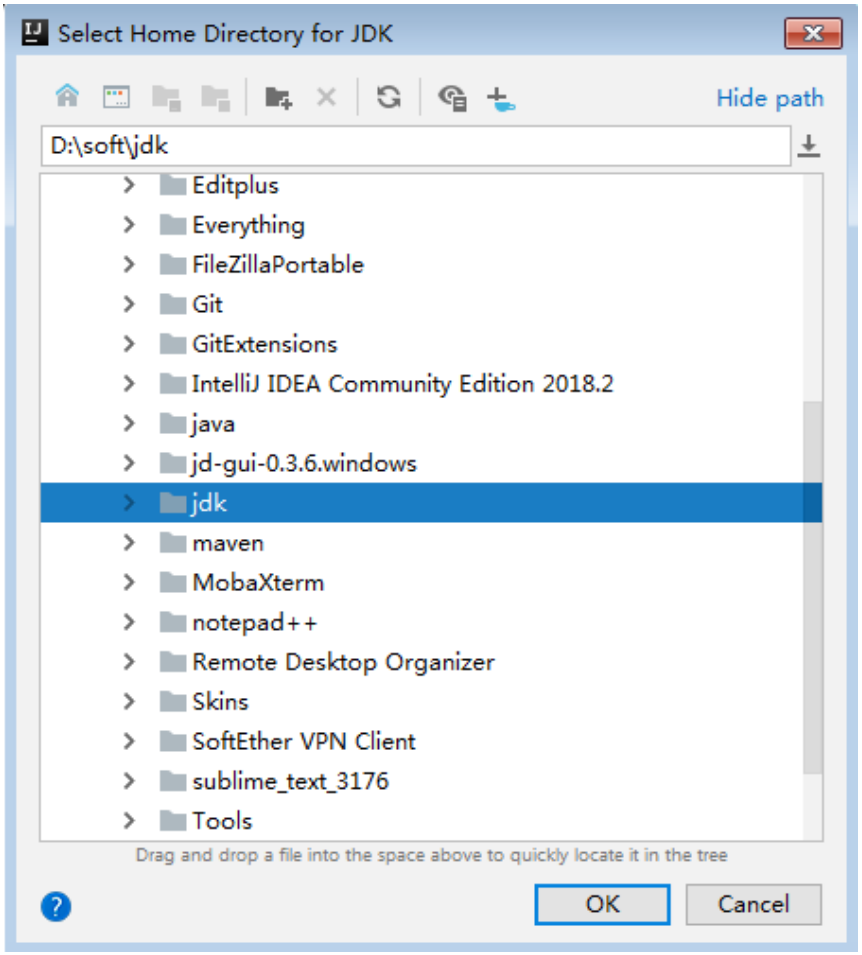
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 20-4 Project Structure for New Projects



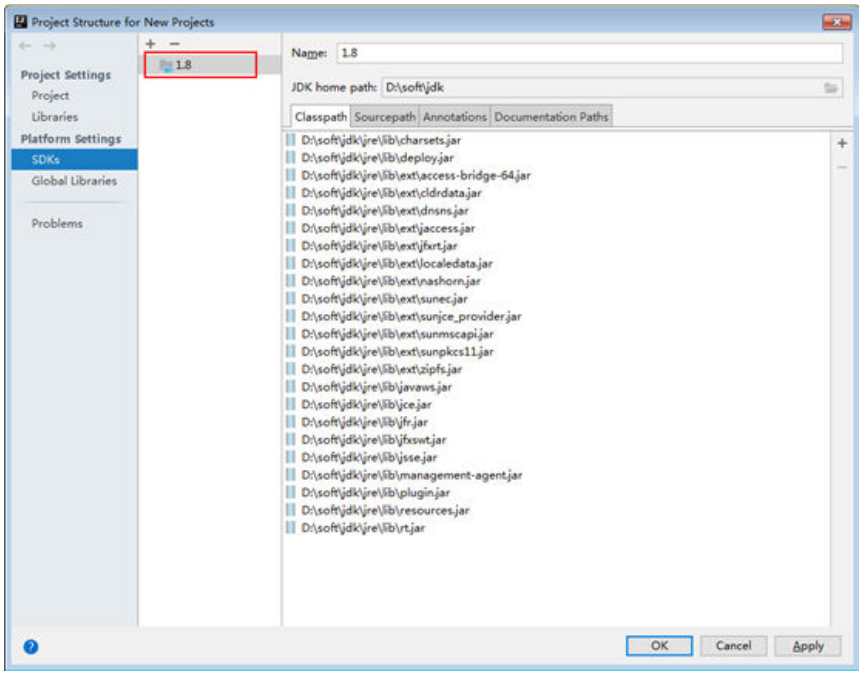
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 20-5 Select Home Directory for JDK

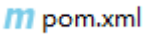


5. 完成JDK选择后，单击“OK”完成配置。

图 20-6 完成 JDK 配置



步骤4 将样例工程导入到IntelliJ IDEA开发环境。

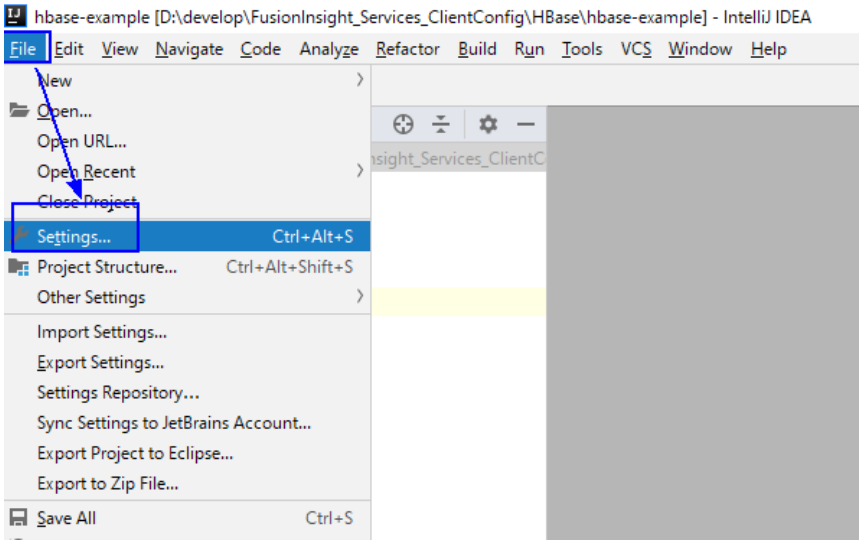
1. 选择“Open”。
   显示“浏览文件夹”对话框。
2. 选择样例工程文件夹，单击“OK”。
3. 导入结束，IDEA主页显示导入的样例工程。
4. 右键单击“pom.xml”，选择“Add as Maven Project”，将该项目添加为Maven Project。若"pom.xml"图标如 

所示，可直接进行下一步骤操作。

步骤5 设置项目使用的Maven版本。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings...”。

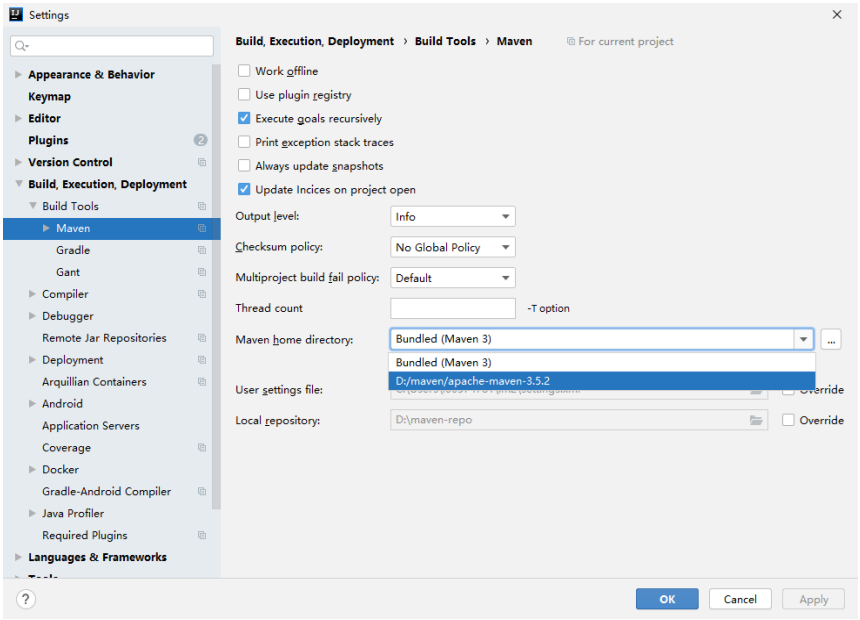
图 20-7 Settings



2. 选择“Build,Execution,Deployment > Maven”，选择“Maven home directory”为本地安装的Maven版本。

   然后根据实际情况设置好“User settings file”和“Local repository”参数，依次单击“Apply”、“OK”。

图 20-8 选择本地 Maven 安装目录



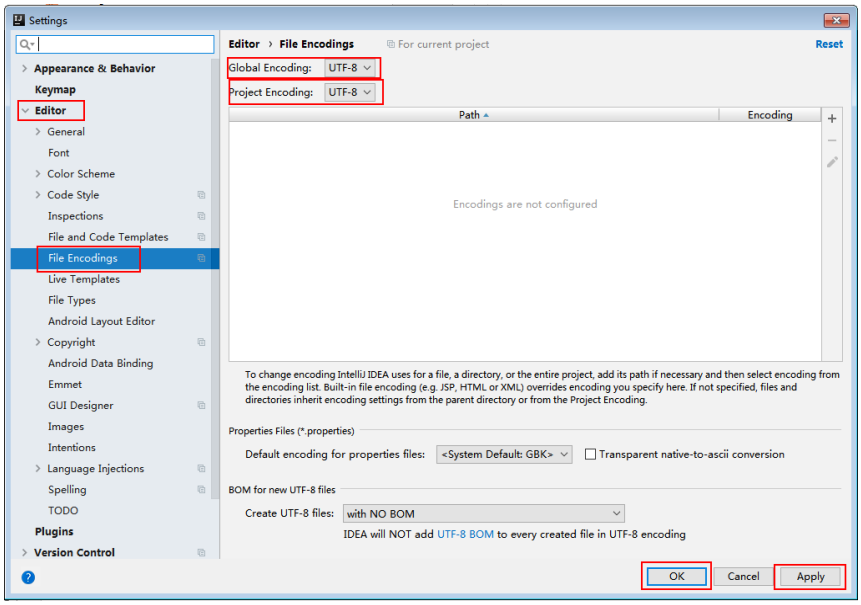
步骤6 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”

   弹出“Settings”窗口。

2. 在左边导航上选择“Editor > File Encodings”，在“Project Encoding”和“Global Encoding”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图20-9所示。

图 20-9 设置 IntelliJ IDEA 的编码格式



----结束

20.5 开发 Kafka 应用

20.5.1 Kafka 样例程序开发思路

场景说明

Kafka是一个分布式消息系统，在此系统上用户可以做一些消息的发布和订阅操作，假定用户要开发一个Producer，让其每秒向Kafka集群某Topic发送一条消息，另外还需要实现一个Consumer，订阅该Topic，实时消费该类消息。

开发思路

1. 使用Linux客户端创建一个Topic。
2. 开发一个Producer向该Topic生产数据。
3. 开发一个Consumer消费该Topic的数据。

性能调优建议

1. 建议预先创建Topic，根据业务需求合理规划Partition数目，Partition数目限制了消费者的并发数。
2. 消息key值选取一定是可变的，防止由于消息key值不变导致消息分布不均匀。
3. 消费者尽量使用主动提交offset的方式，避免重复消费。

20.5.2 使用 Producer API 向安全 Topic 生产消息

功能简介

下面代码片段在**com.huawei.bigdata.kafka.example.Producer**类的run方法中，用于实现Producer API向安全Topic生产消息。

代码样例

```
/**
 * 生产者线程执行函数,循环发送消息。
 */
public void run() {
    LOG.info("New Producer: start.");
    int messageNo = 1;

    while (messageNo <= MESSAGE_NUM) {
        String messageStr = "Message_" + messageNo;
        long startTime = System.currentTimeMillis();

        // 构造消息记录
        ProducerRecord<Integer, String> record = new ProducerRecord<Integer, String>(topic, messageNo,
messageStr);

        if (isAsync) {
            // 异步发送
            producer.send(record, new DemoCallBack(startTime, messageNo, messageStr));
        } else {
            try {
                // 同步发送
                producer.send(record).get();
                long elapsedTime = System.currentTimeMillis() - startTime;
                LOG.info("message(" + messageNo + ", " + messageStr + ") sent to topic(" + topic + ") in " +
elapsedTime + " ms.");
            } catch (InterruptedException ie) {
                LOG.info("The InterruptedException occured : {}.", ie);
            } catch (ExecutionException ee) {
                LOG.info("The ExecutionException occured : {}.", ee);
            }
        }
        messageNo++;
    }
}
```

20.5.3 使用 Consumer API 订阅安全 Topic 并消费

功能简介

下面代码片段在**com.huawei.bigdata.kafka.example.Consumer**类中，用于实现使用Consumer API订阅安全Topic，并进行消息消费。

代码样例

```
/**
 * Consumer构造函数。
 * @param topic 订阅的Topic名称。
 */

public Consumer(String topic) {
    super("KafkaConsumerExample", false);
    // 初始化consumer启动所需的配置参数，详见代码。
    Properties props = initProperties();
    consumer = new KafkaConsumer<Integer, String>(props);
```

```
    this.topic = topic;
}

public void doWork() {
    // 订阅
    consumer.subscribe(Collections.singletonList(this.topic));
    // 消息消费请求
    ConsumerRecords<Integer, String> records = consumer.poll(waitTime);
    // 消息处理
    for (ConsumerRecord<Integer, String> record : records) {
        LOG.info("[ConsumerExample], Received message: (" + record.key() + ", " + record.value() + ") at
offset " + record.offset());
    }
}
```

20.5.4 使用多线程 Producer 发送消息

功能简介

在**使用Producer API向安全Topic生产消息**基础上，实现了多线程Producer，可启动多个Producer线程，并通过指定相同key值的方式，使每个线程对应向特定Partition发送消息。

下面代码片段在**com.huawei.bigdata.kafka.example.ProducerMultThread**类的run方法中，用于实现多线程生产数据。

代码样例

```
/**
 * 指定Key值为当前ThreadId，发送数据。
 */
public void run()
{
LOG.info("Producer: start.");

    // 用于记录消息条数。
int messageCount = 1;

    // 每个线程发送的消息条数。
int messagesPerThread = 5;
while (messageCount <= messagesPerThread)
    {

        // 待发送的消息内容。
        String messageStr = new String("Message_" + sendThreadId + "_" + messageCount);

        // 此处对于同一线程指定相同Key值，确保每个线程只向同一个Partition生产消息。
        String key = String.valueOf(sendThreadId);

        // 消息发送。
        producer.send(new KeyedMessage<String, String>(sendTopic, key, messageStr));
LOG.info("Producer: send " + messageStr + " to " + sendTopic + " with key: " + key);
        messageCount++;
     }
}
```

20.5.5 使用多线程 Consumer 消费消息

功能简介

在**使用Consumer API订阅安全Topic并消费**基础上，实现了多线程并发消费，可根据Topic的Partition数目启动相应个数的Consumer线程来对应消费每个Partition上的消息。

下面代码片段在**com.huawei.bigdata.kafka.example.ConsumerMultThread**类的run方法中，用于实现对指定Topic的并发消费。

代码样例

```
/**
 * 启动多线程并发消费Consumer。
 */
public void run() {
    LOG.info("Consumer: start.");
    Properties props = Consumer.initProperties();
    // 启动指定个数Consuemr线程来消费
    // 注意：当该参数大于待消费Topic的Partition个数时，多出的线程将无法消费到数据
    for (int threadNum = 0; threadNum < CONCURRENCY_THREAD_NUM; threadNum++) {
        new ConsumerThread(threadNum, topic, props).start();
        LOG.info("Consumer Thread " + threadNum + " Start.");
    }
}

private class ConsumerThread extends ShutdownableThread {
    private int threadNum = 0;
    private String topic;
    private Properties props;
    private KafkaConsumer<String, String> consumer = null;

    /**
     * 消费者线程类构造方法
     *
     * @param threadNum 线程号
     * @param topic     topic
     */
    public ConsumerThread(int threadNum, String topic, Properties props) {
        super("ConsumerThread" + threadNum, true);
        this.threadNum = threadNum;
        this.topic = topic;
        this.props = props;
        this.consumer = new KafkaConsumer<String, String>(props);
    }

    public void doWork() {
        consumer.subscribe(Collections.singleton(this.topic));
        ConsumerRecords<String, String> records = consumer.poll(waitTime);
        for (ConsumerRecord<String, String> record : records) {
            LOG.info("Consumer Thread-" + this.threadNum + " partitions:" + record.partition() + " record: "
                + record.value() + " offsets: " + record.offset());
        }
    }
}
```

20.5.6 使用 KafkaStreams 统计数据

功能简介

以下提供High level KafkaStreams API代码样例及Low level KafkaStreams API代码样例，通过Kafka Streams读取输入Topic中的消息，统计每条消息中的单词个数，从输出Topic消费数据，将统计结果以Key-Value的形式输出，完成单词统计功能。

High Level KafkaStreams API 代码样例

下面代码片段在**com.huawei.bigdata.kafka.example.WordCountDemo**类的createWordCountStream方法中。

```
static void createWordCountStream(final StreamsBuilder builder) {
    // 从 input-topic 接收输入记录
```

```
    final KStream<String, String> source = builder.stream(INPUT_TOPIC_NAME);

    // 聚合 key-value 键值对的计算结果
    final KTable<String, Long> counts = source
            // 处理接收的记录，根据正则表达式REGEX_STRING进行分割
            .flatMapValues(value ->
Arrays.asList(value.toLowerCase(Locale.getDefault()).split(REGEX_STRING)))
            // 聚合key-value键值对的计算结果
            .groupBy((key, value) -> value)
            // 最终结果计数
            .count();

    // 将计算结果的 key-value 键值对从 output topic 输出
    counts.toStream().to(OUTPUT_TOPIC_NAME, Produced.with(Serdes.String(), Serdes.Long()));
}
```

Low Level KafkaStreams API 代码样例

下面代码片段在**com.huawei.bigdata.kafka.example.WordCountProcessorDemo**类中。

```
private static class MyProcessorSupplier implements ProcessorSupplier<String, String> {
    @Override
    public Processor<String, String> get() {
        return new Processor<String, String>() {
            // ProcessorContext实例，它提供对当前正在处理的记录的元数据的访问
            private ProcessorContext context;
            private KeyValueStore<String, Integer> kvStore;

            @Override
            @SuppressWarnings("unchecked")
            public void init(ProcessorContext context) {
                // 在本地保留processor context，因为在punctuate()和commit()时会用到
                this.context = context;
                // 每1秒执行一次punctuate()
                this.context.schedule(Duration.ofSeconds(1), PunctuationType.STREAM_TIME, timestamp -> {
                    try (final KeyValueIterator<String, Integer> iter = kvStore.all()) {
                        System.out.println("----------- " + timestamp + " ----------- ");
                        while (iter.hasNext()) {
                            final KeyValue<String, Integer> entry = iter.next();
                            System.out.println("[" + entry.key + ", " + entry.value + "]");
                            // 将新纪录作为键值对发送到下游处理器
                            context.forward(entry.key, entry.value.toString());
                        }
                    }
                });
                // 检索名称为KEY_VALUE_STATE_STORE_NAME的key-value状态存储区，可用于记忆最近收到的输
入记录等
                this.kvStore = (KeyValueStore<String, Integer>)
context.getStateStore(KEY_VALUE_STATE_STORE_NAME);
            }

            // 对input topic的接收记录进行处理，将记录拆分为单词并计数
            @Override
            public void process(String dummy, String line) {
                String[] words = line.toLowerCase(Locale.getDefault()).split(REGEX_STRING);

                for (String word : words) {
                    Integer oldValue = this.kvStore.get(word);

                    if (oldValue == null) {
                        this.kvStore.put(word, 1);
                    } else {
                        this.kvStore.put(word, oldValue + 1);
                    }
                }
            }

            @Override
```

```
        public void close() {
        }
    };
  }
}
```

20.6 调测 Kafka 应用

20.6.1 调测 Kafka Producer 样例程序

前提条件

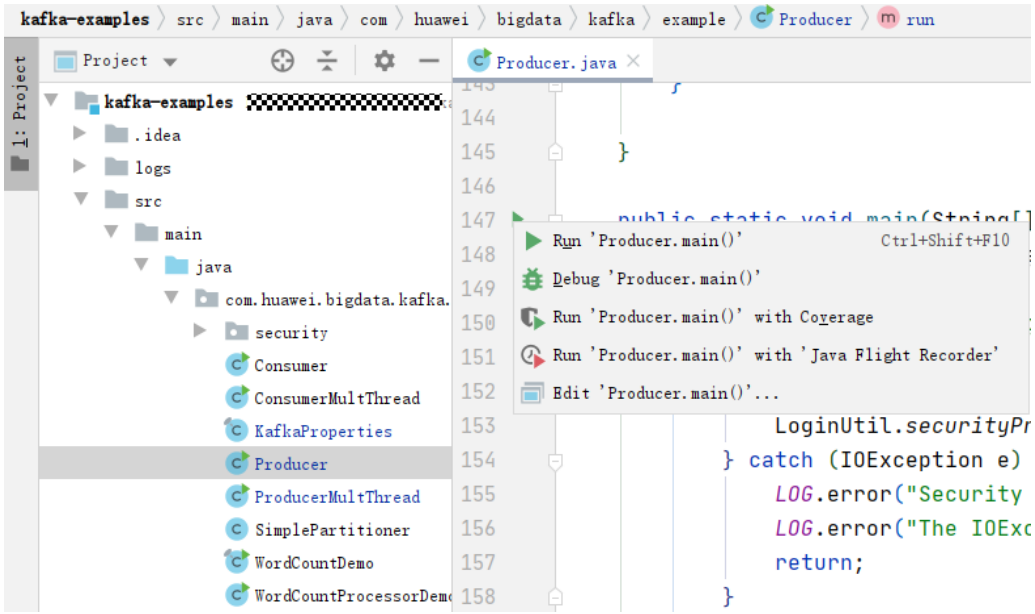
- 如需在Windows调测程序，需要配置Windows通过EIP访问集群Kafka，详情请参见**配置Windows通过EIP访问普通模式集群Kafka**。
- 如需在Linux调测程序，需要确保当前用户对“src/main/resources”目录下和依赖库文件目录下的所有文件，均具有可读权限。同时保证已安装JDK并已设置java相关环境变量。

在 Windows 中调测程序

步骤1 确保本地的hosts文件中配置了远程集群所有主机的主机名和业务IP映射关系。

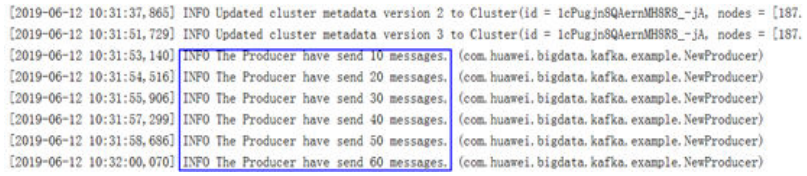
步骤2 通过IntelliJ IDEA可直接运行Producer.java，如**图20-10**所示：

图 20-10 运行 Producer.java



步骤3 运行后弹出控制台窗口，可以查看到Producer正在向默认Topic（example-metric1）发送消息，每发送10条，打印一条日志。

图 20-11 Producer 运行窗口



----结束

在 Linux 调测程序

步骤1 导出jar包。

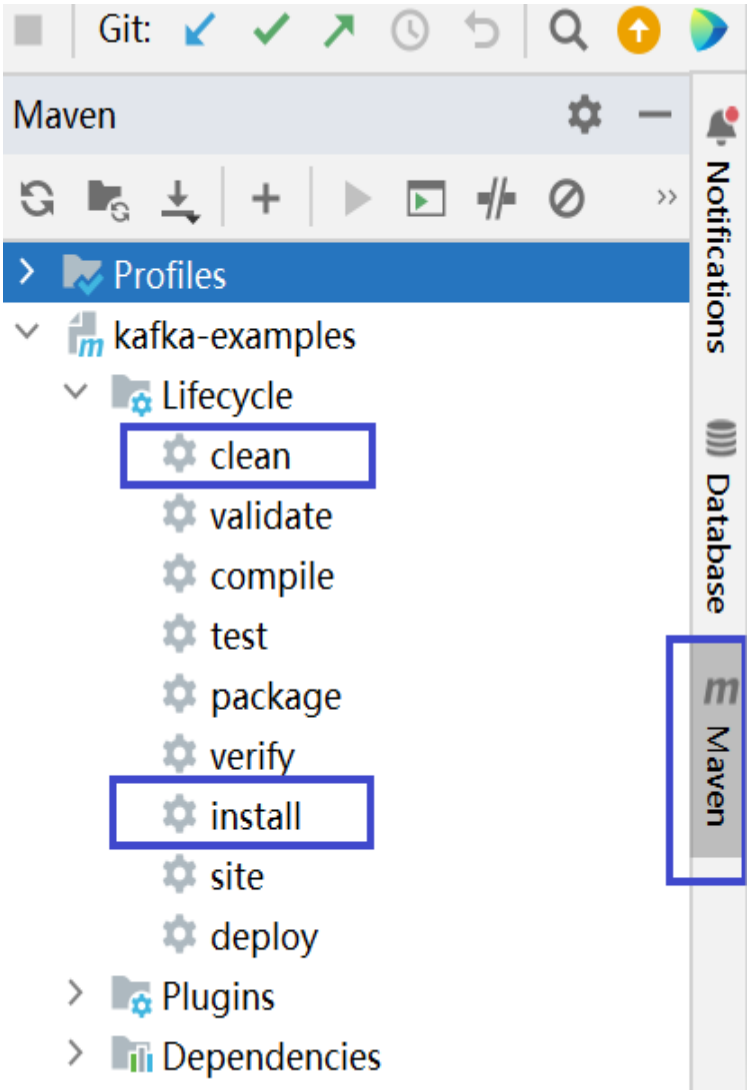
构建jar包方式有以下两种：

- 方法一：

  选择“Maven > *样例工程名称* > Lifecycle > clean”，双击“clean”运行maven的clean命令。

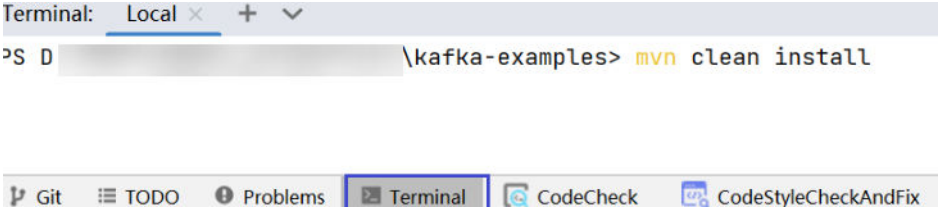
  选择“Maven > *样例工程名称* > Lifecycle > install”，双击“install”运行maven的install命令。

图 20-12 maven 工具 clean 和 install



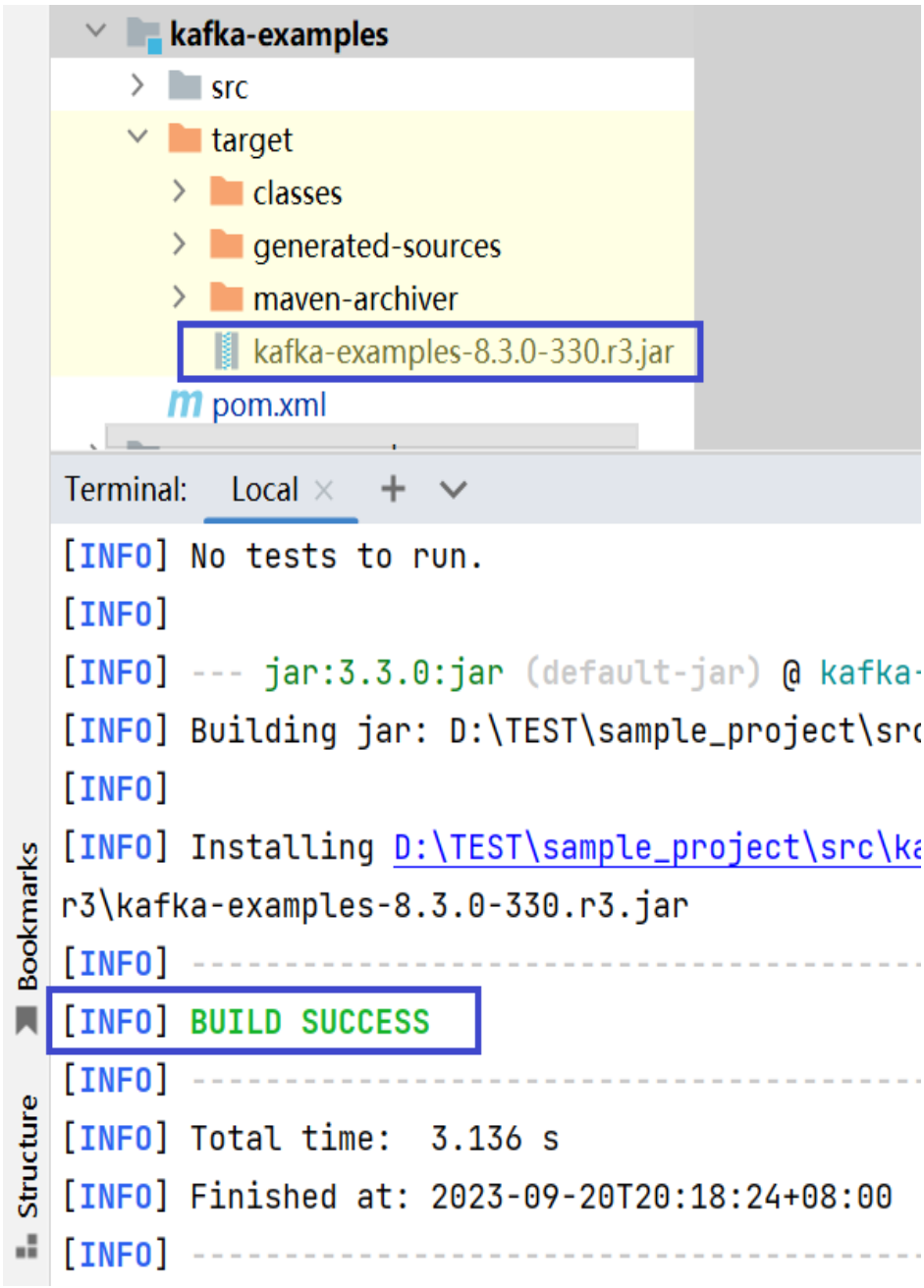
- 方法二：在IDEA的下方Terminal窗口进入“pom.xml”所在目录，手动输入**mvn clean install**命令进行编译

图 20-13 idea termial 输入“mvn clean install”



编译完成，打印“Build Success”，生成target目录，生成jar包在target目录中。

图 20-14 编译完成，生成 jar 包



步骤2 将工程编译生成的jar包复制到“/opt/client/lib”目录下。

步骤3 将IntelliJ IDEA工程“src/main/resources”目录下的所有文件复制到与依赖库文件夹同级的目录“src/main/resources”下，即“/opt/client/src/main/resources”。
“/opt/client”为客户端安装路径，具体以实际为准。

步骤4 进入目录“/opt/client”，首先确保“src/main/resources”目录下和依赖库文件目录下的所有文件，对当前用户均具有可读权限。同时保证已安装JDK并已设置java相关环境变量，然后执行命令运行样例工程，例如：

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.Producer

----结束

20.6.2 调测 Kafka Consumer 样例程序

前提条件

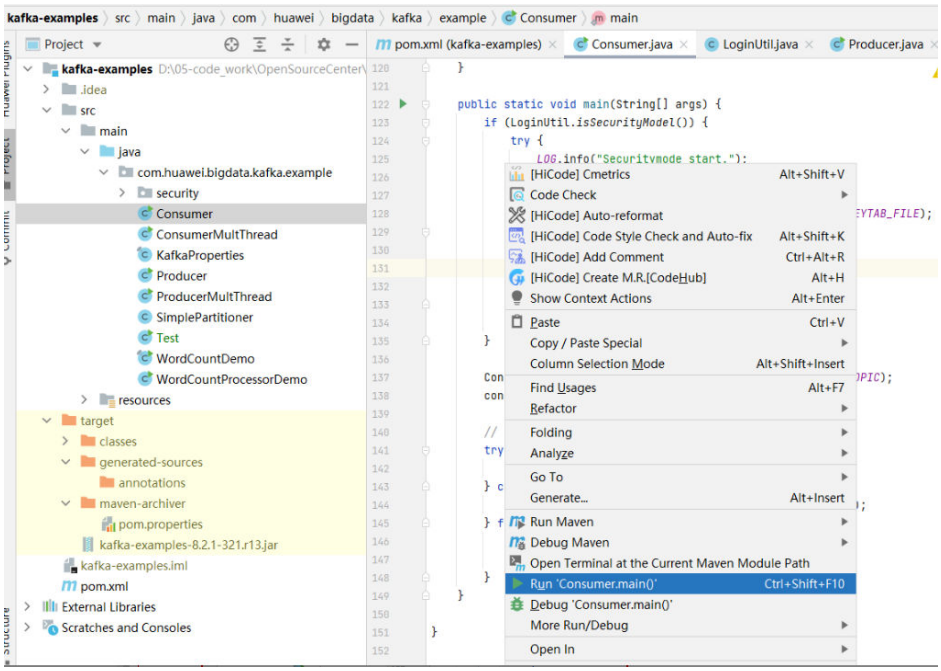
- 如需在Windows调测程序，需要配置Windows通过EIP访问集群Kafka，详情请参见**配置Windows通过EIP访问普通模式集群Kafka**。
- 如需在Linux调测程序，需要确保当前用户对“src/main/resources”目录下和依赖库文件目录下的所有文件，均具有可读权限。同时保证已安装Jdk并已设置java相关环境变量。

在 Windows 中调测程序

步骤1 确保本地的hosts文件中配置了远程集群所有主机的主机名和业务IP映射关系。

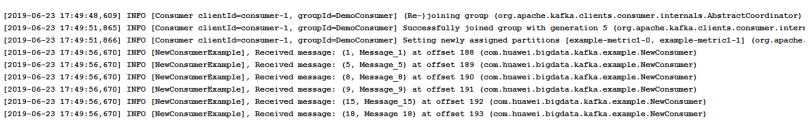
步骤2 通过IntelliJ IDEA可直接运行Consumer.java，如**图20-15**所示：

图 20-15 运行 Consumer.java



步骤3 单击运行后弹出控制台窗口，可以看到Consumer启动成功后，再启动Producer，即可看到实时接收消息：

图 20-16 Consumer.java 运行窗口



----结束

在 Linux 调测程序

步骤1 编译并生成Jar包，并将Jar包复制到与依赖库文件夹同级的目录“src/main/resources”下，具体步骤请参考**在Linux调测程序**。

步骤2 运行Consumer样例工程的命令如下。

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.Consumer

----结束

20.6.3 调测 Kafka High Level KafkaStreams API 样例程序

在 Windows 中调测程序

在Windows环境调测程序步骤请参考**在Windows中调测程序**。

在 Linux 环境调测程序

步骤1 编译并生成Jar包，并将Jar包复制到与依赖库文件夹同级的目录“src/main/resources”下，具体步骤请参考**在Linux调测程序**。

步骤2 使用集群安装用户登录集群客户端节点。

cd /opt/client

source bigdata_env

步骤3 创建输入Topic和输出Topic，与样例代码中指定的Topic名称保持一致，输出Topic的清理策略设置为compact。

kafka-topics.sh --create --zookeeper *quorumpeer实例IP地址:ZooKeeper客户端连接端口*/**kafka --replication-factor 1 --partitions 1 --topic** *Topic名称*

quorumpeer实例IP地址可登录集群的FusionInsight Manager界面，在“集群 > 服务 > ZooKeeper > 实例”界面中查询，多个地址可用“,”分隔。ZooKeeper客户端连接端口可通过ZooKeeper服务配置参数“clientPort”查询，例如端口号为2181。

例如执行以下命令：

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-input

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-output --config cleanup.policy=compact

步骤4 Topic创建成功后，执行以下命令运行程序。

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.WordCountDemo

步骤5 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-producer.sh”向输入Topic中写入消息：

cd /opt/client

source bigdata_env

kafka-console-producer.sh --broker-list *Broker实例IP地址:Kafka连接端口* **--topic streams-wordcount-input --producer.config /opt/client/Kafka/kafka/config/producer.properties**

说明

- *Broker实例IP地址*：登录FusionInsight Manager，选择“集群 > 服务 > Kafka > 实例”，在实例列表页面中查看并记录任意一个Broker实例业务IP地址。
- *Kafka连接端口*：集群已启用Kerberos认证（安全模式）时Broker端口为“sasl.port”参数的值。集群未启用Kerberos认证（普通模式）时Broker端口为“port”的值。

步骤6 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-consumer.sh”从输出Topic消费数据，查看统计结果。

cd /opt/client

source bigdata_env

kafka-console-consumer.sh --topic streams-wordcount-output --bootstrap-server *Broker实例IP地址:Kafka连接端口* **--consumer.config /opt/client/Kafka/kafka/config/consumer.properties --from-beginning --property print.key=true --property print.value=true --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer --formatter kafka.tools.DefaultMessageFormatter**

向输入Topic中写入消息：

```
>This is Kafka Streams test
>test starting
>now Kafka Streams is running
>test end
```

消息输出：

```
this    1
is      1
kafka   1
streams 1
test    1
test    2
starting 1
now     1
kafka   2
streams 2
is      2
running 1
test    3
end     1
```

----结束

20.6.4 调测 Kafka Low Level KafkaStreams API 样例程序

在 Windows 中调测程序

在Windows环境调测程序步骤请参考**在Windows中调测程序**。

在 Linux 环境调测程序

步骤1 编译并生成Jar包，并将Jar包复制到与依赖库文件夹同级的目录“src/main/resources”下，具体步骤请参考**在Linux调测程序**。

步骤2 使用**root**用户登录安装了集群客户端的节点。

cd /opt/client

source bigdata_env

步骤3 创建输入Topic和输出Topic，与样例代码中指定的Topic名称保持一致，输出Topic的清理策略设置为compact。

kafka-topics.sh --create --zookeeper *quorumpeer实例IP地址:ZooKeeper客户端连接端口*/**kafka --replication-factor 1 --partitions 1 --topic** *Topic名称*

quorumpeer实例IP地址可登录集群的FusionInsight Manager界面，在“集群 > 服务 > ZooKeeper > 实例”界面中查询，多个地址可用“,”分隔。ZooKeeper客户端连接端口可通过ZooKeeper服务配置参数“clientPort”查询，例如端口号为2181。

例如执行以下命令：

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-processor-input

kafka-topics.sh --create --zookeeper 192.168.0.17:2181/kafka --replication-factor 1 --partitions 1 --topic streams-wordcount-processor-output --config cleanup.policy=compact

步骤4 Topic创建成功后，执行以下命令运行程序。

java -cp /opt/client/lib/*:/opt/client/src/main/resources com.huawei.bigdata.kafka.example.WordCountDemo

步骤5 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-producer.sh”向输入Topic中写入消息：

cd /opt/client

source bigdata_env

kafka-console-producer.sh --broker-list *Broker实例IP地址:Kafka连接端口* **--topic streams-wordcount-processor-input --producer.config /opt/client/Kafka/kafka/config/producer.properties**

说明

- *Broker实例IP地址*：登录FusionInsight Manager，选择“集群 > 服务 > Kafka > 实例”，在实例列表页面中查看并记录任意一个Broker实例业务IP地址。
- *Kafka连接端口*：集群已启用Kerberos认证（安全模式）时Broker端口为“sasl.port”参数的值。集群未启用Kerberos认证（普通模式）时Broker端口为“port”的值。

步骤6 重新打开一个客户端连接窗口，执行以下命令，使用“kafka-console-consumer.sh”从输出Topic消费数据，查看统计结果。

cd /opt/client

source bigdata_env

kafka-console-consumer.sh --topic streams-wordcount-processor-output --bootstrap-server *Broker实例IP地址:Kafka连接端口* **--consumer.config /opt/client/Kafka/kafka/config/consumer.properties --from-beginning --property print.key=true --property print.value=true**

向输入Topic中写入消息：


```
>This is Kafka Streams test
>test starting
>now Kafka Streams is running
>test end
```

消息输出：

```
this    1
is      1
kafka   1
streams 1
test    1
test    2
starting 1
now     1
kafka   2
streams 2
is      2
running 1
test    3
end     1
```

----结束

20.7 Kafka 应用开发常见问题

20.7.1 Kafka 常用 API 介绍

20.7.1.1 Kafka Shell 命令介绍

- 查看当前集群Topic列表。

  shkafka-topics.sh --list --zookeeper <*ZooKeeper集群IP:2181/**kafka***>**

  shkafka-topics.sh --list --bootstrap-server *<Kafka集群IP:21007>* **--command-config config/client.properties**

- 查看单个Topic详细信息。

  shkafka-topics.sh --describe --zookeeper <*ZooKeeper集群IP:2181/**kafka***> --topic** *<Topic名称>*

  shkafka-topics.sh --describe --bootstrap-server *<Kafka集群IP:21007>* **--command-config config/client.properties --topic** *<Topic名称>*

- 删除Topic，由管理员用户操作。

  shkafka-topics.sh --delete --zookeeper <*ZooKeeper集群IP:2181/**kafka***> --topic** *<Topic名称>*

  sh**kafka-topics.sh --delete --bootstrap-server** *<Kafka集群IP:21007>*-- **command-config config/client.properties --topic** *<Topic名称>*

- 创建Topic，由管理员用户操作。

  shkafka-topics.sh --create --zookeeper <*ZooKeeper集群IP:2181/**kafka***> --partitions 6 --replication-factor 2 --topic** *<Topic名称>*

  shkafka-topics.sh --create --bootstrap-server *<Kafka集群IP:21007>* **--command-config config/client.properties --partitions 6 --replication-factor 2 --topic** *<Topic名称>*

- 赋Consumer权限命令，由管理员用户操作。

shkafka-acls.sh --authorizer-properties zookeeper.connect=<ZooKeeper集群IP:*2181*/**kafka > --add --allow-principal User:***<用户名>* **--consumer --topic** *<Topic名称>* **--group** *<消费者组名称>*

shkafka-acls.sh --bootstrap-server <*Kafka集群IP:**21007***> **--command-config config/client.properties --add --allow-principal User:***<用户名>* **--consumer --topic** *<Topic名称>* **--group** *<消费者组名称>*

- 赋Producer权限命令，由管理员用户操作。

  shkafka-acls.sh --authorizer-properties zookeeper.connect=<ZooKeeper集群IP:***2181***/kafka > --add --allow-principal User:***<用户名>* **--producer --topic** *<Topic名称>*

  shkafka-acls.sh --bootstrap-server <*Kafka集群IP:**21007***> **--command-config config/client.properties --add --allow-principal User:***<用户名>* **--producer --topic** <*Topic名称*>

- 生产消息，需要拥有该Topic生产者权限。

  shkafka-console-producer.sh --broker-list <*Kafka集群IP:**21007***> **--topic** *<Topic名称>* **--producer.config config/producer.properties**

- 消费数据，需要拥有该Topic的消费者权限。

  shkafka-console-consumer.sh --topic *<Topic名称>* **--bootstrap-server** <*Kafka集群IP:**21007***> **--consumer.config config/consumer.properties**

说明

- Shell命令需要在目录“*客户端安装目*录/Kafka/kafka/bin”下执行。
- 凡可指定“ * ”值以代表all value，且格式为“*--参数 参数值*”，例如： --group *，必须通过单引号或者等于号赋值，例如：--group '*' 或者 --group=*。

20.7.1.2 Kafka Java API 介绍

Kafka相关接口同开源社区保持一致，详情请参见**https://kafka.apache.org/24/documentation.html**。

Producer 重要接口

表 20-5 Producer 重要参数

参数	描述	备注
bootstrap.servers	Broker地址列表。	生产者通过此参数值，创建与Broker之间的连接。
sasl.kerberos.service.name	服务名。	Kafka集群运行，所使用的Kerberos用户名（需配置为kafka）。
key.serializer	消息Key值序列化类。	指定消息Key值序列化方式。
value.serializer	消息序列化类。	指定所发送消息的序列化方式。

表 20-6 Producer 重要接口函数

返回值类型	接口函数	描述
java.util.concurrent.Future<RecordMetadata>	send(ProducerRecord<K, V> record)	不带回调函数的发送接口，通常使用Future的get()函数阻塞发送，实现同步发送。
java.util.concurrent.Future<RecordMetadata>	send(ProducerRecord<K, V> record, Callback callback)	带回调函数的发送接口，通常用于异步发送后，通过回调函数实现对发送结果的处理。
void	onCompletion(RecordMetadata metadata, Exception exception);	回调函数接口方法，通过实现Callback中的此方法来进行异步发送结果的处理。

Consumer 重要接口

表 20-7 Consumer 重要参数

参数	描述	备注
bootstrap.servers	Broker地址列表。	消费者通过此参数值，创建与Broker之间的连接。
sasl.kerberos.service.name	服务名。	Kafka集群运行，所使用的Kerberos用户名（需配置为kafka）。
key.deserializer	消息Key值反序列化类。	反序列化消息Key值。
value.deserializer	消息反序列化类。	反序列化所接收的消息。

表 20-8 Consumer 重要接口函数

返回值类型	接口函数	描述
void	close()	关闭Consumer接口方法。
void	subscribe(java.util.Collection<java.lang.String> topics)	Topic订阅接口方法。
ConsumerRecords<K,V>	poll(final Duration timeout)	请求获取消息接口方法。

20.7.2 配置 Windows 通过 EIP 访问普通模式集群 Kafka

操作场景

该章节通过指导用户配置集群绑定EIP，并配置Kafka文件的方法，方便用户可以在本地对样例文件进行编译。

操作步骤

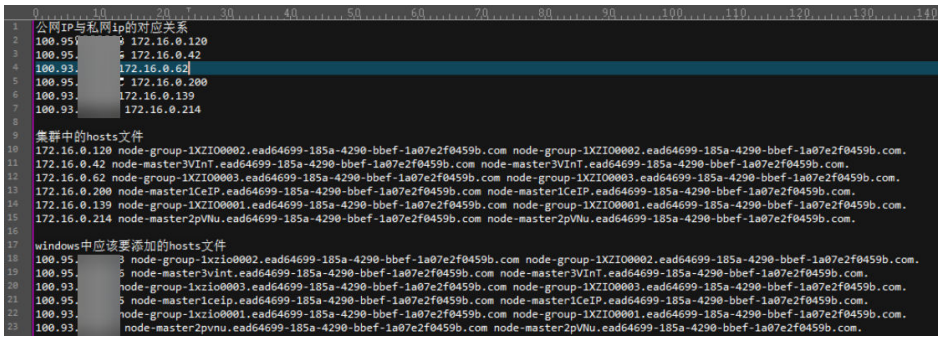
步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。

图 20-17 配置 hosts 文件



步骤2 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

图 20-18 管理安全组规则

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和9092端口。

图 20-19 添加入方向规则

步骤3 在Manager界面选择“集群 > 服务 > Kafka > 配置 > 全部配置”，搜索“kafka.config.expandor”，并添加参数名为“advertised.listeners”，值为“PLAINTEXT://:9092,SSL://:9093,TRACE://:21013”，修改配置保存后重启Kafka集群。

说明

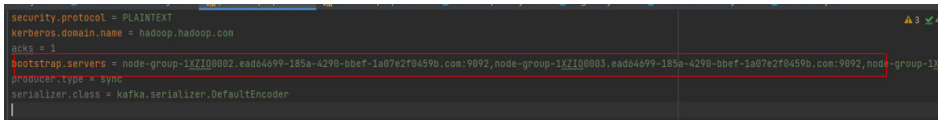
如果当前集群为MRS 3.2.0-LTS.1，执行该步骤无法通过EIP访问Kafka时，可以参考如下操作进行处理：

1. 登录到FusionInsight Manager页面，选择“集群 > 服务 > Kafka > 实例”，勾选所有Broker实例，选择“更多 > 停止实例”验证管理员密码后停止所有Broker实例。(该操作对业务有影响，请在业务低峰期操作)
2. 以**root**用户**分别**登录到Broker节点后台，修改“server.properties”配置。

   vi ${BIGDATA_HOME}/FusionInsight_HD_*/*_*_Broker/etc/server.properties

   将“host.name”修改为当前登录Broker节点的主机名称，将“listeners”及“advertised.listeners”参数值修改为“EXTERNAL_PLAINTEXT://*{主机名称}*:*{port}*”。
3. 登录到FusionInsight Manager页面，选择“集群 > 服务 > Kafka > 实例”，勾选所有Broker实例，单击“启动实例”。
4. 给MRS服务集群的Broker节点分别绑定EIP。
5. 在Windows通过已配置的Broker节点EIP地址和端口连接到Kafka集群并调试代码。

步骤4 运行样例代码前，修改样例代码中Kafka的连接串为*hostname1*:9092,*hostname2*:9092,*hostname3*:9092；修改代码中的域名。



说明

用户可登录FusionInsight Manager，选择“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。

----结束

20.7.3 运行 Producer.java 样例报错获取元数据失败“ERROR fetching topic metadata...”

解决步骤

步骤1 检查工程conf目录下“producer.properties”中配置的“bootstrap.servers”配置值中访问的IP和端口是否正确：

- 如果IP与Kafka集群部署的业务IP不一致，那么需要修改为当前集群正确的IP地址。
- 如果配置中的端口为21007（Kafka安全模式端口），那么修改该端口为9092（Kafka普通模式端口）。

步骤2 检查网络是否正常，确保当前机器能够正常访问Kafka集群。

----结束

21 Kudu 开发指南（安全模式）

21.1 Kudu 应用开发概述

21.1.1 Kudu 应用开发简介

Kudu 简介

Kudu是专为Apache Hadoop平台开发的列式存储管理器，具有Hadoop生态系统应用程序的共同技术特性：在通用的商用硬件上运行，可水平扩展，提供高可用性。

Kudu的设计具有以下优点：

- 能够快速处理OLAP工作负载。
- 支持与MapReduce，Spark和其他Hadoop生态系统组件集成。
- 与Apache Impala的紧密集成，使其成为将HDFS与Apache Parquet结合使用的更好选择。
- 提供强大而灵活的一致性模型，允许您根据每个请求选择一致性要求，包括用于严格可序列化的一致性的选项。
- 提供同时运行顺序读写和随机读写的良好性能。
- 易于管理。
- 高可用性。Master和TServer采用raft算法，该算法可确保只要副本总数的一半以上可用，tablet就可以进行读写操作。例如，如果3个副本中有2个副本或5个副本中有3个副本可用，则tablet可用。即使主tablet出现故障，也可以通过只读的副tablet提供读取服务。
- 支持结构化数据模型。

通过结合所有以上属性，Kudu的目标是支持在当前Hadoop存储技术上难以实现或无法实现的应用。

Kudu的应用场景有：

- 需要最终用户立即使用新到达数据的报告型应用。
- 同时支持大量历史数据查询和细粒度查询的时序应用。

- 使用预测模型并基于所有历史数据定期刷新预测模型来做出实时决策的应用。

Kudu 开发接口简介

Kudu本身是由C++语言开发的，但它支持使用C++、Java、Python等语言进行程序开发，推荐用户使用Java语言进行Kudu应用程序开发。

Kudu采用的接口与Apache Kudu保持一致，请参考**https://kudu.apache.org/apidocs/**。

21.1.2 Kudu 应用开发常用概念

Table

Kudu Table可以创建为内部表或外部表，其中内部表由Impala管理，而外部表不由Impala管理，但可以通过Impala进行查询。

Table有schema和primary key属性，且可以划分为多个tablet。

Tablet

Tablet是指数据分片，可以指定副本数，存放在多个tablet server上，多个副本中有一个是leader tablet；所有的副本都可以读，但是写操作只有leader tablet可以，写操作利用一致性算法（Raft）。

Tablet server

Tablet server是数据存储节点，存放tablet并且响应client请求，一个tablet server存放多个tablet。

Master

Master是中心管理节点，负责管理所有的tablet、tablet server以及副本之间的关联关系。同一时间集群中只有一个acting master（leader master），如果leader master故障，一个新的master会通过Raft算法选举出来。所有的master数据都存放在一个tablet中，这个tablet会被复制到所有的candidate master上；tablet server会定期向master发送心跳。

Kudu

Kudu的管理工具，可以用来检查集群的健康状况、日常运维等操作。

keytab 文件

存放用户信息的密钥文件，应用程序采用此密钥文件在组件中进行API方式认证。

Schema

表信息，用来表示表中列的信息。

21.1.3 Kudu 应用开发流程

开发流程中各阶段的说明如**图21-1**和**表21-1**所示。

图 21-1 Kudu 应用程序开发流程

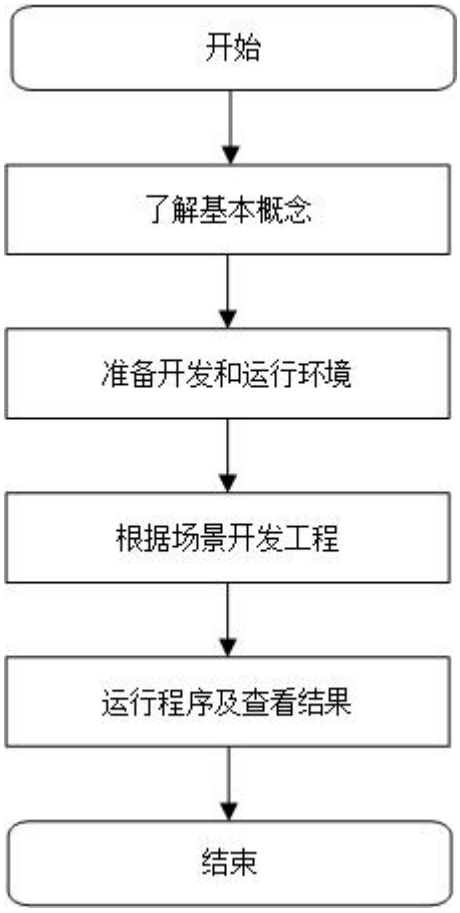


表 21-1 Kudu 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Kudu的基本概念。	**Kudu应用开发常用概念**
准备开发和运行环境	Kudu的应用程序支持多种语言进行开发，一般使用Java为主，推荐使用Eclipse或者IntelliJ IDEA工具，请根据指导完成开发环境配置。	**准备本地应用开发环境**
根据场景开发工程	提供样例工程，帮助用户快速了解Kudu各部件的编程接口。	**开发Kudu应用**
查看程序运行结果	指导用户将开发好的程序编译提交运行并查看结果。	**调测Kudu应用**

21.2 准备 Kudu 应用开发环境

21.2.1 准备本地应用开发环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如**表21-2**所示。

表 21-2 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境： Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_242，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK，支持1.8版本。 • TaiShan客户端：OpenJDK，支持1.8.0_242版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
准备开发用户	参考**准备MRS应用开发用户**进行操作，准备用于应用开发的集群用户并授予相应权限。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的Linux环境，用于验证应用程序运行正常。

1. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。
   客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。
   集群的Master节点或者Core节点已默认安装好客户端，可直接使用，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。
2. **登录FusionInsight Manager页面**，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig/Kudu/config”路径下的所有配置文件至客户端节点，放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

准备开发用户时获取的keytab文件也放置于该目录下，主要配置文件说明如**表21-3**所示。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp Kudu/config/* root@*客户端节点IP地址*:/opt/client/conf

表 21-3 配置文件

文件名称	作用
user.keytab	对于Kerberos安全认证提供用户信息。
krb5.conf	Kerberos Server配置信息。

3. 检查客户端节点网络连接。

   在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

21.2.2 准备 Kudu 应用安全认证

场景说明

访问安全集群环境中的服务，需要先通过Kerberos安全认证。所以Kudu应用程序中需要有安全认证代码，确保Kudu程序能够正常运行。

安全认证有两种方式：

- 命令行认证：

  提交Kudu应用程序运行前，在Kudu客户端执行如下命令进行认证。

  kinit *组件业务用户*

  说明

  该方式仅适用于Linux操作系统，且安装了Kudu的客户端。

- 代码认证：

  通过获取客户端的principal配置文件和keytab文件进行认证。

21.3 开发 Kudu 应用

21.3.1 Kudu 应用程序开发思路

开发思路

通过典型场景，可以快速学习和掌握Kudu的开发过程，并对关键的接口函数有所了解。

作为存储引擎，通常情况下Kudu会和计算引擎一起协同工作：

- 首先在计算引擎上（比如Impala）用SQL语句创建表对象；
- 然后通过Kudu的驱动往这个表里写数据；
- 在计算引擎上直接查询这个表里的数据。

在本开发程序示例中，为了不引入额外的计算引擎，将以Kudu为主，全部通过Java API接口来进行描述：

- 建立Kudu连接
- 创建Kudu表
- 写Kudu数据
- 修改Kudu表
- 删除Kudu表

21.3.2 开发 Kudu 应用

21.3.2.1 建立 Kudu 连接

功能简介

通过KuduClient.KuduClientBuilder(KUDU_MASTERS).build()方法创建KuduClient对象。传入的参数KUDU_MASTERS为Kudu集群的Master地址列表，如果有多个Master节点，则中间用半角逗号隔开。

代码样例

如下是建立连接代码片段：

```
// 创建Kudu连接对象
KuduClient client = new KuduClient.KuduClientBuilder(KUDU_MASTERS).build();
```

说明

示例代码中，KUDU_MASTERS为Kudu集群的Masters地址列表，例如：192.168.0.100:7051, 192.168.0.101:7051, 192.168.0.102:7051。格式为地址:端口，中间用半角逗号隔开；建议使用长连接对象。

21.3.2.2 创建 Kudu 表

功能简介

通过KuduClient.createTable(String name, Schema schema, CreateTableOptions builder)方法创建表对象，其中需要指定表的schema和分区信息。

代码样例

如下是创建表的代码片段：

```
// Set up a table name.
String tableName = "example";

// Set up a simple schema.
List<ColumnSchema> columns = new ArrayList<>(2);
columns.add(new ColumnSchema.ColumnSchemaBuilder("key", Type.INT32)
        .key(true)
        .build());
columns.add(new ColumnSchema.ColumnSchemaBuilder("value",Type.STRING).nullable(true)
        .build());
Schema schema = new Schema(columns);

// Set up the partition schema, which distributes rows to different tablets by hash.
// Kudu also supports partitioning by key range. Hash and range partitioning can be combined.
// For more information, see http://kudu.apache.org/docs/schema_design.html.
CreateTableOptions cto = new CreateTableOptions();
List<String> hashKeys = new ArrayList<>(1);
hashKeys.add("key");
int numBuckets = 8;
cto.addHashPartitions(hashKeys, numBuckets);

// Create the table.
client.createTable(tableName, schema, cto);
```

说明

示例代码中，定义了一张表，包含key和value两列。其中key是int32类型的主键字段，value是string类型的非主键字段且可以为空；同时该表在主键字段key上做了8个hash分区，表示数据会分成8个独立的tablet。

21.3.2.3 打开 Kudu 表

功能简介

通过KuduClient.openTable(final String name)方法打开表对象。

代码样例

如下是打开表的代码片段：

```
// 打开Kudu表
KuduTable table = client.openTable(tableName);
```

说明

示例代码中，tableName需要打开的表名。

21.3.2.4 修改 Kudu 表

功能简介

通过KuduClient.alterTable(String name, AlterTableOptions ato)方法修改表对象。

代码样例

如下是写数据的代码片段：

```
// Alter the table, adding a column with a default value.
// Note: after altering the table, the table needs to be re-opened.
AlterTableOptions ato = new AlterTableOptions();
ato.addColumn("added", org.apache.kudu.Type.DOUBLE, DEFAULT_DOUBLE);
client.alterTable(tableName, ato);
```

说明

示例代码中，AlterTableOptions是要修改表属性的集合，这里需要在表格中新增一列。

21.3.2.5 写 Kudu 数据

功能简介

通过KuduClient.newSession()方法生成一个KuduSession对象，然后再把插入记录动作执行到Kudu表里。

代码样例

如下是写数据的代码片段：

```
// Create a KuduSession.
KuduSession session = client.newSession();
for (int i = 0; i < numRows; i++) {
  Insert insert = table.newInsert();
  PartialRow row = insert.getRow();
  row.addInt("key", i);
  // Make even-keyed row have a null 'value'.
  if (i % 2 == 0) {
    row.setNull("value");
  } else {
    row.addString("value", "value " + i);
  }
  session.apply(insert);
}

// Call session.close() to end the session and ensure the rows are
// flushed and errors are returned.
// You can also call session.flush() to do the same without ending the session.
// When flushing in AUTO_FLUSH_BACKGROUND mode (the default mode recommended)
// for most workloads, you must check the pending errors as shown below, since
// write operations are flushed to Kudu in background threads.
session.close();
if (session.countPendingErrors() != 0) {
System.out.println("errors inserting rows");
  org.apache.kudu.client.RowErrorsAndOverflowStatus roStatus = session.getPendingErrors();
  org.apache.kudu.client.RowError[] errs = roStatus.getRowErrors();
  int numErrs = Math.min(errs.length, 5);
  System.out.println("there were errors inserting rows to Kudu");
  System.out.println("the first few errors follow:");
  for (int i = 0; i < numErrs; i++) {
    System.out.println(errs[i]);
  }
  if (roStatus.isOverflowed()) {
    System.out.println("error buffer overflowed: some errors were discarded");
  }
  throw new RuntimeException("error inserting rows to Kudu");
}
System.out.println("Inserted " + numRows + " rows");
```

说明

示例代码中，numRows是要写入的记录条数；需要特别注意写入请求的响应结果。

21.3.2.6 读 Kudu 数据

功能简介

通过KuduClient.newScannerBuilder(KuduTable table)方法生成一个KuduScanner对象，然后再通过设置谓词条件从Kudu表里过滤读取数据。

代码样例

如下是读取数据的代码片段：

```
KuduTable table = client.openTable(tableName);
Schema schema = table.getSchema();

// Scan with a predicate on the 'key' column, returning the 'value' and "added" columns.
List<String> projectColumns = new ArrayList<>(2);
projectColumns.add("key");
projectColumns.add("value");
projectColumns.add("added");
int lowerBound = 0;
KuduPredicate lowerPred = KuduPredicate.newComparisonPredicate(
schema.getColumn("key"),
    ComparisonOp.GREATER_EQUAL,
    lowerBound);
int upperBound = numRows / 2;
KuduPredicate upperPred = KuduPredicate.newComparisonPredicate(
schema.getColumn("key"),
    ComparisonOp.LESS,
    upperBound);
KuduScanner scanner = client.newScannerBuilder(table)
    .setProjectedColumnNames(projectColumns)
    .addPredicate(lowerPred)
    .addPredicate(upperPred)
    .build();

// Check the correct number of values and null values are returned, and
// that the default value was set for the new column on each row.
// Note: scanning a hash-partitioned table will not return results in primary key order.
int resultCount = 0;
int nullCount = 0;
while (scanner.hasMoreRows()) {
RowResultIterator results = scanner.nextRows();
  while (results.hasNext()) {
    RowResult result = results.next();
    if (result.isNull("value")) {
      nullCount++;
    }
    double added = result.getDouble("added");
    if (added != DEFAULT_DOUBLE) {
      throw new RuntimeException("expected added=" + DEFAULT_DOUBLE +
          " but got added= " + added);
    }
    resultCount++;
  }
}
```

说明

示例代码中：

- projectColumns表示要返回的列信息。
- lowerPred和upperBound表示作用在主键key上的谓词。

21.3.2.7 删除 Kudu 表

功能简介

通过KuduClient.deleteTable(String name)方法删除表对象。

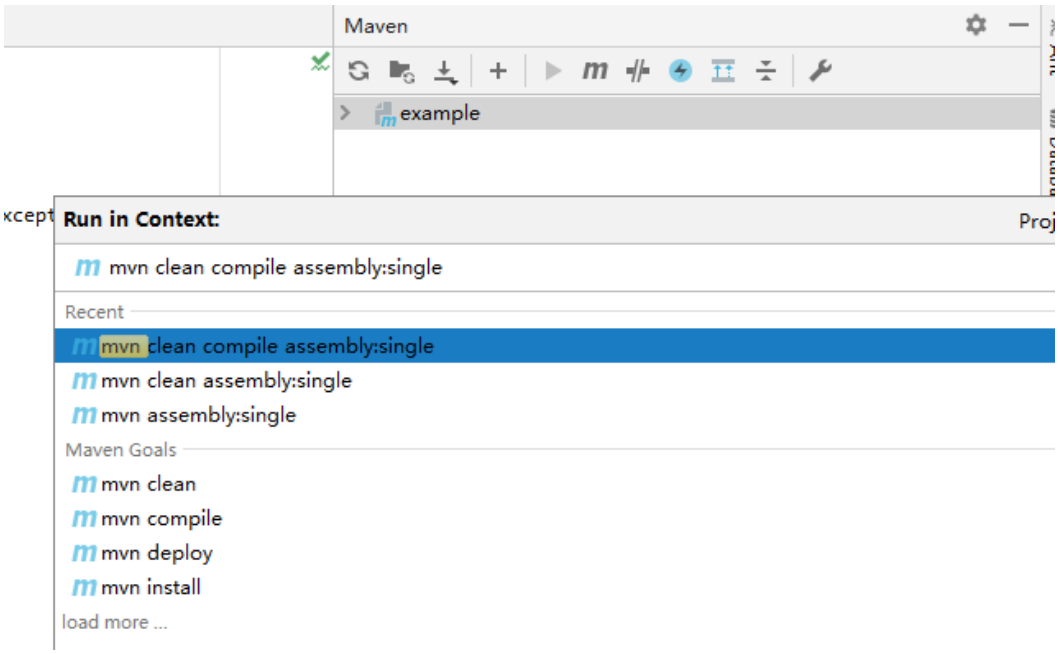
代码样例

如下是删除表的代码片段：

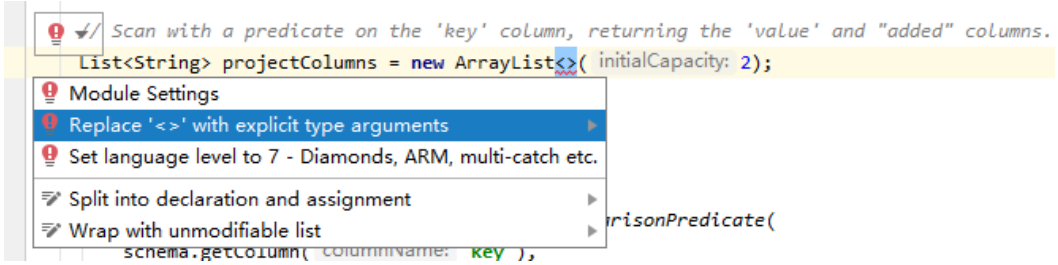
```
// Delete the table.
client.deleteTable(tableName);
```

21.4 调测 Kudu 应用

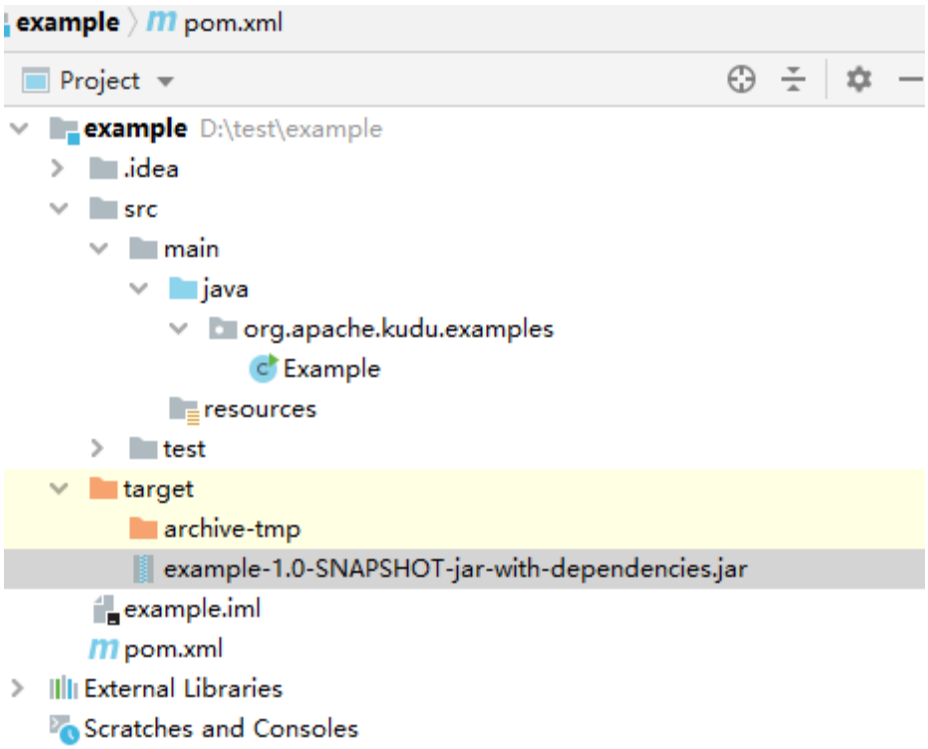
步骤1 使用Maven编译。



步骤2 解决“Error:(123,49) java: -source 1.5 中不支持 diamond 运算符”错误：



得到输出包：



步骤3 上传jar包到Linux服务器执行。

```
[root@ecs-a4e6 ~]# java -DkuduMasters=localhost:7051 -jar example-1.0-SNAPSHOT-jar-with-dependencies.jar
-----------------------------------------------
Will try to connect to Kudu master(s) at localhost:7051
Run with -DkuduMasters=master-0:port,master-1:port,... to override.
-----------------------------------------------
log4j:WARN No appenders could be found for logger (org.apache.kudu.shaded.io.netty.util.internal.logging.InternalLoggerFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Created table java_example-1599014880440
Inserted 150 rows
Altered the table
Scanned some rows and checked the results
Deleted the table
[root@ecs-a4e6 ~]#
```

----结束

21.5 Kudu 应用开发常见问题

Kudu Java API 接口介绍

Kudu完整和详细的接口可以直接参考官方网站上的描述： **https://kudu.apache.org/apidocs**。

22 Kudu 开发指南（普通模式）

22.1 Kudu 应用开发概述

22.1.1 Kudu 应用开发简介

Kudu 简介

Kudu是专为Apache Hadoop平台开发的列式存储管理器，具有Hadoop生态系统应用程序的共同技术特性：在通用的商用硬件上运行，可水平扩展，提供高可用性。

Kudu的设计具有以下优点：

- 能够快速处理OLAP工作负载。
- 支持与MapReduce，Spark和其他Hadoop生态系统组件集成。
- 与Apache Impala的紧密集成，使其成为将HDFS与Apache Parquet结合使用的更好选择。
- 提供强大而灵活的一致性模型，允许您根据每个请求选择一致性要求，包括用于严格可序列化的一致性的选项。
- 提供同时运行顺序读写和随机读写的良好性能。
- 易于管理。
- 高可用性。Master和TServer采用raft算法，该算法可确保只要副本总数的一半以上可用，tablet就可以进行读写操作。例如，如果3个副本中有2个副本或5个副本中有3个副本可用，则tablet可用。即使主tablet出现故障，也可以通过只读的副tablet提供读取服务。
- 支持结构化数据模型。

通过结合所有以上属性，Kudu的目标是支持在当前Hadoop存储技术上难以实现或无法实现的应用。

Kudu的应用场景有：

- 需要最终用户立即使用新到达数据的报告型应用。
- 同时支持大量历史数据查询和细粒度查询的时序应用。

- 使用预测模型并基于所有历史数据定期刷新预测模型来做出实时决策的应用。

Kudu 开发接口简介

Kudu本身是由C++语言开发的，但它支持使用C++、Java、Python等语言进行程序开发，推荐用户使用Java语言进行Kudu应用程序开发。

Kudu采用的接口与Apache Kudu保持一致，请参考**https://kudu.apache.org/apidocs/**。

22.1.2 Kudu 应用开发常用概念

Table

Table有schema和primary key属性，且可以划分为多个tablet。

Tablet

Tablet是指数据分片，可以指定副本数，存放在多个tablet server上，多个副本中有一个是leader tablet；所有的副本都可以读，但是写操作只有leader可以，写操作利用一致性算法（Raft）。

Tablet server

Tablet server是数据存储节点，存放tablet并且响应client请求，一个tablet server存放多个tablet。

Master

Master是中心管理节点，负责管理所有的tablet、tablet server以及副本之间的关联关系。同一时间集群中只有一个acting master（leader master），如果leader master挂了，一个新的master会通过Raft算法选举出来。所有的master数据都存放在一个tablet中，这个tablet会被复制到所有的candidate master上；tablet server会定期向master发送心跳。

kudu

kudu的管理工具，可以用来检查集群的健康状况、日常运维等操作。

keytab 文件

存放用户信息的密钥文件，应用程序采用此密钥文件在组件中进行API方式认证。

Schema

表信息，用来表示表中列的信息。

22.1.3 Kudu 应用开发流程

开发流程中各阶段的说明如**图22-1**和**表22-1**所示。

图 22-1 Kudu 应用程序开发流程

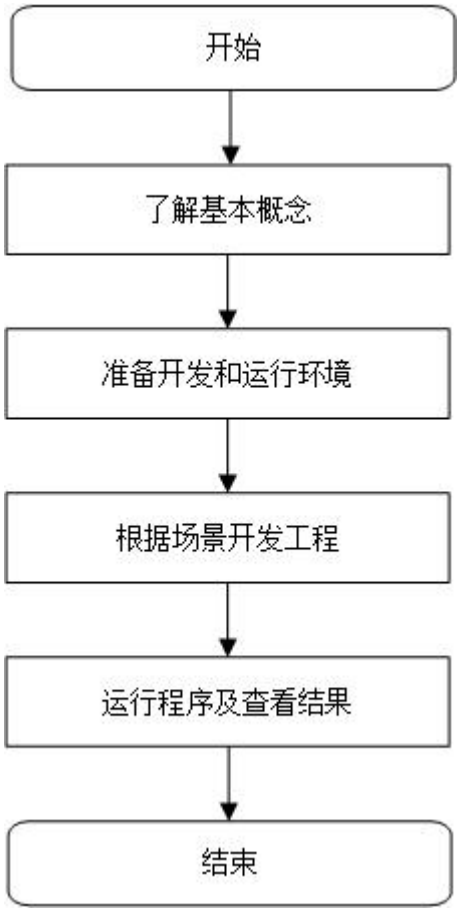


表 22-1 Kudu 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Kudu的基本概念。	**Kudu应用开发常用概念**
准备开发和运行环境	Kudu的应用程序支持多种语言进行开发，一般使用Java为主，推荐使用Eclipse 或者IntelliJ IDEA工具，请根据指导完成开发环境配置。	**准备本地应用开发环境**
根据场景开发工程	提供样例工程，帮助用户快速了解Kudu各部件的编程接口。	**开发Kudu应用**
查看程序运行结果	指导用户将开发好的程序编译提交运行并查看结果。	**调测Kudu应用**

22.2 准备 Kudu 应用开发环境

22.2.1 准备本地应用开发环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如表22-2所示。

表 22-2 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows7以上版本。 • 运行环境： Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_242，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK，支持1.8版本。 • TaiShan客户端：OpenJDK，支持1.8.0_242版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的Linux环境，用于验证应用程序运行正常。

1. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。

   客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。

   集群的Master节点或者Core节点已默认安装好客户端，可直接使用，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

2. **登录FusionInsight Manager页面**，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig/Kudu/config”路径下的所有配置文件至客户端节点，放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

   例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp Kudu/config/* root@*客户端节点IP地址*:/opt/client/conf

3. 检查客户端节点网络连接。

   在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

22.3 开发 Kudu 应用

22.3.1 Kudu 应用程序开发思路

通过典型场景，可以快速学习和掌握Kudu的开发过程，并对关键的接口函数有所了解。

开发思路

作为存储引擎，通常情况下会和计算引擎一起协同工作：

- 首先在计算引擎上（比如Impala）用SQL语句创建表对象；
- 然后通过Kudu的驱动往这个表里写数据；
- 于此同时可以在计算引擎上直接查询这个表里的数据。

在本开发程序示例中，为了不引入额外的计算引擎，将以Kudu为主，全部通过Java API接口来进行描述：

- 建立Kudu连接
- 创建Kudu表
- 写Kudu数据
- 修改Kudu表
- 删除Kudu表

22.3.2 开发 Kudu 应用

22.3.2.1 建立 Kudu 连接

功能简介

通过KuduClient.KuduClientBuilder(KUDU_MASTERS).build()方法创建KuduClient对象。传入的参数KUDU_MASTERS为kudu集群的masters地址列表，如果有多个master节点，则中间用半角逗号隔开。

代码样例

如下是建立连接代码片段：

```
// 创建Kudu连接对象
KuduClient client = new KuduClient.KuduClientBuilder(KUDU_MASTERS).build();
```

说明

示例代码中，KUDU_MASTERS为kudu集群的masters地址列表，例如：192.168.0.100:7051, 192.168.0.101:7051, 192.168.0.102:7051。格式为地址:端口，中间用半角逗号隔开；生产上建议使用长连接对象。

22.3.2.2 创建 Kudu 表

功能简介

通过KuduClient.createTable(String name, Schema schema, CreateTableOptions builder)方法创建表对象，其中需要指定表的schema和分区信息。

代码样例

如下是创建表的代码片段：

```
// Set up a table name.
String tableName = “example”;

// Set up a simple schema.
List<ColumnSchema> columns = new ArrayList<>(2);
columns.add(new ColumnSchema.ColumnSchemaBuilder("key", Type.INT32)
        .key(true)
        .build());
columns.add(new ColumnSchema.ColumnSchemaBuilder("value",Type.STRING).nullable(true)
        .build());
Schema schema = new Schema(columns);

// Set up the partition schema, which distributes rows to different tablets by hash.
// Kudu also supports partitioning by key range. Hash and range partitioning can be combined.
// For more information, see http://kudu.apache.org/docs/schema_design.html.
CreateTableOptions cto = new CreateTableOptions();
List<String> hashKeys = new ArrayList<>(1);
hashKeys.add("key");
int numBuckets = 8;
cto.addHashPartitions(hashKeys, numBuckets);

// Create the table.
client.createTable(tableName, schema, cto);
```

说明

示例代码中，定义了一张表，包含key和value两列。其中key是int32类型的主键字段，value是string类型的非主键字段且可以为空；同时该表在主键字段key上做了8个hash分区，表示数据会分成8个独立的tablet。

22.3.2.3 打开 Kudu 表

功能简介

通过KuduClient.openTable(final String name)方法打开表对象。

代码样例

如下是打开表的代码片段：

```
// 打开Kudu表
KuduTable table = client.openTable(tableName);
```

说明

示例代码中，tableName是要打开的表名。

22.3.2.4 修改 Kudu 表

功能简介

通过KuduClient.alterTable(String name, AlterTableOptions ato)方法修改表对象。

代码样例

如下是写数据的代码片段：

```
// Alter the table, adding a column with a default value.
// Note: after altering the table, the table needs to be re-opened.
AlterTableOptions ato = new AlterTableOptions();
ato.addColumn("added", org.apache.kudu.Type.DOUBLE, DEFAULT_DOUBLE);
client.alterTable(tableName, ato);
```

说明

示例代码中，AlterTableOptions是要修改表属性的集合，这里往表里新增加了一列。

22.3.2.5 写 Kudu 数据

功能简介

通过KuduClient.newSession()方法生成一个KuduSession对象，然后再把插入记录动作执行到Kudu表里。

代码样例

如下是写数据的代码片段：

```
// Create a KuduSession.
KuduSession session = client.newSession();
for (int i = 0; i < numRows; i++) {
  Insert insert = table.newInsert();
  PartialRow row = insert.getRow();
  row.addInt("key", i);
  // Make even-keyed row have a null 'value'.
  if (i % 2 == 0) {
    row.setNull("value");
  } else {
    row.addString("value", "value " + i);
  }
  session.apply(insert);
}

// Call session.close() to end the session and ensure the rows are
// flushed and errors are returned.
// You can also call session.flush() to do the same without ending the session.
// When flushing in AUTO_FLUSH_BACKGROUND mode (the default mode recommended)
```

```
// for most workloads, you must check the pending errors as shown below, since
// write operations are flushed to Kudu in background threads.
session.close();
if (session.countPendingErrors() != 0) {
System.out.println("errors inserting rows");
  org.apache.kudu.client.RowErrorsAndOverflowStatus roStatus = session.getPendingErrors();
  org.apache.kudu.client.RowError[] errs = roStatus.getRowErrors();
  int numErrs = Math.min(errs.length, 5);
  System.out.println("there were errors inserting rows to Kudu");
  System.out.println("the first few errors follow:");
  for (int i = 0; i < numErrs; i++) {
    System.out.println(errs[i]);
  }
  if (roStatus.isOverflowed()) {
    System.out.println("error buffer overflowed: some errors were discarded");
  }
  throw new RuntimeException("error inserting rows to Kudu");
}
System.out.println("Inserted " + numRows + " rows");
```

说明

示例代码中，numRows是要写入的记录条数；需要特别注意写入请求的响应结果。

22.3.2.6 读 Kudu 数据

功能简介

通过KuduClient.newScannerBuilder(KuduTable table)方法生成一个KuduScanner对象，然后再通过设置谓词条件从Kudu表里过滤读取数据。

代码样例

如下是读取数据的代码片段：

```
KuduTable table = client.openTable(tableName);
Schema schema = table.getSchema();

// Scan with a predicate on the 'key' column, returning the 'value' and "added" columns.
List<String> projectColumns = new ArrayList<>(2);
projectColumns.add("key");
projectColumns.add("value");
projectColumns.add("added");
int lowerBound = 0;
KuduPredicate lowerPred = KuduPredicate.newComparisonPredicate(
schema.getColumn("key"),
    ComparisonOp.GREATER_EQUAL,
    lowerBound);
int upperBound = numRows / 2;
KuduPredicate upperPred = KuduPredicate.newComparisonPredicate(
schema.getColumn("key"),
    ComparisonOp.LESS,
    upperBound);
KuduScanner scanner = client.newScannerBuilder(table)
    .setProjectedColumnNames(projectColumns)
    .addPredicate(lowerPred)
    .addPredicate(upperPred)
    .build();

// Check the correct number of values and null values are returned, and
// that the default value was set for the new column on each row.
// Note: scanning a hash-partitioned table will not return results in primary key order.
int resultCount = 0;
int nullCount = 0;
while (scanner.hasMoreRows()) {
RowResultIterator results = scanner.nextRows();
```

```
  while (results.hasNext()) {
    RowResult result = results.next();
    if (result.isNull("value")) {
      nullCount++;
    }
    double added = result.getDouble("added");
    if (added != DEFAULT_DOUBLE) {
      throw new RuntimeException("expected added=" + DEFAULT_DOUBLE +
          " but got added= " + added);
    }
    resultCount++;
  }
}
```

说明

示例代码中：

- projectColumns表示要返回的列信息。
- lowerPred和upperBound表示作用在主键key上的谓词。

22.3.2.7 删除 Kudu 表

功能简介

通过KuduClient.deleteTable(String name)方法删除表对象。

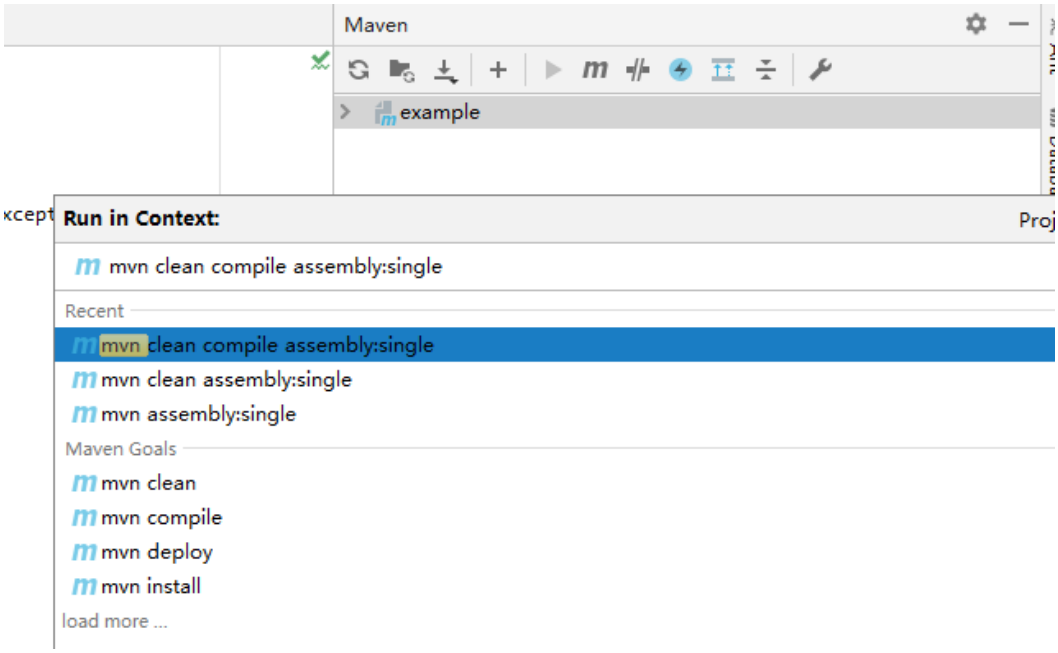
代码样例

如下是删除表的代码片段：

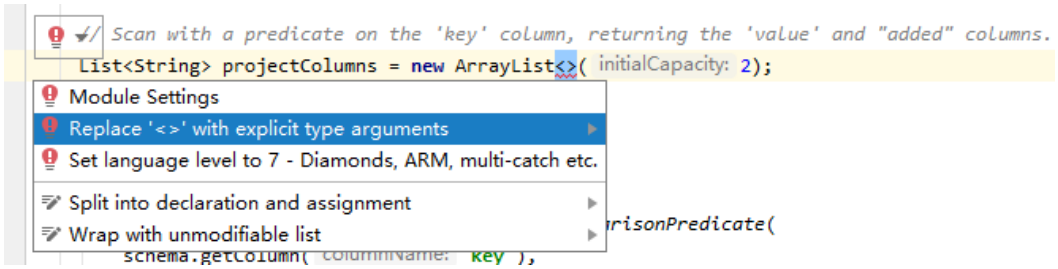
```
// Delete the table.
client.deleteTable(tableName);
```

22.4 调测 Kudu 应用

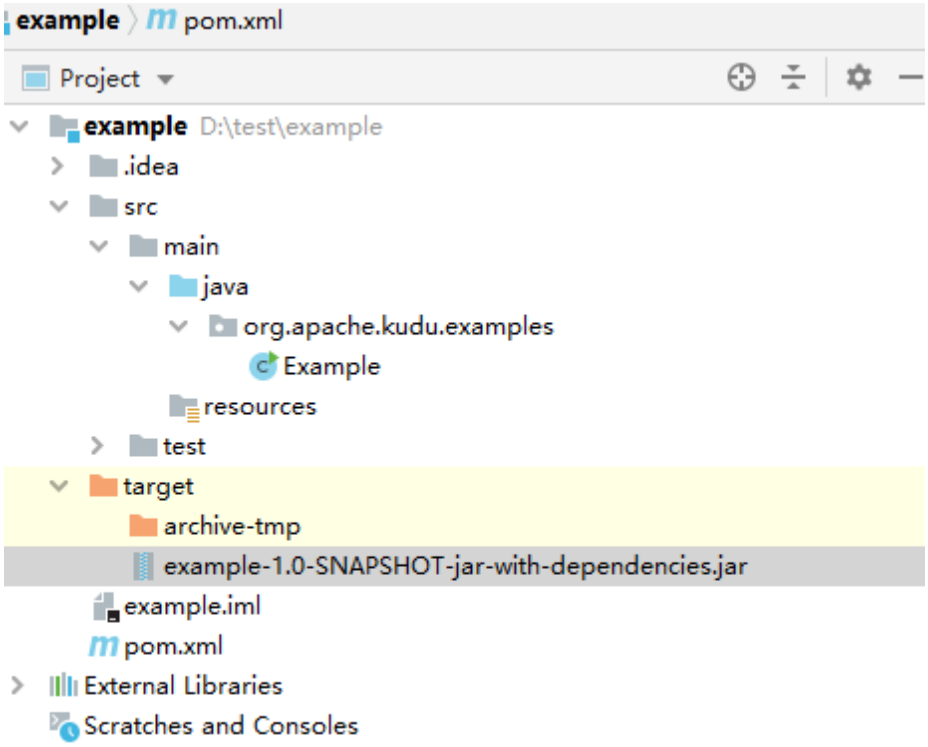
步骤1 使用Maven编译。



步骤2 解决“Error:(123,49) java: -source 1.5 中不支持 diamond 运算符”错误：



得到输出包：



步骤3 上传jar包到Linux服务器执行。

```
[root@ecs-a4e6 ~]# java -DkuduMasters=localhost:7051 -jar example-1.0-SNAPSHOT-jar-with-dependencies.jar
-------------------------------------------
Will try to connect to Kudu master(s) at localhost:7051
Run with -DkuduMasters=master-0:port,master-1:port,... to override.
-------------------------------------------
log4j:WARN No appenders could be found for logger (org.apache.kudu.shaded.io.netty.util.internal.logging.InternalLoggerFactory).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Created table java_example-1599014880440
Inserted 150 rows
Altered the table
Scanned some rows and checked the results
Deleted the table
[root@ecs-a4e6 ~]#
```

----结束

22.5 Kudu 应用开发常见问题

Kudu Java API 接口介绍

Kudu完整和详细的接口可以直接参考官方网站上的描述： **https://kudu.apache.org/apidocs**。

23 MapReduce 开发指南（安全模式）

23.1 MapReduce 应用开发简介

MapReduce 简介

Hadoop MapReduce是一个使用简易的并行计算软件框架，基于它写出来的应用程序能够运行在由上千个服务器组成的大型集群上，并以一种可靠容错的方式并行处理上TB级别的数据集。

一个MapReduce作业（application/job）通常会把输入的数据集切分为若干独立的数据块，由map任务（task）以完全并行的方式来处理。框架会对map的输出先进行排序，然后把结果输入给reduce任务，最后返回给客户端。通常作业的输入和输出都会被存储在文件系统中。整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

MapReduce主要特点如下：

- 大规模并行计算
- 适用于大型数据集
- 高容错性和高可靠性
- 合理的资源调度

常用概念

- Hadoop shell命令

  Hadoop基本shell命令，包括提交MapReduce作业，kill MapReduce作业，进行HDFS文件系统各项操作等。

- MapReduce输入输出(InputFormat，OutputFormat)

  MapReduce框架根据用户指定的InputFormat切割数据集，读取数据，并提供给map任务多条键值对进行处理，决定并行启动的map任务数目。MapReduce框架根据用户指定的OutputFormat，把生成的键值对输出为特定格式的数据。

  map、reduce两个阶段都处理在<key,value>键值对上，也就是说，框架把作业的输入作为一组<key,value>键值对，同样也产出一组<key,value>键值对作为作业的输出，这两组键值对的类型可能不同。对单个map和reduce而言，对键值对的处理为单线程串行处理。

框架需要对key和value的类(classes)进行序列化操作，因此，这些类需要实现Writable接口。另外，为了方便框架执行排序操作，key类必须实现WritableComparable接口。

一个MapReduce作业的输入和输出类型如下所示：

(input)<k1,v1> —> map —> <k2,v2> —> 汇总数据 —> <k2,List(v2)> —> reduce —> <k3,v3>(output)

- 业务核心

  应用程序通常只需要分别继承Mapper类和Reducer类，并重写其map和reduce方法来实现业务逻辑，它们组成作业的核心。

- MapReduce WebUI界面

  用于监控正在运行的或者历史的MapReduce作业在MapReduce框架各个阶段的细节，以及提供日志显示，帮助用户更细粒度地去开发、配置和调优作业。

- Keytab文件

  存放用户信息的密钥文件。应用程序采用此密钥文件在产品中进行API方式认证。

- 归档

  用来保证所有映射的键值对中的每一个共享相同的键组。

- 混洗

  从Map任务输出的数据到Reduce任务的输入数据的过程称为Shuffle。

- 映射

  用来把一组键值对映射成一组新的键值对。

23.2 MapReduce 应用开发流程介绍

开发流程中各阶段的说明如**图23-1**和**表23-1**所示。

图 23-1 MapReduce 应用程序开发流程

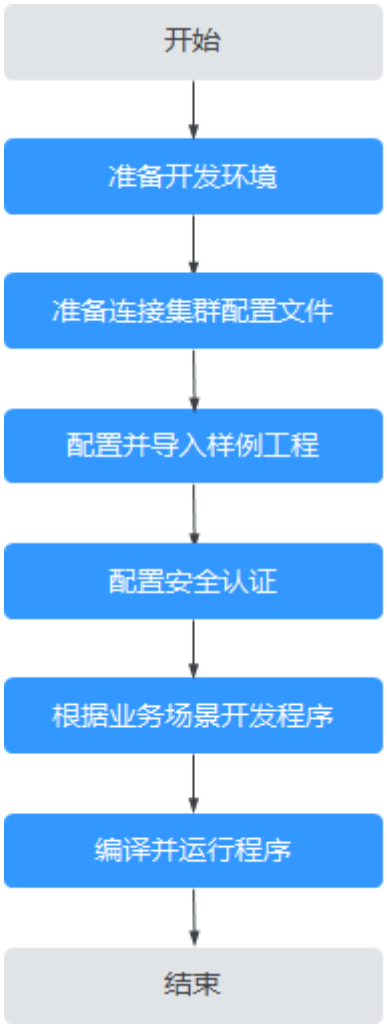


表 23-1 MapReduce 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	在进行应用开发前，需首先准备开发环境，推荐使用Java语言进行开发，使用IntelliJ IDEA工具，同时完成JDK、Maven等初始配置。	**准备MapReduce开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备连接MapReduce集群配置文件**

阶段	说明	参考文档
配置并导入样例工程	MapReduce提供了不同场景下的多种样例程序，用户可获取样例工程并导入本地开发环境中进行程序学习。	导入并配置MapReduce样例工程
配置安全认证	如果您使用的是开启了Kerberos认证的MRS集群，需要进行安全认证。	配置MapReduce应用安全认证
根据业务场景开发程序	根据实际业务场景开发程序，调用组件接口实现对应功能。	开发MapReduce应用
编译并运行程序	将开发好的程序编译运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。	调测MapReduce应用

23.3 MapReduce 样例工程介绍

MRS样例工程获取地址为https://github.com/huaweicloud/huaweicloud-mrs-example，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下MapReduce相关样例工程：

表 23-2 MapReduce 相关样例工程

样例工程位置	描述
mapreduce-example-security	• MapReduce统计数据的应用开发示例： 提供了一个MapReduce统计数据的应用开发示例，通过类CollectionMapper实现数据分析、处理，并输出满足用户需要的数据信息。 相关样例介绍请参见MapReduce统计样例程序。 • MapReduce作业访问多组件的应用开发示例： 以MapReduce访问HDFS、HBase、Hive为例，介绍如何编写MapReduce作业访问多个服务组件。帮助用户理解认证、配置加载等关键使用方式。 相关样例介绍请参见MapReduce访问多组件样例程序。

23.4 准备 MapReduce 应用开发环境

23.4.1 准备 MapReduce 开发环境

在进行应用开发时，要准备的开发和运行环境如表23-3所示。

表 23-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程
安装Maven	开发环境基本配置。用于项目管理，贯穿软件开发生命周期。
安装JDK	开发和运行环境的基本配置，版本要求如下： 服务端和客户端仅支持集群自带的OpenJDK，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的： • X86客户端： – Oracle JDK：支持1.8版本； – IBM JDK：支持1.8.0.7.20和1.8.0.6.15版本。 • ARM客户端： – OpenJDK：支持1.8.0_272版本（集群自带JDK，可通过集群客户端安装目录中“JDK”文件夹下获取）。 – 毕昇JDK：支持1.8.0_272版本。 **说明** • 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 • IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情可参考**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。 • 毕昇JDK详细信息可参考**https://www.hikunpeng.com/zh/developer/devkit/compiler/jdk**。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

23.4.2 准备连接 MapReduce 集群配置文件

说明

如果需要使用访问多组件样例程序，请确保集群已安装Hive、HBase服务。

准备集群认证用户信息

对于开启Kerberos认证的MRS集群，需提前准备具有相关组件操作权限的用户，并下载认证凭据文件用于程序认证。

以下MapReduce权限配置示例供参考，在实际业务场景中可根据业务需求灵活调整。

步骤1 登录FusionInsight Manager。

步骤2 选择“系统 > 权限 > 角色 > 添加角色”。

步骤3 填写角色的名称，例如**developrole**。

步骤4 在“配置资源权限”的表格中配置以下参数后，单击“确定”保存。

选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/”，勾选“user”的“读”、“写”、“执行”和“递归”。

如果要执行多组件用例，还需：

- 选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > tmp”，勾选“hive-scratch”的“读”、“写”、“执行”，若存在“examples”，勾选“examples”的“读”、“写”、“执行”和“递归”。
- 选择“*待操作集群的名称* > HBase > HBase Scope > global”勾选“default”的“创建”。
- 选择“待操作集群的名称 > HBase > HBase Scope > global > hbase”，勾选“hbase:meta”的“执行”。
- 选择“*待操作集群的名称* > Hive > Hive读写权限”，勾选“default”的 “查询”、“插入”、“建表”、“递归”。

步骤5 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选“default”的“提交”，单击“确定”保存。

步骤6 选择“用户 > 添加用户”，在新增用户界面，创建一个机机用户，例如**developuser**。

- “用户组”需加入“hadoop”用户组。
- “角色”加入新增的角色，例如**developrole**。

步骤7 使用**admin**用户登录FusionInsight Manager，选择“系统 > 权限 > 用户”，在用户名为**developuser**的操作列选择“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到该用户的“user.keytab”文件与“krb5.conf”文件。

----结束

准备运行环境配置文件

应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。

用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。
  a. 登录FusionInsight Manager页面，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“完整客户端”，根据待安装客户端节点

的节点类型选择正确的平台类型后单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig.tar”，继续解压该文件。

b. 进入解压后的“FusionInsight_Cluster_1_Services_ClientConfig”文件夹，获取表23-4中配置文件。

表 23-4 配置文件

获取路径	配置文件	作用
Yarn\config	core-site.xml	配置Hadoop Core详细参数。
	hbase-site.xml	配置HBase详细参数。
	hdfs-site.xml	配置HDFS详细参数。
	mapred-site.xml	Hadoop MapReduce配置文件。
	yarn-site.xml	配置Yarn详细参数。
HBase\config	hbase-site.xml	配置HBase详细参数。
Hive\config	hive-site.xml	配置Hive详细参数。
	hiveclient.properties	配置Hive详细参数。

说明

如果不运行MapReduce访问多组件样例程序，则不需要获取HBase和Hive的hbase-site.xml、hive-site.xml、hiveclient.properties配置文件。

c. 复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与解压目录下“hosts”文件中所列出的各主机在网络上互通。

说明

- 当客户端所在主机不是集群中的节点时，配置客户端网络连接，可避免执行客户端命令时出现错误。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。

  a. 在节点中安装客户端。

  如果需要使用MapReduce访问多组件样例程序，请确保集群已安装Hive、HBase服务。

  例如客户端安装目录为“/opt/client”。

  b. 登录FusionInsight Manager页面，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。

c. 以**root**用户登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”）解压软件包。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”，则执行以下命令进行解压：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar

cd /tmp/FusionInsight-Client/FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles

d. 参考表23-4将相关配置文件上传到“conf”目录下（后续编译出的jar包也需要放置在此目录），用于后续调测。例如“/opt/hadoopclient/conf”，该目录需要提前在客户端节点上创建。

例如，上传Yarn客户端的“core-site.xml”文件则执行以下命令：

scp Yarn/config/core-site.xml root@*客户端节点IP地址*:/opt/hadoopclient/conf

参考以上命令依次上传表23-4中的所有配置文件。

e. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

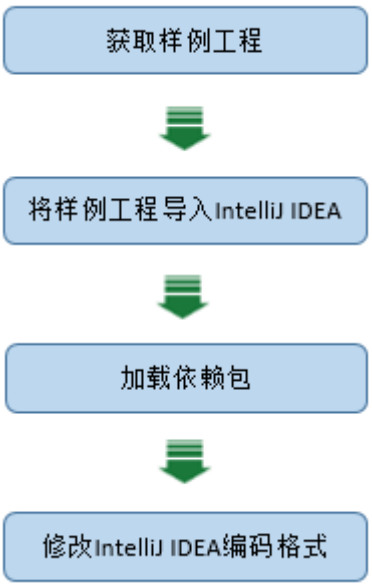
23.4.3 导入并配置 MapReduce 样例工程

操作场景

MapReduce针对多个场景提供样例工程，帮助客户快速学习MapReduce工程。

以下操作步骤以导入MapReduce样例代码为例。操作流程如图23-2所示。

图 23-2 导入样例工程流程



前提条件

- 确保本地环境的时间与MRS集群的时间差要小于5分钟，若无法确定，请联系系统管理员。MRS集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备连接MapReduce集群配置文件**。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程，可根据实际业务场景选择对应的样例，相关样例介绍请参见**MapReduce样例工程介绍**。

步骤2 将已获取的认证文件和配置文件，放置在MapReduce样例工程的“../src/mapreduce-example-security/conf”路径下。

说明

不使用访问多组件样例程序时，如果不影响统计样例程序的正常编译，可忽略多组件样例程序相关报错信息，否则请在导入样例工程后将多组件样例程序类文件删除。

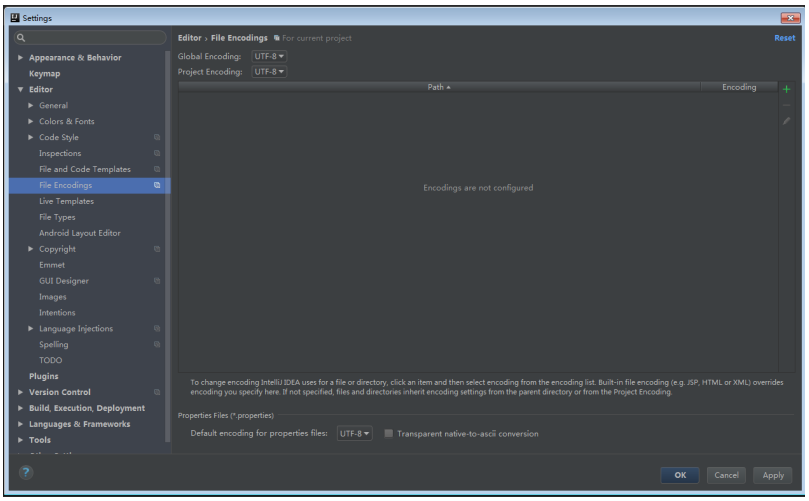
步骤3 导入样例工程到IntelliJ IDEA开发环境。

1. 打开IntelliJ IDEA，依次选择“File > Open”。
2. 在弹出的Open File or Project会话框中选择样例工程文件夹“mapreduce-example-security”，单击“OK”。

步骤4 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File> Settings”。
2. 在弹出“Settings”窗口左边导航上选择“Editor > File Encodings”，在“Global Encoding”和“Project Encodings”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如**图23-3**所示。

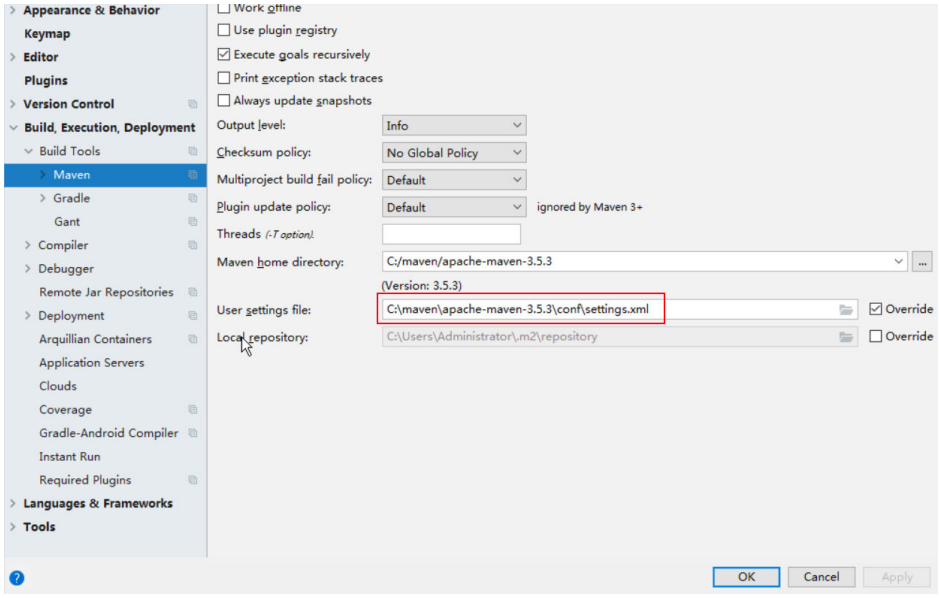
图 23-3 设置 IntelliJ IDEA 的编码格式



步骤5 参考**配置华为开源镜像仓**章节描述，增加开源镜像仓地址等配置信息到本地Maven的“setting.xml”配置文件。

修改完成后，使用IntelliJ IDEA开发工具时，可选择“File > Settings > Build, Execution, Deployment > Build Tools > Maven”查看当前“settings.xml”文件放置目录，确保该目录为“*<本地Maven安装目录>*\conf\settings.xml”。

图 23-4 “settings.xml”文件放置目录



----结束

23.4.4 （可选）创建 MapReduce 样例工程

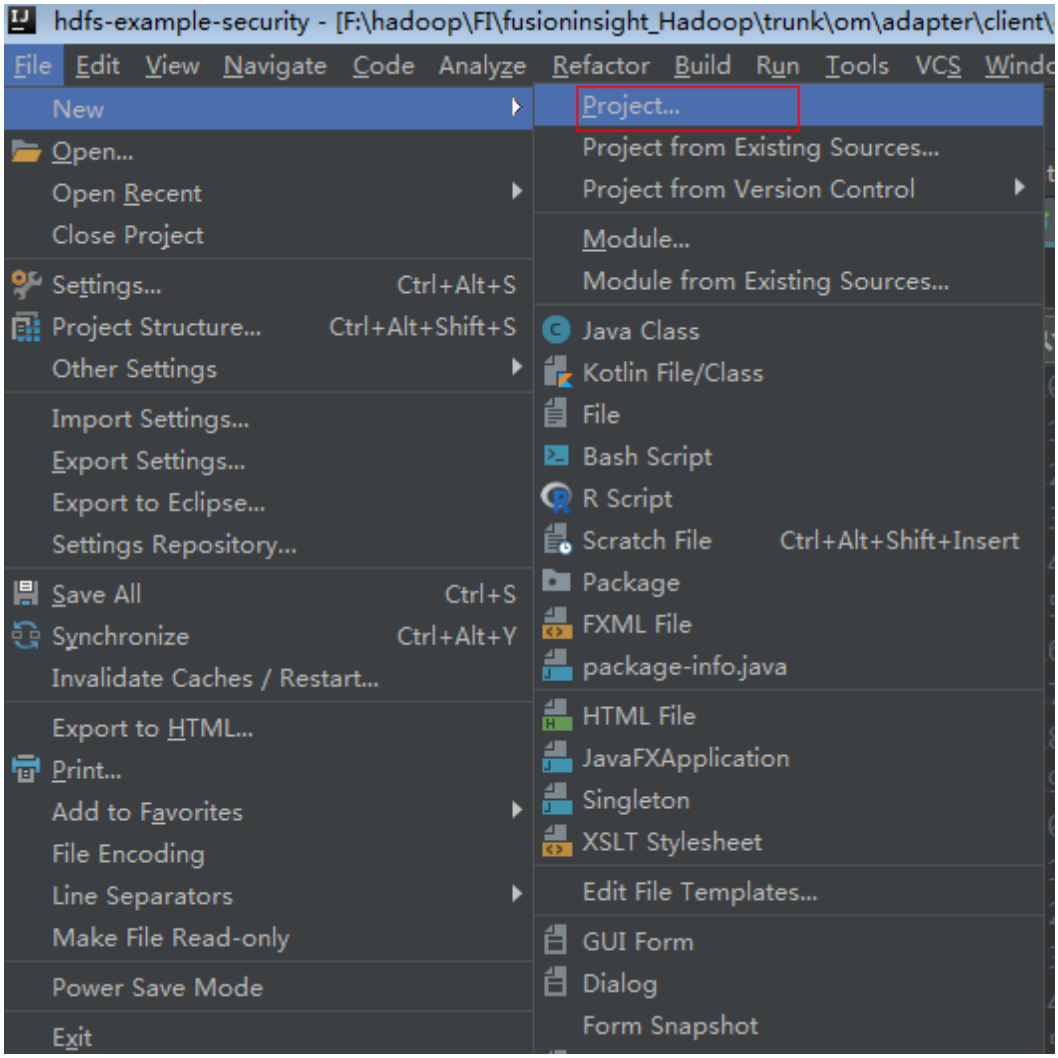
操作场景

除了导入MapReduce样例工程，您还可以使用IntelliJ IDEA新建一个MapReduce工程。

操作步骤

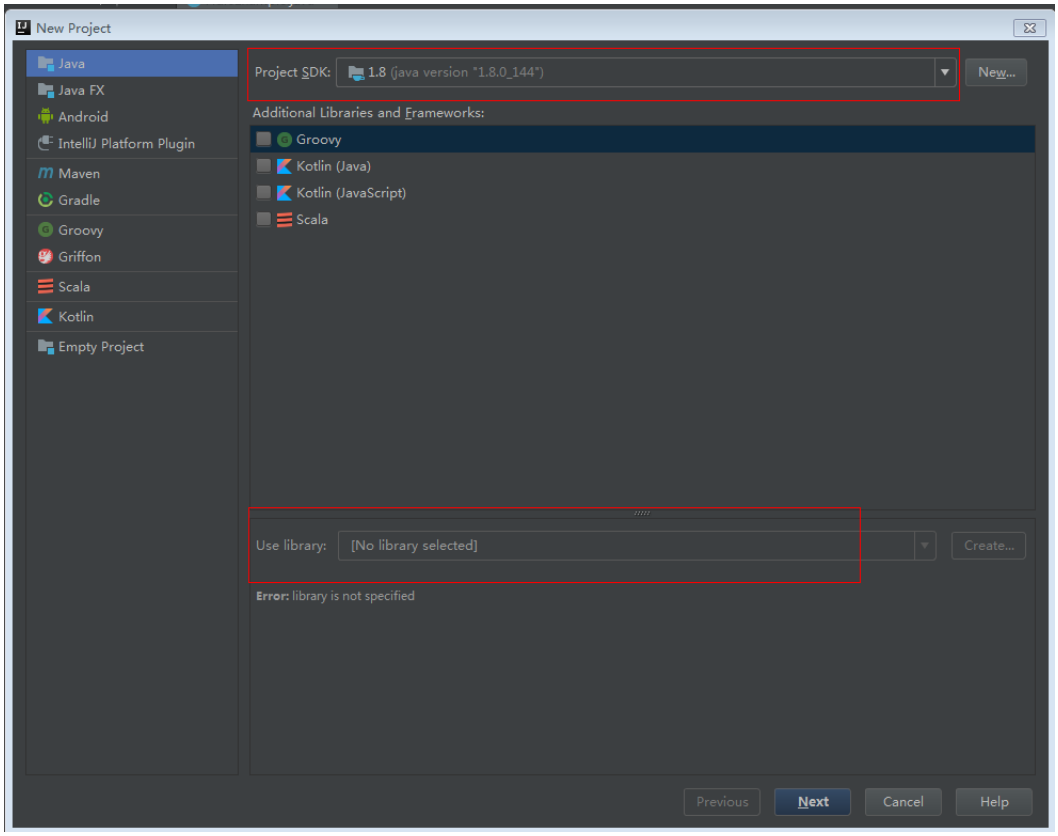
步骤1 打开IntelliJ IDEA工具，选择“File > New > Project”，如图23-5所示。

图 23-5 创建工程



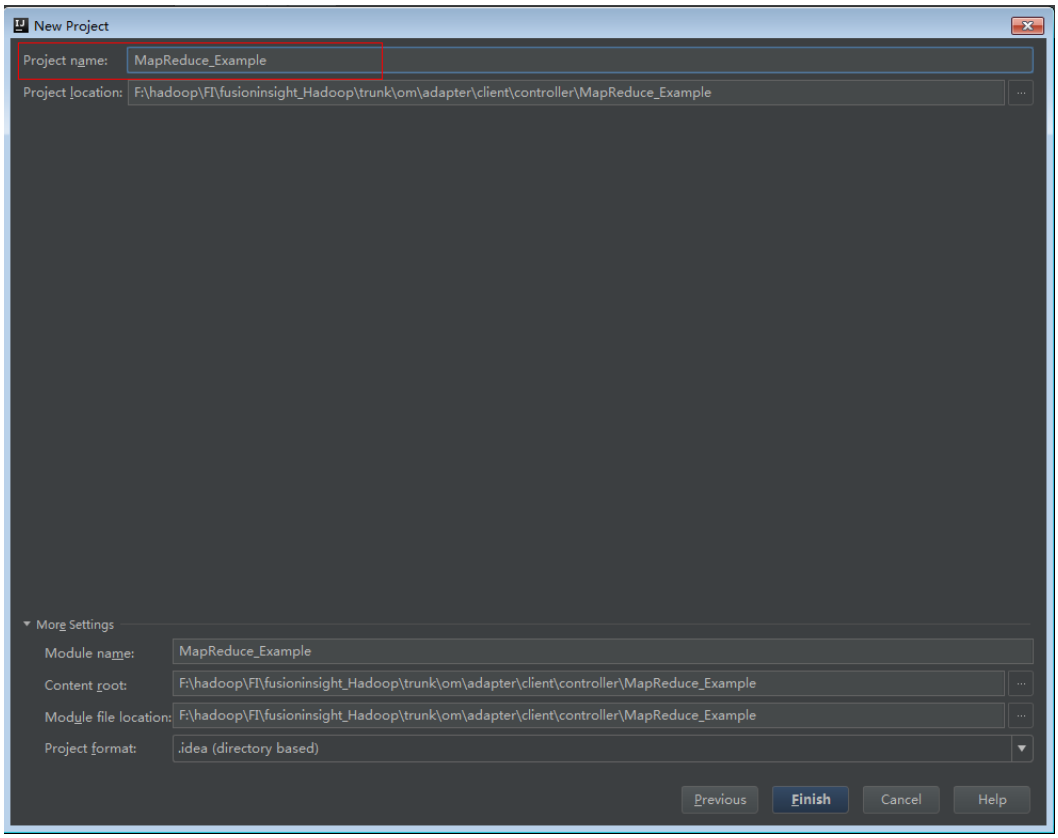
步骤2 在“New Project”页面选择“Java”，然后配置工程需要的JDK和其他Java库。如下图所示。配置完成后单击“Next”。

图 23-6 配置工程所需 SDK 信息



步骤3 在会话框中填写新建的工程名称。然后单击Finish完成创建。

图 23-7 填写工程名称



----结束

23.4.5 配置 MapReduce 应用安全认证

场景说明

在安全集群环境下，各个组件之间的相互通信不能够简单的互通，而需要在通信之前进行相互认证，以确保通信的安全性。

用户在提交MapReduce应用程序时，需要与Yarn、HDFS等之间进行通信。那么提交MapReduce的应用程序中需要写入安全认证代码，确保MapReduce程序能够正常运行。

安全认证有两种方式：

- 命令行认证：
  提交MapReduce应用程序运行前，在MapReduce客户端执行如下命令获得认证。
  ***kinit** 组件业务用户*
- 代码认证：
  通过获取客户端的principal和keytab文件在应用程序中进行认证。

MapReduce 的安全认证代码

目前是统一调用LoginUtil类进行安全认证。

在MapReduce样例工程的“com.huawei.bigdata.mapreduce.examples”包的“FemaleInfoCollector”类的代码中，test@*<系统域名>*、user.keytab和krb5.conf为

示例，实际操作时需要已将相应账号对应权限的keytab文件和krb5.conf文件放入到“conf”目录，安全登录方法如下代码所示。

```
public static final String PRINCIPAL= "test@<系统域名>";
public static final String KEYTAB =
FemaleInfoCollector.class.getClassLoader().getResource("user.keytab").getPath();
public static final String KRB =
FemaleInfoCollector.class.getClassLoader().getResource("krb5.conf").getPath();
...
// Security login
LoginUtil.login(PRINCIPAL, KEYTAB, KRB, conf);
```

说明

- test：为**准备集群认证用户信息**创建的用户名称，例如**developuser**。
- 系统域名：登录FusionInsight Manager后，选择“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。

23.5 开发 MapReduce 应用

23.5.1 MapReduce 统计样例程序

23.5.1.1 MapReduce 统计样例程序开发思路

场景说明

假定用户有某个周末网民网购停留时间的日志文本，基于某些业务要求，要求开发MapReduce应用程序实现如下功能：

- 统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。
- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

log1.txt：周六网民停留日志

```
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

log2.txt：周日网民停留日志

```
LiuYang,female,20
YuanJing,male,10
CaiXuyu,female,50
FangBo,female,50
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
CaiXuyu,female,50
FangBo,female,50
LiuYang,female,20
```

```
YuanJing,male,10
FangBo,female,50
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

- 读取原文件数据。
- 筛选女性网民上网时间数据信息。
- 汇总每个女性上网总时间。
- 筛选出停留总时间大于2个小时的女性网民信息。

23.5.1.2 MapReduce 统计样例代码

功能介绍

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为三个部分：

- 从原文件中筛选女性网民上网时间数据信息，通过类CollectionMapper继承Mapper抽象类实现。
- 汇总每个女性上网时间，并输出时间大于两个小时的女性网民信息，通过类CollectionReducer继承Reducer抽象类实现。
- main方法提供建立一个MapReduce job，并提交MapReduce作业到hadoop集群。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.mapreduce.examples.FemaleInfoCollector类：

样例1：类CollectionMapper定义Mapper抽象类的map()方法和setup()方法。

```
public static class CollectionMapper extends
        Mapper<Object, Text, Text, IntWritable> {

// 分隔符。
String delim;
// 性别筛选。
String sexFilter;

// 姓名信息。
private Text nameInfo = new Text();

// 输出的key,value要求是序列化的。
private IntWritable timeInfo = new IntWritable(1);

 /**
 * 分布式计算
 *
 * @param key Object : 原文件位置偏移量。
 * @param value Text : 原文件的一行字符数据。
 * @param context Context : 出参。
```



```
   * @throws IOException , InterruptedException
   */
      public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException
      {

         String line = value.toString();

         if (line.contains(sexFilter))
         {

               // 读取的一行字符串数据。
                 String name = line.substring(0, line.indexOf(delim));
                 nameInfo.set(name);
               // 获取上网停留时间。
                 String time = line.substring(line.lastIndexOf(delim) + 1,
                           line.length());
                  timeInfo.set(Integer.parseInt(time));

               // map输出key，value键值对。
            context.write(nameInfo, timeInfo);
         }
      }

      /**
   * map调用，做一些初始工作。
   *
   * @param context Context
   */
      public void setup(Context context) throws IOException,
            InterruptedException
  {

               // 通过Context可以获得配置信息。
     delim = context.getConfiguration().get("log.delimiter", ",");

         sexFilter = delim
                     + context.getConfiguration()
                           .get("log.sex.filter", "female") + delim;
      }
  }
```

样例2：类CollectionReducer定义Reducer抽象类的reduce()方法。

```
   public static class CollectionReducer extends
         Reducer<Text, IntWritable, Text, IntWritable>
{

   // 统计结果。
      private IntWritable result = new IntWritable();

      // 总时间门槛。
      private int timeThreshold;

    /**
     * @param key Text : Mapper后的key项。
     * @param values Iterable : 相同key项的所有统计结果。
     * @param context Context
     * @throws IOException , InterruptedException
     */
      public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException
 {
         int sum = 0;
         for (IntWritable val : values) {
            sum += val.get();
         }

         // 如果时间小于门槛时间，不输出结果。
         if (sum < timeThreshold)
```



```
    {
            return;
        }
        result.set(sum);

        // reduce输出为key：网民的信息，value：该网民上网总时间。
        context.write(key, result);
    }

    /**
     * setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次。
 *
 * @param context Context
 * @throws IOException , InterruptedException
 */
    public void setup(Context context) throws IOException,
            InterruptedException
{

        // Context可以获得配置信息。
        timeThreshold = context.getConfiguration().getInt(
                "log.time.threshold", 120);
    }
}
```

样例3：main()方法创建一个job，指定参数，提交作业到hadoop集群。

```
public static void main(String[] args) throws Exception {
  // 初始化环境变量。
  Configuration conf = new Configuration();

  // 安全登录。
  LoginUtil.login(PRINCIPAL, KEYTAB, KRB, conf);

  // 获取入参。
  String[] otherArgs = new GenericOptionsParser(conf, args)
      .getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: collect female info <in> <out>");
    System.exit(2);
  }

  // 初始化Job任务对象。
  @SuppressWarnings("deprecation")
  Job job = new Job(conf, "Collect Female Info");
  job.setJarByClass(FemaleInfoCollector.class);

  // 设置运行时执行map，reduce的类，也可以通过配置文件指定。
  job.setMapperClass(CollectionMapper.class);
  job.setReducerClass(CollectionReducer.class);

  // 设置combiner类，默认不使用，使用时通常使用和reduce一样的类。
  // Combiner类需要谨慎使用，也可以通过配置文件指定。
  job.setCombinerClass(CollectionReducer.class);

  // 设置作业的输出类型。
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

  // 提交任务交到远程环境上执行。
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

样例4：类CollectionCombiner实现了在map端先合并map输出的数据，减少map和reduce之间传输的数据量。

```
/**
 * Combiner class
```



```
 */
public static class CollectionCombiner extends
Reducer<Text, IntWritable, Text, IntWritable> {

// Intermediate statistical results
private IntWritable intermediateResult = new IntWritable();

/**
 * @param key      Text : key after Mapper
 * @param values  Iterable : all results with the same key in this map task
 * @param context Context
 * @throws IOException , InterruptedException
 */
public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values) {
sum += val.get();
}

intermediateResult.set(sum);

// In the output information, key indicates netizen information,
// and value indicates the total online time of the netizen in this map task.
context.write(key, intermediateResult);
}

}
```

23.5.2 MapReduce 访问多组件样例程序

23.5.2.1 MapReduce 访问多组件样例程序开发思路

场景说明

该样例以MapReduce访问HDFS、HBase、Hive为例，介绍如何编写MapReduce作业访问多个服务组件。帮助用户理解认证、配置加载等关键使用方式。

该样例逻辑过程如下：

以HDFS文本文件为输入数据：

log1.txt：数据输入文件

```
YuanJing,male,10
GuoYijun,male,5
```

Map阶段：

1. 获取输入数据的一行并提取姓名信息。
2. 查询HBase一条数据。
3. 查询Hive一条数据。
4. 将HBase查询结果与Hive查询结果进行拼接作为Map输出。

Reduce阶段：

1. 获取Map输出中的最后一条数据。
2. 将数据输出到HBase。
3. 将数据保存到HDFS。

23.5.2.2 MapReduce 访问多组件样例代码

功能介绍

主要分为三个部分：

- 从HDFS原文件中抽取name信息，查询HBase、Hive相关数据，并进行数据拼接，通过类MultiComponentMapper继承Mapper抽象类实现。
- 获取拼接后的数据取最后一条输出到HBase、HDFS，通过类MultiComponentReducer继承Reducer抽象类实现。
- main方法提供建立一个MapReduce job，并提交MapReduce作业到Hadoop集群。

代码样例

下面代码片段仅为演示，具体代码请参见com.huawei.bigdata.mapreduce.examples.MultiComponentExample类：

样例1：类MultiComponentMapper定义Mapper抽象类的map方法。

```
private static class MultiComponentMapper extends Mapper<Object, Text, Text, Text> {

   Configuration conf;

   @Override protected void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

     conf = context.getConfiguration();

     // 对于需要访问ZooKeeper的组件，需要提供jaas和krb5配置
     // 在Map中不需要重复login，会使用main方法中配置的鉴权信息
     String krb5 = "krb5.conf";
     String jaas = "jaas_mr.conf";
     // 这些文件上传自main方法
     File jaasFile = new File(jaas);
     File krb5File = new File(krb5);
     System.setProperty("java.security.auth.login.config", jaasFile.getCanonicalPath());
     System.setProperty("java.security.krb5.conf", krb5File.getCanonicalPath());
     System.setProperty("zookeeper.sasl.client", "true");

     LOG.info("UGI :" + UserGroupInformation.getCurrentUser());

     String name = "";
     String line = value.toString();
     if (line.contains("male")) {
       name = line.substring(0, line.indexOf(","));
     }
     // 1. 读取HBase数据
     String hbaseData = readHBase();

     // 2. 读取Hive数据
     String hiveData = readHive(name);

     // Map输出键值对，内容为HBase与Hive数据拼接的字符串
     context.write(new Text(name), new Text("hbase:" + hbaseData + ", hive:" + hiveData));
   }
```

样例2：HBase数据读取的readHBase方法。

```
   private String readHBase() {
     String tableName = "table1";
     String columnFamily = "cf";
     String hbaseKey = "1";
     String hbaseValue;
```

```
    Configuration hbaseConfig = HBaseConfiguration.create(conf);
    org.apache.hadoop.hbase.client.Connection conn = null;
    try {
      // 建立HBase连接
      conn = ConnectionFactory.createConnection(hbaseConfig);
      // 获取HBase表
      Table table = conn.getTable(TableName.valueOf(tableName));
      // 创建一个HBase Get请求实例
      Get get = new Get(hbaseKey.getBytes());
      // 提交Get请求
      Result result = table.get(get);
      hbaseValue = Bytes.toString(result.getValue(columnFamily.getBytes(), "cid".getBytes()));

      return hbaseValue;

    } catch (IOException e) {
      LOG.warn("Exception occur ", e);
    } finally {
        if (conn != null) {
                try {
                    conn.close();
                } catch (Exception e1) {
                    LOG.error("Failed to close the connection ", e1);
                }
              }
            }
    return "";
  }
```

样例3：Hive数据读取的readHive方法。

```
  private String readHive(String name) throws IOException {
    //加载配置文件
    Properties clientInfo = null;
    String userdir = System.getProperty("user.dir") + "/";
    InputStream fileInputStream = null;
    try {
      clientInfo = new Properties();
      String hiveclientProp = userdir + "hiveclient.properties";
      File propertiesFile = new File(hiveclientProp);
      fileInputStream = new FileInputStream(propertiesFile);
      clientInfo.load(fileInputStream);
    } catch (Exception e) {
      throw new IOException(e);
    } finally {
      if (fileInputStream != null) {
        fileInputStream.close();
      }
    }
    String zkQuorum = clientInfo.getProperty("zk.quorum");
    String zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    String serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");

    // 请仔细阅读此内容:
    // MapReduce任务通过JDBC方式访问Hive
    // Hive会将sql查询封装成另一个MapReduce任务并提交
    // 所以不建议在MapReduce作业中调用Hive
    final String driver = "org.apache.hive.jdbc.HiveDriver";

    String sql = "select name,sum(stayTime) as " + "stayTime from person where name = '" + name + "'
group by name";

    StringBuilder sBuilder = new StringBuilder("jdbc:hive2://").append(zkQuorum).append("/");
    // 在map或reduce中，Hive连接方式使用'auth=delegationToken'
    sBuilder
        .append(";serviceDiscoveryMode=")
        .append(serviceDiscoveryMode)
        .append(";zooKeeperNamespace=")
        .append(zooKeeperNamespace)
```



```
        .append(";auth=delegationToken;");
    String url = sBuilder.toString();
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
      Class.forName(driver);
      connection = DriverManager.getConnection(url, "", "");
      statement = connection.prepareStatement(sql);
      resultSet = statement.executeQuery();

      if (resultSet.next()) {
        return resultSet.getString(1);
      }
    } catch (ClassNotFoundException e) {
      LOG.warn("Exception occur ", e);
    } catch (SQLException e) {
      LOG.warn("Exception occur ", e);
    } finally {
        if (null != resultSet) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    // handle exception
                }
            }
            if (null != statement) {
                try {
                    statement.close();
                } catch (SQLException e) {
                    // handle exception
                }
            }
            if (null != connection) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    // handle exception
                }
            }
    }

    return "";
  }
```

样例4：类MultiComponentReducer定义Reducer抽象类的reduce方法。

```
 private static class MultiComponentReducer extends Reducer<Text, Text, Text, Text> {
   Configuration conf;

   public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
InterruptedException {
     conf = context.getConfiguration();

     // 对于需要访问ZooKeeper的组件，需要提供jaas和krb5配置
     // 在Reduce中不需要重复login，会使用main方法中配置的鉴权信息
     String krb5 = "krb5.conf";
     String jaas = "jaas_mr.conf";
     // 这些文件上传自main方法
     File jaasFile = new File(jaas);
     File krb5File = new File(krb5);
     System.setProperty("java.security.auth.login.config", jaasFile.getCanonicalPath());
     System.setProperty("java.security.krb5.conf", krb5File.getCanonicalPath());
     System.setProperty("zookeeper.sasl.client", "true");

     Text finalValue = new Text("");

     for (Text value : values) {
       finalValue = value;
     }
```



```
    // 将结果输出到HBase
    writeHBase(key.toString(), finalValue.toString());

    // 将结果保存到HDFS
    context.write(key, finalValue);
  }
```

样例5：结果输出到HBase的writeHBase方法。

```
  private void writeHBase(String rowKey, String data) {
    String tableName = "table1";
    String columnFamily = "cf";

    try {
      LOG.info("UGI read :" + UserGroupInformation.getCurrentUser());
    } catch (IOException e1) {
      // handler exception
    }

    Configuration hbaseConfig = HBaseConfiguration.create(conf);
    org.apache.hadoop.hbase.client.Connection conn = null;
    try {
      // 创建HBase连接
      conn = ConnectionFactory.createConnection(hbaseConfig);
      // 获取HBase表
      Table table = conn.getTable(TableName.valueOf(tableName));

      // 创建一个HBase Put请求实例
      List<Put> list = new ArrayList<Put>();
      byte[] row = Bytes.toBytes("row" + rowKey);
      Put put = new Put(row);
      byte[] family = Bytes.toBytes(columnFamily);
      byte[] qualifier = Bytes.toBytes("value");
      byte[] value = Bytes.toBytes(data);
      put.addColumn(family, qualifier, value);
      list.add(put);
      // 执行Put请求
      table.put(list);
    } catch (IOException e) {
      LOG.warn("Exception occur ", e);
    } finally {

            if (conn != null) {
                try {
                    conn.close();
                } catch (Exception e1) {
                    LOG.error("Failed to close the connection ", e1);
                }
            }
    }

  }
```

样例6：main()方法创建一个job，配置相关依赖，配置相关鉴权信息，提交作业到hadoop集群。

```
  public static void main(String[] args) throws Exception {
    //加载hiveclient.properties配置文件
      Properties clientInfo = null;
      try {
          clientInfo = new Properties();

clientInfo.load(MultiComponentExample.class.getClassLoader().getResourceAsStream("hiveclient.properties")
);
      } catch (Exception e) {
          throw new IOException(e);
      } finally {

      }
```

```
    String zkQuorum = clientInfo.getProperty("zk.quorum");
    String zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    String serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
    String principal = clientInfo.getProperty("principal");
    String auth = clientInfo.getProperty("auth");
    String sasl_qop = clientInfo.getProperty("sasl.qop");
    String hbaseKeytab =
MultiComponentExample.class.getClassLoader().getResource("user.keytab").getPath();
    String hbaseJaas =
MultiComponentExample.class.getClassLoader().getResource("jaas_mr.conf").getPath();
    String hiveClientProperties =
MultiComponentExample.class.getClassLoader().getResource("hiveclient.properties").getPath();
    // 拼接文件列表，以逗号分隔
    String files = "file://" + KEYTAB + "," + "file://" + KRB + "," + "file://" + JAAS;
    files = files + "," + "file://" + hbaseKeytab;
    files = files + "," + "file://" + hbaseJaas;
    files = files + "," + "file://" + hiveClientProperties;
    // tmpfiles属性所涉及文件将会在Job提交时上传到HDFS
    config.set("tmpfiles", files);

    // 清理所需目录
    MultiComponentExample.cleanupBeforeRun();

    // 安全集群login
    LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, PRINCIPAL, hbaseKeytab);
    LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);
    LoginUtil.login(PRINCIPAL, KEYTAB, KRB, config);

    // 查找Hive依赖jar包
    Class hiveDriverClass = Class.forName("org.apache.hive.jdbc.HiveDriver");
    Class thriftClass = Class.forName("org.apache.thrift.TException");
    Class serviceThriftCLIClass = Class.forName("org.apache.hive.service.rpc.thrift.TCLIService");
    Class hiveConfClass = Class.forName("org.apache.hadoop.hive.conf.HiveConf");
    Class hiveTransClass = Class.forName("org.apache.thrift.transport.HiveTSaslServerTransport");
    Class hiveMetaClass = Class.forName("org.apache.hadoop.hive.metastore.api.MetaException");
    Class hiveShimClass =
Class.forName("org.apache.hadoop.hive.metastore.security.HadoopThriftAuthBridge23");
    Class thriftCLIClass = Class.forName("org.apache.hive.service.cli.thrift.ThriftCLIService");
    // 添加Hive运行依赖到Job
    JarFinderUtil.addDependencyJars(config, hiveDriverClass, serviceThriftCLIClass, thriftCLIClass, thriftClass,
            hiveConfClass, hiveTransClass, hiveMetaClass, hiveShimClass);
    // 添加Hive配置文件
    config.addResource("hive-site.xml");
    // 添加HBase配置文件
    Configuration conf = HBaseConfiguration.create(config);

    // 实例化Job
    Job job = Job.getInstance(conf);
    job.setJarByClass(MultiComponentExample.class);

    // 设置mapper&reducer类
    job.setMapperClass(MultiComponentMapper.class);
    job.setReducerClass(MultiComponentReducer.class);

    //设置Job输入输出路径
    FileInputFormat.addInputPath(job, new Path(baseDir, INPUT_DIR_NAME + File.separator + "data.txt"));
    FileOutputFormat.setOutputPath(job, new Path(baseDir, OUTPUT_DIR_NAME));

    // 设置输出键值类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

    // HBase提供工具类添加HBase运行依赖到Job
    TableMapReduceUtil.addDependencyJars(job);

    // 安全模式下必须要执行这个操作
    // HBase添加鉴权信息到Job，map或reduce任务将会使用此处的鉴权信息
    TableMapReduceUtil.initCredentials(job);
```

```
    // 创建Hive鉴权信息

    StringBuilder sBuilder = new StringBuilder("jdbc:hive2://").append(zkQuorum).append("/");

sBuilder.append(";serviceDiscoveryMode=").append(serviceDiscoveryMode).append(";zooKeeperNamespace=
")
        .append(zooKeeperNamespace)
        .append(";sasl.qop=")
        .append(sasl_qop)
        .append(";auth=")
        .append(auth)
        .append(";principal=")
        .append(principal)
        .append(";");
    String url = sBuilder.toString();
    Connection connection = DriverManager.getConnection(url, "", "");
    String tokenStr = ((HiveConnection) connection)
        .getDelegationToken(UserGroupInformation.getCurrentUser().getShortUserName(), PRINCIPAL);
    connection.close();
    Token<DelegationTokenIdentifier> hive2Token = new Token<DelegationTokenIdentifier>();
    hive2Token.decodeFromUrlString(tokenStr);
    // 添加Hive鉴权信息到Job
    job.getCredentials().addToken(new Text("hive.server2.delegation.token"), hive2Token);
    job.getCredentials().addToken(new Text(HiveAuthConstants.HS2_CLIENT_TOKEN), hive2Token);

    // 提交作业
    System.exit(job.waitForCompletion(true) ? 0 : 1);

  }
```

说明

样例中所有zkQuorum对象需替换为实际ZooKeeper集群节点信息。

23.6 调测 MapReduce 应用

23.6.1 准备 MapReduce 样例初始数据

操作场景

在调测程序之前，需要提前准备将待处理的数据。

- 运行MapReduce统计样例程序，请参考**规划MapReduce统计样例程序数据**。
- 运行MapReduce访问多组件样例程序，请参考**规划MapReduce访问多组件样例程序数据**。

规划 MapReduce 统计样例程序数据

将待处理的日志文件放置在HDFS系统中。

步骤1 在Linux系统中新建文本文件，将待处理的数据复制到文件中。例如将**MapReduce统计样例程序开发思路**中log1.txt中的内容复制保存到input_data1.txt，将log2.txt中的内容复制保存到input_data2.txt。

步骤2 在HDFS上建立一个文件夹“/tmp/input”，并上传input_data1.txt，input_data2.txt到此目录，操作如下：

1. 执行以下命令进入HDFS客户端目录并认证用户。

   cd *HDFS客户端安装目录*

source bigdata_env

kinit *组件业务用户*（该用户需要具有操作HDFS的权限，首次认证需要修改密码）

2. 执行以下命令创建“/tmp/input”目录。

   hdfs dfs -mkdir /tmp/input

3. 执行以下命令将已准备好的文件上传至HDFS客户端的“/tmp/input”目录下。

   hdfs dfs -put *local_filepath*/**input_data1.txt /tmp/input**

   hdfs dfs -put *local_filepath*/**input_data2.txt /tmp/input**

----结束

规划 MapReduce 访问多组件样例程序数据

步骤1 创建HDFS数据文件。

1. 在Linux系统中新建文本文件，将待处理的数据复制到文件中。例如将**MapReduce访问多组件样例程序开发思路**中log1.txt中的内容复制保存到data.txt。
2. 执行以下命令进入HDFS客户端目录并认证用户。

   cd *HDFS客户端安装目录*

   source bigdata_env

   kinit *组件业务用户*（该用户需要具有操作HDFS的权限，首次认证需要修改密码）

3. 在HDFS上创建一个文件夹“/tmp/examples/multi-components/mapreduce/input/”，并上传data.txt到此目录，操作如下：

   a. 在HDFS客户端使用以下命令创建目录。

      hdfs dfs -mkdir -p /tmp/examples/multi-components/mapreduce/input/

   b. 执行以下命令上传文件至HDFS。

      hdfs dfs -put *local_filepath*/**data.txt /tmp/examples/multi-components/mapreduce/input/**

步骤2 创建HBase表并插入数据。

1. 执行以下命令进入HBase客户端。

   cd *HBase客户端安装目录*

   source bigdata_env

   kinit *组件业务用户*

   hbase shell

2. 执行以下命令在HBase shell交互窗口创建数据表table1，该表有一个列族cf。

   create 'table1', 'cf'

3. 执行以下命令插入一条rowkey为1、列名为cid、数据值为123的数据。

   put 'table1', '1', 'cf:cid', '123'

4. 执行以下命令退出HBase客户端。

   quit

步骤3 创建Hive表并载入数据。

1. 使用以下命令进入Hive客户端。

   cd *Hive客户端安装目录*

   source bigdata_env

   kinit *组件业务用户*

   beeline

2. 执行以下命令在Hive beeline交互窗口创建数据表person，该表有3个字段：name/gender/stayTime。

   CREATE TABLE person(name STRING, gender STRING, stayTime INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' stored as textfile;

3. 执行以下命令在Hive beeline交互窗口加载数据文件。

   LOAD DATA INPATH '/tmp/examples/multi-components/mapreduce/input/' OVERWRITE INTO TABLE person;

4. 执行命令**!q**退出。

步骤4 由于Hive加载数据将HDFS对应数据目录清空，所以需再次执行**步骤1**。

----结束

23.6.2 在本地 Windows 环境中调测 MapReduce 应用

操作场景

在程序代码完成开发后，您可以在Windows环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

MapReduce应用程序运行完成后，可通过如下方式查看应用程序的运行情况。

- 在IntelliJ IDEA中查看应用程序运行情况。
- 通过MapReduce日志获取应用程序运行情况。
- 登录MapReduce WebUI查看应用程序运行情况。
- 登录Yarn WebUI查看应用程序运行情况。

说明

- 如果Windows运行环境中使用IBM JDK，不支持在Windows环境中直接运行应用程序。
- 在MapReduce任务运行过程中禁止重启HDFS服务，否则可能会导致任务失败。

前提条件

- 已将**准备连接MapReduce集群配置文件**获取的配置文件放置到MapReduce样例工程的“../src/mapreduce-example-security/conf”路径下。
- 已参考**规划MapReduce统计样例程序数据**将待处理数据上传至HDFS。

运行统计样例程序

步骤1 确保样例工程依赖的所有jar包已正常获取。

步骤2 在IntelliJ IDEA开发环境中，打开样例工程中“LocalRunner.java”工程，右键工程，选择“Run > LocalRunner.main()”运行应用工程。

----结束

运行多组件样例程序

步骤1 在放置MapReduce工程样例的“../src/mapreduce-example-security/conf”目录下创建“jaas_mr.conf”文件并添加如下内容：

```
Client {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
keyTab="user.keytab"
principal="test@<系统域名>"
useTicketCache=false
storeKey=true
debug=true;
};
```

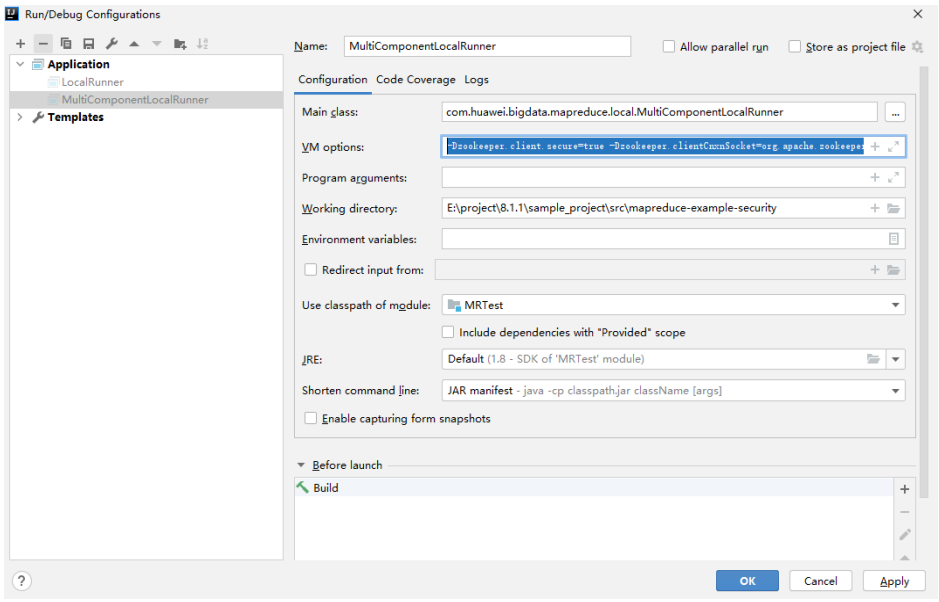
说明

- test：为**准备集群认证用户信息**创建的用户名称，例如**developuser**。
- 系统域名：登录FusionInsight Manager后，选择“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。

步骤2 确保样例工程依赖的所有Hive、HBase相关jar包已正常获取。

步骤3 在IntelliJ IDEA开发环境中，选中“MultiComponentLocalRunner.java”工程，单击运行对应的应用程序工程。或者右键工程，选择“Run MultiComponentLocalRunner.main()”运行应用工程。

如果集群开启了ZooKeeper SSL，则运行该样例前需要在下图所示位置增加“-Dzookeeper.client.secure=true -Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty”参数。



----结束

查看调测结果

- **查看运行结果获取应用运行情况**

  如下所示，通过控制台输出结果查看应用运行情况。

  ```
  1848 [main] INFO  org.apache.hadoop.security.UserGroupInformation  - Login successful for user admin@<系统域名> using keytab file
  ```

```
Login success!!!!!!!!!!!!!!
7093 [main] INFO  org.apache.hadoop.hdfs.PeerCache  - SocketCache disabled.
9614 [main] INFO  org.apache.hadoop.hdfs.DFSClient  - Created HDFS_DELEGATION_TOKEN token 45
for admin on ha-hdfs:hacluster
9709 [main] INFO  org.apache.hadoop.mapreduce.security.TokenCache  - Got dt for hdfs://hacluster;
Kind: HDFS_DELEGATION_TOKEN,
Service: ha-hdfs:hacluster, Ident:
(HDFS_DELEGATION_TOKEN token 45 for admin)
10914 [main] INFO  org.apache.hadoop.yarn.client.ConfiguredRMFailoverProxyProvider  - Failing over
to 53
12136 [main] INFO  org.apache.hadoop.mapreduce.lib.input.FileInputFormat  - Total input files to
process : 2
12731 [main] INFO  org.apache.hadoop.mapreduce.JobSubmitter  - number of splits:2
13405 [main] INFO  org.apache.hadoop.mapreduce.JobSubmitter  - Submitting tokens for job:
job_1456738266914_0006
13405 [main] INFO  org.apache.hadoop.mapreduce.JobSubmitter  - Kind: HDFS_DELEGATION_TOKEN,
Service: ha-hdfs:hacluster,
Ident: (HDFS_DELEGATION_TOKEN token 45 for admin)
16019 [main] INFO  org.apache.hadoop.yarn.client.api.impl.YarnClientImpl  - Application submission is
not finished,
submitted application application_1456738266914_0006 is still in NEW
16975 [main] INFO  org.apache.hadoop.yarn.client.api.impl.YarnClientImpl  - Submitted application
application_1456738266914_0006
17069 [main] INFO  org.apache.hadoop.mapreduce.Job  - The url to track the job:
https://linux2:8090/proxy/application_1456738266914_0006/
17086 [main] INFO  org.apache.hadoop.mapreduce.Job  - Running job: job_1456738266914_0006
29811 [main] INFO  org.apache.hadoop.mapreduce.Job  - Job job_1456738266914_0006 running in
uber mode : false
29811 [main] INFO  org.apache.hadoop.mapreduce.Job  -  map 0% reduce 0%
41492 [main] INFO  org.apache.hadoop.mapreduce.Job  -  map 100% reduce 0%
53161 [main] INFO  org.apache.hadoop.mapreduce.Job  -  map 100% reduce 100%
53265 [main] INFO  org.apache.hadoop.mapreduce.Job  - Job job_1456738266914_0006 completed
successfully
53393 [main] INFO  org.apache.hadoop.mapreduce.Job  - Counters: 50
```

说明

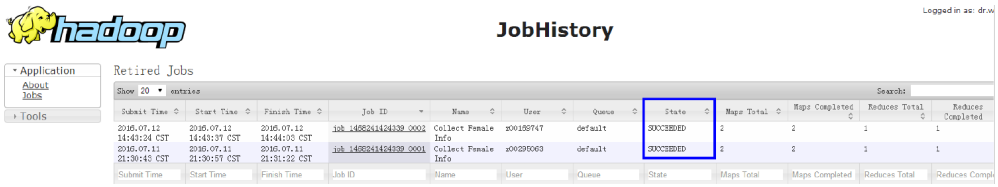
在Windows环境运行样例代码时会出现下面的异常，但是不影响业务：

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

- **通过MapReduce服务的WebUI进行查看**

  使用具有任务查看权限的用户登录FusionInsight Manager，选择“集群 > 服务 > Mapreduce > JobHistoryServer”进入Web界面后查看任务执行状态。

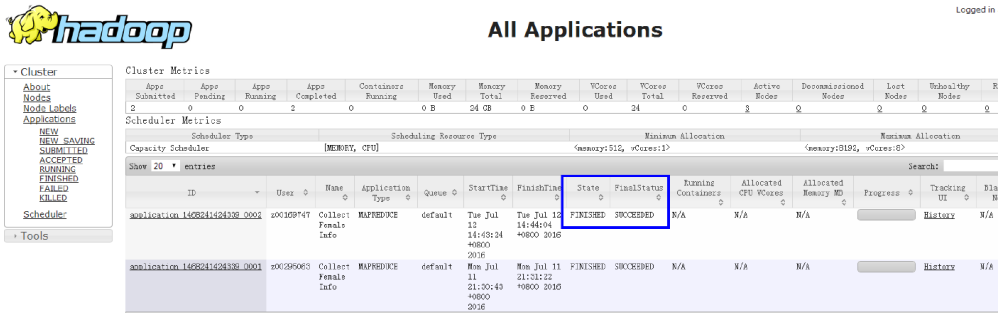
图 23-8 JobHistory Web UI 界面



- **通过YARN服务的WebUI进行查看**

  使用具有任务查看权限的用户登录FusionInsight Manager，选择“集群 > 服务 > Yarn > ResourceManager(主)”进入Web界面后查看任务执行状态。

图 23-9 ResourceManager Web UI 页面



- **查看MapReduce日志获取应用运行情况**

  您可以查看MapReduce日志了解应用运行情况，并根据日志信息调整应用程序。

23.6.3 在 Linux 环境中调测 MapReduce 应用

操作场景

在程序代码完成开发后，您可以在Linux环境中运行应用。

MapReduce应用程序运行完成后，可通过如下方式查看应用程序的运行情况。

- 通过运行结果查看程序运行情况。
- 登录MapReduce WebUI查看应用程序运行情况。
- 登录Yarn WebUI查看应用程序运行情况。
- 通过MapReduce日志获取应用程序运行情况。

前提条件

- 已将**准备连接MapReduce集群配置文件**获取的配置文件放置到“conf”目录下（例如“**/opt/client/conf**”，该目录需要与**步骤2**上传的“MRTest-XXX.jar”包所在目录相同）。
- 已参考**规划MapReduce访问多组件样例程序数据**准备好待处理的数据。

运行程序

步骤1 进入样例工程本地根目录，在Windows命令提示符窗口中执行下面命令进行打包。

mvn -s "*{maven_setting_path}***" clean package**

说明

- 上述打包命令中的{maven_setting_path}为本地Maven的settings.xml文件路径。
- 打包成功之后，在工程根目录的“target”子目录下获取打好的jar包，例如“*MRTest-XXX.jar*”，jar包名称以实际打包结果为准。

步骤2 上传生成的应用包“MRTest-XXX.jar”到Linux客户端上，例如“**/opt/client/conf**”，与配置文件位于同一目录下。

步骤3 在Linux环境下运行样例工程。

- 对于MapReduce统计样例程序，执行如下命令。

yarn jar MRTest-XXX.jar com.huawei.bigdata.mapreduce.examples.FemaleInfoCollector *<inputPath> <outputPath>*

此命令包含了设置参数和提交job的操作，其中*<inputPath>*指HDFS文件系统中input的路径，*<outputPath>*指HDFS文件系统中output的路径。

说明

- 在执行**yarn jar MRTest-XXX.jar com.huawei.bigdata.mapreduce.examples.FemaleInfoCollector** *<inputPath> <outputPath>*命令之前，<outputPath>目录必须不存在，否则会报错。
- 在MapReduce任务运行过程中禁止重启HDFS服务，否则可能会导致任务失败。

- 对于MapReduce访问多组件样例程序，操作步骤如下。

  a. 在“/opt/client/conf”文件夹中创建文件“jaas_mr.conf”，文件内容如下：

  ```
  Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="user.keytab"
  principal="test@<系统域名>"
  useTicketCache=false
  storeKey=true
  debug=true;
  };
  ```

  说明

  - test：为**准备集群认证用户信息**创建的用户名称，例如**developuser**。
  - 系统域名：登录FusionInsight Manager后，选择“系统 > 权限 > 域和互信”，查看“本端域”参数，即为当前系统域名。

  b. 在Linux环境中添加样例工程运行所需的classpath，例如：

  export YARN_USER_CLASSPATH=/opt/client/conf:/opt/client/HBase/hbase/lib/*:/opt/client/HBase/hbase/lib/client-facing-thirdparty/*:/opt/client/Hive/Beeline/lib/*

  c. 提交MapReduce任务，执行如下命令，运行样例工程。

  yarn jar MRTest-XXX.jar com.huawei.bigdata.mapreduce.examples.MultiComponentExample

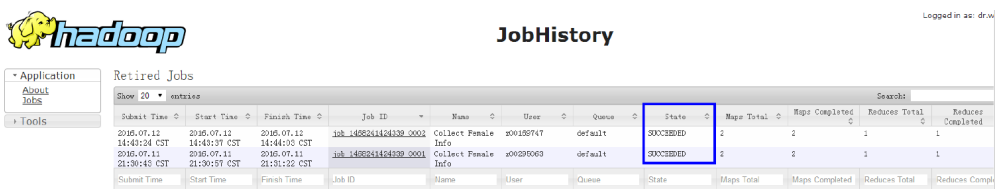
----结束

查看调测结果

- **通过MapReduce服务的WebUI进行查看**

  使用具有任务查看权限的用户登录FusionInsight Manager，选择“集群 >服务 > Mapreduce > JobHistoryServer”进入Web界面后查看任务执行状态。

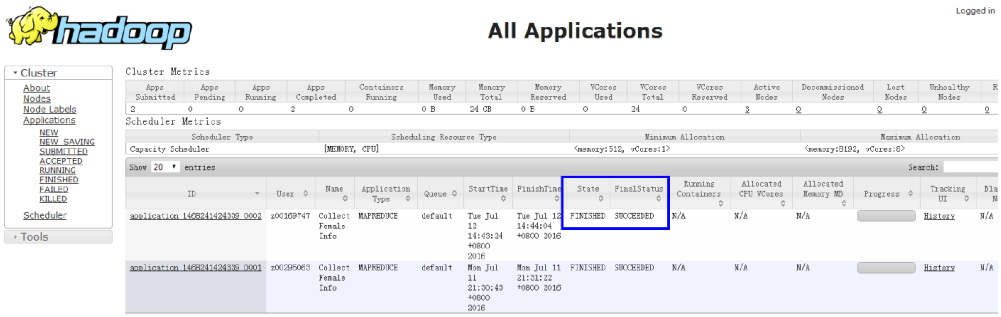
图 23-10 JobHistory Web UI 界面



- **通过YARN服务的WebUI进行查看**

  使用具有任务查看权限的用户登录FusionInsight Manager，选择“集群 > 服务 > Yarn > ResourceManager(主)”进入Web界面后查看任务执行状态。

  图 23-11 ResourceManager Web UI 页面



- **查看MapReduce应用运行结果数据。**
  - 当用户在Linux环境下执行**yarn jar MRTest-*XXX*.jar**命令后，可以通过执行结果显示正在执行的应用的运行情况。例如：

```
linux1:/opt # yarn jar MRTest-XXX.jar /user/mapred/example/input/ /output6
16/02/24 15:45:40 INFO security.UserGroupInformation: Login successful for user admin@<系统域名> using keytab file user.keytab
Login success!!!!!!!!!!!!!!
16/02/24 15:45:40 INFO hdfs.PeerCache: SocketCache disabled.
16/02/24 15:45:41 INFO hdfs.DFSClient: Created HDFS_DELEGATION_TOKEN token 28 for admin on ha-hdfs:hacluster
16/02/24 15:45:41 INFO security.TokenCache: Got dt for hdfs://hacluster; Kind: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:hacluster, Ident: (HDFS_DELEGATION_TOKEN token 28 for admin)
16/02/24 15:45:41 INFO input.FileInputFormat: Total input files to process : 2
16/02/24 15:45:41 INFO mapreduce.JobSubmitter: number of splits:2
16/02/24 15:45:42 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1455853029114_0027
16/02/24 15:45:42 INFO mapreduce.JobSubmitter: Kind: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:hacluster, Ident: (HDFS_DELEGATION_TOKEN token 28 for admin)
16/02/24 15:45:42 INFO impl.YarnClientImpl: Submitted application application_1455853029114_0027
16/02/24 15:45:42 INFO mapreduce.Job: The url to track the job: https://linux1:8090/proxy/application_1455853029114_0027/
16/02/24 15:45:42 INFO mapreduce.Job: Running job: job_1455853029114_0027
16/02/24 15:45:50 INFO mapreduce.Job: Job job_1455853029114_0027 running in uber mode : false
16/02/24 15:45:50 INFO mapreduce.Job:  map 0% reduce 0%
16/02/24 15:45:56 INFO mapreduce.Job:  map 100% reduce 0%
16/02/24 15:46:03 INFO mapreduce.Job:  map 100% reduce 100%
16/02/24 15:46:03 INFO mapreduce.Job: Job job_1455853029114_0027 completed successfully
16/02/24 15:46:03 INFO mapreduce.Job: Counters: 49
```

  - 在Linux环境下执行**yarn application -status <ApplicationID>**，可以通过执行结果显示正在执行的应用的运行情况。例如：

```
linux1:/opt # yarn application -status application_1455853029114_0027
Application Report :
        Application-Id : application_1455853029114_0027
        Application-Name : Collect Female Info
        Application-Type : MAPREDUCE
        User : admin
        Queue : default
        Start-Time : 1456299942302
        Finish-Time : 1456299962343
        Progress : 100%
        State : FINISHED
        Final-State : SUCCEEDED
        Tracking-URL : https://linux1:26014/jobhistory/job/job_1455853029114_0027
        RPC Port : 27100
```

```
AM Host : SZV1000044726
Aggregate Resource Allocation : 114106 MB-seconds, 42 vcore-seconds
Log Aggregation Status : SUCCEEDED
Diagnostics : Application finished execution.
Application Node Label Expression : <Not set>
AM container Node Label Expression : <DEFAULT_PARTITION>
```

- **查看MapReduce日志获取应用运行情况。**

  您可以查看MapReduce日志了解应用运行情况，并根据日志信息调整应用程序。

23.7 MapReduce 应用开发常见问题

23.7.1 MapReduce 接口介绍

23.7.1.1 MapReduce Java API 接口介绍

关于MapReduce的详细API可以直接参考官方网站上的描述：

http://hadoop.apache.org/docs/r3.1.1/api/index.html

常用接口

MapReduce中常见的类如下：

- org.apache.hadoop.mapreduce.Job：用户提交MR作业的接口，用于设置作业参数、提交作业、控制作业执行以及查询作业状态。
- org.apache.hadoop.mapred.JobConf：MapReduce作业的配置类，是用户向Hadoop提交作业的主要配置接口。

表 23-5 类 org.apache.hadoop.mapreduce.Job 的常用接口

功能	说明
Job(Configuration conf, String jobName)，Job(Configuration conf)	新建一个MapReduce客户端，用于配置作业属性，提交作业。
setMapperClass(Class<extends Mapper> cls)	核心接口，指定MapReduce作业的Mapper类，默认为空。也可以在“mapred-site.xml”中配置“mapreduce.job.map.class”项。
setReducerClass(Class<extends Reducer> cls)	核心接口，指定MapReduce作业的Reducer类，默认为空。也可以在“mapred-site.xml”中配置“mapreduce.job.reduce.class”项。
setCombinerClass(Class<extends Reducer> cls)	指定MapReduce作业的Combiner类，默认为空。也可以在“mapred-site.xml”中配置“mapreduce.job.combine.class”项。需要保证reduce的输入输出key，value类型相同才可以使用，谨慎使用。

功能	说明
setInputFormatClass(Class<extends InputFormat> cls)	核心接口，指定MapReduce作业的InputFormat类，默认为TextInputFormat。也可以在“mapred-site.xml”中配置“mapreduce.job.inputformat.class”项。该设置用来指定处理不同格式的数据时需要的InputFormat类，用来读取数据，切分数据块。
setJarByClass(Class< > cls)	核心接口，指定执行类所在的jar包本地位置。java通过class文件找到执行jar包，该jar包被上传到HDFS。
setJar(String jar)	指定执行类所在的jar包本地位置。直接设置执行jar包所在位置，该jar包被上传到HDFS。与setJarByClass(Class< > cls)选择使用一个。也可以在“mapred-site.xml”中配置“mapreduce.job.jar”项。
setOutputFormatClass(Class<extends OutputFormat> theClass)	核心接口，指定MapReduce作业的OutputFormat类，默认为TextOutputFormat。也可以在“mapred-site.xml”中配置“mapred.output.format.class”项，指定输出结果的数据格式。例如默认的TextOutputFormat把每条key，value记录写为文本行。通常场景不配置特定的OutputFormat。
setOutputKeyClass(Class< > theClass)	核心接口，指定MapReduce作业的输出key的类型，也可以在“mapred-site.xml”中配置“mapreduce.job.output.key.class”项。
setOutputValueClass(Class< > theClass)	核心接口，指定MapReduce作业的输出value的类型，也可以在“mapred-site.xml”中配置“mapreduce.job.output.value.class”项。
setPartitionerClass(Class<extends Partitioner> theClass)	指定MapReduce作业的Partitioner类。也可以在“mapred-site.xml”中配置“mapred.partitioner.class”项。该方法用来分配map的输出结果到哪个reduce类，默认使用HashPartitioner，均匀分配map的每条键值对记录。例如在hbase应用中，不同的键值对应的region不同，这就需要设定特殊的partitioner类分配map的输出结果。
setSortComparatorClass(Class<extends RawComparator> cls)	指定MapReduce作业的map任务的输出结果压缩类，默认不使用压缩。也可以在“mapred-site.xml”中配置“mapreduce.map.output.compress”和“mapreduce.map.output.compress.codec”项。当map的输出数据大，减少网络压力，使用压缩传输中间数据。
setPriority(JobPriority priority)	指定MapReduce作业的优先级，共有5个优先级别，VERY_HIGH、HIGH、NORMAL、LOW、VERY_LOW，默认级别为NORMAL。也可以在“mapred-site.xml”中配置“mapreduce.job.priority”项。

表 23-6 类 org.apache.hadoop.mapred.JobConf 的常用接口

方法	说明
setNumMapTasks(int n)	核心接口，指定MapReduce作业的map个数。也可以在“mapred-site.xml”中配置“mapreduce.job.maps”项。 **说明** 指定的InputFormat类用来控制map任务个数，注意该类是否支持客户端设定map个数。
setNumReduceTasks(int n)	核心接口，指定MapReduce作业的reduce个数。默认只启动1个。也可以在“mapred-site.xml”中配置“mapreduce.job.reduces”项。reduce个数由用户控制，通常场景reduce个数是map个数的1/4。
setQueueName(String queueName)	指定MapReduce作业的提交队列。默认使用default队列。也可以在“mapred-site.xml”中配置“mapreduce.job.queuename”项。

23.7.1.2 MapReduce REST API 接口介绍

功能简介

通过HTTP REST API来查看更多MapReduce任务的信息。目前Mapresuce的REST接口可以查询已完成任务的状态信息。完整和详细的接口请直接参考官网上的描述以了解其使用：

http://hadoop.apache.org/docs/r3.1.1/hadoop-mapreduce-client/hadoop-mapreduce-client-hs/HistoryServerRest.html

准备运行环境

1. 在节点上安装客户端，例如安装到“/opt/client”目录，可参考“安装客户端”。
2. 进入客户端安装目录“/opt/client”，执行下列命令初始化环境变量。

   source *bigdata_env*

   kinit *组件业务用户*

   说明

   kinit一次票据时效24小时。24小时后再次运行样例，需要重新kinit。

3. 与HTTP服务访问相比，以HTTPS方式访问Mapreduce时，由于使用了SSL安全加密，需要确保Curl命令所支持的SSL协议在集群中已添加支持。若不支持，可对应修改集群中SSL协议。例如，若Curl仅支持TLSv1协议，修改方法如下：

登录FusionInsight Manager页面，单击“集群 > *待操作集群的名称* > 服务 > Yarn > 配置 >全部配置”，在“搜索”框里搜索“hadoop.ssl.enabled.protocols”，查看参数值是否包含“TLSv1”，若不包含，则在配置项“hadoop.ssl.enabled.protocols”中追加“,TLSv1”。清空“ssl.server.exclude.cipher.list”配置项的值，否则以HTTPS访

问不了Yarn。单击“保存”，在“保存配置”中单击“确定”，保存完成后选择“更多>重启服务”重启该服务。

说明

1. MapReduce的配置项hadoop.ssl.enabled.protocols和ssl.server.exclude.cipher.list的值直接引用Yarn中对应配置项的值，因此需要修改Yarn中对应配置项的值并重启Yarn和MapReduce服务。
2. TLSv1协议存在安全漏洞，请谨慎使用。

操作步骤

获取MapReduce上已完成任务的具体信息

- 命令：
  ```
  curl -k -i --negotiate -u : "https://10.120.85.2:26014/ws/v1/history/mapreduce/jobs"
  ```
  其中10.120.85.2为MapReduce的“JHS_FLOAT_IP”参数的参数值，26014为JobHistoryServer的端口号。

  说明

  在Red Hat 6.x以及CentOS 6.x版本，使用curl命令访问JobHistoryServer会有兼容性问题，导致无法返回正确结果。
- 用户能看到历史任务的状态信息（任务id，开始时间，结束时间，是否执行成功等信息）
- 运行结果
  ```
  {
     "jobs":{
        "job":[
           {
              "submitTime":1525693184360,
              "startTime":1525693194840,
              "finishTime":1525693215540,
              "id":"job_1525686535456_0001",
              "name":"QuasiMonteCarlo",
              "queue":"default",
              "user":"mapred",
              "state":"SUCCEEDED",
              "mapsTotal":1,
              "mapsCompleted":1,
              "reducesTotal":1,
              "reducesCompleted":1
           }
        ]
     }
  }
  ```
- 结果分析：

  通过这个接口，可以查询当前集群中已完成的MapReduce任务，并且可以得到表23-7

表 23-7 常用信息

参数	参数描述
submitTime	任务提交时间
startTime	任务开始执行时间
finishTime	任务执行完成时间

参数	参数描述
queue	任务队列
user	提交这个任务的用户
state	任务执行成功或失败

23.7.2 提交 MapReduce 任务时客户端长时间无响应

问题

向YARN服务器提交MapReduce任务后，客户端提示如下信息后长时间无响应。

```
16/03/03 16:44:56 INFO hdfs.DFSClient: Created HDFS_DELEGATION_TOKEN token 44 for admin on ha-hdfs:hacluster
16/03/03 16:44:56 INFO security.TokenCache: Got dt for hdfs://hacluster; Kind: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:hacluster,
Ident: (HDFS_DELEGATION_TOKEN token 44 for admin)
16/03/03 16:44:56 INFO client.ConfiguredRMFailoverProxyProvider: Failing over to 53
16/03/03 16:44:57 INFO input.FileInputFormat: Total input files to process : 200
16/03/03 16:44:57 INFO mapreduce.JobSubmitter: number of splits:200
16/03/03 16:44:57 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1456738266914_0005
16/03/03 16:44:57 INFO mapreduce.JobSubmitter: Kind: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:hacluster,
Ident: (HDFS_DELEGATION_TOKEN token 44 for admin)
16/03/03 16:44:57 INFO impl.YarnClientImpl: Submitted application application_1456738266914_0005
16/03/03 16:44:57 INFO mapreduce.Job: The url to track the job: https://linux2:8090/proxy/application_1456738266914_0005/
16/03/03 16:44:57 INFO mapreduce.Job: Running job: job_1456738266914_0005
```

回答

对于上述出现的问题，ResourceManager在其WebUI上提供了MapReduce作业关键步骤的诊断信息，对于一个已经提交到YARN上的MapReduce任务，用户可以通过该诊断信息获取当前作业的状态以及处于该状态的原因。

具体操作：登录FusionInsight Manager，单击“集群 > *待操作集群的名称* > 服务 > Yarn > ResourceManager(主)”打开WebUI界面，在ResourceManager(主)的WebUI界面中，单击提交的MapReduce任务，在打开的页面中查看诊断信息，根据诊断信息再采取相应的措施。

或者也可以通过查看MapReduce日志了解应用运行情况，并根据日志信息调整应用程序。

23.7.3 网络问题导致运行应用程序时出现异常

问题

应用程序在Windows环境下运行时，发现连接不上MRS集群，而在Linux环境下（和安装了MRS集群的机器是同一个网络）却运行正常。

回答

由于kerberos认证需要使用UDP协议，而防火墙做了特殊处理关掉了需要使用的UDP端口，导致应用程序在Windows环境下运行的机器与MRS集群的网络不是完全通畅

的，需要重新设置防火墙，把需要使用的UDP端口都打开，保证应用程序在Windows环境下运行的机器与MRS集群的网络是完全通畅的。

23.7.4 MapReduce 二次开发远程调试

问题

MapReduce二次开发过程中如何远程调试业务代码?

回答

MapReduce开发调试采用的原理是Java的远程调试机制，在Map/Reduce任务启动时，添加Java远程调试命令。

步骤1 首先理解两个参数：“mapreduce.map.java.opts”和“mapreduce.reduce.java.opts”，这两个参数为客户端参数，分别指定了Map/Reduce任务对应的JVM启动参数。

修改客户端“mapred-site.xml”配置文件中“mapreduce.map.java.opts”和“mapreduce.reduce.java.opts”参数，分别加入调试命令“-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8000”，保存文件。

步骤2 MapReduce为分布式计算框架，Map/Reduce任务启动所在的节点存在不确定性，建议将集群内NodeManager实例只保留一个运行，其他全部停止，以保证任务一定会在这个唯一运行的NodeManager节点上启动。

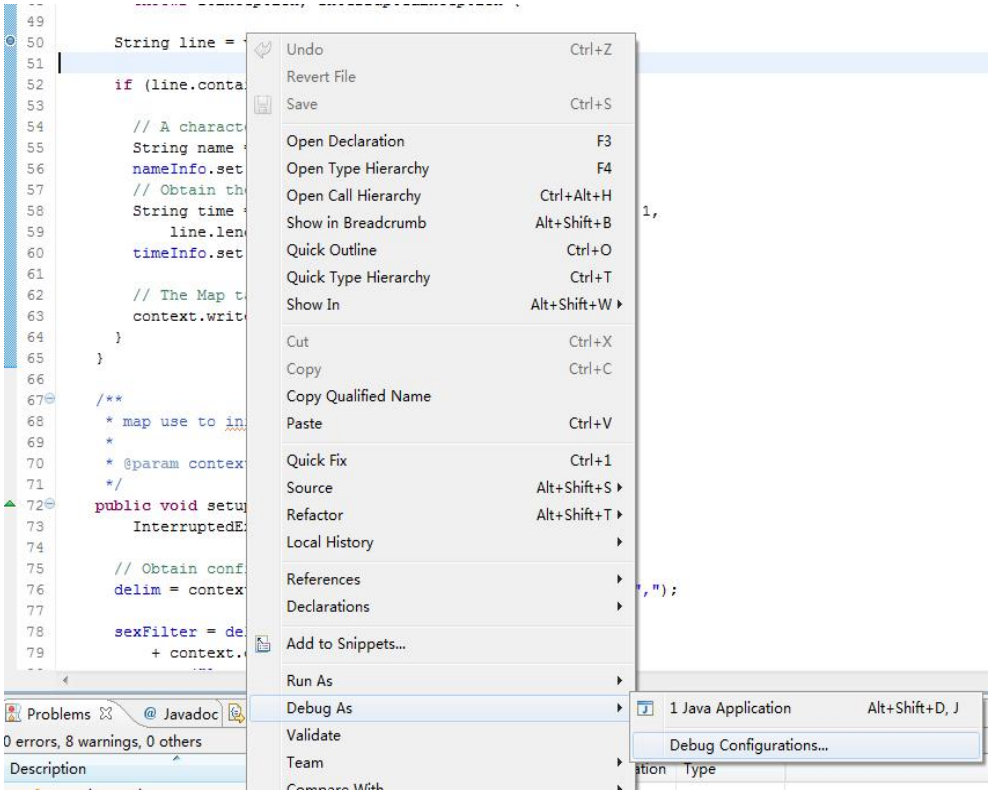
步骤3 在客户端提交MapReduce任务，在Map/Reduce任务启动时会挂起并监测8000端口，等待远程调试。

步骤4 在IDE上，选择MapReduce任务的实现类，通过配置远程调试信息，执行Debug。

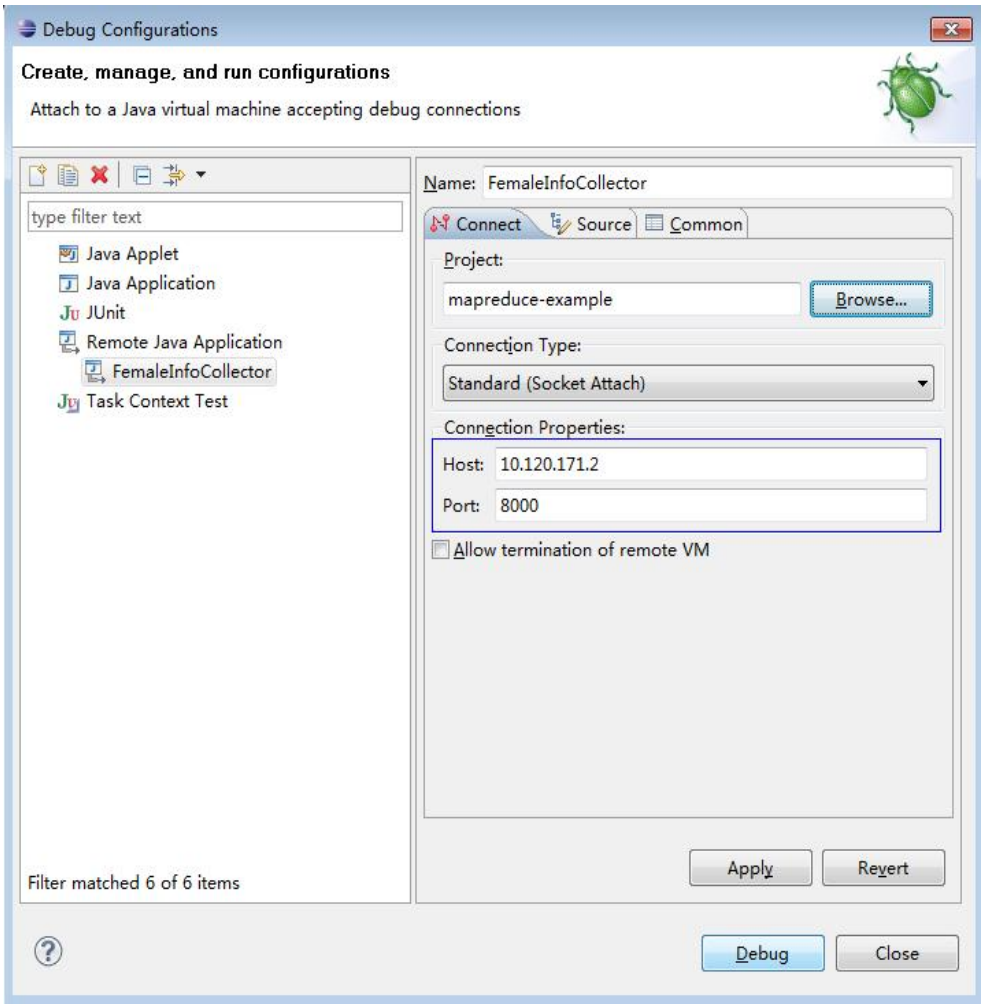
1. 设置断点，双击蓝框区域设置或取消断点。


```
39      /**
40       * Distributed computing
41       *
42       * @param key     Object : location offset of the source file
43       * @param value   Text : a row of characters in the source file
44       * @param context Context : output parameter
45       * @throws IOException , InterruptedException
46       */
47      public void map(Object key, Text value, Context context)
48          throws IOException, InterruptedException {
49
50        String line = value.toString();
51
52        if (line.contains(sexFilter)) {
53
54          // A character string that has been read
55          String name = line.substring(0, line.indexOf(delim));
56          nameInfo.set(name);
57          // Obtain the dwell duration.
58          String time = line.substring(line.lastIndexOf(delim) + 1,
59              line.length());
60          timeInfo.set(Integer.parseInt(time));
61
62          // The Map task outputs a key-value pair.
63          context.write(nameInfo, timeInfo);
64        }
65      }
66
```

2. 配置远程调试信息，“右键->Debug As->Debug Configurations...”。



3. 在弹出的页面，双击“Remote Java Application”，设置Connection Properties，其中Host为运行的NodeManager节点IP，Port端口号为8000，然后单击“Debug”。



----结束

说明

若使用IDE直接提交MapReduce任务，则IDE即成为客户端的角色，参考**步骤1**修改二次开发工程中的“mapred-site.xml”即可。

24 MapReduce 开发指南（普通模式）

24.1 MapReduce 应用开发简介

MapReduce 简介

Hadoop MapReduce是一个使用简易的并行计算软件框架，基于它写出来的应用程序能够运行在由上千个服务器组成的大型集群上，并以一种可靠容错的方式并行处理上T级别的数据集。

一个MapReduce作业（application/job）通常会把输入的数据集切分为若干独立的数据块，由map任务（task）以完全并行的方式来处理。框架会对map的输出先进行排序，然后把结果输入给reduce任务，最后返回给客户端。通常作业的输入和输出都会被存储在文件系统中。整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

MapReduce主要特点如下：

- 大规模并行计算
- 适用于大型数据集
- 高容错性和高可靠性
- 合理的资源调度

常用概念

- Hadoop shell命令

  Hadoop基本shell命令，包括提交MapReduce作业，kill MapReduce作业，进行HDFS文件系统各项操作等。

- MapReduce输入输出(InputFormat，OutputFormat)

  MapReduce框架根据用户指定的InputFormat切割数据集，读取数据，并提供给map任务多条键值对进行处理，决定并行启动的map任务数目。MapReduce框架根据用户指定的OutputFormat，把生成的键值对输出为特定格式的数据。

  map、reduce两个阶段都处理在<key,value>键值对上，也就是说，框架把作业的输入作为一组<key,value>键值对，同样也产出一组<key,value>键值对作为作业的输出，这两组键值对的类型可能不同。对单个map和reduce而言，对键值对的处理为单线程串行处理。

框架需要对key和value的类(classes)进行序列化操作，因此，这些类需要实现Writable接口。另外，为了方便框架执行排序操作，key类必须实现WritableComparable接口。

一个MapReduce作业的输入和输出类型如下所示：

(input)<k1,v1> —> map —> <k2,v2> —> 汇总数据 —> <k2,List(v2)> —> reduce —> <k3,v3>(output)

- 业务核心

  应用程序通常只需要分别继承Mapper类和Reducer类，并重写其map和reduce方法来实现业务逻辑，它们组成作业的核心。

- MapReduce WebUI界面

  用于监控正在运行的或者历史的MapReduce作业在MapReduce框架各个阶段的细节，以及提供日志显示，帮助用户更细粒度地去开发、配置和调优作业。

- 归档

  用来保证所有映射的键值对中的每一个共享相同的键组。

- 混洗

  从Map任务输出的数据到Reduce任务的输入数据的过程称为Shuffle。

- 映射

  用来把一组键值对映射成一组新的键值对。

24.2 MapReduce 应用开发流程介绍

开发流程中各阶段的说明如**图24-1**和**表24-1**所示。

图 24-1 MapReduce 应用程序开发流程

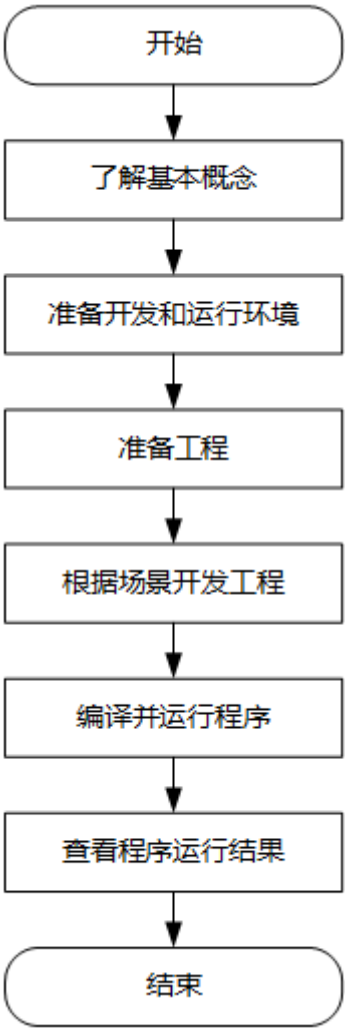


表 24-1 MapReduce 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解MapReduce的基本概念。	**MapReduce应用开发简介**
准备开发和运行环境	使用IntelliJ IDEA工具，请根据指导完成开发环境配置。 MapReduce的运行环境即MapReduce客户端，请根据指导完成客户端的安装和配置。	**准备MapReduce开发和运行环境**
准备工程	MapReduce提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。或者您可以根据指导，新建一个MapReduce工程。	**导入并配置MapReduce样例工程**
根据场景开发工程	提供了样例工程。 帮助用户快速了解MapReduce各部件的编程接口。	**开发MapReduce应用**

阶段	说明	参考文档
编译并运行程序	指导用户将开发好的程序编译并提交运行。	调测MapReduce应用
查看程序运行结果	程序运行结果会写在用户指定的路径下。用户还可以通过UI查看应用运行情况。	调测MapReduce应用

24.3 MapReduce 样例工程介绍

MRS样例工程获取地址为https://github.com/huaweicloud/huaweicloud-mrs-example，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下MapReduce相关样例工程：

表 24-2 MapReduce 相关样例工程

样例工程位置	描述
mapreduce-example-normal	• MapReduce统计数据的应用开发示例： 提供了一个MapReduce统计数据的应用开发示例，通过类CollectionMapper实现数据分析、处理，并输出满足用户需要的数据信息。 相关样例介绍请参见MapReduce统计样例程序。 • MapReduce作业访问多组件的应用开发示例： 以MapReduce访问HDFS、HBase、Hive为例，介绍如何编写MapReduce作业访问多个服务组件。帮助用户理解认证、配置加载等关键使用方式。 相关样例介绍请参见MapReduce访问多组件样例程序。

24.4 准备 MapReduce 应用开发环境

24.4.1 准备 MapReduce 开发和运行环境

准备开发环境

在进行应用开发时，要准备的开发和运行环境如表24-3所示。

表 24-3 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装和配置IntelliJ IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程
安装Maven	开发环境基本配置，用于项目管理，贯穿软件开发生命周期。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
7-zip	用于解压“*.zip”和“*.rar”文件。 支持7-Zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果本地Windows开发环境和集群业务平面网络互通，可下载集群客户端到本地，获取相关调测程序所需的集群配置文件及配置网络连通后，然后直接在Windows中进行程序调测。
  a. **登录FusionInsight Manager页面**，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“完整客户端”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig.tar”，继续解压该文件。解压到本地PC的“D:\FusionInsight_Cluster_1_Services_ClientConfig”目录下（路径中不能有空格）。

b. 进入客户端解压路径“FusionInsight_Cluster_1_Services_ClientConfig\Yarn\config”，获取集群相关配置文件，并将配置文件导入到样例工程的配置文件目录中（通常为“conf”文件夹）。

c. 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与解压目录下“hosts”文件中所列出的各主机在网络上互通。

说明

- 当客户端所在主机不是集群中的节点时，配置客户端网络连接，可避免执行客户端命令时出现错误。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 如果使用Linux环境调测程序，需准备安装集群客户端的Linux节点并获取相关配置文件。

a. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。

客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

b. **登录FusionInsight Manager页面**，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig/Yarn/config”路径下的所有配置文件至客户端节点，放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp Yarn/config/* root@*客户端节点IP地址*:/opt/client/conf

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

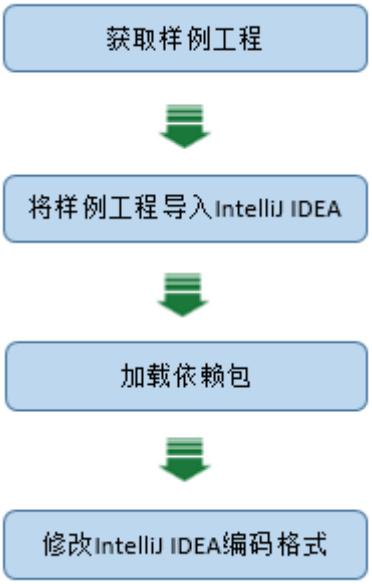
24.4.2 导入并配置 MapReduce 样例工程

操作场景

MapReduce针对多个场景提供样例工程，帮助客户快速学习MapReduce工程。

以下操作步骤以导入MapReduce样例代码为例。操作流程如**图24-2**所示。

图 24-2 导入样例工程流程



操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src”目录下的样例工程文件夹“mapreduce-example-normal”。

步骤2 导入样例工程到IntelliJ IDEA开发环境。

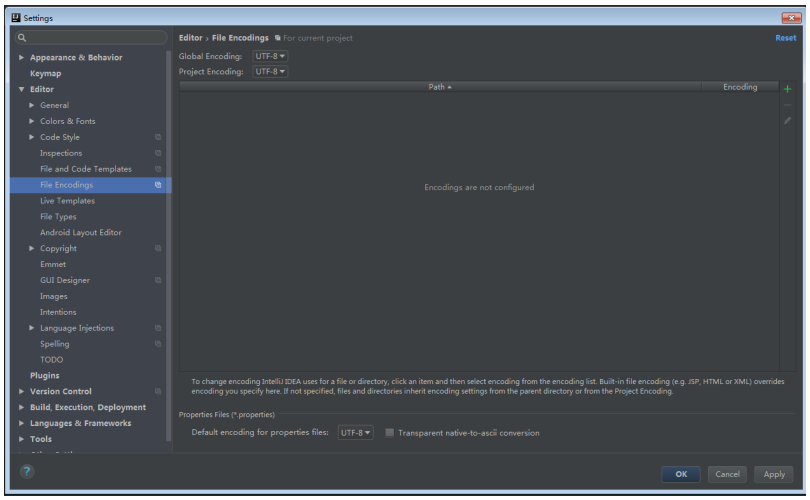
1. 打开IntelliJ IDEA，依次选择“File > Open”。
2. 在弹出的Open File or Project会话框中选择样例工程文件夹“mapreduce-example-normal”，单击“OK”。

步骤3 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File> Settings”。

   弹出“Settings”窗口。
2. 在左边导航上选择“Editor > File Encodings”，在“Global Encoding”和“Project Encodings”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如**图24-3**所示。

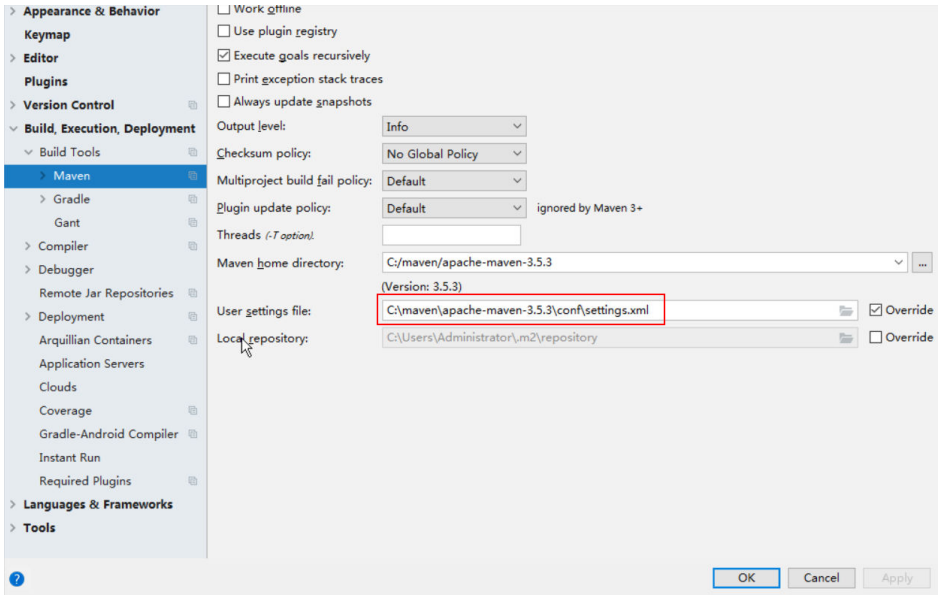
图 24-3 设置 IntelliJ IDEA 的编码格式



步骤4 参考**配置华为开源镜像仓**章节描述，增加开源镜像仓地址等配置信息到本地Maven的“setting.xml”配置文件。

修改完成后，使用IntelliJ IDEA开发工具时，可选择“File > Settings > Build, Execution, Deployment > Build Tools > Maven”查看当前“settings.xml”文件放置目录，确保该目录为“*<本地Maven安装目录>*\conf\settings.xml”。

图 24-4 “settings.xml”文件放置目录



----结束

参考信息

针对MapReduce提供的几个样例程序，其对应的依赖包如下：

- MapReduce统计样例程序
  没有需要额外导入的jar包
- MapReduce访问多组件样例程序

说明

- 导入样例工程之后，如果需要使用访问多组件样例程序，请确保集群已安装Hive、HBase服务。
- 不使用访问多组件样例程序时，如果不影响统计样例程序的正常编译，可忽略多组件样例程序相关报错信息，否则请在导入样例工程后将多组件样例程序类文件删除。

24.4.3 （可选）创建 MapReduce 样例工程

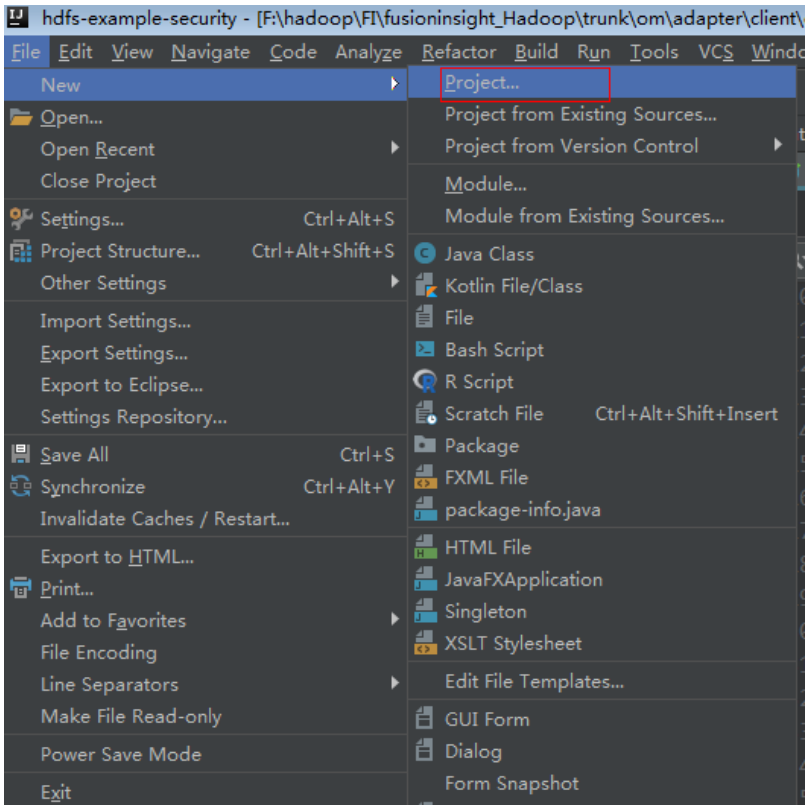
操作场景

除了导入MapReduce样例工程，您还可以使用IntelliJ IDEA新建一个MapReduce工程。

操作步骤

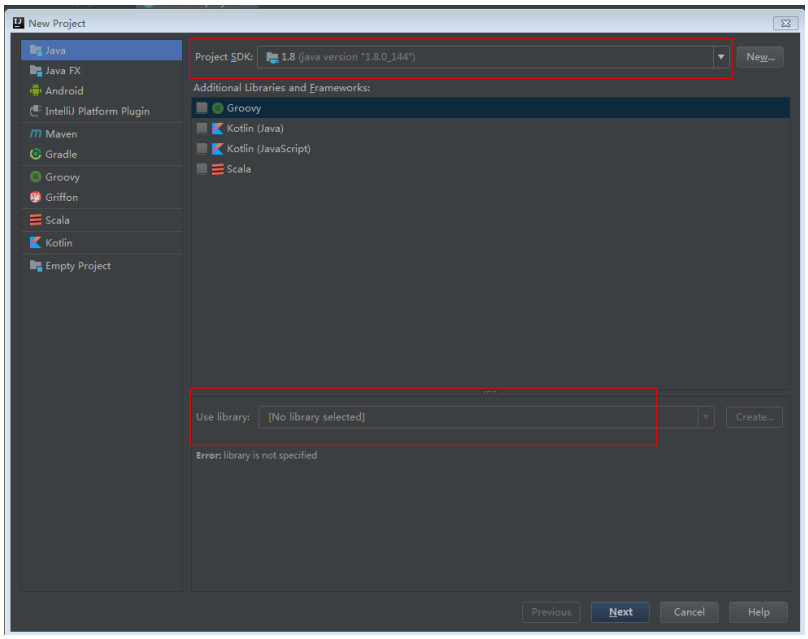
步骤1 打开IntelliJ IDEA工具，选择“File > New > Project”，如图24-5所示。

图 24-5 创建工程



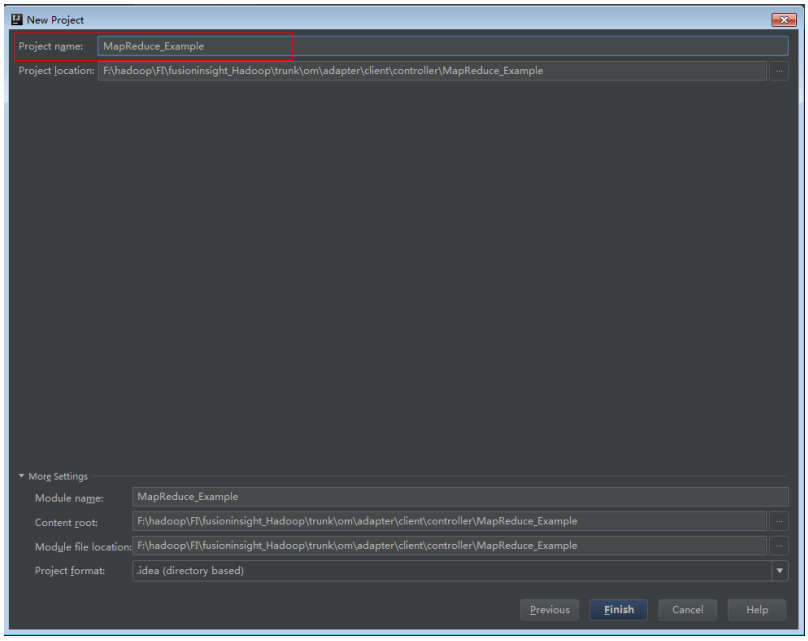
步骤2 在“New Project”页面选择“Java”，然后配置工程需要的JDK和其他Java库。如图24-6所示。配置完成后单击“Next”。

图 24-6 配置工程所需 SDK 信息



步骤3 在会话框中填写新建的工程名称。然后单击Finish完成创建。

图 24-7 填写工程名称



----结束

24.5 开发 MapReduce 应用

24.5.1 MapReduce 统计样例程序

24.5.1.1 MapReduce 统计样例程序开发思路

场景说明

假定用户有某个周末网民网购停留时间的日志文本，基于某些业务要求，要求开发MapReduce应用程序实现如下功能：

- 统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。
- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

log1.txt：周六网民停留日志

```
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

log2.txt：周日网民停留日志

```
LiuYang,female,20
YuanJing,male,10
CaiXuyu,female,50
FangBo,female,50
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
CaiXuyu,female,50
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
FangBo,female,50
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

数据规划

首先需要把原日志文件放置在HDFS系统里。

1. 在Linux系统上新建两个文本文件，将log1.txt中的内容复制保存到input_data1.txt，将log2.txt中的内容复制保存到input_data2.txt。
2. 在HDFS上建立一个文件夹，“/tmp/input”，并上传input_data1.txt，input_data2.txt到此目录，命令如下：
   a. 在Linux系统HDFS客户端使用命令***hdfs dfs -mkdir /tmp/input***
   b. 在Linux系统HDFS客户端使用命令***hdfs dfs -put*** *local_filepath /tmp/input*

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

- 读取原文件数据。
- 筛选女性网民上网时间数据信息。
- 汇总每个女性上网总时间。
- 筛选出停留总时间大于2个小时的女性网民信息。

24.5.1.2 MapReduce 统计样例代码

功能介绍

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为三个部分：

- 从原文件中筛选女性网民上网时间数据信息，通过类CollectionMapper继承Mapper抽象类实现。
- 汇总每个女性上网时间，并输出时间大于两个小时的女性网民信息，通过类CollectionReducer继承Reducer抽象类实现。
- main方法提供建立一个MapReduce job，并提交MapReduce作业到hadoop集群。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.mapreduce.examples.FemaleInfoCollector类：

样例1：类CollectionMapper定义Mapper抽象类的map()方法和setup()方法。

```
public static class CollectionMapper extends
        Mapper<Object, Text, Text, IntWritable> {

// 分隔符。
String delim;
// 性别筛选。
String sexFilter;

// 姓名信息。
private Text nameInfo = new Text();

// 输出的key,value要求是序列化的。
private IntWritable timeInfo = new IntWritable(1);

 /**
 * 分布式计算
 *
 * @param key Object : 原文件位置偏移量。
 * @param value Text : 原文件的一行字符数据。
 * @param context Context : 出参。
 * @throws IOException , InterruptedException
 */
    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException
    {

        String line = value.toString();

        if (line.contains(sexFilter))
        {

                // 读取的一行字符串数据。
                String name = line.substring(0, line.indexOf(delim));
```

```
                nameInfo.set(name);
                // 获取上网停留时间。
                String time = line.substring(line.lastIndexOf(delim) + 1,
                        line.length());
                timeInfo.set(Integer.parseInt(time));

                // map输出key，value键值对。
                context.write(nameInfo, timeInfo);
            }
        }

        /**
     * map调用，做一些初始工作。
     *
     * @param context Context
     */
        public void setup(Context context) throws IOException,
                InterruptedException
    {

                // 通过Context可以获得配置信息。
        delim = context.getConfiguration().get("log.delimiter", ",");


sexFilter = delim + context.getConfiguration().get("log.sex.filter", "female") + delim;
        }
    }
```

样例2：类CollectionReducer定义Reducer抽象类的reduce()方法。

```
    public static class CollectionReducer extends
            Reducer<Text, IntWritable, Text, IntWritable>
{

    // 统计结果。
        private IntWritable result = new IntWritable();

        // 总时间门槛。
        private int timeThreshold;

     /**
     * @param key Text : Mapper后的key项。
     * @param values Iterable : 相同key项的所有统计结果。
     * @param context Context
     * @throws IOException , InterruptedException
     */
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context) throws IOException, InterruptedException
 {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }

            // 如果时间小于门槛时间，不输出结果。
            if (sum < timeThreshold)
      {
                return;
            }
            result.set(sum);

            // reduce输出为key：网民的信息，value：该网民上网总时间。
            context.write(key, result);
        }

        /**
         * setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次。
     *
     * @param context Context
```

```
   * @throws IOException , InterruptedException
   */
      public void setup(Context context) throws IOException,
            InterruptedException
  {

        // Context可以获得配置信息。
        timeThreshold = context.getConfiguration().getInt(
              "log.time.threshold", 120);
      }
  }
```

样例3：main()方法创建一个job，指定参数，提交作业到hadoop集群。

```
public static void main(String[] args) throws Exception {
  // 初始化环境变量。
  Configuration conf = new Configuration();

  // 获取入参。
  String[] otherArgs = new GenericOptionsParser(conf, args)
      .getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: collect female info <in> <out>");
    System.exit(2);
  }

  // 初始化Job任务对象。
   Job job = Job.getInstance(conf, "Collect Female Info");
  job.setJarByClass(FemaleInfoCollector.class);

  // 设置运行时执行map，reduce的类，也可以通过配置文件指定。
  job.setMapperClass(CollectionMapper.class);
  job.setReducerClass(CollectionReducer.class);

  // 设置combiner类，默认不使用，使用时通常使用和reduce一样的类。
  // Combiner类需要谨慎使用，也可以通过配置文件指定。
  job.setCombinerClass(CollectionCombiner.class);

  // 设置作业的输出类型。
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

  // 提交任务交到远程环境上执行。
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

样例4：类CollectionCombiner实现了在map端先合并map输出的数据，减少map和reduce之间传输的数据量。

```
/**
 * Combiner class
 */
public static class CollectionCombiner extends
Reducer<Text, IntWritable, Text, IntWritable> {

// Intermediate statistical results
private IntWritable intermediateResult = new IntWritable();

/**
 * @param key     Text : key after Mapper
 * @param values  Iterable : all results with the same key in this map task
 * @param context Context
 * @throws IOException , InterruptedException
 */
public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values) {
```



```
sum += val.get();
}

intermediateResult.set(sum);

// In the output information, key indicates netizen information,
// and value indicates the total online time of the netizen in this map task.
context.write(key, intermediateResult);
}

}
```

24.5.2 MapReduce 访问多组件样例程序

24.5.2.1 MapReduce 访问多组件样例程序开发思路

场景说明

该样例以MapReduce访问HDFS、HBase、Hive为例，介绍如何编写MapReduce作业访问多个服务组件。帮助用户理解认证、配置加载等关键使用方式。

该样例逻辑过程如下：

以HDFS文本文件为输入数据：

log1.txt：数据输入文件

```
YuanJing,male,10
GuoYijun,male,5
```

Map阶段：

1. 获取输入数据的一行并提取姓名信息。
2. 查询HBase一条数据。
3. 查询Hive一条数据。
4. 将HBase查询结果与Hive查询结果进行拼接作为Map输出。

Reduce阶段：

1. 获取Map输出中的最后一条数据。
2. 将数据输出到HBase。
3. 将数据保存到HDFS。

数据规划

1. 创建HDFS数据文件。
   a. 在Linux系统上新建文本文件，将log1.txt中的内容复制保存到data.txt。
   b. 在HDFS上创建一个文件夹，“/tmp/examples/multi-components/mapreduce/input/”，并上传data.txt到此目录，命令如下：
      i. 在Linux系统HDFS客户端使用命令***hdfs dfs -mkdir -p*** */tmp/examples/multi-components/mapreduce/input/*
      ii. 在Linux系统HDFS客户端使用命令***hdfs dfs -put*** *data.txt /tmp/examples/multi-components/mapreduce/input/*
2. 创建HBase表并插入数据。

a. 在Linux系统HBase客户端执行**source bigdata_env**，并使用命令***hbase shell***。
b. 在HBase shell交互窗口创建数据表table1，该表有一个列族cf，使用命令***create 'table1', 'cf'***。
c. 插入一条rowkey为1、列名为cid、数据值为123的数据，使用命令***put 'table1', '1', 'cf:cid', '123'***。
d. 执行命令***quit***退出。

3. 创建Hive表并载入数据。
a. 在Linux系统Hive客户端使用命令***beeline***。
b. 在Hive beeline交互窗口创建数据表person，该表有3个字段：name/gender/stayTime，使用命令***CREATE TABLE person(name STRING, gender STRING, stayTime INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' stored as textfile;***。
c. 在Hive beeline交互窗口加载数据文件，***LOAD DATA INPATH** '/tmp/examples/multi-components/mapreduce/input/' **OVERWRITE INTO TABLE person;***。
d. 执行命令***!q***退出。

4. 由于Hive加载数据将HDFS对应数据目录清空，所以需再次执行**1**。

24.5.2.2 MapReduce 访问多组件样例代码

功能介绍

主要分为三个部分：

- 从HDFS原文件中抽取name信息，查询HBase、Hive相关数据，并进行数据拼接，通过类MultiComponentMapper继承Mapper抽象类实现。
- 获取拼接后的数据取最后一条输出到HBase、HDFS，通过类MultiComponentReducer继承Reducer抽象类实现。
- main方法提供建立一个MapReduce job，并提交MapReduce作业到Hadoop集群。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.mapreduce.examples.MultiComponentExample类：

样例1：类MultiComponentMapper定义Mapper抽象类的map方法。

```
 private static class MultiComponentMapper extends Mapper<Object, Text, Text, Text> {

   Configuration conf;

   @Override protected void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

     conf = context.getConfiguration();

     String name = "";
     String line = value.toString();
     if (line.contains("male")) {
       // A character string that has been read
       name = line.substring(0, line.indexOf(","));
```

```
  }
  // 1. 读取HBase数据
  String hbaseData = readHBase();

  // 2. 读取Hive数据
  String hiveData = readHive(name);

  // Map输出键值对，内容为HBase与Hive数据拼接的字符串
  context.write(new Text(name), new Text("hbase:" + hbaseData + ", hive:" + hiveData));
 }
```

样例2：HBase数据读取的readHBase方法。

```
private String readHBase() {
  String tableName = "table1";
  String columnFamily = "cf";
  String hbaseKey = "1";
  String hbaseValue;

  Configuration hbaseConfig = HBaseConfiguration.create(conf);
  org.apache.hadoop.hbase.client.Connection conn = null;
  try {
    // 建立HBase连接
    conn = ConnectionFactory.createConnection(hbaseConfig);
    // 获取HBase表
    Table table = conn.getTable(TableName.valueOf(tableName));
    // 创建一个HBase Get操作实例
    Get get = new Get(hbaseKey.getBytes());
    // 提交Get请求
    Result result = table.get(get);
    hbaseValue = Bytes.toString(result.getValue(columnFamily.getBytes(), "cid".getBytes()));

    return hbaseValue;

  } catch (IOException e) {
    LOG.warn("Exception occur ", e);
  } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (Exception e1) {
                LOG.error("Failed to close the connection ", e1);
            }
          }



  }

  return "";
}
```

样例3：Hive数据读取的readHive方法。

```
private String readHive(String name) throws IOException {
  //加载配置信息
  Properties clientInfo = null;
  String userdir = System.getProperty("user.dir") + "/";
  InputStream fileInputStream = null;
  try {
    clientInfo = new Properties();
    String hiveclientProp = userdir + "hiveclient.properties";
    File propertiesFile = new File(hiveclientProp);
    fileInputStream = new FileInputStream(propertiesFile);
    clientInfo.load(fileInputStream);
  } catch (Exception e) {
    throw new IOException(e);
  } finally {
    if (fileInputStream != null) {
      fileInputStream.close();
```

```
      }
    }
    String zkQuorum = clientInfo.getProperty("zk.quorum");
    String zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    String serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
    // 请仔细阅读此内容:
    // MapReduce任务通过JDBC方式访问Hive
    // Hive会将sql查询封装成另一个MapReduce任务并提交
    // 所以不建议在MapReduce作业中调用Hive
    final String driver = "org.apache.hive.jdbc.HiveDriver";
    // 集群zookeeper节点信息

    String sql = "select name,sum(stayTime) as " + "stayTime from person where name = '" + name + "'
group by name";

    StringBuilder sBuilder = new StringBuilder("jdbc:hive2://").append(zkQuorum).append("/");
    sBuilder
        .append(";serviceDiscoveryMode=")
        .append(serviceDiscoveryMode)
        .append(";zooKeeperNamespace=")
        .append(zooKeeperNamespace)
        .append(";");
    String url = sBuilder.toString();
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
      Class.forName(driver);
      connection = DriverManager.getConnection(url, "", "");
      statement = connection.prepareStatement(sql);
      //执行查询
      resultSet = statement.executeQuery();

      if (resultSet.next()) {
        return resultSet.getString(1);
      }
    } catch (ClassNotFoundException e) {
      LOG.warn("Exception occur ", e);
    } catch (SQLException e) {
      LOG.warn("Exception occur ", e);
    } finally {

            if (null != resultSet) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    // handle exception
                }
            }
            if (null != statement) {
                try {
                    statement.close();
                } catch (SQLException e) {
                    // handle exception
                }
            }
            if (null != connection) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    // handle exception
                }
            }
    }
    return "";
  }
```

须知

样例中zkQuorum对象需替换为实际ZooKeeper集群节点信息。

样例4：类MultiComponentReducer定义Reducer抽象类的reduce方法。

```
private static class MultiComponentReducer extends Reducer<Text, Text, Text, Text> {
  Configuration conf;

  public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
InterruptedException {
    conf = context.getConfiguration();

    Text finalValue = new Text("");
    // 取最后一行数据作为最终结果
    for (Text value : values) {
      finalValue = value;
    }

    // 将结果输出到HBase
    writeHBase(key.toString(), finalValue.toString());

    // 将结果保存到HDFS
    context.write(key, finalValue);
  }
```

样例5：结果输出到HBase的writeHBase方法。

```
  private void writeHBase(String rowKey, String data) {
    String tableName = "table1";
    String columnFamily = "cf";

    Configuration hbaseConfig = HBaseConfiguration.create(conf);
    org.apache.hadoop.hbase.client.Connection conn = null;
    try {
      // 创建HBase连接
      conn = ConnectionFactory.createConnection(hbaseConfig);
      // 获取HBase表
      Table table = conn.getTable(TableName.valueOf(tableName));

      // 创建一个HBase Put请求实例
      List<Put> list = new ArrayList<Put>();
      byte[] row = Bytes.toBytes("row" + rowKey);
      Put put = new Put(row);
      byte[] family = Bytes.toBytes(columnFamily);
      byte[] qualifier = Bytes.toBytes("value");
      byte[] value = Bytes.toBytes(data);
      put.addColumn(family, qualifier, value);
      list.add(put);
      // 执行Put请求
      table.put(list);
    } catch (IOException e) {
      LOG.warn("Exception occur ", e);
    } finally {
        if (conn != null) {
                try {
                    conn.close();
                } catch (Exception e1) {
                    LOG.error("Failed to close the connection ", e1);
                }
            }
        }
    }

  }
```

样例6：main()方法创建一个job，配置相关依赖，提交作业到hadoop集群。


```
public static void main(String[] args) throws Exception {
    String hiveClientProperties =
MultiComponentExample.class.getClassLoader().getResource("hiveclient.properties").getPath();
    // 包含配置信息的文件
    String file =  "file://" + hiveClientProperties;
    // 运行时，把配置信息放到HDFS上
    config.set("tmpfiles", file);
    // 提交作业前清理所需目录
    MultiComponentExample.cleanupBeforeRun();

    // 查找Hive运行依赖
    Class hiveDriverClass = Class.forName("org.apache.hive.jdbc.HiveDriver");
    Class thriftClass = Class.forName("org.apache.thrift.TException");
    Class serviceThriftCLIClass = Class.forName("org.apache.hive.service.rpc.thrift.TCLIService");
    Class hiveConfClass = Class.forName("org.apache.hadoop.hive.conf.HiveConf");
    Class hiveTransClass = Class.forName("org.apache.thrift.transport.HiveTSaslServerTransport");
    Class hiveMetaClass = Class.forName("org.apache.hadoop.hive.metastore.api.MetaException");
    Class hiveShimClass =
Class.forName("org.apache.hadoop.hive.metastore.security.HadoopThriftAuthBridge23");
    Class thriftCLIClass = Class.forName("org.apache.hive.service.cli.thrift.ThriftCLIService");
    Class thriftType = Class.forName("org.apache.hadoop.hive.serde2.thrift.Type");
    // 添加Hive依赖到作业

    JarFinderUtil.addDependencyJars(config, hiveDriverClass, serviceThriftCLIClass, thriftCLIClass, thriftClass,
            hiveConfClass, hiveTransClass, hiveMetaClass, hiveShimClass, thriftType);
    // 添加Hive配置文件
    config.addResource("hive-site.xml");
    // 添加HBase配置文件
    Configuration conf = HBaseConfiguration.create(config);

    // 实例化作业对象
    Job job = Job.getInstance(conf);
    job.setJarByClass(MultiComponentExample.class);

    // 配置mapper&reducer类
    job.setMapperClass(MultiComponentMapper.class);
    job.setReducerClass(MultiComponentReducer.class);

    // 配置数据输入路径和输出路径
    FileInputFormat.addInputPath(job, new Path(baseDir, INPUT_DIR_NAME + File.separator + "data.txt"));
    FileOutputFormat.setOutputPath(job, new Path(baseDir, OUTPUT_DIR_NAME));

    // 设置输出键值类型
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

    // HBase提供工具类添加HBase的运行依赖
    TableMapReduceUtil.addDependencyJars(job);

    // 提交作业
    System.exit(job.waitForCompletion(true) ? 0 : 1);

  }
```

24.6 调测 MapReduce 应用

24.6.1 在本地 Windows 环境中调测 MapReduce 应用

操作场景

在程序代码完成开发后，您可以在Windows环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

MapReduce应用程序运行完成后，可通过如下方式查看应用程序的运行情况。

- 在IntelliJ IDEA中查看应用程序运行情况。
- 通过MapReduce日志获取应用程序运行情况。
- 登录MapReduce WebUI查看应用程序运行情况。
- 登录Yarn WebUI查看应用程序运行情况。

说明

在MapReduce任务运行过程中禁止重启HDFS服务，否则可能会导致任务失败。

运行统计样例程序

步骤1 确保样例工程依赖的所有jar包已正常获取。

步骤2 在IntelliJ IDEA开发环境中，打开样例工程中“LocalRunner.java”工程，右键工程，选择“Run LocalRunner.main()”运行应用工程。

----结束

运行多组件样例程序

步骤1 将hive-site.xml、hbase-site.xml、hiveclient.properties放入工程的conf目录。

步骤2 确保样例工程依赖的所有Hive、HBase相关jar包已正常获取。

步骤3 打开MultiComponentLocalRunner.java，确认代码中System.setProperty("HADOOP_USER_NAME", "root");设置了用户为root，请确保场景说明中上传的数据的用户为root，或者在代码中将root修改为上传数据的用户名。

步骤4 在IntelliJ IDEA开发环境中，选中“MultiComponentLocalRunner.java”工程，单击运行对应的应用程序工程。或者右键工程，选择“Run MultiComponentLocalRunner.main()”运行应用工程。

----结束

查看调测结果

- **查看运行结果获取应用运行情况**

  如下所示，通过控制台输出结果查看应用运行情况。

```
3614 [main] INFO  org.apache.hadoop.hdfs.PeerCache  - SocketCache disabled.
10159 [main] INFO  org.apache.hadoop.mapreduce.lib.input.FileInputFormat  - Total input files to process : 2
11378 [main] INFO  org.apache.hadoop.mapreduce.JobSubmitter  - number of splits:2
12707 [main] INFO  org.apache.hadoop.mapreduce.JobSubmitter  - Submitting tokens for job: job_1468241424339_0002
16434 [main] INFO  org.apache.hadoop.yarn.client.api.impl.YarnClientImpl  - Submitted application application_1468241424339_0002
16656 [main] INFO  org.apache.hadoop.mapreduce.Job  - The url to track the job: http://10-120-180-170:8088/proxy/application_1468241424339_0002/
16657 [main] INFO  org.apache.hadoop.mapreduce.Job  - Running job: job_1468241424339_0002
31177 [main] INFO  org.apache.hadoop.mapreduce.Job  - Job job_1468241424339_0002 running in uber mode : false
31200 [main] INFO  org.apache.hadoop.mapreduce.Job  -  map 0% reduce 0%
45893 [main] INFO  org.apache.hadoop.mapreduce.Job  -  map 100% reduce 0%
57172 [main] INFO  org.apache.hadoop.mapreduce.Job  -  map 100% reduce 100%
58554 [main] INFO  org.apache.hadoop.mapreduce.Job  - Job job_1468241424339_0002 completed successfully
58908 [main] INFO  org.apache.hadoop.mapreduce.Job  - Counters: 49
File System Counters
FILE: Number of bytes read=75
```



```
FILE: Number of bytes written=436979
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=674
HDFS: Number of bytes written=23
HDFS: Number of read operations=9
HDFS: Number of large read operations=0
HDFS: Number of write operations=2
Job Counters
Launched map tasks=2
Launched reduce tasks=1
Data-local map tasks=2
Total time spent by all maps in occupied slots (ms)=206088
Total time spent by all reduces in occupied slots (ms)=73824
Total time spent by all map tasks (ms)=25761
Total time spent by all reduce tasks (ms)=9228
Total vcore-seconds taken by all map tasks=25761
Total vcore-seconds taken by all reduce tasks=9228
Total megabyte-seconds taken by all map tasks=105517056
Total megabyte-seconds taken by all reduce tasks=37797888
Map-Reduce Framework
Map input records=26
Map output records=16
Map output bytes=186
Map output materialized bytes=114
Input split bytes=230
Combine input records=16
Combine output records=6
Reduce input groups=3
Reduce shuffle bytes=114
Reduce input records=6
Reduce output records=2
Spilled Records=12
Shuffled Maps =2
Failed Shuffles=0
Merged Map outputs=2
GC time elapsed (ms)=356
CPU time spent (ms)=2860
Physical memory (bytes) snapshot=1601576960
Virtual memory (bytes) snapshot=12999819264
Total committed heap usage (bytes)=2403336192
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=444
File Output Format Counters
Bytes Written=23
```

说明

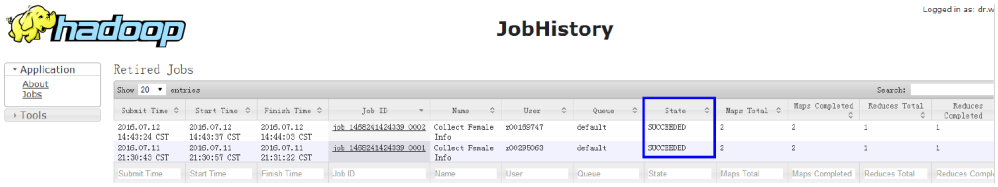
在Windows环境运行样例代码时会出现下面的异常，但是不影响业务：

java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.

- **通过MapReduce服务的WebUI进行查看**

  使用具有任务查看权限的用户登录FusionInsight Manager，单击“集群 > *待操作集群的名称* > 服务 > Mapreduce > JobHistoryServer”进入Web界面后查看任务执行状态。

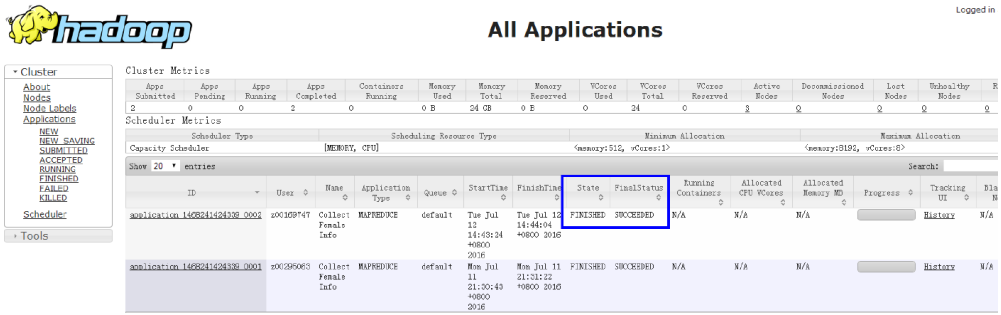
图 24-8 JobHistory Web UI 界面



- **通过YARN服务的WebUI进行查看**

  使用具有任务查看权限的用户登录FusionInsight Manager，单击“集群 > *待操作集群的名称* > 服务 > Yarn > ResourceManager(主)”进入Web界面后查看任务执行状态。

图 24-9 ResourceManager Web UI 页面



- **查看MapReduce日志获取应用运行情况**

  您可以查看MapReduce日志了解应用运行情况，并根据日志信息调整应用程序。

24.6.2 在 Linux 环境中调测 MapReduce 应用

操作场景

在程序代码完成开发后，可以在Linux环境中运行应用。

MapReduce应用程序运行完成后，可通过如下方式查看应用程序的运行情况。

- 通过运行结果查看程序运行情况。
- 登录MapReduce WebUI查看应用程序运行情况。
- 登录Yarn WebUI查看应用程序运行情况
- 通过MapReduce日志获取应用程序运行情况。

运行程序

步骤1 进入样例工程本地根目录，在Windows命令提示符窗口中执行下面命令进行打包。

mvn -s "*{maven_setting_path}*" clean package

说明

- 上述打包命令中的{maven_setting_path}为本地Maven的setting.xml文件路径。
- 打包成功之后，在工程根目录的target子目录下获取打好的jar包，例如“MRTest-XXX.jar”，jar包名称以实际打包结果为准。

步骤2 上传生成的应用包“MRTest-XXX.jar”到Linux客户端上，例如**/opt/client/conf**，与配置文件位于同一目录下。

步骤3 在Linux环境下运行样例工程。

- 对于MapReduce统计样例程序，执行如下命令。

  yarn jar MRTest-XXX.jar com.huawei.bigdata.mapreduce.examples.FemaleInfoCollector *<inputPath> <outputPath>*

  此命令包含了设置参数和提交job的操作，其中*<inputPath>*指HDFS文件系统中input的路径，*<outputPath>*指HDFS文件系统中output的路径。

  说明

  - 在执行**yarn jar MRTest-XXX.jar com.huawei.bigdata.mapreduce.examples.FemaleInfoCollector** *<inputPath> <outputPath>*命令之前，需要把log1.txt和log2.txt这两个文件上传到HDFS的<inputPath>目录下。参考**MapReduce统计样例程序开发思路**。
  - 在执行**yarn jar MRTest-XXX.jar com.huawei.bigdata.mapreduce.examples.FemaleInfoCollector** *<inputPath> <outputPath>*命令之前，<outputPath>目录必须不存在，否则会报错。
  - 在MapReduce任务运行过程中禁止重启HDFS服务，否则可能会导致任务失败。

- 对于MapReduce访问多组件样例程序，操作步骤如下。

  a. 获取“hbase-site.xml”、“hiveclient.properties”和“hive-site.xml”这三个配置文件，并在Linux环境上创建文件夹保存这三个配置文件，例如“/opt/client/conf”。

  说明

  “hbase-site.xml”从HBase客户端获取，“hiveclient.properties”和“hive-site.xml”从Hive客户端获取。

  b. 在Linux环境中添加样例工程运行所需的classpath，例如

  export YARN_USER_CLASSPATH=/opt/client/conf:/opt/client/HBase/hbase/lib/*:/opt/client/HBase/hbase/lib/client-facing-thirdparty/*:/opt/client/Hive/Beeline/lib/*

  c. 提交MapReduce任务，执行如下命令，运行样例工程。

  yarn jar MRTest-XXX.jar com.huawei.bigdata.mapreduce.examples.MultiComponentExample

----结束

查看调测结果

- **通过MapReduce服务的WebUI进行查看**

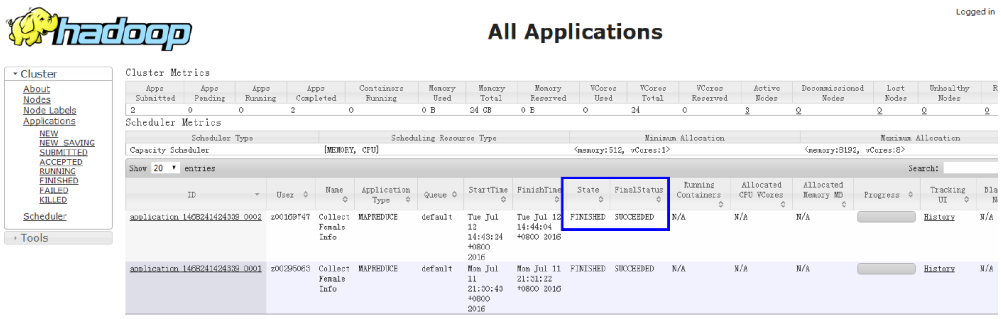
  使用具有任务查看权限的用户登录FusionInsight Manager，单击“集群 > *待操作集群的名称* > 服务 > Mapreduce > JobHistoryServer”进入Web界面后查看任务执行状态。

图 24-10 JobHistory Web UI 界面

- **通过YARN服务的WebUI进行查看**

  使用具有任务查看权限的用户登录FusionInsight Manager，单击“集群 > *待操作集群的名称* > 服务 > Yarn > ResourceManager(主)”进入Web界面后查看任务执行状态。

  图 24-11 ResourceManager Web UI 页面



- **查看MapReduce应用运行结果数据。**
  - 当用户在Linux环境下执行**yarn jar MRTest-*XXX*.jar**命令后，可以通过执行结果显示正在执行的应用的运行情况。例如：

```
yarn jar MRTest-XXX.jar /tmp/mapred/example/input/ /tmp/root/output/1
16/07/12 17:07:16 INFO hdfs.PeerCache: SocketCache disabled.
16/07/12 17:07:17 INFO input.FileInputFormat: Total input files to process : 2
16/07/12 17:07:18 INFO mapreduce.JobSubmitter: number of splits:2
16/07/12 17:07:18 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_1468241424339_0006
16/07/12 17:07:18 INFO impl.YarnClientImpl: Submitted application
application_1468241424339_0006
16/07/12 17:07:18 INFO mapreduce.Job: The url to track the job: http://10-120-180-170:8088/
proxy/application_1468241424339_0006/
16/07/12 17:07:18 INFO mapreduce.Job: Running job: job_1468241424339_0006
16/07/12 17:07:31 INFO mapreduce.Job: Job job_1468241424339_0006 running in uber mode :
false
16/07/12 17:07:31 INFO mapreduce.Job:  map 0% reduce 0%
16/07/12 17:07:41 INFO mapreduce.Job:  map 50% reduce 0%
16/07/12 17:07:43 INFO mapreduce.Job:  map 100% reduce 0%
16/07/12 17:07:51 INFO mapreduce.Job:  map 100% reduce 100%
16/07/12 17:07:51 INFO mapreduce.Job: Job job_1468241424339_0006 completed successfully
16/07/12 17:07:51 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=75
                FILE: Number of bytes written=435659
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=674
                HDFS: Number of bytes written=23
                HDFS: Number of read operations=9
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=2
                Launched reduce tasks=1
                Data-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=144984
                Total time spent by all reduces in occupied slots (ms)=56280
                Total time spent by all map tasks (ms)=18123
                Total time spent by all reduce tasks (ms)=7035
                Total vcore-milliseconds taken by all map tasks=18123
                Total vcore-milliseconds taken by all reduce tasks=7035
                Total megabyte-milliseconds taken by all map tasks=74231808
                Total megabyte-milliseconds taken by all reduce tasks=28815360
        Map-Reduce Framework
                Map input records=26
```



```
        Map output records=16
        Map output bytes=186
        Map output materialized bytes=114
        Input split bytes=230
        Combine input records=16
        Combine output records=6
        Reduce input groups=3
        Reduce shuffle bytes=114
        Reduce input records=6
        Reduce output records=2
        Spilled Records=12
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=202
        CPU time spent (ms)=2720
        Physical memory (bytes) snapshot=1595645952
        Virtual memory (bytes) snapshot=12967759872
        Total committed heap usage (bytes)=2403860480
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=444
    File Output Format Counters
        Bytes Written=23
```

- 在Linux环境下执行***yarn application -status <ApplicationID>***，可以通过执行结果显示正在执行的应用的运行情况。例如：

```
yarn application -status application_1468241424339_0006
Application Report :
    Application-Id : application_1468241424339_0006
    Application-Name : Collect Female Info
    Application-Type : MAPREDUCE
    User : root
    Queue : default
    Start-Time : 1468314438442
    Finish-Time : 1468314470080
    Progress : 100%
    State : FINISHED
    Final-State : SUCCEEDED
    Tracking-URL : http://10-120-180-170:19888/jobhistory/job/job_1468241424339_0006
    RPC Port : 27100
    AM Host : 10-120-169-46
    Aggregate Resource Allocation : 172153 MB-seconds, 64 vcore-seconds
    Log Aggregation Status : SUCCEEDED
    Diagnostics : Application finished execution.
    Application Node Label Expression : <Not set>
    AM container Node Label Expression : <DEFAULT_PARTITION>
```

- **查看MapReduce日志获取应用运行情况。**

  您可以查看MapReduce日志了解应用运行情况，并根据日志信息调整应用程序。

24.7 MapReduce 应用开发常见问题

24.7.1 MapReduce 接口介绍

24.7.1.1 MapReduce Java API 接口介绍

关于MapReduce的详细API可以直接参考官方网站上的描述：

http://hadoop.apache.org/docs/r3.1.1/api/index.html

常用接口

MapReduce中常见的类如下：

- org.apache.hadoop.mapreduce.Job：用户提交MR作业的接口，用于设置作业参数、提交作业、控制作业执行以及查询作业状态。
- org.apache.hadoop.mapred.JobConf：MapReduce作业的配置类，是用户向Hadoop提交作业的主要配置接口。

表 24-4 类 org.apache.hadoop.mapreduce.Job 的常用接口

功能	说明
Job(Configuration conf, String jobName)，Job(Configuration conf)	新建一个MapReduce客户端，用于配置作业属性，提交作业。
setMapperClass(Class<extends Mapper> cls)	核心接口，指定MapReduce作业的Mapper类，默认为空。也可以在“mapred-site.xml”中配置“mapreduce.job.map.class”项。
setReducerClass(Class<extends Reducer> cls)	核心接口，指定MapReduce作业的Reducer类，默认为空。也可以在“mapred-site.xml”中配置“mapreduce.job.reduce.class”项。
setCombinerClass(Class<extends Reducer> cls)	指定MapReduce作业的Combiner类，默认为空。也可以在“mapred-site.xml”中配置“mapreduce.job.combine.class”项。需要保证reduce的输入输出key，value类型相同才可以使用，谨慎使用。
setInputFormatClass(Class<extends InputFormat> cls)	核心接口，指定MapReduce作业的InputFormat类，默认为TextInputFormat。也可以在“mapred-site.xml”中配置“mapreduce.job.inputformat.class”项。该设置用来指定处理不同格式的数据时需要的InputFormat类，用来读取数据，切分数据块。
setJarByClass(Class< > cls)	核心接口，指定执行类所在的jar包本地位置。java通过class文件找到执行jar包，该jar包被上传到HDFS。
setJar(String jar)	指定执行类所在的jar包本地位置。直接设置执行jar包所在位置，该jar包被上传到HDFS。与setJarByClass(Class< > cls)选择使用一个。也可以在“mapred-site.xml”中配置“mapreduce.job.jar”项。
setOutputFormatClass(Class<extends OutputFormat> theClass)	核心接口，指定MapReduce作业的OutputFormat类，默认为TextOutputFormat。也可以在“mapred-site.xml”中配置“mapred.output.format.class”项，指定输出结果的数据格式。例如默认的TextOutputFormat把每条key，value记录写为文本行。通常场景不配置特定的OutputFormat。

功能	说明
setOutputKeyClass(Class< > theClass)	核心接口，指定MapReduce作业的输出key的类型，也可以在“mapred-site.xml”中配置“mapreduce.job.output.key.class”项。
setOutputValueClass(Class < > theClass)	核心接口，指定MapReduce作业的输出value的类型，也可以在“mapred-site.xml”中配置“mapreduce.job.output.value.class”项。
setPartitionerClass(Class<extends Partitioner> theClass)	指定MapReduce作业的Partitioner类。也可以在“mapred-site.xml”中配置“mapred.partitioner.class”项。该方法用来分配map的输出结果到哪个reduce类，默认使用HashPartitioner，均匀分配map的每条键值对记录。例如在hbase应用中，不同的键值对应的region不同，这就需要设定特殊的partitioner类分配map的输出结果。
setSortComparatorClass(Class<extends RawComparator> cls)	指定MapReduce作业的map任务的输出结果压缩类，默认不使用压缩。也可以在“mapred-site.xml”中配置“mapreduce.map.output.compress”和“mapreduce.map.output.compress.codec”项。当map的输出数据大，减少网络压力，使用压缩传输中间数据。
setPriority(JobPriority priority)	指定MapReduce作业的优先级，共有5个优先级别，VERY_HIGH、HIGH、NORMAL、LOW、VERY_LOW，默认级别为NORMAL。也可以在“mapred-site.xml”中配置“mapreduce.job.priority”项。

表 24-5 类 org.apache.hadoop.mapred.JobConf 的常用接口

方法	说明
setNumMapTasks(int n)	核心接口，指定MapReduce作业的map个数。也可以在“mapred-site.xml”中配置“mapreduce.job.maps”项。 **说明** 指定的InputFormat类用来控制map任务个数，注意该类是否支持客户端设定map个数。
setNumReduceTasks(int n)	核心接口，指定MapReduce作业的reduce个数。默认只启动1个。也可以在“mapred-site.xml”中配置“mapreduce.job.reduces”项。reduce个数由用户控制，通常场景reduce个数是map个数的1/4。

方法	说明
setQueueName(String queueName)	指定MapReduce作业的提交队列。默认使用default队列。也可以在“mapred-site.xml”中配置“mapreduce.job.queuename”项。

24.7.1.2 MapReduce REST API 接口介绍

功能简介

通过HTTP REST API来查看更多MapReduce任务的信息。目前Mapresuce的REST接口可以查询已完成任务的状态信息。完整和详细的接口请直接参考官网上的描述以了解其使用：

http://hadoop.apache.org/docs/r3.1.1/hadoop-mapreduce-client/hadoop-mapreduce-client-hs/HistoryServerRest.html

准备运行环境

1. 在节点上安装客户端，例如安装到“/opt/client”目录，可参考“安装客户端”。
2. 进入客户端安装目录“/opt/client”，执行下列命令初始化环境变量。

source *bigdata_env*

操作步骤

获取MapReduce上已完成任务的具体信息

- 命令：
  ```
  curl -k -i --negotiate -u : "http://10.120.85.2:19888/ws/v1/history/mapreduce/jobs"
  ```
  其中10.120.85.2为MapReduce的“JHS_FLOAT_IP”参数的参数值，19888为JobHistoryServer的端口号。

  说明

  在Red Hat 6.x以及CentOS 6.x版本，使用curl命令访问JobHistoryServer会有兼容性问题，导致无法返回正确结果。
- 用户能看到历史任务的状态信息（任务id，开始时间，结束时间，是否执行成功等信息）
- 运行结果
  ```
  {
    "jobs":{
      "job":[
        {
          "submitTime":1525693184360,
          "startTime":1525693194840,
          "finishTime":1525693215540,
          "id":"job_1525686535456_0001",
          "name":"QuasiMonteCarlo",
          "queue":"default",
          "user":"mapred",
          "state":"SUCCEEDED",
          "mapsTotal":1,
          "mapsCompleted":1,
          "reducesTotal":1,
          "reducesCompleted":1
        }
  ```

```
    ]
  }
}
```

- 结果分析：

  通过这个接口，可以查询当前集群中已完成的MapReduce任务，并且可以得到**表 24-6**

  表 24-6 常用信息

参数	参数描述
submitTime	任务提交时间
startTime	任务开始执行时间
finishTime	任务执行完成时间
queue	任务队列
user	提交这个任务的用户
state	任务执行成功或失败

24.7.2 提交 MapReduce 任务时客户端长时间无响应

问题

向YARN服务器提交MapReduce任务后，客户端长时间无响应。

回答

对于上述出现的问题，ResourceManager在其WebUI上提供了MapReduce作业关键步骤的诊断信息，对于一个已经提交到YARN上的MapReduce任务，用户可以通过该诊断信息获取当前作业的状态以及处于该状态的原因。

具体操作：登录FusionInsight Manager，单击“集群 > *待操作集群的名称* > 服务 > Yarn > ResourceManager(主)”打开WebUI界面，在ResourceManager(主)的WebUI界面中，单击提交的MapReduce任务，在打开的页面中查看诊断信息，根据诊断信息再采取相应的措施。

或者也可以通过查看MapReduce日志了解应用运行情况，并根据日志信息调整应用程序。

24.7.3 MapReduce 二次开发远程调试

问题

MapReduce二次开发过程中如何远程调试业务代码?

回答

MapReduce开发调试采用的原理是Java的远程调试机制，在Map/Reduce任务启动时，添加Java远程调试命令。

步骤1 首先理解两个参数：“mapreduce.map.java.opts”和“mapreduce.reduce.java.opts”，这两个参数为客户端参数，分别指定了Map/Reduce任务对应的JVM启动参数。

修改客户端“mapred-site.xml”配置文件中“mapreduce.map.java.opts”和“mapreduce.reduce.java.opts”参数，分别加入调试命令“-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8000”，保存文件。

步骤2 MapReduce为分布式计算框架，Map/Reduce任务启动所在的节点存在不确定性，建议将集群内NodeManager实例只保留一个运行，其他全部停止，以保证任务一定会在这个唯一运行的NodeManager节点上启动。

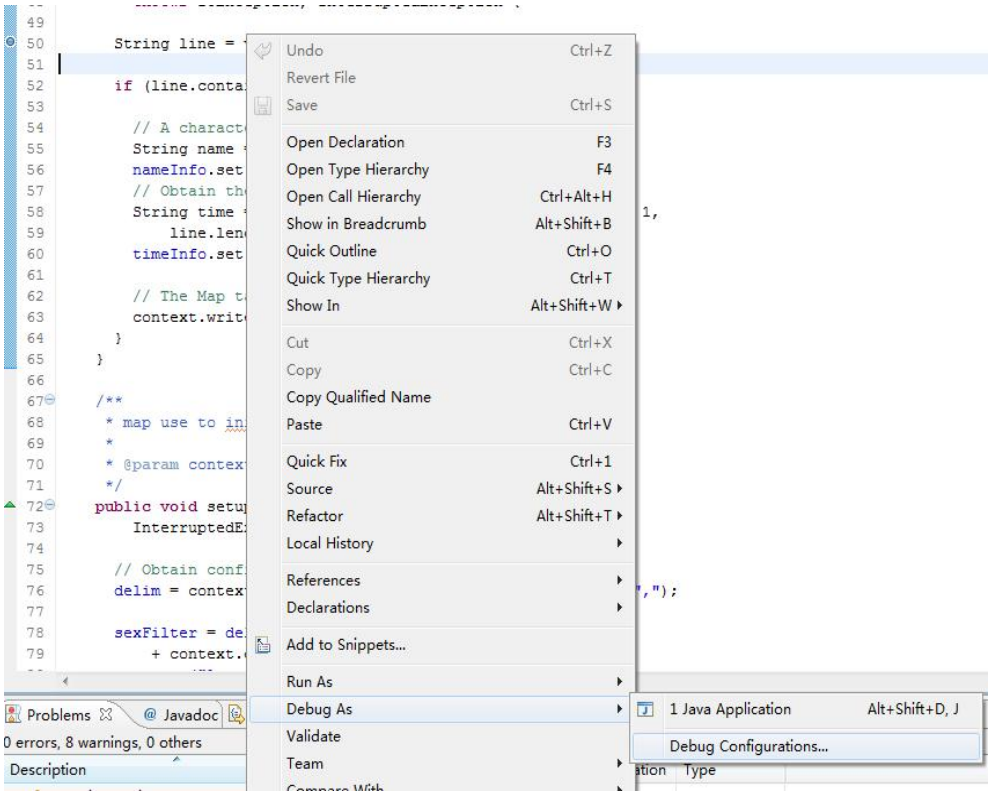
步骤3 在客户端提交MapReduce任务，在Map/Reduce任务启动时会挂起并监测8000端口，等待远程调试。

步骤4 在IDE上，选择MapReduce任务的实现类，通过配置远程调试信息，执行Debug。

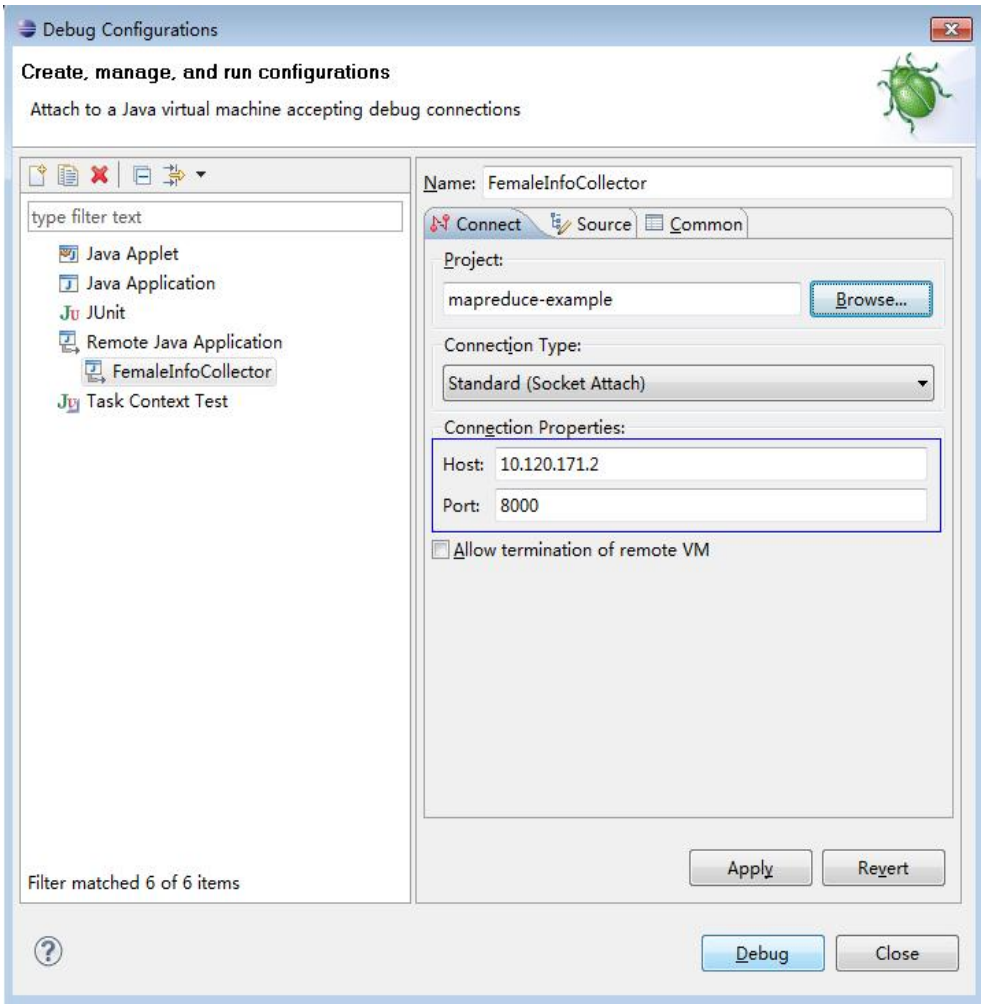
1. 设置断点，双击蓝框区域设置或取消断点。

```
39    /**
40     * Distributed computing
41     *
42     * @param key     Object : location offset of the source file
43     * @param value   Text : a row of characters in the source file
44     * @param context Context : output parameter
45     * @throws IOException , InterruptedException
46     */
47    public void map(Object key, Text value, Context context)
48        throws IOException, InterruptedException {
49
50      String line = value.toString();
51
52      if (line.contains(sexFilter)) {
53
54        // A character string that has been read
55        String name = line.substring(0, line.indexOf(delim));
56        nameInfo.set(name);
57        // Obtain the dwell duration.
58        String time = line.substring(line.lastIndexOf(delim) + 1,
59            line.length());
60        timeInfo.set(Integer.parseInt(time));
61
62        // The Map task outputs a key-value pair.
63        context.write(nameInfo, timeInfo);
64      }
65    }
66
```

2. 配置远程调试信息，“右键->Debug As->Debug Configurations...”。



3. 在弹出的页面，双击“Remote Java Application”，设置Connection Properties，其中Host为运行的NodeManager节点IP，Port端口号为8000，然后单击“Debug”。



----结束

说明

若使用IDE直接提交MapReduce任务，则IDE即成为客户端的角色，参考**步骤1**修改二次开发工程中的“mapred-site.xml”即可。

25 Oozie 开发指南（安全模式）

25.1 Oozie 应用开发概述

25.1.1 Oozie 应用开发应用开发简介

Oozie 简介

Oozie是一个用来管理Hadoop任务的工作流引擎，Oozie流程基于有向无环图（Directed Acyclical Graph）来定义和描述，支持多种工作流模式及流程定时触发机制。易扩展、易维护、可靠性高，与Hadoop生态系统各组件紧密结合。

Oozie流程的三种类型：

- Workflow

  描述一个完整业务的基本流程。
- Coordinator

  Coordinator流程构建在Workflow流程之上，实现了对Workflow流程的定时触发、按条件触发功能。
- Bundle

  Bundle流程构建在Coordinator流程之上，提供对多个Coordinator流程的统一调度、控制和管理功能。

Oozie主要特点：

- 支持分发、聚合、选择等工作流程模式。
- 与Hadoop生态系统各组件紧密结合。
- 流程变量支持参数化。
- 支持流程定时触发。
- 自带一个Web Console，提供了流程查看、流程监控、日志查看等功能。

25.1.2 Oozie 应用开发常用概念

- **流程定义文件**

描述业务逻辑的XML文件，包括“workflow.xml”、“coordinator.xml”、“bundle.xml”三类，最终由Oozie引擎解析并执行。

- **流程属性文件**

  流程运行期间的参数配置文件，对应文件名为“job.properties”，每个流程定义有且仅有一个该属性文件。

- **keytab文件**

  存放用户信息的密钥文件。在安全模式下，应用程序采用此密钥文件进行API方式认证。

- **Client**

  客户端直接面向用户，可通过Java API、Shell API、 REST API或者Web UI访问Oozie服务端。

- **Oozie WebUI界面**

  通过“https://*Oozie服务器IP*地址:21003/oozie”登录Oozie WebUI界面。

25.1.3 Oozie 应用开发流程

本文档主要基于java API对Oozie进行应用开发。

开发流程中各阶段的说明如**图25-1**和**表25-1**所示。

图 25-1 Oozie 应用程序开发流程

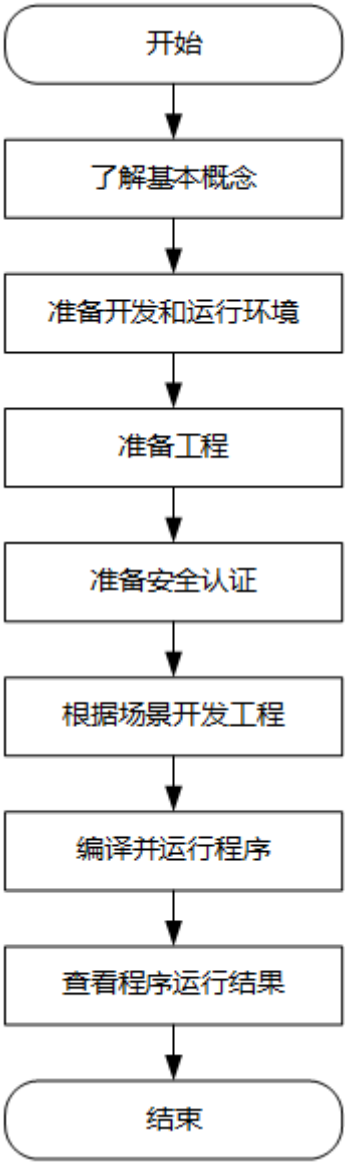


表 25-1 Oozie 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Oozie的基本概念，了解场景需求等。	**Oozie应用开发常用概念**
准备开发和运行环境	Oozie的应用程序当前推荐使用Java语言进行开发。可使用IDEA工具。	**准备本地应用开发环境**
准备工程	Oozie提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。	**导入并配置Oozie样例工程**

阶段	说明	参考文档
准备安全认证	如果您使用的是安全集群，需要进行安全认证。	配置Oozie应用安全认证
根据场景开发工程	提供了Java语言的样例工程。	开发Oozie应用
编译并运行程序	指导用户将开发好的程序编译并提交运行。	调测Oozie应用
查看程序运行结果	程序运行结果会写在用户指定的路径下。用户还可以通过UI查看应用运行情况。	调测Oozie应用

25.1.4 Oozie 应用开发样例工程介绍

MRS样例工程获取地址为https://github.com/huaweicloud/huaweicloud-mrs-example，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Oozie相关样例工程：

表 25-2 Oozie 相关样例工程

样例工程位置	描述
oozie-examples/ooziesecurity-examples/OozieMapReduceExample	Oozie提交MapReduce任务示例程序。 本示例演示了如何通过Java API提交MapReduce作业和查询作业状态，对网站的日志文件进行离线分析。
oozie-examples/ooziesecurity-examples/OozieSparkHBaseExample	使用Oozie调度Spark访问HBase的示例程序。
oozie-examples/ooziesecurity-examples/OozieSparkHiveExample	使用Oozie调度Spark访问Hive的示例程序。

25.2 准备 Oozie 应用开发环境

25.2.1 准备本地应用开发环境

在进行二次开发时，要准备的开发和运行环境如表25-3所示。

表 25-3 开发环境

准备项	说明
操作系统	Windows系统，支持Windows 7以上版本。 开发和运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 ● X86客户端：Oracle JDK，支持1.8版本；IBM JDK，支持1.8.5.11版本。 ● TaiShan客户端：OpenJDK，支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。
安装和配置IDEA	开发环境的基本配置，建议使用2019.1或其他兼容版本。 **说明** ● 若使用IBM JDK，请确保IDEA中的JDK配置为IBM JDK。 ● 若使用Oracle JDK，请确保IDEA中的JDK配置为Oracle JDK。 ● 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 ● 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
准备开发用户	参考**准备MRS应用开发用户**进行操作，准备用于应用开发的集群用户并授予相应权限。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-zip 16.04版本。

25.2.2 导入并配置 Oozie 样例工程

操作场景

将下载的样例工程导入到Windows开发环境IDEA中即可开始样例学习。

前提条件

- 已按照**准备本地应用开发环境**章节准备好开发用户，例如**developuser**，并下载用户的认证凭据文件到本地。
  用户需要具备Oozie的普通用户权限，HDFS访问权限，Hive表读写权限，HBase读写权限以及Yarn的队列提交权限。
- 已在Linux环境中安装了完整的集群客户端。
- 获取Oozie服务器URL（任意节点），这个URL将是客户端提交流程任务的目标地址。
  URL格式为：https://*Oozie节点业务IP*:21003/oozie。可登录FusionInsight Manager，选择“集群 > 服务 > Oozie > 实例”，即可获取任一oozie实例的IP地

址；单击“配置”，在搜索框中搜索“OOZIE_HTTPS_PORT”，即可查看使用的端口号。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\oozie-examples”目录下的样例工程文件夹“ooziesecurity-examples”中的OozieMapReduceExample、OozieSparkHBaseExample和OozieSparkHiveExample三个样例工程。

步骤2 将**准备MRS应用开发用户**时得到的keytab文件“user.keytab”和“krb5.conf”用户认证凭据文件复制到OozieMapReduceExample、OozieSparkHBaseExample和OozieSparkHiveExample样例工程的“\src\main\resources”路径。

步骤3 在应用开发环境中，导入样例工程到IDEA开发环境。

1. 在IDEA中选择“File > Open”，弹出“浏览文件夹”对话框。
2. 选择样例工程文件夹，单击“OK”。

步骤4 修改样例工程中的如下参数，请参考**表25-4**。

表 25-4 文件参数修改列表

文件名	参数名	值	取值样例
\src\main\resources\job.properties	userName	提交作业的用户	developuser
\src\main\resources\application.properties	submit_user	提交作业的用户	developuser
	oozie_url_default	https://*Oozie业务IP*:21003/oozie	https://10.10.10.176:21003/oozie

步骤5 选择运行的样例工程：

- OozieMapReduceExcample样例工程，执行**步骤6**。
- OozieSparkHBaseExample和OozieSparkHiveExample样例工程，请参考**使用Oozie调度Spark2x访问HBase以及Hive**。

步骤6 使用客户端上传Oozie的examples文件夹到HDFS。

1. 登录客户端所在节点，切换到客户端所在目录，例如“/opt/client”。

   cd /opt/client
2. 执行以下命令配置环境变量。

   source bigdata_env
3. 执行以下命令认证用户并登录。首次登录需要修改密码。

   kinit developuser
4. 执行以下命令在HDFS创建目录并上传样例工程到该目录。

   hdfs dfs -mkdir /user/developuser

   hdfs dfs -put -f /opt/client/Oozie/oozie-client-*/examples /user/developuser

说明

命令行中 “oozie-client-*”涉及的版本号以实际版本号为准。

----结束

25.2.3 配置 Oozie 应用安全认证

场景说明

在安全集群环境下，各个组件需要在通信之前进行相互认证，以确保通信的安全性。

用户在开发Oozie应用程序时，某些场景下需要Oozie与Hadoop、Hive等之间进行通信。那么Oozie应用程序中需要写入安全认证代码，确保Oozie程序能够正常运行。

安全认证有两种方式：

- 命令行认证：

  提交Oozie应用程序运行前，在Oozie客户端执行如下命令获得认证。

  kinit *组件业务用户*

- 代码认证（Kerberos安全认证）：

  通过获取客户端的principal和keytab文件在应用程序中进行认证，用于Kerberos安全认证的keytab文件和principal文件您可以联系管理员创建并获取，具体使用方法在样例代码中会有详细说明。

  目前样例代码统一调用LoginUtil类进行安全认证，支持Oracle JAVA平台和IBM JAVA平台。

  代码示例中请根据实际情况，修改“USERNAME”为实际用户名，例如“developuser”。

```
private static void login(String keytabFilePath, String krb5FilePath, String user) throws IOException {
    Configuration conf = new Configuration();
    conf.set(KERBEROS_PRINCIPAL, user);
    conf.set(KEYTAB_FILE, keytabFilePath);
    conf.set(HADOOP_SECURITY_AUTHENTICATION, "kerberos");
    conf.set(HADOOP_SECURITY_AUTHORIZATION, "true");

    /*
     * if need to connect zk, please provide jaas info about zk. of course,
     * you can do it as below:
     * System.setProperty("java.security.auth.login.config", confDirPath +
     * "jaas.conf"); but the demo can help you more : Note: if this process
     * will connect more than one zk cluster, the demo may be not proper. you
     * can contact us for more help
     */
    LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, user, keytabFilePath);
    LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);
    LoginUtil.login(user, keytabFilePath, krb5FilePath, conf);
}
```

25.3 开发 Oozie 应用

25.3.1 开发 Oozie 配置文件

25.3.1.1 Oozie 样例程序开发思路

开发流程

1. 工作流配置文件“workflow.xml”（“coordinator.xml”是对工作流进行调度，“bundle.xml”是对一组Coordinator进行管理）与“job.properties”。
2. 如果有实现代码，需要开发对应的jar包，例如Java Action；如果是Hive，则需要开发SQL文件。
3. 上传配置文件与jar包（包括依赖的jar包）到HDFS，上传的路径取决于“workflow.xml”中的“oozie.wf.application.path”参数配置的路径。
4. 提供三种方式对工作流进行操作，详情请参见**Oozie应用开发常见问题**。
   - Shell命令
   - Java API
   - Hue
5. Oozie客户端提供了比较完整的examples示例供用户参考，包括各种类型的Action，以及Coordinator以及Bundle的使用。以客户端安装目录为“/opt/client”为例，examples具体目录为“/opt/client/Oozie/oozie-client-*/examples”。

如下通过一个MapReduce工作流的示例演示如何配置，并通过Shell命令调用。

场景说明

假设存在这样的业务需求：

每天需要对网站的日志文件进行离线分析，统计出网站各模块的访问频率（日志文件存放在HDFS中）。

通过客户端中模板与配置文件提交任务。

25.3.1.2 Oozie 应用开发步骤

步骤1 业务分析。

1. 可以使用客户端样例目录中MapReduce程序对日志目录的数据进行分析、处理。
2. 将MapReduce程序的分析结果移动到数据分析结果目录，并将数据文件的权限设置成**660**。
3. 为了满足每天分析一次的需求，需要每天重复执行一次**步骤1.1**～**步骤1.2**。

步骤2 业务实现。

1. 登录Oozie客户端所在节点，新建“dataLoad”目录，作为程序运行目录，后面编写的文件均保存在该目录下。例如“/opt/client/Oozie/oozie-client-*/examples/apps/dataLoad/”。

   说明

   可以直接复制样例目录中“map-reduce”文件夹内的内容到“dataLoad”文件夹，然后进行编辑。

   目录中 “oozie-client-*”涉及的版本号以实际版本号为准。

2. 编写流程任务属性文件（job.properties）。

   请参见**配置Oozie作业运行参数**。

3. 编写Workflow任务文件“workflow.xml”。

表 25-5 流程 Action

编号	步骤	描述
1	定义startaction	请参见**配置Oozie作业执行入口**
2	定义 MapReduceaction	请参见**配置Oozie MapReduce作业**
3	定义FS action	请参见**配置Oozie作业操作HDFS文件**
4	定义end action	请参见**配置Oozie作业执行终点**
5	定义killaction	请参见**配置Oozie作业异常结束打印信息**

说明

依赖或新开发的jar包需要放在“dataLoad/lib”目录下。

流程文件样例：

```
<workflow-app xmlns="uri:oozie:workflow:1.0" name="data_load">
  <start to="mr-dataLoad"/>
  <action name="mr-dataLoad">
     <map-reduce>
        <resource-manager>${resourceManager}</resource-manager>
        <name-node>${nameNode}</name-node>
        <prepare>
           <delete path="${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-
reduce"/>
        </prepare>
        <configuration>
           <property>
              <name>mapred.job.queue.name</name>
              <value>${queueName}</value>
           </property>
           <property>
              <name>mapred.mapper.class</name>
              <value>org.apache.oozie.example.SampleMapper</value>
           </property>
           <property>
              <name>mapred.reducer.class</name>
              <value>org.apache.oozie.example.SampleReducer</value>
           </property>
           <property>
              <name>mapred.map.tasks</name>
              <value>1</value>
           </property>
           <property>
              <name>mapred.input.dir</name>
              <value>/user/oozie/${dataLoadRoot}/input-data/text</value>
           </property>
           <property>
              <name>mapred.output.dir</name>
              <value>/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce</value>
           </property>
        </configuration>
     </map-reduce>
     <ok to="copyData"/>
     <error to="fail"/>
  </action>
```



```
  <action name="copyData">
      <fs>
         <delete path='${nameNode}/user/oozie/${dataLoadRoot}/result'/>
         <move source='${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce'
               target='${nameNode}/user/oozie/${dataLoadRoot}/result'/>
         <chmod path='${nameNode}/user/oozie/${dataLoadRoot}/result' permissions='-rwxrw-rw-'
dir-files='true'></chmod>
      </fs>
      <ok to="end"/>
      <error to="fail"/>
  </action>

  <kill name="fail">
      <message>This workflow failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</
message>
  </kill>
  <end name="end"/>
</workflow-app>
```

4. 编写Coordinator任务文件“coordinator.xml”。

   完成每天一次的定时数据分析工作，请参见**配置Coordinator定时调度作业**。

步骤3 上传流程文件。

1. 使用或切换到拥有HDFS上传权限的用户，准备用户可参见**准备本地应用开发环境**。
2. 使用该用户进行Kerberos认证。
3. 使用HDFS上传命令，将“dataLoad”目录上传到HDFS某个指定目录（**developuser**用户需要对该目录有读写权限）。

   说明

   该指定目录需要与之前“job.properties”中定义的“oozie.coord.application.path”属性和“workflowAppUri”属性的值保持一致。

步骤4 执行流程文件。

1. 登录客户端节点，使用**developuser**用户进行Kerberos认证。

   cd /opt/client

   source bigdata_env

   kinit developuser

2. 启动流程。

   命令：

   oozie job -oozie *https://oozie server hostname:port/oozie* **-config** *job.properties文件所在路径* **-run**

   参数列表：

   表 25-6 参数列表

参数	含义
job	表示执行的是job任务
-oozie	Oozie服务器地址（任意节点）
-config	“job.properties”文件所在路径

参数	含义
-run	表示启动流程

例如：

oozie job -oozie https://10-1-130-10:21003/oozie -config job.properties -run

----结束

25.3.2 Oozie 代码样例说明

25.3.2.1 配置 Oozie 作业运行参数

功能描述

流程的属性定义文件，定义了流程运行期间使用的外部参数值对。

参数解释

“job.properties”文件中包含的各参数及其含义，请参见表25-7。

表 25-7 参数含义

参数	含义
nameNode	HDFS NameNode集群地址
resourceManager	Yarn ResourceManager地址
queueName	流程任务处理时使用的MapReduce队列名
dataLoadRoot	流程任务所在目录名
oozie.coord.application.path	Coordinator流程任务在HDFS上的存放路径
start	定时流程任务启动时间
end	定时流程任务终止时间
workflowAppUri	Workflow流程任务在HDFS上的存放路径

说明

可以根据业务需要，以“*key*= *values*”的格式自定义参数及值。

样例代码

```
nameNode=hdfs://hacluster

resourceManager=10.1.130.10:26004
queueName=QueueA
dataLoadRoot=examples

oozie.coord.application.path=${nameNode}/user/oozie_cli/${dataLoadRoot}/apps/dataLoad
start=2013-04-02T00:00Z
end=2014-04-02T00:00Z
workflowAppUri=${nameNode}/user/oozie_cli/${dataLoadRoot}/apps/dataLoad
```

25.3.2.2 配置 Oozie 业务运行流程

功能描述

描述了一个完整业务的流程定义文件。一般由一个start节点、一个end节点和多个实现具体业务的action节点组成。

参数解释

“workflow.xml”文件中包含的各参数及其含义，请参见表25-8。

表 25-8 参数含义

参数	含义
name	流程文件名
start	流程开始节点
end	流程结束节点
action	实现具体业务动作的节点（可以是多个）

样例代码

```
<workflow-app xmlns="uri:oozie:workflow:1.0" name="data_load">
   <start to="copyData"/>
   <action name="copyData">
   </action>
     ......
   <end name="end"/>
</workflow-app>
```

25.3.2.3 配置 Oozie 作业执行入口

功能描述

流程任务的执行入口，每个流程任务有且仅有一个该节点。

参数解释

Start Action节点中包含的各参数及其含义，请参见表25-9。

表 25-9 参数含义

参数	含义
to	后继action节点的名称

样例代码

```
<start to="mr-dataLoad"/>
```

25.3.2.4 配置 Oozie MapReduce 作业

功能描述

MapReduce任务节点，负责执行一个map-reduce任务。

参数解释

MapReduce Action节点中包含的各参数及其含义，请参见**表25-10**。

表 25-10 参数含义

参数	含义
name	map-reduce action的名称
resourceManager	MapReduce ResourceManager地址
name-node	HDFS NameNode地址
queueName	任务处理时使用的MapReduce队列名
mapred.mapper.class	Mapper类名
mapred.reducer.class	Reducer类名
mapred.input.dir	MapReduce处理数据的输入目录
mapred.output.dir	MapReduce处理后结果数据输出目录
mapred.map.tasks	MapReduce map任务个数

说明

“${变量名}”表示：该值来自job.properties所定义。
例如：${nameNode}表示的就是“hdfs://hacluster”。（可参见**配置Oozie作业运行参数**）

样例代码

```
<action name="mr-dataLoad">
    <map-reduce>
        <resource-manager>${resourceManager}</resource-manager>
        <name-node>${nameNode}</name-node>
        <prepare>
```

```
            <delete path="${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce"/>
        </prepare>
        <configuration>
            <property>
                <name>mapred.job.queue.name</name>
                <value>${queueName}</value>
            </property>
            <property>
                <name>mapred.mapper.class</name>
                <value>org.apache.oozie.example.SampleMapper</value>
            </property>
            <property>
                <name>mapred.reducer.class</name>
                <value>org.apache.oozie.example.SampleReducer</value>
            </property>
            <property>
                <name>mapred.map.tasks</name>
                <value>1</value>
            </property>
            <property>
                <name>mapred.input.dir</name>
                <value>/user/oozie/${dataLoadRoot}/input-data/text</value>
              </property>
              <property>
                <name>mapred.output.dir</name>
                <value>/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce</value>
              </property>
          </configuration>
      </map-reduce>
      <ok to="copyData"/>
      <error to="fail"/>
  </action>
```

25.3.2.5 配置 Oozie 作业操作 HDFS 文件

功能描述

HDFS文件操作节点，支持对HDFS文件及目录的创建、删除、授权功能。

参数解释

FS Action节点中包含的各参数及其含义，请参见表25-11。

表 25-11 参数含义

参数	含义
name	FS活动的名称
delete	删除指定的文件和目录的标签
move	将文件从源目录移动到目标目录的标签
chmod	修改文件或目录权限的标签
path	当前文件路径
source	源文件路径
target	目标文件路径
permissions	权限字符串

说明

“${变量名}”表示：该值来自job.properties所定义。

例如：${nameNode}表示的就是“hdfs://hacluster”。（可参见**配置Oozie作业运行参数**）

样例代码

```
<action name="copyData">
   <fs>
      <delete path='${nameNode}/user/oozie_cli/${dataLoadRoot}/result'/>
      <move source='${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce' target='${nameNode}/user/oozie_cli/${dataLoadRoot}/result'/>
      <chmod path='${nameNode}/user/oozie_cli/${dataLoadRoot}/reuslt' permissions='-rwxrw-rw-' dir-files='true'></chmod>
    </fs>
    <ok to="end"/>
    <error to="fail"/>
</action>
```

25.3.2.6 配置 Oozie 作业执行终点

功能描述

流程任务执行的终点，每个流程任务有且仅有一个该节点。

参数解释

End Action节点中包含的各参数及其含义，请参见**表25-12**。

表 25-12 参数含义

参数	含义
name	end活动的名称

样例代码

```
  <end name="end"/>
```

25.3.2.7 配置 Oozie 作业异常结束打印信息

功能描述

流程任务运行期间发生异常后，流程的异常结束节点。

参数解释

Kill Action节点中包含的各参数及其含义，请参见**表25-13**。

表 25-13 参数含义

参数	含义
name	kill活动的名称
message	根据业务需要，自定义的流程异常打印信息
${wf:errorMessage(wf:lastErrorNode())}	Oozie系统内置的异常信息函数

样例代码

```
<kill name="fail">
    <message>
        This workflow failed, error message[${wf:errorMessage(wf:lastErrorNode())}]
    </message>
</kill>
```

25.3.2.8 配置 Coordinator 定时调度作业

功能描述

周期性执行Workflow类型任务的流程定义文件。

参数解释

“coordinator.xml”中包含的各参数及其含义，请参见**表25-14**。

表 25-14 参数含义

参数	含义
frequency	流程定时执行的时间间隔
start	定时流程任务启动时间
end	定时流程任务终止时间
workflowAppUri	Workflow流程任务在HDFS上的存放路径
resourceManager	MapReduce ResourceManager地址
queueName	任务处理时使用的MapReduce队列名
nameNode	HDFS NameNode集群地址

说明

“${变量名}”表示：该值来自job.properties所定义。
例如：${nameNode}表示的就是“hdfs://hacluster”。（可参见**配置Oozie作业运行参数**）

样例代码

```
<coordinator-app name="cron-coord" frequency="${coord:days(1)}" start="${start}" end="${end}"
timezone="UTC" xmlns="uri:oozie:coordinator:0.2">
      <action>
      <workflow>
         <app-path>${workflowAppUri}</app-path>
         <configuration>
            <property>
               <name>resourceManager</name>
               <value>${resourceManager}</value>
            </property>
            <property>
               <name>nameNode</name>
               <value>${nameNode}</value>
            </property>
            <property>
               <name>queueName</name>
               <value>${queueName}</value>
            </property>
         </configuration>
      </workflow>
   </action>
</coordinator-app>
```

25.3.3 通过 Java API 提交 Oozie 作业

25.3.3.1 通过 Java API 提交 Oozie 作业开发思路

通过典型场景，用户可以快速学习和掌握Oozie的开发过程，并且对关键的接口函数有所了解。

本示例演示了如何通过Java API提交MapReduce作业和查询作业状态，代码示例只涉及了MapReduce作业，其他作业的API调用代码是一样的，仅job配置“job.properties”与工作流配置文件“workflow.xml”需根据实际情况设置。

说明

完成**导入并配置Oozie样例工程**操作后即可执行通过Java API提交MapReduce作业和查询作业状态。

25.3.3.2 通过 Java API 提交 Oozie 作业

功能简介

Oozie通过org.apache.oozie.client.OozieClient的run方法提交作业，通过getJobInfo获取作业信息。

代码样例

代码示例中请根据实际情况，修改“OOZIE_URL_DEFALUT”为实际的任意Oozie节点的主机名，例如“https://10-1-131-131:21003/oozie/”。

```
   public void test(String jobFilePath) {
      try {
         UserGroupInformation.getLoginUser()
               .doAs(
                     new PrivilegedExceptionAction<Void>() {
                        /**
                         * run job
```

```
                         *
                         * @return null
                         * @throws Exception exception
                         */
                        public Void run() throws Exception {
                            runJob(jobFilePath);
                            return null;
                        }
                    });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void runJob(String jobFilePath) throws OozieClientException, InterruptedException {

        Properties conf = getJobProperties(jobFilePath);

        // submit and start the workflow job
        String jobId = wc.run(conf);

        logger.info("Workflow job submitted : {}" , jobId);

        // wait until the workflow job finishes printing the status every 10 seconds
        while (wc.getJobInfo(jobId).getStatus() == WorkflowJob.Status.RUNNING) {
            logger.info("Workflow job running ... {}" , jobId);
            Thread.sleep(10 * 1000);
        }

        // print the final status of the workflow job
        logger.info("Workflow job completed ... {}" , jobId);
        logger.info(String.valueOf(wc.getJobInfo(jobId)));
    }

    /**
     * Get job.properties File in filePath
     *
     * @param filePath file path
     * @return job.properties
     * @since 2020-09-30
     */
    public Properties getJobProperties(String filePath) {
        File configFile = new File(filePath);
        if (!configFile.exists()) {
            logger.info(filePath , "{} is not exist.");
        }

        InputStream inputStream = null;

        // create a workflow job configuration
        Properties properties = wc.createConfiguration();
        try {
            inputStream = new FileInputStream(filePath);
            properties.load(inputStream);

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        return properties;
    }
```


注意事项

通过Java API访问Oozie需要先参考环境准备章节进行安全认证，并将依赖的配置文件（配置文件Workflow.xml的开发参见**配置Oozie业务运行流程**）与jar包上传到HDFS，并确保进行过安全认证的用户有权限访问HDFS上对应的目录（目录的属主是该用户，或与该用户属于同一个用户组）。

25.3.4 使用 Oozie 调度 Spark2x 访问 HBase 以及 Hive

前提条件

已经配置完成**导入并配置Oozie样例工程**的前提条件。

开发环境配置

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\oozie-examples”目录下的样例工程文件夹“ooziesecurity-examples”中的OozieSparkHBaseExample和OozieSparkHiveExample样例工程。

步骤2 将**准备MRS应用开发用户**时得到的keytab文件“user.keytab”和“krb5.conf”用户认证凭据文件复制到OozieSparkHBaseExample和OozieSparkHiveExample样例工程的“\src\main\resources”路径。

步骤3 修改样例工程中的如下参数，请参考**表25-15**。

表 25-15 文件参数修改列表

文件名	参数名	值	取值样例
src\main\resources\application.properties	submit_user	提交任务的用户	developuser
	oozie_url_default	https://*Oozie业务IP*:21003/oozie/	https://10.10.10.176:21003/oozie/
src\main\resources\job.properties	userName	提交任务的用户	developuser
	examplesRoot	默认值或根据实际情况修改	myjobs
	oozie.wf.application.path	默认值或根据实际情况修改	${nameNode}/user/${userName}/${examplesRoot}/apps/spark2x **须知** 需要保证此路径和“src\main\resources\workflow.xml”文件中的<jar>标签和<spark-opts>标签路径一致

文件名	参数名	值	取值样例
src\main\resources\workflow.xml	<jar> </jar>	将“OoizeSparkHBase-1.0.jar”修改成实际打包的jar包名称	<jar>${nameNode}/user/${userName}/${examplesRoot}/apps/spark2x/lib/OoizeSparkHBase-1.0.jar</jar>

说明

进入项目根目录，比如“D:\sample_project\src\oozie-examples\ooziesecurity-examples\OozieSparkHBaseExample” 然后执行**mvn clean package -DskipTests**，打包成功之后样例工程在target目录里面。

步骤4 根据**步骤3**配置的路径，在HDFS客户端上新建如下文件夹：

hdfs dfs -mkdir -p /user/developuser/myjobs/apps/spark2x/lib

hdfs dfs -mkdir -p /user/developuser/myjobs/apps/spark2x/hbase

hdfs dfs -mkdir -p /user/developuser/myjobs/apps/spark2x/hive

步骤5 将表格中的文件上传到对应目录，请参考**表25-16**。

表 25-16 文件上传列表

初始文件路径	文件	上传目标目录
Spark客户端目录（如“/opt/client/Spark2x/spark/conf”）	hive-site.xml	HDFS的“/user/developuser/myjobs/apps/spark2x”目录
	hbase-site.xml	
准备MRS应用开发用户时得到的keytab文件	user.keytab	
	krb5.conf	
Spark客户端目录（如“/opt/client/Spark2x/spark/jars”）	jar包	Oozie的share HDFS的“/user/oozie/share/lib/spark2x/”目录
将待使用样例工程的项目打包成jar包,比如OoizeSparkHBase-1.0.jar	jar包	HDFS的“/user/developuser/myjobs/apps/spark2x/lib/”目录
OozieSparkHiveExample样例工程目录“src\main\resources”	workflow.xml	HDFS的“/user/developuser/myjobs/apps/spark2x/hive/”目录 **说明** <spark-opts> 中的spark-archive-2x.zip路径需要根据实际HDFS文件路径进行修改。

初始文件路径	文件	上传目标目录
OozieSparkHBaseExample样例工程目录“src\main\resources”	workflow.xml	HDFS的“/user/developuser/myjobs/apps/spark2x/hbase/”目录 **说明** <spark-opts> 中的spark-archive-2x.zip路径需要根据实际HDFS文件路径进行修改。

步骤6 修改上传后HDFS的“/user/developuser/myjobs/apps/spark2x”目录下的“hive-site.xml”中“hive.security.authenticator.manager”参数的值，从“org.apache.hadoop.hive.ql.security.SessionStateUserMSGroupAuthenticator”改为“org.apache.hadoop.hive.ql.security.SessionStateUserGroupAuthenticator”。

步骤7 如果开启了ZooKeeper SSL，需要在上传后HDFS的“/user/developuser/myjobs/apps/spark2x”目录下的“hbase-site.xml”文件中新增以下内容：

```
<property>
<name>HBASE_ZK_SSL_ENABLED</name>
<value>true</value>
</property>
```

步骤8 执行以下命令创建Hive表。

可以在Hue WebUI中Hive面板直接输入以下SQL创建表。

CREATE DATABASE test;

CREATE TABLE IF NOT EXISTS `test`.`usr` (user_id int comment 'userID',user_name string comment 'userName',age int comment 'age')PARTITIONED BY (country string)STORED AS PARQUET;

CREATE TABLE IF NOT EXISTS `test`.`usr2` (user_id int comment 'userID',user_name string comment 'userName',age int comment 'age')PARTITIONED BY (country string)STORED AS PARQUET;

INSERT INTO TABLE test.usr partition(country='CN') VALUES(1,'maxwell',45),(2,'minwell',30),(3,'mike',22);

INSERT INTO TABLE test.usr partition(country='USA') VALUES(4,'minbin',35);

步骤9 使用**hbase shell**，执行以下命令创建HBase表。

create 'SparkHBase',{NAME=>'cf1'}

put 'SparkHBase','01','cf1:name','Max'

put 'SparkHBase','01','cf1:age','23'

----结束

25.4 调测 Oozie 应用

25.4.1 在本地 Windows 环境中调测 Oozie 应用

操作场景

在使用Java接口完成开发程序代码后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

操作步骤

- 在Windows本地运行程序，需要配置https ssl证书。
  a. 登录集群任意节点，进入如下目录下载ca.crt文件。
     cd ${BIGDATA_HOME}/om-agent_8.1.0.1/nodeagent/security/cert/subcert/certFile/
  b. 将ca.crt文件下载到本地，以管理员的身份打开cmd。
     输入如下命令：
     keytool -import -v -trustcacerts -alias ca -file "*D:|xx|ca.crt***" -storepass changeit -keystore "***%JAVA_HOME%|jre|lib|security|cacerts***"**
     其中“D:\xx\ca.crt”是实际ca.crt文件存放路径；“%JAVA_HOME% ”为JDK安装路径。
  c. 在开发环境中（例如IDEA中），右击OozieRestApiMain.java，单击“Run 'OozieRestApiMain.main()'”运行对应的应用程序工程。
- 使用Oozie客户端执行以下命令运行样例程序：
  oozie job -oozie https://*Oozie业务IP***:21003/oozie -config job.properties -run**
  其中需要提前将待使用样例工程目录“src\main\resources”中的“job.properties”文件复制到Oozie客户端所在目录。

25.4.2 查看 Oozie 应用调测结果

操作场景

Oozie样例工程运行完成后可以通过控制台查看输出结果。

操作步骤

控制台显示运行结果会有如下成功信息：

```
log4j:WARN No appenders could be found for logger (com.huawei.hadoop.security.LoginUtil).
log4j:WARN Please initialize the log4j system properly.
```



```
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/D:/temp/newClientSec/oozie-example/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/D:/temp/newClientSec/oozie-example/lib/slf4j-simple-1.7.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
current user is developuser@<系统域名> (auth:KERBEROS)
login user is developuser@<系统域名> (auth:KERBEROS)
cluset status is true
Warning: Could not get charToByteConverterClass!
Workflow job submitted: 0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job running ...0000071-160729120057089-oozie-omm-W
Workflow job completed ...0000071-160729120057089-oozie-omm-W
Workflow id[0000071-160729120057089-oozie-omm-W] status[SUCCEEDED]
----------finish Oozie --------------------
```

同时在HDFS上生成目录“/user/developuser/examples/output-data/map-reduce”，包括如下两个文件：

- _SUCCESS
- part-00000

可以通过Hue的文件浏览器或者通过HDFS如下命令行查看：

hdfs dfs -ls /user/developuser/examples/output-data/map-reduce

说明

在Windows下面执行的时候可能会出现下面的异常，但是不影响业务：

```
java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.
```

25.5 Oozie 应用开发常见问题

25.5.1 常用 Oozie API 接口介绍

25.5.1.1 Oozie Shell 接口介绍

表 25-17 接口参数说明

命令	参数	含义
oozie version	无	显示oozie版本信息
oozie job	-config <arg>	指定job配置文件（job.properties）路径
	-oozie <arg>	指定oozie server地址

命令	参数	含义
	-run	运行job
	-start <arg>	启动指定的job
	-submit	提交job
	-kill <arg>	删除指定的job
	-suspend <arg>	暂停指定的job
	-resume <arg>	恢复指定的job
	-D <property=value>	给指定的属性赋值
oozie admin	-oozie <arg>	指定oozie server地址
	-status	显示oozie服务状态

Oozie其他的命令和参数可参见以下地址：**https://oozie.apache.org/docs/5.1.0/DG_CommandLineTool.html**。

25.5.1.2 Oozie Java 接口介绍

Java API主要由org.apache.oozie.client.OozieClient提供。

表 25-18 接口介绍

方法	说明
public String run(Properties conf)	运行job
public void start(String jobId)	启动指定的job
public String submit(Properties conf)	提交job
public void kill(String jobId)	删除指定的job
public void suspend(String jobId)	暂停指定的job
public void resume(String jobId)	恢复指定的job
public WorkflowJob getJobInfo(String jobId)	获取job信息

25.5.1.3 Oozie REST 接口介绍

常用接口与JAVA一样，详情请参见**http://oozie.apache.org/docs/5.1.0/WebServicesAPI.html**。

26 Oozie 开发指南（普通模式）

26.1 Oozie 应用开发概述

26.1.1 Oozie 应用开发简介

Oozie 简介

Oozie是一个用来管理Hadoop任务的工作流引擎，Oozie流程基于有向无环图（Directed Acyclical Graph）来定义和描述，支持多种工作流模式及流程定时触发机制。易扩展、易维护、可靠性高，与Hadoop生态系统各组件紧密结合。

Oozie流程的三种类型：

- Workflow

  描述一个完整业务的基本流程。
- Coordinator

  Coordinator流程构建在Workflow流程之上，实现了对Workflow流程的定时触发、按条件触发功能。
- Bundle

  Bundle流程构建在Coordinator流程之上，提供对多个Coordinator流程的统一调度、控制和管理功能。

Oozie主要特点：

- 支持分发、聚合、选择等工作流程模式。
- 与Hadoop生态系统各组件紧密结合。
- 流程变量支持参数化。
- 支持流程定时触发。
- 自带一个Web Console，提供了流程查看、流程监控、日志查看等功能。

26.1.2 Oozie 应用开发常用概念

- **流程定义文件**

描述业务逻辑的XML文件，包括“workflow.xml”、“coordinator.xml”、“bundle.xml”三类，最终由Oozie引擎解析并执行。

- **流程属性文件**

  流程运行期间的参数配置文件，对应文件名为“job.properties”，每个流程定义有且仅有一个该属性文件。

- **Client**

  客户端直接面向用户，可通过Java API、Shell API、 REST API或者Web UI访问Oozie服务端。

- **Oozie WebUI界面**

  通过“https://*Oozie服务器IP地址*:21003/oozie”登录Oozie WebUI界面。

26.1.3 Oozie 应用开发流程

本文档主要基于java API对Oozie进行应用开发。

开发流程中各阶段的说明如图26-1和表26-1所示。

图 26-1 Oozie 应用程序开发流程

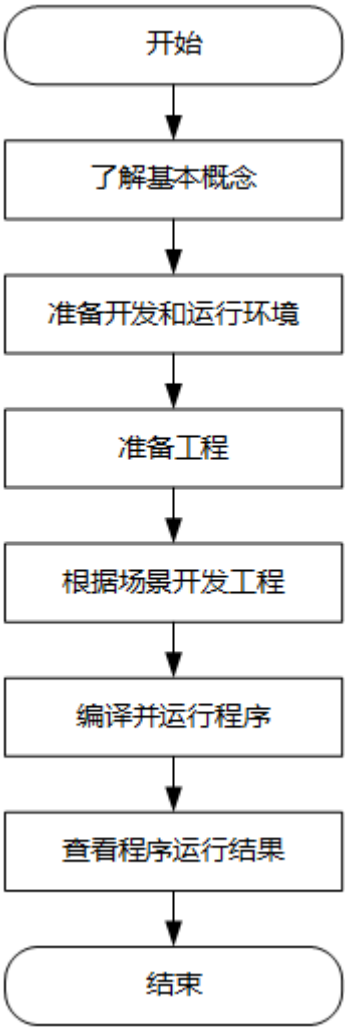


表 26-1 Oozie 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Oozie的基本概念，了解场景需求等。	**Oozie应用开发常用概念**
准备开发和运行环境	Oozie的应用程序当前推荐使用Java语言进行开发。可使用IDEA工具。	**准备本地应用开发环境**
准备工程	Oozie提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。	**导入并配置Oozie样例工程**
根据场景开发工程	提供了Java语言的样例工程。	**开发Oozie应用**
编译并运行程序	指导用户将开发好的程序编译并提交运行。	**调测Oozie应用**
查看程序运行结果	程序运行结果会写在用户指定的路径下。用户还可以通过UI查看应用运行情况。	**调测Oozie应用**

26.1.4 Oozie 应用开发样例工程介绍

MRS样例工程获取地址为**https://github.com/huaweicloud/huaweicloud-mrs-example**，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Oozie相关样例工程：

表 26-2 Oozie 相关样例工程

样例工程位置	描述
oozie-examples/oozienormal-examples/OozieMapReduceExample	Oozie提交MapReduce任务示例程序。 本示例演示了如何通过Java API提交MapReduce作业和查询作业状态，对网站的日志文件进行离线分析。
oozie-examples/oozienormal-examples/OozieSparkHBaseExample	使用Oozie调度Spark访问HBase的示例程序。
oozie-examples/oozienormal-examples/OozieSparkHiveExample	使用Oozie调度Spark访问Hive的示例程序。

26.2 准备 Oozie 应用开发环境

26.2.1 准备本地应用开发环境

在进行二次开发时，要准备的开发和运行环境如表26-3所示。

表 26-3 开发环境

准备项	说明
操作系统	Windows系统，支持Windows 7以上版本。 开发和运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 ● X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 ● TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。
安装和配置IDEA	用于开发Oozie应用程序的工具。版本要求：支持JDK1.8以上的版本。 **说明** ● 若使用IBM JDK，请确保IDEA中的JDK配置为IBM JDK。 ● 若使用Oracle JDK，请确保IDEA中的JDK配置为Oracle JDK。 ● 若使用Open JDK，请确保IDEA中的JDK配置为Open JDK。 ● 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
7-zip	用于解压“*.zip”和“*.rar”文件。 支持7-zip 16.04版本。

26.2.2 导入并配置 Oozie 样例工程

操作场景

将下载的样例工程导入到Windows开发环境IDEA中即可开始样例学习。

前提条件

- 已在Linux环境中安装了完整客户端。
- 获取Oozie服务器URL（任意节点），这个URL将是客户端提交流程任务的目标地址。

  URL格式为：https://*oozie实例业务IP*:21003/oozie。可登录FusionInsight Manager，选择“集群 > 服务 > Oozie > 实例”，即可获取任一oozie实例的IP地址；单击“配置”，在搜索框中搜索“OOZIE_HTTPS_PORT”，即可查看使用的端口号。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\oozie-examples”目录下的样例工程文件夹“oozienormal-examples”中的OozieMapReduceExample，OozieSparkHBaseExample和OozieSparkHiveExample三个样例工程。

步骤2 在应用开发环境中，导入样例工程到IDEA开发环境。

1. 在IDEA中选择“File > Open”，弹出“浏览文件夹”对话框。
2. 选择样例工程文件夹，单击“OK”。

步骤3 修改样例工程中的如下参数，请参考**表26-4**。

表 26-4 文件参数修改列表

文件名	参数名	值	取值样例
\src\main \resources \job.properties	userName	提交作业的用户	developuser
\src\main \resources \application.properties	submit_user	提交作业的用户	developuser
	oozie_url_default	https://*oozie实例的业务IP:21003*/oozie	https://10.10.10.176:21003/oozie

步骤4 选择运行的样例工程：

- OozieMapReduceExcample样例工程，执行**步骤5**。
- OozieSparkHBaseExample和OozieSparkHiveExample样例工程，请参考**使用Oozie调度Spark2x访问HBase以及Hive**。

步骤5 使用客户端上传Oozie的example文件到HDFS。

1. 登录客户端所在节点，切换到客户端所在目录，例如“/opt/client”。

   cd /opt/client
2. 执行以下命令配置环境变量。

   source bigdata_env
3. 执行以下命令在HDFS创建目录并上传样例工程到该目录。

   hdfs dfs -mkdir /user/developuser

hdfs dfs -put -f /opt/client/Oozie/oozie-client-*/examples /user/developuser

说明

命令行中 “oozie-client-*”涉及的版本号以实际版本号为准。

----结束

26.3 开发 Oozie 应用

26.3.1 开发 Oozie 配置文件

26.3.1.1 Oozie 样例程序开发思路

开发流程

1. 工作流配置文件“workflow.xml”（“coordinator.xml”是对工作流进行调度，“bundle.xml”是对一组Coordinator进行管理）与“job.properties”。
2. 如果有实现代码，需要开发对应的jar包，例如Java Action；如果是Hive，则需要开发SQL文件。
3. 上传配置文件与jar包（包括依赖的jar包）到HDFS，上传的路径取决于“workflow.xml”中的“oozie.wf.application.path”参数配置的路径。
4. 提供三种方式对工作流进行操作，详情请参见**Oozie应用开发常见问题**。
   - Shell命令
   - Java API
   - Hue
5. Oozie客户端提供了比较完整的examples示例供用户参考，包括各种类型的Action，以及Coordinator以及Bundle的使用。以客户端安装目录为“/opt/client”为例，examples具体目录为“/opt/client/Oozie/oozie-client-*/examples”。

如下通过一个MapReduce工作流的示例演示如何配置文件，并通过Shell命令调用。

场景说明

假设存在这样的业务需求：

每天需要对网站的日志文件进行离线分析，统计出网站各模块的访问频率（日志文件存放在HDFS中）。

通过客户端中模板与配置文件提交任务。

26.3.1.2 Oozie 应用开发步骤

步骤1 业务分析。

1. 可以使用客户端样例目录中MapReduce程序对日志目录的数据进行分析、处理。
2. 将MapReduce程序的分析结果移动到数据分析结果目录，并将数据文件的权限设置成**660**。

3. 为了满足每天分析一次的需求，需要每天重复执行一次**步骤1.1**～**步骤1.2**。

步骤2 业务实现。

1. 登录客户端所在节点，新建“dataLoad”目录，作为程序运行目录，后面编写的文件均保存在该目录下。例如“/opt/client/Oozie/oozie-client-*/examples/apps/dataLoad/”。

   说明

   可以直接复制样例目录中“map-reduce”文件夹内的内容到“dataLoad”文件夹，然后进行编辑。

   目录中 “oozie-client-*”涉及的版本号以实际版本号为准。

2. 编写流程任务属性文件（job.properties）。

   请参见**配置Oozie作业运行参数**。

3. 编写Workflow任务：“workflow.xml”。

   表 26-5 流程 Action

编号	步骤	描述
1	定义start action	请参见**配置Oozie作业执行入口**
2	定义MapReduce action	请参见**配置Oozie MapReduce作业**
3	定义FS action	请参见**配置Oozie作业操作HDFS文件**
4	定义end action	请参见**配置Oozie作业执行终点**
5	定义kill action	请参见**配置Oozie作业异常结束打印信息**

   说明

   依赖或新开发的jar包需要放在“dataLoad/lib”目录下。

   流程文件样例：

```
<workflow-app xmlns="uri:oozie:workflow:1.0" name="data_load">
  <start to="mr-dataLoad"/>
  <action name="mr-dataLoad">
     <map-reduce><resource-manager>${resourceManager}</resource-manager>
        <name-node>${nameNode}</name-node>
        <prepare>
           <delete path="${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce"/>
        </prepare>
        <configuration>
           <property>
              <name>mapred.job.queue.name</name>
              <value>${queueName}</value>
           </property>
           <property>
              <name>mapred.mapper.class</name>
              <value>org.apache.oozie.example.SampleMapper</value>
           </property>
           <property>
              <name>mapred.reducer.class</name>
              <value>org.apache.oozie.example.SampleReducer</value>
```



```
            </property>
            <property>
               <name>mapred.map.tasks</name>
               <value>1</value>
            </property>
            <property>
               <name>mapred.input.dir</name>
               <value>/user/oozie/${dataLoadRoot}/input-data/text</value>
            </property>
            <property>
               <name>mapred.output.dir</name>
               <value>/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce</value>
            </property>
         </configuration>
      </map-reduce>
      <ok to="copyData"/>
      <error to="fail"/>
   </action>

   <action name="copyData">
       <fs>
          <delete path='${nameNode}/user/oozie/${dataLoadRoot}/result'/>
          <move source='${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce'
              target='${nameNode}/user/oozie/${dataLoadRoot}/result'/>
          <chmod path='${nameNode}/user/oozie/${dataLoadRoot}/result' permissions='-rwxrw-rw-'
dir-files='true'></chmod>
       </fs>
      <ok to="end"/>
      <error to="fail"/>
   </action>

   <kill name="fail">
      <message>This workflow failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</
message>
   </kill>
   <end name="end"/>
</workflow-app>
```

4. 编写Coordinator任务：“coordinator.xml”。

   完成每天一次的定时数据分析工作，请参见**配置Coordinator定时调度作业**。

步骤3 上传流程文件。

1. 使用或切换到拥有HDFS上传权限的用户。
2. 使用HDFS上传命令，将“dataLoad”目录上传到HDFS某个指定目录（**oozie_cli**用户需要对该目录有读写权限）。

   说明

   该指定目录需要与之前“job.properties”中定义的“oozie.coord.application.path”属性和workflowAppUri属性的值保持一致。

步骤4 执行流程文件

命令：

oozie job -oozie *https://oozie server hostname:port/oozie* **-config** *job.properties文件所在路径* **-run**

参数列表：

表 26-6 参数列表

参数	含义
job	表示执行的是job任务
-oozie	Oozie服务器地址（任意节点）
-config	job.properties文件所在路径
-run	表示启动流程

例如：

oozie job -oozie https://10-1-130-10:21003/oozie -config job.properties -run

----结束

26.3.2 Oozie 样例代码说明

26.3.2.1 配置 Oozie 作业运行参数

功能描述

流程的属性定义文件，定义了流程运行期间使用的外部参数值对。

参数解释

“job.properties”文件中包含的各参数及其含义，请参见表26-7。

表 26-7 参数含义

参数	含义
nameNode	HDFS NameNode集群地址
resourceManager	Yarn ResourceManager地址
queueName	流程任务处理时使用的MapReduce队列名
dataLoadRoot	流程任务所在目录名
oozie.coord.application.path	Coordinator流程任务在HDFS上的存放路径
start	定时流程任务启动时间
end	定时流程任务终止时间
workflowAppUri	Workflow流程任务在HDFS上的存放路径

说明

可以根据业务需要，以“*key*= *values*”的格式自定义参数及值。

样例代码

```
nameNode=hdfs://hacluster

resourceManager=10.1.130.10:26004
queueName=QueueA
dataLoadRoot=examples

oozie.coord.application.path=${nameNode}/user/oozie_cli/${dataLoadRoot}/apps/dataLoad
start=2013-04-02T00:00Z
end=2014-04-02T00:00Z
workflowAppUri=${nameNode}/user/oozie_cli/${dataLoadRoot}/apps/dataLoad
```

26.3.2.2 配置 Oozie 业务运行流程

功能描述

描述了一个完整业务的流程定义文件。一般由一个start节点、一个end节点和多个实现具体业务的action节点组成。

参数解释

“workflow.xml”文件中包含的各参数及其含义，请参见表26-8。

表 26-8 参数含义

参数	含义
name	流程文件名
start	流程开始节点
end	流程结束节点
action	实现具体业务动作的节点（可以是多个）

样例代码

```
<workflow-app xmlns="uri:oozie:workflow:1.0" name="data_load">
   <start to="copyData"/>
   <action name="copyData">
   </action>
     ......
   <end name="end"/>
</workflow-app>
```

26.3.2.3 配置 Oozie 作业执行入口

功能描述

流程任务的执行入口，每个流程任务有且仅有一个该节点。

参数解释

Start Action节点中包含的各参数及其含义，请参见表26-9。

表 26-9 参数含义

参数	含义
to	后继action节点的名称

样例代码

```
<start to="mr-dataLoad"/>
```

26.3.2.4 配置 Oozie MapReduce 作业

功能描述

MapReduce任务节点，负责执行一个map-reduce任务。

参数解释

MapReduce Action节点中包含的各参数及其含义，请参见表26-10。

表 26-10 参数含义

参数	含义
name	map-reduce action的名称
resourceManager	MapReduce ResourceManager地址
name-node	HDFS NameNode地址
queueName	任务处理时使用的MapReduce队列名
mapred.mapper.class	Mapper类名
mapred.reducer.class	Reducer类名
mapred.input.dir	MapReduce处理数据的输入目录
mapred.output.dir	MapReduce处理后结果数据输出目录
mapred.map.tasks	MapReduce map任务个数

说明

“${变量名}”表示：该值来自“job.properties”所定义。
例如：${nameNode}表示的就是“hdfs://hacluster”。（可参见配置Oozie作业运行参数）

样例代码

```
<action name="mr-dataLoad">
    <map-reduce>
        <resource-manager>${resourceManager}</resource-manager>
        <name-node>${nameNode}</name-node>
        <prepare>
           <delete path="${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce"/>
        </prepare>
        <configuration>
           <property>
              <name>mapred.job.queue.name</name>
              <value>${queueName}</value>
           </property>
           <property>
              <name>mapred.mapper.class</name>
              <value>org.apache.oozie.example.SampleMapper</value>
           </property>
           <property>
              <name>mapred.reducer.class</name>
              <value>org.apache.oozie.example.SampleReducer</value>
           </property>
           <property>
              <name>mapred.map.tasks</name>
              <value>1</value>
           </property>
           <property>
              <name>mapred.input.dir</name>
              <value>/user/oozie/${dataLoadRoot}/input-data/text</value>
            </property>
            <property>
               <name>mapred.output.dir</name>
               <value>/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce</value>
            </property>
         </configuration>
      </map-reduce>
      <ok to="copyData"/>
      <error to="fail"/>
  </action>
```

26.3.2.5 配置 Oozie 作业操作 HDFS 文件

功能描述

HDFS文件操作节点，支持对HDFS文件及目录的创建、删除、授权功能。

参数解释

FS Action节点中包含的各参数及其含义，请参见表26-11。

表 26-11 参数含义

参数	含义
name	FS活动的名称
delete	删除指定的文件和目录的标签
move	将文件从源目录移动到目标目录的标签
chmod	修改文件或目录权限的标签
path	当前文件路径

参数	含义
source	源文件路径
target	目标文件路径
permissions	权限字符串

说明

“${变量名}”表示：该值来自“job.properties”所定义。

例如：${nameNode}表示的就是“hdfs://hacluster”。（可参见**配置Oozie作业运行参数**）

样例代码

```
<action name="copyData">
   <fs>
      <delete path='${nameNode}/user/oozie_cli/${dataLoadRoot}/result'/>
      <move source='${nameNode}/user/${wf:user()}/${dataLoadRoot}/output-data/map-reduce' target='${nameNode}/user/oozie_cli/${dataLoadRoot}/result'/>
      <chmod path='${nameNode}/user/oozie_cli/${dataLoadRoot}/reuslt' permissions='-rwxrw-rw-' dir-files='true'></chmod>
    </fs>
    <ok to="end"/>
    <error to="fail"/>
</action>
```

26.3.2.6 配置 Oozie 作业执行终点

功能描述

流程任务执行的终点，每个流程任务有且仅有一个该节点。

参数解释

End Action节点中包含的各参数及其含义，请参见**表26-12**。

表 26-12 参数含义

参数	含义
name	end活动的名称

样例代码

```
  <end name="end"/>
```

26.3.2.7 配置 Oozie 作业异常结束打印信息

功能描述

流程任务运行期间发生异常后，流程的异常结束节点。

参数解释

Kill Action节点中包含的各参数及其含义，请参见表26-13。

表 26-13 参数含义

参数	含义
name	kill活动的名称
message	根据业务需要，自定义的流程异常打印信息
${wf:errorMessage(wf:lastErrorNode())}	Oozie系统内置的异常信息函数

样例代码

```
<kill name="fail">
    <message>
        This workflow failed, error message[${wf:errorMessage(wf:lastErrorNode())}]
    </message>
 </kill>
```

26.3.2.8 配置 Coordinator 定时调度作业

功能描述

周期性执行Workflow类型任务的流程定义文件。

参数解释

“coordinator.xml”中包含的各参数及其含义，请参见表26-14。

表 26-14 参数含义

参数	含义
frequency	流程定时执行的时间间隔
start	定时流程任务启动时间
end	定时流程任务终止时间
workflowAppUri	Workflow流程任务在HDFS上的存放路径
resourceManager	MapReduce ResourceManager地址
queueName	任务处理时使用的MapReduce队列名
nameNode	HDFS NameNode集群地址

说明

“${变量名}”表示：该值来自“job.properties”所定义。

例如：${nameNode}表示的就是“hdfs://hacluster”。（可参见**配置Oozie作业运行参数**）

样例代码

```
<coordinator-app name="cron-coord" frequency="${coord:days(1)}" start="${start}" end="${end}"
timezone="UTC" xmlns="uri:oozie:coordinator:0.2">
      <action>
      <workflow>
         <app-path>${workflowAppUri}</app-path>
         <configuration>
            <property>
               <name>resourceManager</name>
               <value>${resourceManager}</value>
            </property>
            <property>
               <name>nameNode</name>
               <value>${nameNode}</value>
            </property>
            <property>
               <name>queueName</name>
               <value>${queueName}</value>
            </property>
         </configuration>
      </workflow>
   </action>
</coordinator-app>
```

26.3.3 通过 Java API 提交 Oozie 作业

26.3.3.1 通过 Java API 提交 Oozie 作业开发思路

通过典型场景，用户可以快速学习和掌握Oozie的开发过程，并且对关键的接口函数有所了解。

本示例演示了如何通过Java API提交MapReduce作业和查询作业状态，代码示例只涉及了MapReduce作业，其他作业的API调用代码是一样的，仅job配置“job.properties”与工作流配置“workflow.xml”需根据实际情况设置。

说明

完成**导入并配置Oozie样例工程**操作后即可执行通过Java API提交MapReduce作业和查询作业状态。

26.3.3.2 通过 Java API 提交 Oozie 作业

功能简介

Oozie通过org.apache.oozie.client.OozieClient的run方法提交作业，通过getJobInfo获取作业信息。

代码样例

代码示例中请根据实际情况，修改“OOZIE_URL_DEFALUT”为实际的任意Oozie的主机名，例如“https://10-1-131-131:21003/oozie/”。

```
public void test(String jobFilePath) {
    try {
        runJob(jobFilePath);
    } catch (Exception exception) {
        exception.printStackTrace();
    }
}

private void runJob(String jobFilePath) throws OozieClientException, InterruptedException {

    Properties conf = getJobProperties(jobFilePath);
    String user = PropertiesCache.getInstance().getProperty("submit_user");
    conf.setProperty("user.name", user);

    // submit and start the workflow job
    String jobId = oozieClient.run(conf);

    logger.info("Workflow job submitted: {}" , jobId);

    // wait until the workflow job finishes printing the status every 10 seconds
    while (oozieClient.getJobInfo(jobId).getStatus() == WorkflowJob.Status.RUNNING) {
        logger.info("Workflow job running ... {}" , jobId);
        Thread.sleep(10 * 1000);
    }

    // print the final status of the workflow job
    logger.info("Workflow job completed ... {}" , jobId);
    logger.info(String.valueOf(oozieClient.getJobInfo(jobId)));
}

/**
 * Get job.properties File in filePath
 *
 * @param filePath file path
 * @return job.properties
 * @since 2020-09-30
 */
public Properties getJobProperties(String filePath) {
    File configFile = new File(filePath);
    if (!configFile.exists()) {
        logger.info(filePath , "{} is not exist.");
    }

    InputStream inputStream = null;

    // create a workflow job configuration
    Properties properties = oozieClient.createConfiguration();
    try {
        inputStream = new FileInputStream(filePath);
        properties.load(inputStream);

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (inputStream != null) {
            try {
                inputStream.close();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }

    return properties;
}
```


26.3.4 使用 Oozie 调度 Spark2x 访问 HBase 以及 Hive

前提条件

已经配置完成**导入并配置Oozie样例工程**的前提条件。

开发环境配置

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\oozie-examples”目录下的样例工程文件夹“oozienormal-examples”文件夹中OozieMapReduceExample，OozieSparkHBaseExample和OozieSparkHiveExample样例工程。

步骤2 修改样例工程中的如下参数，请参考**表26-15**。

表 26-15 文件参数修改列表

文件名	参数名	值	取值样例
src\main\resources\application.properties	submit_user	提交任务的用户	developuser
	oozie_url_default	https://*Oozie业务IP*:21003/oozie/	https://10.10.10.233:21003/oozie/
src\main\resources\job.properties	userName	提交任务的用户	developuser
	examplesRoot	默认值或根据实际情况修改	myjobs
	oozie.wf.application.path	默认值或根据实际情况修改	${nameNode}/user/${userName}/${examplesRoot}/apps/spark2x **须知** 需要保证此路径和“src\main\resources\workflow.xml”文件中的<jar>标签和<spark-opts>标签路径一致
src\main\resources\workflow.xml	<jar> </jar>	将“OoizeSparkHBase-1.0.jar”修改成实际打包的jar包名称	<jar>${nameNode}/user/${userName}/${examplesRoot}/apps/spark2x/lib/OoizeSparkHBase-1.0.jar</jar>

说明

进入项目根目录，比如“D:\sample_project\src\oozie-examples\oozienormal-examples\OozieSparkHBaseExample” 然后执行**mvn clean package -DskipTests**，打包成功之后样例工程在target目录里面。

步骤3 根据步骤2配置的路径，在HDFS客户端上新建如下文件夹：

/user/developuser/myjobs/apps/spark2x/lib

/user/developuser/myjobs/apps/spark2x/hbase

/user/developuser/myjobs/apps/spark2x/hive

步骤4 将表格中的文件上传到对应目录，请参考表26-16。

表 26-16 文件上传列表

初始文件路径	文件	上传目标目录
Spark客户端目录（如“/opt/client/Spark2x/spark/conf”）	hive-site.xml	HDFS的“/user/developuser/myjobs/apps/spark2x”目录
	hbase-site.xml	
Spark客户端目录（如“/opt/client/Spark2x/spark/jars”）	jar包	Oozie的share HDFS的“/user/oozie/share/lib/spark2x”目录 **说明** 请执行**su - oozie**切换到**oozie**用户，使用**oozie**用户上传文件。 上传结束后再重启Oozie服务。
将待使用样例工程的项目打包成jar包	jar包	HDFS的“/user/developuser/myjobs/apps/spark2x/lib/”目录
OozieSparkHiveExample样例工程目录“src\main\resources”	workflow.xml	HDFS的“/user/developuser/myjobs/apps/spark2x/hive”目录 **说明** <spark-opts> 中的spark-archive-2x.zip路径需要根据实际HDFS文件路径进行修改。
OozieSparkHBaseExample样例工程目录“src\main\resources”	workflow.xml	HDFS的“/user/developuser/myjobs/apps/spark2x/hbase”目录 **说明** <spark-opts> 中的spark-archive-2x.zip路径需要根据实际HDFS文件路径进行修改。

步骤5 修改上传后HDFS的“/user/developuser/myjobs/apps/spark2x”目录下的“hive-site.xml”中“hive.security.authenticator.manager”参数的值，从“org.apache.hadoop.hive.ql.security.SessionStateUserMSGroupAuthenticator”改为“org.apache.hadoop.hive.ql.security.SessionStateUserGroupAuthenticator”。

步骤6 执行以下命令创建Hive表。

可以在Hue WebUI中的Hive面板直接输入以下SQL创建表。

CREATE DATABASE test;

CREATE TABLE IF NOT EXISTS `test`.`usr` (user_id int comment 'userID',user_name string comment 'userName',age int comment 'age')PARTITIONED BY (country string)STORED AS PARQUET;

CREATE TABLE IF NOT EXISTS `test`.`usr2` (user_id int comment 'userID',user_name string comment 'userName',age int comment 'age')PARTITIONED BY (country string)STORED AS PARQUET;

INSERT INTO TABLE test.usr partition(country='CN') VALUES(1,'maxwell',45),(2,'minwell',30),(3,'mike',22);

INSERT INTO TABLE test.usr partition(country='USA') VALUES(4,'minbin',35);

步骤7 使用**hbase shell**，执行以下命令创建HBase表。

create 'SparkHBase',{NAME=>'cf1'}

put 'SparkHBase','01','cf1:name','Max'

put 'SparkHBase','01','cf1:age','23'

----结束

26.4 调测 Oozie 应用

26.4.1 在本地 Windows 环境中调测 Oozie 应用

操作场景

在使用Java接口完成开发程序代码后，您可以在Windows开发环境中运行应用。本地和集群业务平面网络互通时，您可以直接在本地进行调测。

操作步骤

- 在Windows本地运行程序，需要配置HTTPS SSL证书。
  a. 登录集群任意节点，进入如下目录下载ca.crt文件。
     cd ${BIGDATA_HOME}/om-agent_8.1.0.1/nodeagent/security/cert/subcert/certFile/

b. 将ca.crt文件下载到本地，以管理员的身份打开cmd。

输入如下命令：

keytool -import -v -trustcacerts -alias ca -file "*D:|xx|ca.crt***" -storepass changeit -keystore "***%JAVA_HOME%|jre|lib|security|cacerts***"**

其中“D:\xx\ca.crt”是实际ca.crt文件存放路径；“%JAVA_HOME% ”为JDK安装路径。

c. 在开发环境中（例如IDEA中），右击OozieRestApiMain.java，单击“Run 'OozieRestApiMain.main()'”运行对应的应用程序工程。

- 使用Oozie客户端执行以下命令运行样例程序：

  oozie job -oozie https://*Oozie业务IP*:21003/oozie -config job.properties -run

  其中需要提前将待使用样例工程目录“src\main\resources”中的“job.properties”文件复制到Oozie客户端所在目录。

26.4.2 查看 Oozie 应用调测结果

操作场景

Oozie样例工程运行完成后可以通过控制台查看输出结果。

操作步骤

控制台显示运行结果会有如下成功信息：

```
cluset status is false
Warning: Could not get charToByteConverterClass!
Workflow job submitted: 0000067-160729120057089-oozie-omm-W
Workflow job running ...0000067-160729120057089-oozie-omm-W
Workflow job running ...0000067-160729120057089-oozie-omm-W
Workflow job running ...0000067-160729120057089-oozie-omm-W
Workflow job running ...0000067-160729120057089-oozie-omm-W
Workflow job running ...0000067-160729120057089-oozie-omm-W
Workflow job running ...0000067-160729120057089-oozie-omm-W
Workflow job running ...0000067-160729120057089-oozie-omm-W
Workflow job completed ...0000067-160729120057089-oozie-omm-W
Workflow id[0000067-160729120057089-oozie-omm-W] status[SUCCEEDED]
----------finish Oozie --------------------
```

同时在HDFS上生成目录“/user/developuser/examples/output-data/map-reduce”，包括如下两个文件：

- _SUCCESS
- part-00000

可以通过Hue的文件浏览器或者通过HDFS如下命令行查看：

hdfs dfs -ls /user/developuser/examples/output-data/map-reduce

说明

在Windows下面执行的时候可能会出现下面的异常，但是不影响业务：

```
java.io.IOException: Could not locate executable null\bin\winutils.exe in the Hadoop binaries.
```

26.5 Oozie 应用开发常见问题

26.5.1 常用 Oozie API 接口介绍

26.5.1.1 Oozie Shell 接口介绍

表 26-17 接口参数说明

命令	参数	含义
oozie version	无	显示Oozie版本信息
oozie job	-config <arg>	指定job配置文件（job.properties）路径
	-oozie <arg>	指定oozie server地址
	-run	运行job
	-start <arg>	启动指定的job
	-submit	提交job
	-kill <arg>	删除指定的job
	-suspend <arg>	暂停指定的job
	-resume <arg>	恢复指定的job
	-D <property=value>	给指定的属性赋值
oozie admin	-oozie <arg>	指定oozie server地址
	-status	显示oozie服务状态

Oozie其他的命令和参数可参见以下地址：**https://oozie.apache.org/docs/5.1.0/DG_CommandLineTool.html**。

26.5.1.2 Oozie Java 接口介绍

Java API主要由org.apache.oozie.client.OozieClient提供。

表 26-18 接口介绍

方法	说明
public String run(Properties conf)	运行job
public void start(String jobId)	启动指定的job
public String submit(Properties conf)	提交job

方法	说明
public void kill(String jobId)	删除指定的job
public void suspend(String jobId)	暂停指定的job
public void resume(String jobId)	恢复指定的job
public WorkflowJob getJobInfo(String jobId)	获取job信息

26.5.1.3 Oozie Rest 接口介绍

常用接口与JAVA一样，详情请参见**http://oozie.apache.org/docs/5.1.0/WebServicesAPI.html**。

27 Spark2x 开发指南（安全模式）

27.1 Spark 应用开发简介

Spark 简介

Spark是分布式批处理框架，提供分析挖掘与迭代式内存计算能力，支持多种语言（Scala/Java/Python）的应用开发。 适用以下场景：

- 数据处理（Data Processing）：可以用来快速处理数据，兼具容错性和可扩展性。
- 迭代计算（Iterative Computation）：支持迭代计算，有效应对多步的数据处理逻辑。
- 数据挖掘（Data Mining）：在海量数据基础上进行复杂的挖掘分析，可支持各种数据挖掘和机器学习算法。
- 流式处理（Streaming Processing）：支持秒级延迟的流式处理，可支持多种外部数据源。
- 查询分析（Query Analysis）：支持标准SQL查询分析，同时提供DSL（DataFrame）， 并支持多种外部输入。

本文档重点介绍Spark、Spark SQL和Spark Streaming应用开发指导。

Spark 开发接口简介

Spark支持使用Scala、Java和Python语言进行程序开发，由于Spark本身是由Scala语言开发出来的，且Scala语言具有简洁易懂的特性，推荐用户使用Scala语言进行Spark应用程序开发。

按不同的语言分，Spark的API接口如**表27-1**所示。

表 27-1 Spark API 接口

功能	说明
Scala API	提供Scala语言的API，Spark Core、SparkSQL和Spark Streaming模块的常用接口请参见**Spark Scala API接口介绍**。由于Scala语言的简洁易懂，推荐用户使用Scala接口进行程序开发。
Java API	提供Java语言的API，Spark Core、SparkSQL和Spark Streaming模块的常用接口请参见**Spark Java API接口介绍**。
Python API	提供Python语言的API，Spark Core、SparkSQL和Spark Streaming模块的常用接口请参见**Spark Python API接口介绍**。

按不同的模块分，Spark Core和Spark Streaming使用上表中的API接口进行程序开发。而SparkSQL模块，支持CLI或者JDBCServer两种方式访问。其中JDBCServer的连接方式也有Beeline和JDBC客户端代码两种。详情请参见**Spark JDBCServer接口介绍**。

说明

spark-sql脚本、spark-shell脚本和spark-submit脚本（运行的应用中带SQL操作），不支持使用proxy user参数去提交任务。另外，由于本文档中涉及的样例程序已添加安全认证，建议不要使用proxy user参数去提交任务。

基本概念

- **RDD**

  即弹性分布数据集（Resilient Distributed Dataset），是Spark的核心概念。指的是一个只读的，可分区的分布式数据集，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用。

  RDD的生成：

  - 从HDFS输入创建，或从与Hadoop兼容的其他存储系统中输入创建。
  - 从父RDD转换得到新RDD。
  - 从数据集合转换而来，通过编码实现。

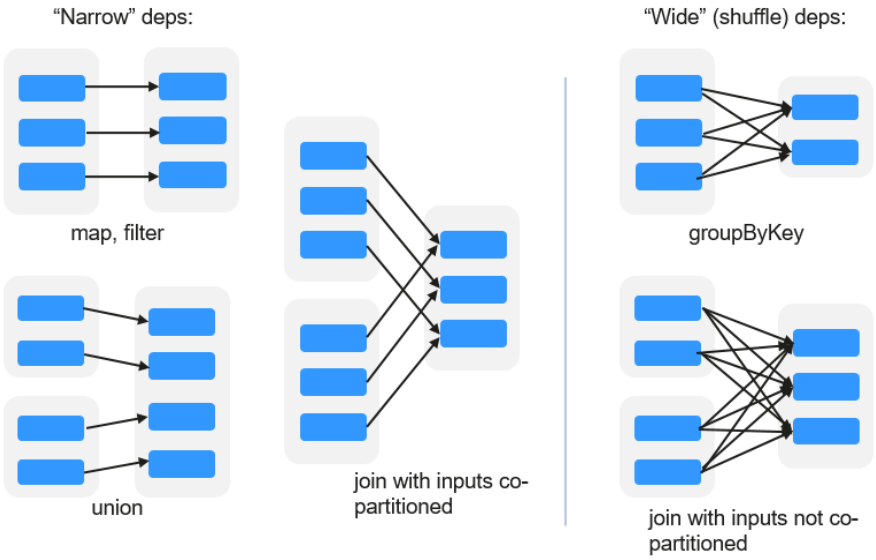
  RDD的存储：

  - 用户可以选择不同的存储级别缓存RDD以便重用（RDD有11种存储级别）。
  - 当前RDD默认是存储于内存，但当内存不足时，RDD会溢出到磁盘中。

- **Dependency（RDD的依赖）**

  RDD的依赖分别为：窄依赖和宽依赖。

图 27-1 RDD 的依赖



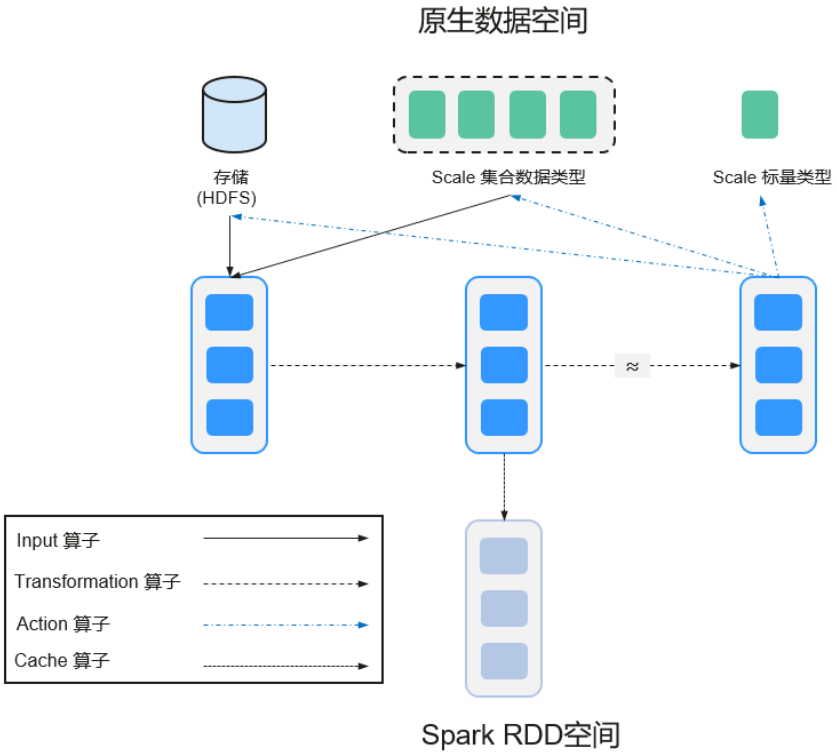
- **窄依赖：**指父RDD的每一个分区最多被一个子RDD的分区所用。
- **宽依赖：**指子RDD的分区依赖于父RDD的所有分区。

窄依赖对优化很有利。逻辑上，每个RDD的算子都是一个fork/join（此join非上文的join算子，而是指同步多个并行任务的barrier）：把计算fork到每个分区，算完后join，然后fork/join下一个RDD的算子。如果直接翻译到物理实现，是很不经济的：一是每一个RDD（即使是中间结果）都需要物化到内存或存储中，费时费空间；二是join作为全局的barrier，是很昂贵的，会被最慢的那个节点拖死。如果子RDD的分区到父RDD的分区是窄依赖，就可以实施经典的fusion优化，把两个fork/join合为一个；如果连续的变换算子序列都是窄依赖，就可以把很多个fork/join并为一个，不但减少了大量的全局barrier，而且无需物化很多中间结果RDD，这将极大地提升性能。Spark把这个叫做流水线（pipeline）优化。

- **Transformation和Action（RDD的操作）**

  对RDD的操作包含Transformation（返回值还是一个RDD）和Action（返回值不是一个RDD）两种。RDD的操作流程如图27-2所示。其中Transformation操作是Lazy的，也就是说从一个RDD转换生成另一个RDD的操作不是马上执行，Spark在遇到Transformations操作时只会记录需要这样的操作，并不会去执行，需要等到有Actions操作的时候才会真正启动计算过程进行计算。Actions操作会返回结果或把RDD数据写到存储系统中。Actions是触发Spark启动计算的动因。

图 27-2 RDD 操作示例



RDD看起来与Scala集合类型没有太大差别，但数据和运行模型大相迥异。

```
val file = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()
errors.count()
```

a. textFile算子从HDFS读取日志文件，返回file（作为RDD）。

b. filter算子筛出带“ERROR”的行，赋给errors（新RDD）。filter算子是一个Transformation操作。

c. cache算子缓存下来以备未来使用。

d. count算子返回errors的行数。count算子是一个Action操作。

Transformation操作可以分为如下几种类型：

- 视RDD的元素为简单元素。

  输入输出一对一，且结果RDD的分区结构不变，主要是map。

  输入输出一对多，且结果RDD的分区结构不变，如flatMap（map后由一个元素变为一个包含多个元素的序列，然后展平为一个个的元素）。

  输入输出一对一，但结果RDD的分区结构发生了变化，如union（两个RDD合为一个，分区数变为两个RDD分区数之和）、coalesce（分区减少）。

  从输入中选择部分元素的算子，如filter、distinct（去除重复元素）、subtract（本RDD有、其他RDD无的元素留下来）和sample（采样）。

- 视RDD的元素为Key-Value对。

  对单个RDD做一对一运算，如mapValues（保持源RDD的分区方式，这与map不同）；

  对单个RDD重排，如sort、partitionBy（实现一致性的分区划分，这个对数据本地性优化很重要）；

对单个RDD基于key进行重组和reduce，如groupByKey、reduceByKey；

对两个RDD基于key进行join和重组，如join、cogroup。

说明

后三种操作都涉及重排，称为shuffle类操作。

Action操作可以分为如下几种：

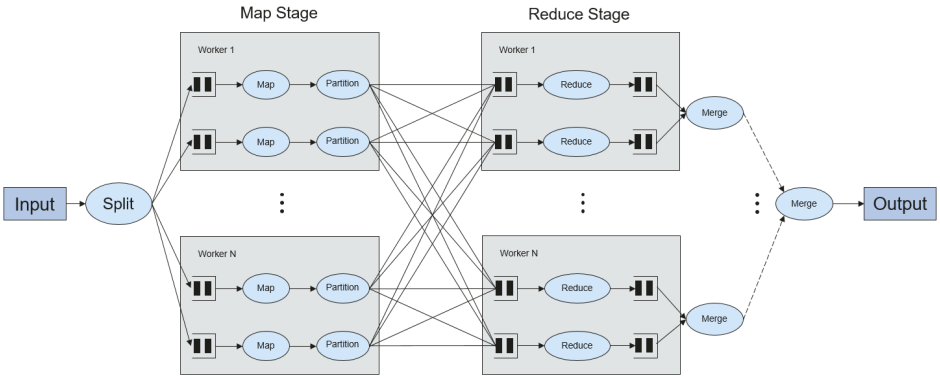
- 生成标量，如count（返回RDD中元素的个数）、reduce、fold/aggregate（返回几个标量）、take（返回前几个元素）。
- 生成Scala集合类型，如collect（把RDD中的所有元素倒入Scala集合类型）、lookup（查找对应key的所有值）。
- 写入存储，如与前文textFile对应的saveAsTextFile。
- 还有一个检查点算子checkpoint。当Lineage特别长时（这在图计算中时常发生），出错时重新执行整个序列要很长时间，可以主动调用checkpoint把当前数据写入稳定存储，作为检查点。

- **Shuffle**

  Shuffle是MapReduce框架中的一个特定的phase，介于Map phase和Reduce phase之间，当Map的输出结果要被Reduce使用时，每一条输出结果需要按key哈希，并且分发到对应的Reducer上去，这个过程就是shuffle。由于shuffle涉及到了磁盘的读写和网络的传输，因此shuffle性能的高低直接影响到了整个程序的运行效率。

  下图清晰地描述了MapReduce算法的整个流程。

  图 27-3 算法流程



  概念上shuffle就是一个沟通数据连接的桥梁，实际上shuffle这一部分是如何实现的呢，下面就以Spark为例讲解shuffle在Spark中的实现。

  Shuffle操作将一个Spark的Job分成多个Stage，前面的stages会包括一个或多个ShuffleMapTasks，最后一个stage会包括一个或多个ResultTask。

- **Spark Application的结构**

  Spark Application的结构可分为两部分：初始化SparkContext和主体程序。

  - 初始化SparkContext：构建Spark Application的运行环境。

    构建SparkContext对象，如：

    ```
    new SparkContext(master, appName, [SparkHome], [jars])
    ```

    参数介绍：

    master：连接字符串，连接方式有local、yarn-cluster、yarn-client等。

appName：构建的Application名称。

SparkHome：集群中安装Spark的目录。

jars：应用程序代码和依赖包。

- 主体程序：处理数据

提交Application的描述请参见：**https://spark.apache.org/docs/3.1.1/submitting-applications.html**

- **Spark shell命令**

  Spark基本shell命令，支持提交Spark应用。命令为：

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

  参数解释：

  --class：Spark应用的类名。

  --master：Spark用于所连接的master，如yarn-client，yarn-cluster等。

  application-jar：Spark应用的jar包的路径。

  application-arguments：提交Spark应用的所需要的参数（可以为空）。

- **Spark JobHistory Server**

  用于监控正在运行的或者历史的Spark作业在Spark框架各个阶段的细节以及提供日志显示，帮助用户更细粒度地去开发、配置和调优作业。

Spark SQL 常用概念

DataSet

DataSet是一个由特定域的对象组成的强类型集合，可通过功能或关系操作并行转换其中的对象。 每个Dataset还有一个非类型视图，即由多个列组成的DataSet，称为DataFrame。

DataFrame是一个由多个列组成的结构化的分布式数据集合，等同于关系数据库中的一张表，或者是R/Python中的data frame。DataFrame是Spark SQL中的最基本的概念，可以通过多种方式创建，例如结构化的数据集、Hive表、外部数据库或者是RDD。

Spark Streaming 常用概念

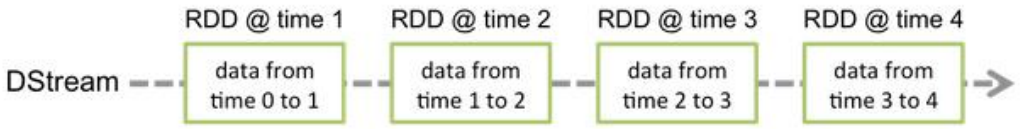
Dstream

DStream(又称Discretized Stream)是Spark Streaming提供的抽象概念。

DStream表示一个连续的数据流，是从数据源获取或者通过输入流转换生成的数据流。从本质上说，一个DStream表示一系列连续的RDD。RDD是一个只读的、可分区的分布式数据集。

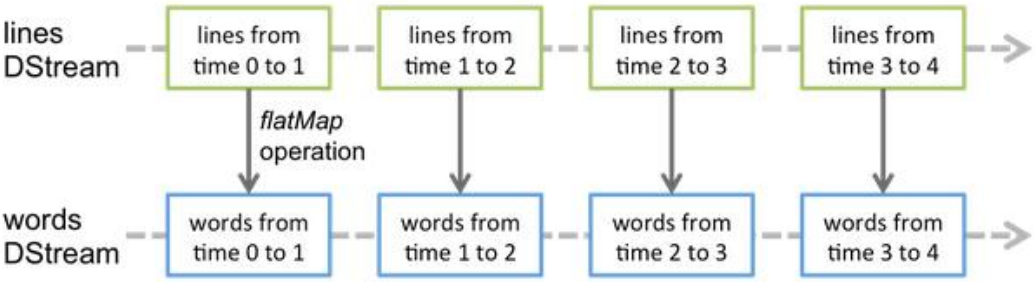
DStream中的每个RDD包含了一个区间的数据。如**图27-4**所示。

图 27-4 DStream 与 RDD 关系



应用到DStream上的所有算子会被转译成下层RDD的算子操作，如图27-5所示。这些下层的RDD转换会通过Spark引擎进行计算。DStream算子隐藏大部分的操作细节，并且提供了方便的High-level API给开发者使用。

图 27-5 DStream 算子转译



Structured Streaming 常用概念

- **Input Source**

  输入数据源，数据源需要支持根据offset重放数据，不同的数据源有不同的容错性。

- **Sink**

  数据输出，Sink要支持幂等性写入操作，不同的sink有不同的容错性。

- **outputMode**

  结果输出模式，当前支持3种输出模：

  - Complete Mode：整个更新的结果集都会写入外部存储。整张表的写入操作将由外部存储系统的连接器完成。
  - Append Mode：当时间间隔触发时，只有在Result Table中新增加的数据行会被写入外部存储。这种方式只适用于结果集中已经存在的内容不希望发生改变的情况下，如果已经存在的数据会被更新，不适合适用此种方式。
  - Update Mode：当时间间隔触发时，只有在Result Table中被更新的数据才会被写入外部存储系统。注意，和Complete Mode方式的不同之处是不更新的结果集不会写入外部存储。

- **Trigger**

  输出触发器，当前支持以下几种trigger：

  - 默认：以微批模式执行，每个批次完成后自动执行下个批次。
  - 固定间隔：固定时间间隔执行。
  - 一次执行：只执行一次query，完成后退出。
  - 连续模式：实验特性，可实现低至1ms延迟的流处理（推荐100ms）。

Structured Streaming支持微批模式和连续模式。微批模式不能保证对数据的低延迟处理，但是在相同时间下有更大的吞吐量；连续模式适合毫秒级的数据处理延迟，当前暂时还属于实验特性。

说明

在当前版本中，若需要使用流流Join功能，则output模式只能选择append模式。

图 27-6 微批模式运行过程简图

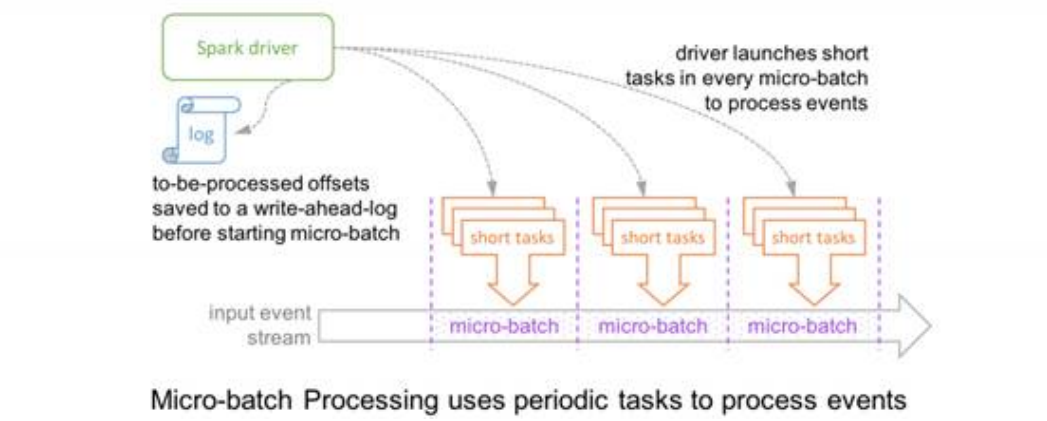
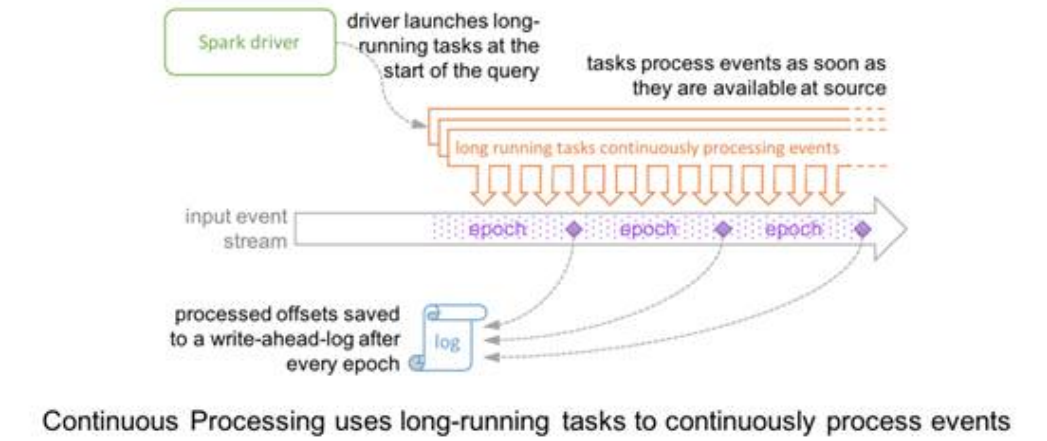


图 27-7 连续模式运行过程简图



27.2 Spark 应用开发流程介绍

Spark 应用程序开发流程

Spark包含Spark Core、Spark SQL和Spark Streaming三个组件，其应用开发流程都是相同的。

开发流程中各阶段的说明如图27-8和表27-2所示。

图 27-8 Spark 应用程序开发流程

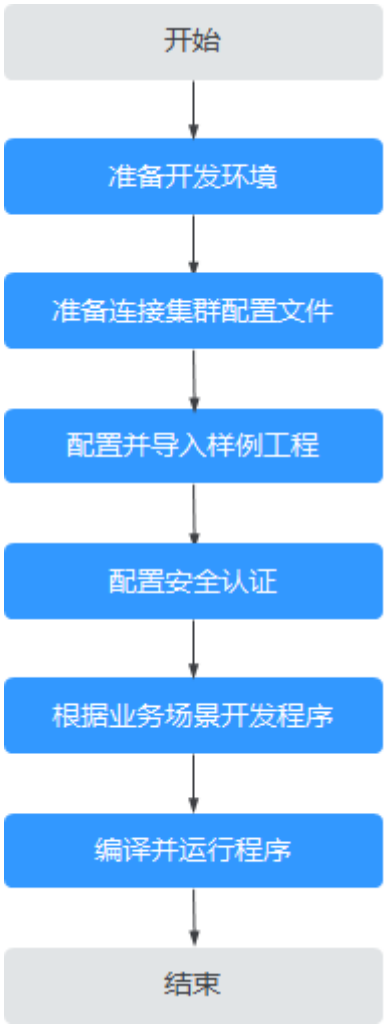


表 27-2 Spark 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	Spark的应用程序支持使用Scala、Java、Python三种语言进行开发。推荐使用IDEA工具，请根据指导完成不同语言的开发环境配置。Spark的运行环境即Spark客户端，请根据指导完成客户端的安装和配置。	**准备Spark本地应用开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。	**准备Spark连接集群配置文件**
配置并导入样例工程	Spark提供了不同场景下的多种样例程序，用户可以可获取样例工程并导入本地开发环境中进行程序学习，或者可以根据指导，新建一个Spark工程。	**导入并配置Spark样例工程** **新建Spark样例工程（可选）**

阶段	说明	参考文档
配置安全认证	如果您使用的是开启了kerberos认证的MRS集群，需要进行安全认证。	配置Spark应用安全认证
根据场景开发工程	提供了Scala、Java、Python三种不同语言的样例工程，还提供了Streaming、SQL、JDBC客户端程序以及Spark on HBase四种不同场景的样例工程。 帮助用户快速了解Spark各部件的编程接口。	开发Spark应用
编译并运行程序	将开发好的程序编译并运行，用户可在本地Windows开发环境中进行程序调测运行，也可以将程序编译为Jar包后，提交到Linux节点上运行。 **说明** 用户还可以根据程序运行情况，对程序进行调优，使其性能满足业务场景诉求。调优完成后，请重新进行编译和运行。具体请参考中Spark2x性能调优。	在Linux环境中编包并运行Spark程序

27.3 Spark2x 样例工程介绍

MRS样例工程获取地址为https://github.com/huaweicloud/huaweicloud-mrs-example，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Spark2x相关样例工程：

表 27-3 Spark2x 相关样例工程

样例工程位置	描述
sparksecurity-examples/SparkHbasetoCarbonJavaExample	Spark同步HBase数据到CarbonData的Java示例程序。 本示例工程中，应用将数据实时写入HBase，用于点查业务。数据每隔一段时间批量同步到CarbonData表中，用于分析型查询业务。
sparksecurity-examples/SparkHbasetoHbaseJavaExample	Spark从HBase读取数据再写入HBase的Java/Scala/Python示例程序。 本示例工程中，Spark应用程序实现两个HBase表数据的分析汇总。
sparksecurity-examples/SparkHbasetoHbasePythonExample	
sparksecurity-examples/SparkHbasetoHbaseScalaExample	
sparksecurity-examples/SparkHivetoHbaseJavaExample	Spark从Hive读取数据再写入到HBase的Java/Scala/Python示例程序。 本示例工程中，Spark应用程序实现分析处理Hive表中的数据，并将结果写入HBase表。
sparksecurity-examples/SparkHivetoHbasePythonExample	

样例工程位置	描述
sparksecurity-examples/SparkHivetoHbaseScalaExample	
sparksecurity-examples/SparkJavaExample	Spark Core任务的Java/Python/Scala/R示例程序。 本工程应用程序实现从HDFS上读取文本数据并计算分析。 SparkRExample示例不支持未开启Kerberos认证的集群。
sparksecurity-examples/SparkPythonExample	
sparksecurity-examples/SparkRExample	
sparksecurity-examples/SparkScalaExample	
sparksecurity-examples/SparkLauncherJavaExample	使用Spark Launcher提交作业的Java/Scala示例程序。 本工程应用程序通过org.apache.spark.launcher.SparkLauncher类采用Java/Scala命令方式提交Spark应用。
sparksecurity-examples/SparkLauncherScalaExample	
sparksecurity-examples/SparkOnClickHouseJavaExample	Spark通过ClickHouse JDBC的原生接口，以及Spark JDBC驱动，实现对ClickHouse数据库和表的创建、查询、插入等操作样例代码。
sparksecurity-examples/SparkOnClickHousePythonExample	
sparksecurity-examples/SparkOnClickHouseScalaExample	
sparksecurity-examples/SparkOnHbaseJavaExample	Spark on HBase场景的Java/Scala/Python示例程序。 本工程应用程序以数据源的方式去使用HBase，将数据以Avro格式存储在HBase中，并从中读取数据以及对读取的数据进行过滤等操作。
sparksecurity-examples/SparkOnHbasePythonExample	
sparksecurity-examples/SparkOnHbaseScalaExample	
sparksecurity-examples/SparkOnHudiJavaExample	Spark on Hudi场景的Java/Scala/Python示例程序。 本工程应用程序使用Spark操作Hudi执行插入数据、查询数据、更新数据、增量查询、特定时间点查询、删除数据等操作。
sparksecurity-examples/SparkOnHudiPythonExample	
sparksecurity-examples/SparkOnHudiScalaExample	
sparksecurity-examples/SparkOnMultiHbaseScalaExample	Spark同时访问两个集群中的HBase的Scala示例程序。

样例工程位置	描述
sparksecurity-examples/SparkSQLJavaExample	Spark SQL任务的Java/Python/Scala示例程序。 本工程应用程序实现从HDFS上读取文本数据并计算分析。
sparksecurity-examples/SparkSQLPythonExample	
sparksecurity-examples/SparkSQLScalaExample	
sparksecurity-examples/SparkStreamingKafka010JavaExample	Spark Streaming从Kafka接收数据并进行统计分析的Java/Scala示例程序。 本工程应用程序实时累加计算Kafka中的流数据，统计每个单词的记录总数。
sparksecurity-examples/SparkStreamingKafka010PythonExample	
sparksecurity-examples/SparkStreamingtoHbaseJavaExample010	Spark Streaming读取Kafka数据并写入HBase的Java/Scala/Python示例程序。 本工程应用程序每5秒启动一次任务，读取Kafka中的数据并更新到指定的HBase表中。
sparksecurity-examples/SparkStreamingtoHbasePythonExample010	
sparksecurity-examples/SparkStreamingtoHbaseScalaExample010	
sparksecurity-examples/SparkStructuredStreamingJavaExample	在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。
sparksecurity-examples/SparkStructuredStreamingPythonExample	
sparksecurity-examples/SparkStructuredStreamingScalaExample	
sparksecurity-examples/SparkThriftServerJavaExample	通过JDBC访问Spark SQL的Java/Scala示例程序。 本示例中，用户自定义JDBCServer的客户端，使用JDBC连接来进行表的创建、数据加载、查询和删除。
sparksecurity-examples/SparkThriftServerScalaExample	
sparksecurity-examples/StructuredStreamingADScalaExample	使用Structured Streaming，从kafka中读取广告请求数据、广告展示数据、广告点击数据，实时获取广告有效展示统计数据和广告有效点击统计数据，将统计结果写入kafka中。

样例工程位置	描述
sparksecurity-examples/StructuredStreamingStateScalaExample	在Spark结构流应用中，跨批次统计每个session期间发生了多少次event以及本session的开始和结束timestamp；同时输出本批次被更新状态的session。

27.4 准备 Spark 应用开发环境

27.4.1 准备 Spark 本地应用开发环境

Spark2x可以使用Java/Scala/Python语言进行应用开发，要准备的开发和运行环境如表27-4所示。

表 27-4 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	Java/Scala开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls。

准备项	说明
安装和配置IntelliJ IDEA	用于开发Spark应用程序的工具，建议使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
安装Scala	Scala开发环境的基本配置。版本要求：2.12.10。
安装Scala插件	Scala开发环境的基本配置。版本要求：2018.2.11或其他兼容版本。
安装Editra	Python开发环境的编辑器，用于编写Python程序。或者使用其他编写Python应用程序的IDE。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。
安装Python	版本要求不低于3.6。

27.4.2 准备 Spark 连接集群配置文件

准备集群认证用户信息

对于开启Kerberos认证的MRS集群，需提前准备具有相关组件操作权限的用户用于程序认证。

以下Spark2x权限配置示例供参考，在实际业务场景中可根据业务需求灵活调整。

步骤1 登录FusionInsight Manager。

步骤2 选择“集群 > 服务 > Spark2x > 更多 > 启用Ranger鉴权”，查看该参数是否置灰。

- 是，创建用户并在Ranger中赋予该用户相关操作权限：
  a. 选择“系统 > 权限 > 用户 > 添加用户”，在新增用户界面创建一个机机用户，例如**developuser**。
     “用户组”需加入**developgroup**组。若用户需要对接Kafka，则需要添加kafkaadmin用户组。
  b. 使用Ranger管理员用户**rangeradmin**登录Ranger管理页面。
  c. 在首页中单击“HADOOP SQL”区域的组件插件名称如“Hive”。
  d. 单击“Policy Name”名称为“all - database, table, column”操作列的 。
  e. 在“Allow Conditions”区域新增策略允许条件，“Select User”列勾选**步骤2.a**新建的用户名称，“Permissions”列勾选“Select All”。

f. 单击“Save”。

- 否，创建用户并在Manager赋予用户相关操作权限：

a. 选择“系统 > 权限 > 角色 > 添加角色”。

i. 填写角色的名称，例如**developrole**。

ii. （若安装了HBase，则配置）在“配置资源权限”表格中选择“*待操作集群的名称* > HBase > HBase Scope > global”，勾选“default”的“创建”，单击“确定”保存。

iii. （若安装了HBase，则配置）编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > HBase > HBase Scope > global > hbase”，勾选“hbase:meta”的“执行”，单击“确定”保存。

iv. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ > user”，勾选“hive”的“执行”，单击“确定”保存。

v. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > HDFS > 文件系统 > hdfs://hacluster/ >user >hive”，勾选“warehouse”的“读”、“写”和“执行”，单击“确定”保存。

vi. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > Hive > Hive读写权限”，勾选“default”的“建表”，单击“确定”保存。

vii. 编辑角色，在“配置资源权限”的表格中选择“*待操作集群的名称* > Yarn > 调度队列 > root”，勾选“default”的“提交”，单击“确定”保存。

b. 选择“用户 > 添加用户”，在新增用户界面，创建一个机机用户，例如**developuser**。

- “用户组”需加入“hadoop”用户组。
- “角色”加入**步骤2.a**新增的角色。

步骤3 使用**admin**用户登录FusionInsight Manager，选择“系统 > 权限 > 用户”，在用户名为**developuser**的操作列选择“更多 > 下载认证凭据”下载认证凭据文件，保存后解压得到该用户的“user.keytab”文件与“krb5.conf”文件。

----结束

准备运行环境配置文件

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。

a. 登录FusionInsight Manager页面，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。

b. 进入客户端配置文件解压路径“*\Spark\config”，获取Spark配置文件，并所有的配置文件导入到Spark样例工程的配置文件目录中（通常为“resources”文件夹）。

准备集群认证用户信息时获取的keytab文件也放置于该目录下。

c. 复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中。

说明

- 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
- 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 场景二：准备Linux环境运行程序所需配置文件。

a. 在节点中安装MRS集群客户端，例如客户端安装目录为“/opt/client”。

b. 获取配置文件：

i. 登录FusionInsight Manager，在“主页”右上方选择“更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。

ii. 以**root**登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“*\Spark\config”路径下的配置文件。并将所有的配置文件放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

cp -r Spark/config/* /opt/client/conf

准备集群认证用户信息时获取的keytab文件也放置于该目录下。

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

27.4.3 导入并配置 Spark 样例工程

操作场景

Spark针对多个场景提供样例工程，包含Java样例工程和Scala样例工程等，帮助客户快速学习Spark工程。

针对Java和Scala不同语言的工程，其导入方式相同。使用Python开发的样例工程不需要导入，直接打开Python文件（*.py）即可。

以下操作步骤以导入Java样例代码为例。操作流程如图27-9所示。

图 27-9 导入样例工程流程

前提条件

- 确保本地环境的时间与MRS集群的时间差要小于5分钟，若无法确定，请联系系统管理员。MRS集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备Spark连接集群配置文件**。

操作步骤

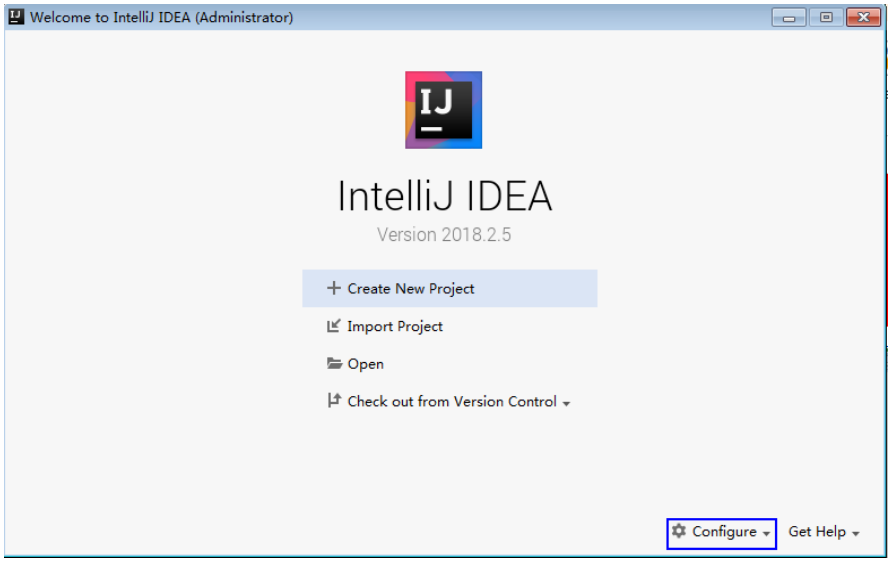
步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“spark-examples”目录下的“sparksecurity-examples”文件夹中的Scala、Spark Streaming等多个样例工程。

步骤2 若需要在本地Windows调测Spark样例代码，需参考**准备Spark连接集群配置文件**获取各样例项目所需的配置文件、认证文件，并手动将配置文件导入到Spark样例工程的配置文件目录中。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA中配置JDK。

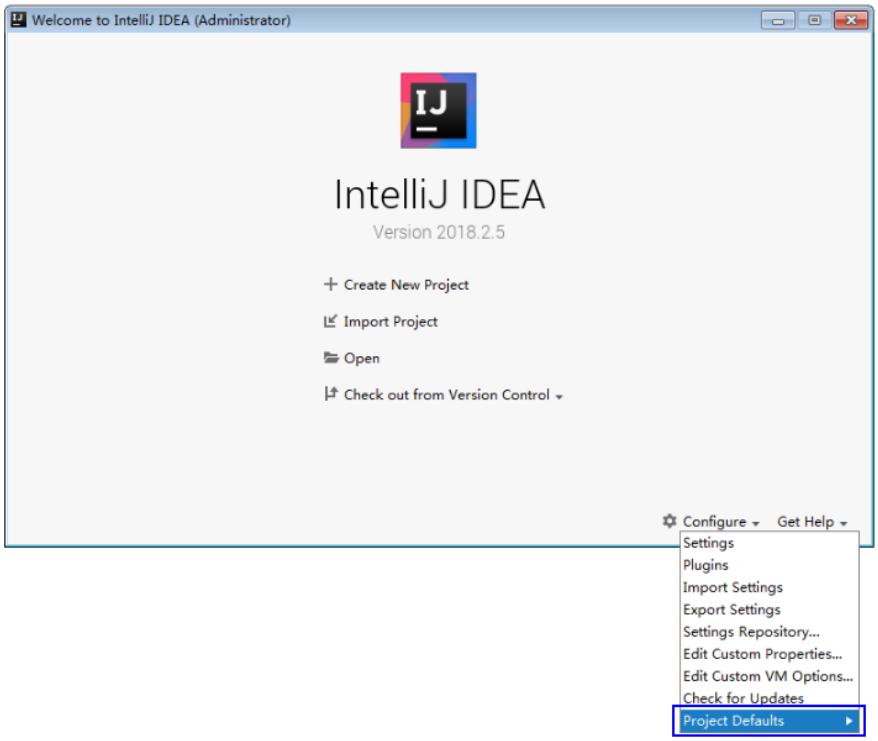
1. 打开IntelliJ IDEA，选择“Configure”。

图 27-10 Quick Start



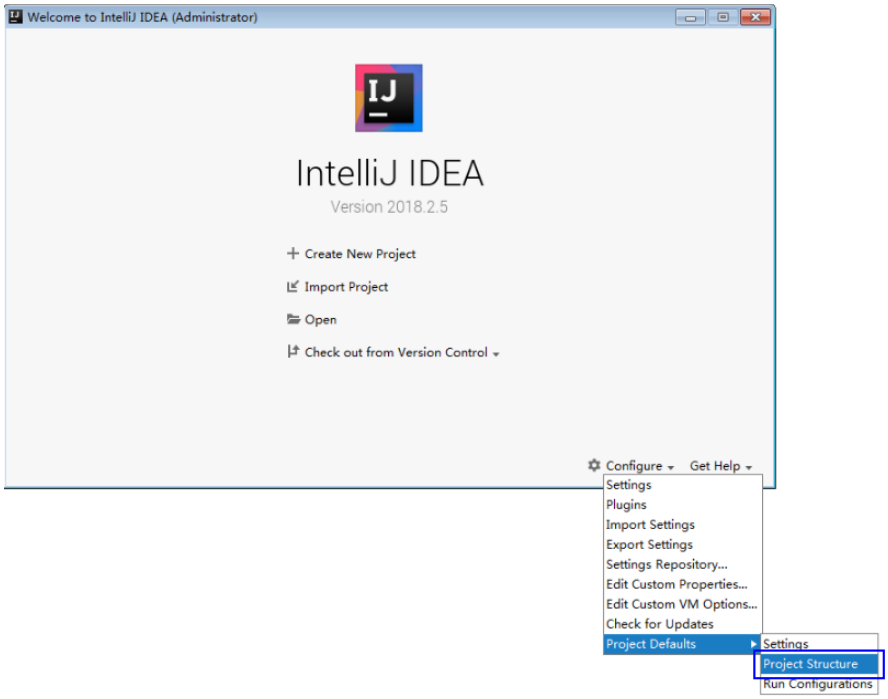
2. 在“Configure”下拉菜单中单击“Project Defaults”。

图 27-11 Configure



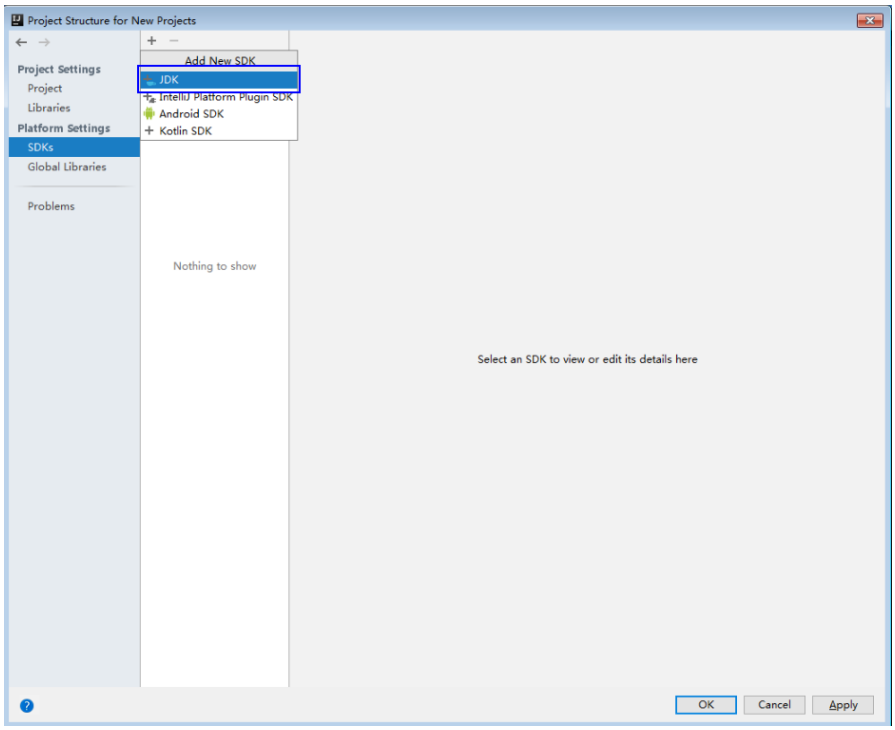
3. 在“Project Defaults”菜单中选择“Project Structure”。

图 27-12 Project Defaults



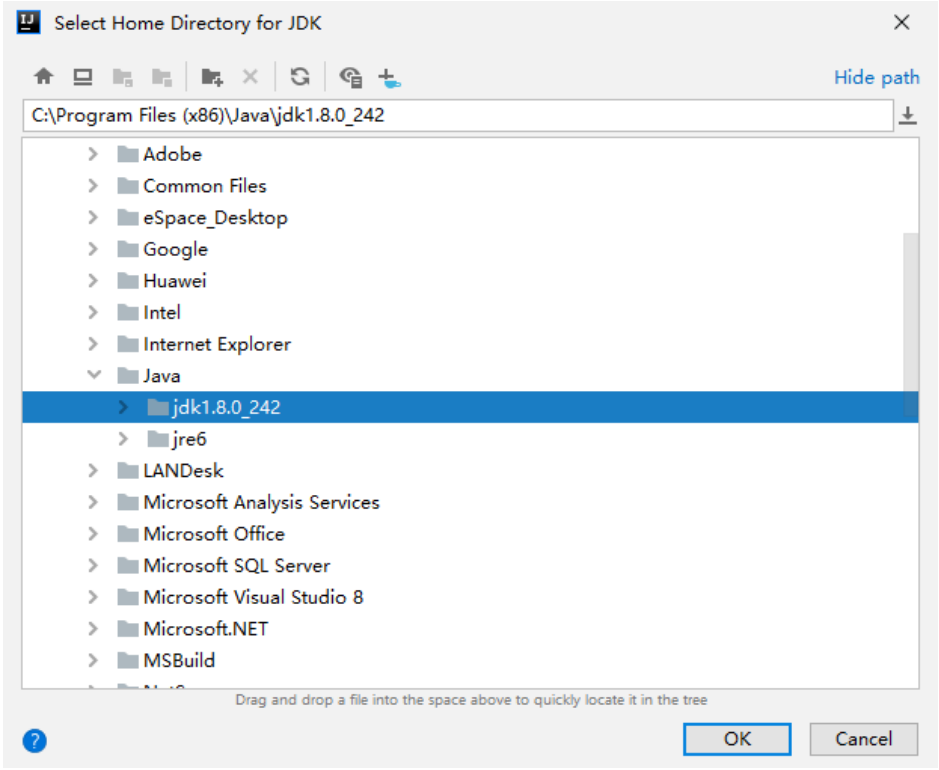
4. 在打开的“Project Structure”页面中，选择“SDKs”，单击加号添加JDK。

图 27-13 添加 JDK



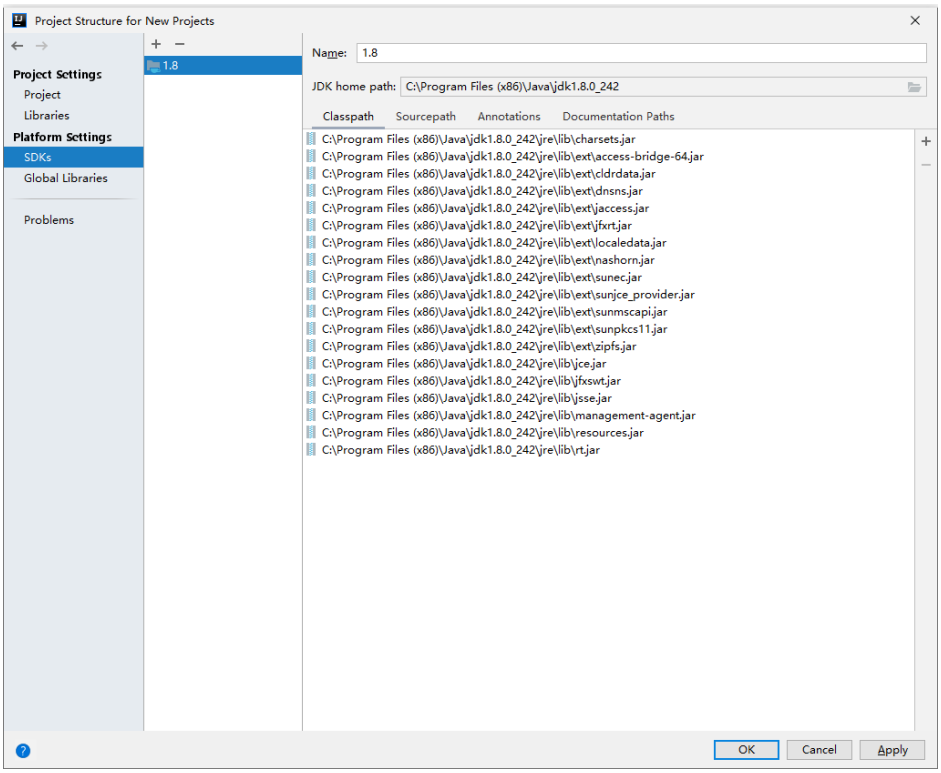
5. 在弹出的“Select Home Directoty for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 27-14 选择 JDK 目录



6. 完成JDK选择后，单击“OK”完成配置。

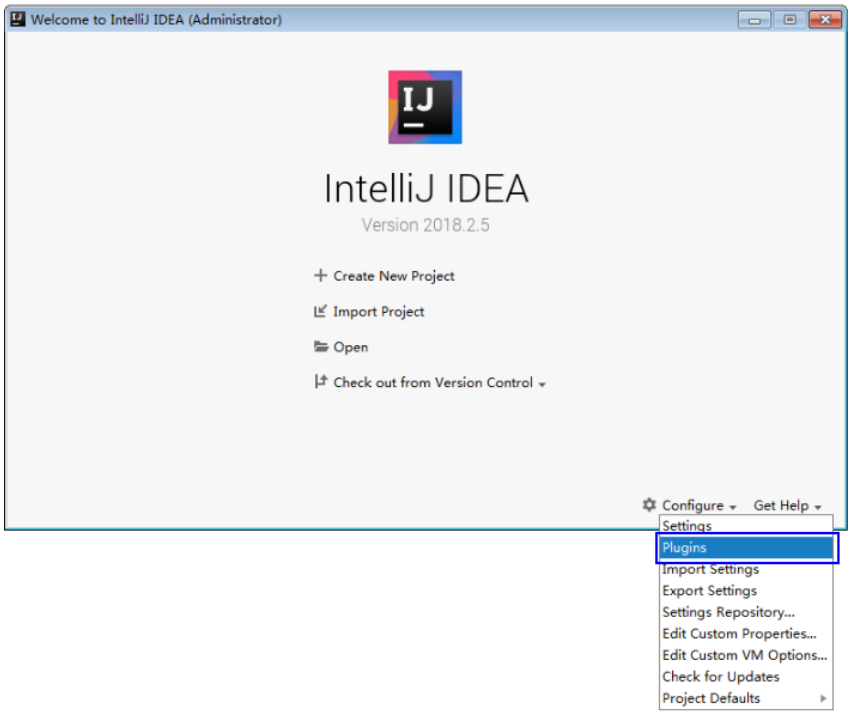
图 27-15 完成 JDK 配置



步骤4 （可选）如果是Scala开发环境，还需要在IntelliJ IDEA中安装Scala插件。

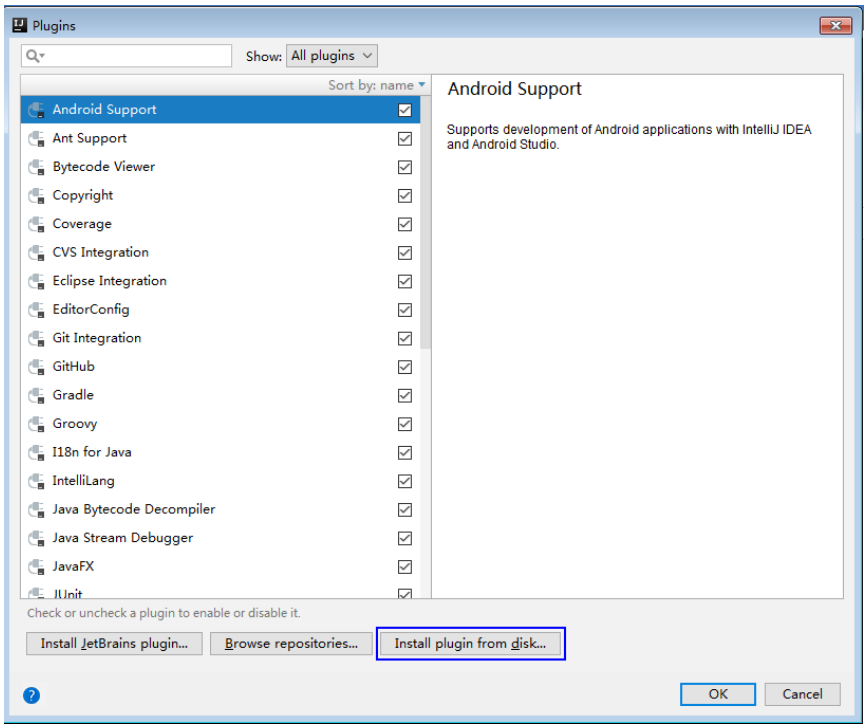
1. 在“Configure”下拉菜单中，单击“Plugins”。

图 27-16 Plugins

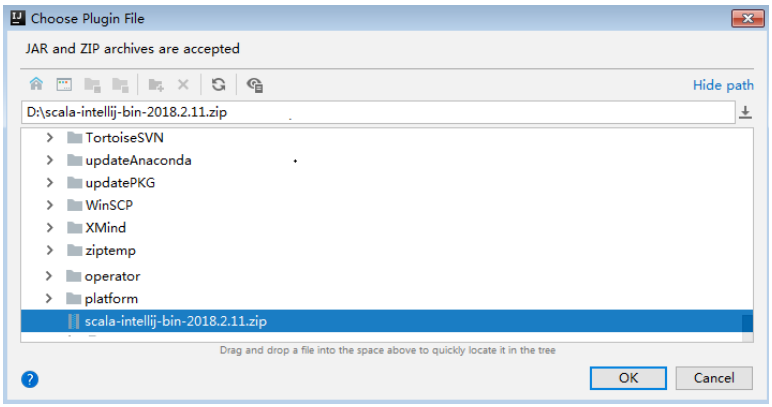


2. 在“Plugins”页面，选择“Install plugin from disk”。

图 27-17 Install plugin from disk

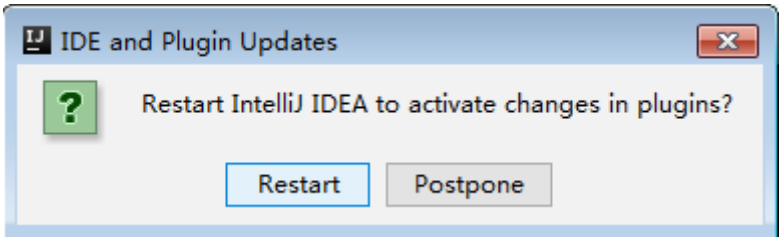


3. 在“Choose Plugin File”页面，选择对应版本的Scala插件包，单击“OK”。



4. 在“Plugins”页面，单击“Apply”安装Scala插件。
5. 在弹出的“Plugins Changed”页面，单击“Restart”，使配置生效。

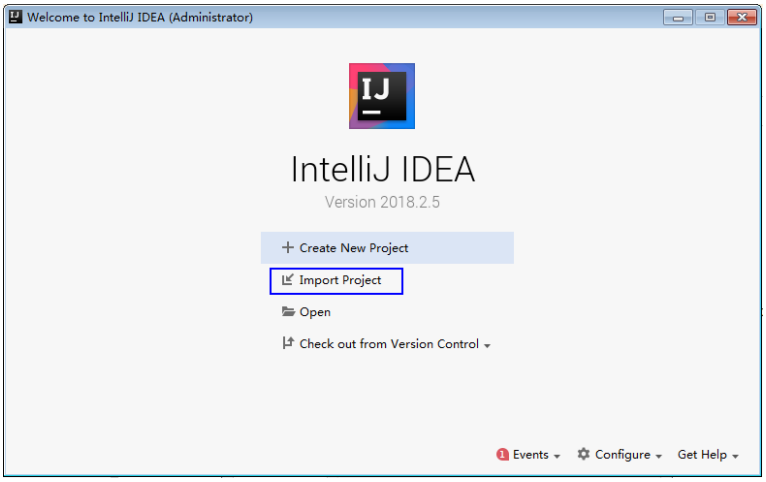
图 27-18 Plugins Changed



步骤5 将Java样例工程导入到IDEA中。

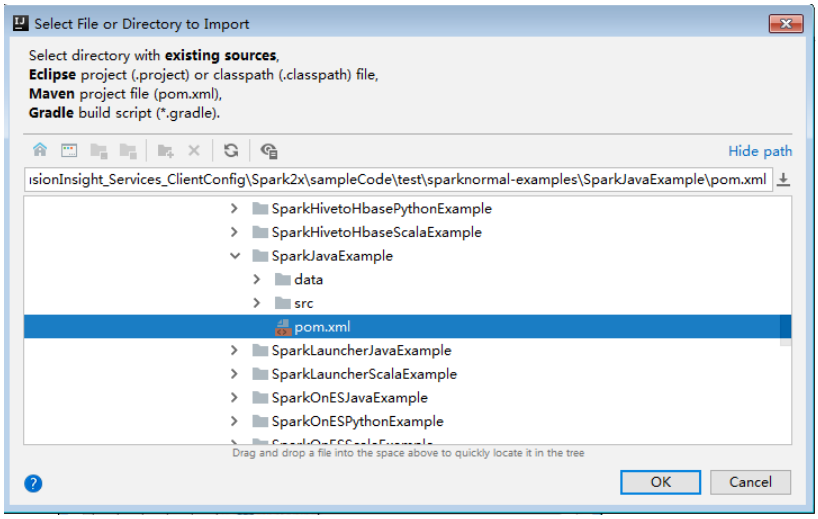
1. 打开IntelliJ IDEA。在“Quick Start”页面选择“Import Project”。
   或者，针对已使用过的IDEA工具，您可以从IDEA主界面直接添加。选择“File > Import project...”导入工程。

图 27-19 Import Project（Quick Start 页面）



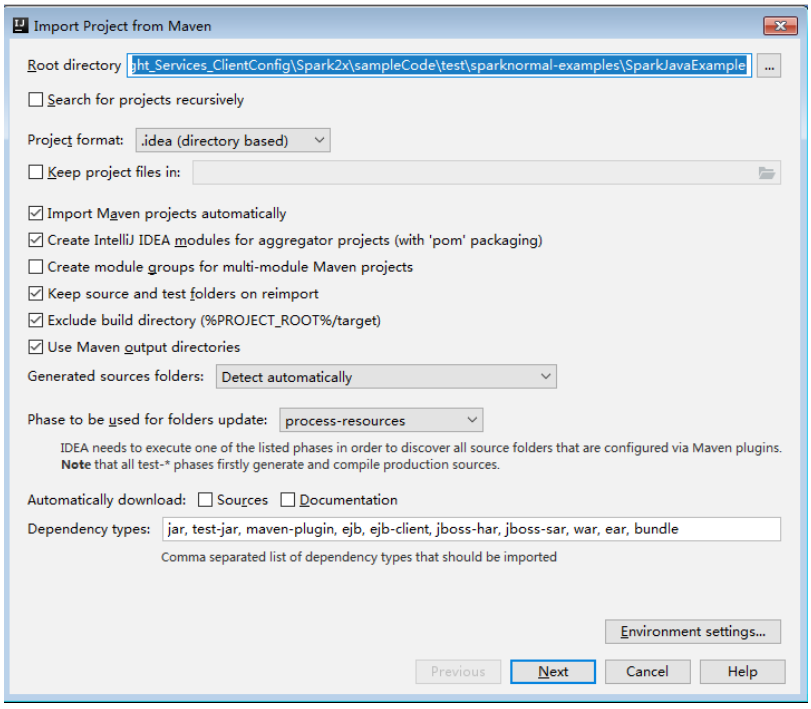
2. 选择需导入的样例工程存放路径及其pom文件，然后单击“OK”。

图 27-20 Select File or Directory to Import



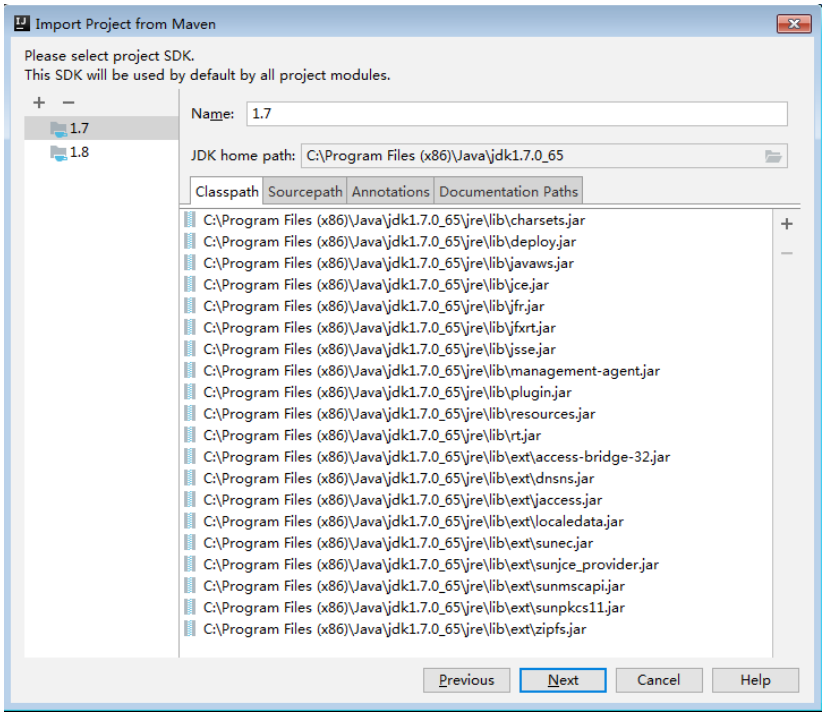
3. 确认导入路径和名称，单击“Next”。

图 27-21 Import Project from Maven



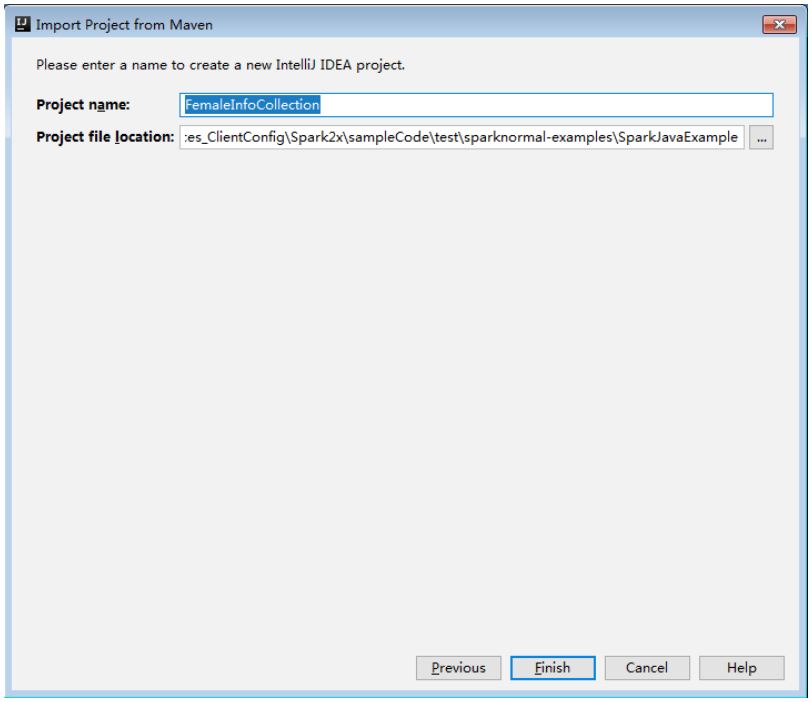
4. 选择需要导入的工程，然后单击“Next”。
5. 确认工程所用JDK，然后单击“Next”。

图 27-22 Select project SDK



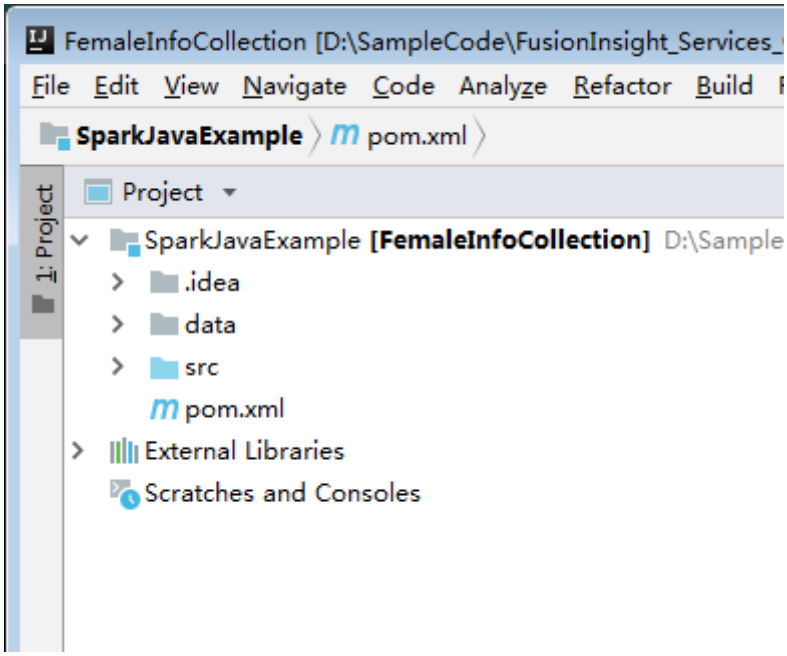
6. 确认工程名称和路径，单击“Finish”完成导入。

图 27-23 Confirm the project name and file location



7. 导入完成后，IDEA主页显示导入的样例工程。

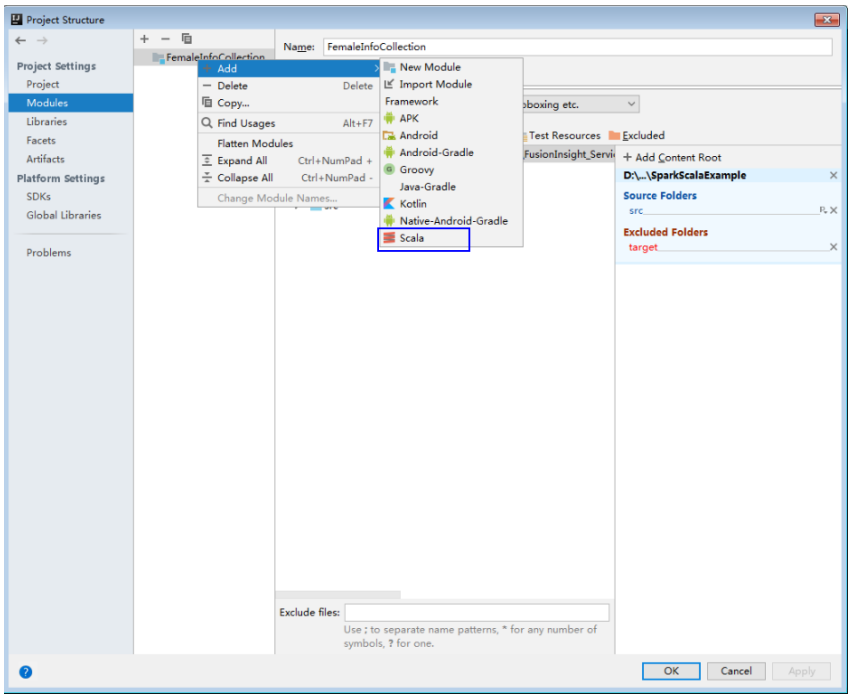
图 27-24 已导入工程



步骤6 （可选）如果导入Scala语言开发的样例程序，还需要为工程设置语言。

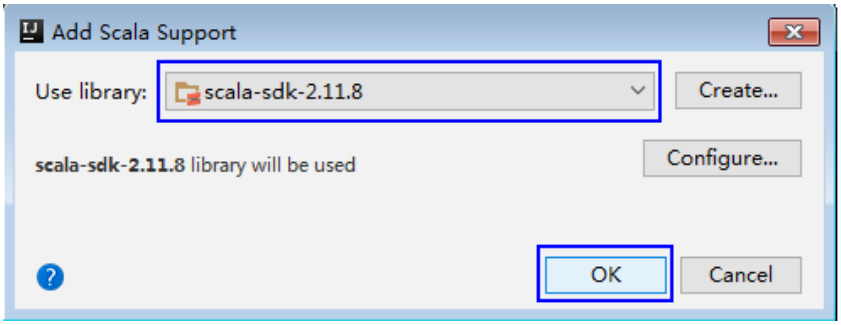
1. 在IDEA主页，选择“File>Project Structures...”进入“Project Structure”页面。
2. 选择“Modules”，选中工程名称，然后右键选择“Add > Scala”。

图 27-25 选择 Scala 语言



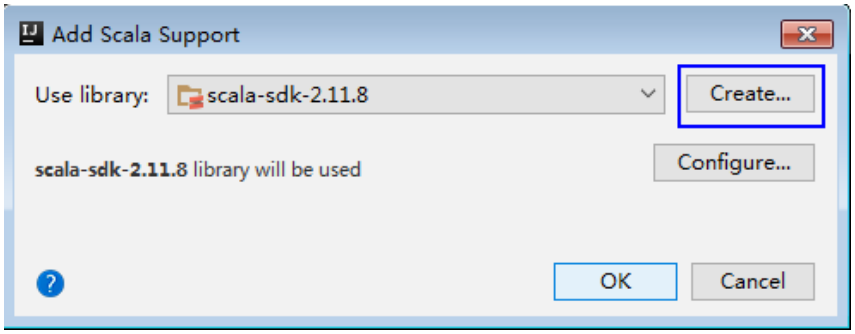
3. 当IDEA可以识别出Scala SDK时，在设置界面，选择编译的依赖jar包，然后单击“OK”应用设置。

图 27-26 Add Scala Support



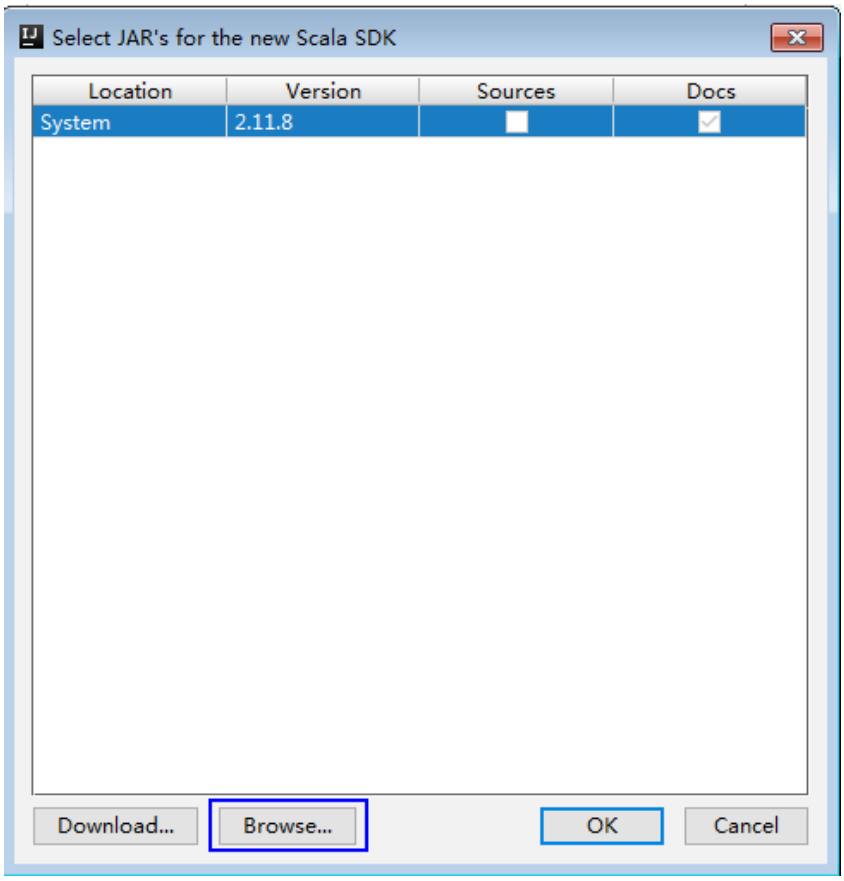
4. 当系统无法识别出Scala SDK时，需要自行创建。
   a. 单击“Create...”。

   图 27-27 Create...

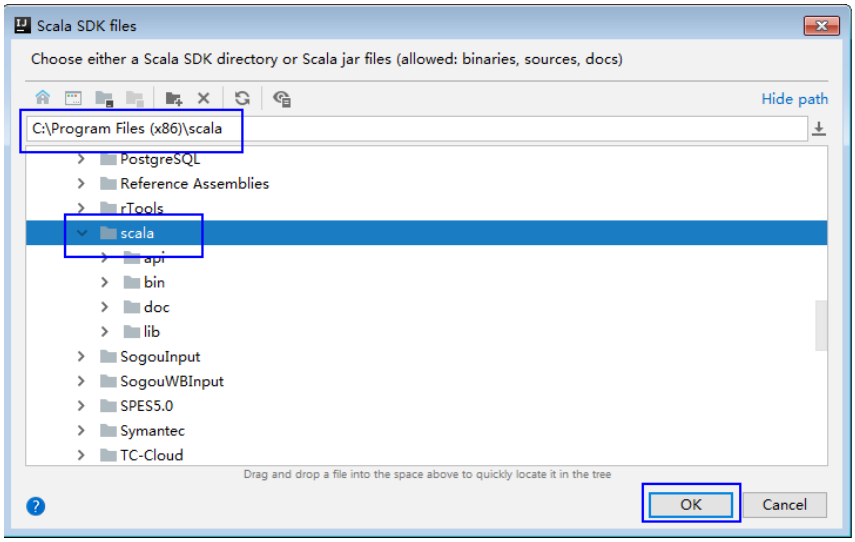
   b. 在“Select JAR's for the new Scala SDK”页面单击“Browse...”。

图 27-28 Select JAR's for the new Scala SDK



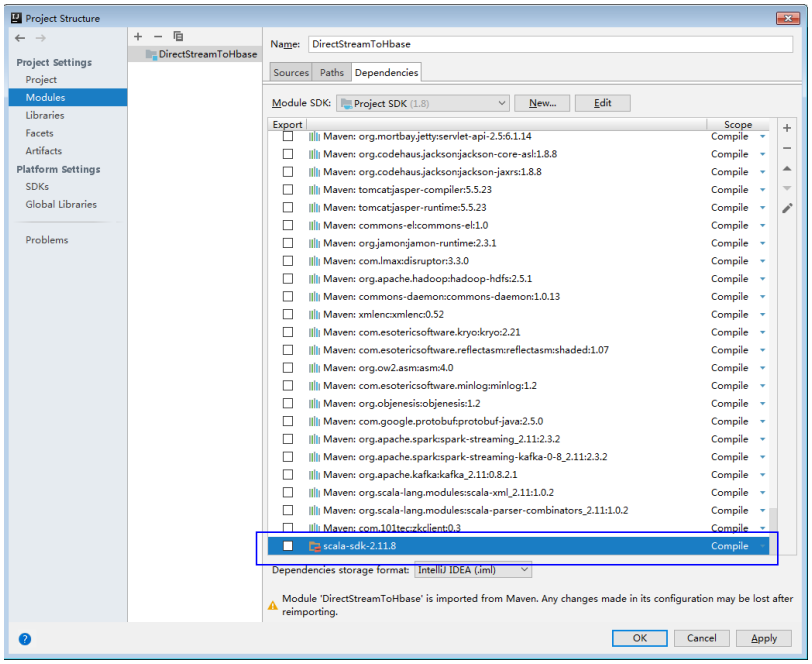
c. 在“Scala SDK files”页面选择scala sdk目录，单击“OK”。

图 27-29 Scala SDK files



5. 设置成功，单击“OK”保存设置。

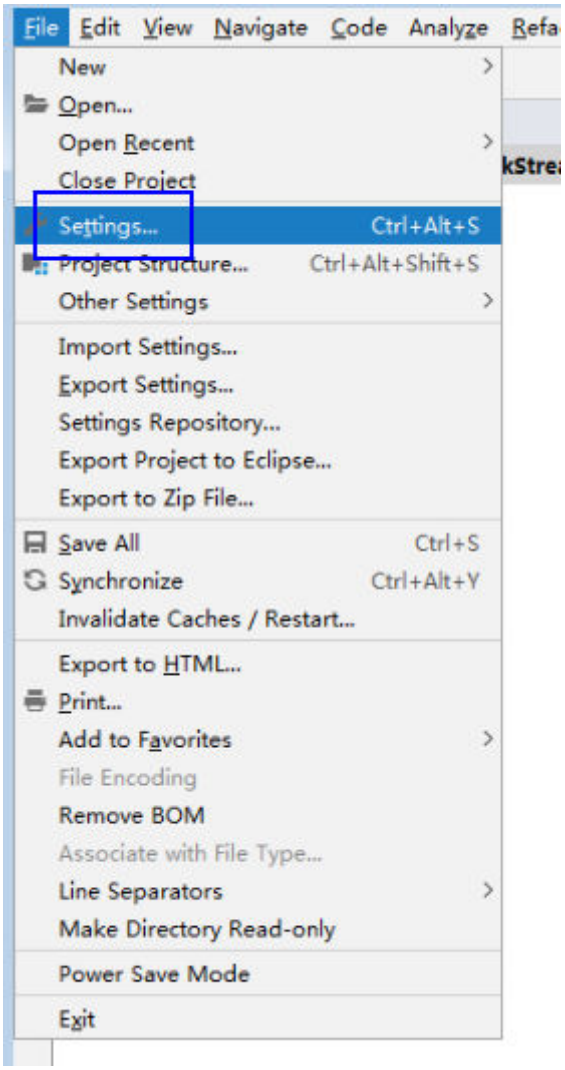
图 27-30 设置成功



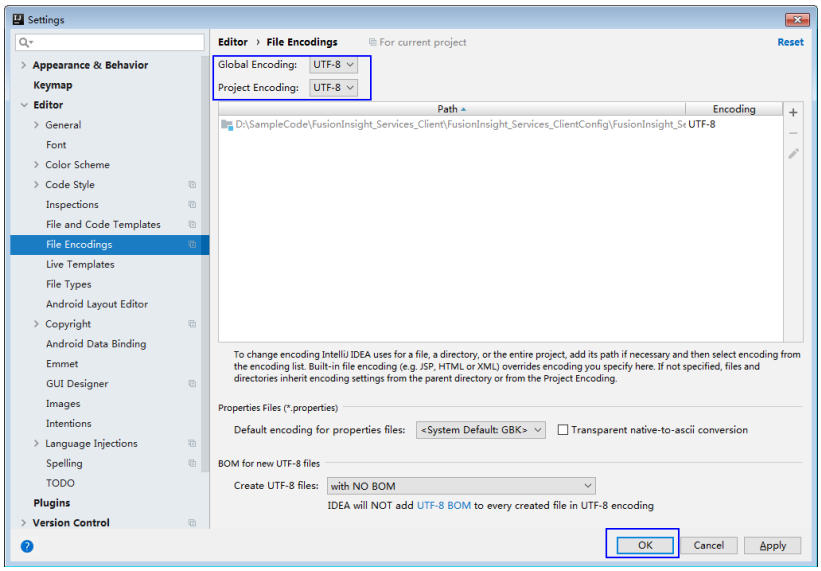
步骤7 设置IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IDEA首页，选择“File > Settings...”。

图 27-31 选择 Settings



2. 编码配置。
   a. 在“Settings”页面，展开“Editor”，选择“File Encodings”。
   b. 分别在右侧的“Global Encoding”和“Project Encoding”的下拉框中，选择“UTF-8”。
   c. 单击“Apply”应用配置。
   d. 单击“OK”完成编码配置。



----结束

样例代码路径说明

表 27-5 样例代码路径说明

样例代码项目	样例名称	样例语言
SparkJavaExample	Spark Core程序	Java
SparkScalaExample	Spark Core程序	Scala
SparkPythonExample	Spark Core程序	Python
SparkSQLJavaExample	Spark SQL程序	Java
SparkSQLScalaExample	Spark SQL程序	Scala
SparkSQLPythonExample	Spark SQL程序	Python
SparkThriftServerJavaExample	通过JDBC访问Spark SQL的程序	Java
SparkThriftServerScalaExample	通过JDBC访问Spark SQL的程序	Scala
SparkOnHbaseJavaExample-AvroSource	Spark on HBase 程序-操作Avro格式数据	Java
SparkOnHbaseScalaExample-AvroSource	Spark on HBase 程序-操作Avro格式数据	Scala
SparkOnHbasePythonExample-AvroSource	Spark on HBase 程序-操作Avro格式数据	Python
SparkOnHbaseJavaExample-HbaseSource	Spark on HBase 程序-操作HBase数据源	Java

样例代码项目	样例名称	样例语言
SparkOnHbaseScalaExample-HbaseSource	Spark on HBase 程序-操作HBase数据源	Scala
SparkOnHbasePythonExample-HbaseSource	Spark on HBase 程序-操作HBase数据源	Python
SparkOnHbaseJavaExample-JavaHBaseBulkPutExample	Spark on HBase 程序-BulkPut接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkPutExample	Spark on HBase 程序-BulkPut接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkPutExample	Spark on HBase 程序-BulkPut接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseBulkGetExample	Spark on HBase 程序-BulkGet接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkGetExample	Spark on HBase 程序-BulkGet接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkGetExample	Spark on HBase 程序-BulkGet接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseBulkDeleteExample	Spark on HBase 程序-BulkDelete接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkDeleteExample	Spark on HBase 程序-BulkDelete接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkDeleteExample	Spark on HBase 程序-BulkDelete接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseBulkLoadExample	Spark on HBase 程序-BulkLoad接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkLoadExample	Spark on HBase 程序-BulkLoad接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkLoadExample	Spark on HBase 程序-BulkLoad接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseForEachPartitionExample	Spark on HBase 程序-foreachPartition接口使用	Java
SparkOnHbaseScalaExample-HBaseForEachPartitionExample	Spark on HBase 程序-foreachPartition接口使用	Scala
SparkOnHbasePythonExample-HBaseForEachPartitionExample	Spark on HBase 程序-foreachPartition接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseDistributedScanExample	Spark on HBase 程序-分布式Scan HBase表	Java

样例代码项目	样例名称	样例语言
SparkOnHbaseScalaExample-HBaseDistributedScanExample	Spark on HBase 程序-分布式Scan HBase表	Scala
SparkOnHbasePythonExample-HBaseDistributedScanExample	Spark on HBase 程序-分布式Scan HBase表	Python
SparkOnHbaseJavaExample-JavaHBaseMapPartitionExample	Spark on HBase 程序-mapPartitions接口使用	Java
SparkOnHbaseScalaExample-HBaseMapPartitionExample	Spark on HBase 程序-mapPartitions接口使用	Scala
SparkOnHbasePythonExample-HBaseMapPartitionExample	Spark on HBase 程序-mapPartitions接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseStreamingBulkPutExample	Spark on HBase 程序-SparkStreaming批量写入HBase表	Java
SparkOnHbaseScalaExample-HBaseStreamingBulkPutExample	Spark on HBase 程序-SparkStreaming批量写入HBase表	Scala
SparkOnHbasePythonExample-HBaseStreamingBulkPutExample	Spark on HBase 程序-SparkStreaming批量写入HBase表	Python
SparkHbasetoHbaseJavaExample	从HBase读取数据再写入HBase	Java
SparkHbasetoHbaseScalaExample	从HBase读取数据再写入HBase	Scala
SparkHbasetoHbasePythonExample	从HBase读取数据再写入HBase	Python
SparkHivetoHbaseJavaExample	从Hive读取数据再写入HBase	Java
SparkHivetoHbaseScalaExample	从Hive读取数据再写入HBase	Scala
SparkHivetoHbasePythonExample	从Hive读取数据再写入HBase	Python
SparkStreamingKafka010JavaExample	Spark Streaming对接Kafka0-10程序	Java
SparkStreamingKafka010ScalaExample	Spark Streaming对接Kafka0-10程序	Scala
SparkStructuredStreamingJavaExample	Structured Streaming程序	Java
SparkStructuredStreamingScalaExample	Structured Streaming程序	Scala

样例代码项目	样例名称	样例语言
SparkStructuredStreamingPythonExample	Structured Streaming程序	Python
StructuredStreamingADScalaExample	Structured Streaming流流Join	Scala
StructuredStreamingStateScalaExample	Structured Streaming 状态操作	Scala
SparkOnMultiHbaseScalaExample	Spark同时访问两个HBase	Scala
SparkOnHudiJavaExample	使用Spark执行Hudi基本操作	Java
SparkOnHudiPythonExample	使用Spark执行Hudi基本操作	Python
SparkOnHudiScalaExample	使用Spark执行Hudi基本操作	Scala

27.4.4 新建 Spark 样例工程（可选）

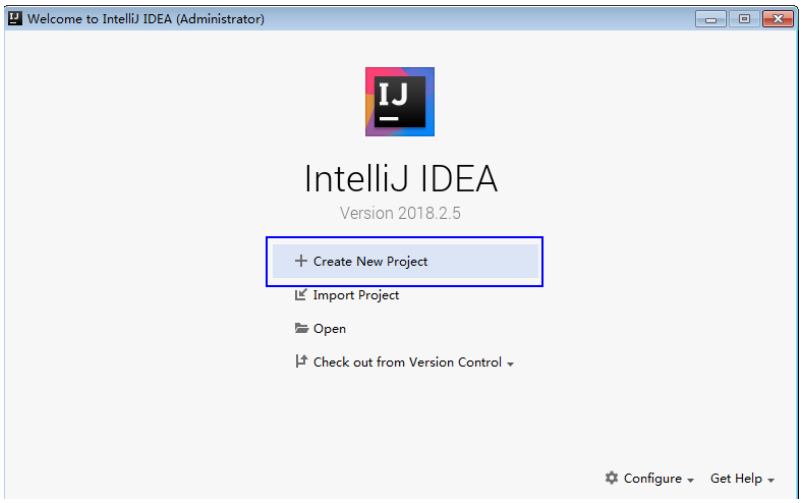
操作场景

除了导入Spark样例工程，您还可以使用IDEA新建一个Spark工程。如下步骤以创建一个Scala工程为例进行说明。

操作步骤

步骤1 打开IDEA工具，选择“Create New Project”。

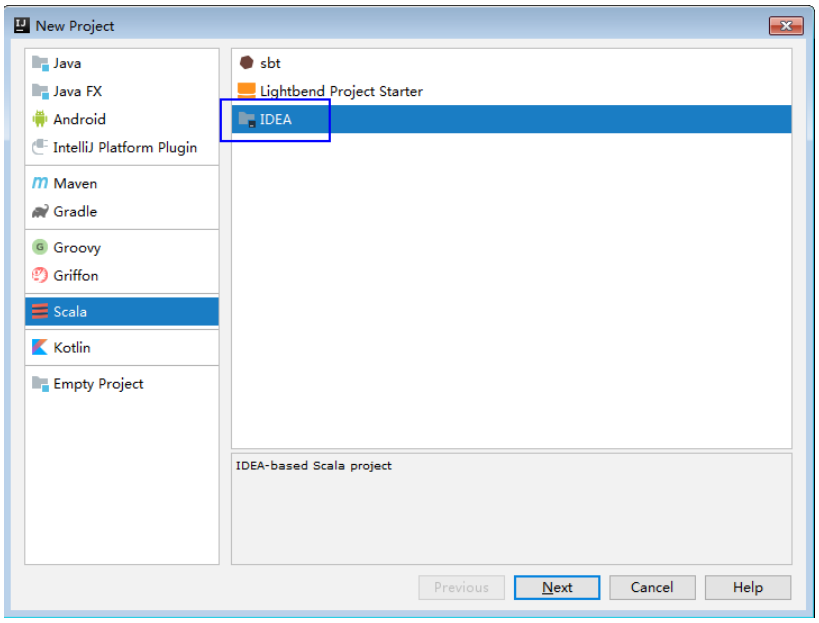
图 27-32 创建工程



步骤2 在“New Project”页面，选择“Scala”开发环境，并选择“IDEA”，然后单击“Next”。

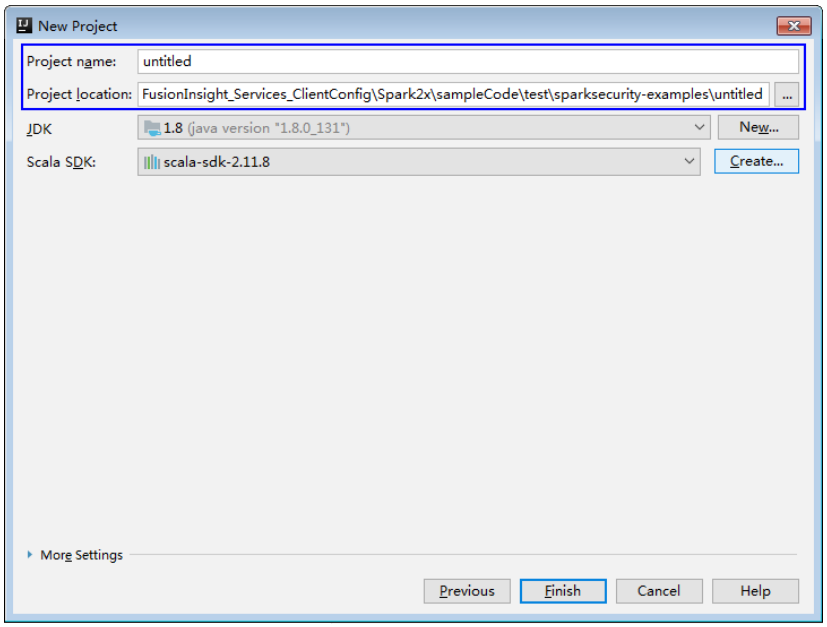
如果您需要新建Java语言的工程，选择对应参数即可。

图 27-33 选择开发环境



步骤3 在工程信息页面，填写工程名称和存放路径，设置JDK版本、Scala SDK版本，然后单击“Finish”完成工程创建。

图 27-34 填写工程信息



----结束

27.4.5 配置 Spark 应用安全认证

场景说明

在安全集群环境下，各个组件之间的相互通信不能够简单的互通，而需要在通信之前进行相互认证，以确保通信的安全性。

用户在开发Spark应用程序时，某些场景下，需要Spark与Hadoop、HBase等之间进行通信。那么Spark应用程序中需要写入安全认证代码，确保Spark程序能够正常运行。

安全认证有三种方式：

- 命令认证：

  提交Spark应用程序运行前，或者在使用CLI连接SparkSQL前，在Spark客户端执行如下命令获得认证。

  kinit *组件业务用户*

- 配置认证：

  可以通过以下3种方式的任意一种指定安全认证信息。

  - 在客户端的“spark-defaults.conf”配置文件中，配置“spark.kerberos.keytab”和“spark.kerberos.principal”参数指定认证信息。
  - 执行**bin/spark-submit**的命令中添加如下参数来指定认证信息。

    --conf spark.kerberos.keytab=*<keytab文件路径>* **--conf spark.kerberos.principal=***<Principal账号>*
  - 执行**bin/spark-submit**的命令中添加如下参数来指定认证信息。

    --keytab *<keytab文件路径>* **--principal** *<Principal账号>*

- 代码认证：

  通过获取客户端的principal和keytab文件在应用程序中进行认证。

在安全集群环境下，样例代码需要使用的认证方式如表27-6所示：

表 27-6 安全认证方式

样例代码	模式	安全认证方式
sparknormal-examples	yarn-client	命令认证、配置认证或代码认证，三种任选一种。
	yarn-cluster	命令认证或者配置认证，两种任选一种。
sparksecurity-examples（已包含安全认证代码）	yarn-client	代码认证。
	yarn-cluster	不支持。

说明

- 如上表所示，yarn-cluster模式中不支持在Spark工程代码中进行安全认证，因为需要在应用启动前已完成认证。
- 未提供Python样例工程的安全认证代码，推荐在运行应用程序命令中设置安全认证参数。

安全认证代码（Java 版）

目前样例代码统一调用LoginUtil类进行安全认证。安全登录流程请参见安全认证接口章节。

在Spark样例工程代码中，不同的样例工程，使用的认证代码不同，基本安全认证或带ZooKeeper认证。样例工程中使用的示例认证参数如**表27-7**所示，请根据实际情况修改对应参数值。

表 27-7 参数描述

参数	示例参数值	描述
userPrincipal	sparkuser	用户用于认证的账号Principal，使用**准备集群认证用户信息**中创建的用户。
userKeytabPath	/opt/FIclient/user.keytab	用户用于认证的Keytab文件，将准备的开发用户的user.keytab文件复制到示例参数值的路径下。
ZKServerPrincipal	zookeeper/hadoop.*<系统域名>*	ZooKeeper服务端principal。请联系管理员获取对应账号。

下列代码片段在样例工程中com.huawei.bigdata.spark.examples包的FemaleInfoCollection类的main方法中。

- 基本安全认证：

  Spark Core和Spark SQL程序不需要访问HBase或ZooKeeper，所以使用基本的安全认证代码即可。请在程序中添加如下代码，并根据实际情况设置安全认证相关参数：

  ```
  String userPrincipal = "sparkuser";
  String userKeytabPath = "/opt/FIclient/user.keytab";
  String krb5ConfPath = "/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf";
  Configuration hadoopConf = new Configuration();
  LoginUtil.login(userPrincipal, userKeytabPath, krb5ConfPath, hadoopConf);
  ```

- 带ZooKeeper认证：

  由于“Spark Streaming”、“通过JDBC访问Spark SQL”和“Spark on HBase”样例程序，不仅需要基础安全认证，还需要添加ZooKeeper服务端Principal才能完成安全认证。请在程序中添加如下代码，并根据实际情况设置安全认证相关参数：

  ```
  String userPrincipal = "sparkuser";
  String userKeytabPath = "/opt/FIclient/user.keytab";
  String krb5ConfPath = "/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf";
  String ZKServerPrincipal = "zookeeper/hadoop.<系统域名>";

  String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
  String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";

  Configuration hadoopConf = new Configuration();
  LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, userPrincipal, userKeytabPath);
  LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY, ZKServerPrincipal);
  LoginUtil.login(userPrincipal, userKeytabPath, krb5ConfPath, hadoopConf);
  ```

安全认证代码（Scala 版）

目前样例代码统一调用LoginUtil类进行安全认证，安全登录流程请参见统一认证章节。

在Spark样例工程代码中，不同的样例工程，使用的认证代码不同，基本安全认证或带ZooKeeper认证。样例工程中使用的示例认证参数如**表27-8**所示，请根据实际情况修改对应参数值。

表 27-8 参数描述

参数	示例参数值	描述
userPrincipal	sparkuser	用户用于认证的账号Principal，使用**准备集群认证用户信息**章节中创建的用户。
userKeytabPath	/opt/FIclient/user.keytab	用户用于认证的Keytab文件，将准备的开发用户的user.keytab文件复制到示例参数值的路径下。
ZKServerPrincipal	zookeeper/hadoop.*<系统域名>*	ZooKeeper服务端principal。请联系管理员获取对应账号。

- 基本安全认证：

  Spark Core和Spark SQL程序不需要访问HBase或ZooKeeper，所以使用基本的安全认证代码即可。请在程序中添加如下代码，并根据实际情况设置安全认证相关参数：

  ```
  val userPrincipal = "sparkuser"
  val userKeytabPath = "/opt/FIclient/user.keytab"
  val krb5ConfPath = "/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf"
  val hadoopConf: Configuration  = new Configuration()
  LoginUtil.login(userPrincipal, userKeytabPath, krb5ConfPath, hadoopConf);
  ```

- 带ZooKeeper认证：

  由于“Spark Streaming”、“通过JDBC访问Spark SQL”和“Spark on HBase”样例程序，不仅需要基础安全认证，还需要添加ZooKeeper服务端Principal才能完成安全认证。请在程序中添加如下代码，并根据实际情况设置安全认证相关参数：

  ```
  val userPrincipal = "sparkuser"
  val userKeytabPath = "/opt/FIclient/user.keytab"
  val krb5ConfPath = "/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf"
  val ZKServerPrincipal = "zookeeper/hadoop.<系统域名>"

  val ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME: String = "Client"
  val ZOOKEEPER_SERVER_PRINCIPAL_KEY: String = "zookeeper.server.principal"
  val hadoopConf: Configuration  = new Configuration();
  LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, userPrincipal, userKeytabPath)
  LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY, ZKServerPrincipal)
  LoginUtil.login(userPrincipal, userKeytabPath, krb5ConfPath, hadoopConf);
  ```

27.4.6 配置 Spark Python3 样例工程

操作场景

为了运行MRS产品Spark2x组件的Python3接口样例代码，需要完成下面的操作。

操作步骤

步骤1 客户端机器必须安装有Python3，其版本不低于3.6。

在客户端机器的命令行终端输入**python3**可查看Python版本号。如下显示Python版本为3.8.2。

```
Python 3.8.2 (default, Jun 23 2020, 10:26:03)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```


步骤2 客户端机器必须安装有setuptools，版本为47.3.1。

具体软件，请到对应的官方网站获取。

https://pypi.org/project/setuptools/#files

将下载的setuptools压缩文件复制到客户端机器上，解压后进入解压目录，在客户端机器的命令行终端执行**python3 setup.py install**。

如下内容表示安装setuptools的47.3.1版本成功。

```
Finished processing dependencies for setuptools==47.3.1
```

步骤3 安装Python客户端到客户端机器。

1. 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“python3-examples”。
2. 进入“python3-examples”文件夹。
3. 根据python3的版本，选择进入“dependency_python3.6”或“dependency_python3.7”或“dependency_python3.8”文件夹。
4. 执行**whereis easy_install**命令，找到easy_install程序路径。如果有多个路径，使用**easy_install --version**确认选择setuptools对应版本的easy_install，如/usr/local/bin/easy_install
5. 使用对应的easy_install命令，依次安装dependency_python3.x文件夹下的egg文件。如：

   /usr/local/bin/easy_install future-0.18.2-py3.8.egg

   输出以下关键内容表示安装egg文件成功。

   ```
   Finished processing dependencies for future==0.18.2
   ```

----结束

27.5 开发 Spark 应用

27.5.1 Spark Core 样例程序

27.5.1.1 Spark Core 样例程序开发思路

场景说明

假定用户有某个周末网民网购停留时间的日志文本，基于某些业务要求，要求开发Spark应用程序实现如下功能：

- 统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。
- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

  log1.txt：周六网民停留日志

  ```
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  FangBo,female,50
  ```

```
LiuYang,female,20
YuanJing,male,10
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

log2.txt：周日网民停留日志

```
LiuYang,female,20
YuanJing,male,10
CaiXuyu,female,50
FangBo,female,50
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
CaiXuyu,female,50
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
FangBo,female,50
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

数据规划

首先需要把原日志文件放置在HDFS系统里。

1. 本地新建两个文本文件input_data1.txt和input_data2.txt，将log1.txt中的内容复制保存到input_data1.txt，将log2.txt中的内容复制保存到input_data2.txt。
2. 在HDFS客户端路径下建立一个文件夹，“/tmp/input”，并上传input_data1.txt，input_data2.txt到此目录，命令如下：
   a. 在HDFS客户端，执行如下命令获取安全认证。

      cd *{客户端安装目录}*

      source bigdata_env

      kinit *<用于认证的业务用户>*

   b. 在Linux系统HDFS客户端使用命令***hadoop fs -mkdir /tmp/input***（hdfs dfs命令有同样的作用），创建对应目录。
   c. 进入到HDFS客户端下的“/tmp/input”目录，在Linux系统HDFS客户端使用命令***hadoop fs -put*** *input_data1.txt /tmp/input*和***hadoop fs -put*** *input_data2.txt /tmp/input*，上传数据文件。

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

- 读取原文件数据。
- 筛选女性网民上网时间数据信息。
- 汇总每个女性上网总时间。
- 筛选出停留时间大于两个小时的女性网民信息。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager

中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

1. 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
2. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

   说明

   - 编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。
   - 运行Python样例代码无需通过Maven打包，只需要上传user.keytab、krb5.conf 文件到客户端所在服务器上。
3. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“ /opt/female/” ）下。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Scala**和**Java**样例程序

  bin/spark-submit --class *com.huawei.bigdata.spark.examples.FemaleInfoCollection* **--master yarn --deploy-mode client** */opt/female/FemaleInfoCollection-1.0.jar <inputPath>*

  其中，<inputPath>指HDFS文件系统中input的路径。
- 运行**Python**样例程序

  说明

  - 由于Python样例代码中未给出认证信息，请在执行应用程序时通过配置项“--keytab”和“--principal”指定认证信息。

  bin/spark-submit --master yarn --deploy-mode client --keytab /opt/FIclient/user.keytab --principal sparkuser */opt/female/SparkPythonExample/collectFemaleInfo.py <inputPath>*

  其中，<inputPath>指HDFS文件系统中input的路径。

27.5.1.2 Spark Core 样例程序（Java）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection类：

```
//创建一个配置类SparkConf，然后创建一个SparkContext
SparkSession spark = SparkSession
```



```
    .builder()
    .appName("CollectFemaleInfo")
    .config("spark.some.config.option", "some-value")
    .getOrCreate();

  //读取原文件数据,每一行记录转成RDD里面的一个元素
  JavaRDD<String> data = spark.read()
    .textFile(args[0])
    .javaRDD();

  //将每条记录的每列切割出来，生成一个Tuple
  JavaRDD<Tuple3<String,String,Integer>> person = data.map(new
Function<String,Tuple3<String,String,Integer>>()
  {
      private static final long serialVersionUID = -2381522520231963249L;
      public Tuple3<String, String, Integer> call(String s) throws Exception
      {
          //按逗号分割一行数据
          String[] tokens = s.split(",");

          //将分割后的三个元素组成一个三元Tuple
          Tuple3<String, String, Integer> person = new Tuple3<String, String, Integer>(tokens[0], tokens[1],
Integer.parseInt(tokens[2]));
          return person;
      }
  });

  //使用filter函数筛选出女性网民上网时间数据信息
  JavaRDD<Tuple3<String,String,Integer>> female = person.filter(new
Function<Tuple3<String,String,Integer>, Boolean>()
  {
      private static final long serialVersionUID = -4210609503909770492L;

      public Boolean call(Tuple3<String, String, Integer> person) throws Exception
      {
          //根据第二列性别，筛选出是female的记录
          Boolean isFemale = person._2().equals("female");
          return isFemale;
      }
  });

  //汇总每个女性上网总时间
  JavaPairRDD<String, Integer> females = female.mapToPair(new PairFunction<Tuple3<String, String,
Integer>, String, Integer>()
  {
      private static final long serialVersionUID = 8313245377656164868L;

      public Tuple2<String, Integer> call(Tuple3<String, String, Integer> female) throws Exception
      {
          //取出姓名和停留时间两列，用于后面按名字求逗留时间的总和
          Tuple2<String, Integer> femaleAndTime = new  Tuple2<String, Integer>(female._1(), female._3());
          return femaleAndTime;
      }
  });
    JavaPairRDD<String, Integer> femaleTime = females.reduceByKey(new Function2<Integer, Integer,
Integer>()
  {
      private static final long serialVersionUID = -3271456048413349559L;

      public Integer call(Integer integer, Integer integer2) throws Exception
      {
          //将同一个女性的两次停留时间相加，求和
          return (integer + integer2);
      }
  });

  //筛选出停留时间大于两个小时的女性网民信息
  JavaPairRDD<String, Integer> rightFemales = females.filter(new Function<Tuple2<String, Integer>,
Boolean>()
```



```
{
    private static final long serialVersionUID = -3178168214712105171L;

    public Boolean call(Tuple2<String, Integer> s) throws Exception
    {
        //取出女性用户的总停留时间，并判断是否大于2小时
        if(s._2() > (2 * 60))
        {
            return true;
        }
        return false;
    }
});

//对符合的female信息进行打印显示
for(Tuple2<String, Integer> d: rightFemales.collect())
{
    System.out.println(d._1() + "," + d._2());
}
```

27.5.1.3 Spark Core 样例程序（Scala）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection：

样例：类CollectMapper

```
val spark = SparkSession
  .builder()
  .appName("CollectFemaleInfo")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

//读取数据。其是传入参数args(0)指定数据路径
val text = spark.sparkContext.textFile(args(0))
//筛选女性网民上网时间数据信息
val data = text.filter(_.contains("female"))
//汇总每个女性上网时间
val femaleData:RDD[(String,Int)] = data.map{line =>
    val t= line.split(',')
    (t(0),t(2).toInt)
}.reduceByKey(_ + _)
//筛选出时间大于两个小时的女性网民信息，并输出
val result = femaleData.filter(line => line._2 > 120)
result.collect().map(x => x._1 + ',' + x._2).foreach(println)
spark.stop()
```

27.5.1.4 Spark Core 样例程序（Python）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见collectFemaleInfo.py：

```
def contains(str, substr):
  if substr in str:
    return True
  return False

if __name__ == "__main__":
  if len(sys.argv) < 2:
    print "Usage: CollectFemaleInfo <file>"
    exit(-1)

  spark = SparkSession \
    .builder \
    .appName("CollectFemaleInfo") \
    .getOrCreate()

  """
  以下程序主要实现以下几步功能：
  1.读取数据。其是传入参数argv[1]指定数据路径 - text
  2.筛选女性网民上网时间数据信息 - filter
  3.汇总每个女性上网时间 - map/map/reduceByKey
  4.筛选出时间大于两个小时的女性网民信息 - filter
  """
  inputPath = sys.argv[1]
  result = spark.read.text(inputPath).rdd.map(lambda r: r[0])\
    .filter(lambda line: contains(line, "female")) \
    .map(lambda line: line.split(',')) \
    .map(lambda dataArr: (dataArr[0], int(dataArr[2]))) \
    .reduceByKey(lambda v1, v2: v1 + v2) \
    .filter(lambda tupleVal: tupleVal[1] > 120) \
    .collect()
  for (k, v) in result:
    print k + "," + str(v)

  # 停止SparkContext
  spark.stop()
```

27.5.2 Spark SQL 样例程序

27.5.2.1 Spark SQL 样例程序开发思路

场景说明

假定用户有某个周末网民网购停留时间的日志文本，基于某些业务要求，要求开发Spark应用程序实现如下功能：

- 统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。
- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

  log1.txt：周六网民停留日志

  ```
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  FangBo,female,50
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,50
  CaiXuyu,female,50
  FangBo,female,60
  ```

  log2.txt：周日网民停留日志

```
LiuYang,female,20
YuanJing,male,10
CaiXuyu,female,50
FangBo,female,50
GuoYijun,male,5
CaiXuyu,female,50
Liyuan,male,20
CaiXuyu,female,50
FangBo,female,50
LiuYang,female,20
YuanJing,male,10
FangBo,female,50
GuoYijun,male,50
CaiXuyu,female,50
FangBo,female,60
```

数据规划

首先需要把原日志文件放置在HDFS系统里。

1. 本地新建两个文本文件input_data1.txt和input_data2.txt，将log1.txt中的内容复制保存到input_data1.txt，将log2.txt中的内容复制保存到input_data2.txt。
2. 在HDFS客户端上建立一个文件夹，“/tmp/input”，并上传input_data1.txt，input_data2.txt到此目录，命令如下：
   a. 在HDFS客户端，执行如下命令获取安全认证。

      cd */opt/hadoopclient*

      source bigdata_env

      kinit *<用于认证的业务用户>*

   b. 在Linux系统HDFS客户端使用命令***hadoop fs -mkdir /tmp/input***（hdfs dfs命令有同样的作用），创建对应目录。
   c. 进入到HDFS客户端下的“/tmp/input”目录，在Linux系统HDFS客户端使用命令***hadoop fs -put*** *input_data1.txt /tmp/input*和***hadoop fs -put*** *input_data2.txt /tmp/input*，上传数据文件。

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

- 创建表，将日志文件数据导入到表中。
- 筛选女性网民，提取上网时间数据信息。
- 汇总每个女性上网总时间。
- 筛选出停留时间大于两个小时的女性网民信息。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

1. 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
2. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

   说明

   - 编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。
   - 运行Python样例代码无需通过Maven打包，只需要上传user.keytab、krb5.conf 文件到客户端所在服务器上。
3. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/female/”）下。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Scala**和**Java**样例程序

  bin/spark-submit --class *com.huawei.bigdata.spark.examples.FemaleInfoCollection* **--master yarn --deploy-mode client** */opt/female/SparkSqlScalaExample-1.0.jar <inputPath>*

  其中，<inputPath>指HDFS文件系统中input的路径。
- 运行**Python**样例程序

  说明

  - 由于Python样例代码中未给出认证信息，请在执行应用程序时通过配置项“--keytab”和“--principal”指定认证信息。

  bin/spark-submit --master yarn --deploy-mode client --keytab /opt/FIclient/user.keytab --principal sparkuser */opt/female/SparkPythonExample/SparkSQLPythonExample.py <inputPath>*

  其中，<inputPath>指HDFS文件系统中input的路径。

27.5.2.2 Spark SQL 样例程序（Java）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection：

```
public static void main(String[] args) throws Exception {
SparkSession spark = SparkSession
    .builder()
    .appName("CollectFemaleInfo")
    .config("spark.some.config.option", "some-value")
    .getOrCreate();
```

```
    // 通过隐式转换，将RDD转换成DataFrame
    JavaRDD<FemaleInfo> femaleInfoJavaRDD = spark.read().textFile(args[0]). javaRDD().map(
            new Function<String, FemaleInfo>() {
                @Override
                public FemaleInfo call(String line) throws Exception {
                    String[] parts = line.split(",");
                    FemaleInfo femaleInfo = new FemaleInfo();
                    femaleInfo.setName(parts[0]);
                    femaleInfo.setGender(parts[1]);
                    femaleInfo.setStayTime(Integer.parseInt(parts[2].trim()));
                    return femaleInfo;
                }
            });

    // 注册表。
    Dataset<ROW> schemaFemaleInfo = spark.createDataFrame(femaleInfoJavaRDD,FemaleInfo.class);
    schemaFemaleInfo.registerTempTable("FemaleInfoTable");

    // 执行SQL查询
    Dataset<ROW> femaleTimeInfo = spark.sql("select * from " +
            "(select name,sum(stayTime) as totalStayTime from FemaleInfoTable " +
            "where gender = 'female' group by name )" +
            " tmp where totalStayTime >120");

   // 显示结果。
    List<String> result = femaleTimeInfo.javaRDD().map(new Function<Row, String>() {
        public String call(Row row) {
            return  row.getString(0) + "," + row.getLong(1);
        }
    }).collect();
    System.out.println(result);
    spark.stop();
}
```

上面是简单示例，其它sparkSQL特性请参见如下链接：**http://spark.apache.org/docs/3.1.1/sql-programming-guide.html#running-sql-queries-programmatically**

27.5.2.3 Spark SQL 样例程序（Scala）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection：

```
object FemaleInfoCollection
 {
  //表结构，后面用来将文本数据映射为df
  case class FemaleInfo(name: String, gender: String, stayTime: Int)
  def main(args: Array[String]) {
    //配置Spark应用名称
    val spark = SparkSession
      .builder()
      .appName("FemaleInfo")
      .config("spark.some.config.option", "some-value")
      .getOrCreate()
    import spark.implicits._
    //通过隐式转换，将RDD转换成DataFrame，然后注册表
    spark.sparkContext.textFile(args(0)).map(_.split(","))
      .map(p => FemaleInfo(p(0), p(1), p(2).trim.toInt))
```

```
    .toDF.registerTempTable("FemaleInfoTable")
  //通过sql语句筛选女性上网时间数据, 对相同名字行进行聚合
  val femaleTimeInfo = spark.sql("select name,sum(stayTime) as stayTime from FemaleInfoTable where
gender = 'female' group by name")
  //筛选出时间大于两个小时的女性网民信息，并输出
  val c = femaleTimeInfo.filter("stayTime >= 120").collect().foreach(println)
  spark.stop()
 }
}
```

上面是简单示例，其它sparkSQL特性请参见如下链接：**http://spark.apache.org/docs/3.1.1/sql-programming-guide.html#running-sql-queries-programmatically**

27.5.2.4 Spark SQL 样例程序（Python）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见SparkSQLPythonExample：

```
# -*- coding:utf-8 -*-

import sys
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext


def contains(str1, substr1):
    if substr1 in str1:
        return True
    return False


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print "Usage: SparkSQLPythonExample.py <file>"
        exit(-1)

    # 初始化SparkSession和SQLContext
    sc = SparkSession.builder.appName("CollectFemaleInfo").getOrCreate()
    sqlCtx = SQLContext(sc)

    # RDD转换为DataFrame
    inputPath = sys.argv[1]
    inputRDD = sc.read.text(inputPath).rdd.map(lambda r: r[0])\
        .map(lambda line: line.split(","))\
        .map(lambda dataArr: (dataArr[0], dataArr[1], int(dataArr[2])))\
        .collect()
    df = sqlCtx.createDataFrame(inputRDD)

    # 注册表
    df.registerTempTable("FemaleInfoTable")

    # 执行SQL查询并显示结果
    FemaleTimeInfo = sqlCtx.sql("SELECT * FROM " +
              "(SELECT _1 AS Name,SUM(_3) AS totalStayTime FROM FemaleInfoTable " +
              "WHERE _2 = 'female' GROUP BY _1 )" +
              " WHERE totalStayTime >120").show()

    sc.stop()
```

27.5.3 通过 JDBC 访问 Spark SQL 样例程序

27.5.3.1 通过 JDBC 访问 Spark SQL 样例程序开发思路

场景说明

用户自定义JDBCServer的客户端，使用JDBC连接来进行表的创建、数据加载、查询和删除。

数据规划

将数据文件上传至HDFS中。

步骤1 确保以多主实例模式启动了JDBCServer服务，并至少有一个实例可连接客户端。在Linux系统HDFS客户端新建一个文本文件“data”，内容如下：

```
Miranda,32
Karlie,23
Candice,27
```

步骤2 在HDFS路径下建立一个目录，例如创建“/home”，并上传“data”文件到此目录，命令如下：

1. 登录HDFS客户端节点，执行如下命令：

   cd *{客户端安装目录}*

   source bigdata_env

   kinit *<用于认证的业务用户>*
2. 执行如下命令创建目录“/home”。

   hdfs dfs -mkdir /home
3. 执行如下命令上传数据文件。

   hdfs dfs -put data /home

步骤3 确保其对启动JDBCServer的用户有读写权限。

步骤4 确保客户端classpath下有“hive-site.xml”文件，且根据实际集群情况配置所需要的参数。JDBCServer相关参数详情，请参见**Spark JDBCServer接口介绍**。

----结束

开发思路

1. 在default数据库下创建child表。
2. 把“/home/data”的数据加载进child表中。
3. 查询child表中的数据。
4. 删除child表。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

- 将krb5.conf和user.keytab 文件上传到客户端多在服务器上。
- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

  说明

  编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/female/”）下。

运行任务

进入Spark客户端目录，使用java -cp命令运行代码（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Java**样例代码：

  java -cp *$SPARK_HOME/jars/*:$SPARK_HOME/jars/hive/*:$SPARK_HOME/conf:/opt/female/SparkThriftServerJavaExample-1.0.jar com.huawei.bigdata.spark.examples.ThriftServerQueriesTest $SPARK_HOME/conf/hive-site.xml $SPARK_HOME/conf/spark-defaults.conf*
- 运行**Scala**样例代码：

  java -cp *$SPARK_HOME/jars/*:$SPARK_HOME/jars/hive/*:$SPARK_HOME/conf:/opt/female/SparkThriftServerExample-1.0.jar com.huawei.bigdata.spark.examples.ThriftServerQueriesTest $SPARK_HOME/conf/hive-site.xml $SPARK_HOME/conf/spark-defaults.conf*

  说明

  集群开启ZooKeeper的SSL特性后（查看ZooKeeper服务的ssl.enabled参数），请在执行命令中添加-Dzookeeper.client.secure=true -Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty两项参数：

  java -Dzookeeper.client.secure=true -Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty **-cp** *$SPARK_HOME/jars/*:$SPARK_HOME/jars/hive/*:$SPARK_HOME/conf:/opt/female/SparkThriftServerJavaExample-1.0.jar com.huawei.bigdata.spark.examples.ThriftServerQueriesTest $SPARK_HOME/conf/hive-site.xml $SPARK_HOME/conf/spark-defaults.conf*

27.5.3.2 通过 JDBC 访问 Spark SQL 样例程序（Java）

功能简介

使用自定义客户端的JDBC接口提交数据分析任务，并返回结果。

样例代码

步骤1 定义SQL语句。SQL语句必须为单条语句，注意其中不能包含“;”。示例：

```
ArrayList<String> sqlList = new ArrayList<String>();
sqlList.add("CREATE TABLE CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE");
sqlList.add("LOAD DATA INPATH 'hdfs://hacluster/home/data' INTO TABLE CHILD");
sqlList.add("SELECT * FROM child");
```

```
sqlList.add("DROP TABLE child");
executeSql(url, sqlList);
```

步骤2 拼接JDBC URL。

```
String securityConfig = ";saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.<系统域名>@<系统域名>;user.principal=sparkuser;user.keytab=/opt/FIclient/user.keytab;";
Configuration config = new Configuration();
config.addResource(new Path(args[0]));
String zkUrl = config.get("spark.deploy.zookeeper.url");

String zkNamespace = null;
zkNamespace = fileInfo.getProperty("spark.thriftserver.zookeeper.namespace");
if (zkNamespace != null) {
   //从配置项中删除冗余字符
   zkNamespace = zkNamespace.substring(1);
}

StringBuilder sb = new StringBuilder("jdbc:hive2://"
      + zkUrl
      + ";serviceDiscoveryMode=zooKeeper;zooKeeperNamespace="
      + zkNamespace
      + securityConfig);
String url = sb.toString();
```

说明

由于KERBEROS认证成功后，默认有效期为1天，超过有效期后，如果客户端需要和JDBCServer新建连接则需要重新认证，否则就会执行失败。因此，若长期执行应用过程中需要新建连接，用户需要在“url”中添加user.principal和user.keytab认证信息，以保证每次建立连接时认证成功，例如，“url”中需要加上“user.principal=sparkuser;user.keytab=/opt/client/user.keytab”。

步骤3 加载Hive JDBC驱动。

```
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
```

步骤4 获取JDBC连接，执行HQL，输出查询的列名和结果到控制台，关闭JDBC连接。

连接字符串中的“zk.quorum”也可以使用配置文件中的配置项“spark.deploy.zookeeper.url”来代替。

在网络拥塞的情况下，您还可以设置客户端与JDBCServer连接的超时时间，可以避免客户端由于无限等待服务端的返回而挂起。使用方式如下：

在执行“DriverManager.getConnection”方法获取JDBC连接前，添加“DriverManager.setLoginTimeout(n)”方法来设置超时时长，其中n表示等待服务返回的超时时长，单位为秒，类型为Int，默认为“0”（表示永不超时）。

```
static void executeSql(String url, ArrayList<String> sqls) throws ClassNotFoundException, SQLException {
      try {
         Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
      } catch (Exception e) {
         e.printStackTrace();
      }
      Connection connection = null;
      PreparedStatement statement = null;

      try {
         connection = DriverManager.getConnection(url);
         for (int i =0 ; i < sqls.size(); i++) {
            String sql = sqls.get(i);
            System.out.println("---- Begin executing sql: " + sql +  " ----");
            statement = connection.prepareStatement(sql);
            ResultSet result = statement.executeQuery();
            ResultSetMetaData resultMetaData = result.getMetaData();
            Integer colNum = resultMetaData.getColumnCount();
            for (int j =1; j <= colNum; j++) {
               System.out.println(resultMetaData.getColumnLabel(j) + "\t");
```



```
            }
            System.out.println();

            while (result.next()) {
                for (int j =1; j <= colNum; j++){
                    System.out.println(result.getString(j) + "\t");
                }
                System.out.println();
            }
            System.out.println("---- Done executing sql: " + sql +  " ----");
        }


    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (null != statement) {
            statement.close();
        }
        if (null != connection) {
            connection.close();
        }
    }
}
```

----结束

27.5.3.3 通过 JDBC 访问 Spark SQL 样例程序（Scala）

功能简介

使用自定义客户端的JDBC接口提交数据分析任务，并返回结果。

样例代码

步骤1 定义SQL语句。SQL语句必须为单条语句，注意其中不能包含“;”。示例：

```
val sqlList = new ArrayBuffer[String]
sqlList += "CREATE TABLE CHILD (NAME STRING, AGE INT) " +
"ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE"
sqlList += "LOAD DATA INPATH 'hdfs://hacluster/home/data' INTO TABLE CHILD"
sqlList += "SELECT * FROM child"
sqlList += "DROP TABLE child"
```

步骤2 拼接JDBC URL。

```
val securityConfig = ";saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.<系统域名>@<系统域名
>;user.principal=sparkuser;user.keytab=/opt/FIclient/user.keytab;"
val config: Configuration = new Configuration()
config.addResource(new Path(args(0)))
val zkUrl = config.get("spark.deploy.zookeeper.url")

var zkNamespace: String = null
zkNamespace = fileInfo.getProperty("spark.thriftserver.zookeeper.namespace")
//从配置项中删除冗余字符
if (zkNamespace != null) zkNamespace = zkNamespace.substring(1)
val sb = new StringBuilder("jdbc:hive2://"
  + zkUrl
  + ";serviceDiscoveryMode=zooKeeper;zooKeeperNamespace="
  + zkNamespace
  + securityConfig)
val url = sb.toString()
```


说明

由于KERBEROS认证成功后，默认有效期为1天，超过有效期后，如果客户端需要和JDBCServer新建连接则需要重新认证，否则就会执行失败。因此，若长期执行应用过程中需要新建连接，用户需要在“url”中添加user.principal和user.keytab认证信息，以保证每次建立连接时认证成功，例如，“url”中需要加上“user.principal=sparkuser;user.keytab=/opt/client/user.keytab”。

步骤3 加载Hive JDBC驱动，获取JDBC连接，执行HQL，输出查询的列名和结果到控制台，关闭JDBC连接。

连接字符串中的“zk.quorum”也可以使用配置文件中的配置项“spark.deploy.zookeeper.url”来代替。

在网络拥塞的情况下，您还可以设置客户端与JDBCServer连接的超时时间，可以避免客户端由于无限等待服务端的返回而挂起。使用方式如下：

在执行“DriverManager.getConnection”方法获取JDBC连接前，添加“DriverManager.setLoginTimeout(n)”方法来设置超时时长，其中n表示等待服务返回的超时时长，单位为秒，类型为Int，默认为“0”（表示永不超时）。

```
def executeSql(url: String, sqls: Array[String]): Unit = {
//加载Hive JDBC驱动。
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance()

var connection: Connection = null
var statement: PreparedStatement = null
try {
  connection = DriverManager.getConnection(url)
  for (sql <- sqls) {
    println(s"---- Begin executing sql: $sql ----")
    statement = connection.prepareStatement(sql)

    val result = statement.executeQuery()

    val resultMetaData = result.getMetaData
    val colNum = resultMetaData.getColumnCount
    for (i <- 1 to colNum) {
      print(resultMetaData.getColumnLabel(i) + "\t")
    }
    println()

    while (result.next()) {
      for (i <- 1 to colNum) {
        print(result.getString(i) + "\t")
      }
      println()
    }
    println(s"---- Done executing sql: $sql ----")
  }
} finally {
  if (null != statement) {
    statement.close()
  }

  if (null != connection) {
    connection.close()
  }
}
}
```

----结束

27.5.4 Spark 读取 HBase 表样例程序

27.5.4.1 操作 Avro 格式数据

场景说明

用户可以在Spark应用程序中以数据源的方式去使用HBase，本例中将数据以Avro格式存储在HBase中，并从中读取数据以及对读取的数据进行过滤等操作。

数据规划

在客户端执行**hbase shell**，进入HBase命令行，使用下面的命令创建样例代码中要使用的HBase表：

create *'ExampleAvrotable','rowkey','cf1'*（如果表已经存在，则每次执行**提交命令**前需清空表里的数据：truncate *'ExampleAvrotable'*）

create *'ExampleAvrotableInsert','rowkey','cf1'*（如果表已经存在，则每次执行**提交命令**前需清空表里的数据：truncate *'ExampleAvrotable'*）

开发思路

1. 创建RDD。
2. 以数据源的方式操作HBase，将上面生成的RDD写入HBase表中。
3. 读取HBase表中的数据，并且对其进行简单的操作。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行。

- yarn-client模式：

java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

bin/spark-submit --master yarn --deploy-mode client --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.AvroSource SparkOnHbaseJavaExample.jar

python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

bin/spark-submit --master yarn --deploy-mode client --conf spark.yarn.user.classpath.first=true --jars SparkOnHbaseJavaExample.jar,/opt/female/protobuf-java-2.5.0.jar AvroSource.py

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.AvroSource --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --conf spark.yarn.user.classpath.first=true --jars SparkOnHbaseJavaExample.jar,/opt/female/protobuf-java-2.5.0.jar AvroSource.py

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中的AvroSource文件：

```
public static void main(JavaSparkContext jsc) throws IOException {
    LoginUtil.loginWithUserKeytab();
    SQLContext sqlContext = new SQLContext(jsc);
    Configuration hbaseconf = new HBaseConfiguration().create();
    JavaHBaseContext hBaseContext = new JavaHBaseContext(jsc, hbaseconf);
    List list = new ArrayList<AvroHBaseRecord>();
    for(int i=0; i<=255 ; ++i){
       list.add(AvroHBaseRecord.apply(i));
    }
    try{
       Map<String, String> map = new HashMap<String, String>();
       map.put(HBaseTableCatalog.tableCatalog(), catalog);
       map.put(HBaseTableCatalog.newTable(), "5");
       sqlContext.createDataFrame(list,
AvroHBaseRecord.class).write().options(map).format("org.apache.hadoop.hbase.spark").save();
       Dataset<Row> ds = withCatalog(sqlContext,catalog);
       ds.show();
       ds.printSchema();
       ds.registerTempTable("ExampleAvrotable");
       Dataset<Row> c= sqlContext.sql("select count(1) from ExampleAvrotable");
       c.show();
       Dataset<Row> filtered = ds.select("col0", "col1.favorite_array").where("col0 = 'name1'");
       filtered.show();
       java.util.List<Row> collected = filtered.collectAsList();
       if (collected.get(0).get(1).toString().equals("number1")) {
          throw new UserCustomizedSampleException("value invalid", new Throwable());
       }
       if (collected.get(0).get(1).toString().equals("number2")) {
          throw new UserCustomizedSampleException("value invalid", new Throwable());
```



```
            }
            Map avroCatalogInsertMap = new HashMap<String,String>();
            avroCatalogInsertMap.put("avroSchema" , AvroHBaseRecord.schemaString);
            avroCatalogInsertMap.put(HBaseTableCatalog.tableCatalog(), avroCatalogInsert);
            ds.write().options(avroCatalogInsertMap).format("org.apache.hadoop.hbase.spark").save();
            Dataset<Row> newDS = withCatalog(sqlContext,avroCatalogInsert);
            newDS.show();
            newDS.printSchema();
            if (newDS.count() != 256) {
                throw new UserCustomizedSampleException("value invalid", new Throwable());
            }
            ds.filter("col1.name = 'name5' || col1.name <= 'name5'").select("col0","col1.favorite_color",
"col1.favorite_number").show();
        } finally{
            jsc.stop();
        }
    }
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中的AvroSource文件：

```
 def main(args: Array[String]) {
    LoginUtil.loginWithUserKeytab()
    val sparkConf = new SparkConf().setAppName("AvroSourceExample")
    val sc = new SparkContext(sparkConf)
    val sqlContext = new SQLContext(sc)
    val hbaseConf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, hbaseConf)
    import sqlContext.implicits._
    def withCatalog(cat: String): DataFrame = {
      sqlContext
        .read
        .options(Map("avroSchema" -> AvroHBaseRecord.schemaString, HBaseTableCatalog.tableCatalog ->
avroCatalog))
        .format("org.apache.hadoop.hbase.spark")
        .load()
    }
    val data = (0 to 255).map { i =>
      AvroHBaseRecord(i)
    }
    try {
      sc.parallelize(data).toDF.write.options(
        Map(HBaseTableCatalog.tableCatalog -> catalog, HBaseTableCatalog.newTable -> "5"))
        .format("org.apache.hadoop.hbase.spark")
        .save()

      val df = withCatalog(catalog)
      df.show()
      df.printSchema()
      df.registerTempTable("ExampleAvrotable")
      val c = sqlContext.sql("select count(1) from ExampleAvrotable")
      c.show()

      val filtered = df.select($"col0", $"col1.favorite_array").where($"col0" === "name001")
      filtered.show()
      val collected = filtered.collect()
      if (collected(0).getSeq[String](1)(0) != "number1") {
        throw new UserCustomizedSampleException("value invalid")
      }
      if (collected(0).getSeq[String](1)(1) != "number2") {
        throw new UserCustomizedSampleException("value invalid")
      }

      df.write.options(
        Map("avroSchema" -> AvroHBaseRecord.schemaString, HBaseTableCatalog.tableCatalog ->
avroCatalogInsert,
          HBaseTableCatalog.newTable -> "5"))
```

```
      .format("org.apache.hadoop.hbase.spark")
      .save()
    val newDF = withCatalog(avroCatalogInsert)
    newDF.show()
    newDF.printSchema()
    if (newDF.count() != 256) {
      throw new UserCustomizedSampleException("value invalid")
    }
    df.filter($"col1.name" === "name005" || $"col1.name" <= "name005")
      .select("col0", "col1.favorite_color", "col1.favorite_number")
      .show()
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中的AvroSource文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("AvroSourceExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.datasources.AvroSource')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.AvroSource().execute(spark._jsc)
# 停止SparkSession
spark.stop()
```

27.5.4.2 操作 HBase 数据源

场景说明

用户可以在Spark应用程序中以数据源的方式去使用HBase，将dataFrame写入HBase中，并从HBase读取数据以及对读取的数据进行过滤等操作。

数据规划

在客户端执行**hbase shell**，进入HBase命令行，使用下面的命令创建样例代码中要使用的HBase表：

create *'HBaseSourceExampleTable','rowkey','cf1','cf2','cf3','cf4','cf5','cf6','cf7', 'cf8'*

开发思路

1. 创建RDD。
2. 以数据源的方式操作HBase，将上面生成的RDD写入HBase表中。
3. 读取HBase表中的数据，并且对其进行简单的操作。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.HBaseSource SparkOnHbaseJavaExample.jar

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --conf spark.yarn.user.classpath.first=true --jars SparkOnHbaseJavaExample.jar,/opt/female/protobuf-java-2.5.0.jar HBaseSource.py

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.HBaseSource --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --files /opt/user.keytab,/opt/krb5.conf --conf spark.yarn.user.classpath.first=true --jars

SparkOnHbaseJavaExample.jar,/opt/female/protobuf-java-2.5.0.jar
HBaseSource.py

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中的HBaseSource文件：

```
  public static void main(String args[]) throws IOException{
    LoginUtil.loginWithUserKeytab();
    SparkConf sparkConf = new SparkConf().setAppName("HBaseSourceExample");
    JavaSparkContext jsc = new JavaSparkContext(sparkConf);
    SQLContext sqlContext = new SQLContext(jsc);


    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc,conf);
    try{
      List<HBaseRecord> list = new ArrayList<HBaseRecord>();
      for(int i=0 ; i<256; i++){
        list.add(new HBaseRecord(i));
      }
      Map map = new HashMap<String, String>();
      map.put(HBaseTableCatalog.tableCatalog(), cat);
      map.put(HBaseTableCatalog.newTable(), "5");
      System.out.println("Before insert data into hbase table");
      sqlContext.createDataFrame(list,
HBaseRecord.class).write().options(map).format("org.apache.hadoop.hbase.spark").save();
      Dataset<Row> ds = withCatalog(sqlContext, cat);
      System.out.println("After insert data into hbase table");
      ds.printSchema();
      ds.show();
      ds.filter("key <= 'row5'").select("key","col1").show();
      ds.registerTempTable("table1");
      Dataset<Row> tempDS = sqlContext.sql("select count(col1) from table1 where key < 'row5'");
      tempDS.show();
    } finally {
      jsc.stop();
    }
  }
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中的HBaseSource文件：

```
def main(args: Array[String]) {
  LoginUtil.loginWithUserKeytab()
  val sparkConf = new SparkConf().setAppName("HBaseSourceExample")
  val sc = new SparkContext(sparkConf)
  val sqlContext = new SQLContext(sc)
  val conf = HBaseConfiguration.create()
  val hbaseContext = new HBaseContext(sc,conf)
  import sqlContext.implicits._
  def withCatalog(cat: String): DataFrame = {
    sqlContext
      .read
      .options(Map(HBaseTableCatalog.tableCatalog->cat))
      .format("org.apache.hadoop.hbase.spark")
      .load()
  }
  val data = (0 to 255).map { i =>
    HBaseRecord(i)
  }
  try{
    sc.parallelize(data).toDF.write.options(
```



```
      Map(HBaseTableCatalog.tableCatalog -> cat, HBaseTableCatalog.newTable -> "5"))
      .format("org.apache.hadoop.hbase.spark")
      .save()
    val df = withCatalog(cat)
    df.show()
    df.filter($"col0" <= "row005")
      .select($"col0", $"col1").show
    df.registerTempTable("table1")
    val c = sqlContext.sql("select count(col1) from table1 where col0 < 'row050'")
    c.show()
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中的HBaseSource文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("HBaseSourceExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.datasources.HBaseSource')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.HBaseSource().execute(spark._jsc)
# 停止SparkSession
spark.stop()
```

27.5.4.3 BulkPut 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将构建的RDD写入HBase中。

数据规划

在客户端执行**hbase shell**，进入HBase命令行，使用下面的命令创建样例代码中要使用的HBase表：

create '*bulktable***','***cf1***'**

开发思路

1. 创建RDD。
2. 以HBaseContext的方式操作HBase，将上面生成的RDD写入HBase表中。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端$SPARK_HOME目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkPutExample SparkOnHbaseJavaExample.jar bulktable cf1

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseBulkPutExample.py bulktable cf1

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkPutExample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar bulktable cf1

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseBulkPutExample.py bulktable cf1

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中的JavaHBaseBulkPutExample文件：

```
public static void main(String[] args) throws Exception{
  if (args.length < 2) {
    System.out.println("JavaHBaseBulkPutExample  " +
          "{tableName} {columnFamily}");
    return;
  }
  LoginUtil.loginWithUserKeytab();
  String tableName = args[0];
  String columnFamily = args[1];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkPutExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<String> list = new ArrayList<String>(5);
    list.add("1," + columnFamily + ",1,1");
    list.add("2," + columnFamily + ",1,2");
    list.add("3," + columnFamily + ",1,3");
    list.add("4," + columnFamily + ",1,4");
    list.add("5," + columnFamily + ",1,5");
    list.add("6," + columnFamily + ",1,6");
    list.add("7," + columnFamily + ",1,7");
    list.add("8," + columnFamily + ",1,8");
    list.add("9," + columnFamily + ",1,9");
    list.add("10," + columnFamily + ",1,10");
    JavaRDD<String> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.bulkPut(rdd,
          TableName.valueOf(tableName),
          new PutFunction());
    System.out.println("Bulk put into Hbase successfully!");
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkPutExample文件：

```
def main(args: Array[String]) {
  if (args.length < 2) {
    System.out.println("HBaseBulkPutTimestampExample {tableName} {columnFamily} are missing an
argument")
    return
  }
  LoginUtil.loginWithUserKeytab()
  val tableName = args(0)
  val columnFamily = args(1)
  val sparkConf = new SparkConf().setAppName("HBaseBulkPutTimestampExample " +
    tableName + " " + columnFamily)
  val sc = new SparkContext(sparkConf)
  try {
    val rdd = sc.parallelize(Array(
      (Bytes.toBytes("1"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("1")))),
      (Bytes.toBytes("2"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("2")))),
      (Bytes.toBytes("3"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("3")))),
      (Bytes.toBytes("4"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("4")))),
      (Bytes.toBytes("5"),
```



```
      Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("5")))),
    (Bytes.toBytes("6"),
      Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("6")))),
    (Bytes.toBytes("7"),
      Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("7")))),
    (Bytes.toBytes("8"),
      Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("8")))),
    (Bytes.toBytes("9"),
      Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("9")))),
    (Bytes.toBytes("10"),
      Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("10"))))))
  val conf = HBaseConfiguration.create()
  val timeStamp = System.currentTimeMillis()
  val hbaseContext = new HBaseContext(sc, conf)
  hbaseContext.bulkPut[(Array[Byte], Array[(Array[Byte], Array[Byte], Array[Byte])])](rdd,
    TableName.valueOf(tableName),
    (putRecord) => {
      val put = new Put(putRecord._1)
      putRecord._2.foreach((putValue) => put.addColumn(putValue._1, putValue._2,
        timeStamp, putValue._3))
      put
    })
 } finally {
  sc.stop()
 }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkPutExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkPutExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkPutExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseBulkPutExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

27.5.4.4 BulkGet 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将要获取的数据的rowKey构造成rdd，然后通过HBaseContext的bulkGet接口获取对HBase表上这些rowKey对应的数据。

数据规划

基于**BulkPut接口使用**章节中创建的HBase表及其中的数据进行操作。

开发思路

1. 创建包含了要获取的rowkey信息的RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的bulkGet接口获取HBase表上这些rowKey对应的数据。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkGetExample SparkOnHbaseJavaExample.jar bulktable

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseBulkGetExample.py bulktable

- yarn-cluster模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkGetExample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar bulktable

python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseBulkGetExample.py bulktable

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中HBaseBulkGetExample文件：

```
public static void main(String[] args) throws IOException{
  if (args.length < 1) {
    System.out.println("JavaHBaseBulkGetExample  {tableName}");
    return;
  }
  LoginUtil.loginWithUserKeytab();
  String tableName = args[0];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkGetExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<byte[]> list = new ArrayList<byte[]>(5);
    list.add(Bytes.toBytes("1"));
    list.add(Bytes.toBytes("2"));
    list.add(Bytes.toBytes("3"));
    list.add(Bytes.toBytes("4"));
    list.add(Bytes.toBytes("5"));
    JavaRDD<byte[]> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    List resultList = hbaseContext.bulkGet(TableName.valueOf(tableName), 2, rdd, new GetFunction(),
          new ResultFunction()).collect();
    for(int i =0 ;i<resultList.size();i++){
      System.out.println(resultList.get(i));
    }
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkGetExample文件：

```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseBulkGetExample {tableName} missing an argument")
    return
  }
  LoginUtil.loginWithUserKeytab()
  val tableName = args(0)
  val sparkConf = new SparkConf().setAppName("HBaseBulkGetExample " + tableName)
  val sc = new SparkContext(sparkConf)
  try {
    //[(Array[Byte])]
    val rdd = sc.parallelize(Array(
      Bytes.toBytes("1"),
      Bytes.toBytes("2"),
      Bytes.toBytes("3"),
      Bytes.toBytes("4"),
      Bytes.toBytes("5"),
      Bytes.toBytes("6"),
      Bytes.toBytes("7")))
    val conf = HBaseConfiguration.create()
```

```
    val hbaseContext = new HBaseContext(sc, conf)
    val getRdd = hbaseContext.bulkGet[Array[Byte], String](
      TableName.valueOf(tableName),
      2,
      rdd,
      record => {
        System.out.println("making Get")
        new Get(record)
      },
      (result: Result) => {
        val it = result.listCells().iterator()
        val b = new StringBuilder
        b.append(Bytes.toString(result.getRow) + ":")
        while (it.hasNext) {
          val cell = it.next()
          val q = Bytes.toString(CellUtil.cloneQualifier(cell))
          if (q.equals("counter")) {
            b.append("(" + q + "," + Bytes.toLong(CellUtil.cloneValue(cell)) + ")")
          } else {
            b.append("(" + q + "," + Bytes.toString(CellUtil.cloneValue(cell)) + ")")
          }
        }
        b.toString()
      })
    getRdd.collect().foreach(v => println(v))
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkGetExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkGetExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkGetExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseBulkGetExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

27.5.4.5 BulkDelete 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将要删除的数据的rowKey构造成rdd，然后通过HBaseContext的bulkDelete接口对HBase表上这些rowKey对应的数据进行删除。

数据规划

基于**BulkPut接口使用**章节创建的HBase表及其中的数据进行操作。

开发思路

1. 创建包含了要删除的rowkey信息的RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的bulkDelete接口对HBase表上这些rowKey对应的数据进行删除。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkDeleteExample SparkOnHbaseJavaExample.jar bulktable

  python版本（文件名等于实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseBulkDeleteExample.py bulktable

- yarn-cluster模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkDeleteE

xample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar bulktable

python版本（文件名等于实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseBulkDeleteExample.py bulktable

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中HBaseBulkDeleteExample文件：

```
public static void main(String[] args) throws IOException {
  if (args.length < 1) {
    System.out.println("JavaHBaseBulkDeleteExample  {tableName}");
    return;
  }
  LoginUtil.loginWithUserKeytab();
  String tableName = args[0];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkDeleteExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<byte[]> list = new ArrayList<byte[]>(5);
    list.add(Bytes.toBytes("1"));
    list.add(Bytes.toBytes("2"));
    list.add(Bytes.toBytes("3"));
    list.add(Bytes.toBytes("4"));
    list.add(Bytes.toBytes("5"));
    JavaRDD<byte[]> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.bulkDelete(rdd,
          TableName.valueOf(tableName), new DeleteFunction(), 4);
    System.out.println("Bulk Delete successfully!");
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkDeleteExample文件：

```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseBulkDeleteExample {tableName} missing an argument")
    return
  }
  LoginUtil.loginWithUserKeytab()
  val tableName = args(0)
  val sparkConf = new SparkConf().setAppName("HBaseBulkDeleteExample " + tableName)
  val sc = new SparkContext(sparkConf)
  try {
    //[Array[Byte]]
    val rdd = sc.parallelize(Array(
      Bytes.toBytes("1"),
      Bytes.toBytes("2"),
      Bytes.toBytes("3"),
      Bytes.toBytes("4"),
      Bytes.toBytes("5")
    ))
    val conf = HBaseConfiguration.create()
```

```
    val hbaseContext = new HBaseContext(sc, conf)
    hbaseContext.bulkDelete[Array[Byte]](rdd,
      TableName.valueOf(tableName),
      putRecord => new Delete(putRecord),
      4)
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkDeleteExample文件：

```
 def main(args: Array[String]) {
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkDeleteExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkDeleteExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseBulkDeleteExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

27.5.4.6 BulkLoad 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将要插入的数据的rowKey构造成rdd，然后通过HBaseContext的bulkLoad接口将rdd写入HFile中。将生成的HFile文件导入HBase表的操作采用如下格式的命令，不属于本接口范围，不在此进行详细说明：

hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles {hfilePath} {tableName}

数据规划

1. 在客户端执行：**hbase shell**命令进入HBase命令行。
2. 使用下面的命令创建HBase表：

   create *'bulkload-table-test','f1','f2'*

开发思路

1. 将要导入的数据构造成RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的bulkLoad接口将rdd写入HFile中。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkLoadExample SparkOnHbaseJavaExample.jar /tmp/hfile bulkload-table-test

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseBulkLoadExample.py /tmp/hfile bulkload-table-test

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkLoadExample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar /tmp/hfile bulkload-table-test

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseBulkLoadExample.py /tmp/hfile bulkload-table-test

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseBulkLoadExample文件：

```
public static void main(String[] args) throws IOException{
  if (args.length < 2) {
    System.out.println("JavaHBaseBulkLoadExample {outputPath} {tableName}");
    return;
  }
  LoginUtil.loginWithUserKeytab();
  String outputPath = args[0];
  String tableName = args[1];
  String columnFamily1 = "f1";
  String columnFamily2 = "f2";
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkLoadExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<String> list= new ArrayList<String>();
    // row1
    list.add("1," + columnFamily1 + ",b,1");
    // row3
    list.add("3," + columnFamily1 + ",a,2");
    list.add("3," + columnFamily1 + ",b,1");
    list.add("3," + columnFamily2 + ",a,1");
    /* row2 */
    list.add("2," + columnFamily2 + ",a,3");
    list.add("2," + columnFamily2 + ",b,3");
    JavaRDD<String> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.bulkLoad(rdd, TableName.valueOf(tableName),new BulkLoadFunction(), outputPath,
        new HashMap<byte[], FamilyHFileWriteOptions>(), false, HConstants.DEFAULT_MAX_FILE_SIZE);
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkLoadExample文件：

```
def main(args: Array[String]) {
    if(args.length < 2) {
      println("HBaseBulkLoadExample {outputPath} {tableName}
      return
    }
    LoginUtil.loginWithUserKeytab()
    val Array(outputPath, tableName) = args
    val columnFamily1 = "f1"
    val columnFamily2 = "f2"
    val sparkConf = new SparkConf().setAppName("JavaHBaseBulkLoadExample " + tableName)
    val sc = new SparkContext(sparkConf)
    try {
      val arr = Array("1," + columnFamily1 + ",b,1",
                      "2," + columnFamily1 + ",a,2",
                      "3," + columnFamily1 + ",b,1",
                      "3," + columnFamily2 + ",a,1",
                      "4," + columnFamily2 + ",a,3",
                      "5," + columnFamily2 + ",b,3")

        val rdd = sc.parallelize(arr)
        val config = HBaseConfiguration.create
        val hbaseContext = new HBaseContext(sc, config)
        hbaseContext.bulkLoad[String](rdd,
          TableName.valueOf(tableName),
          (putRecord) => {
```

```
            if(putRecord.length > 0) {
                val strArray = putRecord.split(",")
                val kfq = new KeyFamilyQualifier(Bytes.toBytes(strArray(0)), Bytes.toBytes(strArray(1)),
Bytes.toBytes(strArray(2)))
                val ite = (kfq, Bytes.toBytes(strArray(3)))
                val itea = List(ite).iterator
                itea
              } else {
                null
              }
            },
            outputPath)
        } finally {
          sc.stop()
        }
    }
 }
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkLoadPythonExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkLoadExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.HBaseBulkLoadPythonExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.HBaseBulkLoadPythonExample().hbaseBulkLoad(spark._jsc, sys.argv[1], sys.argv[2])
# 停止SparkSession
spark.stop()
```

27.5.4.7 foreachPartition 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，将要插入的数据的rowKey构造成rdd，然后通过HBaseContext的mapPartition接口将rdd并发写入HBase表中。

数据规划

1. 在客户端执行：**hbase shell**命令进入HBase命令行。
2. 使用下面的命令创建HBase表：

   create *'table2','cf1'*

开发思路

1. 将要导入的数据构造成RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的foreachPatition接口将数据并发写入HBase中。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseForEachPartitionExample SparkOnHbaseJavaExample.jar table2 cf1

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseForEachPartitionExample.py table2 cf1

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseForEachPartitionExample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar table2 cf1

python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseForEachPartitionExample.py table2 cf1

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseForEachPartitionExample文件：

```
public static void main(String[] args) throws IOException {
  if (args.length < 1) {
    System.out.println("JavaHBaseForEachPartitionExample {tableName} {columnFamily}");
    return;
  }
  LoginUtil.loginWithUserKeytab();
  final String tableName = args[0];
  final String columnFamily = args[1];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkGetExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<byte[]> list = new ArrayList<byte[]>(5);
    list.add(Bytes.toBytes("1"));
    list.add(Bytes.toBytes("2"));
    list.add(Bytes.toBytes("3"));
    list.add(Bytes.toBytes("4"));
    list.add(Bytes.toBytes("5"));
    JavaRDD<byte[]> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.foreachPartition(rdd,
            new VoidFunction<Tuple2<Iterator<byte[]>, Connection>>() {
       public void call(Tuple2<Iterator<byte[]>, Connection> t)
            throws Exception {
         Connection con = t._2();
         Iterator<byte[]> it = t._1();
         BufferedMutator buf = con.getBufferedMutator(TableName.valueOf(tableName));
         while (it.hasNext()) {
           byte[] b = it.next();
           Put put = new Put(b);
           put.add(Bytes.toBytes(columnFamily), Bytes.toBytes("cid"), b);
           buf.mutate(put);
         }
         buf.flush();
         buf.close();

       }
     });
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseForEachPartitionExample文件：

```
def main(args: Array[String]) {
  if (args.length < 2) {
    println("HBaseForeachPartitionExample {tableName} {columnFamily} are missing an arguments")
    return
  }
  LoginUtil.loginWithUserKeytab()
```

```
  val tableName = args(0)
  val columnFamily = args(1)
  val sparkConf = new SparkConf().setAppName("HBaseForeachPartitionExample " +
    tableName + " " + columnFamily)
  val sc = new SparkContext(sparkConf)
  try {
    //[(Array[Byte], Array[(Array[Byte], Array[Byte], Array[Byte])])]
    val rdd = sc.parallelize(Array(
      (Bytes.toBytes("1"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("1")))),
      (Bytes.toBytes("2"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("2")))),
      (Bytes.toBytes("3"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("3")))),
      (Bytes.toBytes("4"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("4")))),
      (Bytes.toBytes("5"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("5"))))
    ))
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
    rdd.hbaseForeachPartition(hbaseContext,
      (it, connection) => {
        val m = connection.getBufferedMutator(TableName.valueOf(tableName))
        it.foreach(r => {
          val put = new Put(r._1)
          r._2.foreach((putValue) =>
            put.addColumn(putValue._1, putValue._2, putValue._3))
          m.mutate(put)
        })
        m.flush()
        m.close()
      })
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseForEachPartitionExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseForEachPartitionExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm,
'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseForEachPartitionExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseForEachPartitionExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

27.5.4.8 分布式 Scan HBase 表

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，使用hbaseRDD方法以特定的规则扫描HBase表。

数据规划

使用**操作Avro格式数据**章节中创建的hbase数据表。

开发思路

1. 设置scan的规则，例如：setCaching。
2. 使用特定的规则扫描Hbase表。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseDistributedScanExample SparkOnHbaseJavaExample.jar ExampleAvrotable

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseDistributedScanExample.py ExampleAvrotable

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseDistributedScanExample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar ExampleAvrotable

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseDistributedScanExample.py ExampleAvrotable

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseDistributedScanExample文件：

```
public static void main(String[] args) throws IOException{
  if (args.length < 1) {
    System.out.println("JavaHBaseDistributedScan {tableName}");
    return;
  }
  LoginUtil.loginWithUserKeytab();
  String tableName = args[0];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseDistributedScan " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    Scan scan = new Scan();
    scan.setCaching(100);
    JavaRDD<Tuple2<ImmutableBytesWritable, Result>> javaRdd =
          hbaseContext.hbaseRDD(TableName.valueOf(tableName), scan);
    List<String> results = javaRdd.map(new ScanConvertFunction()).collect();
    System.out.println("Result Size: " + results.size());
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseDistributedScanExample文件：

```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseDistributedScanExample {tableName} missing an argument")
    return
  }
  LoginUtil.loginWithUserKeytab()
  val tableName = args(0)
  val sparkConf = new SparkConf().setAppName("HBaseDistributedScanExample " + tableName )
  val sc = new SparkContext(sparkConf)
  try {
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
```



```
    val scan = new Scan()
    scan.setCaching(100)
    val getRdd = hbaseContext.hbaseRDD(TableName.valueOf(tableName), scan)
    getRdd.foreach(v => println(Bytes.toString(v._1.get())))
    println("Length: " + getRdd.map(r => r._1.copyBytes()).collect().length);
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseDistributedScanExample文件：

```
# -*- coding:utf-8 -*-
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseDistributedScan")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm,
'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseDistributedScanExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseDistributedScan().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

27.5.4.9 mapPartitions 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，使用mapPartition接口并行遍历HBase表。

数据规划

使用**foreachPartition接口使用**章节创建的HBase数据表。

开发思路

1. 构造需要遍历的HBase表中rowkey的RDD。
2. 使用mapPartition接口遍历上述rowkey对应的数据信息，并进行简单的操作。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为 、spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseMapPartitionExample SparkOnHbaseJavaExample.jar table2

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseMapPartitionExample.py table2

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseMapPartitionExample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar table2

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseMapPartitionExample.py table2

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseMapPartitionExample文件：

```
public static void main(String args[]) throws IOException {
    if(args.length <1){
        System.out.println("JavaHBaseMapPartitionExample {tableName} is missing an argument");
        return;
```

```
    }
    LoginUtil.loginWithUserKeytab();
    final String tableName = args[0];
    SparkConf sparkConf = new SparkConf().setAppName("HBaseMapPartitionExample " + tableName);
    JavaSparkContext jsc = new JavaSparkContext(sparkConf);
    try{
        List<byte []>  list = new ArrayList();
        list.add(Bytes.toBytes("1"));
        list.add(Bytes.toBytes("2"));
        list.add(Bytes.toBytes("3"));
        list.add(Bytes.toBytes("4"));
        list.add(Bytes.toBytes("5"));


        JavaRDD<byte []> rdd = jsc.parallelize(list);
        Configuration hbaseconf = HBaseConfiguration.create();
        JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, hbaseconf);
        JavaRDD  getrdd = hbaseContext.mapPartitions(rdd, new
FlatMapFunction<Tuple2<Iterator<byte[]>,Connection>, Object>() {
            public Iterator call(Tuple2<Iterator<byte[]>, Connection> t)
                    throws Exception {
                Table table = t._2.getTable(TableName.valueOf(tableName));
                //go through rdd
                List<String> list = new ArrayList<String>();
                while(t._1.hasNext()){
                    byte[] bytes = t._1.next();
                    Result result = table.get(new Get(bytes));
                    Iterator<Cell> it = result.listCells().iterator();
                    StringBuilder sb = new StringBuilder();
                    sb.append(Bytes.toString(result.getRow()) + ":");
                    while(it.hasNext()){
                        Cell cell = it.next();
                        String column = Bytes.toString(cell.getQualifierArray());
                        if(column.equals("counter")){
                            sb.append("(" + column + "," + Bytes.toLong(cell.getValueArray()) + ")");
                        } else {
                            sb.append("(" + column + "," + Bytes.toString(cell.getValueArray()) + ")");
                        }
                    }
                    list.add(sb.toString());
                }
                return list.iterator();
            }
        });
        List<byte[]> resultList = getrdd.collect();
        if(null == resultList || 0 == resultList.size()){
            System.out.println("Nothing matches!");
        }else{
            for(int i =0; i< resultList.size(); i++){
                System.out.println(resultList.get(i));
            }
        }
    } finally {
        jsc.stop();
    }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseMapPartitionExample文件：

```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseMapPartitionExample {tableName} is missing an argument")
    return
  }
  LoginUtil.loginWithUserKeytab()
  val tableName = args(0)
```



```
  val sparkConf = new SparkConf().setAppName("HBaseMapPartitionExample " + tableName)
  val sc = new SparkContext(sparkConf)
  try {
    //[(Array[Byte])]
    val rdd = sc.parallelize(Array(
      Bytes.toBytes("1"),
      Bytes.toBytes("2"),
      Bytes.toBytes("3"),
      Bytes.toBytes("4"),
      Bytes.toBytes("5")))
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
    val b = new StringBuilder
    val getRdd = rdd.hbaseMapPartitions[String](hbaseContext, (it, connection) => {
      val table = connection.getTable(TableName.valueOf(tableName))
      it.map{r =>
        //batching would be faster.  This is just an example
        val result = table.get(new Get(r))
        val it = result.listCells().iterator()
        b.append(Bytes.toString(result.getRow) + ":")
        while (it.hasNext) {
          val cell = it.next()
          val q = Bytes.toString(cell.getQualifierArray)
          if (q.equals("counter")) {
            b.append("(" + q + "," + Bytes.toLong(cell.getValueArray) + ")")
          } else {
            b.append("(" + q + "," + Bytes.toString(cell.getValueArray) + ")")
          }
        }
        b.toString()
      }
    })
    getRdd.collect().foreach(v => println(v))
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseMapPartitionExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseMapPartitionExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseMapPartitionExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseMapPartitionExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```


27.5.4.10 SparkStreaming 批量写入 HBase 表

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，使用streamBulkPut接口将流数据写入HBase表中。

数据规划

1. 在客户端执行**hbase shell**进入HBase命令行。
2. 在hbase命令执行下面的命令创建HBbase表：

   create *'streamingTable','cf1'*
3. 在客户端另外一个session通过linux命令构造一个端口进行接收数据（不同操作系统的机器，命令可能不同，suse尝试使用netcat -lk 9999）：

   nc -lk *9999*

   提交任务命令执行之后，在该命令下输入要提交的数据，通过HBase表进行接收。

   说明

   在构造一个端口进行接收数据时，需要在客户端所在服务器上安装netcat。

开发思路

1. 使用SparkStreaming持续读取特定端口的数据。
2. 将读取到的Dstream通过streamBulkPut接口写入HBase表中。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：super，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例），${ip}请使用实际执行nc -lk 9999的命令的机器ip

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.streaming.JavaHBaseStreamingBulkPutExample SparkOnHbaseJavaExample.jar ${ip} 9999 streamingTable cf1

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample.jar HBaseStreamingBulkPutExample.py ${ip} 9999 streamingTable cf1

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例），${ip}请使用实际执行nc -lk 9999的命令的机器ip

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.streaming.JavaHBaseStreamingBulkPutExample --files /opt/user.keytab,/opt/krb5.conf SparkOnHbaseJavaExample.jar ${ip} 9999 streamingTable cf1

  python版本（文件名等请与实际保持一致，此处仅为示例），假设对应的Java代码打包后包名为**SparkOnHbaseJavaExample.jar**，且放在当前提交目录。

  bin/spark-submit --master yarn --deploy-mode cluster --files /opt/user.keytab,/opt/krb5.conf --jars SparkOnHbaseJavaExample.jar HBaseStreamingBulkPutExample.py ${ip} 9999 streamingTable cf1

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseStreamingBulkPutExample文件：

说明

代码中通过awaitTerminationOrTimeout()方法设置了任务超时时间（单位为毫秒），建议根据期望的任务运行时间调整参数大小。

```
public static void main(String[] args) throws IOException {
  if (args.length < 4) {
    System.out.println("JavaHBaseBulkPutExample  " +
          "{host} {port} {tableName}");
    return;
  }
  LoginUtil.loginWithUserKeytab();
  String host = args[0];
  String port = args[1];
  String tableName = args[2];
  String columnFamily = args[3];
  SparkConf sparkConf =
        new SparkConf().setAppName("JavaHBaseStreamingBulkPutExample " +
              tableName + ":" + port + ":" + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
```



```
    JavaStreamingContext jssc =
            new JavaStreamingContext(jsc, new Duration(1000));
    JavaReceiverInputDStream<String> javaDstream =
            jssc.socketTextStream(host, Integer.parseInt(port));
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.streamBulkPut(javaDstream,
            TableName.valueOf(tableName),
            new PutFunction(columnFamily));
    jssc.start();
    jssc.awaitTerminationOrTimeout(60000);
    jssc.stop(false,true);
  }catch(InterruptedException e){
    e.printStackTrace();
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseStreamingBulkPutExample文件：

说明

代码中通过awaitTerminationOrTimeout()方法设置了任务超时时间（单位为毫秒），建议根据期望的任务运行时间调整参数大小。

```
def main(args: Array[String]): Unit = {
  loginUtil.loginWithUserKeytab()
  val host = args(0)
  val port = args(1)
  val tableName = args(2)
  val columnFamily = args(3)
  val conf = new SparkConf()
  conf.setAppName("HBase Streaming Bulk Put Example")
  val sc = new SparkContext(conf)
  try {
    val config = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, config)
    val ssc = new StreamingContext(sc, Seconds(1))
    val lines = ssc.socketTextStream(host, port.toInt)
    hbaseContext.streamBulkPut[String](lines,
      TableName.valueOf(tableName),
      (putRecord) => {
        if (putRecord.length() > 0) {
          val put = new Put(Bytes.toBytes(putRecord))
          put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes("foo"), Bytes.toBytes("bar"))
          put
        } else {
          null
        }
      })
    ssc.start()
    ssc.awaitTerminationOrTimeout(60000)
    ssc.stop(stopSparkContext = false)
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseStreamingBulkPutExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
(1)由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
(2)如果使用yarn-client模式运行,请确认Spark2x客户端Spark2x/spark/conf/spark-defaults.conf中
   spark.yarn.security.credentials.hbase.enabled参数配置为true
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseStreamingBulkPutExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm,
'com.huawei.bigdata.spark.examples.streaming.JavaHBaseStreamingBulkPutExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseStreamingBulkPutExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

27.5.5 Spark 从 HBase 读取数据再写入 HBase 样例程序

27.5.5.1 Spark 从 HBase 读取数据再写入 HBase 样例程序开发思路

场景说明

假定HBase的table1表存储用户当天消费的金额信息，table2表存储用户历史消费的金额信息。

现table1表有记录key=1,cf:cid=100，表示用户1在当天消费金额为100元。

table2表有记录key=1,cf:cid=1000，表示用户1的历史消息记录金额为1000元。

基于某些业务要求，要求开发Spark应用程序实现如下功能：

根据用户名累计用户的历史消费金额，即用户总消费金额=100(用户当天的消费金额) + 1000(用户历史消费金额)。

上例所示，运行结果table2表用户key=1的总消费金融为cf:cid=1100元。

数据规划

使用Spark-Beeline工具创建Spark和HBase表table1、table2，并通过HBase插入数据。

步骤1 确保JDBCServer已启动。然后在Spark2x客户端，使用Spark-Beeline工具执行如下操作。

步骤2 使用Spark-beeline工具创建Spark表table1。

create table table1

(

key string,

cid string

)

using org.apache.spark.sql.hbase.HBaseSource

options(

hbaseTableName "table1",

keyCols "key",

colsMapping "cid=cf.cid");

步骤3 通过HBase插入数据，命令如下：

put 'table1', '1', 'cf:cid', '100'

步骤4 使用Spark-Beeline工具创建Spark表table2。

create table table2

(

key string,

cid string

)

using org.apache.spark.sql.hbase.HBaseSource

options(

hbaseTableName "table2",

keyCols "key",

colsMapping "cid=cf.cid");

步骤5 通过HBase插入数据，命令如下：

put 'table2', '1', 'cf:cid', '1000'

----结束

开发思路

1. 查询table1表的数据。
2. 根据table1表数据的key值去table2表做查询。
3. 把前两步相应的数据记录做相加操作。
4. 把上一步骤的结果写到table2表。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

1. 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。

2. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

   说明

   - 编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。
   - 运行样例程序前，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”）。

3. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/female/”）下。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行Java或Scala样例代码

  bin/spark-submit --conf spark.yarn.user.classpath.first=true --class *com.huawei.bigdata.spark.examples.SparkHbasetoHbase* **--master yarn --deploy-mode client** */opt/female/SparkHbasetoHbase-1.0.jar*

- 运行**Python**样例程序

  说明

  - 由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。将所提供Java代码使用maven打包成jar，并放在相同目录下，运行python程序时要使用--jars把jar包加载到classpath中。
  - 由于Python样例代码中未给出认证信息，请在执行应用程序时通过配置项“--keytab”和“--principal”指定认证信息。

  bin/spark-submit --master yarn --deploy-mode client --keytab /opt/FIclient/user.keytab --principal sparkuser --conf spark.yarn.user.classpath.first=true --jars */opt/female/SparkHbasetoHbasePythonExample/SparkHbasetoHbase-1.0.jar,/opt/female/protobuf-java-2.5.0.jar /opt/female/SparkHbasetoHbasePythonExample/SparkHbasetoHbasePythonExample.py*

27.5.5.2 Spark 从 HBase 读取数据再写入 HBase 样例程序（Java）

功能介绍

用户可以使用Spark调用HBase接口来操作HBase table1表，然后把table1表的数据经过分析后写到HBase table2表中。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SparkHbasetoHbase。

```
/**
 * 从table1表读取数据，根据key值去table2表获取相应记录，把两者数据后，更新到table2表
*/
```

```
public class SparkHbasetoHbase {

 public static void main(String[] args) throws Exception {
   SparkConf conf = new SparkConf().setAppName("SparkHbasetoHbase");
   conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
   conf.set("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator");
   JavaSparkContext jsc = new JavaSparkContext(conf);
   // 建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
   Configuration hbConf = HBaseConfiguration.create(jsc.hadoopConfiguration());

   // 声明表的信息
   Scan scan = new org.apache.hadoop.hbase.client.Scan();
   scan.addFamily(Bytes.toBytes("cf"));//colomn family
   org.apache.hadoop.hbase.protobuf.generated.ClientProtos.Scan proto = ProtobufUtil.toScan(scan);
   String scanToString = Base64.encodeBytes(proto.toByteArray());
   hbConf.set(TableInputFormat.INPUT_TABLE, "table1");//table name
   hbConf.set(TableInputFormat.SCAN, scanToString);

   // 通过spark接口获取表中的数据
   JavaPairRDD rdd = jsc.newAPIHadoopRDD(hbConf, TableInputFormat.class,
ImmutableBytesWritable.class, Result.class);

   // 遍历hbase table1表中的每一个partition, 然后更新到Hbase table2表
   // 如果数据条数较少，也可以使用rdd.foreach()方法
   rdd.foreachPartition(
     new VoidFunction<Iterator<Tuple2<ImmutableBytesWritable, Result>>>() {
       public void call(Iterator<Tuple2<ImmutableBytesWritable, Result>> iterator) throws Exception {
         hBaseWriter(iterator);
       }
     }
   );

   jsc.stop();
 }

/**
  * 在executor端更新table2表记录
  *
  * @param iterator table1表的partition数据
  */
 private static void hBaseWriter(Iterator<Tuple2<ImmutableBytesWritable, Result>> iterator) throws
IOException {
   // 准备读取hbase
   String tableName = "table2";
   String columnFamily = "cf";
   String qualifier = "cid";
   Configuration conf = HBaseConfiguration.create();
   Connection connection = null;
   Table table = null;
   try {
     connection = ConnectionFactory.createConnection(conf);
     table = connection.getTable(TableName.valueOf(tableName));
     List<Get> rowList = new ArrayList<Get>();
     List<Tuple2<ImmutableBytesWritable, Result>> table1List = new
ArrayList<Tuple2<ImmutableBytesWritable, Result>>();
     while (iterator.hasNext()) {
       Tuple2<ImmutableBytesWritable, Result> item = iterator.next();
       Get get = new Get(item._2().getRow());
       table1List.add(item);
       rowList.add(get);
     }
     // 获取table2表记录
     Result[] resultDataBuffer = table.get(rowList);
     // 修改table2表记录
     List<Put> putList = new ArrayList<Put>();
     for (int i = 0; i < resultDataBuffer.length; i++) {
       Result resultData = resultDataBuffer[i];//hbase2 row
       if (!resultData.isEmpty()) {
         // 查询hbase1Value
```

```
        String hbase1Value = "";
        Iterator<Cell> it = table1List.get(i)._2().listCells().iterator();
        while (it.hasNext()) {
          Cell c = it.next();
          // 判断cf和qualifile是否相同
          if (columnFamily.equals(Bytes.toString(CellUtil.cloneFamily(c)))
            && qualifier.equals(Bytes.toString(CellUtil.cloneQualifier(c)))) {
            hbase1Value = Bytes.toString(CellUtil.cloneValue(c));
          }
        }
        String hbase2Value = Bytes.toString(resultData.getValue(columnFamily.getBytes(),
qualifier.getBytes()));
        Put put = new Put(table1List.get(i)._2().getRow());
        // 计算结果
        int resultValue = Integer.parseInt(hbase1Value) + Integer.parseInt(hbase2Value);
        // 设置结果到put对象
        put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(qualifier),
Bytes.toBytes(String.valueOf(resultValue)));
        putList.add(put);
      }
    }
    if (putList.size() > 0) {
      table.put(putList);
    }
  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    if (table != null) {
      try {
        table.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    if (connection != null) {
      try {
        // 关闭Hbase连接
        connection.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
 }
}
```

27.5.5.3 Spark 从 HBase 读取数据再写入 HBase 样例程序（Scala）

功能介绍

用户可以使用Spark调用HBase接口来操作HBase table1表，然后把table1表的数据经过分析后写到HBase table2表中。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SparkHbasetoHbase。

```
/**
 * 从table1表读取数据，根据key值去table2表获取相应记录，把两者数据后，更新到table2表
 */
object SparkHbasetoHbase {

 case class FemaleInfo(name: String, gender: String, stayTime: Int)

 def main(args: Array[String]) {
```



```
  val conf = new SparkConf().setAppName("SparkHbasetoHbase")
  conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  conf.set("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator")
  val sc = new SparkContext(conf)
  // 建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
  val hbConf = HBaseConfiguration.create(sc.hadoopConfiguration)

  // 声明表的信息
  val scan = new Scan()
  scan.addFamily(Bytes.toBytes("cf"))//colomn family
  val proto = ProtobufUtil.toScan(scan)
  val scanToString = Base64.encodeBytes(proto.toByteArray)
  hbConf.set(TableInputFormat.INPUT_TABLE, "table1")//table name
  hbConf.set(TableInputFormat.SCAN, scanToString)

  // 通过spark接口获取表中的数据
  val rdd = sc.newAPIHadoopRDD(hbConf, classOf[TableInputFormat], classOf[ImmutableBytesWritable],
classOf[Result])

  // 遍历hbase table1表中的每一个partition, 然后更新到Hbase table2表
  // 如果数据条数较少，也可以使用rdd.foreach()方法
  rdd.foreachPartition(x => hBaseWriter(x))

  sc.stop()
 }
 /**
  * 在executor端更新table2表记录
  *
  * @param iterator  table1表的partition数据
  */
 def hBaseWriter(iterator: Iterator[(ImmutableBytesWritable, Result)]): Unit = {
  // 准备读取hbase
  val tableName = "table2"
  val columnFamily = "cf"
  val qualifier = "cid"
  val conf = HBaseConfiguration.create()
  var table: Table = null
  var connection: Connection = null
  try {
    connection = ConnectionFactory.createConnection(conf)
    table = connection.getTable(TableName.valueOf(tableName))
    val iteratorArray = iterator.toArray
    val rowList = new util.ArrayList[Get]()
    for (row <- iteratorArray) {
      val get = new Get(row._2.getRow)
      rowList.add(get)
    }
    // 获取table2表记录
    val resultDataBuffer = table.get(rowList)
    // 修改table2表记录
    val putList = new util.ArrayList[Put]()
    for (i <- 0 until iteratorArray.size) {
      val resultData = resultDataBuffer(i) //hbase2 row
      if (!resultData.isEmpty) {
        // 查询hbase1Value
        var hbase1Value = ""
        val it = iteratorArray(i)._2.listCells().iterator()
        while (it.hasNext) {
          val c = it.next()
          // 判断cf和qualifile是否相同
          if (columnFamily.equals(Bytes.toString(CellUtil.cloneFamily(c)))
            && qualifier.equals(Bytes.toString(CellUtil.cloneQualifier(c)))) {
            hbase1Value = Bytes.toString(CellUtil.cloneValue(c))
          }
        }
        val hbase2Value = Bytes.toString(resultData.getValue(columnFamily.getBytes, qualifier.getBytes))
        val put = new Put(iteratorArray(i)._2.getRow)
        // 计算结果
        val resultValue = hbase1Value.toInt + hbase2Value.toInt
```

```
        // 设置结果到put对象
        put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(qualifier),
Bytes.toBytes(resultValue.toString))
        putList.add(put)
      }
    }
    if (putList.size() > 0) {
      table.put(putList)
    }
  } catch {
    case e: IOException =>
      e.printStackTrace();
  } finally {
    if (table != null) {
      try {
        table.close()
      } catch {
        case e: IOException =>
          e.printStackTrace();
      }
    }
    if (connection != null) {
      try {
        //关闭Hbase连接
        connection.close()
      } catch {
        case e: IOException =>
          e.printStackTrace()
      }
    }
  }
 }
}


/**
 * 序列化辅助类
 */
class MyRegistrator extends KryoRegistrator {
 override def registerClasses(kryo: Kryo) {
   kryo.register(classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable])
   kryo.register(classOf[org.apache.hadoop.hbase.client.Result])
   kryo.register(classOf[Array[(Any, Any)]])
   kryo.register(classOf[Array[org.apache.hadoop.hbase.Cell]])
   kryo.register(classOf[org.apache.hadoop.hbase.NoTagsKeyValue])
   kryo.register(classOf[org.apache.hadoop.hbase.protobuf.generated.ClientProtos.RegionLoadStats])
 }
}
```

27.5.5.4 Spark 从 HBase 读取数据再写入 HBase 样例程序（Python）

功能介绍

用户可以使用Spark调用HBase接口来操作HBase table1表，然后把table1表的数据经过分析后写到HBase table2表中。

代码样例

由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。

下面代码片段仅为演示，具体代码参见SparkHbasetoHbasePythonExample：

```
# -*- coding:utf-8 -*-

from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
```

```
# 创建SparkSession，设置kryo序列化
spark = SparkSession\
        .builder\
        .appName("SparkHbasetoHbase") \
        .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer") \
        .config("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator") \
        .getOrCreate()

# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.SparkHbasetoHbase')

# 创建类实例并调用方法
spark._jvm.SparkHbasetoHbase().hbasetohbase(spark._jsc)

# 停止SparkSession
spark.stop()
```

27.5.6 Spark 从 Hive 读取数据再写入 HBase 样例程序

27.5.6.1 Spark 从 Hive 读取数据再写入 HBase 样例程序开发思路

场景说明

假定Hive的person表存储用户当天消费的金额信息，HBase的table2表存储用户历史消费的金额信息。

现person表有记录name=1,account=100，表示用户1在当天消费金额为100元。

table2表有记录key=1,cf:cid=1000，表示用户1的历史消息记录金额为1000元。

基于某些业务要求，要求开发Spark应用程序实现如下功能：

根据用户名累计用户的历史消费金额，即用户总消费金额=100(用户当天的消费金额) + 1000(用户历史消费金额)。

上例所示，运行结果table2表用户key=1的总消费金融为cf:cid=1100元。

数据规划

在开始开发应用前，需要创建Hive表，命名为person，并插入数据。同时，创建HBase table2表，用于将分析后的数据写入。

步骤1 将原日志文件放置到HDFS系统中。

1. 在本地新建一个空白的log1.txt文件，并在文件内写入如下内容：
   ```
   1,100
   ```
2. 在HDFS中新建一个目录/tmp/input，并将log1.txt文件上传至此目录。
   a. 在HDFS客户端，执行如下命令获取安全认证。

      cd */opt/hadoopclient*

      kinit *<用于认证的业务用户>*

   b. 在Linux系统HDFS客户端使用命令***hadoop fs -mkdir /tmp/input***（hdfs dfs命令有同样的作用），创建对应目录。

   c. 在Linux系统HDFS客户端使用命令***hadoop fs -put*** *log1.txt /tmp/input*，上传数据文件。

步骤2 将导入的数据放置在Hive表里。

首先，确保JDBCServer已启动。然后使用Beeline工具，创建Hive表，并插入数据。

1. 执行如下命令，创建命名为person的Hive表。

   create table person
   (
   name STRING,
   account INT
)
   ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ESCAPED BY '||'
   STORED AS TEXTFILE;

2. 执行如下命令插入数据。

   load data inpath '/tmp/input/log1.txt' into table person;

步骤3 创建HBase表。

确保JDBCServer已启动，然后使用Spark-beeline工具，创建HBase表，并插入数据。

1. 执行如下命令，创建命名为table2的HBase表。

   create table table2
   (
   key string,
   cid string
)
   using org.apache.spark.sql.hbase.HBaseSource
   options(
   hbaseTableName "table2",
   keyCols "key",
   colsMapping "cid=cf.cid"
   );

2. 通过HBase插入数据，执行如下命令。

   put 'table2', '1', 'cf:cid', '1000'

----结束

开发思路

1. 查询Hive person表的数据。
2. 根据person表数据的key值去table2表做查询。
3. 把前两步相应的数据记录做相加操作。
4. 把上一步骤的结果写到table2表。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

1. 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
2. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

   说明

   - 编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。
   - 运行样例程序前，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”）。
3. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“ /opt/female/” ）下。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Java**或**Scala**样例代码

  bin/spark-submit --class *com.huawei.bigdata.spark.examples.SparkHivetoHbase* **--master yarn --deploy-mode client** */opt/female/SparkHivetoHbase-1.0.jar*

- 运行**Python**样例程序

  说明

  - 由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。将所提供Java代码使用maven打包成jar，并放在相同目录下，运行python程序时要使用--jars把jar包加载到classpath中。
  - 由于Python样例代码中未给出认证信息，请在执行应用程序时通过配置项“--keytab”和“--principal”指定认证信息。

  bin/spark-submit --master yarn --deploy-mode client --keytab /opt/FIclient/user.keytab --principal sparkuser --jars */opt/female/SparkHivetoHbasePythonExample/SparkHivetoHbase-1.0.jar /opt/female/SparkHivetoHbasePythonExample/SparkHivetoHbasePythonExample.py*

27.5.6.2 Spark 从 Hive 读取数据再写入 HBase 样例程序（Java）

功能介绍

在Spark应用中，通过使用Spark调用Hive接口来操作hive表，然后把Hive表的数据经过分析后写到HBase表。

代码样例

下面代码片段仅为演示，具体代码参见：
com.huawei.bigdata.spark.examples.SparkHivetoHbase

```
/**
 * 从hive表读取数据，根据key值去hbase表获取相应记录，把两者数据做操作后，更新到hbase表
```

```
*/
public class SparkHivetoHbase {

 public static void main(String[] args) throws Exception {

  String userPrincipal = "sparkuser";
  String userKeytabPath = "/opt/FIclient/user.keytab";
  String krb5ConfPath = "/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf";
  Configuration hadoopConf = new Configuration();
  LoginUtil.login(userPrincipal, userKeytabPath, krb5ConfPath, hadoopConf);
  // 通过spark接口获取表中的数据
  SparkConf conf = new SparkConf().setAppName("SparkHivetoHbase");
  JavaSparkContext jsc = new JavaSparkContext(conf);
  HiveContext sqlContext = new org.apache.spark.sql.hive.HiveContext(jsc);

  Dataset<Row> dataFrame = sqlContext.sql("select name, account from person");

  // 遍历hive表中的每一个partition, 然后更新到hbase表
  // 如果数据条数较少，也可以使用foreach()方法
  dataFrame.toJavaRDD().foreachPartition(
    new VoidFunction<Iterator<Row>>() {
      public void call(Iterator<Row> iterator) throws Exception {
        hBaseWriter(iterator);
      }
    }
  );

  spark.stop();
 }

 /**
  * 在executor端更新hbase表记录
  *
  * @param iterator hive表的partition数据
  */
 private static void hBaseWriter(Iterator<Row> iterator) throws IOException {
  // 读取hbase
  String tableName = "table2";
  String columnFamily = "cf";
  Configuration conf = HBaseConfiguration.create();
  Connection connection = null;
  Table table = null;
  try {
    connection = ConnectionFactory.createConnection(conf);
    table = connection.getTable(TableName.valueOf(tableName));
    List<Row> table1List = new ArrayList<Row>();
    List<Get> rowList = new ArrayList<Get>();
    while (iterator.hasNext()) {
      Row item = iterator.next();
      Get get = new Get(item.getString(0).getBytes());
      table1List.add(item);
      rowList.add(get);
    }
    // 获取hbase表记录
    Result[] resultDataBuffer = table.get(rowList);
    // 修改hbase表记录
    List<Put> putList = new ArrayList<Put>();
    for (int i = 0; i < resultDataBuffer.length; i++) {
      // hive表值
      Result resultData = resultDataBuffer[i];
      if (!resultData.isEmpty()) {
        // get hiveValue
        int hiveValue = table1List.get(i).getInt(1);
        // 根据列簇和列，获取hbase值
        String hbaseValue = Bytes.toString(resultData.getValue(columnFamily.getBytes(), "cid".getBytes()));
        Put put = new Put(table1List.get(i).getString(0).getBytes());
        // 计算结果
        int resultValue = hiveValue + Integer.valueOf(hbaseValue);
        // 设置结果到put对象
```



```
        put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes("cid"),
Bytes.toBytes(String.valueOf(resultValue)));
        putList.add(put);
      }
    }
    if (putList.size() > 0) {
      table.put(putList);
    }
  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    if (table != null) {
      try {
        table.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    if (connection != null) {
      try {
        // 关闭Hbase连接.
        connection.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
 }
}
```

27.5.6.3 Spark 从 Hive 读取数据再写入 HBase 样例程序（Scala）

功能介绍

在Spark应用中，通过使用Spark调用Hive接口来操作hive表，然后把Hive表的数据经过分析后写到HBase表。

代码样例

下面代码片段仅为演示，具体代码参见：
com.huawei.bigdata.spark.examples.SparkHivetoHbase

```
/**
 * 从hive表读取数据，根据key值去hbase表获取相应记录，把两者数据做操作后，更新到hbase表
 */
object SparkHivetoHbase {
 case class FemaleInfo(name: String, gender: String, stayTime: Int)
 def main(args: Array[String]) {

   String userPrincipal = "sparkuser";
   String userKeytabPath = "/opt/FIclient/user.keytab";
   String krb5ConfPath = "/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf";
   Configuration hadoopConf = new Configuration();
   LoginUtil.login(userPrincipal, userKeytabPath, krb5ConfPath, hadoopConf);
   // 通过spark接口获取表中的数据

    val spark = SparkSession
     .builder()
     .appName("SparkHiveHbase")
     .config("spark.sql.warehouse.dir", "spaek-warehouse")
     .enableHiveSupport()
     .getOrCreate()

   import spark.implicits._
   val dataFrame = spark.sql("select name, account from person")
   // 遍历hive表中的每一个partition, 然后更新到hbase表
```



```
  // 如果数据条数较少，也可以使用foreach()方法
  dataFrame.rdd.foreachPartition(x => hBaseWriter(x))
  spark.stop()
 }
 /**
  * 在executor端更新hbase表记录
  *
  * @param iterator hive表的partition数据
  */
 def hBaseWriter(iterator: Iterator[Row]): Unit = {
  // 读取hbase
  val tableName = "table2"
  val columnFamily = "cf"
  val conf = HBaseConfiguration.create()
  var table: Table = null
  var connection: Connection = null
  try {
   connection = ConnectionFactory.createConnection(conf)
   table = connection.getTable(TableName.valueOf(tableName))
   val iteratorArray = iterator.toArray
   val rowList = new util.ArrayList[Get]()
   for (row <- iteratorArray) {
    val get = new Get(row.getString(0).getBytes)
    rowList.add(get)
   }
   // 获取hbase表记录
   val resultDataBuffer = table.get(rowList)
   // 修改hbase表记录
   val putList = new util.ArrayList[Put]()
   for (i <- 0 until iteratorArray.size) {
    // hbase row
    val resultData = resultDataBuffer(i)
    if (!resultData.isEmpty) {
     // hive表值
     var hiveValue = iteratorArray(i).getInt(1)
     // 根据列簇和列，获取hbase值
     val hbaseValue = Bytes.toString(resultData.getValue(columnFamily.getBytes, "cid".getBytes))
     val put = new Put(iteratorArray(i).getString(0).getBytes)
     // 计算结果
     val resultValue = hiveValue + hbaseValue.toInt
     // 设置结果到put对象
     put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes("cid"),
Bytes.toBytes(resultValue.toString))
     putList.add(put)
    }
   }
   if (putList.size() > 0) {
    table.put(putList)
   }
  } catch {
   case e: IOException =>
    e.printStackTrace();
  } finally {
   if (table != null) {
    try {
     table.close()
    } catch {
     case e: IOException =>
      e.printStackTrace();
    }
   }
   if (connection != null) {
    try {
     //关闭Hbase连接.
     connection.close()
    } catch {
     case e: IOException =>
      e.printStackTrace()
    }
```

```
    }
  }
 }

}
```

27.5.6.4 Spark 从 Hive 读取数据再写入 HBase 样例程序（Python）

功能介绍

在Spark应用中，通过使用Spark调用Hive接口来操作hive表，然后把Hive表的数据经过分析后写到HBase表。

代码样例

由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。

下面代码片段仅为演示，具体代码参见SparkHivetoHbasePythonExample:

```
# -*- coding:utf-8 -*-

from py4j.java_gateway import java_import
from pyspark.sql import SparkSession

# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("SparkHivetoHbase") \
        .getOrCreate()

# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.SparkHivetoHbase')

# 创建类实例并调用方法
spark._jvm.SparkHivetoHbase().hivetohbase(spark._jsc)

# 停止SparkSession
spark.stop()
```

27.5.7 Spark Streaming 对接 Kafka0-10 样例程序

27.5.7.1 Spark Streaming 对接 Kafka0-10 样例程序开发思路

场景说明

假定某个业务Kafka每1秒就会收到1个单词记录。

基于某些业务要求，开发的Spark应用程序实现如下功能：

实时累加计算每个单词的记录总数。

“log1.txt”示例文件：

```
LiuYang
YuanJing
GuoYijun
CaiXuyu
Liyuan
FangBo
LiuYang
YuanJing
```

```
GuoYijun
CaiXuyu
FangBo
```

数据规划

Spark Streaming样例工程的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户）。

1. 确保集群安装完成，包括HDFS、Yarn、Spark和Kafka。
2. 本地新建文件“input_data1.txt”，将“log1.txt”的内容复制保存到“input_data1.txt”。

   在客户端安装节点下创建文件目录：“/home/data”。将上述文件上传到此“/home/data”目录下。
3. 将Kafka的Broker配置参数“allow.everyone.if.no.acl.found”的值修改为“true”。
4. 创建Topic。

   {zkQuorum}表示ZooKeeper集群信息，格式为IP:port。

   $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper {zkQuorum}/kafka --replication-factor 1 --partitions 3 --topic {Topic}
5. 启动Kafka的Producer，向Kafka发送数据。

   java -cp {ClassPath} com.huawei.bigdata.spark.examples.StreamingExampleProducer {BrokerList} {Topic}

   其中，ClassPath除样例jar包路径外，还应包含Spark客户端Kafka jar包的绝对路径，例如：/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/jars/streamingClient010/*:{ClassPath}

开发思路

1. 接收Kafka中数据，生成相应DStream。
2. 对单词记录进行分类统计。
3. 计算结果，并进行打印。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

  说明

  编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。

- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“ /opt” ）下。

运行任务

在运行样例程序时需要指定<checkpointDir> <brokers> <topic> <batchTime>，其中<checkPointDir>指应用程序结果备份到HDFS的路径，<brokers>指获取元数据的Kafka地址，<topic>指读取Kafka上的topic名称，<batchTime>指Streaming分批的处理间隔。

说明

由于Spark Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Streaming Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

由于运行模式为安全模式，需要添加新配置并修改命令参数：

1. $SPARK_HOME/conf/jaas.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=false
   useTicketCache=true
   debug=false;
   };
   ```
2. $SPARK_HOME/conf/jaas-zk.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   keyTab="./user.keytab"
   principal="sparkuser@<系统域名>"
   useTicketCache=false
   storeKey=true
   debug=true;
   };
   ```
3. 使用--files和相对路径提交keytab文件，这样才能保证keytab文件被加载到executor的container中。
4. <brokers>中的端口，Kafka 0-10 Write To Print样例请使用SASL_PLAINTEXT协议端口号，Write To Kafka 0-10样例请使用PLAINTEXT协议端口号。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- **Spark Streaming读取Kafka 0-10 Write To Print代码样例**

  bin/spark-submit --master yarn --deploy-mode client --files ./jaas.conf,./user.keytab **--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")** --class com.huawei.bigdata.spark.examples.SecurityKafkaWordCount /opt/SparkStreamingKafka010JavaExample-1.0.jar <checkpointDir> <brokers> <topic> <batchTime>

  其中配置示例如下：

  ```
  --files ./jaas.conf,./user.keytab //使用--files指定jaas.conf和keytab文件。
  ```

- **Spark Streaming Write To Kafka 0-10代码样例：**

  bin/spark-submit --master yarn --deploy-mode client --jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}") --class

com.huawei.bigdata.spark.examples.JavaDstreamKafkaWriter /opt/SparkStreamingKafka010JavaExample-1.0.jar <groupId> <brokers> <topics>

27.5.7.2 Spark Streaming 对接 Kafka0-10 样例程序（Java）

功能介绍

在Spark应用中，通过使用Streaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数，或将数据写入Kafka0-10。

Streaming 读取 Kafka0-10 代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SecurityKafkaWordCount。

```
/**
 *从Kafka的一个或多个主题消息。
 * <checkPointDir>是Spark Streaming检查点目录。
 * <brokers>是用于自举，制作人只会使用它来获取元数据
 * <topics>是要消费的一个或多个kafka主题的列表
 * <batchTime>是Spark Streaming批次持续时间（以秒为单位）。
 */
public class SecurityKafkaWordCount
{
  public static void main(String[] args) throws Exception {
    JavaStreamingContext ssc = createContext(args);

    //启动Streaming系统。
    ssc.start();
    try {
      ssc.awaitTermination();
    } catch (InterruptedException e) {
    }
  }

  private static JavaStreamingContext createContext(String[] args) throws Exception {
    String checkPointDir = args[0];
    String brokers = args[1];
    String topics = args[2];
    String batchTime = args[3];

    //新建一个Streaming启动环境.
    SparkConf sparkConf = new SparkConf().setAppName("KafkaWordCount");
    JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, new
Duration(Long.parseLong(batchTime) * 1000));

    //配置Streaming的CheckPoint目录。
    //由于窗口概念的存在，此参数是必需的。
    ssc.checkpoint(checkPointDir);

    //获取kafka使用的topic列表。
    String[] topicArr = topics.split(",");
    Set<String> topicSet = new HashSet<String>(Arrays.asList(topicArr));
    Map<String, Object> kafkaParams = new HashMap();
    kafkaParams.put("bootstrap.servers", brokers);
    kafkaParams.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    kafkaParams.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    kafkaParams.put("group.id", "DemoConsumer");
    kafkaParams.put("security.protocol", "SASL_PLAINTEXT");
    kafkaParams.put("sasl.kerberos.service.name", "kafka");
    kafkaParams.put("kerberos.domain.name", "hadoop.<系统域名>");

    LocationStrategy locationStrategy = LocationStrategies.PreferConsistent();
    ConsumerStrategy consumerStrategy = ConsumerStrategies.Subscribe(topicSet, kafkaParams);
```



```
    //用brokers and topics新建direct kafka stream
    //从Kafka接收数据并生成相应的DStream。
    JavaInputDStream<ConsumerRecord<String, String>> messages = KafkaUtils.createDirectStream(ssc,
locationStrategy, consumerStrategy);

    //获取每行中的字段属性。
    JavaDStream<String> lines = messages.map(new Function<ConsumerRecord<String, String>, String>() {
      @Override
      public String call(ConsumerRecord<String, String> tuple2) throws Exception {
        return tuple2.value();
      }
    });

    //汇总计算字数的总时间。
    JavaPairDStream<String, Integer> wordCounts = lines.mapToPair(
        new PairFunction<String, String, Integer>() {
          @Override
          public Tuple2<String, Integer> call(String s) {
            return new Tuple2<String, Integer>(s, 1);
          }
        }).reduceByKey(new Function2<Integer, Integer, Integer>() {
      @Override
      public Integer call(Integer i1, Integer i2) {
        return i1 + i2;
      }
    }).updateStateByKey(
        new Function2<List<Integer>, Optional<Integer>, Optional<Integer>>() {
          @Override
          public Optional<Integer> call(List<Integer> values, Optional<Integer> state) {
            int out = 0;
            if (state.isPresent()) {
              out += state.get();
            }
            for (Integer v : values) {
              out += v;
            }
            return Optional.of(out);
          }
        });

    //打印结果
    wordCounts.print();
    return ssc;
  }
}
```

Streaming Write To Kafka 0-10 样例代码

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.DstreamKafkaWriter。

说明

建议使用新的API **createDirectStream**代替旧的API **createStream**进行应用程序开发。旧的API仍然可以使用，但新的API性能和稳定性更好。

```
/**
 * 参数解析:
 * <groupId>为客户的组编号。
 * <brokers>为获取元数据的Kafka地址。
 * <topic>为Kafka中订阅的主题。
 */
public class JavaDstreamKafkaWriter {

  public static void main(String[] args) throws InterruptedException {
    if (args.length != 3) {
      System.err.println("Usage: JavaDstreamKafkaWriter <groupId> <brokers> <topic>");
      System.exit(1);
```



```
  }

  final String groupId = args[0];
  final String brokers = args[1];
  final String topic = args[2];

  SparkConf sparkConf = new SparkConf().setAppName("KafkaWriter");

  // 填写Kafka的properties。
  Map<String, Object> kafkaParams = new HashMap<String, Object>();
  kafkaParams.put("value.deserializer","org.apache.kafka.common.serialization.StringDeserializer");
  kafkaParams.put("key.deserializer" , "org.apache.kafka.common.serialization.StringDeserializer");
  kafkaParams.put("value.serializer" , "org.apache.kafka.common.serialization.ByteArraySerializer");
  kafkaParams.put("key.serializer" , "org.apache.kafka.common.serialization.StringSerializer");
  kafkaParams.put("bootstrap.servers", brokers);
  kafkaParams.put("group.id", groupId);
  kafkaParams.put("auto.offset.reset", "smallest");

  // 创建Java Spark Streaming的Context。
  JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.milliseconds(500));

  // 填写写入Kafka的数据。
  List<String> sentData = new ArrayList();
  sentData.add("kafka_writer_test_msg_01");
  sentData.add("kafka_writer_test_msg_02");
  sentData.add("kafka_writer_test_msg_03");

  // 创建Java RDD队列。
  Queue<JavaRDD<String>> sent = new LinkedList();
  sent.add(ssc.sparkContext().parallelize(sentData));

  // 创建写数据的Java DStream。
  JavaDStream wStream = ssc.queueStream(sent);

  // 写入Kafka。
  JavaDStreamKafkaWriterFactory.fromJavaDStream(wStream).writeToKafka(
      JavaConverters.mapAsScalaMapConverter(kafkaParams).asScala(),
      new Function<String, ProducerRecord<String, byte[]>>() {
        public ProducerRecord<String, byte[]> call(String s) throws Exception {
          return new ProducerRecord(topic, s.toString().getBytes());
        }
      });

  ssc.start();
  ssc.awaitTermination();
 }
}
```

27.5.7.3 Spark Streaming 对接 Kafka0-10 样例程序（Scala）

功能介绍

在Spark应用中，通过使用Streaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数，或将数据写入Kafka0-10。

Streaming 读取 Kafka0-10 代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SecurityKafkaWordCount。

```
/**
 *从Kafka的一个或多个主题消息。
 * <checkPointDir>是Spark Streaming检查点目录。
 * <brokers>是用于自举，制作人只会使用它来获取元数据
 * <topics>是要消费的一个或多个kafka主题的列表
 * <batchTime>是Spark Streaming批次持续时间（以秒为单位）。
```

```
 */
object SecurityKafkaWordCount {

 def main(args: Array[String]) {
   val ssc = createContext(args)

   //启动Streaming系统。
   ssc.start()
   ssc.awaitTermination()
 }

 def createContext(args : Array[String]) : StreamingContext = {









   val Array(checkPointDir, brokers, topics, batchSize) = args

   //新建一个Streaming启动环境。
   val sparkConf = new SparkConf().setAppName("KafkaWordCount")
   val ssc = new StreamingContext(sparkConf, Seconds(batchSize.toLong))

   //配置Streaming的CheckPoint目录。
   //由于窗口概念的存在，此参数是必需的。
   ssc.checkpoint(checkPointDir)

   //获取kafka使用的topic列表。
   val topicArr = topics.split(",")
   val topicSet = topicArr.toSet
   val kafkaParams = Map[String, String](
     "bootstrap.servers" -> brokers,
     "value.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
     "key.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
     "group.id" -> "DemoConsumer",
     "security.protocol" -> "SASL_PLAINTEXT",
     "sasl.kerberos.service.name" -> "kafka",
     "kerberos.domain.name" -> "hadoop.<系统域名>"
   );

   val locationStrategy = LocationStrategies.PreferConsistent
   val consumerStrategy = ConsumerStrategies.Subscribe[String, String](topicSet, kafkaParams)

   // 用brokers and topics新建direct kafka stream
   //从Kafka接收数据并生成相应的DStream。
   val stream = KafkaUtils.createDirectStream[String, String](ssc, locationStrategy, consumerStrategy)

   //获取每行中的字段属性。
   val tf = stream.transform ( rdd =>
     rdd.map(r => (r.value, 1L))
   )

   //汇总计算字数的总时间。
   val wordCounts = tf.reduceByKey(_ + _)
   val totalCounts = wordCounts.updateStateByKey(updataFunc)
   totalCounts.print()
   ssc
 }

 def updataFunc(values : Seq[Long], state : Option[Long]) : Option[Long] =
   Some(values.sum + state.getOrElse(0L))
}
```

Streaming Write To Kafka 0-10 样例代码

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.DstreamKafkaWriter。

说明

建议使用新的API **createDirectStream**代替原有API **createStream**进行应用程序开发。原有API仍然可以使用，但新的API性能和稳定性更好。

```
/**
 * 参数解析:
 * <checkPointDir>为checkPoint目录。
 * <topics>为Kafka中订阅的主题，多以逗号分隔。
 * <brokers>为获取元数据的Kafka地址。
 */
object DstreamKafkaWriterTest1 {

  def main(args: Array[String]) {
    if (args.length != 4) {
      System.err.println("Usage: DstreamKafkaWriterTest <checkPointDir> <brokers> <topic>")
      System.exit(1)
    }

    val Array(checkPointDir, brokers, topic) = args
    val sparkConf = new SparkConf().setAppName("KafkaWriter")

    //填写Kafka的properties。
    val kafkaParams = Map[String, String](
      "bootstrap.servers" -> brokers,
      "value.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
      "key.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
      "value.serializer" -> "org.apache.kafka.common.serialization.ByteArraySerializer",
      "key.serializer" -> "org.apache.kafka.common.serialization.StringSerializer",
      "group.id" -> "dstreamKafkaWriterFt",
      "auto.offset.reset" -> "latest"
    )

    //创建Streaming的context。
    val ssc = new StreamingContext(sparkConf, Milliseconds(500));
    val sentData = Seq("kafka_writer_test_msg_01", "kafka_writer_test_msg_02", "kafka_writer_test_msg_03")

    //创建RDD队列。
    val sent = new mutable.Queue[RDD[String]]()
    sent.enqueue(ssc.sparkContext.makeRDD(sentData))

    //创建写数据的DStream。
    val wStream = ssc.queueStream(sent)

    //使用writetokafka API把数据写入Kafka。
    wStream.writeToKafka(kafkaParams,
      (x: String) => new ProducerRecord[String, Array[Byte]](topic, x.getBytes))

    //启动streaming的context。
    ssc.start()
    ssc.awaitTermination()
  }
}
```

27.5.8 Spark Structured Streaming 样例程序

27.5.8.1 Spark Structured Streaming 样例程序开发思路

场景说明

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

数据规划

StructuredStreaming样例工程的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户）。

1. 确保集群安装完成，包括HDFS、Yarn、Spark和Kafka。
2. 将Kafka的Broker配置参数“allow.everyone.if.no.acl.found”的值修改为“true”。
3. 创建Topic。

   {zkQuorum}表示ZooKeeper集群信息，格式为IP:port。

   $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper {zkQuorum}/kafka --replication-factor 1 --partitions 1 --topic {Topic}
4. 启动Kafka的Producer，向Kafka发送数据。

   {ClassPath}表示工程jar包的存放路径，详细路径由用户指定，可参考**在Linux环境中编包并运行Spark程序**章节中导出jar包的操作步骤。

   java -cp $SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:{ClassPath} com.huawei.bigdata.spark.examples.KafkaWordCountProducer {BrokerList} {Topic} {messagesPerSec} {wordsPerMessage}

开发思路

1. 接收Kafka中数据，生成相应DataStreamReader。
2. 对单词记录进行分类统计。
3. 计算结果，并进行打印。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

  说明

  编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt”）下。

运行任务

在运行样例程序时需要指定<brokers> <subscribe-type> <topic> <protocol> <service> <domain><checkpointDir>。

- <brokers>指获取元数据的Kafka地址。
- <subscribe-type>指Kafka订阅类型（如subscribe）。
- <topic>指读取Kafka上的topic名称。
- <protocol>指安全访问协议（如SASL_PLAINTEXT）。
- <service>指kerberos服务名称（如kafka）。
- <domain>指kerberos域名（如hadoop.*<系统域名>*）。
- <checkpointDir>指checkpoint文件存放路径，本地或者HDFS路径下。

说明

由于Spark Structured Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Structured Streaming Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

由于运行模式为安全模式，需要添加新配置并修改命令参数：

1. $SPARK_HOME/conf/jaas.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=false
   useTicketCache=true
   debug=false;
   };
   ```
2. $SPARK_HOME/conf/jaas-zk.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   keyTab="./user.keytab"
   principal="sparkuser@<系统域名>"
   useTicketCache=false
   storeKey=true
   debug=true;
   };
   ```
3. 使用--files和相对路径提交keytab文件，这样才能保证keytab文件被加载到executor的container中。

注意

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，当前版本用户除了这一步外还需要将**$SPARK_HOME/jars/streamingClient010**目录中的kafka-clients jar包复制到**$SPARK_HOME/jars**目录下，否则会报class not found异常。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Java**或**Scala**样例代码：

  bin/spark-submit --master yarn --deploy-mode client --files *<local Path>/jaas.conf,<local path>/user.keytab* **--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}") --class** *com.huawei.bigdata.spark.examples.SecurityKafkaWordCount /opt/SparkStructuredStreamingScalaExample-1.0.jar <brokers> <subscribe-type> <topic> <protocol> <service> <domain> <checkpointDir>*

  其中配置示例如下：


```
--files <local Path>/jaas.conf,<local Path>/user.keytab //使用--files指定jaas.conf和keytab文件。
```

- 运行**Python**样例代码：

  说明

  运行**Python**样例代码时需要将打包后的Java项目的jar包添加到streamingClient010/目录下。

  bin/spark-submit --master yarn --deploy-mode client --files /opt/FIclient/user.keytab --jars *$(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")* /opt/female/SparkStructuredStreamingPythonExample/SecurityKafkaWordCount.py <brokers> <subscribe-type> <topic> <protocol> <service> <domain> <checkpointDir>

27.5.8.2 Spark Structured Streaming 样例程序（Java）

功能介绍

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SecurityKafkaWordCount。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。

```
public class SecurityKafkaWordCount
{
  public static void main(String[] args) throws Exception {
    if (args.length < 6) {
      System.err.println("Usage: SecurityKafkaWordCount <bootstrap-servers> " +
        "<subscribe-type> <topics> <protocol> <service> <domain>");
      System.exit(1);
    }

    String bootstrapServers = args[0];
    String subscribeType = args[1];
    String topics = args[2];
    String protocol = args[3];
    String service = args[4];
    String domain = args[5];

    SparkSession spark = SparkSession
      .builder()
      .appName("SecurityKafkaWordCount")
      .getOrCreate();


    //创建表示来自kafka的输入行流的DataSet。
    Dataset<String> lines = spark
      .readStream()
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option(subscribeType, topics)
      .option("kafka.security.protocol", protocol)
      .option("kafka.sasl.kerberos.service.name", service)
      .option("kafka.kerberos.domain.name", domain)
```



```
    .load()
    .selectExpr("CAST(value AS STRING)")
    .as(Encoders.STRING());

  //生成运行字数。
  Dataset<Row> wordCounts = lines.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterator<String> call(String x) {
      return Arrays.asList(x.split(" ")).iterator();
    }
  }, Encoders.STRING()).groupBy("value").count();

  //开始运行将运行计数打印到控制台的查询。
  StreamingQuery query = wordCounts.writeStream()
    .outputMode("complete")
    .format("console")
    .start();

  query.awaitTermination();
 }
}
```

27.5.8.3 Spark Structured Streaming 样例程序（Scala）

功能介绍

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SecurityKafkaWordCount。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。

```
object SecurityKafkaWordCount {
 def main(args: Array[String]): Unit = {
  if (args.length < 6) {
    System.err.println("Usage: SecurityKafkaWordCount <bootstrap-servers> " +
      "<subscribe-type> <topics> <protocol> <service> <domain>")
    System.exit(1)
  }

  val Array(bootstrapServers, subscribeType, topics, protocol, service, domain) = args

  val spark = SparkSession
    .builder
    .appName("SecurityKafkaWordCount")
    .getOrCreate()

  import spark.implicits._

  //创建表示来自kafka的输入行流的DataSet。
  val lines = spark
    .readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", bootstrapServers)
    .option(subscribeType, topics)
    .option("kafka.security.protocol", protocol)
    .option("kafka.sasl.kerberos.service.name", service)
    .option("kafka.kerberos.domain.name", domain)
```



```
    .load()
    .selectExpr("CAST(value AS STRING)")
    .as[String]

  //生成运行字数。
  val wordCounts = lines.flatMap(_.split(" ")).groupBy("value").count()

  //开始运行将运行计数打印到控制台的查询。
  val query = wordCounts.writeStream
    .outputMode("complete")
    .format("console")
    .start()

  query.awaitTermination()
 }
}
```

27.5.8.4 Spark Structured Streaming 样例程序（Python）

功能介绍

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

代码样例

下面代码片段仅为演示，具体代码参见：SecurityKafkaWordCount。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

if __name__ == "__main__":
    if len(sys.argv) < 6:
        print("Usage: <bootstrapServers> <subscribeType> <topics> <protocol> <service> <domain>")
        exit(-1)

    bootstrapServers = sys.argv[1]
    subscribeType = sys.argv[2]
    topics = sys.argv[3]
    protocol = sys.argv[4]
    service = sys.argv[5]
    domain = sys.argv[6]

    # 初始化sparkSession
    spark = SparkSession.builder.appName("SecurityKafkaWordCount").getOrCreate()

    # 创建表示来自kafka的input lines stream的DataFrame
    # 安全模式要修改spark/conf/jaas.conf和jaas-zk.conf为KafkaClient
    lines = spark.readStream.format("kafka")\
    .option("kafka.bootstrap.servers", bootstrapServers)\
    .option(subscribeType, topics)\
    .option("kafka.security.protocol", protocol)\
    .option("kafka.sasl.kerberos.service.name", service)\
    .option("kafka.kerberos.domain.name", domain)\
    .load()\
    .selectExpr("CAST(value AS STRING)")
```



```
# 将lines切分为word
words = lines.select(explode(split(lines.value, " ")).alias("word"))
# 生成正在运行的word count
wordCounts = words.groupBy("word").count()

# 开始运行将running counts打印到控制台的查询
query = wordCounts.writeStream\
.outputMode("complete")\
.format("console")\
.start()

query.awaitTermination()
```

27.5.9 Spark Structured Streaming 对接 Kafka 样例程序

27.5.9.1 Spark Structured Streaming 对接 Kafka 样例程序开发思路

场景说明

假定一个广告业务，存在广告请求事件、广告展示事件、广告点击事件，广告主需要实时统计有效的广告展示和广告点击数据。

已知：

1. 终端用户每次请求一个广告后，会生成广告请求事件，保存到kafka的adRequest topic中。
2. 请求一个广告后，可能用于多次展示，每次展示，会生成广告展示事件，保存到kafka的adShow topic中。
3. 每个广告展示，可能会产生多次点击，每次点击，会生成广告点击事件，保存到kafka的adClick topic中。
4. 广告有效展示的定义如下：
   a. 请求到展示的时长超过A分钟算无效展示。
   b. A分钟内多次展示，每次展示事件为有效展示。
5. 广告有效点击的定义如下：
   a. 展示到点击时长超过B分钟算无效点击。
   b. B分钟内多次点击，仅首次点击事件为有效点击。

基于此业务场景，模拟简单的数据结构如下：

- 广告请求事件
  数据结构：adID^reqTime
- 广告展示事件
  数据结构：adID^showID^showTime
- 广告点击事件
  数据结构：adID^showID^clickTime

数据关联关系如下：

- 广告请求事件与广告展示事件通过adID关联。
- 广告展示事件与广告点击事件通过adID+showID关联。

数据要求：

- 数据从产生到到达流处理引擎的延迟时间不超过2小时
- 广告请求事件、广告展示事件、广告点击事件到达流处理引擎的时间不能保证有序和时间对齐

数据规划

1. 在kafka中生成模拟数据（需要有Kafka权限用户）。

   java -cp $SPARK_HOME/conf:$SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:{ClassPath} com.huawei.bigdata.spark.examples.KafkaADEventProducer {BrokerList} {timeOfProduceReqEvent} {eventTimeBeforeCurrentTime} {reqTopic} {reqEventCount} {showTopic} {showEventMaxDelay} {clickTopic} {clickEventMaxDelay}

   说明

   - 确保集群安装完成，包括HDFS、Yarn、Spark2x和Kafka。
   - 将Kafka的Broker配置参数“allow.everyone.if.no.acl.found”的值修改为“true”。
   - 启动Kafka的Producer，向Kafka发送数据。
   - {ClassPath}表示工程jar包的存放路径，详细路径由用户指定，可参考**在Linux环境中编包并运行Spark程序**章节中导出jar包的操作步骤。

   命令举例：

   java -cp /opt/client/Spark2x/spark/conf:/opt/StructuredStreamingADScalaExample-1.0.jar:/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/jars/streamingClient010/* com.huawei.bigdata.spark.examples.KafkaADEventProducer 10.132.190.170:21005,10.132.190.165:21005 2h 1h req 10000000 show 5m click 5m

   此命令将在kafka上创建3个topic：req、show、click，在2h内生成1千万条请求事件数据，请求事件的时间取值范围为{当前时间-1h 至 当前时间}，并为每条请求事件随机生成0-5条展示事件，展示事件的时间取值范围为{请求事件时间 至请求事件时间+5m }，为每条展示事件随机生成0-5条点击事件，点击事件的时间取值范围为{展示事件时间 至展示事件时间+5m }

开发思路

1. 使用Structured Streaming接收Kafka中数据，生成请求流、展示流、点击流。
2. 对请求流、展示流、点击流的数据进行关联查询。
3. 统计结果写入kafka。
4. 应用中监控流处理任务的状态。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

  说明

  编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/female/user.keytab”，“/opt/female/krb5.conf”。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt”）下。

运行任务

在运行样例程序时需要指定 <kafkaBootstrapServers> <maxEventDelay> <reqTopic> <showTopic> <maxShowDelay> <clickTopic> <maxClickDelay> <triggerInterver> <checkpointDir> <kafkaProtocol> <kafkaService> <kafkaDomain>。

- <kafkaBootstrapServers>指获取元数据的Kafka地址。
- <maxEventDelay>指数据从生成到被流处理引擎的最大延迟时间。
- <reqTopic>指请求事件的topic名称。
- <showTopic>指展示事件的topic名称。
- <maxShowDelay>指有效展示事件的最大延迟时间。
- <clickTopic>指点击事件的topic名称。
- <maxClickDelay>指有效点击事件的最大延迟时间。
- <triggerInterver>指流处理任务的触发间隔。
- <checkpointDir>指checkpoint文件存放路径，本地或者HDFS路径下。
- <kafkaProtocol>指安全访问协议（如SASL_PLAINTEXT）。
- <kafkaService>指kerberos服务名称（如kafka）。
- <kafkaDomain>指kerberos域名（如hadoop.*<系统域名>*）。

说明

由于Spark Structured Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Structured Streaming Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

由于运行模式为安全模式，需要添加新配置并修改命令参数：

1. $SPARK_HOME/conf/jaas.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=false
   useTicketCache=true
   debug=false;
   };
   ```
2. $SPARK_HOME/conf/jaas-zk.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   keyTab="./user.keytab"
   principal="sparkuser@<系统域名>"
   useTicketCache=false
   storeKey=true
   debug=true;
   };
   ```
3. 使用--files和相对路径提交keytab文件，这样才能保证keytab文件被加载到executor的container中

注意

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，当前版本用户除了这一步外还需要将**$SPARK_HOME/jars/streamingClient010**目录中的kafka-clients jar包复制到**$SPARK_HOME/jars**目录下，否则会报class not found异常。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

bin/spark-submit --master yarn --deploy-mode client --files *<local Path>/jaas.conf,<local path>/user.keytab* **--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}") --conf "spark.sql.streaming.statefulOperator.checkCorrectness.enabled=false" --class** *com.huawei.bigdata.spark.examples.KafkaADCount /opt/StructuredStreamingADScalaExample-1.0.jar <kafkaBootstrapServers> <maxEventDelay> <reqTopic> <showTopic> <maxShowDelay> <clickTopic> <maxClickDelay> <triggerInterver> <checkpointDir> <kafkaProtocol> <kafkaService> <kafkaDomain>*

其中配置示例如下：

```
--files ./jaas.conf,./user.keytab //使用--files指定jaas.conf和keytab文件。
```


27.5.9.2 Spark Structured Streaming 对接 Kafka 样例程序（Scala）

功能介绍

使用Structured Streaming，从kafka中读取广告请求数据、广告展示数据、广告点击数据，实时获取广告有效展示统计数据和广告有效点击统计数据，将统计结果写入kafka中。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.KafkaADCount。

```
/**
 * 运行Structured Streaming任务，统计广告的有效展示和有效点击数据，结果写入kafka中
 */

object KafkaADCount {
  def main(args: Array[String]): Unit = {
    if (args.length < 12) {
      System.err.println("Usage: KafkaWordCount <bootstrap-servers> " +
        "<maxEventDelay> <reqTopic> <showTopic> <maxShowDelay> " +
        "<clickTopic> <maxClickDelay> <triggerInterver> " +
        "<checkpointLocation> <protocol> <service> <domain>")
      System.exit(1)
    }

    val Array(bootstrapServers, maxEventDelay, reqTopic, showTopic,
    maxShowDelay, clickTopic, maxClickDelay, triggerInterver, checkpointLocation,
    protocol, service, domain) = args

    val maxEventDelayMills = JavaUtils.timeStringAs(maxEventDelay, TimeUnit.MILLISECONDS)
    val maxShowDelayMills = JavaUtils.timeStringAs(maxShowDelay, TimeUnit.MILLISECONDS)
    val maxClickDelayMills = JavaUtils.timeStringAs(maxClickDelay, TimeUnit.MILLISECONDS)
    val triggerMills = JavaUtils.timeStringAs(triggerInterver, TimeUnit.MILLISECONDS)

    val spark = SparkSession
      .builder
      .appName("KafkaADCount")
      .getOrCreate()

    spark.conf.set("spark.sql.streaming.checkpointLocation", checkpointLocation)

    import spark.implicits._

    // Create DataSet representing the stream of input lines from kafka
    val reqDf = spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option("kafka.security.protocol", protocol)
      .option("kafka.sasl.kerberos.service.name", service)
      .option("kafka.kerberos.domain.name", domain)
      .option("subscribe", reqTopic)
      .load()
      .selectExpr("CAST(value AS STRING)")
      .as[String]
      .map{
        _.split('^') match {
          case Array(reqAdID, reqTime) => ReqEvent(reqAdID,
            Timestamp.valueOf(reqTime))
        }
      }
      .as[ReqEvent]
      .withWatermark("reqTime", maxEventDelayMills +
        maxShowDelayMills + " millisecond")
```

```
val showDf = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("kafka.security.protocol", protocol)
  .option("kafka.sasl.kerberos.service.name", service)
  .option("kafka.kerberos.domain.name", domain)
  .option("subscribe", showTopic)
  .load()
  .selectExpr("CAST(value AS STRING)")
  .as[String]
  .map{
    _.split('^') match {
      case Array(showAdID, showID, showTime) => ShowEvent(showAdID,
        showID, Timestamp.valueOf(showTime))
    }
  }
  .as[ShowEvent]
  .withWatermark("showTime", maxEventDelayMills +
    maxShowDelayMills + maxClickDelayMills + " millisecond")

val clickDf = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("kafka.security.protocol", protocol)
  .option("kafka.sasl.kerberos.service.name", service)
  .option("kafka.kerberos.domain.name", domain)
  .option("subscribe", clickTopic)
  .load()
  .selectExpr("CAST(value AS STRING)")
  .as[String]
  .map{
    _.split('^') match {
      case Array(clickAdID, clickShowID, clickTime) => ClickEvent(clickAdID,
        clickShowID, Timestamp.valueOf(clickTime))
    }
  }
  .as[ClickEvent]
  .withWatermark("clickTime", maxEventDelayMills + " millisecond")

val showStaticsQuery = reqDf.join(showDf,
  expr(s"""
  reqAdID = showAdID
  AND showTime >= reqTime + interval ${maxShowDelayMills} millisecond
  """))
  .selectExpr("concat_ws('^', showAdID, showID, showTime) as value")
  .writeStream
  .queryName("showEventStatics")
  .outputMode("append")
  .trigger(Trigger.ProcessingTime(triggerMills.millis))
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("kafka.security.protocol", protocol)
  .option("kafka.sasl.kerberos.service.name", service)
  .option("kafka.kerberos.domain.name", domain)
  .option("topic", "showEventStatics")
  .start()

val clickStaticsQuery = showDf.join(clickDf,
  expr(s"""
  showAdID = clickAdID AND
  showID = clickShowID AND
  clickTime >= showTime + interval ${maxClickDelayMills} millisecond
  """), joinType = "rightouter")
  .dropDuplicates("showAdID")
  .selectExpr("concat_ws('^', clickAdID, clickShowID, clickTime) as value")
  .writeStream
```

```
    .queryName("clickEventStatics")
    .outputMode("append")
    .trigger(Trigger.ProcessingTime(triggerMills.millis))
    .format("kafka")
    .option("kafka.bootstrap.servers", bootstrapServers)
    .option("kafka.security.protocol", protocol)
    .option("kafka.sasl.kerberos.service.name", service)
    .option("kafka.kerberos.domain.name", domain)
    .option("topic", "clickEventStatics")
    .start()

   new Thread(new Runnable {
    override def run(): Unit = {
     while(true) {
      println("-------------get showStatic progress---------")
      //println(showStaticsQuery.lastProgress)
      println(showStaticsQuery.status)
      println("-------------get clickStatic progress---------")
      //println(clickStaticsQuery.lastProgress)
      println(clickStaticsQuery.status)
      Thread.sleep(10000)
     }
    }
   }).start

   spark.streams.awaitAnyTermination()

  }
}
```

27.5.10 Spark Structured Streaming 状态操作样例程序

27.5.10.1 Spark Structured Streaming 状态操作样例程序开发思路

场景说明

假设需要跨批次统计每个session期间发生了多少次event以及本session的开始和结束timestamp；

同时输出本批次被更新状态的session。

数据规划

1. 在kafka中生成模拟数据（需要有Kafka权限用户）。
2. 确保集群安装完成，包括安装HDFS、Yarn、Spark2x和Kafka服务。
3. 将Kafka的Broker配置参数“allow.everyone.if.no.acl.found”的值修改为“true”。
4. 创建Topic。

   {zkQuorum}表示ZooKeeper集群信息，格式为IP:port。

   $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper {zkQuorum}/kafka --replication-factor 1 --partitions 1 --topic {Topic}
5. 启动Kafka的Producer，向Kafka发送数据。

   {ClassPath}表示工程jar包的存放路径，详细路径由用户指定，可参考**在Linux环境中编包并运行Spark程序**章节中导出jar包的操作步骤。

   java -cp $SPARK_HOME/conf:$SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:{ClassPath}

com.huawei.bigdata.spark.examples.KafkaProducer {brokerlist} {topic} {number of events produce every 0.02s}

示例：

java -cp /opt/client/Spark2x/spark/conf:/opt/StructuredStreamingState-1.0.jar:/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/jars/streamingClient010/* com.huawei.bigdata.spark.examples.KafkaProducer xxx.xxx.xxx.xxx:21005,xxx.xxx.xxx.xxx:21005,xxx.xxx.xxx.xxx:21005 mytopic 10

开发思路

1. 接收Kafka中数据，生成相应DataStreamReader。
2. 进行分类统计。
3. 计算结果，并进行打印。

运行前置操作

安全模式下Spark Core样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt”）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上（文件上传的路径需要和生成的jar包路径一致）。

运行任务

在运行样例程序时需要指定 <brokers> <subscribe-type><kafkaProtocol> <kafkaService> <kafkaDomain> <topic> <checkpointLocation>，其中<brokers>指获取元数据的Kafka地址（需使用21007端口），<subscribe-type> 指定kakfa的消费方式，<kafkaProtocol>指安全访问协议（如SASL_PLAINTEXT），<kafkaService>指kerberos服务名称（如kafka），<kafkaDomain>指kerberos域名（如hadoop.*<系统域名>*），<topic>指要消费的kafka topic，<checkpointLocation> 指spark任务的checkpoint保存地址。

说明

由于Spark Structured Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Streaming Structured Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

由于运行模式为安全模式，需要添加新配置并修改命令参数：

1. $SPARK_HOME/conf/jaas.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=false
   useTicketCache=true
   debug=false;
   };
   ```
2. $SPARK_HOME/conf/jaas-zk.conf添加新配置：
   ```
   KafkaClient {
   com.sun.security.auth.module.Krb5LoginModule required
   useKeyTab=true
   keyTab="./user.keytab"
   principal="sparkuser@<系统域名>"
   useTicketCache=false
   storeKey=true
   debug=true;
   };
   ```
3. 使用--files和相对路径提交keytab文件，这样才能保证keytab文件被加载到executor的container中

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- **bin/spark-submit --master yarn --deploy-mode client --files** *<local Path>/jaas.conf,<local path>/user.keytab* ***--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")*** **--class** *com.huawei.bigdata.spark.examples.kafkaSessionization /opt/StructuredStreamingState-1.0.jar <brokers> <subscribe-type> <kafkaProtocol> <kafkaService> <kafkaDomain>* <topic> <checkpointLocation>

  其中配置示例如下：

  ```
  --files ./jaas.conf,./user.keytab //使用--files指定jaas.conf和keytab文件。
  ```

注意

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，当前版本用户除了这一步外还需要将**$SPARK_HOME/jars/streamingClient010**目录中的kafka-clients jar包复制到**$SPARK_HOME/jars**目录下，否则会报class not found异常。

27.5.10.2 Spark Structured Streaming 状态操作样例程序（Scala）

功能介绍

在Spark结构流应用中，跨批次统计每个session期间发生了多少次event以及本session的开始和结束timestamp；同时输出本批次被更新状态的session。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.kafkaSessionization。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。。

```
object kafkaSessionization {
  def main(args: Array[String]): Unit = {
    if (args.length < 7) {
      System.err.println("Usage: kafkaSessionization <bootstrap-servers> " +
        "<subscribe-type> <protocol> <service> <domain> <topics> <checkpointLocation>")
      System.exit(1)
    }

    val Array(bootstrapServers, subscribeType, protocol, service, domain,topics，checkpointLocation) = args

    val spark = SparkSession
      .builder
      .appName("kafkaSessionization")
      .getOrCreate()

    spark.conf.set("spark.sql.streaming.checkpointLocation", checkpointLocation)

    spark.streams.addListener(new StreamingQueryListener {

      @volatile private var startTime: Long = 0L
      @volatile private var endTime: Long = 0L
      @volatile private var numRecs: Long = 0L

      override def onQueryStarted(event: StreamingQueryListener.QueryStartedEvent): Unit = {
        println("Query started: " + event.id)
        startTime = System.currentTimeMillis
      }

      override def onQueryProgress(event: StreamingQueryListener.QueryProgressEvent): Unit = {
        println("Query made progress: " + event.progress)
        numRecs += event.progress.numInputRows
      }

      override def onQueryTerminated(event: StreamingQueryListener.QueryTerminatedEvent): Unit = {
        println("Query terminated: " + event.id)
        endTime = System.currentTimeMillis
      }
    })


    import spark.implicits._

    val df = spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option("kafka.security.protocol", protocol)
      .option("kafka.sasl.kerberos.service.name", service)
      .option("kafka.kerberos.domain.name", domain)
      .option(subscribeType, topics)
      .load()
      .selectExpr("CAST(value AS STRING)")
      .as[String]
      .map { x =>
        val splitStr = x.split(",")
        (splitStr(0), Timestamp.valueOf(splitStr(1)))
      }.as[(String, Timestamp)].flatMap { case(line, timestamp) =>
      line.split(" ").map(word => Event(sessionId = word, timestamp))}
```

```
// Sessionize the events. Track number of events, start and end timestamps of session, and
// and report session updates.
val sessionUpdates = df
  .groupByKey(event => event.sessionId)
  .mapGroupsWithState[SessionInfo, SessionUpdate](GroupStateTimeout.ProcessingTimeTimeout) {

  case (sessionId: String, events: Iterator[Event], state: GroupState[SessionInfo]) =>

    // If timed out, then remove session and send final update
    if (state.hasTimedOut) {
      val finalUpdate =
        SessionUpdate(sessionId, state.get.durationMs, state.get.numEvents, expired = true)
      state.remove()
      finalUpdate
    } else {
      // Update start and end timestamps in session
      val timestamps = events.map(_.timestamp.getTime).toSeq
      val updatedSession = if (state.exists) {
        val oldSession = state.get
        SessionInfo(
          oldSession.numEvents + timestamps.size,
          oldSession.startTimestampMs,
          math.max(oldSession.endTimestampMs, timestamps.max))
      } else {
        SessionInfo(timestamps.size, timestamps.min, timestamps.max)
      }
      state.update(updatedSession)

      // Set timeout such that the session will be expired if no data received for 10 seconds
      state.setTimeoutDuration("10 seconds")
      SessionUpdate(sessionId, state.get.durationMs, state.get.numEvents, expired = false)
    }
}
// Start running the query that prints the session updates to the console
val query = sessionUpdates
  .writeStream
  .outputMode("update")
  .format("console")
  .start()

query.awaitTermination()
}
```

27.5.11 Spark 同时访问两个 HBase 样例程序

27.5.11.1 Spark 同时访问两个 HBase 样例程序开发思路

场景说明

spark支持同时访问两个集群中的HBase，前提是两个集群配置了互信。

数据规划

1. 将cluster2集群的所有Zookeeper节点和HBase节点的IP和主机名配置到cluster1集群的客户端节点的“/etc/hosts”文件中。
2. 分别将cluster1和cluster2集群Spark2x客户端conf下的hbase-site.xml文件放到“/opt/example/A”，“/opt/example/B”两个目录下。
3. 用spark-submit提交命令：

说明

运行样例程序前，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”）。

```
spark-submit --master yarn --deploy-mode client --files /opt/example/B/hbase-site.xml --keytab /opt/FIclient/user.keytab --principal sparkuser  --class com.huawei.spark.examples.SparkOnMultiHbase /opt/example/SparkOnMultiHbase-1.0.jar
```

开发思路

1. 用户访问HBase时，需要使用对应集群的配置文件创建Configuration对象，用于创建Connection对象。
2. 用对应的Connection对象操作HBase表，包括建表、插入数据、查看数据并进行打印。

27.5.11.2 Spark 同时访问两个 HBase 样例程序（Scala）

下面代码片段仅为演示，具体代码参见：com.huawei.spark.examples.SparkOnMultiHbase

```
def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("SparkOnMultiHbaseExample")
    conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    conf.set("spark.kryo.registrator", "com.huawei.spark.examples.MyRegistrator")
    val sc = new SparkContext(conf)val tableName = "SparkOnMultiHbase"
    val clusterFlagList=List("B", "A")
    clusterFlagList.foreach{ item =>
        val hbaseConf = getConf( item )
        println(hbaseConf.get("hbase.zookeeper.quorum"))
        val hbaseUtil = new HbaseUtil(sc,hbaseConf)
        hbaseUtil.writeToHbase(tableName)
        hbaseUtil.readFromHbase(tableName)
    }
    sc.stop()
}
private def getConf(item:String):Configuration={
    val conf: Configuration = HBaseConfiguration.create()
    val url = "/opt" + File.separator + "example" + File.separator + item + File.separator + "hbase-site.xml"
    conf.addResource(new File(url).toURI.toURL)
    conf
}
```

27.5.12 Spark 同步 HBase 数据到 CarbonData 样例程序

27.5.12.1 Spark 同步 HBase 数据到 CarbonData 开发思路

场景说明

数据实时写入HBase，用于点查业务，数据每隔一段时间批量同步到CarbonData表中，用于分析型查询业务。

运行前置操作

安全模式下该样例代码需要读取两个文件（user.keytab、krb5.conf）。user.keytab和krb5.conf文件为安全模式下的认证文件，需要在FusionInsight Manager中下载

principal用户的认证凭证，样例代码中使用的用户为：sparkuser，需要修改为准备好的开发用户。

打包项目

1. 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
2. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

   说明

   - 编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。例如：“/opt/user.keytab”，“/opt/krb5.conf”。
   - 运行样例程序前，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”）。
3. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/”）下。

数据规划

1. 创建HBase表，构造数据，列需要包含key，modify_time，valid。其中每条数据key值全表唯一，modify_time代表修改时间，valid代表是否为有效数据（该样例中'1'为有效，'0'为无效数据）。

   示例：进入hbase shell，执行如下命令：

   create 'hbase_table','key','info'

   put 'hbase_table','1','info:modify_time',*'2019-11-22 23:28:39'*

   put 'hbase_table','1','info:valid','1'

   put 'hbase_table','2','info:modify_time',*'2019-11-22 23:28:39'*

   put 'hbase_table','2','info:valid','1'

   put 'hbase_table','3','info:modify_time',*'2019-11-22 23:28:39'*

   put 'hbase_table','3','info:valid','0'

   put 'hbase_table','4','info:modify_time',*'2019-11-22 23:28:39'*

   put 'hbase_table','4','info:valid','1'

   说明

   上述数据的modify_time列可设置为当前时间之前的值。

   put 'hbase_table','5','info:modify_time',*'2021-03-03 15:20:39'*

   put 'hbase_table','5','info:valid','1'

   put 'hbase_table','6','info:modify_time',*'2021-03-03 15:20:39'*

   put 'hbase_table','6','info:valid','1'

   put 'hbase_table','7','info:modify_time',*'2021-03-03 15:20:39'*

   put 'hbase_table','7','info:valid','0'

   put 'hbase_table','8','info:modify_time',*'2021-03-03 15:20:39'*

   put 'hbase_table','8','info:valid','1'

   put 'hbase_table','4','info:valid','0'

put 'hbase_table','4','info:modify_time','*2021-03-03 15:20:39*'

说明

上述数据的modify_time列可设置为样例程序启动后30分钟内的时间值（此处的30分钟为样例程序默认的同步间隔时间，可修改）。

put 'hbase_table','9','info:modify_time','*2021-03-03 15:32:39*'

put 'hbase_table','9','info:valid','1'

put 'hbase_table','10','info:modify_time','*2021-03-03 15:32:39*'

put 'hbase_table','10','info:valid','1'

put 'hbase_table','11','info:modify_time','*2021-03-03 15:32:39*'

put 'hbase_table','11','info:valid','0'

put 'hbase_table','12','info:modify_time','*2021-03-03 15:32:39*'

put 'hbase_table','12','info:valid','1'

说明

上述数据的modify_time列可设置为样例程序启动后30分钟到60分钟内的时间值，即第二次同步周期。

2. 在sparksql中创建HBase的hive外表，命令如下：

   create table *external_hbase_table(key string ,modify_time STRING, valid STRING)*

   using org.apache.spark.sql.hbase.HBaseSource

   options*(hbaseTableName "hbase_table", keyCols "key", colsMapping "modify_time=info.modify_time,valid=info.valid");*

3. 在sparksql中创建CarbonData表：

   create table *carbon01(key string,modify_time STRING, valid STRING)* stored as carbondata;

4. 初始化加载当前hbase表中所有数据到CarbonData表；

   insert into table *carbon01* select * **from** *external_hbase_table* where *valid='1'*;

5. 用spark-submit提交命令：

```
spark-submit --master yarn --deploy-mode client --keytab /opt/FIclient/user.keytab --principal sparkuser  --class com.huawei.bigdata.spark.examples.HBaseExternalHivetoCarbon /opt/example/HBaseExternalHivetoCarbon-1.0.jar
```

27.5.12.2 Spark 同步 HBase 数据到 CarbonData（Java）

下面代码片段仅为演示，具体代码参见：com.huawei.spark.examples.HBaseExternalHivetoCarbon。

```
public static void main(String[] args) throws Exception {
spark = SparkSession.builder().appName("HBaseExternalHiveToCarbon").getOrCreate();

Timer timer = new Timer();
timer.schedule(new TimerTask() {
public void run() {
timeEnd = timeStart + TIMEWINDOW;

queryTimeStart = transferDateToStr(timeStart);
queryTimeEnd = transferDateToStr(timeEnd);

//run delete logic
cmdsb = new StringBuilder();
cmdsb.append("delete from ")
```

```
.append(carbonTableName)
.append("  where key in (select key from ")
.append(externalHiveTableName)
.append(" where modify_time>'")
.append(queryTimeStart)
.append("' and modify_time<'")
.append(queryTimeEnd)
.append("' and valid='0')");
spark.sql(cmdsb.toString());

//run insert logic
cmdsb = new StringBuilder();
cmdsb.append("insert into ")
.append(carbonTableName)
.append("  select * from ")
.append(externalHiveTableName)
.append(" where modify_time>'")
.append(queryTimeStart)
.append("'  and modify_time<'")
.append(queryTimeEnd)
.append("'  and valid='1'");
spark.sql(cmdsb.toString());

timeStart = timeEnd;
}
}, TIMEWINDOW, TIMEWINDOW);
}
```

27.5.13 使用 Spark 执行 Hudi 样例程序

27.5.13.1 使用 Spark 执行 Hudi 样例程序开发思路

场景说明

本章节介绍如何使用Spark操作Hudi执行插入数据、查询数据、更新数据、增量查询、特定时间点查询、删除数据等操作。

详细代码请参考样例代码。

打包项目

1. 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上。
2. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

   说明

   - 编译打包前，样例代码中的user.keytab、krb5.conf文件路径需要修改为该文件所在客户端服务器的实际路径。
   - 运行Python样例代码无需通过Maven打包，只需要上传user.keytab、krb5.conf 文件到客户端所在服务器上。
3. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/example/”）下。

运行任务

1. 登录Spark客户端节点，执行如下命令：

   source *客户端安装目录*/**bigdata_env**

source *客户端安装目录*/**Hudi/component_env**

kinit *Hudi开发用户*

2. 编译构建样例代码后可以使用**spark-submit**提交命令，执行命令后会依次执行写入、更新、查询、删除等操作：
   - 运行Java样例程序：

     spark-submit --keytab <user_keytab_path> --principal=<principal_name> --class com.huawei.bigdata.hudi.examples.HoodieWriteClientExample /opt/example/hudi-java-security-examples-1.0.jar hdfs://hacluster/tmp/example/hoodie_java hoodie_java

     其中：“ <user_keytab_path> ”为认证文件路径，“<principal_name>”为认证用户名，“/opt/example/hudi-java-examples-1.0.jar”为jar包路径，“hdfs://hacluster/tmp/example/hoodie_java”为Hudi表的存储路径，“ hoodie_java”为Hudi表的表名。
   - 运行Scala样例程序：

     spark-submit --keytab <user_keytab_path> --principal=<principal_name> --class com.huawei.bigdata.hudi.examples.HoodieDataSourceExample /opt/example/hudi-scala-security-examples-1.0.jar hdfs://hacluster/tmp/example/hoodie_scala hoodie_scala

     其中：“/opt/example/hudi-scala-examples-1.0.jar”为jar包路径，“ <user_keytab_path> ”为认证文件路径，“<principal_name>”为认证用户名，“hdfs://hacluster/tmp/example/hoodie_scala”为Hudi表的存储路径，“ hoodie_Scala”为Hudi表的表名。
   - 运行Python样例程序：

     spark-submit /opt/example/HudiPythonExample.py hdfs://hacluster/tmp/huditest/example/python hudi_trips_cow

     其中：“hdfs://hacluster/tmp/huditest/example/python”为Hudi表的存储路径，“ hudi_trips_cow”为Hudi表的表名。

27.5.13.2 使用 Spark 执行 Hudi 样例程序（Java）

下面代码片段仅为演示，具体代码参见：
com.huawei.bigdata.hudi.examples.HoodieWriteClientExample

创建客户端对象来操作Hudi：

```
String tablePath = args[0];
String tableName = args[1];
SparkConf sparkConf = HoodieExampleSparkUtils.defaultSparkConf("hoodie-client-example");
JavaSparkContext jsc = new JavaSparkContext(sparkConf);
// Generator of some records to be loaded in.
HoodieExampleDataGenerator<HoodieAvroPayload> dataGen = new HoodieExampleDataGenerator<>();
// initialize the table, if not done already
Path path = new Path(tablePath);
FileSystem fs = FSUtils.getFs(tablePath, jsc.hadoopConfiguration());
if (!fs.exists(path)) {
HoodieTableMetaClient.initTableType(jsc.hadoopConfiguration(), tablePath,
HoodieTableType.valueOf(tableType),
tableName, HoodieAvroPayload.class.getName());
}

// Create the write client to write some records in
HoodieWriteConfig cfg = HoodieWriteConfig.newBuilder().withPath(tablePath)
   .withSchema(HoodieExampleDataGenerator.TRIP_EXAMPLE_SCHEMA).withParallelism(2, 2)
```



```
    .withDeleteParallelism(2).forTable(tableName)
    .withIndexConfig(HoodieIndexConfig.newBuilder().withIndexType(HoodieIndex.IndexType.BLOOM).build()
)
    .withCompactionConfig(HoodieCompactionConfig.newBuilder().archiveCommitsWith(20,
30).build()).build();
SparkRDDWriteClient<HoodieAvroPayload> client = new SparkRDDWriteClient<>(new
HoodieSparkEngineContext(jsc), cfg);
```

插入数据：

```
String newCommitTime = client.startCommit();
LOG.info("Starting commit " + newCommitTime);
List<HoodieRecord<HoodieAvroPayload>> records = dataGen.generateInserts(newCommitTime, 10);
List<HoodieRecord<HoodieAvroPayload>> recordsSoFar = new ArrayList<>(records);
JavaRDD<HoodieRecord<HoodieAvroPayload>> writeRecords = jsc.parallelize(records, 1);
client.upsert(writeRecords, newCommitTime);
```

更新数据：

```
newCommitTime = client.startCommit();
LOG.info("Starting commit " + newCommitTime);
List<HoodieRecord<HoodieAvroPayload>> toBeUpdated = dataGen.generateUpdates(newCommitTime, 2);
records.addAll(toBeUpdated);
recordsSoFar.addAll(toBeUpdated);
writeRecords = jsc.parallelize(records, 1);
client.upsert(writeRecords, newCommitTime);
```

删除数据：

```
newCommitTime = client.startCommit();
LOG.info("Starting commit " + newCommitTime);
// just delete half of the records
int numToDelete = recordsSoFar.size() / 2;
List<HoodieKey> toBeDeleted =
recordsSoFar.stream().map(HoodieRecord::getKey).limit(numToDelete).collect(Collectors.toList());
JavaRDD<HoodieKey> deleteRecords = jsc.parallelize(toBeDeleted, 1);
client.delete(deleteRecords, newCommitTime);
```

压缩数据：

```
if (HoodieTableType.valueOf(tableType) == HoodieTableType.MERGE_ON_READ) {
    Option<String> instant = client.scheduleCompaction(Option.empty());
    JavaRDD<WriteStatus> writeStatues = client.compact(instant.get());
    client.commitCompaction(instant.get(), writeStatues, Option.empty());
}
```

27.5.13.3 使用 Spark 执行 Hudi 样例程序（Scala）

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.hudi.examples.HoodieDataSourceExample。

插入数据：

```
def insertData(spark: SparkSession, tablePath: String, tableName: String, dataGen:
HoodieExampleDataGenerator[HoodieAvroPayload]): Unit = {
val commitTime: String = System.currentTimeMillis().toString
val inserts = dataGen.convertToStringList(dataGen.generateInserts(commitTime, 20))
spark.sparkContext.parallelize(inserts, 2)
val df = spark.read.json(spark.sparkContext.parallelize(inserts, 1))df.write.format("org.apache.hudi").
     options(getQuickstartWriteConfigs).
     option(PRECOMBINE_FIELD_OPT_KEY, "ts").
     option(RECORDKEY_FIELD_OPT_KEY, "uuid").
     option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
     option(TABLE_NAME, tableName).
     mode(Overwrite).
     save(tablePath)}
```

查询数据：


```
def queryData(spark: SparkSession, tablePath: String, tableName: String, dataGen:
HoodieExampleDataGenerator[HoodieAvroPayload]): Unit = {
val roViewDF = spark.
      read.
      format("org.apache.hudi").
      load(tablePath + "/*/*/*/*")
roViewDF.createOrReplaceTempView("hudi_ro_table")
spark.sql("select fare, begin_lon, begin_lat, ts from  hudi_ro_table where fare > 20.0").show()
//  +-----------------+-------------------+-------------------+---+
//  |             fare|          begin_lon|          begin_lat| ts|
//  +-----------------+-------------------+-------------------+---+
//  |98.88075495133515|0.3955604862303160 3|0.17851135255091155|0.0|
//  ...
spark.sql("select _hoodie_commit_time, _hoodie_record_key, _hoodie_partition_path, rider, driver, fare from
hudi_ro_table").show()
//  +-------------------+--------------------+----------------------+-------------------+--------------------
+-----------------+
//  |_hoodie_commit_time|  _hoodie_record_key|_hoodie_partition_path|               rider|
driver|               fare|
//  +-------------------+--------------------+----------------------+-------------------+--------------------
+-----------------+
//  |     20191231181501|31cafb9f-0196-4b1...|            2020/01/02|rider-1577787297889|
driver-1577787297889| 98.88075495133515|
//  ...
}
```

更新数据：

```
def updateData(spark: SparkSession, tablePath: String, tableName: String, dataGen:
HoodieExampleDataGenerator[HoodieAvroPayload]): Unit = {
val commitTime: String = System.currentTimeMillis().toString
val updates = dataGen.convertToStringList(dataGen.generateUpdates(commitTime, 10))
val df = spark.read.json(spark.sparkContext.parallelize(updates, 1))
df.write.format("org.apache.hudi").
      options(getQuickstartWriteConfigs).
      option(PRECOMBINE_FIELD_OPT_KEY, "ts").
      option(RECORDKEY_FIELD_OPT_KEY, "uuid").
      option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
      option(TABLE_NAME, tableName).
      mode(Append).
      save(tablePath)}
```

增量查询：

```
def incrementalQuery(spark: SparkSession, tablePath: String, tableName: String) {
import spark.implicits._
val commits = spark.sql("select distinct(_hoodie_commit_time) as commitTime from hudi_ro_table order by
commitTime").map(k => k.getString(0)).take(50)
val beginTime = commits(commits.length - 2)

val incViewDF = spark.
      read.
      format("org.apache.hudi").
      option(QUERY_TYPE_OPT_KEY, QUERY_TYPE_INCREMENTAL_OPT_VAL).
      option(BEGIN_INSTANTTIME_OPT_KEY, beginTime).
      load(tablePath)
incViewDF.createOrReplaceTempView("hudi_incr_table")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from hudi_incr_table where fare >
20.0").show()}
```

特定时间点查询：

```
def pointInTimeQuery(spark: SparkSession, tablePath: String, tableName: String) {
import spark.implicits._
val commits = spark.sql("select distinct(_hoodie_commit_time) as commitTime from  hudi_ro_table order by
commitTime").map(k => k.getString(0)).take(50)
val beginTime = "000"
// Represents all commits > this time.
val endTime = commits(commits.length - 2)
// commit time we are interested in
```

```
//incrementally query data
val incViewDF = spark.read.format("org.apache.hudi").
    option(QUERY_TYPE_OPT_KEY, QUERY_TYPE_INCREMENTAL_OPT_VAL).
    option(BEGIN_INSTANTTIME_OPT_KEY, beginTime).
    option(END_INSTANTTIME_OPT_KEY, endTime).
    load(tablePath)
incViewDF.createOrReplaceTempView("hudi_incr_table")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from  hudi_incr_table where fare > 20.0").show()}
```

27.5.13.4 使用 Spark 执行 Hudi 样例程序（Python）

下面代码片段仅为演示，具体代码参见：HudiPythonExample.py。

插入数据：

```
#insert
inserts = sc._jvm.org.apache.hudi.QuickstartUtils.convertToStringList(dataGen.generateInserts(10))
df = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
hudi_options = {
'hoodie.table.name': tableName,
'hoodie.datasource.write.recordkey.field': 'uuid',
'hoodie.datasource.write.partitionpath.field': 'partitionpath',
'hoodie.datasource.write.table.name': tableName,
'hoodie.datasource.write.operation': 'insert',
'hoodie.datasource.write.precombine.field': 'ts',
'hoodie.upsert.shuffle.parallelism': 2,'
hoodie.insert.shuffle.parallelism': 2
}
df.write.format("hudi"). \
    options(**hudi_options). \
    mode("overwrite"). \
    save(basePath)
```

查询数据：

```
tripsSnapshotDF = spark. \
    read. \
    format("hudi"). \
    load(basePath + "/*/*/*/*")
tripsSnapshotDF.createOrReplaceTempView("hudi_trips_snapshot")
spark.sql("select fare, begin_lon, begin_lat, ts from  hudi_trips_snapshot where fare > 20.0").show()
spark.sql("select _hoodie_commit_time, _hoodie_record_key, _hoodie_partition_path, rider, driver, fare from hudi_trips_snapshot").show()
```

更新数据：

```
updates = sc._jvm.org.apache.hudi.QuickstartUtils.convertToStringList(dataGen.generateUpdates(10))
df = spark.read.json(spark.sparkContext.parallelize(updates, 2))
df.write.format("hudi"). \
    options(**hudi_options). \
    mode("append"). \
    save(basePath)
```

增量查询：

```
spark. \
    read. \
    format("hudi"). \
    load(basePath + "/*/*/*/*"). \
    createOrReplaceTempView("hudi_trips_snapshot")
incremental_read_options = {
    'hoodie.datasource.query.type': 'incremental',
    'hoodie.datasource.read.begin.instanttime': beginTime,
}
tripsIncrementalDF = spark.read.format("hudi"). \
    options(**incremental_read_options). \
    load(basePath)
tripsIncrementalDF.createOrReplaceTempView("hudi_trips_incremental")
```

```
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from  hudi_trips_incremental where
fare > 20.0").show()
```

特定时间点查询：

```
# Represents all commits > this time.
beginTime = "000"
endTime = commits[len(commits) - 2]
point_in_time_read_options = {
    'hoodie.datasource.query.type': 'incremental',
    'hoodie.datasource.read.end.instanttime': endTime,
    'hoodie.datasource.read.begin.instanttime': beginTime
}

tripsPointInTimeDF = spark.read.format("hudi"). \
    options(**point_in_time_read_options). \
    load(basePath)

tripsPointInTimeDF.createOrReplaceTempView("hudi_trips_point_in_time")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from hudi_trips_point_in_time where
fare > 20.0").show()
```

删除数据：

```
# 获取记录总数
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").count()
# 拿到两条将被删除的记录
ds = spark.sql("select uuid, partitionpath from hudi_trips_snapshot").limit(2)
# 执行删除
hudi_delete_options = {
    'hoodie.table.name': tableName,
    'hoodie.datasource.write.recordkey.field': 'uuid',
    'hoodie.datasource.write.partitionpath.field': 'partitionpath',
    'hoodie.datasource.write.table.name': tableName,
    'hoodie.datasource.write.operation': 'delete',
    'hoodie.datasource.write.precombine.field': 'ts',
    'hoodie.upsert.shuffle.parallelism': 2,
    'hoodie.insert.shuffle.parallelism': 2
}
from pyspark.sql.functions import lit
deletes = list(map(lambda row: (row[0], row[1]), ds.collect()))
df = spark.sparkContext.parallelize(deletes).toDF(['uuid', 'partitionpath']).withColumn('ts', lit(0.0))
df.write.format("hudi"). \
    options(**hudi_delete_options). \
    mode("append"). \
    save(basePath)
# 像之前一样运行查询
roAfterDeleteViewDF = spark. \
    read. \
    format("hudi"). \
    load(basePath + "/*/*/*/*")
roAfterDeleteViewDF.registerTempTable("hudi_trips_snapshot")
# 应返回 (total - 2) 条记录
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").count()
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").show()
```

27.5.14 Hudi 的自定义配置项样例程序

27.5.14.1 HoodieDeltaStreamer

编写自定义的转化类实现Transformer。

编写自定义的Schema实现SchemaProvider。

在执行HoodieDeltaStreamer时加入参数：

```
--schemaprovider-class 定义的schema类 --transformer-class 定义的transform类
```

Transformer和SchemaProvider样例：

```
public class TransformerExample implements Transformer, Serializable {
    @Override
    public Dataset<Row> apply(JavaSparkContext jsc, SparkSession sparkSession, Dataset<Row> rowDataset,
                              TypedProperties properties) {
        JavaRDD<Row> rowJavaRdd = rowDataset.toJavaRDD();
        List<Row> rowList =new ArrayList<>();
        for(Row row: rowJavaRdd.collect()){
            rowList.add(buildRow(row));
        }
        JavaRDD<Row> stringJavaRdd = jsc.parallelize(rowList);
        List<StructField> fields = new ArrayList<>();
        builFields(fields);
        StructType schema = DataTypes.createStructType(fields);
        Dataset<Row> dataFrame = sparkSession.createDataFrame(stringJavaRdd, schema);
        return dataFrame;
    }

    private void builFields(List<StructField> fields) {
        fields.add(DataTypes.createStructField("age", DataTypes.StringType, true));
        fields.add(DataTypes.createStructField("id", DataTypes.StringType, true));
        fields.add(DataTypes.createStructField("name", DataTypes.StringType, true));
        fields.add(DataTypes.createStructField("job", DataTypes.StringType, true));
    }

    private Row buildRow(Row row){
        String age = row.getString(0);
        String id = row.getString(1);
        String job = row.getString(2);
        String name = row.getString(3);
        Row returnRow = RowFactory.create(age, id, job, name);
        return returnRow;
    }
}
public class DataSchemaProviderExample extends SchemaProvider {

    public DataSchemaProviderExample(TypedProperties props, JavaSparkContext jssc) {
        super(props, jssc);
    }

    @Override
    public Schema getSourceSchema() {
        Schema avroSchema = new Schema.Parser().parse(
            "{\"type\":\"record\",\"name\":\"hoodie_source\",\"fields\":[{\"name\":\"age\",\"type\":\"string\"},
{\"name\":\"id\",\"type\":\"string\"},{\"name\":\"job\",\"type\":\"string\"},{\"name\":\"name\",\"type
\":\"string\"}]}");
        return avroSchema;
    }

    @Override
    public Schema getTargetSchema() {
        Schema avroSchema = new Schema.Parser().parse(
            "{\"type\":\"record\",\"name\":\"mytest_record\",\"namespace\":\"hoodie.mytest\",\"fields\":
[{\"name\":\"age\",\"type\":\"string\"},{\"name\":\"id\",\"type\":\"string\"},{\"name\":\"job\",\"type\":\"string
\"},{\"name\":\"name\",\"type\":\"string\"}]}");
        return avroSchema;
    }

}
```

27.5.14.2 自定义排序器

编写自定义排序类继承BulkInsertPartitioner，在写入Hudi时加入配置：

```
.option(BULKINSERT_USER_DEFINED_PARTITIONER_CLASS, <自定义排序类的包名加类名>)
```


自定义分区排序器样例：

```
public class HoodieSortExample<T extends HoodieRecordPayload>
    implements BulkInsertPartitioner<JavaRDD<HoodieRecord<T>>> {
    @Override
    public JavaRDD<HoodieRecord<T>> repartitionRecords(JavaRDD<HoodieRecord<T>> records, int
outputSparkPartitions) {
        JavaPairRDD<String,
            HoodieRecord<T>> stringHoodieRecordJavaPairRDD = records.coalesce(outputSparkPartitions)
                .mapToPair(record -> new Tuple2<>(new StringBuilder().append(record.getPartitionPath())
                    .append("+")
                    .append(record.getRecordKey())
                    .toString(), record));
        JavaRDD<HoodieRecord<T>> hoodieRecordJavaRDD =
stringHoodieRecordJavaPairRDD.mapPartitions(partition -> {
            List<Tuple2<String, HoodieRecord<T>>> recordList = new ArrayList<>();
            for (; partition.hasNext();) {
                recordList.add(partition.next());
            }
            Collections.sort(recordList, (o1, o2) -> {
                if (o1._1().split("[+]")[0] == o2._1().split("[+]")[0]) {
                    return Integer.parseInt(o1._1().split("[+]")[1]) - Integer.parseInt(o2._1().split("[+]")[1]);
                } else {
                    return o1._1().split("[+]")[0].compareTo(o2._1().split("[+]")[0]);
                }
            });
            return recordList.stream().map(e -> e._2).iterator();
        });
        return hoodieRecordJavaRDD;
    }

    @Override
    public boolean arePartitionRecordsSorted() {
        return true;
    }
}
```

27.6 调测 Spark 应用

27.6.1 在本地 Windows 环境中调测 Spark 应用

27.6.1.1 配置 Windows 通过 EIP 访问集群 Spark

操作场景

该章节通过指导用户配置集群绑定EIP，并配置Spark文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行SparkScalaExample样例为例进行说明。

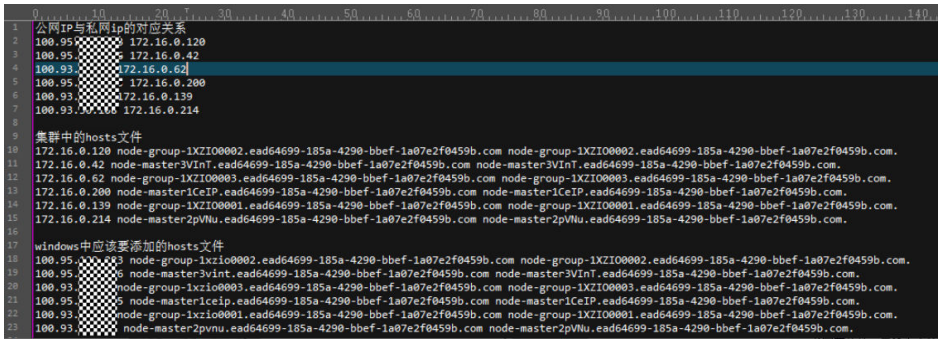
操作步骤

步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。



步骤2 将krb5.conf文件中的IP地址修改为对应IP的主机名称。

步骤3 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和21730TCP、21731TCP/UDP、21732TCP/UDP端口。

步骤4 在Manager界面选择“集群 > 服务 > Spark2x > 更多 > 下载客户端”，将客户端中的core-site.xml和hdfs-site.xml复制到样例工程的conf目录下。

对hdfs-site.xml添加如下内容：

```
<property>
    <name>dfs.client.use.datanode.hostname</name>
    <value>true</value>
</property>
```


对pom.xml文件加入如下内容：

```
<dependency>
    <groupId>com.huawei.mrs</groupId>
    <artifactId>hadoop-plugins</artifactId>
    <version>部件包版本-302002</version>
</dependency>
```

步骤5 运行样例代码前，对SparkSession加入.master("local").config("spark.driver.host", "localhost")，配置Spark为本地运行模式。并将样例代码中的PRNCIPAL_NAME改为安全认证的用户名。

```
object FemaleInfoCollection {
  def main (args: Array[String]) {
//    if (args.length < 1) {
//      System.err.println("Usage: CollectFemaleInfo <file>")
//      System.exit(1)
//    }

    // Configure the Spark application name.
    val spark = SparkSession
      .builder()
      .appName( name = "CollectFemaleInfo")
      .master( master = "local")
      .config("spark.driver.host", "localhost")
      .getOrCreate()
    // Initializing Spark


    // Read data. This code indicates the data path that the input parameter args(0) specifies.
    val text = spark.sparkContext.textFile( path = "/tmp/sparktest/")
    // Filter the data information about the time that female netizens spend online.
    val data = text.filter(_.contains("female"))
```

----结束

27.6.1.2 在本地 Windows 环境中编包并运行 Spark 程序

操作场景

在程序代码完成开发后，您可以在Windows环境中运行应用。使用Scala或Java语言开发的应用程序在IDEA端的运行步骤是一样的。

说明

- Windows环境中目前只提供通过JDBC访问Spark SQL的程序样例代码的运行，其他样例代码暂不提供。
- 用户需保证Maven已配置华为镜像站中SDK的Maven镜像仓库，具体可参考**配置华为开源镜像仓**。

操作步骤

步骤1 获取样例代码。

下载样例工程的Maven工程源码和配置文件，请参见**获取代码样例工程**。

将样例代码导入IDEA中。

步骤2 获取配置文件。

1. 从集群的客户端中获取文件。在“$SPARK_HOME/conf”中下载hive-site.xml与spark-defaults.conf文件到本地。
2. 在集群的FusionInsight Manager页面下载所使用用户的认证文件到本地。

步骤3 在HDFS中上传数据。

1. 在Liunx中新建文本文件data，将如下数据内容保存到data文件中。

```
Miranda,32
Karlie,23
Candice,27
```

2. 在HDFS客户端，执行如下命令获取安全认证。

   cd *{客户端安装目录}*

   kinit *{用于认证的业务用户}*

3. 在Linux系统HDFS客户端使用命令**hadoop fs -mkdir /data**（hdfs dfs命令有同样的作用），创建对应目录。
4. 在Linux系统HDFS客户端使用命令**hadoop fs -put data /data**，上传数据文件。

步骤4 在样例代码中配置相关参数。

1. 认证用户配置。

   userPrincipal配置为所使用的用户。

   userKeytabPath配置为下载的keytab文件的路径。

   Krb5ConfPath配置为下载的krb5.conf文件的路径。

```
public class ThriftServerQueriesTest {
    public static void main(String[] args) throws SQLException, ClassNotFoundException, IOException {
        String userPrincipal = "admintest";
        String userKeytabPath = "D:\\conf\\keytab\\user.keytab";
        String krb5ConfPath = "D:\\conf\\keytab\\krb5.conf";
        String principalName = KerberosUtil.DEFAULT_REALM;
        String ZKServerPrincipal = "zookeeper/hadoop." + principalName;

        String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
        String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
```

   域名配置为DEFAULT_REALM，在KerberosUtil类中修改DEFAULT_REALM为集群的域名。

```
public class KerberosUtil {
    private static Logger logger = Logger.getLogger(KerberosUtil.class);

    public static final String JAVA_VENDER = "java.vendor";
    public static final String IBM_FLAG = "IBM";
    public static final String CONFIG_CLASS_FOR_IBM = "com.ibm.security.krb5.internal.Config";
    public static final String CONFIG_CLASS_FOR_SUN = "sun.security.krb5.Config";
    public static final String METHOD_GET_INSTANCE = "getInstance";
    public static final String METHOD_GET_DEFAULT_REALM = "getDefaultRealm";
    public static final String DEFAULT_REALM = "HADOOP.COM";
```

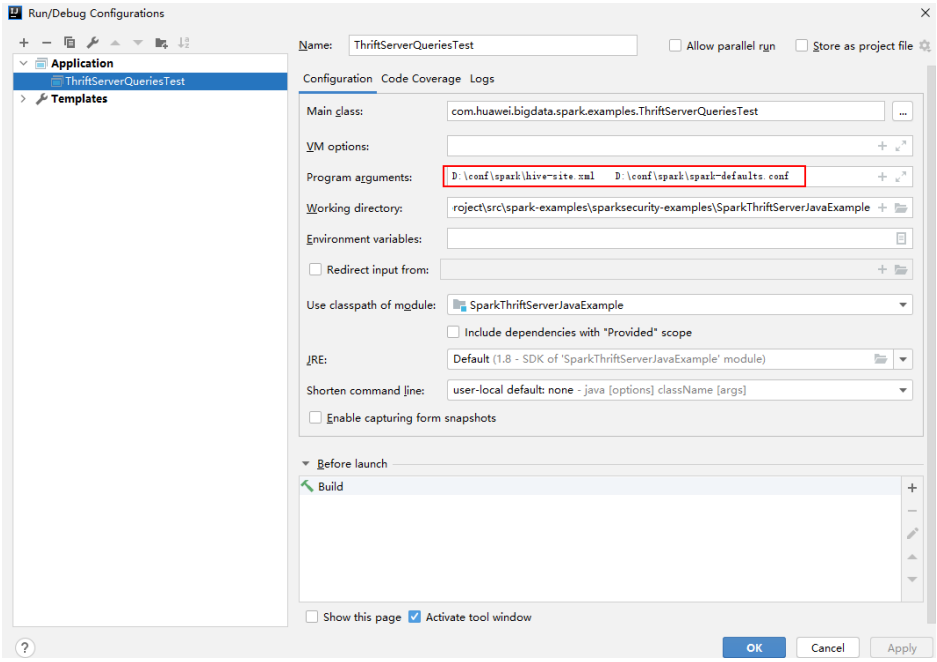
2. 将securityConfig拼接的字符串中user.principal与user.keytab修改为相应的用户名与路径。注意这里keytab的路径需要使用“/”。

```
String securityConfig = ";saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop." + principalName + "@"
    + principalName + ";user.principal=admintest;user.keytab=D:/conf/keytab/user.keytab;";
```

3. 将加载数据的sql语句改为“LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD”。

```
ArrayList<String> sqlList = new ArrayList<~>();
sqlList.add("CREATE TABLE IF NOT EXISTS CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY"
    + " ','");
sqlList.add("LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD");
sqlList.add("SELECT * FROM child");
sqlList.add("DROP TABLE child");
executeSql(url, sqlList);
```

步骤5 在程序运行时添加运行参数，分别为hive-site.xml与spark-defaults.conf文件的路径。



步骤6 运行程序。

----结束

27.6.1.3 在本地 Windows 环境中查看 Spark 程序调试结果

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/D:/mavenlocal/org/apache/logging/log4j/log4j-slf4j-impl/2.6.2/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/D:/mavenlocal/org/slf4j/slf4j-log4j12/1.7.30/slf4j-log4j12-1.7.30.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console.
---- Begin executing sql: CREATE TABLE IF NOT EXISTS CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ----
Result
---- Done executing sql: CREATE TABLE IF NOT EXISTS CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ----
---- Begin executing sql: LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD ----
Result
---- Done executing sql: LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD ----
---- Begin executing sql: SELECT * FROM child ----
NAME    AGE
Miranda    32
Karlie    23
Candice    27
---- Done executing sql: SELECT * FROM child ----
---- Begin executing sql: DROP TABLE child ----
Result
---- Done executing sql: DROP TABLE child ----

Process finished with exit code 0
```

27.6.2 在 Linux 环境中调测 Spark 应用

27.6.2.1 在 Linux 环境中编包并运行 Spark 程序

操作场景

在程序代码完成开发后，您可以上传至Linux客户端环境中运行应用。使用Scala或Java语言开发的应用程序在Spark客户端的运行步骤是一样的。

说明

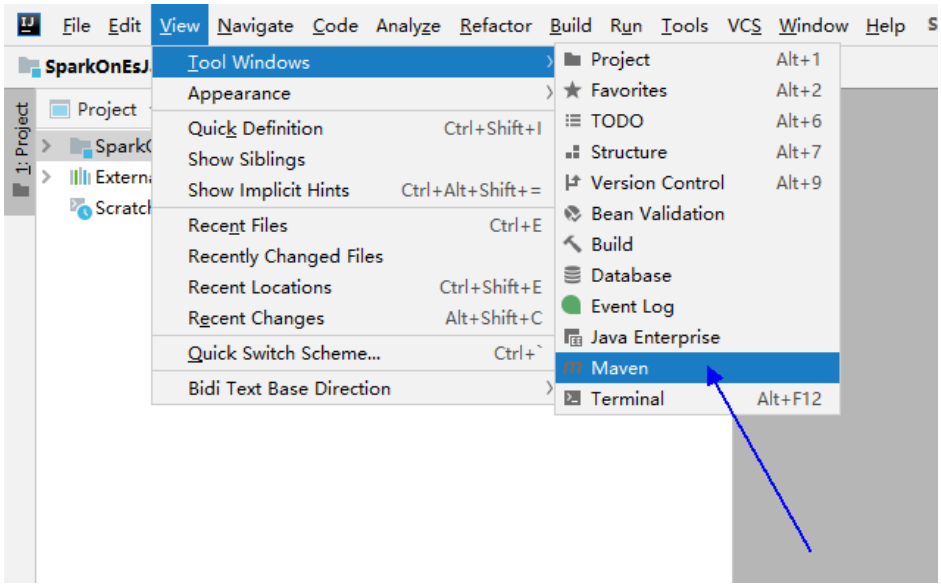
- 使用Python开发的Spark应用程序无需打包成jar，只需将样例工程复制到编译机器上即可。
- 用户需保证worker和driver的Python版本一致，否则将报错："Python in worker has different version %s than that in driver %s."。
- 用户需保证Maven已配置华为镜像站中SDK的Maven镜像仓库，具体可参考**配置华为开源镜像仓**

操作步骤

步骤1 在IntelliJ IDEA中，打开Maven工具窗口。

在IDEA主页面，选择“View->Tool Windows->Maven”打开“Maven”工具窗口。

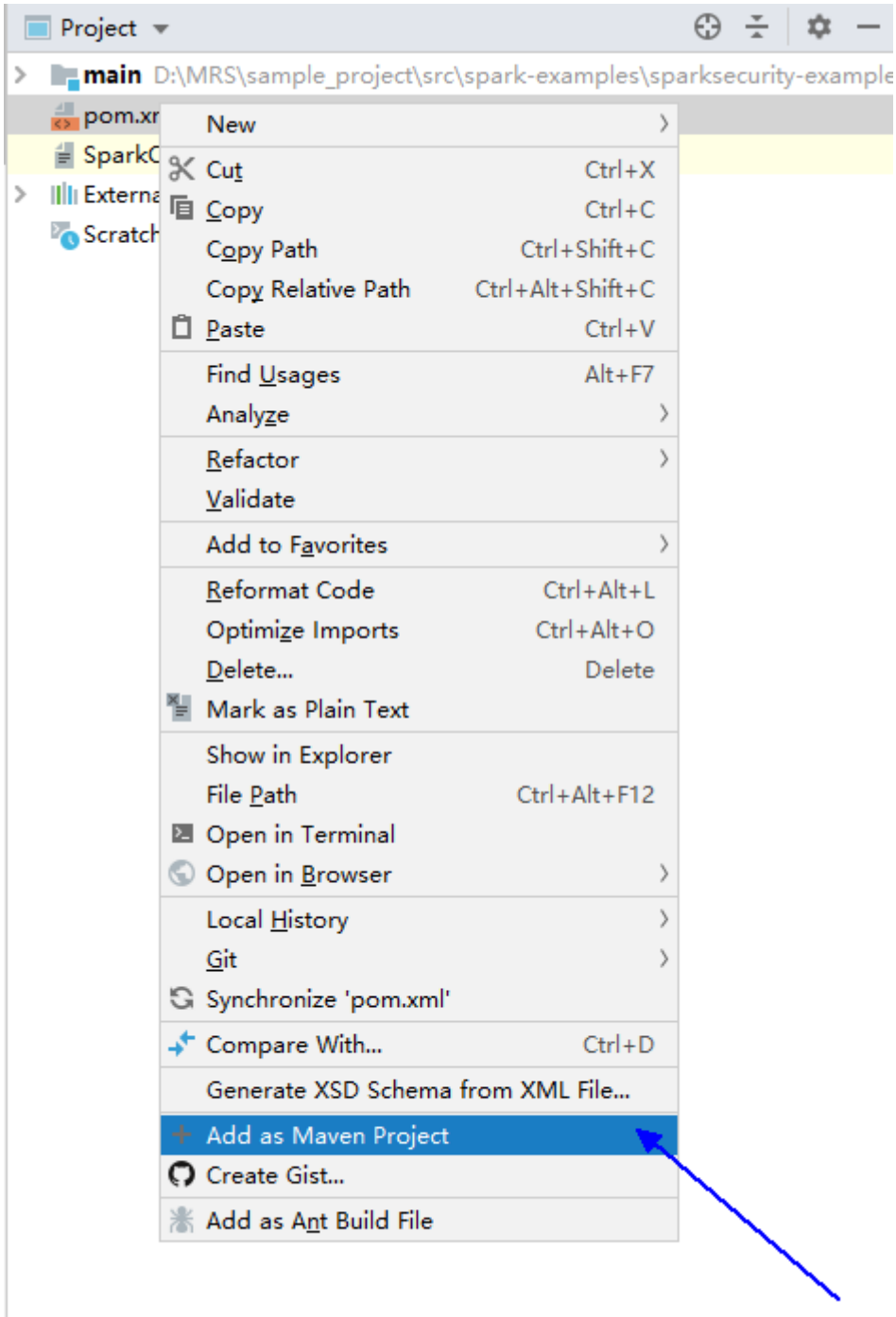
图 27-35 打开 Maven 工具窗口



若项目未通过maven导入，需要执行以下操作：

右键选择单击样例代码项目中的pom文件，选择“Add as Maven Project”，添加Maven项目。

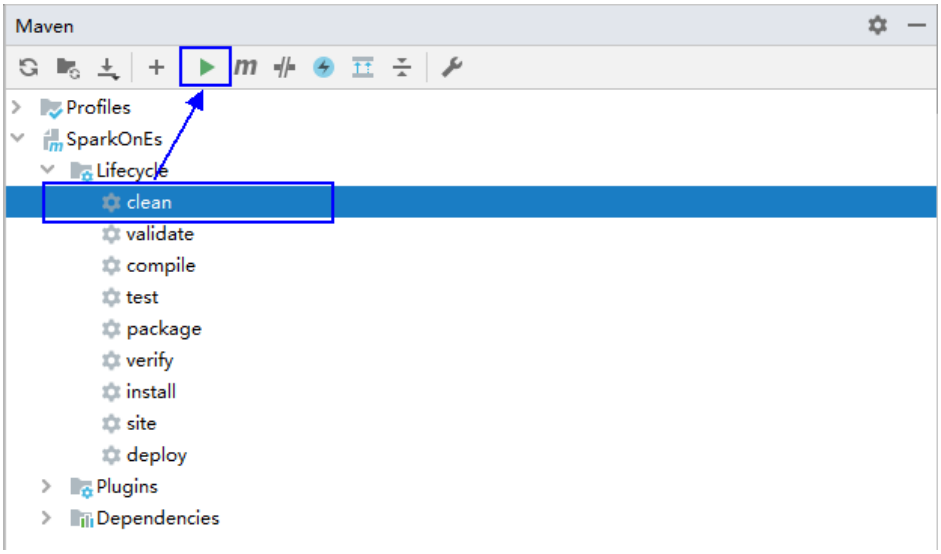
图 27-36 添加 Maven 项目



步骤2 通过Maven生成Jar包。

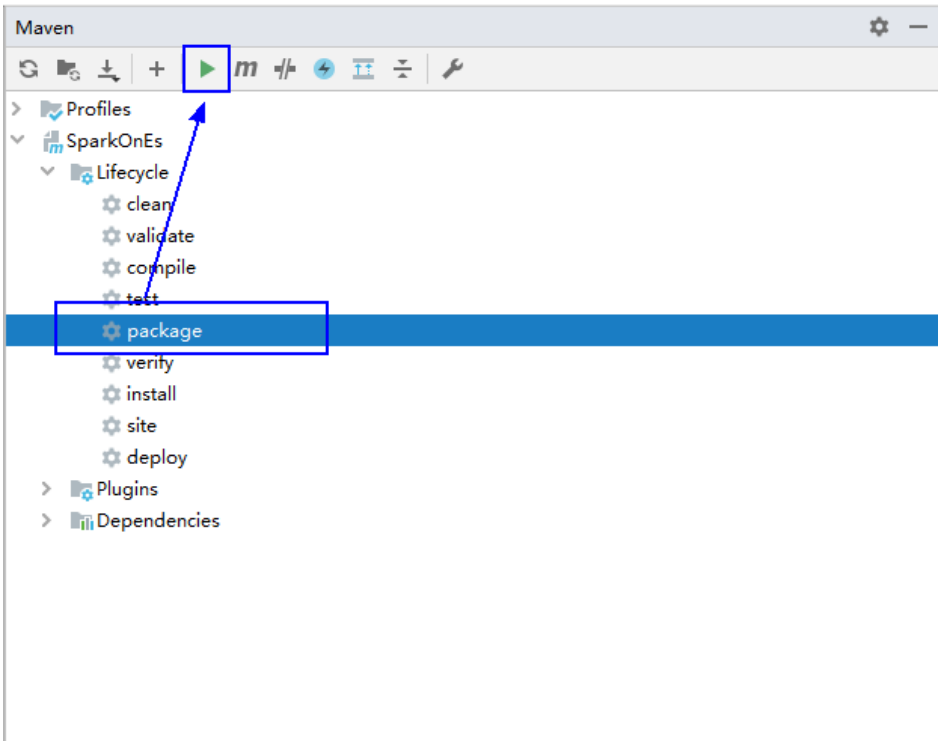
1. 在Maven工具窗口，选择clean生命周期，执行Maven构建过程。

图 27-37 选择 clean 生命周期，执行 Maven 构建过程



2. 在Maven工具窗口，选择package生命周期，执行Maven构建过程。

图 27-38 选择 package 生命周期，执行 Maven 构建过程



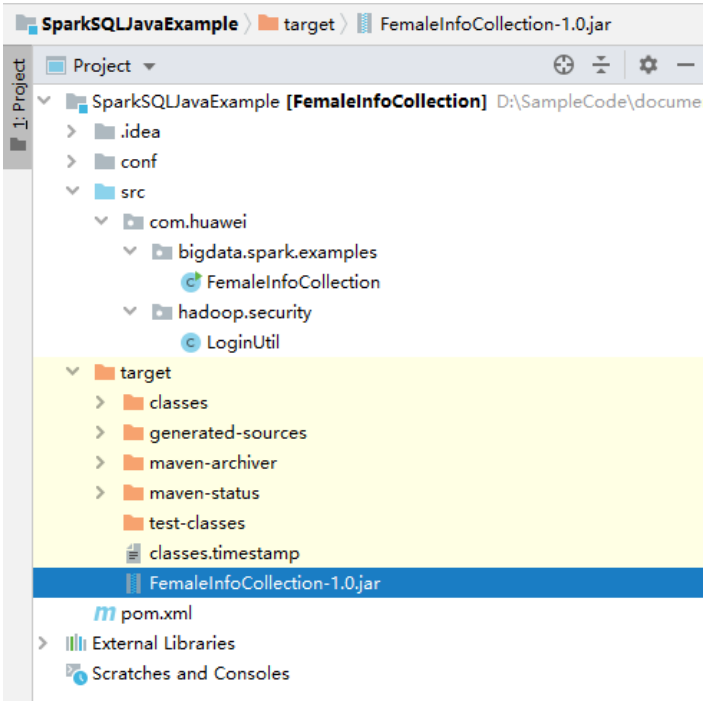
在Run:中出现下面提示，则说明打包成功。

图 27-39 打包成功提示

```
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ FemaleInfoCollection ---
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ FemaleInfoCollection ---
[INFO] Building jar: D:\SampleCode\document\VT\code\sparksecurity-examples\SparkSQLJavaExample\target\FemaleInfoCollection-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  19.427 s
[INFO] Finished at: 2020-09-21T11:17:31+08:00
[INFO] ------------------------------------------------------------------------
```

3. 您可以从项目目录下的target文件夹中获取到Jar包。

图 27-40 获取 jar 包



步骤3 将步骤2中生成的Jar包（如CollectFemaleInfo.jar）复制到Spark运行环境下（即Spark客户端），如“/opt/female”。运行Spark应用程序，具体样例程序可参考开发Spark应用。

注意

在Spark任务运行过程中禁止重启HDFS服务或者重启所有DataNode实例，否则可能会导致任务失败，并可能导致JobHistory部分数据丢失。

----结束

27.6.2.2 在 Linux 环境中查看 Spark 程序调测结果

操作场景

Spark应用程序运行完成后，可通过如下方式查看应用程序的运行情况。

- 通过运行结果数据查看应用程序运行情况。

- 登录Spark WebUI查看应用程序运行情况。
- 通过Spark日志获取应用程序运行情况。

操作步骤

- **查看Spark应用运行结果数据。**

  结果数据存储路径和格式已经由Spark应用程序指定，可通过指定文件获取。

- **查看Spark应用程序运行情况。**

  Spark主要有两个Web页面。

  - Spark UI页面，用于展示正在执行的应用的运行情况。

    页面主要包括了Jobs、Stages、Storage、Environment和Executors五个部分。Streaming应用会多一个Streaming标签页。

    页面入口：在YARN的Web UI界面，查找到对应的Spark应用程序。单击应用信息的最后一列“ApplicationMaster”，即可进入SparkUI页面。

  - History Server页面，用于展示已经完成的和未完成的Spark应用的运行情况。

    页面包括了应用ID、应用名称、开始时间、结束时间、执行时间、所属用户等信息。单击应用ID，页面将跳转到该应用的SparkUI页面。

- **查看Spark日志获取应用运行情况。**

  您可以查看Spark日志了解应用运行情况，并根据日志信息调整应用程序。相关日志信息可参考**Spark2x日志介绍**。

27.7 Spark 应用开发常见问题

27.7.1 Spark 常用 API 介绍

27.7.1.1 Spark Java API 接口介绍

由于Spark开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

Spark Core 常用接口

Spark主要使用到如下这几个类：

- JavaSparkContext：是Spark的对外接口，负责向调用该类的Java应用提供Spark的各种功能，如连接Spark集群，创建RDD，累积量和广播量等。它的作用相当于一个容器。
- SparkConf：Spark应用配置类，如设置应用名称，执行模式，executor内存等。
- JavaRDD：用于在java应用中定义JavaRDD的类，功能类似于scala中的RDD(Resilient Distributed Dataset)类。
- JavaPairRDD：表示key-value形式的JavaRDD类。提供的方法有groupByKey，reduceByKey等。
- Broadcast：广播变量类。广播变量允许保留一个只读的变量，缓存在每一台机器上，而非每个任务保存一份复制。

- StorageLevel：数据存储级别。有内存（MEMORY_ONLY），磁盘（DISK_ONLY），内存+磁盘（MEMORY_AND_DISK）等。

JavaRDD支持两种类型的操作：Transformation和Action，这两种类型的常用方法如表27-9和表27-10。

表 27-9 Transformation

方法	说明
<R> JavaRDD<R> map(Function<T,R> f)	对RDD中的每个element使用Function。
JavaRDD<T> filter(Function<T,Boolean> f)	对RDD中所有元素调用Function，返回为true的元素。
<U> JavaRDD<U> flatMap(FlatMapFunction<T,U> f)	先对RDD所有元素调用Function，然后将结果扁平化。
JavaRDD<T> sample(boolean withReplacement, double fraction, long seed)	抽样。
JavaRDD<T> distinct(int numPartitions)	去除重复元素。
JavaPairRDD<K,Iterable<V>> groupByKey(int numPartitions)	返回(K,Seq[V])，将key相同的value组成一个集合。
JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> func, int numPartitions)	对key相同的value调用Function。
JavaPairRDD<K,V> sortByKey(boolean ascending, int numPartitions)	按照key来进行排序，是升序还是降序，ascending是boolean类型。
JavaPairRDD<K,scala.Tuple2<V,W>> join(JavaPairRDD<K,W> other)	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,(V,W))的dataset，numTasks为并发的任务数。
JavaPairRDD<K,scala.Tuple2<Iterable<V>,Iterable<W>>> cogroup(JavaPairRDD<K,W> other, int numPartitions)	当有两个KV的dataset(K,V)和(K,W)，返回的是<K,scala.Tuple2<Iterable<V>,Iterable<W>>>的dataset，numTasks为并发的任务数。

方法	说明
JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)	返回该RDD与其它RDD的笛卡尔积。

表 27-10 Action

方法	说明
T reduce(Function2<T,T,T> f)	对RDD中的元素调用Function2。
java.util.List<T> collect()	返回包含RDD中所有元素的一个数组。
long count()	返回的是dataset中的element的个数。
T first()	返回的是dataset中的第一个元素。
java.util.List<T> take(int num)	返回前n个elements。
java.util.List<T> takeSample(boolean withReplacement, int num, long seed)	对dataset随机抽样，返回有num个元素组成的数组。withReplacement表示是否使用replacement。
void saveAsTextFile(String path, Class<? extends org.apache.hadoop.io.compress.CompressionCodec> codec)	把dataset写到一个text file、hdfs、或者hdfs支持的文件系统中，spark把每条记录都转换为一行记录，然后写到file中。
java.util.Map<K,Object> countByKey()	对每个key出现的次数做统计。
void foreach(VoidFunction<T> f)	在数据集的每一个元素上，运行函数func。
java.util.Map<T,Long> countByValue()	对RDD中每个元素出现的次数进行统计。

表 27-11 Spark Core 新增接口

API	说明
public java.util.concurrent.atomic.AtomicBoolean isSparkContextDown()	该接口可判断sparkContext是否已完全stop，初始值为false。 若接口值为true，则代表sparkContext已完全stop。 若接口值为false，则代表sparkContext没有完成stop。 例如：用户根据jsc.sc().isSparkContextDown().get() == true 可判断sparkContext已完全stop。

Spark Streaming 常用接口

Spark Streaming中常见的类有：

- JavaStreamingContext：是Spark Streaming功能的主入口，负责提供创建DStreams的方法，入参中需要设置批次的时间间隔。
- JavaDStream：是一种代表RDDs连续序列的数据类型，代表连续数据流。
- JavaPairDStream：KV DStream的接口，提供reduceByKey和join等操作。
- JavaReceiverInputDStream<T>：定义任何从网络接受数据的输入流。

Spark Streaming的常见方法与Spark Core类似，下表罗列了Spark Streaming特有的一些方法。

表 27-12 Spark Streaming 方法

方法	说明
JavaReceiverInputDStream<java.lang.String> socketStream(java.lang.String hostname,int port)	创建一个输入流，通过TCP socket从对应的hostname和端口接受数据。接受的字节被解析为UTF8格式。默认的存储级别为Memory+Disk。
JavaDStream<java.lang.String> textFileStream(java.lang.String directory)	入参directory为HDFS目录，该方法创建一个输入流检测可兼容Hadoop文件系统的新文件，并且读取为文本文件。
void start()	启动Spark Streaming计算。
void awaitTermination()	当前进程等待终止，如Ctrl+C等。
void stop()	终止Spark Streaming计算。

方法	说明
<T> JavaDStream<T> transform(java.util.List<JavaDStream<?>> dstreams,Function2<java.util.List<JavaRDD<?>>,Time,JavaRDD<T>> transformFunc)	对每个RDD进行function操作，得到一个新的DStream。这个函数中JavaRDDs的顺序和list中对应的DStreams保持一致。
<T> JavaDStream<T> union(JavaDStream<T> first,java.util.List<JavaDStream<T>> rest)	从多个具备相同类型和滑动时间的DStream中创建统一的DStream。

表 27-13 Spark Streaming 增强特性接口

方法	说明
JAVADStreamKafkaWriter.writeToKafka()	支持将DStream中的数据批量写入到Kafka。
JAVADStreamKafkaWriter.writeToKafkaBySingle()	支持将DStream中的数据逐条写入到Kafka。

Spark SQL 常用接口

Spark SQL中重要的类有：

- SQLContext：是Spark SQL功能和DataFrame的主入口。
- DataFrame：是一个以命名列方式组织的分布式数据集
- DataFrameReader：从外部存储系统加载DataFrame的接口。
- DataFrameStatFunctions：实现DataFrame的统计功能。
- UserDefinedFunction：用户自定义的函数。

常见的Actions方法有：

表 27-14 Spark SQL 方法介绍

方法	说明
Row[] collect()	返回一个数组，包含DataFrame的所有列。
long count()	返回DataFrame的行数。
DataFrame describe(java.lang.String... cols)	计算统计信息，包含计数，平均值，标准差，最小值和最大值。

方法	说明
Row first()	返回第一行。
Row[] head(int n)	返回前n行。
void show()	用表格形式显示DataFrame的前20行。
Row[] take(int n)	返回DataFrame中的前n行。

表 27-15 基本的 DataFrame Functions 介绍

方法	说明
void explain(boolean extended)	打印出SQL语句的逻辑计划和物理计划。
void printSchema()	打印schema信息到控制台。
registerTempTable	将DataFrame注册为一张临时表，其周期和SQLContext绑定在一起。
DataFrame toDF(java.lang.String... colNames)	返回一个列重命名的DataFrame。
DataFrame sort(java.lang.String sortCol,java.lang.String... sortCols)	根据不同的列，按照升序或者降序排序。
GroupedData rollup(Column... cols)	对当前的DataFrame特定列进行多维度的回滚操作。

27.7.1.2 Spark Scala API 接口介绍

由于Spark开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

Spark Core 常用接口

Spark主要使用到如下这几个类：

- SparkContext：是Spark的对外接口，负责向调用该类的scala应用提供Spark的各种功能，如连接Spark集群，创建RDD等。
- SparkConf：Spark应用配置类，如设置应用名称，执行模式，executor内存等。
- RDD（Resilient Distributed Dataset）：用于在Spark应用程序中定义RDD的类，该类提供数据集的操作方法，如map，filter。
- PairRDDFunctions：为key-value对的RDD数据提供运算操作，如groupByKey。
- Broadcast：广播变量类。广播变量允许保留一个只读的变量，缓存在每一台机器上，而非每个任务保存一份复制。

- StorageLevel：数据存储级别。有内存（MEMORY_ONLY），磁盘（DISK_ONLY），内存+磁盘（MEMORY_AND_DISK）等。

RDD上支持两种类型的操作：Transformation和Action，这两种类型的常用方法如表27-16和表27-17所示。

表 27-16 Transformation

方法	说明
map[U](f: (T) => U): RDD[U]	对调用map的RDD数据集中的每个element都使用f方法，生成新的RDD。
filter(f: (T) => Boolean): RDD[T]	对RDD中所有元素调用f方法，生成将满足条件数据集以RDD形式返回。
flatMap[U](f: (T) => TraversableOnce[U])(implicit arg0: ClassTag[U]): RDD[U]	先对RDD所有元素调用f方法，然后将结果扁平化，生成新的RDD。
sample(withReplacement: Boolean, fraction: Double, seed: Long = Utils.random.nextLong): RDD[T]	抽样，返回RDD一个子集。
union(other: RDD[T]): RDD[T]	返回一个新的RDD，包含源RDD和给定RDD的元素的集合。
distinct([numPartitions: Int]): RDD[T]	去除重复元素，生成新的RDD。
groupByKey(): RDD[(K, Iterable[V])]	返回(K,Iterable[V])，将key相同的value组成一个集合。
reduceByKey(func: (V, V) => V[, numPartitions: Int]): RDD[(K, V)]	对key相同的value调用func。
sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length): RDD[(K, V)]	按照key来进行排序，是升序还是降序，ascending是boolean类型。
join[W](other: RDD[(K, W)][, numPartitions: Int]): RDD[(K, (V, W))]	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,(V,W))的dataset，numPartitions为并发的任务数。

方法	说明
cogroup[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W]))]	将当有两个key-value对的dataset(K,V)和(K,W)，返回的是(K, (Iterable[V], Iterable[W]))的dataset，numPartitions为并发的任务数。
cartesian[U](other: RDD[U])(implicit arg0: ClassTag[U]): RDD[(T, U)]	返回该RDD与其它RDD的笛卡尔积。

表 27-17 Action

API	说明
reduce(f: (T, T) => T):	对RDD中的元素调用f。
collect(): Array[T]	返回包含RDD中所有元素的一个数组。
count(): Long	返回的是dataset中的element的个数。
first(): T	返回的是dataset中的第一个元素。
take(num: Int): Array[T]	返回前n个elements。
takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong): Array[T]	takeSample(withReplacement，num，seed)对dataset随机抽样，返回有num个元素组成的数组。withReplacement表示是否使用replacement。
saveAsTextFile(path: String): Unit	把dataset写到一个text file、HDFS或者HDFS支持的文件系统中，spark把每条记录都转换为一行记录，然后写到file中。
saveAsSequenceFile(path: String, codec: Option[Class[_ <: CompressionCodec]] = None): Unit	只能用在key-value对上，然后生成SequenceFile写到本地或者hadoop文件系统。
countByKey(): Map[K, Long]	对每个key出现的次数做统计。
foreach(func: (T) => Unit): Unit	在数据集的每一个元素上，运行函数func。
countByValue()(implicit ord: Ordering[T] = null): Map[T, Long]	对RDD中每个元素出现的次数进行统计。

表 27-18 Spark Core 新增接口

API	说明
isSparkContextDown:AtomicBoolean	该接口可判断sparkContext是否已完全stop，初始值为false。 若接口值为true，则代表sparkContext已完全stop。 若接口值为false，则代表sparkContext没有完成stop。 例如：用户根据sc.isSparkContextDown.get() == true可判断sparkContext已完全stop。

Spark Streaming 常用接口

Spark Streaming中常见的类有：

- StreamingContext：是Spark Streaming功能的主入口，负责提供创建DStreams的方法，入参中需要设置批次的时间间隔。
- dstream.DStream：是一种代表RDDs连续序列的数据类型，代表连续数据流。
- dstream.PariDStreamFunctions：键值对的DStream，常见的操作如groupByKey和reduceByKey。

  对应的Spark Streaming的JAVA API是JavaStreamingContext，JavaDStream和JavaPairDStream。

Spark Streaming的常见方法与Spark Core类似，下表罗列了Spark Streaming特有的一些方法。

表 27-19 Spark Streaming 方法介绍

方法	说明
socketTextStream(hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[String]	从TCP源主机：端口创建一个输入流。
start():Unit	启动Spark Streaming计算。
awaitTermination(timeout: long):Unit	当前进程等待终止，如Ctrl+C等。
stop(stopSparkContext: Boolean, stopGracefully: Boolean): Unit	终止Spark Streaming计算。

方法	说明
transform[T] (dstreams: Seq[DStream[_]], transformFunc: (Seq[RDD[_]], Time) ? RDD[T]) (implicit arg0: ClassTag[T]): DStream[T]	对每一个RDD应用function操作得到一个新的DStream。
UpdateStateByKey(func)	更新DStream的状态。使用此方法，需要定义状态和状态更新函数。
window(windowLength, slideInterval)	根据源DStream的窗口批次计算得到一个新的DStream。
countByWindow(windowLength, slideInterval)	返回流中滑动窗口元素的个数。
reduceByWindow(func, windowLength, slideInterval)	当调用在DStream的KV对上，返回一个新的DStream的KV对，其中每个Key的Value根据滑动窗口中批次的reduce函数聚合得到。
join(otherStream, [numTasks])	实现不同的Spark Streaming之间做合并操作。
DStreamKafkaWriter.writeToKafka()	支持将DStream中的数据批量写入到Kafka。
DStreamKafkaWriter.writeToKafkaBySingle()	支持将DStream中的数据逐条写入到Kafka。

表 27-20 Spark Streaming 增强特性接口

方法	说明
DStreamKafkaWriter.writeToKafka()	支持将DStream中的数据批量写入到Kafka。
DStreamKafkaWriter.writeToKafkaBySingle()	支持将DStream中的数据逐条写入到Kafka。

SparkSQL 常用接口

Spark SQL中常用的类有：

- SQLContext：是Spark SQL功能和DataFrame的主入口。

- DataFrame：是一个以命名列方式组织的分布式数据集。
- HiveContext：获取存储在Hive中数据的主入口。

表 27-21 常用的 Actions 方法

方法	说明
collect(): Array[Row]	返回一个数组，包含DataFrame的所有列。
count(): Long	返回DataFrame中的行数。
describe(cols: String*): DataFrame	计算统计信息，包含计数，平均值，标准差，最小值和最大值。
first(): Row	返回第一行。
Head(n:Int): Row	返回前n行。
show(numRows: Int, truncate: Boolean): Unit	用表格形式显示DataFrame。
take(n:Int): Array[Row]	返回DataFrame中的前n行。

表 27-22 基本的 DataFrame Functions

方法	说明
explain(): Unit	打印出SQL语句的逻辑计划和物理计划。
printSchema(): Unit	打印schema信息到控制台。
registerTempTable(tableName: String): Unit	将DataFrame注册为一张临时表，其周期和SQLContext绑定在一起。
toDF(colNames: String*): DataFrame	返回一个列重命名的DataFrame。

27.7.1.3 Spark Python API 接口介绍

由于Spark开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

Spark Core 常用接口

Spark主要使用到如下这几个类：

- pyspark.SparkContext：是Spark的对外接口。负责向调用该类的python应用提供Spark的各种功能，如连接Spark集群、创建RDD、广播变量等。
- pyspark.SparkConf：Spark应用配置类。如设置应用名称，执行模式，executor内存等。

- pyspark.RDD（Resilient Distributed Dataset）：用于在Spark应用程序中定义RDD的类，该类提供数据集的操作方法，如map，filter。
- pyspark.Broadcast：广播变量类。广播变量允许保留一个只读的变量，缓存在每一台机器上，而非每个任务保存一份复制。
- pyspark.StorageLevel： 数据存储级别。有内存（MEMORY_ONLY），磁盘（DISK_ONLY），内存+磁盘（MEMORY_AND_DISK）等。
- pyspark.sql.SQLContext：是SparkSQL功能的主入口。可用于创建DataFrame，注册DataFrame为一张表，表上执行SQL等。
- pyspark.sql.DataFrame：分布式数据集。DataFrame等效于SparkSQL中的关系表，可被SQLContext中的方法创建。
- pyspark.sql.DataFrameNaFunctions：DataFrame中处理数据缺失的函数。
- pyspark.sql.DataFrameStatFunctions：DataFrame中统计功能的函数，可以计算列之间的方差，样本协方差等。

RDD上支持两种类型的操作：transformation和action，这两种类型的常用方法如表27-23和表27-24。

表 27-23 Transformation

方法	说明
map(f, preservesPartitioning=False)	对调用map的RDD数据集中的每个element都使用Func，生成新的RDD。
filter(f)	对RDD中所有元素调用Func，生成将满足条件数据集以RDD形式返回。
flatMap(f, preservesPartitioning=False)	先对RDD所有元素调用Func，然后将结果扁平化，生成新的RDD。
sample(withReplacement, fraction, seed=None)	抽样，返回RDD一个子集。
union(rdds)	返回一个新的RDD，包含源RDD和给定RDD的元素的集合。
distinct([numPartitions: Int]): RDD[T]	去除重复元素，生成新的RDD。
groupByKey(): RDD[(K, Iterable[V])]	返回(K,Iterable[V])，将key相同的value组成一个集合。
reduceByKey(func, numPartitions=None)	对key相同的value调用Func。
sortByKey(ascending=True, numPartitions=None, keyfunc=function <lambda>)	按照key来进行排序，是升序还是降序，ascending是boolean类型。

方法	说明
join(other, numPartitions)	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,(V,W))的dataset,numPartitions为并发的任务数。
cogroup(other, numPartitions)	将当有两个key-value对的dataset(K,V)和(K,W)，返回的是(K, (Iterable[V], Iterable[W]))的dataset,numPartitions为并发的任务数。
cartesian(other)	返回该RDD与其它RDD的笛卡尔积。

表 27-24 Action

API	说明
reduce(f)	对RDD中的元素调用Func。
collect()	返回包含RDD中所有元素的一个数组。
count()	返回的是dataset中的element的个数。
first()	返回的是dataset中的第一个元素。
take(num)	返回前num个elements。
takeSample(withReplacement, num, seed)	takeSample(withReplacement，num，seed)对dataset随机抽样，返回有num个元素组成的数组。withReplacement表示是否使用replacement。
saveAsTextFile(path, compressionCodecClass)	把dataset写到一个text file、HDFS或者HDFS支持的文件系统中，spark把每条记录都转换为一行记录，然后写到file中。
saveAsSequenceFile(path, compressionCodecClass=None)	只能用在key-value对上，然后生成SequenceFile写到本地或者hadoop文件系统。
countByKey()	对每个key出现的次数做统计。
foreach(func)	在数据集的每一个元素上，运行函数。
countByValue()	对RDD中每个不同value出现的次数进行统计。

Spark Streaming 常用接口

Spark Streaming中常见的类有：

- pyspark.streaming.StreamingContext：是Spark Streaming功能的主入口，负责提供创建DStreams的方法，入参中需要设置批次的时间间隔。
- pyspark.streaming.DStream：是一种代表RDDs连续序列的数据类型，代表连续数据流。

- dsteam.PariDStreamFunctions：键值对的DStream，常见的操作如groupByKey和reduceByKey。

  对应的Spark Streaming的JAVA API是JavaStreamingContext，JavaDStream和JavaPairDStream。

Spark Streaming的常见方法与Spark Core类似，下表罗列了Spark Streaming特有的一些方法。

表 27-25 Spark Streaming 常用接口介绍

方法	说明
socketTextStream(hostname, port, storageLevel)	从TCP源主机：端口创建一个输入流。
start()	启动Spark Streaming计算。
awaitTermination(timeout)	当前进程等待终止，如Ctrl+C等。
stop(stopSparkContext, stopGraceFully)	终止Spark Streaming计算，stopSparkContext用于判断是否需要终止相关的SparkContext，StopGracefully用于判断是否需要等待所有接受到的数据处理完成。
UpdateStateByKey(func)	更新DStream的状态。使用此方法，需要定义State和状态更新函数。
window(windowLength, slideInterval)	根据源DStream的窗口批次计算得到一个新的DStream。
countByWindow(windowLength, slideInterval)	返回流中滑动窗口元素的个数。
reduceByWindow(func, windowLength, slideInterval)	当调用在DStream的KV对上，返回一个新的DStream的KV对，其中每个Key的Value根据滑动窗口中批次的reduce函数聚合得到。
join(other,numPartitions)	实现不同的Spark Streaming之间做合并操作。

SparkSQL 常用接口

Spark SQL中在Python中重要的类有：

- pyspark.sql.SQLContext：是Spark SQL功能和DataFrame的主入口。
- pyspark.sql.DataFrame：是一个以命名列方式组织的分布式数据集。
- pyspark.sql.HiveContext：获取存储在Hive中数据的主入口。
- pyspark.sql.DataFrameStatFunctions：统计功能中一些函数。
- pyspark.sql.functions：DataFrame中内嵌的函数。
- pyspark.sql.Window：sql中提供窗口功能。

表 27-26 Spark SQL 常用的 Action

方法	说明
collect()	返回一个数组，包含DataFrame的所有列。
count()	返回DataFrame中的行数。
describe()	计算统计信息，包含计数，平均值，标准差，最小值和最大值。
first()	返回第一行。
head(n)	返回前n行。
show()	用表格形式显示DataFrame。
take(num)	返回DataFrame中的前num行。

表 27-27 基本的 DataFrame Functions

方法	说明
explain()	打印出SQL语句的逻辑计划和物理计划。
printSchema()	打印schema信息到控制台。
registerTempTable(name)	将DataFrame注册为一张临时表，命名为name，其周期和SQLContext绑定在一起。
toDF()	返回一个列重命名的DataFrame。

27.7.1.4 Spark REST API 接口介绍

功能简介

Spark的REST API以JSON格式展现Web UI的一些指标，提供用户一种更简单的方法去创建新的展示和监控的工具，并且支持查询正在运行的app和已经结束的app的相关信息。开源的Spark REST接口支持对Jobs、Stages、Storage、Environment和Executors的信息进行查询，FusionInsight版本中添加了查询SQL、JDBC Server和Streaming的信息的REST接口。开源REST接口完整和详细的描述请参考官网上的文档以了解其使用方法：**https://spark.apache.org/docs/3.1.1/monitoring.html#rest-api**。

准备运行环境

安装客户端。在节点上安装客户端，如安装到“/opt/client”目录。

REST 接口

通过以下命令可跳过REST接口过滤器获取相应的应用信息。

须知

- 安全模式下，JobHistory仅支持https协议，故在如下命令的url中请使用https协议。
- 安全模式下，需要设置spark.ui.customErrorPage=false并重启spark2x服务（JobHistory2x、JDBCServer2x和SparkResource2x三个实例对应的参数都需要修改）。

说明

升级更新节点环境上的curl版本。具体curl版本升级方法如下：

1. 下载curl安装包（**http://curl.haxx.se/download/**）。
2. 使用如下命令进行安装包解压：
   tar -xzvf curl-x.x.x.tar.gz
3. 使用如下命令覆盖安装：
   cd curl-x.x.x
   ./configure
   make
   make install
4. 使用如下命令更新curl的动态链接库：
   ldconfig
5. 安装成功后，重新登录节点环境，使用如下命令查看curl版本是否更新成功：
   curl --version

- 获取JobHistory中所有应用信息：
  - 命令：
    ```
    curl -k -i --negotiate -u: "https://192.168.227.16:18080/api/v1/applications"
    ```
    其中192.168.227.16为JobHistory节点的业务IP，18080为JobHistory的端口号。
  - 结果：
    ```
    [ {
      "id" : "application_1517290848707_0008",
      "name" : "Spark Pi",
      "attempts" : [ {
        "startTime" : "2018-01-30T15:05:37.433CST",
        "endTime" : "2018-01-30T15:06:04.625CST",
        "lastUpdated" : "2018-01-30T15:06:04.848CST",
        "duration" : 27192,
        "sparkUser" : "sparkuser",
        "completed" : true,
        "startTimeEpoch" : 1517295937433,
        "endTimeEpoch" : 1517295964625,
        "lastUpdatedEpoch" : 1517295964848
      } ]
    }, {
      "
    id" : "application_1517290848707_0145",
      "name" : "Spark shell",
      "attempts" : [ {
        "startTime" : "2018-01-31T15:20:31.286CST",
        "endTime" : "1970-01-01T07:59:59.999CST",
        "lastUpdated" : "2018-01-31T15:20:47.086CST",
        "duration" : 0,
        "sparkUser" : "admintest",
        "completed" : false,
        "startTimeEpoch" : 1517383231286,
    ```

```
  "endTimeEpoch" : -1,
  "lastUpdatedEpoch" : 1517383247086
 } ]
}]
```

- 结果分析：

  通过这个命令，可以查询当前集群中所有的Spark应用（包括正在运行的应用和已经完成的应用），每个应用的信息如下**表27-28**。

  表 27-28 应用常用信息

参数	描述
id	应用的ID
name	应用的Name
attempts	应用的尝试，包含了开始时间、结束时间、执行用户、是否完成等信息

- 获取JobHistory中某个应用的信息：
  - 命令：

    ```
    curl -k -i --negotiate -u: "https://192.168.227.16:18080/api/v1/applications/application_1517290848707_0008"
    ```

    其中192.168.227.16为JobHistory节点的业务IP，18080为JobHistory的端口号，application_1517290848707_0008为应用的id。
  - 结果：

    ```
    {
     "id" : "application_1517290848707_0008",
     "name" : "Spark Pi",
     "attempts" : [ {
       "startTime" : "2018-01-30T15:05:37.433CST",
       "endTime" : "2018-01-30T15:06:04.625CST",
       "lastUpdated" : "2018-01-30T15:06:04.848CST",
       "duration" : 27192,
       "sparkUser" : "sparkuser",
       "completed" : true,
       "startTimeEpoch" : 1517295937433,
       "endTimeEpoch" : 1517295964625,
       "lastUpdatedEpoch" : 1517295964848
     } ]
    }
    ```

  - 结果分析：

    通过这个命令，可以查询某个Spark应用的信息，显示的信息如**表27-28**所示。
- 获取正在执行的某个应用的Executor信息：
  - 针对alive executor命令：

    ```
    curl -k -i --negotiate -u: "https://192.168.169.84:8090/proxy/application_1478570725074_0046/api/v1/applications/application_1478570725074_0046/executors"
    ```

  - 针对全部executor（alive&dead）命令：

    ```
    curl -k -i --negotiate -u: "https://192.168.169.84:8090/proxy/application_1478570725074_0046/api/v1/applications/application_1478570725074_0046/allexecutors"
    ```

    其中192.168.169.84为ResourceManager主节点的业务IP，8090为ResourceManager的端口号，application_1478570725074_0046为在YARN中的应用ID。

- 结果：

```
[{
  "id" : "driver",
  "hostPort" : "192.168.169.84:23886",
  "isActive" : true,
  "rddBlocks" : 0,
  "memoryUsed" : 0,
  "diskUsed" : 0,
  "activeTasks" : 0,
  "failedTasks" : 0,
  "completedTasks" : 0,
  "totalTasks" : 0,
  "totalDuration" : 0,
  "totalInputBytes" : 0,
  "totalShuffleRead" : 0,
  "totalShuffleWrite" : 0,
  "maxMemory" : 278019440,
  "executorLogs" : { }
}, {
  "id" : "1",
  "hostPort" : "192.168.169.84:23902",
  "isActive" : true,
  "rddBlocks" : 0,
  "memoryUsed" : 0,
  "diskUsed" : 0,
  "totalCores" : 1,
  "maxTasks" : 1,
  "activeTasks" : 0,
  "failedTasks" : 0,
  "completedTasks" : 0,
  "totalTasks" : 0,
  "totalDuration" : 0,
  "totalGCTime" : 139,
  "totalInputBytes" : 0,
  "totalShuffleRead" : 0,
  "totalShuffleWrite" : 0,
  "maxMemory" : 555755765,
  "executorLogs" : {
    "stdout" : "https://XTJ-224:8044/node/containerlogs/
container_1478570725074_0049_01_000002/admin/stdout?start=-4096",
    "stderr" : "https://XTJ-224:8044/node/containerlogs/
container_1478570725074_0049_01_000002/admin/stderr?start=-4096"
  }
} ]
```

- 结果分析：

  通过这个命令，可以查询当前应用的所有Executor信息（包括Driver），每个Executor的信息包含如下表27-29所示的常用信息。

  表 27-29 Executor 常用信息

参数	描述
id	Executor的ID
hostPort	Executor所在节点的ip：端口
executorLogs	Executor的日志查看路径

REST API 增强

- SQL相关的命令：获取所有SQL语句和执行时间最长的SQL语句
  - SparkUI命令：


```
curl -k -i --negotiate -u: "https://192.168.195.232:8090/proxy/
application_1476947670799_0053/api/v1/applications/application_1476947670799_0053/SQL"
```

其中192.168.195.232为ResourceManager主节点的业务IP，8090为ResourceManager的端口号，application_1476947670799_0053为在YARN中的应用ID。

说明

可以在命令后的url路径增加相应的参数设置，搜索对应的SQL语句。

例如，查看100条sql语句：

```
curl -k -i --negotiate -u: "https://192.168.195.232:8090/proxy/
application_1476947670799_0053/api/v1/applications/application_1476947670799_0053/
SQL?limit=100"
```

查看正在运行的参数：

```
curl -k -i --negotiate -u: "https://192.168.195.232:8090/proxy/
application_1476947670799_0053/api/v1/applications/application_1476947670799_0053/
SQL?completed=false"
```

- JobHistory命令：

```
curl -k -i --negotiate -u: "https://192.168.227.16:18080/api/v1/applications/
application_1478570725074_0004/SQL"
```

其中192.168.227.16为JobHistory节点的业务IP，18080为JobHistory的端口号，application_1478570725074_0004为应用ID。

- 结果：

SparkUI命令和JobHistory命令的查询结果均为：

```
{
 "longestDurationOfCompletedSQL" : [ {
  "id" : 0,
  "status" : "COMPLETED",
  "description" : "getCallSite at SQLExecution.scala:48",
  "submissionTime" : "2016/11/08 15:39:00",
  "duration" : "2 s",
  "runningJobs" : [ ],
  "successedJobs" : [ 0 ],
  "failedJobs" : [ ]
 } ],
 "sqls" : [ {
  "id" : 0,
  "status" : "COMPLETED",
  "description" : "getCallSite at SQLExecution.scala:48",
  "submissionTime" : "2016/11/08 15:39:00",
  "duration" : "2 s",
  "runningJobs" : [ ],
  "successedJobs" : [ 0 ],
  "failedJobs" : [ ]
 }]
}
```

- 结果分析：

通过这个命令，可以查询当前应用的所有SQL语句的信息（即结果中“sqls”的部分），执行时间最长的SQL语句的信息（即结果中“longestDurationOfCompletedSQL”的部分）。每个SQL语句的信息如下**表27-30**。

表 27-30 SQL 的常用信息

参数	描述
id	SQL语句的ID

参数	描述
status	SQL语句的执行状态，有RUNNING、COMPLETED、FAILED三种
runningJobs	SQL语句产生的job中，正在执行的job列表
successedJobs	SQL语句产生的job中，执行成功的job列表
failedJobs	SQL语句产生的job中，执行失败的job列表

- JDBC Server相关的命令：获取连接数，正在执行的SQL数，所有session信息，所有SQL的信息
  - 命令：

    ```
    curl -k -i --negotiate -u: "https://192.168.195.232:8090/proxy/
    application_1476947670799_0053/api/v1/applications/application_1476947670799_0053/
    sqlserver"
    ```

    其中192.168.195.232为ResourceManager主节点的业务IP，8090为ResourceManager的端口号，application_1476947670799_0053为在YARN中的应用ID。
  - 结果：

    ```
    {
      "sessionNum" : 1,
      "runningSqlNum" : 0,
      "sessions" : [ {
        "user" : "spark",
        "ip" : "192.168.169.84",
        "sessionId" : "9dfec575-48b4-4187-876a-71711d3d7a97",
        "startTime" : "2016/10/29 15:21:10",
        "finishTime" : "",
        "duration" : "1 minute 50 seconds",
        "totalExecute" : 1
      } ],
      "sqls" : [ {
        "user" : "spark",
        "jobId" : [ ],
        "groupId" : "e49ff81a-230f-4892-a209-a48abea2d969",
        "startTime" : "2016/10/29 15:21:13",
        "finishTime" : "2016/10/29 15:21:14",
        "duration" : "555 ms",
        "statement" : "show tables",
        "state" : "FINISHED",
        "detail" : "== Parsed Logical Plan ==\nShowTablesCommand None\n\n== Analyzed Logical
    Plan ==\ntableName: string, isTemporary: boolean\nShowTablesCommand None\n\n== Cached
    Logical Plan ==\nShowTablesCommand None\n\n== Optimized Logical Plan ==
    \nShowTablesCommand None\n\n== Physical Plan ==\nExecutedCommand
    ShowTablesCommand None\n\nCode Generation: true"
      } ]
    }
    ```
  - 结果分析：

    通过这个命令，可以查询当前JDBC应用的session连接数，正在执行的SQL数，所有的session和SQL信息。每个session的信息如下**表27-31**，每个SQL的信息如下**表27-32**。

表 27-31 session 常用信息

参数	描述
user	该session连接的用户
ip	session所在的节点IP
sessionId	session的ID
startTime	session开始连接的时间
finishTime	session结束连接的时间
duration	session连接时长
totalExecute	在该session上执行的SQL数

表 27-32 sql 常用信息

参数	描述
user	SQL执行的用户
jobId	SQL语句包含的job id列表
groupId	SQL所在的group id
startTime	SQL开始时间
finishTime	SQL结束时间
duration	SQL执行时长
statement	对应的语句
detail	对应的逻辑计划，物理计划

- JDBC api增强通过beeline里面获取的executionID 取消当前正在执行的SQL
  - 命令：
    ```
    curl -k -i --negotiate -X PUT -u: "https://192.168.195.232:8090/proxy/application_1477722033672_0008/api/v1/applications/application_1477722033672_0008/cancel/execution?executionId=8"
    ```
  - 结果：

    取消executionId 执行序号为8的job任务。
  - 补充说明：

    spark-beeline里面执行SQL语句，如果该SQL语句产生spark任务，该SQL的executionId将会被打印在beeline里面，这个时候如果想取消这条sql的执行，可以用上述命令。
- Streaming相关的命令：获取平均输入频率，平均调度时延，平均执行时长，总时延平均值
  - 命令：
    ```
    curl -k -i --negotiate -u: "https://192.168.195.232:8090/proxy/application_1477722033672_0008/api/v1/applications/application_1477722033672_0008/streaming/statistics"
    ```


其中192.168.195.232为ResourceManager主节点的业务IP，8090为ResourceManager的端口号，application_1477722033672_0008为在YARN中的应用ID。

- 结果：

```
{
"startTime" : "2018-12-25T08:58:10.836GMT",
"batchDuration" : 1000,
"numReceivers" : 1,
"numActiveReceivers" : 1,
"numInactiveReceivers" : 0,
"numTotalCompletedBatches" : 373,
"numRetainedCompletedBatches" : 373,
"numActiveBatches" : 0,
"numProcessedRecords" : 1,
"numReceivedRecords" : 1,
"avgInputRate" : 0.002680965147453083,
"avgSchedulingDelay" : 14,
"avgProcessingTime" : 47,
"avgTotalDelay" : 62
}
```

- 结果分析：

  通过这个命令，可以查询当前Streaming应用的平均输入频率（events/sec），平均调度时延（ms），平均执行时长（ms），总时延平均值（ms）。

27.7.1.5 Spark client CLI 介绍

Spark CLI详细的使用方法参考官方网站的描述：**http://spark.apache.org/docs/3.1.1/quick-start.html**。

常用 CLI

Spark常用的CLI如下所示：

- **spark-shell**

  提供了一个简单学习API的方法，类似于交互式数据分析的工具。同时支持Scala和Python两种语言。在Spark目录下，执行***./bin/spark-shell***即可进入Scala交互式界面从HDFS中获取数据，再操作RDD。

  示例：一行代码可以实现统计一个文件中所有单词。

  scala> sc.textFile("hdfs://10.96.1.57:9000//wordcount_data.txt").flatMap(l => l.split(" ")).map(w => (w,1)).reduceByKey(_+_).collect()

  您可以直接在命令行中指定Keytab和Principal以获取认证，定期更新登录票据和授权tokens，避免认证过期。示例如下：

  spark-shell --principal spark2x/hadoop.*<系统域名>*@*<系统域名>* --keytab ${BIGDATA_HOME}/FusionInsight_Spark2x_8.1.0.1/install/FusionInsight-Spark2x-3.1.1/keytab/spark2x/SparkResource/spark2x.keytab --master yarn

- **spark-submit**

  用于提交Spark应用到Spark集群中运行，返回运行结果。需要指定class、master、jar包以及入参。

  示例：执行jar包中的GroupByTest例子，入参为4个，指定集群运行模式是local单核运行。

./bin/spark-submit --class org.apache.spark.examples.GroupByTest --master local[1] examples/jars/spark-examples_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar 6 10 10 3

您可以直接在命令行中指定Keytab和Principal以获取认证，定期更新登录票据和授权tokens，避免认证过期。示例如下：

spark-submit --class org.apache.spark.examples.GroupByTest --master yarn --principal spark2x/hadoop.*<系统域名>*@*<系统域名>* --keytab ${BIGDATA_HOME}/FusionInsight_Spark2x_8.1.0.1/install/FusionInsight-Spark2x-3.1.1/keytab/spark2x/SparkResource/spark2x.keytab examples/jars/spark-examples_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar 6 10 10 3

- **spark-sql**

  可用于local模式或者集群模式运行Hive元数据服务以及命令行查询。如果需要查看其逻辑计划，只需在SQL语句前面加上explain extended即可。

  示例：

  Select key from src group by key

  您可以直接在命令行中指定Keytab和Principal以获取认证，定期更新登录票据和授权tokens，避免认证过期。示例如下：

  spark-sql --principal spark2x/hadoop.*<系统域名>*@*<系统域名>* --keytab ${BIGDATA_HOME}/FusionInsight_Spark2x_8.1.0.1/install/FusionInsight-Spark2x-3.1.1/keytab/spark2x/SparkResource/spark2x.keytab --master yarn

- **run-example**

  用来运行或者调试Spark开源社区中的自带的example。

  示例：执行SparkPi。

  ./run-example SparkPi 100

27.7.1.6 Spark JDBCServer 接口介绍

简介

JDBCServer是Hive中的HiveServer2的另外一个实现，它底层使用了Spark SQL来处理SQL语句，从而比Hive拥有更高的性能。

JDBCServer是一个JDBC接口，用户可以通过JDBC连接JDBCServer来访问SparkSQL的数据。JDBCServer在启动的时候，会启动一个sparkSQL的应用程序，而通过JDBC连接进来的客户端共同分享这个sparkSQL应用程序的资源，也就是说不同的用户之间可以共享数据。JDBCServer启动时还会开启一个侦听器，等待JDBC客户端的连接和提交查询。所以，在配置JDBCServer的时候，至少要配置JDBCServer的主机名和端口，如果要使用hive数据的话，还要提供hive metastore的uris。

JDBCServer默认在安装节点上的22550端口起一个JDBC服务（通过参数hive.server2.thrift.port配置），可以通过Beeline或者JDBC客户端代码来连接它，从而执行SQL命令。

如果您需要了解JDBCServer的其他信息，请参见Spark官网：**http://spark.apache.org/docs/3.1.1/sql-programming-guide.html#distributed-sql-engine**。

Beeline

开源社区提供的Beeline连接方式，请参见：**https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients**。

为了解决Beeline两种场景下连接问题，增加了在Beeline连接中添加认证信息。在JDBC的URL中增添了user.keytab和user.principal这两个参数。当票据过期时，会自动读入客户端的登录信息，就可以重新获得连接。

用户不希望通过kinit命令进行票据认证，因为票据信息每隔24小时会过期。其中Keytab文件及principal信息请联系管理员获取，Beeline的连接样例如下所示：

sh CLIENT_HOME/spark/bin/beeline -u "jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;user.principal=spark2x/hadoop.*<系统域名>*@*<系统域名>*;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.*<系统域名>*@*<系统域名>*;"

说明

- 其中“<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>”是Zookeeper的URL，多个URL以逗号隔开。例如“192.168.81.37:2181,192.168.195.232:2181,192.168.169.84:2181”。
- 其中“sparkthriftserver2x”是Zookeeper上的目录，表示客户端从该目录下随机选择JDBCServer实例进行连接。

JDBC 客户端代码

通过JDBC客户端代码连接JDBCServer，来访问SparkSQL的数据。详细指导请参见**通过JDBC访问Spark SQL样例程序**。

增强特性

对比开源社区，华为还提供了两个增强特性，JDBCServer HA方案和设置JDBCServer连接的超时时间。

- JDBCServer的HA方案，多个JDBCServer主节点同时提供服务，当其中一个节点发生故障时，新的客户端连接会分配到其他主节点上，从而保障无间断为集群提供服务。Beeline和JDBC客户端代码两种连接方式的操作相同。
- 设置客户端与JDBCServer连接的超时时间。
  - Beeline

    在网络拥塞的情况下，这个特性可以避免beeline由于无限等待服务端的返回而挂起。使用方式如下：

    启动beeline时，在后面追加“--socketTimeOut=n”，其中“n”表示等待服务返回的超时时长，单位为秒，默认为“0”（表示永不超时）。建议根据业务场景，设置为业务所能容忍的最大等待时长。
  - JDBC客户端代码

    在网络拥塞的情况下，这个特性可以避免客户端由于无限等待服务端的返回而挂起。使用方式如下：

    在执行“DriverManager.getConnection”方法获取JDBC连接前，添加“DriverManager.setLoginTimeout(n)”方法来设置超时时长，其中n表示等

待服务返回的超时时长，单位为秒，类型为Int，默认为“0”（表示永不超时）。建议根据业务场景，设置为业务所能容忍的最大等待时长。

27.7.2 structured streaming 功能与可靠性介绍

Structured Streaming 支持的功能

1. 支持对流式数据的ETL操作。
2. 支持流式DataFrames或Datasets的schema推断和分区。
3. 流式DataFrames或Datasets上的操作：包括无类型，类似SQL的操作（比如select、where、groupBy），以及有类型的RDD操作（比如map、filter、flatMap）。
4. 支持基于Event Time的聚合计算，支持对迟到数据的处理。
5. 支持对流式数据的去除重复数据操作。
6. 支持状态计算。
7. 支持对流处理任务的监控。
8. 支持批流join，流流join。

   当前JOIN操作支持列表如下：

左表	右表	支持的Join类型	说明
Static	Static	全部类型	即使在流处理中，不涉及流数据的join操作也能全部支持
Stream	Static	Inner	支持，但是无状态
		Left Outer	支持，但是无状态
		Right Outer	不支持
		Full Outer	不支持
Stream	Stream	Inner	支持，左右表可选择使用watermark或者时间范围进行状态清理
		Left Outer	有条件的支持，左表可选择使用watermark进行状态清理，右表必须使用watermark+时间范围
		Right Outer	有条件的支持，右表可选择使用watermark进行状态清理，左表必须使用watermark+时间范围
		Full Outer	不支持

Structured Streaming 不支持的功能

1. 不支持多个流聚合。
2. 不支持limit、first、take这些取N条Row的操作。
3. 不支持Distinct。

4. 只有当output mode为complete时才支持排序操作。
5. 有条件地支持流和静态数据集之间的外连接。
6. 不支持部分DataSet上立即运行查询并返回结果的操作：
   - count()：无法从流式Dataset返回单个计数，而是使用ds.groupBy().count()返回一个包含运行计数的streaming Dataset。
   - foreach()：使用ds.writeStream.foreach(...)代替。
   - show()：使用输出console sink代替。

Structured Streaming 可靠性说明

Structured Streaming通过checkpoint和WAL机制，对可重放的sources，以及支持重复处理的幂等性sinks，可以提供端到端的exactly-once容错语义。

1. 用户可在程序中设置option("checkpointLocation", "checkpoint路径")启用checkpoint。

   从checkpoint恢复时，应用程序或者配置可能发生变更，有部分变更会导致从checkpoint恢复失败，具体限制如下：

   a. 不允许source的个数或者类型发生变化。
   b. source的参数变化，这种情况是否能被支持，取决于source类型和查询语句，例如：
      - 速率控制相关参数的添加、删除和修改，此种情况能被支持，如：spark.readStream.format("kafka").option("subscribe", "topic")变更为spark.readStream.format("kafka").option("subscribe", "topic").option("maxOffsetsPerTrigger", ...)
      - 修改消费的topic/files可能会出现不可预知的问题，如：spark.readStream.format("kafka").option("subscribe", "topic")变更为spark.readStream.format("kafka").option("subscribe", "newTopic")
   c. sink的类型发生变化：允许特定的几个sink的组合，具体场景需要验证确认，例如：
      - File sink允许变更为kafka sink，kafka中只处理新数据。
      - kafka sink不允许变更为file sink。
      - kafka sink允许变更为foreach sink，反之亦然。
   d. sink的参数变化，这种情况是否能被支持，取决于sink类型和查询语句，例如：
      - 不允许file sink的输出路径发生变更。
      - 允许Kafka sink的输出topic发生变更。
      - 允许foreach sink中的自定义算子代码发生变更，但是变更结果取决于用户代码。
   e. Projection、filter和map-like操作变更，局部场景下能够支持，例如：
      - 支持Filter的添加和删除，如：sdf.selectExpr("a")变更为sdf.where(...).selectExpr("a").filter(...)

- Output schema相同时，projections允许变更，如：sdf.selectExpr("stringColumn AS json").writeStream变更为sdf.select(to_json(...).as("json")).writeStream
- Output schema不相同时，projections在部分条件下允许变更，如：sdf.selectExpr("a").writeStream变更为sdf.selectExpr("b").writeStream，只有当sink支持“a”到“b”的schema转换时才不会出错。

f. 状态操作的变更，在部分场景下会导致状态恢复失败：

- Streaming aggregation：如sdf.groupBy("a").agg(...)操作中，不允许分组键或聚合键的类型或者数量发生变化。
- Streaming deduplication：如：sdf.dropDuplicates("a")操作中，不允许分组键或聚合键的类型或者数量发生变化。
- Stream-stream join：如sdf1.join(sdf2, ...)操作中，关联键的schema不允许发生变化，join类型不允许发生变化，其他join条件的变更可能导致不确定性结果。
- 任意状态计算：如sdf.groupByKey(...).mapGroupsWithState(...)或者sdf.groupByKey(...).flatMapGroupsWithState(...)操作中，用户自定义状态的schema或者超时类型都不允许发生变化；允许用户自定义state-mapping函数变化，但是变更结果取决于用户代码；如果需要支持schema变更，用户可以将状态数据编码/解码成二进制数据以支持schema迁移。

2. Source的容错性支持列表

Sources	支持的Options	容错支持	说明
File source	path：必填，文件路径 maxFilesPerTrigger：每次trigger最大文件数（默认无限大） latestFirst：是否有限处理新文件（默认值: false） fileNameOnly：是否以文件名作为新文件校验，而不是使用完整路径进行判断（默认值: false）	支持	支持通配符路径，但不支持以逗号分隔的多个路径。 文件必须以原子方式放置在给定的目录中，这在大多数文件系统中可以通过文件移动操作实现。
Socket Source	host：连接的节点ip，必填 port：连接的端口，必填	不支持	-
Rate Source	rowsPerSecond：每秒产生的行数，默认值1 rampUpTime：在达到rowsPerSecond速度之前的上升时间 numPartitions：生成数据行的并行度	支持	-

Sources	支持的Options	容错支持	说明
Kafka Source	参见https://spark.apache.org/docs/3.1.1/structured-streaming-kafka-integration.html	支持	-

3. Sink的容错性支持列表

Sinks	支持的output模式	支持Options	容错性	说明
File Sink	Append	Path：必须指定 指定的文件格式，参见DataFrameWriter中的相关接口	exactly-once	支持写入分区表，按时间分区用处较大
Kafka Sink	Append, Update, Complete	参见：https://spark.apache.org/docs/3.1.1/structured-streaming-kafka-integration.html	at-least-once	参见https://spark.apache.org/docs/3.1.1/structured-streaming-kafka-integration.html
Foreach Sink	Append, Update, Complete	None	依赖于ForeachWriter实现	参见https://spark.apache.org/docs/3.1.1/structured-streaming-programming-guide.html#using-foreach
ForeachBatch Sink	Append, Update, Complete	None	依赖于算子实现	参见https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#using-foreach-and-foreachbatch
Console Sink	Append, Update, Complete	numRows：每轮打印的行数，默认20 truncate：输出太长时是否清空，默认true	不支持容错	-

Sinks	支持的output模式	支持Options	容错性	说明
Memory Sink	Append, Complete	None	不支持容错，在complete模式下，重启query会重建整个表	-

27.7.3 如何添加自定义代码的依赖包

问题

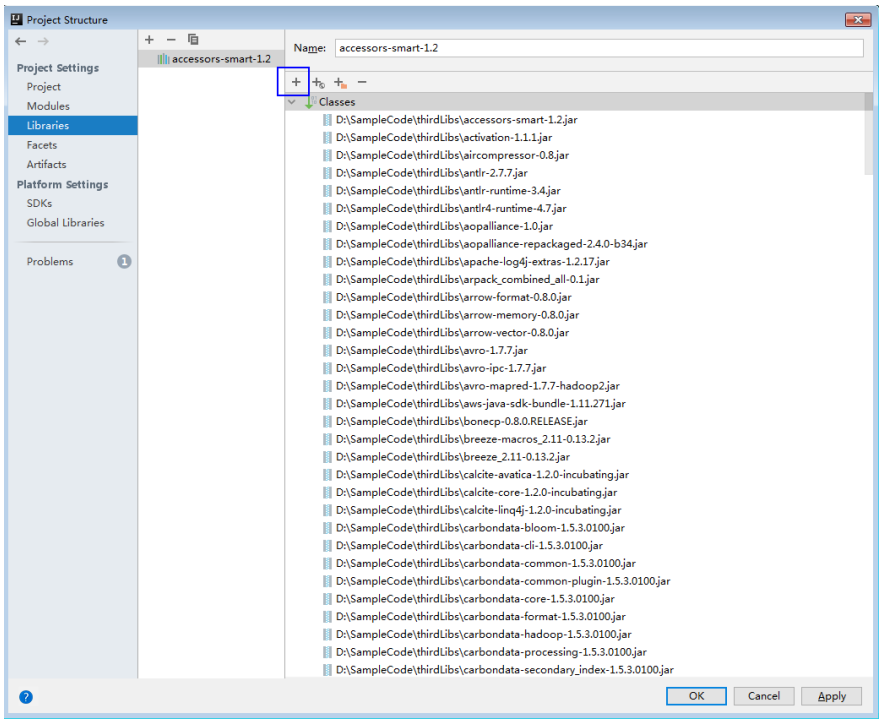
用户在开发Spark程序时，会添加样例程序外的自定义依赖包。针对自定义代码的依赖包，如何使用IDEA添加到工程中?

回答

步骤1 在IDEA主页面，选择“File > Project Structures...”进入“Project Structure”页面。

步骤2 选择“Libraries”页签，然后在如下页面，单击“+”，添加本地的依赖包。

图 27-41 添加依赖包

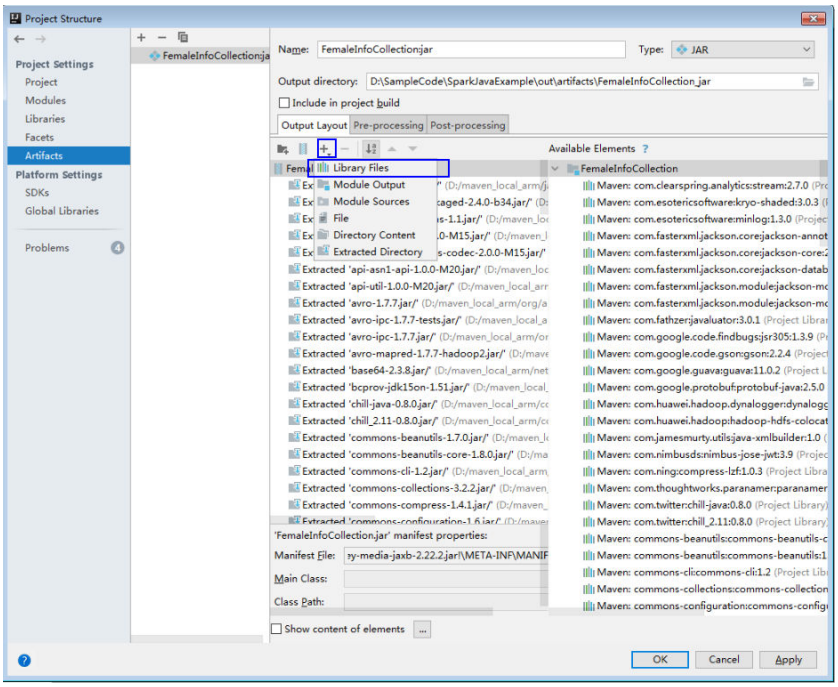


步骤3 单击“Apply”加载依赖包，然后单击“OK”完成配置。

步骤4 由于运行环境不存在用户自定义的依赖包，您还需要在编包时添加此依赖包。以便生成的jar包已包含自定义的依赖包，确保Spark程序能正常运行。

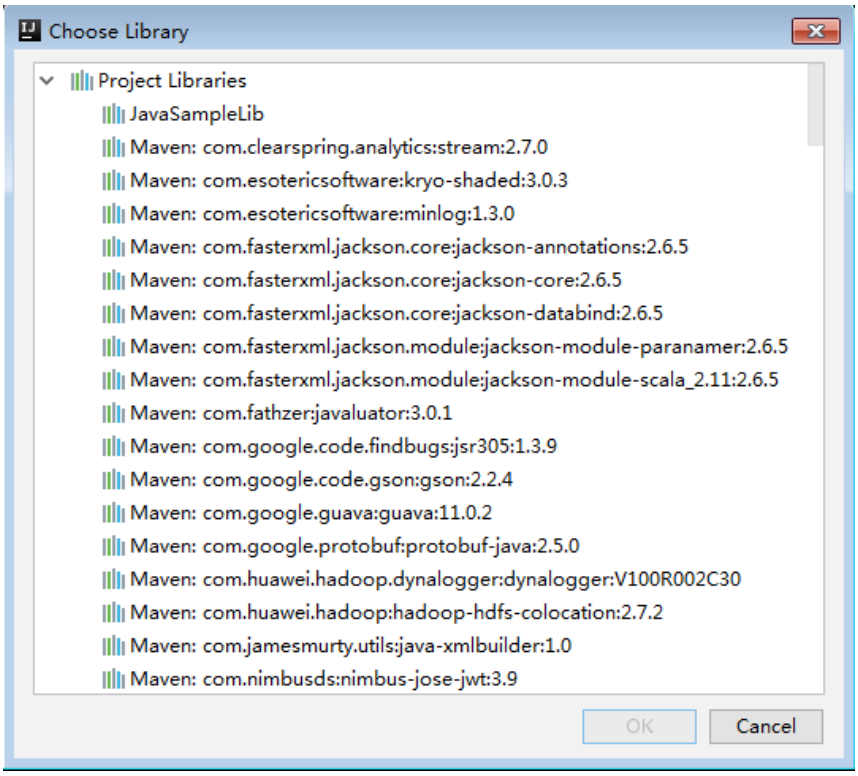
1. 在“Project Structure”页面，选择“Artifacts”页签。
2. 在右侧窗口中单击“+”，选择“Library Files”添加依赖包。

图 27-42 添加 Library Files



3. 选择需要添加的依赖包，然后单击“OK”。

图 27-43 Choose Libraries



4. 单击“Apply”加载依赖包，然后单击“OK”完成配置。

----结束

27.7.4 如何处理自动加载的依赖包

问题

在使用IDEA导入工程前，如果IDEA工具中已经进行过Maven配置时，会导致工具自动加载Maven配置中的依赖包。当自动加载的依赖包与应用程序不配套时，导致工程Build失败。如何处理自动加载的依赖包?

回答

建议在导入工程后，手动删除自动加载的依赖。步骤如下：

1. 在IDEA工具中，选择“File > Project Structures...”,。
2. 选择“Libraries”，选中自动导入的依赖包，右键选择“Delete”。

27.7.5 运行 SparkStreamingKafka 样例工程时报“类不存在”问题

问题

通过spark-submit脚本提交KafkaWordCount（org.apache.spark.examples.streaming.KafkaWordCount）任务时，日志中报Kafka相关的类不存在的错误。KafkaWordCount样例为Spark开源社区提供的。

回答

Spark部署时，如下jar包存放在客户端的“${SPARK_HOME}/jars/streamingClient010”目录以及服务端的“${BIGDATA_HOME}/FusionInsight_Spark2x_8.1.0.1/install/FusionInsight-Spark2x-3.1.1/spark/jars/streamingClient010”目录：

- kafka-clients-xxx.jar
- kafka_2.12-xxx.jar
- spark-streaming-kafka-0-10_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar
- spark-token-provider-kafka-0-10_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar

由于“$SPARK_HOME/jars/streamingClient010/*”默认没有添加到classpath，所以需要手动配置。

在提交应用程序运行时，在命令中添加如下参数即可，详细示例可参考**在Linux环境中编包并运行Spark程序**。

--jars $SPARK_CLIENT_HOME/jars/streamingClient010/kafka-client-2.4.0.jar,$SPARK_CLIENT_HOME/jars/streamingClient010/kafka_2.12-*.jar,$SPARK_CLIENT_HOME/jars/streamingClient010/spark-streaming-kafka-0-10_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar

用户自己开发的应用程序以及样例工程都可使用上述命令提交。

但是Spark开源社区提供的KafkaWordCount等样例程序，不仅需要添加--jars参数，还需要配置其他，否则会报“ClassNotFoundException”错误，yarn-client和yarn-cluster模式下稍有不同。

- yarn-client模式下

  在除--jars参数外，在客户端“spark-defaults.conf”配置文件中，将“spark.driver.extraClassPath”参数值中添加客户端依赖包路径，如“$SPARK_HOME/jars/streamingClient010/*”。

- yarn-cluster模式下

  除--jars参数外，还需要配置其他，有三种方法任选其一即可，具体如下：

  - 在客户端spark-defaults.conf配置文件中，在“spark.yarn.cluster.driver.extraClassPath”参数值中添加服务端的依赖包路径，如“${BIGDATA_HOME}/FusionInsight_Spark2x_8.1.0.1/install/FusionInsight-Spark2x-3.1.1/spark/jars/streamingClient010/*”。
  - 将各服务端节点的“original-spark-examples_2.12-3.1.1-xxx.jar”包删除。
  - 在客户端“spark-defaults.conf”配置文件中，修改或增加配置选项“spark.driver.userClassPathFirst” = “true”。

27.7.6 SparkSQL UDF 功能的权限控制机制

问题

SparkSQL中UDF功能的权限控制机制是怎样的?

回答

目前已有的SQL语句无法满足用户场景时，用户可使用UDF功能进行自定义操作。

为确保数据安全以及UDF中的恶意代码对系统造成破坏，SparkSQL的UDF功能只允许具备admin权限的用户注册，由admin用户保证自定义的函数的安全性。

27.7.7 由于 Kafka 配置的限制，导致 Spark Streaming 应用运行失败

问题

使用运行的Spark Streaming任务回写Kafka时，Kafka上接收不到回写的数据，且Kafka日志报错信息如下：

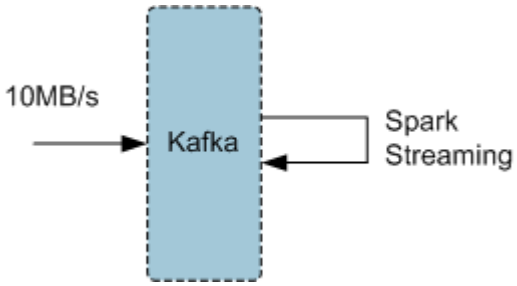
```
2016-03-02 17:46:19,017 | INFO | [kafka-network-thread-21005-1] | Closing socket connection to /
10.91.8.208 due to invalid request: Request of length
 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. | kafka.network.Processor
(Logging.scala:68)
2016-03-02 17:46:19,155 | INFO | [kafka-network-thread-21005-2] | Closing socket connection to /
10.91.8.208. | kafka.network.Processor (Logging.scala:68)
2016-03-02 17:46:19,270 | INFO | [kafka-network-thread-21005-0] | Closing socket connection to /
10.91.8.208 due to invalid request:
Request of length 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. |
kafka.network.Processor (Logging.scala:68)
2016-03-02 17:46:19,513 | INFO | [kafka-network-thread-21005-1] | Closing socket connection to /
10.91.8.208 due to invalid request:
Request of length 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. |
kafka.network.Processor (Logging.scala:68)
2016-03-02 17:46:19,763 | INFO | [kafka-network-thread-21005-2] | Closing socket connection to /
10.91.8.208 due to invalid request:
Request of length 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. |
kafka.network.Processor (Logging.scala:68)
53393 [main] INFO  org.apache.hadoop.mapreduce.Job  - Counters: 50
```

回答

如下图所示，Spark Streaming应用中定义的逻辑为，从Kafka中读取数据，执行对应处理之后，然后将结果数据回写至Kafka中。

例如：Spark Streming中定义了批次时间，如果数据传入Kafka的速率为10MB/s，而Spark Streaming中定义了每60s一个批次，回写数据总共为600MB。而Kafka中定义了接收数据的阈值大小为500MB。那么此时回写数据已超出阈值。此时，会出现上述错误。

图 27-44 应用场景



解决措施：

方式一：推荐优化Spark Streaming应用程序中定义的批次时间，降低批次时间，可避免超过Kafka定义的阈值。一般建议以5-10秒/次为宜。

方式二：将Kafka的阈值调大，建议在FusionInsight Manager中的Kafka服务进行参数设置，将socket.request.max.bytes参数值根据应用场景，适当调整。

27.7.8 执行 Spark Core 应用，尝试收集大量数据到 Driver 端，当 Driver 端内存不足时，应用挂起不退出

问题

执行Spark Core应用，尝试收集大量数据到Driver端，当Driver端内存不足时，应用挂起不退出，日志内容如下。

```
16/04/19 15:56:22 ERROR Utils: Uncaught exception in thread task-result-getter-2
java.lang.OutOfMemoryError: Java heap space
at java.lang.reflect.Array.newArray(Native Method)
at java.lang.reflect.Array.newInstance(Array.java:75)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1671)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1707)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
at org.apache.spark.serializer.JavaDeserializationStream.readObject(JavaSerializer.scala:71)
at org.apache.spark.serializer.JavaSerializerInstance.deserialize(JavaSerializer.scala:91)
at org.apache.spark.scheduler.DirectTaskResult.value(TaskResult.scala:94)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply$mcV
$sp(TaskResultGetter.scala:66)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.util.Utils$.logUncaughtExceptions(Utils.scala:1716)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3.run(TaskResultGetter.scala:56)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)
Exception in thread "task-result-getter-2" java.lang.OutOfMemoryError: Java heap space
at java.lang.reflect.Array.newArray(Native Method)
at java.lang.reflect.Array.newInstance(Array.java:75)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1671)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1707)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
at org.apache.spark.serializer.JavaDeserializationStream.readObject(JavaSerializer.scala:71)
at org.apache.spark.serializer.JavaSerializerInstance.deserialize(JavaSerializer.scala:91)
at org.apache.spark.scheduler.DirectTaskResult.value(TaskResult.scala:94)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply$mcV
$sp(TaskResultGetter.scala:66)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.util.Utils$.logUncaughtExceptions(Utils.scala:1716)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3.run(TaskResultGetter.scala:56)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
```

```
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)
```

回答

用户尝试收集大量数据到Driver端，如果Driver端的内存不足以存放这些数据，那么就会抛出OOM(OutOfMemory)的异常，然后Driver端一直在进行GC，尝试回收垃圾来存放返回的数据，导致应用长时间挂起。

解决措施：

如果用户需要在OOM场景下强制将应用退出，那么可以在启动Spark Core应用时，在客户端配置文件“$SPARK_HOME/conf/spark-defaults.conf”中的配置项“spark.driver.extraJavaOptions”中添加如下内容：

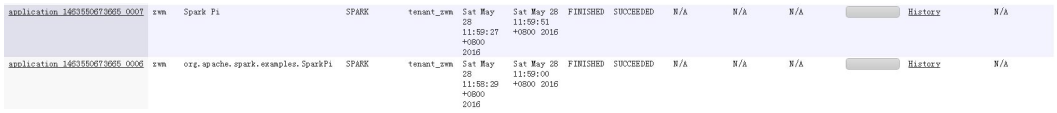
```
-XX:OnOutOfMemoryError='kill -9 %p'
```

27.7.9 Spark 应用名在使用 yarn-cluster 模式提交时不生效

问题

Spark应用名在使用yarn-cluster模式提交时不生效，在使用yarn-client模式提交时生效，如图27-45所示，第一个应用是使用yarn-client模式提交的，正确显示代码里设置的应用名Spark Pi，第二个应用是使用yarn-cluster模式提交的，设置的应用名没有生效。

图 27-45 提交应用



回答

导致这个问题的主要原因是，yarn-client和yarn-cluster模式在提交任务时setAppName的执行顺序不同导致，yarn-client中setAppName是在向yarn注册Application之前读取，yarn-cluser模式则是在向yarn注册Application之后读取，这就导致yarn-cluster模式设置的应用名不生效。

解决措施：

在spark-submit脚本提交任务时用--name设置应用名和sparkconf.setAppName(appname)里面的应用名一样。

比如代码里设置的应用名为Spark Pi，用yarn-cluster模式提交应用时可以这样设置，在--name后面添加应用名，执行的命令如下：

***./spark-submit --class** org.apache.spark.examples.SparkPi **--master yarn --deploy-mode cluster --name** SparkPi jars/original-spark-examples*.jar 10*

27.7.10 如何使用 IDEA 远程调试

问题

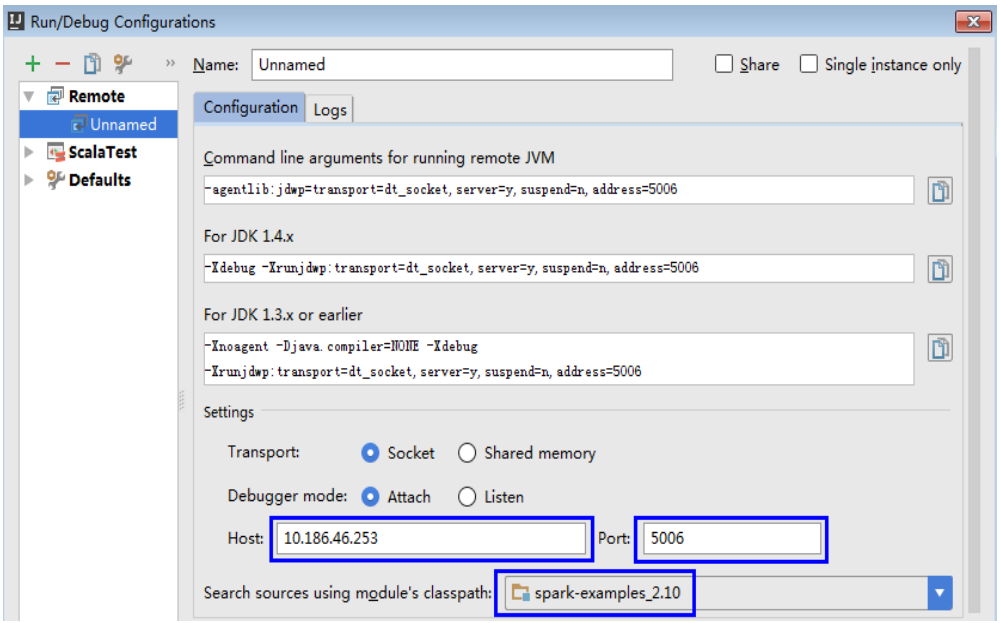
在Spark二次开发中如何使用IDEA远程调试?

回答

以调试SparkPi程序为例，演示如何进行IDEA的远程调试：

1. 打开工程，在菜单栏中选择“Run > Edit Configurations”。
2. 在弹出的配置窗口中用鼠标左键单击左上角的+号，在下拉菜单中选择Remote，如图27-46所示。

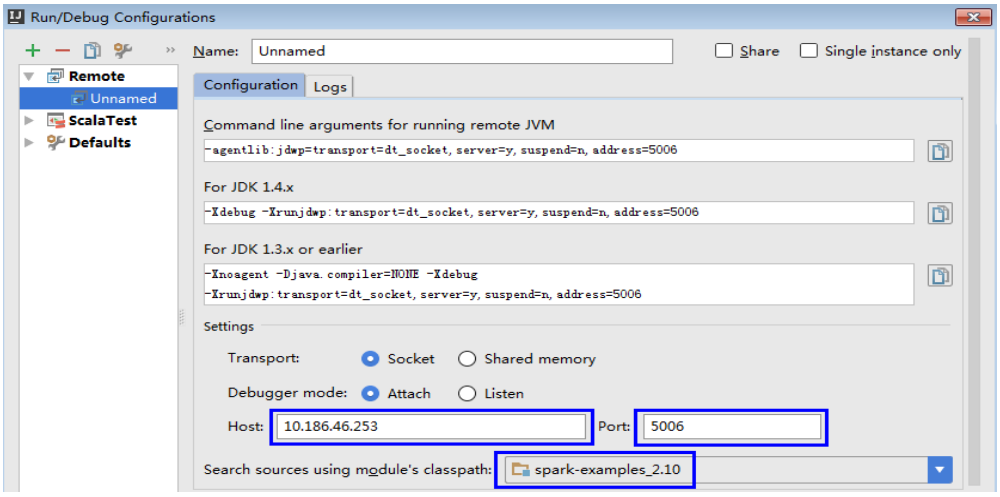
图 27-46 选择 Remote



3. 选择对应要调试的源码模块路径，并配置远端调试参数Host和Port，如图27-47所示。

   其中Host为Spark运行机器IP地址，Port为调试的端口号（确保该端口在运行机器上没被占用）。

图 27-47 配置参数



说明

当改变Port端口号时，For JDK1.4.x对应的调试命令也跟着改变，比如Port设置为5006，对应调试命令会变更为-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006，这个调试命令在启动Spark程序时要用到。

4. 执行以下命令，远端启动Spark运行SparkPi。

   ./spark-submit --master yarn-client --driver-java-options *"-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006" --class org.apache.spark.examples.SparkPi /opt/FI-Client/Spark2x/spark/examples/jars/spark-examples_2.12-3.1.1-xxx.jar*

   用户调试时需要把--class和jar包换成自己的程序，-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006需要换成3获取到的For JDK1.4.x对应的调试命令。

   图 27-48 Spark 运行命令

   ```
   [root@host2 bin]#./spark-submit --master yarn-client --driver-java-options "-Xde
   bug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006" --class org.a
   pache.spark.examples.SparkPi /opt/client/Spark2x/spark/examples/jars/spark-examp
   les_2.11-2.1.0.jar
   Listening for transport dt_socket at address: 5006
   ```

5. 设置调试断点。

   在IDEA代码编辑窗口左侧空白处单击鼠标左键设置相应代码行断点，如图27-49所示，在SparkPi.scala的29行设置断点。

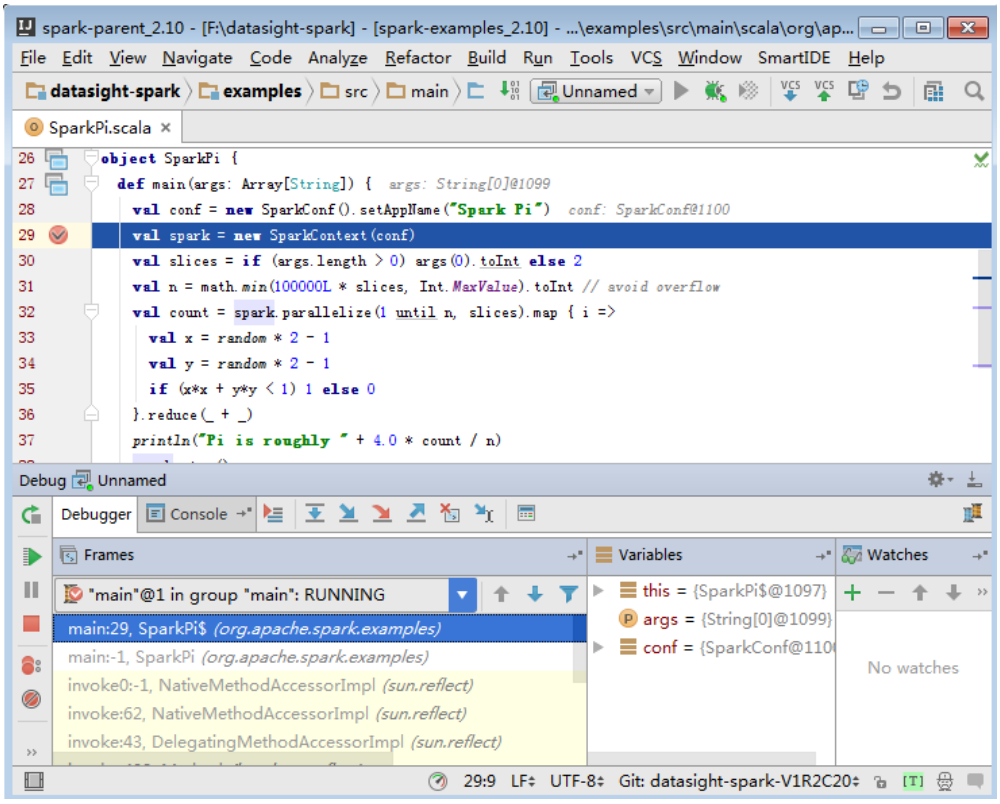
   图 27-49 设置断点

   ```
   26  object SparkPi {
   27    def main(args: Array[String]) {
   28      val conf = new SparkConf().setAppName("Spark Pi")
   29      val spark = new SparkContext(conf)
   30      val slices = if (args.length > 0) args(0).toInt else 2
   31      val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid over
   32      val count = spark.parallelize(1 until n, slices).map { i =>
   33        val x = random * 2 - 1
   34        val y = random * 2 - 1
   35        if (x*x + y*y < 1) 1 else 0
   36      }.reduce(_ + _)
   37      println("Pi is roughly " + 4.0 * count / n)
   38      spark.stop()
   39    }
   40  }
   ```

6. 启动调试。

   在IDEA菜单栏中选择“Run > Debug 'Unnamed'”开启调试窗口，接着开始SparkPi的调试，比如单步调试、查看调用栈、跟踪变量值等，如图27-50所示。

图 27-50 调试



27.7.11 如何采用 Java 命令提交 Spark 应用

问题

除了spark-submit命令提交应用外，如何采用Java命令提交Spark应用?

回答

您可以通过org.apache.spark.launcher.SparkLauncher类采用java命令方式提交Spark应用。详细步骤如下：

步骤1 定义org.apache.spark.launcher.SparkLauncher类。默认提供了SparkLauncherJavaExample和SparkLauncherScalaExample示例，您需要根据实际业务应用程序修改示例代码中的传入参数。

- 如果您使用Java语言开发程序，您可以参考如下示例，编写SparkLauncher类。

```
public static void main(String[] args) throws Exception {
    System.out.println("com.huawei.bigdata.spark.examples.SparkLauncherExample <mode>
<jarParh> <app_main_class> <appArgs>");
    SparkLauncher launcher = new SparkLauncher();
    launcher.setMaster(args[0])
        .setAppResource(args[1]) // Specify user app jar path
        .setMainClass(args[2]);
    if (args.length > 3) {
        String[] list = new String[args.length - 3];
        for (int i = 3; i < args.length; i++) {
            list[i-3] = args[i];
        }
        // Set app args
        launcher.addAppArgs(list);
    }
```



```
    // Launch the app
    Process process = launcher.launch();
    // Get Spark driver log
    new Thread(new ISRRunnable(process.getErrorStream())).start();
    int exitCode = process.waitFor();
    System.out.println("Finished! Exit code is " + exitCode);
  }
```

- 如果您使用Scala语言开发程序，您可以参考如下示例，编写SparkLauncher类。

```
def main(args: Array[String]) {
  println(s"com.huawei.bigdata.spark.examples.SparkLauncherExample <mode> <jarParh>
<app_main_class> <appArgs>")
  val launcher = new SparkLauncher()
  launcher.setMaster(args(0))
    .setAppResource(args(1)) // Specify user app jar path
    .setMainClass(args(2))
    if (args.drop(3).length > 0) {
      // Set app args
      launcher.addAppArgs(args.drop(3): _*)
    }

  // Launch the app
  val process = launcher.launch()
  // Get Spark driver log
  new Thread(new ISRRunnable(process.getErrorStream)).start()
  val exitCode = process.waitFor()
  println(s"Finished! Exit code is $exitCode")
}
```

步骤2 根据业务逻辑，开发对应的Spark应用程序。并设置用户编写的Spark应用程序的主类等常数。不同场景的示例请参考**开发Spark应用**。

- 如果您使用的安全模式，建议按照安全要求，准备安全认证代码、业务应用代码及其相关配置。

  说明

  yarn-cluster模式中不支持在Spark工程中添加安全认证。因为需要在应用启动前已完成安全认证。所以用户需要在Spark应用之外添加安全认证代码或使用命令行进行认证。由于提供的示例代码默认提供安全认证代码，请在yarn-cluster模式下时，修改对应安全代码后再运行应用。

- 如果您使用的是普通模式，准备业务应用代码及其相关配置即可。

步骤3 调用org.apache.spark.launcher.SparkLauncher.launch()方法，将用户的应用程序提交。

1. 将SparkLauncher程序和用户应用程序分别生成Jar包，并上传至运行此应用的Spark节点中。生成Jar包的操作步骤请参见**在Linux环境中编包并运行Spark程序**章节。
   - SparkLauncher程序的编译依赖包为spark-launcher_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar，请从软件发布包中Software文件夹下“FusionInsight_Spark2x_8.1.0.1.tar.gz”压缩包中的“jars”目录中获取。
   - 用户应用程序的编译依赖包根据代码不同而不同，需用户根据自己编写的代码进行加载。
2. 将运行程序的依赖Jar包上传至需要运行此应用的节点中，例如“$SPARK_HOME/jars”路径。

   用户需要将SparkLauncher类的运行依赖包和应用程序运行依赖包上传至客户端的jars路径。文档中提供的示例代码，其运行依赖包在客户端jars中已存在。

说明

Spark Launcher的方式依赖Spark客户端，即运行程序的节点必须已安装Spark客户端，且客户端可用。运行过程中依赖客户端已配置好的环境变量、运行依赖包和配置文件，

3. 在Spark应用程序运行节点，执行如下命令使用Spark Launcher方式提交。之后，可通过Spark WebUI查看运行情况，或通过获取指定文件查看运行结果，可参见**在Linux环境中查看Spark程序调测结果**。

   java -cp $SPARK_HOME/conf:$SPARK_HOME/jars/*:SparkLauncherExample.jar com.huawei.bigdata.spark.examples.SparkLauncherExample yarn-client /opt/female/FemaleInfoCollection.jar com.huawei.bigdata.spark.examples.FemaleInfoCollection <inputPath>

----结束

27.7.12 使用 IBM JDK 产生异常，提示“Problem performing GSS wrap”信息

问题

使用IBM JDK产生异常，提示“Problem performing GSS wrap”信息

回答

问题原因：

在IBM JDK下建立的JDBC connection时间超过登录用户的认证超时时间（默认一天），导致认证失败。

说明

IBM JDK的机制跟Oracle JDK的机制不同，IBM JDK在认证登录后的使用过程中做了时间检查却没有检测外部的时间更新，导致即使显式调用relogin也无法得到刷新。

解决措施：

通常情况下，在发现JDBC connection不可用的时候，可以关闭该connection，重新创建一个connection继续执行。

27.7.13 Structured Streaming 的 cluster 模式，在数据处理过程中终止 ApplicationManager，应用失败

问题

Structured Streaming的cluster模式，在数据处理过程中终止ApplicationManager，执行应用时显示如下异常。

```
2017-05-09 20:46:02,393 | INFO  | main |
 client token: Token { kind: YARN_CLIENT_TOKEN, service:  }
 diagnostics: User class threw exception: org.apache.spark.sql.AnalysisException: This query does not support recovering from checkpoint location. Delete hdfs://hacluster/structuredtest/checkpoint/offsets to start over.;
 ApplicationMaster host: 10.96.101.170
 ApplicationMaster RPC port: 0
 queue: default
```

```
 start time: 1494333891969
 final status: FAILED
 tracking URL: https://9-96-101-191:8090/proxy/application_1493689105146_0052/
 user: spark2x | org.apache.spark.internal.Logging$class.logInfo(Logging.scala:54)
Exception in thread "main" org.apache.spark.SparkException: Application application_1493689105146_0052
finished with failed status
```

回答

原因分析：显示该异常是因为“recoverFromCheckpointLocation”的值判定为false，但却配置了checkpoint目录。

参数“recoverFromCheckpointLocation”的值为代码中“outputMode == OutputMode.Complete()”语句的判断结果（outputMode的默认输出方式为“append”）。

处理方法：编写应用时，用户可以根据具体情况修改数据的输出方式。

将输出方式修改为“complete”，“recoverFromCheckpointLocation”的值会判定为true。此时配置了checkpoint目录时就不会显示异常。

27.7.14 从 checkpoint 恢复 spark 应用的限制

问题

Spark应用可以从checkpoint恢复，用于从上次任务中断处继续往下执行，以保证数据不丢失。但是，在某些情况下，从checkpoint恢复应用会失败。

回答

由于checkpoint中包含了spark应用的对象序列化信息、task执行状态信息、配置信息等，因此，当存在以下问题时，从checkpoint恢复spark应用将会失败。

1. 业务代码变更且变更类未明确指定SerialVersionUID。
2. spark内部类变更，且变更类未明确指定SerialVersionUID。

另外，由于checkpoint保存了部分配置项，因此可能导致业务修改了部分配置项后，从checkpoint恢复时，配置项依然保持为旧值的情况。当前只有以下部分配置会在从checkpoint恢复时重新加载。

```
"spark.yarn.app.id",
 "spark.yarn.app.attemptId",
 "spark.driver.host",
 "spark.driver.bindAddress",
 "spark.driver.port",
 "spark.master",
 "spark.yarn.jars",
 "spark.yarn.keytab",
 "spark.yarn.principal",
 "spark.yarn.credentials.file",
 "spark.yarn.credentials.renewalTime",
 "spark.yarn.credentials.updateTime",
 "spark.ui.filters",
 "spark.mesos.driver.frameworkId",
 "spark.yarn.jars"
```

解决方法

手动删除checkpoint目录，重启业务程序。

说明

删除文件为高危操作，在执行操作前请务必确认对应文件是否不再需要。

27.7.15 第三方 jar 包跨平台（x86、TaiShan）支持

问题

用户自己写的jar包（例如自定义udf包）区分x86和TaiShan版本，如何让Spark2x支持其正常运行。

回答

第三方jar包（例如自定义udf）区分x86和TaiShan版本时，混合使用方案：

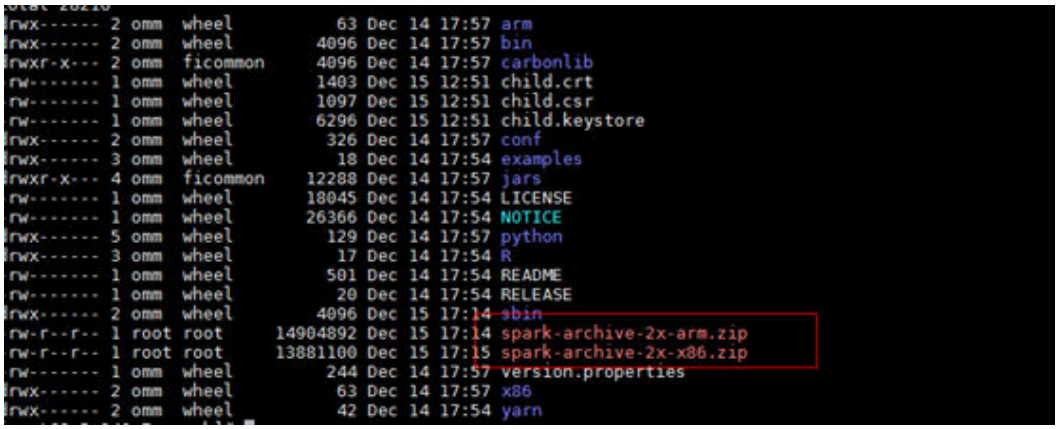
步骤1 进入到服务端Spark2x SparkResource的安装目录（集群安装时，SparkResource可能会安装在多个节点上，登录任意一个SparkResource节点，进入到SparkResource的安装目录）。

步骤2 准备好自己的jar包，例如xx.jar的x86版本和TaiShan版本。将x86版本和TaiShan版本的xx.jar分别复制到当前目录的x86文件夹和TaiShan文件夹里面。

步骤3 在当前目录下执行以下命令将jar包打包：

zip -qDj spark-archive-2x-x86.zip x86/*

zip -qDj spark-archive-2x-arm.zip arm/*



步骤4 执行以下命令查看hdfs上的spark2x依赖的jar包：

hdfs dfs -ls /user/spark2x/jars/8.1.0.1

说明

8.1.0.1是版本号，不同版本不同。

执行以下命令移动hdfs上旧的jar包文件到其他目录，例如移动到“tmp”目录。

hdfs dfs -mv /user/spark2x/jars/8.1.0.1/spark-archive-2x-arm.zip /tmp

hdfs dfs -mv /user/spark2x/jars/8.1.0.1/spark-archive-2x-x86.zip /tmp

步骤5 上传步骤3中打包的spark-archive-2x-arm.zip和spark-archive-2x-x86.zip到hdfs的/user/spark2x/jars/8.1.0.1目录下，上传命令如下：

hdfs dfs -put spark-archive-2x-arm.zip /user/spark2x/jars/8.1.0.1/

hdfs dfs -put spark-archive-2x-x86.zip /user/spark2x/jars/8.1.0.1/

上传完毕后删除本地的spark-archive-2x-arm.zip，spark-archive-2x-x86.zip文件。

步骤6 对其他的sparkResource安装节点执行**步骤1**~**步骤2**。

步骤7 进入webUI重启spark2x的jdbcServer实例。

步骤8 重启后，需要更新客户端配置。按照客户端所在的机器类型（x86、TaiShan）复制xx.jar的相应版本到客户端的Spark2x安装目录“${install_home}/Spark2x/spark/jars”文件夹中。${install_home}是用户的客户端安装路径，用户需要填写实际的安装目录；若本地的安装目录为“/opt/hadoopclient”，那么就复制相应版本xx.jar到“/opt/hadoopclient/Spark2x/spark/jars”文件夹里。

----结束

27.7.16 在客户端安装节点的/tmp 目录下残留了很多 blockmgr-开头和 spark-开头的目录

问题

系统长时间运行后，在客户端安装节点的/tmp目录下，发现残留了很多blockmgr-开头和spark-开头的目录。

图 27-51 残留目录样例

```
blockmgr-934dc20a-d5f0-4adf-a28f-c376ef0fe01d
blockmgr-f514f38b-209c-4a65-985a-2a6c61d0ee00
spark-33f95b4b-be82-4290-bde3-07b76c797085
spark-988e28a7-0416-4115-8d6e-3a62a75f1f46
```

回答

Spark任务在运行过程中，driver会创建一个spark-开头的本地临时目录，用于存放业务jar包，配置文件等，同时在本地创建一个blockmgr-开头的本地临时目录，用于存放block data。此两个目录会在Spark应用运行结束时自动删除。

此两个目录的存放路径优先通过SPARK_LOCAL_DIRS环境变量指定，若不存在该环境变量，则设置为spark.local.dir的值，若此配置还不存在，则使用java.io.tmpdir的值。客户端默认配置中spark.local.dir被设置为/tmp，因此默认使用系统/tmp目录。

但存在一些特殊情况，如driver进程未正常退出，比如被kill -9命令结束进程，或者Java虚拟机直接崩溃等场景，导致driver的退出流程未正常执行，则可能导致该部分目录无法被正常清理，残留在系统中。

当前只有yarn-client模式和local模式的driver进程会产生上述问题，在yarn-cluster模式中，已将container内进程的临时目录设置为container临时目录，当container退出时，由container自动清理该目录，因此yarn-cluster模式不存在此问题。

解决措施

可在Linux下设置/tmp临时目录自动清理，或修改客户端中spark-defaults.conf配置文件的spark.local.dir配置项的值，将临时目录指定到特定的目录，再对该目录单独设置清理机制。

27.7.17 ARM 环境 python pipeline 运行报 139 错误码

问题

在TaiShan服务器上，使用python插件的pipeline报错显示139错误。具体报错如下：

```
subprocess exited with status 139
```

回答

该python程序既使用了libcrypto.so，也使用了libssl.so。而一旦LD_LIBRARY_PATH添加了hadoop的native库目录，则使用的就是hadoop native库中的libcrypto.so，而使用系统自带的libssl.so（因为hadoop native目录没有带该包）。由于这两个库版本不匹配，导致了python文件运行时出现段错误。

解决方案

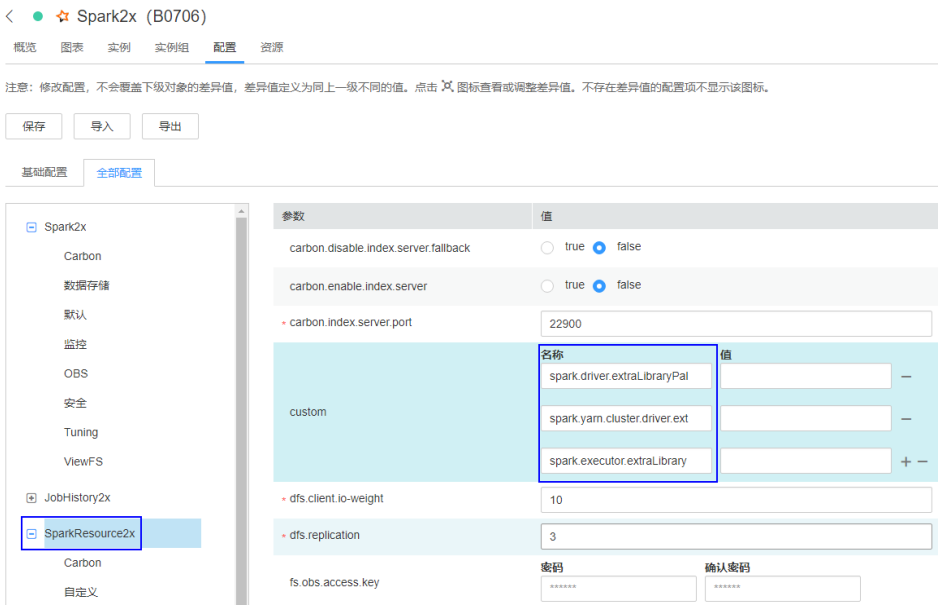
方案一：

修改Spark2x客户端conf目录下spark-default.conf文件，清空（直接赋值为空）配置项spark.driver.extraLibraryPath、spark.yarn.cluster.driver.extraLibraryPath和spark.executor.extraLibraryPath的值。

方案二：

在FusionInsight Mnanager中Spark2x界面中修改上述三个参数然后重启Spark2x实例之后重新下载客户端，具体步骤如下：

1. 登录FusionInsight Mnanager界面，选择“集群 > *待操作集群的名称* > 服务 > Spark2x > 配置 > 全部配置”，搜索参数spark.driver.extraLibraryPath和spark.executor.extraLibraryPath，并清空其参数值。
2. 在“全部配置”中选择“ SparkResource2x”。 在SparkResource2x中的custom中添加方案一中的三个参数，如下图所示：



3. 单击“保存”，完成后重启过期的spark2x实例，并重新下载安装客户端。

27.7.18 Structured Streaming 任务提交方式变更

问题

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，例如**--jars /kafkadir/kafka-clients-x.x.x.jar,/kafkadir/kafka_2.11-x.x.x.jar**。当前版本用户除了这一步外还需要额外的配置项，否则会报class not found异常。

回答

当前版本的Spark内核直接依赖于Kafka相关的jar包（结构流使用），因此提交结构流任务时，需要把Kafka相关jar包加入到结构流任务driver端的库目录下，确保driver能够正常加载kafka包。

解决方案

1. 提交yarn-client模式的结构流任务时需要额外如下操作：

   将Spark客户端目录下spark-default.conf文件中的spark.driver.extraClassPath配置复制出来，并将Kafka相关jar包路径追加到该配置项之后，提交结构流任务时需要通过--conf将该配置项给加上。例如：Kafka相关jar包路径为“/kafkadir”，提交任务需要增加--conf spark.driver.extraClassPath=/opt/client/Spark2x/spark/conf/:/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/x86/*:/kafkadir/*。

2. 提交yarn-cluster模式的结构流任务时需要额外如下操作：

   将Spark客户端目录下spark-default.conf文件中的spark.yarn.cluster.driver.extraClassPath配置给复制出来，并将Kafka相关jar包相对路径追加到该配置项之后，提交结构流任务时需要通过--conf 将该配置项给加上。例如：kafka相关包为kafka-clients-x.x.x.jar，kafka_2.11-x.x.x.jar，提交任务需要增加--conf spark.yarn.cluster.driver.extraClassPath=/home/huawei/Bigdata/common/runtime/security:./kafka-clients-x.x.x.jar:./kafka_2.11-x.x.x.jar。

3. 当前版本Spark结构流部分不再支持kafka2.x之前的版本，对于升级场景请继续使用旧的客户端。

27.7.19 常见 jar 包冲突处理方式

问题现象

Spark能对接很多的第三方工具，因此在使用过程中经常会依赖一堆的三方包。而有一些包MRS已经自带，这样就有可能造成代码使用的jar包版本和集群自带的jar包版本不一致，在使用过程中就有可能出现jar包冲突的情况。

常见的jar包冲突报错有：

1、报错类找不到：java.lang.NoClassDefFoundError

2、报错方法找不到：java.lang.NoSuchMethodError

原因分析

以自定义UDF为例：

```
2021-02-08 10:51:04,249 | INFO  | main | Total input files to process : 2 | org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:256)
Exception in thread "main" java.lang.NoClassDefFoundError: com/huawei/udf/HelloUDF
        at com.huawei.SparkWordCount$.main(SparkWordCount.scala:32)
        at com.huawei.SparkWordCount.main(SparkWordCount.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runMain(SparkSubmit.scala:813)
        at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:187)
        at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)
        at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)
        at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
Caused by: java.lang.ClassNotFoundException: com.huawei.udf.HelloUDF
        at java.net.URLClassLoader.findClass(URLClassLoader.java:382)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        ... 11 more
```

报错信息显示是找不到类。

1. 首先需要确认的是这个类属于的jar包是否在jvm的classpath里面， spark自带的jar都在“*spark客户端目录*/jars/”。
2. 确认是否存在多个jar包拥有这个类。
3. 如果是其他依赖包，可能是没有使用--jars添加到任务里面。
4. 如果是已经添加到任务里面，但是依旧没有取到，可能是因为配置文件的driver或者executor的classpath配置不正确，可以查看日志确认是否加载到环境。
5. 另外可能报错是类初始化失败导致后面使用这个类的时候出现上述报错，需要确认是否在之前就有初始化失败或者其他报错的情况发生。

```
Exception in thread "main" java.lang.NoSuchMethodError: com.huawei.udf.HelloUDF.evaluate(Ljava/lang/String;)Ljava/lang/String;
        at com.huawei.SparkWordCount$.main(SparkWordCount.scala:32)
        at com.huawei.SparkWordCount.main(SparkWordCount.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runMain(SparkSubmit.scala:813)
        at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:187)
        at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)
        at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)
        at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
```

报错信息显示找不到方法。

1. 确认这个方法对应的类所在的jar包是否加载到jvm的classpath里面，spark自带的类都在“*spark客户端目录*/jars/”。
2. 确认是否有多个jar包包含这个类（尤其注意相同工具的不同版本）。
3. 如果报错是Hadoop相关的包，有可能是因为使用的Hadoop版本不一致导致部分方法已经更改。
4. 如果报错的是三方包里面的类，可能是因为Spark已经自带了相关的jar包，但是和代码中使用的版本不一致。

操作步骤

方案一：

针对jar包冲突的问题，可以确认是否不需使用三方工具的包，如果可以更改为集群相同版本的包，则修改引入的依赖版本。

说明

建议用户尽量使用MRS集群自带的依赖包。

方案二：

jar包版本修改演示

以MRS_2.1版本为例：

1. 在pom.xml文件中添加“<properties>”参数，填写变量，方便后面统一修改版本。


```
<groupId>com.huawei</groupId>
<artifactId>SparkDemo</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <spark.version>2.3.2-mrs-2.1</spark.version>
    <hbase.version>2.1.1.0101-mrs-2.1</hbase.version>
    <hadoop.version>3.1.1-mrs-2.1</hadoop.version>
</properties>
```

2. 在“dependencies”参数中设置各个jar包的版本的时候可以直接使用上述定义的参数传递。

```
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>${hadoop.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-common</artifactId>
        <version>${hbase.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-client</artifactId>
        <version>${hbase.version}</version>
    </dependency>
```

如果遇到其他三方包冲突，可以通过查找依赖关系确认是否存在相同包不同版本的情况，尽量修改成集群自带的jar包版本。

可以参考MRS样例工程自带的pom.xml文件：**通过开源镜像站获取样例工程**。

3. 打印依赖树方式：

在pom.xml文件同目录下执行命令：**mvn dependency:tree**

28 Spark2x 开发指南（普通模式）

28.1 Spark 应用开发简介

Spark 简介

Spark是分布式批处理框架，提供分析挖掘与迭代式内存计算能力，支持多种语言（Scala/Java/Python）的应用开发。 适用以下场景：

- 数据处理（Data Processing）：可以用来快速处理数据，兼具容错性和可扩展性。
- 迭代计算（Iterative Computation）：支持迭代计算，有效应对多步的数据处理逻辑。
- 数据挖掘（Data Mining）：在海量数据基础上进行复杂的挖掘分析，可支持各种数据挖掘和机器学习算法。
- 流式处理（Streaming Processing）：支持秒级延迟的流式处理，可支持多种外部数据源。
- 查询分析（Query Analysis）：支持标准SQL查询分析，同时提供DSL（DataFrame）， 并支持多种外部输入。

本文档重点介绍Spark、Spark SQL和Spark Streaming应用开发指导。

Spark 开发接口简介

Spark支持使用Scala、Java和Python语言进行程序开发，由于Spark本身是由Scala语言开发出来的，且Scala语言具有简洁易懂的特性，推荐用户使用Scala语言进行Spark应用程序开发。

按不同的语言分，Spark的API接口如**表28-1**所示。

表 28-1 Spark API 接口

功能	说明
Scala API	提供Scala语言的API，Spark Core、SparkSQL和Spark Streaming模块的常用接口请参见**Spark Scala API接口介绍**。由于Scala语言的简洁易懂，推荐用户使用Scala接口进行程序开发。
Java API	提供Java语言的API，Spark Core、SparkSQL和Spark Streaming模块的常用接口请参见**Spark Java API接口介绍**。
Python API	提供Python语言的API，Spark Core、SparkSQL和Spark Streaming模块的常用接口请参见**Spark Python API接口介绍**。

按不同的模块分，Spark Core和Spark Streaming使用上表中的API接口进行程序开发。而SparkSQL模块，支持CLI或者JDBCServer两种方式访问。其中JDBCServer的连接方式也有Beeline和JDBC客户端代码两种。详情请参见**Spark JDBCServer接口介绍**。

说明

spark-sql脚本、spark-shell脚本和spark-submit脚本（运行的应用中带SQL操作），不支持使用proxy user参数去提交任务。

基本概念

- **RDD**

  即弹性分布数据集（Resilient Distributed Dataset），是Spark的核心概念。指的是一个只读的，可分区的分布式数据集，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用。

  RDD的生成：

  - 从HDFS输入创建，或从与Hadoop兼容的其他存储系统中输入创建。
  - 从父RDD转换得到新RDD。
  - 从数据集合转换而来，通过编码实现。

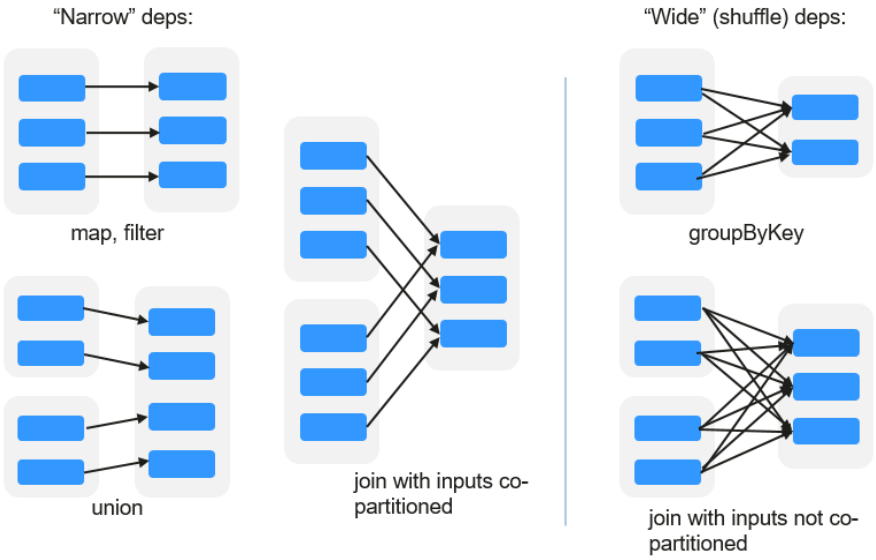
  RDD的存储：

  - 用户可以选择不同的存储级别缓存RDD以便重用（RDD有11种存储级别）。
  - 当前RDD默认是存储于内存，但当内存不足时，RDD会溢出到磁盘中。

- **Dependency（RDD的依赖）**

  RDD的依赖分别为：窄依赖和宽依赖。

图 28-1 RDD 的依赖



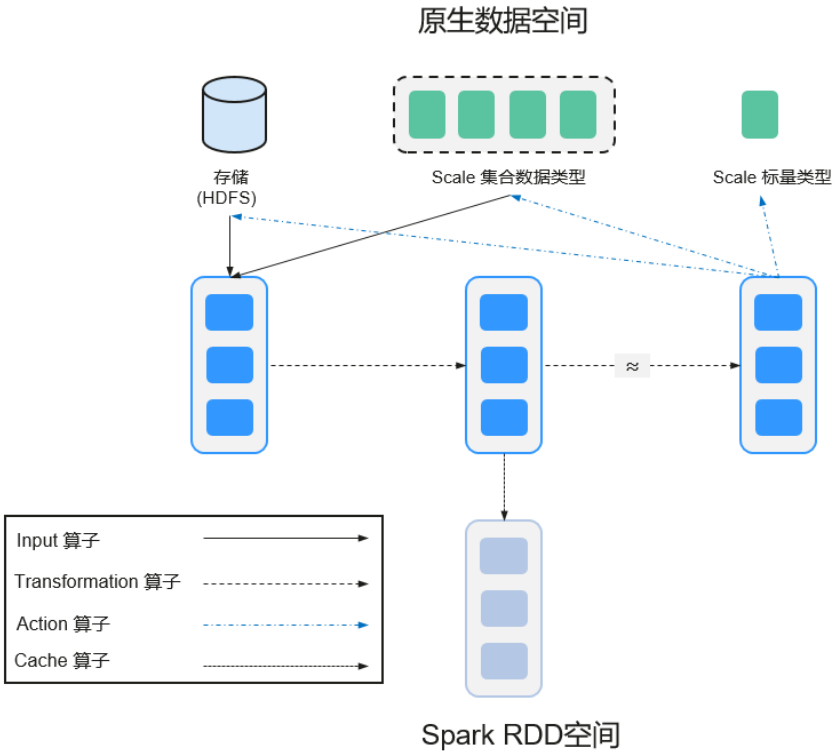
- **窄依赖：**指父RDD的每一个分区最多被一个子RDD的分区所用。
- **宽依赖：**指子RDD的分区依赖于父RDD的所有分区。

窄依赖对优化很有利。逻辑上，每个RDD的算子都是一个fork/join（此join非上文的join算子，而是指同步多个并行任务的barrier）：把计算fork到每个分区，算完后join，然后fork/join下一个RDD的算子。如果直接翻译到物理实现，是很不经济的：一是每一个RDD（即使是中间结果）都需要物化到内存或存储中，费时费空间；二是join作为全局的barrier，是很昂贵的，会被最慢的那个节点拖死。如果子RDD的分区到父RDD的分区是窄依赖，就可以实施经典的fusion优化，把两个fork/join合为一个；如果连续的变换算子序列都是窄依赖，就可以把很多个fork/join并为一个，不但减少了大量的全局barrier，而且无需物化很多中间结果RDD，这将极大地提升性能。Spark把这个叫做流水线（pipeline）优化。

- **Transformation和Action（RDD的操作）**

  对RDD的操作包含Transformation（返回值还是一个RDD）和Action（返回值不是一个RDD）两种。RDD的操作流程如图28-2所示。其中Transformation操作是Lazy的，也就是说从一个RDD转换生成另一个RDD的操作不是马上执行，Spark在遇到Transformations操作时只会记录需要这样的操作，并不会去执行，需要等到有Actions操作的时候才会真正启动计算过程进行计算。Actions操作会返回结果或把RDD数据写到存储系统中。Actions是触发Spark启动计算的动因。

图 28-2 RDD 操作示例



RDD看起来与Scala集合类型没有太大差别，但数据和运行模型大相迥异。

```
val file = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()
errors.count()
```

a. textFile算子从HDFS读取日志文件，返回file（作为RDD）。

b. filter算子筛出带“ERROR”的行，赋给errors（新RDD）。filter算子是一个Transformation操作。

c. cache算子缓存下来以备未来使用。

d. count算子返回errors的行数。count算子是一个Action操作。

Transformation操作可以分为如下几种类型：

- 视RDD的元素为简单元素。

  输入输出一对一，且结果RDD的分区结构不变，主要是map。

  输入输出一对多，且结果RDD的分区结构不变，如flatMap（map后由一个元素变为一个包含多个元素的序列，然后展平为一个个的元素）。

  输入输出一对一，但结果RDD的分区结构发生了变化，如union（两个RDD合为一个，分区数变为两个RDD分区数之和）、coalesce（分区减少）。

  从输入中选择部分元素的算子，如filter、distinct（去除重复元素）、subtract（本RDD有、其他RDD无的元素留下来）和sample（采样）。

- 视RDD的元素为Key-Value对。

  对单个RDD做一对一运算，如mapValues（保持源RDD的分区方式，这与map不同）；

  对单个RDD重排，如sort、partitionBy（实现一致性的分区划分，这个对数据本地性优化很重要）；

对单个RDD基于key进行重组和reduce，如groupByKey、reduceByKey；

对两个RDD基于key进行join和重组，如join、cogroup。

说明

后三种操作都涉及重排，称为shuffle类操作。

Action操作可以分为如下几种：

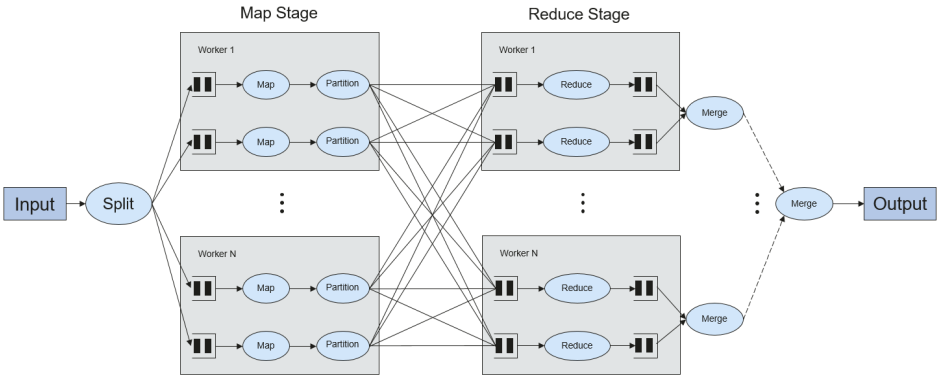
- 生成标量，如count（返回RDD中元素的个数）、reduce、fold/aggregate（返回几个标量）、take（返回前几个元素）。
- 生成Scala集合类型，如collect（把RDD中的所有元素倒入Scala集合类型）、lookup（查找对应key的所有值）。
- 写入存储，如与前文textFile对应的saveAsTextFile。
- 还有一个检查点算子checkpoint。当Lineage特别长时（这在图计算中时常发生），出错时重新执行整个序列要很长时间，可以主动调用checkpoint把当前数据写入稳定存储，作为检查点。

- **Shuffle**

  Shuffle是MapReduce框架中的一个特定的phase，介于Map phase和Reduce phase之间，当Map的输出结果要被Reduce使用时，每一条输出结果需要按key哈希，并且分发到对应的Reducer上去，这个过程就是shuffle。由于shuffle涉及到了磁盘的读写和网络的传输，因此shuffle性能的高低直接影响到了整个程序的运行效率。

  下图清晰地描述了MapReduce算法的整个流程。

  图 28-3 算法流程



  概念上shuffle就是一个沟通数据连接的桥梁，实际上shuffle这一部分是如何实现的呢，下面就以Spark为例讲解shuffle在Spark中的实现。

  Shuffle操作将一个Spark的Job分成多个Stage，前面的stages会包括一个或多个ShuffleMapTasks，最后一个stage会包括一个或多个ResultTask。

- **Spark Application的结构**

  Spark Application的结构可分为两部分：初始化SparkContext和主体程序。

  - 初始化SparkContext：构建Spark Application的运行环境。

    构建SparkContext对象，如：

    ```
    new SparkContext(master, appName, [SparkHome], [jars])
    ```

    参数介绍：

    master：连接字符串，连接方式有local、yarn-cluster、yarn-client等。

appName：构建的Application名称。

SparkHome：集群中安装Spark的目录。

jars：应用程序代码和依赖包。

- 主体程序：处理数据

提交Application的描述请参见：**https://spark.apache.org/docs/3.1.1/submitting-applications.html**

- **Spark shell命令**

  Spark基本shell命令，支持提交Spark应用。命令为：

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

  参数解释：

  --class：Spark应用的类名。

  --master：Spark用于所连接的master，如yarn-client，yarn-cluster等。

  application-jar：Spark应用的jar包的路径。

  application-arguments：提交Spark应用的所需要的参数（可以为空）。

- **Spark JobHistory Server**

  用于监控正在运行的或者历史的Spark作业在Spark框架各个阶段的细节以及提供日志显示，帮助用户更细粒度地去开发、配置和调优作业。

Spark SQL 常用概念

DataSet

DataSet是一个由特定域的对象组成的强类型集合，可通过功能或关系操作并行转换其中的对象。 每个Dataset还有一个非类型视图，即由多个列组成的DataSet，称为DataFrame。

DataFrame是一个由多个列组成的结构化的分布式数据集合，等同于关系数据库中的一张表，或者是R/Python中的data frame。DataFrame是Spark SQL中的最基本的概念，可以通过多种方式创建，例如结构化的数据集、Hive表、外部数据库或者是RDD。

Spark Streaming 常用概念

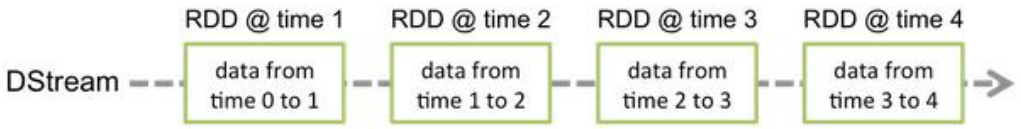
Dstream

DStream(又称Discretized Stream)是Spark Streaming提供的抽象概念。

DStream表示一个连续的数据流，是从数据源获取或者通过输入流转换生成的数据流。从本质上说，一个DStream表示一系列连续的RDD。RDD是一个只读的、可分区的分布式数据集。

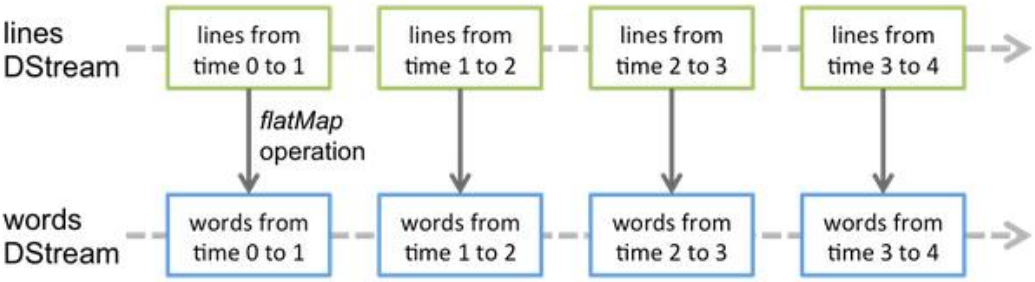
DStream中的每个RDD包含了一个区间的数据。如**图28-4**所示。

图 28-4 DStream 与 RDD 关系



应用到DStream上的所有算子会被转译成下层RDD的算子操作，如图28-5所示。这些下层的RDD转换会通过Spark引擎进行计算。DStream算子隐藏大部分的操作细节，并且提供了方便的High-level API给开发者使用。

图 28-5 DStream 算子转译



Structured Streaming 常用概念

- **Input Source**

  输入数据源，数据源需要支持根据offset重放数据，不同的数据源有不同的容错性。

- **Sink**

  数据输出，Sink要支持幂等性写入操作，不同的sink有不同的容错性。

- **outputMode**

  结果输出模式，当前支持3种输出模：

  - Complete Mode：整个更新的结果集都会写入外部存储。整张表的写入操作将由外部存储系统的连接器完成。
  - Append Mode：当时间间隔触发时，只有在Result Table中新增加的数据行会被写入外部存储。这种方式只适用于结果集中已经存在的内容不希望发生改变的情况下，如果已经存在的数据会被更新，不适合适用此种方式。
  - Update Mode：当时间间隔触发时，只有在Result Table中被更新的数据才会被写入外部存储系统。注意，和Complete Mode方式的不同之处是不更新的结果集不会写入外部存储。

- **Trigger**

  输出触发器，当前支持以下几种trigger：

  - 默认：以微批模式执行，每个批次完成后自动执行下个批次。
  - 固定间隔：固定时间间隔执行。
  - 一次执行：只执行一次query，完成后退出。
  - 连续模式：实验特性，可实现低至1ms延迟的流处理（推荐100ms）。

Structured Streaming支持微批模式和连续模式。微批模式不能保证对数据的低延迟处理，但是在相同时间下有更大的吞吐量；连续模式适合毫秒级的数据处理延迟，当前暂时还属于实验特性。

说明

在当前版本中，若需要使用流流Join功能，则output模式只能选择append模式。

图 28-6 微批模式运行过程简图

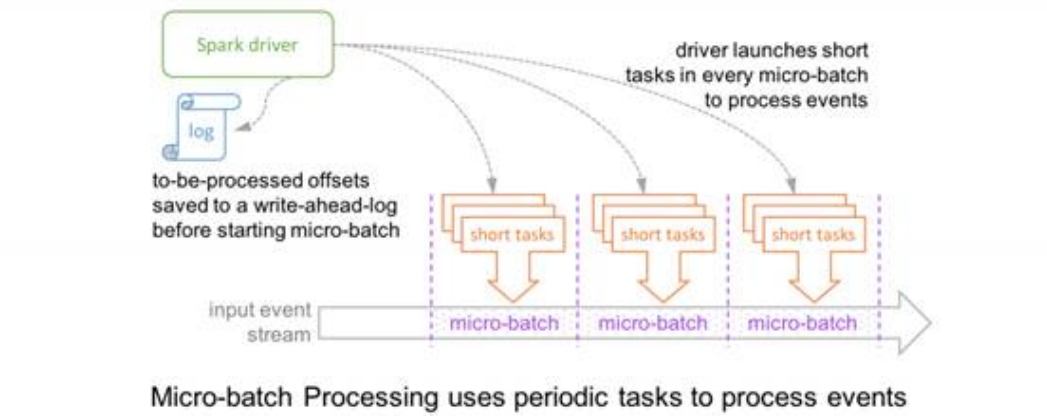
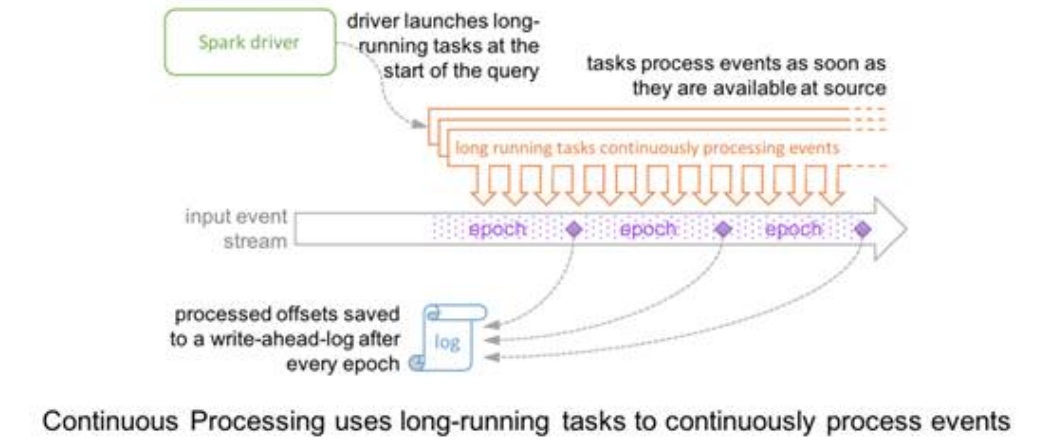


图 28-7 连续模式运行过程简图



28.2 Spark 应用开发流程介绍

Spark包含Spark Core、Spark SQL和Spark Streaming三个组件，其应用开发流程都是相同的。

开发流程中各阶段的说明如图28-8和表28-2所示。

图 28-8 Spark 应用程序开发流程

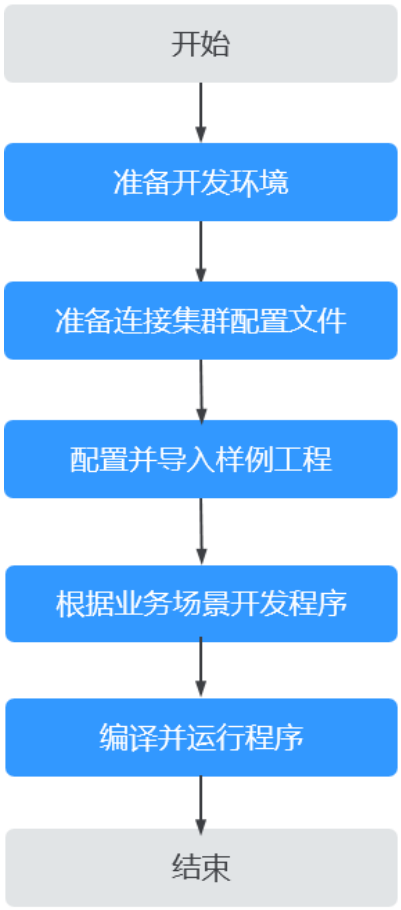


表 28-2 Spark 应用开发的流程说明

阶段	说明	参考文档
准备开发环境	Spark的应用程序支持使用Scala、Java、Python三种语言进行开发。推荐使用IDEA工具，请根据指导完成不同语言的开发环境配置。Spark的运行环境即Spark客户端，请根据指导完成客户端的安装和配置。	**准备Spark本地应用开发环境**
准备连接集群配置文件	应用程序开发或运行过程中，需通过集群相关配置文件信息连接MRS集群，配置文件通常包括集群组件信息文件以及用于安全认证的用户文件，可从已创建好的MRS集群中获取相关内容。 用于程序调测或运行的节点，需要与MRS集群内节点网络互通，同时配置hosts域名信息。	**准备Spark连接集群配置文件**
准备工程	Spark提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。或者您可以根据指导，新建一个Spark工程。	**导入并配置Spark样例工程** **新建Spark样例工程（可选）**

阶段	说明	参考文档
根据场景开发工程	提供了Scala、Java、Python三种不同语言的样例工程，还提供了Streaming、SQL、JDBC客户端程序以及Spark on HBase四种不同场景的样例工程。 帮助用户快速了解Spark各部件的编程接口。	开发Spark应用
编译并运行程序	指导用户将开发好的程序编译并提交运行。	在Linux环境中编包并运行Spark程序

28.3 Spark2x 样例工程介绍

MRS样例工程获取地址为https://github.com/huaweicloud/huaweicloud-mrs-example，切换分支为与MRS集群相匹配的版本分支，然后下载压缩包到本地后解压，即可获取各组件对应的样例代码工程。

当前MRS提供以下Spark2x相关样例工程：

表 28-3 Spark2x 相关样例工程

样例工程位置	描述
sparknormal-examples/SparkHbasetoCarbonJavaExample	Spark同步HBase数据到CarbonData的应用开发样例代码。 本示例工程中，应用将数据实时写入HBase，用于点查业务。数据每隔一段时间批量同步到CarbonData表中，用于分析型查询业务。
sparknormal-examples/SparkHbasetoHbaseJavaExample	Spark从HBase读取数据再写入HBase的Java/Scala/Python示例程序。 本示例工程中，Spark应用程序实现两个HBase表数据的分析汇总。
sparknormal-examples/SparkHbasetoHbasePythonExample	
sparknormal-examples/SparkHbasetoHbaseScalaExample	
sparknormal-examples/SparkHivetoHbaseJavaExample	Spark从Hive读取数据再写入到HBase的应用开发样例代码。
sparknormal-examples/SparkHivetoHbasePythonExample	
sparknormal-examples/SparkHivetoHbaseScalaExample	

样例工程位置	描述
sparknormal-examples/SparkJavaExample	Spark Core任务的Java/Python/Scala示例程序。 本工程应用程序实现从HDFS上读取文本数据并计算分析。
sparknormal-examples/SparkPythonExample	
sparknormal-examples/SparkSQLJavaExample	
sparknormal-examples/SparkLauncherJavaExample	使用Spark Launcher提交作业的Java/Scala示例程序。 本工程应用程序通过org.apache.spark.launcher.SparkLauncher类采用Java/Scala命令方式提交Spark应用。
sparknormal-examples/SparkLauncherScalaExample	
sparknormal-examples/SparkOnClickHouseJavaExample	Spark通过ClickHouse JDBC的原生接口，以及Spark JDBC驱动，实现对ClickHouse数据库和表的创建、查询、插入等操作样例代码。
sparknormal-examples/SparkOnClickHousePythonExample	
sparknormal-examples/SparkOnClickHouseScalaExample	
sparknormal-examples/SparkOnHbaseJavaExample	Spark on HBase场景的Java/Scala/Python示例程序。 本工程应用程序以数据源的方式去使用HBase，将数据以Avro格式存储在HBase中，并从中读取数据以及对读取的数据进行过滤等操作。
sparknormal-examples/SparkOnHbasePythonExample	
sparknormal-examples/SparkOnHbaseScalaExample	
sparknormal-examples/SparkOnHudiJavaExample	Spark on Hudi场景的Java/Scala/Python示例程序。 本工程应用程序使用Spark操作Hudi执行插入数据、查询数据、更新数据、增量查询、特定时间点查询、删除数据等操作。
sparknormal-examples/SparkOnHudiPythonExample	
sparknormal-examples/SparkOnHudiScalaExample	
sparknormal-examples/SparkSQLJavaExample	Spark SQL任务的Java/Python/Scala示例程序。 本工程应用程序实现从HDFS上读取文本数据并计算分析。
sparknormal-examples/SparkSQLPythonExample	
sparknormal-examples/SparkSQLScalaExample	

样例工程位置	描述
sparknormal-examples/SparkStreamingKafka010JavaExample	Spark Streaming从Kafka接收数据并进行统计分析的Java/Scala示例程序。 本工程应用程序实时累加计算Kafka中的流数据，统计每个单词的记录总数。
sparknormal-examples/SparkStreamingKafka010PythonExample	
sparknormal-examples/SparkStreamingtoHbaseJavaExample010	Spark Streaming读取Kafka数据并写入HBase的Java/Scala/Python示例程序。 本工程应用程序每5秒启动一次任务，读取Kafka中的数据并更新到指定的HBase表中。
sparknormal-examples/SparkStreamingtoHbasePythonExample010	
sparknormal-examples/SparkStreamingtoHbaseScalaExample010	
sparknormal-examples/SparkStructuredStreamingJavaExample	在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。
sparknormal-examples/SparkStructuredStreamingPythonExample	
sparknormal-examples/SparkStructuredStreamingScalaExample	
sparknormal-examples/SparkThriftServerJavaExample	通过JDBC访问Spark SQL的Java/Scala示例程序。 本示例中，用户自定义JDBCServer的客户端，使用JDBC连接来进行表的创建、数据加载、查询和删除。
sparknormal-examples/SparkThriftServerScalaExample	
sparknormal-examples/StructuredStreamingADScalaExample	使用Structured Streaming，从kafka中读取广告请求数据、广告展示数据、广告点击数据，实时获取广告有效展示统计数据和广告有效点击统计数据，将统计结果写入kafka中。
sparknormal-examples/StructuredStreamingStateScalaExample	Spark结构流应用中，跨批次统计每个session期间发生了多少次event以及本session的开始和结束timestamp；同时输出本批次被更新状态的session。

28.4 准备 Spark 应用开发环境

28.4.1 准备 Spark 本地应用开发环境

在进行应用开发时，要准备的开发和运行环境如表28-4所示。

表 28-4 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端：Oracle JDK：支持1.8版本；IBM JDK：支持1.8.5.11版本。 • TaiShan客户端：OpenJDK：支持1.8.0_272版本。 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls。
安装和配置IntelliJ IDEA	用于开发Spark应用程序的工具，建议使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。 • 不同的IntelliJ IDEA不要使用相同的workspace和相同路径下的示例工程。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
安装Scala	Scala开发环境的基本配置。版本要求：2.12.10。
安装Scala插件	Scala开发环境的基本配置。版本要求：2018.2.11或其他兼容版本。
安装Editra	Python开发环境的编辑器，用于编写Python程序。或者使用其他编写Python应用程序的IDE。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。
安装Python	版本要求不低于3.6。

28.4.2 准备 Spark 连接集群配置文件

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 场景一：准备本地Windows开发环境调测程序所需配置文件。
  a. 登录FusionInsight Manager页面，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

     例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig_ConfigFiles.tar”，继续解压该文件。
  b. 进入客户端配置文件解压路径“*\Spark\config”，获取Spark配置文件，并所有的配置文件导入到Spark样例工程的配置文件目录中（通常为“resources”文件夹）。
  c. 复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中。

     说明

     - 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要确保本地节点能与“hosts”文件中所列出的各主机在网络上互通。
     - 如果当前节点与MRS集群所在网络平面不互通，可以通过绑定EIP的方式访问MRS集群。
     - Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。
- 场景二：准备Linux环境运行程序所需配置文件。
  a. 在节点中安装MRS集群客户端，例如客户端安装目录为“/opt/client”。
  b. 获取配置文件：
     i. 登录FusionInsight Manager，在“主页”右上方选择“更多 > 下载客户端”，“选择客户端类型”设置为“仅配置文件”，勾选“仅保存到如下路径”，单击“确定”，下载客户端配置文件至集群主OMS点。
     ii. 以**root**登录主OMS节点，进入客户端配置文件所在路径（默认为“/tmp/FusionInsight-Client/”），解压软件包后获取“*\Spark\config”路径下的配置文件。并将所有的配置文件放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

     例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

     cd /tmp/FusionInsight-Client

     tar -xvf FusionInsight_Cluster_1_Services_Client.tar

     tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

     cd FusionInsight_Cluster_1_Services_ClientConfig

     cp -r Spark/config/* /opt/client/conf

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

28.4.3 导入并配置 Spark 样例工程

操作场景

Spark针对多个场景提供样例工程，包含Java样例工程和Scala样例工程等，帮助客户快速学习Spark工程。

针对Java和Scala不同语言的工程，其导入方式相同。使用Python开发的样例工程不需要导入，直接打开Python文件（*.py）即可。

以下操作步骤以导入Java样例代码为例。操作流程如图28-9所示。

图 28-9 导入样例工程流程

前提条件

- 确保本地环境的时间与MRS集群的时间差要小于5分钟，若无法确定，请联系系统管理员。MRS集群的时间可通过FusionInsight Manager页面右下角查看。
- 已准备开发环境及MRS集群相关配置文件，详情请参考**准备Spark连接集群配置文件**。

操作步骤

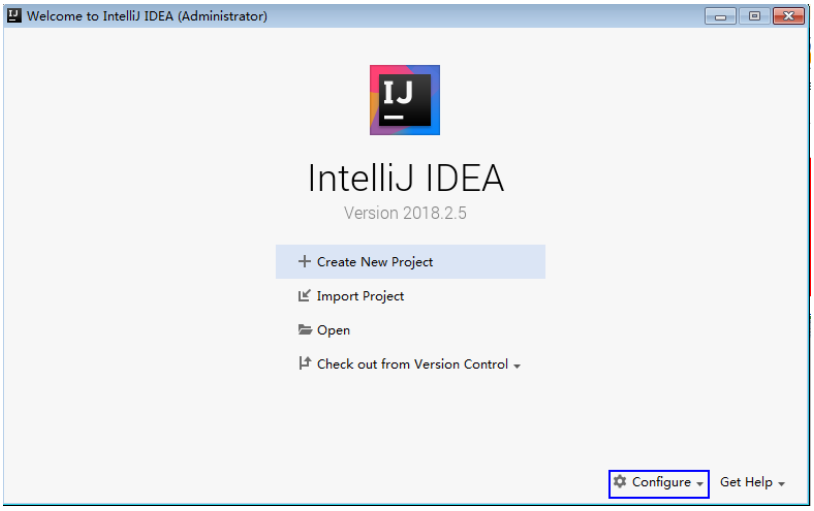
步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“spark-examples”目录下的“sparknormal-examples”文件夹中的Scala、Spark Streaming等多个样例工程。

步骤2 若需要在本地Windows调测Spark样例代码，需参考**准备Spark连接集群配置文件**获取各样例项目所需的配置文件，并手动将配置文件导入到Spark样例工程的配置文件目录中。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA中配置JDK。

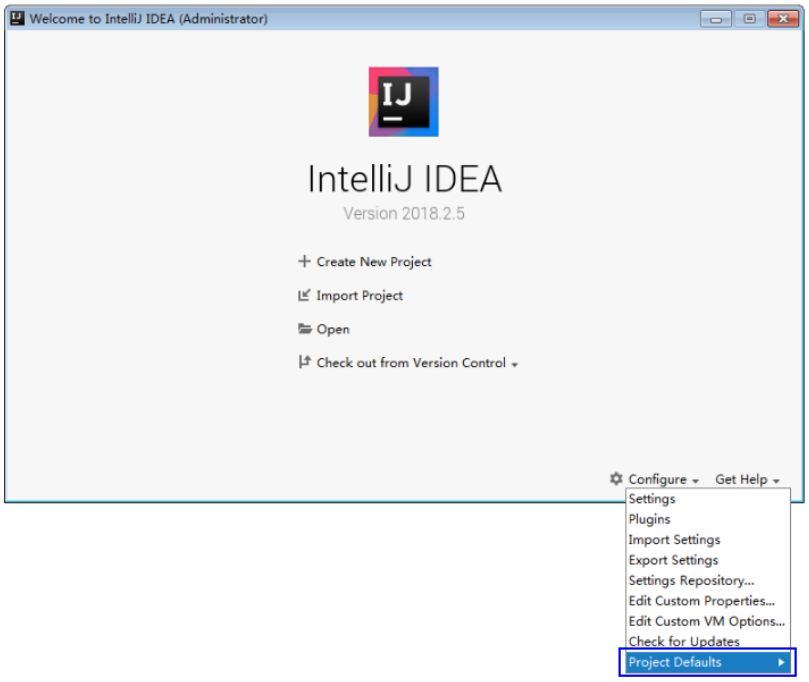
1. 打开IntelliJ IDEA，选择“Configure”。

图 28-10 Quick Start



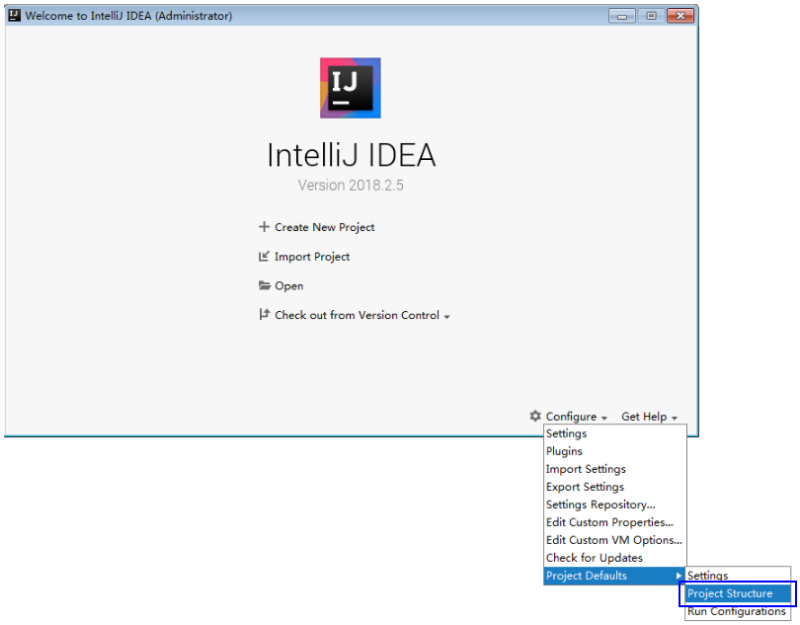
2. 在“Configure”下拉菜单中单击“Project Defaults”。

图 28-11 Configure



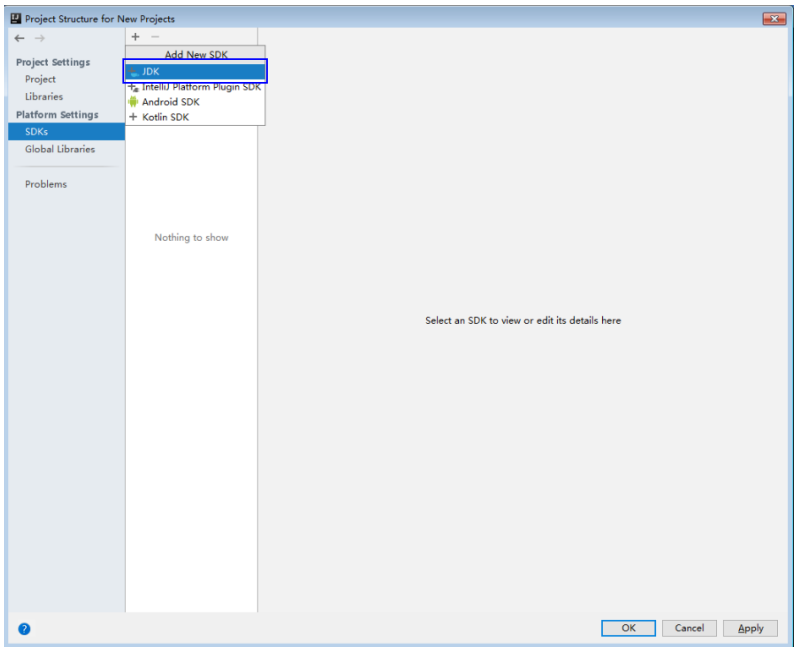
3. 在“Project Defaults”菜单中选择“Project Structure”。

图 28-12 Project Defaults



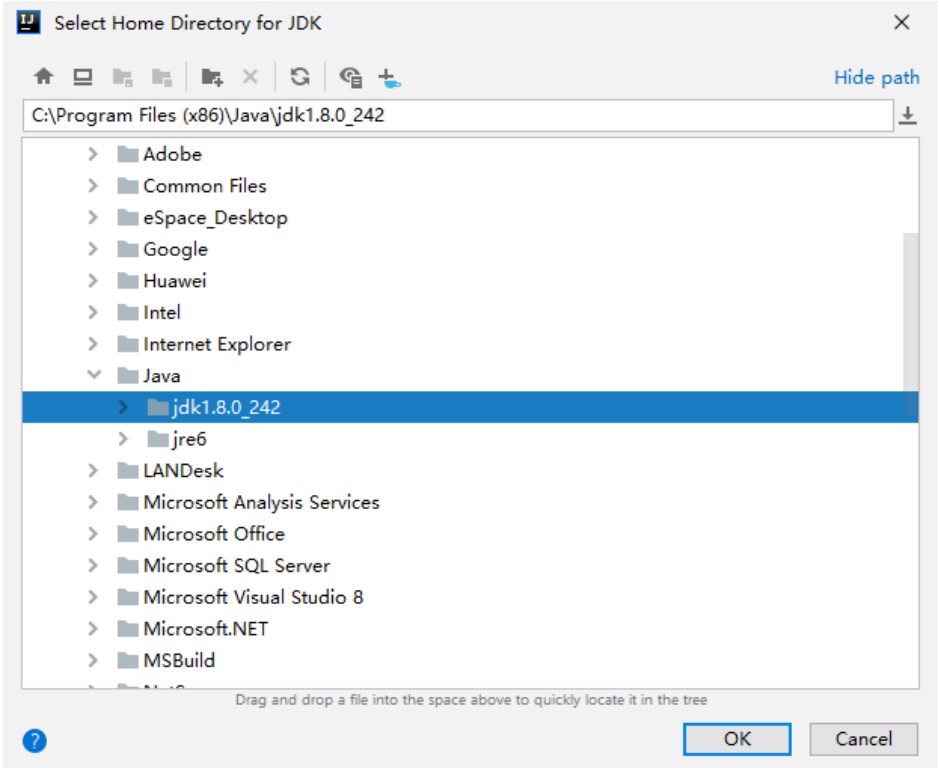
4. 在打开的“Project Structure”页面中，选择“SDKs”，单击加号添加JDK。

图 28-13 添加 JDK



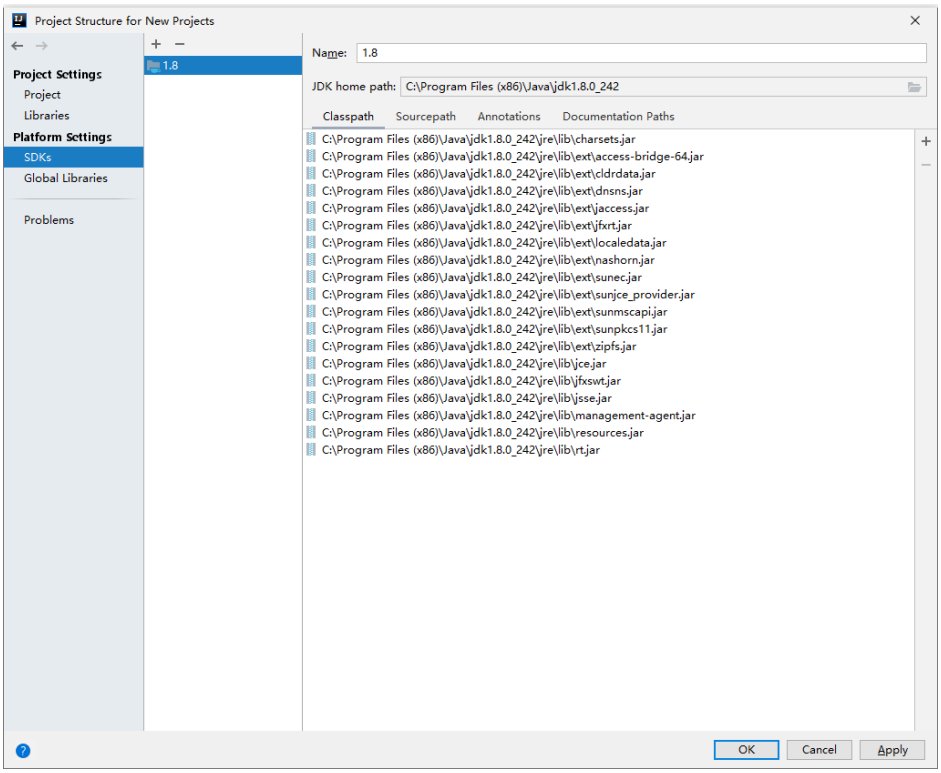
5. 在弹出的“Select Home Directoty for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 28-14 选择 JDK 目录



6. 完成JDK选择后，单击“OK”完成配置。

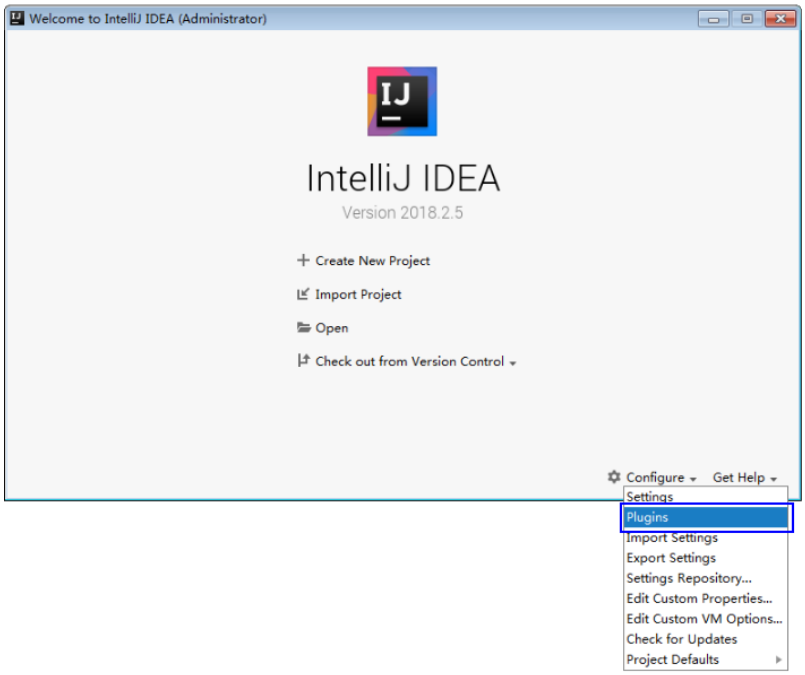
图 28-15 完成 JDK 配置



步骤4 （可选）如果是Scala开发环境，还需要在IntelliJ IDEA中安装Scala插件。

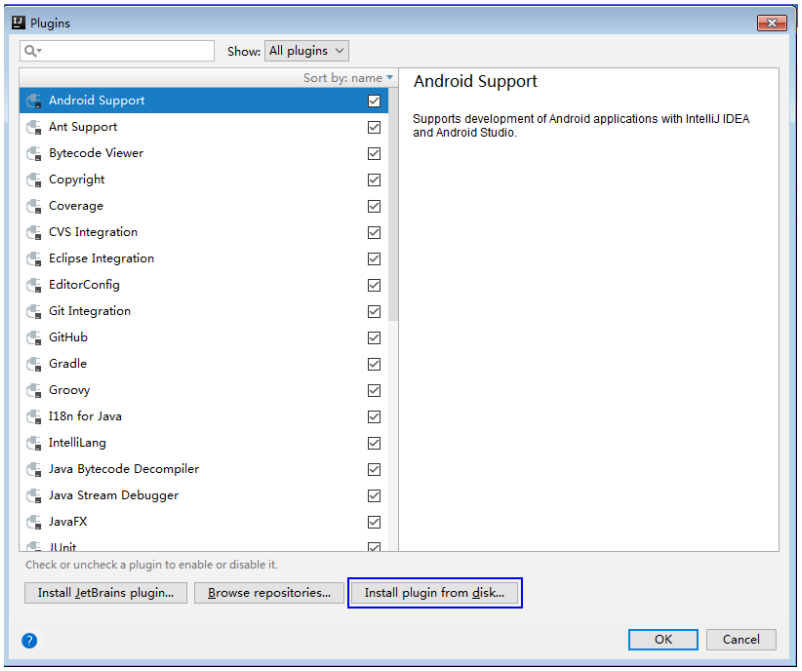
1. 在“Configure”下拉菜单中，单击“Plugins”。

图 28-16 Plugins

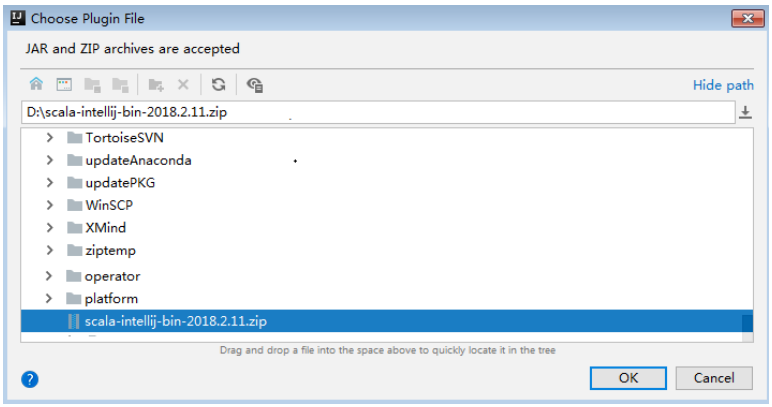


2. 在“Plugins”页面，选择“Install plugin from disk”。

图 28-17 Install plugin from disk

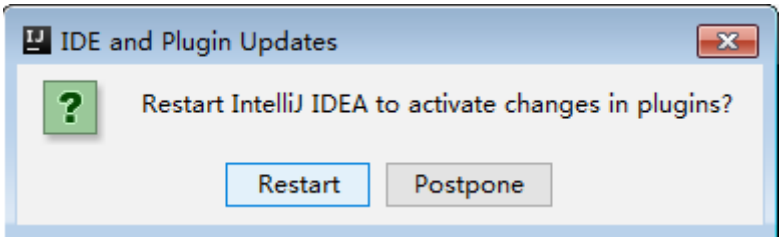


3. 在“Choose Plugin File”页面，选择对应版本的Scala插件包，单击“OK”。



4. 在“Plugins”页面，单击“Apply”安装Scala插件。
5. 在弹出的“Plugins Changed”页面，单击“Restart”，使配置生效。

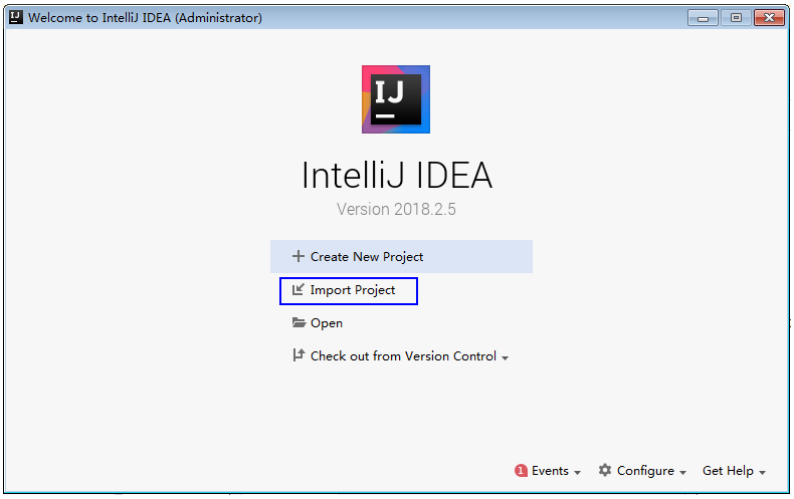
图 28-18 Plugins Changed



步骤5 将Java样例工程导入到IDEA中。

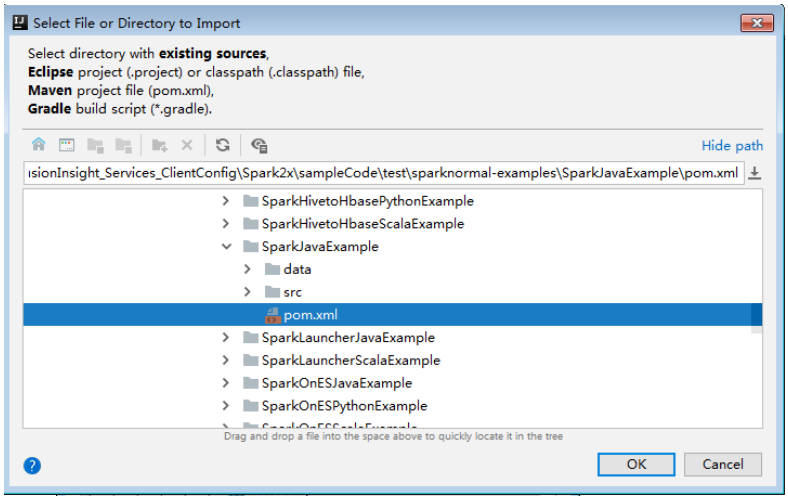
1. 打开IntelliJ IDEA。在“Quick Start”页面选择“Import Project”。
   或者，针对已使用过的IDEA工具，您可以从IDEA主界面直接添加。选择“File > Import project...”导入工程。

图 28-19 Import Project（Quick Start 页面）



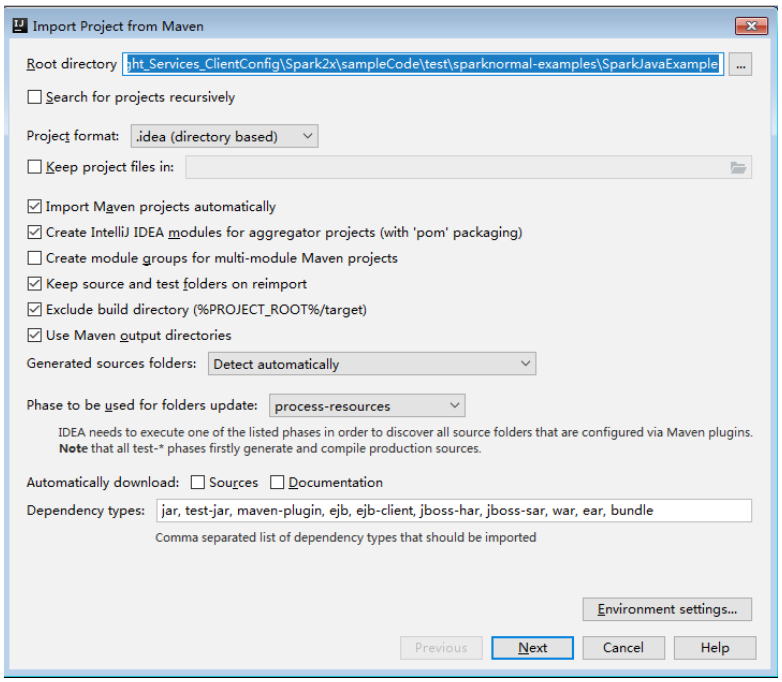
2. 选择需导入的样例工程存放路径及其pom文件，然后单击“OK”。

图 28-20 Select File or Directory to Import



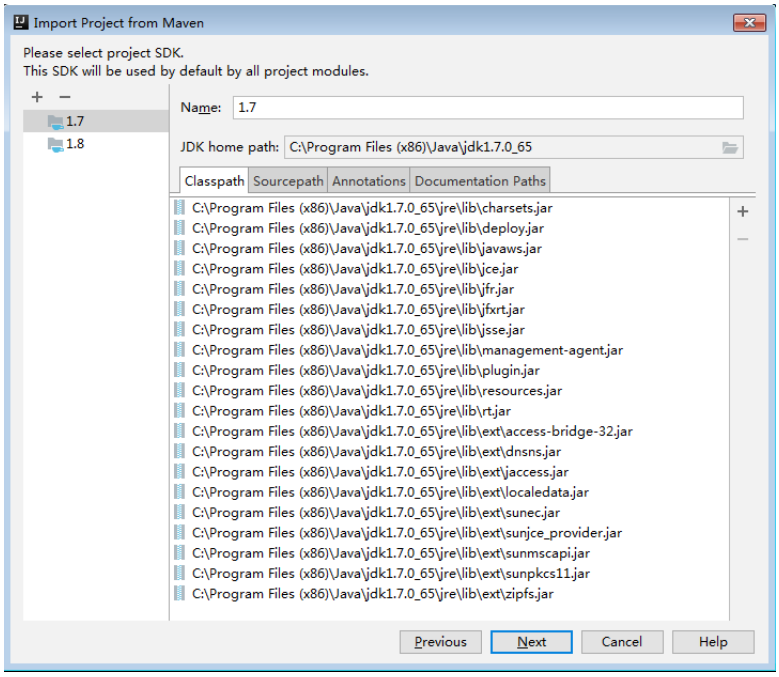
3. 确认导入路径和名称，单击“Next”。

图 28-21 Import Project from Maven



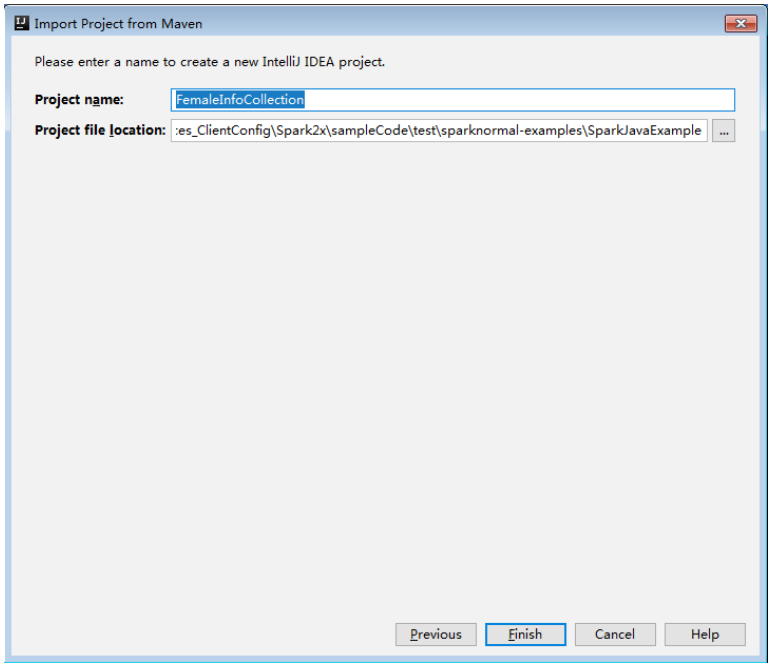
4. 选择需要导入的工程，然后单击“Next”。
5. 确认工程所用JDK，然后单击“Next”。

图 28-22 Select project SDK



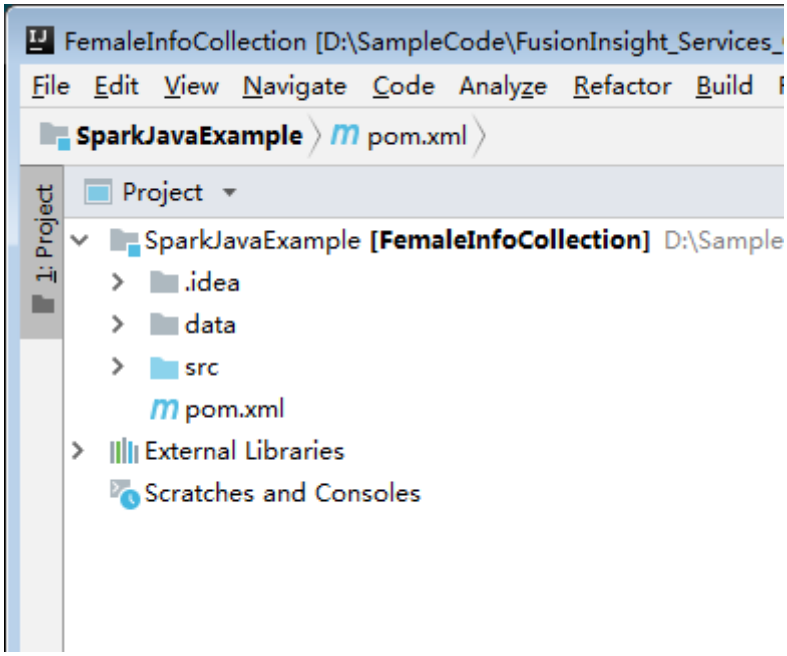
6. 确认工程名称和路径，单击“Finish”完成导入。

图 28-23 Confirm the project name and file location



7. 导入完成后，IDEA主页显示导入的样例工程。

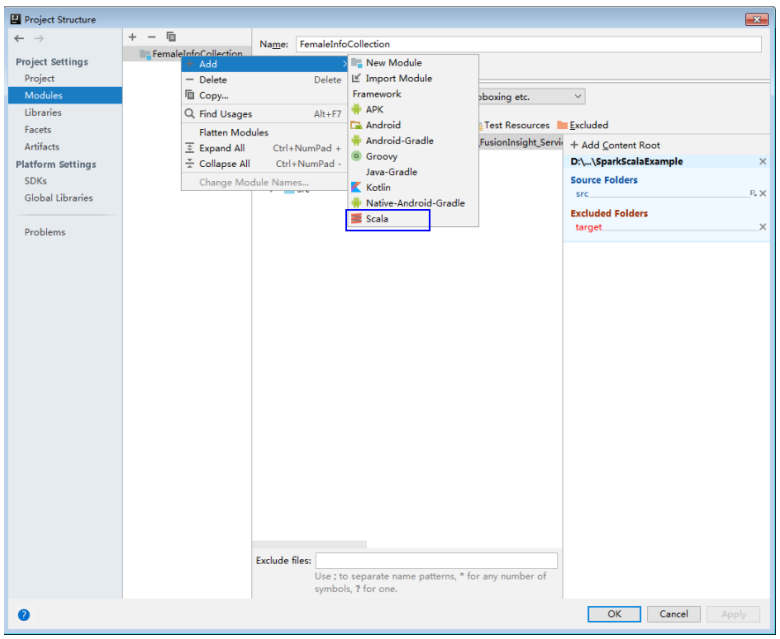
图 28-24 已导入工程



步骤6 （可选）如果导入Scala语言开发的样例程序，还需要为工程设置语言。

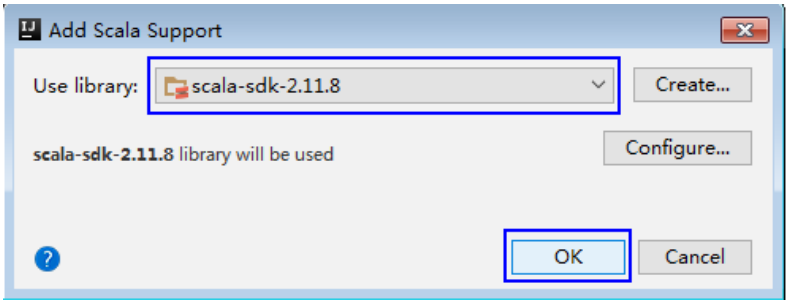
1. 在IDEA主页，选择“File > Project Structures...”进入“Project Structure”页面。
2. 选择“Modules”，选中工程名称，然后右键选择“Add > Scala”。

图 28-25 选择 Scala 语言



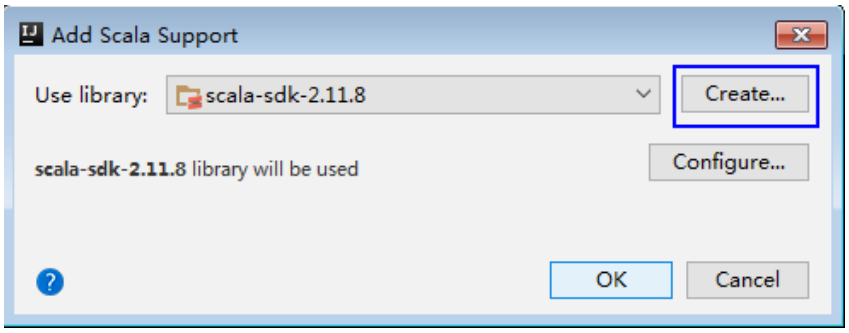
3. 当IDEA可以识别出Scala SDK时，在设置界面，选择编译的依赖jar包，然后单击“OK”应用设置。

图 28-26 Add Scala Support



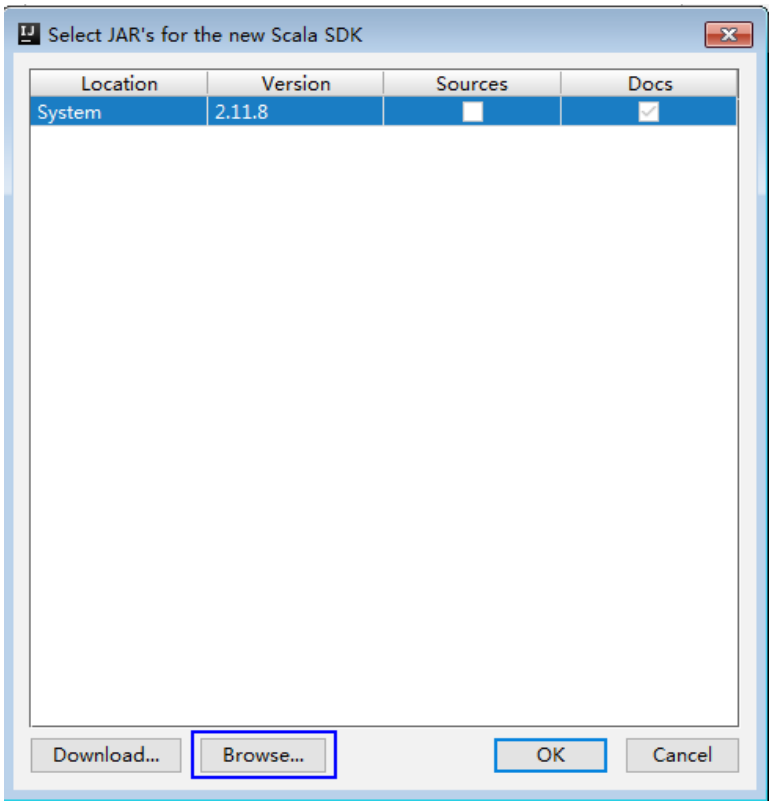
4. 当系统无法识别出Scala SDK时，需要自行创建。
   a. 单击“Create...”。

图 28-27 Create...



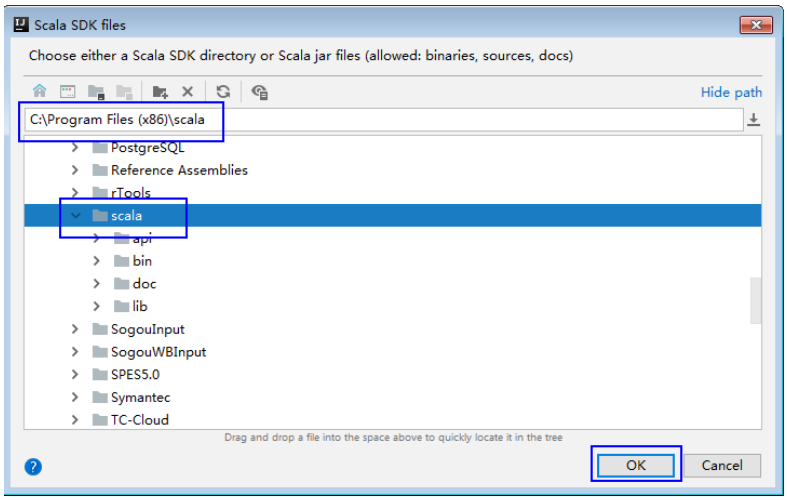
   b. 在“Select JAR's for the new Scala SDK”页面单击“Browse...”。

图 28-28 Select JAR's for the new Scala SDK



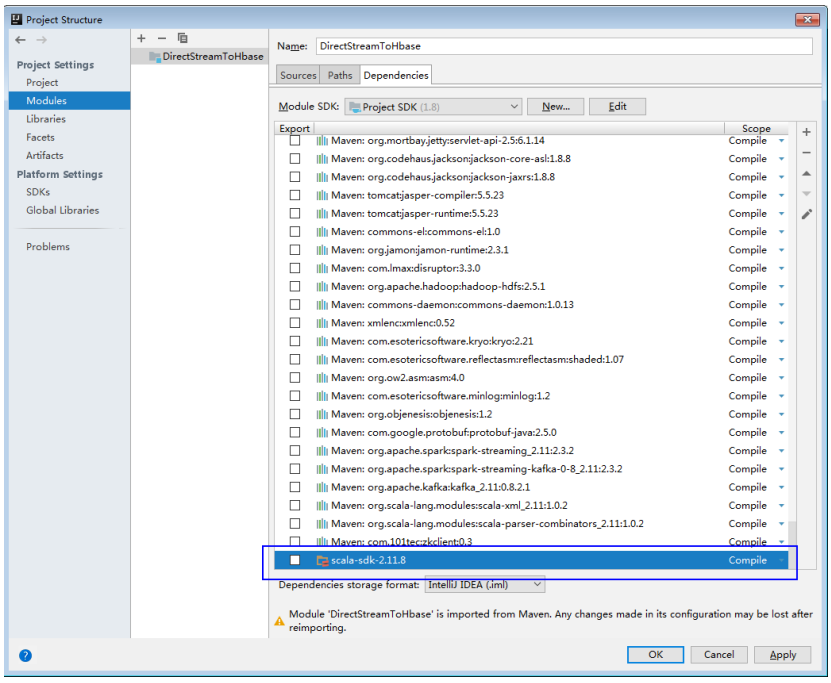
c. 在“Scala SDK files”页面选择scala sdk目录，单击“OK”。

图 28-29 Scala SDK files



5. 设置成功，单击“OK”保存设置。

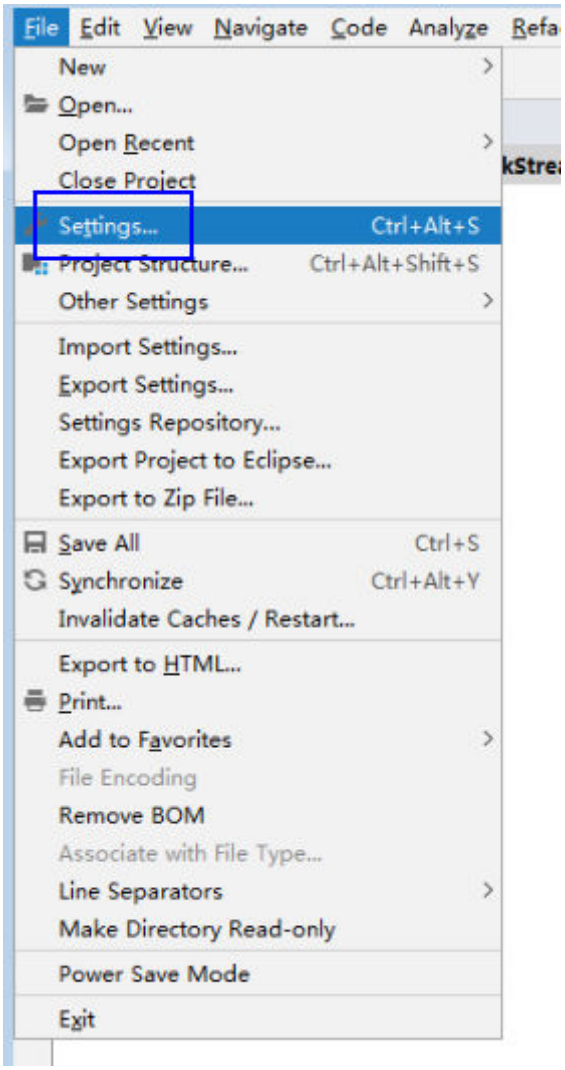
图 28-30 设置成功



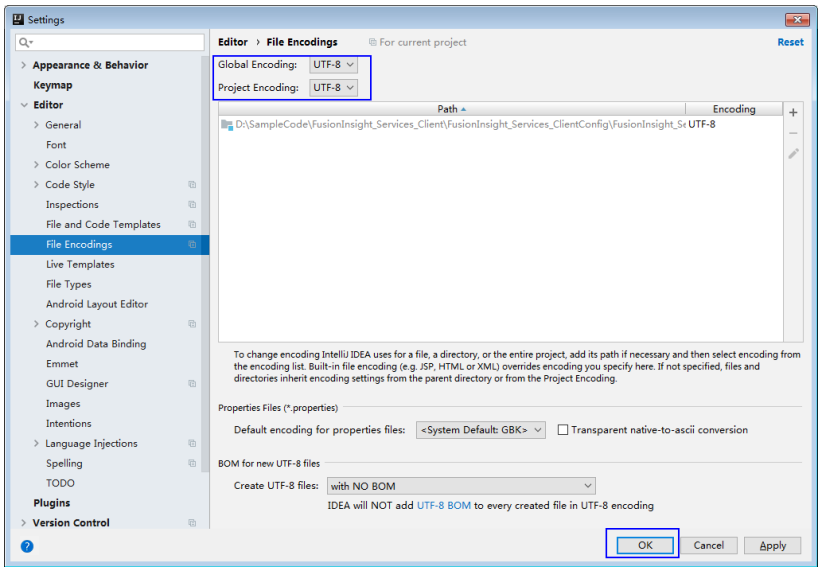
步骤7 设置IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IDEA首页，选择“File > Settings...”。

图 28-31 选择 Settings



2. 编码配置。
   a. 在“Settings”页面，展开“Editor”，选择“File Encodings”。
   b. 分别在右侧的“Global Encoding”和“Project Encoding”的下拉框中，选择“UTF-8”。
   c. 单击“Apply”应用配置。
   d. 单击“OK”完成编码配置。



----结束

样例代码路径说明

表 28-5 样例代码路径说明

样例代码项目	样例名称	样例语言
SparkJavaExample	Spark Core程序	Java
SparkScalaExample	Spark Core程序	Scala
SparkPythonExample	Spark Core程序	Python
SparkSQLJavaExample	Spark SQL程序	Java
SparkSQLScalaExample	Spark SQL程序	Scala
SparkSQLPythonExample	Spark SQL程序	Python
SparkThriftServerJavaExample	通过JDBC访问Spark SQL的程序	Java
SparkThriftServerScalaExample	通过JDBC访问Spark SQL的程序	Scala
SparkOnHbaseJavaExample-AvroSource	Spark on HBase 程序-操作Avro格式数据	Java
SparkOnHbaseScalaExample-AvroSource	Spark on HBase 程序-操作Avro格式数据	Scala
SparkOnHbasePythonExample-AvroSource	Spark on HBase 程序-操作Avro格式数据	Python
SparkOnHbaseJavaExample-HbaseSource	Spark on HBase 程序-操作HBase数据源	Java

样例代码项目	样例名称	样例语言
SparkOnHbaseScalaExample-HbaseSource	Spark on HBase 程序-操作HBase数据源	Scala
SparkOnHbasePythonExample-HbaseSource	Spark on HBase 程序-操作HBase数据源	Python
SparkOnHbaseJavaExample-JavaHBaseBulkPutExample	Spark on HBase 程序-BulkPut接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkPutExample	Spark on HBase 程序-BulkPut接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkPutExample	Spark on HBase 程序-BulkPut接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseBulkGetExample	Spark on HBase 程序-BulkGet接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkGetExample	Spark on HBase 程序-BulkGet接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkGetExample	Spark on HBase 程序-BulkGet接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseBulkDeleteExample	Spark on HBase 程序-BulkDelete接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkDeleteExample	Spark on HBase 程序-BulkDelete接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkDeleteExample	Spark on HBase 程序-BulkDelete接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseBulkLoadExample	Spark on HBase 程序-BulkLoad接口使用	Java
SparkOnHbaseScalaExample-HBaseBulkLoadExample	Spark on HBase 程序-BulkLoad接口使用	Scala
SparkOnHbasePythonExample-HBaseBulkLoadExample	Spark on HBase 程序-BulkLoad接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseForEachPartitionExample	Spark on HBase 程序-foreachPartition接口使用	Java
SparkOnHbaseScalaExample-HBaseForEachPartitionExample	Spark on HBase 程序-foreachPartition接口使用	Scala
SparkOnHbasePythonExample-HBaseForEachPartitionExample	Spark on HBase 程序-foreachPartition接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseDistributedScanExample	Spark on HBase 程序-分布式Scan HBase表	Java

样例代码项目	样例名称	样例语言
SparkOnHbaseScalaExample-HBaseDistributedScanExample	Spark on HBase 程序-分布式Scan HBase表	Scala
SparkOnHbasePythonExample-HBaseDistributedScanExample	Spark on HBase 程序-分布式Scan HBase表	Python
SparkOnHbaseJavaExample-JavaHBaseMapPartitionExample	Spark on HBase 程序-mapPartitions接口使用	Java
SparkOnHbaseScalaExample-HBaseMapPartitionExample	Spark on HBase 程序-mapPartitions接口使用	Scala
SparkOnHbasePythonExample-HBaseMapPartitionExample	Spark on HBase 程序-mapPartitions接口使用	Python
SparkOnHbaseJavaExample-JavaHBaseStreamingBulkPutExample	Spark on HBase 程序-SparkStreaming批量写入HBase表	Java
SparkOnHbaseScalaExample-HBaseStreamingBulkPutExample	Spark on HBase 程序-SparkStreaming批量写入HBase表	Scala
SparkOnHbasePythonExample-HBaseStreamingBulkPutExample	Spark on HBase 程序-SparkStreaming批量写入HBase表	Python
SparkHbasetoHbaseJavaExample	从HBase读取数据再写入HBase	Java
SparkHbasetoHbaseScalaExample	从HBase读取数据再写入HBase	Scala
SparkHbasetoHbasePythonExample	从HBase读取数据再写入HBase	Python
SparkHivetoHbaseJavaExample	从Hive读取数据再写入HBase	Java
SparkHivetoHbaseScalaExample	从Hive读取数据再写入HBase	Scala
SparkHivetoHbasePythonExample	从Hive读取数据再写入HBase	Python
SparkStreamingKafka010JavaExample	Spark Streaming对接Kafka0-10程序	Java
SparkStreamingKafka010ScalaExample	Spark Streaming对接Kafka0-10程序	Scala
SparkStructuredStreamingJavaExample	Structured Streaming程序	Java
SparkStructuredStreamingScalaExample	Structured Streaming程序	Scala

样例代码项目	样例名称	样例语言
SparkStructuredStreamingPythonExample	Structured Streaming程序	Python
StructuredStreamingADScalaExample	Structured Streaming流流Join	Scala
StructuredStreamingStateScalaExample	Structured Streaming 状态操作	Scala
SparkOnHudiJavaExample	使用Spark执行Hudi基本操作	Java
SparkOnHudiPythonExample	使用Spark执行Hudi基本操作	Python
SparkOnHudiScalaExample	使用Spark执行Hudi基本操作	Scala

28.4.4 新建 Spark 样例工程（可选）

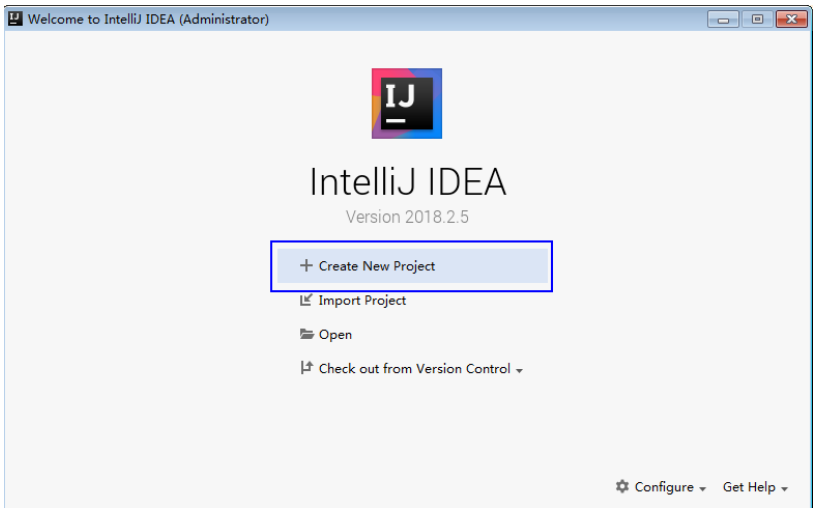
操作场景

除了导入Spark样例工程，您还可以使用IDEA新建一个Spark工程。如下步骤以创建一个Scala工程为例进行说明。

操作步骤

步骤1 打开IDEA工具，选择“Create New Project”。

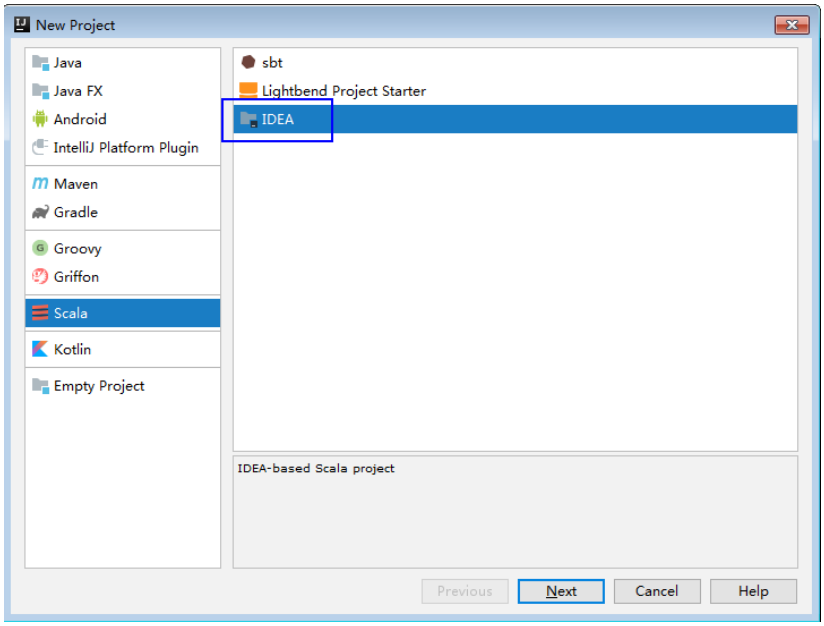
图 28-32 创建工程



步骤2 在“New Project”页面，选择“Scala”开发环境，并选择“IDEA”，然后单击“Next”。

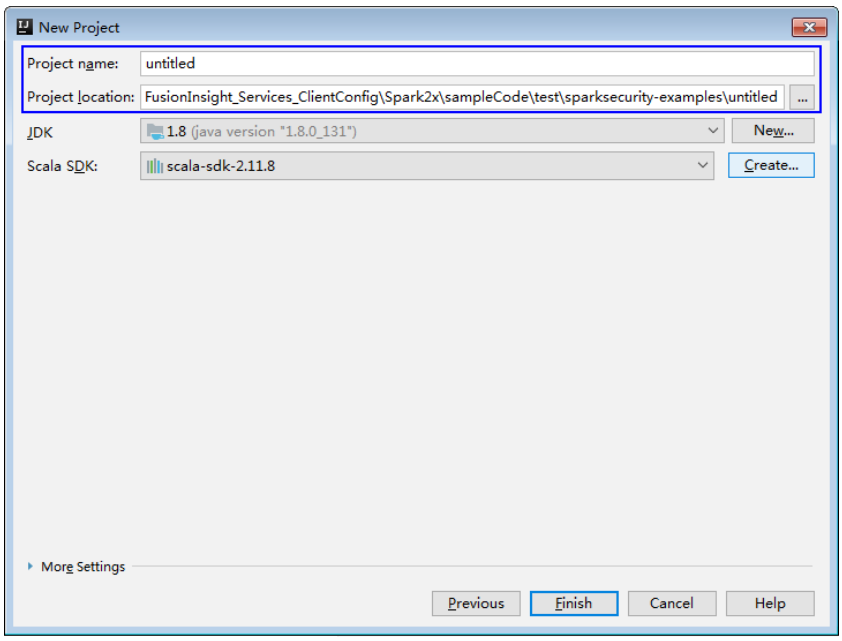
如果您需要新建Java语言的工程，选择对应参数即可。

图 28-33 选择开发环境



步骤3 在工程信息页面，填写工程名称和存放路径，设置JDK版本、Scala SDK版本，然后单击“Finish”完成工程创建。

图 28-34 填写工程信息



----结束

28.4.5 配置 Spark Python3 样例工程

操作场景

为了运行MRS产品Spark2x组件的Python3接口样例代码，需要完成下面的操作。

操作步骤

步骤1 客户端机器必须安装有Python3，其版本不低于3.6。

在客户端机器的命令行终端输入**python3**可查看Python版本号。如下显示Python版本为3.8.2。

```
Python 3.8.2 (default, Jun 23 2020, 10:26:03)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

步骤2 客户端机器必须安装有setuptools，版本为47.3.1。

具体软件，请到对应的官方网站获取。

https://pypi.org/project/setuptools/#files

将下载的setuptools压缩文件复制到客户端机器上，解压后进入解压目录，在客户端机器的命令行终端执行**python3 setup.py install**。

如下内容表示安装setuptools的47.3.1版本成功。

```
Finished processing dependencies for setuptools==47.3.1
```

步骤3 安装Python客户端到客户端机器。

1. 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\hive-examples”目录下的样例工程文件夹“python3-examples”。
2. 进入“python3-examples”文件夹。
3. 根据python3的版本，选择进入“dependency_python3.6”或“dependency_python3.7”或“dependency_python3.8”文件夹。
4. 执行**whereis easy_install**命令，找到easy_install程序路径。如果有多个路径，使用**easy_install --version**确认选择setuptools对应版本的easy_install，如/usr/local/bin/easy_install
5. 使用对应的easy_install命令，依次安装dependency_python3.x文件夹下的egg文件。如：

   /usr/local/bin/easy_install future-0.18.2-py3.8.egg

   输出以下关键内容表示安装egg文件成功。

   ```
   Finished processing dependencies for future==0.18.2
   ```

----结束

28.5 开发 Spark 应用

28.5.1 Spark Core 样例程序

28.5.1.1 Spark Core 样例程序开发思路

场景说明

假定用户有某个周末网民网购停留时间的日志，基于某些业务要求，要求开发Spark应用程序实现如下功能：

- 统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

  log1.txt：周六网民停留日志

  ```
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  FangBo,female,50
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,50
  CaiXuyu,female,50
  FangBo,female,60
  ```

  log2.txt：周日网民停留日志

  ```
  LiuYang,female,20
  YuanJing,male,10
  CaiXuyu,female,50
  FangBo,female,50
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  CaiXuyu,female,50
  FangBo,female,50
  LiuYang,female,20
  YuanJing,male,10
  FangBo,female,50
  GuoYijun,male,50
  CaiXuyu,female,50
  FangBo,female,60
  ```

数据规划

首先需要把原日志文件放置在HDFS系统里。

1. 本地新建两个文本文件input_data1.txt和input_data2.txt，将log1.txt中的内容复制保存到input_data1.txt，将log2.txt中的内容复制保存到input_data2.txt。
2. 在HDFS客户端路径下建立一个文件夹，“/tmp/input”，并上传input_data1.txt，input_data2.txt到此目录，命令如下：
   a. 在Linux系统HDFS客户端使用命令***hadoop fs -mkdir /tmp/input***（hdfs dfs命令有同样的作用），创建对应目录。
   b. 进入到HDFS客户端下的“/tmp/input”目录，在Linux系统HDFS客户端使用命令***hadoop fs -put****input_data1.txt /tmp/input*和***hadoop fs -put****input_data2.txt /tmp/input*，上传数据文件。

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

- 读取原文件数据。
- 筛选女性网民上网时间数据信息。
- 汇总每个女性上网总时间。
- 筛选出停留时间大于两个小时的女性网民信息。

打包项目

1. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
2. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/female/”）下。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Scala**和**Java**样例程序
  - **bin/spark-submit --class** *com.huawei.bigdata.spark.examples.FemaleInfoCollection* **--master yarn --deploy-mode client** */opt/female/FemaleInfoCollection-1.0.jar <inputPath>*
  - 其中，<inputPath>指HDFS文件系统中input的路径。
- 运行**Python**样例程序
  - **bin/spark-submit --master yarn --deploy-mode client** */opt/female/SparkPythonExample/collectFemaleInfo.py <inputPath>*
  - 其中，<inputPath>指HDFS文件系统中input的路径

28.5.1.2 Spark Core 样例程序（Java）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection类：

```
//创建一个配置类SparkConf，然后创建一个SparkContext
SparkSession spark = SparkSession
  .builder()
  .appName("CollectFemaleInfo")
  .config("spark.some.config.option", "some-value")
  .getOrCreate();

//读取原文件数据,每一行记录转成RDD里面的一个元素
JavaRDD<String> data = spark.read()
  .textFile(args[0])
  .javaRDD();

//将每条记录的每列切割出来，生成一个Tuple
JavaRDD<Tuple3<String,String,Integer>> person = data.map(new
Function<String,Tuple3<String,String,Integer>>()
{
    private static final long serialVersionUID = -2381522520231963249L;
    public Tuple3<String, String, Integer> call(String s) throws Exception
    {
        //按逗号分割一行数据
        String[] tokens = s.split(",");

        //将分割后的三个元素组成一个三元Tuple
```



```
        Tuple3<String, String, Integer> person = new Tuple3<String, String, Integer>(tokens[0], tokens[1],
Integer.parseInt(tokens[2]));
        return person;
    }
});

//使用filter函数筛选出女性网民上网时间数据信息
JavaRDD<Tuple3<String,String,Integer>> female = person.filter(new
Function<Tuple3<String,String,Integer>, Boolean>()
{
    private static final long serialVersionUID = -4210609503909770492L;

    public Boolean call(Tuple3<String, String, Integer> person) throws Exception
    {
        //根据第二列性别，筛选出是female的记录
        Boolean isFemale = person._2().equals("female");
        return isFemale;
    }
});

//汇总每个女性上网总时间
JavaPairRDD<String, Integer> females = female.mapToPair(new PairFunction<Tuple3<String, String,
Integer>, String, Integer>()
{
    private static final long serialVersionUID = 8313245377656164868L;

    public Tuple2<String, Integer> call(Tuple3<String, String, Integer> female) throws Exception
    {
        //取出姓名和停留时间两列，用于后面按名字求逗留时间的总和
        Tuple2<String, Integer> femaleAndTime = new  Tuple2<String, Integer>(female._1(), female._3());
        return femaleAndTime;
    }
});
 JavaPairRDD<String, Integer> femaleTime = females.reduceByKey(new Function2<Integer, Integer,
Integer>()
{
    private static final long serialVersionUID = -3271456048413349559L;

    public Integer call(Integer integer, Integer integer2) throws Exception
    {
        //将同一个女性的两次停留时间相加，求和
        return (integer + integer2);
    }
});

//筛选出停留时间大于两个小时的女性网民信息
JavaPairRDD<String, Integer> rightFemales = females.filter(new Function<Tuple2<String, Integer>,
Boolean>()
{
    private static final long serialVersionUID = -3178168214712105171L;

    public Boolean call(Tuple2<String, Integer> s) throws Exception
    {
        //取出女性用户的总停留时间，并判断是否大于2小时
        if(s._2() > (2 * 60))
        {
            return true;
        }
        return false;
    }
});

//对符合的female信息进行打印显示
for(Tuple2<String, Integer> d: rightFemales.collect())
{
    System.out.println(d._1() + "," + d._2());
}
```

28.5.1.3 Spark Core 样例程序（Scala）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection：

样例：类CollectMapper

```
val spark = SparkSession
  .builder()
  .appName("CollectFemaleInfo")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

//读取数据。其是传入参数args(0)指定数据路径
val text = spark.sparkContext.textFile(args(0))
//筛选女性网民上网时间数据信息
val data = text.filter(_.contains("female"))
//汇总每个女性上网时间
val femaleData:RDD[(String,Int)] = data.map{line =>
    val t= line.split(',')
    (t(0),t(2).toInt)
}.reduceByKey(_ + _)
//筛选出时间大于两个小时的女性网民信息，并输出
val result = femaleData.filter(line => line._2 > 120)
result.collect().map(x => x._1 + ',' + x._2).foreach(println)
spark.stop()
```

28.5.1.4 Spark Core 样例程序（Python）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见collectFemaleInfo.py：

```
def contains(str, substr):
  if substr in str:
    return True
  return False

if __name__ == "__main__":
  if len(sys.argv) < 2:
    print "Usage: CollectFemaleInfo <file>"
    exit(-1)

  spark = SparkSession \
    .builder \
    .appName("CollectFemaleInfo") \
    .getOrCreate()

  """
  以下程序主要实现以下几步功能：
  1.读取数据。其是传入参数argv[1]指定数据路径 - text
  2.筛选女性网民上网时间数据信息 - filter
  3.汇总每个女性上网时间 - map/map/reduceByKey
```



```
4.筛选出时间大于两个小时的女性网民信息 - filter
"""
inputPath = sys.argv[1]
result = spark.read.text(inputPath).rdd.map(lambda r: r[0])\
  .filter(lambda line: contains(line, "female")) \
  .map(lambda line: line.split(',')) \
  .map(lambda dataArr: (dataArr[0], int(dataArr[2]))) \
  .reduceByKey(lambda v1, v2: v1 + v2) \
  .filter(lambda tupleVal: tupleVal[1] > 120) \
  .collect()
for (k, v) in result:
  print k + "," + str(v)

# 停止SparkContext
spark.stop()
```

28.5.2 Spark SQL 样例程序

28.5.2.1 Spark SQL 样例程序开发思路

场景说明

假定用户有某个周末网民网购停留时间的日志文本，基于某些业务要求，要求开发Spark应用程序实现如下功能：

- 统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。
- 周末两天的日志文件第一列为姓名，第二列为性别，第三列为本次停留时间，单位为分钟，分隔符为“,”。

  log1.txt：周六网民停留日志

  ```
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  FangBo,female,50
  LiuYang,female,20
  YuanJing,male,10
  GuoYijun,male,50
  CaiXuyu,female,50
  FangBo,female,60
  ```

  log2.txt：周日网民停留日志

  ```
  LiuYang,female,20
  YuanJing,male,10
  CaiXuyu,female,50
  FangBo,female,50
  GuoYijun,male,5
  CaiXuyu,female,50
  Liyuan,male,20
  CaiXuyu,female,50
  FangBo,female,50
  LiuYang,female,20
  YuanJing,male,10
  FangBo,female,50
  GuoYijun,male,50
  CaiXuyu,female,50
  FangBo,female,60
  ```

数据规划

首先需要把原日志文件放置在HDFS系统里。

1. 本地新建两个文本文件input_data1.txt和input_data2.txt，将log1.txt中的内容复制保存到input_data1.txt，将log2.txt中的内容复制保存到input_data2.txt。
2. 在HDFS客户端路径下建立一个文件夹，“/tmp/input”，并上传input_data1.txt，input_data2.txt到此目录，命令如下：
   a. 在Linux系统HDFS客户端使用命令***hadoop fs -mkdir /tmp/input***（hdfs dfs命令有同样的作用），创建对应目录。
   b. 进入到HDFS客户端下的“/tmp/input”目录，在Linux系统HDFS客户端使用命令***hadoop fs -put*** *input_data1.txt /tmp/input*和***hadoop fs -put*** *input_data2.txt /tmp/input*，上传数据文件。

开发思路

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

主要分为四个部分：

- 创建表，将日志文件数据导入到表中。
- 筛选女性网民，提取上网时间数据信息。
- 汇总每个女性上网总时间。
- 筛选出停留时间大于两个小时的女性网民信息。

打包项目

1. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
2. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“ /opt/female/” ）下。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Scala**和**Java**样例程序
  - **bin/spark-submit** --class *com.huawei.bigdata.spark.examples.FemaleInfoCollection* **--master yarn --deploy-mode client** */opt/female/SparkSqlScalaExample-1.0.jar <inputPath>*
  - 其中，<inputPath>指HDFS文件系统中input的路径。
- 运行**Python**样例程序
  - **bin/spark-submit --master yarn --deploy-mode client** */opt/female/SparkSQLPythonExample/SparkSQLPythonExample.py <inputPath>*
  - 其中，<inputPath>指HDFS文件系统中input的路径

28.5.2.2 Spark SQL 样例程序（Java）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection：

```
public static void main(String[] args) throws Exception {
SparkSession spark = SparkSession
    .builder()
    .appName("CollectFemaleInfo")
    .config("spark.some.config.option", "some-value")
    .getOrCreate();

     // 通过隐式转换，将RDD转换成DataFrame
     JavaRDD<FemaleInfo> femaleInfoJavaRDD = spark.read().textFile(args[0]). javaRDD().map(
            new Function<String, FemaleInfo>() {
                @Override
                public FemaleInfo call(String line) throws Exception {
                    String[] parts = line.split(",");
                    FemaleInfo femaleInfo = new FemaleInfo();
                    femaleInfo.setName(parts[0]);
                    femaleInfo.setGender(parts[1]);
                    femaleInfo.setStayTime(Integer.parseInt(parts[2].trim()));
                    return femaleInfo;
                }
            });

     // 注册表。
     Dataset<ROW> schemaFemaleInfo = spark.createDataFrame(femaleInfoJavaRDD,FemaleInfo.class);
     schemaFemaleInfo.registerTempTable("FemaleInfoTable");

     // 执行SQL查询
     Dataset<ROW> femaleTimeInfo = spark.sql("select * from " +
            "(select name,sum(stayTime) as totalStayTime from FemaleInfoTable " +
            "where gender = 'female' group by name )" +
            " tmp where totalStayTime >120");

    // 显示结果。
     List<String> result = femaleTimeInfo.javaRDD().map(new Function<Row, String>() {
        public String call(Row row) {
            return  row.getString(0) + "," + row.getLong(1);
        }
     }).collect();
     System.out.println(result);
     spark.stop();
  }
```

上面是简单示例，其它sparkSQL特性请参见如下链接：**http://spark.apache.org/docs/3.1.1/sql-programming-guide.html#running-sql-queries-programmatically**

28.5.2.3 Spark SQL 样例程序（Scala）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.FemaleInfoCollection：

```
object FemaleInfoCollection
 {
  //表结构，后面用来将文本数据映射为df
```

```
 case class FemaleInfo(name: String, gender: String, stayTime: Int)
 def main(args: Array[String]) {
   //配置Spark应用名称
   val spark = SparkSession
     .builder()
     .appName("FemaleInfo")
     .config("spark.some.config.option", "some-value")
     .getOrCreate()
   import spark.implicits._
   //通过隐式转换，将RDD转换成DataFrame，然后注册表
   spark.sparkContext.textFile(args(0)).map(_.split(","))
     .map(p => FemaleInfo(p(0), p(1), p(2).trim.toInt))
     .toDF.registerTempTable("FemaleInfoTable")
   //通过sql语句筛选女性上网时间数据, 对相同名字行进行聚合
   val femaleTimeInfo = spark.sql("select name,sum(stayTime) as stayTime from FemaleInfoTable where
gender = 'female' group by name")
   //筛选出时间大于两个小时的女性网民信息，并输出
   val c = femaleTimeInfo.filter("stayTime >= 120").collect().foreach(println)
   spark.stop()
 }
}
```

上面是简单示例，其它sparkSQL特性请参见如下链接：**http://spark.apache.org/docs/3.1.1/sql-programming-guide.html#running-sql-queries-programmatically**

28.5.2.4 Spark SQL 样例程序（Python）

功能简介

统计日志文件中本周末网购停留总时间超过2个小时的女性网民信息。

代码样例

下面代码片段仅为演示，具体代码参见SparkSQLPythonExample：

```
# -*- coding:utf-8 -*-

import sys
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext


def contains(str1, substr1):
    if substr1 in str1:
        return True
    return False


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print "Usage: SparkSQLPythonExample.py <file>"
        exit(-1)

    # 初始化SparkSession和SQLContext
    sc = SparkSession.builder.appName("CollectFemaleInfo").getOrCreate()
    sqlCtx = SQLContext(sc)

    # RDD转换为DataFrame
    inputPath = sys.argv[1]
    inputRDD = sc.read.text(inputPath).rdd.map(lambda r: r[0])\
        .map(lambda line: line.split(","))\
        .map(lambda dataArr: (dataArr[0], dataArr[1], int(dataArr[2])))\
        .collect()
    df = sqlCtx.createDataFrame(inputRDD)
```

```
# 注册表
df.registerTempTable("FemaleInfoTable")

# 执行SQL查询并显示结果
FemaleTimeInfo = sqlCtx.sql("SELECT * FROM " +
        "(SELECT _1 AS Name,SUM(_3) AS totalStayTime FROM FemaleInfoTable " +
        "WHERE _2 = 'female' GROUP BY _1 )" +
        " WHERE totalStayTime >120").show()

sc.stop()
```

28.5.3 通过 JDBC 访问 Spark SQL 样例程序

28.5.3.1 通过 JDBC 访问 Spark SQL 样例程序开发思路

场景说明

用户自定义JDBCServer的客户端，使用JDBC连接来进行表的创建、数据加载、查询和删除。

数据规划

将数据文件上传至HDFS中。

步骤1 确保以多主实例模式启动了JDBCServer服务，并至少有一个实例可连接客户端。在Linux系统HDFS客户端新建一个文本文件“data”，内容如下：

```
Miranda,32
Karlie,23
Candice,27
```

步骤2 在HDFS路径下建立一个目录，例如创建“/home”，并上传“data”文件到此目录，命令如下：

1. 登录HDFS客户端节点，执行如下命令：

   cd *{客户端安装目录}*

   source bigdata_env

2. 执行如下命令创建目录“/home”：

   hdfs dfs -mkdir /home

3. 执行如下命令上传数据文件：

   hdfs dfs -put data /home

步骤3 确保其对启动JDBCServer的用户有读写权限。

步骤4 确保客户端classpath下有“hive-site.xml”文件，且根据实际集群情况配置所需要的参数。JDBCServer相关参数详情，请参见**Spark JDBCServer接口介绍**。

----结束

开发思路

1. 在default数据库下创建child表。
2. 把“/home/data”的数据加载进child表中。
3. 查询child表中的数据。
4. 删除child表。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/female/”）下。

运行任务

进入Spark客户端目录，使用java -cp命令运行代码（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Java**样例代码：

  java -cp *$SPARK_HOME/jars/*:$SPARK_HOME/jars/hive/*:$SPARK_HOME/conf:/opt/female/SparkThriftServerJavaExample-1.0.jar com.huawei.bigdata.spark.examples.ThriftServerQueriesTest $SPARK_HOME/conf/hive-site.xml $SPARK_HOME/conf/spark-defaults.conf*

- 运行**Scala**样例代码：

  java -cp *$SPARK_HOME/jars/*:$SPARK_HOME/jars/hive/*:$SPARK_HOME/conf:/opt/female/SparkThriftServerExample-1.0.jar com.huawei.bigdata.spark.examples.ThriftServerQueriesTest $SPARK_HOME/conf/hive-site.xml $SPARK_HOME/conf/spark-defaults.conf*

  说明

  集群开启ZooKeeper的SSL特性后（查看ZooKeeper服务的ssl.enabled参数），请在执行命令中添加-Dzookeeper.client.secure=true -Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty两项参数：

  java -Dzookeeper.client.secure=true -Dzookeeper.clientCnxnSocket=org.apache.zookeeper.ClientCnxnSocketNetty **-cp** *$SPARK_HOME/jars/*:$SPARK_HOME/jars/hive/*:$SPARK_HOME/conf:/opt/female/SparkThriftServerJavaExample-1.0.jar com.huawei.bigdata.spark.examples.ThriftServerQueriesTest $SPARK_HOME/conf/hive-site.xml $SPARK_HOME/conf/spark-defaults.conf*

28.5.3.2 通过 JDBC 访问 Spark SQL 样例程序（Java）

功能简介

使用自定义客户端的JDBC接口提交数据分析任务，并返回结果。

样例代码

步骤1 定义SQL语句。SQL语句必须为单条语句，注意其中不能包含“;”。示例：

```
ArrayList<String> sqlList = new ArrayList<String>();
sqlList.add("CREATE TABLE CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE");
sqlList.add("LOAD DATA INPATH 'hdfs://hacluster/home/data' INTO TABLE CHILD");
sqlList.add("SELECT * FROM child");
sqlList.add("DROP TABLE child");
executeSql(url, sqlList);
```

步骤2 拼接JDBC URL。

```
Configuration config = new Configuration();
config.addResource(new Path(args[0]));
String zkUrl = config.get("spark.deploy.zookeeper.url");
```

```
String zkNamespace = null;
zkNamespace = fileInfo.getProperty("spark.thriftserver.zookeeper.namespace");
if (zkNamespace != null) {
  //从配置项中删除冗余字符
  zkNamespace = zkNamespace.substring(1);
}

StringBuilder sb = new StringBuilder("jdbc:hive2://"
      + zkUrl
      + "/;serviceDiscoveryMode=zooKeeper;"
      + "zooKeeperNamespace="
      + zkNamespace + ";");
String url = sb.toString();
```

步骤3 加载Hive JDBC驱动。

```
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
```

步骤4 获取JDBC连接，执行HQL，输出查询的列名和结果到控制台，关闭JDBC连接。

连接字符串中的“zk.quorum”也可以使用配置文件中的配置项“spark.deploy.zookeeper.url”来代替。

在网络拥塞的情况下，您还可以设置客户端与JDBCServer连接的超时时间，可以避免客户端由于无限等待服务端的返回而挂起。使用方式如下：

在执行“DriverManager.getConnection”方法获取JDBC连接前，添加“DriverManager.setLoginTimeout(n)”方法来设置超时时长，其中n表示等待服务返回的超时时长，单位为秒，类型为Int，默认为“0”（表示永不超时）。

```
static void executeSql(String url, ArrayList<String> sqls) throws ClassNotFoundException, SQLException {
    try {
        Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
    } catch (Exception e) {
        e.printStackTrace();
    }
    Connection connection = null;
    PreparedStatement statement = null;

    try {
        connection = DriverManager.getConnection(url);
        for (int i = 0 ; i < sqls.size(); i++) {
            String sql = sqls.get(i);
            System.out.println("---- Begin executing sql: " + sql + " ----");
            statement = connection.prepareStatement(sql);
            ResultSet result = statement.executeQuery();
            ResultSetMetaData resultMetaData = result.getMetaData();
            Integer colNum = resultMetaData.getColumnCount();
            for (int j = 1; j <= colNum; j++) {
                System.out.println(resultMetaData.getColumnLabel(j) + "\t");
            }
            System.out.println();

            while (result.next()) {
                for (int j = 1; j <= colNum; j++){
                    System.out.println(result.getString(j) + "\t");
                }
                System.out.println();
            }
            System.out.println("---- Done executing sql: " + sql + " ----");
        }


    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (null != statement) {
            statement.close();
```



```
        }
        if (null != connection) {
            connection.close();
        }
    }
  }
```

----结束

28.5.3.3 过 JDBC 访问 Spark SQL 样例程序（Scala）

功能简介

使用自定义客户端的JDBC接口提交数据分析任务，并返回结果。

样例代码

步骤1 定义SQL语句。SQL语句必须为单条语句，注意其中不能包含“;”。示例：

```
val sqlList = new ArrayBuffer[String]
sqlList += "CREATE TABLE CHILD (NAME STRING, AGE INT) " +
"ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE"
sqlList += "LOAD DATA INPATH 'hdfs://hacluster/home/data' INTO TABLE CHILD"
sqlList += "SELECT * FROM child"
sqlList += "DROP TABLE child"
```

步骤2 拼接JDBC URL。

```
val config: Configuration = new Configuration()
config.addResource(new Path(args(0)))
val zkUrl = config.get("spark.deploy.zookeeper.url")

var zkNamespace: String = null
zkNamespace = fileInfo.getProperty("spark.thriftserver.zookeeper.namespace")
//从配置项中删除冗余字符
if (zkNamespace != null) zkNamespace = zkNamespace.substring(1)

val sb = new StringBuilder("jdbc:hive2://"
  + zkUrl
  + "/;serviceDiscoveryMode=zooKeeper;"
  + "zooKeeperNamespace="
  + zkNamespace + ";");
val url = sb.toString()
```

步骤3 加载Hive JDBC驱动。获取JDBC连接，执行HQL，输出查询的列名和结果到控制台，关闭JDBC连接。

连接字符串中的“zk.quorum”也可以使用配置文件中的配置项“spark.deploy.zookeeper.url”来代替。

在网络拥塞的情况下，您还可以设置客户端与JDBCServer连接的超时时间，可以避免客户端由于无限等待服务端的返回而挂起。使用方式如下：

在执行“DriverManager.getConnection”方法获取JDBC连接前，添加“DriverManager.setLoginTimeout(n)”方法来设置超时时长，其中n表示等待服务返回的超时时长，单位为秒，类型为Int，默认为“0”（表示永不超时）。

```
def executeSql(url: String, sqls: Array[String]): Unit = {
//加载Hive JDBC驱动。
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance()

var connection: Connection = null
var statement: PreparedStatement = null
try {
  connection = DriverManager.getConnection(url)
```



```
  for (sql <- sqls) {
    println(s"---- Begin executing sql: $sql ----")
    statement = connection.prepareStatement(sql)

    val result = statement.executeQuery()

    val resultMetaData = result.getMetaData
    val colNum = resultMetaData.getColumnCount
    for (i <- 1 to colNum) {
      print(resultMetaData.getColumnLabel(i) + "\t")
    }
    println()

    while (result.next()) {
      for (i <- 1 to colNum) {
        print(result.getString(i) + "\t")
      }
      println()
    }
    println(s"---- Done executing sql: $sql ----")
  }
} finally {
  if (null != statement) {
    statement.close()
  }

  if (null != connection) {
    connection.close()
  }
}
}
```

----结束

28.5.4 Spark 读取 HBase 表样例程序

28.5.4.1 操作 Avro 格式数据

场景说明

用户可以在Spark应用程序中以数据源的方式去使用HBase，本例中将数据以Avro格式存储在HBase中，并从中读取数据以及对读取的数据进行过滤等操作。

数据规划

在客户端执行**hbase shell**，进入HBase命令行，使用下面的命令创建样例代码中要使用的HBase表：

create *'ExampleAvrotable','rowkey','cf1'*（如果表已经存在，则每次执行**提交命令**前需清空表里的数据：truncate *'ExampleAvrotable'*）

create *'ExampleAvrotableInsert','rowkey','cf1'*（如果表已经存在，则每次执行**提交命令**前需清空表里的数据：truncate *'ExampleAvrotableInsert'*）

开发思路

1. 创建RDD。
2. 以数据源的方式操作HBase，将上面生成的RDD写入HBase表中。
3. 读取HBase表中的数据，并且对其进行简单的操作。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.AvroSource SparkOnHbaseJavaExample-1.0.jar

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --conf spark.yarn.user.classpath.first=true --jars SparkOnHbaseJavaExample-1.0.jar,/opt/female/protobuf-java-2.5.0.jar AvroSource.py

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.AvroSource SparkOnHbaseJavaExample-1.0.jar

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --conf spark.yarn.user.classpath.first=true --jars SparkOnHbaseJavaExample-1.0.jar,/opt/female/protobuf-java-2.5.0.jar AvroSource.py

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中的AvroSource文件：

```
public static void main(JavaSparkContext jsc) throws IOException {
    SQLContext sqlContext = new SQLContext(jsc);
    Configuration hbaseconf = new HBaseConfiguration().create();
    JavaHBaseContext hBaseContext = new JavaHBaseContext(jsc, hbaseconf);
    List list = new ArrayList<AvroHBaseRecord>();
    for(int i=0; i<=255 ; ++i){
```

```
            list.add(AvroHBaseRecord.apply(i));
        }
        try{
            Map<String, String> map = new HashMap<String, String>();
            map.put(HBaseTableCatalog.tableCatalog(), catalog);
            map.put(HBaseTableCatalog.newTable(), "5");
            sqlContext.createDataFrame(list,
AvroHBaseRecord.class).write().options(map).format("org.apache.hadoop.hbase.spark").save();
            Dataset<Row> ds = withCatalog(sqlContext,catalog);
            ds.show();
            ds.printSchema();
            ds.registerTempTable("ExampleAvrotable");
            Dataset<Row> c= sqlContext.sql("select count(1) from ExampleAvrotable");
            c.show();
            Dataset<Row> filtered = ds.select("col0", "col1.favorite_array").where("col0 = 'name1'");
            filtered.show();
            java.util.List<Row> collected = filtered.collectAsList();
            if (collected.get(0).get(1).toString().equals("number1")) {
                throw new UserCustomizedSampleException("value invalid", new Throwable());
            }
            if (collected.get(0).get(1).toString().equals("number2")) {
                throw new UserCustomizedSampleException("value invalid", new Throwable());
            }
            Map avroCatalogInsertMap = new HashMap<String,String>();
            avroCatalogInsertMap.put("avroSchema" , AvroHBaseRecord.schemaString);
            avroCatalogInsertMap.put(HBaseTableCatalog.tableCatalog(), avroCatalogInsert);
            ds.write().options(avroCatalogInsertMap).format("org.apache.hadoop.hbase.spark").save();
            Dataset<Row> newDS = withCatalog(sqlContext,avroCatalogInsert);
            newDS.show();
            newDS.printSchema();
            if (newDS.count() != 256) {
                throw new UserCustomizedSampleException("value invalid", new Throwable());
            }
            ds.filter("col1.name = 'name5' || col1.name <= 'name5'").select("col0","col1.favorite_color",
"col1.favorite_number").show();
        } finally{
            jsc.stop();
        }
    }
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中的AvroSource文件：

```
def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("AvroSourceExample")
    val sc = new SparkContext(sparkConf)
    val sqlContext = new SQLContext(sc)
    val hbaseConf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, hbaseConf)
    import sqlContext.implicits._
    def withCatalog(cat: String): DataFrame = {
      sqlContext
        .read
        .options(Map("avroSchema" -> AvroHBaseRecord.schemaString, HBaseTableCatalog.tableCatalog ->
avroCatalog))
        .format("org.apache.hadoop.hbase.spark")
        .load()
    }
    val data = (0 to 255).map { i =>
      AvroHBaseRecord(i)
    }
    try {
      sc.parallelize(data).toDF.write.options(
        Map(HBaseTableCatalog.tableCatalog -> catalog, HBaseTableCatalog.newTable -> "5"))
        .format("org.apache.hadoop.hbase.spark")
        .save()
```

```
    val df = withCatalog(catalog)
    df.show()
    df.printSchema()
    df.registerTempTable("ExampleAvrotable")
    val c = sqlContext.sql("select count(1) from ExampleAvrotable")
    c.show()

    val filtered = df.select($"col0", $"col1.favorite_array").where($"col0" === "name001")
    filtered.show()
    val collected = filtered.collect()
    if (collected(0).getSeq[String](1)(0) != "number1") {
      throw new UserCustomizedSampleException("value invalid")
    }
    if (collected(0).getSeq[String](1)(1) != "number2") {
      throw new UserCustomizedSampleException("value invalid")
    }

    df.write.options(
      Map("avroSchema" -> AvroHBaseRecord.schemaString, HBaseTableCatalog.tableCatalog ->
avroCatalogInsert,
        HBaseTableCatalog.newTable -> "5"))
      .format("org.apache.hadoop.hbase.spark")
      .save()
    val newDF = withCatalog(avroCatalogInsert)
    newDF.show()
    newDF.printSchema()
    if (newDF.count() != 256) {
      throw new UserCustomizedSampleException("value invalid")
    }
    df.filter($"col1.name" === "name005" || $"col1.name" <= "name005")
      .select("col0", "col1.favorite_color", "col1.favorite_number")
      .show()
  } finally {
    sc.stop()
  }
 }
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中的AvroSource文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("AvroSourceExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.datasources.AvroSource')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.AvroSource().execute(spark._jsc)
# 停止SparkSession
spark.stop()
```

28.5.4.2 操作 HBase 数据源

场景说明

用户可以在Spark应用程序中以数据源的方式去使用HBase，将dataFrame写入HBase中，并从HBase读取数据以及对读取的数据进行过滤等操作。

数据规划

在客户端执行**hbase shell**，进入HBase命令行，使用下面的命令创建样例代码中要使用的HBase表：

create *'HBaseSourceExampleTable','rowkey','cf1','cf2','cf3','cf4','cf5','cf6','cf7', 'cf8'*

开发思路

1. 创建RDD.
2. 以数据源的方式操作HBase，将上面生成的RDD写入HBase表中.
3. 读取HBase表中的数据，并且对其进行简单的操作。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.HBaseSource SparkOnHbaseJavaExample-1.0.jar

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --conf spark.yarn.user.classpath.first=true --jars SparkOnHbaseJavaExample-1.0.jar,/opt/female/protobuf-java-2.5.0.jar HBaseSource.py

- yarn-cluster模式：

java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

bin/spark-submit --master yarn --deploy-mode cluster --jars /opt/female/protobuf-java-2.5.0.jar --conf spark.yarn.user.classpath.first=true --class com.huawei.bigdata.spark.examples.datasources.HBaseSource SparkOnHbaseJavaExample-1.0.jar

python版本（文件名等请与实际保持一致，此处仅为示例）

bin/spark-submit --master yarn --deploy-mode cluster --conf spark.yarn.user.classpath.first=true --jars SparkOnHbaseJavaExample-1.0.jar,/opt/female/protobuf-java-2.5.0.jar HBaseSource.py

Java 样例代码

面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中的HBaseSource文件：

```
  public static void main(String args[]) throws IOException{
      SparkConf sparkConf = new SparkConf().setAppName("HBaseSourceExample");
      JavaSparkContext jsc = new JavaSparkContext(sparkConf);
      SQLContext sqlContext = new SQLContext(jsc);


      Configuration conf = HBaseConfiguration.create();
      JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc,conf);
      try{
          List<HBaseRecord> list = new ArrayList<HBaseRecord>();
          for(int i=0 ; i<256; i++){
              list.add(new HBaseRecord(i));
          }
          Map map = new HashMap<String, String>();
          map.put(HBaseTableCatalog.tableCatalog(), cat);
          map.put(HBaseTableCatalog.newTable(), "5");
          System.out.println("Before insert data into hbase table");
          sqlContext.createDataFrame(list,
HBaseRecord.class).write().options(map).format("org.apache.hadoop.hbase.spark").save();
          Dataset<Row> ds = withCatalog(sqlContext, cat);
          System.out.println("After insert data into hbase table");
          ds.printSchema();
          ds.show();
          ds.filter("key <= 'row5'").select("key","col1").show();
          ds.registerTempTable("table1");
          Dataset<Row> tempDS = sqlContext.sql("select count(col1) from table1 where key < 'row5'");
          tempDS.show();
      } finally {
          jsc.stop();
      }
  }
```

Scala 样例代码

面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中的HBaseSource文件：

```
def main(args: Array[String]) {
   val sparkConf = new SparkConf().setAppName("HBaseSourceExample")
   val sc = new SparkContext(sparkConf)
   val sqlContext = new SQLContext(sc)
   val conf = HBaseConfiguration.create()
   val hbaseContext = new HBaseContext(sc,conf)
   import sqlContext.implicits._
   def withCatalog(cat: String): DataFrame = {
     sqlContext
       .read
```



```
      .options(Map(HBaseTableCatalog.tableCatalog->cat))
      .format("org.apache.hadoop.hbase.spark")
      .load()
  }
  val data = (0 to 255).map { i =>
    HBaseRecord(i)
  }
  try{
    sc.parallelize(data).toDF.write.options(
      Map(HBaseTableCatalog.tableCatalog -> cat, HBaseTableCatalog.newTable -> "5"))
      .format("org.apache.hadoop.hbase.spark")
      .save()
    val df = withCatalog(cat)
    df.show()
    df.filter($"col0" <= "row005")
      .select($"col0", $"col1").show
    df.registerTempTable("table1")
    val c = sqlContext.sql("select count(col1) from table1 where col0 < 'row050'")
    c.show()
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中的HBaseSource文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("HBaseSourceExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.datasources.HBaseSource')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.HBaseSource().execute(spark._jsc)
# 停止SparkSession
spark.stop()
```

28.5.4.3 BulkPut 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将构建的RDD写入HBase中。

数据规划

在客户端执行**hbase shell**，进入HBase命令行，使用下面的命令创建样例代码中要使用的Hase表：

create *'b*ulktable','cf1'

开发思路

1. 创建RDD。
2. 以HBaseContext的方式操作HBase，将上面生成的RDD写入HBase表中。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在$SPARK_HOME目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkPutExample SparkOnHbaseJavaExample-1.0.jar bulktable cf1

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkPutExample.py bulktable cf1

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkPutExample SparkOnHbaseJavaExample-1.0.jar bulktable cf1

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkPutExample.py bulktable cf1

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中的JavaHBaseBulkPutExample文件：

```
public static void main(String[] args) throws Exception{
  if (args.length < 2) {
    System.out.println("JavaHBaseBulkPutExample  " +
```

```
            "{tableName} {columnFamily}");
    return;
  }
  String tableName = args[0];
  String columnFamily = args[1];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkPutExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<String> list = new ArrayList<String>(5);
    list.add("1," + columnFamily + ",1,1");
    list.add("2," + columnFamily + ",1,2");
    list.add("3," + columnFamily + ",1,3");
    list.add("4," + columnFamily + ",1,4");
    list.add("5," + columnFamily + ",1,5");
    list.add("6," + columnFamily + ",1,6");
    list.add("7," + columnFamily + ",1,7");
    list.add("8," + columnFamily + ",1,8");
    list.add("9," + columnFamily + ",1,9");
    list.add("10," + columnFamily + ",1,10");
    JavaRDD<String> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.bulkPut(rdd,
            TableName.valueOf(tableName),
            new PutFunction());
    System.out.println("Bulk put into Hbase successfully!");
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkPutExample文件：

```
 def main(args: Array[String]) {
   if (args.length < 2) {
     System.out.println("HBaseBulkPutTimestampExample {tableName} {columnFamily} are missing an
argument")
     return
   }
   val tableName = args(0)
   val columnFamily = args(1)
   val sparkConf = new SparkConf().setAppName("HBaseBulkPutTimestampExample " +
     tableName + " " + columnFamily)
   val sc = new SparkContext(sparkConf)
   try {
     val rdd = sc.parallelize(Array(
       (Bytes.toBytes("1"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("1")))),
       (Bytes.toBytes("2"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("2")))),
       (Bytes.toBytes("3"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("3")))),
       (Bytes.toBytes("4"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("4")))),
       (Bytes.toBytes("5"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("5")))),
       (Bytes.toBytes("6"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("6")))),
       (Bytes.toBytes("7"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("7")))),
       (Bytes.toBytes("8"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("8")))),
       (Bytes.toBytes("9"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("9")))),
       (Bytes.toBytes("10"),
         Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("10"))))))
```

```
    val conf = HBaseConfiguration.create()
    val timeStamp = System.currentTimeMillis()
    val hbaseContext = new HBaseContext(sc, conf)
    hbaseContext.bulkPut[(Array[Byte], Array[(Array[Byte], Array[Byte], Array[Byte])])](rdd,
      TableName.valueOf(tableName),
      (putRecord) => {
        val put = new Put(putRecord._1)
        putRecord._2.foreach((putValue) => put.addColumn(putValue._1, putValue._2,
          timeStamp, putValue._3))
        put
      })
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkPutExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkPutExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkPutExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseBulkPutExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

28.5.4.4 BulkGet 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将要获取的数据的rowKey构造成rdd，然后通过HBaseContext的bulkGet接口获取对HBase表上这些rowKey对应的数据。

数据规划

基于**BulkPut接口使用**章节创建的HBase表及其中的数据进行操作。

开发思路

1. 创建包含了要获取的rowkey信息的RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的bulkGet接口获取HBase表上这些rowKey对应的数据。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkGetExample SparkOnHbaseJavaExample-1.0.jar bulktable

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkGetExample.py bulktable
- yarn-cluster模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkGetExample SparkOnHbaseJavaExample-1.0.jar bulktable

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkGetExample.py bulktable

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中HBaseBulkGetExample文件：

```
public static void main(String[] args) throws IOException{
  if (args.length < 1) {
    System.out.println("JavaHBaseBulkGetExample  {tableName}");
    return;
  }
  String tableName = args[0];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkGetExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<byte[]> list = new ArrayList<byte[]>(5);
    list.add(Bytes.toBytes("1"));
    list.add(Bytes.toBytes("2"));
```

```
    list.add(Bytes.toBytes("3"));
    list.add(Bytes.toBytes("4"));
    list.add(Bytes.toBytes("5"));
    JavaRDD<byte[]> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    List resultList = hbaseContext.bulkGet(TableName.valueOf(tableName), 2, rdd, new GetFunction(),
            new ResultFunction()).collect();
    for(int i =0 ;i<resultList.size();i++){
      System.out.println(resultList.get(i));
    }
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkGetExample文件：

```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseBulkGetExample {tableName} missing an argument")
    return
  }
  val tableName = args(0)
  val sparkConf = new SparkConf().setAppName("HBaseBulkGetExample " + tableName)
  val sc = new SparkContext(sparkConf)
  try {
    //[(Array[Byte])]
    val rdd = sc.parallelize(Array(
      Bytes.toBytes("1"),
      Bytes.toBytes("2"),
      Bytes.toBytes("3"),
      Bytes.toBytes("4"),
      Bytes.toBytes("5"),
      Bytes.toBytes("6"),
      Bytes.toBytes("7")))
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
    val getRdd = hbaseContext.bulkGet[Array[Byte], String](
      TableName.valueOf(tableName),
      2,
      rdd,
      record => {
        System.out.println("making Get")
        new Get(record)
      },
      (result: Result) => {
        val it = result.listCells().iterator()
        val b = new StringBuilder
        b.append(Bytes.toString(result.getRow) + ":")
        while (it.hasNext) {
          val cell = it.next()
          val q = Bytes.toString(CellUtil.cloneQualifier(cell))
          if (q.equals("counter")) {
            b.append("(" + q + "," + Bytes.toLong(CellUtil.cloneValue(cell)) + ")")
          } else {
            b.append("(" + q + "," + Bytes.toString(CellUtil.cloneValue(cell)) + ")")
          }
        }
        b.toString()
      })
    getRdd.collect().foreach(v => println(v))
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkGetExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkGetExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkGetExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseBulkGetExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

28.5.4.5 BulkDelete 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将要删除的数据的rowKey构造成rdd，然后通过HBaseContext的bulkDelete接口对HBase表上这些rowKey对应的数据进行删除。

数据规划

基于**BulkPut接口使用**章节创建的HBase表及其中的数据进行操作。

开发思路

1. 创建包含了要删除的rowkey信息的RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的bulkDelete接口对HBase表上这些rowKey对应的数据进行删除。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkDeleteExample SparkOnHbaseJavaExample-1.0.jar bulktable

  python版本（文件名等于实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkDeleteExample.py bulktable

- yarn-cluster模式：

  java/scala 版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkDeleteExample SparkOnHbaseJavaExample-1.0.jar bulktable

  python版本（文件名等于实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkDeleteExample.py bulktable

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中HBaseBulkDeleteExample文件：

```
public static void main(String[] args) throws IOException {
  if (args.length < 1) {
    System.out.println("JavaHBaseBulkDeleteExample  {tableName}");
    return;
  }
  String tableName = args[0];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkDeleteExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<byte[]> list = new ArrayList<byte[]>(5);
    list.add(Bytes.toBytes("1"));
    list.add(Bytes.toBytes("2"));
    list.add(Bytes.toBytes("3"));
    list.add(Bytes.toBytes("4"));
    list.add(Bytes.toBytes("5"));
    JavaRDD<byte[]> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.bulkDelete(rdd,
          TableName.valueOf(tableName), new DeleteFunction(), 4);
    System.out.println("Bulk Delete successfully!");
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkDeleteExample文件：


```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseBulkDeleteExample {tableName} missing an argument")
    return
  }
  val tableName = args(0)
  val sparkConf = new SparkConf().setAppName("HBaseBulkDeleteExample " + tableName)
  val sc = new SparkContext(sparkConf)
  try {
    //[Array[Byte]]
    val rdd = sc.parallelize(Array(
      Bytes.toBytes("1"),
      Bytes.toBytes("2"),
      Bytes.toBytes("3"),
      Bytes.toBytes("4"),
      Bytes.toBytes("5")
    ))
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
    hbaseContext.bulkDelete[Array[Byte]](rdd,
      TableName.valueOf(tableName),
      putRecord => new Delete(putRecord),
      4)
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkDeleteExample文件：

```
  def main(args: Array[String]) {
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkDeleteExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkDeleteExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseBulkDeleteExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

28.5.4.6 BulkLoad 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去使用HBase，将要插入的数据的rowKey构造成rdd，然后通过HBaseContext的bulkLoad接口将rdd写入HFile中。将生成的HFile导入HBase表的操作采用如下格式的命令，不属于本接口范围，不在此进行详细说明：

hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles {hfilePath} {tableName}

数据规划

1. 在客户端执行：**hbase shell**命令进入HBase命令行。
2. 使用下面的命令创建HBase表：

   create *'bulkload-table-test','f1','f2'*

开发思路

1. 将要导入的数据构造成RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的bulkLoad接口将rdd写入HFile中。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkLoadExample SparkOnHbaseJavaExample-1.0.jar /tmp/hfile bulkload-table-test

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkLoadExample.py /tmp/hfile bulkload-table-test

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseBulkLoadExample SparkOnHbaseJavaExample-1.0.jar /tmp/hfile bulkload-table-test

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseBulkLoadExample.py /tmp/hfile bulkload-table-test

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseBulkLoadExample文件：

```
public static void main(String[] args) throws IOException{
  if (args.length < 2) {
    System.out.println("JavaHBaseBulkLoadExample {outputPath} {tableName}");
    return;
  }
  String outputPath = args[0];
  String tableName = args[1];
  String columnFamily1 = "f1";
  String columnFamily2 = "f2";
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkLoadExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    List<String> list= new ArrayList<String>();
    // row1
    list.add("1," + columnFamily1 + ",b,1");
    // row3
    list.add("3," + columnFamily1 + ",a,2");
    list.add("3," + columnFamily1 + ",b,1");
    list.add("3," + columnFamily2 + ",a,1");
    /* row2 */
    list.add("2," + columnFamily2 + ",a,3");
    list.add("2," + columnFamily2 + ",b,3");
    JavaRDD<String> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.bulkLoad(rdd, TableName.valueOf(tableName),new BulkLoadFunction(), outputPath,
        new HashMap<byte[], FamilyHFileWriteOptions>(), false, HConstants.DEFAULT_MAX_FILE_SIZE);
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseBulkLoadExample文件：

```
def main(args: Array[String]) {
    if(args.length < 2) {
      println("HBaseBulkLoadExample {outputPath} {tableName}
      return
    }
    LoginUtil.loginWithUserKeytab()
    val Array(outputPath, tableName) = args
    val columnFamily1 = "f1"
    val columnFamily2 = "f2"
    val sparkConf = new SparkConf().setAppName("JavaHBaseBulkLoadExample " + tableName)
    val sc = new SparkContext(sparkConf)
    try {
      val arr = Array("1," + columnFamily1 + ",b,1",
                      "2," + columnFamily1 + ",a,2",
                      "3," + columnFamily1 + ",b,1",
                      "3," + columnFamily2 + ",a,1",
                      "4," + columnFamily2 + ",a,3",
                      "5," + columnFamily2 + ",b,3")

        val rdd = sc.parallelize(arr)
        val config = HBaseConfiguration.create
        val hbaseContext = new HBaseContext(sc, config)
        hbaseContext.bulkLoad[String](rdd,
          TableName.valueOf(tableName),
          (putRecord) => {
            if(putRecord.length > 0) {
```

```
                val strArray = putRecord.split(",")
                val kfq = new KeyFamilyQualifier(Bytes.toBytes(strArray(0)), Bytes.toBytes(strArray(1)),
Bytes.toBytes(strArray(2)))
                val ite = (kfq, Bytes.toBytes(strArray(3)))
                val itea = List(ite).iterator
                itea
              } else {
                null
              }
            },
            outputPath)
        } finally {
          sc.stop()
        }
    }
 }
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseBulkLoadPythonExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseBulkLoadExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.HBaseBulkLoadPythonExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.HBaseBulkLoadPythonExample().hbaseBulkLoad(spark._jsc, sys.argv[1], sys.argv[2])
# 停止SparkSession
spark.stop()
```

28.5.4.7 foreachPartition 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，将要插入的数据的rowKey构造成rdd，然后通过HBaseContext的mapPartition接口将rdd并发写入HBase表中。

数据规划

1. 在客户端执行：**hbase shell**命令进入HBase命令行。
2. 使用下面的命令创建HBase表：

   create *'table2','cf1'*

开发思路

1. 将要导入的数据构造成RDD。
2. 以HBaseContext的方式操作HBase，通过HBaseContext的foreachPatition接口将数据并发写入HBase中。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseForEachPartitionExample SparkOnHbaseJavaExample-1.0.jar table2 cf1

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample-1.0.jar HBaseForEachPartitionExample.py table2 cf1

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseForEachPartitionExample SparkOnHbaseJavaExample-1.0.jar table2 cf1

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseForEachPartitionExample.py table2 cf1

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseForEachPartitionExample文件：

```
public static void main(String[] args) throws IOException {
  if (args.length < 1) {
    System.out.println("JavaHBaseForEachPartitionExample {tableName} {columnFamily}");
    return;
  }
  final String tableName = args[0];
  final String columnFamily = args[1];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkGetExample " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
```

```
    List<byte[]> list = new ArrayList<byte[]>(5);
    list.add(Bytes.toBytes("1"));
    list.add(Bytes.toBytes("2"));
    list.add(Bytes.toBytes("3"));
    list.add(Bytes.toBytes("4"));
    list.add(Bytes.toBytes("5"));
    JavaRDD<byte[]> rdd = jsc.parallelize(list);
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.foreachPartition(rdd,
            new VoidFunction<Tuple2<Iterator<byte[]>, Connection>>() {
       public void call(Tuple2<Iterator<byte[]>, Connection> t)
                throws Exception {
         Connection con = t._2();
         Iterator<byte[]> it = t._1();
         BufferedMutator buf = con.getBufferedMutator(TableName.valueOf(tableName));
         while (it.hasNext()) {
           byte[] b = it.next();
           Put put = new Put(b);
           put.add(Bytes.toBytes(columnFamily), Bytes.toBytes("cid"), b);
           buf.mutate(put);
         }
         mutator.flush();
         mutator.close();

       }
     });
  } finally {
    jsc.stop();
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseForEachPartitionExample文件：

```
def main(args: Array[String]) {
  if (args.length < 2) {
    println("HBaseForeachPartitionExample {tableName} {columnFamily} are missing an arguments")
    return
  }
  val tableName = args(0)
  val columnFamily = args(1)
  val sparkConf = new SparkConf().setAppName("HBaseForeachPartitionExample " +
    tableName + " " + columnFamily)
  val sc = new SparkContext(sparkConf)
  try {
    //[(Array[Byte], Array[(Array[Byte], Array[Byte], Array[Byte])])]
    val rdd = sc.parallelize(Array(
      (Bytes.toBytes("1"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("1")))),
      (Bytes.toBytes("2"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("2")))),
      (Bytes.toBytes("3"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("3")))),
      (Bytes.toBytes("4"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("4")))),
      (Bytes.toBytes("5"),
        Array((Bytes.toBytes(columnFamily), Bytes.toBytes("1"), Bytes.toBytes("5"))))
    ))
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
    rdd.hbaseForeachPartition(hbaseContext,
      (it, connection) => {
        val m = connection.getBufferedMutator(TableName.valueOf(tableName))
        it.foreach(r => {
          val put = new Put(r._1)
          r._2.foreach((putValue) =>
```



```
            put.addColumn(putValue._1, putValue._2, putValue._3))
          m.mutate(put)
        })
        m.flush()
        m.close()
      })
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseForEachPartitionExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseForEachPartitionExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm,
'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseForEachPartitionExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseForEachPartitionExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

28.5.4.8 分布式 Scan HBase 表

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，使用hbaseRDD方法以特定的规则扫描HBase表。

数据规划

使用**操作Avro格式数据**章节中创建的HBase数据表。

开发思路

1. 设置scan的规则，例如：setCaching。
2. 使用特定的规则扫描Hbase表。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

说明

若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端$SPARK_HOME目录下，以下命令均在$SPARK_HOME目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseDistributedScanExample SparkOnHbaseJavaExample-1.0.jar ExampleAvrotable

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample-1.0.jar HBaseDistributedScanExample.py ExampleAvrotable

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseDistributedScanExample SparkOnHbaseJavaExample-1.0.jar ExampleAvrotable

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseDistributedScanExample.py ExampleAvrotable

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseDistributedScanExample文件：

```
public static void main(String[] args) throws IOException{
  if (args.length < 1) {
    System.out.println("JavaHBaseDistributedScan {tableName}");
    return;
  }
  String tableName = args[0];
  SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseDistributedScan " + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    Scan scan = new Scan();
    scan.setCaching(100);
    JavaRDD<Tuple2<ImmutableBytesWritable, Result>> javaRdd =
          hbaseContext.hbaseRDD(TableName.valueOf(tableName), scan);
    List<String> results = javaRdd.map(new ScanConvertFunction()).collect();
    System.out.println("Result Size: " + results.size());
  } finally {
    jsc.stop();
```



```
  }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseDistributedScanExample文件：

```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseDistributedScanExample {tableName} missing an argument")
    return
  }
  val tableName = args(0)
  val sparkConf = new SparkConf().setAppName("HBaseDistributedScanExample " + tableName )
  val sc = new SparkContext(sparkConf)
  try {
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
    val scan = new Scan()
    scan.setCaching(100)
    val getRdd = hbaseContext.hbaseRDD(TableName.valueOf(tableName), scan)
    getRdd.foreach(v => println(Bytes.toString(v._1.get())))
    println("Length: " + getRdd.map(r => r._1.copyBytes()).collect().length);
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseDistributedScanExample文件：

```
# -*- coding:utf-8 -*-
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseDistributedScan")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm,
'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseDistributedScanExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseDistributedScan().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

28.5.4.9 mapPartition 接口使用

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，使用mapPartition接口并行遍历HBase表。

数据规划

使用**foreachPartition接口使用**章节创建的HBase数据表。

开发思路

1. 构造需要遍历的HBase表中rowkey的RDD。
2. 使用mapPartition接口遍历上述rowkey对应的数据信息，并进行简单的操作。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseMapPartitionExample SparkOnHbaseJavaExample-1.0.jar table2

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode client --jars SparkOnHbaseJavaExample-1.0.jar HBaseMapPartitionExample.py table2

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --class com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseMapPartitionExample SparkOnHbaseJavaExample-1.0.jar table2

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseMapPartitionExample.py table2

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseMapPartitionExample文件：

```
public static void main(String args[]) throws IOException {
    if(args.length <1){
        System.out.println("JavaHBaseMapPartitionExample {tableName} is missing an argument");
        return;
    }
    final String tableName = args[0];
    SparkConf sparkConf = new SparkConf().setAppName("HBaseMapPartitionExample " + tableName);
    JavaSparkContext jsc = new JavaSparkContext(sparkConf);
    try{
        List<byte []>  list = new ArrayList();
        list.add(Bytes.toBytes("1"));
        list.add(Bytes.toBytes("2"));
        list.add(Bytes.toBytes("3"));
        list.add(Bytes.toBytes("4"));
        list.add(Bytes.toBytes("5"));
        JavaRDD<byte []> rdd = jsc.parallelize(list);
        Configuration hbaseconf = HBaseConfiguration.create();
        JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, hbaseconf);
        JavaRDD  getrdd = hbaseContext.mapPartitions(rdd, new
FlatMapFunction<Tuple2<Iterator<byte[]>,Connection>, Object>() {
            public Iterator call(Tuple2<Iterator<byte[]>, Connection> t)
                    throws Exception {
                Table table = t._2.getTable(TableName.valueOf(tableName));
                //go through rdd
                List<String> list = new ArrayList<String>();
                while(t._1.hasNext()){
                    byte[] bytes = t._1.next();
                    Result result = table.get(new Get(bytes));
                    Iterator<Cell> it = result.listCells().iterator();
                    StringBuilder sb = new StringBuilder();
                    sb.append(Bytes.toString(result.getRow()) + ":");
                    while(it.hasNext()){
                        Cell cell = it.next();
                        String column = Bytes.toString(cell.getQualifierArray());
                        if(column.equals("counter")){
                            sb.append("(" + column + "," + Bytes.toLong(cell.getValueArray()) + ")");
                        } else {
                            sb.append("(" + column + "," + Bytes.toString(cell.getValueArray()) + ")");
                        }
                    }
                    list.add(sb.toString());
                }
                return list.iterator();
            }
        });
        List<byte[]> resultList = getrdd.collect();
        if(null == resultList || 0 == resultList.size()){
            System.out.println("Nothing matches!");
        }else{
            for(int i =0; i< resultList.size(); i++){
                System.out.println(resultList.get(i));
            }
        }
    } finally {
        jsc.stop();
    }
}
```

Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseMapPartitionExample文件：

```
def main(args: Array[String]) {
  if (args.length < 1) {
    println("HBaseMapPartitionExample {tableName} is missing an argument")
    return
  }
  val tableName = args(0)
```

```
  val sparkConf = new SparkConf().setAppName("HBaseMapPartitionExample " + tableName)
  val sc = new SparkContext(sparkConf)
  try {
    //[(Array[Byte])]
    val rdd = sc.parallelize(Array(
      Bytes.toBytes("1"),
      Bytes.toBytes("2"),
      Bytes.toBytes("3"),
      Bytes.toBytes("4"),
      Bytes.toBytes("5")))
    val conf = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, conf)
    val b = new StringBuilder
    val getRdd = rdd.hbaseMapPartitions[String](hbaseContext, (it, connection) => {
      val table = connection.getTable(TableName.valueOf(tableName))
      it.map{r =>
        //batching would be faster.  This is just an example
        val result = table.get(new Get(r))
        val it = result.listCells().iterator()
        b.append(Bytes.toString(result.getRow) + ":")
        while (it.hasNext) {
          val cell = it.next()
          val q = Bytes.toString(cell.getQualifierArray)
          if (q.equals("counter")) {
            b.append("(" + q + "," + Bytes.toLong(cell.getValueArray) + ")")
          } else {
            b.append("(" + q + "," + Bytes.toString(cell.getValueArray) + ")")
          }
        }
        b.toString()
      }
    })
    getRdd.collect().foreach(v => println(v))
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseMapPartitionExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseMapPartitionExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.hbasecontext.JavaHBaseMapPartitionExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseMapPartitionExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

28.5.4.10 SparkStreaming 批量写入 HBase 表

场景说明

用户可以在Spark应用程序中使用HBaseContext的方式去操作HBase，使用streamBulkPut接口将流数据写入Hbase表中。

数据规划

1. 在客户端执行**hbase shell**进入HBase命令行。
2. 在HBase命令执行下面的命令创建HBase表：

   create *'streamingTable','cf1'*
3. 在客户端另外一个session通过linux命令构造一个端口进行接收数据（不同操作系统的机器，命令可能不同，suse尝试使用netcat -lk 9999）：

   nc -lk *9999*

   说明

   在构造一个端口进行接收数据时，需要在客户端所在服务器上安装netcat

开发思路

1. 使用SparkStreaming持续读取特定端口的数据。
2. 将读取到的Dstream通过streamBulkPut接口写入hbase表中。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“$SPARK_HOME”）下。

  说明

  若运行“Spark on HBase”样例程序，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”），将配置项“spark.inputFormat.cache.enabled”设置为“false”。

提交命令

假设用例代码打包后的jar包名为spark-hbaseContext-test-1.0.jar，并将jar包放在客户端“$SPARK_HOME”目录下，以下命令均在“$SPARK_HOME”目录执行，Java接口对应的类名前有Java字样，请参考具体样例代码进行书写。

- yarn-client模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例），${ip}请使用实际执行nc -lk 9999的命令的机器ip

  bin/spark-submit --master yarn --deploy-mode client --class com.huawei.bigdata.spark.examples.streaming.JavaHBaseStreamingBulkPutExample SparkOnHbaseJavaExample-1.0.jar ${ip} 9999 streamingTable cf1

python版本（文件名等请与实际保持一致，此处仅为示例）

bin/spark-submit --master yarn --jars SparkOnHbaseJavaExample-1.0.jar HBaseStreamingBulkPutExample.py ${ip} 9999 streamingTable cf1

- yarn-cluster模式：

  java/scala版本（类名等请与实际代码保持一致，此处仅为示例），${ip}请使用实际执行nc -lk 9999的命令的机器ip

  bin/spark-submit --master yarn --deploy-mode client --deploy-mode cluster --class com.huawei.bigdata.spark.examples.streaming.JavaHBaseStreamingBulkPutExample SparkOnHbaseJavaExample-1.0.jar ${ip} 9999 streamingTable cf1

  python版本（文件名等请与实际保持一致，此处仅为示例）

  bin/spark-submit --master yarn --deploy-mode cluster --jars SparkOnHbaseJavaExample-1.0.jar HBaseStreamingBulkPutExample.py ${ip} 9999 streamingTable cf1

Java 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample中JavaHBaseStreamingBulkPutExample文件：

说明

代码中通过awaitTerminationOrTimeout()方法设置了任务超时时间（单位为毫秒），建议根据期望的任务运行时间调整参数大小。

```
public static void main(String[] args) throws IOException {
  if (args.length < 4) {
    System.out.println("JavaHBaseBulkPutExample  " +
          "{host} {port} {tableName}");
    return;
  }
  String host = args[0];
  String port = args[1];
  String tableName = args[2];
  String columnFamily = args[3];
  SparkConf sparkConf =
        new SparkConf().setAppName("JavaHBaseStreamingBulkPutExample " +
              tableName + ":" + port + ":" + tableName);
  JavaSparkContext jsc = new JavaSparkContext(sparkConf);
  try {
    JavaStreamingContext jssc =
          new JavaStreamingContext(jsc, new Duration(1000));
    JavaReceiverInputDStream<String> javaDstream =
          jssc.socketTextStream(host, Integer.parseInt(port));
    Configuration conf = HBaseConfiguration.create();
    JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
    hbaseContext.streamBulkPut(javaDstream,
          TableName.valueOf(tableName),
          new PutFunction(columnFamily));
    jssc.start();
    jssc.awaitTerminationOrTimeout(60000);
    jssc.stop(false,true);
  }catch(InterruptedException e){
    e.printStackTrace();
  } finally {
    jsc.stop();
  }
}
```


Scala 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample中HBaseStreamingBulkPutExample文件：

说明

代码中通过awaitTerminationOrTimeout()方法设置了任务超时时间（单位为毫秒），建议根据期望的任务运行时间调整参数大小。

```
def main(args: Array[String]): Unit = {
  val host = args(0)
  val port = args(1)
  val tableName = args(2)
  val columnFamily = args(3)
  val conf = new SparkConf()
  conf.setAppName("HBase Streaming Bulk Put Example")
  val sc = new SparkContext(conf)
  try {
    val config = HBaseConfiguration.create()
    val hbaseContext = new HBaseContext(sc, config)
    val ssc = new StreamingContext(sc, Seconds(1))
    val lines = ssc.socketTextStream(host, port.toInt)
    hbaseContext.streamBulkPut[String](lines,
      TableName.valueOf(tableName),
      (putRecord) => {
        if (putRecord.length() > 0) {
          val put = new Put(Bytes.toBytes(putRecord))
          put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes("foo"), Bytes.toBytes("bar"))
          put
        } else {
          null
        }
      })
    ssc.start()
    ssc.awaitTerminationOrTimeout(60000)
    ssc.stop(stopSparkContext = false)
  } finally {
    sc.stop()
  }
}
```

Python 样例代码

下面代码片段仅为演示，具体代码参见SparkOnHbasePythonExample中HBaseStreamingBulkPutExample文件：

```
# -*- coding:utf-8 -*-
"""
【说明】
由于pyspark不提供Hbase相关api,本样例使用Python调用Java的方式实现
"""
from py4j.java_gateway import java_import
from pyspark.sql import SparkSession
# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("JavaHBaseStreamingBulkPutExample")\
        .getOrCreate()
# 向sc._jvm中导入要运行的类
java_import(spark._jvm,
'com.huawei.bigdata.spark.examples.streaming.JavaHBaseStreamingBulkPutExample')
# 创建类实例并调用方法，传递sc._jsc参数
spark._jvm.JavaHBaseStreamingBulkPutExample().execute(spark._jsc, sys.argv)
# 停止SparkSession
spark.stop()
```

28.5.5 Spark 从 HBase 读取数据再写入 HBase 样例程序

28.5.5.1 Spark 从 HBase 读取数据再写入 HBase 样例程序（Java）

场景说明

假定HBase的table1表存储用户当天消费的金额信息，table2表存储用户历史消费的金额信息。

现table1表有记录key=1,cf:cid=100，表示用户1在当天消费金额为100元。

table2表有记录key=1,cf:cid=1000，表示用户1的历史消息记录金额为1000元。

基于某些业务要求，要求开发Spark应用程序实现如下功能：

根据用户名累计用户的历史消费金额，即用户总消费金额=100(用户当天的消费金额) + 1000(用户历史消费金额)。

上例所示，运行结果table2表用户key=1的总消费金融为cf:cid=1100元。

数据规划

使用Spark-Beeline工具创建Spark和HBase表table1、table2，并通过HBase插入数据。

步骤1 确保JDBCServer已启动。然后在Spark2x客户端，使用Spark-Beeline工具执行如下操作。

步骤2 使用Spark-Beeline工具创建Spark表table1。

```
create table table1
(
key string,
cid string
)
using org.apache.spark.sql.hbase.HBaseSource
options(
hbaseTableName "table1",
keyCols "key",
colsMapping "cid=cf.cid");
```

步骤3 通过HBase插入数据，命令如下：

```
put 'table1', '1', 'cf:cid', '100'
```

步骤4 使用Spark-Beeline工具创建Spark表table2。

```
create table table2
(
```


key string,

cid string

)

using org.apache.spark.sql.hbase.HBaseSource

options(

hbaseTableName "table2",

keyCols "key",

colsMapping "cid=cf.cid");

步骤5 通过HBase插入数据，命令如下：

put 'table2', '1', 'cf:cid', '1000'

----结束

开发思路

1. 查询table1表的数据。
2. 根据table1表数据的key值去table2表做查询。
3. 把前两步相应的数据记录做相加操作。
4. 把上一步骤的结果写到table2表。

打包项目

1. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
2. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“ /opt/female/” ）下。

说明

运行样例程序前，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”）。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行Java或Scala样例代码

  bin/spark-submit --conf spark.yarn.user.classpath.first=true --class *com.huawei.bigdata.spark.examples.SparkHbasetoHbase* **--master yarn --deploy-mode client** */opt/female/SparkHbasetoHbase-1.0.jar*

- 运行**Python**样例程序

说明

- 由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。将所提供Java代码使用maven打包成jar，并放在相同目录下，运行python程序时要使用--jars把jar包加载到classpath中。

bin/spark-submit --master yarn --deploy-mode client --conf spark.yarn.user.classpath.first=true --jars */opt/female/SparkHbasetoHbasePythonExample/SparkHbasetoHbase-1.0.jar,*/opt/female/protobuf-java-2.5.0.jar */opt/female/SparkHbasetoHbasePythonExample/SparkHbasetoHbasePythonExample.py*

28.5.5.2 Spark 从 HBase 读取数据再写入 HBase 样例程序（Java）

功能介绍

用户可以使用Spark调用HBase接口来操作HBase table1表，然后把table1表的数据经过分析后写到HBase table2表中。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SparkHbasetoHbase。

```
/**
 * 从table1表读取数据，根据key值去table2表获取相应记录，把两者数据后，更新到table2表
*/
public class SparkHbasetoHbase {

  public static void main(String[] args) throws Exception {
    SparkConf conf = new SparkConf().setAppName("SparkHbasetoHbase");
    conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
    conf.set("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator");
    JavaSparkContext jsc = new JavaSparkContext(conf);
    // 建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
    Configuration hbConf = HBaseConfiguration.create(jsc.hadoopConfiguration());

    // 声明表的信息
    Scan scan = new org.apache.hadoop.hbase.client.Scan();
    scan.addFamily(Bytes.toBytes("cf"));//colomn family
    org.apache.hadoop.hbase.protobuf.generated.ClientProtos.Scan proto = ProtobufUtil.toScan(scan);
    String scanToString = Base64.encodeBytes(proto.toByteArray());
    hbConf.set(TableInputFormat.INPUT_TABLE, "table1");//table name
    hbConf.set(TableInputFormat.SCAN, scanToString);

    // 通过spark接口获取表中的数据
    JavaPairRDD rdd = jsc.newAPIHadoopRDD(hbConf, TableInputFormat.class,
ImmutableBytesWritable.class, Result.class);

    // 遍历hbase table1表中的每一个partition, 然后更新到Hbase table2表
    // 如果数据条数较少，也可以使用rdd.foreach()方法
    rdd.foreachPartition(
      new VoidFunction<Iterator<Tuple2<ImmutableBytesWritable, Result>>>() {
        public void call(Iterator<Tuple2<ImmutableBytesWritable, Result>> iterator) throws Exception {
          hBaseWriter(iterator);
        }
      }
    );

    jsc.stop();
  }

/**
```



```
  * 在executor端更新table2表记录
  *
  * @param iterator table1表的partition数据
  */
 private static void hBaseWriter(Iterator<Tuple2<ImmutableBytesWritable, Result>> iterator) throws
IOException {
   // 准备读取hbase
   String tableName = "table2";
   String columnFamily = "cf";
   String qualifier = "cid";
   Configuration conf = HBaseConfiguration.create();
   Connection connection = null;
   Table table = null;
   try {
     connection = ConnectionFactory.createConnection(conf);
     table = connection.getTable(TableName.valueOf(tableName));
     List<Get> rowList = new ArrayList<Get>();
     List<Tuple2<ImmutableBytesWritable, Result>> table1List = new
ArrayList<Tuple2<ImmutableBytesWritable, Result>>();
     while (iterator.hasNext()) {
       Tuple2<ImmutableBytesWritable, Result> item = iterator.next();
       Get get = new Get(item._2().getRow());
       table1List.add(item);
       rowList.add(get);
     }
     // 获取table2表记录
     Result[] resultDataBuffer = table.get(rowList);
     // 修改table2表记录
     List<Put> putList = new ArrayList<Put>();
     for (int i = 0; i < resultDataBuffer.length; i++) {
       Result resultData = resultDataBuffer[i];//hbase2 row
       if (!resultData.isEmpty()) {
         // 查询hbase1Value
         String hbase1Value = "";
         Iterator<Cell> it = table1List.get(i)._2().listCells().iterator();
         while (it.hasNext()) {
           Cell c = it.next();
           // 判断cf和qualifile是否相同
           if (columnFamily.equals(Bytes.toString(CellUtil.cloneFamily(c)))
             && qualifier.equals(Bytes.toString(CellUtil.cloneQualifier(c)))) {
             hbase1Value = Bytes.toString(CellUtil.cloneValue(c));
           }
         }
         String hbase2Value = Bytes.toString(resultData.getValue(columnFamily.getBytes(),
qualifier.getBytes()));
         Put put = new Put(table1List.get(i)._2().getRow());
         // 计算结果
         int resultValue = Integer.parseInt(hbase1Value) + Integer.parseInt(hbase2Value);
         // 设置结果到put对象
         put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(qualifier),
Bytes.toBytes(String.valueOf(resultValue)));
         putList.add(put);
       }
     }
     if (putList.size() > 0) {
       table.put(putList);
     }
   } catch (IOException e) {
     e.printStackTrace();
   } finally {
     if (table != null) {
       try {
         table.close();
       } catch (IOException e) {
         e.printStackTrace();
       }
     }
     if (connection != null) {
       try {
```

```
        // 关闭Hbase连接
        connection.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
 }
}
```

28.5.5.3 Spark 从 HBase 读取数据再写入 HBase 样例程序（Scala）

功能介绍

用户可以使用Spark调用HBase接口来操作HBase table1表，然后把table1表的数据经过分析后写到HBase table2表中。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.SparkHbasetoHbase。

```
/**
 * 从table1表读取数据，根据key值去table2表获取相应记录，把两者数据后，更新到table2表
 */
object SparkHbasetoHbase {

 case class FemaleInfo(name: String, gender: String, stayTime: Int)

 def main(args: Array[String]) {
  val conf = new SparkConf().setAppName("SparkHbasetoHbase")
  conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  conf.set("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator")
  val sc = new SparkContext(conf)
  // 建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
  val hbConf = HBaseConfiguration.create(sc.hadoopConfiguration)

  // 声明表的信息
  val scan = new Scan()
  scan.addFamily(Bytes.toBytes("cf"))//colomn family
  val proto = ProtobufUtil.toScan(scan)
  val scanToString = Base64.encodeBytes(proto.toByteArray)
  hbConf.set(TableInputFormat.INPUT_TABLE, "table1")//table name
  hbConf.set(TableInputFormat.SCAN, scanToString)

  // 通过spark接口获取表中的数据
  val rdd = sc.newAPIHadoopRDD(hbConf, classOf[TableInputFormat], classOf[ImmutableBytesWritable],
classOf[Result])

  // 遍历hbase table1表中的每一个partition, 然后更新到Hbase table2表
  // 如果数据条数较少，也可以使用rdd.foreach()方法
  rdd.foreachPartition(x => hBaseWriter(x))

  sc.stop()
 }
 /**
  * 在executor端更新table2表记录
  *
  * @param iterator  table1表的partition数据
  */
 def hBaseWriter(iterator: Iterator[(ImmutableBytesWritable, Result)]): Unit = {
  // 准备读取hbase
  val tableName = "table2"
  val columnFamily = "cf"
  val qualifier = "cid"
  val conf = HBaseConfiguration.create()
```



```
    var table: Table = null
    var connection: Connection = null
    try {
      connection = ConnectionFactory.createConnection(conf)
      table = connection.getTable(TableName.valueOf(tableName))
      val iteratorArray = iterator.toArray
      val rowList = new util.ArrayList[Get]()
      for (row <- iteratorArray) {
        val get = new Get(row._2.getRow)
        rowList.add(get)
      }
      // 获取table2表记录
      val resultDataBuffer = table.get(rowList)
      // 修改table2表记录
      val putList = new util.ArrayList[Put]()
      for (i <- 0 until iteratorArray.size) {
        val resultData = resultDataBuffer(i) //hbase2 row
        if (!resultData.isEmpty) {
          // 查询hbase1Value
          var hbase1Value = ""
          val it = iteratorArray(i)._2.listCells().iterator()
          while (it.hasNext) {
            val c = it.next()
            // 判断cf和qualifile是否相同
            if (columnFamily.equals(Bytes.toString(CellUtil.cloneFamily(c)))
              && qualifier.equals(Bytes.toString(CellUtil.cloneQualifier(c)))) {
              hbase1Value = Bytes.toString(CellUtil.cloneValue(c))
            }
          }
          val hbase2Value = Bytes.toString(resultData.getValue(columnFamily.getBytes, qualifier.getBytes))
          val put = new Put(iteratorArray(i)._2.getRow)
          // 计算结果
          val resultValue = hbase1Value.toInt + hbase2Value.toInt
          // 设置结果到put对象
          put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes(qualifier),
Bytes.toBytes(resultValue.toString))
          putList.add(put)
        }
      }
      if (putList.size() > 0) {
        table.put(putList)
      }
    } catch {
      case e: IOException =>
        e.printStackTrace();
    } finally {
      if (table != null) {
        try {
          table.close()
        } catch {
          case e: IOException =>
            e.printStackTrace();
        }
      }
      if (connection != null) {
        try {
          //关闭Hbase连接
          connection.close()
        } catch {
          case e: IOException =>
            e.printStackTrace()
        }
      }
    }
  }
}


/**
```

```
 * 序列化辅助类
 */
class MyRegistrator extends KryoRegistrator {
  override def registerClasses(kryo: Kryo) {
    kryo.register(classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable])
    kryo.register(classOf[org.apache.hadoop.hbase.client.Result])
    kryo.register(classOf[Array[(Any, Any)]])
    kryo.register(classOf[Array[org.apache.hadoop.hbase.Cell]])
    kryo.register(classOf[org.apache.hadoop.hbase.NoTagsKeyValue])
    kryo.register(classOf[org.apache.hadoop.hbase.protobuf.generated.ClientProtos.RegionLoadStats])
  }
}
```

28.5.5.4 Spark 从 HBase 读取数据再写入 HBase 样例程序（Python）

功能介绍

用户可以使用Spark调用HBase接口来操作HBase table1表，然后把table1表的数据经过分析后写到HBase table2表中。

代码样例

由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。

下面代码片段仅为演示，具体代码参见SparkHbasetoHbasePythonExample：

```
# -*- coding:utf-8 -*-

from py4j.java_gateway import java_import
from pyspark.sql import SparkSession

# 创建SparkSession，设置kryo序列化
spark = SparkSession\
        .builder\
        .appName("SparkHbasetoHbase") \
        .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer") \
        .config("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator") \
        .getOrCreate()

# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.SparkHbasetoHbase')

# 创建类实例并调用方法
spark._jvm.SparkHbasetoHbase().hbasetohbase(spark._jsc)

# 停止SparkSession
spark.stop()
```

28.5.6 Spark 从 Hive 读取数据再写入 HBase 样例程序

28.5.6.1 Spark 从 Hive 读取数据再写入 HBase 样例程序开发思路

场景说明

假定Hive的person表存储用户当天消费的金额信息，HBase的table2表存储用户历史消费的金额信息。

现person表有记录name=1,account=100，表示用户1在当天消费金额为100元。

table2表有记录key=1,cf:cid=1000，表示用户1的历史消息记录金额为1000元。

基于某些业务要求，要求开发Spark应用程序实现如下功能：

根据用户名累计用户的历史消费金额，即用户总消费金额=100(用户当天的消费金额) + 1000(用户历史消费金额)。

上例所示，运行结果table2表用户key=1的总消费金融为cf:cid=1100元。

数据规划

在开始开发应用前，需要创建Hive表，命名为person，并插入数据。同时，创建HBase table2表，用于将分析后的数据写入。

步骤1 将原日志文件放置到HDFS系统中。

1. 在本地新建一个空白的log1.txt文件，并在文件内写入如下内容：
   ```
   1,100
   ```
2. 在HDFS中新建一个目录/tmp/input，并将log1.txt文件上传至此目录。
   a. 在Linux系统HDFS客户端使用命令***hadoop fs -mkdir /tmp/input***（hdfs dfs命令有同样的作用），创建对应目录。
   b. 在Linux系统HDFS客户端使用命令***hadoop fs -put*** *log1.txt /tmp/input*，上传数据文件。

步骤2 将导入的数据放置在Hive表里。

首先，确保JDBCServer已启动。然后使用Beeline工具，创建Hive表，并插入数据。

1. 执行如下命令，创建命名为person的Hive表。
   create table person
   (
   name STRING,
   account INT
)
   ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ESCAPED BY '\\' STORED AS TEXTFILE;
2. 执行如下命令插入数据。
   load data inpath '/tmp/input/log1.txt' into table person;

步骤3 创建HBase表。

确保JDBCServer已启动，然后使用Spark-beeline工具，创建HBase表，并插入数据。

1. 执行如下命令，创建命名为table2的HBase表。
   create table table2
   (
   key string,
   cid string
)
   using org.apache.spark.sql.hbase.HBaseSource
   options(
   hbaseTableName "table2",
   keyCols "key",

colsMapping "cid=cf.cid");

2. 通过HBase插入数据，执行如下命令。

   put 'table2', '1', 'cf:cid', '1000'

----结束

开发思路

1. 查询Hive person表的数据。
2. 根据person表数据的key值去table2表做查询。
3. 把前两步相应的数据记录做相加操作。
4. 把上一步骤的结果写到table2表。

打包项目

1. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
2. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/female/”）下。

说明

运行样例程序前，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”）。

运行任务

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行Java或Scala样例代码
  - **bin/spark-submit --class** *com.huawei.bigdata.spark.examples.SparkHivetoHbase* **--master yarn --deploy-mode client** */opt/female/SparkHivetoHbase-1.0.jar*
- 运行**Python**样例程序

  说明

  - 由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。将所提供Java代码使用maven打包成jar，并放在相同目录下，运行python程序时要使用--jars把jar包加载到classpath中。

  - **bin/spark-submit --master yarn --deploy-mode client --jars** */opt/female/SparkHivetoHbasePythonExample/SparkHivetoHbase-1.0.jar /opt/female/SparkHivetoHbasePythonExample/SparkHivetoHbasePythonExample.py*

28.5.6.2 Spark 从 Hive 读取数据再写入 HBase 样例程序（Java）

功能介绍

在Spark应用中，通过使用Spark调用Hive接口来操作hive表，然后把Hive表的数据经过分析后写到HBase表。

代码样例

下面代码片段仅为演示，具体代码参见：
com.huawei.bigdata.spark.examples.SparkHivetoHbase

```
/**
 * 从hive表读取数据，根据key值去hbase表获取相应记录，把两者数据做操作后，更新到hbase表
 */
public class SparkHivetoHbase {

 public static void main(String[] args) throws Exception {
  // 通过spark接口获取表中的数据
  SparkConf conf = new SparkConf().setAppName("SparkHivetoHbase");
  JavaSparkContext jsc = new JavaSparkContext(conf);
  HiveContext sqlContext = new org.apache.spark.sql.hive.HiveContext(jsc);
  SparkSession spark = SparkSession

  Dataset<Row> dataFrame = sqlContext.sql("select name, account from person");

  // 遍历hive表中的每一个partition, 然后更新到hbase表
  // 如果数据条数较少，也可以使用foreach()方法
  dataFrame.toJavaRDD().foreachPartition(
    new VoidFunction<Iterator<Row>>() {
      public void call(Iterator<Row> iterator) throws Exception {
        hBaseWriter(iterator);
      }
    }
  );

  spark.stop();
 }

 /**
  * 在executor端更新hbase表记录
  *
  * @param iterator hive表的partition数据
  */
 private static void hBaseWriter(Iterator<Row> iterator) throws IOException {
  // 读取hbase
  String tableName = "table2";
  String columnFamily = "cf";
  Configuration conf = HBaseConfiguration.create();
  Connection connection = null;
  Table table = null;
  try {
    connection = ConnectionFactory.createConnection(conf);
    table = connection.getTable(TableName.valueOf(tableName));
    List<Row> table1List = new ArrayList<Row>();
    List<Get> rowList = new ArrayList<Get>();
    while (iterator.hasNext()) {
      Row item = iterator.next();
      Get get = new Get(item.getString(0).getBytes());
      table1List.add(item);
      rowList.add(get);
    }
    // 获取hbase表记录
    Result[] resultDataBuffer = table.get(rowList);
    // 修改hbase表记录
    List<Put> putList = new ArrayList<Put>();
```

```
    for (int i = 0; i < resultDataBuffer.length; i++) {
      // hive表值
      Result resultData = resultDataBuffer[i];
      if (!resultData.isEmpty()) {
        // get hiveValue
        int hiveValue = table1List.get(i).getInt(1);
        // 根据列簇和列，获取hbase值
        String hbaseValue = Bytes.toString(resultData.getValue(columnFamily.getBytes(), "cid".getBytes()));
        Put put = new Put(table1List.get(i).getString(0).getBytes());
        // 计算结果
        int resultValue = hiveValue + Integer.valueOf(hbaseValue);
        // 设置结果到put对象
        put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes("cid"),
Bytes.toBytes(String.valueOf(resultValue)));
        putList.add(put);
      }
    }
    if (putList.size() > 0) {
      table.put(putList);
    }
  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    if (table != null) {
      try {
        table.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    if (connection != null) {
      try {
        // 关闭Hbase连接.
        connection.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
 }
}
```

28.5.6.3 Spark 从 Hive 读取数据再写入 HBase 样例程序（Scala）

功能介绍

在Spark应用中，通过使用Spark调用Hive接口来操作hive表，然后把Hive表的数据经过分析后写到HBase表。

代码样例

下面代码片段仅为演示，具体代码参见：
com.huawei.bigdata.spark.examples.SparkHivetoHbase

```
/**
 * 从hive表读取数据，根据key值去hbase表获取相应记录，把两者数据做操作后，更新到hbase表
 */
object SparkHivetoHbase {
 case class FemaleInfo(name: String, gender: String, stayTime: Int)
 def main(args: Array[String]) {
   // 通过spark接口获取表中的数据

    val spark = SparkSession
     .builder()
     .appName("SparkHiveHbase")
     .config("spark.sql.warehouse.dir", "spaek-warehouse")
```



```
    .enableHiveSupport()
    .getOrCreate()

  import spark.implicits._
  val dataFrame = spark.sql("select name, account from person")
  // 遍历hive表中的每一个partition, 然后更新到hbase表
  // 如果数据条数较少，也可以使用foreach()方法
  dataFrame.rdd.foreachPartition(x => hBaseWriter(x))
  spark.stop()
}
/**
 * 在executor端更新hbase表记录
 *
 * @param iterator hive表的partition数据
 */
def hBaseWriter(iterator: Iterator[Row]): Unit = {
  // 读取hbase
  val tableName = "table2"
  val columnFamily = "cf"
  val conf = HBaseConfiguration.create()
  var table: Table = null
  var connection: Connection = null
  try {
    connection = ConnectionFactory.createConnection(conf)
    table = connection.getTable(TableName.valueOf(tableName))
    val iteratorArray = iterator.toArray
    val rowList = new util.ArrayList[Get]()
    for (row <- iteratorArray) {
      val get = new Get(row.getString(0).getBytes)
      rowList.add(get)
    }
    // 获取hbase表记录
    val resultDataBuffer = table.get(rowList)
    // 修改hbase表记录
    val putList = new util.ArrayList[Put]()
    for (i <- 0 until iteratorArray.size) {
      // hbase row
      val resultData = resultDataBuffer(i)
      if (!resultData.isEmpty) {
        // hive表值
        var hiveValue = iteratorArray(i).getInt(1)
        // 根据列簇和列，获取hbase值
        val hbaseValue = Bytes.toString(resultData.getValue(columnFamily.getBytes, "cid".getBytes))
        val put = new Put(iteratorArray(i).getString(0).getBytes)
        // 计算结果
        val resultValue = hiveValue + hbaseValue.toInt
        // 设置结果到put对象
        put.addColumn(Bytes.toBytes(columnFamily), Bytes.toBytes("cid"),
Bytes.toBytes(resultValue.toString))
        putList.add(put)
      }
    }
    if (putList.size() > 0) {
      table.put(putList)
    }
  } catch {
    case e: IOException =>
      e.printStackTrace();
  } finally {
    if (table != null) {
      try {
        table.close()
      } catch {
        case e: IOException =>
          e.printStackTrace();
      }
    }
    if (connection != null) {
      try {
```



```
        //关闭Hbase连接.
        connection.close()
      } catch {
        case e: IOException =>
          e.printStackTrace()
      }
    }
  }
}

}
```

28.5.6.4 Spark 从 Hive 读取数据再写入 HBase 样例程序（Python）

功能介绍

在Spark应用中，通过使用Spark调用Hive接口来操作hive表，然后把Hive表的数据经过分析后写到HBase表。

代码样例

由于pyspark不提供Hbase相关api，本样例使用Python调用Java的方式实现。

下面代码片段仅为演示，具体代码参见SparkHivetoHbasePythonExample：

```
# -*- coding:utf-8 -*-

from py4j.java_gateway import java_import
from pyspark.sql import SparkSession

# 创建SparkSession
spark = SparkSession\
        .builder\
        .appName("SparkHivetoHbase") \
        .getOrCreate()

# 向sc._jvm中导入要运行的类
java_import(spark._jvm, 'com.huawei.bigdata.spark.examples.SparkHivetoHbase')

# 创建类实例并调用方法
spark._jvm.SparkHivetoHbase().hivetohbase(spark._jsc)

# 停止SparkSession
spark.stop()
```

28.5.7 Spark Streaming 对接 Kafka0-10 样例程序

28.5.7.1 Spark Streaming 对接 Kafka0-10 样例程序开发思路

场景说明

假定某个业务Kafka每1秒就会收到1个单词记录。

基于某些业务要求，开发的Spark应用程序实现如下功能：

实时累加计算每个单词的记录总数。

“log1.txt”示例文件：

```
LiuYang
YuanJing
```



```
GuoYijun
CaiXuyu
Liyuan
FangBo
LiuYang
YuanJing
GuoYijun
CaiXuyu
FangBo
```

数据规划

Spark Streaming样例工程的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户）。

1. 确保集群安装完成，包括HDFS、Yarn、Spark和Kafka。
2. 本地新建文件“input_data1.txt”，将“log1.txt”的内容复制保存到“input_data1.txt”。

   在客户端安装节点下创建文件目录：“/home/data”。将上述文件上传到此“/home/data”目录下。
3. 创建Topic。

   {zkQuorum}表示ZooKeeper集群信息，格式为IP:port。

   $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper {zkQuorum}/kafka --replication-factor 1 --partitions 3 --topic {Topic}
4. 启动Kafka的Producer，向Kafka发送数据。

   java -cp {ClassPath} com.huawei.bigdata.spark.examples.StreamingExampleProducer {BrokerList} {Topic}

   其中，ClassPath除样例工程jar包路径外，还应包含Spark客户端Kafka jar包的绝对路径，例如：/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/jars/streamingClient010/*:{ClassPath}

开发思路

1. 接收Kafka中数据，生成相应DStream。
2. 对单词记录进行分类统计。
3. 计算结果，并进行打印。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“ /opt” ）下。

运行任务

在运行样例程序时需要指定<checkpointDir> <brokers> <topic> <batchTime>，其中<checkPointDir>指应用程序结果备份到HDFS的路径，<brokers>指获取元数据的Kafka地址，<topic>指读取Kafka上的topic名称，<batchTime>指Streaming分批的处理间隔。

说明

由于Spark Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Streaming Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- **Spark Streaming读取Kafka 0-10 Write To Print代码样例**

  bin/spark-submit --master yarn --deploy-mode client --jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}") --class com.huawei.bigdata.spark.examples.KafkaWordCount /opt/SparkStreamingKafka010JavaExample-1.0.jar <checkpointDir> <brokers> <topic> <batchTime>

- **Spark Streaming Write To Kafka 0-10代码样例：**

  bin/spark-submit --master yarn --deploy-mode client --jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}") --class com.huawei.bigdata.spark.examples.JavaDstreamKafkaWriter /opt/SparkStreamingKafka010JavaExample-1.0.jar <groupId> <brokers> <topics>

28.5.7.2 Spark Streaming 对接 Kafka0-10 样例程序（Java）

功能介绍

在Spark应用中，通过使用Streaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数，或将数据写入Kafka0-10。

Streaming 读取 Kafka0-10 代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.KafkaWordCount。

```
/**
 * 从Kafka的一个或多个主题消息。
 * <checkPointDir>是Spark Streaming检查点目录。
 * <brokers>是用于自举，制作人只会使用它来获取元数据
 * <topics>是要消费的一个或多个kafka主题的列表
 * <batchTime>是Spark Streaming批次持续时间（以秒为单位）。
 */
public class KafkaWordCount
{
  public static void main(String[] args) {
    JavaStreamingContext ssc = createContext(args);
    //启动Streaming系统。
    ssc.start();
    try {
      ssc.awaitTermination();
    } catch (InterruptedException e) {
    }
  }

  private static JavaStreamingContext createContext(String[] args) {
    String checkPointDir = args[0];
    String brokers = args[1];
    String topics = args[2];
```



```
    String batchTime = args[3];

    // 新建一个Streaming启动环境。
    SparkConf sparkConf = new SparkConf().setAppName("KafkaWordCount");
    JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, new
Duration(Long.parseLong(batchTime) * 1000));

    //配置Streaming的CheckPoint目录。
    //由于窗口概念的存在，此参数是必需的。
    ssc.checkpoint(checkPointDir);

    // 获取kafka使用的topic列表。
    String[] topicArr = topics.split(",");
    Set<String> topicSet = new HashSet<String>(Arrays.asList(topicArr));
    Map<String, Object> kafkaParams = new HashMap();
    kafkaParams.put("bootstrap.servers", brokers);
    kafkaParams.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    kafkaParams.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
    kafkaParams.put("group.id", "DemoConsumer");
    LocationStrategy locationStrategy = LocationStrategies.PreferConsistent();
    ConsumerStrategy consumerStrategy = ConsumerStrategies.Subscribe(topicSet, kafkaParams);

    // 用brokers and topics新建direct kafka stream
    //从Kafka接收数据并生成相应的DStream。
    JavaInputDStream<ConsumerRecord<String, String>> messages = KafkaUtils.createDirectStream(ssc,
locationStrategy, consumerStrategy);

    // 获取每行中的字段属性。
    JavaDStream<String> lines = messages.map(new Function<ConsumerRecord<String, String>, String>() {
      @Override
      public String call(ConsumerRecord<String, String> tuple2) throws Exception {
        return tuple2.value();
      }
    });

    // 汇总计算字数的总时间。
    JavaPairDStream<String, Integer> wordCounts = lines.mapToPair(
        new PairFunction<String, String, Integer>() {
       @Override
          public Tuple2<String, Integer> call(String s) {
            return new Tuple2<String, Integer>(s, 1);
          }
        }).reduceByKey(new Function2<Integer, Integer, Integer>() {
      @Override
      public Integer call(Integer i1, Integer i2) {
        return i1 + i2;
      }
    }).updateStateByKey(
        new Function2<List<Integer>, Optional<Integer>, Optional<Integer>>() {
          @Override
          public Optional<Integer> call(List<Integer> values, Optional<Integer> state) {
            int out = 0;
            if (state.isPresent()) {
              out += state.get();
            }
            for (Integer v : values) {
              out += v;
            }
            return Optional.of(out);
          }
        });

    // 打印结果
    wordCounts.print();
    return ssc;
  }
}
```

Streaming Write To Kafka 0-10 样例代码

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.DstreamKafkaWriter。

说明

建议使用新的API **createDirectStream**代替旧的API **createStream**进行应用程序开发。旧的API仍然可以使用，但新的API性能和稳定性更好。

```
/**
 * 参数解析:
 * <groupId>为客户的组编号。
 * <brokers>为获取元数据的Kafka地址。
 * <topic>为Kafka中订阅的主题。
 */
public class JavaDstreamKafkaWriter {

 public static void main(String[] args) throws InterruptedException {

   if (args.length != 3) {
     System.err.println("Usage: JavaDstreamKafkaWriter <groupId> <brokers> <topic>");
     System.exit(1);
   }

   final String groupId = args[0];
   final String brokers = args[1];
   final String topic = args[2];

   SparkConf sparkConf = new SparkConf().setAppName("KafkaWriter");

   // 填写Kafka的properties。
   Map<String, Object> kafkaParams = new HashMap<String, Object>();
   kafkaParams.put("value.deserializer","org.apache.kafka.common.serialization.StringDeserializer");
   kafkaParams.put("key.deserializer" , "org.apache.kafka.common.serialization.StringDeserializer");
   kafkaParams.put("value.serializer" , "org.apache.kafka.common.serialization.ByteArraySerializer");
   kafkaParams.put("key.serializer" , "org.apache.kafka.common.serialization.StringSerializer");
   kafkaParams.put("bootstrap.servers", brokers);
   kafkaParams.put("group.id", groupId);
   kafkaParams.put("auto.offset.reset", "smallest");

   // 创建Java Spark Streaming的Context。
   JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.milliseconds(500));

   // 填写写入Kafka的数据。
   List<String> sentData = new ArrayList();
   sentData.add("kafka_writer_test_msg_01");
   sentData.add("kafka_writer_test_msg_02");
   sentData.add("kafka_writer_test_msg_03");

   // 创建Java RDD队列。
   Queue<JavaRDD<String>> sent = new LinkedList();
   sent.add(ssc.sparkContext().parallelize(sentData));

   // 创建写数据的Java DStream。
   JavaDStream wStream = ssc.queueStream(sent);

   // 写入Kafka。
   JavaDStreamKafkaWriterFactory.fromJavaDStream(wStream).writeToKafka(
       JavaConverters.mapAsScalaMapConverter(kafkaParams).asScala(),
       new Function<String, ProducerRecord<String, byte[]>>() {
         public ProducerRecord<String, byte[]> call(String s) throws Exception {
           return new ProducerRecord(topic, s.toString().getBytes());
         }
       });

   ssc.start();
   ssc.awaitTermination();
```



```
  }
}
```

28.5.7.3 Spark Streaming 对接 Kafka0-10 样例程序（Scala）

功能介绍

在Spark应用中，通过使用Streaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数，或将数据写入Kafka0-10。

Streaming 读取 Kafka0-10 代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.KafkaWordCount。

```
/**
 * 从Kafka的一个或多个主题消息。
 * <checkPointDir>是Spark Streaming检查点目录。
 * <brokers>是用于自举，制作人只会使用它来获取元数据
 * <topics>是要消费的一个或多个kafka主题的列表
 * <batchTime>是Spark Streaming批次持续时间（以秒为单位）。
 */
object KafkaWordCount {

 def main(args: Array[String]) {
  val ssc = createContext(args)

  //启动Streaming系统。
  ssc.start()
  ssc.awaitTermination()
 }

 def createContext(args : Array[String]) : StreamingContext = {
  val Array(checkPointDir, brokers, topics, batchSize) = args

  // 新建一个Streaming启动环境。
  val sparkConf = new SparkConf().setAppName("KafkaWordCount")
  val ssc = new StreamingContext(sparkConf, Seconds(batchSize.toLong))

  //配置Streaming的CheckPoint目录。
  //由于窗口概念的存在，此参数是必需的。
  ssc.checkpoint(checkPointDir)

  // 获取kafka使用的topic列表。
  val topicArr = topics.split(",")
  val topicSet = topicArr.toSet
  val kafkaParams = Map[String, String](
   "bootstrap.servers" -> brokers,
   "value.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
   "key.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
   "group.id" -> "DemoConsumer"
  );

  val locationStrategy = LocationStrategies.PreferConsistent
  val consumerStrategy = ConsumerStrategies.Subscribe[String, String](topicSet, kafkaParams)

  // 用brokers and topics新建direct kafka stream
  //从Kafka接收数据并生成相应的DStream。
  val stream = KafkaUtils.createDirectStream[String, String](ssc, locationStrategy, consumerStrategy)

  // 获取每行中的字段属性。
  val tf = stream.transform ( rdd =>
   rdd.map(r => (r.value, 1L))
  )

  // 汇总计算字数的总时间。
```

```
  val wordCounts = tf.reduceByKey(_ + _)
  val totalCounts = wordCounts.updateStateByKey(updataFunc)
  totalCounts.print()

  ssc
 }

 def updataFunc(values : Seq[Long], state : Option[Long]) : Option[Long] =
  Some(values.sum + state.getOrElse(0L))
}
```

Streaming Write To Kafka 0-10 样例代码

下面代码片段仅为演示，具体代码参见com.huawei.bigdata.spark.examples.DstreamKafkaWriter。

说明

建议使用新的API **createDirectStream**代替原有API **createStream**进行应用程序开发。原有API仍然可以使用，但新的API性能和稳定性更好。

```
/**
 * 参数解析:
 * <checkPointDir>为checkPoint目录。
 * <topics>为Kafka中订阅的主题，多以逗号分隔。
 * <brokers>为获取元数据的Kafka地址。
 */
object DstreamKafkaWriterTest1 {

 def main(args: Array[String]) {
  if (args.length != 4) {
   System.err.println("Usage: DstreamKafkaWriterTest <checkPointDir> <brokers> <topic>")
   System.exit(1)
  }

  val Array(checkPointDir, brokers, topic) = args
  val sparkConf = new SparkConf().setAppName("KafkaWriter")

  //填写Kafka的properties。
  val kafkaParams = Map[String, String](
   "bootstrap.servers" -> brokers,
   "value.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
   "key.deserializer" -> "org.apache.kafka.common.serialization.StringDeserializer",
   "value.serializer" -> "org.apache.kafka.common.serialization.ByteArraySerializer",
   "key.serializer" -> "org.apache.kafka.common.serialization.StringSerializer",
   "group.id" -> "dstreamKafkaWriterFt",
   "auto.offset.reset" -> "latest"
  )

  //创建Streaming的context。
  val ssc = new StreamingContext(sparkConf, Milliseconds(500));
  val sentData = Seq("kafka_writer_test_msg_01", "kafka_writer_test_msg_02", "kafka_writer_test_msg_03")

  //创建RDD队列。
  val sent = new mutable.Queue[RDD[String]]()
  sent.enqueue(ssc.sparkContext.makeRDD(sentData))

  //创建写数据的DStream。
  val wStream = ssc.queueStream(sent)

  //使用writetokafka API把数据写入Kafka。
  wStream.writeToKafka(kafkaParams,
   (x: String) => new ProducerRecord[String, Array[Byte]](topic, x.getBytes))

  //启动streaming的context。
  ssc.start()
  ssc.awaitTermination()
```



```
 }
}
```

28.5.8 Spark Structured Streaming 样例程序

28.5.8.1 Spark Structured Streaming 样例程序开发思路

场景说明

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

数据规划

StructuredStreaming样例工程的数据存储在Kafka组件中。向Kafka组件发送数据（需要有Kafka权限用户）。

1. 确保集群安装完成，包括HDFS、Yarn、Spark和Kafka。
2. 创建Topic。

   {zkQuorum}表示ZooKeeper集群信息，格式为IP:port。

   $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper {zkQuorum}/kafka --replication-factor 1 --partitions 1 --topic {Topic}
3. 启动Kafka的Producer，向Kafka发送数据。

   {ClassPath}表示工程jar包的存放路径，详细路径由用户指定，可参考**在Linux环境中编包并运行Spark程序**章节中导出jar包的操作步骤。

   java -cp $SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:{ClassPath} com.huawei.bigdata.spark.examples.KafkaWordCountProducer {BrokerList} {Topic} {messagesPerSec} {wordsPerMessage}

开发思路

1. 接收Kafka中数据，生成相应DataStreamReader。
2. 对单词记录进行分类统计。
3. 计算结果，并进行打印。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/female/”）下。

运行任务

在运行样例程序时需要指定<brokers> <subscribe-type> <topic> <checkpointDir>。

- <brokers>指获取元数据的Kafka地址。
- <subscribe-type>指Kafka订阅类型（如subscribe）。
- <topic>指读取Kafka上的topic名称。

- <checkpointDir>指checkpoint文件存放路径，本地或者HDFS路径下。

说明

由于Spark Structured Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Structured Streaming Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

注意

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，当前版本用户除了这一步外还需要将**$SPARK_HOME/jars/streamingClient010**目录中的kafka-clients jar包复制到**$SPARK_HOME/jars**目录下，否则会报class not found异常。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令分别如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- 运行**Java**或**Scala**样例代码：

  bin/spark-submit --master yarn --deploy-mode client --jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}") --class *com.huawei.bigdata.spark.examples.KafkaWordCount /opt/SparkStructuredStreamingScalaExample-1.0.jar <brokers> <subscribe-type> <topic> <checkpointDir>*

  其中配置示例如下：

  如果报没有权限读写本地目录的错误，需要指定“spark.sql.streaming.checkpointLocation”参数，且用户必须具有该参数指定的目录的读、写权限。

- 运行**Python**样例代码：

  说明

  运行**Python**样例代码时需要将打包后的Java项目的jar包添加到streamingClient010/目录下。

  bin/spark-submit --master yarn --deploy-mode client --jars ***$(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")*** */opt/female/SparkStructuredStreamingPythonExample/KafkaWordCount.py <brokers> <subscribe-type> <topic> <checkpointDir>*

28.5.8.2 Spark Structured Streaming 样例程序（Java）

功能介绍

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

代码样例

下面代码片段仅为演示，具体代码参见：
com.huawei.bigdata.spark.examples.KafkaWordCount。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。

```
public class KafkaWordCount
{
  public static void main(String[] args) throws Exception {
    if (args.length < 3) {
      System.err.println("Usage: KafkaWordCount <bootstrap-servers> " +
        "<subscribe-type> <topics>");
      System.exit(1);
    }

    String bootstrapServers = args[0];
    String subscribeType = args[1];
    String topics = args[2];

    SparkSession spark = SparkSession
      .builder()
      .appName("KafkaWordCount")
      .getOrCreate();


    //创建表示来自kafka的输入行流的DataSet。
    Dataset<String> lines = spark
      .readStream()
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option(subscribeType, topics)
      .load()
      .selectExpr("CAST(value AS STRING)")
      .as(Encoders.STRING());

    //生成运行字数。
    Dataset<Row> wordCounts = lines.flatMap(new FlatMapFunction<String, String>() {
      @Override
      public Iterator<String> call(String x) {
        return Arrays.asList(x.split(" ")).iterator();
      }
    }, Encoders.STRING()).groupBy("value").count();

    //开始运行将运行计数打印到控制台的查询。
    StreamingQuery query = wordCounts.writeStream()
      .outputMode("complete")
      .format("console")
      .start();

    query.awaitTermination();
  }
}
```

28.5.8.3 Spark Structured Streaming 样例程序（Scala）

功能介绍

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

代码样例

下面代码片段仅为演示，具体代码参见：
com.huawei.bigdata.spark.examples.KafkaWordCount。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。

```
object KafkaWordCount {
  def main(args: Array[String]): Unit = {
    if (args.length < 3) {
      System.err.println("Usage: KafkaWordCount <bootstrap-servers> " +
        "<subscribe-type> <topics>")
      System.exit(1)
    }

    val Array(bootstrapServers, subscribeType, topics) = args

    val spark = SparkSession
      .builder
      .appName("KafkaWordCount")
      .getOrCreate()

    import spark.implicits._

    //创建表示来自kafka的输入行流的DataSet。
    val lines = spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option(subscribeType, topics)
      .load()
      .selectExpr("CAST(value AS STRING)")
      .as[String]

    //生成运行字数。
    val wordCounts = lines.flatMap(_.split(" ")).groupBy("value").count()

    //开始运行将运行计数打印到控制台的查询。
    val query = wordCounts.writeStream
      .outputMode("complete")
      .format("console")
      .start()

    query.awaitTermination()
  }
}
```

28.5.8.4 Spark Structured Streaming 样例程序（Python）

功能介绍

在Spark应用中，通过使用StructuredStreaming调用Kafka接口来获取单词记录，然后把单词记录分类统计，得到每个单词记录数。

代码样例

下面代码片段仅为演示，具体代码参见：SecurityKafkaWordCount。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
```



```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Usage: <bootstrapServers> <subscribeType> <topics>")
        exit(-1)

    bootstrapServers = sys.argv[1]
    subscribeType = sys.argv[2]
    topics = sys.argv[3]

    # 初始化sparkSession
    spark = SparkSession.builder.appName("KafkaWordCount").getOrCreate()

    # 创建表示来自kafka的input lines stream的DataFrame
    # 安全模式要修改spark/conf/jaas.conf和jaas-zk.conf为KafkaClient
    lines = spark.readStream.format("kafka")\
    .option("kafka.bootstrap.servers", bootstrapServers)\
    .option(subscribeType, topics)\
    .load()\
    .selectExpr("CAST(value AS STRING)")


    # 将lines切分为word
    words = lines.select(explode(split(lines.value, " ")).alias("word"))
    # 生成正在运行的word count
    wordCounts = words.groupBy("word").count()

    # 开始运行将running counts打印到控制台的查询
    query = wordCounts.writeStream\
    .outputMode("complete")\
    .format("console")\
    .start()

    query.awaitTermination()
```

28.5.9 Spark Structured Streaming 对接 Kafka 样例程序

28.5.9.1 Spark Structured Streaming 对接 Kafka 样例程序开发思路

场景说明

假定一个广告业务，存在广告请求事件、广告展示事件、广告点击事件，广告主需要实时统计有效的广告展示和广告点击数据。

已知：

1. 终端用户每次请求一个广告后，会生成广告请求事件，保存到kafka的adRequest topic中。
2. 请求一个广告后，可能用于多次展示，每次展示，会生成广告展示事件，保存到kafka的adShow topic中。
3. 每个广告展示，可能会产生多次点击，每次点击，会生成广告点击事件，保存到kafka的adClick topic中。
4. 广告有效展示的定义如下：
   a. 请求到展示的时长超过A分钟算无效展示。
   b. A分钟内多次展示，每次展示事件为有效展示。
5. 广告有效点击的定义如下：

a. 展示到点击时长超过B分钟算无效点击。
b. B分钟内多次点击，仅首次点击事件为有效点击。

基于此业务场景，模拟简单的数据结构如下：

- 广告请求事件
  数据结构：adID^reqTime
- 广告展示事件
  数据结构：adID^showID^showTime
- 广告点击事件
  数据结构：adID^showID^clickTime

数据关联关系如下：

- 广告请求事件与广告展示事件通过adID关联。
- 广告展示事件与广告点击事件通过adID+showID关联。

数据要求：

- 数据从产生到到达流处理引擎的延迟时间不超过2小时
- 广告请求事件、广告展示事件、广告点击事件到达流处理引擎的时间不能保证有序和时间对齐

数据规划

1. 在kafka中生成模拟数据（需要有Kafka权限用户）。

   java -cp $SPARK_HOME/conf:$SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:{ClassPath} com.huawei.bigdata.spark.examples.KafkaADEventProducer {BrokerList} {timeOfProduceReqEvent} {eventTimeBeforeCurrentTime} {reqTopic} {reqEventCount} {showTopic} {showEventMaxDelay} {clickTopic} {clickEventMaxDelay}

   说明

   - 确保集群安装完成，包括HDFS、Yarn、Spark2x和Kafka。
   - 启动Kafka的Producer，向Kafka发送数据。
   - {ClassPath}表示工程jar包的存放路径，详细路径由用户指定，可参考**在Linux环境中编包并运行Spark程序**章节中导出jar包的操作步骤。

   命令举例：

   java -cp /opt/client/Spark2x/spark/conf:/opt/StructuredStreamingADScalaExample-1.0.jar:/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/jars/streamingClient010/* com.huawei.bigdata.spark.examples.KafkaADEventProducer 10.132.190.170:21005,10.132.190.165:21005 2h 1h req 10000000 show 5m click 5m

   此命令将在kafka上创建3个topic：req、show、click，在2h内生成1千万条请求事件数据，请求事件的时间取值范围为{当前时间-1h 至 当前时间}，并为每条请求事件随机生成0-5条展示事件，展示事件的时间取值范围为{请求事件时间 至请求事件时间+5m }，为每条展示事件随机生成0-5条点击事件，点击事件的时间取值范围为{展示事件时间 至展示事件时间+5m }

开发思路

1. 使用Structured Streaming接收Kafka中数据，生成请求流、展示流、点击流。
2. 对请求流、展示流、点击流的数据进行关联查询。
3. 统计结果写入kafka。
4. 应用中监控流处理任务的状态。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“ /opt” ）下。
- 将user.keytab、krb5.conf 两个文件上传客户端所在服务器上

运行任务

在运行样例程序时需要指定 <kafkaBootstrapServers> <maxEventDelay> <reqTopic> <showTopic> <maxShowDelay> <clickTopic> <maxClickDelay> <triggerInterver> <checkpointDir>。

- <kafkaBootstrapServers>指获取元数据的Kafka地址。
- <maxEventDelay>指数据从生成到被流处理引擎的最大延迟时间。
- <reqTopic>指请求事件的topic名称。
- <showTopic>指展示事件的topic名称。
- <maxShowDelay>指有效展示事件的最大延迟时间。
- <clickTopic>指点击事件的topic名称。
- <maxClickDelay>指有效点击事件的最大延迟时间。
- <triggerInterver>指流处理任务的触发间隔。
- <checkpointDir>指checkpoint文件存放路径，本地或者HDFS路径下。

说明

由于Spark Structured Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Structured Streaming Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

注意

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，当前版本用户除了这一步外还需要将**$SPARK_HOME/jars/streamingClient010**目录中的kafka-clients jar包复制到**$SPARK_HOME/jars**目录下，否则会报class not found异常。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

bin/spark-submit --master yarn --deploy-mode client --jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}") --conf "spark.sql.streaming.statefulOperator.checkCorrectness.enabled=false" --class *com.huawei.bigdata.spark.examples.KafkaADCount /opt/StructuredStreamingADScalaExample-1.0.jar <kafkaBootstrapServers> <maxEventDelay> <reqTopic> <showTopic> <maxShowDelay> <clickTopic> <maxClickDelay> <triggerInterver> <checkpointDir>*

28.5.9.2 Spark Structured Streaming 对接 Kafka 样例程序（Scala）

功能介绍

使用Structured Streaming，从kafka中读取广告请求数据、广告展示数据、广告点击数据，实时获取广告有效展示统计数据和广告有效点击统计数据，将统计结果写入kafka中。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.KafkaADCount。

```
/**
 * 运行Structured Streaming任务，统计广告的有效展示和有效点击数据，结果写入kafka中
 */

object KafkaADCount {
  def main(args: Array[String]): Unit = {
    if (args.length < 12) {
      System.err.println("Usage: KafkaWordCount <bootstrap-servers> " +
        "<maxEventDelay> <reqTopic> <showTopic> <maxShowDelay> " +
        "<clickTopic> <maxClickDelay> <triggerInterver> " +
        "<checkpointLocation> <protocol> <service> <domain>")
      System.exit(1)
    }

    val Array(bootstrapServers, maxEventDelay, reqTopic, showTopic,
    maxShowDelay, clickTopic, maxClickDelay, triggerInterver, checkpointLocation,
    protocol, service, domain) = args

    val maxEventDelayMills = JavaUtils.timeStringAs(maxEventDelay, TimeUnit.MILLISECONDS)
    val maxShowDelayMills = JavaUtils.timeStringAs(maxShowDelay, TimeUnit.MILLISECONDS)
    val maxClickDelayMills = JavaUtils.timeStringAs(maxClickDelay, TimeUnit.MILLISECONDS)
    val triggerMills = JavaUtils.timeStringAs(triggerInterver, TimeUnit.MILLISECONDS)

    val spark = SparkSession
      .builder
      .appName("KafkaADCount")
      .getOrCreate()

    spark.conf.set("spark.sql.streaming.checkpointLocation", checkpointLocation)

    import spark.implicits._

    // Create DataSet representing the stream of input lines from kafka
    val reqDf = spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", bootstrapServers)
      .option("kafka.security.protocol", protocol)
      .option("kafka.sasl.kerberos.service.name", service)
      .option("kafka.kerberos.domain.name", domain)
      .option("subscribe", reqTopic)
      .load()
```



```
  .selectExpr("CAST(value AS STRING)")
  .as[String]
  .map{
    _.split('^') match {
      case Array(reqAdID, reqTime) => ReqEvent(reqAdID,
        Timestamp.valueOf(reqTime))
    }
  }
  .as[ReqEvent]
  .withWatermark("reqTime", maxEventDelayMills +
    maxShowDelayMills + " millisecond")

val showDf = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("kafka.security.protocol", protocol)
  .option("kafka.sasl.kerberos.service.name", service)
  .option("kafka.kerberos.domain.name", domain)
  .option("subscribe", showTopic)
  .load()
  .selectExpr("CAST(value AS STRING)")
  .as[String]
  .map{
    _.split('^') match {
      case Array(showAdID, showID, showTime) => ShowEvent(showAdID,
        showID, Timestamp.valueOf(showTime))
    }
  }
  .as[ShowEvent]
  .withWatermark("showTime", maxEventDelayMills +
    maxShowDelayMills + maxClickDelayMills + " millisecond")

val clickDf = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("kafka.security.protocol", protocol)
  .option("kafka.sasl.kerberos.service.name", service)
  .option("kafka.kerberos.domain.name", domain)
  .option("subscribe", clickTopic)
  .load()
  .selectExpr("CAST(value AS STRING)")
  .as[String]
  .map{
    _.split('^') match {
      case Array(clickAdID, clickShowID, clickTime) => ClickEvent(clickAdID,
        clickShowID, Timestamp.valueOf(clickTime))
    }
  }
  .as[ClickEvent]
  .withWatermark("clickTime", maxEventDelayMills + " millisecond")

val showStaticsQuery = reqDf.join(showDf,
  expr(s"""
  reqAdID = showAdID
  AND showTime >= reqTime + interval ${maxShowDelayMills} millisecond
  """))
  .selectExpr("concat_ws('^', showAdID, showID, showTime) as value")
  .writeStream
  .queryName("showEventStatics")
  .outputMode("append")
  .trigger(Trigger.ProcessingTime(triggerMills.millis))
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("kafka.security.protocol", protocol)
  .option("kafka.sasl.kerberos.service.name", service)
  .option("kafka.kerberos.domain.name", domain)
  .option("topic", "showEventStatics")
```

```
    .start()

  val clickStaticsQuery = showDf.join(clickDf,
    expr(s"""
    showAdID = clickAdID AND
    showID = clickShowID AND
    clickTime >= showTime + interval ${maxClickDelayMills} millisecond
    """), joinType = "rightouter")
    .dropDuplicates("showAdID")
    .selectExpr("concat_ws('^', clickAdID, clickShowID, clickTime) as value")
    .writeStream
    .queryName("clickEventStatics")
    .outputMode("append")
    .trigger(Trigger.ProcessingTime(triggerMills.millis))
    .format("kafka")
    .option("kafka.bootstrap.servers", bootstrapServers)
    .option("kafka.security.protocol", protocol)
    .option("kafka.sasl.kerberos.service.name", service)
    .option("kafka.kerberos.domain.name", domain)
    .option("topic", "clickEventStatics")
    .start()

  new Thread(new Runnable {
    override def run(): Unit = {
      while(true) {
        println("-------------get showStatic progress---------")
        //println(showStaticsQuery.lastProgress)
        println(showStaticsQuery.status)
        println("-------------get clickStatic progress---------")
        //println(clickStaticsQuery.lastProgress)
        println(clickStaticsQuery.status)
        Thread.sleep(10000)
      }
    }
  }).start

  spark.streams.awaitAnyTermination()

 }
}
```

28.5.10 Spark Structured Streaming 状态操作样例程序

28.5.10.1 Spark Structured Streaming 状态操作样例程序开发思路

场景说明

假设需要跨批次统计每个session期间发生了多少次event以及本session的开始和结束timestamp；

同时输出本批次被更新状态的session。

数据规划

1. 在kafka中生成模拟数据（需要有Kafka权限用户）
2. 确保集群安装完成，包括HDFS、Yarn、Spark2x和Kafka。
3. 创建Topic。

   {zkQuorum}表示ZooKeeper集群信息，格式为IP:port。

   $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper {zkQuorum}/kafka --replication-factor 1 --partitions 1 --topic {Topic}

4. 启动Kafka的Producer，向Kafka发送数据。

   {ClassPath}表示工程jar包的存放路径，详细路径由用户指定，可参考**在Linux环境中编包并运行Spark程序**章节中导出jar包的操作步骤。

   java -cp $SPARK_HOME/conf:$SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:{ClassPath} com.huawei.bigdata.spark.examples.KafkaProducer {brokerlist} {topic} {number of events produce every 0.02s}

   示例：

   java -cp /opt/client/Spark2x/spark/conf:/opt/StructuredStreamingState-1.0.jar:/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/jars/streamingClient010/* com.huawei.bigdata.spark.examples.KafkaProducer xxx.xxx.xxx.xxx:21005,xxx.xxx.xxx.xxx:21005,xxx.xxx.xxx.xxx:21005 mytopic 10

开发思路

1. 接收Kafka中数据，生成相应DataStreamReader。
2. 进行分类统计。
3. 计算结果，并进行打印。

打包项目

- 通过IDEA自带的Maven工具，打包项目，生成jar包
- 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt”）下。

运行任务

在运行样例程序时需要指定 <brokers> <subscribe-type> <topic> <checkpointLocation>。

- <brokers>指获取元数据的Kafka地址。
- <subscribe-type> 指定kakfa的消费方式。
- <topic>指要消费的kafka topic。
- <checkpointLocation> 指spark任务的checkpoint保存HDFS路径下。

说明

由于Spark Structured Streaming Kafka的依赖包在客户端的存放路径与其他依赖包不同，如其他依赖包路径为“$SPARK_HOME/jars”，而Spark Streaming Structured Kafka依赖包路径为“$SPARK_HOME/jars/streamingClient010”。所以在运行应用程序时，需要在spark-submit命令中添加配置项，指定Spark Streaming Kafka的依赖包路径，如**--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")**

进入Spark客户端目录，调用bin/spark-submit脚本运行代码，运行命令如下（类名与文件名等请与实际代码保持一致，此处仅为示例）：

- **bin/spark-submit --master yarn --deploy-mode client** ***--jars $(files=($SPARK_HOME/jars/streamingClient010/*.jar); IFS=,; echo "${files[*]}")*** **--class** *com.huawei.bigdata.spark.examples.kafkaSessionization /opt/StructuredStreamingState-1.0.jar <brokers> <subscribe-type>* <topic> <checkpointLocation>

注意

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，当前版本用户除了这一步外还需要将**$SPARK_HOME/jars/streamingClient010**目录中的kafka-clients jar包复制到**$SPARK_HOME/jars**目录下，否则会报class not found异常。

28.5.10.2 Spark Structured Streaming 状态操作样例程序（Scala）

功能介绍

在Spark结构流应用中，跨批次统计每个session期间发生了多少次event以及本session的开始和结束timestamp；同时输出本批次被更新状态的session。

代码样例

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.spark.examples.kafkaSessionization。

说明

当Streaming DataFrame/Dataset中有新的可用数据时，outputMode用于配置写入Streaming接收器的数据。

```
object kafkaSessionization {
  def main(args: Array[String]): Unit = {
    if (args.length < 7) {
      System.err.println("Usage: kafkaSessionization <bootstrap-servers> " +
        "<subscribe-type> <protocol> <service> <domain> <topics> <checkpointLocation>")
      System.exit(1)
    }

    val Array(bootstrapServers, subscribeType, protocol, service, domain,topics，checkpointLocation) = args

    val spark = SparkSession
      .builder
      .appName("kafkaSessionization")
      .getOrCreate()

    spark.conf.set("spark.sql.streaming.checkpointLocation", checkpointLocation)

    spark.streams.addListener(new StreamingQueryListener {

      @volatile private var startTime: Long = 0L
      @volatile private var endTime: Long = 0L
      @volatile private var numRecs: Long = 0L

      override def onQueryStarted(event: StreamingQueryListener.QueryStartedEvent): Unit = {
        println("Query started: " + event.id)
        startTime = System.currentTimeMillis
      }

      override def onQueryProgress(event: StreamingQueryListener.QueryProgressEvent): Unit = {
        println("Query made progress: " + event.progress)
        numRecs += event.progress.numInputRows
      }

      override def onQueryTerminated(event: StreamingQueryListener.QueryTerminatedEvent): Unit = {
        println("Query terminated: " + event.id)
        endTime = System.currentTimeMillis
      }
    })
```

```
import spark.implicits._

val df = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", bootstrapServers)
  .option("kafka.security.protocol", protocol)
  .option("kafka.sasl.kerberos.service.name", service)
  .option("kafka.kerberos.domain.name", domain)
  .option(subscribeType, topics)
  .load()
  .selectExpr("CAST(value AS STRING)")
  .as[String]
  .map { x =>
    val splitStr = x.split(",")
    (splitStr(0), Timestamp.valueOf(splitStr(1)))
  }.as[(String, Timestamp)].flatMap { case(line, timestamp) =>
  line.split(" ").map(word => Event(sessionId = word, timestamp))}

// Sessionize the events. Track number of events, start and end timestamps of session, and
// and report session updates.
val sessionUpdates = df
  .groupByKey(event => event.sessionId)
  .mapGroupsWithState[SessionInfo, SessionUpdate](GroupStateTimeout.ProcessingTimeTimeout) {

  case (sessionId: String, events: Iterator[Event], state: GroupState[SessionInfo]) =>

    // If timed out, then remove session and send final update
    if (state.hasTimedOut) {
      val finalUpdate =
        SessionUpdate(sessionId, state.get.durationMs, state.get.numEvents, expired = true)
      state.remove()
      finalUpdate
    } else {
      // Update start and end timestamps in session
      val timestamps = events.map(_.timestamp.getTime).toSeq
      val updatedSession = if (state.exists) {
        val oldSession = state.get
        SessionInfo(
          oldSession.numEvents + timestamps.size,
          oldSession.startTimestampMs,
          math.max(oldSession.endTimestampMs, timestamps.max))
      } else {
        SessionInfo(timestamps.size, timestamps.min, timestamps.max)
      }
      state.update(updatedSession)

      // Set timeout such that the session will be expired if no data received for 10 seconds
      state.setTimeoutDuration("10 seconds")
      SessionUpdate(sessionId, state.get.durationMs, state.get.numEvents, expired = false)
    }
}
// Start running the query that prints the session updates to the console
val query = sessionUpdates
  .writeStream
  .outputMode("update")
  .format("console")
  .start()

query.awaitTermination()
}
```

28.5.11 Spark 同步 HBase 数据到 CarbonData 样例程序

28.5.11.1 Spark 同步 HBase 数据到 CarbonData 开发思路

场景说明

数据实时写入HBase，用于点查业务，数据每隔一段时间批量同步到CarbonData表中，用于分析型查询业务。

数据规划

说明

运行样例程序前，需要在Spark客户端的“spark-defaults.conf”配置文件中将配置项“spark.yarn.security.credentials.hbase.enabled”设置为“true”（该参数值默认为“false”，改为“true”后对已有业务没有影响。如果要卸载HBase服务，卸载前请将此参数值改回“false”）。

1. 创建HBase表，构造数据，列需要包含key，modify_time，valid。其中每条数据key值全表唯一，modify_time代表修改时间，valid代表是否为有效数据（该样例中'1'为有效，'0'为无效数据）。

   示例：进入hbase shell，执行如下命令：

   create 'hbase_table','key','info'

   put 'hbase_table','1','info:modify_time','*2019-11-22 23:28:39*'

   put 'hbase_table','1','info:valid','1'

   put 'hbase_table','2','info:modify_time','*2019-11-22 23:28:39*'

   put 'hbase_table','2','info:valid','1'

   put 'hbase_table','3','info:modify_time','*2019-11-22 23:28:39*'

   put 'hbase_table','3','info:valid','0'

   put 'hbase_table','4','info:modify_time','*2019-11-22 23:28:39*'

   put 'hbase_table','4','info:valid','1'

   说明

   上述数据的modify_time列可设置为当前时间之前的值。

   put 'hbase_table','5','info:modify_time','*2021-03-03 15:20:39*'

   put 'hbase_table','5','info:valid','1'

   put 'hbase_table','6','info:modify_time','*2021-03-03 15:20:39*'

   put 'hbase_table','6','info:valid','1'

   put 'hbase_table','7','info:modify_time','*2021-03-03 15:20:39*'

   put 'hbase_table','7','info:valid','0'

   put 'hbase_table','8','info:modify_time','*2021-03-03 15:20:39*'

   put 'hbase_table','8','info:valid','1'

   put 'hbase_table','4','info:valid','0'

   put 'hbase_table','4','info:modify_time','*2021-03-03 15:20:39*'

   说明

   上述数据的modify_time列可设置为样例程序启动后30分钟内的时间值（此处的30分钟为样例程序默认的同步间隔时间，可修改）。

   put 'hbase_table','9','info:modify_time','*2021-03-03 15:32:39*'

put 'hbase_table','9','info:valid','1'

put 'hbase_table','10','info:modify_time','*2021-03-03 15:32:39***'**

put 'hbase_table','10','info:valid','1'

put 'hbase_table','11','info:modify_time','*2021-03-03 15:32:39***'**

put 'hbase_table','11','info:valid','0'

put 'hbase_table','12','info:modify_time','*2021-03-03 15:32:39***'**

put 'hbase_table','12','info:valid','1'

说明

上述数据的modify_time列可设置为样例程序启动后30分钟到60分钟内的时间值，即第二次同步周期。

2. 在sparksql中创建HBase的hive外表，命令如下：

   create table *external_hbase_table(key string ,modify_time STRING, valid STRING)*

   using org.apache.spark.sql.hbase.HBaseSource

   options*(hbaseTableName "hbase_table", keyCols "key", colsMapping "modify_time=info.modify_time,valid=info.valid")*;

3. 在sparksql中创建CarbonData表：

   create table *carbon01(key string,modify_time STRING, valid STRING)* stored as carbondata;

4. 初始化加载当前hbase表中所有数据到CarbonData表；

   insert into table *carbon01* select * **from** *external_hbase_table* where *valid='1'*;

5. 用spark-submit提交命令：

```
spark-submit --master yarn --deploy-mode client --class
com.huawei.bigdata.spark.examples.HBaseExternalHivetoCarbon /opt/example/
HBaseExternalHivetoCarbon-1.0.jar
```

28.5.11.2 Spark 同步 HBase 数据到 CarbonData（Java）

下面代码片段仅为演示，具体代码参见：com.huawei.spark.examples.HBaseExternalHivetoCarbon。

```
public static void main(String[] args) throws Exception {
spark = SparkSession.builder().appName("HBaseExternalHiveToCarbon").getOrCreate();

Timer timer = new Timer();
timer.schedule(new TimerTask() {
public void run() {
timeEnd = timeStart + TIMEWINDOW;

queryTimeStart = transferDateToStr(timeStart);
queryTimeEnd = transferDateToStr(timeEnd);

//run delete logic
cmdsb = new StringBuilder();
cmdsb.append("delete from ")
.append(carbonTableName)
.append("  where key in (select key from ")
.append(externalHiveTableName)
.append(" where modify_time>'")
.append(queryTimeStart)
.append("' and modify_time<'")
.append(queryTimeEnd)
.append("' and valid='0')");
spark.sql(cmdsb.toString());
```

```
//run insert logic
cmdsb = new StringBuilder();
cmdsb.append("insert into ")
.append(carbonTableName)
.append("  select * from ")
.append(externalHiveTableName)
.append(" where modify_time>'")
.append(queryTimeStart)
.append("'  and modify_time<'")
.append(queryTimeEnd)
.append("'  and valid='1'");
spark.sql(cmdsb.toString());

timeStart = timeEnd;
}
}, TIMEWINDOW, TIMEWINDOW);
}
```

28.5.12 使用 Spark 执行 Hudi 样例程序

28.5.12.1 使用 Spark 执行 Hudi 样例程序开发思路

场景说明

本章节介绍如何使用Spark操作Hudi执行插入数据、查询数据、更新数据、增量查询、特定时间点查询、删除数据等操作。

详细代码请参考样例代码。

打包项目

1. 通过IDEA自带的Maven工具，打包项目，生成jar包。具体操作请参考**在Linux环境中编包并运行Spark程序**。

   说明

   运行Python样例代码无需通过Maven打包。

2. 将打包生成的jar包上传到Spark客户端所在服务器的任意目录（例如“/opt/example/”）下。

运行任务

1. 登录Spark客户端节点，执行如下命令：

   source *客户端安装目录*/**bigdata_env**

   source *客户端安装目录*/**Hudi/component_env**

2. 编译构建样例代码后可以使用spark-submit提交命令，执行命令后会依次执行写入、更新、查询、删除等操作：
   - 运行Java样例程序：

     spark-submit --class com.huawei.bigdata.hudi.examples.HoodieWriteClientExample /opt/example/hudi-java-examples-1.0.jar hdfs://hacluster/tmp/example/hoodie_java hoodie_java

     其中：“/opt/example/hudi-java-examples-1.0.jar”为jar包路径，“hdfs://hacluster/tmp/example/hoodie_java”为Hudi表的存储路径，“hoodie_java”为Hudi表的表名。

- 运行Scala样例程序：

  spark-submit --class com.huawei.bigdata.hudi.examples.HoodieDataSourceExample /opt/example/hudi-scala-examples-1.0.jar hdfs://hacluster/tmp/example/hoodie_scala hoodie_scala

  其中：“/opt/example/hudi-scala-examples-1.0.jar”为jar包路径，“hdfs://hacluster/tmp/example/hoodie_scala”为Hudi表的存储路径，“ hoodie_Scala”为Hudi表的表名。

- 运行Python样例程序：

  spark-submit /opt/example/HudiPythonExample.py hdfs://hacluster/tmp/huditest/example/python hudi_trips_cow

  其中：“hdfs://hacluster/tmp/huditest/example/python”为Hudi表的存储路径，“ hudi_trips_cow”为Hudi表的表名。

28.5.12.2 使用 Spark 执行 Hudi 样例程序（Java）

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.hudi.examples.HoodieWriteClientExample。

创建客户端对象来操作Hudi：

```
String tablePath = args[0];
String tableName = args[1];
SparkConf sparkConf = HoodieExampleSparkUtils.defaultSparkConf("hoodie-client-example");
JavaSparkContext jsc = new JavaSparkContext(sparkConf);
// Generator of some records to be loaded in.
HoodieExampleDataGenerator<HoodieAvroPayload> dataGen = new HoodieExampleDataGenerator<>();
// initialize the table, if not done already
Path path = new Path(tablePath);
FileSystem fs = FSUtils.getFs(tablePath, jsc.hadoopConfiguration());
if (!fs.exists(path)) {
HoodieTableMetaClient.initTableType(jsc.hadoopConfiguration(), tablePath,
HoodieTableType.valueOf(tableType),
tableName, HoodieAvroPayload.class.getName());
}

// Create the write client to write some records in
HoodieWriteConfig cfg = HoodieWriteConfig.newBuilder().withPath(tablePath)
    .withSchema(HoodieExampleDataGenerator.TRIP_EXAMPLE_SCHEMA).withParallelism(2, 2)
    .withDeleteParallelism(2).forTable(tableName)
    .withIndexConfig(HoodieIndexConfig.newBuilder().withIndexType(HoodieIndex.IndexType.BLOOM).build()
)
    .withCompactionConfig(HoodieCompactionConfig.newBuilder().archiveCommitsWith(20,
30).build()).build();
SparkRDDWriteClient<HoodieAvroPayload> client = new SparkRDDWriteClient<>(new
HoodieSparkEngineContext(jsc), cfg);
```

插入数据：

```
String newCommitTime = client.startCommit();
LOG.info("Starting commit " + newCommitTime);
List<HoodieRecord<HoodieAvroPayload>> records = dataGen.generateInserts(newCommitTime, 10);
List<HoodieRecord<HoodieAvroPayload>> recordsSoFar = new ArrayList<>(records);
JavaRDD<HoodieRecord<HoodieAvroPayload>> writeRecords = jsc.parallelize(records, 1);
client.upsert(writeRecords, newCommitTime);
```

更新数据：

```
newCommitTime = client.startCommit();
LOG.info("Starting commit " + newCommitTime);
List<HoodieRecord<HoodieAvroPayload>> toBeUpdated = dataGen.generateUpdates(newCommitTime, 2);
records.addAll(toBeUpdated);
```



```
recordsSoFar.addAll(toBeUpdated);
writeRecords = jsc.parallelize(records, 1);
client.upsert(writeRecords, newCommitTime);
```

删除数据：

```
newCommitTime = client.startCommit();
LOG.info("Starting commit " + newCommitTime);
// just delete half of the records
int numToDelete = recordsSoFar.size() / 2;
List<HoodieKey> toBeDeleted =
recordsSoFar.stream().map(HoodieRecord::getKey).limit(numToDelete).collect(Collectors.toList());
JavaRDD<HoodieKey> deleteRecords = jsc.parallelize(toBeDeleted, 1);
client.delete(deleteRecords, newCommitTime);
```

压缩数据：

```
if (HoodieTableType.valueOf(tableType) == HoodieTableType.MERGE_ON_READ) {
   Option<String> instant = client.scheduleCompaction(Option.empty());
   JavaRDD<WriteStatus> writeStatues = client.compact(instant.get());
   client.commitCompaction(instant.get(), writeStatues, Option.empty());
}
```

28.5.12.3 使用 Spark 执行 Hudi 样例程序（Scala）

下面代码片段仅为演示，具体代码参见：com.huawei.bigdata.hudi.examples.HoodieDataSourceExample。

插入数据：

```
def insertData(spark: SparkSession, tablePath: String, tableName: String, dataGen:
HoodieExampleDataGenerator[HoodieAvroPayload]): Unit = {
val commitTime: String = System.currentTimeMillis().toString
val inserts = dataGen.convertToStringList(dataGen.generateInserts(commitTime, 20))
spark.sparkContext.parallelize(inserts, 2)
val df = spark.read.json(spark.sparkContext.parallelize(inserts, 1))df.write.format("org.apache.hudi").
     options(getQuickstartWriteConfigs).
     option(PRECOMBINE_FIELD_OPT_KEY, "ts").
     option(RECORDKEY_FIELD_OPT_KEY, "uuid").
     option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
     option(TABLE_NAME, tableName).
     mode(Overwrite).
     save(tablePath)}
```

查询数据：

```
def queryData(spark: SparkSession, tablePath: String, tableName: String, dataGen:
HoodieExampleDataGenerator[HoodieAvroPayload]): Unit = {
val roViewDF = spark.
     read.
     format("org.apache.hudi").
     load(tablePath + "/*/*/*/*")
roViewDF.createOrReplaceTempView("hudi_ro_table")
spark.sql("select fare, begin_lon, begin_lat, ts from  hudi_ro_table where fare > 20.0").show()
//  +-----------------+-------------------+-------------------+---+
//  |             fare|          begin_lon|          begin_lat| ts|
//  +-----------------+-------------------+-------------------+---+
//  |98.88075495133515|0.3955604862303160 3|0.17851135255091155|0.0|
//  ...
spark.sql("select _hoodie_commit_time, _hoodie_record_key, _hoodie_partition_path, rider, driver, fare from
hudi_ro_table").show()
//  +-------------------+--------------------+----------------------+-------------------+--------------------
+-------------------+
//  |_hoodie_commit_time|  _hoodie_record_key|_hoodie_partition_path|              rider|
driver|               fare|
//  +-------------------+--------------------+----------------------+-------------------+--------------------
+-------------------+
//  |     20191231181501|31cafb9f-0196-4b1...|            2020/01/02|rider-1577787297889|
driver-1577787297889| 98.88075495133515|
```

```
// ...
}
```

更新数据：

```
def updateData(spark: SparkSession, tablePath: String, tableName: String, dataGen:
HoodieExampleDataGenerator[HoodieAvroPayload]): Unit = {
val commitTime: String = System.currentTimeMillis().toString
val updates = dataGen.convertToStringList(dataGen.generateUpdates(commitTime, 10))
val df = spark.read.json(spark.sparkContext.parallelize(updates, 1))
df.write.format("org.apache.hudi").
    options(getQuickstartWriteConfigs).
    option(PRECOMBINE_FIELD_OPT_KEY, "ts").
    option(RECORDKEY_FIELD_OPT_KEY, "uuid").
    option(PARTITIONPATH_FIELD_OPT_KEY, "partitionpath").
    option(TABLE_NAME, tableName).
    mode(Append).
    save(tablePath)}
```

增量查询：

```
def incrementalQuery(spark: SparkSession, tablePath: String, tableName: String) {
import spark.implicits._
val commits = spark.sql("select distinct(_hoodie_commit_time) as commitTime from hudi_ro_table order by
commitTime").map(k => k.getString(0)).take(50)
val beginTime = commits(commits.length - 2)

val incViewDF = spark.
    read.
    format("org.apache.hudi").
    option(QUERY_TYPE_OPT_KEY, QUERY_TYPE_INCREMENTAL_OPT_VAL).
    option(BEGIN_INSTANTTIME_OPT_KEY, beginTime).
    load(tablePath)
incViewDF.createOrReplaceTempView("hudi_incr_table")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from hudi_incr_table where fare >
20.0").show()}
```

特定时间点查询：

```
def pointInTimeQuery(spark: SparkSession, tablePath: String, tableName: String) {
import spark.implicits._
val commits = spark.sql("select distinct(_hoodie_commit_time) as commitTime from  hudi_ro_table order by
commitTime").map(k => k.getString(0)).take(50)
val beginTime = "000"
// Represents all commits > this time.
val endTime = commits(commits.length - 2)
// commit time we are interested in
//incrementally query data
val incViewDF = spark.read.format("org.apache.hudi").
    option(QUERY_TYPE_OPT_KEY, QUERY_TYPE_INCREMENTAL_OPT_VAL).
    option(BEGIN_INSTANTTIME_OPT_KEY, beginTime).
    option(END_INSTANTTIME_OPT_KEY, endTime).
    load(tablePath)
incViewDF.createOrReplaceTempView("hudi_incr_table")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from  hudi_incr_table where fare >
20.0").show()}
```

28.5.12.4 使用 Spark 执行 Hudi 样例程序（Python）

使用 python 写 Hudi 表

下面代码片段仅为演示，具体代码参见：sparknormal-examples.SparkOnHudiPythonExample.hudi_python_write_example。

插入数据：

```
#insert
inserts = sc._jvm.org.apache.hudi.QuickstartUtils.convertToStringList(dataGen.generateInserts(10))
```

```
df = spark.read.json(spark.sparkContext.parallelize(inserts, 2))
hudi_options = {
'hoodie.table.name': tableName,
'hoodie.datasource.write.recordkey.field': 'uuid',
'hoodie.datasource.write.partitionpath.field': 'partitionpath',
'hoodie.datasource.write.table.name': tableName,
'hoodie.datasource.write.operation': 'insert',
'hoodie.datasource.write.precombine.field': 'ts',
'hoodie.upsert.shuffle.parallelism': 2,'
hoodie.insert.shuffle.parallelism': 2
}
df.write.format("hudi"). \
    options(**hudi_options). \
    mode("overwrite"). \
    save(basePath)
```

查询数据：

```
tripsSnapshotDF = spark. \
    read. \
    format("hudi"). \
    load(basePath + "/*/*/*/*")
tripsSnapshotDF.createOrReplaceTempView("hudi_trips_snapshot")
spark.sql("select fare, begin_lon, begin_lat, ts from  hudi_trips_snapshot where fare > 20.0").show()
spark.sql("select _hoodie_commit_time, _hoodie_record_key, _hoodie_partition_path, rider, driver, fare from
hudi_trips_snapshot").show()
```

更新数据：

```
updates = sc._jvm.org.apache.hudi.QuickstartUtils.convertToStringList(dataGen.generateUpdates(10))
df = spark.read.json(spark.sparkContext.parallelize(updates, 2))
df.write.format("hudi"). \
    options(**hudi_options). \
    mode("append"). \
    save(basePath)
```

增量查询：

```
spark. \
    read. \
    format("hudi"). \
    load(basePath + "/*/*/*/*"). \
    createOrReplaceTempView("hudi_trips_snapshot")
incremental_read_options = {
    'hoodie.datasource.query.type': 'incremental',
    'hoodie.datasource.read.begin.instanttime': beginTime,
}
tripsIncrementalDF = spark.read.format("hudi"). \
    options(**incremental_read_options). \
    load(basePath)
tripsIncrementalDF.createOrReplaceTempView("hudi_trips_incremental")
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from  hudi_trips_incremental where
fare > 20.0").show()
```

特定时间点查询：

```
# Represents all commits > this time.
beginTime = "000"
endTime = commits[len(commits) - 2]
point_in_time_read_options = {
    'hoodie.datasource.query.type': 'incremental',
    'hoodie.datasource.read.end.instanttime': endTime,
    'hoodie.datasource.read.begin.instanttime': beginTime
}

tripsPointInTimeDF = spark.read.format("hudi"). \
    options(**point_in_time_read_options). \
    load(basePath)

tripsPointInTimeDF.createOrReplaceTempView("hudi_trips_point_in_time")
```



```
spark.sql("select `_hoodie_commit_time`, fare, begin_lon, begin_lat, ts from hudi_trips_point_in_time where fare > 20.0").show()
```

删除数据：

```
# 获取记录总数
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").count()
# 拿到两条将被删除的记录
ds = spark.sql("select uuid, partitionpath from hudi_trips_snapshot").limit(2)
# 执行删除
hudi_delete_options = {
    'hoodie.table.name': tableName,
    'hoodie.datasource.write.recordkey.field': 'uuid',
    'hoodie.datasource.write.partitionpath.field': 'partitionpath',
    'hoodie.datasource.write.table.name': tableName,
    'hoodie.datasource.write.operation': 'delete',
    'hoodie.datasource.write.precombine.field': 'ts',
    'hoodie.upsert.shuffle.parallelism': 2,
    'hoodie.insert.shuffle.parallelism': 2
}
from pyspark.sql.functions import lit
deletes = list(map(lambda row: (row[0], row[1]), ds.collect()))
df = spark.sparkContext.parallelize(deletes).toDF(['uuid', 'partitionpath']).withColumn('ts', lit(0.0))
df.write.format("hudi"). \
    options(**hudi_delete_options). \
    mode("append"). \
    save(basePath)
# 像之前一样运行查询
roAfterDeleteViewDF = spark. \
    read. \
    format("hudi"). \
    load(basePath + "/*/*/*/*")
roAfterDeleteViewDF.registerTempTable("hudi_trips_snapshot")
# 应返回 (total - 2) 条记录
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").count()
spark.sql("select uuid, partitionpath from hudi_trips_snapshot").show()
```

28.5.13 Hudi 的自定义配置项样例程序

28.5.13.1 HoodieDeltaStreamer

编写自定义的转化类实现Transformer。

编写自定义的Schema实现SchemaProvider。

在执行HoodieDeltaStreamer时加入参数：

```
--schemaprovider-class 定义的schema类 --transformer-class 定义的transform类
```

Transformer和SchemaProvider样例：

```
public class TransformerExample implements Transformer, Serializable {
    @Override
    public Dataset<Row> apply(JavaSparkContext jsc, SparkSession sparkSession, Dataset<Row> rowDataset,
                              TypedProperties properties) {
        JavaRDD<Row> rowJavaRdd = rowDataset.toJavaRDD();
        List<Row> rowList =new ArrayList<>();
        for(Row row: rowJavaRdd.collect()){
            rowList.add(buildRow(row));
        }
        JavaRDD<Row> stringJavaRdd = jsc.parallelize(rowList);
        List<StructField> fields = new ArrayList<>();
        builFields(fields);
        StructType schema = DataTypes.createStructType(fields);
        Dataset<Row> dataFrame = sparkSession.createDataFrame(stringJavaRdd, schema);
        return dataFrame;
```



```
    }

    private void builFields(List<StructField> fields) {
        fields.add(DataTypes.createStructField("age", DataTypes.StringType, true));
        fields.add(DataTypes.createStructField("id", DataTypes.StringType, true));
        fields.add(DataTypes.createStructField("name", DataTypes.StringType, true));
        fields.add(DataTypes.createStructField("job", DataTypes.StringType, true));
    }

    private Row buildRow(Row row){
        String age = row.getString(0);
        String id = row.getString(1);
        String job = row.getString(2);
        String name = row.getString(3);
        Row returnRow = RowFactory.create(age, id, job, name);
        return returnRow;
    }
}
public class DataSchemaProviderExample extends SchemaProvider {

    public DataSchemaProviderExample(TypedProperties props, JavaSparkContext jssc) {
        super(props, jssc);
    }

    @Override
    public Schema getSourceSchema() {
        Schema avroSchema = new Schema.Parser().parse(
            "{\"type\":\"record\",\"name\":\"hoodie_source\",\"fields\":[{\"name\":\"age\",\"type\":\"string\"},
{\"name\":\"id\",\"type\":\"string\"},{\"name\":\"job\",\"type\":\"string\"},{\"name\":\"name\",\"type
\":\"string\"}]}");
        return avroSchema;
    }

    @Override
    public Schema getTargetSchema() {
        Schema avroSchema = new Schema.Parser().parse(
            "{\"type\":\"record\",\"name\":\"mytest_record\",\"namespace\":\"hoodie.mytest\",\"fields\":
[{\"name\":\"age\",\"type\":\"string\"},{\"name\":\"id\",\"type\":\"string\"},{\"name\":\"job\",\"type\":\"string
\"},{\"name\":\"name\",\"type\":\"string\"}]}");
        return avroSchema;
    }

}
```

28.5.13.2 自定义排序器

编写自定义排序类继承BulkInsertPartitioner，在写入Hudi时加入配置：

```
.option(BULKINSERT_USER_DEFINED_PARTITIONER_CLASS, <自定义排序类的包名加类名>)
```

自定义分区排序器样例：

```
public class HoodieSortExample<T extends HoodieRecordPayload>
    implements BulkInsertPartitioner<JavaRDD<HoodieRecord<T>>> {
    @Override
    public JavaRDD<HoodieRecord<T>> repartitionRecords(JavaRDD<HoodieRecord<T>> records, int
outputSparkPartitions) {
        JavaPairRDD<String,
            HoodieRecord<T>> stringHoodieRecordJavaPairRDD = records.coalesce(outputSparkPartitions)
                .mapToPair(record -> new Tuple2<>(new StringBuilder().append(record.getPartitionPath())
                    .append("+")
                    .append(record.getRecordKey())
                    .toString(), record));
        JavaRDD<HoodieRecord<T>> hoodieRecordJavaRDD =
stringHoodieRecordJavaPairRDD.mapPartitions(partition -> {
            List<Tuple2<String, HoodieRecord<T>>> recordList = new ArrayList<>();
            for (; partition.hasNext();) {
                recordList.add(partition.next());
```



```
            }
            Collections.sort(recordList, (o1, o2) -> {
                if (o1._1().split("[+]")[0] == o2._1().split("[+]")[0]) {
                    return Integer.parseInt(o1._1().split("[+]")[1]) - Integer.parseInt(o2._1().split("[+]")[1]);
                } else {
                    return o1._1().split("[+]")[0].compareTo(o2._1().split("[+]")[0]);
                }
            });
            return recordList.stream().map(e -> e._2).iterator();
        });
        return hoodieRecordJavaRDD;
    }

    @Override
    public boolean arePartitionRecordsSorted() {
        return true;
    }
}
```

28.6 调测 Spark 应用

28.6.1 在本地 Windows 环境中调测 Spark 应用

28.6.1.1 配置 Windows 通过 EIP 访问集群 Spark

操作场景

该章节通过指导用户配置集群绑定EIP，并配置Spark文件的方法，方便用户可以在本地对样例文件进行编译。

本章节以运行SparkScalaExample样例为例进行说明。

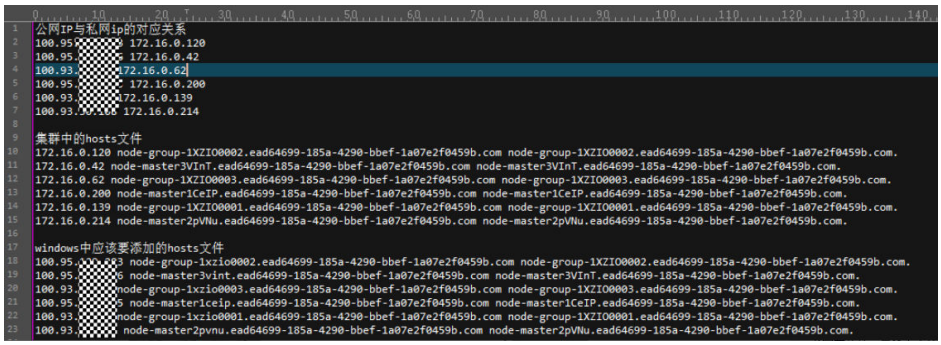
操作步骤

步骤1 为集群的每个节点申请弹性公网IP，并将本地Windows的hosts文件添加所有节点的公网IP对应主机域名的组合（注意如果主机名中出现大写字母要改成小写）。

1. 在虚拟私有云管理控制台，申请弹性公网IP（集群有几个节点就买几个），并分别单击MRS集群的节点名称，在节点的“弹性公网IP”页面绑定弹性公网IP。

   具体操作请参见“虚拟私有云 > 用户指南 > 弹性公网IP > 为弹性云服务器申请和绑定弹性公网IP”。

2. 记录公网IP和私网IP的对应关系将hosts文件中的私网IP改为对应的公网IP。



步骤2 配置集群安全组规则。

1. 在集群“概览”界面，选择“添加安全组规则 > 管理安全组规则”。

2. 在“入方向规则”页签，选择“添加规则”，在“添加入方向规则”窗口配置Windows的IP和8020、9866端口。

步骤3 在Manager界面选择“集群 > 服务 > Spark2x > 更多 > 下载客户端”，将客户端中的core-site.xml和hdfs-site.xml复制到样例工程的conf目录下。

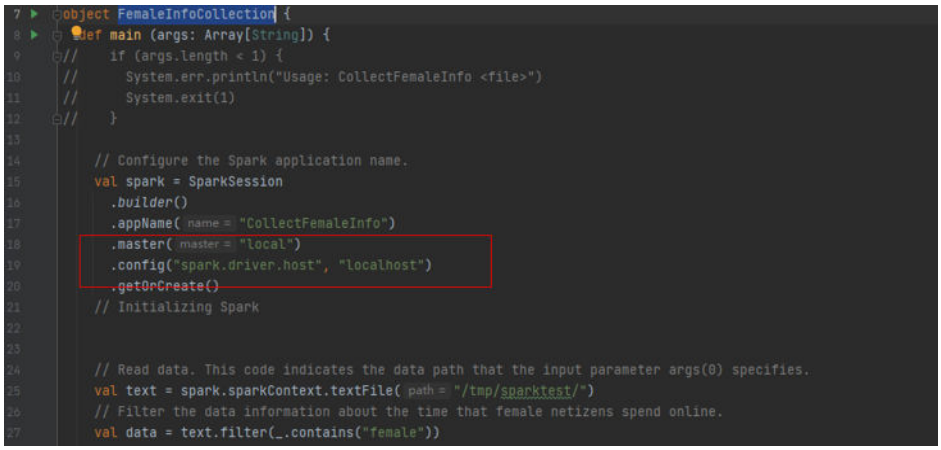
对hdfs-site.xml添加如下内容：

```
<property>
      <name>dfs.client.use.datanode.hostname</name>
      <value>true</value>
</property>
```

对pom.xml文件加入如下内容：

```
<dependency>
    <groupId>com.huawei.mrs</groupId>
    <artifactId>hadoop-plugins</artifactId>
    <version>部件包版本-302002</version>
</dependency>
```

步骤4 运行样例代码前，对SparkSession加入.master("local").config("spark.driver.host", "localhost")，配置Spark为本地运行模式。



----结束

28.6.1.2 在本地 Windows 环境中编包并运行 Spark 程序

操作场景

在程序代码完成开发后，您可以在Windows环境中运行应用。使用Scala或Java语言开发的应用程序在IDEA端的运行步骤是一样的。

说明

- Windows环境中目前只提供通过JDBC访问Spark SQL的程序样例代码的运行，其他样例代码暂不提供。
- 用户需保证Maven已配置华为镜像站中SDK的Maven镜像仓库，具体可参考**配置华为开源镜像仓**。

操作步骤

步骤1 获取样例代码。

下载样例工程的Maven工程源码和配置文件，请参见**获取代码样例工程**。

将样例代码导入IDEA中。

步骤2 获取配置文件。

从集群的客户端中获取文件。在“$SPARK_HOME/conf”中下载hive-site.xml与spark-defaults.conf文件到本地。

步骤3 在HDFS中上传数据。

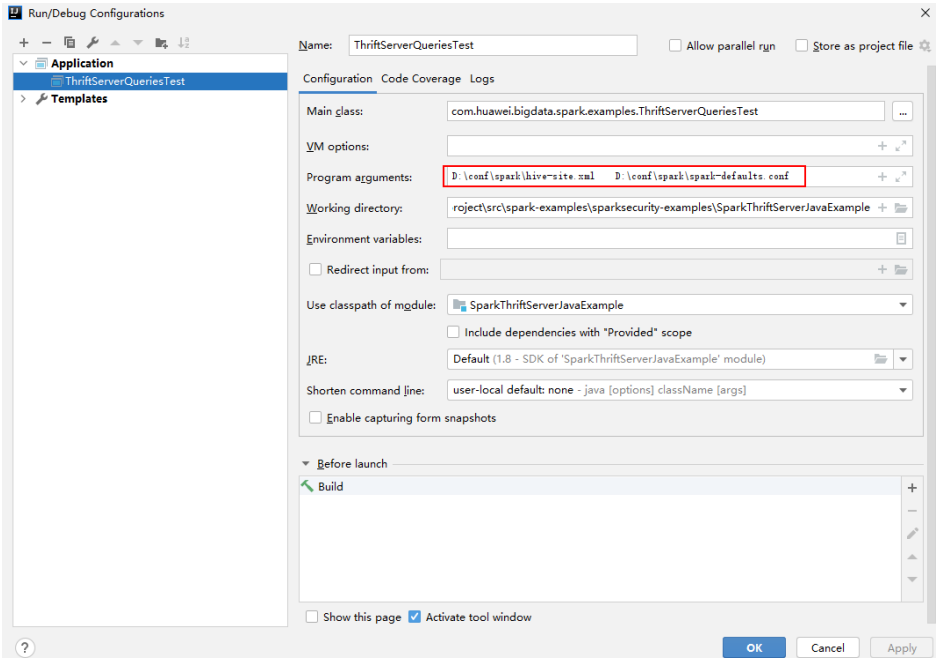
1. 在Liunx中新建文本文件data，将如下数据内容保存到data文件中。
   ```
   Miranda,32
   Karlie,23
   Candice,27
   ```
2. 在Linux系统HDFS客户端使用命令**hadoop fs -mkdir /data**（hdfs dfs命令有同样的作用），创建对应目录。
3. 在Linux系统HDFS客户端使用命令**hadoop fs -put data /data**，上传数据文件。

步骤4 在样例代码中配置相关参数。

将加载数据的sql语句改为“LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD”。


```
ArrayList<String> sqlList = new ArrayList<>();
sqlList.add("CREATE TABLE IF NOT EXISTS CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY"
    + " ','");
sqlList.add("LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD");
sqlList.add("SELECT * FROM child");
sqlList.add("DROP TABLE child");
executeSql(url, sqlList);
```

步骤5 在程序运行时添加运行参数，分别为hive-site.xml与spark-defaults.conf文件的路径。



步骤6 运行程序。

----结束

28.6.1.3 在本地 Windows 环境中查看 Spark 程序调试结果

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/D:/mavenlocal/org/apache/logging/log4j/log4j-slf4j-impl/2.6.2/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/D:/mavenlocal/org/slf4j/slf4j-log4j12/1.7.30/slf4j-log4j12-1.7.30.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
ERROR StatusLogger No log4j2 configuration file found. Using default configuration: logging only errors to the console.
---- Begin executing sql: CREATE TABLE IF NOT EXISTS CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ----
Result
---- Done executing sql: CREATE TABLE IF NOT EXISTS CHILD (NAME STRING, AGE INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ----
---- Begin executing sql: LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD ----
Result
---- Done executing sql: LOAD DATA INPATH 'hdfs:/data/data' INTO TABLE CHILD ----
---- Begin executing sql: SELECT * FROM child ----
NAME    AGE
Miranda    32
Karlie    23
Candice    27
---- Done executing sql: SELECT * FROM child ----
---- Begin executing sql: DROP TABLE child ----
Result
---- Done executing sql: DROP TABLE child ----
```

```
Process finished with exit code 0
```

28.6.2 在 Linux 环境中调测 Spark 应用

28.6.2.1 在 Linux 环境中编包并运行 Spark 程序

操作场景

在程序代码完成开发后，您可以上传至Linux客户端环境中运行应用。使用Scala或Java语言开发的应用程序在Spark客户端的运行步骤是一样的。

说明

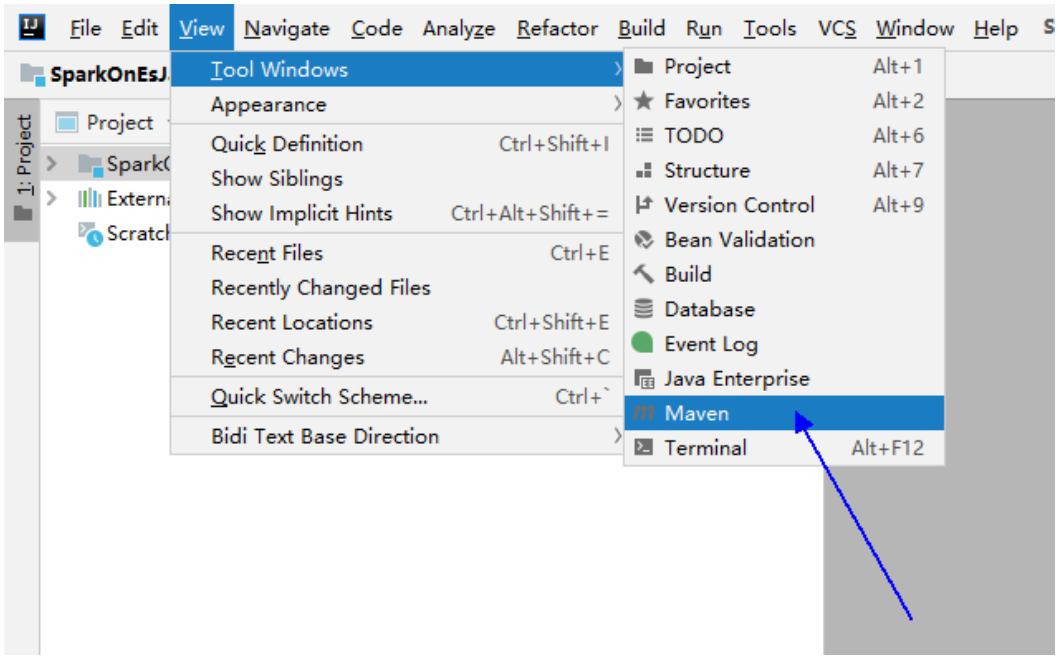
- 使用Python开发的Spark应用程序无需打包成jar，只需将样例工程复制到编译机器上即可。
- 用户需保证worker和driver的Python版本一致，否则将报错："Python in worker has different version %s than that in driver %s."。
- 用户需保证Maven已配置华为镜像站中SDK的Maven镜像仓库，具体可参考**配置华为开源镜像仓**

操作步骤

步骤1 在IntelliJ IDEA中，打开Maven工具窗口。

在IDEA主页面，选择“View->Tool Windows-> > Maven”打开“Maven”工具窗口。

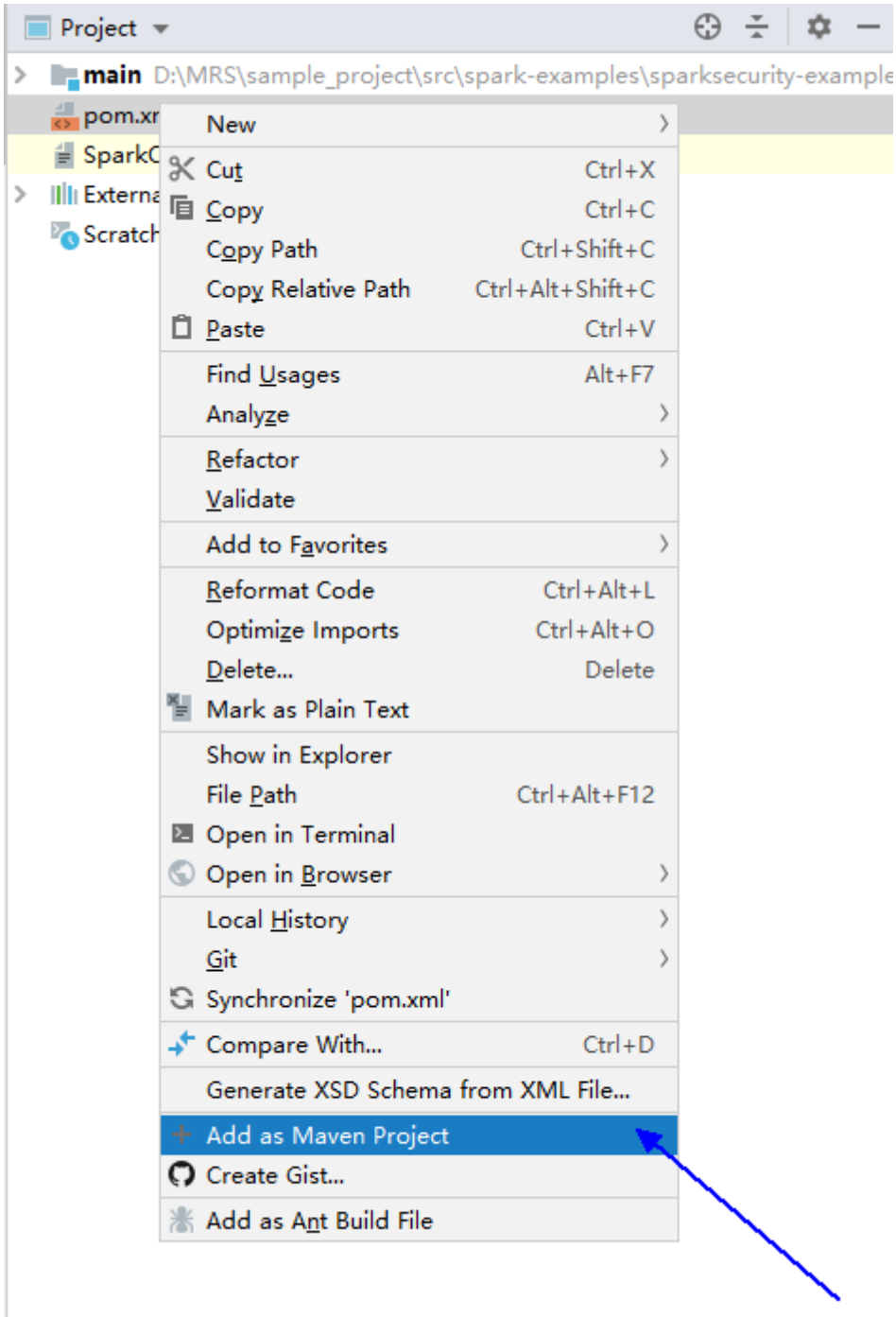
图 28-35 打开 Maven 工具窗口



若项目未通过maven导入，需要执行以下步骤：

右键选择单击样例代码项目中的pom文件，选择“Add as Maven Project”，添加Maven项目

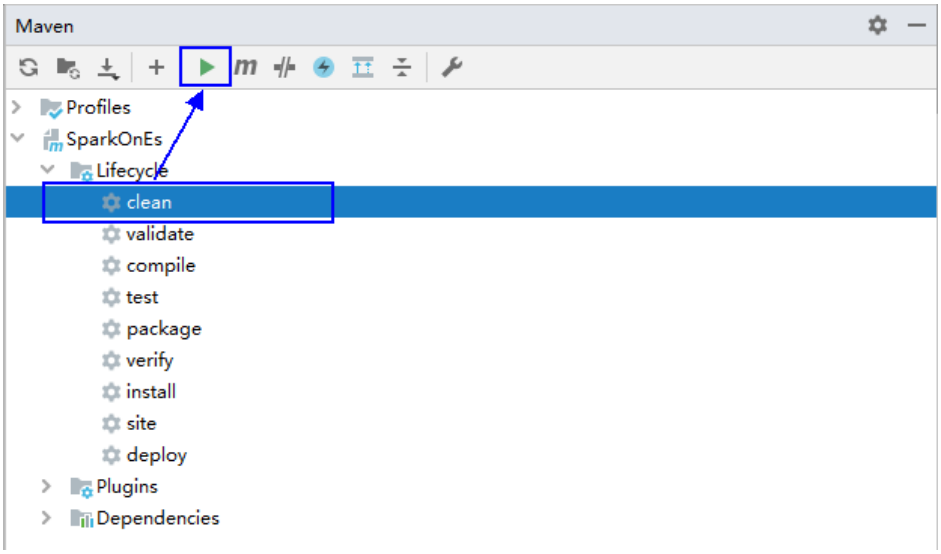
图 28-36 添加 Maven 项目



步骤2 通过Maven生成Jar包。

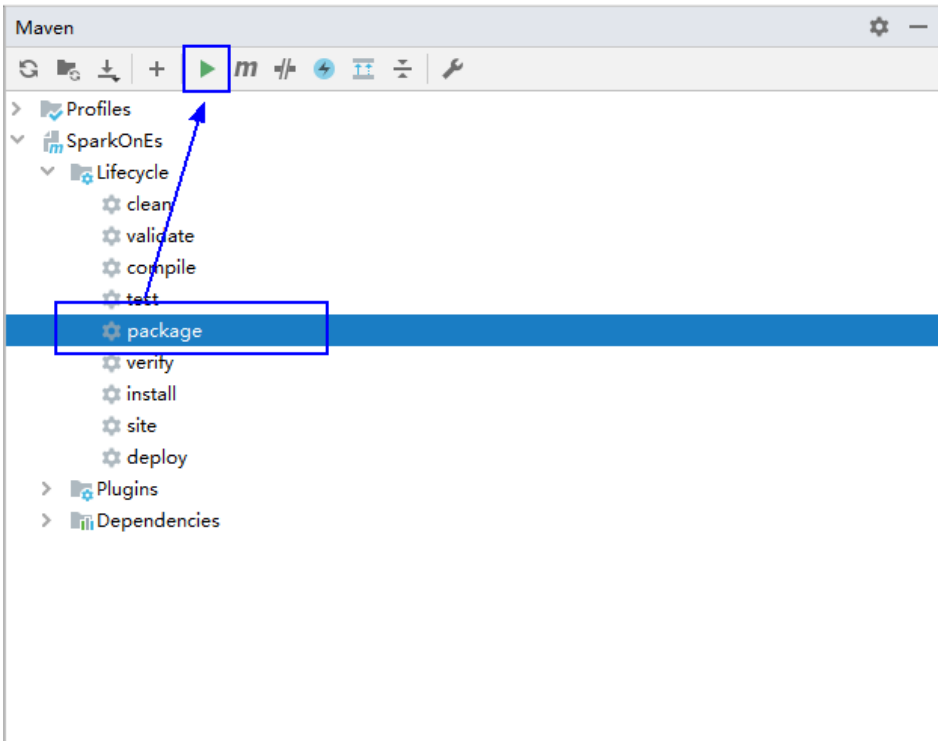
1. 在Maven工具窗口，选择clean生命周期，执行Maven构建过程。

图 28-37 选择 clean 生命周期，执行 Maven 构建过程



2. 在Maven工具窗口，选择package生命周期，执行Maven构建过程。

图 28-38 选择 package 生命周期，执行 Maven 构建过程



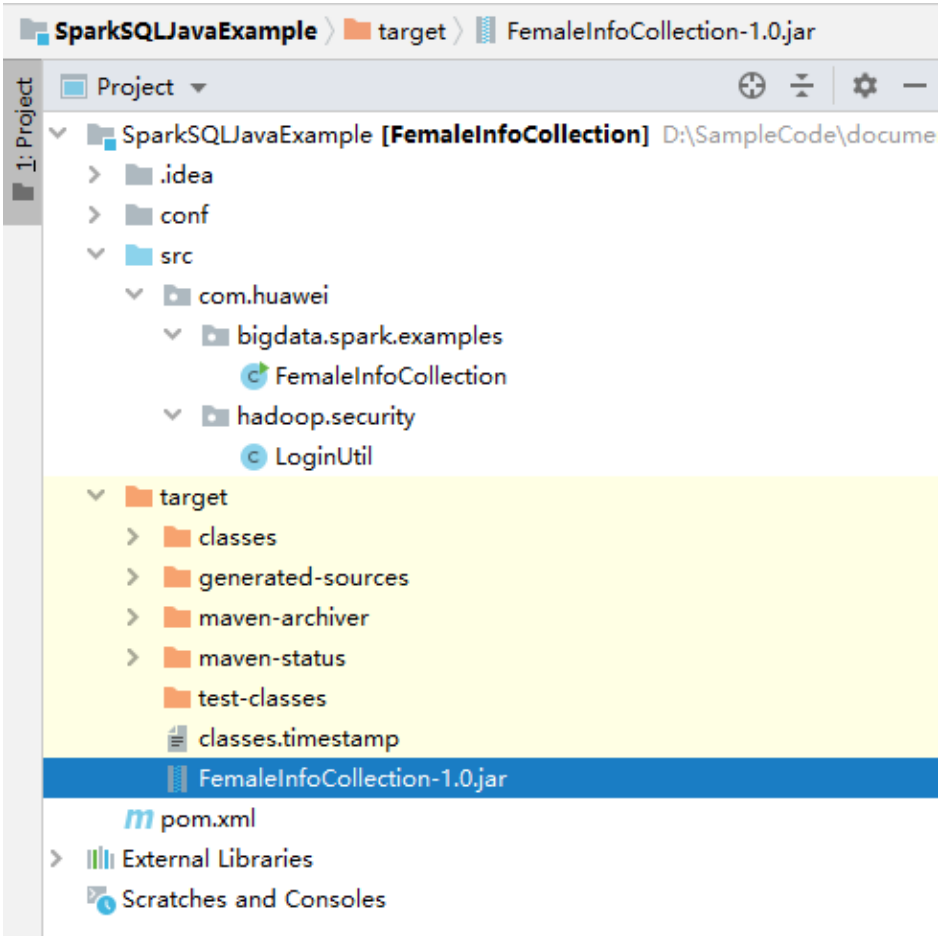
在Run:中出现下面提示，则说明打包成功

图 28-39 打包成功提示

```
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ FemaleInfoCollection ---
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ FemaleInfoCollection ---
[INFO] Building jar: D:\SampleCode\document\VT\code\sparksecurity-examples\SparkSQLJavaExample\target\FemaleInfoCollection-1.0.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  19.427 s
[INFO] Finished at: 2020-09-21T11:17:31+08:00
[INFO] ------------------------------------------------------------------------
```

3. 您可以从项目目录下的target文件夹中获取到Jar包。

图 28-40 获取 jar 包



步骤3 将步骤2中生成的Jar包（如CollectFemaleInfo.jar）复制到Spark运行环境下（即Spark客户端），如“/opt/female”。运行Spark应用程序，具体样例程序可参考开发Spark应用。

注意

在Spark任务运行过程中禁止重启HDFS服务或者重启所有DataNode实例，否则可能会导致任务失败，并可能导致JobHistory部分数据丢失。

----结束

28.6.2.2 在 Linux 环境中查看 Spark 程序调测结果

操作场景

Spark应用程序运行完成后，可通过如下方式查看应用程序的运行情况。

- 通过运行结果数据查看应用程序运行情况。
- 登录Spark WebUI查看应用程序运行情况。
- 通过Spark日志获取应用程序运行情况。

操作步骤

- **查看Spark应用运行结果数据。**
  结果数据存储路径和格式已经由Spark应用程序指定，可通过指定文件获取。
- **查看Spark应用程序运行情况。**
  Spark主要有两个Web页面。
  - Spark UI页面，用于展示正在执行的应用的运行情况。
    页面主要包括了Jobs、Stages、Storage、Environment和Executors五个部分。Streaming应用会多一个Streaming标签页。
    页面入口：在YARN的Web UI界面，查找到对应的Spark应用程序。单击应用信息的最后一列“ApplicationMaster”，即可进入SparkUI页面。
  - History Server页面，用于展示已经完成的和未完成的Spark应用的运行情况。
    页面包括了应用ID、应用名称、开始时间、结束时间、执行时间、所属用户等信息。单击应用ID，页面将跳转到该应用的SparkUI页面。
- **查看Spark日志获取应用运行情况。**
  您可以查看Spark日志了解应用运行情况，并根据日志信息调整应用程序。相关日志信息可参考**Spark2x日志介绍**。

28.7 Spark 应用开发常见问题

28.7.1 Spark 常用 API 介绍

28.7.1.1 Spark Java API 接口介绍

由于Spark开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

Spark Core 常用接口

Spark主要使用到如下这几个类：

- JavaSparkContext：是Spark的对外接口，负责向调用该类的Java应用提供Spark的各种功能，如连接Spark集群，创建RDD，累积量和广播量等。它的作用相当于一个容器。
- SparkConf：Spark应用配置类，如设置应用名称，执行模式，executor内存等。

- JavaRDD：用于在java应用中定义JavaRDD的类，功能类似于scala中的RDD(Resilient Distributed Dataset)类。
- JavaPairRDD：表示key-value形式的JavaRDD类。提供的方法有groupByKey，reduceByKey等。
- Broadcast：广播变量类。广播变量允许保留一个只读的变量，缓存在每一台机器上，而非每个任务保存一份复制。
- StorageLevel：数据存储级别。有内存（MEMORY_ONLY），磁盘（DISK_ONLY），内存+磁盘（MEMORY_AND_DISK）等。

JavaRDD支持两种类型的操作：Transformation和Action，这两种类型的常用方法如表28-6和表28-7。

表 28-6 Transformation

方法	说明
<R> JavaRDD<R> map(Function<T,R> f)	对RDD中的每个element使用Function。
JavaRDD<T> filter(Function<T,Boolean> f)	对RDD中所有元素调用Function，返回为true的元素。
<U> JavaRDD<U> flatMap(FlatMapFunction<T,U> f)	先对RDD所有元素调用Function，然后将结果扁平化。
JavaRDD<T> sample(boolean withReplacement, double fraction, long seed)	抽样。
JavaRDD<T> distinct(int numPartitions)	去除重复元素。
JavaPairRDD<K,Iterable<V>> groupByKey(int numPartitions)	返回(K,Seq[V])，将key相同的value组成一个集合。
JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> func, int numPartitions)	对key相同的value调用Function。
JavaPairRDD<K,V> sortByKey(boolean ascending, int numPartitions)	按照key来进行排序，是升序还是降序，ascending是boolean类型。

方法	说明
JavaPairRDD<K,scala.Tuple2<V,W>> join(JavaPairRDD<K,W> other)	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,(V,W))的dataset，numTasks为并发的任务数。
JavaPairRDD<K,scala.Tuple2<Iterable<V>,Iterable<W>>> cogroup(JavaPairRDD<K,W> other, int numPartitions)	当有两个KV的dataset(K,V)和(K,W)，返回的是<K,scala.Tuple2<Iterable<V>,Iterable<W>>>的dataset，numTasks为并发的任务数。
JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)	返回该RDD与其它RDD的笛卡尔积。

表 28-7 Action

方法	说明
T reduce(Function2<T,T,T> f)	对RDD中的元素调用Function2。
java.util.List<T> collect()	返回包含RDD中所有元素的一个数组。
long count()	返回的是dataset中的element的个数。
T first()	返回的是dataset中的第一个元素。
java.util.List<T> take(int num)	返回前n个elements。
java.util.List<T> takeSample(boolean withReplacement, int num, long seed)	对dataset随机抽样，返回有num个元素组成的数组。withReplacement表示是否使用replacement。
void saveAsTextFile(String path, Class<? extends org.apache.hadoop.io.compress.CompressionCodec> codec)	把dataset写到一个text file、hdfs、或者hdfs支持的文件系统中，spark把每条记录都转换为一行记录，然后写到file中。
java.util.Map<K,Object> countByKey()	对每个key出现的次数做统计。
void foreach(VoidFunction<T> f)	在数据集的每一个元素上，运行函数func。

方法	说明
java.util.Map<T,Long> countByValue()	对RDD中每个元素出现的次数进行统计。

表 28-8 Spark Core 新增接口

API	说明
public java.util.concurrent.atomic.AtomicBoolean isSparkContextDown()	该接口可判断sparkContext是否已完全stop，初始值为false。 若接口值为true，则代表sparkContext已完全stop。 若接口值为false，则代表sparkContext没有完成stop。 例如：用户根据jsc.sc().isSparkContextDown().get() == true 可判断sparkContext已完全stop。

Spark Streaming 常用接口

Spark Streaming中常见的类有：

- JavaStreamingContext：是Spark Streaming功能的主入口，负责提供创建DStreams的方法，入参中需要设置批次的时间间隔。
- JavaDStream：是一种代表RDDs连续序列的数据类型，代表连续数据流。
- JavaPairDStream：KV DStream的接口，提供reduceByKey和join等操作。
- JavaReceiverInputDStream<T>：定义任何从网络接受数据的输入流。

Spark Streaming的常见方法与Spark Core类似，下表罗列了Spark Streaming特有的一些方法。

表 28-9 Spark Streaming 方法

方法	说明
JavaReceiverInputDStream<java.lang.String> socketStream(java.lang.String hostname,int port)	创建一个输入流，通过TCP socket从对应的hostname和端口接受数据。接受的字节被解析为UTF8格式。默认的存储级别为Memory+Disk。
JavaDStream<java.lang.String> textFileStream(java.lang.String directory)	入参directory为HDFS目录，该方法创建一个输入流检测可兼容Hadoop文件系统的新文件，并且读取为文本文件。
void start()	启动Spark Streaming计算。

方法	说明
void awaitTermination()	当前进程等待终止，如Ctrl+C等。
void stop()	终止Spark Streaming计算。
<T> JavaDStream<T> transform(java.util.List<JavaDStream<?>> dstreams,Function2<java.util.List<JavaRDD<?>>,Time,JavaRDD<T>> transformFunc)	对每个RDD进行function操作，得到一个新的DStream。这个函数中JavaRDDs的顺序和list中对应的DStreams保持一致。
<T> JavaDStream<T> union(JavaDStream<T> first,java.util.List<JavaDStream<T>> rest)	从多个具备相同类型和滑动时间的DStream中创建统一的DStream。

表 28-10 Spark Streaming 增强特性接口

方法	说明
JAVADStreamKafkaWriter.writeToKafka()	支持将DStream中的数据批量写入到Kafka。
JAVADStreamKafkaWriter.writeToKafkaBySingle()	支持将DStream中的数据逐条写入到Kafka。

Spark SQL 常用接口

Spark SQL中重要的类有：

- SQLContext：是Spark SQL功能和DataFrame的主入口。
- DataFrame：是一个以命名列方式组织的分布式数据集
- DataFrameReader：从外部存储系统加载DataFrame的接口。
- DataFrameStatFunctions：实现DataFrame的统计功能。
- UserDefinedFunction：用户自定义的函数。

常见的Actions方法有：

表 28-11 Spark SQL 方法介绍

方法	说明
Row[] collect()	返回一个数组，包含DataFrame的所有列。
long count()	返回DataFrame的行数。

方法	说明
DataFrame describe(java.lang.String... cols)	计算统计信息，包含计数，平均值，标准差，最小值和最大值。
Row first()	返回第一行。
Row[] head(int n)	返回前n行。
void show()	用表格形式显示DataFrame的前20行。
Row[] take(int n)	返回DataFrame中的前n行。

表 28-12 基本的 DataFrame Functions 介绍

方法	说明
void explain(boolean extended)	打印出SQL语句的逻辑计划和物理计划。
void printSchema()	打印schema信息到控制台。
registerTempTable	将DataFrame注册为一张临时表，其周期和SQLContext绑定在一起。
DataFrame toDF(java.lang.String... colNames)	返回一个列重命名的DataFrame。
DataFrame sort(java.lang.String sortCol,java.lang.String... sortCols)	根据不同的列，按照升序或者降序排序。
GroupedData rollup(Column... cols)	对当前的DataFrame特定列进行多维度的回滚操作。

28.7.1.2 Spark Scala API 接口介绍

由于Spark开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

Spark Core 常用接口

Spark主要使用到如下这几个类：

- SparkContext：是Spark的对外接口，负责向调用该类的scala应用提供Spark的各种功能，如连接Spark集群，创建RDD等。
- SparkConf：Spark应用配置类，如设置应用名称，执行模式，executor内存等。
- RDD（Resilient Distributed Dataset）：用于在Spark应用程序中定义RDD的类，该类提供数据集的操作方法，如map，filter。

- PairRDDFunctions：为key-value对的RDD数据提供运算操作，如groupByKey。
- Broadcast：广播变量类。广播变量允许保留一个只读的变量，缓存在每一台机器上，而非每个任务保存一份复制。
- StorageLevel：数据存储级别。有内存（MEMORY_ONLY），磁盘（DISK_ONLY），内存+磁盘（MEMORY_AND_DISK）等。

RDD上支持两种类型的操作：Transformation和Action，这两种类型的常用方法如表28-13和表28-14所示。

表 28-13 Transformation

方法	说明
map[U](f: (T) => U): RDD[U]	对调用map的RDD数据集中的每个element都使用f方法，生成新的RDD。
filter(f: (T) => Boolean): RDD[T]	对RDD中所有元素调用f方法，生成将满足条件数据集以RDD形式返回。
flatMap[U](f: (T) => TraversableOnce[U])(implicit arg0: ClassTag[U]): RDD[U]	先对RDD所有元素调用f方法，然后将结果扁平化，生成新的RDD。
sample(withReplacement: Boolean, fraction: Double, seed: Long = Utils.random.nextLong): RDD[T]	抽样，返回RDD一个子集。
union(other: RDD[T]): RDD[T]	返回一个新的RDD，包含源RDD和给定RDD的元素的集合。
distinct([numPartitions: Int]): RDD[T]	去除重复元素，生成新的RDD。
groupByKey(): RDD[(K, Iterable[V])]	返回(K,Iterable[V])，将key相同的value组成一个集合。
reduceByKey(func: (V, V) => V[, numPartitions: Int]): RDD[(K, V)]	对key相同的value调用func。
sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.length): RDD[(K, V)]	按照key来进行排序，是升序还是降序，ascending是boolean类型。
join[W](other: RDD[(K, W)][, numPartitions: Int]): RDD[(K, (V, W))]	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,(V,W))的dataset，numPartitions为并发的任务数。

方法	说明
cogroup[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W]))]	将当有两个key-value对的dataset(K,V)和(K,W)，返回的是(K, (Iterable[V], Iterable[W]))的dataset，numPartitions为并发的任务数。
cartesian[U](other: RDD[U])(implicit arg0: ClassTag[U]): RDD[(T, U)]	返回该RDD与其它RDD的笛卡尔积。

表 28-14 Action

API	说明
reduce(f: (T, T) => T):	对RDD中的元素调用f。
collect(): Array[T]	返回包含RDD中所有元素的一个数组。
count(): Long	返回的是dataset中的element的个数。
first(): T	返回的是dataset中的第一个元素。
take(num: Int): Array[T]	返回前n个elements。
takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong): Array[T]	takeSample(withReplacement，num，seed)对dataset随机抽样，返回有num个元素组成的数组。withReplacement表示是否使用replacement。
saveAsTextFile(path: String): Unit	把dataset写到一个text file、HDFS或者HDFS支持的文件系统中，spark把每条记录都转换为一行记录，然后写到file中。
saveAsSequenceFile(path: String, codec: Option[Class[_ <: CompressionCodec]] = None): Unit	只能用在key-value对上，然后生成SequenceFile写到本地或者hadoop文件系统。
countByKey(): Map[K, Long]	对每个key出现的次数做统计。
foreach(func: (T) => Unit): Unit	在数据集的每一个元素上，运行函数func。
countByValue()(implicit ord: Ordering[T] = null): Map[T, Long]	对RDD中每个元素出现的次数进行统计。

表 28-15 Spark Core 新增接口

API	说明
isSparkContextDown:AtomicBoolean	该接口可判断sparkContext是否已完全stop，初始值为false。 若接口值为true，则代表sparkContext已完全stop。 若接口值为false，则代表sparkContext没有完成stop。 例如：用户根据sc.isSparkContextDown.get() == true可判断sparkContext已完全stop。

Spark Streaming 常用接口

Spark Streaming中常见的类有：

- StreamingContext：是Spark Streaming功能的主入口，负责提供创建DStreams的方法，入参中需要设置批次的时间间隔。
- dstream.DStream：是一种代表RDDs连续序列的数据类型，代表连续数据流。
- dstream.PariDStreamFunctions：键值对的DStream，常见的操作如groupByKey和reduceByKey。

  对应的Spark Streaming的JAVA API是JavaStreamingContext，JavaDStream和JavaPairDStream。

Spark Streaming的常见方法与Spark Core类似，下表罗列了Spark Streaming特有的一些方法。

表 28-16 Spark Streaming 方法介绍

方法	说明
socketTextStream(hostname: String, port: Int, storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2): ReceiverInputDStream[String]	从TCP源主机：端口创建一个输入流。
start():Unit	启动Spark Streaming计算。
awaitTermination(timeout: long):Unit	当前进程等待终止，如Ctrl+C等。
stop(stopSparkContext: Boolean, stopGracefully: Boolean): Unit	终止Spark Streaming计算。

方法	说明
transform[T] (dstreams: Seq[DStream[_]], transformFunc: (Seq[RDD[_]], Time) ? RDD[T])(implicit arg0: ClassTag[T]): DStream[T]	对每一个RDD应用function操作得到一个新的DStream。
UpdateStateByKey(func)	更新DStream的状态。使用此方法，需要定义状态和状态更新函数。
window(windowLength, slideInterval)	根据源DStream的窗口批次计算得到一个新的DStream。
countByWindow(windowLength, slideInterval)	返回流中滑动窗口元素的个数。
reduceByWindow(func, windowLength, slideInterval)	当调用在DStream的KV对上，返回一个新的DStream的KV对，其中每个Key的Value根据滑动窗口中批次的reduce函数聚合得到。
join(otherStream, [numTasks])	实现不同的Spark Streaming之间做合并操作。
DStreamKafkaWriter.writeToKafka()	支持将DStream中的数据批量写入到Kafka。
DStreamKafkaWriter.writeToKafkaBySingle()	支持将DStream中的数据逐条写入到Kafka。

表 28-17 Spark Streaming 增强特性接口

方法	说明
DStreamKafkaWriter.writeToKafka()	支持将DStream中的数据批量写入到Kafka。
DStreamKafkaWriter.writeToKafkaBySingle()	支持将DStream中的数据逐条写入到Kafka。

SparkSQL 常用接口

Spark SQL中常用的类有：

- SQLContext：是Spark SQL功能和DataFrame的主入口。

- DataFrame：是一个以命名列方式组织的分布式数据集。
- HiveContext：获取存储在Hive中数据的主入口。

表 28-18 常用的 Actions 方法

方法	说明
collect(): Array[Row]	返回一个数组，包含DataFrame的所有列。
count(): Long	返回DataFrame中的行数。
describe(cols: String*): DataFrame	计算统计信息，包含计数，平均值，标准差，最小值和最大值。
first(): Row	返回第一行。
Head(n:Int): Row	返回前n行。
show(numRows: Int, truncate: Boolean): Unit	用表格形式显示DataFrame。
take(n:Int): Array[Row]	返回DataFrame中的前n行。

表 28-19 基本的 DataFrame Functions

方法	说明
explain(): Unit	打印出SQL语句的逻辑计划和物理计划。
printSchema(): Unit	打印schema信息到控制台。
registerTempTable(tableName: String): Unit	将DataFrame注册为一张临时表，其周期和SQLContext绑定在一起。
toDF(colNames: String*): DataFrame	返回一个列重命名的DataFrame。

28.7.1.3 Spark Python API 接口介绍

由于Spark开源版本升级，为避免出现API兼容性或可靠性问题，建议用户使用配套版本的API。

Spark Core 常用接口

Spark主要使用到如下这几个类：

- pyspark.SparkContext：是Spark的对外接口。负责向调用该类的python应用提供Spark的各种功能，如连接Spark集群、创建RDD、广播变量等。
- pyspark.SparkConf：Spark应用配置类。如设置应用名称，执行模式，executor内存等。

- pyspark.RDD（Resilient Distributed Dataset）：用于在Spark应用程序中定义RDD的类，该类提供数据集的操作方法，如map，filter。
- pyspark.Broadcast：广播变量类。广播变量允许保留一个只读的变量，缓存在每一台机器上，而非每个任务保存一份复制。
- pyspark.StorageLevel： 数据存储级别。有内存（MEMORY_ONLY），磁盘（DISK_ONLY），内存+磁盘（MEMORY_AND_DISK）等。
- pyspark.sql.SQLContext：是SparkSQL功能的主入口。可用于创建DataFrame，注册DataFrame为一张表，表上执行SQL等。
- pyspark.sql.DataFrame：分布式数据集。DataFrame等效于SparkSQL中的关系表，可被SQLContext中的方法创建。
- pyspark.sql.DataFrameNaFunctions：DataFrame中处理数据缺失的函数。
- pyspark.sql.DataFrameStatFunctions：DataFrame中统计功能的函数，可以计算列之间的方差，样本协方差等。

RDD上支持两种类型的操作：transformation和action，这两种类型的常用方法如表28-20和表28-21。

表 28-20 Transformation

方法	说明
map(f, preservesPartitioning=False)	对调用map的RDD数据集中的每个element都使用Func，生成新的RDD。
filter(f)	对RDD中所有元素调用Func，生成将满足条件数据集以RDD形式返回。
flatMap(f, preservesPartitioning=False)	先对RDD所有元素调用Func，然后将结果扁平化，生成新的RDD。
sample(withReplacement, fraction, seed=None)	抽样，返回RDD一个子集。
union(rdds)	返回一个新的RDD，包含源RDD和给定RDD的元素的集合。
distinct([numPartitions: Int]): RDD[T]	去除重复元素，生成新的RDD。
groupByKey(): RDD[(K, Iterable[V])]	返回(K,Iterable[V])，将key相同的value组成一个集合。
reduceByKey(func, numPartitions=None)	对key相同的value调用Func。
sortByKey(ascending=True, numPartitions=None, keyfunc=function <lambda>)	按照key来进行排序，是升序还是降序，ascending是boolean类型。

方法	说明
join(other, numPartitions)	当有两个KV的dataset(K,V)和(K,W)，返回的是(K,(V,W))的dataset,numPartitions为并发的任务数。
cogroup(other, numPartitions)	将当有两个key-value对的dataset(K,V)和(K,W)，返回的是(K, (Iterable[V], Iterable[W]))的dataset,numPartitions为并发的任务数。
cartesian(other)	返回该RDD与其它RDD的笛卡尔积。

表 28-21 Action

API	说明
reduce(f)	对RDD中的元素调用Func。
collect()	返回包含RDD中所有元素的一个数组。
count()	返回的是dataset中的element的个数。
first()	返回的是dataset中的第一个元素。
take(num)	返回前num个elements。
takeSample(withReplacement, num, seed)	takeSample(withReplacement，num，seed)对dataset随机抽样，返回有num个元素组成的数组。withReplacement表示是否使用replacement。
saveAsTextFile(path, compressionCodecClass)	把dataset写到一个text file、HDFS或者HDFS支持的文件系统中，spark把每条记录都转换为一行记录，然后写到file中。
saveAsSequenceFile(path, compressionCodecClass=None)	只能用在key-value对上，然后生成SequenceFile写到本地或者hadoop文件系统。
countByKey()	对每个key出现的次数做统计。
foreach(func)	在数据集的每一个元素上，运行函数。
countByValue()	对RDD中每个不同value出现的次数进行统计。

Spark Streaming 常用接口

Spark Streaming中常见的类有：

- pyspark.streaming.StreamingContext：是Spark Streaming功能的主入口，负责提供创建DStreams的方法，入参中需要设置批次的时间间隔。
- pyspark.streaming.DStream：是一种代表RDDs连续序列的数据类型，代表连续数据流。

- dsteam.PariDStreamFunctions：键值对的DStream，常见的操作如groupByKey和reduceByKey。

  对应的Spark Streaming的JAVA API是JavaStreamingContext，JavaDStream和JavaPairDStream。

Spark Streaming的常见方法与Spark Core类似，下表罗列了Spark Streaming特有的一些方法。

表 28-22 Spark Streaming 常用接口介绍

方法	说明
socketTextStream(hostname, port, storageLevel)	从TCP源主机：端口创建一个输入流。
start()	启动Spark Streaming计算。
awaitTermination(timeout)	当前进程等待终止，如Ctrl+C等。
stop(stopSparkContext, stopGraceFully)	终止Spark Streaming计算，stopSparkContext用于判断是否需要终止相关的SparkContext，StopGracefully用于判断是否需要等待所有接受到的数据处理完成。
UpdateStateByKey(func)	更新DStream的状态。使用此方法，需要定义State和状态更新函数。
window(windowLength, slideInterval)	根据源DStream的窗口批次计算得到一个新的DStream。
countByWindow(windowLength, slideInterval)	返回流中滑动窗口元素的个数。
reduceByWindow(func, windowLength, slideInterval)	当调用在DStream的KV对上，返回一个新的DStream的KV对，其中每个Key的Value根据滑动窗口中批次的reduce函数聚合得到。
join(other,numPartitions)	实现不同的Spark Streaming之间做合并操作。

SparkSQL 常用接口

Spark SQL中在Python中重要的类有：

- pyspark.sql.SQLContext：是Spark SQL功能和DataFrame的主入口。
- pyspark.sql.DataFrame：是一个以命名列方式组织的分布式数据集。
- pyspark.sql.HiveContext：获取存储在Hive中数据的主入口。
- pyspark.sql.DataFrameStatFunctions：统计功能中一些函数。
- pyspark.sql.functions：DataFrame中内嵌的函数。
- pyspark.sql.Window：sql中提供窗口功能。

表 28-23 Spark SQL 常用的 Action

方法	说明
collect()	返回一个数组，包含DataFrame的所有列。
count()	返回DataFrame中的行数。
describe()	计算统计信息，包含计数，平均值，标准差，最小值和最大值。
first()	返回第一行。
head(n)	返回前n行。
show()	用表格形式显示DataFrame。
take(num)	返回DataFrame中的前num行。

表 28-24 基本的 DataFrame Functions

方法	说明
explain()	打印出SQL语句的逻辑计划和物理计划。
printSchema()	打印schema信息到控制台。
registerTempTable(name)	将DataFrame注册为一张临时表，命名为name，其周期和SQLContext绑定在一起。
toDF()	返回一个列重命名的DataFrame。

28.7.1.4 Spark client CLI 介绍

Spark CLI详细的使用方法参考官方网站的描述：**http://spark.apache.org/docs/3.1.1/quick-start.html**

常用 CLI

Spark常用的CLI如下所示：

- **spark-shell**

  提供了一个简单学习API的方法，类似于交互式数据分析的工具。同时支持Scala和Python两种语言。在Spark目录下，执行***./bin/spark-shell***即可进入Scala交互式界面从HDFS中获取数据，再操作RDD。

  示例：一行代码可以实现统计一个文件中所有单词。

  scala> sc.textFile("hdfs://10.96.1.57:9000//wordcount_data.txt").flatMap(l => l.split(" ")).map(w => (w,1)).reduceByKey(_+_).collect()

- **spark-submit**

  用于提交Spark应用到Spark集群中运行，返回运行结果。需要指定class、master、jar包以及入参。

示例：执行jar包中的GroupByTest例子，入参为4个，指定集群运行模式是local单核运行。

./bin/spark-submit --class org.apache.spark.examples.GroupByTest --master local[1] examples/jars/spark-examples_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar 6 10 10 3

- **spark-sql**

  可用于local模式或者集群模式运行Hive元数据服务以及命令行查询。如果需要查看其逻辑计划，只需在SQL语句前面加上explain extended即可。

  示例：

  Select key from src group by key

- **run-example**

  用来运行或者调试Spark开源社区中的自带的example。

  示例：执行SparkPi。

  ./run-example SparkPi 100

28.7.1.5 Spark JDBCServer 接口介绍

简介

JDBCServer是Hive中的HiveServer2的另外一个实现，它底层使用了Spark SQL来处理SQL语句，从而比Hive拥有更高的性能。

JDBCServer是一个JDBC接口，用户可以通过JDBC连接JDBCServer来访问SparkSQL的数据。JDBCServer在启动的时候，会启动一个sparkSQL的应用程序，而通过JDBC连接进来的客户端共同分享这个sparkSQL应用程序的资源，也就是说不同的用户之间可以共享数据。JDBCServer启动时还会开启一个侦听器，等待JDBC客户端的连接和提交查询。所以，在配置JDBCServer的时候，至少要配置JDBCServer的主机名和端口，如果要使用hive数据的话，还要提供hive metastore的uris。

JDBCServer默认在安装节点上的22550端口起一个JDBC服务（通过参数hive.server2.thrift.port配置），可以通过Beeline或者JDBC客户端代码来连接它，从而执行SQL命令。

如果您需要了解JDBCServer的其他信息，请参见Spark官网：**http://spark.apache.org/docs/3.1.1/sql-programming-guide.html#distributed-sql-engine**。

Beeline

开源社区提供的Beeline连接方式，请参见：**https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients**。

sh CLIENT_HOME/spark/bin/beeline -u "jdbc:hive2://<zkNode_IP>:<zkNode_Port>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;"

说明

- 其中“<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>”是Zookeeper的URL，多个URL以逗号隔开。例如“192.168.81.37:2181,192.168.195.232:2181,192.168.169.84:2181”。
- 其中“sparkthriftserver2x”是Zookeeper上的目录，表示客户端从该目录下随机选择JDBCServer实例进行连接。

JDBC 客户端代码

通过JDBC客户端代码连接JDBCServer，来访问SparkSQL的数据。详细指导请参见**通过JDBC访问Spark SQL样例程序**。

增强特性

对比开源社区，华为还提供了两个增强特性，JDBCServerHA方案和设置JDBCServer连接的超时时间。

- JDBCServerHA方案，多个JDBCServer主节点同时提供服务，当其中一个节点发生故障时，新的客户端连接会分配到其他主节点上，从而保障无间断为集群提供服务。Beeline和JDBC客户端代码两种连接方式的操作相同。
- 设置客户端与JDBCServer连接的超时时间。
  - Beeline

    在网络拥塞的情况下，这个特性可以避免Beeline由于无限等待服务端的返回而挂起。使用方式如下：

    启动Beeline时，在后面追加“--socketTimeOut=n”，其中“n”表示等待服务返回的超时时长，单位为秒，默认为“0”（表示永不超时）。建议根据业务场景，设置为业务所能容忍的最大等待时长。
  - JDBC客户端代码

    在网络拥塞的情况下，这个特性可以避免客户端由于无限等待服务端的返回而挂起。使用方式如下：

    在执行“DriverManager.getConnection”方法获取JDBC连接前，添加“DriverManager.setLoginTimeout(n)”方法来设置超时时长，其中n表示等待服务返回的超时时长，单位为秒，类型为Int，默认为“0”（表示永不超时）。建议根据业务场景，设置为业务所能容忍的最大等待时长。

28.7.2 structured streaming 功能与可靠性介绍

Structured Streaming 支持的功能

1. 支持对流式数据的ETL操作。
2. 支持流式DataFrames或Datasets的schema推断和分区。
3. 流式DataFrames或Datasets上的操作：包括无类型，类似SQL的操作（比如select、where、groupBy），以及有类型的RDD操作（比如map、filter、flatMap）。
4. 支持基于Event Time的聚合计算，支持对迟到数据的处理。
5. 支持对流式数据的去除重复数据操作。
6. 支持状态计算。

7. 支持对流处理任务的监控。
8. 支持批流join，流流join。

   当前JOIN操作支持列表如下：

左表	右表	支持的Join类型	说明
Static	Static	全部类型	即使在流处理中，不涉及流数据的join操作也能全部支持
Stream	Static	Inner	支持，但是无状态
		Left Outer	支持，但是无状态
		Right Outer	不支持
		Full Outer	不支持
Stream	Stream	Inner	支持，左右表可选择使用watermark或者时间范围进行状态清理
		Left Outer	有条件的支持，左表可选择使用watermark进行状态清理，右表必须使用watermark+时间范围
		Right Outer	有条件的支持，右表可选择使用watermark进行状态清理，左表必须使用watermark+时间范围
		Full Outer	不支持

Structured Streaming 不支持的功能

1. 不支持多个流聚合。
2. 不支持limit、first、take这些取N条Row的操作。
3. 不支持Distinct。
4. 只有当output mode为complete时才支持排序操作。
5. 有条件地支持流和静态数据集之间的外连接。
6. 不支持部分DataSet上立即运行查询并返回结果的操作：
   - count()：无法从流式Dataset返回单个计数，而是使用ds.groupBy().count()返回一个包含运行计数的streaming Dataset。
   - foreach()：使用ds.writeStream.foreach(...)代替。
   - show()：使用输出console sink代替。

Structured Streaming 可靠性说明

Structured Streaming通过checkpoint和WAL机制，对可重放的sources，以及支持重复处理的幂等性sinks，可以提供端到端的exactly-once容错语义。

1. 用户可在程序中设置option("checkpointLocation", "checkpoint路径")启用checkpoint。

从checkpoint恢复时，应用程序或者配置可能发生变更，有部分变更会导致从checkpoint恢复失败，具体限制如下：

a. 不允许source的个数或者类型发生变化。

b. source的参数变化，这种情况是否能被支持，取决于source类型和查询语句，例如：

   - 速率控制相关参数的添加、删除和修改，此种情况能被支持，如：spark.readStream.format("kafka").option("subscribe", "topic")变更为spark.readStream.format("kafka").option("subscribe", "topic").option("maxOffsetsPerTrigger", ...)
   - 修改消费的topic/files可能会出现不可预知的问题，如：spark.readStream.format("kafka").option("subscribe", "topic")变更为spark.readStream.format("kafka").option("subscribe", "newTopic")

c. sink的类型发生变化：允许特定的几个sink的组合，具体场景需要验证确认，例如：

   - File sink允许变更为kafka sink，kafka中只处理新数据。
   - kafka sink不允许变更为file sink。
   - kafka sink允许变更为foreach sink，反之亦然。

d. sink的参数变化，这种情况是否能被支持，取决于sink类型和查询语句，例如：

   - 不允许file sink的输出路径发生变更。
   - 允许Kafka sink的输出topic发生变更。
   - 允许foreach sink中的自定义算子代码发生变更，但是变更结果取决于用户代码。

e. Projection、filter和map-like操作变更，局部场景下能够支持，例如：

   - 支持Filter的添加和删除，如：sdf.selectExpr("a")变更为sdf.where(...).selectExpr("a").filter(...)
   - Output schema相同时，projections允许变更，如：sdf.selectExpr("stringColumn AS json").writeStream变更为sdf.select(to_json(...).as("json")).writeStream
   - Output schema不相同时，projections在部分条件下允许变更，如：sdf.selectExpr("a").writeStream变更为sdf.selectExpr("b").writeStream，只有当sink支持“a”到“b”的schema转换时才不会出错。

f. 状态操作的变更，在部分场景下会导致状态恢复失败：

   - Streaming aggregation：如sdf.groupBy("a").agg(...)操作中，不允许分组键或聚合键的类型或者数量发生变化。
   - Streaming deduplication：如：sdf.dropDuplicates("a")操作中，不允许分组键或聚合键的类型或者数量发生变化。

- Stream-stream join：如sdf1.join(sdf2, ...)操作中，关联键的schema不允许发生变化，join类型不允许发生变化，其他join条件的变更可能导致不确定性结果。
- 任意状态计算：如sdf.groupByKey(...).mapGroupsWithState(...)或者sdf.groupByKey(...).flatMapGroupsWithState(...)操作中，用户自定义状态的schema或者超时类型都不允许发生变化；允许用户自定义state-mapping函数变化，但是变更结果取决于用户代码；如果需要支持schema变更，用户可以将状态数据编码/解码成二进制数据以支持schema迁移。

2. Source的容错性支持列表

Sources	支持的Options	容错支持	说明
File source	path：必填，文件路径 maxFilesPerTrigger：每次trigger最大文件数（默认无限大） latestFirst：是否有限处理新文件（默认值: false） fileNameOnly：是否以文件名作为新文件校验，而不是使用完整路径进行判断（默认值: false）	支持	支持通配符路径，但不支持以逗号分隔的多个路径。 文件必须以原子方式放置在给定的目录中，这在大多数文件系统中可以通过文件移动操作实现。
Socket Source	host：连接的节点ip，必填 port：连接的端口，必填	不支持	-
Rate Source	rowsPerSecond：每秒产生的行数，默认值1 rampUpTime：在达到rowsPerSecond速度之前的上升时间 numPartitions：生成数据行的并行度	支持	-
Kafka Source	参见**https://spark.apache.org/docs/3.1.1/structured-streaming-kafka-integration.html**	支持	-

3. Sink的容错性支持列表

Sinks	支持的output模式	支持Options	容错性	说明
File Sink	Append	Path：必须指定 指定的文件格式，参见DataFrameWriter中的相关接口	exactly-once	支持写入分区表，按时间分区用处较大

Sinks	支持的output模式	支持Options	容错性	说明
Kafka Sink	Append, Update, Complete	参见：https://spark.apache.org/docs/3.1.1/structured-streaming-kafka-integration.html	at-least-once	参见https://spark.apache.org/docs/3.1.1/structured-streaming-kafka-integration.html
Foreach Sink	Append, Update, Complete	None	依赖于ForeachWriter实现	参见https://spark.apache.org/docs/3.1.1/structured-streaming-programming-guide.html#using-foreach
ForeachBatch Sink	Append, Update, Complete	None	依赖于算子实现	参见https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#using-foreach-and-foreachbatch
Console Sink	Append, Update, Complete	numRows：每轮打印的行数，默认20 truncate：输出太长时是否清空，默认true	不支持容错	-
Memory Sink	Append, Complete	None	不支持容错，在complete模式下，重启query会重建整个表	-

28.7.3 如何添加自定义代码的依赖包

问题

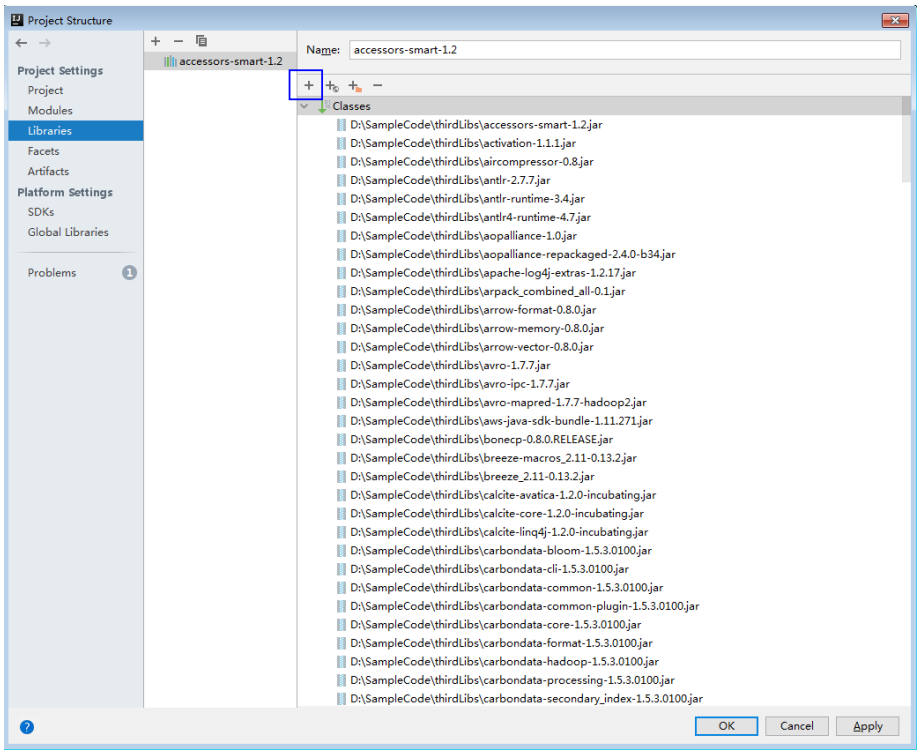
用户在开发Spark程序时，会添加样例程序外的自定义依赖包。针对自定义代码的依赖包，如何使用IDEA添加到工程中?

回答

步骤1 在IDEA主页面，选择“File > Project Structures...”进入“Project Structure”页面。

步骤2 选择“Libraries”页签，然后在如下页面，单击“+”，添加本地的依赖包。

图 28-41 添加依赖包

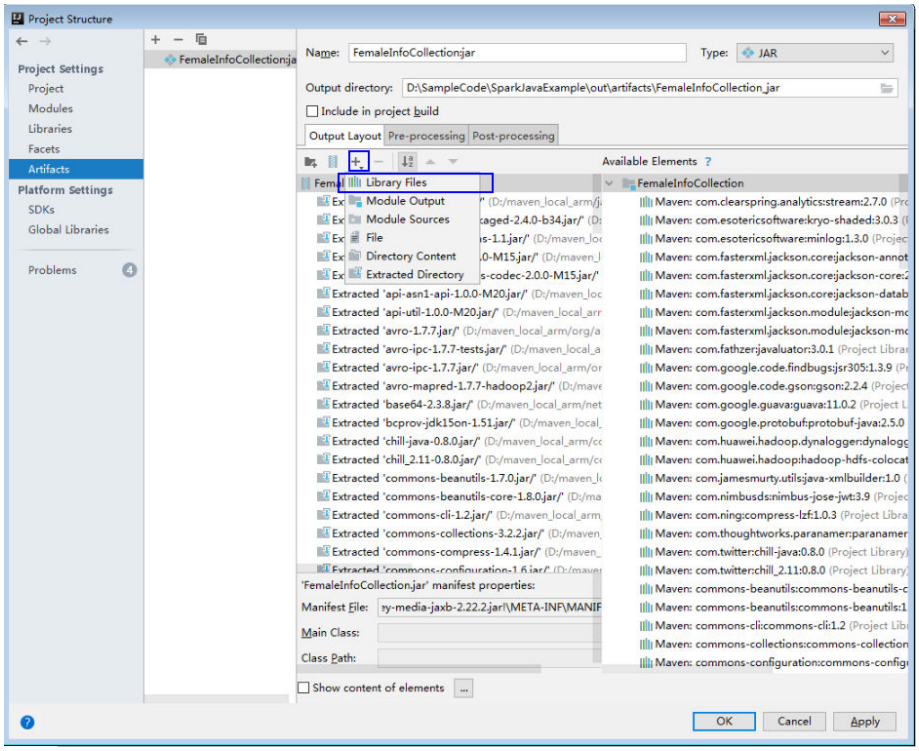


步骤3 单击“Apply”加载依赖包，然后单击“OK”完成配置。

步骤4 由于运行环境不存在用户自定义的依赖包，您还需要在编包时添加此依赖包。以便生成的jar包已包含自定义的依赖包，确保Spark程序能正常运行。

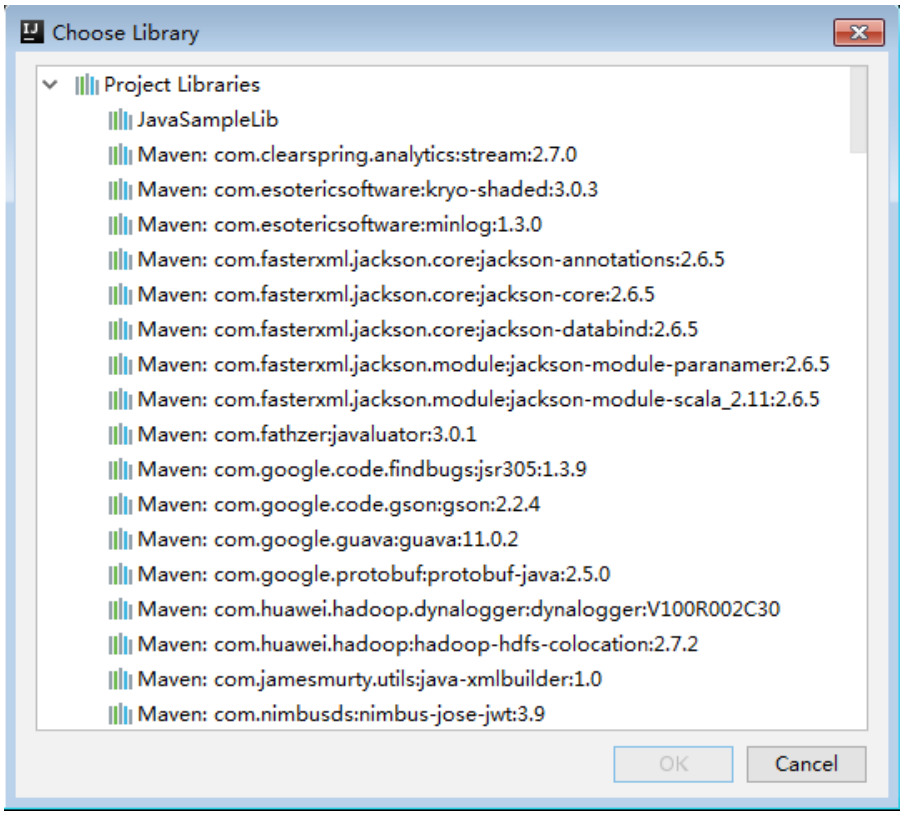
1. 在“Project Structure”页面，选择“Artifacts”页签。
2. 在右侧窗口中单击“+”，选择“Library Files”添加依赖包。

图 28-42 添加 Library Files



3. 选择需要添加的依赖包，然后单击“OK”。

图 28-43 Choose Libraries



4. 单击“Apply”加载依赖包，然后单击“OK”完成配置。

----结束

28.7.4 如何处理自动加载的依赖包

问题

在使用IDEA导入工程前，如果IDEA工具中已经进行过Maven配置时，会导致工具自动加载Maven配置中的依赖包。当自动加载的依赖包与应用程序不配套时，导致工程Build失败。如何处理自动加载的依赖包?

回答

建议在导入工程后，手动删除自动加载的依赖。步骤如下：

1. 在IDEA工具中，选择“File > Project Structures...”,。
2. 选择“Libraries”，选中自动导入的依赖包，右键选择“Delete”。

28.7.5 运行 SparkStreamingKafka 样例工程时报“类不存在”问题

问题

通过spark-submit脚本提交KafkaWordCount（org.apache.spark.examples.streaming.KafkaWordCount）任务时，日志中报Kafka相关的类不存在的错误。KafkaWordCount样例为Spark开源社区提供的。

回答

Spark部署时，如下jar包存放在客户端的“${SPARK_HOME}/jars/streamingClient010”目录以及服务端的“${BIGDATA_HOME}/FusionInsight_Spark2x_8.1.0.1/install/FusionInsight-Spark2x-3.1.1/spark/jars/streamingClient010”目录：

- kafka-clients-xxx.jar
- kafka_2.12-xxx.jar
- spark-streaming-kafka-0-10_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar
- spark-token-provider-kafka-0-10_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar

由于“$SPARK_HOME/jars/streamingClient010/*”默认没有添加到classpath，所以需要手动配置。

在提交应用程序运行时，在命令中添加如下参数即可，详细示例可参考**在Linux环境中编包并运行Spark程序**。

--jars $SPARK_CLIENT_HOME/jars/streamingClient010/kafka-client-2.4.0.jar,$SPARK_CLIENT_HOME/jars/streamingClient010/kafka_2.12-*.jar,$SPARK_CLIENT_HOME/jars/streamingClient010/spark-streaming-kafka-0-10_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar

用户自己开发的应用程序以及样例工程都可使用上述命令提交。

但是Spark开源社区提供的KafkaWordCount等样例程序，不仅需要添加--jars参数，还需要配置其他，否则会报“ClassNotFoundException”错误，yarn-client和yarn-cluster模式下稍有不同。

- yarn-client模式下

  在除--jars参数外，在客户端“spark-defaults.conf”配置文件中，将“spark.driver.extraClassPath”参数值中添加客户端依赖包路径，如“$SPARK_HOME/jars/streamingClient010/*”。

- yarn-cluster模式下

  除--jars参数外，还需要配置其他，有三种方法任选其一即可，具体如下：

  - 在客户端spark-defaults.conf配置文件中，在“spark.yarn.cluster.driver.extraClassPath”参数值中添加服务端的依赖包路径，如“${BIGDATA_HOME}/FusionInsight_Spark2x_8.1.0.1/install/FusionInsight-Spark2x-3.1.1/spark/jars/streamingClient010/*”。
  - 将各服务端节点的“original-spark-examples_2.12-3.1.1-xxx.jar”包删除。
  - 在客户端“spark-defaults.conf”配置文件中，修改或增加配置选项“spark.driver.userClassPathFirst” = “true”。

28.7.6 由于 Kafka 配置的限制，导致 Spark Streaming 应用运行失败

问题

使用运行的Spark Streaming任务回写Kafka时，Kafka上接收不到回写的数据，且Kafka日志报错信息如下：

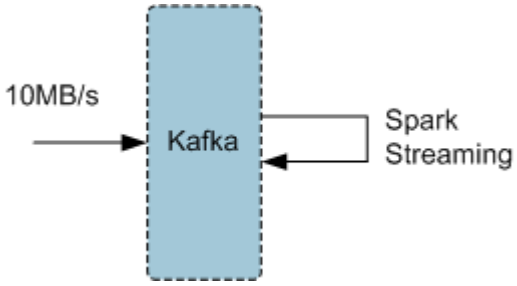
```
2016-03-02 17:46:19,017 | INFO | [kafka-network-thread-21005-1] | Closing socket connection to /
10.91.8.208 due to invalid request: Request of length
 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. | kafka.network.Processor
(Logging.scala:68)
2016-03-02 17:46:19,155 | INFO | [kafka-network-thread-21005-2] | Closing socket connection to /
10.91.8.208. | kafka.network.Processor (Logging.scala:68)
2016-03-02 17:46:19,270 | INFO | [kafka-network-thread-21005-0] | Closing socket connection to /
10.91.8.208 due to invalid request:
Request of length 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. |
kafka.network.Processor (Logging.scala:68)
2016-03-02 17:46:19,513 | INFO | [kafka-network-thread-21005-1] | Closing socket connection to /
10.91.8.208 due to invalid request:
Request of length 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. |
kafka.network.Processor (Logging.scala:68)
2016-03-02 17:46:19,763 | INFO | [kafka-network-thread-21005-2] | Closing socket connection to /
10.91.8.208 due to invalid request:
Request of length 122371301 is not valid, it is larger than the maximum size of 104857600 bytes. |
kafka.network.Processor (Logging.scala:68)
53393 [main] INFO  org.apache.hadoop.mapreduce.Job  - Counters: 50
```

回答

如下图所示，Spark Streaming应用中定义的逻辑为，从Kafka中读取数据，执行对应处理之后，然后将结果数据回写至Kafka中。

例如：Spark Streming中定义了批次时间，如果数据传入Kafka的速率为10MB/s，而Spark Streaming中定义了每60s一个批次，回写数据总共为600MB。而Kafka中定义了接收数据的阈值大小为500MB。那么此时回写数据已超出阈值。此时，会出现上述错误。

图 28-44 应用场景



解决措施：

方式一：推荐优化Spark Streaming应用程序中定义的批次时间，降低批次时间，可避免超过Kafka定义的阈值。一般建议以5-10秒/次为宜。

方式二：将Kafka的阈值调大，建议在FusionInsight Manager中的Kafka服务进行参数设置，将socket.request.max.bytes参数值根据应用场景，适当调整。

28.7.7 执行 Spark Core 应用，尝试收集大量数据到 Driver 端，当 Driver 端内存不足时，应用挂起不退出

问题

执行Spark Core应用，尝试收集大量数据到Driver端，当Driver端内存不足时，应用挂起不退出，日志内容如下。

```
16/04/19 15:56:22 ERROR Utils: Uncaught exception in thread task-result-getter-2
java.lang.OutOfMemoryError: Java heap space
at java.lang.reflect.Array.newArray(Native Method)
at java.lang.reflect.Array.newInstance(Array.java:75)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1671)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1707)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
at org.apache.spark.serializer.JavaDeserializationStream.readObject(JavaSerializer.scala:71)
at org.apache.spark.serializer.JavaSerializerInstance.deserialize(JavaSerializer.scala:91)
at org.apache.spark.scheduler.DirectTaskResult.value(TaskResult.scala:94)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply$mcV
$sp(TaskResultGetter.scala:66)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.util.Utils$.logUncaughtExceptions(Utils.scala:1716)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3.run(TaskResultGetter.scala:56)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)
Exception in thread "task-result-getter-2" java.lang.OutOfMemoryError: Java heap space
at java.lang.reflect.Array.newArray(Native Method)
at java.lang.reflect.Array.newInstance(Array.java:75)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1671)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
```



```
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.defaultReadFields(ObjectInputStream.java:2000)
at java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:1924)
at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:1801)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1351)
at java.io.ObjectInputStream.readArray(ObjectInputStream.java:1707)
at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1345)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:371)
at org.apache.spark.serializer.JavaDeserializationStream.readObject(JavaSerializer.scala:71)
at org.apache.spark.serializer.JavaSerializerInstance.deserialize(JavaSerializer.scala:91)
at org.apache.spark.scheduler.DirectTaskResult.value(TaskResult.scala:94)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply$mcV
$sp(TaskResultGetter.scala:66)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3$$anonfun$run$1.apply(TaskResultGetter.scala:57)
at org.apache.spark.util.Utils$.logUncaughtExceptions(Utils.scala:1716)
at org.apache.spark.scheduler.TaskResultGetter$$anon$3.run(TaskResultGetter.scala:56)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)
```

回答

用户尝试收集大量数据到Driver端，如果Driver端的内存不足以存放这些数据，那么就会抛出OOM(OutOfMemory)的异常，然后Driver端一直在进行GC，尝试回收垃圾来存放返回的数据，导致应用长时间挂起。

解决措施：

如果用户需要在OOM场景下强制将应用退出，那么可以在启动Spark Core应用时，在客户端配置文件“$SPARK_HOME/conf/spark-defaults.conf”中的配置项“spark.driver.extraJavaOptions”中添加如下内容：

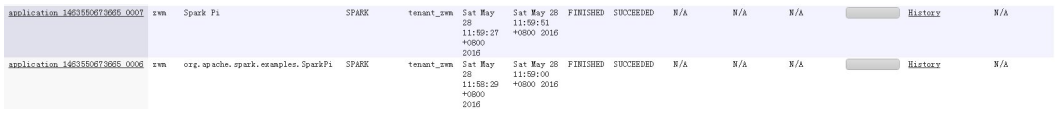
```
-XX:OnOutOfMemoryError='kill -9 %p'
```

28.7.8 Spark 应用名在使用 yarn-cluster 模式提交时不生效

问题

Spark应用名在使用yarn-cluster模式提交时不生效，在使用yarn-client模式提交时生效，如图28-45所示，第一个应用是使用yarn-client模式提交的，正确显示代码里设置的应用名Spark Pi，第二个应用是使用yarn-cluster模式提交的，设置的应用名没有生效。

图 28-45 提交应用



回答

导致这个问题的主要原因是，yarn-client和yarn-cluster模式在提交任务时setAppName的执行顺序不同导致，yarn-client中setAppName是在向yarn注册Application之前读取，yarn-cluser模式则是在向yarn注册Application之后读取，这就导致yarn-cluster模式设置的应用名不生效。

解决措施：

在spark-submit脚本提交任务时用--name设置应用名和sparkconf.setAppName(appname)里面的应用名一样。

比如代码里设置的应用名为Spark Pi，用yarn-cluster模式提交应用时可以这样设置，在--name后面添加应用名，执行的命令如下：

***./spark-submit --class** org.apache.spark.examples.SparkPi **--master yarn --deploy-mode cluster --name** SparkPi jars/original-spark-examples*.jar 10*

28.7.9 如何使用 IDEA 远程调试

问题

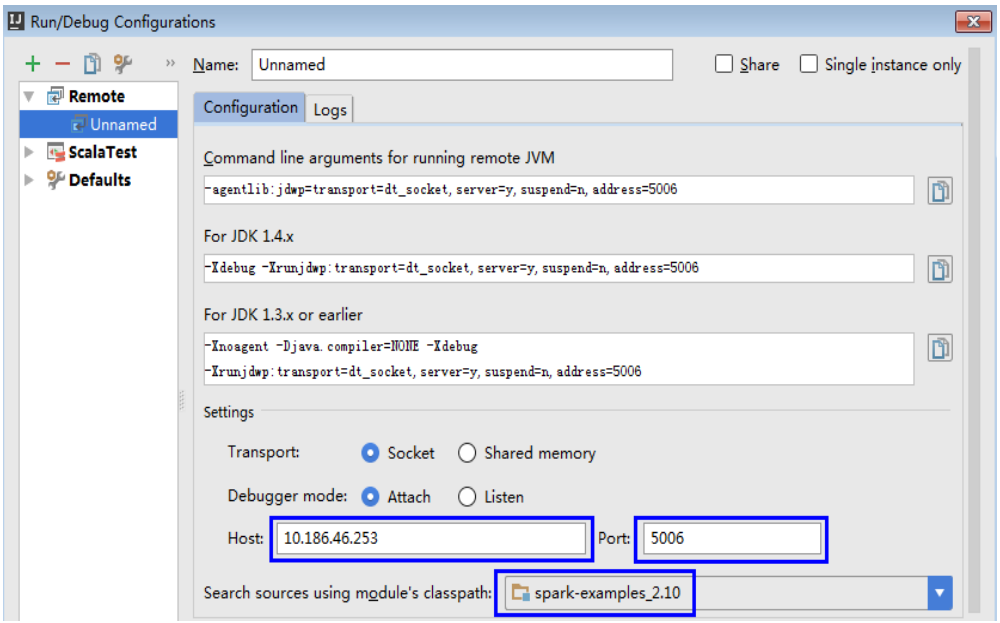
在Spark二次开发中如何使用IDEA远程调试?

回答

以调试SparkPi程序为例，演示如何进行IDEA的远程调试：

1. 打开工程，在菜单栏中选择“Run > Edit Configurations”。
2. 在弹出的配置窗口中用鼠标左键单击左上角的+号，在下拉菜单中选择Remote，如图28-46所示。

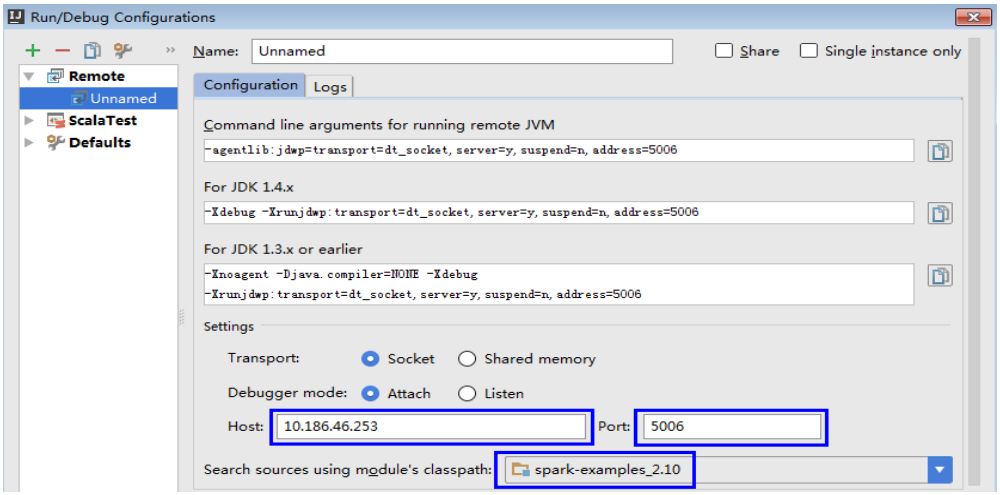
图 28-46 选择 Remote



3. 选择对应要调试的源码模块路径，并配置远端调试参数Host和Port，如图28-47所示。

   其中Host为Spark运行机器IP地址，Port为调试的端口号（确保该端口在运行机器上没被占用）。

图 28-47 配置参数



说明

当改变Port端口号时，For JDK1.4.x对应的调试命令也跟着改变，比如Port设置为5006，对应调试命令会变更为-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006，这个调试命令在启动Spark程序时要用到。

4. 执行以下命令，远端启动Spark运行SparkPi。

   ./spark-submit --master yarn-client --driver-java-options *"-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006" --class org.apache.spark.examples.SparkPi /opt/FI-Client/Spark2x/spark/examples/jars/spark-examples_2.12-3.1.1-xxx.jar*

   用户调试时需要把--class和jar包换成自己的程序，-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006需要换成3获取到的For JDK1.4.x对应的调试命令。

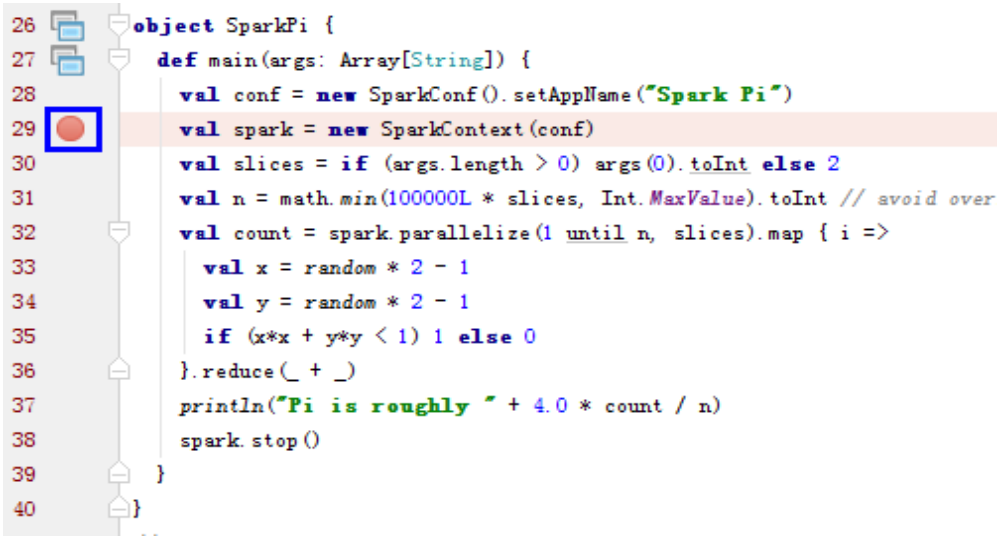
图 28-48 Spark 运行命令

```
[root@host2 bin]#./spark-submit --master yarn-client --driver-java-options "-Xde
bug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006" --class org.a
pache.spark.examples.SparkPi /opt/client/Spark2x/spark/examples/jars/spark-examp
les_2.11-2.1.0.jar
Listening for transport dt_socket at address: 5006
```

5. 设置调试断点。

   在IDEA代码编辑窗口左侧空白处单击鼠标左键设置相应代码行断点，如图28-49所示，在SparkPi.scala的29行设置断点。

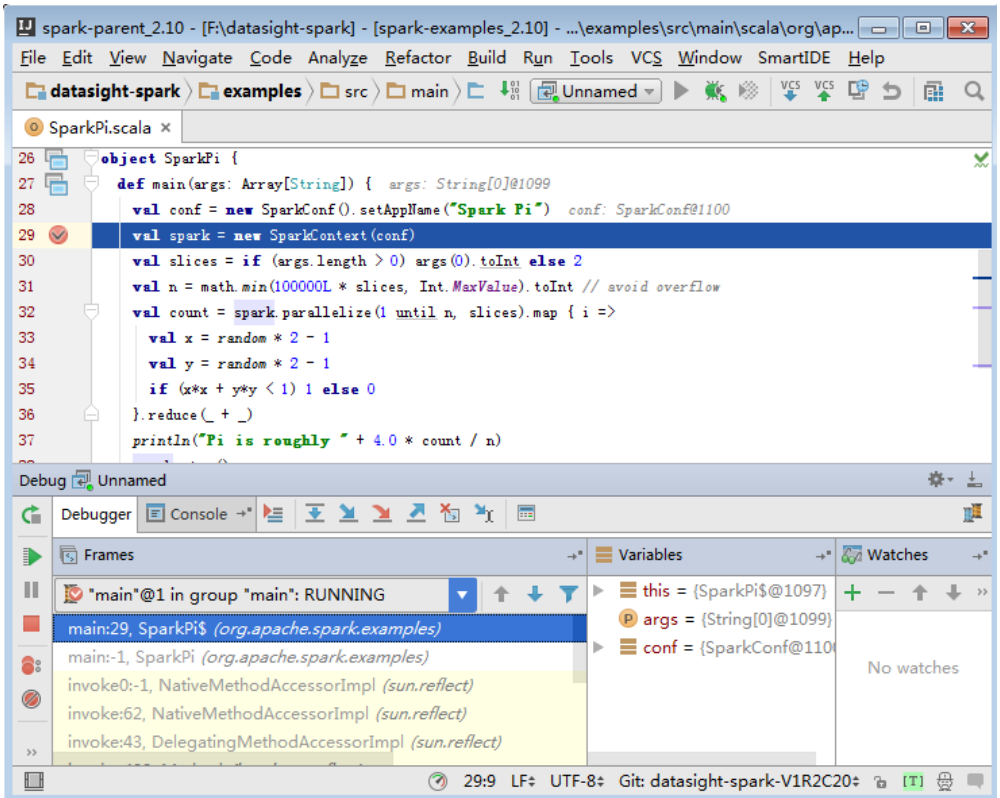
图 28-49 设置断点



6. 启动调试。

   在IDEA菜单栏中选择“Run > Debug 'Unnamed'”开启调试窗口，接着开始SparkPi的调试，比如单步调试、查看调用栈、跟踪变量值等，如图28-50所示。

图 28-50 调试



28.7.10 如何采用 Java 命令提交 Spark 应用

问题

除了spark-submit命令提交应用外，如何采用Java命令提交Spark应用?

回答

您可以通过org.apache.spark.launcher.SparkLauncher类采用java命令方式提交Spark应用。详细步骤如下：

步骤1 定义org.apache.spark.launcher.SparkLauncher类。默认提供了SparkLauncherJavaExample和SparkLauncherScalaExample示例，您需要根据实际业务应用程序修改示例代码中的传入参数。

- 如果您使用Java语言开发程序，您可以参考如下示例，编写SparkLauncher类。

```
public static void main(String[] args) throws Exception {
    System.out.println("com.huawei.bigdata.spark.examples.SparkLauncherExample <mode> <jarParh> <app_main_class> <appArgs>");
    SparkLauncher launcher = new SparkLauncher();
    launcher.setMaster(args[0])
        .setAppResource(args[1]) // Specify user app jar path
        .setMainClass(args[2]);
    if (args.length > 3) {
        String[] list = new String[args.length - 3];
        for (int i = 3; i < args.length; i++) {
            list[i-3] = args[i];
        }
        // Set app args
        launcher.addAppArgs(list);
    }

    // Launch the app
    Process process = launcher.launch();
    // Get Spark driver log
    new Thread(new ISRRunnable(process.getErrorStream())).start();
    int exitCode = process.waitFor();
    System.out.println("Finished! Exit code is " + exitCode);
}
```

- 如果您使用Scala语言开发程序，您可以参考如下示例，编写SparkLauncher类。

```
def main(args: Array[String]) {
  println(s"com.huawei.bigdata.spark.examples.SparkLauncherExample <mode> <jarParh> <app_main_class> <appArgs>")
  val launcher = new SparkLauncher()
  launcher.setMaster(args(0))
    .setAppResource(args(1)) // Specify user app jar path
    .setMainClass(args(2))
    if (args.drop(3).length > 0) {
      // Set app args
      launcher.addAppArgs(args.drop(3): _*)
    }


  // Launch the app
  val process = launcher.launch()
  // Get Spark driver log
  new Thread(new ISRRunnable(process.getErrorStream)).start()
  val exitCode = process.waitFor()
  println(s"Finished! Exit code is $exitCode")
}
```

步骤2 根据业务逻辑，开发对应的Spark应用程序。并设置用户编写的Spark应用程序的主类等常数。不同场景的示例请参考**开发Spark应用**。

- 如果您使用的安全模式，建议按照安全要求，准备安全认证代码、业务应用代码及其相关配置。

  说明

  yarn-cluster模式中不支持在Spark工程中添加安全认证。因为需要在应用启动前已完成安全认证。所以用户需要在Spark应用之外添加安全认证代码或使用命令行进行认证。由于提供的示例代码默认提供安全认证代码，请在yarn-cluster模式下时，修改对应安全代码后再运行应用。

- 如果您使用的是普通模式，准备业务应用代码及其相关配置即可。

步骤3 调用org.apache.spark.launcher.SparkLauncher.launch()方法，将用户的应用程序提交。

1. 将SparkLauncher程序和用户应用程序分别生成Jar包，并上传至运行此应用的Spark节点中。生成Jar包的操作步骤请参见**在Linux环境中编包并运行Spark程序**章节。
   - SparkLauncher程序的编译依赖包为spark-launcher_2.12-3.1.1-hw-ei-311001-SNAPSHOT.jar，请从软件发布包中Software文件夹下“FusionInsight_Spark2x_8.1.0.1.tar.gz”压缩包中的“jars”目录中获取。
   - 用户应用程序的编译依赖包根据代码不同而不同，需用户根据自己编写的代码进行加载。
2. 将运行程序的依赖Jar包上传至需要运行此应用的节点中，例如“$SPARK_HOME/jars”路径。

   用户需要将SparkLauncher类的运行依赖包和应用程序运行依赖包上传至客户端的jars路径。文档中提供的示例代码，其运行依赖包在客户端jars中已存在。

   说明

   Spark Launcher的方式依赖Spark客户端，即运行程序的节点必须已安装Spark客户端，且客户端可用。运行过程中依赖客户端已配置好的环境变量、运行依赖包和配置文件，
3. 在Spark应用程序运行节点，执行如下命令使用Spark Launcher方式提交。之后，可通过Spark WebUI查看运行情况，或通过获取指定文件查看运行结果，可参见**在Linux环境中查看Spark程序调测结果**。

   java -cp $SPARK_HOME/conf:$SPARK_HOME/jars/*:SparkLauncherExample.jar com.huawei.bigdata.spark.examples.SparkLauncherExample yarn-client /opt/female/FemaleInfoCollection.jar com.huawei.bigdata.spark.examples.FemaleInfoCollection <inputPath>

----结束

28.7.11 使用 IBM JDK 产生异常，提示“Problem performing GSS wrap”信息

问题

使用IBM JDK产生异常，提示“Problem performing GSS wrap”信息

回答

问题原因：

在IBM JDK下建立的JDBC connection时间超过登录用户的认证超时时间（默认一天），导致认证失败。

说明

IBM JDK的机制跟Oracle JDK的机制不同，IBM JDK在认证登录后的使用过程中做了时间检查却没有检测外部的时间更新，导致即使显式调用relogin也无法得到刷新。

解决措施：

通常情况下，在发现JDBC connection不可用的时候，可以关闭该connection，重新创建一个connection继续执行。

28.7.12 Structured Streaming 的 cluster 模式，在数据处理过程中终止 ApplicationManager，应用失败

问题

Structured Streaming的cluster模式，在数据处理过程中终止ApplicationManager，执行应用时显示如下异常。

```
2017-05-09 20:46:02,393 | INFO  | main |
 client token: Token { kind: YARN_CLIENT_TOKEN, service:  }
 diagnostics: User class threw exception: org.apache.spark.sql.AnalysisException: This query does not support recovering from checkpoint location. Delete hdfs://hacluster/structuredtest/checkpoint/offsets to start over.;
 ApplicationMaster host: 10.96.101.170
 ApplicationMaster RPC port: 0
 queue: default
 start time: 1494333891969
 final status: FAILED
 tracking URL: https://9-96-101-191:8090/proxy/application_1493689105146_0052/
 user: spark2x | org.apache.spark.internal.Logging$class.logInfo(Logging.scala:54)
Exception in thread "main" org.apache.spark.SparkException: Application application_1493689105146_0052 finished with failed status
```

回答

原因分析：显示该异常是因为“recoverFromCheckpointLocation”的值判定为false，但却配置了checkpoint目录。

参数“recoverFromCheckpointLocation”的值为代码中“outputMode == OutputMode.Complete()”语句的判断结果（outputMode的默认输出方式为“append”）。

处理方法：编写应用时，用户可以根据具体情况修改数据的输出方式。

将输出方式修改为“complete”，“recoverFromCheckpointLocation”的值会判定为true。此时配置了checkpoint目录时就不会显示异常。

28.7.13 从 checkpoint 恢复 spark 应用的限制

问题

Spark应用可以从checkpoint恢复，用于从上次任务中断处继续往下执行，以保证数据不丢失。但是，在某些情况下，从checkpoint恢复应用会失败。

回答

由于checkpoint中包含了spark应用的对象序列化信息、task执行状态信息、配置信息等，因此，当存在以下问题时，从checkpoint恢复spark应用将会失败。

1. 业务代码变更且变更类未明确指定SerialVersionUID。
2. spark内部类变更，且变更类未明确指定SerialVersionUID。

另外，由于checkpoint保存了部分配置项，因此可能导致业务修改了部分配置项后，从checkpoint恢复时，配置项依然保持为旧值的情况。当前只有以下部分配置会在从checkpoint恢复时重新加载。

```
"spark.yarn.app.id",
 "spark.yarn.app.attemptId",
 "spark.driver.host",
 "spark.driver.bindAddress",
 "spark.driver.port",
 "spark.master",
 "spark.yarn.jars",
 "spark.yarn.keytab",
 "spark.yarn.principal",
 "spark.yarn.credentials.file",
 "spark.yarn.credentials.renewalTime",
 "spark.yarn.credentials.updateTime",
 "spark.ui.filters",
 "spark.mesos.driver.frameworkId",
 "spark.yarn.jars"
```

解决方法

手动删除checkpoint目录，重启业务程序。

说明

删除文件为高危操作，在执行操作前请务必确认对应文件是否不再需要。

28.7.14 第三方 jar 包跨平台（x86、TaiShan）支持

问题

用户自己写的jar包（例如自定义udf包）区分x86和TaiShan版本，如何让Spark2x支持其正常运行。

回答

第三方jar包（例如自定义udf）区分x86和TaiShan版本时，混合使用方案：

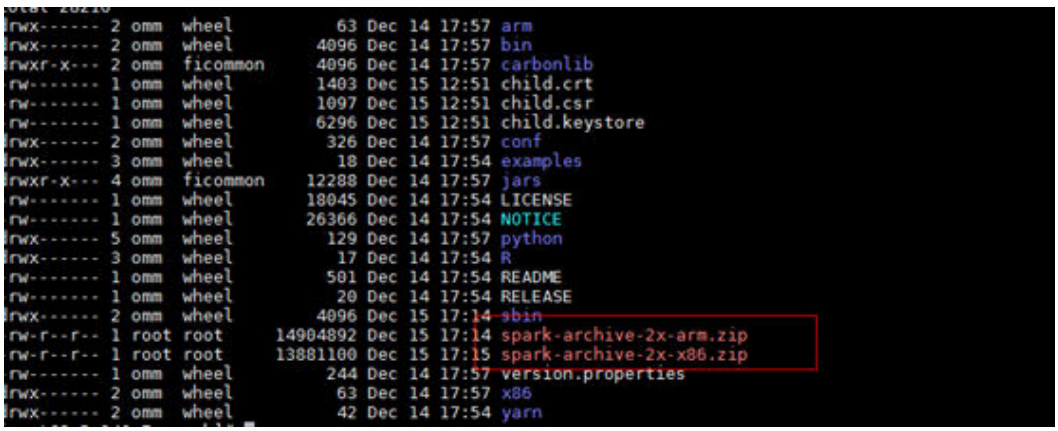
步骤1 进入到服务端Spark2x SparkResource的安装目录（集群安装时，SparkResource可能会安装在多个节点上，登录任意一个SparkResource节点，进入到SparkResource的安装目录）。

步骤2 准备好自己的jar包，例如xx.jar的x86版本和TaiShan版本。将x86版本和TaiShan版本的xx.jar分别复制到当前目录的x86文件夹和TaiShan文件夹里面。

步骤3 在当前目录下执行以下命令将jar包打包：

zip -qDj spark-archive-2x-x86.zip x86/*

zip -qDj spark-archive-2x-arm.zip arm/*



步骤4 执行以下命令查看hdfs上的spark2x依赖的jar包：

hdfs dfs -ls /user/spark2x/jars/8.1.0.1

说明

8.1.0.1是版本号，不同版本不同。

执行以下命令移动hdfs上旧的jar包文件到其他目录，例如移动到“tmp”目录。

hdfs dfs -mv /user/spark2x/jars/8.1.0.1/spark-archive-2x-arm.zip /tmp

hdfs dfs -mv /user/spark2x/jars/8.1.0.1/spark-archive-2x-x86.zip /tmp

步骤5 上传**步骤3**中打包的spark-archive-2x-arm.zip和spark-archive-2x-x86.zip到hdfs的/user/spark2x/jars/8.1.0.1目录下，上传命令如下：

hdfs dfs -put spark-archive-2x-arm.zip /user/spark2x/jars/8.1.0.1/

hdfs dfs -put spark-archive-2x-x86.zip /user/spark2x/jars/8.1.0.1/

上传完毕后删除本地的spark-archive-2x-arm.zip，spark-archive-2x-x86.zip文件。

步骤6 对其他的sparkResource安装节点执行**步骤1**~**步骤2**。

步骤7 进入webUI重启spark2x的jdbcServer实例。

步骤8 重启后，需要更新客户端配置。按照客户端所在的机器类型（x86、TaiShan）复制xx.jar的相应版本到客户端的Spark2x安装目录“${install_home}/Spark2x/spark/jars”文件夹中。${install_home}是用户的客户端安装路径，用户需要填写实际的安装目录；若本地的安装目录为“/opt/hadoopclient”，那么就复制相应版本xx.jar到“/opt/hadoopclient/Spark2x/spark/jars”文件夹里。

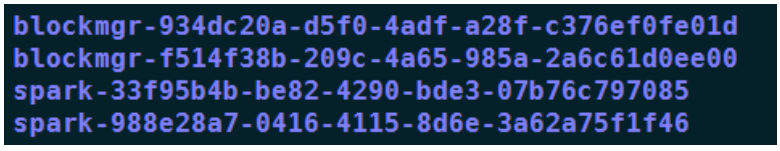
----结束

28.7.15 在客户端安装节点的/tmp 目录下残留了很多 blockmgr-开头和 spark-开头的目录

问题

系统长时间运行后，在客户端安装节点的/tmp目录下，发现残留了很多blockmgr-开头和spark-开头的目录。

图 28-51 残留目录样例



回答

Spark任务在运行过程中，driver会创建一个spark-开头的本地临时目录，用于存放业务jar包，配置文件等，同时在本地创建一个blockmgr-开头的本地临时目录，用于存放block data。此两个目录会在Spark应用运行结束时自动删除。

此两个目录的存放路径优先通过SPARK_LOCAL_DIRS环境变量指定，若不存在该环境变量，则设置为spark.local.dir的值，若此配置还不存在，则使用java.io.tmpdir的值。客户端默认配置中spark.local.dir被设置为/tmp，因此默认使用系统/tmp目录。

但存在一些特殊情况，如driver进程未正常退出，比如被kill -9命令结束进程，或者Java虚拟机直接崩溃等场景，导致driver的退出流程未正常执行，则可能导致该部分目录无法被正常清理，残留在系统中。

当前只有yarn-client模式和local模式的driver进程会产生上述问题，在yarn-cluster模式中，已将container内进程的临时目录设置为container临时目录，当container退出时，由container自动清理该目录，因此yarn-cluster模式不存在此问题。

解决措施

可在Linux下设置/tmp临时目录自动清理，或修改客户端中spark-defaults.conf配置文件的spark.local.dir配置项的值，将临时目录指定到特定的目录，再对该目录单独设置清理机制。

28.7.16 ARM 环境 python pipeline 运行报 139 错误码规避方案

问题

在TaiShan服务器上，使用python插件的pipeline报错显示139错误。具体报错如下：

```
subprocess exited with status 139
```

回答

该python程序既使用了libcrypto.so，也使用了libssl.so。而一旦LD_LIBRARY_PATH添加了hadoop的native库目录，则使用的就是hadoop native库中的libcrypto.so，而使用系统自带的libssl.so（因为hadoop native目录没有带该包）。由于这两个库版本不匹配，导致了python文件运行时出现段错误。

解决方案

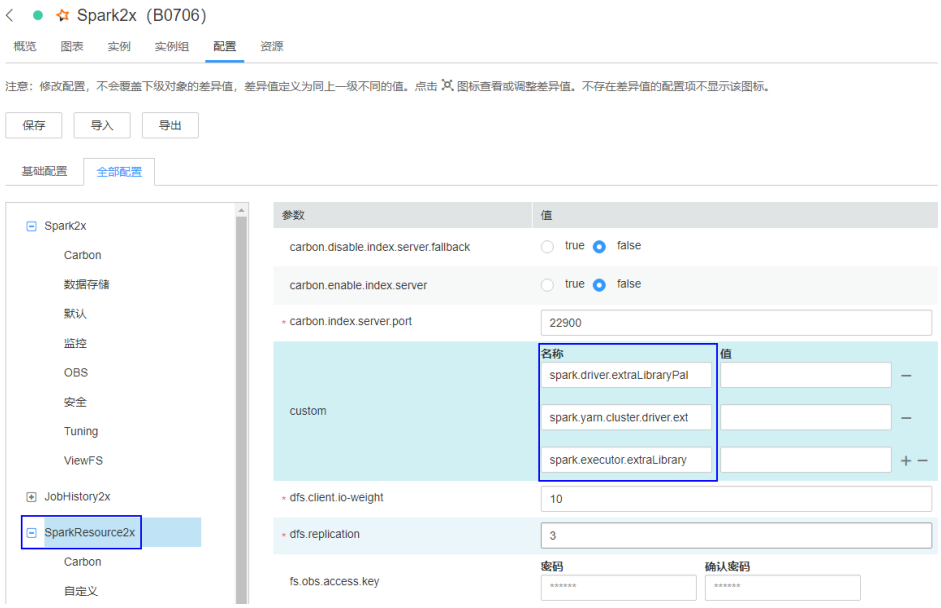
方案一：

修改Spark2x客户端conf目录下spark-default.conf文件，清空（直接赋值为空）配置项spark.driver.extraLibraryPath、spark.yarn.cluster.driver.extraLibraryPath和spark.executor.extraLibraryPath的值。

方案二：

在FusionInsight Mnanager中Spark2x界面中修改上述三个参数然后重启Spark2x实例之后重新下载客户端，具体步骤如下：

1. 登录FusionInsight Mnanager界面，选择“集群 > *待操作集群的名称* > 服务 > Spark2x > 配置 > 全部配置”，搜索参数spark.driver.extraLibraryPath和spark.executor.extraLibraryPath，并清空其参数值。
2. 在“全部配置”中选择“ SparkResource2x”。 在SparkResource2x中的custom中添加方案一中的三个参数，如下图所示：



3. 单击“保存”，完成后重启过期的spark2x实例，并重新下载安装客户端。

28.7.17 Structured Streaming 任务提交方式变更

问题

用户提交结构流任务时，通常需要通过**--jars**命令指定kafka相关jar包的路径，例如**--jars /kafkadir/kafka-clients-x.x.x.jar,/kafkadir/kafka_2.11-x.x.x.jar**。当前版本用户除了这一步外还需要额外的配置项，否则会报class not found异常。

回答

当前版本的Spark内核直接依赖于kafka相关的jar包（结构流使用），因此提交结构流任务时，需要把Kafka相关jar包加入到结构流任务driver端的库目录下，确保driver能够正常加载kafka包。

解决方案

1. 提交yarn-client模式的结构流任务时需要额外如下操作：

   将Spark客户端目录下spark-default.conf文件中的spark.driver.extraClassPath配置复制出来，并将Kafka相关jar包路径追加到该配置项之后，提交结构流任务时需要通过--conf将该配置项给加上。例如：kafka相关jar包路径为“/kafkadir”，提交任务需要增加--conf spark.driver.extraClassPath=/opt/client/Spark2x/spark/conf/:/opt/client/Spark2x/spark/jars/*:/opt/client/Spark2x/spark/x86/*:/kafkadir/*。

2. 提交yarn-cluster模式的结构流任务时需要额外如下操作：

   将Spark客户端目录下spark-default.conf文件中的spark.yarn.cluster.driver.extraClassPath配置给复制出来，并将Kafka相关jar包相对路径追加到该配置项之后，提交结构流任务时需要通过--conf 将该配置项给加上。例如：kafka相关包为kafka-clients-x.x.x.jar，kafka_2.11-x.x.x.jar，提交任务需要增加--conf spark.yarn.cluster.driver.extraClassPath=/home/huawei/Bigdata/common/runtime/security:./kafka-clients-x.x.x.jar:./kafka_2.11-x.x.x.jar。

3. 当前版本Spark结构流部分不再支持kafka2.x之前的版本，对于升级场景请继续使用旧的客户端。

28.7.18 常见 jar 包冲突处理方式

问题现象

Spark能对接很多的第三方工具，因此在使用过程中经常会依赖一堆的三方包。而有一些包MRS已经自带，这样就有可能造成代码使用的jar包版本和集群自带的jar包版本不一致，在使用过程中就有可能出现jar包冲突的情况。

常见的jar包冲突报错有：

1、报错类找不到：java.lang.NoClassDefFoundError

2、报错方法找不到：java.lang.NoSuchMethodError

原因分析

以自定义UDF为例：

```
2021-02-08 10:51:04,249 | INFO  | main | Total input files to process : 2 | org.apache.hadoop.mapred.FileInputFormat.listStatus(FileInputFormat.java:256)
Exception in thread "main" java.lang.NoClassDefFoundError: com/huawei/udf/HelloUDF
        at com.huawei.SparkWordCount$.main(SparkWordCount.scala:32)
        at com.huawei.SparkWordCount.main(SparkWordCount.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runMain(SparkSubmit.scala:813)
        at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:187)
        at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)
        at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)
        at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
Caused by: java.lang.ClassNotFoundException: com.huawei.udf.HelloUDF
        at java.net.URLClassLoader.findClass(URLClassLoader.java:382)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        ... 11 more
```

报错信息显示是找不到类。

1. 首先需要确认的是这个类属于的jar包是否在jvm的classpath里面， spark自带的jar都在“*spark客户端目录*/jars/”。
2. 确认是否存在多个jar包拥有这个类。
3. 如果是其他依赖包，可能是没有使用--jars添加到任务里面。
4. 如果是已经添加到任务里面，但是依旧没有取到，可能是因为配置文件的driver或者executor的classpath配置不正确，可以查看日志确认是否加载到环境。
5. 另外可能报错是类初始化失败导致后面使用这个类的时候出现上述报错，需要确认是否在之前就有初始化失败或者其他报错的情况发生。

```
Exception in thread "main" java.lang.NoSuchMethodError: com.huawei.udf.HelloUDF.evaluate(Ljava/lang/String;)Ljava/lang/String;
        at com.huawei.SparkWordCount$.main(SparkWordCount.scala:32)
        at com.huawei.SparkWordCount.main(SparkWordCount.scala)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:498)
        at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$runMain(SparkSubmit.scala:813)
        at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:187)
        at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)
        at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)
        at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
```

报错信息显示找不到方法。

1. 确认这个方法对应的类所在的jar包是否加载到jvm的classpath里面，spark自带的类都在“*spark客户端目录*/jars/”。
2. 确认是否有多个jar包包含这个类（尤其注意相同工具的不同版本）。
3. 如果报错是Hadoop相关的包，有可能是因为使用的Hadoop版本不一致导致部分方法已经更改。
4. 如果报错的是三方包里面的类，可能是因为Spark已经自带了相关的jar包，但是和代码中使用的版本不一致。

操作步骤

方案一：

针对jar包冲突的问题，可以确认是否不需使用三方工具的包，如果可以更改为集群相同版本的包，则修改引入的依赖版本。

说明

建议用户尽量使用MRS集群自带的依赖包。

方案二：

jar包版本修改演示

以MRS_2.1版本为例：

1. 在pom.xml文件中添加“<properties>”参数，填写变量，方便后面统一修改版本。

```
<groupId>com.huawei</groupId>
<artifactId>SparkDemo</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <spark.version>2.3.2-mrs-2.1</spark.version>
    <hbase.version>2.1.1.0101-mrs-2.1</hbase.version>
    <hadoop.version>3.1.1-mrs-2.1</hadoop.version>
</properties>
```

2. 在“dependencies”参数中设置各个jar包的版本的时候可以直接使用上述定义的参数传递。

```
<dependencies>
    <dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-core_2.11</artifactId>
        <version>${spark.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>${hadoop.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-common</artifactId>
        <version>${hbase.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-client</artifactId>
        <version>${hbase.version}</version>
    </dependency>
```

如果遇到其他三方包冲突，可以通过查找依赖关系确认是否存在相同包不同版本的情况，尽量修改成集群自带的jar包版本。

可以参考MRS样例工程自带的pom.xml文件：**获取MRS应用开发样例工程**。

3. 打印依赖树方式：

在pom.xml文件同目录下执行命令：**mvn dependency:tree**

29 Storm 开发指南（安全模式）

29.1 Storm 应用开发概述

29.1.1 Storm 应用开发简介

简介

Storm是一个分布式的、可靠的、容错的数据流处理系统。它会把工作任务委托给不同类型的组件，每个组件负责处理一项简单特定的任务。Storm的目标是提供对大数据流的实时处理，可以可靠地处理无限的数据流。

Storm有很多适用的场景：实时分析、在线机器学习、持续计算和分布式ETL等，易扩展、支持容错，可确保数据得到处理，易于构建和操控。

Storm有如下几个特点：

- 适用场景广泛
- 易扩展，可伸缩性高
- 保证无数据丢失
- 容错性好
- 多语言
- 易于构建和操控

29.1.2 Storm 应用开发常用概念

Topology

拓扑是一个计算流图。其中每个节点包含处理逻辑，而节点间的连线则表明了节点间的数据是如何流动的。

Spout

在一个Topology中产生源数据流的组件。通常情况下Spout会从外部数据源中读取数据，然后转换为Topology内部的源数据。

Bolt

在一个Topology中接受数据然后执行处理的组件。Bolt可以执行过滤、函数操作、合并、写数据库等任何操作。

Tuple

一次消息传递的基本单元。

Stream

流是一组（无穷）元素的集合，流上的每个元素都属于同一个schema；每个元素都和逻辑时间有关；即流包含了元组和时间的双重属性。流上的任何一个元素，都可以用Element<tuple,Time>的方式来表示，tuple是元组，包含了数据结构和数据内容，Time就是该数据的逻辑时间。

keytab文件

存放用户信息的密钥文件。应用程序采用此密钥文件在组件中进行API方式认证。

29.1.3 Storm 应用开发流程

本文档主要基于Java API进行Storm拓扑的开发。

开发流程中各阶段的说明如**图29-1**和**表29-1**所示：

图 29-1 拓扑开发流程

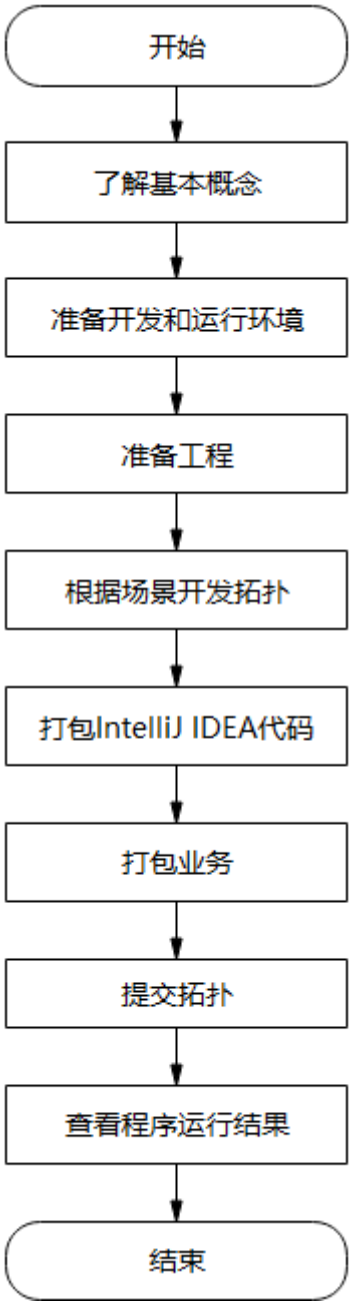


表 29-1 Storm 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Storm的基本概念，了解场景需求，拓扑等。	Storm应用开发常用概念

阶段	说明	参考文档
准备开发和运行环境	Storm的应用程序当前推荐使用Java语言进行开发。可使用IntelliJ IDEA工具。 Storm的运行环境即Storm客户端，请根据指导完成客户端的安装和配置。	**准备Storm应用开发和运行环境**
准备工程	Storm提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。	**导入并配置Storm样例工程**
根据场景开发拓扑	提供了Storm拓扑的构造和Spout/Bolt开发过程。	**开发Storm应用**
打包IntelliJ IDEA代码	Storm样例程序是在Linux环境下运行，需要将IntelliJ IDEA中的代码打包成jar包。	**打包Storm样例工程应用**
打包业务	将IntelliJ IDEA代码生成的jar包与工程依赖的jar包，合并导出可提交的source.jar。	**打包Storm业务**
提交拓扑	指导用户将开发好的程序提交运行。	**提交Storm拓扑**
查看程序运行结果	指导用户提交拓扑后查看程序运行结果。	**查看Storm应用调测结果**

29.2 准备 Storm 应用开发环境

29.2.1 准备 Storm 应用开发和运行环境

开发环境准备分为应用开发客户端和应用提交客户端；应用开发一般在Windows环境下进行；应用提交一般在Linux环境下进行。

准备开发环境

在进行二次开发时，要准备的开发和运行环境如**表29-2**所示：

表 29-2 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置，版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端： – Oracle JDK：支持1.8版本 – IBM JDK：支持1.8.5.11版本 • TaiShan客户端： – OpenJDK：支持1.8.0_272版本 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装和配置IntelliJ IDEA	用于开发Storm应用程序的工具。版本要求：JDK使用1.8版本，IntelliJ IDEA使用2019.1或其他兼容版本。 **说明** 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。
准备开发用户	参考**准备MRS应用开发用户**进行操作，准备用于应用开发的集群用户并授予相应权限。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果本地Windows开发环境和集群业务平面网络互通，可下载集群客户端到本地，获取相关调测程序所需的集群配置文件及配置网络连通后，然后直接在Windows中进行程序调测。

a. **登录FusionInsight Manager页面**，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“完整客户端”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。

例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到“FusionInsight_Cluster_1_Services_ClientConfig.tar”，继续解压该文件。解压到本地PC的“D:\FusionInsight_Cluster_1_Services_ClientConfig”目录下（路径中不能有空格）。

b. 进入客户端解压路径“FusionInsight_Cluster_1_Services_ClientConfig\Storm\config”，获取相关配置文件。

主要配置文件说明如**表29-3**所示。

表 29-3 配置文件

文件名称	作用
storm.yaml	配置Storm集群信息。
user.keytab	对于Kerberos安全认证提供用户信息。
krb5.conf	Kerberos Server配置信息。
streaming-site.xml	配置Storm详细参数。

c. 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与解压目录下“hosts”文件中所列出的各主机在网络上互通。

说明

- 当客户端所在主机不是集群中的节点时，配置客户端网络连接，可避免执行客户端命令时出现错误。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 使用Linux环境调测程序，需准备安装集群客户端的Linux节点并获取相关配置文件。

a. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。

客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

b. **登录FusionInsight Manager页面**，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig/Storm/config”路径下的所有配置文件至客户端节点，放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp Storm/config/* root@*客户端节点IP地址*:/opt/client/conf

准备MRS应用开发用户时获取的keytab文件也需放置于该目录下，主要配置文件说明如**表29-4**所示。

表 29-4 配置文件

文件名称	作用
storm.yaml	配置Storm集群信息。
user.keytab	对于Kerberos安全认证提供用户信息。
krb5.conf	Kerberos Server配置信息。
streaming-site.xml	配置Storm详细参数。

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

29.2.2 导入并配置 Storm 样例工程

背景信息

Storm客户端安装程序目录中包含了Storm开发样例工程，将工程导入到IntelliJ IDEA开始样例学习。

前提条件

确保本地PC的时间与集群的时间差要小于5分钟，若无法确定，请联系系统管理员。集群的时间可通过FusionInsight Manager页面右下角查看。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中的“src\storm-examples”目录下的“storm-examples”样例工程文件夹。

步骤2 将**准备MRS应用开发用户**时得到的keytab文件“user.keytab”和“krb5.conf”文件及**准备运行环境**时获取的配置文件放到样例工程的“storm-examples\src\main\resources”目录下。

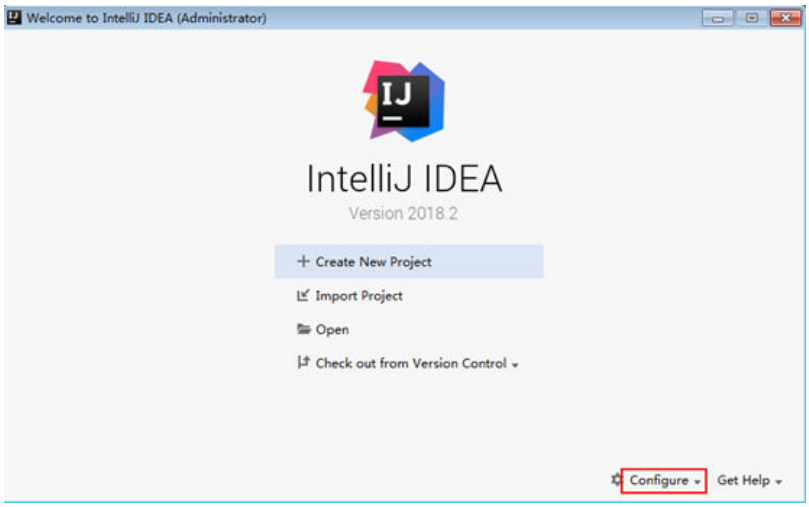
说明

- 若要在Windows或Linux中未安装客户端时提交拓扑，则需要将“streaming-site.xml”和“storm.yaml”都放入样例工程的“storm-examples\src\main\resources”目录下。
- 若要在Linux安装客户端时提交拓扑，只需要将“streaming-site.xml”放入样例工程的“storm-examples\src\main\resources”目录下即可。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

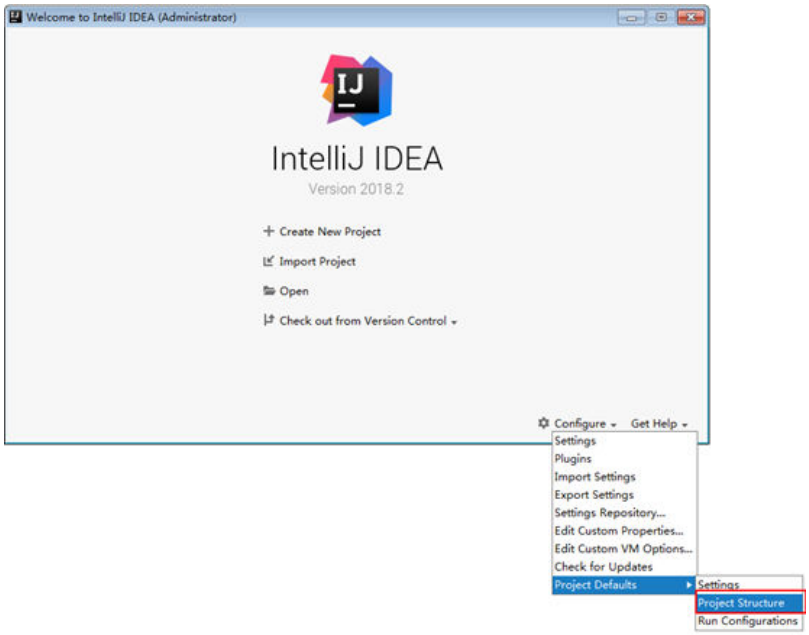
1. 打开IntelliJ IDEA，选择“Configure”。

图 29-2 Quick Start



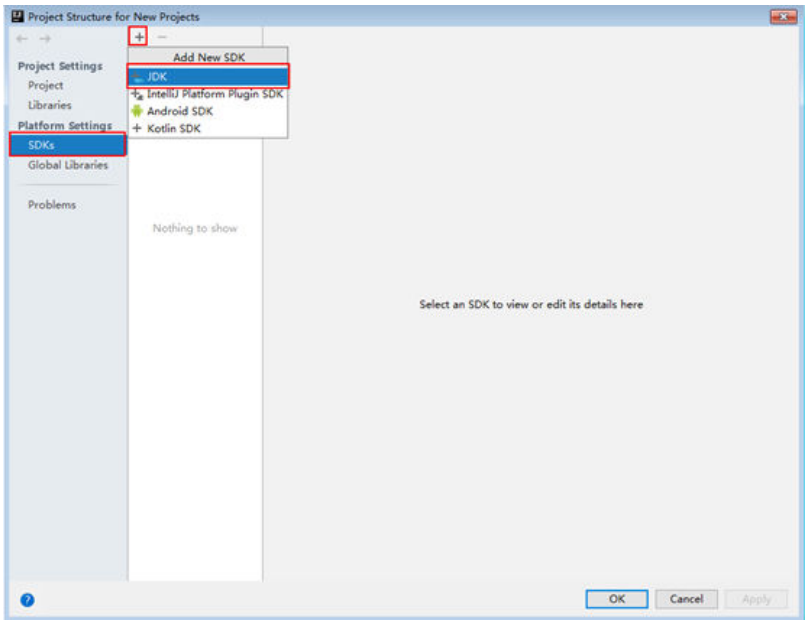
2. 在下拉框中选择“Project Defaults > Project Structure”。

图 29-3 Configure



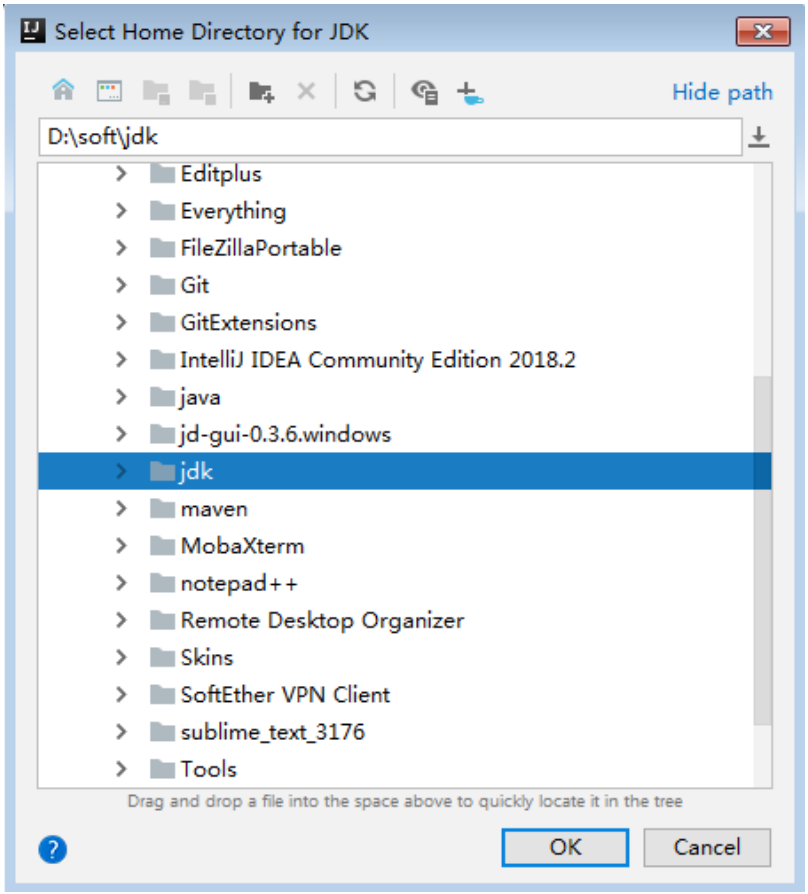
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 29-4 Project Structure for New Projects



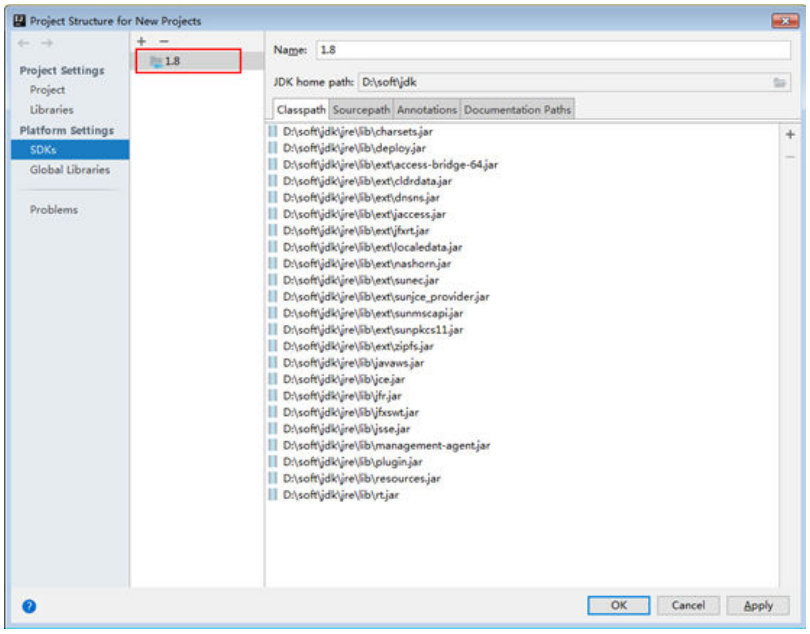
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 29-5 Select Home Directory for JDK



5. 完成JDK选择后，单击“OK”完成配置。

图 29-6 完成 JDK 配置

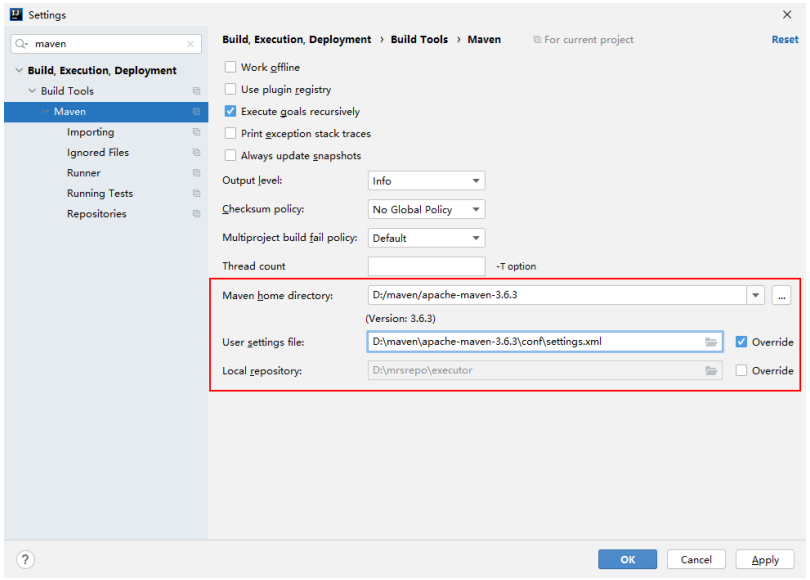


步骤4 在应用开发环境中，导入样例工程到IntelliJ IDEA开发环境。

1. 配置IntelliJ IDEA maven工程环境。

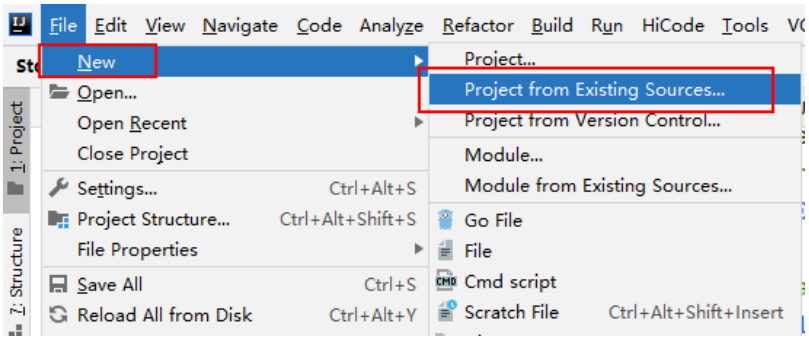
   选择“File > Settings”，搜索“maven”，配置maven工程，选择“Apply > OK”

图 29-7 配置 IntelliJ IDEA maven 工程环境



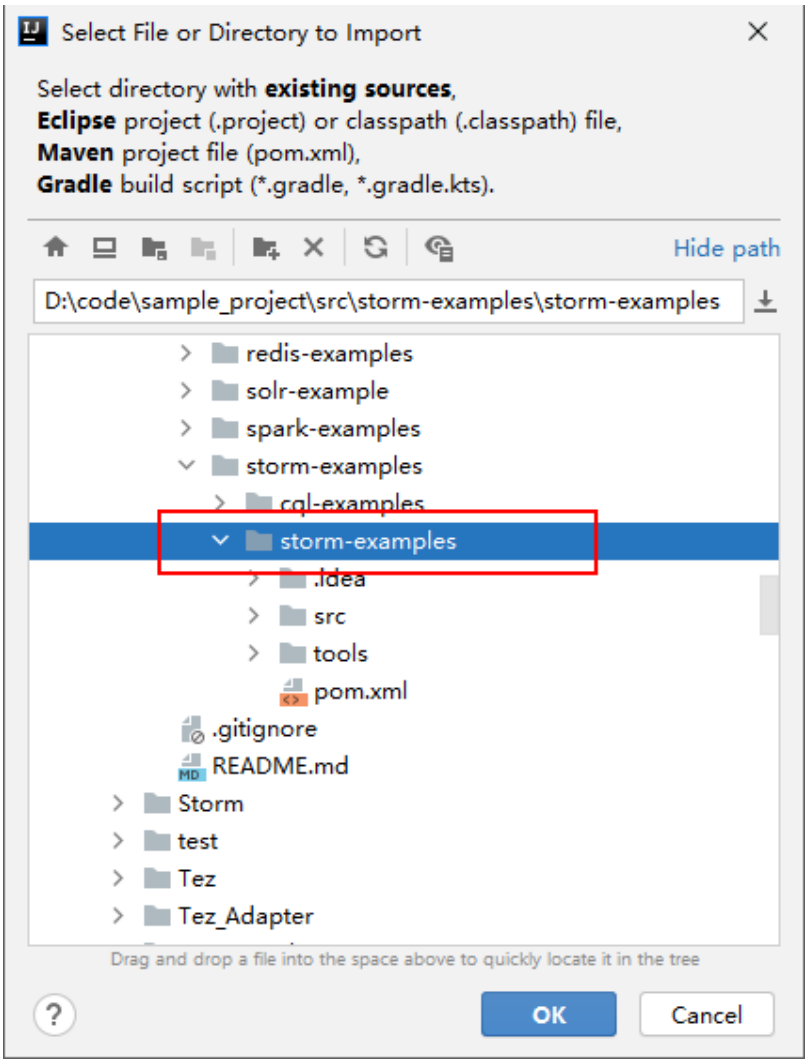
2. 选择“File > New > Project from Existing Sources…”

图 29-8 进入“Project from Existing Sources....”配置页面



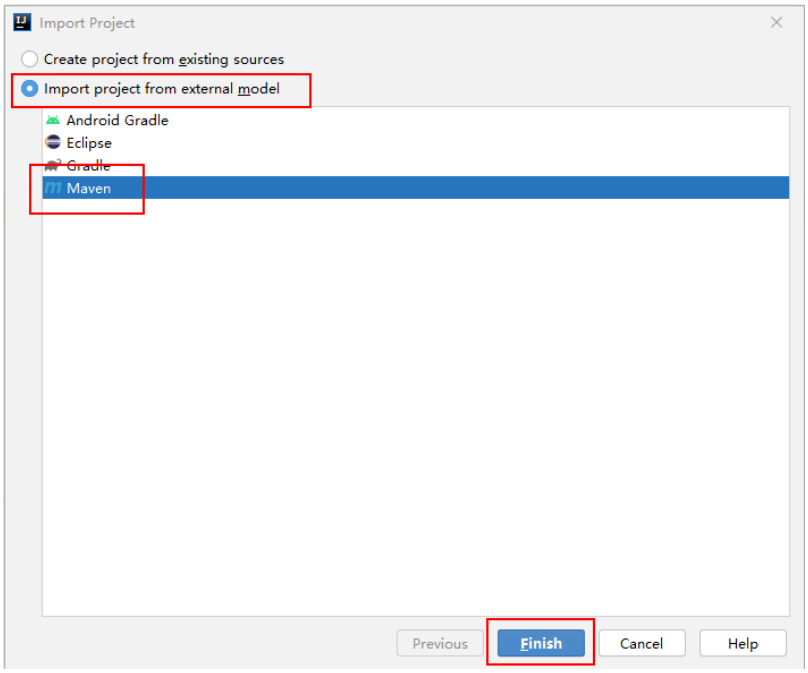
3. 选择要导入的样例工程，例如“storm-examples”。

图 29-9 选择要导入的样例工程



4. 选择以maven工程的形式导入。

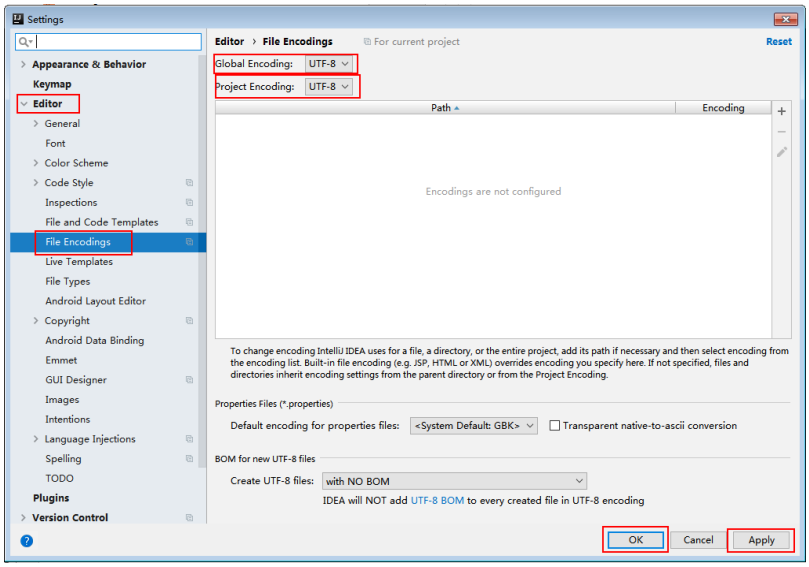
图 29-10 以 maven 工程的形式导入



步骤5 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”。
   弹出“Settings”窗口。
2. 在左边导航上选择“Editor > File Encodings”，在“Project Encoding”和“Global Encoding”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图29-11所示。

图 29-11 设置 IntelliJ IDEA 的编码格式



----结束

29.3 开发 Storm 应用

29.3.1 Storm 样例程序开发思路

通过典型场景，可以快速学习和掌握Storm拓扑的构造和Spout/Bolt开发过程。

场景说明

一个动态单词统计系统，数据源为持续生产随机文本的逻辑单元，业务处理流程如下：

- 数据源持续不断地发送随机文本给文本拆分逻辑，如“apple orange apple”。
- 单词拆分逻辑将数据源发送的每条文本按空格进行拆分，如“apple”，“orange”，“apple”，随后将每个单词逐一发给单词统计逻辑。
- 单词统计逻辑每收到一个单词就进行加一操作，并将实时结果打印输出，如：

  apple：1

  orange：1

  apple：2

功能分解

根据上述场景进行功能分解，如表29-5所示：

表 29-5 在应用中开发的功能

序号	步骤	代码示例
1	创建一个Spout用来生成随机文本	请参见**创建Storm Spout**
2	创建一个Bolt用来将收到的随机文本拆分成一个个单词	请参见**创建Storm Bolt**
3	创建一个Blot用来统计收到的各单词次数	请参见**创建Storm Bolt**
4	创建topology	请参见**创建Storm Topology**

部分代码请参考**开发Storm应用**，完整代码请参考Strom-examples示例工程。

29.3.2 创建 Storm Spout

功能介绍

Spout是Storm的消息源，它是Topology的消息生产者，一般来说消息源会从一个外部源读取数据并向Topology中发送消息（Tuple）。

一个消息源可以发送多条消息流Stream，可以使用OutputFieldsDeclarer.declarerStream来定义多个Stream，然后使用SpoutOutputCollector来发射指定的Stream。

代码样例

下面代码片段在com.huawei.storm.example.common包的RandomSentenceSpout类的nextTuple方法中，作用在于将收到的字符串拆分成单词。

```
/**
  * {@inheritDoc}
  */
 @Override
 public void nextTuple()
 {
     Utils.sleep(100);
     String[] sentences =
         new String[] {"the cow jumped over the moon",
                       "an apple a day keeps the doctor away",
                       "four score and seven years ago",
                       "snow white and the seven dwarfs",
                       "i am at two with nature"};
     String sentence = sentences[random.nextInt(sentences.length)];
     collector.emit(new Values(sentence));
 }
```

29.3.3 创建 Storm Bolt

功能介绍

所有的消息处理逻辑都被封装在各个Bolt中。Bolt包含多种功能：过滤、聚合等等。

如果Bolt之后还有其他拓扑算子，可以使用OutputFieldsDeclarer.declareStream定义Stream，使用OutputCollector.emit来选择要发射的Stream。

代码样例

下面代码片段在com.huawei.storm.example.common包的“SplitSentenceBolt”类的“execute”方法中，作用在于拆分每条语句为单个单词并发送。

```
/**
  * {@inheritDoc}
  */
 @Override
 public void execute(Tuple input, BasicOutputCollector collector)
 {
     String sentence = input.getString(0);
     String[] words = sentence.split(" ");
     for (String word : words)
     {
         word = word.trim();
         if (!word.isEmpty())
         {
             word = word.toLowerCase();
             collector.emit(new Values(word));
         }
     }
 }
```

下面代码片段在com.huawei.storm.example.wordcount包的“WordCountBolt”类的execute方法中，作用在于统计收到的每个单词的数量。

```
  @Override
    public void execute(Tuple tuple, BasicOutputCollector collector)
```

```
{
    String word = tuple.getString(0);
    Integer count = counts.get(word);
    if (count == null)
    {
        count = 0;
    }
    count++;
    counts.put(word, count);
    System.out.println("word: " + word + ", count: " + count);
}
```

29.3.4 创建 Storm Topology

功能介绍

一个Topology是Spouts和Bolts组成的有向无环图。

应用程序是通过storm jar的方式提交，则需要在main函数中调用创建Topology的函数，并在storm jar参数中指定main函数所在类。

代码样例

下面代码片段在com.huawei.storm.example.wordcount包的“WordCountTopology”类的“main”方法中，作用在于构建应用程序并提交。

```
public static void main(String[] args)
        throws Exception
{
    TopologyBuilder builder = buildTopology();

    /*
     * 任务的提交认为三种方式
     * 1、命令行方式提交，这种需要将应用程序jar包复制到客户端机器上执行客户端命令提交
     * 2、远程方式提交，这种需要将应用程序的jar包打包好之后在IntelliJ IDEA中运行main方法提交
     * 3、本地提交 ，在本地执行应用程序，一般用来测试
     * 命令行方式和远程方式安全和普通模式都支持
     * 本地提交仅支持普通模式
     *
     * 用户同时只能选择一种任务提交方式，默认命令行方式提交，如果是其他方式，请删除代码注释即可
     */

    submitTopology(builder, SubmitType.CMD);

}

private static void submitTopology(TopologyBuilder builder, SubmitType type) throws Exception
{
    switch (type)
    {
        case CMD:
        {
            cmdSubmit(builder, null);
            break;
        }
        case REMOTE:
        {
            remoteSubmit(builder);
            break;
        }
        case LOCAL:
        {
            localSubmit(builder);
            break;
        }
    }
```

```
    }

    /**
     * 命令行方式远程提交
     * 步骤如下：
     * 打包成Jar包，然后在客户端命令行上面进行提交
     *   远程提交的时候，要先将该应用程序和其他外部依赖(非excemple工程提供，用户自己程序依赖)的jar包打包成一个大的jar包
     *   再通过storm客户端中storm -jar的命令进行提交
     *
     * 如果是安全环境，客户端命令行提交之前，必须先通过kinit命令进行安全登录
     *
     * 运行命令如下：
     * ./storm jar ../example/example.jar com.huawei.storm.example.WordCountTopology
     */
    private static void cmdSubmit(TopologyBuilder builder, Config conf)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException, AuthorizationException
    {
        if (conf == null)
        {
            conf = new Config();
        }
        conf.setNumWorkers(1);

        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, conf, builder.createTopology());
    }

     private static void localSubmit(TopologyBuilder builder)
         throws InterruptedException
     {
         Config conf = new Config();
         conf.setDebug(true);
         conf.setMaxTaskParallelism(3);
         LocalCluster cluster = new LocalCluster();
         cluster.submitTopology(TOPOLOGY_NAME, conf, builder.createTopology());
         Thread.sleep(10000);
         cluster.shutdown();
     }


    private static void remoteSubmit(TopologyBuilder builder)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException, AuthorizationException,
        IOException
    {
        Config config = createConf();

        String userJarFilePath = "替换为用户jar包地址";
        System.setProperty(STORM_SUBMIT_JAR_PROPERTY, userJarFilePath);

        //安全模式下的一些准备工作
        if (isSecurityModel())
        {
            securityPrepare(config);
        }
        config.setNumWorkers(1);
        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, config, builder.createTopology());
    }

       private static TopologyBuilder buildTopology()
     {
         TopologyBuilder builder = new TopologyBuilder();
         builder.setSpout("spout", new RandomSentenceSpout(), 5);
         builder.setBolt("split", new SplitSentenceBolt(), 8).shuffleGrouping("spout");
         builder.setBolt("count", new WordCountBolt(), 12).fieldsGrouping("split", new Fields("word"));
         return builder;
     }
```


说明

如果拓扑开启了ack，推荐acker的数量不大于所设置的worker数量。

29.4 调测 Storm 应用

29.4.1 打包 Storm 样例工程应用

操作场景

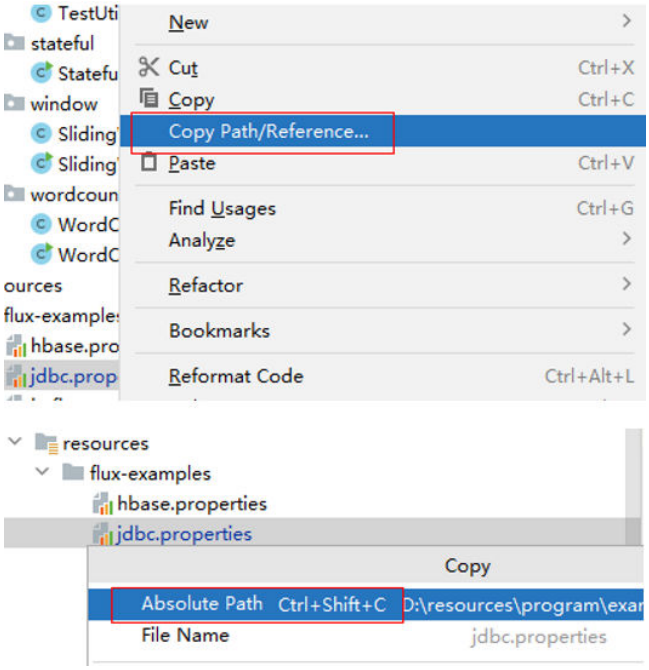
通过IntelliJ IDEA导出Jar包并指定导出jar包名称，比如“storm-examples.jar”。

操作步骤

步骤1 若Storm-JDBC样例需要在Windows下运行，则需要替换配置文件路径；否则，不需要执行此步骤。

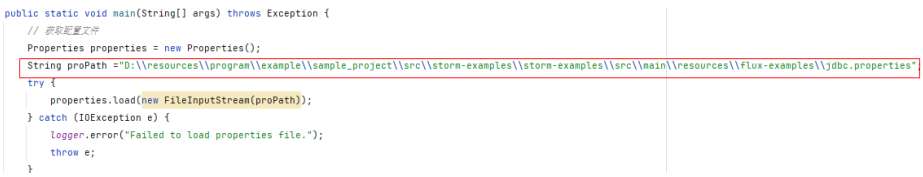
1. 在IDEA界面右键单击“jdbc.properties”文件，选择“Copy Path/Reference > Absolute Path”，复制“jdbc.properties”文件路径。

图 29-12 复制“jdbc.properties”文件路径



2. 修改“SimpleJDBCTopology.java”的**main()**方法中**proPath**值为**步骤1.1**复制的“jdbc.properties”文件路径。

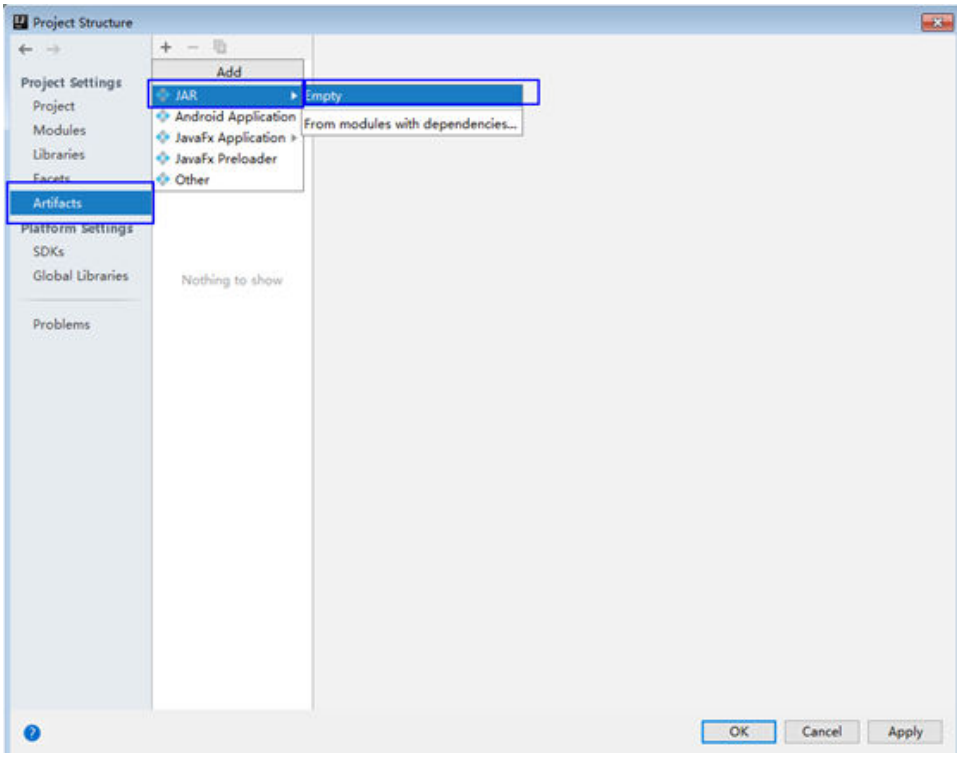
图 29-13 proPath 路径修改



步骤2 在IntelliJ IDEA中，在生成Jar包之前配置工程的Artifacts信息。

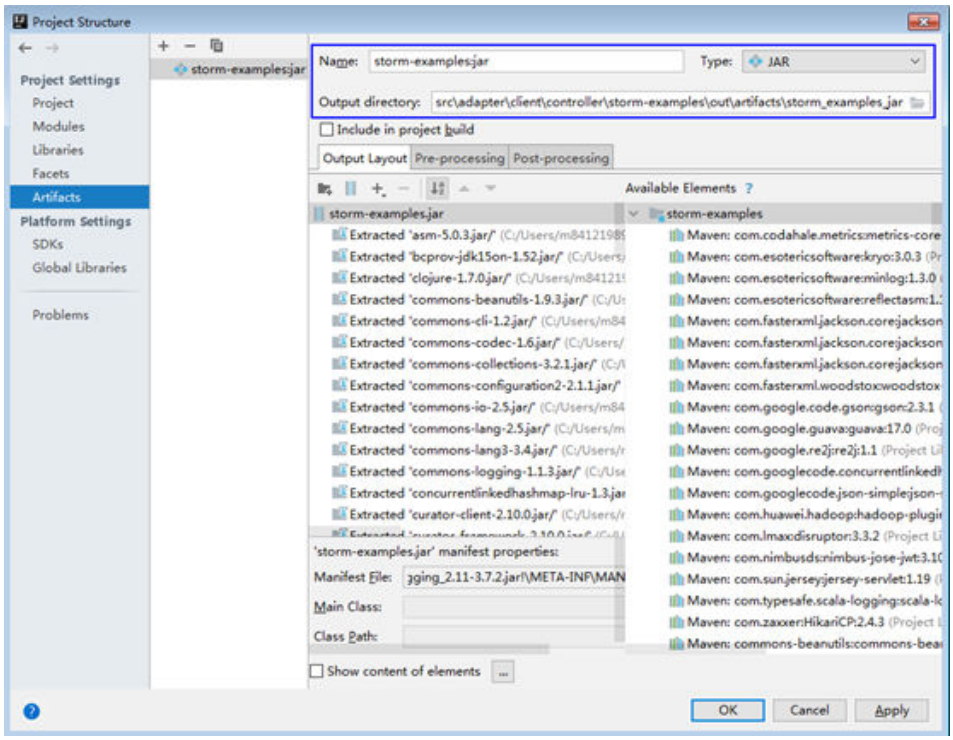
1. 进入IntelliJ IDEA，选择“File > Project Structures...”进入“Project Structure”页面。
2. 在“Project Structure”页面，选择“Artifacts”，单击“+”并选择“JAR > Empty”。

图 29-14 添加 Artifacts



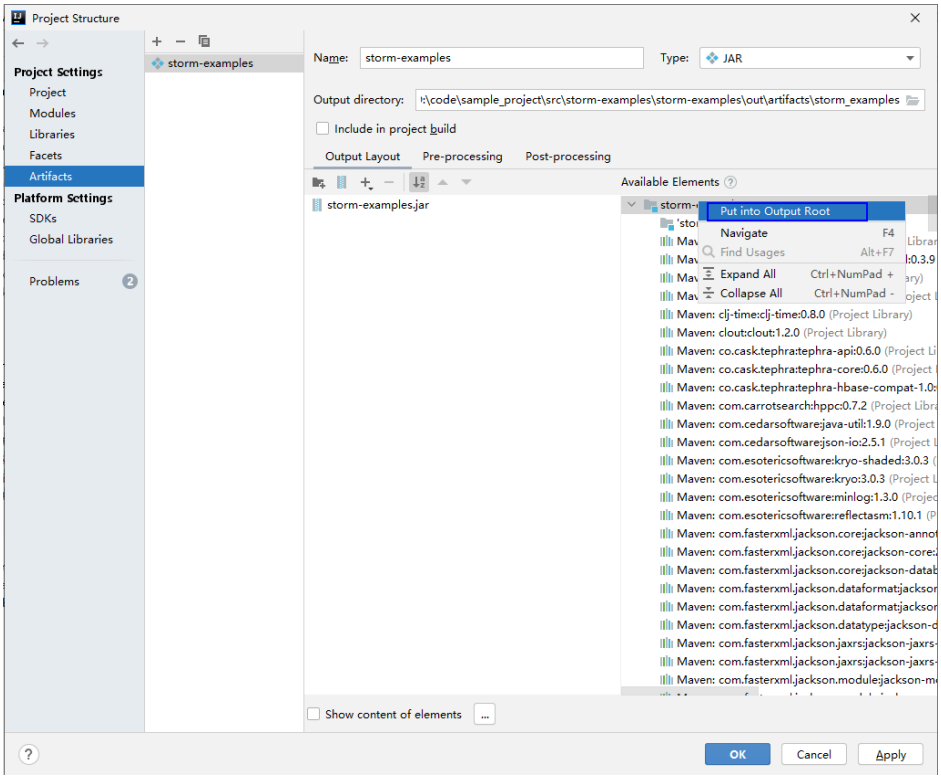
3. 根据实际情况设置Jar包的名称、类型以及输出路径。

图 29-15 设置基本信息



4. 选中“storm-examples”，右键选择“Put into Output Root”。然后单击“Apply”。

图 29-16 Put into Output Root

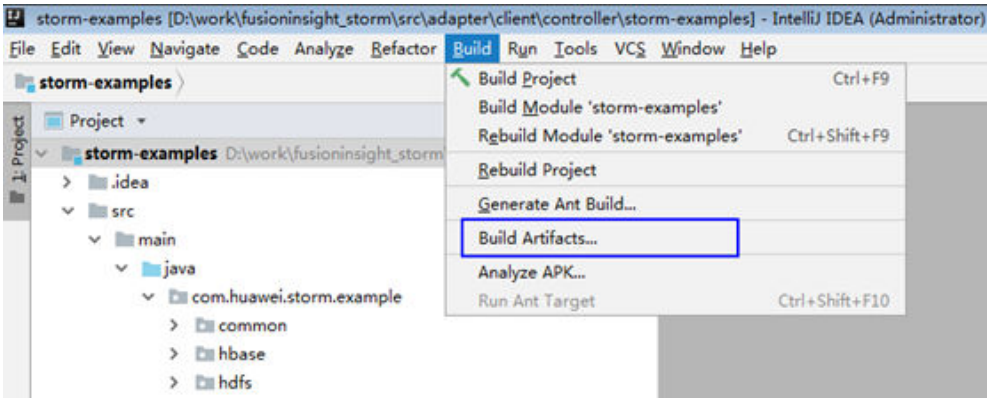


5. 最后单击“OK”完成配置。

步骤3 生成Jar包。

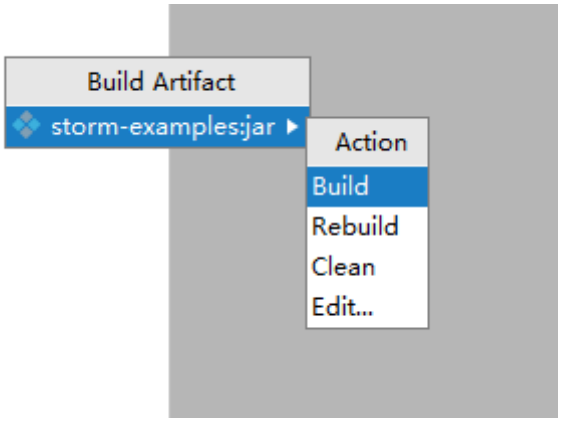
1. 在IDEA主页面，选择“Build > Build Artifacts...”。

图 29-17 Build Artifacts



2. 在弹出的菜单中，选择“storm-examples:jar > Build”开始生成Jar包。

图 29-18 Build



3. 当Event log中出现如下类似日志时，表示Jar包生成成功。您可以在图29-15中配置的路径下获取到Jar包。

```
14:37 Compilation completed successfully in 25 s 991 ms
```

----结束

29.4.2 打包 Storm 业务

29.4.2.1 Linux 下打包 Storm 业务

操作场景

Storm支持在Linux环境下打包。用户可以将从IntelliJ IDEA中导出的Jar包和需要的其他相关Jar包上传到Linux环境中执行打包。

打包业务的目的，是将IntelliJ IDEA代码生成的jar包与工程依赖的jar包，合并导出可提交的source.jar。

打包需使用storm-jartool工具，可在Windows或Linux上进行。

前提条件

- 已安装Storm客户端。
- 已执行**打包Storm样例工程应用**。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

操作步骤

步骤1 将从IntelliJ IDEA中导出的jar包复制到Linux客户端指定目录（例如“/opt/jarsource”）。

步骤2 若业务需要访问外部组件，其所依赖的配置文件请参考相关开发指引，获取到配置文件后将配置文件放在**步骤1**中指定的目录下。

步骤3 若业务需要访问外部组件，其所依赖的jar包请参考相关开发指引，获取到jar包后将jar包放在**步骤1**中指定的目录下。

步骤4 在Storm客户端安装目录“Storm/storm-1.2.1/bin”下执行打包命令，将上述jar包打成一个完整的业务jar包放入指定目录/opt/jartarget（可为任意空目录）。执行**sh storm-jartool.sh /opt/jarsource/ /opt/jartarget**命令后，会在“/opt/jartarget”下生成source.jar。

----结束

29.4.2.2 Windows 下打包 Storm 业务

操作场景

打包业务的目的，是将IntelliJ IDEA代码生成的jar包与工程依赖的jar包，合并导出可提交的source.jar。

打包需使用storm-jartool工具，可在Windows或Linux上进行。

前提条件

已执行**打包Storm样例工程应用**。

操作步骤

步骤1 将从IntelliJ IDEA打包出来的jar包放入指定文件夹（例如“D:\source”）。

步骤2 在样例代码目录“src/storm-examples/storm-examples”下创建“lib”目录，将IntelliJ IDEA中导出的jar包复制到“lib”目录下，并解压。

步骤3 若业务需要访问外部组件，其所依赖的配置文件请参考相关开发指引，获取到配置文件后将配置文件放在**步骤1**中指定的目录下。

步骤4 若业务需要访问外部组件，其所依赖的jar包请参考相关开发指引，获取到jar包后将jar包放在**步骤1**中指定的目录下。

步骤5 在IntelliJ IDEA样例工程的“tools”目录下找到打包工具：“storm-jartool.cmd”。

步骤6 双击打包工具，输入要打包的jar包所在目录（“D:\source”）并回车，再输入打出包存放的目录（“D:\target”），在“D:\target”中，会生成“source.jar”文件。

----结束

29.4.3 提交 Storm 拓扑

29.4.3.1 Linux 中安装客户端时提交 Storm 拓扑

操作场景

在Linux环境下可以使用storm命令行完成拓扑的提交。

前提条件

- 已安装Storm客户端。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。
- 已执行**打包Storm业务**步骤，打出source.jar。

操作步骤

步骤1 安全模式下，请先进行安全认证。

1. 初始化客户端环境变量。

   进入客户端安装目录“/opt/Storm_client”执行以下命令，导入环境变量信息。

   source bigdata_env

2. 使用在“准备开发用户”章节创建的开发用户进行安全登录。

   执行**kinit**命令进行“人机”用户的安全登录。

   kinit *用户名*

   例如：

   kinit developuser

   然后按照提示输入密码，无异常提示返回，则完成了用户的kerberos认证。

步骤2 提交拓扑（以wordcount为例，其它拓扑请参照相关开发指引），进入Storm客户端“Storm/storm-1.2.1/bin”目录，将刚打出的source.jar提交（如果在Windows上进行的打包，则需要将Windows上的source.jar上传到Linux服务器，假定上传到“/opt/jartarget”目录），执行命令：**storm jar /opt/jartarget/source.jar com.huawei.storm.example.wordcount.WordCountTopology**。

步骤3 执行**storm list**命令，查看已经提交的应用程序，如果发现名称为word-count的应用程序，则说明任务提交成功。

说明

如果业务设置为本地模式，且使用命令行方式提交时，请确保提交环境为普通模式环境，当前不支持安全环境下使用命令提交本地模式的业务。

----结束

29.4.3.2 Linux 中未安装客户端时提交 Storm 拓扑

操作场景

Storm支持拓扑在未安装Storm客户端的Linux环境中运行。

前提条件

- 客户端机器的时间与MRS集群的时间要保持一致，时间差要小于5分钟。
- 当Linux环境所在主机不是集群中的节点时，需要在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

操作步骤

步骤1 准备依赖的Jar包和配置文件。

在Linux环境新建目录，例如“/opt/test”，并创建子目录“lib”和“src/main/resources/”。将样例工程中“lib”文件夹下的Jar包上传Linux环境的“lib”目录。将样例工程中“src/main/resources”文件夹下的配置文件上传到Linux环境的“src/main/resources”目录。

步骤2 在IntelliJ IDEA工程中修改WordCountTopology.java类，使用remoteSubmit方式提交应用程序。并替换用户keytab文件名称，用户principal名称，和Jar文件地址。

- 使用remoteSubmit方式提交应用程序

```
public static void main(String[] args)
        throws Exception
    {
        TopologyBuilder builder = buildTopology();

        /*
         * 任务的提交认为三种方式
         * 1、命令行方式提交，这种需要将应用程序jar包复制到客户端机器上执行客户端命令提交
         * 2、远程方式提交，这种需要将应用程序的jar包打包好之后在IntelliJ IDEA中运行main方法提交
         * 3、本地提交 ，在本地执行应用程序，一般用来测试
         * 命令行方式和远程方式安全和普通模式都支持
         * 本地提交仅支持普通安全模式
         *
         * 用户同时只能选择一种任务提交方式，默认命令行方式提交，如果是其他方式，请删除代码注释即
可
         */

        submitTopology(builder, SubmitType.REMOTE);
    }
```

- 修改userJarFilePath为Linux环境指定路径“/opt/test/lib/example.jar”。

```
private static void remoteSubmit(TopologyBuilder builder)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException,
AuthorizationException,
        IOException
    {
        Config config = createConf();

        String userJarFilePath = "/opt/test/lib/example.jar ";
        System.setProperty(STORM_SUBMIT_JAR_PROPERTY, userJarFilePath);

        //安全模式下的一些准备工作
        if (isSecurityModel())
        {
            securityPrepare(config);
        }
        config.setNumWorkers(1);
        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, config,
```

```
builder.createTopology());
    }
```

- 安全模式下需要执行安全准备，根据实际情况修改userKeyTablePath和userPrincipal为**导入并配置Storm样例工程**章节的步骤2中所获取用户的keytab文件名称和principal。

```
private static void securityPrepare(Config config)
        throws IOException
    {
        String userKeyTablePath =
            System.getProperty("user.dir") + File.separator + "src" + File.separator + "main" +
File.separator + "resources" + File.separator + "user.keytab";
        String userPrincipal = "StreamingDeveloper";
        String krbFilePath = System.getProperty("user.dir") + File.separator + "src" + File.separator +
"main" + File.separator + "resources" + File.separator +"krb5.conf";

        //windows路径下分隔符替换
        userKeyTablePath = userKeyTablePath.replace("\\", "\\\\");
        krbFilePath = krbFilePath.replace("\\", "\\\\");

        String principalInstance =
String.valueOf(config.get(Config.STORM_SECURITY_PRINCIPAL_INSTANCE));
        LoginUtil.setKrb5Config(krbFilePath);
        LoginUtil.setZookeeperServerPrincipal("zookeeper/" + principalInstance);
        LoginUtil.setJaasFile(userPrincipal, userKeyTablePath);
    }
```

步骤3 导出Jar包并上传到Linux环境。

- 参考**打包Storm样例工程应用**执行打包，并将jar包命名为“example.jar”。
- 将导出的Jar包复制到Linux环境的“/opt/test/lib”目录下。

步骤4 切换到“/opt/test”，执行以下命令，运行Jar包。

java -classpath /opt/test/lib/*:/opt/test/src/main/resources com.huawei.storm.example.wordcount.WordCountTopology

----结束

29.4.3.3 在 IDEA 中提交 Storm 拓扑

操作场景

Storm支持IntelliJ IDEA远程提交拓扑，目前样例代码中仅WordCountTopology支持远程提交，其他拓扑想实现远程提交，请参考WordCountTopology实现远程提交函数。

前提条件

- 已执行**打包Storm样例工程应用**。
- 调整IntelliJ IDEA客户端机器时间，和Storm集群时间差不超过5分钟。
- 确保本地的hosts文件中配置了远程集群所有主机的主机名和业务IP映射关系。

操作步骤

步骤1 修改WordCountTopology.java类，使用remoteSubmit方式提交应用程序。并替换用户keytab文件名称，用户principal名称，和Jar文件地址。

- 使用remoteSubmit方式提交应用程序

```
public static void main(String[] args)
        throws Exception
    {
```



```
        TopologyBuilder builder = buildTopology();

        /*
         * 任务的提交认为三种方式
         * 1、命令行方式提交，这种需要将应用程序jar包复制到客户端机器上执行客户端命令提交
         * 2、远程方式提交，这种需要将应用程序的jar包打包好之后在IntelliJ IDEA中运行main方法提交
         * 3、本地提交 ，在本地执行应用程序，一般用来测试
         * 命令行方式和远程方式安全和普通模式都支持
         * 本地提交仅支持普通模式
         *
         * 用户同时只能选择一种任务提交方式，默认命令行方式提交，如果是其他方式，请删除代码注释即
可
         */

        submitTopology(builder, SubmitType.REMOTE);
    }
```

- 根据实际情况修改userJarFilePath为实际的拓扑Jar包地址

```
private static void remoteSubmit(TopologyBuilder builder)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException,
AuthorizationException,
        IOException
    {
        Config config = createConf();

        String userJarFilePath = "D:\\example.jar";
        System.setProperty(STORM_SUBMIT_JAR_PROPERTY, userJarFilePath);

        //安全模式下的一些准备工作
        if (isSecurityModel())
        {
            securityPrepare(config);
        }
        config.setNumWorkers(1);
        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, config,
builder.createTopology());
    }
```

- 安全模式下需要执行安全准备，根据实际情况修改userKeyTablePath和userPrincipal为**导入并配置Storm样例工程**章节的步骤2中所获取用户的keytab文件路径和principal

```
private static void securityPrepare(Config config)
        throws IOException
    {
        String userKeyTablePath =
            System.getProperty("user.dir") + File.separator + "src" + File.separator + "main" +
File.separator + "resources" + File.separator + "user.keytab";
        String userPrincipal = "StreamingDeveloper";
        String krbFilePath = System.getProperty("user.dir") + File.separator + "src" + File.separator +
"main" + File.separator + "resources" + File.separator +"krb5.conf";

        //windows路径下分隔符替换
        userKeyTablePath = userKeyTablePath.replace("\\", "\\\\");
        krbFilePath = krbFilePath.replace("\\", "\\\\");

        String principalInstance =
String.valueOf(config.get(Config.STORM_SECURITY_PRINCIPAL_INSTANCE));
        LoginUtil.setKrb5Config(krbFilePath);
        LoginUtil.setZookeeperServerPrincipal("zookeeper/" + principalInstance);
        LoginUtil.setJaasFile(userPrincipal, userKeyTablePath);
    }
```

步骤2 执行WordCountTopology.java类的Main方法提交应用程序。

----结束

29.4.4 查看 Storm 应用调测结果

操作场景

Storm应用程序运行完成后，可通过登录Storm WebUI查看应用程序的运行情况。

操作步骤

步骤1 登录FusionInsight Manager系统。

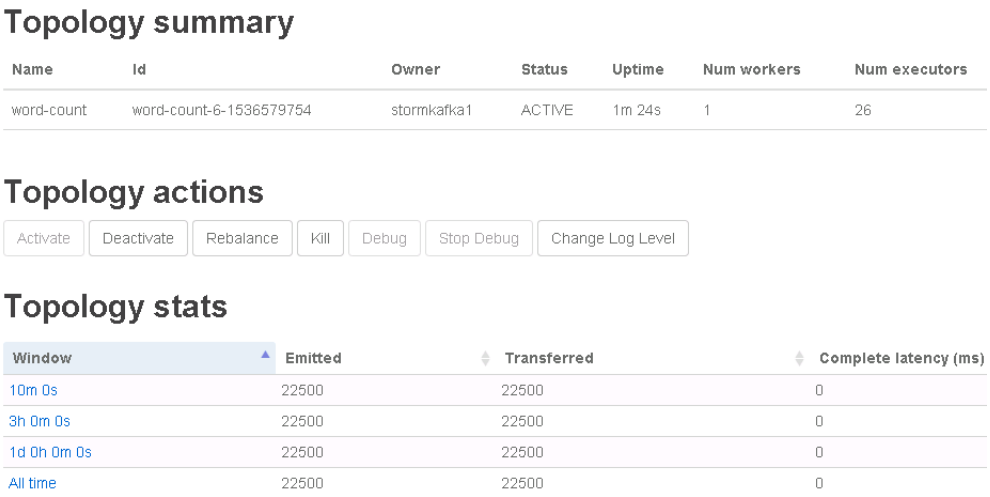
在浏览器地址栏中输入访问地址，地址格式为“https://*FusionInsight Manager系统的WebService浮动IP地址*:28443/web”。

例如，在IE浏览器地址栏中，输入“https://10.0.0.1:28443/web”。

步骤2 选择“集群 > *待操作集群的名称* > 服务 > Storm”，单击进入Storm WebUI。

步骤3 在Storm UI中单击word-count应用，查看应用程序运行情况，如图29-19所示。

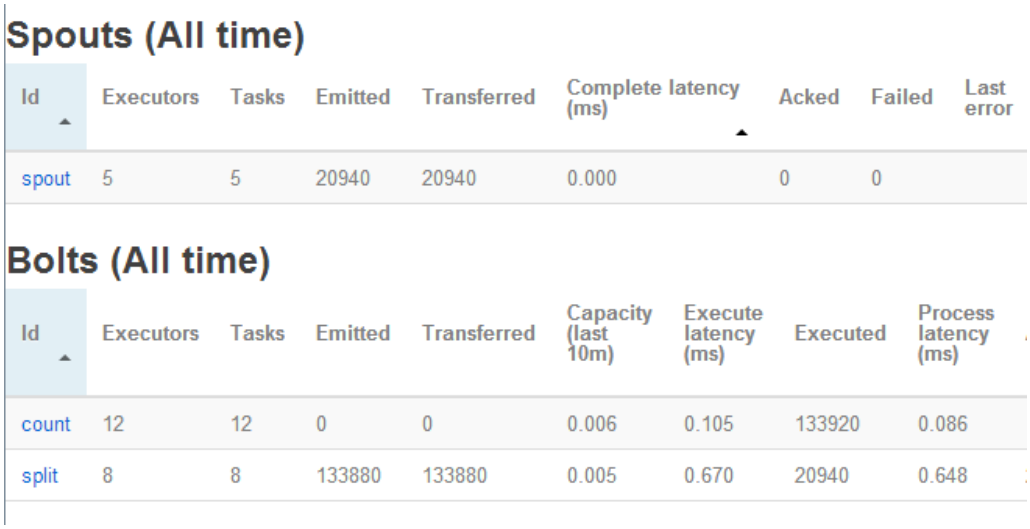
图 29-19 Storm 应用程序执行界面



Topology stats统计了最近各个不同时间段的算子之间发送数据的总数据量。

Spouts中统计了spout算子从启动到现在发送的消息总量。Bolts中统计了Count算子和split算子的发送消息总量，如图29-20所示。

图 29-20 Storm 应用程序算子发送数据总量



Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
spout	5	5	20940	20940	0.000	0	0	

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)
count	12	12	0	0	0.006	0.105	133920	0.086
split	8	8	133880	133880	0.005	0.670	20940	0.648

----结束

29.5 Storm 应用开发常见问题

29.5.1 Storm-Kafka 开发指引

操作场景

本文档主要说明如何使用Storm-Kafka工具包，完成Storm和Kafka之间的交互。包含KafkaSpout和KafkaBolt两部分。KafkaSpout主要完成Storm从Kafka中读取数据的功能；KafkaBolt主要完成Storm向Kafka中写入数据的功能。

本章节代码样例基于Kafka新API，对应IntelliJ IDEA工程中com.huawei.storm.example.kafka.NewKafkaTopology.java。

本章节只适用于MRS产品Storm与Kafka组件间的访问。本章中描述的jar包的具体版本信息请以实际情况为准。

应用开发操作步骤

步骤1 确认Storm和Kafka组件已经安装，并正常运行。

步骤2 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\storm-examples”目录下的样例工程文件夹storm-examples并将storm-examples导入到IntelliJ IDEA开发环境，参见**准备Storm应用开发环境**。

步骤3 在Linux环境下安装Storm客户端。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

步骤4 如果集群启用了安全服务，需要从管理员处获取一个“人机”用户，用于登录FusionInsight Manager平台并通过认证，并且获取到该用户的keytab文件。

说明

- 获取的用户需要同时属于storm组和kafka组。
- 默认情况下，用户的密码有效期是90天，所以获取的keytab文件的有效期是90天。如果需要延长该用户keytab的有效期，修改用户的密码策略并重新获取keytab。

步骤5 下载并安装Kafka客户端程序。

----结束

代码样例

创建拓扑：

```
public static void main(String[] args) throws Exception {

// 设置拓扑配置
Config conf = new Config();

// 配置安全插件
setSecurityPlugin(conf);

if (args.length >= 2) {
// 用户更改了默认的keytab文件名，这里需要将新的keytab文件名通过参数传入
conf.put(Config.TOPOLOGY_KEYTAB_FILE, args[1]);
}

// 定义KafkaSpout
KafkaSpout kafkaSpout = new KafkaSpout<String, String>(
getKafkaSpoutConfig(getKafkaSpoutStreams()));

// CountBolt
CountBolt countBolt = new CountBolt();
//SplitBolt
SplitSentenceBolt splitBolt = new SplitSentenceBolt();

// KafkaBolt配置信息
KafkaBolt<String, String> kafkaBolt = new KafkaBolt<String, String>();
kafkaBolt.withTopicSelector(new DefaultTopicSelector(OUTPUT_TOPIC))
.withTupleToKafkaMapper(
new FieldNameBasedTupleToKafkaMapper("word", "count"));
kafkaBolt.withProducerProperties(getKafkaProducerProps());

// 定义拓扑
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("kafka-spout", kafkaSpout, 10);
builder.setBolt("split-bolt", splitBolt,10).shuffleGrouping("kafka-spout", STREAMS[0]);
builder.setBolt("count-bolt", countBolt, 10).fieldsGrouping(
"split-bolt", new Fields("word"));
builder.setBolt("kafka-bolt", kafkaBolt, 10).shuffleGrouping("count-bolt");

// 命令行提交拓扑
StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
}
```

说明

如果修改了集群域名，在设置Kafka消费者/生产者属性中kerberos域名时，需要将其设置为集群实际域名，例如props.put(KERBEROS_DOMAIN_NAME , "*hadoop.hadoop1.com*")。

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Storm样例工程应用**。

步骤2 获取相关配置文件，获取方式如下：

- 安全模式：参见**步骤4**获取keytab文件。
- 普通模式：无。

步骤3 获取下列jar包：

在安装好的Kafka客户端目录中进入“Kafka/kafka/libs”目录，获取如下jar包：

- kafka_<version>.jar
- scala-library-<version>.jar
- scala-logging_2.11-3.7.2.jar
- metrics-core-<version>.jar
- kafka-clients-<version>.jar
- zookeeper-<version>.jar

在Storm客户端的“streaming-cql-<*HD-Version*>/lib”目录中获取如下jar包：

- storm-kafka-client-<version>.jar
- storm-kafka-<version>.jar
- slf4j-api-<version>.jar
- guava-<version>.jar
- json-simple-<version>.jar
- curator-client-<version>.jar
- curator-framework-<version>.jar
- curator-recipes-<version>.jar

步骤4 将**步骤1**、**步骤2**和**步骤3**中获取的jar包和配置文件合并统一打出完整的业务jar包，请参见**打包Storm业务**。

步骤5 进入Linux系统中进入Kafka客户端所在目录，在Kafka/kafka/bin目录下使用Kafka客户端创建拓扑中所用到的Topic，执行命令：

./kafka-topics.sh --create --topic input --partitions 2 --replication-factor 2 --zookeeper {ip:port}/kafka

./kafka-topics.sh --create --topic output --partitions 2 --replication-factor 2 --zookeeper {ip:port}/kafka

说明

- “--zookeeper”后面填写的是ZooKeeper地址，需要改为安装集群时配置的ZooKeeper地址。
- 安全模式下，需要Kafka管理员用户创建Topic。

步骤6 在Linux系统中完成拓扑的提交。

提交命令示例（拓扑名为kafka-test）：

storm jar /opt/jartarget/source.jar com.huawei.storm.example.kafka.NewKafkaTopology kafka-test

说明

- 安全模式下，在提交“source.jar”之前，请确保已经进行kerberos安全登录，并且keytab方式下，登录用户和所上传keytab所属用户必须是同一个用户。
- 安全模式下，Kafka需要用户有相应Topic的访问权限，因此首先需要在Kafka所在集群上使用Kafka管理员用户登录，之后使用**kafka-acls.sh**命令给用户赋权，成功之后再使用提交用户登录并提交拓扑。Kafka用户赋权详见“Kafka开发指南”的“更多信息”章节。

步骤7 拓扑提交成功后，可以向Kafka中发送数据，观察是否有相关信息生成。

在Linux系统中进入Kafka客户端所在目录，在Kafka/kafka/bin目录下启动consumer观察数据是否生成。执行命令：

./kafka-console-consumer.sh --bootstrap-server {ip:port} --topic output --consumer.config ../config/consumer.properties

同时在Linux系统中进入Kafka客户端所在目录，在Kafka/kafka/bin目录下启动producer，向Kafka中写入数据。执行命令：

./kafka-console-producer.sh --broker-list {ip:port} --topic input --producer.config ../config/producer.properties

向input中写入测试数据，可以观察到output中有对应的数据产生，则说明Storm-Kafka拓扑运行成功。

----结束

29.5.2 Storm-JDBC 开发指引

操作场景

本文档主要说明如何使用开源Storm-JDBC工具包，完成Storm和JDBC之间的交互。Storm-JDBC中包含两类Bolt：JdbcInsertBolt和JdbcLookupBolt。其中，JdbcLookupBolt主要负责从数据库中查数据，JdbcInsertBolt主要向数据库中存数据。当然，JdbcLookupBolt和JdbcInsertBolt中也可以增加处理逻辑对数据进行处理。

本章节只适用Storm与JDBC组件间的访问。本章中描述的jar包的具体版本信息请以实际情况为准。

应用开发操作步骤

步骤1 确认华为MRS产品Storm组件已经安装，且正常运行。

步骤2 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\storm-examples”目录下的样例工程文件夹storm-examples并将storm-examples导入到IntelliJ IDEA开发环境，参见**准备Storm应用开发环境**。

步骤3 工程导入后，修改样例工程“resources/flux-examples”目录下的“jdbc.properties”文件，根据实际环境信息修改相关参数。

```
#配置JDBC服务端IP地址
JDBC_SERVER_NAME=
#配置JDBC服务端端口
JDBC_PORT_NUM=
#配置JDBC登录用户名
JDBC_USER_NAME=
#配置JDBC登录用户密码
#密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全
JDBC_PASSWORD=
```

```
#配置database表名
JDBC_BASE_TBL=
```

步骤4 在Linux环境下安装Storm客户端。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

----结束

数据库配置—Derby 数据库配置过程

步骤1 首先应下载一个数据库，可根据具体场景选择最适合的数据库。

该任务以Derby数据库为例。Derby是一个小型的，java编写的，易于使用却适合大多数应用程序的开放源码数据库。

步骤2 Derby数据库的获取。在官网下载最新版的Derby数据库，将下载下来的数据库将传入Linux客户端(如"/opt")，并解压。

步骤3 在Derby的安装目录下，进入bin目录，输入如下命令：

export DERBY_INSTALL=/opt/db-derby-10.12.1.1-bin

export CLASSPATH=$DERBY_INSTALL/lib/derbytools.jar:$DERBY_INSTALL\lib\derbynet.jar:.

export DERBY_HOME=/opt/db-derby-10.12.1.1-bin

. setNetworkServerCP

./startNetworkServer -h *主机名*

步骤4 执行**./ij**命令，输入**connect 'jdbc:derby://***主机名***:1527/example;create=true';**，建立连接。

步骤5 数据库建立好后，可以执行sql语句进行操作，需要建立两张表ORIGINAL和GOAL，并向ORIGINAL中插入一组数据，命令如下：（表名仅供参考，可自行设定）

CREATE TABLE GOAL(WORD VARCHAR(12),COUNT INT);

CREATE TABLE ORIGINAL(WORD VARCHAR(12),COUNT INT);

INSERT INTO ORIGINAL VALUES('orange',1),('pineapple',1),('banana',1),('watermelon',1);

----结束

代码样例

SimpleJDBCTopology代码样例（代码中涉及到的IP端口请修改为实际的IP及端口）

```
public class SimpleJDBCTopology {

    private static final Logger logger = Logger.getLogger(SimpleJDBCTopology.class);
    private static final String WORD_SPOUT = "WORD_SPOUT";

    private static final String JDBC_INSERT_BOLT = "JDBC_INSERT_BOLT";

    private static final String JDBC_LOOKUP_BOLT = "JDBC_LOOKUP_BOLT";

    // 用户创建的源表表名，可自行修改
    private static final String JDBC_ORIGIN_TBL = "ORIGINAL";
```

```
// 用户创建的目标表表名，可自行修改
private static final String JDBC_INSERT_TBL = "GOAL";

@SuppressWarnings("rawtypes")
public static void main(String[] args) throws Exception {
    // 获取配置文件
    Properties properties = new Properties();
    String proPath =
            System.getProperty("user.dir")
                    + File.separator
                    + "src"
                    + File.separator
                    + "main"
                    + File.separator
                    + "resources"
                    + File.separator
                    + "flux-examples"
                    + File.separator
                    + "jdbc.properties";
    try {
        properties.load(new FileInputStream(proPath));
    } catch (IOException e) {
        logger.error("Failed to load properties file.");
        throw e;
    }
    String serverName = properties.getProperty("JDBC_SERVER_NAME");
    String portNum = properties.getProperty("JDBC_PORT_NUM");
    String userName = properties.getProperty("JDBC_USER_NAME");
    String password = properties.getProperty("JDBC_PASSWORD");
    String baseTbl = properties.getProperty("JDBC_BASE_TBL");

    // connectionProvider配置
    Map<String, Object> hikariConfigMap = Maps.newHashMap();
    hikariConfigMap.put("dataSourceClassName", "org.apache.derby.jdbc.ClientDataSource");
    hikariConfigMap.put("dataSource.serverName", serverName);
    hikariConfigMap.put("dataSource.portNumber", portNum);
    hikariConfigMap.put("dataSource.databaseName", baseTbl);
    hikariConfigMap.put("dataSource.user", userName);
    hikariConfigMap.put("dataSource.password", password);
    hikariConfigMap.put("connectionTestQuery", "select COUNT from " + JDBC_INSERT_TBL);

    Config conf = new Config();

    ConnectionProvider connectionProvider = new HikariCPConnectionProvider(hikariConfigMap);

    // JdbcLookupBolt实例化
    Fields outputFields = new Fields("WORD", "COUNT");
    List<Column> queryParamColumns = Lists.newArrayList(new Column("WORD", Types.VARCHAR));
    SimpleJdbcLookupMapper jdbcLookupMapper = new SimpleJdbcLookupMapper(outputFields,
queryParamColumns);
    String selectSql = "select COUNT from " + JDBC_ORIGIN_TBL + " where WORD = ?";
    JdbcLookupBolt wordLookupBolt = new JdbcLookupBolt(connectionProvider, selectSql,
jdbcLookupMapper);

    // JdbcInsertBolt实例化
    String tableName = JDBC_INSERT_TBL;
    JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper(tableName, connectionProvider);
    JdbcInsertBolt wordInsertBolt =
            new JdbcInsertBolt(connectionProvider, simpleJdbcMapper)
                    .withTableName(JDBC_INSERT_TBL)
                    .withQueryTimeoutSecs(30);

    JDBCSpout wordSpout = new JDBCSpout();

    // 构造拓扑，wordSpout==>wordLookupBolt==>wordInsertBolt
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout(WORD_SPOUT, wordSpout);
    builder.setBolt(JDBC_LOOKUP_BOLT, wordLookupBolt, 1).fieldsGrouping(WORD_SPOUT, new
```

```
Fields("WORD"));
        builder.setBolt(JDBC_INSERT_BOLT, wordInsertBolt, 1).fieldsGrouping(JDBC_LOOKUP_BOLT, new
Fields("WORD"));

        StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
    }
}
```

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Storm样例工程应用**。

步骤2 获取下列jar包：

- 在安装好的db数据库目录下进入lib目录，获取如下jar包：
  - derbyclient.jar
  - derby.jar
- 在Storm客户端的“streaming-cql-<*HD-Version*>/lib”目录中获取如下jar包：
  - storm-jdbc-<version>.jar
  - guava-<version>.jar
  - commons-lang3-<version>.jar
  - commons-lang-<version>.jar
  - HikariCP-<version>.jar

步骤3 将上述两步中获取的jar包合并统一打出完整的业务jar包，请参见**打包Storm业务**。

步骤4 执行命令提交拓扑。提交命令示例（拓扑名为jdbc-test）：

storm jar /opt/jartarget/source.jar com.huawei.storm.example.jdbc.SimpleJDBCTopology jdbc-test

----结束

结果查看

当拓扑提交完成后，可以去数据库中查看对应表中是否有数据插入，具体过程如下：

执行SQL语句**select * from goal;** 查询“goal”表中的数据，如果goal表中有数据添加，则表明整个拓扑运行成功。

29.5.3 Storm-HDFS 开发指引

操作场景

本章节只适用于MRS产品中Storm和HDFS交互的场景。本章中描述的jar包的具体版本信息请以实际情况为准。

安全模式下登录方式分为两种，票据登录和keytab文件登录，两种方式操作步骤基本一致，票据登录方式为开源提供的能力，后期需要人工上传票据，存在可靠性和易用性问题，因此推荐使用keytab方式。

应用开发操作步骤

步骤1 确认Storm和HDFS组件已经安装，并正常运行。

步骤2 将storm-examples导入到IntelliJ IDEA开发环境，请参见**准备Storm应用开发环境**。

步骤3 如果集群启用了安全服务，按登录方式需要进行以下配置：

- keytab方式：需要从管理员处获取一个“人机”用户，用于登录FusionInsight Manager平台并通过认证，并且获取到该用户的keytab文件。
- 票据方式：从管理员处获取一个“人机”用户，用于后续的安全登录，开启Kerberos服务的renewable和forwardable开关并且设置票据刷新周期，开启成功后重启kerberos及相关组件。

说明

- 获取的用户需要属于storm组。
- 默认情况下，用户的密码有效期是90天，所以获取的keytab文件的有效期是90天。如果需要延长该用户keytab的有效期，修改用户的密码策略并重新获取keytab。
- Kerberos服务的renewable、forwardable开关和票据刷新周期的设置在Kerberos服务的配置页面的“系统”标签下，票据刷新周期的修改可以根据实际情况修改“kdc_renew_lifetime”和“kdc_max_renewable_life”的值。

步骤4 下载并安装HDFS客户端。

步骤5 获取相关配置文件。获取方法如下：

在安装好的HDFS客户端目录下找到目录“/opt/clientHDFS/HDFS/hadoop/etc/hadoop”，在该目录下获取到配置文件“core-site.xml”和“hdfs-site.xml”。

如果使用keytab登录方式，按**步骤3**获取keytab文件；如果使用票据方式，则无需获取额外的配置文件。

说明

获取到的keytab文件默认文件名为user.keytab，若用户需要修改，可直接修改文件名，但在提交任务时需要额外上传修改后的文件名作为参数。

步骤6 获取相关jar包。获取方法如下：

- 在安装好的HDFS客户端目录下找到目录HDFS/hadoop/share/hadoop/common/lib，获取如下jar包：
  - commons-cli-<version>.jar
  - commons-io-<version>.jar
  - commons-lang-<version>.jar
  - commons-lang3-<version>.jar
  - commons-collections-<version>.jar
  - commons-configuration2-<version>.jar
  - commons-logging-<version>.jar
  - guava-<version>.jar
  - hadoop-*.jar
  - protobuf-java-<version>.jar
  - jackson-databind-<version>.jar
  - jackson-core-<version>.jar
  - jackson-annotations-<version>.jar
  - re2j-<version>.jar

  - jaeger-core-<version>.jar
  - opentracing-api-<version>.jar
  - opentracing-noop-<version>.jar
  - opentracing-tracerresolver-<version>.jar
  - opentracing-util-<version>.jar
- 在安装好的HDFS客户端目录下找到目录“HDFS/hadoop/share/hadoop/common”，获取该目录下的hadoop-*.jar。
- 在安装好的HDFS客户端目录下找到目录“HDFS/hadoop/share/hadoop/client”，获取该目录下的hadoop-*.jar。
- 在安装好的HDFS客户端目录下找到目录“HDFS/hadoop/share/hadoop/hdfs”，获取该目录下的hadoop-hdfs-*.jar。
- 在样例工程“/src/storm-examples/storm-examples/lib”中获取如下jar包：
  - storm-hdfs-<version>.jar
  - storm-autocreds-<version>.jar

----结束

IntelliJ IDEA 代码样例

创建Topology。

```
public static void main(String[] args) throws Exception
  {
    TopologyBuilder builder = new TopologyBuilder();

    // 分隔符格式，当前采用“|”代替默认的“，”对tuple中的field进行分隔
    // HdfsBolt必选参数
    RecordFormat format = new DelimitedRecordFormat()
            .withFieldDelimiter("|");

    // 同步策略，每1000个tuple对文件系统进行一次同步
    // HdfsBolt必选参数
    SyncPolicy syncPolicy = new CountSyncPolicy(1000);

    // 文件大小循环策略，当文件大小到达5M时，从头开始写
    // HdfsBolt必选参数
    FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.MB);

    // 写入HDFS的目的文件
    // HdfsBolt必选参数
    FileNameFormat fileNameFormat = new DefaultFileNameFormat()
            .withPath("/user/foo/");


    //创建HdfsBolt
    HdfsBolt bolt = new HdfsBolt()
            .withFsUrl(DEFAULT_FS_URL)
            .withFileNameFormat(fileNameFormat)
            .withRecordFormat(format)
            .withRotationPolicy(rotationPolicy)
            .withSyncPolicy(syncPolicy);

    //Spout生成随机语句
    builder.setSpout("spout", new RandomSentenceSpout(), 1);
    builder.setBolt("split", new SplitSentence(), 1).shuffleGrouping("spout");
    builder.setBolt("count", bolt, 1).fieldsGrouping("split", new Fields("word"));

    //增加Kerberos认证所需的plugin到列表中，安全模式必选
    setSecurityConf(conf,AuthenticationType.KEYTAB);
```

```
    Config conf = new Config();
    //将客户端配置的plugin列表写入config指定项中，安全模式必配
    conf.put(Config.TOPOLOGY_AUTO_CREDENTIALS, auto_tgts);

    if(args.length >= 2)
    {
        //用户更改了默认的keytab文件名，这里需要将新的keytab文件名通过参数传入
        conf.put(Config.STORM_CLIENT_KEYTAB_FILE, args[1]);
    }

    //命令行提交拓扑
    StormSubmitter.submitTopology(args[0], conf, builder.createTopology());

}
```

说明

Storm不支持将HDFS的目的文件路径设置为HDFS的SM4加密分区。

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Storm样例工程应用**。

步骤2 将**步骤1**导出的本地Jar包，**步骤5**中获取的配置文件和**步骤6**中获取的jar包合并统一打出完整的业务jar包，请参见**打包Storm业务**。

步骤3 执行命令提交拓扑。

keytab方式下，若用户修改了keytab文件名，如修改为“huawei.keytab”，则需要在命令中增加第二个参数进行说明，提交命令示例（拓扑名为hdfs-test）：

storm jar /opt/jartarget/source.jar com.huawei.storm.example.hdfs.SimpleHDFSTopology hdfs-test huawei.keytab

说明

安全模式下在提交source.jar之前，请确保已经进行kerberos安全登录，并且keytab方式下，登录用户和所上传keytab所属用户必须是同一个用户。

步骤4 拓扑提交成功后请登录HDFS集群查看。

步骤5 如果使用票据登录，则需要使用命令行定期上传票据，具体周期由票据刷新截止时间而定，步骤如下：

1. 在安装好的storm客户端目录的“Storm/storm-1.2.1/conf/storm.yaml”文件尾部新起一行添加如下内容：

   topology.auto-credentials: -org.apache.storm.security.auth.kerberos.AutoTGT

2. 执行命令：**./storm upload-credentials hdfs-test**

----结束

29.5.4 Storm-HBase 开发指引

操作场景

本章节只适用于MRS产品中Storm和HBase交互的场景。本章中描述的jar包的具体版本信息请以实际情况为准。

安全模式下登录方式分为两种，票据登录和keytab文件登录，两种方式操作步骤基本一致。票据登录方式为开源提供的能力，存在票据过期问题，后期需要人工上传票据，并且可靠性和易用性较差，因此推荐使用keytab方式。

应用开发操作步骤

步骤1 确认Storm和HBase组件已经安装，并正常运行。

步骤2 将storm-examples导入到IntelliJ IDEA开发环境，请参见**准备Storm应用开发环境**。

步骤3 如果集群启用了安全服务，按登录方式分为以下两种：

- keytab方式：需要从管理员处获取一个“人机”用户，用于登录FusionInsight Manager平台并通过认证，并且获取到该用户的keytab文件。
- 票据方式：从管理员处获取一个“人机”用户，用于后续的安全登录，开启Kerberos服务的renewable和forwardable开关并且设置票据刷新周期，开启成功后重启kerberos及相关组件。

说明

- 获取的用户需要属于storm组。
- 默认情况下，用户的密码有效期是90天，所以获取的keytab文件的有效期是90天。如果需要延长该用户keytab的有效期，请修改用户的密码策略并重新获取keytab。
- Kerberos服务的renewable、forwardable开关和票据刷新周期的设置在Kerberos服务的配置页面的“系统”标签下，票据刷新周期的修改可以根据实际情况修改“kdc_renew_lifetime”和“kdc_max_renewable_life”的值。

步骤4 下载并安装HBase客户端程序。

步骤5 获取相关配置文件。获取方法如下：

在安装好的hbase客户端目录下找到目录“/opt/clientHbase/HBase/hbase/conf”，在该目录下获取到core-site.xml、hdfs-site.xml、hbase-site.xml配置文件。

如果使用keytab登录方式，按**步骤3**获取keytab文件；如果使用票据方式，则无需获取额外的配置文件。

说明

获取到的keytab文件默认文件名为user.keytab，若用户需要修改，可直接修改文件名，但在提交任务时需要额外上传修改后的文件名作为参数。

步骤6 获取相关jar包。获取方式如下：

- 在安装好的HBase客户端目录下找到目录HBase/hbase/lib，获取如下jar包：
  - hbase-*.jar
  - hadoop-*.jar
  - jackson-core-asl-<version>.jar
  - jackson-mapper-asl-<version>.jar
  - commons-cli-<version>.jar
  - commons-io-<version>.jar
  - commons-lang-<version>.jar
  - commons-lang3-<version>.jar
  - commons-collections-<version>.jar

  - commons-configuration2-<version>.jar
  - guava-<version>.jar
  - protobuf-java-<version>.jar
  - netty-all-<version>.jar
  - zookeeper-<version>.jar
  - zookeeper-jute-<version>.jar
  - metrics-core-<version>.jar
  - commons-validator-<version>.jar
- 在HBase客户端安装目录下找到目录HBase/hbase/lib/client-facing-thirdparty，获取commons-logging-<version>.jar。
- 在HBase客户端安装目录下找到目录HBase/hbase/lib/jdbc，获取htrace-core-<version>-incubating.jar和htrace-core4-<version>-incubating.jar。
- 在样例工程“/src/storm-examples/storm-examples/lib”中获取如下jar包：
  - storm-hbase-<version>.jar
  - storm-autocreds-<version>.jar

----结束

IntelliJ IDEA 代码样例

创建Topology。

```
public static void main(String[] args) throws Exception
    {
        Config conf = new Config();

        //增加kerberos认证所需的plugin到列表中，安全模式必选
        setSecurityConf(conf,AuthenticationType.KEYTAB);

        if(args.length >= 2)
        {
        //用户更改了默认的keytab文件名，这里需要将新的keytab文件名通过参数传入
        conf.put(Config.STORM_CLIENT_KEYTAB_FILE, args[1]);
        }
        //hbase的客户端配置，这里只提供了“hbase.rootdir”配置项，参数可选
        Map<String, Object> hbConf = new HashMap<String, Object>();
        if(args.length >= 3)
        {
            hbConf.put("hbase.rootdir", args[2]);
        }
        //必配参数，若用户不输入，则该项为空
        conf.put("hbase.conf", hbConf);

        //spout为随机单词spout
        WordSpout spout = new WordSpout();
        WordCounter bolt = new WordCounter();

        //HbaseMapper，用于解析tuple内容
        SimpleHBaseMapper mapper = new SimpleHBaseMapper()
                .withRowKeyField("word")
                .withColumnFields(new Fields("word"))
                .withCounterFields(new Fields("count"))
                .withColumnFamily("cf");

        //HBaseBolt，第一个参数为表名
        //withConfigKey("hbase.conf")将hbase的客户端配置传入HBaseBolt
        HBaseBolt hbase = new HBaseBolt("WordCount", mapper).withConfigKey("hbase.conf");
```



```
        // wordSpout ==> countBolt ==> HBaseBolt
        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout(WORD_SPOUT, spout, 1);
        builder.setBolt(COUNT_BOLT, bolt, 1).shuffleGrouping(WORD_SPOUT);
        builder.setBolt(HBASE_BOLT, hbase, 1).fieldsGrouping(COUNT_BOLT, new Fields("word"));
        //命令行提交拓扑
        StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
    }
```

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Storm样例工程应用**。

步骤2 将**步骤1**中导出的本地Jar包，**步骤5**中获取的配置文件和**步骤6**中获取的jar包合并统一打出完整的业务jar包，请参见**打包Storm业务**。

步骤3 执行命令提交拓扑。

keytab方式下，若用户修改了keytab文件名，如修改为“huawei.keytab”，则需要在命令中增加第二个参数进行说明，提交命令示例（拓扑名为hbase-test）：

storm jar /opt/jartarget/source.jar com.huawei.storm.example.hbase.SimpleHBaseTopology hbase-test huawei.keytab

说明

- 安全模式下在提交source.jar之前，请确保已经进行kerberos安全登录，并且keytab方式下，登录用户和所上传keytab所属用户必须是同一个用户。
- 因为示例中的HBaseBolt并没有建表功能，在提交之前确保hbase中存在相应的表，若不存在需要手动建表，hbase shell建表语句如下create 'WordCount', 'cf'。
- 安全模式下hbase需要用户有相应表甚至列族和列的访问权限，因此首先需要在hbase所在集群上使用hbase管理员用户登录，之后在hbase shell中使用grant命令给提交用户申请相应表的权限，如示例中的WordCount，成功之后再使用提交用户登录并提交拓扑。

步骤4 拓扑提交成功后请自行登录HBase集群查看。

步骤5 如果使用票据登录，则需要使用命令行定期上传票据，具体周期由票据刷新截止时间而定，步骤如下：

1. 在安装好的storm客户端目录的“Storm/storm-1.2.1/conf/storm.yaml”文件尾部新起一行添加如下内容：

   topology.auto-credentials: -org.apache.storm.security.auth.kerberos.AutoTGT

2. 执行命令**./storm upload-credentials hbase-test**。

----结束

29.5.5 Storm Flux 开发指引

操作场景

本章节只适用于MRS产品中Storm组件使用Flux框架提交和部署拓扑的场景。本章中描述的jar包的具体版本信息请以实际情况为准。

Flux框架是Storm提供的提高拓扑部署易用性的框架。通过Flux框架，用户可以使用yaml文件来定义和部署拓扑，并且最终通过storm jar命令来提交拓扑的一种方式，极大地方便了拓扑的部署和提交，缩短了业务开发周期。

基本语法说明

使用Flux定义拓扑分为两种场景，定义新拓扑和定义已有拓扑。

1. **使用Flux定义新拓扑**

   使用Flux定义拓扑，即使用yaml文件来描述拓扑，一个完整的拓扑定义需要包含以下几个部分：

   - 拓扑名称
   - 定义拓扑时需要的组件列表
   - 拓扑的配置
   - 拓扑的定义，包含spout列表、bolt列表和stream列表

   定义拓扑名称：

   ```
   name: "yaml-topology"
   ```

   定义组件列表示例:

   ```
   #简单的component定义
    components:
    - id: "stringScheme"
    className: "org.apache.storm.kafka.StringScheme"

    #使用构造函数定义component
    - id: "defaultTopicSelector"
    className: "org.apache.storm.kafka.bolt.selector.DefaultTopicSelector"
    constructorArgs:
    - "output"

    #构造函数入参使用引用，使用`ref`标志来说明引用
    #在使用引用时请确保被引用对象在前面定义
    - id: "stringMultiScheme"
    className: "org.apache.storm.spout.SchemeAsMultiScheme"
    constructorArgs:
    - ref: "stringScheme"

    #构造函数入参引用指定的properties文件中的配置项，使用`${}`标志来表示
    #引用properties文件时，请在使用storm jar命令提交拓扑时使用--filter my-prop.properties的方式指明
   properties文件路径
    - id: "zkHosts"
    className: "org.apache.storm.kafka.ZkHosts"
    constructorArgs:
    - "${kafka.zookeeper.root.list}"

    #构造函数入参引用环境变量，使用`${ENV-[NAME]}`方式来引用
    #NAME必须是一个已经定义的环境变量
    - id: "zkHosts"
    className: "org.apache.storm.kafka.ZkHosts"
    constructorArgs:
    - "${ENV-ZK_HOSTS}"

    #使用`properties`关键字初始化内部私有变量
    - id: spoutConfig
    className: "org.apache.storm.kafka.SpoutConfig"
    constructorArgs:
    - ref: "zkHosts"
    - "input"
    - "/kafka/input"
    - "myId"
    properties:
    - name: "scheme"
    ref: "stringMultiScheme"
   ```

   定义拓扑的配置示例：

   ```
   config:
    #简单配置项
    topology.workers: 1
   ```

```
#配置项值为列表，使用`[]`表示
topology.auto-credentials: ["class1","class2"]

#配置项值为map结构
kafka.broker.properties:
metadata.broker.list: "${metadata.broker.list}"
producer.type: "async"
request.required.acks: "0"
serializer.class: "kafka.serializer.StringEncoder"
```

定义spout/bolt列表示例：

```
#定义spout列表
spouts:
- id: "spout1"
className: "org.apache.storm.kafka.KafkaSpout"
constructorArgs:
- ref: "spoutConfig"
parallelism: 1

#定义bolt列表
bolts:
- id: "bolt1"
className: "com.huawei.storm.example.hbase.WordCounter"
parallelism: 1

#使用方法来初始化对象，关键字为`configMethods`
- id: "bolt2"
className: "org.apache.storm.hbase.bolt.HBaseBolt"
constructorArgs:
- "WordCount"
- ref: "mapper"
configMethods:
- name: "withConfigKey"
args: ["hbase.conf"]
parallelism: 1
```

定义stream列表示例：

```
#定义流式需要制定分组方式，关键字为`grouping`，当前提供的分组方式关键字有：
#`ALL`,`CUSTOM`,`DIRECT`,`SHUFFLE`,`LOCAL_OR_SHUFFLE`,`FIELDS`,`GLOBAL`, 和 `NONE`.
#其中`CUSTOM`为用户自定义分组

#简单流定义，分组方式为SHUFFLE
streams:
- name: "spout1 --> bolt1"
from: "spout1"
to: "bolt1"
grouping:
type: SHUFFLE

#分组方式为FIELDS，需要传入参数
- name: "bolt1 --> bolt2"
from: "bolt1"
to: "bolt2"
grouping:
type: FIELDS
args: ["word"]

#分组方式为CUSTOM，需要指定用户自定义分组类
- name: "bolt-1 --> bolt2"
from: "bolt-1"
to: "bolt-2"
grouping:
type: CUSTOM
customClass:
className: "org.apache.storm.testing.NGrouping"
constructorArgs:
- 1
```

2. **使用Flux定义已有拓扑**

   如果已经拥有拓扑（例如已经使用java代码定义了拓扑），仍然可以使用Flux框架来提交和部署，这时需要在现有的拓扑定义（如MyTopology.java）中实现getTopology()方法，在java中定义如下：

   ```
   public StormTopology getTopology(Config config)
   或者
   public StormTopology getTopology(Map<String, Object> config)
   ```

   这时可以使用如下yaml文件来定义拓扑：

   ```
   name: "existing-topology"  #拓扑名可随意指定
   topologySource:
   className: "custom-class"  #请指定客户端类
   ```

   当然，仍然可以指定其他方法名来获得StormTopology（非getTopology()方法），yaml文件示例如下：

   ```
   name: "existing-topology"
   topologySource:
   className: "custom-class "
   methodName: "getTopologyWithDifferentMethodName"
   ```

   说明

   指定的方法必须接受一个Map<String, Object>类型或者Config类型的入参，并且返回org.apache.storm.generated.StormTopology类型的对象，和getTopology()方法相同。

应用开发操作步骤

步骤1 确认Storm组件已经安装，并正常运行。如果业务需要连接其他组件，请同时安装该组件并运行。

步骤2 将storm-examples导入到IntelliJ IDEA开发环境，请参见**导入并配置Storm样例工程**。

步骤3 参考storm-examples工程src/main/resources/flux-examples目录下的相关yaml应用示例，开发客户端业务。

步骤4 获取相关配置文件。

说明

本步骤只适用于业务中有访问外部组件需求的场景，如HDFS、HBase等，获取方式请参见Storm-HDFS开发指引或者Storm-HBase开发指引。若业务无需获取相关配置文件，请忽略本步骤。

步骤5 获取相关jar包，获取方法如下：

- 在Storm客户端的“streaming-cql-<*HD-Version*>/lib”目录中获取如下jar包：

  flux-core-<version>.jar

  flux-wrappers-<version>.jar
- 获取业务相关其他jar包，如访问HDFS时需要获取的jar包请参见**步骤6**，其他场景类似。

----结束

Flux 配置文件样例

下面是一个完整的访问Kafka业务的yaml文件样例：

```
name: "simple_kafka"

components:
```



```
- id: "zkHosts"  #对象名称
className: "org.apache.storm.kafka.ZkHosts"  #完整的类名
constructorArgs:  #构造函数
- "${kafka.zookeeper.root.list}"  #构造函数的参数

- id: "stringScheme"
className: "org.apache.storm.kafka.StringScheme"

- id: "stringMultiScheme"
className: "org.apache.storm.spout.SchemeAsMultiScheme"
constructorArgs:
- ref: "stringScheme"  #使用了引用，值为前面定义的stringScheme

- id: spoutConfig
className: "org.apache.storm.kafka.SpoutConfig"
constructorArgs:
- ref: "zkHosts"  #使用了引用
- "input"
- "/kafka/input"
- "myId"
properties:  #使用properties来设置本对象中的名为“scheme”的私有变量
- name: "scheme"
ref: "stringMultiScheme"

- id: "defaultTopicSelector"
className: "org.apache.storm.kafka.bolt.selector.DefaultTopicSelector"
constructorArgs:
- "output"

- id: "fieldNameBasedTupleToKafkaMapper"
className: "org.apache.storm.kafka.bolt.mapper.FieldNameBasedTupleToKafkaMapper"
constructorArgs:
- "words"  #构造函数中第一个入参
- "count"  #构造函数中第二个入参

config:
topology.workers: 1  #设置拓扑的worker数量为1
kafka.broker.properties:  #设置kafka相关的配置，值为map结构
metadata.broker.list: "${metadata.broker.list}"
producer.type: "async"
request.required.acks: "0"
serializer.class: "kafka.serializer.StringEncoder"

spouts:
- id: "kafkaSpout"  #spout名称
className: "org.apache.storm.kafka.KafkaSpout"#spout的类名
constructorArgs:  #使用构造函数的方式初始化
- ref: "spoutConfig"  #构造函数的入参使用了引用
parallelism: 1  #该spout的并发设置为1

bolts:
- id: "splitBolt"
className: "com.huawei.storm.example.common.SplitSentenceBolt"
parallelism: 1

- id: "countBolt"
className: "com.huawei.storm.example.kafka.CountBolt"
parallelism: 1

- id: "kafkaBolt"
className: "org.apache.storm.kafka.bolt.KafkaBolt"
configMethods:  #使用调用对象内部方法的形式初始化对象
- name: "withTopicSelector"  #调用的内部方法名
args:  #内部方法需要的入参
- ref: "defaultTopicSelector"  #入参只有一个，使用了引用
- name: "withTupleToKafkaMapper"  #调用第二个内部方法
args:
- ref: "fieldNameBasedTupleToKafkaMapper"
```

```
#定义数据流
streams:
- name: "kafkaSpout --> splitBolt"  #第一个数据流名称，只作为展示
from: "kafkaSpout"  #数据流起点，值为spouts中定义的kafkaSpout
to: "splitBolt"  #数据流终点，值为bolts中定义的splitBolt
grouping:#定义分组方式
type: LOCAL_OR_SHUFFLE  #分组方式为local_or_shuffle

- name: "splitBolt --> countBolt"  #第二个数据流
from: "splitBolt"
to: "countBolt"
grouping:
type: FIELDS  #分组方式为fields
args: ["word"]  #fields方式需要传入参数

- name: "countBolt --> kafkaBolt"  #第三个数据流
from: "countBolt"
to: "kafkaBolt"
grouping:
type: SHUFFLE  #分组方式为shuffle，无需传入参数
```

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Storm样例工程应用**。

步骤2 将**步骤4**中获取的配置文件和**步骤5**中获取的jar包合并统一打出完整的业务jar包，请参见**打包Storm业务**。

步骤3 将开发好的yaml文件及相关的properties文件复制至storm客户端所在主机的任意目录下，如“/opt”。

步骤4 执行命令提交拓扑。

storm jar /opt/jartarget/source.jar org.apache.storm.flux.Flux --remote /opt/my-topology.yaml

如果设置业务以本地模式启动，则提交命令如下：

storm jar /opt/jartarget/source.jar org.apache.storm.flux.Flux --local /opt/my-topology.yaml

说明

如果业务设置为本地模式，请确保提交环境为普通模式环境，当前不支持安全环境下使用命令提交本地模式的业务。

如果使用了properties文件，则提交命令如下：

storm jar /opt/jartarget/source.jar org.apache.storm.flux.Flux --remote /opt/my-topology.yaml --filter /opt/my-prop.properties

步骤5 拓扑提交成功后请自行登录storm UI查看。

----结束

29.5.6 Storm 对外接口介绍

- Storm-HDFS采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-hdfs**。
- Storm-HBase采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-hbase**。

- Storm-Kafka采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-kafka**。
- Storm-JDBC采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-jdbc**。

29.5.7 如何使用 IDEA 远程调试业务

问题

使用Storm客户端提交了业务之后，如何使用IDEA远程调试业务?

回答

以调试WordCount程序为例，演示如何进行IDEA的远程调试：

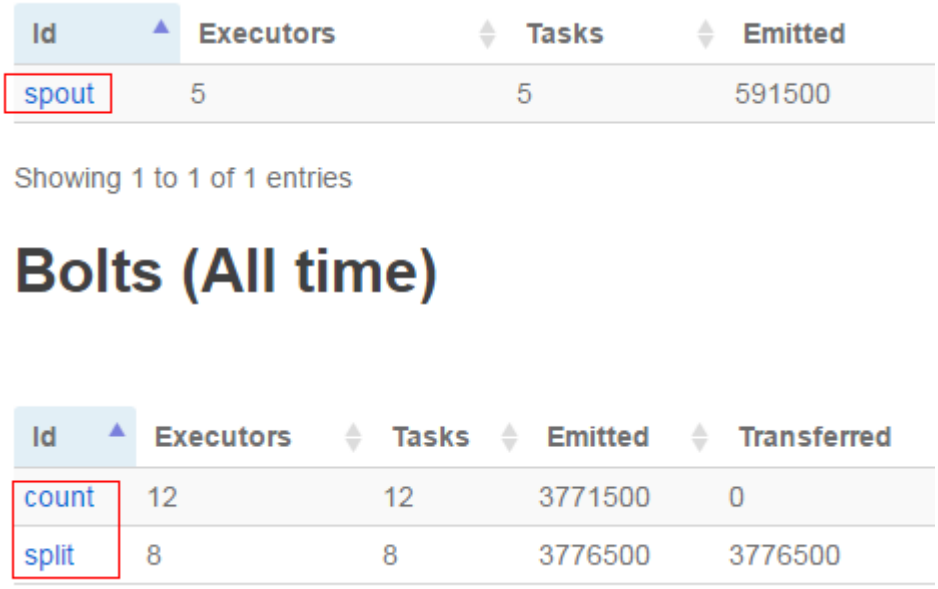
步骤1 登录FusionInsight Manager系统，选择“集群 > *待操作集群的名称* > 服务 > Storm”，选择“配置”选项卡，在搜索框中搜索并调大nimbus.task.timeout.secs和supervisor.worker.start.timeout.secs的值，建议调整为最大值。然后在WORKER_GC_OPTS的现有值后追加-Xdebug -Xrunjdwp:transport=dt_socket,address=5055,suspend=n,server=y，保存配置后重启相关实例。

说明

调试Storm程序需要先修改指定的服务端参数，并在重启服务后生效，建议在测试环境上进行调测。

步骤2 提交拓扑后，在Storm UI上进入到Topology界面，再单击进入要调试组件界面。

图 29-21 进入拓扑的 Component 界面



步骤3 在组件页面获取worker进程运行的主机ip地址，如果有多个则任选一个。

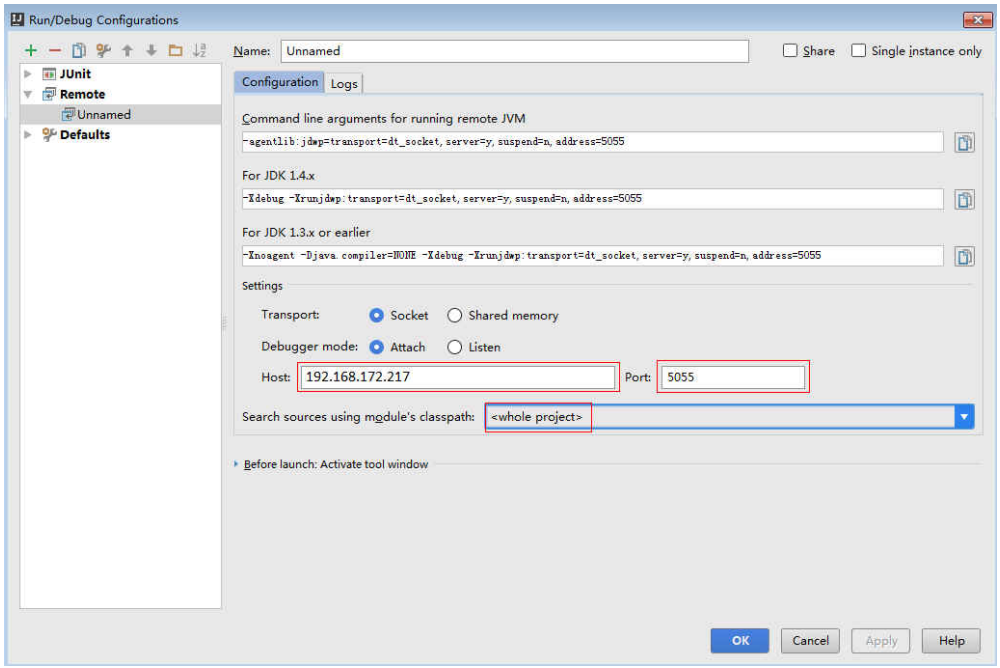
图 29-22 获取 Worker 运行的主机

步骤4 打开IDEA工程，在菜单栏中选择“Run > Edit Configurations”。

步骤5 在弹出的配置窗口中用鼠标左键单击左上角的号，在下拉菜单中选择Remote，然后选择对应要调试的源码模块路径，并配置远端调试参数Host和Port，如下图所示。

其中Host为获取的Worker运行的主机IP地址，Port为调试的端口号（确保该端口在运行机器上没被占用）。

图 29-23 配置参数



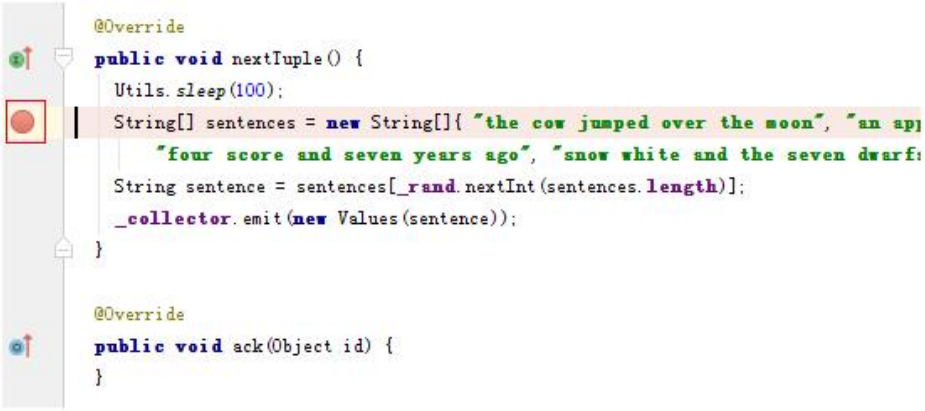
说明

当改变Port端口号时，在WORKER_GC_OPTS中追加的调试参数也要跟着改变，比如Port设置为8011，对应的调试参数则变更为-Xdebug -Xrunjdwp:transport=dt_socket,address=8011,suspend=n,server=y

步骤6 设置调试断点。

在IDEA代码编辑窗口左侧空白处单击鼠标左键设置相应代码行断点，如下图所示。

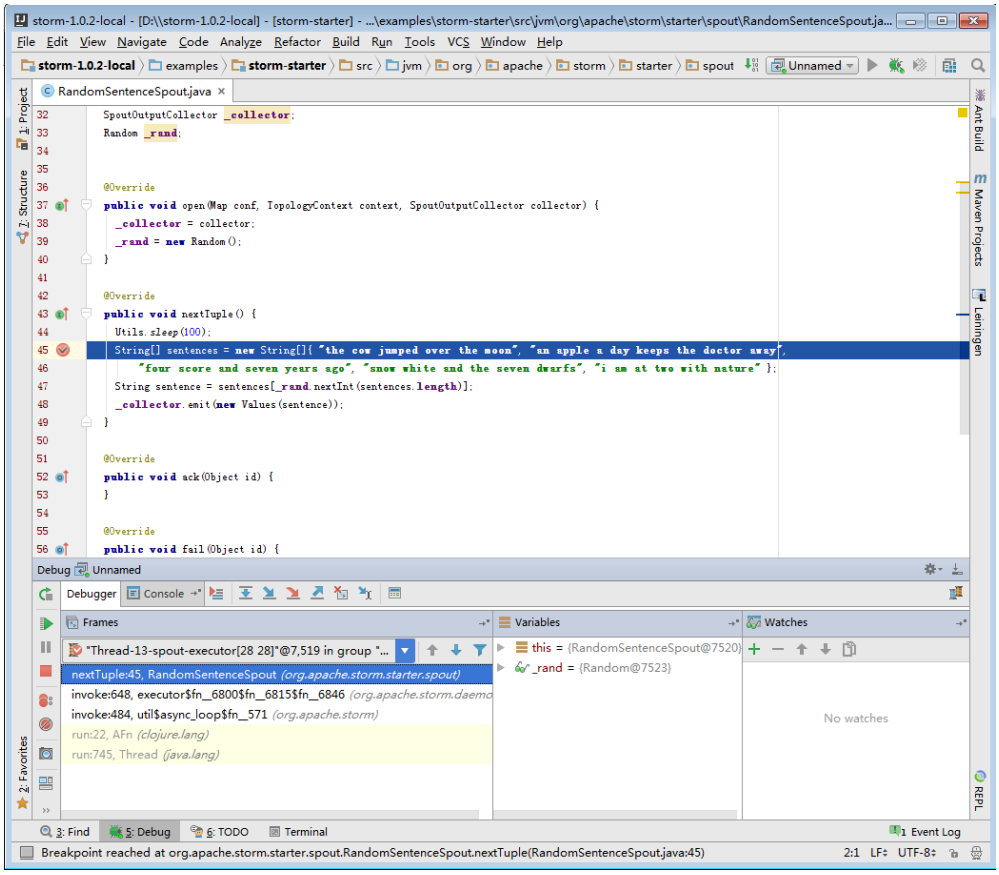
图 29-24 设置断点



步骤7 启动调试。

在IDEA菜单栏中选择“Run > Debug 'Unnamed'”开启调试窗口，接着开始调试，比如单步调试、查看调用栈、跟踪变量值等，如下图所示。

图 29-25 调试



----结束

29.5.8 IntelliJ IDEA 中远程提交拓扑执行 Main 时报错：Command line is too long

问题

IntelliJ IDEA中远程提交拓扑，执行Main方法时IntelliJ IDEA报如下错：

Command line is too long. Shorten command line for ServiceStarter or also for Application default configuration.

回答

步骤1 打开项目中“ .idea\workspace.xml”文件。

步骤2 找到标签“<component name="PropertiesComponent"> ”，在内容中添加“ <property name="dynamic.classpath" value="true" />”，如图29-26。

图 29-26 修改“ .idea\workspace.xml”文件

----结束

30 Storm 开发指南（普通模式）

30.1 Storm 应用开发概述

30.1.1 Storm 应用开发简介

简介

Storm是一个分布式的、可靠的、容错的数据流处理系统。它会把工作任务委托给不同类型的组件，每个组件负责处理一项简单特定的任务。Storm的目标是提供对大数据流的实时处理，可以可靠地处理无限的数据流。

Storm有很多适用的场景：实时分析、在线机器学习、持续计算和分布式ETL等，易扩展、支持容错，可确保数据得到处理，易于构建和操控。

Storm有如下几个特点：

- **适用场景广泛**
- **易扩展，可伸缩性高**
- **保证无数据丢失**
- **容错性好**
- **多语言**
- **易于构建和操控**

30.1.2 Storm 应用开发常用概念

Topology

拓扑是一个计算流图。其中每个节点包含处理逻辑，而节点间的连线则表明了节点间的数据是如何流动的。

Spout

在一个Topology中产生源数据流的组件。通常情况下Spout会从外部数据源中读取数据，然后转换为Topology内部的源数据。

Bolt

在一个Topology中接受数据然后执行处理的组件。Bolt可以执行过滤、函数操作、合并、写数据库等任何操作。

Tuple

一次消息传递的基本单元。

Stream

流是一组（无穷）元素的集合，流上的每个元素都属于同一个schema；每个元素都和逻辑时间有关；即流包含了元组和时间的双重属性。流上的任何一个元素，都可以用Element<tuple,Time>的方式来表示，tuple是元组，包含了数据结构和数据内容，Time就是该数据的逻辑时间。

30.1.3 Storm 应用开发流程

本文档主要基于Java API进行Storm拓扑的开发。

开发流程中各阶段的说明如**图30-1**和**表30-1**所示：

图 30-1 拓扑开发流程

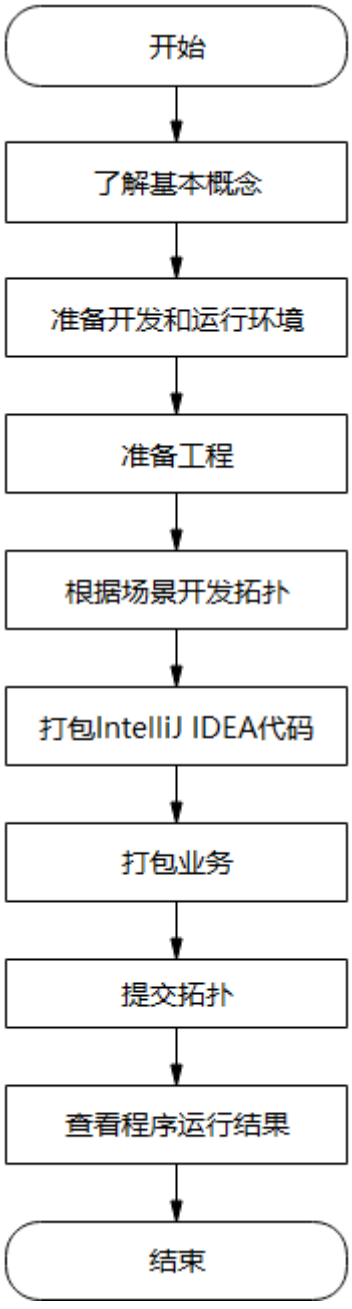


表 30-1 Storm 应用开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发应用前，需要了解Storm的基本概念，了解场景需求，拓扑等。	Storm应用开发常用概念

阶段	说明	参考文档
准备开发和运行环境	Storm的应用程序当前推荐使用Java语言进行开发。可使用IntelliJ IDEA工具。 Storm的运行环境即Storm客户端，请根据指导完成客户端的安装和配置。	**准备Storm应用开发和运行环境**
准备工程	Storm提供了不同场景下的样例程序，您可以导入样例工程进行程序学习。	**导入并配置Storm样例工程**
根据场景开发拓扑	提供了Storm拓扑的构造和Spout/Bolt开发过程。	**开发Storm应用**
打包IntelliJ IDEA代码	Storm样例程序是在Linux环境下运行，需要将IntelliJ IDEA中的代码打包成jar包。	**打包Strom样例工程应用**
打包业务	将IntelliJ IDEA代码生成的jar包与工程依赖的jar包，合并导出可提交的source.jar。	**打包Strom应用业务**
提交拓扑	指导用户将开发好的程序提交运行。	**提交Storm拓扑**
查看程序运行结果	指导用户提交拓扑后查看程序运行结果。	**查看Storm应用调测结果**

30.2 准备 Storm 应用开发环境

30.2.1 准备 Storm 应用开发和运行环境

开发环境准备分为应用开发客户端和应用提交客户端；应用开发一般是在Windows环境下进行；应用提交一般是在Linux环境下进行。

准备开发环境

在进行二次开发时，要准备的开发和运行环境如**表30-2**所示：

表 30-2 开发环境

准备项	说明
操作系统	• 开发环境：Windows系统，支持Windows 7以上版本。 • 运行环境：Windows系统或Linux系统。 如需在本地调测程序，运行环境需要和集群业务平面网络互通。
安装JDK	开发和运行环境的基本配置。版本要求如下： 服务端和客户端仅支持自带的OpenJDK，版本为1.8.0_272，不允许替换。 对于客户应用需引用SDK类的Jar包运行在客户应用进程中的。 • X86客户端： – Oracle JDK：支持1.8版本 – IBM JDK：支持1.8.5.11版本 • TaiShan客户端： – OpenJDK：支持1.8.0_272版本 **说明** 基于安全考虑，服务端只支持TLS V1.2及以上的加密协议。 IBM JDK默认只支持TLS V1.0，若使用IBM JDK，请配置启动参数“com.ibm.jsse2.overrideDefaultTLS”为“true”，设置后可以同时支持TLS V1.0/V1.1/V1.2，详情参见**https://www.ibm.com/support/knowledgecenter/zh/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/matchsslcontext_tls.html#matchsslcontext_tls**。
安装和配置IntelliJ IDEA	用于开发Storm应用程序的工具。版本要求：JDK使用1.8版本，IntelliJ IDEA使用2019.1或其他兼容版本。 **说明** • 若使用IBM JDK，请确保IntelliJ IDEA中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保IntelliJ IDEA中的JDK配置为Oracle JDK。 • 若使用Open JDK，请确保IntelliJ IDEA中的JDK配置为Open JDK。
7-zip	用于解压“*.zip”和“*.rar”文件，支持7-Zip 16.04版本。

准备运行环境

进行应用开发时，需要同时准备代码的运行调测的环境，用于验证应用程序运行正常。

- 如果本地Windows开发环境和集群业务平面网络互通，可下载集群客户端到本地，获取相关调测程序所需的集群配置文件及配置网络连通后，然后直接在Windows中进行程序调测。
  a. **登录FusionInsight Manager页面**，选择“集群 > 概览 > 更多 > 下载客户端”，“选择客户端类型”设置为“完整客户端”，根据待安装客户端节点的节点类型选择正确的平台类型后（x86选择**x86_64**，ARM选择**aarch64**）单击“确定”，等待客户端文件包生成后根据浏览器提示下载客户端到本地并解压。
  例如，客户端文件压缩包为“FusionInsight_Cluster_1_Services_Client.tar”，解压后得到

“FusionInsight_Cluster_1_Services_ClientConfig.tar”，继续解压该文件。解压到本地PC的“D:\FusionInsight_Cluster_1_Services_ClientConfig”目录下（路径中不能有空格）。

b. 进入客户端解压路径“FusionInsight_Cluster_1_Services_ClientConfig\Storm\config”，获取相关配置文件。

主要配置文件说明如**表30-3**所示。

表 30-3 配置文件

文件名称	作用
storm.yaml	配置Storm集群信息。
streaming-site.xml	配置Storm详细参数。

c. 在应用开发过程中，如需在本地Windows系统中调测应用程序，需要复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与解压目录下“hosts”文件中所列出的各主机在网络上互通。

说明

- 当客户端所在主机不是集群中的节点时，配置客户端网络连接，可避免执行客户端命令时出现错误。
- Windows本地hosts文件存放路径举例：“C:\WINDOWS\system32\drivers\etc\hosts”。

- 使用Linux环境调测程序，需准备安装集群客户端的Linux节点并获取相关配置文件。

a. 在节点中安装客户端，例如客户端安装目录为“/opt/client”。

客户端机器的时间与集群的时间要保持一致，时间差小于5分钟。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

b. **登录FusionInsight Manager页面**，下载集群客户端软件包至主管理节点并解压，然后以**root**用户登录主管理节点，进入集群客户端解压路径下，复制“FusionInsight_Cluster_1_Services_ClientConfig/Storm/config”路径下的所有配置文件至客户端节点，放置到与准备放置编译出的jar包同目录的“conf”目录下，用于后续调测，例如“/opt/client/conf”。

例如客户端软件包为“FusionInsight_Cluster_1_Services_Client.tar”，下载路径为主管理节点的“/tmp/FusionInsight-Client”：

cd /tmp/FusionInsight-Client

tar -xvf FusionInsight_Cluster_1_Services_Client.tar

tar -xvf FusionInsight_Cluster_1_Services_ClientConfig.tar

cd FusionInsight_Cluster_1_Services_ClientConfig

scp Storm/config/* root@*客户端节点IP地址*:/opt/client/conf

主要配置文件说明如**表30-4**所示。

表 30-4 配置文件

文件名称	作用
storm.yaml	配置Storm集群信息。
streaming-site.xml	配置Storm详细参数。

c. 检查客户端节点网络连接。

在安装客户端过程中，系统会自动配置客户端节点“hosts”文件，建议检查“/etc/hosts”文件内是否包含集群内节点的主机名信息，如未包含，需要手动复制解压目录下的“hosts”文件中的内容到客户端所在节点的hosts文件中，确保本地机器能与集群各主机在网络上互通。

30.2.2 导入并配置 Storm 样例工程

背景信息

Storm客户端安装程序目录中包含了Storm开发样例工程，将工程导入到IntelliJ IDEA开始样例学习。

前提条件

确保本地PC的时间与集群的时间差要小于5分钟，若无法确定，请联系系统管理员。集群的时间可通过FusionInsight Manager页面右下角查看。

操作步骤

步骤1 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中的“src\storm-examples”目录下的“storm-examples”样例工程文件夹。

步骤2 将**准备运行环境**时获取的配置文件放置在样例工程的“storm-examples\src\main\resources”目录下。

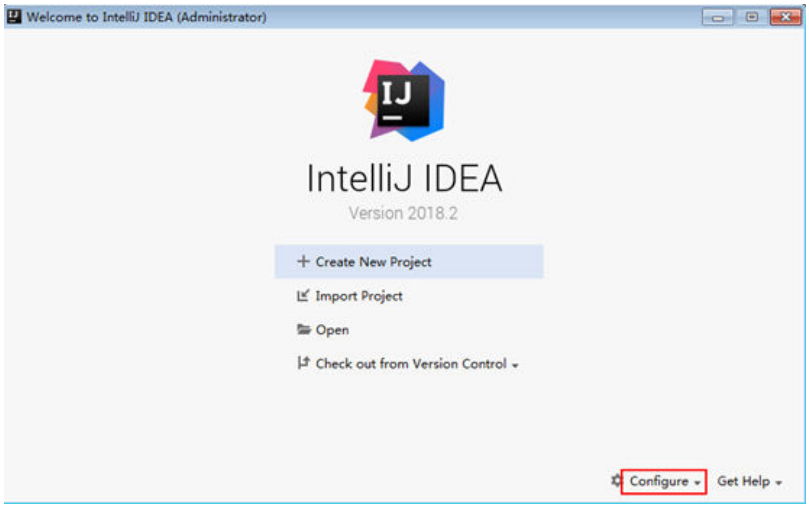
说明

- 若要在Windows或Linux中未安装客户端时提交拓扑，则需要将“streaming-site.xml”和“storm.yaml”都放入样例工程的“storm-examples\src\main\resources”目录下。
- 若要在Linux安装客户端时提交拓扑，只需要将“streaming-site.xml”放入样例工程的“storm-examples\src\main\resources”目录下即可。

步骤3 安装IntelliJ IDEA和JDK工具后，需要在IntelliJ IDEA配置JDK。

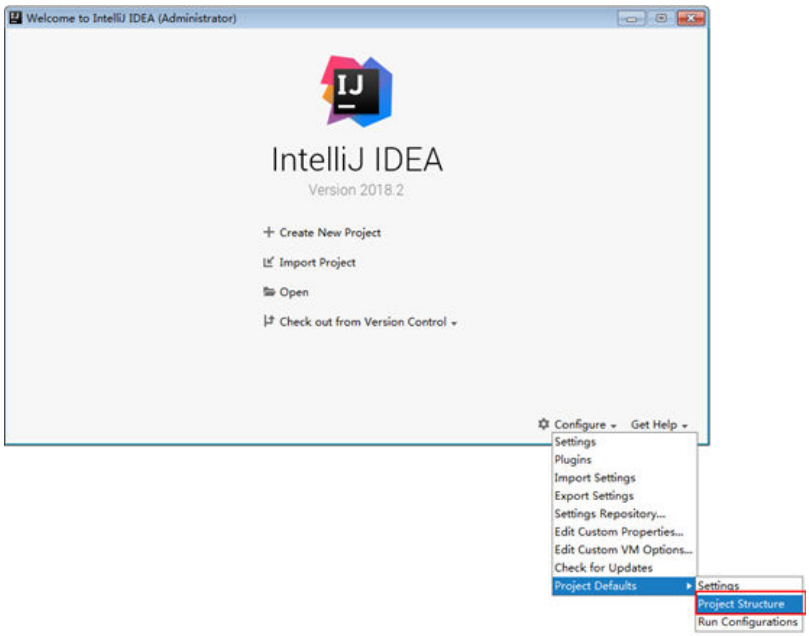
1. 打开IntelliJ IDEA，选择“Configure”。

图 30-2 Quick Start



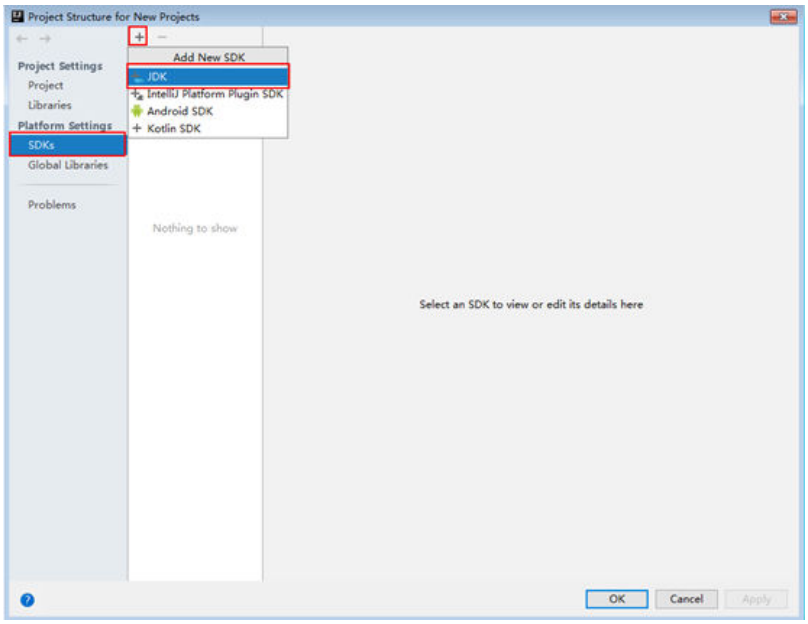
2. 在下拉框中选择“Project Defaults->Project Structure”。

图 30-3 Configure



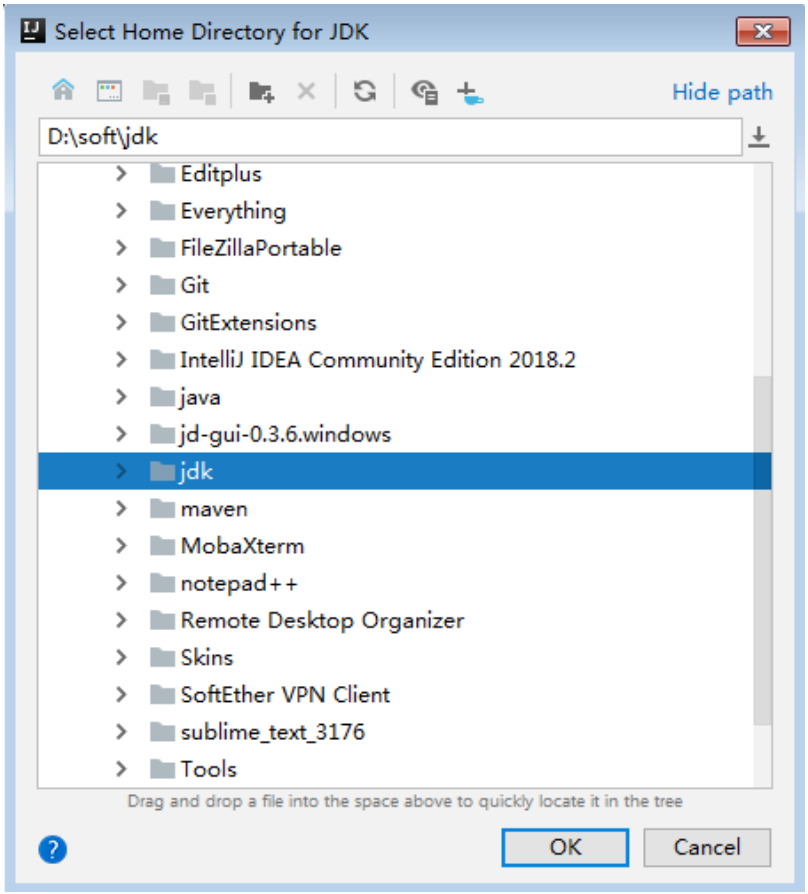
3. 在弹出的“Project Structure for New Projects”页面中，选择“SDKs”，单击加号添加JDK。

图 30-4 Project Structure for New Projects



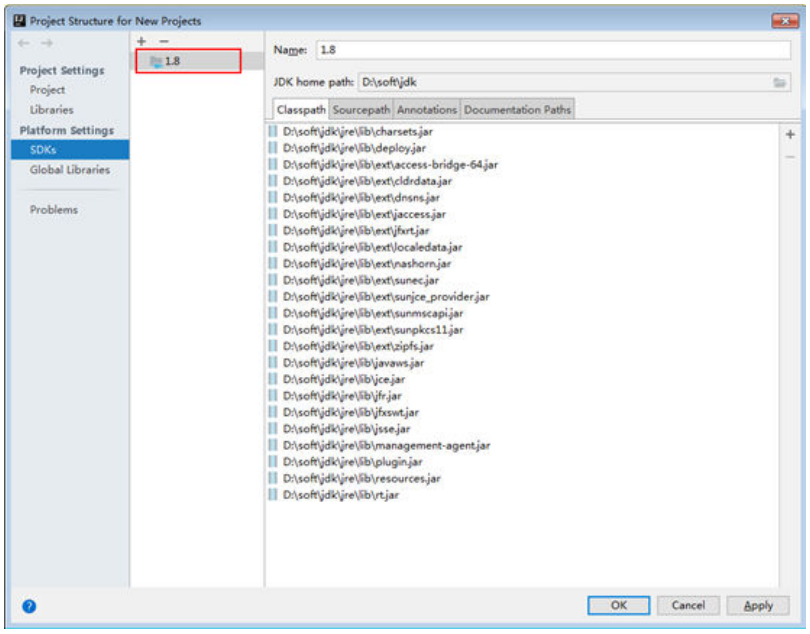
4. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 30-5 Select Home Directory for JDK



5. 完成JDK选择后，单击“OK”完成配置。

图 30-6 完成 JDK 配置

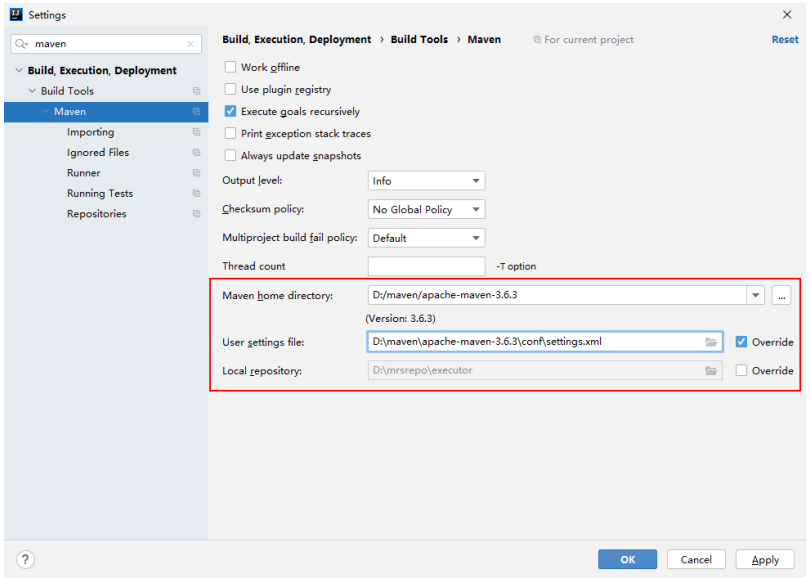


步骤4 在应用开发环境中，导入样例工程到IntelliJ IDEA开发环境。

1. 配置IntelliJ IDEA maven工程环境

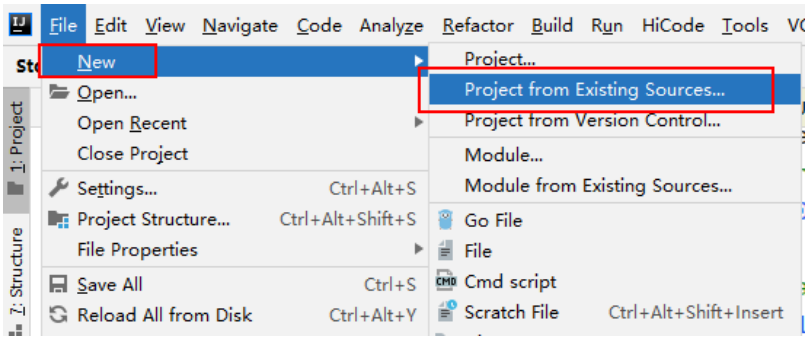
   选择“File > Settings”，搜索“maven”，配置maven工程，选择“Apply > OK”

图 30-7 配置 IntelliJ IDEA maven 工程环境



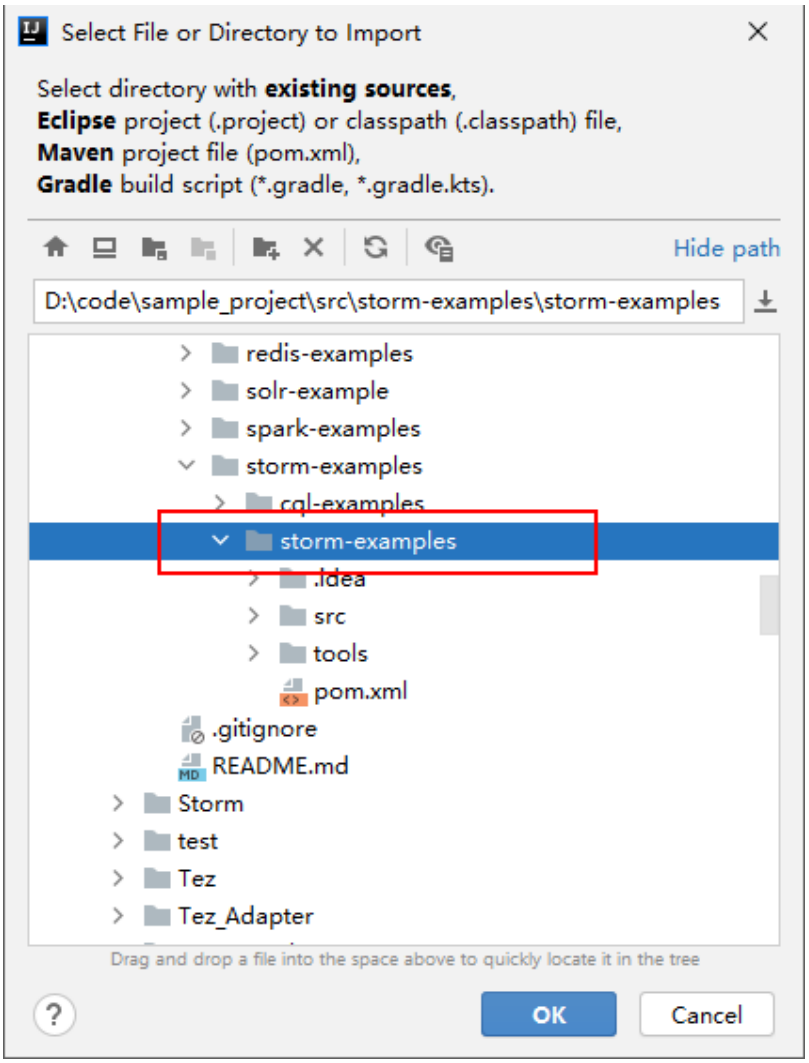
2. 选择“File > New > Project from Existing Sources…”

图 30-8 进入“Project from Existing Sources....”配置页面



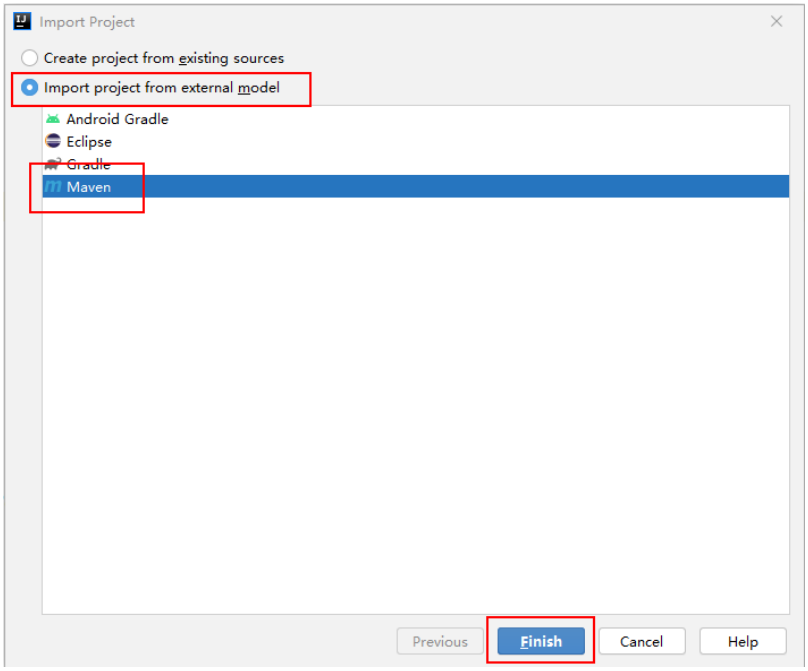
3. 选择要导入的样例工程，例如storm-examples。

图 30-9 选择要导入的样例工程



4. 选择以maven工程的形式导入。

图 30-10 以 maven 工程的形式导入



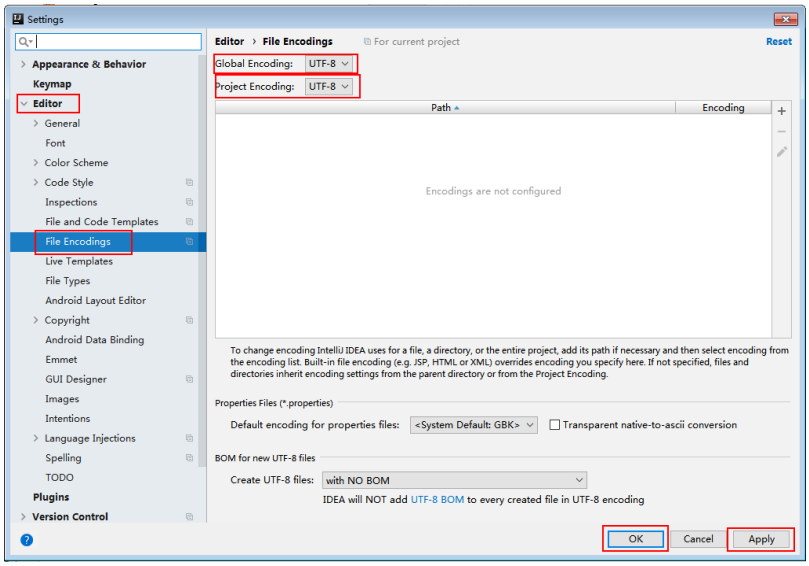
步骤5 设置IntelliJ IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IntelliJ IDEA的菜单栏中，选择“File > Settings”。

   弹出“Settings”窗口。

2. 在左边导航上选择“Editor > File Encodings”，在“Project Encoding”和“Global Encoding”区域，设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图30-11所示。

图 30-11 设置 IntelliJ IDEA 的编码格式



----结束

30.3 开发 Storm 应用

30.3.1 Storm 样例程序开发思路

通过典型场景，用户可以快速学习和掌握Storm拓扑的构造和Spout/Bolt开发过程。

场景说明

一个动态单词统计系统，数据源为持续生产随机文本的逻辑单元，业务处理流程如下：

- 数据源持续不断地发送随机文本给文本拆分逻辑，如“apple orange apple”。
- 单词拆分逻辑将数据源发送的每条文本按空格进行拆分，如“apple”，“orange”，“apple”，随后将每个单词逐一发给单词统计逻辑。
- 单词统计逻辑每收到一个单词就进行加一操作，并将实时结果打印输出，如：

  apple：1

  orange：1

  apple：2

功能分解

根据上述场景进行功能分解，如表30-5所示：

表 30-5 在应用中开发的功能

序号	步骤	代码示例
1	创建一个Spout用来生成随机文本	请参见**创建Strom Spout**
2	创建一个Bolt用来将收到的随机文本拆分成一个个单词	请参见**创建Strom Bolt**
3	创建一个Blot用来统计收到的各单词次数	请参见**创建Strom Bolt**
4	创建topology	请参见**创建Strom Topology**

部分代码请参考**开发Storm应用**，完整代码请参考Strom-examples示例工程。

30.3.2 创建 Strom Spout

功能介绍

Spout是Storm的消息源，它是Topology的消息生产者，一般来说消息源会从一个外部源读取数据并向Topology中发送消息（Tuple）。

一个消息源可以发送多条消息流Stream，可以使用OutputFieldsDeclarer.declarerStream来定义多个Stream，然后使用SpoutOutputCollector来发射指定的Stream。

代码样例

下面代码片段在com.huawei.storm.example.common包的“RandomSentenceSpout”类的“nextTuple”方法中，作用在于将收到的字符串拆分成单词。

```
/**
  * {@inheritDoc}
  */
 @Override
 public void nextTuple()
 {
     Utils.sleep(100);
     String[] sentences =
         new String[] {"the cow jumped over the moon",
                       "an apple a day keeps the doctor away",
                       "four score and seven years ago",
                       "snow white and the seven dwarfs",
                       "i am at two with nature"};
     String sentence = sentences[random.nextInt(sentences.length)];
     collector.emit(new Values(sentence));
 }
```

30.3.3 创建 Strom Bolt

功能介绍

所有的消息处理逻辑都被封装在各个Bolt中。Bolt包含多种功能：过滤、聚合等等。

如果Bolt之后还有其他拓扑算子，可以使用OutputFieldsDeclarer.declareStream定义Stream，使用OutputCollector.emit来选择要发射的Stream。

代码样例

下面代码片段在com.huawei.storm.example.common包的“SplitSentenceBolt”类的“execute”方法中，作用在于拆分每条语句为单个单词并发送。

```
/**
  * {@inheritDoc}
  */
 @Override
 public void execute(Tuple input, BasicOutputCollector collector)
 {
     String sentence = input.getString(0);
     String[] words = sentence.split(" ");
     for (String word : words)
     {
         word = word.trim();
         if (!word.isEmpty())
         {
             word = word.toLowerCase();
             collector.emit(new Values(word));
         }
     }
 }
```

下面代码片段在com.huawei.storm.example.wordcount.WordCountBolt类中，作用在于统计收到的每个单词的数量。


```
@Override
 public void execute(Tuple tuple, BasicOutputCollector collector)
 {
     String word = tuple.getString(0);
     Integer count = counts.get(word);
     if (count == null)
     {
         count = 0;
     }
     count++;
     counts.put(word, count);
     System.out.println("word: " + word + ", count: " + count);
 }
```

30.3.4 创建 Strom Topology

功能介绍

一个Topology是Spouts和Bolts组成的有向无环图。

应用程序是通过storm jar的方式提交，则需要在main函数中调用创建Topology的函数，并在storm jar参数中指定main函数所在类。

代码样例

下面代码片段在com.huawei.storm.example.wordcount包的“WordCountTopology”类的“main”方法中，作用在于构建应用程序并提交。

```
public static void main(String[] args)
        throws Exception
 {
     TopologyBuilder builder = buildTopology();

     /*
      * 任务的提交认为三种方式
      * 1、命令行方式提交，这种需要将应用程序jar包复制到客户端机器上执行客户端命令提交
      * 2、远程方式提交，这种需要将应用程序的jar包打包好之后在IntelliJ IDEA中运行main方法提交
      * 3、本地提交 ，在本地执行应用程序，一般用来测试
      * 命令行方式和远程方式安全和普通模式都支持
      * 本地提交仅支持普通模式
      *
      * 用户同时只能选择一种任务提交方式，默认命令行方式提交，如果是其他方式，请删除代码注释即可
      */

     submitTopology(builder, SubmitType.CMD);

 }

 private static void submitTopology(TopologyBuilder builder, SubmitType type) throws Exception
 {
    switch (type)
    {
        case CMD:
        {
            cmdSubmit(builder, null);
            break;
        }
        case REMOTE:
        {
            remoteSubmit(builder);
            break;
        }
        case LOCAL:
        {
            localSubmit(builder);
            break;
```



```
        }
      }
    }

    /**
     * 命令行方式远程提交
     * 步骤如下：
     * 打包成Jar包，然后在客户端命令行上面进行提交
     *   远程提交的时候，要先将该应用程序和其他外部依赖(非excemple工程提供，用户自己程序依赖)的jar包打包成一个大的jar包
     *   再通过storm客户端中storm -jar的命令进行提交
     *
     * 如果是安全环境，客户端命令行提交之前，必须先通过kinit命令进行安全登录
     *
     * 运行命令如下：
     * ./storm jar ../example/example.jar com.huawei.storm.example.WordCountTopology
     */
    private static void cmdSubmit(TopologyBuilder builder, Config conf)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException,
AuthorizationException
    {
        if (conf == null)
        {
            conf = new Config();
        }
        conf.setNumWorkers(1);

        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, conf, builder.createTopology());
    }

     private static void localSubmit(TopologyBuilder builder)
         throws InterruptedException
     {
         Config conf = new Config();
         conf.setDebug(true);
         conf.setMaxTaskParallelism(3);
         LocalCluster cluster = new LocalCluster();
         cluster.submitTopology(TOPOLOGY_NAME, conf, builder.createTopology());
         Thread.sleep(10000);
         cluster.shutdown();
     }


    private static void remoteSubmit(TopologyBuilder builder)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException,
AuthorizationException,
        IOException
    {
        Config config = createConf();

        String userJarFilePath = "替换为用户jar包地址";
        System.setProperty(STORM_SUBMIT_JAR_PROPERTY, userJarFilePath);

        //安全模式下的一些准备工作
        if (isSecurityModel())
        {
            securityPrepare(config);
        }
        config.setNumWorkers(1);
        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, config,
builder.createTopology());
    }

      private static TopologyBuilder buildTopology()
     {
         TopologyBuilder builder = new TopologyBuilder();
         builder.setSpout("spout", new RandomSentenceSpout(), 5);
         builder.setBolt("split", new SplitSentenceBolt(), 8).shuffleGrouping("spout");
         builder.setBolt("count", new WordCountBolt(), 12).fieldsGrouping("split", new Fields("word"));
```



```
        return builder;
    }
```

说明

如果拓扑开启了ack，推荐acker的数量不大于所设置的worker数量。

30.4 调测 Strom 应用

30.4.1 打包 Strom 样例工程应用

操作场景

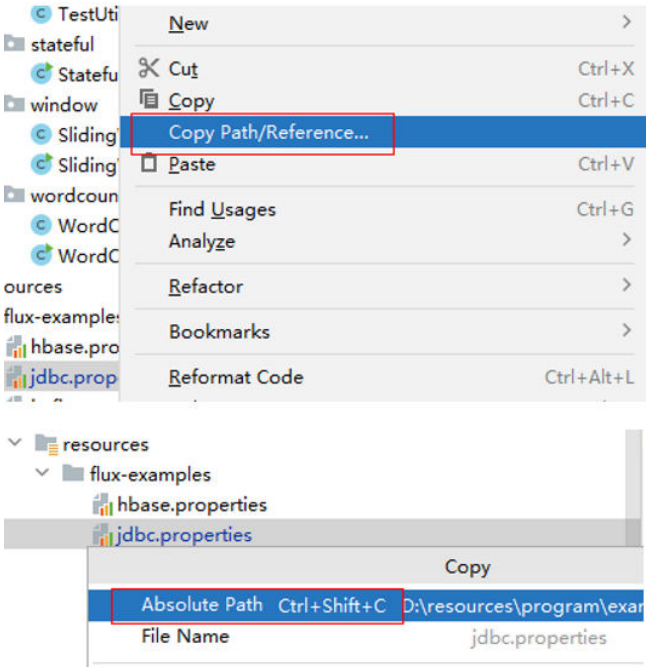
通过IntelliJ IDEA导出Jar包并指定导出jar包名称，比如“storm-examples.jar”。

操作步骤

步骤1 若Storm-JDBC样例需要在Windows下运行，则需要替换配置文件路径；否则，不需要执行此步骤。

1. 在IDEA界面右键单击“jdbc.properties”文件，选择“Copy Path/Reference > Absolute Path”，复制“jdbc.properties”文件路径。

图 30-12 复制“jdbc.properties”文件路径



2. 修改“SimpleJDBCTopology.java”的**main()**方法中**proPath**值为步骤1.1复制的“jdbc.properties”文件路径。

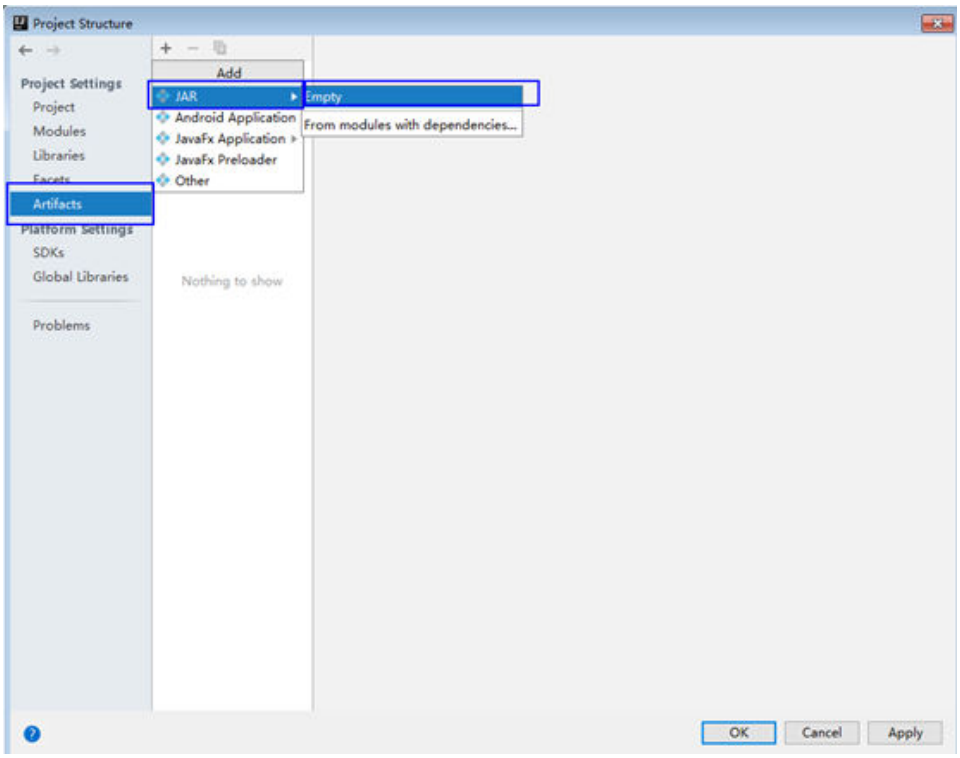
图 30-13 SimpleJDBCTopology.java 路径替换

```
public static void main(String[] args) throws Exception {
    // 获取配置文件
    Properties properties = new Properties();
    String proPath ="D:\\resources\\program\\example\\sample_project\\src\\storm-examples\\storm-examples\\src\\main\\resources\\flux-examples\\jdbc.properties"
    try {
        properties.load(new FileInputStream(proPath));
    } catch (IOException e) {
        logger.error("Failed to load properties file.");
        throw e;
    }
```

步骤2 在IntelliJ IDEA中，在生成Jar包之前配置工程的Artifacts信息。

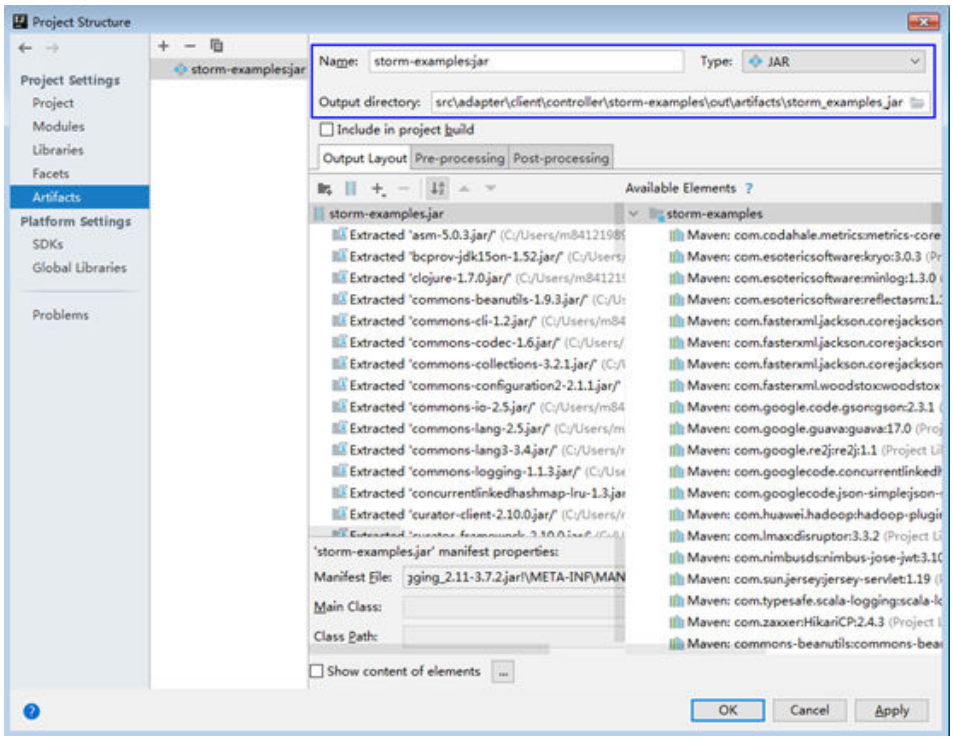
1. 进入IntelliJ IDEA，选择“File > Project Structures...”进入“Project Structure”页面。
2. 在“Project Structure”页面，选择“Artifacts”，单击“+”并选择“JAR > Empty”。

图 30-14 添加 Artifacts



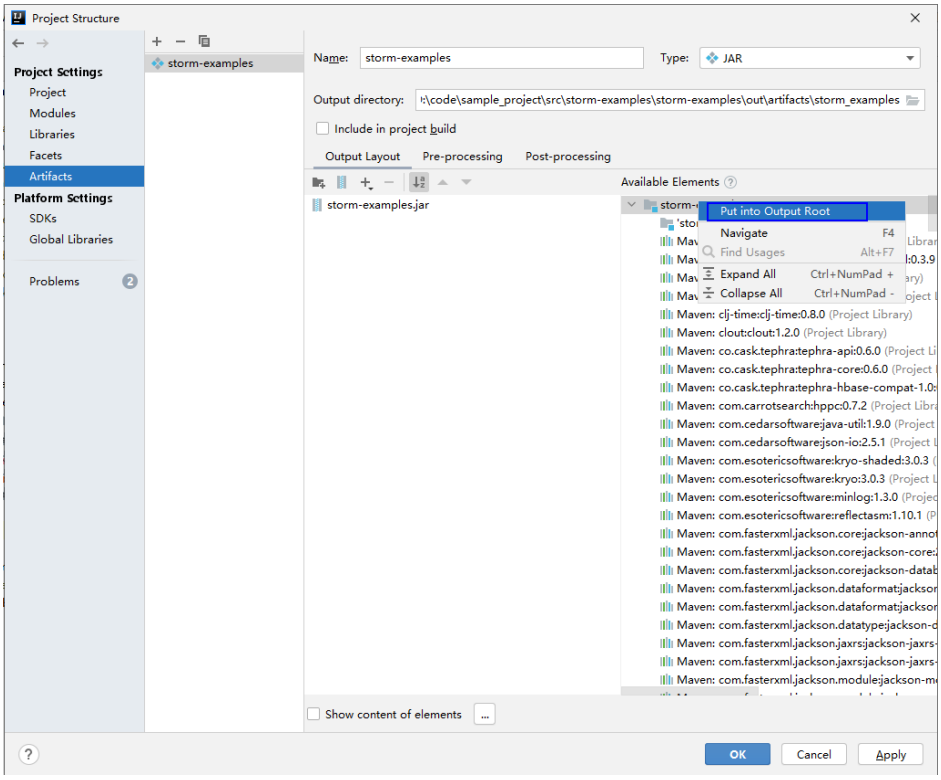
3. 根据实际情况设置Jar包的名称、类型以及输出路径。

图 30-15 设置基本信息



4. 选中“storm-examples”，右键选择“Put into Output Root”。然后单击“Apply”。

图 30-16 Put into Output Root

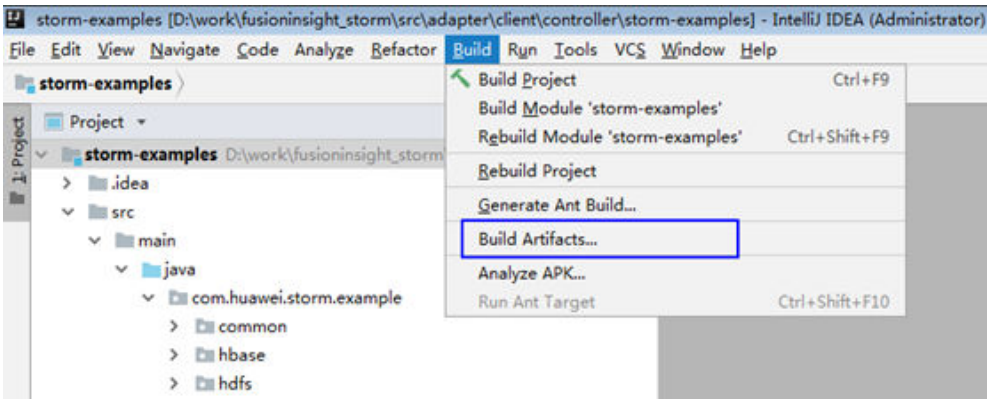


5. 最后单击“OK”完成配置。

步骤3 生成Jar包。

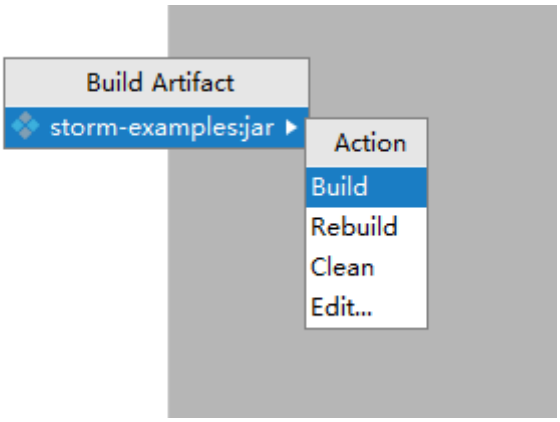
1. 在IntelliJ IDEA中，选择“Build > Build Artifacts...”。

   图 30-17 Build Artifacts

2. 在弹出的菜单中，选择“storm-examples:jar > Build”开始生成Jar包。

   图 30-18 Build

3. 当Event log中出现如下类似日志时，表示Jar包生成成功。您可以在图30-15中配置的路径下获取到Jar包。

   ```
   14:37 Compilation completed successfully in 25 s 991 ms
   ```

----结束

30.4.2 打包 Strom 应用业务

30.4.2.1 Linux 下打包 Storm 业务

操作场景

Storm支持在Linux环境下打包。用户可以将从IntelliJ IDEA中导出的Jar包和需要的其他相关Jar包上传到Linux环境中执行打包。

前提条件

- 已安装Storm客户端。

- 已执行**打包Strom样例工程应用**。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

操作步骤

步骤1 将从IntelliJ IDEA中导出的jar包复制到Linux客户端指定目录（例如“/opt/jarsource”）。

步骤2 若业务需要访问外部组件，其所依赖的配置文件请参考相关开发指引，获取到配置文件后将配置文件放在**步骤1**中指定的目录下。

步骤3 若业务需要访问外部组件，其所依赖的jar包请参考相关开发指引，获取到jar包后将jar包放在**步骤1**中指定的目录下。

步骤4 在Storm客户端安装目录“Storm/storm-1.2.1/bin”下执行打包命令，将上述jar包打成一个完整的业务jar包放入指定目录/opt/jartarget（可为任意空目录）。执行**sh storm-jartool.sh /opt/jarsource/ /opt/jartarget**命令后，会在“/opt/jartarget”下生成source.jar。

----结束

30.4.2.2 Windows 下打包 Storm 业务

操作场景

Storm支持在Windows环境下打包。

打包业务的目的，是将IntelliJ IDEA代码生成的jar包与工程依赖的jar包，合并导出可提交的source.jar。

打包需使用storm-jartool工具，可在Windows或Linux上进行。

前提条件

已执行**打包Strom样例工程应用**。

操作步骤

步骤1 将从IntelliJ IDEA打包出来的jar包放入指定文件夹（例如“D:\source”）。

步骤2 在样例代码目录“src/storm-examples/storm-examples”下创建“lib”目录，将IntelliJ IDEA中导出的jar包复制到“lib”目录下，并解压。

步骤3 若业务需要访问外部组件，其所依赖的配置文件请参考相关开发指引，获取到配置文件后将配置文件放在**步骤1**中指定的目录下。

步骤4 若业务需要访问外部组件，其所依赖的jar包请参考相关开发指引，获取到jar包后将jar包放在**步骤1**中指定的目录下。

步骤5 在IntelliJ IDEA样例工程的“tools”目录下找到打包工具：“storm-jartool.cmd”。

步骤6 双击打包工具，输入要打包的jar包所在目录（“D:\source”）并回车，再输入打出包存放的目录（“D:\target”），在“D:\target”中，会生成“source.jar”文件。

----结束

30.4.3 提交 Storm 拓扑

30.4.3.1 Linux 中安装客户端时提交 Storm 拓扑

操作场景

在Linux环境下可以使用storm命令行完成拓扑的提交。

前提条件

- 已安装Storm客户端。
- 当客户端所在主机不是集群中的节点时，需要在客户端所在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。
- 已执行**打包Strom应用业务**步骤，打出source.jar。

操作步骤

步骤1 提交拓扑（以wordcount为例，其它拓扑请参照相关开发指引），进入Storm客户端“Storm/storm-1.2.1/bin”目录，将刚打出的source.jar提交（如果在Windows上进行的打包，则需要将Windows上的source.jar上传到Linux服务器，假定上传到“/opt/jartarget”目录），执行命令：**storm jar /opt/jartarget/source.jar com.huawei.storm.example.wordcount.WordCountTopology**

步骤2 执行**storm list**命令，查看已经提交的应用程序，如果发现名称为word-count的应用程序，则说明任务提交成功。

说明

如果业务设置为本地模式，且使用命令行方式提交时，请确保提交环境为普通模式环境，当前不支持安全环境下使用命令提交本地模式的业务。

----结束

30.4.3.2 Linux 中未安装客户端时提交 Storm 拓扑

操作场景

Storm支持拓扑在未安装Storm客户端的Linux环境中运行。

前提条件

- 客户端机器的时间与MRS集群的时间要保持一致，时间差要小于5分钟。
- 当Linux环境所在主机不是集群中的节点时，需要在节点的**hosts**文件中设置主机名和IP地址映射。主机名和IP地址请保持一一对应。

操作步骤

步骤1 准备依赖的Jar包和配置文件。

在Linux环境新建目录，例如“/opt/test”，并创建子目录“lib”和“src/main/resources/”。将样例工程中“lib”文件夹下的Jar包上传Linux环境的“lib”目录。将样例工程中“src/main/resources”文件夹下的配置文件上传到Linux环境的“src/main/resources”目录。

步骤2 在IntelliJ IDEA工程中修改WordCountTopology.java类，使用remoteSubmit方式提交应用程序。并替换Jar文件地址。

- 使用remoteSubmit方式提交应用程序

```
public static void main(String[] args)
        throws Exception
    {
        TopologyBuilder builder = buildTopology();

        /*
         * 任务的提交认为三种方式
         * 1、命令行方式提交，这种需要将应用程序jar包复制到客户端机器上执行客户端命令提交
         * 2、远程方式提交，这种需要将应用程序的jar包打包好之后在IntelliJ IDEA中运行main方法提交
         * 3、本地提交 ，在本地执行应用程序，一般用来测试
         * 命令行方式和远程方式安全和普通模式都支持
         * 本地提交仅支持普通模式
         *
         * 用户同时只能选择一种任务提交方式，默认命令行方式提交，如果是其他方式，请删除代码注释即
可
         */

        submitTopology(builder, SubmitType.REMOTE);
    }
```

- 修改userJarFilePath为Linux环境指定路径“/opt/test/lib/example.jar”。

```
private static void remoteSubmit(TopologyBuilder builder)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException,
AuthorizationException,
        IOException
    {
        Config config = createConf();

        String userJarFilePath = "/opt/test/lib/example.jar ";
        System.setProperty(STORM_SUBMIT_JAR_PROPERTY, userJarFilePath);

        //安全模式下的一些准备工作
        if (isSecurityModel())
        {
            securityPrepare(config);
        }
        config.setNumWorkers(1);
        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, config,
builder.createTopology());
    }
```

步骤3 导出Jar包并上传到Linux环境。

- 参考**打包Strom样例工程应用**执行打包，并将jar包命名为“example.jar”。
- 将导出的Jar包复制到Linux环境的“/opt/test/lib”目录下。

步骤4 切换到“/opt/test”，执行以下命令，运行Jar包。

java -classpath /opt/test/lib/*:/opt/test/src/main/resources
com.huawei.storm.example.wordcount.WordCountTopology

----结束

30.4.3.3 在 IDEA 中提交 Storm 拓扑

操作场景

Storm支持IntelliJ IDEA远程提交拓扑，目前样例代码中仅WordCountTopology支持远程提交，其他拓扑想实现远程提交，请参考WordCountTopology实现远程提交函数。

前提条件

- 已执行**打包Strom样例工程应用**。
- 调整IntelliJ IDEA客户端机器时间，和Storm集群时间差不超过5分钟。
- 确保本地的hosts文件中配置了远程集群所有主机的主机名和业务IP映射关系。

操作步骤

步骤1 修改WordCountTopology.java类，使用remoteSubmit方式提交应用程序。并替换Jar文件地址。

- 使用remoteSubmit方式提交应用程序

```
public static void main(String[] args)
        throws Exception
    {
        TopologyBuilder builder = buildTopology();

        /*
         * 任务的提交认为三种方式
         * 1、命令行方式提交，这种需要将应用程序jar包复制到客户端机器上执行客户端命令提交
         * 2、远程方式提交，这种需要将应用程序的jar包打包好之后在IntelliJ IDEA中运行main方法提交
         * 3、本地提交 ，在本地执行应用程序，一般用来测试
         * 命令行方式和远程方式安全和普通模式都支持
         * 本地提交仅支持普通模式
         *
         * 用户同时只能选择一种任务提交方式，默认命令行方式提交，如果是其他方式，请删除代码注释即
可
         */

        submitTopology(builder, SubmitType.REMOTE);
    }
```

- 根据实际情况修改userJarFilePath为实际的拓扑Jar包地址

```
private static void remoteSubmit(TopologyBuilder builder)
        throws AlreadyAliveException, InvalidTopologyException, NotALeaderException,
AuthorizationException,
        IOException
    {
        Config config = createConf();

        String userJarFilePath = "D:\\example.jar";
        System.setProperty(STORM_SUBMIT_JAR_PROPERTY, userJarFilePath);

        //安全模式下的一些准备工作
        if (isSecurityModel())
        {
            securityPrepare(config);
        }
        config.setNumWorkers(1);
        StormSubmitter.submitTopologyWithProgressBar(TOPOLOGY_NAME, config,
builder.createTopology());
    }
```

步骤2 执行WordCountTopology.java类的Main方法提交应用程序。

----结束

30.4.4 查看 Storm 应用调测结果

操作场景

Storm应用程序运行完成后，可通过登录Storm WebUI查看应用程序的运行情况。

操作步骤

步骤1 登录FusionInsight Manager系统。

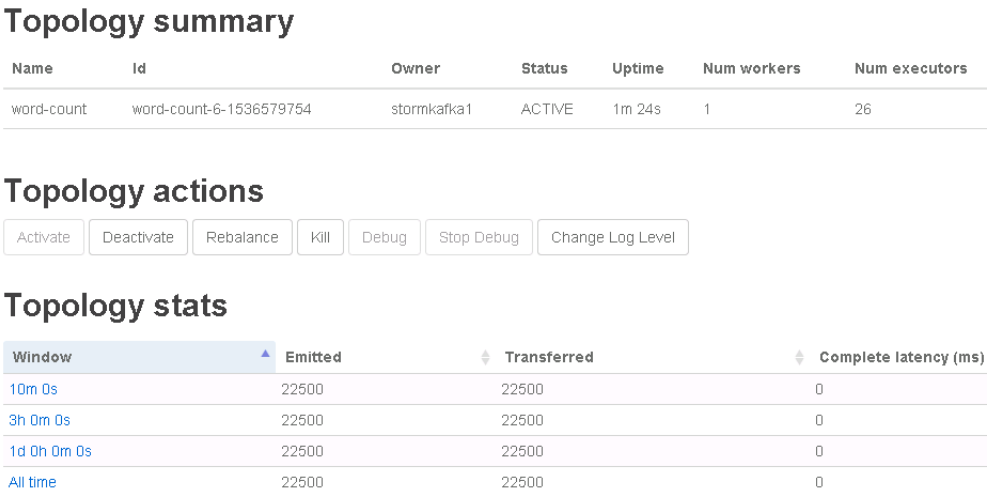
在浏览器地址栏中输入访问地址，地址格式为“https://*FusionInsight Manager系统的WebService浮动IP地址*:28443/web”。

例如，在IE浏览器地址栏中，输入“https://10.0.0.1:28443/web”。

步骤2 选择“集群 > *待操作集群的名称* > 服务 > Storm”，单击进入Storm WebUI。

步骤3 在Storm UI中单击word-count应用，查看应用程序运行情况，如图30-19所示。

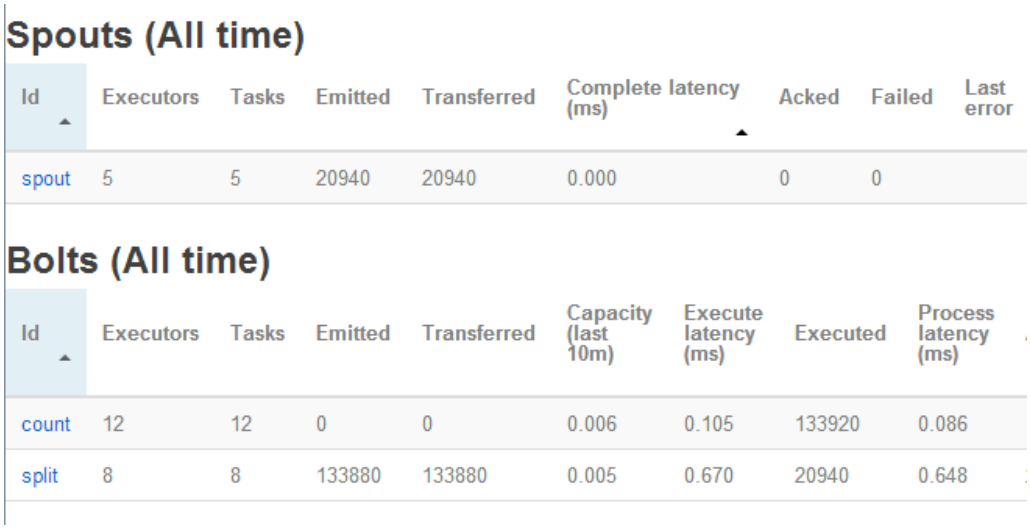
图 30-19 Storm 应用程序执行界面



Topology stats统计了最近各个不同时间段的算子之间发送数据的总数据量。

Spouts中统计了spout算子从启动到现在发送的消息总量。Bolts中统计了Count算子和split算子的发送消息总量，如图30-20所示。

图 30-20 Storm 应用程序算子发送数据总量



----结束

30.5 Storm 应用开发常见问题

30.5.1 Storm-Kafka 开发指引

操作场景

本文档主要说明如何使用Storm-Kafka工具包，完成Storm和Kafka之间的交互。包含KafkaSpout和KafkaBolt两部分。KafkaSpout主要完成Storm从Kafka中读取数据的功能；KafkaBolt主要完成Storm向Kafka中写入数据的功能。

本章节代码样例基于Kafka新API，对应IntelliJ IDEA工程中com.huawei.storm.example.kafka.NewKafkaTopology.java。

本章节只适用于MRS产品Storm与Kafka组件间的访问。本章中描述的jar包的具体版本信息请以实际情况为准。

应用开发操作步骤

步骤1 确认华为MRS产品Storm和Kafka组件已经安装，并正常运行。

步骤2 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\storm-examples”目录下的样例工程文件夹storm-examples并将storm-examples导入到IntelliJ IDEA开发环境，参见**准备Storm应用开发环境**。

步骤3 在Linux环境下安装Storm客户端。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

步骤4 下载并安装Kafka客户端程序。

----结束

代码样例

1. 创建拓扑

```
public static void main(String[] args) throws Exception {

// 设置拓扑配置
Config conf = new Config();

// 配置安全插件
//setSecurityPlugin(conf);

if (args.length >= 2) {
// 用户更改了默认的keytab文件名，这里需要将新的keytab文件名通过参数传入
conf.put(Config.TOPOLOGY_KEYTAB_FILE, args[1]);
}

// 定义KafkaSpout
KafkaSpout kafkaSpout = new KafkaSpout<String, String>(
getKafkaSpoutConfig(getKafkaSpoutStreams()));

// CountBolt
CountBolt countBolt = new CountBolt();
//SplitBolt
SplitSentenceBolt splitBolt = new SplitSentenceBolt();
```

```
// KafkaBolt配置信息
KafkaBolt<String, String> kafkaBolt = new KafkaBolt<String, String>();
kafkaBolt.withTopicSelector(new DefaultTopicSelector(OUTPUT_TOPIC))
.withTupleToKafkaMapper(
new FieldNameBasedTupleToKafkaMapper("word", "count"));
kafkaBolt.withProducerProperties(getKafkaProducerProps());

// 定义拓扑
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("kafka-spout", kafkaSpout, 10);
builder.setBolt("split-bolt", splitBolt,10).shuffleGrouping("kafka-spout", STREAMS[0]);
builder.setBolt("count-bolt", countBolt, 10).fieldsGrouping(
"split-bolt", new Fields("word"));
builder.setBolt("kafka-bolt", kafkaBolt, 10).shuffleGrouping("count-bolt");

// 命令行提交拓扑
StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
}
```

2. NewKafkaTopology类里的getKafkaConsumerProps()

```
private static Map<String, Object> getKafkaConsumerProps() throws Exception {
    Map<String, Object> props = new HashMap<String, Object>();
    props.put(GROUP_ID, DEFAULT_GROUP_ID);
    props.put(SASL_KERBEROS_SERVICE_NAME, DEFAULT_SERVICE_NAME);
    props.put(SECURITY_PROTOCOL, DEFAULT_SECURITY_PROTOCOL);
    props.put(KEY_DESERIALIZER, DEFAULT_DESERIALIZER);
    props.put(VALUE_DESERIALIZER, DEFAULT_DESERIALIZER);
    //props.put(KERBEROS_DOMAIN_NAME, "hadoop." +
KerberosUtil.getDefaultRealm().toLowerCase());
    return props;
}
```

3. NewKafkaTopology类里的getKafkaProducerProps()

```
private static Properties getKafkaProducerProps() throws Exception {
    Properties props = new Properties();
    props.put(BOOTSTRAP_SERVERS, KAFKA_BROKER_LIST);
    props.put(SECURITY_PROTOCOL, DEFAULT_SECURITY_PROTOCOL);
    props.put(KEY_SERIALIZER, DEFAULT_SERIALIZER);
    props.put(VALUE_SERIALIZER, DEFAULT_SERIALIZER);
    props.put(SASL_KERBEROS_SERVICE_NAME, DEFAULT_SERVICE_NAME);
    //props.put(KERBEROS_DOMAIN_NAME, "hadoop." +
KerberosUtil.getDefaultRealm().toLowerCase());
    return props;
}
```

说明

如果修改了集群域名，在设置Kafka消费者/生产者属性中kerberos域名时，需要将其设置为集群实际域名，例如props.put(KERBEROS_DOMAIN_NAME , "*hadoop.hadoop1.com*")。

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Strom样例工程应用**。

步骤2 获取下列jar包：

在安装好的Kafka客户端目录中进入Kafka/kafka/libs目录，获取如下jar包：

- kafka_<version>.jar
- scala-library-<version>.jar
- scala-logging_2.11-3.7.2.jar
- metrics-core-<version>.jar
- kafka-clients-<version>.jar

- zookeeper-<version>.jar

在Storm客户端的“streaming-cql-<*HD-Version*>/lib”目录中获取如下jar包：

- storm-kafka-client-<version>.jar
- storm-kafka-<version>.jar
- slf4j-api-<version>.jar
- guava-<version>.jar
- json-simple-<version>.jar
- curator-client-<version>.jar
- curator-framework-<version>.jar
- curator-recipes-<version>.jar

步骤3 将**步骤1**和**步骤2**中获取的jar包和配置文件合并统一打出完整的业务jar包，请参见**打包Strom应用业务**。

步骤4 进入Linux系统中进入Kafka客户端所在目录，在Kafka/kafka/bin目录下使用Kafka客户端创建拓扑中所用到的Topic，执行命令：

./kafka-topics.sh --create --topic input --partitions 2 --replication-factor 2 --zookeeper {ip:port}/kafka

./kafka-topics.sh --create --topic output --partitions 2 --replication-factor 2 --zookeeper {ip:port}/kafka

说明

“--zookeeper”后面填写的是ZooKeeper地址，需要改为安装集群时配置的ZooKeeper地址。

步骤5 在Linux系统中完成拓扑的提交。提交命令示例（拓扑名为kafka-test）：

storm jar /opt/jartarget/source.jar com.huawei.storm.example.kafka.NewKafkaTopology kafka-test

步骤6 拓扑提交成功后，可以向Kafka中发送数据，观察是否有相关信息生成。

在Linux系统中进入Kafka客户端所在目录，在Kafka/kafka/bin目录下启动consumer观察数据是否生成。执行命令：

./kafka-console-consumer.sh --bootstrap-server {ip:port} --topic output --consumer.config ../config/consumer.properties

同时在Linux系统中进入Kafka客户端所在目录，在Kafka/kafka/bin目录下启动producer，向Kafka中写入数据。执行命令：

./kafka-console-producer.sh --broker-list {ip:port} --topic input --producer.config ../config/producer.properties

向input中写入测试数据，可以观察到output中有对应的数据产生，则说明Storm-Kafka拓扑运行成功。

----结束

30.5.2 Storm-JDBC 开发指引

操作场景

本文档主要说明如何使用开源Storm-JDBC工具包，完成Storm和JDBC之间的交互。Storm-JDBC中包含两类Bolt：JdbcInsertBolt和JdbcLookupBolt。其中，JdbcLookupBolt主要负责从数据库中查数据，JdbcInsertBolt主要向数据库中存数据。当然，JdbcLookupBolt和JdbcInsertBolt中也可以增加处理逻辑对数据进行处理。

本章节只适用Storm与JDBC组件间的访问。本章中描述的jar包的具体版本信息请以实际情况为准。

应用开发操作步骤

步骤1 确认产品Storm组件已经安装，且正常运行。

步骤2 参考**获取MRS应用开发样例工程**，获取样例代码解压目录中“src\storm-examples”目录下的样例工程文件夹storm-examples并将storm-examples导入到IntelliJ IDEA开发环境，参见**准备Storm应用开发环境**。

步骤3 工程导入后，修改样例工程的“resources/flux-examples”目录下的“jdbc.properties”文件，根据实际环境信息修改相关参数。

```
#配置JDBC服务端IP地址
JDBC_SERVER_NAME=
#配置JDBC服务端端口
JDBC_PORT_NUM=
#配置JDBC登录用户名
JDBC_USER_NAME=
#配置JDBC登录用户密码
#密码明文存储存在安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全
JDBC_PASSWORD=
#配置database表名
JDBC_BASE_TBL=
```

步骤4 在Linux环境下安装Storm客户端。

集群的Master节点或者Core节点使用客户端可参考**集群内节点使用MRS客户端**，MRS集群外客户端的安装操作可参考**集群外节点使用MRS客户端**。

----结束

数据库配置—Derby 数据库配置过程

步骤1 首先应下载一个数据库，可根据具体场景选择最适合的数据库。

该任务以Derby数据库为例。Derby是一个小型的，java编写的，易于使用却适合大多数应用程序的开放源码数据库。

步骤2 Derby数据库的获取。在官网下载最新版的Derby数据库，将下载下来的数据库将传入Linux客户端(如"/opt")，并解压。

步骤3 在Derby的安装目录下，进入bin目录，输入如下命令：

export DERBY_INSTALL=/opt/db-derby-10.12.1.1-bin

export CLASSPATH=$DERBY_INSTALL/lib/derbytools.jar:$DERBY_INSTALL\lib\derbynet.jar:.

export DERBY_HOME=/opt/db-derby-10.12.1.1-bin

. setNetworkServerCP

./startNetworkServer -h *主机名*

步骤4 执行**./ij**命令，输入**connect 'jdbc:derby://***主机名***:1527/example;create=true';**，建立连接。

步骤5 数据库建立好后，可以执行sql语句进行操作，需要建立两张表ORIGINAL和GOAL，并向ORIGINAL中插入一组数据，命令如下：（表名仅供参考，可自行设定）

CREATE TABLE GOAL(WORD VARCHAR(12),COUNT INT);

CREATE TABLE ORIGINAL(WORD VARCHAR(12),COUNT INT);

INSERT INTO ORIGINAL VALUES('orange',1),('pineapple',1),('banana',1),('watermelon',1);

----结束

代码样例

SimpleJDBCTopology代码样例（代码中涉及到的IP端口请修改为实际的IP及端口）：

```
public class SimpleJDBCTopology {

    private static final Logger logger = Logger.getLogger(SimpleJDBCTopology.class);
    private static final String WORD_SPOUT = "WORD_SPOUT";

    private static final String JDBC_INSERT_BOLT = "JDBC_INSERT_BOLT";

    private static final String JDBC_LOOKUP_BOLT = "JDBC_LOOKUP_BOLT";

    // 用户创建的源表表名，可自行修改
    private static final String JDBC_ORIGIN_TBL = "ORIGINAL";

    // 用户创建的目标表表名，可自行修改
    private static final String JDBC_INSERT_TBL = "GOAL";

    @SuppressWarnings("rawtypes")
    public static void main(String[] args) throws Exception {
        // 获取配置文件
        Properties properties = new Properties();
        String proPath =
                System.getProperty("user.dir")
                        + File.separator
                        + "src"
                        + File.separator
                        + "main"
                        + File.separator
                        + "resources"
                        + File.separator
                        + "flux-examples"
                        + File.separator
                        + "jdbc.properties";
        try {
            properties.load(new FileInputStream(proPath));
        } catch (IOException e) {
            logger.error("Failed to load properties file.");
            throw e;
        }
        String serverName = properties.getProperty("JDBC_SERVER_NAME");
        String portNum = properties.getProperty("JDBC_PORT_NUM");
        String userName = properties.getProperty("JDBC_USER_NAME");
        String password = properties.getProperty("JDBC_PASSWORD");
        String baseTbl = properties.getProperty("JDBC_BASE_TBL");

        // connectionProvider配置
```

```
        Map<String, Object> hikariConfigMap = Maps.newHashMap();
        hikariConfigMap.put("dataSourceClassName", "org.apache.derby.jdbc.ClientDataSource");
        hikariConfigMap.put("dataSource.serverName", serverName);
        hikariConfigMap.put("dataSource.portNumber", portNum);
        hikariConfigMap.put("dataSource.databaseName", baseTbl);
        hikariConfigMap.put("dataSource.user", userName);
        hikariConfigMap.put("dataSource.password", password);
        hikariConfigMap.put("connectionTestQuery", "select COUNT from " + JDBC_INSERT_TBL);

        Config conf = new Config();

        ConnectionProvider connectionProvider = new HikariCPConnectionProvider(hikariConfigMap);

        // JdbcLookupBolt实例化
        Fields outputFields = new Fields("WORD", "COUNT");
        List<Column> queryParamColumns = Lists.newArrayList(new Column("WORD", Types.VARCHAR));
        SimpleJdbcLookupMapper jdbcLookupMapper = new SimpleJdbcLookupMapper(outputFields,
queryParamColumns);
        String selectSql = "select COUNT from " + JDBC_ORIGIN_TBL + " where WORD = ?";
        JdbcLookupBolt wordLookupBolt = new JdbcLookupBolt(connectionProvider, selectSql,
jdbcLookupMapper);

        // JdbcInsertBolt实例化
        String tableName = JDBC_INSERT_TBL;
        JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper(tableName, connectionProvider);
        JdbcInsertBolt wordInsertBolt =
                new JdbcInsertBolt(connectionProvider, simpleJdbcMapper)
                        .withTableName(JDBC_INSERT_TBL)
                        .withQueryTimeoutSecs(30);

        JDBCSpout wordSpout = new JDBCSpout();

        // 构造拓扑，wordSpout==>wordLookupBolt==>wordInsertBolt
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout(WORD_SPOUT, wordSpout);
        builder.setBolt(JDBC_LOOKUP_BOLT, wordLookupBolt, 1).fieldsGrouping(WORD_SPOUT, new
Fields("WORD"));
        builder.setBolt(JDBC_INSERT_BOLT, wordInsertBolt, 1).fieldsGrouping(JDBC_LOOKUP_BOLT, new
Fields("WORD"));

        StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
    }
}
```

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Strom样例工程应用**。

步骤2 获取下列jar包：

- 在安装好的db数据库目录下进入lib目录，获取如下jar包：
  - derbyclient.jar
  - derby.jar
- 在Storm客户端的“streaming-cql-<*HD-Version*>/lib”目录中获取如下jar包：
  - storm-jdbc-<version>.jar
  - guava-<version>.jar
  - commons-lang3-<version>.jar
  - commons-lang-<version>.jar
  - HikariCP-<version>.jar

步骤3 将上述两步中获取的jar包合并统一打出完整的业务jar包，请参见**打包Strom应用业务**。

步骤4 执行命令提交拓扑。提交命令示例（拓扑名为jdbc-test）：

storm jar /opt/jartarget/source.jar com.huawei.storm.example.jdbc.SimpleJDBCTopology jdbc-test

----结束

结果查看

当拓扑提交完成后，用户可以去数据库中查看对应表中是否有数据插入，具体过程如下：

执行SQL语句**select * from goal;** 查询“goal”表中的数据，如果goal表中有数据添加，则表明整个拓扑运行成功。

30.5.3 Storm-HDFS 开发指引

操作场景

本章节只适用于Storm和HDFS交互的场景。本章中描述的jar包的具体版本信息请以实际情况为准。

应用开发操作步骤

步骤1 确认Storm和HDFS组件已经安装，并正常运行。

步骤2 将storm-examples导入到IntelliJ IDEA开发环境，请参见**准备Storm应用开发环境**。

步骤3 下载并安装HDFS客户端。

步骤4 获取相关配置文件。获取方法如下：

在安装好的HDFS客户端目录下找到目录“/opt/clientHDFS/HDFS/hadoop/etc/hadoop”，在该目录下获取到配置文件“core-site.xml”和“hdfs-site.xml”。

步骤5 获取相关jar包。获取方法如下：

- 在安装好的HDFS客户端目录下找到目录“HDFS/hadoop/share/hadoop/common/lib”，获取如下jar包：
  - commons-cli-<version>.jar
  - commons-io-<version>.jar
  - commons-lang-<version>.jar
  - commons-lang3-<version>.jar
  - commons-collections-<version>.jar
  - commons-configuration2-<version>.jar
  - commons-logging-<version>.jar
  - guava-<version>.jar
  - hadoop-*.jar
  - protobuf-java-<version>.jar
  - jackson-databind-<version>.jar
  - jackson-core-<version>.jar
  - jackson-annotations-<version>.jar

  - re2j-<version>.jar
  - jaeger-core-<version>.jar
  - opentracing-api-<version>.jar
  - opentracing-noop-<version>.jar
  - opentracing-tracerresolver-<version>.jar
  - opentracing-util-<version>.jar
- 在安装好的HDFS客户端目录下找到目录“HDFS/hadoop/share/hadoop/common”，获取该目录下的hadoop-*.jar。
- 在安装好的HDFS客户端目录下找到目录“HDFS/hadoop/share/hadoop/client”，获取该目录下的hadoop-*.jar。
- 在安装好的HDFS客户端目录下找到目录“HDFS/hadoop/share/hadoop/hdfs”，获取该目录下的hadoop-hdfs-*.jar。
- 在样例工程“/src/storm-examples/storm-examples/lib”中获取如下jar包：
  - storm-hdfs-<version>.jar
  - storm-autocreds-<version>.jar

----结束

IntelliJ IDEA 代码样例

创建Topology。

```
public static void main(String[] args) throws Exception
  {
    TopologyBuilder builder = new TopologyBuilder();

    // 分隔符格式，当前采用“|”代替默认的“，”对tuple中的field进行分隔
    // HdfsBolt必选参数
    RecordFormat format = new DelimitedRecordFormat()
            .withFieldDelimiter("|");

    // 同步策略，每1000个tuple对文件系统进行一次同步
    // HdfsBolt必选参数
    SyncPolicy syncPolicy = new CountSyncPolicy(1000);

    // 文件大小循环策略，当文件大小到达5M时，从头开始写
    // HdfsBolt必选参数
    FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.MB);

    // 写入HDFS的目的文件
    // HdfsBolt必选参数
    FileNameFormat fileNameFormat = new DefaultFileNameFormat()
            .withPath("/user/foo/");


    //创建HdfsBolt
    HdfsBolt bolt = new HdfsBolt()
            .withFsUrl(DEFAULT_FS_URL)
            .withFileNameFormat(fileNameFormat)
            .withRecordFormat(format)
            .withRotationPolicy(rotationPolicy)
            .withSyncPolicy(syncPolicy);

    //Spout生成随机语句
    builder.setSpout("spout", new RandomSentenceSpout(), 1);
    builder.setBolt("split", new SplitSentence(), 1).shuffleGrouping("spout");
    builder.setBolt("count", bolt, 1).fieldsGrouping("split", new Fields("word"));

    //增加Kerberos认证所需的plugin到列表中，安全模式必选
```

```
    setSecurityConf(conf,AuthenticationType.KEYTAB);

    Config conf = new Config();
    //将客户端配置的plugin列表写入config指定项中，安全模式必配
    conf.put(Config.TOPOLOGY_AUTO_CREDENTIALS, auto_tgts);

    if(args.length >= 2)
    {
        //用户更改了默认的keytab文件名，这里需要将新的keytab文件名通过参数传入
        conf.put(Config.STORM_CLIENT_KEYTAB_FILE, args[1]);
    }

    //命令行提交拓扑
    StormSubmitter.submitTopology(args[0], conf, builder.createTopology());

}
```

说明

Storm不支持将HDFS的目的文件路径设置为HDFS的SM4加密分区。

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Strom样例工程应用**。

步骤2 将**步骤1**导出的本地Jar包，**步骤4**中获取的配置文件和**步骤5**中获取的jar包合并统一打出完整的业务jar包，请参见**打包Strom应用业务**。

步骤3 执行命令提交拓扑。

storm jar /opt/jartarget/source.jar com.huawei.storm.example.hdfs.SimpleHDFSTopology hdfs-test

步骤4 拓扑提交成功后请登录HDFS集群查看。

----结束

30.5.4 Storm-HBase 开发指引

操作场景

本章节只适用于Storm和HBase交互的场景。本章中描述的jar包的具体版本信息请以实际情况为准。

应用开发操作步骤

步骤1 确认Storm和HBase组件已经安装，并正常运行。

步骤2 将storm-examples导入到IntelliJ IDEA开发环境，请参见**准备Storm应用开发环境**。

步骤3 下载并安装HBase客户端。

步骤4 获取相关配置文件。获取方法如下：

在安装好的HBase客户端目录下找到目录“/opt/clientHbase/HBase/hbase/conf”，在该目录下获取到core-site.xml、hdfs-site.xml、hbase-site.xml配置文件。

步骤5 获取相关jar包。获取方式如下：

- 在安装好的HBase客户端目录下找到目录HBase/hbase/lib，获取如下jar包：

  - hbase-*.jar
  - hadoop-*.jar
  - jackson-core-asl-<version>.jar
  - jackson-mapper-asl-<version>.jar
  - commons-cli-<version>.jar
  - commons-io-<version>.jar
  - commons-lang-<version>.jar
  - commons-lang3-<version>.jar
  - commons-collections-<version>.jar
  - commons-configuration2-<version>.jar
  - guava-<version>.jar
  - protobuf-java-<version>.jar
  - netty-all-<version>.jar
  - zookeeper-<version>.jar
  - zookeeper-<version>.jar
  - zookeeper-jute-<version>.jar
  - metrics-core-<version>.jar
  - commons-validator-<version>.jar
- 在HBase客户端安装目录下找到目录HBase/hbase/lib/client-facing-thirdparty，获取commons-logging-<version>.jar。
- 在HBase客户端安装目录下找到目录HBase/hbase/lib/jdbc，获取htrace-core-<version>-incubating.jar和htrace-core4-<version>-incubating.jar。
- 在样例工程“/src/storm-examples/storm-examples/lib”中获取如下jar包：
  - storm-hbase-<version>.jar
  - storm-autocreds-<version>.jar

----结束

IntelliJ IDEA 代码样例

创建Topology。

```
public static void main(String[] args) throws Exception
    {
        Config conf = new Config();

        //增加kerberos认证所需的plugin到列表中，安全模式必选
        setSecurityConf(conf,AuthenticationType.KEYTAB);

        if(args.length >= 2)
        {
        //用户更改了默认的keytab文件名，这里需要将新的keytab文件名通过参数传入
        conf.put(Config.STORM_CLIENT_KEYTAB_FILE, args[1]);
        }
        //hbase的客户端配置，这里只提供了“hbase.rootdir”配置项，参数可选
        Map<String, Object> hbConf = new HashMap<String, Object>();
        if(args.length >= 3)
        {
            hbConf.put("hbase.rootdir", args[2]);
        }
        //必配参数，若用户不输入，则该项为空
```



```
        conf.put("hbase.conf", hbConf);

        //spout为随机单词spout
        WordSpout spout = new WordSpout();
        WordCounter bolt = new WordCounter();

        //HbaseMapper，用于解析tuple内容
        SimpleHBaseMapper mapper = new SimpleHBaseMapper()
                .withRowKeyField("word")
                .withColumnFields(new Fields("word"))
                .withCounterFields(new Fields("count"))
                .withColumnFamily("cf");

        //HBaseBolt，第一个参数为表名
        //withConfigKey("hbase.conf")将hbase的客户端配置传入HBaseBolt
        HBaseBolt hbase = new HBaseBolt("WordCount", mapper).withConfigKey("hbase.conf");


        // wordSpout ==> countBolt ==> HBaseBolt
        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout(WORD_SPOUT, spout, 1);
        builder.setBolt(COUNT_BOLT, bolt, 1).shuffleGrouping(WORD_SPOUT);
        builder.setBolt(HBASE_BOLT, hbase, 1).fieldsGrouping(COUNT_BOLT, new Fields("word"));
        //命令行提交拓扑
        StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
}
```

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Strom样例工程应用**。

步骤2 将**步骤1**中导出的本地Jar包，**步骤4**中获取的配置文件和**步骤5**中获取的jar包合并统一打出完整的业务jar包，请参见**打包Strom应用业务**。

步骤3 执行命令提交拓扑。

storm jar /opt/jartarget/source.jar com.huawei.storm.example.hbase.SimpleHBaseTopology hbase-test

说明

因为示例中的HBaseBolt并没有建表功能，在提交之前确保HBase中存在相应的表，若不存在需要手动建表，HBase shell建表语句如下**create 'WordCount', 'cf'**。

步骤4 拓扑提交成功后请自行登录HBase集群查看。

----结束

30.5.5 Storm Flux 开发指引

操作场景

本章节只适用于Storm组件使用Flux框架提交和部署拓扑的场景。本章中描述的jar包的具体版本信息请以实际情况为准。

Flux框架是Storm提供的提高拓扑部署易用性的框架。通过Flux框架，用户可以使用yaml文件来定义和部署拓扑，并且最终通过storm jar命令来提交拓扑的一种方式，极大地方便了拓扑的部署和提交，缩短了业务开发周期。

基本语法说明

使用Flux定义拓扑分为两种场景，定义新拓扑和定义已有拓扑。

1. **使用Flux定义新拓扑**

   使用Flux定义拓扑，即使用yaml文件来描述拓扑，一个完整的拓扑定义需要包含以下几个部分：

   - 拓扑名称
   - 定义拓扑时需要的组件列表
   - 拓扑的配置
   - 拓扑的定义，包含spout列表、bolt列表和stream列表

   定义拓扑名称：

   ```
   name: "yaml-topology"
   ```

   定义组件列表示例:

   ```
   #简单的component定义
    components:
    - id: "stringScheme"
    className: "org.apache.storm.kafka.StringScheme"

    #使用构造函数定义component
    - id: "defaultTopicSelector"
    className: "org.apache.storm.kafka.bolt.selector.DefaultTopicSelector"
    constructorArgs:
    - "output"

    #构造函数入参使用引用，使用`ref`标志来说明引用
    #在使用引用时请确保被引用对象在前面定义
    - id: "stringMultiScheme"
    className: "org.apache.storm.spout.SchemeAsMultiScheme"
    constructorArgs:
    - ref: "stringScheme"

    #构造函数入参引用指定的properties文件中的配置项，使用`${}`标志来表示
    #引用properties文件时，请在使用storm jar命令提交拓扑时使用--filter my-prop.properties的方式指明
   properties文件路径
    - id: "zkHosts"
    className: "org.apache.storm.kafka.ZkHosts"
    constructorArgs:
    - "${kafka.zookeeper.root.list}"

    #构造函数入参引用环境变量，使用`${ENV-[NAME]}`方式来引用
    #NAME必须是一个已经定义的环境变量
    - id: "zkHosts"
    className: "org.apache.storm.kafka.ZkHosts"
    constructorArgs:
    - "${ENV-ZK_HOSTS}"

    #使用`properties`关键字初始化内部私有变量
    - id: spoutConfig
    className: "org.apache.storm.kafka.SpoutConfig"
    constructorArgs:
    - ref: "zkHosts"
    - "input"
    - "/kafka/input"
    - "myId"
    properties:
    - name: "scheme"
    ref: "stringMultiScheme"
   ```

   定义拓扑的配置示例：

   ```
   config:
    #简单配置项
    topology.workers: 1
   ```

```
#配置项值为列表，使用`[]`表示
topology.auto-credentials: ["class1","class2"]

#配置项值为map结构
kafka.broker.properties:
metadata.broker.list: "${metadata.broker.list}"
producer.type: "async"
request.required.acks: "0"
serializer.class: "kafka.serializer.StringEncoder"
```

定义spout/bolt列表示例：

```
#定义spout列表
spouts:
- id: "spout1"
className: "org.apache.storm.kafka.KafkaSpout"
constructorArgs:
- ref: "spoutConfig"
parallelism: 1

#定义bolt列表
bolts:
- id: "bolt1"
className: "com.huawei.storm.example.hbase.WordCounter"
parallelism: 1

#使用方法来初始化对象，关键字为`configMethods`
- id: "bolt2"
className: "org.apache.storm.hbase.bolt.HBaseBolt"
constructorArgs:
- "WordCount"
- ref: "mapper"
configMethods:
- name: "withConfigKey"
args: ["hbase.conf"]
parallelism: 1
```

定义stream列表示例：

```
#定义流式需要制定分组方式，关键字为`grouping`，当前提供的分组方式关键字有：
#`ALL`,`CUSTOM`,`DIRECT`,`SHUFFLE`,`LOCAL_OR_SHUFFLE`,`FIELDS`,`GLOBAL`, 和 `NONE`.
#其中`CUSTOM`为用户自定义分组

#简单流定义，分组方式为SHUFFLE
streams:
- name: "spout1 --> bolt1"
from: "spout1"
to: "bolt1"
grouping:
type: SHUFFLE

#分组方式为FIELDS，需要传入参数
- name: "bolt1 --> bolt2"
from: "bolt1"
to: "bolt2"
grouping:
type: FIELDS
args: ["word"]

#分组方式为CUSTOM，需要指定用户自定义分组类
- name: "bolt-1 --> bolt2"
from: "bolt-1"
to: "bolt-2"
grouping:
type: CUSTOM
customClass:
className: "org.apache.storm.testing.NGrouping"
constructorArgs:
- 1
```

2. **使用Flux定义已有拓扑**

   如果已经拥有拓扑（例如已经使用java代码定义了拓扑），仍然可以使用Flux框架来提交和部署，这时需要在现有的拓扑定义（如MyTopology.java）中实现getTopology()方法，在java中定义如下：

   ```
   public StormTopology getTopology(Config config)
   或者
   public StormTopology getTopology(Map<String, Object> config)
   ```

   这时可以使用如下yaml文件来定义拓扑：

   ```
   name: "existing-topology"  #拓扑名可随意指定
   topologySource:
   className: "custom-class"  #请指定客户端类
   ```

   当然，仍然可以指定其他方法名来获得StormTopology（非getTopology()方法），yaml文件示例如下：

   ```
   name: "existing-topology"
   topologySource:
   className: "custom-class "
   methodName: "getTopologyWithDifferentMethodName"
   ```

   说明

   指定的方法必须接受一个Map<String, Object>类型或者Config类型的入参，并且返回org.apache.storm.generated.StormTopology类型的对象，和getTopology()方法相同。

应用开发操作步骤

步骤1 确认Storm组件已经安装，并正常运行。如果业务需要连接其他组件，请同时安装该组件并运行。

步骤2 将storm-examples导入到IntelliJ IDEA开发环境，请参见**导入并配置Storm样例工程**。

步骤3 参考storm-examples工程src/main/resources/flux-examples目录下的相关yaml应用示例，开发客户端业务。

步骤4 获取相关配置文件。

说明

本步骤只适用于业务中有访问外部组件需求的场景，如HDFS、HBase等，获取方式请参见Storm-HDFS开发指引或者Storm-HBase开发指引。若业务无需获取相关配置文件，请忽略本步骤。

步骤5 获取相关jar包，获取方法如下：

- 在Storm客户端的“streaming-cql-<*HD-Version*>/lib”目录中获取如下jar包：
  - flux-core-<version>.jar
  - flux-wrappers-<version>.jar
- 获取业务相关其他jar包，如访问HDFS时需要获取的jar包请参见**步骤5**，其他场景类似。

----结束

Flux 配置文件样例

下面是一个完整的访问Kafka业务的yaml文件样例：

```
name: "simple_kafka"

components:
```



```
- id: "zkHosts"  #对象名称
className: "org.apache.storm.kafka.ZkHosts"  #完整的类名
constructorArgs:  #构造函数
- "${kafka.zookeeper.root.list}"  #构造函数的参数

- id: "stringScheme"
className: "org.apache.storm.kafka.StringScheme"

- id: "stringMultiScheme"
className: "org.apache.storm.spout.SchemeAsMultiScheme"
constructorArgs:
- ref: "stringScheme"  #使用了引用，值为前面定义的stringScheme

- id: spoutConfig
className: "org.apache.storm.kafka.SpoutConfig"
constructorArgs:
- ref: "zkHosts"  #使用了引用
- "input"
- "/kafka/input"
- "myId"
properties:  #使用properties来设置本对象中的名为“scheme”的私有变量
- name: "scheme"
ref: "stringMultiScheme"

- id: "defaultTopicSelector"
className: "org.apache.storm.kafka.bolt.selector.DefaultTopicSelector"
constructorArgs:
- "output"

- id: "fieldNameBasedTupleToKafkaMapper"
className: "org.apache.storm.kafka.bolt.mapper.FieldNameBasedTupleToKafkaMapper"
constructorArgs:
- "words"  #构造函数中第一个入参
- "count"  #构造函数中第二个入参

config:
topology.workers: 1  #设置拓扑的worker数量为1
kafka.broker.properties:  #设置kafka相关的配置，值为map结构
metadata.broker.list: "${metadata.broker.list}"
producer.type: "async"
request.required.acks: "0"
serializer.class: "kafka.serializer.StringEncoder"

spouts:
- id: "kafkaSpout"  #spout名称
className: "org.apache.storm.kafka.KafkaSpout"#spout的类名
constructorArgs:  #使用构造函数的方式初始化
- ref: "spoutConfig"  #构造函数的入参使用了引用
parallelism: 1  #该spout的并发设置为1

bolts:
- id: "splitBolt"
className: "com.huawei.storm.example.common.SplitSentenceBolt"
parallelism: 1

- id: "countBolt"
className: "com.huawei.storm.example.kafka.CountBolt"
parallelism: 1

- id: "kafkaBolt"
className: "org.apache.storm.kafka.bolt.KafkaBolt"
configMethods:  #使用调用对象内部方法的形式初始化对象
- name: "withTopicSelector"  #调用的内部方法名
args:  #内部方法需要的入参
- ref: "defaultTopicSelector"  #入参只有一个，使用了引用
- name: "withTupleToKafkaMapper"  #调用第二个内部方法
args:
- ref: "fieldNameBasedTupleToKafkaMapper"
```

```
#定义数据流
streams:
- name: "kafkaSpout --> splitBolt"  #第一个数据流名称，只作为展示
from: "kafkaSpout"  #数据流起点，值为spouts中定义的kafkaSpout
to: "splitBolt"  #数据流终点，值为bolts中定义的splitBolt
grouping:#定义分组方式
type: LOCAL_OR_SHUFFLE  #分组方式为local_or_shuffle

- name: "splitBolt --> countBolt"  #第二个数据流
from: "splitBolt"
to: "countBolt"
grouping:
type: FIELDS  #分组方式为fields
args: ["word"]  #fields方式需要传入参数

- name: "countBolt --> kafkaBolt"  #第三个数据流
from: "countBolt"
to: "kafkaBolt"
grouping:
type: SHUFFLE  #分组方式为shuffle，无需传入参数
```

部署运行及结果查看

步骤1 导出本地jar包，请参见**打包Strom样例工程应用**。

步骤2 将**步骤4**中获取的配置文件和**步骤5**中获取的jar包合并统一打出完整的业务jar包，请参见**打包Strom应用业务**。

步骤3 将开发好的yaml文件及相关的properties文件复制至storm客户端所在主机的任意目录下，如“/opt”。

步骤4 执行命令提交拓扑。

storm jar /opt/jartarget/source.jar org.apache.storm.flux.Flux --remote /opt/my-topology.yaml

如果设置业务以本地模式启动，则提交命令如下：

storm jar /opt/jartarget/source.jar org.apache.storm.flux.Flux --local /opt/my-topology.yaml

说明

如果业务设置为本地模式，请确保提交环境为普通模式环境，当前不支持安全环境下使用命令提交本地模式的业务。

如果使用了properties文件，则提交命令如下：

storm jar /opt/jartarget/source.jar org.apache.storm.flux.Flux --remote /opt/my-topology.yaml --filter /opt/my-prop.properties

步骤5 拓扑提交成功后请自行登录storm UI查看。

----结束

30.5.6 Storm 对外接口介绍

- Storm-HDFS采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-hdfs**。
- Storm-HBase采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-hbase**。

- Storm-Kafka采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-kafka**。
- Storm-JDBC采用的接口同开源社区版本保持一致，详情参见：**https://github.com/apache/storm/tree/v1.2.1/external/storm-jdbc**。

30.5.7 如何使用 IDEA 远程调试业务

问题

使用Storm客户端提交了业务之后，如何使用IDEA远程调试业务?

回答

以调试WordCount程序为例，演示如何进行IDEA的远程调试：

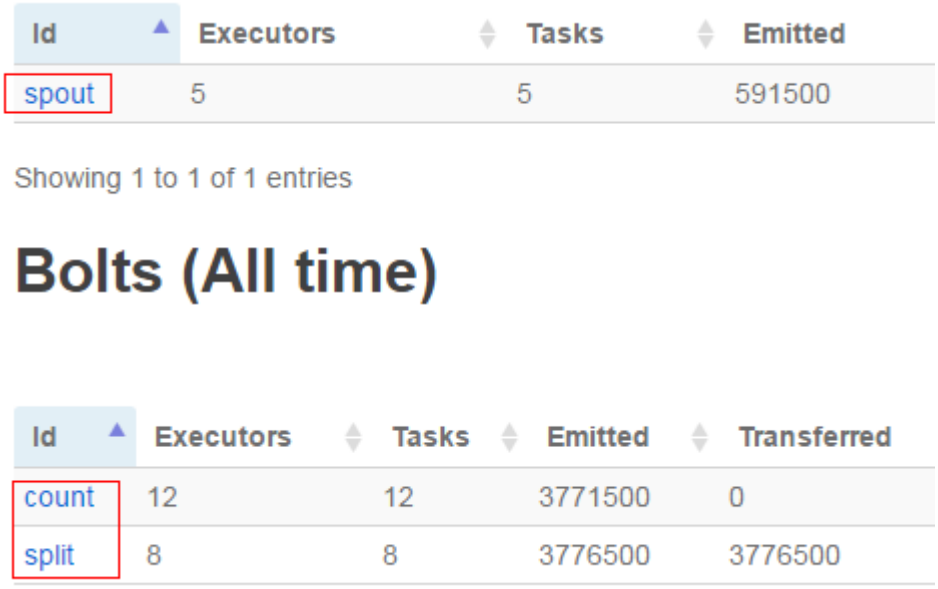
步骤1 登录FusionInsight Manager系统，选择“集群 > *待操作集群的名称* > 服务 > Storm”，选择“配置”选项卡，在搜索框中搜索并调大nimbus.task.timeout.secs和supervisor.worker.start.timeout.secs的值，建议调整为最大值。然后在WORKER_GC_OPTS的现有值后追加-Xdebug -Xrunjdwp:transport=dt_socket,address=5055,suspend=n,server=y，保存配置后重启相关实例。

说明

调试Storm程序需要先修改指定的服务端参数，并在重启服务后生效，建议在测试环境上进行调测。

步骤2 提交拓扑后，在Storm UI上进入到Topology界面，再单击进入要调试组件界面。

图 30-21 进入拓扑的 Component 界面



步骤3 在组件页面获取worker进程运行的主机ip地址，如果有多个则任选一个。

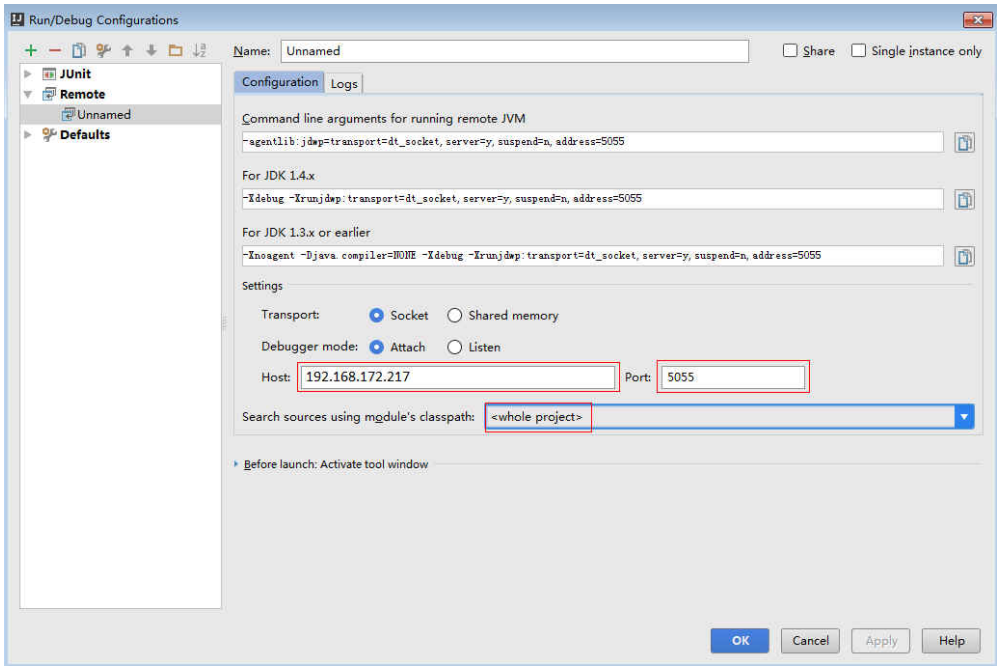
图 30-22 获取 Worker 运行的主机

步骤4 打开IDEA工程，在菜单栏中选择“Run > Edit Configurations”。

步骤5 在弹出的配置窗口中用鼠标左键单击左上角的号，在下拉菜单中选择Remote，然后选择对应要调试的源码模块路径，并配置远端调试参数Host和Port，如下图所示。

其中Host为获取的Worker运行的主机IP地址，Port为调试的端口号（确保该端口在运行机器上没被占用）。

图 30-23 配置参数



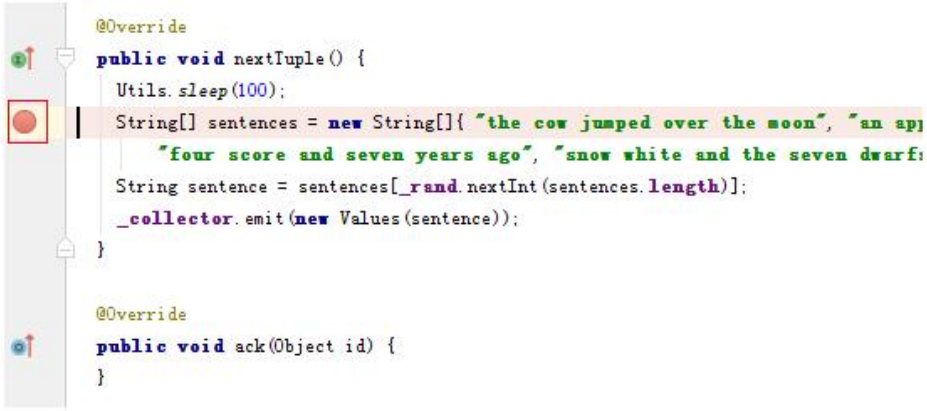
说明

当改变Port端口号时，在WORKER_GC_OPTS中追加的调试参数也要跟着改变，比如Port设置为8011，对应的调试参数则变更为-Xdebug -Xrunjdwp:transport=dt_socket,address=8011,suspend=n,server=y

步骤6 设置调试断点。

在IDEA代码编辑窗口左侧空白处单击鼠标左键设置相应代码行断点，如下图所示。

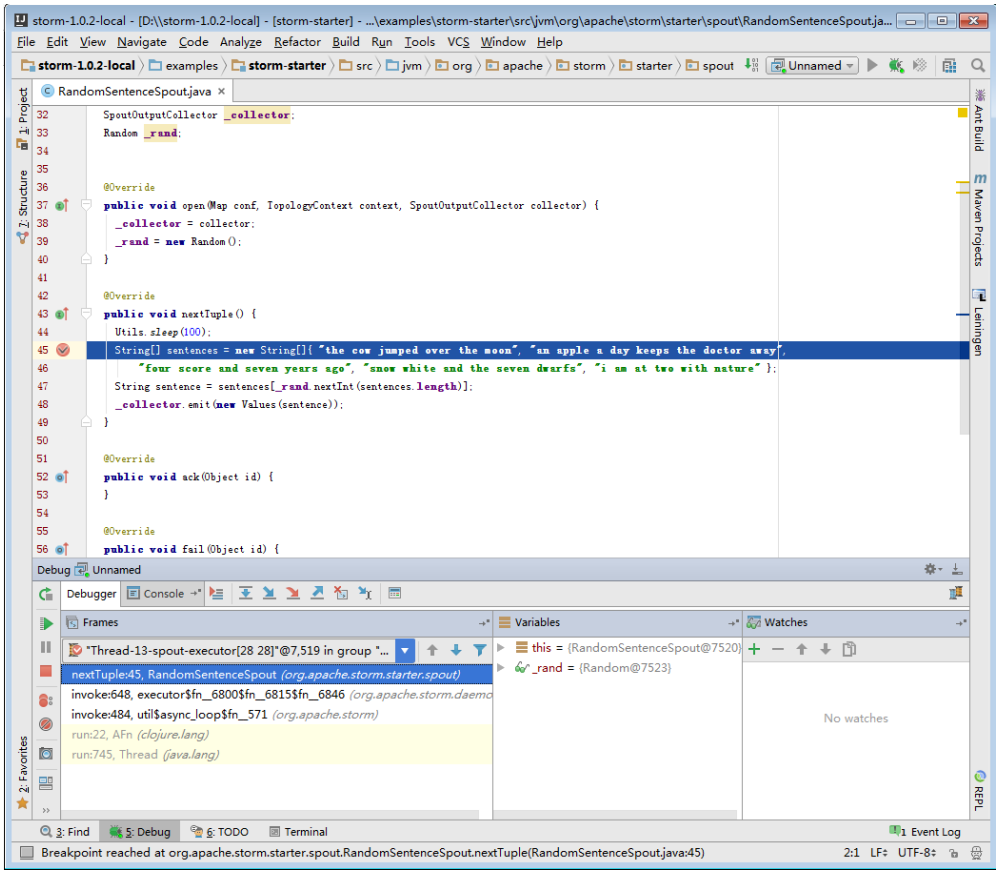
图 30-24 设置断点



步骤7 启动调试。

在IDEA菜单栏中选择“Run > Debug 'Unnamed'”开启调试窗口，接着开始调试，比如单步调试、查看调用栈、跟踪变量值等，如下图所示。

图 30-25 调试



----结束

30.5.8 使用旧插件 storm-kafka 时如何正确设置 offset

问题

当前虽然默认推荐使用storm-kafka-client插件进行安全kafka对接，但仍然存在使用旧插件storm-kafka的用户和场景，在这种场景下如何正确指定消费的offset，避免每次重启拓扑后都从头开始消费?

回答

旧插件storm-kafka中的KafkaSpout使用的是Kafka的“SimpleConsumer”接口，需要自主管理offset，KafkaSpout中根据用户定义的字段将Topic中每个Patition的offset记录在ZooKeeper中，定义如下：

```
public SpoutConfig(BrokerHosts hosts, String topic, String zkRoot, String id) {
    super(hosts, topic);
    this.zkRoot = zkRoot;
    this.id = id;
}
```

其中“hosts”是ZooKeeper的连接串，如：192.168.0.1:2181/kafka，“topic”是待消费的Topic名，“zkRoot”表示在ZooKeeper中的存放数据的根路径，一般为：“/kafka/{topic}”，“id”表示应用的标示，如：app1。读取offset会有以下两种场景：

- 场景1

  当拓扑运行后，KafkaSpout会将offset存放在ZooKeeper路径：“/{zkRoot}/{id}/{partitionId}”下，其中“zkRoot”和“id”是用户指定的，“partitionId”是自动获取的。默认情况下，拓扑在启动后会先从ZooKeeper上的offset存放路径读取历史的offset，用作本次的消费起点，因此只需要正确的指定“zkRoot”和“id”，就可以继承历史记录的offset，不用从头开始消费。

- 场景2

  没有像场景1中那样设置固定的“zkRoot”或者“id”，导致无法读取历史的offset，如此一来每次提交拓扑都会把历史已经消费过的数据再消费一遍，这时需要通过如下方式手动指定：

  ```
  SpoutConfig spoutConfig = new SpoutConfig(hosts, inputTopicName, zkRoot, appId);
  spoutConfig.ignoreZkOffsets = true;
  spoutConfig.startOffsetTime = kafka.api.OffsetRequest.LatestTime();
  ```

  通过指定SpoutConfig中的“ignoreZkOffsets”和“startOffsetTime”来强制消费最新的数据。

说明

在实际使用中推荐使用场景1中的方式，因为场景2中并非从上次commit成功的位置开始，因此可能会存在部分数据遗漏。

30.5.9 IntelliJ IDEA 中远程提交拓扑执行 Main 时报错：Command line is too long

问题

IntelliJ IDEA中远程提交拓扑，执行Main方法时IntelliJ IDEA报如下错 ：

Command line is too long. Shorten command line for ServiceStarter or also for Application default configuration.

回答

步骤1 打开项目中“ .idea\workspace.xml”文件。

步骤2 找到标签“<component name="PropertiesComponent"> ”，在内容中添加“ <property name="dynamic.classpath" value="true" />”，如图30-26。

图 30-26 修改“ .idea\workspace.xml”文件

----结束

31 YARN 开发指南（安全模式）

31.1 YARN 应用开发简介

简介

Yarn是一个分布式的资源管理系统，用于提高分布式的集群环境下的资源利用率，这些资源包括内存、IO、网络、磁盘等。其产生的原因是为了解决原MapReduce框架的不足。最初MapReduce的committer还可以周期性的在已有的代码上进行修改，可是随着代码的增加以及原MapReduce框架设计的不足，在原MapReduce框架上进行修改变得越来越困难，所以MapReduce的committer决定从架构上重新设计MapReduce，使下一代的MapReduce(MRv2/Yarn)框架具有更好的扩展性、可用性、可靠性、向后兼容性和更高的资源利用率，以及能支持除了MapReduce计算框架外的更多的计算框架。

基本概念

- **ResourceManager（RM）**

  RM是一个全局的资源管理器，负责整个系统的资源管理和分配。它主要由两个组件构成：调度器（Scheduler）和应用程序管理器（Applications Manager，ASM）。

- **ApplicationMaster（AM）**

  用户提交的每个应用程序均包含一个AM，主要功能包括：

  - 与RM调度器协商以获取资源（用Container表示）。
  - 将得到的资源进一步分配给内部任务。
  - 与NM通信以启动/停止任务。
  - 监控所有任务的运行状态，并在任务运行失败时重新为任务申请资源以重启任务。

- **NodeManager（NM）**

  NM是每个节点上的资源和任务管理器，一方面，它会定时地向RM汇报本节点上的资源使用情况和各个Container的运行状态；另一方面，它会接收并处理来自AM的Container启动/停止等各种请求。

- **Container**

Container是YARN中的资源抽象，它封装了某个节点上的多维度资源，如内存、CPU、磁盘、网络等，当AM向RM申请资源时，RM为AM返回的资源便是用Container表示的。

31.2 YARN 接口介绍

31.2.1 YARN Command 介绍

您可以使用YARN Commands对YARN集群进行一些操作，例如启动ResourceManager、提交应用程序、中止应用、查询节点状态、下载container日志等操作。

完整和详细的Command描述可以参考官网文档：

http://hadoop.apache.org/docs/r3.1.1/hadoop-yarn/hadoop-yarn-site/YarnCommands.html

常用 Command

YARN Commands可同时供普通用户和管理员用户使用，它包含了少量普通用户可以执行的命令，比如jar、logs。而大部分只有管理员有权限使用。

用户可以通过以下命令查看YARN用法和帮助：

yarn --help

用法：进入Yarn客户端的任意目录，执行**source**命令导入环境变量，直接运行命令即可。

格式如下所示：

yarn [--config *confdir***]** *COMMAND*

其中COMMAND内容请参考**表31-1**。

说明

其中版本号8.1.0.1为示例，具体以实际环境的版本号为准。

表 31-1 常用 Command 描述

COMMAND	描述
resourcemanager	运行一个ResourceManager。 备注：以**omm**用户执行服务端命令前需export环境变量（客户端需要有**omm**用户的执行权限），例如： • **export YARN_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_ResourceManager/etc** • **export HADOOP_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_ResourceManager/etc**

COMMAND	描述
nodemanager	运行一个NodeManager。 备注：以**omm**用户执行服务端命令前需export环境变量（客户端需要有omm用户的执行权限），例如： ● **export YARN_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_NodeManager/etc** ● **export HADOOP_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_NodeManager/etc**
rmadmin	管理员工具（动态更新信息）。
version	打印版本信息。
jar <jar>	运行jar文件。
logs	获取container日志。
classpath	打印获取Hadoop JAR包和其他库文件所需的CLASSPATH路径。
daemonlog	获取或者设置服务LOG级别。
CLASSNAME	运行一个名字为CLASSNAME的类。
top	运行集群利用率监控工具。
-Dmapreduce.job.hdfs-servers	如果对接了OBS，而服务端依然使用HDFS，那么需要显式在命令行使用该参数指定HDFS的地址。格式为hdfs://*{NAMESERVICE}*。其中{NAMESERVICE}为hdfs nameservice名称。 如果当前的HDFS具有多个nameservice，那么需要指定所有的nameservice，并以‘，’隔开。例如：hdfs://nameservice1,hdfs://nameservice2

Superior Scheduler Command

Superior Scheduler引擎提供了输出Superior Scheduler引擎具体信息的CLI。为了执行Superior命令，需要使用“<HADOOP_HOME>/bin/superior”脚本。

以下为superior命令格式：

```
<HADOOP_HOME>/bin/superior

Usage: superior [COMMAND | -help]
  Where COMMAND is one of:
  resourcepool                  prints resource pool status
  queue                         prints queue status
  application                   prints application status
  policy                        prints policy status
```

不带参数调用大多数命令时会显示帮助信息。

- Superior ***resourcepool***命令：
  该命令显示Resource Pool和相关策略的相关状态以及配置信息。

说明

Superior resourcepool命令仅用于管理员用户及拥有yarn管理权限的用户。

用法输出：

```
>superior resourcepool

Usage: resourcepool [-help]
                    [-list]
                    [-status <resourcepoolname>]
 -help                      prints resource pool usage
 -list                      prints all resource pool summary report
 -status <resourcepoolname>   prints status and configuration of specified
                            resource pool
```

- ***resourcepool -list***以表格格式中显示Resource Pool摘要。示例如下：

  ```
  > superior resourcepool -list
  NAME       NUMBER_MEMBER     TOTAL_RESOURCE           AVAILABLE_RESOURCE
  Pool1      4                 vcores 30,memory 1000    vcores 21,memory 80
  Pool2      100                vcores 100,memory 12800  vcores 30,memory 1000
  default    2                 vcores 64,memory 128     vcores 40,memory 28
  ```

- ***resourcepool -status <resourcepoolname>***以列表格式显示资源库详细信息。示例如下：

  ```
  > superior resourcepool -status default
  NAME: default
  DESCRIPTION: System generated resource pool
  TOTAL_RESOURCE: vcores 64,memory 128
  AVAILABLE_RESOURCE: vcores 40,memory 28
  NUMBER_MEMBER: 2
  MEMBERS: node1,node2
  CONFIGURATION:
  |-- RESOURCE_SELECT:
  |__RESOURCES:
  ```

- Superior ***queue***命令

  该命令输出分层队列信息。

  用法输出：

  ```
  >superior queue

  Usage: queue [-help]
            [-list] [-e] [[-name <queue_name>] [-r|-c]]
            [-status <queue_name>]
   -c                  only work with -name <queue_name> option. If this
                       option is used, command  will print information of
                       specified queue and its direct children.
   -e                  only work with -list or -list -name option. If
                       this option is used, command will print effective
                       state of specified queue and all of its
                       descendants.
   -help                 prints queue sub command usage
   -list               prints queue summary report. This option can work
                       with -name <queue_name> and -r options.
   -name <queue_name>     print specified queue, this can work with -r
                       option. By default, it will print queue's own
                       information. When -r is defined, command will
                       print all of its descendant queues. When -c is
                       defined, it will print its direct children queues.
   -r                  only work with -name <queue_name> option. If this
                       option is used, command will print information of
                       specified queue and all of its descendants.
   -status <queue_name>   prints status of specified queue
  ```

  - ***queue -list***以表格格式输出队列摘要信息。命令将基于队列分层样式输出信息。用户可通过SUBMIT ACL或ADMIN ACL的队列权限查看队列。示例如下：

    ```
    > superior queue -list
    NAME         STATE           NRUN_APP     NPEND_APP     NRUN_CONTAINER
    ```



```
NPEND_REQUEST    RES_INUSE              RES_REQUEST
root         OPEN|ACTIVE      10          20          100             200             vcores 100,memory
1000    vcores 200,memory 2000
root.Q1      OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q11  OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q12  CLOSE|INACTIVE   0           0           0               0               vcores 0,memory
0       vcores 0,memory 0
root.Q2      OPEN|INACTIVE    5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q2.Q21  OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
```

- ***queue -list -name root.Q1只***输出root.Q1。

```
> superior queue -list -name root.Q1
NAME            STATE           NRUN_APP     NPEND_APP       NRUN_CONTAINER
NPEND_REQUEST    RES_INUSE           RES_REQUEST
root.Q1         OPEN|ACTIVE     5           10          50              100             vcores 50,memory
500 vcores 100,memory 1000
```

- ***queue -list -name root.Q1 -r***将输出root.Q1及其所有的分支。

```
> superior queue -list -name root.Q1 -r
NAME         STATE           NRUN_APP     NPEND_APP  NRUN_CONTAINER
NPEND_REQUEST    RES_INUSE              RES_REQUEST
root.Q1      OPEN|ACTIVE      5           10         50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q11  OPEN|ACTIVE      5           10         50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q12  CLOSE|INACTIVE   0           0          0               0               vcores 0,memory
0       vcores 0,memory 0
```

- ***queue -list -name root -c***将会输出root及其直系子目录。

```
> superior queue -list -name root -c
NAME                STATE           NRUN_APP     NPEND_APP      NRUN_CONTAINER
NPEND_REQUEST    RES_INUSE              RES_REQUEST
root                OPEN|ACTIVE      10          20          100             200             vcores
100,memory 1000   vcores 200,memory 2000
root.Q1             OPEN|ACTIVE      5           10          50              100             vcores
50,memory 500     vcores 100,memory 1000
root.Q2             OPEN|INACTIVE    5           10          50              100             vcores
50,memory 500     vcores 100,memory 1000
```

- **queue -status <queue_name>**将会输出具体队列状态和配置。

  用户可通过SUBMIT ACL权限查看除队列ACL外的细节信息。

  用户还可通过ADMIN ACL的队列权限查看包括ACL在内的队列细节信息。

```
> superior queue -status root.Q1
NAME: root.Q1
OPEN_STATE:CLOSED
ACTIVE_STATE: INACTIVE
EOPEN_STATE: CLOSED
EACTIVE_STATE: INACTIVE
LEAF_QUEUE: Yes
NUMBER_PENDING_APPLICATION: 100
NUMBER_RUNNING_APPLICATION: 10
NUMBER_PENDING_REQUEST: 10
NUMBER_RUNNING_CONTAINER: 10
NUMBER_RESERVED_CONTAINER: 0
RESOURCE_REQUEST: vcores 3,memory 3072
RESOURCE_INUSE: vcores 2,memory 2048
RESOURCE_RESERVED: vcores 0,memory 0
CONFIGURATION:
|-- DESCRIPTION: Spark session queue
|-- MAX_PENDING_APPLICATION: 10000
|--MAX_RUNNING_APPLICATION: 1000
|--ALLOCATION_ORDER_POLICY: FIFO
|--DEFAULT_RESOURCE_SELECT: label1
|--MAX_MASTER_SHARE: 10%
|--MAX_RUNNING_APPLICATION_PER_USER : -1
```



```
|--MAX_ALLOCATION_UNIT: vcores 32,memory 12800
|--ACL_USERS: user1,user2
|--ACL_USERGROUPS: usergroup1,usergroup2
|-- ACL_ADMINS: user1
|--ACL_ADMINGROUPS: usergroup1
```

- Superior ***application***命令

  该命令输出应用相关信息。

  用法输出：

  ```
  >superior application

  Usage: application [-help]
                [-list]
                [-status <application_id>]
   -help                   prints application sub command usage
   -list                   prints all application summary report
   -status <application_id>   prints status of specified application
  ```

  用户可通过应用的浏览访问权限查看应用相关信息。

  - ***application -list***以表的形式提供所有应用的信息摘要：

    ```
    > superior application -list
    ID                                  QUEUE            USER     NRUN_CONTAINER
    NPEND_REQUEST               NRSV_CONTAINER       RES_INUSE
    RES_REQUEST                 RES_RESERVED
    application_1482743319705_0005           root.SEQ.queueB   hbase    1
    100                     0                   vcores 1,memory 1536            vcores 2000,memory
    409600       vcores 0,memory 0
    application_1482743319705_0006           root.SEQ.queueB   hbase    0
    1                       0                   vcores 0,memory 0               vcores 1,memory 1536
    vcores 0,memory 0
    ```

  - ***application -status <app_id>***命令输出指定应用的详细信息。示例如下：

    ```
    > superior application -status application_1443067302606_0609
    ID: application_1443067302606_0609
    QUEUE: root.Q1.Q11
    USER: cchen
    RESOURCE_REQUEST: vcores 3,memory 3072
    RESOURCE_INUSE: vcores 2,memory 2048
    RESOURCE_RESERVED:vcores 1, memory 1024
    NUMBER_RUNNING_CONTAINER: 2
    NUMBER_PENDING_REQUEST: 3
    NUMBER_RESERVED_CONTAINER: 1
    MASTER_CONTAINER_ID: application_1443067302606_0609_01
    MASTER_CONTAINER_RESOURCE: node1.domain.com
    BLACKLIST: node5,node8
    DEMANDS:
    |-- PRIORITY: 20
    |-- MASTER: true
    |-- CAPABILITY: vcores 2, memory 2048
    |-- COUNT: 1
    |-- RESERVED_RES : vcores 1, memory 1024
    |-- RELAXLOCALITY: true
    |-- LOCALITY: node1/1
    |-- RESOURCESELECT: label1
    |-- PENDINGREASON: “application limit reached”
    |-- ID: application_1443067302606_0609_03
    |-- RESOURCE: node1.domain.com
    |-- RESERVED_RES: vcores 1, memory 1024
    |
    |--PRIORITY: 1
    |-- MASTER: false
    |-- CAPABILITY: vcores 1,memory 1024
    |-- COUNT: 2
    |-- RESERVED_RES: vcores 0, memory 0
    |-- RELAXLOCLITY: true
    |--LOCALITY: node1/1,node2/1,rackA/2
    |-- RESOURCESELECT: label1
    |-- PENDINGREASON: “no available resource”
    ```



```
CONTAINERS:
|-- ID: application_1443067302606_0609_01
|-- RESOURCE: node1.domain.com
|-- CAPABILITY: vcores 1,memory 1024
|
|-- ID: application_1443067302606_0609_02
|-- RESOURCE: node2.domain.com
|-- CAPABILITY: vcores 1,memory 1024
```

- Superior ***policy*** 命令

  该命令输出决策相关信息。

  说明

  Superior policy命令仅限管理员用户及拥有Yarn管理权限的用户使用。

  用法输出：

```
>superior policy

Usage: policy [-help]
            [-list <resourcepoolname>] [-u] [-detail]
            [-status <resourcepoolname>]
 -detail                    only work with -list option to show a
                           summary information of resource pool
                           distribution on queues, including reserve,
                           minimum and maximum
 -help                      prints policy sub command usage
 -list <resourcepoolname>     prints a summary information of resource
                           pool distribution on queue
 -status <resourcepoolname>   prints pool distribution policy
                           configuration and status of specified
                           resource pool
 -u                         only work with -list option to show a
                           summary information of resource pool
                           distribution on queues and also user
                           accounts
```

  - ***policy -list <resourcepoolname>***输出队列分布信息摘要。示例如下：

```
>superior policy -list default
NAME: default
TOTAL_RESOURCE: vcores 16,memory 16384
AVAILABLE_RESOURCE: vcores 16,memory 16384

NAMERES_INUSERES_REQUEST
root.defaultvcores 0,memory 0vcores 0,memory 0
root.productionvcores 0,memory 0vcores 0,memory 0
root.production.BU1vcores 0,memory 0vcores 0,memory 0
root.production.BU2 vcores 0,memory 0vcores 0,memory 0
```

  - ***policy -list <resourcepoolname> -u***输出用户级信息摘要。

```
> superior policy -list default -u
NAME: default
TOTAL_RESOURCE: vcores 16,memory 16384
AVAILABLE_RESOURCE: vcores 16,memory 16384

NAMERES_INUSERES_REQUEST
root.defaultvcores 0,memory 0vcores 0,memory 0
root.default.[_others_]vcores 0,memory 0vcores 0,memory 0
root.productionvcores 0,memory 0vcores 0,memory 0
root.production.BU1vcores 0,memory 0vcores 0,memory 0
root.production.BU1.[_others_]vcores 0,memory 0vcores 0,memory 0
root.production.BU2vcores 0,memory 0vcores 0,memory 0
root.production.BU2.[_others_]vcores 0,memory 0vcores 0,memory 0
```

  - ***policy -status <resourcepoolname>*** 输出指定资源库的策略详细资料。

```
> superior policy -status pool1
NAME: pool1
TOTAL_RESOURCE: vcores 64,memory 128
AVAILABLE_RESOURCE: vcores 40,memory 28
QUEUES:
```



```
|-- NAME: root.Q1
|-- RESOURCE_USE: vcores 20, memory 1000
|-- RESOURCE_REQUEST: vcores 2,memory 100
|--RESERVE: vcores 10, memory 4096
|--MINIMUM: vcore 11, memory 4096
|--MAXIMUM: vcores 500, memory 100000
|--CONFIGURATION:
|-- SHARE: 50%
|-- RESERVE: vcores 10, memory 4096
|-- MINIMUM: vcores 11, memory 4096
|-- MAXIMUM: vcores 500, memory 100000
|-- QUEUES:
|-- NAME: root.Q1.Q11
|-- RESOURCE_USE: vcores 15, memory, 500
|-- RESOURCE_REQUEST: vcores 1, memory 50
|-- RESERVE: vcores 0, memory 0
|-- MINIMUM: vcores 0, memory 0
|-- MAXIMUM: vcores -1, memory -1
|-- USER_ACCOUNTS:
|-- NAME: user1
|-- RESOURCE_USE: vcores 1, memory 10
|-- RESOURCE_REQUEST: vcores 1, memory 50
|
|-- NAME: OTHERS
|--RESOURCE_USE: vcores 0, memory 0
|- RESOURCE_REQUEST: vcores 0, memory 0
|-- CONFIGURATION:
|-- SHARE: 100%
|-- USER_POLICY:
|-- NAME: user1
|-- WEIGHT: 10
|
|-- NAME: OTHERS
|-- WEIGHT: 1
|-- MAXIMUM: vcores 10, memory 1000
```

31.2.2 YARN Java API 接口介绍

关于YARN的详细API可以直接参考官方网站上的描述：

http://hadoop.apache.org/docs/r3.1.1/api/index.html

常用接口

YARN常用的Java类有如下几个。

- **ApplicationClientProtocol**

  用于Client与ResourceManager之间。Client通过该协议可实现将应用程序提交到ResourceManager上，查询应用程序的运行状态或者中止应用程序等功能。

  表 31-2 ApplicationClientProtocol 常用方法

方法	说明
forceKillApplication(KillApplicationRequest request)	Client通过此接口请求RM中止一个已提交的任务。
getApplicationAttemptReport(GetApplicationAttemptReportRequest request)	Client通过此接口从RM获取指定ApplicationAttempt的报告信息。

方法	说明
getApplicationAttempts(GetApplicationAttemptsRequest request)	Client通过此接口从RM获取所有ApplicationAttempt的报告信息。
getApplicationReport(GetApplicationReportRequest request)	Client通过此接口从RM获取某个应用的报告信息。
getApplications(GetApplicationsRequest request)	Client通过此接口从RM获取满足一定过滤条件的应用的报告信息。
getClusterMetrics(GetClusterMetricsRequest request)	Client通过此接口从RM获取集群的Metrics。
getClusterNodes(GetClusterNodesRequest request)	Client通过此接口从RM获取集群中的所有节点信息。
getContainerReport(GetContainerReportRequest request)	Client通过此接口从RM获取某个Container的报告信息。
getContainers(GetContainersRequest request)	Client通过此接口从RM获取某个ApplicationAttemp的所有Container的报告信息。
getDelegationToken(GetDelegationTokenRequest request)	Client通过此接口获取授权票据，用于container访问相应的service。
getNewApplication(GetNewApplicationRequest request)	Client通过此接口获取一个新的应用ID号，用于提交新的应用。
getQueueInfo(GetQueueInfoRequest request)	Client通过此接口从RM中获取队列的相关信息。
getQueueUserAcls(GetQueueUserAclsInfoRequest request)	Client通过此接口从RM中获取当前用户的队列访问权限信息。
moveApplicationAcrossQueues(MoveApplicationAcrossQueuesRequest request)	移动一个应用到新的队列。
submitApplication(SubmitApplicationRequest request)	Client通过此接口提交一个新的应用到RM。

- ApplicationMasterProtocol

  用于ApplicationMaster与ResourceManager之间。ApplicationMaster使用该协议向ResourceManager注册、申请资源、获取各个任务的运行情况等。

表 31-3 ApplicationMasterProtocol 常用方法

方法	说明
allocate(AllocateRequest request)	AM通过此接口提交资源分配申请。
finishApplicationMaster(FinishApplicationMasterRequest request)	AM通过此接口通知RM其运行成功或者失败。
registerApplicationMaster(RegisterApplicationMasterRequest request)	AM通过此接口向RM进行注册。

- ContainerManagementProtocol

  用于ApplicationMaster与NodeManager之间。ApplicationMaster使用该协议要求NodeManager启动/中止Container或者查询Container的运行状态。

表 31-4 ContainerManagementProtocol 常用方法

方法	说明
getContainerStatuses(GetContainerStatusesRequest request)	AM通过此接口向NM请求Containers的当前状态信息。
startContainers(StartContainersRequest request)	AM通过此接口向NM提供需要启动的containers列表的请求。
stopContainers(StopContainersRequest request)	AM通过此接口请求NM停止一系列已分配的Containers。

样例代码

YARN作业提交的样例代码详细可以参考MapReduce开发指南中的**MapReduce访问多组件样例代码**，实现建立一个MapReduce job，并提交MapReduce作业到Hadoop集群。

31.2.3 YARN REST API 接口介绍

功能简介

通过HTTP REST API来查看更多Yarn任务的信息。目前Yarn的REST接口只能进行一些资源或者任务的查询。完整和详细的接口请直接参考官网上的描述以了解其使用：

http://hadoop.apache.org/docs/r3.1.1/hadoop-yarn/hadoop-yarn-site/WebServicesIntro.html

准备运行环境

1. 在节点上安装客户端，例如安装到“/opt/client”目录。

2. 进入客户端安装目录“/opt/client”，执行下列命令初始化环境变量。

   source *bigdata_env*

   kinit *组件业务用户*

   说明

   kinit一次票据时效24小时。24小时后再次运行样例，需要重新kinit。

3. 与HTTP服务访问相比，以HTTPS方式访问Yarn时，由于使用了SSL安全加密，需要确保Curl命令所支持的SSL协议在集群中已添加支持。若不支持，可对应修改集群中SSL协议。例如，若Curl仅支持TLSv1协议，修改方法如下：

   登录FusionInsight Manager页面，选择“集群 > *待操作集群的名称* > 服务 > Yarn > 配置 > 全部配置”，在“搜索”框里搜索“hadoop.ssl.enabled.protocols”，查看参数值是否包含“TLSv1”，若不包含，则在配置项“hadoop.ssl.enabled.protocols”中追加“,TLSv1”。清空“ssl.server.exclude.cipher.list”配置项的值，否则以HTTPS访问不了Yarn。单击“保存”，单击“确定”，保存完成后重启该服务。

   说明

   TLSv1协议存在安全漏洞，请谨慎使用。

操作步骤

1. 获取运行在Yarn上的任务的具体信息。
   - 命令：

     ```
     curl -k -i --negotiate -u : "https://10-120-85-2:8090/ws/v1/cluster/apps/"
     ```

     说明

     - 10-120-85-2：ResourceManager主节点的hostname。

       可以登录Manager界面，选择“集群 > 服务 > Yarn > 实例”查看“ResourceManager(主)”的“主机名称”获取。

     - 8090：ResourceManager的端口号。

       可以登录Manager界面，选择“集群 > 服务 > Yarn > 配置 > 全部配置”搜索并查看“yarn.resourcemanager.webapp.https.port”参数值获取。

   - 用户能看到哪个队列的任务，要看这个用户是否有这个队列的admin权限。

     说明

     如果当前组件使用了Ranger进行权限控制，需基于Ranger配置相关策略进行权限管理。

   - 运行结果：

     ```
     {
       "apps": {
         "app": [
           {
             "id": "application_1461743120947_0001",
             "user": "spark",
             "name": "Spark-JDBCServer",
             "queue": "default",
             "state": "RUNNING",
             "finalStatus": "UNDEFINED",
             "progress": 10,
             "trackingUI": "ApplicationMaster",
             "trackingUrl": "https://10-120-85-2:8090/proxy/application_1461743120947_0001/",
             "diagnostics": "AM is launched. ",
             "clusterId": 1461743120947,
             "applicationType": "SPARK",
     ```

```
            "applicationTags": "",
            "startedTime": 1461804906260,
            "finishedTime": 0,
            "elapsedTime": 6888848,
            "amContainerLogs": "https://10-120-85-2:8044/node/containerlogs/
container_e12_1461743120947_0001_01_000001/spark",
            "amHostHttpAddress": "10-120-85-2:8044",
            "allocatedMB": 1024,
            "allocatedVCores": 1,
            "runningContainers": 1,
            "memorySeconds": 7053309,
            "vcoreSeconds": 6887,
            "preemptedResourceMB": 0,
            "preemptedResourceVCores": 0,
            "numNonAMContainerPreempted": 0,
            "numAMContainerPreempted": 0,
            "resourceRequests": [
               {
                  "capability": {
                     "memory": 1024,
                     "virtualCores": 1
                  },
                  "nodeLabelExpression": "",
                  "numContainers": 0,
                  "priority": {
                     "priority": 0
                  },
                  "relaxLocality": true,
                  "resourceName": "*"
               }
            ],
            "logAggregationStatus": "NOT_START",
            "amNodeLabelExpression": ""
         },
         {
            "id": "application_1461722876897_0002",
            "user": "admin",
            "name": "QuasiMonteCarlo",
            "queue": "default",
            "state": "FINISHED",
            "finalStatus": "SUCCEEDED",
            "progress": 100,
            "trackingUI": "History",
            "trackingUrl": "https://10-120-85-2:8090/proxy/application_1461722876897_0002/",
            "diagnostics": "Attempt recovered after RM restart",
            "clusterId": 1461743120947,
            "applicationType": "MAPREDUCE",
            "applicationTags": "",
            "startedTime": 1461741052993,
            "finishedTime": 1461741079483,
            "elapsedTime": 26490,
            "amContainerLogs": "https://10-120-85-2:8044/node/containerlogs/
container_e11_1461722876897_0002_01_000001/admin",
            "amHostHttpAddress": "10-120-85-2:8044",
            "allocatedMB": -1,
            "allocatedVCores": -1,
            "runningContainers": -1,
            "memorySeconds": 158664,
            "vcoreSeconds": 52,
            "preemptedResourceMB": 0,
            "preemptedResourceVCores": 0,
            "numNonAMContainerPreempted": 0,
            "numAMContainerPreempted": 0,
            "amNodeLabelExpression": ""
         }
      ]
   }
}
```


- 结果分析：

通过这个接口，可以查询当前集群中Yarn上的任务，并且可以得到如下**表31-5**。

表 31-5 常用信息

参数	参数描述
user	运行这个任务的用户。
applicationType	例如MAPREDUCE或者SPARK等。
finalStatus	可以知道任务是成功还是失败。
elapsedTime	任务运行的时间。

2. 获取Yarn资源的总体信息。
   - 命令：
     ```
     curl -k -i --negotiate -u : "https://10-120-85-102:8090/ws/v1/cluster/metrics"
     ```
   - 运行结果：
     ```
     {
        "clusterMetrics": {
           "appsSubmitted": 2,
           "appsCompleted": 1,
           "appsPending": 0,
           "appsRunning": 1,
           "appsFailed": 0,
           "appsKilled": 0,
           "reservedMB": 0,
           "availableMB": 23552,
           "allocatedMB": 1024,
           "reservedVirtualCores": 0,
           "availableVirtualCores": 23,
           "allocatedVirtualCores": 1,
           "containersAllocated": 1,
           "containersReserved": 0,
           "containersPending": 0,
           "totalMB": 24576,
           "totalVirtualCores": 24,
           "totalNodes": 3,
           "lostNodes": 0,
           "unhealthyNodes": 0,
           "decommissionedNodes": 0,
           "rebootedNodes": 0,
           "activeNodes": 3,
           "rmMainQueueSize": 0,
           "schedulerQueueSize": 0,
           "stateStoreQueueSize": 0
        }
     }
     ```
   - 结果分析：

     通过这个接口，可以查询当前集群中如**表31-6**。

     表 31-6 常用信息

参数	参数描述
appsSubmitted	已经提交的任务数。
appsCompleted	已经完成的任务数。

参数	参数描述
appsPending	正在挂起的任务数。
appsRunning	正在运行的任务数。
appsFailed	已经失败的任务数。
appsKilled	已经被kill的任务数。
totalMB	Yarn资源总的内存。
totalVirtualCores	Yarn资源总的VCore数。

31.2.4 Superior Scheduler REST API 接口介绍

功能简介

REST/HTTP是Superior Scheduler在YARN资源管理器主机和YARN资源管理网络服务端口的一部分。通常以*address:port* as *SS_REST_SERVER*.的形式指示YARN。

下面使用HTTPS作为URL的一部分，并且只有HTTPS将得到支持。

Superior Scheduler 接口

- **查询Application**
  - 查询scheduler engine中的所有application。
    - URL

      GET https://<SS_REST_SERVER>/ws/v1/sscheduler/applications/list

      说明

      “SS_REST_SERVER”即：*ResourceManager IP地址:端口*

      - ResourceManager IP地址：可登录FusionInsight Manager界面，选择“集群 > 服务 > Yarn > 实例”查看任一ResourceManager的业务IP获取。
      - 端口：ResourceManager的HTTPS端口。可登录FusionInsight Manager界面，选择“集群 > 服务 > Yarn > 配置 > 全部配置”搜索并查看“yarn.resourcemanager.webapp.https.port”参数值获取。
    - 输入

      无
    - 输出

```
JSON Response:

{
 "applicationlist": [
  {
   "id": "1020201_0123_12",
   "queue": "root.Q1.Q11",
   "user": "cchen",
   "resource_request": {
       "vcores" : 10,
       "memory" : 100
  },
  "resource_inuse": {
```



```
  "vcores" : 100,
  "memory" : 2000
},
"number_running_container": 100,
"number_pending_request": 10
},
{
"id": "1020201_0123_15",
"queue": "root.Q2.Q21",
"user": "Test",
"resource_request": {
  "vcores" : 4,
  "memory" : 100
},
"resource_inuse": {
  "vcores" : 20,
  "memory" : 200
},
"resource_reserved": {
  "vcores" : 10,
  "memory" : 100
},
"number_running_container": 20,
"number_pending_container": 4,
"number_reserved_container":2
 }
]
}
```


表 31-7 all application 参数

参数属性	参数类型	参数描述
applicationlist	array	application ID数组。
queue	String	application队列名称。
user	String	提交application的用户名称。
resource_request	object	当前所需要的资源，包括vcores、内存等。
resource_inuse	object	当前所使用的资源，包括vcores、内存等。
resource_reserved	object	当前所预留的资源，包括vcores、内存等。
number_running_container	int	正在运行的container的总数。这反映了superior引擎的判定数量。
number_pending_request	int	挂起申请的总数。这是所有分配请求总和。
number_reserved_container	int	预留container的总数。这反映了superior引擎的判定数量。
id	String	application ID。

- 查询scheduler engine中的单个application。

- URL

  GET https://<SS_REST_SERVER>/ws/v1/sscheduler/applications/*{application_id}*

- 输入

  无

- 输出

```
JSON Response：

{
 "applicationlist": [
  {
   "id": "1020201_0123_12",
   "queue": "root.Q1.Q11",
   "user": "cchen",
   "resource_request": {
       "vcores" : 3,
       "memory" : 3072
  },
  "resource_inuse": {
    "vcores" : 100,
    "memory" : 2048
  },
  "number_running_container": 2,
  "number_pending_request": 3,
  "number_reserved_container":1
  "master_container_id": 23402_3420842
  "master_container_resource": node1.domain.com
  "blacklist": [
    {
     "resource": "node5"
    },
    {
     "resource": "node8"
    }
  ],
  "demand": [
    {
     "priority": 1,
     "ismaster": true,
     "capability": {
     "vcores": 2,
      "memory": 2048
    },
    "count": 1,
    "relaxlocality": true,
    "locality": [
     {
      "target": "node1",
      "count": 1,
      "strict": false
     }
    ],
    "resourceselect": "label1",
    "pending_reason": "application limit reached",
    "reserved_resource": {
     "vcores":1,
      "memory":1024
    },
    "reservations":[
    "id": "23402_3420878",
    "resource": "node1.domain.com",
    "reservedAmount": {
     "vcores":1,
     "memory":1024
    }
```



```
 ]
},
{
 "priority": 1,
 "ismaster": false,
 "capability": {
 "vcores": 1,
 "memory": 1024
},
"count": 2,
"relaxlocality": true,
"locality": [
 {
  "target": "node1",
  "count": 1,
  "strict": false
 },
 {
  "target": "node2",
  "count": 1,
  "strict": false
 },
 {
  "target": "rackA",
  "count": 2,
  "strict": false
 },
],
"resourceselect": "label1",
"pending_reason": "no available resource"
}
],
"containers": [
 {
  "id": "23402_3420842",
  "resource": "node1.domain.com",
  "capability": {
   "vcores": 1,
   "memory": 1024
 }
},
{
  "id": "23402_3420853",
  "resource": "node2.domain.com",
  "capability": {
   "vcores": 1,
   "memory": 1024
  }
 }
]
}
}
```


- 异常

  未找到应用程序。

表 31-8 single application 参数

参数属性	参数类型	参数描述
application	object	application对象。
id	String	application ID。
queue	String	application队列名称。

参数属性	参数类型	参数描述
user	String	application的用户名称。
resource_request	object	当前所申请的资源，包括vcores、内存等。
resource_inuse	object	当前所使用的资源，包括vcores、内存等。
resource_reserved	object	当前所预留的资源，包括vcores、内存等。
number_running_container	int	正在运行的container的总数。这反映了superior引擎的判定数量。
number_pending_request	int	挂起申请的总数。这反映了superior引擎的判定数量。
number_reserved_container	int	预留container的总数。这反映了superior引擎的判定数量。
master_container_id	String	总containerID。
master_container_resource	String	运行的主container的主机名。
demand	array	demand对象数组。
priority	int	请求的优先级。
ismaster	boolean	判断是否为application master需求。
capability	object	Capability对象。
vcores, memory, ..	int	数值可消耗资源属性，给该命令定义分配“单元”。
count	int	单元所需的数量。
relaxlocality	boolean	本地化需求优先，如果不能满足则不强制满足。
locality	object	本地化对象。
target	string	本地化目标的名称（即：节点1，框架1）。
count	int	资源“单元”数量与所需的本地需求。
strict	boolean	是否强制本地性。
resourceselect	String	application资源选择。
pending_reason	String	该application pending的理由。
resource_reserved	object	当前需求的预留资源，包括vcores、内存等。

参数属性	参数类型	参数描述
reservations	array	预留container对象的数组。
reservations:id	String	预留container的ID。
reservations:resource	String	container的分配地址。
reservations:reserveAmount	object	预留项的总数。
containers	array	分配container对象的数组。
containers:id	String	containerID。
containers:resource	String	container分配的位置。
containers:capability	object	Capability对象。
containers:vcores,memory…	int	分配给该container的可消耗数值型资源属性。

- **查询**Queue
  - 查询scheduler engine中的所有queue，包括叶子节点和所有中间队列。
    - URL
      GET https://<SS_REST_SERVER>/ws/v1/sscheduler/queues/list
    - 输入
      无
    - 输出
      JSON Response:

```
{
  "queuelist": [
    {
      "name": "root.default",
      "eopen_state": "OPEN",
      "eactive_state": "ACTIVE",
      "open_state": "OPEN",
      "active_state": "ACTIVE",
      "number_pending_application": 2,
      "number_running_application": 10,
      "number_pending_request": 2,
      "number_running_container": 10,
      "number_reserved_container":1,
      "resource_inuse" {
          "vcores": 10,
          "memory": 10240
      },
      "resource_request" {
          "vcores": 2,
          "memory": 2048
      },
      "resource_reserved" {
          "vcores": 1
          "memory": 1024
```

```
    }

  },
  {
    "name": "root.dev",
    "eopen_state": "OPEN",
    "eactive_state": "INACTIVE",
    "open_state": "OPEN",
    "active_state": "INACTIVE",
    "number_pending_application": 2,
    "number_running_application": 10,
    "number_pending_request": 2,
    "number_running_container": 10,
    "number_reserved_container":0,
    "resource_inuse" {
        "vcores": 10,
        "memory": 10240
    },
    "resource_request" {
        "vcores": 2,
        "memory": 2048
    },
    "resource_reserved" {
        "vcores": 0
        "memory": 0
    }
  },
  {
    "name": "root.qa",
    "eopen_state" : "CLOSED",
    "eactive_state" : "ACTIVE",
    "open_state": "CLOSED",
    "active_state": "ACTIVE",
    "number_pending_application": 2,
    "number_running_application": 10,
    "number_pending_request": 2,
    "number_running_container": 10,
    "number_reserved_container":0,
    "resource_inuse" {
        "vcores": 10,
        "memory": 10240
    },
    "resource_request" {
        "vcores": 2,
        "memory": 2048
    },
    "resource_reserved" {
        "vcores": 1
        "memory": 1024
    }


  },

 ]
}
```

表 31-9 all queues 参数

参数属性	参数类型	参数描述
queuelist	array	队列名称列表。
name	String	队列名称。

参数属性	参数类型	参数描述
open_state	String	队列的内在状态（自身状态）。表示队列的有效状态为OPEN或CLOSED。CLOSED状态的队列不接受任何新的allocation请求。
eopen_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。CLOSED状态的队列不接受任何新的allocation请求。
active_state	String	队列的内在状态（自身状态）。表示队列的有效状态为ACTIVE或INACTIVE。INACTIVE状态的队列不能调度任何应用程序。
eactive_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。INACTIVE状态的队列不能调度任何应用程序。
number_pending_application	int	挂起应用的总和。
number_running_application	int	正在运行应用的总和。
number_pending_request	int	挂起请求的总和。
number_running_container	int	正在运行container的总和。
numbert_reserved_container	int	预留container的总和。
resource_request	object	以vcores和内存等形式在队列中挂起的资源请求。
resource_inuse	object	以vcores和内存等形式在队列中使用的资源。
resource_reserved	object	以vcores和内存等形式在队列中预留的资源。
active_state	String	描述表示队列ACTIVE或INACTIVE状态。一个INACTIVE队列不能调度任何分配请求。

- 查询scheduler engine中的单个queue，包括叶子节点和所有中间队列。
  - URL

    GET https://<SS_REST_SERVER>/ws/v1/sscheduler/queues/*{queuename}*
  - 输入

    无
  - 输出

    JSON Response:


```
{
  "queue": {
"name": "root.default",
"eopen_state": "CLOSED",
"eactive_state": "INACTIVE",
    "open_state": "CLOSED",
"active_state": "INACTIVE",
"leaf_queue" : yes,
    "number_pending_application": 100,
"number_running_application": 10,
    "number_pending_request": 10,
"number_running_container": 10,
"number_reserved_container:1,
"resource_inuse" {
          "vcores": 10,
          "memory": 10240
      },
      "resource_request" {
          "vcores": 2,
          "memory": 2048
      },
      "resource_reserved" {
           "vcores": 1,
           "memory": 1024
      }

    "configuration": {
 "description": "Production spark queue",
      "max_pending_application": 10000,
      "max_running_application": 1000,
      "allocation_order_policy": "FIFO",
      "default_resource_select": "label1",
      "max_master_share": 10%,
      "max_running_application_per_user": -1,
      "max_allocation_unit": {
           "vcores": 32,
           "memory": 128000
      },
      "user_acl": [
        {
          "user": "user1"
        },
        {
          "group": "group1"
        }
      ],
      "admin_acl": [
        {
          "user": "user2"
        },
        {
          "group": "group2"
        }
      ]
    }
  }
}
```

- 异常

  未找到队列。

表 31-10 single queue 参数

参数属性	参数类型	参数描述
queue	object	队列对象。

参数属性	参数类型	参数描述
name	String	队列名称。
description	String	队列描述。
open_state	String	队列的内在状态（自身状态）。表示队列的有效状态为OPEN或CLOSED。CLOSED状态的队列不接受任何新的allocation请求。
eopen_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。CLOSED状态的队列不接受任何新的allocation请求。
active_state	String	队列的内在状态（自身状态）。表示队列的有效状态为ACTIVE或INACTIVE。INACTIVE状态的队列不能调度任何应用程序。
eactive_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。INACTIVE状态的队列不能调度任何应用程序。
leaf_queue	boolean	表示队列是否在树节点或中间队列。表示叶子节点队列。
number_pending_application	int	当前的挂起请求数量。如果是中间队列/父队列，这是所有子队列的集合。
number_running_application	int	当前正在运行应用的数量。如果是中间队列/父队列，这是所有子队列的集合。
number_pending_request	int	挂起命令的数量；每个未完成命令的总计数。如果是中间队列/父队列，这是所有子队列的集合。
number_running_container	int	正在运行container的数量。如果是中间队列/父队列，这是所有子队列的集合。
number_reserved_container	int	预留container的数量。如果是中间队列/父队列，这是所有子队列的集合。
resource_request	object	以vcores和内存等形式在队列中挂起的资源请求。
resource_inuse	object	以vcores和内存等形式在队列中使用的资源。
resource_reserved	object	以vcores和内存等形式预留在队列中的资源。
configuration	object	队列配置目标。

参数属性	参数类型	参数描述
max_pending_application	int	最大挂起应用数。如果是中间队列/父队列，这是所有子队列的集合。
max_running_application	int	最大运行应用数。如果是中间队列/父队列，这是所有子队列的集合。
allocation_order_policy	String	分配策略，可以使用FIFO原则，PRIORITY原则或者FAIR原则。
max_running_application_per_user	int	每个使用者运行应用的最大数量。
max_master_share	string	该队列共享的百分比。
max_allocation_unit	object	单个container允许的最大资源，该资源以vcores和内存等形式存在。
default_resource_select	String	缺省资源选择表达式。它被使用在当应用没有被指定一个提交区间值时。
user_acl	array	队列中被给予user权限的使用者。
admin_acl	array	该队列中被给予admin权限的使用者。
group	String	用户组名称。
user	String	用户名称。

- **查询Resource Pool**
  - 查询scheduler engine中所有resource pool。
    - URL

      GET https://<SS_REST_SERVER>/ws/v1/sscheduler/resourcepools/list
    - 输入

      无
    - 输出

      JSON Response:

```
{
  "resourcepool_list": [
    {
      "name": "pool1",
      "description": "resource pool for crc",
      "number_member": 5,
      "members": [
        {
          "resource": "node1"
        },
        {
```

```
      "resource": "node2"
    },
    {
      "resource": "node3"
    },
    {
      "resource": "node4"
    },
    {
      "resource": "node5"
    }
  ],
  "available_resource": {
    "vcores": 60,
    "memory": 60000
  },
  "total_resource": {
    "vcores": 100,
    "memory": 128000
  },
  "configuration": {
    "resources": [
      {
        "resource": "node1"
      },
      {
        "resource": "node[2-5]"
      }
    ],
    "resource_select": "label1"
  }
},
{
  "name": "pool2",
  "description": "resource pool for erc",
  "number_member": 4
  "members": [
    {
      "resource": "node6"
    },
    {
      "resource": "node7"
    },
    {
      "resource": "node8"
    },
    {
      "resource": "node9"
    }
  ],
  "available_resource": {
    "vcores": 56,
    "memory": 48000
  },
  "total_resource": {
     "vcores": 100,
     "memory": 128000
  },
  "configuration": {
    "resources": [
      {
        "resource": "node6"
      },
      {
        "resource": "node[7-9]"
      }
    ],
    "resource_select": "label1"
  }
```

```
  },
  {
    "name": "default",
    "description": "system-generated",
    "number_member": 1,
    "members": [
      {
        "resource": "node0"
      }
    ],
    "available_resource": {
      "vcores": 8,
      "memory": 8192
    },
    "total_resource": {
      "vcores": 16,
      "memory": 12800
    }
  }
 ]
}
```

表 31-11 all resource pools 参数

参数属性	参数类型	参数描述
resourcepool_list	array	resource pool对象数组。
name	String	resource pool名称。
number_member	int	resource pool成员数量。
description	String	resource pool描述。
members	array	当前resource pool成员的资源名称数组。
resource	String	资源名称。
available_resource	object	该resource pool中当前可使用的资源。
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中可用资源的属性，该属性的值以数字表示。
total_resource	object	该resource pool所有资源。
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中总资源的属性，该属性的值以数字表示。
configuration	object	配置目标。
resources	array	所配置的资源名称pattern数组。
resource	String	资源名称模式。
resource_select	String	资源选择表达式。

– 查询scheduler engine中单个resource pool

  - URL

    GET https://<SS_REST_SERVER>/ws/v1/sscheduler/resourcepools/*{resourcepoolname}*

  - 输入

    无

  - 输出

```
JSON Response:
{
  "resourcepool": {
    "name": "pool1",
    "description": "resource pool for crc",
    "number_member": 5
    "members": [
      {
        "resource": "node1"
      },
      {
        "resource": "node2"
      },
      {
        "resource": "node3"
      },
      {
        "resource": "node4"
      },
      {
        "resource": "node5"
      }
    ],
    "available_resource": {
      "vcores": 60,
      "memory": 60000
},
"total_resource": {
  "vcores": 100,
  "memory": 128000
},
    "configuration": {
      "resources": [
        {
          "resource": "node6"
        },
        {
          "resource": "node[7-9]"
        }
      ],
      "resource_select": "label1"
    }
  }
}
```

  - 异常

    未找到resource pool。

表 31-12 single resource pool 参数

参数属性	参数类型	参数描述
resourcepool	object	resource pool对象。

参数属性	参数类型	参数描述
name	String	resource pool名称。
description	String	resource pool描述。
number_member	int	resource pool成员数量。
members	array	该resource pool现任成员的资源名称数组。
resource	String	资源名称。
available_resource	object	该resource pool当前可用资源。
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中可用资源的属性，该属性的值以数字表示。
total_resource	object	该resource pool中所有资源。
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中总资源的属性，该属性的值以数字表示。
configuration	object	配置目标。
resources	array	所配置的资源名称pattern数组。
resource	String	资源名称模式。
resource_select	String	资源选择表达式。

- **查询policiesxmlconf**
  - URL

    GET https://<SS_REST_SERVER>/ws/v1/sscheduler/policiesxmlconf/list
  - 输入

    无
  - 输出

```
<policies>
  <policlist>
    <resourcepool>default</resourcepool>
    <queues>
      <name>default</name>
      <fullname>root.default</fullname>
      <share>20.0</share>
      <reserve>memory 0,vcores 0 : 0.0%</reserve>
      <minimum>memory 0,vcores 0 : 20.0%</minimum>
      <maximum>memory 0,vcores 0 : 100.0%</maximum>
      <defaultuser>
        <maximum>100.0%</maximum>
        <weight>1.0</weight>
      </defaultuser>
    </queues>
  </policlist>
</policies>
```


32 YARN 开发指南（普通模式）

32.1 YARN 应用开发简介

简介

Yarn是一个分布式的资源管理系统，用于提高分布式的集群环境下的资源利用率，这些资源包括内存、IO、网络、磁盘等。其产生的原因是为了解决原MapReduce框架的不足。最初MapReduce的committer还可以周期性的在已有的代码上进行修改，可是随着代码的增加以及原MapReduce框架设计的不足，在原MapReduce框架上进行修改变得越来越困难，所以MapReduce的committer决定从架构上重新设计MapReduce，使下一代的MapReduce(MRv2/Yarn)框架具有更好的扩展性、可用性、可靠性、向后兼容性和更高的资源利用率，以及能支持除了MapReduce计算框架外的更多的计算框架。

基本概念

- **ResourceManager（RM）**

  RM是一个全局的资源管理器，负责整个系统的资源管理和分配。它主要由两个组件构成：调度器（Scheduler）和应用程序管理器（Applications Manager，ASM）。

- **ApplicationMaster（AM）**

  用户提交的每个应用程序均包含一个AM，主要功能包括：

  - 与RM调度器协商以获取资源（用Container表示）。
  - 将得到的资源进一步分配给内部任务。
  - 与NM通信以启动/停止任务。
  - 监控所有任务的运行状态，并在任务运行失败时重新为任务申请资源以重启任务。

- **NodeManager（NM）**

  NM是每个节点上的资源和任务管理器，一方面，它会定时地向RM汇报本节点上的资源使用情况和各个Container的运行状态；另一方面，它会接收并处理来自AM的Container启动/停止等各种请求。

- **Container**

Container是YARN中的资源抽象，它封装了某个节点上的多维度资源，如内存、CPU、磁盘、网络等，当AM向RM申请资源时，RM为AM返回的资源便是用Container表示的。

32.2 YARN 接口介绍

32.2.1 YARN Command 介绍

您可以使用YARN Commands对YARN集群进行一些操作，例如启动ResourceManager、提交应用程序、中止应用、查询节点状态、下载container日志等操作。

完整和详细的Command描述可以参考官网文档：

http://hadoop.apache.org/docs/r3.1.1/hadoop-yarn/hadoop-yarn-site/YarnCommands.html

常用 Command

YARN Commands可同时供普通用户和管理员用户使用，它包含了少量普通用户可以执行的命令，比如jar、logs。而大部分只有管理员有权限使用。

用户可以通过以下命令查看YARN用法和帮助：

yarn --help

用法：进入Yarn客户端的任意目录，执行**source**命令导入环境变量，直接运行命令即可。

格式如下所示：

yarn [--config *confdir***]** *COMMAND*

其中COMMAND可以为：

说明

其中版本号8.1.0.1为示例，具体以实际环境的版本号为准。

表 32-1 常用 Command 描述

COMMAND	描述
resourcemanager	运行一个ResourceManager。 备注：以**omm**用户执行服务端命令前需export环境变量（客户端需要有**omm**用户的执行权限），例如： • **export YARN_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_ResourceManager/etc** • **export HADOOP_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_ResourceManager/etc**

COMMAND	描述
nodemanager	运行一个NodeManager。 备注：以**omm**用户执行服务端命令前需export环境变量（客户端需要有**omm**用户的执行权限），例如： ● **export YARN_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_NodeManager/etc** ● **export HADOOP_CONF_DIR=${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/1_10_NodeManager/etc**
rmadmin	管理员工具（动态更新信息）。
version	打印版本信息。
jar <jar>	运行jar文件。
logs	获取container日志。
classpath	打印获取Hadoop JAR包和其他库文件所需的CLASSPATH路径。
daemonlog	获取或者设置服务LOG级别。
CLASSNAME	运行一个名字为CLASSNAME的类。
top	运行集群利用率监控工具。
-Dmapreduce.job.hdfs-servers	如果对接了OBS，而服务端依然使用HDFS，那么需要显式在命令行使用该参数指定HDFS的地址。格式为hdfs://*{NAMESERVICE}*。其中{NAMESERVICE}为hdfs nameservice名称。 如果当前的HDFS具有多个nameservice，那么需要指定所有的nameservice，并以‘，’隔开。例如：hdfs://nameservice1,hdfs://nameservice2

Superior Scheduler Command

Superior Scheduler引擎提供了输出Superior Scheduler引擎具体信息的CLI。为了执行Superior命令，需要使用“<HADOOP_HOME>/bin/superior”脚本。

以下为superior命令格式：

```
<HADOOP_HOME>/bin/superior

Usage: superior [COMMAND | -help]
  Where COMMAND is one of:
  resourcepool                  prints resource pool status
  queue                         prints queue status
  application                   prints application status
  policy                        prints policy status
```

不带参数调用大多数命令时会显示帮助信息。

- Superior ***resourcepool***命令：
  该命令显示Resource Pool和相关策略的相关状态以及配置信息。

说明

Superior resourcepool命令仅用于管理员用户及拥有yarn管理权限的用户。

用法输出：

```
>superior resourcepool

Usage: resourcepool [-help]
                    [-list]
                    [-status <resourcepoolname>]
 -help                      prints resource pool usage
 -list                     prints all resource pool summary report
 -status <resourcepoolname>   prints status and configuration of specified
                           resource pool
```

- ***resourcepool -list***以表格格式中显示Resource Pool摘要。示例如下：

  ```
  > superior resourcepool -list
  NAME       NUMBER_MEMBER     TOTAL_RESOURCE           AVAILABLE_RESOURCE
  Pool1      4                 vcores 30,memory 1000    vcores 21,memory 80
  Pool2      100                vcores 100,memory 12800  vcores 30,memory 1000
  default    2                 vcores 64,memory 128     vcores 40,memory 28
  ```

- ***resourcepool -status <resourcepoolname>***以列表格式显示资源库详细信息。示例如下：

  ```
  > superior resourcepool -status default
  NAME: default
  DESCRIPTION: System generated resource pool
  TOTAL_RESOURCE: vcores 64,memory 128
  AVAILABLE_RESOURCE: vcores 40,memory 28
  NUMBER_MEMBER: 2
  MEMBERS: node1,node2
  CONFIGURATION:
  |-- RESOURCE_SELECT:
  |__RESOURCES:
  ```

- Superior ***queue***命令

  该命令输出分层队列信息。

  用法输出：

  ```
  >superior queue

  Usage: queue [-help]
             [-list] [-e] [[-name <queue_name>] [-r|-c]]
             [-status <queue_name>]
   -c                  only work with -name <queue_name> option. If this
                       option is used, command  will print information of
                       specified queue and its direct children.
   -e                  only work with -list or -list -name option. If
                       this option is used, command will print effective
                       state of specified queue and all of its
                       descendants.
   -help                 prints queue sub command usage
   -list               prints queue summary report. This option can work
                       with -name <queue_name> and -r options.
   -name <queue_name>     print specified queue, this can work with -r
                       option. By default, it will print queue's own
                       information. When -r is defined, command will
                       print all of its descendant queues. When -c is
                       defined, it will print its direct children queues.
   -r                  only work with -name <queue_name> option. If this
                       option is used, command will print information of
                       specified queue and all of its descendants.
   -status <queue_name>   prints status of specified queue
  ```

  - ***queue -list***以表格格式输出队列摘要信息。命令将基于队列分层样式输出信息。用户可通过SUBMIT ACL或ADMIN ACL的队列权限查看队列。示例如下：

    ```
    > superior queue -list
    NAME         STATE           NRUN_APP     NPEND_APP     NRUN_CONTAINER
    ```



```
NPEND_REQUEST    RES_INUSE              RES_REQUEST
root         OPEN|ACTIVE      10          20          100             200             vcores 100,memory
1000    vcores 200,memory 2000
root.Q1      OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q11  OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q12  CLOSE|INACTIVE   0           0           0               0               vcores 0,memory
0       vcores 0,memory 0
root.Q2      OPEN|INACTIVE    5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q2.Q21  OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
```

- ***queue -list -name root.Q1只***输出root.Q1。

```
> superior queue -list -name root.Q1
NAME            STATE           NRUN_APP     NPEND_APP       NRUN_CONTAINER
NPEND_REQUEST    RES_INUSE           RES_REQUEST
root.Q1         OPEN|ACTIVE     5           10          50              100             vcores 50,memory
500 vcores 100,memory 1000
```

- ***queue -list -name root.Q1 -r***将输出root.Q1及其所有的分支。

```
> superior queue -list -name root.Q1 -r
NAME          STATE           NRUN_APP     NPEND_APP  NRUN_CONTAINER
NPEND_REQUEST    RES_INUSE              RES_REQUEST
root.Q1      OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q11  OPEN|ACTIVE      5           10          50              100             vcores 50,memory
500     vcores 100,memory 1000
root.Q1.Q12  CLOSE|INACTIVE   0           0           0               0               vcores 0,memory
0       vcores 0,memory 0
```

- ***queue -list -name root -c***将会输出root及其直系子目录。

```
> superior queue -list -name root -c
NAME                 STATE            NRUN_APP     NPEND_APP     NRUN_CONTAINER
NPEND_REQUEST    RES_INUSE              RES_REQUEST
root                OPEN|ACTIVE       10          20          100             200             vcores
100,memory 1000   vcores 200,memory 2000
root.Q1             OPEN|ACTIVE       5           10          50              100             vcores
50,memory 500     vcores 100,memory 1000
root.Q2             OPEN|INACTIVE     5           10          50              100             vcores
50,memory 500     vcores 100,memory 1000
```

- **queue -status <queue_name>**将会输出具体队列状态和配置。

  用户可通过SUBMIT ACL权限查看除队列ACL外的细节信息。

  用户还可通过ADMIN ACL的队列权限查看包括ACL在内的队列细节信息。

```
> superior queue -status root.Q1
NAME: root.Q1
OPEN_STATE:CLOSED
ACTIVE_STATE: INACTIVE
EOPEN_STATE: CLOSED
EACTIVE_STATE: INACTIVE
LEAF_QUEUE: Yes
NUMBER_PENDING_APPLICATION: 100
NUMBER_RUNNING_APPLICATION: 10
NUMBER_PENDING_REQUEST: 10
NUMBER_RUNNING_CONTAINER: 10
NUMBER_RESERVED_CONTAINER: 0
RESOURCE_REQUEST: vcores 3,memory 3072
RESOURCE_INUSE: vcores 2,memory 2048
RESOURCE_RESERVED: vcores 0,memory 0
CONFIGURATION:
|-- DESCRIPTION: Spark session queue
|-- MAX_PENDING_APPLICATION: 10000
|--MAX_RUNNING_APPLICATION: 1000
|--ALLOCATION_ORDER_POLICY: FIFO
|--DEFAULT_RESOURCE_SELECT: label1
|--MAX_MASTER_SHARE: 10%
|--MAX_RUNNING_APPLICATION_PER_USER : -1
```

```
|--MAX_ALLOCATION_UNIT: vcores 32,memory 12800
|--ACL_USERS: user1,user2
|--ACL_USERGROUPS: usergroup1,usergroup2
|-- ACL_ADMINS: user1
|--ACL_ADMINGROUPS: usergroup1
```

- Superior ***application***命令

  该命令输出应用相关信息。

  用法输出：

```
>superior application

Usage: application [-help]
                [-list]
                [-status <application_id>]
 -help                    prints application sub command usage
 -list                    prints all application summary report
 -status <application_id>   prints status of specified application
```

  用户可通过应用的浏览访问权限查看应用相关信息。

  - ***application -list***以表的形式提供所有应用的信息摘要：

```
> superior application -list
ID                                  QUEUE            USER     NRUN_CONTAINER
NPEND_REQUEST              NRSV_CONTAINER       RES_INUSE
RES_REQUEST                 RES_RESERVED
application_1482743319705_0005           root.SEQ.queueB   hbase    1
100                        0                  vcores 1,memory 1536             vcores 2000,memory
409600       vcores 0,memory 0
application_1482743319705_0006           root.SEQ.queueB   hbase    0
1                          0                  vcores 0,memory 0               vcores 1,memory 1536
vcores 0,memory 0
```

  - ***application -status <app_id>***命令输出指定应用的详细信息。示例如下：

```
> superior application -status application_1443067302606_0609
ID: application_1443067302606_0609
QUEUE: root.Q1.Q11
USER: cchen
RESOURCE_REQUEST: vcores 3,memory 3072
RESOURCE_INUSE: vcores 2,memory 2048
RESOURCE_RESERVED:vcores 1, memory 1024
NUMBER_RUNNING_CONTAINER: 2
NUMBER_PENDING_REQUEST: 3
NUMBER_RESERVED_CONTAINER: 1
MASTER_CONTAINER_ID: application_1443067302606_0609_01
MASTER_CONTAINER_RESOURCE: node1.domain.com
BLACKLIST: node5,node8
DEMANDS:
|-- PRIORITY: 20
|-- MASTER: true
|-- CAPABILITY: vcores 2, memory 2048
|-- COUNT: 1
|-- RESERVED_RES : vcores 1, memory 1024
|-- RELAXLOCALITY: true
|-- LOCALITY: node1/1
|-- RESOURCESELECT: label1
|-- PENDINGREASON: “application limit reached”
|-- ID: application_1443067302606_0609_03
|-- RESOURCE: node1.domain.com
|-- RESERVED_RES: vcores 1, memory 1024
|
|--PRIORITY: 1
|-- MASTER: false
|-- CAPABILITY: vcores 1,memory 1024
|-- COUNT: 2
|-- RESERVED_RES: vcores 0, memory 0
|-- RELAXLOCLITY: true
|--LOCALITY: node1/1,node2/1,rackA/2
|-- RESOURCESELECT: label1
|-- PENDINGREASON: “no available resource”
```



```
CONTAINERS:
|-- ID: application_1443067302606_0609_01
|-- RESOURCE: node1.domain.com
|-- CAPABILITY: vcores 1,memory 1024
|
|-- ID: application_1443067302606_0609_02
|-- RESOURCE: node2.domain.com
|-- CAPABILITY: vcores 1,memory 1024
```

- Superior ***policy*** 命令

  该命令输出决策相关信息。

  说明

  Superior policy命令仅限管理员用户及拥有Yarn管理权限的用户使用。

  用法输出：

```
>superior policy

Usage: policy [-help]
            [-list <resourcepoolname>] [-u] [-detail]
            [-status <resourcepoolname>]
 -detail                    only work with -list option to show a
                           summary information of resource pool
                           distribution on queues, including reserve,
                           minimum and maximum
 -help                      prints policy sub command usage
 -list <resourcepoolname>     prints a summary information of resource
                           pool distribution on queue
 -status <resourcepoolname>   prints pool distribution policy
                           configuration and status of specified
                           resource pool
 -u                         only work with -list option to show a
                           summary information of resource pool
                           distribution on queues and also user
                           accounts
```

  - ***policy -list <resourcepoolname>***输出队列分布信息摘要。示例如下：

```
>superior policy -list default
NAME: default
TOTAL_RESOURCE: vcores 16,memory 16384
AVAILABLE_RESOURCE: vcores 16,memory 16384

NAMERES_INUSERES_REQUEST
root.defaultvcores 0,memory 0vcores 0,memory 0
root.productionvcores 0,memory 0vcores 0,memory 0
root.production.BU1vcores 0,memory 0vcores 0,memory 0
root.production.BU2 vcores 0,memory 0vcores 0,memory 0
```

  - ***policy -list <resourcepoolname> -u***输出用户级信息摘要。

```
> superior policy -list default -u
NAME: default
TOTAL_RESOURCE: vcores 16,memory 16384
AVAILABLE_RESOURCE: vcores 16,memory 16384

NAMERES_INUSERES_REQUEST
root.defaultvcores 0,memory 0vcores 0,memory 0
root.default.[_others_]vcores 0,memory 0vcores 0,memory 0
root.productionvcores 0,memory 0vcores 0,memory 0
root.production.BU1vcores 0,memory 0vcores 0,memory 0
root.production.BU1.[_others_]vcores 0,memory 0vcores 0,memory 0
root.production.BU2vcores 0,memory 0vcores 0,memory 0
root.production.BU2.[_others_]vcores 0,memory 0vcores 0,memory 0
```

  - ***policy -status <resourcepoolname>*** 输出指定资源库的策略详细资料。

```
> superior policy -status pool1
NAME: pool1
TOTAL_RESOURCE: vcores 64,memory 128
AVAILABLE_RESOURCE: vcores 40,memory 28
QUEUES:
```



```
|-- NAME: root.Q1
|-- RESOURCE_USE: vcores 20, memory 1000
|-- RESOURCE_REQUEST: vcores 2,memory 100
|--RESERVE: vcores 10, memory 4096
|--MINIMUM: vcore 11, memory 4096
|--MAXIMUM: vcores 500, memory 100000
|--CONFIGURATION:
|-- SHARE: 50%
|-- RESERVE: vcores 10, memory 4096
|-- MINIMUM: vcores 11, memory 4096
|-- MAXIMUM: vcores 500, memory 100000
|-- QUEUES:
|-- NAME: root.Q1.Q11
|-- RESOURCE_USE: vcores 15, memory, 500
|-- RESOURCE_REQUEST: vcores 1, memory 50
|-- RESERVE: vcores 0, memory 0
|-- MINIMUM: vcores 0, memory 0
|-- MAXIMUM: vcores -1, memory -1
|-- USER_ACCOUNTS:
|-- NAME: user1
|-- RESOURCE_USE: vcores 1, memory 10
|-- RESOURCE_REQUEST: vcores 1, memory 50
|
|-- NAME: OTHERS
|--RESOURCE_USE: vcores 0, memory 0
|- RESOURCE_REQUEST: vcores 0, memory 0
|-- CONFIGURATION:
|-- SHARE: 100%
|-- USER_POLICY:
|-- NAME: user1
|-- WEIGHT: 10
|
|-- NAME: OTHERS
|-- WEIGHT: 1
|-- MAXIMUM: vcores 10, memory 1000
```

32.2.2 YARN Java API 接口介绍

关于YARN的详细API可以直接参考官方网站上的描述：

http://hadoop.apache.org/docs/r3.1.1/api/index.html

常用接口

YARN常用的Java类有如下几个。

- **ApplicationClientProtocol**

  用于Client与ResourceManager之间。Client通过该协议可实现将应用程序提交到ResourceManager上，查询应用程序的运行状态或者中止应用程序等功能。

  表 32-2 ApplicationClientProtocol 常用方法

方法	说明
forceKillApplication(KillApplicationRequest request)	Client通过此接口请求RM中止一个已提交的任务。
getApplicationAttemptReport(GetApplicationAttemptReportRequest request)	Client通过此接口从RM获取指定ApplicationAttempt的报告信息。

方法	说明
getApplicationAttempts(GetApplicationAttemptsRequest request)	Client通过此接口从RM获取所有ApplicationAttempt的报告信息。
getApplicationReport(GetApplicationReportRequest request)	Client通过此接口从RM获取某个应用的报告信息。
getApplications(GetApplicationsRequest request)	Client通过此接口从RM获取满足一定过滤条件的应用的报告信息。
getClusterMetrics(GetClusterMetricsRequest request)	Client通过此接口从RM获取集群的Metrics。
getClusterNodes(GetClusterNodesRequest request)	Client通过此接口从RM获取集群中的所有节点信息。
getContainerReport(GetContainerReportRequest request)	Client通过此接口从RM获取某个Container的报告信息。
getContainers(GetContainersRequest request)	Client通过此接口从RM获取某个ApplicationAttemp的所有Container的报告信息。
getDelegationToken(GetDelegationTokenRequest request)	Client通过此接口获取授权票据，用于container访问相应的service。
getNewApplication(GetNewApplicationRequest request)	Client通过此接口获取一个新的应用ID号，用于提交新的应用。
getQueueInfo(GetQueueInfoRequest request)	Client通过此接口从RM中获取队列的相关信息。
getQueueUserAcls(GetQueueUserAclsInfoRequest request)	Client通过此接口从RM中获取当前用户的队列访问权限信息。
moveApplicationAcrossQueues(MoveApplicationAcrossQueuesRequest request)	移动一个应用到新的队列。
submitApplication(SubmitApplicationRequest request)	Client通过此接口提交一个新的应用到RM。

- ApplicationMasterProtocol

  用于ApplicationMaster与ResourceManager之间。ApplicationMaster使用该协议向ResourceManager注册、申请资源、获取各个任务的运行情况等。

表 32-3 ApplicationMasterProtocol 常用方法

方法	说明
allocate(AllocateRequest request)	AM通过此接口提交资源分配申请。
finishApplicationMaster(FinishApplicationMasterRequest request)	AM通过此接口通知RM其运行成功或者失败。
registerApplicationMaster(RegisterApplicationMasterRequest request)	AM通过此接口向RM进行注册。

- ContainerManagementProtocol

  用于ApplicationMaster与NodeManager之间。ApplicationMaster使用该协议要求NodeManager启动/中止Container或者查询Container的运行状态。

表 32-4 ContainerManagementProtocol 常用方法

方法	说明
getContainerStatuses(GetContainerStatusesRequest request)	AM通过此接口向NM请求Containers的当前状态信息。
startContainers(StartContainersRequest request)	AM通过此接口向NM提供需要启动的containers列表的请求。
stopContainers(StopContainersRequest request)	AM通过此接口请求NM停止一系列已分配的Containers。

样例代码

YARN作业提交的样例代码详细可以参考MapReduce开发指南中的**MapReduce访问多组件样例代码**，实现建立一个MapReduce job，并提交MapReduce作业到Hadoop集群。

32.2.3 YARN REST API 接口介绍

功能简介

通过HTTP REST API来查看更多Yarn任务的信息。目前Yarn的REST接口只能进行一些资源或者任务的查询。完整和详细的接口请直接参考官网上的描述以了解其使用：

http://hadoop.apache.org/docs/r3.1.1/hadoop-yarn/hadoop-yarn-site/WebServicesIntro.html

准备运行环境

1. 在节点上安装客户端，例如安装到“/opt/client”目录。

2. 进入客户端安装目录“/opt/client”，执行下列命令初始化环境变量。
   source *bigdata_env*

操作步骤

1. 获取运行在Yarn上的任务的具体信息。
   - 命令：
     ```
     curl -k -i --negotiate -u : "http://10-120-85-2:8088/ws/v1/cluster/apps/"
     ```

     说明

     - 10-120-85-2：ResourceManager主节点的hostname。
       可以登录Manager界面，选择“集群 > 服务 > Yarn > 实例”查看“ResourceManager(主)”的“主机名称”获取。
     - 8088：ResourceManager的端口号。
       可以登录Manager界面，选择“集群 > 服务 > Yarn > 配置 > 全部配置”搜索并查看“yarn.resourcemanager.webapp.port”参数值获取。
   - 运行结果：
     ```
     {
       "apps": {
         "app": [
           {
             "id": "application_1461743120947_0001",
             "user": "spark",
             "name": "Spark-JDBCServer",
             "queue": "default",
             "state": "RUNNING",
             "finalStatus": "UNDEFINED",
             "progress": 10,
             "trackingUI": "ApplicationMaster",
             "trackingUrl": "http://10-120-85-2:8088/proxy/application_1461743120947_0001/",
             "diagnostics": "AM is launched. ",
             "clusterId": 1461743120947,
             "applicationType": "SPARK",
             "applicationTags": "",
             "startedTime": 1461804906260,
             "finishedTime": 0,
             "elapsedTime": 6888848,
             "amContainerLogs": "http://10-120-85-2:8088/node/containerlogs/
     container_e12_1461743120947_0001_01_000001/spark",
             "amHostHttpAddress": "10-120-85-2:8088",
             "allocatedMB": 1024,
             "allocatedVCores": 1,
             "runningContainers": 1,
             "memorySeconds": 7053309,
             "vcoreSeconds": 6887,
             "preemptedResourceMB": 0,
             "preemptedResourceVCores": 0,
             "numNonAMContainerPreempted": 0,
             "numAMContainerPreempted": 0,
             "resourceRequests": [
               {
                 "capability": {
                   "memory": 1024,
                   "virtualCores": 1
                 },
                 "nodeLabelExpression": "",
                 "numContainers": 0,
                 "priority": {
                   "priority": 0
                 },
                 "relaxLocality": true,
                 "resourceName": "*"
               }
     ```



```
            ],
            "logAggregationStatus": "NOT_START",
            "amNodeLabelExpression": ""
        },
        {
            "id": "application_1461722876897_0002",
            "user": "admin",
            "name": "QuasiMonteCarlo",
            "queue": "default",
            "state": "FINISHED",
            "finalStatus": "SUCCEEDED",
            "progress": 100,
            "trackingUI": "History",
            "trackingUrl": "http://10-120-85-2:8088/proxy/application_1461722876897_0002/",
            "diagnostics": "Attempt recovered after RM restart",
            "clusterId": 1461743120947,
            "applicationType": "MAPREDUCE",
            "applicationTags": "",
            "startedTime": 1461741052993,
            "finishedTime": 1461741079483,
            "elapsedTime": 26490,
            "amContainerLogs": "http://10-120-85-2:8088/node/containerlogs/
container_e11_1461722876897_0002_01_000001/admin",
            "amHostHttpAddress": "10-120-85-2:8088",
            "allocatedMB": -1,
            "allocatedVCores": -1,
            "runningContainers": -1,
            "memorySeconds": 158664,
            "vcoreSeconds": 52,
            "preemptedResourceMB": 0,
            "preemptedResourceVCores": 0,
            "numNonAMContainerPreempted": 0,
            "numAMContainerPreempted": 0,
            "amNodeLabelExpression": ""
        }
    ]
  }
}
```


- 结果分析：

  通过这个接口，可以查询当前集群中Yarn上的任务，并且可以得到如下**表 32-5**。

  表 32-5 常用信息

参数	参数描述
user	运行这个任务的用户。
applicationType	例如MAPREDUCE或者SPARK等。
finalStatus	可以知道任务是成功还是失败。
elapsedTime	任务运行的时间。

2. 获取Yarn资源的总体信息
   - 命令：

     ```
     curl -k -i --negotiate -u : "http://10-120-85-102:8088/ws/v1/cluster/metrics"
     ```

   - 运行结果：


     ```
     {
       "clusterMetrics": {
          "appsSubmitted": 2,
          "appsCompleted": 1,
          "appsPending": 0,
     ```



```
      "appsRunning": 1,
      "appsFailed": 0,
      "appsKilled": 0,
      "reservedMB": 0,
      "availableMB": 23552,
      "allocatedMB": 1024,
      "reservedVirtualCores": 0,
      "availableVirtualCores": 23,
      "allocatedVirtualCores": 1,
      "containersAllocated": 1,
      "containersReserved": 0,
      "containersPending": 0,
      "totalMB": 24576,
      "totalVirtualCores": 24,
      "totalNodes": 3,
      "lostNodes": 0,
      "unhealthyNodes": 0,
      "decommissionedNodes": 0,
      "rebootedNodes": 0,
      "activeNodes": 3,
      "rmMainQueueSize": 0,
      "schedulerQueueSize": 0,
      "stateStoreQueueSize": 0
    }
}
```


- 结果分析：

  通过这个接口，可以查询当前集群中如**表32-6**。

  表 32-6 常用信息

参数	参数描述
appsSubmitted	已经提交的任务数。
appsCompleted	已经完成的任务数。
appsPending	正在挂起的任务数。
appsRunning	正在运行的任务数。
appsFailed	已经失败的任务数。
appsKilled	已经被kill的任务数。
totalMB	Yarn资源总的内存。
totalVirtualCores	Yarn资源总的VCore数。

32.2.4 Superior Scheduler REST API 接口介绍

功能简介

REST/HTTP是Superior Scheduler在YARN资源管理器主机和YARN资源管理网络服务端口的一部分。通常以*address:port* as *SS_REST_SERVER*.的形式指示YARN。

下面使用HTTP作为URL的一部分，并且只有HTTP将得到支持。

Superior Scheduler 接口

- **查询Application**
  - 查询scheduler engine中的所有 application。
    - URL

      GET http://<SS_REST_SERVER>/ws/v1/sscheduler/applications/list

      说明

      “SS_REST_SERVER”即：*ResourceManager IP地址:端口*

      - ResourceManager IP地址：可登录FusionInsight Manager界面，选择“集群 > 服务 > Yarn > 实例”查看任一ResourceManager的业务IP获取。
      - 端口：ResourceManager的HTTP端口。可登录FusionInsight Manager界面，选择“集群 > 服务 > Yarn > 配置 > 全部配置”搜索并查看“yarn.resourcemanager.webapp.port”参数值获取。

    - 输入

      无

    - 输出

      JSON Response：

```
{
 "applicationlist": [
  {
   "id": "1020201_0123_12",
   "queue": "root.Q1.Q11",
   "user": "cchen",
   "resource_request": {
       "vcores" : 10,
       "memory" : 100
  },
  "resource_inuse": {
    "vcores" : 100,
    "memory" : 2000
  },
  "number_running_container": 100,
  "number_pending_request": 10
  },
  {
  "id": "1020201_0123_15",
  "queue": "root.Q2.Q21",
  "user": "Test",
  "resource_request": {
    "vcores" : 4,
    "memory" : 100
  },
  "resource_inuse": {
    "vcores" : 20,
    "memory" : 200
  },
  "resource_reserved": {
    "vcores" : 10,
    "memory" : 100
  },
  "number_running_container": 20,
  "number_pending_container": 4,
  "number_reserved_container":2
  }
 ]
}
```

表 32-7 all application 参数

参数属性	参数类型	参数描述
applicationlist	array	applicationID的数组。
queue	String	application队列名称。
user	String	提交application的用户名称。
resource_request	object	当前所需要的资源，包括vcores、内存等。
resource_inuse	object	当前所使用的资源，包括vcores、内存等。
resource_reserved	object	当前所预留的资源，包括vcores、内存等。
number_running_container	int	正在运行的container的总数。这反映了superior引擎的判定数量。
number_pending_request	int	挂起申请的总数。这是所有分配请求总和。
number_reserved_container	int	预留container的总数。这反映了superior引擎的判定数量。
id	String	application ID。

- 查询scheduler engine中的单个application。
  - URL

    GET http://<SS_REST_SERVER>/ws/v1/sscheduler/applications/*{application_id}*
  - 输入

    无
  - 输出

    JSON Response:

```
{
 "applicationlist": [
 {
  "id": "1020201_0123_12",
  "queue": "root.Q1.Q11",
  "user": "cchen",
  "resource_request": {
      "vcores" : 3,
      "memory" : 3072
 },
 "resource_inuse": {
   "vcores" : 100,
   "memory" : 2048
 },
 "number_running_container": 2,
 "number_pending_request": 3,
 "number_reserved_container":1
 "master_container_id": 23402_3420842
```

```
"master_container_resource": node1.domain.com
"blacklist": [
  {
   "resource": "node5"
  },
  {
   "resource": "node8"
  }
],
"demand": [
  {
   "priority": 1,
   "ismaster": true,
   "capability": {
   "vcores": 2,
    "memory": 2048
  },
  "count": 1,
  "relaxlocality": true,
  "locality": [
   {
    "target": "node1",
    "count": 1,
    "strict": false
   }
  ],
  "resourceselect": "label1",
  "pending_reason": "application limit reached",
  "reserved_resource": {
   "vcores":1,
    "memory":1024
  },
  "reservations":[
  "id": "23402_3420878",
  "resource": "node1.domain.com",
  "reservedAmount": {
   "vcores":1,
   "memory":1024
  }
 ]
},
{
 "priority": 1,
 "ismaster": false,
 "capability": {
 "vcores": 1,
 "memory": 1024
},
"count": 2,
"relaxlocality": true,
"locality": [
 {
  "target": "node1",
  "count": 1,
  "strict": false
 },
 {
  "target": "node2",
  "count": 1,
  "strict": false
 },
 {
  "target": "rackA",
  "count": 2,
  "strict": false
 },
],
"resourceselect": "label1",
"pending_reason": "no available resource"
```



```
 }
],
"containers": [
 {
  "id": "23402_3420842",
  "resource": "node1.domain.com",
  "capability": {
   "vcores": 1,
   "memory": 1024
 }
},
{
  "id": "23402_3420853",
  "resource": "node2.domain.com",
  "capability": {
   "vcores": 1,
   "memory": 1024
  }
 }
]
}
}
```

- 异常

  未找到应用程序。

表 32-8 single application 参数

参数属性	参数类型	参数描述
application	object	application对象。
id	String	application ID。
queue	String	application队列名称。
user	String	application的用户名称。
resource_request	object	当前所申请的资源，包括vcores、内存等。
resource_inuse	object	当前所使用的资源，包括vcores、内存等。
resource_reserved	object	当前所预留的资源，包括vcores、内存等。
number_running_container	int	正在运行的container的总数。这反映了superior引擎的判定数量。
number_pending_request	int	挂起申请的总数。这反映了superior引擎的判定数量。
number_reserved_container	int	预留container的总数。这反映了superior引擎的判定数量。
master_container_id	String	总containerID。
master_container_resource	String	运行的主container的主机名。

参数属性	参数类型	参数描述
demand	array	demand对象数组。
priority	int	请求的优先级。
ismaster	boolean	判断是否为application master需求。
capability	object	Capability对象。
vcores, memory, ..	int	数值可消耗资源属性，给该命令定义分配“单元”。
count	int	单元所需的数量。
relaxlocality	boolean	本地化需求优先，如果不能满足则不强制满足。
locality	object	本地化对象。
target	string	本地化目标的名称（即：节点1，框架1）。
count	int	资源“单元”数量与所需的本地需求。
strict	boolean	是否强制本地性。
resourceselect	String	application资源选择。
pending_reason	String	该application pending的理由。
resource_reserved	object	当前需求的预留资源，包括vcores、内存等。
reservations	Array	预留container对象的数组。
reservations:id	String	预留container的ID。
reservations:resource	String	container的分配地址。
reservations:reserveAmount	Object	预留项的总数。
containers	array	分配container对象的数组。
containers:id	String	containerID。
containers:resource	String	container分配的位置。
containers:capability	object	Capability对象。
containers:vcores,memory…	int	分配给该container的可消耗数值型资源属性。

- **查询Queue**

- 查询scheduler engine中的所有queue，包括叶子节点和所有中间队列。
  - URL
    GET http://<SS_REST_SERVER>/ws/v1/sscheduler/queues/list
  - 输入
    无
  - 输出
    JSON Response:

```
{
 "queuelist": [
  {
    "name": "root.default",
    "eopen_state": "OPEN",
    "eactive_state": "ACTIVE",
    "open_state": "OPEN",
    "active_state": "ACTIVE",
    "number_pending_application": 2,
    "number_running_application": 10,
    "number_pending_request": 2,
    "number_running_container": 10,
    "number_reserved_container":1,
    "resource_inuse" {
        "vcores": 10,
        "memory": 10240
    },
    "resource_request" {
        "vcores": 2,
        "memory": 2048
    },
    "resource_reserved" {
        "vcores": 1
        "memory": 1024
    }

  },
  {
    "name": "root.dev",
    "eopen_state": "OPEN",
    "eactive_state": "INACTIVE",
    "open_state": "OPEN",
    "active_state": "INACTIVE",
    "number_pending_application": 2,
    "number_running_application": 10,
    "number_pending_request": 2,
    "number_running_container": 10,
    "number_reserved_container":0,
    "resource_inuse" {
        "vcores": 10,
        "memory": 10240
    },
    "resource_request" {
        "vcores": 2,
        "memory": 2048
    },
    "resource_reserved" {
        "vcores": 0
        "memory": 0
    }
  },
  {
    "name": "root.qa",
    "eopen_state" : "CLOSED",
    "eactive_state" : "ACTIVE",
    "open_state": "CLOSED",
    "active_state": "ACTIVE",
```



```
    "number_pending_application": 2,
    "number_running_application": 10,
    "number_pending_request": 2,
    "number_running_container": 10,
    "number_reserved_container":0,
    "resource_inuse" {
        "vcores": 10,
        "memory": 10240
    },
    "resource_request" {
        "vcores": 2,
        "memory": 2048
    },
    "resource_reserved" {
        "vcores": 1
        "memory": 1024
    }


  },

 ]
}
```

表 32-9 all queues 参数

参数属性	参数类型	参数描述
queuelist	array	队列名称列表。
name	String	队列名称。
open_state	String	队列的内在状态（自身状态）。表示队列的有效状态为OPEN或CLOSED。CLOSED状态的队列不接受任何新的allocation请求。
eopen_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。CLOSED状态的队列不接受任何新的allocation请求。
active_state	String	队列的内在状态（自身状态）。表示队列的有效状态为ACTIVE或INACTIVE。INACTIVE状态的队列不能调度任何应用程序。
eactive_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。INACTIVE状态的队列不能调度任何应用程序。
number_pending_application	int	挂起应用的总和。
number_running_application	int	正在运行应用的总和。
number_pending_request	int	挂起请求的总和。
number_running_container	int	正在运行container的总和。

参数属性	参数类型	参数描述
numbert_reserved_container	int	预留container的总和。
resource_request	object	以vcores和内存等形式在队列中挂起的资源请求。
resource_inuse	object	以vcores和内存等形式在队列中使用的资源。
resource_reserved	object	以vcores和内存等形式在队列中预留的资源。
active_state	String	描述表示队列ACTIVE或INACTIVE状态。一个INACTIVE队列不能调度任何分配请求。

- 查询scheduler engine中的单个queue，包括叶子节点和所有中间队列。
  - URL

    GET http://<SS_REST_SERVER>/ws/v1/sscheduler/queues/*{queuename}*
  - 输入

    无
  - 输出

```
JSON Response:

{
  "queue": {
"name": "root.default",
"eopen_state": "CLOSED",
"eactive_state": "INACTIVE",
    "open_state": "CLOSED",
"active_state": "INACTIVE",
"leaf_queue" : yes,
    "number_pending_application": 100,
"number_running_application": 10,
    "number_pending_request": 10,
"number_running_container": 10,
"number_reserved_container:1,
"resource_inuse" {
          "vcores": 10,
          "memory": 10240
      },
      "resource_request" {
          "vcores": 2,
          "memory": 2048
      },
      "resource_reserved" {
           "vcores": 1,
           "memory": 1024
      }

   "configuration": {
 "description": "Production spark queue",
      "max_pending_application": 10000,
      "max_running_application": 1000,
      "allocation_order_policy": "FIFO",
      "default_resource_select": "label1",
      "max_master_share": 10%,
```

```
    "max_running_application_per_user": -1,
    "max_allocation_unit": {
         "vcores": 32,
         "memory": 128000
    },
    "user_acl": [
      {
        "user": "user1"
      },
      {
        "group": "group1"
      }
    ],
    "admin_acl": [
      {
        "user": "user2"
      },
      {
        "group": "group2"
      }
    ]
   }
  }
}
```

- 异常

  未找到队列。

表 32-10 single queue 参数

参数属性	参数类型	参数描述
queue	object	队列对象。
name	String	队列名称。
description	String	队列描述。
open_state	String	队列的内在状态（自身状态）。表示队列的有效状态为OPEN或CLOSED。CLOSED状态的队列不接受任何新的allocation请求。
eopen_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。CLOSED状态的队列不接受任何新的allocation请求。
active_state	String	队列的内在状态（自身状态）。表示队列的有效状态为ACTIVE或INACTIVE。INACTIVE状态的队列不能调度任何应用程序。
eactive_state	String	队列的外在状态（父队列状态）。有效状态为队列自身状态及其父队列状态的组合。INACTIVE状态的队列不能调度任何应用程序。
leaf_queue	boolean	表示队列是否在树节点或中间队列。表示叶子节点队列。

参数属性	参数类型	参数描述
number_pending_application	int	当前的挂起请求数量。如果是中间队列/父队列，这是所有子队列的集合。
number_running_application	int	当前正在运行应用的数量。如果是中间队列/父队列，这是所有子队列的集合。
number_pending_request	int	挂起命令的数量；每个未完成命令的总计数。如果是中间队列/父队列，这是所有子队列的集合。
number_running_container	int	正在运行container的数量。如果是中间队列/父队列，这是所有子队列的集合。
number_reserved_container	int	预留container的数量。如果是中间队列/父队列，这是所有子队列的集合。
resource_request	object	以vcores和内存等形式在队列中挂起的资源请求。
resource_inuse	object	以vcores和内存等形式在队列中使用的资源。
resource_reserved	object	以vcores和内存等形式预留在队列中的资源。
configuration	object	队列配置目标。
max_pending_application	int	最大挂起应用数。如果是中间队列/父队列，这是所有子队列的集合。
max_running_application	int	最大运行应用数。如果是中间队列/父队列，这是所有子队列的集合。
allocation_order_policy	String	分配策略，可以使用FIFO原则，PRIORITY原则或者FAIR原则。
max_running_application_per_user	int	每个使用者运行应用的最大数量。
max_master_share	string	该队列共享的百分比。
max_allocation_unit	object	单个container允许的最大资源，该资源以vcores和内存等形式存在。

参数属性	参数类型	参数描述
default_resource_select	String	缺省资源选择表达式。它被使用在当应用没有被指定一个提交区间值时。
user_acl	array	队列中被给予user权限的使用者。
admin_acl	array	该队列中被给予admin权限的使用者。
group	String	用户组名称。
user	String	用户名称。

- **查询Resource Pool**
  - 查询scheduler engine中所有resource pool。
    - URL
      GET https://<SS_REST_SERVER>/ws/v1/sscheduler/resourcepools/list
    - 输入
      无
    - 输出

```
JSON Response:
{
  "resourcepool_list": [
    {
      "name": "pool1",
      "description": "resource pool for crc",
      "number_member": 5,
      "members": [
        {
          "resource": "node1"
        },
        {
          "resource": "node2"
        },
        {
          "resource": "node3"
        },
        {
          "resource": "node4"
        },
        {
          "resource": "node5"
        }
      ],
      "available_resource": {
        "vcores": 60,
        "memory": 60000
      },
      "total_resource": {
        "vcores": 100,
        "memory": 128000
      },
      "configuration": {
        "resources": [
          {
            "resource": "node1"
          },
          {
```



```
          "resource": "node[2-5]"
        }
      ],
      "resource_select": "label1"
    }
  },
  {
    "name": "pool2",
    "description": "resource pool for erc",
    "number_member": 4
    "members": [
      {
        "resource": "node6"
      },
      {
        "resource": "node7"
      },
      {
        "resource": "node8"
      },
      {
        "resource": "node9"
      }
    ],
    "available_resource": {
      "vcores": 56,
      "memory": 48000
    },
    "total_resource": {
       "vcores": 100,
       "memory": 128000
    },
    "configuration": {
      "resources": [
        {
          "resource": "node6"
        },
        {
          "resource": "node[7-9]"
        }
      ],
      "resource_select": "label1"
    }
  },
  {
    "name": "default",
    "description": "system-generated",
    "number_member": 1,
    "members": [
      {
        "resource": "node0"
      }
    ],
    "available_resource": {
      "vcores": 8,
      "memory": 8192
    },
    "total_resource": {
      "vcores": 16,
      "memory": 12800
    }
  }
 ]
}
```


表 32-11 all resource pools 参数

参数属性	参数类型	参数描述
resourcepool_list	array	resource pool对象数组。
name	String	resource pool名称。
number_member	int	resource pool成员数量。
description	String	resource pool描述。
members	array	当前resource pool成员的资源名称数组。
resource	String	资源名称。
available_resource	object	该resource pool中当前可使用的资源。
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中可用资源的属性，该属性的值以数字表示。
total_resource	object	该resource pool所有资源。
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中总资源的属性，该属性的值以数字表示。
configuration	object	配置目标。
resources	array	所配置的资源名称pattern数组。
resource	String	资源名称模式。
resource_select	String	资源选择表达式。

- 查询scheduler engine中单个resource pool
  - URL

    GET https://<SS_REST_SERVER>/ws/v1/sscheduler/resourcepools/*{resourcepoolname}*
  - 输入

    无
  - 输出

    ```
    JSON Response:
    {
      "resourcepool": {
        "name": "pool1",
        "description": "resource pool for crc",
        "number_member": 5
        "members": [
          {
            "resource": "node1"
          },
          {
    ```

```
      "resource": "node2"
    },
    {
      "resource": "node3"
    },
    {
      "resource": "node4"
    },
    {
      "resource": "node5"
    }
  ],
  "available_resource": {
    "vcores": 60,
    "memory": 60000
},
"total_resource": {
  "vcores": 100,
  "memory": 128000
},
   "configuration": {
    "resources": [
      {
        "resource": "node6"
      },
      {
        "resource": "node[7-9]"
      }
    ],
    "resource_select": "label1"
  }
 }
}
```


- 异常

  未找到resource pool。

表 32-12 single resource pool 参数

参数属性	参数类型	参数描述
resourcepool	object	resource pool对象。
name	String	resource pool名称。
description	String	resource pool描述。
number_member	int	resource pool成员数量。
members	array	该resource pool现任成员的资源名称数组。
resource	String	资源名称。
available_resource	object	该resource pool当前可用资源。
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中可用资源的属性，该属性的值以数字表示。
total_resource	object	该resource pool中所有资源。

参数属性	参数类型	参数描述
vcores, memory, ..	int	可消耗数值型资源属性，当前resource pool中总资源的属性，该属性的值以数字表示。
configuration	object	配置目标。
resources	array	所配置的资源名称pattern数组。
resource	String	资源名称模式。
resource_select	String	资源选择表达式。

- **查询policiesxmlconf**
  - URL

    GET http://<SS_REST_SERVER>/ws/v1/sscheduler/policiesxmlconf/list
  - 输入

    无
  - 输出

```
<policies>
  <policlist>
    <resourcepool>default</resourcepool>
    <queues>
      <name>default</name>
      <fullname>root.default</fullname>
      <share>20.0</share>
      <reserve>memory 0,vcores 0 : 0.0%</reserve>
      <minimum>memory 0,vcores 0 : 20.0%</minimum>
      <maximum>memory 0,vcores 0 : 100.0%</maximum>
      <defaultuser>
        <maximum>100.0%</maximum>
        <weight>1.0</weight>
      </defaultuser>
    </queues>
  </policlist>
</policies>
```

