MapReduce 服务

开发指南

文档版本	05
发布日期	2021-07-27

华为技术有限公司

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

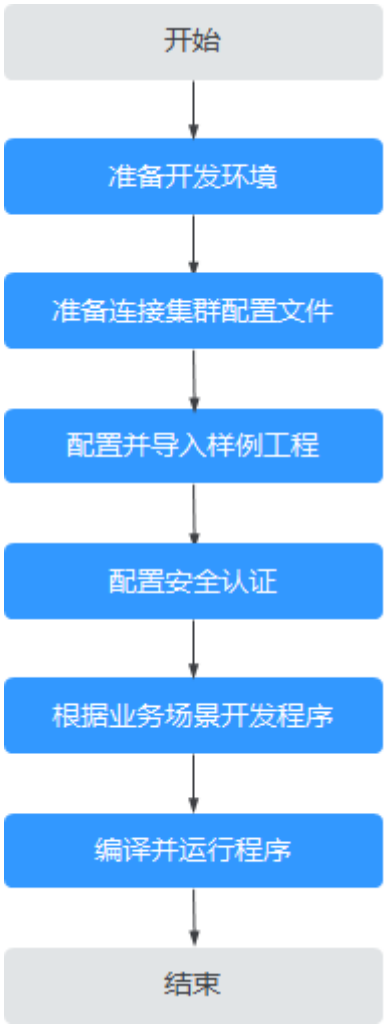


表 1-1 MRS 应用开发流程说明

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

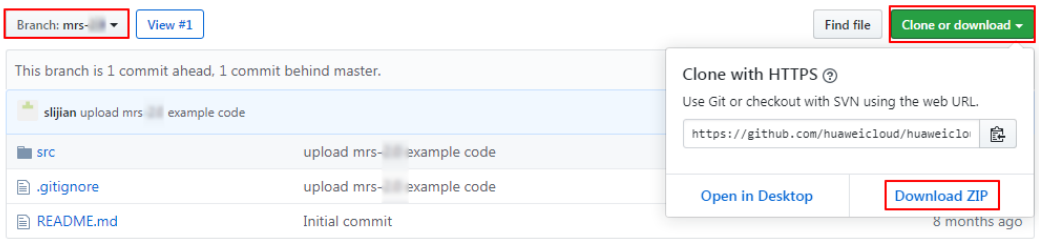
MRS服务样例工程构建流程包括三个主要步骤：

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. 配置华为云镜像站中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

- 华为云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**。
- 华为云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**。
- 华为云MRS服务2.1.x版本的样例工程下载地址为：**https://github.com/huaweicloud/huaweicloud-mrs-example/tree/mrs-2.1**。

图 2-1 样例代码下载



配置华为开源镜像仓

华为云提供开源镜像站，MRS服务样例工程依赖的jar包都需要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。

华为云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 访问并登录华为开源镜像站。

步骤3 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。

若无法直接下载，在华为开源镜像站搜索并单击名称为“HuaweiCloud SDK”的版块，按照页面弹出的设置方法进行操作。

步骤4 参考以下方法手动修改“setting.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**：

  手动在setting.xml配置文件的mirrors节点中添加开源镜像仓地址：

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

  在setting.xml配置文件的profiles节点中添加如下镜像仓地址：

  ```
  <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setting.xml配置文件的activeProfiles节点中添加如下镜像仓地址：

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

  说明

  华为云开源镜像站不提供第三方开源jar包下载，请配置华为云开源镜像后，额外配置第三方Maven镜像仓库地址。

- **配置方法二**：

  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>Mavn Centreal</name>
        <url>https://repo1.maven.org/maven2/</url>
    </repository>

</repositories>
```

----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（2.x）

组件	样例工程位置	描述
Alluxio	alluxio-examples	使用Alluxio通过公共接口连接到存储系统示例程序。可实现写文件、读文件等功能。

组件	样例工程位置	描述
Flink	flink-examples	该样例工程提供以下样例程序： ● DataStream程序 Flink构造DataStream的Java/Scala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。 ● 向Kafka生产并消费数据程序 Flink向Kafka生产并消费数据的Java/Scala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。 ● 异步Checkpoint机制程序 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。 ● Stream SQL Join程序 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。

组件	样例工程位置	描述
HDFS	hdfs-examples	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。
Hive	hive-examples	该样例工程提供以下JDBC/HCatalog样例程序： ● 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。 ● 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。
Impala	impala-examples	Impala JDBC处理数据Java示例程序。 本工程使用JDBC接口连接Impala，在Impala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。
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。

组件	样例工程位置		描述
MapReduce	mapreduce-examples		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。
Presto	presto-examples		该样例工程提供以下JDBC/HCatalog样例程序： ● Presto JDBC处理数据Java示例程序。 本工程使用JDBC接口连接Presto，在Presto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。 ● Presto HCatalog处理数据Java示例程序。
OpenTSDB	opentsdb-examples		通过调用OpenTSDB接口可实现采集大规模集群中的监控类信息，并可实现数据的秒级查询。该样例程序主要实现写入数据、查询数据、删除数据等功能。
Spark	spark-examples	SparkHbasetoHbaseJavaExample	Spark从HBase读取数据再写入HBase的Java/Scala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。
		SparkHbasetoHbaseScalaExample	
		SparkHivetoHbaseJavaExample	Spark从Hive读取数据再写入到HBase的Java/Scala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。
		SparkHivetoHbaseScalaExample	
		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。
		SparkPythonExample	
		SparkScalaExample	

组件	样例工程位置		描述
		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。
		SparkOnHbaseScalaExample	
		SparkSQLJavaExample	Spark SQL任务的Java/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。
		SparkSQLScalaExample	
		SparkStreamingJavaExample	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。
		SparkStreamingScalaExample	
		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	
		SparkStreamingtoHbaseJavaExample	Spark Streaming读取Kafka数据并写入HBase的Java/Scala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。
		SparkStreamingtoHbaseScalaExample	
		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。
		SparkStructuredStreamingScalaExample	

组件	样例工程位置		描述
		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	
Storm	storm-examples	storm-common-examples	构造Storm拓扑和开发Spout/Bolt样例程序。可实现创建Spout、创建Bolt、创建Topology等功能。
		storm-hbase-examples	MRS的Storm与HBase组件实现交互的示例程序。实现提交Storm拓扑将数据存储到HBase的WordCount表中。
		storm-hdfs-examples	MRS的Storm与HDFS组件实现交互的示例程序。实现提交Storm拓扑数据存储到HDFS的功能。
		storm-jdbc-examples	使用JDBC访问MRS Storm的样例程序。实现使用Storm拓扑向表中插入数据功能。
		storm-kafka-examples	MRS的Storm与Kafka组件实现交互的示例程序。实现使用Storm拓扑向Kafka中发送数据并查看。
		storm-obs-examples	MRS的Storm与OBS实现交互的示例程序。实现提交Storm拓扑数据存储到OBS功能。

4 HBase 应用开发

4.1 概述

4.1.1 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设计目标是用来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，请参见：**http://hbase.apache.org/apidocs/index.html**。

HBase通过接口调用，可提供的功能如表4-1所示。

表 4-1 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、协处理器
管理功能	表管理、集群管理

4.1.2 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。

- **Client**

  客户端直接面向用户，可通过Java API或HBase Shell访问服务端，对HBase的表进行读写操作。本文中的HBase客户端特指从装有HBase服务的MRS Manager上下载的HBase client安装包，里面包含通过Java API访问HBase的样例代码。

4.1.3 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4-1和表4-2所示。

图 4-1 HBase 应用程序开发流程

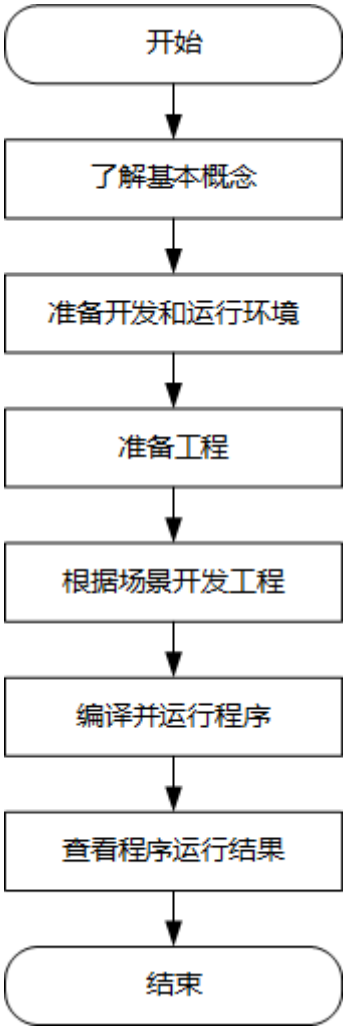


表 4-2 HBase 应用开发的流程说明

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HBase的基本概念，了解场景需求，设计表等。	常用概念
准备开发环境和运行环境	HBase的应用程序当前推荐使用Java语言进行开发。可使用Eclipse工具。HBase的运行环境即HBase客户端，请根据指导完成客户端的安装和配置。	开发和运行环境简介

阶段	说明	参考文档
准备工程	HBase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HBase工程。	配置并导入样例工程
根据场景开发工程	提供了Java语言的样例工程，包含从建表、写入到删除表全流程的样例工程。	典型场景开发思路
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。	调测程序
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。	查看调测结果

4.2 环境准备

4.2.1 开发和运行环境简介

在进行二次开发时，要准备的开发环境如表4-3所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行正常。

表 4-3 开发环境

准备项	说明
操作系统	Windows系统，推荐Windows 7及以上版本。
安装JDK	开发环境的基本配置。版本要求：1.8及以上。
安装和配置Eclipse	用于开发HBase应用程序的工具。
安装Maven	用于编译样例工程。
网络	确保客户端与HBase服务主机在网络上互通。

- 选择Windows开发环境下，安装Eclipse，安装JDK。

请安装JDK1.8及以上版本。Eclipse使用支持JDK1.8及以上的版本，并安装JUnit插件。

说明

- 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
- 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
- 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。

- 准备一个应用程序运行测试的Linux环境。

准备运行调测环境

步骤1 在弹性云服务器管理控制台，申请一个新的弹性云服务器，用于应用开发、运行、调测。

- 弹性云服务器的安全组需要和MRS集群Master节点的安全组相同。
- 弹性云服务器的VPC需要与MRS集群在同一个VPC中。
- 弹性云服务器的网卡需要与MRS集群在同一个网段中。

步骤2 申请弹性IP，绑定新申请的ECS的IP，并配置安全组出入规则。

步骤3 下载客户端程序，请参考**下载MRS客户端**。

步骤4 登录存放下载的客户端的节点，再安装客户端。

1. 执行以下命令解压客户端包：

   cd /opt

   tar -xvf /opt/MRS_Services_Client.tar

2. 执行以下命令校验安装文件包：

   sha256sum -c /opt/MRS_Services_ClientConfig.tar.sha256

   ```
   MRS_Services_ClientConfig.tar:OK
   ```

3. 执行以下命令解压安装文件包：

   tar -xvf /opt/MRS_Services_ClientConfig.tar

4. 执行如下命令安装客户端到指定目录（绝对路径），例如“/opt/client”。目录会自动创建。

   cd /opt/MRS_Services_ClientConfig

   sh install.sh /opt/client

   ```
   Components client installation is complete.
   ```

----结束

4.2.2 准备开发用户

开发用户用于运行样例工程。用户需要有HBase权限，才能运行HBase样例工程。

前提条件

MRS服务集群开启了Kerberos认证，没有开启Kerberos认证的集群忽略该步骤。

操作步骤

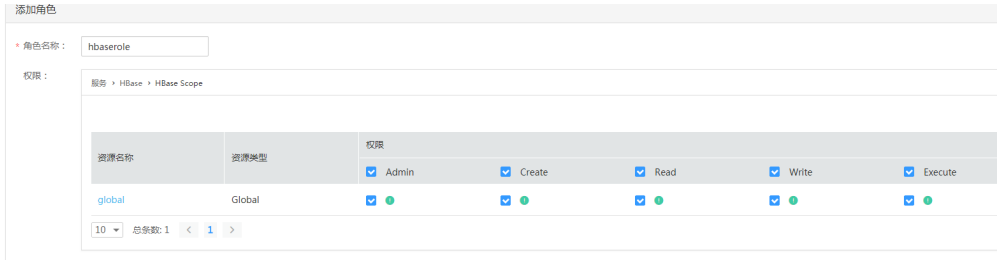
步骤1 登录MRS Manager，请参考**登录MRS Manager**。

步骤2 在MRS Manager界面选择“系统设置 > 角色管理 > 添加角色”，如**图 1 添加角色**所示。

图 4-2 添加角色

1. 填写角色的名称，例如*hbaserole*。
2. 编辑角色，在“权限”的表格中选择“HBase> HBase Scope”，勾选“Admin”、“Create”、“Read”、“Write”和“Execute”，单击“确定”保存，如图4-3所示。

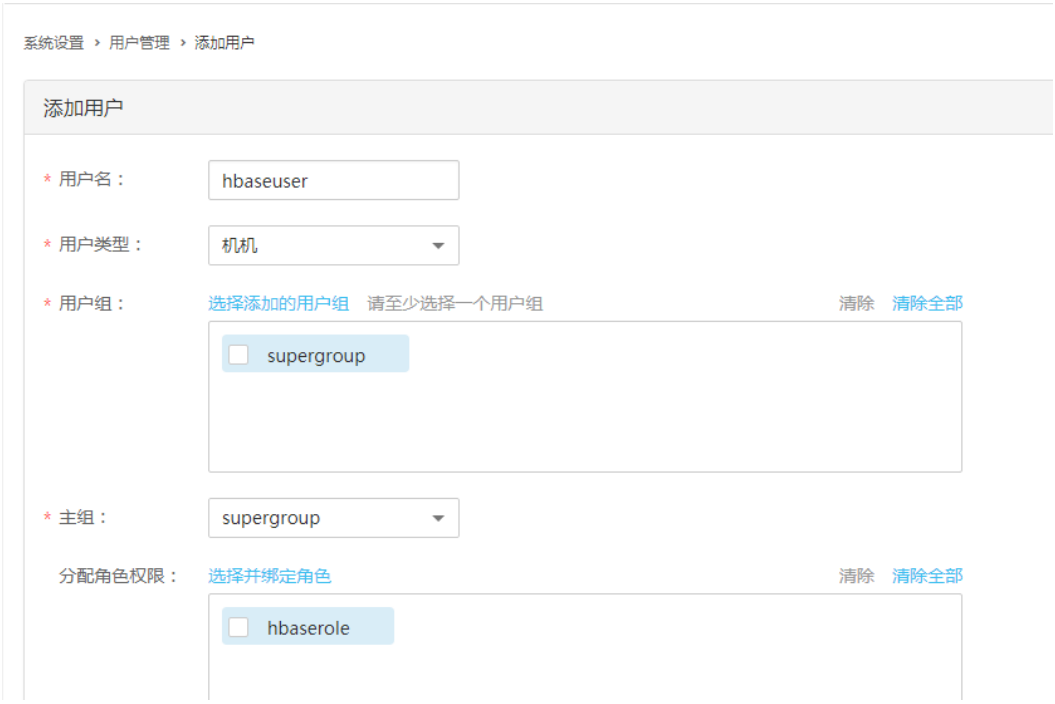
图 4-3 编辑角色



步骤3 单击“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。

步骤4 填写用户名，例如*hbaseuser*，用户类型为“机机”用户，加入用户组**supergroup**，设置其“主组”为**supergroup**，并绑定角色**hbaserole**取得权限，单击“确定”，如图4-4所示。

图 4-4 添加用户



步骤5 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名中选择**hbaseuser**，然后在右侧“操作”列中选择“更多 >下载认证凭据”，保存后解压得到用户的user.keytab文件与krb5.conf文件，用于在样例工程中进行安全认证，如图4-5所示。

图 4-5 下载认证凭据

----结束

参考信息

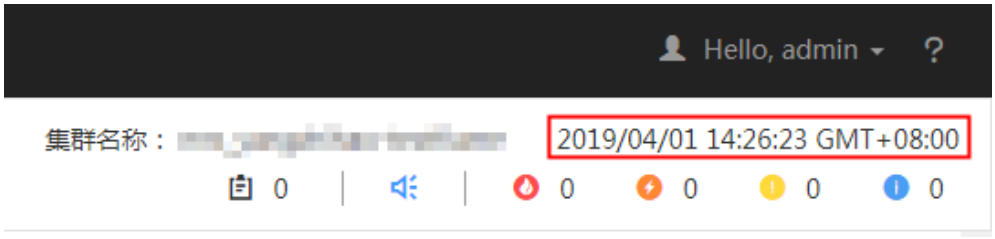
如果修改了组件的配置参数，需重新下载客户端配置文件并更新运行调测环境上的客户端。

4.2.3 配置并导入样例工程

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。MRS集群的时间可通过MRS Manager页面右上角查看。

图 4-6 MRS 集群的时间



操作步骤

步骤1 在**样例工程获取地址** 获取HBase示例工程。

步骤2 在HBase示例工程根目录，即HBase样例工程的“pom.xml”层目录下，打开cmd命令行窗口，执行mvn install编译。

步骤3 在**步骤2**中打开的cmd命令行窗口中，执行mvn eclipse:eclipse创建Eclipse工程。

步骤4 设置Eclipse开发环境。

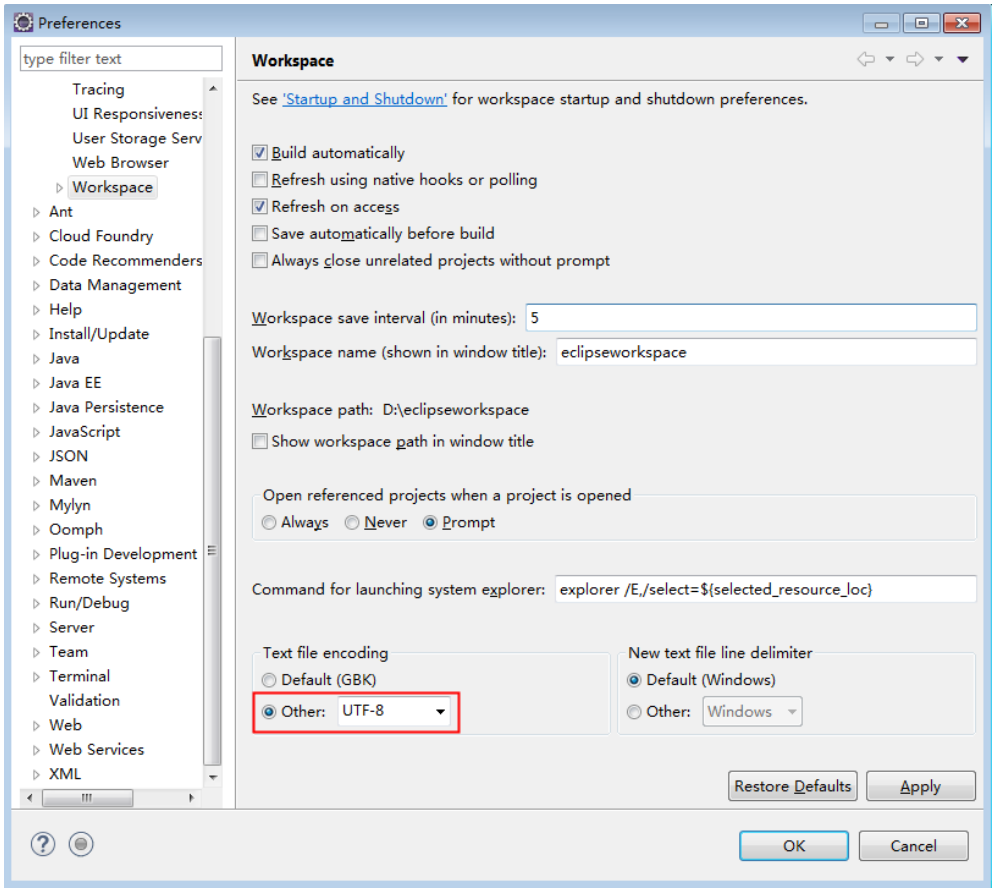
1.在Eclipse的菜单栏中，选择“Window > Preferences”。

弹出“Preferences”窗口。

2.在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”，

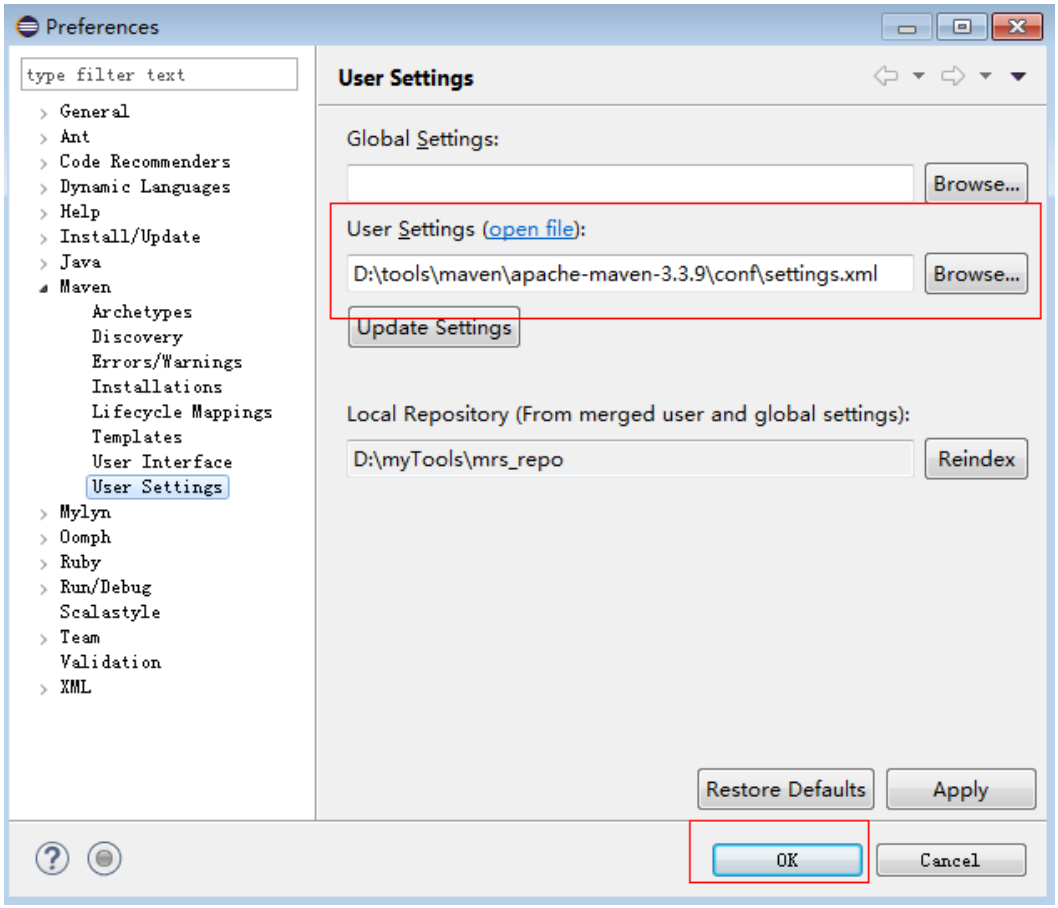
如**图 2 设置Eclipse的编码格式**所示。

图 4-7 设置 Eclipse 的编码格式



3.在左边导航上选择“Maven > User Settings”，在“User Settings”中单击“Browse”导入Maven的settings.xml配置，单击“Apply”，单击“OK”，如图4-8所示。

图 4-8 设置 Eclipse 的 Maven 开发环境



步骤5 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。

   显示“浏览文件夹”对话框。

2. 选择样例工程文件夹，单击“Finish”。

----结束

4.3 开发程序

4.3.1 典型场景开发思路

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4-4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 4-4 用户信息

编号	姓名	性别	年龄	地址
12005000201	Zhang San	Male	19	Shenzhen City, Guangdong Province
12005000202	Li Wanting	Female	23	Hangzhou City, Zhejiang Province
12005000203	Wang Ming	Male	26	Ningbo City, Zhejiang Province
12005000204	Li Gang	Male	18	Xiangyang City, Hubei Province
12005000205	Zhao Enru	Female	21	Shangrao City, Jiangxi Province
12005000206	Chen Long	Male	32	Zhuzhou City, Hunan Province
12005000207	Zhou Wei	Female	29	Nanyang City, Henan Province
12005000208	Yang Yiwen	Female	30	Wenzhou City, Zhejiang Province
12005000209	Xu Bing	Male	26	Weinan City, Shaanxi Province
12005000210	Xiao Kai	Male	25	Dalian City, Liaoning Province

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4-5所示。

表 4-5 在 HBase 中开发的功能

序号	步骤	代码实现
1	根据表4-4中的信息创建表。	请参见创建表。
2	导入用户数据。	请参见插入数据。
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。	请参见修改表。
4	根据用户编号查询用户姓名和地址。	请参见使用Get读取数据。
5	根据用户姓名进行查询。	请参见使用过滤器Filter。
6	用户销户，删除用户信息表中该用户的数据。	请参见删除数据。
7	A业务结束后，删除用户信息表。	请参见删除表。

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。

4.3.2 创建 Configuration

功能介绍

HBase通过加载配置文件来获取配置项，包括用户登录信息配置项。

代码样例

下面代码片段在com.huawei.bigdata.hbase.examples包中。

调用类TestMain下的init()方法会初始化Configuration对象：

```
private static void init() throws IOException {
    // load hbase client info
    if(clientInfo == null) {
        clientInfo = new ClientInfo(CONF_DIR + HBASE_CLIENT_PROPERTIES);
        restServerInfo = clientInfo.getRestServerInfo();
    }
    // Default load from conf directory
```

```
    conf = HBaseConfiguration.create();

    conf.addResource(CONF_DIR + "core-site.xml");
    conf.addResource(CONF_DIR + "hdfs-site.xml");
    conf.addResource(CONF_DIR + "hbase-site.xml");

  }
```

4.3.3 创建 Connection

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

以下代码片段是创建Connection的示例：

```
private TableName tableName = null;
private Configuration conf = null;
private Connection conn = null;
public static final String TABLE_NAME = "hbase_sample_table";

public HBaseExample(Configuration conf) throws IOException {
  this.conf = conf;
  this.tableName = TableName.valueOf(TABLE_NAME);
  this.conn = ConnectionFactory.createConnection(conf);
}
```

说明

1. 样例代码中有很多的操作，如建表、查询、删表等，这里只列举了建表testCreateTable和删除表dropTable这2种操作。可参考对应章节样例。
2. 创建表操作所需的Admin对象是从Connection对象获取。
3. 登录代码要避免重复调用。

4.3.4 创建表

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，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“HBaseExample”类的testCreateTable方法中

```
public void testCreateTable() {
    LOG.info("Entering testCreateTable: " + tableName);

    // Set the column family name to info.
    byte [] fam = Bytes.toBytes("info");
    ColumnFamilyDescriptor familyDescriptor = ColumnFamilyDescriptorBuilder.newBuilder(fam)
            // Set data encoding methods. HBase provides DIFF,FAST_DIFF,PREFIX
            // HBase 2.0 removed `PREFIX_TREE` Data Block Encoding from column families.
            .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 SANPPY
            .setCompressionType(Compression.Algorithm.SNAPPY)
            .build();
    TableDescriptor htd =
TableDescriptorBuilder.newBuilder(tableName).setColumnFamily(familyDescriptor).build();

    Admin admin = null;
    try {
        // Instantiate an Admin object.
        admin = conn.getAdmin();
        if (!admin.tableExists(tableName)) {
            LOG.info("Creating table...");
            admin.createTable(htd);
            LOG.info(admin.getClusterMetrics());
            LOG.info(admin.listNamespaceDescriptors());
            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.");
}
```

4.3.5 删除表

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“HBaseExample”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);//注[1]

            // Delete table.
            admin.deleteTable(tableName);
        }
        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。

4.3.6 修改表

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“HBaseExample”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 = TableDescriptorBuilder.newBuilder(tableName).build();
```

```
        // Check whether the column family is specified before modification.
        if (!htd.hasColumnFamily(familyName)) {
            // Create the column descriptor.
         ColumnFamilyDescriptor cfd = ColumnFamilyDescriptorBuilder.of(familyName);
         TableDescriptor td =
TableDescriptorBuilder.newBuilder(admin.getDescriptor(tableName)).setColumnFamily(cfd).build();
            // Disable the table to get the table offline before modifying
            // the table.
            admin.disableTable(tableName);//注[1]
            // Submit a modifyTable request.
            admin.modifyTable(td);
            // 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.");
}
```

说明

注[1]modifyTable只有表被disable时，才能生效。

4.3.7 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“HBaseExample”类的testPut方法中。其中，com.huawei.bigdata.hbase.examples包可在**样例工程获取地址**中下载的MRS对应版本的样例工程中获取。

```
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 = new Put(Bytes.toBytes("012005000201"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Zhang San"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("19"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shenzhen, Guangdong"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000202"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Li Wanting"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("23"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shijiazhuang, Hebei"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000203"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Wang Ming"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("26"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Ningbo, Zhejiang"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000204"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Li Gang"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("18"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Xiangyang, Hubei"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000205"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Zhao Enru"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("21"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Shangrao, Jiangxi"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000206"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Chen Long"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("32"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Zhuzhou, Hunan"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000207"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Zhou Wei"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("29"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Nanyang, Henan"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000208"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Yang Yiwen"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Female"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("30"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Kaixian, Chongqing"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000209"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Xu Bing"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("26"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("Weinan, Shaanxi"));
puts.add(put);

put = new Put(Bytes.toBytes("012005000210"));
put.addColumn(familyName, qualifiers[0], Bytes.toBytes("Xiao Kai"));
put.addColumn(familyName, qualifiers[1], Bytes.toBytes("Male"));
put.addColumn(familyName, qualifiers[2], Bytes.toBytes("25"));
put.addColumn(familyName, qualifiers[3], Bytes.toBytes("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。

4.3.8 删除数据

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“HBaseExample”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 a 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.");
}
```

4.3.9 使用 Get 读取数据

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“HBaseExample”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 Configuration 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.");
}
```

4.3.10 使用 Scan 读取数据

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，最好指定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“HBaseExample”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 Caching size.
        scan.setCaching(1000);//注[1]

        // 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. 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。

4.3.11 使用过滤器 Filter

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“HBaseExample”类的testFilterList方法中

```
public void testFilterList() {
    LOG.info("Entering testFilterList.");

    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"));

        // Instantiate a FilterList object in which filters have "and"
        // relationship with each other.
        FilterList list = new FilterList(Operator.MUST_PASS_ALL);
        // Obtain data with age of greater than or equal to 20.
        list.addFilter(new SingleColumnValueFilter(Bytes.toBytes("info"), Bytes.toBytes("age"),
                CompareOp.GREATER_OR_EQUAL, Bytes.toBytes(new Long(20))));
        // Obtain data with age of less than or equal to 29.
        list.addFilter(new SingleColumnValueFilter(Bytes.toBytes("info"), Bytes.toBytes("age"),
                CompareOp.LESS_OR_EQUAL, Bytes.toBytes(new Long(29))));

        scan.setFilter(list);

        // 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("Filter list successfully.");
    } catch (IOException e) {
        LOG.error("Filter list 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 testFilterList.");
}
```

4.3.12 添加二级索引

功能介绍

您可以使用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中提供的方法来管理HIndexes。 该类提供了将索引添加到现有表的方法：

根据用户是否希望在添加索引操作期间构建索引数据，有两种不同的方法可将索引添加到表中：

- addIndicesWithData()
- addIndices()

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的addIndicesExample方法中：

addIndices(): 将索引添加到没有数据的表中

```
public void addIndicesExample() {
  LOG.info("Entering Adding a Hindex.");
  // Create index instance
  TableIndices tableIndices = new TableIndices();
  HIndexSpecification spec = new HIndexSpecification(indexNameToAdd);
  spec.addIndexColumn(new HColumnDescriptor("info"), "name", ValueType.STRING);
  tableIndices.addIndex(spec);
  Admin admin = null;
  HIndexAdmin iAdmin = null;
  try {
    admin = conn.getAdmin();
    iAdmin = HIndexClient.newHIndexAdmin(admin);
    // add index to the table
    iAdmin.addIndices(tableName, tableIndices);
    // Alternately, add the specified indices with data
    // iAdmin.addIndicesWithData(tableName, tableIndices);
    LOG.info("Successfully added indices to the table " + tableName);
  } catch (IOException e) {
    LOG.error("Add Indices failed for table " + tableName + "." + e);
  } finally {
    if (iAdmin != null) {
      try {
        // Close the HIndexAdmin object.
        iAdmin.close();
      } catch (IOException e) {
        LOG.error("Failed to close HIndexAdmin ", e);
      }
```



```
    }
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting Adding a Hindex.");
}
```

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的addIndicesExampleWithData方法中：

addIndicesWithData():将索引添加到具有大量预先存在数据的表中

```
public void addIndicesExampleWithData() {
  LOG.info("Entering Adding a Hindex With Data.");
  // Create index instance
  TableIndices tableIndices = new TableIndices();
  HIndexSpecification spec = new HIndexSpecification(indexNameToAdd);
  spec.addIndexColumn(new HColumnDescriptor("info"), "age", ValueType.STRING);
  tableIndices.addIndex(spec);
  Admin admin = null;
  HIndexAdmin iAdmin = null;
  try {
    admin = conn.getAdmin();
    iAdmin = HIndexClient.newHIndexAdmin(admin);
    // add index to the table
    iAdmin.addIndicesWithData(tableName, tableIndices);
    // Alternately, add the specified indices with data
    // iAdmin.addIndicesWithData(tableName, tableIndices);
    LOG.info("Successfully added indices to the table " + tableName);
  } catch (IOException e) {
    LOG.error("Add Indices failed for table " + tableName + "." + e);
  } finally {
    if (iAdmin != null) {
      try {
        // Close the HIndexAdmin object.
        iAdmin.close();
      } catch (IOException e) {
        LOG.error("Failed to close HIndexAdmin ", e);
      }
    }
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting Adding a Hindex With Data.");
}
```

4.3.13 启用/禁用二级索引

功能介绍

您可以使用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中提供的方法来管理HIndexes。 这个类提供了启用/禁用现有索引的方法。

根据用户是否想要启用/禁用表，HIndexAdmin提供以下API：

- disableIndices ()
- enableIndices ()

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的enableIndicesExample方法中

enableIndices ():启用指定的索引（索引状态将从INACTIVE变为ACTIVE状态），因此可用于扫描索引。

```
public void enableIndicesExample() {
  LOG.info("Entering Enabling a Hindex.");
  List<String> indexNameList = new ArrayList<String>();
  indexNameList.add(indexNameToAdd);
  Admin admin = null;
  HIndexAdmin iAdmin = null;
  try {
    admin = conn.getAdmin();
    iAdmin = HIndexClient.newHIndexAdmin(admin);
    // Disable the specified indices
    iAdmin.enableIndices(tableName, indexNameList);
    // Alternately, disable the specified indices
    // iAdmin.disableIndices(tableName, indexNameList)
    LOG.info("Successfully enable indices " + indexNameList + " of the table " + tableName);
  } catch (IOException e) {
    LOG.error("Failed to enable indices " + indexNameList + " of the table " + tableName);
  } finally {
    if (iAdmin != null) {
      try {
        // Close the HIndexAdmin object.
        iAdmin.close();
      } catch (IOException e) {
        LOG.error("Failed to close HIndexAdmin ", e);
      }
    }
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting Enabling a Hindex.");
}
```

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的disableIndicesExample方法中

disableIndices (): 禁用指定的索引（索引状态将从ACTIVE更改为INACTIVE状态），因此对于索引扫描将变得无法使用

```
public void disableIndicesExample() {
  LOG.info("Entering Disabling a Hindex.");
  List<String> indexNameList = new ArrayList<~>();
  indexNameList.add(indexNameToAdd);
  Admin admin = null;
  HIndexAdmin iAdmin = null;
  try {
    admin = conn.getAdmin();
    iAdmin = HIndexClient.newHIndexAdmin(admin);
    // Disable the specified indices
    iAdmin.disableIndices(tableName, indexNameList);
```

```
    // Alternately, enable the specified indices
    // iAdmin.enableIndices(tableName, indexNameList);
    LOG.info("Successfully disabled indices " + indexNameList + " of the table " + tableName);
  } catch (IOException e) {
    LOG.error("Failed to disable indices " + indexNameList + " of the table " + tableName);
  } finally {
    if (iAdmin != null) {
      try {
        // Close the HIndexAdmin object.
        iAdmin.close();
      } catch (IOException e) {
        LOG.error("Failed to close HIndexAdmin ", e);
      }
    }
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting Disabling a Hindex.");
}
```

4.3.14 查询二级索引列表

功能介绍

您可以使用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中提供的方法来管理HIndexes。 该类提供了列出表的现有索引的方法。

HIndexAdmin为给定表格列出索引提供以下API：

- listIndices(): 该API可用于列出给定表的所有索引。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的listIndicesIntable方法中

```
public void listIndicesIntable() {
  LOG.info("Entering Listing Hindex.");
  Admin admin = null;
  HIndexAdmin iAdmin = null;
  try {
    admin = conn.getAdmin();
    iAdmin = HIndexClient.newHIndexAdmin(admin);
    // Retreive the list of indices and print it
    List<Pair<HIndexSpecification, IndexState>> indicesList = iAdmin.listIndices(tableName);
    LOG.info("indicesList:" + indicesList);
    LOG.info("Successfully listed indices for table " + tableName + ".");
  } catch (IOException e) {
    LOG.error("Failed to list indices for table " + tableName + "." + e);
  } finally {
    if (iAdmin != null) {
      try {
        // Close the HIndexAdmin object.
        iAdmin.close();
      } catch (IOException e) {
        LOG.error("Failed to close HIndexAdmin ", e);
      }
    }
    if (admin != null) {
```



```
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting Listing Hindex.");
}
```

4.3.15 使用二级索引读取数据

功能介绍

在具有HIndexes的用户表中，HBase使用Filter来查询数据。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的scanDataByHIndex方法中

```
public void scanDataByHIndex() {
  LOG.info("Entering HIndex-based Query.");
  Table table = null;
  ResultScanner rScanner = null;
  try {
    table = conn.getTable(tableName);
    // Create a filter for indexed column.
    SingleColumnValueFilter filter = new SingleColumnValueFilter(Bytes.toBytes("info"),
Bytes.toBytes("age"),
        CompareOp.GREATER_OR_EQUAL, Bytes.toBytes("26"));
    filter.setFilterIfMissing(true);

    Scan scan = new Scan();
    scan.setFilter(filter);
    rScanner = table.getScanner(scan);

    // Scan the data
    LOG.info("Scan data using indices..");
    for (Result result : rScanner) {
      LOG.info("Scanned row is:");
      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("Successfully scanned data using indices for table " + tableName + ".");
  } catch (IOException e) {
    LOG.error("Failed to scan data using indices for table " + tableName + "." + e);
  } finally {
    if (rScanner != null) {
      rScanner.close();
    }
    if (table != null) {
      try {
        table.close();
      } catch (IOException e) {
        LOG.error("failed to close table, ", e);
      }
    }
  }
  LOG.info("Entering HIndex-based Query.");
}
```

4.3.16 删除二级索引

功能介绍

您可以使用org.apache.hadoop.hbase.hindex.client.HIndexAdmin中提供的方法来管理HIndexes。 该类提供了从表中删除现有索引的方法。

根据用户是否希望删除索引数据以及索引删除操作，有两种不同的API可将索引删除到表中：

- dropIndices()
- dropIndicesWithData()

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的dropIndicesExample方法中

dropIndices():从指定的表中删除指定的索引，但索引数据不会被删除。

```
public void dropIndicesExample() {
  LOG.info("Entering Deleting a Hindex.");
  List<String> indexNameList = new ArrayList<String>();
  indexNameList.add(indexNameToAdd);
  Admin admin = null;
  HIndexAdmin iAdmin = null;
  try {
    admin = conn.getAdmin();
    iAdmin = HIndexClient.newHIndexAdmin(admin);
    // Drop the specified indices without dropping index data
    iAdmin.dropIndices(tableName, indexNameList);
    // Alternately, drop the specified indices with data
    // iAdmin.dropIndicesWithData(tableName, indexNameList);
    LOG.info("Successfully dropped indices " + indexNameList + " from the table " + tableName);
  } catch (IOException e) {
    LOG.error("Failed to drop indices " + indexNameList + " from the table " + tableName);
  } finally {
    if (iAdmin != null) {
      try {
        // Close the HIndexAdmin object.
        iAdmin.close();
      } catch (IOException e) {
        LOG.error("Failed to close HIndexAdmin ", e);
      }
    }
    if (admin != null) {
      try {
        // Close the Admin object.
        admin.close();
      } catch (IOException e) {
        LOG.error("Failed to close admin ", e);
      }
    }
  }
  LOG.info("Exiting Deleting a Hindex.");
}
```

以下代码片段在com.huawei.bigdata.hbase.examples包的“HIndexExample”类的dropIndicesExampleWithData方法中

dropIndicesWithData():从指定的表中删除指定的索引，并从用户表中删除与这些索引对应的所有索引数据。


```
public void dropIndicesExampleWithData() {
 LOG.info("Entering Deleting a Hindex With Data.");
 List<String> indexNameList = new ArrayList<String>();
 indexNameList.add(indexNameToAdd);
 Admin admin = null;
 HIndexAdmin iAdmin = null;
 try {
   admin = conn.getAdmin();
   iAdmin = HIndexClient.newHIndexAdmin(admin);
   // Drop the specified indices without dropping index data
   iAdmin.dropIndicesWithData(tableName, indexNameList);
   // Alternately, drop the specified indices with data
   // iAdmin.dropIndicesWithData(tableName, indexNameList);
   LOG.info("Successfully dropped indices " + indexNameList + " from the table " + tableName);
 } catch (IOException e) {
   LOG.error("Failed to drop indices " + indexNameList + " from the table " + tableName);
 } finally {
   if (iAdmin != null) {
     try {
       // Close the HIndexAdmin object.
       iAdmin.close();
     } catch (IOException e) {
       LOG.error("Failed to close HIndexAdmin ", e);
     }
   }
   if (admin != null) {
     try {
       // Close the Admin object.
       admin.close();
     } catch (IOException e) {
       LOG.error("Failed to close admin ", e);
     }
   }
 }
 LOG.info("Exiting Deleting a Hindex With Data.");
}
```

4.3.17 Region 的多点分割

功能简介

一般通过org.apache.hadoop.hbase.client.HBaseAdmin进行多点分割。

说明

分割操作只对空Region起作用。

可在创建表时对表进行预分区，或者对某些region直接进行split操作来替代。

本例使用multiSplit进行多点分割将HBase表按照“A~D”、“D~F”、“F~H”、“H~Z”分为四个Region。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseExample”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 (HRegionInfo regionInfo : regionSet) {
            ((HBaseAdmin) admin).multiSplit(regionInfo.getRegionName(), sk);
        }
        LOG.info("MultiSplit successfully.");
    } catch (Exception e) {
        LOG.error("MultiSplit failed ", e);
    } 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.");
}
```

4.3.18 ACL 安全配置

功能简介

访问权限控制，在关系型数据库中是一个已经很成熟的技术，HBase实现了一个较为简单的特性。这些特性归纳为读（R）、写（W）、创建（C）、执行（X）和管理（A）等。在普通模式下，该功能只有在开启HBase权限管理时才支持。

ACL的方法定义在工具类org.apache.hadoop.hbase.security.access.AccessControlClient中。

代码样例

以下代码片段在com.huawei.bigdata.hbase.examples包的“HBaseExample”类的grantACL方法中

```
public void grantACL() {
    LOG.info("Entering grantACL.");

    String user = "usertest";
    String permissions = "RW";

    String familyName = "info";
    String qualifierName = "name";

    Table mt = null;
```

```
    Admin hAdmin = null;
    try {
        // Create ACL Instance
        mt = conn.getTable(AccessControlLists.ACL_TABLE_NAME);

        Permission perm = new Permission(Bytes.toBytes(permissions));

        hAdmin = conn.getAdmin();
        HTableDescriptor ht = hAdmin.getTableDescriptor(tableName);

        // Judge whether the table exists
        if (hAdmin.tableExists(mt.getName())) {
            // Judge whether ColumnFamily exists
            if (ht.hasFamily(Bytes.toBytes(familyName))) {
                // grant permission
                AccessControlClient.grant(conn, tableName, user, Bytes.toBytes(familyName),
                        (qualifierName == null ? null : Bytes.toBytes(qualifierName)), perm.getActions());
            } else {
                // grant permission
                AccessControlClient.grant(conn, tableName, user, null, null, perm.getActions());
            }
        }
        LOG.info("Grant ACL successfully.");
    } catch (Throwable e) {
        LOG.error("Grant ACL failed ", e);
    } finally {
        if (mt != null) {
            try {
                // Close
                mt.close();
            } catch (IOException e) {
                LOG.error("Close table failed ", e);
            }
        }

        if (hAdmin != null) {
            try {
                // Close Admin Object
                hAdmin.close();
            } catch (IOException e) {
                LOG.error("Close admin failed ", e);
            }
        }
    }
    LOG.info("Exiting grantACL.");
}
```

Shell命令方式：

```
命令行
# 赋权
grant <user> <permissions>[ <table>[ <column family>[ <column qualifier> ] ] ]

# 撤销权限
revoke <user> <permissions> [ <table> [ <column family> [ <column qualifier> ] ] ]

# 设置表所有者
alter <table> {owner => <user>}

# 显示权限列表
user_permission <table>  # displays existing permissions
```

例如：

```
grant 'user1', 'RWC'
grant 'user2', 'RW', 'tableA'
user_permission 'tableA'
```

4.4 调测程序

4.4.1 在 Windows 中调测程序

4.4.1.1 编译并运行程序

在程序代码完成开发后，您可以在Windows开发环境中运行应用。

操作步骤

步骤1 在windows下使用REST API操作HBase集群时，JDK版本需为jdk1.8.0_60及以上版本。从集群环境中获取jdk的cacerts文件，并拷贝“/opt/Bigdata/jdk/jre/lib/security/cacerts”文件到windows上的JDK环境中“C:\Program Files\Java\jdk1.8.0_60\jre\lib\security”(不使用REST API操作HBase集群可跳过此步骤)。

步骤2 在windows上配置集群的ip与主机名映射关系。登录集群后台，执行命令**cat /etc/hosts**后，把hosts文件中的ip与hostname映射关系拷贝到“C:\Windows\System32\drivers\etc\hosts”中。其中主机名请以查询结果为准。

```
192.168.0.90 node-master1BedB.089d8c43-12d5-410c-b980-c2728a305be3.com
192.168.0.129 node-ana-corezLaR.089d8c43-12d5-410c-b980-c2728a305be3.com
```

说明

使用windows访问MRS集群来操作HBase，有如下两种方式：

- 申请一台windows的ECS访问MRS集群操作hbase。安装开发环境后运行样例代码。申请ECS访问MRS集群的步骤如下：
  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”、“集群控制台地址”，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“弹性云服务器 > 快速入门 > 购买并登录Windows弹性云服务器”
- 使用本机访问MRS集群操作HBase。为MRS集群中HBase服务的所有节点绑定弹性公网IP，在本机(即Windows机器)上配置集群的IP与主机名映射关系时，把IP替换为主机名对应的弹性公网IP，并修改导入样例的krb5.conf文件中"kdc","admin_server",“kpasswd_server”，“kdc_listen”，“kadmind_listen”和“kpasswd_listen”六个参数的ip(单master的集群没有后面三个参数则不必修改)，使其对应于KrbServer中对应的弹性公网IP（由于普通集群未启用kerberos功能，可跳过此修改krb5.conf文件的步骤。），然后运行样例代码。绑定弹性公网IP步骤如下：
  1. 在虚拟私有云管理控制台，申请一个弹性IP地址，并与弹性云服务器绑定。

     具体请参见“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. 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，若集群为安全集群则需要同时将UDP的21731、21732端口，TCP的21730、21731、21732及HBase的HMaster、RegionServer实例的RPC端口和ZooKeeper服务的端口添加在安全组的入方向规则中。请参见“虚拟私有云 > 用户指南 > 安全性 > 安全组 > 添加安全组规则”。

步骤3 修改运行时环境配置。

修改样例代码中自带hbase-site.xml文件的配置，与实际环境中保持一致，配置列表如下：

- hbase.zookeeper.quorum: zookeeper实例的主机名，多个实例时以逗号分隔，必选。
- hbase.regionserver.kerberos.principal: regionserver的principal，与master的principal保持一致，仅开启kerberos功能集群需要配置该参数。
- hbase.master.kerberos.principal: hmaster的principal，与regionserver的principal保持一致，仅开启kerberos功能集群需要配置该参数。
- hadoop.security.authentication：hadoop鉴权模式，开启kerberos功能的集群需要配置该参数值为kerberos，否则不用配置。
- hbase.security.authentication：hbase鉴权模式，开启kerberos功能的集群需要配置该参数值为kerberos，否则不用配置。

说明

上述参数可通过登录任一Master节点，从hbase客户端配置(“/opt/client/HBase/hbase/conf”)中获取。以获取 "hbase.zookeeper.quorum"的值为例，可登录任一Master节点执行如下命令获取：

grep "hbase.zookeeper.quorum" /opt/client/HBase/hbase/conf/* -R -A1

```
[root@node-master1BedB ~]# grep "hbase.zookeeper.quorum" /opt/client/HBase/hbase/conf/* -R -A1
/opt/client/HBase/hbase/conf/hbase-site.xml:<name>hbase.zookeeper.quorum</name>
/opt/client/HBase/hbase/conf/hbase-site.xml-<value>node-master1bedb.089d8c43-12d5-410c-b980-c2728a305be3.com</value>
```

步骤4 修改样例代码。

1. 当前样例代码中操作HBase的接口有三种，分别是普通接口，HFS接口（MRS 1.9.x版本不再支持该接口），REST接口。调试不同API接口操作HBase时可以注释其他接口调用。这里以使用普通接口操作HBase为例，main方法中只包含如下代码段。

```
public static void main(String[] args) {
  try {
    init();
    login();
  } catch (IOException e) {
      LOG.error("Failed to login because ", e);
      return;
  }
  // getDefaultConfiguration();
  conf = HBaseConfiguration.create();
  // test hbase API
  HBaseExample oneSample;
  try {
    oneSample = new HBaseExample(conf);
    oneSample.test();
  } catch (Exception e) {
    LOG.error("Failed to test HBase because ", e);
  }
  LOG.info("-----------finish HBase-------------------");
}
```

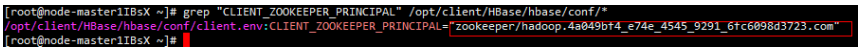
2. 修改样例工程com.huawei.bigdata.hbase.examples.TestMain类中ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL的参数值。（未开启Kerberos认证集群可跳过此步）。

```
private static final String ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/
hadoop.4a049bf4_e74e_4545_9291_6fc6098d3723.com";
```


说明

- 安全集群在Zookeeper认证时是通过四字命令获取Zookeeper服务端的principal，在Zookeeper实例高负载或者实例不稳定时，会由于无法使用四字命令获取服务端的principal导致认证失败，需要通过客户端把该值传入环境中使用，避免认证失败的问题。
- 可通过登录任一Master节点执行如下命令获取该值。

  grep "CLIENT_ZOOKEEPER_PRINCIPAL" /opt/client/HBase/hbase/conf/*

3. 修改样例工程“src/main/resources”下的hbaseclient.properties文件。userKeytab.path，krb5.conf.path对应于从**准备开发用户**获取的文件的地址。（未开启Kerberos认证集群可跳过此步）。

   ```
   user.name=hbaseuser
   userKeytabName=userKeytab.path
   krb5ConfName=krb5.conf.path
   ```

4. 若使用REST接口时需修改hbaseclient.properties中的rest.server.info，使其对应于rest server的ip:port（port默认为21309）。

   ```
   rest.server.info=10.10.10.10:21309
   ```

   说明

   1. 从集群的master节点获取core-site.xml, hdfs-site.xml, hbase-site.xml文件，并放置到样例工程的“resources”下，即“src/main/resources”，文件获取路径为“/opt/client/HBase/hbase/conf”。MRS 1.9.2及其之后版本仅需修改自带的hbase-site.xml中的相应配置即可。
   2. 若使用申请的ECS访问HBase，可直接使用RESTServer的ip；若使用本机访问HBase则需使用该RESTServer绑定的弹性公网IP作为RESTServer的ip。
   3. HIndexExample样例工程仅在MRS 1.9.2及其以后版本中支持，需注意当前集群版本。
   4. HFSSample样例工程在MRS 1.9.x版本已移除，需注意当前集群版本。

步骤5 运行样例。

在开发环境中（例如Eclipse中），右击“TestMain.java”，单击“Run as > Java Application”运行对应的应用程序工程。

----结束

4.4.1.2 查看调测结果

操作场景

HBase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HBase日志获取应用运行情况。

操作步骤

- 运行结果会有如下成功信息：

  ```
  ...
  2020-01-09 10:43:49,338 INFO  [main] examples.HBaseExample: Entering dropTable.
  2020-01-09 10:43:49,341 INFO  [main] client.HBaseAdmin: Started disable of hbase_sample_table
  2020-01-09 10:43:50,080 INFO  [main] client.HBaseAdmin: Operation: DISABLE, Table Name:
  default:hbase_sample_table, procId: 41 completed
  2020-01-09 10:43:50,550 INFO  [main] client.HBaseAdmin: Operation: DELETE, Table Name:
  default:hbase_sample_table, procId: 43 completed
  2020-01-09 10:43:50,550 INFO  [main] examples.HBaseExample: Drop table successfully.
  2020-01-09 10:43:50,550 INFO  [main] examples.HBaseExample: Exiting dropTable.
  2020-01-09 10:43:50,550 INFO  [main] client.ConnectionImplementation: Closing master protocol:
  ```

```
MasterService
2020-01-09 10:43:50,556 INFO  [main] examples.TestMain: ----------finish to test HBase
API------------------
```

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，FATAL）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
```

4.4.2 在 Linux 中调测程序

4.4.2.1 安装客户端时编译并运行程序

HBase应用程序支持在安装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HBase客户端。
- Linux环境已安装JDK，版本号需要和Eclipse导出Jar包使用的JDK版本一致。
- 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 修改样例代码。

1. 当前样例代码中操作HBase的接口有三种，分别是普通接口，HFS接口（MRS 1.9.x版本不再支持该接口），REST接口。调试不同API接口操作HBase时可以注释其他接口调用。这里以使用普通接口操作HBase为例，main方法中只包含如下代码段。

```
public static void main(String[] args) {
  try {
    init();
    login();
  } catch (IOException e) {
      LOG.error("Failed to login because ", e);
      return;
  }
  // getDefaultConfiguration();
  conf = HBaseConfiguration.create();
  // test hbase API
  HBaseExample oneSample;
  try {
    oneSample = new HBaseExample(conf);
    oneSample.test();
```

```
  } catch (Exception e) {
    LOG.error("Failed to test HBase because ", e);
  }
  LOG.info("-----------finish HBase--------------------");
}
```

2. 在调用HFS接口（MRS 1.9.x版本不再支持该接口），REST接口时需要把样例工程src\main\resources下hbaseclient.properties文件拷贝到客户端(客户端目录以/opt/client为例)的HBase/hbase/conf目录下，并修改hbaseclient.properties文件。userKeytabName，krb5ConfName对应于从**步骤2**获取的文件的地址。若使用REST接口时需修改rest.server.info，使其对应于rest server的ip:port（port默认为21309）。

```
rest.server.info=10.10.10.10:21309
user.name=hbaseuser
userKeytabName=user.keytab
krb5ConfName=krb5.conf
```

说明

HFSSample样例工程在MRS 1.9.x版本中已移除，需注意当前集群版本。

步骤2 执行mvn package生成jar包，在工程目录target目录下获取，比如:hbase-examples-mrs-2.0.jar，将获取的包上传到/opt/client/HBase/hbase/lib目录下。

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/HBase/hbase

   说明

   “$BIGDATA_CLIENT_HOME”指的是客户端安装目录。

2. 然后执行：

   source $BIGDATA_CLIENT_HOME/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**步骤2**中生成的Jar包和从**准备开发用户**中获取的krb5.conf和user.keytab文件拷贝上传至客户端运行环境的Hbase/hbase/conf目录下，例如“/opt/client/HBase/hbase/conf”。然后在“/opt/client/HBase/hbase/conf”目录下，如果不存在则创建hbaseclient.properties文件，文件中user.name对应新建的用户hbaseuser，userKeytabName和krb5ConfName值对应从**准备开发用户**中获取的认证相关文件名称，如下（未开启Kerberos认证集群可跳过此步）：

   ```
   user.name=hbaseuser
   userKeytabName=user.keytab
   krb5ConfName=krb5.conf
   ```

4. 运行如下命令执行Jar包。

   hbase com.huawei.bigdata.hbase.examples.TestMain /opt/client/HBase/hbase/conf

   其中，*com.huawei.bigdata.hbase.examples.TestMain*为举例，具体以实际样例代码为准。

   “/opt/client/HBase/hbase/conf”对应于上述中user.keytab、krb5.conf等文件路径。

说明

针对MRS 1.9.2及以后版本请执行**hbase com.huawei.bigdata.hbase.examples.TestMain /opt/client/HBase/hbase/conf**。

----结束

4.4.2.2 未安装客户端时编译并运行程序

HBase应用程序支持在未安装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

- Linux环境已安装JDK，版本号需要和Eclipse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。

操作步骤

步骤1 按**安装客户端时编译并运行程序**中的方式修改样例。

步骤2 执行mvn package生成jar包，在工程目录target目录下获取，比如：hbase-examples-2.0.jar。

步骤3 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”。将集群中任一master节点“/opt/client/HBase/hbase/lib”目录下的jar包，以及**步骤2**中导出的Jar包，上传到当前Linux环境新建目录“/opt/test”的“lib”目录下。将集群中任一master节点“/opt/client/HBase/hbase/conf”目录下的hbase-site.xml，hdfs-site.xml，core-site.xml文件拷贝到“/opt/test”中“conf”目录下。
2. 将**准备开发用户**中获取的krb5.conf和user.keytab文件拷贝上传至“/opt/test/conf”目录中，并新建hbaseclient.properties文件，文件中user.name对应新建的用户hbaseuser，userKeytabName和krb5ConfName路径对应从**准备开发用户**中获取的认证相关文件名称（未开启Kerberos认证集群可跳过此步）。

```
user.name=hbaseuser
userKeytabName=user.keytab
krb5ConfName=krb5.conf
```

3. 在“/opt/test”根目录新建脚本“run.sh”，修改内容如下并保存：

   当前以*com.huawei.bigdata.hbase.examples.TestMain*为举例，具体以实际样例代码为准。

```
#!/bin/sh
BASEDIR=`pwd`
cd ${BASEDIR}
for file in ${BASEDIR}/lib/*.jar
do
i_cp=$i_cp:$file
echo "$file"
done
if [ -d ${BASEDIR}/lib/client-facing-thirdparty ]; then
for file in ${BASEDIR}/lib/client-facing-thirdparty/*.jar
do
i_cp=$i_cp:$file
done
fi
java -cp ${BASEDIR}/conf:${i_cp} com.huawei.bigdata.hbase.examples.TestMain
```

步骤4 切换到“/opt/test”，执行以下命令，运行Jar包。

sh run.sh

----结束

4.4.2.3 查看调测结果

HBase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HBase日志获取应用运行情况。

运行结果会有如下成功信息：

```
2018-01-17 19:44:28,068 INFO  [main] examples.HBaseExample: Entering dropTable.
2018-01-17 19:44:28,074 INFO  [main] client.HBaseAdmin: Started disable of hbase_sample_table
2018-01-17 19:44:30,310 INFO  [main] client.HBaseAdmin: Disabled hbase_sample_table
2018-01-17 19:44:31,727 INFO  [main] client.HBaseAdmin: Deleted hbase_sample_table
2018-01-17 19:44:31,727 INFO  [main] examples.HBaseExample: Drop table successfully.
2018-01-17 19:44:31,727 INFO  [main] examples.HBaseExample: Exiting dropTable.
2018-01-17 19:44:31,727 INFO  [main] client.ConnectionManager$HConnectionImplementation: Closing
master protocol: MasterService
2018-01-17 19:44:31,733 INFO  [main] client.ConnectionManager$HConnectionImplementation: Closing
zookeeper sessionid=0x13002d37b3933708
2018-01-17 19:44:31,736 INFO  [main-EventThread] zookeeper.ClientCnxn: EventThread shut down for
session: 0x13002d37b3933708
2018-01-17 19:44:31,737 INFO  [main] zookeeper.ZooKeeper: Session: 0x13002d37b3933708 closed
2018-01-17 19:44:31,750 INFO  [main] examples.TestMain: ----------finish HBase ------------------
```

4.4.3 HBase Phoenix 样例代码调测

HBase支持通过Phoenix调用JDBC接口来访问HBase服务。调测HBase Phoenix样例程序，默认集群已完成HBase对接Phoenix服务，具体对接步骤详情请参考**HBase配置Phoenix**。

在 Windows 中运行并调测程序

步骤1 搭建Windows开发环境及修改样例程序的公共配置，详情请参见**步骤1-步骤3**。

步骤2 修改样例工程。

1.修改样例工程“src/main/resources”目录下的“jaas.conf”文件，keyTab、principal分别对应用户认证凭据存放路径、用户名，如下所示。（未开启Kerberos认证集群可跳过此步骤）

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="D:\\sample_project\\src\\hbase-examples\\hbase-java-examples\\src\\main\\resources\
\user.keytab"
  principal="hbaseuser"
  useTicketCache=false
  storeKey=true
  debug=true;
};
```

2.修改样例工程“src/main/resources”下的“hbaseclient.properties”文件。user.name, userKeytabName, krb5ConfName对应于在**准备开发用户**创建的用户名，及获取的文件的名称。（未开启Kerberos认证集群可跳过此步）。

```
user.name=hbaseuser
userKeytabName=user.keytab
krb5ConfName=krb5.conf
#for phoenix
#confiugration for security cluster.
jaasConfName=jaas.conf
```

步骤3 在开发环境中（例如Eclipse中），鼠标右键单击“PhoenixExample”，然后鼠标左键单击“Run > PhoenixExample.main()”运行对应的应用程序工程。

说明

若运行报“Message stream modified (41)”的错误，这可能与JDK的版本有关系，可以尝试修改运行样例代码的JDK为8u_242以下版本或删除“krb5.conf”配置文件的“renew_lifetime = 0m”配置项。

步骤4 HBase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。

运行结果出现如下信息表示程序运行成功。

```
2020-03-13 14:54:13,369 INFO  [main] client.HBaseAdmin: Operation: CREATE, Table Name: default:TEST, procId: 60 completed
2020-03-13 14:54:14,269 INFO  [main] examples.PhoenixExample: 1
2020-03-13 14:54:14,270 INFO  [main] examples.PhoenixExample: John
2020-03-13 14:54:14,270 INFO  [main] examples.PhoenixExample: 100000
2020-03-13 14:54:14,271 INFO  [main] examples.PhoenixExample: 1980-01-01
2020-03-13 14:54:14,464 INFO  [main] client.HBaseAdmin: Started disable of TEST
2020-03-13 14:54:15,199 INFO  [main] client.HBaseAdmin: Operation: DISABLE, Table Name: default:TEST, procId: 62 completed
2020-03-13 14:54:15,521 INFO  [main] client.HBaseAdmin: Operation: DELETE, Table Name: default:TEST, procId: 64 completed
```

----结束

在 Linux 中调测 Phoenix 样例

在linux环境中调测Phoenix样例，需有与集群环境网络相通的ECS，详情请参见**开发和运行环境简介**。

步骤1 修改样例。将样例代码TestMain中“enablePhoenix”值改为“true”，开启调用Phoenix样例程序接口。

```
/**
 * Phoenix Example
 * if you would like to operate hbase by SQL, please enable it,
 * and you can refrence the url ("https://support.huaweicloud.com/devg-mrs/mrs_06_0041.html").
 * step:
 * 1.login
 * 2.operate hbase by phoenix.
 */
boolean enablePhoenix = false;
if (enablePhoenix) {
  PhoenixExample phoenixExample;
  try {
    phoenixExample = new PhoenixExample(conf);
    phoenixExample.testSQL();
  } catch (Exception e) {
    LOG.error("Failed to run Phoenix Example, because ", e);
  }
}
```

步骤2 执行mvn package生成jar包，在工程目录target目录下获取，比如:hbase-examples-mrs-2.0.jar，将获取的包上传到/opt/client/Hbase/hbase/lib目录下。

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/HBase/hbase

   说明

   “$BIGDATA_CLIENT_HOME”指的是客户端安装目录。

2. 然后执行：

   source $BIGDATA_CLIENT_HOME/bigdata_env

3. 将**Phoenix**解压后获取其中的phoenix-hbase和phoenix-core包和“htrace-core-3.1.0-incubating.jar”包拷贝到“/opt/client/HBase/hbase/lib”下。

4. 将**步骤2**中生成的Jar包和从**3.2.2-准备开发用户**中获取的krb5.conf和user.keytab文件拷贝上传至客户端运行环境的Hbase/hbase/conf目录下，例如“/opt/client/HBase/hbase/conf”。然后在“/opt/client/HBase/hbase/conf”目录下创建hbaseclient.properties文件，文件中user.name对应新建的用户hbaseuser，userKeytabName和krb5ConfName值对应从**3.2.2-准备开发用户**中获取的认证相关文件名称，如下（未开启Kerberos认证集群可跳过此步）：

   ```
   user.name=hbaseuser
   userKeytabName=user.keytab
   krb5ConfName=krb5.conf
   ```

步骤4 执行jar包程序。

hbase com.huawei.bigdata.hbase.examples.TestMain /opt/client/HBase/hbase/conf

其中，*com.huawei.bigdata.hbase.examples.TestMain*为举例，具体以实际样例代码为准。

“/opt/client/HBase/hbase/conf”对应于上述中user.keytab、krb5.conf等文件路径。

说明

若运行报“Message stream modified (41)”的错误，这可能与JDK的版本有关系，可以尝试修改运行样例代码的JDK为8u_242以下版本或删除“krb5.conf”配置文件的“renew_lifetime = 0m”配置项。

步骤5 phoenix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。

```
2020-03-14 16:20:40,192 INFO  [main] client.HBaseAdmin: Operation: CREATE, Table Name: default:TEST, procId: 923 completed
2020-03-14 16:20:40,806 INFO  [main] examples.PhoenixExample: 1
2020-03-14 16:20:40,807 INFO  [main] examples.PhoenixExample: John
2020-03-14 16:20:40,807 INFO  [main] examples.PhoenixExample: 100000
2020-03-14 16:20:40,807 INFO  [main] examples.PhoenixExample: 1980-01-01
2020-03-14 16:20:40,830 INFO  [main] client.HBaseAdmin: Started disable of TEST
2020-03-14 16:20:41,574 INFO  [main] client.HBaseAdmin: Operation: DISABLE, Table Name: default:TEST, procId: 925 completed
2020-03-14 16:20:41,831 INFO  [main] client.HBaseAdmin: Operation: DELETE, Table Name: default:TEST, procId: 927 completed
```

----结束

4.4.4 HBase python 样例代码调测

仅MRS 1.9.x及之前版本支持HBase python样例代码调测。

HBase支持使用自带的ThriftServer2服务通过python来访问HBase服务。python样例仅支持在Linux环境中运行，调测HBase python样例程序需有与集群环境网络相通的ECS，详情请参见**开发和运行环境简介**，并需要安装python环境，安装包下载详情请参见：**https://www.python.org/**。当前以在集群的master节点上运行样例为例。

步骤1 搭建样例运行环境。

获取运行样例程序时python依赖，请从**https://pypi.org/**地址中搜索下载decorator、gssapi、kerberos、krbcontext、pure-sasl、thrift包（未开启Kerberos认证的普通集群仅需安装thrift包），并上传到master节点上，例如：新建目录“/opt/hbase-examples/python”,并上传到该目录下。

```
decorator-4.3.2.tar.gz
gssapi-1.5.1.tar.gz
kerberos-1.3.0.tar.gz
krbcontext-0.8.tar.gz
pure-sasl-0.6.1.tar.gz
thrift-0.11.0.tar.gz
```

步骤2 将样例工程中的“hbase-python-example”文件夹上传到集群Master节点的“/opt/hbase-examples”下，并上传从**准备开发用户**中获取的认证文件至该目录下。

步骤3 在“/opt/hbase-examples”新建hbasepython.properties文件，并修改配置内容如下。

```
clientHome=/opt/client
exampleCodeDir=/opt/hbase-examples/hbase-python-example
pythonLib=/opt/hbase-examples/python
keyTabFile=/opt/hbase-examples/user.keytab
userName=hbaseuser
thriftIp=xxx.xxx.xx.xxx
```

说明

- clientHome为集群客户端路径
- exampleCodeDir为hbase-python-example文件路径
- pythonLib为**步骤1**中python依赖存放路径
- keyTabFile为从**准备开发用户**获取的用户认证凭据user.keytab
- userName为**准备开发用户**中开发用户名
- thriftIp为安装了thriftserver2的节点的IP地址

步骤4 执行如下命令创建表名为example的HBase表。

source /opt/client/bigdata_env

kinit *用户名*

echo "create 'example','family1','family2'" | hbase shell

步骤5 安装python环境并运行程序。

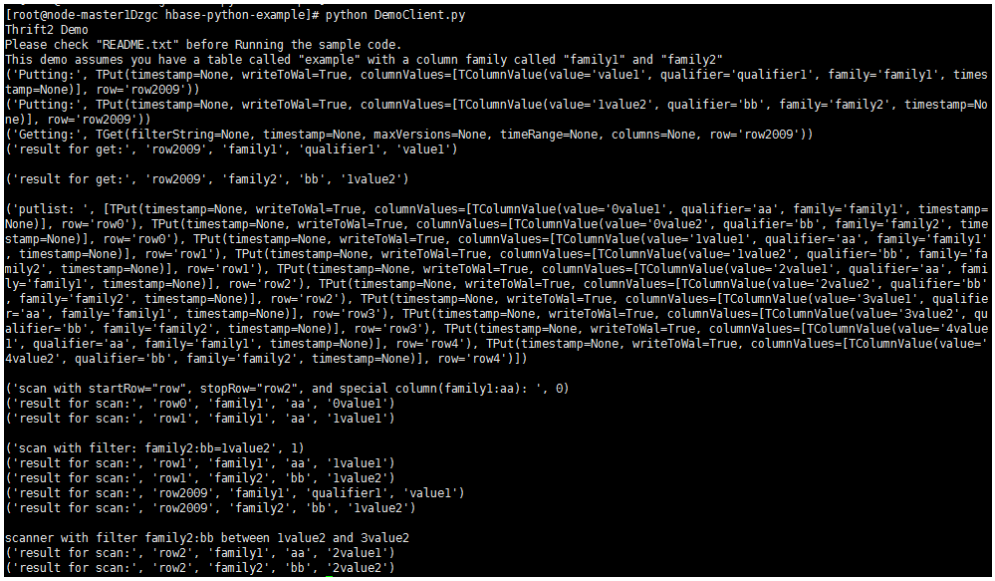
```
cd /opt/hbase-examples/hbase-python-example
sh initEnvAndRunDemo.sh /opt/hbase-examples/hbasepython.properties
```

说明

- 在运行程序之前需要格式化initEnvAndRunDemo.sh脚本。例如执行，dos2unix /opt/hbase-examples/hbasepython.properties
- 在执行脚本前，请确定“example”表包含列族'family1','family2'并已经存在于集群中。
- 再次运行程序时只需进入“/opt/hbase-examples/hbase-python-example”目录下，执行如下命令执行程序调测样例：python DemoClient.py

步骤6 HBase Python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。

图 4-9 程序运行成功信息



----结束

4.5 更多信息

4.5.1 SQL 查询

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);
            // 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。
- MRS 1.9.2之后的版本需要按**Phoenix**中下载和配置开源的phoenix包。

4.5.2 配置 HBase 文件存储

使用条件

集群版本小于MRS 3.x。

使用场景

HBase文件存储模块（HBase FileStream，简称HFS）是HBase的独立模块，它作为对HBase与HDFS接口的封装，应用在MRS的上层应用，为上层应用提供文件的存储、读取、删除等功能。

在Hadoop生态系统中，无论是HDFS，还是HBase，在面对海量文件存储的时候，在某些场景下，都会存在一些很难解决的问题：

- 如果把海量小文件直接保存在HDFS中，会给NameNode带来极大的压力。
- 由于HBase接口以及内部机制的原因，一些较大的文件也不适合直接保存到HBase中。

HFS的出现，就是为了解决需要在Hadoop中存储海量小文件，同时也要存储一些大文件的混合场景。简单来说，就是在HBase表中，需要存放大量的小文件（10MB以下），同时又需要存放一些比较大的文件（10MB以上）。

HFS为以上场景提供了统一的操作接口，这些操作接口与HBase的函数接口类似。必须在HBase的配置参数“hbase.coprocessor.master.classes”中增加一个值：“org.apache.hadoop.hbase.filestream.coprocessor.FileStreamMasterObserver”。

须知

- 如果只有小文件，确定不会有大文件的场景下，建议使用HBase的原始接口进行操作。
- HFS接口需要同时对HBase和HDFS进行操作，所以客户端用户需要同时拥有这两个组件的操作权限。
- 直接存放在HDFS中的大文件，HFS在存储时会加入一些元数据信息，所以存储的文件不是直接等于原文件的。不能直接从HDFS中移动出来使用，而需要用HFS的接口进行读取。
- 使用HFS接口存储在HDFS中的数据，暂不支持备份与容灾。

操作步骤

步骤1 登录**MRS Manager**。

步骤2 单击“服务管理 > HBase > 服务配置”，“参数类别”类型选择“全部配置”，然后在左边窗口选择“HMaster > 系统”。

步骤3 在“hbase.coprocessor.master.classes”配置项中增加值“org.apache.hadoop.hbase.filestream.coprocessor.FileStreamMasterObserver”。

步骤4 单击“保存配置”，在弹出窗口中勾选“重新启动受影响的服务或实例。”选项，然后单击“是”，重启HBase服务。

----结束

4.5.3 HFS 的 JAVA API

使用条件

集群版本小于MRS 3.x。

接口介绍

主要类说明：

接口org.apache.hadoop.hbase.filestream.client.FSTableInterface常用接口说明：

方法	说明
void put(FSPut fsPut)	向HFS表中插入数据
void put(List<FSPut> fsPuts)	向HFS表中批量插入数据
FSResult get(FSGet fsGet)	从HFS表中读取数据
FSResult[] get(List<FSGet> fsGets)	从HFS表中读取多行数据
void delete(FSDelete fsDelete)	从HFS表中删除数据
void delete(List<FSDelete> fsDeletes)	从HFS表中删除多行数据
void close()	关闭表对象

org.apache.hadoop.hbase.filestream.client.FSTable是org.apache.hadoop.hbase.filestream.client.FSTableInterface接口的实现类。

org.apache.hadoop.hbase.filestream.client.FSHColumnDescriptor继承自org.apache.hadoop.hbase.HColumnDescriptor，新增如下接口：

方法	说明
public void setFileColumn()	设置这个列族为存储文件的列族。
public void setFileThreshold(int fileThreshold)	设置存储文件大小的阈值。

org.apache.hadoop.hbase.filestream.client.FSTableDescriptor继承自org.apache.hadoop.hbase.HTableDescriptor，没有新增接口，但是如果要使用JAVA接口创建HFS表来存储文件，必须使用该类。

org.apache.hadoop.hbase.filestream.client.FSPut继承自org.apache.hadoop.hbase.Put，新增如下接口：

方法	说明
public FSPut(byte[] row)	构造函数。通过rowkey来构造对象。
public FSPut(byte[] row, long timestamp)	构造函数。通过rowkey和时间戳来构造对象。
public void addFile(String name, byte[] value)	向HFS表中的存储文件的列族中插入一个文件，以name为列名，value为文件内容。
public void addFile(String name, byte[] value, long ts)	向HFS表中的存储文件的列族中插入一个文件，以name为列名，value为文件内容，ts为指定的时间戳。
public void addFile(String name, InputStream inputStream)	向HFS表中的存储文件的列族中插入一个文件，以name为列名，inputStream为文件的输入流对象。 输入流对象需要调用者自行关闭。
public void addFile(String name, InputStream inputStream, long ts)	向HFS表中的存储文件的列族中插入一个文件，以name为列名，inputStream为文件的输入流对象，ts为指定的时间戳。 输入流对象需要调用者自行关闭。

org.apache.hadoop.hbase.filestream.client.FSGet继承自org.apache.hadoop.hbase.Get，新增如下接口

方法	说明
public FSGet(byte[] row)	构造函数。根据rowkey构造对象。
public void addFile(String fileName)	指定需要返回的文件。
public void addFiles(List<String> fileNames)	指定需要返回的多个文件。

org.apache.hadoop.hbase.filestream.client.FSResult继承自org.apache.hadoop.hbase.Result，新增如下接口：

方法	说明
public FSFile getFile(String fileName)	从查询结果中返回指定文件名的FSFile文件对象。

org.apache.hadoop.hbase.filestream.client.FSFile接口：

方法	说明
public InputStream createInputStream()	从FSFile对象中获取文件的输入流对象。

4.6 HBase 接口

4.6.1 Shell

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：

步骤1 进入HBase客户端任意目录。

步骤2 初始化环境变量。

source /opt/client/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。如果当前集群未启用Kerberos认证，则无需执行此命令。当前用户为**准备开发用户**时增加的开发用户。

人机用户：*kinit MRS集群用户*

例如：**kinit hbaseuser**

机机用户：*kinit -kt 认证凭据路径 MRS集群用户*

例如：**kinit -kt /opt/user.keytab hbaseuser**

步骤4 执行**hbase shell**命令。

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。

----结束

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：（具体以实际节点输出结果为准）

  ```
  version 1.0.2
  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：（具体以实际节点输出结果为准）

  ```
  version 1.0.2
  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：（具体以实际节点输出结果为准）

  ```
  version 1.0.2
  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
  ```

4.6.2 Java API

HBase采用的接口与Apache HBase保持一致，请参见**http://hbase.apache.org/apidocs/index.html**。

新增或修改的接口

- HBase 0.98.3建议使用org.apache.hadoop.hbase.Cell作为Key-value数据对象，而不是HBase 0.94的org.apache.hadoop.hbase.KeyValue。
- HBase 0.98.3建议使用HConnection connection = HConnectionManager.createConnection(conf)来创建连接池，废弃HTablePool。
- 新的EndPoint接口，参见**http://hbase.apache.org/book/cp.html**。
- org.apache.hadoop.hbase.client.Scan中新增反向扫描方法设置isReversed()和setReversed(boolean reversed)。

- HBase 0.98到1.0的API变更，请参考：**https://issues.apache.org/jira/browse/hbase-10602**。
- HBase 1.0 建议不要使用org.apache.hadoop.hbase.mapred，建议使用org.apache.hadoop.hbase.mapreduce。
- 版本详细的信息请参考：**https://blogs.apache.org/hbase/entry/start_of_a_new_era**。
- 获取HBase replication metrics新增的API接口

表 4-6 org.apache.hadoop.hbase.client.replication.ReplicationAdmin

方法	描述
getSourceMetricsSummary(String id)	参数类型：String 需要获取对端id的源指标汇总。 返回类型：Map<String, String> 返回：一个Map，其中键是RegionServer的名称，值是指定对端id的源集群指标的汇总。汇总指标是'sizeOfLogToReplicate'和'timeForLogToReplicate'。
getSourceMetrics(String id)	参数类型：String 需要获取对端id的源指标汇总。 返回类型：Map<String, String> 返回：一个Map，其中键是RegionServer的名称，值是指定对端id源集群的指标。
getSinkMetrics()	返回类型：Map<String, String> 返回：一个Map，其中键是RegionServer的名称，值是指定对端id源集群的sink指标。
getPeerSinkMetrics(String id)	参数类型：String 需要获取对端id的源指标汇总。 返回类型：Map<String, String> 返回：一个Map，其中键是RegionServer的名称，值是指定对端id源集群的sink指标。

说明

所有方法返回一个Map，其中键是“RegionServer名称(IP/Host)”和string类型的值，包含了所有指标，其格式是'Metric Name'='Metric Value' [, 'Metric Name'= 'Metric Value']*

举例：SizeOfHFileRefsQueue=0, AgeOfLastShippedOp=0

表 4-7 org.apache.hadoop.hbase.replication.ReplicationLoadSource

方法	描述
getPeerID()	返回类型：String 返回：对端集群的id
getAgeOfLastShippedOp()	返回类型：long 返回：上次成功的replication请求持续的毫秒数
getSizeOfLogQueue()	返回类型：long 返回：队列中等待replication的WALs
getTimeStampOfLastShippedOp()	返回类型：long 返回：上次成功的replication请求的时间戳
getReplicationLag()	返回类型：long 返回：当前时间和上次成功的replication请求的时间间隔
getShippedOps()	返回类型：long 返回：输送的数据ops总数
getShippedBytes()	返回类型：long 返回：输送的总的数据字节数
getShippedBatches()	返回类型：long 返回：输送的总的数据批数
getLogReadInBytes()	返回类型：long 返回：从wal日志读取的总字节数
getLogEditsRead()	返回类型：long 返回：从wal日志读取的总编辑数
getSizeOfLogToReplicate()	返回类型：long 返回：在队列中等待replicate的总wal日志大小
getTimeForLogToReplicate()	返回类型：long 返回：在队列中replicate wal日志需要花的秒数
getShippedHFiles()	返回类型：long 返回：输送的HFile总数
getSizeOfHFileRefsQueue()	返回类型：long 返回：等待replicate的HFile总数
getLogEditsFiltered()	返回类型：long 返回：过滤的wal编辑总数

方法	描述
getFailedReplicationAttempts()	返回类型：long 返回：在一次请求中不能复制数据的次数。

表 4-8 org.apache.hadoop.hbase.replication.ReplicationLoadSink

方法	描述
getAgeOfLastAppliedOp()	返回类型：long 返回：上次成功的应用wal编辑的持续毫秒数
getTimeStampsOfLastAppliedOp()	返回类型：long 返回：上次成功的应用wal编辑的时间戳
getAppliedBatches()	返回类型：long 返回：应用的数据总批数
getAppliedOps()	返回类型：long 返回：应用数据ops的总数
getAppliedHFiles()	返回类型：long 返回：应用的HFile总数

说明

Replication Admin新的接口，从HMaster获取指标值。每个Region Server在每一个心跳周期（默认是3秒）上报状态给HMaster。所以API通过Region Server在最后一个心跳时上报最新的指标值。

如果需要当前最新的指标值，使用由Region Server提供的JMX接口。

- **1.3.1（MRS 1.9.2）版本的接口变更**
  - 新增HIndex API

表 4-9 org.apache.hadoop.hbase.hindex.client.HIndexAdmin

方法	描述
addIndices(TableName tablename,TableIndices tableIndices)	参数：TableName 用户想要添加指定索引的表的名称。 参数：TableIndices 要添加到表中的表索引 返回类型：viod

方法	描述
addIndicesWithData(TableName tablename,TableIndices tableIndices)	参数：TableName 用户想要添加指定索引的表的名称 参数：TableIndices 要添加到表中的表索引 返回类型：void
dropIndices(TableName tableName,List <String> list)	参数：TableName 用户想要删除索引的表的名称 参数： List<String> 包含要删除的索引名称的列表 返回类型：void
dropIndicesWithData(TableName tableName,List <String> list)	参数：TableName 用户想要删除指定索引的表的名称 参数：List <String> 包含要删除的索引名称的列表 返回类型：void
disableIndices(TableName tableName,List <String> list)	参数：TableName 用户想要禁用指定索引的表的名称 参数：List <String> 包含要禁用的索引名称的列表 返回类型：void
enableIndices(TableName tableName,List <String> list)	参数：TableName 用户希望启用指定索引的表的名称 参数：List <String> 包含要启用的索引名称的列表 返回类型：void
listIndices(TableName tableName)	参数：TableName 用户想要列出所有索引的表的名称 返回类型：List <Pair <HIndexSpecification，IndexState >> 返回：返回二级索引列表，第一个元素是索引规范，第二个元素是该索引的当前状态。

4.6.3 Phoenix

版本关系

若使用Phoenix，需下载与当前使用MRS集群相对应的Phoenix版本，具体请参见**http://phoenix.apache.org**。其对应关系如**表4-10**所示：

表 4-10 MRS 与 Phoenix 版本对应关系一览表

MRS版本	Phoenix版本	备注
MRS 1.9.2	x.xx.x-HBase-1.3	例如，4.14.1-HBase-1.3

配置方式

MRS 3.x之前的版本需要去官网下载第三方的phoenix包，然后进行如下配置，MRS 3.x版本已支持Phoenix，可直接在已安装HBase客户端的节点使用Phoenix，开启了Kerberos认证的集群相关操作请参见**Phoenix命令行**，未开启Kerberos认证的集群相关操作请参见**Phoenix命令行**：

1. 从官网(**https://phoenix.apache.org/download.html**)下载phoenix二进制包上传至集群的任一Master节点，解压后修改相应权限并切换到omm用户下（例如，apache-phoenix-4.14.1-HBase-1.3-bin.tar.gz）。
   ```
   tar -xvf apache-phoenix-4.14.1-HBase-1.3-bin.tar.gz
   chown omm:wheel apache-phoenix-4.14.1-HBase-1.3-bin -R
   su - omm
   ```
2. 进入apache-phoenix-4.14.1-HBase-1.3-bin中，在该目录中编辑如下脚本，例如，脚本名称为“installPhoenixJar.sh”，则需执行命令为：“sh installPhoenixJar.sh <PHOENIX_HBASE_VERSION> <MRS_VERSION> <IPs>”(IP为HBase安装节点的IP，即所有Master和Core节点的IP，并以当前集群实际IP为准)。例如脚本如下所示：
   ```
   #!/bin/bash

   PHOENIX_HBASE_VERSION=$1
   shift
   MRS_VERSION=$1
   shift
   IPs=$1
   shift
   check_cmd_result() {
       echo "executing command: $*"
       str="$@"
       if [ ${#str} -eq 7 ]; then
           echo "please check input args, such as, sh installPhoenixJar.sh 5.0.0-HBase-2.0 2.0.1
   xx.xx.xx.xx,xx.xx.xx.xx,xx.xx.xx.xx"
           exit 1
       fi
       if ! eval $*
       then
           echo "Failed to execute: $*"
           exit 1
       fi
   }

   check_cmd_result [ -n "$PHOENIX_HBASE_VERSION" ]
   check_cmd_result [ -n "$MRS_VERSION" ]
   check_cmd_result [ -n "$IPs" ]

   if [ ${MRS_VERSION}X = "1.8.5"X ]; then
       MRS_VERSION="1.8.3"
   fi
   if [[ ${MRS_VERSION} =~ "1.6" ]]; then
       WORKDIR="FusionInsight"
   elif [[ ${MRS_VERSION} =~ "1.7" ]]; then
       WORKDIR="MRS"
   else
       WORKDIR="MRS_${MRS_VERSION}/install"
   fi
   ```

```
check_cmd_result HBASE_LIBDIR=$(ls -d /opt/Bigdata/${WORKDIR}/FusionInsight-HBase-*/hbase/lib)
# copy jars to local node.
check_cmd_result cp phoenix-${PHOENIX_HBASE_VERSION}-server.jar ${HBASE_LIBDIR}
check_cmd_result cp phoenix-core-${PHOENIX_HBASE_VERSION}.jar ${HBASE_LIBDIR}

check_cmd_result chmod 700 ${HBASE_LIBDIR}/phoenix-${PHOENIX_HBASE_VERSION}-server.jar
check_cmd_result chmod 700 ${HBASE_LIBDIR}/phoenix-core-${PHOENIX_HBASE_VERSION}.jar

check_cmd_result chown omm:wheel ${HBASE_LIBDIR}/phoenix-${PHOENIX_HBASE_VERSION}-
server.jar
check_cmd_result chown omm:wheel ${HBASE_LIBDIR}/phoenix-core-$
{PHOENIX_HBASE_VERSION}.jar

if [[ "$MRS_VERSION" =~ "2." ]]; then
    check_cmd_result rm -rf ${HBASE_LIBDIR}/htrace-core-3.1.0-incubating.jar
    check_cmd_result rm -rf /opt/client/HBase/hbase/lib/joda-time-2.1.jar
    check_cmd_result ln -s /opt/share/htrace-core-3.1.0-incubating/htrace-core-3.1.0-incubating.jar \
${HBASE_LIBDIR}/htrace-core-3.1.0-incubating.jar
    check_cmd_result ln -s /opt/share/joda-time-2.1/joda-time-2.1.jar /opt/client/HBase/hbase/lib/joda-
time-2.1.jar
fi

# copy jars to other nodes.
localIp=$(hostname -i)
ipArr=($(echo "$IPs" | sed "s|\,|\ |g"))
for ip in ${ipArr[@]}
do
    if [ "$ip"X = "$localIp"X ]; then
        echo "skip copying jar to local node."
        continue
    fi
    check_cmd_result scp ${HBASE_LIBDIR}/phoenix-${PHOENIX_HBASE_VERSION}-server.jar ${ip}:$
{HBASE_LIBDIR} 2>/dev/null
    check_cmd_result scp ${HBASE_LIBDIR}/phoenix-core-${PHOENIX_HBASE_VERSION}.jar ${ip}:$
{HBASE_LIBDIR} 2>/dev/null
    if [[ "$MRS_VERSION" =~ "2." ]]; then
        check_cmd_result ssh $ip "rm -rf ${HBASE_LIBDIR}/htrace-core-3.1.0-incubating.jar" 2>/dev/null
        check_cmd_result ssh $ip "rm -rf /opt/client/HBase/hbase/lib/joda-time-2.1.jar" 2>/dev/null
        check_cmd_result ssh $ip "ln -s /opt/share/htrace-core-3.1.0-incubating/htrace-core-3.1.0-
incubating.jar \
${HBASE_LIBDIR}/htrace-core-3.1.0-incubating.jar" 2>/dev/null
        check_cmd_result ssh $ip "ln -s /opt/share/joda-time-2.1/joda-time-2.1.jar /opt/client/HBase/
hbase/lib/joda-time-2.1.jar" 2>/dev/null
    fi
done
echo "installing phoenix jars to hbase successfully..."
```

说明

- 请使用txt文本格式复制导入如上脚本，以避免导入格式错误问题。
- <PHOENIX_HBASE_VERSION>：当前使用的phoenix版本。例如MRS 3.x之前可使用的phoenix版本为4.14.1-HBase-1.3。
- <MRS_VERSION>：当前使用的MRS版本。
- <IPs>：hbase的安装节点ip，以逗号分隔，即当前集群的Master节点和Core节点ip。
- 执行脚本后，打印出“installing phoenix jars to hbase successfully...”字样则表示phoenix已安装成功。

3. 登录**MRS Manager**界面，重启HBase服务。
4. 配置phoenix客户端参数(未开启Kerberos认证集群可跳过此步骤)。
   a. 配置phoenix连接时使用的认证信息。进入$PHOENIX_HOME/bin，编辑hbase-site.xml文件，需配置参数如**表 1 Phoenix参数配置**所示。

表 4-11 Phoenix 参数配置

参数	描述	默认值
hbase.regionserver.kerberos.principal	当前集群regionserver的principal	未设置
hbase.master.kerberos.principal	当前集群hmaster的principal	未设置
hbase.security.authentication	初始化Phoenix连接时所采用的认证方式	kerberos

可配置参数。如下所示，

```
<property>
<name>hbase.regionserver.kerberos.principal</name>
<value>hbase/hadoop.hadoop.com@HADOOP.COM</value>
</property>
<property>
<name>hbase.master.kerberos.principal</name>
<value>hbase/hadoop.hadoop.com@HADOOP.COM</value>
</property>
<property>
<name>hbase.security.authentication</name>
<value>kerberos</value>
</property>
```

说明

其中参数“hbase.master.kerberos.principal”和“hbase.regionserver.kerberos.principal”为开启Kerberos认证的安全集群中hbase的kerberos用户，可搜索客户端中hbase-site.xml文件得到参数值。例如，客户端安装在master节点的/opt/client下，则可使用命令“grep "kerberos.principal" /opt/client/HBase/hbase/conf/hbase-site.xml -A1”获取，如图4-10所示。

图 4-10 获取 hbase 的 principal

```
[root@master1 opt]# grep "kerberos.principal" /opt/client/HBase/hbase/conf/hbase-site.xml -A1
<name>hbase.regionserver.kerberos.principal</name>
<value>hbase/hadoop.hadoop.com@HADOOP.COM</value>
--
<name>hbase.master.kerberos.principal</name>
<value>hbase/hadoop.hadoop.com@HADOOP.COM</value>
--
```

b. 修改Phoenix路径的bin目录下的sqlline.py脚本（例如：apache-phoenix-4.14.1-HBase-1.3-bin/bin/sqlline.py），添加hbase客户端的相关依赖信息如图4-11所示。

图 4-11 Phoenix 依赖及 zookeeper 认证

```
106     colorSetting = "false"
107
108 java_cmd = java + ' $PHOENIX_OPTS $HBASE_OPTS' + \
109     ' -cp "$HBASE_HOME/lib/*:'+ hbase_config_path + os.pathsep + phoenix_utils.hbase_conf_dir + os.pathsep + phoenix_utils.phoenix_client_jar + \
110     os.pathsep + phoenix_utils.hadoop_common_jar + os.pathsep + phoenix_utils.hadoop_hdfs_jar + \
111     os.pathsep + phoenix_utils.hadoop_conf + os.pathsep + phoenix_utils.hadoop_classpath + '" -Dlog4j.configuration=file:' + \
112     os.path.join(phoenix_utils.current_dir, "log4j.properties") + \
113     " sqlline.SqlLine -d org.apache.phoenix.jdbc.PhoenixDriver" + \
114     " -u jdbc:phoenix:" + phoenix_utils.shell_quote([zookeeper]) + \
115     " -n none -p none --color=" + colorSetting + " --fastConnect=" + args.fastconnect + \
116     " --verbose=" + args.verbose + " --incremental=false --isolation=TRANSACTION_READ_COMMITTED " + sqlfile
```

详细配置。如下所示，

```
添加hbase client的lib包(eg，$HBASE_HOME/lib/*:)
添加相关认证（eg，$HBASE_OPTS）
```

使用方法

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，Phoenix同样支持以JDBC的方式来操作HBase，具体请参见**SQL查询**。

1. 连接Phoenix:
   ```
   source /opt/client/bigdata_env
   kinit MRS集群用户（MRS集群用户可以是内置用户hbase，或者已加入hbase组中的其他用户，未开启Kerberos认证集群略过该命令）
   cd $PHOENIX_HOME
   bin/sqlline.py zookeerIp:2181
   ```

   说明

   1.MRS 1.9.2之前版本ZooKeeper端口号为24002，详见MRS Manager的Zookeeper集群配置。

   2.若使用phoenix的索引功能，需在HBase服务端(包括HMaster和RegionServer)添加如下配置（详情请参见**https://phoenix.apache.org/secondary_indexing.html**）：

   <property>

   <name>hbase.regionserver.wal.codec</name>

   <value>org.apache.hadoop.hbase.regionserver.wal.IndexedWALEditCodec</value>

   </property>

2. 建表：
   ```
   CREATE TABLE TEST (id VARCHAR PRIMARY KEY, name VARCHAR);
   ```
3. 插入数据：
   ```
   UPSERT INTO TEST(id,name) VALUES ('1','jamee');
   ```
4. 查询数据：
   ```
   SELECT * FROM TEST;
   ```
5. 删表：
   ```
   DROP TABLE TEST;
   ```

4.6.4 REST

MRS1.6之后，支持采用REST的方式来对HBASE进行相应的业务操作，REST API支持curl命令和Java client来操作HBase，有关curl命令的详细使用方法与Apache HBase保持一致，具体请参见**https://hbase.apache.org/book.html#_rest**。

说明

由于当前默认使用 SSL protocols 为 TLSv1.1,TLSv1.2，所以在启用CURL调用 REST 时需判断当前环境支持的 SSL protocols。

使用 curl 命令

- 未开启Kerberos认证集群

  在未开启Kerberos认证的集群中执行curl命令时增加以下参数。例如，

  ```
  curl -vi -k POST -H "Accept: text/xml"  -H "Content-Type: text/xml" -d '<?xml version="1.0" encoding="UTF-8"?> <TableSchema name="users"><ColumnSchema name="cf" /> </TableSchema>' "https://<HBase安装RESTServer服务的节点Ip>:21309/users/schema"
  ```

- 开启Kerberos认证的安全集群

  在安全集群中执行curl命令时，请遵循以下步骤：

a. 进行kerberos认证。例如，

人机用户：*kinit MRS集群用户*

例如：**kinit hbaseuser**

机机用户：*kinit -kt 认证凭据路径 MRS集群用户*

例如：**kinit -kt /opt/user.keytab hbaseuser**

b. 在curl命令中需在请求类型之前添加--negotiate -u：参数。例如，

```
curl -vi -k --negotiate -u: POST -H "Accept: text/xml"   -H "Content-Type: text/xml" -d '<?xml version="1.0" encoding="UTF-8"?> <TableSchema name="users"><ColumnSchema name="cf" /> </TableSchema>' "https://<HBase安装RESTServer服务的节点Ip>:21309/users/schema"
```

使用 Java Client

使用Java调用REST API，请按照以下步骤。（可参见样例代码中RestExample部分代码）。

1. 进行kerberos认证(未开启Kerberos认证集群可以跳过此步骤)
2. 创建一个org.apache.hadoop.hbase.rest.client.Cluster类的集群对象，通过调用集群类的add方法和REST server的集群IP和端口来添加集群。

```
Cluster cluster = new Cluster();
cluster.add("10.10.10.10:21309");
```

3. 使用在步骤2中添加的集群初始化类“org.apache.hadoop.hbase.rest.client.Client”的客户端对象，调用doAs来操作HBase。

```
Client client = new Client(cluster, true);
UserGroupInformation.getLoginUser().doAs(new PrivilegedAction() {
  public Object run() {

    // Rest client code

    /* Sample code to list all the tables
       client.get("/")
    */

    return null;
  }
});
```

4. 可以参考如下的使用方式来了解如何调用不同的Rest API。

- **使用纯文本的方式获取命名空间**

1. 以包含命名空间的路径作为参数，使用client去调用get方法获取命名空间。响应将被“org.apache.hadoop.hbase.rest.client.Response”类的对象捕获。例如：

```
Response response;
String namespacePath = "/namespaces/" + "nameSpaceName";
response = client.get(namespacePath);
System.out.println(Bytes.toString(response.getBody()));
```

- **创建或修改命名空间**

1. 在创建或修改命名空间时，都是使用NamespacesInstanceModel创建模型并使用buildTestModel()方法构建模型，如下所示。这里创建模型，使模型包含要创建的命名空间的属性信息。

```
Map<String, String> NAMESPACE1_PROPS = new HashMap<String, String>();
NAMESPACE1_PROPS.put("key1", "value1");

NamespacesInstanceModel model = buildTestModel(NAMESPACE1,NAMESPACE1_PROPS);

private static NamespacesInstanceModel buildTestModel(String namespace, Map<String, String> properties) {
NamespacesInstanceModel model = new NamespacesInstanceModel();
```



```
for (String key : properties.keySet()) {
model.addProperty(key, properties.get(key));
}
return model;
}
```

说明

在使用POST/PUT请求创建/修改表时，TableSchemaModel是用来创建模型的类。

2. 检查以下步骤以了解如何使用不同的方式创建和修改命名空间。

- **使用xml的方式创建命名空间**

1. 在使用NamespacesInstanceModel创建模型后，以包含命名空间的路径，内容类型（调用类为'org.apache.hadoop.hbase.rest.Constants'，这里调用的参数为Constants.MIMETYPE_XML）和内容（在这里需要将内容转换为xml，使用下面显示的toXML()方法）作为参数，使用client去调用post方法创建命名空间。响应将被'org.apache.hadoop.hbase.rest.client.Response'类的对象捕获。例如：

```
Response response;
String namespacePath = "/namespaces/" + "nameSpaceName";
response = client.post(namespacePath, Constants.MIMETYPE_XML, toXML(model));

private static byte[] toXML(NamespacesInstanceModel model) throws JAXBException {
StringWriter writer = new StringWriter();
context.createMarshaller().marshal(model, writer);
return Bytes.toBytes(writer.toString());
}
```

2. 在使用xml方式进行Get请求时，可使用如下所示的fromXML()方法，从响应中获取模型，并从模型中获取创建的命名空间的名称。

```
private static <T> T fromXML(byte[] content) throws JAXBException {
return (T) context.createUnmarshaller().unmarshal(new ByteArrayInputStream(content));
}
```

- **使用json的方式修改命名空间**

1. 在使用NamespacesInstanceModel创建模型后，调用客户端类的put方法来创建命名空间，参数包含命名空间的路径，内容类型（调用类为org.apache.hadoop.hbase.rest.Constants，这里调用的参数为Constants.MIMETYPE_JSON）和内容（在这里需要转换内容到json，使用jsonMapper作为参数）。响应被'org.apache.hadoop.hbase.rest.client.Response'类的对象捕获。例如：

```
ObjectMapper jsonMapper = new JacksonProvider().locateMapper(NamespacesInstanceModel.class,
MediaType.APPLICATION_JSON_TYPE);

Response response;
String namespacePath = "/namespaces/" + "nameSpaceName";
String jsonString = jsonMapper.writeValueAsString(model);

response = client.put(namespacePath, Constants.MIMETYPE_JSON, Bytes.toBytes(jsonString));
```

2. 在使用json方式进行Get请求时，jsonMapper具有如下所示的readValue()方法，用于从响应中获取模型，并能从模型中获取创建的命名空间。

```
jsonMapper.readValue(response.getBody(), NamespacesInstanceModel.class);

/*Here second argument should be according to API, if its     **related to table it should be
TableSchemaModel.class*/
```

- **使用protobuf的方式修改命名空间**

1. 在使用NamespacesInstanceModel创建模型后，调用客户端类的put方法来创建命名空间，其参数包含命名空间路径，内容类型（调用类为org.apache.hadoop.hbase.rest.Constants'，这里调用的参数为Constants.MIMETYPE_PROTOBUF）和内容（需要转换的内容如下所示，使用

createProtobufOutput来创建protobuf ）。响应将被'org.apache.hadoop.hbase.rest.client.Response'类的对象捕获。例如：

```
Response response;
String namespacePath = "/namespaces/" + "nameSpaceName";

response = client.put(namespacePath, Constants.MIMETYPE_PROTOBUF,
model.createProtobufOutput());
model.getObjectFromMessage(response.getBody());
```

2. 在使用protobuf的方式进行Get请求时，可使用如下所示的getObjectFromMessage 方法，用于从响应中获取模型，并从模型中获取创建的命名空间。

```
model.getObjectFromMessage(response.getBody());
```

4.7 FAQ

4.7.1 运行 HBase 应用开发程序产生异常

提示信息包含org.apache.hadoop.hbase.ipc.controller.ServerRpcControllerFactory的解决办法

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-4.4.0-HBase-1.0.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

4.7.2 bulkload 和 put 应用场景

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。

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。

4.8 开发规范

4.8.1 规则

Configuration 实例的创建

该类应该通过调用HBaseConfiguration的Create()方法来实例化。否则，将无法正确加载HBase中的相关配置项。

正确示例：

```
//该部分，应该是在类成员变量的声明区域声明
private Configuration hbaseConfig = null;
//最好在类的构造函数中，或者初始化方法中实例化该类
hbaseConfig = HBaseConfiguration.create();
```

错误示例：

```
hbaseConfig = new Configuration();
```

共享 Configuration 实例

HBase客户端代码通过创建一个与Zookeeper之间的HConnection，来获取与一个HBase集群进行交互的权限。一个Zookeeper的HConnection连接，对应着一个Configuration实例，已经创建的HConnection实例，会被缓存起来。也就是说，如果客户端需要与HBase集群进行交互的时候，会传递一个Configuration实例过去，HBase Client部分通过已缓存的HConnection实例，来判断属于这个Configuration实例的HConnection实例是否存在，如果不存在，就会创建一个新的，如果存在，就会直接返回相应的实例。

因此，如果频繁创建Configuration实例，会导致创建很多不必要的HConnection实例，很容易达到Zookeeper的连接数上限。

建议在整个客户端代码范围内，都共用同一个Configuration对象实例。

HTable 实例的创建

HTable类有多种构造函数，如：

1. public HTable(final String tableName)
2. public HTable(final byte [] tableName)
3. public HTable(Configuration conf, final byte [] tableName)
4. public HTable(Configuration conf, final String tableName)
5. public HTable(final byte[] tableName, final HConnection connection, final ExecutorService pool)

建议采用第5种构造函数。之所以不建议使用前面的4种，是因为：前两种方法实例化一个HTable时，没有指定Configuration实例，那么，在实例化的时候，就会自动创建一个Configuration实例。如果需要实例化过多的HTable实例，那么，就可能会出现很多不必要的HConnection（关于这一点，前面部分已经有讲述）。因此，而对于第3、4种构造方法，每个实例都可能会创建一个新的线程池，也可能会创建新的连接，导致性能偏低。

正确示例：

```
private HTable table = null;
public initTable(Configuration config, byte[] tableName)
{
// sharedConn和pool都已经是事先实例化好的。建议在一个进程中共享相同的connection和pool。
// 初始化HConnection的方法：
// HConnection sharedConn =
// HConnectionManager.createConnection(this.config);
table = new HTable(config, tableName, sharedConn, pool);
}
```

错误示例：

```
private HTable table = null;
public initTable(String tableName)
{
table = new HTable(tableName);
}
public initTable(byte[] tableName)
{
table = new HTable(tableName);
}
```

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。

HTable 实例缓存

如果一个HTable实例可能会被长时间且被同一个线程固定且频繁的用到，例如，通过一个线程不断的往一个表内写入数据，那么这个HTable在实例化后，就需要缓存下来，而不是每一次插入操作，都要实例化一个HTable对象（尽管提倡实例缓存，但也不是在一个线程中一直沿用一个实例，个别场景下依然需要重构，可参见下一条规则）。

正确示例：

说明

注意该实例中提供的以Map形式缓存HTable实例的方法，未必通用。这与多线程多HTable实例的设计方案有关。如果确定一个HTable实例仅可能会被用于一个线程，而且该线程也仅有一个HTable实例的话，就无须使用Map。这里提供的思路仅供参考。

```
//该Map中以TableName为Key值，缓存所有已经实例化的HTable
private Map<String, HTable> demoTables = new HashMap<String, HTable>();
//所有的HTable实例，都将共享这个Configuration实例
private Configuration demoConf = null;
/**
* <初始化一个HTable类>
* <功能详细描述>
* @param tableName
* @return
* @throws IOException
* @see [类、类#方法、类#成员]
```



```
*/
private HTable initNewTable(String tableName) throws IOException
{
return new HTable(demoConf, tableName);
}
/**
* <获取HTable实例>
* <功能详细描述>
* @see [类、类#方法、类#成员]
*/
private HTable getTable(String tableName)
{
if (demoTables.containsKey(tableName))
{
return demoTables.get(tableName);
} else {
HTable table = null;
try
{
table = initNewTable(tableName);
demoTables.put(tableName, table);
}
catch (IOException e)
{
// TODO Auto-generated catch block
e.printStackTrace();
}
return table;
}
}
/**
* <写数据>
* <这里未涉及到多线程多HTable实例在设计模式上的优化.这里只所以采用同步方法，
* 考虑到同一个HTable是非线程安全的。通常，建议一个HTable实例，在同一
* 时间只能被用在一个写数据的线程中>
* @param dataList
* @param tableName
* @see [类、类#方法、类#成员]
*/
public void putData(List<Put> dataList, String tableName)
{
HTable table = getTable(tableName);
//关于这里的同步：如果在采用的设计方案中，不存在多线程共用同一个HTable实例
//的可能的话，就无须同步了。这里需要注意的一点，就是HTable实例是非线程安全的
synchronized (table)
{
try
{
table.put(dataList);
table.notifyAll();
}
catch (IOException e)
{
                // 在捕获到IOE时，需要将缓存的实例重构。
try {
    // 关闭之前的Connection.
      table.close();
                // 重新创建这个实例.
                table = new HTable(this.config, "jeason");
} catch (IOException e1) {
// TODO
}
}
}
}
```

错误示例：

```
public void putDataIncorrect(List<Put> dataList, String tableName)
{
```

```
HTable table = null;
try
{
//每次写数据，都创建一个HTable实例
table = new HTable(demoConf, tableName);
table.put(dataList);
}
catch (IOException e1)
{
// TODO Auto-generated catch block
e1.printStackTrace();
}
finally
{
table.close();
}
}
```

HTable 实例写数据的异常处理

尽管在前一条规则中提到了提倡HTable实例的重构，但是，并非提倡一个线程自始至终要沿用同一个HTable实例，当捕获到IOException时，依然需要重构HTable实例。示例代码可参考上一个规则的示例。

另外，勿轻易调用如下两个方法：

- Configuration#clear：

  这个方法，会清理掉所有的已经加载的属性，那么，对于已经在使用这个Configuration的类或线程而言，可能会带来潜在的问题（例如，假如HTable还在使用这个Configuration，那么，调用这个方法后，HTable中的这个Configuration的所有的参数，都被清理掉了），也就是说：只要还有对象或者线程在使用这个Configuration，就不应该调用这个clear方法，除非，所有的类或线程，都已经确定不用这个Configuration了。那么，这个操作，可以在所有的线程要退出的时候来做，而不是每一次。

  因此，不要每次实例化一个HTable就调用此方法， 只有当所有线程都要结束时再调用。

- HConnectionManager#deleteAllConnections:

  这个可能会导致现有的正在使用的连接被从连接集合中清理掉，同时，因为在HTable中保存了原有连接的引用，可能会导致这个连接无法关闭，进而可能会造成泄漏。因此，这个方法不建议使用。

写入失败的数据要做相应的处理

在写数据的过程中，如果进程异常或一些其它的短暂的异常，可能会导致一些写入操作失败。因此，对于操作的数据，需要将其记录下来。在集群恢复正常后，重新将其写入到HBase数据表中。

另外，有一点需要注意：HBase Client返回写入失败的数据，是不会自动重试的，仅会告诉接口调用者哪些数据写入失败了。对于写入失败的数据，一定要做一些安全的处理，例如可以考虑将这些失败的数据，暂时写在文件中，或者，直接缓存在内存中。

正确示例：

```
private List<Row> errorList = new ArrayList<Row>();
/**
* <采用PutList的模式插入数据>
* <如果不是多线程调用该方法，可不采用同步>
* @param put 一条数据记录
* @throws IOException
```



```
* @see [类、类#方法、类#成员]
*/
public synchronized void putData(Put put)
{
// 暂时将数据缓存在该List中
dataList.add(put);
// 当dataList的大小达到PUT_LIST_SIZE之后，就执行一次Put操作
if (dataList.size() >= PUT_LIST_SIZE)
{
try
{
demoTable.put(dataList);
}
catch (IOException e)
{
// 如果是RetriesExhaustedWithDetailsException类型的异常，
// 说明这些数据中有部分是写入失败的这通常都是因为
// HBase集群的进程异常引起，有时也会因为有大量
// 的Region正在被转移，导致尝试一定的次数后失败
if (e instanceof RetriesExhaustedWithDetailsException)
{
RetriesExhaustedWithDetailsException ree =
  (RetriesExhaustedWithDetailsException)e;
int failures = ree.getNumExceptions();
for (int i = 0; i < failures; i++)
{
errorList.add(ree.getRow(i));
}
}
}
dataList.clear();
}
}
```

资源释放

关于ResultScanner和HTable实例，在用完之后，需要调用它们的Close方法，将资源释放掉。Close方法，要放在finally块中，来确保一定会被调用到。

正确示例：

```
ResultScanner scanner = null;
try
{
scanner = demoTable.getScanner(s);
//Do Something here.
}
finally
{
scanner.close();
}
```

错误示例：

1. 在代码中未调用scanner.close()方法释放相关资源。
2. scanner.close()方法未放置在finally块中。

   ```
   ResultScanner scanner = null;
   scanner = demoTable.getScanner(s);
   //Do Something here.
   scanner.close();
   ```

Scan 时的容错处理

Scan时不排除会遇到异常，例如，租约过期。在遇到异常时，建议Scan应该有重试的操作。

事实上，重试在各类异常的容错处理中，都是一种优秀的实践，这一点，可以应用在各类与HBase操作相关的接口方法的容错处理过程中。

不用 HBaseAdmin 时，要及时关闭，HBaseAdmin 实例不应常驻内存

HBaseAdmin的示例应尽量遵循 “用时创建，用完关闭”的原则。不应该长时间缓存同一个HBaseAdmin实例。

暂时不建议使用 HTablePool 获取 HTable 实例

因为当前的HTablePool实现中可能会带来泄露。创建HTable实例的方法，参考**不允许多个线程在同一时间共用同一个HTable实例**。

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;
   }
```

4.8.2 建议

不要调用 HBaseAdmin 的 closeRegion 方法关闭一个 Region

HBaseAdmin中，提供了关闭一个Region的接口：

//hostAndPort可以指定，也可以不指定。

public void closeRegion(final String regionname, final String hostAndPort)

通过该方法关闭一个Region，HBase Client端会直接发RPC请求到Region所在的RegionServer上，整个流程对Master而言，是不感知的。也就是说，尽管RegionServer关闭了这个Region，但是，在Master侧，还以为该Region是在该RegionServer上面打开的。假如，在执行Balance的时候，Master计算出恰好要转移这个Region，那么，这个Region将无法被关闭，本次转移操作将无法完成（关于这个问题，在当前的HBase版本中的处理的确还欠缺妥当）。

因此，暂时不建议使用该方法关闭一个Region。

采用 PutList 模式写数据

HTable类中提供了两种写数据的接口:

1. public void put(final Put put) throws IOException
2. public void put(final List<Put> puts) throws IOException

第1种方法较之第2种方法，在性能上有明显的弱势。因此，写数据时应该采用第2种方法。

Scan 时指定 StartKey 和 EndKey

一个有确切范围的Scan，在性能上会带来较大的好处。

代码示例：

```
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("familyname"),Bytes.toBytes("columnname"));
scan.setStartRow( Bytes.toBytes("rowA")); // 假设起始Key为rowA
scan.setStopRow( Bytes.toBytes("rowB")); // 假设EndKey为rowB
for(Result result : demoTable.getScanner(scan)) {
// process Result instance
}
```

不要关闭 WAL

WAL是Write-Ahead-Log的简称，是指数据在入库之前，首先会写入到日志文件中，借此来确保数据的安全性。

WAL功能默认是开启的，但是，在Put类中提供了关闭WAL功能的接口：

public void setWriteToWAL(boolean write)

因此，不建议调用该方法将WAL关闭（即将writeToWAL设置为False），因为可能会造成最近1S（该值由RegionServer端的配置参数hbase.regionserver.optionallogflushinterval决定，默认为1S）内的数据丢失。但如果在实际应用中，对写入的速率要求很高，并且可以容忍丢失最近1S内的数据的话，可以将该功能关闭。

创建一张表或 Scan 时设定 blockcache 为 true

HBase客户端建表和scan时，设置blockcache=true。需要根据具体的应用需求来设定它的值，这取决于有些数据是否会被反复的查询到，如果存在较多的重复记录，将这个值设置为true可以提升效率，否则，建议关闭。

建议按默认配置，默认就是true，只要不强制设置成false就可以，例如：

HColumnDescriptor fieldADesc = new HColumnDescriptor("value".getBytes());

fieldADesc.setBlockCacheEnabled(false);

4.8.3 示例

Configuration 可以设置的参数

为了能够建立一个HBase Client端到HBase Server端的连接，需要设置如下几个参数。

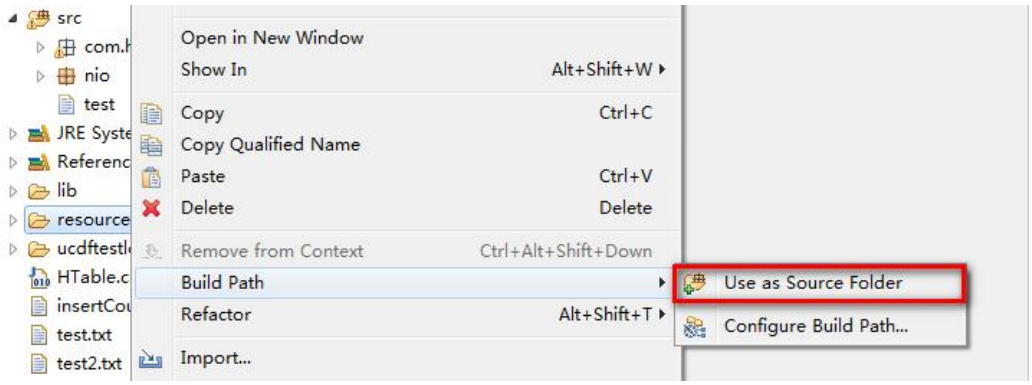
- hbase.zookeeper.quorum： ZooKeeper的IP。多个ZooKeeper节点的话，中间用“,”隔开。
- hbase.zookeeper.property.clientPort： Zookeeper的端口。

说明

通过HBaseConfiguration.create()创建的Configuration实例，会自动加载如下配置文件中的配置项：

- core-default.xml
- core-site.xml
- hbase-default.xml
- hbase-site.xml

因此，这4个配置文件，应该要放置在“Source Folder”下面(将一个文件夹设置为Source Folder的方法：如果在工程下面建立了一个resource的文件夹，那么，可以在该文件夹上右键鼠标，依次选择“Build Path”->“Use as Source Folder”即可，可参考下图)



下面是客户端可配置的一些参数集合。

说明

在通常情况下，这些值都不建议修改。

参数名	参数解释
hbase.client.pause	每次异常或者其它情况下重试等待相关的时间参数(实际等待时间将根据该值与已重试次数计算得出)。
hbase.client.retries.number	异常或者其它情况下的重试次数。
hbase.client.retries.longer.multiplier	与重试次数有关。

参数名	参数解释
hbase.client.rpc.maxattempts	RPC请求不可达时的重试次数。
hbase.regionserver.lease.period	与Scanner超时时间有关（单位ms）。
hbase.client.write.buffer	在启用AutoFlush的情况下，该值不起作用。如果未启用AotoFlush的话，HBase Client端会首先缓存写入的数据，达到设定的大小后才向HBase集群下发一次写入操作。
hbase.client.scanner.caching	Scan时一次next请求获取的行数。
hbase.client.keyvalue.maxsize	一条keyvalue数据的最大值。
hbase.htable.threads.max	HTable实例中与数据操作有关的最大线程数。
hbase.client.prefetch.limit	客户端在写数据或者读取数据时，需要首先获取对应的Region所在的地址。客户端可以预缓存一些Region地址，这个参数就是与缓存的数目有关的配置。

正确设置参数的方法：

hbaseConfig = HBaseConfiguration.create();

//如下参数，如果在配置文件中已经存在，则无须再配置

hbaseConfig.set("hbase.zookeeper.quorum", "10.5.100.1,10.5.100.2,10.5.100.3");

hbaseConfig.set("hbase.zookeeper.property.clientPort", "2181");

HTablePool 在多线程写入操作中的应用

1. 有多个写数据线程时，可以采用HTablePool。现在先简单介绍下该类的使用方法和注意点：
2. 多个写数据的线程之间，应共享同一个HTablePool实例。

   实例化HTablePool的时候，应要指定最大的HTableInterface实例个数maxSize，即需要通过如下构造函数实例化该类：

   public HTablePool(final Configuration config, final int maxSize)

   关于maxSize的值，可以根据写数据的线程数Threads以及涉及到的用户表个数Tables来定，理论上，不应该超过(Threads*Tables)。
3. 客户端线程通过HTablePool#getTable(tableName)的方法，获取一个表名为tableName的HTableInterface实例。
4. 同一个HTableInterface实例，在同一个时刻只能给一个线程使用。
5. 如果HTableInterface使用完了，需要调用HTablePool#putTable(HTableInterface table)方法将它放回去。

示例代码：

```
/**
* 写数据失败后需要一定的重试次数，每一次重试的等待时间，需要根据已经重试的次数而定.
```



```
*/
private static final int[] RETRIES_WAITTIME = {1, 1, 1, 2, 2, 4, 4, 8, 16, 32};
/**
* 限定的重试次数
*/
private static final int RETRIES = 10;
/**
* 失败后等待的基本时间单位
*/
private static final int PAUSE_UNIT = 1000;
private static Configuration hadoopConfig;
private static HTablePool tablePool;
private static String[] tables;
/**
* <初始化HTablePool>
* <功能详细描述>
* @param config
* @see [类、类#方法、类#成员]
*/
public static void initTablePool()
{
DemoConfig config = DemoConfig.getInstance();
if (hadoopConfig == null)
{
hadoopConfig = HBaseConfiguration.create();
hadoopConfig.set("hbase.zookeeper.quorum", config.getZookeepers());
hadoopConfig.set("hbase.zookeeper.property.clientPort", config.getZookeeperPort());
}
if (tablePool == null)
{
tablePool = new HTablePool(hadoopConfig, config.getTablePoolMaxSize());
tables = config.getTables().split(",");
}
}
public void run()
{
// 初始化HTablePool.因为这是多线程间共享的一个实例, 仅被实例化一次.
initTablePool();
for (;;)
{
Map<String, Object> data = DataStorage.takeList();
String tableName = tables[(Integer)data.get("table")];
List<Put> list = (List)data.get("list");
// 以Row为Key，保存List中所有的Put.该集合仅使用于写入失败时查找失败的数据记录.
// 因为从Server端仅返回了失败的数据记录的Row值.
Map<byte[], Put> rowPutMap = null;
// 如果失败了(哪怕是部分数据失败)，需要重试.每一次重试，都仅提交失败的数据条目
INNER_LOOP :
for (int retry = 0; retry < RETRIES; retry++)
{
// 从HTablePool中获取一个HTableInterface实例.用完后需要放回去.
HTableInterface table = tablePool.getTable(tableName);
try
{
table.put(list);
// 如果执行到这里，说明成功了 .
break INNER_LOOP;
}
catch (IOException e)
{
// 如果是RetriesExhaustedWithDetailsException类型的异常，
// 说明这些数据中有部分是写入失败的这通常都是因为HBase集群
// 的进程异常引起，当然有时也会因为有大量的Region正在被转移，
// 导致尝试一定的次数后失败.
// 如果非RetriesExhaustedWithDetailsException异常，则需要将
// list中的所有数据都要重新插入.
if (e instanceof RetriesExhaustedWithDetailsException)
{
RetriesExhaustedWithDetailsException ree =
```

```
(RetriesExhaustedWithDetailsException)e;
int failures = ree.getNumExceptions();
System.out.println("本次插入失败了[" + failures + "]条数据.");
// 第一次失败且重试时，实例化该Map.
if (rowPutMap == null)
{
rowPutMap = new HashMap<byte[], Put>(failures);
for (int m = 0; m < list.size(); m++)
{
Put put = list.get(m);
rowPutMap.put(put.getRow(), put);
}
}
//先Clear掉原数据，然后将失败的数据添加进来
list.clear();
for (int m = 0; m < failures; m++)
{
list.add(rowPutMap.get(ree.getRow(m)));
}
}
}
finally
{
// 用完之后，再将该实例放回去
tablePool.putTable(table);
}
// 如果异常了，就暂时等待一段时间.该等待应该在将HTableInterface实例放回去之后
try
{
sleep(getWaitTime(retry));
}
catch (InterruptedException e1)
{
System.out.println("Interruped");
}
}
}
}
```

Put 实例的创建

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）：

	ColumnFamily01					ColumnFamily02			
	column01	column02	column03	column04	column05	column01	column02	column03	column04
Row--01									
Row--02									
Row--03									
Row--04									
Row--05									
Row--06									
Row--07									
Row--08									

如果要往HBase表中写入一行数据，需要首先构建一个Put实例。Put中包含了数据的Key值和相应的Value值，Value值可以有多个（即可以有多列值）。

有一点需要注意：在往Put实例中add一条KeyValue数据时，传入的family，qualifier，value都是字节数组。在将一个字符串转换为字节数组时，需要使用Bytes.toBytes方法，不要使用String.toBytes方法，因为后者无法保证编码，尤其是在Key或Value中出现中文字符的时候，就会出现问题。

代码示例：

```
//列族的名称为privateInfo
private final static byte[] FAMILY_PRIVATE = Bytes.toBytes("privateInfo");
```

```
//列族privateInfo中总共有两个列"name"&"address"
private final static byte[] COLUMN_NAME = Bytes.toBytes("name");
private final static byte[] COLUMN_ADDR = Bytes.toBytes("address");
/**
* <创建一个Put实例>
* <在该方法中，将会创建一个具有1个列族，2列数据的Put>
* @param rowKey Key值
* @param name 人名
* @param address 地址
* @return
* @see [类、类#方法、类#成员]
*/
public Put createPut(String rowKey, String name, String address)
{
Put put = new Put(Bytes.toBytes(rowKey));
put.add(FAMILY_PRIVATE, COLUMN_NAME, Bytes.toBytes(name));
        put.add(FAMILY_PRIVATE, COLUMN_ADDR, Bytes.toBytes(address));
return put;
}
```

HBaseAdmin 实例的创建以及常用方法

代码示例：

```
private Configuration demoConf = null;
private HBaseAdmin hbaseAdmin = null;
/**
* <构造函数>
* 需要将已经实例化好的Configuration实例传递进来
*/
public HBaseAdminDemo(Configuration conf)
{
this.demoConf = conf;
try
{
// 实例化HBaseAdmin
hbaseAdmin = new HBaseAdmin(this.demoConf);
}
catch (MasterNotRunningException e)
{
e.printStackTrace();
}
catch (ZooKeeperConnectionException e)
{
e.printStackTrace();
}
}
/**
* <一些方法使用示例>
* <更多的方法，请参考HBase接口文档>
* @throws IOException
* @throws ZooKeeperConnectionException
* @throws MasterNotRunningException
* @see [类、类#方法、类#成员]
*/
public void demo() throws MasterNotRunningException, ZooKeeperConnectionException, IOException
{
byte[] regionName = Bytes.toBytes("mrtest,jjj,1315449869513.fc41d70b84e9f6e91f9f01affdb06703.");
byte[] encodeName = Bytes.toBytes("fc41d70b84e9f6e91f9f01affdb06703");
// 重新分配一个Reigon.
hbaseAdmin.unassign(regionName, false);
// 主动触发Balance.
hbaseAdmin.balancer();
// 移动一个Region,第2个参数，是RegionServer的HostName+StartCode,例如：
// host187.example.com,60020,1289493121758.如果将该参数设置为null,则会随机移动该Region
hbaseAdmin.move(encodeName, null);
// 判断一个表是否存在
hbaseAdmin.tableExists("tableName");
// 判断一个表是否被激活
```

```
hbaseAdmin.isTableEnabled("tableName");
}
/**
* <快速创建一个表的方法>
* <首先创建一个HTableDescriptor实例，它里面包含了即将要创建的HTable的描述信息，同时，需要创建相应的列族。列族关联的实例是HColumnDescriptor。在本示例中，创建的列族名称为“columnName”>
* @param tableName 表名
* @return
* @see [类、类#方法、类#成员]
*/
public boolean createTable(String tableName)
{
try {
if (hbaseAdmin.tableExists(tableName)) {
return false;
}
HTableDescriptor tableDesc = new HTableDescriptor(tableName);
HColumnDescriptor fieldADesc = new HColumnDescriptor("columnName".getBytes());
fieldADesc.setBlocksize(640 * 1024);
tableDesc.addFamily(fieldADesc);
hbaseAdmin.createTable(tableDesc);
} catch (Exception e) {
e.printStackTrace();
return false;
}
return true;
}
```

4.8.4 附录

Scan 时的两个关键参数—Batch 和 Caching

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：一个RPC查询请求最大的返回的next数目，与一次RPC获取的**行数**有关。

首先举几个例子，来介绍这两个参数在Scan时所起到的作用：

假设表A的一个Region中存在2行（rowkey）数据，每行有1000column，且每列当前只有一个version，即每行就会有1000个key value。

- **例1**： 查询参数: 不设batch，设定caching=2
  那么，一次RPC请求，就会返回2000个KeyValue。
- **例2**： 查询参数: 设定batch=500，设定caching=2
  那么，一次RPC请求，只能返回1000个KeyValue。
- **例3**： 查询参数: 设定batch=300，设定caching=4
  那么，一次RPC请求，也只能返回1000个KeyValue。

关于Batch和Caching的进一步解释：

- 一次Caching，是一次请求数据的机会。
- 同一行数据是否可以通过一次Caching读完，取决于Batch的设置，如果Batch的值小于一行的总列数，那么，这一行至少需要2次Caching才可以读完（后面的一次Caching的机会，会继续前面读取到的位置继续读取）。
- 一次Caching读取，不能跨行。如果某一行已经读完，并且Batch的值还没有达到设定的大小，也不会继续读下一行了。
  那么，关于例1与例2的结果，就很好解释了：
- 例1的解释：

不设定Batch的时候，默认会读完该行所有的列。那么，在caching为2的时候，一次RPC请求就会返回2000个KeyValue。

- 例2的解释：

  设定Batch为500，caching为2的情况下，也就是说，每一次Caching，最多读取500列数据。那么，第一次Caching，读取到500列，剩余的500列，会在第2次Caching中读取到。因此，两次Caching会返回1000个KeyValue。

- 例3的解释：

  设定Batch为300，caching为4的情况下，读取完1000条数据，正好需要4次caching。因此，只能返回1000条数据。

代码示例：

```
Scan s = new Scan();
//设置查询的起始key和结束key
s.setStartRow(Bytes.toBytes("01001686138100001"));
s.setStopRow(Bytes.toBytes("01001686138100002"));
s.setBatch(1000);
s.setCaching(100);
ResultScanner scanner = null;
try {
scanner = tb.getScanner(s);
for (Result rr = scanner.next(); rr != null; rr = scanner.next()) {
for (KeyValue kv : rr.raw()) {
//显示查询的结果
System.out.println("key:" + Bytes.toString(kv.getRow())
+ "getQualifier:" + Bytes.toString(kv.getQualifier())
+ "value" + Bytes.toString(kv.getValue()));
}
}
} catch (IOException e) {
System.out.println("error!" + e.toString());
} finally {
scanner.close();
}
```

5 Hive 应用开发

5.1 概述

5.1.1 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HiveQL语言操作结构化数据，其基本原理是将HiveQL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HiveQL语言非常容易的完成数据提取、转换和加载（ETL）。
- 通过HiveQL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。

5.1.2 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。本文中的Hive客户端特指Hive client的安装目录，里面包含通过Java API访问Hive的样例代码。

- **HiveQL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Hive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。

5.1.3 开发流程

开发流程中各阶段的说明如图5-1和表5-1所示。

图 5-1 Hive 应用程序开发流程

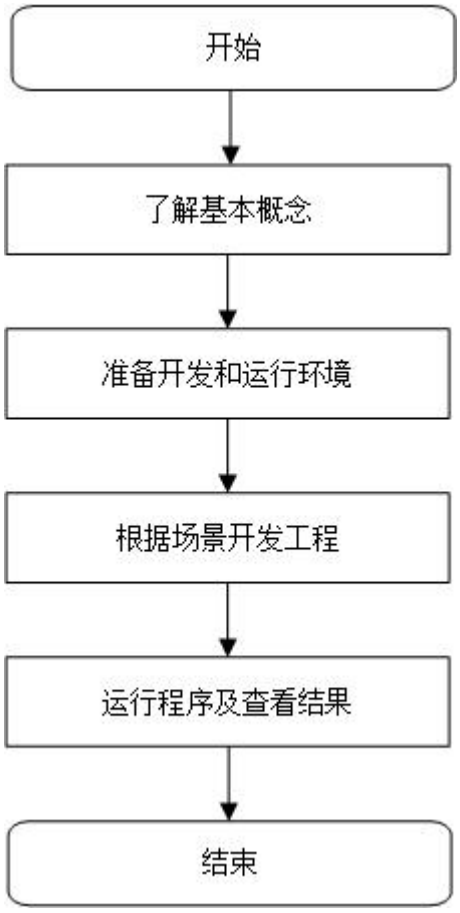


表 5-1 Hive 应用开发的流程说明

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Hive的基本概念。	常用概念
准备开发和运行环境	Hive的应用程序支持使用Java、Python两种语言进行开发。推荐使用Eclipse工具，请根据指导完成不同语言的开发环境配置。	开发环境简介

阶段	说明	参考文档
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。	典型场景说明
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。	• JDBC客户端运行及结果查看 • HCatalog运行及结果查看

5.2 环境准备

5.2.1 开发环境简介

在进行应用开发时，要准备的本地开发环境如表5-2所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行正常。

表 5-2 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：Linux系统。

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。 说明： 在HCatalog的开发环境中，基于安全考虑，MRS服务端只支持TLS 1.1和TLS 1.2加密协议，IBM JDK默认TLS只支持1.0，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TLS1.0/1.1/1.2。 详情请参见：**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Eclipse	用于开发Hive应用程序的工具。版本要求如下： ● JDK使用1.7版本，Eclipse使用3.7.1及以上版本。 JDK使用1.8版本，Eclipse使用4.3.2及以上版本。 说明： 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。
网络	确保客户端与Hive服务主机在网络上互通。

5.2.2 准备环境

- 选择Windows开发环境下，安装Eclipse，安装JDK。
  JDK使用1.8版本，Eclipse使用4.3.2及以上版本。

  说明

  - 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
  - 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
  - 若使用ODBC进行二次开发，请确保JDK版本为1.8及以上版本。
  - 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。
- 准备一个应用程序运行测试的Linux环境。

准备运行调测环境

步骤1 在弹性云服务器管理控制台，申请一个新的弹性云服务器，用于用户应用程序开发、运行、调测。

- 弹性云服务器的安全组需要和MRS集群Master节点的安全组相同。
- 弹性云服务器的VPC需要与MRS集群在同一个VPC中。
- 弹性云服务器的网卡需要与MRS集群在同一个网段中。

步骤2 申请弹性IP，绑定新申请的弹性云主机IP，并配置安全组出入规则。

步骤3 下载客户端程序，请参考**下载MRS客户端**。

步骤4 以root用户安装集群客户端。

1. 执行以下命令解压客户端包。
   tar -xvf /opt/MRS_Services_Client.tar
2. 执行以下命令校验安装文件包。
   sha256sum -c /opt/MRS_Services_ClientConfig.tar.sha256
   MRS_Services_ClientConfig.tar:OK
3. 执行以下命令解压安装文件包。
   tar -xvf /opt/MRS_Services_ClientConfig.tar
4. 执行如下命令安装客户端到指定目录（绝对路径），例如“/opt/client”。目录会自动创建。
   cd /opt/MRS_Services_ClientConfig
   sh install.sh /opt/client
   ```
   Components client installation is complete.
   ```

----结束

5.2.3 准备开发用户

开发用户用于运行样例工程。用户需要有Hive权限，才能运行Hive样例工程。

前提条件

MRS服务集群开启了Kerberos认证需要执行该步骤，没有开启Kerberos认证的集群忽略该步骤。

操作步骤

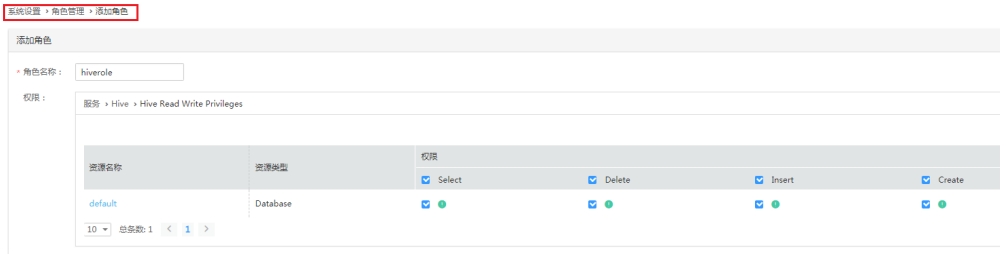
步骤1 登录MRS Manager，请参考**登录MRS Manager**。

步骤2 在MRS Manager界面选择“系统设置 > 角色管理 > 添加角色”，如**图 1 添加角色**所示。

图 5-2 添加 Hive 角色

1. 填写角色的名称，例如*hiverole*。
2. 在“权限”的表格中选择“Hive> Hive Read Write Privileges”，勾选“Select”、“Delete”、“Insert”和“Create”，如**图5-3**所示。

图 5-3 授权 Hive 权限



3. 在“权限”的表格中选择“Yarn > Scheduler Queue > root”，勾选default的“Submit”和“Admin”，如**图5-4**所示。

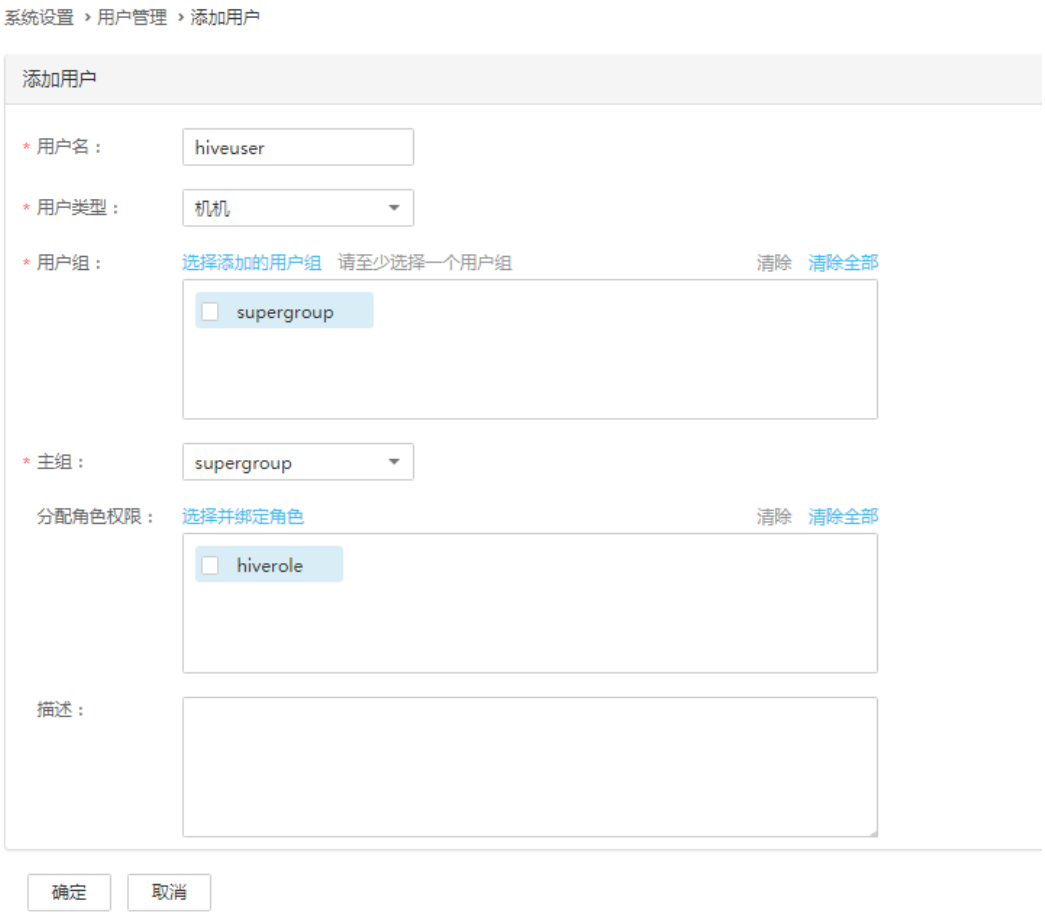
图 5-4 授权 Yarn 权限

4. ，单击“确定”保存。

步骤3 单击“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。

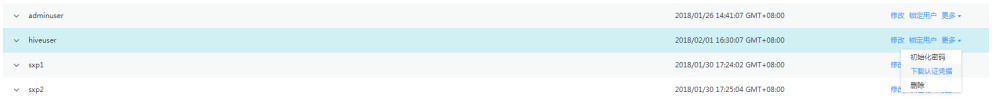
步骤4 填写用户名，例如*hiveuser*，用户类型为“机机”用户，加入用户组**supergroup**，设置其“主组”为**supergroup**，并绑定角色**hiverole**取得权限，单击“确定”，如**图5-5**所示。

图 5-5 添加 Hive 用户



步骤5 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名中选择**hiveuser**，然后在右侧“操作”列中选择“更多 >下载认证凭据”，如图5-6所示。保存后解压得到用户的user.keytab文件与krb5.conf文件。用于在样例工程中进行安全认证。

图 5-6 下载认证凭据



----结束

参考信息

如果修改了组件的配置参数，需重新下载客户端配置文件并更新运行调测环境上的客户端。

5.2.4 准备 JDBC 客户端开发环境

为了运行Hive组件的JDBC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方式连接Hive服务的应用程序为例。

操作步骤

步骤1 在**样例工程获取地址**获取Hive示例工程。

步骤2 在Hive示例工程根目录，执行mvn install编译。

步骤3 在Hive示例工程根目录，执行mvn eclipse:eclipse创建Eclipse工程。

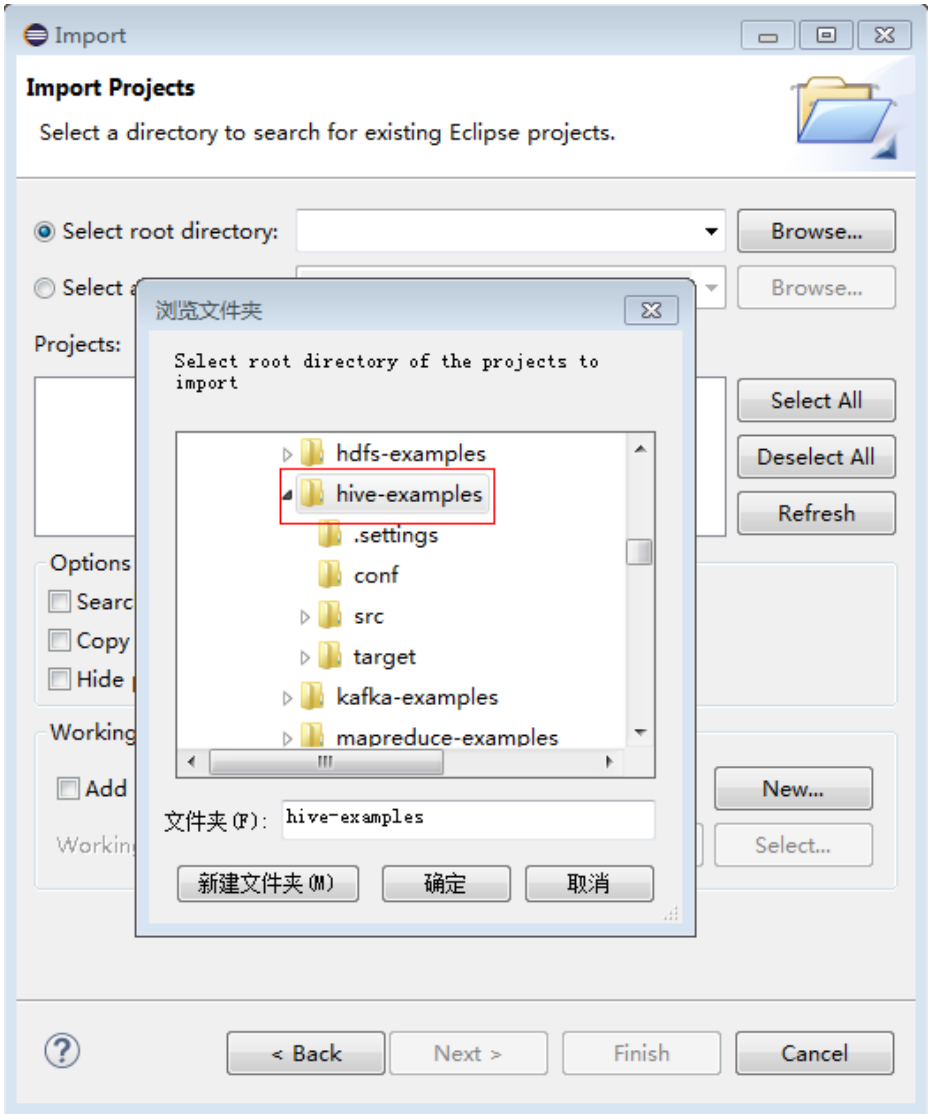
步骤4 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。

   显示“浏览文件夹”对话框。

2. 选择文件夹“hive-examples”，如图5-7所示。Windows下要求该文件夹的完整路径不包含空格。

图 5-7 导入样例工程到 Eclipse 中



单击“Finish”。

导入成功后，com.huawei.bigdata.hive.example包下的JDBCExample类，为JDBC接口样例代码。

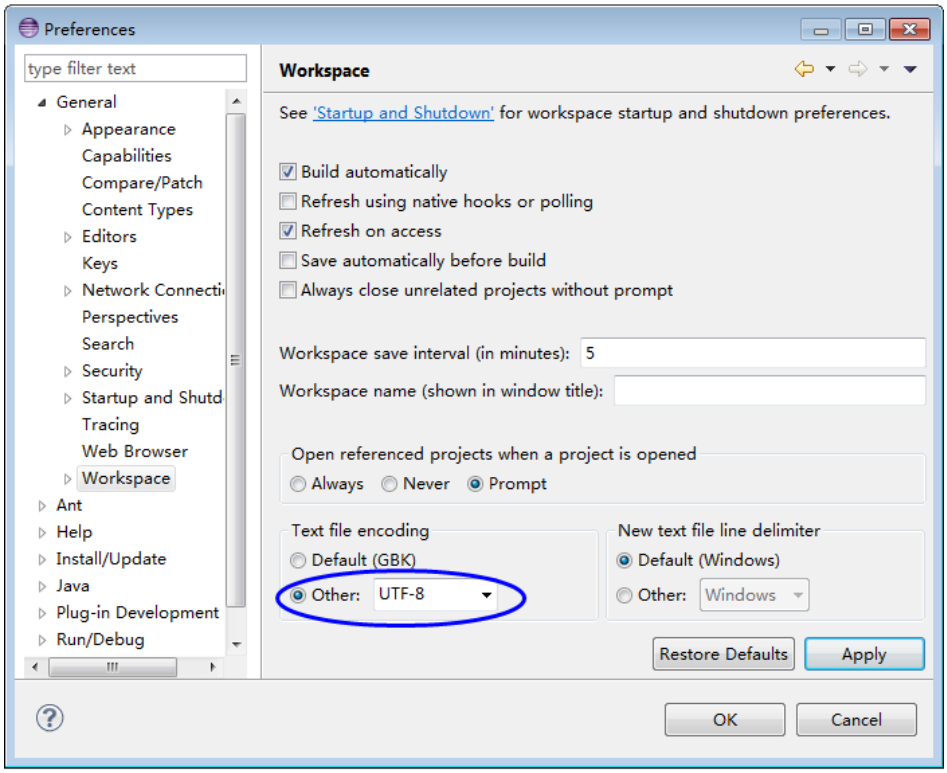
步骤5 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。

   弹出“Preferences”窗口。

2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图5-8所示。

图 5-8 设置 Eclipse 的编码格式



步骤6 修改样例（未开启Kerberos认证集群可跳过此步骤）。

在步骤5获取新建开发用户的krb5.conf和user.keytab文件后，修改ExampleMain.java中的userName为对应的新建用户，例如hiveuser。

```
/**
 * Other way to set conf for zk. If use this way,
 * can ignore the way in the 'login' method
 */
if (isSecurityMode) {
    userName = "hiveuser";
    userKeytabFile = CONF_DIR + "user.keytab";
    krb5File = CONF_DIR + "krb5.conf";
    conf.set(HADOOP_SECURITY_AUTHENTICATION, "kerberos");
    conf.set(HADOOP_SECURITY_AUTHORIZATION, "true");
```

----结束

5.2.5 准备 HCatalog 开发环境

为了运行Hive组件的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HCatalog方式连接Hive服务的应用程序为例。

操作步骤

步骤1 在**样例工程获取地址** 获取Hive示例工程。

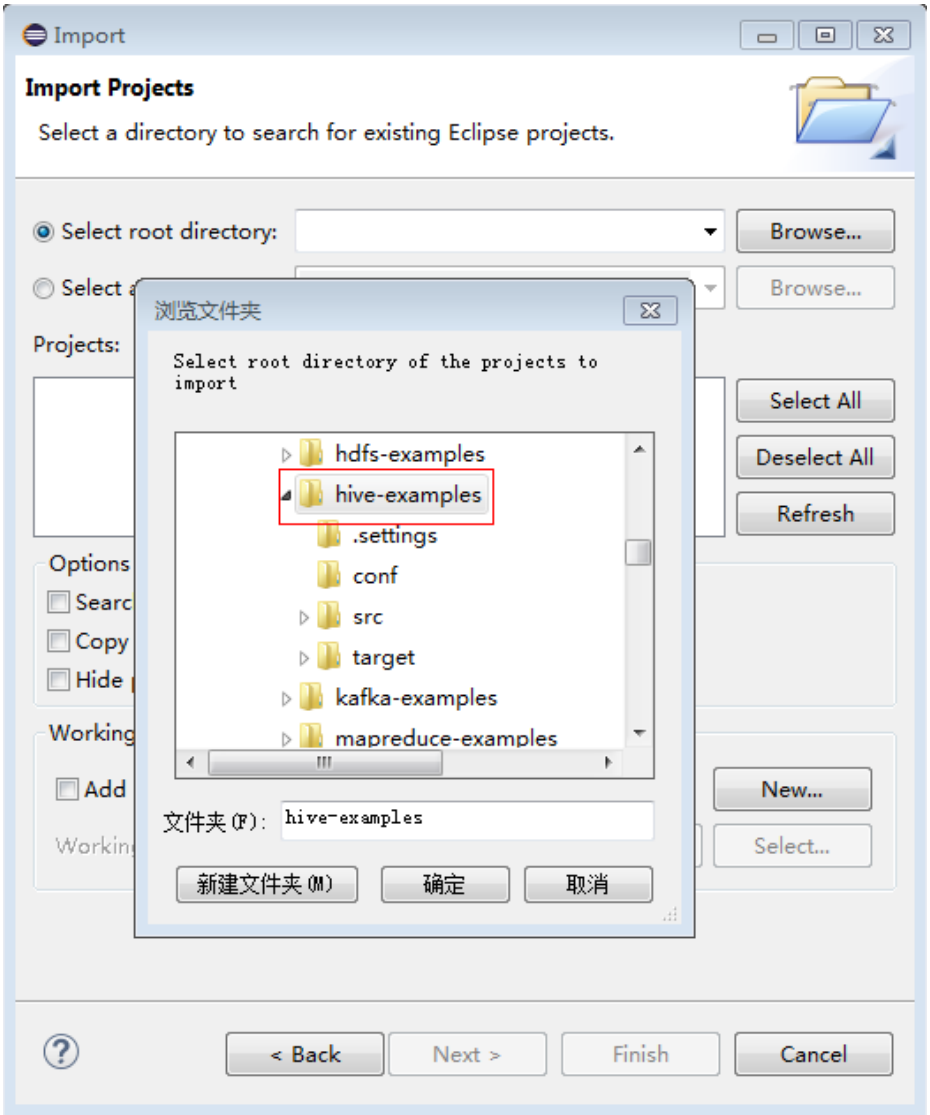
步骤2 在Hive示例工程根目录，执行mvn install编译。

步骤3 在Hive示例工程根目录，执行mvn eclipse:eclipse创建Eclipse工程。

步骤4 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。
   显示“浏览文件夹”对话框。
2. 下载工程后选择文件夹“hive-examples”，如图5-9所示。Windows下要求该文件夹的完整路径不包含空格。

图 5-9 导入样例工程到 Eclipse 中



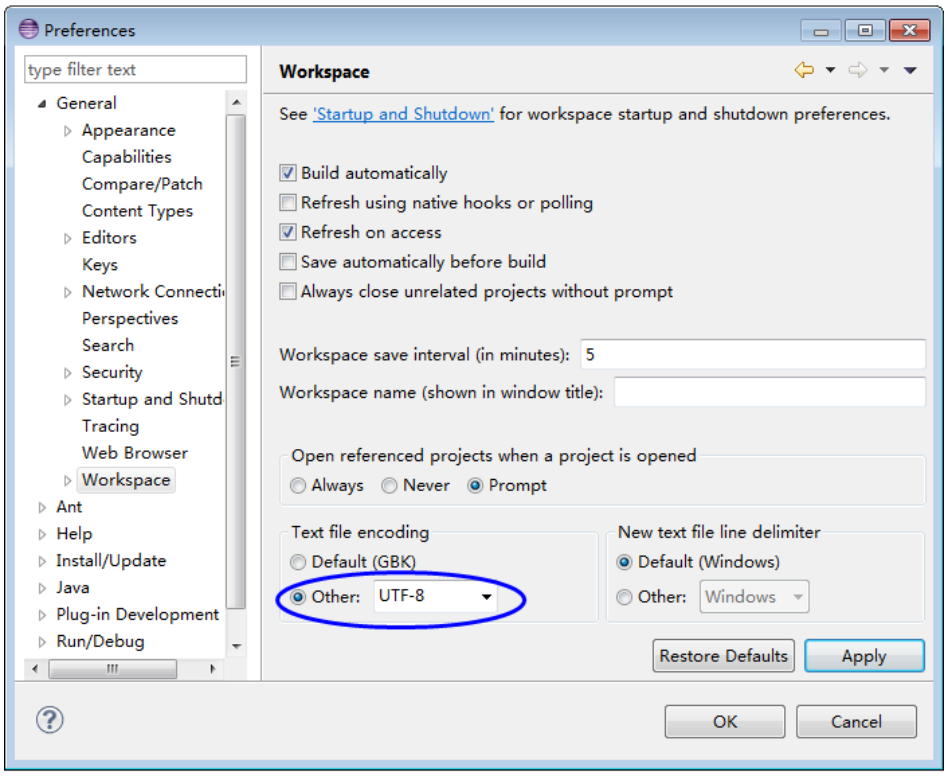
单击“Finish”。

导入成功后，com.huawei.bigdata.hive.example包下的HCatalogExample类，为HCatalog接口样例代码

步骤5 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。弹出“Preferences”窗口。
2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图5-10所示。

图 5-10 设置 Eclipse 的编码格式



----结束

5.3 开发程序

5.3.1 典型场景说明

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5-3、表5-4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**数据加载**。

   雇员信息数据如**表5-3**所示。

   表 5-3 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5-4**所示。

表 5-4 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**数据查询**。

- 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**样例程序指导**。

----结束

5.3.2 创建表

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（如Pig等）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','，"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;

-- 使用CREATE Like创建表.
CREATE TABLE employees_like LIKE employees_info;

-- 使用DESCRIBE查看employees_info、employees_like、 employees_as_select表结构.
DESCRIBE employees_info;
DESCRIBE employees_like;
```

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);
  ```

- 建表时配置Hive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;
  create table seq_Codec (key string, value string) stored as RCFile;
  ```

- HIVE使用OBS存储。

  需要在beeline里面设置指定的参数，AK/SK可登录“OBS控制台”，进入“我的凭证”页面获取。

  ```
  set fs.obs.access.key=AK;
  set fs.obs.secret.key=SK;
  set metaconf:fs.obs.access.key=AK;
  set metaconf:fs.obs.secret.key=SK;
  ```

新建表的存储类型为obs。

```
create table obs(c1 string, c2 string) stored as orc location 'obs://obs-lmm/hive/orctest'
tblproperties('orc.compress'='SNAPPY');
```

说明

当前Hive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";
   ```

5.3.3 数据加载

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集群中加载数据。

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.
LOAD DATA LOCAL INPATH '/opt/hive_examples_data/employee_info.txt' OVERWRITE INTO TABLE
employees_info;

-- 从HDFS上/user/hive_examples_data/employee_info.txt加载进employees_info表中.
LOAD DATA INPATH '/user/hive_examples_data/employee_info.txt' OVERWRITE INTO TABLE employees_info;
```

说明

加载数据的实质是将数据拷贝到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Hive的需求，但是当指定“LOCAL”时，这里的路径指的是当前连接的“HiveServer”的本地文件系统的路径，同时由于当前的“HiveServer”是集群式部署的，客户端在连接时是随机连接所有“HiveServer”中的一个，需要注意当前连接的“HiveServer”的本地文件系统中是否存在需要加载的文件。在无法确定当前连接的是哪一个“HiveServer”的情况下建议在所有的“HiveServer”对应路径下放置相应文件，并注意文件的权限是否正确。

5.3.4 数据查询

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**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**用户自定义函数**。

5.3.5 用户自定义函数

当Hive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分为有如下分类：

- 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。

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。

下面以编写一个AddDoublesUDF为例，说明UDF的编写和使用方法。

功能介绍

AddDoublesUDF主要用来对两个及多个浮点数进行相加。在该样例中可以掌握如何编写和使用UDF。

说明

- 一个普通UDF必须继承自“org.apache.hadoop.hive.ql.exec.UDF”。
- 一个普通UDF必须至少实现一个evaluate()方法，evaluate函数支持重载。

样例代码

以下为UDF示例代码。

```
package com.huawei.bigdata.hive.example.udf;
import org.apache.hadoop.hive.ql.exec.UDF;

public class AddDoublesUDF extends UDF {
 public Double evaluate(Double... a) {
    Double total = 0.0;
    // 处理逻辑部分.
    for (int i = 0; i < a.length; i++)
      if (a[i] != null)
        total += a[i];
    return total;
  }
}
```

如何使用

步骤1 登录MRS Manager页面，为使用函数的Hive业务用户配置Hive管理员权限。

1. 登录MRS Manager页面，单击“系统配置 > 角色管理 > 添加角色”，添加一个拥有Hive Admin Privilege权限的角色。
2. 在MRS Manager页面，单击“系统配置 > 用户管理 ”。
3. 在指定用户对应的“操作”列单击“修改”。
4. 为用户绑定拥有Hive Admin Privilege权限的角色，并单击“确定”完成权限添加。

步骤2 在项目中example目录下新建udf包，并编写AddDoublesUDF类，将项目打包(例如:AddDoublesUDF.jar)，并上传到HDFS指定目录下(如“/user/hive_examples_jars/”)且创建函数的用户与使用函数的用户有该文件的可读权限。示例语句。

hdfs dfs -put AddDoublesUDF.jar /user/hive_examples_jars

hdfs dfs -chmod 777 /user/hive_examples_jars

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。如果当前集群未启用Kerberos认证，则无需执行此命令。当前用户为**准备开发用户**时增加的开发用户。

kinit ***Hive业务用户***

例如：kinit -kt '/opt/conf/user.keytab' hiveuser (user.keytab路径根据自己实际路径填写)

步骤4 执行**set role admin;**命令为用户赋予管理员权限。

步骤5 执行如下命令。

beeline -n *Hive业务用户*

步骤6 在Hive Server中定义该函数，以下语句用于创建永久函数。

CREATE FUNCTION *addDoubles* AS 'com.huawei.bigdata.hive.example.udf.AddDoublesUDF' using jar 'hdfs://hacluster/user/hive_examples_jars/AddDoublesUDF.jar';

其中*addDoubles*是该函数的别名，用于SELECT查询中使用。

以下语句用于创建临时函数。

CREATE TEMPORARY FUNCTION addDoubles AS 'com.huawei.bigdata.hive.example.udf.AddDoublesUDF' using jar 'hdfs://hacluster/user/hive_examples_jars/AddDoublesUDF.jar';

- *addDoubles*是该函数的别名，用于SELECT查询中使用。
- 关键字TEMPORARY说明该函数只在当前这个Hive Server的会话过程中定义使用。

步骤7 在Hive Server中使用该函数，执行SQL语句。

SELECT addDoubles(1,2,3);

说明

若重新连接客户端再使用函数出现[Error 10011]的错误，可执行**reload function;**命令后再使用该函数。

步骤8 在Hive Server中删除该函数，执行SQL语句。

DROP FUNCTION addDoubles;

----结束

5.3.6 样例程序指导

功能介绍

本小节介绍了如何使用样例程序完成分析任务。样例程序主要有以下方式。

- 使用JDBC接口提交数据分析任务。
- 使用HCatalog接口提交数据分析任务。

样例代码

- 使用Hive JDBC接口提交数据分析任务，参考样例程序中的JDBCExample.java。
  a. 定义HiveQL。HiveQL必须为单条语句，注意HiveQL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"};
```

  b. 拼接JDBC URL。

```
// 拼接JDBC URL
StringBuilder sBuilder = new StringBuilder(
  "jdbc:hive2://").append(clientInfo.getZkQuorum()).append("/");

if (isSecurityMode) {
    // 安全模式
    // ZooKeeper登录认证
    sBuilder.append(";serviceDiscoveryMode=")
            .append(clientInfo.getServiceDiscoveryMode())
```

```
            .append(";zooKeeperNamespace=")
            .append(clientInfo.getZooKeeperNamespace())
            .append(";sasl.qop=")
            .append(clientInfo.getSaslQop())
            .append(";auth=")
            .append(clientInfo.getAuth())
            .append(";principal=")
            .append(clientInfo.getPrincipal())
            .append(";");
} else {
    // 普通模式
    sBuilder.append(";serviceDiscoveryMode=")
            .append(clientInfo.getServiceDiscoveryMode())
            .append(";zooKeeperNamespace=")
            .append(clientInfo.getZooKeeperNamespace())
            .append(";auth=none");
}
String url = sBuilder.toString();
```

以上是通过ZooKeeper的方式访问Hive。若直连HiveServer的方式访问Hive，需按如下方式拼接JDBC URL，并将hiveclient.properties文件中的zk.quorum配置项的端口改为10000。

```
// 拼接JDBC URL
StringBuilder sBuilder = new StringBuilder(
  "jdbc:hive2://").append(clientInfo.getZkQuorum()).append("/");

if (isSecurityMode) {
    // 安全模式
    // ZooKeeper登录认证
    sBuilder.append(";sasl.qop=")
            .append(clientInfo.getSaslQop())
            .append(";auth=")
            .append(clientInfo.getAuth())
            .append(";principal=")
            .append(clientInfo.getPrincipal())
            .append(";");
} else {
    // 普通模式
    sBuilder.append(";auth=none");
}
String url = sBuilder.toString();
```

注：直连HiveServer时，若当前连接的HiveServer故障则会导致访问Hive失败；若使用ZooKeeper的访问Hive，只要有任一个HiveServer实例可正常提供服务即可。因此使用JDBC时建议通过ZooKeeper的方式访问Hive。

c. 加载Hive JDBC驱动。

```
  // 加载Hive JDBC驱动
  Class.forName(HIVE_DRIVER);
```

d. 填写正确的用户名，获取JDBC连接，确认HQL的类型（DDL/DML），调用对应的接口执行HiveQL，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();
    }
   }
  }
```

- 使用HCatalog接口提交数据分析任务，参考样例程序中的HCatalogExample.java。

  a. 编写Map类，从Hive的表中获取数据。

```
public static class Map extends
        Mapper<LongWritable, HCatRecord, IntWritable, IntWritable> {
    int age;
    @Override
```

```
    protected void map(
            LongWritable key,
            HCatRecord value,
            Context context)
            throws IOException, InterruptedException {
        age = (Integer) value.get(0);
        context.write(new IntWritable(age), new IntWritable(1));
    }
}
```

b. 编写Reduce类，对从Hive表中读取到的数据进行统计。

```
public static class Reduce extends Reducer<IntWritable, IntWritable,
IntWritable, HCatRecord> {
  @Override
  protected void reduce(
          IntWritable key,
          Iterable<IntWritable> values,
          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);
    }
}
```

c. 在run()方法中配置job后，执行main()方法，提交任务。

```
public int run(String[] args) throws Exception {

    HiveConf.setLoadMetastoreConfig(true);
    Configuration conf = getConf();
    String[] otherArgs = args;

    String inputTableName = otherArgs[0];
    String outputTableName = otherArgs[1];
    String dbName = "default";

    @SuppressWarnings("deprecation")
    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);

    OutputJobInfo outputjobInfo = OutputJobInfo.create(dbName,outputTableName, null);
    HCatOutputFormat.setOutput(job, outputjobInfo);
    HCatSchema schema = outputjobInfo.getOutputSchema();
    HCatOutputFormat.setSchema(job, schema);
    job.setOutputFormatClass(HCatOutputFormat.class);

    return (job.waitForCompletion(true) ? 0 : 1);
}
public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new HCatalogExample(), args);
    System.exit(exitCode);
}
```

5.4 调测程序

5.4.1 在 Windows 中调测程序

5.4.1.1 JDBC 客户端运行及结果查询

JDBC 客户端的命令行形式运行

步骤1 运行样例。

依照**准备JDBC客户端开发环境**中导入和修改样例后，并从集群的任一Master节点的路径“/opt/client/Hive/config/hiveclient.properties”下获取“hiveclient.properties”文件，并放置到样例工程的conf下，即“hive-examples/conf”，即可在开发环境中（例如Eclipse中），右击“ExampleMain.java”，单击“Run as > Java Application”运行对应的应用程序工程。

说明

使用windows访问MRS集群来操作Hive，有如下两种方式。

- 申请一台windows的ECS访问MRS集群操作Hive，此种方式是通过连接zookeeper动态获取HiveServer的地址然后来操作Hive，具有高可用性。
- 使用本机访问MRS集群操作Hive，由于本机与MRS集群的网络不通，只能通过直连HiveServer的方式操作Hive。

方法一：申请一台windows的ECS访问MRS集群操作Hive。在安装开发环境后可直接运行样例代码。申请ECS访问MRS集群的步骤如下。

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”、“集群控制台地址”，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“弹性云服务器 > 快速入门 > 购买并登录Windows弹性云服务器”

方法二：使用本机访问MRS集群操作Hive。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MRS集群中要使用Hive服务的HiveServer节点或ZooKeeper节点绑定弹性公网IP，绑定弹性公网IP步骤如下。
   1. 在虚拟私有云管理控制台，申请一个弹性IP地址，并与弹性云服务器绑定。

      具体请参见“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. 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，若集群为安全集群则需要同时将UDP的21731、21732端口和TCP的21730、21731、21732及Hive的HiveServer实例端口和ZooKeeper服务的端口添加在安全组的入方向规则中。请参见“虚拟私有云 > 用户指南 > 安全性 > 安全组 > 添加安全组规则”。

2. 修改导入的“hiveclient.properties”文件，使得“zk.quorum”参数对应于绑定的HiveServer的弹性公网IP及ZooKeeper端口，并修改样例代码中对连接JDBC时URL的拼接。详情请参见**样例程序指导**中直连HiveServer的方式。
3. 修改导入样例的krb5.conf中"kdc","admin_server",“kpasswd_server”,“kdc_listen”,“kadmind_listen”和“kpasswd_listen”六个参数的ip(单master的集群没有后面三个参数不必修改)，使其对应于KrbServer服务中对应的弹性公网IP（由于普通集群未启用kerberos功能，可跳过此步骤）。并将修改后的krb5.conf和user.keytab文件放置到样例工程的conf目录下。
4. 若通过ZooKeeper的方式访问Hive，需要修改本地hosts文件，添加**步骤1.1**中为节点绑定的公网IP和主机名的映射。
5. 若运行报“Message stream modified (41)”的错误，这可能与JDK的版本有关系，可以尝试修改运行样例代码的JDK为8u_242以下版本或删除“krb5.conf”配置文件的“renew_lifetime = 0m”配置项。

步骤2 查看结果。

查看样例代码中的HiveQL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

5.4.2 在 Linux 中调测程序

5.4.2.1 JDBC 客户端运行及结果查看

步骤1 执行mvn package生成jar包，在工程目录target目录下获取，比如:hive-examples-1.0.jar。

步骤2 在运行调测环境上创建一个目录作为运行目录，如“/opt/hive_examples”(Linux环境)，并在该目录下创建子目录“conf”。

将**步骤1**导出的hive-examples-1.0.jar拷贝到“/opt/hive_examples”下。

将客户端下的配置文件拷贝到“conf”下，开启Kerberos认证的安全集群下把从**步骤5**获取的user.keytab和krb5.conf拷贝到的/opt/hive_examples/conf下，未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。复制${HIVE_HOME}/../config/hiveclient.properties文件到/opt/hive_examples/conf目录下。

```
cd /opt/hive_examples/conf
cp /opt/client/Hive/config/hiveclient.properties .
```

步骤3 准备样例程序相关依赖jar包。

在调测环境上创建一个目录作为存放依赖jar包的目录，如"/opt/hive_examples/lib"(Linux环境)，将${HIVE_HOME}/lib/下面的包全部复制到该目录下，然后删除里面的derby-10.10.2.0.jar(jar包版本号以实际为准)。

```
mkdir /opt/hive_examples/lib
cp ${HIVE_HOME}/lib/* /opt/hive_examples/lib
rm -f /opt/hive_examples/lib/derby-10.10.2.0.jar
```

步骤4 在Linux环境下执行如下命令运行样例程序。

```
chmod +x /opt/hive_examples -R
cd /opt/hive_examples
source /opt/client/bigdata_env
java -cp .:hive-examples-1.0.jar:/opt/hive_examples/conf:/opt/hive_examples/lib/*:/opt/client/HDFS/hadoop/lib/* com.huawei.bigdata.hive.example.ExampleMain
```

步骤5 在命令行终端查看样例代码中的HiveQL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

5.4.2.2 HCatalog 运行及结果查看

步骤1 执行mvn package生成jar包，在工程目录target目录下获取，比如: hive-examples-1.0.jar。

步骤2 将上一步生成的hive-examples-1.0.jar上传至运行调测环境的指定路径，例如“/opt/hive_examples”，记作$HCAT_CLIENT，并确保已经安装好客户端。

```
export HCAT_CLIENT=/opt/hive_examples/
```

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-1.3.0.jar,$HCAT_HOME/lib/hive-metastore-1.3.0.jar,$HIVE_HOME/lib/hive-exec-1.3.0.jar,$HCAT_HOME/lib/libfb303-0.9.3.jar,$HCAT_HOME/lib/slf4j-api-1.7.5.jar,$HCAT_HOME/lib/antlr-2.7.7.jar,$HCAT_HOME/lib/jdo-api-3.0.1.jar,$HCAT_HOME/lib/antlr-runtime-3.4.jar,$HCAT_HOME/lib/datanucleus-api-jdo-3.2.6.jar,$HCAT_HOME/lib/datanucleus-core-3.2.10.jar,$HCAT_HOME/lib/datanucleus-rdbms-3.2.9.jar
export HADOOP_CLASSPATH=$HCAT_HOME/lib/hive-hcatalog-core-1.3.0.jar:$HCAT_HOME/lib/hive-metastore-1.3.0.jar:$HIVE_HOME/lib/hive-exec-1.3.0.jar:$HCAT_HOME/lib/libfb303-0.9.3.jar:$HADOOP_HOME/etc/hadoop:$HCAT_HOME/conf:$HCAT_HOME/lib/slf4j-api-1.7.5.jar:$HCAT_HOME/lib/antlr-2.7.7.jar:$HCAT_HOME/lib/jdo-api-3.0.1.jar:$HCAT_HOME/lib/antlr-runtime-3.4.jar:$HCAT_HOME/lib/datanucleus-api-jdo-3.2.6.jar:$HCAT_HOME/lib/datanucleus-core-3.2.10.jar:$HCAT_HOME/lib/datanucleus-rdbms-3.2.9.jar
```

说明

导入上述环境变量前需确认当前引入的jar包是否存在，具体的版本号可从客户端Hive的lib目录下获取。

步骤4 运行前准备。

1. 如果当前集群已启用Kerberos认证，执行以下命令认证当前用户。如果当前集群未启用Kerberos认证，则无需执行此命令。当前用户为**准备开发用户**时增加的开发用户。

   人机用户：kinit *MRS集群用户*

   例如: **kinit hiveuser**

   机机用户：kinit -kt <user.keytab路径> <MRS集群用户>

   例如：**kinit -kt /opt/hive_examples/conf/user.keytab hiveuser**

   说明

   在连接安全集群时需要在hive客户端的HCatalog的配置文件（例如：/opt/client/Hive/HCatalog/conf/hive-site.xml）中添加如下配置：

   ```
   <property>
   <name>hive.metastore.sasl.enabled</name>
   <value>true</value>
   </property>
   ```

2. 使用Hive客户端，在beeline中创建源表t1：**create table t1(col1 int);**

   执行**insert into t1(col1) values(X);**命令向t1中分别插入如下数据，X表示待插入数据的值。

   ```
   +----------+--+
   | t1.col1  |
   +----------+--+
   | 1        |
   | 1        |
   | 1        |
   | 2        |
   | 2        |
   | 3        |
   ```

3. 创建目的表t2：**create table t2(col1 int,col2 int);**

步骤5 使用YARN客户端提交任务。

yarn --config $HADOOP_HOME/etc/hadoop jar $HCAT_CLIENT/hive-examples-1.0.jar com.huawei.bigdata.hive.example.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://192.168.1.18:24002,192.168.1.> select * from t2;
+----------+----------+--+
| t2.col1  | t2.col2  |
```



```
+----------+----------+--+
| 1        | 3        |
| 2        | 2        |
| 3        | 1        |
+----------+----------+--+
```

----结束

5.5 Hive 接口

5.5.1 JDBC

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

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，执行该类操作会产生“Method not supported”的SQLException异常。

5.5.2 HiveQL

HiveQL支持当前使用的MRS Hive与对应开源Hive版本中的所有特性，详情请参见**https://cwiki.apache.org/confluence/display/hive/languagemanual**。MRS Hive版本与开源Hive版本的对应关系如**表5-5**所示。

表 5-5 MRS Hive 与开源版本对应关系一览表

MRS版本	开源Hive版本
MRS 1.9.x	2.3.3

5.5.3 WebHCat

说明

- 以下示例的IP为WebHCat所在节点的业务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。

: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 -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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	WebChat连接正常，返回OK。
version	字符串，包含版本号，比如v1。

- 例子

```
curl -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/status'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/version'
```

  说明

  - 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
  - MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
  - 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/version/hive'
```

  说明

  - 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
  - MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
  - 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/version/hadoop'
```

  说明

  - 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
  - MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
  - 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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。

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 -ik -u : --negotiate -d exec="show tables" 'http://10.64.35.144:9111/templeton/v1/ddl?user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database?user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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，如果没有备注则其参数值不存在
database	数据库名
owner	数据库的所有者
ownertype	数据库所有者的类型

- 例子

```
curl -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database/default?user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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

参数	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 -ik -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'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate -X DELETE 'http://10.64.35.144:9111/templeton/v1/ddl/database/db3?ifExists=true&user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/table?user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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

  http://www.myserver.com/templeton/v1/ddl/database/:db/table/:table?format=extended

- 参数

参数	描述
:db	数据库名
:table	表名
format	设置"format=extended"可查看表的更多信息。（其作用相当于HQL:“show table extended like tableName”）。

- 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 -ik -u : --negotiate 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1?format=extended&user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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。
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 -ik -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'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate -d rename=table1 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/tbl1?user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate -X DELETE 'http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/table2?ifExists=true&user.name=user1'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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。
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 -ik -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'
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition?user.name=user1
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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

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 -ik -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/x1/partition/dt=1?user.name=user1
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -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
  ```

  说明

  - 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
  - MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
  - 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -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
  ```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column?user.name=user1
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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
: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 -ik -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/column/id?user.name=user1
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -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
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property?user.name=user1
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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
property	属性列表

- 例子

```
curl -ik -u : --negotiate http://10.64.35.144:9111/templeton/v1/ddl/database/default/table/t1/property/last_modified_by?user.name=user1
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -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
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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。
libjars	需要加入的classpath的jar包名，以逗号分隔。

参数	描述
files	需要拷贝到MR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 -ik -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"
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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，WebChat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://10.64.35.144:50111/templeton/v1/mapreduce/streaming?user.name=user1'
```

说明

- 本接口的使用需要前置条件，请参阅**规则**。
- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。

/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	需要拷贝到MR集群的文件名，以逗号分隔。
arg	设置argument。
define	设置hadoop的配置，格式：define=key=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/$attempt_id/syslog
callback	在MR任务执行完的回调地址，使用$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 -ik -u : --negotiate -d execute="select count(*) from t1" -d statusdir="/output" "http://10.64.35.144:9111/templeton/v1/hive?user.name=user1"
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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。
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 -ik -u : --negotiate  "http://10.64.35.144:9111/templeton/v1/jobs?user.name=user1"
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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。
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。
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 -ik -u : --negotiate  "http://10.64.35.144:9111/templeton/v1/jobs/job_1440386556001_0255?user.name=user1"
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- MRS 1.9.2及之后版本默认端口为9111。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.port”配置。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

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 -ik -u : --negotiate -X DELETE  "http://10.64.35.143:9111/templeton/v1/jobs/
job_1440386556001_0265?user.name=user1"
```

说明

- 示例中的IP为WebHCat所在节点的业务IP，端口为安装时设置的WebHCat端口。
- 示例中的协议类型在普通集群中为“http”,安全集群为“https”。详见MRS Manager管理界面“服务管理 > Hive > 服务配置”中“templeton.protocol.type”配置。

5.6 开发规范

5.6.1 规则

Hive JDBC 驱动的加载

客户端程序以JDBC的形式连接HiveServer时，需要首先加载Hive的JDBC驱动类org.apache.hive.jdbc.HiveDriver。

故在客户端程序的开始，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Hive的数据库连接。

Hive的数据库URL为url="jdbc:hive2://xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver;sasl.qop=auth-conf;auth=KERBEROS;principal=hive/hadoop.hadoop.com@HADOOP.COM;user.principal=hive/hadoop.hadoop.com;user.keytab=conf/hive.keytab";

以上已经经过安全认证，所以Hive数据库的用户名和密码为null或者空。

如下：

```
// 建立连接
connection = DriverManager.getConnection(url, "", "");
```

执行 HQL

执行HQL，注意HQL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();
```

关闭数据库连接

客户端程序在执行完HQL之后，注意关闭数据库连接，以免内存泄露，同时这是一个良好的编程习惯。

需要关闭JDK的两个对象statement和connection。

如下：

```
finally {
        if (null != statement) {
            statement.close();
        }

        // 关闭JDBC连接
        if (null != connection) {
            connection.close();
        }
    }
```

HQL 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HQL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。
```

客户端配置参数需要与服务端保持一致

当集群的Hive、YARN、HDFS服务端配置参数发生变化时，客户端程序对应的参数会被改变，用户需要重新审视在配置参数变更之前提交到HiveServer的配置参数是否和服务端配置参数一致，如果不一致，需要用户在客户端重新调整并提交到HiveServer。例如下面的示例中，如果修改了集群中的YARN配置参数时，Hive客户端、示例程序都需要审视并修改之前已经提交到HiveServer的配置参数。

初始状态

集群YARN的参数配置如下：

```
mapreduce.reduce.java.opts=-Xmx2048M
```

客户端的参数配置如下：

```
mapreduce.reduce.java.opts=-Xmx2048M
```

集群YARN修改后，参数配置如下：

```
mapreduce.reduce.java.opts=-Xmx1024M
```

如果此时客户端程序不做调整修改，则还是以客户端参数有效，会导致reducer内存不足而使MR运行失败。

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;
}
```

使用 WebHCat 的 REST 接口以 Streaming 方式提交 MR 任务的前置条件

本接口需要依赖hadoop的streaming包，在以Streaming方式提交MR任务给WebHCat前，需要将“hadoop-streaming-x.x.x.jar”包上传到HDFS的指定路径下“hdfs:///apps/templeton/hadoop-streaming-xxx.jar”，具体步骤如下。

步骤1 登录到安装有客户端和Hive服务的节点上，以客户端安装路径为“/opt/client”为例。

source /opt/client/bigdata_env

步骤2 使用kinit登录集群的人机用户或者机机用户。

步骤3 执行如下命令将streaming包放到hdfs的/apps/templeton下。

- 对于MRS 1.9.2及之后的版本，执行如下命令。

  hdfs dfs -put /opt/Bigdata/MRS_x.x.x/install/FusionInsight-Hadoop-x.x.x/hadoop/share/hadoop/tools/lib/hadoop-streaming-x.x.x-mrs-x.x.jar /apps/templeton

  其中，**/apps/templeton/**需要根据不同的实例进行修改，默认实例使用**/apps/templeton/**，Hive1实例使用**/apps1/templeton/**，以此类推。

hdfs dfs -put /opt/Bigdata/MRS_x.x.x/install/FusionInsight-Hadoop-x.x.x/hadoop/share/hadoop/tools/lib/hadoop-streaming-x.x.x-mrs-x.x.jar /apps/templeton

其中，**/apps/templeton/**需要根据不同的实例进行修改，默认实例使用**/apps/templeton/**，Hive1实例使用**/apps1/templeton/**，以此类推。

----结束

例如：提交streaming的mr作业。

1. 创建hdfs目录。

   hdfs dfs -mkdir /user/root/input/

2. 将自定义数据文件new.txt放到hdfs目录上。

   hdfs dfs -put new.txt /user/root/input/

3. 提交mr作业。
   - 对于MRS 1.9.2及之后的版本，执行如下命令。

     $HADOOP_HOME/bin/hadoop jar /opt/client/HDFS/hadoop/share/hadoop/tools/lib/hadoop-streaming-x.x.x-mrs-x.x.jar -input input/new.txt -output output -mapper 'cut -f 2 -d ,' -reducer 'uniq'

     其中，参数含义如下：

     - -output output的后一个output为生成文件存放的目录，命令执行后会自动创建，命令执行前需保证/user/root/目录下无此目录，否则报目录存在错误。
     - 以上命令中所用的hadoop-streaming版本jar包请根据集群中提供的实际jar包名称来修改。
     - -mapper、-reducer后的参数用户可自定义。

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。

分桶表不支持 insert into

分桶表（bucket table）不支持insert into，仅支持insert overwrite，否则会导致文件个数与桶数不一致。

使用 WebHCat 的部分 REST 接口的前置条件

WebHCat的部分REST接口使用依赖于MapReduce的JobHistoryServer实例，具体接口如下：

- mapreduce/jar(POST)
- mapreduce/streaming(POST)
- hive(POST)
- jobs(GET)
- jobs/:jobid(GET)
- jobs/:jobid(DELETE)

Hive 授权说明

Hive授权（数据库、表或者视图）推荐通过Manager授权界面进行授权，不推荐使用命令行授权，除了“alter databases databases_name set owner='user_name'”场景以外。

不允许创建 Hive on HBase 的分区表

Hive on HBase表将实际数据存储在HBase上。由于HBase会将表划分为多个分区，将分区散列在RegionServer上，因此不允许在Hive中创建Hive on HBase分区表。

Hive on HBase 表不支持 INSERT OVERWRITE

HBase中使用rowkey作为一行记录的唯一标识。在插入数据时，如果rowkey相同，则HBase会覆盖该行的数据。如果在Hive中对一张Hive on HBase表执行INSERT OVERWRITE，会将相同rowkey的行进行覆盖，不相关的数据不会被覆盖。

5.6.2 建议

HQL 编写之隐式类型转换

查询语句使用字段的值做过滤时，不建议通过Hive自身的隐式类型转换来编写HQL。因为隐式类型转换不利于代码的阅读和移植。

建议示例：

```
select * from default.tbl_src where id = 10001;
select * from default.tbl_src where name = 'TestName';
```

不建议示例：

```
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

HQL 编写之对象名称长度

HQL的对象名称，包括表名、字段名、视图名、索引名等，其长度建议不要超过30个字节。

Oracle中任何对象名称长度不允许超过30个字节，超过时会报错。为了兼容Oracle，对对象的名称进行了限制，不允许超过30个字节。

太长不利于阅读、维护、移植。

HQL 编写之记录个数统计

统计某个表所有的记录个数，建议使用“select count(1) from table_name”。

统计某个表某个字段有效的记录个数，建议使用“select count(column_name) from tbale_name”。

JDBC 超时限制

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。

Hive 并发编译 SQL

给HiveServer配置参数hive.driver.parallel.compilation=true并且重启HiveServer，可以使得HiveServer支持多个Session之间并发编译SQL。

在默认情况下，HiveServer是关闭并发编译SQL的，这意味着HiveServer只能串行的编译SQL，当有大量的执行时间短的SQL时，会影响整体性能。例如执行大量Analyse操作时，开启并发编译可以极大提高性能。

建 Hive 分区表策略

当某个表的目录下有海量的数据，使用Hive进行操作时，会搜索这个表的所有文件，这会非常耗时。如果知道这些数据的某些特征，可以事先将其分裂存放到hive的不同目录下，在查询时就可以在where子句中对这些特征进行过滤，那么对数据的操作就只会在符合条件的子目录下进行，其他不符合条件的目录就不会被读取。这种将表中数据分散到子目录的方式就是分区表。

因此建分区表的前提是表存在大量的数据或者未来存放大量数据，否则将会导致文件系统中产生大量的小文件。建议满足如下条件可以建立分区表：

- 表中的数据按照时间维度导入，且每个时间周期内数据大于32M，此时可以将时间字段作为分区字段建立分区表。
- 表的总大小未来预计大于10G时，可以建立分区表。

增加高斯 DB 备节点查询成功率

在备库上执行查询时，经常会出现如下错误：

```
ERROR:  canceling statement due to conflict with recovery
DETAIL:  User query might have needed to see row versions that must be removed.
```

由于备库从主库同步数据时发现备库执行了耗时较长的SQL，会主动将SQL取消，SQL最长执行时间默认为30s。此问题需要在高斯备节点执行如下语句：

```
su ommdba
gs_guc reload -c "hot_standby_feedback=on"
```

这个参数的设置是有利有弊，好处就是减少了冲突，缺点就是由于主库的清理需要等待备库的事务结束，那么在频繁更新的场景下，可能造成主库数据膨胀。因此只建议在备库上执行不频繁且耗时段的SQL。

5.6.3 示例

JDBC 二次开发示例代码一

以下示例代码主要功能如下。

1. 在JDBC URL地址中提供用户名和密钥文件路径，程序自动完成安全登录、建立Hive连接。

   说明

   MRS 1.9.2及之前版本ZooKeeper端口号默认为24002，详见MRS Manager的Zookeeper配置。

2. 执行创建表、查询和删除三类HQL语句。

```
package com.huawei.bigdata.hive.example;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Properties;


import org.apache.hadoop.conf.Configuration;
import com.huawei.bigdata.security.LoginUtil;

public class JDBCExample {
  private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";

  private static final String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
  private static final String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
  private static final String ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/hadoop";

  private static Configuration CONF = null;
  private static String KRB5_FILE = null;
  private static String USER_NAME = null;
  private static String USER_KEYTAB_FILE = null;

  private static String zkQuorum = null;//zookeeper节点ip和端口列表
  private static String auth = null;
  private static String sasl_qop = null;
  private static String zooKeeperNamespace = null;
  private static String serviceDiscoveryMode = null;
  private static String principal = null;
```

```
 private static void init() throws IOException{
   CONF = new Configuration();

   Properties clientInfo = null;
   String userdir = System.getProperty("user.dir") + File.separator
       + "conf" + File.separator;
   System.out.println(userdir);
   InputStream fileInputStream = null;
   try{
     clientInfo = new Properties();
     //"hiveclient.properties"为客户端配置文件，如果使用多实例特性，需要把该文件换成对应实例客户端下的
"hiveclient.properties"
     //"hiveclient.properties"文件位置在对应实例客户端安装包解压目录下的config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;
     }
   }
   //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");
   // 设置新建用户的USER_NAME，其中"xxx"指代之前创建的用户名，例如创建的用户为user，则USER_NAME
为user
   USER_NAME = "userx";

   if ("KERBEROS".equalsIgnoreCase(auth)) {
     // 设置客户端的keytab和krb5文件路径
     USER_KEYTAB_FILE = userdir + "user.keytab";
     KRB5_FILE = userdir + "krb5.conf";

     LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, USER_NAME,
USER_KEYTAB_FILE);
     LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);

     // 安全模式
     // Zookeeper登录认证
     LoginUtil.login(USER_NAME, USER_KEYTAB_FILE, KRB5_FILE, CONF);
   }
 }

 /**
  * 本示例演示了如何使用Hive JDBC接口来执行HQL命令<br>
  * <br>
  *
  * @throws ClassNotFoundException
  * @throws IllegalAccessException
  * @throws InstantiationException
  * @throws SQLException
  * @throws IOException
  */
 public static void main(String[] args) throws InstantiationException,
     IllegalAccessException, ClassNotFoundException, SQLException, IOException{
   // 参数初始化
   init();

   // 定义HQL，HQL为单条语句，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"};

  // 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(";");
  } else {
   //普通模式
    sBuilder.append(";serviceDiscoveryMode=")
            .append(serviceDiscoveryMode)
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";auth=none");
  }
  String url = sBuilder.toString();

  // 加载Hive JDBC驱动
  Class.forName(HIVE_DRIVER);

  Connection connection = null;
  try {
    System.out.println(url);
    // 获取JDBC连接
    // 如果使用的是普通模式，那么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, "", "");

    // 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();
    }
  }
 }
}
```

JDBC 二次开发示例代码二

以下示例代码主要功能如下。

1. 用户自行进行安全登录，不在JDBC URL地址中提供用户和密钥文件路径，建立Hive连接。
2. 执行创建表、查询和删除三类HQL语句。

说明

程序在访问ZooKeeper时会使用jaas配置文件，例如user.hive.jaas.conf，具体信息如下。

```
Client {
com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true
keyTab="D:\\workspace\\jdbc-examples\\conf\\user.keytab"
principal="xxx@HADOOP.COM"
useTicketCache=false
storeKey=true
debug=true;
};
```

用户需要根据实际环境，修改上述配置文件的keyTab路径（绝对路径）和principal，并设置环境变量java.security.auth.login.config指向该文件所在路径。

```
package com.huawei.bigdata.hive.example;

import static
org.apache.hadoop.fs.CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHENTICATION;
import static org.apache.hadoop.fs.CommonConfigurationKeysPublic.HADOOP_SECURITY_AUTHORIZATION;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Properties;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.security.SecurityUtil;
import org.apache.hadoop.security.UserGroupInformation;

public class JDBCExamplePreLogin {
  private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";

  /**
   * 本示例演示了如何使用Hive JDBC接口来执行HQL命令<br>
   * <br>
   *
   * @throws ClassNotFoundException
   * @throws IllegalAccessException
   * @throws InstantiationException
   * @throws SQLException
   */
  public static void main(String[] args) throws InstantiationException,
      IllegalAccessException, ClassNotFoundException, SQLException ,IOException{

    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，如果使用多实例特性，需要把该文件换成对应实例客户端下的
"hiveclient.properties"
      //"hiveclient.properties"文件位置在对应实例客户端安装包解压目录下的config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;
      }
    }
    //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
    String zkQuorum =  clientInfo.getProperty("zk.quorum");
    String auth = clientInfo.getProperty("auth");
    String sasl_qop = clientInfo.getProperty("sasl.qop");
    String zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    String serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
    String principal = clientInfo.getProperty("principal");

    // 定义HQL，HQL为单条语句，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"};

  // 拼接JDBC URL
  StringBuilder sBuilder = new StringBuilder(
      "jdbc:hive2://").append(zkQuorum).append("/");

  if ("KERBEROS".equalsIgnoreCase(auth)) {

    // 设置属性java.security.krb5.conf，以此指定将访问的安全服务的信息
    System.setProperty("java.security.krb5.conf", "conf/krb5.conf");
    // 设置需要使用的jaas配置文件，请根据实际情况修改user.hive.jaas.conf中的keyTab和principal
    System.setProperty("java.security.auth.login.config",
        "conf/user.hive.jaas.conf");

    Configuration conf = new Configuration();
    conf.set(HADOOP_SECURITY_AUTHENTICATION, "kerberos");
    conf.set(HADOOP_SECURITY_AUTHORIZATION, "true");
    String PRINCIPAL = "username.client.kerberos.principal";
    String KEYTAB = "username.client.keytab.file";
    // 设置客户端的keytab文件路径
    conf.set(KEYTAB, "conf/user.keytab");
    // 设置新建用户的userPrincipal，此处填写为带域名的用户名，例如创建的用户为user，域为
HADOOP.COM，则其userPrincipal则为user@HADOOP.COM。
    conf.set(PRINCIPAL, "xxx@xxx");

    // 进行登录认证
    UserGroupInformation.setConfiguration(conf);
    SecurityUtil.login(conf, KEYTAB, PRINCIPAL);

    // 安全模式
    sBuilder.append(";serviceDiscoveryMode=")
            .append(serviceDiscoveryMode)
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";sasl.qop=")
            .append(sasl_qop)
            .append(";auth=")
            .append(auth)
            .append(";principal=")
            .append(principal)
            .append(";");
  } else {
    // 普通模式
    sBuilder.append(";serviceDiscoveryMode=")
            .append(serviceDiscoveryMode)
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";auth=none");
  }
  String url = sBuilder.toString();

  // 加载Hive JDBC驱动
  Class.forName(HIVE_DRIVER);

  Connection connection = null;
  try {
    // 获取JDBC连接
    // 如果使用的是普通模式，那么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, "", "");

    // 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();
    }
  }
}

}
```

HCatalog 二次开发示例代码

以下示例代码演示了如何使用HCatalog提供的HCatInputFormat和HCatOutputFormat接口提交MapReduce任务。

```
public class HCatalogExample extends Configured implements Tool {

  public static class Map extends
        Mapper<LongWritable, HCatRecord, IntWritable, IntWritable> {
    int age;
    @Override
    protected void map(
          LongWritable key,
          HCatRecord value,
          org.apache.hadoop.mapreduce.Mapper<LongWritable, HCatRecord,
                IntWritable, IntWritable>.Context context)
          throws IOException, InterruptedException {
      age = (Integer) value.get(0);
      context.write(new IntWritable(age), new IntWritable(1));
    }
  }

  public static class Reduce extends Reducer<IntWritable, IntWritable,
  IntWritable, HCatRecord> {
    @Override
    protected void reduce(
          IntWritable key,
          java.lang.Iterable<IntWritable> values,
          org.apache.hadoop.mapreduce.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);
    }
  }

  public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    String[] otherArgs = args;

    String inputTableName = otherArgs[0];
    String outputTableName = otherArgs[1];
    String dbName = "default";

    @SuppressWarnings("deprecation")
    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);

    OutputJobInfo outputjobInfo = OutputJobInfo.create(dbName,outputTableName, null);
    HCatOutputFormat.setOutput(job, outputjobInfo);
    HCatSchema schema = outputjobInfo.getOutputSchema();
```

```
        HCatOutputFormat.setSchema(job, schema);
        job.setOutputFormatClass(HCatOutputFormat.class);

        return (job.waitForCompletion(true) ? 0 : 1);
    }
    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new HCatalogExample(), args);
        System.exit(exitCode);
    }
}
```

6 MapReduce 应用开发

6.1 概述

6.1.1 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

6.1.2 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。

6.1.3 开发流程

开发流程中各阶段的说明如图6-1和表6-1所示。

图 6-1 MapReduce 应用程序开发流程

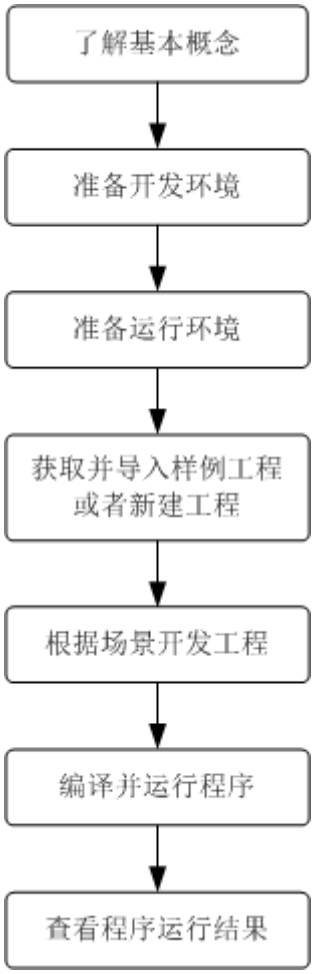


表 6-1 MapReduce 应用开发的流程说明

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。	**常用概念**
准备开发环境	使用Eclipse工具，请根据指导完成开发环境配置。	**准备Eclipse与JDK**
准备运行环境	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Linux客户端运行环境**
获取并导入样例工程 或者新建工程	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。	**获取并导入样例工程**

阶段	说明	参考文档
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。	• MapReduce统计样例程序 • MapReduce访问多组件样例程序
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。	编译并运行程序
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。	查看调测结果

6.2 环境准备

6.2.1 开发环境简介

在进行应用开发时，要准备的开发环境如表6-2所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行正常。

表 6-2 开发环境

准备项	说明
安装Eclipse	开发环境的基本配置。版本要求：4.2。
安装JDK	版本要求：1.8版本。

6.2.2 准备开发用户

开发用户用于运行样例工程。用户需要有组件权限，才能运行样例工程。

前提条件

MRS服务集群开启了Kerberos认证，没有开启Kerberos认证的集群忽略该步骤。

操作步骤

步骤1 登录MRS Manager，在MRS Manager界面选择“系统设置 > 角色管理 > 添加角色”。

1. 填写角色的名称，例如*mrrole*。
2. 编辑角色，在“权限”的表格中选择“Yarn > Scheduler Queue > root”，勾选“Submit”、“Admin”。
3. 在“权限”表格中选择“HBase > HBase Scope”，勾选global的“Create”、“Read”、“Write”、“Execute”。

4. 在“权限”的表格中选择“HDFS > File System > hdfs://hacluster/ ”，勾选“Read”、“Write”和“Execute”。
5. 在“权限”的表格中选择“Hive > Hive Read Write Privileges”，勾选default的“Create”、“Select”、“Delete”、“Insert”。
6. 单击“确定”保存。

步骤2 单击“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。

步骤3 填写用户名，例如*test*，用户类型为“机机”用户，加入用户组**supergroup**，设置其“主组”为**supergroup**，并绑定角色**mrrole**取得权限，单击“确定”。

步骤4 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名中选择test，然后在右侧“操作”列中选择“更多 >下载认证凭据”下载，保存后解压得到用户的user.keytab文件与krb5.conf文件，用于在样例工程中进行安全认证，如**5.2.6-准备kerberos认证**所示。

图 6-2 下载认证凭据

----结束

6.2.3 准备 Eclipse 与 JDK

选择Windows开发环境下，安装Eclipse，安装JDK。

步骤1 开发环境安装Eclipse程序，安装要求Eclipse使用4.2或以上版本。

步骤2 开发环境安装JDK程序，安装要求JDK使用1.8版本。

说明

- 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
- 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
- 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。

----结束

6.2.4 准备 Linux 客户端运行环境

MapReduce的运行环境可以部署在Linux环境下。您可以按照如下操作完成运行环境准备。

操作步骤

步骤1 确认服务端YARN组件和MapReduce组件已经安装，并正常运行。

步骤2 客户端运行环境已安装1.7或1.8版本的JDK。

步骤3 客户端机器的时间与Hadoop集群的时间要保持一致，时间差小于5分钟。

MRS集群的时间可通过登录主管理节点（集群管理IP地址所在节点）运行date命令查询。

步骤4 下载MapReduce客户端程序到客户端机器中。

1. 登录**MRS Manager**系统。

   在浏览器地址栏中输入访问地址，地址格式为“https://MRS Manager系统的WebService浮动IP地址:8080/web”。例如，在IE浏览器地址栏中，输入“https://10.10.10.172:8080/web”。

2. 选择“服务管理 > 下载客户端”，下载客户端程序到客户端机器。

步骤5 解压缩客户端文件包MRS_Services_Client.tar。安装包为tar格式，执行如下命令解压两次。

tar -xvf /opt/MRS_Services_Client.tar

tar -xvf /opt/MRS_Service_ClientConfig.tar

步骤6 为运行环境设置环境变量，假设安装包解压路径为“MRS_Services_ClientConfig/”。

进入解压文件夹，执行如下命令安装客户端。

sh *install.sh* {client_install_home}

步骤7 进入客户端安装目录，执行如下命令初始化环境变量。

source *bigdata_env*

步骤8 将**5.2.2-准备开发用户**中下载的user.keytab和krb5.conf文件拷贝到Linux环境的“/opt/conf”目录下，可参考**5.4.1-编译并运行程序**。

说明

在二次开发过程中，PRINCIPAL需要用到的用户名，应该填写为带域名的用户名，例如创建的用户为test，域名为HADOOP.COM，则其PRINCIPAL用户名则为test@HADOOP.COM，代码举例：

conf.set(PRINCIPAL, "test@HADOOP.COM");

步骤9 执行命令kinit -kt /opt/conf/user.keytab test。

说明

这里的user.keytab文件路径为Linux机器上配置文件的存放路径，后面的test用户名可以更改为**5.2.2-准备开发用户**中新建的用户名。

----结束

6.2.5 获取并导入样例工程

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。

操作步骤

步骤1 参照**样例工程获取地址**，下载样例工程到本地。

步骤2 导入样例工程到Eclipse开发环境。

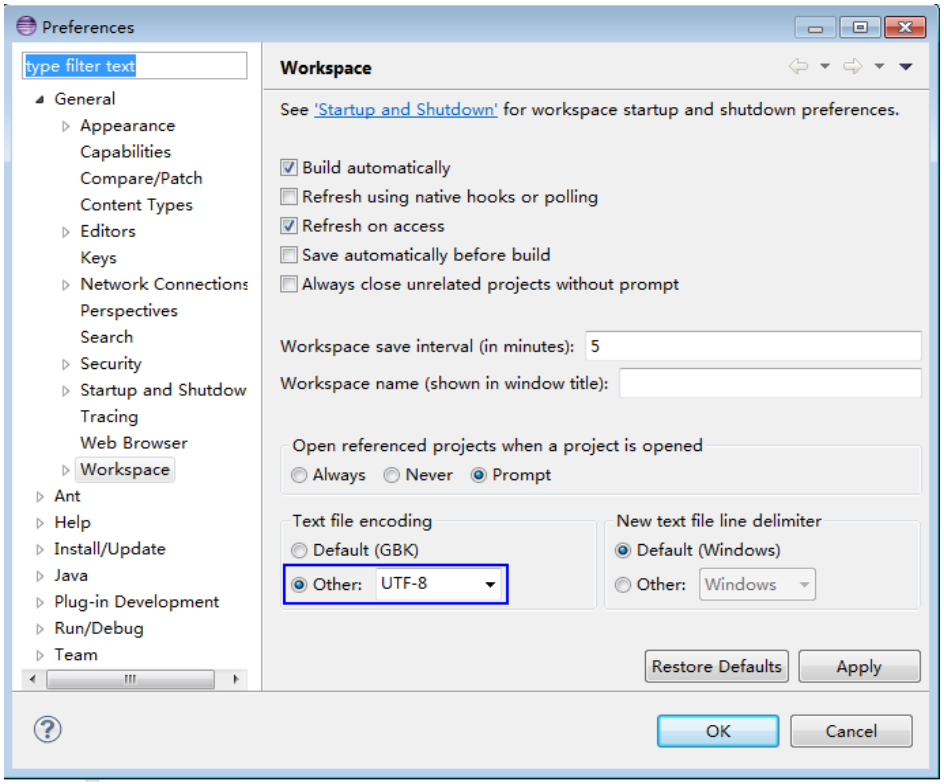
1. 打开Eclipse，选择“File > Import”。显示“Import”窗口，选择Existing Maven Projects，单击“next”按钮。
2. 在“Import Maven Projects”窗口单击“Browse”。显示“Select Root Folder”对话框。
3. 选择样例工程文件夹**mapreduce-examples**，单击“确定”按钮。
4. 在“Import Maven Projects”窗口单击“Finish”按钮。

步骤3 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。

   弹出“Preferences”窗口。
2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图6-3所示。

图 6-3 设置 Eclipse 的编码格式



----结束

6.2.6 准备 kerberos 认证

场景说明

在kerberos认证集群环境下，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。

安全认证代码

目前是统一调用LoginUtil类进行安全认证。

在MapReduce样例工程代码中，test@HADOOP.COM、user.keytab和krb5.conf为示例，实际操作时请联系管理员获取相应账号对应权限的keytab文件和krb5.conf文件，并将keytab文件和krb5.conf文件放入到样例代码中的conf目录，安全登录方法如下代码所示。

说明

认证信息需要根据实际环境修改。

```
public static final String PRINCIPAL= "test@HADOOP.COM";
public static final String KEYTAB =
FemaleInfoCollector.class.getClassLoader().getResource("user.keytab").getPath();
public static final String KRB =
FemaleInfoCollector.class.getClassLoader().getResource("krb5.conf").getPath();
// 判断是否为安全模式
    if("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))){
        //安全登录
        System.setProperty("java.security.krb5.conf", KRB);
        LoginUtil.login(PRINCIPAL, KEYTAB, KRB, conf);
    }
```

6.3 开发程序

6.3.1 MapReduce 统计样例程序

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. 本地新建两个文本文件，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两个小时的女性网民信息。

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);
  }
```

```
  // 判断是否为安全模式
  if("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))){
     //security mode
     System.setProperty("java.security.krb5.conf", KRB);
     LoginUtil.login(PRINCIPAL, KEYTAB, KRB, conf);
  }

  // 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端先合并一下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);
}

}
```

6.3.2 MapReduce 访问多组件样例程序

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。

功能介绍

该样例主要分为三个部分。

- 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 {

     String name = "";
     String line = value.toString();

     //加载配置文件
      conf = context.getConfiguration();

      setJaasInfo("krb5.conf", "jaas.conf");
      LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, "test", KEYTAB);
      LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);

      //准备hive query
      //加载parameter
      Properties clientInfo = null;
      InputStream fileInputStream = null;
      try {
          clientInfo = new Properties();
          File propertiesFile = new File(hiveClientProperties);
          fileInputStream = new FileInputStream(propertiesFile);
          clientInfo.load(fileInputStream);
      } catch (Exception e) {
      } finally {
          if (fileInputStream != null) {
              fileInputStream.close();
          }
      }

      String zkQuorum = clientInfo.getProperty("zk.quorum");
```

```
    String zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    String serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");

    // 创建Hive鉴权信息
    // Read this carefully:
    // MapReduce can only use Hive through JDBC.
    // Hive will submit another MapReduce Job to execute query.
    // So we run Hive in MapReduce is not recommended.
    final String driver = "org.apache.hive.jdbc.HiveDriver";

    String sql = "select name,sum(stayTime) as "
            + "stayTime from person where name = ? group by name";

    StringBuilder sBuilder = new StringBuilder("jdbc:hive2://").append(zkQuorum).append("/");
    // in map or reduce, use 'auth=delegationToken'
    sBuilder
            .append(";serviceDiscoveryMode=")
            .append(serviceDiscoveryMode)
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";auth=delegationToken;");

    String url = sBuilder.toString();

    try {
        Class.forName(driver);
        hiveConn = DriverManager.getConnection(url, "", "");
        statement = hiveConn.prepareStatement(sql);
    } catch (Exception e) {
        LOG.error("Init jdbc driver failed.", e);
    }

    //创建hbase连接
    try {
        // Create a HBase connection
        hbaseConn = ConnectionFactory.createConnection(conf);
        // get table
        table = hbaseConn.getTable(TableName.valueOf(HBASE_TABLE_NAME));
    } catch (IOException e) {
        LOG.error("Exception occur when connect to HBase", e);
        throw e;
    }

   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 {

     // 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 (hbaseConn != null) {
     try {
      hbaseConn.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 int readHive(String name) {

   ResultSet resultSet = null;
   try {
    statement.setString(1, name);
    resultSet = statement.executeQuery();

    if (resultSet.next()) {
     return resultSet.getInt("stayTime");
    }
   } 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 != hiveConn) {
     try {
      hiveConn.close();
     } catch (SQLException e) {
      // handle exception
     }
    }
   }

   return 0;
  }
```

样例4：类MultiComponentReducer定义Reducer抽象类的reduce方法。

```
  public void reduce(Text key, Iterable<Text> values, Context context) throws IOException,
InterruptedException {

    Text finalValue = new Text("");

    setJaasInfo("krb5.conf", "jaas.conf");
```



```
        LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, "test", KEYTAB);
        LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);

        conf = context.getConfiguration();
        try {
            // 创建hbase连接
            conn = ConnectionFactory.createConnection(conf);
            // 得到表
            table = conn.getTable(TableName.valueOf(HBASE_TABLE_NAME));
        } catch (IOException e) {
            LOG.error("Exception occur when connect to HBase", e);
            throw e;
        }

      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) {

      try {
        // 创建一个HBase Put请求实例
        List<Put> list = new ArrayList<Put>();
        byte[] row = Bytes.toBytes("1");
        Put put = new Put(row);
        byte[] family = Bytes.toBytes("cf");
        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 {

      // 清理所需目录
      MultiComponentExample.cleanupBeforeRun();

      // 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 thriftCLIClass = Class.forName("org.apache.hive.service.cli.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.thrift.HadoopThriftAuthBridge23");

    // 添加Hive运行依赖到Job
    JarFinderUtil
        .addDependencyJars(config, hiveDriverClass, thriftCLIClass, thriftClass, hiveConfClass, hiveTransClass,
            hiveMetaClass, hiveShimClass);

    //开启Kerberos认证的安全集群登录
    if("kerberos".equalsIgnoreCase(config.get("hadoop.security.authentication"))){
    //security mode
        LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, PRINCIPAL, KEYTAB);
        LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);
      System.setProperty("java.security.krb5.conf", KRB);
   LoginUtil.login(PRINCIPAL, KEYTAB, KRB, config);
   }
    // 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
    Properties clientInfo = null;
    InputStream fileInputStream = null;
    try {
        clientInfo = new Properties();
        File propertiesFile = new File(hiveClientProperties);
        fileInputStream = new FileInputStream(propertiesFile);
        clientInfo.load(fileInputStream);
    } catch (Exception e) {
    } finally {
        if (fileInputStream != null) {
            fileInputStream.close();
        }
    }
    String zkQuorum = clientInfo.getProperty("zk.quorum");//zookeeper节点ip和端口列表
    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");
    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(HiveAuthFactory.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。

6.4 调测程序

6.4.1 编译并运行程序

在程序代码完成开发后，可以在Linux环境中运行应用。

说明

MapReduce应用程序只支持在Linux环境下运行，不支持在Windows环境下运行。

操作步骤

步骤1 生成MapReduce应用可执行包。

执行mvn package生成jar包，在工程目录target目录下获取，比如“mapreduce-examples-1.0.jar”。

步骤2 上传生成的应用包“mapreduce-examples-1.0.jar”到Linux客户端上。例如“/opt”目录。

步骤3 如果集群开启Kerberos，参考5.2.2-准备开发用户获得的“user.keytab”、“krb5.conf”文件需要在Linux环境上创建文件夹保存这些配置文件，例如“/opt/conf”。并在linux环境上，在客户端路径下（/opt/client/HDFS/hadoop/etc/hadoop/）获得core-site.xml、hdfs-site.xml文件放入上述文件夹里。

步骤4 样例程序如果指定OBS为输入输出的目标文件系统（如obs://<BucketName>/input/），需要进行以下配置。

在$YARN_CONF_DIR/core-site.xml中添加AK配置项“fs.obs.access.key”和SK配置项“fs.obs.secret.key”，AK/SK可登录“OBS控制台”，进入“我的凭证”页面获取。

```
<property>
<name>fs.obs.access.key</name>
```

```
<value>xxxxxxxxxxxxxxxx</value>
</property>
<property>
<name>fs.obs.secret.key</name>
<value>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx</value>
</property>
```

步骤5 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。
  a. 如果集群开启kerberos，在Linux环境中添加样例工程运行所需的classpath，例如

     export YARN_USER_CLASSPATH=/opt/conf/

  b. 执行如下命令:

     ```
     cd /opt
     yarn jar mapreduce-examples-1.0.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 mapreduce-examples-1.0.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。
- 在执行**yarn jar mapreduce-examples-1.0.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。
- **mapreduce-examples-1.0.jar**适用于MRS 1.x版本。
- 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。
- 运行样例工程前需要根据实际环境修改认证信息。
- 针对开启Kerberos认证的安全集群，代码中的“principal”请根据实际环境修改。例如test@FAA12CC3_0996_432F_9D6F_E18F6F9D7F43.COM。

- 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”、“hive-site.xml”和“mapred-site.xml”文件，如果是安全模式集群，还需要同时获取“user.keytab”、“krb5.conf”，并在Linux环境上创建文件夹保存这些配置文件，例如“/opt/conf”。

     说明

     请联系管理员获取相应账号对应权限的“user.keytab”和“krb5.conf”文件，“hbase-site.xml”从HBase客户端获取，例如：/opt/client/HBase/hbase/conf，“hiveclient.properties”和“hive-site.xml”从Hive客户端获取，例如：/opt/client/Hive/config，“mapred-site.xml”文件从Yarn客户端获取，例如：/opt/client/Yarn/config。

  b. 对于安全模式集群，在新建的文件夹中创建文件“jaas_mr.conf”，文件内容如下。

     ```
     Client {
     com.sun.security.auth.module.Krb5LoginModule required
     useKeyTab=true
     keyTab="user.keytab"
     principal="test@FAA12CC3_0996_432F_9D6F_E18F6F9D7F43.COM"
     useTicketCache=false
     ```



```
storeKey=true
debug=true;
};
```

说明

- 文件内容中的test@HADOOP.COM为示例，实际操作时请做相应修改。
- “jaas_mr.conf”文件和代码中的“principal”请根据实际环境修改。例如test@FAA12CC3_0996_432F_9D6F_E18F6F9D7F43.COM。
- 未开启Kerberos认证集群略过此步骤。

c. 在Linux环境中添加样例工程运行所需的classpath，例如（以客户端安装路径为/opt/conf为例）

export YARN_USER_CLASSPATH=/opt/conf/:/opt/client/HBase/hbase/lib/*:/opt/client/Hive/Beeline/lib/*

说明

- 针对MRS 1.9.x版本集群，需要在执行上述命令前或者执行上述命令后执行**mv /opt/client/Hive/Beeline/lib/derby-10.10.2.0.jar derby-10.10.2.0.jar.bak**命令。
- 命令中使用的jar包请根据集群中对应路径下的实际版本修改。

d. 提交MapReduce任务，执行如下命令，运行样例工程。运行样例工程前需要根据实际环境修改认证信息。

yarn jar mapreduce-examples-1.0.jar com.huawei.bigdata.mapreduce.examples.MultiComponentExample

----结束

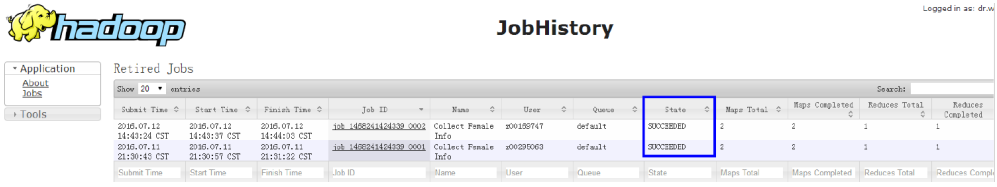
6.4.2 查看调测结果

MapReduce应用程序运行完成后，可以通过WebUI查看应用程序运行情况，也可以通过MapReduce日志获取应用运行情况。

- **通过MapReduce服务的WebUI进行查看**

  登录MRS Manager，单击“服务管理 > MapReduce > JobHistoryServer”进入Web界面后查看任务执行状态。

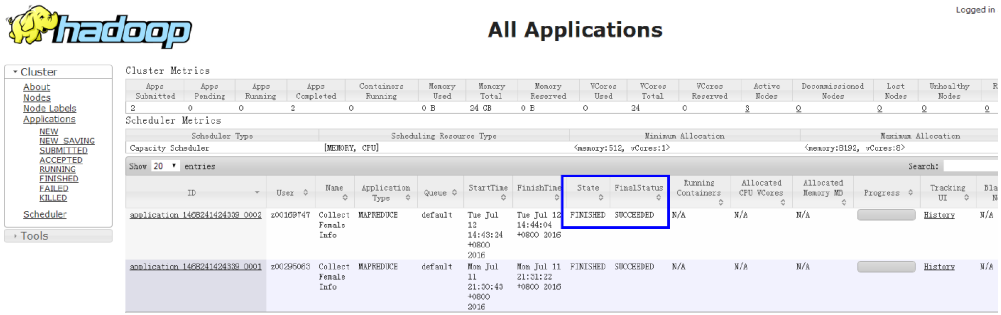
  图 6-4 JobHistory Web UI 界面



- **通过YARN服务的WebUI进行查看**

  登录MRS Manager，单击“服务管理 > Yarn > ResourceManager(主)”进入Web界面后查看任务执行状态。

图 6-5 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 mapreduce-example.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 mapreduce-example.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://10-120-180-170:26000/
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://10-120-180-170:26012/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。

6.5 MapReduce 接口

6.5.1 Java API

MapReduce 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 6-3 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。
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。

功能	说明
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 6-4 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。

6.6 FAQ

6.6.1 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：在公有云管理控制台，选择“基本信息 > Yarn监控信息”进入Web界面，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。

6.7 开发规范

6.7.1 规则

继承 Mapper 抽象类实现

在MapReduce任务的Map阶段，会执行map()及setup()方法。

正确示例：

```
public static class MapperClass extends

Mapper<Object, Text, Text, IntWritable> {
/**
* map的输入，key为原文件位置偏移量，value为原文件的一行字符数据。
* 其map的输入key，value为文件分割方法InputFormat提供，用户不设置，默认 * 使用TextInputFormat。
*/
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
//自定义的实现
}
/**
* 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
*/
public void setup(Context context) throws IOException,
InterruptedException {
//自定义的实现
}
}
```

继承 Reducer 抽象类实现。

在MapReduce任务的Reduce阶段，会执行reduce()及setup()方法。

正确示例：

```
public static class ReducerClass extends

Reducer<Text, IntWritable, Text, IntWritable> {
```



```
/**
* @param 输入为一个key和value值集合迭代器。
* 由各个map汇总相同的key而来。reduce方法汇总相同key的个数。
* 并调用context.write(key, value)输出到指定目录。
* 其reduce的输出的key，value由Outputformat写入文件系统。
* 默认使用TextOutputFormat写入HDFS。
*/

public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
//自定义实现
}

/**
* 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。
*/

public void setup(Context context) throws IOException,
InterruptedException {

// 自定义实现，Context可以获得配置信息。

}
}
```

提交一个 MapReduce 任务

main()方法创建一个job，指定参数，提交作业到hadoop集群。

正确示例：

```
public static void main(String[] args) throws Exception {
Configuration conf = getConfiguration();
// main方法输入参数：args[0]为样例MR作业输入路径，args[1]为样例MR作业输出路径
String[] otherArgs = new GenericOptionsParser(conf, args)
.getRemainingArgs();
if (otherArgs.length != 2) {
System.err.println("Usage: <in> <out>");
System.exit(2);
}
Job job = new Job(conf, "job name");
// 设置找到主任务所在的jar包。
job.setJar("D:\\job-examples.jar");
// job.setJarByClass(TestWordCount.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(TokenizerMapperV1.class);
job.setReducerClass(IntSumReducerV1.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(IntSumReducerV1.class);
// 设置作业的输出类型，也可以通过配置文件指定。
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
// 设置该job的输入输出路径，也可以通过配置文件指定。
Path outputPath = new Path(otherArgs[1]);
FileSystem fs = outputPath.getFileSystem(conf);
// 如果输出路径已存在，删除该路径。
if (fs.exists(outputPath)) {
fs.delete(outputPath, true);
}
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

对资源消耗较大的操作不要放到 map 或 reduce 函数中

对资源消耗较大的操作比如创建数据库链接，打开关闭文件等，不要放到map或reduce函数中。

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;
  }
```

6.7.2 建议

全局使用的配置项，在 mapred-site.xml 中指定

如下给出接口所对应的mapred-site.xml中的配置项:

setMapperClass(Class <extends Mapper> cls) ->“mapreduce.job.map.class”

setReducerClass(Class<extends Reducer> cls) ->“mapreduce.job.reduce.class”

setCombinerClass(Class<extends Reducer> cls) ->“mapreduce.job.combine.class”

setInputFormatClass(Class<extends InputFormat> cls) ->
“mapreduce.job.inputformat.class”

setJar(String jar) ->“mapreduce.job.jar”

setOutputFormat(Class< extends OutputFormat> theClass) ->
“mapred.output.format.class”

setOutputKeyClass(Class<> theClass) ->“mapreduce.job.output.key.class”

setOutputValueClass(Class<> theClass) ->“mapreduce.job.output.value.class”

setPartitionerClass(Class<extends Partitioner> theClass) ->
“mapred.partitioner.class”

setMapOutputCompressorClass(Class<extends CompressionCodec> codecClass)

->“mapreduce.map.output.compress”&
“mapreduce.map.output.compress.codec”

setJobPriority(JobPriority prio) ->“mapreduce.job.priority”

setQueueName(String queueName) ->“mapreduce.job.queuename”

setNumMapTasks(int n) ->“mapreduce.job.maps”

setNumReduceTasks(int n) ->“mapreduce.job.reduces”

6.7.3 示例

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。

1. 从原文件中筛选女性网民上网时间数据信息，通过类MapperClass继承Mapper抽象类实现。
2. 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ReducerClass继承Reducer抽象类实现。
3. main方法提供建立一个MapReduce job，并提交MapReduce作业到hadoop集群

样例1：类MapperClass定义Mapper抽象类的map()方法和setup()方法。

```
public static class MapperClass extends

Mapper<Object, Text, Text, IntWritable> {
// 分隔符。
String delim;
// 性别筛选。
String sexFilter;
private final static IntWritable timeInfo = new IntWritable(1);
private Text nameInfo = new Text();
/**
* map的输入，key为原文件位置偏移量，value为原文件的一行字符数据。
* 其map的输入key，value为文件分割方法InputFormat提供，用户不设置，默认使用TextInputFormat。
*/
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
// 读取的一行字符串数据
String line = value.toString();
if (line.contains(sexFilter)) {
// 获取姓名
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),
line.length());
timeInfo.set(Integer.parseInt(time));
// map输出key，value键值对
context.write(nameInfo, timeInfo);
}
}
```

```
/**
* 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
*/
public void setup(Context context) throws IOException,
InterruptedException {
// 通过Context可以获得配置信息。
sexFilter = delim + context.getConfiguration().get("log.sex.filter", "female") + delim;
}
}
```

样例2：类ReducerClass定义Reducer抽象类的reduce()方法。

```
public static class ReducerClass extends
Reducer<Text, IntWritable, Text, IntWritable> {
private IntWritable result = new IntWritable();
// 总时间门槛。
private int timeThreshold;
/**
* @param 输入为一个key和value值集合迭代器。
* 由各个map汇总相同的key而来。reduce方法汇总相同key的个数。
* 并调用context.write(key, value)输出到指定目录。
* 其reduce的输出的key，value由Outputformat写入文件系统。
* 默认使用TextOutputFormat写入HDFS。
*/
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。
*/
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 {
Configuration conf = getConfiguration();
// main方法输入参数：args[0]为样例MR作业输入路径，args[1]为样例MR作业输出路径
String[] otherArgs = new GenericOptionsParser(conf, args)
.getRemainingArgs();
if (otherArgs.length != 2) {
System.err.println("Usage: <in> <out>");
System.exit(2);
}
Job job = new Job(conf, "Collect Female Info");
// 设置找到主任务所在的jar包。
job.setJar("D:\\mapreduce-examples\\hadoop-mapreduce-examples\\mapreduce-examples.jar");
// job.setJarByClass(TestWordCount.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(TokenizerMapperV1.class);
job.setReducerClass(IntSumReducerV1.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(IntSumReducerV1.class);
```

```
// 设置作业的输出类型，也可以通过配置文件指定。
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
// 设置该job的输入输出路径，也可以通过配置文件指定。
Path outputPath = new Path(otherArgs[1]);
FileSystem fs = outputPath.getFileSystem(conf);
// 如果输出路径已存在，删除该路径。
if (fs.exists(outputPath)) {
fs.delete(outputPath, true);
}
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true)    0 : 1);
}
```

7 HDFS 应用开发

7.1 概述

7.1.1 HDFS 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，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**Java API**。

7.1.2 常用概念

DataNode

将文件切分成大小相同的块（称为“数据块”），存储在不同的DataNode上，并且周期性地向NameNode报告该DataNode的数据存放情况。

NameNode

用于管理文件系统的命名空间、目录结构、元数据信息以及提供备份机制等。

- Active NameNode：主NameNode，管理文件系统的命名空间、维护文件系统的目录结构树以及元数据信息；记录写入的每个“数据块”与其归属文件的对应关系。

- Standby NameNode：备NameNode，与主NameNode中的数据保持同步；随时准备在主NameNode出现异常时接管其服务。

Journalnode

高可用性（High availability，HA）集群下，用于同步主备NameNode之间的元数据信息。

ZKFC

ZKFC是需要和NameNode一一对应的服务，即每个NameNode都需要部署ZKFC。它负责监控NameNode的状态，并及时把状态写入Zookeeper。ZKFC有选择哪个NameNode作为主NameNode的权利。

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，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。

- 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**Java API** 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**C API** 的描述进行应用开发。

- Shell

  提供shell命令 **Shell命令介绍** 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 **HTTP REST API** 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MRS Hadoop组件中进行API方式认证。

7.1.3 开发流程

开发流程中各阶段的说明如**图7-1**和**表7-1**所示。

图 7-1 HDFS 应用程序开发流程

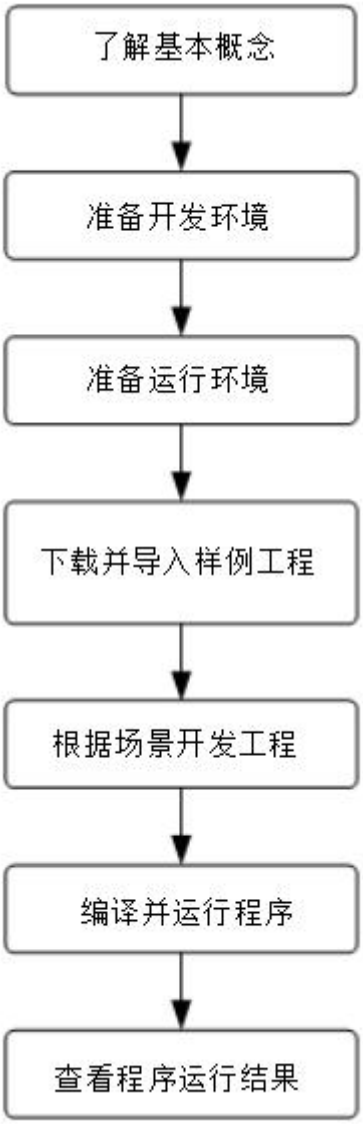


表 7-1 HDFS 应用开发的流程说明

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**
准备开发环境	使用Eclipse工具，请根据指导完成开发环境配置。	**准备Eclipse与JDK**
准备运行环境	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Linux客户端运行环境**
下载并导入样例工程	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。	**获取并导入样例工程**

阶段	说明	参考文档
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。	场景及开发思路
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。	Linux：安装客户端时编译并运行程序
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。	Linux：查看调测结果

7.2 环境准备

7.2.1 开发环境简介

在进行应用开发时，要准备的开发环境如表7-2所示。

表 7-2 开发环境

准备项	说明
Eclipse	开发环境的基本配置。版本要求：4.2或以上。
JDK	JDK使用1.7或者1.8版本。 **说明** 基于安全考虑，MRS集群服务端只支持TLS 1.1和TLS 1.2加密协议，IBM JDK默认TLS只支持1.0，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TLS1.0/1.1/1.2。 详情请参见：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。

7.2.2 准备开发用户

前提条件

MRS服务集群开启了Kerberos认证，没有开启Kerberos认证的集群忽略该步骤。

操作场景

开发用户用于运行样例工程。用户需要有HDFS权限，才能运行HDFS样例工程。

操作步骤

步骤1 登录MRS Manager，在MRS Manager界面选择“系统设置 > 角色管理 > 添加角色”。

1. 填写角色的名称，例如*hdfsrole*。
2. 编辑角色，在“权限”的表格中选择“HDFS > File System > hdfs://hacluster/”，勾选“Read”、“Write”和“Execute”，单击“确定”保存。

步骤2 单击“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。

步骤3 填写用户名，例如*hdfsuser*，用户类型为“机机”用户，加入用户组**supergroup**，设置其“主组”为**supergroup**，并绑定角色**hdfsrole**取得权限，单击“确定”。

步骤4 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名中选择**hdfsuser**，然后在右侧“操作”列中选择“更多 >下载认证凭据”下载，保存后解压得到用户的user.keytab文件与krb5.conf文件，用于在样例工程中进行安全认证，如图7-2所示。

图 7-2 下载认证凭据

----结束

7.2.3 准备 Eclipse 与 JDK

前提条件

MRS服务集群开启了Kerberos认证

操作场景

在Windows环境下需要安装Eclipse和JDK。

操作步骤

步骤1 开发环境安装Eclipse程序，版本要求Eclipse使用4.2或以上版本。

步骤2 开发环境安装JDK程序，版本要求JDK使用1.7或者1.8版本。

说明

- 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
- 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
- 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。

----结束

7.2.4 准备 Linux 客户端运行环境

前提条件

1. 确认服务端HDFS组件已经安装，并正常运行。
2. 客户端运行环境已安装1.7或1.8版本的JDK
3. 获取客户端安装包MRS_Services_Client.tar

操作场景

在Linux上安装客户端。

操作步骤

步骤1 客户端机器的时间与Hadoop集群的时间要保持一致(手动修改客户端机器或者集群的时间)，时间差小于5分钟。

MRS集群的时间可通过登录主管理节点（集群管理IP地址所在节点）运行date命令查询。

步骤2 下载MapReduce客户端程序到客户端机器中。

1. 登录**MRS Manager**系统。
2. 选择“服务管理 > 下载客户端”，下载客户端程序到客户端机器。

步骤3 解压缩客户端文件包MRS_Services_Client.tar。安装包为tar格式，执行如下命令解压两次。

tar -xvf MRS_Services_Client.tar

tar -xvf MRS_Service_ClientConfig.tar

步骤4 为运行环境设置环境变量，假设安装包解压路径为“MRS_Services_ClientConfig/”。

进入解压文件夹，执行如下命令安装客户端。

sh *install.sh* {client_install_home}

步骤5 进入客户端安装目录，执行如下命令初始化环境变量。

source *bigdata_env*

步骤6 从服务端拷贝如下文件至jar包（样例代码导出的jar包可参**安装客户端时编译并运行程序**）同目录的conf目录下。

表 7-3 配置文件

文件名称	作用	获取地址
core-site.xml	配置HDFS详细参数。	${HADOOP_HOME}/etc/hadoop/core-site.xml
hdfs-site.xml	配置HDFS详细参数。	${HADOOP_HOME}/etc/hadoop/hdfs-site.xml

文件名称	作用	获取地址
user.keytab	对于Kerberos安全认证提供HDFS用户信息。	如果是安全模式集群，您可以联系管理员获取相应账号对应权限的keytab文件和krb5文件。
krb5.conf	Kerberos server配置信息。	

说明

- 表7-3中${HADOOP_HOME}表示服务端Hadoop的安装目录。
- keytab认证是24小时有效，超过24小时需要重新认证。
- 样例代码中PRNCIPAL_NAME的用户名要与获取keytab文件和krb5文件的账户名一致。
- 不同集群的user.keytab、krb5.conf不能共用。
- 注意样例代码中，“System.getProperty("user.dir") + File.separator + "conf" + File.separator + "user.keytab"”处使用的keytab文件需与用户的keytab一致。
- conf目录下的log4j.properties文件客户根据自己的需要进行配置。

----结束

7.2.5 获取并导入样例工程

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。

操作步骤

步骤1 参照**样例工程获取地址**，下载样例工程到本地。

步骤2 导入样例工程到Eclipse开发环境。

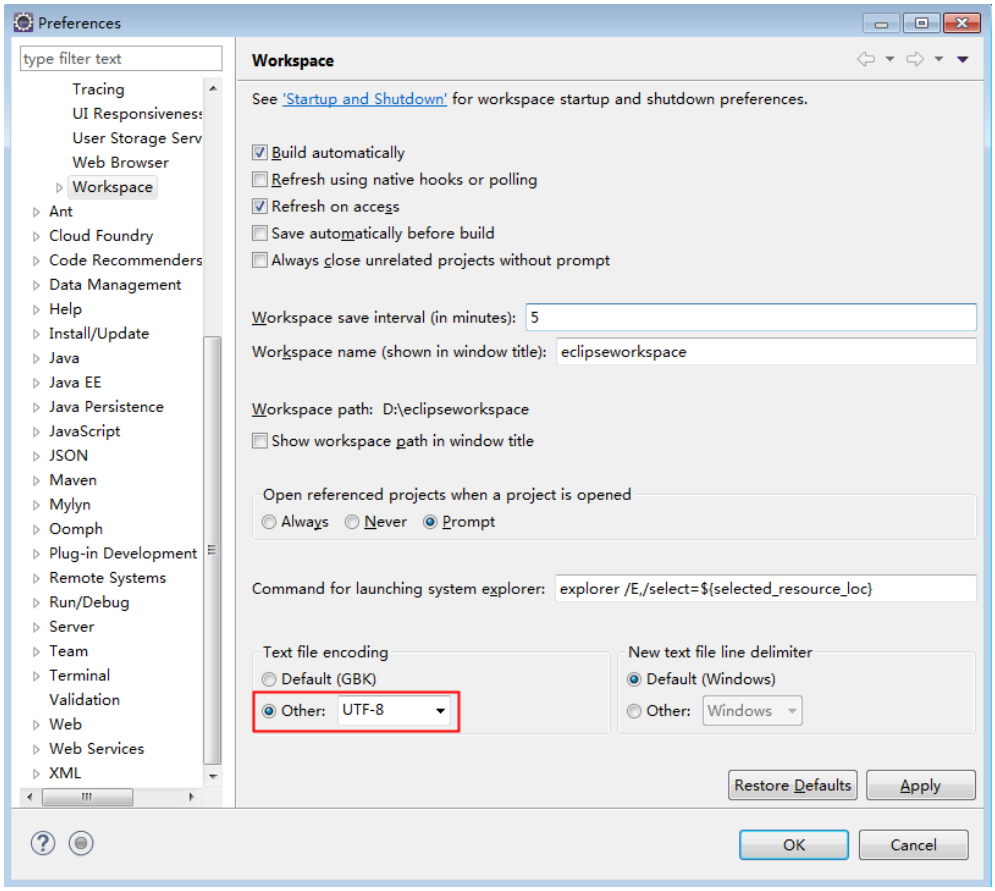
1. 第一种方法：打开Eclipse，选择“File > New > Java Project”。
2. 去掉对“Use default location”的勾选，单击“Browse”。
   显示“浏览文件夹”对话框。
3. 选择样例工程文件夹hdfs-examples，单击“确定”。
4. 在“New Java Project”窗口单击“Finish”。
5. 第二种方法：打开Eclipse，依次选择“File > Import… > Existing maven Projects into Workspace > Next”，在下一个页面单击“Browse”，显示“浏览文件夹”对话框。
6. 选择样例工程文件夹“hdfs-examples”，单击“确定”。
7. 在“Import”窗口单击“Finish”。

步骤3 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。
   弹出“Preferences”窗口。

2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图7-3所示。

图 7-3 设置 Eclipse 的编码格式



----结束

7.3 开发程序

7.3.1 场景及开发思路

场景说明

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

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 **HDFS初始化**

2. 写文件 **写文件**
3. 追加文件内容 **追加文件内容**
4. 读文件 **读文件**
5. 删除文件 **删除文件**
6. Colocation **Colocation**
7. 设置存储策略 **设置存储策略**
8. 访问OBS **访问OBS**

开发思路

根据前述场景说明进行功能分解，以上传一个新员工的信息为例，对该员工的信息进行查询、追加、删除等，可分为以下七部分：

1. 通过kerberos认证。
2. 调用fileSystem中的mkdir接口创建目录。
3. 调用HdfsWriter的dowrite接口写入信息。
4. 调用fileSystem中的open接口读取文件。
5. 调用HdfsWriter的doAppend接口追加信息。
6. 调用fileSystem中的deleteOnExit接口删除文件。
7. 调用fileSystem中的delete接口删除文件夹。

7.3.2 HDFS 初始化

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。
3. 使用HDFS的API。

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7-4**所示的配置文件。这些文件均已导入到“hdfs-example”工程的“conf”目录。

表 7-4 配置文件

文件名称	作用	获取地址
core-site.xml	配置HDFS详细参数。	MRS_Services_ClientConfig\HDFS\config\core-site.xml
hdfs-site.xml	配置HDFS详细参数。	MRS_Services_ClientConfig\HDFS\config\hdfs-site.xml

文件名称	作用	获取地址
user.keytab	对于Kerberos安全认证提供HDFS用户信息。	如果是安全模式集群，您可以联系管理员获取相应账号对应权限的keytab文件和krb5文件。
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中的HdfsMain类。

在Linux客户端运行应用的初始化代码，代码样例如下所示。

```
/**
 * 初始化，获取一个FileSystem实例
 *
 * @throws IOException
 */
private void init() throws IOException {
  confLoad();
  authentication();
  instanceBuild();
}

/**
 *
 * 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 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));
 }

/**
 * kerberos security authentication
 * 如果程序运行在Linux上，则需要krb5.conf和keytab文件的路径，
 * 修改为在Linux下客户端文件的绝对路径。并且需要将样例代码中的keytab文件和principal文件
 * 分别修改为当前用户的keytab文件名和用户名。
 *
 */
private void authentication() throws IOException {
  // 安全模式
  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);
  }
}


/**
 * build HDFS instance
 */
```

```
private void instanceBuild() throws IOException {
  // get filesystem
  fSystem = FileSystem.get(conf);
}
```

在Linux环境下需要运行login的代码样例，用于第一次登录使用，详细代码请参考com.huawei.bigdata.security中的LoginUtil类。

```
public synchronized static void login(String userPrincipal,
    String userKeytabPath, String krb5ConfPath, Configuration conf)
    throws IOException {
  // 1.检查放入的参数
  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.检查是否需要登录
  if (checkNeedLogin(userPrincipal)) {

  // 5.登录hadoop并检查
    loginHadoop(userPrincipal, userKeytabFile.getAbsolutePath());
```

```
    }

    // 6.检查重新登录
    checkAuthenticateOverKrb();
    System.out.println("Login success!!!!!!!!!!!!!!");
  }
```

7.3.3 写文件

功能简介

写文件过程为：

1. 实例化一个FileSystem。
2. 由此FileSystem实例获取写文件的各类资源。
3. 将待写内容写入到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HdfsMain类和HdfsWriter类。

```
/**
 * 创建文件，写文件
 *
 * @throws IOException
 * @throws ParameterException
 */
private void write() throws IOException, ParameterException {
    final String content = "hi, I am bigdata. It is successful if you can see me.";
    InputStream in = (InputStream) new ByteArrayInputStream(
content.getBytes());
    try {
      HdfsWriter writer = new HdfsWriter(fSystem, DEST_PATH
          + File.separator + FILE_NAME);
      writer.doWrite(in);
      System.out.println("success to write.");
    } finally {
      //务必要关闭流资源
      close(in);
    }
}
```

7.3.4 追加文件内容

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. 实例化一个FileSystem。
2. 由此FileSystem实例获取各类相关资源。
3. 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HdfsMain类和HdfsWriter类。

```
/**
* 追加文件内容
*
* @throws IOException
*/
private void append() throws Exception {
    final String content = "I append this content.";
    InputStream in = (InputStream) new ByteArrayInputStream(
    content.getBytes());
  try {
    HdfsWriter writer = new HdfsWriter(fSystem, DEST_PATH
      + File.separator + FILE_NAME);
    writer.doAppend(in);
   System.out.println("success to append.");
   } finally {
//务必要关闭流资源.
   close(in);
   }
}
```

7.3.5 读文件

功能简介

获取HDFS上某个指定文件的内容。

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HdfsMain类。

```
 /**
* 读文件
*
* @throws 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;

    // 写文件
    while ((sTempOneLine = reader.readLine()) != null) {
      strBuffer.append(sTempOneLine);
    }

    System.out.println("result is : " + strBuffer.toString());
    System.out.println("success to read.");
```

```
  } finally {
    //务必要关闭资源.
    close(reader);
    close(in);
  }
}
```

7.3.6 删除文件

功能简介

删除HDFS上某个指定文件或者文件夹。

说明

被删除的文件或文件夹，会被放在当前用户目录下的.Trash/Current文件夹中。若发生误删除，可从该文件夹中恢复。

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HdfsMain类。

```
/**
 * 删除文件
 *
 * @throws IOException
 */
private void delete() throws IOException {
  Path beDeletedPath = new Path(DEST_PATH + File.separator + FILE_NAME);
  fSystem.deleteOnExit(beDeletedPath);
  System.out.println("succee to delete the file " + DEST_PATH
      + File.separator + FILE_NAME);
}
```

7.3.7 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**

  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7-5所示。

表 7-5 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7-6所示。

表 7-6 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。	-

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。
- 如果需要对colocation上传的文件做balance操作，为避免colocation失效，可以通过MRS Manager界面中的oi.dfs.colocation.file.pattern参数进行设置，设置该参数值为对应数据文件块的路径，多个路径之间以逗号分开。例如/test1，/test2。

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。

1. 初始化

   使用Colocation前需要进行kerberos安全认证。

```
private static void init() throws IOException {
    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。

```
public static void main(String[] args) throws IOException {
    init();
    dfsAdmin = new DFSColocationAdmin(conf);
    dfs = new DFSColocationClient();
    dfs.initialize(URI.create(conf.get("fs.defaultFS")), conf);
    createGroup();
    put();
    delete();
    deleteGroup();
    dfs.close();
    dfsAdmin.close();
  }
```

3. 创建group

   样例：创建一个gid01组，组中包含3个locator。

```
  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。

```
  private static void put() throws IOException {
    FSDataOutputStream out = dfs.create(new Path("/testfile.txt"), true,
        COLOCATION_GROUP_GROUP01, "lid01");
    // 待写入到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。

```
public static void delete() throws IOException {
  dfs.delete(new Path("/testfile.txt"));
}
```

6. 删除group

   样例：删除gid01。

```
private static void deleteGroup() throws IOException {
  dfsAdmin.deleteColocationGroup(COLOCATION_GROUP_GROUP01);
}
```

7.3.8 设置存储策略

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. 在“${HADOOP_HOME}/etc/hadoop/”下的“Hdfs-site.xml”中设置如下参数。

```
<name>dfs.storage.policy.enabled</name>
<value>true</value>
```

2. 重启HDFS，如图7-4 所示。

   图 7-4 重启 hdfs

3. 登录MRSManager，选择“服务管理 > HDFS > 服务配置”，将“参数类别”设置为“全部配置”。
4. 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“保存配置”，重启HDFS。
5. 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HdfsMain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;
```

```
      }
      System.out.println("StoragePolicy:" + bs.getName());
    }
    if (!flag) {
      policyName = storage[0].getName();
    }
    dfs.setStoragePolicy(destPath, policyName);
    System.out.println("success to set Storage Policy path " + DEST_PATH);
    rmdir();
  }
  else{
    System.out.println("Storage Policy is only supported in HDFS.");
  }
}
```

7.3.9 访问 OBS

功能简介

访问OBS过程为：

1. 设置“fs.obs.access.key”和“fs.obs.secret.key”。
2. 由此FileSystem实例可以读取、新增和删除各类资源。

说明

不支持追加操作。

前提条件

对接OBS前需要提前在OBS服务中创建相关目录，并确保访问用户具有对应目录的访问操作权限。

代码样例

如下是实例化FileSystem的代码片段，详细代码请参考com.huawei.bigdata.hdfs.examples中的HdfsMain类。

```
/**
  *
  * Add configuration file if the application run on the linux ,then need make
  * the path of the core-site.xml and hdfs-site.xml to in the linux client file
  *
  */
 private 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.set("fs.obs.access.key","*** Provide your Access Key ***");
   conf.set("fs.obs.secret.key","*** Provide your Secret Key ***");
   }

 /**
  * build HDFS instance
  */
 private void instanceBuild() throws IOException {
   // get filesystem
// fSystem = FileSystem.get(conf);
   fSystem =  FileSystem.get(URI.create("obs://[BuketName]"),conf);
 }
```

7.4 调测程序

7.4.1 在 Linux 中调测程序

7.4.1.1 安装客户端时编译并运行程序

操作场景

HDFS应用程序支持在安装HDFS客户端的Linux环境中运行。在程序代码完成开发后，可以上传Jar包至Linux客户端环境中运行应用。

说明

HDFS应用程序只支持在Linux环境下运行，不支持在Windows环境下运行。

前提条件

- 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。

操作步骤

步骤1 执行mvn package生成jar包，在工程目录target目录下获取，比如:hdfs-examples-1.0.jar。

步骤2 将导出的Jar包拷贝上传至Linux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“user.keytab” 和 "krb5.conf"拷贝至“conf”目录。可参考**步骤6** 。

步骤3 配置环境变量。

source /opt/client/bigdata_env

步骤4 执行如下命令，运行Jar包。

hadoop jar hdfs-examples-1.0.jar com.huawei.bigdata.hdfs.examples.HdfsMain

说明

运行命令时，需保持客户端“Yarn/config/hdfs-site.xml”中的kerberos相关信息和“HDFS/hadoop/etc/hadoop/hdfs-site.xml”中的kerberos相关信息一致。“hdfs-site.xml”中kerberos的配置mapred改为hdfs，需要修改的地方如**图7-5**所示。

图 7-5 hdfs-site.xml

```
<name>dfs.datanode.kerberos.https.principal</name>
<value>mapred/hadoop.hadoop.com@HADOOP.COM</value>
<name>dfs.namenode.kerberos.https.principal</name>
<value>mapred/hadoop.hadoop.com@HADOOP.COM</value>
<name>dfs.datanode.kerberos.principal</name>
<value>mapred/hadoop.hadoop.com@HADOOP.COM</value>
<name>dfs.namenode.kerberos.principal</name>
<value>mapred/hadoop.hadoop.com@HADOOP.COM</value>
```

----结束

7.4.1.2 查看调测结果

操作场景

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。

操作步骤

1. **查看运行结果获取应用运行情况**
   - HdfsMain Linux样例程序安全集群运行结果如下所示：

     ```
     [root@node-master1dekG client]# hadoop jar hdfs-examples-1.0.jar
     com.huawei.bigdata.hdfs.examples.HdfsMain
     WARNING: Use "yarn jar" to launch YARN applications.
     20/03/25 16:29:45 INFO security.UserGroupInformation: Login successful for user hdfsuser using
     keytab file user.keytab
     20/03/25 16:29:45 INFO security.LoginUtil: Login success!!!!!!!!!!!!!!
     success to create path /user/hdfs-examples
     success to write.
     result is : hi, I am bigdata. It is successful if you can see me.
     success to read.
     success to delete the file /user/hdfs-examples/test.txt
     success to delete path /user/hdfs-examples
     success to create path /user/hdfs-examples
     StoragePolicy:FROZEN
     StoragePolicy:COLD
     StoragePolicy:WARM
     StoragePolicy:HOT
     StoragePolicy:ONE_SSD
     StoragePolicy:ALL_SSD
     StoragePolicy:LAZY_PERSIST
     succee to set Storage Policy path /user/hdfs-examples
     success to delete path /user/hdfs-examples
     ```

   - HdfsMain Linux样例程序普通集群运行结果如下所示：

     ```
     [root@node-master2VknR client]# hadoop jar hdfs-examples-1.0.jar
     com.huawei.bigdata.hdfs.examples.HdfsMain
     WARNING: Use "yarn jar" to launch YARN applications.
     success to create path /user/hdfs-examplessuccess to write.
     result is : hi, I am bigdata. It is successful if you can see me.
     success to read.
     success to delete the file /user/hdfs-examples/test.txt
     success to delete path /user/hdfs-example
     ssuccess to create path /user/hdfs-examples
     StoragePolicy:FROZEN
     StoragePolicy:COLD
     StoragePolicy:WARM
     StoragePolicy:HOT
     ```

```
StoragePolicy:ONE_SSD
StoragePolicy:ALL_SSD
StoragePolicy:LAZY_PERSIST
succee to set Storage Policy path /user/hdfs-examples
success to delete path /user/hdfs-examples
```

2. **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。

7.5 HDFS 接口

7.5.1 Java API

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/r2.7.2/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7-7**。
- FileStatus：记录文件和目录的状态信息。常用接口参见**表7-8**。
- DFSColocationAdmin：管理colocation组信息的接口。常用接口参见**表7-9**。
- DFSColocationClient：操作colocation文件的接口。常用接口参见**表7-10**。

  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 7-7 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。

接口	说明
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，recursive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。
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 7-8 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 7-9 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 7-10 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 7-11 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 7-12 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。

7.5.2 C API

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/r2.7.2/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HDFS的C样例代码hdfs_test.c“MRS_Services_ClientConfig/HDFS/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 = "/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 = "/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. 拷贝文件。

```
const char* destfile = "/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 = "/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 = "/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. 变量目录。

```
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，安装方法可参考《MapReduce服务用户指南》的“客户端管理”章节。

1. 确认服务端HDFS组件已经安装，并正常运行。
2. 客户端运行环境已安装1.7或1.8版本的JDK。
3. 获取并解压缩安装“MRS_HDFS_Client.tar”包。执行如下命令解压。

   tar -xvf MRS_HDFS_Client.tar

   tar -xvf MRS_HDFS_ClientConfig.tar

   说明

   由于不兼容老版本客户端，建议用户获取与服务端集群相同版本的客户端安装包进行安装部署。

4. 进入解压文件夹，即“MRS_HDFS_ClientConfig”，执行下列命令安装客户端。

   sh *install.sh* /opt/client

   其中“/opt/client”为用户自定义路径，此处仅为举例。

5. 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*

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。

   kinit *hdfs*

   说明

   kinit一次票据时效24小时。24小时后再次运行样例，需要重新执行**kinit**命令。

3. 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*

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! : /nativeTest
   hdfsOpenFile- SUCCESS! for write : /nativeTest/testfile.txt
   hdfsWrite- SUCCESS! : /nativeTest/testfile.txt
   Flushing file data ....
   hdfsCloseFile- SUCCESS! : /nativeTest/testfile.txt
   hdfsOpenFile- SUCCESS! for read : /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 fi|²* rite!
   hdfsCopy- SUCCESS!, File copied, src : /nativeTest/testfile.txt, des : /nativeTest/testfile1.txt
   hdfsMove- SUCCESS!, File moved, src : /nativeTest/testfile1.txt, des : /nativeTest/testfile2.txt
   hdfsRename- SUCCESS!, File renamed, Old name : /nativeTest/testfile2.txt, New name : /nativeTest/
   testfile3.txt
   hdfsDelete- SUCCESS!, File deleted : /nativeTest/testfile3.txt
   hdfsSetReplication- SUCCESS!, Set replication 10 for /nativeTest/testfile.txt
   hdfsChown- SUCCESS!, Chown success for /nativeTest/testfile.txt
   hdfsChmod- SUCCESS!, Chmod success for /nativeTest/testfile.txt
   hdfsUtime- SUCCESS!, Set time success for /nativeTest/testfile.txt
   Name: hdfs://hacluster/nativeTest/testfile.txt, Type: F, Replication: 10, BlockSize: 134217728, Size: 31,
   LastMod: 1480589792, Owner: root, Group: root, Permissions: 511 (rwxrwxrwx)
   hdfsGetPathInfo - SUCCESS!
   Name: hdfs://hacluster/nativeTest/testfile.txt, Type: F, Replication: 10, BlockSize: 134217728, Size: 31,
   LastMod: 1480589792, Owner: root, Group: root, Permissions: 511 (rwxrwxrwx)
   hdfsListDirectory- SUCCESS!, /nativeTest
   ```



```
hdfsTruncateFile- SUCCESS!, /nativeTest/testfile.txt
Block Size : 134217728
hdfsGetDefaultBlockSize- SUCCESS!
Block Size : 134217728 for file /nativeTest/testfile.txt
hdfsGetDefaultBlockSizeAtPath- SUCCESS!
HDFS Capacity : 1569475438758
hdfsGetCapacity- SUCCESS!
HDFS Used : 1122248
hdfsGetCapacity- SUCCESS!
hdfsExists- SUCCESS! /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 : 31,
totalShortCircuitBytesRead : 0, totalZeroCopyBytesRead : 0
[root@10-120-85-2 hdfs-c-example]# make run
./hdfs_test
hdfsConnect- SUCCESS!
hdfsCreateDirectory- SUCCESS! : /nativeTest
hdfsOpenFile- SUCCESS! for write : /nativeTest/testfile.txt
hdfsWrite- SUCCESS! : /nativeTest/testfile.txt
Flushing file data ....
hdfsCloseFile- SUCCESS! : /nativeTest/testfile.txt
hdfsOpenFile- SUCCESS! for read : /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 : /nativeTest/testfile.txt, des : /nativeTest/testfile1.txt
hdfsMove- SUCCESS!, File moved, src : /nativeTest/testfile1.txt, des : /nativeTest/testfile2.txt
hdfsRename- SUCCESS!, File renamed, Old name : /nativeTest/testfile2.txt, New name : /nativeTest/
testfile3.txt
hdfsDelete- SUCCESS!, File deleted : /nativeTest/testfile3.txt
hdfsSetReplication- SUCCESS!, Set replication 10 for /nativeTest/testfile.txt
hdfsChown- SUCCESS!, Chown success for /nativeTest/testfile.txt
hdfsChmod- SUCCESS!, Chmod success for /nativeTest/testfile.txt
hdfsUtime- SUCCESS!, Set time success for /nativeTest/testfile.txt

Name: hdfs://hacluster/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/nativeTest/testfile.txt, Type: F, Replication: 10, BlockSize: 134217728, Size: 31,
LastMod: 1500345260, Owner: root, Group: root, Permissions: 511 (rwxrwxrwx)
hdfsListDirectory- SUCCESS!, /nativeTest
hdfsTruncateFile- SUCCESS!, /nativeTest/testfile.txt
Block Size : 134217728
hdfsGetDefaultBlockSize- SUCCESS!
Block Size : 134217728 for file /nativeTest/testfile.txt
hdfsGetDefaultBlockSizeAtPath- SUCCESS!
HDFS Capacity : 102726873909
hdfsGetCapacity- SUCCESS!
HDFS Used : 4767076324
hdfsGetCapacity- SUCCESS!
hdfsExists- SUCCESS! /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

   执行该命令之前需要安装GDB，安装步骤可参考**安装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/hadoop-client/HDFS/hadoop/hdfs-c-example/hdfs_test...done.
(gdb)
[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)
```

安装 GDB

1. 下载GDB的依赖包termcap的源代码。

   wget https://ftp.gnu.org/gnu/termcap/termcap-1.3.1.tar.gz
2. 解压termcap源码。

   tar -zxvf termcap-1.3.1.tar.gz
3. 编译安装termcap。

   cd termcap-1.3.1/

   ./configure && make && make install
4. 下载GDB源码。

   cd ~

   wget https://ftp.gnu.org/gnu/gdb/gdb-7.6.1.tar.gz
5. 解压GDB源码

   tar -zxvf gdb-7.6.1.tar.gz
6. 编译安装GDB。

   cd gdb-7.6.1/

   ./configure && make && make install
7. 查看GDB是否安装成功。

   gdb --version

   打印出gdb版本信息即为安装成功。

7.5.3 HTTP REST API

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/r2.7.2/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，可安装方法可参考《MapReduce服务用户指南》的“客户端管理”章节。

1. 确认服务端HDFS组件已经安装，并正常运行。
2. 客户端运行环境已安装1.7或1.8版本的JDK。
3. 获取并解压缩安装包“MRS_HDFS_Client.tar”。执行如下命令解压。

   tar -xvf MRS_HDFS_Client.tar

   tar -xvf MRS_HDFS_ClientConfig.tar

   说明

   由于不兼容老版本客户端，建议用户获取与服务端集群相同版本的客户端安装包进行安装部署。

4. 进入解压文件夹，即“MRS_HDFS_ClientConfig”，执行下列命令安装客户端。

   sh *install.sh* /opt/client

   其中“/opt/client”为用户自定义路径，此处仅为举例。

5. 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*

6. 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（普通集群请跳过此步操作）。

   kinit *hdfs*

   说明

   kinit一次的时效24小时。24小时后再次运行样例，需要重新执行**kinit**。

7. 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 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(仅安全集群支持)，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MRS Manager页面，单击“服务管理 > HDFS > 服务配置”，在“参数类别”选择“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“保存配置”，并勾选“重新启动受影响的服务或实例。”，单击“是”，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MRS Manager页面，单击“服务管理 > HDFS > 服务配置”，在“参数类别”选择“全部配置”，在“搜索”框里搜索“dfs.http.policy”，然后勾选“HTTP_AND_HTTPS”，单击“保存配置”，并勾选“重新启动受影响的服务或实例。”，单击“是”，重启HDFS服务。

----结束

操作步骤

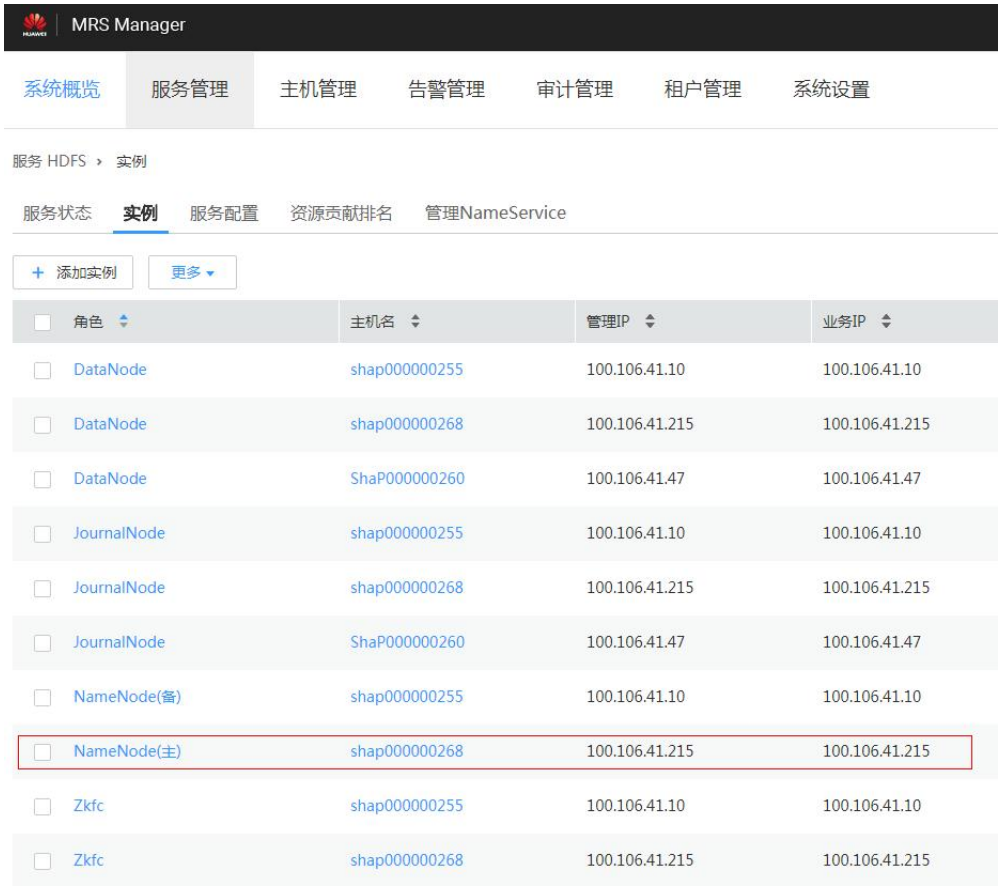
步骤1 登录MRS Manager，单击“服务管理”，选择“HDFS”，单击进入HDFS服务状态页面。

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”，进入图7-6界面，找到“NameNode(hacluster,主)”的主机名（host）和对应的IP。

图 7-6 HDFS 实例

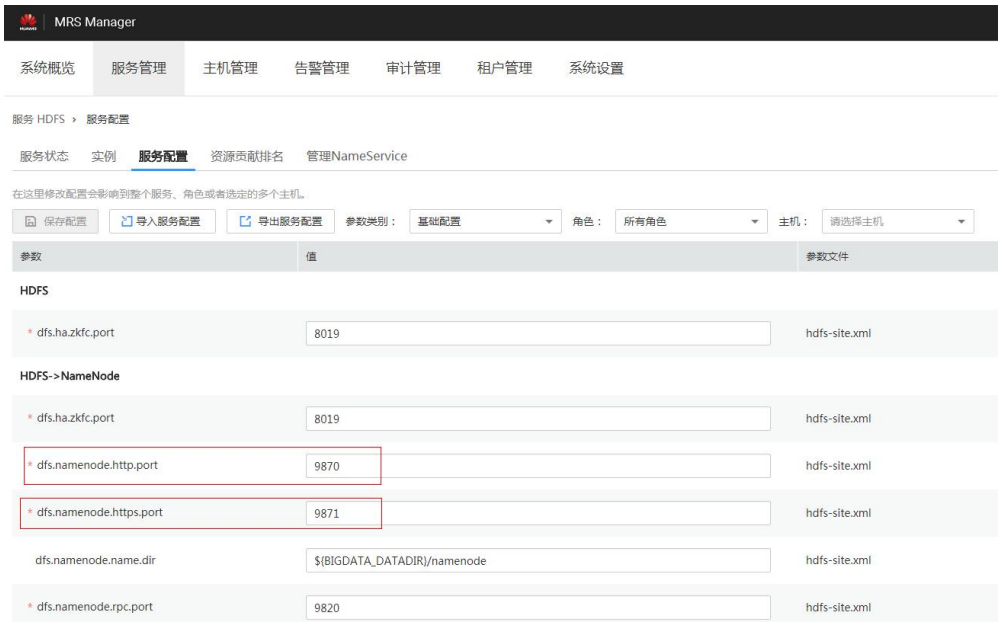


2. 单击“服务配置”，进入图7-7界面，找到“namenode.http.port”（9870）和“namenode.https.port”（9871）。

说明

MRS 1.9.2以下版本，上述端口默认值分别为25002和25003，详见MRS用户指南，组件端口信息章节。

图 7-7 HDFS 服务配置

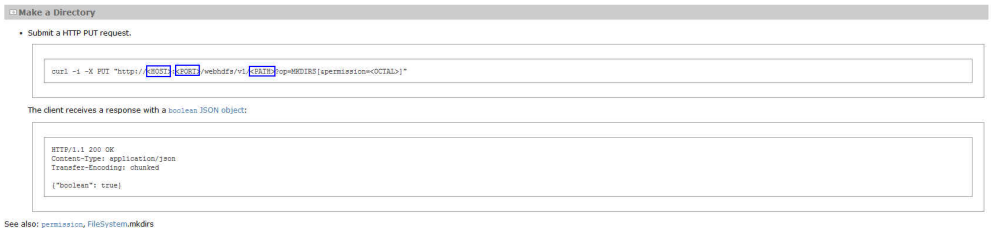


步骤2 参考如下链接，创建目录。

http://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/WebHDFS.html#Make_a_Directory

单击链接，如图7-8所示。

图 7-8 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7-8**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>都可以，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linux1表示<HOST>，9870表示<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@HADOOP.COM&t=kerberos&e=1462453809395&s=wiRF4rdTWpm
     3tDST+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://linux1:9871/webhdfs/v1/huawei?op=MKDIRS"
     ```

     其中linux1表示<HOST>，9871表示<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@HADOOP.COM&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"
```

其中linux1表示<HOST>，9870表示<PORT>。

- 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@HADOOP.COM&t=kerberos&e=1462464588403&s=qry87vAyYzSn9VsS6
Rm6vKLhKeU="; Path=/; Expires=Thu, 05-May-2016 16:09:48 GMT; HttpOnly
Location: http://linux1:25010/webhdfs/v1/huawei/testHdfs?
op=CREATE&delegation=HgAFYWRtaW4FYWRtaW4AigFUf4lZdIoBVKOV3XQOCBSyXvFAp92alcRs4j-
KNulnN6wUoBJXRUJIREZTIGRlbGVnYXRpb24UMTAuMTIwLjE3Mi4xMDk6MjUwMDA&namenoderpcadd
ress=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"
```

其中linux1表示<HOST>，9871表示<PORT>。

- 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@HADOOP.COM&t=kerberos&e=1462455978166&s=F4rXUwEevHZze3PR
8TxkzcV7RQQ="; 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&namenoderpcadd
ress=hacluster&overwrite=false"
```

其中linux1表示<HOST>，9864表示<PORT>。

- 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"
  ```
  其中linux1表示<HOST>，9870表示<PORT>。
- 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"
  ```
  其中linux1表示<HOST>，9871表示<PORT>。
- 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"
  ```
  其中linux1表示<HOST>，9870表示<PORT>。
- 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@HADOOP.COM&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&namenoderpcaddress=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"
  ```
  其中linux1表示<HOST>，9871表示<PORT>。
- 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@HADOOP.COM&t=kerberos&e=1462461641713&s=tGwwOH9scmnNtxP
jlnu28SFtex0="; 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"
  ```

  其中linux1表示<HOST>，9864表示<PORT>。

- 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"
  ```

  其中linux1表示<HOST>，9865表示<PORT>。

- 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"
  ```

  其中linux1表示<HOST>，9870表示<PORT>。

- 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"
```
其中linux1表示<HOST>，9871表示<PORT>。
- 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"
```
其中linux1表示<HOST>，9870表示<PORT>。
- 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"
```
其中linux1表示<HOST>，9871表示<PORT>。
- 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"
```
其中linux1表示<HOST>，9870表示<PORT>。
- 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"
```
其中linux1表示<HOST>，9871表示<PORT>。
- 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"
```

其中linux1表示<HOST>，9870表示<PORT>。

- 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@HADOOP.COM&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"
```

其中linux1表示<HOST>，9871表示<PORT>。

- 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@HADOOP.COM&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/r2.7.2/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。

7.5.4 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：

步骤1 初始化环境变量。

source /opt/client/bigdata_env

步骤2 如果当前集群已启用Kerberos认证，执行以下命令认证当前用户。如果当前集群未启用Kerberos认证，则无需执行此命令。当前用户为**准备开发用户**时增加的开发用户。

人机用户：*kinit MRS集群用户*

例如：**kinit hdfsuser**

机机用户：*kinit -kt 认证凭据路径 MRS集群用户*

例如：**kinit -kt /opt/user.keytab hdfsuser**

步骤3 执行HDFS SHELL命令。

HDFS SHELL 操作格式：hadoop fs <args>

直接输入命令即可。例如：

- 查看对应目录下内容 **hadoop fs -ls /tmp/input/**
- 创建目录 **hadoop fs -mkdir /tmp/input/new_dir**
- 查看文件内容 **hadoop fs -cat /tmp/input/file1**
- 清除文件 **hadoop fs -rm -r /tmp/input/file1**
- 执行如下命令查询HDFS命令的帮助**hadoop fs --help**

HDFS命令行参考请参见官网：

https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html

----结束

表 7-13 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。

7.6 开发规范

7.6.1 规则

HDFS NameNode 元数据存储路径

NameNode元数据信息的默认存储路径为“${BIGDATA_DATA_HOME}/namenode/data”，该参数用于确定HDFS文件系统的元数据信息的保存路径。

HDFS 需要开启 NameNode 镜像备份

NameNode的镜像备份参数为“fs.namenode.image.backup.enable”，需要设置该值为“true”，系统即可定期备份NameNode的数据。

HDFS 需要开启 DataNode 数据存储路径

DataNode默认存储路径配置为：${BIGDATA_DATA_HOME}/hadoop/dataN/dn/datadir（N≥1），N为数据存放的目录个数。

例如：${BIGDATA_DATA_HOME}/hadoop/data1/dn/datadir、${BIGDATA_DATA_HOME}/hadoop/data2/dn/datadir

设置后，数据会存储到节点上每个挂载磁盘的对应目录下面。

HDFS 提高读取写入性能方式

写入数据流程：HDFS Client收到业务数据后，从NameNode获取到数据块编号、位置信息后，联系DataNode，并将需要写入数据的DataNode建立起流水线，完成后，客户端再通过自有协议写入数据到Datanode1，再有DataNode1复制到DataNode2、DataNode3（三备份）。写完的数据，将返回确认信息给HDFS Client。

1. 合理设置块大小，如设置dfs.blocksize为 268435456（即256MB）。
2. 对于一些不可能重用的大数据，缓存在操作系统的缓存区是无用的。可将以下两参数设置为false。

   dfs.datanode.drop.cache.behind.reads和dfs.datanode.drop.cache.behind.writes

MapReduce 中间文件存放路径

MapReduce默认中间文件夹存放路径只有一个，${hadoop.tmp.dir}/mapred/local，建议修改为每个磁盘下均可存放中间文件。

例如：/hadoop/hdfs/data1/mapred/local、/hadoop/hdfs/data2/mapred/local、/hadoop/hdfs/data3/mapred/local等，不存在的目录会自动忽略。

JAVA 开发时，申请资源须在 finally 释放

申请的HDFS资源需要在try/finally中释放，而不能只在try语句之外释放，否则会导致异常情况下的资源泄漏。

HDFS 文件操作 API 概述

Hadoop中关于文件操作类基本上全部是在"org.apache.hadoop.fs"包中，这些API能够支持的操作包含：打开文件，读写文件，删除文件等。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，只能通过来类的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：

```
static FileSystem get(Configuration conf);
```

该类封装了几乎所有的文件操作，例如mkdir，delete等。综上基本可以得出操作文件的程序库框架：

```
operator()
{
   得到Configuration对象
   得到FileSystem对象
   进行文件操作
}
```

HDFS 初始化方法

HDFS初始化是指在使用HDFS提供的API之前，需要做的必要工作。

大致过程为：加载HDFS服务配置文件，并进行Kerberos安全认证，认证通过后再实例化Filesystem，之后使用HDFS的API。此处Kerberos安全认证需要使用到的keytab文件，请提前准备，可参考**准备开发用户**。

正确示例：

```
private  void init() throws IOException {
Configuration conf = new Configuration();
// 读取配置文件
conf.addResource("user-hdfs.xml");
// 安全模式下，先进行安全认证
if ("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
String PRINCIPAL = "username.client.kerberos.principal";
String KEYTAB = "username.client.keytab.file";
// 设置keytab密钥文件
conf.set(KEYTAB, System.getProperty("user.dir") + File.separator + "conf"
?????+ File.separator + conf.get(KEYTAB));

// 设置kerberos配置文件路径 */
String krbfilepath = System.getProperty("user.dir") + File.separator + "conf"
?????+ File.separator + "krb5.conf";
System.setProperty("java.security.krb5.conf", krbfilepath);
// 进行登录认证 */
SecurityUtil.login(conf, KEYTAB, PRINCIPAL);
}

// 实例化文件系统对象
fSystem = FileSystem.get(conf);
}
```

HDFS 上传本地文件

通过FileSystem.copyFromLocalFile（Path src，Patch dst）可将本地文件上传到HDFS的制定位置上，其中src和dst均为文件的完整路径。

正确示例：

```
public class CopyFile {
        public static void main(String[] args) throws Exception {
         Configuration conf=new Configuration();
         FileSystem hdfs=FileSystem.get(conf);
              //本地文件
        Path src =new Path("D:\\HebutWinOS");
        //HDFS为止
         Path dst =new Path("/");
              hdfs.copyFromLocalFile(src, dst);
         System.out.println("Upload to"+conf.get("fs.default.name"));
            FileStatus files[]=hdfs.listStatus(dst);
            for(FileStatus file:files){
                System.out.println(file.getPath());
            }
        }
}
```

HDFS 创建目录

通过"FileSystem.mkdirs（Path f）"可在HDFS上创建目录，其中f为目录的完整路径。

正确示例：

```
public class CreateDir {
    public static void main(String[] args) throws Exception{
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
              Path dfs=new Path("/TestDir");
              hdfs.mkdirs(dfs);
    }
}
```

查看 HDFS 文件的最后修改时间

通过FileSystem.getModificationTime()可查看指定HDFS文件的修改时间。

正确示例：

```
    public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path fpath =new Path("/user/hadoop/test/file1.txt");
        FileStatus fileStatus=hdfs.getFileStatus(fpath);
        long modiTime=fileStatus.getModificationTime();
        System.out.println("file1.txt的修改时间是"+modiTime); }
```

读取 HDFS 某个目录下的所有文件

通过FileStatus.getPath( )可查看指定HDFS中某个目录下所有文件。

正确示例：

```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);

        Path listf =new Path("/user/hadoop/test");

        FileStatus stats[]=hdfs.listStatus(listf);
        for(int i = 0; i < stats.length; ++i)                                                   {
       System.out.println(stats[i].getPath().toString())
       }
        hdfs.close();
   }
```

查找某个文件在 HDFS 集群的位置

通过FileSystem.getFileBlockLocation（FileStatus file，long start，long len）可查找指定文件在HDFS集群上的位置，其中file为文件的完整路径，start和len来标识查找文件的路径。

正确示例：

```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path fpath=new Path("/user/hadoop/cygwin");

        FileStatus filestatus = hdfs.getFileStatus(fpath);
        BlockLocation[] blkLocations = hdfs.getFileBlockLocations(filestatus, 0, filestatus.getLen());

        int blockLen = blkLocations.length;
        for(int i=0;i < blockLen；i++){
        String[] hosts = blkLocations[i].getHosts();
        System.out.println("block_"+i+"_location:"+hosts[0]);
        }
   }
```

获取 HDFS 集群上所有节点名称信息

通过DatanodeInfo.getHostName( )可获取HDFS集群上的所有节点名称。

正确示例：

```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem fs=FileSystem.get(conf);
```

```
    DistributedFileSystem hdfs = (DistributedFileSystem)fs;
    DatanodeInfo[] dataNodeStats = hdfs.getDataNodeStats();

    for(int i=0;i < dataNodeStats.length;i++){
        System.out.println("DataNode_"+i+"_Name:"+dataNodeStats[i].getHostName());
    }
  }
```

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;
   }
```

7.6.2 建议

HDFS 的读写文件注意点

HDFS不支持随机读和写。

HDFS追加文件内容只能在文件末尾添加，不能随机添加。

只有存储在HDFS文件系统中的数据才支持append，edit.log以及数据元文件不支持Append。Append追加文件时，需要将《hdfs-site.xml》中的“dfs.support.append”参数值设置为true。

说明

dfs.support.append参数在开源社区版本中默认值是关闭，在MRS版本默认值是开启。

该参数为服务器端参数。建议开启，开启后才能使用Append功能。

不适用HDFS场景可以考虑使用其他方式来存储数据，如HBase。

HDFS 不适用于存储大量小文件

HDFS不适用于存储大量的小文件，因为大量小文件的元数据会占用NameNode的大量内存。

HDFS 中数据的备份数量 3 份即可

DataNode数据备份数量3份即可，增加备份数量不能提升系统效率，只会提升系统数据的安全系数；在某个节点损坏时，该节点上的数据会被均衡到其他节点上。

8 Spark 应用开发

8.1 概述

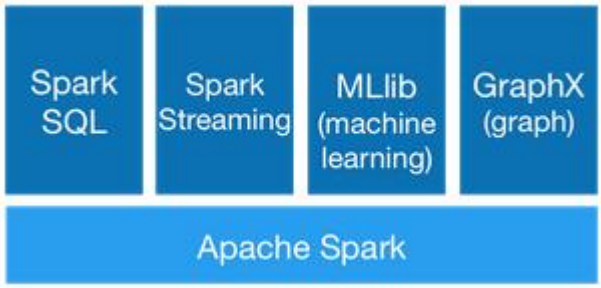
8.1.1 Spark 应用开发简介

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。
- Apache Spark部件架构如图8-1所示。本文档重点介绍Spark、Spark SQL和Spark Streaming应用开发指导。MLlib和GraghX的详细指导请参见Spark官方网站：http://spark.apache.org/docs/2.2.2/。

图 8-1 Spark 架构



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8-1所示。

表 8-1 Spark API 接口

接口	说明
Scala API	提供Scala语言的API。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。
Python API	提供Python语言的API。

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ThriftServer两种方式访问。其中ThriftServer的连接方式也有Beeline和JDBC客户端代码两种。

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。

8.1.2 常用概念

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。

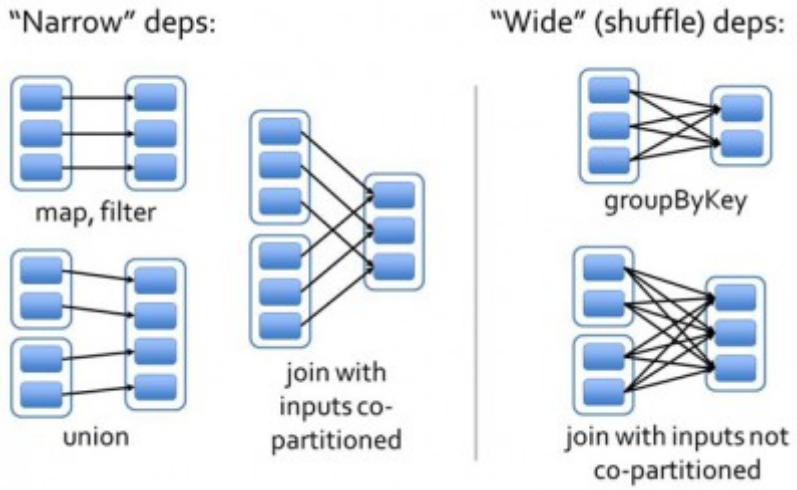
  RDD的存储：

  - 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 8-2 RDD 的依赖



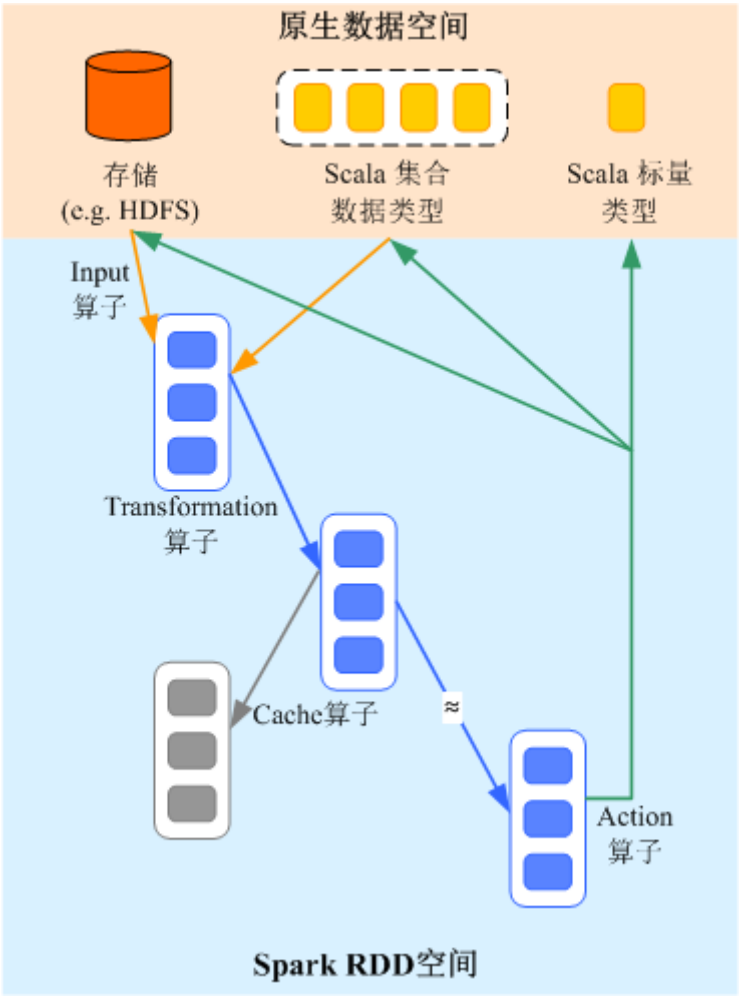
- **窄依赖：**指父RDD的每一个分区最多被一个子RDD的分区所用。
- **宽依赖：**指子RDD的分区依赖于父RDD的所有分区。

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8-3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 8-3 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：

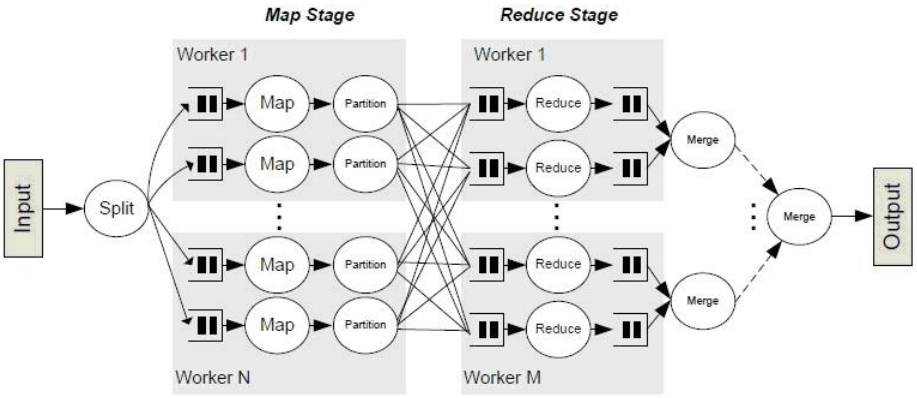
- 生成标量，如count（返回RDD中元素的个数）、reduce、fold/aggregate（返回几个标量）、take（返回前几个元素）。
- 生成Scala集合类型，如collect（把RDD中的所有元素倒入Scala集合类型）、lookup（查找对应key的所有值）。
- 写入存储，如与前文textFile对应的saveAsTextFile。
- 还有一个检查点算子checkpoint。当Lineage特别长时（这在图计算中时常发生），出错时重新执行整个序列要很长时间，可以主动调用checkpoint把当前数据写入稳定存储，作为检查点。

- **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 8-4 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为例讲一下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/2.2.2/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开发、配置和调优作业。

Spark SQL 常用概念

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RDD。

Spark SQL的程序入口是SQLContext类（或其子类），创建SQLContext时需要一个SparkContext对象作为其构造参数。SQLContext其中一个子类是HiveContext，相较于其父类，HiveContext添加了HiveQL的parser、UDF以及读取存量Hive数据的功能等。但注意，HiveContext并不依赖运行时的Hive，只是依赖Hive的类库。

由SQLContext及其子类可以方便的创建SparkSQL中的基本数据集DataFrame，DataFrame向上提供多种多样的编程接口，向下兼容多种不同的数据源，例如Parquet、JSON、Hive数据、Database、HBase等，这些数据源都可以使用统一的语法来读取。

Spark Streaming 常用概念

Dstream

DStream(又称Discretized Stream)是Spark Streaming提供的抽象概念。

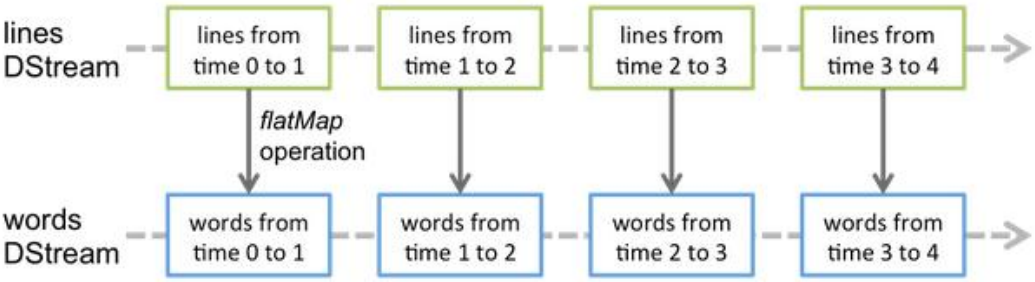
DStream表示一个连续的数据流，是从数据源获取或者通过输入流转换生成的数据流。从本质上说，一个DStream表示一系列连续的RDD。RDD一个只读的、可分区的分布式数据集。

DStream中的每个RDD包含了一个区间的数据。如图8-5所示。

图 8-5 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8-6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 8-6 DStream 算子转译



8.1.3 开发流程

Spark包含Spark Core、Spark SQL和Spark Streaming三个组件，其应用开发流程相同。

开发流程中各阶段的说明如图8-7和表8-2所示。

图 8-7 Spark 应用程序开发流程

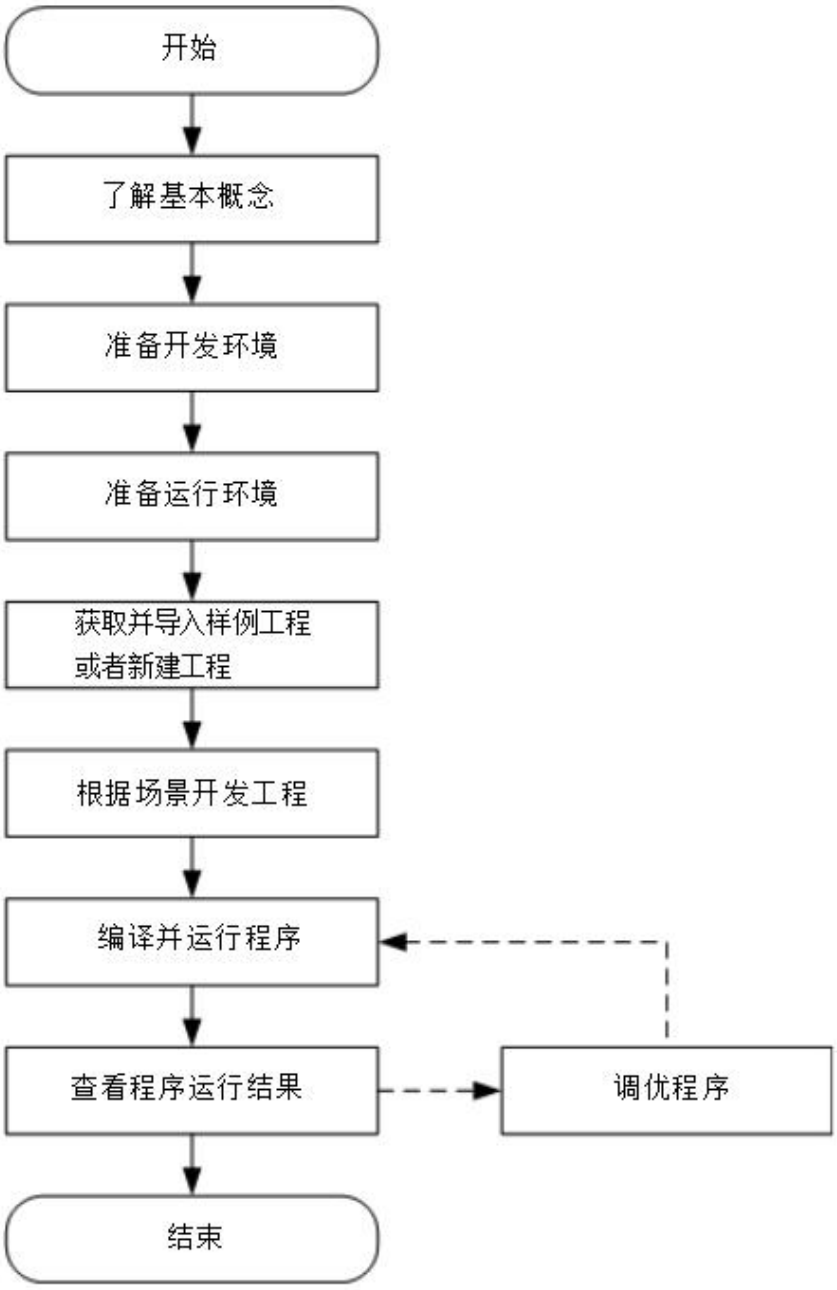


表 8-2 Spark 应用开发的流程说明

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Spark的基本概念，根据实际场景选择需要了解的概念，分为Spark Core基本概念、Spark SQL基本概念和Spark Streaming基本概念。	常用概念

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。	请参考**准备Java开发环境**至**准备Python开发环境**章节
准备运行环境	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。	**准备运行环境**
获取并导入样例工程 或者新建工程	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。	**下载并导入样例工程**
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。	请参考**场景说明**至**Scala样例代码**章节
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。	**编包并运行程序**
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诉求。 调优完成后，请重新进行编译和运行	请参考**数据序列化**至**Spark CBO调优**章节

8.2 环境准备

8.2.1 环境简介

在进行应用开发时，要准备的开发环境如**表8-3**所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行正常。

表 8-3 开发环境

准备项	说明
安装JDK	开发环境的基本配置。版本要求：1.7或者1.8。 **说明** 基于安全考虑，MRS 服务端只支持TLS 1.1和TLS 1.2加密协议，IBM JDK默认TLS只支持1.0，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TLS1.0/1.1/1.2。 详情请参见**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IDEA	用于开发Spark应用程序的工具。版本要求：13.1.6及以上版本。
安装Scala	Scala开发环境的基本配置。版本要求：2.11.0及以上版本。
安装Scala插件	Scala开发环境的基本配置。版本要求：0.35.683及以上版本。
安装Python	Python开发环境的基本配置。版本要求：Python2.7.x及以上版本。

8.2.2 准备开发用户

前提条件

MRS服务集群开启了Kerberos认证，没有开启Kerberos认证的集群忽略该步骤。

操作场景

开发用户用于运行样例工程。用户需要有HDFS、YARN和Hive权限，才能运行Spark样例工程。

操作步骤

步骤1 登录MRS Manager，请参考**登录MRS Manager**。

步骤2 在MRS Manager界面选择“系统设置 > 角色管理 > 添加角色”。

1. 填写角色的名称，例如sparkrole。
2. 在“权限”表格中选择“HBase > HBase Scope > global”，勾选default的“Create”。
3. 在“权限”的表格中选择“HBase > HBase Scope > global > hbase”，勾选hbase:meta的“Execute”。
4. 修改角色，在“权限”的表格中选择“HDFS > File System ”，勾选“Read”、“Write”和“Execute”。
5. 在“权限”的表格中选择“HDFS > File System > hdfs://hacluster/ > user >hive”，勾选“Execute”。
6. 在“权限”的表格中选择“HDFS > File System > hdfs://hacluster/ >user >hive > warehouse”，勾选“Read”、“Write”和“Execute”。
7. 在“权限”的表格中选择“Hive > Hive Read Write Privileges”，勾选default的“Create”。
8. 在“权限”的表格中选择“Yarn > Scheduler Queue > root”，勾选default的“Submit”。
9. 单击“确定”保存。

步骤3 在MRS Manager界面选择“系统设置>用户管理>添加用户”，为样例工程创建一个用户。填写用户名例如sparkuser，用户类型为“机机”用户，加入用户组**supergroup**和**kafkaadmin**，设置其“主组”为**supergroup**，并绑定角色sparkrole取得权限，单击“确定”。

说明

Spark Streaming程序使用的用户需要加kafkaadmin组权限，用来操作Kafka组件。

步骤4 在MRS Manager界面选择“系统设置>用户管理”，在用户名中选择sparkuser，单击操作中下载认证凭据文件，保存后解压得到用户的keytab文件与krb5.conf文件。用于在样例工程中进行安全认证，具体使用请参考准备认证机制代码。

----结束

8.2.3 准备 Java 开发环境

操作场景

Java开发环境可以搭建在Windows环境下，而运行环境（即客户端）只能部署在Linux环境下。

操作步骤

步骤1 对于Java开发环境，推荐使用IDEA工具，安装要求如下。

- JDK使用1.7版本（或1.8版本）
- IntelliJ IDEA（版本：13.1.6）

说明

- Spark不支持当客户端程序使用IBM JDK 1.7运行时，使用yarn-client模式向服务端提交Spark任务。
- Oracle JDK需进行安全加固，具体操作如下。

1. 到Oracle官方网站获取与JDK版本对应的JCE（Java Cryptography Extension）文件。JCE文件解压后包含“local_policy.jar”和“US_export_policy.jar”。拷贝jar包到如下路径。

   Linux：JDK安装目录/jre/lib/security

   Windows：JDK安装目录\jre\lib\security

2. 将“客户端安装目录/JDK/jdk/jre/lib/ext/”目录下“SMS4JA.jar”拷贝到如下路径。

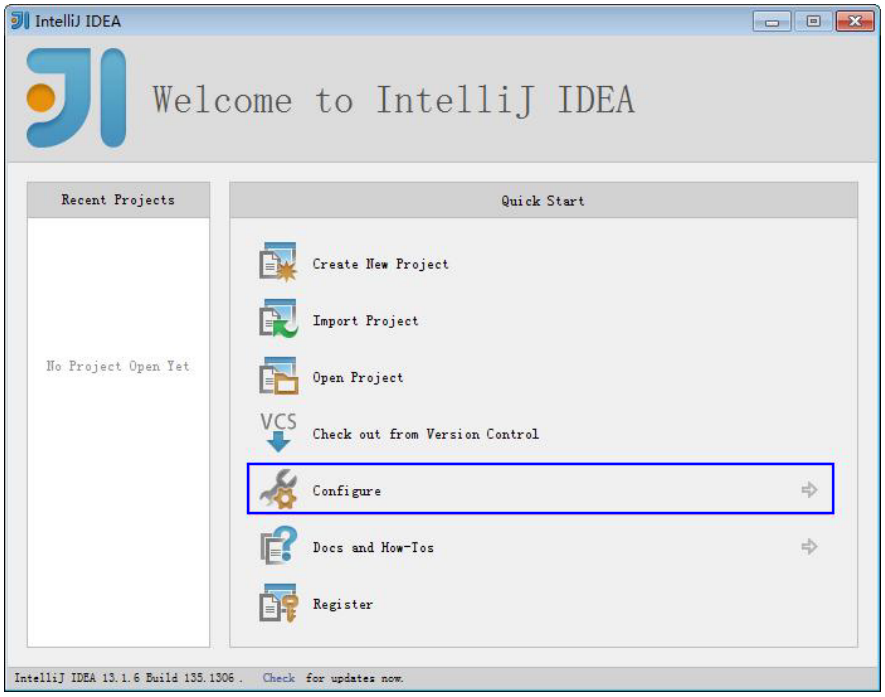
   Linux：JDK安装目录/jre/lib/ext/

   Windows：JDK安装目录\jre\lib\ext\

步骤2 安装IntelliJ IDEA和JDK工具，并进行相应的配置。

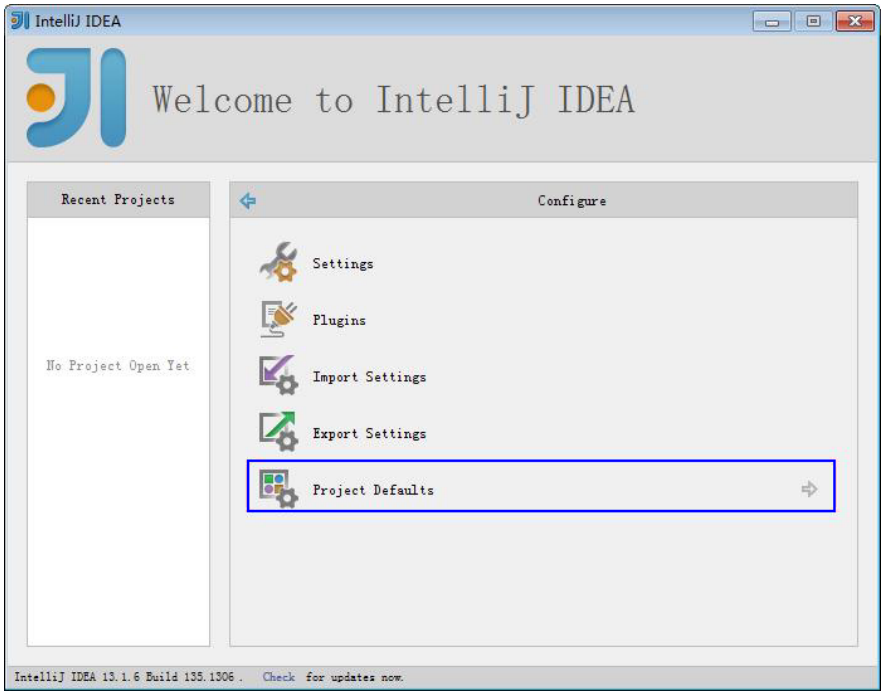
1. 安装JDK。
2. 安装IntelliJ IDEA工具。
3. 在IntelliJ IDEA中配置JDK。
   a. 打开IntelliJ IDEA，选择“Configure”。

图 8-8 Quick Start



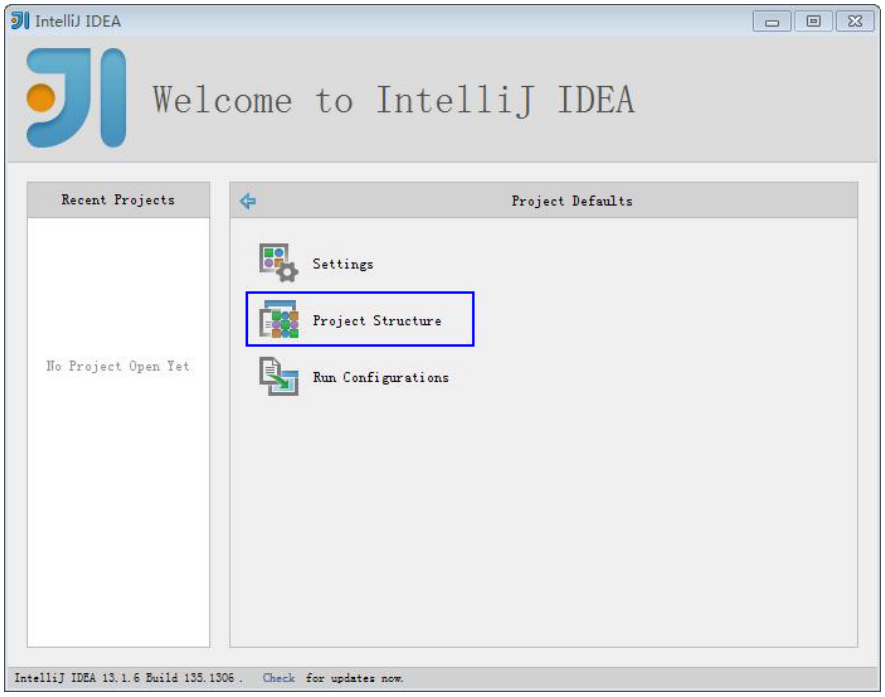
b. 在“Configure”页面中选择的“Project Defaults”。

图 8-9 Configure



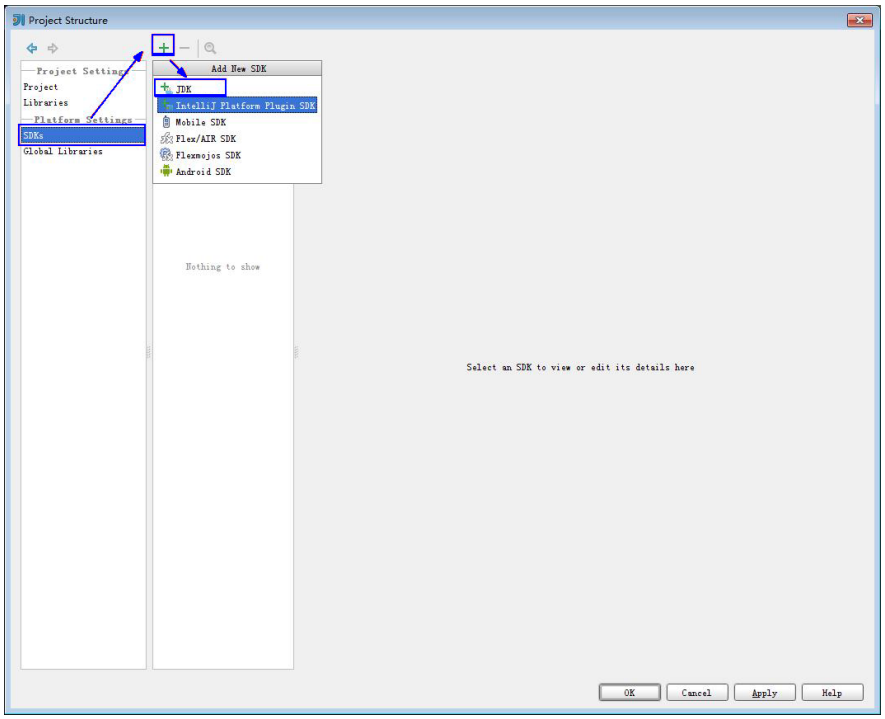
c. 在“Project Defaults”页面中，选择“Project Structure”。

图 8-10 Project Defaults



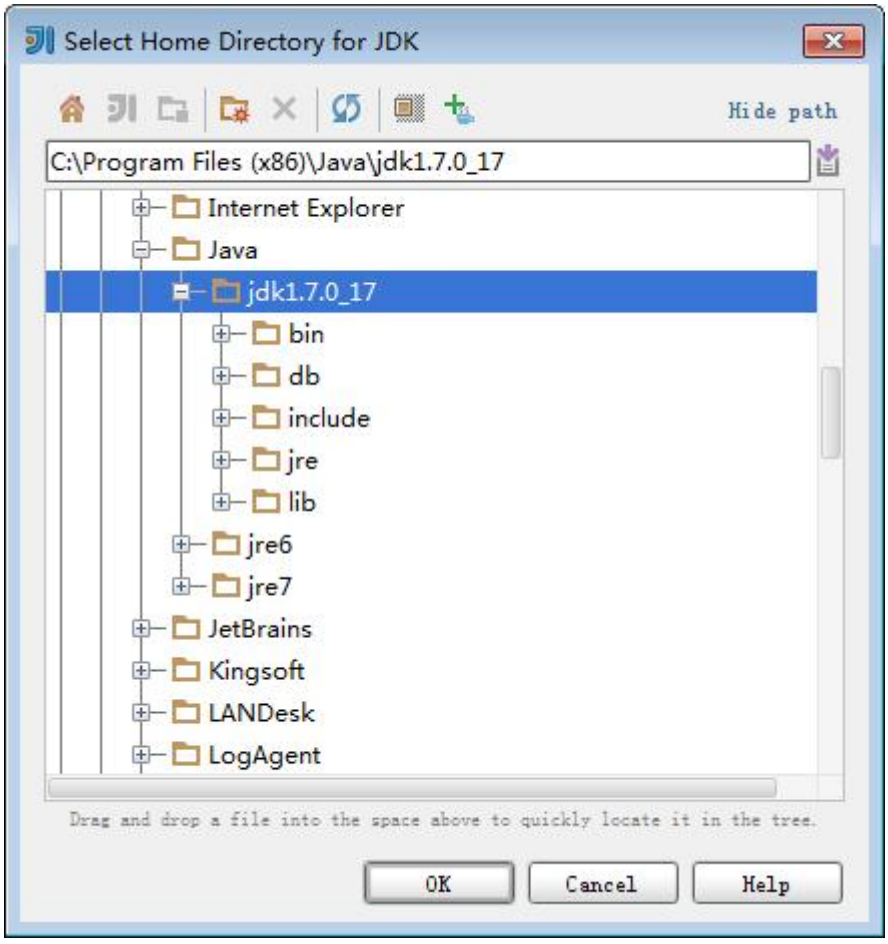
d. 在打开的“Project Structure”页面中，选择“SDKs”，单击绿色加号添加JDK。

图 8-11 添加 JDK



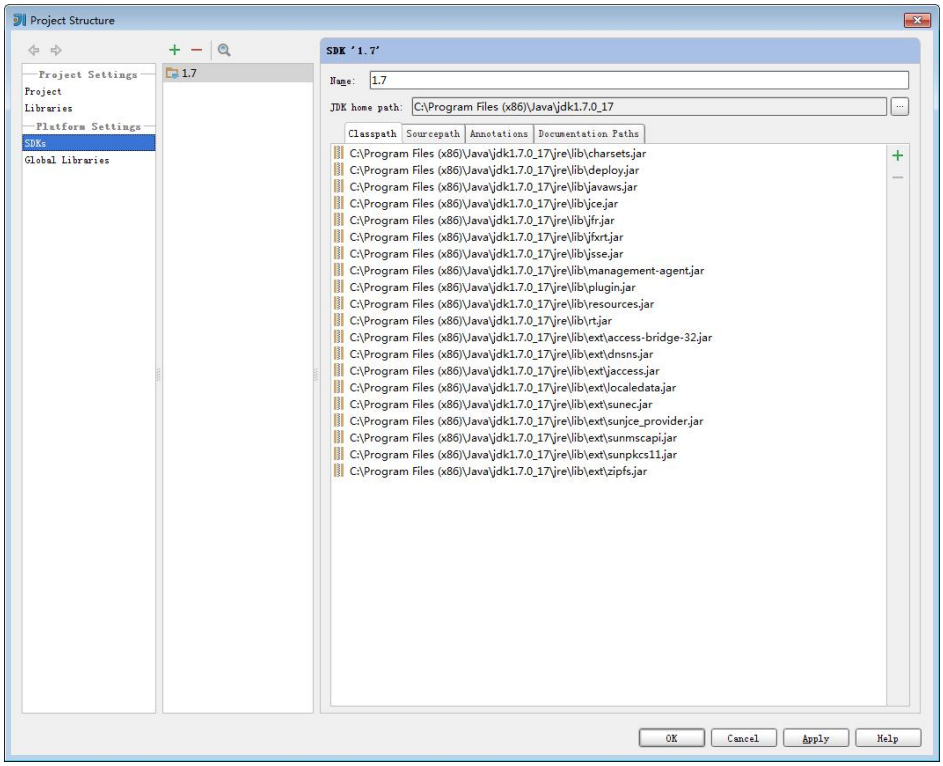
e. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 8-12 选择 JDK 目录



f. 完成JDK选择后，单击“OK”完成配置。

图 8-13 完成 JDK 配置



----结束

8.2.4 准备 Scala 开发环境

操作场景

Scala开发环境可以搭建在Windows环境下，而运行环境（即客户端）只能部署在Linux环境下。

操作步骤

步骤1 对于Scala开发环境，推荐使用IDEA工具，安装要求如下。

- JDK使用1.7版本（或1.8版本）
- IntelliJ IDEA（版本：13.1.6）
- Scala（版本：2.11.8）
- Scala插件（版本：0.35.683）

说明

- Spark不支持当客户端程序使用IBM JDK 1.7运行时，使用yarn-client模式向服务端提交Spark任务。
- Oracle JDK需进行安全加固，具体操作如下。

1. 到Oracle官方网站获取与JDK版本对应的JCE（Java Cryptography Extension）文件。JCE文件解压后包含“local_policy.jar”和“US_export_policy.jar”。拷贝jar包到如下路径。

   Linux：JDK安装目录/jre/lib/security

   Windows：JDK安装目录\jre\lib\security

2. 将“客户端安装目录/JDK/jdk/jre/lib/ext/”目录下“SMS4JA.jar”拷贝到如下路径。

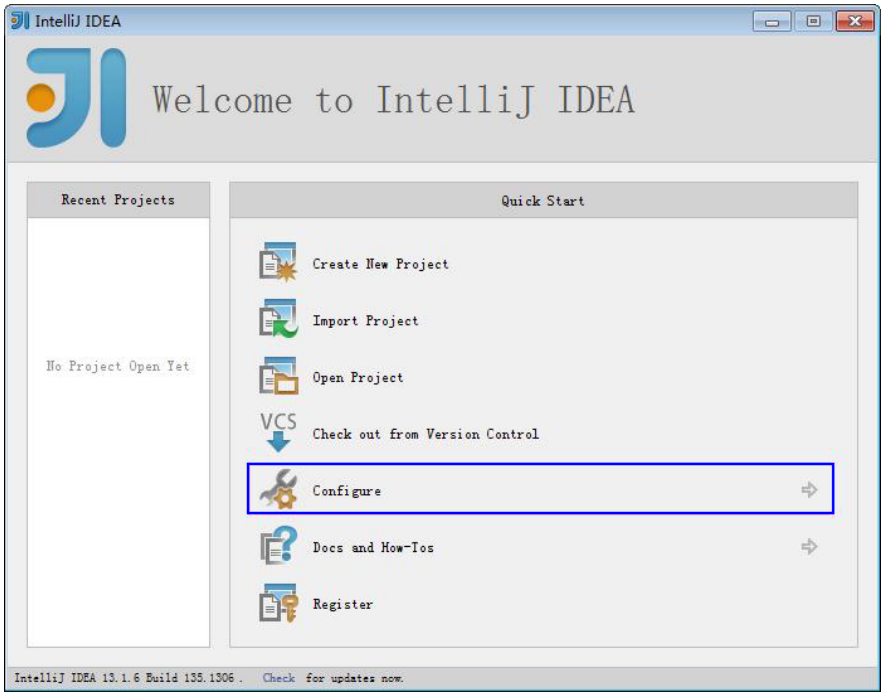
   Linux：JDK安装目录/jre/lib/ext/

   Windows：JDK安装目录\jre\lib\ext\

步骤2 安装IntelliJ IDEA、JDK和Scala工具，并进行相应的配置。

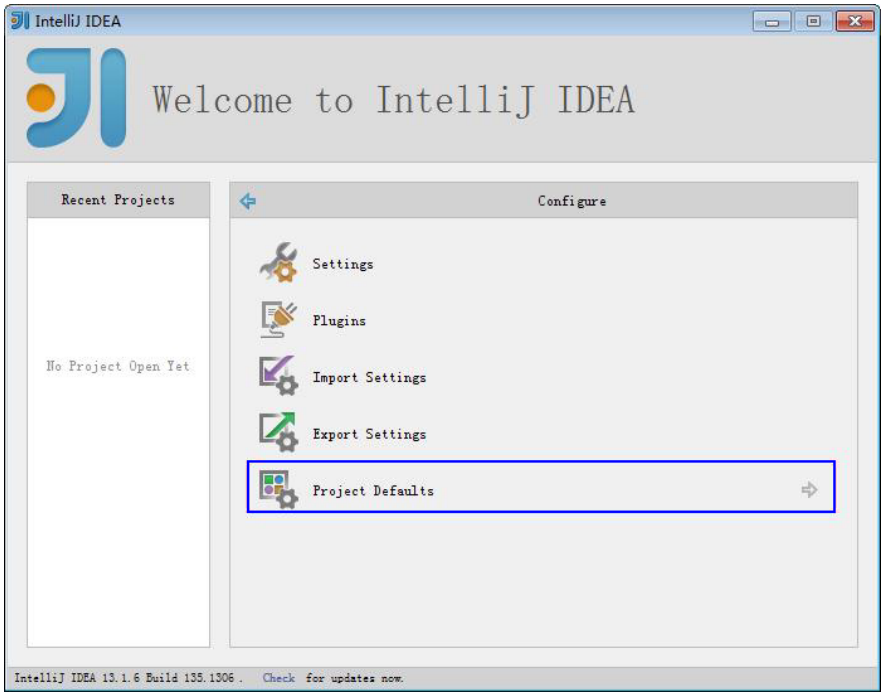
1. 安装JDK。
2. 安装IntelliJ IDEA。
3. 安装Scala工具。
4. 在IntelliJ IDEA中配置JDK。
   a. 打开IntelliJ IDEA，选择“Configure”。

图 8-14 Quick Start



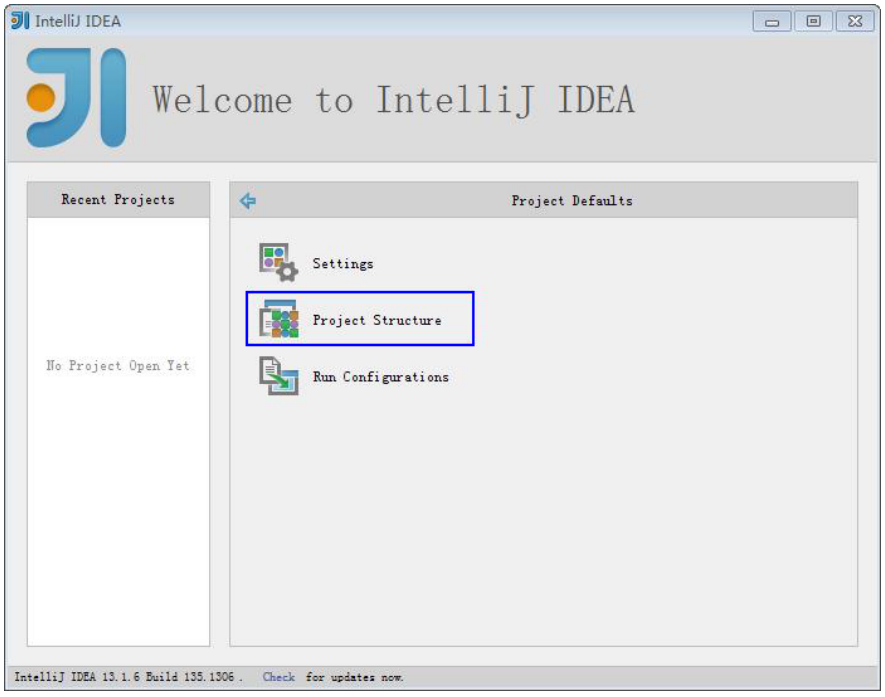
   b. 在“Configure”页面中选择的“Project Defaults”。

图 8-15 Configure



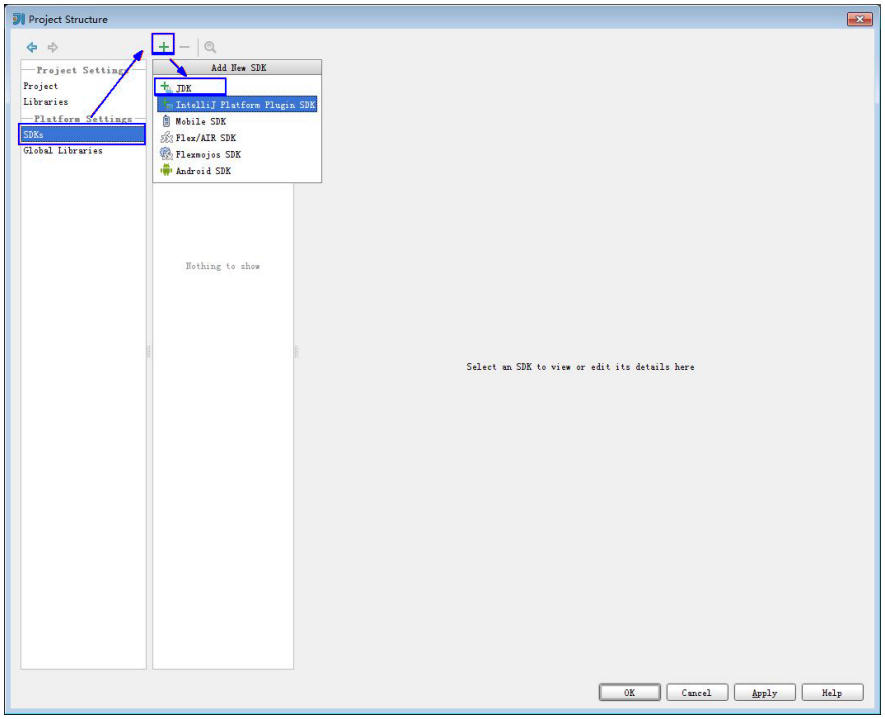
c. 在“Project Defaults”页面中，选择“Project Structure”。

图 8-16 Project Defaults



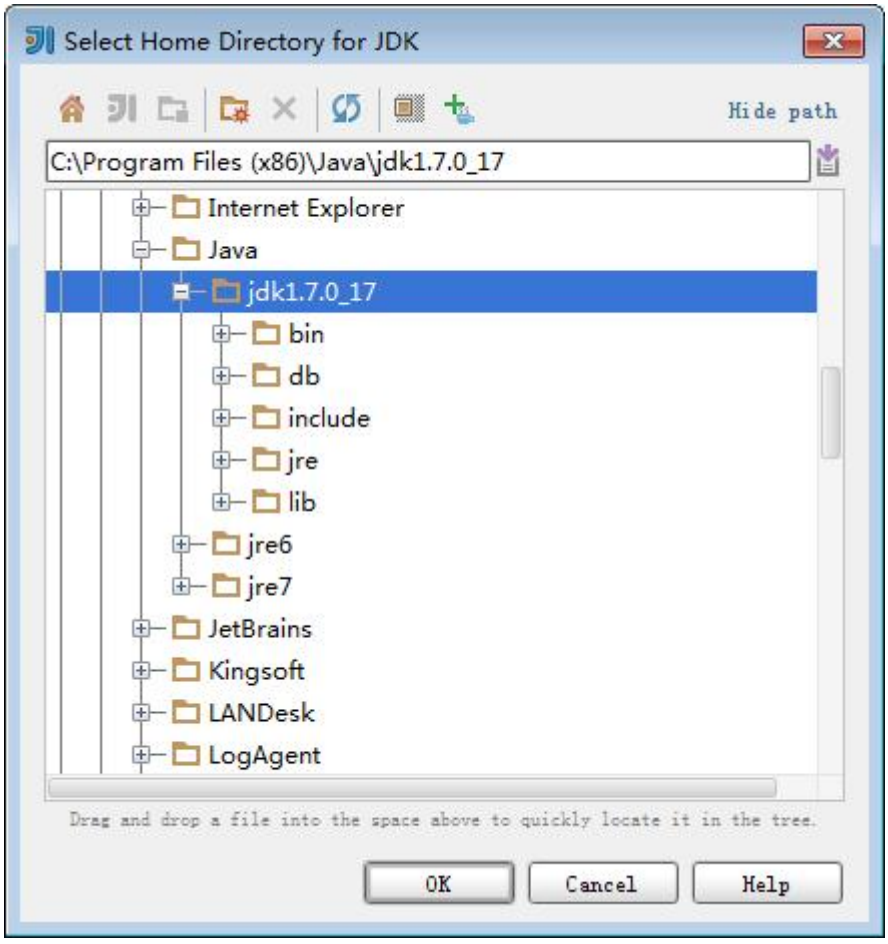
d. 在打开的“Project Structure”页面中，选择“SDKs”，单击绿色加号添加JDK。

图 8-17 添加 JDK



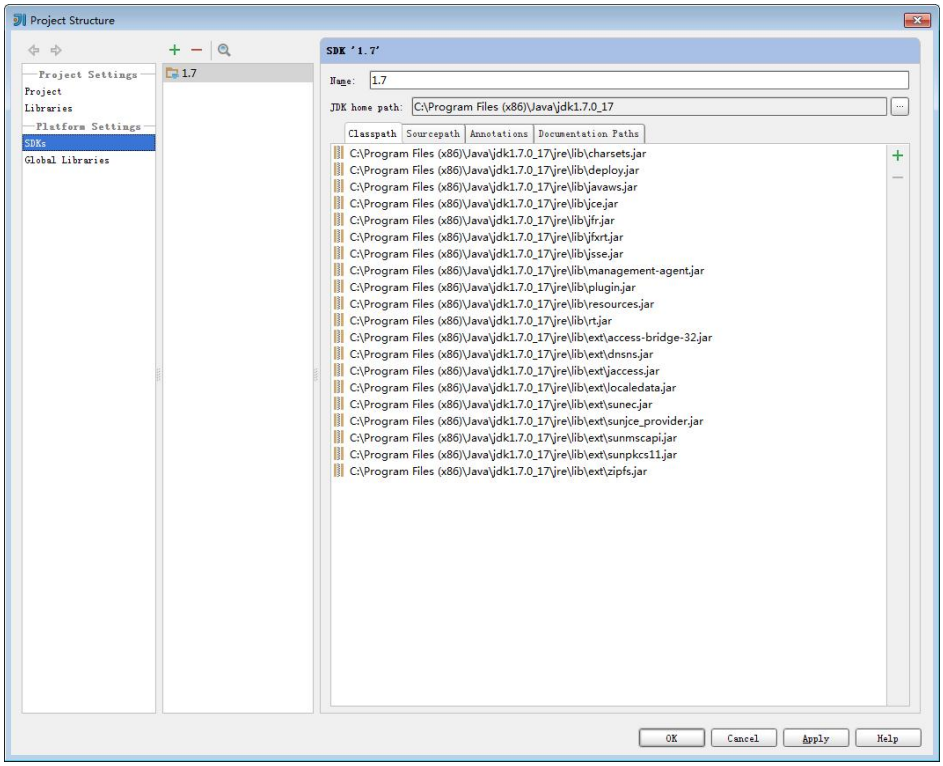
e. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 8-18 选择 JDK 目录



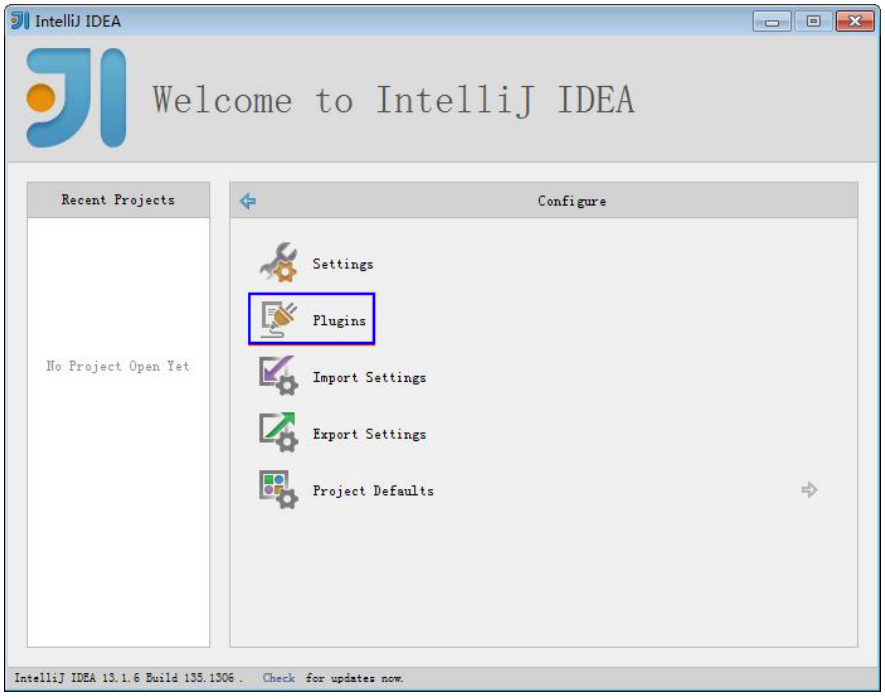
f. 完成JDK选择后，单击“OK”完成配置。

图 8-19 完成 JDK 配置



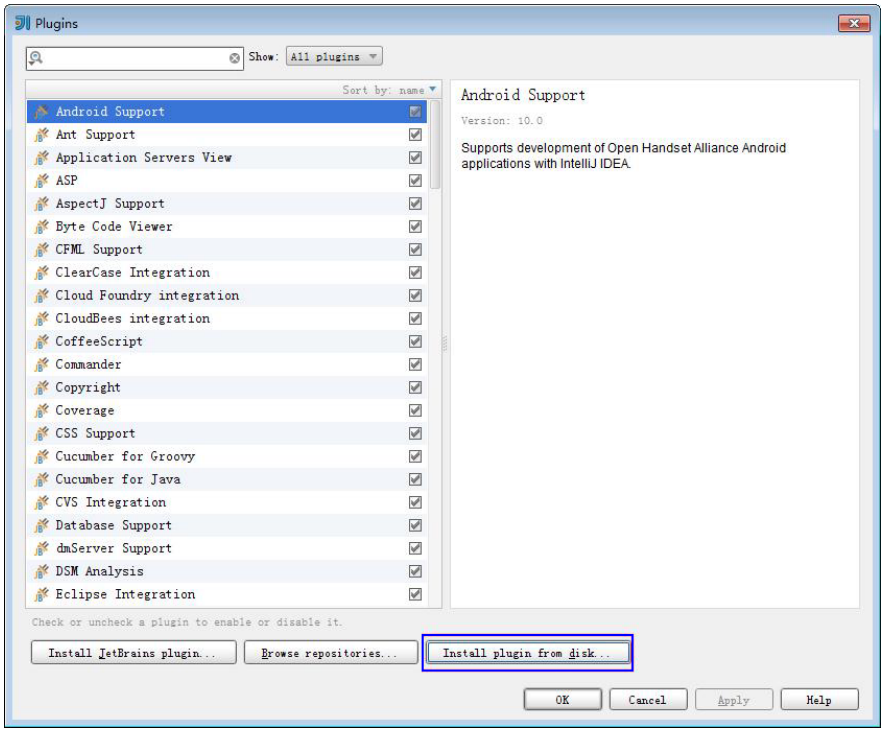
5. 在IntelliJ IDEA中安装Scala插件。
   a. 在“Configure”页面，选择“Plugins”。

图 8-20 Plugins

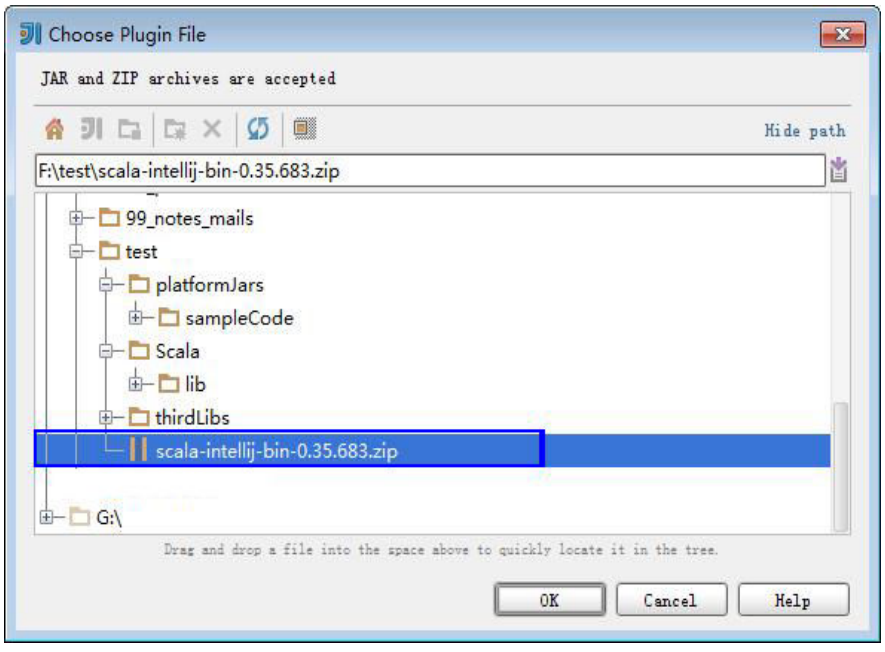


   b. 在“Plugins”页面，选择“Install plugin from disk”。

图 8-21 Install plugin from disk



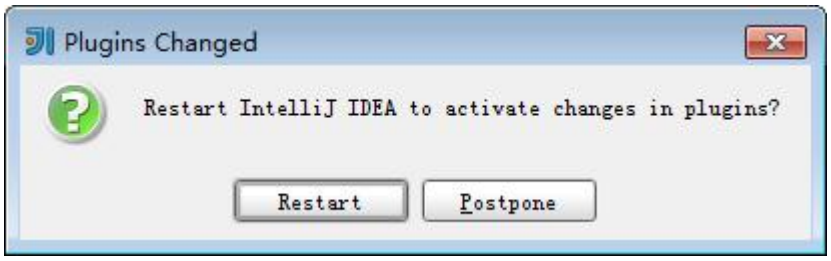
c. 在“Choose Plugin File”页面，选择对应版本的Scala插件包，单击“OK”。



d. 在“Plugins”页面，单击“Apply”安装Scala插件。

e. 在弹出的“Plugins Changed”页面，单击“Restart”，使配置生效。

图 8-22 Plugins Changed



----结束

8.2.5 准备 Python 开发环境

操作场景

Python开发环境可以搭建在Windows环境下，而运行环境（即客户端）只能部署在Linux环境下。

操作步骤

步骤1 对于Python开发环境，直接使用Editra编辑器（或其他编写Python应用程序的IDE）即可。

步骤2 下载客户端样例配置程序到本地开发环境。

使用FTP工具，将运行调测环境的客户端包文件“MRS_Service_client”下载到本地，并解压得到目录“MRS_Services_ClientConfig”。

----结束

8.2.6 准备运行环境

操作场景

Spark的运行环境（即客户端）只能部署在Linux环境下。您可以执行如下操作完成运行环境准备。

准备运行调测环境

步骤1 在弹性云服务器管理控制台，申请一个新的弹性云服务器，用于应用开发运行调测。

- 弹性云服务器的安全组需要和MRS集群Master节点的安全组相同。
- 弹性云服务器的VPC需要与MRS集群在同一个VPC中。
- 弹性云服务器的网卡需要与MRS集群在同一个网段中。

步骤2 申请弹性IP，绑定新申请的弹性云主机IP，并配置安全组出入规则。

步骤3 下载客户端程序，请参考**下载MRS客户端**。

步骤4 登录客户端下载目标节点，以root用户安装集群客户端。

1. 执行以下命令解压客户端包。

   tar -xvf /opt/MRS_Services_Client.tar

2. 执行以下命令校验安装文件包。

   sha256sum -c /opt/MRS_Services_ClientConfig.tar.sha256

   ```
   MRS_Services_ClientConfig.tar:OK
   ```

3. 执行以下命令解压安装文件包。

   tar -xvf MRS_Services_ClientConfig.tar

4. 执行如下命令安装客户端到指定目录（绝对路径），例如“/opt/client”。目录会自动创建。

   cd /opt/MRS_Services_ClientConfig

   sh install.sh /opt/client

   ```
   Components client installation is complete.
   ```

----结束

8.2.7 下载并导入样例工程

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8-23所示。

图 8-23 导入样例工程流程

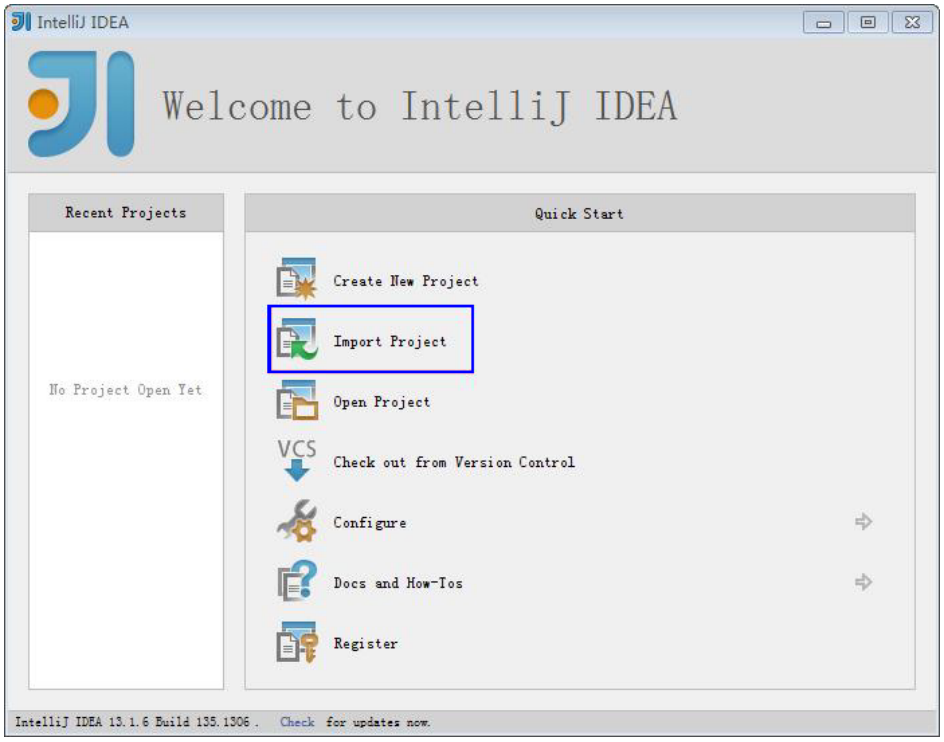
操作步骤

步骤1 参照样例工程获取地址，下载样例工程到本地。

步骤2 将Java样例工程导入到IDEA中。

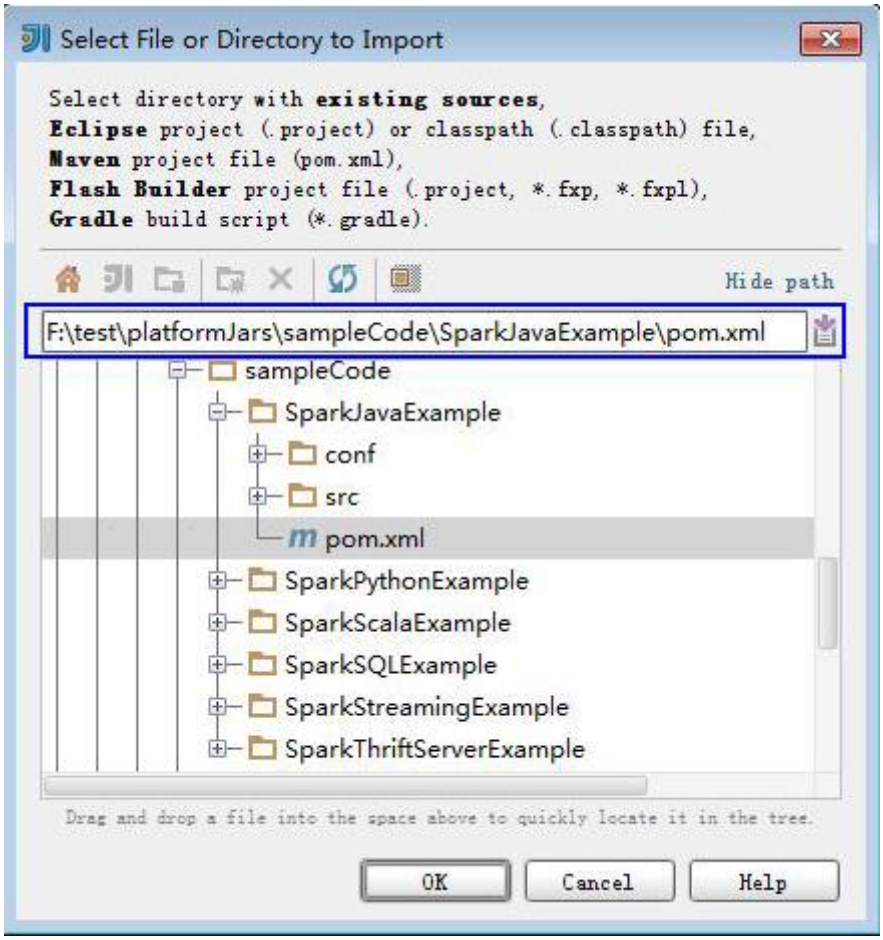
1. 打开IntelliJ IDEA。在“Quick Start”页面选择“Import Project”。
   或者，针对已使用过的IDEA工具，您可以从IDEA主界面直接添加。选择“File > Import project...”导入工程。

   图 8-24 Import Project（Quick Start 页面）



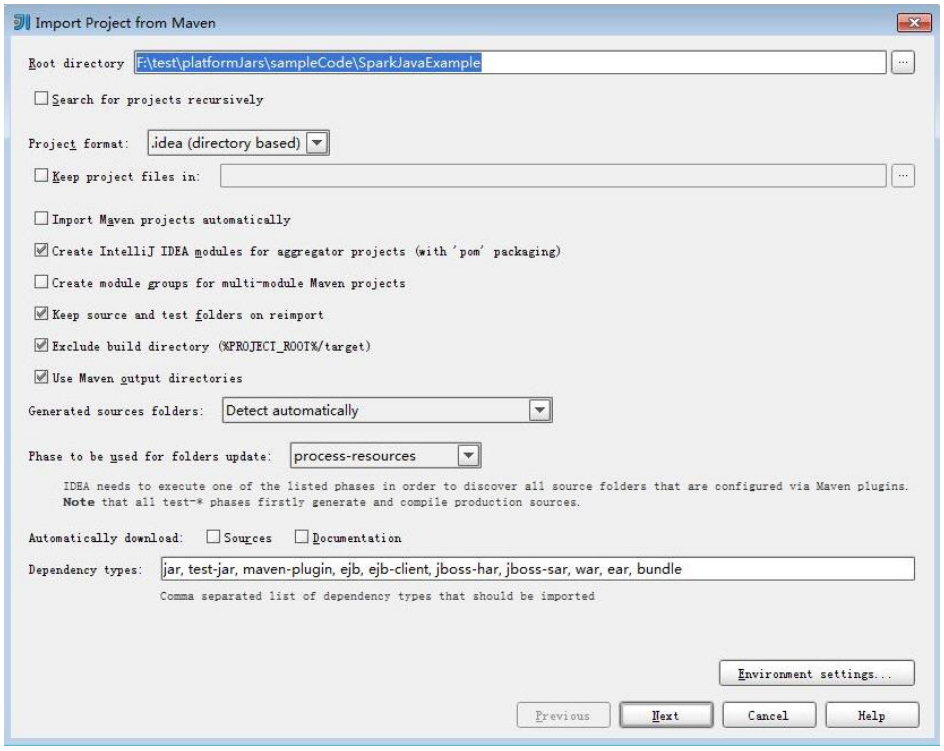
2. 选择需导入的样例工程存放路径，单击“OK”。

图 8-25 Select File or Directory to Import



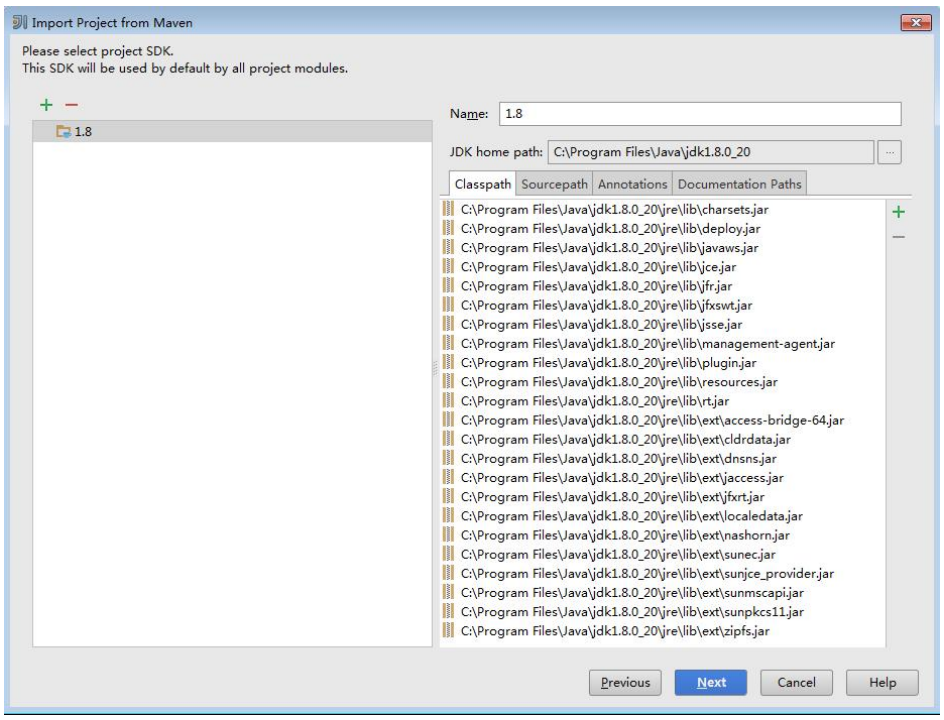
3. 确认导入路径和名称，单击“Next”。

图 8-26 Import Project from Maven



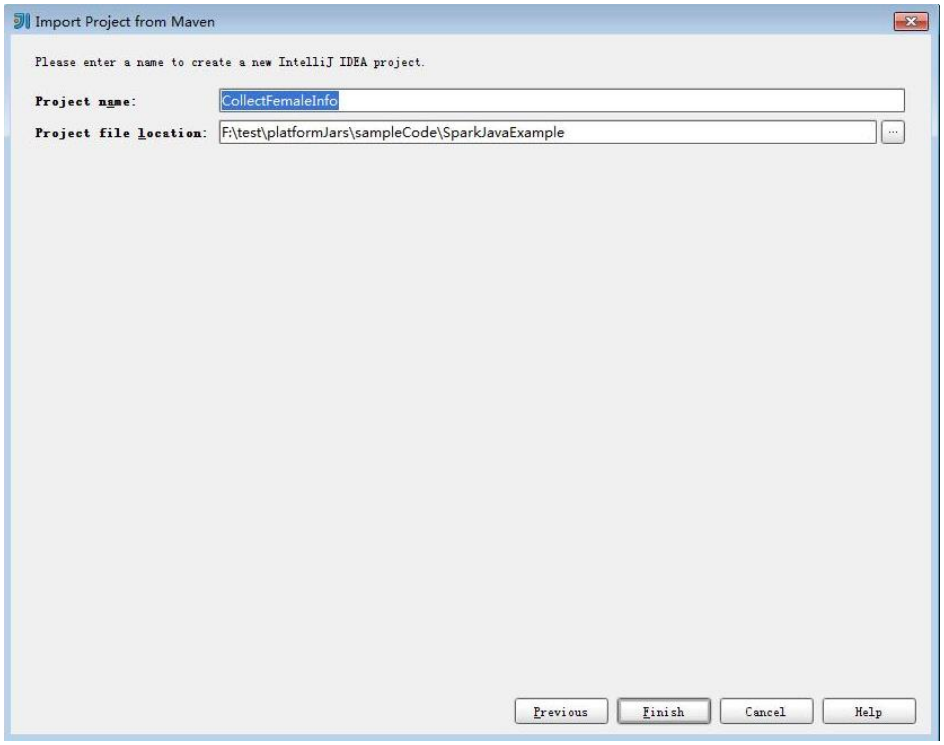
4. 选择需要导入的工程，单击“Next”。
5. 确认工程所用JDK，单击“Next”。

图 8-27 Select project SDK



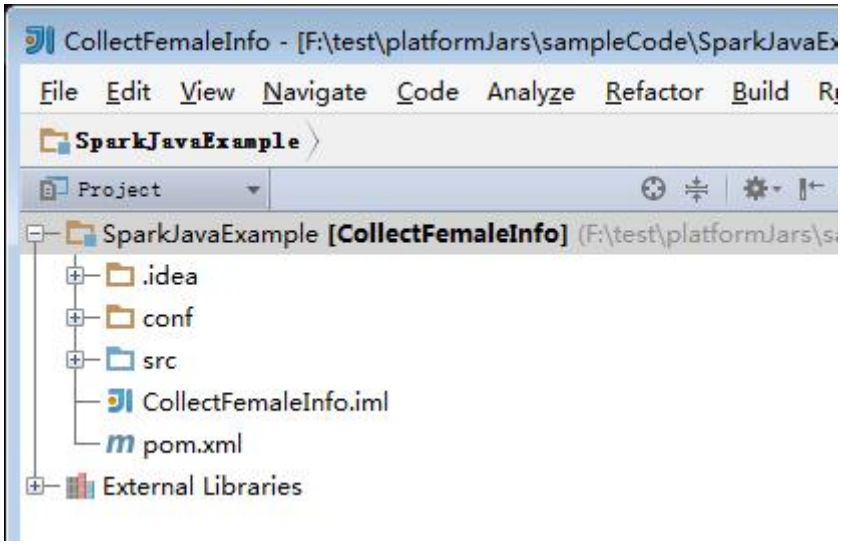
6. 确认工程名称和路径，单击“Finish”完成导入。

图 8-28 Select project to import



7. 导入完成后，IDEA主页显示导入的样例工程。

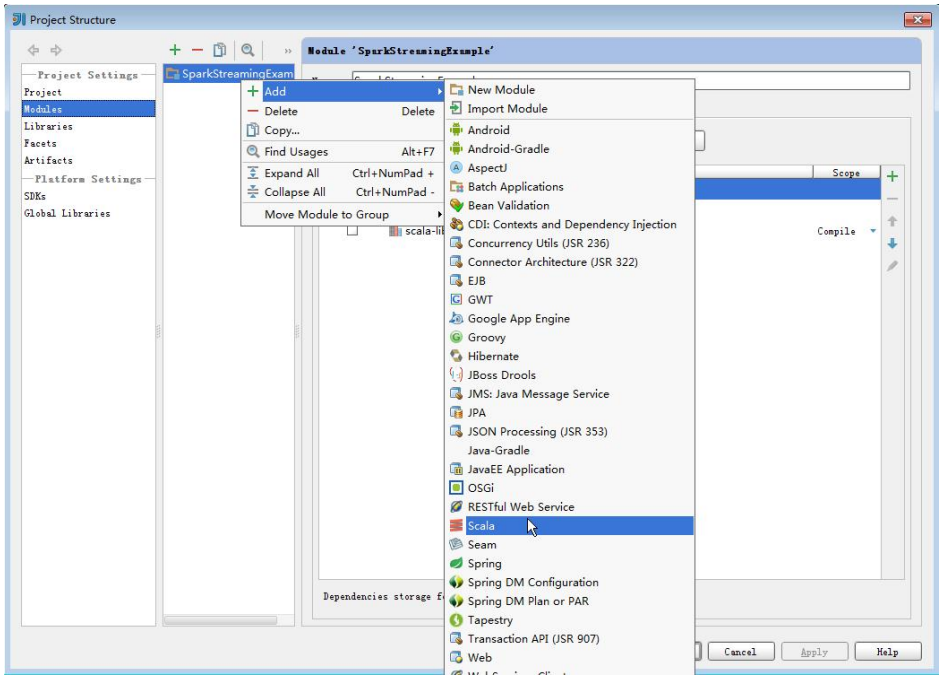
图 8-29 已导入工程



步骤3 （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。

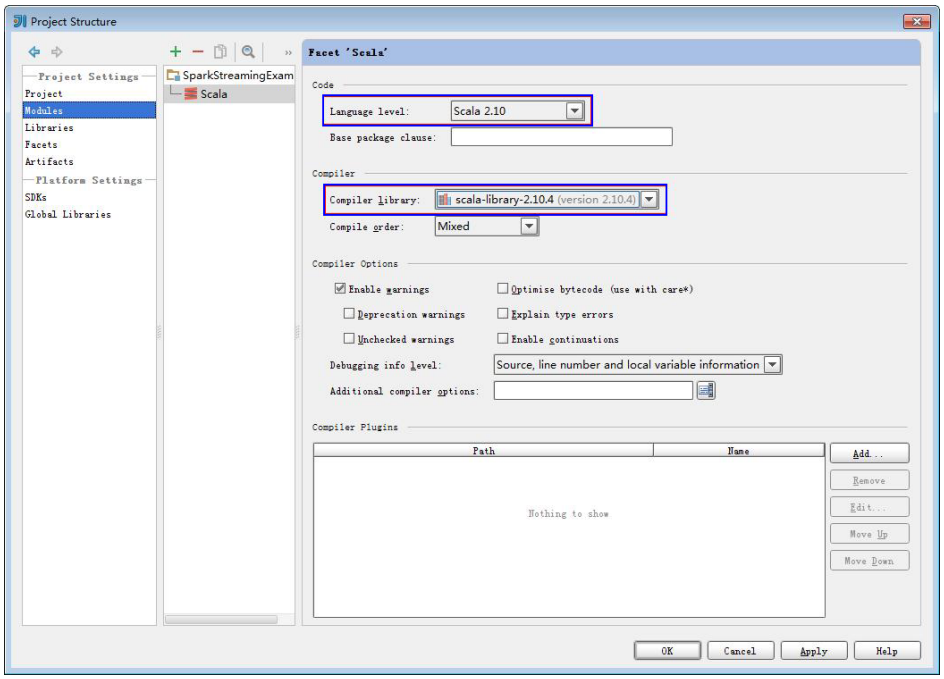
1. 在IDEA主页，选择“File > Project Structures...”进入“Project Structure”页面。
2. 选择“Modules”，选中工程名称，然后右键选择“Add > Scala”。

图 8-30 选择 Scala 语言



3. 在设置界面，选择编译的依赖jar包，单击“Apply”。

图 8-31 选择编译依赖包

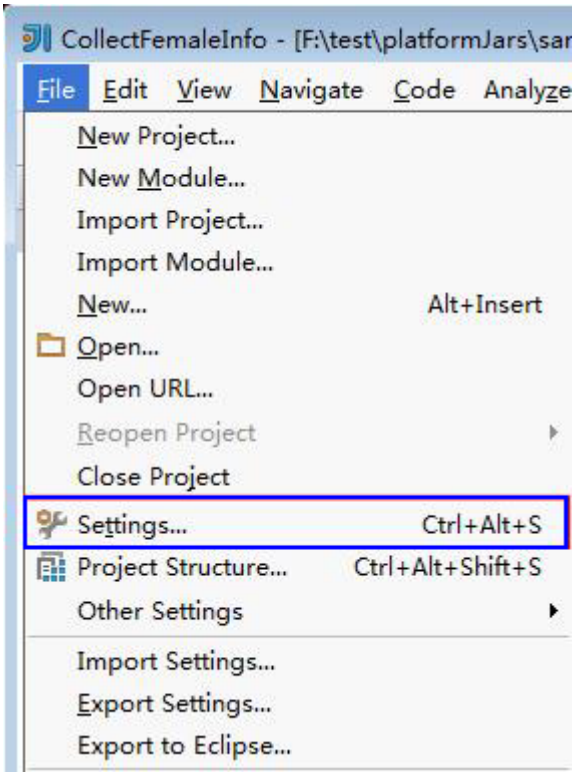


4. 单击“OK”保存设置。

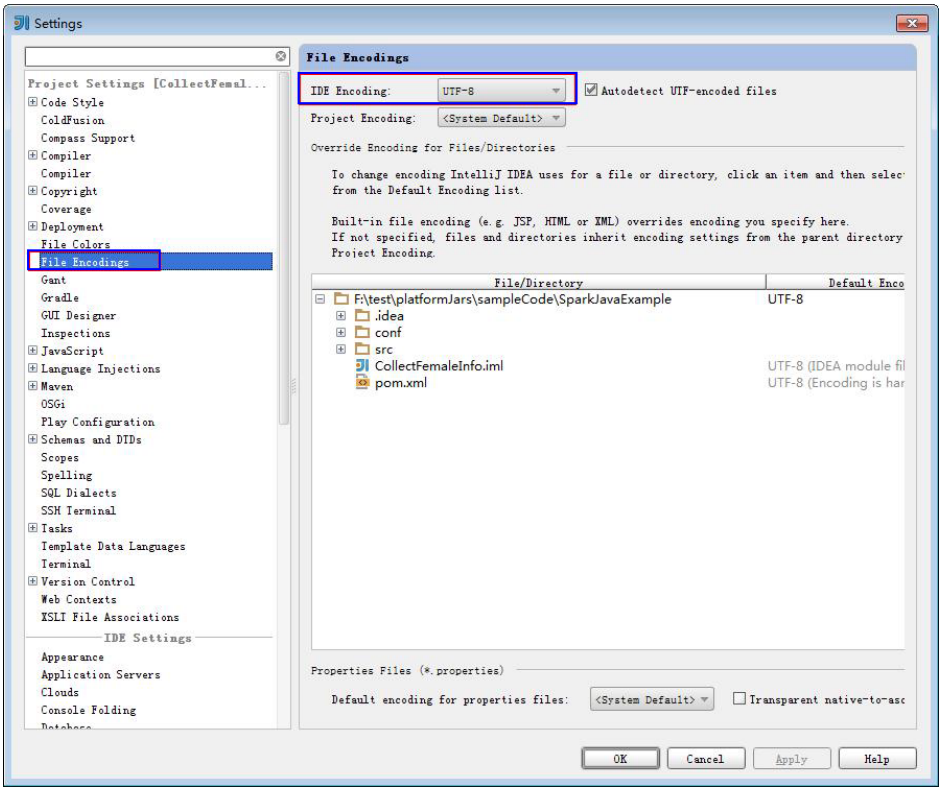
步骤4 设置IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IDEA首页，选择“File > Settings...”。

图 8-32 选择 Settings



2. 在“Settings”页面，选择“File Encodings”。然后在右侧的“IDE Encoding”的下拉框中，选择“UTF-8”。单击“Apply”。



3. 单击“OK”完成编码配置。

----结束

8.2.8 新建工程（可选）

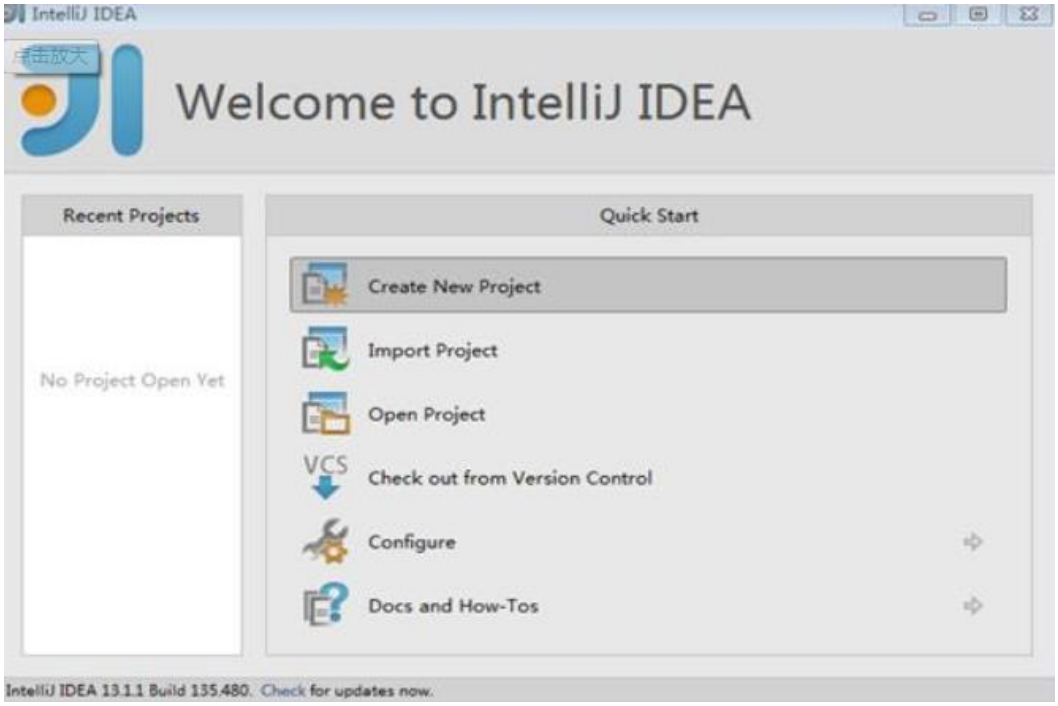
操作场景

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

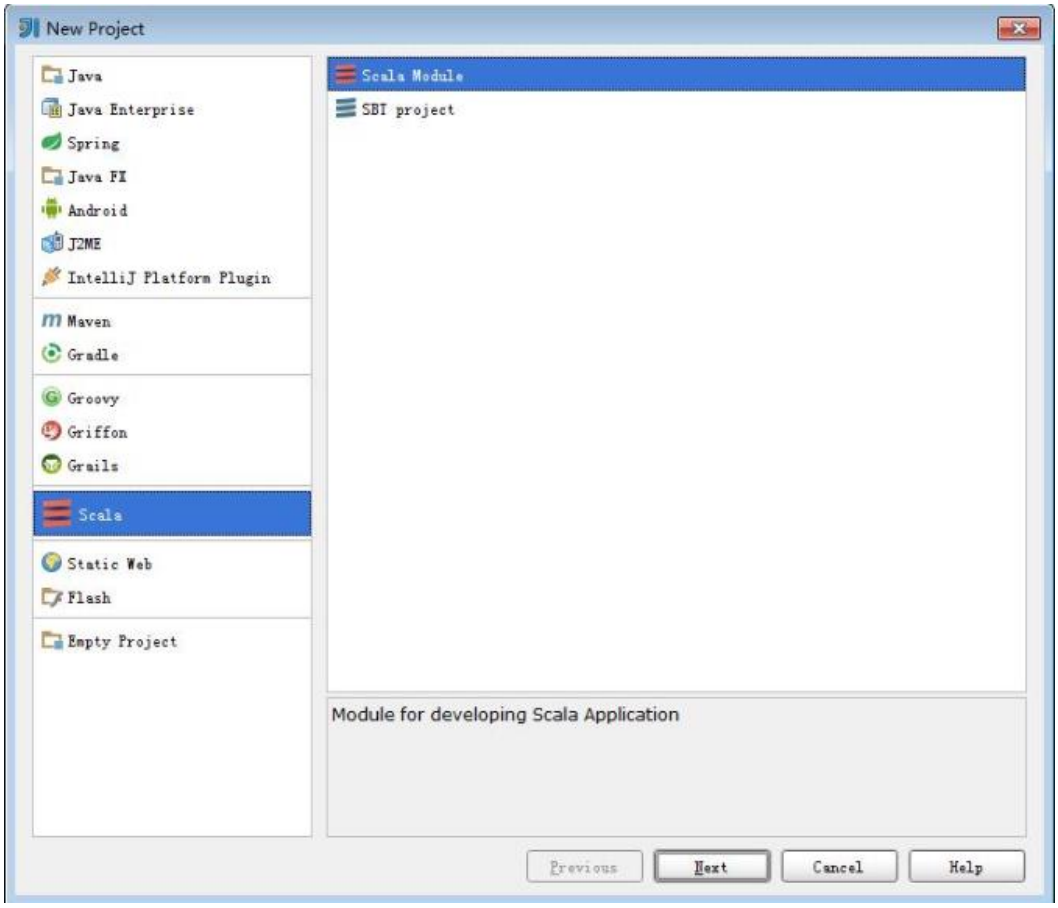
步骤1 打开IDEA工具，选择“Create New Project”。

图 8-33 创建工程



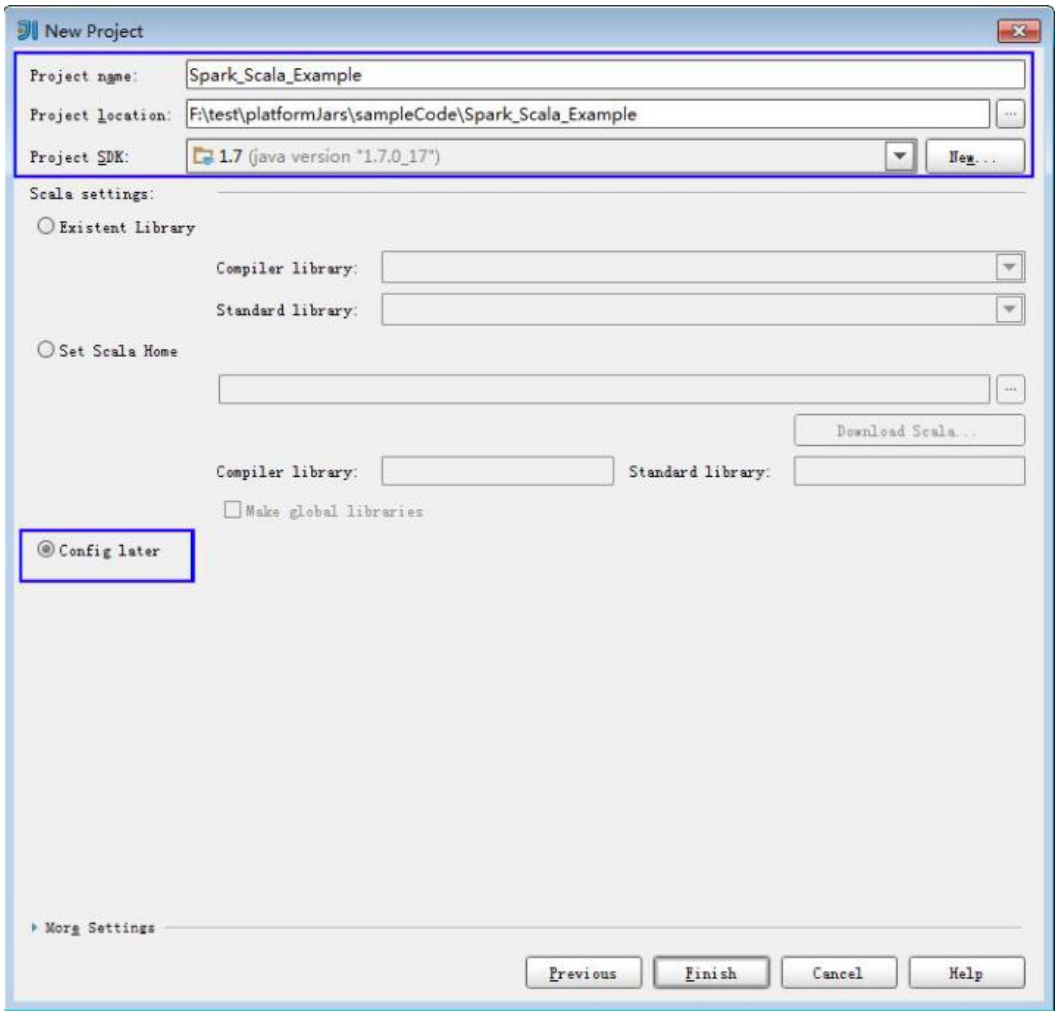
步骤2 在“New Project”页面，选择“Scala”开发环境，并选择“Scala Module”，然后单击“Next”。如果您需要新建Java语言的工程，选择对应参数即可。

图 8-34 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，并勾选“Config later”（待工程创建完毕后引入scala的编译库文件），然后单击“Finish”完成工程创建。

图 8-35 填写工程信息



----结束

8.2.9 准备认证机制代码

前提条件

MRS服务集群开启了Kerberos认证。

场景说明

在集群开启Kerberos认证的环境下，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。

  a. 在客户端的“spark-default.conf”配置文件中，配置“spark.yarn.keytab”和“spark.yarn.principal”参数指定认证信息。

  b. 执行**bin/spark-submit**的命令中添加如下参数来指定认证信息。

  --conf spark.yarn.keytab=<keytab文件路径> **--conf spark.yarn.principal**=<Principal账号>

  c. 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。

  在集群开启Kerberos认证环境下，样例代码需要使用的认证方式如表1所示：

表 8-4 安全认证方式

样例代码	模式	安全认证方式
spark-examples-normal	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。
spark-examples-security（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。

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2所示，请根据实际情况修改对应参数值。

表 8-5 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，您可以联系管理员获取此账号。
userKeytabPath	/opt/FIclient/user.keytab	用户用于认证的Keytab文件，您可以联系管理员获取文件。
krb5ConfPath	/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf	krb5.conf文件路径和文件名称。
ZKServerPrincipal	zookeeper/hadoop.hadoop.com	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：

  ```
  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.hadoop.com";
  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。

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3所示，请根据实际情况修改对应参数值。

表 8-6 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，您可以联系管理员获取此账号。
userKeytabPath	/opt/FIclient/user.keytab	用户用于认证的Keytab文件，您可以联系管理员获取文件。
krb5ConfPath	/opt/FIclient/KrbClient/kerberos/var/krb5kdc/krb5.conf	krb5.conf文件路径和文件名称。
ZKServerPrincipal	zookeeper/hadoop.hadoop.com	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.hadoop.com"
  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);
  ```

8.3 开发程序

8.3.1 Spark Core 程序

8.3.1.1 场景说明

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，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HDFS客户端，执行如下命令获取安全认证。

      cd /opt/client

      kinit -kt '/opt/client/Spark/spark/conf/user.keytab' <用于认证的业务用户>

      说明

      user.keytab文件位置请根据自己实际路径填写。

   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*** *input_xxx.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。

8.3.1.2 Java 样例代码

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
  SparkConf conf = new SparkConf().setAppName("CollectFemaleInfo");
  JavaSparkContext jsc = new JavaSparkContext(conf);

  //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 = jsc.textFile(args[0]);

  //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;

      @Override
      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;

      @Override
      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;

    @Override
    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;

    @Override
    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;

    @Override
    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());
  }
```

8.3.1.3 Scala 样例代码

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：

```
//配置Spark应用名称
val conf = new SparkConf().setAppName("CollectFemaleInfo")

//提交Spark作业
val sc = new SparkContext(conf)
//读取数据。其是传入参数args(0)指定数据路径
val text = sc.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.foreach(println)
```

8.3.1.4 Python 样例代码

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)

  # 创建SparkContext，设置AppName
  sc = SparkContext(appName = "CollectFemaleInfo")?

  """
  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 - textFile
  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 = sc.textFile(name = inputPath, use_unicode = False) \
    .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
  sc.stop()
```

8.3.2 Spark SQL 程序

8.3.2.1 场景说明

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，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HDFS客户端，执行如下命令获取安全认证。

      cd /opt/client

      kinit -kt '/opt/client/Spark/spark/conf/user.keytab' <用于认证的业务用户>

      说明

      user.keytab文件位置请根据自己实际路径填写。

   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*** *input_xxx.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。

8.3.2.2 Java 样例代码

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：

```
SparkConf conf = new SparkConf().setAppName("CollectFemaleInfo");
JavaSparkContext jsc = new JavaSparkContext(conf);
SQLContext sqlContext = new org.apache.spark.sql.SQLContext(jsc);

// 通过隐式转换，将RDD转换成DataFrame
JavaRDD<FemaleInfo> femaleInfoJavaRDD = jsc.textFile(args[0]).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。
DataFrame schemaFemaleInfo = sqlContext.createDataFrame(femaleInfoJavaRDD,FemaleInfo.class);
schemaFemaleInfo.registerTempTable("FemaleInfoTable");

// 执行SQL查询
DataFrame femaleTimeInfo = sqlContext.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);
jsc.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/latest/sql-programming-guide.html#running-sql-queries-programmatically**。

8.3.2.3 Scala 样例代码

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 CollectFemaleInfo {
  //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 sparkConf = new SparkConf().setAppName("FemaleInfo")
    val sc = new SparkContext(sparkConf)
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    import sqlContext.implicits._
    //通过隐式转换，将RDD转换成DataFrame，然后注册表
    sc.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 = sqlContext.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()
    c.foreach(println)
    sc.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/latest/sql-programming-guide.html#running-sql-queries-programmatically**

8.3.3 Spark Streaming 程序

8.3.3.1 场景说明

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：

- 实时统计连续网购时间超过半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

Spark Streaming样例工程的数据存储在Kafka组件中（需要有Kafka权限用户）。

1. 本地新建两个文本文件input_data1.txt和input_data2.txt，将log1.txt的内容复制保存到input_data1.txt，将log2.txt的内容复制保存到input_data2.txt。
2. 在客户端安装节点下创建文件目录：“/home/data”。将上述两个文件上传到此“/home/data”目录下。
3. 将Kafka的Broker配置参数“allow.everyone.if.no.acl.found”值设置为“true”(普通集群不需配置)。
4. 启动样例代码的Producer，向Kafka发送数据。

   java -cp $SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient/*:{JAR_PATH} com.huawei.bigdata.spark.examples.StreamingExampleProducer {BrokerList} {Topic}

   说明

   - JAR_PATH为程序jar包所在路径。
   - brokerlist格式为brokerIp:9092。

开发思路

统计日志文件中本周末网购停留总时间超过半个小时的女性网民信息。

主要分为四个部分：

- 接收Kafka中数据，生成相应DStream。
- 筛选女性网民上网时间数据信息。
- 汇总在一个时间窗口内每个女性上网时间。
- 筛选连续上网时间超过阈值的用户，并获取结果。

8.3.3.2 Java 样例代码

功能介绍

实时统计连续网购时间超过半个小时的女性网民信息，将统计结果直接打印或者输出写入到Kafka中。

Spark Streaming Write To Print 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.FemaleInfoCollectionPrint：

```
    // 参数解析:
    // <batchTime>为Streaming分批的处理间隔。
    // <windowTime>为统计数据的时间跨度,时间单位都是秒。
    // <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。
public class FemaleInfoCollectionPrint {
    public static void main(String[] args) throws Exception {

        String batchTime = args[0];
        final String windowTime = args[1];
        String topics = args[2];
        String brokers = args[3];

        Duration batchDuration = Durations.seconds(Integer.parseInt(batchTime));
        Duration windowDuration = Durations.seconds(Integer.parseInt(windowTime));


        SparkConf conf = new SparkConf().setAppName("DataSightStreamingExample");
        JavaStreamingContext jssc = new JavaStreamingContext(conf, batchDuration);

        // 设置Streaming的CheckPoint目录，由于窗口概念存在，该参数必须设置
        jssc.checkpoint("checkpoint");

        // 组装Kafka的主题列表
        HashSet<String> topicsSet = new HashSet<String>(Arrays.asList(topics.split(",")));
        HashMap<String, String> kafkaParams = new HashMap<String, String>();
        kafkaParams.put("metadata.broker.list", brokers);

        // 通过brokers和topics直接创建kafka stream
        // 1.接收Kafka中数据，生成相应DStream
        JavaDStream<String> lines = KafkaUtils.createDirectStream(jssc,String.class,String.class,
                StringDecoder.class,  StringDecoder.class, kafkaParams, topicsSet).map(
                new Function<Tuple2<String, String>, String>() {
                    @Override
                    public String call(Tuple2<String, String> tuple2) {
                        return tuple2._2();
                    }
                }
        );

        // 2.获取每一个行的字段属性
        JavaDStream<Tuple3<String, String, Integer>> records = lines.map(
                new Function<String, Tuple3<String, String, Integer>>() {
                    @Override
                    public Tuple3<String, String, Integer> call(String line) throws Exception {
                        String[] elems = line.split(",");
                        return new Tuple3<String, String, Integer>(elems[0], elems[1], Integer.parseInt(elems[2]));
                    }
                }
        );

        // 3.筛选女性网民上网时间数据信息
        JavaDStream<Tuple2<String, Integer>> femaleRecords = records.filter(new Function<Tuple3<String,
String, Integer>, Boolean>() {
```

```
            public Boolean call(Tuple3<String, String, Integer> line) throws Exception {
                if (line._2().equals("female")) {
                    return true;
                } else {
                    return false;
                }
            }
        }).map(new Function<Tuple3<String, String, Integer>, Tuple2<String, Integer>>() {
            public Tuple2<String, Integer> call(Tuple3<String, String, Integer> stringStringIntegerTuple3) throws
Exception {
                return new Tuple2<String, Integer>(stringStringIntegerTuple3._1(),
stringStringIntegerTuple3._3());
            }
        });

        // 4.汇总在一个时间窗口内每个女性上网时间
        JavaPairDStream<String, Integer> aggregateRecords =
JavaPairDStream.fromJavaDStream(femaleRecords)
                .reduceByKeyAndWindow(new Function2<Integer, Integer, Integer>() {
                    public Integer call(Integer integer, Integer integer2) throws Exception {
                        return integer + integer2;
                    }
                }, new Function2<Integer, Integer, Integer>() {
                    public Integer call(Integer integer, Integer integer2) throws Exception {
                        return integer - integer2;
                    }
                }, windowDuration, batchDuration);


        JavaPairDStream<String, Integer> upTimeUser = aggregateRecords.filter(new Function<Tuple2<String,
Integer>, Boolean>() {
            public Boolean call(Tuple2<String, Integer> stringIntegerTuple2) throws Exception {
                if (stringIntegerTuple2._2() > 0.9 * Integer.parseInt(windowTime)) {
                    return true;
                } else {
                    return false;
                }
            }
        });

        // 5.筛选连续上网时间超过阈值的用户，并获取结果
        upTimeUser.print();

        // 6.Streaming系统启动
        jssc.start();
        jssc.awaitTermination();

    }
```

Spark Streaming Write To Kafka 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.JavaDstreamKafkaWriter：

说明

- Spark版本升级后，推荐使用新接口createDirectStream，老接口createStream仍然存在，但是性能和稳定性差，建议不要使用老接口开发应用程序。
- 该样例代码只存在于mrs-sample-project-1.6.0.zip中

```
    // 参数解析:
    //<groupId> 消费者的group.id.
    //<brokers> broker的IP和端口.
    //<topic> kafka的topic.
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");

  // 配置Kafka
  Properties kafkaParams = new Properties();
  kafkaParams.put("metadata.broker.list", brokers);
  kafkaParams.put("group.id", groupId);
  kafkaParams.put("auto.offset.reset", "smallest");

  // 创建一个Java streaming context
  JavaStreamingContext ssc = new JavaStreamingContext(sparkConf, Durations.milliseconds(500));

  // 向kafka发送数据
  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RDD queue
  Queue<JavaRDD<String>> sent = new LinkedList();
  sent.add(ssc.sparkContext().parallelize(sentData));

  // 使用写入的数据创建Dstream
  JavaDStream wStream = ssc.queueStream(sent);

  // 写入Kafka
  JavaDStreamKafkaWriterFactory.fromJavaDStream(wStream).writeToKafka(kafkaParams,
      new Function<String, KeyedMessage<String, byte[]>>() {
          public KeyedMessage<String, byte[]> call(String s) {
            return new KeyedMessage(topic, s.getBytes());
          }
  });

  ssc.start();
  ssc.awaitTermination();
 }
}
```

8.3.3.3 Scala 样例代码

功能介绍

实时统计连续网购时间超过半个小时的女性网民信息，将统计结果直接打印或者输出写入到Kafka中。

Spark Streaming Write To Print 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.FemaleInfoCollectionPrint：

```
  // 参数解析:
  // <batchTime>为Streaming分批的处理间隔。
  // <windowTime>为统计数据的时间跨度,时间单位都是秒。
  // <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。
  val Array(batchTime, windowTime, topics, brokers) = args
  val batchDuration = Seconds(batchTime.toInt)
```

```
  val windowDuration = Seconds(windowTime.toInt)

  // 建立Streaming启动环境
  val sparkConf = new SparkConf()
  sparkConf.setAppName("DataSightStreamingExample")
  val ssc = new StreamingContext(sparkConf, batchDuration)

  // 设置Streaming的CheckPoint目录，由于窗口概念存在，该参数必须设置
  ssc.checkpoint("checkpoint")

  // 组装Kafka的主题列表
   val topicsSet = topics.split(",").toSet

  // 通过brokers和topics直接创建kafka stream
  // 1.接收Kafka中数据，生成相应DStream
  val kafkaParams = Map[String, String]("metadata.broker.list" -> brokers)
  val lines = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
    ssc, kafkaParams, topicsSet).map(_._2)

  // 2.获取每一个行的字段属性
  val records = lines.map(getRecord)

  // 3.筛选女性网民上网时间数据信息
  val femaleRecords = records.filter(_._2 == "female")
    .map(x => (x._1, x._3))

  // 4.汇总在一个时间窗口内每个女性上网时间
  val aggregateRecords = femaleRecords
    .reduceByKeyAndWindow(_ + _, _ - _, windowDuration)

  // 5.筛选连续上网时间超过阈值的用户，并获取结果
  aggregateRecords.filter(_._2 > 0.9 * windowTime.toInt).print()

  // 6.Streaming系统启动
  ssc.start()
  ssc.awaitTermination()
```

上述代码会引用以下函数

```
// 获取字段函数
def getRecord(line: String): (String, String, Int) = {
  val elems = line.split(",")
  val name = elems(0)
  val sexy = elems(1)
  val time = elems(2).toInt
  (name, sexy, time)
}
```

Spark Streaming Write To Kafka 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.DstreamKafkaWriter：

说明

- Spark版本升级后，推荐使用新接口createDirectStream，老接口createStream仍然存在，但是性能和稳定性差，建议不要使用老接口开发应用程序。
- 该样例代码只存在于mrs-sample-project-1.6.0.zip中

```
  // 参数解析:
  //<groupId> 消费者的group.id.
  //<brokers> broker的IP和端口.
  //<topic> kafka的topic.
  if (args.length != 3) {
    System.err.println("Usage: DstreamKafkaWriter <groupId> <brokers> <topic>")
    System.exit(1)
  }
```

```
val Array(groupId, brokers, topic) = args
val sparkConf = new SparkConf().setAppName("KafkaWriter")

// 配置Kafka
val kafkaParams = new Properties()
kafkaParams.put("metadata.broker.list", brokers)
kafkaParams.put("group.id", groupId)
kafkaParams.put("auto.offset.reset", "smallest")

// 创建一个Java streaming context
val ssc = new StreamingContext(sparkConf, Milliseconds(500))

// 向kafka发送数据
val sentData = Seq("kafka_writer_test_msg_01", "kafka_writer_test_msg_02",
  "kafka_writer_test_msg_03")

// 创建RDD queue
val sent = new mutable.Queue[RDD[String]]()
sent.enqueue(ssc.sparkContext.makeRDD(sentData))

// 使用写入的数据创建Dstream
val wStream = ssc.queueStream(sent)

// 写入Kafka
wStream.writeToKafka(kafkaParams,
  (x: String) => new KeyedMessage[String, Array[Byte]](topic, x.getBytes))

ssc.start()
ssc.awaitTermination()
```

8.3.4 通过 JDBC 访问 Spark SQL 的程序

8.3.4.1 场景说明

场景说明

用户自定义JDBCServer的客户端，使用JDBC连接来进行数据表的创建、数据加载、查询和删除。

数据规划

步骤1 确保以HA模式启动了JDBCServer服务，并至少有一个实例对外服务。在hdfs上创建"/home/data"目录，新增包含如下内容的文件并上传到hdfs的"/home/data"目录下。

```
Miranda,32
Karlie,23
Candice,27
```

步骤2 确保其对启动JDBCServer的用户有读写权限。

步骤3 确保$SPARK_HOME/conf下有"hive-site.xml文件，且根据实际集群情况配置所需要的参数。

示例

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration>
   <property>
      <name>spark.thriftserver.ha.enabled</name>
      <value>true</value>
   </property>
</configuration>
```

步骤4 将代码的ThriftServerQueriesTest类中principal的值改为集群中$SPARK_HOME/conf/spark-defaults.conf配置文件中配置项spark.beeline.principal的值。

----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。

8.3.4.2 Java 样例代码

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 ',"");
sqlList.add("LOAD DATA INPATH '/home/data' INTO TABLE CHILD");
sqlList.add("SELECT * FROM child");
sqlList.add("DROP TABLE child");
executeSql(url, sqlList);
```

说明

- 样例工程中的data文件需要放到HDFS上的home目录下
- 保证data文件和创建的表的所属的用户和用户组保持一致

步骤2 拼接JDBC URL。

说明

HA模式下url的host和port必须为“ha-cluster”。

普通集群需要将样例代码中com.huawei.bigdata.spark.examples.ThriftServerQueriesTest类中第67、68行代码

StringBuilder sb = new StringBuilder("jdbc:hive2://ha-cluster/default"

+ securityConfig);

改为StringBuilder sb = new StringBuilder("jdbc:hive2://ha-cluster/default");

```
String HA_CLUSTER_URL = "ha-cluster";
StringBuilder sb = new StringBuilder("jdbc:hive2://" + HA_CLUSTER_URL + "/default;");
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。

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

8.3.4.3 Scala 样例代码

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 ','"
sqlList += "LOAD DATA INPATH '/home/data' INTO TABLE CHILD"
```

```
sqlList += "SELECT * FROM child"
sqlList += "DROP TABLE child"
```

说明

- 样例工程中的data文件需要放到JDBCServer所在机器的home目录下
- 保证本地的data文件和创建的表的所属的用户和用户组保持一致

步骤2 拼接JDBC URL。

说明

HA模式下url的host和port必须为“ha-cluster”。

普通集群需要将样例代码中com.huawei.bigdata.spark.examples.ThriftServerQueriesTest.scala中第61、62行代码

val sb = new StringBuilder("jdbc:hive2://ha-cluster/default"

+ securityConfig)

改为val sb = new StringBuilder("jdbc:hive2://ha-cluster/default");

```
val HA_CLUSTER_URL = "ha-cluster"
val sb = new StringBuilder(s"jdbc:hive2://$HA_CLUSTER_URL/default;")
val 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。

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）。

```
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

8.3.4.4 Python 样例代码

功能简介

通过连接zookeeper上的对应znode获取到当前主JDBCServer的IP和PORT，然后使用pyhive连接到这个JDBCServer，从而实现在JDBCServer-ha模式下，出现主备倒换后不需要修改代码依旧就能直接访问新的主JDBCServer服务。

该功能仅支持普通集群（未开启Kerberos认证的集群）使用。

环境准备

1. 安装支持环境。（开发环境请参考**环境简介**准备）

   执行以下命令安装编译工具：

   yum install cyrus-sasl-devel -y

   yum install gcc-c++ -y

2. 安装相应的python模块。

   需要安装sasl，thrift，thrift-sasl，PyHive。

   pip install sasl

   pip install thrift

   pip install thrift-sasl

   pip install PyHive

3. 安装python连接zookeeper工具。

   pip install kazoo

4. 从MRS集群上获取相应参数。
   - zookeeper的IP和PORT：

     可以查看配置文件/opt/client/Spark/spark/conf/hive-site.xml中的配置项spark.deploy.zookeeper.url
   - zookeeper 上存放JDBCServer主节点的IP和PORT：

     可以查看配置文件/opt/client/Spark/spark/conf/hive-site.xml中的配置项spark.thriftserver.zookeeper.dir（默认是/thriftserver），在此znode子节点（active_thriftserver）上存放了JDBCServer主节点的IP和PORT

样例代码

```
from kazoo.client import KazooClient
zk = KazooClient(hosts='ZookeeperHost')
zk.start()
result=zk.get("/thriftserver/active_thriftserver")
result=result[0].decode('utf-8')
JDBCServerHost=result[0].split(":")[0]
JDBCServerPort=result[0].split(":")[1]
from pyhive import hive
conn = hive.Connection(host=JDBCServerHost, port=JDBCServerPort,database='default')
cursor=conn.cursor()
```

```
cursor.execute("select * from test")
for result in cursor.fetchall():
    print result
```

其中，**ZookeeperHost**使用**4**获取到的zookeeper IP和PORT替换。

8.3.5 Spark on HBase 程序

8.3.5.1 场景说明

场景说明

用户可以使用Spark调用HBase的接口来操作HBase表的功能。在Spark应用中，用户可以自由使用HBase的接口来实现创建表、读取表、往表中插入数据等操作。

数据规划

首先需要把数据文件放置在HDFS系统里。

1. 本地新建文本文件，将以下内容复制保存到input_data1.txt。
   ```
   20,30,40,xxx
   ```
2. 在HDFS上建立一个文件夹，“/tmp/input”，并上传input_data1.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/client

      kinit -kt '/opt/client/Spark/spark/conf/user.keytab' <用于认证的业务用户>

      说明

      user.keytab文件位置请根据自己实际路径填写。

   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*** *input_xxx.txt /tmp/input*，上传数据文件。

说明

如果开启了kerberos认证，需要将客户端的配置文件“spark-defaults.conf”中的配置项spark.yarn.security.credentials.hbase.enabled置为true。

开发思路

1. 创建HBase表。
2. 往HBase表中插入数据。
3. 通过Spark Application读取HBase表的数据。

8.3.5.2 Java 样例代码

功能简介

在Spark应用中，通过使用HBase接口来实现创建表，读取表，往表中插入数据等操作。

代码样例

下面代码片段仅为演示，具体代码参见SparkOnHbaseJavaExample：

样例：创建HBase表

```
public class TableCreation {
   public static void main (String[] args) throws IOException {

      SparkConf conf = new SparkConf().setAppName("CollectFemaleInfo");
      JavaSparkContext jsc = new JavaSparkContext(conf);
      Configuration hbConf = HBaseConfiguration.create(jsc.hadoopConfiguration());

      // 创建和hbase的连接通道
      Connection connection = ConnectionFactory.createConnection(hbConf);

      // 声明表的描述信息
      TableName userTable  = TableName.valueOf("shb1");
      HTableDescriptor tableDescr = new HTableDescriptor(userTable);
      tableDescr.addFamily(new HColumnDescriptor("info".getBytes()));

      // 创建表
      System.out.println("Creating table shb1. ");
      Admin admin = connection.getAdmin();
      if (admin.tableExists(userTable)) {
         admin.disableTable(userTable);
         admin.deleteTable(userTable);
      }
      admin.createTable(tableDescr);

      connection.close();
      jsc.stop();
      System.out.println("Done!");

   }
}
```

样例：在HBase表中插入数据

```
public class TableInputData {
   public static void main (String[] args) throws IOException {

      // 建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
      SparkConf conf = new SparkConf().setAppName("CollectFemaleInfo");
      JavaSparkContext jsc = new JavaSparkContext(conf);
      Configuration hbConf = HBaseConfiguration.create(jsc.hadoopConfiguration());

      // 声明表的信息
      Table table = null;
      String tableName = "shb1";
      byte[] familyName = Bytes.toBytes("info");
      Connection connection = null;

      try {
         // 获取hbase连接
         connection = ConnectionFactory.createConnection(hbConf);
         // 获取table对象
         table = connection.getTable(TableName.valueOf(tableName));
         List<Tuple4<String, String, String, String>> data = jsc.textFile(args[0]).map(
```

```
                new Function<String, Tuple4<String, String, String, String>>() {
                    @Override
                    public Tuple4<String, String, String, String> call(String s) throws Exception {
                        String[] tokens = s.split(",");

                        return new Tuple4<String, String, String, String>(tokens[0], tokens[1], tokens[2],
tokens[3]);
                    }
                }).collect();

            Integer i = 0;
            for(Tuple4<String, String, String, String> line:  data) {
                Put put = new Put(Bytes.toBytes("row" + i));
                put.addColumn(familyName, Bytes.toBytes("c11"), Bytes.toBytes(line._1()));
                put.addColumn(familyName, Bytes.toBytes("c12"), Bytes.toBytes(line._2()));
                put.addColumn(familyName, Bytes.toBytes("c13"), Bytes.toBytes(line._3()));
                put.addColumn(familyName, Bytes.toBytes("c14"), Bytes.toBytes(line._4()));
                i += 1;
                table.put(put);
            }

        } 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();
                 }
             }
             jsc.stop();
        }
    }
}
```

样例：读取HBase表数据

```
public class TableOutputData {
    public static void main(String[] args) throws IOException {

        System.setProperty("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
        System.setProperty("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator");

        // 建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
        SparkConf conf = new SparkConf().setAppName("CollectFemaleInfo");
        JavaSparkContext jsc = new JavaSparkContext(conf);
        Configuration hbConf = HBaseConfiguration.create(jsc.hadoopConfiguration());

        // 声明要查的表的信息
        Scan scan = new org.apache.hadoop.hbase.client.Scan();
        scan.addFamily(Bytes.toBytes("info"));
        org.apache.hadoop.hbase.protobuf.generated.ClientProtos.Scan proto = ProtobufUtil.toScan(scan);
        String scanToString = Base64.encodeBytes(proto.toByteArray());
        hbConf.set(TableInputFormat.INPUT_TABLE, "shb1");
        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表中的每一行，并打印结果
```

```
        List<Tuple2<ImmutableBytesWritable, Result>> rddList = rdd.collect();
        for (int i = 0; i < rddList.size(); i++) {
            Tuple2<ImmutableBytesWritable, Result> t2 = rddList.get(i);
            ImmutableBytesWritable key = t2._1();
            Iterator<Cell> it = t2._2().listCells().iterator();
            while (it.hasNext()) {
                Cell c = it.next();
                String family = Bytes.toString(CellUtil.cloneFamily(c));
                String qualifier = Bytes.toString(CellUtil.cloneQualifier(c));
                String value = Bytes.toString(CellUtil.cloneValue(c));
                Long tm = c.getTimestamp();
                System.out.println(" Family=" + family + " Qualifier=" + qualifier + " Value=" + value + "
TimeStamp=" + tm);
            }
        }
        jsc.stop();
    }
}
```

8.3.5.3 Scala 样例代码

功能简介

在Spark应用中，通过使用HBase接口来实现创建表，读取表，往表中插入数据等操作。

代码样例

下面代码片段仅为演示，具体代码参见SparkOnHbaseScalaExample：

样例：创建HBase表

```
//建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
val conf: SparkConf = new SparkConf
val sc: SparkContext = new SparkContext(conf)
val hbConf: Configuration = HBaseConfiguration.create(sc.hadoopConfiguration)
//创建和hbase的连接通道
val connection: Connection = ConnectionFactory.createConnection(hbConf)

//声明表的描述信息
val userTable = TableName.valueOf("shb1")
val tableDescr = new HTableDescriptor(userTable)
tableDescr.addFamily(new HColumnDescriptor("info".getBytes))

//创建表
println("Creating table shb1. ")
val admin = connection.getAdmin
if (admin.tableExists(userTable)) {
  admin.disableTable(userTable)
  admin.deleteTable(userTable)
}
admin.createTable(tableDescr)

connection.close()
sc.stop()
println("Done!")
```

样例：在HBase表中插入数据

```
//建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
val conf = new SparkConf()
val sc = new SparkContext(conf)
val hbConf = HBaseConfiguration.create(sc.hadoopConfiguration)

//声明表的信息
val table: HTable = null
```

```
  val tableName = "shb1"
  val familyName = Bytes.toBytes("info");
  var connection: Connection = null
  try {
    //获取hbase连接
    connection = ConnectionFactory.createConnection(hbConf);
    //获取table对象
    val table = connection.getTable(TableName.valueOf(tableName));
    val data = sc.textFile(args(0)).map { line =>
      val value = line.split(",")
      (value(0), value(1), value(2), value(3))
    }.collect()

    var i = 0
    for (line <- data) {
      val put = new Put(Bytes.toBytes("row" + i));
      put.addColumn(familyName, Bytes.toBytes("c11"), Bytes.toBytes(line._1))
      put.addColumn(familyName, Bytes.toBytes("c12"), Bytes.toBytes(line._2))
      put.addColumn(familyName, Bytes.toBytes("c13"), Bytes.toBytes(line._3))
      put.addColumn(familyName, Bytes.toBytes("c14"), Bytes.toBytes(line._4))
      i += 1
      table.put(put)
    }
  } catch {
    case e: IOException =>
      e.printStackTrace();
  } finally {
    if (table != null) {
      try {
        // 关闭HTable对象
        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();
      }
    }
     sc.stop()
  }
```

样例：读取HBase表数据

```
  //建立连接hbase的配置参数，此时需要保证hbase-site.xml在classpath中
  val conf = new SparkConf()
  val sc = new SparkContext(conf)
  val hbConf = HBaseConfiguration.create(sc.hadoopConfiguration)

  //声明要查的表的信息
  val scan = new Scan()
  scan.addFamily(Bytes.toBytes("info"))
  val proto = ProtobufUtil.toScan(scan)
  val scanToString = Base64.encodeBytes(proto.toByteArray)
  hbConf.set(TableInputFormat.INPUT_TABLE, "shb1")
  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表中的每一行，并打印结果
  rdd.collect().foreach(x => {
    val key = x._1.toString
```

```
    val it = x._2.listCells().iterator()
    while (it.hasNext) {
      val c = it.next()
      val family = Bytes.toString(CellUtil.cloneFamily(c))
      val qualifier = Bytes.toString(CellUtil.cloneQualifier(c))
      val value = Bytes.toString(CellUtil.cloneValue(c))
      val tm = c.getTimestamp
      println(" Family=" + family + " Qualifier=" + qualifier + " Value=" + value + " TimeStamp=" + tm)
    }
  })
  sc.stop()
```

8.3.6 从 HBase 读取数据再写入 HBase

8.3.6.1 场景说明

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HBase shell工具，创建HBase table1和table2，并分别插入数据。

步骤1 通过HBase创建名为table1的表，命令如下。

create 'table1', 'cf'

步骤2 通过HBase插入数据，命令如下：

put 'table1', '1', 'cf:cid', '100'

步骤3 通过HBase创建名为table2的表，命令如下。

create 'table2', 'cf'

步骤4 通过HBase插入数据。

put 'table2', '1', 'cf:cid', '1000'

说明

如果开启了kerberos认证，需要将客户端的配置文件“spark-defaults.conf”和sparkJDBC服务端中的配置项spark.yarn.security.credentials.hbase.enabled置为true。

----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连接操作。
4. 把上一步骤的结果写到table2表。

8.3.6.2 Java 样例代码

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 {
    if (args.length < 1) {
      printUsage();
    }

    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
    final  String zkQuorum = args[0];
    rdd.foreachPartition(
      new VoidFunction<Iterator<Tuple2<ImmutableBytesWritable, Result>>>() {
        public void call(Iterator<Tuple2<ImmutableBytesWritable, Result>> iterator) throws Exception {
          hBaseWriter(iterator, zkQuorum);
        }
      }
    );

    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, String
zkQuorum) throws IOException {
   // 准备读取hbase
   String tableName = "table2";
   String columnFamily = "cf";
   String qualifier = "cid";
   Configuration conf = HBaseConfiguration.create();
   conf.set("hbase.zookeeper.property.clientPort", "24002");
   conf.set("hbase.zookeeper.quorum", zkQuorum);
   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();
      }
    }
  }
 }
 private static void printUsage() {
   System.out.println("Usage: {zkQuorum}");
   System.exit(1);
 }
}
```

8.3.6.3 Scala 样例代码

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]) {
   if (args.length < 1) {
     printUsage
   }

   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, args(0)))

   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)], zkQuorum: String): Unit = {
  // 准备读取hbase
  val tableName = "table2"
  val columnFamily = "cf"
  val qualifier = "cid"
  val conf = HBaseConfiguration.create()
  conf.set("hbase.zookeeper.property.clientPort", "24002")
  conf.set("hbase.zookeeper.quorum", zkQuorum)
  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()
      }
    }
  }
 }
 private def printUsage {
   System.out.println("Usage: {zkQuorum}")
   System.exit(1)
 }
}


/**
 * 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])
 }
}
```

8.3.7 从 Hive 读取数据再写入 HBase

8.3.7.1 场景说明

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/client

kinit -kt '/opt/client/Spark/spark/conf/user.keytab' <用于认证的业务用户>

说明

user.keytab文件位置请根据自己实际路径填写。

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ThriftServer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。

1. 通过HBase创建名为table2的表，命令如下。

create 'table2', 'cf'

2. 通过HBase插入数据，执行如下命令。

put 'table2', '1', 'cf:cid', '1000'

说明

如果开启了kerberos认证，需要将客户端的配置文件“spark-default.conf”和sparkJDBC服务端中的配置项spark.yarn.security.credentials.hbase.enabled置为true。

----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。

8.3.7.2 Java 样例代码

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 {
   if (args.length < 1) {
     printUsage();
   }

   // 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);
   DataFrame 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
   final String zkQuorum = args[0];
   dataFrame.toJavaRDD().foreachPartition(
     new VoidFunction<Iterator<Row>>() {
       public void call(Iterator<Row> iterator) throws Exception {
         hBaseWriter(iterator,zkQuorum);
       }
     }
   );

   jsc.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, String zkQuorum) throws IOException {
   // 读取hbase
   String tableName = "table2";
   String columnFamily = "cf";
   Configuration conf = HBaseConfiguration.create();
   conf.set("hbase.zookeeper.property.clientPort", "24002");
   conf.set("hbase.zookeeper.quorum", zkQuorum);
   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();
      }
    }
  }
 }


 private static void printUsage() {
   System.out.println("Usage: {zkQuorum}");
   System.exit(1);
 }
}
```

8.3.7.3 Scala 样例代码

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]) {
    if (args.length < 1) {
      printUsage
    }
    // 通过spark接口获取表中的数据
    val sparkConf = new SparkConf().setAppName("SparkHivetoHbase")
    val sc = new SparkContext(sparkConf)
    val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
    import sqlContext.implicits._
    val 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.rdd.foreachPartition(x => hBaseWriter(x, args(0)))
    sc.stop()
  }
  /**
   * 在executor端更新hbase表记录
   *
   * @param iterator hive表的partition数据
   */
  def hBaseWriter(iterator: Iterator[Row], zkQuorum: String): Unit = {
    // 读取hbase
    val tableName = "table2"
    val columnFamily = "cf"
    val conf = HBaseConfiguration.create()
    conf.set("hbase.zookeeper.property.clientPort", "24002")
    conf.set("hbase.zookeeper.quorum", zkQuorum)
    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()
      }
    }
  }
}

 private def printUsage {
   System.out.println("Usage: {zkQuorum}")
   System.exit(1)
 }
}
```

8.3.8 Streaming 从 Kafka 读取数据再写入 HBase

8.3.8.1 场景说明

场景说明

假定某个业务Kafka每30秒就会收到5个用户的消费记录。Hbase的table1表存储用户历史消费的金额信息。

现table1表有10条记录，表示有用户名分别为1-10的用户，用户的历史消费金额初始化都是0元。

基于某些业务要求，开发的Spark应用程序实现如下功能：

实时累加计算用户的消费金额信息：即用户总消费金额=用户的消费金额(kafka数据) + 用户历史消费金额(table1表的值)，更新到table1表。

数据规划

步骤1 创建HBase表，并插入数据。

1. 通过HBase创建名为table1的表，命令如下。

   create 'table1', 'cf'

2. 通过HBase执行如下命令，将数据插入table1表中。

```
put 'table1', '1', 'cf:cid', '0'
put 'table1', '2', 'cf:cid', '0'
put 'table1', '3', 'cf:cid', '0'
put 'table1', '4', 'cf:cid', '0'
put 'table1', '5', 'cf:cid', '0'
put 'table1', '6', 'cf:cid', '0'
put 'table1', '7', 'cf:cid', '0'
put 'table1', '8', 'cf:cid', '0'
put 'table1', '9', 'cf:cid', '0'
put 'table1', '10', 'cf:cid', '0'
```

步骤2 Spark Streaming样例工程的数据存储在Kafka中。

1. 确保集群安装完成，包括HDFS、Yarn、Spark。
2. 将kafka的Broker配置参数“allow.everyone.if.no.acl.found”的值修改为“true”(普通集群不需配置)。
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. 启动样例代码的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**编包并运行程序**章节中导出jar包的操作步骤。

   java -cp $SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient/*:{JAR_PATH} com.huawei.bigdata.spark.examples.streaming.StreamingExampleProducer {BrokerList} {Topic}

说明

- 如果开启了kerberos认证，需要将客户端的配置文件“spark-defaults.conf”和sparkJDBC服务端中的配置项spark.yarn.security.credentials.hbase.enabled置为true。
- {zkQuorum}格式为zkIp:2181。
- JAR_PATH为程序jar包所在路径。
- brokerlist格式为brokerIp:9092。

----结束

开发思路

1. 接收Kafka中数据，生成相应DStream。
2. 筛选数据信息并分析。
3. 找到对应的HBase表记录。
4. 计算结果，写到HBase表。

8.3.8.2 Java 样例代码

功能介绍

在Spark应用中，通过使用Streaming调用kafka接口来获取数据,然后把数据经过分析后，找到对应的HBase表记录，再写到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.SparkOnStreamingToHbase

```
/**
 * 运行streaming任务，根据value值从hbase table1表读取数据，把两者数据做操作后，更新到hbase table1表
 */
public class SparkOnStreamingToHbase {
  public static void main(String[] args) throws Exception {
    if (args.length < 4) {
```

```
    printUsage();
  }

  String checkPointDir = args[0];
  String topics = args[1];
  final String brokers = args[2];
  final String zkQuorum = args[3];

  Duration batchDuration = Durations.seconds(5);
  SparkConf sparkConf = new SparkConf().setAppName("SparkOnStreamingToHbase");
  JavaStreamingContext jssc = new JavaStreamingContext(sparkConf, batchDuration);

  // 设置Streaming的CheckPoint目录
  if (!"nocp".equals(checkPointDir)) {
    jssc.checkpoint(checkPointDir);
  }

  final String columnFamily = "cf";
  final String zkClientPort = "24002";
  HashMap<String, String> kafkaParams = new HashMap<String, String>();
  kafkaParams.put("metadata.broker.list", brokers);

  String[] topicArr = topics.split(",");
  Set<String> topicSet = new HashSet<String>(Arrays.asList(topicArr));

  // 通过brokers和topics直接创建kafka stream
  // 接收Kafka中数据，生成相应DStream
  JavaDStream<String> lines = KafkaUtils.createDirectStream(jssc, String.class, String.class,
    StringDecoder.class, StringDecoder.class, kafkaParams, topicSet).map(
    new Function<Tuple2<String, String>, String>() {
      public String call(Tuple2<String, String> tuple2) {
        // map(_._1)是消息的key, map(_._2)是消息的value
        return tuple2._2();
      }
    }
  );

  lines.foreachRDD(
    new Function<JavaRDD<String>, Void>() {
      public Void call(JavaRDD<String> rdd) throws Exception {
        rdd.foreachPartition(
          new VoidFunction<Iterator<String>>() {
            public void call(Iterator<String> iterator) throws Exception {
              hBaseWriter(iterator, zkClientPort, zkQuorum, columnFamily);
            }
          }
        );
        return null;
      }
    }
  );

  jssc.start();
  jssc.awaitTermination();
}

/**
 * 在executor端写入数据
 * @param iterator  消息
 * @param zkClientPort
 * @param zkQuorum
 * @param columnFamily
 */
 private static void hBaseWriter(Iterator<String> iterator, String zkClientPort, String zkQuorum, String
columnFamily) throws IOException {
   Configuration conf = HBaseConfiguration.create();
   conf.set("hbase.zookeeper.property.clientPort", zkClientPort);
   conf.set("hbase.zookeeper.quorum", zkQuorum);
   Connection connection = null;
```

```
    Table table = null;
    try {
      connection = ConnectionFactory.createConnection(conf);
      table = connection.getTable(TableName.valueOf("table1"));
      List<Get> rowList = new ArrayList<Get>();
      while (iterator.hasNext()) {
        Get get = new Get(iterator.next().getBytes());
        rowList.add(get);
      }
      // 获取table1的数据
      Result[] resultDataBuffer = table.get(rowList);
      // 设置table1的数据
      List<Put> putList = new ArrayList<Put>();
      for (int i = 0; i < resultDataBuffer.length; i++) {
        String row = new String(rowList.get(i).getRow());
        Result resultData = resultDataBuffer[i];
        if (!resultData.isEmpty()) {
          // 根据列簇和列，获取旧值
          String aCid = Bytes.toString(resultData.getValue(columnFamily.getBytes(), "cid".getBytes()));
          Put put = new Put(Bytes.toBytes(row));
          // 计算结果
          int resultValue = Integer.valueOf(row) + Integer.valueOf(aCid);
          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();
        }
      }
    }
  }


  private static void printUsage() {
    System.out.println("Usage: {checkPointDir} {topic} {brokerList} {zkQuorum}");
    System.exit(1);
  }
}
```

8.3.8.3 Scala 样例代码

功能介绍

在Spark应用中，通过使用Streaming调用kafka接口来获取数据，然后把数据经过分析后，找到对应的HBase表记录，再写到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.SparkOnStreamingToHbase

```
/**
 * 运行streaming任务，根据value值从hbase table1表读取数据，把两者数据做操作后，更新到hbase table1表
 */
object SparkOnStreamingToHbase {
  def main(args: Array[String]) {
    if (args.length < 4) {
      printUsage
    }

    val Array(checkPointDir, topics, brokers, zkQuorum) = args
    val sparkConf = new SparkConf().setAppName("DirectStreamToHbase")
    val ssc = new StreamingContext(sparkConf, Seconds(5))

    // 设置Streaming的CheckPoint目录
    if (!"nocp".equals(checkPointDir)) {
      ssc.checkpoint(checkPointDir)
    }

    val columnFamily = "cf"
    val zkClientPort = "24002"
    val kafkaParams = Map[String, String](
      "metadata.broker.list" -> brokers
    )

    val topicArr = topics.split(",")
    val topicSet = topicArr.toSet
    // map(_._1)是消息的key, map(_._2)是消息的value
    val lines = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](ssc, kafkaParams,
topicSet).map(_._2)
    lines.foreachRDD(rdd => {
      //partition运行在executor上
      rdd.foreachPartition(iterator => hBaseWriter(iterator, zkClientPort, zkQuorum, columnFamily))
    })

    ssc.start()
    ssc.awaitTermination()
  }


  /**
   * 在executor端写入数据
   * @param iterator  消息
   * @param zkClientPort
   * @param zkQuorum
   * @param columnFamily
   */
  def hBaseWriter(iterator: Iterator[String], zkClientPort: String, zkQuorum: String, columnFamily: String):
Unit = {
    val conf = HBaseConfiguration.create()
    conf.set("hbase.zookeeper.property.clientPort", zkClientPort)
    conf.set("hbase.zookeeper.quorum", zkQuorum)
    var table: Table = null
    var connection: Connection = null
    try {
      connection = ConnectionFactory.createConnection(conf)
      table = connection.getTable(TableName.valueOf("table1"))
      val iteratorArray = iterator.toArray
      val rowList = new util.ArrayList[Get]()
      for (row <- iteratorArray) {
        val get = new Get(row.getBytes)
        rowList.add(get)
      }
      // 获取table1的数据
      val resultDataBuffer = table.get(rowList)
```

```
    // 设置table1的数据
    val putList = new util.ArrayList[Put]()
    for (i <- 0 until iteratorArray.size) {
      val row = iteratorArray(i)
      val resultData = resultDataBuffer(i)
      if (!resultData.isEmpty) {
        // 根据列簇和列，获取旧值
        val aCid = Bytes.toString(resultData.getValue(columnFamily.getBytes, "cid".getBytes))
        val put = new Put(Bytes.toBytes(row))
        // 计算结果
        val resultValue = row.toInt + aCid.toInt
        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()
      }
    }
  }
 }


 private def printUsage {
   System.out.println("Usage: {checkPointDir} {topic} {brokerList} {zkQuorum}")
   System.exit(1)
 }
}
```

8.3.9 Spark Streaming 对接 kafka0-10 程序

8.3.9.1 场景说明

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”(普通集群不需配置)。
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 $SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:$KAFKA_HOME/libs/*:{JAR_PATH} com.huawei.bigdata.spark.examples.StreamingExampleProducer {BrokerList} {Topic}

   说明

   - JAR_PATH为程序jar包所在路径，BrokerList格式为brokerIp:9092。
   - 需要修改程序SecurityKafkaWordCount类中kerberos.domain.name的值为$KAFKA_HOME/config/consumer.properties文件中kerberos.domain.name配置项的值。
   - 若用户需要对接安全Kafka，则还需要在spark客户端的conf目录下的“jaas.conf”文件中增加“KafkaClient”的配置信息，示例如下：

     ```
     KafkaClient {
     com.sun.security.auth.module.Krb5LoginModule required
     useKeyTab=true
     keyTab = "./user.keytab"
     principal="leoB@HADOOP.COM"
     useTicketCache=false
     storeKey=true
     debug=true;
     };
     ```

     在Spark on YARN模式下，jaas.conf和user.keytab通过YARN分发到Spark on YARN的container目录下，因此KafkaClient中对于“keyTab”的配置路径必须为相对jaas.conf的所在路径，例如“./user.keytab”。principal修改为自己创建的用户名及集群域名。

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。

8.3.9.2 Java 样例代码

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。样例代码下载地址请参见**样例工程获取地址**。

说明

普通集群需要将样例代码中com.huawei.bigdata.spark.examples.SecurityKafkaWordCount类中第78行代码

“kafkaParams.put("security.protocol", "SASL_PLAINTEXT");”注释掉。

```
/**
 *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 batchSize = 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(batchSize) * 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.hadoop.com");

    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。
- 该样例代码只存在于mrs-sample-project-1.6.0.zip中。

```
/**
 * 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。
 */
public class JavaDstreamKafkaWriter {

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

    if (args.length != 4) {
      System.err.println("Usage: DstreamKafkaWriter <checkPointDir> <brokers> <topic>");
      System.exit(1);
    }

    String checkPointDir = args[0];
    String brokers = args[1];
    String topic = args[2];

    SparkConf sparkConf = new SparkConf().setAppName("KafkaWriter");

    //填写Kafka的properties。
    Map kafkaParams = new HashMap<String, Object>();
    kafkaParams.put("zookeeper.connect", brokers);
    kafkaParams.put("metadata.broker.list", brokers);
    kafkaParams.put("group.id", "dstreamKafkaWriterFt08");
    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<String>();
    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.mapAsScalaMapC
onverter(kafkaParams),
        new Function<String, ProducerRecord<String, byte[]>>() {
          public ProducerRecord<String, byte[]> call(String s) {
            return new ProducerRecord(topic, s.getBytes());
          }
        });

    ssc.start();
    ssc.awaitTermination();
  }
}
```

8.3.9.3 Scala 样例代码

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。

说明

普通集群需要将样例代码中com.huawei.bigdata.spark.examples.SecurityKafkaWordCount.scala类中第60行代码“"security.protocol" -> "SASL_PLAINTEXT",”注释掉。

```
/**
 *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.hadoop.com"
  );

  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。
- 该样例代码只存在于mrs-sample-project-1.6.0.zip中。

```
/**
 * 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()
  }
}
```

8.3.10 Structured Streaming 程序

8.3.10.1 场景说明

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”(普通集群不需配置)。
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**编包并运行程序**。

java -cp $SPARK_HOME/jars/*:$SPARK_HOME/jars/streamingClient010/*:{JAR_PATH} com.huawei.bigdata.spark.examples.KafkaWordCountProducer {BrokerList} {Topic} {messagesPerSec} {wordsPerMessage}

说明

- JAR_PATH为程序jar包所在路径; BrokerList格式为brokerIp:9092；
- 若用户需要对接安全Kafka，则还需要在spark客户端的conf目录下的“jaas.conf”文件中增加“KafkaClient”的配置信息，示例如下：

  ```
  KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab = "./user.keytab"
  principal="leoB@HADOOP.COM"
  useTicketCache=false
  storeKey=true
  debug=true;
  };
  ```

  在Spark on YARN模式下，jaas.conf和user.keytab通过YARN分发到Spark on YARN的container目录下，因此KafkaClient中对于“keyTab”的配置路径必须为相对jaas.conf的所在路径，例如“./user.keytab”。principal修改为自己创建的用户名及集群域名。

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。

8.3.10.2 Java 样例代码

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

- 普通集群需要将样例代码中com.huawei.bigdata.spark.examples.SecurityKafkaWordCount类中第61行代码“.option("kafka.security.protocol", protocol)”注释掉。
- 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。其默认值为“append”。

```
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();
 }
}
```

8.3.10.3 Scala 样例代码

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

- 普通集群需要将样例代码中com.huawei.bigdata.spark.examples.SecurityKafkaWordCount.scala中第49行代码“.option("kafka.security.protocol", protocol)”注释掉。
- 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。其默认值为“append”。

```
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()
 }
}
```

8.4 调测程序

8.4.1 编包并运行程序

操作场景

在程序代码完成开发后，您可以将打包好的jar包上传至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

- Spark应用程序只支持在Linux环境下运行，不支持在Windows环境下运行。
- 使用Python开发的Spark应用程序无需打包成jar，只需将样例工程拷贝到编译机器上即可。

运行 Spark Core 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark Core（Scala和Java）样例程序。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

其中，<inputPath>指HDFS文件系统中input的路径。

bin/spark-submit --class com.huawei.bigdata.spark.examples.*FemaleInfoCollection* ***--master yarn --deploy-mode client*** */opt/female/FemaleInfoCollection.jar <inputPath>*

----结束

运行 Spark SQL 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark SQL样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

其中，<inputPath>指HDFS文件系统中input的路径。

bin/spark-submit --class com.huawei.bigdata.spark.examples.*FemaleInfoCollection* ***--master yarn --deploy-mode client*** */opt/female/FemaleInfoCollection.jar <inputPath>*

----结束

运行 Spark Streaming 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark Streaming样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

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/streamingClient”。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 $SPARK_HOME/jars/streamingClient/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient/kafka_*.jar,$SPARK_HOME/jars/streamingClient/spark-streaming-kafka-0-8_*.jar*。

- Spark Streaming Write To Print代码样例

  bin/spark-submit --master yarn --deploy-mode client *--jars $SPARK_HOME/jars/streamingClient/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient/kafka_*.jar,$SPARK_HOME/jars/streamingClient/spark-streaming-kafka-*.jar* ***--class*** com.huawei.bigdata.spark.examples.*FemaleInfoCollectionPrint /opt/female/FemaleInfoCollectionPrint.jar <checkPointDir> <batchTime> <topics> <brokers>*

  说明

  - --jars中的jar版本名称根据集群实际情况而定。
  - brokers格式为brokerIp:9092。
  - <checkPointDir>指应用程序结果备份到HDFS的路径，<batchTime>指Streaming分批的处理间隔。

- Spark Streaming Write To Kafka代码样例

  bin/spark-submit --master yarn --deploy-mode client *--jars $SPARK_HOME/jars/streamingClient/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient/kafka_*.jar,$SPARK_HOME/jars/streamingClient/spark-streaming-kafka-*.jar* ***--***

  class *com.huawei.bigdata.spark.examples.DstreamKafkaWriter/opt/female/SparkStreamingExample-1.0.jar <groupId> <brokers> <topic>*

----结束

运行“通过 JDBC 访问 Spark SQL”样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取

的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行“通过JDBC访问Spark SQL”样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，使用java -cp命令运行代码。

java -cp ${SPARK_HOME}/jars/*:${SPARK_HOME}/conf:/opt/female/SparkThriftServerJavaExample-*.jar com.huawei.bigdata.spark.examples.ThriftServerQueriesTest ${SPARK_HOME}/conf/hive-site.xml ${SPARK_HOME}/conf/spark-defaults.conf

说明

普通集群需要注释掉安全配置部分代码，详情请参见**步骤2**和**步骤2**。

上面的命令行中，您可以根据不同样例工程，最小化选择其对应的运行依赖包。样例工程对应的运行依赖包详情，请参见**步骤1**。

----结束

运行“Spark on HBase”样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行“Spark on HBase”样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。运行样例程序时，程序运行顺序为：TableCreation、TableInputData、TableOutputData。

其中，在运行TableInputData样例程序时需要指定<inputPath>，<inputPath>指HDFS文件系统中input的路径。

bin/spark-submit --class com.huawei.bigdata.spark.examples.*TableInputData* ***--master yarn --deploy-mode client*** */opt/female/TableInputData.jar <inputPath>*

说明

spark任务在连接hbase读写数据的时候，如果开启了kerberos认证，需要将客户端的配置文件“spark-defaults.conf”中的配置项spark.yarn.security.credentials.hbase.enabled置为true。所有连接hbase读写数据spark任务均需修改该配置。

----结束

运行 Spark HBase to HBase 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark HBase to HBase样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

其中，在运行样例程序时需要指定<zkQuorum>，<zkQuorum>指ZooKeeper的IP地址。

bin/spark-submit --class com.huawei.bigdata.spark.examples.*SparkHbasetoHbase* ***--master yarn --deploy-mode client*** */opt/female/FemaleInfoCollection.jar* *<zkQuorum>*

----结束

运行 Spark Hive to HBase 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark Hive to HBase样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

在运行样例程序时需要指定<zkQuorum>，<zkQuorum>指ZooKeeper服务器ip地址。

bin/spark-submit --class com.huawei.bigdata.spark.examples.*SparkHivetoHbase* ***--master yarn --deploy-mode client*** */opt/female/FemaleInfoCollection.jar* *<zkQuorum>*

----结束

运行 Spark Streaming Kafka to HBase 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark Streaming Kafka to HBase样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

在运行样例程序时需要指定<checkPointDir><topic><brokerList>，其中<checkPointDir>指应用程序结果备份到HDFS的路径，<topic>指读取kafka上的topic名称，<brokerList>指Kafka服务器IP地址。

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“$SPARK_HOME/lib”，而Spark Streaming Kafka依赖包路径为“$SPARK_HOME/lib/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 $SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-streaming-kafka-*.jar*。

Spark Streaming To HBase代码样例

bin/spark-submit --master yarn --deploy-mode client --jars *$SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-streaming-kafka-0*.jar* ***--class*** com.huawei.bigdata.spark.examples.*streaming.SparkOnStreamingToHbase /opt/female/FemaleInfoCollectionPrint.jar <checkPointDir> <topic> <brokerList>*

说明

- --jars中的jar名称根据集群实际情况而定。
- brokerlist格式为brokerIp:9092。

----结束

运行 Spark Streaming 对接 Kafka0-10 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark Streaming 对接Kafka0-10样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

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，安全集群格式为brokerIp:21007，普通群格式为brokerIp:9092，<topic>指读取Kafka上的topic名称，<batchTime>指Streaming分批的处理间隔。

Spark Streaming读取Kafka 0-10代码样例：

- 安全集群任务提交命令：

  bin/spark-submit --master yarn --deploy-mode client --files ./conf/jaas.conf,./conf/user.keytab --driver-java-options "-Djava.security.auth.login.config=./jaas.conf" --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=./jaas.conf" --jars $SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-streaming-kafka-*.jar --class com.huawei.bigdata.spark.examples.SecurityKafkaWordCount /opt/SparkStreamingKafka010JavaExample-*.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。
  --driver-java-options "-Djava.security.auth.login.config=./jaas.conf" //指定driver侧jaas.conf文件路径，yarn-client模式下使用--driver-java-options "-Djava.security.auth.login.config"指定；yarn-cluster模式下使用--conf "spark.yarn.cluster.driver.extraJavaOptions"指定。
  --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=./jaas.conf"//指定executor侧jaas.conf文件路径。
  ```

- 普通集群任务提交命令：

  bin/spark-submit --master yarn --deploy-mode client --jars $SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-streaming-kafka-*.jar --class com.huawei.bigdata.spark.examples.SecurityKafkaWordCount /opt/SparkStreamingKafka010JavaExample-*.jar <checkpointDir> <brokers> <topic> <batchTime>

  Spark Streaming Write To Kafka 0-10代码样例(该样例只存在于mrs-sample-project-1.6.0.zip中)：

  bin/spark-submit --master yarn --deploy-mode client --jars $SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/

jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-streaming-kafka-*.jar --class com.huawei.bigdata.spark.examples.JavaDstreamKafkaWriter /opt/JavaDstreamKafkaWriter.jar <checkPointDir> <brokers> <topics>

----结束

运行 Spark Structured Streaming 样例程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 运行Spark Structured Streaming样例程序（Scala和Java语言）。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端的conf目录下，调用spark-submit脚本运行代码。

在运行样例程序时需要指定<brokers> <subscribe-type> <topic> <protocol> <service> <domain>，其中<brokers>指获取元数据的Kafka地址，<subscribe-type>指Kafka订阅类型（一般为subscribe，代表订阅指定的topic），<topic>指读取Kafka上的topic名称，<protocol>指安全访问协议，<service>指kerberos服务名称，<domain>指kerberos域名。

说明

普通集群需要注释掉配置kafka安全协议部分代码，详情请参见**Java样例代码**和**Scala样例代码**章节中的说明部分。

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 $SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-sql-kafka-*.jar。

Spark Structured Streaming 对接Kafka代码样例

- 安全集群任务提交命令：

  cd /opt/client/Spark/spark/conf

spark-submit --master yarn --deploy-mode client --files ./jaas.conf,./user.keytab --driver-java-options "-Djava.security.auth.login.config=./jaas.conf" --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=./jaas.conf" --jars $SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-sql-kafka-*.jar --class com.huawei.bigdata.spark.examples.SecurityKafkaWordCount /root/jars/SparkStructuredStreamingJavaExample-*.jar <brokers> <subscribe-type> <topic> <protocol> <service> <domain>

- 普通集群任务提交命令：

  spark-submit --master yarn --deploy-mode client --jars $SPARK_HOME/jars/streamingClient010/kafka-clients-*.jar,$SPARK_HOME/jars/streamingClient010/kafka_*.jar,$SPARK_HOME/jars/streamingClient010/spark-sql-kafka-*.jar --class com.huawei.bigdata.spark.examples.SecurityKafkaWordCount /root/jars/SparkStructuredStreamingJavaExample-*.jar <brokers> <subscribe-type> <topic> <protocol> <service> <domain>

  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。
  --driver-java-options "-Djava.security.auth.login.config=<local Path>/jaas.conf" //指定driver侧jaas.conf文件路径，yarn-client模式下使用--driver-java-options "-Djava.security.auth.login.config"指定；yarn-cluster模式下使用--conf "spark.yarn.cluster.driver.extraJavaOptions"指定。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。
  --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=./jaas.conf"  //指定executor侧jaas.conf文件路径。
  --jars中的jar名称根据集群实际情况而定。
  安全集群<brokers>格式为brokerIp:21007,<protocol> <service> <domain>可以参考$KAFKA_HOME/config/consumer.properties文件。
  普通集群<brokers>格式为brokerIp:9092, <domain>可以参考$KAFKA_HOME/config/consumer.properties文件，<protocol>用null代替，<service>为kafka。
  ```

----结束

提交 Python 语言开发的应用程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 提交Python语言开发的应用程序。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

进入Spark客户端目录，调用bin/spark-submit脚本运行代码。

其中，<inputPath>指HDFS文件系统中input的路径。

说明

由于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“spark.yarn.keytab”和“spark.yarn.principal”指定认证信息。

bin/spark-submit --master yarn --deploy-mode client --conf spark.yarn.keytab=/opt/FIclient/user.keytab --conf spark.yarn.principal=sparkuser */opt/female/SparkPythonExample/collectFemaleInfo.py <inputPath>*

----结束

提交 SparkLauncher 应用程序

步骤1 在工程目录下执行**mvn package**命令生成jar包，在工程目录target目录下获取，比如:FemaleInfoCollection.jar

步骤2 将生成的Jar包（如CollectFemaleInfo.jar）拷贝到Spark运行环境下（即Spark客户端），如“/opt/female”。开启Kerberos认证的安全集群下把从**准备开发用户**中获取的user.keytab和krb5.conf文件拷贝到Spark客户端conf目录下，如：/opt/client/Spark/spark/conf；未开启Kerberos认证集群可不必拷贝user.keytab和krb5.conf文件。

步骤3 提交SparkLauncher应用程序。

须知

- 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。
- 运行程序时可根据需要选择运行模式：
  - ***--deploy-mode client***：driver进程在客户端运行，运行结果在程序运行后直接输出。
  - ***--deploy-mode cluster***：driver进程在Yarn的ApplicationMaster（AM）中运行，运行结果和日志在Yarn的WebUI界面输出。

```
java -cp $SPARK_HOME/jars/*:{JAR_PATH}
com.huawei.bigdata.spark.examples.SparkLauncherExample yarn-client
{TARGET_JAR_PATH} { TARGET_JAR_MAIN_CLASS} {args}
```

说明

- JAR_PATH为SparkLauncher应用程序jar包所在路径。
- TARGET_JAR_PATH为待提交的spark application应用程序jar包所在路径。
- args为待提交的spark application应用程序的参数。

----结束

参考信息

“通过JDBC访问Spark SQL”样例程序（Scala和Java语言），其对应的运行依赖包如下：

- 通过JDBC访问Spark SQL样例工程（Scala）
  - commons-collections-<version>.jar
  - commons-configuration-<version>.jar
  - commons-io-<version>.jar
  - commons-lang-<version>.jar
  - commons-logging-<version>.jar
  - guava-<version>.jar
  - hadoop-auth-<version>.jar
  - hadoop-common-<version>.jar
  - hadoop-mapreduce-client-core-<version>.jar
  - hive-exec-<version>.spark2.jar
  - hive-jdbc-<version>.spark2.jar
  - hive-metastore-<version>.spark2.jar
  - hive-service-<version>.spark2.jar
  - httpclient-<version>.jar
  - httpcore-<version>.jar
  - libthrift-<version>.jar
  - log4j-<version>.jar
  - slf4j-api-<version>.jar
  - zookeeper-<version>.jar
  - scala-library-<version>.jar
- 通过JDBC访问Spark SQL样例工程（Java）
  - commons-collections-<version>.jar
  - commons-configuration-<version>.jar
  - commons-io-<version>.jar
  - commons-lang-<version>.jar
  - commons-logging-<version>.jar
  - guava-<version>.jar

- hadoop-auth-<version>.jar
- hadoop-common-<version>.jar
- hadoop-mapreduce-client-core-<version>.jar
- hive-exec-<version>.spark2.jar
- hive-jdbc-<version>.spark2.jar
- hive-metastore-<version>.spark2.jar
- hive-service-<version>.spark2.jar
- httpclient-<version>.jar
- httpcore-<version>.jar
- libthrift-<version>.jar
- log4j-<version>.jar
- slf4j-api-<version>.jar
- zookeeper-<version>.jar

8.4.2 查看调测结果

操作场景

Spark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Spark WebUI查看应用程序运行情况。

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与Spark应用程序指定，您可以通过指定文件中获取到运行结果数据。

- **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、Executors和SQL等部分。Streaming应用会多一个Streaming标签页。

    页面入口：请参考**登录MRS Manager**登录MRS Manager页面，选择“服务管理 > Yarn”，单击“ResourceManager Web UI”对应的“ResourceManager”进入Web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Spark UI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。

    页面入口：请参考**登录MRS Manager**登录MRS Manager页面，选择“服务管理 > Spark”，单击“Spark Web UI”对应的“JobHistory”进入Web界面。

- **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。

8.5 调优程序

8.5.1 Spark Core 调优

8.5.1.1 数据序列化

操作场景

Spark支持两种方式的序列化：

- Java原生序列化JavaSerializer
- Kryo序列化KryoSerializer

序列化对于Spark应用的性能来说，具有很大的影响。在特定的数据格式的情况下，KryoSerializer的性能可以达到JavaSerializer的10倍以上，而对于一些Int之类的基本类型数据，性能的提升就几乎可以忽略。

KryoSerializer依赖Twitter的Chill库来实现，相对于JavaSerializer，主要的问题在于不是所有的Java Serializable对象都能支持，兼容性不好，所以需要手动注册类。

序列化功能用在两个地方：序列化任务和序列化数据。Spark任务序列化只支持JavaSerializer，数据序列化支持JavaSerializer和KryoSerializer。

操作步骤

Spark程序运行时，在shuffle和RDD Cache等过程中，会有大量的数据需要序列化，默认使用JavaSerializer，通过配置让KryoSerializer作为数据序列化器来提升序列化性能。

在开发应用程序时，添加如下代码来使用KryoSerializer作为数据序列化器。

- 实现类注册器并手动注册类。

```
package com.etl.common;

import com.esotericsoftware.kryo.Kryo;
import org.apache.spark.serializer.KryoRegistrator;

public class DemoRegistrator implements KryoRegistrator
{
   @Override
   public void registerClasses(Kryo kryo)
   {
      //以下为示例类，请注册自定义的类
      kryo.register(AggrateKey.class);
      kryo.register(AggrateValue.class);
   }
}
```

  您可以在Spark客户端对spark.kryo.registrationRequired参数进行配置，设置是否需要Kryo注册序列化。

  当参数设置为true时，如果工程中存在未被序列化的类，则会抛出异常。如果设置为false（默认值），Kryo会自动将未注册的类名写到对应的对象中。此操作会对系统性能造成影响。设置为true时，用户需手动注册类，针对未序列化的类，系统不会自动写入类名，而是抛出异常，相对比false，其性能较好。

- 配置KryoSerializer作为数据序列化器和类注册器。
```
val conf = new SparkConf()
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
.set("spark.kryo.registrator", "com.etl.common.DemoRegistrator")
```

8.5.1.2 配置内存

操作场景

Spark是内存计算框架，计算过程中内存不够对Spark的执行效率影响很大。可以通过监控GC（Garbage Collection），评估内存中RDD的大小来判断内存是否变成性能瓶颈，并根据情况优化。

监控节点进程的GC情况（在客户端的conf/spark-defaults.conf配置文件中，在spark.driver.extraJavaOptions和spark.executor.extraJavaOptions配置项中添加参数："-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps"

），如果频繁出现Full GC，需要优化GC。把RDD做Cache操作，通过日志查看RDD在内存中的大小，如果数据太大，需要改变RDD的存储级别来优化。

操作步骤

- 优化GC，调整老年代和新生代的大小和比例。在客户端的conf/spark-defaults.conf配置文件中，在spark.driver.extraJavaOptions和spark.executor.extraJavaOptions配置项中添加参数：-XX:NewRatio。如，"-XX:NewRatio=2"，则新生代占整个堆空间的1/3，老年代占2/3。
- 开发Spark应用程序时，优化RDD的数据结构。
  - 使用原始类型数组替代集合类，如可使用fastutil库。
  - 避免嵌套结构。
  - Key尽量不要使用String。
- 开发Spark应用程序时，建议序列化RDD。

  RDD做cache时默认是不序列化数据的，可以通过设置存储级别来序列化RDD减小内存。例如：
```
testRDD.persist(StorageLevel.MEMORY_ONLY_SER)
```

8.5.1.3 设置并行度

操作场景

并行度控制任务的数量，影响shuffle操作后数据被切分成的块数。调整并行度让任务的数量和每个任务处理的数据与机器的处理能力达到最优。

查看CPU使用情况和内存占用情况，当任务和数据不是平均分布在各节点，而是集中在个别节点时，可以增大并行度使任务和数据更均匀的分布在各个节点。增加任务的并行度，充分利用集群机器的计算能力，一般并行度设置为集群CPU总和的2-3倍。

操作步骤

并行度可以通过如下三种方式来设置，用户可以根据实际的内存、CPU、数据以及应用程序逻辑的情况调整并行度参数。

- 在会产生shuffle的操作函数内设置并行度参数，优先级最高。
```
testRDD.groupByKey(24)
```

- 在代码中配置“spark.default.parallelism”设置并行度，优先级次之。
  ```
  val conf = new SparkConf()
  conf.set("spark.default.parallelism", 24)
  ```
- 在“$SPARK_HOME/conf/spark-defaults.conf”文件中配置“spark.default.parallelism”的值，优先级最低。
  ```
  spark.default.parallelism    24
  ```

8.5.1.4 使用广播变量

操作场景

Broadcast（广播）可以把数据集合分发到每一个节点上，Spark任务在执行过程中要使用这个数据集合时，就会在本地查找Broadcast过来的数据集合。如果不使用Broadcast，每次任务需要数据集合时，都会把数据序列化到任务里面，不但耗时，还使任务变得很大。

1. 每个任务分片在执行中都需要同一份数据集合时，就可以把公共数据集Broadcast到每个节点，让每个节点在本地都保存一份。
2. 大表和小表做join操作时可以把小表Broadcast到各个节点，从而就可以把join操作转变成普通的操作，减少了shuffle操作。

操作步骤

在开发应用程序时，添加如下代码，将“testArr”数据广播到各个节点。

```
def main(args: Array[String]) {
  ...
  val testArr: Array[Long] = new Array[Long](200)
  val testBroadcast: Broadcast[Array[Long]] = sc.broadcast(testArr)
  val resultRdd: RDD[Long] = inpputRdd.map(input => handleData(testBroadcast, input))
  ...
}

def handleData(broadcast: Broadcast[Array[Long]], input: String) {
  val value = broadcast.value
  ...
}
```

8.5.1.5 使用 External Shuffle Service 提升性能

操作场景

Spark系统在运行含shuffle过程的应用时，Executor进程除了运行task，还要负责写shuffle数据，给其他Executor提供shuffle数据。当Executor进程任务过重，导致GC而不能为其他Executor提供shuffle数据时，会影响任务运行。

External shuffle Service是长期存在于NodeManager进程中的一个辅助服务。通过该服务来抓取shuffle数据，减少了Executor的压力，在Executor GC的时候也不会影响其他Executor的任务运行。

操作步骤

1. 在NodeManager中启动External shuffle Service。
   a. 通过MRS Manager页面（可参考**登录MRS Manager**）的“服务管理 > Yarn > 服务配置”页面的“Yarn > 自定义”在“yarn-site.xml”中添加如下配置项：


```
<property>
   <name>yarn.nodemanager.aux-services</name>
   <value>spark_shuffle</value>
</property>
<property>
   <name>yarn.nodemanager.aux-services.spark_shuffle.class</name>
   <value>org.apache.spark.network.yarn.YarnShuffleService</value>
</property>
```

配置参数	描述
yarn.nodemanager.aux-services	NodeManager中一个长期运行的辅助服务，用于提升Shuffle计算性能。
yarn.nodemanager.aux-services.spark_shuffle.class	NodeManager中辅助服务对应的类。

b. 添加依赖的jar包。

拷贝“${SPARK_HOME}/lib/spark-1.5.1-yarn-shuffle.jar”到“${HADOOP_HOME}/share/hadoop/yarn/lib/”目录下。

c. 重启NodeManager进程，也就启动了External shuffle Service。

2. Spark应用使用External shuffle Service。

– 在客户端的安装目录“/Spark/spark/conf/spark-defaults.conf”中必须添加如下配置项：

```
spark.shuffle.service.enabled   true
spark.shuffle.service.port      7337
```

配置参数	描述
spark.shuffle.service.enabled	NodeManager中一个长期运行的辅助服务，用于提升Shuffle计算性能。默认为false，表示不启用该功能。
spark.shuffle.service.port	Shuffle服务监听数据获取请求的端口。可选配置，默认值为“7337”。

说明

1.如果“yarn.nodemanager.aux-services”配置项已存在，则在value中添加“spark_shuffle”，且用逗号和其他值分开。

2.“spark.shuffle.service.port”的值需要和上面“yarn-site.xml”中的值一样。

8.5.1.6 Yarn 模式下动态资源调度

操作场景

对于Spark应用来说，资源是影响Spark应用执行效率的一个重要因素。当一个长期运行的服务（比如JDBCServer），若分配给它多个Executor，可是却没有任何任务分配给它，而此时有其他的应用却资源紧张，这就造成了很大的资源浪费和资源不合理的调度。

动态资源调度就是为了解决这种场景，根据当前应用任务的负载情况，实时的增减Executor个数，从而实现动态分配资源，使整个Spark系统更加健康。

操作步骤

1. 需要先配置External shuffle service，具体请参考**使用External Shuffle Service提升性能**。
2. 在“spark-defaults.conf”配置文件中必须添加配置项“spark.dynamicAllocation.enabled”，并将该参数的值设置为“true”，表示开启动态资源调度功能。默认情况下关闭此功能。
3. 下面是一些可选配置，如**表8-7**所示。

表 8-7 动态资源调度参数

配置项	说明	默认值
spark.dynamicAllocation.minExecutors	最小Executor个数。	0
spark.dynamicAllocation.initialExecutors	初始Executor个数。	spark.dynamicAllocation.minExecutors
spark.dynamicAllocation.maxExecutors	最大executor个数。	Integer.MAX_VALUE
spark.dynamicAllocation.schedulerBacklogTimeout	调度第一次超时时间。	1(s)
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout	调度第二次及之后超时时间。	spark.dynamicAllocation.schedulerBacklogTimeout
spark.dynamicAllocation.executorIdleTimeout	普通Executor空闲超时时间。	60(s)
spark.dynamicAllocation.cachedExecutorIdleTimeout	含有cached blocks的Executor空闲超时时间。	Integer.MAX_VALUE

说明

- 使用动态资源调度功能，必须配置External Shuffle Service。如果没有使用External Shuffle Service，Executor被杀时会丢失shuffle文件。
- 如果通过spark.executor.instances或者--num-executors指定了Executor的个数，即使配置了动态资源调度功能，动态资源调度功能也不会生效。
- 当前动态资源分配功能开启后，不能完全避免task被分配到即将要移除的executor，但是一般情况下只会导致该task失败，只有同一个task失败4次（可通过spark.task.maxFailures配置）才会导致job失败，所以正常情况下基本不会因为task被分配到即将要移除的executor导致job失败，并且可以通过调大spark.task.maxFailures来减小问题发生的概率。

8.5.1.7 配置进程参数

操作场景

Spark on YARN模式下，有Driver、ApplicationMaster、Executor三种进程。在任务调度和运行的过程中，Driver和Executor承担了很大的责任，而ApplicationMaster主要负责container的启停。

因而Driver和Executor的参数配置对spark应用的执行有着很大的影响意义。用户可通过如下操作对Spark集群性能做优化。

操作步骤

步骤1 配置Driver内存。

Driver负责任务的调度，和Executor、AM之间的消息通信。当任务数变多，任务平行度增大时，Driver内存都需要相应增大。

您可以根据实际任务数量的多少，为Driver设置一个合适的内存。

- 将“spark-defaults.conf”中的“spark.driver.memory”配置项或者“spark-env.sh”中的“SPARK_DRIVER_MEMORY”配置项设置为合适大小。
- 在使用spark-submit命令时，添加“--driver-memory MEM”参数设置内存。

步骤2 配置Executor个数。

每个Executor每个核同时能跑一个task，所以增加了Executor的个数相当于增大了任务的并发度。在资源充足的情况下，可以相应增加Executor的个数，以提高运行效率。

- 将“spark-defaults.conf”中的“spark.executor.instance”配置项或者“spark-env.sh”中的“SPARK_EXECUTOR_INSTANCES”配置项设置为合适大小。您还可以设置动态资源调度功能进行优化。
- 在使用spark-submit命令时，添加“--num-executors NUM”参数设置Executor个数。

步骤3 配置Executor核数。

每个Executor多个核同时能跑多个task，相当于增大了任务的并发度。但是由于所有核共用Executor的内存，所以要在内存和核数之间做好平衡。

- 将“spark-defaults.conf”中的“spark.executor.cores”配置项或者“spark-env.sh”中的“SPARK_EXECUTOR_CORES”配置项设置为合适大小。
- 在使用spark-submit命令时，添加“--executor-cores NUM”参数设置核数。

步骤4 配置Executor内存。

Executor的内存主要用于任务执行、通信等。当一个任务很大的时候，可能需要较多资源，因而内存也可以做相应的增加；当一个任务较小运行较快时，就可以增大并发度减少内存。

- 将“spark-defaults.conf”中的“spark.executor.memory”配置项或者“spark-env.sh”中的“SPARK_EXECUTOR_MEMORY”配置项设置为合适大小。
- 在使用spark-submit命令时，添加“--executor-memory MEM”参数设置内存。

----结束

示例

- 在执行spark wordcount计算中。1.6T数据，250个executor。

  在默认参数下执行失败，出现Futures timed out和OOM错误。

  因为数据量大，task数多，而wordcount每个task都比较小，完成速度快。当task数多时driver端相应的一些对象就变大了，而且每个task完成时executor和driver都要通信，这就会导致由于内存不足，进程之间通信断连等问题。

  当把Driver的内存设置到4g时，应用成功跑完。
- 使用ThriftServer执行TPC-DS测试套，默认参数配置下也报了很多错误：Executor Lost等。而当配置Driver内存为30g，executor核数为2，executor个数为125，executor内存为6g时，所有任务才执行成功。

8.5.1.8 设计 DAG

操作场景

合理的设计程序结构，可以优化执行效率。在程序编写过程中要尽量减少shuffle操作，合并窄依赖操作。

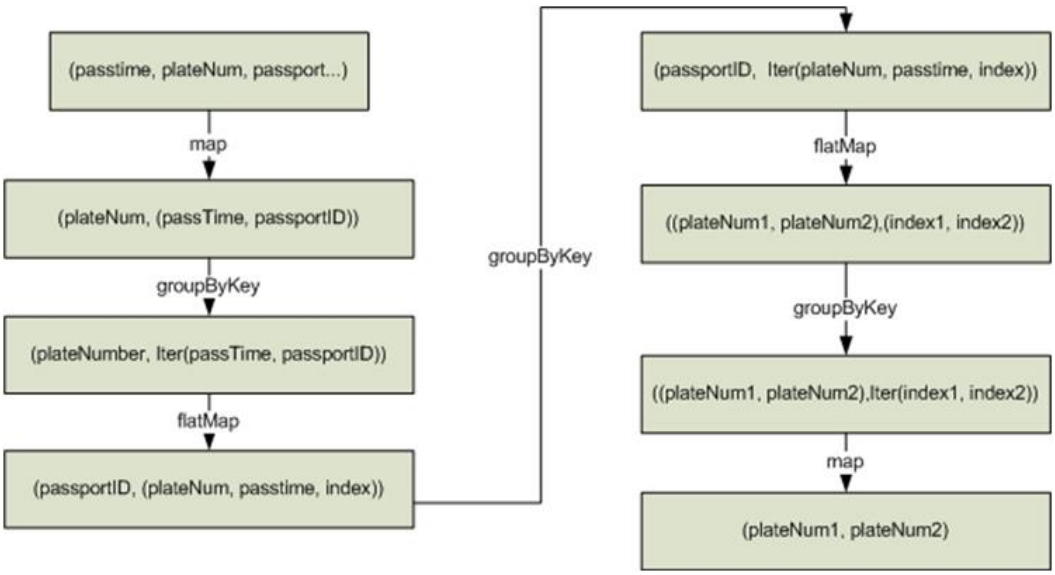
操作步骤

以“同行车判断”例子讲解DAG设计的思路。

- **数据格式：**通过收费站时间、车牌号、收费站编号.....
- **逻辑：**以下两种情况下判定这两辆车是同行车
  - 如果两辆车都通过相同序列的收费站，
  - 通过同一收费站之间的时间差小于一个特定的值。

该例子有两种实现模式，其中实现1的逻辑如图8-36所示，实现2的逻辑如图8-37所示。

图 8-36 实现 1 逻辑



实现1的逻辑说明：

1. 根据车牌号聚合该车通过的所有收费站并排序，处理后数据如下。

   车牌号1，[（通过时间，收费站3），（通过时间，收费站2），（通过时间，收费站4），（通过时间，收费站5）]

2. 标识该收费站是这辆车通过的第几个收费站。

   （收费站3，（车牌号1，通过时间，通过的第1个收费站））

   （收费站2，（车牌号1，通过时间，通过的第2个收费站））

   （收费站4，（车牌号1，通过时间，通过的第3个收费站））

   （收费站5，（车牌号1，通过时间，通过的第4个收费站））

3. 根据收费站聚合数据。

   收费站1，[（车牌号1，通过时间，通过的第1个收费站），（车牌号2，通过时间，通过的第5个收费站），（车牌号3，通过时间，通过的第2个收费站）]

4. 判断两辆车通过该收费站的时间差是否满足同行车的要求，如果满足则取出这两辆车。

   （车牌号1，车牌号2），（通过的第1个收费站，通过的第5个收费站）

   （车牌号1，车牌号3），（通过的第1个收费站，通过的第2个收费站）

5. 根据通过相同收费站的两辆车的车牌号聚合数据，如下。

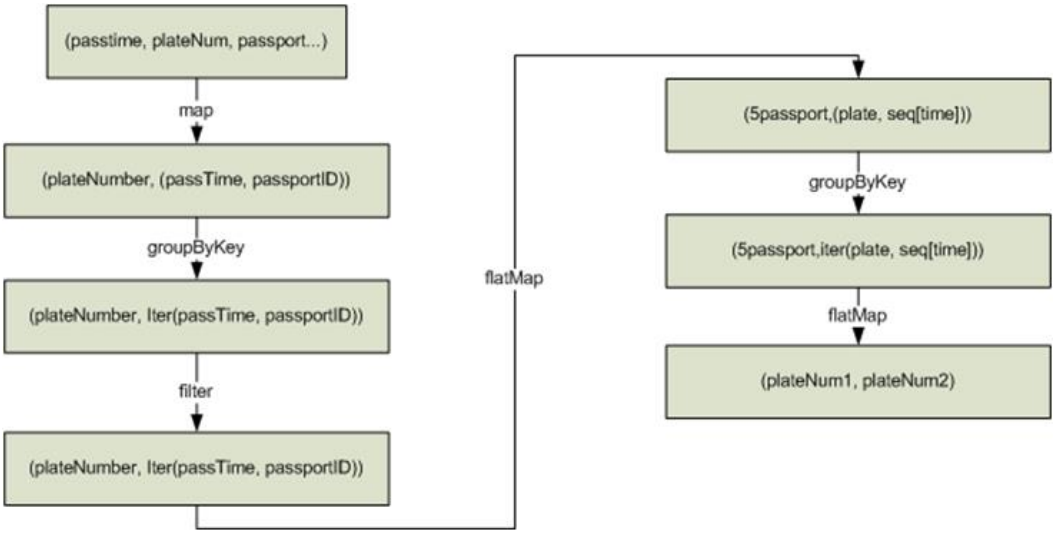
   （车牌号1，车牌号2），[（通过的第1个收费站，通过的第5个收费站），（通过的第2个收费站，通过的第6个收费站），（通过的第1个收费站，通过的第7个收费站），（通过的第3个收费站，通过的第8个收费站）]

6. 如果车牌号1和车牌号2通过相同收费站是顺序排列的（比如收费站3、4、5是车牌1通过的第1、2、3个收费站，是车牌2通过的第6、7、8个收费站）且数量大于同行车要求的数量则这两辆车是同行车。

实现1逻辑的缺点：

- 逻辑复杂
- 实现过程中shuffle操作过多，对性能影响较大。

图 8-37 实现 2 逻辑



实现2的逻辑说明：

1. 根据车牌号聚合该车通过的所有收费站并排序，处理后数据如下：

   车牌号1，[（通过时间，收费站3），（通过时间，收费站2），（通过时间，收费站4），（通过时间，收费站5）]

2. 根据同行车要通过的收费站数量（例子里为3）分段该车通过的收费站序列，如上面的数据被分解成：

   收费站3->收费站2->收费站4，（车牌号1，[收费站3时间，收费站2时间，收费站4时间]）

   收费站2->收费站4->收费站5，（车牌号1，[收费站2时间，收费站4时间，收费站5时间]）

3. 把通过相同收费站序列的车辆聚合，如下：

   收费站3->收费站2->收费站4，[（车牌号1，[收费站3时间，收费站2时间，收费站4时间]），（车牌号2，[收费站3时间，收费站2时间，收费站4时间]），（车牌号3，[收费站3时间，收费站2时间，收费站4时间]）]

4. 判断通过相同序列收费站的车辆通过相同收费站的时间差是不是满足同行车的要求，如果满足则说明是同行车。

实现2的优点如下：

- 简化了实现逻辑。
- 减少了一个groupByKey，也就减少了一次shuffle操作，提升了性能。

8.5.1.9 经验总结

使用 mapPartitions，按每个分区计算结果

如果每条记录的开销太大，例

```
rdd.map{x=>conn=getDBConn;conn.write(x.toString);conn.close}
```

则可以使用MapPartitions，按每个分区计算结果，如

```
rdd.mapPartitions(records => conn.getDBConn;for(item <- records)
write(item.toString); conn.close)
```

使用mapPartitions可以更灵活地操作数据，例如对一个很大的数据求TopN，当N不是很大时，可以先使用mapPartitions对每个partition求TopN，collect结果到本地之后再做排序取TopN。这样相比直接对全量数据做排序取TopN效率要高很多。

使用 coalesce 调整分片的数量

coalesce可以调整分片的数量。coalesce函数有两个参数

```
coalesce(numPartitions: Int, shuffle: Boolean = false)
```

当shuffle为true的时候，函数作用与repartition(numPartitions: Int)相同，会将数据通过Shuffle的方式重新分区；当shuffle为false的时候，则只是简单的将父RDD的多个partition合并到同一个task进行计算，shuffle为false时，如果numPartitions大于父RDD的切片数，那么分区不会重新调整。

遇到下列场景，可选择使用coalesce算子

- 当之前的操作有很多filter时，使用coalesce减少空运行的任务数量。此时使用coalesce(numPartitions, false)，numPartitions小于父RDD切片数。
- 当输入切片个数太大，导致程序无法正常运行时使用。

- 当任务数过大时候Shuffle压力太大导致程序挂住不动，或者出现linux资源受限的问题。此时需要对数据重新进行分区，使用coalesce(numPartitions, true)。

localDir 配置

Spark的Shuffle过程需要写本地磁盘，Shuffle是Spark性能的瓶颈，I/O是Shuffle的瓶颈。配置多个磁盘则可以并行的把数据写入磁盘。如果节点中挂载多个磁盘，则在每个磁盘配置一个Spark的localDir，这将有效分散Shuffle文件的存放，提高磁盘I/O的效率。如果只有一个磁盘，配置了多个目录，性能提升效果不明显。

Collect 小数据

大数据量不适用collect操作。

collect操作会将Executor的数据发送到Driver端，因此使用collect前需要确保Driver端内存足够，以免Driver进程发生OutOfMemory异常。当不确定数据量大小时，可使用saveAsTextFile等操作把数据写入HDFS中。只有在能够大致确定数据大小且driver内存充足的时候，才能使用collect。

使用 reduceByKey

reduceByKey会在Map端做本地聚合，使得Shuffle过程更加平缓，而groupByKey等Shuffle操作不会在Map端做聚合。因此能使用reduceByKey的地方尽量使用该算子，避免出现groupByKey().map(x=>(x._1,x._2.size))这类实现方式。

广播 map 代替数组

当每条记录需要查表，如果是Driver端用广播方式传递的数据，数据结构优先采用set/map而不是Iterator，因为Set/Map的查询速率接近O(1)，而Iterator是O(n)。

数据倾斜

当数据发生倾斜（某一部分数据量特别大），虽然没有GC（Garbage Collection，垃圾回收），但是task执行时间严重不一致。

- 需要重新设计key，以更小粒度的key使得task大小合理化。
- 修改并行度。

优化数据结构

- 把数据按列存放，读取数据时就可以只扫描需要的列。
- 使用Hash Shuffle时，通过设置spark.shuffle.consolidateFiles为true，来合并shuffle中间文件，减少shuffle文件的数量，减少文件IO操作以提升性能。最终文件数为reduce tasks数目。

8.5.2 SQL 和 DataFrame 调优

8.5.2.1 Spark SQL join 优化

操作场景

Spark SQL中，当对两个表进行join操作时，利用Broadcast特性（请参见**使用广播变量**），将小表BroadCast到各个节点上，从而转变成非shuffle操作，提高任务执行性能。

说明

这里join操作，只指inner join。

操作步骤

在Spark SQL中进行Join操作时，可以按照以下步骤进行优化。为了方便说明，设表A和表B，且A、B表都有个名为name的列。对A、B表进行join操作。

1. 估计表的大小。

   根据每次加载数据的大小，来估计表大小。

   也可以在Hive的数据库存储路径下直接查看表的大小。首先在Spark的配置文件“hive-site.xml”中，查看Hive的数据库路径的配置，默认为“/user/hive/warehouse”。

   ```
   <property>
     <name>hive.metastore.warehouse.dir</name>
    <value>/user/hive/warehouse</value>
   </property>
   ```

   然后通过hadoop命令查看对应表的大小。如查看表A的大小命令为：

   ```
   hadoop fs -du -s -h ${test.warehouse.dir}/a
   ```

   说明

   进行广播操作，对表有要求：

   1. 至少有一个表不是空表；
   2. 表不能是“external table”；
   3. 表的储存方式需为textfile（默认是textfile文件格式），如

      ```
      create table A( name string ) stored as textfile;
      或：
      create table A( name string );
      ```

2. 配置自动广播的阈值。

   Spark中，判断表是否广播的阈值为10485760（即10M）。如果两个表的大小至少有一个小于10M时，可以跳过该步骤。

   自动广播阈值的配置参数介绍，见**表8-8**。

   表 8-8 参数介绍

参数	默认值	描述
spark.sql.autoBroadcastJoinThreshold	10485760	当进行join操作时，配置广播的最大值；当表的字节数小于该值时便进行广播。当配置为-1时，将不进行广播。 参见**https://spark.apache.org/docs/latest/sql-programming-guide.html**

配置自动广播阈值的方法：

- 在Spark的配置文件“spark-defaults.conf”中，设置“spark.sql.autoBroadcastJoinThreshold”的值。其中，<size>根据场景而定，但要求该值至少比其中一个表大。
  ```
  spark.sql.autoBroadcastJoinThreshold = <size>
  ```
- 利用Hive CLI命令，设置阈值。在运行Join操作时，提前运行下面语句
  ```
  SET spark.sql.autoBroadcastJoinThreshold=<size>
  ```
  其中，<size>根据场景而定，但要求该值至少比其中一个表大。

3. 进行join操作。

   这时join的两个table，至少有个表是小于阈值的。

   如果A表和B表都小于阈值，且A表的字节数小于B表时，则运行B join A，如

   ```
   SELECT A.name FROM B JOIN A ON A.name = B.name;
   ```

   否则运行A join B。

   ```
   SELECT A.name FROM A JOIN B ON A.name = B.name;
   ```

8.5.2.2 INSERT...SELECT 操作调优

操作场景

在以下几种情况下，执行INSERT...SELECT操作可以进行一定的调优操作。

- 查询的数据是大量的小文件。
- 查询的数据是较多的大文件。
- 在beeline/thriftserver模式下使用非spark用户操作。

操作步骤

可对INSERT...SELECT操作做如下的调优操作。

- 如果建的是Hive表，将存储类型设为Parquet，从而减少执行INSERT...SELECT语句的时间。
- 建议使用spark-sql或者在beeline/thriftserver模式下使用spark用户来执行INSERT...SELECT操作，避免执行更改文件owner的操作，从而减少执行INSERT...SELECT语句的时间。

  说明

  在beeline/thriftserver模式下，executor的用户跟driver是一致的，driver是thriftserver服务的一部分，是由spark用户启动的，因此其用户也是spark用户，且当前无法实现在运行时将beeline端的用户透传到executor，因此使用非spark用户时需要对文件进行更改owner为beeline端的用户，即实际用户。

8.5.3 Spark Streaming 调优

操作场景

Streaming作为一种mini-batch方式的流式处理框架，它主要的特点是秒级时延和高吞吐量。因此Streaming调优的目标是在秒级延迟的情景下，提高Streaming的吞吐能力，在单位时间处理尽可能多的数据。

说明

本章节适用于输入数据源为Kafka的使用场景。

操作步骤

一个简单的流处理系统由以下三部分组件组成：数据源 + 接收器 + 处理器。数据源为Kafka，接受器为Streaming中的Kafka数据源接收器，处理器为Streaming。

对Streaming调优，就必须使三个部件的性能都最优化。

- **数据源调优**

  在实际的应用场景中，数据源为了保证数据的容错性，会将数据保存在本地磁盘中，而Streaming的计算结果往往全部在内存中完成，数据源很有可能成为流式系统的最大瓶颈点。

  对Kafka的性能调优，有以下几个点：

  - 使用Kafka-0.8.2以后版本，可以使用异步模式的新Producer接口。
  - 配置多个Broker的目录，设置多个IO线程，配置Topic合理的Partition个数。

  详情请参见Kafka开源文档中的“性能调优”部分：**http://kafka.apache.org/documentation.html**。

- **接收器调优**

  Streaming中已有多种数据源的接收器，例如Kafka、Flume、MQTT、ZeroMQ等，其中Kafka的接收器类型最多，也是最成熟一套接收器。

  Kafka包括三种模式的接收器API：

  - KafkaReceiver：直接接收Kafka数据，进程异常后，可能出现数据丢失。
  - ReliableKafkaReceiver：通过ZooKeeper记录接收数据位移。
  - DirectKafka：直接通过RDD读取Kafka每个Partition中的数据，数据高可靠。

  从实现上来看，DirectKafka的性能会是最好的，实际测试上来看，DirectKafka也确实比其他两个API性能好。因此推荐使用DirectKafka的API实现接收器。

  数据接收器作为一个Kafka的消费者，对于它的配置优化，请参见Kafka开源文档：**http://kafka.apache.org/documentation.html**。

- **处理器调优**

  Streaming的底层由Spark执行，因此大部分对于Spark的调优措施，都可以应用在Streaming之中，例如：

  - 数据序列化
  - 配置内存
  - 设置并行度
  - 使用External Shuffle Service提升性能

  说明

  在做Spark Streaming的性能优化时需注意一点，越追求性能上的优化，Streaming整体的可靠性会越差。例如：

  “spark.streaming.receiver.writeAheadLog.enable”配置为“false”的时候，会明显减少磁盘的操作，提高性能，但由于缺少WAL机制，会出现异常恢复时，数据丢失。

  因此，在调优Streaming的时候，这些保证数据可靠性的配置项，在生产环境中是不能关闭的。

8.5.4 Spark CBO 调优

操作场景

SQL语句转化为具体执行计划是由SQL查询编译器决定的，同一个SQL语句可以转化成多种物理执行计划，如何指导编译器选择效率最高的执行计划，这就是优化器的主要作用。传统数据库（例如Oracle）的优化器有两种：基于规则的优化器(Rule-Based Optimization,RBO)和基于代价的优化器(Cost-Based Optimization,CBO)。

- RBO

  RBO使用的规则是根据经验形成的，只要按照这个规则去写SQL语句，无论数据表中的内容怎样、数据分布如何，都不会影响到执行计划。

- CBO

  CBO是根据实际数据分布和组织情况，评估每个计划的执行代价，从而选择代价最小的执行计划。

目前Spark的优化器都是基于RBO的，已经有数十条优化规则，例如谓词下推、常量折叠、投影裁剪等，这些规则是有效的，但是它对数据是不敏感的。导致的问题是数据表中数据分布发生变化时，RBO是不感知的，基于RBO生成的执行计划不能确保是最优的。而CBO的重要作用就是能够根据实际数据分布估算出SQL语句，生成一组可能被使用的执行计划中代价最小的执行计划，从而提升性能。

目前CBO主要的优化点是Join算法选择。举个简单例子，当两个表做Join操作，如果其中一张原本很大的表经过Filter操作之后结果集小于BroadCast的阈值，在没有CBO情况下是无法感知大表过滤后变小的情况，采用的是SortMergeJoin算法，涉及到大量Shuffle操作，很耗费性能；在有CBO的情况下是可以感知到结果集的变化，采用的是BroadcastHashJoin算法，会将过滤后的小表BroadCast到每个节点，转变为非Shuffle操作，从而大大提高性能。

操作步骤

Spark CBO的设计思路是，基于表和列的统计信息，对各个操作算子（Operator）产生的中间结果集大小进行估算，最后根据估算的结果来选择最优的执行计划。

1. 设置配置项。
   - 在“spark-defaults.conf”配置文件中增加配置项“spark.sql.cbo”，将其设置为true，默认为false。
   - 在客户端执行SQL语句***set spark.sql.cbo=true***进行配置。
2. 执行统计信息生成命令，得到统计信息。

   说明

   此步骤只需在运行所有SQL前执行一次。如果数据集发生了变化（插入、更新或删除），为保证CBO的优化效果，需要对有变化的表或者列再次执行统计信息生成命令重新生成统计信息，以得到最新的数据分布情况。

   - 表：执行***COMPUTE STATS FOR TABLE src***命令计算表的统计信息，统计信息包括记录条数、文件数和物理存储总大小。
   - 列：
     - 执行***COMPUTE STATS FOR TABLE src ON COLUMNS***命令计算所有列的统计信息。
     - 执行***COMPUTE STATS FOR TABLE src ON COLUMNS*** *name,age*命令计算表中name和age两个字段的统计信息。

当前列的统计信息支持四种类型：数值类型、日期类型、时间类型和字符串类型。对于数值类型、日期类型和时间类型，统计信息包括：Max、Min、不同值个数(Number of Distinct Value,NDV)、空值个数(Number of Null)和Histogram（支持等宽、等高直方图）；对于字符串类型，统计信息包括：Max、Min、Max Length、Average Length、不同值个数(Number of Distinct Value,NDV)、空值个数(Number of Null)和Histogram（支持等宽直方图）。

3. CBO调优
   - 自动优化：用户根据自己的业务场景，输入SQL语句查询，程序会自动去判断输入的SQL语句是否符合优化的场景，从而自动选择Join优化算法。
   - 手动优化：用户可以通过***DESC FORMATTED src***命令查看统计信息，根据统计信息的分布，人工优化SQL语句。

8.6 Spark 接口

8.6.1 Java

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开源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拷贝。
- 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8-9和表8-10。

表 8-9 Transformation

方法	说明
<R> JavaRDD<R> map(Function<T,R> f)	对RDD中的每个element使用Function。

方法	说明
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，ascending为true时是升序否则为降序。
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 8-10 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。

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 8-11 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Streaming计算。
void awaitTermination()	当前进程等待终止，如Ctrl+C等。
void stop()	终止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 8-12 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 8-13 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。
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 8-14 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。

8.6.2 Scala

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开源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拷贝。
- 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8-15和表8-16所示。

表 8-15 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。

方法	说明
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。
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 8-16 Action

方法	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。

方法	说明
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。

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JavaSteamingContext，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 8-17 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]	用TCP协议（源主机:端口）创建一个输入流。
start():Unit	启动Streaming计算。
awaitTermination(timeout: long):Unit	当前进程等待终止，如Ctrl+C等。

方法	说明
stop(stopSparkContext: Boolean, stopGracefully: Boolean): Unit	终止Streaming计算。
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 8-18 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 8-19 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 8-20 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。

8.6.3 Python

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开源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拷贝。
- 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8-21和表8-22。

表 8-21 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。

方法	说明
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。
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 8-22 Action

方法	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。
- 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JavaSteamingContext，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 8-23 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Streaming计算。
awaitTermination(timeout)	当前进程等待终止，如Ctrl+C等。
stop(stopSparkContext, stopGraceFully)	终止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 8-24 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 8-25 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。

8.6.4 REST API

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，MRS版本中添加了查询SQL、JDBC/ODBC 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/2.2.2/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。

1. 确认服务端Spark组件已经安装，并正常运行。
2. 客户端运行环境已安装1.7或1.8版本的JDK。
3. 获取并解压缩安装包“MRS_Spark_Client.tar”。执行如下命令解压。

   tar -xvf MRS_Spark_Client.tar

   tar -xvf MRS_Spark_ClientConfig.tar

   说明

   由于不兼容老版本客户端，建议用户获取与服务端集群相同版本的客户端安装包进行安装部署。
4. 进入解压文件夹，即“MRS_Spark_ClientConfig”，执行下列命令安装客户端

   sh install.sh /opt/client

   其中“/opt/client”为用户自定义路径，此处仅为举例。
5. 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

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。

- 获取JobHistory中所有应用信息：
  - 命令：

    ```
    curl https://192.168.227.16:18080/api/v1/applications?mode=monitoring --insecure
    ```

    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_1478570725074_0042",
    "name" : "Spark-JDBCServer",
    "attempts" : [ {
    "startTime" : "2016-11-09T16:57:15.237CST",
    "endTime" : "2016-11-09T17:01:22.573CST",
    "lastUpdated" : "2016-11-09T17:01:22.614CST",
    "duration" : 247336,
    "sparkUser" : "spark",
    "completed" : true
    } ]
    }, {
    "id" : "application_1478570725074_0047-part1",
    "name" : "SparkSQL::192.168.169.84",
    "attempts" : [ {
    "startTime" : "2016-11-10T11:57:36.626CST",
    "endTime" : "1969-12-31T07:59:59.999CST",
    "lastUpdated" : "2016-11-10T11:57:48.613CST",
    "duration" : 0,
    "sparkUser" : "admin",
    "completed" : false
    } ]
    }]
    ```
  - 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 1** 。

表 8-26 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 https://192.168.227.16:18080/api/v1/applications/application_1478570725074_0042?mode=monitoring --insecure
    ```
    其中192.168.227.16为JobHistory节点的业务IP，18080为JobHistory的端口号，application_1478570725074_0042为应用的id。
  - 结果：
    ```
    {
    "id" : "application_1478570725074_0042",
    "name" : "Spark-JDBCServer",
    "attempts" : [ {
    "startTime" : "2016-11-09T16:57:15.237CST",
    "endTime" : "2016-11-09T17:01:22.573CST",
    "lastUpdated" : "2016-11-09T17:01:22.614CST",
    "duration" : 247336,
    "sparkUser" : "spark",
    "completed" : true
    } ]
    }
    ```
  - 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1**所示。
- 获取正在执行的某个应用的Executor信息：
  - 针对alive executor命令：
    ```
    curl https://192.168.169.84:26001/proxy/application_1478570725074_0046/api/v1/applications/application_1478570725074_0046/executors?mode=monitoring --insecure
    ```
  - 针对全部executor（alive&dead）命令：
    ```
    curl https://192.168.169.84:26001/proxy/application_1478570725074_0046/api/v1/applications/application_1478570725074_0046/allexecutors?mode=monitoring --insecure
    ```
    其中192.168.195.232为ResourceManager主节点的业务IP，26001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,
"activeTasks" : 0,
"failedTasks" : 0,
"completedTasks" : 0,
"totalTasks" : 0,
"totalDuration" : 0,
"totalInputBytes" : 0,
"totalShuffleRead" : 0,
"totalShuffleWrite" : 0,
"maxMemory" : 555755765,
"executorLogs" : {
"stdout" : "https://XTJ-224:26010/node/containerlogs/
container_1478570725074_0049_01_000002/admin/stdout?start=-4096",
"stderr" : "https://XTJ-224:26010/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 2**所示的常用信息。

  表 8-27 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 https://192.168.195.232:26001/proxy/application_1476947670799_0053/api/v1/applications/
    Spark-JDBCServerapplication_1476947670799_0053/SQL?mode=monitoring --insecure
    ```

    其中192.168.195.232为ResourceManager主节点的业务IP，26001为ResourceManager的端口号，application_1476947670799_0053为在YARN中的应用ID，Spark-JDBCServer是Spark应用的name。
  - JobHistory命令：

    ```
    curl https://192.168.227.16:22500/api/v1/applications/application_1478570725074_0004-part1/
    SQL?mode=monitoring --insecure
    ```

    其中192.168.227.16为JobHistory节点的业务IP，22500为JobHistory的端口号，application_1478570725074_0004-part1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3。

  表 8-28 SQL 的常用信息

参数	描述
id	SQL语句的ID
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/ODBC 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 https://192.168.195.232:26001/proxy/application_1476947670799_0053/api/v1/applications/application_1476947670799_0053/sqlserver?mode=monitoring --insecure
    ```

    其中192.168.195.232为ResourceManager主节点的业务IP，26001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/ODBC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 4**，每个SQL的信息如下**表8-30**。

表 8-29 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 8-30 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

参数	描述
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

- 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 https://192.168.195.232:26001/proxy/application_1477722033672_0008/api/v1/applications/NetworkWordCountapplication_1477722033672_0008/streaming?mode=monitoring --insecure
    ```
    其中192.168.195.232为ResourceManager主节点的业务IP，26001为ResourceManager的端口号，application_1477722033672_0008为在YARN中的应用ID，NetworkWordCount是Spark应用的name。
  - 结果：
    ```
    {
    "avgInputRate" : "0.00 events/sec",
    "avgSchedulingDelay" : "1 ms",
    "avgProcessingTime" : "72 ms",
    "avgTotalDelay" : "73 ms"
    }
    ```
  - 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，平均调度时延，平均执行时长，总时延平均值。

8.6.5 ThriftServer 接口介绍

简介

ThriftServer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。

ThriftServer是一个JDBC接口，用户可以通过JDBC连接ThriftServer来访问SparkSQL的数据。ThriftServer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。ThriftServer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ThriftServer的时候，至少要配置ThriftServer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。

ThriftServer默认在安装节点上的10000端口起一个JDBC服务，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ThriftServer的其他信息，请参见Spark官网：**http://spark.apache.org/docs/1.5.1/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**。

JDBC 客户端代码

通过JDBC客户端代码连接ThriftServer，来访问SparkSQL的数据。

增强特性

对比开源社区，MRS还提供了两个增强特性，ThriftServer HA方案和设置ThriftServer连接的超时时间。

- ThriftServer HA方案，当ThriftServer主节点发生故障时，备节点能够主动切换为主节点，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。

  连接HA模式下的ThriftServer，连接字符串和非HA模式下的区别在于需要将ip:port替换为ha-cluster，使用到的其他参数见表8-31。

  表 8-31 客户端参数列表

参数名称	含义	默认值
spark.thriftserver.ha.enabled	是否启用HA模式，设置为true表示启用。如果启用了HA，需要在连接字符串中将host:port修改为“ha-cluster”。否则会自动退出HA模式。	false
spark.thriftserver.zookeeper.dir	ThriftServer在ZooKeeper上存储元数据的路径，同服务端的同名参数，且需要和服务端配置一致。在此目录下存储命名为“active_thriftserver”的子目录，用于存储Hive ThriftServer的IP和端口号。	/thriftserver
spark.deploy.zookeeper.url	ZooKeeper的URL路径，同服务端的同名参数，且需要和服务端配置一致。	-
spark.thriftserver.retry.times	尝试连接服务端的最大次数。如果设置为负数或零，客户端将不会重新尝试连接服务端。	5
spark.thriftserver.retry.wait.time	重连服务端时的尝试时间间隔，单位秒。	10

  表8-31中的参数应配置在客户端classpath下的“hive-site.xml”文件中，例：

  ```
  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <configuration>
    <property>
      <name>spark.thriftserver.ha.enabled</name>
      <value>true</value>
    </property>
  </configuration>
  ```

  “spark.deploy.zookeeper.url”参数也可以通过在连接字符串中的“zk.quorum”代替，例：

  ```
  !connect jdbc:hive2://ha-cluster/default;zk.quorum=spark25:2181,spark
  26:2181,spark27:2181
  ```

- 设置客户端与ThriftServer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。

8.6.6 常用命令介绍

Spark命令详细的使用方法参考官方网站的描述：**http://spark.apache.org/docs/latest/quick-start.html**。

常用命令

Shell命令执行方法：

步骤1 进入Spark客户端目录。

步骤2 初始化环境变量。

source /opt/client/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。如果当前集群未启用Kerberos认证，则无需执行此命令。当前用户为**准备开发用户**时增加的开发用户。

kinit *MRS集群用户*

例如：

- 开发用户为“机机”用户时请执行：**kinit -kt user.keytab sparkuser**
- 开发用户为“人机”用户时请执行：**kinit sparkuser**

步骤4 执行Spark shell命令。

----结束

Spark常用的命令如下所示：

- ***spark-shell***

  提供了一个简单的调试工具，支持Scala语言。

  在shell控制台执行： ***spark-shell*** 即可进入Scala交互式界面，从HDFS中获取数据，再操作RDD进行计算，输出并打印结果。

  示例：一行代码可以实现统计一个文件中所有单词出现的频次。

  scala> sc.textFile("hdfs://hacluster/tmp/wordcount_data.txt").flatMap(line=> line.split(" ")).map(w => (w,1)).reduceByKey(_+_).collect()

- ***spark-submit***

  用于提交Spark应用到MRS集群中运行，并返回运行结果。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yarn-client。

spark-submit --class org.apache.spark.examples.GroupByTest --master yarn --deploy-mode client ${SPARK_HOME}/examples/jars/spark-examples_2.11-2.3.2-mrs-2.0.jar 6 3000 3000 3

- ***spark-sql***

  启动一个Spark应用，执行Spark SQL。可以指定local(--master local)或是集群模式(--master yarn)。

  启动示例：

  spark-sql --master yarn

  SQL示例：

  - ***SELECT key FROM src GROUP BY key;***
  - ***EXPLAIN EXTENDED SHOW TABLES;***

- ***spark-beeline***

  调用Spark的JDBCServer执行Spark SQL，可以实现对海量数据高效的计算和统计分析。JDBCServer包含一个长时运行的Spark任务，在spark-beeline中执行的语句都会交给该任务执行。

  启动示例：

  cd $SPARK_HOME/bin

  spark-beeline

  SQL示例：

  a. ***CREATE TABLE info(id int, name string, company string);***
  b. ***INSERT INTO TABLE info values(001,'jack','huawei');***
  c. ***SELECT * FROM info;***

- ***beeline***

  调用Spark的JDBCServer执行Spark SQL，可以实现对海量数据高效的计算和统计分析。JDBCServer包含一个长时运行的Spark任务，在beeline中执行的语句都会交给该任务执行。

  开启Kerberos认证的安全集群启动示例：

  cd $SPARK_HOME/bin

  ./beeline -u 'jdbc:hive2://ha-cluster/default;user.principal=spark/hadoop.COM;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.COM;'

  说明

  spark/hadoop.COM字符串在本集群上使用**klist -kt /opt/Bigdata/MRS_XXX/1_20_SparkResource/etc/spark.keytay**命令展示后的principal字符串获取，可粘贴到beeline命令中直接使用。

  未开启Kerberos认证的普通集群启动示例：

  cd $SPARK_HOME/bin

  beeline

  SQL示例：

  a. ***CREATE TABLE info(id int, name string, company string);***
  b. ***INSERT INTO TABLE info values(001,'jack','huawei');***

c. ***SELECT * FROM info;***

推荐使用 spark-beeline，因为spark-beeline是在beeline的基础上面做的封装，用户可直接运行spark-beeline。

- ***run-example***

  用来运行或者调试Spark开源社区中自带的样例代码。

  示例：执行SparkPi

  run-example --master yarn --deploy-mode client SparkPi 100

8.7 FAQ

8.7.1 如何添加自定义代码的依赖包

问题

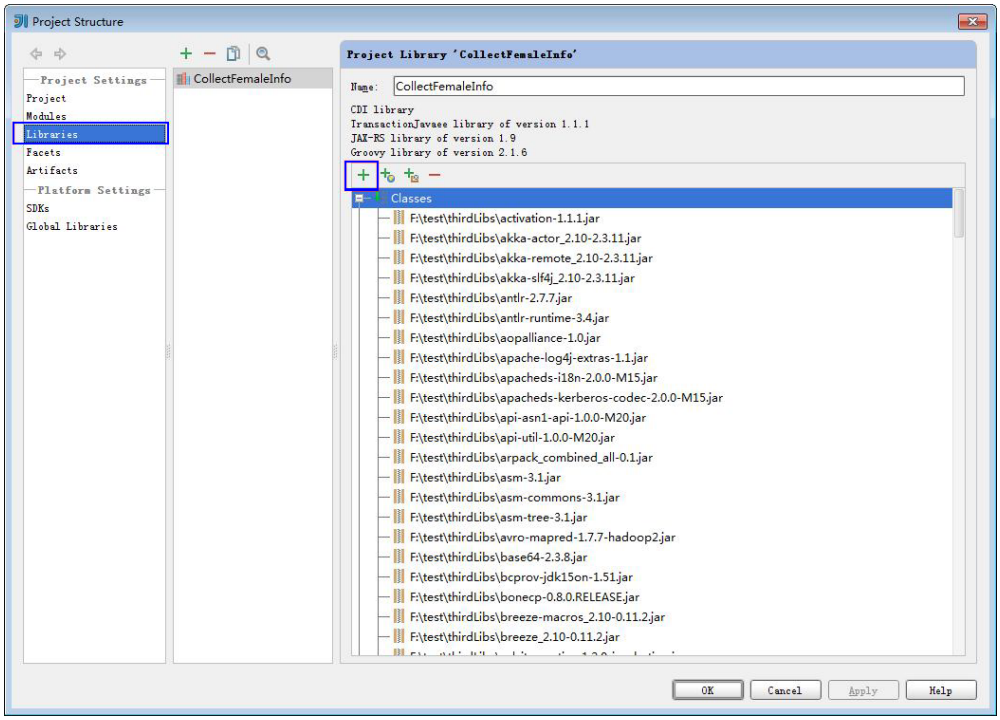
用户在开发Spark程序时，会添加样例程序外的自定义依赖包。针对自定义代码的依赖包，如何使用IDEA添加到工程中?

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 8-38 添加依赖包

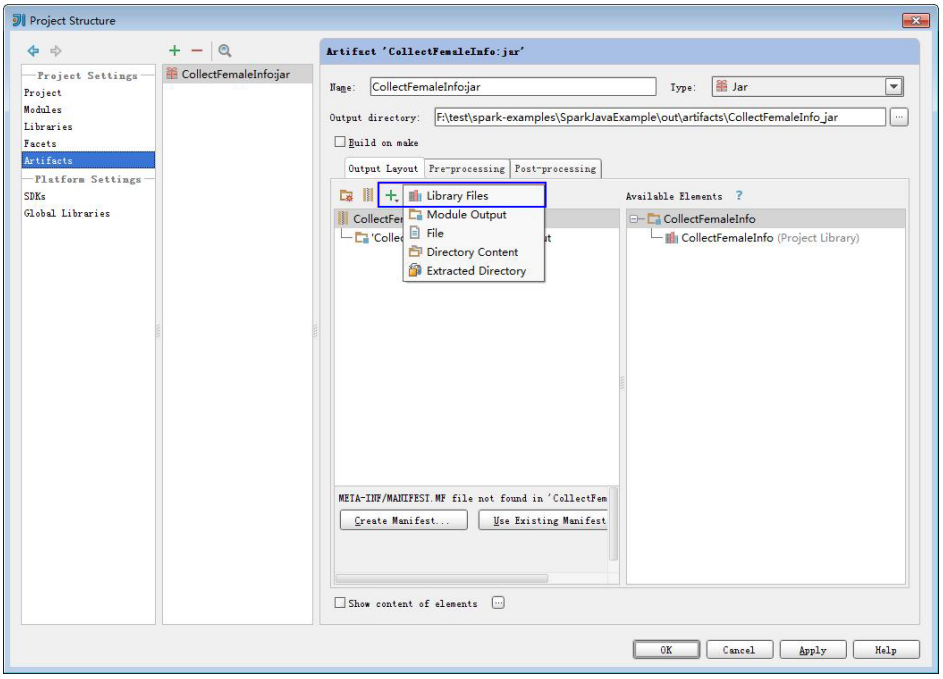


步骤3 单击“Apply”加载依赖包，然后单击“OK”完成配置。

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。

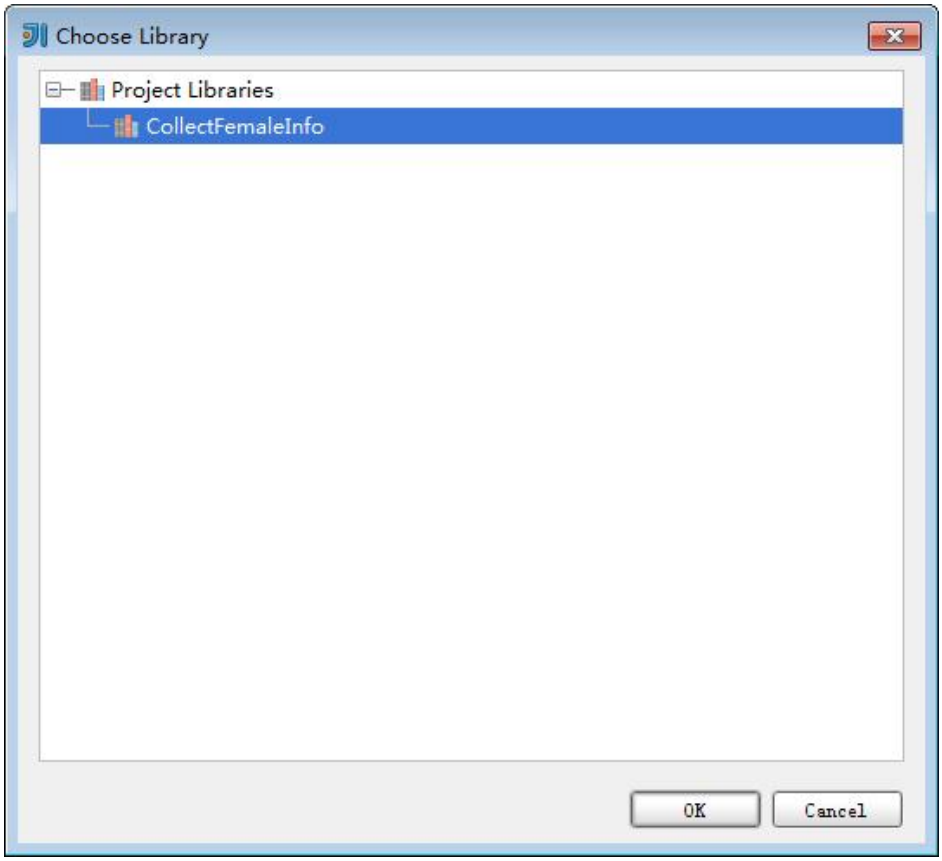
1. 在“Project Structure”页面，选择“Artifacts”页签。
2. 在右侧窗口中单击“+”，选择“Library Files”添加依赖包。

图 8-39 添加 Library Files



3. 选择需要添加的依赖包，然后单击“OK”。

图 8-40 Choose Library



4. 单击“Apply”加载依赖包，然后单击“OK”完成配置。

----结束

8.7.2 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”。

8.7.3 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/streamingClient”目录以及服务端的“/opt/Bigdata/MRS/FusionInsight-Spark-2.2.1/spark/jars/streamingClient”目录：

- kafka-clients-0.8.2.1.jar
- kafka_2.10-0.8.2.1.jar
- spark-streaming-kafka_2.10-1.5.1.jar

由于$SPARK_HOME/lib/streamingClient/*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：

--jars $SPARK_CLIENT_HOME/jars/streamingClient/kafka-clients-0.8.2.1.jar,$SPARK_CLIENT_HOME/jars/streamingClient/kafka_2.10-0.8.2.1.jar,$SPARK_CLIENT_HOME/jars/streamingClient/park-streaming-kafka_2.10-1.5.1.jar

用户自己开发的应用程序以及样例工程都支持上述参数。

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/lib/streamingClient/*”。
- 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“/opt/huawei/Bigdata/FusionInsight/spark/spark/lib/streamingClient/*”。
  - 将各服务端节点的“spark-examples_2.10-1.5.1.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”。

8.7.4 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：

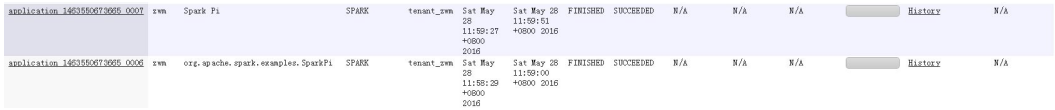
```
-XX:OnOutOfMemoryError='kill -9 %p'
```

8.7.5 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8-41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 8-41 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。

例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 lib/spark-examples*.jar 10*

8.7.6 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。

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。
   - SparkLauncher程序的编译依赖包为spark-launcher_2.10-1.5.1.jar。
   - 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/lib”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lib路径。文档中提供的示例代码，其运行依赖包在客户端lib中已存在。

   说明

   SparkLauncher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SparkLauncher方式提交。

   java -cp $SPARK_HOME/conf:$SPARK_HOME/lib/*:SparkLauncherExample.jar

com.huawei.bigdata.spark.examples.SparkLauncherExample yarn-client /opt/female/FemaleInfoCollection.jar com.huawei.bigdata.spark.examples.FemaleInfoCollection <inputPath>

----结束

8.7.7 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。

8.7.8 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：

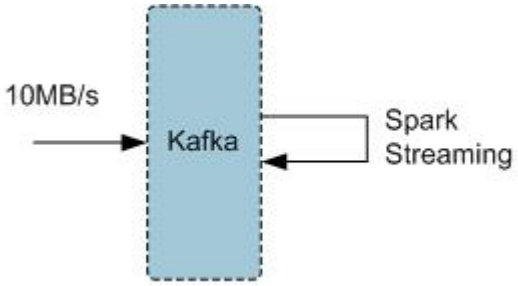
```
2016-03-02 17:46:19,017 | INFO | [kafka-network-thread-21005-1] | Closing socket connection to /
10.91.8.208 due to invalid request: Request of length
122371301 is not valid, it is larger than the maximum size of 104857600 bytes. | kafka.network.Processor
(Logging.scala:68)
2016-03-02 17:46:19,155 | 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 8-42 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MRS 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。

8.7.9 如何使用 IDEA 远程调试

问题

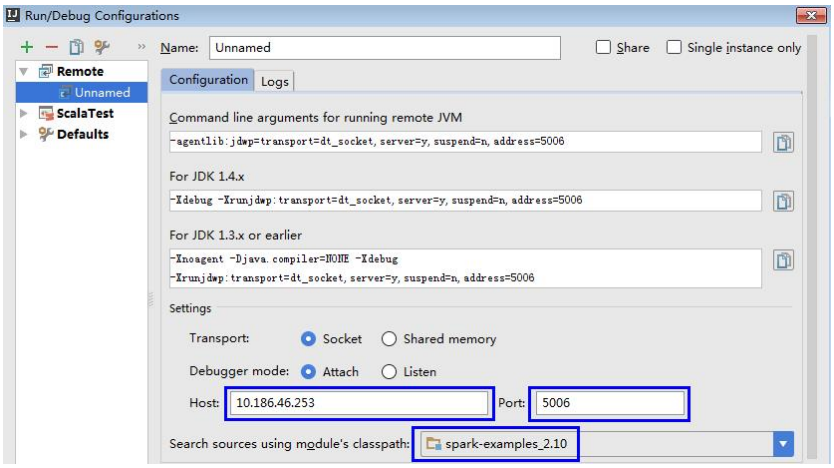
在Spark二次开发中如何使用IDEA远程调试?

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8-43所示。

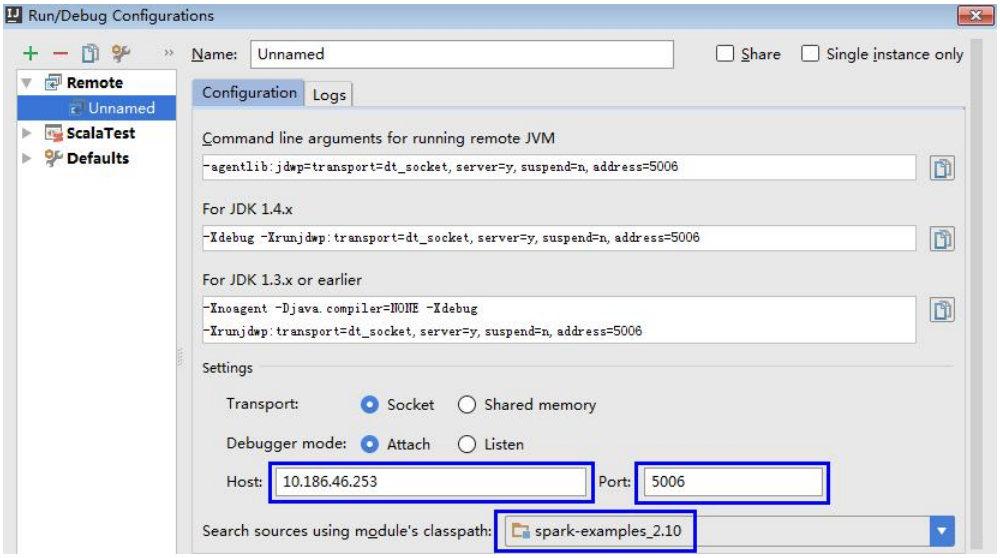
图 8-43 选择 Remote



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2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 8-44 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。

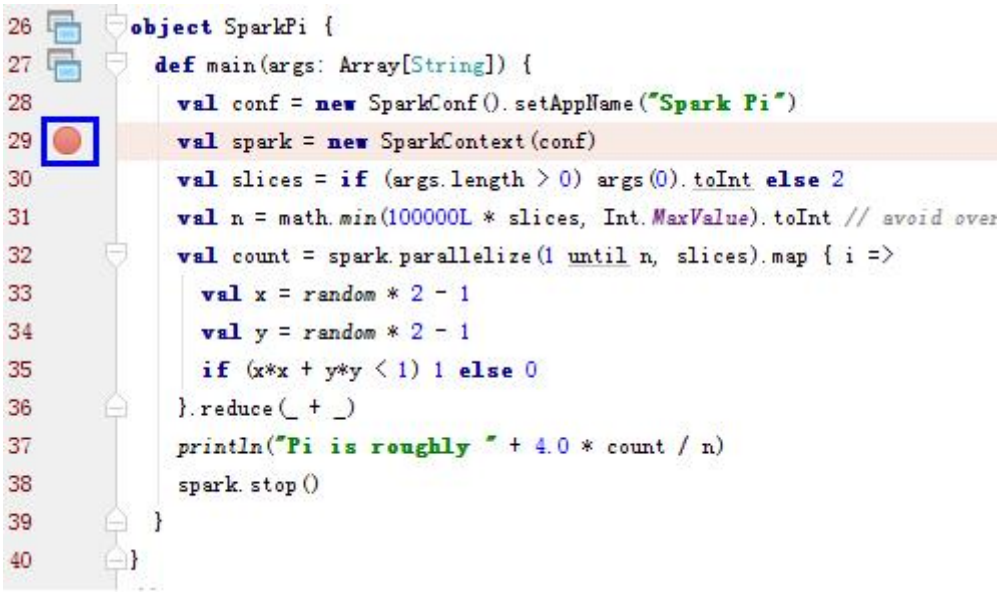
   ./spark-submit --master yarn-client --driver-java-options "-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006" --class org.apache.spark.examples.SparkPi /opt/client/Spark/spark/examples/jars/spark-examples-<version>.jar

   - org.apache.spark.examples.SparkPi，opt/client/Spark/spark/examples/jars/spark-examples-<version>.jar：用户调试时需要换成自己的主类和jar包路径。
   - -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5006：需要换成3获取到的For JDK1.4.x对应的调试端口。

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4所示，在SparkPi.scala的29行设置断点。

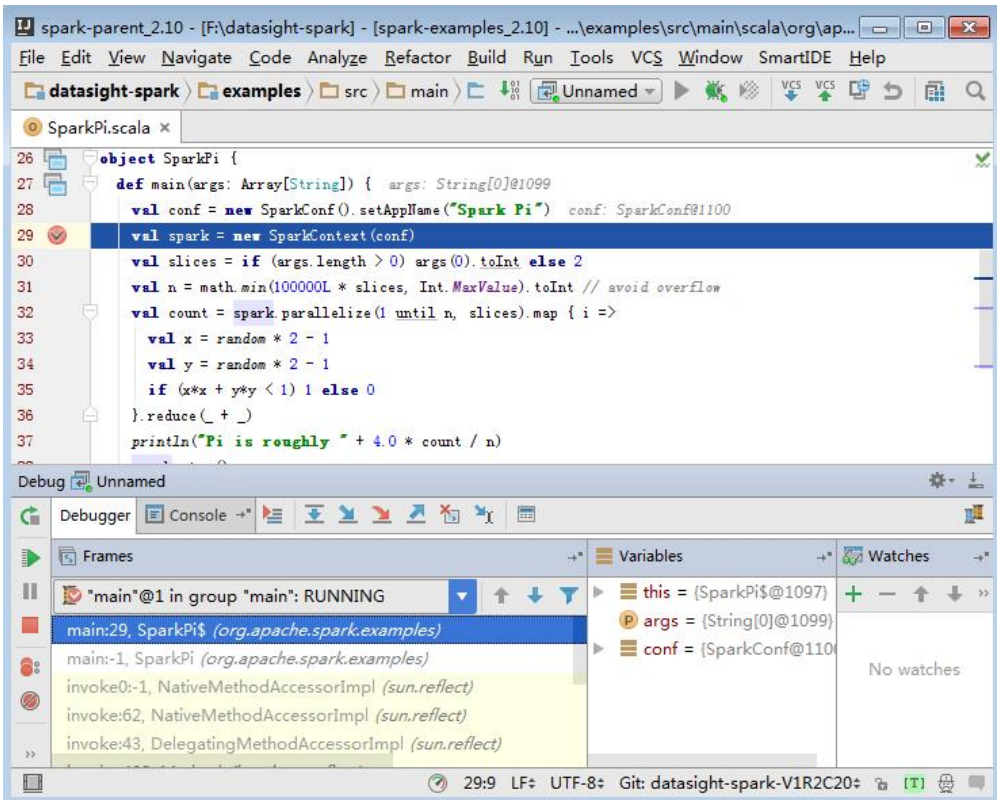
图 8-45 设置断点



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5所示。

图 8-46 调试



8.7.10 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。

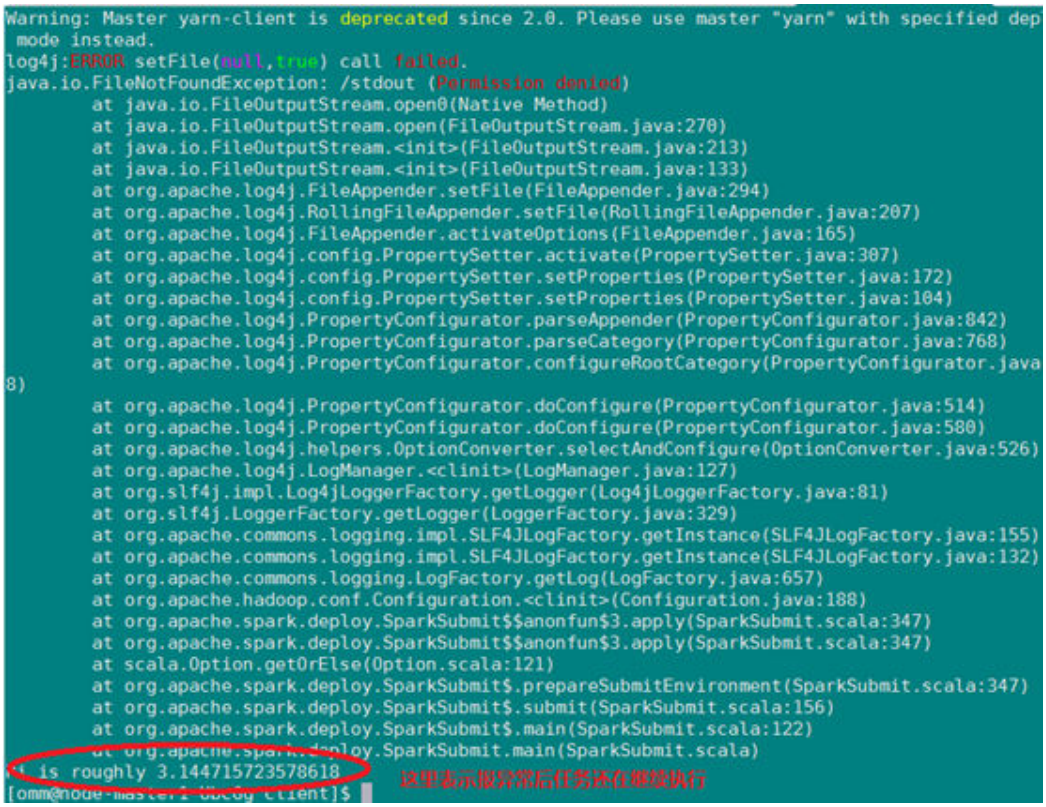
解决措施：

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。

8.7.11 Spark on Yarn 的 client 模式下 spark-submit 提交任务出现 FileNotFoundException 异常

问题

在omm用户（非root用户）下，通过spark-submit提交yarn-client模式的任务，会出现FileNotFoundException异常，任务还能继续执行，但无法查看Driver端日志。例如：执行命令 spark-submit --class org.apache.spark.examples.SparkPi --master yarn-client /opt/client/Spark/spark/examples/jars/spark-examples_2.11-2.2.1-mrs-1.7.0.jar ，结果如下图所示。



回答

原因分析：

在yarn-client模式下执行任务时，Spark的Driver程序在本地执行；其中通过-Dlog4j.configuration=./log4j-executor.properties配置了Driver的日志文件，log4j-executor.properties配置文件设置日志输出到${spark.yarn.app.container.log.dir}/stdout文件中，由于Spark Driver在本地执行时${spark.yarn.app.container.log.dir}没有设置即为空，则输出目录为/stdout，而非root用户下，在根目录下面没有创建和修改stdout的权限，就会报FileNotFoundException异常。而在yarn-cluster模式下执行任务时，Spark的Driver程序在Application Master下执行，而在Application Master启动时就会通过-D${spark.yarn.app.container.log.dir}设置了输出目录，因此在yarn-cluster模式执行任务不会报FileNotFoundException异常。

处理方法：

注：下面所说的$SPAKR_HOME默认是/opt/client/Spark/spark

解决方案1：手动切换日志配置文件。修改文件$SPARK_HOME/conf/spark-defaults.conf中spark.driver.extraJavaOptions的配置项-Dlog4j.configuration=./log4j-executor.properties（默认情况下为./log4j-executor.properties），在yarn-client模式下，修改为-Dlog4j.configuration=./log4j.properties，在yarn-cluster模式下修改为-Dlog4j.configuration=./log4j-executor.properties。

解决方案2：修改启动脚本$SPARK_HOME/bin/spark-class。在spark-class脚本#!/usr/bin/env bash下面添加。

```
# Judge mode: client and cluster; Default: client
argv=`echo $@ | tr [A-Z] [a-z]`
if [[ "$argv" =~ "--master" ]];then
   mode=`echo $argv | sed -e 's/.*--master //'`
```

```
    master=`echo $mode | awk '{print $1}'`
    case $master in
    "yarn")
        deploy=`echo $mode | awk '{print $3}'`
        if [[ "$mode" =~ "--deploy-mode" ]];then
            deploy=$deploy
        else
            deploy="client"
        fi
    ;;
  "yarn-client"|"local")
        deploy="client"
    ;;
    "yarn-cluster")
        deploy="cluster"
    ;;
    esac
else
    deploy="client"
fi
# modify the spark-defaults.conf
number=`sed -n -e '/spark.driver.extraJavaOptions/=' $SPARK_HOME/conf/spark-defaults.conf`
if [ "$deploy"x = "client"x ];then
    `sed -i "${number}s/-Dlog4j.configuration=.*properties /-Dlog4j.configuration=.\/log4j.properties /g"
$SPARK_HOME/conf/spark-defaults.conf`
else
    `sed -i "${number}s/-Dlog4j.configuration=.*properties /-Dlog4j.configuration=.\/log4j-
executor.properties /g" $SPARK_HOME/conf/spark-defaults.conf`
fi
```

这些脚本行的功能和解决方案1类似，通过判断yarn的模式来修改文件$SPARK_HOME/conf/spark-defaults.conf中spark.driver.extraJavaOptions的配置项-Dlog4j.configuration=./log4j-executor.properties。

8.7.12 Spark 任务读取 HBase 报错“had a not serializable result”

问题

Spark任务读取HBase报错，报错信息：Task 0.0 in stage 0.0 (TID 0) had a not serializable result: org.apache.hadoop.hbase.io.ImmutableBytesWritable，应该如何处理?

回答

可通过如下两种方式处理：

- 在代码的SparkConf初始化之前执行以下两行代码：
```
System.setProperty("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
System.setProperty("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator");
```
- 在SparkConf对象使用set方法设置，代码如下：
```
val conf = new SparkConf().setAppName("HbaseTest");
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
conf.set("spark.kryo.registrator", "com.huawei.bigdata.spark.examples.MyRegistrator");
```

8.7.13 本地运行 Spark 程序连接 MRS 集群的 Hive、HDFS

问题

本地运行Spark程序时，如何连接MRS集群的Hive和HDFS?

回答

步骤1 为每一个Master节点申请并绑定弹性公网IP。

步骤2 在本地Windows上配置集群的ip与主机名映射关系。登录集群后台，执行命令**cat /etc/hosts**后，把hosts文件中的ip与hostname映射关系拷贝到“C:\Windows\System32\drivers\etc\hosts”中。其中主机名请以查询结果为准。

```
192.168.0.90 node-master1BedB.089d8c43-12d5-410c-b980-c2728a305be3.com
192.168.0.129 node-ana-corezLaR.089d8c43-12d5-410c-b980-c2728a305be3.com
```

步骤3 以**root**用户登录MRS集群任意一个Master后台，执行命令**cat /etc/hosts**，获取hosts文件中的IP与hostname映射关系。

步骤4 在本地Windows的“C:\Windows\System32\drivers\etc\hosts”中，配置**步骤3**中获取的映射关系，并将所有Master节点的IP修改为对应节点绑定的弹性公网IP。

步骤5 将MRS集群中的“/opt/client/Hive/Beeline/conf/core-site.xml”、“/opt/client/Hive/config/hiveclient.properties”、“/opt/client/Hive/config/hive-site.xml”放入工程的**conf**目录中。

步骤6 登录MRS Manager，选择“系统设置 > 用户管理”。

步骤7 在用户名中选择一个拥有Hive权限的用户，然后在右侧“操作”列中选择“更多 >下载认证凭据”，保存后解压得到用户的user.keytab文件与krb5.conf文件。

步骤8 将krb5.conf文件中Master节点对用的IP修改为对应节点绑定的弹性公网IP。并将user.keytab文件与krb5.conf文件放到工程的**conf**目录中。

步骤9 修改MRS集群的安全组规则，将IDEA所在Windows的IP策略改为全部放通。

----结束

8.8 开发规范

8.8.1 规则

Spark 应用中，需引入 Spark 的类

- **对于Java开发语言，正确示例：**

```
//创建SparkContext时所需引入的类。
import org.apache.spark.api.java.JavaSparkContext
```

```
//RDD操作时引入的类。
import org.apache.spark.api.java.JavaRDD
//创建SparkConf时引入的类。
import org.apache.spark.SparkConf
```

- **对于Scala开发语言，正确示例：**

```
//创建SparkContext时所需引入的类。
import org.apache.spark.SparkContext
//RDD操作时引入的类。
import org.apache.spark.SparkContext._
//创建SparkConf时引入的类。
import org.apache.spark.SparkConf
```

自己抛出的异常必须要填写详细的描述信息

说明：便于问题定位。

正确示例：

```
//抛出异常时，写出详细描述信息。
throw new  IOException("Writing data error! Data: " + data.toString());
```

错误示例：

```
throw new  IOException("Writing data error! ");
```

集群模式下，应注意 Driver 和 worker 节点之间的参数传递

在Spark编程时，总是有一些代码逻辑中需要根据输入参数来判断，这种时候往往会想到这种做法，将参数设置为全局变量，先给定一个空值（null），在main函数中，实例化SparkContext对象之前对这个变量赋值。然而，在集群模式下，执行程序的jar包会被发送到每个Worker上执行。如果只在main函数的节点上全局变量改变了，而未传给执行任务的函数中，将会报空指针异常。

正确示例：

```
object Test
{
private var testArg: String = null;
def main(args: Array[String])
{
testArg = …;
val sc: SparkContext = new SparkContext(…);

sc.textFile(…)
.map(x => testFun(x, testArg));
}

private def testFun(line: String, testArg: String): String =
{
testArg.split(…);
return …;
}
}
```

错误示例：

```
//定义对象。
object Test
{
// 定义全局变量，赋为空值（null）；在main函数中，实例化SparkContext对象之前对这个变量赋值。
private var testArg: String = null;
//main函数
def main(args: Array[String])
{
```

```
testArg = …;
val sc: SparkContext = new SparkContext(…);

sc.textFile(…)
.map(x => testFun(x));
}

private def testFun(line: String): String =
{
testArg.split(...);
return …;
}
}
```

运行错误示例，在Spark的local模式下能正常运行，而在集群模式情况下，会在红色加粗代码处报错，提示空指针异常，这是由于在集群模式下，执行程序的jar包会被发送到每个Worker上执行，当执行到testFun函数时，需要从内存中取出testArg的值，但是testArg的值只在启动main函数的节点改变了，其他节点无法获取这些变化，因此它们从内存中取出的就是初始化这个变量时的值null，这就是空指针异常的原因。

应用程序结束之前必须调用 SparkContext.stop

利用spark做二次开发时，当应用程序结束之前必须调用SparkContext.stop()。

说明

利用Java语言开发时，应用程序结束之前必须调用JavaSparkContext.stop().
利用Scala语言开发时，应用程序结束之前必须调用SparkContext.stop()。。

以Scala语言开发应用程序为例，分别介绍下正确示例与错误示例。

正确示例：

```
//提交spark作业
val sc = new SparkContext(conf)

//具体的任务
...

//应用程序结束
sc.stop()
```

错误示例：

```
//提交spark作业
val sc = new SparkContext(conf)

//具体的任务
...
```

如果不添加SparkContext.stop，界面会显示失败。如图8-47，同样的任务，前一个程序是没有添加SparkContext.stop，后一个程序添加了SparkContext.stop()。

图 8-47 添加 SparkContext.stop()和不添加的区别

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;
}
```

自建用户执行 sparksql 命令时，需赋予用户相应写权限

用户可通过系统提供的spark用户来进行sparksql操作，也可通过在MRS Manager界面新建用户来执行sparksql操作。

在通过自建用户执行sparksql相应写操作时，可通过为用户选择supergroup组，赋予用户SystemAdministrator角色等方式赋予用户相应写操作的权限；如果新建用户为hadoop组用户且没有赋予特定角色，在一些涉及到写入操作的场景下可能会因不具备相应操作权限引起异常。

8.8.2 建议

RDD 多次使用时，建议将 RDD 持久化

RDD在默认情况下的存储级别是StorageLevel.NONE，即既不存磁盘也不放在内存中，如果某个RDD需要多次使用，可以考虑将该RDD持久化，方法如下：

调用spark.RDD中的cache()、persist()、persist(newLevel: StorageLevel)函数均可将RDD持久化，cache()和persist()都是将RDD的存储级别设置为StorageLevel.MEMORY_ONLY，persist(newLevel: StorageLevel)可以为RDD设置其他存储级别，但是要求调用该方法之前RDD的存储级别为StorageLevel.NONE或者与newLevel相同，也就是说，RDD的存储级别一旦设置为StorageLevel.NONE之外的级别，则无法改变。

如果想要将RDD去持久化，那么可以调用unpersist(blocking: Boolean = true)，将该RDD从持久化列表中移除，并将RDD的存储级别重新设置为StorageLevel.NONE。

调用有 shuffle 过程的算子时，尽量提取需要使用的信息

该类算子称为宽依赖算子，其特点是父RDD的一个partition影响子RDD得多个partition，RDD中的元素一般都是<key, value>对。执行过程中都会涉及到RDD的partition重排，这个操作称为shuffle

由于shuffle类算子存在节点之间的网络传输，因此对于数据量很大的RDD，应该尽量提取需要使用的信息，减小其单条数据的大小，然后再调用shuffle类算子。

常用的有如下几种：

- combineByKey() : RDD[(K, V)] => RDD[(K, C)]，是将RDD[(K, V)]中key相同的数据的所有value转化成为一个类型为C的值。
- groupByKey() 和reduceByKey()是combineByKey的两种具体实现，对于数据聚合比较复杂而groupByKey和reduceByKey不能满足使用需求的场景，可以使用自己定义的聚合函数作为combineByKey的参数来实现。
- distinct(): RDD[T] => RDD[T]，作用是去除重复元素的算子。其处理过程代码如下：

  map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(_._1)

  这个过程比较耗时，尤其是数据量很大时，建议不要直接对大文件生成的RDD使用。
- join() : (RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (V, W))]，作用是将两个RDD通过key做连接。
- 如果RDD[(K, V)]中某个key有X个value，而RDD[(K, W)]中相同key有Y个value，那么最终在RDD[(K, (V, W))]中会生成X*Y条记录。

在一条语句中如果连续调用多个方法，每个方法占用一行，以符号“.”对齐

这样做，可以增强代码可读性，明确代码执行流程。

代码示例：

```
val data: RDD[String] = sc.textFile(inputPath)
                          .map(line => (line.split(",")(0), line))
                          .groupByKey()
                          .collect()
                          .sortBy(pair => pair._1)
                          .flatMap(pair => pair._2);
```

异常捕获尽量不要直接 catch（Exception ex），应该把异常细分处理

可以设计更合理异常处理分支。

明确方法功能，精确（而不是近似）地实现方法设计，一个函数仅完成一件功能，即使简单功能也编写方法实现

虽然为仅用一两行就可完成的功能去编方法好象没有必要，但用方法可使功能明确化，增加程序可读性，亦可方便维护、测试。下面举个例子说明。

不推荐的做法：

```
//该示例中map方法中，实现了对数据映射操作。
val data：RDD[(String, String)] = sc.textFile(input)
                                    .map(record =>
                                    {
                                      val elems = record.split(",");
```

```
                (elems(0) + "," + elems(1), elems(2));
            });
```

推荐的做法：

```
//将上述示例中的map方法中的操作提取出来，增加了程序可读性，便于维护和测试。
val data：RDD[(String, String)] = sc.textFile(input)                                    .map(record =>
deSomething(record));

def deSomething(record: String): (String, String) =
{
  val elems = x.split(",");
  return (elems(0) + "," + elems(1), elems(2));
}
```

Spark SQL 动态分区插入优化之 Distributeby

如下SQL中，p1和p2是target表的分区字段，使用**distribute by**关键字来减少小文件的产生。

```
insert overwrite table target partition(p1,p2)
select * from source
distribute by p1, p2
```

Spark程序以动态分区的方式写入Hive表时，会出现了大量的小文件，导致最后移动文件到hive表目录非常耗时，这是因为在Shuffle时Hive多个分区的数据随机落到Spark的多个Task中，此时Task与Hive分区数据的关系是多对多，即每个Task会包含多个分区的部分数据，每个Task中包含的每个分区的数据都很少，最终会导致Task写多个分区文件，每个分区文件都比较小。

为了减少小文件的数量，需要将数据按照分区字段进行Shuffle，将各个分区的数据尽量各自集中在一个Task，在Spark SQL中就是通过**distribute by**关键字来完成这个功能的。

当使用**distribute by**关键字在后出现了数据倾斜，即有的分区数据多，有的分区数据少，也会导致spark 作业整体耗时长。需要在distribute by后面增加随机数，例如：

```
insert overwrite table target partition(p1,p2)
select * from source
distribute by p1, p2, cast(rand() * N as int)
```

N值可以在文件数量和倾斜度之间做权衡。

在Spark SQL中使用动态分区写入数据，需要同时使用**distribute by**和**sort by**才能有效减少小文件数量。

Spark SQL 动态分区插入优化之 sortby

如下SQL中，p1和p2是target表的分区字段，使用**sort by**关键字来减少小文件的产生。

```
insert overwrite table target partition(p1,p2)
select * from source
distribute by p1, p2
sort by p1,p2
```

经过动态分区Shuffle优化之后，每一个Hive分区的数据都会集中在一个Spark Task中，但是由于Hive分区的数量远远大于Task数量，所以一个Task中会包含多个Hive分区的数据，即Task与Hive分区的关系是一对多。

每一个Task会将其包含的数据按照行顺序写入文件，文件所在的分区由该行数据中的分区字段值决定。Task在写入第一行数据时会创建一个新文件，随后写的每行数据都会判断该行数据的分区字段值与上一行数据的分区字段值是否相同，如果不相同就会

新建一个文件并将该行数据写入，否则将该行数据写入上一条数据所在的文件。因此在Task写动态分区数据时，相邻两行数据如果分区字段值相同，就会写入同一个文件，否则就会写入不同的文件。假设Task有N行数据，在最坏情况下，所有相邻数据的分区字段值都不相同，那么Task将会写N个文件，每个文件只有一行数据。

为了将一个Task中相同分区的数据集中在一起，减少Task写的文件数量，需要将数据按照分区字段进行排序。假设一个Task中包含M个分区数据，排序之后，一个Task中相同分区的数据就会相邻，最终一个Task只会写M个文件。在Spark SQL中增加**sort by**关键词可完成排序功能。

在Spark SQL中使用动态分区写入数据，需要同时使用**distribute by**和**sort by**才能有效减少小文件数量。

Spark 建议使用 Commit V2 算法

在Spark提交作业时配置参数spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2，使用commit v2算法，例如在DataArts Studio提交作业增加—conf配置项，值为spark.hadoop.mapreduce.fileoutputcommitter.algorithm.version=2。

Spark默认采用Commit V1算法，该算法原理为：Task执行完毕后将数据写入了临时目录，Driver等待所有Task执行完毕后，串行的将每个Task的临时输出文件移动到最终的目录中。如果输出的小文件过多，Driver串行移动文件耗时会过长，最终导致Commit过程比较耗时。

将Commit算法修改为V2，使得每个Task在执行成功后将临时文件移动到最终目录，相当于Driver中串行的移动操作优化为Task并行的移动文件操作。V2算法的缺点在于Spark作业执行过程中，最终目录的文件是对外可见的，如果此时有其他程序读取了最终目录的数据，那么其他程序处理的数据出现不一致问题。

然而使用Spark写Hive表，Spark作业的最终目录也是一个临时目录，通过load操作将临时目录数据导入hive表，所以Hive表的目录才是真正的最终目录，外部作业是无法读取到中间临时生成目录，因此针对Spark写Hive场景推荐使用Commit V2算法。

批量删除分区策略

当批量删除分区时，例如删除一个月所有的分区数据，可以使用的方式有两种：

- 把所有的分区列出来，然后批量删除。假设city的数量有100个，那么删除一个月的所有分区需要列出3100个分区，然后批量删除。
  ```
  alter table target drop partition(city=c1,date=p1), partition(city=c2,date=p2),…
  ```
- 只列出一个月的时间分区，然后批量删除。这种方式只需要列出31个时间分区。
  ```
  alter table target drop partition(date=p1), partition(date=p2)
  ```

这两种方式都可以删除一个月的所有分区，但是第二种删除分区的方式性能高于第一种。这是因为在删除分区前会查询分区是否存在，第一种方式会调用3100次查询API，而第二种方式只需要查询31次。

9 Storm 应用开发

9.1 概述

9.1.1 应用开发简介

目标读者

本文档提供给需要Storm二次开发的用户使用。本指南主要适用于具备Java开发经验的开发人员。

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**

9.1.2 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

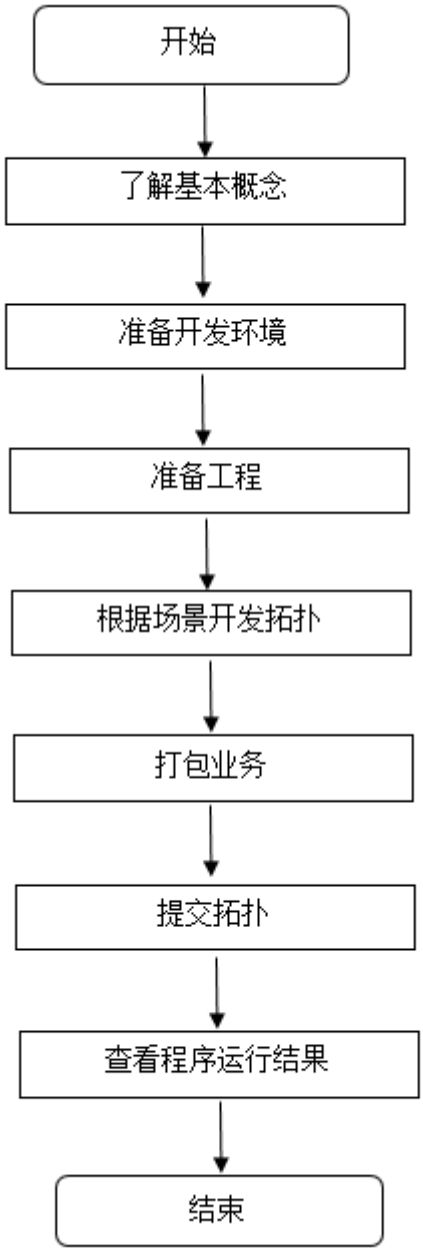
存放用户信息的密钥文件。应用程序采用此密钥文件在MRS产品中进行API方式认证。

9.1.3 开发流程

本文档主要基于Java API进行Storm拓扑的开发。

开发流程如图9-1所示。

图 9-1 拓扑开发流程



9.2 Linux 客户端环境准备

背景信息

安装Linux客户端用于拓扑的提交。

前提条件

- 确认Storm组件已经安装，并正常运行。
- 客户端机器的时间与集群的时间要保持一致，时间差要小于5分钟。

操作步骤

步骤1 下载Storm客户端程序。

1. 登录**MRS Manager**系统。
2. 选择“服务管理 > Storm > 下载客户端 > 完整客户端”，下载客户端程序到“远端主机”，即目标ECS。

步骤2 登录到客户端下载的目标ECS。

步骤3 在Linux系统中，使用如下命令解压客户端压缩包。

tar -xvf MRS_Storm_Client.tar

tar -xvf MRS_Storm_ClientConfig.tar

步骤4 进入“MRS_Services_ClientConfig”中，执行“install.sh”脚本安装客户端，将客户端安装到一个空文件夹，命令为：**./install.sh /opt/Storm_Client**（此处/opt/Storm_Client表示的是Storm安装目录，此目录必为空目录，且必须是绝对路径）。

步骤5 初始化客户端环境变量。

进入安装目录“/opt/Storm_Client”执行以下命令，导入环境变量信息。

source bigdata_env

步骤6 开启Kerberos认证的集群，需要申请人机用户，并进行安全登录。

1. 从管理员处获取一个“人机”用户，用于服务认证。例如：账号**john**。

   说明

   获取的用户需要属于storm组。

2. 执行kinit命令进行“人机”用户的安全登录。

   kinit *用户名*

   例如：

   kinit john

   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7 执行如下命令。

storm list

如果可以正常打印出storm集群正在运行的任务信息，则说明客户端安装成功。

----结束

9.3 Windows 开发环境准备

9.3.1 开发环境简介

本开发指南提供了MRS产品Storm组件基于开源Storm的Eclipse样例工程和常用接口说明，便于开发者快速熟悉Storm开发。

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。

在进行二次开发时，要准备的开发环境如表9-1所示。

表 9-1 开发环境

准备项	说明
操作系统	Windows系统，推荐Windows 7以上版本。
安装JDK	开发环境的基本配置。版本要求：1.7或者1.8。 **说明** 基于安全考虑，服务端只支持TLS 1.1和TLS 1.2加密协议，IBM JDK默认TLS只支持1.0，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TLS1.0/1.1/1.2。 详情请参见：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Eclipse	用于开发Storm应用程序的工具。
网络	确保客户端与Storm服务主机在网络上互通。

9.3.2 准备 Eclipse 与 JDK

操作场景

开发环境可以搭建在Windows环境下。

操作步骤

步骤1 安装Eclipse程序。安装要求Eclipse使用3.0及以上版本。

步骤2 安装JDK程序。安装要求JDK使用1.7及或者1.8版本，支持IBM JDK和Oracle JDK。

说明

- 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
- 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
- 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。

----结束

9.3.3 配置并导入工程

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Eclipse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MRS Manager页面右上角查看。

操作步骤

步骤1 在Storm示例工程根目录，执行mvn install编译

步骤2 在Storm示例工程根目录，执行mvn eclipse:eclipse创建Eclipse工程。

步骤3 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。

   显示“浏览文件夹”对话框。

2. 选择样例工程文件夹，单击“Finish”。

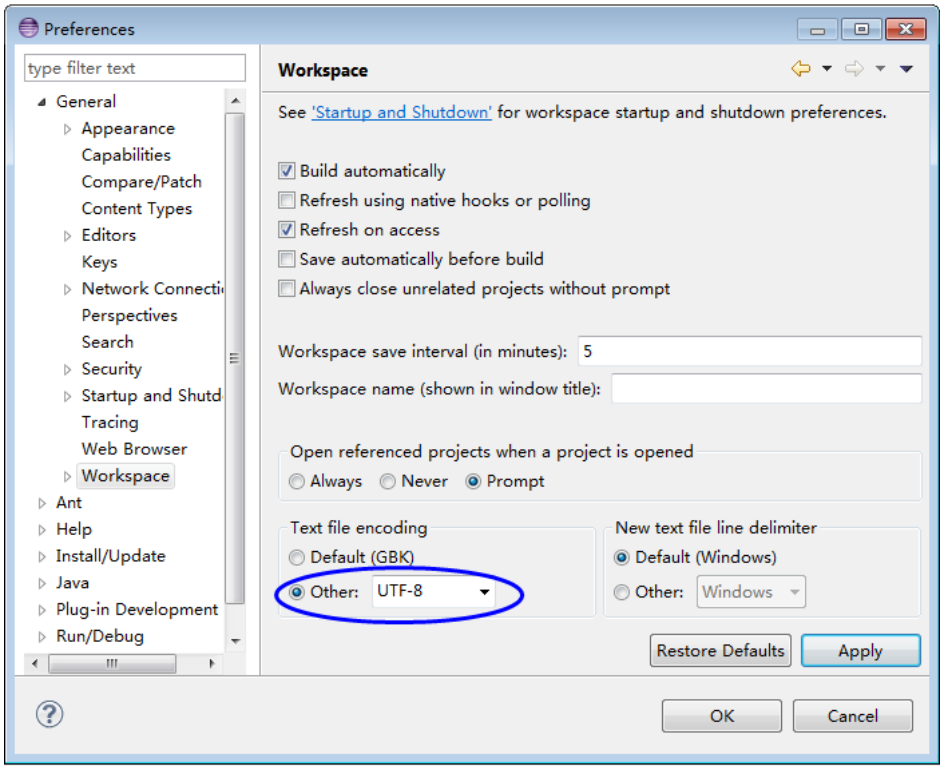
步骤4 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。

   弹出“Preferences”窗口。

2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图9-2所示。

图 9-2 设置 Eclipse 的编码格式



步骤5 Windows环境下直接提交任务到集群场景，需要将集群ip映射配到本地host文件中。以Windows7为例，路径为“C:\Windows\System32\drivers\etc\hosts”。

例如，集群有3个节点10.1.131.131，10.1.131.132，10.1.131.133。

则需要检查**hosts**文件中是否配置了以下内容。

```
10.1.131.131 10-1-131-131
10.1.131.132 10-1-131-132
10.1.131.133 10-1-131-133
```

----结束

9.4 开发程序

9.4.1 典型场景说明

通过典型场景，您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9-2**所示。

表 9-2 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Bolt**
3	创建一个Blot用来统计收到的各单词次数	请参见**创建Bolt**
4	创建topology	请参见**创建Topology**

部分代码请参考**创建Spout**，**创建Bolt**，**创建Topology**，完整代码请参考Storm-examples示例工程。

9.4.2 创建 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.RandomSentenceSpout类中，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));
 }
```

9.4.3 创建 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。

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.SplitSentenceBolt类中，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);
  }
```

9.4.4 创建 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.WordCountTopology类中，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Eclipse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：
     * 1、打包成Jar包，然后在客户端命令行上面进行提交
     * 2、远程提交的时候，要先将该应用程序和其他外部依赖(非example工程提供，用户自己程序依赖)的jar包
打包成一个大的jar包
     * 3、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.streaming.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;
    }
```

9.5 运行应用

9.5.1 生成示例 Jar 包

操作场景

通过命令行生成示例代码的jar包。

操作步骤

在Storm示例代码根目录执行如下命令打包："mvn package"。执行成功后，将会在target目录生成storm-examples-1.0.jar。

9.5.2 Linux 中安装客户端时提交拓扑

操作场景

说明

Storm应用程序不支持在Windows环境下运行，只支持在Linux环境下运行。

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**生成示例Jar包**步骤，生成storm-examples-1.0.jar，并放置到/opt/jartarget/。

操作步骤

步骤1 安全模式下，请先进行安全认证，参见**Linux客户端环境准备**。

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客户端目录“storm-0.10.0/bin”，执行命令：**storm jar /opt/jartarget/**storm-examples-1.0.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

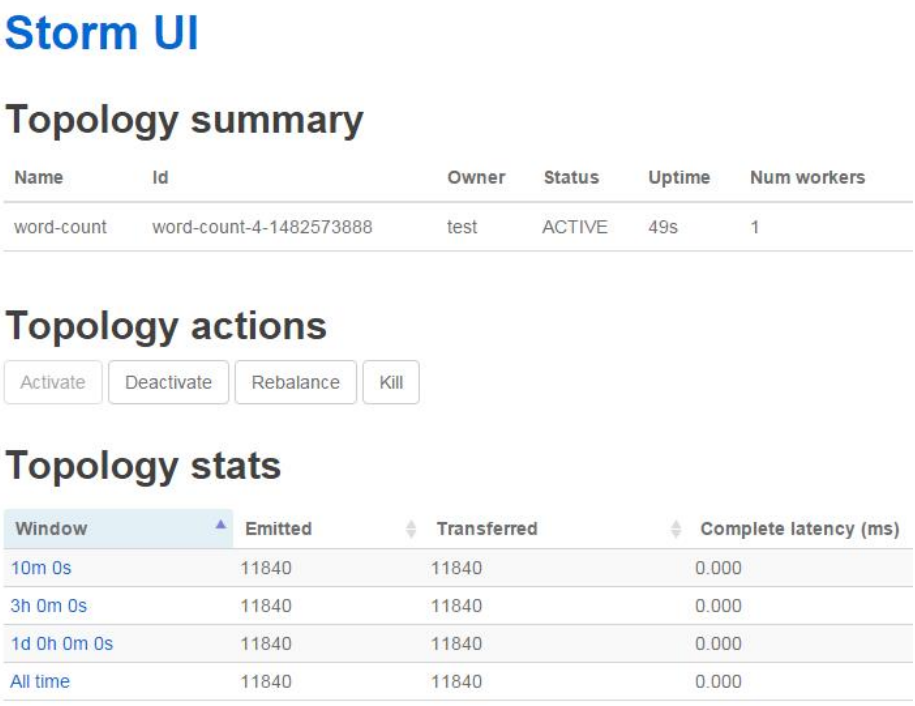
9.5.3 查看结果

操作步骤

步骤1 参考《访问开源组件UI界面》章节，访问Storm Web界面 。

步骤2 在Storm UI中单击word-count应用，查看应用程序运行情况，如图9-3所示。

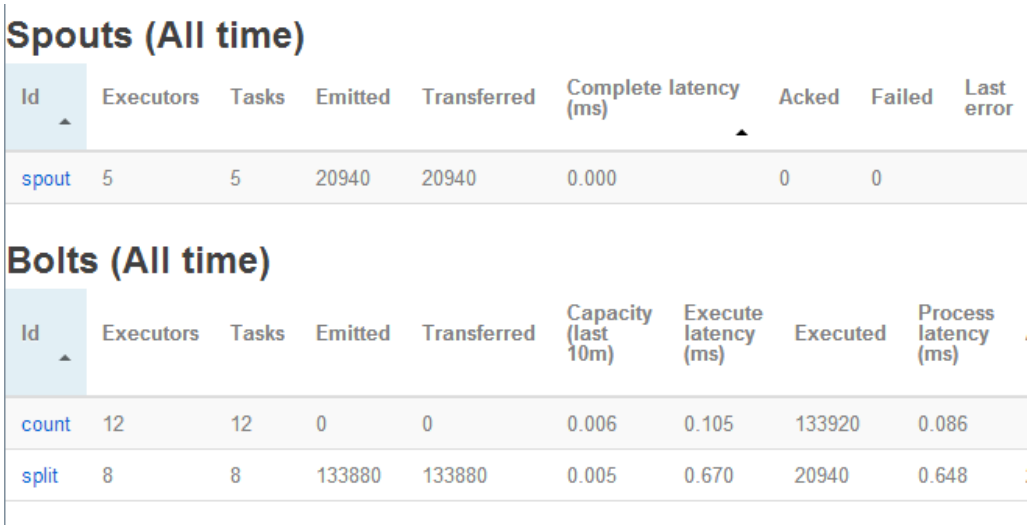
图 9-3 Storm 应用程序执行界面



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9-4所示。

图 9-4 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

9.6 更多信息

9.6.1 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Eclipse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MRS产品Storm和Kafka组件已经安装，并正常运行。

步骤2 已搭建Storm示例代码工程，将storm-examples导入到Eclipse开发环境，参见**配置并导入工程**。

步骤3 用WinScp工具将Storm客户端安装包导入Linux环境并安装客户端，参见**Linux客户端环境准备**。

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，用于认证，并且获取到该用户的keytab文件。将获取到的文件拷贝到示例工程的 src/main/resources 目录。

说明

- 获取的用户需要同时属于storm组和kafka组。

步骤5 下载并安装Kafka客户端程序，参见《 Kafka应用开发 》。

----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
conf.put(KafkaBolt.KAFKA_BROKER_PROPERTIES, getKafkaProducerProps());
KafkaBolt<String, String> kafkaBolt = new KafkaBolt<String, String>();
kafkaBolt.withTopicSelector(new DefaultTopicSelector(OUTPUT_TOPIC))
.withTupleToKafkaMapper(
new FieldNameBasedTupleToKafkaMapper("word", "count"));

// 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());
}
```

部署运行及结果查看

步骤1 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2 在Storm示例代码根目录执行如下命令打包："mvn package"。执行成功后，将会在target目录生成storm-examples-1.0.jar。

步骤3 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4 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/storm-examples-1.0.jar **com.huawei.storm.example.kafka.NewKafkaTopology kafka-test**

说明

- 安全模式下，在提交storm-examples-1.0.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，因此首先需要给用户赋权，再提交拓扑。

步骤5 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 --new-consumer --consumer.config ../../../Kafka/kafka/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 ../../../Kafka/kafka/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

9.6.2 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。

本章节只适用与MRS产品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Storm组件已经安装，且正常运行。

步骤2 下载Storm客户端，将Storm样例工程导入到Eclipse开发环境，参见**配置并导入工程**。

步骤3 用WinScp工具将Storm客户端导入Linux环境并安装，具体请参见**Linux客户端环境准备**。

----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(本示例使用10.14.1.0)，通过WinScp等工具传入Linux客户端，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-10.14.1.0**-bin**

export CLASSPATH=$DERBY_INSTALL/lib/derbytools.jar:$DERBY_INSTALL\lib\derbynet.jar:.

export DERBY_HOME=/opt/db-derby-10.14.1.0**-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。

说明

- 执行./ij命令前，需要确保已配置java_home,可通过which java命令检查是否已配置。

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 String WORD_SPOUT = "WORD_SPOUT";
  private static final String COUNT_BOLT = "COUNT_BOLT";
  private static final String JDBC_INSERT_BOLT = "JDBC_INSERT_BOLT";
  private static final String JDBC_LOOKUP_BOLT = "JDBC_LOOKUP_BOLT";
  @SuppressWarnings ("unchecked")
  public static void main(String[] args) throws Exception{
    //connectionProvider配置
    Map hikariConfigMap = Maps.newHashMap();
    hikariConfigMap.put("dataSourceClassName", "org.apache.derby.jdbc.ClientDataSource");
```

```
    hikariConfigMap.put("dataSource.serverName", "192.168.0.1");//请改为实际的IP
    hikariConfigMap.put("dataSource.portNumber", "1527");//请改为实际的端口

    hikariConfigMap.put("dataSource.databaseName", "example");
    hikariConfigMap.put("connectionTestQuery", "select COUNT from  GOAL"); //表名需与建表时保持一致
    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 ORIGINAL where WORD = ?";
    JdbcLookupBolt wordLookupBolt = new JdbcLookupBolt(connectionProvider, selectSql,
jdbcLookupMapper);
    //JdbcInsertBolt 实例化
    String tableName = "GOAL";
    JdbcMapper simpleJdbcMapper = new SimpleJdbcMapper(tableName, connectionProvider);
    JdbcInsertBolt userPersistanceBolt = new JdbcInsertBolt(connectionProvider,
simpleJdbcMapper).withTableName("GOAL").withQueryTimeoutSecs(30);
    WordSpout wordSpout = new WordSpout();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, userPersistanceBolt,1).fieldsGrouping(JDBC_LOOKUP_BOLT,new
Fields("WORD"));StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
  }
 }
```

部署运行

步骤1 在Storm示例代码根目录执行如下命令打包："mvn package"。执行成功后，将会在target目录生成storm-examples-1.0.jar。

步骤2 执行命令提交拓扑。提交命令示例（拓扑名为jdbc-test）。

storm jar /opt/jartarget/storm-examples-1.0.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。

9.6.3 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。

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Eclipse开发环境，请参见**配置并导入工程**。

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。
- 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，参见《准备Linux客户端运行环境》。

步骤5 获取HDFS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/client/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。

将获取到的这些文件拷贝到示例工程的 src/main/resources目录。

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。

----结束

Eclipse 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()
        .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());

}
```

部署运行及结果查看

步骤1 在Storm示例代码根目录执行如下命令打包："mvn package"。执行成功后，将会在target目录生成storm-examples-1.0.jar。

步骤2 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/storm-examples-1.0.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3 拓扑提交成功后，请登录HDFS集群查看/user/foo目录下是否有文件生成。

步骤4 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-0.10.0/conf/storm.yaml文件尾部新起一行添加如下内容。

   topology.auto-credentials:

   - backtype.storm.security.auth.kerberos.AutoTGT

2. 执行命令：**./storm upload-credentials hdfs-test**

----结束

9.6.4 Storm-OBS 开发指引

操作场景

本章节只适用于MRS产品中Storm和OBS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已经安装，并正常运行。

步骤2 将storm-examples导入到Eclipse开发环境，请参见**配置并导入工程**。

步骤3 下载并安装HDFS客户端，参见**准备Linux客户端运行环境**。

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/client/HDFS/hadoop/etc/hadoop”，在该目录下获取到配置文件“core-site.xml”和“hdfs-site.xml”。将这些文件拷贝到示例工程的 src/main/resources目录。并在core-site.xml中增加如下配置项:

```
<property>
<name>fs.obs.connection.ssl.enabled</name>
<value>true</value>
</property>
<property>
<name>fs.obs.endpoint</name>
<value></value>
</property>
<property>
<name>fs.obs.access.key</name>
<value></value>
</property>
<property>
<name>fs.obs.secret.key</name>
<value></value>
</property>
```

具体AK,SK等的获取，请参考OBS相关帮助。

----结束

Eclipse 代码样例

创建Topology。

```
private static final String DEFAULT_FS_URL = "obs://mybucket";

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.KB);

    // 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"));

    Config conf = new Config();


    //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 在Storm示例代码根目录执行如下命令打包："mvn package"。执行成功后，将会在target目录生成storm-examples-1.0.jar。

步骤2 执行命令提交拓扑。

提交命令示例（拓扑名为obs-test）。

storm jar /opt/jartarget/storm-examples-1.0.jar
com.huawei.storm.example.obs.SimpleOBSTopology obs://my-bucket obs-test

步骤3 拓扑提交成功后请登录OBS Browser查看。

----结束

9.6.5 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Eclipse开发环境，请参见**配置并导入工程**。

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。
- 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/client/HBase/hbase/conf”，在该目录下获取到core-site.xml、hdfs-site.xml、hbase-site.xml配置文件。将这些文件拷贝到示例工程的 src/main/resources目录。

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。

----结束

Eclipse 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，安全模式需要设置
        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 在Storm示例代码根目录执行如下命令打包："mvn package"。执行成功后，将会在target目录生成storm-examples-1.0.jar。

步骤2 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/storm-examples-1.0.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3 拓扑提交成功后请自行登录HBase集群查看WordCount表是否有数据生成。

步骤4 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-0.10.0/conf/storm.yaml文件尾部新起一行添加如下内容。

   topology.auto-credentials:

   - backtype.storm.security.auth.kerberos.AutoTGT

2. 执行命令：**./storm upload-credentials hbase-test**

----结束

9.6.6 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 0.10.0版本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KafkaBolt使用的properties
- id: "kafkaProducerProps"
    className: "java.util.Properties"
    configMethods:
      - name: "put"
        args:
          - "bootstrap.servers"
          - "${metadata.broker.list}"
      - name: "put"
        args:
          - "acks"
          - "1"
      - name: "put"
        args:
          - "key.serializer"
          - "org.apache.kafka.common.serialization.StringSerializer"
      - name: "put"
        args:
          - "value.serializer"
          - "org.apache.kafka.common.serialization.StringSerializer"
```

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

- id: "kafkaBolt"
 className: "org.apache.storm.kafka.bolt.KafkaBolt"
 configMethods:
   - name: "withTopicSelector"
     args:
       - ref: "defaultTopicSelector"
   - name: "withProducerProperties"
     args: [ref: "kafkaProducerProps"]
   - name: "withTupleToKafkaMapper"
     args:
       - ref: "fieldNameBasedTupleToKafkaMapper"
```

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backtype.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Eclipse开发环境，请参见**Windows开发环境准备**。

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。

----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: "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 使用如下命令打包：“mvn package”。执行成功后，将会在target目录生成storm-examples-1.0.jar。

步骤2 将打好的jar包，以及开发好的yaml文件及相关的properties文件拷贝至storm客户端所在主机的任意目录下，如“/opt”。

步骤3 执行命令提交拓扑。

storm jar /opt/jartarget/storm-examples-1.0.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/storm-examples-1.0.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/storm-examples-1.0.jar **org.apache.storm.flux.Flux --remote /opt/my-topology.yaml --filter /opt/my-prop.properties**

步骤4 拓扑提交成功后请自行登录storm UI查看。

----结束

9.6.7 对外接口

Storm采用的接口同开源社区版本保持一致，详情请参见：

http://storm.apache.org/documentation/Home.html。

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/v0.10.0/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/v0.10.0/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/v0.10.0/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/v0.10.0/external/storm-jdbc。

9.7 开发规范

9.7.1 规则

不允许将“storm.yaml”打包到应用程序 jar 包中。

若将“storm.yaml”打包到应用程序jar包中，应用提交后会导致应用程序使用的“storm.yaml”与集群中的“storm.yaml”冲突，造成应用程序不可用。

打包完成后，可使用解压工具查看jar包根目录下是否存在该文件，若存在则删除。

不允许将 log4j 相关 jar 包打包到应用程序 jar 包中。

若将log4j相关jar包打包到应用程序jar包中，应用提交后会导致系统中的log4j与应用程序中的log4j冲突，造成应用程序不可用。

打包前，需要排查jar包工程中是否存在log4j相关jar包，若存在则删除后再重新打包。

不允许将“storm-core-X.X.X.jar”打包到应用程序 jar 包中。

若将“storm-core-X.X.X.jar”打包到应用程序jar包中，应用提交后会导致系统中的“storm-core-0.10.0.jar”与应用程序中的“storm-core-X.X.X.jar”冲突，造成应用程序不可用。

打包前，需要排查jar包工程中是否存在“storm-core-X.X.X.jar”，若存在则删除后再重新打包。

使用远程提交方式提交拓扑时，应保证所打 jar 包和本地工程代码的一致性。

若提交的应用程序jar包与本地工程代码不一致，将导致提交的应用在运行期间报错。

9.7.2 建议

一个 Topology 所使用的 worker 数量，建议不要超过 12 个

虽然一个Topology所使用的worker数量越多，能够处理的数据量越大，但worker间通信的成本就越高，建议不超过12个worker。

一个 Topology 的层级建议不要超过 6 层，层级的增加会导致 Spout 吞吐量下降

一个Topology的层级越深，数据在集群内的传输消耗越大，因此导致性能显著下降。

10 Kafka 应用开发

10.1 概述

10.1.1 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**Java API**。

10.1.2 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。

10.1.3 开发流程

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10-1和表10-1所示。

图 10-1 Kafka 客户端程序开发流程

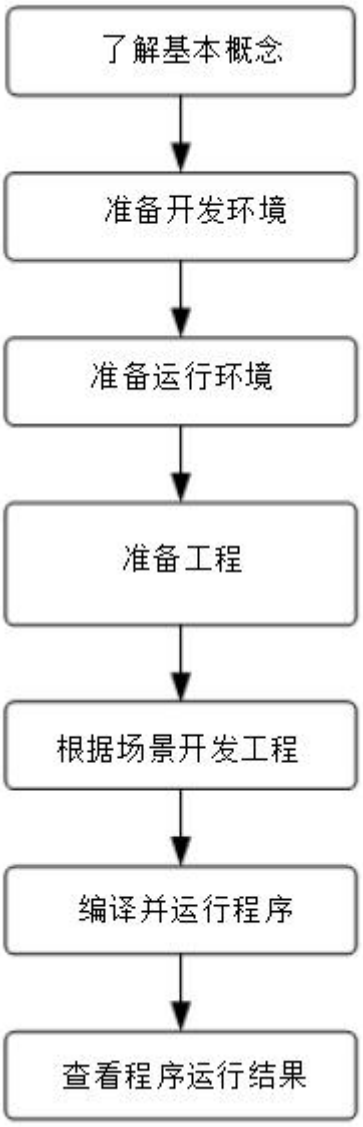


表 10-1 Kafka 客户端程序开发的流程说明

阶段	说明	参考文档
了解基本概念	在开始开发客户端前，需要了解Kafka的基本概念，根据实际场景判断，需要开发的角色是Producer还是Consumer。	常用概念
准备开发环境	Kafka的客户端程序当前推荐使用java语言进行开发，并使用Maven工具构建工程。	准备Maven和JDK
准备运行环境	Kafka的样例程序运行环境即MRS服务所VPC集群的节点。	-
准备工程	Kafka提供了不同场景下的样例程序，您可以下载样例工程进行程序学习。或者您可以根据指导，新建一个Kafka工程。	导入样例工程
根据场景开发工程	提供了Producer和Consumer相关API的使用样例，包含了新旧API和多线程的使用场景，帮助用户快速熟悉Kafka接口。	典型场景说明
编译并运行程序	指导用户将开发好的程序编译并打包，上传到VPC的Linux节点运行。	在Linux中调测程序
查看程序运行结果	程序运行结果可以输出到Linux命令行页面。也可通过Linux客户端进行Topic数据消费的方式查看数据是否写入成功。	在Linux中调测程序

10.2 环境准备

10.2.1 开发环境简介

Kafka开发应用时，需要准备的开发环境如下表所示：

表 10-2 开发环境

准备项	说明
操作系统	Windows系统，推荐Windows 7以上版本。
安装JDK和Maven	开发环境的基本配置。JDK版本要求：1.7或者1.8。Maven版本要求：3.3.0及以上
安装和配置Eclipse或IntelliJ IDEA	用于开发Kafka应用程序的工具。
网络	确保本地与Kafka服务所在的VPC的至少一个节点在网络上互通。
访问云服务器的安全认证	本地可以通过密钥或密码方式登录访问Linux弹性云服务器

10.2.2 准备 Maven 和 JDK

操作场景

开发环境搭建在Windows环境下。

操作步骤

1. 开发环境安装Eclipse程序，安装要求如下。
   - Eclipse使用3.0及以上版本。
   - IntelliJ IDEA使用15.0以上版本。
2. 开发环境安装JDK环境，安装要求如下。
   JDK使用1.7或者1.8版本。支持IBM JDK和Oracle JDK。

   说明

   - 若使用IBM JDK，请确保Eclipse或者IntelliJ IDEA中的JDK配置为IBM JDK。
   - 若使用Oracle JDK，请确保Eclipse或者IntelliJ IDEA中的JDK配置为Oracle JDK。
   - 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。
3. 开发环境安装Mave环境，安装版本3.0.0以上。

10.2.3 导入样例工程

操作步骤

步骤1 参照**样例工程获取地址**，下载样例工程到本地。

步骤2 解压样例工程并找到kafka-examples目录。

步骤3 导入样例工程到Eclipse开发环境。

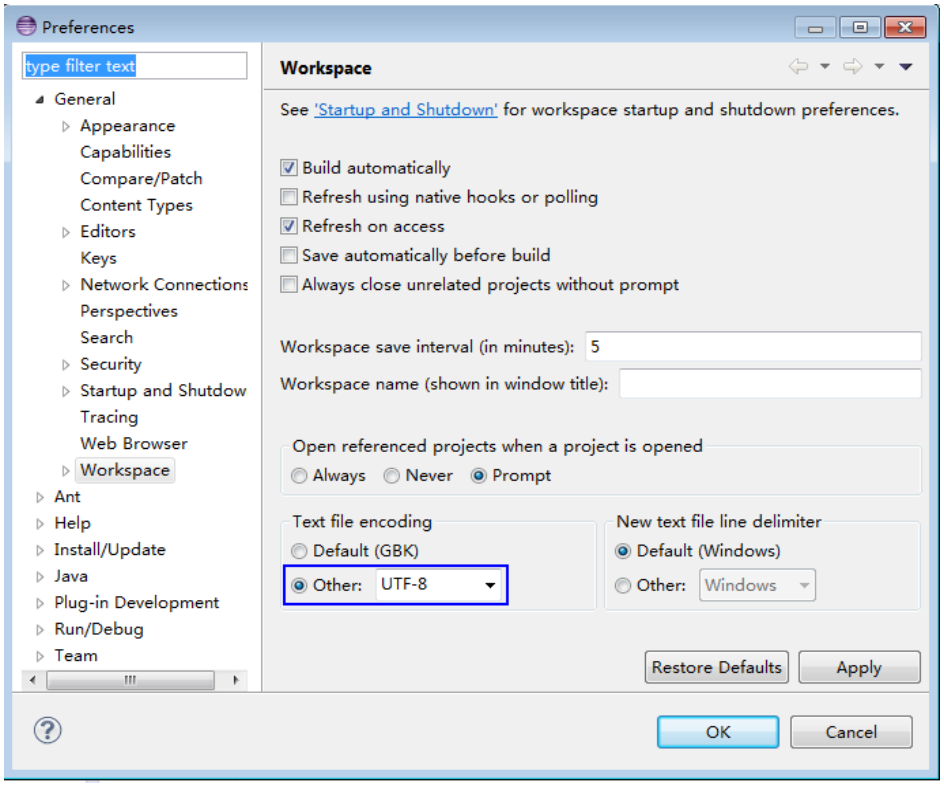
1. 打开Eclipse，选择“File > Import”。显示“Import”窗口，选择Existing Maven Projects，单击“next”按钮。

2. 在“Import Maven Projects”窗口单击“Browse”。显示“Select Root Folder”对话框。
3. 选择样例工程文件夹**kafka-examples**，单击“确定”按钮。
4. 在“Import Maven Projects”窗口单击“Finish”按钮。

步骤4 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。
   弹出“Preferences”窗口。
2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图10-2所示。

图 10-2 设置 Eclipse 的编码格式



----结束

10.2.4 准备安全认证

前提条件

MRS服务集群开启了Kerberos认证，没有开启Kerberos认证的集群忽略该步骤。

准备认证机制代码

在开启Kerberos认证的环境下，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。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。提供了LoginUtil相关接口来完成这些配置，如下样例代码中只需要配

置用户自己申请的账号名称和对应的keytab文件名称即可，由于人机账号的keytab会随用户密码过期而失效，故建议使用机机账号进行配置。

认证样例代码：

设置keytab认证文件模块

```
/**
 * 用户自己申请的账号keytab文件名称
 */
private static final String USER_KEYTAB_FILE = "用户自己申请的账号keytab文件名称";

/**
* 用户自己申请的账号名称
*/
private static final String USER_PRINCIPAL = "用户自己申请的账号名称";
```

MRS服务Kerberos认证模块，如果服务没有开启kerberos认证，这块逻辑不执行

```
public static void securityPrepare() throws IOException
{
    String filePath = System.getProperty("user.dir") + File.separator + "conf" + 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.hadoop.com");
    LoginUtil.setJaasFile(USER_PRINCIPAL, userKeyTableFile);
}
```

说明

如果修改了集群kerberos域名，需要在代码中增加kerberos.domain.name的配置，并按照hadoop.expr=toLowerCase(%{default_realm}%{KerberosServer})规则配置正确的域名信息。例如：修改域名为HUAWEI.COM，则配置为hadoop.huawei.com。

keytab 文件获取方式

1. 访问开启Kerberos的MRS Manager，访问方式详见《MapReduce服务用户指南》的“访问 MRS Manager”章节。
2. 进入“系统设置->用户管理”，在指定的用户所在行单击“更多 >下载认证凭据”。
3. 将下载获取到的zip文件解压缩，获取krb5.conf和该用户的keytab文件。
4. 将krb5.conf和该用户的keytab文件拷贝到样例工程的conf目录中。

10.3 开发程序

10.3.1 典型场景说明

场景说明

Kafka是一个分布式消息系统，在此系统上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。

10.3.2 Old Producer API 使用样例

功能介绍

Producer是消息生产者的角色，负责发布消息到Kafka Broker。

下面代码片段在com.huawei.bigdata.kafka.example.Old_Producer类中，作用在于每秒向指定的Topic发送一条消息。（注意：Old Producer API仅支持通过不启用Kerberos认证模式端口访问未设置ACL的Topic，安全接口说明见**安全接口说明**）

样例代码

Old Producer API的run方法中的逻辑

```
/*
 * 启动执行producer，每秒发送一条消息。
 */
public void run()
{
    LOG.info("Old Producer: start.");
    int messageNo = 1;

    while (true)
    {
        String messageStr = new String("Message_" + messageNo);

        // 指定消息序号作为key值
        String key = String.valueOf(messageNo);
        producer.send(new KeyedMessage<String, String>(topic, key, messageStr));
        LOG.info("Producer: send " + messageStr + " to " + topic);
        messageNo++;

        // 每隔1s，发送1条消息
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

10.3.3 Old Consumer API 使用样例

功能介绍

每一个Consumer实例都属于一个Consumer group，每一条消息只会被同一个Consumer group里的一个Consumer实例消费（不同的Consumer group可以同时消费同一条消息）。

下面代码片段在com.huawei.bigdata.kafka.example.Old_Consumer类中，作用在于订阅指定Topic的消息。（注意：旧Consumer API仅支持访问未设置ACL的Topic，安全接口说明见**安全接口说明**）

样例代码

Old Consumer API线程run方法中的消费逻辑

```
/**  *启动执行Consumer，订阅Kafka上指定topic消息。  */
 public void run()
 {
   LOG.info("Consumer: start.");

   Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
   topicCountMap.put(topic, new Integer(1));
   Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer.createMessageStreams(topicCountMap);
   List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);

   LOG.info("Consumerstreams size is : " + streams.size());

   for (KafkaStream<byte[], byte[]> stream : streams)
   {
     ConsumerIterator<byte[], byte[]> it = stream.iterator();

     while (it.hasNext())
     {
       LOG.info("Consumer: receive " + new String(it.next().message()) + " from " + topic);
     }
   }

   LOG.info("Consumer End.");
 }
```

10.3.4 Producer API 使用样例

功能介绍

下面代码片段在com.huawei.bigdata.kafka.example.Producer类中，用于实现新Producer API向安全Topic生产消息。

样例代码

Producer线程run方法中的消费逻辑

```
    public void run()
    {
        LOG.info("New Producer: start.");
        int messageNo = 1;
        // 指定发送多少条消息后sleep1秒
        int intervalMessages=10;

        while (messageNo <= messageNumToSend)
        {
            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();
            }
            catch (InterruptedException ie)
            {
                LOG.info("The InterruptedException occured : {}.", ie);
            }
            catch (ExecutionException ee)
            {
                LOG.info("The ExecutionException occured : {}.", ee);
            }
        }
        messageNo++;

        if (messageNo % intervalMessages == 0)
        {
            // 每发送intervalMessage条消息sleep1秒
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
            LOG.info("The Producer have send {} messages.", messageNo);
        }
    }

}
```

10.3.5 Consumer API 使用样例

功能介绍

下面代码片段在com.huawei.bigdata.kafka.example.Consumer类中，用于消费订阅的Topic消息。

代码样例

Consumer线程的dowork方法逻辑，该方法是run方法的重写

```
/**
 * 订阅Topic的消息处理函数
 */
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("[NewConsumerExample], Received message: (" + record.key() + ", " + record.value()
            + ") at offset " + record.offset());
    }
}
```

10.3.6 多线程 Producer API 使用样例

功能介绍

在**Producer API使用样例**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类中，用于实现多线程生产数据。

代码样例

生产者类线程类的run方法逻辑

```
    /**
     * 生产者线程执行函数，循环发送消息。
     */
    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
            Integer key = new Integer(sendThreadId);

            long startTime = System.currentTimeMillis();

            // 构造消息记录
            ProducerRecord<Integer, String> record = new ProducerRecord<Integer, String>(topic, key,
messageStr);

            if (isAsync)
            {
                // 异步发送
                producer.send(record, new DemoCallBack(startTime, key, messageStr));
            }
            else
            {
                try
                {
                    // 同步发送
                    producer.send(record).get();
                }
                catch (InterruptedException ie)
                {
                    LOG.info("The InterruptedException occured : {}.", ie);
                }
                catch (ExecutionException ee)
                {
                    LOG.info("The ExecutionException occured : {}.", ee);
                }
            }
            LOG.info("Producer: send " + messageStr + " to " + topic + " with key: " + key);
            messageCount++;

            // 每隔1s，发送1条消息
            try
```

```
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
```

ProducerMultThread主类的线程启动逻辑

```
/**
 * 启动多个线程进行发送
 */
public void run()
{
    // 是否使用异步发送模式
    final boolean asyncEnable = false;

    // 指定的线程号，仅用于区分不同的线程
    for (int threadNum = 0; threadNum < PRODUCER_THREAD_COUNT; threadNum++)
    {
        ProducerThread producerThread = new ProducerThread(topic, asyncEnable, threadNum);
        producerThread.start();
    }

}
```

10.3.7 多线程 Consumer API 使用样例

功能介绍

在**Consumer API使用样例**基础上，实现了多线程并发消费，可根据Topic的Partition数目起相应个数的Consumer线程来对应消费消息。

下面代码片段在com.huawei.bigdata.kafka.example.ConsumerMultThread类中，用于实现对指定Topic的并发消费。

Kafka不支持无缝集成SpringBoot项目。

代码样例

单个消费者线程的doWork()方法逻辑（ run方法重写 ）

```
    /**
     * 订阅Topic的消息处理函数
     */
    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(receivedThreadId+"Received message: (" + record.key() + ", " + record.value()
                    + ") at offset " + record.offset());
        }
    }
```

ConsumerMultThread主类的线程启动逻辑

```
public void run()
{
    LOG.info("Consumer: start.");
```

```
    for (int threadNum = 0; threadNum < CONCURRENCY_THREAD_NUM; threadNum++)
    {
        Consumer consumerThread = new Consumer(KafkaProperties.TOPIC,threadNum);
        consumerThread.start();
    }
}
```

10.3.8 SimpleConsumer API 使用样例

功能介绍

下面代码片段在com.huawei.bigdata.kafka.example.SimpleConsumerDemo类中，用于实现使用新SimpleConsumer API订阅Topic，并进行消息消费。（注意：SimpleConsumer API仅支持访问未设置ACL的Topic，安全接口说明见**安全接口说明**）

SimpleConsumer API属于lowlevel的Consumer API需要访问zookeeper元数据，管理消费Topic队列的offset，一般情况不推荐使用。

代码样例

SimpleConsumer API主方法需要传入三个参数，最大消费数量、消费Topic、消费的Topic分区

```
public static void main(String args[])
  {
    // 允许读取的最大消息数
    long maxReads = 0;

    try
    {
        maxReads = Long.valueOf(args[0]);
    }
    catch (Exception e)
    {
        log.error("args[0] should be a number for maxReads.\n" +
            "args[1] should be a string for topic. \n" +
            "args[2] should be a number for partition.");
        return;
    }

    if (null == args[1])
    {
        log.error("args[0] should be a number for maxReads.\n" +
            "args[1] should be a string for topic. \n" +
            "args[2] should be a number for partition.");
        return;
    }

    // 消费的消息主题
    // String topic = KafkaProperties.TOPIC;
    String topic = args[1];

    // 消息的消息分区
    int partition = 0;
    try
    {
        partition = Integer.parseInt(args[2]);
    }
    catch (Exception e)
    {
        log.error("args[0] should be a number for maxReads.\n" +
            "args[1] should be a string for topic. \n" +
            "args[2] should be a number for partition.");
    }
```

```
    // Broker List
    String bkList = KafkaProperties.getInstance().getValues("metadata.broker.list", "localhost:9092");

    Map<String, Integer> ipPort = getIpPortMap(bkList);

    SimpleConsumerDemo example = new SimpleConsumerDemo();
    try
    {
        example.run(maxReads, topic, partition, ipPort);
    }
    catch (Exception e)
    {
        log.info("Oops:" + e);
        e.printStackTrace();
    }
}
```

10.3.9 样例工程配置文件说明

Conf目录个各配置文件及重要参数配置说明

- Producer API配置项

表 10-3 producer.properties 文件配置项

参数	描述	备注
security.protocol	安全协议类型	生产者使用的安全协议类型，当前Kerberos开启的模式下仅支持SASL协议，需要配置为SASL_PLAINTEXT。Kerberos未开启的模式下配置为PLAINTEXT。
kerberos.domain.name	域名	MRS服务集群的Kerberos域名，未开启Kerberos认证的集群无需配置。
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）。未开启Kerberos认证的集群无需配置。

- Consumer API配置项

表 10-4 consumer.properties 文件配置项

参数	描述	备注
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Kerberos开启的模式下仅支持SASL协议，需要配置为SASL_PLAINTEXT。Kerberos未开启的模式下配置为PLAINTEXT。
kerberos.domain.name	域名	MRS服务集群的Kerberos域名，未开启Kerberos认证的集群无需配置。
group.id	消费者的group id	-
auto.commit.interval.ms	是否自动提交offset	布尔值参数，默认值为true
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）。未开启Kerberos认证的集群无需配置。

- 客户端信息配置项

表 10-5 client.properties 文件配置项

参数	描述	备注
metadata.broker.list	元数据Broker地址列表	通过此参数值，创建与元数据Broker之间的连接，需要直接访问元数据的API需要用到此参数。访问端口仅支持不开启Kerberos模式下的端口，端口说明详见**安全接口说明**
kafka.client.zookeeper.principal	kafka集群访问zookeeper的认证和域名	-
bootstrap.servers	Broker地址列表	通过此参数值，创建与Broker之间的连接。端口配置项详见**安全接口说明**
zookeeper.connect	zookeeper地址列表	通过此参数，访问zookeeper，末尾需要带上kafka服务名kafka

- MRS服务是否开启Kerberos认证配置项

表 10-6 kafkaSecurityMode 文件配置项

参数	描述	备注
kafka.client.security.mode	kafka所在的MRS服务集群是否开启Kerberos认证配置项	若开启了Kerberos认证，设置为yes，否则设置为no。

- log4j日志配置项文件log4j.properties
  log4j日志框架的配置文件，默认情况不输入样例工程运行日志。

10.4 调测程序

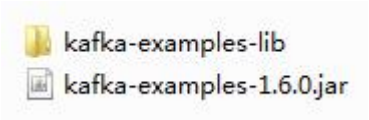
10.4.1 在 Linux 中调测程序

前提条件

- 客户端本地能登录MRS服务的弹性云服务器，登录方式详见“弹性云服务器《用户指南》”中“入门 > 登录弹性云服务器”的SSH登录方式。
- 样例工程在已经通过Maven编译。

示例：Maven 工程打包到 Linux 下运行样例

1. 执行**mvn package生成**jar包，在工程目录target目录下获取，比如:kafka-examples-1.6.0.jar。
2. 执行**mvn dependency:copy-dependencies -DoutputDirectory=kafka-examples-lib -DincludeScope=compile**，导出kafka样例工程依赖的jar包，比如放到kafka-examples-lib目录。
3. 在第一步指定的目录下生成一个Jar包和一个存放lib的文件夹。

4. 将刚才生成的依赖库文件夹（此处为“kafka-examples-lib”）拷贝到MRS服务的某个Linux环境上任意目录下，例如：“/opt/example”，然后将刚才生成的jar包拷贝到“/opt/example/kafka-examples-lib”目录下。
5. 将样例工程的conf目录拷贝到与依赖库文件夹同级目录下，即“/opt/example”目录下，并创建logs目录，用于记录jar包运行日志。
6. 切换到root用户，将拷贝进去的conf，kafka-examples-lib，logs目录修改为omm：wheel用户组所有，执行以下命令切换用户。

   sudo su - root

   chown -R omm:wheel /opt/example/*

   ```
   drwxr-xr-x. 2 omm wheel 4096 Jan 16 16:48 conf
   drwxr-xr-x. 2 omm wheel 4096 Jan 16 16:50 kafka-examples-lib
   drwxr-xr-x. 2 omm wheel 4096 Jan 16 16:50 logs
   ```

7. 切换为omm用户，进入拷贝目录下“/opt/example”，首先确保conf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，如**java -cp .:/opt/example/conf:/opt/example/kafka-examples-lib/* com.huawei.bigdata.kafka.example.Producer**，运行样例工程。

   su - omm

   chmod 750 /opt/example

   cd /opt/example

   java -cp .:/opt/example/conf:/opt/example/kafka-examples-lib/* com.huawei.bigdata.kafka.example.Producer

运行结果观察方式

样例程序工程jar包运行结果可以在logs目录下的client.log观察，默认状态下的log4j.properties没有将运行状态输出，若需要观察程序运行的信息，需将log4j.properties按如下方式配置：

```
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

kafka.logs.dir=logs

log4j.rootLogger=INFO, stdout, kafkaAppender

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.logger.kafka=INFO, kafkaAppender

log4j.appender.kafkaAppender=org.apache.log4j.DailyRollingFileAppender
log4j.appender.kafkaAppender.DatePattern='.'yyyy-MM-dd-HH
log4j.appender.kafkaAppender.File=${kafka.logs.dir}/client.log
log4j.appender.kafkaAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.kafkaAppender.layout.ConversionPattern=[%d] %p %m (%c)%n

# Turn on all our debugging info
#log4j.logger.kafka.producer.async.DefaultEventHandler=DEBUG, kafkaAppender
#log4j.logger.kafka.client.ClientUtils=DEBUG, kafkaAppender
#log4j.logger.kafka.perf=DEBUG, kafkaAppender
#log4j.logger.kafka.perf.ProducerPerformance$ProducerThread=DEBUG, kafkaAppender
#log4j.logger.org.I0Itec.zkclient.ZkClient=DEBUG
```

将kafkaAppender添加到rootLogger，并将日志级别调整到需要观察的级别。

10.5 Kafka 接口

10.5.1 Shell 命令

前提条件

Kafka的Linux客户端已安装。安装方法可参考**安装客户端**。

常用的 Shell 命令指南

Shell命令执行方法：

步骤1 进入Kafka客户端任意目录。

步骤2 初始化环境变量。

source /opt/client/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（该用户需要加入kafkaadmin用户组拥有Kafka管理员权限）。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 admin**

步骤4 进入“*客户端安装目*录/Kafka/kafka/bin”执行Kafka shell命令。

----结束

常用的命令如下：

- 查看当前集群Topic列表。

  sh kafka-topics.sh --list --zookeeper <ZooKeeper集群IP:2181/kafka>
- 查看单个Topic详细信息。

  sh kafka-topics.sh --describe --zookeeper <ZooKeeper集群IP:2181/kafka> --topic <Topic名称>
- 删除Topic，由管理员用户操作。

  sh kafka-topics.sh --delete --zookeeper <ZooKeeper集群IP:2181/kafka> --topic <Topic名称>
- 创建Topic，由管理员用户操作。

  sh kafka-topics.sh --create --zookeeper <ZooKeeper集群IP:2181/kafka> --partitions 6 --replication-factor 2 --topic <Topic名称>
- Old Producer API生产数据。

  sh kafka-console-producer.sh --broker-list <Kafka集群IP:9092> --topic <Topic名称> --old-producer -sync
- Old Consumer API消费数据。

  sh kafka-console-consumer.sh --zookeeper <ZooKeeper集群IP:2181/kafka> --topic <Topic名称> --from-beginning
- Producer API生产消息，需要拥有该Topic生产者权限。

  sh kafka-console-producer.sh --broker-list <Kafka集群IP:21007> --topic <Topic名称> --producer.config config/producer.properties
- Consumer API消费数据，需要拥有该Topic的消费者权限

sh kafka-console-consumer.sh --topic <Topic名称> --bootstrap-server <Kafka集群IP:21007> --new-consumer --consumer.config config/consumer.properties

说明

- 未开启Kerberos认证的Kafka集群端口默认为9092，开启Kerberos认证的Kafka集群端口默认为21007。
- 登录**MRS Manager**，选择“服务管理 > ZooKeeper > 实例”，获取ZooKeeper的quorumpeer实例的“管理IP”地址。
- 登录**MRS Manager**，选择“服务管理 > Kafka > 实例”，获取Kafka的broker实例的“管理IP”地址。

10.5.2 Java API

Kafka相关接口同开源社区保持一致，详情请参见**http://kafka.apache.org/documentation.html#api**。

10.5.3 安全接口说明

1. 访问开启Kerberos认证模式Kafka集群端口默认为21007，访问没有开启Kerberos认证模式集群端口默认为21005。
2. 旧API仅支持访问9092端口；新API兼容访问没有开启Kerberos模式集群端口9092和开启Kerberos认证模式集群端口21007。

10.6 FAQ

10.6.1 已经拥有 Topic 访问权限，但是运行 Producer.java 样例运行获取元数据失败“ERROR fetching topic metadata for topics...”的解决办法

解决步骤

1. 检查工程conf目录下“client.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（开启kerberos认证模式端口），那么修改该端口为9092（没有开启kerberos认证模式端口）。
2. 检查网络是否正常，确保当前机器能够正常访问Kafka集群。

10.7 开发规范

10.7.1 规则

调用 Kafka API（AdminUtils.createTopic）创建 Topic 时，需要设置 ZkStringSerializer

- 对于Java开发语言，正确示例：

```
import org.I0Itec.zkclient.ZkClient;
import kafka.utils.ZKStringSerializer$;
…
ZkClient zkClient = new ZkClient(zkconnectstring, zkSessionTimeout, zkConnectionTimeout,
ZKStringSerializer$.MODULE$);
AdminUtils.createTopic(zkClient, topic, partitions, replicationFactor, new Properties());
…
```

- 对于Scala开发语言，正确示例：

```
import org.I0Itec.zkclient.ZkClient;
import kafka.utils.ZKStringSerializer;
…
var zkclient: ZkClient = new ZkClient(zkconnectstring, zkSessionTimeout, zkConnectionTimeout,
ZKStringSerializer)
AdminUtils.createTopic(zkClient, topic, partitions, replicationFactor, new Properties())
```

Partition 的副本数不要超过节点个数

Kafka中Topic的Partition的副本是为了提升数据的可靠性而存在的，同一个Partition的副本会分布在不同的节点，因此副本数不允许超过节点个数。

Consumer 客户端的配置参数“fetch.message.max.bytes”大小

Consumer客户端的配置参数“fetch.message.max.bytes”必须大于等于Producer客户端每次产生的消息最大字节数。如果参数的值太小，可能导致Producer产生的消息无法被Consumer成功消费。

10.7.2 建议

同一个组的消费者的数量建议与待消费的 Topic 下的 Partition 数保持一致

若同一个组的消费者数量多于Topic的Partition数时，会有多余的消费者一直无法消费该Topic的消息，若消费者数量少于Topic的Partition数时，并发消费得不到完全体现，因此建议两者相等。

11 Presto 应用开发

11.1 概述

11.1.1 应用开发简介

Presto 简介

Presto是一种开源、分布式SQL查询引擎，用于对千兆字节至PB级大小的数据源进行交互式分析查询。

Presto主要特点如下：

- 多数据源：Presto可以支持Mysql，Hive，JMX等多种Connector。
- 支持SQL：Presto完全支持ANSI SQL，用户可以直接使用SQL Shell进行查询。
- 混合计算：用户可以对多个Catalog进行join查询。

11.1.2 常用概念

- **Connector**
  Connector将Presto适配到如Hive或关系型数据库的数据源。
- **Catalog**
  Catalog包含schema以及引用通过connector连接的数据源
- **Schema**
  Schema是组织数据表的一种形式。

11.1.3 开发流程

开发流程中各阶段的说明如**图11-1**和**表11-1**所示。

图 11-1 Presto 应用程序开发流程

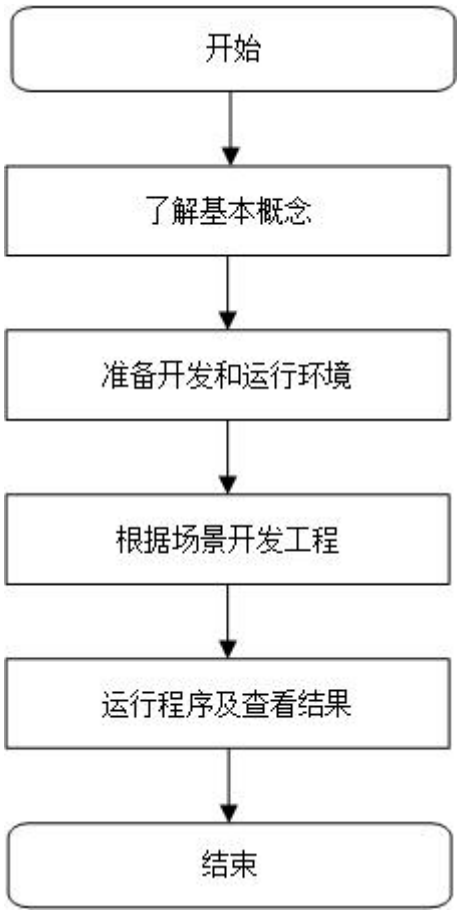


表 11-1 Presto 应用开发的流程说明

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Presto的基本概念。	**常用概念**
准备开发和运行环境	Presto的应用程序支持使用Java进行开发。推荐使用Eclipse工具，请根据指导完成开发环境配置。	**开发环境简介**
根据场景开发工程	提供了Java语言的样例工程和数据查询的样例工程。	**典型场景说明**
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。	**JDBC客户端运行及结果查看**

11.2 环境准备

11.2.1 开发环境简介

在进行应用开发时，要准备的本地开发环境如表11-2所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行是否正常。

表 11-2 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：Linux系统。
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。 **说明** 在Presto的开发环境中，基于安全考虑，MRS服务端只支持TLS 1.1和TLS 1.2加密协议。由于IBM JDK默认TLS只支持1.0，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TLS1.0/1.1/1.2。详情请参见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Eclipse	用于开发Presto应用程序的工具。版本要求如下： • JDK使用1.7版本，Eclipse使用3.7.1及以上版本。 • JDK使用1.8版本，Eclipse使用4.3.2及以上版本。 说明： • 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。 • 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。 • 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。
网络	确保客户端与Presto服务主机在网络上互通。

11.2.2 准备环境

- 选择Windows开发环境下，安装Eclipse，安装JDK。
  建议JDK使用1.8版本，Eclipse使用4.3.2及以上版本。

  说明

  - 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
  - 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
  - 若使用ODBC进行二次开发，请确保JDK版本为1.8及以上版本。
  - 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。
- 准备一个应用程序运行测试的Linux环境。

准备运行调测环境

步骤1 在弹性云服务器管理控制台，申请一个新的弹性云服务器，用于用户应用程序开发、运行、调测。

- 弹性云服务器的安全组需要和MRS集群Master节点的安全组相同。
- 弹性云服务器的VPC需要与MRS集群在同一个VPC中。
- 弹性云服务器的网卡需要与MRS集群在同一个网段中。

步骤2 在弹性云服务器页面申请并绑定弹性云服务器IP，具体请参考**为弹性云服务器申请和绑定弹性公网IP**。

步骤3 配置安全组出入规则，具体请参考**配置安全组规则**。

步骤4 下载客户端程序。

1. 登录MRS Manager系统。
2. 选择“服务管理 > 下载客户端 ”，下载“完整客户端”到“远端主机”上，即下载客户端程序到新申请的弹性云服务器上。

步骤5 以root用户安装集群客户端。

1. 执行以下命令解压客户端包。

   tar -xvf /opt/MRS_Services_Client.tar

2. 执行以下命令校验安装文件包。

   sha256sum -c /opt/MRS_Services_ClientConfig.tar.sha256

   ```
   MRS_Services_ClientConfig.tar:OK
   ```

3. 执行以下命令解压安装文件包。

   tar -xvf /opt/MRS_Services_ClientConfig.tar

4. 执行以下命令安装客户端到指定目录（绝对路径），例如“/opt/client”。目录会自动创建。

   cd /opt/MRS_Services_ClientConfig

   sh install.sh /opt/client

   ```
   Components client installation is complete.
   ```

----结束

11.2.3 准备开发用户

开发用户用于运行样例工程。用户需要有Presto权限，才能运行Presto样例工程。若MRS集群开启了Kerberos认证需要执行该步骤，没有开启Kerberos认证的集群请忽略该步骤。

操作步骤

步骤1 登录MRS Manager页面。

步骤2 单击“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。

步骤3 填写用户名，例如*prestouser*，用户类型为“机机”用户，加入用户组**presto**，设置其“主组”为**presto**，单击“确定”。

步骤4 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名*prestouser*对应的“操作”列选择“更多 > 下载认证凭据”，如图11-2所示。保存后解压得到用户的user.keytab文件与krb5.conf文件。用于在样例工程中进行安全认证。

图 11-2 下载认证凭据

说明

如果修改了组件的配置参数，需重新下载客户端配置文件并更新运行调测环境上的客户端。

----结束

11.2.4 准备 JDBC 客户端开发环境

为了运行Presto组件的JDBC接口样例代码，需要完成下面的操作。此处以在Windows环境下开发JDBC方式连接Presto服务的应用程序为例。

操作步骤

步骤1 在**样例工程获取地址**获取Presto示例工程。

步骤2 在Presto示例工程根目录，执行mvn install编译。

步骤3 在Presto示例工程根目录，执行mvn eclipse:eclipse创建Eclipse工程。

步骤4 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。

   显示“浏览文件夹”对话框。

2. 选择文件夹“presto-examples”。Windows下要求该文件夹的完整路径不包含空格。
3. 单击“Finish”。

   导入成功后，PrestoJDBCExample类，为JDBC接口样例代码。

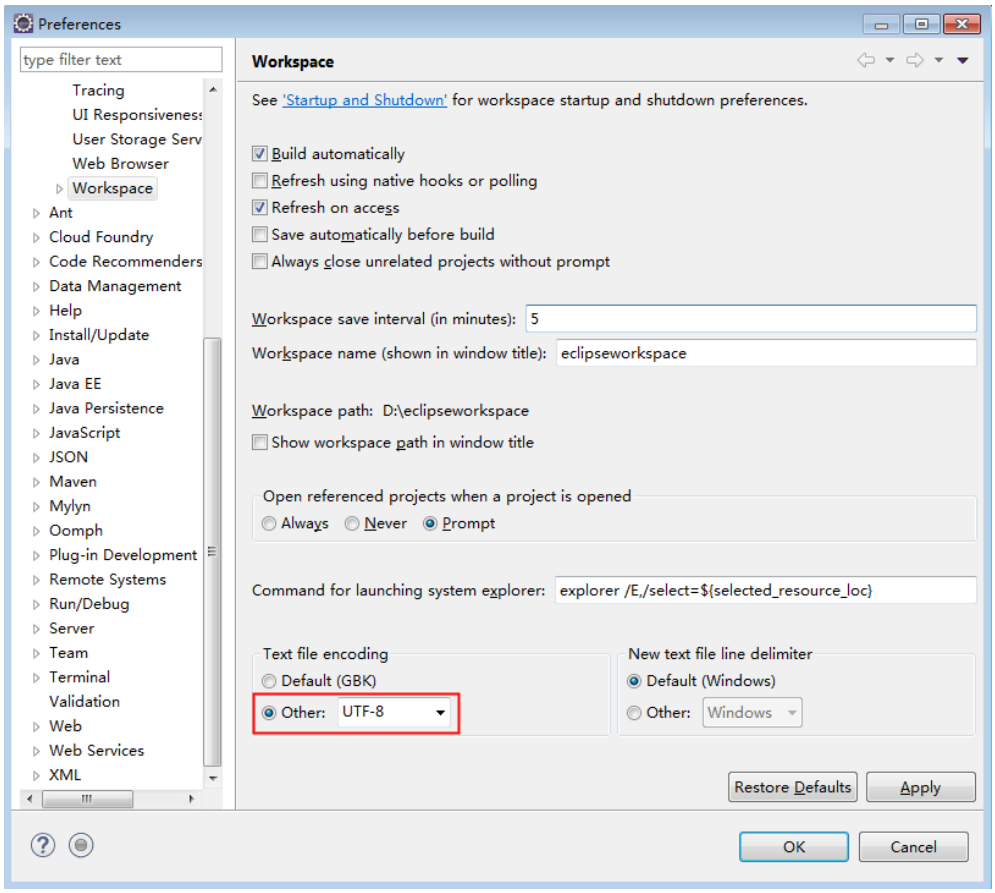
步骤5 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。

   弹出“Preferences”窗口。

2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如**图11-3**所示。

图 11-3 设置 Eclipse 的编码格式



步骤6 修改样例（未开启Kerberos认证集群可跳过此步骤）。

在**步骤4**获取新建开发用户的krb5.conf和user.keytab文件后，修改presto.properties中的KerberosPrincipal为对应新建用户的principal，修改KerberosConfigPath为对应新建用户的krb5.conf文件路径，KerberosKeytabPath为对应新建用户的keytab文件路径。

----结束

11.2.5 准备 HCatalog 开发环境

为了运行Presto组件的HCatalog接口样例代码，需要完成下面的操作。此处以在Windows环境下开发HCatalog方式连接Presto服务的应用程序为例。

操作步骤

步骤1 在**样例工程获取地址** 获取Presto示例工程。

步骤2 在Presto示例工程根目录，执行mvn install编译。

步骤3 在Presto示例工程根目录，执行mvn eclipse:eclipse创建Eclipse工程。

步骤4 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。

   显示“浏览文件夹”对话框。

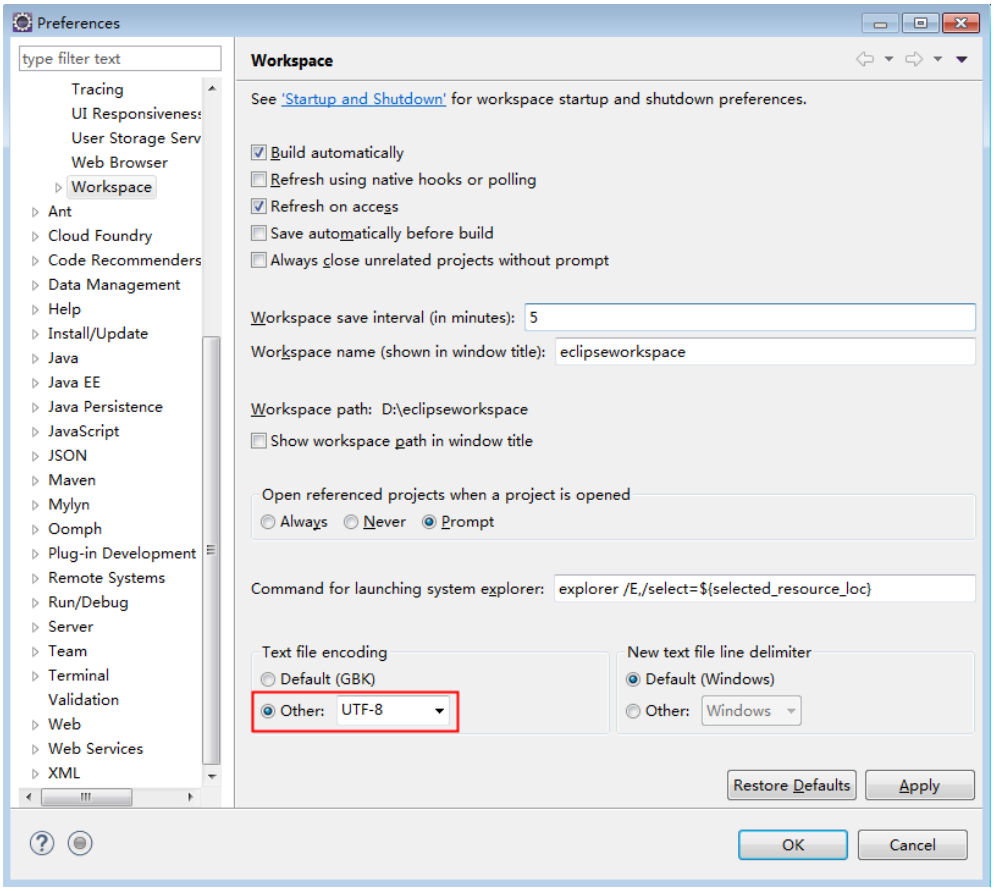
2. 下载工程后选择文件夹“presto-examples”。Windows下要求该文件夹的完整路径不包含空格。
3. 单击“Finish”。

   导入成功后，com.huawei.bigdata.presto.example包下的HCatalogExample类，为HCatalog接口样例代码。

步骤5 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。弹出“Preferences”窗口。
2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图11-4所示。

图 11-4 设置 Eclipse 的编码格式



----结束

11.3 开发程序

11.3.1 典型场景说明

场景说明

假定用户开发一个Presto数据分析应用，用于获取Presto提供的TPCDS Catalog的call_center表。

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**数据加载**。

   雇员信息数据如**表11-3**所示。

表 11-3 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11-4所示。

表 11-4 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**数据查询**。

- 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**样例程序指导**。

----结束

11.3.2 样例代码说明

Presto JDBC 使用样例

下面的代码片段在PrestoJDBCExample类中，用于实现JDBC连接Presto TPCDS Catalog。

```
private static Connection connection;
private static Statement statement;
/**
 * Only when Kerberos authentication enabled, configurations in presto-examples/conf/presto.properties
 * should be set. More details please refer to https://prestodb.io/docs/0.215/installation/jdbc.html.
*/
private static void initConnection(String url, boolean krbsEnabled) throws SQLException {
    if (krbsEnabled) {
        String filePath = System.getProperty("user.dir") + File.separator + "conf" + File.separator;
        File proFile = new File(filePath + "presto.properties");if (proFile.exists()) {
            Properties props = new Properties();
            try {
                props.load(new FileInputStream(proFile));
            } catch (IOException e) {
                e.printStackTrace();
            }
            connection = DriverManager.getConnection(url, props);
        }
    } else {
```



```
            connection = DriverManager.getConnection(url, "presto", null);
        }
        statement = connection.createStatement();
    }

    private static void releaseConnection() throws SQLException {
        statement.close();
        connection.close();
    }

    public static void main(String[] args) throws SQLException {
        try {
            /**
             * Replace example_ip with your cluster presto server ip.
             * By default, Kerberos authentication disabled cluster presto service port is 7520, Kerberos
             * authentication enabled cluster presto service port is 7521
             * The postfix /tpcds/sf1 means to use tpcds catalog and sf1 schema, you can use hive catalog as well
             * If Kerberos authentication enabled, set the second param to true.
             * see PrestoJDBCExample#initConnection(java.lang.String, boolean).
             */
            initConnection("jdbc:presto://example_ip:7520/tpcds/sf1", false);
            //initConnection("jdbc:presto://example_ip:7521/tpcds/sf1", true);
            ResultSet resultSet = statement.executeQuery("select * from call_center");
            while (resultSet.next()) {
                System.out.println(resultSet.getString("cc_name") + " : " + resultSet.getString("cc_employees"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            releaseConnection();
        }
    }
```

11.4 调测程序

JDBC 客户端运行及结果查看

步骤1 执行mvn clean compile assembly:single生成jar包，在工程目录target目录下获取，比如:presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar。

步骤2 在运行调测环境上创建一个目录作为运行目录，如或“/opt/presto_examples”(Linux环境)，并在该目录下创建子目录“conf”。

将**步骤1**导出的presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar拷贝到“/opt/presto_examples”下。

步骤3 开启Kerberos认证集群需要将**步骤4**获取的user.keytab和krb5.conf拷贝到的/opt/presto_examples/conf下，并修改样例代码中conf目录下的presto.preperties。未开启Kerberos认证集群无须执行此步骤。

表 11-5 presto.preperties 参数说明

参数	说明
user	用于Kerberos认证的用户名，即**准备开发用户**中创建的开发用户的用户名。
KerberosPrincipal	用于认证的名字，即认证**准备开发用户**中创建的开发用户的用户名。
KerberosConfigPath	krb5.conf的路径。

参数	说明
KerberosKeytabPath	user.keytab的路径。

presto.preperties样例

```
user = prestouser
SSL = true
KerberosRemoteServiceName = HTTP
KerberosPrincipal = prestouser
KerberosConfigPath = /opt/presto_examples/conf/krb5.conf
KerberosKeytabPath = /opt/presto_examples/conf/user.keytab
```

步骤4 在Linux环境下执行运行样例程序。

```
chmod +x /opt/presto_examples -R
cd /opt/presto_examples
java -jar presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar
```

步骤5 在命令行终端查看样例代码所查询出的结果。

Linux环境运行成功结果会有如下信息：

```
NY Metro : 2
Mid Atlantic : 6
Mid Atlantic : 6
North Midwest : 1
North Midwest : 3
North Midwest : 7
```

----结束

11.4.1 在 Windows 中调测程序

步骤1 申请一台Windows的ECS访问MRS集群操作Presto。申请ECS访问MRS集群的步骤如下：

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“弹性云服务器 > 快速入门 > 购买并登录Windows弹性云服务器”。

步骤2 开启Kerberos认证集群需要在Windows上配置集群的ip与主机名映射关系。登录集群后台，执行命令**cat /etc/hosts**后，把hosts文件中的ip与hostname映射关系拷贝到ECS的“C:\Windows\System32\drivers\etc\hosts”中。未开启Kerberos认证集群无须执行此步骤。

步骤3 在Windows下操作Presto集群时，JDK版本需为jdk1.8.0_60及以上版本。开启Kerberos认证集群需要拷贝MRS集群主节点“/opt/Bigdata/om-0.0.1/packaged-distributables/client_packet/ca.crt”到Windows的ECS上，在jdk/bin目录打开cmd命令行执行如下命令，并修改样例代码中conf目录下的presto.properties文件的配置。

jdk1.8.0_242以上版本需要删除krb5.conf中[libdefaults]的renew_lifetime = 0m。

未开启Kerberos认证集群无须执行此步骤。

keytool -import -v -trustcacerts -alias presto_trust -file <ca.crt_path> -keystore <keystore_path>/truststore.jks –keypass <password>

其中，<ca.crt_path>为拷贝的ca.crt文件路径，<keystore_path>为truststore.jks文件生成路径，<password>为truststore密码，可根据需要指定。命令中如果携带认证密码信息可能存在安全风险，在执行命令前建议关闭系统的history命令记录功能，避免信息泄露。

表 11-6 presto.properties 参数说明

参数	说明
user	用于Kerberos认证的用户名，即**准备开发用户**中创建的开发用户的用户名。
KerberosPrincipal	用于认证的名字，即认证**准备开发用户**中创建的开发用户的用户名。
KerberosConfigPath	krb5.conf的路径。 注意转义“\”
KerberosKeytabPath	user.keytab的路径。 注意转义“\”
SSLTrustStorePath	truststore.jks的路径。 注意转义“\”
SSLTrustStorePassword	truststore的密码。

步骤4 修改并运行样例。

1. 在开发环境中（例如Eclipse中），修改样例代码example_ip、端口号和krbsEnabled配置。
2. 右键单击“PrestoJDBCExample.java”。
3. 单击“Run as > Java Application”运行对应的应用程序工程。

步骤5 查看结果。运行成功结果会有如下信息：

```
NY Metro : 2
Mid Atlantic : 6
Mid Atlantic : 6
North Midwest : 1
North Midwest : 3
North Midwest : 7
```

----结束

11.4.2 在 Linux 中调测程序

JDBC 客户端运行及结果查看

步骤1 执行mvn clean compile assembly:single生成jar包，在工程目录target目录下获取，比如:presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar。

步骤2 在运行调测环境上创建一个目录作为运行目录，如或“/opt/presto_examples”(Linux环境)，并在该目录下创建子目录“conf”。

将**步骤1**导出的presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar拷贝到“/opt/presto_examples”下。

步骤3 开启Kerberos认证集群需要将**步骤4**获取的user.keytab和krb5.conf拷贝到的/opt/presto_examples/conf下，并修改样例代码中conf目录下的presto.preperties。未开启Kerberos认证集群无须执行此步骤。

表 11-7 presto.preperties 参数说明

参数	说明
user	用于Kerberos认证的用户名，即**准备开发用户**中创建的开发用户的用户名。
KerberosPrincipal	用于认证的名字，即认证**准备开发用户**中创建的开发用户的用户名。
KerberosConfigPath	krb5.conf的路径。
KerberosKeytabPath	user.keytab的路径。

presto.preperties样例

```
user = prestouser
SSL = true
KerberosRemoteServiceName = HTTP
KerberosPrincipal = prestouser
KerberosConfigPath = /opt/presto_examples/conf/krb5.conf
KerberosKeytabPath = /opt/presto_examples/conf/user.keytab
```

步骤4 在Linux环境下执行运行样例程序。

```
chmod +x /opt/presto_examples -R
cd /opt/presto_examples
java -jar presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar
```

步骤5 在命令行终端查看样例代码所查询出的结果。

Linux环境运行成功结果会有如下信息：

```
NY Metro : 2
Mid Atlantic : 6
Mid Atlantic : 6
North Midwest : 1
North Midwest : 3
North Midwest : 7
```

----结束

11.5 Presto 接口

Presto JDBC接口遵循标准的JAVA JDBC驱动标准，详情请参见JDK1.7 API。Presto JDBC使用参见**https://prestodb.io/docs/current/installation/jdbc.html**。

11.6 FAQ

11.6.1 在集群外节点运行 PrestoJDBCExample 缺少证书

问题

presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar在集群内节点运行时正常，但在集群外节点运行PrestoJDBCExample连接开启Kerberos认证的集群缺少证书，报错如下:

```
java.sql.SQLException: Error executing query
        at
com.facebook.presto.jdbc.PrestoStatement.internalExecute(PrestoStatement.java:274)
        at com.facebook.presto.jdbc.PrestoStatement.execute(PrestoStatement.java:227)
        at
com.facebook.presto.jdbc.PrestoStatement.executeQuery(PrestoStatement.java:76)
        at
PrestoJDBCExample.main(PrestoJDBCExample.java:65)
Caused by: java.io.UncheckedIOException:
javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException:
PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find
valid certification path to requested target
        at
com.facebook.presto.jdbc.internal.client.JsonResponse.execute(JsonResponse.java:154)
        at
com.facebook.presto.jdbc.internal.client.StatementClientV1.<init>(StatementClientV1.java:129)
        at
com.facebook.presto.jdbc.internal.client.StatementClientFactory.newStatementClient(StatementClientFactory.
java:24)
        at
com.facebook.presto.jdbc.QueryExecutor.startQuery(QueryExecutor.java:46)
        at
com.facebook.presto.jdbc.PrestoConnection.startQuery(PrestoConnection.java:683)
        at
com.facebook.presto.jdbc.PrestoStatement.internalExecute(PrestoStatement.java:239)
        ... 3 more
Caused by: javax.net.ssl.SSLHandshakeException:
sun.security.validator.ValidatorException: PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find
valid certification path to requested target
        at
sun.security.ssl.Alerts.getSSLException(Alerts.java:192)
        at
sun.security.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1959)
        at
sun.security.ssl.Handshaker.fatalSE(Handshaker.java:302)
        at
sun.security.ssl.Handshaker.fatalSE(Handshaker.java:296)
        at
sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1514)
        at
sun.security.ssl.ClientHandshaker.processMessage(ClientHandshaker.java:216)
        at
sun.security.ssl.Handshaker.processLoop(Handshaker.java:1026)
        at
sun.security.ssl.Handshaker.process_record(Handshaker.java:961)
        at
sun.security.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:1072)
        at
sun.security.ssl.SSLSocketImpl.performInitialHandshake(SSLSocketImpl.java:1385)
        at
sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:1413)
```

```
        at
sun.security.ssl.SSLSocketImpl.startHandshake(SSLSocketImpl.java:1397)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.connection.RealConnection.connectTls(RealConnection.java:318)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.connection.RealConnection.establishProtocol(RealConnection.java:282)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.connection.RealConnection.connect(RealConnection.java:167)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.connection.StreamAllocation.findConnection(StreamAllocation.java:257)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.connection.StreamAllocation.findHealthyConnection(StreamAllocation.java:135)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.connection.StreamAllocation.newStream(StreamAllocation.java:114)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.connection.ConnectInterceptor.intercept(ConnectInterceptor.java:42)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.cache.CacheInterceptor.intercept(CacheInterceptor.java:93)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.BridgeInterceptor.intercept(BridgeInterceptor.java:93)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RetryAndFollowUpInterceptor.intercept(RetryAndFollowUpInterceptor.java:126)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
        at
com.facebook.presto.jdbc.internal.client.SpnegoHandler.intercept(SpnegoHandler.java:109)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
        at
com.facebook.presto.jdbc.internal.client.OkHttpUtil.lambda$userAgent$0(OkHttpUtil.java:77)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
        at
com.facebook.presto.jdbc.internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
        at
com.facebook.presto.jdbc.internal.okhttp3.RealCall.getResponseWithInterceptorChain(RealCall.java:200)
```

```
        at
com.facebook.presto.jdbc.internal.okhttp3.RealCall.execute(RealCall.java:77)
        at
com.facebook.presto.jdbc.internal.client.JsonResponse.execute(JsonResponse.java:131)
        ... 8 more
Caused by: sun.security.validator.ValidatorException: PKIX
path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find
valid certification path to requested target
        at
sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:397)
        at
sun.security.validator.PKIXValidator.engineValidate(PKIXValidator.java:302)
        at
sun.security.validator.Validator.validate(Validator.java:260)
        at
sun.security.ssl.X509TrustManagerImpl.validate(X509TrustManagerImpl.java:324)
        at
sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:229)
        at
sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:124)
        at
sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1496)
        ... 41 more
Caused by: sun.security.provider.certpath.SunCertPathBuilderException:
unable to find valid certification path to requested target
        at
sun.security.provider.certpath.SunCertPathBuilder.build(SunCertPathBuilder.java:141)
        at
sun.security.provider.certpath.SunCertPathBuilder.engineBuild(SunCertPathBuilder.java:126)
        at
java.security.cert.CertPathBuilder.build(CertPathBuilder.java:280)
        at
sun.security.validator.PKIXValidator.doBuild(PKIXValidator.java:392)
        ... 47 more
```

回答

通过https协议连接安全集群，服务端的证书没有被认证，导致连接失败。

可以用集群内节点上java jdk目录下的cacerts（例如：/opt/Bigdata/jdk1.8.0_232/jre/lib/security/cacerts）替换当前节点java jdk目录下的cacerts来解决。

11.6.2 在集群外节点连接开启 Kerberos 认证的集群，HTTP 在 Kerberos 数据库中无法找到相应的记录

问题

presto-examples-1.0-SNAPSHOT-jar-with-dependencies.jar在集群内节点运行时正常，但在集群外节点运行PrestoJDBCExample连接开启Kerberos的集群时出现以下两种报错。

报错1：

```
java.sql.SQLException:
Kerberos error for [HTTP@10.33.11.138]: No valid credentials provided
(Mechanism level: No valid credentials provided (Mechanism level: Server not
found in Kerberos database (7) - UNKNOWN_SERVER))
at
io.prestosql.jdbc.PrestoStatement.internalExecute(PrestoStatement.java:281)
at
io.prestosql.jdbc.PrestoStatement.execute(PrestoStatement.java:229)
at
io.prestosql.jdbc.PrestoStatement.executeQuery(PrestoStatement.java:78)
```

```
at PrestoJDBCExample.main(PrestoJDBCExample.java:68)
Caused by:
io.prestosql.jdbc.$internal.client.ClientException: Kerberos error for
[HTTP@10.33.11.138]: No valid credentials provided (Mechanism level: No valid
credentials provided (Mechanism level: Server not found in Kerberos database
(7) - UNKNOWN_SERVER))
at
io.prestosql.jdbc.$internal.client.SpnegoHandler.generateToken(SpnegoHandler.java:174)
at
io.prestosql.jdbc.$internal.client.SpnegoHandler.authenticate(SpnegoHandler.java:140)
at
io.prestosql.jdbc.$internal.client.SpnegoHandler.authenticate(SpnegoHandler.java:128)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RetryAndFollowUpInterceptor.followUpRequest(RetryAndFollo
wUpInterceptor.java:289)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RetryAndFollowUpInterceptor.intercept(RetryAndFollowUpInt
erceptor.java:157)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
at
io.prestosql.jdbc.$internal.client.SpnegoHandler.intercept(SpnegoHandler.java:115)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
at
io.prestosql.jdbc.$internal.client.OkHttpUtil.lambda$userAgent$0(OkHttpUtil.java:64)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:147)
at
io.prestosql.jdbc.$internal.okhttp3.internal.http.RealInterceptorChain.proceed(RealInterceptorChain.java:121)
at
io.prestosql.jdbc.$internal.okhttp3.RealCall.getResponseWithInterceptorChain(RealCall.java:200)
at
io.prestosql.jdbc.$internal.okhttp3.RealCall.execute(RealCall.java:77)
at
io.prestosql.jdbc.$internal.client.JsonResponse.execute(JsonResponse.java:131)
at
io.prestosql.jdbc.$internal.client.StatementClientV1.<init>(StatementClientV1.java:132)
at
io.prestosql.jdbc.$internal.client.StatementClientFactory.newStatementClient(StatementClientFactory.java:24)
at
io.prestosql.jdbc.QueryExecutor.startQuery(QueryExecutor.java:46)
at io.prestosql.jdbc.PrestoConnection.startQuery(PrestoConnection.java:714)
at
io.prestosql.jdbc.PrestoStatement.internalExecute(PrestoStatement.java:241)
... 3 more
Caused by: GSSException:
No valid credentials provided (Mechanism level: No valid credentials provided
(Mechanism level: Server not found in Kerberos database (7) - UNKNOWN_SERVER))
at
sun.security.jgss.spnego.SpNegoContext.initSecContext(SpNegoContext.java:454)
at
sun.security.jgss.GSSContextImpl.initSecContext(GSSContextImpl.java:248)
at sun.security.jgss.GSSContextImpl.initSecContext(GSSContextImpl.java:179)
at
io.prestosql.jdbc.$internal.client.SpnegoHandler.generateToken(SpnegoHandler.java:167)
... 23 more
Caused by: GSSException:
No valid credentials provided (Mechanism level: Server not found in Kerberos database
(7) - UNKNOWN_SERVER)
at
sun.security.jgss.krb5.Krb5Context.initSecContext(Krb5Context.java:772)
at
sun.security.jgss.GSSContextImpl.initSecContext(GSSContextImpl.java:248)
at
```



```
sun.security.jgss.GSSContextImpl.initSecContext(GSSContextImpl.java:179)
at
sun.security.jgss.spnego.SpNegoContext.GSS_initSecContext(SpNegoContext.java:882)
at
sun.security.jgss.spnego.SpNegoContext.initSecContext(SpNegoContext.java:317)
... 26 more
Caused by: KrbException:
Server not found in Kerberos database (7) - UNKNOWN_SERVER
at
sun.security.krb5.KrbTgsRep.<init>(KrbTgsRep.java:73)
at
sun.security.krb5.KrbTgsReq.getReply(KrbTgsReq.java:251)
at
sun.security.krb5.KrbTgsReq.sendAndGetCreds(KrbTgsReq.java:262)
at
sun.security.krb5.internal.CredentialsUtil.serviceCreds(CredentialsUtil.java:308)
at
sun.security.krb5.internal.CredentialsUtil.acquireServiceCreds(CredentialsUtil.java:126)
at
sun.security.krb5.Credentials.acquireServiceCreds(Credentials.java:466)
at sun.security.jgss.krb5.Krb5Context.initSecContext(Krb5Context.java:695)
... 30 more
Caused by: KrbException:
Identifier doesn't match expected value (906)
at
sun.security.krb5.internal.KDCRep.init(KDCRep.java:140)
at
sun.security.krb5.internal.TGSRep.init(TGSRep.java:65)
at
sun.security.krb5.internal.TGSRep.<init>(TGSRep.java:60)
at
sun.security.krb5.KrbTgsRep.<init>(KrbTgsRep.java:55)
... 36 more
```

报错2：

```
java.sql.SQLException:
Authentication failed: Authentication failed for token:
...
        at
com.facebook.presto.jdbc.PrestoStatement.internalExecute(PrestoStatement.java:271)
        at
com.facebook.presto.jdbc.PrestoStatement.execute(PrestoStatement.java:227)
        at
com.facebook.presto.jdbc.PrestoStatement.executeQuery(PrestoStatement.java:76)
        at
PrestoJDBCExample.main(PrestoJDBCExample.java:65)
Caused by:
com.facebook.presto.jdbc.internal.client.ClientException: Authentication failed:
Authentication failed for token:
...
        at
com.facebook.presto.jdbc.internal.client.StatementClientV1.requestFailedException(StatementClientV1.java:4
32)
        at
com.facebook.presto.jdbc.internal.client.StatementClientV1.<init>(StatementClientV1.java:132)
        at
com.facebook.presto.jdbc.internal.client.StatementClientFactory.newStatementClient(StatementClientFactory.
java:24)
        at
com.facebook.presto.jdbc.QueryExecutor.startQuery(QueryExecutor.java:46)
        at
com.facebook.presto.jdbc.PrestoConnection.startQuery(PrestoConnection.java:683)
        at
com.facebook.presto.jdbc.PrestoStatement.internalExecute(PrestoStatement.java:239)
        ... 3 more
```

回答

客户端拼接出的HTTP的principal与Kerberos数据库中的不一致（报错1）或获取的token无法链接Presto。

在集群上执行**cat /etc/hosts**，将Presto coordinator的IP和hostname加入当前节点的/etc/hosts中。

12 OpenTSDB 应用开发

12.1 概述

12.1.1 应用开发简介

OpenTSDB 简介

OpenTSDB是一个基于HBase的分布式、可伸缩的时间序列数据库。OpenTSDB的设计目标是用来采集大规模集群中的监控类信息，并可实现数据的秒级查询，解决海量监控类数据在普通数据库中查询存储的局限性。

OpenTSDB使用场景有如下几个特点：

- 采集指标在某一时间点具有唯一值，没有复杂的结构及关系。
- 监控的指标具有随着时间不断变化的特点。
- 具有HBase的高吞吐，良好的伸缩性等特点。

接口类型简介

OpenTSDB提供基于HTTP的应用程序编程接口，以实现与外部系统的集成。 几乎所有OpenTSDB功能都可通过API访问，例如查询时间序列数据，管理元数据和存储数据点。详情请参见：**http://opentsdb.net/docs/build/html/api_http/index.html**。

12.1.2 常用概念

基本概念

- **data point**：时间序列数据点，包括metric、timestamp、value和tag。表示某个metric在某个时间点的数值。
- **metric**：指标项。例如，在系统监控中的CPU使用率、内存、IO等指标。
- **timestamp**：UNIX时间戳（自Epoch以来的秒或毫秒），即value产生的时间。
- **value**：某个metric的值，是JSON格式的事件或直方图/摘要。
- **tag**：标签，是由Tagk和Tagv组成的键值对。用于描述该点所属的时间序列。

标签允许您从不同的源或相关实体中分离出类似的数据点，因此您可以轻松地单独或成组地绘制它们。标签的一个常见用法是使用生成数据点的机器名称以及机器所属的集群或池的名称来注释数据点。这使您可以轻松地制作显示每个服务器的服务状态的仪表盘，以及显示跨逻辑服务器池的聚合状态的仪表盘。

OpenTSDB 系统表简介

OpenTSDB是基于HBase存储时序数据的，在集群中开启OpenTSDB后，系统会在集群中创建4张HBase表。OpenTSDB系统表如所表12-1示。

说明

请不要人为去修改这4张HBase表，因为这可能会导致OpenTSDB不可用。

表 12-1 Opentsdb 系统表

表名	说明
tsdb	用于存储数据点，OpenTSDB的所有数据都存储在这个表中。
tsdb-meta	用于存储时间序列索引和元数据。
tsdb-tree	用于存储metric的结构信息。
tsdb-uid	用于存储UID映射，数据点中的每个metric，tag都会映射成UID，同时每个UID反向映射为metric，tag，这些映射关系存储在这个表中。

12.1.3 开发流程

开发流程中各阶段的说明如图12-1和表12-2所示。

图 12-1 Opentsdb 应用程序开发流程

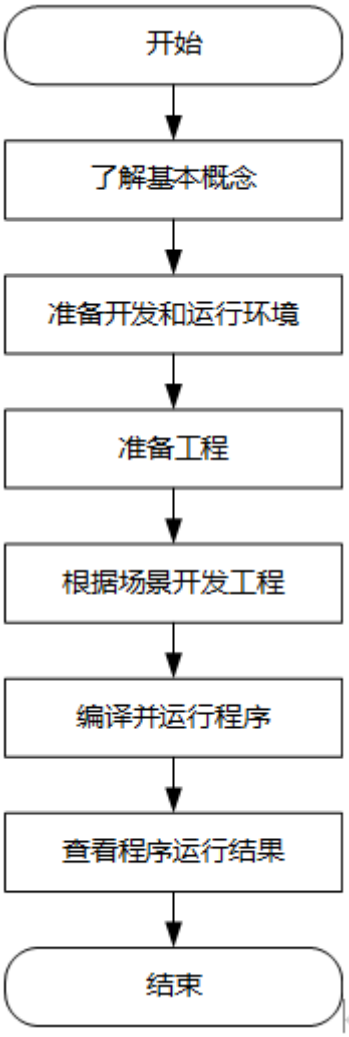


表 12-2 Opentsdb 应用开发流程说明

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OpenTSDB的基本概念，了解场景需求，设计表等。	**常用概念**
准备开发环境和运行环境	OpenTSDB的应用程序当前推荐使用Java语言进行开发。可使用Eclipse工具。OpenTSDB的运行环境即OpenTSDB客户端，请根据指导完成客户端的安装和配置。	**开发环境简介**

阶段	说明	参考文档
准备工程	OpenTSDB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OpenTSDB工程。	配置并导入样例工程
根据场景开发工程	提供了Java语言的样例工程，包含从创建metric、写入到查询流程的样例工程。	典型场景开发思路
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。	编译并运行程序
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查看导入数据的状态。	查看调测结果

12.2 环境准备

12.2.1 开发环境简介

在进行二次开发时，要准备的开发环境如表12-3所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行正常。

表 12-3 开发环境

准备	说明
操作系统	Windows系统，推荐Windows 7及以上版本。
安装JDK	开发环境的基本配置。版本要求：1.8及以上。
安装和配置Eclipse	用于开发OpenTSDB应用程序的工具。
网络	确保客户端与OpenTSDB服务主机在网络上互通。

12.2.2 准备环境

- 选择Windows开发环境下，安装Eclipse，安装JDK。

  请安装JDK1.8及以上版本。Eclipse使用支持JDK1.8及以上的版本，并安装JUnit插件。

说明

- 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
- 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
- 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。

- 准备一个应用程序运行测试的环境。

准备运行调测环境

步骤1 在弹性云服务器管理控制台，申请一个新的弹性云服务器，用于应用开发运行调测。

- 弹性云服务器的安全组需要和MRS集群Master节点的安全组相同。
- 弹性云服务器的VPC需要与MRS集群在同一个VPC中。
- 弹性云服务器的网卡需要与MRS集群在同一个网段中。

步骤2 申请弹性IP，并与新申请的ECS绑定，并配置安全组出入规则。

步骤3 下载客户端程序。

1. 登录**MRS Manager**系统。
2. 选择“服务管理 > 下载客户端 ”，下载“完整客户端”到“远端主机”上，即下载客户端程序到新申请的弹性云服务器上。

步骤4 登录存放下载的客户端的节点，再安装客户端。

1. 执行以下命令解压客户端包：

   cd /opt

   tar -xvf /opt/MRS_Services_Client.tar

2. 执行以下命令校验安装文件包：

   sha256sum -c /opt/MRS_Services_ClientConfig.tar.sha256

   ```
   MRS_Services_ClientConfig.tar:OK
   ```

3. 执行以下命令解压安装文件包：

   tar -xvf /opt/MRS_Services_ClientConfig.tar

4. 执行如下命令安装客户端到指定目录（绝对路径），例如“/opt/client”。目录会自动创建。

   cd /opt/MRS_Services_ClientConfig

   sh install.sh /opt/client

   ```
   Components client installation is complete.
   ```

----结束

12.2.3 准备开发用户

开发用户用于运行样例工程。用户需要有HBase权限，才能运行OpenTSDB样例工程。

前提条件

MRS服务集群开启了Kerberos认证，没有开启Kerberos认证的集群忽略该步骤。

操作步骤

步骤1 登录**MRS Manager**，在MRS Manager界面选择“系统设置 > 角色管理 > 添加角色”，如**图 1 添加角色**所示。

图 12-2 添加角色

1. 填写角色的名称，例如*opentsdbrole*。
2. 编辑角色，在“权限”的表格中选择“HBase> HBase Scope > global”，勾选“Read”、“Write”和“Execute”，单击“确定”保存，如**图12-3**所示。

图 12-3 编辑角色

步骤2 单击“系统设置 > 用户组管理 > 添加用户组”，为样例工程创建一个用户组，例如*opentsdgroup*。

步骤3 单击“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。

步骤4 填写用户名，例如*opentsdbuser*，用户类型为“人机”用户，加入到用户组**opentsdb**，**hbase**，**opentsdbgroup**和**supergroup**，设置其“主组”为**opentsdbgroup**，并绑定角色**opentsdbrole**取得权限，单击“确定”，如**图12-4**所示。

图 12-4 添加用户

步骤5 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名中选择**opentsdbuser**，修改密码后，在右侧“操作”列中选择“更多 >下载认证凭据”下载认证凭据，保存后解压得到用户的user.keytab文件与krb5.conf文件。用于在样例工程中进行安全认证，如图12-5所示。

图 12-5 下载认证凭据

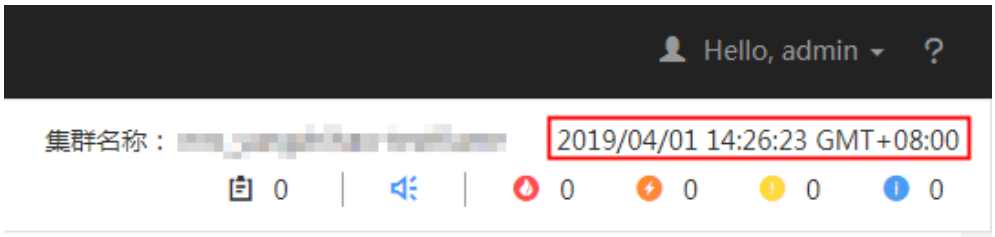
----结束

12.2.4 配置并导入样例工程

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。MRS集群的时间可通过MRS Manager页面右上角查看，如图12-6所示。

图 12-6 MRS 集群的时间



操作步骤

步骤1 在**样例工程获取地址** 获取OpenTSDB示例工程。

步骤2 在OpenTSDB示例工程根目录，执行**mvn install**编译。

步骤3 在Opentsdb示例工程根目录，执行**mvn eclipse:eclipse**创建Eclipse工程。

步骤4 在应用开发环境中，导入样例工程到Eclipse开发环境。

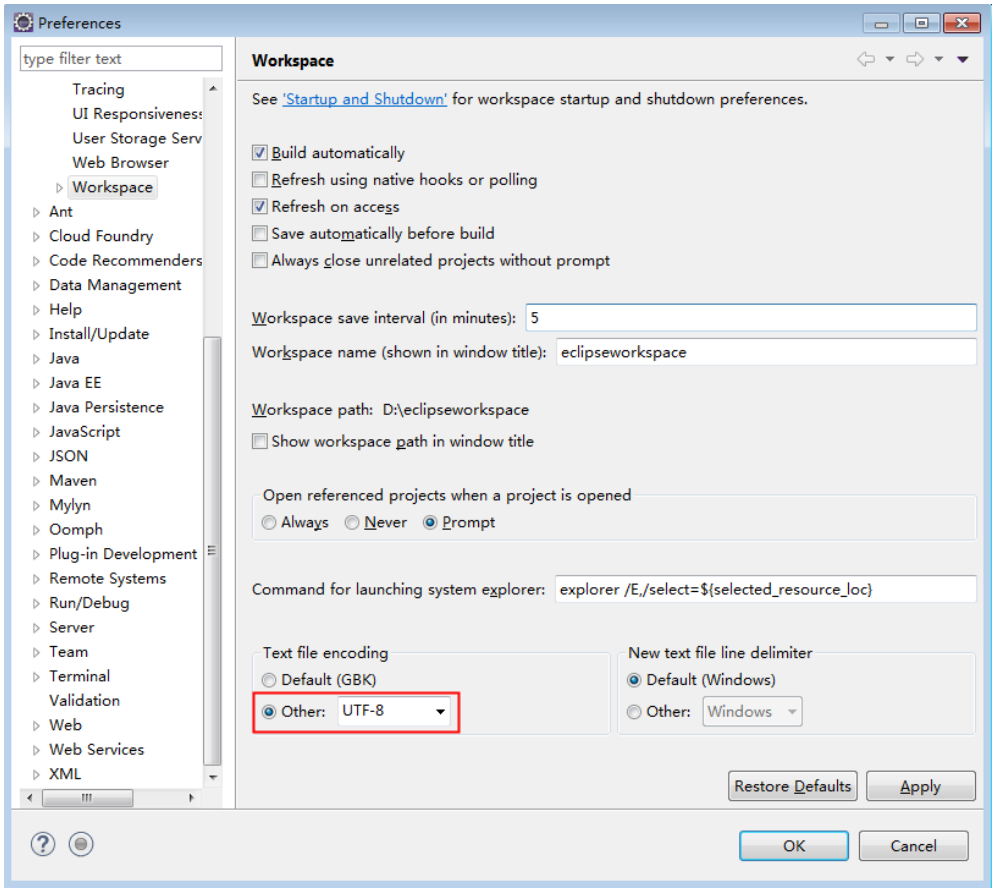
1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。
   显示“浏览文件夹”对话框。

2. 选择样例工程文件夹，单击“Finish”。

步骤5 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。
   弹出“Preferences”窗口。
2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如**图 2 设置Eclipse的编码格式**所示。
   设置Eclipse的编码格式

图 12-7 设置 Eclipse 的编码格式



----结束

12.3 开发程序

12.3.1 典型场景开发思路

通过典型场景，您可以快速学习和掌握OpenTSDB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，用于记录和查询城市的气象信息，记录数据如下表**表12-4**，**表12-5**和**表12-6**所示。

表 12-4 原始数据

城市	区域	时间	温度	湿度
Shenzhen	Longgang	2017/7/1 00:00:00	28	54
Shenzhen	Longgang	2017/7/1 01:00:00	27	53
Shenzhen	Longgang	2017/7/1 02:00:00	27	52
Shenzhen	Longgang	2017/7/1 03:00:00	27	51
Shenzhen	Longgang	2017/7/1 04:00:00	27	50
Shenzhen	Longgang	2017/7/1 05:00:00	27	49
Shenzhen	Longgang	2017/7/1 06:00:00	27	48
Shenzhen	Longgang	2017/7/1 07:00:00	27	46
Shenzhen	Longgang	2017/7/1 08:00:00	29	46
Shenzhen	Longgang	2017/7/1 09:00:00	30	48
Shenzhen	Longgang	2017/7/1 10:00:00	32	48
Shenzhen	Longgang	2017/7/1 11:00:00	32	49
Shenzhen	Longgang	2017/7/1 12:00:00	33	49
Shenzhen	Longgang	2017/7/1 13:00:00	33	50
Shenzhen	Longgang	2017/7/1 14:00:00	32	50
Shenzhen	Longgang	2017/7/1 15:00:00	32	50

城市	区域	时间	温度	湿度
Shenzhen	Longgang	2017/7/1 16:00:00	31	51
Shenzhen	Longgang	2017/7/1 17:00:00	30	51
Shenzhen	Longgang	2017/7/1 18:00:00	30	51
Shenzhen	Longgang	2017/7/1 19:00:00	29	51
Shenzhen	Longgang	2017/7/1 20:00:00	29	52
Shenzhen	Longgang	2017/7/1 21:00:00	29	53
Shenzhen	Longgang	2017/7/1 22:00:00	28	54
Shenzhen	Longgang	2017/7/1 23:00:00	28	54

该场景里记录了深圳市龙岗区在2017年7月1日零时的温度和湿度数据，这里通过OpenTSDB的方式建模实质上是两组数据点。

表 12-5 指标数据点 1

指标项 (metric)	城市(city)	区域 (region)	Unix timestamp	指标数值 (value)
city.temp	Shenzhen	Longgang	1498838400	28
city.temp	Shenzhen	Longgang	1498842000	27
city.temp	Shenzhen	Longgang	1498845600	27
city.temp	Shenzhen	Longgang	1498849200	27
city.temp	Shenzhen	Longgang	1498852800	27
city.temp	Shenzhen	Longgang	1498856400	27
city.temp	Shenzhen	Longgang	1498860000	27
city.temp	Shenzhen	Longgang	1498863600	27
city.temp	Shenzhen	Longgang	1498867200	29
city.temp	Shenzhen	Longgang	1498870800	30
city.temp	Shenzhen	Longgang	1498874400	32

指标项 (metric)	城市(city)	区域 (region)	Unix timestamp	指标数值 (value)
city.temp	Shenzhen	Longgang	1498878000	32
city.temp	Shenzhen	Longgang	1498881600	33
city.temp	Shenzhen	Longgang	1498885200	33
city.temp	Shenzhen	Longgang	1498888800	32
city.temp	Shenzhen	Longgang	1498892400	32
city.temp	Shenzhen	Longgang	1498896000	31
city.temp	Shenzhen	Longgang	1498899600	30
city.temp	Shenzhen	Longgang	1498903200	30
city.temp	Shenzhen	Longgang	1498906800	29
city.temp	Shenzhen	Longgang	1498910400	29
city.temp	Shenzhen	Longgang	1498914000	29
city.temp	Shenzhen	Longgang	1498917600	28
city.temp	Shenzhen	Longgang	1498921200	28

表 12-6 指标数据点 2

指标项 (metric)	城市(city)	区域 (region)	Unix timestamp	指标数值 (value)
city.hum	Shenzhen	Longgang	1498838400	54
city.hum	Shenzhen	Longgang	1498842000	53
city.hum	Shenzhen	Longgang	1498845600	52
city.hum	Shenzhen	Longgang	1498849200	51
city.hum	Shenzhen	Longgang	1498852800	50
city.hum	Shenzhen	Longgang	1498856400	49
city.hum	Shenzhen	Longgang	1498860000	48
city.hum	Shenzhen	Longgang	1498863600	46
city.hum	Shenzhen	Longgang	1498867200	46
city.hum	Shenzhen	Longgang	1498870800	48
city.hum	Shenzhen	Longgang	1498874400	48
city.hum	Shenzhen	Longgang	1498878000	49

指标项(metric)	城市(city)	区域(region)	Unix timestamp	指标数值(value)
city.hum	Shenzhen	Longgang	1498881600	49
city.hum	Shenzhen	Longgang	1498885200	50
city.hum	Shenzhen	Longgang	1498888800	50
city.hum	Shenzhen	Longgang	1498892400	50
city.hum	Shenzhen	Longgang	1498896000	51
city.hum	Shenzhen	Longgang	1498899600	51
city.hum	Shenzhen	Longgang	1498903200	51
city.hum	Shenzhen	Longgang	1498906800	51
city.hum	Shenzhen	Longgang	1498910400	52
city.hum	Shenzhen	Longgang	1498914000	53
city.hum	Shenzhen	Longgang	1498917600	54
city.hum	Shenzhen	Longgang	1498921200	54

其中这两组指标数据点都有2个标签：

- 标签(tag)：城市city、区域region
- 标签值(tag value)：ShenZhen、Longgang

用户可以执行以下数据操作：

- 获取每天的监控数据，通过OpenTSDB的put接口将两个组数据点写入数据库中。
- 对已有的数据使用OpenTSDB的query接口进行数据查询和分析。

功能分解

根据上述的业务场景进行功能开发，需要开发的功能如**表12-7**所示。

表 12-7 在 OpenTSDB 中开发的功能

序号	步骤	代码实现
1	根据典型场景说明建立了数据模型	请参见**配置参数**
2	写入指标数据	请参见**写入数据**
3	根据指标项进行数据查询	请参见**查询数据**
4	删除指定范围的数据	请参见**删除数据**

12.3.2 配置参数

步骤1 执行样例代码前，必须在样例代码工程“resources”目录下的opentsdb.properties中修改如下参数：

```
tsd_hostname = node-ana-coreYQnTx
tsd_port = 4242
tsd_protocol = https
```

- tsd_hostname：修改为连接OpenTSDB服务的TSD实例的主机名或IP。

  说明

  - 若当前运行环境与OpenTSDB安装环境处于同一个VPC网络中，使用连接的TSD实例IP或主机名均可。
  - 若当前运行环境与OpenTSDB安装环境位于不同VPC中，仅可使用主机名进行访问。同时需在连接的TSD实例上绑定EIP，并把该EIP及该TSD实例的主机名配置到hosts中，linux环境需修改文件位置为“/etc/hosts”，Windows环境需修改的文件位置为“C:\Windows\System32\drivers\etc\hosts”。

  例如，访问的TSD实例主机名为node-ana-corexxqm，其对应绑定的EIP为100.94.10.10，则需录入如下配置：

  ```
  100.94.10.10 node-ana-coreYQnTx
  ```

- tsd_port：TSD端口，默认使用4242。
- tsd_protocol：请求协议，默认使用https。

步骤2 （可选）如果不使用样例工程，可在自己工程的pom.xml文件中添加依赖如下：

- guava

  ```
  <!-- https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient -->
  <dependency>
      <groupId>com.google.guava</groupId>
      <artifactId>guava</artifactId>
      <version>18.0</version>
  </dependency>
  ```

- gson

  ```
  <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
  <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.2.4</version>
  </dependency>
  ```

- httpcore

  ```
  <!-- https://mvnrepository.com/artifact/org.apache.httpcomponents/httpcore -->
  <dependency>
      <groupId>org.apache.httpcomponents</groupId>
      <artifactId>httpcore</artifactId>
      <version>4.4.4</version>
  </dependency>
  ```

- httpclient

  ```
  <!-- https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient -->
  <dependency>
      <groupId>org.apache.httpcomponents</groupId>
      <artifactId>httpclient</artifactId>
      <version>4.5.2</version>
  </dependency>
  ```

步骤3 每个HTTP请求都应该设置超时间，设置超时时间的方法如下：

```
public static void addTimeout(HttpRequestBase req) {
  RequestConfig requestConfig = RequestConfig.custom().setConnectTimeout(5000)
      .setConnectionRequestTimeout(10000).setSocketTimeout(60000).build();
```



```
  req.setConfig(requestConfig);
}
```

----结束

12.3.3 写入数据

功能简介

使用OpenTSDB的接口(/api/put)写入数据。

函数genWeatherData()模拟生成的气象数据，函数putData()发送气象数据到OpenTSDB服务端。

样例代码

以下代码片段在com.huawei.bigdata.opentsdb.examples包的"OpentsdbExample"类的putData方法中。

```
private void putData(String tmpURL) {
  PUT_URL = BASE_URL + tmpURL;
  LOG.info("start to put data in opentsdb, the url is " + PUT_URL);
  try (CloseableHttpClient httpClient = HttpClients.createDefault()) {
    HttpPost httpPost = new HttpPost(PUT_URL);//请求需要设置超时时间
    addTimeout(httpPost);
    String weatherData = genWeatherData();
    StringEntity entity = new StringEntity(weatherData, "ISO-8859-1");
    entity.setContentType("application/json");
    httpPost.setEntity(entity);
    HttpResponse response = httpClient.execute(httpPost);
    int statusCode = response.getStatusLine().getStatusCode();
    LOG.info("Status Code : " + statusCode);
    if (statusCode != HttpStatus.SC_NO_CONTENT) {
      LOG.info("Request failed! " + response.getStatusLine());
    }
    LOG.info("put data to opentsdb successfully.");
  } catch (IOException e) {
    LOG.error("Failed to put data.", e);
  }
}

static class DataPoint {
  public String metric;
  public Long timestamp;
  public Double value;
  public Map<String, String> tags;
  public DataPoint(String metric, Long timestamp, Double value, Map<String, String> tags) {
    this.metric = metric;
    this.timestamp = timestamp;
    this.value = value;
    this.tags = tags;
  }
}

private String genWeatherData() {
  List<DataPoint> dataPoints = new ArrayList<DataPoint>();
  Map<String, String> tags = ImmutableMap.of("city", "Shenzhen", "region", "Longgang");

  // Data of air temperature
  dataPoints.add(new DataPoint("city.temp", 1498838400L, 28.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498842000L, 27.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498845600L, 27.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498849200L, 27.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498852800L, 27.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498856400L, 27.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498860000L, 27.0, tags));
```

```
  dataPoints.add(new DataPoint("city.temp", 1498863600L, 27.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498867200L, 29.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498870800L, 30.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498874400L, 32.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498878000L, 32.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498881600L, 33.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498885200L, 33.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498888800L, 32.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498892400L, 32.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498896000L, 31.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498899600L, 30.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498903200L, 30.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498906800L, 29.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498910400L, 29.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498914000L, 29.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498917600L, 28.0, tags));
  dataPoints.add(new DataPoint("city.temp", 1498921200L, 28.0, tags));

  // Data of humidity
  dataPoints.add(new DataPoint("city.hum", 1498838400L, 54.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498842000L, 53.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498845600L, 52.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498849200L, 51.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498852800L, 50.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498856400L, 49.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498860000L, 48.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498863600L, 46.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498867200L, 46.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498870800L, 48.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498874400L, 48.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498878000L, 49.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498881600L, 49.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498885200L, 50.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498888800L, 50.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498892400L, 50.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498896000L, 51.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498899600L, 51.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498903200L, 51.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498906800L, 51.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498910400L, 52.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498914000L, 53.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498917600L, 54.0, tags));
  dataPoints.add(new DataPoint("city.hum", 1498921200L, 54.0, tags));

  Gson gson = new Gson();
  return gson.toJson(dataPoints);
}
```

说明

PUT_URL中加入了sync参数，表示必须等到数据写入HBase后才可以返回，强烈建议使用此参数；如果不使用sync，表示采用异步写入HBase的方式，可能存在丢失数据的风险。具体信息请参考**OpenTSDB接口**。

12.3.4 查询数据

功能简介

使用OpenTSDB的查询接口(/api/query)读取数据。

函数genQueryReq()生成查询请求，函数queryData()把查询请求发送到OpenTSDB服务端。

样例代码

以下代码片段在com.huawei.bigdata.opentsdb.examples包的"OpentsdbExample"类的queryData方法中。

```
private void queryData(String dataPoint) {
  QUERY_URL = BASE_URL + dataPoint;
  LOG.info("start to query data in opentsdb, the url is " + QUERY_URL);
  try (CloseableHttpClient httpClient = HttpClients.createDefault()) {
    HttpPost httpPost = new HttpPost(QUERY_URL);// 请求需要设置超时时间
    addTimeout(httpPost);
    String queryRequest = genQueryReq();
    StringEntity entity = new StringEntity(queryRequest, "utf-8");
    entity.setContentType("application/json");
    httpPost.setEntity(entity);
    HttpResponse response = httpClient.execute(httpPost);
    int statusCode = response.getStatusLine().getStatusCode();
    LOG.info("Status Code : " + statusCode);
    if (statusCode != HttpStatus.SC_OK) {
      LOG.info("Request failed! " + response.getStatusLine());
    }
    String body = EntityUtils.toString(response.getEntity(), "utf-8");
    LOG.info("Response content : " + body);
    LOG.info("query data to opentsdb successfully.");
  } catch (IOException e) {
    LOG.error("Failed to query data.", e);
  }
}

static class Query {
  public Long start;
  public Long end;
  public boolean delete = false;
  public List<SubQuery> queries;
}

static class SubQuery {
  public String metric;
  public String aggregator;
  public SubQuery(String metric, String aggregator) {
    this.metric = metric;
    this.aggregator = aggregator;
  }
}

String genQueryReq() {
  Query query = new Query();
  query.start = 1498838400L;
  query.end = 1498921200L;
  query.queries = ImmutableList.of(new SubQuery("city.temp", "sum"), new SubQuery("city.hum", "sum"));
  Gson gson = new Gson();
  return gson.toJson(query);
}
```

12.3.5 删除数据

功能简介

在OpenTSDB的查询接口中增加delete参数，并且设置delete参数为true。函数genQueryReq()生成删除请求，函数deleteData()把删除请求发送到OpenTSDB服务端。

样例代码

以下代码片段在com.huawei.bigdata.opentsdb.examples包的"OpentsdbExample"类的deleteData方法中。

```
public void deleteData(String dataPoint) {
  QUERY_URL = BASE_URL + dataPoint;
  try (CloseableHttpClient httpClient = HttpClients.createDefault()) {
    HttpPost httpPost = new HttpPost(QUERY_URL);addTimeout(httpPost);
    String deleteRequest = genDeleteReq();
    StringEntity entity = new StringEntity(deleteRequest, "utf-8");
    entity.setContentType("application/json");
    httpPost.setEntity(entity);
    HttpResponse response = httpClient.execute(httpPost);
    int statusCode = response.getStatusLine().getStatusCode();
    LOG.info("Status Code : " + statusCode);
    if (statusCode != HttpStatus.SC_OK) {
      LOG.info("Request failed! " + response.getStatusLine());
    }
  } catch (IOException e) {
    LOG.error("Failed to delete data.", e);
  }
}

static class Query {
  public Long start;
  public Long end;
  public boolean delete = false;
  public List<SubQuery> queries;
}

static class SubQuery {
  public String metric;
  public String aggregator;
  public SubQuery(String metric, String aggregator) {
    this.metric = metric;
    this.aggregator = aggregator;
  }
}

String genQueryReq() {
  Query query = new Query();
  query.start = 1498838400L;
  query.end = 1498921200L;
  query.queries = ImmutableList.of(new SubQuery("city.temp", "sum"), new SubQuery("city.hum", "sum"));
  Gson gson = new Gson();
  return gson.toJson(query);
}

String genDeleteReq() {
  Query query = new Query();
  query.start = 1498838400L;
  query.end = 1498921200L;
  query.queries = ImmutableList.of(new SubQuery("city.temp", "sum"), new SubQuery("city.hum", "sum"));
  query.delete = true;

  Gson gson = new Gson();
  return gson.toJson(query);
}
```

12.4 调测程序

12.4.1 在 Windows 中调测程序

12.4.1.1 编译并运行程序

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。

操作步骤

步骤1 在Windows上配置集群的IP与主机名映射关系。登录集群后台，执行命令**cat /etc/hosts**后，把hosts文件中的IP与主机名映射关系拷贝到“C:\Windows\System32\drivers\etc\hosts”中。

```
xx.xx.xx.xx node-ana-corejnWt
xx.xx.xx.xx node-ana-coreddlI
```

说明

使用Windows访问MRS集群来操作OpenTSDB，有如下两种方式：

- 申请一台Windows的ECS访问MRS集群操作OpenTSDB。安装开发环境后运行样例代码。申请ECS访问MRS集群的步骤如下：
  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”、“集群控制台地址”，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“弹性云服务器 > 快速入门 > 购买并登录Windows弹性云服务器”
- 使用本机访问MRS集群操作OpenTSDB。为MRS集群中OpenTSDB服务所要访问的TSD实例绑定弹性公网IP，在本机(即Windows机器)上配置集群的IP与主机名映射关系时，把IP替换为主机名对应的弹性公网IP，运行样例代码。绑定弹性公网IP步骤如下：
  1. 在虚拟私有云管理控制台，申请一个弹性IP地址，并与弹性云服务器绑定。

     具体请参见“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. 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。请参见“虚拟私有云 > 用户指南 > 安全性 > 安全组 > 添加安全组规则”。

步骤2 将集群中master节点的“/opt/Bigdata/jdk1.8.0_212/jre/lib/security/cacerts”文件替换到windows的jdk对应的路径下，比如：“C:\Program Files\Java\jdk1.8.0_73\jre\lib\security”。

步骤3 修改配置。修改样例工程中“resources”目录下的“opentsdb.properties”文件，配置OpenTSDB相关属性。

```
tsd_hostname = node-ana-corejnWt
tsd_port = 4242
tsd_protocol = https
```

- tsd_hostname： 访问OpenTSDB时所连接的TSD实例主机名称。
- tsd_port：访问OpenTSDB的端口，默认端口为4242。
- tsd_protocol：访问OpenTSDB的请求协议。默认为https。

步骤4 运行样例。

开发完成后，在开发环境中（例如Eclipse中），右击“OpentsdbExample.java”，单击“Run as > Java Application”运行对应的应用程序工程。

----结束

12.4.1.2 查看调测结果

操作场景

OpenTSDB样例程序运行完后，可直接通过运行结果查看应用程序运行情况，也可以通过运行日志获取应用运行情况。

操作步骤

- 运行结果会有如下成功信息：

```
2019-06-27 14:05:20,713 INFO  [main] examples.OpentsdbExample: start to put data in opentsdb, the url is https://node-ana-corejnWt:4242/api/put/?sync&sync_timeout=60000
2019-06-27 14:05:23,680 INFO  [main] examples.OpentsdbExample: Status Code : 204
2019-06-27 14:05:23,680 INFO  [main] examples.OpentsdbExample: put data to opentsdb successfully.
2019-06-27 14:05:23,681 INFO  [main] examples.OpentsdbExample: start to query data in opentsdb, the url is https://node-ana-corejnWt:4242/api/query
2019-06-27 14:05:23,895 INFO  [main] examples.OpentsdbExample: Status Code : 200
2019-06-27 14:05:23,897 INFO  [main] examples.OpentsdbExample: Response content :
[{"metric":"city.temp","tags":{"region":"Longgang","city":"Shenzhen"},"aggregateTags":[],"dps":
{"1498838400":28.0,"1498842000":27.0,"1498845600":27.0,"1498849200":27.0,"1498852800":27.0,"1498856400":27.0,"1498860000":27.0,"1498863600":27.0,"1498867200":29.0,"1498870800":30.0,"1498874400":32.0,"1498878000":32.0,"1498881600":33.0,"1498885200":33.0,"1498888800":32.0,"1498892400":32.0,"1498896000":31.0,"1498899600":30.0,"1498903200":30.0,"1498906800":29.0,"1498910400":29.0,"1498914000":29.0,"1498917600":28.0,"1498921200":28.0}},{"metric":"city.hum","tags":
{"region":"Longgang","city":"Shenzhen"},"aggregateTags":[],"dps":
{"1498838400":54.0,"1498842000":53.0,"1498845600":52.0,"1498849200":51.0,"1498852800":50.0,"1498856400":49.0,"1498860000":48.0,"1498863600":46.0,"1498867200":46.0,"1498870800":48.0,"1498874400":48.0,"1498878000":49.0,"1498881600":49.0,"1498885200":50.0,"1498888800":50.0,"1498892400":50.0,"1498896000":51.0,"1498899600":51.0,"1498903200":51.0,"1498906800":51.0,"1498910400":52.0,"1498914000":53.0,"1498917600":54.0,"1498921200":54.0}}]
2019-06-27 14:05:23,897 INFO  [main] examples.OpentsdbExample: query data to opentsdb successfully.
2019-06-27 14:05:23,897 INFO  [main] examples.OpentsdbExample: start to delete data in opentsdb, the url is https://node-ana-corejnWt:4242/api/query
2019-06-27 14:05:24,112 INFO  [main] examples.OpentsdbExample: Status Code : 200
2019-06-27 14:05:24,112 INFO  [main] examples.OpentsdbExample: delete data to opentsdb successfully.
```

- 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，FATAL）来显示更详细的信息。可以通过修改log4j.properties文件来实现，如：

```
# Define some default values that can be overridden by system properties
opentsdb.root.logger=INFO,console
# Define the root logger to the system property "opentsdb.root.logger".
log4j.rootLogger=${opentsdb.root.logger}
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{ISO8601} %-5p [%t] %c{2}: %m%n
```

12.4.2 在 Linux 中调测程序

12.4.2.1 编译并运行程序

OpenTSDB应用程序支持在安装OpenTSDB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

- Linux环境已安装JDK，版本号需要和Eclipse导出Jar包使用的JDK版本一致。
- 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 登录linux环境，创建运行OpenTSDB样例的工作目录，比如“/opt/opentsdb-example”,配置文件存放目录，比如“/opt/opentsdb-example/conf”,并编辑配置文件“/opt/opentsdb-example/conf/opentsdb.properties”使其对应于实际环境中的信息。

```
mkdir -p /opt/opentsdb-example/conf
[root@node-master1rLqO ~]# cat /opt/opentsdb-example/conf/opentsdb.properties
tsd_hostname = node-ana-corejnWt
tsd_port = 4242
tsd_protocol = https
```

步骤2 执行**mvn package**生成jar包，在工程目录target目录下获取，比如:opentsdb-examples-mrs-xxx-jar-with-dependencies.jar，其中mrs-xxx表示MRS的版本号，将获取的包上传到“/opt/opentsdb-example”目录下。

步骤3 执行Jar包。

加载环境变量。

```
source /opt/client/bigdata_env
```

认证集群用户（未启用kerberos的集群可跳过此步骤）。

人机用户：*kinit kerberos用户*

机机用户： *kinit -kt 认证文件路径 kerberos用户*

运行opentsdb样例程序。

```
java -cp /opt/opentsdb-example/conf:/opt/opentsdb-example/opentsdb-examples-mrs-xxx-jar-with-dependencies.jar com.huawei.bigdata.opentsdb.examples.OpentsdbExample
```

----结束

12.4.2.2 查看调测结果

操作场景

OpenTSDB样例程序运行完后，可直接通过运行结果查看应用程序运行情况，也可以通过运行日志获取应用运行情况。

操作步骤

- 运行结果会有如下成功信息：

```
2019-06-27 14:05:20,713 INFO  [main] examples.OpentsdbExample: start to put data in opentsdb, the url is https://node-ana-corejnWt:4242/api/put/?sync&sync_timeout=60000
2019-06-27 14:05:23,680 INFO  [main] examples.OpentsdbExample: Status Code : 204
```



```
2019-06-27 14:05:23,680 INFO  [main] examples.OpentsdbExample: put data to opentsdb successfully.
2019-06-27 14:05:23,681 INFO  [main] examples.OpentsdbExample: start to query data in opentsdb, the url is https://node-ana-corejnWt:4242/api/query
2019-06-27 14:05:23,895 INFO  [main] examples.OpentsdbExample: Status Code : 200
2019-06-27 14:05:23,897 INFO  [main] examples.OpentsdbExample: Response content :
[{"metric":"city.temp","tags":{"region":"Longgang","city":"Shenzhen"},"aggregateTags":[],"dps":
{"1498838400":28.0,"1498842000":27.0,"1498845600":27.0,"1498849200":27.0,"1498852800":27.0,"1498856400":27.0,"1498860000":27.0,"1498863600":27.0,"1498867200":29.0,"1498870800":30.0,"1498874400":32.0,"1498878000":32.0,"1498881600":33.0,"1498885200":33.0,"1498888800":32.0,"1498892400":32.0,"1498896000":31.0,"1498899600":30.0,"1498903200":30.0,"1498906800":29.0,"1498910400":29.0,"1498914000":29.0,"1498917600":28.0,"1498921200":28.0}},{"metric":"city.hum","tags":
{"region":"Longgang","city":"Shenzhen"},"aggregateTags":[],"dps":
{"1498838400":54.0,"1498842000":53.0,"1498845600":52.0,"1498849200":51.0,"1498852800":50.0,"1498856400":49.0,"1498860000":48.0,"1498863600":46.0,"1498867200":46.0,"1498870800":48.0,"1498874400":48.0,"1498878000":49.0,"1498881600":49.0,"1498885200":50.0,"1498888800":50.0,"1498892400":50.0,"1498896000":51.0,"1498899600":51.0,"1498903200":51.0,"1498906800":51.0,"1498910400":52.0,"1498914000":53.0,"1498917600":54.0,"1498921200":54.0}}]
2019-06-27 14:05:23,897 INFO  [main] examples.OpentsdbExample: query data to opentsdb successfully.
2019-06-27 14:05:23,897 INFO  [main] examples.OpentsdbExample: start to delete data in opentsdb, the url is https://node-ana-corejnWt:4242/api/query
2019-06-27 14:05:24,112 INFO  [main] examples.OpentsdbExample: Status Code : 200
2019-06-27 14:05:24,112 INFO  [main] examples.OpentsdbExample: delete data to opentsdb successfully.
```

- 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，FATAL）来显示更详细的信息。可以通过修改log4j.properties文件来实现，如：

```
# Define some default values that can be overridden by system properties
opentsdb.root.logger=INFO,console
# Define the root logger to the system property "opentsdb.root.logger".
log4j.rootLogger=${opentsdb.root.logger}
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{ISO8601} %-5p [%t] %c{2}: %m%n
```

12.5 OpenTSDB 接口

12.5.1 CLI Tools

OpenTSDB提供了客户端工具，可以直接调用相关命令对OpenTSDB进行操作。客户端工具同开源社区版本保持一致，请参见**https://opentsdb.net/docs/build/html/user_guide/cli/index.html**。

客户端工具使用方法：

步骤1 登录任意一个Master节点。

步骤2 初始化环境变量。

source /opt/client/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。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 opentsdbuser**

步骤4 执行**tsdb**命令。例如执行**tsdb**可以打印出当前opentsdb所支持的所有命令，如，fsck, import, mkmetric, query, tsd, scan, search, uid, version。

```
tsdb: error: unknown command ''
usage: tsdb <command> [args]
Valid commands: fsck, import, mkmetric, query, tsd, scan, search, uid, version
```

----结束

创建 OpenTSDB 指标

创建存入到OpenTSDB中的指标名称，可执行**tsdb mkmetric sys.cpu.user**命令创建sys.cpu.user。

```
Start run net.opentsdb.tools.UidManager, args: assign metrics sys.cpu.user
metrics sys.cpu.user: [0, 0, 6]
```

查询 OpenTSDB 指标

tsdb命令可以获取到当前opentsdb存入的指标，可执行**tsdb uid metrics sys.cpu.user**命令。

```
Start run net.opentsdb.tools.UidManager, args: metrics sys.cpu.user
metrics sys.cpu.user: [0, 0, 6]
```

若想获取更多信息，可通过执行命令**tsdb uid**。

```
Start run net.opentsdb.tools.UidManager, args:
Not enough arguments
Usage: uid <subcommand> args
Sub commands:
  grep [kind] <RE>: Finds matching IDs.
  assign <kind> <name> [names]: Assign an ID for the given name(s).
  rename <kind> <name> <newname>: Renames this UID.
  delete <kind> <name>: Deletes this UID.
  fsck: [fix] [delete_unknown] Checks the consistency of UIDs.
        fix            - Fix errors. By default errors are logged.
        delete_unknown - Remove columns with unknown qualifiers.
                         The "fix" flag must be supplied as well.

  [kind] <name>: Lookup the ID of this name.
  [kind] <ID>: Lookup the name of this ID.
  metasync: Generates missing TSUID and UID meta entries, updates
            created timestamps
  metapurge: Removes meta data entries from the UID table
  treesync: Process all timeseries meta objects through tree rules
  treepurge <id> [definition]: Purge a tree and/or the branches
            from storage. Provide an integer Tree ID and optionally
            add "true" to delete the tree definition

Example values for [kind]: metrics, tagk (tag name), tagv (tag value).
  --config=PATH    Path to a configuration file (default: Searches for file see docs).
  --idwidth=N      Number of bytes on which the UniqueId is encoded.
  --ignore-case    Ignore case distinctions when matching a regexp.
  --table=TABLE    Name of the HBase table where to store the time series (default: tsdb).
  --uidtable=TABLE Name of the HBase table to use for Unique IDs (default: tsdb-uid).
  --verbose        Print more logging messages and not just errors.
  --zkbasedir=PATH Path under which is the znode for the -ROOT- region (default: /hbase).
  --zkquorum=SPEC  Specification of the ZooKeeper quorum to use (default: localhost).
  -i               Short for --ignore-case.
  -v               Short for --verbose.
```

向 OpenTSDB 指标中导入数据

tsdb命令可以使用“tsdb import”命令批量导入指标数据，可执行如下命令：

- 准备指标数据，如包含如下内容的importData.txt文件。

  ```
  sys.cpu.user 1356998400 41 host=web01 cpu=0
  sys.cpu.user 1356998401 42 host=web01 cpu=0
  ```



```
sys.cpu.user 1356998402 44 host=web01 cpu=0
sys.cpu.user 1356998403 47 host=web01 cpu=0
sys.cpu.user 1356998404 42 host=web01 cpu=0
sys.cpu.user 1356998405 42 host=web01 cpu=0
```

- 导入指标数据，可执行如下命令**tsdb import importData.txt**。

```
Start run net.opentsdb.tools.TextImporter, args: importData.txt
2019-06-26 15:45:22,091 INFO  [main] TextImporter: reading from file:importData.txt
2019-06-26 15:45:22,102 INFO  [main] TextImporter: Processed importData.txt in 11 ms, 6 data points (545.5 points/s)
2019-06-26 15:45:22,102 INFO  [main] TextImporter: Total: imported 6 data points in 0.012s (504.0 points/s)
```

扫描 OpenTSDB 的指标数据

tsdb命令可以使用“tsdb query”命令批量查询导入的指标数据，例如执行**tsdb query 0 1h-ago sum sys.cpu.user host=web01**命令。

```
Start run net.opentsdb.tools.CliQuery, args: 0 1h-ago sum sys.cpu.user host=web01
sys.cpu.user 1356998400000 41 {host=web01, cpu=0}
sys.cpu.user 1356998401000 42 {host=web01, cpu=0}
sys.cpu.user 1356998402000 44 {host=web01, cpu=0}
sys.cpu.user 1356998403000 47 {host=web01, cpu=0}
sys.cpu.user 1356998404000 42 {host=web01, cpu=0}
sys.cpu.user 1356998405000 42 {host=web01, cpu=0}
```

删除录入的 OpenTSDB 指标

tsdb命令可以使用“tsdb uid delete”命令删除录入的指标及值，例如执行**tsdb uid delete metrics sys.cpu.user**命令。

```
Start run net.opentsdb.tools.UidManager, args: delete metrics sys.cpu.user
```

12.5.2 HTTP API

OpenTSDB提供了基于HTTP或HTTPS的应用程序接口。请求方式是通过向资源对应的路径发送标准的HTTP请求，请求包含GET、POST方法。它的接口与开源OpenTSDB保持一致，请参见**https://opentsdb.net/docs/build/html/api_http/index.html**。

请求以及响应实体的类型为：application/JSON

请求以及响应实体的编码为：ISO-8859-1

说明

- HTTP协议本身有安全风险，HTTPS是安全协议，建议使用HTTPS连接方式。
- OpenTSDB基于HTTP提供了访问其的RESTful接口，而RESTful接口本身具有语言无关性的特点，凡是支持HTTP请求的语言都可以对接OpenTSDB。

使用 Java API 操作 OpenTSDB

OpenTSDB提供了基于HTTP或HTTPS的应用程序接口，可以使用Java API调用相关接口操作其数据，详情请参考开发程序章节。

使用 curl 命令操作 Opentsdb

- 写入数据。例如，录入一个指标名称为testdata，时间戳为1524900185，值为true，标签为key，value的指标数据。

```
curl -ki -X POST -d '{"metric":"testdata", "timestamp":1524900185, "value":"true", "tags":{"key":"value"}}' https://<tsd_ip>:4242/api/put?sync
```

<tsd_ip>表示所需写入数据的Opentsdb服务的TSD实例的IP地址。

```
HTTP/1.1 204 No Content
Content-Type: application/json; charset=UTF-8
Content-Length: 0
```

- 查询数据。例如，可查询指标testdata在过去三年中的汇总信息。

```
curl -ks https://<tsd_ip>:4242/api/query?start=3y-ago\&m=sum:testdata | python -m json.tool
```

  - <tsd_ip>：所需访问Opentsdb服务的TSD实例IP或主机名。
  - <start=3y-ago\&m=sum:testdata>：在请求中可能无法识别“&”符号，需对其进行转义。
  - <python -m json.tool>（可选）：把响应的请求转换为json格式。

```
[
  {
    "aggregateTags": [],
    "dps": {
      "1524900185": 1
    },
    "metric": "testdata",
    "tags": {
      "key": "value"
    }
  }
]
```

- 查询tsd状态信息。例如，可查询连接HBase的客户端信息。

```
curl -ks https://<tsd_ip>:4242/api/stats/region_clients | python -m json.tool
```

<tsd_ip>：所需访问Opentsdb服务的TSD实例IP地址。

```
[
  {
    "dead": false,
    "endpoint": "/192.168.2.187:16020",
    "inflightBreached": 0,
    "pendingBatchedRPCs": 0,
    "pendingBreached": 0,
    "pendingRPCs": 0,
    "rpcResponsesTimedout": 0,
    "rpcResponsesUnknown": 0,
    "rpcid": 78,
    "rpcsInFlight": 0,
    "rpcsSent": 79,
    "rpcsTimedout": 0,
    "writesBlocked": 0
  }
]
```

13 Flink 应用开发

13.1 概述

13.1.1 应用开发简介

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。

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 DataStream API提供Scala和Java两种语言的开发方式，如表13-1所示。

表 13-1 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。

有关Flink的详细信息，请参见：**https://flink.apache.org/**

13.1.2 常用概念

- **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。

- **Topology**

  一个Topology代表用户的一个执行任务。一个Topology由输入（如kafka soruce）、输出（如kafka sink）和多个Data Transformation组成。

- **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。

13.1.3 开发流程

开发流程中各阶段的说明如图13-1和表13-2所示。

图 13-1 Flink 应用程序开发流程

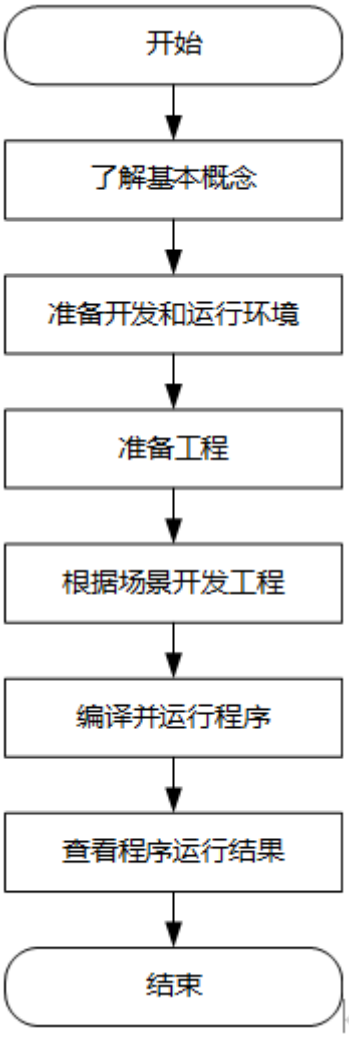


表 13-2 Flink 应用开发流程说明

阶段	说明	参考文档
了解基本概念	开始开发应用前，需要了解Flink的基本概念。	常用概念
准备开发环境和运行环境	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。	开发和运行环境简介

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。	**配置并导入样例工程**
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。	**场景说明**
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。	**编译并运行程序**
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。	**性能调优**

13.2 环境准备

13.2.1 开发和运行环境简介

在进行应用开发时，要准备的开发环境如**表13-3**所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行正常。

表 13-3 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

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。 **说明** ● 如果开发环境使用的是JDK1.7版本，则Flink集群的运行环境可以是JDK1.7也可以是JDK1.8。 ● 如果开发环境使用的是JDK1.8版本，则Flink集群的运行环境必须是JDK1.8，否则，如果运行环境是JDK1.7，则会报JDK版本错误的信息。
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：2.11.12。
安装Scala插件	Scala开发环境的基本配置。版本要求：1.5.4。
准备开发用户	参考**准备开发用户**章节配置。
安装客户端	参考**安装客户端**章节配置。

13.2.2 准备开发用户

开发用户用于运行样例工程。在安全集群中，用户需要有HDFS、YARN、Kafka和Flink权限，才能运行Flink样例工程。

前提条件

MRS服务集群开启了Kerberos认证，没有开启Kerberos认证的集群忽略该步骤。

操作步骤

步骤1 登录MRS Manager，在MRS Manager界面选择“系统设置 > 角色管理 > 添加角色”。

1. 填写角色的名称，例如flinkrole。
2. 在“权限”的表格中选择“HDFS > File System > hdfs://hacluster/”，勾选“Read”、“Write”和“Execute”，单击“权限”表格中“服务”返回。
3. 在“权限”的表格中选择“Yarn > Scheduler Queue > root”，勾选default的“Submit”，单击“确定”保存。

说明

根据以上角色的设置，用户提交应用后，会在客户端打印WARN日志。出现WARN日志是由于Flink会去YARN获取资源剩余值并进行检测评估，但该操作需要admin操作权限，可用户并没有设置该权限。该问题不影响任务提交执行，可以忽略。WARN日志如下：

```
Get node resource from yarn cluster. Yarn cluster occur exception:
org.apache.hadoop.yarn.exceptions.YarnPermissionDeniedException: User flinkuser does not have privilage to see, admin only
```

步骤2 在MRS Manager界面选择“系统设置 > 用户组管理 > 添加用户组”，为样例工程创建一个用户组，例如flinkgroup。

步骤3 在MRS Manager界面选择“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。填写用户名例如**flinkuser**，用户类型为“人机”用户，加入用户组**flinkgroup**和**hadoop**，并绑定角色**flinkrole**取得权限，单击“确定”。

说明

- 在客户端修改flinkuser密码后才能使用本用户。
- 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flinkuser用户加入“kafkaadmin”用户组。
- 当用户需要运行**向Kafka生产并消费数据程序**（Scala和Java语言）样例程序时，需要加入“kafkaadmin”用户组。

步骤4 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名中选择flinkuser，单击操作中下载认证凭据文件，保存后解压得到用户的keytab文件与krb5.conf文件，并将krb5.conf文件拷贝到客户端机器的**/etc**目录下。用于在样例工程中进行安全认证，具体使用请参考**准备安全认证**。

----结束

13.2.3 安装客户端

Flink使用Windows环境进行开发，运行环境则建议部署在Linux环境下，MRS客户端不支持安装在Windows环境。您可以执行如下操作完成客户端的配置。

操作步骤

步骤1 安装Flink客户端。

1. 确认服务端Flink组件已经安装。
2. 下载Flink客户端程序。
   a. 登录MRS Manager。
   b. 单击“服务管理 > Flink > 下载客户端”，在“客户端类型”勾选“完整客户端”，“下载路径”选择“服务器端”，单击“确定”下载客户端到服务器端。
3. 执行如下命令解压缩客户端安装包“MRS_Flink_Client.tar”。

   tar -xvf /tmp/MRS-client/MRS_Flink_Client.tar

   tar -xvf /tmp/MRS-client/MRS_Flink_ClientConfig.tar
4. 进入解压文件夹（**/tmp/MRS-client/MRS_Flink_ClientConfig**），执行**./install.sh*#{client_install_home}***安装客户端。

   例如：**./install.sh /opt/flinkclient**

说明

若集群开启Kerberos认证，如需在集群外的节点上使用客户端，请在该客户端的flink配置文件flink-conf.yaml的配置项“jobmanager.web.allow-access-address”中添加该客户端所在节点的IP。若集群未开启Kerberos认证则无需修改该配置项。

步骤2 配置客户端网络连接。

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。

1. 确认客户端与服务端各个主机网络上互通。
2. 将服务端主机名与IP映射关系添加到客户端的hosts文件中。
3. 如果用户使用yarn-client模式，还需要将客户端的主机名与IP的映射关系添加到Yarn ResourceManager节点的hosts文件中。

   说明

   linux环境文件位置为“/etc/hosts”，Windows环境文件位置为“C:\Windows\System32\drivers\etc\hosts”
4. 确认客户端和集群时间一致性。客户端机器的时间与Flink集群的时间要保持一致，时间差要小于5分钟。
5. 检查Flink客户端的“flink-conf.yaml”配置文件中的配置项是否配置正确。

----结束

13.2.4 配置并导入样例工程

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13-2所示。

图 13-2 导入样例工程流程

操作步骤

步骤1 参照**样例工程获取地址**，下载样例工程到本地。

步骤2 将**安装客户端**章节中下载的客户端安装包复制到Windows服务器中。

步骤3 将Windows服务器中的“MRS_Flink_Client.tar”文件解压缩得到“MRS_Flink_ClientConfig.tar”，再解压缩“MRS_Flink_ClientConfig.tar”得到“MRS_Flink_ClientConfig”文件夹。

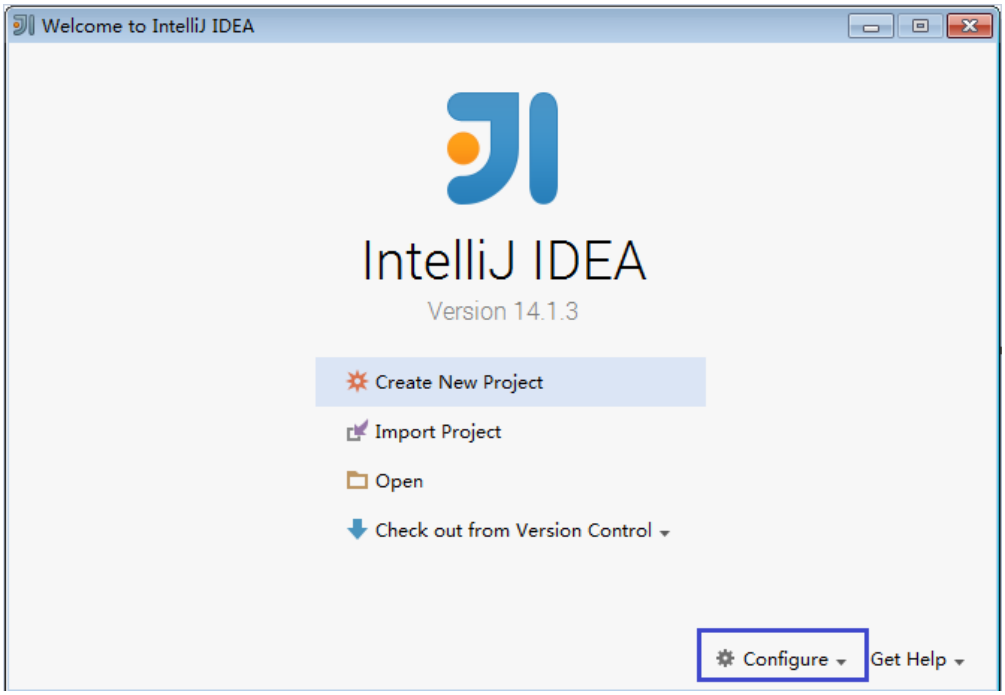
步骤4 双击“MRS_Flink_ClientConfig/Flink”目录下的“flink_install.bat”脚本，安装成功后得到“lib”文件夹、“examples”文件夹。

- “lib”文件夹：只包含Flink依赖jar包，Kafka依赖jar包请去对应组件的服务端安装目录查找并添加。
- “examples”文件夹：可获取开源样例jar包。

步骤5 在导入样例工程之前，IntelliJ IDEA需要进行配置JDK。

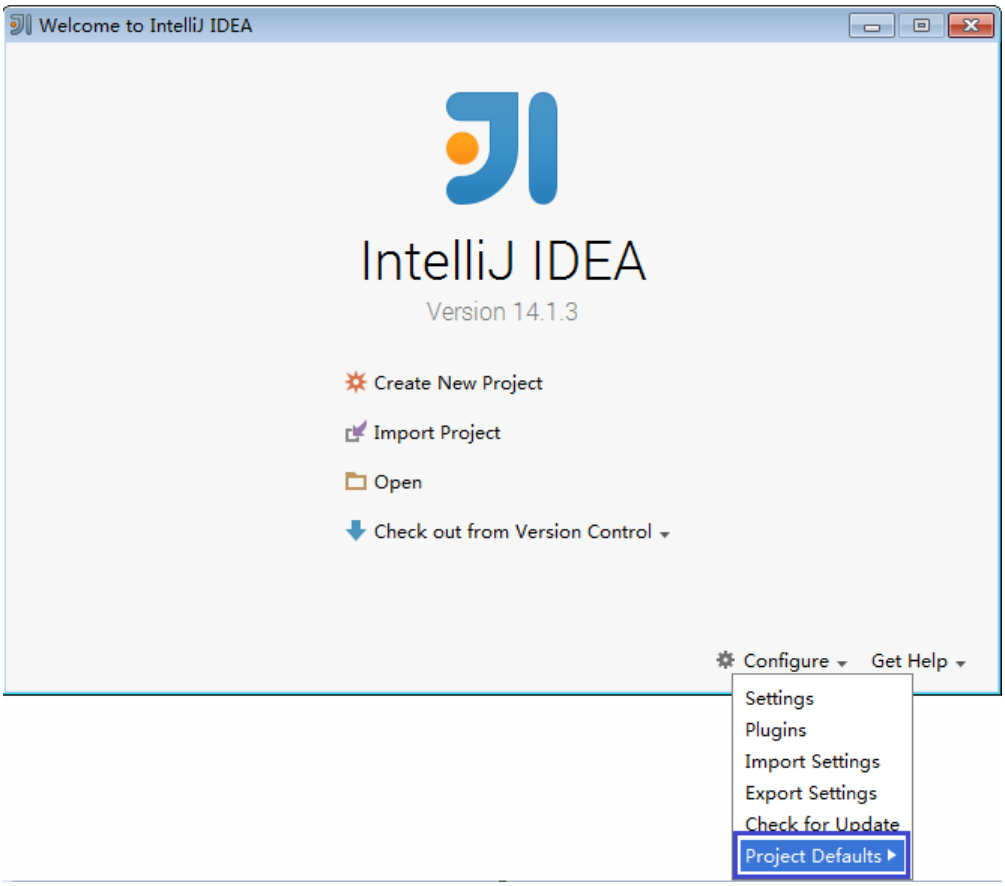
1. 打开IntelliJ IDEA，单击“Configure”下拉按钮。

图 13-3 Choosing Configure



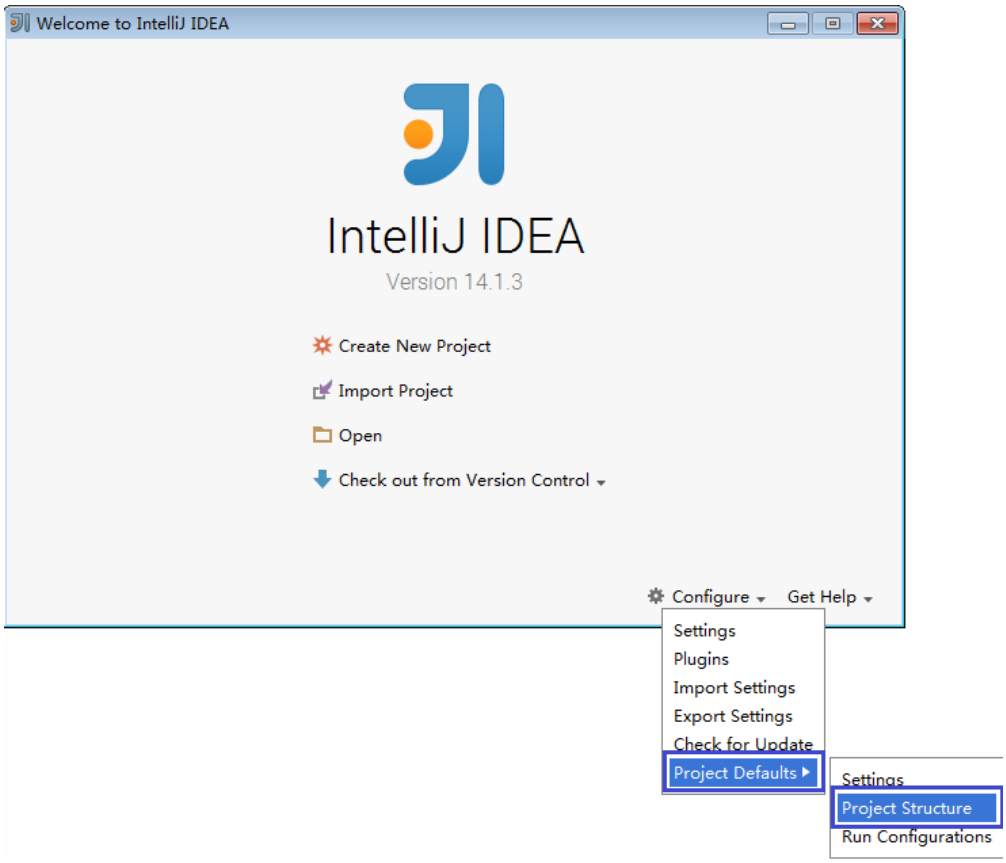
2. 在“Configure”下拉菜单中单击“Project Defaults”。

图 13-4 Choosing Project Defaults



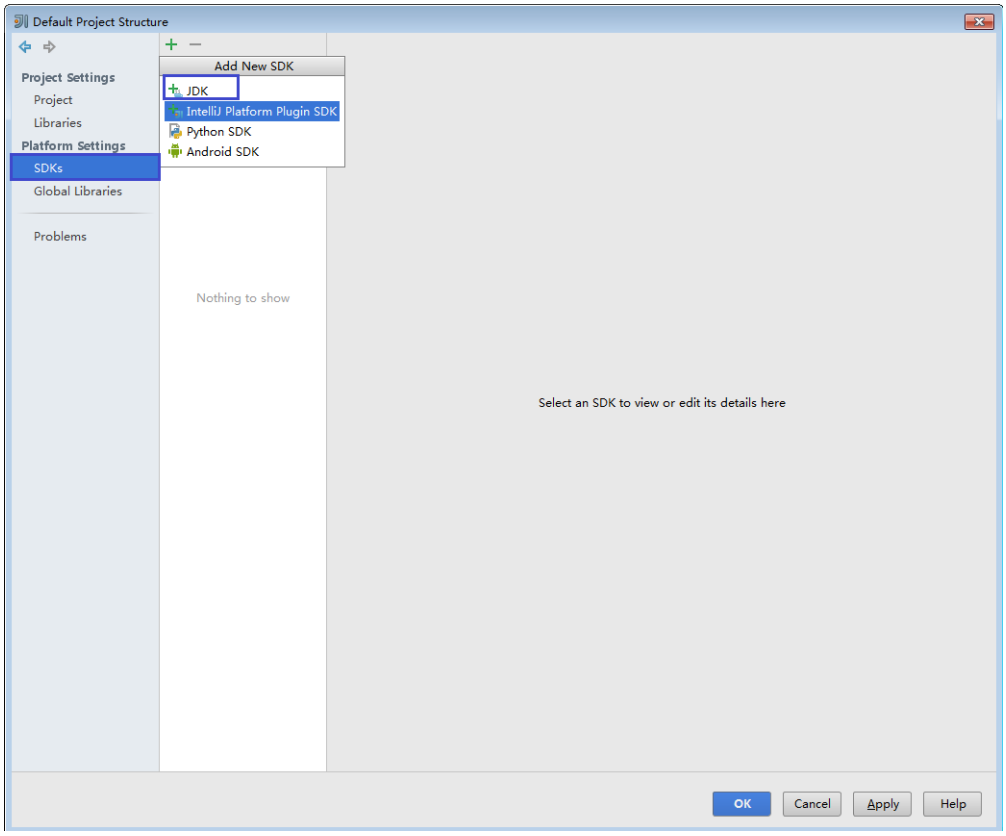
3. 在“Project Defaults”菜单中选择“Project Structure”。

图 13-5 Project Defaults



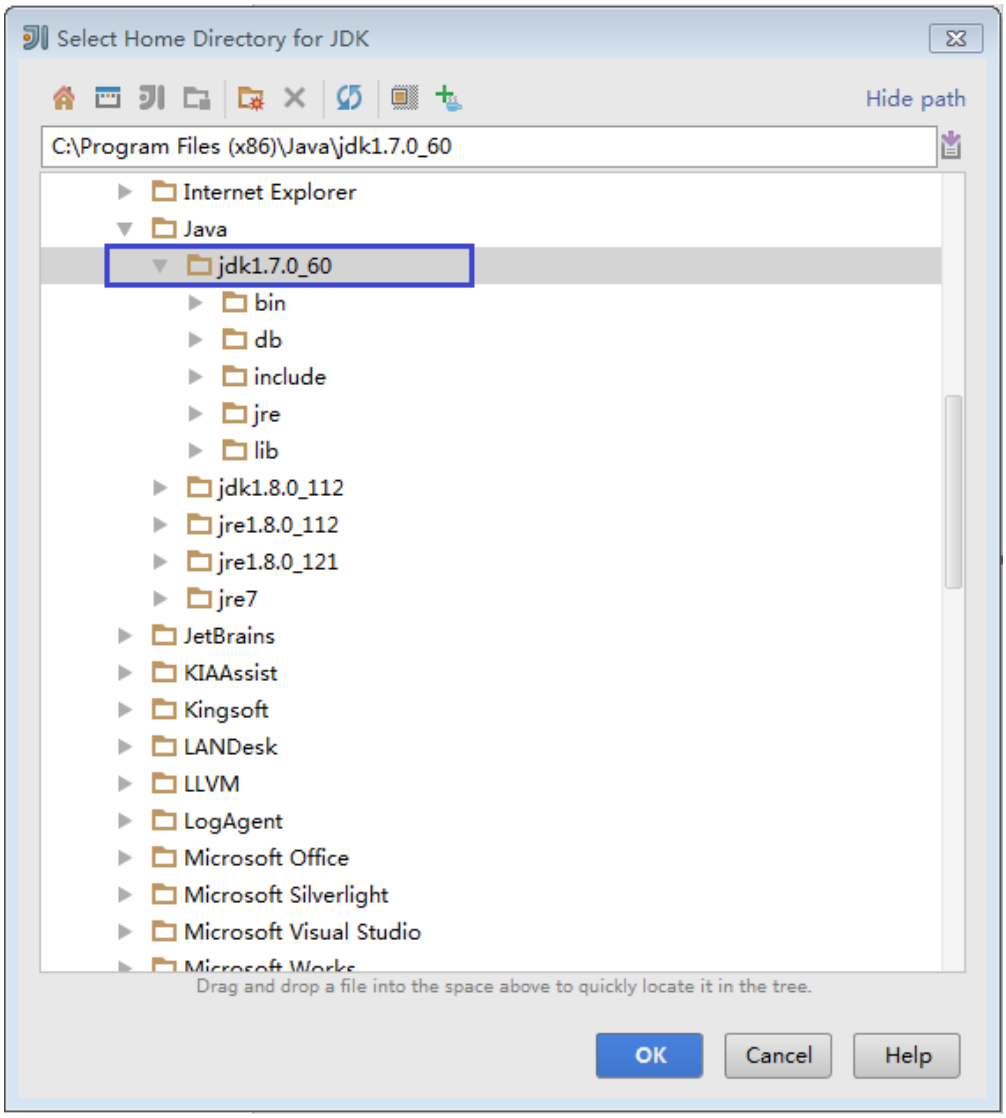
4. 在打开的“Project Structure”页面中，选择“SDKs”，单击绿色加号添加JDK。

图 13-6 添加 JDK



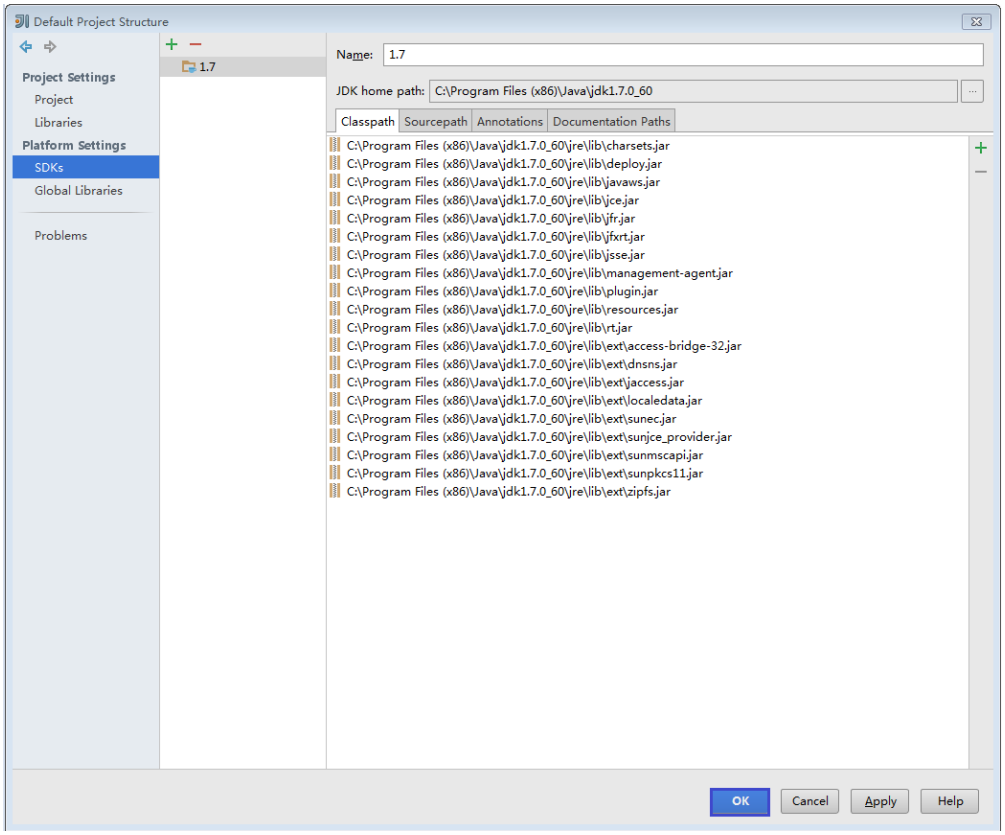
5. 在弹出的“Select Home Directory for JDK”窗口，选择对应的JDK目录，然后单击“OK”。

图 13-7 选择 JDK 目录



6. 完成JDK选择后，单击“OK”完成配置。

图 13-8 完成 JDK 配置

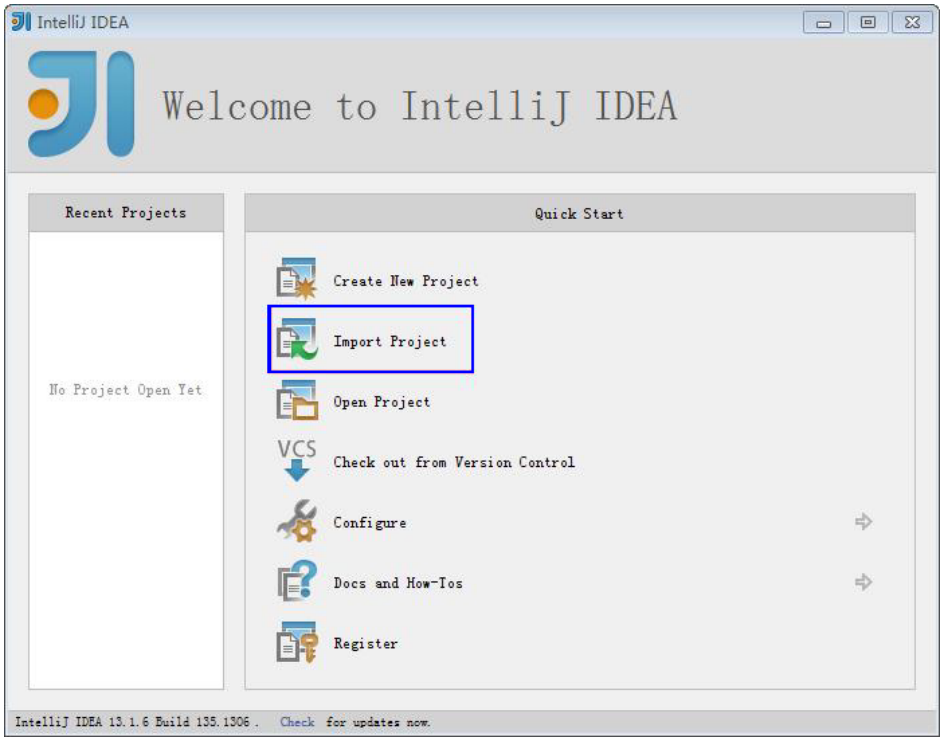


步骤6 将Java样例工程导入到IDEA中。

1. 打开IntelliJ IDEA。在“Quick Start”页面选择“Import Project”。

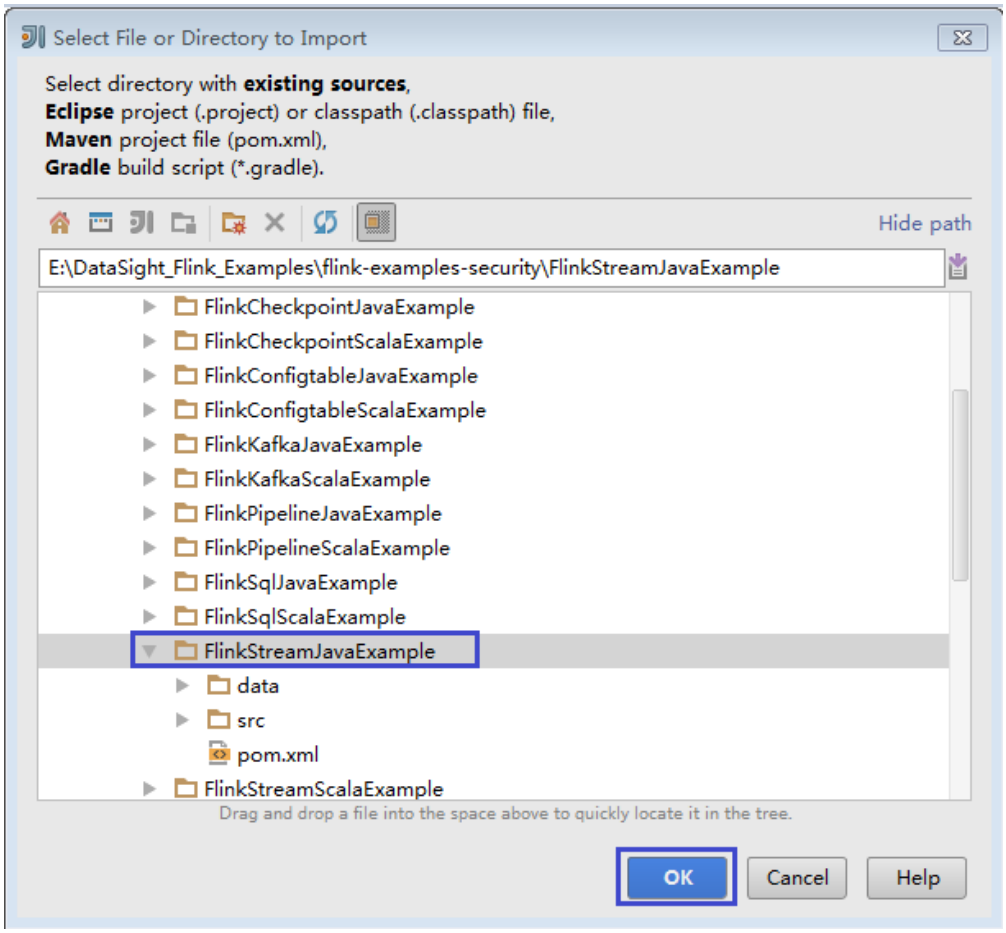
   或者，针对已使用过的IDEA工具，您可以从IDEA主界面直接添加。选择“File > Import project...”导入工程。

图 13-9 Import Project（Quick Start 页面）



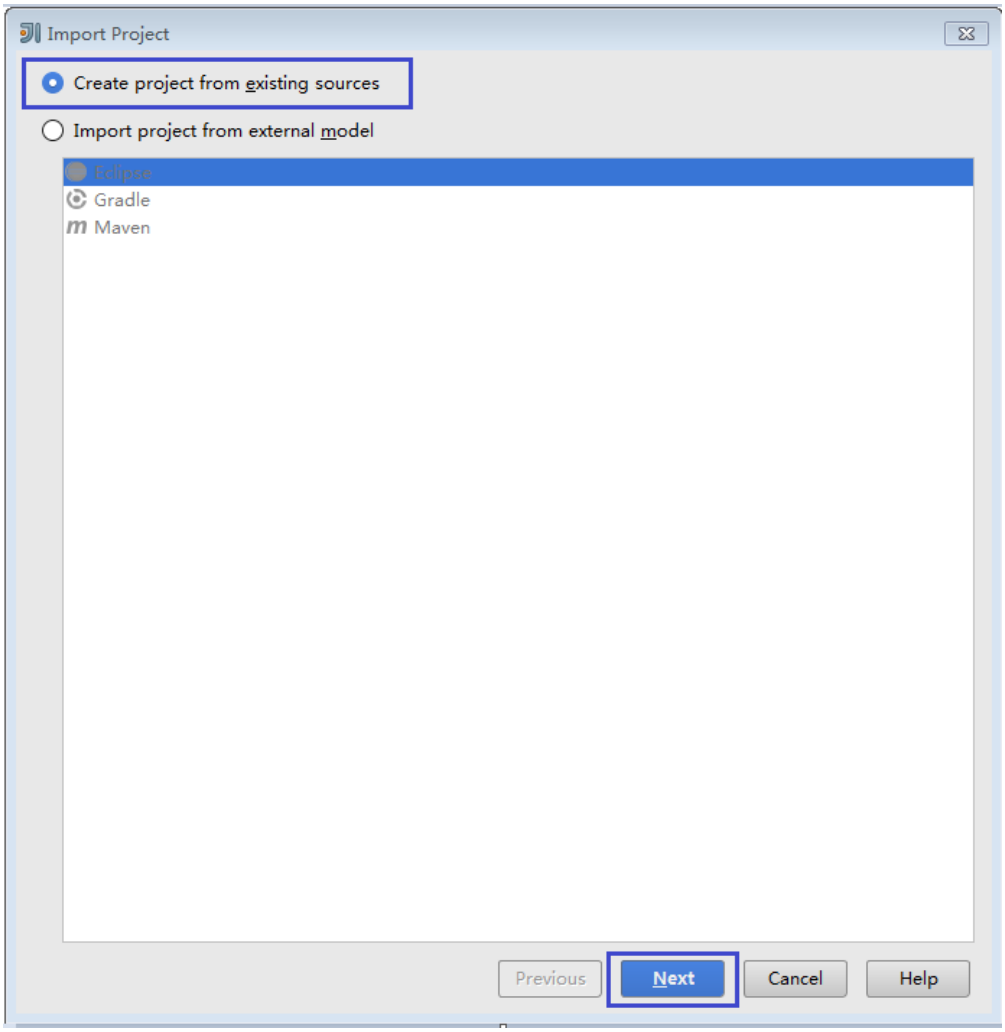
2. 选择需导入的样例工程存放路径，单击“OK”。

图 13-10 Select File or Directory to Import



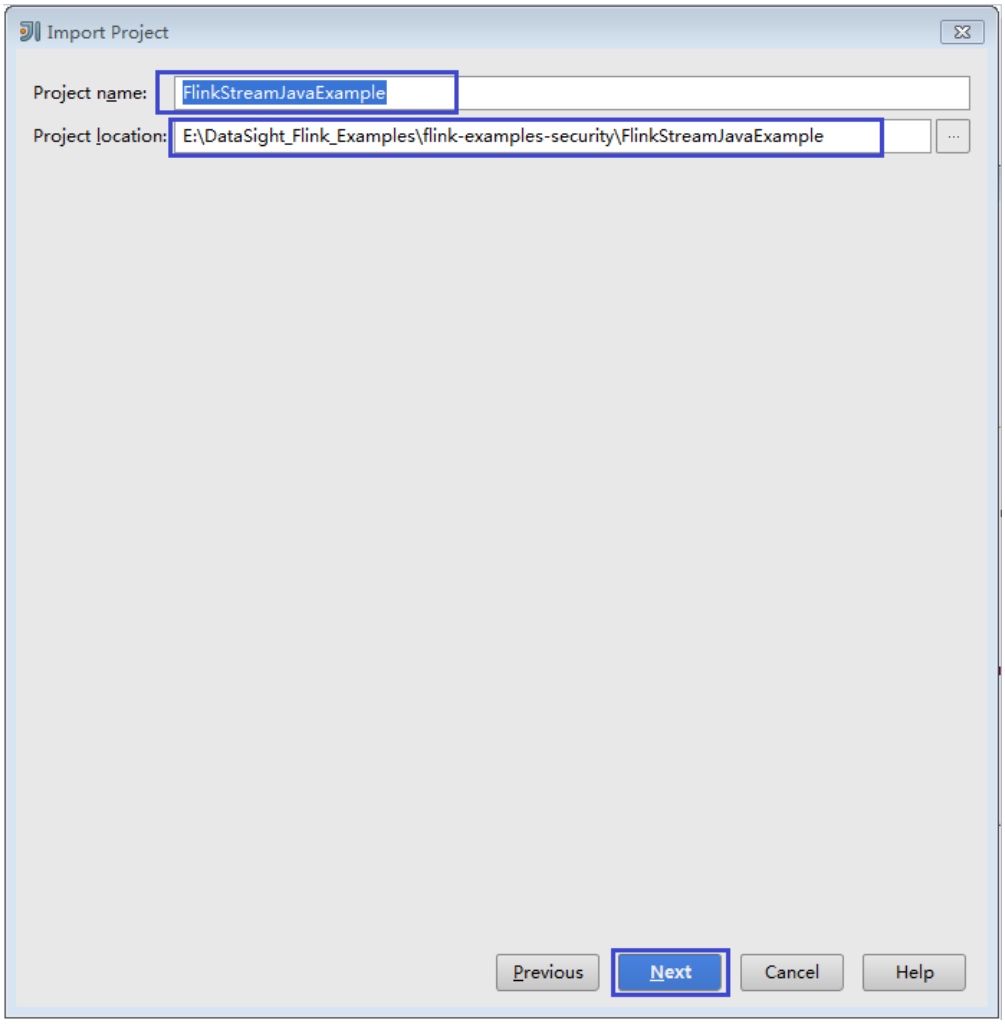
3. 选择从已存在的源码创建工程，然后单击“Next”。

图 13-11 Create project from existing sources



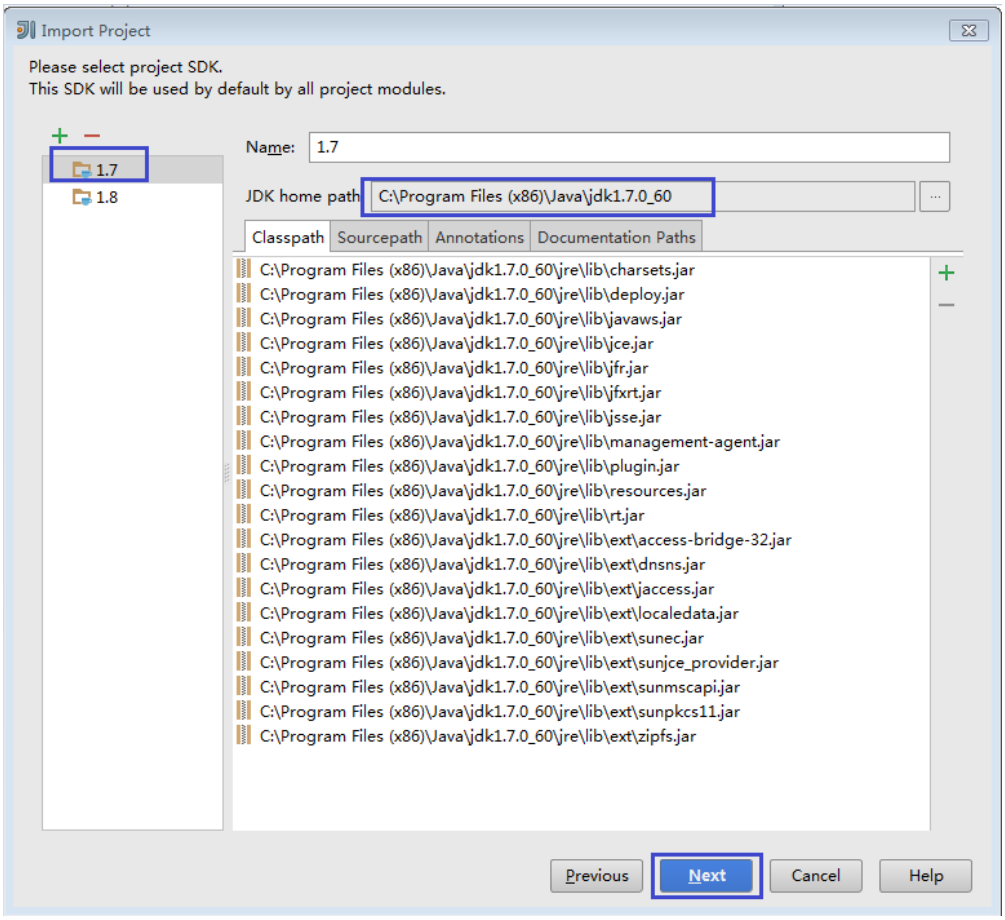
4. 确认导入路径和名称，单击“Next”。

图 13-12 Import Project



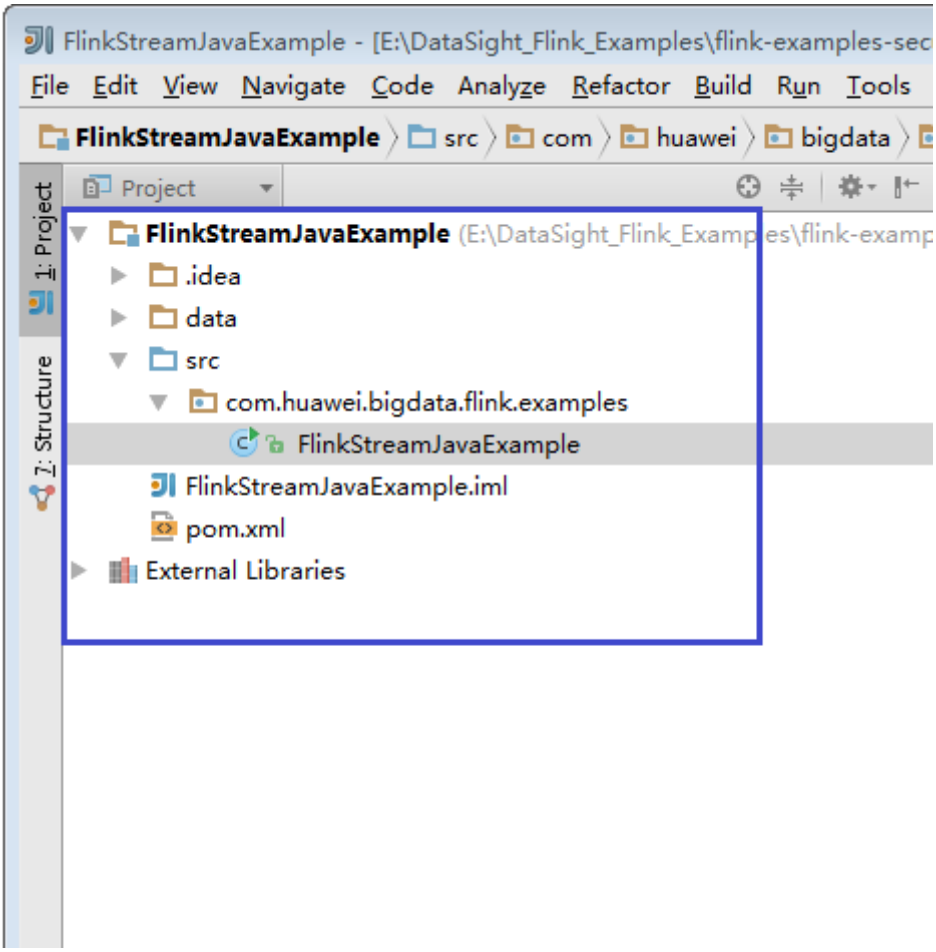
5. 确认导入工程的root目录，默认即可，单击“Next”。
6. 确认IDEA自动识别的依赖库以及建议的模块结构，默认即可，单击“Next”。
7. 确认工程所用JDK，单击“Next”。

图 13-13 Select project SDK



8. 导入结束，单击“Finish”，IDEA主页显示导入的样例工程。

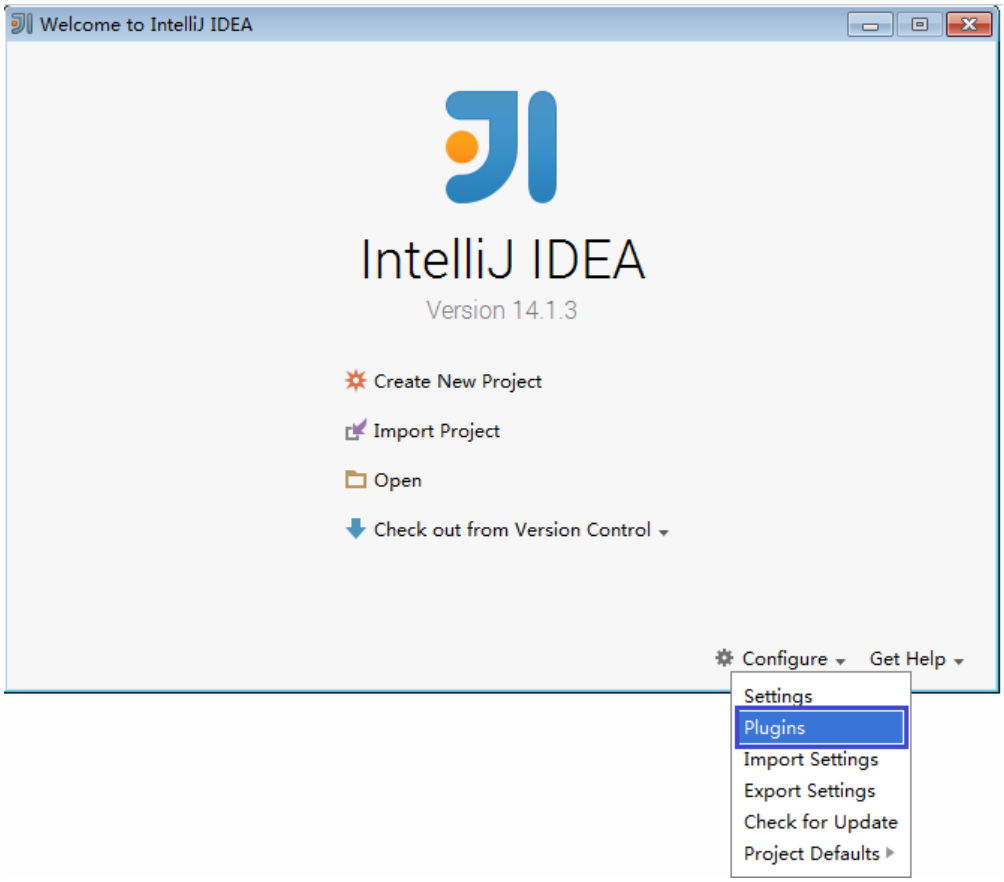
图 13-14 已导入工程



步骤7 （可选）如果导入Scala语言开发的样例程序，还需要在IntelliJ IDEA中安装Scala插件。

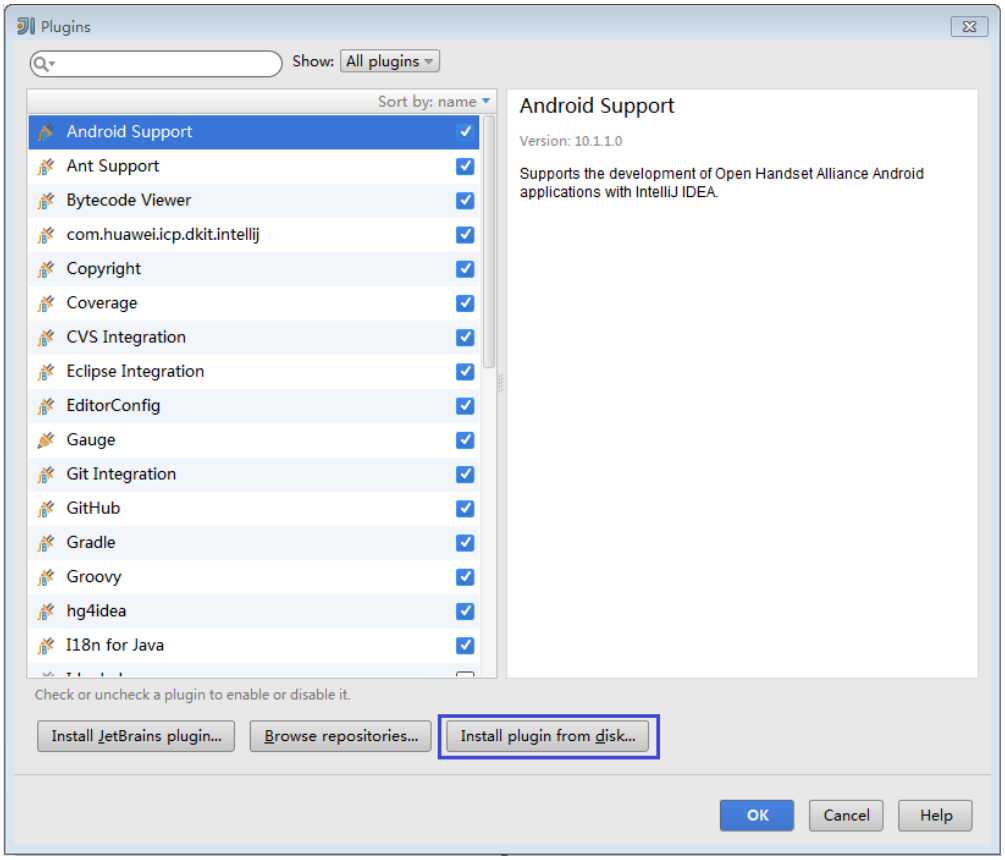
1. 在“Configure”下拉菜单中，单击“Plugins”。

图 13-15 Plugins



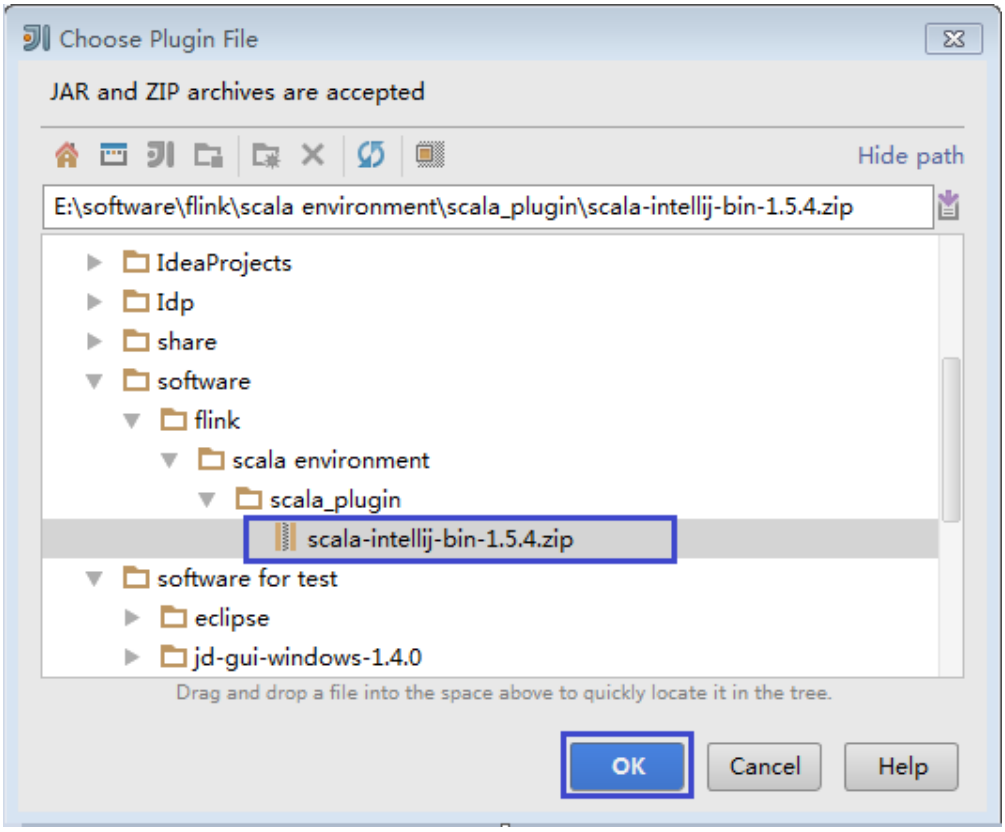
2. 在“Plugins”页面，选择“Install plugin from disk”。

图 13-16 Install plugin from disk



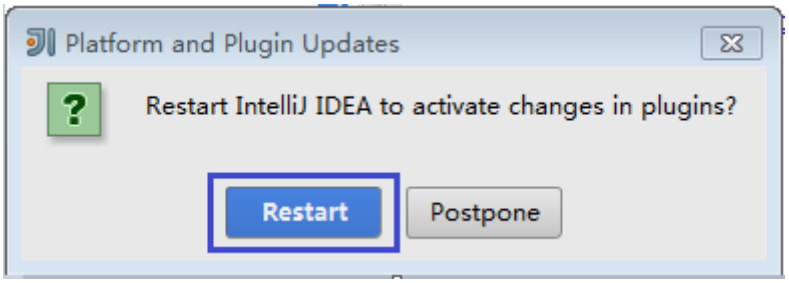
3. 在“Choose Plugin File”页面，选择对应版本的Scala插件包，单击“OK”。

图 13-17 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。

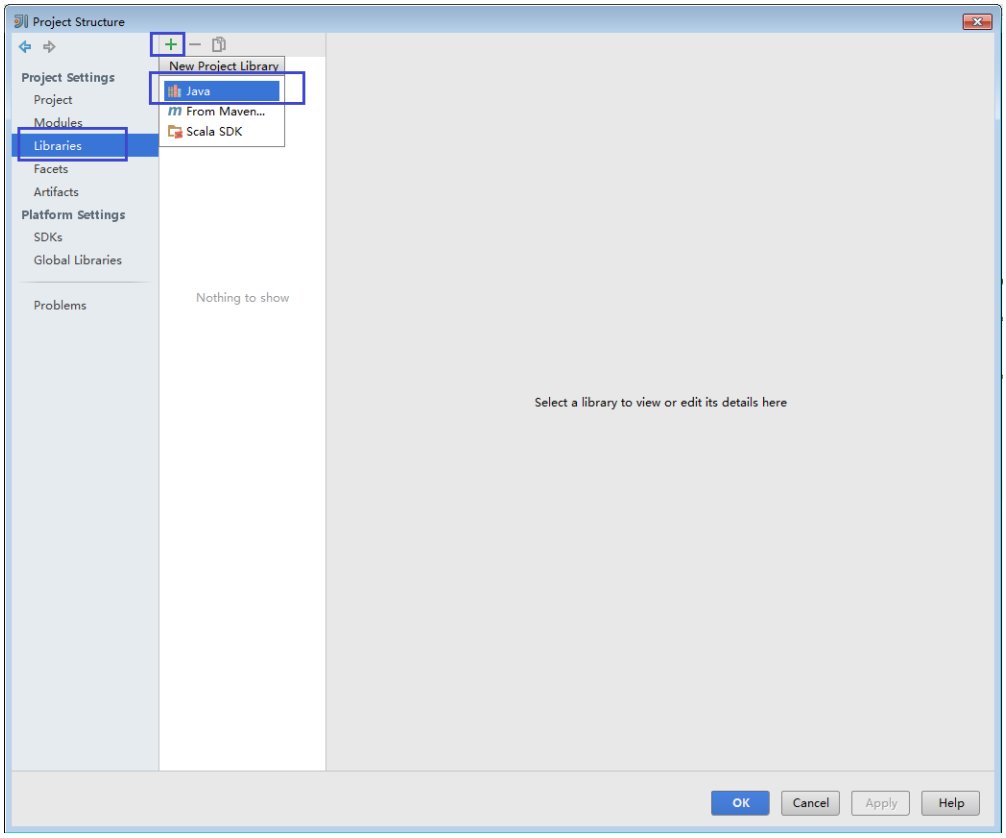
图 13-18 Platform and Plugin Updates



步骤8 导入样例工程依赖的Jar包。

1. 在IDEA主页，选择“File>Project Structures...”进入“Project Structure”页面。
2. 选择“Libraries”，然后单击“+”，增加“Java”的依赖包。

图 13-19 Add Java



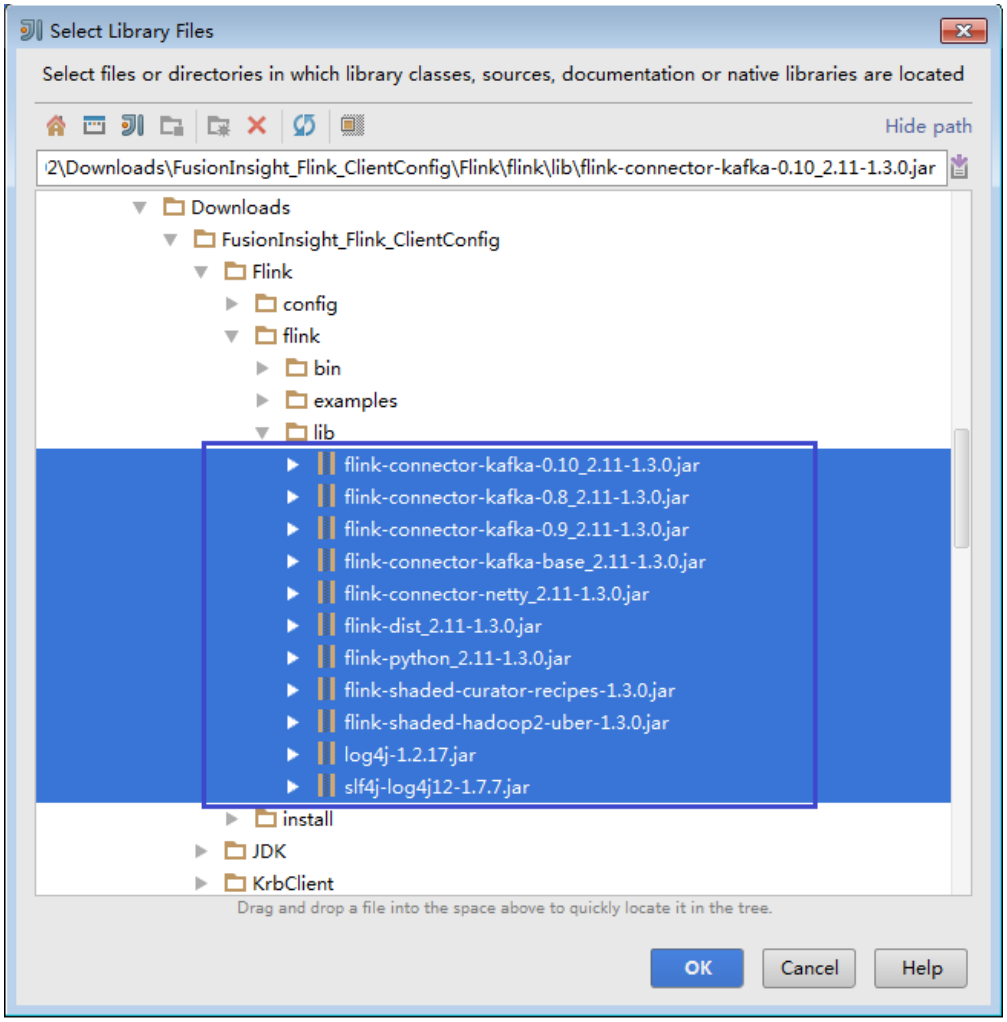
3. 在“Select Library Files”页面，选中“lib”目录下的所有Jar包，然后单击“OK”。

   Flink相关的依赖包：选择“lib”目录下的所有Jar包。或者可以根据不同样例工程，最小化选择其对应的Jar包。

   说明

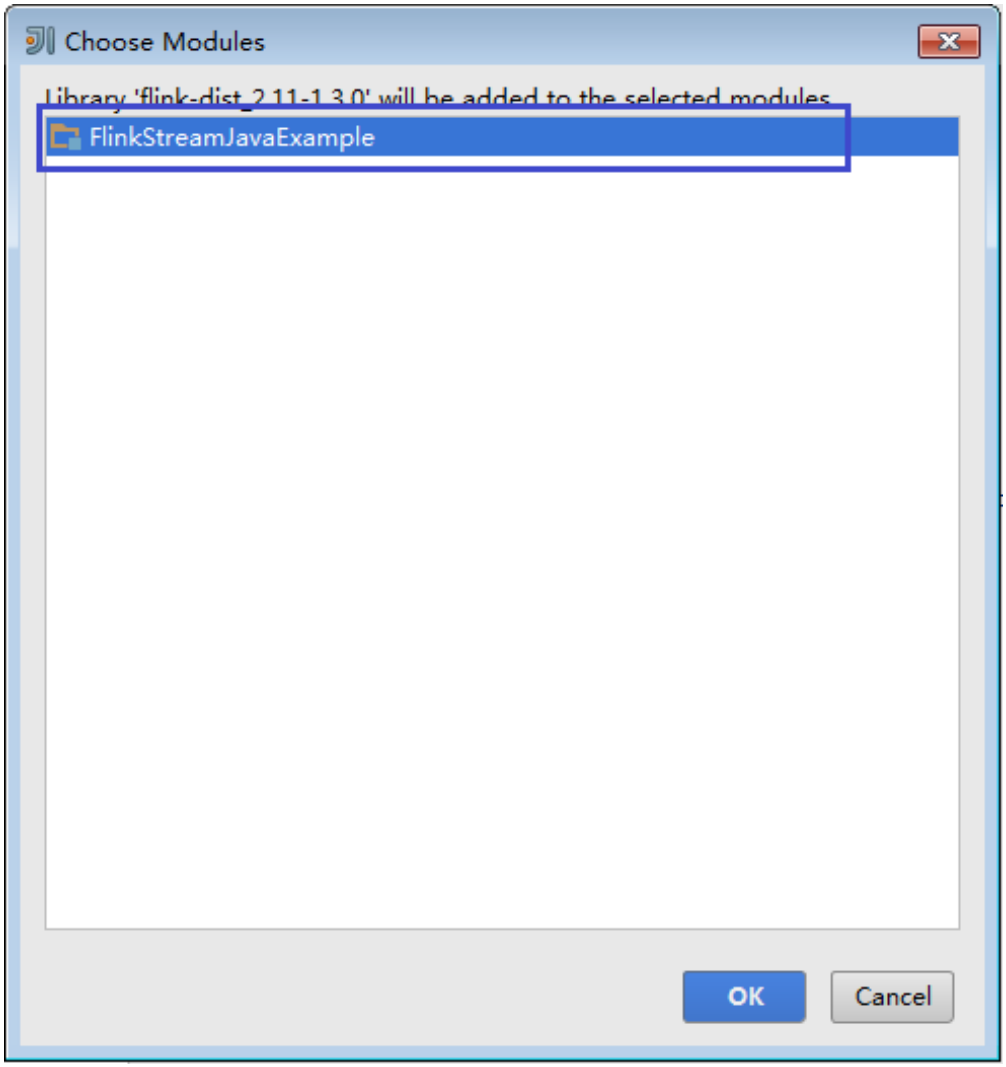
   当样例代码使用其他MRS组件时，请去对应MRS组件的服务端安装目录查找并添加依赖包。

图 13-20 Select Library Files



在“Choose Modules”页面，选择对应的模块，样例工程选择所有模块即可。然后单击“OK”。

图 13-21 Choose Modules

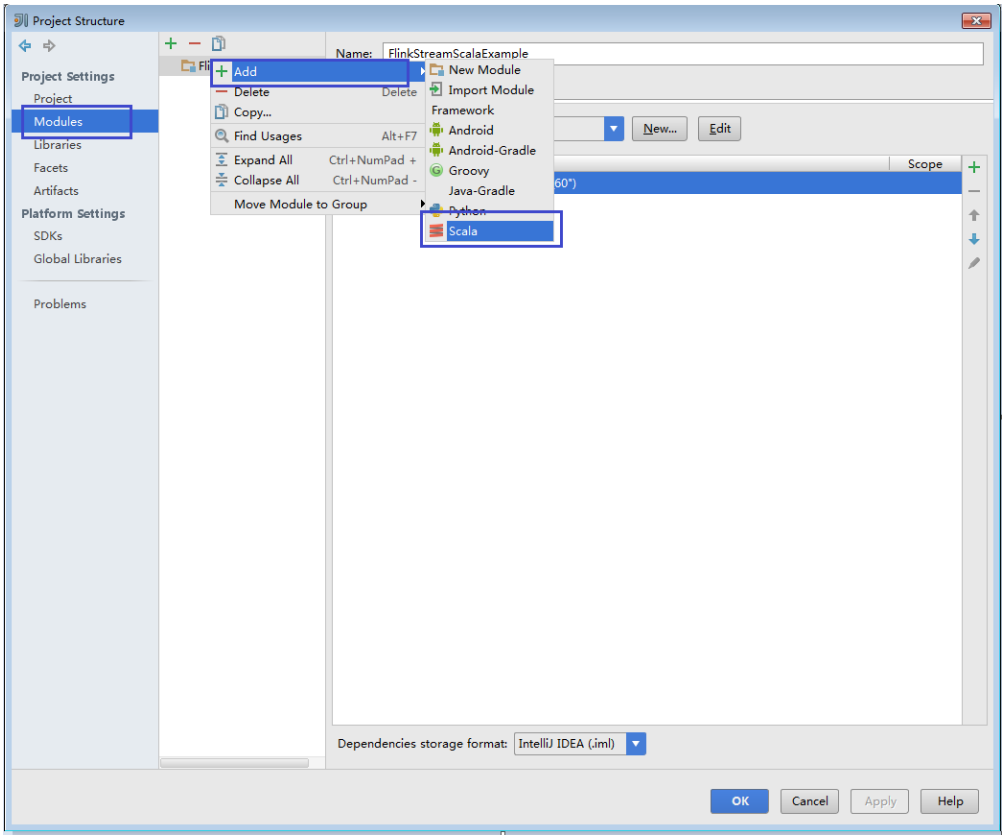


4. 最后，单击“OK”完成依赖包导入。

步骤9 （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。

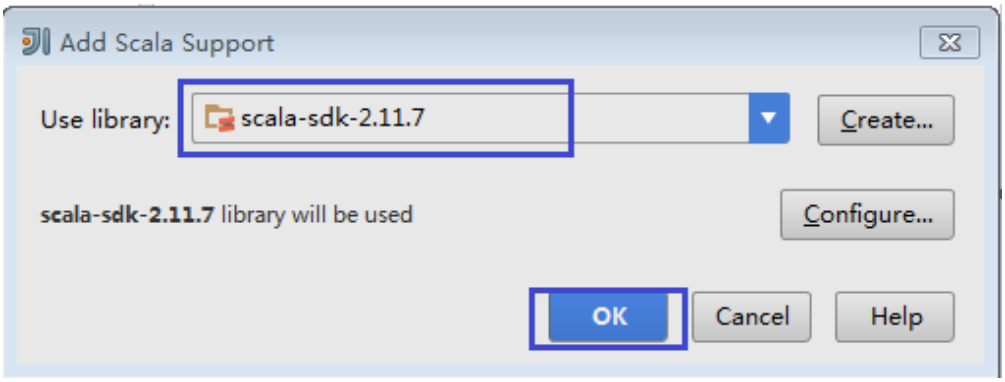
1. 在IDEA主页，选择“File>Project Structures...”进入“Project Structure”页面。
2. 选择“Modules”，选中工程名称，然后右键选择“Add > Scala”。

图 13-22 选择 Scala 语言



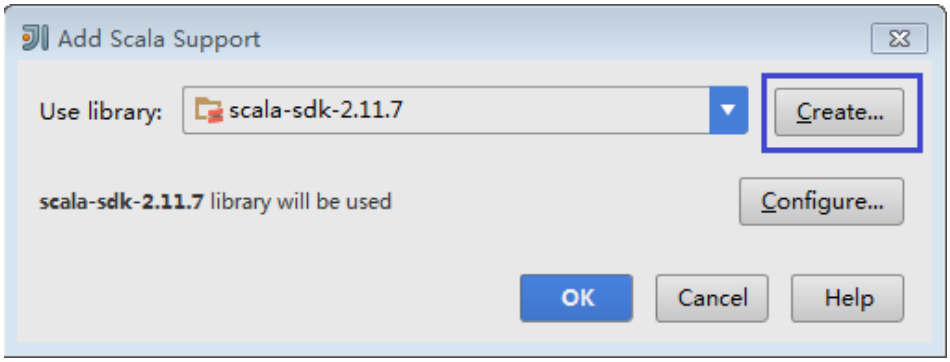
3. 当IDEA可以识别出Scala SDK时，在设置界面，选择编译的依赖jar包，然后单击“OK”应用设置

图 13-23 Add Scala Support



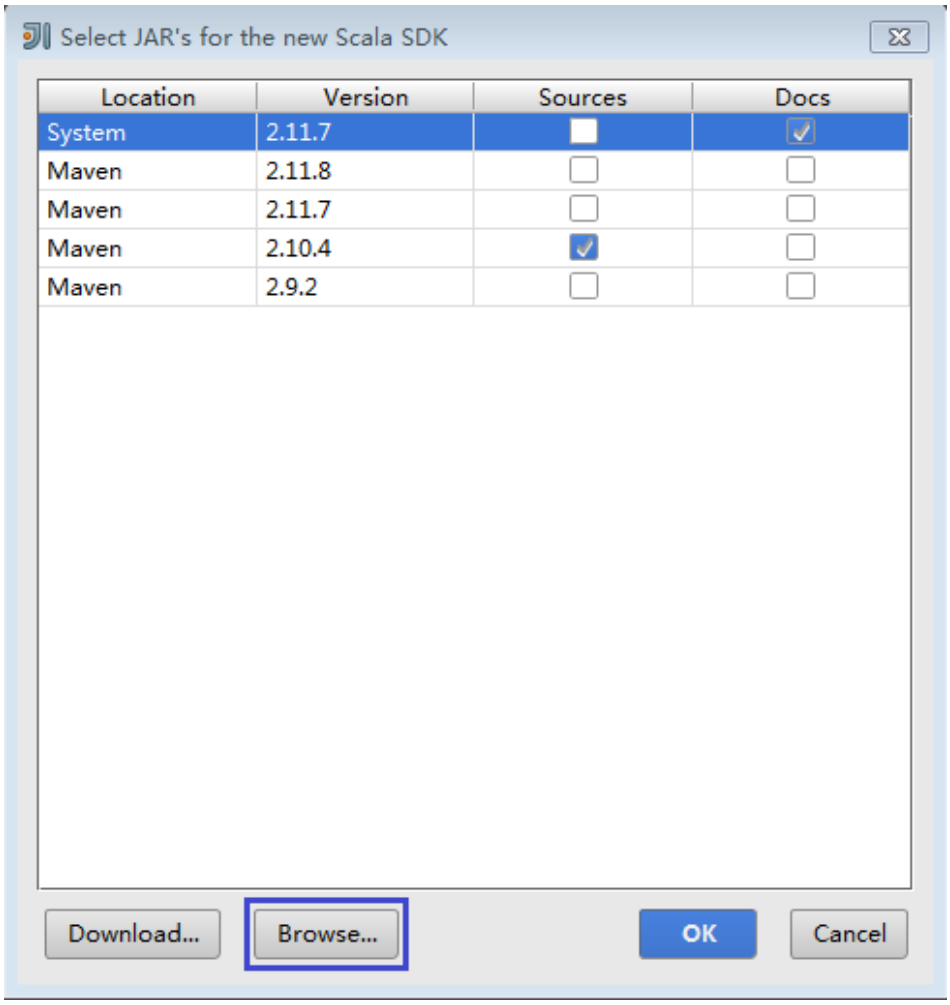
4. 当系统无法识别出Scala SDK时，需要自行创建。
   a. 单击“Create...”。

图 13-24 Create...



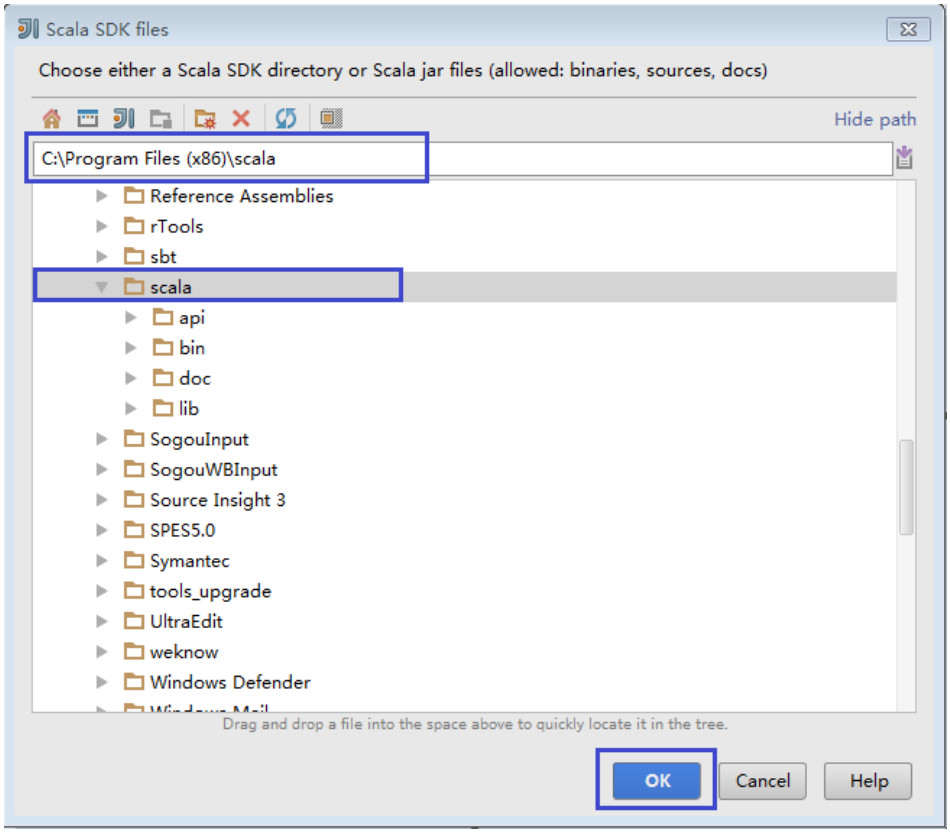
b. 在“Select JAR's for the new Scala SDK”页面单击“Browse...”。

图 13-25 Select JAR's for the new Scala SDK



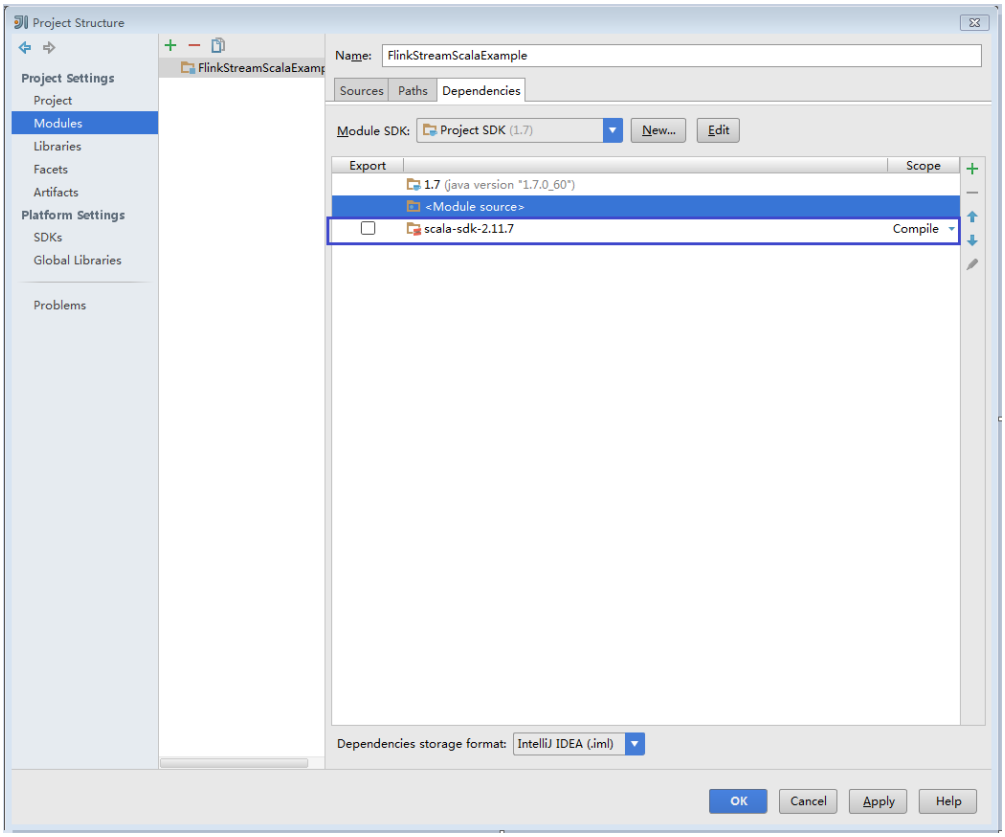
c. 在“Scala SDK files”页面选择scala sdk目录，单击“OK”。

图 13-26 Scala SDK files



5. 设置成功，单击“OK”保存设置。

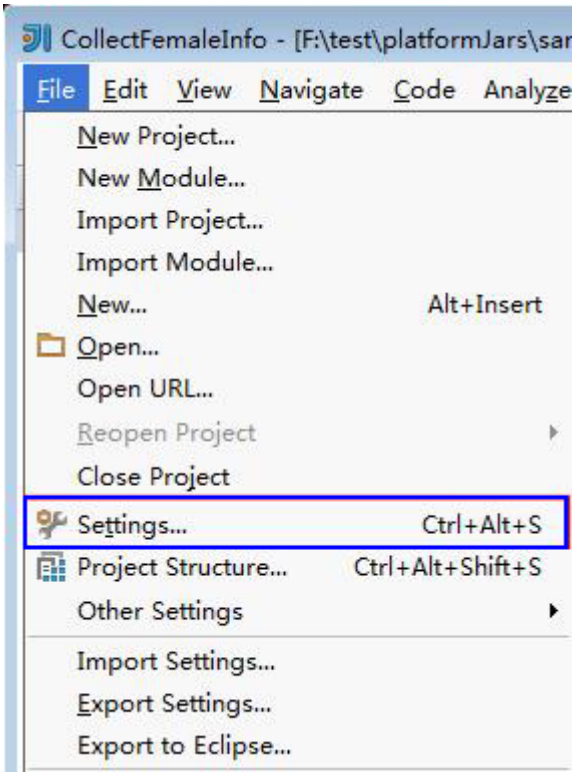
图 13-27 设置成功



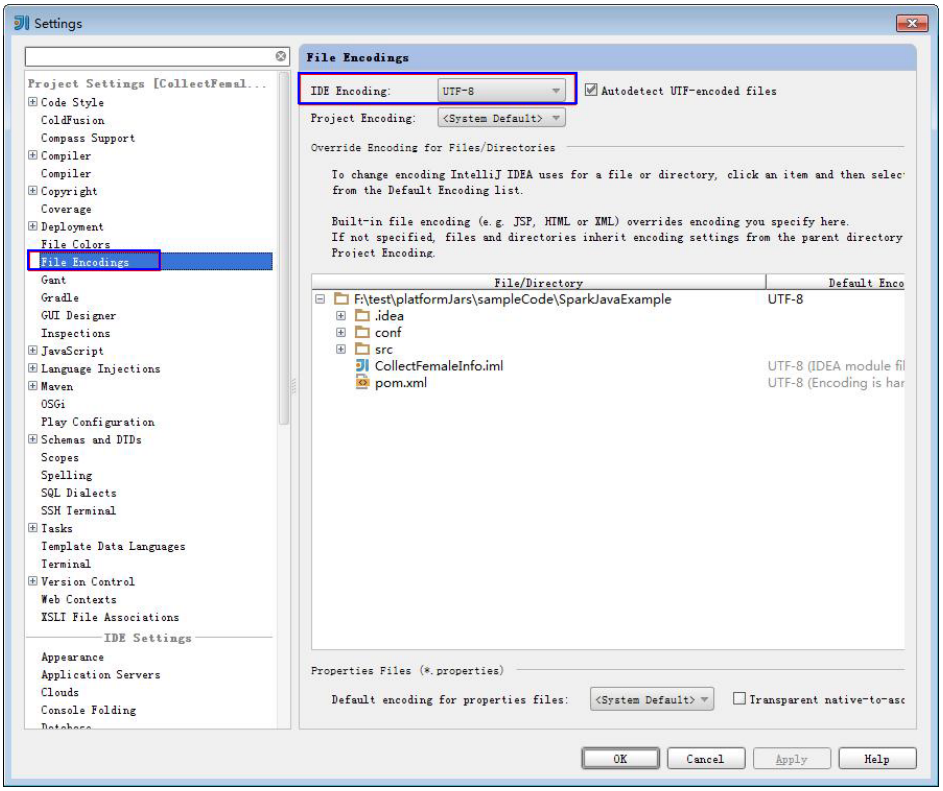
步骤10 设置IDEA的文本文件编码格式，解决乱码显示问题。

1. 在IDEA首页，选择“File > Settings...”。

图 13-28 选择 Settings



2. 在“Settings”页面，展开“Editor”，选择“File Encodings”。然后分别在右侧的“IDE Encoding”和“Project Encoding”的下拉框中，选择“UTF-8”。单击“Apply”。



3. 单击“OK”完成编码配置。

----结束

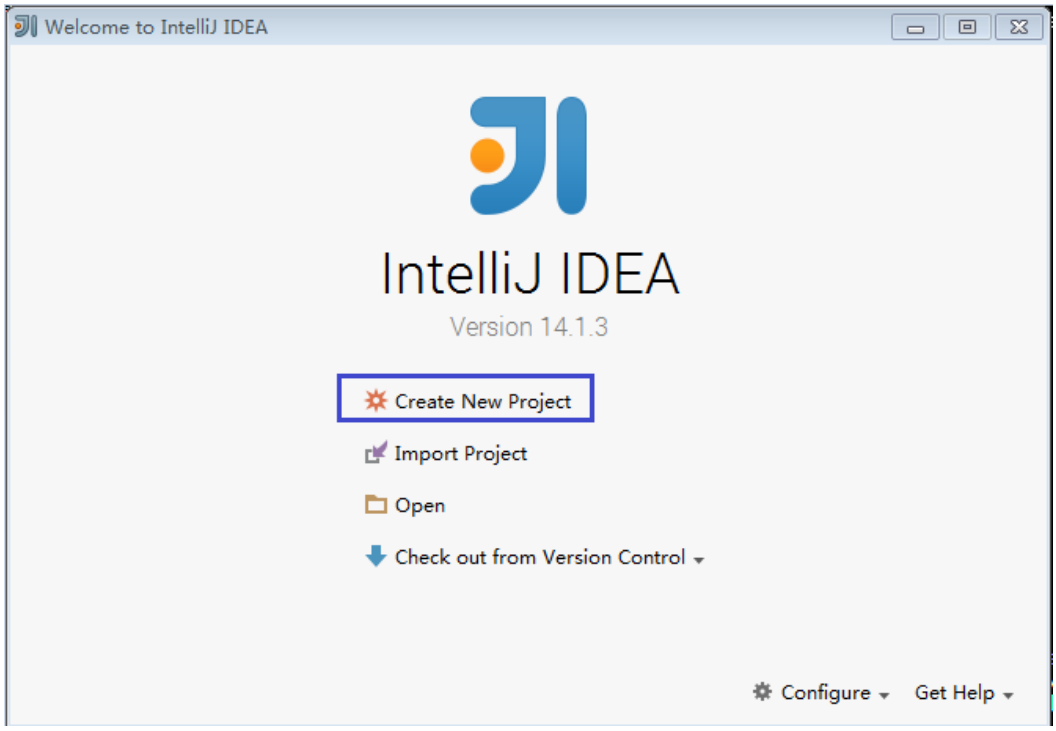
13.2.5 新建工程（可选）

除了导入Flink样例工程，您还可以使用IDEA新建一个Flink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”。

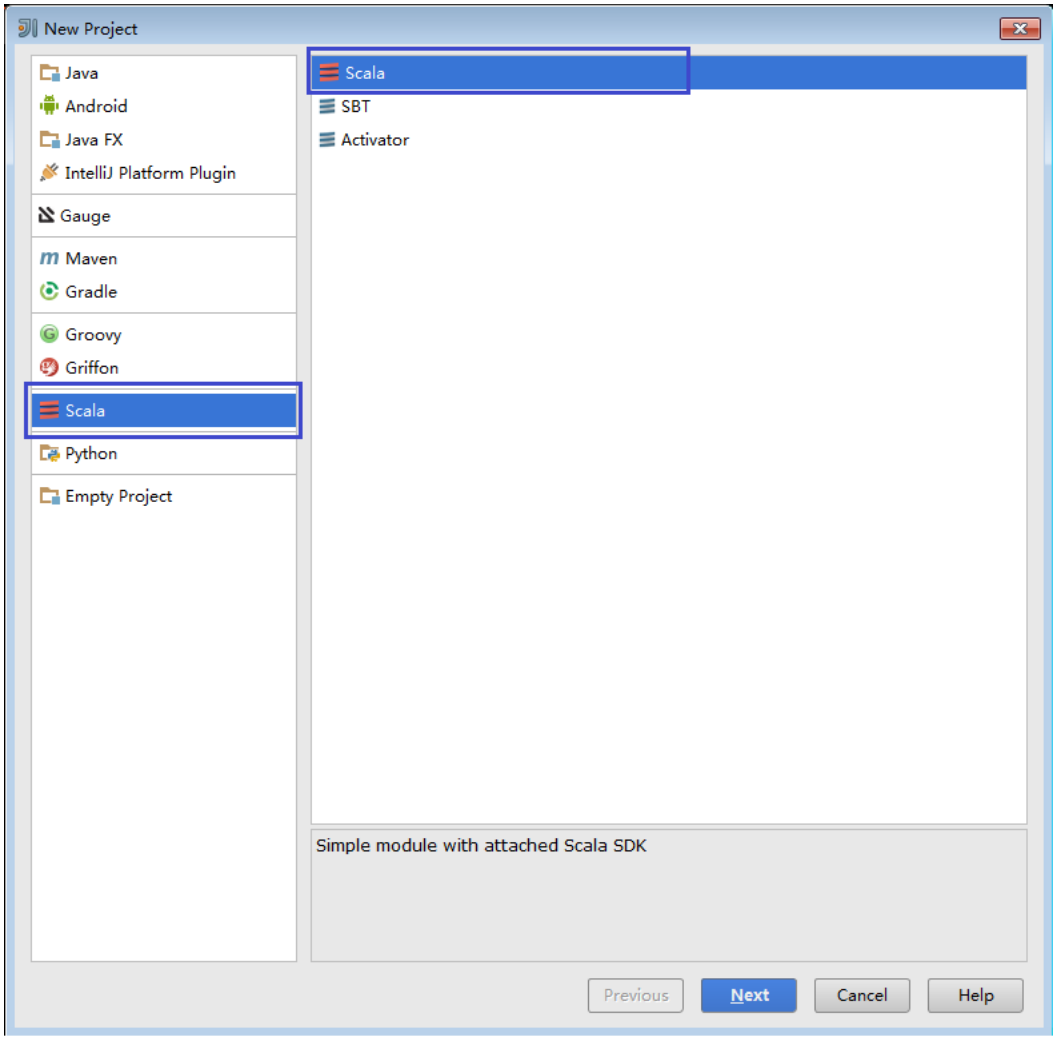
图 13-29 创建工程



步骤2 在“New Project”页面，选择“Scala”开发环境，并选择“Scala Module”，然后单击“Next”。

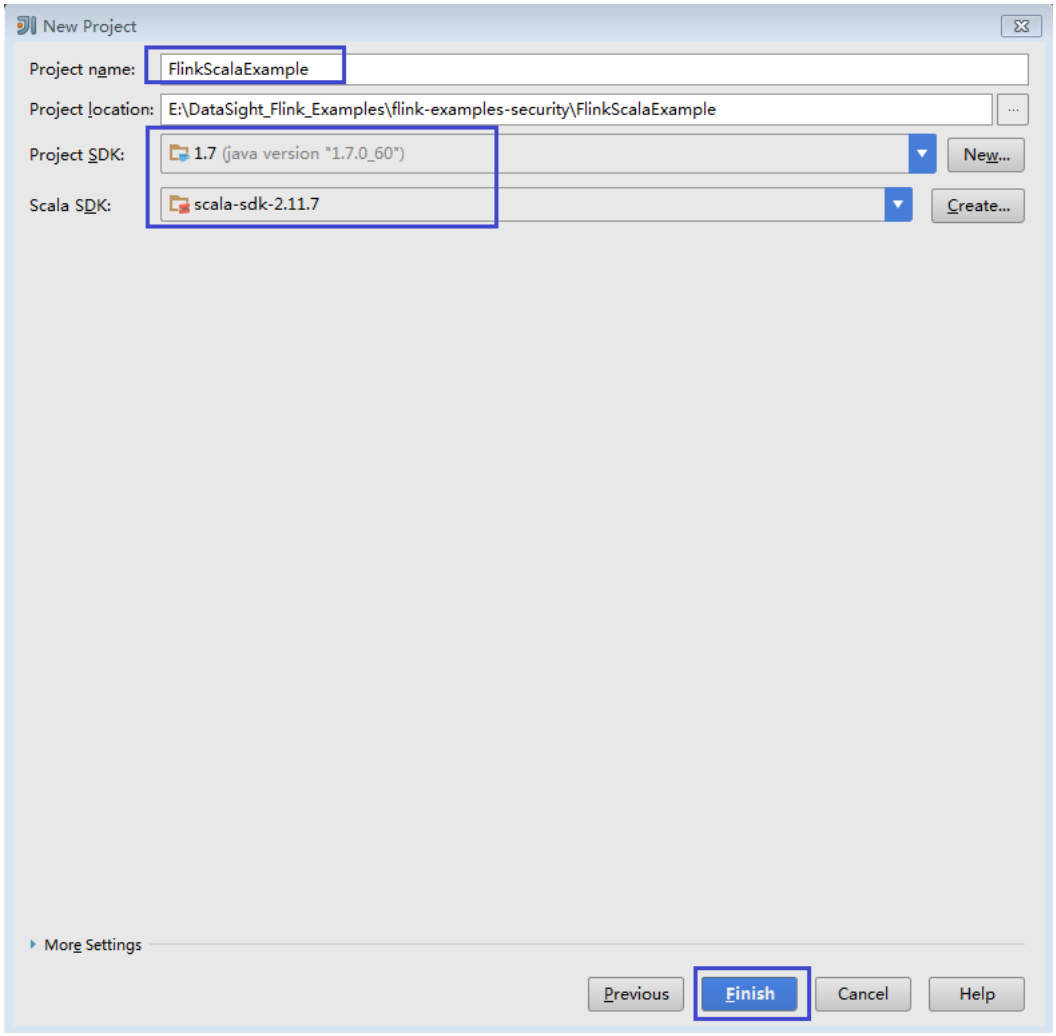
如果您需要新建Java语言的工程，选择对应参数即可。

图 13-30 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版本和Scala SDK，然后单击“Finish”完成工程创建。

图 13-31 填写工程信息



----结束

13.2.6 准备安全认证

MRS服务集群开启了Kerberos认证需要执行以下步骤，没有开启Kerberos认证的集群忽略该步骤。

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

Flink整个系统有两种认证方式：

- 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。

表 13-4 安全认证方式

安全认证方式	说明	配置方法
Kerberos认证	当前只支持keytab认证方式。	1. 从KDC服务器上下载用户keytab，并将keytab放到Flink客户端所在主机的某个文件夹下(例如/home/flinkuser/keytab)。 2. 在“${FLINK_HOME}/conf/flink-conf.yaml”上配置： a. keytab路径。 security.kerberos.login.keytab: /home/flinkuser/keytab/user.keytab **说明** “/home/flinkuser/keytab/”表示的是用户保存keytab的目录。 b. principal名(即开发用户名)。 security.kerberos.login.principal:flinkuser c. 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。配置如下： zookeeper.sasl.disable: false security.kerberos.login.contexts: Client d. 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
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

Flink整个系统有三种加密传输方式：

- 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：

步骤1 打开SSL开关和设置SSL加密算法，配置参数如表2所示，请根据实际情况修改对应参数值。

表 13-5 参数描述

参数	参数值示例	描述
security.ssl.internal.enabled	true	打开内部SSL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。
security.ssl.algorithms	TLS_RSA_WITH_AES_128_CBC_SHA256	设置SSL加密的算法。

如下参数MRS的Flink默认配置没有，用户可以根据需要进行添加，外部连接开启SSL，YARN的代理无法访问Flink页面。这是由于YARN不支持https代理。配置文件中包含认证密码信息可能存在安全风险，建议当前场景执行完毕后删除相关配置文件或加强安全管理。

参数	参数值示例	描述
security.ssl.rest.enabled	true	打开外部SSL开关。
security.ssl.rest.keystore	${path}/flink.keystore	keystore的存放路径。
security.ssl.rest.keystore-password	123456	keystore的password，“123456”表示需要用户输入自定义设置的密码值。
security.ssl.rest.key-password	123456	ssl key的password，“123456”表示需要用户输入自定义设置的密码值。
security.ssl.rest.truststore	${path}/flink.truststore	truststore存放路径。

参数	参数值示例	描述
security.ssl.rest.truststore-password	123456	truststore的password，“123456”表示需要用户输入自定义设置的密码值。

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。

步骤2 在Flink客户端的bin目录下，执行命令***sh generate_keystore.sh <password>***，**表13-6**中的配置项会被默认赋值，用户也可以手动配置。命令中如果携带认证密码信息可能存在安全风险，在执行命令前建议关闭系统的history命令记录功能，避免信息泄露。

表 13-6 参数描述

参数	参数值示例	描述
security.ssl.internal.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.internal.keystore-password	123456	keystore的password，“123456”表示需要用户输入自定义设置的密码值。
security.ssl.internal.key-password	123456	ssl key的password，“123456”表示需要用户输入自定义设置的密码值。
security.ssl.internal.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.internal.truststore-password	123456	truststore的password，“123456”表示需要用户输入自定义设置的密码值。

如果开启外部连接SSL，即 security.ssl.rest.enabled 配置为 true，则如下参数用户需要配置

参数	参数值示例	描述
security.ssl.rest.keystore	${path}/flink.keystore	keystore的存放路径。
security.ssl.rest.keystore-password	123456	keystore的password，“123456”表示需要用户输入自定义设置的密码值。
security.ssl.rest.key-password	123456	ssl key的password，“123456”表示需要用户输入自定义设置的密码值。
security.ssl.rest.truststore	${path}/flink.truststore	truststore存放路径。
security.ssl.rest.truststore-password	123456	truststore的password，“123456”表示需要用户输入自定义设置的密码值。

“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，详细说明如下。

说明

- 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
  ```
  security.ssl.internal.keystore: ssl/flink.keystore
  security.ssl.internal.truststore: ssl/flink.truststore
  ```
- 配置keystore或truststore文件路径为**绝对路径**时，需要在Flink Client以及各个节点的该绝对路径上放置keystore或truststore文件。
  ```
  security.ssl.internal.keystore: /opt/client/Flink/flink/conf/flink.keystore
  security.ssl.internal.truststore: /opt/client/Flink/flink/conf/flink.truststore
  ```

- 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。如：
    ```
    ./bin/yarn-session.sh -t ssl/ -n 2
    ```
  - 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。如：
    ```
    ./bin/flink run -yt ssl/ -ys 3 -yn 3 -m yarn-cluster -c com.huawei.SocketWindowWordCount lib/flink-eg-1.0.jar --hostname r3-d3 --port 9000
    ```

    说明

    - 在举例当中的“ssl/”是Flink Client端目录下的子目录，该目录是用来存放SSL keystore、truststore相关配置文件。
    - Flink Client执行命令的当前路径需要能访问到“ssl/”相对路径。

- 配置keystore或truststore文件路径为绝对路径时，需要在Flink Client以及各个节点的该绝对路径上放置keystore和truststore文件，并且提交作业的用户需要具有读取keystore和truststore文件的权限。

  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 -n 2
  ```
  – 使用Flink run命令执行应用程序。如：
  ```
  ./bin/flink run  -ys 3 -yn 3 -m yarn-cluster -c com.huawei.SocketWindowWordCount lib/flink-eg-1.0.jar --hostname r3-d3 --port 9000
  ```

----结束

13.3 开发程序

13.3.1 DataStream 程序

13.3.1.1 场景说明

假定用户有某个网站网民周末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。

```
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

DataStream样例工程的数据存储在文本中。

将log1.txt和log2.txt文件放置在用户开发程序的某路径下，例如"/opt/log1.txt"和"/opt/log2.txt"。

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. 筛选上网时间超过阈值的用户，并获取结果。

13.3.1.2 Java 样例代码

功能简介

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，具体代码参见com.huawei.flink.example.stream.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.flink.examples.stream.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 start execution, and use the value to check if the
window elements are 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
```

13.3.1.3 Scala 样例代码

功能简介

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。

样例代码

下面代码片段仅为演示，具体代码参见com.huawei.flink.example.stream.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 start execution, and use the value to check if the window
elements are 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)
```

13.3.2 向 Kafka 生产并消费数据程序

13.3.2.1 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。Flink向Kafka组件发送数据（需要有kafka权限用户），并从Kafka组件获取数据。

步骤1 确保集群安装完成，包括HDFS、Yarn、Flink和Kafka。

步骤2 创建Topic。

1. 在服务端配置用户创建topic的权限。

   开启Kerberos认证的安全集群将Kafka的Broker配置参数“allow.everyone.if.no.acl.found”的值修改为“true”。配置完后重启kafka服务。未开启Kerberos认证的普通集群无需此配置。

2. 用户使用Linux命令创建topic，如果是安全集群，用户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**准备开发用户**。

   创建topic的命令格式：

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

   表 13-7 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，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/kafka --partitions 5 --replication-factor 1 --topic topic1
```

步骤3 如果集群开启了kerberos， 执行该步骤进行安全认证，否则跳过该步骤。

- **Kerberos认证配置**

  a. 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-1.2.1/keytab/admin.keytab
security.kerberos.login.principal: admin
security.kerberos.login.contexts: Client,KafkaClient
security.kerberos.login.use-ticket-cache: false
```

  b. 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 //10.96.101.32:21007表示kafka服务器的IP:port
```

----结束

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。

13.3.2.2 Java 样例代码

功能简介

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

用户在开发前需要使用对接安全模式的Kafka，则需要引入MRS的kafka-client-xx.x.x.jar，该jar包可在MRS client目录下获取。

样例代码

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.flink.example.kafka.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 10.91.8.218:21005");
      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 FlinkKafkaProducer010<>(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.flink.example.kafka.ReadFromKafka" +
          " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21005");
      System.out.println("./bin/flink run --class com.huawei.flink.example.kafka.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
FlinkKafkaConsumer010<>(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();
    }
  }
```

13.3.2.3 Scala 样例代码

功能简介

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

用户在开发前需要使用对接安全模式的Kafka，则需要引入MRS的kafka-client-xx.x.x.jar，该jar包可在MRS client目录下获取。

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.flink.example.kafka.ReadFromKafka

```
//producer代码
object WriteIntoKafkaScala {
  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.flink.example.kafka.WriteIntoKafkaScala" +

      " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21005")

    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 SimpleStringGeneratorScala)

    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 SimpleStringGeneratorScala 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 ReadFromKafkaScala {
  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.flink.example.kafka.ReadFromKafkaScala" +

      " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21005")

    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()

 }

}
```

13.3.3 异步 Checkpoint 机制程序

13.3.3.1 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。

13.3.3.2 Java 样例代码

代码样例

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。

- 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;
    }
}
```

- 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.SourceFunction;

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

public class SimpleSourceWithCheckPoint implements SourceFunction<Tuple4<Long, String, String,
Integer>>, ListCheckpointed<UDFState> {

    private long count = 0;
    private boolean isRunning = true;
    private String alphabet = "justtest";

    @Override
    public List<UDFState> snapshotState(long l, long l1) throws Exception
    {
        UDFState udfState = new UDFState();
        List<UDFState> udfStateList = new ArrayList<UDFState>();
        udfState.setCount(count);
        udfStateList.add(udfState);
        return udfStateList;
    }

    @Override
    public void restoreState(List<UDFState> list) throws Exception
    {
        UDFState udfState = list.get(0);
        count = udfState.getCount();
    }

    @Override
    public void run(SourceContext<Tuple4<Long, String, String, Integer>> sourceContext) throws
Exception
    {
        Random random = new Random();
        while (isRunning) {
            for (int i = 0; i < 10000; i++) {
                sourceContext.collect(Tuple4.of(random.nextLong(), "hello" + count, alphabet, 1));
                count ++;
            }
            Thread.sleep(1000);
        }
    }

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

- 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;

public class WindowStatisticWithChk implements WindowFunction<Tuple4<Long, String, String,
Integer>, Long, Tuple, TimeWindow>, ListCheckpointed<UDFState> {

    private long total = 0;

    @Override
    public List<UDFState> snapshotState(long l, long l1) throws Exception
    {
        UDFState udfState = new UDFState();
        List<UDFState> list = new ArrayList<UDFState>();
        udfState.setCount(total);
        list.add(udfState);
        return list;
    }

    @Override
    public void restoreState(List<UDFState> list) throws Exception
    {
        UDFState udfState = list.get(0);
        total = udfState.getCount();
    }

    @Override
    public void apply(Tuple tuple, TimeWindow timeWindow, Iterable<Tuple4<Long, String, String,
Integer>> iterable, Collector<Long> collector) throws Exception
    {
        long count = 0L;
        for (Tuple4<Long, String, String, Integer> tuple4 : iterable) {
            count ++;
        }
        total += count;
        collector.collect(total);
    }
}
```

- 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.api.java.utils.ParameterTool;
import org.apache.flink.runtime.state.StateBackend;
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
    {
        String chkPath = ParameterTool.fromArgs(args).get("chkPath", "hdfs://hacluster/flink/
checkpoints/");
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        env.setStateBackend((StateBackend) new FsStateBackend((chkPath)));
        env.enableCheckpointing(6000, CheckpointingMode.EXACTLY_ONCE);
        env.addSource(new SimpleSourceWithCheckPoint())
                .keyBy(0)
                .window(SlidingProcessingTimeWindows.of(Time.seconds(4), Time.seconds(1)))
                .apply(new WindowStatisticWithChk())
```

```
            .print();

        env.execute();
    }
}
```

13.3.3.3 Scala 样例代码

代码样例

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。

- 发送数据形式

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

- 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 UDFStateScala 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
}
```

- 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

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

// 该类是带有checkpoint的source算子
class SEventSourceWithChk extends RichSourceFunction[SEvent] with
ListCheckpointed[UDFStateScala]{
  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[UDFStateScala] = {
    val udfList: util.ArrayList[UDFStateScala] = new util.ArrayList[UDFStateScala]
    val udfState = new UDFStateScala
    udfState.setState(count)
    udfList.add(udfState)
    udfList
  }

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

- 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[UDFStateScala]{
  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[UDFStateScala] = {
    val udfList: util.ArrayList[UDFStateScala] = new util.ArrayList[UDFStateScala]
    val udfState = new UDFStateScala
    udfState.setState(total)
    udfList.add(udfState)
    udfList
  }

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

- 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 org.apache.flink.runtime.state.filesystem.FsStateBackend
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
```

```
object FlinkEventTimeAPIChkMain {
  def main(args: Array[String]): Unit ={
    val chkPath = ParameterTool.fromArgs(args).get("chkPath", "hdfs://hacluster/flink/checkpoint/
checkpoint/")
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStateBackend(new FsStateBackend(chkPath))
    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()
  }
}
```

13.3.4 Stream SQL Join 程序

13.3.4.1 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监听的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socket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。

13.3.4.2 Java 样例代码

功能简介

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

用户在开发前需要使用对接安全模式的Kafka，则需要引入MRS的kafka-client-xx.x.x.jar，该jar包可在MRS client目录下获取。

代码样例

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

- 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
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;

import java.util.Random;

public class WriteIntoKafka4SQLJoin {

    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.flink.example.sqljoin.WriteIntoKafka4SQLJoin" +
                " /opt/test.jar --topic topic-test -bootstrap.servers 10.91.8.218:21005");
        System.out.println("./bin/flink run --class
com.huawei.flink.example.sqljoin.WriteIntoKafka4SQLJoin" +
                " /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 producer = new FlinkKafkaProducer<>(paraTool.get("topic"), new
SimpleStringSchema(), paraTool.getProperties());
        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;
        }
    }
}
```

- 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。

```
import org.apache.calcite.interpreter.Row;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.TableEnvironment;
import org.apache.flink.table.api.java.StreamTableEnvironment;

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.flink.example.sqljoin.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.flink.example.sqljoin.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;
        }

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        StreamTableEnvironment tableEnv = TableEnvironment.getTableEnvironment(env);

        //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();
```

```
    }
}
```

13.4 调测程序

13.4.1 编译并运行程序

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

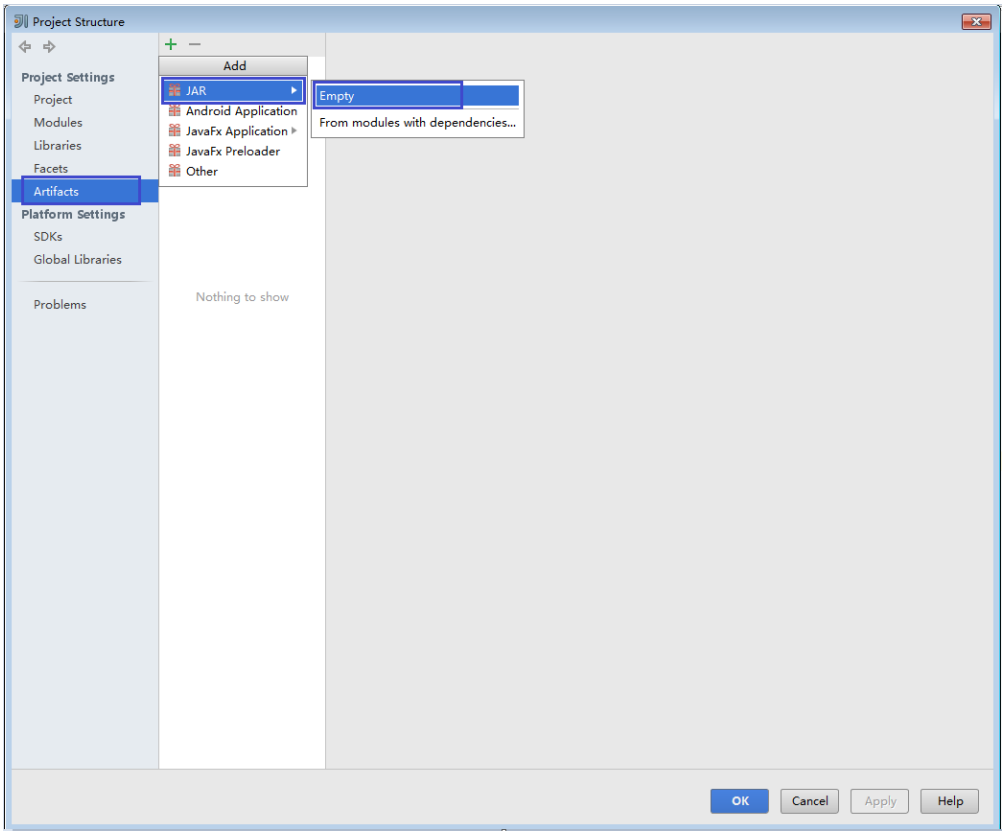
基于YARN集群的Flink应用程序不支持在Windows环境下运行，只支持在Linux环境下运行。

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。

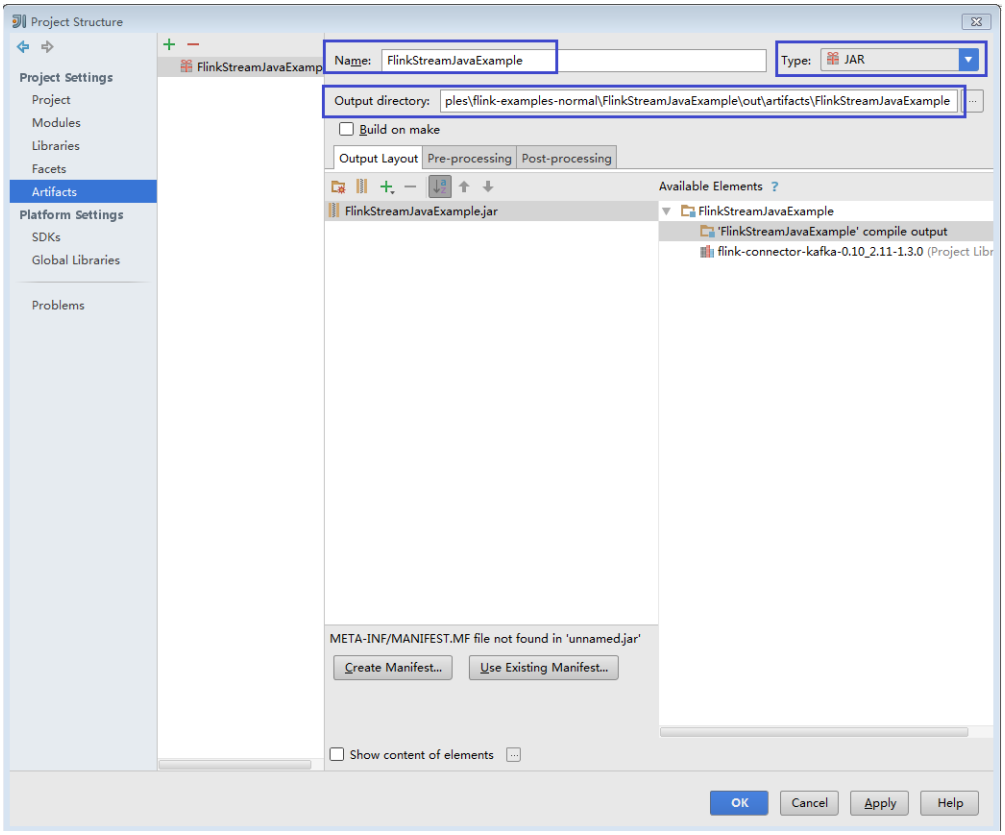
1. 在IDEA主页面，选择“File > Project Structures...”进入“Project Structure”页面。
2. 在“Project Structure”页面，选择“Artifacts”，单击“+”并选择“JAR > Empty”。

图 13-32 添加 Artifacts



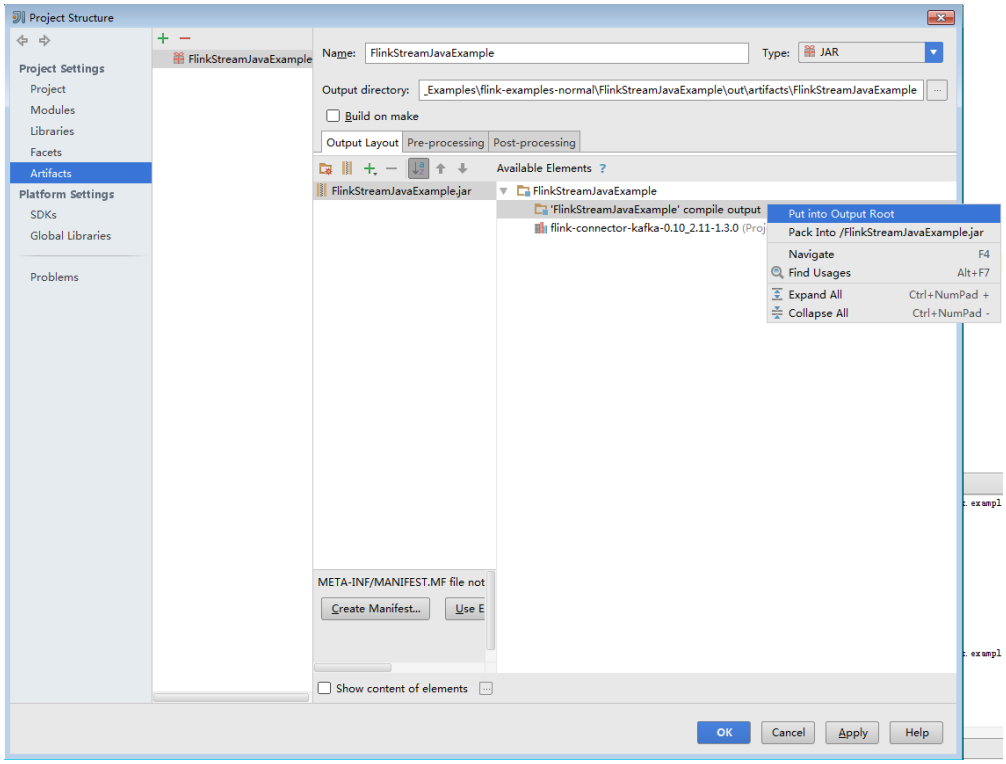
3. 您可以根据实际情况设置Jar包的名称、类型以及输出路径。

图 13-33 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 13-34 Put into Output Root

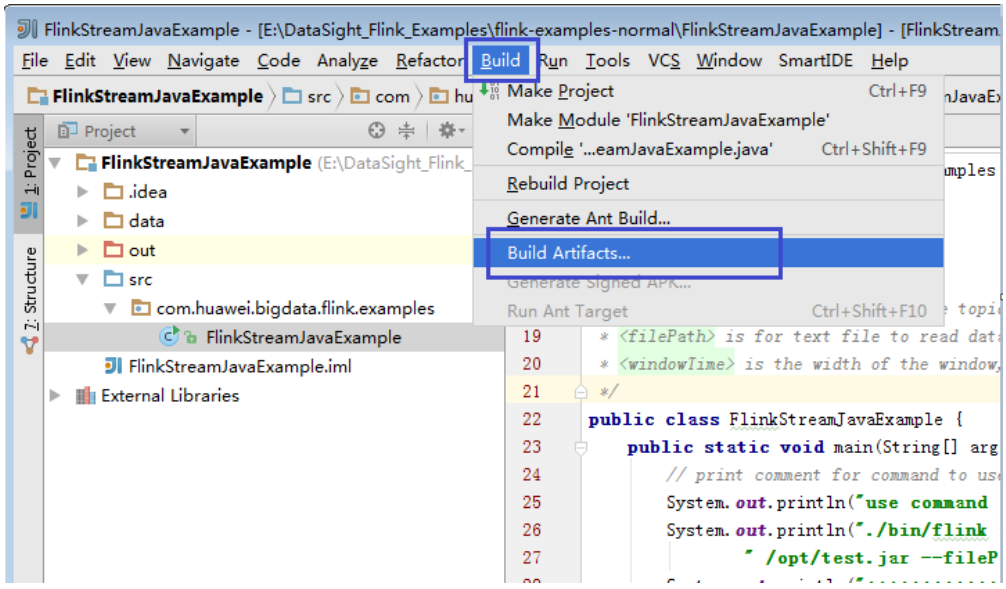


5. 最后单击“OK”完成配置。

步骤2 生成Jar包。

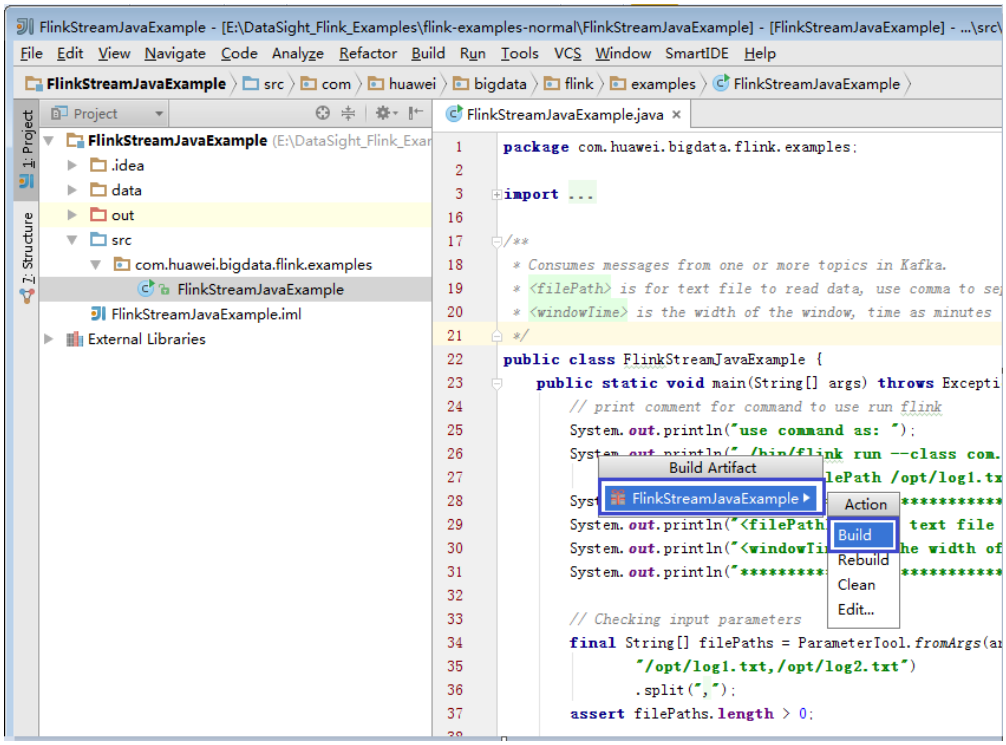
1. 在IDEA主页面，选择“Build > Build Artifacts...”。

图 13-35 Build Artifacts



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

图 13-36 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）拷贝到Linux环境的Flink运行环境下（即Flink客户端），如“/opt/Flink_test”。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 -n 3 -jm 1024 -tm 1024
```

说明

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。

- 运行DataStream样例程序（Scala和Java语言）。

  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，例如：

  ```
  bin/flink run --class com.huawei.flink.example.stream.FlinkStreamJavaExample /opt/Flink_test/flink-examples-1.0.jar --filePath /opt/log1.txt,/opt/log2.txt --windowTime 2

  bin/flink run --class com.huawei.flink.example.stream.FlinkStreamScalaExample /opt/Flink_test/flink-examples-1.0.jar --filePath /opt/log1.txt,/opt/log2.txt --windowTime 2
  ```

表 13-8 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并使用**chmod 755 *文件名***命令为用户赋予读、写、执行权限，而属组用户和其他用户只有读、执行权限。可以默认，也可以自行设置。
<windowTime>	指窗口时间大小，以分钟为单位。可以默认，也可以自行设置。

- 运行向Kafka生产并消费数据样例程序（Scala和Java语言）。

  执行命令启动程序生产数据。

  ```
  bin/flink run --class com.huawei.flink.example.kafka.WriteIntoKafka /opt/Flink_test/flink-examples-1.0.jar <topic> <bootstrap.servers> [security.protocol] [sasl.kerberos.service.name] [kerberos.domain.name] [ssl.truststore.location] [ssl.truststore.password]
  ```

  执行命令启动程序消费数据。命令中如果携带认证密码信息可能存在安全风险，在执行命令前建议关闭系统的history命令记录功能，避免信息泄露。

  ```
  bin/flink run --class com.huawei.flink.example.kafka.ReadFromKafka /opt/Flink_test/flink-examples-1.0.jar <topic> <bootstrap.servers> [security.protocol] [sasl.kerberos.service.name] [kerberos.domain.name] [ssl.truststore.location] [ssl.truststore.password]
  ```

  表 13-9 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MRS 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。
kerberos.domain.name	Kafka Domain名称。	否 **说明** security.protocol配置了SASL时必须配置。

四种类型实际命令示例，以ReadFromKafka为例如下：

```
bin/flink run --class com.huawei.flink.example.kafka.ReadFromKafka /opt/Flink_test/flink-examples-1.0.jar --topic topic1 --bootstrap.servers 10.96.101.32:21005
bin/flink run --class com.huawei.flink.example.kafka.ReadFromKafka /opt/Flink_test/flink-examples-1.0.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
bin/flink run --class com.huawei.flink.example.kafka.ReadFromKafka /opt/Flink_test/flink-examples-1.0.jar --topic topic1 --bootstrap.servers 10.96.101.32:21008 --security.protocol SSL --ssl.truststore.location /home/truststore.jks --ssl.truststore.password xxx
bin/flink run --class com.huawei.flink.example.kafka.ReadFromKafka /opt/Flink_test/flink-examples-1.0.jar --topic topic1 --bootstrap.servers 10.96.101.32:21009 --security.protocol SASL_SSL --sasl.kerberos.service.name kafka --kerberos.domain.name hadoop.hadoop.com --ssl.truststore.location /home/truststore.jks --ssl.truststore.password xxx
```

- 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.flink.example.checkpoint.FlinkProcessingTimeAPIChkMain /opt/Flink_test/flink-examples-1.0.jar --chkPath hdfs://hacluster/flink-checkpoint/
      ```

    - Scala

      ```
      bin/flink run --class com.huawei.flink.example.checkpoint.FlinkEventTimeAPIChkMain /opt/Flink_test/flink-examples-1.0.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.flink.example.checkpoint.FlinkProcessingTimeAPIChkMain /opt/Flink_test/flink-examples-1.0.jar --chkPath file:///home/zzz/flink-checkpoint/
      ```

    - Scala

      ```
      bin/flink run --class com.huawei.flink.example.checkpoint.FlinkEventTimeAPIChkMain /opt/Flink_test/flink-examples-1.0.jar --chkPath file:///home/zzz/flink-checkpoint/
      ```

    说明

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

      其中，flink/checkpoint/checkpoint表示指定的根目录。

      fd5f5b3d08628d83038a30302b611表示以jobID命名的第二次目录。

      chk-X中"X"为checkpoint编号，第三层目录。

      4f854bf4-ea54-4595-a9d9-9b9080779ffe表示checkpoint源文件。

    - Flink在集群模式下checkpoint将文件放到HDFS。

- 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（Java...**

  ```
  bin/flink run --class com.huawei.flink.example.sqljoin.WriteIntoKafka4SQLJoin /opt/Flink_test/flink-examples-1.0.jar --topic topic-test --bootstrap.servers xxx.xxx.xxx.xxx:21005
  ```

  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.flink.example.sqljoin.SqlJoinWithSocket /opt/Flink_test/flink-examples-1.0.jar --topic topic-test --bootstrap.servers xxx.xxx.xxx.xxx:21005 --hostname xxx.xxx.xxx.xxx --port 9000
```

----结束

13.4.2 查看调测结果

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应用程序指定途径，您可以通过指定文件或其他途径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，用户可以通过Flink的WebUI查看“task manager”标签下的out按钮查看。
  - 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Stream SQL Join结果**

  结果在flink的“taskmanager.out”文件中，用户可以通过Flink的WebUI查看“task manager”标签下的out按钮查看。
- **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**

  有两种方式获取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。

13.5 性能调优

配置内存

Flink是依赖内存计算，计算过程中内存不够对Flink的执行效率影响很大。可以通过监控GC（Garbage Collection），评估内存使用及剩余情况来判断内存是否变成性能瓶颈，并根据情况优化。

监控节点进程的YARN的Container GC日志，如果频繁出现Full GC，需要优化GC。

说明

GC的配置：在客户端的“conf/flink-conf.yaml”配置文件中，在“env.java.opts”配置项中添加参数：“-Xloggc:<LOG_DIR>/gc.log -XX:+PrintGCDetails -XX:-OmitStackTraceInFastThrow -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=20 -XX:GCLogFileSize=20M”。 此处默认已经添加GC日志。

- 优化GC。

  调整老年代和新生代的比值。在客户端的“conf/flink-conf.yaml”配置文件中，在“env.java.opts”配置项中添加参数：“-XX:NewRatio”。如“-XX:NewRatio=2”，则表示老年代与新生代的比值为2:1，新生代占整个堆空间的1/3，老年代占2/3。

- 开发Flink应用程序时，优化DataStream的数据分区或分组操作。
  - 当分区导致数据倾斜时，需要考虑优化分区。
  - 避免非并行度操作，有些对DataStream的操作会导致无法并行，例如WindowAll。
  - keyBy尽量不要使用String。

设置并行度

并行度控制任务的数量，影响操作后数据被切分成的块数。调整并行度让任务的数量和每个任务处理的数据与机器的处理能力达到最优。

查看CPU使用情况和内存占用情况，当任务和数据不是平均分布在各节点，而是集中在个别节点时，可以增大并行度使任务和数据更均匀的分布在各个节点。增加任务的并行度，充分利用集群机器的计算能力。

任务的并行度可以通过以下四种层次（按优先级从高到低排列）指定，用户可以根据实际的内存、CPU、数据以及应用程序逻辑的情况调整并行度参数。

- 算子层次

  一个算子、数据源和sink的并行度可以通过调用setParallelism()方法来指定，例如

  ```
  final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
  DataStream<String> text = [...]
  DataStream<Tuple2<String, Integer>> wordCounts = text
      .flatMap(new LineSplitter())
      .keyBy(0)
      .timeWindow(Time.seconds(5))
      .sum(1).setParallelism(5);
  wordCounts.print();
  env.execute("Word Count Example");
  ```

- 执行环境层次

Flink程序运行在执行环境中。执行环境为所有执行的算子、数据源、data sink定义了一个默认的并行度。

执行环境的默认并行度可以通过调用setParallelism()方法指定。例如：

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(3);
DataStream<String> text = [...]
DataStream<Tuple2<String, Integer>> wordCounts = [...]
wordCounts.print();
env.execute("Word Count Example");
```

- 客户端层次

  并行度可以在客户端将job提交到Flink时设定。对于CLI客户端，可以通过“-p”参数指定并行度。例如：

  ```
  ./bin/flink run -p 10 ../examples/*WordCount-java*.jar
  ```

- 系统层次

  在系统级可以通过修改Flink客户端conf目录下“flink-conf.yaml”文件中的“parallelism.default”配置选项来指定所有执行环境的默认并行度。

配置进程参数

Flink on YARN模式下，有JobManager和TaskManager两种进程。在任务调度和运行的过程中，JobManager和TaskManager承担了很大的责任。

因而JobManager和TaskManager的参数配置对Flink应用的执行有着很大的影响意义。用户可通过如下操作对Flink集群性能做优化。

步骤1 配置JobManager内存。

JobManager负责任务的调度，以及TaskManager、RM之间的消息通信。当任务数变多，任务平行度增大时，JobManager内存都需要相应增大。

您可以根据实际任务数量的多少，为JobManager设置一个合适的内存。

- 在使用yarn-session命令时，添加“-jm MEM”参数设置内存。
- 在使用yarn-cluster命令时，添加“-yjm MEM”参数设置内存。

步骤2 配置TaskManager个数。

每个TaskManager每个核同时能跑一个task，所以增加了TaskManager的个数相当于增大了任务的并发度。在资源充足的情况下，可以相应增加TaskManager的个数，以提高运行效率。

- 在使用yarn-session命令时，添加“-n NUM”参数设置TaskManager个数。
- 在使用yarn-cluster命令时，添加“-yn NUM”参数设置TaskManager个数。

步骤3 配置TaskManager Slot数。

每个TaskManager多个核同时能跑多个task，相当于增大了任务的并发度。但是由于所有核共用TaskManager的内存，所以要在内存和核数之间做好平衡。

- 在使用yarn-session命令时，添加“-s NUM”参数设置SLOT数。
- 在使用yarn-cluster命令时，添加“-ys NUM”参数设置SLOT数。

步骤4 配置TaskManager内存。

TaskManager的内存主要用于任务执行、通信等。当一个任务很大的时候，可能需要较多资源，因而内存也可以做相应的增加。

- 将在使用yarn-session命令时，添加“-tm MEM”参数设置内存。
- 将在使用yarn-cluster命令时，添加“-ytm MEM”参数设置内存。

----结束

设计分区方法

合理的设计分区依据，可以优化task的切分。在程序编写过程中要尽量分区均匀，这样可以实现每个task数据不倾斜，防止由于某个task的执行时间过长导致整个任务执行缓慢。

以下是几种分区方法。

- **随机分区：**将元素随机的进行分区。
  ```
  dataStream.shuffle();
  ```
- **Rebalancing (Round-robin partitioning)：**基于round-robin对元素进行分区，使得每个分区负载均衡。对于存在数据倾斜的性能优化是很有用的。
  ```
  dataStream.rebalance();
  ```
- **Rescaling：**以round-robin的形式将元素分区到下游操作的子集中。如果你想要将数据从一个源的每个并行实例中散发到一些mappers的子集中，用来分散负载，但是又不想要完全的rebalance 介入（引入`rebalance()`），这会非常有用。
  ```
  dataStream.rescale();
  ```
- **广播：**广播每个元素到所有分区。
  ```
  dataStream.broadcast();
  ```
- **自定义分区：**使用一个用户自定义的Partitioner对每一个元素选择目标task，由于用户对自己的数据更加熟悉，可以按照某个特征进行分区，从而优化任务执行。

  简单示例如下所示：

  ```
  // fromElements构造简单的Tuple2流
  DataStream<Tuple2<String, Integer>> dataStream = env.fromElements(Tuple2.of("hello",1),
  Tuple2.of("test",2), Tuple2.of("world",100));
  // 定义用于分区的key值，返回即属于哪个partition的，该值加1就是对应的子任务的id号
  Partitioner<Tuple2<String, Integer>> strPartitioner = new Partitioner<Tuple2<String, Integer>>() {
      @Override
      public int partition(Tuple2<String, Integer> key, int numPartitions) {
          return (key.f0.length() + key.f1) % numPartitions;
      }
  };
  // 使用Tuple2进行分区的key值
  dataStream.partitionCustom(strPartitioner, new KeySelector<Tuple2<String, Integer>, Tuple2<String,
  Integer>>() {
      @Override
      public Tuple2<String, Integer> getKey(Tuple2<String, Integer> value) throws Exception {
          return value;
      }
  }).print();
  ```

配置 netty 网络通信

Flink通信主要依赖netty网络，所以在Flink应用执行过程中，netty的设置尤为重要，网络通信的好坏直接决定着数据交换的速度以及任务执行的效率。

以下配置均可在客户端的“conf/flink-conf.yaml”配置文件中进行修改适配，默认已经是相对较优解，请谨慎修改，防止性能下降。

- “taskmanager.network.netty.num-arenas”： 默认是“taskmanager.numberOfTaskSlots”，表示netty的域的数量。
- “taskmanager.network.netty.server.numThreads”和“taskmanager.network.netty.client.numThreads”： 默认是

“taskmanager.numberOfTaskSlots”，表示netty的客户端和服务端的线程数目设置。

- “taskmanager.network.netty.client.connectTimeoutSec”：默认是120s，表示taskmanager的客户端连接超时的时间。
- “taskmanager.network.netty.sendReceiveBufferSize”：默认是系统缓冲区大小(cat /proc/sys/net/ipv4/tcp_[rw]mem)，一般为4MB，表示netty的发送和接收的缓冲区大小。
- “taskmanager.network.netty.transport”：默认为“nio”方式，表示netty的传输方式，有“nio”和“epoll”两种方式。

经验总结

数据倾斜

当数据发生倾斜（某一部分数据量特别大），虽然没有GC（Gabage Collection，垃圾回收），但是task执行时间严重不一致。

- 需要重新设计key，以更小粒度的key使得task大小合理化。
- 修改并行度。
- 调用rebalance操作，使数据分区均匀。

缓冲区超时设置

- 由于task在执行过程中存在数据通过网络进行交换，数据在不同服务器之间传递的缓冲区超时时间可以通过setBufferTimeout进行设置。
- 当设置“setBufferTimeout(-1)”，会等待缓冲区满之后才会刷新，使其达到最大吞吐量；当设置“setBufferTimeout(0)”时，可以最小化延迟，数据一旦接收到就会刷新；当设置“setBufferTimeout”大于0时，缓冲区会在该时间之后超时，然后进行缓冲区的刷新。

  示例可以参考如下：

  ```
  env.setBufferTimeout(timeoutMillis);
  env.generateSequence(1,10).map(new MyMapper()).setBufferTimeout(timeoutMillis);
  ```

13.6 更多信息

13.6.1 Savepoints CLI 介绍

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。

13.6.2 Flink Client CLI 介绍

Flink CLI详细的使用方法参考官网描述：**https://ci.apache.org/projects/flink/flink-docs-release-1.7/ops/cli.html**。

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 -n 3
     ```
   - 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。

     - 以单机模式运行作业的一个示例如下：
       ```
       bin/flink run -m yarn-cluster -yn 2 examples/streaming/WindowJoin.jar
       ```

       说明

       通过参数-m yarn-cluster使作业以单机模式运行，-yn表示TaskManager的数量。
   - 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 -n 3 -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 -n 3
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。

13.7 FAQ

13.7.1 Savepoints 相关问题解决方案

1. 用户必须为job中的所有算子均分配ID吗?

   严格的说，用户只给有状态的算子分配IDs即可，因为在savepoint中仅包括有状态的算子的状态，没有状态的算子并不包含在savepoint中。

   在实际应用中，强烈建议用户给所有的算子均分配ID，因为有些Flink的内置算子，如window算子是有状态的。具体哪个算子是有状态的，哪个算子是无状态的，不是十分明显。如果用户十分确定某个算子是无状态的，该算子可以不调用uid()方法分配ID。

2. 如果用户在升级作业时新添加一个有状态的算子有什么影响?

   当用户在作业中新添加一个有状态的算子时，由于该算子是新添加的，无保存的旧状态，因此无状态恢复，从0开始运行。

3. 如果用户在升级作业时从作业中删除一个有状态的算子有什么影响?

   默认情况下，savepoint会尝试将所有保存的状态恢复。如果用户使用的savepoint中包含已经删除算子的状态，恢复将会失败。

   用户可以通过--allowNonRestoredState(简写为-n)参数跳过恢复已经删除的算子的状态：

   ```
   $ bin/flink run -s savepointPath -n [runArgs]
   ```

4. 如果用户重新编排有状态的算子的顺序有什么影响?
   - 如果用户已经给这些算子分配IDs，那么这些状态会正常恢复。
   - 如果用户没有给这些算子分配IDs， 这些算子将会按新的顺序自动分配新的ID，这将导致状态恢复失败。

5. 如果用户在作业中删除或添加或更改无状态算子的顺序有什么影响?
   - 如果用户已经给有状态的算子分配ID，那么无状态的算子并不会影响从savepoint进行状态恢复。
   - 如果用户没有分配IDs，有状态算子的IDs由于顺序变化可能会被分配新的IDs，这将导致状态恢复失败。

6. 如果用户在状态恢复时改变了算子的并发度会有什么影响?

   如果Flink版本高于1.2.0且不使用已经废弃的状态API，如checkpointed，用户可以从savepoint中进行状态恢复。否则，无法恢复。

13.7.2 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?

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13-37**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13-38**所示。

图 13-37 时间监控信息

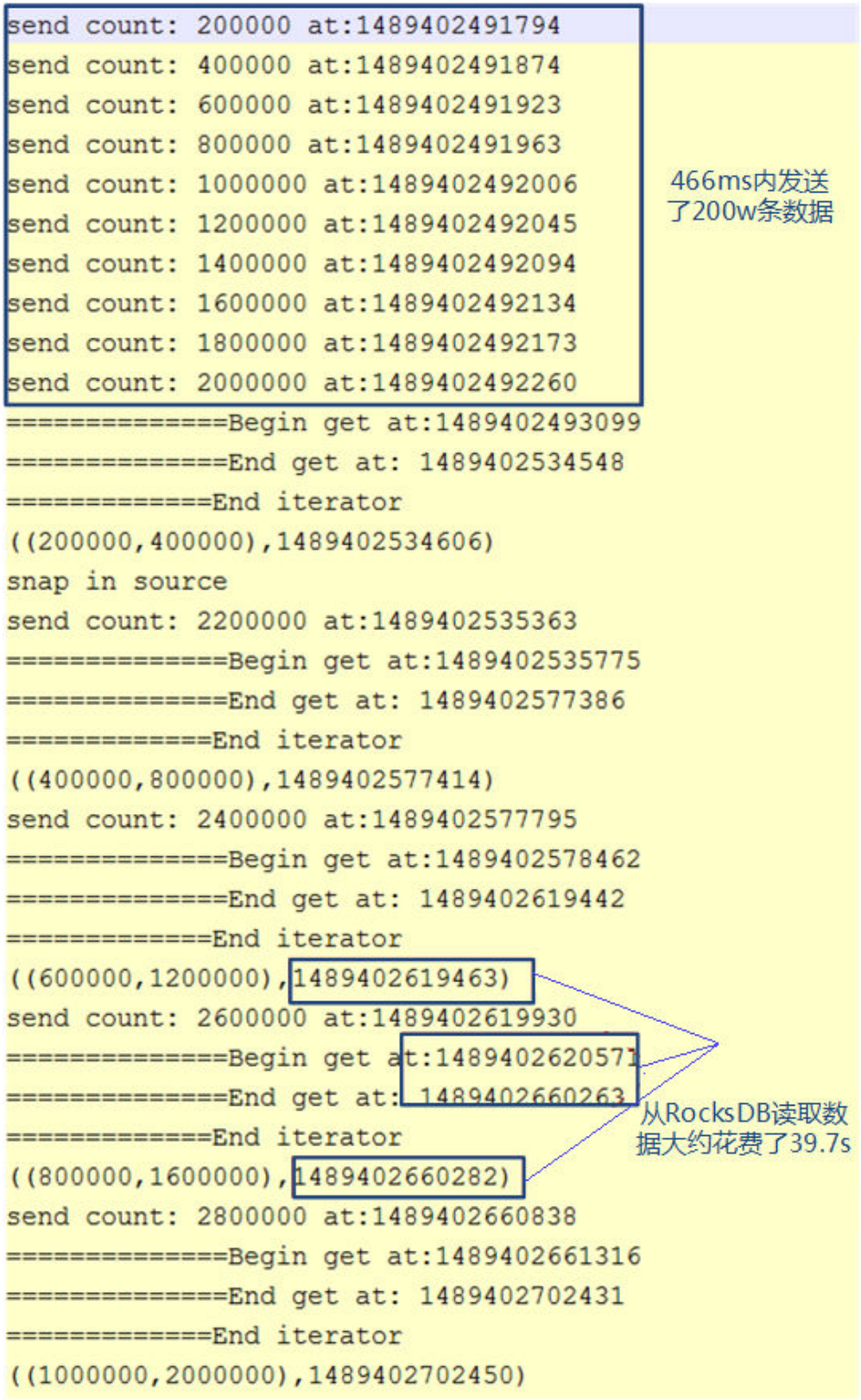
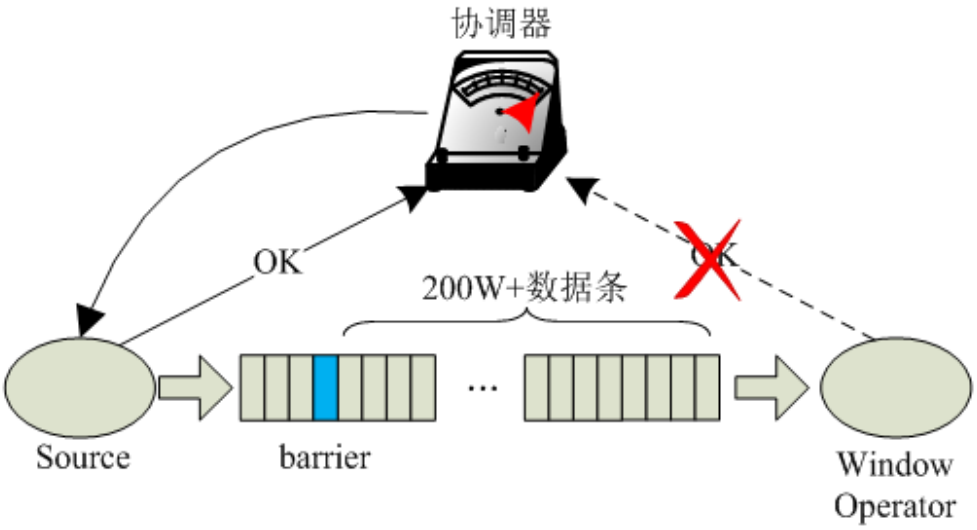


图 13-38 关系图



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/checkpoint/"));
```

13.7.3 如何处理 blob.storage.directory 配置/home 目录时，启动 yarn-session 失败的问题?

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

1. 建议将"blob.storage.directory"配置选项设置成“/tmp”或者“/opt/Bigdata/tmp”。
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MRS的admin用户为例。
   a. 修改Flink客户端配置文件conf/flink-conf.yaml，配置blob.storage.directory: /home/testdir/testdirdir/xxx。
   b. 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。

```
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。

c. 进入客户端路径，执行命令**./bin/yarn-session.sh -n 3 -jm 2048 -tm 3072**，可以看到yarn-session正常启动并且成功创建目录。

13.7.4 为什么非 static 的 KafkaPartitioner 类对象去构造 FlinkKafkaProducer010，运行时会报错?

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。

13.7.5 为什么新创建的 Flink 用户提交任务失败，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/application_1499222480199_0013
```

回答

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目录，导致新创建的用户访问不了ZooKeeper上的/flink目录。

新用户可以通过以下操作来解决问题。

1. 查看客户端的配置文件“conf/flink-conf.yaml”。
2. 修改配置项“high-availability.zookeeper.path.root”对应的ZooKeeper目录，例如：/flink2。
3. 重新提交任务。

13.7.6 为什么 Flink Web 页面无法直接连接?

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。用户可以通过以下步骤来解决问题。

1. 查看客户端的配置文件“conf/flink-conf.yaml”。
2. 确认配置项“jobmanager.web.ssl.enabled”的值是“false”。
   - 如果不是，请修改配置项的值为“false”。
   - 如果是，请执行3。
3. 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: 192.168.252.35,192.168.24.216
   jobmanager.web.allow-access-address: 192.168.252.35,192.168.24.216
   ```

14 Impala 应用开发

14.1 概述

14.1.1 应用开发简介

Impala 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。

14.1.2 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。

14.1.3 开发流程

开发流程中各阶段的说明如图14-1和表14-1所示。

图 14-1 Impala 应用程序开发流程

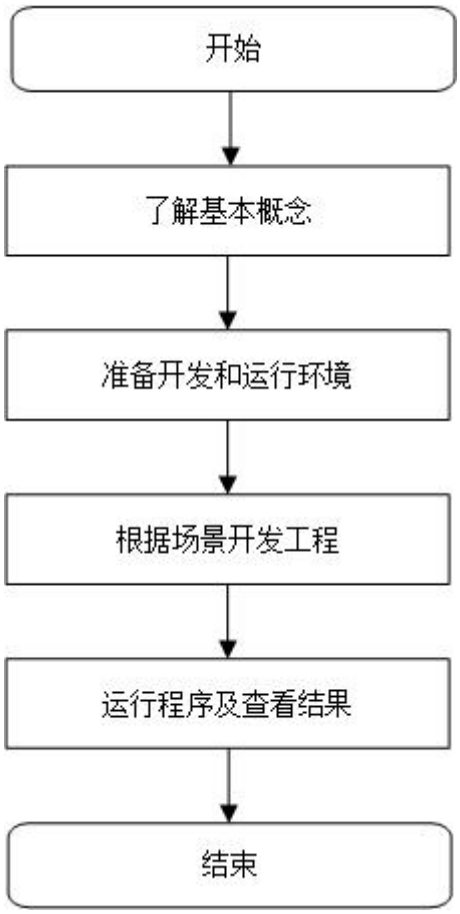


表 14-1 Impala 应用开发的流程说明

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。	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Eclipse工具，请根据指导完成不同语言的开发环境配置。	开发环境简介
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。	典型场景说明
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。	JDBC客户端运行及结果查看

14.2 环境准备

14.2.1 开发环境简介

在进行应用开发时，要准备的本地开发环境如表14-2所示。同时需要准备运行调测的环境，用于验证应用程序运行正常。

表 14-2 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：Linux系统。
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。

准备项	说明
安装和配置Eclipse	用于开发Impalad应用程序的工具。版本要求如下： • JDK使用1.7版本，Eclipse使用3.7.1及以上版本。 • JDK使用1.8版本，Eclipse使用4.3.2及以上版本。 说明： 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。
网络	确保客户端与Impala服务主机在网络上互通。

14.2.2 准备环境

- 选择Windows开发环境下，安装Eclipse，安装JDK。
  JDK使用1.8版本，Eclipse使用4.3.2及以上版本。

  说明

  - 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
  - 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
  - 若使用ODBC进行二次开发，请确保JDK版本为1.8及以上版本。
  - 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。
- 准备一个应用程序运行测试的Linux环境。

准备运行调测环境

步骤1 在弹性云服务器管理控制台，申请一个新的弹性云服务器，用于用户应用程序开发、运行、调测。

- 弹性云服务器的安全组需要和MRS集群Master节点的安全组相同。
- 弹性云服务器的VPC需要与MRS集群在同一个VPC中。
- 弹性云服务器的网卡需要与MRS集群在同一个网段中。

步骤2 申请弹性IP，绑定新申请的弹性云主机IP，并配置安全组出入规则。

步骤3 下载客户端程序。

1. 登录**MRS Manager**系统。
2. 选择“服务管理 > 下载客户端 ”，下载“完整客户端”到“远端主机”上，即下载客户端程序到新申请的弹性云服务器上。

步骤4 以root用户安装集群客户端。

1. 执行以下命令解压客户端包。

   tar -xvf /opt/MRS_Services_Client.tar

2. 执行以下命令校验安装文件包。

   sha256sum -c /opt/MRS_Services_ClientConfig.tar.sha256

   ```
   MRS_Services_ClientConfig.tar:OK
   ```

3. 执行以下命令解压安装文件包。

   tar -xvf /opt/MRS_Services_ClientConfig.tar

4. 执行如下命令安装客户端到指定目录（绝对路径），例如“/opt/client”。目录会自动创建。

   cd /opt/MRS_Services_ClientConfig

   sh install.sh /opt/client

   ```
   Components client installation is complete.
   ```

步骤5 执行以下命令，更新客户端配置。

sh /opt/client/refreshConfig.sh *客户端安装目录 客户端配置文件压缩包完整路径*

例如，执行命令

sh /opt/client/refreshConfig.sh /opt/client /opt/MRS_Services_Client.tar

说明

如果修改了组件的配置参数，需重新下载客户端配置文件并更新运行调测环境上的客户端。

----结束

14.2.3 准备开发用户

开发用户用于运行样例工程。用户需要有Impala权限，才能运行Impala样例工程。

前提条件

MRS服务集群开启了Kerberos认证时请执行该步骤，没有开启Kerberos认证的集群忽略该步骤。

操作步骤

步骤1 登录**MRS Manager**。

步骤2 单击“系统设置 > 用户管理 > 添加用户”，为样例工程创建一个用户。

步骤3 填写用户名，例如*impalauser*，用户类型为“机机”用户，加入用户组**impala**和**supergroup**，设置其“主组”为**supergroup**，单击“确定”，如图14-2所示。

图 14-2 添加用户

步骤4 在MRS Manager界面选择“系统设置 > 用户管理”，在用户名**impalauser**所在行的“操作”列选择“更多 > 下载认证凭据”。保存后解压得到用户的user.keytab文件与krb5.conf文件。用于在样例工程中进行安全认证。

----结束

参考信息

如果修改了组件的配置参数，需重新下载客户端配置文件并更新运行调测环境上的客户端。

14.2.4 准备 JDBC 客户端开发环境

为了运行Impala组件的JDBC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方式连接Impala服务的应用程序为例。

操作步骤

步骤1 在**样例工程获取地址**获取Impala示例工程。

步骤2 在Impala示例工程根目录，执行mvn install编译。

步骤3 在Impala示例工程根目录，执行mvn eclipse:eclipse创建Eclipse工程。

步骤4 在应用开发环境中，导入样例工程到Eclipse开发环境。

1. 选择“File > Import > General > Existing Projects into Workspace > Next >Browse”。

   显示“浏览文件夹”对话框。

2. 选择文件夹“impala-examples”。Windows下要求该文件夹的完整路径不包含空格。

   单击“Finish”。

   导入成功后，com.huawei.bigdata.impala.example包下的JDBCExample类，为JDBC接口样例代码。

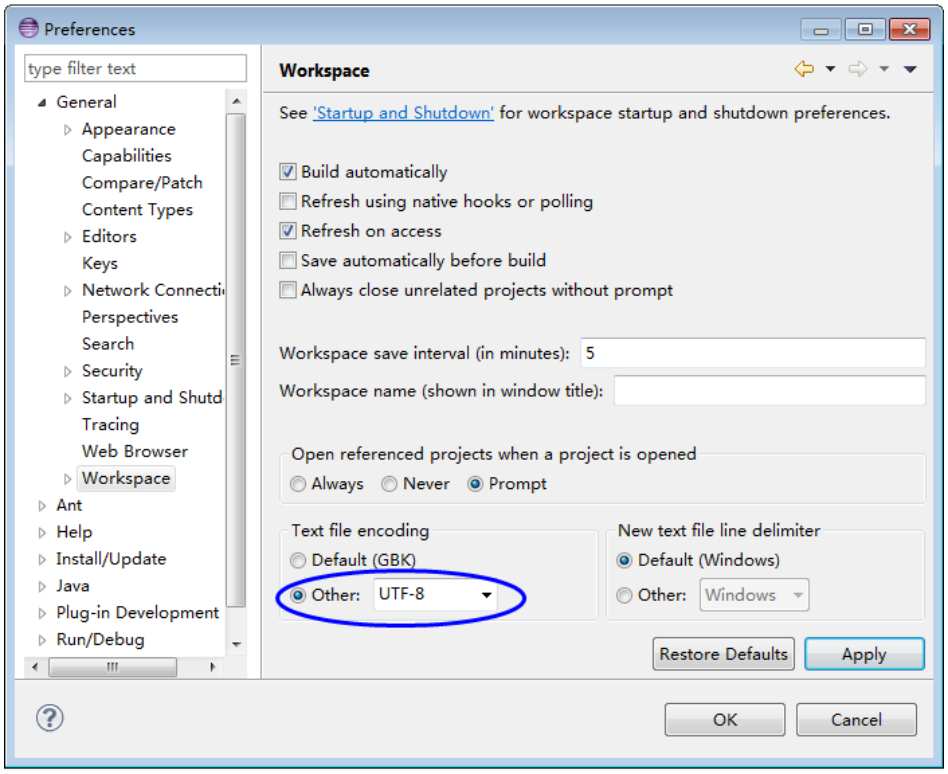
步骤5 设置Eclipse的文本文件编码格式，解决乱码显示问题。

1. 在Eclipse的菜单栏中，选择“Window > Preferences”。

   弹出“Preferences”窗口。

2. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如**图14-3**所示。

图 14-3 设置 Eclipse 的编码格式



步骤6 修改样例（未开启Kerberos认证集群可跳过此步骤）。

在步骤4获取新建开发用户的krb5.conf和user.keytab文件后，修改ExampleMain.java中的userName为对应的新建用户，例如impalauser。

```
/**
 * Other way to set conf for zk. If use this way,
 * can ignore the way in the 'login' method
 */
if (isSecurityMode) {
    userName = "impalauser";
    userKeytabFile = CONF_DIR + "user.keytab";
    krb5File = CONF_DIR + "krb5.conf";
    conf.set(HADOOP_SECURITY_AUTHENTICATION, "kerberos");
    conf.set(HADOOP_SECURITY_AUTHORIZATION, "true");
```

----结束

14.3 开发程序

14.3.1 典型场景说明

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14-3、表14-4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数据加载。

   雇员信息数据如表14-3所示。

表 14-3 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

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
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14-4**所示。

表 14-4 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**数据查询**。

- 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**样例程序指导**。

----结束

14.3.2 创建表

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（如Pig等）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MRS 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";
     ```

14.3.3 数据加载

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拷贝到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。

14.3.4 数据查询

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**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**用户自定义函数**。

14.3.5 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
```

14.3.6 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();
    }
  }
 }
```

14.4 调测程序

14.4.1 在 Windows 中调测程序

14.4.1.1 JDBC 客户端运行及结果查询

JDBC 客户端的命令行形式运行

步骤1 运行样例。

依照**准备JDBC客户端开发环境**中导入和修改样例后，并从集群Manager获取到keytab文件放置到样例工程的conf下（普通模式集群可忽略），即“impala-examples/conf”。即可在开发环境中（例如Eclipse中），右击“JDBCExample.java”，单击“Run as > Java Application”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。

- 申请一台windows的ECS访问MRS集群操作Impala。
- 使用本机访问MRS集群操作Impala。

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。申请ECS访问MRS集群的步骤如下。

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“弹性云服务器 > 快速入门 > 购买并登录Windows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。步骤如下。
   1. 在虚拟私有云管理控制台，申请一个弹性IP地址，并与弹性云服务器绑定。

      具体请参见“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. 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。请参见“虚拟私有云 > 用户指南 > 安全性 > 安全组 > 添加安全组规则”。

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）（由于普通集群未启用kerberos功能，可跳过此步骤）。

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

14.4.2 在 Linux 中调测程序

14.4.2.1 JDBC 客户端运行及结果查看

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”(Linux环境)，并在该目录下创建子目录“conf”。

步骤2 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 ，拷贝到“/opt/impala_examples”下

步骤3 开启Kerberos认证的安全集群下把从**步骤4**获取的user.keytab和krb5.conf拷贝到/opt/impala_examples/conf下。普通集群可跳过该步骤。

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

14.5 Impala 接口

14.5.1 JDBC

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。

14.5.2 Impala SQL

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**。

14.6 开发规范

14.6.1 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。

14.6.2 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。

14.6.3 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();
        }
    }
  }
}
```

15 Alluxio 应用开发

15.1 概述

15.1.1 应用开发简介

Alluxio 简介

Alluxio是一个面向基于云的数据分析和人工智能的开源的数据编排技术。它为数据驱动型应用和存储系统构建了桥梁, 将数据从存储层移动到距离数据驱动型应用更近的位置，从而能够更容易、更快得被访问。同时使得应用程序能够通过一个公共接口连接到许多存储系统。

Alluxio主要特点如下：

- 提供内存级I/O 吞吐率，同时降低具有弹性扩张特性的数据驱动型应用的成本开销
- 简化云存储和对象存储接入
- 简化数据管理，提供对多数据源的单点访问
- 应用程序部署简易

Alluxio 接口开发简介

Alluxio支持使用Java进行程序开发，具体的API接口内容请参考**https://docs.alluxio.io/os/javadoc/2.0/index.html**。

15.1.2 常用概念

Masters

由两个进程组成，一个是处理用户请求和管理Journal存储系统元数据的Alluxio Master，另一个是调度文件系统操作的Alluxio Job Master。

Workers

负责管理用户可配置的本地资源（例如：内存、SDD、HDD），对底层存储进行数据操作。

Client

Alluxio Client主要包括三种方式：Java API、Shell、HTTP REST API。

- Java API
  提供Alluxio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Alluxio客户端的开发。
- Shell
  提供shell命令完成Alluxio文件系统的基本操作。
- HTTP REST API
  提供除Shell、Java API以外的其他接口，可通过此接口查询信息，具体请参考**Alluxio接口**。

Namespace

透明命名机制：保证了Alluxio和底层存储系统的命名空间是一致的。

统一命名空间：Alluxio提供了一个挂载API，通过该API能够在Alluxio中访问多个数据源中的数据。

15.1.3 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 Alluxio 应用程序开发流程

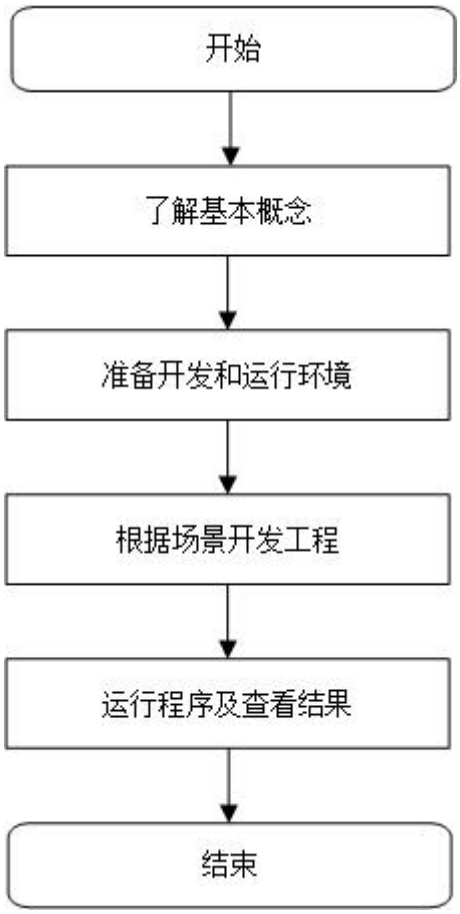


表 15-1 Alluxio 应用开发的流程说明

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Alluxio的基本概念。	常用概念
准备开发和运行环境	Alluxio的客户端程序当前推荐使用java语言进行开发，并使用Maven工具构建工程。样例程序运行环境即MRS服务所VPC集群的节点。	开发环境简介
根据场景开发工程	提供了Java语言的样例工程和数据查询的样例工程。	场景说明
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。	调测程序

15.2 环境准备

15.2.1 开发环境简介

在进行应用开发时，要准备的本地开发环境如表15-2所示。同时需要准备运行调测的Linux环境，用于验证应用程序运行是否正常。

表 15-2 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：Linux系统。
安装JDK和Maven	开发环境的基本配置：Java JDK 8或以上、Maven 3.3.9或以上
安装和配置Eclipse或IntelliJ IDEA	用于开发Alluxio应用程序的工具。
网络	确保客户端与Alluxio服务主机在网络上互通。

15.2.2 准备环境

- 选择Windows开发环境下，安装Eclipse，安装JDK。
  建议JDK使用1.8版本，Eclipse使用4.3.2及以上版本。

说明

- 若使用IBM JDK，请确保Eclipse中的JDK配置为IBM JDK。
- 若使用Oracle JDK，请确保Eclipse中的JDK配置为Oracle JDK。
- 若使用ODBC进行二次开发，请确保JDK版本为1.8及以上版本。
- 不同的Eclipse不要使用相同的workspace和相同路径下的示例工程。

- 准备一个应用程序运行测试的Linux环境。

准备运行调测环境

步骤1 在弹性云服务器管理控制台，申请一个新的弹性云服务器，用于用户应用程序开发、运行、调测。

- 弹性云服务器的主机操作系统选择“EulerOS”，版本请根据需要选择。
- 弹性云服务器的安全组需要和MRS集群Master节点的安全组相同。
- 弹性云服务器的VPC需要与MRS集群在同一个VPC中。
- 弹性云服务器的网卡需要与MRS集群在同一个网段中。

步骤2 在弹性云服务器页面申请并绑定弹性云服务器IP，具体请参考**为弹性云服务器申请和绑定弹性公网IP**。

步骤3 配置安全组出入规则，具体请参考**配置安全组规则**。

步骤4 下载客户端程序。

1. 登录**MRS Manager**系统。
2. 选择“服务管理 > 下载客户端 ”，下载“完整客户端”到“远端主机”上，即下载客户端程序到新申请的弹性云服务器上。

步骤5 以root用户登录存放下载的客户端的节点，再安装客户端。

1. 执行以下命令解压客户端包。

   cd /opt

   tar -xvf /opt/MRS_Services_Client.tar

2. 执行以下命令校验安装文件包。

   sha256sum -c /opt/MRS_Services_ClientConfig.tar.sha256

   ```
   MRS_Services_ClientConfig.tar:OK
   ```

3. 执行以下命令解压安装文件包。

   tar -xvf MRS_Services_ClientConfig.tar

4. 执行以下命令安装客户端到指定目录（绝对路径），例如“/opt/client”。目录会自动创建。

   cd /opt/MRS_Services_ClientConfig

   sh install.sh /opt/client

   ```
   Components client installation is complete.
   ```

----结束

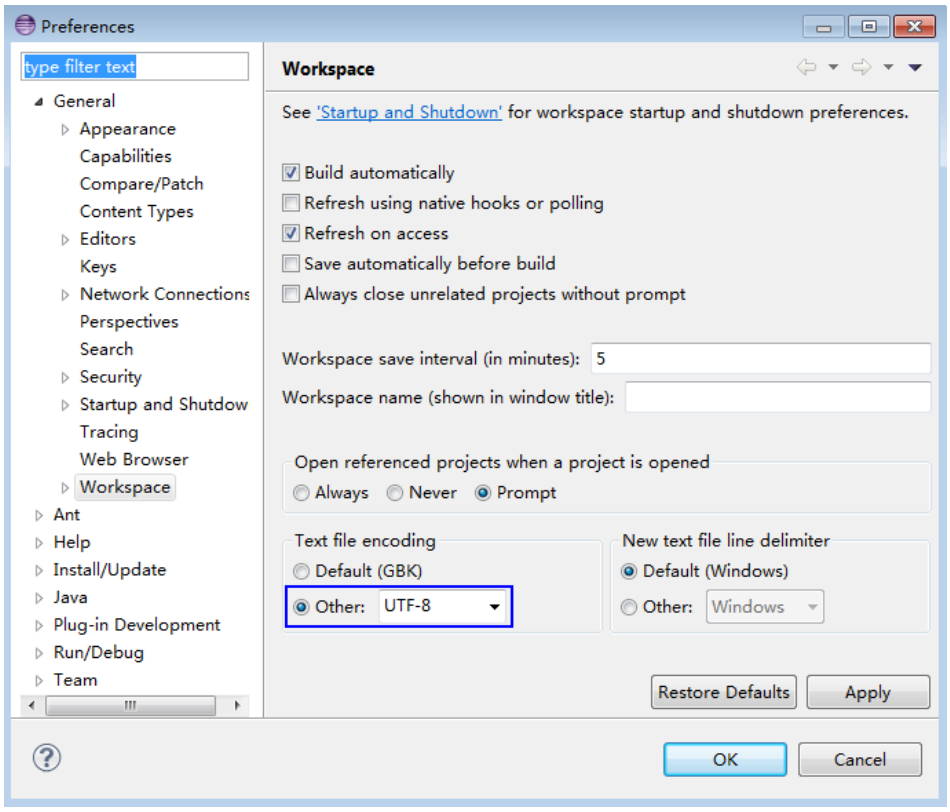
15.2.3 获取并导入样例工程

1. 在**样例工程获取地址**，下载样例工程到本地。
2. 导入样例工程到Eclipse开发环境。

a. 打开Eclipse，选择“File > Import”。显示“Import”窗口，选择Existing Maven Projects，单击“next”按钮。
b. 在“Import Maven Projects”窗口单击“Browse”。显示“Select Root Folder”对话框。
c. 选择样例工程文件夹alluxio-examples，单击“确定”按钮。
d. 在“Import Maven Projects”窗口单击“Finish”按钮。

3. 设置Eclipse的文本文件编码格式，解决乱码显示问题。
   a. 在Eclipse的菜单栏中，选择“Window > Preferences”。
      弹出“Preferences”窗口。
   b. 在左边导航上选择“General > Workspace”，在“Text file encoding”区域，选中“Other”，并设置参数值为“UTF-8”，单击“Apply”后，单击“OK”，如图15-2所示。

图 15-2 设置 Eclipse 的编码格式



15.3 开发程序

15.3.1 场景说明

场景说明

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

Alluxio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和对文件的读写；Alluxio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. 文件系统初始化
2. 写文件
3. 读文件

开发思路

1. 调用FileSystem中的create接口获取文件系统客户端
2. 调用FileSystem中的crteateFile接口创建文件
3. 调用FileOutStream中的wirte接口写文件
4. 调用FileSystem中的openFile接口打开文件
5. 调用FileInStream中的in接口读取文件

15.3.2 Alluxio 初始化

功能简介

在使用Alluxio提供的API之前，需要先进行Alluxio初始化操作。过程为：

1. 加载HDFS服务配置文件。
2. 实例化Filesystem。
3. 使用HDFS的API。

代码样例

如下是代码片段，详细代码请参考ExampleClient类。

```
/**
* load configurations from alluxio-site.properties
* @throws IOException
*/
private void loadConf() throws IOException {
    InputStream fileInputStream = null;
    alluxioConf = new Properties();
    File propertiesFile = new File(PATH_TO_ALLUXIO_SITE_PROPERTIES);
    try {
        fileInputStream = new FileInputStream(propertiesFile);
        alluxioConf.load(fileInputStream);
    }
    catch (FileNotFoundException e) {
        System.out.println(PATH_TO_ALLUXIO_SITE_PROPERTIES + "does not exist. Exception: " + e);
    }
    catch (IOException e) {
        System.out.println("Failed to load configuration file. Exception: " + e);
    }
    finally{
        close(fileInputStream);
    }
}

/**
* build Alluxio instance
*/
```

```
private void instanceBuild() throws IOException {
// get filesystem
    InstancedConfiguration conf = new InstancedConfiguration(ConfigurationUtils.defaults());
    conf.set(PropertyKey.MASTER_RPC_ADDRESSES, alluxioConf.get("alluxio.master.rpc.addresses"));
    FileSystemContext fsContext = FileSystemContext.create(conf);
    fSystem = FileSystem.Factory.create(fsContext);
}
```

15.3.3 写文件

功能简介

写文件过程为：

1. 实例化一个FileSystem。
2. 由此FileSystem实例获取写文件的各类资源。
3. 将待写内容写入到Alluxio的指定文件中。

代码样例

```
/**
* create file,write file
*/
private void write() throws IOException {
    final String content = "hi, I am bigdata. It is successful if you can see me.";
    FileOutStream out = null;
    try {
        AlluxioURI path = new AlluxioURI(testFilePath);
        out = fSystem.createFile(path);
        out.write(content.getBytes());
    }
    catch (Exception e){
        System.out.println("Failed to write file. Exception:" + e);
    }
    finally {
        close(out);
    }
}
```

15.3.4 读文件

功能简介

获取Alluxio上某个指定文件的内容。

代码样例

```
/**
* read file
* @throws java.io.IOException
*/
private void read() throws IOException {
    AlluxioURI path = new AlluxioURI(testFilePath);
    FileInStream in = null;
    try{
        in = fSystem.openFile(path);
        byte[] buffer = new byte[1024];
        int len;
        String content = "";
        while((len = in.read(buffer)) != -1){
            String bufferStr = new String(buffer,0, len);
            content += bufferStr;
```

```
        }
        System.out.println(content);
    }
    catch (Exception e){
        System.out.println("Failed to read file. Exception:" + e);
    }
    finally {
        close(in);
    }
}
```

15.4 调测程序

Alluxio 客户端运行及结果查看

步骤1 执行mvn clean compile assembly:single生成jar包，在工程目录target目录下获取，比如:alluxio-examples-mrs-1.9-jar-with-dependencies.jar。

步骤2 在运行调测环境上创建一个目录作为运行目录，如或“/opt/alluxio_examples”(Linux环境)，并在该目录下创建子目录“conf”。

将**步骤1**导出的alluxio-examples-mrs-1.9-jar-with-dependencies.jar拷贝到“/opt/alluxio_examples”下。

将客户端下的配置文件“/opt/client/Alluxio/alluxio/conf/alluxio-site.properties”拷贝到“conf”下。

说明

当Alluxio集群启动时，每一个Alluxio服务端进程（包括master和worke）在目录“${CLASSPATH}”，“${HOME}/.alluxio/”，“/etc/alluxio/”，和“${ALLUXIO_HOME}/conf”下顺序读取alluxio-site.properties，当alluxio-site.properties文件被读取到则跳过剩余路径的查找，所以请根据实际环境情况存放alluxio-site.properties文件。

步骤3 在Linux环境下执行运行样例程序。

```
chmod +x /opt/alluxio_examples -R
cd /opt/alluxio_examples
java -jar alluxio-examples-mrs-1.9-jar-with-dependencies.jar /testFlie.txt
```

步骤4 在命令行终端查看样例代码所查询出的结果。

Linux环境运行成功结果会有如下信息：

```
hi, I am bigdata. It is successful if you can see me.
```

----结束

15.5 Alluxio 接口

Java 接口

Alluxio接口遵循标准的Alluxio Parent API标准，详情请见**https://docs.alluxio.io/os/javadoc/2.0/index.html**。

HTTP REST API

- Master REST API：**https://docs.alluxio.io/os/restdoc/2.0/master/index.html**

- Worker REST API：**https://docs.alluxio.io/os/restdoc/2.0/worker/index.html**
- Proxy REST API：**https://docs.alluxio.io/os/restdoc/2.0/proxy/index.html**
- Job REST API：**https://docs.alluxio.io/os/restdoc/2.0/job/index.html**

16 附录

16.1 登录 MRS Manager

MRS Manager支持监控、配置和管理MRS集群，用户可以在MRS控制台页面打开Manager管理页面。

本章节介绍如何打开MRS Manager方法。

登录 MRS Manager

步骤1 登录MRS管理控制台页面。

步骤2 在“现有集群” 列表，单击指定的集群名称，进入集群信息页面。

步骤3 单击“点击管理”，打开“访问MRS Manager页面”。

- 若用户创建集群时已经绑定弹性公网IP，如**图16-1**所示。
  a. 选择待添加的安全组规则所在安全组，该安全组在创建群时配置。
  b. 添加安全组规则，默认填充的是用户访问公网IP地址9022端口的规则，如需开放多个IP段为可信范围用于访问MRS Manager页面，请参考**添加安全组规则**。如需对安全组规则进行查看，修改和删除操作，请单击“管理安全组规则”。

     说明

     - 自动获取的访问公网IP与用户本机IP不一致，属于正常现象，无需处理。
     - 9022端口为knox的端口，需要开启访问knox的9022端口权限，才能访问MRS Manager服务。

  c. 勾选“我确认xx.xx.xx.xx为可信任的公网访问IP，并允许从该IP访问MRS Manager页面。”

图 16-1 添加访问 MRS Manager 的安全组规则

- 若用户创建集群时暂未绑定弹性公网IP，如图16-2所示。
  a. 在弹性公网IP下拉框中选择可用的弹性公网IP或单击“管理弹性公网IP”创建弹性公网IP。
  b. 选择待添加的安全组规则所在安全组，该安全组在创建群时配置。
  c. 添加安全组规则，默认填充的是用户访问公网IP地址9022端口的规则，如需开放多个IP段为可信范围用于访问MRS Manager页面，请参考**添加安全组规则**。如需对安全组规则进行查看，修改和删除操作，请单击“管理安全组规则”。

     说明

     - 自动获取的访问公网IP与用户本机IP不一致，属于正常现象，无需处理。
     - 9022端口为knox的端口，需要开启访问knox的9022端口权限，才能访问MRS Manager服务。

  d. 勾选“我确认xx.xx.xx.xx为可信任的公网访问IP，并允许从该IP访问MRS Manager页面。”

图 16-2 绑定弹性公网 IP

步骤4 单击“确定”，进入MRS Manager登录页面。

步骤5 输入创建集群时默认的用户名“admin”及设置的密码，单击“登录”进入MRS Manager页面。

----结束

添加安全组规则

如需为其他用户开通访问MRS Manager的权限，参考该小节内容添加对应用户访问公网的IP地址为可信范围。

步骤1 在MRS管理控制台，在“现有集群” 列表，单击指定的集群名称，进入集群信息页面。

步骤2 单击弹性公网IP后边的“添加安全组规则”，如图16-3所示。

图 16-3 集群详情

步骤3 进入“添加安全组规则”页面，添加需要开放权限用户访问公网的IP地址段并勾选“我确认这里设置的授权对象是可信任的公网访问IP范围，禁止使用0.0.0.0/0,否则会有安全风险。”如图16-4所示。

图 16-4 添加安全组规则

默认填充的是用户访问公网的IP地址，用户可根据需要修改IP地址段，如需开放多个IP段为可信范围，请重复执行步骤6-步骤9。如需对安全组规则进行查看，修改和删除操作，请单击“管理安全组规则”。

步骤4 单击“确定”完成安全组规则添加。

----结束

16.2 下载 MRS 客户端

步骤1 登录MRS Manager，请参考**登录MRS Manager**。

步骤2 选择“服务管理”。

步骤3 单击“下载客户端”。

步骤4 在“客户端类型”选择“完整客户端”。

步骤5 在“下载路径”选择“远端主机”。

步骤6 将“主机IP”设置为新申请的弹性云服务器的IP地址，设置“主机端口”为“22”，并将“存放路径”设置为“/tmp”。

- 如果使用SSH登录ECS的默认端口“22”被修改，请将“主机端口”设置为新端口。
- “保存路径”最多可以包含256个字符。

步骤7 “登录用户”设置为“root”。

如果使用其他用户，请确保该用户对保存目录拥有读取、写入和执行权限。

步骤8 在“登录方式”选择“密码”或“SSH私钥”。

- 密码：输入创建集群时设置的root用户密码。
- SSH私钥：选择并上传创建集群时使用的密钥文件。

图 16-5 下载客户端

步骤9 单击“确定”开始生成客户端文件。

- 若界面显示以下提示信息表示客户端包已经成功保存。单击“关闭”。客户端文件请到下载客户端时设置的远端主机的“存放路径”中获取。
  下载客户端文件到远端主机成功。
- 若界面显示以下提示信息，请检查用户名密码及远端主机的安全组配置，确保用户名密码正确，及远端主机的安全组已增加SSH(22)端口的入方向规则。然后从步骤2执行重新开始下载客户端。
  连接到服务器失败，请检查网络连接或参数设置。

说明

生成客户端会占用大量的磁盘IO，不建议在集群处于安装中、启动中、打补丁中等非稳态场景下载客户端。

----结束

16.3 修订记录

发布日期	更新特性
2020-04-20	第五次正式发布： 新增MRS 1.9版本样例代码，详见**样例工程获取地址**。 新增Alluxio应用开发章节，详见**应用开发简介**。
2019-11-28	第四次正式发布： 新增Impala应用开发章节，详见**应用开发简介**。
2019-04-02	第三次正式发布： 新增如下章节： • **HBase Phoenix样例代码调测** • **HBase python样例代码调测** • **应用开发简介** • **常用概念** • **开发流程** • **开发环境简介** • **准备环境** • **准备开发用户** • **准备JDBC客户端开发环境** • **准备HCatalog开发环境** • **典型场景说明** • **样例代码说明** • **调测程序** • **Presto接口**
2019-02-20	第二次正式发布： 修改改如下章节： • **创建表** • **修改表** • **使用过滤器Filter** • **添加二级索引** • **使用二级索引读取数据** • **配置HBase文件存储** • **HFS的JAVA API**
2019-01-11	第一次正式发布。