数据湖探索

开发指南

文档版本 01

发布日期 2025-02-17

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。

安全声明

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

目 录

1 使用客户端工具连接 DLI

1.1 使用 JDBC 提交 SQL 作业

本文介绍通过JDBC连接DLI并提交SQL作业。

1.1.1 下载并安装 JDBC 驱动包

操作场景

JDBC用于连接DLI服务，您可以在Maven获取JDBC安装包，或在DLI管理控制台下载JDBC驱动文件。

本文介绍通过JDBC连接DLI并提交SQL作业。

获取服务端连接地址

连接DLI服务的地址格式为：jdbc:dli://<endPoint>/<projectId>。因此您需要获取对应的Endpoint和项目编号。

在**地区和终端节点**获取DLI对应的Endpoint；在华为云页面上方菜单栏，单击用户名，然后在“我的凭证”页面获取项目编号。

示例：jdbc:dli://dli.cn-north-1.myhuaweicloud.com/96a17d961b84434baec6a58b9e567908。

下载并安装 JDBC 驱动

说明

JDBC版本2.X版本功能重构后，仅支持从DLI作业桶读取查询结果，如需使用该特性需具备以下条件：

- 在DLI管理控制台“全局配置 > 工程配置”中完成作业桶的配置。
- 2024年5月起，新用户可以直接使用DLI服务的“查询结果写入桶”功能，无需开通白名单。
  对于2024年5月之前开通并使用DLI服务的用户，如需使用“查询结果写入桶”功能，必须提交工单申请加入白名单。

- **方式一：使用Maven中央库来添加JDBC驱动**
  Maven中央库是Apache Maven项目的一部分，提供了Java库和框架。

在不指定JDBC获取方式的情况下，默认使用Maven中央库的方式来添加JDBC驱动。

使用maven构加入huaweicloud-dli-jdbc依赖的maven配置项为（此为默认操作无需单独配置。）

```
<dependency>
  <groupId>com.huawei.dli</groupId>
  <artifactId>huaweicloud-dli-jdbc</artifactId>
  <version>x.x.x</version>
</dependency>
```

- **方式二：通过Maven配置华为镜像源来获取JDBC驱动**

  在使用Maven管理项目依赖时，可以通过修改settings.xml文件来配置华为镜像源以获取JDBC驱动。

  ```
  <mirror>
    <id>huaweicloud</id>
    <mirrorOf>*</mirrorOf>
    <url>https://mirrors.huaweicloud.com/repository/maven/</url>
  </mirror>
  ```

- **方式三：在DLI管理控制台下载JDBC驱动文件**

  a. 登录DLI管理控制台。

  b. 单击总览页右侧“常用链接”中的“**SDK下载**”。

  c. 在“DLI SDK DOWNLOAD”页面，选择相应驱动下载。

  单击“huaweicloud-dli-jdbc-x.x.x”即可下载对应版本的JDBC驱动包。

  说明

  JDBC驱动包命名为“huaweicloud-dli-jdbc-<version>.zip”，支持在所有平台（Linux、Windows等）所有版本中使用，且依赖JDK 1.7及以上版本。

  d. 下载的JDBC驱动包中包含了.bat（Windows）或.sh（Linux/Mac）脚本，这些脚本用于自动化安装JDBC驱动到本地Maven仓库。

  您可以根据操作系统运行相应的脚本安装JDBC驱动

  - Windows：双击.bat文件或在命令行中运行。
  - Linux/Mac：运行.sh脚本。

认证鉴权

使用JDBC建立DLI驱动连接时，需要对用户进行认证鉴权。

目前JDBC支持两种认证鉴权方式，Access Key/Secret Key (AK/SK)和Token，其中Token认证仅dli-jdbc-1.x版本支持。推荐使用AK/SK认证方式。

- **（推荐）生成AK/SK**

  a. 登录DLI管理控制台。

  b. 在页面右上角的用户名的下拉列表中选择“我的凭证”。

  c. 在“我的凭证”页面，默认显示“项目列表”，切换到“管理访问密钥”页面。

  d. 单击左侧“新增访问密钥”按钮，输入“登录密码”和“短息验证码”。

  e. 单击“确定”，下载证书。

  f. 下载成功后，在credentials文件中即可获取AK和SK信息。

说明

认证用的AK和SK硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- **获取Token**

  当您使用Token认证方式完成认证鉴权时，需要获取用户Token并在JDBC连接参数中配置Token信息，获取Token的详细步骤如下。

  a. 发送***POST https://<IAM_Endpoint>/v3/auth/tokens***，请参见**地区和终端节点**，获取命令中IAM的Endpoint及消息体中的区域名称。

  请求内容示例如下。

  说明

  下面示例代码中的斜体字需要替换为实际内容，详情请参考**《统一身份认证服务API参考》**。

```
{
  "auth": {
    "identity": {
      "methods": [
        "password"
      ],
      "password": {
        "user": {
          "name": "username",
          "password": "password",
          "domain": {
            "name": "domainname"
          }
        }
      }
    },
    "scope": {
      "project": {
        "id": "0aa253a31a2f4cfda30eaa073fee6477" //假设project_id是“0aa253a31a2f4cfda30eaa073fee6477”
      }
    }
  }
}
```

  b. 请求响应成功后在响应消息头中包含的“X-Subject-Token”的值即为Token值。

1.1.2 使用 JDBC 连接 DLI 并提交 SQL 作业

操作场景

在Linux或Windows环境下您可以使用JDBC应用程序连接DLI服务端提交作业。

说明

- 使用JDBC连接DLI提交的作业运行在Spark引擎上。
- JDBC版本2.X版本功能重构后，仅支持从DLI作业桶读取查询结果，如需使用该特性需具备以下条件：
  - 在DLI管理控制台“全局配置 > 工程配置”中完成作业桶的配置。
  - 2024年5月起，新用户可以直接使用DLI服务的“查询结果写入桶”功能，无需开通白名单。

    对于2024年5月之前首次使用DLI服务的用户，如需使用“查询结果写入桶”功能，必须提交工单申请加入白名单。

DLI支持13种数据类型，每一种类型都可以映射成一种JDBC类型，在使用JDBC连接服务器时，请使用映射后的JAVA类型，映射关系如**表1-1**所示。

表 1-1 数据类型映射

DLI类型	JDBC类型	JAVA类型
INT	INTEGER	java.lang.Integer
STRING	VARCHAR	java.lang.String
FLOAT	FLOAT	java.lang.Float
DOUBLE	DOUBLE	java.lang.Double
DECIMAL	DECIMAL	java.math.BigDecimal
BOOLEAN	BOOLEAN	java.lang.Boolean
SMALLINT/SHORT	SMALLINT	java.lang.Short
TINYINT	TINYINT	java.lang.Short
BIGINT/LONG	BIGINT	java.lang.Long
TIMESTAMP	TIMESTAMP	java.sql.Timestamp
CHAR	CHAR	Java.lang.Character
VARCHAR	VARCHAR	java.lang.String
DATE	DATE	java.sql.Date

前提条件

在使用JDBC前，需要进行如下操作：

1. 授权。

   DLI使用统一身份认证服务（Identity and Access Management，简称IAM）进行精细的企业级多租户管理。该服务提供用户身份认证、权限分配、访问控制等功能，可以帮助您安全地控制华为云资源的访问。

   通过IAM，您可以在华为云账号中给员工创建IAM用户，并使用策略来控制他们对华为云资源的访问范围。

   目前包括角色（粗粒度授权）和策略（细粒度授权）。具体的权限介绍和授权操作请参考《**数据湖探索用户指南**》。
2. 创建队列。在“资源管理 > 队列管理”下，单击右上角“购买队列”，进入购买队列页面选择“通用队列”，即Spark作业的计算资源。

   说明

   如果创建队列的用户不是管理员用户，在创建队列后，需要管理员用户赋权后才可使用。关于赋权的具体操作请参考《**数据湖探索用户指南**》。

操作步骤

步骤1 在使用JDBC的机器中安装JDK，JDK版本为1.7或以上版本，并配置环境变量。

步骤2 参考**下载并安装JDBC驱动包**章节，获取DLI JDBC驱动包“huaweicloud-dli-jdbc-<version>.zip”，解压，获得“huaweicloud-dli-jdbc-<version>-jar-with-dependencies.jar”。

步骤3 在使用JDBC的机器中，将上一步解压的文件“huaweicloud-dli-jdbc-1.1.1-jar-with-dependencies.jar”添加至Java工程的“classpath”路径下。

步骤4 DLI JDBC提供两种身份认证模式连接到DLI服务，即Token和AK/SK。获取Token和AK/SK的方法请参见**认证鉴权**。

步骤5 使用Class.forName（ ）加载DLI JDBC驱动程序。

```
Class.forName("com.huawei.dli.jdbc.DliDriver");
```

步骤6 通过DriverManager的GetConnection方法创建Connection。

```
Connection conn = DriverManager.getConnection(String url, Properties info);
```

其中，JDBC的配置项通过url传入，请参考**表1-2**配置参数。JDBC配置对象，除了在url中以分号间隔设置配置项外，还可以通过Info对象动态设置属性项，具体属性项参见**表1-3**。

表 1-2 数据库连接参数

参数	描述
url	url的格式如下。 jdbc:dli://<endPoint>/projectId? <key1>=<val1>;<key2>=<val2>… • endpoint指DLI的域名。projectId指项目ID。 在**地区和终端节点**获取DLI对应的Endpoint，从华为云“用户名”>“我的凭证”页面获取项目编号。 • “？”后面接其他配置项，每个配置项以“key=value”的形式列出，配置项之间以“;”隔开，这些配置项也可以通过Info对象传入。
Info	Info传入自定义的配置项,若Info没有属性项传入，可设为null。配置格式为：info.setProperty("属性项", "属性值")。

表 1-3 属性项

属性项	必须配置	默认值	描述	不同版本dli-jdbc支持情况
queuename	是	-	DLI服务的队列名称。	dli-jdbc-1.x dli-jdbc-2.x

属性项	必须配置	默认值	描述	不同版本dli-jdbc支持情况
databasename	否	-	数据库名称。	dli-jdbc-1.x dli-jdbc-2.x
authenticationmode	否	token	身份认证方式，当前支持两种：token或aksk。	dli-jdbc-1.x
accesskey	是	-	AK/SK认证密钥，获取方式请参考**认证鉴权**。	dli-jdbc-1.x dli-jdbc-2.x
secretkey	是	-	AK/SK认证密钥，获取方式请参考**认证鉴权**。	dli-jdbc-1.x dli-jdbc-2.x
regionname	authenticationmode=aksk时必须配置	-	区域名称，具体区域请参考**地区和终端节点**。	dli-jdbc-1.x dli-jdbc-2.x
token	authenticationmode=token时必须配置	-	Token认证，认证方式请参考**认证鉴权**。	dli-jdbc-1.x
charset	否	UTF-8	JDBC编码方式。	dli-jdbc-1.x dli-jdbc-2.x
usehttpproxy	否	false	是否使用访问代理。	dli-jdbc-1.x
proxyhost	usehttpproxy=true时必须配置	-	访问代理host。	dli-jdbc-1.x dli-jdbc-2.x
proxyport	usehttpproxy=true时必须配置	-	访问代理端口。	dli-jdbc-1.x dli-jdbc-2.x

属性项	必须配置	默认值	描述	不同版本dli-jdbc支持情况
dli.sql.checkNoResultQuery	否	false	是否允许调用executeQuery接口执行没有返回结果的语句（如DDL）。 ● “false”表示允许调用。 ● “true”表示不允许调用。	dli-jdbc-1.x dli-jdbc-2.x
jobtimeout	否	300	提交作业终止时间，单位：秒。	dli-jdbc-1.x dli-jdbc-2.x
directfetchthreshold	否	1000	请您根据业务情况判断返回结果数是否超过设置的阈值。 默认阈值1000。	dli-jdbc-1.x

步骤7 创建Statement对象，设置相关参数并提交Spark SQL到DLI服务。

Statement statement = conn.createStatement();

statement.execute("SET dli.sql.spark.sql.forcePartitionPredicatesOnPartitionedTable.enabled=true");

statement.execute("select * from tb1");

步骤8 获取结果。

ResultSet rs = statement.getResultSet();

步骤9 显示结果。

```
while (rs.next()) {
int a = rs.getInt(1);
int b = rs.getInt(2);
}
```

步骤10 关闭连接。

conn.close();

----结束

示例

说明

- 认证用的ak和sk硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。
- 本示例以ak和sk保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量System.getenv("AK")和System.getenv("SK")。

```
import java.sql.*;
import java.util.Properties;
```



```
public class DLIJdbcDriverExample {

    public static void main(String[] args) throws ClassNotFoundException, SQLException {
        Connection conn = null;
        try {
            Class.forName("com.huawei.dli.jdbc.DliDriver");
            String url = "jdbc:dli://<endpoint>/<projectId>?databasename=db1;queuename=testqueue";
            Properties info = new Properties();
            info.setProperty("authenticationmode", "aksk");
            info.setProperty("regionname", "<real region name>");
            info.setProperty("accesskey", "<System.getenv("AK")>");
            info.setProperty("secretkey", "<System.getenv("SK")>");
            conn = DriverManager.getConnection(url, info);
            Statement statement = conn.createStatement();
            statement.execute("select * from tb1");
            ResultSet rs = statement.getResultSet();
            int line = 0;
            while (rs.next()) {
                line ++;
                int a = rs.getInt(1);
                int b = rs.getInt(2);
                System.out.println("Line:" + line + ":" + a + "," + b);
            }
            statement.execute("SET dli.sql.spark.sql.forcePartitionPredicatesOnPartitionedTable.enabled=true");
            statement.execute("describe tb1");
            ResultSet rs1 = statement.getResultSet();
            line = 0;
            while (rs1.next()) {
                line ++;
                String a = rs1.getString(1);
                String b = rs1.getString(2);
                System.out.println("Line:" + line + ":" + a + "," + b);
            }
        } catch (SQLException ex) {
        } finally {
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

1.1.3 DLI JDBC Driver 支持的 API 列表

DLI JDBC Driver支持JDBC标准的众多API，也有部分API不支持用户调用，例如涉及事务调用的API“prepareCall”，调用这类API将抛出“SQLFeatureNotSupportedException”异常。API详情请参考JDBC官网**https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html**。

支持的 API 列表

DLI JDBC Driver支持的API列表如下，对可能与JDBC标准产生歧义的地方加以备注说明。

- Connection API支持的常用方法签名：
  - Statement createStatement()
  - PreparedStatement prepareStatement(String sql)
  - void close()
  - boolean isClosed()
  - DatabaseMetaData getMetaData()
  - PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)

- Driver API支持的常用方法签名：
  - Connection connect(String url, Properties info)
  - boolean acceptsURL(String url)
  - DriverPropertyInfo[] getPropertyInfo(String url, Properties info)
- ResultSetMetaData API支持的常用方法签名：
  - String getColumnClassName(int column)
  - int getColumnCount()
  - int getColumnDisplaySize(int column)
  - String getColumnLabel(int column)
  - String getColumnName(int column)
  - int getColumnType(int column)
  - String getColumnTypeName(int column)
  - int getPrecision(int column)
  - int getScale(int column)
  - boolean isCaseSensitive(int column)
- Statement API支持的常用方法签名：
  - ResultSet executeQuery(String sql)
  - int executeUpdate(String sql)
  - boolean execute(String sql)
  - void close()
  - int getMaxRows()
  - void setMaxRows(int max)
  - int getQueryTimeout()
  - void setQueryTimeout(int seconds)
  - void cancel()
  - ResultSet getResultSet()
  - int getUpdateCount()
  - boolean isClosed()
- PreparedStatement API支持的常用方法签名：
  - void clearParameters()
  - boolean execute()
  - ResultSet executeQuery()
  - int executeUpdate()
  - PreparedStatement Set系列方法
- ResultSet API支持的常用方法签名：
  - int getRow()
  - boolean isClosed()
  - boolean next()
  - void close()
  - int findColumn(String columnLabel)

  - boolean wasNull()
  - get系列方法
- DatabaseMetaData API支持的常用方法签名
  - ResultSet getCatalogs()

    说明

    在DLI服务中没有Catalog的概念，返回空的ResultSet。
  - ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)
  - Connection getConnection()
  - getTables(String catalog, String schemaPattern,String tableNamePattern, String types[])

    说明

    该方法不采纳Catalog参数，schemaPattern对应DLI服务的database的概念。
  - ResultSet getTableTypes()
  - ResultSet getSchemas()
  - ResultSet getSchemas(String catalog, String schemaPattern)

1.2 使用 Spark-submit 提交 Spark Jar 作业

DLI Spark-submit 简介

DLI Spark-submit是一个用于提交Spark作业到DLI服务端的命令行工具，该工具提供与开源Spark兼容的命令行。

准备工作

1. 授权。

   DLI使用统一身份认证服务（Identity and Access Management，简称IAM）进行精细的企业级多租户管理。该服务提供用户身份认证、权限分配、访问控制等功能，可以帮助您安全地控制华为云资源的访问。

   通过IAM，您可以在华为云账号中给员工创建IAM用户，并使用策略来控制他们对华为云资源的访问范围。

   目前包括角色（粗粒度授权）和策略（细粒度授权）。具体的权限介绍和授权操作请参考《**数据湖探索用户指南**》。
2. 创建队列。在“资源管理 > 队列管理”下，单击右上角“购买队列”，进入购买队列页面选择“通用队列”，即Spark作业的计算资源。

   说明

   如果创建队列的用户不是管理员用户，在创建队列后，需要管理员用户赋权后才可使用。关于赋权的具体操作请参考《**数据湖探索用户指南**》。

DLI 客户端工具下载

您可以在DLI管理控制台下载DLI客户端工具。

步骤1 登录DLI管理控制台。

步骤2 单击总览页右侧“常用链接”中的“SDK下载”。

步骤3 在“DLI SDK DOWNLOAD”页面，单击“dli-clientkit-<version>”即可下载DLI客户端工具。

说明

DLI客户端空间命名为“dli-clientkit-<version>-bin.tar.gz”，支持在Linux环境中使用，且依赖JDK 1.8及以上版本。

----结束

配置 DLI Spark-submit

使用spark-submit的机器安装JDK 1.8或以上版本并配置环境变量，当前仅在Linux环境下使用spark-submit工具。

步骤1 下载并解压工具包“dli-clientkit-<version>-bin.tar.gz”，其中version为版本号，以实际版本号为准。

步骤2 进入解压目录，里面有三个子目录bin、conf、lib，分别存放了Spark-submit相关的执行脚本、配置文件和依赖包。

步骤3 进入配置文件conf目录，修改“client.properties”中的配置项，（具体配置项参考表1-4）。

表 1-4 DLI 客户端工具配置参数

属性项	必须配置	默认值	描述
dliEndPoint	否	-	DLI服务的域名。 在地区和终端节点获取DLI对应区域的域名。 如果不配置，程序根据region参数来确定域名。
obsEndPoint	是	obs.cn-north-1.myhuaweicloud.com	OBS服务的域名。 在地区和终端节点获取OBS对应区域的域名。
bucketName	是	-	OBS上的桶名称。该桶用于存放Spark程序中使用的jar包、Python程序文件、配置文件等。
obsPath	是	dli-spark-submit-resources	OBS上存放jar包、Python程序文件、配置文件等的目录，该目录在bucketName指定的桶下。如果该目录不存在，程序会自动创建。

属性项	必须配置	默认值	描述
localFilePath	是	-	存放Spark程序中使用的jar包、Python程序文件、配置文件等的本地目录。 程序会自动将Spark程序依赖到的相关文件上传的OBS路径，并加载到DLI服务端资源包。
ak	是	-	用户的Access Key。
sk	是	-	用户的Secret Key。
projectId	是	-	用户访问的DLI服务使用的项目编号。
region	是	-	对接的DLI服务的Region。

根据Spark应用程序的需要，修改“spark-defaults.conf”中的配置项，配置项兼容开源Spark配置项，参考开源Spark的配置项说明。

----结束

使用 Spark-submit 提交 Spark 作业

步骤1 进入工具文件bin目录，执行spark-submit命令，并携带相关参数。

命令执行格式：

```
spark-submit [options] <app jar | python file> [app arguments]
```

表 1-5 DLI Spark-submit 参数列表

参数名称	参数值	描述
--class	<CLASS_NAME>	提交的Java/Scala应用程序的主类名称。
--conf	<PROP=VALUE>	Spark程序的参数，可以通过在conf目录下的spark-defaults.conf中配置。如果命令中与配置文件中同时配置，优先使用命令指定的参数值。 **说明** 多个conf时，格式为：--conf key1=value1 --conf key2=value2
--jars	<JARS>	Spark应用依赖的jar包名称，存在多个时使用","分隔。jar包文件需要提前保存在client.properties文件中localFilePath配置的本地路径中。
--name	<NAME>	Spark应用的名称。
--queue	<QUEUE_NAME>	DLI服务端Spark队列名称，作业会提交到该队列中执行。

参数名称	参数值	描述
--py-files	<PY_FILES>	Spark应用依赖的Python程序文件名称，存在多个时使用","分隔。Python程序文件文件需要提前保存在client.properties文件中localFilePath配置的本地路面中。
-s,--skip-upload-resources	<all | app | deps>	是否跳过，将jar包、Python程序文件、配置文件上传到OBS和加载到DLI服务端资源列表。当相关资源文件已经加载到DLI服务资源列表中，可以使用该参数跳过该步骤。 不携带该参数时，默认会上传和加载命令中的所有资源文件到DLI服务中。 ● all：跳过所有资源文件的上传和加载 ● app：跳过Spark应用程序文件的上传和加载 ● deps：跳过所有依赖文件的上传和加载
-h,--help	-	打印命令帮助

命令举例：

```
./spark-submit --name <name> --queue <queue_name> --class org.apache.spark.examples.SparkPi spark-examples_2.11-2.1.0.luxor.jar 10
./spark-submit --name <name> --queue <queue_name> word_count.py
```

说明

请使用"./spark-submit"，不要使用"spark-submit"，后者可能会使用本地环境中已有的Spark环境，而不是DLI队列。

----结束

1.3 使用 Livy 提交 Spark Jar 作业

DLI Livy 简介

DLI Livy是基于开源的Apache Livy用于提交Spark作业到DLI的客户端工具。

准备工作

- 创建DLI队列。在“队列类型”中选择“通用队列”，即Spark作业的计算资源。具体请参考**创建队列**。
- 准备一个linux弹性云服务器ECS，用于安装DLI Livy。
  - ECS需要放通30000至32767端口、8998端口。具体操作请参考**添加安全组规则**。
  - ECS需安装Java JDK，JDK版本建议为1.8。配置Java环境变量JAVA_HOME。
  - 查询弹性云服务器ECS详细信息，获取ECS的“私有IP地址”。
- 使用增强型跨源连接打通DLI队列和Livy实例所在的VPC网络。具体操作可以参考**增强型跨源连接**。

DLI Livy 工具下载及安装

说明

本次操作下载的DLI Livy版本为**apache-livy-0.7.2.0107-bin.tar.gz**，后续版本变化请根据实际情况修改。

步骤1 单击**下载链接**，获取DLI Livy工具压缩包。

步骤2 使用WinSCP工具，将获取的工具压缩包上传到准备好的ECS服务器目录下。

步骤3 使用root用户登录ECS服务器，执行以下命令安装DLI Livy工具。

1. 执行以下命令创建工具安装路径。

   mkdir *livy安装路径*

   例如新建路径/opt/livy：**mkdir /opt/livy**。后续操作步骤均默认以**/opt/livy**安装路径演示，请根据实际情况修改。

2. 解压工具压缩包到安装路径。

   tar --extract --file apache-livy-0.7.2.0107-bin.tar.gz --directory /opt/livy --strip-components 1 --no-same-owner

3. 执行以下命令修改配置文件名称。

   cd /opt/livy/conf

   mv livy-client.conf.template livy-client.conf

   mv livy.conf.template livy.conf

   mv livy-env.sh.template livy-env.sh

   mv log4j.properties.template log4j.properties

   mv spark-blacklist.conf.template spark-blacklist.conf

   touch spark-defaults.conf

----结束

修改 DLI Livy 工具配置文件

步骤1 上传指定的DLI Livy工具jar资源包到OBS桶路径下。

1. 登录OBS控制台，在指定的OBS桶下创建一个存放Livy工具jar包的资源目录。例如：“obs://bucket/livy/jars/”。
2. 进入**步骤3.1**中DLI Livy工具所在ECS服务器的安装目录，获取以下jar包，将获取的jar包上传到**步骤1.1**创建的OBS桶资源目录下。

   例如，当前Livy工具安装路径为“/opt/livy”，则当前需要上传的jar包名称如下：

   ```
   /opt/livy/rsc-jars/livy-api-0.7.2.0107.jar
   /opt/livy/rsc-jars/livy-rsc-0.7.2.0107.jar
   /opt/livy/repl_2.11-jars/livy-core_2.11-0.7.2.0107.jar
   /opt/livy/repl_2.11-jars/livy-repl_2.11-0.7.2.0107.jar
   ```

步骤2 修改DLI Livy工具配置文件。

1. 编辑修改配置文件“ /opt/livy/conf/livy-client.conf”。

   vi /opt/livy/conf/livy-client.conf

   添加如下内容，并根据注释修改配置项。

   ```
   #当前ECS的私有IP地址，也可以使用ifconfig命令查询。
   livy.rsc.launcher.address = X.X.X.X
   #当前ECS服务器放通的端口号
   livy.rsc.launcher.port.range = 30000~32767
   ```


2. 编辑修改配置文件“ /opt/livy/conf/livy.conf”。

   vi /opt/livy/conf/livy.conf

   添加如下内容。根据注释说明修改具体的配置项。

```
livy.server.port = 8998
livy.spark.master = yarn

livy.server.contextLauncher.custom.class=org.apache.livy.rsc.DliContextLauncher
livy.server.batch.custom.class=org.apache.livy.server.batch.DliBatchSession
livy.server.interactive.custom.class=org.apache.livy.server.interactive.DliInteractiveSession
livy.server.sparkApp.custom.class=org.apache.livy.utils.SparkDliApp

livy.server.recovery.mode = recovery
livy.server.recovery.state-store = filesystem
#以下文件路径请根据情况修改
livy.server.recovery.state-store.url = file:///opt/livy/store/

livy.server.session.timeout-check = true
livy.server.session.timeout = 1800s
livy.server.session.state-retain.sec = 1800s

livy.dli.spark.version = 2.3.2
livy.dli.spark.scala-version = 2.11

# 填入存储livy jar包资源的OBS桶路径。
livy.repl.jars = obs://bucket/livy/jars/livy-core_2.11-0.7.2.0107.jar, obs://bucket/livy/jars/livy-repl_2.11-0.7.2.0107.jar
livy.rsc.jars = obs://bucket/livy/jars/livy-api-0.7.2.0107.jar, obs://bucket/livy/jars/livy-rsc-0.7.2.0107.jar
```

3. 编辑修改配置文件“/opt/livy/conf/spark-defaults.conf”。

   vi /opt/livy/conf/spark-defaults.conf

   添加如下必选参数内容。配置项参数填写说明，详见**表1-6**。

```
# 以下参数均支持在提交作业时覆盖。
spark.yarn.isPython=true
spark.pyspark.python=python3

# 当前参数值为生产环境web地址
spark.dli.user.uiBaseAddress=https://console.huaweicloud.com/dli/web
# 队列所在的region。
spark.dli.user.regionName=XXXX

# dli endpoint 地址。
spark.dli.user.dliEndPoint=XXXX

# 用于指定队列，填写已创建DLI的队列名。
spark.dli.user.queueName=XXXX

# 提交作业使用的access key。
spark.dli.user.access.key=XXXX
# 提交作业使用的secret key。
spark.dli.user.secret.key=XXXX

# 提交作业使用的projectId。
spark.dli.user.projectId=XXXX
```

表 1-6 spark-defaults.conf 必选参数说明

参数名	参数填写说明
spark.dli.user.regionName	DLI队列所在的区域名。 从**地区和终端节点**获取，对应“区域”列就是regionName。

参数名	参数填写说明
spark.dli.user.dliEndPoint	DLI队列所在的终端节点。 从**地区和终端节点**获取，对应的“终端节点（Endpoint）”就是该参数取值。
spark.dli.user.queueName	DLI队列名称。
spark.dli.user.access.key	对应用户的访问密钥。该用户需要有Spark作业相关权限，权限说明详见**权限管理**。 密钥获取方式请参考**获取AK/SK**。
spark.dli.user.secret.key	
spark.dli.user.projectId	参考**获取项目ID**获取项目ID。

以下参数为可选参数，请根据参数说明和实际情况配置。详细参数说明请参考**Spark Configuration**。

表 1-7 spark-defaults.conf 可选参数说明

Spark作业参数	对应Spark批处理参数	备注
spark.dli.user.file	file	如果是对接notebook工具场景时不需要设置。
spark.dli.user.className	class_name	如果是对接notebook工具场景时不需要设置。
spark.dli.user.scType	sc_type	推荐使用livy原生配置。
spark.dli.user.args	args	推荐使用livy原生配置。
spark.submit.pyFiles	python_files	推荐使用livy原生配置。
spark.files	files	推荐使用livy原生配置。
spark.dli.user.modules	modules	-
spark.dli.user.image	image	提交作业使用的自定义镜像，仅容器集群支持该参数，默认不设置。
spark.dli.user.autoRecovery	auto_recovery	-
spark.dli.user.maxRetryTimes	max_retry_times	-

Spark作业参数	对应Spark批处理参数	备注
spark.dli.user.catalogName	catalog_name	访问元数据时，需要将该参数配置为dli。

----结束

启动 DLI Livy 工具

步骤1 进入到工具安装目录。

例如：**cd /opt/livy**

步骤2 执行以下命令启动DLI Livy。

./bin/livy-server start

----结束

通过 DLI Livy 工具提交 Spark 作业到 DLI

本示例演示通过curl命令使用DLI Livy工具将Spark作业提交到DLI。

步骤1 将开发好的Spark作业程序jar包上传到OBS路径下。

例如，本示例上传“spark-examples_2.11-XXXX.jar”到“obs://bucket/path”路径下。

步骤2 以root用户登录到安装DLI Livy工具的ECS服务器。

步骤3 执行curl命令通过DLI Livy工具提交Spark作业请求到DLI。

说明

ECS_IP为当前安装DLI Livy工具所在的弹性云服务器的私有IP地址。

```
curl --location --request POST 'http://ECS_IP:8998/batches' \
--header 'Content-Type: application/json' \
--data '{
    "driverMemory": "3G",
    "driverCores": 1,
    "executorMemory": "2G",
    "executorCores": 1,
    "numExecutors": 1,
    "args": [
        "1000"
    ],
    "file": "obs://bucket/path/spark-examples_2.11-XXXX.jar",
    "className": "org.apache.spark.examples.SparkPi",
    "conf": {
        "spark.dynamicAllocation.minExecutors": 1,
        "spark.executor.instances": 1,
        "spark.dynamicAllocation.initialExecutors": 1,
        "spark.dynamicAllocation.maxExecutors": 2
    }
}'
```

----结束

2 SQL 作业开发指南

2.1 使用 Spark SQL 作业分析 OBS 数据

DLI支持将数据存储到OBS上，后续再通过创建OBS表即可对OBS上的数据进行分析和处理。

本指导中的操作内容包括：创建OBS表、导入OBS表数据、插入和查询OBS表数据等内容来帮助您更好的在DLI上对OBS表数据进行处理。

前提条件

- 已创建OBS的桶。具体OBS操作可以参考《**对象存储服务用户指南**》。本指导中的OBS桶名都为“dli-test-021”。
- 已创建DLI的SQL队列。创建队列详细介绍请参考**创建队列**。

  注意：创建队列时，队列类型必须要选择为：**SQL队列。**

前期准备

创建DLI数据库

1. 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列。
2. 在SQL编辑器中输入以下语句创建数据库“testdb”。详细的DLI创建数据库的语法可以参考**创建DLI数据库**。

   ```
   create database testdb;
   ```

后续章节操作都需要在testdb数据库下进行操作。

DataSource 和 Hive 两种语法创建 OBS 表的区别

DataSource语法和Hive语法主要区别在于支持的表数据存储格式范围、支持的分区数等有差异。两种语法创建OBS表主要差异点参见**表2-1**。

表 2-1 DataSource 语法和 Hive 语法创建 OBS 表的差异点

语法	支持的数据类型范围	创建分区表时分区字段差异	支持的分区数
DataSource语法	支持ORC，PARQUET，JSON，CSV，AVRO类型	创建分区表时，分区字段在表名和PARTITIONED BY后都需要指定。具体可以参考**DataSource语法创建单分区OBS表**。	单表分区数最多允许7000个。
Hive语法	支持TEXTFILE, AVRO, ORC, SEQUENCEFILE, RCFILE, PARQUET	创建分区表时，指定的分区字段不能出现在表后，只能通过PARTITIONED BY指定分区字段名和类型。具体可以参考**Hive语法创建OBS分区表**。	单表分区数最多允许100000个。

创建OBS表的DataSource语法可以参考**使用DataSource语法创建OBS表**。

创建OBS表的Hive语法可以参考**使用Hive语法创建OBS表**。

使用 DataSource 语法创建 OBS 表

以下通过创建CSV格式的OBS表举例，创建其他数据格式的OBS表方法类似，此处不一一列举。

- 创建OBS非分区表
  - 指定OBS数据文件，创建csv格式的OBS表。
    i. 按照以下文件内容创建“test.csv”文件，并将“test.csv”文件上传到OBS桶“dli-test-021”的根目录下。

```
Jordon,88,23
Kim,87,25
Henry,76,26
```

    ii. 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”，执行以下命令创建OBS表。

```
CREATE TABLE testcsvdatasource (name STRING, score DOUBLE, classNo INT
) USING csv OPTIONS (path "obs://dli-test-021/test.csv");
```

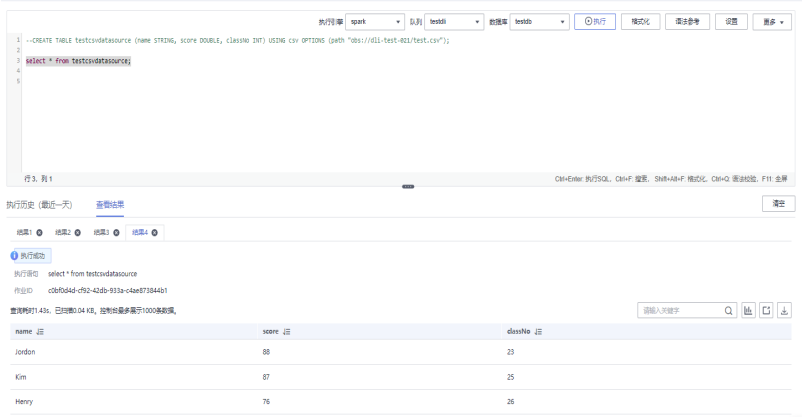
    注意

    如果是通过指定的数据文件创建的OBS表，后续不支持在DLI通过insert表操作插入数据。OBS文件内容和表数据保持同步。

    iii. 查询已创建的“testcsvdatasource”表数据。

```
select * from testcsvdatasource;
```

图 2-1 查询结果



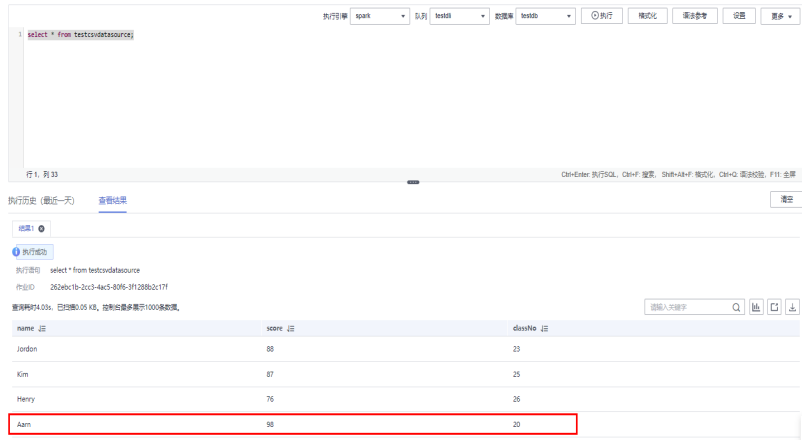
iv. 本地修改原始的OBS表文件“test.csv”，增加一行“Aarn,98,20”数据，重新替换OBS桶目录下的“test.csv”文件。

```
Jordon,88,23
Kim,87,25
Henry,76,26
Aarn,98,20
```

v. 在DLI的SQL编辑器中再次查询“testcsvdatasource”表数据，DLI上可以查询到新增的“Aarn,98,20”数据。

```
select * from testcsvdatasource;
```

图 2-2 查询结果



- 指定OBS数据文件目录，创建csv格式的OBS表。
  - 指定的OBS数据目录不包含数据文件。
    1) 在OBS桶“dli-test-021”根目录下创建数据文件目录“data”。
    2) 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”。在DLI的“testdb”数据库下创建OBS表“testcsvdata2source”。

       ```
       CREATE TABLE testcsvdata2source (name STRING, score DOUBLE, classNo INT)
       USING csv OPTIONS (path "obs://dli-test-021/data");
       ```

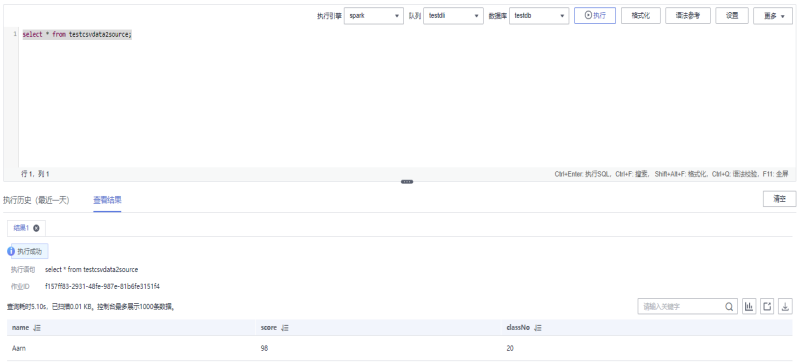
    3) 通过insert语句插入表数据。

       ```
       insert into testcsvdata2source VALUES('Aarn','98','20');
       ```

    4) insert作业运行成功后，查询OBS表“testcsvdata2source”数据。

       ```
       select * from testcsvdata2source;
       ```

图 2-3 查询结果



5) 在OBS桶的“obs://dli-test-021/data”目录下刷新后查询，生成了csv数据文件，文件内容为insert插入的数据内容。

图 2-4 查询结果

- 指定的OBS数据目录包含数据文件。

  1) 在OBS桶“dli-test-021”根目录下创建数据文件目录“data2”。创建如下内容的测试数据文件“test.csv”，并上传文件到“obs://dli-test-021/data2”目录下。

  ```
  Jordon,88,23
  Kim,87,25
  Henry,76,26
  ```

  2) 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”。在DLI的“testdb”数据库下创建OBS表“testcsvdata3source”。

  ```
  CREATE TABLE testcsvdata3source (name STRING, score DOUBLE, classNo INT)
  USING csv OPTIONS (path "obs://dli-test-021/data2");
  ```

  3) 通过insert语句插入表数据。

  ```
  insert into testcsvdata3source VALUES('Aarn','98','20');
  ```

  4) insert作业运行成功后，查询OBS表“testcsvdata3source”数据。

  ```
  select * from testcsvdata3source;
  ```

  图 2-5 查询结果

name	score	classNo
Jordon	88	23
Kim	87	25
Henry	76	26
Aarn	98	20

  5) 在OBS桶的“obs://dli-test-021/data2”目录下刷新后查询，生成了一个csv数据文件，内容为insert插入的表数据内容。

图 2-6 查询结果

- 创建OBS分区表
  - 创建单分区OBS表
    i. 在OBS桶“dli-test-021”根目录下创建数据文件目录“data3”。
    ii. 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”。在DLI的“testdb”数据库下创建以“classNo”列为分区的OBS分区表“testcsvdata4source”，指定OBS目录“obs://dli-test-021/data3”。
    ```
    CREATE TABLE testcsvdata4source (name STRING, score DOUBLE, classNo INT) USING csv OPTIONS (path "obs://dli-test-021/data3") PARTITIONED BY (classNo);
    ```
    iii. 在OBS桶的“obs://dli-test-021/data3”目录下创建“classNo=25”的分区目录。根据以下文件内容创建数据文件“test.csv”，并上传到OBS的“obs://dli-test-021/data3/classNo=25”目录下。
    ```
    Jordon,88,25
    Kim,87,25
    Henry,76,25
    ```
    iv. 在SQL编辑器中执行以下命令，导入分区数据到OBS表“testcsvdata4source ”。
    ```
    ALTER TABLE
      testcsvdata4source
    ADD
      PARTITION (classNo = 25) LOCATION 'obs://dli-test-021/data3/classNo=25';
    ```
    v. 查询OBS表“testcsvdata4source ”classNo分区为“25”的数据：
    ```
    select * from testcsvdata4source where classNo = 25;
    ```

图 2-7 查询结果

name	score	classNo
Jordon	88	25
Kim	87	25
Henry	76	25

    vi. 插入如下数据到OBS表“testcsvdata4source ”：
    ```
    insert into testcsvdata4source VALUES('Aarn','98','25');
    insert into testcsvdata4source VALUES('Adam','68','24');
    ```
    vii. 查询OBS表“testcsvdata4source ”classNo分区为“25”和“24”的数据。

注意

分区表在进行查询时where条件中必须携带分区字段，否则会查询失败，报：DLI.0005: There should be at least one partition pruning predicate on partitioned table。

```
select * from testcsvdata4source where classNo = 25;
```

图 2-8 查询结果

name	score	classNo
Jordon	88	25
Kim	87	25
Henry	76	25
Aarn	98	25

```
select * from testcsvdata4source where classNo = 24;
```

图 2-9 查询结果

name	score	classNo
Adam	68	24

viii. 在OBS桶的“obs://dli-test-021/data3”目录下点击刷新，该目录下生成了对应的分区文件，分别存放新插入的表数据。

图 2-10 OBS 上 classNo 分区为“25”文件数据

图 2-11 OBS 上 classNo 分区为“24”文件数据

- 创建多分区OBS表

  i. 在OBS桶“dli-test-021”根目录下创建数据文件目录“data4”。

  ii. 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”。在“testdb”数据库下创建以“classNo”和“dt”列为分区的OBS分区表“testcsvdata5source”，指定OBS目录“obs://dli-test-021/data4”。

  ```
  CREATE TABLE testcsvdata5source (name STRING, score DOUBLE, classNo INT, dt varchar(16)) USING csv OPTIONS (path "obs://dli-test-021/data4") PARTITIONED BY (classNo,dt);
  ```

  iii. 给 testcsvdata5source表插入如下测试数据：

  ```
  insert into testcsvdata5source VALUES('Aarn','98','25','2021-07-27');
  insert into testcsvdata5source VALUES('Adam','68','25','2021-07-28');
  ```

  iv. 根据classNo分区列查询testcsvdata5source数据。

  ```
  select * from testcsvdata5source where classNo = 25;
  ```

图 2-12 查询结果

name	score	classNo	dt
Aarn	98	25	2021-07-27
Adam	68	25	2021-07-28

v. 根据dt分区列查询testcsvdata5source数据。

```
select * from testcsvdata5source where dt like '2021-07%';
```

图 2-13 查询结果

name	score	classNo	dt
Aarn	98	25	2021-07-27
Adam	68	25	2021-07-28

vi. 在OBS桶“obs://dli-test-021/data4”目录下刷新后查询，会生成如下数据文件：

- 文件目录1：obs://dli-test-021/data4/*xxxxxx*/classNo=25/dt=2021-07-27

图 2-14 查询结果

- 文件目录2：obs://dli-test-021/data4/*xxxxxx*/classNo=25/dt=2021-07-28

图 2-15 查询结果

vii. 在OBS桶的“obs://dli-test-021/data4”目录下创建“classNo=24”的分区目录，再在“classNo=24”目录下创建子分区目录“dt=2021-07-29”。根据以下文件内容创建数据文件“test.csv”，并上传到OBS的“obs://dli-test-021/data4/classNo=24/dt=2021-07-29”目录下。

```
Jordon,88,24,2021-07-29
Kim,87,24,2021-07-29
Henry,76,24,2021-07-29
```

viii. 在SQL编辑器中执行以下命令，导入分区数据到OBS表“testcsvdata5source”。

```
ALTER TABLE
  testcsvdata5source
ADD
  PARTITION (classNo = 24,dt='2021-07-29') LOCATION 'obs://dli-test-021/data4/
classNo=24/dt=2021-07-29';
```

ix. 根据classNo分区列查询testcsvdata5source数据。

```
select * from testcsvdata5source where classNo = 24;
```

图 2-16 查询结果

name	score	classNo	dt
Jordon	88	24	2021-07-29
Kim	87	24	2021-07-29
Henry	76	24	2021-07-29

x. 根据dt分区列查询所有“2021-07”月的所有数据。

```
select * from testcsvdata5source where dt like '2021-07%';
```

图 2-17 查询结果

name	score	classNo	dt
Jordon	88	24	2021-07-29
Kim	87	24	2021-07-29
Henry	76	24	2021-07-29
Aarn	98	25	2021-07-27
Adam	68	25	2021-07-28

使用 Hive 语法创建 OBS 表

以下通过创建TEXTFILE格式的OBS表举例，创建其他数据格式的OBS表方法类似，此处不一一列举。

- 创建OBS非分区表
  a. 在OBS桶的“dli-test-021”根目录下创建数据文件目录“data5”。根据以下文件内容创建数据文件“test.txt”并上传到OBS的“obs://dli-test-021/data5”目录下。

```
Jordon,88,23
Kim,87,25
Henry,76,26
```

  b. 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”。使用Hive语法创建OBS表，指定OBS文件路径为“obs://dli-test-021/data5/test.txt”，行数据分隔符为','。

```
CREATE TABLE hiveobstable (name STRING, score DOUBLE, classNo INT) STORED AS TEXTFILE
LOCATION 'obs://dli-test-021/data5' ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

  说明

  ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ： 表示每行记录通过',' 进行分隔。

  c. 查询hiveobstable表数据。

```
select * from hiveobstable;
```

图 2-18 查询结果

name	score	classNo
Jordon	88	23
Kim	87	25
Henry	76	26

  d. 插入表数据：

```
insert into hiveobstable VALUES('Aarn','98','25');
insert into hiveobstable VALUES('Adam','68','25');
```

  e. 查询表数据：

```
select * from hiveobstable;
```

图 2-19 查询结果

name	score	classNo
Adam	68	25
Aarn	98	25
Jordon	88	23
Kim	87	25
Henry	76	26

f. 在OBS桶“obs://dli-test-021/data5”目录下刷新后查询，生成了两个数据文件，分别对应新插入的数据。

图 2-20 查询结果

创建表字段为复杂数据格式的OBS表

a. 在OBS桶的“dli-test-021”根目录下创建数据文件目录“data6”。根据以下文件内容创建数据文件“test.txt”并上传到OBS的“obs://dli-test-021/data6”目录下。

```
Jordon,88-22,23:21
Kim,87-22,25:22
Henry,76-22,26:23
```

b. 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”。使用Hive语法创建OBS表，指定OBS文件路径为“obs://dli-test-021/data6”。

```
CREATE TABLE hiveobstable2 (name STRING, hobbies ARRAY<string>, address
map<string,string>) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data6'
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '-'
MAP KEYS TERMINATED BY ':';
```

说明

- ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' ：表示每条记录通过','进行分隔。
- COLLECTION ITEMS TERMINATED BY '-'：表示第二个字段hobbies是array形式，元素与元素之间通过'-'分隔。
- MAP KEYS TERMINATED BY ':'：表示第三个字段address是k-v形式，每组k-v内部由':'分隔。

c. 查询hiveobstable2表数据。

```
select * from hiveobstable2;
```

图 2-21 查询结果

name	hobbies	address
Jordon	["88-22"]	{"23":"21"}
Kim	["87-22"]	{"25":"22"}
Henry	["76-22"]	{"26":"23"}

- 创建OBS分区表

a. 在OBS桶的“dli-test-021”根目录下创建数据文件目录“data7”。

b. 登录DLI管理控制台，选择“SQL编辑器”，在SQL编辑器中“执行引擎”选择“spark”，“队列”选择已创建的SQL队列，数据库选择“testdb”。创建以classNo为分区列的OBS分区表，指定OBS路径“obs://dli-test-021/data7”。

```
CREATE TABLE IF NOT EXISTS hiveobstable3(name STRING, score DOUBLE) PARTITIONED BY
(classNo INT) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7' ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',';
```


注意

创建Hive语法的OBS分区表时，分区字段只能通过PARTITIONED BY指定，该分区字段不能出现在表名后的字段列表中。如下就是错误的示例：

CREATE TABLE IF NOT EXISTS hiveobstable3(name STRING, score DOUBLE, classNo INT) PARTITIONED BY (classNo) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7';

c. 插入表数据：

```
insert into hiveobstable3 VALUES('Aarn','98','25');
insert into hiveobstable3 VALUES('Adam','68','25');
```

d. 查询表数据：

```
select * from hiveobstable3 where classNo = 25;
```

图 2-22 查询结果

name	score	classNo
Adam	68	25
Aarn	98	25

e. 在OBS桶的“obs://dli-test-021/data7”目录下刷新后查询，新生成了分区目录“classno=25”，该分区目录下文件内容为新插入的表数据。

图 2-23 查询结果

f. 在OBS桶的“obs://dli-test-021/data7”目录下，创建分区目录“classno=24”。根据以下文件内容创建文件“test.txt”，并上传该文件到“obs://dli-test-021/data7/classno=24”目录下。

```
Jordon,88,24
Kim,87,24
Henry,76,24
```

g. 在SQL编辑器中执行以下命令，手工导入分区数据到OBS表“hiveobstable3”。

```
ALTER TABLE
  hiveobstable3
ADD
  PARTITION (classNo = 24) LOCATION 'obs://dli-test-021/data7/classNo=24';
```

h. 查询表“hiveobstable3”数据。

```
select * from hiveobstable3 where classNo = 24;
```

图 2-24 查询结果

name	score	classNo
Jordon	88	24
Kim	87	24
Henry	76	24

常见问题

- **问题一**：查询OBS分区表报错，报错信息如下：
  DLI.0005: There should be at least one partition pruning predicate on partitioned table `xxxx`.`xxxx`.;

  问题根因：查询OBS分区表时没有携带分区字段。

  解决方案：查询OBS分区表时，where条件中至少包含一个分区字段。
- **问题二**：使用DataSource语法指定OBS文件路径创建OBS表，insert数据到OBS表，显示作业运行失败，报：“DLI.0007: The output path is a file, don't support INSERT...SELECT” 错误。

  问题示例语句参考如下：
  CREATE TABLE testcsvdatasource (name string, id int) USING csv OPTIONS (path "**obs://dli-test-021/data/test.csv**");

  问题根因：创建OBS表指定的OBS路径为具体文件，导致不能插入数据。例如上述示例中的OBS路径为：**"obs://dli-test-021/data/test.csv**"。

  解决方案：使用DataSource语法创建OBS表指定的OBS文件路径改为文件目录即可，后续即可通过insert插入数据。上述示例，建表语句可以修改为：
  CREATE TABLE testcsvdatasource (name string, id int) USING csv OPTIONS (path "**obs://dli-test-021/data**");
- **问题三**：使用Hive语法创建OBS分区表时，提示语法格式不对。例如，如下使用Hive语法创建以classNo为分区的OBS表：
  CREATE TABLE IF NOT EXISTS testtable(name STRING, score DOUBLE, classNo INT) PARTITIONED BY (classNo) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7';

  问题根因：使用Hive语法创建OBS分区表时，分区字段不能出现在表名后的字段列表中，只能定义在PARTITIONED BY后。

  解决方案：使用Hive语法创建OBS分区表时，分区字段指定在PARTITIONED BY后。例如：
  CREATE TABLE IF NOT EXISTS testtable(name STRING, score DOUBLE) PARTITIONED BY (classNo INT) STORED AS TEXTFILE LOCATION 'obs://dli-test-021/data7';

2.2 在 DataArts Studio 开发 DLI SQL 作业

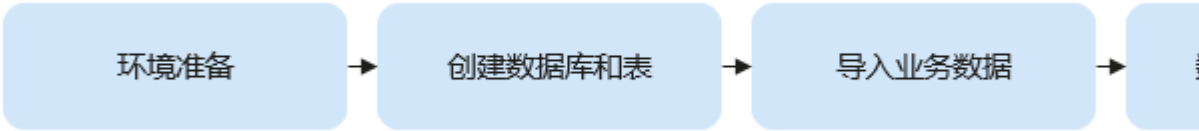
操作场景

华为云数据治理中心DataArts Studio提供了一站式数据治理平台，可以实现与DLI服务的对接，从而提供统一的数据集成、数据开发服务，方便企业对全部数据进行管控。

本节操作介绍在DataArts Studio开发DLI SQL作业的操作步骤。

开发流程

图 2-25 在 DataArts Studio 开发 DLI SQL 作业的流程图



1. 环境准备：准备执行作业所需的DLI资源和DataArts Studio资源。请参考**环境准备**。

2. 创建数据库和表：提交SQL脚本创建数据库和表。请参考**步骤1：创建数据库和表**。
3. 导入业务数据：提交SQL脚本导入业务数据。请参考**步骤2：业务数据的计算与处理**。
4. 数据查询与分析：提交SQL脚本分析业务数据，例如查询单日销售情况。请参考**步骤3：销售情况的查询与分析**。
5. 作业编排：将数据处理和数据分析脚本编排成一个pipeline。DataArts会按照编排好的pipeline顺序执行各个节点。请参考**步骤4：作业编排**。
6. 测试作业运行：测试作业运行。请参考**步骤5：测试作业运行**。
7. 设置作业调度与监控：设置作业调度属性与监控规则。请参考**步骤6：设置作业周期调度**和**相关操作**。

环境准备

- **DLI资源环境准备**
  - **配置DLI作业桶**

    使用DLI服务前需配置DLI作业桶，该桶用于存储DLI作业运行过程中产生的临时数据，例如：作业日志、作业结果。具体操作请参考：**配置DLI作业桶**。
  - **创建弹性资源池并添加SQL队列**

    弹性资源池为DLI作业运行提供所需的计算资源（CPU和内存），用于灵活应对业务对计算资源变化的需求。

    创建弹性资源池后，您可以在弹性资源池中创建多个队列，队列关联到具体的作业和数据处理任务，是资源池中资源被实际使用和分配的基本单元，即队列是执行作业所需的具体的计算资源。

    同一弹性资源池中，队列之间的计算资源支持共享。 通过合理设置队列的计算资源分配策略，可以提高计算资源利用率。

    具体操作请参考：**创建弹性资源池并添加队列**。
- **DataArts Studio资源环境准备**
  - **购买DataArts Studio实例**

    在使用DataArts Studio提交DLI作业前，需要先购买DataArts Studio实例。

    具体操作请参考**购买DataArts Studio基础包**。
  - **进入DataArts Studio实例空间**

    i. 购买完成DataArts Studio实例后，单击“进入控制台”。

    图 2-26 进入 DataArts Studio 实例控制台

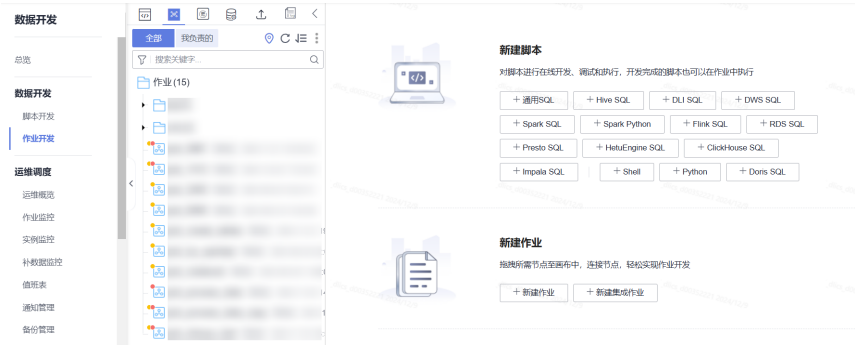
    ii. 单击“空间管理”，进入数据开发页面。

购买DataArts Studio实例的用户，系统将默认为其创建一个默认的工作空间“default”，并赋予该用户为管理员角色。您可以使用默认的工作空间，也可以参考本章节的内容创建一个新的工作空间。

如需创建新的空间请参考**创建并管理工作空间**。

图 2-27 进入 DataArts Studio 实例空间

图 2-28 进入 DataArts Studio 数据开发页面



步骤 1：创建数据库和表

步骤1 开发创建数据库和表的SQL脚本

数据库和表是SQL作业开发的基础，在执行作业前您需要根据业务场景定义数据库和表。

本节操作介绍提交SQL脚本创建数据库和表的操作步骤。

1. 在DataArts Studio数据开发页面，选择左侧导航栏的“数据开发 > 脚本开发”。
2. 单击“新建DLI SQL脚本”。

图 2-29 新建 DLI SQL 脚本

3. 在脚本编辑页面输入创建数据库和表的示例代码。

```SQL -- 创建数据库
CREATE DATABASE IF not EXISTS supermarket_db;-- 创建商品维表
CREATE TABLE IF not EXISTS supermarket*db.products ( product*id INT, -- 商品编号 productname STRING, -- 商品名称 category STRING, -- 商品类别 price DECIMAL(10,2) -- 单价 ) using parquet;-- 创建交易表
CREATE TABLE IF not EXISTS supermarket*db.transactions (transaction*id INT, -- 交易编号 productid INT, -- 商品编号 quantity INT, -- 数量 dt STRING -- 日期 ) using parquet partitioned by (dt);-- 创建销售分析表
CREATE TABLE IF not EXISTS supermarket*db.analyze (transaction*id INT, -- 交易编号 product*id INT, -- 商品编号 product*name STRING, -- 商品名称 quantity INT, -- 数量 dt STRING -- 日期 ) using parquet partitioned by (dt);
```

4. 单击“保存”，保存SQL脚本，本例定义脚本名称为 create_tables。
5. 单击“提交”按钮执行脚本创建数据库和表。

步骤2 创建SQL作业运行脚本

1. 在DataArts Studio数据开发页面，选择左侧导航栏的“数据开发 > 作业开发”。

图 2-30 新建作业

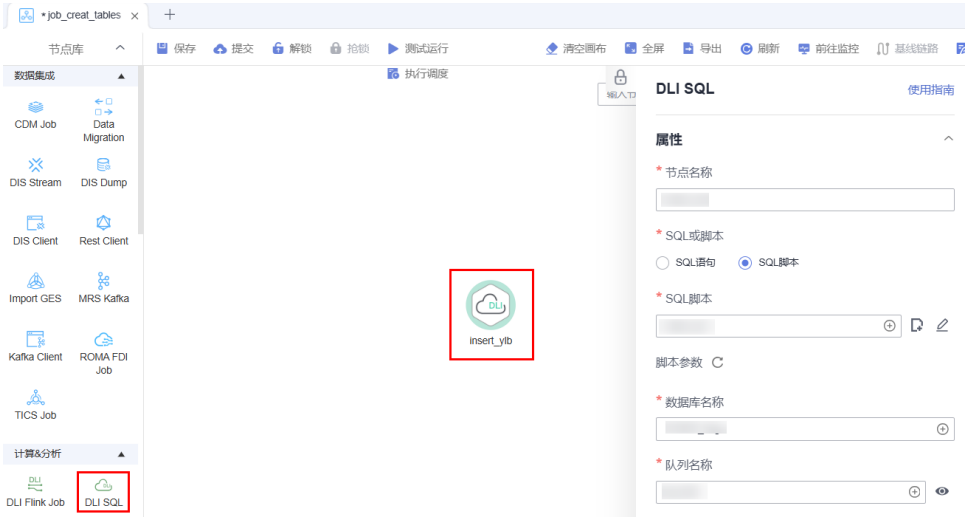
2. 编辑作业信息，本例定义SQL作业名称为 “job_create_tables”。

图 2-31 编辑作业信息

3. 在作业开发页面，拖动DLI SQL节点到画布中，并单击节点编辑属性。
   - SQL或脚本：本例选择“SQL脚本”。并选择**步骤2.2**中创建的脚本。
   - 数据库名称：选择SQL脚本中设置的数据库。
   - 队列名称：选择步骤**▪创建弹性资源池并添加SQL队列**中创建的SQL队列。

   更多属性参数配置请参考**DLI SQL属性参数说明**。

图 2-32 编辑 DLI SQL 节点属性



4. 属性编辑完成后，单击“保存”，保存属性配置信息。

步骤3 配置作业调度

由于创建库表只需要执行一次，所以本示例只设置为单次调度。

1. 鼠标左键单击作业画布空白处。
2. 单击“调度配置”，选择“单次调度”（该作业只会被调度一次，后续不会再被自动调度）。

图 2-33 配置作业调度

3. 完成调度配置后单击“执行调度”。
   单击“前往运维中心”可以查看作业运行状况。

----结束

步骤 2：业务数据的计算与处理

步骤1 开发导入业务数据的SQL脚本

本节操作介绍提交SQL脚本导入业务数据的操作步骤。

1. 在DataArts Studio数据开发页面，选择左侧导航栏的“数据开发 > 脚本开发”。
2. 单击“新建DLI SQL脚本”。

图 2-34 新建 DLI SQL 脚本

3. 在脚本编辑页面输入分析数据的示例代码。

```
SQL -- 实际业务中数据一般来自其他数据源，本示例简化了数据入库逻辑，模拟插入商品数据。
```



```
INSERT INTO supermarketdb.products (productid, productname, category, price) VALUES
(1001, '洗发水', '日用品', 39.90),
(1002, '牙膏', '日用品', 15.90),
(1003, '方便面', '食品', 4.50),
(1004, '可乐', '饮料', 3.50);
-- 实际业务中数据一般来自其他数据源，本示例简化了数据入库逻辑，模拟插入交易记录。
INSERT INTO supermarketdb.transactions (transactionid, productid, quantity, dt) VALUES (1, 1001, 50, '2024-11-01'), -- 销售50瓶洗发水
(2, 1002, 100, '2024-11-01'), -- 销售100支牙膏 (3, 1003, 30, '2024-11-02'), -- 销售30包方便面
(4, 1004, 24, '2024-11-02'); -- 销售24瓶可乐
-- 模拟超市业务分析，查询某个商品的交易记录
INSERT INTO supermarketdb.analyze SELECT t.transactionid, t.productid, p.productname, t.quantity, t.dt
FROM supermarketdb.transactions t
JOIN supermarketdb.products p ON t.productid = p.productid
WHERE t.dt = '2024-11-01';
```

4. 单击“保存”，保存SQL脚本，本例定义脚本名称为 job_process_data。
5. 单击“提交”执行脚本。

步骤2 创建SQL作业

1. 在DataArts Studio数据开发页面，选择左侧导航栏的“数据开发 > 作业开发”。

图 2-35 新建作业

2. 编辑作业信息，本例定义SQL作业名称为 “job_process_data”。

图 2-36 编辑作业信息

3. 在作业开发页面，拖动DLI SQL节点到画布中，并单击节点编辑属性。
   - SQL或脚本：本例选择“SQL脚本”。并选择**步骤1**中创建的脚本。
   - 数据库名称：选择SQL脚本中设置的数据库。
   - 队列名称：选择步骤**▪创建弹性资源池并添加SQL队列**中创建的SQL队列。
   - 环境变量："DLI环境变量" 为可选项。

     本例中添加的参数含义如下：

     spark.sql.optimizer.dynamicPartitionPruning.enabled = true

     - 该配置项用于启用或禁用动态分区修剪。在执行SQL查询时，动态分区修剪可以帮助减少需要扫描的数据量，提高查询性能。
     - 配置为true时，代表启用动态分区修剪，SQL会在查询中自动检测并删除那些不满足WHERE子句条件的分区，适用于在处理具有大量分区的表时。
     - 如果SQL查询中包含大量的嵌套left join操作，并且表有大量的动态分区时，这可能会导致在数据解析时消耗大量的内存资源，导致Driver节点的内存不足，并触发频繁的Full GC。
     - 在这种情况下，可以配置该参数为false即禁用动态分区修剪优化，有助于减少内存使用，避免内存溢出和频繁的Full GC。

     但禁用此优化可能会降低查询性能，禁用后Spark将不会自动修剪掉那些不满足条件的分区。

更多属性参数配置请参考**DLI SQL属性参数说明**。

图 2-37 编辑 DLI SQL 节点属性

4. 属性编辑完成后，单击“保存”，保存属性配置信息。

----结束

步骤 3：销售情况的查询与分析

开发数据分析与处理的SQL脚本

本节操作介绍提交SQL脚本分析数据的操作步骤。

1. 在DataArts Studio数据开发页面，选择左侧导航栏的“数据开发 > 脚本开发”。
2. 单击“新建DLI SQL脚本”。

图 2-38 新建 DLI SQL 脚本

3. 在脚本编辑页面输入分析数据的示例代码。

```
-- 查询单日销售情况
SELECT transaction_id, productid, productname, quantity, dt FROM supermarket_db.analyze WHERE dt = '2024-11-01';
```

4. 单击“保存”，保存SQL脚本，本例定义脚本名称为 select_analyze_data。
5. 单击“提交”执行脚本。

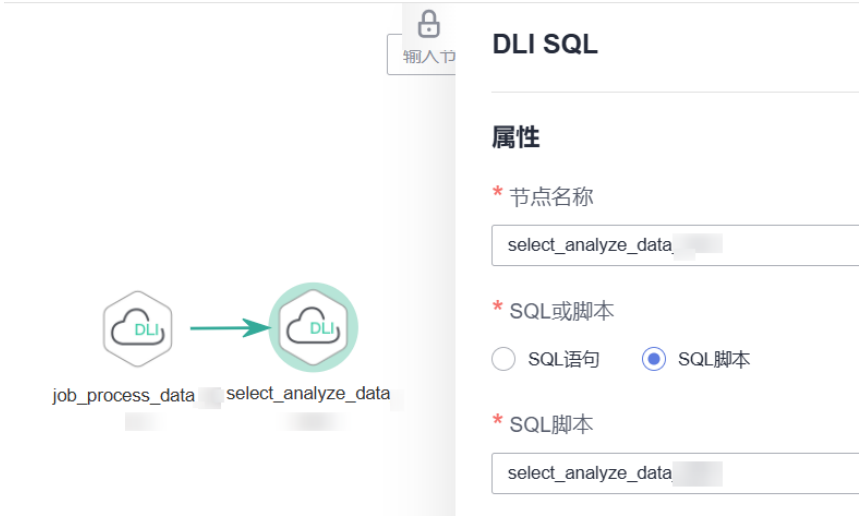
步骤 4：作业编排

1. 在作业 “job_process_data”中新建一个DLI SQL节点 "select_analyze_data"。并单击节点编辑属性。

- SQL或脚本：本例选择“SQL脚本”。并选择**步骤1**中创建的脚本。
- 数据库名称：选择SQL脚本中设置的数据库。
- 队列名称：选择步骤**▪创建弹性资源池并添加SQL队列**中创建的SQL队列。

更多属性参数配置请参考**DLI SQL属性参数说明**。

图 2-39 编辑 DLI SQL 节点属性



2. 属性编辑完成后，单击“保存”，保存属性配置信息。
3. 将这两个节点编排成一个pipeline。DataArt会按照编排好的pipeline顺序执行各个节点。然后单击左上角 “保存” 和 “提交” 。

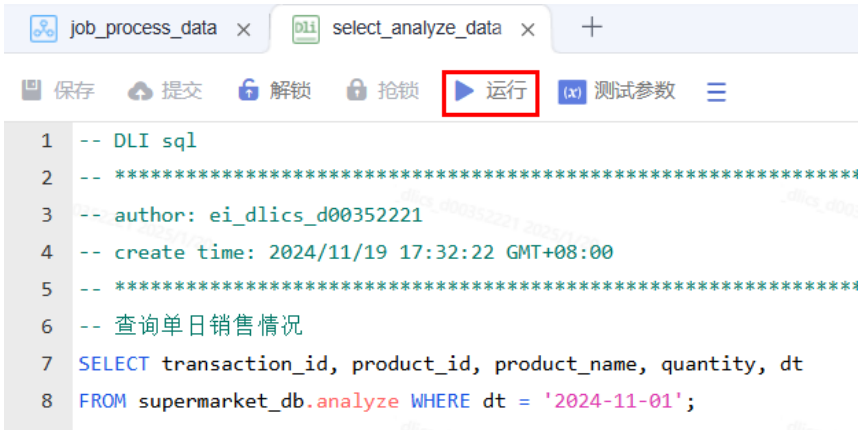
步骤 5：测试作业运行

作业编排完成后，单击“测试运行”，测试运行作业。

运行结束后，可打开“select_analyze_data”SQL脚本，单击“运行”，查询分析销售明细。

如果查询结果符合预期，可以继续执行**步骤6：设置作业周期调度**设置作业周期调度。

图 2-40 执行 select_analyze_data 脚本



步骤 6：设置作业周期调度

1. 在DataArts Studio数据开发页面，选择左侧导航栏的“数据开发 > 作业开发”。
2. 双击“job_process_data”。
3. 单击右侧导航栏中的 “调度配置”。
4. 选择周期调度，并设置调度属性。

   本例中，该作业的调度策略从2024/11/22 10:15:00开始生效，且首次执行调度的时间是2024/11/22 10:20:00，调度周期间隔1天，即后续每天10:20:00 AM会自动调度这些这个作业，会按照编排好的pipeline顺序执行作业中的每个节点。

   图 2-41 配置作业调度

5. 依次单击 “保存” 、“提交” 和 “执行调度” 按钮，即可完成作业周期调度配置。

了解更多作业调度设置请参考**调度作业**。

相关操作

- **设置作业监控**

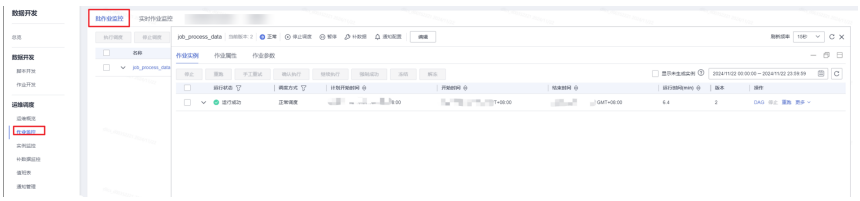
  DataArts Studio提供了对批处理作业的状态进行监控的能力。

  批作业是由一个或多个节点组成的流水线，以流水线作为一个整体被调度。

  您可以在 DataArts 左侧导航栏 “作业监控 > 批作业监控” 页面查看批处理作业的调度状态、调度周期、调度开始时间等信息。

  了解更多**DataArts运维调度作业监控**。

  图 2-42 设置作业监控

- **设置实例监控**

  作业每次运行，都会对应产生一次作业实例记录。

  在数据开发模块控制台的左侧导航栏，选择 “运维调度”，进入实例监控列表页面，您可以在该页面中查看作业的实例信息，并根据需要对实例进行更多操作。

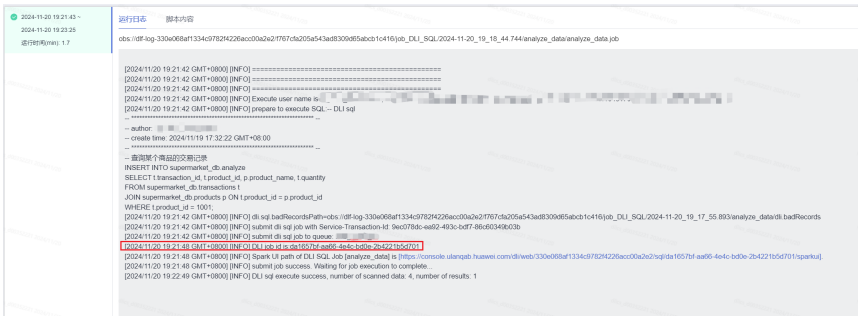
  了解更多**实例监控**。

常见问题

- **如果 DataArts 作业失败，且 DataArts 提供的日志不够详细，怎么办？还能从哪里找更具体的日志?**

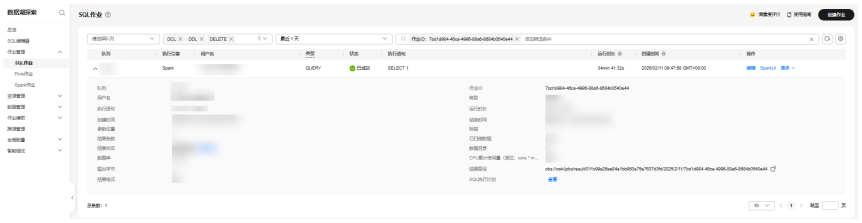
  您可以可通过 DataArts 的日志找到DLI job id，然后根据DLI job id 在DLI控制台中找到具体的作业。

  图 2-43 监控日志

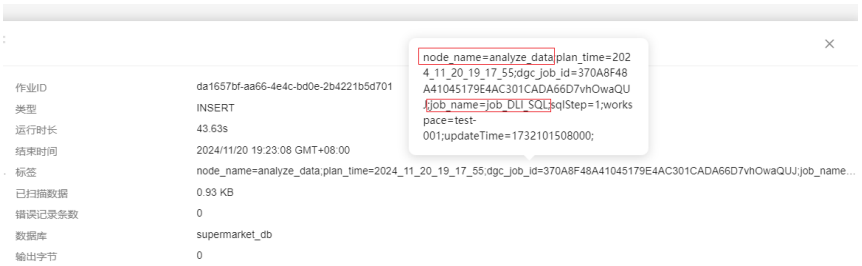
  在DLI控制台中找到具体的作业，单击归档日志即可查看详细日志：

  图 2-44 输入作业 ID

  可以通过DataArts的node*name*或*job*name在DLI 控制台搜索作业：

  图 2-45 nodename 或 jobname

- **如果在运行复杂DLI作业时遇到权限类报错，应该怎么办?**

  使用DLI的过程中需要与其他云服务协同工作，因此需要您将部分服务的操作权限委托给DLI服务，确保DLI具备基本使用的权限，让DLI服务以您的身份使用其他云服务，代替您进行一些资源运维工作。

  了解更多：**配置DLI云服务委托权限**

2.3 在 Spark SQL 作业中使用 UDF

操作场景

DLI支持用户使用Hive UDF（User Defined Function，用户定义函数）进行数据查询等操作，UDF只对单行数据产生作用，适用于一进一出的场景。

约束限制

- 在DLI Console上执行UDF相关操作时，需要使用自建的SQL队列。
- 跨账号使用UDF时，除了创建UDF函数的用户，其他用户如果需要使用时，需要先进行授权才可使用对应的UDF函数。授权操作参考如下：

  登录DLI管理控制台，选择“ 数据管理 > 程序包管理”页面，选择对应的UDF Jar包，单击“操作”列中的“权限管理”，进入权限管理页面，单击右上角“授权”，勾选对应权限。
- 自定义函数中引用static类或接口时，必须要加上“try catch”异常捕获，否则可能会造成包冲突，导致函数功能异常。

环境准备

在进行UDF开发前，请准备以下开发环境。

表 2-2 UDF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。

开发流程

DLI下UDF函数开发流程参考如下：

图 2-46 开发流程

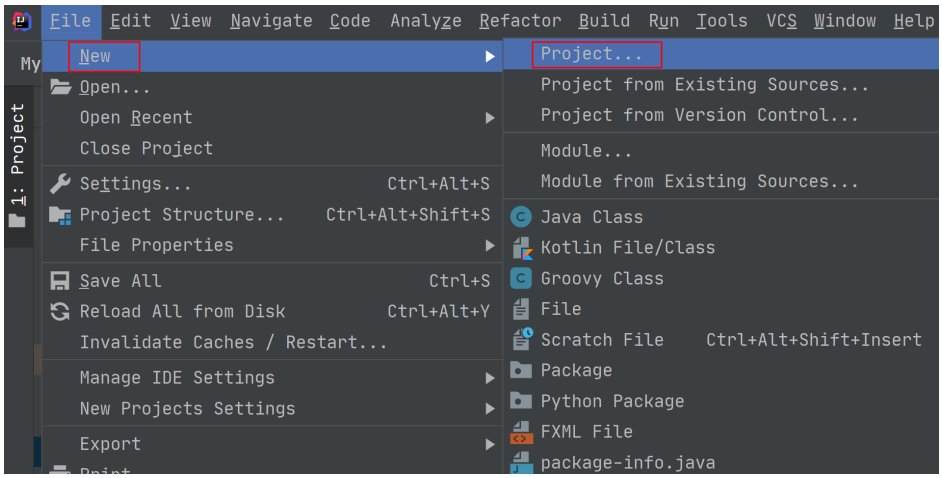
表 2-3 开发流程说明

序号	阶段	操作界面	说明
1	新建Maven工程，配置pom文件	IntelliJ IDEA	参考**操作步骤**说明，编写UDF函数代码。
2	编写UDF函数代码		
3	调试，编译代码并导出Jar包		
4	上传Jar包到OBS	OBS控制台	将生成的UDF函数Jar包文件上传到OBS目录下。
5	创建DLI的UDF函数	DLI控制台	在DLI控制台的SQL作业管理界面创建使用的UDF函数。
6	验证和使用DLI的UDF函数	DLI控制台	在DLI作业中使用创建的UDF函数。

操作步骤

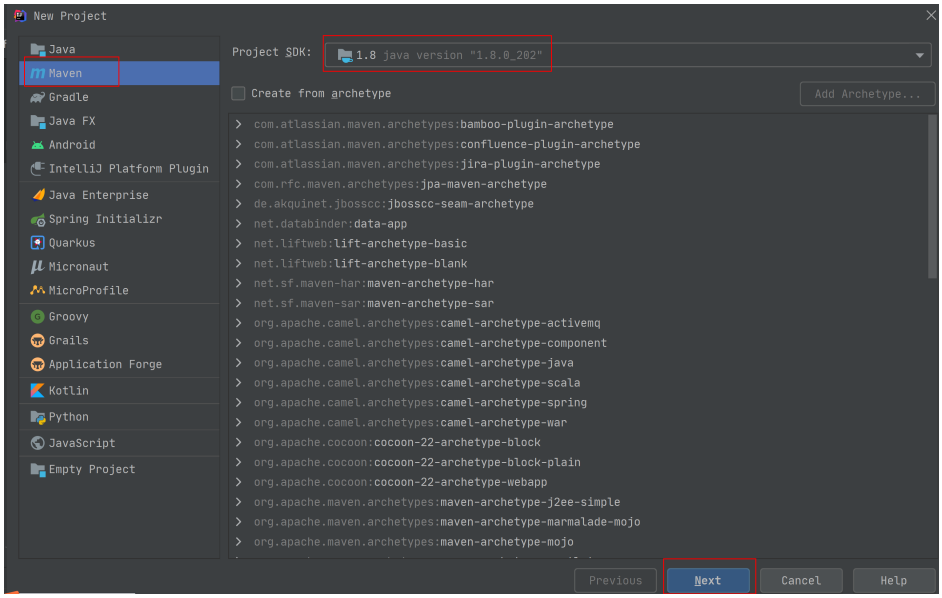
1. 新建Maven工程，配置pom文件。以下通过IntelliJ IDEA 2020.2工具操作演示。
   a. 打开IntelliJ IDEA，选择“File > New > Project”。

   图 2-47 新建 Project

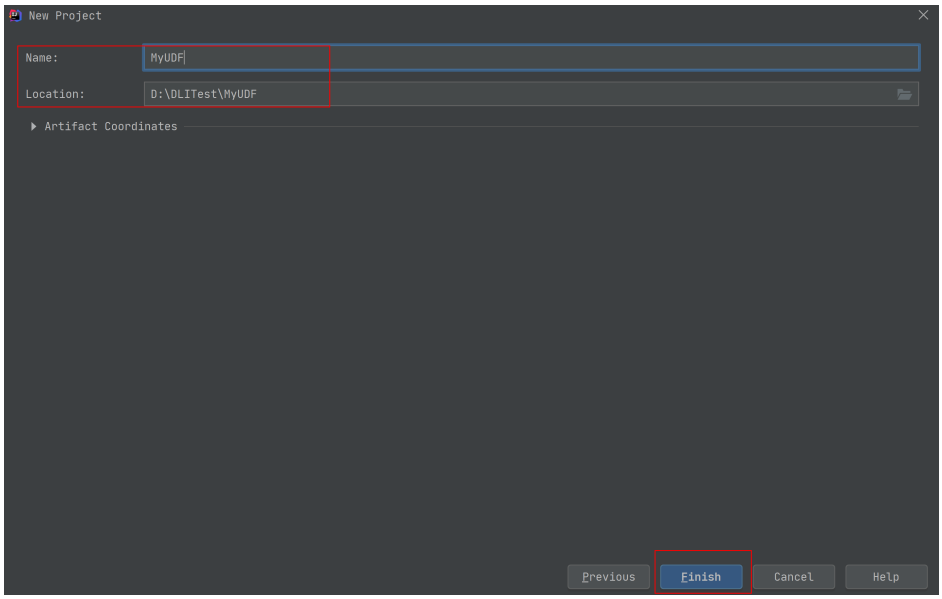
   b. 选择Maven，Project SDK选择1.8，单击“Next”。

图 2-48 选择 Maven



c. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。

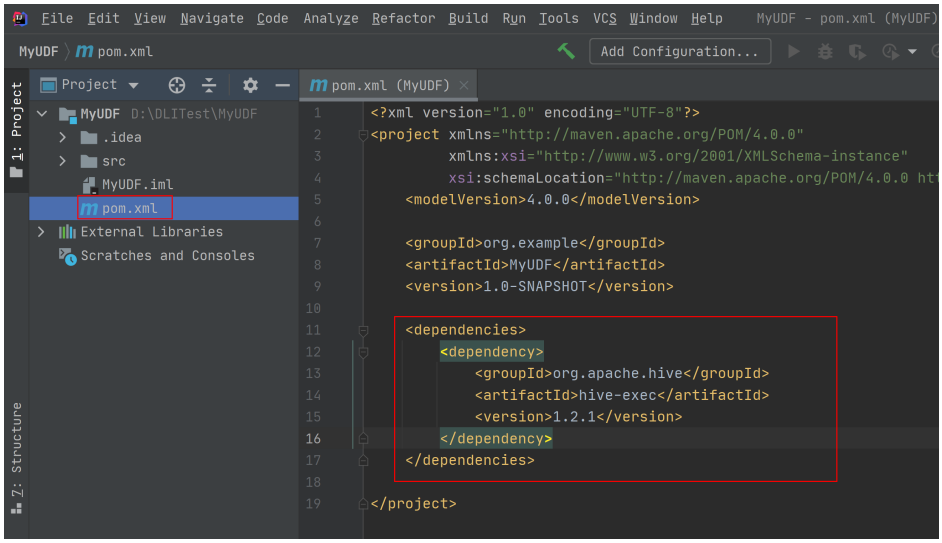
图 2-49 创建工程



d. 在pom.xml文件中添加如下配置。

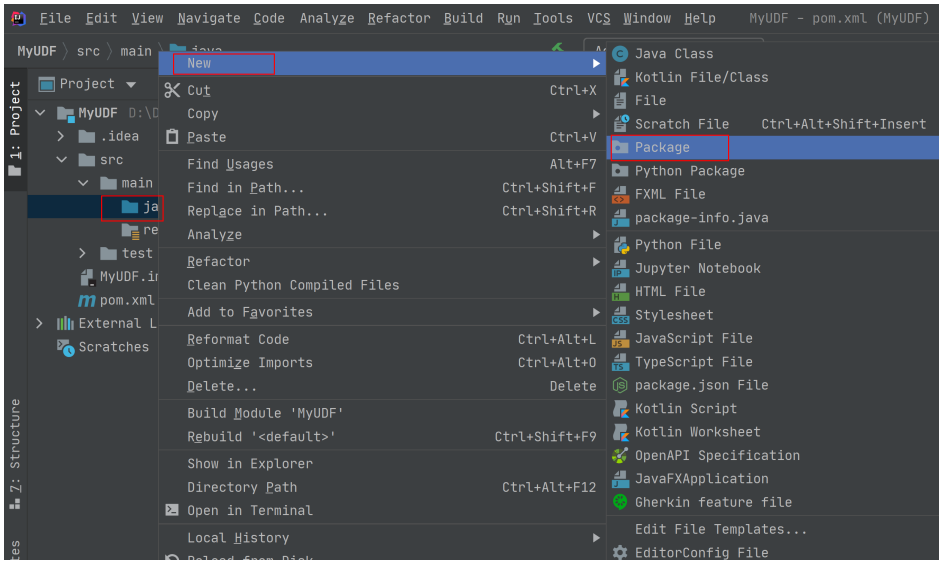
```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

图 2-50 pom 文件中添加配置



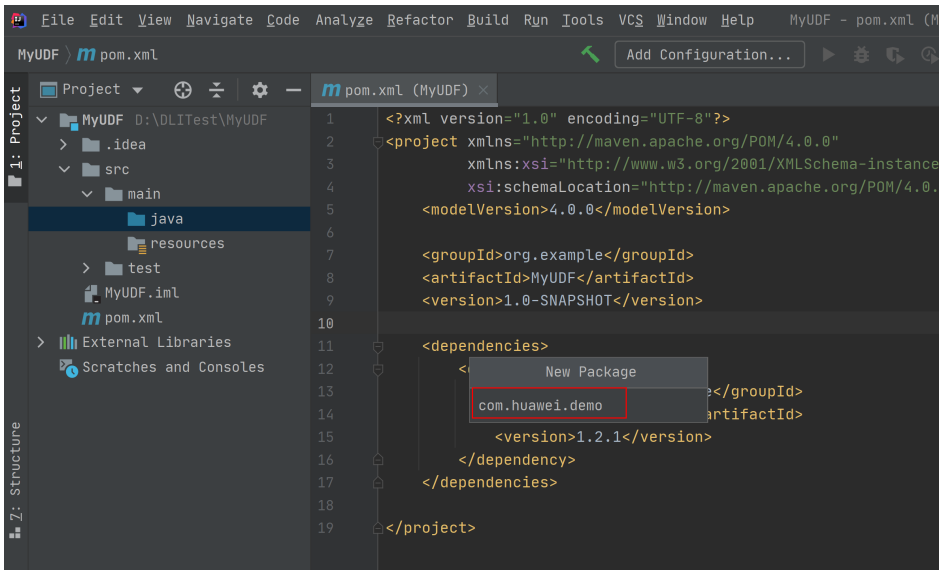
e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

图 2-51 新建 Package 和类文件



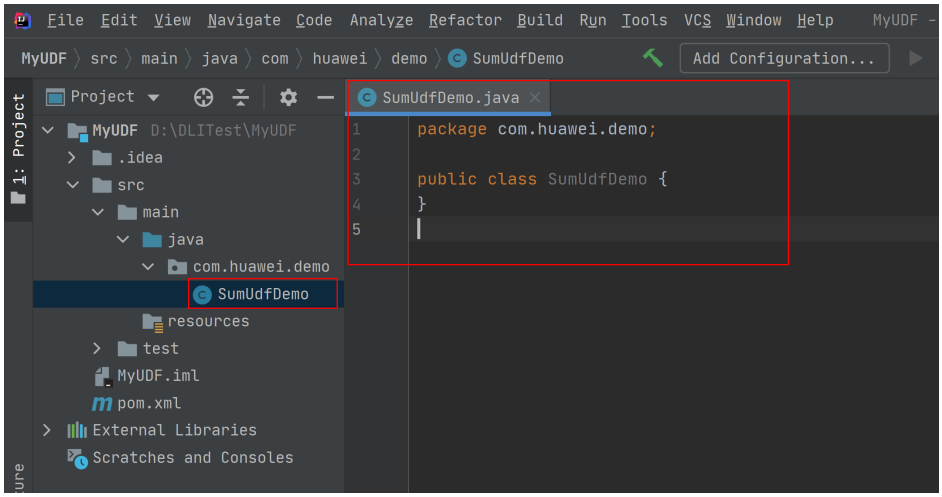
Package根据需要定义，本示例定义为：“com.huawei.demo”，完成后回车。

图 2-52 自定义 Package



在包路径下新建Java Class文件，本示例定义为：SumUdfDemo。

图 2-53 新建 Java Class 文件



2. 编写UDF函数代码。UDF函数实现，主要注意以下几点：
   a. 自定义UDF需要继承org.apache.hadoop.hive.ql.exec.UDF。
   b. 需要实现evaluate函数，evaluate函数支持重载。

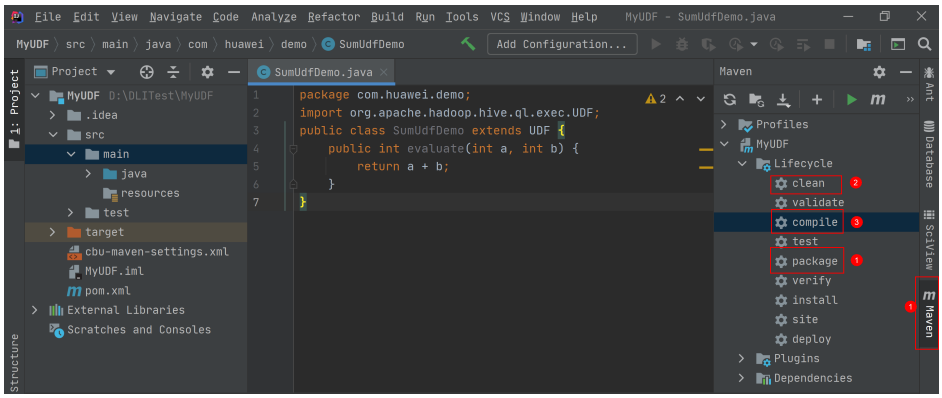
   详细UDF函数实现，可以参考如下样例代码：

```
package com.huawei.demo;
import org.apache.hadoop.hive.ql.exec.UDF;
 public class SumUdfDemo extends UDF {
   public int evaluate(int a, int b) {
    return a + b;
 }
}
```

3. 编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。
   a. 单击工具右侧的“Maven”，参考下图分别单击“clean”、“compile”对代码进行编译。

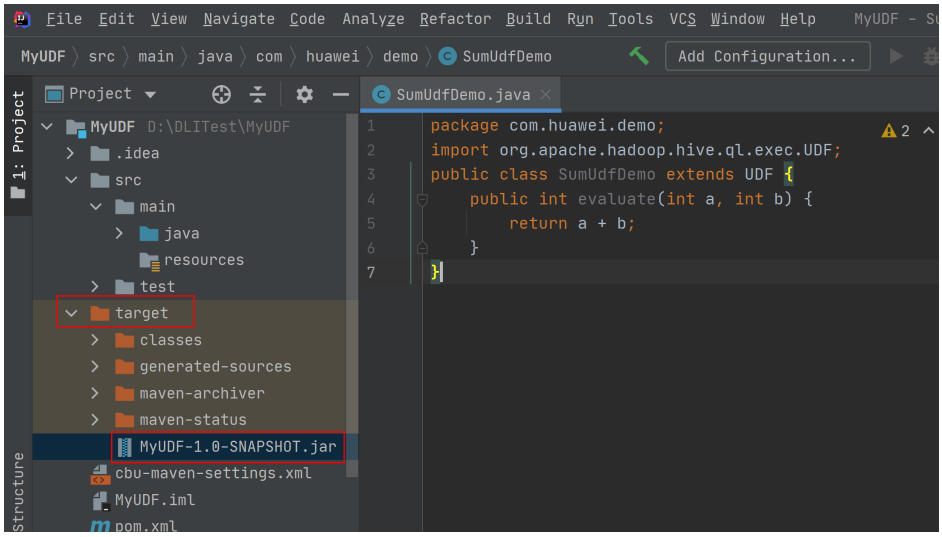
编译成功后，单击“package”对代码进行打包。

图 2-54 编译打包



打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\DLITest\MyUDF\target”下名为“MyUDF-1.0-SNAPSHOT.jar”。

图 2-55 生成 Jar 包



4. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

   说明

   Jar包文件上传的OBS桶所在的区域需与DLI的队列区域相同，不可跨区域执行操作。

5. （可选）可以将Jar包文件上传到DLI的程序包管理中，方便后续统一管理。
   a. 登录DLI管理控制台，单击“数据管理 > 程序包管理”。
   b. 在“程序包管理”页面，单击右上角的“创建”创建程序包。
   c. 在“创建程序包”对话框，配置以下参数。
      i. 包类型：选择“JAR”。
      ii. OBS路径：程序包所在的OBS路径。
      iii. 分组设置和组名称根据情况选择设置，方便后续识别和管理程序包。
   d. 单击“确定”，完成创建程序包。

6. 创建UDF函数。
   a. 登录DLI管理控制台，单击“SQL编辑器”，执行引擎选择“spark”，选择已创建的SQL队列和数据库。

      图 2-56 选择队列和数据库

   b. 在SQL编辑区域输入下列命令创建UDF函数，单击“执行”提交创建。

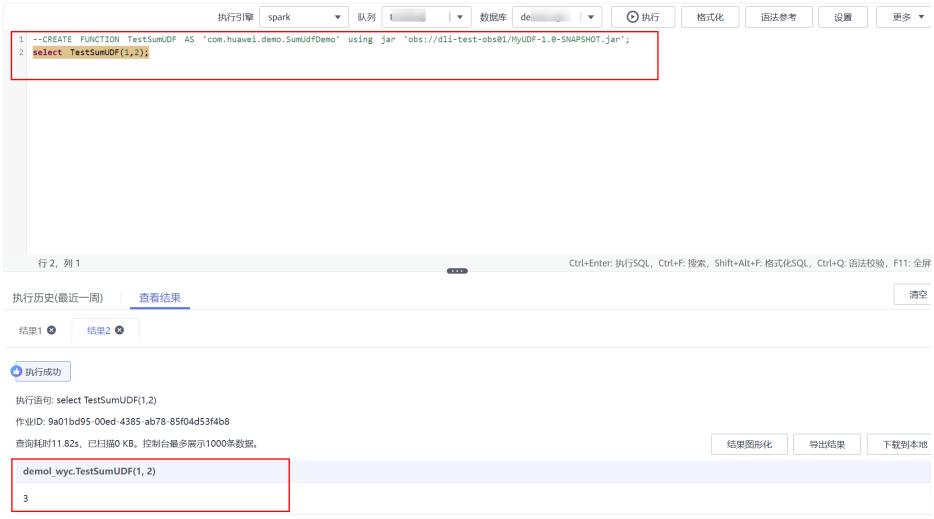
```
CREATE FUNCTION TestSumUDF AS 'com.huawei.demo.SumUdfDemo' using jar 'obs://dli-test-obs01/MyUDF-1.0-SNAPSHOT.jar';
```

7. 重启原有SQL队列，使得创建的Function生效。
   a. 登录数据湖探索管理控制台，选择“队列管理”，在对应“SQL队列”类型作业的“操作”列，单击“重启”。
   b. 在“重启队列”界面，选择“确定”完成队列重启。
8. 使用UDF函数。

   在查询语句中使用6中创建的UDF函数:

```
select TestSumUDF(1,2);
```

   图 2-57 执行结果

9. （可选）删除UDF函数。

   如果不再使用UDF函数，可执行以下语句删除该函数:

```
Drop FUNCTION TestSumUDF;
```

2.4 在 Spark SQL 作业中使用 UDTF

操作场景

DLI支持用户使用Hive UDTF（User-Defined Table-Generating Functions）自定义表值函数，UDTF用于解决一进多出业务场景，即其输入与输出是一对多的关系，读入一行数据，输出多个值。

约束限制

- 在DLI Console上执行UDTF相关操作时，需要使用自建的SQL队列。
- 不同的IAM用户使用UDTF时，除了创建UDTF函数的用户，其他用户如果需要使用时，需要先进行授权才可使用对应的UDTF函数。授权操作参考如下：

  登录DLI管理控制台，选择“ 数据管理 > 程序包管理”页面，选择对应的UDTF Jar包，单击“操作”列中的“权限管理”，进入权限管理页面，单击右上角“授权”，勾选对应权限。
- 自定义函数中引用static类或接口时，必须要加上“try catch”异常捕获，否则可能会造成包冲突，导致函数功能异常。

环境准备

在进行UDTF开发前，请准备以下开发环境。

表 2-4 UDTF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。

开发流程

DLI下UDTF函数开发流程参考如下：

图 2-58 UDTF 开发流程

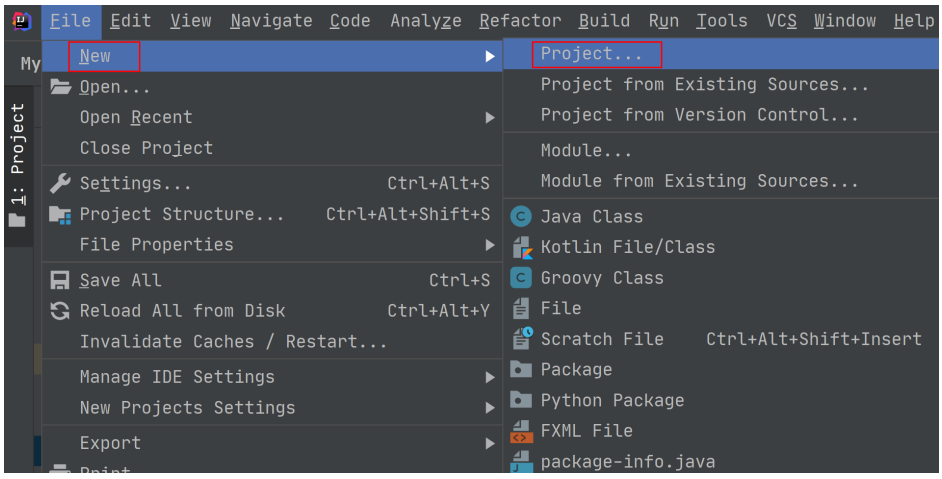
表 2-5 开发流程说明

序号	阶段	操作界面	说明
1	新建Maven工程，配置pom文件	IntelliJ IDEA	参考**操作步骤**说明，编写UDTF函数代码。
2	编写UDTF函数代码		
3	调试，编译代码并导出Jar包		
4	上传Jar包到OBS	OBS控制台	将生成的UDTF函数Jar包文件上传到OBS目录下。
5	创建DLI的UDTF函数	DLI控制台	在DLI控制台的SQL作业管理界面创建使用的UDTF函数。
6	验证和使用DLI的UDTF函数	DLI控制台	在DLI作业中使用创建的UDTF函数。

操作步骤

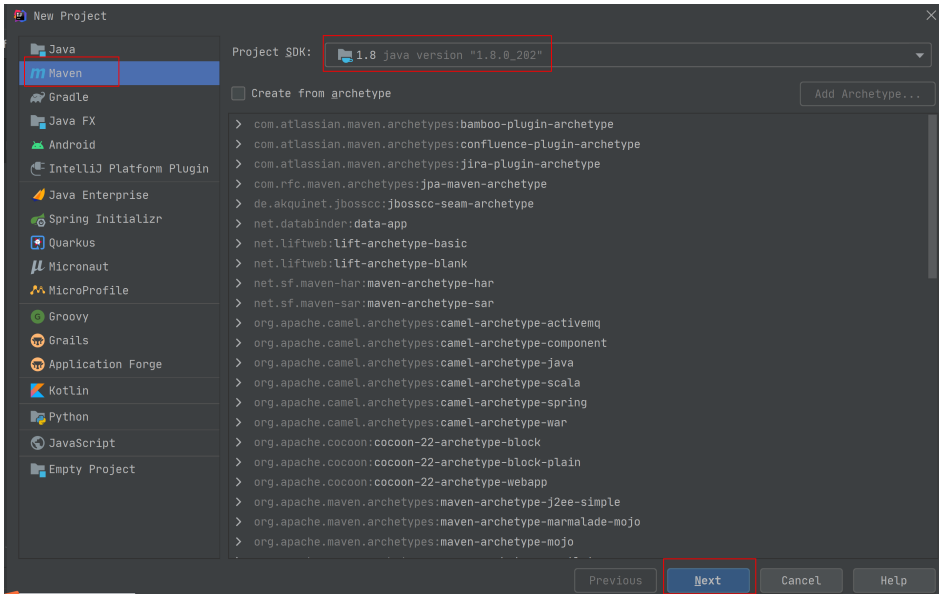
1. 新建Maven工程，配置pom文件。以下通过IntelliJ IDEA 2020.2工具操作演示。
   a. 打开IntelliJ IDEA，选择“File > New > Project”。

   图 2-59 新建 Project

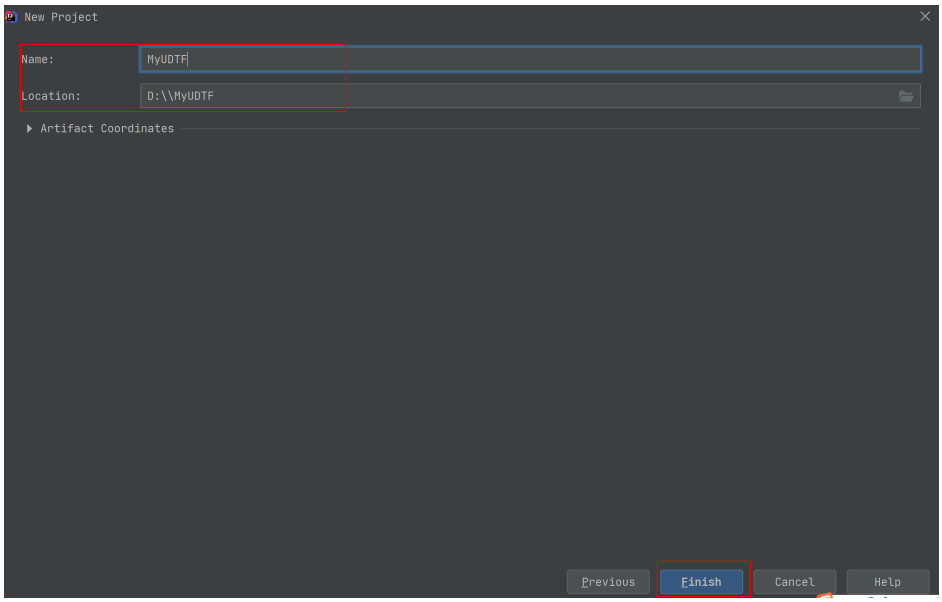
   b. 选择Maven，Project SDK选择1.8，单击“Next”。

图 2-60 选择 Maven



c. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。

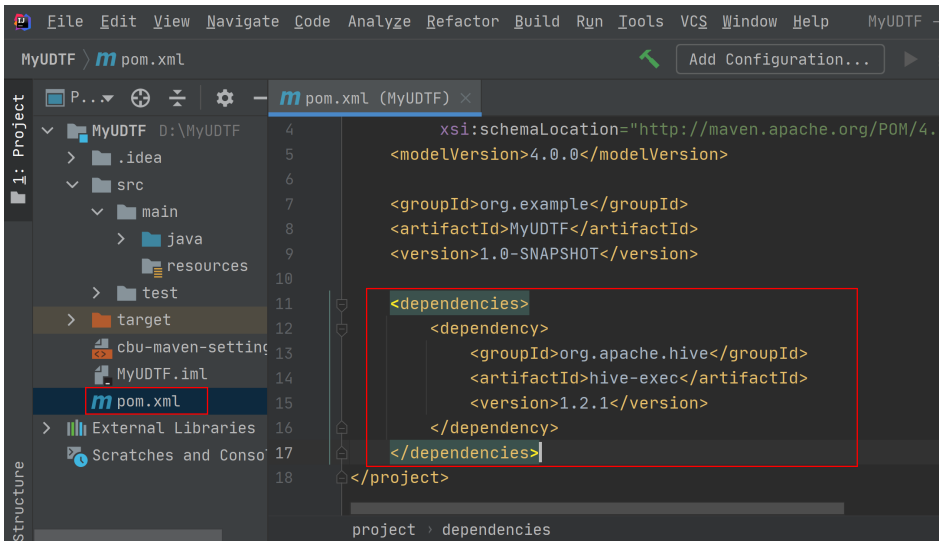
图 2-61 创建工程



d. 在pom.xml文件中添加如下配置。

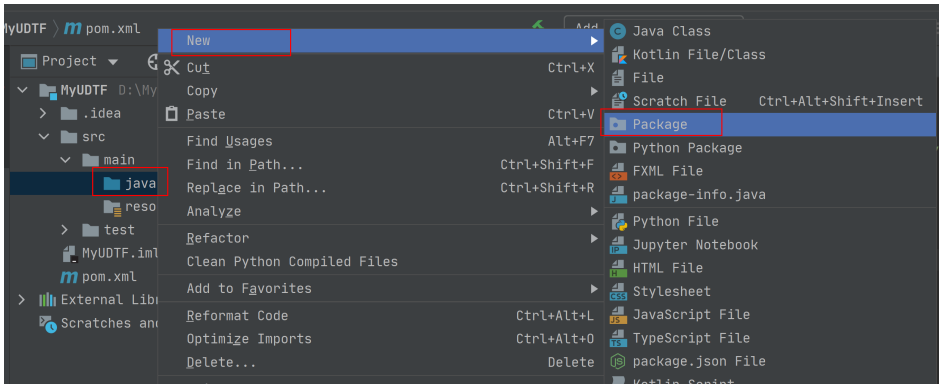
```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

图 2-62 pom 文件中添加配置



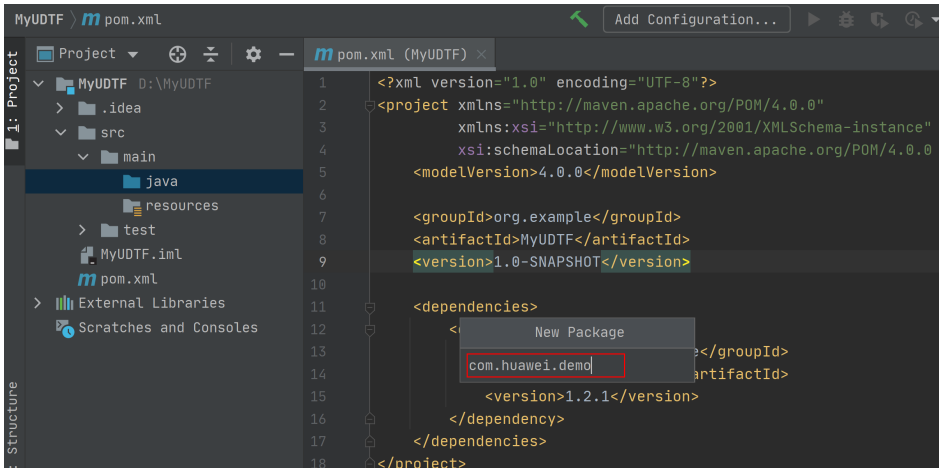
e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

图 2-63 新建 Package 和类文件



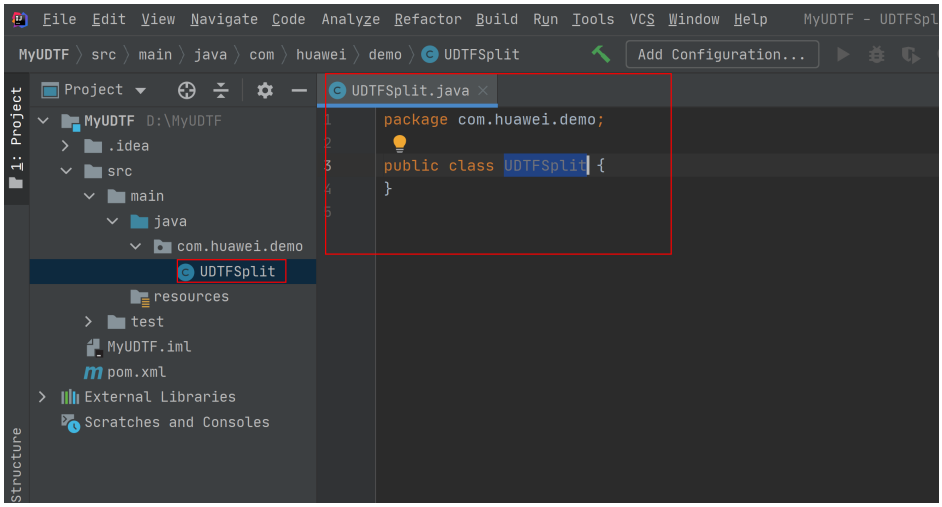
Package根据需要定义，本示例定义为：“com.huawei.demo”，完成后回车。

图 2-64 自定义 Package



在包路径下新建Java Class文件，本示例定义为：UDTFSplit。

图 2-65 新建 Java Class 文件



2. 编写UDTF函数代码。完整样例代码请参考**样例代码**。

   UDTF的类需要继承“org.apache.hadoop.hive.ql.udf.generic.GenericUDTF”，实现initialize，process，close三个方法。

   a. UDTF首先会调用initialize方法，此方法返回UDTF的返回行的信息，如，返回个数，类型等。

   b. 初始化完成后，会调用process方法，真正处理在process函数中，在process中，每一次forward()调用产生一行。

      如果产生多列可以将多个列的值放在一个数组中，然后将该数组传入到forward()函数。

```
public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }
```

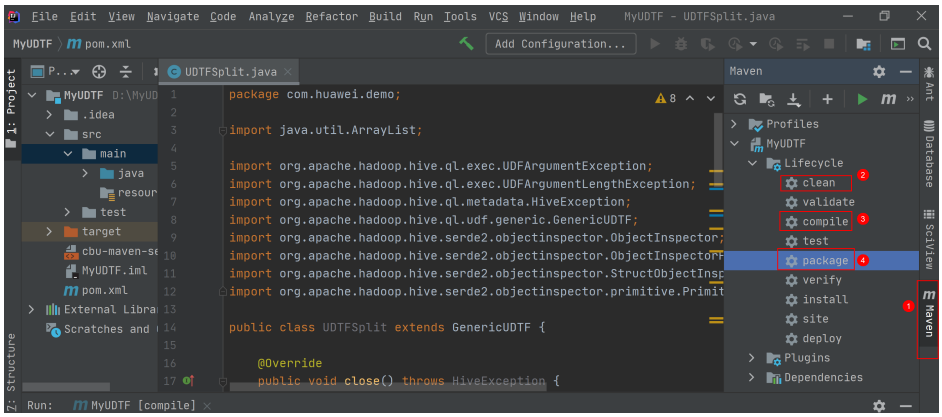
   c. 最后调用close方法，对需要清理的方法进行清理。

3. 编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。

   a. 单击工具右侧的“Maven”，参考下图分别单击“clean”、“compile”对代码进行编译。

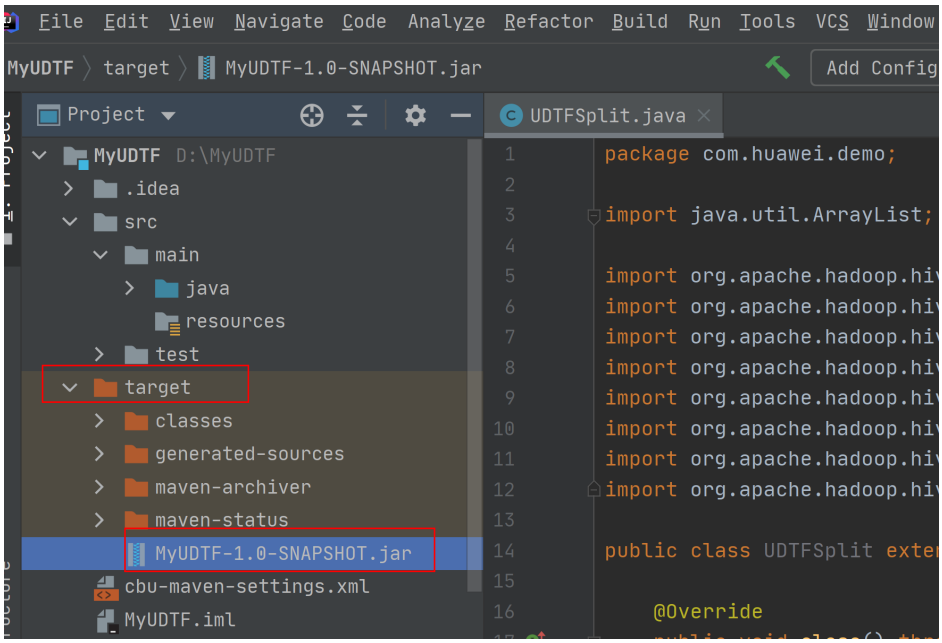
      编译成功后，单击“package”对代码进行打包。

图 2-66 编译打包



打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\MyUDTF\target”下名为“MyUDTF-1.0-SNAPSHOT.jar”。

图 2-67 生成 Jar 包



4. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

   说明

   Jar包文件上传的OBS桶所在的区域需与DLI的队列区域相同，不可跨区域执行操作。

5. （可选）可以将Jar包文件上传到DLI的程序包管理中，方便后续统一管理。
   a. 登录DLI管理控制台，单击“数据管理 > 程序包管理”。
   b. 在“程序包管理”页面，单击右上角的“创建”创建程序包。
   c. 在“创建程序包”对话框，配置以下参数。
      i. 包类型：选择“JAR”。
      ii. OBS路径：程序包所在的OBS路径。
      iii. 分组设置和组名称根据情况选择设置，方便后续识别和管理程序包。

d. 单击“确定”，完成创建程序包。

图 2-68 创建程序包

6. 创建DLI的UDTF函数。

a. 登录DLI管理控制台，单击“SQL编辑器”，执行引擎选择“spark”，选择已创建的SQL队列和数据库。

图 2-69 选择队列和数据库

b. 在SQL编辑区域输入实际上传Jar包的路径创建UDTF函数，单击“执行”提交创建。

```
CREATE FUNCTION mytestsplit AS 'com.huawei.demo.UDTFSplit' using jar 'obs://dli-test-obs01/MyUDTF-1.0-SNAPSHOT.jar';
```

7. 重启原有SQL队列，使得创建的UDTF函数生效。

a. 登录数据湖探索管理控制台，选择“资源管理 > 队列管理”，在对应“SQL队列”类型作业的“操作”列，单击“重启”。

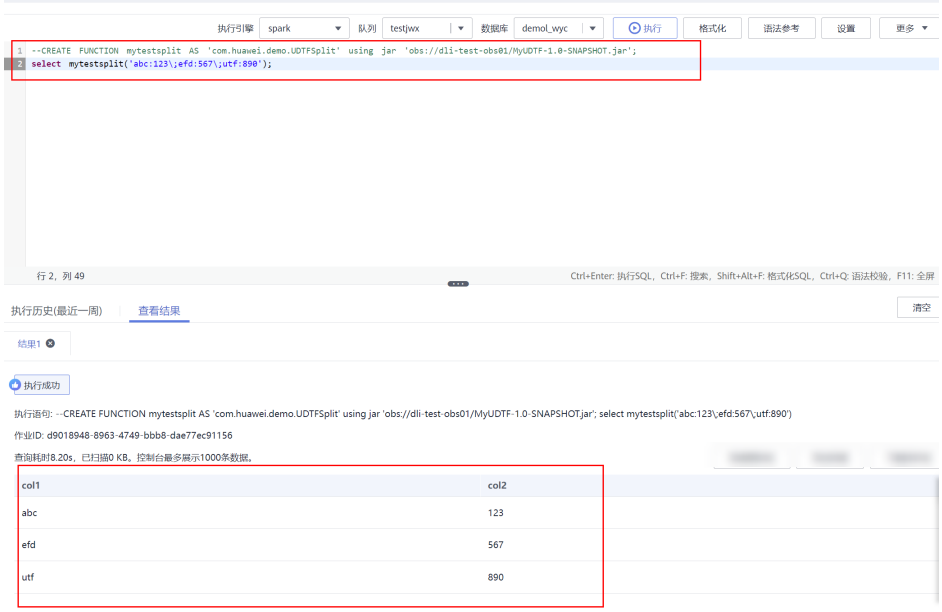
b. 在“重启队列”界面，选择“确定”完成队列重启。

8. 验证和使用创建的UDTF函数。

在查询语句中使用6中创建的UDTF函数，如：

```
select mytestsplit('abc:123\;efd:567\;utf:890');
```


图 2-70 执行结果



9. （可选）删除UDTF函数。

   如果不再使用该Function，可执行以下语句删除UDTF函数:

   ```
   Drop FUNCTION mytestsplit;
   ```

样例代码

UDTFSplit.java完整的样例代码参考如下所示：

```
package com.huawei.demo;
import java.util.ArrayList;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentLengthException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;

public class UDTFSplit extends GenericUDTF {

    @Override
    public void close() throws HiveException {
        // TODO Auto-generated method stub

    }

    @Override
    public void process(Object[] args) throws HiveException {
        // TODO Auto-generated method stub
        if(args.length == 0){
            return;
        }
        String input = args[0].toString();
        if(StringUtils.isEmpty(input)){
            return;
        }
        String[] test = input.split(";");
```



```
        for (int i = 0; i < test.length; i++) {
            try {
                String[] result = test[i].split(":");
                forward(result);
            } catch (Exception e) {
                continue;
            }
        }

    }

    @Override
    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException {
        if (args.length != 1) {
            throw new UDFArgumentLengthException("ExplodeMap takes only one argument");
        }
        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE) {
            throw new UDFArgumentException("ExplodeMap takes string as a parameter");
        }

        ArrayList<String> fieldNames = new ArrayList<String>();
        ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();
        fieldNames.add("col1");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        fieldNames.add("col2");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);

        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);
    }

}
```

2.5 在 Spark SQL 作业中使用 UDAF

操作场景

DLI支持用户使用Hive UDAF（User Defined Aggregation Function，用户定义聚合函数）可对多行数据产生作用，通常与groupBy联合使用；等同于SQL中常用的SUM()，AVG()，也是聚合函数。

约束限制

- 在DLI Console上执行UDAF相关操作时，需要使用自建的SQL队列。
- 跨账号使用UDAF时，除了创建UDAF函数的用户，其他用户如果需要使用时，需要先进行授权才可使用对应的UDAF函数。

  授权操作参考如下：登录DLI管理控制台，选择“ 数据管理 > 程序包管理”页面，选择对应的UDAF Jar包，单击“操作”列中的“权限管理”，进入权限管理页面，单击右上角“授权”，勾选对应权限。
- 自定义函数中引用static类或接口时，必须要加上“try catch”异常捕获，否则可能会造成包冲突，导致函数功能异常。

环境准备

在进行UDAF开发前，请准备以下开发环境。

表 2-6 UDAF 开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本（访问**Java官网**）。
安装和配置IntelliJ IDEA	**IntelliJ IDEA**为进行应用开发的工具，版本要求使用2019.1或其2019.1往后的版本。
安装Maven	开发环境的基本配置（**下载**并**安装 Maven**）。用于项目管理，贯穿软件开发生命周期。

开发流程

DLI下UDAF函数开发流程参考如下：

图 2-71 UDAF 开发流程

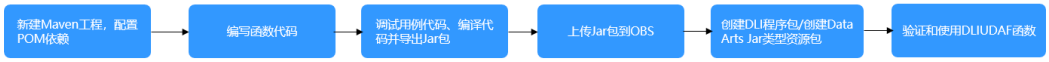


表 2-7 开发流程说明

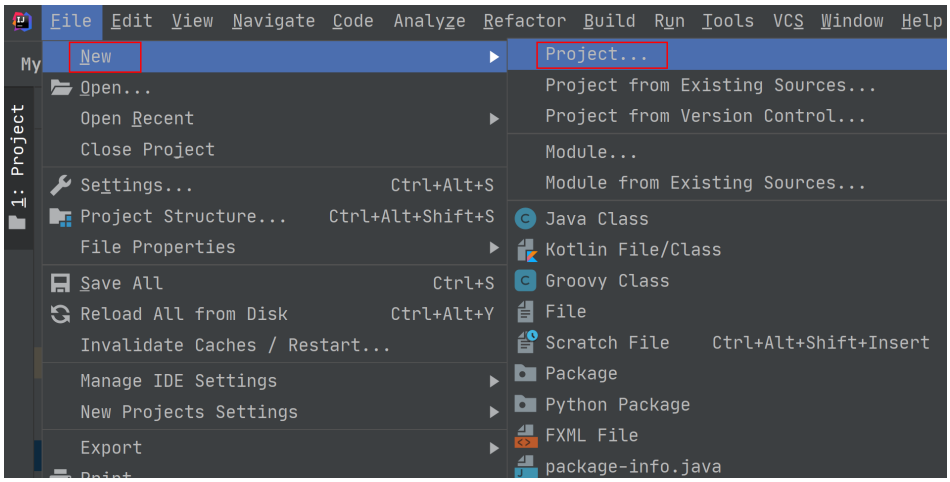
序号	阶段	操作界面	说明
1	新建Maven工程，配置pom文件	IntelliJ IDEA	参考**操作步骤**说明，编写UDAF函数代码。
2	编写UDAF函数代码		
3	调试，编译代码并导出Jar包		
4	上传Jar包到OBS	OBS控制台	将生成的UDAF函数Jar包文件上传到OBS目录下。
5	创建DLI程序包	DLI控制台	选择刚上传到OBS的UDAF函数的Jar文件，由DLI进行纳管。
6	创建DLI的UDAF函数	DLI控制台	在DLI控制台的SQL作业管理界面创建使用的UDAF函数。
7	验证和使用DLI的UDAF函数	DLI控制台	在DLI作业中使用创建的UDAF函数。

操作步骤

1. 新建Maven工程，配置pom文件。以下通过IntelliJ IDEA 2020.2工具操作演示。

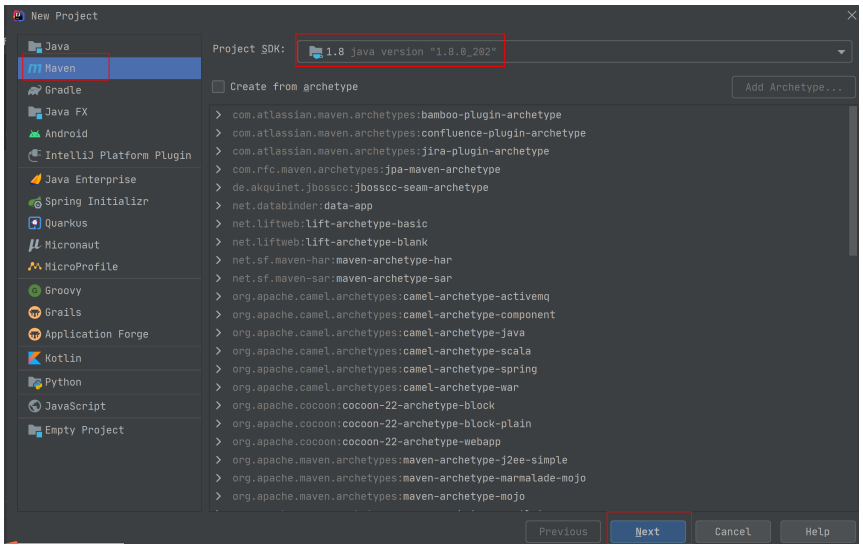
a. 打开IntelliJ IDEA，选择“File > New > Project”。

图 2-72 新建 Project



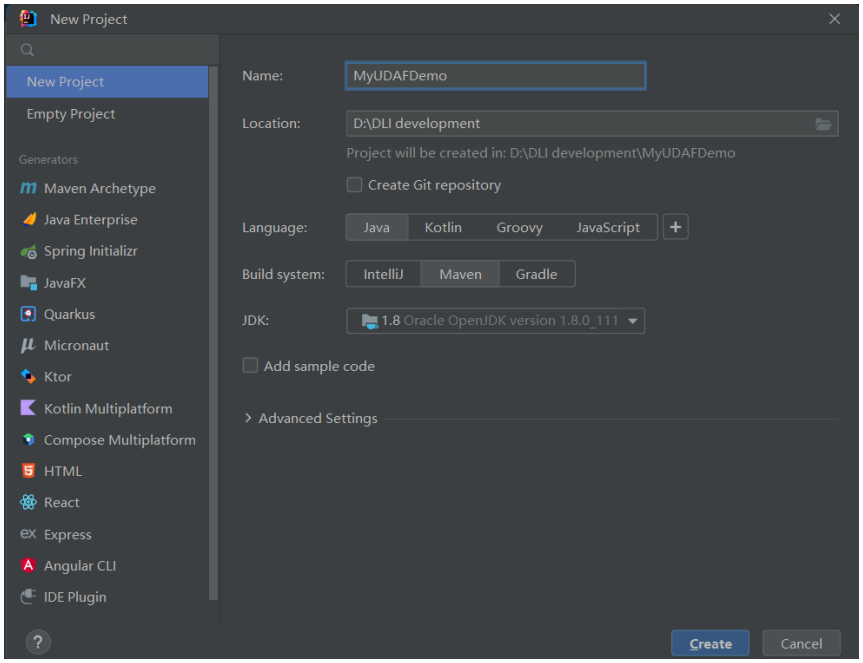
b. 选择Maven，Project SDK选择1.8，单击“Next”。

图 2-73 配置 Project SDK



c. 定义样例工程名和配置样例工程存储路径，单击“Create",下一步单击弹窗中的“Finish”完成工程创建。

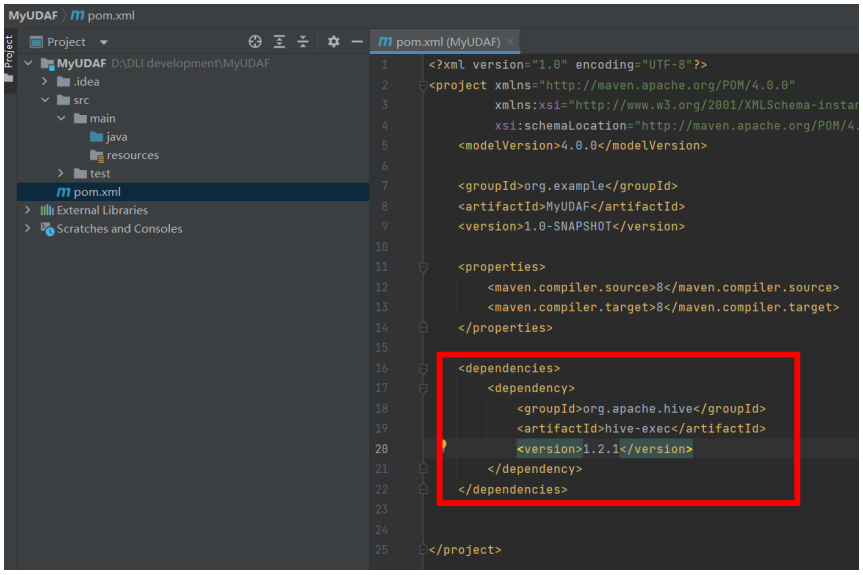
图 2-74 完成 Project 创建



d. 在pom.xml文件中添加如下配置。

```
<dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>1.2.1</version>
        </dependency>
</dependencies>
```

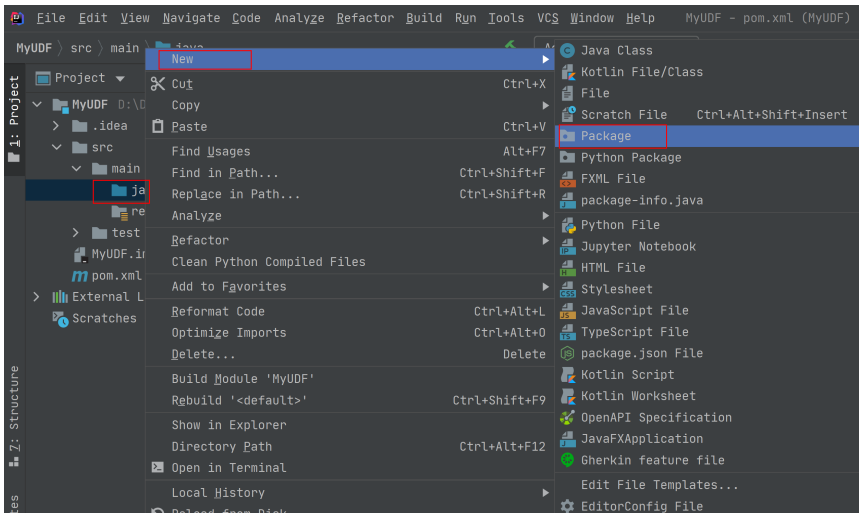
图 2-75 pom 文件中添加配置



e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

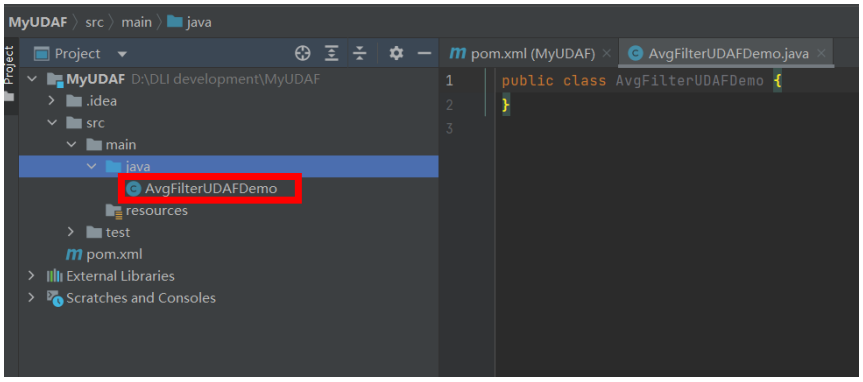
Package根据需要定义，本示例定义为：“com.dli.demo”

图 2-76 新建 Package



在包路径下新建Java Class文件，本示例定义为：AvgFilterUDAFDemo。

图 2-77 创建类



2. 编写UDAF函数代码。UDAF函数实现，主要注意以下几点：
   - 自定义UDAF需要继承org.apache.hadoop.hive.ql.exec.UDAF和org.apache.hadoop.hive.ql.exec.UDAFEvaluator类。函数类需要继承UDAF类，计算类Evaluator实现UDAFEvaluator接口。
   - Evaluator需要实现UDAFEvaluator的**init**、**iterate**、**terminatePartial**、**merge**、**terminate**这几个函数。
     - **init**函数实现接口UDAFEvaluator的init函数。
     - **iterate**接收传入的参数，并进行内部的迭代。
     - **terminatePartial**无参数，其为iterate函数遍历结束后，返回遍历得到的数据，terminatePartial类似于 hadoop的Combiner。
     - **merge**接收terminatePartial的返回结果。
     - **terminate**返回最终的聚集函数结果。

   详细UDAF函数实现，可以参考如下样例代码：

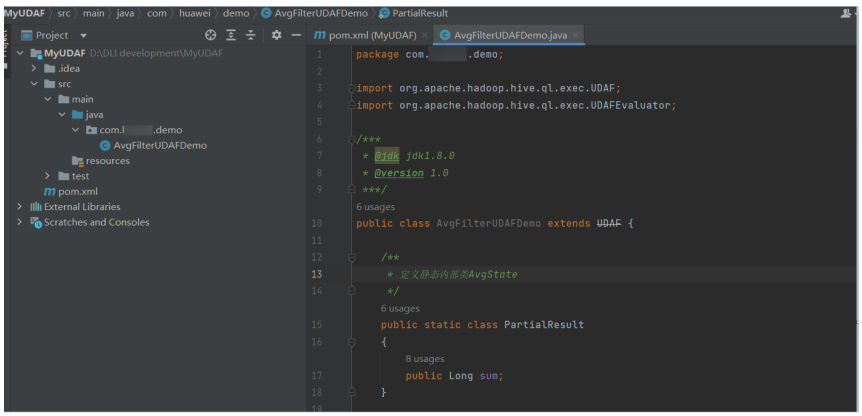
```
package com.dli.demo;
```



```
import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

/***
 * @jdk jdk1.8.0
 * @version 1.0
 ***/
public class AvgFilterUDAFDemo extends UDAF {

    /**
     * 定义静态内部类AvgFilter
     */
    public static class PartialResult
    {
        public Long sum;
    }

    public static class VarianceEvaluator implements UDAFEvaluator {

        //初始化PartialResult对象
        private AvgFilterUDAFDemo.PartialResult partial;

        //创建VarianceEvaluator无参构造函数
        public VarianceEvaluator(){

            this.partial = new AvgFilterUDAFDemo.PartialResult();

            init();
        }

        /**
         * init函数类似于构造函数，用于UDAF的初始化
         */
        @Override
        public void init() {

            //设置sum初始值
            this.partial.sum = 0L;
        }

        /**
         * iterate接收传入的参数，并进行内部的轮转。
         * @param x
         * @return
         */
        public void iterate(Long x) {
            if (x == null) {
                return;
            }
            AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
            tmp9_6.sum = tmp9_6.sum | x;
        }

        /**
         * terminatePartial无参数，其为iterate函数遍历结束后，返回轮转数据，
         * terminatePartial类似于hadoop的Combiner
         * @return
         */
        public AvgFilterUDAFDemo.PartialResult terminatePartial()
        {
            return this.partial;
        }

        /**
         * merge接收terminatePartial的返回结果，进行数据merge操作
         * @param
         * @return
         */
        public void merge(AvgFilterUDAFDemo.PartialResult pr)
```

```
    {
        if (pr == null) {
            return;
        }
        AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
        tmp9_6.sum = tmp9_6.sum | pr.sum;
    }

    /**
     * terminate返回最终的聚集函数结果
     * @return
     */
    public Long terminate()
    {
        if (this.partial.sum == null) {
            return 0L;
        }
        return this.partial.sum;
    }
  }
}
```

图 2-78 编写 UDAF 函数代码

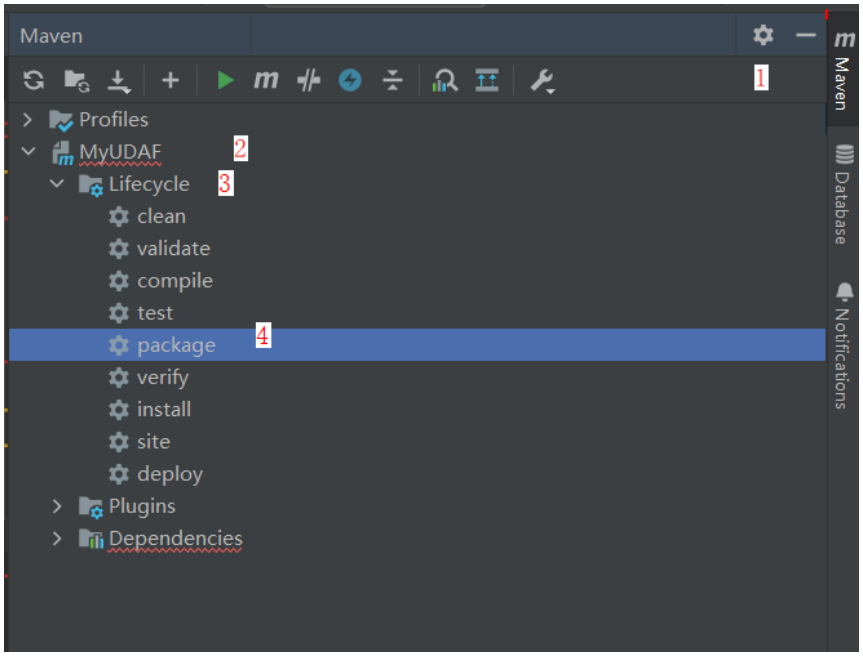


3. 编写调试完成代码后，通过IntelliJ IDEA工具编译代码并导出Jar包。

   a. 单击工具右侧的“Maven”，参考下图分别单击“clean”、“compile”对代码进行编译。

      编译成功后，单击“package”对代码进行打包。

图 2-79 导出 jar 包



b. 打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\DLITest\MyUDAF\target”下名为“MyUDAF-1.0-SNAPSHOT.jar”。

4. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

   说明

   Jar包文件上传的OBS桶所在的区域需与DLI的队列区域相同，不可跨区域执行操作。

5. （可选）可以将Jar包文件上传到DLI的程序包管理中，方便后续统一管理。
   a. 登录DLI管理控制台，单击“数据管理 > 程序包管理”。
   b. 在“程序包管理”页面，单击右上角的“创建”创建程序包。
   c. 在“创建程序包”对话框，配置以下参数。
      - 包类型：选择“JAR”。
      - OBS路径：程序包所在的OBS路径。
      - 分组设置和组名称根据情况选择设置，方便后续识别和管理程序包。
   d. 单击“确定”，完成创建程序包。
6. 创建UDAF函数。
   a. 登录登录DLI管理控制台，创建SQL队列和数据库。
   b. 登录DLI管理控制台，单击“SQL编辑器”，执行引擎选择“spark”，选择已创建的SQL队列和数据库。
   c. 在SQL编辑区域输入下列命令创建UDAF函数，单击“执行”提交创建。

      说明

      如果该客户开启了自定义函数热加载功能，注册语句会发生变化。

      详情可参考**注册函数**。

      ```
      CREATE FUNCTION AvgFilterUDAFDemo AS 'com.dli.demo.AvgFilterUDAFDemo' using jar 'obs://dli-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
      ```

或

```
CREATE OR REPLACE FUNCTION AvgFilterUDAFDemo AS 'com.dli.demo.AvgFilterUDAFDemo'
using jar 'obs://dli-test-obs01/MyUDAF-1.0-SNAPSHOT.jar';
```

7. 重启原有SQL队列，使得创建的Function生效。
   a. 登录数据湖探索管理控制台，选择“资源管理”》“队列管理”，在对应“SQL队列”类型作业的“操作”列，单击“更多”》“重启”。
   b. 在“重启队列”界面，选择“确定”完成队列重启
8. 使用UDAF函数。

   在查询语句中使用**6**中创建的UDAF函数:

   ```
   select AvgFilterUDAFDemo(real_stock_rate) AS show_rate FROM dw_ad_estimate_real_stock_rate
   limit 1000;
   ```

9. （可选）删除UDAF函数。

   如果不再使用UDAF函数，可执行以下语句删除该函数:

   ```
   Drop FUNCTION AvgFilterUDAFDemo;
   ```

3 Flink 作业开发指南

3.1 流生态作业开发指引

流生态系统基于Flink和Spark双引擎，完全兼容Flink/Storm/Spark开源社区版本接口，并且在此基础上做了特性增强和性能提升，为用户提供易用、低时延、高吞吐的数据湖探索。

数据湖探索的流生态开发包括云服务生态、开源生态和自拓展生态：

- 云服务生态

  DLI服务在Stream SQL中支持与其他服务的连通。用户可以直接使用SQL从这些服务中读写数据，如DIS、OBS、CloudTable、MRS、RDS、SMN、DCS等。

- 开源生态

  通过对等连接建立与其他VPC的网络连接后，用户可以在DLI的租户独享集群中访问所有Flink和Spark支持的数据源与输出源，如Kafka、Hbase、ElasticSearch等。

- 自拓展生态

  用户可通过编写代码实现从想要的云生态或者开源生态获取数据，作为Flink作业的输入数据。

DLI Flink Connector支持列表请参考**Connector概述**。

3.2 Flink OpenSource SQL 作业开发

3.2.1 从 Kafka 读取数据写入到 RDS

须知

本指导仅适用于Flink 1.12版本。

场景描述

该场景为根据商品的实时点击量，获取每小时内点击量最高的3个商品及其相关信息。商品的实时点击量数据为输入源发送到Kafka中，再将Kafka数据的分析结果输出到RDS中。

例如，输入如下样例数据：

```
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:01:00", "product_id":"0002", "product_name":"name1"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:02:00", "product_id":"0002", "product_name":"name1"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:06:00", "product_id":"0004", "product_name":"name2"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:10:00", "product_id":"0003", "product_name":"name3"}
{"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:15:00", "product_id":"0005", "product_name":"name4"}
{"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:16:00", "product_id":"0005", "product_name":"name4"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:56:00", "product_id":"0004", "product_name":"name2"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:05:00", "product_id":"0005", "product_name":"name4"}
{"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:10:00", "product_id":"0006", "product_name":"name5"}
{"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 09:13:00", "product_id":"0006", "product_name":"name5"}
```

预期输出：

```
2021-03-24 08:00:00 - 2021-03-24 08:59:59,0002,name1,2
2021-03-24 08:00:00 - 2021-03-24 08:59:59,0004,name2,2
2021-03-24 08:00:00 - 2021-03-24 08:59:59,0005,name4,2
2021-03-24 09:00:00 - 2021-03-24 09:59:59,0006,name5,2
2021-03-24 09:00:00 - 2021-03-24 09:59:59,0005,name4,1
```

前提条件

1. 已创建DMS Kafka实例。
   具体步骤可参考：**DMS Kafka入门指引**。

   注意

   创建DMS Kafka实例时，**不能开启Kafka SASL_SSL**。

2. 已创建RDS MySQL实例。
   本示例创建的RDS MySQL数据库版本选择为：8.0。
   具体步骤可参考：**购买RDS for MySQL实例**。

整体作业开发流程

整体作业开发流程参考**图3-1**。

图 3-1 作业开发流程

步骤1：创建队列：创建DLI作业运行的队列。

步骤2：创建Kafka的Topic：创建Kafka生产消费数据的Topic。

步骤3：创建RDS数据库和表：创建RDS MySQL数据库和表信息。

步骤4：创建增强型跨源连接：DLI上创建连接Kafka和RDS的跨源连接，打通网络。

步骤5：运行作业：DLI上创建和运行Flink OpenSource作业。

步骤6：发送数据和查询结果：Kafka上发送流数据，在RDS上查看运行结果。

步骤 1：创建队列

1. 登录DLI管理控制台，在左侧导航栏单击“资源管理 > 队列管理”，可进入队列管理页面。
2. 在队列管理界面，单击界面右上角的“购买队列”。
3. 在“购买队列”界面，填写具体的队列配置参数，具体参数填写参考如下。
   - 计费模式：选择“包年/包月”或“按需计费”。本例选择“按需计费”。
   - 区域和项目：保持默认值即可。
   - 名称：填写具体的队列名称。

     说明

     新建的队列名称，名称只能包含数字、英文字母和下划线，但不能是纯数字，且不能以下划线开头。长度限制：1~128个字符。

     队列名称不区分大小写，系统会自动转换为小写。
   - 类型：队列类型选择“通用队列”。**“按需计费”时需要勾选“专属资源模式”**。
   - AZ策略、CPU架构、规格：保持默认即可。
   - 企业项目：当前选择为“default”。
   - 高级选项：选择“自定义”。
   - 网段：配置队列网段。例如，当前配置为10.0.0.0/16。

     注意

     队列的网段不能和DMS Kafka、RDS MySQL实例的子网网段有重合，否则后续创建跨源连接会失败。
   - 其他参数根据需要选择和配置。
4. 参数配置完成后，单击“立即购买”，确认配置信息无误后，单击“提交”完成队列创建。

步骤 2：创建 Kafka 的 Topic

1. 登录Kafka管理控制台，选择“Kafka专享版”，单击对应的Kafka实例名称，进入到Kafka实例的基本信息页面。
2. 单击“Topic管理 > 创建Topic”，创建一个Topic。Topic配置参数如下：
   - Topic名称。本示例输入为：testkafkatopic。
   - 分区数：1。

- 副本数：1。

其他参数保持默认即可。

步骤 3：创建 RDS 数据库和表

1. 登录RDS管理控制台，在“实例管理”界面，选择已创建的RDS MySQL实例，选择操作列的“更多 > 登录”，进入数据管理服务实例登录界面。
2. 输入实例登录的用户名和密码。单击“登录”，即可进入RDS MySQL数据库并进行管理。
3. 在数据库实例界面，单击“新建数据库”，数据库名定义为：testrdsdb，字符集保持默认即可。
4. 在已创建的数据库的操作列，单击“SQL查询”，输入以下创建表语句，创建RDS MySQL表。

```
CREATE TABLE clicktop (
    `range_time` VARCHAR(64) NOT NULL,
    `product_id` VARCHAR(32) NOT NULL,
    `product_name` VARCHAR(32),
    `event_count` VARCHAR(32),
    PRIMARY KEY (`range_time`,`product_id`)
)   ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4;
```

步骤 4：创建增强型跨源连接

- **创建DLI连接Kafka的增强型跨源连接**

  a. 在Kafka管理控制台，选择“Kafka专享版”，单击对应的Kafka名称，进入到Kafka的基本信息页面。

  b. 在“连接信息”中获取该Kafka的“内网连接地址”，在“基本信息”的“网络”中获取该实例的“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

  c. 单击“网络”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

  d. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

  e. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

     - 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_kafka。
     - 弹性资源池：选择**步骤1：创建队列**中已经创建的队列名称。（未添加至资源池的队列，请直接选择队列名称。）
     - 虚拟私有云：选择Kafka的虚拟私有云。
     - 子网：选择Kafka的子网。
     - 其他参数可以根据需要选择配置。

     参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为“已激活”后可以进行后续步骤。

  f. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

g. 在“测试连通性”界面，根据**b**中获取的Kafka连接信息，地址栏输入“Kafka内网地址:Kafka数据库端口”，单击“测试”测试DLI到Kafka网络是否可达。

- **创建DLI连接RDS的增强型跨源连接**

a. 在RDS管理控制台，选择“实例管理”，单击对应的RDS实例名称，进入到RDS的基本信息页面。

b. 在“基本信息”的“连接信息”中获取该实例的“内网地址”、“数据库端口”、“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

c. 单击“连接信息”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. Kafka和RDS实例属于同一VPC和子网下?

i. 是，执行**g**。Kafka和RDS实例在同一VPC和子网，不用再重复创建增强型跨源连接。

ii. 否，执行**e**。Kafka和RDS实例分别在两个VPC和子网下，则要分别创建增强型跨源连接打通网络。

e. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

f. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

- 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_rds。
- 弹性资源池：选择**步骤1：创建队列**中已经创建的队列名称。（未添加至资源池的队列，请直接选择队列名称。）
- 虚拟私有云：选择RDS的虚拟私有云。
- 子网：选择RDS的子网。
- 其他参数可以根据需要选择配置。

参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

g. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

h. 在“测试连通性”界面，根据**b**中获取的RDS连接信息，地址栏输入“RDS内网地址:RDS数据库端口”，单击“测试”测试DLI到RDS网络是否可达。

步骤 5：运行作业

1. 在DLI管理控制台，单击“作业管理 > Flink作业”，在Flink作业管理界面，单击“创建作业”。
2. 在创建作业界面，作业类型选择“Flink OpenSource SQL”，名称填写为：FlinkKafkaRds。单击“确定”，跳转到Flink作业编辑界面。
3. 在Flink OpenSource SQL作业编辑界面，配置如下参数，其他参数默认即可。
   - 所属队列：选择**步骤1：创建队列**中创建的队列。
   - Flink版本：选择1.12。

- 保存作业日志：勾选。
- OBS桶：选择保存作业日志的OBS桶，根据提示进行OBS桶权限授权。
- 开启Checkpoint：勾选。
- Flink作业编辑框中输入具体的作业SQL，本示例作业参考如下。SQL中加粗的参数需要根据实际情况修改。

说明

本示例使用的Flink版本为1.12，故Flink OpenSource SQL语法也是1.12。本示例数据源是Kafka，写入结果数据到RDS。

请参考**Flink OpenSource SQL 1.12创建Kafka源表**、**Flink OpenSource SQL 1.12创建JDBC结果表**（RDS连接）。

```
create table click_product(
  user_id string, --点击用户的id
  user_name string, --用户名称
  event_time string, --点击时间
  product_id string, --商品id
  product_name string --商品名称
) with (
  "connector" = "kafka",
  "properties.bootstrap.servers" = "10.128.0.120:9092,10.128.0.89:9092,10.128.0.83:9092",--替换为kafka的内网连接地址和端口
  "properties.group.id" = "click",
  "topic" = "testkafkatopic",--创建的Kafka Topic名称
  "format" = "json",
  "scan.startup.mode" = "latest-offset"
);

--结果表
create table top_product (
  range_time string, --计算的时间范围
  product_id string, --商品id
  product_name string, --商品名称
  event_count bigint, --点击次数
  primary key (range_time, product_id) not enforced
) with (
  "connector" = "jdbc",
  "url" = "jdbc:mysql://192.168.12.148:3306/testrdsdb",--testrdsdb为创建的RDS的数据库名，IP和端口替换为RDS MySQL的实例IP和端口
  "table-name" = "clicktop",
  "pwd_auth_name"="xxxxx", --DLI侧创建的Password类型的跨源认证名称。使用跨源认证则无需在作业中配置账号和密码。
  "sink.buffer-flush.max-rows" = "1000",
  "sink.buffer-flush.interval" = "1s"
);

create view current_event_view
as
  select product_id, product_name, count(1) as click_count, concat(substring(event_time, 1, 13), ":00:00") as min_event_time, concat(substring(event_time, 1, 13), ":59:59") as max_event_time
  from click_product group by substring (event_time, 1, 13), product_id, product_name;

insert into top_product
  select
    concat(min_event_time, " - ", max_event_time) as range_time,
    product_id,
    product_name,
    click_count
  from (
    select *,
    row_number() over (partition by min_event_time order by click_count desc) as row_num
    from current_event_view
  )
  where row_num <= 3
```

4. 单击“语义校验”确保SQL语义校验成功。单击“保存”，保存作业。单击“启动”，启动作业，确认作业参数信息，单击“立即启动”开始执行作业。等待作业运行状态变为“运行中”。

步骤 6：发送数据和查询结果

1. 使用Kafka客户端向**步骤2：创建Kafka的Topic**中的Topic发送数据，模拟实时数据流。

   Kafka生产和发送数据的方法请参考：**DMS - 连接实例生产消费信息**。

   发送样例数据如下：

   ```
   {"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:01:00", "product_id":"0002", "product_name":"name1"}
   {"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:02:00", "product_id":"0002", "product_name":"name1"}
   {"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 08:06:00", "product_id":"0004", "product_name":"name2"}
   {"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:10:00", "product_id":"0003", "product_name":"name3"}
   {"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:15:00", "product_id":"0005", "product_name":"name4"}
   {"user_id":"0003", "user_name":"Cindy", "event_time":"2021-03-24 08:16:00", "product_id":"0005", "product_name":"name4"}
   {"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 08:56:00", "product_id":"0004", "product_name":"name2"}
   {"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:05:00", "product_id":"0005", "product_name":"name4"}
   {"user_id":"0001", "user_name":"Alice", "event_time":"2021-03-24 09:10:00", "product_id":"0006", "product_name":"name5"}
   {"user_id":"0002", "user_name":"Bob", "event_time":"2021-03-24 09:13:00", "product_id":"0006", "product_name":"name5"}
   ```

2. 登录RDS控制台，单击RDS数据库实例，单击创建的数据库名，如“testrdsdb”，在创建的表“clicktop”所在行的“操作”列，单击“SQL查询”，输入以下查询语句。

   ```
   select * from `clicktop`;
   ```

3. 在“SQL查询”界面，单击“执行SQL”，查看RDS表数据已写入成功。

   图 3-2 RDS 表数据

	range_time	product_id	product_name	event_count
1	2021-03-24 08:00:00 - 2021-03-24 08:59:59	0002	name1	2
2	2021-03-24 08:00:00 - 2021-03-24 08:59:59	0004	name2	2
3	2021-03-24 08:00:00 - 2021-03-24 08:59:59	0005	name4	2
4	2021-03-24 09:00:00 - 2021-03-24 09:59:59	0005	name4	1
5	2021-03-24 09:00:00 - 2021-03-24 09:59:59	0006	name5	2

3.2.2 从 Kafka 读取数据写入到 DWS

须知

本指导仅适用于Flink 1.12版本。

场景描述

该场景为对汽车驾驶的实时数据信息进行分析，将满足特定条件的数据结果进行汇总。汽车驾驶的实时数据信息为数据源发送到Kafka中，再将Kafka数据的分析结果输出到DWS中。

例如，输入如下样例数据：

```
{"car_id":"3027", "car_owner":"lilei", "car_age":"7", "average_speed":"76", "total_miles":"15000"}
{"car_id":"3028", "car_owner":"hanmeimei", "car_age":"6", "average_speed":"92", "total_miles":"17000"}
{"car_id":"3029", "car_owner":"Ann", "car_age":"10", "average_speed":"81", "total_miles":"230000"}
```

预期输出：average_speed <= 90 和 total_miles <= 200000的车辆，即：

```
{"car_id":"3027", "car_owner":"lilei", "car_age":"7", "average_speed":"76", "total_miles":"15000"}
```

前提条件

1. 已创建DMS Kafka实例。
   具体步骤可参考：**DMS Kafka入门指引**。

   注意

   创建DMS Kafka实例时，**不能开启Kafka SASL_SSL**。

2. 已创建DWS实例。
   具体创建DWS集群的操作可以参考**创建DWS集群**。

整体作业开发流程

整体作业开发流程参考**图3-3**。

图 3-3 作业开发流程

步骤1：创建队列：创建DLI作业运行的队列。

步骤2：创建Kafka的Topic：创建Kafka生产消费数据的Topic。

步骤3：创建DWS数据库和表：创建DWS数据库和表信息。

步骤4：创建增强型跨源连接：DLI上创建连接Kafka和DWS的跨源连接，打通网络。

步骤5：运行作业：DLI上创建和运行Flink OpenSource作业。

步骤6：发送数据和查询结果：Kafka上发送流数据，在RDS上查看运行结果。

步骤 1：创建队列

1. 登录DLI管理控制台，在左侧导航栏单击“资源管理 > 队列管理”，可进入队列管理页面。
2. 在队列管理界面，单击界面右上角的“购买队列”。
3. 在“购买队列”界面，填写具体的队列配置参数，具体参数填写参考如下。
   - 计费模式：选择“包年/包月”或“按需计费”。本例选择“按需计费”。
   - 区域和项目：保持默认值即可。
   - 名称：填写具体的队列名称。

> **说明**
>
> 新建的队列名称，名称只能包含数字、英文字母和下划线，但不能是纯数字，且不能以下划线开头。长度限制：1~128个字符。
>
> **队列名称不区分大小写，系统会自动转换为小写。**

- 类型：队列类型选择“通用队列”。**“按需计费”时需要勾选“专属资源模式”**。
- AZ策略、CPU架构、规格：保持默认即可。
- 企业项目：当前选择为“default”。
- 高级选项：选择“自定义”。
- 网段：配置队列网段。例如，当前配置为10.0.0.0/16。

> **注意**
>
> 队列的网段不能和DMS Kafka、RDS MySQL实例的子网网段有重合，否则后续创建跨源连接会失败。

- 其他参数根据需要选择和配置。

4. 参数配置完成后，单击“立即购买”，确认配置信息无误后，单击“提交”完成队列创建。

步骤 2：创建 Kafka 的 Topic

1. 在Kafka管理控制台，选择“Kafka专享版”，单击对应的Kafka名称，进入到Kafka的基本信息页面。
2. 单击“Topic管理 > 创建Topic”，创建一个Topic。Topic配置参数如下：
   - Topic名称。本示例输入为：testkafkatopic。
   - 分区数：1。
   - 副本数：1。

   其他参数保持默认即可。

步骤 3：创建 DWS 数据库和表

1. 参考**使用gsql命令行客户端连接DWS集群**连接已创建的DWS集群。
2. 执行以下命令连接DWS集群的默认数据库“gaussdb”：
   ```
   gsql -d gaussdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
   ```
   - gaussdb：DWS集群默认数据库。
   - DWS集群连接地址：请参见**获取集群连接地址**进行获取。如果通过公网地址连接，请指定为集群“公网访问地址”或“公网访问域名”，如果通过内网地址连接，请指定为集群“内网访问地址”或“内网访问域名”。如果通过弹性负载均衡连接，请指定为“弹性负载均衡地址”。
   - dbadmin：创建集群时设置的默认管理员用户名。
   - password ： 默认管理员用户的密码。
3. 在命令行窗口输入以下命令创建数据库“testdwsdb”。
   ```
   CREATE DATABASE testdwsdb;
   ```
4. 执行以下命令，退出gaussdb数据库，连接新创建的数据库“testdwsdb”。
   ```
   \q
   gsql -d testdwsdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
   ```


5. 执行以下命令创建表。

```
create schema test;
set current_schema= test;
drop table if exists qualified_cars;
CREATE TABLE qualified_cars
(
    car_id VARCHAR,
    car_owner VARCHAR,
    car_age INTEGER ,
    average_speed FLOAT8,
    total_miles FLOAT8
);
```

步骤 4：创建增强型跨源连接

- **创建DLI连接Kafka的增强型跨源连接**

  a. 在Kafka管理控制台，选择“Kafka专享版”，单击对应的Kafka名称，进入到Kafka的基本信息页面。

  b. 在“连接信息”中获取该Kafka的“内网连接地址”，在“基本信息”的“网络”中获取获取该实例的“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

  c. 单击“网络”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

  d. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

  e. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

     - 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_kafka。
     - 弹性资源池：选择**步骤1：创建队列**中已经创建的队列名。
     - 虚拟私有云：选择Kafka的虚拟私有云。
     - 子网：选择Kafka的子网。
     - 其他参数可以根据需要选择配置。

     参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

  f. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

  g. 在“测试连通性”界面，根据**b**中获取的Kafka连接信息，地址栏输入“Kafka内网地址:Kafka数据库端口”，单击“测试”测试DLI到Kafka网络是否可达。

- **创建DLI连接DWS的增强型跨源连接**

  a. 在DWS管理控制台，选择“集群管理”，单击已创建的DWS集群名称，进入到DWS的基本信息页面。

  b. 在“基本信息”的“数据库属性”中获取该实例的“内网IP”、“端口”，“基本信息”页面的“网络”中获取“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

c. 单击“连接信息”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. Kafka和DWS实例属于同一VPC和子网下?

   i. 是，执行**g**。Kafka和DWS实例在同一VPC和子网，不用再重复创建增强型跨源连接。

   ii. 否，执行**e**。Kafka和DWS实例分别在两个VPC和子网下，则要分别创建增强型跨源连接打通网络。

e. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

f. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

   - 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_dws。
   - 弹性资源池：选择**步骤1：创建队列**中已经创建的队列名。
   - 虚拟私有云：选择DWS的虚拟私有云。
   - 子网：选择DWS的子网。
   - 其他参数可以根据需要选择配置。

   参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

g. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

h. 在“测试连通性”界面，根据**b**中获取的DWS连接信息，地址栏输入“DWS内网IP:DWS端口”，单击“测试”测试DLI到DWS网络是否可达。

步骤 5：运行作业

1. 在DLI管理控制台，单击“作业管理 > Flink作业”，在Flink作业管理界面，单击“创建作业”。
2. 在创建队列界面，类型选择“Flink OpenSource SQL”，名称填写为：FlinkKafkaDWS。单击“确定”，跳转到Flink作业编辑界面。
3. 在Flink OpenSource SQL作业编辑界面，配置如下参数，其他参数默认即可。
   - 所属队列：选择**步骤1：创建队列**中创建的队列。
   - Flink版本：选择1.12。
   - 保存作业日志：勾选。
   - OBS桶：选择保存作业日志的OBS桶，根据提示进行OBS桶权限授权。
   - 开启Checkpoint：勾选。
   - Flink作业编辑框中输入具体的作业SQL，本示例作业参考如下。SQL中加粗的参数需要根据实际情况修改。

说明

本示例使用的Flink版本为1.12，故Flink OpenSource SQL语法也是1.12。本示例数据源是Kafka，写入结果数据到DWS。

请参考**Flink OpenSource SQL 1.12创建Kafka源表**和**Flink OpenSource SQL 1.12创建DWS结果表**。

```
create table car_infos(
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed DOUBLE,
  total_miles DOUBLE
) with (
    "connector" = "kafka",
    "properties.bootstrap.servers" = "10.128.0.120:9092,10.128.0.89:9092,10.128.0.83:9092",--替换为kafka的内网连接地址和端口
    "properties.group.id" = "click",
    "topic" = "testkafkatopic",--创建的Kafka的Topic
    "format" = "json",
    "scan.startup.mode" = "latest-offset"
);

create table qualified_cars (
  car_id STRING,
  car_owner STRING,
  car_age INT,
  average_speed DOUBLE,
  total_miles DOUBLE
)
WITH (
  'connector' = 'gaussdb',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url' = 'jdbc:gaussdb://192.168.168.16:8000/testdwsdb', ---192.168.168.16:8000替换为DWS的内网IP和端口，testdwsdb为创建的DWS数据库名
  'table-name' = 'test\".\"qualified_cars', ---test为创建的DWS表的schema，qualified_cars为对应的DWS表名
  'pwd_auth_name'= 'xxxxx', --DLI侧创建的Password类型的跨源认证名称。使用跨源认证则无需在作业中配置账号和密码。
  'write.mode' = 'insert'
);

/** 将合格车辆信息输出 **/
INSERT INTO qualified_cars
SELECT *
FROM car_infos
where average_speed <= 90 and total_miles <= 200000;
```

4. 单击“语义校验”确保SQL语义校验成功。单击“保存”，保存作业。单击“启动”，启动作业，确认作业参数信息，单击“立即启动”开始执行作业。等待作业运行状态变为“运行中”。

步骤 6：发送数据和查询结果

1. 使用Kafka客户端向**步骤2：创建Kafka的Topic**中的Topic发送数据，模拟实时数据流。

   Kafka生产和发送数据的方法请参考**DMS - 连接实例生产消费信息**。

   发送样例数据如下：

   ```
   {"car_id":"3027", "car_owner":"lilei", "car_age":"7", "average_speed":"76", "total_miles":"15000"}
   {"car_id":"3028", "car_owner":"hanmeimei", "car_age":"6", "average_speed":"92", "total_miles":"17000"}
   {"car_id":"3029", "car_owner":"Ann", "car_age":"10", "average_speed":"81", "total_miles":"230000"}
   ```

2. 连接已创建的DWS集群。

   具体操作请参考**使用gsql命令行客户端连接DWS集群**。

3. 执行以下命令连接DWS集群的默认数据库“testdwsdb”：
```
gsql -d testdwsdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
```
4. 查询DWS的表数据。
```
select * from test.qualified_cars;
```
查询结果参考如下：
```
car_id  car_owner  car_age  average_speed  total_miles
3027      lilei      7          76.0        15000.0
```

3.2.3 从 Kafka 读取数据写入到 Elasticsearch

须知

本指导仅适用于Flink 1.12版本。

场景描述

本示例场景对用户购买商品的数据信息进行分析，将满足特定条件的数据结果进行汇总输出。购买商品数据信息为数据源发送到Kafka中，再将Kafka数据的分析结果输出到Elasticsearch中。

例如，输入如下样例数据：

```
{"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

{"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0002", "user_name":"Jason", "area_id":"330106"}
```

DLI从Kafka读取数据写入Elasticsearch，在Elasticsearch集群的Kibana中查看相应结果。

前提条件

1. 已创建DMS Kafka实例。
   具体步骤可参考：**DMS Kafka入门指引**。

   ⚠ 注意

   创建DMS Kafka实例时，**不能开启Kafka SASL_SSL**。

2. 已创建Elasticsearch类型的CSS集群。
   具体创建CSS集群的操作可以参考**创建CSS集群**。
   本示例创建的CSS集群版本为：7.6.2，集群为非安全集群。

整体作业开发流程

整体作业开发流程参考**图3-4**。

图 3-4 作业开发流程

步骤1：创建队列：创建DLI作业运行的队列。

步骤2：创建Kafka的Topic：创建Kafka生产消费数据的Topic。

步骤3：创建Elasticsearch搜索索引：创建Elasticsearch搜索索引用于接收结果数据。

步骤4：创建增强型跨源连接：DLI上创建连接Kafka和CSS的跨源连接，打通网络。

步骤5：运行作业：DLI上创建和运行Flink OpenSource作业。

步骤6：发送数据和查询结果：Kafka上发送流数据，在CSS上查看运行结果。

步骤 1：创建队列

1. 登录DLI管理控制台，在左侧导航栏单击“资源管理 > 队列管理”，可进入队列管理页面。
2. 在队列管理界面，单击界面右上角的“购买队列”。
3. 在“购买队列”界面，填写具体的队列配置参数，具体参数填写参考如下。
   - 计费模式：选择“包年/包月”或“按需计费”。本示例选择“按需计费”。
   - 区域和项目：保持默认值即可。
   - 名称：填写具体的队列名称。

     说明

     新建的队列名称，名称只能包含数字、英文字母和下划线，但不能是纯数字，且不能以下划线开头。长度限制：1~128个字符。

     队列名称不区分大小写，系统会自动转换为小写。
   - 类型：队列类型选择“通用队列”。**“按需计费”时需要勾选“专属资源模式”**。
   - AZ策略、CPU架构、规格：保持默认即可。
   - 企业项目：当前选择为“default”。
   - 高级选项：选择“自定义”。
   - 网段：配置队列网段。例如，当前配置为10.0.0.0/16。

     注意

     队列的网段不能和DMS Kafka、RDS MySQL实例的子网网段有重合，否则后续创建跨源连接会失败。
   - 其他参数根据需要选择和配置。
4. 参数配置完成后，单击“立即购买”，确认配置信息无误后，单击“提交”完成队列创建。

步骤 2：创建 Kafka 的 Topic

1. 在Kafka管理控制台，选择“Kafka专享版”，单击对应的Kafka名称，进入到Kafka的基本信息页面。
2. 单击“Topic管理 > 创建Topic”，创建一个Topic。Topic配置参数如下：
   - Topic名称。本示例输入为：testkafkatopic。
   - 分区数：1。

– 副本数：1。

其他参数保持默认即可。

步骤 3：创建 Elasticsearch 搜索索引

1. 登录CSS管理控制台，选择“集群管理 > Elasticsearch”。
2. 在集群管理界面，在已创建的CSS集群的“操作”列，单击“Kibana”访问集群。
3. 在Kibana的左侧导航中选择“Dev Tools”，进入到Console界面。
4. 在Console界面，执行如下命令创建索引“shoporders”。

```
PUT /shoporders
{
  "settings": {
    "number_of_shards": 1
  },
    "mappings": {
      "properties": {
        "order_id": {
          "type": "text"
        },
        "order_channel": {
          "type": "text"
        },
        "order_time": {
          "type": "text"
        },
        "pay_amount": {
          "type": "double"
        },
        "real_pay": {
          "type": "double"
        },
        "pay_time": {
          "type": "text"
        },
        "user_id": {
          "type": "text"
        },
        "user_name": {
          "type": "text"
        },
        "area_id": {
          "type": "text"
        }
      }
    }
}
```

步骤 4：创建增强型跨源连接

- **创建DLI连接Kafka的增强型跨源连接**

  a. 在Kafka管理控制台，选择“Kafka专享版”，单击对应的Kafka名称，进入到Kafka的基本信息页面。

  b. 在“连接信息”中获取该Kafka的“内网连接地址”，在“基本信息”的“网络”中获取获取该实例的“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

  c. 单击“网络”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

e. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

- 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_kafka。
- 弹性资源池：选择**步骤1：创建队列**中已经创建的队列。
- 虚拟私有云：选择Kafka的虚拟私有云。
- 子网：选择Kafka的子网。
- 其他参数可以根据需要选择配置。

参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

f. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中添加的队列，在操作列，单击“更多 > 测试地址连通性”。

g. 在“测试连通性”界面，根据中获取的Kafka连接信息，地址栏输入“Kafka内网地址:Kafka数据库端口”，单击“测试”测试DLI到Kafka网络是否可达。

- **创建DLI连接CSS的增强型跨源连接**

a. 在CSS管理控制台，选择“集群管理”，单击已创建的CSS集群名称，进入到CSS的基本信息页面。

b. 在“基本信息”中获取CSS的“内网访问地址”、“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

c. 单击“连接信息”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. Kafka和CSS实例属于同一VPC和子网下?

i. 是，执行**g**。Kafka和CSS实例在同一VPC和子网，不用再重复创建增强型跨源连接。

ii. 否，执行**e**。Kafka和CSS实例分别在两个VPC和子网下，则要分别创建增强型跨源连接打通网络。

e. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

f. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

- 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_css。
- 弹性资源池：选择**步骤1：创建队列**中已经创建的队列。
- 虚拟私有云：选择CSS的虚拟私有云。
- 子网：选择CSS的子网。
- 其他参数可以根据需要选择配置。

参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

g. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中添加的队列，在操作列，单击“更多 > 测试地址连通性”。

h. 在“测试连通性”界面，根据**b**获取的CSS连接信息，地址栏输入“CSS内网地址:CSS内网端口”，单击“测试”测试DLI到CSS网络是否可达。

步骤 5：运行作业

1. 在DLI管理控制台，单击“作业管理 > Flink作业”，在Flink作业管理界面，单击“创建作业”。
2. 在创建队列界面，类型选择“Flink OpenSource SQL”，名称填写为：FlinkKafkaES。单击“确定”，跳转到Flink作业编辑界面。
3. 在Flink OpenSource SQL作业编辑界面，配置如下参数，其他参数默认即可。
   - 所属队列：选择**步骤1：创建队列**中创建的队列。
   - Flink版本：选择1.12。
   - 保存作业日志：勾选。
   - OBS桶：选择保存作业日志的OBS桶，根据提示进行OBS桶权限授权。
   - 开启Checkpoint：勾选。
   - Flink作业编辑框中输入具体的作业SQL，本示例作业参考如下。SQL中加粗的参数需要根据实际情况修改。

     说明

     本示例使用的Flink版本为1.12，故Flink OpenSource SQL语法也是1.12。本示例数据源是Kafka，写入结果数据到Elasticsearch。

     请参考**Flink OpenSource SQL 1.12创建Kafka源表**和**Flink OpenSource SQL 1.12创建Elasticsearch结果表**。

   - 创建Kafka源表，将DLI和Kafka数据源进行连接。

```
CREATE TABLE kafkaSource (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
) with (
  "connector" = "kafka",
  "properties.bootstrap.servers" = "10.128.0.120:9092,10.128.0.89:9092,10.128.0.83:9092",--替换为kafka的内网连接地址和端口
  "properties.group.id" = "click",
  "topic" = "testkafkatopic", --创建的Kafka Topic
  "format" = "json",
  "scan.startup.mode" = "latest-offset"
);
```

   - 创建Elasticsearch结果表，将DLI分析后的数据的结果展示在Elasticsearch结果表上。

```
CREATE TABLE elasticsearchSink (
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id string
```



```
) WITH (
  'connector' = 'elasticsearch-7',
  'hosts' = '192.168.168.125:9200', --替换为CSS集群的内网地址和端口
  'index' = 'shoporders' --创建的Elasticsearch搜索引擎
);
--将Kafka数据写入到Elasticsearch索引中
insert into
  elasticsearchSink
select
  *
from
  kafkaSource;
```

4. 单击“语义校验”确保SQL语义校验成功。单击“保存”，保存作业。单击“启动”，启动作业，确认作业参数信息，单击“立即启动”开始执行作业。等待作业运行状态变为“运行中”。

步骤 6：发送数据和查询结果

1. Kafaka端发送数据。

   使用Kafka客户端向**步骤2：创建Kafka的Topic**中的Topic发送数据，模拟实时数据流。

   Kafka生产和发送数据的方法请参考：**DMS - 连接实例生产消费信息**。

   发送样例数据如下：

   ```
   {"order_id":"202103241000000001", "order_channel":"webShop", "order_time":"2021-03-24 10:00:00", "pay_amount":"100.00", "real_pay":"100.00", "pay_time":"2021-03-24 10:02:03", "user_id":"0001", "user_name":"Alice", "area_id":"330106"}

   {"order_id":"202103241606060001", "order_channel":"appShop", "order_time":"2021-03-24 16:06:06", "pay_amount":"200.00", "real_pay":"180.00", "pay_time":"2021-03-24 16:10:06", "user_id":"0002", "user_name":"Jason", "area_id":"330106"}
   ```

2. 查看Elasticsearch端数据处理后的相应结果。

   发送成功后，在CSS集群的Kibana中执行下述语句并查看相应结果：

   ```
   GET shoporders/_search
   ```

   查询结果返回如下：

   ```
   {
    "took" : 0,
    "timed_out" : false,
    "_shards" : {
      "total" : 1,
      "successful" : 1,
      "skipped" : 0,
      "failed" : 0
    },
    "hits" : {
      "total" : {
        "value" : 2,
        "relation" : "eq"
      },
      "max_score" : 1.0,
      "hits" : [
        {
          "_index" : "shoporders",
          "_type" : "_doc",
          "_id" : "6fswzIAByVjqg3_qAyM1",
          "_score" : 1.0,
          "_source" : {
            "order_id" : "202103241000000001",
            "order_channel" : "webShop",
            "order_time" : "2021-03-24 10:00:00",
            "pay_amount" : 100.0,
            "real_pay" : 100.0,
            "pay_time" : "2021-03-24 10:02:03",
   ```



```
        "user_id" : "0001",
        "user_name" : "Alice",
        "area_id" : "330106"
      }
    },
    {
      "_index" : "shoporders",
      "_type" : "_doc",
      "_id" : "6vs1zIAByVjqg3_qyyPp",
      "_score" : 1.0,
      "_source" : {
        "order_id" : "202103241606060001",
        "order_channel" : "appShop",
        "order_time" : "2021-03-24 16:06:06",
        "pay_amount" : 200.0,
        "real_pay" : 180.0,
        "pay_time" : "2021-03-24 16:10:06",
        "user_id" : "0002",
        "user_name" : "Jason",
        "area_id" : "330106"
      }
    }
  ]
 }
}
```


3.2.4 从 MySQL CDC 源表读取数据写入到 DWS

须知

本指导仅适用于Flink 1.12版本。

场景描述

CDC是变更数据捕获（Change Data Capture）技术的缩写，它可以将源数据库的增量变动记录，同步到一个或多个数据目的中。CDC在数据同步过程中，还可以对数据进行一定的处理，例如分组（GROUP BY）、多表的关联（JOIN）等。

本示例通过创建MySQL CDC源表来监控MySQL的数据变化，并将变化的数据信息插入到DWS数据库中。

前提条件

1. 已创建RDS MySQL实例。本示例创建的RDS MySQL数据库版本选择为：8.0。
   具体步骤可参考**购买RDS for MySQL实例**。
2. 已创建DWS实例。
   具体创建DWS集群的操作可以参考**创建DWS集群**。

整体作业开发流程

整体作业开发流程参考**图3-5**。

图 3-5 作业开发流程

步骤1：创建队列：创建DLI作业运行的队列。

步骤2：创建RDS MySQL数据库和表：创建RDS MySQL的数据库和表。

步骤3：创建DWS数据库和表：创建用于接收数据的DWS数据库和表。

步骤4：创建增强型跨源连接：DLI上创建连接RDS和DWS的跨源连接，打通网络。

步骤5：运行作业：DLI上创建和运行Flink OpenSource作业。

步骤6：发送数据和查询结果：RDS MySQL的表上插入数据，在DWS上查看运行结果。

步骤 1：创建队列

1. 登录DLI管理控制台，在左侧导航栏单击“资源管理 > 队列管理”，可进入队列管理页面。
2. 在队列管理界面，单击界面右上角的“购买队列”。
3. 在“购买队列”界面，填写具体的队列配置参数，具体参数填写参考如下。
   - 计费模式：选择“包年/包月”或“按需计费”。本示例选择“按需计费”。
   - 区域和项目：保持默认值即可。
   - 名称：填写具体的队列名称。

     说明

     新建的队列名称，名称只能包含数字、英文字母和下划线，但不能是纯数字，且不能以下划线开头。长度限制：1~128个字符。

     队列名称不区分大小写，系统会自动转换为小写。
   - 类型：队列类型选择“通用队列”。**“按需计费”时需要勾选“专属资源模式”**。
   - AZ策略、CPU架构、规格：保持默认即可。
   - 企业项目：当前选择为“default”。
   - 高级选项：选择“自定义”。
   - 网段：配置队列网段。例如，当前配置为10.0.0.0/16。

     注意

     队列的网段不能和DMS Kafka、RDS MySQL实例的子网网段有重合，否则后续创建跨源连接会失败。
   - 其他参数根据需要选择和配置。
4. 参数配置完成后，单击“立即购买”，确认配置信息无误后，单击“提交”完成队列创建。

步骤 2：创建 RDS MySQL 数据库和表

1. 登录RDS管理控制台，在“实例管理”界面，选择已创建的RDS MySQL实例，选择操作列的“更多 > 登录”，进入数据管理服务实例登录界面。
2. 输入实例登录的用户名和密码。单击“登录”，即可进入RDS MySQL数据库并进行管理。

3. 在数据库实例界面，单击“新建数据库”，数据库名定义为：testrdsdb，字符集保持默认即可。
4. 在已创建的数据库的操作列，单击“SQL查询”，输入以下创建表语句，创建RDS MySQL表。

```
CREATE TABLE mysqlcdc (
    `order_id` VARCHAR(64) NOT NULL,
    `order_channel` VARCHAR(32) NOT NULL,
    `order_time` VARCHAR(32),
    `pay_amount` DOUBLE,
    `real_pay` DOUBLE,
    `pay_time` VARCHAR(32),
    `user_id` VARCHAR(32),
    `user_name` VARCHAR(32),
    `area_id` VARCHAR(32)

)    ENGINE = InnoDB
    DEFAULT CHARACTER SET = utf8mb4;
```

步骤 3：创建 DWS 数据库和表

1. 连接已创建的DWS集群。

   请参考**使用gsql命令行客户端连接DWS集群**。
2. 执行以下命令连接DWS集群的默认数据库“gaussdb”：

   ```
   gsql -d gaussdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
   ```

   - gaussdb：DWS集群默认数据库。
   - DWS集群连接地址：请参见**获取集群连接地址**进行获取。如果通过公网地址连接，请指定为集群“公网访问地址”或“公网访问域名”，如果通过内网地址连接，请指定为集群“内网访问地址”或“内网访问域名”。如果通过弹性负载均衡连接，请指定为“弹性负载均衡地址”。
   - dbadmin：创建集群时设置的默认管理员用户名。
   - -W：默认管理员用户的密码。
3. 在命令行窗口输入以下命令创建数据库“testdwsdb”。

   ```
   CREATE DATABASE testdwsdb;
   ```
4. 执行以下命令，退出gaussdb数据库，连接新创建的数据库“testdwsdb”。

   ```
   \q
   gsql -d testdwsdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
   ```
5. 执行以下命令创建表。

   ```
   create schema test;
   set current_schema= test;
   drop table if exists dwsresult;
   CREATE TABLE dwsresult
   (
       car_id VARCHAR,
       car_owner VARCHAR,
       car_age INTEGER ,
       average_speed FLOAT8,
       total_miles FLOAT8
   );
   ```

步骤 4：创建增强型跨源连接

- **创建DLI连接RDS的增强型跨源连接**

  a. 在RDS管理控制台，选择“实例管理”，单击对应的RDS实例名称，进入到RDS的基本信息页面。

  b. 在“基本信息”的“连接信息”中获取该实例的“内网地址”、“数据库端口”、“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

c. 单击“连接信息”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

e. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

- 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_rds。
- 弹性资源池：选择**步骤1：创建队列**中已经创建的队列。
- 虚拟私有云：选择RDS的虚拟私有云。
- 子网：选择RDS的子网。
- 其他参数可以根据需要选择配置。

参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

f. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

g. 在“测试连通性”界面，根据**b**中获取的RDS连接信息，地址栏输入“RDS内网地址:RDS数据库端口”，单击“测试”测试DLI到RDS网络是否可达。

- **创建DLI连接DWS的增强型跨源连接**

a. 在DWS管理控制台，选择“集群管理”，单击已创建的DWS集群名称，进入到DWS的基本信息页面。

b. 在“基本信息”的“数据库属性”中获取该实例的“内网IP”、“端口”，“基本信息”页面的“网络”中获取“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

c. 单击“连接信息”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. RDS和DWS实例属于同一VPC和子网下?

i. 是，执行**g**。RDS和DWS实例在同一VPC和子网，不用再重复创建增强型跨源连接。

ii. 否，执行**e**。RDS和DWS实例分别在两个VPC和子网下，则要分别创建增强型跨源连接打通网络。

e. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

f. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

- 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_dws。
- 弹性资源池：选择**步骤1：创建队列**中已经创建的队列。
- 虚拟私有云：选择DWS的虚拟私有云。

- 子网：选择DWS的子网。
- 其他参数可以根据需要选择配置。

参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

g. 单击“队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

h. 在“测试连通性”界面，根据**b**中获取的DWS连接信息，地址栏输入“DWS内网IP:DWS端口”，单击“测试”测试DLI到DWS网络是否可达。

步骤 5：运行作业

1. 在DLI管理控制台，单击“作业管理 > Flink作业”，在Flink作业管理界面，单击“创建作业”。
2. 在创建队列界面，类型选择“Flink OpenSource SQL”，名称填写为：FlinkCDCMySQLDWS。单击“确定”，跳转到Flink作业编辑界面。
3. 在Flink OpenSource SQL作业编辑界面，配置如下参数，其他参数默认即可。
   - 所属队列：选择**步骤1：创建队列**中创建的队列。
   - Flink版本：选择1.12。
   - 保存作业日志：勾选。
   - OBS桶：选择保存作业日志的OBS桶，根据提示进行OBS桶权限授权。
   - 开启Checkpoint：勾选。
   - Flink作业编辑框中输入具体的作业SQL，本示例作业参考如下。SQL中加粗的参数需要根据实际情况修改。

     说明

     本示例使用的Flink版本为1.12，故Flink OpenSource SQL语法也是1.12。本示例数据源是Kafka，写入结果数据到Elasticsearch。

     请参考**Flink OpenSource SQL 1.12创建MySQL CDC源表**和**Flink OpenSource SQL 1.12创建DWS结果表**。

```
create table mysqlCdcSource(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING
) with (
  'connector' = 'mysql-cdc',
  'hostname' = '192.168.12.148',--IP替换为RDS MySQL的实例IP
  'port' = '3306',--端口替换为RDS MySQL的实例端口
  'pwd_auth_name'= 'xxxxx', --DLI侧创建的Password类型的跨源认证名称。使用跨源认证则无需在作业中配置账号和密码。
  'database-name' = 'testrdsdb',--RDS MySQL实例的数据库名
  'table-name' = 'mysqlcdc'--RDS MySQL实例的数据库下的表名
);

create table dwsSink(
  order_channel string,
  pay_amount double,
  real_pay double,
```



```
  primary key(order_channel) not enforced
) with (
  'connector' = 'gaussdb',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url' = 'jdbc:gaussdb://192.168.168.16:8000/testdwsdb', ---192.168.168.16:8000替换为DWS的内网IP和端口，testdwsdb为创建的DWS数据库名
  'table-name' = 'test\".\"dwsresult', ---test为创建的DWS表的schema，dwsresult为对应的DWS表名
  'pwd_auth_name'= 'xxxxx', --DLI侧创建的Password类型的跨源认证名称。使用跨源认证则无需在作业中配置账号和密码。
  'write.mode' = 'insert'
);

insert into dwsSink select order_channel, sum(pay_amount),sum(real_pay) from
mysqlCdcSource group by order_channel;
```

4. 单击“语义校验”确保SQL语义校验成功。单击“保存”，保存作业。单击“启动”，启动作业，确认作业参数信息，单击“立即启动”开始执行作业。等待作业运行状态变为“运行中”。

步骤 6：发送数据和查询结果

1. 登录RDS管理控制台，在“实例管理”界面，选择已创建的RDS MySQL实例，选择操作列的“更多 > 登录”，进入数据管理服务实例登录界面。
2. 输入实例登录的用户名和密码。单击“登录”，即可进入RDS MySQL数据库并进行管理。
3. 在已创建的数据库的操作列，单击“SQL查询”，输入以下创建表语句，插入测试数据。

```
insert into mysqlcdc values
('202103241000000001','webShop','2021-03-24 10:00:00','100.00','100.00','2021-03-24 10:02:03','0001','Alice','330106'),
('202103241206060001','appShop','2021-03-24 12:06:06','200.00','180.00','2021-03-24 16:10:06','0002','Jason','330106'),
('202103241403000001','webShop','2021-03-24 14:03:00','300.00','100.00','2021-03-24 10:02:03','0003','Lily','330106'),
('202103241636060001','appShop','2021-03-24 16:36:06','200.00','150.00','2021-03-24 16:10:06','0001','Henry','330106');
```

4. 参考**使用gsql命令行客户端连接DWS集群**连接已创建的DWS集群。
5. 执行以下命令连接DWS集群的默认数据库“testdwsdb”：

```
gsql -d testdwsdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
```

6. 执行以下命令，查询DWS的表数据。

```
select * from test.dwsresult;
```

查询结果参考如下：

```
order_channel  pay_amount  real_pay
appShop         400.0       330.0
webShop         400.0       200.0
```

3.2.5 从 PostgreSQL CDC 源表读取数据写入到 DWS

须知

本指导仅适用于Flink 1.12版本。

场景描述

CDC是变更数据捕获（Change Data Capture）技术的缩写，它可以将源数据库的增量变动记录，同步到一个或多个数据目的中。CDC在数据同步过程中，还可以对数据进行一定的处理，例如分组（GROUP BY）、多表的关联（JOIN）等。

本示例通过创建PostgreSQL CDC源表来监控Postgres的数据变化，并将变化的数据信息插入到DWS数据库中。

前提条件

1. 已创建RDS Postgres实例。本示例创建的RDS Postgres数据库版本选择为：11。
   具体步骤可参考：**RDS PostgreSQL快速入门**。

   说明

   创建的RDS Postgres数据库版本不能低于11。
2. 已创建DWS实例。
   具体创建DWS集群的操作可以参考**创建DWS集群**。

整体作业开发流程

整体作业开发流程参考**图3-6**。

图 3-6 作业开发流程

步骤1：创建队列：创建DLI作业运行的队列。

步骤2：创建RDS Postgres数据库：创建RDS Postgres的数据库和表。

步骤3：创建DWS数据库和表：创建用于接收数据的DWS数据库和表。

步骤4：创建增强型跨源连接：DLI上创建连接RDS和DWS的跨源连接，打通网络。

步骤5：运行作业：DLI上创建和运行Flink OpenSource作业。

步骤6：发送数据和查询结果：RDS Postgres的表上插入数据，在DWS上查看运行结果。

步骤 1：创建队列

1. 登录DLI管理控制台，在左侧导航栏单击“资源管理 > 队列管理”，可进入队列管理页面。
2. 在队列管理界面，单击界面右上角的“购买队列”。
3. 在“购买队列”界面，填写具体的队列配置参数，具体参数填写参考如下。
   - 计费模式：选择“包年/包月”或“按需计费”。本示例选择“按需计费”。
   - 区域和项目：保持默认值即可。
   - 名称：填写具体的队列名称。

> **说明**
>
> 新建的队列名称，名称只能包含数字、英文字母和下划线，但不能是纯数字，且不能以下划线开头。长度限制：1~128个字符。
>
> **队列名称不区分大小写，系统会自动转换为小写。**

- 类型：队列类型选择“通用队列”。**“按需计费”时需要勾选“专属资源模式”**。
- AZ策略、CPU架构、规格：保持默认即可。
- 企业项目：当前选择为“default”。
- 高级选项：选择“自定义”。
- 网段：配置队列网段。例如，当前配置为10.0.0.0/16。

> **注意**
>
> 队列的网段不能和DMS Kafka、RDS MySQL实例的子网网段有重合，否则后续创建跨源连接会失败。

- 其他参数根据需要选择和配置。

4. 参数配置完成后，单击“立即购买”，确认配置信息无误后，单击“提交”完成队列创建。

步骤 2：创建 RDS Postgres 数据库

1. 登录RDS管理控制台，在“实例管理”界面，选择已创建的RDS Postgres实例，选择操作列的“更多 > 登录”，进入数据管理服务实例登录界面。
2. 输入实例登录的用户名和密码。单击“登录”，即可进入RDS Postgres数据库并进行管理。
3. 新建数据库实例testrdsdb。
4. 在testrdsdb数据库下，新建名称为test的Schema。
5. 单击“SQL操作 > SQL查询 ”，进入SQL查询页面创建RDS Postgres表。

```
create table test.cdc_order(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
  user_name VARCHAR,
  area_id VARCHAR,
  primary key(order_id));
```

在Postgre中执行下列SQL语句。

```
ALTER TABLE test.cdc_order REPLICA IDENTITY FULL;
```

步骤 3：创建 DWS 数据库和表

1. 连接已创建的DWS集群。

   请参考**使用gsql命令行客户端连接DWS集群**。

2. 执行以下命令连接DWS集群的默认数据库“gaussdb”：

```
gsql -d gaussdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
```

- gaussdb：DWS集群默认数据库。

- DWS集群连接地址：请参见**获取集群连接地址**进行获取。如果通过公网地址连接，请指定为集群“公网访问地址”或“公网访问域名”，如果通过内网地址连接，请指定为集群“内网访问地址”或“内网访问域名”。如果通过弹性负载均衡连接，请指定为“弹性负载均衡地址”。
- dbadmin：创建集群时设置的默认管理员用户名。
- -W：默认管理员用户的密码。

3. 在命令行窗口输入以下命令创建数据库“testdwsdb”。
```
CREATE DATABASE testdwsdb;
```
4. 执行以下命令，退出gaussdb数据库，连接新创建的数据库“testdwsdb”。
```
\q
gsql -d testdwsdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
```
5. 执行以下命令创建表。
```
create schema test;
set current_schema= test;
drop table if exists dws_order;
CREATE TABLE dws_order
(
  order_id VARCHAR,
  order_channel VARCHAR,
  order_time VARCHAR,
  pay_amount FLOAT8,
  real_pay FLOAT8,
  pay_time VARCHAR,
  user_id VARCHAR,
  user_name VARCHAR,
  area_id VARCHAR
);
```

步骤 4：创建增强型跨源连接

- **创建DLI连接RDS的增强型跨源连接**

a. 在RDS管理控制台，选择“实例管理”，单击对应的RDS实例名称，进入到RDS的基本信息页面。

b. 在“基本信息”的“连接信息”中获取该实例的“内网地址”、“数据库端口”、“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

c. 单击“连接信息”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

e. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

- 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_rds。
- 弹性资源池：选择**步骤1：创建队列**中已经创建的队列。
- 虚拟私有云：选择RDS的虚拟私有云。
- 子网：选择RDS的子网。
- 其他参数可以根据需要选择配置。

参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

f. 单击“资源管理 > 队列管理 ”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

g. 在“测试连通性”界面，根据**b**中获取的RDS连接信息，地址栏输入“RDS内网地址:RDS数据库端口”，单击“测试”测试DLI到RDS网络是否可达。

- **创建DLI连接DWS的增强型跨源连接**

a. 在DWS管理控制台，选择“集群管理”，单击已创建的DWS集群名称，进入到DWS的基本信息页面。

b. 在“基本信息”的“数据库属性”中获取该实例的“内网IP”、“端口”，“基本信息”页面的“网络”中获取“虚拟私有云”和“子网”信息，方便后续操作步骤使用。

c. 单击“连接信息”中的安全组名称，在“入方向规则”中添加放通队列网段的规则。例如，本示例队列网段为“10.0.0.0/16”，则规则添加为：优先级选择：1，策略选择：允许，协议选择：TCP，端口值不填，类型：IPv4，源地址为：10.0.0.0/16，单击“确定”完成安全组规则添加。

d. RDS和DWS实例属于同一VPC和子网下?

i. 是，执行**g**。RDS和DWS实例在同一VPC和子网，不用再重复创建增强型跨源连接。

ii. 否，执行**e**。RDS和DWS实例分别在两个VPC和子网下，则要分别创建增强型跨源连接打通网络。

e. 登录DLI管理控制台，在左侧导航栏单击“跨源管理”，在跨源管理界面，单击“增强型跨源”，单击“创建”。

f. 在增强型跨源创建界面，配置具体的跨源连接参数。具体参考如下。

- 连接名称：设置具体的增强型跨源名称。本示例输入为：dli_dws。
- 弹性资源池：选择**步骤1：创建队列**中已经创建的队列。
- 虚拟私有云：选择DWS的虚拟私有云。
- 子网：选择DWS的子网。
- 其他参数可以根据需要选择配置。

参数配置完成后，单击“确定”完成增强型跨源配置。单击创建的跨源连接名称，查看跨源连接的连接状态，等待连接状态为：“已激活”后可以进行后续步骤。

g. 单击“资源管理 > 队列管理”，选择操作的队列，本示例为**步骤1：创建队列**中创建的队列，在操作列，单击“更多 > 测试地址连通性”。

h. 在“测试连通性”界面，根据**b**中获取的DWS连接信息，地址栏输入“DWS内网IP:DWS端口”，单击“测试”测试DLI到DWS网络是否可达。

步骤 5：运行作业

1. 在DLI管理控制台，单击“作业管理 > Flink作业”，在Flink作业管理界面，单击“创建作业”。
2. 在创建队列界面，类型选择“Flink OpenSource SQL”，名称填写为：FlinkCDCPostgreDWS。单击“确定”，跳转到Flink作业编辑界面。
3. 在Flink OpenSource SQL作业编辑界面，配置如下参数，其他参数默认即可。
   - 所属队列：选择**步骤1：创建队列**中创建的队列。

- Flink版本：选择1.12。
- 保存作业日志：勾选。
- OBS桶：选择保存作业日志的OBS桶，根据提示进行OBS桶权限授权。
- 开启Checkpoint：勾选。
- Flink作业编辑框中输入具体的作业SQL，本示例作业参考如下。SQL中加粗的参数需要根据实际情况修改。

说明

本示例使用的Flink版本为1.12，故Flink OpenSource SQL语法也是1.12。本示例数据源是Kafka，写入结果数据到Elasticsearch。

请参考**Flink OpenSource SQL 1.12创建Postgres CDC源表**和**Flink OpenSource SQL 1.12创建DWS结果表**。

表 3-1 作业运行参数说明

参数	参数说明
所属队列	默认选择“共享队列”，可以按需选择自定义的CCE独享队列，并配置以下参数。 “UDF Jar”：用户自定义UDF文件，在选择UDF Jar之前需要将对应的jar包上传至OBS桶中，并在“数据管理>程序包管理”中创建程序包，具体操作请参考**创建程序包**。 用户可以在SQL中调用插入Jar包中的自定义函数。 **说明** 当子用户在创建作业时，子用户只能选择已经被分配的队列。 当所选择队列的剩余容量不能满足作业需求时，系统会自动扩容，将按照增加的容量计费。当队列空闲时，系统也会自动缩容。
Flink版本	具体参数说明如下： • 1.10：具体SQL语法参考**Flink OpenSource SQL1.10语法**。 • 1.12：具体SQL语法参考**Flink OpenSource SQL1.12语法**。
CU数量	CU数量为DLI的计算单元数量和管理单元数量总和，CU也是DLI的计费单位，1CU=1核4G。 当前配置的CU数量为运行作业时所需的CU数，不能超过其绑定队列的CU数量。
管理单元	管理单元CU数量。
并行数	并行数是指同时运行Flink OpenSource SQL作业的最大任务数。 **说明** 最大并行数不能大于计算单元（CU数量-管理单元）的4倍。

参数	参数说明
TaskManager配置	用于设置TaskManager资源参数。 勾选后需配置下列参数： ● “单TM所占CU数”：每个TaskManager占用的资源数量。 ● “单TM Slot”：每个TaskManager包含的Slot数量。
OBS桶	选择OBS桶用于保存用户作业日志信息、checkpoint等信息。如果选择的OBS桶是未授权状态，需要单击“OBS授权”。
保存作业日志	设置是否将作业运行时的日志信息保存到OBS。日志信息的保存路径为：“桶名/jobs/logs/作业id开头的目录”。 **注意** 该参数建议勾选，否则作业运行完成后不会生成运行日志，后续如果作业运行异常则无法获取运行日志进行定位。 勾选后需配置下列参数： “OBS桶”：选择OBS桶用于保存用户作业日志信息。如果选择的OBS桶是未授权状态，需要单击“OBS授权”。 **说明** 如果同时勾选了“开启Checkpoint”和“保存作业日志”，OBS授权一次即可。
作业异常告警	设置是否将作业异常告警信息，如作业出现运行异常或者欠费情况，以SMN的方式通知用户。 勾选后需配置下列参数： “SMN主题”： 选择一个自定义的SMN主题。如何自定义SMN主题，请参见《消息通知服务用户指南》中“创建主题”章节。
开启Checkpoint	设置是否开启作业快照，开启后可基于Checkpoint（一致性检查点）恢复作业。 勾选后需配置下列参数： ● “Checkpoint间隔”：Checkpoint的时间间隔，单位为秒，输入范围 1~999999，默认值为30s。 ● “Checkpoint模式”：支持如下两种模式： – At least once：事件至少被处理一次。 – Exactly once：事件仅被处理一次。 ● “OBS桶”：选择OBS桶用于保存用户Checkpoint。如果选择的OBS桶是未授权状态，需要单击“OBS授权”。 Checkpoint保存路径为：“桶名/jobs/checkpoint/作业id开头的目录”。 **说明** 如果同时勾选了“开启Checkpoint”和“保存作业日志”，OBS授权一次即可。

参数	参数说明
异常自动重启	设置是否启动异常自动重启功能，当作业异常时将自动重启并恢复作业。 勾选后需配置下列参数： ● “异常重试最大次数”：配置异常重试最大次数。单位为“次/小时”。 – 无限：无限次重试。 – 有限：自定义重试次数。 ● “从Checkpoint恢复”：需要同时勾选“开启Checkpoint”才可配置该参数。
空闲状态保留时长	用于清除GroupBy或Window经过最大保留时间后仍未更新的中间状态，默认设置为1小时。
脏数据策略	选择处理脏数据的策略。支持如下三种策略：“忽略”，“抛出异常”和“保存”。 “脏数据策略”选择“保存”时，配置“脏数据转储地址”。单击地址框选择保存脏数据的OBS路径。

```
create table PostgreCdcSource(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key (order_id) not enforced
) with (
  'connector' = 'postgres-cdc',
  'hostname' = '192.168.15.153',--IP替换为RDS Postgres的实例IP
  'port' = '5432',--端口替换为RDS Postgres的实例端口
  'pwd_auth_name'= 'xxxxx', --DLI侧创建的Password类型的跨源认证名称。使用跨源认证则无需在作业中配置账号和密码。
  'database-name' = 'testrdsdb',--RDS Postgres实例的数据库名
  'schema-name' = 'test',--RDS Postgres数据库下的schema
  'table-name' = 'cdc_order'--RDS Postgres数据库下的表名
);

create table dwsSink(
  order_id string,
  order_channel string,
  order_time string,
  pay_amount double,
  real_pay double,
  pay_time string,
  user_id string,
  user_name string,
  area_id STRING,
  primary key(order_id) not enforced
) with (
  'connector' = 'gaussdb',
  'driver' = 'com.huawei.gauss200.jdbc.Driver',
  'url' = 'jdbc:gaussdb://192.168.168.16:8000/testdwsdb', ---192.168.168.16:8000替换为DWS的内网IP和端口，testdwsdb为创建的DWS数据库名
  'table-name' = 'test\".\"dws_order', ---test为创建的DWS表的schema，dws_order为对应的DWS
```

```
表名
  'username' = 'xxxxx',--替换为DWS实例的用户名
  'password' = 'xxxxx',--替换为DWS实例的用户密码
  'write.mode' = 'insert'
);

insert into dwsSink select * from PostgreCdcSource where pay_amount > 100;
```

4. 单击“语义校验”确保SQL语义校验成功。单击“保存”，保存作业。单击“启动”，启动作业，确认作业参数信息，单击“立即启动”开始执行作业。等待作业运行状态变为“运行中”。

步骤 6：发送数据和查询结果

1. 登录RDS管理控制台，在“实例管理”界面，选择已创建的RDS Postgres实例，选择操作列的“更多 > 登录”，进入数据管理服务实例登录界面。
2. 输入实例登录的用户名和密码。单击“登录”，即可进入RDS Postgres数据库并进行管理。
3. 在已创建的数据库的操作列，单击“SQL查询”，输入以下创建表语句，插入测试数据。

```
insert into test.cdc_order values
('202103241000000001','webShop','2021-03-24 10:00:00','50.00','100.00','2021-03-24
10:02:03','0001','Alice','330106'),
('202103251606060001','appShop','2021-03-24 12:06:06','200.00','180.00','2021-03-24
16:10:06','0002','Jason','330106'),
('202103261000000001','webShop','2021-03-24 14:03:00','300.00','100.00','2021-03-24
10:02:03','0003','Lily','330106'),
('202103271606060001','appShop','2021-03-24 16:36:06','99.00','150.00','2021-03-24
16:10:06','0001','Henry','330106');
```

4. 连接已创建的DWS集群。

   请参考**使用gsql命令行客户端连接DWS集群**。

5. 执行以下命令连接DWS集群的默认数据库“testdwsdb”：

```
gsql -d testdwsdb -h DWS集群连接地址 -U dbadmin -p 8000 -W password -r
```

6. 执行以下语句，查询DWS的表数据。

```
select * from test.dws_order;
```

   查询结果参考如下：

```
order_channel             order_channel     order_time            pay_amount  real_pay
pay_time            user_id  user_name  area_id
202103251606060001        appShop        2021-03-24 12:06:06       200.0       180.0    2021-03-24
16:10:06       0002       Jason      330106
202103261000000001        webShop        2021-03-24 14:03:00       300.0       100.0    2021-03-24
10:02:03       0003       Lily       330106
```

3.2.6 Flink 作业高可靠推荐配置指导（异常自动重启）

操作场景

本节操作介绍创建Flink作业时，配置流应用实现高可靠性能的操作方法。

操作步骤

1. 用户在消息通知服务（SMN）中提前创建一个“主题”，并将其指定的邮箱或者手机号添加至主题订阅中。此时指定的邮箱或者手机会收到请求订阅的通知，单击链接确认订阅即可。

图 3-7 创建主题

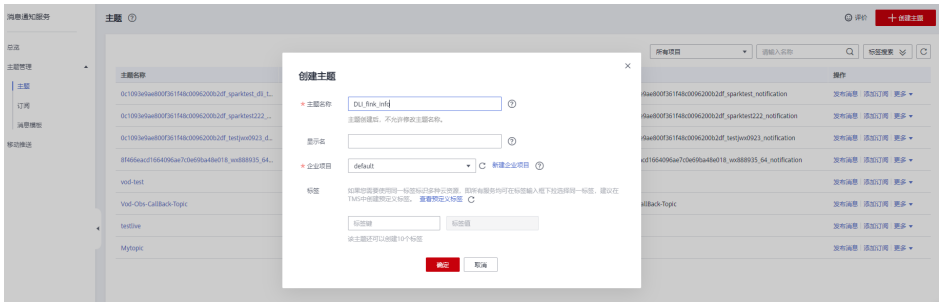


图 3-8 添加订阅

2. 登录DLI控制台，创建Flink作业，编写作业SQL后，配置“运行参数”。本例对重点参数加以说明，其他参数根据业务情况自行配置即可。

   说明

   Flink Jar作业可靠性配置与SQL作业相同，不再另行说明。

   a. 根据如下公式，配置作业的“CU数量”、“管理单元”与“最大并行数”：

      CU数量 = 管理单元 + (算子总并行数 / 单TM Slot数) * 单TM所占CU数

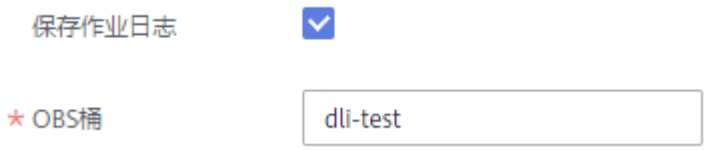
      例如：CU数量为9CU，管理单元为1CU，最大并行数为16，则计算单元为8CU。

      如果不手动配置TaskManager资源，则单TM所占CU数默认为1，单TM slot数显示值为0，但实际上，单TM slot数值依据上述公式计算结果为2。

      如果手动配置TaskManager资源，请依据上述公式计算配置，建议作业最大并行数为计算单元2倍为宜。

   b. 勾选“保存作业日志”，选择一个OBS桶。如果该桶未授权，需要单击“立即授权”进行授权。配置该参数，可以在作业异常失败后，将作业日志保存到用户的OBS桶下，方便用户定位故障原因。

      图 3-9 保存作业日志

c. 勾选“作业异常告警”，选择1中创建的“SMN主题”。配置该参数，可以在作业异常情况下，向用户指定邮箱或者手机发送消息通知，方便客户及时感知异常。

图 3-10 作业异常告警

d. 勾选“开启Checkpoint”，依据自身业务情况调整Checkpoint间隔和模式。Flink Checkpoint机制可以保证Flink任务突然失败时，能够从最近的Checkpoint进行状态恢复重启。

图 3-11 checkpoint 参数

说明

- “Checkpoint间隔”为两次触发Checkpoint的间隔，执行Checkpoint机制会影响实时计算性能，配置间隔时间需权衡对业务的性能影响及恢复时长，**最好大于Checkpoint的完成时间**，建议设置为5分钟。
- Exactly Once模式保证每条数据只被消费一次，At Least Once模式每条数据至少被消费一次，请依据业务情况选择。

e. 勾选“异常自动恢复”和“从Checkpoint恢复”，根据自身业务情况选择重试次数。

f. 配置“脏数据策略”，依据自身的业务逻辑和数据特征选择忽略、抛出异常或者保存脏数据。

g. 选择“运行队列”。提交并运行作业。

3. 登录云监控服务CES控制台，在“云服务监控”列表中找到“数据湖探索”服务。在Flink作业中找到目标作业，单击“创建告警规则”。创建警告规则具体步骤请参考《**创建告警规则**》。

   DLI 为Flink作业提供了丰富的监控指标，用户可以依据自身需求使用不同的监控指标定义告警规则，实现更细粒度的作业监控。

   监控指标说明请参考《数据湖探索用户指南》>《**数据湖探索监控指标说明**》。

3.3 Flink Jar 作业开发基础样例

概述

用户可以基于Flink的API进行二次开发，构建自己的应用Jar包，提交到DLI队列运行，实现与MRS Kafka、HBase、Hive、HDFS，DWS，DCS等数据源的交互。

本章节以通过自定义作业与MRS进行交互为例进行说明。

更多样例代码请通过**DLI样例代码**获取。

环境准备

1. 登录MRS管理控制台，创建MRS集群，选择“开启kerberos”，勾选“Kafka”，“HBase”，“HDFS”等。请参见**《MapReduce服务用户指南》**的“购买自定义集群”的章节创建MRS。
2. “安全组规则”开通对应UDP/TCP端口。详细内容请参考**《私有云用户指南》**中的“添加安全组规则”章节。
3. 进入MRS manager管理界面：
   a. 创建机机账号，需确保该用户含有“hdfs_admin”，“hbase_admin”权限，下载该用户认证凭据，其中包含“user.keytab” 和 “krb5.conf” 文件。

      说明

      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。

   b. 单击“服务管理”，下载客户端，单击“确定”。
   c. 在MRS节点上下载配置文件，所需集群配置文件包含“hbase-site.xml”和“hiveclient.properties”。
4. 创建弹性资源池和队列。

   弹性资源池与队列为DLI作业提供计算资源，**创建弹性资源池**，**弹性资源池添加队列**。
5. 使用该DLI独享队列与MRS集群建立增强型跨源连接，且用户可以根据实际所需设置相应安全组规则。

   如何建立增强型跨源连接，请参考《数据湖探索用户指南》中**增强型跨源连接**章节。

   如何设置安全组规则，请参见**《虚拟私有云用户指南》**中“安全组”章节。
6. 获取MRS集群全部节点的ip和域名映射，在DLI跨源连接修改主机信息中配置host映射。

   如何添加IP域名映射，请参见《数据湖探索用户指南》中“**修改主机信息**”章节。

   说明

   Kafka服务端的端口如果监听在hostname上，则需要将Kafka Broker节点的hostname和IP的对应关系添加到DLI队列中。Kafka Broker节点的hostname和IP请联系Kafka服务的部署人员。

前提条件

- 确保已创建弹性资源池和队列。
- 用户运行Flink Jar作业时，需要将二次开发的应用代码构建为Jar包，上传到已经创建的OBS桶中。并在DLI“数据管理”>“程序包管理”页面创建程序包，具体请参考**创建程序包**。

  说明

  DLI不支持下载功能，如果需要更新已上传的数据文件，可以将本地文件更新后重新上传。

- 由于DLI服务端已经内置了Flink的依赖包，并且基于开源社区版本做了安全加固。
  为了避免依赖包兼容性问题或日志输出及转储问题，打包时请注意排除以下文件：
  - 系统内置的依赖包，或者在Maven或者Sbt构建工具中将scope设为provided
  - 日志配置文件（例如：“log4j.properties”或者“logback.xml”等）
  - 日志输出实现类JAR包（例如：log4j等）
- 上传自定义配置到OBS的桶必须为主账号下所创建的OBS桶。
- 使用Flink1.15版本的计算引擎时，需要用户自行配置委托，否则可能影响作业运行。
  详细操作请参考**自定义DLI委托权限**。

使用方法

创建并提交Flink jar作业，详细操作步骤请参见《数据湖探索用户指南》中**创建Flink Jar作业**章节。

步骤1 在DLI管理控制台的左侧导航栏中，单击“作业管理”>“Flink作业”，进入“Flink作业”页面。

步骤2 在“Flink作业”页面右上角单击“新建作业”，弹出“新建作业”对话框。

图 3-12 新建 Flink Jar 作业

步骤3 配置作业信息。

表 3-2 作业配置信息

参数	参数说明
类型	选择Flink Jar。
名称	作业名称，只能由英文、中文、数字、中划线和下划线组成，并且长度为1～57字节。 **说明** 作业名称必须是唯一的。
描述	作业的相关描述，且长度为0～512字节。
标签	使用标签标识云资源。包括“标签键”和“标签值”。如果您需要使用同一标签标识多种云资源，即所有服务均可在标签输入框下拉选择同一标签，建议在标签管理服务（TMS）中创建预定义标签。 具体请参考《**标签管理服务用户指南**》。 **说明** • 最多支持20个标签。 • 一个“键”只能添加一个“值”。 • 标签键：在输入框中输入标签键名称。 **说明** – 标签键的最大长度为36个字符，不能包含“=”，“*”，“,”，“<”，“>”，“\”，“|”，“/”，且首尾字符不能为空格。 – 若有预定义标签，可在输入框的下拉列表中进行选择。 • 标签值：在输入框中输入标签值。 **说明** – 标签值的最大长度为43个字符，不能包含“=”，“*”，“,”，“<”，“>”，“\”，“|”，“/”，且首尾字符不能为空格。 – 若有预定义标签，可在输入框的下拉列表中进行选择。

步骤4 单击“确定”，进入“编辑”页面。

步骤5 选择队列。Flink Jar作业只能运行在通用队列上。

图 3-13 选择队列

步骤6 上传Jar包。

Flink版本需要和用户Jar包指定的Flink版本保持一致。

图 3-14 上传 Jar 包

表 3-3 参数说明

名称	描述
应用程序	用户自定义的程序包。在选择程序包之前需要将对应的Jar包上传至OBS桶中，并在“数据管理>程序包管理”中创建程序包，，具体请参考**创建程序包**。
主类	指定加载的Jar包类名，如KafkaMessageStreaming。 • 默认：根据Jar包文件的Manifest文件指定。 • 指定：必须输入“类名”并确定类参数列表（参数间用空格分隔）。 **说明** 当类属于某个包时，需携带包路径，例如：packagePath.KafkaMessageStreaming
参数	指定类的参数列表，参数之间使用空格分隔。
依赖jar包	用户自定义的依赖程序包。在选择程序包之前需要将对应的Jar包上传至OBS桶中，并在“数据管理>程序包管理”中创建程序包，包类型选择“jar”，具体请参考**创建程序包**。

名称	描述
其他依赖文件	用户自定义的依赖文件。在选择依赖文件之前需要将对应的文件上传至OBS桶中，并在“数据管理>程序包管理”中创建程序包，包类型没有限制，具体请参考**创建程序包**。 通过在应用程序中添加以下内容可访问对应的依赖文件。其中，“fileName”为需要访问的文件名，“ClassName”为需要访问该文件的类名。 `ClassName.class.getClassLoader().getResource("userData/fileName")`
作业特性	队列为CCE队列时，显示该参数。 • 基础型 • 自定义镜像：选择镜像名称和镜像版本。用户可在“容器镜像服务”设置的镜像。具体操作请参考《**容器镜像服务用户指南**》。
Flink版本	选择Flink版本前，需要先选择所属的队列。推荐选择Flink 1.15版本。

步骤7 配置作业参数。

注意

Flink Jar作业最小化提交是指Flink仅提交作业必须的依赖项，而不是整个Flink环境。通过将非Connector的Flink依赖项（以flink-开头）和第三方库（如Hadoop、Hive、Hudi、Mysql-cdc）的作用域设置为provided，可以确保这些依赖项不会被包含在Jar作业中，从而实现最小化提交，避免依赖包与flink内核中依赖包冲突：

- 仅Flink 1.15版本支持Flink Jar作业最小化提交，通过在运行优化参数中配置flink.dli.job.jar.minimize-submission.enabled=true可以开启最小化提交。
- Flink相关依赖作用域请使用provided，即在依赖中添加<scope>provided</scope>。主要包含org.apache.flink组下以flink-开头的非Connector依赖。
- Hadoop、Hive、Hudi、Mysql-cdc相关依赖，作用域请使用provided，即在依赖中添加<scope>provided</scope>。
- Flink源代码中只有明确标注了@Public或者@PublicEvolving的才是公开供用户调用的方法，DLI只对这些方法的兼容性做出产品保证。

图 3-15 配置参数

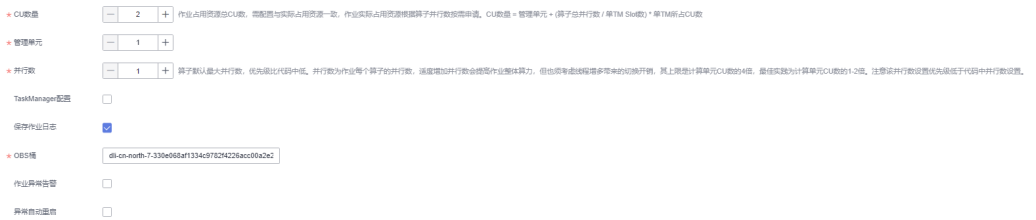


表 3-4 参数说明

名称	描述
CU数量	一个CU为1核4G的资源量。CU数量范围为2~400个。
管理单元	设置管理单元的CU数，支持设置1~4个CU，默认值为1个CU。
并行数	作业中每个算子的最大并行数。 **说明** • 并行数不能大于计算单元（CU数量-管理单元CU数量）的4倍。 • 并行数最好大于用户作业里设置的并发数，否则有可能提交失败。
TaskManager配置	用于设置TaskManager资源参数。 勾选后需配置下列参数： • “单TM所占CU数”：每个TaskManager占用的资源数量。 • “单TM Slot”：每个TaskManager包含的Slot数量。
保存作业日志	设置是否将作业运行时的日志信息保存到OBS。 勾选后需配置下列参数： “OBS桶”：选择OBS桶用于保存用户作业日志信息。如果选择的OBS桶是未授权状态，需要单击“OBS授权”。
作业异常告警	设置是否将作业异常告警信息，如作业出现运行异常或者欠费情况，以SMN的方式通知用户。 勾选后需配置下列参数： “SMN主题”： 选择一个自定义的SMN主题。如何自定义SMN主题，请参见《消息通知服务用户指南》中“创建主题”章节。
异常自动重启	设置是否启动异常自动重启功能，当作业异常时将自动重启并恢复作业。 勾选后需配置下列参数： • “异常重试最大次数”：配置异常重试最大次数。单位为“次/小时”。 – 无限：无限次重试。 – 有限：自定义重试次数。 • “从Checkpoint恢复”：从最新保存的checkpoint恢复作业。勾选该参数后，还需要选择“Checkpoint路径”。 “Checkpoint路径”：选择checkpoint保存路径。必须和应用程序中配置的Checkpoint地址相对应。且不同作业的路径不可一致，否则无法获取准确的Checkpoint。

步骤8 单击右上角“保存”，保存作业和相关参数。

步骤9 单击右上角“启动”，进入“启动Flink作业”页面，确认作业规格和费用，单击“立即启动”，启动作业。

启动作业后，系统将自动跳转到Flink作业管理页面，新创建的作业将显示在作业列表中，在“状态”列中可以查看作业状态。作业提交成功后，状态将由“提交中”变为“运行中”。运行完成后显示“已完成”。

如果作业状态为“提交失败”或“运行异常”，表示作业提交或运行失败。用户可以在作业列表中的“状态”列中，将鼠标移动到状态图标上查看错误信息，单击可以复制错误信息。根据错误信息解决故障后，重新提交。

说明

其他功能按钮说明如下：

另存为：将新建作业另存为一个新作业。

----结束

相关操作

- **怎样设置作业的参数?**
  a. 在FLink作业列表中选择待编辑的作业。
  b. 单击操作列“编辑”。
  c. 在参数区域输入参数信息。
  指定类的参数列表，参数之间使用空格分隔。
  参数输入格式：--key1 value1 --key2 value2
  例如：控制台入输入的参数
  --bootstrap.server 192.168.168.xxx:9092
  通过ParameterTool解析后的参数如下所示：

  图 3-16 解析后的参数

- **怎样查看作业日志?**
  a. 在FLink作业列表中单击作业名称，进入作业详情页面。
  b. 单击“运行日志”，即可在控制台查看作业日志。
  此处只展示最新的运行日志，更多信息请查看保存日志的OBS桶。

3.4 使用 Flink Jar 写入数据到 OBS 开发指南

概述

DLI提供了使用自定义Jar运行Flink作业并将数据写入到OBS的能力。本章节JAVA样例代码演示将kafka数据处理后写入到OBS，具体参数配置请根据实际环境修改。

环境准备

已安装和配置IntelliJ IDEA等开发工具以及安装JDK和Maven。

说明

- Maven工程的pom.xml文件配置请参考**JAVA样例代码（Flink 1.12）**中“pom文件配置”说明。
- 确保本地编译环境可以正常访问公网。

约束与限制

- 需要在DLI控制台下“全局配置 > 服务授权”开启Tenant Adminstrator（全局服务）。
- 写入数据到OBS的桶必须为主账号下所创建的OBS桶。
- 使用Flink1.15版本的计算引擎时，需要用户自行配置委托，否则可能影响作业运行。

  详细操作请参考**自定义DLI委托权限**。

Java 样例代码（Flink 1.15）

- pom文件配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.huaweicloud</groupId>
        <artifactId>dli-flink-demo</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <groupId>org.example</groupId>
    <artifactId>flink-1.15-demo</artifactId>
    <properties>
        <flink.version>1.15.0</flink.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-statebackend-rocksdb</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-table-planner_2.12</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.14.2</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
```



```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
               <execution>
                  <phase>package</phase>
                  <goals>
                     <goal>single</goal>
                  </goals>
               </execution>
            </executions>
            <configuration>
               <archive>
                  <manifest>
                     <mainClass>com.huawei.dli.GetUserConfigFileDemo</mainClass>
                  </manifest>
               </archive>
               <descriptorRefs>
                  <descriptorRef>jar-with-dependencies</descriptorRef>
               </descriptorRefs>
            </configuration>
         </plugin>
      </plugins>
      <resources>
         <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
            <includes>
               <include>**/*.*</include>
            </includes>
         </resource>
      </resources>
   </build>
</project>
```

- 示例代码

```
package com.huawei.dli;

import com.huawei.dli.source.CustomParallelSource;

import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.EmbeddedRocksDBStateBackend;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.OnCheckpointRollingPolicy;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;

public class GetUserConfigFileDemo {
   private static final Logger LOG = LoggerFactory.getLogger(GetUserConfigFileDemo.class);

   public static void main(String[] args) {
      try {
         ParameterTool params = ParameterTool.fromArgs(args);
         LOG.info("Params: " + params.toString());

         StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
```



```
        // set checkpoint
        String checkpointPath = params.get("checkpoint.path", "obs://bucket/checkpoint/
jobId_jobName/");
        LocalDateTime localDateTime = LocalDateTime.ofEpochSecond(System.currentTimeMillis() /
1000,
            0, ZoneOffset.ofHours(8));
        String dt = localDateTime.format(DateTimeFormatter.ofPattern("yyyyMMdd_HH:mm:ss"));
        checkpointPath = checkpointPath + dt;

        streamEnv.setStateBackend(new EmbeddedRocksDBStateBackend());
        streamEnv.getCheckpointConfig().setCheckpointStorage(checkpointPath);
        streamEnv.getCheckpointConfig().setExternalizedCheckpointCleanup(
            CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
        streamEnv.enableCheckpointing(30 * 1000);

        DataStream<String> stream = streamEnv.addSource(new CustomParallelSource())
            .setParallelism(1)
            .disableChaining();

        String outputPath = params.get("output.path", "obs://bucket/outputPath/jobId_jobName");

        // Get user dependents config
        URL url = GetUserConfigFileDemo.class.getClassLoader().getResource("userData/user.config");
        if (url != null) {
            Path filePath = org.apache.flink.util.FileUtils.absolutizePath(new Path(url.getPath()));
            try {
                String config = org.apache.flink.util.FileUtils.readFileUtf8(new File(filePath.getPath()));
                LOG.info("config is {}", config);
                // Do something by config
            } catch (IOException e) {
                LOG.error(e.getMessage(), e);
            }
        }

        // Sink OBS
        final StreamingFileSink<String> sinkForRow = StreamingFileSink
            .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
            .withRollingPolicy(OnCheckpointRollingPolicy.build())
            .build();

        stream.addSink(sinkForRow);

        streamEnv.execute("sinkForRow");
      } catch (Throwable e) {
        LOG.error(e.getMessage(), e);
      }
    }
}
```

表 3-5 参数说明

参数名	具体含义	举例
output.path	数据写入的OBS路径	obs://bucket/output
checkpoint.path	checkpoint的OBS路径	obs://bucket/checkpoint

JAVA 样例代码（Flink 1.12）

- pom文件配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <parent>
        <artifactId>Flink-demo</artifactId>
        <groupId>com.huaweicloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>flink-kafka-to-obs</artifactId>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!--Flink 版本-->
        <flink.version>1.12.2</flink.version>
        <!--JDK 版本-->
        <java.version>1.8</java.version>
        <!--Scala 2.11 版本-->
        <scala.binary.version>2.11</scala.binary.version>
        <slf4j.version>2.13.3</slf4j.version>
        <log4j.version>2.10.0</log4j.version>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- flink -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>

        <!-- kafka -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-kafka_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <!-- logging -->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>${slf4j.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>${log4j.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
```



```
            <artifactId>log4j-core</artifactId>
            <version>${log4j.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-jcl</artifactId>
            <version>${log4j.version}</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>3.3.0</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>com.huaweicloud.dli.FlinkKafkaToObsExample</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
            </plugin>
        </plugins>
        <resources>
            <resource>
                <directory>../main/config</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*.*</include>
                </includes>
            </resource>
        </resources>
    </build>
</project>
```

- 示例代码

```
import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend;
import org.apache.flink.core.fs.Path;
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import
org.apache.flink.streaming.api.functions.sink.filesystem.bucketassigners.DateTimeBucketAssigner;
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.OnCheckpointRollingPolicy;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
import java.util.Properties;

/**
 * @author xxx
 * @date 6/26/21
 */
public class FlinkKafkaToObsExample {
    private static final Logger LOG = LoggerFactory.getLogger(FlinkKafkaToObsExample.class);

    public static void main(String[] args) throws Exception {
        LOG.info("Start Kafka2OBS Flink Streaming Source Java Demo.");
        ParameterTool params = ParameterTool.fromArgs(args);
        LOG.info("Params: " + params.toString());

        // Kafka连接地址
        String bootstrapServers;
        // Kafka消费组
        String kafkaGroup;
        // Kafka topic
        String kafkaTopic;
        // 消费策略，只有当分区没有Checkpoint或者Checkpoint过期时，才会使用此配置的策略；
        //          如果存在有效的Checkpoint，则会从此Checkpoint开始继续消费
        // 取值有： LATEST,从最新的数据开始消费，此策略会忽略通道中已有数据
        //         EARLIEST,从最初的数据开始消费，此策略会获取通道中所有的有效数据
        String offsetPolicy;
        // OBS文件输出路径，格式obs://bucket/path
        String outputPath;
        // Checkpoint输出路径，格式obs://bucket/path
        String checkpointPath;

        bootstrapServers = params.get("bootstrap.servers", "xxxx:9092,xxxx:9092,xxxx:9092");
        kafkaGroup = params.get("group.id", "test-group");
        kafkaTopic = params.get("topic", "test-topic");
        offsetPolicy = params.get("offset.policy", "earliest");
        outputPath = params.get("output.path", "obs://bucket/output");
       checkpointPath = params.get("checkpoint.path", "obs://bucket/checkpoint");

        try {
            // 创建执行环境
            StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
            streamEnv.setParallelism(4);
            RocksDBStateBackend rocksDbBackend = new RocksDBStateBackend(checkpointPath, true);
            RocksDBStateBackend rocksDbBackend = new RocksDBStateBackend(new
FsStateBackend(checkpointPath), true);
            streamEnv.setStateBackend(rocksDbBackend);
            // 开启Flink CheckPoint配置，开启时若触发CheckPoint，会将Offset信息同步到Kafka
            streamEnv.enableCheckpointing(300000);
            // 设置两次checkpoint的最小间隔时间
            streamEnv.getCheckpointConfig().setMinPauseBetweenCheckpoints(60000);
            // 设置checkpoint超时时间
            streamEnv.getCheckpointConfig().setCheckpointTimeout(60000);
            // 设置checkpoint最大并发数
            streamEnv.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
            // 设置作业取消时保留checkpoint
            streamEnv.getCheckpointConfig().enableExternalizedCheckpoints(
                    CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

            // Source: 连接kafka数据源
            Properties properties = new Properties();
            properties.setProperty("bootstrap.servers", bootstrapServers);
            properties.setProperty("group.id", kafkaGroup);
            properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, offsetPolicy);
            String topic = kafkaTopic;

            // 创建kafka consumer
            FlinkKafkaConsumer<String> kafkaConsumer =
                    new FlinkKafkaConsumer<>(topic, new SimpleStringSchema(), properties);
            /**
```

```
            * 从 Kafka brokers 中的 consumer 组（consumer 属性中的 group.id 设置）提交的偏移量中开始读取分区。
            * 如果找不到分区的偏移量，那么将会使用配置中的 auto.offset.reset 设置。
            * 详情 https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/datastream/kafka/
            */
            kafkaConsumer.setStartFromGroupOffsets();

            //将kafka 加入数据源
            DataStream<String> stream = streamEnv.addSource(kafkaConsumer).setParallelism(3).disableChaining();

            // 创建文件输出流
            final StreamingFileSink<String> sink = StreamingFileSink
                    // 指定文件输出路径与行编码格式
                    .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
                    // 指定文件输出路径批量编码格式，以parquet格式输出
                    //.forBulkFormat(new Path(outputPath), ParquetAvroWriters.forGenericRecord(schema))
                    // 指定自定义桶分配器
                    .withBucketAssigner(new DateTimeBucketAssigner<>())
                    // 指定滚动策略
                    .withRollingPolicy(OnCheckpointRollingPolicy.build())
                    .build();

            // Add sink for DIS Consumer data source
            stream.addSink(sink).disableChaining().name("obs");

            // stream.print();
            streamEnv.execute();
        } catch (Exception e) {
            LOG.error(e.getMessage(), e);
        }
    }
}
```

表 3-6 参数说明

参数名	具体含义	举例
bootstrap.servers	kafka连接地址	kafka服务IP地址1:9092,kafka服务IP地址2:9092,kafka服务IP地址3:9092
group.id	kafka消费组	如当前kafka消费组为test-group
topic	kafka消费topic	如当前kafka消费topic为test-topic
offset.policy	kafka的offset策略	如当前kafka的offset策略为earliest
output.path	数据写入的OBS路径	obs://bucket/output
checkpoint.path	checkpoint的OBS路径	obs://bucket/checkpoint

编译运行

应用程序开发完成后，参考**Flink Jar作业开发基础样例**将编译打包的JAR包上传到DLI运行，查看对应OBS路径下是否有相关的数据信息。

3.5 使用 Flink Jar 连接开启 SASL_SSL 认证的 Kafka

概述

本节操作介绍使用Flink Jar连接开启SASL_SSL认证的Kafka的操作方法。

如需使用Flink OpenSource SQL连接开启SASL_SSL认证的Kafka，请参考**Flink SQL语法参考-Kafka源表**。

环境准备

- 已在DLI控制台购买了通用队列。
- 已购买了Kafka实例，并开启了SASL_SSL认证。
- 已在DLI创建增强型跨源并绑定队列，确保DLI队列和Kafka连通。

操作须知

- 连接带SASL_SSL的Kafka，无论是消费者还是生产者，在对应的properties中都需要指定truststore文件的路径。
- 初始化consumer/producer都是在taskmanager里执行的，所以需要获取到taskmanager对应container下truststore文件的路径，在初始化前将其引入properties中才能生效。
- kafka source可以在open里引入。

  图 3-17 获取 kafka source

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.DeserializationSchema;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

import java.util.Properties;

public class SourceKafkaConsumer<T> extends FlinkKafkaConsumer<T> {

    public SourceKafkaConsumer(String topic, DeserializationSchema<T> valueDeserializer, Properties props) {
        super(topic, valueDeserializer, props);
    }

    @Override
    public void open(Configuration configuration) throws Exception {
        String jksPath = SourceKafkaConsumer.class.getClassLoader().getResource( name: "userData/client.jks").getPath();
        super.properties.setProperty("ssl.truststore.location", jksPath);
        super.open(configuration);
    }
}
```

- kafka sink可以在initializeState里引入。

图 3-18 获取 kafka sink

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.SerializationSchema;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;

import java.util.Properties;

public class SinkKafkaProducer<IN> extends FlinkKafkaProducer<IN>{

    public SinkKafkaProducer(String topicId, SerializationSchema<IN> serializationSchema, Properties producerConfig) {
        super(topicId, serializationSchema, producerConfig);
    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {
        String jksPath = SinkKafkaProducer.class.getClassLoader().getResource( name: "userData/client.jks").getPath();
        producerConfig.setProperty("ssl.truststore.location", jksPath);
        super.initializeState(context);
    }
}
```

操作步骤

步骤1 从Kafka实例的基本信息页面下载SSL证书，解压后将其中的clinet.jks文件上传到OBS。

图 3-19 下载 SSL 证书

步骤2 在DLI控制台，打开“数据管理 > 程序包管理”，单击“创建”，创建clinet.jks对应的程序包。

主要参数的填写说明：

- 包类型：File
- OBS路径：clinet.jks所在的OBS路径。
- 分组名称：自定义分组或选择已有的分组名称。

图 3-20 创建 clinet.jks

步骤3 将示例代码打包，在DLI控制台，打开“数据管理 > 程序包管理”，单击“创建”，创建Flink Jar对应的程序包。代码样例请参考**Kafaka ToKafakaExample.java样例**、**•SinkKafkaProducer.java样...**、**SourceKafkaConsumer.java...**。

主要参数的填写说明：

- 包类型：JAR
- OBS路径：Flink Jar所在的OBS路径。
- 分组名称：自定义分组或选择已有的分组名称。

图 3-21 创建 Flink Jar 对应的程序包

步骤4 在DLI控制台，打开“数据管理 > 程序包管理”，单击“创建”，创建KafkaToKafka.properties对应的程序包。代码样例请参考**•KafkaToKafkaExample.pro...**。

主要参数的填写说明：

- 包类型：File
- OBS路径：KafkaToKafka.properties所在的OBS路径。
- 分组名称：自定义分组或选择已有的分组名称。

图 3-22 创建 KafkaToKafka.properties 程序包

步骤5 创建Flink Jar作业并运行。

创建Flink Jar作业，在应用程序、其他依赖文件选择步骤**步骤3**导入Flink Jar，并指定主类。

主要参数的填写说明：

- 所属队列：选择Flink Jar作业运行的队列。
- 应用程序：自定义的程序包
- 主类：指定
- 类名：输入类名并确定类参数列表（参数间用空格分隔）。
- 其他依赖文件：自定义的依赖文件。选择**步骤2**和**步骤4**导入的jks和properties文件。
- Flink版本：1.10

图 3-23 创建 Flink Jar 作业

步骤6 结果校验。

作业处于运行中状态时，向kafka source.topic发送数据，验证kafka sink.topic能否收到数据。

图 3-24 查看作业任务

图 3-25 查看 kafka sink.topic

----结束

JAVA 样例代码

- pom文件配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <parent>
        <artifactId>Flink-demo</artifactId>
        <groupId>com.huaweicloud</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>flink-kafka-to-obs</artifactId>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!--Flink 版本-->
        <flink.version>1.12.2</flink.version>
        <!--JDK 版本-->
        <java.version>1.8</java.version>
        <!--Scala 2.11 版本-->
        <scala.binary.version>2.11</scala.binary.version>
        <slf4j.version>2.13.3</slf4j.version>
        <log4j.version>2.10.0</log4j.version>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- flink -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
```

```
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- kafka -->
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka_2.11</artifactId>
        <version>${flink.version}</version>
    </dependency>

    <!-- logging -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>${slf4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-jcl</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.huaweicloud.dli.FlinkKafkaToObsExample</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
```

```
                </descriptorRefs>
            </configuration>
        </plugin>
    </plugins>
    <resources>
        <resource>
            <directory>../main/config</directory>
            <filtering>true</filtering>
            <includes>
                <include>**/*.*</include>
            </includes>
        </resource>
    </resources>
</build>
</project>
```

- KafkaToKafkaExample.java样例

  userData为固定文件路径名，不支持修改或自定义其他路径名。

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Properties;


public class KafkaToKafkaExample {
    private static final Logger LOG = LoggerFactory.getLogger(KafkaToKafkaExample.class);

    public static void main(String[] args) throws Exception {
        LOG.info("Start Kafka2Kafka Flink Streaming Source Java Demo.");
        String propertiesPath = KafkaToKafkaExample.class.getClassLoader()
            .getResource("userData/KafkaToKafka.properties").getPath();
        ParameterTool params = ParameterTool.fromPropertiesFile(propertiesPath);
        LOG.info("Params: " + params.toString());

        // Kafka连接地址
        String bootstrapServers;
        // Kafka消费组
        String kafkaGroup;
        // Kafka topic
        String sourceTopic;
        String sinkTopic;
        // 消费策略，只有当分区没有Checkpoint或者Checkpoint过期时，才会使用此配置的策略；
        //          如果存在有效的Checkpoint，则会从此Checkpoint开始继续消费
        // 取值有： LATEST,从最新的数据开始消费，此策略会忽略通道中已有数据
        //         EARLIEST,从最初的数据开始消费，此策略会获取通道中所有的有效数据
        String offsetPolicy;
        // SASL_SSL相关配置项。设置jaas账号和密码，username和password为创建Kafka实例过程中开启
SASL_SSL时填入的用户名和密码，
        // 或者创建SASL_SSL用户时设置的用户名和密码。格式如下，
        // org.apache.kafka.common.security.plain.PlainLoginModule required
        // username="yourUsername"
        // password="yourPassword";
        String saslJaasConfig;
        // Checkpoint输出路径，格式obs://bucket/path
        String checkpointPath;

        bootstrapServers = params.get("bootstrap.servers", "xxx:9093,xxx:9093,xxx:9093");
        kafkaGroup = params.get("source.group", "test-group");
        sourceTopic = params.get("source.topic", "test-source-topic");
        sinkTopic = params.get("sink.topic", "test-sink-topic");
```



```
        offsetPolicy = params.get("offset.policy", "earliest");
        saslJaasConfig = params.get("sasl.jaas.config",
            "org.apache.kafka.common.security.plain.PlainLoginModule "
                + "required\nusername=\"yourUsername\"\npassword=\"yourPassword\";");
        checkpointPath = params.get("checkpoint.path", "obs://bucket/path");


        try {
            // 创建执行环境
            StreamExecutionEnvonment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
            RocksDBStateBackend rocksDbBackend = new RocksDBStateBackend(checkpointPath, true);
            streamEnv.setStateBackend(rocksDbBackend);
            // 开启Flink CheckPoint配置，开启时若触发CheckPoint，会将Offset信息同步到Kafka
            streamEnv.enableCheckpointing(300000);
            // 设置两次checkpoint的最小间隔时间
            streamEnv.getCheckpointConfig().setMinPauseBetweenCheckpoints(60000);
            // 设置checkpoint超时时间
            streamEnv.getCheckpointConfig().setCheckpointTimeout(60000);
            // 设置checkpoint最大并发数
            streamEnv.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
            // 设置作业取消时保留checkpoint
            streamEnv.getCheckpointConfig().enableExternalizedCheckpoints(
                CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

            // Source: 连接kafka数据源
            Properties sourceProperties = new Properties();
            sourceProperties.setProperty("bootstrap.servers", bootstrapServers);
            sourceProperties.setProperty("group.id", kafkaGroup);
            sourceProperties.setProperty("sasl.jaas.config", saslJaasConfig);
            sourceProperties.setProperty("sasl.mechanism", "PLAIN");
            sourceProperties.setProperty("security.protocol", "SASL_SSL");
            sourceProperties.setProperty("ssl.truststore.password", "dms@kafka");
            sourceProperties.setProperty("ssl.endpoint.identification.algorithm", "");
            sourceProperties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, offsetPolicy);


            // 创建kafka consumer
            SourceKafkaConsumer<String> kafkaConsumer =
                new SourceKafkaConsumer<>(sourceTopic, new SimpleStringSchema(), sourceProperties);
            /**
             * 从 Kafka brokers 中的 consumer 组（consumer 属性中的 group.id 设置）提交的偏移量中开始
读取分区。
             * 如果找不到分区的偏移量，那么将会使用配置中的 auto.offset.reset 设置。
             * 详情 https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/
datastream/kafka/
             */
            kafkaConsumer.setStartFromGroupOffsets();

            //将kafka 加入数据源
            DataStream<String> stream =
streamEnv.addSource(kafkaConsumer).name("kafka_source").setParallelism(1).disableChaining();

            // Sink: 连接kafka数据汇
            Properties sinkProperties = new Properties();
            sinkProperties.setProperty("bootstrap.servers", bootstrapServers);
            sinkProperties.setProperty("sasl.jaas.config", saslJaasConfig);
            sinkProperties.setProperty("sasl.mechanism", "PLAIN");
            sinkProperties.setProperty("security.protocol", "SASL_SSL");
            sinkProperties.setProperty("ssl.truststore.password", "dms@kafka");
            sinkProperties.setProperty("ssl.endpoint.identification.algorithm", "");

            // 创建kafka producer
            SinkKafkaProducer<String> kafkaProducer = new SinkKafkaProducer<>(sinkTopic, new
SimpleStringSchema(),
                sinkProperties);

            //将kafka 加入数据汇
            stream.addSink(kafkaProducer).name("kafka_sink").setParallelism(1).disableChaining();
```

```
            // stream.print();
            streamEnv.execute();
        } catch (Exception e) {
            LOG.error(e.getMessage(), e);
        }
    }
}
```

- SinkKafkaProducer.java样例

  userData为固定文件路径名，不支持修改或自定义其他路径名。

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.SerializationSchema;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;

import java.util.Properties;

public class SinkKafkaProducer<IN> extends FlinkKafkaProducer<IN>{

    public SinkKafkaProducer(String topicId, SerializationSchema<IN> serializationSchema, Properties
producerConfig) {
        super(topicId, serializationSchema, producerConfig);
    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception {
        String jksPath = SinkKafkaProducer.class.getClassLoader().getResource("userData/
client.jks").getPath();
        producerConfig.setProperty("ssl.truststore.location", jksPath);
        super.initializeState(context);
    }
}
```

- SourceKafkaConsumer.java样例

  userData为固定文件路径名，不支持修改或自定义其他路径名。

```
package kafka_to_kafka;

import org.apache.flink.api.common.serialization.DeserializationSchema;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

import java.util.Properties;

public class SourceKafkaConsumer<T> extends FlinkKafkaConsumer<T> {

    public SourceKafkaConsumer(String topic, DeserializationSchema<T> valueDeserializer, Properties
props) {
        super(topic, valueDeserializer, props);
    }

    @Override
    public void open(Configuration configuration) throws Exception {
        String jksPath = SourceKafkaConsumer.class.getClassLoader().getResource("userData/
client.jks").getPath();
        super.properties.setProperty("ssl.truststore.location", jksPath);
        super.open(configuration);
    }
}
```

- KafkaToKafkaExample.properties样例

```
bootstrap.servers=xxx:9093,xxx:9093,xxx:9093
checkpoint.path=obs://bucket/path
source.group=swq-test-group
source.topic=topic-swq
sink.topic=topic-swq-out
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required\n\
```



```
username="xxxx"\n\
password="xxxx";
```

3.6 使用 Flink Jar 读写 DIS 开发指南

概述

本节操作介绍基于Flink 1.12版本的Flink Jar作业读写DIS数据的操作方法。

Flink 1.12版本Flink Opensource SQL作业不支持使用DLI提供的**connector**读写DIS，因此推荐您使用本节操作提供的方法。

Flink 1.15不再推荐使用DIS服务， 建议搭配DMS kafka使用。请参考**Kafka connector**。

环境准备

- 已在DLI控制台购买了通用队列。
- 已购买了DIS通道。**开通DIS通道**。
- 用户在使用Flink 1.12版本，则依赖的Dis connector版本需要不低于2.0.1，详细代码参考**DISFlinkConnector相关依赖**，如何配置connector，详细参考**自定义Flink Streaming作业**。

  说明

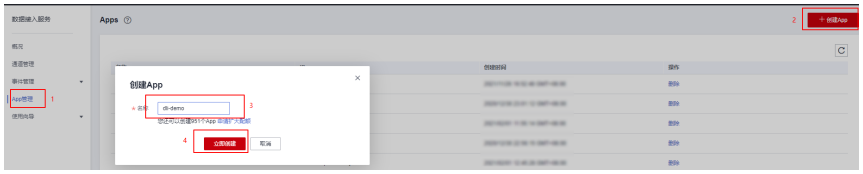
  若读取DIS，且配置groupId，则需要提前在DIS的“App管理”中创建所需的App名称。
- 请勿将disToDis.properties放在生成的jar包中，在代码里有关于disToDis.properties的路径，如果放在jar包中，代码会找不到disToDis.properties路径。

操作步骤

1. 创建所需要的DIS通道，具体流程可参**开通DIS通道**。

   在DIS控制台，打开“App管理 > 创建App”，填写App名称，App名称对应的是代码中的groupId。

   图 3-26 创建 App

2. 创建Flink Jar对应的程序包。

   在DLI控制台，打开“数据管理 > 程序包管理”，单击“创建程序包”，创建Flink Jar对应的程序包。代码样例请参考**FlinkDisToDisExample.java**样例。

表 3-7 创建 Flink Jar 对应的程序包主要参数说明

参数名称	说明	示例
包类型	支持的包类型如下： • JAR：用户jar文件 • PyFile：用户Python文件 • File：用户文件 • ModelFile：用户AI模型文件	JAR
OBS路径	选择对应程序包的OBS路径。 **说明** • 程序包需提前上传至OBS服务中保存。 • 只支持选择文件。	Flink Jar所在的OBS路径
分组名称	• 选择“已有分组”：可选择已有的分组。 • 选择“创建新分组”：可输入自定义的组名称。 • 选择“不分组”：不需要选择或输入组名称。	自定义分组或选择已有的分组名称。

图 3-27 创建对应的 Flink Jar 包

3. 创建distoDis对应的包。

   在DLI控制台，打开“数据管理 > 程序包管理”，单击“创建程序包”，创建disToDis.properties对应的程序包。代码样例请参考**disToDis.properties样例**。

表 3-8 创建 disToDis.properties 对应的程序包主要参数说明

参数名称	说明	示例
包类型	支持的包类型如下： • JAR：用户jar文件 • PyFile：用户Python文件 • File：用户文件 • ModelFile：用户AI模型文件	File
OBS路径	选择对应程序包的OBS路径。 **说明** • 程序包需提前上传至OBS服务中保存。 • 只支持选择文件。	disToDis.properties所在的OBS路径。
分组名称	• 选择“已有分组”：可选择已有的分组。 • 选择“创建新分组”：可输入自定义的组名称。 • 选择“不分组”：不需要选择或输入组名称。	自定义分组或选择已有的分组名称。

图 3-28 创建 disToDis.properties 对应的程序包

4. 创建Flink Jar作业并运行。

   详情参考**创建Flink Jar作业**。在应用程序中选择步骤**2**中创建的Flink Jar文件，在其他依赖文件中选择步骤**3**中创建的properties文件，并指定主类。

表 3-9 创建 Flink Jar 作业参数说明

参数	说明	示例
所属队列	**说明** • Flink Jar作业只能运行在预先创建的独享队列上。 • 如果“所属队列”下拉框中无可用的独享队列，请先创建一个独享队列并将该队列绑定到当前用户	选择Flink Jar作业运行的队列
应用程序	用户自定义的程序包	自定义的程序包
主类	指定加载的Jar包类名，如FlinkDisToDisExample。 • 默认：根据Jar包文件的Manifest文件指定。 • 指定：必须输入“类名”并确定类参数列表（参数间用空格分隔） **说明** 当类属于某个包时，主类路径需要包含完整包路径，例如：packagePath.KafkaMessageStreaming	指定
其他依赖文件	用户自定义的依赖文件。其他依赖文件需要自行在代码中引用。 在选择依赖文件之前需要将对应的文件上传至OBS桶中，并在“数据管理>程序包管理”中创建程序包，包类型没有限制。具体操作请参考**创建程序包**。 通过在应用程序中添加以下内容可访问对应的依赖文件。其中，“fileName”为需要访问的文件名，“ClassName”为需要访问该文件的类名。 `ClassName.class.getClassLoader().getResource("userData/fileName")`	选择3的properties文件。
Flink版本	选择Flink版本前，需要先选择所属的队列。当前支持“1.10”和“1.11”和“1.12”。	1.12

图 3-29 创建 Flink Jar 作业

5. 结果校验。

作业处于运行中状态时，向DIS的source通道发送数据，验证DIS的sink通道能否收到数据。发送和接受都有字节数证明接收到数据。

图 3-30 查看校验结果

JAVA 样例代码

- DIS Flink Connector相关依赖

```
<dependency>
        <groupId>com.huaweicloud.dis</groupId>
        <artifactId>huaweicloud-dis-flink-connector_2.11</artifactId>
        <version>2.0.1</version>
        <exclusions>
            <exclusion>
                <groupId>org.apache.flink</groupId>
                <artifactId>*</artifactId>
            </exclusion>
            <exclusion>
                <groupId>org.apache.logging.log4j</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
```

- pom文件配置

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Flink-dis-12</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!--Flink 版本-->
        <flink.version>1.12.2</flink.version>
        <!--JDK 版本-->
        <java.version>1.8</java.version>
        <!--Scala 2.11 版本-->
        <scala.binary.version>2.11</scala.binary.version>
        <slf4j.version>2.13.3</slf4j.version>
        <log4j.version>2.10.0</log4j.version>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- dis -->
        <dependency>
            <groupId>com.huaweicloud.dis</groupId>
            <artifactId>huaweicloud-dis-flink-connector_2.11</artifactId>
            <version>2.0.1</version>
            <exclusions>
                <exclusion>
                    <groupId>org.apache.flink</groupId>
                    <artifactId>*</artifactId>
                </exclusion>
                <exclusion>
                    <groupId>org.apache.logging.log4j</groupId>
                    <artifactId>*</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <!-- flink -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>${flink.version}</version>
            <scope>provided</scope>
        </dependency>
```

```
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
        <version>${flink.version}</version>
        <scope>provided</scope>
    </dependency>

    <!-- logging -->
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>${slf4j.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-jcl</artifactId>
        <version>${log4j.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>3.3.0</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.flink.FlinkDisToDisExample</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </plugin>
    </plugins>
```

```
        <resources>
            <resource>
                <directory>../main/config</directory>
                <filtering>true</filtering>
                <includes>
                    <include>**/*.*</include>
                </includes>
            </resource>
        </resources>
    </build>
</project>
```

- FlinkDisToDisExample.java样例（可参考**自定义Flink Streaming作业**）

```
package com.flink;
import com.huaweicloud.dis.DISConfig;
import com.huaweicloud.dis.adapter.common.consumer.DisConsumerConfig;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.RocksDBStateBackend;
import org.apache.flink.runtime.state.filesystem.FsStateBackend;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.dis.FlinkDisConsumer;
import org.apache.flink.streaming.connectors.dis.FlinkDisProducer;
import org.apache.flink.util.TernaryBoolean;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Collections;

/**
 * Read data from dis and then write them into another dis channel.
 */
public class FlinkDisToDisExample {
    private static final Logger LOG = LoggerFactory.getLogger(FlinkDisToDisExample.class);

    public static void main(String[] args) throws IOException {
        LOG.info("Read data from dis and write them into dis.");
        String propertiesPath = FlinkDisToDisExample.class.getClassLoader()
            .getResource("userData/disToDis.properties").getPath();
        ParameterTool params = ParameterTool.fromPropertiesFile(propertiesPath);

        // DIS终端节点，如 https://dis.cn-north-1.myhuaweicloud.com
        String endpoint = params.get("disEndpoint");
        // DIS服务所在区域ID，如 cn-north-1
        String region = params.get("region");
        // 用户的AK
        String ak = params.get("ak");
        // 用户的SK
        String sk = params.get("sk");
        // 用户的项目ID
        String projectId = params.get("projectId");
        // 作为source的DIS通道名称
        String sourceChannel = params.get("sourceChannel");
        // 作为sink的DIS通道名称
        String sinkChannel = params.get("sinkChannel");
        // 消费策略，只有当分区没有Checkpoint或者Checkpoint过期时，才会使用此配置的策略；如果存在有效的Checkpoint，则会从此Checkpoint开始继续消费
        // 取值有： LATEST     从最新的数据开始消费，此策略会忽略通道中已有数据
        //         EARLIEST    从最初的数据开始消费，此策略会获取通道中所有的有效数据
        String startingOffsets = params.get("startOffset");
        // 消费组标识，同一个消费组下的不同客户端可以同时消费同一个通道
        String groupId = params.get("groupId");
        // Checkpoint输出路径，格式obs://bucket/path/
        String checkpointBucket = params.get("checkpointPath");
```



```
        // DIS Config
        DISConfig disConfig = DISConfig.buildDefaultConfig();
        disConfig.put(DISConfig.PROPERTY_ENDPOINT, endpoint);
        disConfig.put(DISConfig.PROPERTY_REGION_ID, region);
        disConfig.put(DISConfig.PROPERTY_AK, ak);
        disConfig.put(DISConfig.PROPERTY_SK, sk);
        disConfig.put(DISConfig.PROPERTY_PROJECT_ID, projectId);
        disConfig.put(DisConsumerConfig.AUTO_OFFSET_RESET_CONFIG, startingOffsets);
        disConfig.put(DisConsumerConfig.GROUP_ID_CONFIG, groupId);
        // 是否主动更新分片信息及更新时间间隔（毫秒），若有主动扩缩容需求，可以开启
        disConfig.put(FlinkDisConsumer.KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS, "10000");

        SimpleDateFormat dateTimeFormat = new SimpleDateFormat("yyyy-MM-dd_HH-mm-ss");
        String time = dateTimeFormat.format(System.currentTimeMillis());
        String checkpointPath = checkpointBucket + time;

        try {
            StreamExecutionEnvironment env =
                StreamExecutionEnvironment.getExecutionEnvironment();
            // 设置checkpoint
            RocksDBStateBackend rocksDBStateBackend = new RocksDBStateBackend(new
FsStateBackend(checkpointPath),
                TernaryBoolean.TRUE);
            env.setStateBackend(rocksDBStateBackend);
            // 开启Flink CheckPoint配置，开启时若触发CheckPoint，会将Offset信息同步到Kafka
            env.enableCheckpointing(180000);
            // 设置两次checkpoint的最小间隔时间
            env.getCheckpointConfig().setMinPauseBetweenCheckpoints(60000);
            // 设置checkpoint超时时间
            env.getCheckpointConfig().setCheckpointTimeout(60000);
            // 设置checkpoint最大并发数
            env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
            // 设置作业取消时保留checkpoint
            env.getCheckpointConfig().enableExternalizedCheckpoints(
                CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

            FlinkDisConsumer<String> consumer =
                new FlinkDisConsumer<>(
                    Collections.singletonList(sourceChannel), new SimpleStringSchema(), disConfig);
            DataStream<String> sourceStream = env.addSource(consumer, "disSource");

            FlinkDisProducer<String> producer = new FlinkDisProducer<>(
                sinkChannel, new SimpleStringSchema(), disConfig);
            sourceStream.addSink(producer).disableChaining().name("dis-to-dis");

            env.execute();
        } catch (Exception ex) {
            LOG.error(ex.getMessage(), ex);
        }
    }
}
```

- disToDis.properties样例（注意：groupId的值为步骤1中创建的App名称）

  说明

  认证用的ak和sk硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
# dis所在局点的endpoint，如 https://dis.cn-north-1.myhuaweicloud.com
disEndpoint=xx
# DIS服务所在区域ID，如 cn-north-1
region=xx
# 用户的AK
ak=xx
# 用户的SK
sk=xx
# id示例：6m3nhAGTLxmNfZ4HOit
projectId=xx
# 作为source的dis通道，如：OpenSource_outputmTtkR
```

```
sourceChannel=xx
# 作为sink的dis通道，如：OpenSource_disQFXD
sinkChannel=xx
# dis消费组，需要提前在dis的Apps中创建
groupId=xx
# 消费模式，从最开始消费或者EARLIEST从最新开始消费LATEST
startOffset=LATEST
# flink保存checkpoint的路径
checkpointPath=obs://bucket/path/
```

常见问题

- Q：作业运行失败，运行日志中有如下报错信息，应该怎么解决?

  ```
  java.lang.NoSuchMethodError: org.apache.flink.api.java.ClosureCleaner.clean(Ljava/lang/Object;Z)V
  ```

  A：该问题是因为所选择的huaweicloud-dis-flink-connector_2.11版本过低导致，请选择2.0.1及以上版本。

- Q：运行作业读取DIS数据时，无法读出数据且Taskmanager的运行日志中有如下报错信息，应该怎么解决?

  ```
  ERROR com.huaweicloud.dis.adapter.common.consumer.Coordinator     [] - Failed to getCheckpointAsync, error : [400 : {"errorCode":"DIS.4332","message":"app not found. "}], request : [{"stream_name":"xx","partition_id":"shardId-0000000000","checkpoint_type":"LAST_READ","app_name":"xx"}]
  ```

  A: 该问题是因为读取DIS数据所使用的group.id在DIS的Apps中并没有提前创建。

3.7 Flink 作业委托场景开发指导

3.7.1 Flink Opensource SQL 使用 DEW 管理访问凭据

操作场景

DLI将Flink作业的输出数据写入到Mysql或DWS时，需要在connector中设置账号、密码等属性。但是账号密码等信息属于高度敏感数据，需要做加密处理，以保障用户的数据隐私安全。

数据加密服务（Data Encryption Workshop，DEW）、云凭据管理服务（Cloud Secret Management Service，CSMS），提供一种安全、可靠、简单易用隐私数据加解密方案。

用户或应用程序通过凭据管理服务，创建、检索、更新、删除凭据，轻松实现对敏感凭据的全生命周期的统一管理，有效避免程序硬编码或明文配置等问题导致的敏感信息泄露以及权限失控带来的业务风险。

本节操作介绍Flink Opensource SQL场景使用DEW管理和访问凭据的操作指导。

前提条件

- 已在DEW服务创建通用凭证，并存入凭据值。具体操作请参考：**创建通用凭据**。
- 已创建DLI访问DEW的委托并完成委托授权。该委托需具备以下权限：
  - DEW中的查询凭据的版本与凭据值ShowSecretVersion接口权限，csms:secretVersion:get。
  - DEW中的查询凭据的版本列表ListSecretVersions接口权限，csms:secretVersion:list。
  - DEW解密凭据的权限，kms:dek:decrypt。

委托权限示例请参考**自定义DLI委托权限**和**常见场景的委托权限策略**。

- 仅支持Flink1.15版本使用DEW管理访问凭据，在创建作业时，请配置作业使用Flink1.15版本、且已在作业中配置允许DLI访问DEW的委托信息。
- 在DLI管理控制台新建“增强型跨源连接”配置DLI与数据源的网络连通。

  具体操作请参考**增强型跨源连接**。

语法格式

```
create table tableName(
  attr_name attr_type
  (',' attr_name attr_type)*
  (',' WATERMARK FOR rowtime_column_name AS watermark-strategy_expression)
)
with (
   ...
  'dew.endpoint'='',
  'dew.csms.secretName'='',
  'dew.csms.decrypt.fields'='',
  'dew.projectId'='',
  'dew.csms.version'=''

);
```

参数说明

表 3-10 参数说明

参数	是否必选	默认值	数据类型	参数说明
dew.endpoint	是	无	String	指定要使用的DEW服务所在的endpoint信息。 获取**地区和终端节点**。 配置示例：'dew.endpoint'='kms.cn-xxxx.myhuaweicloud.com'
dew.projectId	否	有	String	DEW所在的项目ID， 默认是Flink作业所在的项目ID。 **获取项目ID**
dew.csms.secretName	是	无	String	在DEW服务的凭据管理中新建的通用凭据的名称。 配置示例：'dew.csms.secretName'='*secretInfo*'
dew.csms.decrypt.fields	是	无	String	指定connector with属性中，哪些字段属性需要使用DEW云凭据管理服务进行解密。字段属性之间用逗号分隔，例如：'dew.csms.decrypt.fields'='field1,field2,field3'

参数	是否必选	默认值	数据类型	参数说明
dew.csms.version	否	最新的version	String	在DEW服务的凭据管理中新建的通用凭据的版本号(凭据的版本标识符)。 若不指定，则默认获取该通用凭证的最新版本号。 配置示例： 'dew.csms.version'='v1'

示例

本例以通过DataGen表产生随机数据并输出到Mysql结果表中为例，介绍Flink Opensource SQL使用DEW管理访问凭据的配置方法。

1. 创建DLI与Mysql的增强型跨源连接。详细步骤请参考**创建增强型跨源连接**。
2. 创建DLI访问DEW的委托并完成委托授权。详细步骤请参考**自定义DLI委托权限**。
3. 在DEW创建通用凭证。详细操作请参考**创建通用凭据**。
   a. 登录DEW管理控制台
   b. 选择“凭据管理”，进入“凭据管理”页面。
   c. 单击“创建凭据”。配置凭据基本信息
   d. 本例配置Mysql的凭据值：
      - "MySQLUsername":"demo"
      - "MySQLPassword":"*******",其中"*******"为Mysql的访问密码
4. Flink作业编辑框中输入具体的作业SQL，本示例作业参考如下。

```
create table dataGenSource(
  user_id string,
  amount int
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1', --每秒生成一条数据
  'fields.user_id.kind' = 'random', --为字段user_id指定random生成器
  'fields.user_id.length' = '3' --限制user_id长度为3
);

CREATE TABLE jdbcSink (
  user_id string,
  amount int
 )
WITH (
 'connector' = 'jdbc',
 'url' = 'jdbc:mysql://MySQLAddress:MySQLPort/flink',--其中url中的flink表示MySQL中orders表所在的数据库名
 'table-name' = 'orders',
 'username' = 'MySQLUsername', -- DEW服务中，名称为secretInfo，且版本号v1的的通用凭证，定义凭证值的键MySQLUsername，它的值为用户的敏感信息。
 'password' = 'MySQLPassword', -- DEW服务中，名称为secretInfo，且版本号v1的的通用凭证，定义凭证值的键MySQLPassword，它的值为用户的敏感信息。
 'sink.buffer-flush.max-rows' = '1',
 'dew.endpoint'='kms.cn-xxxx.myhuaweicloud.com', --使用的DEW服务所在的endpoint信息
 'dew.csms.secretName'='secretInfo', --DEW服务通用凭据的凭据名称
 'dew.csms.decrypt.fields'='username,password', --其中username,password字段值，需要利用DEW凭证管理,进行解密替换。
 'dew.csms.version'='v1'
);
```



```
insert into jdbcSink select * from dataGenSOurce;
```

3.7.2 Flink Jar 使用 DEW 获取访问凭证读写 OBS

操作场景

DLI将Flink Jar作业的输出数据写入到OBS时，需要配置AKSK访问OBS，为了确保AKSK数据安全，您可以通过数据加密服务（Data Encryption Workshop，DEW）、云凭据管理服务（Cloud Secret Management Service，CSMS），对AKSK统一管理，有效避免程序硬编码或明文配置等问题导致的敏感信息泄露以及权限失控带来的业务风险。

本例以获取访问OBS的AKSK为例介绍Flink Jar使用DEW获取访问凭证读写OBS的操作指导。

前提条件

- 已在DEW服务创建通用凭证，并存入凭据值。具体操作请参考：**创建通用凭据**。
- 已创建DLI访问DEW的委托并完成委托授权。该委托需具备以下权限：
  - DEW中的查询凭据的版本与凭据值ShowSecretVersion接口权限，csms:secretVersion:get。
  - DEW中的查询凭据的版本列表ListSecretVersions接口权限，csms:secretVersion:list。
  - DEW解密凭据的权限，kms:dek:decrypt。

  委托权限示例请参考**自定义DLI委托权限**和**常见场景的委托权限策略**。
- 仅支持Flink1.15版本使用DEW管理访问凭据，在创建作业时，请配置作业使用Flink1.15版本、且已在作业中配置允许DLI访问DEW的委托信息。自定义委托及配置请参考**自定义DLI委托权限**。
- 使用该功能，所有涉及OBS的桶，都需要进行配置AKSK。

语法格式

在Flink jar作业编辑界面，选择配置优化参数，配置信息如下:

不同的OBS桶，使用不同的AKSK认证信息。 可以使用如下配置方式，根据桶指定不同的AKSK信息，参数说明详见**表3-11**。

```
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key=USER_AK_CSMS_KEY
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key=USER_SK_CSMS_KEY
flink.hadoop.fs.obs.security.provider=com.dli.provider.UserObsBasicCredentialProvider
flink.hadoop.fs.dew.csms.secretName=CredentialName
flink.hadoop.fs.dew.endpoint=ENDPOINT
flink.hadoop.fs.dew.csms.version=VERSION_ID
flink.hadoop.fs.dew.csms.cache.time.second=CACHE_TIME
flink.dli.job.agency.name=USER_AGENCY_NAME
```

参数说明

表 3-11 参数说明

参数	是否必选	默认值	数据类型	参数说明
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key	是	无	String	USER_BUCKET_NAME为用户的桶名，需要进行替换为用户的使用的OBS桶名。 参数的值为用户定义在CSMS通用凭证中的键key， 其Key对应的value为用户的AK（Access Key Id），需要具备访问OBS对应桶的权限。
flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key	是	无	String	USER_BUCKET_NAME为用户的桶名，需要进行替换为用户的使用的OBS桶名。 参数的值为用户定义在CSMS通用凭证中的键key， 其Key对应的value为用户的SK（Secret Access Key），需要具备访问OBS对应桶的权限。
flink.hadoop.fs.obs.security.provider	是	无	String	OBS AKSK认证机制，使用DEW服务中的CSMS凭证管理，获取OBS的AK、SK。 默认取值为com.dli.provider.UserObsBasicCredentialProvider
flink.hadoop.fs.dew.endpoint	是	无	String	指定要使用的DEW服务所在的endpoint信息。 获取**地区和终端节点**。 配置示例： flink.hadoop.fs.dew.endpoint=kms.cn-xxxx.myhuaweicloud.com
flink.hadoop.fs.dew.projectId	否	有	String	DEW所在的项目ID， 默认是Flink作业所在的项目ID。 **获取项目ID**
flink.hadoop.fs.dew.csms.secretName	是	无	String	在DEW服务的凭据管理中新建的通用凭据的名称。 配置示例： flink.hadoop.fs.dew.csms.secretName=*secretInfo*
flink.hadoop.fs.dew.csms.version	否	最新的version	String	在DEW服务的凭据管理中新建的通用凭据的版本号(凭据的版本标识符)。 若不指定，则默认获取该通用凭证的最新版本号。 配置示例： flink.hadoop.fs.dew.csms.version=v1

参数	是否必选	默认值	数据类型	参数说明
flink.hadoop.fs.dew.csms.cache.time.second	否	3600	Long	Flink作业访问获取CSMS通用凭证后，缓存的时间。 单位为秒。默认值为3600秒。
flink.dli.job.agency.name	是	-	String	自定义委托名称。

样例代码

本章节JAVA样例代码演示将DataGen数据处理后写入到OBS，具体参数配置请根据实际环境修改。

1. 创建DLI访问DEW的委托并完成委托授权。详细步骤请参考**自定义DLI委托权限**。
2. 在DEW创建通用凭证。详细操作请参考**创建通用凭据**。
   a. 登录DEW管理控制台
   b. 选择“凭据管理”，进入“凭据管理”页面。
   c. 单击“创建凭据”。配置凭据基本信息
3. DLI Flink jar作业编辑界面设置作业参数。
   - 类名
     ```
     com.dli.demo.dew.DataGen2FileSystemSink
     ```
   - 参数
     ```
     --checkpoint.path obs://test/flink/jobs/checkpoint/120891/
     --output.path obs://dli/flink.db/79914/DataGen2FileSystemSink
     ```
   - 优化参数：
     ```
     flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key=USER_AK_CSMS_KEY
     flink.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key=USER_SK_CSMS_KEY
     flink.hadoop.fs.obs.security.provider=com.dli.provider.UserObsBasicCredentialProvider
     flink.hadoop.fs.dew.csms.secretName=obsAksK
     flink.hadoop.fs.dew.endpoint=kmsendpoint
     flink.hadoop.fs.dew.csms.version=v6
     flink.hadoop.fs.dew.csms.cache.time.second=3600
     flink.dli.job.agency.name=***
     ```
4. Flink Jar作业示例。
   - **环境准备**

     已安装和配置IntelliJ IDEA等开发工具以及安装JDK和Maven。

     pom文件配置中依赖包

     ```
     <properties>
         <flink.version>1.15.0</flink.version>
     </properties>

     <dependencies>
         <dependency>
             <groupId>org.apache.flink</groupId>
             <artifactId>flink-statebackend-rocksdb</artifactId>
             <version>${flink.version}</version>
             <scope>provided</scope>
         </dependency>
     ```



```
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-java</artifactId>
      <version>${flink.version}</version>
      <scope>provided</scope>
    </dependency>

    <!-- fastjson -->
    <dependency>
      <artifactId>fastjson</artifactId>
      <version>2.0.15</version>
    </dependency>
  </dependencies>
```

- **示例代码**

```
package com.huawei.dli.demo.dew;

import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.java.utils.ParameterTool;
import org.apache.flink.contrib.streaming.state.EmbeddedRocksDBStateBackend;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.OnCheckpointRollingPolicy;
import org.apache.flink.streaming.api.functions.source.ParallelSourceFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.time.LocalDateTime;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.Random;

public class DataGen2FileSystemSink {
    private static final Logger LOG = LoggerFactory.getLogger(DataGen2FileSystemSink.class);

    public static void main(String[] args) {
        ParameterTool params = ParameterTool.fromArgs(args);
        LOG.info("Params: " + params.toString());
        try {
            StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();

            // set checkpoint
            String checkpointPath = params.get("checkpoint.path", "obs://bucket/checkpoint/
jobId_jobName/");
            LocalDateTime localDateTime =
LocalDateTime.ofEpochSecond(System.currentTimeMillis() / 1000,
                0, ZoneOffset.ofHours(8));
            String dt =
localDateTime.format(DateTimeFormatter.ofPattern("yyyyMMdd_HH:mm:ss"));
            checkpointPath = checkpointPath + dt;

            streamEnv.setStateBackend(new EmbeddedRocksDBStateBackend());
            streamEnv.getCheckpointConfig().setCheckpointStorage(checkpointPath);
            streamEnv.getCheckpointConfig().setExternalizedCheckpointCleanup(
                CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
            streamEnv.enableCheckpointing(30 * 1000);

            DataStream<String> stream = streamEnv.addSource(new DataGen())
                .setParallelism(1)
                .disableChaining();

            String outputPath = params.get("output.path", "obs://bucket/outputPath/
jobId_jobName");

            // Sink OBS
```



```
            final StreamingFileSink<String> sinkForRow = StreamingFileSink
                .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
                .withRollingPolicy(OnCheckpointRollingPolicy.build())
                .build();

            stream.addSink(sinkForRow);

            streamEnv.execute("sinkForRow");
        } catch (Exception e) {
            LOG.error(e.getMessage(), e);
        }
    }
}

class DataGen implements ParallelSourceFunction<String> {

    private boolean isRunning = true;

    private Random random = new Random();

    @Override
    public void run(SourceContext<String> ctx) throws Exception {
        while (isRunning) {
            JSONObject jsonObject = new JSONObject();
            jsonObject.put("id", random.nextLong());
            jsonObject.put("name", "Molly" + random.nextInt());
            jsonObject.put("address", "hangzhou" + random.nextInt());
            jsonObject.put("birthday", System.currentTimeMillis());
            jsonObject.put("city", "hangzhou" + random.nextInt());
            jsonObject.put("number", random.nextInt());
            ctx.collect(jsonObject.toJSONString());
            Thread.sleep(1000);
        }
    }

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

3.7.3 获取 Flink 作业委托临时凭证用于访问其他云服务

功能描述

DLI提供了一个通用接口，可用于获取用户在启动Flink作业时设置的委托的临时凭证。该接口将获取到的该作业委托的临时凭证封装到com.huaweicloud.sdk.core.auth.BasicCredentials类中。

- 获取到的委托的临时认证封装到com.huaweicloud.sdk.core.auth.ICredentialProvider接口的getCredentials()返回值中。
- 返回类型为com.huaweicloud.sdk.core.auth.BasicCredentials。
- 仅支持获取AK、SK、SecurityToken。
- 获取到AK、SK、SecurityToken后，请参考**如何使用凭据管理服务替换硬编码的数据库账号密码**查询凭据。

约束限制

- 仅支持Flink1.15版本使用委托授权访问临时凭证：
  - 在创建作业时，请配置作业使用Flink1.15版本

  - 已在作业中配置允许DLI访问DEW的委托信息。flink.dli.job.agency.name=*自定义委托名称*。

    自定义委托请参考**自定义DLI委托权限**。

    请注意配置参数不需要用"" 或 '' 包裹。
- Flink1.15基础镜像内置了3.1.62版本的huaweicloud-sdk-core。

准备环境

已安装和配置IntelliJ IDEA等开发工具以及安装JDK和Maven。

Maven工程的pom.xml文件配置请参考**JAVA样例代码**中“pom文件配置”说明。

pom文件配置中依赖包

```
<dependency>
   <groupId>com.huaweicloud.sdk</groupId>
   <artifactId>huaweicloud-sdk-core</artifactId>
   <version>3.1.62</version>
   <scope>provided</scope>
</dependency>
```

示例代码

本章节JAVA样例代码演示如何获取BasicCredentials，以及取临时委托的AK、SK、SecurityToken。

- **Flink UDF 获取作业委托凭证**

```
package com.huawei.dli.demo;

import static com.huawei.dli.demo.utils.DLIJobAgencyCredentialUtils.getICredentialProvider;

import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.core.auth.ICredentialProvider;

import org.apache.flink.table.functions.FunctionContext;
import org.apache.flink.table.functions.ScalarFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class GetUserAgencyCredentialUDF extends ScalarFunction {
   private static final Logger LOG = LoggerFactory.getLogger(GetUserAgencyCredentialUDF.class);

   ICredentialProvider credentialProvider;

   @Override
   public void open(FunctionContext context) throws Exception {
      credentialProvider = getICredentialProvider();
   }

   public String eval(String value) {
      BasicCredentials basicCredentials = (BasicCredentials) credentialProvider.getCredentials();

      String ak = basicCredentials.getAk();
      String sk = basicCredentials.getSk();
      String securityToken = basicCredentials.getSecurityToken();
      LOG.info(">>> ak " + ak + " sk " + sk.length() + " token " + securityToken.length());
      return value + "_demo";
   }
}
```

- **Flink Jar作业获取作业委托凭证**

```
package com.huawei.dli.demo;
```



```
import static com.huawei.dli.demo.utils.DLIJobAgencyCredentialUtils.getICredentialProvider;

import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.core.auth.ICredentialProvider;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class GetUserCredentialsFlinkStream {
    private static final Logger LOG = LoggerFactory.getLogger(GetUserCredentialsFlinkStream.class);

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<String> stream = streamEnv.addSource(new DataGen()).disableChaining();
        ICredentialProvider credentialProvider = getICredentialProvider();

        BasicCredentials basicCredentials = (BasicCredentials) credentialProvider.getCredentials();
        String ak = basicCredentials.getAk();
        String sk = basicCredentials.getSk();
        String securityToken = basicCredentials.getSecurityToken();
        LOG.info(">>" + " ak " + ak + " sk " + sk.length() + " token " + securityToken.length());

        stream.print();
        streamEnv.execute("GetUserCredentialsFlinkStream");
    }
}
```

- **获取作业委托的工具类**

```
package com.huawei.dli.demo.utils;

import com.huaweicloud.sdk.core.auth.ICredentialProvider;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

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

public class DLIJobAgencyCredentialUtils {

    public static ICredentialProvider getICredentialProvider() {
        List<ICredentialProvider> credentialProviders = new ArrayList<>();
        ServiceLoader.load(ICredentialProvider.class, StreamExecutionEnvironment.class.getClassLoader())
            .iterator()
            .forEachRemaining(credentialProviders::add);

        if (credentialProviders.size() != 1) {
            throw new RuntimeException("Failed to obtain temporary user credential");
        }
        return credentialProviders.get(0);
    }
}
```


4 Spark Jar 作业开发指南

4.1 使用 Spark Jar 作业读取和查询 OBS 数据

操作场景

DLI完全兼容开源的Apache Spark，支持用户开发应用程序代码来进行作业数据的导入、查询以及分析处理。本示例从编写Spark程序代码读取和查询OBS数据、编译打包到提交Spark Jar作业等完整的操作步骤说明来帮助您在DLI上进行作业开发。

环境准备

在进行Spark Jar作业开发前，请准备以下开发环境。

表 4-1 Spark Jar 作业开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。

开发流程

DLI进行Spark Jar作业开发流程参考如下：

图 4-1 Spark Jar 作业开发流程

表 4-2 开发流程说明

序号	阶段	操作界面	说明
1	创建DLI通用队列	DLI控制台	创建作业运行的DLI队列。
2	上传数据到OBS桶	OBS控制台	将测试数据上传到OBS桶下。
3	新建Maven工程，配置pom文件	IntelliJ IDEA	参考样例代码说明，编写程序代码读取OBS数据。
4	编写程序代码		
5	调试，编译代码并导出Jar包		
6	上传Jar包到OBS和DLI	OBS控制台 DLI控制台	将生成的Spark Jar包文件上传到OBS目录下和DLI程序包中。
7	创建Spark Jar作业	DLI控制台	在DLI控制台创建Spark Jar作业并提交运行作业。
8	查看作业运行结果	DLI控制台	查看作业运行状态和作业运行日志。

步骤 1：创建 DLI 通用队列

提交Spark作业需要先创建队列，本例创建名为“sparktest”的通用队列。

1. 登录DLI管理控制台。
2. 在左侧导航栏单击“资源管理 > 弹性资源池”，可进入弹性资源池管理页面。
3. 在弹性资源池管理界面，单击界面右上角的“购买弹性资源池”。
4. 在“购买弹性资源池”界面，填写具体的弹性资源池参数。

   本例在华东-上海二区域购买按需计费的弹性资源池。相关参数说明如**表4-3**所示。

   表 4-3 参数说明

参数名称	参数说明	配置样例
计费模式	选择弹性资源池计费模式。	按需计费
区域	选择弹性资源池所在区域。	华东-上海二
项目	每个区域默认对应一个项目，由系统预置。	系统默认项目

参数名称	参数说明	配置样例
名称	弹性资源池名称。	dli_resource_pool
规格	选择弹性资源池规格。	标准版
CU范围	弹性资源池最大最小CU范围。	64-64
网段	规划弹性资源池所属的网段。如需使用DLI增强型跨源，弹性资源池网段与数据源网段不能重合。**弹性资源池网段设置后不支持更改**。	172.16.0.0/19
企业项目	选择对应的企业项目。	default

5. 参数填写完成后，单击“立即购买”，在界面上确认当前配置是否正确。
6. 单击“提交”完成弹性资源池的创建。
7. 在弹性资源池的列表页，选择要操作的弹性资源池，单击操作列的“添加队列”。
8. 配置队列的基础配置，具体参数信息如下。

表 4-4 弹性资源池添加队列基础配置

参数名称	参数说明	配置样例
名称	弹性资源池添加的队列名称。	dli_queue_01
类型	选择创建的队列类型。 ● 执行SQL作业请选择SQL队列。 ● 执行Flink或Spark作业请选择通用队列。	_
执行引擎	SQL队列可以选择队列引擎为Spark或者HetuEngine。	_
企业项目	选择对应的企业项目。	default

9. 单击“下一步”，配置队列的扩缩容策略。

   单击“新增”，可以添加不同优先级、时间段、“最小CU”和“最大CU”扩缩容策略。

   本例配置的扩缩容策略如图4-2所示。

图 4-2 添加队列时配置扩缩容策略

表 4-5 扩缩容策略参数说明

参数名称	参数说明	配置样例
优先级	当前弹性资源池中的优先级数字越大表示优先级越高。本例设置一条扩缩容策略，默认优先级为1。	1
时间段	首条扩缩容策略是默认策略，不能删除和修改时间段配置。 即设置00-24点的扩缩容策略。	00-24
最小CU	设置扩缩容策略支持的最小CU数。	16
最大CU	当前扩缩容策略支持的最大CU数。	64

10. 单击“确定”完成添加队列配置。

步骤 2：上传数据到 OBS 桶

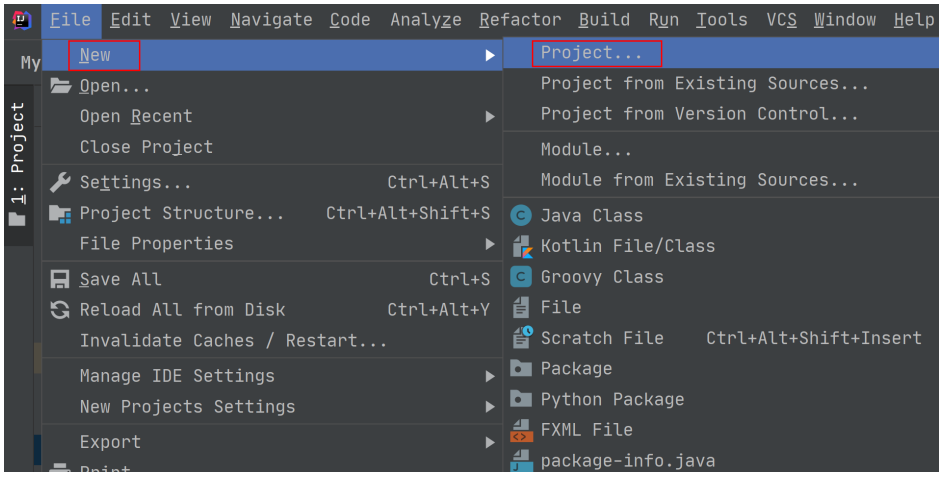
1. 根据如下数据，创建people.json文件。
   ```
   {"name":"Michael"}
   {"name":"Andy", "age":30}
   {"name":"Justin", "age":19}
   ```
2. 进入OBS管理控制台，在“桶列表”下，单击已创建的OBS桶名称，本示例桶名为“dli-test-obs01”。
3. 单击“上传对象”，将people.json文件上传到OBS桶根目录下。
4. 在OBS桶根目录下，单击“新建文件夹”，创建名为“result”的文件夹。
5. 单击“result”的文件夹，在“result”下单击“新建文件夹”，创建名为“parquet”的文件夹。

步骤 3：新建 Maven 工程，配置 pom 依赖

以下通过IntelliJ IDEA 2020.2工具操作演示。

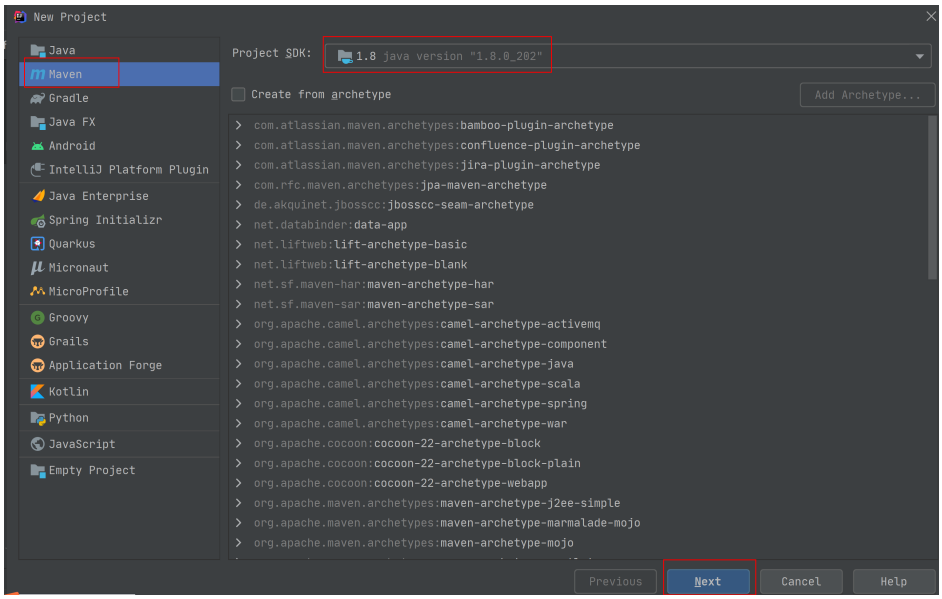
1. 打开IntelliJ IDEA，选择“File > New > Project”。

图 4-3 新建 Project



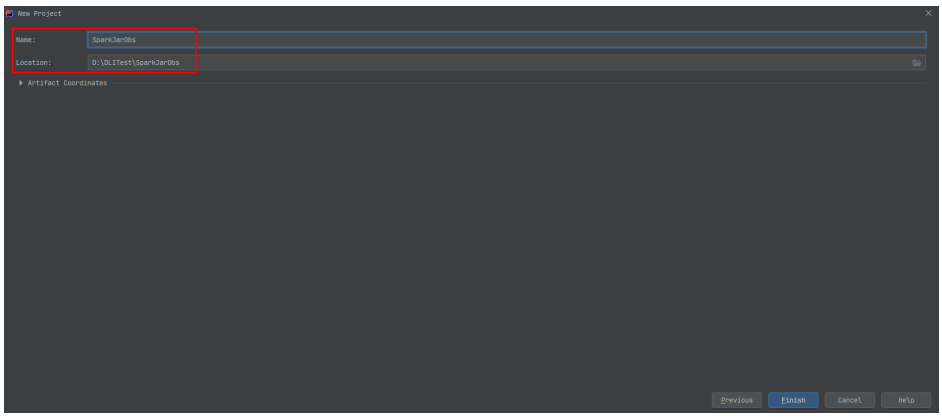
2. 选择Maven，Project SDK选择1.8，单击“Next”。

图 4-4 新建 Project



3. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。

图 4-5 创建工程

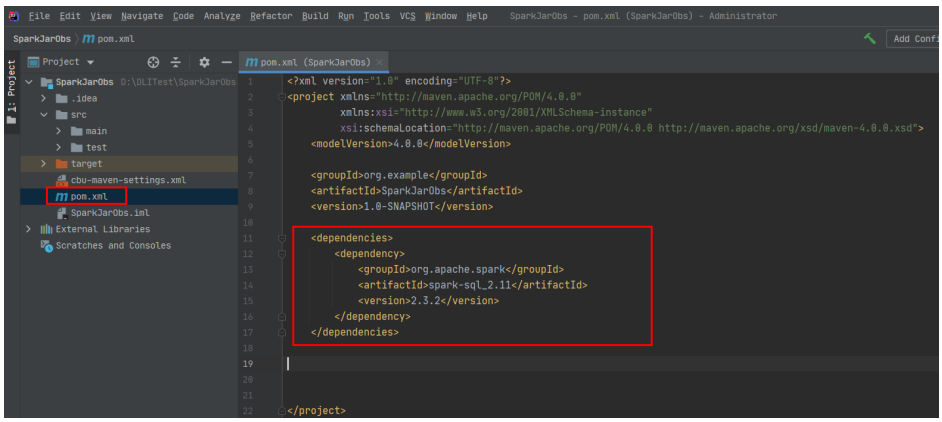


如上图所示，本示例创建Maven工程名为：SparkJarObs，Maven工程路径为：“D:\DLITest\SparkJarObs”。

4. 在pom.xml文件中添加如下配置。

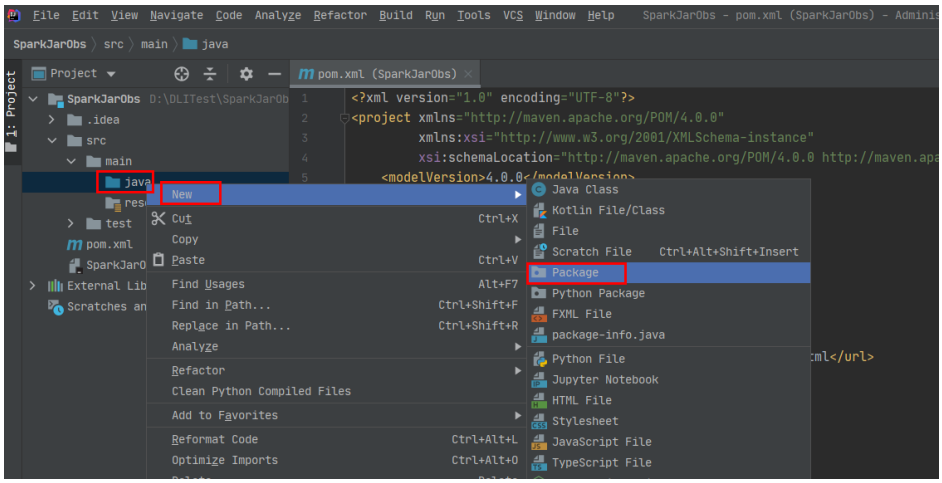
```
<dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
        </dependency>
</dependencies>
```

图 4-6 修改 pom.xml 文件



5. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

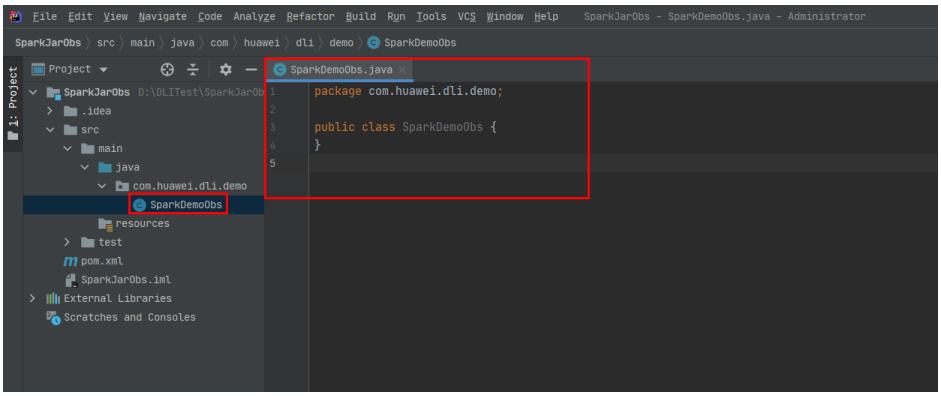
图 4-7 新建 Package



Package根据需要定义，本示例定义为：“com.huawei.dli.demo”，完成后回车。

在包路径下新建Java Class文件，本示例定义为：SparkDemoObs。

图 4-8 新建 Java Class 文件



步骤 4：编写代码

编写SparkDemoObs程序读取OBS桶下的1的“people.json”文件，并创建和查询临时表“people”。

完整的样例请参考**完整样例代码参考**，样例代码分段说明如下：

1. 导入依赖的包。

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.SparkSession;

import static org.apache.spark.sql.functions.col;
```

2. 通过当前账号的AK和SK创建SparkSession会话spark 。

```
SparkSession spark = SparkSession
        .builder()
        .config("spark.hadoop.fs.obs.access.key", "xxx")
        .config("spark.hadoop.fs.obs.secret.key", "yyy")
        .appName("java_spark_demo")
        .getOrCreate();
```

   - "spark.hadoop.fs.obs.access.key"参数对应的值"*xxx*"需要替换为账号的AK值。
   - "spark.hadoop.fs.obs.secret.key"参数对应的值“yyy”需要替换为账号的SK值。

   AK和SK值获取请参考：**如何获取AK和SK**。

3. 读取OBS桶中的“people.json”文件数据。

   其中“dli-test-obs01”为演示的OBS桶名，请根据实际的OBS桶名替换。

```
Dataset<Row> df = spark.read().json("obs://dli-test-obs01/people.json");
df.printSchema();
```

4. 通过创建临时表“people”读取文件数据。

```
df.createOrReplaceTempView("people");
```

5. 查询表“people”数据。

```
Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
```

6. 将表“people”数据以parquet格式输出到OBS桶的“result/parquet”目录下。

```
sqlDF.write().mode(SaveMode.Overwrite).parquet("obs://dli-test-obs01/result/parquet");
spark.read().parquet("obs://dli-test-obs01/result/parquet").show();
```

7. 关闭SparkSession会话spark。

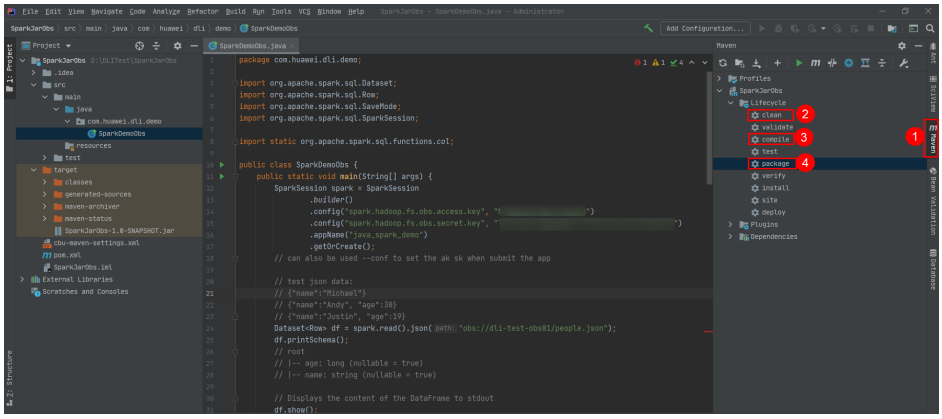
```
spark.stop();
```


步骤 5：调试、编译代码并导出 Jar 包

1. 双击IntelliJ IDEA工具右侧的“Maven”，参考下图分别双击“clean”、“compile”对代码进行编译。

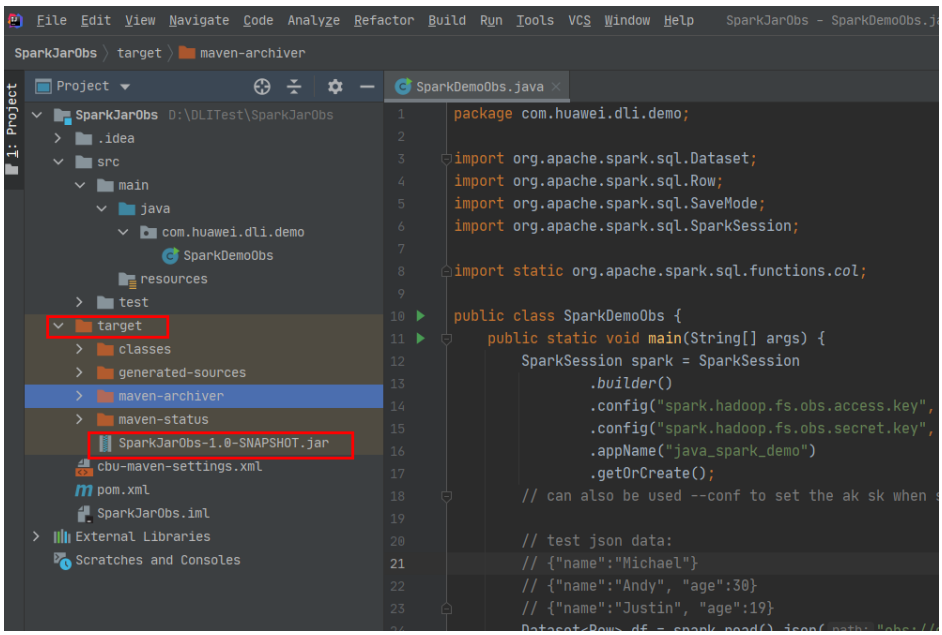
   编译成功后，双击“package”对代码进行打包。

图 4-9 编译打包



打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\DLITest\SparkJarObs\target”下名为“SparkJarObs-1.0-SNAPSHOT.jar”。

图 4-10 导出 jar 包



步骤 6：上传 Jar 包到 OBS 和 DLI 下

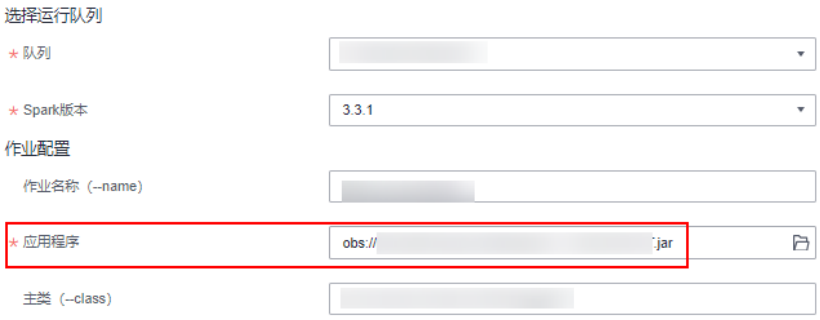
- **Spark 3.3及以上版本：**

  仅支持在创建Spark作业时，配置“应用程序”，从OBS选择作业所需的Jar包。

  a. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

b. 登录DLI控制台，选择“作业管理 > Spark作业”。
c. 单击操作列“编辑”。
d. 编辑“应用程序”，选择a上传的OBS地址。

图 4-11 配置应用程序



- **Spark 3.3以下版本：**
  分别上传Jar包到OBS和DLI下。
  a. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。
  b. 将Jar包文件上传到DLI的程序包管理中，方便后续统一管理。
     i. 登录DLI管理控制台，单击“数据管理 > 程序包管理”。
     ii. 在“程序包管理”页面，单击右上角的“创建程序包”。
     iii. 在“创建程序包”对话框，配置以下参数。
        1) 包类型：选择“JAR”。
        2) OBS路径：程序包所在的OBS路径。
        3) 分组设置和组名称根据情况选择设置，方便后续识别和管理程序包。
     iv. 单击“确定”，完成创建程序包。

图 4-12 创建程序包

步骤 7：创建 Spark Jar 作业

1. 登录DLI控制台，单击“作业管理 > Spark作业”。
2. 在“Spark作业”管理界面，单击“创建作业”。
3. 在作业创建界面，配置对应作业运行参数。具体说明如下：
   - 所属队列：选择已创建的DLI通用队列。例如当前选择**步骤1：创建DLI通用队列**创建的通用队列“sparktest”。
   - 在下拉列表中选择支持的Spark版本，推荐使用最新版本。
   - 作业名称（--name）：自定义Spark Jar作业运行的名称。当前定义为：SparkTestObs。
   - 应用程序：选择**步骤6：上传Jar包到OBS和DLI下**中上传到DLI程序包。例如当前选择为：“SparkJarObs-1.0-SNAPSHOT.jar”。
   - 主类：格式为：程序包名+类名。例如当前为：com.huawei.dli.demo.SparkDemoObs。

   其他参数可暂不选择。

   了解更多Spark Jar作业提交说明可以参考**创建Spark作业**。
4. 单击“执行”，提交该Spark Jar作业。在Spark作业管理界面显示已提交的作业运行状态。

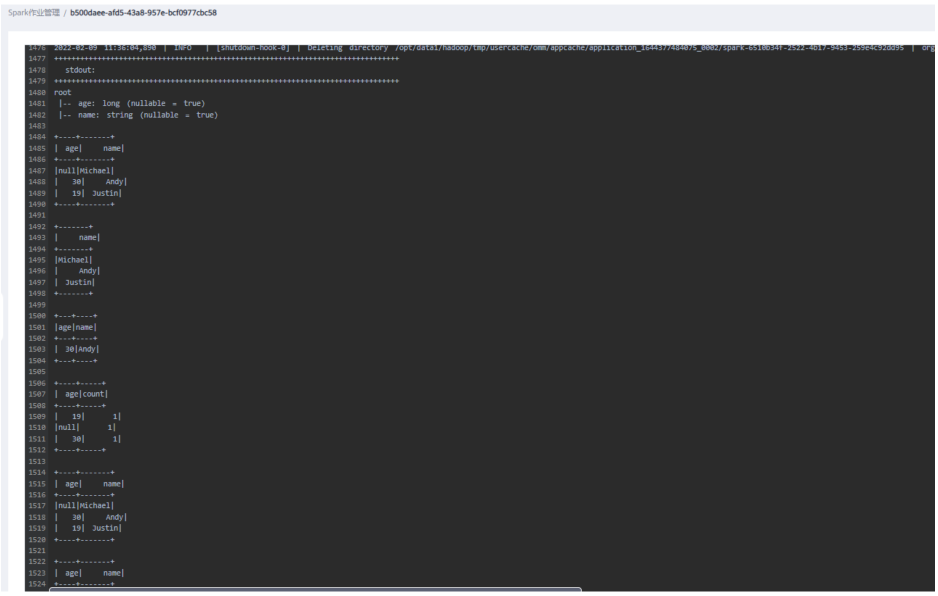
图 4-13 作业运行状态

步骤 8：查看作业运行结果

1. 在Spark作业管理界面显示已提交的作业运行状态。初始状态显示为“启动中”。
2. 如果作业运行成功则作业状态显示为“已成功”，单击“操作”列“更多”下的“Driver日志”，显示当前作业运行的日志。

图 4-14 diver 日志

图 4-15 “Driver 日志”中的作业执行日志



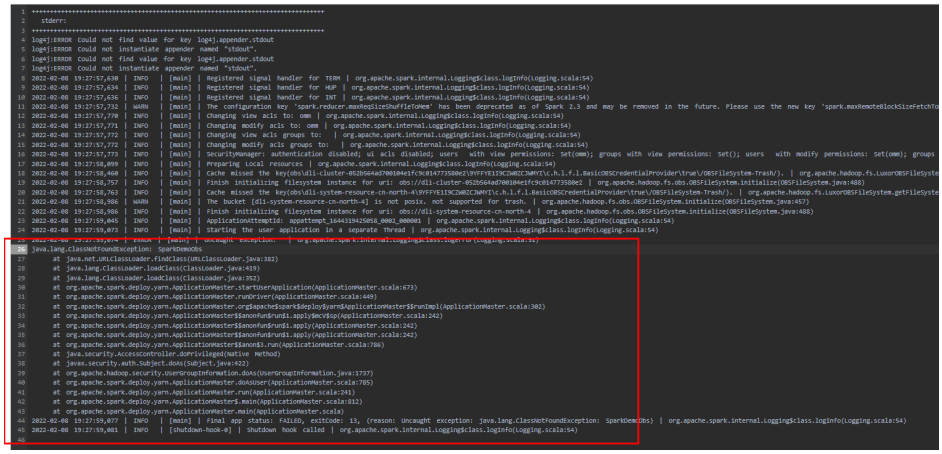
3. 如果作业运行成功，本示例进入OBS桶下的“result/parquet”目录，查看已生成预期的parquet文件。

图 4-16 obs 桶文件

4. 如果作业运行失败，单击“操作”列“更多”下的“Driver日志”，显示具体的报错日志信息，根据报错信息定位问题原因。

   例如，如下截图信息因为创建Spark Jar作业时主类名没有包含包路径，报找不到类名“SparkDemoObs”。

图 4-17 报错信息



可以在“操作”列，单击“编辑”，修改“主类”参数为正确的：com.huawei.dli.demo.SparkDemoObs，单击“执行”重新运行该作业即可。

后续指引

- 如果您想通过Spark Jar作业访问其他数据源，请参考《**使用Spark作业跨源访问数据源**》。
- 如果您想通过Spark Jar作业在DLI创建数据库和表，请参考《**使用Spark作业访问DLI元数据**》。

完整样例代码参考

说明

认证用的access.key和secret.key硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
package com.huawei.dli.demo;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SaveMode;
import org.apache.spark.sql.SparkSession;

import static org.apache.spark.sql.functions.col;

public class SparkDemoObs {
```



```
public static void main(String[] args) {
    SparkSession spark = SparkSession
            .builder()
            .config("spark.hadoop.fs.obs.access.key", "xxx")
            .config("spark.hadoop.fs.obs.secret.key", "yyy")
            .appName("java_spark_demo")
            .getOrCreate();
    // can also be used --conf to set the ak sk when submit the app

    // test json data:
    // {"name":"Michael"}
    // {"name":"Andy", "age":30}
    // {"name":"Justin", "age":19}
    Dataset<Row> df = spark.read().json("obs://dli-test-obs01/people.json");
    df.printSchema();
    // root
    // |-- age: long (nullable = true)
    // |-- name: string (nullable = true)

    // Displays the content of the DataFrame to stdout
    df.show();
    // +----+-------+
    // | age|   name|
    // +----+-------+
    // |null|Michael|
    // |  30|   Andy|
    // |  19| Justin|
    // +----+-------+

    // Select only the "name" column
    df.select("name").show();
    // +-------+
    // |   name|
    // +-------+
    // |Michael|
    // |   Andy|
    // | Justin|
    // +-------+

    // Select people older than 21
    df.filter(col("age").gt(21)).show();
    // +---+----+
    // |age|name|
    // +---+----+
    // | 30|Andy|
    // +---+----+

    // Count people by age
    df.groupBy("age").count().show();
    // +----+-----+
    // | age|count|
    // +----+-----+
    // |  19|    1|
    // |null|    1|
    // |  30|    1|
    // +----+-----+

    // Register the DataFrame as a SQL temporary view
    df.createOrReplaceTempView("people");

    Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
    sqlDF.show();
    // +----+-------+
    // | age|   name|
    // +----+-------+
    // |null|Michael|
    // |  30|   Andy|
    // |  19| Justin|
    // +----+-------+
```

```
        sqlDF.write().mode(SaveMode.Overwrite).parquet("obs://dli-test-obs01/result/parquet");
        spark.read().parquet("obs://dli-test-obs01/result/parquet").show();

        spark.stop();
    }
}
```

4.2 使用 Spark 作业访问 DLI 元数据

操作场景

DLI支持用户编写代码创建Spark作业来创建数据库、创建DLI表或OBS表和插入表数据等操作。本示例完整的演示通过编写java代码、使用Spark作业创建数据库、创建表和插入表数据的详细操作，帮助您在DLI上进行作业开发。

说明

该功能公测阶段，如需使用请提交工单申请开通“使用Spark作业访问DLI元数据”的使用权限。

约束限制

- 如果使用Spark 3.1访问元数据，则必须新建队列。
- **不支持**的场景：
  - 在SQL作业中创建了数据库（database），编写程序代码指定在该数据库下创建表。

    例如在DLI的SQL编辑器中的某SQL队列下，创建了数据库testdb。后续通过编写程序代码在testdb下创建表testTable，编译打包后提交的Spark Jar作业则会运行失败。
- **支持**的场景
  - 在SQL作业中创建数据库（database），表（table），通过SQL或Spark程序作业读取插入数据。
  - 在Spark程序作业中创建数据库（database），表（table），通过SQL或Spark程序作业读取插入数据。

环境准备

在进行Spark 作业访问DLI元数据开发前，请准备以下开发环境。

表 4-6 Spark Jar 作业开发环境

准备项	说明
操作系统	Windows系统，支持Windows7以上版本。
安装JDK	JDK使用1.8版本。
安装和配置IntelliJ IDEA	IntelliJ IDEA为进行应用开发的工具，版本要求使用2019.1或其他兼容版本。

准备项	说明
安装Maven	开发环境的基本配置。用于项目管理，贯穿软件开发生命周期。

开发流程

DLI进行Spark作业访问DLI元数据开发流程参考如下：

图 4-18 Spark 作业访问 DLI 元数据开发流程

表 4-7 开发流程说明

序号	阶段	操作界面	说明
1	创建DLI通用队列	DLI控制台	创建作业运行的DLI队列。
2	OBS桶文件配置	OBS控制台	• 如果是创建OBS表，则需要上传文件数据到OBS桶下。 • 配置Spark创建表的元数据信息的存储路径。该文件夹路径用来存储Spark创建表的元数据信息“spark.sql.warehouse.dir”。
3	新建Maven工程，配置pom文件	IntelliJ IDEA	参考样例代码说明，编写程序代码创建DLI表或OBS表。
4	编写程序代码		
5	调试，编译代码并导出Jar包		
6	上传Jar包到OBS和DLI	OBS控制台 DLI控制台	将生成的Spark Jar包文件上传到OBS目录下和DLI程序包中。
7	创建Spark Jar作业	DLI控制台	在DLI控制台创建Spark Jar作业并提交运行作业。
8	查看作业运行结果	DLI控制台	查看作业运行状态和作业运行日志。

步骤 1：创建 DLI 通用队列

提交Spark作业需要先创建队列，本例创建名为“sparktest”的通用队列。

1. 登录DLI管理控制台。
2. 在左侧导航栏单击“资源管理 > 弹性资源池”，可进入弹性资源池管理页面。
3. 在弹性资源池管理界面，单击界面右上角的“购买弹性资源池”。
4. 在“购买弹性资源池”界面，填写具体的弹性资源池参数。
   本例在华东-上海二区域购买按需计费的弹性资源池。相关参数说明如表4-8所示。

表 4-8 参数说明

参数名称	参数说明	配置样例
计费模式	选择弹性资源池计费模式。	按需计费
区域	选择弹性资源池所在区域。	华东-上海二
项目	每个区域默认对应一个项目，由系统预置。	系统默认项目
名称	弹性资源池名称。	dli_resource_pool
规格	选择弹性资源池规格。	标准版
CU范围	弹性资源池最大最小CU范围。	64-64
网段	规划弹性资源池所属的网段。如需使用DLI增强型跨源，弹性资源池网段与数据源网段不能重合。**弹性资源池网段设置后不支持更改**。	172.16.0.0/19
企业项目	选择对应的企业项目。	default

5. 参数填写完成后，单击“立即购买”，在界面上确认当前配置是否正确。
6. 单击“提交”完成弹性资源池的创建。
7. 在弹性资源池的列表页，选择要操作的弹性资源池，单击操作列的“添加队列”。
8. 配置队列的基础配置，具体参数信息如下。

表 4-9 弹性资源池添加队列基础配置

参数名称	参数说明	配置样例
名称	弹性资源池添加的队列名称。	dli_queue_01
类型	选择创建的队列类型。 • 执行SQL作业请选择SQL队列。 • 执行Flink或Spark作业请选择通用队列。	_
执行引擎	SQL队列可以选择队列引擎为Spark或者HetuEngine。	_
企业项目	选择对应的企业项目。	default

9. 单击“下一步”，配置队列的扩缩容策略。

   单击“新增”，可以添加不同优先级、时间段、“最小CU”和“最大CU”扩缩容策略。

   本例配置的扩缩容策略如图4-19所示。

   图 4-19 添加队列时配置扩缩容策略

   表 4-10 扩缩容策略参数说明

参数名称	参数说明	配置样例
优先级	当前弹性资源池中的优先级数字越大表示优先级越高。本例设置一条扩缩容策略，默认优先级为1。	1
时间段	首条扩缩容策略是默认策略，不能删除和修改时间段配置。 即设置00-24点的扩缩容策略。	00-24
最小CU	设置扩缩容策略支持的最小CU数。	16
最大CU	当前扩缩容策略支持的最大CU数。	64

10. 单击“确定”完成添加队列配置。

步骤 2：OBS 桶文件配置

1. 如果需要创建OBS表，则需要先上传数据到OBS桶目录下。

   本次演示的样例代码创建了OBS表，测试数据内容参考如下示例，创建名为的testdata.csv文件。

   ```
   12,Michael
   27,Andy
   30,Justin
   ```

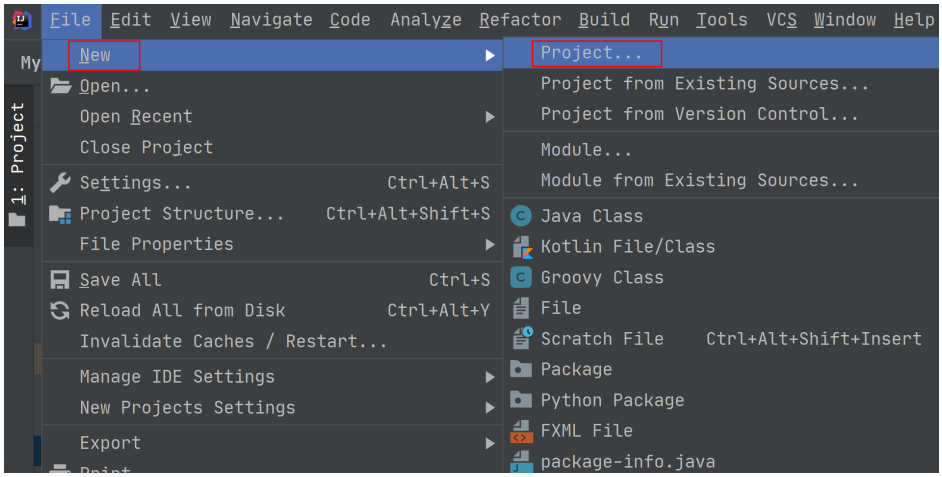
2. 进入OBS管理控制台，在“桶列表”下，单击已创建的OBS桶名称，本示例桶名为“dli-test-obs01”。
3. 单击“上传对象”，将testdata.csv文件上传到OBS桶根目录下。
4. 在OBS桶根目录下，单击“新建文件夹”，创建名为“warehousepath”的文件夹。该文件夹路径用来存储Spark创建表的元数据信息“spark.sql.warehouse.dir”。

步骤 3：新建 Maven 工程，配置 pom 依赖

以下通过IntelliJ IDEA 2020.2工具操作演示。

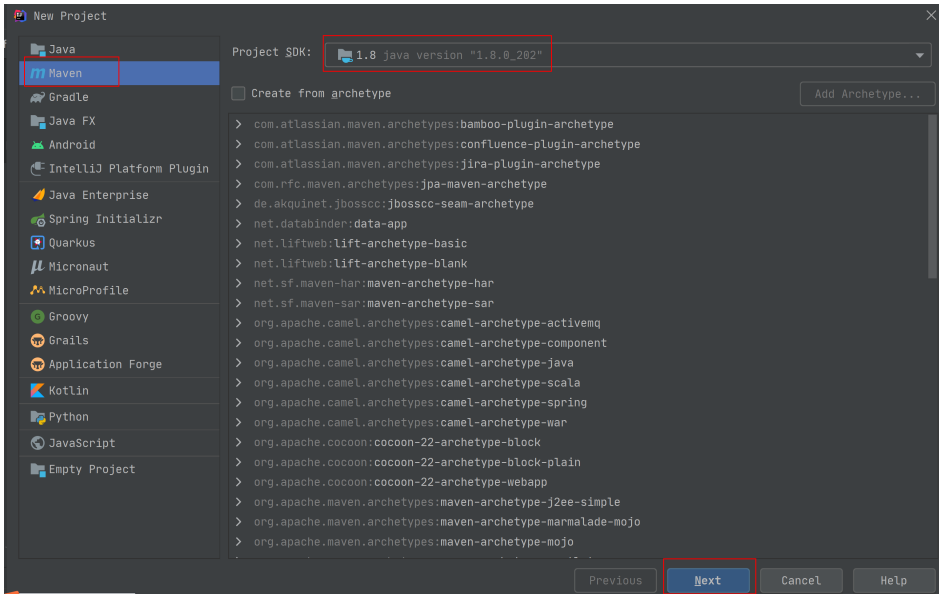
1. 打开IntelliJ IDEA，选择“File > New > Project”。

图 4-20 新建 Project



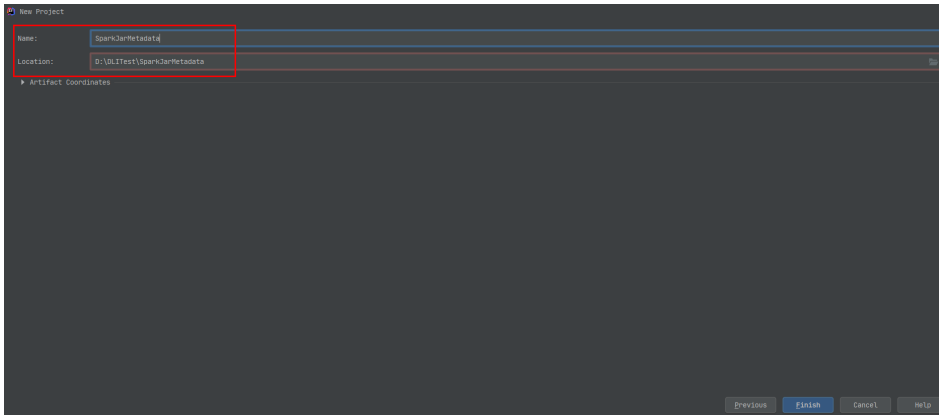
2. 选择Maven，Project SDK选择1.8，单击“Next”。

图 4-21 选择 SDK



3. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。

图 4-22 新建工程

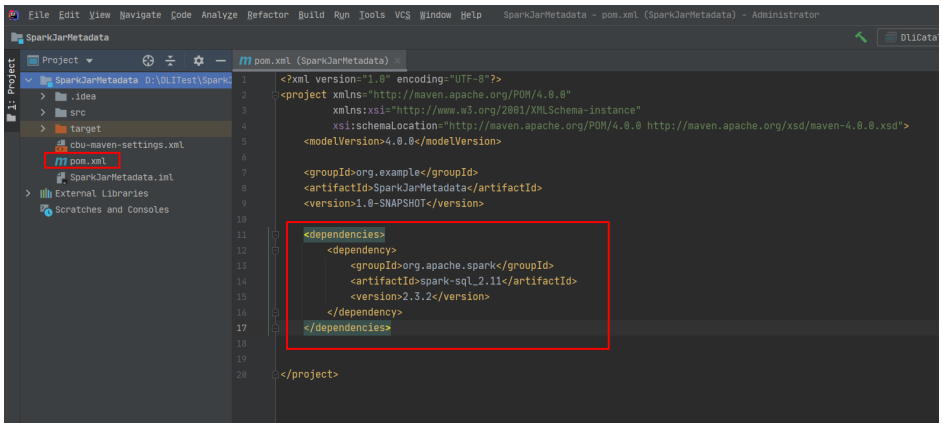


如上图所示，本示例创建Maven工程名为：SparkJarMetadata，Maven工程路径为：“D:\DLITest\SparkJarMetadata”。

4. 在pom.xml文件中添加如下配置。

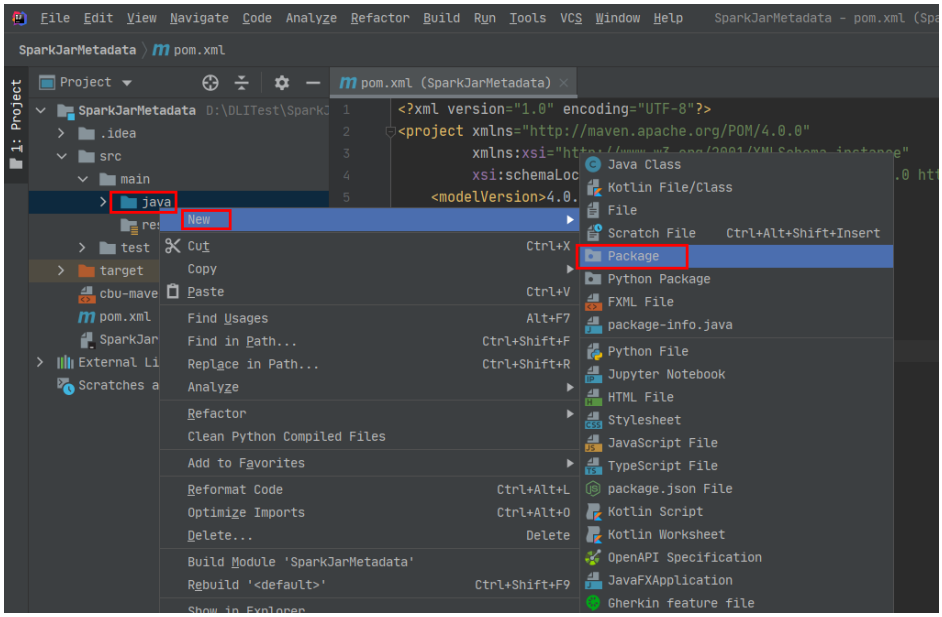
```
<dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
        </dependency>
</dependencies>
```

图 4-23 修改 pom.xml 文件



5. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建Package和类文件。

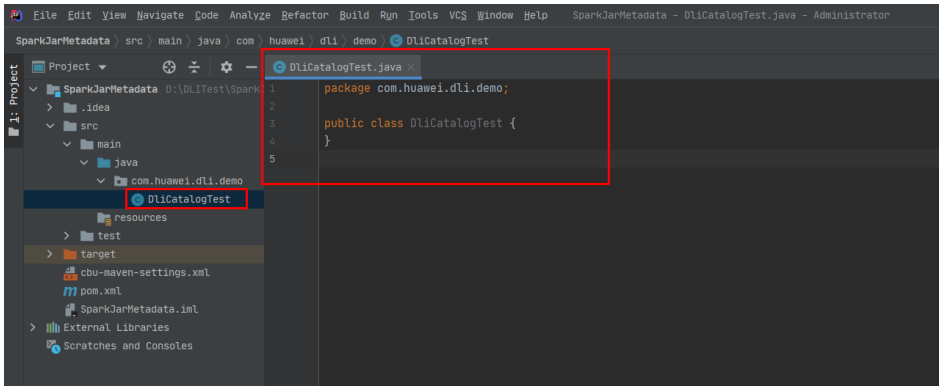
图 4-24 新建 Package



Package根据需要定义，本示例定义为：“com.huawei.dli.demo”，完成后回车。

在包路径下新建Java Class文件，本示例定义为：DliCatalogTest。

图 4-25 新建 Java Class 文件



步骤 4：编写代码

编写DliCatalogTest程序创建数据库、DLI表和OBS表。

完整的样例请参考**Java样例代码**，样例代码分段说明如下：

1. 导入依赖的包。

   ```
   import org.apache.spark.sql.SparkSession;
   ```

2. 创建SparkSession会话。

   创建SparkSession会话时需要指定Spark参数：**"spark.sql.session.state.builder"**、**"spark.sql.catalog.class"**和**"spark.sql.extensions"**，按照样例配置即可。

   - Spark2.3.x版本

```
SparkSession spark = SparkSession
                .builder()
                .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
                .config("spark.sql.catalog.class",
"org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
                .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
                .appName("java_spark_demo")
                .getOrCreate();
```

- Spark2.4.x版本

```
SparkSession spark = SparkSession
                .builder()
                .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
                .config("spark.sql.catalog.class",
"org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
                .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
                .config("spark.sql.hive.implementation","org.apache.spark.sql.hive.client.DliHiveClientI
mpl")
                .appName("java_spark_demo")
                .getOrCreate();
```

- Spark3.1.x版本

```
SparkSession spark = SparkSession
                .builder()
                .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
                .config("spark.sql.catalog.class",
"org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
                .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
                .appName("java_spark_demo")
                .getOrCreate();
```

- Spark3.3.x版本

```
SparkSession spark = SparkSession
        .builder()
        .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.DliLakeHouseBuilder")
        .config("spark.sql.catalog.class", "org.apache.spark.sql.hive.DliLakeHouseCatalog")
        .appName("java_spark_demo")
        .getOrCreate();
```

3. 创建数据库。

   如下样例代码演示，创建名为test_sparkapp的数据库。

   ```
   spark.sql("create database if not exists test_sparkapp").collect();
   ```

4. 创建DLI表并插入测试数据。

   ```
   spark.sql("drop table if exists test_sparkapp.dli_testtable").collect();
   spark.sql("create table test_sparkapp.dli_testtable(id INT, name STRING)").collect();
   spark.sql("insert into test_sparkapp.dli_testtable VALUES (123,'jason')").collect();
   spark.sql("insert into test_sparkapp.dli_testtable VALUES (456,'merry')").collect();
   ```

5. 创建OBS表。如下示例中的OBS路径需要根据**步骤2：OBS桶文件配置**中的实际数据路径修改。

   ```
   spark.sql("drop table if exists test_sparkapp.dli_testobstable").collect();
   spark.sql("create table test_sparkapp.dli_testobstable(age INT, name STRING) using csv options (path
   'obs://dli-test-obs01/testdata.csv')").collect();
   ```

6. 关闭SparkSession会话spark。

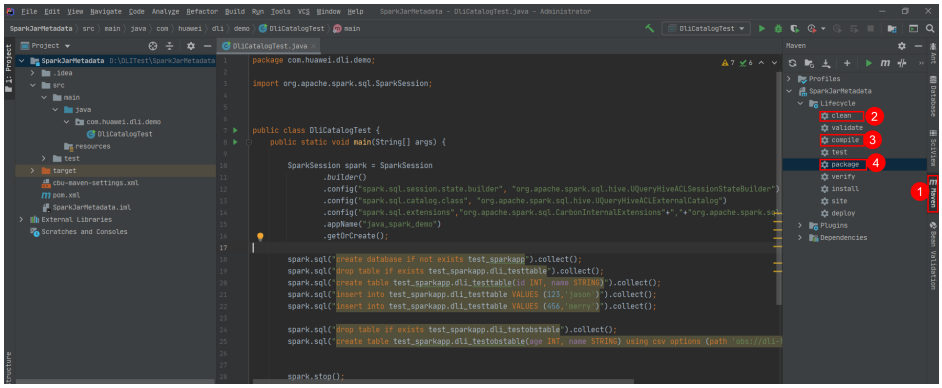
   ```
   spark.stop();
   ```

步骤 5：调试、编译代码并导出 Jar 包

1. 双击IntelliJ IDEA工具右侧的“Maven”，参考下图分别双击“clean”、“compile”对代码进行编译。

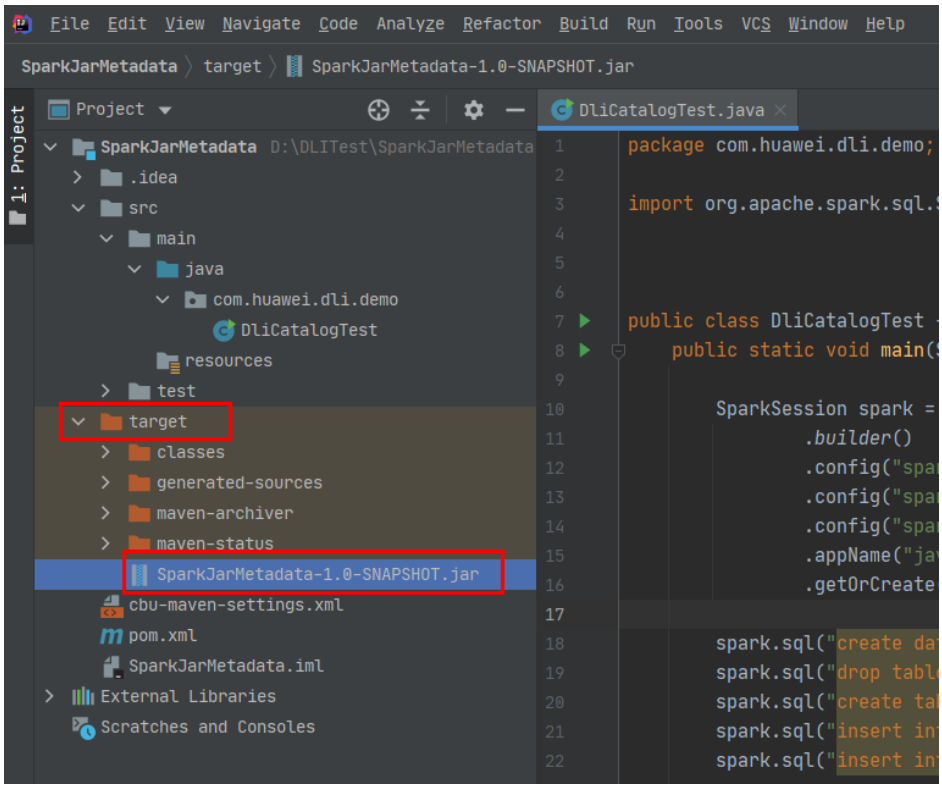
   编译成功后，双击“package”对代码进行打包。

图 4-26 编译打包



打包成功后，生成的Jar包会放到target目录下，以备后用。本示例将会生成到：“D:\DLITest\SparkJarMetadata\target”下名为“SparkJarMetadata-1.0-SNAPSHOT.jar”。

图 4-27 导出 jar 包



步骤 6：上传 Jar 包到 OBS 和 DLI 下

- **Spark 3.3及以上版本：**

  仅支持在创建Spark作业时，配置“应用程序”，从OBS选择作业所需的Jar包。

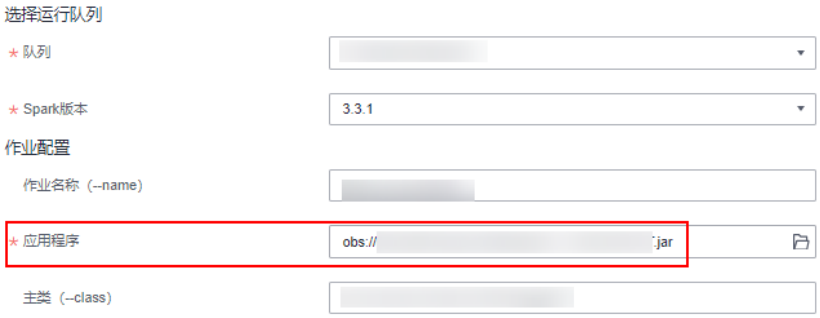
  a. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。

  b. 登录DLI控制台，选择“作业管理 > Spark作业”。

  c. 单击操作列“编辑”。

d. 编辑“应用程序”，选择a上传的OBS地址。

图 4-28 配置应用程序



- **Spark 3.3以下版本：**
  分别上传Jar包到OBS和DLI下。
  a. 登录OBS控制台，将生成的Jar包文件上传到OBS路径下。
  b. 将Jar包文件上传到DLI的程序包管理中，方便后续统一管理。
     i. 登录DLI管理控制台，单击“数据管理 > 程序包管理”。
     ii. 在“程序包管理”页面，单击右上角的“创建程序包”。
     iii. 在“创建程序包”对话框，配置以下参数。
        1) 包类型：选择“JAR”。
        2) OBS路径：程序包所在的OBS路径。
        3) 分组设置和组名称根据情况选择设置，方便后续识别和管理程序包。
     iv. 单击“确定”，完成创建程序包。

图 4-29 创建程序包

步骤 7：创建 Spark Jar 作业

1. 登录DLI控制台，单击“作业管理 > Spark作业”。
2. 在“Spark作业”管理界面，单击“创建作业”。
3. 在作业创建界面，配置对应作业运行参数。
   具体说明如**表4-11**所示，其他参数保持默认值即可。

表 4-11 Spark Jar 作业参数填写

参数名	参数值
所属队列	选择已创建的DLI通用队列。例如当前选择**步骤1：创建DLI通用队列**创建的通用队列“sparktest”。
Spark版本	选择Spark版本。在下拉列表中选择支持的Spark版本，推荐使用最新版本。
作业名称（--name）	自定义Spark Jar作业运行的名称。当前定义为：SparkTestMeta。
应用程序	选择**步骤6：上传Jar包到OBS和DLI下**中上传到DLI程序包。例如当前选择为：“SparkJarMetadata-1.0-SNAPSHOT.jar”。
主类	格式为：程序包名+类名。例如当前为：com.huawei.dli.demo.DliCatalogTest。

参数名	参数值
Spark参数（--conf）	spark.dli.metaAccess.enable=true spark.sql.warehouse.dir=*obs://dli-test-obs01/warehousepath* **说明** spark.sql.warehouse.dir参数的OBS路径为**步骤2：OBS桶文件配置**中配置创建。
访问元数据	选择：是

4. 单击“执行”，提交该Spark Jar作业。在Spark作业管理界面显示已提交的作业运行状态。

图 4-30 查看作业运行状态

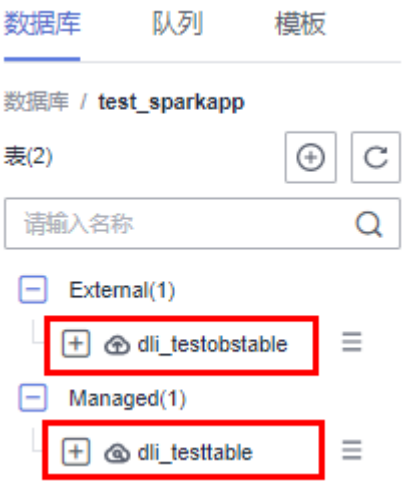
步骤 8：查看作业运行结果

1. 在Spark作业管理界面显示已提交的作业运行状态。初始状态显示为“启动中”。
2. 如果作业运行成功则作业状态显示为“已成功”，通过以下操作查看创建的数据库和表。
   a. 可以在DLI控制台，左侧导航栏，单击“SQL编辑器”。在“数据库”中已显示创建的数据库“test_sparkapp”。

图 4-31 查看创建的数据库

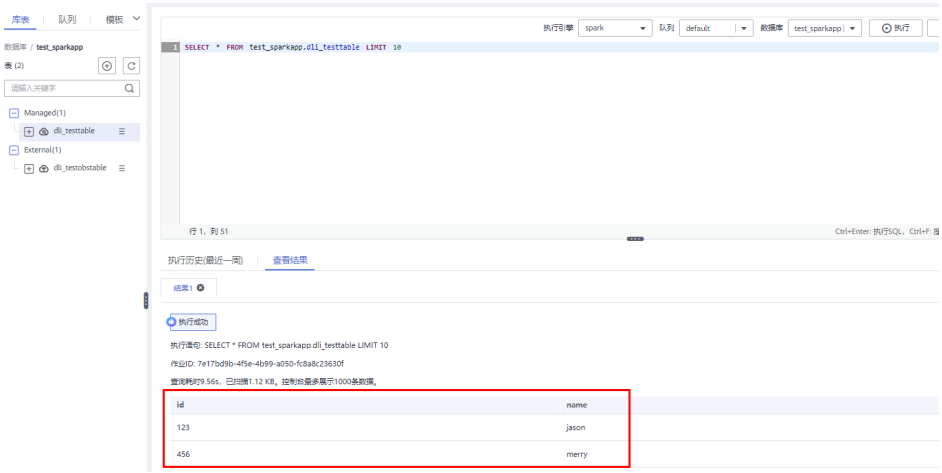
b. 双击数据库名，可以在数据库下查看已创建成功的DLI和OBS表。

图 4-32 查看表



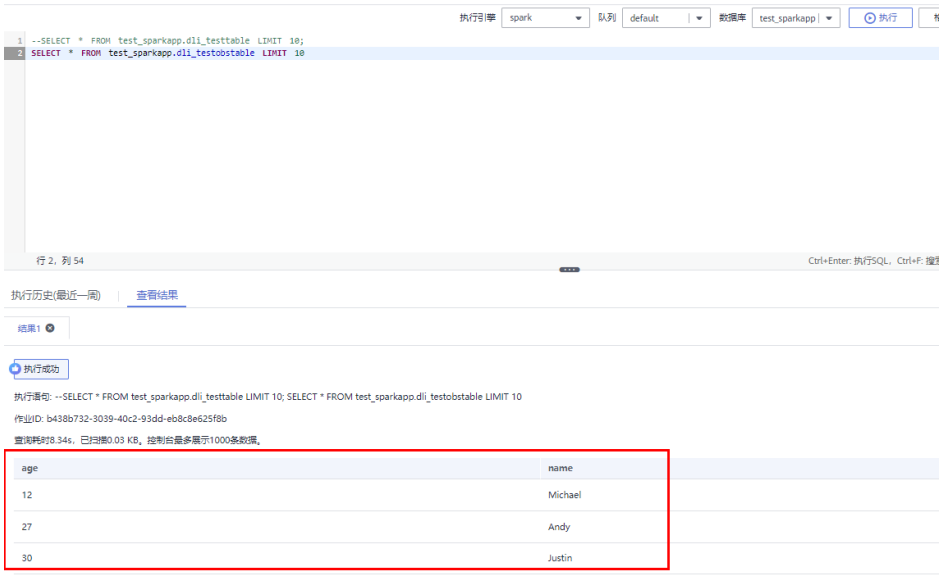
c. 双击DLI表名dli_testtable，单击“执行”查询DLI表数据。

图 4-33 查询 DLI 表数据



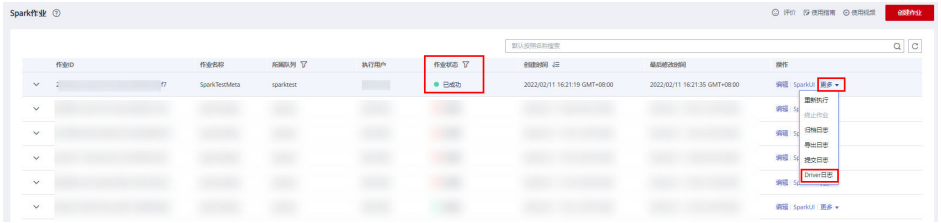
d. 注释掉DLI表查询语句，双击OBS表名dli_testobstable，单击“执行”查询OBS表数据。

图 4-34 查询 OBS 表数据



3. 如果作业运行失败则作业状态显示为“已失败”，单击“操作”列“更多”下的“Driver日志”，显示当前作业运行的日志，分析报错原因。

图 4-35 查看 Driver 日志



原因定位解决后，可以在作业“操作”列，单击“编辑”，修改作业相关参数后，单击“执行”重新运行该作业即可。

后续指引

- 如果您想通过Spark Jar作业访问其他数据源，请参考《**使用Spark作业跨源访问数据源**》。
- 创建DLI表的语法请参考**创建DLI表**，创建OBS表的语法请参考**创建OBS表**。
- 如果是通过API接口调用提交该作业请参考以下操作说明：

  调用创建批处理作业接口，参考以下请求参数说明。

  详细的API参数说明请参考《数据湖探索API参考》>《**创建批处理作业**》。

  - 将请求参数中的“catalog_name”参数设置为“dli”。
  - conf 中需要增加"spark.dli.metaAccess.enable":"true"。

    如果需要执行DDL，则还要在conf中配置"spark.sql.warehouse.dir": "obs://bucket/warehousepath"。

    完整的API请求参数可以参考如下示例说明。

    ```
    {
      "queue":"citest",
      "file":"SparkJarMetadata-1.0-SNAPSHOT.jar",
    ```



```
  "className":"DliCatalogTest",
  "conf":{"spark.sql.warehouse.dir": "obs://bucket/warehousepath",
  "spark.dli.metaAccess.enable":"true"},
  "sc_type":"A",
  "executorCores":1,
  "numExecutors":6,
  "executorMemory":"4G",
  "driverCores":2,
  "driverMemory":"7G",
  "catalog_name": "dli"
}
```

Java 样例代码

本示例操作步骤采用Java进行编码，具体完整的样例代码参考如下：

```
package com.huawei.dli.demo;

import org.apache.spark.sql.SparkSession;

public class DliCatalogTest {
    public static void main(String[] args) {

        SparkSession spark = SparkSession
                .builder()
                .config("spark.sql.session.state.builder",
"org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
                .config("spark.sql.catalog.class", "org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
                .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
                .appName("java_spark_demo")
                .getOrCreate();

        spark.sql("create database if not exists test_sparkapp").collect();
        spark.sql("drop table if exists test_sparkapp.dli_testtable").collect();
        spark.sql("create table test_sparkapp.dli_testtable(id INT, name STRING)").collect();
        spark.sql("insert into test_sparkapp.dli_testtable VALUES (123,'jason')").collect();
        spark.sql("insert into test_sparkapp.dli_testtable VALUES (456,'merry')").collect();

        spark.sql("drop table if exists test_sparkapp.dli_testobstable").collect();
        spark.sql("create table test_sparkapp.dli_testobstable(age INT, name STRING) using csv options (path
'obs://dli-test-obs01/testdata.csv')").collect();


        spark.stop();

    }
}
```

scala 样例代码

```
object DliCatalogTest {
  def main(args:Array[String]): Unit = {
    val sql = args(0)
    val runDdl =
Try(args(1).toBoolean).getOrElse(true)
    System.out.println(s"sql is $sql
runDdl is $runDdl")
    val sparkConf = new SparkConf(true)
    sparkConf
      .set("spark.sql.session.state.builder","org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
      .set("spark.sql.catalog.class","org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog")
    sparkConf.setAppName("dlicatalogtester")

    val spark = SparkSession.builder
      .config(sparkConf)
      .enableHiveSupport()
      .config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension")
      .appName("SparkTest")
```

```
    .getOrCreate()

  System.out.println("catalog is "
+ spark.sessionState.catalog.toString)
  if (runDdl) {
    val df = spark.sql(sql).collect()
  } else {
    spark.sql(sql).show()
  }

  spark.close()
 }

}
```

Python 样例代码

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

from __future__ import print_function

import sys

from pyspark.sql import SparkSession

if __name__ == "__main__":
    url = sys.argv[1]
    creatTbl = "CREATE TABLE test_sparkapp.dli_rds USING JDBC OPTIONS ('url'='jdbc:mysql://%s'," \
               "'driver'='com.mysql.jdbc.Driver','dbtable'='test.test'," \
               " 'passwdauth' = 'DatasourceRDSTest_pwd','encryption' = 'true')" % url

    spark = SparkSession \
        .builder \
        .enableHiveSupport() \
.config("spark.sql.session.state.builder","org.apache.spark.sql.hive.UQueryHiveACLSessionStateBuilder")
\
.config("spark.sql.catalog.class", "org.apache.spark.sql.hive.UQueryHiveACLExternalCatalog") \
.config("spark.sql.extensions","org.apache.spark.sql.DliSparkExtension") \
        .appName("python Spark test catalog") \
        .getOrCreate()

    spark.sql("CREATE database if not exists test_sparkapp").collect()
    spark.sql("drop table if exists test_sparkapp.dli_rds").collect()
    spark.sql(creatTbl).collect()
    spark.sql("select * from test_sparkapp.dli_rds").show()
    spark.sql("insert into table test_sparkapp.dli_rds select 12,'aaa'").collect()
    spark.sql("select * from test_sparkapp.dli_rds").show()
    spark.sql("insert overwrite table test_sparkapp.dli_rds select 1111,'asasasa'").collect()
    spark.sql("select * from test_sparkapp.dli_rds").show()
    spark.sql("drop table test_sparkapp.dli_rds").collect()
    spark.stop()
```

4.3 使用 Spark 作业跨源访问数据源

4.3.1 概述

DLI支持原生Spark的DataSource能力，并在其基础上进行了扩展，能够通过SQL语句或者Spark作业访问其他数据存储服务并导入、查询、分析处理其中的数据，目前支持的DLI跨源访问服务有：表格存储服务CloudTable，云搜索服务CSS，分布式缓存服务DCS，文档数据库服务DDS，数据仓库服务GaussDB（DWS），MapReduce服务MRS，云数据库RDS等。使用DLI的跨源能力，需要先创建跨源连接。

管理控制台界面具体操作请参考《**数据湖探索用户指南**》。

使用Spark作业跨源访问数据源支持使用scala，pyspark和java三种语言进行开发。

- 表格存储服务CloudTable相关介绍请参考《**表格存储服务产品介绍**》。
- 云搜索服务CSS相关介绍请参考《**云搜索服务产品介绍**》。
- 分布式缓存服务DCS相关介绍请参考《**分布式缓存服务产品介绍**》。
- 文档数据库服务DDS相关介绍请参考《**文档数据库服务产品介绍**》。
- 数据仓库服务GaussDB（DWS）相关介绍请参考《**数据仓库服务产品介绍**》。
- MapReduce服务MRS相关介绍请参考《**MapReduce服务产品介绍**》。
- 云数据库RDS相关介绍请参考《**云数据库服务产品介绍**》。

4.3.2 对接 CSS

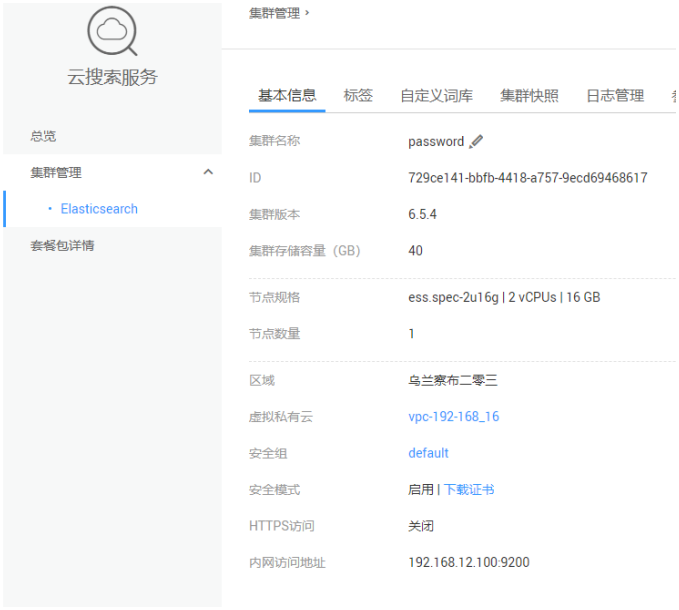
4.3.2.1 CSS 安全集群配置

准备工作

当前CSS服务提供的Elasticsearch 6.5.4或以上集群版本为用户增加了安全模式功能，开启安全模式后，将会为用户提供身份验证、授权以及加密等功能。DLI服务对接CSS安全集群时，需要先进行以下准备工作。

1. 选择CSS Elasticsearch 6.5.4或以上集群版本，创建CSS安全集群，并下载安全集群证书（CloudSearchService.cer）。
   a. 登录云搜索服务控制台，单击“集群管理”，选择需要建立跨源连接的集群，如图4-36所示：

   图 4-36 CSS 集群管理

   b. 单击“安全模式”中的“下载证书”下载安全证书。
2. 使用keytool工具生成keystore和truststore文件。
   a. 使用keytool工具生成keystore和truststore文件，其中需要使用到安全集群的安全证书（CloudSearchService.cer），keytool工具还有其他参数，可根据需求设置。

i. 打开cmd，输入下列命令生成含有一个私钥的keystore文件。
```
keytool -genkeypair -alias certificatekey -keyalg RSA -keystore transport-keystore.jks
```
ii. 使用keytool工具生成keystore和truststore文件后，可以在文件夹中看到transport-keystore.jks文件，使用如下命令验证keystore文件和证书信息。
```
keytool -list -v -keystore transport-keystore.jks
```
正确输入keystore password后即可看见相应的信息。
iii. 使用如下命令创建truststore.jks文件并进行验证 。
```
keytool -import -alias certificatekey -file CloudSearchService.cer  -keystore truststore.jks
keytool -list -v -keystore truststore.jks
```

b. 将生成的keystore和truststore文件上传到OBS桶中。

CSS 安全集群参数配置

具体参数请参考**表4-12**，这里主要说明配置CSS安全集群连接参数时需要注意的内容。

```
.option("es.net.http.auth.user", "admin") .option("es.net.http.auth.pass", "***")
```

此处的参数为身份验证的账号和密码，也是登录Kibana的账号和密码。

```
.option("es.net.ssl", "true")
```

- 如果CSS安全集群开启了HTTPS访问，此处需要设置为“true”，并且需要继续设置后面的安全证书、文件地址等参数。
- 如果CSS安全集群未开启HTTPS访问，此处需要设置为“false”，则不需要设置后面安全证书、文件地址等参数。

```
.option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")
.option("es.net.ssl.keystore.pass", "***")
```

此处设置keystore.jks文件的位置以及进入这个文件的密钥。在**准备工作**中生成的keystore.jks文件需要先放到OBS桶中，然后填入ak和sk以及jks文件的具体位置。最后在“es.net.ssl.keystore.pass”填入进入文件的密钥。

```
.option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
.option("es.net.ssl.truststore.pass", "***")
```

此处是truststore.jks文件的设置参数，与keystore.jks文件的设置参数基本一致，按照keystore.jks文件设置步骤进行操作即可。

4.3.2.2 scala 样例代码

前提条件

在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。

CSS 非安全集群

- 开发说明
  - 构造依赖信息，创建SparkSession
    i. 导入依赖

    涉及到的mvn依赖库
```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```


import相关依赖包

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}
```

ii. 创建会话。

```
val sparkSession = SparkSession.builder().getOrCreate()
```

- 通过SQL API访问

i. 创建DLI跨源访问 CSS的关联表。

```
sparkSession.sql("create table css_table(id int, name string) using css options(
  'es.nodes' 'to-css-1174404221-Y2bKVIqY.datasource.com:9200',
  'es.nodes.wan.only'='true',
  'resource' '/mytest/css')")
```

表 4-12 创建表参数

参数	说明
es.nodes	CSS的连接地址，需要先创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。 创建经典型跨源连接后，使用经典型跨源连接中返回的连接地址。 创建增强型跨源连接后，使用CSS提供的"内网访问地址"，格式为"IP1:PORT1,IP2:PORT2"。
resource	指定在CSS关联的资源名，用"/index/type"指定资源位置（可简单理解index为database，type为table，但绝不等同）。 **说明** • ES 6.X版本中，单个Index只支持唯一type，type名可以自定义。 • ES 7.X版本中，单个Index将使用“_doc”作为type名，不再支持自定义。若访问ES 7.X版本时，该参数只需要填写index即可。
pushdown	CSS的下压功能是否开启，默认为“true”。包含大量IO传输的表在有where过滤条件的情况下能够开启pushdown降低IO。
strict	CSS的下压是否是严格的，默认为“false”。精确匹配的场景下比pushdown降低更多IO。
batch.size.entries	单次batch插入entry的条数上限，默认为1000。如果单条数据非常大，在bulk存储设置的数据条数前提前到达了单次batch的总数据量上限，则停止存储数据，以batch.size.bytes为准，提交该批次的数据。
batch.size.bytes	单次batch的总数据量上限，默认为1mb。如果单条数据非常小，在bulk存储到总数据量前提前到达了单次batch的条数上限，则停止存储数据，以batch.size.entries为准，提交该批次的数据。
es.nodes.wan.only	是否仅通过域名访问es节点，默认为false。使用经典型跨源的连接地址作为es.nodes时，该参数需要配置为true；使用css服务提供的原始内网IP地址作为es.nodes时，不需要填写该参数或者配置为false。

参数	说明
es.mapping.id	指定一个字段，其值作为es中Document的id。 **说明** • 相同/index/type下的Document id是唯一的。如果作为Document id的字段存在重复值，则在执行插入es时，重复id的Document将会被覆盖。 • 该特性可以用作容错解决方案。当插入数据执行一半时，DLI作业失败，会有部分数据已经插入到es中，这部分为冗余数据。如果设置了Document id，则在重新执行DLI作业时，会覆盖上一次的冗余数据。

说明

batch.size.entries和batch.size.bytes分别对数据条数和数据量大小进行限制。

ii. 插入数据。

```
sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")
```

iii. 查询数据。

```
val dataFrame = sparkSession.sql("select * from css_table")
dataFrame.show()
```

插入数据前：

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
+---+----+\n
```

插入数据后：

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
| 13|John|\n
| 22| Bob|\n
+---+----+\n
```

iv. 删除数据表

```
sparkSession.sql("drop table css_table")
```

- 通过DataFrame API访问

i. 连接配置。

```
val resource = "/mytest/css"
val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"
```

ii. 构造schema，并添加数据。

```
val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name", StringType, false)))
val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))
```

iii. 导入数据到CSS。

```
val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)
dataFrame_1.write
```



```
.format("css")
.option("resource", resource)
.option("es.nodes", nodes)
.mode(SaveMode.Append)
.save()
```

说明

SaveMode 有四种保存类型：

- ErrorIfExis：如果已经存在数据，则抛出异常。
- Overwrite：如果已经存在数据，则覆盖原数据。
- Append：如果已经存在数据，则追加保存。
- Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

iv. 读取CSS上的数据

```
val dataFrameR =
sparkSession.read.format("css").option("resource",resource).option("es.nodes",
nodes).load()
dataFrameR.show()
```

插入数据前：

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
| 22| Bob|\n
+---+----+\n
```

插入数据后：

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
| 12|John|\n
| 21| Bob|\n
| 22| Bob|\n
+---+----+\n
```

- 提交Spark作业

i. 将写好的代码生成jar包，上传至OBS桶中。

ii. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择Spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.css。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在“Spark参数（--conf）” 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

- 完整示例代码
  - Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  - 通过SQL API访问

```
import org.apache.spark.sql.SparkSession

object Test_SQL_CSS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a DLI data table for DLI-associated CSS
    sparkSession.sql("create table css_table(id long, name string) using css options(
    'es.nodes' = 'to-css-1174404217-QG2SwbVV.datasource.com:9200',
    'es.nodes.wan.only' = 'true',
    'resource' = '/mytest/css')")

    //*****************************SQL model***********************************
    // Insert data into the DLI data table
    sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")

    // Read data from DLI data table
    val dataFrame = sparkSession.sql("select * from css_table")
    dataFrame.show()

    // drop table
    sparkSession.sql("drop table css_table")

    sparkSession.close()
  }
}
```

  - 通过DataFrame API访问

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession};
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType};

object Test_SQL_CSS {
  def main(args: Array[String]): Unit = {
    //Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    //*****************************DataFrame model***********************************
    // Setting the /index/type of CSS
    val resource = "/mytest/css"

    // Define the cross-origin connection address of the CSS cluster
    val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"

    //Setting schema
    val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name",
StringType, false)))

    // Construction data
    val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))

    // Create a DataFrame from RDD and schema
    val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)

   //Write data to the CSS
   dataFrame_1.write.format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .mode(SaveMode.Append)
    .save()
```

```
    //Read data
    val dataFrameR = sparkSession.read.format("css").option("resource",
resource).option("es.nodes", nodes).load()
    dataFrameR.show()

    spardSession.close()
  }
}
```

CSS 安全集群

- 开发说明
  - 构造依赖信息，创建SparkSession
    i. 导入依赖

    涉及到的mvn依赖库

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

    import相关依赖包

    ```
    import org.apache.spark.sql.{Row, SaveMode, SparkSession}
    import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}
    ```

    ii. 创建会话，并设置AK/SK。

    说明

    认证用的ak和sk硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

    ```
    val sparkSession = SparkSession.builder().getOrCreate()
    sparkSession.conf.set("fs.obs.access.key", ak)
    sparkSession.conf.set("fs.obs.secret.key", sk)
    sparkSession.conf.set("fs.obs.endpoint", enpoint)
    sparkSession.conf.set("fs.obs.connecton.ssl.enabled", "false")
    ```

  - 通过SQL API访问
    i. 创建DLI跨源访问 CSS的关联表。

    ```
    sparkSession.sql("create table css_table(id int, name string) using css options(
      'es.nodes' 'to-css-1174404221-Y2bKVIqY.datasource.com:9200',
      'es.nodes.wan.only'='true',
      'resource'='/mytest/css',
       'es.net.ssl'='true',
      'es.net.ssl.keystore.location'='obs://桶名/path/transport-keystore.jks',
      'es.net.ssl.keystore.pass'='***',
      'es.net.ssl.truststore.location'='obs://桶名/path/truststore.jks',
      'es.net.ssl.truststore.pass'='***',
      'es.net.http.auth.user'='admin',
      'es.net.http.auth.pass'='***')")
    ```

表 4-13 创建表参数

参数	说明
es.nodes	CSS的连接地址，需要先创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。 创建经典型跨源连接后，使用经典型跨源连接中返回的连接地址。 创建增强型跨源连接后，使用CSS提供的"内网访问地址"，格式为"IP1:PORT1,IP2:PORT2"。
resource	指定在CSS关联的资源名，用"/index/type"指定资源位置（可简单理解index为database，type为table，但绝不等同）。 **说明** 1. ES 6.X版本中，单个Index只支持唯一type，type名可以自定义。 2. ES 7.X版本中，单个Index将使用“_doc”作为type名，不再支持自定义。若访问ES 7.X版本时，该参数只需要填写index即可。
pushdown	CSS的下压功能是否开启，默认为“true”。包含大量IO传输的表在有where过滤条件的情况下能够开启pushdown降低IO。
strict	CSS的下压是否是严格的，默认为“false”。精确匹配的场景下比pushdown降低更多IO。
batch.size.entries	单次batch插入entry的条数上限，默认为1000。如果单条数据非常大，在bulk存储设置的数据条数前提前到达了单次batch的总数据量上限，则停止存储数据，以batch.size.bytes为准，提交该批次的数据。
batch.size.bytes	单次batch的总数据量上限，默认为1mb。如果单条数据非常小，在bulk存储到总数据量前提前到达了单次batch的条数上限，则停止存储数据，以batch.size.entries为准，提交该批次的数据。
es.nodes.wan.only	是否仅通过域名访问es节点，默认为“false”。使用经典型跨源的连接地址作为es.nodes时，该参数需要配置为“true”；使用CSS服务提供的原始内网IP地址作为es.nodes时，不需要填写该参数或者配置为“false”。
es.mapping.id	指定一个字段，其值作为es中Document的id。 **说明** • 相同/index/type下的Document id是唯一的。如果作为Document id的字段存在重复值，则在执行插入es时，重复id的Document将会被覆盖。 • 该特性可以用作容错解决方案。当插入数据执行一半时，DLI作业失败，会有部分数据已经插入到es中，这部分为冗余数据。如果设置了Document id，则在重新执行DLI作业时，会覆盖上一次的冗余数据。
es.net.ssl	连接安全CSS集群，默认值为“false”。

参数	说明
es.net.ssl.keystore.location	安全CSS集群的证书，生成的keystore文件在OBS上的地址。
es.net.ssl.keystore.pass	安全CSS集群的证书，生成的keystore文件时的密码。
es.net.ssl.truststore.location	安全CSS集群的证书，生成的truststore文件在OBS上的地址。
es.net.ssl.truststore.pass	安全CSS集群的证书，生成的truststore文件时的密码。
es.net.http.auth.user	安全CSS集群的用户名。
es.net.http.auth.pass	安全CSS集群的密码。

说明

“batch.size.entries”和“batch.size.bytes”分别对数据条数和数据量大小进行限制。

ii. 插入数据。

```
sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")
```

iii. 查询数据。

```
val dataFrame = sparkSession.sql("select * from css_table")
dataFrame.show()
```

插入数据前：

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
+---+----+\n
```

插入数据后：

```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
|  2| Bob|\n
| 13|John|\n
| 22| Bob|\n
+---+----+\n
```

iv. 删除数据表

```
sparkSession.sql("drop table css_table")
```

- 通过DataFrame API访问

  i. 连接配置。

  ```
  val resource = "/mytest/css"
  val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"
  ```

  ii. 构造schema，并添加数据。

  ```
  val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name", StringType, false)))
  val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))
  ```

  iii. 导入数据到CSS。

  ```
  val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)
  dataFrame_1.write
    .format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .option("es.net.ssl", "true")
    .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")
    .option("es.net.ssl.keystore.pass", "***")
    .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")
    .option("es.net.http.auth.user", "admin")
    .option("es.net.http.auth.pass", "***")
    .mode(SaveMode.Append)
    .save()
  ```

  说明

  SaveMode 有四种保存类型：

  - ErrorIfExis：如果已经存在数据，则抛出异常。
  - Overwrite：如果已经存在数据，则覆盖原数据。
  - Append：如果已经存在数据，则追加保存。
  - Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

  iv. 读取CSS上的数据

  ```
  val dataFrameR = sparkSession.read.format("css")
        .option("resource",resource)
        .option("es.nodes", nodes)
        .option("es.net.ssl", "true")
        .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")
        .option("es.net.ssl.keystore.pass", "***")
        .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
        .option("es.net.ssl.truststore.pass", "***")
        .option("es.net.http.auth.user", "admin")
        .option("es.net.http.auth.pass", "***")
        .load()
  dataFrameR.show()
  ```

  插入数据前：

  ```
  +---+----+\n
  | id|name|\n
  +---+----+\n
  |  1|John|\n
  | 22| Bob|\n
  +---+----+\n
  ```

  插入数据后：


```
+---+----+\n
| id|name|\n
+---+----+\n
|  1|John|\n
| 12|John|\n
| 21| Bob|\n
| 22| Bob|\n
+---+----+\n
```

- 提交Spark作业

  i. 将写好的代码生成jar包，上传至OBS桶中。

  ii. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**创建批处理作业**》。

  说明

  - 提交作业时，需要指定Module模块，名称为：sys.datasource.css。
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

- 完整示例代码
  - Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  - 通过SQL API访问

```
import org.apache.spark.sql.SparkSession

object csshttpstest {
  def main(args: Array[String]): Unit = {
    //Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    // Create a DLI data table for DLI-associated CSS
    sparkSession.sql("create table css_table(id long, name string) using css options('es.nodes' = 
'192.168.6.204:9200','es.nodes.wan.only' = 'false','resource' = '/
mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'obs://xietest1/lzq/
keystore.jks','es.net.ssl.keystore.pass' = '**','es.net.ssl.truststore.location'='obs://xietest1/lzq/
truststore.jks','es.net.ssl.truststore.pass'='**','es.net.http.auth.user'='admin','es.net.http.auth.pass'='*
*')")

    //*****************************SQL model***********************************
    // Insert data into the DLI data table
    sparkSession.sql("insert into css_table values(13, 'John'),(22, 'Bob')")

    // Read data from DLI data table
    val dataFrame = sparkSession.sql("select * from css_table")
    dataFrame.show()

    // drop table
    sparkSession.sql("drop table css_table")

    sparkSession.close()
  }
}
```

  - 通过DataFrame API访问

说明

认证用的ak和sk硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession};
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType};

object Test_SQL_CSS {
  def main(args: Array[String]): Unit = {
    //Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    sparkSession.conf.set("fs.obs.access.key", ak)
    sparkSession.conf.set("fs.obs.secret.key", sk)

    //****************************DataFrame model**********************************
    // Setting the /index/type of CSS
    val resource = "/mytest/css"

    // Define the cross-origin connection address of the CSS cluster
    val nodes = "to-css-1174405013-Ht7O1tYf.datasource.com:9200"

    //Setting schema
    val schema = StructType(Seq(StructField("id", IntegerType, false), StructField("name",
StringType, false)))

    // Construction data
    val rdd = sparkSession.sparkContext.parallelize(Seq(Row(12, "John"),Row(21,"Bob")))

    // Create a DataFrame from RDD and schema
    val dataFrame_1 = sparkSession.createDataFrame(rdd, schema)

   //Write data to the CSS
   dataFrame_1.write .format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .option("es.net.ssl", "true")
    .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")
    .option("es.net.ssl.keystore.pass", "***")
    .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")
    .option("es.net.http.auth.user", "admin")
    .option("es.net.http.auth.pass", "***")
    .mode(SaveMode.Append)
    .save();

    //Read data
    val dataFrameR = sparkSession.read.format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .option("es.net.ssl", "true")
    .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")
    .option("es.net.ssl.keystore.pass", "***")
    .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")
    .option("es.net.http.auth.user", "admin")
    .option("es.net.http.auth.pass", "***")
    .load()
    dataFrameR.show()

    spardSession.close()
  }
}
```

4.3.2.3 pyspark 样例代码

前提条件

在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。

CSS 非安全集群

- 开发说明
  - 代码实现详解
    i. import相关依赖包
    ```
    from __future__ import print_function
    from pyspark.sql.types import StructType, StructField, IntegerType, StringType, Row
    from pyspark.sql import SparkSession
    ```
    ii. 创建会话
    ```
    sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
    ```
  - 通过DataFrame API 访问
    i. 连接配置
    ```
    resource = "/mytest"
    nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"
    ```

    说明

    resource为指定在CSS关联的资源名。格式可以用"/index/type"指定资源位置（可简单理解index为database，type为table，但绝不等同）。
    - ES 6.X版本中，单个Index只支持唯一type，type名可以自定义。
    - ES 7.X版本中，单个Index将使用“_doc”作为type名，不再支持自定义。若访问ES 7.X版本时，该参数只需要填写index即可。

    ii. 构造schema，并添加数据
    ```
    schema = StructType([StructField("id", IntegerType(), False),
                         StructField("name", StringType(), False)])
    rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])
    ```
    iii. 构造DataFrame
    ```
    dataFrame = sparkSession.createDataFrame(rdd, schema)
    ```
    iv. 保存数据到CSS
    ```
    dataFrame.write.format("css").option("resource", resource).option("es.nodes",
    nodes).mode("Overwrite").save()
    ```

    说明

    mode 有四种保存类型：
    - ErrorIfExis：如果已经存在数据，则抛出异常。
    - Overwrite：如果已经存在数据，则覆盖原数据。
    - Append：如果已经存在数据，则追加保存。
    - Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

    v. 读取CSS上的数据
    ```
    jdbcDF = sparkSession.read.format("css").option("resource", resource).option("es.nodes",
    nodes).load()
    jdbcDF.show()
    ```
    vi. 操作结果

```
+---+----+
| id|name|
+---+----+
|  2| Bob|
|  1|John|
+---+----+
```

- 通过SQL API 访问
  i. 创建DLI跨源访问 CSS的关联表。

```
sparkSession.sql(
   "create table css_table(id long, name string) using css options(
   'es.nodes'='to-css-1174404953-hDTx3UPK.datasource.com:9200',
   'es.nodes.wan.only'='true',
   'resource'='/mytest')")
```

  说明

  创建CSS跨源表的参数详情可参考**表4-12**。

  ii. 插入数据

```
sparkSession.sql("insert into css_table values(3,'tom')")
```

  iii. 查询数据

```
jdbcDF = sparkSession.sql("select * from css_table")
jdbcDF.show()
```

  iv. 操作结果

```
+---+----+
| id|name|
+---+----+
|  3| tom|
|  2| Bob|
|  1|John|
+---+----+
```

- 提交Spark作业
  i. 将写好的python代码文件上传至OBS桶中。
  ii. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  说明

  - 如果选择Spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.css。
  - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在“Spark参数（--conf）” 配置

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

- 完整示例代码
  - 通过DataFrame API 访问

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import Row, StructType, StructField, IntegerType, StringType
```

```
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()

  # Setting cross-source connection parameters
  resource = "/mytest"
  nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),
                       StructField("name", StringType(), False)])

  # Construction data
  rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(rdd, schema)

  # Write data to the CSS
  dataFrame.write.format("css").option("resource", resource).option("es.nodes",
nodes).mode("Overwrite").save()

  # Read data
  jdbcDF = sparkSession.read.format("css").option("resource", resource).option("es.nodes",
nodes).load()
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

- 通过SQL API 访问

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()

  # Create a DLI data table for DLI-associated CSS
  sparkSession.sql(
      "create table css_table(id long, name string) using css options( \
      'es.nodes'='to-css-1174404953-hDTx3UPK.datasource.com:9200',\
      'es.nodes.wan.only'='true',\
      'resource'='/mytest')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into css_table values(3,'tom')")

  # Read data from DLI data table
  jdbcDF = sparkSession.sql("select * from css_table")
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

CSS 安全集群

- 开发说明
  - 代码实现详解
    i. import相关依赖包

```
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, Row
from pyspark.sql import SparkSession
```

ii. 创建会话并设置AK/SK

说明

认证用的ak和sk硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
sparkSession.conf.set("fs.obs.access.key", ak)
sparkSession.conf.set("fs.obs.secret.key", sk)
sparkSession.conf.set("fs.obs.endpoint", enpoint)
sparkSession.conf.set("fs.obs.connecton.ssl.enabled", "false")
```

- 通过DataFrame API 访问

i. 连接配置

```
resource = "/mytest";
nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"
```

说明

resource为指定在CSS关联的资源名。格式可以用"/index/type"指定资源位置（可简单理解index为database，type为table，但绝不等同）。

- ES 6.X版本中，单个Index只支持唯一type，type名可以自定义。
- ES 7.X版本中，单个Index将使用“_doc”作为type名，不再支持自定义。若访问ES 7.X版本时，该参数只需要填写index即可。

ii. 构造schema，并添加数据

```
schema = StructType([StructField("id", IntegerType(), False),
                     StructField("name", StringType(), False)])
rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])
```

iii. 构造DataFrame

```
dataFrame = sparkSession.createDataFrame(rdd, schema)
```

iv. 保存数据到CSS

```
dataFrame.write.format("css")
  .option("resource", resource)
  .option("es.nodes", nodes)
  .option("es.net.ssl", "true")
  .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")
  .option("es.net.ssl.keystore.pass", "***")
  .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
  .option("es.net.ssl.truststore.pass", "***")
  .option("es.net.http.auth.user", "admin")
  .option("es.net.http.auth.pass", "***")
  .mode("Overwrite")
  .save()
```

说明

mode 有四种保存类型：

- ErrorIfExis：如果已经存在数据，则抛出异常。
- Overwrite：如果已经存在数据，则覆盖原数据。
- Append：如果已经存在数据，则追加保存。
- Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

v. 读取CSS上的数据

```
jdbcDF = sparkSession.read.format("css")\
  .option("resource", resource)\
  .option("es.nodes", nodes)\
  .option("es.net.ssl", "true")\
  .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")\
  .option("es.net.ssl.keystore.pass", "***")\
  .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")\
  .option("es.net.ssl.truststore.pass", "***")\
```

```
  .option("es.net.http.auth.user", "admin")\
  .option("es.net.http.auth.pass", "***")\
  .load()
jdbcDF.show()
```

vi. 操作结果

```
+---+----+
| id|name|
+---+----+
|  2| Bob|
|  1|John|
+---+----+
```

- 通过SQL API 访问

i. 创建DLI跨源访问 CSS的关联表。

```
sparkSession.sql(
      "create table css_table(id long, name string) using css options(\
      'es.nodes'='to-css-1174404953-hDTx3UPK.datasource.com:9200',\
      'es.nodes.wan.only'='true',\
      'resource'='/mytest',\
   'es.net.ssl'='true',\
   'es.net.ssl.keystore.location'='obs://桶名/path/transport-keystore.jks',\
   'es.net.ssl.keystore.pass'='***',\
   'es.net.ssl.truststore.location'='obs://桶名/path/truststore.jks',\
   'es.net.ssl.truststore.pass'='***',\
   'es.net.http.auth.user'='admin',\
   'es.net.http.auth.pass'='***')")
```

说明

创建CSS跨源表的参数详情可参考**表4-12**。

ii. 插入数据

```
sparkSession.sql("insert into css_table values(3,'tom')")
```

iii. 查询数据

```
jdbcDF = sparkSession.sql("select * from css_table")
jdbcDF.show()
```

iv. 操作结果

```
+---+----+
| id|name|
+---+----+
|  3| tom|
|  2| Bob|
|  1|John|
+---+----+
```

- 提交Spark作业

i. 将写好的python代码文件上传至DLI中。控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**上传资源包**》。

ii. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**创建批处理作业**》。

说明

- 提交作业时，需要指定Module模块，名称为：sys.datasource.css。
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

- 完整示例代码
  - 通过DataFrame API 访问

    说明

    认证用的ak和sk硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import Row, StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
  sparkSession.conf.set("fs.obs.access.key", ak)
  sparkSession.conf.set("fs.obs.secret.key", sk)
  sparkSession.conf.set("fs.obs.endpoint", enpoint)
  sparkSession.conf.set("fs.obs.connecton.ssl.enabled", "false")

  # Setting cross-source connection parameters
  resource = "/mytest";
  nodes = "to-css-1174404953-hDTx3UPK.datasource.com:9200"

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),
                       StructField("name", StringType(), False)])

  # Construction data
  rdd = sparkSession.sparkContext.parallelize([Row(1, "John"), Row(2, "Bob")])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(rdd, schema)

  # Write data to the CSS
  dataFrame.write.format("css")
    .option("resource", resource)
    .option("es.nodes", nodes)
    .option("es.net.ssl", "true")
    .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")
    .option("es.net.ssl.keystore.pass", "***")
    .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")
    .option("es.net.http.auth.user", "admin")
    .option("es.net.http.auth.pass", "***")
    .mode("Overwrite")
    .save()

  # Read data
  jdbcDF = sparkSession.read.format("css")\
    .option("resource", resource)\
    .option("es.nodes", nodes)\
    .option("es.net.ssl", "true")\
    .option("es.net.ssl.keystore.location", "obs://桶名/path/transport-keystore.jks")\
    .option("es.net.ssl.keystore.pass", "***")\
    .option("es.net.ssl.truststore.location", "obs://桶名/path/truststore.jks")
    .option("es.net.ssl.truststore.pass", "***")\
    .option("es.net.http.auth.user", "admin")\
```

```
  .option("es.net.http.auth.pass", "***")\
  .load()
jdbcDF.show()

# close session
sparkSession.stop()
```

- 通过SQL API 访问

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession
import os

if __name__ == "__main__":

  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-css").getOrCreate()
  # Create a DLI data table for DLI-associated CSS
  sparkSession.sql("create table css_table(id int, name string) using css options(\
                  'es.nodes'='192.168.6.204:9200',\
                  'es.nodes.wan.only'='true',\
                  'resource'='/mytest',\
                  'es.net.ssl'='true',\
                  'es.net.ssl.keystore.location' = 'obs://xietest1/lzq/keystore.jks',\
                  'es.net.ssl.keystore.pass' = '**',\
                  'es.net.ssl.truststore.location'='obs://xietest1/lzq/truststore.jks',\
                  'es.net.ssl.truststore.pass'='**',\
                  'es.net.http.auth.user'='admin',\
                  'es.net.http.auth.pass'='**')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into css_table values(3,'tom')")

  # Read data from DLI data table
  jdbcDF = sparkSession.sql("select * from css_table")
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

4.3.2.4 java 样例代码

前提条件

在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。

CSS 非安全集群

- 开发说明
  - 代码实现
    - 构造依赖信息，创建SparkSession

      1) 导入依赖

      涉及到的mvn依赖库

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.3.2</version>
</dependency>
```

      import相关依赖包

```
import org.apache.spark.sql.SparkSession;
```

      2) 创建会话

```
SparkSession sparkSession = SparkSession.builder().appName("datasource-
css").getOrCreate();
```

- 通过SQL API访问
  i. 创建DLI跨源访问 CSS关联表。
  ```
  sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes'
  = '192.168.9.213:9200', 'es.nodes.wan.only' = 'true','resource' ='/mytest')");
  ```
  ii. 插入数据。
  ```
  sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");
  ```
  iii. 查询数据。
  ```
  sparkSession.sql("select * from css_table").show();
  ```
  iv. 删除数据表。
  ```
  sparkSession.sql("drop table css_table");
  ```
- 提交Spark作业
  i. 将写好的代码文件生成jar包，上传至OBS桶中。
  ii. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  说明

  - 如果选择Spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.css。
  - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在“Spark参数（--conf）” 配置

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/css/*
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

- 完整示例代码
  - Maven依赖
  ```
  <dependency>
          <groupId>org.apache.spark</groupId>
          <artifactId>spark-sql_2.11</artifactId>
          <version>2.3.2</version>
  </dependency>
  ```
  - 通过SQL API 访问
  ```
  import org.apache.spark.sql.*;

  public class java_css_unsecurity {

      public static void main(String[] args) {
          SparkSession sparkSession = SparkSession.builder().appName("datasource-css-
  unsecurity").getOrCreate();

          // Create a DLI data table for DLI-associated CSS
          sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes'
  = '192.168.15.34:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/mytest')");

          //*****************************SQL model***********************************
          // Insert data into the DLI data table
          sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");

          // Read data from DLI data table
          sparkSession.sql("select * from css_table").show();

          // drop table
  ```



```
        sparkSession.sql("drop table css_table");

        sparkSession.close();
    }
}
```

CSS 安全集群

- 准备工作

  请参考**CSS安全集群配置**，准备工作的主要目的是为了生成keystore.jks文件和truststore.jks文件，并将其上传至OBS桶中。

- 开发说明-https off

  如果没有开启https访问的话，不需要去生成keystore.jks和truststore.jks文件的，只需要设置好ssl访问和账号密码参数即可。

  - 构造依赖信息，创建SparkSession

    i. 导入依赖。

    涉及到的mvn依赖库：

    ```
    <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
    </dependency>
    ```

    import相关依赖包：

    ```
    import org.apache.spark.sql.SparkSession;
    ```

    ii. 创建会话。

    ```
    SparkSession sparkSession = SparkSession.builder().appName("datasource-css").getOrCreate();
    ```

  - 通过SQL API 访问

    i. 创建DLI跨源访问 CSS的关联表。

    ```
    sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes' = '192.168.9.213:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/mytest','es.net.ssl'='false','es.net.http.auth.user'='admin','es.net.http.auth.pass'='*******')");
    ```

    说明

    - 创建CSS跨源表的参数详情可参考**表4-12**。
    - 上述示例中，因为CSS安全集群关闭了https访问，所以“es.net.ssl”参数要设置为“false”。“es.net.http.auth.user”以及“es.net.http.auth.pass”为创建集群时设置的账号和密码。

    ii. 插入数据

    ```
    sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");
    ```

    iii. 查询数据

    ```
    sparkSession.sql("select * from css_table").show();
    ```

    iv. 删除数据表

    ```
    sparkSession.sql("drop table css_table");
    ```

  - 提交Spark作业

    i. 将写好的代码文件生成jar包，上传至DLI中。

    控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**上传资源包**》。

    ii. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

    控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**创建批处理作业**》。

说明

- 提交作业时，需要指定Module模块，名称为：sys.datasource.css。
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

- 完整示例代码
  - Maven依赖

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.3.2</version>
</dependency>
```

- 开发说明-https on
  - 构造依赖信息，创建SparkSession
    i. 导入依赖。

    涉及到的mvn依赖库：

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.3.2</version>
</dependency>
```

    import相关依赖包：

```
import org.apache.spark.SparkFiles;
import org.apache.spark.sql.SparkSession;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
```

    ii. 创建会话。

```
SparkSession sparkSession = SparkSession.builder().appName("datasource-
css").getOrCreate();
```

    iii. 拷贝证书。

```
    sparkSession.sparkContext().addFile("obs://桶名/地址/transport-keystore.jks");
    sparkSession.sparkContext().addFile("obs://桶名/地址/truststore.jks");

      // 获取当前工作目录的路径
      String pathUser = System.getProperty("user.dir");
      System.out.println("path_user is " + pathUser);

      // 获取文件名
      String esTransportKeystoreFileName = SparkFiles.get("transport-keystore.jks");
      String esTruststoreFileName = SparkFiles.get("truststore.jks");

      System.out.println("esTransportKeystoreFileName is " +
esTransportKeystoreFileName);
      System.out.println("esTruststoreFileName is " + esTruststoreFileName);
      // 拼接文件完整路径
      String esTransportKeystoreLocalPath = pathUser + "/" + "transport-keystore.jks";
      String esTruststoreLocalPath = pathUser + "/" + "truststore.jks";

      System.out.println("esTransportKeystoreLocalPath is " +
esTransportKeystoreLocalPath);
      System.out.println("esTruststoreLocalPath is " + esTruststoreLocalPath);
      try {
         // 拷贝 keystore 文件
         copyFile(esTransportKeystoreFileName, esTransportKeystoreLocalPath);
         // 拷贝 truststore 文件
         copyFile(esTruststoreFileName, esTruststoreLocalPath);
```

```
            // 等待一段时间
            Thread.sleep(2000);

            System.out.println("Files copied successfully:");
            System.out.println("es_transport-keystore.jks: " + esTransportKeystoreLocalPath);
            System.out.println("es_truststore.jks: " + esTruststoreLocalPath);
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
```

- 通过SQL API 访问

  i. 创建DLI跨源访问 CSS的关联表。

  ```
  sparkSession.sql("create table css_table(id long, name string) using css options( 'es.nodes' = '192.168.13.189:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'file://" + esTransportKeystoreLocalPath + "','es.net.ssl.keystore.pass' = '**', 'es.net.ssl.truststore.location'='file://" + esTruststoreLocalPath + "', 'es.net.ssl.truststore.pass'='***','es.net.http.auth.user'='admin','es.net.http.auth.pass'='**')");
  ```

  说明

  创建CSS跨源表的参数详情可参考**表4-12**。

  ii. 插入数据

  ```
  sparkSession.sql("insert into css_table values(18, 'John'),(28, 'Bob')");
  ```

  iii. 查询数据

  ```
  sparkSession.sql("select * from css_table").show();
  ```

  iv. 删除数据表

  ```
  sparkSession.sql("drop table css_table");
  ```

- 提交Spark作业

  i. 将写好的代码文件生成jar包，上传至DLI中。

  控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**上传资源包**》。

  ii. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**创建批处理作业**》。

  说明

  - 提交作业时，需要指定Module模块，名称为：sys.datasource.css。
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“表6-选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

- 完整示例代码

  - Maven依赖

  ```
  <dependency>
          <groupId>org.apache.spark</groupId>
          <artifactId>spark-sql_2.11</artifactId>
          <version>2.3.2</version>
  </dependency>
  ```

  - 通过SQL API 访问

  ```
  import org.apache.spark.SparkFiles;
  import org.apache.spark.sql.SparkSession;
  import java.io.IOException;
  import java.nio.file.Files;
  import java.nio.file.Paths;
  ```

```
public class java_css_security_httpson {
    public static void main(String[] args) {
        SparkSession sparkSession = SparkSession.builder().appName("datasource-
css").getOrCreate();

        sparkSession.sparkContext().addFile("obs://桶名/地址/transport-keystore.jks");
        sparkSession.sparkContext().addFile("obs://桶名/地址/css/truststore.jks");

        // 获取当前工作目录的路径
        String pathUser = System.getProperty("user.dir");
        System.out.println("path_user is " + pathUser);

        // 获取文件名
        String esTransportKeystoreFileName = SparkFiles.get("transport-keystore.jks");
        String esTruststoreFileName = SparkFiles.get("truststore.jks");

        System.out.println("esTransportKeystoreFileName is " +
esTransportKeystoreFileName);
        System.out.println("esTruststoreFileName is " + esTruststoreFileName);
        // 拼接文件完整路径
        String esTransportKeystoreLocalPath = pathUser + "/" + "transport-keystore.jks";
        String esTruststoreLocalPath = pathUser + "/" + "truststore.jks";

        System.out.println("esTransportKeystoreLocalPath is " +
esTransportKeystoreLocalPath);
        System.out.println("esTruststoreLocalPath is " + esTruststoreLocalPath);
        try {
            // 拷贝 keystore 文件
            copyFile(esTransportKeystoreFileName, esTransportKeystoreLocalPath);
            // 拷贝 truststore 文件
            copyFile(esTruststoreFileName, esTruststoreLocalPath);
            // 等待一段时间
            Thread.sleep(2000);

            System.out.println("Files copied successfully:");
            System.out.println("es_transport-keystore.jks: " + esTransportKeystoreLocalPath);
            System.out.println("es_truststore.jks: " + esTruststoreLocalPath);
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }

        // Create a DLI data table for DLI-associated CSS
        sparkSession.sql("create table css_table(id long, name string) using css
options( 'es.nodes' = '192.168.13.189:9200', 'es.nodes.wan.only' = 'true', 'resource' = '/
mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'file://" +
esTransportKeystoreLocalPath + "','es.net.ssl.keystore.pass' =
'**','es.net.ssl.truststore.location'='file://" + esTruststoreLocalPath +
"','es.net.ssl.truststore.pass'='**','es.net.http.auth.user'='admin','es.net.http.auth.pass'='**')");

        //*****************************SQL model***********************************
        // Insert data into the DLI data table
        sparkSession.sql("insert into css_table values(34, 'Yuan'),(28, 'Kids')");

        // Read data from DLI data table
        sparkSession.sql("select * from css_table").show();

        // drop table
        sparkSession.sql("drop table css_table");

        sparkSession.close();
    }
    private static void copyFile(String sourcePath, String destinationPath) throws
IOException {
        // 从远程存储复制文件到本地的操作
        byte[] fileContent = Files.readAllBytes(Paths.get(sourcePath));
        Files.write(Paths.get(destinationPath), fileContent);
    }
}
```

4.3.3 对接 DWS

4.3.3.1 scala 样例代码

操作场景

本例提供使用Spark作业访问DWS数据源的scala样例代码。

在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

说明

认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

操作前准备

构造依赖信息，创建SparkSession

1. 导入依赖

   涉及到的mvn依赖库

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

   import相关依赖包

```
import java.util.Properties
import org.apache.spark.sql.{Row,SparkSession}
import org.apache.spark.sql.SaveMode
```

2. 创建会话。

```
val sparkSession = SparkSession.builder().getOrCreate()
```

通过 SQL API 访问数据源

1. 创建DLI跨源访问DWS的关联表。

```
sparkSession.sql(
  "CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (
     'url'='jdbc:postgresql://to-dws-1174404209-cA37siB6.datasource.com:8000/postgres',
     'dbtable'='customer',
     'user'='dbadmin',
     'passwdauth'='######'//DLI侧创建的Password类型的跨源认证名称。使用跨源认证则无需在作业中配置账号和密码。
)"
)
```

表 4-14 创建表参数

参数	说明
url	DWS的连接地址，需要先创建跨源连接，管理控制台操作请参考《数据湖探索用户指南》。 创建经典型跨源连接后，使用经典型跨源连接中返回的连接地址。 创建增强型跨源连接后，可以使用DWS提供的"JDBC连接字符串（内网）"，或者内网地址和内网端口访问，格式为"协议头://内网IP:内网端口/数据库名"，例如："jdbc:postgresql://192.168.0.77:8000/postgres"，获取方式请参考“图 DWS集群信息”。 **说明** DWS的连接地址格式为："协议头://访问地址:访问端口/数据库名" 例如： jdbc:postgresql://to-dws-1174405119-ihlUr78j.datasource.com:8000/postgres 如果想要访问DWS中自定义数据库，请在这个连接里将"postgres"修改为对应的数据库名字。
passwdauth	DLI侧创建的Password类型的跨源认证名称。使用跨源认证则无需在作业中配置账号和密码。
dbtable	数据库postgres中的数据表。
partitionColumn	读取数据时，用于设置并发使用的数值型字段。 **说明** • “partitionColumn”，“lowerBound”，“upperBound”，“numPartitions”4个参数必须同时设置，不支持仅设置其中一部分。 • 为了提升并发读取的性能，建议使用自增列。
lowerBound	partitionColumn设置的字段数据最小值，该值包含在返回结果中。
upperBound	partitionColumn设置的字段数据最大值，该值不包含在返回结果中。
numPartitions	读取数据时并发数。 **说明** 实际读取数据时，会根据lowerBound与upperBound，平均分配给每个task获取其中一部分的数据。例如： 'partitionColumn'='id', 'lowerBound'='0', 'upperBound'='100', 'numPartitions'='2' DLI中会起2个并发task，一个task执行id>=0 and id < 50，另一个task执行id >=50 and id < 100。
fetchsize	读取数据时，每一批次获取数据的记录数，默认值1000。设置越大性能越好，但占用内存越多，该值设置过大会有内存溢出的风险。

参数	说明
batchsize	写入数据时，每一批次写入数据的记录数，默认值1000。设置越大性能越好，但占用内存越多，该值设置过大会有内存溢出的风险。
truncate	执行overwrite时是否不删除原表，直接执行清空表操作，取值范围： • true • false 默认为“false”，即在执行overwrite操作时，先将原表删除再重新建表。
isolationLevel	事务隔离级别，取值范围： • NONE • READ_UNCOMMITTED • READ_COMMITTED • REPEATABLE_READ • SERIALIZABLE 默认值为“READ_UNCOMMITTED”。

图 4-37 DWS 集群信息

2. 插入数据

```
sparkSession.sql("insert into dli_to_dws values(1, 'John',24),(2, 'Bob',32)")
```

3. 查询数据

```
val dataFrame = sparkSession.sql("select * from dli_to_dws")
dataFrame.show()
```

插入数据前：


```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  7|   chm| 13|
|  6|    qz| 13|
|  3|  mark| 20|
+---+------+---+
```

插入数据后：

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  6|    qz| 13|
|  7|   chm| 13|
|  3|  mark| 20|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  1|  John| 24|
|  2|   Bob| 32|
+---+------+---+
```

4. 删除关联表

```
sparkSession.sql("drop table dli_to_dws")
```

通过 DataFrame API 访问数据源

1. 连接配置。

```
val url = "jdbc:postgresql://to-dws-1174405057-EA1Kgo8H.datasource.com:8000/postgres"
val username = "dbadmin"
val password = "######"
val dbtable = "customer"
```

2. 创建DataFrame，添加数据，并重命名字段。

```
var dataFrame_1 = sparkSession.createDataFrame(List((8, "Jack_1", 18)))
val df = dataFrame_1.withColumnRenamed("_1", "id")
                    .withColumnRenamed("_2", "name")
                    .withColumnRenamed("_3", "age")
```

3. 导入数据到DWS。

```
df.write.format("jdbc")
  .option("url", url)
  .option("dbtable", dbtable)
  .option("user", username)
  .option("password", password)
  .mode(SaveMode.Append)
  .save()
```

说明

SaveMode 有四种保存类型：

- ErrorIfExis：如果已经存在数据，则抛出异常。
- Overwrite：如果已经存在数据，则覆盖原数据。
- Append：如果已经存在数据，则追加保存。
- Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

4. 读取DWS上的数据。

   - 方式一：read.format()方法

     ```
     val jdbcDF = sparkSession.read.format("jdbc")
                  .option("url", url)
                  .option("dbtable", dbtable)
                  .option("user", username)
                  .option("password", password)
                  .load()
     ```

   - 方式二：read.jdbc()方法

     ```
     val properties = new Properties()
      properties.put("user", username)
      properties.put("password", password)
      val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)
     ```

   插入数据前：

   ```
   +---+------+---+\n
   | id|  name|age|\n
   +---+------+---+\n
   |  4|  kobe| 24|\n
   |  3|  mark| 20|\n
   |  7|   chm| 13|\n
   |  6|    qz| 13|\n
   |  1|   tom| 18|\n
   |  2|  ammy| 18|\n
   |  5|jordan| 22|\n
   +---+------+---+\n
   ```

   插入数据后：

   ```
   +---+------+---+\n
   | id|  name|age|\n
   +---+------+---+\n
   |  7|   chm| 13|\n
   |  1|   tom| 18|\n
   |  2|  ammy| 18|\n
   |  5|jordan| 22|\n
   |  8|Jack_1| 18|\n
   |  3|  mark| 20|\n
   |  6|    qz| 13|\n
   |  4|  kobe| 24|\n
   +---+------+---+\n
   ```

   使用上述read.format()或者read.jdbc()方法读取到的dateFrame注册为临时表，就可使用sql语句进行数据查询了。

   ```
   jdbcDF.registerTempTable("customer_test")
    sparkSession.sql("select * from customer_test where id = 1").show()
   ```

   查询结果：


```
+---+----+---+
| id|name|age|
+---+----+---+
|  1| tom| 18|
+---+----+---+
```

DataFrame 相关操作

createDataFrame() 方法创建的数据和read.format() 方法及read.jdbc() 方法查询的数据都为DataFrame对象，可以直接进行查询单条记录等操作（在“**通过DataFrame API访问数据源**”中，提到将DataFrame数据注册为临时表）。

- where

  where 方法中可传入包含and 和 or 的条件筛选表达式，返回过滤后的DataFrame对象，示例如下：

  ```
  jdbcDF.where("id = 1 or age <=10").show()
  ```

  ```
  +---+------+---+
  | id|  name|age|
  +---+------+---+
  |  7|   chm| 13|
  |  1|   tom| 18|
  |  2|  ammy| 18|
  |  5|jordan| 22|
  |  6|    qz| 13|
  |  3|  mark| 20|
  +---+------+---+
  ```

- filter

  filter同where的使用方式一致，传入条件筛选表达式，返回过滤后的结果 。示例如下：

  ```
  jdbcDF.filter("id = 1 or age <=10").show()
  ```

  ```
  +---+------+---+
  | id|  name|age|
  +---+------+---+
  |  4|  kobe| 24|
  |  7|   chm| 13|
  |  5|jordan| 22|
  |  6|    qz| 13|
  |  3|  mark| 20|
  +---+------+---+
  ```

- select

  传入待查询的字段，返回指定字段的DataFrame对象，并且可多个字段查询，示例如下：

  - 示例1：

    ```
    jdbcDF.select("id").show()
    ```

```
+---+
| id|
+---+
|  4|
|  7|
|  3|
|  6|
|  1|
|  2|
|  5|
+---+
```

- 示例2：
  jdbcDF.select("**id**", "**name**").show()

```
+---+------+
| id|  name|
+---+------+
|  4|  kobe|
|  7|   chm|
|  6|    qz|
|  3|  mark|
|  1|   tom|
|  2|  ammy|
|  5|jordan|
+---+------+
```

- 示例3：
  jdbcDF.select("**id**","**name**").where("**id<4**").show()

```
+---+----+
| id|name|
+---+----+
|  1| tom|
|  2|ammy|
|  3|mark|
+---+----+
```

- selectExpr

  对字段进行特殊处理。例如，可使用selectExpr修改字段名。示例如下：

  将name字段取名name_test，age数据加1。

  jdbcDF.selectExpr("**id**", "**name as name_test**", "**age+1**").show()

- col

  获取指定字段。不同于select，col每次只能获取一个字段，返回类型为Column类型，示例如下：

  val idCol = jdbcDF.col("**id**")

- drop

  删除指定字段。传入要删除的字段，返回不包含此字段的DataFrame对象，并且每次只能删除一个字段，示例如下：

  jdbcDF.drop("**id**").show()


```
+----+---+
|name|age|
+----+---+
|  qz| 13|
| chm| 13|
| tom| 18|
|ammy| 18|
+----+---+
```

提交作业

1. 将写好的代码生成jar包，上传至OBS桶中。
2. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

   说明

   - 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.dws。
   - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

     spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*

     spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*
   - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
   - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- 通过SQL API访问

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
import java.util.Properties
import org.apache.spark.sql.SparkSession

object Test_SQL_DWS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    // Create a data table for DLI-associated DWS
    sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (
      'url'='jdbc:postgresql://to-dws-1174405057-EA1Kgo8H.datasource.com:8000/postgres',
      'dbtable'='customer',
      'user'='dbadmin',
      'password'='######')")

    //*****************************SQL model***********************************
    //Insert data into the DLI data table
    sparkSession.sql("insert into dli_to_dws values(1,'John',24),(2,'Bob',32)")
```

```
  //Read data from DLI data table
  val dataFrame = sparkSession.sql("select * from dli_to_dws")
  dataFrame.show()

  //drop table
  sparkSession.sql("drop table dli_to_dws")

  sparkSession.close()
 }
}
```

- 通过DataFrame API访问

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
import java.util.Properties
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.SaveMode

object Test_SQL_DWS {
 def main(args: Array[String]): Unit = {
  // Create a SparkSession session.
  val sparkSession = SparkSession.builder().getOrCreate()

  //*****************************DataFrame model***********************************
  // Set the connection configuration parameters. Contains url, username, password, dbtable.
  val url = "jdbc:postgresql://to-dws-1174405057-EA1Kgo8H.datasource.com:8000/postgres"
  val username = "dbadmin"
  val password = "######"
  val dbtable = "customer"

  //Create a DataFrame and initialize the DataFrame data.
  var dataFrame_1 = sparkSession.createDataFrame(List((1, "Jack", 18)))

  //Rename the fields set by the createDataFrame() method.
  val df = dataFrame_1.withColumnRenamed("_1", "id")
              .withColumnRenamed("_2", "name")
              .withColumnRenamed("_3", "age")

  //Write data to the dws_table_1 table
  df.write.format("jdbc")
    .option("url", url)
    .option("dbtable", dbtable)
    .option("user", username)
    .option("password", password)
    .mode(SaveMode.Append)
    .save()

  // DataFrame object for data manipulation
  //Filter users with id=1
  var newDF = df.filter("id!=1")
  newDF.show()

  // Filter the id column data
  var newDF_1 = df.drop("id")
  newDF_1.show()

  // Read the data of the customer table in the RDS database
  //Way one：Read data from DWS using read.format()
  val jdbcDF = sparkSession.read.format("jdbc")
              .option("url", url)
              .option("dbtable", dbtable)
              .option("user", username)
              .option("password", password)
              .option("driver", "org.postgresql.Driver")
              .load()
  //Way two：Read data from DWS using read.jdbc()
```



```
    val properties = new Properties()
    properties.put("user", username)
    properties.put("password", password)
    val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)

    /**
     * Register the dateFrame read by read.format() or read.jdbc() as a temporary table, and query the
data
     * using the sql statement.
     */
    jdbcDF.registerTempTable("customer_test")
    val result = sparkSession.sql("select * from customer_test where id = 1")
    result.show()

    sparkSession.close()
  }
}
```

4.3.3.2 pyspark 样例代码

操作场景

本例提供使用Spark作业访问DWS数据源的pyspark样例代码。

在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

说明

认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

操作前准备

1. import相关依赖包

```
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession
```

2. 创建会话

```
sparkSession = SparkSession.builder.appName("datasource-dws").getOrCreate()
```

通过 DataFrame API 访问数据源

1. 连接参数配置

```
url = "jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres"
dbtable = "customer"
user = "dbadmin"
password = "######"
driver = "org.postgresql.Driver"
```

2. 设置数据

```
dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19)])
```

3. 设置schema

```
schema = StructType([StructField("id", IntegerType(), False),\
                     StructField("name", StringType(), False),\
                     StructField("age", IntegerType(), False)])
```

4. 创建DataFrame

```
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

5. 保存数据到DWS

```
dataFrame.write \
    .format("jdbc") \
```

```
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .mode("Overwrite") \
    .save()
```

说明

mode 有四种保存类型：

- ErrorIfExis：如果已经存在数据，则抛出异常。
- Overwrite：如果已经存在数据，则覆盖原数据。
- Append：如果已经存在数据，则追加保存。
- Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

6. 读取DWS上的数据

```
jdbcDF = sparkSession.read \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .load()
jdbcDF.show()
```

7. 操作结果

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  1|Katie| 19|
+---+-----+---+
```

通过 SQL API 访问数据源

1. 创建DLI跨源访问 dws 的关联表。

```
sparkSession.sql(
    "CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (
    'url'='jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres',\
    'dbtable'='customer',\
    'user'='dbadmin',\
    'password'='######',\
    'driver'='org.postgresql.Driver')")
```

说明

建表参数详情可参考**表4-14**。

2. 插入数据

```
sparkSession.sql("insert into dli_to_dws values(2,'John',24)")
```

3. 查询数据

```
jdbcDF = sparkSession.sql("select * from dli_to_dws").show()
```

4. 操作结果

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  1|Katie| 19|
|  2| John| 24|
+---+-----+---+
```


提交 Spark 作业

1. 将写好的python代码文件上传至OBS桶中。
2. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.dws。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf)' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- 通过DataFrame API访问

说明

认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-dws").getOrCreate()

  # Set cross-source connection parameters
  url = "jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres"
  dbtable = "customer"
  user = "dbadmin"
  password = "######"
  driver = "org.postgresql.Driver"

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19)])

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),\
                       StructField("name", StringType(), False),\
                       StructField("age", IntegerType(), False)])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(dataList, schema)

  # Write data to the DWS table
  dataFrame.write \
      .format("jdbc") \
      .option("url", url) \
      .option("dbtable", dbtable) \
      .option("user", user) \
      .option("password", password) \
      .option("driver", driver) \
      .mode("Overwrite") \
      .save()
```

```
# Read data
jdbcDF = sparkSession.read \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .load()
jdbcDF.show()

# close session
sparkSession.stop()
```

- 通过SQL API访问

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-dws").getOrCreate()

  # Createa data table for DLI - associated DWS
  sparkSession.sql(
      "CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS (\
      'url'='jdbc:postgresql://to-dws-1174404951-W8W4cW8I.datasource.com:8000/postgres',\
      'dbtable'='customer',\
      'user'='dbadmin',\
      'password'='######',\
      'driver'='org.postgresql.Driver')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into dli_to_dws values(2,'John',24)")

  # Read data from DLI data table
  jdbcDF = sparkSession.sql("select * from dli_to_dws").show()

  # close session
  sparkSession.stop()
```

4.3.3.3 java 样例代码

操作场景

本例提供使用Spark作业访问DWS数据源的java样例代码。

在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

说明

认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

操作前准备

1. 导入依赖
   - 涉及到的mvn依赖库

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```


- import相关依赖包
  ```
  import org.apache.spark.sql.SparkSession;
  ```

2. 创建会话
   ```
   SparkSession sparkSession = SparkSession.builder().appName("datasource-dws").getOrCreate();
   ```

通过 SQL API 访问数据源

1. 创建DLI跨源访问DWS的关联表，填写连接参数。
   ```
   sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS
   ('url'='jdbc:postgresql://10.0.0.233:8000/postgres','dbtable'='test','user'='dbadmin','password'='**')");
   ```
2. 插入数据
   ```
   sparkSession.sql("insert into dli_to_dws values(3,'Liu'),(4,'Xie')");
   ```
3. 查询数据
   ```
   sparkSession.sql("select * from dli_to_dws").show();
   ```
   插入数据后：

   ```
   +----+-----------+
   |  id|       name|
   +----+-----------+
   |1111|gff_test001|
   |   3|        Liu|
   |   1|       John|
   |   2|        Bob|
   |   4|        Xie|
   +----+-----------+
   ```

提交 Spark 作业

1. 将写好的代码文件生成jar包，上传至OBS桶中。
2. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

   说明

   - 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.dws。
   - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

     spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*

     spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/dws/*
   - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
   - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

通过SQL API 访问DWS表

```
import org.apache.spark.sql.SparkSession;

public class java_dws {
    public static void main(String[] args) {
        SparkSession sparkSession = SparkSession.builder().appName("datasource-dws").getOrCreate();

        sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_dws USING JDBC OPTIONS
('url'='jdbc:postgresql://10.0.0.233:8000/postgres','dbtable'='test','user'='dbadmin','password'='**')");

        //*****************************SQL model***********************************
```



```
        //Insert data into the DLI data table
        sparkSession.sql("insert into dli_to_dws values(3,'Liu'),(4,'Xie')");

        //Read data from DLI data table
        sparkSession.sql("select * from dli_to_dws").show();

        //drop table
        sparkSession.sql("drop table dli_to_dws");

        sparkSession.close();
    }
}
```

4.3.4 对接 HBase

4.3.4.1 MRS 配置

DLI 跨源连接中配置 MRS 主机信息

1. 在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。
2. 对接MRS HBase需要在DLI队列的host文件中添加MRS集群节点的/etc/hosts信息。

   详细操作请参考《数据湖探索用户指南》中的“**修改主机信息**”章节描述。

开启 Kerberos 认证时的相关配置文件

1. 参考《**从零开始使用Kerberos认证集群**》中的“创建安全集群并登录其Manager”章节创建Kerberos认证集群。参考“创建角色和用户”章节添加用户并赋权。
2. 参考《**使用HBase客户端**》使用1中创建的用户认证登录。“人机”用户第一次登录时需修改密码。
3. 登录Manager界面，选择“系统 > 权限 > 用户”，选择新建用户，选择“更多 > 下载认证凭据”，保存后解压得到用户的keytab文件与krb5.conf文件。

创建 MRS HBase 表

创建DLI表关联MRS HBase表之前确保HBase的表是存在的。以样例代码为例，具体的流程是：

1. 远程登录ECS，通过hbase shell命令查看表信息。其中，“hbtest”是要查询的表名。

   ```
   describe 'hbtest'
   ```

2. （可选）如果不存在对应的HBase表，可以创建该表，具体的命令是：

   ```
   create ‘hbtest’, ‘info’, ‘detail’
   ```

   其中，“hbtest”是表名，其余为列族名。
3. 配置好连接信息。“TableName”对应HBase表的表名，“Rowkey”和“Cols”请参考《**创建DLI表关联HBase**》进行配置。

4.3.4.2 scala 样例代码

开发说明

支持对接CloudTable的HBase和MRS的HBase。

- 前提条件

  在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 构造依赖信息，创建SparkSession

  a. 导入依赖

  涉及到的mvn依赖库

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  import相关依赖包

```
import scala.collection.mutable
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._
```

  b. 创建会话。

```
val sparkSession = SparkSession.builder().getOrCreate()
```

  c. 创建DLI跨源访问 HBase的关联表。

  - 如果对接的HBase集群**未开启Kerberos认证**，则样例代码参考如下。

```
sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING,
'booleanf' BOOLEAN,
      'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase
OPTIONS (
   'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
             cloudtable-cf82-zk2-weBklrjI.cloudtable.com:2181,
             cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
   'TableName'='table_DupRowkey1',
   'RowKey'='id:5,location:6,city:7',

'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,d
oublef:CF1.doublef')"
)
```

  - 如果对接的HBase集群**开启了Kerberos认证**，则样例代码参考如下。

```
sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING,
'booleanf' BOOLEAN,
      'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase
OPTIONS (
   'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
             cloudtable-cf82-zk2-weBklrjI.cloudtable.com:2181,
             cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
   'TableName'='table_DupRowkey1',
   'RowKey'='id:5,location:6,city:7',

'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,d
oublef:CF1.doublef',
   'krb5conf'='./krb5.conf',
   'keytab' = './user.keytab',
   'principal' = 'krbtest')")
```

表 4-15 创建表参数

参数	说明
ZKHost	HBase集群的ZK连接地址。 获取ZK连接地址需要先创建跨源连接。具体操作请参考《数据湖探索用户指南》。 • 访问CloudTable集群，填写ZK连接地址（内网）。 • 访问MRS集群，填写ZK所在节点IP与ZK对外端口，格式为："ZK_IP1:ZK_PORT1,ZK_IP2:ZK_PORT2"。 **说明** 访问MRS集群，只支持创建增强型跨源连接并且需要配置主机信息，管理控制台操作请参考《数据湖探索用户指南》中的“增强型跨源连接”，相关API信息请参考**创建增强型跨源连接**。
RowKey	指定作为rowkey的dli关联表字段，支持单rowkey与组合rowkey。单rowkey支持数值与String类型，不需要指定长度。组合rowkey仅支持String类型定长数据，格式为：属性名1:长度,属性名2:长度。
Cols	定义dli表字段和ct表字段之间的对应关系；其中，“:”前放dli表字段，冒号后放ct表信息，用“.”分隔ct表的列族和列名。 例如：“dli表字段1:ct表.ct表字段1, dli表字段2:ct表.ct表字段2, dli表字段3:ct表.ct表字段3”。
krb5conf	开启Kerberos认证后的krb5.conf文件路径，格式为'./krb5.conf'。具体详情参考**开启Kerberos认证时的相关配置文件**。
keytab	开启Kerberos认证后的keytab文件路径，格式为'./user.keytab'。具体详情参考**开启Kerberos认证时的相关配置文件**。
principal	开启Kerberos认证后创建的用户名。

通过 SQL API 访问数据源

1. 插入数据
```
sparkSession.sql("insert into test_hbase values('12345','abc','guiyang',false,null,3,23,2.3,2.34)")
```
2. 查询数据
```
sparkSession.sql("select * from test_hbase").show ()
```
返回结果：

通过 DataFrame API 访问数据源

1. 构造schema
```
val attrId = new StructField("id",StringType)
val location = new StructField("location",StringType)
val city = new StructField("city",StringType)
val booleanf = new StructField("booleanf",BooleanType)
val shortf = new StructField("shortf",ShortType)
val intf = new StructField("intf",IntegerType)
val longf = new StructField("longf",LongType)
val floatf = new StructField("floatf",FloatType)
```

```
val doublef = new StructField("doublef",DoubleType)
val attrs = Array(attrId, location,city,booleanf,shortf,intf,longf,floatf,doublef)
```

2. 根据schema的类型构造数据

```
val mutableRow: Seq[Any] = Seq("12345","abc","city1",false,null,3,23,2.3,2.34)
val rddData: RDD[Row] = sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)), 1)
```

3. 导入数据到HBase

```
sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("test_hbase")
```

4. 读取HBase上的数据

```
val map = new mutable.HashMap[String, String]()
map("TableName") = "table_DupRowkey1"
map("RowKey") = "id:5,location:6,city:7"
map("Cols") =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doubl
ef"
map("ZKHost")="cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
            cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
            cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181"
sparkSession.read.schema(new StructType(attrs)).format("hbase").options(map.toMap).load().show()
```

返回结果：

```
+-----+--------+-------+--------+------+----+-----+------+-------+\n
|   id|location|   city|booleanf|shortf|intf|longf|floatf|doublef|\n
+-----+--------+-------+--------+------+----+-----+------+-------+\n
|12345|  huawei|guiyang|   false|  null|   3|   23|   2.3|   2.34|\n
|abcde|  abcdef|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
|check|  abcdef|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
\n+-----+--------+-------+--------+------+----+-----+------+-------+\n\n
```

提交 Spark 作业

1. 将写好的代码生成jar包，上传至OBS桶中。
2. 如果MRS集群开启了Kerberos认证，创建Spark作业时需要将krb5.conf和user.keytab文件添加到作业的其他依赖文件中，未开启Kerberos认证该步骤忽略。如图4-38所示：

图 4-38 添加依赖文件

3. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.hbase。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- 通过SQL API访问
  - **未开启Kerberos认证**样例代码

```
import org.apache.spark.sql.SparkSession

object Test_SparkSql_HBase {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    /**
     * Create an association table for the DLI association Hbase table
     */
    sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING,
'booleanf' BOOLEAN,
        'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase
OPTIONS (
    'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
              cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
              cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
    'TableName'='table_DupRowkey1',
    'RowKey'='id:5,location:6,city:7',
    'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,
        longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doublef')")

    //****************************SQL model**********************************
    sparkSession.sql("insert into test_hbase values('12345','abc','city1',false,null,3,23,2.3,2.34)")
    sparkSession.sql("select * from test_hbase").collect()

    sparkSession.close()
  }
}
```

  - **开启Kerberos认证**样例代码

```
import org.apache.spark.SparkFiles
import org.apache.spark.sql.SparkSession

import java.io.{File, FileInputStream, FileOutputStream}

object Test_SparkSql_HBase_Kerberos {

  def copyFile2(Input:String)(OutPut:String): Unit ={
    val fis = new FileInputStream(Input)
    val fos = new FileOutputStream(OutPut)
    val buf = new Array[Byte](1024)
    var len = 0
    while ({len = fis.read(buf);len} != -1){
      fos.write(buf,0,len)
    }
    fos.close()
    fis.close()
  }

  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()
    val sc = sparkSession.sparkContext
    sc.addFile("krb5.conf的obs地址")
    sc.addFile("user.keytab的obs地址")
    Thread.sleep(10)
```

```
    val krb5_startfile = new File(SparkFiles.get("krb5.conf"))
    val keytab_startfile = new File(SparkFiles.get("user.keytab"))
    val path_user = System.getProperty("user.dir")
    val keytab_endfile = new File(path_user + "/" + keytab_startfile.getName)
    val krb5_endfile = new File(path_user + "/" + krb5_startfile.getName)
    println(keytab_endfile)
    println(krb5_endfile)

    var krbinput = SparkFiles.get("krb5.conf")
    var krboutput = path_user+"/krb5.conf"
    copyFile2(krbinput)(krboutput)

    var keytabinput = SparkFiles.get("user.keytab")
    var keytaboutput = path_user+"/user.keytab"
    copyFile2(keytabinput)(keytaboutput)
    Thread.sleep(10)
    /**
     * Create an association table for the DLI association Hbase table
     */
    sparkSession.sql("CREATE TABLE testhbase(id string,booleanf boolean,shortf short,intf
int,longf long,floatf float,doublef double) " +
      "using hbase OPTIONS(" +
      "'ZKHost'='10.0.0.146:2181'," +
      "'TableName'='hbtest'," +
      "'RowKey'='id:100'," +

"'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF2.longf,floatf:CF1.floatf,doubl
ef:CF2.doublef'," +
      "'krb5conf'='" + path_user + "/krb5.conf'," +
      "'keytab'='" + path_user+ "/user.keytab'," +
      "'principal'='krbtest') ")

    //*****************************SQL model***********************************
    sparkSession.sql("insert into testhbase values('newtest',true,1,2,3,4,5)")
    val result = sparkSession.sql("select * from testhbase")
    result.show()

    sparkSession.close()
    }
}
```

- 通过DataFrame API访问

```
import scala.collection.mutable

import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._

object Test_SparkSql_HBase {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create an association table for the DLI association Hbase table
    sparkSession.sql("CREATE TABLE test_hbase('id' STRING, 'location' STRING, 'city' STRING, 'booleanf'
BOOLEAN,
        'shortf' SHORT, 'intf' INT, 'longf' LONG, 'floatf' FLOAT,'doublef' DOUBLE) using hbase OPTIONS (
    'ZKHost'='cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
              cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
              cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',
    'TableName'='table_DupRowkey1',
    'RowKey'='id:5,location:6,city:7',

'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.
doublef')")

    //*****************************DataFrame model***********************************
    // Setting schema
    val attrId = new StructField("id",StringType)
    val location = new StructField("location",StringType)
```

```
    val city = new StructField("city",StringType)
    val booleanf = new StructField("booleanf",BooleanType)
    val shortf = new StructField("shortf",ShortType)
    val intf = new StructField("intf",IntegerType)
    val longf = new StructField("longf",LongType)
    val floatf = new StructField("floatf",FloatType)
    val doublef = new StructField("doublef",DoubleType)
    val attrs = Array(attrId, location,city,booleanf,shortf,intf,longf,floatf,doublef)

    // Populate data according to the type of schema
    val mutableRow: Seq[Any] = Seq("12345","abc","city1",false,null,3,23,2.3,2.34)
    val rddData: RDD[Row] = sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)),
1)

    // Import the constructed data into Hbase
    sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("test_hbase")

    // Read data on Hbase
    val map = new mutable.HashMap[String, String]()
    map("TableName") = "table_DupRowkey1"
    map("RowKey") = "id:5,location:6,city:7"
    map("Cols") =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doubl
ef"
    map("ZKHost")="cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,
                   cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
                   cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181"
    sparkSession.read.schema(new
StructType(attrs)).format("hbase").options(map.toMap).load().collect()

    sparkSession.close()
  }
}
```

4.3.4.3 pyspark 样例代码

开发说明

支持对接CloudTable的HBase和MRS的HBase。

- 前提条件

  在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现详解

  a. import相关依赖包

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType, BooleanType,
  ShortType, LongType, FloatType, DoubleType
  from pyspark.sql import SparkSession
  ```

  b. 创建会话

  ```
  sparkSession = SparkSession.builder.appName("datasource-hbase").getOrCreate()
  ```

- 通过SQL API访问

  a. 创建DLI跨源访问HBase的关联表

  - 如果对接的HBase集群**未开启Kerberos认证**，样例代码参考如下。

    ```
    sparkSession.sql(
      "CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase
    ```

```
OPTIONS (\
   'ZKHost' = '192.168.0.189:2181',\
   'TableName' = 'hbtest',\
   'RowKey' = 'id:5',\
   'Cols' = 'location:info.location,city:detail.city')")
```

- 如果对接的HBase集群**开启了Kerberos认证**，样例代码参考如下。

```
sparkSession.sql(
   "CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase
OPTIONS (\
   'ZKHost' = '192.168.0.189:2181',\
   'TableName' = 'hbtest',\
   'RowKey' = 'id:5',\
   'Cols' = 'location:info.location,city:detail.city',\
   'krb5conf' = './krb5.conf',\
   'keytab'='./user.keytab',\
   'principal' ='krbtest')")
```

与未开启kerberos认证相比，开启了kerberos认证需要多设置三个参数，如表4-16所示。

表 4-16 参数说明

参数名称与参数值	参数说明
'krb5conf' = './krb5.conf'	krb5.conf的地址。
'keytab'='./user.keytab'	Keytab的地址。
'principal' ='krbtest'	认证用户名。

krb5.conf和keytab文件获取请具体参考**开启Kerberos认证时的相关配置文件**操作说明。

说明

表参数详情可参考**表4-15**。

b. 导入数据到HBase

```
sparkSession.sql("insert into testhbase values('95274','abc','Jinan')")
```

c. 读取HBase上的数据

```
sparkSession.sql("select * from testhbase").show()
```

- 通过DataFrame API访问

a. 创建DLI跨源访问HBase的关联表

```
sparkSession.sql(\
  "CREATE TABLE test_hbase(id STRING, location STRING, city STRING, booleanf BOOLEAN,
shortf SHORT, intf INT, longf LONG,
     floatf FLOAT, doublef DOUBLE) using hbase OPTIONS (\
   'ZKHost' = 'cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,\
               cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,\
               cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',\
   'TableName' = 'table_DupRowkey1',\
   'RowKey' = 'id:5,location:6,city:7',\
   'Cols' = 'booleanf:CF1.booleanf, shortf:CF1.shortf, intf:CF1.intf, \  longf:CF1.longf,
floatf:CF1.floatf, doublef:CF1.doublef')")
```

说明

- ZKHost、RowKey、Cols三个参数详情讲解可参考**表4-15**。
- TableName：CloudTable中的表名，在保存时如果没有表名，系统会自动创建。

b. 构造schema

```
schema = StructType([StructField("id", StringType()),\
                     StructField("location", StringType()),\
                     StructField("city", StringType()),\
                     StructField("booleanf", BooleanType()),\
                     StructField("shortf", ShortType()),\
                     StructField("intf", IntegerType()),\
                     StructField("longf", LongType()),\
                     StructField("floatf", FloatType()),\
                     StructField("doublef", DoubleType())])
```

c. 设置数据

```
dataList = sparkSession.sparkContext.parallelize([("11111", "aaa", "aaa", False, 4, 3, 23, 2.3, 2.34)])
```

d. 创建DataFrame

```
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

e. 导入数据到HBase

```
dataFrame.write.insertInto("test_hbase")
```

f. 读取HBase上的数据

```
// Set cross-source connection parameters
TableName = "table_DupRowkey1"
RowKey = "id:5,location:6,city:7"
Cols =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doublef"
ZKHost = "cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
        cloudtable-cf82-zk1- WY09px9l.cloudtable.com:2181"

// select
jdbcDF = sparkSession.read.schema(schema)\
                .format("hbase")\
                .option("ZKHost",ZKHost)\
                .option("TableName",TableName)\
                .option("RowKey",RowKey)\
                .option("Cols",Cols)\
                .load()
jdbcDF.filter("id = '12333' or id='11111'").show()
```

说明

id、location、city：限定了长度，插入数据时须按长度给定数据值，否则查询时会发生编码格式错误。

g. 操作结果；

```
+-----+--------+-------+--------+------+----+-----+------+-------+
|   id|location|   city|booleanf|shortf|intf|longf|floatf|doublef|
+-----+--------+-------+--------+------+----+-----+------+-------+
|11111|  beijin|beijing|   false|     4|   3|   23|   2.3|   2.34|
|12333|  nanjin|nanjing|   false|  null|   3|   23|   2.3|   2.34|
+-----+--------+-------+--------+------+----+-----+------+-------+
```

- 提交Spark作业

a. 将写好的python代码文件上传至OBS桶中。

b. 如果MRS集群开启了Kerberos认证，创建Spark作业时需要将krb5.conf和user.keytab文件添加到作业的其他依赖文件中，未开启Kerberos认证该步骤忽略。如**图4-39**所示：

图 4-39 添加依赖文件

c. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.hbase。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- 通过SQL API访问MRS HBase
  - **未开启kerberos认证**样例代码

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, BooleanType, ShortType, LongType, FloatType, DoubleType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-hbase").getOrCreate()

  sparkSession.sql(
    "CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase OPTIONS (\
    'ZKHost' = '192.168.0.189:2181',\
    'TableName' = 'hbtest',\
    'RowKey' = 'id:5',\
    'Cols' = 'location:info.location,city:detail.city')")


  sparkSession.sql("insert into testhbase values('95274','abc','Jinan')")

  sparkSession.sql("select * from testhbase").show()
  # close session
  sparkSession.stop()
```

  - **开启kerberos认证**样例代码

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark import SparkFiles
from pyspark.sql import SparkSession
import shutil
import time
import os

if __name__ == "__main__":
```

```
    # Create a SparkSession session.
    sparkSession =
SparkSession.builder.appName("Test_HBase_SparkSql_Kerberos").getOrCreate()
    sc = sparkSession.sparkContext
    time.sleep(10)

    krb5_startfile = SparkFiles.get("krb5.conf")
    keytab_startfile = SparkFiles.get("user.keytab")
    path_user = os.getcwd()
    krb5_endfile = path_user + "/" + "krb5.conf"
    keytab_endfile = path_user + "/" + "user.keytab"
    shutil.copy(krb5_startfile, krb5_endfile)
    shutil.copy(keytab_startfile, keytab_endfile)
    time.sleep(20)

    sparkSession.sql(
      "CREATE TABLE testhbase(id string,booleanf boolean,shortf short,intf int,longf long,floatf
float,doublef double) " +
      "using hbase OPTIONS(" +
      "'ZKHost'='10.0.0.146:2181'," +
      "'TableName'='hbtest'," +
      "'RowKey'='id:100'," +

"'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF2.longf,floatf:CF1.floatf,doubl
ef:CF2.doublef'," +
      "'krb5conf'='" + path_user + "/krb5.conf'," +
      "'keytab'='" + path_user+ "/user.keytab'," +
      "'principal'='krbtest') ")

      sparkSession.sql("insert into testhbase values('95274','abc','Jinan')")

    sparkSession.sql("select * from testhbase").show()
    # close session
    sparkSession.stop()
```

- 通过DataFrame API访问HBase

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, BooleanType,
ShortType, LongType, FloatType, DoubleType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-hbase").getOrCreate()

  # Createa data table for DLI-associated ct
  sparkSession.sql(\
   "CREATE TABLE test_hbase(id STRING, location STRING, city STRING, booleanf BOOLEAN, shortf
SHORT, intf INT, longf LONG,floatf FLOAT,doublef DOUBLE) using hbase OPTIONS ( \
    'ZKHost' = 'cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,\
               cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,\
               cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181',\
    'TableName' = 'table_DupRowkey1',\
    'RowKey' = 'id:5,location:6,city:7',\
    'Cols' =
'booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.double
f')")

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([("11111", "aaa", "aaa", False, 4, 3, 23, 2.3, 2.34)])

  # Setting schema
  schema = StructType([StructField("id", StringType()),
                       StructField("location", StringType()),
                       StructField("city", StringType()),
                       StructField("booleanf", BooleanType()),
                       StructField("shortf", ShortType()),
                       StructField("intf", IntegerType()),
                       StructField("longf", LongType()),
```

```
                    StructField("floatf", FloatType()),
                    StructField("doublef", DoubleType())])

# Create a DataFrame from RDD and schema
dataFrame = sparkSession.createDataFrame(dataList, schema)

# Write data to the cloudtable-hbase
dataFrame.write.insertInto("test_hbase")

# Set cross-source connection parameters
TableName = "table_DupRowkey1"
RowKey = "id:5,location:6,city:7"
Cols =
"booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF1.longf,floatf:CF1.floatf,doublef:CF1.doublef"
ZKHost = "cloudtable-cf82-zk3-pa6HnHpf.cloudtable.com:2181,cloudtable-cf82-zk2-weBkIrjI.cloudtable.com:2181,
        cloudtable-cf82-zk1-WY09px9l.cloudtable.com:2181"
# Read data on CloudTable-HBase
jdbcDF = sparkSession.read.schema(schema)\
                    .format("hbase")\
                    .option("ZKHost", ZKHost)\
                    .option("TableName",TableName)\
                    .option("RowKey", RowKey)\
                    .option("Cols", Cols)\
                    .load()
jdbcDF.filter("id = '12333' or id='11111'").show()

# close session
sparkSession.stop()
```

4.3.4.4 java 样例代码

开发说明

本样例只适用于MRS的HBase。

- 前提条件

  在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现

  a. 导入依赖

  - 涉及到的mvn依赖库

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

  - import相关依赖包

    ```
    import org.apache.spark.sql.SparkSession;
    ```

  b. 创建会话

  ```
  parkSession = SparkSession.builder().appName("datasource-HBase-MRS").getOrCreate();
  ```

- 通过SQL API 访问

  - 未开启Kerberos认证

i. 创建DLI跨源访问MRS HBase的关联表，填写连接参数。

```
sparkSession.sql("CREATE TABLE testhbase(id STRING, location STRING, city STRING)
using hbase
OPTIONS('ZKHost'='10.0.0.63:2181','TableName'='hbtest','RowKey'='id:5','Cols'='location:info.location,city:detail.city') ");
```

ii. 插入数据

```
sparkSession.sql("insert into testhbase values('12345','abc','xxx')");
```

iii. 查询数据

```
sparkSession.sql("select * from testhbase").show();
```

插入数据后：

```
+-----+--------+--------+
|   id|location|    city|
+-----+--------+--------+
|12342|Shanghai|Shanghai|
|12345|     abc|  guiyang|
+-----+--------+--------+
```

- 开启Kerberos认证

i. 创建DLI跨源访问MRS HBase的关联表，填写连接参数。

```
sparkSession.sql("CREATE TABLE testhbase(id STRING, location STRING, city STRING)
using hbase
OPTIONS('ZKHost'='10.0.0.63:2181','TableName'='hbtest','RowKey'='id:5','Cols'='location:info.location,city:detail.city,'krb5conf'='./krb5.conf','keytab'='./user.keytab','principal'='krbtest') ");
```

与未开启kerberos认证相比，开启了kerberos认证需要多设置三个参数，如表4-17所示。

表 4-17 参数说明

参数名称与参数值	参数说明
'krb5conf' = './krb5.conf'	krb5.conf的地址。
'keytab'='./user.keytab'	Keytab的地址。
'principal' ='krbtest'	认证用户名。

krb5.conf和keytab文件获取请具体参考开启Kerberos认证时的相关配置文件操作说明。

ii. 插入数据

```
sparkSession.sql("insert into testhbase values('95274','abc','Hongkong')");
```

iii. 查询数据

```
sparkSession.sql("select * from testhbase").show();
```

- 提交Spark作业

a. 将写好的代码文件生成jar包，上传至OBS桶中。

b. 如果MRS集群开启了Kerberos认证，创建Spark作业时需要将krb5.conf和user.keytab文件添加到作业的依赖文件中，未开启Kerberos认证该步骤忽略。如图4-40所示：

图 4-40 添加依赖文件

c. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.hbase。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/hbase/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- 通过SQL API访问
  - **未开启Kerberos**完整代码示例

```
import org.apache.spark.sql.SparkSession;

public class java_mrs_hbase {

    public static void main(String[] args) {
        //create a SparkSession session
        SparkSession sparkSession = SparkSession.builder().appName("datasource-HBase-MRS").getOrCreate();

        sparkSession.sql("CREATE TABLE testhbase(id STRING, location STRING, city STRING) using hbase OPTIONS('ZKHost'='10.0.0.63:2181','TableName'='hbtest','RowKey'='id:5','Cols'='location:info.location,city:detail.city') ");

        //*****************************SQL model***********************************
        sparkSession.sql("insert into testhbase values('95274','abc','Hongkong')");
        sparkSession.sql("select * from testhbase").show();

        sparkSession.close();
    }
}
```

  - **开启Kerberos**完整代码示例

```
import org.apache.spark.SparkContext;
import org.apache.spark.SparkFiles;
import org.apache.spark.sql.SparkSession;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Test_HBase_SparkSql_Kerberos {
```

```
    private static void copyFile(File src,File dst) throws IOException {
        InputStream input  = null;
        OutputStream output = null;
        try {
            input = new FileInputStream(src);
            output = new FileOutputStream(dst);
            byte[] buf = new byte[1024];
            int bytesRead;
            while ((bytesRead = input.read(buf)) > 0) {
                output.write(buf, 0, bytesRead);
            }
        } finally {
            input.close();
            output.close();
        }
    }

    public static void main(String[] args) throws InterruptedException, IOException {
        SparkSession sparkSession =
SparkSession.builder().appName("Test_HBase_SparkSql_Kerberos").getOrCreate();
        SparkContext sc = sparkSession.sparkContext();
        sc.addFile("obs://xietest1/lzq/krb5.conf");
        sc.addFile("obs://xietest1/lzq/user.keytab");
        Thread.sleep(20);

        File krb5_startfile = new File(SparkFiles.get("krb5.conf"));
        File keytab_startfile = new File(SparkFiles.get("user.keytab"));
        String path_user = System.getProperty("user.dir");
        File keytab_endfile = new File(path_user + "/" + keytab_startfile.getName());
        File krb5_endfile = new File(path_user + "/" + krb5_startfile.getName());
        copyFile(krb5_startfile,krb5_endfile);
        copyFile(keytab_startfile,keytab_endfile);
        Thread.sleep(20);

        /**
         * Create an association table for the DLI association Hbase table
         */
        sparkSession.sql("CREATE TABLE testhbase(id string,booleanf boolean,shortf short,intf
int,longf long,floatf float,doublef double) " +
                "using hbase OPTIONS(" +
                "'ZKHost'='10.0.0.146:2181'," +
                "'TableName'='hbtest'," +
                "'RowKey'='id:100'," +

"'Cols'='booleanf:CF1.booleanf,shortf:CF1.shortf,intf:CF1.intf,longf:CF2.longf,floatf:CF1.floatf,doubl
ef:CF2.doublef'," +
                "'krb5conf'='" + path_user + "/krb5.conf'," +
                "'keytab'='" + path_user+ "/user.keytab'," +
                "'principal'='krbtest') ");

        //*****************************SQL model***********************************
        sparkSession.sql("insert into testhbase values('newtest',true,1,2,3,4,5)");
        sparkSession.sql("select * from testhbase").show();
        sparkSession.close();
    }
}
```

4.3.4.5 故障处理

问题 1：运行 Spark 作业，作业运行失败，作业日志中提示 java server connection 或 container 启动失败

- 问题现象

  运行Spark作业，作业运行失败，作业日志中提示java server connection或container启动失败。

- 解决方案

  确认是否已修改跨源连接的主机信息，如果没有，请参考**DLI跨源连接中配置MRS主机信息**修改主机信息。重新创建和提交Spark作业。

问题 2：运行 Spark 作业，作业运行失败，作业日志中提示 KrbException: Message stream modified (41)

- 问题现象

  运行Spark作业，作业运行失败，作业日志中提示KrbException: Message stream modified (41)

- 解决方案

  编辑“krb5.conf” 配置文件，将文件中所有“renew_lifetime = xxx”配置删除。重新创建和提交Spark作业。

4.3.5 对接 OpenTSDB

4.3.5.1 scala 样例代码

开发说明

支持对接CloudTable的OpenTSDB和MRS的OpenTSDB。

- 前提条件

  在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 构造依赖信息，创建SparkSession

  a. 导入依赖。

  涉及到mvn依赖

  ```
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.3.2</version>
  </dependency>
  ```

  import相关依赖包

  ```
  import scala.collection.mutable
  import org.apache.spark.sql.{Row, SparkSession}
  import org.apache.spark.rdd.RDD
  import org.apache.spark.sql.types._
  ```

  b. 创建会话。

  ```
  val sparkSession = SparkSession.builder().getOrCreate()
  ```

  c. 创建DLI关联跨源访问 OpenTSDB的关联表。

  ```
  sparkSession.sql("create table opentsdb_test using opentsdb options(
      'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
          'metric'='ctopentsdb',
      'tags'='city,location')")
  ```

表 4-18 创建表参数

参数	说明
host	OpenTSDB连接地址。 ● 访问CloudTable OpenTSDB，填写OpenTSDB链接地址，具体可以登录CloudTable控制台，单击“集群模式 > *集群名称*”，在集群信息获取OpenTSDB链接地址。 ● 访问MRS OpenTSDB，若使用增强型跨源连接，填写OpenTSDB所在节点IP与端口，格式为"IP:PORT"，OpenTSDB存在多个节点时，用分号隔开，获取方式请参考“图 MRS集群OpenTSDB IP信息”和“图 MRS集群OpenTSDB 端口信息”。若使用经典型跨源，填写经典型跨源返回的连接地址，管理控制台操作请参考《数据湖探索用户指南》。
metric	所创建的dli表对应的OpenTSDB中的指标名称。
tags	metric对应的标签，用于归类、过滤、快速检索等操作，可以是1到8个，以“，”分隔，包括对应metric下的所有tagk的值。

- 通过SQL API访问
  a. 插入数据
  ```
  sparkSession.sql("insert into opentsdb_test values('futian', 'abc', '1970-01-02 18:17:36', 30.0)")
  ```
  b. 查询数据
  ```
  sparkSession.sql("select * from opentsdb_test").show()
  ```
  返回结果：
  ```
  +-------+--------+-------------------+-----+\n
  |   city|location|          timestamp|value|\n
  +-------+--------+-------------------+-----+\n
  | futian|  huawei|1970-01-02 18:17:36| 30.0|\n
  |beijing|  huawei|1970-01-02 18:17:36| 30.0|\n
  +-------+--------+-------------------+-----+\n\n
  ```
- 通过DataFrame API访问
  a. 构造schema
  ```
  val attrTag1Location = new StructField("location", StringType)
  val attrTag2Name = new StructField("name", StringType)
  val attrTimestamp = new StructField("timestamp", LongType)
  val attrValue = new StructField("value", DoubleType)
  val attrs = Array(attrTag1Location, attrTag2Name, attrTimestamp, attrValue)
  ```
  b. 根据schema的类型构造数据
  ```
  val mutableRow: Seq[Any] = Seq("aaa", "abc", 123456L, 30.0)
  val rddData: RDD[Row] =
  sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)), 1)
  ```
  c. 导入数据到OpenTSDB
  ```
  sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("opentsdb_test")
  ```
  d. 读取OpenTSDB上的数据
  ```
  val map = new mutable.HashMap[String, String]()
  map("metric") = "ctopentsdb"
  map("tags") = "city,location"
  map("Host") = "opentsdb-3xcl8dir15m58z3.cloudtable.com:4242"
  sparkSession.read.format("opentsdb").options(map.toMap).load().show()
  ```

返回结果：

```
+-------+--------+-------------------+-----+\n
|   city|location|          timestamp|value|\n
+-------+--------+-------------------+-----+\n
| futian|  huawei|1970-01-02 18:17:36| 30.0|\n
|beijing|  huawei|1970-01-02 18:17:36| 30.0|\n
+-------+--------+-------------------+-----+\n\n
```

- 提交Spark作业
  a. 将写好的代码生成jar包，上传至OBS桶中。
  b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  说明

  - 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.opentsdb。
  - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- 通过SQL API访问

```
import org.apache.spark.sql.SparkSession

object Test_OpenTSDB_CT {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a data table for DLI association OpenTSDB
    sparkSession.sql("create table opentsdb_test using opentsdb options(
    'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
    'metric'='ctopentsdb',
    'tags'='city,location')")

    //****************************SQL module*********************************
    sparkSession.sql("insert into opentsdb_test values('futian', 'abc', '1970-01-02 18:17:36', 30.0)")
    sparkSession.sql("select * from opentsdb_test").show()

    sparkSession.close()
  }
}
```

- 通过DataFrame API访问

```
import scala.collection.mutable
import org.apache.spark.sql.{Row, SparkSession}
```

```
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._

object Test_OpenTSDB_CT {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a data table for DLI association OpenTSDB
    sparkSession.sql("create table opentsdb_test using opentsdb options(
    'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
    'metric'='ctopentsdb',
    'tags'='city,location')")

    //*****************************DataFrame model***********************************
    // Setting schema
    val attrTag1Location = new StructField("location", StringType)
    val attrTag2Name = new StructField("name", StringType)
    val attrTimestamp = new StructField("timestamp", LongType)
    val attrValue = new StructField("value", DoubleType)
    val attrs = Array(attrTag1Location, attrTag2Name, attrTimestamp,attrValue)

    // Populate data according to the type of schema
    val mutableRow: Seq[Any] = Seq("aaa", "abc", 123456L, 30.0)
    val rddData: RDD[Row] = sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)),
1)

    //Import the constructed data into OpenTSDB
    sparkSession.createDataFrame(rddData, new StructType(attrs)).write.insertInto("opentsdb_test")

    //Read data on OpenTSDB
    val map = new mutable.HashMap[String, String]()
    map("metric") = "ctopentsdb"
    map("tags") = "city,location"
    map("Host") = "opentsdb-3xcl8dir15m58z3.cloudtable.com:4242"
    sparkSession.read.format("opentsdb").options(map.toMap).load().show()

    sparkSession.close()
  }
}
```

4.3.5.2 pyspark 样例代码

开发说明

支持对接CloudTable的OpenTSDB和MRS的OpenTSDB。

- 前提条件

  在DLI管理控制台上已完成创建跨源连接。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现详解

  a. import相关依赖包

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, StringType, LongType, DoubleType
  from pyspark.sql import SparkSession
  ```

  b. 创建会话

  ```
  sparkSession = SparkSession.builder.appName("datasource-opentsdb").getOrCreate()
  ```

c. 创建DLI跨源访问 OpenTSDB的关联表

```
sparkSession.sql("create table opentsdb_test using opentsdb options(
  'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',
  'metric'='ct_opentsdb',
  'tags'='city,location')")
```

说明

Host、metric、tags三个参数详情讲解可参考**表4-18**。

- 通过SQL API访问

a. 插入数据

```
sparkSession.sql("insert into opentsdb_test values('aaa', 'abc', '2021-06-30 18:00:00', 30.0)")
```

b. 查询数据

```
result = sparkSession.sql("SELECT * FROM opentsdb_test")
```

- 通过DataFrame API 访问

a. 构造schema

```
schema = StructType([StructField("location", StringType()),\
                     StructField("name", StringType()), \
                     StructField("timestamp", LongType()),\
                     StructField("value", DoubleType())])
```

b. 设置数据

```
dataList = sparkSession.sparkContext.parallelize([("aaa", "abc", 123456L, 30.0)])
```

c. 创建DataFrame

```
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

d. 导入数据到OpenTSDB

```
dataFrame.write.insertInto("opentsdb_test")
```

e. 读取OpenTSDB上的数据

```
jdbdDF = sparkSession.read
    .format("opentsdb")\
    .option("Host","opentsdb-3xcl8dir15m58z3.cloudtable.com:4242")\
    .option("metric","ctopentsdb")\
    .option("tags","city,location")\
    .load()
jdbdDF.show()
```

f. 操作结果

```
+-------+--------+-------------------+-----+
|   city|location|          timestamp|value|
+-------+--------+-------------------+-----+
| futian|  huawei|1970-01-02 18:17:36| 30.0|
|nanjing|     tom|2019-08-28 00:00:00| 30.0|
|beijing|  huawei|1970-01-02 18:17:36| 30.0|
+-------+--------+-------------------+-----+
```

- 提交Spark作业

a. 将写好的python代码文件上传至OBS桶中。

b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.opentsdb。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- 通过SQL API访问MRS的OpenTSDB

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, StringType, LongType, DoubleType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-opentsdb").getOrCreate()


  # Create a DLI cross-source association opentsdb data table
  sparkSession.sql(\
    "create table opentsdb_test using opentsdb options(\
    'Host'='10.0.0.171:4242',\
    'metric'='cts_opentsdb',\
    'tags'='city,location')")

  sparkSession.sql("insert into opentsdb_test values('aaa', 'abc', '2021-06-30 18:00:00', 30.0)")

  result = sparkSession.sql("SELECT * FROM opentsdb_test")
  result.show()

  # close session
  sparkSession.stop()
```

- 通过DataFrame API访问OpenTSDB

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, StringType, LongType, DoubleType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-opentsdb").getOrCreate()

  # Create a DLI cross-source association opentsdb data table
  sparkSession.sql(
    "create table opentsdb_test using opentsdb options(\
    'Host'='opentsdb-3xcl8dir15m58z3.cloudtable.com:4242',\
    'metric'='ct_opentsdb',\
    'tags'='city,location')")

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([("aaa", "abc", 123456L, 30.0)])

  # Setting schema
  schema = StructType([StructField("location", StringType()),\
```

```
                StructField("name", StringType()),\
                StructField("timestamp", LongType()),\
                StructField("value", DoubleType())])

# Create a DataFrame from RDD and schema
dataFrame = sparkSession.createDataFrame(dataList, schema)

# Set cross-source connection parameters
metric = "ctopentsdb"
tags = "city,location"
Host = "opentsdb-3xcl8dir15m58z3.cloudtable.com:4242"

# Write data to the cloudtable-opentsdb
dataFrame.write.insertInto("opentsdb_test")
# ******* Opentsdb does not currently implement the ctas method to save data, so the save() method cannot be used.*******
# dataFrame.write.format("opentsdb").option("Host", Host).option("metric", metric).option("tags", tags).mode("Overwrite").save()

# Read data on CloudTable-OpenTSDB
jdbdDF = sparkSession.read\
    .format("opentsdb")\
    .option("Host",Host)\
    .option("metric",metric)\
    .option("tags",tags)\
    .load()
jdbdDF.show()

# close session
sparkSession.stop()
```

4.3.5.3 java 样例代码

开发说明

本样例只适用于MRS的OpenTSDB。

- 前提条件

  在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现

  a. 导入依赖

  - 涉及到的mvn依赖库

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

  - import相关依赖包

    ```
    import org.apache.spark.sql.SparkSession;
    ```

  b. 创建会话

  ```
  sparkSession = SparkSession.builder().appName("datasource-opentsdb").getOrCreate();
  ```

- 通过SQL API 访问

  – 创建DLI跨源访问MRS OpenTSDB的关联表，填写连接参数。

```
sparkSession.sql("create table opentsdb_new_test using opentsdb
options('Host'='10.0.0.171:4242','metric'='ctopentsdb','tags'='city,location')");
```

说明

Host、metric、tags三个参数详情讲解可参考**表4-18**。

- 插入数据

```
sparkSession.sql("insert into opentsdb_new_test values('Penglai', 'abc', '2021-06-30 18:00:00',
30.0)");
```

- 查询数据

```
sparkSession.sql("select * from opentsdb_new_test").show();
```

插入数据后：

```
+-------+--------+-------------------+-----+
|   city|location|          timestamp|value|
+-------+--------+-------------------+-----+
|Penglai|     abc|2021-06-30 18:00:00| 30.0|
+-------+--------+-------------------+-----+
```

- 提交Spark作业
  a. 将写好的代码文件生成jar包，上传至OBS桶中。
  b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.opentsdb。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/opentsdb/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- Maven依赖

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-sql_2.11</artifactId>
        <version>2.3.2</version>
</dependency>
```

- 通过SQL API访问

```
import org.apache.spark.sql.SparkSession;

public class java_mrs_opentsdb {

    private static SparkSession sparkSession = null;

    public static void main(String[] args) {
        //create a SparkSession session
        sparkSession = SparkSession.builder().appName("datasource-opentsdb").getOrCreate();

        sparkSession.sql("create table opentsdb_new_test using opentsdb
```



```
options('Host'='10.0.0.171:4242','metric'='ctopentsdb','tags'='city,location')");

        //*****************************SQL module*********************************
        sparkSession.sql("insert into opentsdb_new_test values('Penglai', 'abc', '2021-06-30 18:00:00',
30.0)");
        System.out.println("Penglai new timestamp");
        sparkSession.sql("select * from opentsdb_new_test").show();

       sparkSession.close();

    }
}
```

4.3.5.4 故障处理

运行 Spark 作业，作业运行失败，作业日志中提示 No respond 错误

- 问题现象

  运行Spark作业，作业运行失败，作业日志中提示No respond错误

- 解决方案

  重新创建Spark作业，创建作业时需要在“Spark参数（ --conf ）”中添加配置：“spark.sql.mrs.opentsdb.ssl.enabled=true”。

4.3.6 对接 RDS

4.3.6.1 scala 样例代码

开发说明

- 前提条件

  在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 构造依赖信息，创建SparkSession

  a. 导入依赖

  涉及到的mvn依赖库

  ```
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.3.2</version>
  </dependency>
  ```

  import相关依赖包

  ```
  import java.util.Properties
  import org.apache.spark.sql.{Row,SparkSession}
  import org.apache.spark.sql.SaveMode
  ```

  b. 创建会话。

  ```
  val sparkSession = SparkSession.builder().getOrCreate()
  ```

- 通过SQL API 访问

  a. 创建DLI跨源访问 rds的关联表，填写连接参数。

```
sparkSession.sql(
  "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (
     'url'='jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306',//根据实际url修改
     'dbtable'='test.customer',
     'user'='root',//根据实际user修改
     'password'='######',//根据实际password修改
     'driver'='com.mysql.jdbc.Driver')")
```

表 4-19 创建表参数

参数	说明
url	RDS的连接地址，需要先创建跨源连接，管理控制台操作请参考《 数据湖探索用户指南 》。 创建经典型跨源连接后，使用经典型跨源连接中返回的连接地址。 创建增强型跨源连接后，使用RDS提供的"内网域名"或者内网地址和数据库端口访问，MySQL格式为"协议头://内网IP:内网端口"，PostGre格式为"协议头://内网IP:内网端口/数据库名"。 例如："jdbc:mysql://192.168.0.193:3306"或者"jdbc:postgresql://192.168.0.193:3306/postgres"，获取方式请参考“图 RDS集群信息”。 **说明** 经典型跨源的连接地址默认格式为："协议头://访问地址:访问端口" 例如：jdbc:mysql://to-rds-1174405119-oLRHAGE7.datasource.com:3306 如果想要访问RDS的postgre集群，需要将连接地址中的协议头修改为"jdbc:postgresql"，并在连接地址最后加上"/数据库名"。 例如：jdbc:postgresql://to-rds-1174405119-oLRHAGE7.datasource.com:3306/postgreDB
dbtable	访问MySQL集群填写"数据库名.表名"，访问PostGre集群填写"模式名.表名"。 **说明** 如果数据库和表不存在，请先创建数据库和表，否则系统会报错并且运行失败。
user	RDS数据库用户名。
password	RDS数据库用户名对应密码。
driver	jdbc驱动类名，访问MySQL集群请填写："com.mysql.jdbc.Driver"，访问PostGre集群请填写："org.postgresql.Driver"。
partitionColumn	读取数据时，用于设置并发使用的数值型字段。 **说明** • “partitionColumn”，“lowerBound”，“upperBound”，“numPartitions”4个参数必须同时设置，不支持仅设置其中一部分 • 为了提升并发读取的性能，建议使用自增列。

参数	说明
lowerBound	partitionColumn设置的字段数据最小值，该值包含在返回结果中。
upperBound	partitionColumn设置的字段数据最大值，该值不包含在返回结果中。
numPartitions	读取数据时并发数。 **说明** 实际读取数据时，会根据lowerBound与upperBound，平均分配给每个task获取其中一部分的数据。例如： 'partitionColumn'='id', 'lowerBound'='0', 'upperBound'='100', 'numPartitions'='2' DLI中会起2个并发task，一个task执行id>=0 and id < 50，另一个task执行id >=50 and id < 100。
fetchsize	读取数据时，每一批次获取数据的记录数，默认值1000。设置越大性能越好，但占用内存越多，该值设置过大会有内存溢出的风险。
batchsize	写入数据时，每一批次写入数据的记录数，默认值1000。设置越大性能越好，但占用内存越多，该值设置过大会有内存溢出的风险。
truncate	执行overwrite时是否不删除原表，直接执行清空表操作，取值范围： • true • false 默认为'false'，即在执行overwrite操作时，先将原表删除再重新建表。
isolationLevel	事务隔离级别，取值范围： • NONE • READ_UNCOMMITTED • READ_COMMITTED • REPEATABLE_READ • SERIALIZABLE 默认值为'READ_UNCOMMITTED'。

图 4-41 RDS 集群信息

b. 插入数据

```
sparkSession.sql("insert into dli_to_rds values(1, 'John',24),(2, 'Bob',32)")
```

c. 查询数据

```
val dataFrame = sparkSession.sql("select * from dli_to_rds")
dataFrame.show()
```

插入数据前：

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  7|   chm| 13|
|  6|    qz| 13|
|  3|  mark| 20|
+---+------+---+
```

插入数据后：

```
+---+------+---+
| id|  name|age|
+---+------+---+
|  4|  kobe| 24|
|  6|    qz| 13|
|  7|   chm| 13|
|  3|  mark| 20|
|  1|   tom| 18|
|  2|  ammy| 18|
|  5|jordan| 22|
|  1|  John| 24|
|  2|   Bob| 32|
+---+------+---+
```

d. 删除关联表

```
sparkSession.sql("drop table dli_to_rds")
```

- 通过DataFrame API 访问

a. 连接参数配置

```
val url = "jdbc:mysql://to-rds-1174405057-EA1Kgo8H.datasource.com:3306"
val username = "root"
val password = "######"
val dbtable = "test.customer"
```

b. 创建DataFrame，添加数据，并重命名字段。

```
var dataFrame_1 = sparkSession.createDataFrame(List((8, "Jack_1", 18)))
val df = dataFrame_1.withColumnRenamed("_1", "id")
                    .withColumnRenamed("_2", "name")
                    .withColumnRenamed("_3", "age")
```

c. 导入数据到RDS。

```
df.write.format("jdbc")
  .option("url", url)
  .option("dbtable", dbtable)
  .option("user", username)
  .option("password", password)
  .option("driver", "com.mysql.jdbc.Driver")
  .mode(SaveMode.Append)
  .save()
```

说明

SaveMode 有四种保存类型：

- ErrorIfExis：如果已经存在数据，则抛出异常。
- Overwrite：如果已经存在数据，则覆盖原数据。
- Append：如果已经存在数据，则追加保存。
- Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

d. 读取RDS上的数据。

- 方式一：read.format()方法

```
val jdbcDF = sparkSession.read.format("jdbc")
            .option("url", url)
            .option("dbtable", dbtable)
            .option("user", username)
            .option("password", password)
            .option("driver", "org.postgresql.Driver")
            .load()
```

- 方式二：read.jdbc()方法

```
val properties = new Properties()
properties.put("user", username)
properties.put("password", password)
val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)
```

插入数据前：

```
+---+------+---+\n
| id|  name|age|\n
+---+------+---+\n
|  4|  kobe| 24|\n
|  3|  mark| 20|\n
|  7|   chm| 13|\n
|  6|    qz| 13|\n
|  1|   tom| 18|\n
|  2|  ammy| 18|\n
|  5|jordan| 22|\n
+---+------+---+\n
```

插入数据后：

```
+---+------+---+\n
| id|  name|age|\n
+---+------+---+\n
|  7|   chm| 13|\n
|  1|   tom| 18|\n
|  2|  ammy| 18|\n
|  5|jordan| 22|\n
|  8|Jack_1| 18|\n
|  3|  mark| 20|\n
|  6|    qz| 13|\n
|  4|  kobe| 24|\n
+---+------+---+\n
```

使用上述read.format()或者read.jdbc()方法读取到的dataFrame注册为临时表，就可使用sql语句进行数据查询了。

```
jdbcDF.registerTempTable("customer_test")
sparkSession.sql("select * from customer_test where id = 1").show()
```

查询结果：


```
+---+----+---+
| id|name|age|
+---+----+---+
|  1| tom| 18|
+---+----+---+
```

- DataFrame相关操作

  createDataFrame() 方法创建的数据和read.format() 方法及read.jdbc() 方法查询的数据都为DataFrame对象，可以直接进行查询单条记录等操作（在“步骤**d**”中，提到将DataFrame数据注册为临时表）。

  - where

    where 方法中可传入包含and 和 or 的条件筛选表达式，返回过滤后的DataFrame对象，示例如下：

    ```
    jdbcDF.where("id = 1 or age <=10").show()
    ```

    ```
    +---+------+---+
    | id|  name|age|
    +---+------+---+
    |  7|   chm| 13|
    |  1|   tom| 18|
    |  2|  ammy| 18|
    |  5|jordan| 22|
    |  6|    qz| 13|
    |  3|  mark| 20|
    +---+------+---+
    ```

  - filter

    filter同where的使用方式一致，传入条件筛选表达式，返回过滤后的结果 。示例如下：

    ```
    jdbcDF.filter("id = 1 or age <=10").show()
    ```

    ```
    +---+------+---+
    | id|  name|age|
    +---+------+---+
    |  4|  kobe| 24|
    |  7|   chm| 13|
    |  5|jordan| 22|
    |  6|    qz| 13|
    |  3|  mark| 20|
    +---+------+---+
    ```

  - select

    传入待查询的字段，返回指定字段的DataFrame对象，并且可多个字段查询，示例如下：

    - 示例1：

      ```
      jdbcDF.select("id").show()
      ```

```
+---+
| id|
+---+
|  4|
|  7|
|  3|
|  6|
|  1|
|  2|
|  5|
+---+
```

- 示例2：
  jdbcDF.select(**"id"**, **"name"**).show()

```
+---+------+
| id|  name|
+---+------+
|  4|  kobe|
|  7|   chm|
|  6|    qz|
|  3|  mark|
|  1|   tom|
|  2|  ammy|
|  5|jordan|
+---+------+
```

- 示例3：
  jdbcDF.select(**"id"**,**"name"**).where(**"id<4"**).show()

```
+---+----+
| id|name|
+---+----+
|  1| tom|
|  2|ammy|
|  3|mark|
+---+----+
```

– selectExpr

  对字段进行特殊处理。例如，可使用selectExpr修改字段名。示例如下：

  将name字段取名name_test，age数据加1。

  jdbcDF.selectExpr(**"id"**, **"name as name_test"**, **"age+1"**).show()

– col

  获取指定字段。不同于select，col每次只能获取一个字段，返回类型为Column类型，示例如下：

  val idCol = jdbcDF.col(**"id"**)

– drop

  删除指定字段。传入要删除的字段，返回不包含此字段的DataFrame对象，并且每次只能删除一个字段，示例如下：

  jdbcDF.drop(**"id"**).show()


```
+----+---+
|name|age|
+----+---+
|  qz| 13|
| chm| 13|
| tom| 18|
|ammy| 18|
+----+---+
```

- 提交Spark作业
  a. 将写好的代码生成jar包，上传至OBS桶中。
  b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  说明

  - 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.rds。
  - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- 通过SQL API访问

```
import java.util.Properties
import org.apache.spark.sql.SparkSession

object Test_SQL_RDS {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().getOrCreate()

    // Create a data table for DLI-associated RDS
    sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (
   'url'='jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306,
      'dbtable'='test.customer',
      'user'='root',
      'password'='######',
          'driver'='com.mysql.jdbc.Driver')")

    //*****************************SQL model***********************************
    //Insert data into the DLI data table
    sparkSession.sql("insert into dli_to_rds values(1,'John',24),(2,'Bob',32)")

    //Read data from DLI data table
    val dataFrame = sparkSession.sql("select * from dli_to_rds")
    dataFrame.show()
```

```
  //drop table
  sparkSession.sql("drop table dli_to_rds")

  sparkSession.close()
 }
}
```

- 通过DataFrame API访问

```
import java.util.Properties
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.SaveMode

object Test_SQL_RDS {
 def main(args: Array[String]): Unit = {
  // Create a SparkSession session.
  val sparkSession = SparkSession.builder().getOrCreate()

  //*****************************DataFrame model***********************************
  // Set the connection configuration parameters. Contains url, username, password, dbtable.
  val url = "jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306"
  val username = "root"
  val password = "######"
  val dbtable = "test.customer"

  // Create a DataFrame and initialize the DataFrame data.
  var dataFrame_1 = sparkSession.createDataFrame(List((1, "Jack", 18)))

  // Rename the fields set by the createDataFrame() method.
  val df = dataFrame_1.withColumnRenamed("_1", "id")
            .withColumnRenamed("_2", "name")
            .withColumnRenamed("_3", "age")

  // Write data to the rds_table_1 table
  df.write.format("jdbc")
   .option("url", url)
   .option("dbtable", dbtable)
   .option("user", username)
   .option("password", password)
   .option("driver", "com.mysql.jdbc.Driver")
   .mode(SaveMode.Append)
   .save()

  // DataFrame object for data manipulation
  //Filter users with id=1
  var newDF = df.filter("id!=1")
  newDF.show()

  // Filter the id column data
  var newDF_1 = df.drop("id")
  newDF_1.show()

  // Read the data of the customer table in the RDS database
  // Way one：Read data from RDS using read.format()
  val jdbcDF = sparkSession.read.format("jdbc")
            .option("url", url)
            .option("dbtable", dbtable)
            .option("user", username)
            .option("password", password)
            .option("driver", "com.mysql.jdbc.Driver")
            .load()
  // Way two：Read data from RDS using read.jdbc()
  val properties = new Properties()
  properties.put("user", username)
  properties.put("password", password)
  val jdbcDF2 = sparkSession.read.jdbc(url, dbtable, properties)

  /**
   * Register the dateFrame read by read.format() or read.jdbc() as a temporary table, and query the
data
```



```
    * using the sql statement.
    */
   jdbcDF.registerTempTable("customer_test")
   val result = sparkSession.sql("select * from customer_test where id = 1")
   result.show()

   sparkSession.close()
  }
}
```

- DataFrame相关操作

```
  // The where() method uses " and" and "or" for condition filters, returning filtered DataFrame
objects
  jdbcDF.where("id = 1 or age <=10").show()

  // The filter() method is used in the same way as the where() method.
  jdbcDF.filter("id = 1 or age <=10").show()

  // The select() method passes multiple arguments and returns the DataFrame object of the specified
field.
  jdbcDF.select("id").show()
  jdbcDF.select("id", "name").show()
  jdbcDF.select("id","name").where("id<4").show()

  /**
   * The selectExpr() method implements special handling of fields, such as renaming, increasing or
   * decreasing data values.
   */
  jdbcDF.selectExpr("id", "name as name_test", "age+1").show()

  // The col() method gets a specified field each time, and the return type is a Column type.
  val idCol = jdbcDF.col("id")

  /**
   * The drop() method returns a DataFrame object that does not contain deleted fields, and only one
field
   * can be deleted at a time.
   */
  jdbcDF.drop("id").show()
```

4.3.6.2 pyspark 样例代码

开发说明

- 前提条件

  在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现详解

  a. import相关依赖包

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType
  from pyspark.sql import SparkSession
  ```

  b. 创建会话

  ```
  sparkSession = SparkSession.builder.appName("datasource-rds").getOrCreate()
  ```

- 通过DataFrame API 访问

  a. 连接参数配置

  ```
  url = "jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306"
  dbtable = "test.customer"
  ```

```
user = "root"
password = "######"
driver = "com.mysql.jdbc.Driver"
```

参数说明请参考**表4-19**。

b. 设置数据

```
dataList = sparkSession.sparkContext.parallelize([(123, "Katie", 19)])
```

c. 设置schema

```
schema = StructType([StructField("id", IntegerType(), False),\
                     StructField("name", StringType(), False),\
                     StructField("age", IntegerType(), False)])
```

d. 创建DataFrame

```
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

e. 保存数据到RDS

```
dataFrame.write \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .mode("Append") \
    .save()
```

说明

mode 有四种保存类型：

- ErrorIfExis：如果已经存在数据，则抛出异常。
- Overwrite：如果已经存在数据，则覆盖原数据。
- Append：如果已经存在数据，则追加保存。
- Ignore：如果已经存在数据，则不做操作。这类似于SQL中的“如果不存在则创建表”。

f. 读取RDS上的数据

```
jdbcDF = sparkSession.read \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .load()
jdbcDF.show()
```

g. 操作结果

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|123|Katie| 19|
+---+-----+---+
```

- 通过SQL API 访问

a. 创建DLI跨源访问rds的关联表，填写连接参数。

```
sparkSession.sql(
    "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (\
    'url'='jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306',\
    'dbtable'='test.customer',\
    'user'='root',\
    'password'='######',\
    'driver'='com.mysql.jdbc.Driver')")
```

创建表参数请参考**表4-19**。

b. 插入数据
```
sparkSession.sql("insert into dli_to_rds values(3,'John',24)")
```

c. 查询数据
```
jdbcDF_after = sparkSession.sql("select * from dli_to_rds")
jdbcDF_after.show()
```

d. 操作结果
```
+---+-----+---+
| id| name|age|
+---+-----+---+
|123|Katie| 19|
|  3| John| 24|
+---+-----+---+
```

- 提交Spark作业
  a. 将写好的python代码文件上传至OBS桶中。
  b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。
  c. 创建Spark作业完成后，在控制台单击右上角“执行”提交作业，页面显示“批处理作业提交成功”说明Spark作业提交成功，可以在Spark作业管理页面查看提交的作业的状态和日志。

  说明

  - 创建Spark作业时选择的“所属队列”为创建跨源连接时所绑定的队列。
  - 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.rds。
  - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf)' 配置

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

说明

直接复制如下样例代码到py文件中后，需要注意文件内容中的“\”后面可能会有unexpected character的问题。需要将“\”后面的缩进或是空格全部删除。

- 通过DataFrame API访问
```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession
if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-rds").getOrCreate()

  # Set cross-source connection parameters.
  url = "jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306"
  dbtable = "test.customer"
  user = "root"
  password = "######"
  driver = "com.mysql.jdbc.Driver"
```

```
# Create a DataFrame and initialize the DataFrame data.
dataList = sparkSession.sparkContext.parallelize([(123, "Katie", 19)])

# Setting schema
schema = StructType([StructField("id", IntegerType(), False),\
                     StructField("name", StringType(), False),\
                     StructField("age", IntegerType(), False)])

# Create a DataFrame from RDD and schema
dataFrame = sparkSession.createDataFrame(dataList, schema)

# Write data to the RDS.
dataFrame.write \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .mode("Append") \
    .save()

# Read data
jdbcDF = sparkSession.read \
    .format("jdbc") \
    .option("url", url) \
    .option("dbtable", dbtable) \
    .option("user", user) \
    .option("password", password) \
    .option("driver", driver) \
    .load()
jdbcDF.show()

# close session
sparkSession.stop()
```

- 通过SQL API访问

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-rds").getOrCreate()

  # Createa data table for DLI - associated RDS
  sparkSession.sql(
       "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (\
       'url'='jdbc:mysql://to-rds-1174404952-ZgPo1nNC.datasource.com:3306',\
       'dbtable'='test.customer',\
       'user'='root',\
       'password'='######',\
       'driver'='com.mysql.jdbc.Driver')")

  # Insert data into the DLI data table
  sparkSession.sql("insert into dli_to_rds values(3,'John',24)")

  # Read data from DLI data table
  jdbcDF = sparkSession.sql("select * from dli_to_rds")
  jdbcDF.show()

  # close session
  sparkSession.stop()
```

4.3.6.3 java 样例代码

开发说明

- 前提条件

  在DLI管理控制台上已完成创建跨源连接并绑定队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现

  a. 导入依赖

  - 涉及到的mvn依赖库

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

  - import相关依赖包

    ```
    import org.apache.spark.sql.SparkSession;
    ```

  b. 创建会话

  ```
  SparkSession sparkSession = SparkSession.builder().appName("datasource-rds").getOrCreate();
  ```

- 通过SQL API 访问

  – 创建DLI跨源访问RDS的关联表，填写连接参数。

  ```
  sparkSession.sql(
    "CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS (
       'url'='jdbc:mysql://to-rds-1174404209-cA37siB6.datasource.com:3306',  //根据实际url修改
       'dbtable'='test.customer',
       'user'='root',  //根据实际user修改
       'password'='######',  //根据实际password修改
       'driver'='com.mysql.jdbc.Driver')")
  ```

  创建表参数说明请参考**表4-19**。

  – 插入数据

  ```
  sparkSession.sql("insert into dli_to_rds values (1,'John',24)");
  ```

  – 查询数据

  ```
  sparkSession.sql("select * from dli_to_rd").show();
  ```

  插入数据后：

  ```
  +---+----+---+
  | id|name|age|
  +---+----+---+
  |  1|John| 24|
  +---+----+---+
  ```

- 提交Spark作业

  a. 将写好的代码生成jar包，上传至OBS桶中。

  b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  c. 创建Spark作业完成后，在控制台单击右上角“执行”提交作业，页面显示“批处理作业提交成功”说明Spark作业提交成功，可以在Spark作业管理页面查看提交的作业的状态和日志。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.rds。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/rds/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

通过SQL API访问

```
import org.apache.spark.sql.SparkSession;

public class java_rds {

    public static void main(String[] args) {
        SparkSession sparkSession = SparkSession.builder().appName("datasource-rds").getOrCreate();

        // Create a data table for DLI-associated RDS
        sparkSession.sql("CREATE TABLE IF NOT EXISTS dli_to_rds USING JDBC OPTIONS ('url'='jdbc:mysql://
192.168.6.150:3306','dbtable'='test.customer','user'='root','password'='**','driver'='com.mysql.jdbc.Driver')");

        //*****************************SQL model***********************************
        //Insert data into the DLI data table
        sparkSession.sql("insert into dli_to_rds values(3,'Liu',21),(4,'Joey',34)");

        //Read data from DLI data table
        sparkSession.sql("select * from dli_to_rds");

        //drop table
        sparkSession.sql("drop table dli_to_rds");

        sparkSession.close();
    }
}
```

4.3.7 对接 Redis

4.3.7.1 scala 样例代码

开发说明

redis只支持增强型跨源。只能使用包年包月队列。

- 前提条件

  在DLI管理控制台上已完成创建增强跨源连接，并绑定包年包月队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 构造依赖信息，创建SparkSession
  a. 导入依赖。

     涉及到mvn依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.1.0</version>
</dependency>
<dependency>
  <groupId>com.redislabs</groupId>
  <artifactId>spark-redis</artifactId>
  <version>2.4.0</version>
</dependency>
```

     import相关依赖包

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types._
import com.redislabs.provider.redis._
import scala.reflect.runtime.universe._
import org.apache.spark.{SparkConf, SparkContext}
```

- 通过DataFrame API访问
  a. 创建session

```
val sparkSession = SparkSession.builder().appName("datasource_redis").getOrCreate()
```

  b. 构造schema

```
//method one
var schema = StructType(Seq(StructField("name", StringType, false), StructField("age",
IntegerType, false)))
var rdd = sparkSession.sparkContext.parallelize(Seq(Row("abc",34),Row("Bob",19)))
var dataFrame = sparkSession.createDataFrame(rdd, schema)
// //method two
// var jdbcDF= sparkSession.createDataFrame(Seq(("Jack",23)))
// val dataFrame = jdbcDF.withColumnRenamed("_1", "name").withColumnRenamed("_2",
"age")
// //method three
// case class Person(name: String, age: Int)
// val dataFrame = sparkSession.createDataFrame(Seq(Person("John", 30), Person("Peter", 45)))
```

     说明

     case class Person(name: String, age: Int) 须写在object之外，可参考**•通过DataFrame API访问**。

  c. 导入数据到Redis

```
dataFrame .write
  .format("redis")
  .option("host","192.168.4.199")
  .option("port","6379")
  .option("table","person")
  .option("password","******")
  .option("key.column","name")
  .mode(SaveMode.Overwrite)
  .save()
```

表 4-20 redis 操作参数

参数	描述
host	需要连接的redis集群的IP。 获取方式为：登录华为云官网，之后搜索redis，进入“分布式缓存服务”,接着选择“缓存管理”，根据主机名称需要的IP，可选择其中任意一个IP进行复制即可（其中也包含了port信息），请参考图4-42。
port	访问端口。
password	连接密码。无密码时可以不填写该参数。
table	对应Redis中的Key或Hash Key。 • 插入redis数据时必填。 • 查询redis数据时与“keys.pattern”参数二选一。
keys.pattern	使用正则表达式匹配多个Key或Hash Key。该参数仅用于查询时使用。查询redis数据时与“table”参数二选一。
key.column	指定列为key值（非必选）。如果写入数据时指定了key，则查询时必须指定key，否则查询时会异常加载key。
partitions.number	读取数据时，并发task数。
scan.count	每批次读取的数据记录数，默认为100。如果在读取过程中，redis集群中的CPU使用率还有提升空间，可以调大该参数。
iterator.grouping.size	每批次插入的数据记录数，默认为100。如果在插入过程中，redis集群中的CPU使用率还有提升空间，可以调大该参数。
timeout	连接redis的超时时间，单位ms，默认值2000（2秒超时）。

说明

- 保存类型：Overwrite、Append、ErrorIfExis、Ignore 四种
- 如果需要保存嵌套的DataFrame，则通过“.option("model","binary")”进行保存
- 指定数据过期时间：“.option("ttl",1000)”;秒为单位

图 4-42 获取 redis 的 ip 及端口

d. 读取Redis上的数据

```
sparkSession.read
 .format("redis")
 .option("host","192.168.4.199")
 .option("port","6379")
 .option("table", "person")
 .option("password","######")
 .option("key.column","name")
 .load()
 .show()
```

操作结果：

```
+------+---+\n
|  name|age|\n
+------+---+\n
|huawei| 34|\n
+------+---+\n
|   Bob| 19|\n
+------+---+\n\n
```

- RDD操作

a. 创建连接

```
val sparkContext = new SparkContext(new SparkConf()
  .setAppName("datasource_redis")
  .set("spark.redis.host", "192.168.4.199")
  .set("spark.redis.port", "6379")
  .set("spark.redis.auth", "######")
  .set("spark.driver.allowMultipleContexts","true"))
```

说明

spark.driver.allowMultipleContexts：true 表示在启动多个context时，只使用当前的，防止发生context调用冲突。

b. 插入数据

i. String 保存

```
val stringRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("high","111"), ("together","333")))
sparkContext.toRedisKV(stringRedisData)
```

ii. hash 保存

```
val hashRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("saprk","123"), ("data","222")))
sparkContext.toRedisHASH(hashRedisData, "hashRDD")
```

iii. list 保存

```
val data = List(("school","112"), ("tom","333"))
val listRedisData:RDD[String] = sparkContext.parallelize(Seq[(String)](data.toString()))
sparkContext.toRedisLIST(listRedisData, "listRDD")
```

iv. set 保存

```
val setData = Set(("bob","133"),("kity","322"))
val setRedisData:RDD[(String)] = sparkContext.parallelize(Seq[(String)]
(setData.mkString))
sparkContext.toRedisSET(setRedisData, "setRDD")
```

v. zset 保存

```
val zsetRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("whight","234"), ("bobo","343")))
sparkContext.toRedisZSET(zsetRedisData, "zsetRDD")
```

c. 查询数据

i. 通过遍历key查询

```
val keysRDD = sparkContext.fromRedisKeys(Array("high","together", "hashRDD",
"listRDD", "setRDD","zsetRDD"), 6)
```

```
keysRDD.getKV().collect().foreach(println)
keysRDD.getHash().collect().foreach(println)
keysRDD.getList().collect().foreach(println)
keysRDD.getSet().collect().foreach(println)
keysRDD.getZSet().collect().foreach(println)
```

ii. string 查询

```
sparkContext.fromRedisKV(Array( "high","together")).collect().foreach{println}
```

iii. hash 查询

```
sparkContext.fromRedisHash(Array("hashRDD")).collect().foreach{println}
```

iv. list 查询

```
sparkContext.fromRedisList(Array("listRDD")).collect().foreach{println}
```

v. set 查询

```
sparkContext.fromRedisSet(Array("setRDD")).collect().foreach{println}
```

vi. zset 查询

```
sparkContext.fromRedisZSet(Array("zsetRDD")).collect().foreach{println}
```

- 通过SQL API 访问

a. 创建DLI关联跨源访问 Redis的关联表。

```
sparkSession.sql(
    "CREATE TEMPORARY VIEW person (name STRING, age INT) USING
org.apache.spark.sql.redis OPTIONS (
    'host' = '192.168.4.199',
    'port' = '6379',
    'password' = '######',
    table  'person')".stripMargin)
```

b. 插入数据

```
sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)
```

c. 查询数据

```
sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)
```

- 提交Spark作业

a. 将写好的代码生成jar包，上传至OBS桶中。

b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**创建批处理作业**》。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.redis。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
```



```
  <version>2.3.2</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.1.0</version>
</dependency>
<dependency>
  <groupId>com.redislabs</groupId>
  <artifactId>spark-redis</artifactId>
  <version>2.4.0</version>
</dependency>
```

- 通过SQL API访问

```
import org.apache.spark.sql.{SparkSession};

object Test_Redis_SQL {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkSession = SparkSession.builder().appName("datasource_redis").getOrCreate();

    sparkSession.sql(
      "CREATE TEMPORARY VIEW person (name STRING, age INT) USING org.apache.spark.sql.redis
OPTIONS (
        'host' = '192.168.4.199', 'port' = '6379', 'password' = '******',table 'person')".stripMargin)

    sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)

    sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)

    sparkSession.close()
  }
}
```

- 通过DataFrame API访问

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types._

object Test_Redis_SparkSql {
  def main(args: Array[String]): Unit = {
  // Create a SparkSession session.
  val sparkSession = SparkSession.builder().appName("datasource_redis").getOrCreate()

  // Set cross-source connection parameters.
  val host = "192.168.4.199"
  val port = "6379"
  val table = "person"
  val auth = "######"
  val key_column = "name"

  // ******** setting DataFrame ********
  // method one
  var schema = StructType(Seq(StructField("name", StringType, false),StructField("age", IntegerType,
false)))
  var rdd = sparkSession.sparkContext.parallelize(Seq(Row("huawei",34),Row("Bob",19)))
  var dataFrame = sparkSession.createDataFrame(rdd, schema)

// // method two
// var jdbcDF= sparkSession.createDataFrame(Seq(("Jack",23)))
// val dataFrame = jdbcDF.withColumnRenamed("_1", "name").withColumnRenamed("_2", "age")

// // method three
// val dataFrame = sparkSession.createDataFrame(Seq(Person("John", 30), Person("Peter", 45)))

  // Write data to redis
  dataFrame.write.format("redis").option("host",host).option("port",port).option("table",
table).option("password",auth).mode(SaveMode.Overwrite).save()

  // Read data from redis
  sparkSession.read.format("redis").option("host",host).option("port",port).option("table",
```

```
table).option("password",auth).load().show()

  // Close session
  sparkSession.close()
  }
}
// methoe two
// case class Person(name: String, age: Int)
```

- RDD 操作

```
import com.redislabs.provider.redis._
import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

object Test_Redis_RDD {
  def main(args: Array[String]): Unit = {
    // Create a SparkSession session.
    val sparkContext = new SparkContext(new SparkConf()
          .setAppName("datasource_redis")
          .set("spark.redis.host", "192.168.4.199")
          .set("spark.redis.port", "6379")
          .set("spark.redis.auth", "@@@@@@")
          .set("spark.driver.allowMultipleContexts","true"))

    //****************** Write data to redis **********************
    // Save String type data
    val stringRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("high","111"), ("together","333")))
    sparkContext.toRedisKV(stringRedisData)

    // Save Hash type data
    val hashRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("saprk","123"), ("data","222")))
    sparkContext.toRedisHASH(hashRedisData, "hashRDD")

    // Save List type data
    val data = List(("school","112"), ("tom","333"));
    val listRedisData:RDD[String] = sparkContext.parallelize(Seq[(String)](data.toString()))
    sparkContext.toRedisLIST(listRedisData, "listRDD")

    // Save Set type data
    val setData = Set(("bob","133"),("kity","322"))
    val setRedisData:RDD[(String)] = sparkContext.parallelize(Seq[(String)](setData.mkString))
    sparkContext.toRedisSET(setRedisData, "setRDD")

    // Save ZSet type data
    val zsetRedisData:RDD[(String,String)] = sparkContext.parallelize(Seq[(String,String)]
(("whight","234"), ("bobo","343")))
    sparkContext.toRedisZSET(zsetRedisData, "zsetRDD")

    // ***************************** Read data from redis *****************************************
    // Traverse the specified key and get the value
    val keysRDD = sparkContext.fromRedisKeys(Array("high","together", "hashRDD", "listRDD",
"setRDD","zsetRDD"), 6)
    keysRDD.getKV().collect().foreach(println)
    keysRDD.getHash().collect().foreach(println)
    keysRDD.getList().collect().foreach(println)
    keysRDD.getSet().collect().foreach(println)
    keysRDD.getZSet().collect().foreach(println)

    // Read String type data//
    val stringRDD = sparkContext.fromRedisKV("keyPattern *")
    sparkContext.fromRedisKV(Array( "high","together")).collect().foreach{println}

    // Read Hash type data//
    val hashRDD = sparkContext.fromRedisHash("keyPattern *")
    sparkContext.fromRedisHash(Array("hashRDD")).collect().foreach{println}

    // Read List type data//
    val listRDD = sparkContext.fromRedisList("keyPattern *")
```

```
  sparkContext.fromRedisList(Array("listRDD")).collect().foreach{println}

  // Read Set type data//
  val setRDD = sparkContext.fromRedisSet("keyPattern *")
  sparkContext.fromRedisSet(Array("setRDD")).collect().foreach{println}

  // Read ZSet type data//
  val zsetRDD = sparkContext.fromRedisZSet("keyPattern *")
  sparkContext.fromRedisZSet(Array("zsetRDD")).collect().foreach{println}

  // close session
  sparkContext.stop()
 }
}
```

4.3.7.2 pyspark 样例代码

开发说明

redis只支持增强型跨源。只能使用包年包月队列。

- 前提条件

  在DLI管理控制台上已完成创建增强跨源连接，并绑定包年包月队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 通过DataFrame API 访问

  a. import相关依赖

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType
  from pyspark.sql import SparkSession
  ```

  b. 创建session

  ```
  sparkSession = SparkSession.builder.appName("datasource-redis").getOrCreate()
  ```

  c. 设置连接参数

  ```
  host = "192.168.4.199"
  port = "6379"
  table = "person"
  auth = "@@@@@@"
  ```

  d. 创建DataFrame

  i. 方式一

  ```
  dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19),(2,"Tom",20)])
  schema = StructType([StructField("id", IntegerType(), False),
                       StructField("name", StringType(), False),
                       StructField("age", IntegerType(), False)])
  dataFrame = sparkSession.createDataFrame(dataList, schema)
  ```

  ii. 方式二

  ```
  jdbcDF = sparkSession.createDataFrame([(3,"Jack", 23)])
  dataFrame = jdbcDF.withColumnRenamed("_1", "id").withColumnRenamed("_2",
  "name").withColumnRenamed("_3", "age")
  ```

  e. 导入数据到redis

  ```
  dataFrame.write
    .format("redis")\
    .option("host", host)\
    .option("port", port)\
    .option("table", table)\
    .option("password", auth)\
  ```

```
.mode("Overwrite")\
.save()
```

说明

- 保存类型：Overwrite、Append、ErrorIfExis、Ignore 四种
- 如果需要指定key，则通过“.option("key.column","name")”指定，name为列名
- 如果需要保存嵌套的DataFrame，则通过“.option("model","binary")”进行保存
- 如果需要指定数据过期时间：“.option("ttl",1000)”;秒为单位

f. 读取redis上的数据

```
sparkSession.read.format("redis").option("host", host).option("port", port).option("table", table).option("password", auth).load().show()
```

g. 操作结果

```
+---+-----+---+\n
| id| name|age|\n
+---+-----+---+\n
|  2|  Tom| 20|\n
|  1|Katie| 19|\n
+---+-----+---+\n\n
```

- 通过SQL API 访问

a. 创建DLI关联跨源访问 Redis的关联表。

```
sparkSession.sql(
    "CREATE TEMPORARY VIEW person (name STRING, age INT) USING
org.apache.spark.sql.redis OPTIONS (
    'host' = '192.168.4.199',
    'port' = '6379',
    'password' = '######',
    table  'person')".stripMargin)
```

b. 插入数据

```
sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)
```

c. 查询数据

```
sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)
```

- 提交Spark作业

a. 将写好的python代码文件上传至OBS桶中。

b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.redis。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- 通过DataFrame API 访问

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession
if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-redis").getOrCreate()

  # Set cross-source connection parameters.
  host = "192.168.4.199"
  port = "6379"
  table = "person"
  auth = "######"

  # Create a DataFrame and initialize the DataFrame data.
  # *******    method noe    *********
  dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19),(2,"Tom",20)])
  schema = StructType([StructField("id", IntegerType(), False),StructField("name", StringType(),
False),StructField("age", IntegerType(), False)])
  dataFrame_one = sparkSession.createDataFrame(dataList, schema)

  # ****** method two ******
  # jdbcDF = sparkSession.createDataFrame([(3,"Jack", 23)])
  # dataFrame = jdbcDF.withColumnRenamed("_1", "id").withColumnRenamed("_2",
"name").withColumnRenamed("_3", "age")

  # Write data to the redis table
  dataFrame.write.format("redis").option("host", host).option("port", port).option("table",
table).option("password", auth).mode("Overwrite").save()
  # Read data
  sparkSession.read.format("redis").option("host", host).option("port", port).option("table",
table).option("password", auth).load().show()

  # close session
  sparkSession.stop()
```

- 通过SQL API 访问

```
# _*_ coding: utf-8 _*_
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession
  sparkSession = SparkSession.builder.appName("datasource_redis").getOrCreate()

  sparkSession.sql(
    "CREATE TEMPORARY VIEW person (name STRING, age INT) USING org.apache.spark.sql.redis
OPTIONS (\
    'host' = '192.168.4.199', \
    'port' = '6379',\
    'password' = '######',\
    'table'= 'person')".stripMargin);

  sparkSession.sql("INSERT INTO TABLE person VALUES ('John', 30),('Peter', 45)".stripMargin)

  sparkSession.sql("SELECT * FROM person".stripMargin).collect().foreach(println)

  # close session
  sparkSession.stop()
```

4.3.7.3 java 样例代码

开发说明

redis只支持增强型跨源。只能使用包年包月队列。

- 前提条件

  在DLI管理控制台上已完成创建增强跨源连接，并绑定包年包月队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现

  a. 导入依赖。

  - 涉及到的mvn依赖库

    ```
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.11</artifactId>
      <version>2.3.2</version>
    </dependency>
    ```

  - import相关依赖包

    ```
    import org.apache.spark.SparkConf;
    import org.apache.spark.api.java.JavaRDD;
    import org.apache.spark.api.java.JavaSparkContext;
    import org.apache.spark.sql.*;
    import org.apache.spark.sql.types.DataTypes;
    import org.apache.spark.sql.types.StructField;
    import org.apache.spark.sql.types.StructType;
    import java.util.*;
    ```

  b. 创建会话

  ```
  SparkConf sparkConf = new SparkConf();
  sparkConf.setAppName("datasource-redis")
        .set("spark.redis.host", "192.168.4.199")
        .set("spark.redis.port", "6379")
        .set("spark.redis.auth", "******")
        .set("spark.driver.allowMultipleContexts","true");
  JavaSparkContext javaSparkContext = new JavaSparkContext(sparkConf);
  SQLContext sqlContext = new SQLContext(javaSparkContext);
  ```

- 通过DataFrame API 访问

  a. 读取json数据为DataFrame

  ```
  JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList(
        "{\"id\":\"1\",\"name\":\"Ann\",\"age\":\"18\"}",
        "{\"id\":\"2\",\"name\":\"lisi\",\"age\":\"21\"}"));
  Dataset dataFrame = sqlContext.read().json(javaRDD);
  ```

  b. 构造redis连接配置参数

  ```
  Map map = new HashMap<String, String>();
  map.put("table","person");
  map.put("key.column","id");
  ```

  c. 保存数据到redis

  ```
  dataFrame.write().format("redis").options(map).mode(SaveMode.Overwrite).save();
  ```

  d. 读取redis中数据

  ```
  sqlContext.read().format("redis").options(map).load().show();
  ```

  e. 操作结果

  ```
  +---+--------+---+\n
  | id|    name|age|\n
  +---+--------+---+\n
  |  1|zhangsan| 18|\n
  +---+--------+---+\n
  |  2|    lisi| 21|\n
  +---+--------+---+\n\n
  ```

- 提交Spark作业
  a. 将写好的java代码文件上传至OBS桶中。
  b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  说明

  - 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.redis。
  - 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

    spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*

    spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/redis/*
  - 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
  - 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

```
public class Test_Redis_DaraFrame {
  public static void main(String[] args) {
    //create a SparkSession session
    SparkConf sparkConf = new SparkConf();
    sparkConf.setAppName("datasource-redis")
            .set("spark.redis.host", "192.168.4.199")
            .set("spark.redis.port", "6379")
            .set("spark.redis.auth", "*******")
            .set("spark.driver.allowMultipleContexts","true");
    JavaSparkContext javaSparkContext = new JavaSparkContext(sparkConf);
    SQLContext sqlContext = new SQLContext(javaSparkContext);

    //Read RDD in JSON format to create DataFrame
    JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList(
            "{\"id\":\"1\",\"name\":\"Ann\",\"age\":\"18\"}",
            "{\"id\":\"2\",\"name\":\"lisi\",\"age\":\"21\"}"));
    Dataset dataFrame = sqlContext.read().json(javaRDD);

    Map map = new HashMap<String, String>();
    map.put("table","person");
    map.put("key.column","id");
    dataFrame.write().format("redis").options(map).mode(SaveMode.Overwrite).save();
    sqlContext.read().format("redis").options(map).load().show();

  }
}
```

4.3.7.4 故障处理

问题 1：将代码直接复制到 py 文件中后，'\'后出现“unexpected character”问题。

- 问题
  将代码直接复制到py文件中后，'\'后出现“unexpected character”问题。
- 解决方案
  将'\'后面的缩进或是空格全部删除。

4.3.8 对接 Mongo

4.3.8.1 scala 样例代码

开发说明

mongo只支持增强型跨源。只能使用包年包月队列。

说明

DDS即文档数据库服务，兼容MongoDB协议。

在DLI管理控制台上已完成创建增强跨源连接，并绑定包年/包月队列。具体操作请参考《**数据湖探索用户指南**》。

说明

认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 构造依赖信息，创建SparkSession
  a. 导入依赖。

  涉及到mvn依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  import相关依赖包

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}
```

  创建session

```
val sparkSession = SparkSession.builder().appName("datasource-mongo").getOrCreate()
```

- 通过SQL API访问
  a. 创建DLI跨源访问 mongo的关联表

```
sparkSession.sql(
  "create table test_dds(id string, name string, age int) using mongo options(
    'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',
    'uri' = 'mongodb://username:pwd@host:8635/db',
    'database' = 'test',
    'collection' = 'test',
    'user' = 'rwuser',
    'password' = '######')")
```

表 4-21 创建表参数

参数	说明
url	• url的格式为："IP:PORT[,IP:PORT]/[DATABASE][.COLLECTION][AUTH_PROPERTIES]" 例如： "192.168.4.62:8635/test?authSource=admin" • url需要在mongo（DDS）的连接地址的截取得到。获取到的mongo的连接地址格式为："协议头://用户名:密码@访问地址:访问端口/数据库名?authSource=admin" 例如： mongodb://rwuser:****@192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin
uri	uri的格式为：mongodb://username:pwd@host:8635/db 其中以下参数需要修改为实际值： • “username”为创建的mongo（DDS）数据库用户名。 • “pwd”为创建的mongo（DDS）数据库用户名对应的密码。 • “host”为创建的mongo（DDS）数据库实例IP。 • “db”为创建的mongo（DDS）数据库名称。 mongo（DDS）数据库用户创建详见：**创建DDS数据库账户**。
database	DDS的数据库名，如果在"url"中同时指定了数据库名，则"url"中的数据库名不生效。
collection	DDS中的collection名，如果在"url"中同时指定了collection，则"url"中的collection不生效。 **说明** 如果在DDS中已存在collection，则建表可以不指定schema信息，DLI会根据collection中的数据自动生成schema信息。
user	访问DDS集群用户名。
password	访问DDS集群密码。

b. 插入数据

```
sparkSession.sql("insert into test_dds values('3', 'Ann',23)")
```

c. 查询数据

```
sparkSession.sql("select * from test_dds").show()
```

操作结果

```
+---+--------+---+\n
| id|    name|age|\n
+---+--------+---+\n
|  2|     Bob| 32|\n
|  1|    John| 23|\n
|  3|zhangsan| 23|\n
+---+--------+---+\n\n
```

- 通过DataFrame API访问
  a. 设置连接参数
  ```
  val url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
  val uri = "mongodb://username:pwd@host:8635/db"
  val user = "rwuser"
  val database = "test"
  val collection = "test"
  val password = "######"
  ```
  b. 构造schema
  ```
  val schema = StructType(List(StructField("id", StringType), StructField("name", StringType), StructField("age", IntegerType)))
  ```
  c. 构造DataFrame
  ```
  val rdd = spark.sparkContext.parallelize(Seq(Row("1", "John", 23), Row("2", "Bob", 32)))
  val dataFrame = spark.createDataFrame(rdd, schema)
  ```
  d. 导入数据到mongo
  ```
  dataFrame.write.format("mongo")
    .option("url", url)
    .option("uri", uri)
    .option("database", database)
    .option("collection", collection)
    .option("user", user)
    .option("password", password)
    .mode(SaveMode.Overwrite)
    .save()
  ```
  说明

  保存类型：Overwrite、Append、ErrorIfExis、Ignore 四种。
  e. 读取mongo上的数据
  ```
  val jdbcDF = spark.read.format("mongo").schema(schema)
    .option("url", url)
    .option("uri", uri)
    .option("database", database)
    .option("collection", collection)
    .option("user", user)
    .option("password", password)
    .load()
  ```
  操作结果
  ```
  +---+----+---+\n
  | id|name|age|\n
  +---+----+---+\n
  |  2| Bob| 32|\n
  |  1|John| 23|\n
  +---+----+---+\n\n
  ```
- 提交Spark作业
  a. 将写好的代码生成jar包，上传至OBS桶中。
  b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

  控制台操作请参考《**数据湖探索用户指南**》。API操作请参考《数据湖探索API参考》>《**创建批处理作业**》。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.mongo。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- Maven依赖

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

- 通过SQL API访问

```
import org.apache.spark.sql.SparkSession

object TestMongoSql {
  def main(args: Array[String]): Unit = {
    val sparkSession = SparkSession.builder().getOrCreate()
    sparkSession.sql(
      "create table test_dds(id string, name string, age int) using mongo options(
        'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',
        'uri' = 'mongodb://username:pwd@host:8635/db',
        'database' = 'test',
        'collection' = 'test',
        'user' = 'rwuser',
        'password' = '######')")
    sparkSession.sql("insert into test_dds values('3', 'Ann',23)")
    sparkSession.sql("select * from test_dds").show()
    sparkSession.close()
  }
}
```

- 通过DataFrame API访问

```
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}

object Test_Mongo_SparkSql {
  def main(args: Array[String]): Unit = {
  //  Create a SparkSession session.
  val spark = SparkSession.builder().appName("mongodbTest").getOrCreate()

  // Set the connection configuration parameters.
  val url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
  val uri = "mongodb://username:pwd@host:8635/db"
  val user = "rwuser"
  val database = "test"
  val collection = "test"
  val password = "######"

  // Setting up the schema
  val schema = StructType(List(StructField("id", StringType), StructField("name", StringType),
StructField("age", IntegerType)))
```



```
// Setting up the DataFrame
val rdd = spark.sparkContext.parallelize(Seq(Row("1", "John", 23), Row("2", "Bob", 32)))
val dataFrame = spark.createDataFrame(rdd, schema)


// Write data to mongo
dataFrame.write.format("mongo")
  .option("url", url)
  .option("uri", uri)
  .option("database", database)
  .option("collection", collection)
  .option("user", user)
  .option("password", password)
  .mode(SaveMode.Overwrite)
  .save()

// Reading data from mongo
val jdbcDF = spark.read.format("mongo").schema(schema)
  .option("url", url)
  .option("uri", uri)
  .option("database", database)
  .option("collection", collection)
  .option("user", user)
  .option("password", password)
  .load()
jdbcDF.show()

 spark.close()
 }
}
```

4.3.8.2 pyspark 样例代码

开发说明

mongo只支持增强型跨源。只能使用包年包月队列。

说明

DDS即文档数据库服务，兼容MongoDB协议。

- 前提条件

  在DLI管理控制台上已完成创建增强跨源连接，并绑定包年/包月队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 通过DataFrame API 访问

  a. import相关依赖

  ```
  from __future__ import print_function
  from pyspark.sql.types import StructType, StructField, IntegerType, StringType
  from pyspark.sql import SparkSession
  ```

  b. 创建session

  ```
  sparkSession = SparkSession.builder.appName("datasource-mongo").getOrCreate()
  ```

  c. 设置连接参数

  ```
  url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
  uri = "mongodb://username:pwd@host:8635/db"
  user = "rwuser"
  database = "test"
  collection = "test"
  password = "######"
  ```

> **说明**
>
> 详细的参数说明请参考**表4-21**。

d. 创建DataFrame

```
dataList = sparkSession.sparkContext.parallelize([(1, "Katie", 19),(2,"Tom",20)])
schema = StructType([StructField("id", IntegerType(), False),
                     StructField("name", StringType(), False),
                     StructField("age", IntegerType(), False)])
dataFrame = sparkSession.createDataFrame(dataList, schema)
```

e. 导入数据到mongo

```
dataFrame.write.format("mongo")
  .option("url", url)\
  .option("uri", uri)\
  .option("user",user)\
  .option("password",password)\
  .option("database",database)\
  .option("collection",collection)\
  .mode("Overwrite")\
  .save()
```

f. 读取Mongo上的数据

```
jdbcDF = sparkSession.read
  .format("mongo")\
  .option("url", url)\
  .option("uri", uri)\
  .option("user",user)\
  .option("password",password)\
  .option("database",database)\
  .option("collection",collection)\
  .load()
jdbcDF.show()
```

g. 操作结果

```
+---+-----+---+\n
| id| name|age|\n
+---+-----+---+\n
|  2|  Tom| 20|\n
|  1|Katie| 19|\n
+---+-----+---+\n\n
```

- 通过SQL API 访问

a. 创建DLI关联跨源访问 Mongo的关联表。

```
sparkSession.sql(
    "create table test_dds(id string, name string, age int) using mongo options(
    'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',
    'uri' = 'mongodb://username:pwd@host:8635/db',
    'database' = 'test',
    'collection' = 'test',
    'user' = 'rwuser',
    'password' = '######')")
```

> **说明**
>
> 详细的参数说明请参考**表4-21**。

b. 插入数据

```
sparkSession.sql("insert into test_dds values('3', 'Ann',23)")
```

c. 查询数据

```
sparkSession.sql("select * from test_dds").show()
```

- 提交Spark作业

a. 将写好的python代码文件上传至OBS桶中。

b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.mongo。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

- 通过DataFrame API 访问

```
from __future__ import print_function
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-mongo").getOrCreate()

  # Create a DataFrame and initialize the DataFrame data.
  dataList = sparkSession.sparkContext.parallelize([("1", "Katie", 19),("2","Tom",20)])

  # Setting schema
  schema = StructType([StructField("id", IntegerType(), False),StructField("name", StringType(), False),
StructField("age", IntegerType(), False)])

  # Create a DataFrame from RDD and schema
  dataFrame = sparkSession.createDataFrame(dataList, schema)

  # Setting connection parameters
  url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin"
  uri = "mongodb://username:pwd@host:8635/db"
  user = "rwuser"
  database = "test"
  collection = "test"
  password = "######"

  # Write data to the mongodb table
  dataFrame.write.format("mongo")
    .option("url", url)\
    .option("uri", uri)\
    .option("user",user)\
    .option("password",password)\
    .option("database",database)\
    .option("collection",collection)
    .mode("Overwrite").save()

  # Read data
  jdbcDF = sparkSession.read.format("mongo")
    .option("url", url)\
    .option("uri", uri)\
    .option("user",user)\
    .option("password",password)\
    .option("database",database)\
    .option("collection",collection)\
    .load()
```



```
jdbcDF.show()

# close session
sparkSession.stop()
```

- 通过SQL API 访问

```
from __future__ import print_function
from pyspark.sql import SparkSession

if __name__ == "__main__":
  # Create a SparkSession session.
  sparkSession = SparkSession.builder.appName("datasource-mongo").getOrCreate()

  # Createa data table for DLI - associated mongo
    sparkSession.sql(
      "create table test_dds(id string, name string, age int) using mongo options(\
      'url' = '192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin',\
      'uri' = 'mongodb://username:pwd@host:8635/db',\
      'database' = 'test',\
      'collection' = 'test', \
      'user' = 'rwuser', \
      'password' = '######')")

  # Insert data into the DLI-table
  sparkSession.sql("insert into test_dds values('3', 'Ann',23)")

  # Read data from DLI-table
  sparkSession.sql("select * from test_dds").show()

  # close session
  sparkSession.stop()
```

4.3.8.3 java 样例代码

开发说明

mongo只支持增强型跨源。只能使用包年包月队列。

说明

DDS即文档数据库服务，兼容MongoDB协议。

- 前提条件

  在DLI管理控制台上已完成创建增强跨源连接，并绑定包年/包月队列。具体操作请参考《**数据湖探索用户指南**》。

  说明

  认证用的password硬编码到代码中或者明文存储都有很大的安全风险，建议在配置文件或者环境变量中密文存放，使用时解密，确保安全。

- 代码实现详解

  a. 导入依赖

  - 涉及到的mvn依赖库

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.3.2</version>
</dependency>
```

  - import相关依赖包

```
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaRDD;
```

```
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SQLContext;
import org.apache.spark.sql.SaveMode;
```

b. 创建会话

```
SparkContext sparkContext = new SparkContext(new SparkConf().setAppName("datasource-mongo"));
JavaSparkContext javaSparkContext = new JavaSparkContext(sparkContext);
SQLContext sqlContext = new SQLContext(javaSparkContext);
```

- 通过DataFrame API 访问

a. 读取json数据为DataFrame

```
JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList("{\"id\":\"5\",\"name\":\"Ann\",\"age\":\"23\"}"));
Dataset<Row> dataFrame = sqlContext.read().json(javaRDD);
```

b. 设置连接参数

```
String url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin";
String uri = "mongodb://username:pwd@host:8635/db";
String user = "rwuser";
String database = "test";
String collection = "test";
String password = "######";
```

说明

详细的参数说明请参考**表4-21**。

c. 导入数据到mongo

```
dataFrame.write().format("mongo")
    .option("url",url)
    .option("uri",uri)
    .option("database",database)
    .option("collection",collection)
    .option("user",user)
    .option("password",password)
    .mode(SaveMode.Overwrite)
    .save();
```

d. 读取mongo上的数据

```
sqlContext.read().format("mongo")
   .option("url",url)
   .option("uri",uri)
   .option("database",database)
   .option("collection",collection)
   .option("user",user)
   .option("password",password)
   .load().show();
```

e. 操作结果

```
+---+--------+---+\n
| id|    name|age|\n
+---+--------+---+\n
|  5|zhangsan| 23|\n
+---+--------+---+\n\n
```

- 提交Spark作业

a. 将写好的java代码文件上传至OBS桶中。

b. 在Spark作业编辑器中选择对应的Module模块并执行Spark作业。

说明

- 如果选择spark版本为2.3.2（即将下线）或2.4.5提交作业时，需要指定Module模块，名称为：sys.datasource.mongo。
- 如果选择Spark版本为3.1.1及以上版本时，无需选择Module模块， 需在 'Spark参数（--conf）' 配置

  spark.driver.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*

  spark.executor.extraClassPath=/usr/share/extension/dli/spark-jar/datasource/mongo/*
- 通过控制台提交作业请参考《**数据湖探索用户指南**》中的“选择依赖资源参数说明”。
- 通过API提交作业请参考《数据湖探索API参考》>《**创建批处理作业**》中“表2-请求参数说明”关于“modules”参数的说明。

完整示例代码

```
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SQLContext;
import org.apache.spark.sql.SaveMode;
import java.util.Arrays;

public class TestMongoSparkSql {
  public static void main(String[] args) {
    SparkContext sparkContext = new SparkContext(new SparkConf().setAppName("datasource-mongo"));
    JavaSparkContext javaSparkContext = new JavaSparkContext(sparkContext);
    SQLContext sqlContext = new SQLContext(javaSparkContext);

//    // Read json file as DataFrame, read csv / parquet file, same as json file distribution
//    DataFrame dataFrame = sqlContext.read().format("json").load("filepath");

    // Read RDD in JSON format to create DataFrame
    JavaRDD<String> javaRDD = javaSparkContext.parallelize(Arrays.asList("{\"id\":\"5\",\"name\":\"Ann
\",\"age\":\"23\"}"));
    Dataset<Row> dataFrame = sqlContext.read().json(javaRDD);

    String url = "192.168.4.62:8635,192.168.5.134:8635/test?authSource=admin";
    String uri = "mongodb://username:pwd@host:8635/db";
    String user = "rwuser";
    String database = "test";
    String collection = "test";
    String password = "######";

    dataFrame.write().format("mongo")
            .option("url",url)
            .option("uri",uri)
            .option("database",database)
            .option("collection",collection)
            .option("user",user)
            .option("password",password)
            .mode(SaveMode.Overwrite)
            .save();

    sqlContext.read().format("mongo")
            .option("url",url)
            .option("uri",uri)
            .option("database",database)
            .option("collection",collection)
            .option("user",user)
            .option("password",password)
```

```
            .load().show();
    sparkContext.stop();
    javaSparkContext.close();
  }
}
```

4.4 Spark Jar 使用 DEW 获取访问凭证读写 OBS

操作场景

DLI将Spark Jar作业并的输出数据写入到OBS时，需要配置AKSK访问OBS，为了确保AKSK数据安全，您可以通过数据加密服务（Data Encryption Workshop，DEW）、云凭据管理服务（Cloud Secret Management Service，CSMS），对AKSK统一管理，有效避免程序硬编码或明文配置等问题导致的敏感信息泄露以及权限失控带来的业务风险。

本例以获取访问OBS的AKSK为例介绍Spark Jar使用DEW获取访问凭证读写OBS的操作指导。

前提条件

- 已在DEW服务创建通用凭证，并存入凭据值。具体操作请参考：**创建通用凭据**。
- 已创建DLI访问DEW的委托并完成委托授权。该委托需具备以下权限：
  - DEW中的查询凭据的版本与凭据值ShowSecretVersion接口权限，csms:secretVersion:get。
  - DEW中的查询凭据的版本列表ListSecretVersions接口权限，csms:secretVersion:list。
  - DEW解密凭据的权限，kms:dek:decrypt。

    委托权限示例请参考**自定义DLI委托权限**和**常见场景的委托权限策略**。
- 仅支持Spark3.3.1（Spark通用队列场景）及以上版本使用DEW管理访问凭据，在创建作业时，请配置作业使用Spark3.3.1版本、且已在作业中配置允许DLI访问DEW的委托信息。

  自定义委托及配置请参考**自定义DLI委托权限**。
- 使用该功能，所有涉及OBS的桶，都需要进行配置AKSK。

语法格式

在Spark Jar作业编辑界面，选择配置优化参数，配置信息如下:

不同的OBS桶，使用不同的AKSK认证信息。 可以使用如下配置方式，根据桶指定不同的AKSK信息，参数说明详见**表4-22**。

```
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key= USER_AK_CSMS_KEY
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key= USER_SK_CSMS_KEY
spark.hadoop.fs.obs.security.provider = com.dli.provider.UserObsBasicCredentialProvider
spark.hadoop.fs.dew.csms.secretName= CredentialName
spark.hadoop.fs.dew.endpoint=ENDPOINT
spark.hadoop.fs.dew.csms.version=VERSION_ID
spark.hadoop.fs.dew.csms.cache.time.second =CACHE_TIME
spark.dli.job.agency.name=USER_AGENCY_NAME
```

参数说明

表 4-22 参数说明

参数	是否必选	默认值	数据类型	参数说明
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key	是	无	String	其中USER_BUCKET_NAME为用户的桶名，需要进行替换为用户的使用的OBS桶名。 参数的值为用户定义在CSMS通用凭证中的键key， 其Key对应的value为用户的AK（Access Key Id），需要具备访问OBS对应桶的权限。
spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key	是	无	String	其中USER_BUCKET_NAME为用户的桶名，需要进行替换为用户的使用的OBS桶名。 参数的值为用户定义在CSMS通用凭证中的键key， 其Key对应的value为用户的SK（Secret Access Key），需要具备访问OBS对应桶的权限。
spark.hadoop.fs.obs.security.provider	是	无	String	OBS AKSK认证机制，使用DEW服务中的CSMS凭证管理，获取OBS的AK、SK。 默认取值为com.dli.provider.UserObsBasicCredentialProvider
spark.hadoop.fs.dew.csms.secretName	是	无	String	在DEW服务的凭据管理中新建的通用凭据的名称。 配置示例：spark.hadoop.fs.dew.csms.secretName=secretInfo
spark.hadoop.fs.dew.endpoint	是	无	String	指定要使用的DEW服务所在的endpoint信息。 获取**地区和终端节点**。 配置示例：spark.hadoop.fs.dew.endpoint=kms.cn-xxxx.myhuaweicloud.com
spark.hadoop.fs.dew.csms.version	否	最新的version	String	在DEW服务的凭据管理中新建的通用凭据的版本号(凭据的版本标识符)。 若不指定，则默认获取该通用凭证的最新版本号。 配置示例：spark.hadoop.fs.dew.csms.version=v1

参数	是否必选	默认值	数据类型	参数说明
spark.hadoop.fs.dew.csms.cache.time.second	否	3600	Long	Spark作业访问获取CSMS通用凭证后，缓存的时间。 单位为秒。默认值为3600秒。
spark.hadoop.fs.dew.projectId	否	有	String	DEW所在的项目ID， 默认是Spark作业所在的项目ID。 **获取项目ID**
spark.dli.job.agency.name	是	-	String	自定义委托名称。

样例代码

本章节JAVA样例代码演示将DataGen数据处理后写入到OBS，具体参数配置请根据实际环境修改。

1. 创建DLI访问DEW的委托并完成委托授权。
   详细步骤请参考**自定义DLI委托权限**。
2. 在DEW创建通用凭证。详细操作请参考**创建通用凭据**。
   a. 登录DEW管理控制台
   b. 选择“凭据管理”，进入“凭据管理”页面。
   c. 单击“创建凭据”。配置凭据基本信息
3. DLI Spark jar作业编辑界面设置作业参数。
   Spark参数：
   ```
   spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.access.key= USER_AK_CSMS_KEY
   spark.hadoop.fs.obs.bucket.USER_BUCKET_NAME.dew.secret.key= USER_SK_CSMS_KEY
   spark.hadoop.fs.obs.security.provider=com.dli.provider.UserObsBasicCredentialProvider
   spark.hadoop.fs.dew.csms.secretName=obsAkSk
   spark.hadoop.fs.dew.endpoint=kmsendpoint
   spark.hadoop.fs.dew.csms.version=v3
   spark.dli.job.agency.name=agency
   ```
4. 示例代码
   示例代码请参考**使用Spark Jar作业读取和查询OBS数据**。

4.5 获取 Spark 作业委托临时凭证用于访问其他云服务

功能描述

DLI提供了一个通用接口，可用于获取用户在启动Spark作业时设置的委托的临时凭证。该接口将获取到的该作业委托的临时凭证封装到com.huaweicloud.sdk.core.auth.BasicCredentials类中。

- 获取到的委托的临时认证封装到com.huaweicloud.sdk.core.auth.ICredentialProvider接口的getCredentials()返回值中。

- 返回类型为com.huaweicloud.sdk.core.auth.BasicCredentials。
- 仅支持获取AK、SK、SecurityToken。
- 获取到AK、SK、SecurityToken后，请参考**如何使用凭据管理服务替换硬编码的数据库账号密码**查询凭据。

约束限制

- 仅支持Spark3.3.1版本（Spark通用队列场景）使用委托授权访问临时凭证：
  - 在创建作业时，请配置作业使用Spark3.3.1版本
  - 已在作业中配置允许DLI访问DEW的委托信息。spark.dli.job.agency.name=*自定义委托名称*。

    自定义委托请参考**自定义DLI委托权限**。

    请注意配置参数不需要用"" 或 '' 包裹。
- Spark3.3.1基础镜像内置了3.1.62版本的huaweicloud-sdk-core。

准备环境

已安装和配置IntelliJ IDEA等开发工具以及安装JDK和Maven。

pom文件配置中依赖包

```
<dependency>
   <groupId>com.huaweicloud.sdk</groupId>
   <artifactId>huaweicloud-sdk-core</artifactId>
   <version>3.1.62</version>
   <scope>provided</scope>
</dependency>
```

示例代码

本章节JAVA样例代码演示如何获取BasicCredentials，以及取临时委托的AK、SK、SecurityToken。

- **Spark Jar作业获取作业委托凭证**

```
import org.apache.spark.sql.SparkSession;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import com.huaweicloud.sdk.core.auth.BasicCredentials;
import com.huaweicloud.sdk.core.auth.ICredentialProvider;
import static com.huawei.dli.demo.DLIJobAgencyCredentialUtils.getICredentialProvider;
public class GetUserCredentialsSparkJar {
private static final Logger LOG = LoggerFactory.getLogger(GetUserCredentialsSparkJar.class);
public static void main(String[] args) throws Exception {
        SparkSession spark = SparkSession
                        .builder()
                        .appName("test_spark")
                        .getOrCreate();
 ICredentialProvider credentialProvider = getICredentialProvider();
 BasicCredentials basicCredentials = (BasicCredentials) credentialProvider.getCredentials();
 String ak = basicCredentials.getAk();
 String sk = basicCredentials.getSk();
 String securityToken = basicCredentials.getSecurityToken();
 LOG.info(">>" + " ak " + ak + " sk " + sk.length() + " token " + securityToken.length());
 spark.stop();
 }
}
```

- **获取作业委托的工具类**

```
import com.huaweicloud.sdk.core.auth.ICredentialProvider;
import org.apache.spark.sql.SparkSession;
```



```
import java.util.ArrayList;
import java.util.List;
import java.util.ServiceLoader;
public class DLIJobAgencyCredentialUtils {
public static ICredentialProvider getICredentialProvider() {
        List<ICredentialProvider> credentialProviders = new ArrayList<>();
        ServiceLoader.load(ICredentialProvider.class, SparkSession.class.getClassLoader())
                .iterator()
                .forEachRemaining(credentialProviders::add);
        if (credentialProviders.size() != 1) {
           throw new RuntimeException("Failed to obtain temporary user credential");
        }
        return credentialProviders.get(0);
    }
}
```