数据湖探索（DLI）

开发指南

天翼云科技有限公司

目 录

1 Spark SQL 作业开发指南

1.1 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《对象存储服务控制台指南》。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《数据湖探索用户指南》>《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”。

```
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

两种语法创建 OBS 表主要差异点参见表 1-1。

表1-1 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

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。

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;
```

    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;
```

- 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;
       ```

    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。
  - 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;
       ```

    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。

- 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;
```

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;
select * from testcsvdata4source where classNo = 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。

- 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;
```

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%';
```

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

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;
```

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%';
```

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”，行数据分割符为','。

```
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;
```

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;
```

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。

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;
```

- 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;
```

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。

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;
```

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';
  ```

1.2 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1-2 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1-1 开发流程

表1-3 开发流程说明

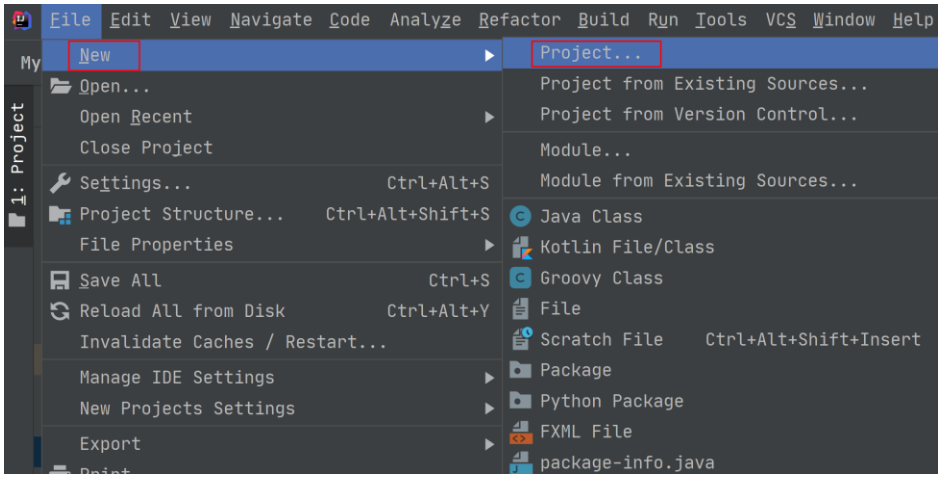
序号	阶段	操作界面	说明
1	新建 Maven 工程，配置 pom 文件	IntelliJ IDEA	参考操作步骤说明，编写 UDF 函数代码。
2	编写 UDF 函数代码		
3	调试，编译代码并导出 Jar 包		

序号	阶段	操作界面	说明
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。

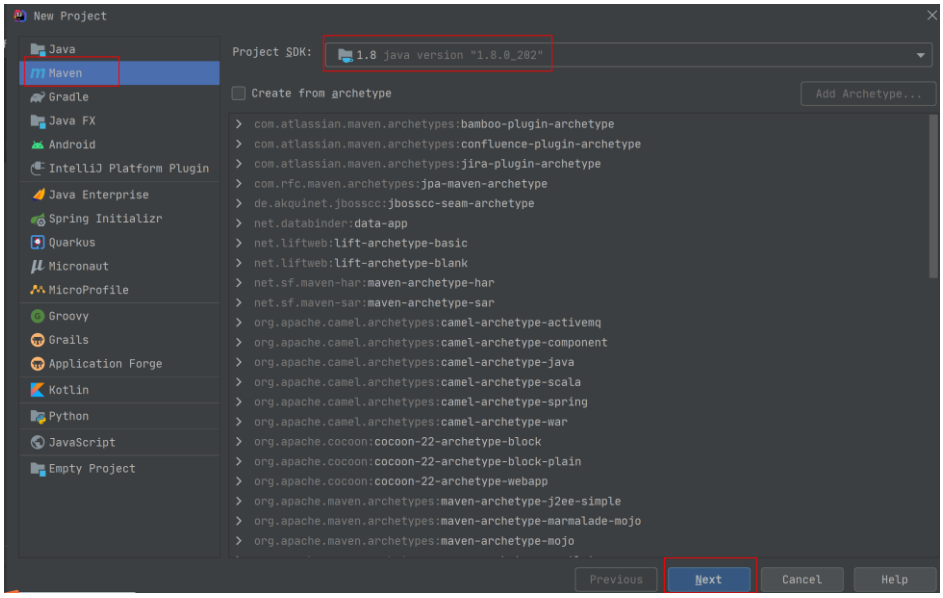
操作步骤

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”。

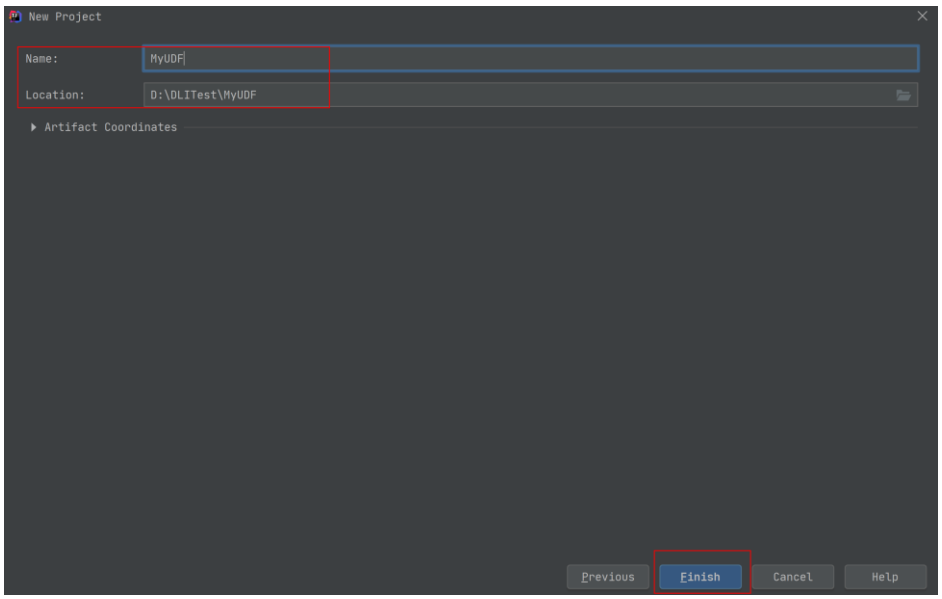
   图1-2 新建 Project



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



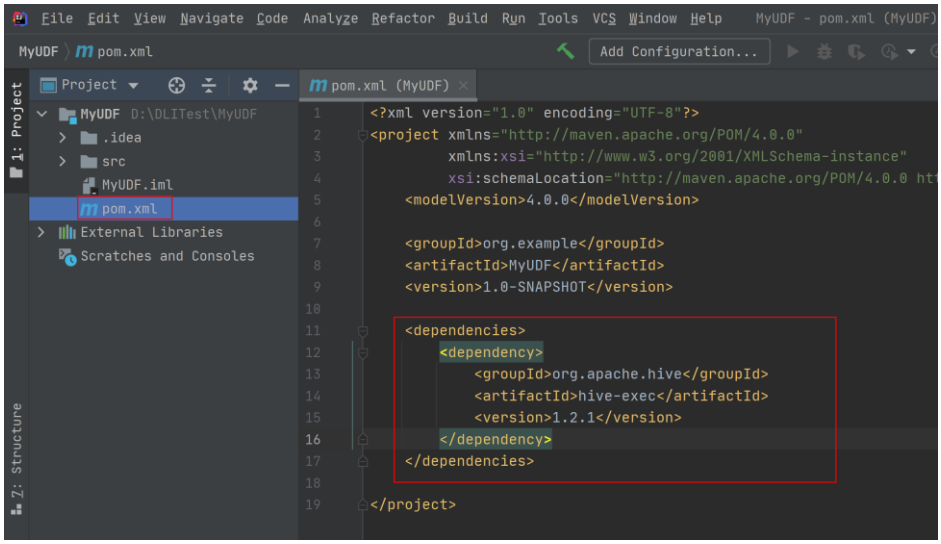
c. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。



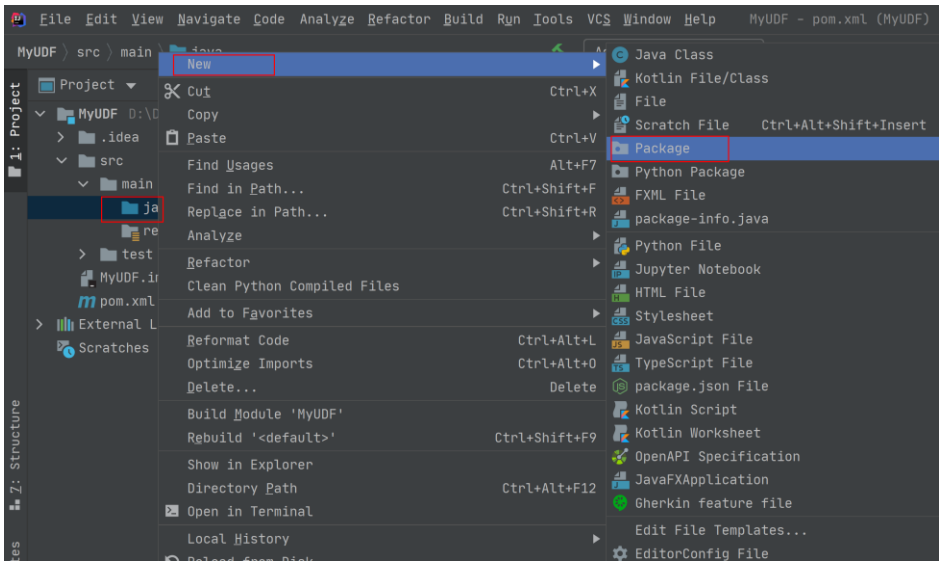
d. 在 pom.xml 文件中添加如下配置。

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

图1-3 pom 文件中添加配置

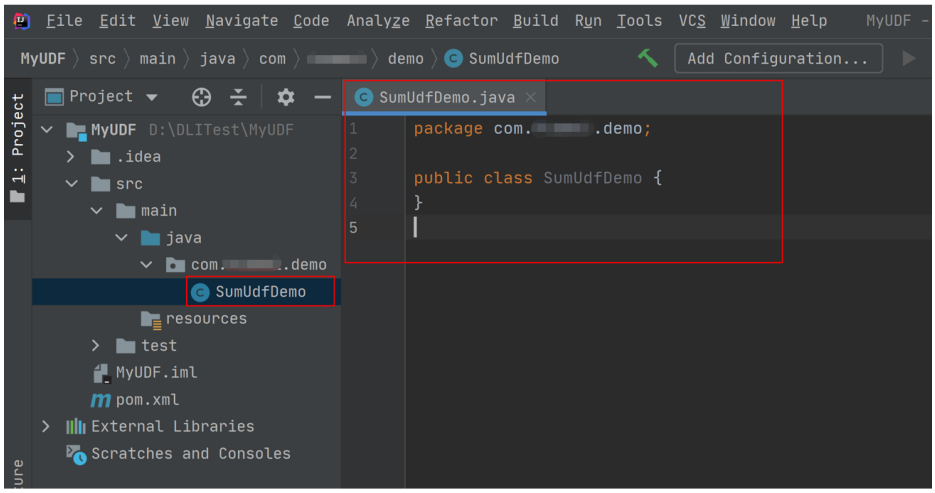


e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建 Package 和类文件。



Package 根据需要定义，完成后回车。

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



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.UDF。

   b. 需要实现 evaluate 函数，evaluate 函数支持重载。

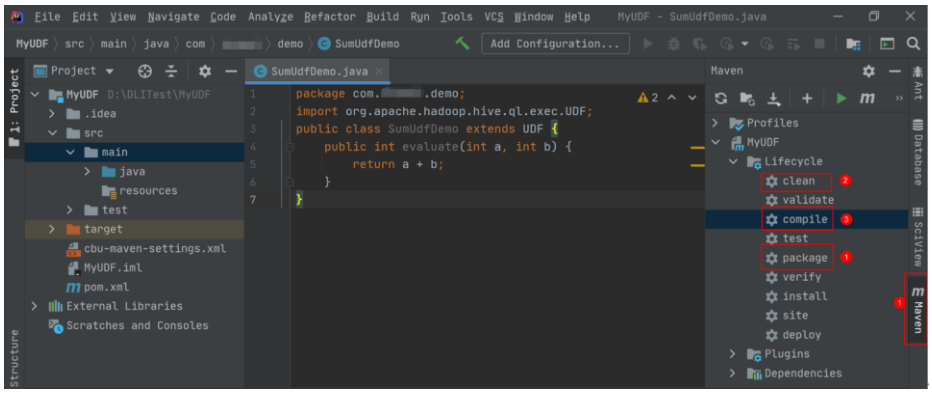
   详细 UDF 函数实现，可以参考如下样例代码：

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

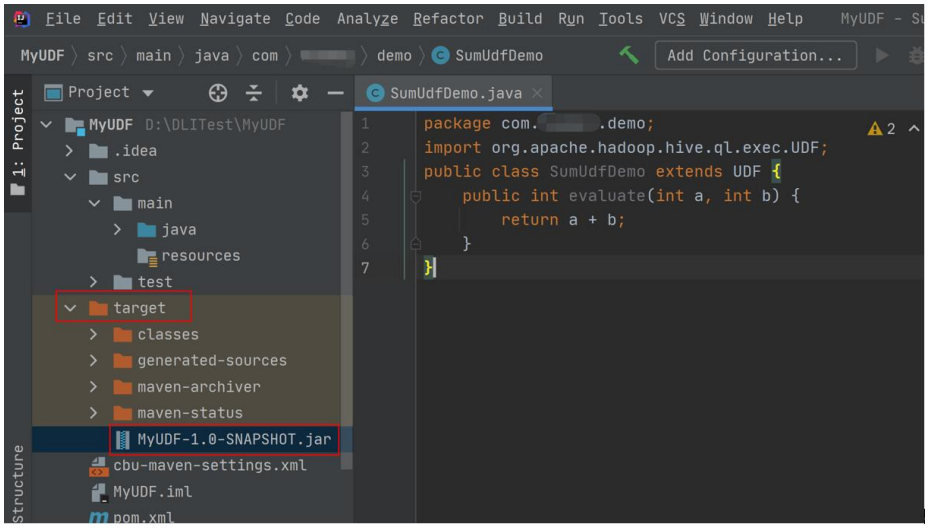
3. 编写调试完成代码后，通过 IntelliJ IDEA 工具编译代码并导出 Jar 包。

   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。



   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”。



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。
   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.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);
```

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;
```

1.3 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1-4 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1-4 UDTF 开发流程

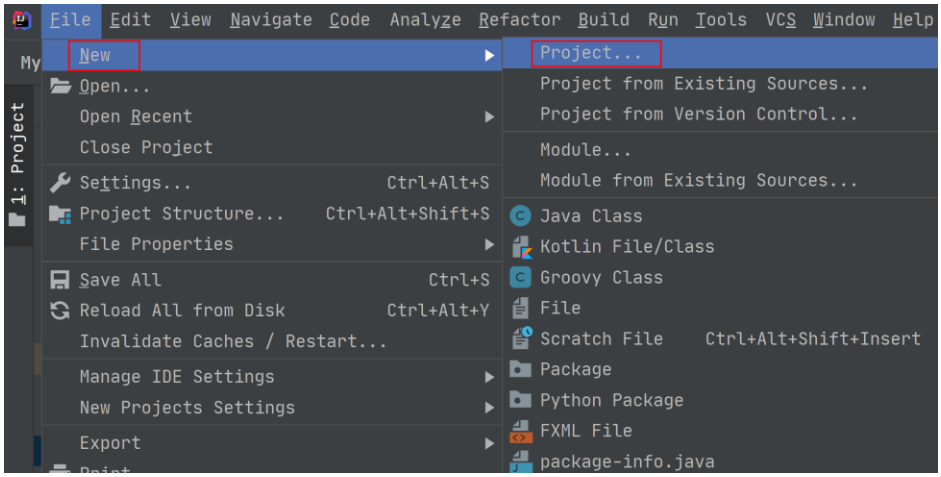
表1-5 开发流程说明

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。

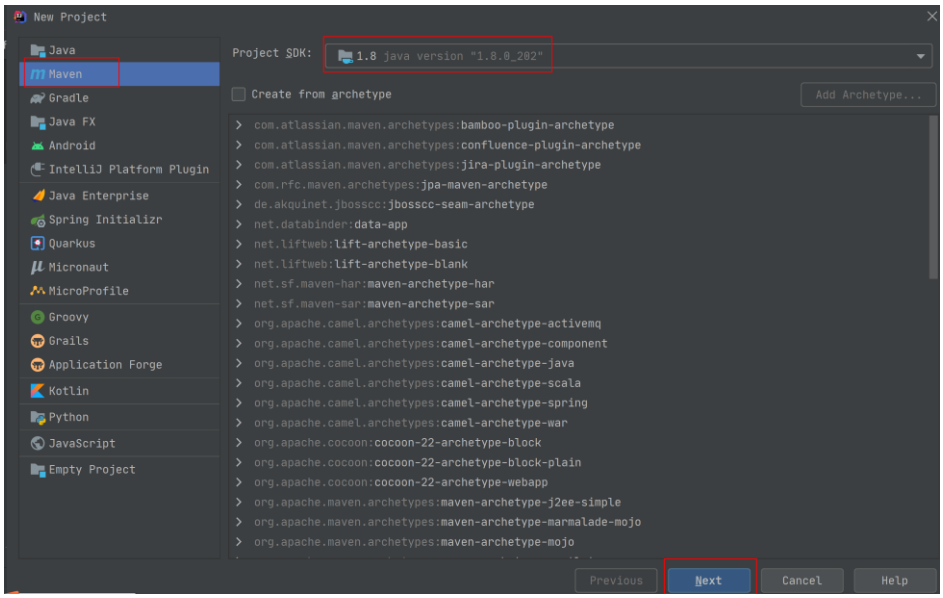
操作步骤

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”。

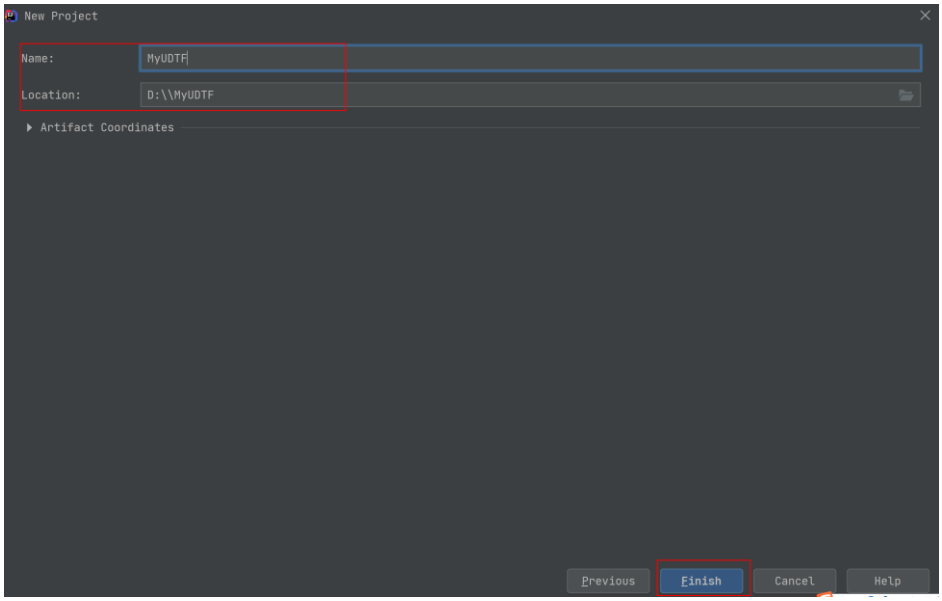
图1-5 新建 Project



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



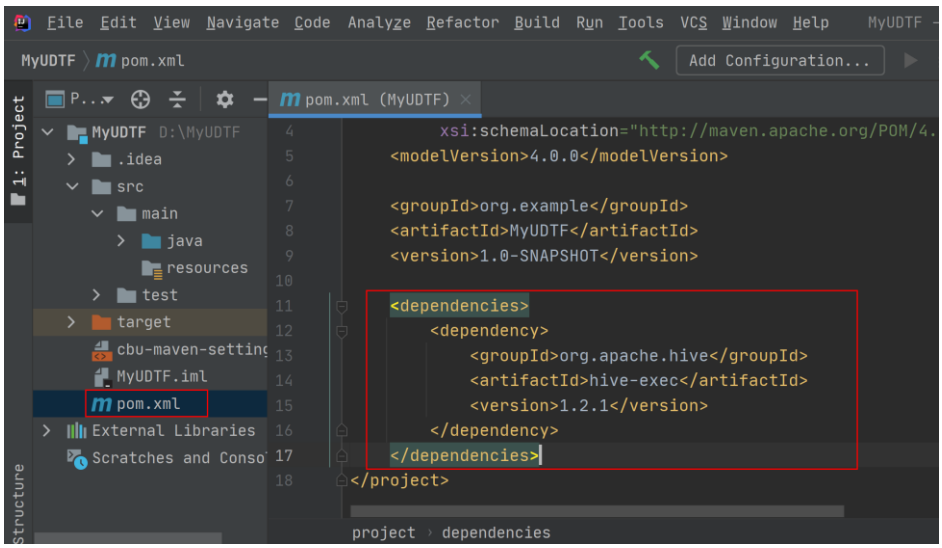
c. 定义样例工程名和配置样例工程存储路径，单击“Finish”完成工程创建。



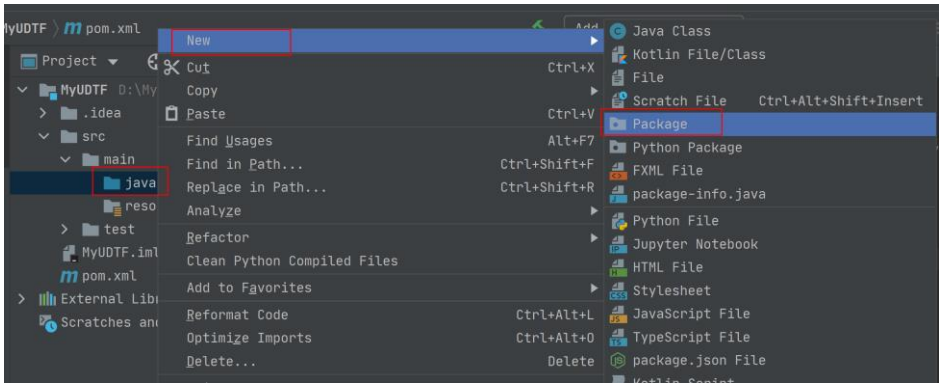
d. 在 pom.xml 文件中添加如下配置。

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

图1-6 pom 文件中添加配置

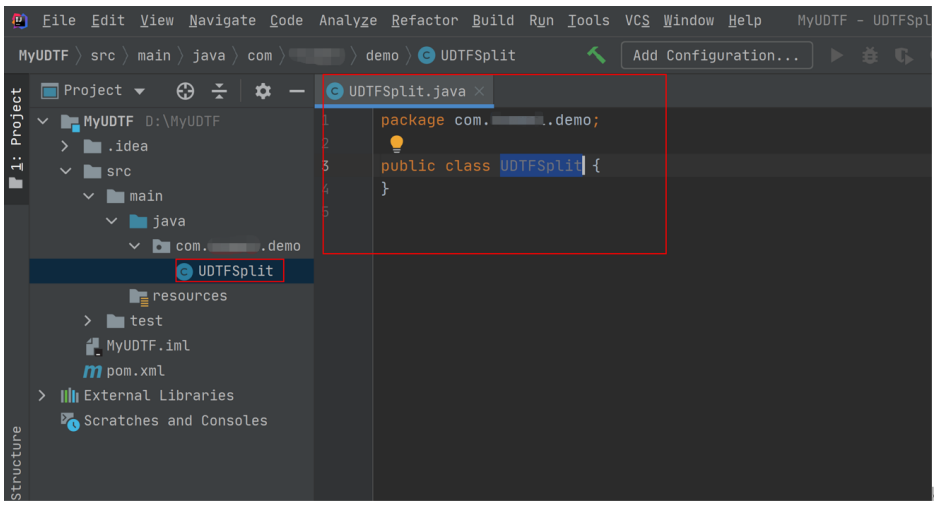


e. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建 Package 和类文件。



Package 根据需要定义，完成后回车。

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



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;
            }
        }

    }
```

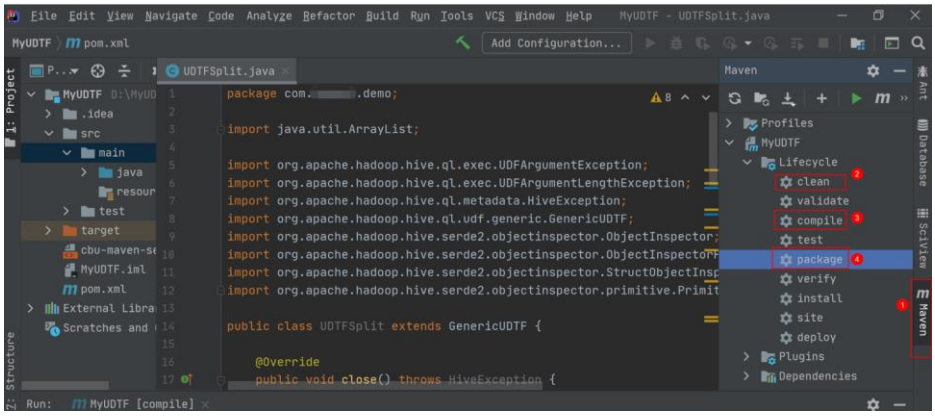
c. 最后调用 close 方法，对需要清理的方法进行清理。

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。

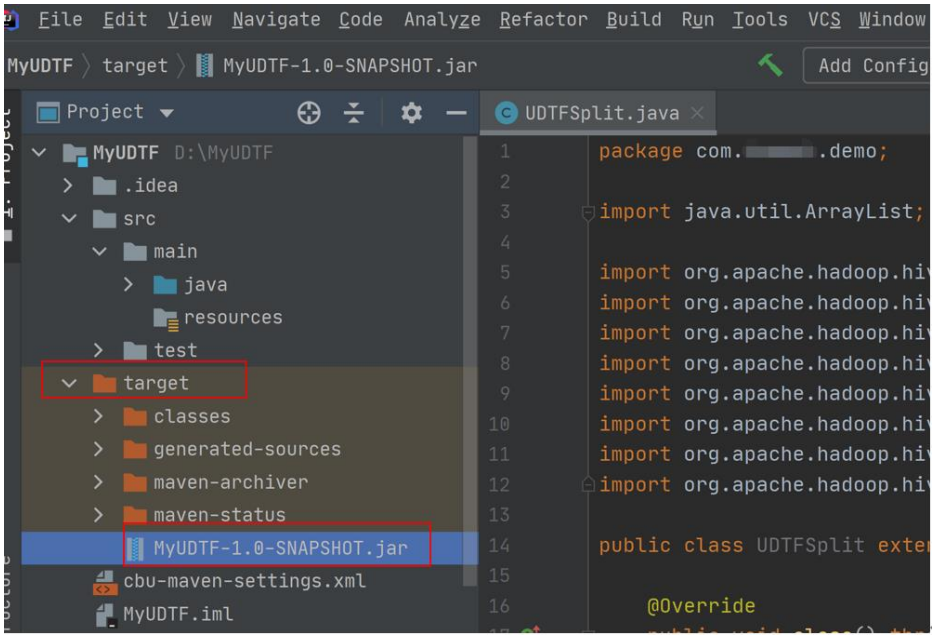
a. 单击工具右侧的“Maven”，参考下图分别单击“clean”、“compile”对代码进行编译。

编译成功后，单击“package”对代码进行打包。

图1-7 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”。



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

```
CREATE FUNCTION mytestsplit AS 'com.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“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');
```

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：

```
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.PrimitiveObjectInspectorFac
tory;

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);
   }
```

```
}
```

1.4 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1-6 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1-8 UDAF 开发流程

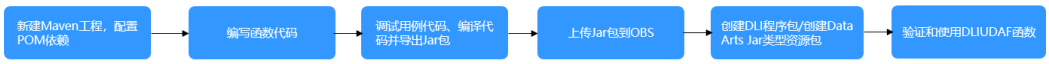


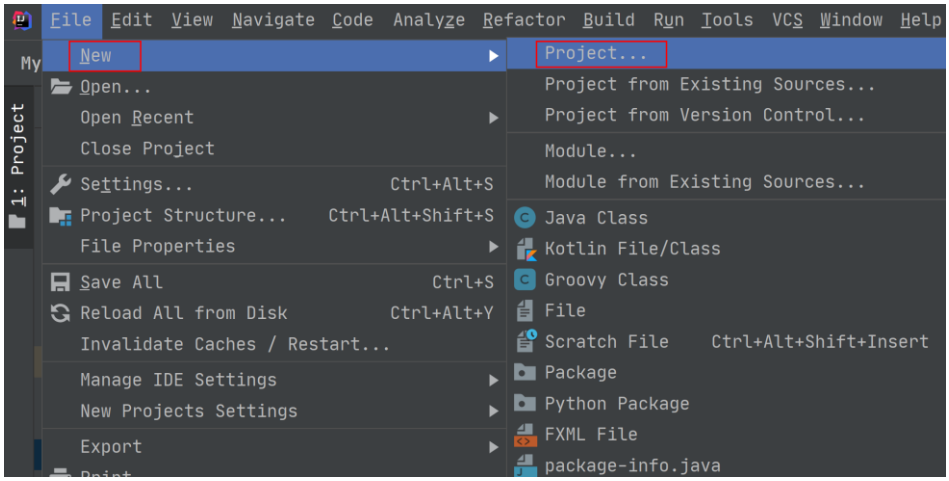
表1-7 开发流程说明

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

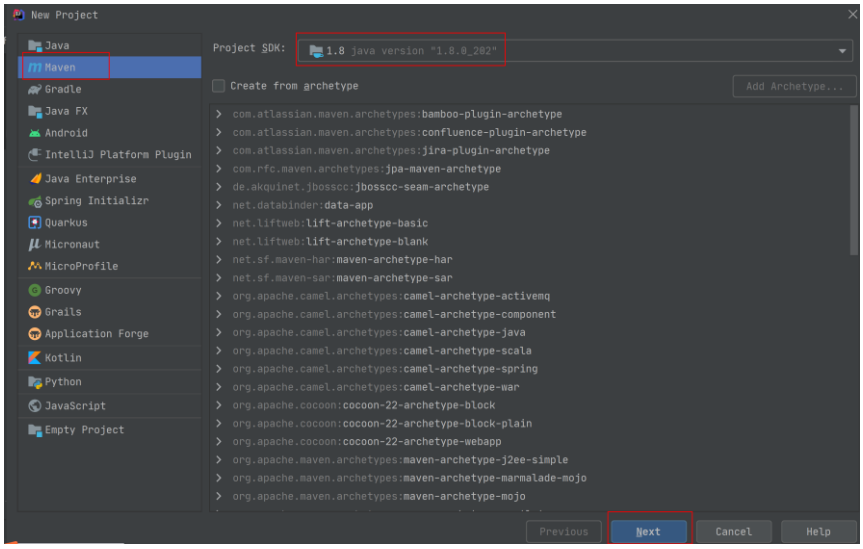
1. 新建 Maven 工程，配置 pom 文件。以下通过 IntelliJ IDEA 2020.2 工具操作演示。
   a. 打开 IntelliJ IDEA，选择“File > New > Project”。

图1-9 新建 Project



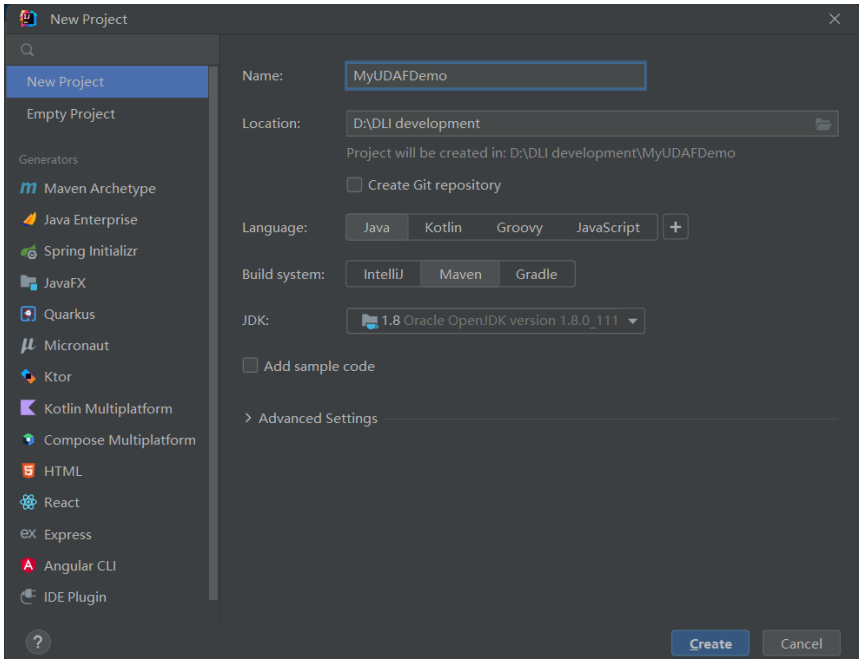
b. 选择 Maven，Project SDK 选择 1.8，单击“Next”。

图1-10 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。

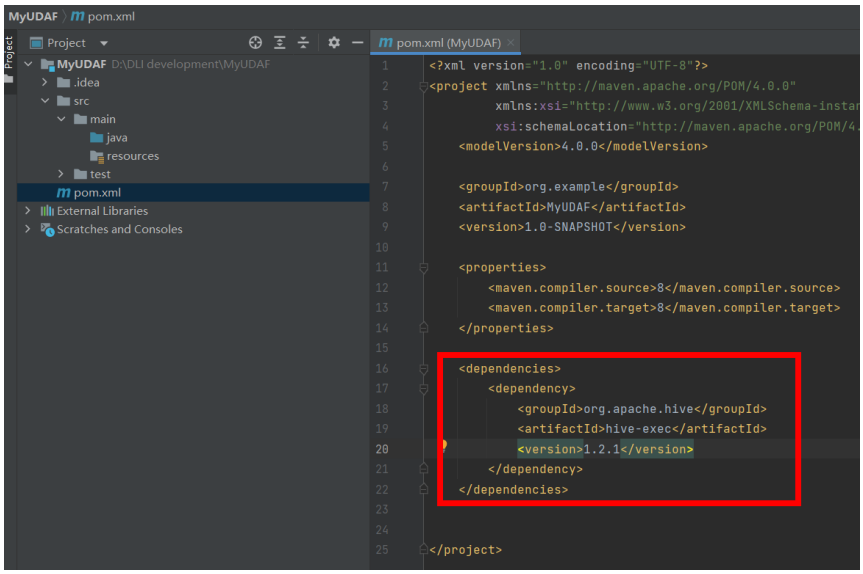
图1-11 完成 Project 创建



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>
```

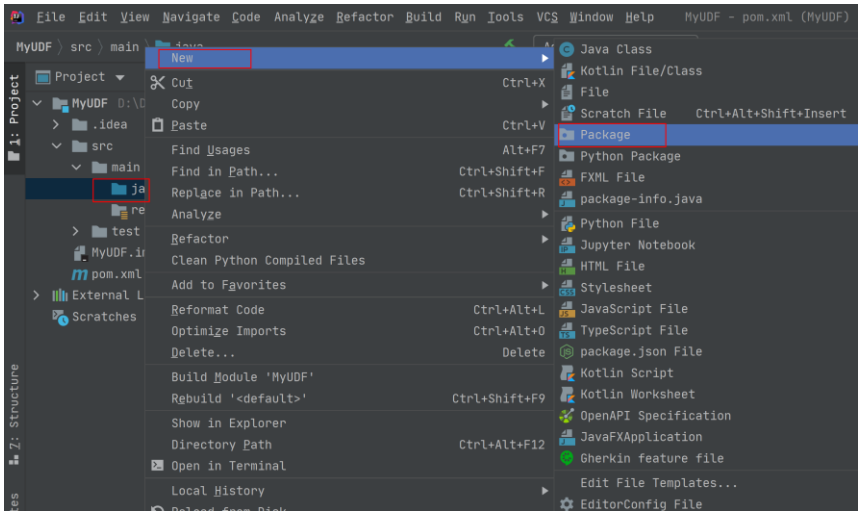
图1-12 pom 文件中添加配置



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。

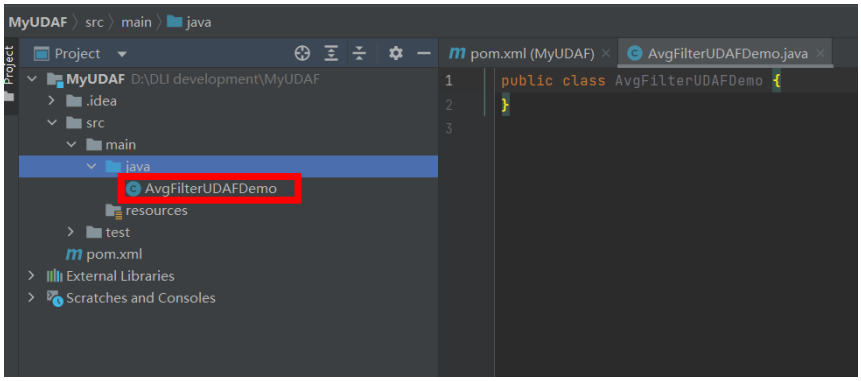
Package 根据需要定义，本示例定义为：“com.dli.demo”

图1-13 新建 Package



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

图1-14 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) {
```

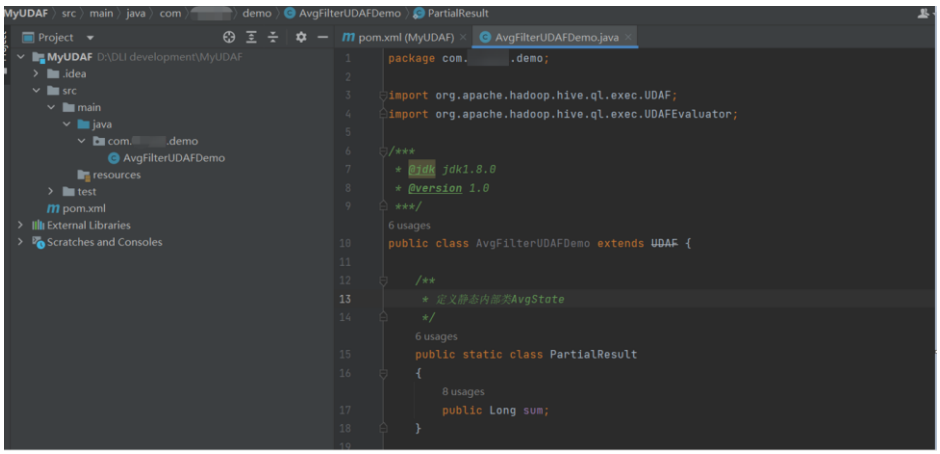
```
            if (x == null) {
                return;
            }
            AvgFilterUDAFDemo.PartialResult tmp9_6 = this.partial;
            tmp9_6.sum = tmp9_6.sum | x;
        }

        /**
         * 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1-15 编写 UDAF 函数代码

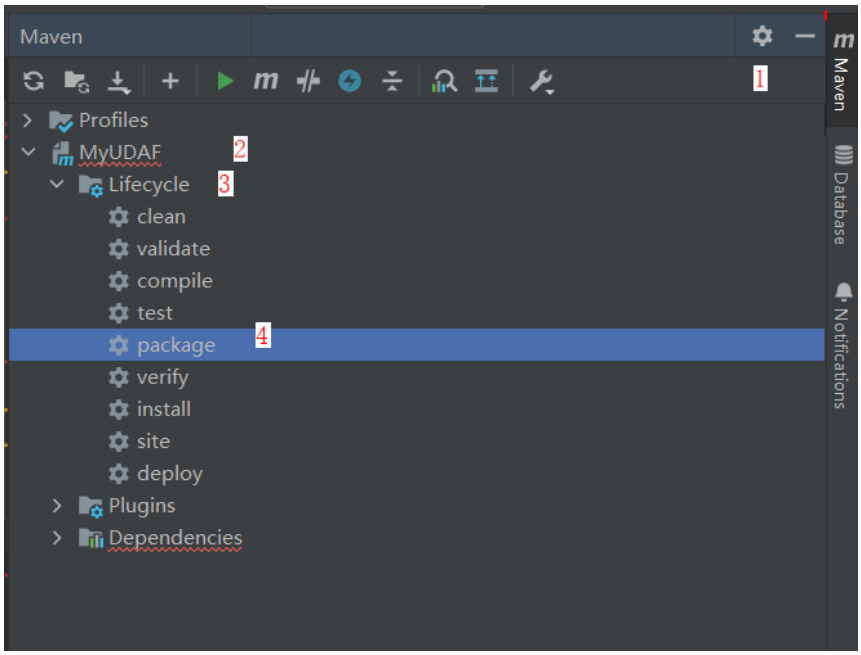


3. 编写调试完成代码后，通过 IntelliJ IDEA 工具编译代码并导出 Jar 包。

   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1-16 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。

```
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;
```

1.5 使用 Beeline 提交 Spark SQL 作业

DLI Beeline 简介

DLI Beeline 是一个用于连接 DLI 服务的客户端命令行交互工具，该工具基于 DLI JDBC 实现，提供 SQL 命令交互和批量 SQL 脚本执行的功能。

准备工作

在使用 DLI Beeline 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云资源的访问。

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

   目前包括角色（粗粒度授权）和策略（细粒度授权）。

2. 创建队列。在“队列类型”中选择“SQL 队列”，即 SQL 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。

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 向管理员获取 SDK 驱动包下载地址。

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

使用 DLI Beeline 连接服务端

使用 DLI Beeline 的机器安装 JDK 1.8 或以上版本并配置环境变量，推荐在 Linux 环境下使用 Beeline 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 Beeline 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，将“connection.properties.template”重命名成“connection.properties”，并配置连接参数。

步骤 4 进入执行脚本 bin 目录，启动 beeline 脚本，执行连接命令即可执行 SQL 语句，如下所示：

```
%bin/beeline
Start Beeline
Welcome to DLI service !
beeline> !connect
Connecting from the default connection.properties
Connecting to jdbc:dli://dli.xxx.com/8fc20d97a4444cafba3c3a8639380003
Connected to: DLI service
jdbc:dli://dli.xxx.. (not set)> show databases;
+--------------------+
|   databaseName    |
+--------------------+
| bjhk              |
| db_xd             |
| dimensions_adgame  |
| odbc_db           |
| sdk_db            |
| tpch_csv_1024      |
| tpch_csv_4g_new    |
| tpchnewtest       |
| tpchtest          |
| xunjian           |
+--------------------+
10 rows selected (0.338 seconds)
```

用户也可以在启动 beeline 脚本时通过命令行选项设置连接参数，如下所示，如果连接参数不全，Beeline 会提示补全相关信息。

```
%bin/beeline -u 'jdbc:dli://dli.xxx.com/8fc20d97a4444cafba3c3a8639380003?
authenticationmode=aksk'
Start Beeline
Connecting to
jdbc:dli://dli.xxx.com/8fc20d97a4444cafba3c3a8639380003?usehttpproxy=true;proxyhost
=10.186.60.154;proxyport=3128;authenticationmode=aksk
Enter region name: xxx
Enter service name: DLI
Enter access key(AK): <real access key>
Enter secret key(SK): ************************************
Enter queue name: default
Connected to: DLI service
Welcome to DLI service !
jdbc:dli://dli.xxx.. (not set)>
```

----结束

DLI Beeline 支持的命令

DLI Beeline 支持一系列命令，每个命令以“!” 这个符号开始，例如“!connect”。具体请参考表 1-8。

表1-8 DLI Beeline 支持的命令

命令	描述
!connect	通过输入连接参数的方式连接到 DLI 服务，若不输入参数，则加载默认

命令	描述
	的“connection.properties”文件连接。
!help	打印命令行的帮助文档。
!history	展示命令行执行历史。
!outputformat	设置查询结果的输出格式，支持table,vertical,csv2,dsv,tsv2,xmlattr,xmlelements，每一种格式的具体说明详参见查询输出格式。
!properties	通过加载“connection.properties”文件连接到 DLI 服务。该功能与!connect 相同，但是可以指定连接配置文件。
!quit	退出 beeline 会话。
!run	执行一个 SQL 脚本。使用方法： ***!run <scriptfile>***
!save	将当前的会话属性保存至 beeline.properties 中，下次开启 beeline 时会自动加载这些属性。
!script	将执行的命令保存至一个文件中。例如： ***!script /tmp/mysession.script*** 执行该语句之后，后续的命令将被保存至“/tmp/mysession.script”。 再次执行!script，结束脚本记录。执行如下语句，将会重新执行记录下来的命令。 ***!run /tmp/mysession.script***
!set	设置 Beeline 变量，例如： ***!set color true*** !set 后面不接参数则显示所有变量值。
!sh	执行一个 Shell 脚本。例如： ***!sh <shellscript>***
!sql	显性地执行一条 SQL 语句，beeline 中不带命令的语句默认会转换成!sql 命令，sql 语句必须以分号结尾。例如： ***!sql <sql>***
!dliconf	查看 DLI 自定义配置。

DLI Beeline 支持的命令行选项

DLI Beeline 支持的启动命令行选项请参考表 1-9。

表1-9 DLI Beeline 支持的启动命令行选项

命令行选项	描述
-u <database URL>	连接 DLI JDBC 的 url，其中 url 需要采用单引号括起来。使用方式： **beeline －u db_URL**
-e <query>	需要执行的 SQL 语句,可以输入多条语句，以分号间隔，语句需要采用单引号括起来。
-f <file>	需要执行的脚本文件。
--dliconf property=value	待设置的 DLI 属性。
--property-file=<property-file>	通过指定的方式获取连接属性文件并连接到 DLI 服务。
--help	打印命令行选项帮助。

查询输出格式

DLI Beeline 支持多种查询结果输出格式，输出格式可以通过!outputformat 指定。DLI Beeline 支持的输出格式包括：table，vertical，csv2，dsv，tsv2，xmlattr，xmlelements。

- **table**

  table 格式输出的结果以表的形式展示，例如：

  !outputformat table

  select id, value, comment from test_table;

```
+-----+---------+-----------------+
| id  |  value  |     comment     |
+-----+---------+-----------------+
| 1   | Value1  | Test comment 1  |
| 2   | Value2  | Test comment 2  |
| 3   | Value3  | Test comment 3  |
+-----+---------+-----------------+
```

- **vertical**

  以行为单元组织数据，每一个属性以 key-value 的形式展示，例如：

  !outputformat vertical

  select id, value, comment from test_table;

```
id      1
value    Value1
comment  Test comment 1
id      2
value    Value2
comment  Test comment 2
id      3
```

```
value    Value3
comment  Test comment 3
```

- **csv2**

  以纯文本形式存储表格数据（数字和文本），使用逗号（,）作为分隔符，例如：

  !outputformat csv2

  select id, value, comment from test_table;

```
id,value,comment
1,Value1,Test comment 1
2,Value2,Test comment 2
3,Value3,Test comment 3
```

- **dsv**

  每一行储存一条记录， 每条记录的各个字段间以制表符作为分隔，例如：

  !outputformat dsv

  select id, value, comment from test_table;

```
id|value |comment
1 |Value1|Test comment 1
2 |Value2|Test comment 2
3 |Value3|Test comment 3
```

- **tsv2**

  每一行储存一条记录， 每条记录的各个字段间以空格作为分隔，例如：

  !outputformat tsv2

  select id, value, comment from test_table;

```
id value     comment
1  Value1Test comment 1
2  Value2Test comment 2
3  Value3Test comment 3
```

- **xmlattr**

  用于在 SQL 查询返回的 XML 元素中设置属性的函数，例如：

  !outputformat xmlattr

  select id, value, comment from test_table;

```
<resultset>
  <result id="1" value="Value1" comment="Test comment 1"/>
  <result id="2" value="Value2" comment="Test comment 2"/>
  <result id="3" value="Value3" comment="Test comment 3"/>
</resultset>
```

- **xmlelements**

  将一个关系值转换为 XML 元素的函数，格式为<elementName>值</elementName>，例如：

  !outputformat xmlelements

  select id, value, comment from test_table;

```
<resultset>
  <result>
    <id>1</id>
    <value>Value1</value>
    <comment>Test comment 1</comment>
  </result>
```

```
  <result>
    <id>2</id>
    <value>Value2</value>
    <comment>Test comment 2</comment>
  </result>
  <result>
    <id>3</id>
    <value>Value3</value>
    <comment>Test comment 3</comment>
  </result>
</resultset>
```

1.6 使用 JDBC 提交 Spark SQL 作业

1.6.1 获取服务端连接地址

操作场景

DLI 支持在互联网环境下连接服务端进行数据查询操作。首先，需要根据如下指导获取连接信息，包括了 Endpoint 和项目编号。

操作步骤

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。

在请向管理员获取 DLI 对应的 Endpoint；在云页面上方菜单栏，单击用户名，然后在“我的凭证”页面获取项目编号。

1.6.2 下载 JDBC 驱动包

操作场景

JDBC 用于连接 DLI 服务，您可以在 DLI 管理控制台下载驱动文件。

操作步骤

步骤 1 登录 DLI 管理控制台。

步骤 2 向管理员获取 SDK 驱动包下载地址。

步骤 3 在“DLI SDK DOWNLOAD”页面，选择相应驱动下载。

- JDBC 驱动包

  例如单击“dli-jdbc-1.1.2”即可下载 1.1.2 版本 JDBC 驱动包。

  说明

  JDBC 驱动包命名为“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。

----结束

1.6.3 认证

操作场景

使用 JDBC 建立 DLI 驱动连接时，需要对用户进行认证鉴权。

操作步骤

目前 JDBC 支持两种认证鉴权方式，Access Key/Secret Key (AK/SK)和 Token，您选择其中一种方式进行认证即可。云账号认证，可以支持使用云账号和 IAM 子用户两种登录方式，与登录云控制台方式类似。

- （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。
- 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***，请向管理员获取终端节点信息，获取命令中 IAM 的 Endpoint。

     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。

1.6.4 使用 JDBC 提交作业

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。

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 1-10 所示。

表1-10 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云资源的访问。

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

   目前包括角色（粗粒度授权）和策略（细粒度授权）。

2. 创建队列。在“队列类型”中选择“SQL 队列”，即 SQL 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。

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 1.6.2 下载 JDBC 驱动包章节，获取 DLI JDBC 驱动包“dli-jdbc-<version>.zip”，解压，获得“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“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 1.6.3 认证。

步骤 5 使用 Class.forName（）加载 DLI JDBC 驱动程序。

Class.forName("com.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 1-11 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 1-12。

表1-11 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。 向管理员获取 DLI 对应的 Endpoint，从云“用户名”>“我的凭证”页面获取项目编号。

参数	描述
	• “？”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1-12 属性项

属性项	必须配置	默认值	描述
queuename	是	-	DLI 服务的队列名称。
databasename	否	-	数据库名称。
authenticationmode	否	token	身份认证方式，当前支持两种：token 或 aksk。
accesskey	是	-	AK/SK 认证密钥，获取方式请参考 1.6.3 认证。
secretkey	是	-	AK/SK 认证密钥，获取方式请参考 1.6.3 认证。
servicename	authentication mode=aksk 时必须配置	-	服务名称，即“dli”。
token	authentication mode=token 时必须配置	-	Token 认证，认证方式请参考 1.6.3 认证。
charset	否	UTF-8	JDBC 编码方式。
usehttpproxy	否	false	是否使用访问代理。
proxyhost	usehttpproxy=true 时必须配置	-	访问代理 host。
proxyport	usehttpproxy=true 时必须配置	-	访问代理端口。
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。
jobtimeout	否	300	提交作业终止时间，单位：秒。

属性项	必须配置	默认值	描述
iam.endpoint	否，默认根据 regionName 自动拼接	-	-
obs.endpoint	否，默认根据 regionName 自动拼接	-	-
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。 当获取查询结果时小于等于该预置值，可以调用 getJobResult 获取结果。 当查询结果大于该预置值，您需要手动导出结果来返回查询结果。

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

```
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.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", "<real ak>");
            info.setProperty("secretkey", "<real 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();
            }
        }
    }
}
```

开启重试功能

开启 JDBC 重试功能，执行查询操作失败时系统会进行重试。

说明

- 为避免重复数据插入等操作，非查询语句不进行重试。
- 1.1.5 及以上版本 JDBC 驱动包具有该功能。如果需要使用该功能，请获取最新版本 JDBC 驱动包。

开启重试功能需在 Info 参数中添加如表 1-13 所示属性项。

表1-13 重试功能属性项

属性项	必须配置	默认值	描述
USE_RETRY_KEY	是	false	是否开启重试。设置为“true”，表示开启重试。
RETRY_TIMES_KEY	是	3000	重试时间间隔（毫秒）。建议设置为30000ms。
RETRY_INTERVALS_KEY	是	3	重试次数。建议设置为3~5次。

设置 JDBC 配置项参数，开启重试功能，创建链接，示例如下：

```
import com.xxx.dli.jdbs.utils.ConnectionResource;//引入“ConnectionResource”,请按需修改类别名称
import java.sql.*;
import java.util.Properties;

public class DLIJdbcDriverExample {

   private static final String X_AUTH_TOKEN_VALUE = "<realtoken>";
   public static void main(String[] args) throws ClassNotFoundException,
SQLException {
       Connection conn = null;
       try {
          Class.forName("com.dli.jdbc.DliDriver");
          String url =
"jdbc:dli://<endpoint>/<projectId>?databasename=db1;queuename=testqueue";
          Properties info = new Properties();
          info.setProperty("token", X_AUTH_TOKEN_VALUE);
          info.setProperty(ConnectionResource.USE_RETRY_KEY, "true"); //开启重试
          info.setProperty(ConnectionResource.RETRY_TIMES_KEY, "30000");// 重试间隔ms
          info.setProperty(ConnectionResource.RETRY_INTERVALS_KEY, "5");// 重试次数
          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("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();
         }
      }
   }
}
```

1.6.5 JDBC API 参考

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)

1.7 在 Spark SQL 作业中使用地理空间函数

1.7.1 地理空间查询基本概念

地理空间数据概念

地理空间数据又称为几何数据，可用来表示物体的位置、形态、大小分布等各方面的信息，是对现实世界中存在的具有定位意义的事物和现象的定量描述。

通常，地理空间数据以点、线、面、体的形式表示。通过对地理空间数据的查询，可以获得被查询对象的面积、长度、空间关系等。

DLI 支持的地理空间数据类型

- Point（点）
- LineString（线）
- Polygon（面）
- MultiPoint（多点）
- MultiLineString（多线）
- MultiPolygon（多面）

说明

上述数据类型都称为一种特定的 Geometry 类型。

应用场景

地理空间查询用于统计某空间范围内兴趣点的个数，检查两个区域是否重叠、两个地点之间的距离等。

1.7.2 DLI 支持的地理空间查询函数分类

DLI 集成了开源地理空间套件 Geomesa 的空间查询函数接口，这些函数可以分为：

- 地理构造函数（Geometry Constructors）
- 地理访问函数（Geometry Accessors）
- 地理转换函数（Geometry Cast）
- 地理编辑函数（Geometry Editors）
- 地理输出函数（Geometry Outputs）
- 地理关系函数（Spatial Relationships）
- 地理处理函数（Geometry Processing）

本章节将以一个示例展示空间查询函数的使用方式。

1.7.3 地理空间查询准备工作

1. 创建 DLI 表 geotbl：

   create table geotbl(id String,name String,loadtime date,location String)

2. 向表中插入数据若干，其中 location 字段插入以 WKT 字符串表示的空间数据坐标串。例如：

   insert into geotbl select '1','amy','2015-05-01','POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))'

   insert into geotbl select '2','amy','2015-05-01','POLYGON ((60 10,70 60,80 0,60 10))'

   insert into geotbl select '3','amy','2015-05-01','MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))'

   insert into geotbl select '4','amy','2015-05-01','POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10),(20 30, 35 35, 30 20, 20 30))'

   insert into geotbl select '5','amy','2015-05-01','LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)'

   insert into geotbl select '6','amy','2015-05-01','POINT (11 22)'

   insert into geotbl select '7','amy','2015-05-01','MULTILINESTRING ((30 10, 40 40, 20 40, 10 20, 30 10),(30 10, 40 40, 20 40, 10 20, 30 10))'

   insert into geotbl select '8','amy','2015-05-01','MULTIPOINT ((11 22),(10 22))'

后续空间查询的示例将基于这个表进行展示。

说明

当前版本不支持直接创建有空间类型字段的表和空间数据批量导入。

1.7.4 地理构造函数

地理构造函数可以实现从 WKT、WKB、GeoHash 编码等输入中构造出 Geometry。

st_geomFromGeoHash

```
Geometry st_geomFromGeoHash(String geohash, Int prec)
```

返回与 Geohash 字符串 geohash（base-32 编码）对应的边界框的 Geometry，其精度为 prec 位。 有关 GeoHashes 的更多信息，请参阅 Geohash。

示例：

- 查询命令：

  select st_astext(st_geomFromGeoHash('ssf17',25))

- 查询结果：

  ```
  POLYGON ((25.4443359375 26.9384765625, 25.4443359375 26.982421875, 25.48828125 26.982421875, 25.48828125 26.9384765625, 25.4443359375 26.9384765625))
  ```

st_box2DFromGeoHash

```
Geometry st_box2DFromGeoHash(String geohash, Int prec)
```

st_geomFromGeoHash 的别称，示例同 st_geomFromGeoHash。

st_geomFromWKB

```
Geometry st_geomFromWKB(Array[Byte] wkb)
```

从给定的已知文本标记语言的二进制表示（WKB）创建 Geometry。

示例：

- 查询命令：

  select st_astext((st_geomFromWKB(st_asBinary(st_geomFromText(location))))) from geotbl where id='3'

- 查询结果：

  ```
  MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))
  ```

st_geomFromWKT

```
Geometry st_geomFromWKT(String wkt)
```

根据给定的已知文本标记语言（WKT）创建 Geometry。

st_geomFromText

```
Geometry st_geomFromText(String wkt)
```

st_geomFromWKT 的别称。

示例：

- 查询命令：

  select st_astext((st_geomFromText(location))) from geotbl where id='1'

- 查询结果：

  ```
  POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
  ```

st_geometryFromText(String wkt)

```
Geometry st_geometryFromText(String wkt)
```

st_geomFromWKT 的别称。

示例：

- 查询命令：

  select st_astext((st_geometryFromText(location))) from geotbl where id='5'

- 查询结果：

  ```
  LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)
  ```

st_lineFromText

```
LineString st_lineFromText(String wkt)
```

从给定的 WKT 创建 LineString。

示例：

- 查询命令：

  select st_astext((st_lineFromText(location))) from geotbl where id='5'
- 查询结果：

```
LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)
```

st_mLineFromText

```
MultiLineString st_mLineFromText(String wkt)
```

创建与给定 WKT 相对应的 MultiLineString。

示例：

- 查询命令：

  select st_astext((st_mLineFromText(location))) from geotbl where id='7'
- 查询结果：

```
MULTILINESTRING ((30 10, 40 40, 20 40, 10 20, 30 10), (30 10, 40 40, 20 40, 10 20, 30 10))
```

st_mPointFromText

```
MultiPoint st_mPointFromText(String wkt)
```

创建与给定 WKT 相对应的 MultiPoint。

示例：

- 查询命令：

  select st_astext((st_mPointFromText(location))) from geotbl where id='8'
- 查询结果：

```
MULTIPOINT ((11 22), (10 22))
```

st_mPolyFromText

```
MultiPolygon st_mPolyFromText(String wkt)
```

创建与给定 WKT 相对应的 MultiPolygon。

示例：

- 查询命令：

  select st_astext((st_mPolyFromText(location))) from geotbl where id='3'
- 查询结果：

```
MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))
```

st_makeBBOX

```
Geometry st_makeBBOX(Double lowerX, Double lowerY, Double upperX, Double upperY)
```

创建具有给定边界的边界框的 Geometry。

示例：

- 查询命令：

  select st_astext((st_makeBBOX(10,20,10,20)))

- 查询结果：

```
POINT (10 20)
```

st_makeBox2D

```
Geometry st_makeBox2D(Point lowerLeft, Point upperRight)
```

创建一个由给定 points 定义的 Geometry。

示例：

- 查询命令：

  select st_astext(st_makeBox2D(st_castToPoint(st_geomFromWKT('POINT (11 22)')),st_castToPoint(st_geomFromWKT('POINT (10 20)'))))

- 查询结果：

```
POLYGON ((10 20, 10 22, 11 22, 11 20, 10 20))
```

st_makePoint

```
Point st_makePoint(Double x, Double y)
```

创建一个带有 x 和 y 坐标的 point。

示例：

- 查询命令：

  select st_astext(st_makePoint(1,2))

- 查询结果：

```
POINT (1 2)
```

st_makePointM

```
Point st_makePointM(Double x, Double y, Double m)
```

创建一个带有 x，y 和 m 坐标的 point。

示例：

- 查询命令：

  select st_astext(st_makePointM(1,2,2))

- 查询结果：

```
POINT (1 2)
```

st_makePolygon

```
Polygon st_makePolygon(LineString shell)
```

创建由给定 LineString shell 形成的 Polygon，该 LingString 首尾必须是闭合的。

示例：

- 查询命令：

  select st_astext(st_makePolygon(st_castToLineString(st_geomFromWKT('LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)'))))

- 查询结果：

```
POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
```

st_point

```
Point st_point(Double x, Double y)
```

返回具有给定坐标值的 point。 这是 st_makePoint 的 OGC 别名。

示例：

- 查询命令：

  select st_astext(st_point(1,2))

- 查询结果：

```
POINT (1 2)
```

st_pointFromGeoHash

```
Point st_pointFromGeoHash(String geohash, Int prec)
```

返回由 Geohash 字符串 geohash（base-32 编码）定义的边界框的几何对象中心处的 point，其精度为 prec 位。 有关 Geohash 的更多信息，请参见 Geohash。

示例：

- 查询命令：

  select st_astext(st_pointFromGeoHash('s5zv4',25))

- 查询结果：

```
POINT (11.00830078125 21.99462890625)
```

st_pointFromText

```
Point st_pointFromText(String wkt)
```

创建与给定 WKT 相对应的 point。

示例：

- 查询命令：

  select st_astext(st_pointFromText('POINT (1 2)'))

- 查询结果：

```
POINT (1 2)
```

st_pointFromWKB

```
Point st_pointFromWKB(Array[Byte] wkb)
```

创建与给定 WKB 相对应的 point。

示例：

- 查询命令：

  select st_astext((st_pointFromWKB(st_asBinary(st_geomFromText(location))))) from geotbl where id='6'

- 查询结果：

```
POINT (11 22)
```

st_polygon

```
Polygon st_polygon(LineString shell)
```

创建由给定 LineString shell 形成的 polygon，且该 polygon 必须是闭合的。

示例：

- 查询命令：

  select st_astext(st_polygon(st_castToLineString(st_geomFromWKT('LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)'))))

- 查询结果：

```
POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
```

st_polygonFromText

```
Polygon st_polygonFromText(String wkt)
```

创建与给定 WKT 相对应的 polygon。

示例：

- 查询命令：

  select st_astext(st_polygonFromText('POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))'))

- 查询结果：

```
POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
```

1.7.5 地理访问函数

st_boundary

```
Geometry st_boundary(Geometry geom)
```

返回 geom 边界的 Geometry。如果 geom 为空，则返回空的 Geometry。

示例：

- 查询命令：

  select st_astext(st_boundary(st_geomFromWKT(location))) from geotbl where id='1'

- 查询结果：

```
LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)
```

st_coordDim

```
Int st_coordDim(Geometry geom)
```

返回 Geometry geom 坐标的维数。

示例：

- 查询命令：

  select st_coordDim(st_boundary(st_geomFromWKT(location))) from geotbl where id='2'

- 查询结果：

  ```
  2
  ```

st_dimension

```
Int st_dimension(Geometry geom)
```

返回此 Geometry 对象的固有维度，该维度必须小于或等于坐标维度。

示例：

- 查询命令：

  select st_dimension(st_boundary(st_geomFromWKT(location))) from geotbl where id='2'

- 查询结果：

  ```
  1
  ```

st_envelope

```
Geometry st_envelope(Geometry geom)
```

返回表示 geom 边界框的 Geometry。

示例：

- 查询命令：

  select st_astext(st_envelope(st_geomFromWKT(location))) from geotbl where id='1'

- 查询结果：

  ```
  POLYGON ((10 10, 10 40, 40 40, 40 10, 10 10))
  ```

st_exteriorRing

```
LineString st_exteriorRing(Geometry geom)
```

返回 geom 外部环的 LineString；如果 geom 不是 Polygon，则返回 null。

示例：

- 查询命令：

  select st_astext(st_exteriorRing(st_geomFromWKT(location))) from geotbl where id='1'

- 查询结果：

  ```
  LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)
  ```

st_geometryN

```
Geometry st_geometryN(Geometry geom, Geometry n)
```

如果 geom 是 GeometryCollection，则返回 geom 的第 n 个 Geometry（从 1 开始的索引）;如果不是，则返回 geom。

示例：

- 查询命令：

  select st_astext(st_geometryN(st_geomFromWKT(location),1)) from geotbl where id='1'

- 查询结果：

```
POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
```

st_interiorRingN

```
Geometry st_interiorRingN(Geometry geom, Int n)
```

返回 geom 的第 n 个内部 LineString 环。 如果 geom 不是 Polygon 或给定的 n 超出范围，则返回 null。

示例：

- 查询命令：

  select st_astext(st_interiorRingN(st_geomFromWKT(location),1)) from geotbl where id='4'

- 查询结果：

```
LINESTRING (20 30, 35 35, 30 20, 20 30)
```

st_isClosed

```
Boolean st_isClosed(Geometry geom)
```

如果 geom 是 LineString 或 MultiLineString 并且其起点和终点重合，则返回 true。 所有其他 Geometry 类型返回 true。

示例：

- 查询命令：

  select st_isClosed(st_geomFromWKT(location)) from geotbl where id='6'

- 查询结果：

```
TRUE
```

st_isCollection

```
Boolean st_isCollection(Geometry geom)
```

如果 geom 是 GeometryCollection，则返回 true。

示例：

- 查询命令：

  select st_isCollection(st_geomFromWKT(location)) from geotbl where id='7'

- 查询结果：

```
TRUE
```

st_isEmpty

```
Boolean st_isEmpty(Geometry geom)
```

如果 geom 为空，则返回 true。

示例：

- 查询命令：

  select st_isEmpty(st_geomFromWKT(location)) from geotbl where id='1'
- 查询结果：

```
TRUE
```

st_isRing

```
Boolean st_isRing(Geometry geom)
```

如果 geom 是 LineString 或 MultiLineString 并且闭环且是简单的（简单的定义详见 st_isSimple），则返回 true。

示例：

- 查询命令：

  select st_isRing(st_geomFromWKT(location)) from geotbl where id='1'
- 查询结果：

```
TRUE
```

st_isSimple

```
Boolean st_isSimple(Geometry geom)
```

如果 geom 没有异常几何点，例如自相交或自相切，则返回 true。

示例：

- 查询命令：

  select st_isSimple(st_geomFromWKT(location)) from geotbl where id='2'
- 查询结果：

```
TRUE
```

st_isValid

```
Boolean st_isValid(Geometry geom)
```

如果根据 OGC SFS 规范，Geometry 在拓扑上有效，则返回 true。

示例：

- 查询命令：

  select st_isValid(st_geomFromWKT(location)) from geotbl where id='3'

- 查询结果：

```
TRUE
```

st_numGeometries

```
Int st_numGeometries(Geometry geom)
```

如果 geom 是 GeometryCollection，则返回 Geometry 的个数。 否则，返回 1。

示例：

- 查询命令：

  select st_numGeometries(st_geomFromWKT(location)) from geotbl where id='3'
- 查询结果：

```
2
```

st_numPoints

```
Int st_numPoints(Geometry geom)
```

返回 geom 中的顶点数。

示例：

- 查询命令：

  select st_numPoints(st_geomFromWKT(location)) from geotbl where id='4'
- 查询结果：

```
9
```

st_pointN

```
Point st_pointN(Geometry geom, Int n)
```

如果 geom 是 LineString，则返回 geom 的第 n 个顶点作为 Point。 如果 n 是负值则从 LineString 的末尾向前计数。 如果 geom 不是 LineString，则返回 null。

示例：

- 查询命令：

  select st_astext(st_pointN(st_geomFromWKT(location),2)) from geotbl where id='5'
- 查询结果：

```
POINT (40 40)
```

st_x

```
Float st_x(Geometry geom)
```

如果 geom 是 Point，则返回该点的 X 坐标。

示例：

- 查询命令：

  select st_x(st_geomFromWKT(location)) from geotbl where id='6'

- 查询结果：

```
11
```

st_y

```
Float st_y(Geometry geom)
```

如果 geom 是 Point，则返回该点的 Y 坐标。

示例：

- 查询命令：

  select st_y(st_geomFromWKT(location)) from geotbl where id='6'
- 查询结果：

```
22
```

1.7.6 地理转换函数

st_castToLineString

```
LineString st_castToLineString(Geometry g)
```

将 Geometry g 转换为 LineString。

示例：

- 查询命令：

  select st_astext(st_castToLineString(st_geomFromWKT(location))) from geotbl where id='5'
- 查询结果：

```
LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)
```

st_castToPoint

```
Point st_castToPoint(Geometry g)
```

将 Geometry g 转换为 Point。

示例：

- 查询命令：

  select st_astext(st_castToPoint(st_geomFromWKT(location))) from geotbl where id='6'
- 查询结果：

```
POINT (11 22)
```

st_castToPolygon

```
Polygon st_castToPolygon(Geometry g)
```

将 Geometry g 转换为 Polygon。

示例：

- 查询命令：

  select st_astext(st_castToPolygon(st_geomFromWKT(location))) from geotbl where id='1'

- 查询结果：

```
POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
```

st_byteArray

```
Array[Byte] st_byteArray(String s)
```

使用 UTF-8 字符集将字符串 s 编码为字节数组。

示例：

- 查询命令：

  select st_byteArray(location) from geotbl where id='2'

- 查询结果：

```
UE9MWUdPTiAoKDYwIDEwLDcwIDYwLDgwIDAsNjAgMTApKQ==
```

1.7.7 地理编辑函数

st_translate

```
Geometry st_translate(Geometry geom, Double deltaX, Double deltaY)
```

返回由 deltaX 和 deltaY 转换 geom 时生成的 Geometry。

示例：

- 查询命令：

  select st_asText(st_translate(st_geomFromWKT(location),1,2)) from geotbl where id='4'

- 查询结果：

```
POLYGON ((36 12, 46 47, 16 42, 11 22, 36 12), (21 32, 36 37, 31 22, 21 32))
```

1.7.8 地理输出函数

st_asBinary

```
Array[Byte] st_asBinary(Geometry geom)
```

返回 WKB 表示的 Geometry geom。

示例：

- 查询命令：

  select st_asBinary(st_geomFromWKT(location)) from geotbl where id='1'

- 查询结果：

```
AAAAAAMAAAABAAAABUA+AAAAAAAAQCQAAAAAAABARAAAAAAAAEBEAAAAAAAAQDQAAAAAAABARAAAAAA
AAEAkAAAAAAAAQDQAAAAAAABAPgAAAAAAAEAkAAAAAAAA
```

st_asGeoJSON

```
String st_asGeoJSON(Geometry geom)
```

返回 GeoJSON 表示的 Geometry geom。

示例：

- 查询命令：

  select st_asGeoJSON(st_geomFromWKT(location)) from geotbl where id='3'

- 查询结果：

```
{"type":"MultiPolygon","coordinates":[[[[30,20],[45,40],[10,40],[30,20]]],[[[15
,5],[40,10],[10,20],[5,10],[15,5]]]]}
```

st_asLatLonText

```
String st_asLatLonText(Point p)
```

返回描述 Point p 的纬度和经度的 String，以度、分和秒为单位。 （这里假设 p 的坐标单位是纬度和经度。）

示例：

- 查询命令：

  select st_asLatLonText(st_castToPoint(st_geomFromWKT(location))) from geotbl where id='6'

- 查询结果：

```
22°0'0.000\"N 11°0'0.000\"E
```

st_asText

```
String st_asText(Geometry geom)
```

返回 WKT 表示的 Geometry geom。

st_geoHash

```
String st_geoHash(Geometry geom, Int prec)
```

返回 Geometry geom 内部点的 Geohash（以 base-32 表示），其中，prec 为编码精度。有关 Geohash 的更多信息，请参见 Geohash。

示例：

- 查询命令：

  select st_geoHash(st_geomFromWKT(location),25) from geotbl where id='1'

- 查询结果：

```
ssf17
```

1.7.9 地理关系函数

st_area

```
Double st_area(Geometry g)
```

如果 Geometry g 是面状几何对象，则以坐标参照系的平方单位返回其表面的面积（例如，degrees^2 for EPSG:4326）。 对于非面状几何对象（例如，点，非闭合 LineStrings 等），返回 0.0。

示例：

- 查询命令：

  select st_area(st_geomFromWKT(location)) from geotbl where id='1'
- 查询结果：

```
550.0
```

st_centroid

```
Point st_centroid(Geometry g)
```

返回 g 的几何中心点。

示例：

- 查询命令：

  select st_asText(st_centroid(st_geomFromWKT(location))) from geotbl where id='4'
- 查询结果：

```
POINT (27.40740740740741 28.765432098765434)
```

st_closestPoint

```
Point st_closestPoint(Geometry a, Geometry b)
```

返回 a 中最接近 b 的 Point。 即 a、b 之间最短连线的第 1 个点。

示例：

- 查询命令：

  select st_asText(st_closestPoint((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2')))
- 查询结果：

```
POINT (40 40)
```

st_contains

```
Boolean st_contains(Geometry a, Geometry b)
```

当且仅当没有 b 的点在 a 的外部时，返回 true，并且 b 至少有一个点位于 a 的内部。

示例：

- 查询命令：

  select st_contains((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))
- 查询结果：

```
FALSE
```

st_covers

```
Boolean st_covers(Geometry a, Geometry b)
```

如果 Geometry b 中没有点在 Geometry a 之外，则返回 true。

示例：

- 查询命令：

  select st_covers((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

  ```
  FALSE
  ```

st_crosses

```
Boolean st_crosses(Geometry a, Geometry b)
```

如果第一个 Geometry 与第二个 Geometry 存在一些相同但不全相同的 Point，则返回 true。

示例：

- 查询命令：

  select st_crosses((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

  ```
  FALSE
  ```

st_disjoint

```
Boolean st_disjoint(Geometry a, Geometry b)
```

如果 a 和 b 不是“空间相交”的，则返回 true；即，他们不共享任何空间。 相当于 NOT st_intersects（a，b）。

示例：

- 查询命令：

  select st_disjoint((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

  ```
  TRUE
  ```

st_distance

```
Double st_distance(Geometry a, Geometry b)
```

返回 a 和 b 之间的 2D 笛卡尔距离（例如，EPSG 的度数：4236）。

示例：

- 查询命令：

select st_distance((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

```
25.4950975679639
```

st_distanceSphere

```
Double st_distanceSphere(Geometry a, Geometry b)
```

返回 a 和 b 之间经度/纬度几何对象之间的近似最小球面距离。

示例：

- 查询命令：

  select st_distanceSphere((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

```
3284010.93490738
```

st_distanceSpheroid

```
Double st_distanceSpheroid(Geometry a, Geometry b)
```

返回 a 和 b 之间经度/纬度几何对象之间的最小 WGS84 椭球距离。

示例：

- 查询命令：

  select st_distanceSpheroid((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

```
3288016.93279476
```

st_equals

```
Boolean st_equals(Geometry a, Geometry b)
```

不考虑方向的条件下，如果给定的 Geometry a 和 b 在逻辑上是相同的，则返回 true。

示例：

- 查询命令：

  select st_equals((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

```
FALSE
```

st_intersects

```
Boolean st_intersects(Geometry a, Geometry b)
```

如果 a 和 b 在 2D 中空间相交（即共享空间的任何部分），则返回 true。 相当于 NOT st_disjoint（a，b）。

示例：

- 查询命令：

  select st_intersects((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

```
FALSE
```

st_length

```
Double st_length(Geometry geom)
```

返回线性几何的 2D 路径长度或面几何的周长（例如，EPSG 的度数：4236）。 对于其他几何类型（例如 Point），返回 0.0。

示例：

- 查询命令：

  select st_length(st_geomFromWKT(location)) from geotbl where id='4'

- 查询结果：

```
160.12062648706927
```

st_lengthSphere

```
Double st_lengthSphere(LineString line)
```

使用球形地球模型近似得到的 LineString 几何体的 2D 路径长度。 返回的长度以米为单位。 与 st_lengthSpheroid 的计算误差在 0.3%范围内，并且计算效率更高。

示例：

- 查询命令：

  select st_lengthSphere(st_castToLineString(st_geomFromWKT(location))) from geotbl where id='5'

- 查询结果：

```
1.0013773137296816E7
```

st_lengthSpheroid

```
Double st_lengthSpheroid(LineString line)
```

返回在 WGS84 球体上使用经度/纬度坐标定义的 LineString 几何体的 2D 路径长度。返回的长度以米为单位。

示例：

- 查询命令：

  select st_lengthSpheroid(st_castToLineString(st_geomFromWKT(location))) from geotbl where id='5'

- 查询结果：

```
1.0001465052274073E7
```

st_overlaps

```
Boolean st_overlaps(Geometry a, Geometry b)
```

如果 a 和 b 具有一些但不是所有的共同点，具有相同的维度，并且 a 和 b 的内部的交点与几何本身具有相同的维度，则返回 true。

示例：

- 查询命令：

  select st_overlaps((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

  ```
  FALSE
  ```

st_relate

```
String st_relate(Geometry a, Geometry b)
```

返回描述 a 和 b 的内部、边界和外部之间的交叉点的维度的 DE-9IM 3x3 交互矩阵模式。

示例：

- 查询命令：

  select st_relate((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

  ```
  FF2FF1212
  ```

st_relateBool

```
Boolean st_relateBool(Geometry a, Geometry b, String mask)
```

如果 DE-9IM 交互矩阵 mask 与从 st_relate（a，b）获得的交互矩阵模式匹配，则返回 true。

示例：

- 查询命令：

  select st_relateBool((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'),'FF2FF1212')

- 查询结果：

  ```
  TRUE
  ```

st_touches

```
Boolean st_touches(Geometry a, Geometry b)
```

如果 a 和 b 具有至少一个共同点，但它们的内部不相交，则返回 true。

示例：

- 查询命令：

select st_touches((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

```
FALSE
```

st_within

```
Boolean st_within(Geometry a, Geometry b)
```

如果几何体 a 完全位于几何体 b 内，则返回 true。

示例：

- 查询命令：

  select st_within((select st_geomFromWKT(location) from geotbl where id='1'),(select st_geomFromWKT(location) from geotbl where id='2'))

- 查询结果：

```
FALSE
```

1.7.10 地理处理函数

st_antimeridianSafeGeom

```
Geometry st_antimeridianSafeGeom(Geometry geom)
```

如果 geom 跨越 antimeridian，会尝试将 geom 转换为“antimeridian-safe”的等价形式（即输出的 Geometry 会被 BOX（-180 -90,180 90）覆盖）。 在某些情况下，此方法可能会失败，在这种情况下将返回输入的 geom，并将记录错误。

示例：

- 查询命令：

  select st_astext(st_antimeridianSafeGeom(st_geomFromWKT(location))) from geotbl where id='5'

- 查询结果：

```
LINESTRING (30 10, 40 40, 20 40, 10 20, 30 10)
```

st_idlSafeGeom

st_antimeridianSafeGeom 的别名。

示例：

- 查询命令：

  select st_astext(st_idlSafeGeom(st_geomFromWKT(location))) from geotbl where id='6'

- 查询结果：

```
POINT (11 22)
```

st_bufferPoint

```
Geometry st_bufferPoint(Point p, Double buffer)
```

返回覆盖 Point p 给定半径内所有点的 Geometry，其中半径以米为单位。

示例：

- 查询命令：

  select st_astext(st_bufferPoint(st_castToPoint(st_geomFromWKT(location)),0.1)) from geotbl where id='6'

- 查询结果：

```
10.999999429878716 22.000000784704625, 10.999999381731708 22.00000074735798,
10.999999336024723 22.00000070706185, 10.999999292938147 22.000000663975275,
10.99999925264202 22.00000061826829, 10.999999215295372 22.000000570121284,
10.999999181045595 22.000000519724267, 10.99999915002786 22.000000467276138,
10.999999122364573 22.000000412983884, 10.999999098164913 22.00000035706177,
10.999999077524384 22.000000299730495, 10.999999060524447 22.000000241216323,
10.999999047232189 22.00000018175018, 10.99999903770007 22.000000121566753,
10.999999031965709 22.000000060903556, 10.999999030051738 22,
10.999999031965709 21.999999939096444, 10.99999903770007 21.999999878433247,
10.999999047232189 21.99999981824982, 10.999999060524447 21.999999758783677,
10.999999077524384 21.999999700269505, 10.999999098164913 21.99999964293823,
10.999999122364573 21.999999587016116, 10.99999915002786 21.999999532723862,
10.999999181045595 21.999999480275733, 10.999999215295372 21.999999429878716,
10.99999925264202 21.99999938173171, 10.999999292938147 21.999999336024725,
10.999999336024723 21.99999929293815, 10.999999381731708 21.99999925264202,
10.999999429878716 21.999999215295375, 10.999999480275731 21.9999991810456,
10.99999953272386 21.99999915002786, 10.999999587016116 21.999999122364574,
10.99999964293823 21.999999098164913, 10.999999700269504 21.999999077524386,
10.999999758783677 21.99999906052445, 10.999999818249819 21.99999904723219,
10.999999878433249 21.99999903770007, 10.999999939096444 21.99999903196571, 11
21.99999903005174, 11.000000060903556 21.99999903196571, 11.000000121566751
21.99999903770007, 11.000000181750181 21.99999904723219, 11.000000241216323
21.99999906052445, 11.000000299730496 21.999999077524386, 11.00000035706177
21.999999098164913, 11.000000412983884 21.999999122364574, 11.00000046727614
21.99999915002786, 11.000000519724269 21.9999991810456, 11.000000570121284
21.999999215295375, 11.000000618268292 21.99999925264202, 11.000000663975277
21.99999929293815, 11.000000707061853 21.999999336024725, 11.00000074735798
21.99999938173171, 11.000000784704628 21.999999429878716, 11.000000818954405
21.999999480275733, 11.00000084997214 21.999999532723862, 11.000000877635427
21.999999587016116, 11.000000901835087 21.99999964293823, 11.000000922475616
21.999999700269505, 11.000000939475553 21.999999758783677, 11.000000952767811
21.99999981824982, 11.00000096229993 21.999999878433247, 11.000000968034291
21.999999939096444, 11.000000969948262 22))
```

st_convexHull

```
Geometry st_convexHull(Geometry geom)
```

聚合函数。 几何体的凸包表示包含聚合行中所有几何图形的最小凸面几何体。

2 Flink OpenSource SQL 作业开发指南

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。

> **⚠ 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。

整体作业开发流程

整体作业开发流程参考图 2-1。

图2-1 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“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、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：优先级选为：1，策略选为：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：优先级选为：1，策略选为：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。

```
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",
    "username" = "xxxxx",  --替换为 RDS MySQL 的实例的用户名
    "password" = "xxxxx",  --替换为 RDS MySQL 的实例的用户密码
    "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。

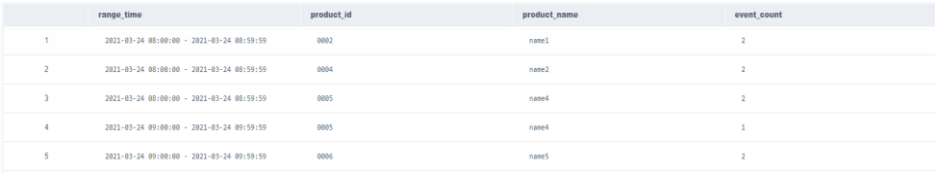
   发送样例数据如下：

```
{"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。



	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

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":"zhangsan", "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。

注意

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

2. 已创建 DWS 实例。

整体作业开发流程

整体作业开发流程参考图 2-2。

图2-2 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“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、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. 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：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 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：优先级选为：1，策略选为：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：优先级选为：1，策略选为：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。

```
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.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
  'username' = 'xxxx',--替换为 DWS 实例的用户名
```

```
  'password' = 'xxxx',--替换为 DWS 实例的用户密码
  '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。

   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":"zhangsan", "car_age":"10", "average_speed":"81",
   "total_miles":"230000"}
   ```

2. 连接已创建的 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
   ```

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。

> **⚠ 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。
   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 2-3。

图2-3 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 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“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、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：优先级选为：1，策略选为：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：优先级选为：1，策略选为：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。

- 创建 Kafka 源表，将 DLI 和 Kafka 数据源进行链接。

```
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。

   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"
        }
      }
    ]
  }
}
```


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。
2. 已创建 DWS 实例，

整体作业开发流程

整体作业开发流程参考图 2-4。

图2-4 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 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“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、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。
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：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：优先级选为：1，策略选为：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：优先级选为：1，策略选为：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。

```
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
  'username' = 'xxx',--RDS MySQL 实例的数据库用户名
  'password' = 'xxx',--RDS MySQL 实例的数据库用户密码
  '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.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
```

```
  'username' = 'xxx',--替换为 DWS 实例的用户名
  'password' = 'xxx',--替换为 DWS 实例的用户密码
  '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. 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
```

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。

   说明

   创建的 RDS Postgres 数据库版本不能低于 11。

2. 已创建 DWS 实例。

整体作业开发流程

整体作业开发流程参考图 2-5。

图2-5 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 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“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、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. 在 SQL 查询页输入以下语句，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。
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：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：优先级选为：1，策略选为：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：优先级选为：1，策略选为：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：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。

```
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
  'username' = 'xxxxx',--RDS Postgres 实例的数据库用户名
  'password' = 'xxxxx',-RDS Postgres 实例的数据库用户密码
  '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.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。
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
```

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。
2. 登录 DLI 控制台，创建 Flink SQL 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。

   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。

   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。

   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”。

   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。

3 Flink Jar 作业开发指南

3.1 流生态作业开发指引

概述

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。

流生态开发支持的数据格式

DLI Flink 作业支持如下数据格式：

Avro，Avro_merge，BLOB，CSV，EMAIL，JSON，ORC，Parquet，XML。

3.2 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。

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。“安全组规则”开通对应 UDP/TCP 端口。
2. 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”。
3. 创建 DLI 独享队列。
4. 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。
5. 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，请参考《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

- 确保已创建独享队列。
- 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。

  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）

使用方法

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。

步骤 3 配置作业信息。

表3-1 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。

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。

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。

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。

表3-2 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，。

名称	描述
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”。
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。 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")`
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”版本。

步骤 7 配置作业参数。

表3-3 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。
最大并行数	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。

名称	描述
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”：
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，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3-1 解析后的参数

- **怎样查看作业日志？**
  a. 在 FLink 作业列表中点击作业名称，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。

3.3 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。

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>
        <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.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.DateTi
meBucketAssigner;
import
org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.OnChec
kpointRollingPolicy;
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,从最老的数据开始消费，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();
            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3-4 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

参数名	具体含义	举例
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 3.2 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。

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	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

第一次提交 Spark 作业，需要先创建队列，例如创建名为“sparktest”的队列，队列类型选择为“通用队列”。

1. 在 DLI 管理控制台的左侧导航栏中，选择“队列管理”。
2. 单击“队列管理”页面右上角“创建队列”进行创建队列。
3. 创建名为“sparktest”的队列，队列类型选择为“通用队列”。创建队列详细介绍请参考《数据湖探索用户指南》>《创建队列》。
4. 单击“立即创建”，完成队列创建。

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}
```

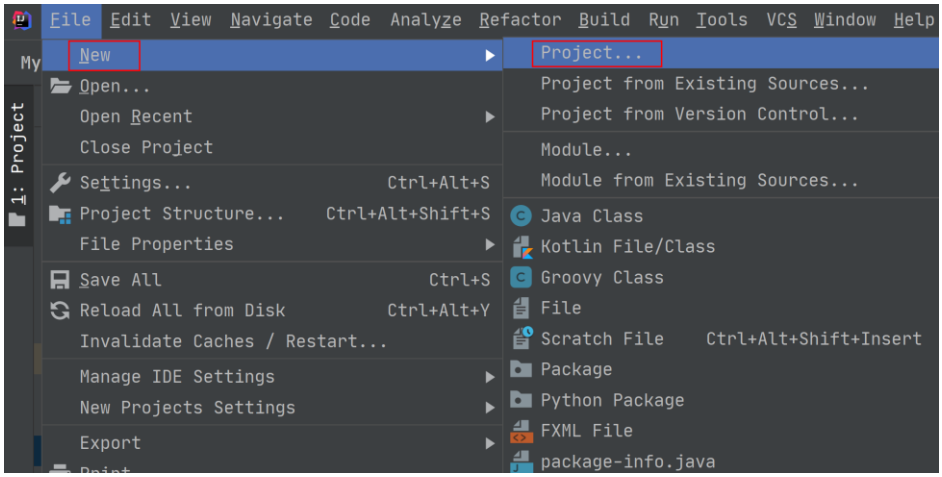
2. 进入 OBS 管理控制台，在“桶列表”下，单击已创建的 OBS 桶名称，本示例桶名为“dli-test-obs01”，进入“概览”页面。
3. 单击左侧列表中的“对象”，选择“上传对象”，将 people.json 文件上传到 OBS 桶根目录下。
4. 在 OBS 桶根目录下，单击“新建文件夹”，创建名为“result”的文件夹。
5. 单击“result”的文件夹，在“result”下单击“新建文件夹”，创建名为“parquet”的文件夹。

步骤 3：新建 Maven 工程，配置 pom 依赖

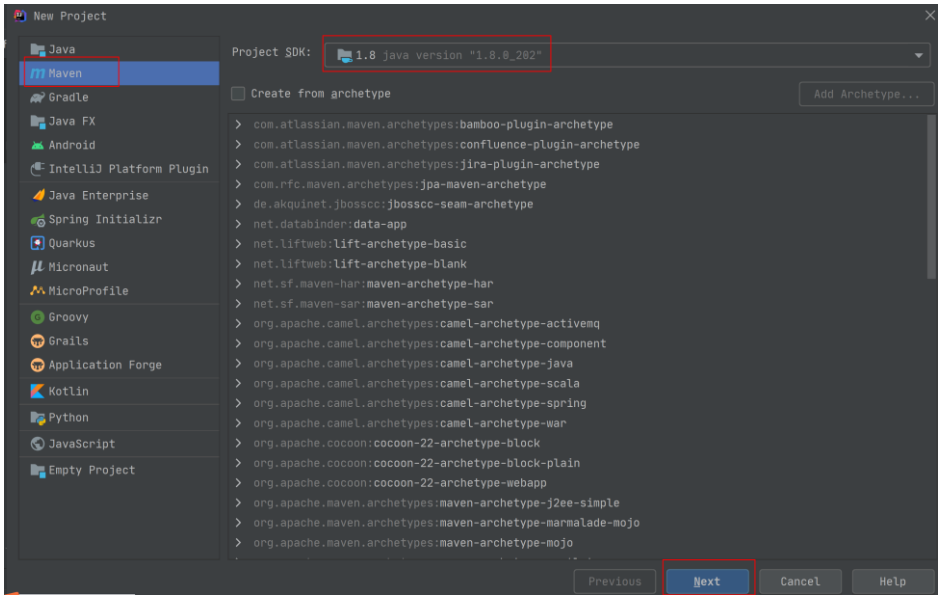
以下通过 IntelliJ IDEA 2020.2 工具操作演示。

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

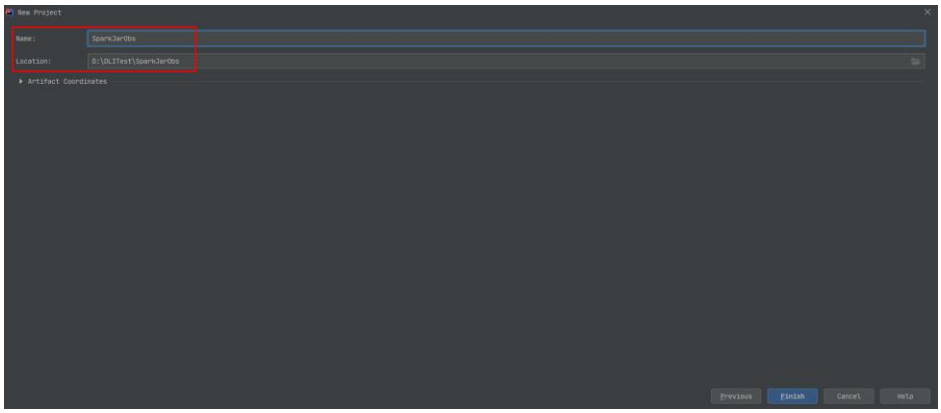
图4-2 新建 Project



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



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。

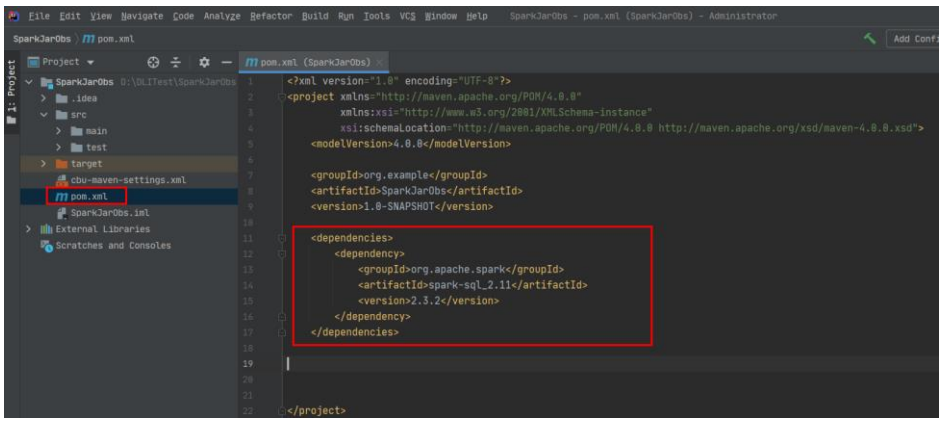


如上图所示，本示例创建 Maven 工程名为：SparkJarObs，Maven 工程路径为：“D:\DLITest\SparkJarObs”。

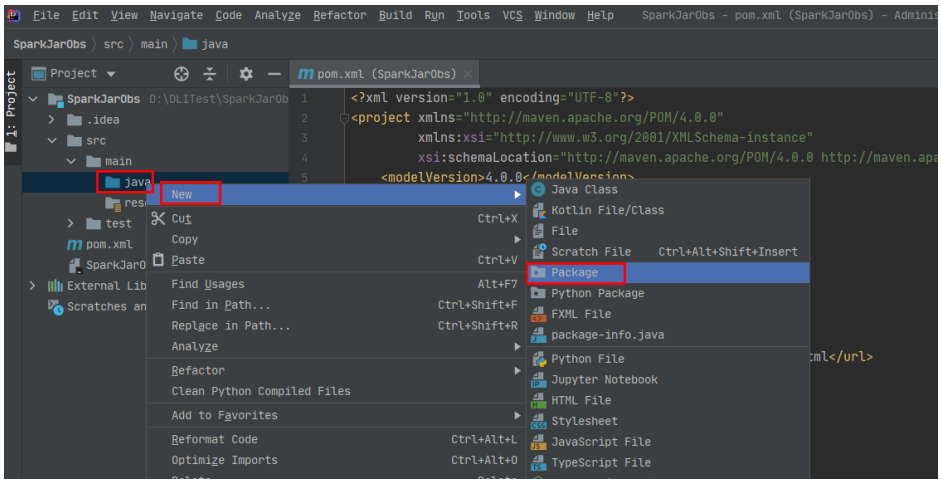
4. 在 pom.xml 文件中添加如下配置。

```
<dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>2.3.2</version>
        </dependency>
</dependencies>
```

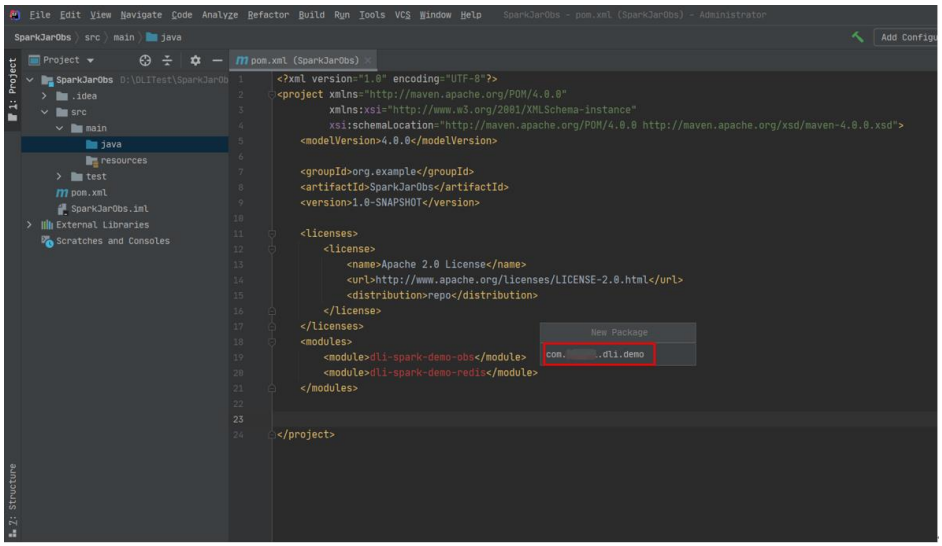
图4-3 修改 pom.xml 文件



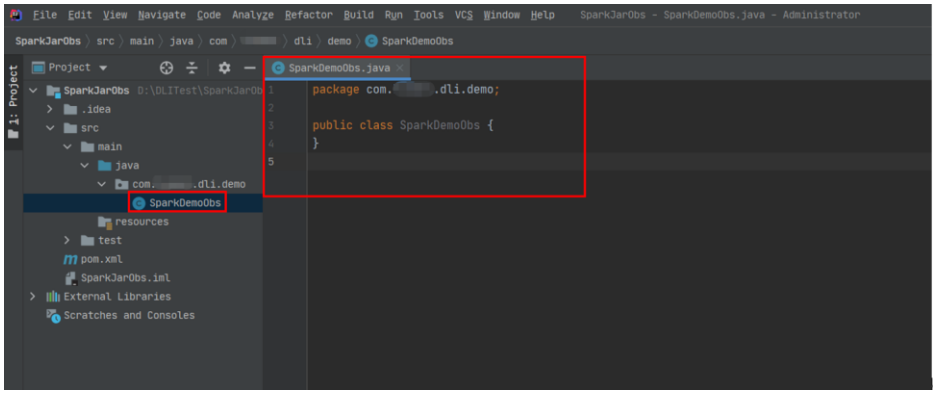
5. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建 Package 和类文件。



Package 根据需要定义，完成后回车。



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



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. 通过当前帐号的 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*"需要替换为帐号的 AK 值。
   - "spark.hadoop.fs.obs.secret.key"参数对应的值“yyy”需要替换为帐号的 SK 值。

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();
```

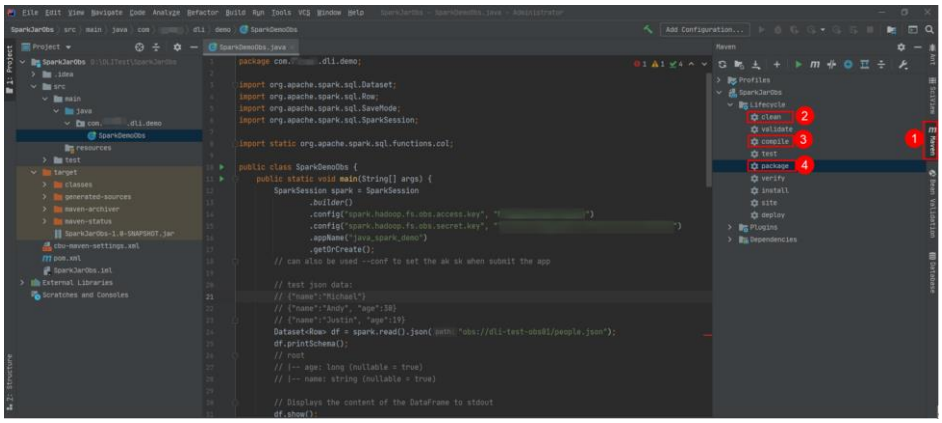
7. 关闭 SparkSession 会话 spark。

```
spark.stop();
```

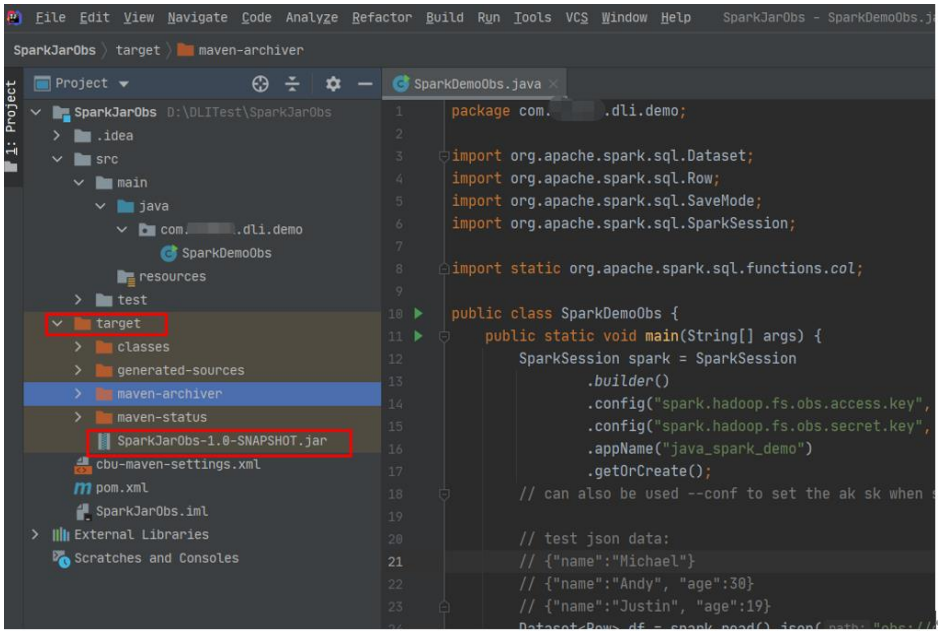
步骤 5：调试、编译代码并导出 Jar 包

1. 单击 IntelliJ IDEA 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。



   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”。



步骤 6：上传 Jar 包到 OBS 和 DLI 下

1. 登录 OBS 控制台，将生成的“SparkJarObs-1.0-SNAPSHOT.jar”Jar 包文件上传到 OBS 路径下。
2. 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。

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”。
   - 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。

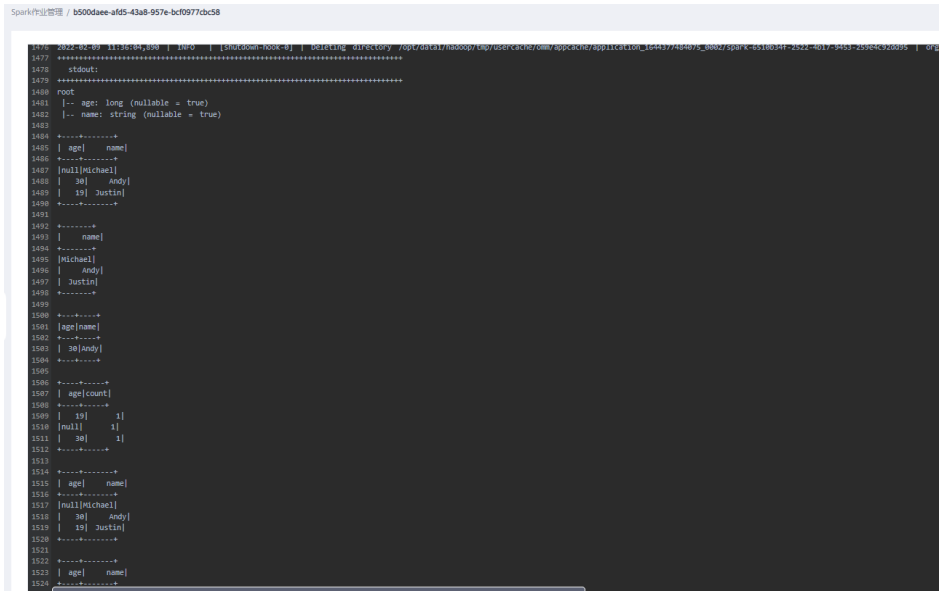
   其他参数可暂不选择。

   图4-4 创建 Spark Jar 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。

步骤 8：查看作业运行结果

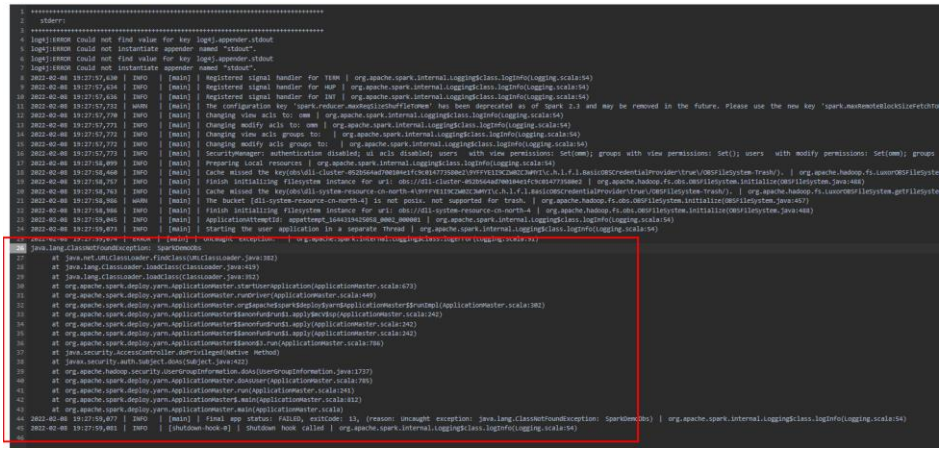
1. 在 Spark 作业管理界面显示已提交的作业运行状态。初始状态显示为“启动中”。
2. 如果作业运行成功则作业状态显示为“已成功”，单击“操作”列“更多”下的“Driver 日志”，显示当前作业运行的日志。

图4-5 “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。
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”。



   可以在“操作”列，单击“编辑”，修改“主类”参数为正确的：com.SparkDemoObs，单击“执行”重新运行该作业即可。

完整样例代码参考

```
package com.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。

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4-3 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 作业访问 DLI 元数据开发流程参考如下：

图4-6 Spark 作业访问 DLI 元数据开发流程

表4-4 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“spark.sql.warehouse.dir”的存储路径。
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		

序号	阶段	操作界面	说明
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	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

第一次提交 Spark 作业，需要先创建队列，例如创建名为“sparktest”的队列，队列类型选择为“通用队列”。

1. 在 DLI 管理控制台的左侧导航栏中，选择“队列管理”。
2. 单击“队列管理”页面右上角“创建队列”进行创建队列。
3. 创建名为“sparktest”的队列，队列类型选择为“通用队列”。创建队列详细介绍请参考《数据湖探索用户指南》>《创建队列》。
4. 单击“立即创建”，完成队列创建。

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
   ```

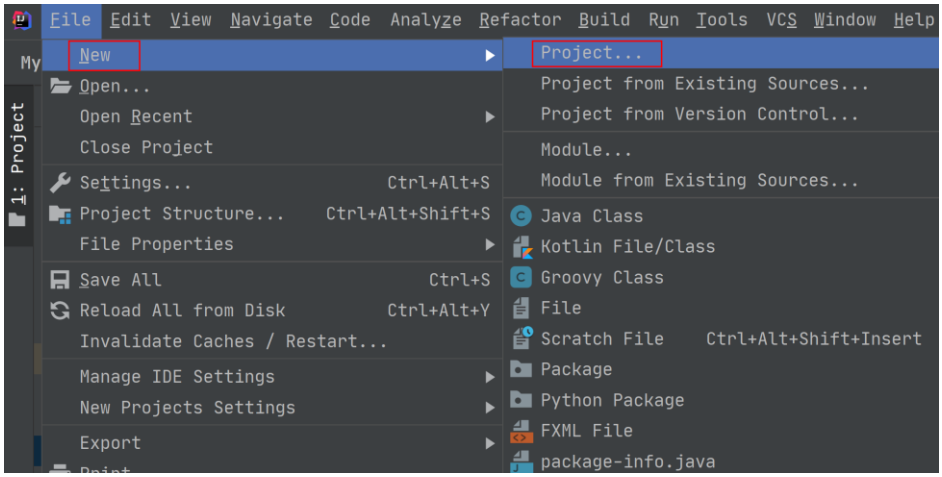
2. 进入 OBS 管理控制台，在“桶列表”下，单击已创建的 OBS 桶名称，本示例桶名为“dli-test-obs01”，进入“概览”页面。
3. 单击左侧列表中的“对象”，选择“上传对象”，将 testdata.csv 文件上传到 OBS 桶根目录下。
4. 在 OBS 桶根目录下，单击“新建文件夹”，创建名为“warehousepath”的文件夹。该文件夹路径用来存储 Spark 创建表的元数据信息“spark.sql.warehouse.dir”。

步骤 3：新建 Maven 工程，配置 pom 依赖

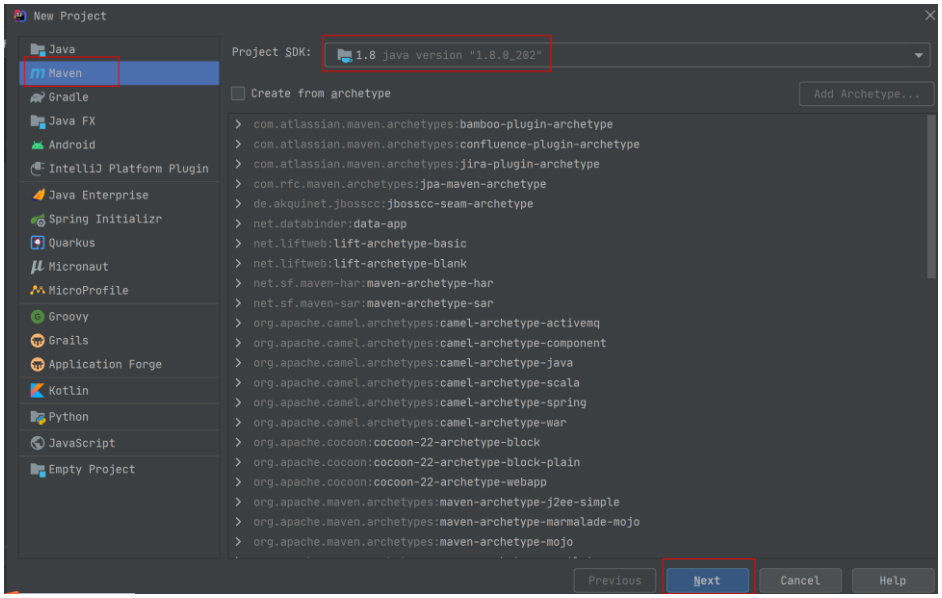
以下通过 IntelliJ IDEA 2020.2 工具操作演示。

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

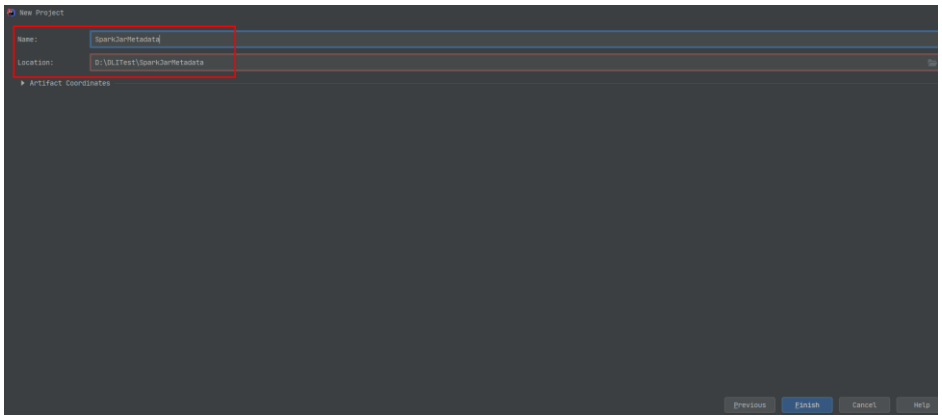
图4-7 新建 Project



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



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。

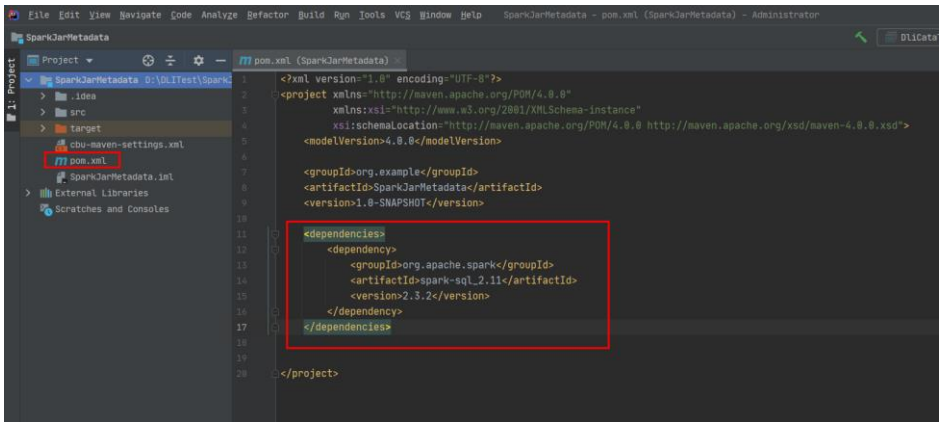


如上图所示，本示例创建 Maven 工程名为：SparkJarMetadata，Maven 工程路径为：“D:\DLITest\SparkJarMetadata”。

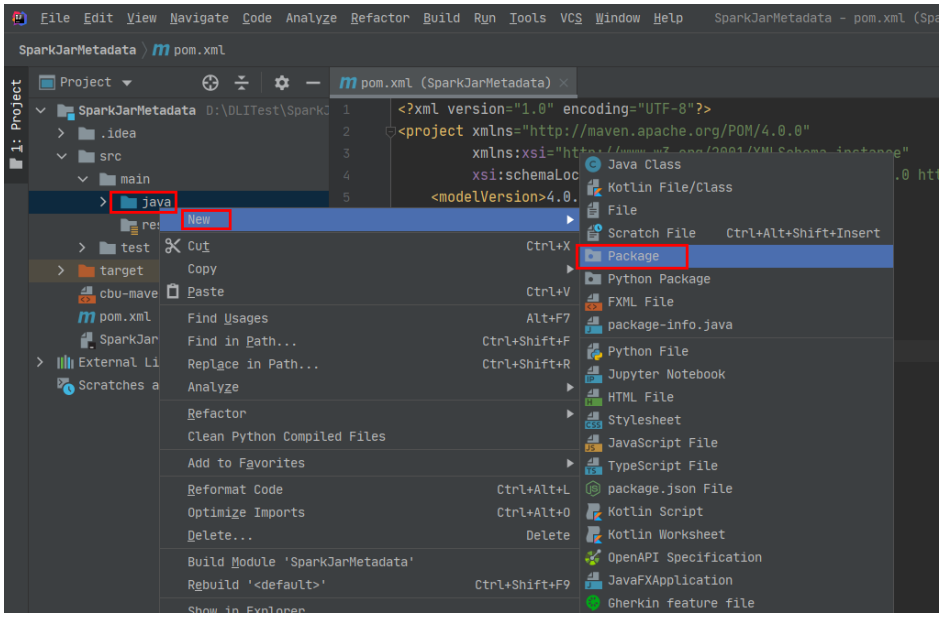
4. 在 pom.xml 文件中添加如下配置。

```
<dependencies>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql 2.11</artifactId>
            <version>2.3.2</version>
        </dependency>
</dependencies>
```

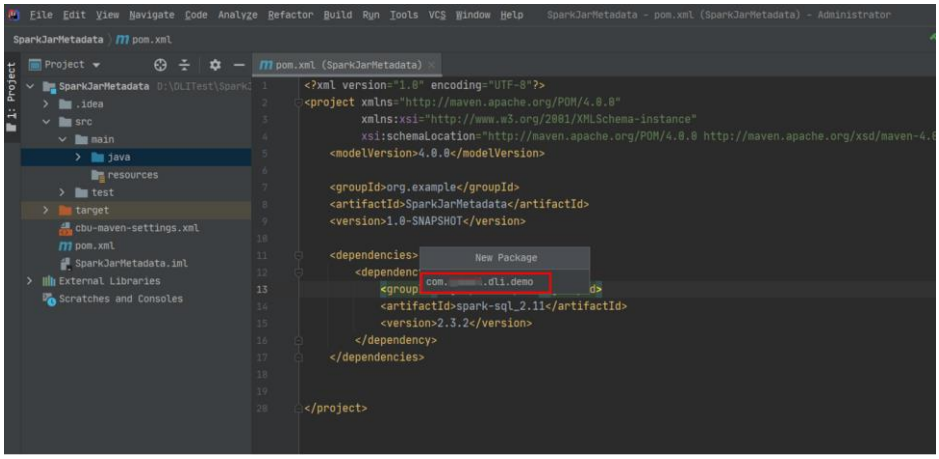
图4-8 修改 pom.xml 文件



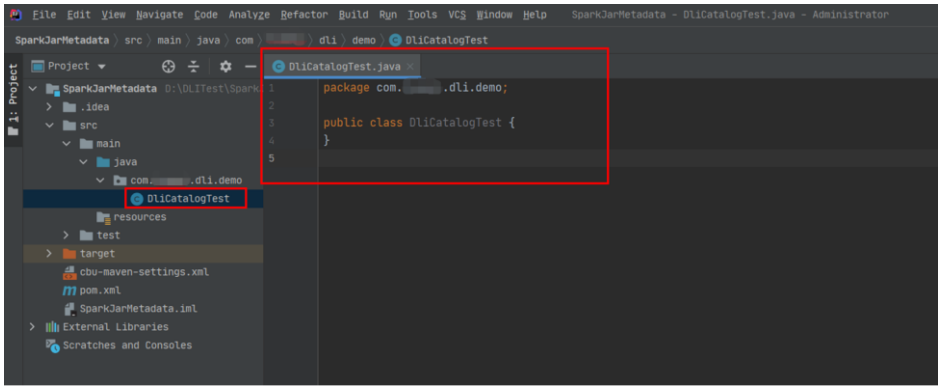
5. 在工程路径的“src > main > java”文件夹上鼠标右键，选择“New > Package”，新建 Package 和类文件。



Package 根据需要定义，本示例定义为：“com.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。



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。

```
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.DliSparkExte
nsion")
                .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();
```

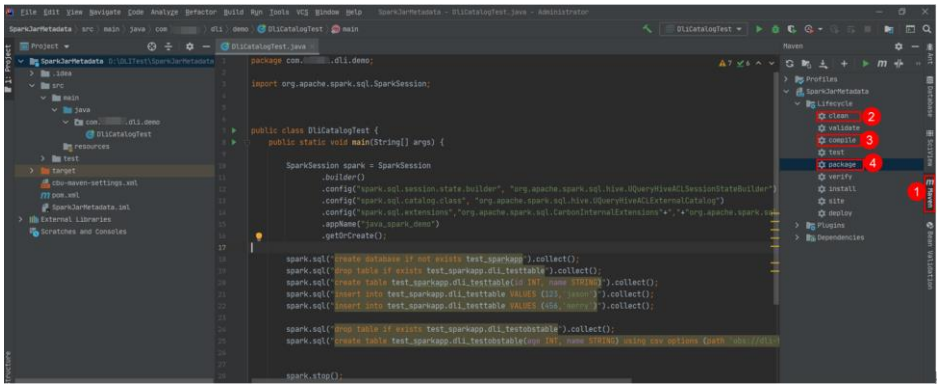
6. 关闭 SparkSession 会话 spark。

```
spark.stop();
```

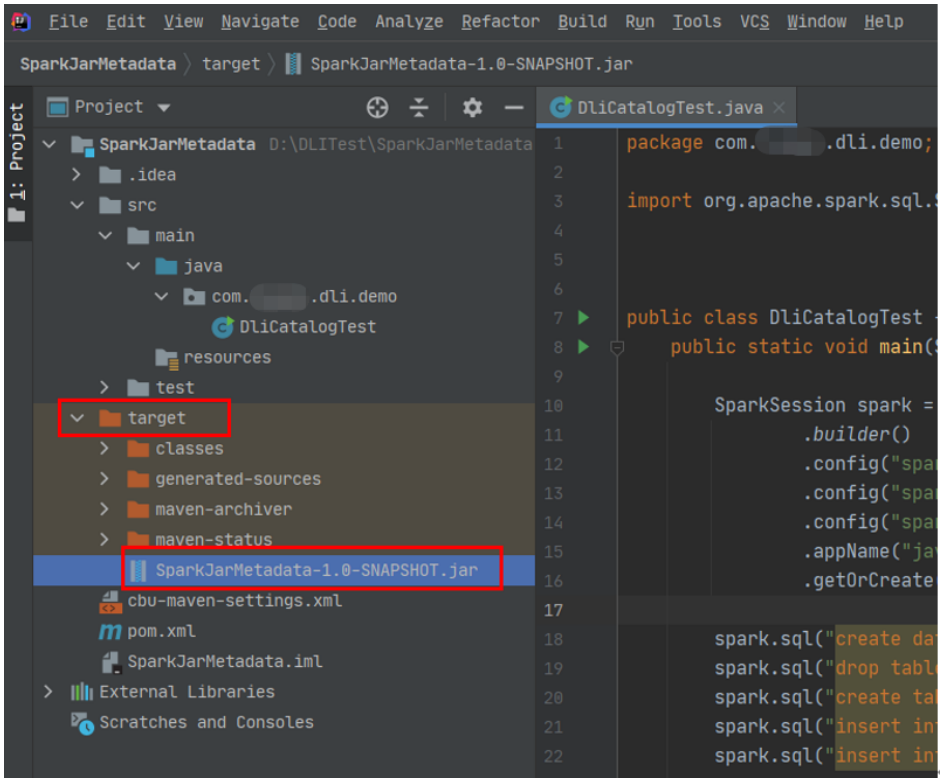
步骤 5：调试、编译代码并导出 Jar 包

1. 单击 IntelliJ IDEA 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。



   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”。



步骤 6：上传 Jar 包到 OBS 和 DLI 下

1. 登录 OBS 控制台，将生成的“SparkJarMetadata-1.0-SNAPSHOT.jar”Jar 包文件上传到 OBS 路径下。
2. 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。

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：

表4-5 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

参数名	参数值
	通用队列创建的通用队列“sparktest”。
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：“SparkJarObs-1.0-SNAPSHOT.jar”。
主类	格式为：程序包名+类名。
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

其他参数保持默认值即可。

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。

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”。
   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。
   c. 双击 DLI 表名 dli_testtable，单击“执行”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。
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。

   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

- 创建 DLI 表的语法请参考《数据湖探索 SQL 语法参考》中的“批作业 SQL 语法 > 创建 DLI 表”章节，创建 OBS 表的语法请参考《数据湖探索 SQL 语法参考》中的“批作业 SQL 语法 > 创建 OBS 表”章节。
- 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。

- 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.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.UQueryHiveAC
LSessionStateBuilder")
      .set("spark.sql.catalog.class","org.apache.spark.sql.hive.UQueryHiveACLExterna
lCatalog")
    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.UQueryHiveACLS
essionStateBuilder") \
.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-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

在使用 DLI Beeline 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云资源的访问。

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

   目前包括角色（粗粒度授权）和策略（细粒度授权）。

2. 创建队列。在“队列类型”中选择“SQL 队列”，即 SQL 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。

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 向管理员获取 SDK 驱动包下载地址。

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，推荐在 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 4-6）。

表4-6 DLI 客户端工具配置参数

属性项	必须配置	默认值	描述
dliEndPont	否	-	DLI 服务的域名。 如果不配置，程序根据 region 参数来确定域名。
obsEndPoint	是	-	OBS 服务的域名。 向管理员询问获取 OBS 对应的域名。
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

属性项	必须配置	默认值	描述
			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，改目录在 bucketName 指定的桶下。如果该目录不存在，程序会自动创建。
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4-7 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

参数名称	参数值	描述
--jars	<JARS>	Spark 应用依赖的 jar 包名称，存在多个时使用","分割。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。
--py-files	<PY_FILES>	Spark 应用依赖的 Python 程序文件名称，存在多个时使用","分割。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

4.4 使用 Spark 作业跨源访问数据源

4.4.1 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。

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。

4.4.2 对接 CSS

4.4.2.1 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
   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 4-8，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。

4.4.2.2 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。

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4-8 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。 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 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。

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 DLI 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 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。
- 通过 API 提交作业请参考

- 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。

    ```
    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4-9 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。 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 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”。
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。

参数	说明
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 DLI 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

- 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。
- 通过 API 提交作业请参考

- 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

```
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()
  }
}
```

4.4.2.3 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。

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 4-8。

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。
  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 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。
- 通过 API 提交作业请参考

- 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

```
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 4-8。

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。
  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

- 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。
- 通过 API 提交作业请参考

- 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()
  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()
```

4.4.2.4 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。

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 DLI 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 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。
- 通过 API 提交作业请参考

- 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 4.4.2.1 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 4-8。
  - 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。

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

- 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。
- 通过 API 提交作业请参考
  - 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.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' = '192.168.13.189:9200', 'es.nodes.wan.only' =
'true', 'resource' =
'/mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'obs://
桶名/地址/transport-keystore.jks','es.net.ssl.keystore.pass' = '**',
'es.net.ssl.truststore.location'='obs://桶名/地址/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 4-8。

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。

ii. 如果是开启 https 访问场景，在创建 Spark 作业时，需要同时上传依赖文件“hadoop-site.xml”。“hadoop-site.xml”文件具体内容参考如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--
  Licensed under the Apache License, Version 2.0 (the "License");
  you may not use this file except in compliance with the License.
  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing, software
  distributed under the License is distributed on an "AS IS" BASIS,
  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
  See the License for the specific language governing permissions and
  limitations under the License. See accompanying LICENSE file.
-->

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
    <name>fs.obs.bucket.桶名.access.key</name>
    <value>AK</value>
  </property>
<property>
    <name>fs.obs.bucket.桶名.secret.key </name>
    <value>SK</value>
  </property>
</configuration>
```

说明

<name>fs.obs.bucket.*桶名*.access.key</name>是为了更好的定位桶地址，该桶名为存放 keystore.jks 和 truststore.jks 文件的桶名。

iii. 在 Spark 作业编辑器中选择对应的 Module 模块并执行 Spark 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。
- 通过 API 提交作业请参考

- 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_css_security_httpson {
   public static void main(String[] args) {
       SparkSession 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' = '192.168.13.189:9200',
'es.nodes.wan.only' = 'true', 'resource' =
'/mytest','es.net.ssl'='true','es.net.ssl.keystore.location' = 'obs://
桶名/地址/transport-keystore.jks','es.net.ssl.keystore.pass' =
'**','es.net.ssl.truststore.location'='obs://桶名/地址
/truststore.jks','es.net.ssl.truststore.pass'='**','es.net.http.auth.u
ser'='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();
   }
}
```

4.4.3 对接 DWS

4.4.3.1 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。

- 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 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',
    'password'='######')")
```

表4-10 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。
user	DWS 数据仓库用户名。
password	DWS 数据仓库用户名对应密码。
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。

参数	说明
	• 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。
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”。

b. 插入数据

```
sparkSession.sql("insert into dli_to_dws values(1, 'John',24),(2,
'Bob',32)")
```

c. 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|
+---+------+---+
```

d. 删除关联表

```
sparkSession.sql("drop table dli_to_dws")
```

- 通过 DataFrame API 访问

a. 连接配置。

```
val url = "jdbc:postgresql://to-dws-1174405057-
EA1Kgo8H.datasource.com:8000/postgres"
val username = "dbadmin"
val password = "######"
val dbtable = "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 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”。

d. 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“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，示列如下：

    ```
    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，示列如下：

    - 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，示列如下：

```
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 DLI 中。
  b. 在 Spark 作业编辑器中选择对应的 Module 模块并执行 Spark 作业。

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_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

```
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()
  }
}
```

4.4.3.2 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。

- 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-dws").getOrCreate()
  ```

- 通过 DataFrame API 访问

  a. 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"
  ```

  b. 设置数据

  ```
  dataList = sparkSession.sparkContext.parallelize([(1, "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 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”。

f. 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()
```

g. 操作结果

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  1|Katie| 19|
+---+-----+---+
```

- 通过 SQL API 访问

a. 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 4-10。

b. 插入数据

```
sparkSession.sql("insert into dli_to_dws values(2,'John',24)")
```

c. 查询数据

```
jdbcDF = sparkSession.sql("select * from dli_to_dws").show()
```

d. 操作结果

```
+---+-----+---+
| id| name|age|
+---+-----+---+
|  1|Katie| 19|
|  2| John| 24|
+---+-----+---+
```

- 提交 Spark 作业
  a. 将写好的 python 代码文件上传至 DLI 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

- 提交作业时，需要指定 Module 模块，名称为：sys.datasource.dws。
- 通过控制台提交作业请参考。
- 通过 API 提交作业请参考

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-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()
```

4.4.3.3 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。

- 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-dws").getOrCreate();
```

- 通过 SQL API 访问

  a. 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'='**')");
```

  b. 插入数据

```
sparkSession.sql("insert into dli_to_dws values(3,'Liu'),(4,'Xie')");
```

  c. 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

  a. 将写好的代码文件生成 jar 包，上传至 DLI 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.dws。

- 如果选择 Spark 版本为 3.1.1 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。
- 通过 API 提交作业请参考

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'='dbadmi
n','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();
   }
}
```

4.4.4 对接 HBase

4.4.4.1 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。
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。

开启 Kerberos 认证时的相关配置文件

1. 参考中的“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 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。

4.4.4.2 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。

- 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-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:CF
1.longf,floatf:CF1.floatf,doublef: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-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:CF
1.longf,floatf:CF1.floatf,doublef:CF1.doublef',
   'krb5conf'='./krb5.conf',
   'keytab' = './user.keytab',
    'principal' = 'krbtest')")
```

表4-11 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。 • 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"。
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

  a. 插入数据

  ```
  sparkSession.sql("insert into test_hbase
  values('12345','abc','guiyang',false,null,3,23,2.3,2.34)")
  ```

  b. 查询数据

  ```
  sparkSession.sql("select * from test_hbase").show ()
  ```

  返回结果：

  ```
  +-----+--------+-------+--------+------+----+-----+------+-------+\n
  |   id|location|   city|booleanf|shortf|intf|longf|floatf|doublef|\n
  +-----+--------+-------+--------+------+----+-----+------+-------+\n
  |12345|  [illegible]|guiyang|   false|  null|   3|   23|   2.3|   2.34|\n
  |abcde|  [illegible]|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
  |check|  [illegible]|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
  \n+-----+--------+-------+--------+------+----+-----+------+-------+\n\n
  ```

- 通过 DataFrame API 访问

  a. 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)
  ```

  b. 根据 schema 的类型构造数据

  ```
  val mutableRow: Seq[Any] =
  Seq("12345","abc","guiyang",false,null,3,23,2.3,2.34)
  val rddData: RDD[Row] =
  sparkSession.sparkContext.parallelize(Array(Row.fromSeq(mutableRow)), 1)
  ```

  c. 导入数据到 HBase

  ```
  sparkSession.createDataFrame(rddData, new
  StructType(attrs)).write.insertInto("test_hbase")
  ```

  d. 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,flo
  atf:CF1.floatf,doublef:CF1.doublef"
  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|  [illegible]|guiyang|   false|  null|   3|   23|   2.3|   2.34|\n
|abcde|  [illegible]|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
|check|  [illegible]|abcdefg|    true|     1|   2|    3|   4.0|    5.0|\n
\n+-----+--------+-------+--------+------+----+-----+------+-------+\n\n
```

- 提交 Spark 作业
  a. 将写好的代码生成 jar 包，上传至 DLI 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。
  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 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。
- 通过 API 提交作业请参考

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','guiyang',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.l
ongf,floatf:CF1.floatf,doublef: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.lo
ngf,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","guiyang",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:C
F1.floatf,doublef:CF1.doublef"
    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()
  }
}
```

4.4.4.3 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。
- 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 4-12 所示。

表4-12 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 4-11。

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 4-11。
- 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,flo
atf: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 DLI 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。
  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 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。
- 通过 API 提交作业请参考

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.l
ongf,floatf:CF1.floatf,doublef: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:C
F1.floatf,doublef:CF1.doublef')")

  # 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:C
F1.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()
```

4.4.4.4 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。

- 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') ");
```

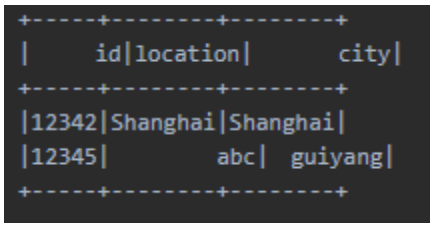
    ii. 插入数据

```
sparkSession.sql("insert into testhbase
values('12345','abc','guiyang')");
```

iii. 查询数据

```
sparkSession.sql("select * from testhbase").show();
```

插入数据后：



- 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.co
nf','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 4-13 所示。

表4-13 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 DLI 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。

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 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。
- 通过 API 提交作业请参考

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','Co
ls'='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.l
ongf,floatf:CF1.floatf,doublef: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();
    }
}
```

4.4.4.5 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。

4.4.5 对接 OpenTSDB

4.4.5.1 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。

- 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4-14 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。
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|  [redacted]|1970-01-02 18:17:36| 30.0|\n
|beijing|  [redacted]|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|  [illegible]|1970-01-02 18:17:36| 30.0|\n
|beijing|  [illegible]|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 DLI 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 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。
- 通过 API 提交作业请参考

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()
  }
}
```

4.4.5.2 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。

- 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 4-14。

- 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|  [illegible]|1970-01-02 18:17:36| 30.0|
|nanjing|  [illegible]|2019-08-28 00:00:00| 30.0|
|beijing|  [illegible]|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 DLI 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 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。
- 通过 API 提交作业请参考

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()
```

4.4.5.3 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。

- 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 4-14。

- 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 DLI 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 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。
- 通过 API 提交作业请参考

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();

   }
}
```

4.4.5.4 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”。

4.4.6 对接 RDS

4.4.6.1 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。

- 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4-15 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"。
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。
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

参数	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'。

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，示列如下：

    ```
    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，示列如下：

    - 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，示列如下：

```
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 DLI 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 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。
- 通过 API 提交作业请参考

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()
```

4.4.6.2 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。

- 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 4-15。

  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 4-15。

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 DLI 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 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。
- 通过 API 提交作业请参考

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()
```

4.4.6.3 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。

- 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 4-15。

  - 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 DLI 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 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。
- 通过 API 提交作业请参考

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','p
assword'='**','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();
   }
}
```

4.4.7 对接 Redis

4.4.7.1 scala 样例代码

开发说明

redis 只支持增强型跨源。

- 前提条件

  在 DLI 管理控制台上已完成创建增强跨源连接，并绑定队列。

- 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4-16 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）。
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

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
|[illegible]| 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 DLI 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 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。
- 通过 API 提交作业请参考

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("xxx",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).optio
n("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()
  }
}
```

4.4.7.2 pyspark 样例代码

开发说明

redis 只支持增强型跨源。

- 前提条件

  在 DLI 管理控制台上已完成创建增强跨源连接，并绑定队列。

- 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 DLI 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 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。
- 通过 API 提交作业请参考

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()
```

4.4.7.3 java 样例代码

开发说明

redis 只支持增强型跨源。

- 前提条件

  在 DLI 管理控制台上已完成创建增强跨源连接，并绑定队列。

- 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\":\"zhangsan\",\"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).sa
ve();
```

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 DLI 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 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。
- 通过 API 提交作业请参考

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\":\"zhangsan\",\"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();

  }
}
```

4.4.7.4 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。

4.4.8 对接 Mongo

4.4.8.1 scala 样例代码

开发说明

mongo 只支持增强型跨源。

- 前提条件

  在 DLI 管理控制台上已完成创建增强跨源连接，并绑定队列。

- 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_mongo(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4-17 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。

参数	说明
	说明 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。
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_mongo values('3', 'zhangsan',23)")
```

c. 查询数据

```
sparkSession.sql("select * from test_mongo").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 DLI 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 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。
- 通过 API 提交作业请参考

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_mongo(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 mongo values('3', 'zhangsan',23)")
    sparkSession.sql("select * from test mongo").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()
 }
}
```

4.4.8.2 pyspark 样例代码

开发说明

mongo 只支持增强型跨源。

- 前提条件

  在 DLI 管理控制台上已完成创建增强跨源连接，并绑定队列。

- 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 4-17。

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_mongo(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 4-17。

b. 插入数据

```
sparkSession.sql("insert into test_mongo values('3', 'zhangsan',23)")
```

c. 查询数据

```
sparkSession.sql("select * from test_mongo").show()
```

- 提交 Spark 作业
  a. 将写好的 python 代码文件上传至 DLI 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 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。
- 通过 API 提交作业请参考

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_mongo(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 mongo values('3', 'zhangsan',23)")
```

```
# Read data from DLI-table
sparkSession.sql("select * from test_mongo").show()

# close session
sparkSession.stop()
```

4.4.8.3 java 样例代码

开发说明

mongo 只支持增强型跨源。

- 前提条件

  在 DLI 管理控制台上已完成创建增强跨源连接，并绑定队列。

- 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\":\"zhang
san\",\"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 4-17。

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 DLI 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 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。
- 通过 API 提交作业请参考

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\":\"zhangsan\",\"a
ge\":\"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();
  }
}
```