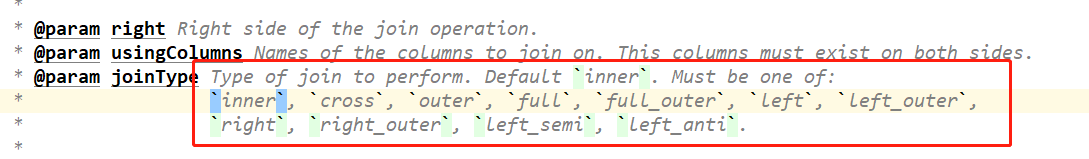
# 多表关联join

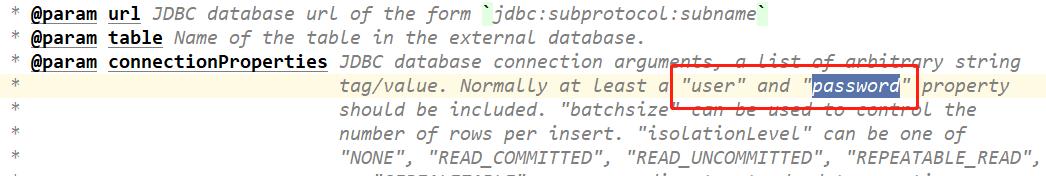
DSL语法风格中的N多种join方式：

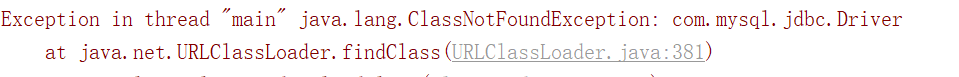


|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()   **import** spark.implicits.\_   *// userInfo name age sex pcode* **val** user: Dataset[String] = spark.createDataset(*List*(**"dingding 18 F henan"**, **"dangdang 16 F henan"**, **"dongdong 28 M shandong"**,  **"laozhao 30 M heilongjiang"**, **"cangls 36 F jp"**,**"meixi 30 M agt"**))   *// province pcode pname* **val** province: Dataset[String] = spark.createDataset(*List*(**"henan,河南省"**, **"shandong,山东省"**, **"heilongjiang,黑龙江"**, **"jp,日本省"**, **"tw,台湾省"**))    *// 对数据进行切分* **val** userSplit: Dataset[Row] = user.map(str => {  **val** split = str.split(**" "**)  (split(0), split(1).toInt, split(2), split(3))  }).toDF(**"name"**, **"age"**, **"gender"**, **"pcode"**)  **val** proSplit: Dataset[Row] = province.map(str => {  **val** split = str.split(**","**)  (split(0), split(1))  }).toDF(**"pcode"**, **"province"**)   *// SQL  // 注册2个临时视图 关联查询* userSplit.createTempView(**"v\_user"**)  proSplit.createTempView(**"v\_pro"**)  *// val result1: DataFrame = spark.sql("select \* from v\_user t1 join v\_pro t2 on t1.pcode = t2.pcode ") // result1.show()   //DSL join 添加条件 判断的标准是 ===* userSplit.join(proSplit).where(userSplit(**"pcode"**) === proSplit(**"pcode"**)) *// .show()* userSplit.join(proSplit,userSplit(**"pcode"**) === proSplit(**"pcode"**),**"right"**).show()   *// 如果参与join的字段名称一致 可以直接使用一个* userSplit.join(proSplit,**"pcode"**)*// .show()   // 默认使用的是inner join* spark.stop() |

# 根据IP地址求归属地，写mysql

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()  **import** spark.implicits.\_  *// 读取数据* **val** ipruleDs: Dataset[String] = spark.read.textFile(**"f:/mrdata/ipdata/ip.txt"**) **val** logDs: Dataset[String] = spark.read.textFile(**"f:/mrdata/ipdata/ipaccess.log"**)  *// 数据切分* **val** ipRules: Dataset[Row] = ipruleDs.map(str => {  **val** split = str.split(**"\\|"**)  (split(2).toLong, split(3).toLong, split(6)) }).toDF(**"start"**, **"end"**, **"province"**)  **val** longIps = logDs.map(str => {  **val** split = str.split(**"\\|"**)  **val** ip = split(1)  IpUtils.*ip2Long*(ip) }).toDF(**"longIp"**)   *// 数据匹配聚合 关联查询* ipRules.createTempView(**"v\_ipRules"**)  longIps.createTempView(**"v\_longIp"**)  **val** result: DataFrame = spark.sql(**"select province,count(\*) as cnts from v\_ipRules t1 join "** +  **"v\_longIp t2 on t2.longIp between t1.start and t1.end "** +  **"group by province "** +  **" order by cnts desc "**) result.show() |





添加mysql的驱动jar包：

|  |
| --- |
| *<!--mysql的驱动jar包-->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.38</**version**> </**dependency**> |

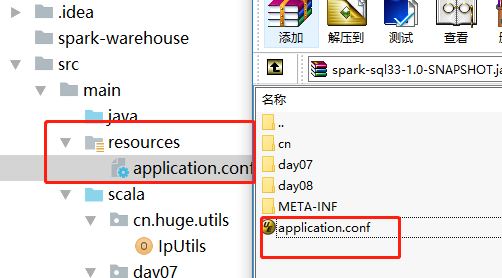
|  |
| --- |
| *// 写入mysql* **val** url = **"jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8" val** table = **"access\_log10" val** conn = **new** Properties() conn.setProperty(**"user"**, **"root"**) conn.setProperty(**"password"**, **"123"**) conn.setProperty(**"driver"**, **"com.mysql.jdbc.Driver"**)  result.write.jdbc(url, table, conn) |

## 读取配置文件

### 利用类加载器获取配置文件

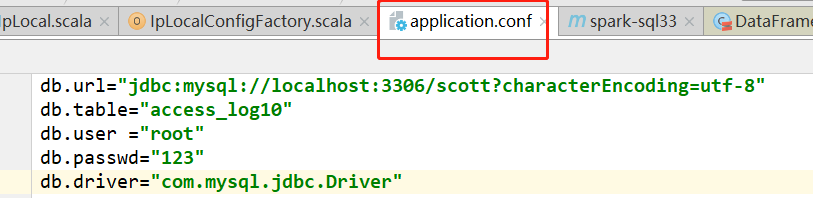
|  |
| --- |
| *// 传统的通过类加载器获取某一个配置文件 // val asStream = IpLocal.getClass.getClassLoader.getResourceAsStream("application.conf") // val prop = new Properties() // prop.load(asStream) // prop.getProperty("")* |

### 利用ConfigFactory来加载配置文件



必须导入依赖jar包

|  |
| --- |
| *<!-- <dependency>  <groupId>com.typesafe.akka</groupId>  <artifactId>akka-remote\_2.11</artifactId>  <version>2.4.17</version>  </dependency>-->  <!--ConfigFactory所在的jar包-->* <**dependency**>  <**groupId**>com.typesafe</**groupId**>  <**artifactId**>config</**artifactId**>  <**version**>1.3.0</**version**>  </**dependency**> |



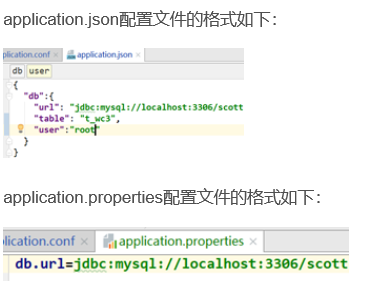
|  |
| --- |
| *// 默认的加载顺序 application.conf --> application.json ---> application.properties* **val** load = ConfigFactory.*load*() **val** conf1 = load.getString(**"db.url"**) *println*(conf1) |

把mysql的配置写入到配置文件中：

在代码中通过配置来获取参数：

|  |
| --- |
| *// 写入mysql* **val** url = config.getString(**"db.url"**) **val** table = config.getString(**"db.table"**) **val** conn = **new** Properties() *// user是关键字 ，不能用于key* conn.setProperty(**"user"**, config.getString(**"db.user"**)) conn.setProperty(**"password"**, config.getString(**"db.passwd"**)) conn.setProperty(**"driver"**, config.getString(**"db.driver"**))  result.write.*/\*mode(SaveMode.Append).\*/*jdbc(url, table, conn) |

不同的配置文件，有不同的写法：

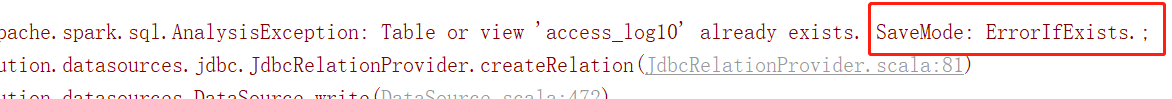


详情可参考： <https://blog.csdn.net/qq_21439395/article/details/80678524>

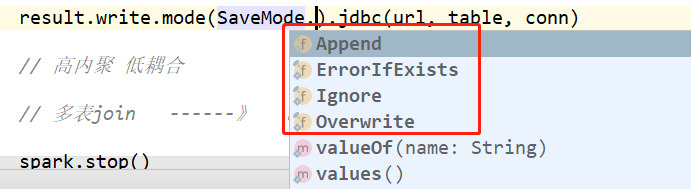
# 保存模式

## 简介

当第2次写入数据到mysql的时候，报错了：



原因： write 写的API，默认使用的模式是：SaveMode.ErrorIfExists 。 如果写入的路径存在，就会报错。



## 4种模式的说明：

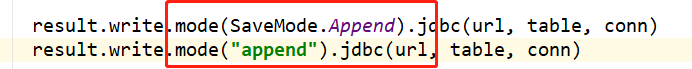
ErrorIfExists 默认模式, 如果存在，即报错。

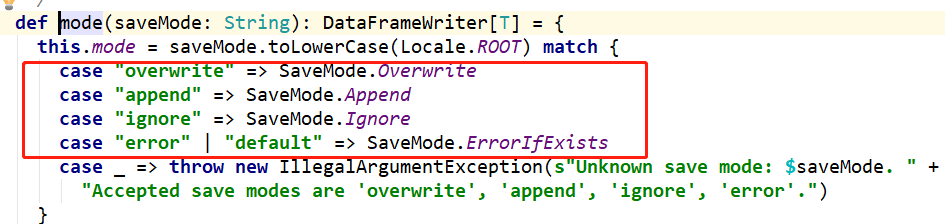
Append : 追加模式， 如果存在，就追加

Ignore： 忽略模式， 如果存在，不写了。

Overwrite: 覆盖模式，先删除，再创建

2种写法： SaveMode.Append ； “append”





推荐使用SaveMode.Append 写法。不容易写错。

只要使用writeAPI 进行数据存储，都可以使用mode方法。

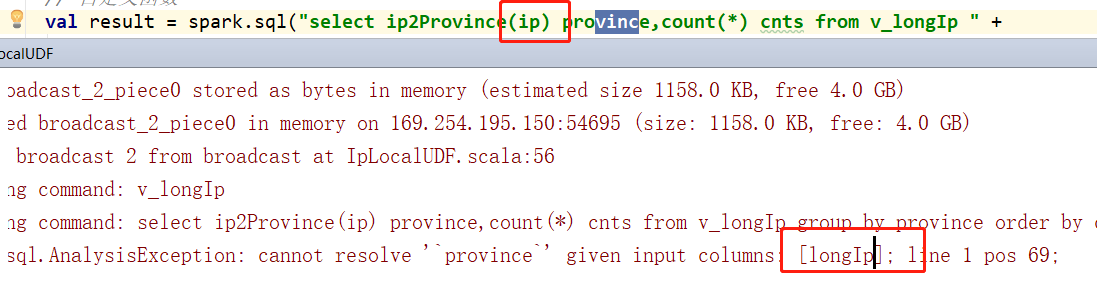
# 自定义函数

UDF： User defined Functions

UDAF User defined aggregate Function

UDTF



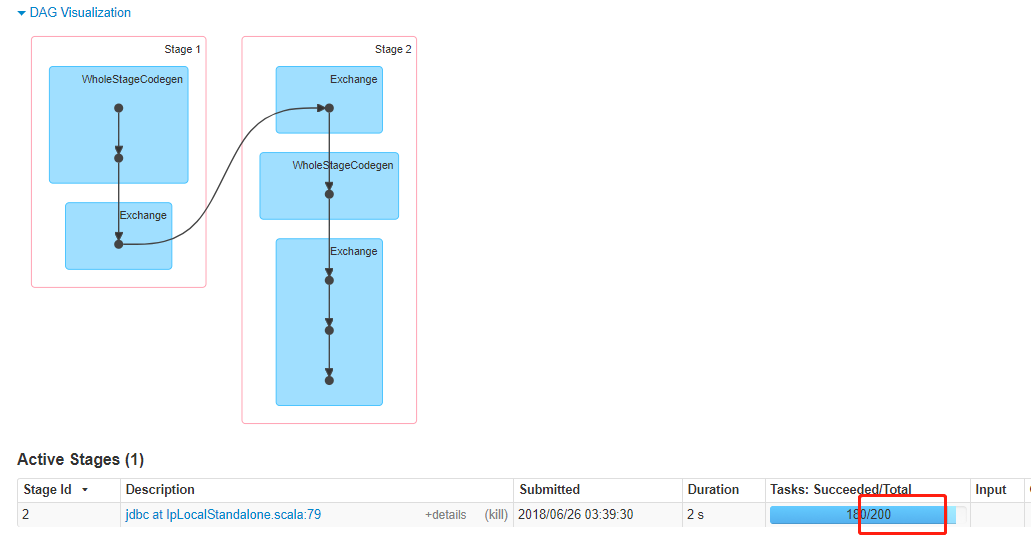


|  |
| --- |
| **val** config = ConfigFactory.*load*()**val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()  **import** spark.implicits.\_  *// 读取数据* **val** ipruleDs: Dataset[String] = spark.read.textFile(**"f:/mrdata/ipdata/ip.txt"**) **val** logDs: Dataset[String] = spark.read.textFile(**"f:/mrdata/ipdata/ipaccess.log"**)  *// 数据切分* **val** ipRules: Dataset[(Long, Long, String)] = ipruleDs.map(str => {  **val** split = str.split(**"\\|"**)  (split(2).toLong, split(3).toLong, split(6)) })  *// 可以 先把收集收集起来 ds --> Array // ipRules.rdd.collect() DataFrame .collect ---> Array[Row]* **val** ipRulesArr: Array[(Long, Long, String)] = ipRules.collect()  *// 把ip规则库数据进行广播* **val** bc: Broadcast[Array[(Long, Long, String)]] = spark.sparkContext.broadcast(ipRulesArr)  **val** longIps = logDs.map(str => {  **val** split = str.split(**"\\|"**)  **val** ip = split(1)  IpUtils.*ip2Long*(ip) }).toDF(**"longIp"**)  longIps.createTempView(**"v\_longIp"**)  *// 自定义函数的名称* spark.udf.register(**"ip2Province"**, (ip: Long) => {  **val** newIpRules: Array[(Long, Long, String)] = bc.value  *// 根据10 进制的ip地址，得到省份* IpUtils.*binarySearchFun*(newIpRules, ip) }) *// 自定义函数* **val** result = spark.sql(**"select ip2Province(longIp) province,count(\*) cnts from v\_longIp where ip2Province(longIp) != 'unknown' "** +  **"group by province"** +  **" order by cnts desc"**)  result.printSchema()  *// 写入mysql* **val** url = config.getString(**"db.url"**) **val** table = config.getString(**"db.table"**) **val** conn = **new** Properties() *// user是关键字 ，不能用于key* conn.setProperty(**"user"**, config.getString(**"db.user"**)) conn.setProperty(**"password"**, config.getString(**"db.passwd"**)) conn.setProperty(**"driver"**, config.getString(**"db.driver"**)) result.write.mode(SaveMode.*Overwrite*).jdbc(url, table, conn) |

# 把sparksql的任务提交到集群运行

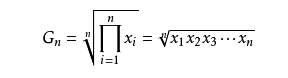
通过—jars 导入jar包，或者把jar包放到spark的jars目录下。

|  |
| --- |
| spark-submit --master spark://hdp-01:7077 --class day08.IpLocalStandalone --jars mysql-connector-java-5.1.38.jar,config-1.3.0.jar /root/spark-sql33-1.0-SNAPSHOT.jar hdfs://hdp-01:9000/ip.txt hdfs://hdp-01:9000/ipaccess.log |



# UDAF-几何平均数

几何平均数：



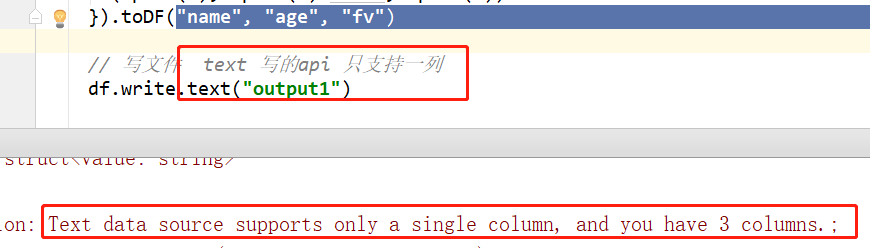
掌握实现的思路即可。

# SparkSql中的数据源

spark sql 读取 写入 数据的格式

file,jdbc,json,csv,parquet

## file



|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()  **import** spark.implicits.\_  *// 读取文件* **val** file: Dataset[String] = spark.read.textFile(**"person.txt"**)   *// val text: DataFrame = spark.read.text("person.txt")* **val** df: DataFrame = file.map(t => {  **val** split = t.split(**" "**)  (split(0), split(1).toInt, split(2)) }).toDF(**"name"**, **"age"**, **"fv"**)  *// 写文件 text 写的api 只支持一列* df.select(**"name"**).write.text(**"output1"**)  *// saveApi 写的文件格式是Parquet文件 独有的读写API* df.write.save(**"output2"**)  spark.close() |

## JDBC数据源

JDBC去连接任意的数据库。

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()  **import** spark.implicits.\_  **val** config = ConfigFactory.*load*() *// 读取mysql数据库 ---》 操作 之后 ---》 写到mysql中* **val** url = config.getString(**"db.url"**) **val** conn = **new** Properties() conn.setProperty(**"user"**, config.getString(**"db.user"**)) conn.setProperty(**"password"**, config.getString(**"db.passwd"**))  *// 连接mysql数据库 设置参数url driver dbtable user password* **val** empData: DataFrame = spark.read.format(**"jdbc"**).options(  *Map*(  **"url"** -> **"jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8"**,  **"driver"** -> **"com.mysql.jdbc.Driver"**,  **"dbtable"** -> **"emp"**,  **"user"** -> **"root"**,  **"password"** -> **"123"** )).load() spark.read.format(**"jdbc"**).jdbc(url,**"emp"**,conn)  *// 读取数据* **val** jdbc: DataFrame = spark.read.jdbc(url, **"emp"**, conn)  jdbc.printSchema()  **val** result1: Dataset[Row] = jdbc.where(**"sal > 2500"**).select(**"empno"**)  *// 写数据* result1.write.mode(SaveMode.*Append*).jdbc(url, **"emp10"**, conn) spark.close() |

注意： 读取mysql的时候，必须导入myql的驱动jar包。

## JSON数据源

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()   *// dataset[Row]* **val** json: DataFrame = spark.read.json(**"product.json"**)   json.printSchema()   *// json 格式的数据，生成的schema信息不是按照原有的数据顺序，是按照字典顺序得到的。 // json.toDF("pname1","price1","amount1").show()* **val** result: Dataset[Row] = json.filter(**"price > 1000"**)   *// 读寄送 写 json* result.write.mode(SaveMode.*Append*).json(**"jsonout1"**)   spark.close() |

{**"name"**:**"zs"**,**"address"**:{**"city"**:**"bj"**,**"street"**:**"cp"**}}

|  |
| --- |
| *// dataset[Row]* **val** json: DataFrame = spark.read.json(**"address.json"**)  json.printSchema()  json.createTempView(**"v\_json"**)  **val** result1 = spark.sql(**"select address.city,address.street from v\_json"**) |

{**"name"**:**"zs"**,**"address"**:[{**"city"**:**"bj"**},{**"city"**:**"cp2"**}]}

|  |
| --- |
| *// dataset[Row]* **val** json: DataFrame = spark.read.json(**"address2.json"**) json.printSchema() **import** org.apache.spark.sql.functions.\_ json.select(**$"name"**, *explode*(**$"address"**)).toDF(**"name"**, **"city"**) .show() |

## csv

python

默认的分隔符是逗号， 可以直接使用excel 打开。

本身没有schema信息，读取和存入的数据都不带有schema。

默认生成的shema信息是 \_c0 \_c1 …. 默认的类型都是String

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()  **import** spark.implicits.\_  *// 读取文件* **val** file: Dataset[Row] = spark.read.csv(**"product.csv"**)  file.printSchema() */\*\*  \* root  \* |-- \_c0: string (nullable = true)  \* |-- \_c1: string (nullable = true)  \* |-- \_c2: string (nullable = true)  \*/* **val** config = ConfigFactory.*load*()  *// 读取mysql数据库 ---》 操作 之后 ---》 写到mysql中* **val** url = config.getString(**"db.url"**) **val** conn = **new** Properties() conn.setProperty(**"user"**, config.getString(**"db.user"**)) conn.setProperty(**"password"**, config.getString(**"db.passwd"**))  *// 读取数据* **val** jdbc: DataFrame = spark.read.jdbc(url, **"emp"**, conn)  jdbc.filter(**"sal >2600"**).write.csv(**"csvpath1"**)  spark.close() |

## parquet数据

实用

列式存储，

parquet数据格式的优势？？

列式存储，使用时只需要读取需要的列，支持向量运算，能获得更好的扫描行。 压缩编码可以降低磁盘存储空间，由于同一列的数据类型是一样的，可以使用不同的压缩编码。 可以跳过不符合条件的数据，只读取需要的数据，降低IO的数据量 通用的，适配多种计算框架，查询引擎（hive,impala,pig等），计算框架（mapreduce,spark等），数据模型（avro,thrift,json等）

优点：

列式存，读取操作方便。

支持压缩编码，有效减少磁盘占用空间。压缩比高：

通用性。

缺点：

不支持update操作。生成之后，不能修改。

parquet有自己独特的API。

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()  **import** spark.implicits.\_  *// 读取 parquet文件的数据* **val** file: Dataset[Row] = spark.read.parquet(**"output2"**)  file.printSchema()  file.show()  **val** result: Dataset[Row] = file.filter(**"age > 30"**)  *// 写到parquet* result.write.parquet(**"parquetout1"**)  spark.close() |

## 日志数据量补充：

处理的数据量/ 天 1000gb

1. 看每一条日志数据的大小， 有多少字段，看存储什么数据

0.5kb 80个字段

35M --- 7万条数据

天的数据量 = 有多少用户 \* 每一个用户 每天大概产生几条日志。

100000000 \* 10 \* 0.5KB

条数 / 数据量的大小

1TB 1个亿

# hiveOnSpark

利用Hive的规范，利用hive的元数据库，把数据存储在hdfs中。

HQL 编程，程序运行在 spark的集群中。

## 有了hive

如果已经存在了hive。直接把hive的配置，hive-site.xml，放到spark的安装conf目录下即可。

## 如果没有hive

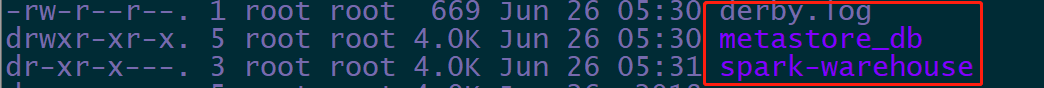
mysql的元数据库 ---》 hdfs之间的映射关系。

## spark-sql

相当于spark core 模式下的 spark-shell。

提供了一个编写sql语句的交互式命令行。

执行spark-sql，创建的数据库，的元数据？ 数据？ 都存放在哪里？



默认情况下，执行spark-sql,使用的元数据库是derby，元数据库的存储位置是metastore\_db,数据存储位置是spark-warehouse。

在哪里执行spark-sql命令，就在哪个目录下生成这几个目录。

derby --🡪 mysql

spark-warehouse -🡪 hdfs的目录。

## 创建一个mysql数据库：

只需要安装一台即可。

这里用的mysql版本比较低。 安装完成之后，没有默认的密码。

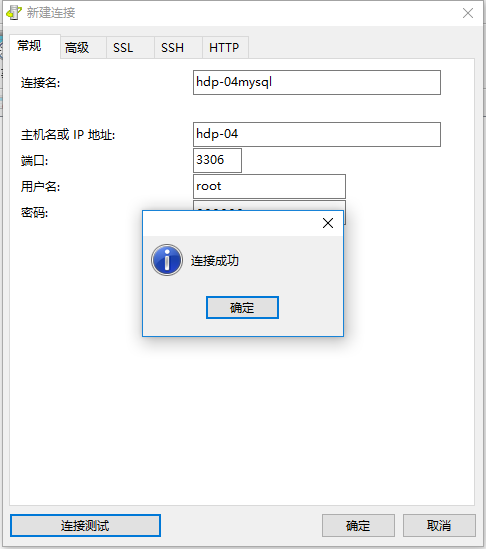
|  |
| --- |
| # 安装mysql  yum -y install mysql mysql-server  # 启动mysql  service mysqld start  #安全配置  /usr/bin/mysql\_secure\_installation |

对root用户进行授权：

# mysql -uroot -p111111

|  |
| --- |
| mysql> GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%' IDENTIFIED BY '111111' WITH GRANT OPTION;  Query OK, 0 rows affected (0.00 sec)  mysql> FLUSH PRIVILEGES;  Query OK, 0 rows affected (0.00 sec) |

远程连接，确保防火墙是关闭的。



## 配置hive-site.xml文件

如何让创建的数据库，使用自己的mysql呢？

启动spark-sql的地方，称之为客户端。 hive-site.xml 只需要在客户端配置。

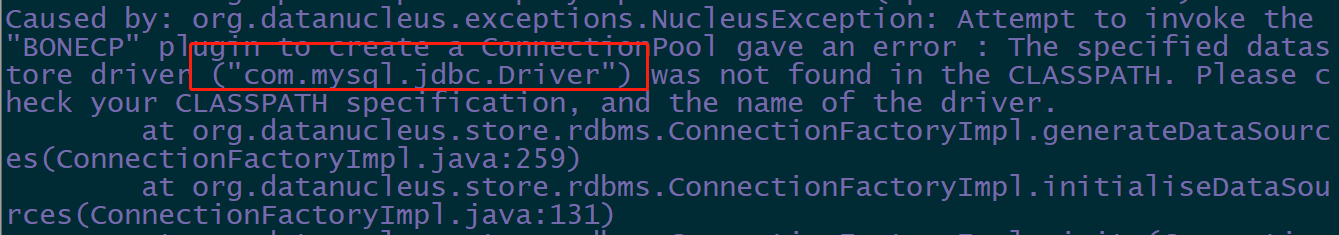
假定客户端使用hdp-02机器。

在spark的conf目录下，创建hive-site.xml文件：

|  |
| --- |
| <**configuration**>  <**property**>  <**name**>javax.jdo.option.ConnectionURL</**name**>  <**value**>jdbc:mysql://hdp-04:3306/hivedb?createDatabaseIfNotExist=true</**value**>  <**description**>JDBC connect string for a JDBC metastore</**description**>  </**property**>   <**property**>  <**name**>javax.jdo.option.ConnectionDriverName</**name**>  <**value**>com.mysql.jdbc.Driver</**value**>  <**description**>Driver class name for a JDBC metastore</**description**>  </**property**>   <**property**>  <**name**>javax.jdo.option.ConnectionUserName</**name**>  <**value**>root</**value**>  <**description**>username to use against metastore database</**description**>  </**property**>   <**property**>  <**name**>javax.jdo.option.ConnectionPassword</**name**>  <**value**>111111</**value**>  <**description**>password to use against metastore database</**description**>  </**property**> </**configuration**> |

直接重新启动sprak-sql 即可为我们创建好数据库。

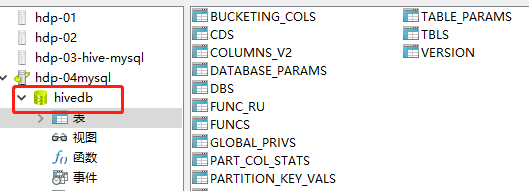
需要导入msyql的驱动jar包。



spark-sql –jars mysql..jar

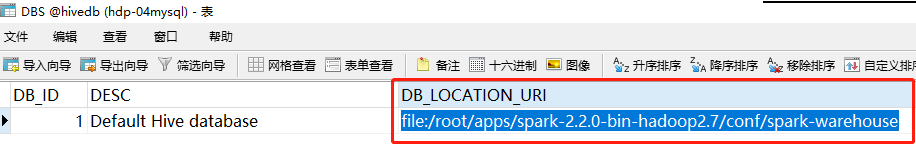
当spark-sql启动成功之后，可以发现，在hdp-04的数据库中，创建好了29张表。

mysql元数据库创建好了。

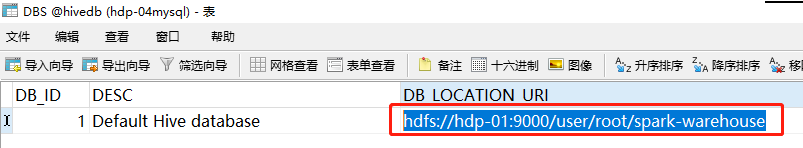


但是，目前再在spark-sql 中创建数据库表，依然不能使用hdfs的存储路径。（也就是数据库表的数据依然存在本地磁盘中）

修改表的路径：



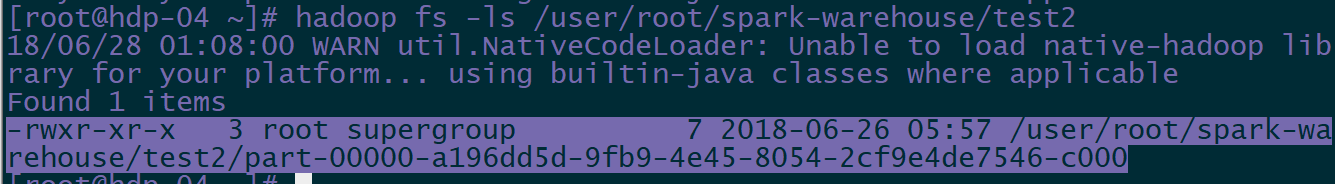
新的hdfs的路径： hdfs://hdp-01:9000/user/root/spark-warehouse



执行测试：

|  |
| --- |
| Spark-sql > create table test2(name varchar(20));  Spark-sql > insert into test2 values("xxxooo"); |

在 hdfs的指定目录下，有文件产生。



现在，就把hive -on -spark的配置完全搞定。

最终的一点，就是hive-site.xml, 放到spark conf 目录下。

## IDEA编程

环境准备：

导入jar包：

|  |
| --- |
| *<!--必须导入spark对hive的支持jar包-->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-hive\_2.11</**artifactId**>  <**version**>${spark.version}</**version**> </**dependency**> |

开启spark对hive的支持：

|  |
| --- |
| **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  *// 必须开始spark对hive的支持* **.enableHiveSupport()**  .getOrCreate() |

本地测试：

需要把hive-site.xml文件，添加到本地资源目录下。

‘

基本的HQL 操作：

|  |
| --- |
| *// 伪装用户身份* System.*setProperty*(**"HADOOP\_USER\_NAME"**, **"root"**)   **val** spark: SparkSession = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  *// 必须开始spark对hive的支持* .enableHiveSupport()  .getOrCreate()   **import** spark.implicits.\_   *// 查询* spark.sql(**"select \* from t\_access\_times"**)  *// .show()   // 创建表   // 求每个用户的每月总金额  // spark.sql("select username,month,sum(salary) as salary from t\_access\_times group by username,month")  // 创建表  // spark.sql("create table t\_access1(username string,month string,salary int) row format delimited fields terminated by ','")   // 删除表  // spark.sql("drop table t\_access1")   // 插入数据  // spark.sql("insert into t\_access1 select \* from t\_access\_times")  // .show()  // 覆盖写数据  // spark.sql("insert overwrite table t\_access1 select \* from t\_access\_times where username='A'")   // 覆盖load新数据  // C,2015-01,10  // C,2015-01,20  // spark.sql("load data local inpath 't\_access\_time\_log' overwrite into table t\_access1")   // 清空数据 // spark.sql("truncate table t\_access1")  // .show()   // 写入自定义数据* **val** access: Dataset[String] = spark.createDataset(*List*(**"b,2015-01,10"**, **"c,2015-02,20"**))   **val** accessdf = access.map({  t =>  **val** lines = t.split(**","**)  (lines(0), lines(1), lines(2).toInt)  }).toDF(**"username"**, **"month"**, **"salary"**)   *// .show()* accessdf.createTempView(**"t\_ac"**) *// spark.sql("insert into t\_access1 select \* from t\_ac")   // insertInto的api 入库 // database.table* accessdf.write.mode(SaveMode.*Append*).insertInto(**"t\_access1"**)   *// new HiveContext(new SparkContext())* spark.stop() |

## TOpK

row\_number over( partition by xxx )

rank() 全局的排序 支持并列排序

dense\_rank() 选中的进行全局排序。

|  |
| --- |
| **object** TopK {  **val** *topK* = 2   **def** main(args: Array[String]): Unit = {  **val** session = SparkSession.*builder*()  .master(**"local[\*]"**)  .appName(**this**.getClass.getSimpleName)  .getOrCreate()   **import** session.implicits.\_   **val** file = session.read.textFile(**"f:/mrdata/teacher.log"**)  **val** st: DataFrame = file.map({  t =>  **val** index = t.lastIndexOf(**"/"**)  *// 截取字符串* **val** tName = t.substring(index + 1)  *// 封装数据为URL，然后获取Host内容* **val** uri = **new** URL(t.substring(0, index))  **val** host = uri.getHost   **val** hostArray = host.split(**"[.]"**) *// 要对特殊的字符进行转义  // 获取学科名称* **val** sub = hostArray(0)  *// 返回元组* (sub, tName)  }).toDF(**"subject"**, **"teacher"**)   st.createTempView(**"t\_sub\_teacher"**)   *// 该学科下的老师的访问次数* **val** sql = session.sql(**"select subject,teacher,count(\*) cnts from t\_sub\_teacher group By subject,teacher "**) *//order by cnts desc  // sql   // sql.show()   // 全局topK  // sql.limit(3).show()* sql.createTempView(**"v\_tmp"**)   *// 求分组topK 分学科的老师 排序  // val groupedTop = session.sql("select subject,teacher,cnts,row\_number() over(partition by subject order by cnts desc) sub\_order from v\_tmp order by cnts desc")  // groupedTop.show()  // 分学科的排序 取topK  // val groupedTop = session.sql(s"select \* from (select subject,teacher,cnts,  // row\_number() over(partition by subject order by cnts desc) sub\_order  // from v\_tmp order by cnts desc ) where sub\_order <= $topK")   // 分学科的排序 + 全局排序  // val groupedTop = session.sql("select subject,teacher,cnts,  // row\_number() over(partition by subject order by cnts desc) sub\_order,  // row\_number() over(order by cnts desc) g\_order  // from v\_tmp order by cnts desc")   // 分学科的TopK 排序 + 全局排序 rank 支持并列排序 // val groupedTop = session.sql("select \* from (select subject,teacher,cnts," + // "row\_number() over(partition by subject order by cnts desc) sub\_order," + // "rank() over(order by cnts desc) g\_order " + // s"from v\_tmp ) where sub\_order <=$topK") // groupedTop.show()   // 全局排序，分学科topk排序，选中的全局排序（可使用row\_number over() rank() over () dense\_rank()）* **val** groupedTop = session.sql(**"select \* ,dense\_rank() over(order by cnts desc) choose\_num from (select subject,teacher,cnts,"** +  **"row\_number() over(partition by subject order by cnts desc) sub\_order,"** +  **"rank() over(order by cnts desc) g\_order "** +  **s"from v\_tmp) where sub\_order <=$***topK***"**)   groupedTop.show()   *// 释放资源* session.close()  } } |