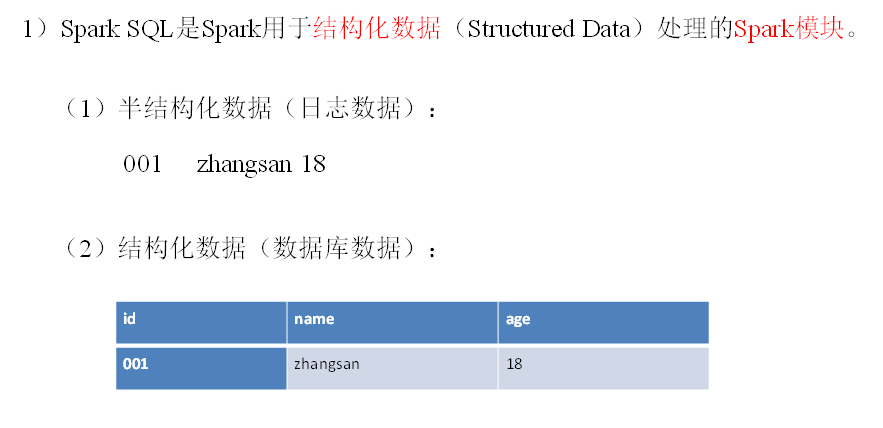
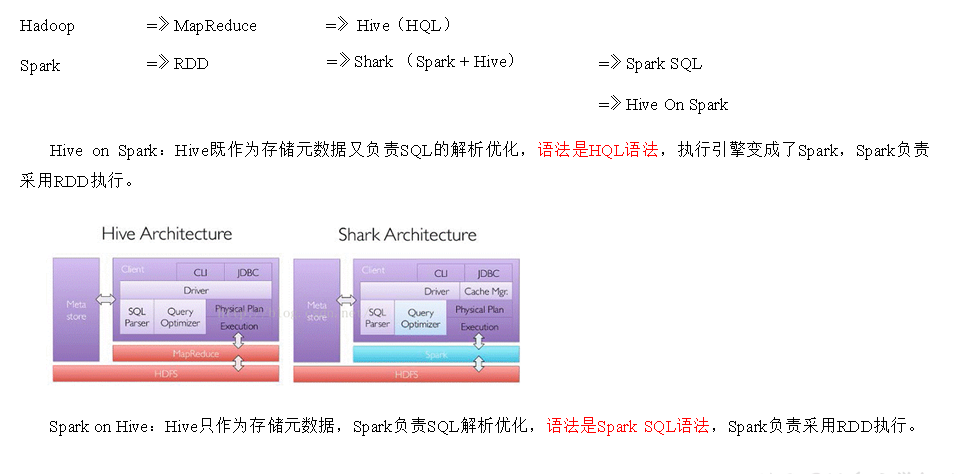
# 第1章 Spark SQL概述

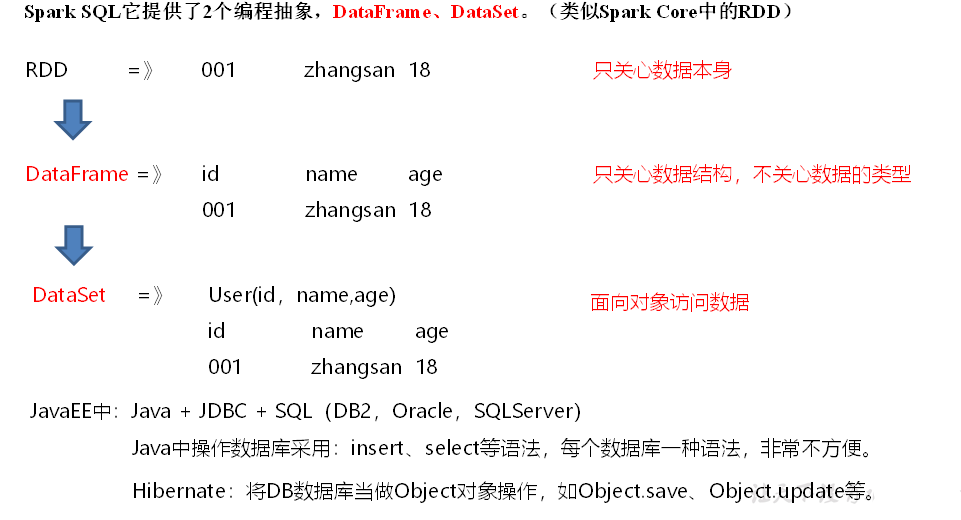
## 1.1 什么是Spark SQL



## 1.2 为什么要有Spark SQL



## 1.3 Spark SQL原理



### 1.3.1 什么是DataFrame

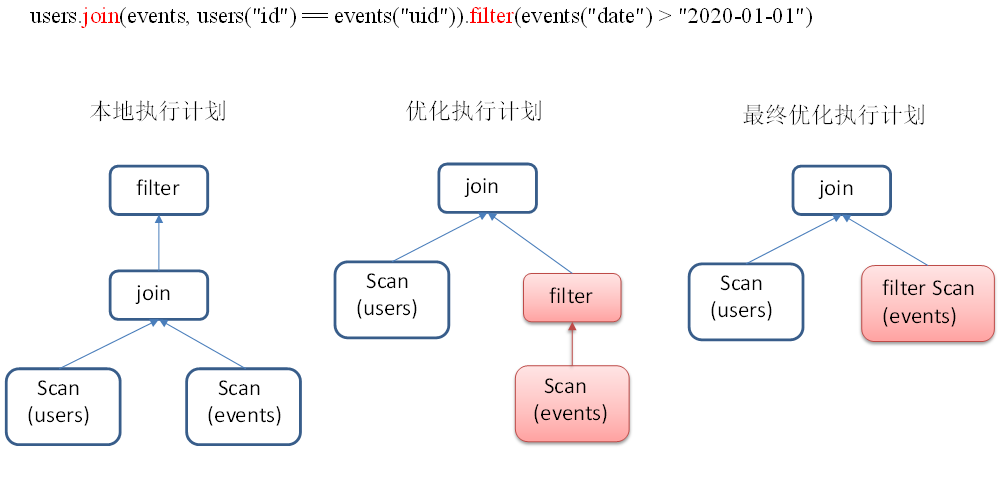
1）在Spark中，DataFrame是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表格。

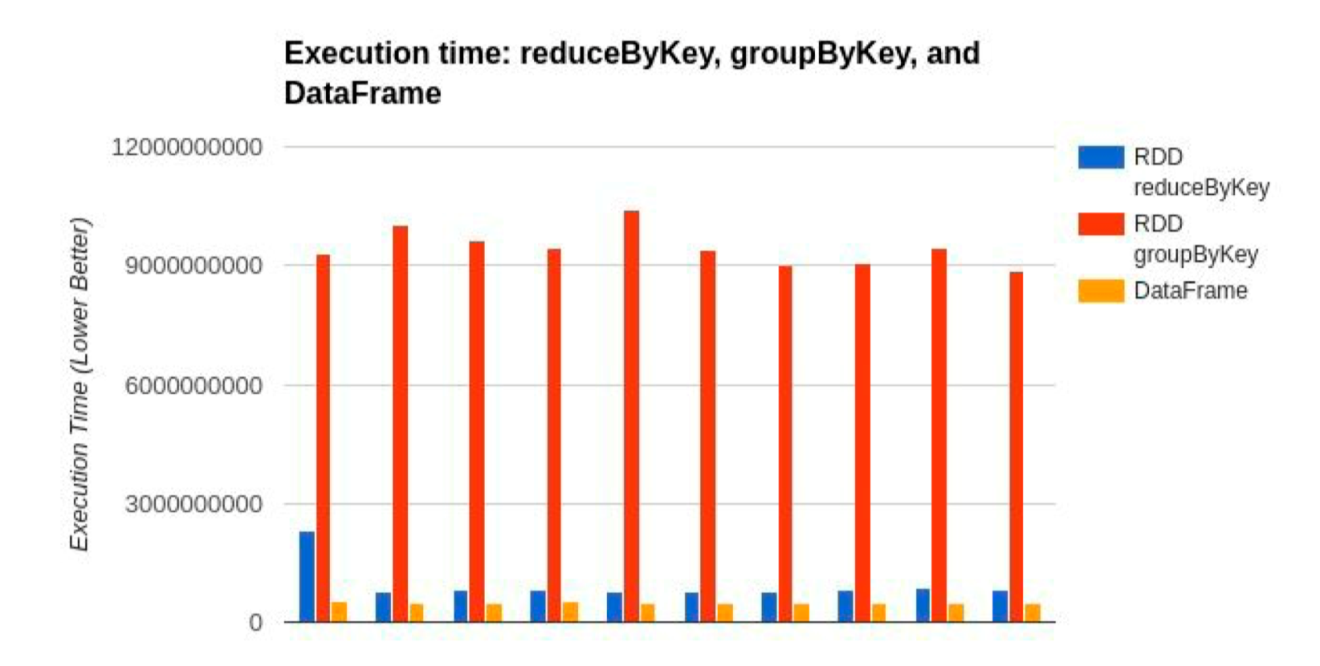
2）DataFrame与RDD的主要区别在于，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有名称和类型。



左侧的RDD[Person]虽然以Person为类型参数，但Spark框架本身不了解Person类的内部结构。而右侧的DataFrame却提供了详细的结构信息，使得Spark SQL可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。

3）Spark SQL性能上比RDD要高。因为Spark SQL了解数据内部结构，从而对藏于DataFrame背后的数据源以及作用于DataFrame之上的变换进行了针对性的优化，最终达到大幅提升运行时效率的目标。反观RDD，由于无从得知所存数据元素的具体内部结构，Spark Core只能在Stage层面进行简单、通用的流水线优化。





4）DataFrameAPI提供的是一套高层的关系操作，比函数式的RDD API要更加友好，门槛更低。

5）DataFrame是为数据提供了Schema的视图，可以把它当做数据库中的一张表来对待。

### 1.3.2 什么是DataSet

DataSet是分布式数据集合。

* DataSet是强类型的。比如可以有DataSet[Car]，DataSet[User]。
* 是DataFrame API的一个扩展，是Spark1.6中添加的一个新的数据抽象
* **DataFrame是DataSet的特列**，type DataFrame=DataSet[Row] ，Row是一个类型，跟Car、User这些的类型一样，所有的表结构信息都用Row来表示。
* 用户友好的API风格，既具有类型安全检查也具有DataFrame的查询优化特性；
* 用样例类来定义DataSet中数据的结构信息，样例类中每个属性的名称直接映射到DataSet中的字段名称；

### 1.3.3 RDD、DataFrame和DataSet之间关系

1）发展历史

RDD（Spark1.0）=》Dataframe（Spark1.3）=》Dataset（Spark1.6）

如果同样的数据都给到这三个数据结构，他们分别计算之后，都会给出相同的结果。不同是的他们的执行效率和执行方式。在后期的Spark版本中，DataSet有可能会逐步取代RDD和DataFrame成为唯一的API接口。

2）三者的共性

（1）RDD、DataFrame、DataSet全都是Spark平台下的分布式弹性数据集，为处理超大型数据提供便利

（2）三者都有惰性机制，在进行创建、转换，如map方法时，不会立即执行，只有在遇到Action如foreach时，三者才会开始遍历运算

（3）三者有许多共同的函数，如filter，排序等

（4）三者都会根据Spark的内存情况自动缓存运算

（5）三者都有分区的概念

## 1.4 Spark SQL的特点

1）易整合

无缝的整合了SQL查询和Spark编程。



2）统一的数据访问方式

使用相同的方式连接不同的数据源。



3）兼容Hive

在已有的仓库上直接运行SQL或者HiveQL。



4）标准的数据连接

通过JDBC或者ODBC来连接



# 第2章 Spark SQL编程

## 2.1 SparkSession新的起始点

在老的版本中，SparkSQL提供两种SQL查询起始点：

* 一个叫SQLContext，用于Spark自己提供的SQL查询；
* 一个叫HiveContext，用于连接Hive的查询。

SparkSession是Spark最新的SQL查询起始点，实质上是SQLContext和HiveContext的组合，所以在SQLContext和HiveContext上可用的API在SparkSession上同样是可以使用的。

SparkSession内部封装了SparkContext，所以计算实际上是由SparkContext完成的。当我们使用spark-shell的时候，Spark框架会自动的创建一个名称叫做Spark的SparkSession，就像我们以前可以自动获取到一个sc来表示SparkContext。

[user1@hadoop102 spark-local]$ bin/spark-shell

20/09/12 11:16:35 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties

Setting default log level to "WARN".

To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

Spark context Web UI available at http://hadoop102:4040

Spark context available as 'sc' (master = local[\*], app id = local-1599880621394).

Spark session available as 'spark'.

Welcome to

\_\_\_\_ \_\_

/ \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_

\_\ \/ \_ \/ \_ `/ \_\_/ '\_/

/\_\_\_/ .\_\_/\\_,\_/\_/ /\_/\\_\ version 3.0.0

/\_/

Using Scala version 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_212)

Type in expressions to have them evaluated.

Type :help for more information.

## 2.2 DataFrame

Spark SQL的DataFrame API允许我们使用DataFrame而不用必须去注册临时表或者生成SQL表达式。DataFrame API既有Transformation操作也有Action操作，DataFrame的转换从本质上来说更具有关系，而DataSet API提供了更加函数式的API。

### 2.2.1 创建DataFrame

在Spark SQL中SparkSession是创建DataFrame和执行SQL的入口，创建DataFrame有三种方式：

* 通过Spark的数据源进行创建；
* 从一个存在的RDD进行转换；
* 还可以从Hive Table进行查询返回。

1）从Spark数据源进行创建

（1）数据准备，在/opt/module/spark-local目录下创建一个user.json文件

{"age":20,"name":"qiaofeng"}

{"age":19,"name":"xuzhu"}

{"age":18,"name":"duanyu"}

（2）查看Spark支持创建文件的数据源格式，使用tab键查看

scala> spark.read.

csv format jdbc json load option options orc parquet schema table text textFile

（3）读取json文件创建DataFrame

scala> val df = spark.read.json("/opt/module/spark-local/user.json")

df: org.apache.spark.sql.DataFrame = [age: bigint， name: string]

注意：如果从内存中获取数据，spark可以知道数据类型具体是什么，如果是数字，默认作为Int处理；但是从文件中读取的数字，不能确定是什么类型，所以用bigint接收，可以和Long类型转换，但是和Int不能进行转换。

（4）展示结果

scala> df.show

+---+--------+

|age| name|

+---+--------+

| 20|qiaofeng|

| 19| xuzhu|

| 18| duanyu|

+---+--------+

2）从RDD进行转换

3）Hive Table进行查询返回

### 2.2.2 SQL风格语法

SQL语法风格是指我们查询数据的时候使用SQL语句来查询，这种风格的查询必须要有临时视图或者全局视图来辅助。

视图：对特定表的数据的查询结果重复使用。View只能查询，不能修改和插入。

select \* from t\_user where age > 30 作为查询结果重复使用; => select \* from v\_user\_age

**1）临时视图**

（1）创建一个DataFrame

scala> val df = spark.read.json("/opt/module/spark-local/user.json")

df: org.apache.spark.sql.DataFrame = [age: bigint， name: string]

（2）对DataFrame创建一个临时视图

scala> df.createOrReplaceTempView("user")

（3）通过SQL语句实现查询全表

scala> val sqlDF = spark.sql("SELECT \* FROM user")

sqlDF: org.apache.spark.sql.DataFrame = [age: bigint， name: string]

（4）结果展示

scala> sqlDF.show

+---+--------+

|age| name|

+---+--------+

| 20|qiaofeng|

| 19| xuzhu|

| 18| duanyu|

+---+--------+

（5）求年龄的平均值

scala> val sqlDF = spark.sql("SELECT avg(age) from user")

sqlDF: org.apache.spark.sql.DataFrame = [avg(age): double]

（6）结果展示

scala> sqlDF.show

+--------+

|avg(age)|

+--------+

| 19.0|

+--------+

（7）创建一个新会话再执行，发现视图找不到

scala> spark.newSession().sql("SELECT avg(age) from user ").show()

org.apache.spark.sql.AnalysisException: Table or view not found: user; line 1 pos 14;

注意：普通临时视图是Session范围内的，如果想全局有效，可以创建全局临时视图。

**2）全局视图**

（1）对于DataFrame创建一个全局视图

scala> df.createGlobalTempView("user")

（2）通过SQL语句实现查询全表

scala> spark.sql("SELECT \* FROM global\_temp.user").show()

+---+--------+

|age| name|

+---+--------+

| 20|qiaofeng|

| 19| xuzhu|

| 18| duanyu|

+---+--------+

（3）新建session，通过SQL语句实现查询全表

scala> spark.newSession().sql("SELECT \* FROM global\_temp.user").show()

+---+--------+

|age| name|

+---+--------+

| 20|qiaofeng|

| 19| xuzhu|

| 18| duanyu|

+---+--------+

### 2.2.3 DSL风格语法

DataFrame提供一个特定领域语言（domain-specific language，DSL）去管理结构化的数据，可以在Scala，Java，Python和R中使用DSL，使用DSL语法风格不必去创建临时视图了。

1）创建一个DataFrame

scala> val df = spark.read.json("/opt/module/spark-local/user.json")

df: org.apache.spark.sql.DataFrame = [age: bigint， name: string]

2）查看DataFrame的Schema信息

scala> df.printSchema

root

|-- age: Long (nullable = true)

|-- name: string (nullable = true)

3）只查看“name”列数据

scala> df.select("name").show()

+--------+

| name|

+--------+

|qiaofeng|

| xuzhu|

| duanyu|

+--------+

4）查看所有列

scala> df.select("\*").show

+---+--------+

|age| name|

+---+--------+

| 20|qiaofeng|

| 19| xuzhu|

| 18| duanyu|

+---+--------+

5）查看“name”列数据以及“age+1”数据

注意：涉及到运算的时候，每列都必须使用$，或者采用引号表达式：单引号+字段名

scala> df.select($"name",$"age" + 1).show

scala> df.select('username, 'age + 1).show()

scala> df.select('username, 'age + 1 as "newage").show()

+--------+---------+

| name|(age + 1)|

+--------+---------+

|qiaofeng| 21|

| xuzhu| 20|

| duanyu| 19|

+--------+---------+

6）查看“age”大于“19”的数据

scala> df.filter($"age">19).show

+---+--------+

|age| name|

+---+--------+

| 20|qiaofeng|

+---+--------+

7）按照“age”分组，查看数据条数

scala> df.groupBy("age").count.show

+---+-----+

|age|count|

+---+-----+

| 19| 1|

| 18| 1|

| 20| 1|

+---+-----+

## 2.3 DataSet

DataSet是具有强类型的数据集合，需要提供对应的类型信息。

### 2.3.1 创建DataSet（基本类型序列）

使用基本类型的序列创建DataSet

（1）将集合转换为DataSet

scala> val ds = Seq(1,2,3,4,5,6).toDS

ds: org.apache.spark.sql.Dataset[Int] = [value: int]

（2）查看DataSet的值

scala> ds.show

+-----+

|value|

+-----+

| 1|

| 2|

| 3|

| 4|

| 5|

| 6|

+-----+

### 2.3.2 创建DataSet（样例类序列）

使用样例类序列创建DataSet

（1）创建一个User的样例类

scala> case class User(name: String, age: Long)

defined class User

（2）将集合转换为DataSet

scala> val caseClassDS = Seq(User("wangyuyan",2)).toDS()

caseClassDS: org.apache.spark.sql.Dataset[User] = [name: string, age: bigint]

（3）查看DataSet的值

scala> caseClassDS.show

+---------+---+

| name|age|

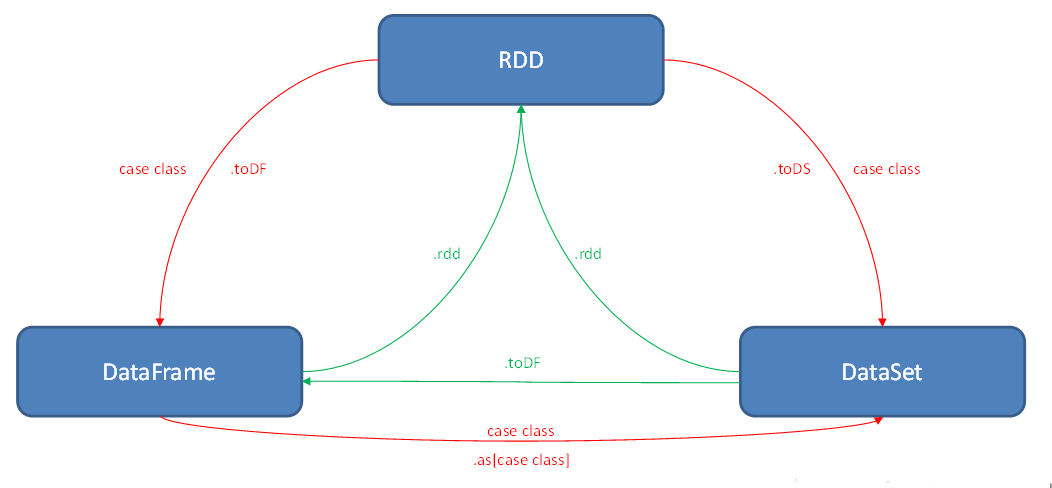
+---------+---+

|wangyuyan| 2|

+---------+---+

注意：在实际使用的时候，很少用到把序列转换成DataSet，更多是通过RDD来得到DataSet

## 2.4 RDD、DataFrame、DataSet相互转换



### 2.4.1 IDEA创建SparkSQL工程

1）创建一个maven工程SparkSQLTest

2）在项目SparkSQLTest上点击右键，Add Framework Support=》勾选scala

3）在main下创建scala文件夹，并右键Mark Directory as Sources Root=>在scala下创建包名为com.user1.sparksql

4）输入文件夹准备：在新建的SparkSQLTest项目名称上右键=》新建input文件夹=》在input文件夹上右键=》新建user.json。并输入如下内容：

{"age":20,"name":"qiaofeng"}  
{"age":19,"name":"xuzhu"}  
{"age":18,"name":"duanyu"}

5）在pom.xml文件中添加如下依赖

<dependencies>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-sql\_2.12</artifactId>

<version>3.0.0</version>

</dependency>

</dependencies>

6）代码实现

package com.user1.sparksql

import org.apache.spark.SparkConf

import org.apache.spark.sql.{DataFrame, SparkSession}

object SparkSQL01\_input {

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

// 1 创建上下文环境配置对象

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")

// 2 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

// 3 读取数据

val df: DataFrame = spark.read.json("input/user.json")

// 4 可视化

df.show()

// 5 释放资源

spark.stop()

}

}

### 2.4.2 RDD与DataFrame相互转换

1）RDD转换为DataFrame

* 手动转：RDD.toDF(“列名1”, “列名2”)
* 通过样例类反射转：UserRDD.map{ x=>User(x.\_1,x.\_2) }.toDF()

2）DataFrame转换为RDD

DataFrame.rdd

3）在input/目录下准备user.txt

qiaofeng,20  
xuzhu,19  
duanyu,18

4）代码实现

package com.user1.sparksql

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.sql.{DataFrame, Row, SparkSession}

object SparkSQL02\_RDDAndDataFrame {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.1 获取数据

val LineRDD: RDD[String] = sc.textFile("input/user.txt")

//3.2 RDD准备完成

val UserRDD: RDD[(String, Int)] = LineRDD.map {

x =>

val fields = x.split(",")

(fields(0), fields(1).trim.toInt)

}

//4. 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

//5.1 RDD和DataFrame、DataSet转换必须要导的包

import spark.implicits.\_

//5.2 RDD转换为DataFrame（手动转）

UserRDD.toDF("name","age").show

//5.3 RDD转换为DataFrame（通过样例类反射转）

val userDataFrame: DataFrame = UserRDD.map {

x => User(x.\_1, x.\_2)

}.toDF()

userDataFrame.show()

//5.4 DataFrame 转换为RDD

val userRDD: RDD[Row] = userDataFrame.rdd

userRDD.collect().foreach(println)

//6.关闭连接

sc.stop()

}

case class User(name:String,age:Int)

}

### 2.4.3 RDD与DataSet相互转换

1）RDD转换为DataSet

RDD.map { x => User(x.\_1, x.\_2) }.toDS()

SparkSQL能够自动将包含有样例类的RDD转换成DataSet，样例类定义了table的结构，样例类属性通过反射变成了表的列名。样例类可以包含诸如Seq或者Array等复杂的结构。

2）DataSet转换为RDD

DS.rdd

3）代码实现

package com.user1.sparksql

import org.apache.spark.rdd.RDD

import org.apache.spark.sql.{Dataset, SparkSession}

import org.apache.spark.{SparkConf, SparkContext}

object SparkSQL03\_RDDAndDataSet {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.1 获取数据

val LineRDD: RDD[String] = sc.textFile("input/user.txt")

//4. 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

//5.1 RDD和DataFrame、DataSet转换必须要导的包

import spark.implicits.\_

//5.2 RDD转换为DataSet

val userDataSet: Dataset[User] = LineRDD.map {

line =>

val fields = line.split(",")

User(fields(0), fields(1).toInt)

}.toDS()

userDataSet.show()

//5.3 DataSet转换为RDD

val userRDD: RDD[User] = userDataSet.rdd

userRDD.collect().foreach(println)

//6.关闭连接

sc.stop()

}

case class User(name:String,age:Int)

}

### 2.4.4 DataFrame与DataSet相互转换

1）DataFrame转为DataSet

df.as[User]

2）Dataset转为DataFrame

ds.toDF

3）代码实现

package com.user1.sparksql  
  
import org.apache.spark.SparkConf  
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}  
  
object SparkSQL04\_DataFrameAndDataSet {  
  
 def main(args: Array[String]): Unit = {  
  
 // 1 创建上下文环境配置对象  
 val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")  
  
 // 2 创建SparkSession对象  
 val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()  
  
 // 3 读取数据  
 val df: DataFrame = spark.read.json("input/user.json")  
  
 //4.1 RDD和DataFrame、DataSet转换必须要导的包  
 import spark.implicits.\_  
  
 // 4.2 DataFrame 转换为DataSet  
 val userDataSet: Dataset[User] = df.as[User]  
 userDataSet.show()  
  
 // 4.3 DataSet转换为DataFrame  
 val userDataFrame: DataFrame = userDataSet.toDF()  
 userDataFrame.show()  
  
 // 5 释放资源  
 spark.stop()  
 }  
  
 case class User(name: String,age: Long)  
}

## 2.5 用户自定义函数

### 2.5.1 UDF

1）UDF：一行进入，一行出

2）代码实现

package com.user1.sparksql  
  
import org.apache.spark.SparkConf  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
object SparkSQL05\_UDF{  
  
 def main(args: Array[String]): Unit = {  
  
 // 1 创建上下文环境配置对象  
 val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")  
  
 // 2 创建SparkSession对象  
 val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()  
  
 // 3 读取数据  
 val df: DataFrame = spark.read.json("input/user.json")  
  
 // 4 注册UDF函数。功能：在数据前添加字符串“Name:”  
 spark.udf.register("addName", (x:String) => "Name:"+ x)  
  
 // 5 创建DataFrame临时视图  
 df.createOrReplaceTempView("user")  
  
 // 6 调用自定义UDF函数  
 spark.sql("select addName(name),age from user").show()  
  
 // 7 释放资源  
 spark.stop()  
 }  
}

### 2.5.2 UDAF

1）UDAF：输入多行，返回一行。

2）Spark3.x推荐使用extends Aggregator自定义UDAF，属于强类型的Dataset方式。

3）Spark2.x使用extends UserDefinedAggregateFunction，属于弱类型的DataFrame

4）案例实操

需求：实现求平均年龄

（1）自定义聚合函数实现-强类型（应用于DataSet的DSL更方便）

package com.user1.sparksql

import org.apache.spark.SparkConf

import org.apache.spark.sql.expressions.Aggregator

import org.apache.spark.sql.{DataFrame, Encoder, Encoders, SparkSession, functions}

object SparkSQL06\_UDAF{

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

// 1 创建上下文环境配置对象

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")

// 2 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

// 3 读取数据

val df: DataFrame = spark.read.json("input/user.json")

// 4 注册UDAF

spark.udf.register("myAvg",functions.udaf(new MyAvgUDAF()))

// 5 创建DataFrame临时视图

df.createOrReplaceTempView("user")

// 6 调用自定义UDAF函数

spark.sql("select myAvg(age) from user").show()

// 7 释放资源

spark.stop()

}

//输入数据类型

case class User(age:Long, name:String)

case class Buff(var sum:Long, var cnt:Long)

/\*\*

\* 1,20岁； 2,19岁； 3,18岁

\* IN:聚合函数的输入类型：Long

\* BUF：

\* OUT:聚合函数的输出类型:Double (18+19+20)/3

\*/

class MyAvgUDAF extends Aggregator[User,Buff,Double]{

// 初始化缓冲区

override def zero: Buff = {

Buff(0L,0L)

}

// 将输入的年龄和缓冲区的数据进行聚合

override def reduce(buff: Buff, user: User): Buff = {

buff.sum = buff.sum + user.age

buff.cnt = buff.cnt + 1

buff

}

// 多个缓冲区数据合并

override def merge(buff1: Buff, buff2: Buff): Buff = {

buff1.sum = buff1.sum + buff2.sum

buff1.cnt = buff1.cnt + buff2.cnt

buff1

}

// 完成聚合操作，获取最终结果

override def finish(buff: Buff): Double = {

buff.sum.toDouble / buff.cnt

}

// sparksql对传递的对象的序列化操作（编码）

//自定义类型就是product 自带类型根据类型选择

override def bufferEncoder: Encoder[Buff] = Encoders.product

override def outputEncoder: Encoder[Double] = Encoders.scalaDouble

}

}

（2）自定义聚合函数实现-弱类型（过时——了解）

package com.user1.sparksql

import com.user1.sparksql.SparkSQL06\_UDAF.Buff

import org.apache.spark.SparkConf

import org.apache.spark.sql.expressions.{Aggregator, MutableAggregationBuffer, UserDefinedAggregateFunction}

import org.apache.spark.sql.\_

import org.apache.spark.sql.types.{DataType, DoubleType, LongType, StructField, StructType}

object SparkSQL07\_UDAF{

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

// 1 创建上下文环境配置对象

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")

// 2 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

// 3 读取数据

val df: DataFrame = spark.read.json("input/user.json")

// 4 注册UDAF

spark.udf.register("myAvg",new MyAvgUDAF())

// 5 创建DataFrame临时视图

df.createOrReplaceTempView("user")

// 6 调用自定义UDAF函数

spark.sql("select myAvg(age) from user").show()

// 7 释放资源

spark.stop()

}

class MyAvgUDAF extends UserDefinedAggregateFunction {

// 聚合函数输入参数的数据类型：age(Long)

override def inputSchema: StructType = {

StructType(Array(

StructField("age",LongType)

))

}

// 聚合函数缓冲区中值的数据类型(age,count)

override def bufferSchema: StructType = {

StructType(Array(

StructField("sum",LongType),

StructField("count",LongType)

))

}

// 函数返回值的数据类型

override def dataType: DataType = DoubleType

// 稳定性：对于相同的输入是否一直返回相同的输出。

override def deterministic: Boolean = true

// 函数缓冲区初始化

override def initialize(buffer: MutableAggregationBuffer): Unit = {

// 存年龄的总和

buffer.update(0, 0L)

// 存年龄的个数

buffer.update(1, 0L)

}

// 更新缓冲区中的数据

override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {

if (!input.isNullAt(0)) {

buffer(0) = buffer.getLong(0) + input.getLong(0)

buffer(1) = buffer.getLong(1) + 1

}

}

// 合并缓冲区

override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {

buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)

buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)

}

// 计算最终结果

override def evaluate(buffer: Row): Any = {

buffer.getLong(0).toDouble / buffer.getLong(1)

}

}

}

### 2.5.3 UDTF（没有）

输入一行，返回多行（Hive）；

SparkSQL中没有UDTF，Spark中用flatMap即可实现该功能

# 第3章 SparkSQL数据的加载与保存

## 3.1 加载数据

1）加载数据通用方法

spark.read.load是加载数据的通用方法

2）代码实现

package com.user1.sparksql  
  
import org.apache.spark.SparkConf  
import org.apache.spark.sql.\_  
import org.apache.spark.sql.expressions.{MutableAggregationBuffer, UserDefinedAggregateFunction}  
import org.apache.spark.sql.types.\_  
  
object SparkSQL08\_Load{  
  
 def main(args: Array[String]): Unit = {  
  
 // 1 创建上下文环境配置对象  
 val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")  
  
 // 2 创建SparkSession对象  
 val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()  
  
 // 3.1 spark.read直接读取数据：csv format jdbc json load option  
 // options orc parquet schema table text textFile  
 // 注意：加载数据的相关参数需写到上述方法中，  
 // 如：textFile需传入加载数据的路径，jdbc需传入JDBC相关参数。  
 spark.read.json("input/user.json").show()  
  
 // 3.2 format指定加载数据类型  
 // spark.read.format("…")[.option("…")].load("…")  
 // format("…")：指定加载的数据类型，包括"csv"、"jdbc"、"json"、"orc"、"parquet"和"textFile"  
 // load("…")：在"csv"、"jdbc"、"json"、"orc"、"parquet"和"textFile"格式下需要传入加载数据的路径  
 // option("…")：在"jdbc"格式下需要传入JDBC相应参数，url、user、password和dbtable  
 spark.read.format("json").load ("input/user.json").show  
  
 // 3.3 在文件上直接运行SQL（了解）  
 // 说明: json表示文件的格式。后面的文件具体路径需要用反引号括起来。  
 spark.sql("select \* from json.`input/user.json`").show  
  
 // 4 释放资源  
 spark.stop()  
 }  
}

## 3.2 保存数据

1）保存数据通用方法

df.write.save是保存数据的通用方法

2）代码实现

package com.user1.sparksql  
  
import org.apache.spark.SparkConf  
import org.apache.spark.sql.\_  
  
object SparkSQL09\_Save{  
  
 def main(args: Array[String]): Unit = {  
  
 // 1 创建上下文环境配置对象  
 val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")  
  
 // 2 创建SparkSession对象  
 val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()  
  
 // 3 获取数据  
 val df: DataFrame = spark.read.json("input/user.json")  
  
 // 4.1 df.write.保存数据：csv jdbc json orc parquet textFile… …  
 // 注意：保存数据的相关参数需写到上述方法中。如：textFile需传入加载数据的路径，JDBC需传入JDBC相关参数。  
 // 默认保存为parquet文件  
 df.write.save("output")

// 默认读取文件parquet

// 修改配置项spark.sql.sources.default，可修改默认数据源格式。

spark.read.load("output").show()  
  
 // 4.2 可以指定为保存格式，直接保存，不需要再调用save了  
 df.write.json("output1")  
  
 // 4.3 format指定保存数据类型  
 // df.write.format("…")[.option("…")].save("…")  
 // format("…")：指定保存的数据类型，包括"csv"、"jdbc"、"json"、"orc"、"parquet"和"textFile"。  
 // save ("…")：在"csv"、"orc"、"parquet"和"textFile"格式下需要传入保存数据的路径。  
 // option("…")：在"jdbc"格式下需要传入JDBC相应参数，url、user、password和dbtable  
 df.write.format("json").save("output2")  
  
 // 4.4 如果文件已经存在则追加  
 df.write.mode("append").json("output2")  
  
 // 如果文件已经存在则忽略  
 df.write.mode("ignore").json("output2")  
  
 // 如果文件已经存在则覆盖  
 df.write.mode("overwrite").json("output2")  
  
 // 默认default:如果文件已经存在则抛出异常  
 // path file:/E:/ideaProject2/SparkSQLTest/output2 already exists.;  
 df.write.mode("error").json("output2")  
  
 // 5 释放资源  
 spark.stop()  
 }  
}

## 3.3 与MySQL交互

1）导入依赖

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.27</version>

</dependency>

2）从MySQL读数据

package com.user1.sparksql

import org.apache.spark.SparkConf

import org.apache.spark.sql.\_

object SparkSQL10\_MySQL\_Read{

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

// 1 创建上下文环境配置对象

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")

// 2 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

import spark.implicits.\_

// 3.1 通用的load方法读取

val df: DataFrame = spark.read.format("jdbc")

.option("url", "jdbc:mysql://hadoop102:3306/gmall")

.option("driver", "com.mysql.jdbc.Driver")

.option("user", "root")

.option("password", "000000")

.option("dbtable", "user\_info")

.load()

// 3.2 创建视图

df.createOrReplaceTempView("user")

// 3.3 查询想要的数据

spark.sql("select id, name from user").show()

// 4 释放资源

spark.stop()

}

}

3）向MySQL写数据

package com.user1.sparksql

import org.apache.spark.SparkConf

import org.apache.spark.rdd.RDD

import org.apache.spark.sql.\_

object SparkSQL11\_MySQL\_Write {

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

// 1 创建上下文环境配置对象

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")

// 2 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

import spark.implicits.\_

// 3 准备数据

// 注意：id是主键，不能和MySQL数据库中的id重复

val rdd: RDD[User] = spark.sparkContext.makeRDD(List(User(3000, "zhangsan"), User(3001, "lisi")))

val ds: Dataset[User] = rdd.toDS

// 4 向MySQL中写入数据

ds.write

.format("jdbc")

.option("url", "jdbc:mysql://hadoop102:3306/gmall")

.option("user", "root")

.option("password", "000000")

.option("dbtable", "user\_info")

.mode(SaveMode.Append)

.save()

// 5 释放资源

spark.stop()

}

case class User(id: Int, name: String)

}

## 3.4 与Hive交互

SparkSQL可以采用内嵌Hive，也可以采用外部Hive。企业开发中，通常采用外部Hive。

### 3.4.1 内嵌Hive应用

1）如果使用Spark内嵌的Hive，则什么都不用做，直接使用即可。

scala> spark.sql("show tables").show

注意：执行完后，发现多了$SPARK\_HOME/metastore\_db，用于存储元数据

2）创建一个数据库

scala> spark.sql("create table user(id int, name string)")

注意：执行完后，发现多了$SPARK\_HOME/spark-warehouse/user，用于存储数据库数据

3）查看数据库

scala> spark.sql("show tables").show

4）向表中插入数据

scala> spark.sql("insert into user values(1,'zs')")

5）查询数据

scala> spark.sql("select \* from user").show

注意：然而在实际使用中，几乎没有任何人会使用内置的Hive

### 3.4.2 外部Hive应用

如果Spark要接管Hive外部已经部署好的Hive，需要通过以下几个步骤。

0）为了说明内嵌Hive和外部Hive区别：删除内嵌Hive的metastore\_db和spark-warehouse

[user1@hadoop102 spark-local]$ rm -rf metastore\_db/ spark-warehouse/

1）确定原有Hive是正常工作的

[user1@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh

[user1@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh

[user1@hadoop102 hive]$ bin/hive

2）需要把hive-site.xml拷贝到spark的conf/目录下

[user1@hadoop102 conf]$ cp hive-site.xml /opt/module/spark-local/conf/

3）如果以前hive-site.xml文件中，配置过Tez相关信息，注释掉（不是必须）

4）把MySQL的驱动copy到Spark的jars/目录下

[user1@hadoop102 software]$ cp mysql-connector-java-5.1.48.jar /opt/module/spark-local/jars/

5）需要提前启动hive服务，/opt/module/hive/bin/hiveservices.sh start（不是必须）

6）如果访问不到HDFS，则需把core-site.xml和hdfs-site.xml拷贝到conf/目录（不是必须）

7）启动 spark-shell

[user1@hadoop102 spark-local]$ bin/spark-shell

8）查询表

scala> spark.sql("show tables").show

9）创建一个数据库

scala> spark.sql("create table user(id int, name string)")

10）向表中插入数据

scala> spark.sql("insert into user values(1,'zs')")

11）查询数据

scala> spark.sql("select \* from user").show

### 3.4.3 运行Spark SQL CLI

Spark SQL CLI可以很方便的在本地运行Hive元数据服务以及从命令行执行查询任务。在Spark目录下执行如下命令启动Spark SQL CLI，直接执行SQL语句，类似Hive窗口。

[user1@hadoop102 spark-local]$ bin/spark-sql

### IDEA操作Hive

1）添加依赖

<dependencies>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-sql\_2.12</artifactId>

<version>3.0.0</version>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-hive\_2.12</artifactId>

<version>3.0.0</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.27</version>

</dependency>

</dependencies>

2）拷贝hive-site.xml到resources目录（如果需要操作Hadoop，需要拷贝hdfs-site.xml、core-site.xml、yarn-site.xml）

3）代码实现

package com.user1.sparksql

import org.apache.spark.SparkConf

import org.apache.spark.sql.\_

object SparkSQL12\_Hive {

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

System.setProperty("HADOOP\_USER\_NAME","user1")

// 1 创建上下文环境配置对象

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQLTest")

// 2 创建SparkSession对象

val spark: SparkSession = SparkSession.builder().enableHiveSupport().config(conf).getOrCreate()

import spark.implicits.\_

// 3 连接外部Hive，并进行操作

spark.sql("show tables").show()

spark.sql("create table user3(id int, name string)")

spark.sql("insert into user3 values(1,'zs')")

spark.sql("select \* from user3").show

// 4 释放资源

spark.stop()

}

}

# 第4章 SparkSQL项目实战

## 4.1 准备数据

我们这次Spark-sql操作所有的数据均来自Hive，首先在Hive中创建表，并导入数据。一共有3张表：1张用户行为表，1张城市表，1张产品表。

1）将city\_info.txt、product\_info.txt、user\_visit\_action.txt上传到/opt/module/data

[user1@hadoop102 module]$ mkdir data

2）将创建对应的三张表

spark-sql (default)>

CREATE TABLE `user\_visit\_action`(  
 `date` string,  
 `user\_id` bigint,  
 `session\_id` string,  
 `page\_id` bigint,  
 `action\_time` string,  
 `search\_keyword` string,  
 `click\_category\_id` bigint,  
 `click\_product\_id` bigint, --点击商品id，没有商品用-1表示。  
 `order\_category\_ids` string,  
 `order\_product\_ids` string,  
 `pay\_category\_ids` string,  
 `pay\_product\_ids` string,  
 `city\_id` bigint

)  
row format delimited fields terminated by '\t';

CREATE TABLE `city\_info`(  
 `city\_id` bigint,  
 `city\_name` string,  
 `area` string

)  
row format delimited fields terminated by '\t';

CREATE TABLE `product\_info`(  
 `product\_id` bigint,  
 `product\_name` string,  
 `extend\_info` string

)  
row format delimited fields terminated by '\t';

3）并加载数据

spark-sql (default)>

load data local inpath '/opt/module/data/user\_visit\_action.txt' into table user\_visit\_action;  
load data local inpath '/opt/module/data/product\_info.txt' into table product\_info;  
load data local inpath '/opt/module/data/city\_info.txt' into table city\_info;

4）测试一下三张表数据是否正常

spark-sql (default)>

select \* from user\_visit\_action limit 5;

select \* from product\_info limit 5;

select \* from city\_info limit 5;

## 4.2 需求：各区域热门商品Top3

### 4.2.1 需求简介

这里的热门商品是从点击量的维度来看的，计算各个区域前三大热门商品，并备注上每个商品在主要城市中的分布比例，超过两个城市用其他显示。

例如：

|  |  |  |  |
| --- | --- | --- | --- |
| **地区** | **商品名称** | **点击次数** | **城市备注** |
| **华北** | 商品A | 100000 | 北京21.2%，天津13.2%，其他65.6% |
| **华北** | 商品P | 80200 | 北京63.0%，太原10%，其他27.0% |
| **华北** | 商品M | 40000 | 北京63.0%，太原10%，其他27.0% |
| **东北** | 商品J | 92000 | 大连28%，辽宁17.0%，其他 55.0% |

### 4.2.2 思路分析

* 使用sql来完成，碰到复杂的需求，可以使用udf或udaf

1）查询出来所有的点击记录，并与city\_info表连接，得到每个城市所在的地区，与 Product\_info表连接得到产品名称

2）按照地区和商品名称分组，统计出每个商品在每个地区的总点击次数

3）每个地区内按照点击次数降序排列

4）只取前三名，并把结果保存在数据库中

5）城市备注需要自定义UDAF函数

### 4.2.3 代码实现

package com.user1.sparksql

import org.apache.spark.sql.{Encoder, Encoders, SparkSession, functions}

import org.apache.spark.sql.expressions.Aggregator

import scala.collection.mutable

import scala.collection.mutable.ListBuffer

object SparkSQL12\_TopN {

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

val spark: SparkSession = SparkSession.builder()

.master("local[\*]")

.appName("SparkSQLTest")

.enableHiveSupport()

.getOrCreate()

spark.sql("use default")

// 0 注册自定义聚合函数

spark.udf.register("city\_remark", functions.udaf(new CityRemarkUDAF()))

// 1. 查询出所有的点击记录,并和城市表产品表做内连接

spark.sql(

"""

|select

| v.click\_product\_id,

| c.area,

| c.city\_name,

| p.product\_name

|from user\_visit\_action v

|join city\_info c

|on v.city\_id = c.city\_id

|join product\_info p

|on v.click\_product\_id = p.product\_id

|where click\_product\_id > -1

|""".stripMargin).createOrReplaceTempView("t1")

// 2. 计算每个区域, 每个产品的点击量

spark.sql(

"""

|select

| t1.area,

| t1.product\_name,

| count(\*) click\_count,

| city\_remark(t1.city\_name)

|from t1

|group by t1.area, t1.product\_name

""".stripMargin).createOrReplaceTempView("t2")

// 3. 对每个区域内产品的点击量进行倒序排列

spark.sql(

"""

|select

| \*,

| rank() over(partition by t2.area order by t2.click\_count desc) rank

|from t2

""".stripMargin).createOrReplaceTempView("t3")

// 4. 每个区域取top3

spark.sql(

"""

|select

| \*

|from t3

|where rank <= 3

""".stripMargin).show

// 5. 释放资源

spark.stop()

}

case class Buffer(var totalcnt:Long, var cityMap: mutable.Map[String, Long])

/\*\*

\* IN: 城市名称 String

\* BUFF: Map[(cityName, 点击数量)], totalcnt

\* OUT:城市备注 String

\*/

class CityRemarkUDAF extends Aggregator[String, Buffer,String]{

override def zero: Buffer = {

Buffer(0L,mutable.Map[String, Long]())

}

override def reduce(buffer: Buffer, city: String): Buffer = {

buffer.totalcnt += 1

val newCount = buffer.cityMap.getOrElse(city, 0L) + 1

buffer.cityMap.update(city, newCount)

buffer

}

override def merge(b1: Buffer, b2: Buffer): Buffer = {

// 合并所有城市的点击数量的总和

b1.totalcnt += b2.totalcnt

// 合并城市Map(2个Map合并)

val map1 = b1.cityMap

val map2 = b2.cityMap

map2.foreach{

case(city, cnt) =>{

val newCnt = map1.getOrElse(city, 0L) + cnt

map1.update(city, newCnt)

}

}

b1.cityMap = map1

b1

}

// 计算结果：字符串（每个城市的点击比率）

override def finish(buffer: Buffer): String = {

val remarkList = ListBuffer[String]()

val totalcnt: Long = buffer.totalcnt

val cityMap = buffer.cityMap

// 将统计的城市点击数量的集合进行排序

val cityCountList: List[(String, Long)] = buffer.cityMap.toList.sortWith(

(left, right) => {

left.\_2 > right.\_2

}

).take(2)

// 判断是否存在其他的城市

val hasOtherCity = cityMap.size > 2

var sum = 0L

cityCountList.foreach{

case (city, cnt) =>{

val r = cnt \* 100 / totalcnt

remarkList.append(city +" " + r + "%")

sum += r

}

}

// 判断是否存在其他的城市

if (hasOtherCity){

remarkList.append("其他 "+ (100 - sum) +"%")

}

remarkList.mkString(",")

}

override def bufferEncoder: Encoder[Buffer] = Encoders.product

override def outputEncoder: Encoder[String] = Encoders.STRING

}

}