尚硅谷大数据技术之SparkSQL

(作者：尚硅谷大数据研发部)

版本：V2.0

# Spark SQL概述

## 什么是Spark SQL

Spark SQL是Spark用于结构化数据(structured data)处理的Spark模块。

与基本的Spark RDD API不同，Spark SQL的抽象数据类型为Spark提供了关于数据结构和正在执行的计算的更多信息。

在内部，Spark SQL使用这些额外的信息去做一些额外的优化，有多种方式与Spark SQL进行交互，比如: SQL和DatasetAPI。

当计算结果的时候，使用的是相同的执行引擎，不依赖你正在使用哪种API或者语言。这种统一也就意味着开发者可以很容易在不同的API之间进行切换，这些API提供了最自然的方式来表达给定的转换。

我们已经学习了Hive，它是将Hive SQL转换成 MapReduce然后提交到集群上执行，大大简化了编写 MapReduce的程序的复杂性，由于MapReduce这种计算模型执行效率比较慢。所以Spark SQL的应运而生，它是将Spark SQL转换成RDD，然后提交到集群执行，执行效率非常快！

Spark SQL它提供了2个编程抽象，类似Spark Core中的RDD

* **DataFrame**
* **DataSet**

## Spark SQL的特点

### 易整合

无缝的整合了 SQL 查询和 Spark 编程



### 统一的数据访问方式

使用相同的方式连接不同的数据源



### 兼容Hive

在已有的仓库上直接运行 SQL 或者 HiveQL



### 标准的数据连接

通过 JDBC 或者 ODBC 来连接



## 什么是DataFrame

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。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。这使得Spark SQL得以洞察更多的结构信息，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。

同时，与Hive类似，DataFrame也支持嵌套数据类型（struct、array和map）。从 API 易用性的角度上看，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。



上图直观地体现了DataFrame和RDD的区别。

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。

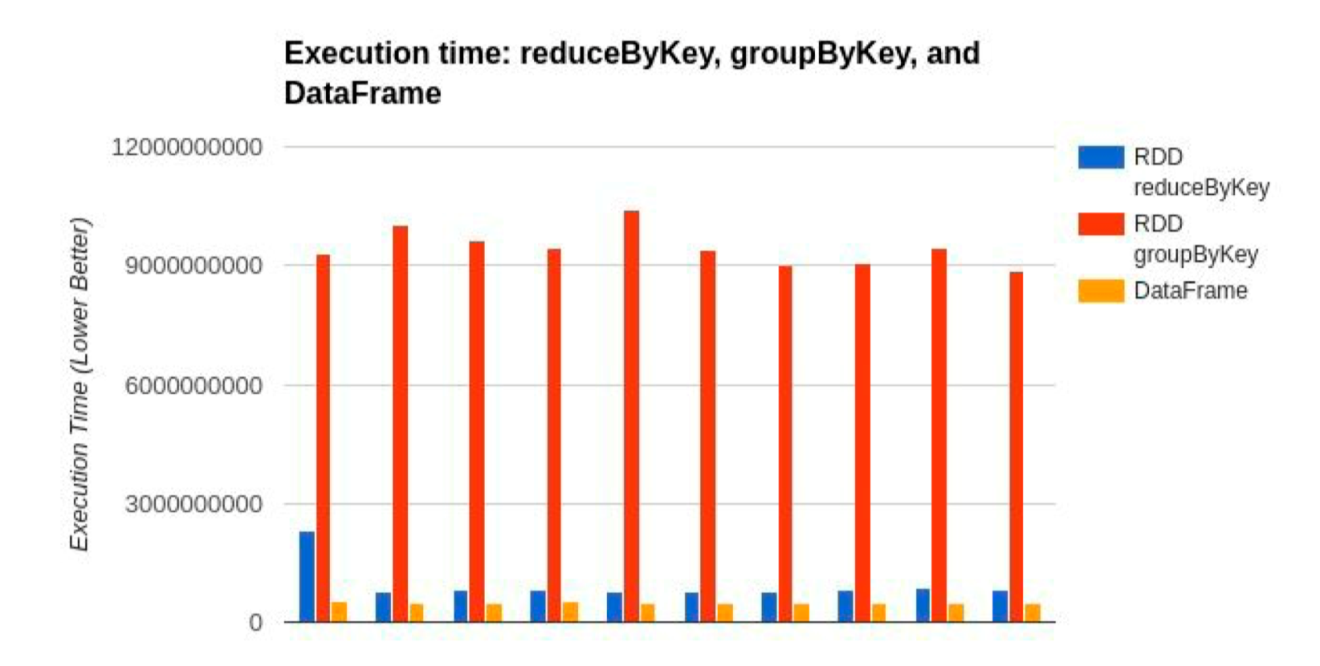
DataFrame是为数据提供了Schema的视图。可以把它当做数据库中的一张表来对待

DataFrame也是懒执行的，但性能上比RDD要高，主要原因：优化的执行计划，即查询计划通过Spark catalyst optimiser进行优化。比如下面一个例子:



为了说明查询优化，我们来看上图展示的人口数据分析的示例。图中构造了两个DataFrame，将它们join之后又做了一次filter操作。

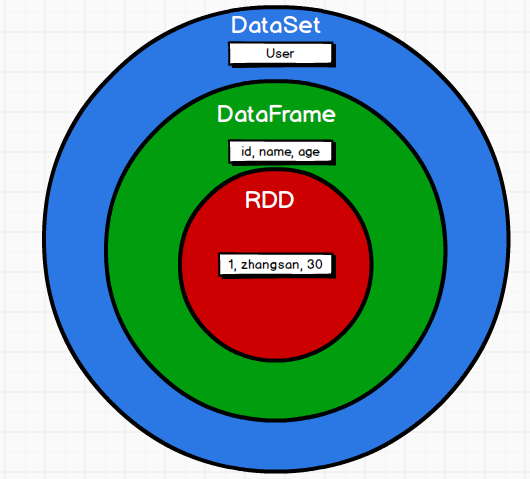
如果原封不动地执行这个执行计划，最终的执行效率是不高的。因为join是一个代价较大的操作，也可能会产生一个较大的数据集。如果我们能将filter下推到 join下方，先对DataFrame进行过滤，再join过滤后的较小的结果集，便可以有效缩短执行时间。而Spark SQL的查询优化器正是这样做的。简而言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本操作的过程。



## 什么是DataSet

DataSet是分布式数据集合。DataSet是Spark 1.6中添加的一个新抽象，是DataFrame的一个扩展。它提供了RDD的优势（强类型，使用强大的lambda函数的能力）以及Spark SQL优化执行引擎的优点。DataSet也可以使用功能性的转换（操作map，flatMap，filter等等）。

* 是DataFrame API的一个扩展，是SparkSQL最新的数据抽象
* 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；
* DataSet是强类型的。比如可以有DataSet[Car]，DataSet[Person]。
* **DataFrame是DataSet的特列**，DataFrame=DataSet[Row] ，所以可以通过as方法将DataFrame转换为DataSet。Row是一个类型，跟Car、Person这些的类型一样，所有的表结构信息都用Row来表示。



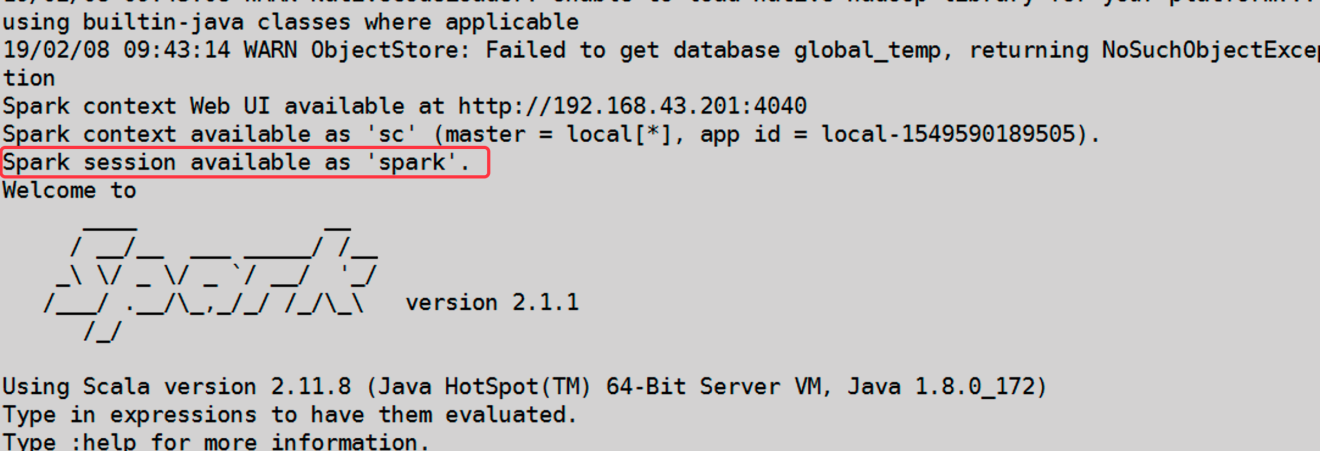
# Spark SQL编程

本章重点学习如何使用 DataFrame和DataSet进行编程，已经他们之间的关系和转换，关于具体的SQL书写不是本章的重点。

## 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SQLContex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 会自动的创建一个叫做spark的SparkSession, 就像我们以前可以自动获取到一个sc来表示SparkContext



## 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

### 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

* 查看Spark支持创建文件的数据源格式

scala> spark.read.

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

* 读取json文件创建DataFrame

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

* 展示结果

scala> df.show

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

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

1. 从RDD进行转换

2.5节我们专门讨论

1. Hive Table进行查询返回

3.4节我们专门讨论

### 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

1. 创建一个DataFrame

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

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

1. 对DataFrame创建一个临时表

scala> df.createOrReplaceTempView("people")

1. 通过SQL语句实现查询全表

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

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

1. 结果展示

scala> sqlDF.show

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

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

注意：普通临时表是Session范围内的，如果想应用范围内有效，可以使用全局临时表。使用全局临时表时需要全路径访问，如：global\_temp.people

1. 对于DataFrame创建一个全局表

scala> df.createGlobalTempView("people")

1. 通过SQL语句实现查询全表

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

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

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

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

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

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

### 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 /people.json")

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

1. 查看DataFrame的Schema信息

scala> df.printSchema

root

|-- age: Long (nullable = true)

|-- name: string (nullable = true)

1. 只查看”name”列数据

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

+--------+

| name|

+--------+

|qiaofeng|

| duanyu|

| xuzhu|

+--------+

1. 查看所有列

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

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

| name |age|

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

|qiaofeng| 18|

| duanyu| 19|

| xuzhu| 20|

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

1. 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$

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

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

| name|(age + 1)|

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

|qiaofeng| 19|

| duanyu| 20|

| xuzhu| 21|

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

1. 查看”age”大于”19”的数据

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

+---+-----+

|age| name|

+---+-----+

| 20|xuzhu|

+---+-----+

1. 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|

+---+-----+

### RDD转换为DataFrame

注意：如果需要RDD与DF或者DS之间操作，那么都需要引入 import spark.implicits.\_ （spark不是包名，而是sparkSession对象的名称，所以必须先创建SparkSession对象再导入. implicits是一个内部object）

前置条件

* 导入隐式转换并创建一个RDD
* 在/opt/module/spark-local/目录下准备people.txt

qiaofeng,18

xuzhu,19

duanyu,20

scala> import spark.implicits.\_

import spark.implicits.\_

scala> val peopleRDD = sc.textFile("/opt/module/spark-local/people.txt")

**输出**

peopleRDD: org.apache.spark.rdd.RDD[String] = /opt/module/spark-local /people.txt MapPartitionsRDD[3] at textFile at <console>:27

1. 通过手动确定转换

scala> peopleRDD.map{x=> val fields=x.split(",");(fields(0),fields(1).trim.toInt)}.toDF("name","age").show

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

| name|age|

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

|qiaofeng| 18|

| xuzhu| 19|

| duanyu| 20|

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

1. 通过样例类反射转换（常用）

* 创建一个样例类

scala> case class People(name:String,age:Int)

* 根据样例类将RDD转换为DataFrame

scala> peopleRDD.map{x=> var fields=x.split(",");People(fields(0),fields(1).toInt)}.toDF.show

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

| name|age|

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

|qiaofeng| 18|

| xuzhu| 19|

| duanyu| 20|

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

1. 通过编程的方式（了解，一般编程直接操作RDD较少，操作hive或数据文件等较多）

package day05  
  
import org.apache.spark.SparkContext  
import org.apache.spark.rdd.RDD  
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}  
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}  
  
object DataFrameDemo2 {  
 def main(args: Array[String]): Unit = {  
 val spark: SparkSession = SparkSession.builder()  
 .master("local[\*]")  
 .appName("Word Count")  
 .getOrCreate()  
 val sc: SparkContext = spark.sparkContext  
 val rdd: RDD[(String, Int)] = sc.parallelize(Array(("lisi", 10), ("zs", 20), ("zhiling", 40)))  
 // 映射出来一个 RDD[Row], 因为 DataFrame其实就是 DataSet[Row]  
 val rowRdd: RDD[Row] = rdd.map(x => Row(x.\_1, x.\_2))  
 // 创建 StructType 类型  
 val types = StructType(Array(StructField("name", StringType), StructField("age", IntegerType)))  
 val df: DataFrame = spark.createDataFrame(rowRdd, types)  
 df.show  
 }  
}

### DataFrame转换为RDD

直接调用rdd即可

1. 创建一个DataFrame

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

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

1. 将DataFrame转换为RDD注意：得到的RDD存储类型为Row

scala> val dfToRDD = df.rdd

dfToRDD: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[19] at rdd at <console>:29

1. 打印RDD

scala> dfToRDD.collect

res3: Array[org.apache.spark.sql.Row] = Array([18,qiaofeng], [19,duanyu], [20,xuzhu])

## DataSet

DataSet是具有强类型的数据集合，需要提供对应的类型信息。

### 创建DataSet

1. 使用样例类序列创建DataSet

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

defined class Person

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

caseClassDS: org.apache.spark.sql.Dataset[Person] = [name: string, age: Long]

scala> caseClassDS.show

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

| name|age|

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

|wangyuyan| 2|

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

1. 使用基本类型的序列创建DataSet

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

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

scala> ds.show

+-----+

|value|

+-----+

| 1|

| 2|

| 3|

| 4|

| 5|

| 6|

+-----+

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

### RDD转换为DataSet

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。

1. 创建一个RDD

scala> val peopleRDD = sc.textFile("/opt/module/spark-local/people.txt")

peopleRDD: org.apache.spark.rdd.RDD[String] = /opt/module/spark-local/people.txt MapPartitionsRDD[19] at textFile at <console>:24

1. 创建一个样例类

scala> case class Person(name:String,age:Int)

defined class Person

1. 将RDD转化为DataSet

scala> peopleRDD.map(line => {val fields = line.split(",");Person(fields(0),fields(1). toInt)}).toDS

res0: org.apache.spark.sql.Dataset[Person] = [name: string, age: Long]

### DataSet转换为RDD

调用rdd方法即可。

1. 创建一个DataSet

scala> val DS = Seq(Person("zhangcuishan", 32)).toDS()

DS: org.apache.spark.sql.Dataset[Person] = [name: string, age: Long]

1. 将DataSet转换为RDD

scala> DS.rdd

res1: org.apache.spark.rdd.RDD[Person] = MapPartitionsRDD[6] at rdd at <console>:28

## DataFrame与DataSet的互操作

### DataFrame转为DataSet

1. 创建一个DateFrame

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

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

1. 创建一个样例类

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

defined class Person

1. 将DataFrame转化为DataSet

scala> df.as[Person]

res5: org.apache.spark.sql.Dataset[Person] = [age: bigint, name: string]

这种方法就是在给出每一列的类型后，使用as方法，转成Dataset，这在数据类型是DataFrame又需要针对各个字段处理时极为方便。在使用一些特殊的操作时，一定要加上 import spark.implicits.\_ 不然toDF、toDS无法使用。

### Dataset转为DataFrame

1. 创建一个样例类

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

defined class Person

1. 创建DataSet

scala> val ds = Seq(Person("zhangwuji",32)).toDS()

ds: org.apache.spark.sql.Dataset[Person] = [name: string, age: bigint]

1. 将DataSet转化为DataFrame

scala> var df = ds.toDF

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

1. 展示

scala> df.show

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

| name|age|

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

|zhangwuji| 32|

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

## RDD、DataFrame和DataSet之间的关系

在SparkSQL中Spark为我们提供了两个新的抽象，分别是DataFrame和DataSet。他们和RDD有什么区别呢？首先从版本的产生上来看：

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。

### 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. 在对DataFrame和Dataset进行操作许多操作都需要这个包:import spark.implicits.\_（在创建好SparkSession对象后尽量直接导入）
5. 三者都会根据 Spark 的内存情况自动缓存运算，这样即使数据量很大，也不用担心会内存溢出
6. 三者都有partition的概念
7. DataFrame和Dataset均可使用模式匹配获取各个字段的值和类型

### 三者的区别

1. RDD
   * RDD一般和Spark MLib同时使用
   * RDD不支SparkSQL操作
2. DataFrame

* 与RDD和Dataset不同，DataFrame每一行的类型固定为Row，每一列的值没法直接访问，只有通过解析才能获取各个字段的值
* DataFrame与DataSet一般不与 Spark MLib 同时使用
* DataFrame与DataSet均支持 SparkSQL 的操作，比如select，groupby之类，还能注册临时表/视窗，进行 sql 语句操作
* DataFrame与DataSet支持一些特别方便的保存方式，比如保存成csv，可以带上表头，这样每一列的字段名一目了然(后面专门讲解)

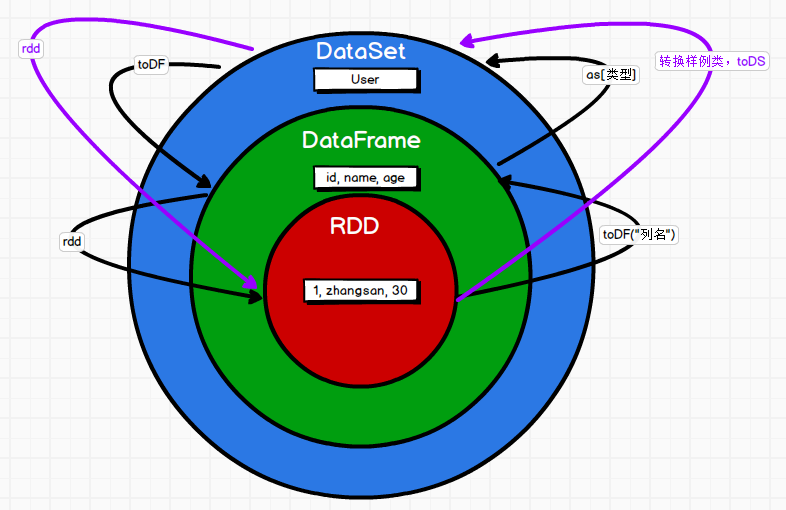
1. DataSet

* Dataset和DataFrame拥有完全相同的成员函数，区别只是每一行的数据类型不同。 DataFrame其实就是DataSet的一个特例

type DataFrame = Dataset[Row]

* DataFrame也可以叫Dataset[Row],每一行的类型是Row，不解析，每一行究竟有哪些字段，各个字段又是什么类型都无从得知，只能用上面提到的getAS方法或者共性中的第七条提到的模式匹配拿出特定字段。而Dataset中，每一行是什么类型是不一定的，在自定义了case class之后可以很自由的获得每一行的信息

### 三者的互相转化



## IDEA创建SparkSQL程序

1. 添加依赖

<dependency>

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

<artifactId>spark-sql\_2.11</artifactId>

<version>2.1.1</version>

</dependency>

1. 代码实现

**object** SparkSQL01\_Demo {

**def** main(args: Array[String]): Unit = {  
 //创建上下文环境配置对象  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("SparkSQL01\_Demo")  
  
 //创建SparkSession对象  
 **val** spark: SparkSession = SparkSession.*builder*().config(conf).getOrCreate()  
 //RDD=>DataFrame=>DataSet转换需要引入隐式转换规则，否则无法转换  
 //spark不是包名，是上下文环境对象名  
 **import** spark.implicits.\_  
  
 //读取json文件 创建DataFrame {"username": "lisi","age": 18}  
 **val** df: DataFrame = spark.read.json("D:\\dev\\workspace\\spark-bak\\spark-bak-00\\input\\test.json")  
 //df.show()  
  
 //SQL风格语法  
 df.createOrReplaceTempView("user")  
 //spark.sql("select avg(age) from user").show  
  
 //DSL风格语法  
 //df.select("username","age").show()  
  
 //\*\*\*\*\*RDD=>DataFrame=>DataSet\*\*\*\*\*  
 //RDD  
 **val** rdd1: RDD[(Int, String, Int)] = spark.sparkContext.makeRDD(*List*((1,"qiaofeng",30),(2,"xuzhu",28),(3,"duanyu",20)))  
  
 //DataFrame  
 **val** df1: DataFrame = rdd1.toDF("id","name","age")  
 //df1.show()  
  
 //DateSet  
 **val** ds1: Dataset[User] = df1.as[User]  
 //ds1.show()  
  
 //\*\*\*\*\*DataSet=>DataFrame=>RDD\*\*\*\*\*  
 //DataFrame  
 **val** df2: DataFrame = ds1.toDF()  
  
 //RDD 返回的RDD类型为Row，里面提供的getXXX方法可以获取字段值，类似jdbc处理结果集，但是索引从0开始  
 **val** rdd2: RDD[Row] = df2.*rdd* //rdd2.foreach(a=>println(a.getString(1)))  
  
 //\*\*\*\*\*RDD=>DataSe\*\*\*\*\*  
 rdd1.map{  
 **case** (id,name,age)=>*User*(id,name,age)  
 }.toDS()  
  
 //\*\*\*\*\*DataSet=>=>RDD\*\*\*\*\*  
 ds1.*rdd* //释放资源  
 spark.stop()  
 }  
}  
**case class** User(id:Int,name:String,age:Int)

## 用户自定义函数

### UDF

输入一行，返回一个结果。在Shell窗口中可以通过spark.udf功能用户可以自定义函数。

1. 创建DataFrame

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

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

1. 打印数据

scala> df.show

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

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

1. 注册UDF，功能为在数据前添加字符串

scala> spark.udf.register("addName",(x:String)=> "Name:"+x)

res9: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,StringType,Some(List(StringType)))

1. 创建临时表

scala> df.createOrReplaceTempView("people")

1. 应用UDF

scala> spark.sql("Select addName(name),age from people").show()

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

|UDF:addName(name)|age|

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

| Name:qiaofeng| 18|

| Name:duanyu| 19|

| Name:xuzhu| 20|

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

### UDAF

输入多行,返回一行。强类型的Dataset和弱类型的DataFrame都提供了相关的聚合函数， 如 count()，countDistinct()，avg()，max()，min()。除此之外，用户可以设定自己的自定义聚合函数。通过继承UserDefinedAggregateFunction来实现用户自定义聚合函数。

需求：实现求平均年龄

1. RDD算子方式实现

**object** Spark00\_TestAgeAvg {

**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)  
 **val** res: (Int, Int) = sc.makeRDD(*List*(("zhangsan", 20), ("lisi", 30), ("wangw", 40))).map {  
 **case** (name, age) => {  
 (age, 1)  
 }  
 }.reduce {  
 (t1, t2) => {  
 (t1.\_1 + t2.\_1, t1.\_2 + t2.\_2)  
 }  
 }  
 *println*(res.\_1/res.\_2)  
 // 关闭连接  
 sc.stop()  
 }  
}

1. 自定义累加器方式实现（减少Shuffle）提高效率（模仿LongAccumulator累加器）

**object** Spark01\_TestSer {

**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)  
  
 **var** sumAc = **new** MyAC  
 sc.register(sumAc)  
 sc.makeRDD(*List*(("zhangsan",20),("lisi",30),("wangw",40))).foreach{  
 **case** (name,age)=>{  
 sumAc.add(age)  
 }  
 }  
 *println*(sumAc.value)  
  
 // 关闭连接  
 sc.stop()  
 }  
}  
  
**class** MyAC **extends** AccumulatorV2[Int,Int]{  
 **var** *sum*:Int = 0  
 **var** *count*:Int = 0  
 **override def** isZero: Boolean = {  
 **return** *sum* ==0 && *count* == 0  
 }  
  
 **override def** copy(): AccumulatorV2[Int, Int] = {  
 **val** newMyAc = **new** MyAC  
 newMyAc.*sum* = **this**.*sum* newMyAc.*count* = **this**.*count* newMyAc  
 }  
  
 **override def** reset(): Unit = {  
 *sum* =0  
 *count* = 0  
 }  
  
 **override def** add(v: Int): Unit = {  
 *sum* += v  
 *count* += 1  
 }  
  
 **override def** merge(other: AccumulatorV2[Int, Int]): Unit = {  
 other **match** {  
 **case** o:MyAC=>{  
 *sum* += o.*sum  
 count* += o.*count* }  
 **case** \_=>  
 }  
  
 }  
  
 **override def** value: Int = *sum*/*count*}

1. 自定义聚合函数实现-弱类型（应用于SparkSQL更方便）

**object** Spark00\_TestAgeAvg {

**def** main(args: Array[String]): Unit = {  
 //创建上下文环境配置对象  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("SparkSQL01\_Demo")  
 //创建SparkSession对象  
 **val** spark: SparkSession = SparkSession.*builder*().config(conf).getOrCreate()  
 **import** spark.implicits.\_  
  
 //创建聚合函数  
 **var** myAverage = **new** MyAveragUDAF  
  
 //在spark中注册聚合函数  
 spark.udf.register("avgAge",myAverage)  
  
 //读取数据 {"username": "zhangsan","age": 20}  
 **val** df: DataFrame = spark.read.json("D:\\dev\\workspace\\spark-bak\\spark-bak-00\\input\\test.json")  
  
 //创建临时视图  
 df.createOrReplaceTempView("user")  
  
 //使用自定义函数查询  
 spark.sql("select avgAge(age) from user").show()  
 }  
}  
/\*

定义类继承UserDefinedAggregateFunction，并重写其中方法

\*/  
**class** MyAveragUDAF **extends** UserDefinedAggregateFunction {  
  
 // 聚合函数输入参数的数据类型  
 **def** inputSchema: StructType = StructType(*Array*(*StructField*("age",IntegerType)))  
  
 // 聚合函数缓冲区中值的数据类型(age,count)  
 **def** bufferSchema: StructType = {  
 StructType(*Array*(*StructField*("sum",LongType),*StructField*("count",LongType)))  
 }  
  
 // 函数返回值的数据类型  
 **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。  
 **def** deterministic: Boolean = **true** // 函数缓冲区初始化  
 **def** initialize(buffer: MutableAggregationBuffer): Unit = {  
 // 存年龄的总和  
 buffer(0) = 0L  
 // 存年龄的个数  
 buffer(1) = 0L  
 }  
  
 // 更新缓冲区中的数据  
 **def** update(buffer: MutableAggregationBuffer,input: Row): Unit = {  
 **if** (!input.isNullAt(0)) {  
 buffer(0) = buffer.getLong(0) + input.getInt(0)  
 buffer(1) = buffer.getLong(1) + 1  
 }  
 }  
  
 // 合并缓冲区  
 **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  
 **def** evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)  
}

1. 自定义聚合函数实现-强类型（应用于DataSet的DSL更方便）

**object** Spark04\_TestAgeAvg {

**def** main(args: Array[String]): Unit = {  
 //创建上下文环境配置对象  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("SparkSQL01\_Demo")  
 //创建SparkSession对象  
 **val** spark: SparkSession = SparkSession.*builder*().config(conf).getOrCreate()  
 **import** spark.implicits.\_  
  
 //读取数据 {"username": "zhangsan","age": 20}  
 **val** df: DataFrame = spark.read.json("D:\\dev\\workspace\\spark-bak\\spark-bak-00\\input\\test.json")  
  
 //封装为DataSet  
 **val** ds: Dataset[User01] = df.as[User01]  
  
 //创建聚合函数  
 **var** myAgeUdtf1 = **new** MyAveragUDAF1  
 //将聚合函数转换为查询的列  
 **val** col: TypedColumn[User01, Double] = myAgeUdtf1.toColumn  
  
 //查询  
 ds.select(col).show()  
  
 // 关闭连接  
 spark.stop()  
 }  
}  
  
//输入数据类型  
**case class** User01(username:String,age:Long)  
//缓存类型  
**case class** AgeBuffer(**var** sum:Long,**var** count:Long)  
*/\*\*  
 \* 定义类继承org.apache.spark.sql.expressions.Aggregator  
 \* 重写类中的方法  
 \*/***class** MyAveragUDAF1 **extends** Aggregator[User01,AgeBuffer,Double]{  
 **override def** zero: AgeBuffer = {  
 *AgeBuffer*(0L,0L)  
 }  
  
 **override def** reduce(b: AgeBuffer, a: User01): AgeBuffer = {  
 b.sum = b.sum + a.age  
 b.count = b.count + 1  
 b  
 }  
  
 **override def** merge(b1: AgeBuffer, b2: AgeBuffer): AgeBuffer = {  
 b1.sum = b1.sum + b2.sum  
 b1.count = b1.count + b2.count  
 b1  
 }  
  
 **override def** finish(buff: AgeBuffer): Double = {  
 buff.sum.toDouble/buff.count  
 }  
 //DataSet默认额编解码器，用于序列化，固定写法  
 //自定义类型就是produce 自带类型根据类型选择  
 **override def** bufferEncoder: Encoder[AgeBuffer] = {  
 Encoders.*product* }  
  
 **override def** outputEncoder: Encoder[Double] = {  
 Encoders.*scalaDouble* }  
}

**输出结果：**

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

|MyAveragUDAF1(com.atguigu.spark.core.day05.User01)|

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

| 18.0|

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

### UDTF

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

# SparkSQL数据的加载与保存

## 通用的加载和保存方式

* **spark.**read.load 是加载数据的通用方法
* **df.**write.save 是保存数据的通用方法

### 加载数据

1. read直接加载数据

scala> spark.read.

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。

**例如：直接加载Json数据**

scala> spark.read.json("/opt/module/spark-local/people.json").show

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

1. format指定加载数据类型

scala> spark.read.format("…")[.option("…")].load("…")

用法详解：

* 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

**例如：使用format指定加载Json类型数据**

scala> spark.read.format("json").load ("/opt/module/spark-local/people.json").show

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

1. 在文件上直接运行SQL

我们前面都是使用read API 先把文件加载到 DataFrame然后再查询，其实，我们也可以直接在文件上进行查询

scala> spark.sql("select \* from json.`/opt/module/spark-local/people.json`").show

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

|age| name|

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

| 18|qiaofeng|

| 19| duanyu|

| 20| xuzhu|

+---+--------+|

**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.

### 保存数据

1. write直接保存数据

scala> df.write.

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。

**例如：直接将df中数据保存到指定目录**

//默认保存格式为parquet

scala> df.write.save("/opt/module/spark-local/output")

//可以指定为保存格式，直接保存，不需要再调用save了

scala> df.write.json("/opt/module/spark-local/output")

1. format指定保存数据类型

scala> df.write.format("…")[.option("…")].save("…")

用法详解：

* 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

1. 文件保存选项

保存操作可以使用 SaveMode, 用来指明如何处理数据，使用mode()方法来设置。

有一点很重要: 这些 SaveMode 都是没有加锁的, 也不是原子操作。

SaveMode是一个枚举类，其中的常量包括：

|  |  |  |
| --- | --- | --- |
| Scala/Java | Any Language | Meaning |
| SaveMode.ErrorIfExists(default) | "error"(default) | 如果文件已经存在则抛出异常 |
| SaveMode.Append | "append" | 如果文件已经存在则追加 |
| SaveMode.Overwrite | "overwrite" | 如果文件已经存在则覆盖 |
| SaveMode.Ignore | "ignore" | 如果文件已经存在则忽略 |

**例如：使用指定format指定保存类型进行保存**

df.write.mode("append").json("/opt/module/spark-local/output")

### 默认数据源

Spark SQL的默认数据源为Parquet格式。数据源为Parquet文件时，Spark SQL可以方便的执行所有的操作，不需要使用format。修改配置项spark.sql.sources.default，可修改默认数据源格式。

1. 加载数据

val df = spark.read.load("/opt/module/spark-local/examples/src/main/resources/users.parquet").show

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

| name|favorite\_color|favorite\_numbers|

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

|Alyssa| null| [3, 9, 15, 20]|

| Ben| red| []|

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

df: Unit = ()

1. 保存数据

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

//保存为parquet格式

scala> df.write.mode("append").save("/opt/module/spark-local/output")

## JSON文件

Spark SQL 能够自动推测 JSON数据集的结构，并将它加载为一个Dataset[Row]. 可以通过SparkSession.read.json()去加载一个 一个JSON 文件。

注意：这个JSON文件不是一个传统的JSON文件，每一行都得是一个JSON串。格式如下：

{"name":"Michael"}

{"name":"Andy","age":30}

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

1. 导入隐式转换

import spark.implicits.\_

1. 加载JSON文件

val path = "/opt/module/spark-local/people.json"  
val peopleDF = spark.read.json(path)

1. 创建临时表

peopleDF.createOrReplaceTempView("people")

1. 数据查询

val teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")

teenagerNamesDF.show()

+------+

| name|

+------+

|Justin|

+------+

## MySQL

Spark SQL可以通过JDBC从关系型数据库中读取数据的方式创建DataFrame，通过对DataFrame一系列的计算后，还可以将数据再写回关系型数据库中。

如果使用spark-shell操作，可在启动shell时指定相关的数据库驱动路径或者将相关的数据库驱动放到spark的类路径下。

bin/spark-shell

--jars mysql-connector-java-5.1.27-bin.jar

我们这里只演示在Idea中通过JDBC对Mysql进行操作

**导入依赖**

<dependency>

<groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.27</version>  
</dependency>

### 从JDBC读数据

**object** SparkSQL02\_Datasource {

**def** main(args: Array[String]): Unit = {  
 //创建上下文环境配置对象  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("SparkSQL01\_Demo")  
  
 //创建SparkSession对象  
 **val** spark: SparkSession = SparkSession.*builder*().config(conf).getOrCreate()  
  
 **import** spark.implicits.\_  
  
 //方式1：通用的load方法读取  
 spark.read.format("jdbc")  
 .option("url", "jdbc:mysql://hadoop202:3306/test")  
 .option("driver", "com.mysql.jdbc.Driver")  
 .option("user", "root")  
 .option("password", "123456")  
 .option("dbtable", "user")  
 .load().show  
  
   
 //方式2:通用的load方法读取 参数另一种形式  
 spark.read.format("jdbc")  
 .options(*Map*("url"->"jdbc:mysql://hadoop202:3306/test?user=root&password=123456",  
 "dbtable"->"user","driver"->"com.mysql.jdbc.Driver")).load().show  
  
 //方式3:使用jdbc方法读取  
 **val** props: Properties = **new** Properties()  
 props.setProperty("user", "root")  
 props.setProperty("password", "123456")  
 **val** df: DataFrame = spark.read.jdbc("jdbc:mysql://hadoop202:3306/test", "user", props)  
 df.show  
  
 //释放资源  
 spark.stop()  
 }  
}

### 向JDBC写数据

**object** SparkSQL03\_Datasource {

**def** main(args: Array[String]): Unit = {  
 //创建上下文环境配置对象  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("SparkSQL01\_Demo")  
  
 //创建SparkSession对象  
 **val** spark: SparkSession = SparkSession.*builder*().config(conf).getOrCreate()  
 **import** spark.implicits.\_  
  
 **val** rdd: RDD[User2] = spark.sparkContext.makeRDD(*List*(*User2*("lisi", 20), *User2*("zs", 30)))  
 **val** ds: Dataset[User2] = rdd.toDS  
 //方式1：通用的方式 format指定写出类型  
 ds.write  
 .format("jdbc")  
 .option("url", "jdbc:mysql://hadoop202:3306/test")  
 .option("user", "root")  
 .option("password", "123456")  
 .option("dbtable", "user")  
 .mode(SaveMode.*Append*)  
 .save()  
  
 //方式2：通过jdbc方法  
 **val** props: Properties = **new** Properties()  
 props.setProperty("user", "root")  
 props.setProperty("password", "123456")  
 ds.write.mode(SaveMode.*Append*).jdbc("jdbc:mysql://hadoop202:3306/test", "user", props)  
  
 //释放资源  
 spark.stop()  
 }  
}  
  
**case class** User2(name: String, age: Long)

## Hive

Apache Hive 是 Hadoop 上的 SQL 引擎，Spark SQL编译时可以包含 Hive 支持，也可以不包含。

包含 Hive 支持的 Spark SQL 可以支持 Hive 表访问、UDF (用户自定义函数)以及 Hive 查询语言(HiveQL/HQL)等。需要强调的一点是，如果要在 Spark SQL 中包含Hive 的库，并不需要事先安装 Hive。一般来说，最好还是在编译Spark SQL时引入Hive支持，这样就可以使用这些特性了。如果你下载的是二进制版本的 Spark，它应该已经在编译时添加了 Hive 支持。

若要把 Spark SQL 连接到一个部署好的 Hive 上，你必须把 hive-site.xml 复制到 Spark的配置文件目录中($SPARK\_HOME/conf)。即使没有部署好 Hive，Spark SQL 也可以运行，需要注意的是，如果你没有部署好Hive，Spark SQL 会在当前的工作目录中创建出自己的 Hive 元数据仓库，叫作 metastore\_db。此外，对于使用部署好的Hive，如果你尝试使用 HiveQL 中的 CREATE TABLE (并非 CREATE EXTERNAL TABLE)语句来创建表，这些表会被放在你默认的文件系统中的 /user/hive/warehouse 目录中(如果你的 classpath 中有配好的 hdfs-site.xml，默认的文件系统就是 HDFS，否则就是本地文件系统)。

spark-shell默认是Hive支持的；代码中是默认不支持的，需要手动指定（加一个参数即可）。

### 使用内嵌Hive

如果使用 Spark 内嵌的 Hive, 则什么都不用做, 直接使用即可.

Hive 的元数据存储在 derby 中, 仓库地址:$SPARK\_HOME/spark-warehouse

scala> spark.sql("show tables").show  
+--------+---------+-----------+  
|database|tableName|isTemporary|  
+--------+---------+-----------+  
+--------+---------+-----------+  
  
  
scala> spark.sql("create table aa(id int)")  
19/02/09 18:36:10 WARN HiveMetaStore: Location: file:/opt/module/spark-local/spark-warehouse/aa specified for non-external table:aa  
res2: org.apache.spark.sql.DataFrame = []  
  
scala> spark.sql("show tables").show  
+--------+---------+-----------+  
|database|tableName|isTemporary|  
+--------+---------+-----------+  
| default| aa| false|  
+--------+---------+-----------+

向表中加载本地数据数据

scala> spark.sql("load data local inpath './ids.txt' into table aa")  
res8: org.apache.spark.sql.DataFrame = []  
  
scala> spark.sql("select \* from aa").show  
+---+  
| id|  
+---+  
|100|  
|101|  
|102|  
|103|  
|104|  
|105|  
|106|  
+---+

然而在实际使用中, 几乎没有任何人会使用内置的 Hive

### 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。

* 确定原有Hive是正常工作的
* 需要把hive-site.xml拷贝到spark的conf/目录下
* 如果以前hive-site.xml文件中，配置过Tez相关信息，注释掉
* 把Mysql的驱动copy到Spark的jars/目录下
* 需要提前启动hive服务，hive/bin/hiveservices.sh start
* 如果访问不到hdfs，则需把core-site.xml和hdfs-site.xml拷贝到conf/目录

**启动 spark-shell**

scala> spark.sql("show tables").show  
+--------+---------+-----------+  
|database|tableName|isTemporary|  
+--------+---------+-----------+  
| default| emp| false|  
+--------+---------+-----------+  
scala> spark.sql("select \* from emp").show  
19/02/09 19:40:28 WARN LazyStruct: Extra bytes detected at the end of the row! Ignoring similar problems.  
+-----+-------+---------+----+----------+------+------+------+  
|empno| ename| job| mgr| hiredate| sal| comm|deptno|  
+-----+-------+---------+----+----------+------+------+------+  
| 7369| SMITH| CLERK|7902|1980-12-17| 800.0| null| 20|  
| 7499| ALLEN| SALESMAN|7698| 1981-2-20|1600.0| 300.0| 30|  
| 7521| WARD| SALESMAN|7698| 1981-2-22|1250.0| 500.0| 30|  
| 7566| JONES| MANAGER|7839| 1981-4-2|2975.0| null| 20|  
| 7654| MARTIN| SALESMAN|7698| 1981-9-28|1250.0|1400.0| 30|  
| 7698| BLAKE| MANAGER|7839| 1981-5-1|2850.0| null| 30|  
| 7782| CLARK| MANAGER|7839| 1981-6-9|2450.0| null| 10|  
| 7788| SCOTT| ANALYST|7566| 1987-4-19|3000.0| null| 20|  
| 7839| KING|PRESIDENT|null|1981-11-17|5000.0| null| 10|  
| 7844| TURNER| SALESMAN|7698| 1981-9-8|1500.0| 0.0| 30|  
| 7876| ADAMS| CLERK|7788| 1987-5-23|1100.0| null| 20|  
| 7900| JAMES| CLERK|7698| 1981-12-3| 950.0| null| 30|  
| 7902| FORD| ANALYST|7566| 1981-12-3|3000.0| null| 20|  
| 7934| MILLER| CLERK|7782| 1982-1-23|1300.0| null| 10|  
| 7944|zhiling| CLERK|7782| 1982-1-23|1300.0| null| 50|  
+-----+-------+---------+----+----------+------+------+------+

### 运行Spark SQL CLI

Spark SQLCLI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 SQ LCLI，直接执行SQL语句，类似Hive窗口。

bin/spark-sql

### 代码中操作Hive

1. 添加依赖

<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-hive\_2.11</artifactId>  
 <version>2.1.1</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.hive</groupId>  
 <artifactId>hive-exec</artifactId>  
 <version>1.2.1</version>  
</dependency>

1. 拷贝hive-site.xml到resources目录
2. 代码实现

**object** SparkSQL08\_Hive{

**def** main(args: Array[String]): Unit = {  
 //创建上下文环境配置对象  
 **val** conf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("SparkSQL01\_Demo")  
 **val** spark: SparkSession = SparkSession  
 .*builder*()  
 .enableHiveSupport()  
 .master("local[\*]")  
 .appName("SQLTest")  
 .getOrCreate()  
 spark.sql("show tables").show()  
 //释放资源  
 spark.stop()  
 }  
}

# SparkSQL项目实战

## 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

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,  
 `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';  
load data local inpath '/opt/module/data/user\_visit\_action.txt' into table sparkpractice.user\_visit\_action;  
  
CREATE TABLE `product\_info`(  
 `product\_id` bigint,  
 `product\_name` string,  
 `extend\_info` string)  
row format delimited fields terminated by '\t';  
load data local inpath '/opt/module/data/product\_info.txt' into table sparkpractice.product\_info;  
  
CREATE TABLE `city\_info`(  
 `city\_id` bigint,  
 `city\_name` string,  
 `area` string)  
row format delimited fields terminated by '\t';  
load data local inpath '/opt/module/data/city\_info.txt' into table sparkpractice.city\_info;

## 需求：各区域热门商品Top3

### 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% |

### 思路分析

* 使用 sql 来完成，碰到复杂的需求，可以使用 udf 或 udaf
* 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
* 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
* 每个地区内按照点击次数降序排列
* 只取前三名，并把结果保存在数据库中
* 城市备注需要自定义 UDAF 函数

### 代码实现

1. udaf 函数定义

**class** AreaClickUDAF **extends** UserDefinedAggregateFunction {

// 输入数据的类型: 北京 String  
 **override def** inputSchema: StructType = {  
 *StructType*(*StructField*("city\_name", StringType) :: *Nil*)  
 // StructType(Array(StructField("city\_name", StringType)))  
 }  
  
 // 缓存的数据的类型: 北京->1000, 天津->5000 Map, 总的点击量 1000/?  
 **override def** bufferSchema: StructType = {  
 // MapType(StringType, LongType) 还需要标注 map的key的类型和value的类型  
 *StructType*(*StructField*("city\_count", *MapType*(StringType, LongType)) :: *StructField*("total\_count", LongType) :: *Nil*)  
 }  
  
 // 输出的数据类型 "北京21.2%，天津13.2%，其他65.6%" String  
 **override def** dataType: DataType = StringType  
  
 // 相同的输入是否应用有相同的输出.  
 **override def** deterministic: Boolean = **true** // 给存储数据初始化  
 **override def** initialize(buffer: MutableAggregationBuffer): Unit = {  
 //初始化map缓存  
 buffer(0) = *Map*[String, Long]()  
 // 初始化总的点击量  
 buffer(1) = 0L  
 }  
  
 // 分区内合并 Map[城市名, 点击量]  
 **override def** update(buffer: MutableAggregationBuffer, input: Row): Unit = {  
 // 首先拿到城市名, 然后把城市名作为key去查看map中是否存在, 如果存在就把对应的值 +1, 如果不存在, 则直接0+1  
 **val** cityName = input.getString(0)  
 // val map: collection.Map[String, Long] = buffer.getMap[String, Long](0)  
 **val** map: Map[String, Long] = buffer.getAs[Map[String, Long]](0)  
 buffer(0) = map + (cityName -> (map.getOrElse(cityName, 0L) + 1L))  
 // 碰到一个城市, 则总的点击量要+1  
 buffer(1) = buffer.getLong(1) + 1L  
 }  
  
 // 分区间的合并  
 **override def** merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {  
 **val** map1 = buffer1.getAs[Map[String, Long]](0)  
 **val** map2 = buffer2.getAs[Map[String, Long]](0)  
  
 // 把map1的键值对与map2中的累积, 最后赋值给buffer1  
 buffer1(0) = map1.foldLeft(map2) {  
 **case** (map, (k, v)) =>  
 map + (k -> (map.getOrElse(k, 0L) + v))  
 }  
  
 buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)  
 }  
  
 // 最终的输出. "北京21.2%，天津13.2%，其他65.6%"  
 **override def** evaluate(buffer: Row): Any = {  
 **val** cityCountMap = buffer.getAs[Map[String, Long]](0)  
 **val** totalCount = buffer.getLong(1)  
  
 **var** citysRatio: List[CityRemark] = cityCountMap.toList.sortBy(-\_.\_2).take(2).map {  
 **case** (cityName, count) => {  
 *CityRemark*(cityName, count.toDouble / totalCount)  
 }  
 }  
 // 如果城市的个数超过2才显示其他  
 **if** (cityCountMap.size > 2) {  
 citysRatio = citysRatio :+ *CityRemark*("其他", citysRatio.foldLeft(1D)(\_ - \_.cityRatio))  
 }  
 citysRatio.mkString(", ")  
 }  
}  
  
  
**case class** CityRemark(cityName: String, cityRatio: Double) {  
 **val** *formatter* = **new** DecimalFormat("0.00%")  
 **override def** toString: String = s"**$**cityName:**$**{*formatter*.format(cityRatio)}"  
}

1. 具体实现

**object** SparkSQL04\_TopN {

**def** main(args: Array[String]): Unit = {  
 **val** spark: SparkSession = SparkSession  
 .*builder*()  
 .master("local[2]")  
 .appName("AreaClickApp")  
 .enableHiveSupport()  
 .getOrCreate()  
 spark.sql("use sparkpractice")  
 // 0 注册自定义聚合函数  
 spark.udf.register("city\_remark", **new** AreaClickUDAF)  
 // 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  
 | c.\*,  
 | v.click\_product\_id,  
 | p.product\_name  
 |from user\_visit\_action v join city\_info c join product\_info p on v.city\_id=c.city\_id and 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  
  
 //释放资源  
 spark.stop()  
  
 }  
}