# Spark SQL

sparkSQL官方文档<http://spark.apache.org/docs/latest/sql-programming-guide.html>

Spark sql是对spark的扩展、优化，让我们增添了一种新的使用spark进行数据处理的方式

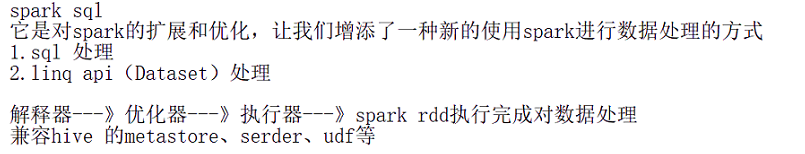
1. sql处理
2. Linq api(Dataset)处理

Shark(hive on spark)

虽然利用了spark进行执行，但是对于hive的依赖性比较大

查看hive中sql的执行计划

Explain select \* from tableName



编程模型

1. sparkConf---->sparkContext------>RDD---->RDD调用transformation和action
2. sparkSession---->DataSet/DataFrame-----> 直接调用sql或者直接调用DS/DF的transformation和action linq模式获取数据方法
3. SparkConf---->StreamingContext----->DStream------>对Dstream调用transformation和action 以及计算窗口函数

## Sparksql的文件格式

SparkSql DateSet保存成text文件的时候DataSet中只能有1个字段，读取进来的时候也是只能读取一个

保存的时候不管ds字段叫什么名称，读取进来的时候名称都是value，都是字符串类型

### csv

Csv格式，只保留字段数量和字段之间的分隔，字段名称没有保存，字段类型没有保存

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** saveCsv() = {  **val** ds = *spark*.read.text(**"/orderdata/orders"**)  ds.printSchema()  **val** orderSplit = ds.map(x => {  **val** info = x.getString(0).split(**"\\|"**)  (info(0), info(1), info(2), info(3))  })  orderSplit.printSchema()  orderSplit.createOrReplaceTempView(**"order"**)  *// 获取订单状态为COMPLET的订单信息* **val** result = sql(**"select \* from order where \_4 = 'CANCELED'"**)  result.show()  *// 根据核心数，默认启动2个，每个输出一个* result.coalesce(1)  result.write.csv(**"/spark/ordercsv"**) }  **def** readCsv() = {  **val** ds = *spark*.read.csv(**"/spark/ordercsv"**)  ds.printSchema()  *// ds.show()* ds.createOrReplaceTempView(**"orders"**)  **val** result = sql(  **"""  |select \_c0  | ,\_c3  |from orders  """**.stripMargin).show() } |

### Parquet

Parquet，保存字段数量和分隔，保存字段名称，字段类型

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** writerParquet() = {  **val** ds = *spark*.read.text(**"/orderdata/orders"**)  ds.printSchema()  **val** orderSplit = ds.map(x => {  **val** info = x.getString(0).split(**"\\|"**)  (info(0), info(1), info(2), info(3), 1)  })  orderSplit.printSchema()  orderSplit.write.parquet(**"/spark/orderparquet"**) }  **def** readParquet() = {  **val** ds = *spark*.read.parquet(**"/spark/orderparquet"**)  ds.printSchema() } |

### orc

Orc,保存字段数量和分隔，保存字段名称，保存字段类型，文件是二进制

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** writeOrc() = {  **val** ds = *spark*.read.text(**"/orderdata/orders"**)  ds.printSchema()  **val** orderSplit = ds.map(x => {  **val** info = x.getString(0).split(**"\\|"**)  (info(0), info(1), info(2), info(3), 1)  })  orderSplit.printSchema()  orderSplit.write.orc(**"/spark/orderorc"**) }  **def** readOrc() = {  **val** ds = *spark*.read.orc(**"/spark/orderorc"**)  ds.printSchema() } |

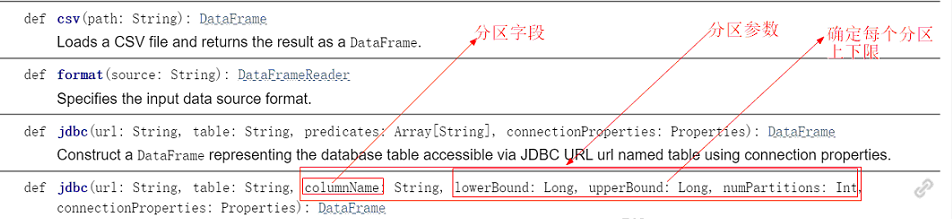
### Json

Json，保存字段数量和分隔，保存字段名称，保存字段类型，文件是文本类型，但是json字段类型不丰富

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** writeJson() = {  **val** ds = *spark*.read.text(**"/orderdata/orders"**)  ds.printSchema()  **val** orderSplit = ds.map(x => {  **val** info = x.getString(0).split(**"\\|"**)  (info(0), info(1), info(2), info(3), 1)  })  orderSplit.printSchema()  orderSplit.write.json(**"/spark/orderjson"**) }  **def** readJson() = {  **val** ds = *spark*.read.json(**"/spark/orderjson"**)  ds.printSchema() } |

### Jdbc

Jdbc。从关系型数据库中读取或保存数据，会存储数据的字段名称，字段类型信息等信息完整保存



|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** readJdbc() = {  **val** url = **"jdbc:mysql://localhost:3306/xs"  val** prop = **new** Properties()  prop.put(**"user"**, **"root"**)  prop.put(**"password"**, **"root"**)  **val** ds = *spark*.read.jdbc(url, **"t\_user"**, prop)  ds.printSchema()  ds.show() }  **def** writeJdbc() = {  **val** ds = *spark*.read.text(**"/orderdata/orders"**)  ds.printSchema()  **val** orderSplit = ds.map(x => {  **val** info = x.getString(0).split(**"\\|"**)  (info(0), info(1), info(2), info(3))  })  orderSplit.printSchema()  orderSplit.createOrReplaceTempView(**"order"**)  *// 获取订单状态为COMPLET的订单信息* **val** complete = sql(**"select \* from order where \_4 = 'CANCELED'"**)  *// complete.show()* **val** url = **"jdbc:mysql://localhost:3306/xs"  val** prop = **new** Properties()  prop.put(**"user"**, **"root"**)  prop.put(**"password"**, **"root"**)  complete.write.jdbc(url, **"complete\_order"**, prop) } |

## Rdd与ds、df之间的转换

Rdd加载文件时，文件的每一行构成rdd的一个元素

DataSet/DataFrame 加载文件时，文件的每一行构成DataSet/DataFrame的一条有模式的记录

使用sql的条件：模式 + 元组

Rdd--加模式----》DataSet/DataFrame 例如:rdd.toDF() rdd.toDF() spark.createDataSet() spark.createDataFrame()

DataSet/DataFrame ----加模式--》RDD

ds.rdd df.rdd

## Dataset的保存文件的模式

SaveMode:Overwrite: overwrite the existing data.

SaveMode:Append: append the data.

SaveMode:ignore: ignore the operation

SaveMode:ErrorIfExists: default option,throw an exception at runtime

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** saveMode() = {  **val** ds = *spark*.read.text(**"/orderdata/orders"**)  ds.printSchema()  **val** orderSplit = ds.map(x => {  **val** info = x.getString(0).split(**"\\|"**)  (info(0), info(1), info(2), info(3),1)  })  orderSplit.createOrReplaceTempView(**"order"**)  **val** complete = sql(**"select \* from order where \_4 = 'CANCELED'"**)  *//mode(SaveMode.Append)枚举类型设置存储方式* complete.write.mode(SaveMode.*Append*).parquet(**"/spark/modeParquet"**) } |

使用模式的方法：

使用sparksession的read方法加载数据文件时，结果中(DataSet/DataFrame)从文件中加载模式

### 自动添加

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  *// 自动加载：1. 从文件中自动加载模式 // 2. 从数据中自动加载模式* **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"添加模式"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** addSchemaFromData() = {  **val** list = *List*(1, 2, 3, 4, 5)  **val** rdd = *spark*.sparkContext.parallelize(list)  **val** ds1 = rdd.toDS()  ds1.printSchema()  **val** ds2 = *spark*.createDataset(list)  ds2.printSchema()   **val** list2 = *List*((1, 2, 3), (4, 5, 6), (7, 8, 9))  **val** ds3 = *spark*.createDataset(list2)  ds3.printSchema() } |

### 通过DataFrame添加模式

* 1. 数据元素用Row类型来封装
  2. 模式用StructType来描述表结构，用StructField来描述字段(名称，类型，是否为空)
  3. 使用DataType的子类来描述字段的类型
  4. 使用

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"sparksql存取文件"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  *// 手动给元组添加模式* **def** addSchemaByFrame() = {  **val** list = *List*(1, 2, 3, 4, 5)  **val** structType = *StructType*(*StructField*(**"column1"**, IntegerType, **true**) :: *Nil*)  **import** scala.collection.JavaConversions.\_  **val** ds1 = *spark*.createDataFrame(list.map(x => *Row*(x)), structType)   ds1.printSchema()  **val** students = *List*(*Row*(**"小张"**, **"女"**, 19), *Row*(**"小李"**, **"女"**, 17), *Row*(**"小王"**, **"男"**, 18)) *// 方式一*  **val** studentSchema = *StructType*(*StructField*(**"name"**,StringType,**true**)::  *StructField*(**"gender"**,StringType,**true**)::  *StructField*(**"age"**,IntegerType,**true**)::  *Nil*)  */\*// 方式二  val studentSchema = (new StructType())  .add(StructField("name", StringType, true))  .add("gender", StringType, true)  .add(StructField("age", IntegerType, true)) \*/*  **val** ds2 = *spark*.createDataFrame(students,studentSchema)  ds2.printSchema()  ds2.createOrReplaceTempView(**"student"**)  sql(**"select \* from student where age > 18"**).show() } |

### Case class添加模式

在数据自动加载模式的时候，如果每个元素的数据类型是一个case class，那么sparksql会自动把case class的字段模式添加到dataset数据集上，即每个case class的属性名称将会作为模式的字段名称，属性的类型将会作为模式的字段类型

Caseclass如果定义在Object里面，则相当于在object里面定义一个静态的内部类，类似于java中的public static class

如果定义在class里面，相当于定义一个内部类 public class

|  |
| --- |
| **case class** Student(name: String, sex: String, age: Int)  *// cass class 的方式给数据添加模式* **def** caseClassSchema() = {  **val** list = *List*(*Student*(**"小张"**, **"男"**, 18), *Student*(**"小王"**, **"女"**, 17), *Student*(**"小李"**, **"男"**, 19))  **val** ds1 = *spark*.createDataset(list)  ds1.printSchema()  ds1.createOrReplaceTempView(**"student"**)  sql(**"select \* from student where age > 18"**).show() } |

## Dataset和DataFrame

dataFrame 是一种更加具体的dataset

Dataset中的每个元素可以是任意的类型

而dataFrame中的元素必须是Row的对象，Row在声明的时候初始化方法和Tuple一样，一个Row里面可以放多个数据，位置是有序的，多个数据之间是不同的数据类型，在取值的时候row和array、list类似，使用小括号以位置标识作为参数来取值

|  |
| --- |
| **val** row = Row(1,**"a"**,**true**,3.14) row (0) row (1) |

### DataSet转换成DataFrame

DataSet 可以通过调用toDF方法转换成DataFrame

两个类型中都保存有模式

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"添加模式"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** caseClassSchemaRdd() = {  **val** rdd = *spark*.sparkContext.textFile(**"/orderdata/orders"**)  **val** cRdd = rdd.map(x => {  **val** info = x.split(**"\\|"**)  *Order*(info(0), info(1), info(2), info(3))  })  **val** ds = cRdd.toDS()  ds.printSchema()  ds.createOrReplaceTempView(**"order"**)  *// 统计每个客户的订单数* sql(  **"""  |select customerId  | ,count(1)  |from order  |group by customerId  """**.stripMargin).show()   *// dataFrame每个元素都是row对象* **val** df = ds.toDF()  df.printSchema()  ds.printSchema()  df.take(10).foreach(*println*)  ds.take(10).foreach(*println*) } |

### 获取ds、df的模式

要获取ds或df模式，使用schema方法，获取的对象是structty

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"添加模式"**).getOrCreate()  **import** *spark*.implicits.\_ **import** *spark*.sql  **def** dsSchemaType() = {  **val** list = *List*(1, 2, 3, 4, 5)  **val** structType = *StructType*(*StructField*(**"column1"**, IntegerType, **true**) :: *Nil*)  **import** scala.collection.JavaConversions.\_  **val** ds1 = *spark*.createDataFrame(list.map(x => *Row*(x)), structType)   **val** sc = ds1.schema.printTreeString()  ds1.printSchema() } |

## udf自定义函数

SparkSession.udf ====> UDFRegistration

Spark.udf.register(“函数名称”,函数字面量)

在接下来的spark环境下就可以直接调用sql中 “函数名称”来使用函数了

|  |
| --- |
| **val** *spark* = SparkSession.*builder*().master(**"local[\*]"**).appName(**"用户注册"**).getOrCreate() **import** *spark*.implicits.\_ **import** *spark*.sql **case class** Order(orderId: String, orderData: String, customerId: String, status: String)  **def** udfTest() = {  **val** rdd = *spark*.sparkContext.textFile(**"/orderdata/orders"**)  **val** ds = rdd.map(x => {  **val** info = x.split(**"\\|"**)  *Order*(info(0),info(1),info(2),info(3))  }).toDS()  *// 自定义一个函数把2013-07-25 00:00:00转换成日期格式2013-07-25  spark*.udf.register(**"my\_dt\_to\_date"**,(dateTime:String) => {  **val** sformat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**)  **val** tformat = **new** SimpleDateFormat(**"yyyy-MM-dd"**)  **val** date = sformat.parse(dateTime)  tformat.format(date)  })   ds.createOrReplaceTempView(**"order"**)  **val** udfResult = sql(  **"""  |select orderId  | ,my\_dt\_to\_date(orderData)  | ,customerId  | ,status  |from order  """**.stripMargin)  udfResult.printSchema()  udfResult.show() } |

## Spark读写hbase

### 读Hbase

使用sparkContext的newHadoopAPIDataSet，从Hbase中加载数据，生成RDD

RDD[(ImmutableWritable,Result)]

调用RDD的map方法或flatMap方法。把每一行数据转换成class case类型

调用RDD的toDs()方法，获取DataSet

### 写Hbase

调用DataSet或者DataFrame的rdd方法将其转换成rdd

调用rdd的saveNewHadoopApiDataSet方法，将要保存的数据封装到一个Put对象中，然后在rdd的value上，保存到hbase中去

### 作业

用户ip次数

从hbase中把user\_ip\_times中获取出来。用sparksql统计出每个ip长度访问次数，把结果保存到hbase。（使用flatMap进行转换）

## Spark sql操作Hive

构建sparksession的时候调用其builder的enableHiveSupport方法支持hive

1. 创建scala项目

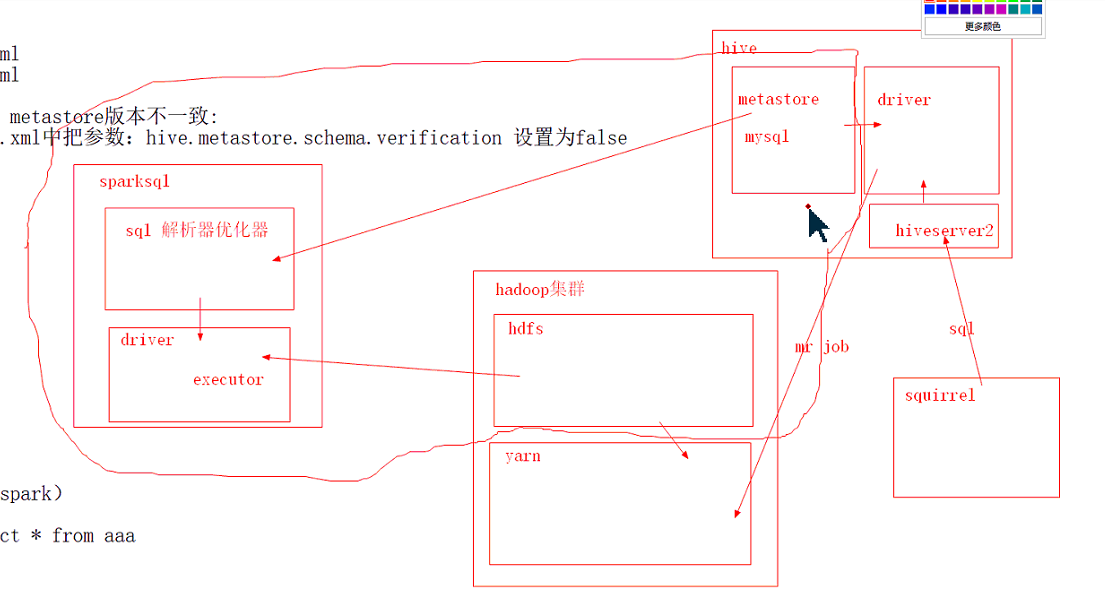
2. 添加依赖

|  |
| --- |
| *<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-core\_2.11</**artifactId**>  <**version**>2.2.0</**version**> </**dependency**> *<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-sql -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql\_2.11</**artifactId**>  <**version**>2.2.0</**version**> </**dependency**> *<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.38</**version**> </**dependency**> *<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-hive -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-hive\_2.11</**artifactId**>  <**version**>2.2.0</**version**> </**dependency**> |

3. 添加core-site.xml和hive.site.xml文件

如果出现如下错误，修改hive-site.xml文件

|  |
| --- |
| <**property**>  <**name**>hive.metastore.schema.verification</**name**>  <**value**>false</**value**> </**property**> |



spark的数据仓库目录位置

Spark.sql.warehourse.dir --- /user/hive/warehouse

Hive的数据仓库目录位置

Hive.metastore.warehourse.dir

Sparksql是支持hive而不是依赖hive

如果不和hive集成，在insert数据到sparksql的数据仓库中的话，需要指定dpark.sql.warehourse.dir为某个hdfs上的路径

如果和hive集成，则spark.sql.warehouse.dir如果不填就默认是Hive.metastore.warehourse.dir配置的目录。保持文件存储位置就在hive的仓库目录下