目录

[Spark SQL简介： 4](#_Toc487441729)

[Datasets和DataFrames简介： 4](#_Toc487441730)

[开始学习Spark SQL： 4](#_Toc487441731)

[SparkSession 4](#_Toc487441732)

[创建DataFrames: 4](#_Toc487441733)

[对无类型Dataset的操作（aka DataFrame Operation）： 5](#_Toc487441734)

[编程的方式执行SQL查询： 7](#_Toc487441735)

[全局临时视图（Global Temporary View）： 8](#_Toc487441736)

[创建Datasets 9](#_Toc487441737)

[Datasets与RDD互操作 10](#_Toc487441738)

[1.通过反射来推断schema 10](#_Toc487441739)

[2.通过编程指定schema 12](#_Toc487441740)

[聚合 13](#_Toc487441741)

[1.无类型的用户自定义聚合函数 13](#_Toc487441742)

[2.强类型的用户自定义函数 16](#_Toc487441743)

[RDD,DataFrame与DataSet区别（来自网络） 19](#_Toc487441744)

[1.RDD与DataFrame的区别 19](#_Toc487441745)

[2.提升执行效率 20](#_Toc487441746)

[3.减少数据读取 20](#_Toc487441747)

[4.执行优化 21](#_Toc487441748)

[5.RDD和DataSet 21](#_Toc487441749)

[6.DataFrame和DataSet 21](#_Toc487441750)

[数据源 23](#_Toc487441751)

[通用加载/保存函数 23](#_Toc487441752)

[手动指定选项 23](#_Toc487441753)

[直接对文件使用SQL 23](#_Toc487441754)

[保存模式 23](#_Toc487441755)

[保存到持久化表 24](#_Toc487441756)

[Parquet文件 24](#_Toc487441757)

[编程方式加载数据 24](#_Toc487441758)

[分区发现 25](#_Toc487441759)

[Schema合并 27](#_Toc487441760)

[Hive metastore Parquet table转换 28](#_Toc487441761)

[Hive/Parquet schema调和 28](#_Toc487441762)

[刷新元数据 28](#_Toc487441763)

[配置 29](#_Toc487441764)

[JSON数据集 30](#_Toc487441765)

[Hive表 31](#_Toc487441766)

[和不同版本的Hive Metastore交互 34](#_Toc487441767)

[用JDBC连接其他数据库 35](#_Toc487441768)

[疑难解答 37](#_Toc487441769)

[性能调整 37](#_Toc487441770)

[内存缓存 37](#_Toc487441771)

[其他配置选项 37](#_Toc487441772)

[分布式SQL引擎 38](#_Toc487441773)

[运行Thrift JDBC/ODBC server 38](#_Toc487441774)

[使用Spark SQL命令行工具 40](#_Toc487441775)

[升级指南 40](#_Toc487441776)

[Shark用户迁移指南 44](#_Toc487441777)

[兼容Apache Hive 45](#_Toc487441778)

[参考 48](#_Toc487441779)

### Spark SQL简介：

Spark SQL是Spark中处理结构化数据的模块。与基础的Spark RDD API不同，Spark SQL的接口提供了更多关于数据的结构信息和计算任务的运行时信息。在Spark内部，Spark SQL会能够用于做优化的信息比RDD API更多一些。Spark SQL如今有了三种不同的API：SQL语句、DataFrame API和最新的Dataset API。不过真正运行计算的时候，无论你使用哪种API或语言，Spark SQL使用的执行引擎都是同一个。这种底层的统一，使开发者可以在不同的API之间来回切换，你可以选择一种最自然的方式，来表达你的需求。

### Datasets和DataFrames简介：

Dataset是Spark-1.6新增的一种API，目前还是实验性的。Dataset想要把RDD的优势（强类型，可以使用lambda表达式函数）和Spark SQL的优化执行引擎的优势结合到一起。Dataset可以由JVM对象构建（[constructed](http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-datasets) ）得到，而后Dataset上可以使用各种transformation算子（map，flatMap，filter 等）。Dataset API 对 [Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset) 和 [Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html)的支持接口是一致的，但目前还不支持Python，不过Python自身就有语言动态特性优势（例如，你可以使用字段名来访问数据，row.columnName）。对Python的完整支持在未来的版本会增加进来。

DataFrame是按有名字的列的形式组织的。从概念上说，它相当于关系数据库中的一个表，或者是R／Python中的DataFrame，但是在执行引擎下有更丰富的优化。Dataframe的构建可以从广泛的来源，如：结构化数据文件，Hive表，外部数据库，或存在的RDDs。DataFrame API在Scala中是可用的，java、python的API是可用的，和R在Scala和java，一个DataFrame是由数据行表示。在Scala API，DataFrame简直就是一种Dataset[Row]别名。同时，在java的API，用户需要使用Dataset[Row]来表示一个DataFrame。

### 开始学习Spark SQL：

### SparkSession

spark中所有函数的入口都是SparkSession，通过Sparksession.builder()来创建，具体如下：

import org.apache.spark.sql.SparkSession

val spark =SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option","some-value")

.getOrCreate()

//用于包含RDD到DataFrame隐式转换操作

import spark.implicits.\_

### 创建DataFrames:

有了SparkSession，应用可以通过现有的RDD、hive表、Spark数据源来创建DataFrames。下面通过外部数据源来创建DataFrame:

**val** df **=** spark.read.json("examples/src/main/resources/people.json")

*// 显示dataFrame内容*

df.show()

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

### 对无类型Dataset的操作（DataFrame Operation）：

DataFrame是DataSet中的行，所以对DataFrame的“无类型”操作等价Scala/Java Datasets中“有类型”的操作。下面是通过Datasets来操作结构化数据的一些例子：

*//这个导入需要使用$标记*

**import spark.implicits.\_**

*// 以树的形式打印Schema*

df.printSchema()

*// root*

*// |-- age: long (nullable = true)*

*// |-- name: string (nullable = true)*

*// 仅查询name列*

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

*// +-------+*

*// | name|*

*// +-------+*

*// |Michael|*

*// | Andy|*

*// | Justin|*

*// +-------+*

*//查询每个人，年龄+1*

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

*// +-------+---------+*

*// | name|(age + 1)|*

*// +-------+---------+*

*// |Michael| null|*

*// | Andy| 31|*

*// | Justin| 20|*

*// +-------+---------+*

*//查询年龄大于21的*

df.filter($"age">21).show()

*// +---+----+*

*// |age|name|*

*// +---+----+*

*// | 30|Andy|*

*// +---+----+*

*// 根据年龄统计*

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

*// +----+-----+*

*// | age|count|*

*// +----+-----+*

*// | 19| 1|*

*// |null| 1|*

*// | 30| 1|*

*// +----+-----+*

### 编程的方式执行SQL查询：

SparkSession中的SQL函数使得应用能够通过编程的方式来执行SQL查询并且返回DataFrame。如下：

*//将DataFrame注册为SQL临时视图*

df.createOrReplaceTempView("people")

df.show()

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

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

sqlDF.show()

*// | 19| Justin|*

*// +----+-------+*

### 全局临时视图（Global Temporary View）：

SparkSQL中的临时视图是session级别的，一个session结束或者中断后就会消失。如果想要创建一个临时视图被所有的session所共享，并且在spark应用程序关闭前一直存活，此时就需要创建全局临时视图。全局临时视图与系统保存的global\_temp数据库相绑定，并且必须通过有资格的名字来引用它。例如SELECT \* FROM global\_temp.view1.

*//将DataFrame 注册为全局临时视图*

df.createGlobalTempView("people")

*//全局临时视图与系统保留的数据库‘global\_temp’绑定*

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

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

*//全局临时视图是跨session的*

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

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

### 创建Datasets

DataSet结合了RDD和DataFrame的优点, 并带来的一个新的概念Encoder

Datasets与RDDs相似，不同的是其采用特殊的Encoder来序列化对象。虽然encoder和标准的序列化都是将对象转化为字节数组，但是encoder是动态生成字节并且采用了一种允许spark不需要对字节数据反序列化成对象就能进行诸如filtering、sorting、hashing等操作的机制。

*//Case classes 在Scala2.10中最多支持22个字段，如果要突破此限制可以使用custom classes 来实现Product interface*

**case class Person**(name**:**String,age**:**Long)

*为case classes 创建的编码器*

**val** caseClassDS**=Seq**(**Person**("Andy",32)).toDS()

caseClassDS.show()

*// +----+---+*

*// |name|age|*

*// +----+---+*

*// |Andy| 32|*

*// +----+---+*

*// 对普通类型数据的Encoder是由 importing spark.implicits.\_ 自动提供的*

**val** primitiveDS**=Seq**(1,2,3).toDS()

primitiveDS.map(**\_**+1).collect()*// Returns: Array(2, 3, 4)*

*//DataFrame 可以转化为Dataset 通过提供一个类，映射会通过名字完成*

**val** path**=**"examples/src/main/resources/people.json"

**val** peopleDS**=**spark.read.json(path).as[Person]

peopleDS.show()

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

### 与RDD互操作

Spark SQL 支持两种将现有的RDDs转化为Datasets的方法。第一种方法通过反射来推断一个包含特殊类型对象的RDD的schema。这种基于反射的方法使得代码更简洁。

用于创建Datasets的另一种方法是通过编程接口来构建一个schema，然后把它应用到现有的RDD。虽然此方法更为详细，但它允许在运行前不知道列及其类型时构建数据集。

### 1.通过反射来推断schema

scala 的Spark SQL接口支持自动地将一个包含case class的RDD转化为DataFrame。case class定义了表的schema，case class的参数通过反射来读取并变为表的列名。Case classes也可以嵌套或包含复杂类型，如Seqs或Arrays。此RDD可以隐式（implicit）转换为DataFrame，然后注册为表。

**import org.apache.spark.sql.catalyst.encoders.ExpressionEncoder**

**import org.apache.spark.sql.Encoder**

*//为了隐式的将RDDs转化为DataFrame，而且要在方法中导入，在类外面导入可能出错*

**imports park.implicits.\_**

*//从一个文本文件创建一个person对象的RDD，将其转化为DataFrame*

**val** peopleDF**=**spark.sparkContext

.textFile("examples/src/main/resources/people.txt")

.map(**\_**.split(","))

.map(attributes**=>Person**(attributes(0),attributes(1).trim.toInt))

.toDF()

*// 将DaraFrame注册为临时视图*

peopleDF.createOrReplaceTempView("people")

//*使用Spark提供的sql方法来运行SQL语句*

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

*// 通过属性索引来获取结果中每一行某列的数据*

teenagersDF.map(teenager**=>**"Name: "+teenager(0)).show()

*// +------------+*

*// | value|*

*// +------------+*

*// |Name: Justin|*

*// +------------+*

*// 或者通过属性名来获取*

teenagersDF.map(teenager**=>**"Name: "+teenager.getAs[String]("name")).show()

*// +------------+*

*// | value|*

*// +------------+*

*// |Name: Justin|*

*// +------------+*

*//* Dataset [Map [K，V]]没有预定义的编码器，需要明确定义

**implicit val** mapEncoder**=**org.apache.spark.sql.**Encoders**.kryo[Map[String, Any]]

*//原始类型和case classes 也可以采用如下定义：*

*// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()*

// *row.getValuesMap[T]可以在一个Map[String,T]中同时检索多个列*

teenagersDF.map(teenager**=>**teenager.getValuesMap[Any](**List**("name","age"))).collect()

*// Array(Map("name" -> "Justin", "age" -> 19))*

### 2.通过编程指定schema

当案例类不能提前定义时（例如，记录的结构被编码在字符串中，或者文本数据集被解析，而对于不同的用户而言，不同的字段投影给不同的用）可以通过三个步骤以编程方式创建一个DataFrame。

1. 从原始RDD创建行的RDD
2. 通过步骤1中StructType和行构造匹配的RDD表示Schema
3. 通过SparkSession提供的createDataframe方法将schema应用于Rows的RDD

例如：

**import org.apache.spark.sql.types.\_**

*// 创建 RDD*

**val** peopleRDD**=**spark.sparkContext.textFile("examples/src/main/resources/people.txt")

*// schema编码为字符串*

**val** schemaString**=**"name age"

*//根据schema的字符串生成Schema*

**val** fields**=**schemaString.split(" ")

.map(fieldName**=>StructField**(fieldName,**StringType**,nullable**=true**))

**//** *StructField(列, 数据类型, 是否允许空)等价于表的结构*

**val** schema**=StructType**(fields)

*//将记录的RDD（people）转化为Rows*

**val** rowRDD**=**peopleRDD

.map(**\_**.split(","))

.map(attributes**=>Row**(attributes(0),attributes(1).trim))

*//将schema应用到RDD*

**val** peopleDF**=**spark.createDataFrame(rowRDD,schema)

*//通过DataFrame来创建临时视图*

peopleDF.createOrReplaceTempView("people")

*// SQL can be run over a temporary view created using DataFrames*

*//SQL 可以直接在DataFrame创建的临时视图上执行*

**val** results**=**spark.sql("SELECT name FROM people")

*//SQL 查询得到的结果是DataFrame，支持所有普通的RDD操作*

*//结果中的一行中的某个列可以通过列索引或者字段名来访问*

results.map(attributes**=>**"Name: "+attributes(0)).show()

*// +-------------+*

*// | value|*

*// +-------------+*

*// |Name: Michael|*

*// | Name: Andy|*

*// | Name: Justin|*

*// +-------------+*

### 聚合

内置的DataFrame函数提供了常见的聚合函数如count()，countdistinct()，avg()，max()，min()等。虽然这些函数为DataFrame设计，Spark SQL在Scala和Java中还有类型安全的版本用于工作在强类型的数据集上。同时，用户不会受到预定义聚合函数的限制并可以自定义聚合函数。

### 1.无类型的用户自定义聚合函数

用户必须继承[UserDefinedAggregateFunction](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.expressions.UserDefinedAggregateFunction) 这个抽象类来实现一个典型的无类型用户自定义聚合函数。例如，用户自定义的无类型聚合函数求平均值如下所示：

**import org.apache.spark.sql.expressions.MutableAggregationBuffer**

**import org.apache.spark.sql.expressions.UserDefinedAggregateFunction**

**import org.apache.spark.sql.types.\_**

**import org.apache.spark.sql.Row**

**import org.apache.spark.sql.SparkSession**

**object MyAverage extends UserDefinedAggregateFunction**{

*//聚合函数输入参数的数据类型*

**def** inputSchema**:**StructType=**StructType**(**StructField**("inputColumn",**LongType**)::**Nil**)

*//聚合缓存的数据类型*

**def** bufferSchema**:**StructType={

**StructType**(**StructField**("sum",**LongType**)::**StructField**("count",**LongType**)::**Nil**)

}

*//返回值的数据类型*

**def** dataType**:**DataType=**DoubleType**

*//对于同样的输入这个函数是否总是有相同的输出*

**def** deterministic**:**Boolean=**true**

*//初始化给定的聚合缓冲区。缓冲区本身是一个“行”，除了标准的方法，例如通过索引检索一个值（例如，get()，getboolean()），还提供了对values的更新。注意，缓冲区内的arrays和map仍然是不可改变的。*

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

buffer(0)**=**0L

buffer(1)**=**0L

}

*//根据输入数据更新给定的聚合缓冲的缓冲区*

**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

}

}

*//合并两个聚合缓冲区并将更新后的缓冲区值保存到‘buffer1’*

**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)

}

*// 注册需要访问的函数*

spark.udf.register("myAverage",**MyAverage**)

**val** df**=**spark.read.json("examples/src/main/resources/employees.json")

df.createOrReplaceTempView("employees")

df.show()

*// +-------+------+*

*// | name|salary|*

*// +-------+------+*

*// |Michael| 3000|*

*// | Andy| 4500|*

*// | Justin| 3500|*

*// | Berta| 4000|*

*// +-------+------+*

**val** result**=**spark.sql("SELECT myAverage(salary) as average\_salary FROM employees")

result.show()

*// +--------------+*

*// |average\_salary|*

*// +--------------+*

*// | 3750.0|*

*// +--------------+*

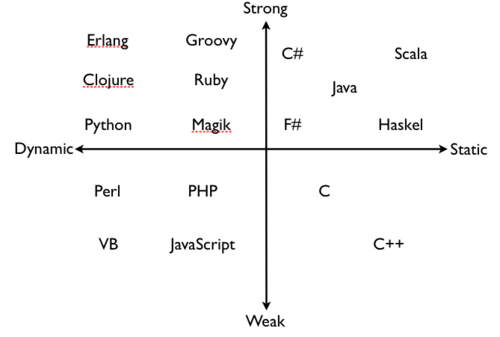
### 2.强类型的用户自定义函数

 动态类型语言：在运行期进行类型检查的语言，也就是在编写代码的时候可以不指定变量的数据类型，比如Python和Ruby

 静态类型语言：它的数据类型是在编译期进行检查的，也就是说变量在使用前要声明变量的数据类型，这样的好处是把类型检查放在编译期，提前检查可能出现的类型错误，典型代表C/C++和Java

 强类型语言，一个变量不经过强制转换，它永远是这个数据类型，不允许隐式的类型转换。举个例子：如果你定义了一个double类型变量a,不经过强制类型转换那么程序int b = a无法通过编译。典型代表是Java。

 弱类型语言：它与强类型语言定义相反,允许编译器进行隐式的类型转换，典型代表C/C++。



用户自定义的针对强类型数据集的聚合函数需要继承Aggregator抽象类，例如一个用户自定义的类型安全的聚合函数求平均值如下：

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

**import org.apache.spark.sql.Encoder**

**import org.apache.spark.sql.Encoders**

**import org.apache.spark.sql.SparkSession**

**caseclass Employee**(name**:**String,salary**:**Long)

**caseclass Average**(**var** sum**:**Long,**var** count**:**Long)

**object MyAverage extends Aggregator**[Employee, Average, Double]{

*//对于聚合函数的一个零值，需要满足b+0=b的条件*

**def** zero**:** Average=**Average**(0L,0L)

*//合并两个值生成一个新值。为了性能，函数将会修改‘buffer’并且返回而不是创建一个新的对象*

**def** reduce(buffer**:**Average,employee**:**Employee)**:**Average={

buffer.sum+=employee.salary

buffer.count+=1

buffer

}

*//合并另个中间值*

**def** merge(b1**:**Average,b2**:**Average)**:**Average={

b1.sum+=b2.sum

b1.count+=b2.count

b1

}

*//对reduction的输出结果进行转化*

**def** finish(reduction**:**Average)**:**Double=reduction.sum.toDouble/reduction.count

*//为中间值提供的特殊编码器*

**def** bufferEncoder**:**Encoder[Average]**=Encoders**.product

*//最终输出值类型的编码器*

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

}

**val** ds**=**spark.read.json("examples/src/main/resources/employees.json").as[Employee]

ds.show()

*// +-------+------+*

*// | name|salary|*

*// +-------+------+*

*// |Michael| 3000|*

*// | Andy| 4500|*

*// | Justin| 3500|*

*// | Berta| 4000|*

*// +-------+------+*

*//将函数转化为有类型的列，并给它一个名字*

**val** averageSalary**=MyAverage**.toColumn.name("average\_salary")

**val** result**=**ds.select(averageSalary)

result.show()

*// +--------------+*

*// |average\_salary|*

*// +--------------+*

*// | 3750.0|*

*// +--------------+*

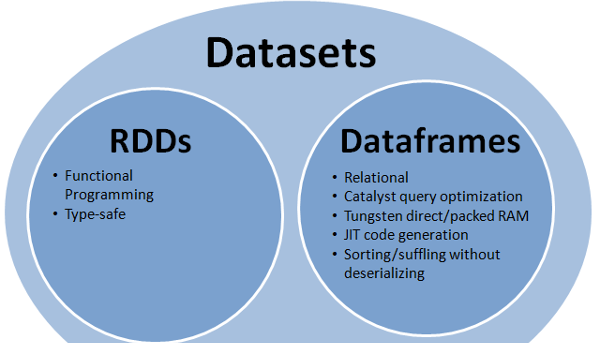
### [RDD、DataFrame、Dataset的区别和各自的优势](http://www.cnblogs.com/starwater/p/6841807.html)

**（来自网络** 原文链接地址：

<http://www.jianshu.com/p/c0181667daa0>

http://www.cnblogs.com/starwater/p/6841807.html

**）**



# RDD

优点:

1. 编译时类型安全   
   编译时就能检查出类型错误
2. 面向对象的编程风格   
   直接通过类名点的方式来操作数据

缺点:

1. 序列化和反序列化的性能开销   
   无论是集群间的通信, 还是IO操作都需要对对象的结构和数据进行序列化和反序列化.
2. GC的性能开销   
   频繁的创建和销毁对象, 势必会增加GC

import org.apache.spark.sql.SQLContext

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

object Run {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("test").setMaster("local")

val sc = new SparkContext(conf)

sc.setLogLevel("WARN")

val sqlContext = new SQLContext(sc)

/\*\*

\* id age

\* 1 30

\* 2 29

\* 3 21

\*/

case class Person(id: Int, age: Int)

val idAgeRDDPerson = sc.parallelize(Array(Person(1, 30), Person(2, 29), Person(3, 21)))

// 优点1

// idAge.filter(\_.age > "") // 编译时报错, int不能跟String比

// 优点2

idAgeRDDPerson.filter(\_.age > 25) // 直接操作一个个的person对象

}

}

# DataFrame

DataFrame引入了schema和off-heap

* schema : RDD每一行的数据, 结构都是一样的. 这个结构就存储在schema中. [**Spark**](http://lib.csdn.net/base/spark)通过schame就能够读懂数据, 因此在通信和IO时就只需要序列化和反序列化数据, 而结构的部分就可以省略了.
* off-heap : 意味着JVM堆以外的内存, 这些内存直接受[**操作系统**](http://lib.csdn.net/base/operatingsystem)管理（而不是JVM）。Spark能够以二进制的形式序列化数据(不包括结构)到off-heap中, 当要操作数据时, 就直接操作off-heap内存. 由于Spark理解schema, 所以知道该如何操作.

off-heap就像地盘, schema就像地图, Spark有地图又有自己地盘了, 就可以自己说了算了, 不再受JVM的限制, 也就不再收GC的困扰了.

通过schema和off-heap, DataFrame解决了RDD的缺点, 但是却丢了RDD的优点. DataFrame不是类型安全的, API也不是面向对象风格的.

import org.apache.spark.sql.types.{DataTypes, StructField, StructType}

import org.apache.spark.sql.{Row, SQLContext}

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

object Run {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("test").setMaster("local")

val sc = new SparkContext(conf)

sc.setLogLevel("WARN")

val sqlContext = new SQLContext(sc)

/\*\*

\* id age

\* 1 30

\* 2 29

\* 3 21

\*/

val idAgeRDDRow = sc.parallelize(Array(Row(1, 30), Row(2, 29), Row(4, 21)))

val schema = StructType(Array(StructField("id", DataTypes.IntegerType), StructField("age", DataTypes.IntegerType)))

val idAgeDF = sqlContext.createDataFrame(idAgeRDDRow, schema)

// API不是面向对象的

idAgeDF.filter(idAgeDF.col("age") > 25)

// 不会报错, DataFrame不是编译时类型安全的

idAgeDF.filter(idAgeDF.col("age") > "")

}

}

# DataSet

DataSet结合了RDD和DataFrame的优点, 并带来的一个新的概念Encoder

当序列化数据时, Encoder产生字节码与off-heap进行交互, 能够达到按需访问数据的效果, 而不用反序列化整个对象. Spark还没有提供自定义Encoder的API, 但是未来会加入.

下面看DataFrame和DataSet在2.0.0-preview中的实现

下面这段代码, 在1.6.x中创建的是DataFrame

// 上文DataFrame示例中提取出来的

val idAgeRDDRow = sc.parallelize(Array(Row(1, 30), Row(2, 29), Row(4, 21)))

val schema = StructType(Array(StructField("id", DataTypes.IntegerType), StructField("age", DataTypes.IntegerType)))

val idAgeDF = sqlContext.createDataFrame(idAgeRDDRow, schema)6

但是同样的代码在2.0.0-preview中, 创建的虽然还叫DataFrame

// sqlContext.createDataFrame(idAgeRDDRow, schema) 方法的实现, 返回值依然是DataFrame

def createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame = {

sparkSession.createDataFrame(rowRDD, schema)

}

但是其实却是DataSet, 因为DataFrame被声明为Dataset[Row]

package object sql {

// ...省略了不相关的代码

type DataFrame = Dataset[Row]

}

因此当我们从1.6.x迁移到2.0.0的时候, 无需任何修改就直接用上了DataSet.

下面是一段DataSet的示例代码

import org.apache.spark.sql.types.{DataTypes, StructField, StructType}

import org.apache.spark.sql.{Row, SQLContext}

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

object Test {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("test").setMaster("local") // 调试的时候一定不要用local[\*]

val sc = new SparkContext(conf)

val sqlContext = new SQLContext(sc)

import sqlContext.implicits.\_

val idAgeRDDRow = sc.parallelize(Array(Row(1, 30), Row(2, 29), Row(4, 21)))

val schema = StructType(Array(StructField("id", DataTypes.IntegerType), StructField("age", DataTypes.IntegerType)))

// 在2.0.0-preview中这行代码创建出的DataFrame, 其实是DataSet[Row]

val idAgeDS = sqlContext.createDataFrame(idAgeRDDRow, schema)

// 在2.0.0-preview中, 还不支持自定的Encoder, Row类型不行, 自定义的bean也不行

// 官方文档也有写通过bean创建Dataset的例子，但是我运行时并不能成功

// 所以目前需要用创建DataFrame的方法, 来创建DataSet[Row]

// sqlContext.createDataset(idAgeRDDRow)

// 目前支持String, Integer, Long等类型直接创建Dataset

Seq(1, 2, 3).toDS().show()

sqlContext.createDataset(sc.parallelize(Array(1, 2, 3))).show()

}

}

### 1.RDD与DataFrame的区别

下面的图直观地体现了DataFrame和RDD的区别。左侧的RDD[Person]虽然以Person为类型参数，但[**Spark**](http://lib.csdn.net/base/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多了数据的结构信息，即schema。RDD是分布式的[**Java**](http://lib.csdn.net/base/java)对象的集合。DataFrame是分布式的Row对象的集合。DataFrame除了提供了比RDD更丰富的算子以外，更重要的特点是提升执行效率、减少数据读取以及执行计划的优化，比如filter下推、裁剪等。   


### 2.提升执行效率

RDD API是函数式的，强调不变性，在大部分场景下倾向于创建新对象而不是修改老对象。这一特点虽然带来了干净整洁的API，却也使得Spark应用程序在运行期倾向于创建大量临时对象，对GC造成压力。在现有RDD API的基础之上，我们固然可以利用mapPartitions方法来重载RDD单个分片内的数据创建方式，用复用可变对象的方式来减小对象分配和GC的开销，但这牺牲了代码的可读性，而且要求开发者对Spark运行时机制有一定的了解，门槛较高。另一方面，Spark SQL在框架内部已经在各种可能的情况下尽量重用对象，这样做虽然在内部会打破了不变性，但在将数据返回给用户时，还会重新转为不可变数据。利用 DataFrame API进行开发，可以免费地享受到这些优化效果。

### 3.减少数据读取

分析[**大数据**](http://lib.csdn.net/base/hadoop)，最快的方法就是 ——忽略它。这里的“忽略”并不是熟视无睹，而是根据查询条件进行恰当的剪枝。

当查询的过滤条件中涉及到分区列时，我们可以根据查询条件剪掉肯定不包含目标数据的分区目录，从而减少IO。

对于一些“[**智能**](http://lib.csdn.net/base/aiplanning)”数据格式，Spark SQL还可以根据数据文件中附带的统计信息来进行剪枝。简单来说，在这类数据格式中，数据是分段保存的，每段数据都带有最大值、最小值、null值数量等 一些基本的统计信息。当统计信息表名某一数据段肯定不包括符合查询条件的目标数据时，该数据段就可以直接跳过（例如某整数列a某段的最大值为100，而查询条件要求a > 200）。

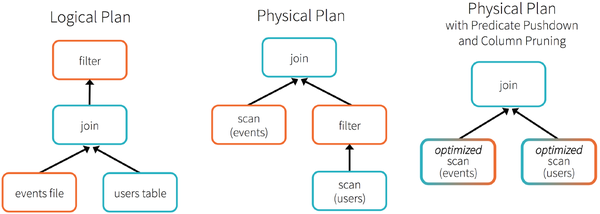
此外，Spark SQL也可以充分利用RCFile、ORC、Parquet等列式存储格式的优势，仅扫描查询真正涉及的列，忽略其余列的数据。

### 4.执行优化

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。

得到的优化执行计划在转换成物 理执行计划的过程中，还可以根据具体的数据源的特性将过滤条件下推至数据源内。最右侧的物理执行计划中Filter之所以消失不见，就是因为溶入了用于执行最终的读取操作的表扫描节点内。

对于普通开发者而言，查询优化 器的意义在于，即便是经验并不丰富的程序员写出的次优的查询，也可以被尽量转换为高效的形式予以执行。



### 5.RDD和DataSet

1.DataSet以Catalyst逻辑执行计划表示，并且数据以编码的二进制形式被存储，不需要反序列化就可以执行sorting、shuffle等操作。   
2.DataSet创立需要一个显式的Encoder，把对象序列化为二进制，可以把对象的scheme映射为Spark SQl类型，然而RDD依赖于运行时反射机制。

通过上面两点，DataSet的性能比RDD的要好很多。

### 6.DataFrame和DataSet

Dataset可以认为是DataFrame的一个特例，主要区别是Dataset每一个record存储的是一个强类型值而不是一个Row。因此具有如下三个特点：   
1.DataSet可以在编译时检查类型   
2.并且是面向对象的编程接口。

3.后面版本DataFrame会继承DataSet，DataFrame是面向Spark SQL的接口。

DataFrame和DataSet可以相互转化，df.as[ElementType]这样可以把DataFrame转化为DataSet，ds.toDF()这样可以把DataSet转化为DataFrame。

### 数据源

Spark SQL支持基于DataFrame操作一系列不同的数据源。DataFrame既可以当成一个普通RDD来操作，也可以将其注册成一个临时表来查询。把DataFrame注册为table之后，你就可以基于这个table执行SQL语句了。本节将描述加载和保存数据的一些通用方法，包含了不同的Spark数据源，然后深入介绍一下内建数据源可用选项。

### 通用加载/保存函数

在最简单的情况下，所有操作都会以默认类型数据源来加载数据（默认是Parquet，除非修改了spark.sql.sources.default 配置）。

**val** usersDF **=** spark.read.load("examples/src/main/resources/users.parquet")

usersDF.select("name", "favorite\_color").write.save("namesAndFavColors.parquet")

### 手动指定选项

你也可以手动指定数据源，并设置一些额外的选项参数。数据源可由其全名指定（如，org.apache.spark.sql.parquet），而对于内建支持的数据源，可以使用简写名（json, parquet, jdbc）。任意类型数据源创建的DataFrame都可以用下面这种语法转成其他类型数据格式。

**val** peopleDF **=** spark.read.format("json").load("examples/src/main/resources/people.json")

peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

### 直接对文件使用SQL

Spark SQL还支持直接对文件使用SQL查询，不需要用read方法把文件加载成DataFrame。

**val** sqlDF **=** spark.sql("SELECT \* FROM parquet.`examples/src/main/resources/users.parquet`")

### 保存模式

Save操作有一个可选参数SaveMode，用这个参数可以指定如何处理数据已经存在的情况。很重要的一点是，这些保存模式都没有加锁，所以其操作也不是原子性的。另外，如果使用Overwrite模式，实际操作是，先删除数据，再写新数据。

|  |  |  |
| --- | --- | --- |
| Scala/Java | 所有支持的语言 | 含义 |
| SaveMode.ErrorIfExists (default) | "error" (default) | （默认模式）从DataFrame向数据源保存数据时，如果数据已经存在，则抛异常。 |
| SaveMode.Append | "append" | 如果数据或表已经存在，则将DataFrame的数据追加到已有数据的尾部。 |
| SaveMode.Overwrite | "overwrite" | 如果数据或表已经存在，则用DataFrame数据覆盖之。 |
| SaveMode.Ignore | "ignore" | 如果数据已经存在，那就放弃保存DataFrame数据。这和SQL里CREATE TABLE IF NOT EXISTS有点类似。 |

### 保存到持久化表

在使用HiveContext的时候，DataFrame可以用saveAsTable方法，将数据保存成持久化的表。与registerTempTable不同，saveAsTable会将DataFrame的实际数据内容保存下来，并且在HiveMetastore中创建一个游标指针。持久化的表会一直保留，即使Spark程序重启也没有影响，只要你连接到同一个metastore就可以读取其数据。读取持久化表时，只需要用用表名作为参数，调用SparkSession.table方法即可得到对应DataFrame。

默认情况下，saveAsTable会创建一个”managed table“，也就是说这个表数据的位置是由metastore控制的。同样，如果删除表，其数据也会同步删除。

### Parquet文件

[Parquet](http://parquet.io/)是一种流行的列式存储格式。Spark SQL提供对Parquet文件的读写支持，而且Parquet文件能够自动保存原始数据的schema。写Parquet文件的时候，所有的字段都会自动转成nullable，以便向后兼容。

### 编程方式加载数据

仍然使用上面例子中的数据：

**Parquet FIles：**

Parquet是柱状格式文件，被许多其它数据处理系统支持。Spark SQL支持读取和写入Parquet文件，可自动保留原始数据的模式。当编写Parquet文件时，由于兼容性原因，所有列都将自动转换为允许为空。

**通过编程的方法加载数据：**

**import** **spark.implicits.\_**

**val** peopleDF **=** spark.read.json("examples/src/main/resources/people.json")

*// // DataFrames可以保存为Parquet文件，维护Schema信息*

peopleDF.write.parquet("people.parquet")

*//读取上面创建的parquet文件  
 // Parquet文件是自描述的，所以模式被保留  
 //加载Parquet文件的结果也是一个DataFrame***val** parquetFileDF **=** spark.read.parquet("people.parquet")

*// Parquet文件也可用于创建临时视图，然后在SQL语句中使用*parquetFileDF.createOrReplaceTempView("parquetFile")

**val** namesDF **=** spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19")

namesDF.map(attributes **=>** "Name: " + attributes(0)).show()

*// +------------+*

*// | value|*

*// +------------+*

*// |Name: Justin|*

*// +------------+*

### 分区发现

像Hive这样的系统，一个很常用的优化手段就是表分区。在一个支持分区的表中，数据是保存在不同的目录中的，并且将分区键以编码方式保存在各个分区目录路径中。Parquet数据源现在也支持自动发现和推导分区信息。例如，我们可以把之前用的人口数据存到一个分区表中，其目录结构如下所示，其中有2个额外的字段，gender和country，作为分区键：

path

└── to

└── table

├── gender=male

│ ├── ...

│ │

│ ├── country=US

│ │ └── data.parquet

│ ├── country=CN

│ │ └── data.parquet

│ └── ...

└── gender=female

├── ...

│

├── country=US

│ └── data.parquet

├── country=CN

│ └── data.parquet

└── ...

在这个例子中，如果需要读取Parquet文件数据，我们只需要把 path/to/table 作为参数传给 SparkSession.read.parquet 或者 SparkSession.read.load。Spark SQL能够自动的从路径中提取出分区信息，随后返回的DataFrame的schema如下：

root

|-- name: string (nullable = true)

|-- age: long (nullable = true)

|-- gender: string (nullable = true)

|-- country: string (nullable = true)

注意，分区键的数据类型将是自动推导出来的。目前，只支持数值类型和字符串类型数据作为分区键。

有的用户可能不想要自动推导出来的分区键数据类型。这种情况下，你可以通过 spark.sql.sources.partitionColumnTypeInference.enabled （默认是true）来禁用分区键类型推导。禁用之后，分区键总是被当成字符串类型。

从Spark-1.6.0开始，分区发现默认只在指定目录的子目录中进行。以上面的例子来说，如果用户把 path/to/table/gender=male 作为参数传给 SparkSession.read.parquet 或者 SparkSession.read.load，那么gender就不会被作为分区键。如果用户想要指定分区发现的基础目录，可以通过basePath选项指定。例如，如果把 path/to/table/gender=male作为数据目录，并且将basePath设为 path/to/table，那么gender仍然会最为分区键。

### Schema合并

像ProtoBuffer、Avro和Thrift一样，Parquet也支持schema演变。用户从一个简单的schema开始，逐渐增加所需的新字段。这样的话，用户最终会得到多个schema不同但互相兼容的Parquet文件。目前，Parquet数据源已经支持自动检测这种情况，并合并所有文件的schema。

因为schema合并相对代价比较大，并且在多数情况下不是必要的，所以从Spark-1.5.0之后，默认是被禁用的。你可以这样启用这一功能：

1. 读取Parquet文件时，将选项mergeSchema设为true（见下面的示例代码）
2. 或者，将全局选项spark.sql.parquet.mergeSchema设为true

*//为了支持RDD隐式转换为DataFrame*

**import** **spark.implicits.\_**

*// 创建另一个DataFrame放到新的分区目录中，*

**val** squaresDF **=** spark.sparkContext.makeRDD(1 to 5).map(i **=>** (i, i \* i)).toDF("value", "square")

squaresDF.write.parquet("data/test\_table/key=1")

*// 并增加一个新字段，丢弃一个老字段*

**val** cubesDF **=** spark.sparkContext.makeRDD(6 to 10).map(i **=>** (i, i \* i \* i)).toDF("value", "cube")

cubesDF.write.parquet("data/test\_table/key=2")

*// 读取分区表*

**val** mergedDF **=** spark.read.option("mergeSchema", "true").parquet("data/test\_table")

mergedDF.printSchema()

*// 最终的schema将由3个字段组成（single，double，triple）*

*// 并且分区键出现在目录路径中*

*// root*

*// |-- value: int (nullable = true)*

*// |-- square: int (nullable = true)*

*// |-- cube: int (nullable = true)*

*// |-- key: int (nullable = true)*

### Hive metastore Parquet table转换

在读写Hive metastore Parquet 表时，Spark SQL用的是内部的Parquet支持库，而不是Hive SerDe，因为这样性能更好。这一行为是由spark.sql.hive.convertMetastoreParquet 配置项来控制的，而且默认是启用的。

### Hive/Parquet schema调和

Hive和Parquet在表结构处理上主要有2个不同点：

1. Hive大小写敏感，而Parquet不是
2. Hive所有字段都是nullable的，而Parquet需要显示设置

由于以上原因，我们必须在Hive metastore Parquet table转Spark SQL Parquet table的时候，对Hive metastore schema做调整，调整规则如下：

1. 两种schema中字段名和字段类型必须一致（不考虑nullable）。调和后的字段类型必须在Parquet格式中有相对应的数据类型，所以nullable是也是需要考虑的。
2. 调和后Spark SQL Parquet table schema将包含以下字段：
   * 只出现在Parquet schema中的字段将被丢弃
   * 只出现在Hive metastore schema中的字段将被添加进来，并显式地设为nullable。

### 刷新元数据

Spark SQL会缓存Parquet元数据以提高性能。如果Hive metastore Parquet table转换被启用的话，那么转换过来的schema也会被缓存。这时候，如果这些表由Hive或其他外部工具更新了，你必须手动刷新元数据。

*// 注意，这里SparkSession是一个HiveContext*

spark.catalog.refreshTable("my\_table")

### 配置

Parquet配置可以通过 SparkSession.setConf 或者 SQL语句中 SET key=value来指定。

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.sql.parquet.binaryAsString | false | 有些老系统，如：特定版本的Impala，Hive，或者老版本的Spark SQL，不区分二进制数据和字符串类型数据。这个标志的意思是，让Spark SQL把二进制数据当字符串处理，以兼容老系统。 |
| spark.sql.parquet.int96AsTimestamp | true | 有些老系统，如：特定版本的Impala，Hive，把时间戳存成INT96。这个配置的作用是，让Spark SQL把这些INT96解释为timestamp，以兼容老系统。 |
| spark.sql.parquet.cacheMetadata | true | 缓存Parquet schema元数据。可以提升查询静态数据的速度。 |
| spark.sql.parquet.compression.codec | gzip | 设置Parquet文件的压缩编码格式。可接受的值有：uncompressed, snappy, gzip（默认）, lzo |
| spark.sql.parquet.filterPushdown | true | 启用过滤器下推优化，可以讲过滤条件尽量推导最下层，已取得性能提升 |
| spark.sql.hive.convertMetastoreParquet | true | 如果禁用，Spark SQL将使用Hive SerDe，而不是内建的对Parquet tables的支持 |
| spark.sql.parquet.output.committer.class | org.apache.parquet.hadoop. ParquetOutputCommitter | Parquet使用的数据输出类。这个类必须是 org.apache.hadoop.mapreduce.OutputCommitter的子类。一般来说，它也应该是 org.apache.parquet.hadoop.ParquetOutputCommitter的子类。注意：1. 如果启用spark.speculation, 这个选项将被自动忽略  2. 这个选项必须用hadoop configuration设置，而不是Spark SQLConf  3. 这个选项会覆盖 spark.sql.sources.outputCommitterClass  Spark SQL有一个内建的org.apache.spark.sql.parquet.DirectParquetOutputCommitter, 这个类的在输出到S3的时候比默认的ParquetOutputCommitter类效率高。 |
| spark.sql.parquet.mergeSchema | false | 如果设为true，那么Parquet数据源将会merge 所有数据文件的schema，否则，schema是从summary file获取的（如果summary file没有设置，则随机选一个） |

### JSON数据集

Spark SQL在加载JSON数据的时候，可以自动推导其schema并返回DataFrame。用SparkSession.read.json读取一个包含String的RDD或者JSON文件，即可实现这一转换。

注意，通常所说的json文件只是包含一些json数据的文件，而不是我们所需要的JSON格式文件。JSON格式文件必须每一行是一个独立、完整的的JSON对象。因此，一个常规的多行json文件经常会加载失败。

*//一个JSON 数据集是由路径指定的*

*// 路径既可以是单个文件，也可以还是存储文本文件的目录*

**val** path **=** "examples/src/main/resources/people.json"

**val** peopleDF **=** spark.read.json(path)

*//推导出来的schema，可由printSchema打印出来*

peopleDF.printSchema()

*// root*

*// |-- age: long (nullable = true)*

*// |-- name: string (nullable = true)*

*// 将DataFrame注册为table*

peopleDF.createOrReplaceTempView("people")

*// 可以根据spark提供的sql运行SQL语句*

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

teenagerNamesDF.show()

*// +------+*

*// | name|*

*// +------+*

*// |Justin|*

*// +------+*

*// 另一种方法是，用一个包含JSON字符串的RDD来创建DataFrame*

**val** otherPeopleRDD **=** spark.sparkContext.makeRDD(

"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: **Nil**)

**val** otherPeople **=** spark.read.json(otherPeopleRDD)

otherPeople.show()

*// +---------------+----+*

*// | address|name|*

*// +---------------+----+*

*// |[Columbus,Ohio]| Yin|*

*// +---------------+----+*

### Hive表

Spark SQL还支持读取和写入存储在Apache Hive中的数据。但是，由于Hive具有大量依赖关系，因此这些依赖关系不包含在默认Spark分支中。如果再类路径中找到Hive依赖项，Spark将自动加载他们。请注意，这些Hive依赖项也必须存在于所有工作节点上，因为他们将需要访问Hive序列化和反序列化库（SerDes），以便访问Hive中存储的数据。

Hive的配置是通过/conf中放置你的hive-site.xml，core-site.xml（用于安全配置）和hdfs-site.xml（用于HDFS配置）文件来完成

使用Hive时，必须使用Hive支持实例化SparkSession，包括连接到存在的Hive metastore，支持Hive SerDes和Hive用户定义的功能。没有现有Hive部署的用户仍然可以启动Hive支持。当hive-site.xml未配置时，上下文会自动在当前目录中创建metastore\_db,并创建由spark.sql.warehouse.dir配置的目录，该目录默认为Spark应用程序当前目录中的spark-warehouse目录，请注意，自从2.0.0依赖，hive-site.xml中的hive.metastore.warehouse.dir属性已被启用。而是使用spark.sql.warehouse.dir来指定仓库中数据库的默认位置。您可能需要向启动Spark应用程序的用户授予写权限。

**import** **org.apache.spark.sql.Row**

**import** **org.apache.spark.sql.SparkSession**

**case** **class** **Record**(key**:** Int, value**:** String)

*// warehouselocation指定管理数据库和表的默认位置*

**val** warehouseLocation **=** "spark-warehouse"

**val** spark **=** **SparkSession**

.builder()

.appName("Spark Hive Example")

.config("spark.sql.warehouse.dir", warehouseLocation)

.enableHiveSupport()

.getOrCreate()

**import** **spark.implicits.\_**

**import** **spark.sql**

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

*// 查询转化为HiveQL*

sql("SELECT \* FROM src").show()

*// +---+-------+*

*// |key| value|*

*// +---+-------+*

*// |238|val\_238|*

*// | 86| val\_86|*

*// |311|val\_311|*

*// ...*

*//支持聚合查询*

sql("SELECT COUNT(\*) FROM src").show()

*// +--------+*

*// |count(1)|*

*// +--------+*

*// | 500 |*

*// +--------+*

*//SQL查询的记过还是DataFrame，并且支持常见的函数*

**val** sqlDF **=** sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

*//DataFrame中每项的类型是Row,* *你可以按顺序访问每一列*

**val** stringsDS **=** sqlDF.map {

**case** **Row**(key**:** Int, value**:** String) **=>** s"Key: $key, Value: $value"

}

stringsDS.show()

*// +--------------------+*

*// | value|*

*// +--------------------+*

*// |Key: 0, Value: val\_0|*

*// |Key: 0, Value: val\_0|*

*// |Key: 0, Value: val\_0|*

*// ...*

*//还可以通过SparkSession在DataFrame上创建临时视图*

**val** recordsDF **=** spark.createDataFrame((1 to 100).map(i **=>** **Record**(i, s"val\_$i")))

recordsDF.createOrReplaceTempView("records")

*//可以对DataFrame和Hive中的数据进行join查询*

sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show()

*// +---+------+---+------+*

*// |key| value|key| value|*

*// +---+------+---+------+*

*// | 2| val\_2| 2| val\_2|*

*// | 4| val\_4| 4| val\_4|*

*// | 5| val\_5| 5| val\_5|*

*// ...*

### 和不同版本的Hive Metastore交互

Spark SQL对Hive最重要的支持之一就是和Hive metastore进行交互，这使得Spark SQL可以访问Hive表的元数据。从Spark-1.4.0开始，Spark SQL有专门单独的二进制build版本，可以用来访问不同版本的Hive metastore，其配置表如下。注意，不管所访问的hive是什么版本，Spark SQL内部都是以Hive 1.2.1编译的，而且内部使用的Hive类也是基于这个版本（serdes，UDFs，UDAFs等）

以下选项可用来配置Hive版本以便访问其元数据：

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.sql.hive.metastore.version | 1.2.1 | Hive metastore版本，可选的值为0.12.0 到 1.2.1 |
| spark.sql.hive.metastore.jars | builtin | 初始化HiveMetastoreClient的jar包。这个属性可以是以下三者之一：  1.builtin  目前内建为使用Hive-1.2.1，编译的时候启用-Phive，则会和spark一起打包。如果没有-Phive，那么spark.sql.hive.metastore.version要么是1.2.1，要就是未定义  2.maven  使用maven仓库下载的jar包版本。这个选项建议不要再生产环境中使用  3.JVM格式的classpath。这个classpath必须包含所有Hive及其依赖的jar包，且包含正确版本的hadoop。这些jar包必须部署在driver节点上，如果你使用yarn-cluster模式，那么必须确保这些jar包也随你的应用程序一起打包 |
| spark.sql.hive.metastore.sharedPrefixes | com.mysql.jdbc, org.postgresql, com.microsoft.sqlserver, oracle.jdbc | 一个逗号分隔的类名前缀列表，这些类使用classloader加载，且可以在Spark SQL和特定版本的Hive间共享。例如，用来访问hive metastore 的JDBC的driver就需要这种共享。其他需要共享的类，是与某些已经共享的类有交互的类。例如，自定义的log4j appender |
| spark.sql.hive.metastore.barrierPrefixes | (empty) | 一个逗号分隔的类名前缀列表，这些类在每个Spark SQL所访问的Hive版本中都会被显式的reload。例如，某些在共享前缀列表（spark.sql.hive.metastore.sharedPrefixes）中声明为共享的Hive UD函数 |

### 用JDBC连接其他数据库

Spark SQL也可以用JDBC访问其他数据库。这一功能应该优先于使用JdbcRDD。因为它返回一个DataFrame，而DataFrame在Spark SQL中操作更简单，且更容易和来自其他数据源的数据进行交互关联。JDBC数据源在java和python中用起来也很简单，不需要用户提供额外的ClassTag。（注意，这与Spark SQL JDBC server不同，Spark SQL JDBC server允许其他应用执行Spark SQL查询）

首先，你需要在spark classpath中包含对应数据库的JDBC driver，下面这行包括了用于访问postgres的数据库driver

SPARK\_CLASSPATH=postgresql-9.3-1102-jdbc41.jar bin/spark-shell

远程数据库的表可以通过Data Sources API，用DataFrame或者SparkSQL 临时表来装载。以下是选项列表：

|  |  |
| --- | --- |
| 属性名 | 含义 |
| url | 需要连接的JDBC URL |
| dbtable | 需要读取的JDBC表。注意，任何可以填在SQL的where子句中的东西，都可以填在这里。（既可以填完整的表名，也可填括号括起来的子查询语句） |
| driver | JDBC driver的类名。这个类必须在master和worker节点上都可用，这样各个节点才能将driver注册到JDBC的子系统中。 |
| partitionColumn, lowerBound, upperBound, numPartitions | 这几个选项，如果指定其中一个，则必须全部指定。他们描述了多个worker如何并行的读入数据，并将表分区。partitionColumn必须是所查询的表中的一个数值字段。注意，lowerBound和upperBound只是用于决定分区跨度的，而不是过滤表中的行。因此，表中所有的行都会被分区然后返回。 |
| fetchSize | JDBC fetch size，决定每次获取多少行数据。在JDBC驱动上设成较小的值有利于性能优化（如，Oracle上设为10） |

val jdbcDF = SparkSession.read.format("jdbc").options(

Map("url" -> "jdbc:postgresql:dbserver",

"dbtable" -> "schema.tablename")).load()

### 疑难解答

* JDBC driver class必须在所有client session或者executor上，对java的原生classloader可见。这是因为Java的DriverManager在打开一个连接之前，会做安全检查，并忽略所有对原声classloader不可见的driver。最简单的一种方法，就是在所有worker节点上修改compute\_classpath.sh，并包含你所需的driver jar包。
* 一些数据库，如H2，会把所有的名字转大写。对于这些数据库，在Spark SQL中必须也使用大写。

### 性能调整

对于有一定计算量的Spark作业来说，可能的性能改进的方式，不是把数据缓存在内存里，就是调整一些开销较大的选项参数。

### 内存缓存

Spark SQL可以通过调用SparkSession.cacheTable(“tableName”)或者DataFrame.cache()把tables以列存储格式缓存到内存中。随后，Spark SQL将会扫描必要的列，并自动调整压缩比例，以减少内存占用和GC压力。你也可以用SparkSession.uncacheTable(“tableName”)来删除内存中的table。

你还可以使用SparkSession.setConf 或在SQL语句中运行SET key=value命令，来配置内存中的缓存。

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.sql.inMemoryColumnarStorage.compressed | true | 如果设置为true，Spark SQL将会根据数据统计信息，自动为每一列选择单独的压缩编码方式。 |
| spark.sql.inMemoryColumnarStorage.batchSize | 10000 | 控制列式缓存批量的大小。增大批量大小可以提高内存利用率和压缩率，但同时也会带来OOM（Out Of Memory）的风险。 |

### 其他配置选项

以下选项同样也可以用来给查询任务调性能。不过这些选项在未来可能被放弃，因为spark将支持越来越多的自动优化。

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.sql.autoBroadcastJoinThreshold | 10485760 (10 MB) | 配置join操作时，能够作为广播变量的最大table的大小。设置为-1，表示禁用广播。注意，目前的元数据统计仅支持Hive metastore中的表，并且需要运行这个命令：ANALYSE TABLE <tableName> COMPUTE STATISTICS noscan |
| spark.sql.tungsten.enabled | true | 设为true，则启用优化的Tungsten物理执行后端。Tungsten会显式的管理内存，并动态生成表达式求值的字节码 |
| spark.sql.shuffle.partitions | 200 | 配置数据混洗（shuffle）时（join或者聚合操作），使用的分区数。 |

### 分布式SQL引擎

Spark SQL可以作为JDBC/ODBC或者命令行工具的分布式查询引擎。在这种模式下，终端用户或应用程序，无需写任何代码，就可以直接在Spark SQL中运行SQL查询。

### 运行Thrift JDBC/ODBC server

这里实现的Thrift JDBC/ODBC server和Hive-1.2.1中的[HiveServer2](https://cwiki.apache.org/confluence/display/Hive/Setting+Up+HiveServer2)是相同的。你可以使用beeline脚本来测试Spark或者Hive-1.2.1的JDBC server。

在Spark目录下运行下面这个命令，启动一个JDBC/ODBC server

./sbin/start-thriftserver.sh

这个脚本能接受所有 bin/spark-submit 命令支持的选项参数，外加一个 –hiveconf 选项，来指定Hive属性。运行./sbin/start-thriftserver.sh –help可以查看完整的选项列表。默认情况下，启动的server将会在localhost:10000端口上监听。要改变监听主机名或端口，可以用以下环境变量：

export HIVE\_SERVER2\_THRIFT\_PORT=<listening-port>

export HIVE\_SERVER2\_THRIFT\_BIND\_HOST=<listening-host>

./sbin/start-thriftserver.sh \

--master <master-uri> \

...

或者Hive系统属性来指定

./sbin/start-thriftserver.sh \

--hiveconf hive.server2.thrift.port=<listening-port> \

--hiveconf hive.server2.thrift.bind.host=<listening-host> \

--master <master-uri>

...

接下来，你就可以开始在beeline中测试这个Thrift JDBC/ODBC server:

./bin/beeline

下面的指令，可以连接到一个JDBC/ODBC server

beeline> !connect jdbc:hive2://localhost:10000

可能需要输入用户名和密码。在非安全模式下，只要输入你本机的用户名和一个空密码即可。对于安全模式，请参考[beeline documentation](https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients).

Hive的配置是在conf/目录下的hive-site.xml，core-site.xml，hdfs-site.xml中指定的。

你也可以在beeline的脚本中指定。

Thrift JDBC server也支持通过HTTP传输Thrift RPC消息。以下配置（在conf/hive-site.xml中）将启用HTTP模式：

hive.server2.transport.mode - Set this to value: http

hive.server2.thrift.http.port - HTTP port number fo listen on; default is 10001

hive.server2.http.endpoint - HTTP endpoint; default is cliservice

同样，在beeline中也可以用HTTP模式连接JDBC/ODBC server:

beeline> !connect jdbc:hive2://<host>:<port>/<database>?hive.server2.transport.mode=http;hive.server2.thrift.http.path=<http\_endpoint>

### 使用Spark SQL命令行工具

Spark SQL CLI是一个很方便的工具，它可以用local mode运行hive metastore service，并且在命令行中执行输入的查询。注意Spark SQL CLI目前还不支持和Thrift JDBC server通信。

用如下命令，在spark目录下启动一个Spark SQL CLI

./bin/spark-sql

Hive配置在conf目录下hive-site.xml，core-site.xml，hdfs-site.xml中设置。你可以用这个命令查看完整的选项列表：./bin/spark-sql –help

### 升级指南

**2.0到2.1：**

从Spark SQL2.0升级到2.1

数据源表现在将分区元数据存储在Hive metastore中。这意味着Hive DDL如ALTER TABLE PARTITION...SET LOCATION现在可用于使用Datasource API创建的表。

遗留的数据源表可以通过MSCK PEPAIR TABLE命令迁移到这种格式。建议迁移遗留表以利用Hive DDL支持和改进的计划性能。

要确定表是否已迁移，请在表上发出DESRIBE FORMATTED时查找PartitionProvider：Catalog属性。

更改INSERT OVERWRITE TABLE... PARTITION...数据源表的行为。

在先前的Spark版本中，及时给定了分区规范，INSERT OVERWRITE也覆盖了整个Datasource表。现在只有匹配规范的分区被覆盖。

请注意，这仍然与Hive表的行为不同，Hive表仅覆盖与新插入数据重叠的分区。

**1.6到2.0：**

SparkSession现在是Spark的新入口点，可以替换旧的SQLContext和HiveContext。请注意，旧的SQLContext和HiveContext保持向后兼容。可以从SparkSession访问新的目录界面 - 数据库上的现有API和诸如listTable，createExternalTable，dropTempView，cacheTable之类的表访问。

Dataset API和DataFrame API是统一的。在Scala中，DataFrame成为Dataset [Row]的类型别名，而Java API用户必须使用Dataset <Row>替换DataFrame。类型化变换（例如，map，filter和groupByKey）和无类型转换（例如select和groupBy）在Dataset类上都可用。由于Python和R中的编译时类型安全性不是语言特性，因此Dataset的概念不适用于这些语言的API。相反，DataFrame仍然是主要的编程抽象，它类似于这些语言中的单节点数据帧概念。

DataSet集和DataFrame API unionAll已被弃用并替换为union

DataSet和DataFrame API explode 已被弃用，替代方案是使用带有select或flatMap的functions.explode（）

DataSet和DataFrame API registerTempTable已被弃用，并被createOrReplaceTempView替换

更改CREATE TABLE ... Hive表的LOCATION行为。

从Spark 2.0，CREATE TABLE ... LOCATION等同于CREATE EXTERNAL TABLE ... LOCATION，以防止意外丢弃用户提供的位置中的现有数据。这意味着，在用户指定位置的Spark SQL中创建的Hive表始终是Hive外部表。删除外部表将不会删除数据。用户不能指定Hive管理表的位置。请注意，这与Hive行为不同。

1.5升级到1.6

* 从Spark-1.6.0起，默认Thrift server 将运行于多会话并存模式下（multi-session）。这意味着，每个JDBC/ODBC连接有其独立的SQL配置和临时函数注册表。table的缓存仍然是公用的。如果你更喜欢老的单会话模式，只需设置spark.sql.hive.thriftServer.singleSession为true即可。当然，你也可在spark-defaults.conf中设置，或者将其值传给start-thriftserver.sh –conf（如下）：

./sbin/start-thriftserver.sh \

--conf spark.sql.hive.thriftServer.singleSession=true \

...

1.4升级到1.5

* Tungsten引擎现在默认是启用的，Tungsten是通过手动管理内存优化执行计划，同时也优化了表达式求值的代码生成。这两个特性都可以通过把spark.sql.tungsten.enabled设为false来禁用。
* Parquet schema merging默认不启用。需要启用的话，设置spark.sql.parquet.mergeSchema为true即可
* Python接口支持用点(.)来访问字段内嵌值，例如df[‘table.column.nestedField’]。但这也意味着，如果你的字段名包含点号(.)的话，你就必须用重音符来转义，如：table.`column.with.dots`.nested。
* 列式存储内存分区剪枝默认是启用的。要禁用，设置spark.sql.inMemoryColumarStorage.partitionPruning为false即可
* 不再支持无精度限制的decimal。Spark SQL现在强制最大精度为38位。对于BigDecimal对象，类型推导将会使用（38，18）精度的decimal类型。如果DDL中没有指明精度，默认使用的精度是（10，0）
* 时间戳精确到1us（微秒），而不是1ns（纳秒）
* 在“sql”这个SQL变种设置中，浮点数将被解析为decimal。HiveQL解析保持不变。
* 标准SQL/DataFrame函数均为小写，例如：sum vs SUM。
* 当推测任务被启用是，使用DirectOutputCommitter是不安全的，因此，DirectOutputCommitter在推测任务启用时，将被自动禁用，且忽略相关配置。
* JSON数据源不再自动加载其他程序产生的新文件（例如，不是Spark SQL插入到dataset中的文件）。对于一个JSON的持久化表（如：Hive metastore中保存的表），用户可以使用REFRESH TABLE这个SQL命令或者HiveContext.refreshTable来把新文件包括进来。

1.3升级到1.4

DataFrame数据读写接口

根据用户的反馈，我们提供了一个新的，更加流畅的API，用于数据读（SparkSession.read）写（DataFrame.write），同时老的API（如：SparkSession.parquetFile, SparkSession.jsonFile）将被废弃。

有关SparkSession.read和DataFrame.write的更详细信息，请参考API文档。

DataFrame.groupBy保留分组字段

根据用户的反馈，我们改变了DataFrame.groupBy().agg()的默认行为，在返回的DataFrame结果中保留了分组字段。如果你想保持1.3中的行为，设置spark.sql.retainGroupColumns为false即可。

// 在1.3.x中，如果要保留分组字段"department", 你必须显式的在agg聚合时包含这个字段

df.groupBy("department").agg($"department", max("age"), sum("expense"))

// 而在1.4+，分组字段"department"默认就会包含在返回的DataFrame中

df.groupBy("department").agg(max("age"), sum("expense"))

// 要回滚到1.3的行为（不包含分组字段），按如下设置即可：

SparkSession.setConf("spark.sql.retainGroupColumns", "false")

1.2升级到1.3

在Spark 1.3中，我们去掉了Spark SQL的”Alpha“标签，并清理了可用的API。从Spark 1.3起，Spark SQL将对1.x系列二进制兼容。这个兼容性保证不包括显式的标注为”unstable（如：DeveloperAPI或Experimental）“的API。

SchemaRDD重命名为DataFrame

对于用户来说，Spark SQL 1.3最大的改动就是SchemaRDD改名为DataFrame。主要原因是，DataFrame不再直接由RDD派生，而是通过自己的实现提供RDD的功能。DataFrame只需要调用其rdd方法就能转成RDD。

在Scala中仍然有SchemaRDD，只不过这是DataFrame的一个别名，以便兼容一些现有代码。但仍然建议用户改用DataFrame。Java和Python用户就没这个福利了，他们必须改代码。

统一Java和Scala API

在Spark 1.3之前，有单独的java兼容类（JavaSparkSession和JavaSchemaRDD）及其在Scala API中的镜像。Spark 1.3中将Java API和Scala API统一。两种语言的用户都应该使用SparkSession和DataFrame。一般这些类中都会使用两种语言中都有的类型（如：Array取代各语言独有的集合）。有些情况下，没有通用的类型（例如：闭包或者maps），将会使用函数重载来解决这个问题。

另外，java特有的类型API被删除了。Scala和java用户都应该用org.apache.spark.sql.types来编程描述一个schema。

隐式转换隔离，DSL包移除 – 仅针对scala

Spark 1.3之前的很多示例代码，都在开头用 import SparkSession.\_，这行将会导致所有的SparkSession的函数都被引入进来。因此，在Spark 1.3我们把RDDs到DataFrames的隐式转换隔离出来，单独放到SparkSession.implicits对象中。用户现在应该这样写：import SparkSession.implicits.\_

另外，隐式转换也支持由Product（如：case classes或tuples）组成的RDD，但需要调用一个toDF方法，而不是自动转换。

如果需要使用DSL（被DataFrame取代的API）中的方法，用户之前需要导入DSL（import org.apache.spark.sql.catalyst.dsl），而现在应该要导入 DataFrame API（import org.apache.spark.sql.functions.\_）

移除org.apache.spark.sql中DataType别名 – 仅针对scala

Spark 1.3删除了sql包中的DataType类型别名。现在，用户应该使用 org.apache.spark.sql.types中的类。

UDF注册挪到SparkSession.udf中 – 针对java和scala

注册UDF的函数，不管是DataFrame，DSL或者SQL中用到的，都被挪到SparkSession.udf中。

SparkSession.udf.register("strLen", (s: String) => s.length())

Python UDF注册保持不变。

Python DataTypes不再是单例

在python中使用DataTypes，你需要先构造一个对象（如：StringType()），而不是引用一个单例。

### Shark用户迁移指南

调度

用户可以通过如下命令，为JDBC客户端session设定一个[Fair Scheduler](http://spark.apache.org/docs/latest/job-scheduling.html#fair-scheduler-pools) pool。

SET spark.sql.thriftserver.scheduler.pool=accounting;

Reducer个数

在Shark中，默认的reducer个数是1，并且由mapred.reduce.tasks设定。Spark SQL废弃了这个属性，改为 spark.sql.shuffle.partitions, 并且默认200，用户可通过如下SET命令来自定义：

SET spark.sql.shuffle.partitions=10;

SELECT page, count(\*) c

FROM logs\_last\_month\_cached

GROUP BY page ORDER BY c DESC LIMIT 10;

你也可以把这个属性放到hive-site.xml中来覆盖默认值。

目前，mapred.reduce.tasks属性仍然能被识别，并且自动转成spark.sql.shuffle.partitions

缓存

shark.cache表属性已经不存在了，并且以”\_cached”结尾命名的表也不再会自动缓存。取而代之的是，CACHE TABLE和UNCACHE TABLE语句，用以显式的控制表的缓存：

CACHE TABLE logs\_last\_month;

UNCACHE TABLE logs\_last\_month;

注意：CACHE TABLE tbl 现在默认是饥饿模式，而非懒惰模式。再也不需要手动调用其他action来触发cache了！

从Spark-1.2.0开始，Spark SQL新提供了一个语句，让用户自己控制表缓存是否是懒惰模式

CACHE [LAZY] TABLE [AS SELECT] ...

以下几个缓存相关的特性不再支持：

* 用户定义分区级别的缓存逐出策略
* RDD 重加载
* 内存缓存直接写入策略

### 兼容Apache Hive

Spark SQL设计时考虑了和Hive metastore，SerDes以及UDF的兼容性。目前这些兼容性斗是基于Hive-1.2.1版本，并且Spark SQL可以连到不同版本的Hive metastore（从0.12.0到1.2.1，参考：<http://spark.apache.org/docs/latest/sql-programming-guide.html#interacting-with-different-versions-of-hive-metastore>）

部署在已有的Hive仓库之上

Spark SQL Thrift JDBC server采用了”out of the box”（开箱即用）的设计，使用很方便，并兼容已有的Hive安装版本。你不需要修改已有的Hive metastore或者改变数据的位置，或者表分区。

支持的Hive功能

Spark SQL 支持绝大部分Hive功能，如：

* Hive查询语句：
  + SELECT
  + GROUP BY
  + ORDER BY
  + CLUSTER BY
  + SORT BY
* 所有的Hive操作符：
  + Relational operators (=, ⇔, ==, <>, <, >, >=, <=, etc)
  + Arithmetic operators (+, -, \*, /, %, etc)
  + Logical operators (AND, &&, OR, ||, etc)
  + Complex type constructors
  + Mathematical functions (sign, ln, cos, etc)
  + String functions (instr, length, printf, etc)
* 用户定义函数（UDF）
* 用户定义聚合函数（UDAF）
* 用户定义序列化、反序列化（SerDes）
* 窗口函数（Window functions）
* Joins
  + JOIN
  + {LEFT|RIGHT|FULL} OUTER JOIN
  + LEFT SEMI JOIN
  + CROSS JOIN
* Unions
* 查询子句
  + SELECT col FROM ( SELECT a + b AS col from t1) t2
* 采样
* 执行计划详细（Explain）
* 分区表，包括动态分区插入
* 视图
* 所有Hive DDL（data definition language）：
  + CREATE TABLE
  + CREATE TABLE AS SELECT
  + ALTER TABLE
* 绝大部分Hive数据类型：
  + TINYINT
  + SMALLINT
  + INT
  + BIGINT
  + BOOLEAN
  + FLOAT
  + DOUBLE
  + STRING
  + BINARY
  + TIMESTAMP
  + DATE
  + ARRAY<>
  + MAP<>
  + STRUCT<>

不支持的Hive功能

以下是目前不支持的Hive特性的列表。多数是不常用的。

不支持的Hive常见功能

* bucket表：butcket是Hive表的一个哈希分区

不支持的Hive高级功能

* UNION类操作
* 去重join
* 字段统计信息收集：Spark SQL不支持同步的字段统计收集

Hive输入、输出格式

* CLI文件格式：对于需要回显到CLI中的结果，Spark SQL仅支持TextOutputFormat。
* Hadoop archive — Hadoop归档

Hive优化

一些比较棘手的Hive优化目前还没有在Spark中提供。有一些（如索引）对应Spark SQL这种内存计算模型来说并不重要。另外一些，在Spark SQL未来的版本中会支持。

* 块级别位图索引和虚拟字段（用来建索引）
* 自动计算reducer个数（join和groupBy算子）：目前在Spark SQL中你需要这样控制混洗后（post-shuffle）并发程度：”SET spark.sql.shuffle.partitions=[num\_tasks];”
* 元数据查询：只查询元数据的请求，Spark SQL仍需要启动任务来计算结果
* 数据倾斜标志：Spark SQL不会理会Hive中的数据倾斜标志
* STREAMTABLE join提示：Spark SQL里没有这玩艺儿
* 返回结果时合并小文件：如果返回的结果有很多小文件，Hive有个选项设置，来合并小文件，以避免超过HDFS的文件数额度限制。Spark SQL不支持这个。

### 参考

数据类型

Spark SQL和DataFrames支持如下数据类型：

* Numeric types（数值类型）
  + ByteType: 1字节长的有符号整型，范围：-128到127.
  + ShortType: 2字节长有符号整型，范围：-32768到32767.
  + IntegerType: 4字节有符号整型，范围：-2147483648到2147483647.
  + LongType: 8字节有符号整型，范围：-9223372036854775808 to 9223372036854775807.
  + FloatType: 4字节单精度浮点数。
  + DoubleType: 8字节双精度浮点数
  + DecimalType: 任意精度有符号带小数的数值。内部使用java.math.BigDecimal, BigDecimal包含任意精度的不缩放整型，和一个32位的缩放整型
* String type（字符串类型）
  + StringType: 字符串
* Binary type（二进制类型）
  + BinaryType: 字节序列
* Boolean type（布尔类型）
  + BooleanType: 布尔类型
* Datetime type（日期类型）
  + TimestampType: 表示包含年月日、时分秒等字段的日期
  + DateType: 表示包含年月日字段的日期
* Complex types（复杂类型）
  + ArrayType(elementType, containsNull)：数组类型，表达一系列的elementType类型的元素组成的序列，containsNull表示数组能否包含null值
  + MapType(keyType, valueType, valueContainsNull)：映射集合类型，表示一个键值对的集合。键的类型是keyType，值的类型则由valueType指定。对应MapType来说，键是不能为null的，而值能否为null则取决于valueContainsNull。
  + StructType(fields)：表示包含StructField序列的结构体。
    - StructField(name, datatype, nullable): 表示StructType中的一个字段，name是字段名，datatype是数据类型，nullable表示该字段是否可以为空

所有Spark SQL支持的数据类型都在这个包里：org.apache.spark.sql.types，你可以这样导入之：

import org.apache.spark.sql.types.\_

|  |  |  |
| --- | --- | --- |
| Data type | Value type in Scala | API to access or create a data type |
| ByteType | Byte | ByteType |
| ShortType | Short | ShortType |
| IntegerType | Int | IntegerType |
| LongType | Long | LongType |
| FloatType | Float | FloatType |
| DoubleType | Double | DoubleType |
| DecimalType | java.math.BigDecimal | DecimalType |
| StringType | String | StringType |
| BinaryType | Array[Byte] | BinaryType |
| BooleanType | Boolean | BooleanType |
| TimestampType | java.sql.Timestamp | TimestampType |
| DateType | java.sql.Date | DateType |
| ArrayType | scala.collection.Seq | ArrayType(*elementType*, [*containsNull*])注意：默认containsNull为true |
| MapType | scala.collection.Map | MapType(*keyType*, *valueType*, [*valueContainsNull*])注意：默认valueContainsNull为true |
| StructType | org.apache.spark.sql.Row | StructType(*fields*)注意：fields是一个StructFields的序列，并且同名的字段是不允许的。 |
| StructField | 定义字段的数据对应的Scala类型（例如，如果StructField的dataType为IntegerType，则其数据对应的scala类型为Int） | StructField(name, dataType, nullable) |

NaN语义

这是Not-a-Number的缩写，某些float或double类型不符合标准浮点数语义，需要对其特殊处理：

* NaN == NaN，即：NaN和NaN总是相等
* 在聚合函数中，所有NaN分到同一组
* NaN在join操作中可以当做一个普通的join key
* NaN在升序排序中排到最后，比任何其他数值都大