1 Spark SQL介绍

1.1 Spark SQL 是什么

Spark SQL是一个用来处理结构化数据的Spark组件。

问题

结构化数据和非结构化数据？

结构化数据指的是数据在一个文件里以固定字段存储的数据。它通常包括关系数据库里存储的数据和表格数据。结构化数据一般依赖于一个数据模型，该数据模型定义了数据是怎么样被存储，处理和获取， 也包括要存储的字段信息以及如何如何存储：数据类型（数值型、字符型，日期型，地址等）和对数据的限制，可以限制使用的具体内容字段。

非结构化数据涉及的信息要么没有预定义的数据模型或者没有预定义的组织方式。非结构化数据比如大的文本文件，可能包含数据比如日期、数字、和事实信息等，由于数据本身没有规律甚至概念模糊，使得对这些数据进行分析难度比较大。

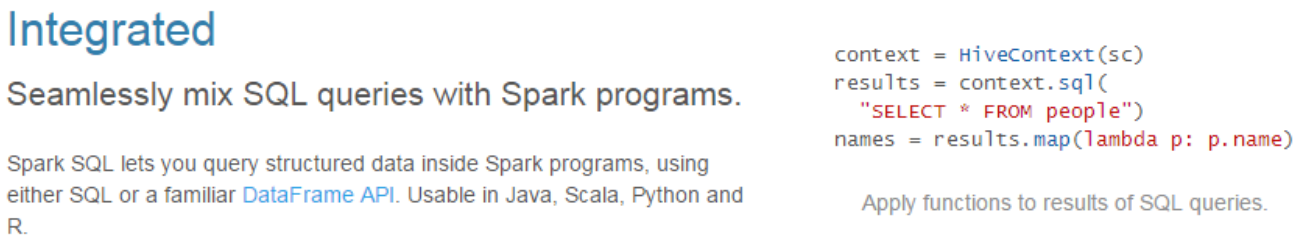
1.2 Spark SQL的由来

Spark 1.0版本开始，推出了Spark SQL。它能够利用Spark进行结构化数据的存储和操作，结构化数据既可以来自外部结构化数据源（支持Hive，JSON,Parquet等），也可以通过向已有RDD增加Schema的方式得到。 其实最早使用的，都是Hadoop自己的Hive查询引擎；但是后来Spark提供了Shark； 再后来Shark被淘汰，推出了Spark SQL。 Shark的性能比Hive就要高出一个数量级，而Spark SQL的性能又比Shark高出一个数量级。

1.3 了解Spark SQL

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。所有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！

1.易整合



2.统一的数据访问方式

Uniform Data Access

Connect to any data source the same way.

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. You can even join data across these sources.

```
context.jsonFile("s3n://...")
  .registerTempTable("json")
results = context.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

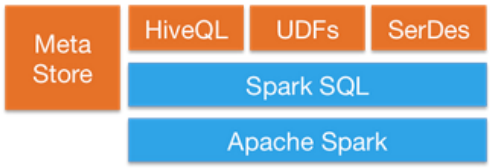
Query and join different data sources.

3.兼容Hive

Hive Compatibility

Run unmodified Hive queries on existing data.

Spark SQL reuses the Hive frontend and metastore, giving you full compatibility with existing Hive data, queries, and UDFs. Simply install it alongside Hive.



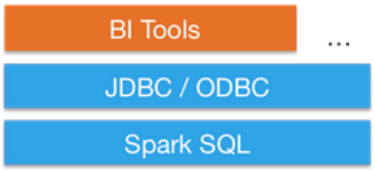
Spark SQL can use existing Hive metastores, SerDes, and UDFs.

4.标准的数据连接

Standard Connectivity

Connect through JDBC or ODBC.

A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.



Use your existing BI tools to query big data.

总结

SparkSQL是Spark上的高级模块，SparkSQL是一个SQL解析引擎，将SQL解析成特殊的RDD（DataFrame），然后在Spark集群中运行

SparkSQL是用来处理结构化数据的（先将非结构化的数据转换成结构化数据）

SparkSQL支持两种编程API 1.SQL方式 2.DataFrame的方式（DSL）

SparkSQL兼容hive（元数据库、SQL语法、UDF、序列化、反序列化机制）

SparkSQL支持统一的数据源，课程读取多种类型的数据

SparkSQL提供了标准的连接（JDBC、ODBC），以后可以对接一下BI工具

2. DataFrame

2.1 DataFrame是什么

在Spark中，DataFrame是一种按列组织的分布式数据集，概念上等价于关系数据库中一个表或者是Python中的data frame，但是在底层进行了更丰富的优化。

2.2 DataFrame与RDD对比

RDD和DataFrame的区别

DataFrame里面存放的结构化数据的描述信息，DataFrame要有表头（表的描述信息），描述了有多少列，每一列数叫什么名字、什么类型、能不能为空？

DataFrame是特殊的RDD（RDD+Schema信息就变成了DataFrame）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。

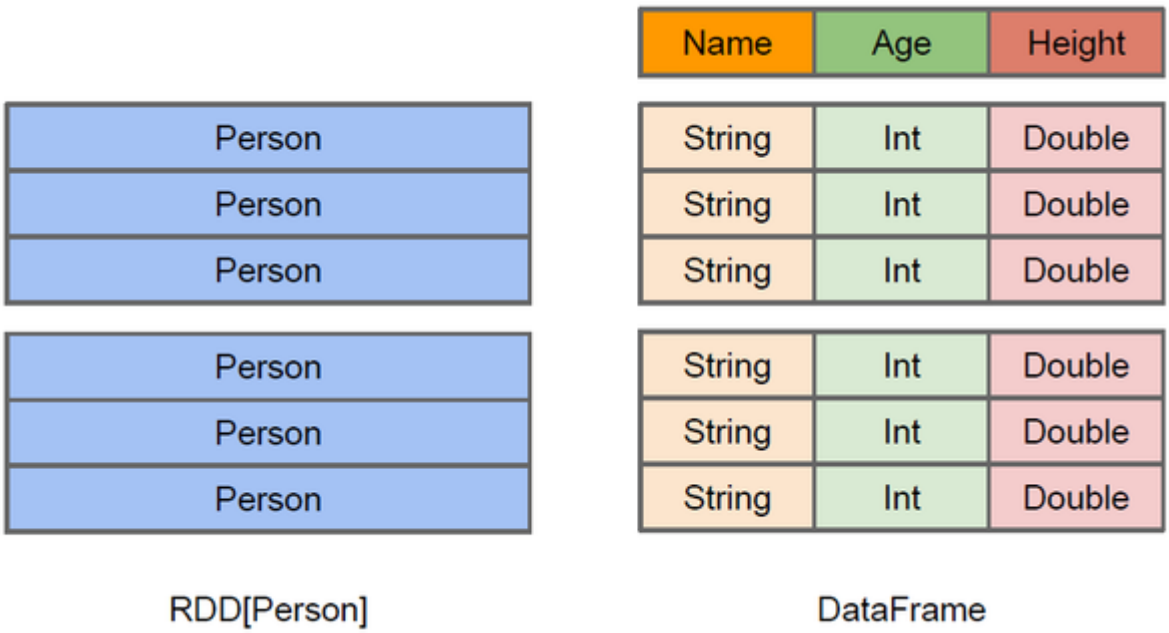
与RDD的主要区别在于：前者带有Schema元数据，即DataFrame所表示的二维数据集的每一列都有名称和类型。由于无法知道RDD数据集内部的结构，Spark执行作业只能在调度阶段进行简单通用的优化，而DataFrame带有数据集内部的结构，可以根据这些信息进行针对性的优化，最终实现优化运行效率。

DataFrame带来的好处：

精简代码

提升执行效率

减少数据读取：忽略无关数据，根据查询条件进行适当裁剪。



在老的版本中，SparkSQL提供两种SQL查询起始点，一个叫SQLContext，用于Spark自己提供的SQL查询，一个叫HiveContext，用于连接Hive的查询，SparkSession是Spark最新的SQL查询起始点，实质上是SQLContext和HiveContext的组合，所以在SQLContext和HiveContext上可用的API在SparkSession上同样是可以使用的。SparkSession内部封装了sparkContext，所以计算实际上是由sparkContext完成的。

2.3 Spark-shell基本操作

ps:数据使用的是Spark中所提供的样例数据

```
spark.read.json("/opt/software/spark-2.2.0-bin-
hadoop2.7/examples/src/main/resources/people.json")
df.show()
df.filter($"age" > 21).show()
df.createOrReplaceTempView("persons")
spark.sql("select* from persons").show()

spark.sql("select * from persons where age > 21").show()
```

2.4 IDEA编写SparkSQL

在maven的pom.xml配置文件中添加配置

```
<dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.11</artifactId>
            <version>${spark.version}</version>
        </dependency>
```

SparkSession的三种创建方式

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.{DataFrame, SparkSession}
/**
  * SparkSession三种创建方式
  */
object SparkSQLDemo {
  def main(args: Array[String]): Unit = {
    /**
      * 创建SparkSession方式1
      * builder用于创建一个SparkSession。
      * appName设置App的名字
      * master设置运行模式(集群模式不用设置)
      * getOrCreate 进行创建
      */
    val sparks1 = SparkSession.builder().appName("SparkSQLDemo").master("local").getOrCreate()
    /**
      * 创建SparkSession方式2
      * 先通过SparkConf创建配置对象
      * SetAppName设置应用的名字
      * SetMaster设置运行模式(集群模式不用设置)
      * 在通过SparkSession创建对象
      * 通过config传入conf配置对象来创建SparkSession
      * getOrCreate 进行创建
      */
    val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")
    val sparks2 = SparkSession.builder().config(conf).getOrCreate()
    /**
      * 创建SparkSession方式3(操作hive)
      * uilder用于创建一个SparkSession。
      * appName设置App的名字
      * master设置运行模式(集群模式不用设置)
      * enableHiveSupport 开启hive操作
      * getOrCreate 进行创建
      */
    val sparkh = SparkSession.builder().appName("SparkSQLDemo").
master("local").enableHiveSupport().getOrCreate()


    //关闭
```

```
    sparks1.stop()
    sparks2.stop()
    sparkh.stop()
  }
}
```

2.5 RDD到DataFrame

2.5.1直接手动确定

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.{DataFrame, SparkSession}

import org.apache.spark.rdd.RDD
/**
  * RDD--->DataFrame 直接手动确定
  */
object SparkSQLDemo {
  def main(args: Array[String]): Unit = {
    //创建SparkConf()并设置App名称
    val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")
    //SQLContext要依赖SparkContext
    val sc = new SparkContext(conf)
    //从指定的地址创建RDD
    val lineRDD = sc.textFile("dir/people.txt").map(_.split(","))
    //这里是将数据转换为元组,数据量少可以使用这种方式
    val tuple: RDD[(String, Int)] = lineRDD.map(x => (x(0),x(1).trim().toInt))
    val spark = SparkSession.builder().config(conf).getOrCreate()
    //如果需要RDD于DataFrame之间操作,那么需要引用 import spark.implicits._ [spark不是包名,是
SparkSession对象]
    import spark.implicits._
    val frame: DataFrame = tuple.toDF("name","age")
    frame.show()
    sc.stop();
  }
}
```

2.5.2 反射来创建DataFrame

在1.x的基础上，由普通的RDD转化成DataFrame，然后执行SQL，具体步骤如下：

1.创建SparkSession 2.先创建RDD，对数据进行整理，然后关联case class，将非结构化数据转换成结构化数据 3.显示的调用toDF方法将RDD转换成DataFrame 4.注册临时表 5.执行SQL（Transformation，lazy） 6.执行Action

由普通的RDD转化成DataFrame:

使用反射来推断包含了特定数据类型的RDD的元数据。这种基于反射的方式，代码比较简洁，当你已经知道你的RDD的元数据时，是一种非常不错的方式。

```
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.{DataFrame, SQLContext}
import org.apache.spark.{SparkConf, SparkContext}

object SQLDemo1 {

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

    //提交的这个程序可以连接到Spark集群中
    val conf = new SparkConf().setAppName("SQLDemo1").setMaster("local[2]")

    /* 1.创建SparkSession*/

 val sparks1 = SparkSession.builder().appName("SparkSQLDemo").master("local").getOrCreate()
    //创建特殊的RDD（DataFrame），就是有schema信息的RDD

   /* 2.先创建RDD，然后关联case class，将非结构化数据转换成结构化数据*/

    //先有一个普通的RDD，然后在关联上schema，进而转成DataFrame
    val lines = sc.textFile("c://data/person.txt")
    //将数据进行整理，给原有数据增加schema信息
    val personRDD: RDD[Person] = lines.map(line => {
      val fields = line.split(",")
      val id = fields(0).toLong
      val name = fields(1)
      val age = fields(2).toInt
      val fv = fields(3).toDouble
      Person(id, name, age, fv)
    })
      //case class就定义了元数据。Spark SQL会通过反射读取传递给case class的参数的名称，然后将其作为列名。
    //该RDD装的是Person类型的数据，有了shcma信息，但是还是一个RDD

    /* 3.显示的调用toDF方法将RDD转换成DataFrame */

    //将RDD转换成DataFrame
    //导入隐式转换
    import sqlContext.implicits._
    val bdf: DataFrame = personRDD.toDF

    /* 4.注册临时表 */

    //变成DF后就可以使用两种API进行编程了
    //把DataFrame先注册临时表
    bdf.registerTempTable("t_person")

    /* 5.执行SQL（Transformation，lazy）*/

    //书写SQL（SQL方法应其实是Transformation）
    val result: DataFrame = sqlContext.sql("SELECT * FROM t_person ORDER BY fv desc, age asc")
```

```
    /* 6.执行Action */

    //查看结果（触发Action）
    result.show()
    //释放资源
    sc.stop()
  }
}

case class Person(id: Long, name: String, age: Int, fv: Double)
```

2.5.3 编程创建DataFrame

第二种方式，是通过编程接口来创建DataFrame，你可以在程序运行时动态构建一份元数据，然后将其应用到已经存在的RDD上。这种方式的代码比较冗长，但是如果在编写程序时，还不知道RDD的元数据，只有在程序运行时，才能动态得知其元数据，那么只能通过这种动态构建元数据的方式。

1.创建sparkContext，然后再创建SQLContext 2.先创建RDD，对数据进行整理，然后关联Row，将非结构化数据转换成结构化数据 3.定义schema 4.调用sqlContext的createDataFrame方法 5.注册临时表 6.执行SQL（Transformation，lazy） 7.执行Action

```
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._
import org.apache.spark.sql.{DataFrame, Row, SQLContext}
import org.apache.spark.{SparkConf, SparkContext}

object SQLDemo2 {

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

    //创建sparkSession
   val sparkSession =
SparkSession.builder().appName("SparkSQLDemo").master("local").getOrCreate()
    //创建特殊的RDD（DataFrame），就是有schema信息的RDD

    //先有一个普通的RDD，然后在关联上schema，进而转成DataFrame

    val lines = sc.textFile("c://data/person.txt")
    //将数据进行整理
    val rowRDD: RDD[Row] = lines.map(line => {
      val fields = line.split(",")
      val id = fields(0).toLong
      val name = fields(1)
      val age = fields(2).toInt
      val fv = fields(3).toDouble
      Row(id, name, age, fv)
    })


    //结果类型，其实就是表头，用于描述DataFrame
```

```
    val sch: StructType = StructType(List(
      StructField("id", LongType, true),
      StructField("name", StringType, true),
      StructField("age", IntegerType, true),
      StructField("fv", DoubleType, true)
    ))

    //将RowRDD关联schema
    val bdf: DataFrame = sparkSession.createDataFrame(rowRDD, sch)

    //变成DF后就可以使用两种API进行编程了
    //把DataFrame先注册临时表
    bdf.registerTempTable("t_person")

    //书写SQL（SQL方法应其实是Transformation）
    val result: DataFrame = sparkSession.sql("SELECT * FROM t_person ORDER BY fv desc, age asc")

    //查看结果（触发Action）
    result.show()
    sc.stop()

  }
}
```

2. 7 DSL风格，使用dataframe的API

```
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.types._
import org.apache.spark.sql.{DataFrame, Dataset, Row, SQLContext}
import org.apache.spark.{SparkConf, SparkContext}

object SQLDemo3 {

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

    //提交的这个程序可以连接到Spark集群中
    val conf = new SparkConf().setAppName("SQLDemo3").setMaster("local[2]")

    //创建SparkSQL的连接（程序执行的入口）
    val sc = new SparkContext(conf)
    //sparkContext不能创建特殊的RDD（DataFrame）
    //将SparkContext包装进而增强
    val sqlContext = new SQLContext(sc)
    //创建特殊的RDD（DataFrame），就是有schema信息的RDD

    //先有一个普通的RDD，然后在关联上schema，进而转成DataFrame

    val lines = sc.textFile("c://data/person.txt")
    //将数据进行整理
```

```
    val rowRDD: RDD[Row] = lines.map(line => {
      val fields = line.split(",")
      val id = fields(0).toLong
      val name = fields(1)
      val age = fields(2).toInt
      val fv = fields(3).toDouble
      Row(id, name, age, fv)
    })

    //结果类型，其实就是表头，用于描述DataFrame
    val sch: StructType = StructType(List(
      StructField("id", LongType, true),
      StructField("name", StringType, true),
      StructField("age", IntegerType, true),
      StructField("fv", DoubleType, true)
    ))

    //将RowRDD关联schema
    val bdf: DataFrame = sqlContext.createDataFrame(rowRDD, sch)

    //不使用SQL的方式，就不用注册临时表了
    val df1: DataFrame = bdf.select("name", "age", "fv")
    import sqlContext.implicits._
    val df2: DataFrame = df1.orderBy($"fv" desc, $"age" asc)
    df2.show()
    sc.stop()

  }
}
```

版本更换问题：

1.java.lang.NoSuchMethodError: scala.Predef$.ArrowAssoc

解决方案：

版本问题，增加scala 2.11.6，注意删除之前使用的jar包；

2.IDEA运行scala程序报错：Error:scalac: bad option: '-make:transitive'

解决方案：

1）找到你该项目的所在目录，进入这个项目根目录下；

2）执行命令：cd .idea;

3）打开scala_compiler.xml文件，将此行注释掉；

4）重启IDEA即可。

2.6 DataFrame转换成RDD

```
无论是通过那这种方式获取的DataFrame都可以使用一个方法转换
val frameToRDD: RDD[Row] = frame.rdd
frameToRDD.foreach(x => println(x.getString(0)+","+x.getInt(1)))
```

3. DataSets

3.1 DataSets概念

官方文档：

A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. A Dataset can be constructed from JVM objects and then manipulated using functional transformations (`map` , `flatMap` , `filter` , etc.). The Dataset API is available in Scala and Java. Python does not have the support for the Dataset API. But due to Python's dynamic nature, many of the benefits of the Dataset API are already available (i.e. you can access the field of a row by name naturally `row.columnName`). The case for R is similar.

源码：

DataSets类定义说明

```
/**
 * A Dataset is a strongly typed collection of domain-specific objects that can be transformed
 * in parallel using functional or relational operations. Each Dataset also has an untyped view
 * called a `DataFrame`, which is a Dataset of [[Row]].
 *
 * Operations available on Datasets are divided into transformations and actions.
Transformations
 * are the ones that produce new Datasets, and actions are the ones that trigger computation and
 * return results. Example transformations include map, filter, select, and aggregate
(`groupBy`).
 * Example actions count, show, or writing data out to file systems.
 *
 * Datasets are "lazy", i.e. computations are only triggered when an action is invoked.
Internally,
 * a Dataset represents a logical plan that describes the computation required to produce the
data.
 * When an action is invoked, Spark's query optimizer optimizes the logical plan and generates a
 * physical plan for efficient execution in a parallel and distributed manner. To explore the
 * logical plan as well as optimized physical plan, use the `explain` function.
 *
 * To efficiently support domain-specific objects, an [[Encoder]] is required. The encoder maps
 * the domain specific type `T` to Spark's internal type system. For example, given a class
`Person`
 * with two fields, `name` (string) and `age` (int), an encoder is used to tell Spark to
generate


 * code at runtime to serialize the `Person` object into a binary structure. This binary
```

```
structure
 * often has much lower memory footprint as well as are optimized for efficiency in data
processing
 * (e.g. in a columnar format). To understand the internal binary representation for data, use
the
 * `schema` function.
```

```
// Encoders for most common types are automatically provided by importing sqlContext.implicits._
val ds = Seq(1, 2, 3).toDS()
ds.map(_ + 1).collect() // Returns: Array(2, 3, 4)

// Encoders are also created for case classes.
case class Person(name: String, age: Long)
val ds = Seq(Person("Andy", 32)).toDS()

// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name.
val path = "examples/src/main/resources/people.json"
val people = sqlContext.read.json(path).as[Person]
```

3.2 DataSet转化到RDD

```
//调用rdd方法
val dataSetTORDD:RDD[People] = dataSet.rdd
dataSetTORDD.foreach(x => println(x.name+","+x.age));
```

3.3 DataSet转换DataFrame

```
//调用toDF方法,直接服用case class中定义的属性
val frame: DataFrame = dataSet.toDF()
frame.show()
```

3.4 DataFrame转换DataSet

```
 val value: Dataset[People] = frame.as[People]
 case class People(name:String,age:Int)
```

总结:

SparkSQL支持两种类型分别为DataSet和DataFrame,这两种类型都支持从RDD转换为DataSet或DataFrame

RDD转DataFrame有三种方法是

1.直接转换即使用元组的模式存储在转换

2.使用样例类的模式匹配Scheam在转换

3.StructType直接指定Schema在转换

RDD转DataSet

1.使用样例类的模式匹配Scheam在转换

ps:其余读取文件的方式可以直接获取对应的DataFrame

DataSet和DataFrame之间的互相转换

DataSet转换DataFrame

调用toDF方法,直接服用case class中定义的属性

DataFrame转换DataSet

调用as[对应样例类类名]

DataSet和DataFrame转换我RDD

DataSet对象或DataFrame对象调用rdd方法就可以转换为rdd

3.5 数据操作方法

DSL风格语法

```
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.{SparkConf, SparkContext}

object SparkSQL {
  def main(args:Array[String]):Unit = {
  //创建SparkConf()并设置App名称
    val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")
    val spark = SparkSession.builder().config(conf).getOrCreate()
    val df: DataFrame = spark.read.json("dir/people.json")
    //DSL风格语法:
    df.show()
    import spark.implicits._
    // 打印Schema信息
    df.printSchema()
    df.select("name").show()
    df.select($"name", $"age" + 1).show()
    df.filter($"age" > 21).show()
    df.groupBy("age").count().show()
    spark.stop()
  }
}
```

SQL风格语法

```
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.{SparkConf, SparkContext}

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

  //创建SparkConf()并设置App名称
```

```
    val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")
    val spark = SparkSession.builder().config(conf).getOrCreate()
    val df: DataFrame = spark.read.json("dir/people.json")
    //SQL风格语法:
    //临时表是Session范围内的，Session退出后，表就失效了
    //一个SparkSession结束后,表自动删除
    df.createOrReplaceTempView("people")
    val sqlDF = spark.sql("SELECT * FROM people")
    sqlDF.show()
    //如果想应用范围内有效，可以使用全局表。注意使用全局表时需要全路径访问，如：global_temp.people
    //应用级别内可以访问,一个SparkContext结束后,表自动删除 一个SparkContext可以多次创建SparkSession
    //使用的比较少
    df.createGlobalTempView("people")
   //创建名后需要必须添加global_temp才可以
    spark.sql("SELECT * FROM global_temp.people").show()
    spark.newSession().sql("SELECT * FROM global_temp.people").show()
    spark.stop()
  }
}
```

ps:需要打包上传集群,可以在集群中使用这个jar包

在安装spark的目录下进入到bin目录执行spark-submit

例如 :/opt/software/spark-2.2.0-bin-hadoop2.7/bin

--class 需要执行类的类全限定名(从包开始到类名结束) \

--master 指定主节点 \

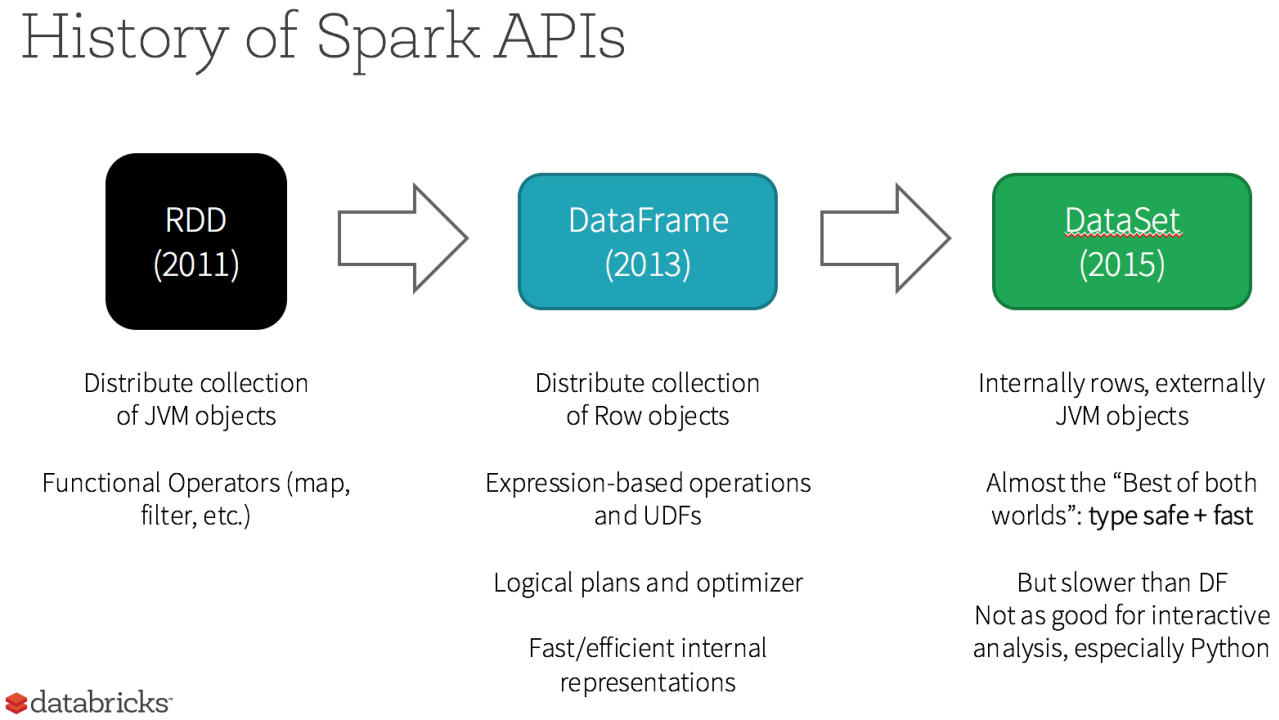
上传jar所在的位置 \

数据输入路径 \

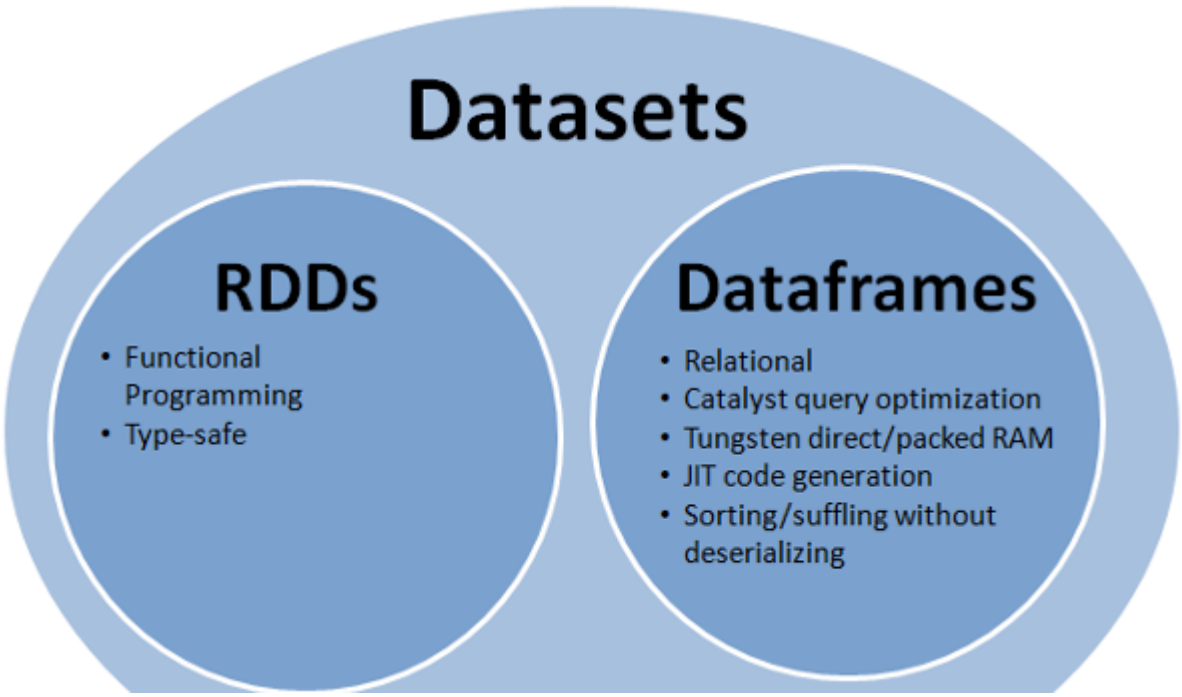
数据输出路径 --> 没有可以不写

3.6 RDD，DataFrame，DataSets

3.6.1 spark历史发展



3.6.2 三者关系



可以借助API，在 DataFrame、Dataset 、RDDs之间无缝迁移，而且DataFrame、 Dataset是建立在RDD的基础上的。

3.6.3 对比

RDD

RDD是Spark建立之初的核心API。RDD是不可变分布式弹性数据集，在Spark集群中可跨节点分区，并提供分布式low-level API来操作RDD，包括transformation和action。

何时使用RDD？

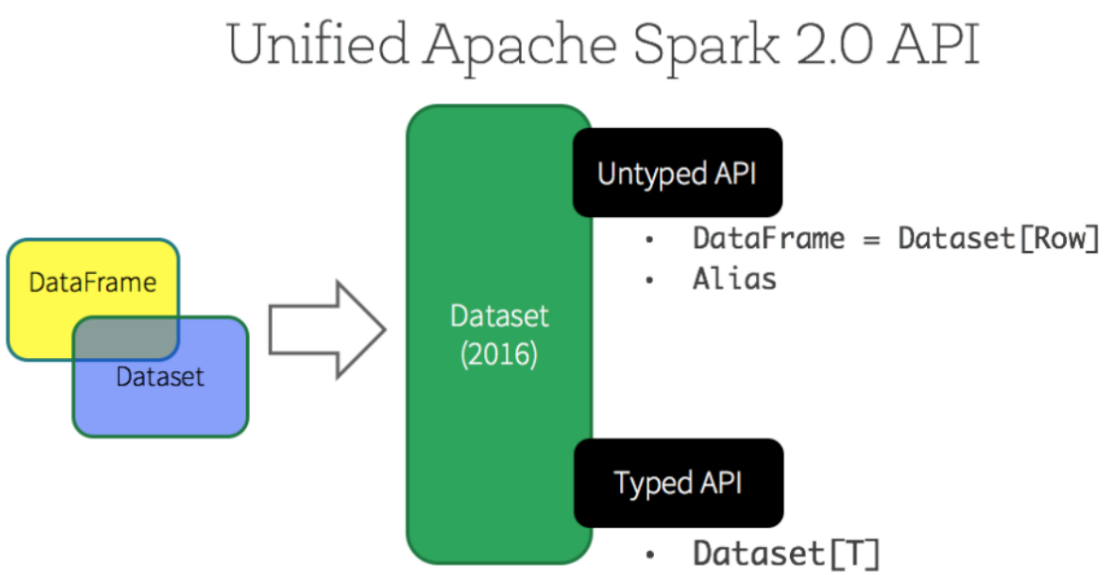
使用RDD的一般场景： 你需要使用偏底层的transformation和action来控制你的数据集； 你的数据集非结构化，比如：流媒体或者文本流； 你想使用函数式编程来操作你的数据，而不是用特定领域语言(DSL)表达； 你不想加入schema，比如，当通过名字或者列处理(或访问)数据属性不在意列式存储格式； 当你可以放弃使用DataFrame和Dataset来优化结构化和半结构化数据集的时候。

DataFrames

与RDD类似，DataFrame是不可变的分布式数据集合，与RDD不同的是，数据按列的方式组织，类似于关系型数据库的一张表。设计的初衷是使得大数据集的处理更简单， DataFrame允许用户在分布式数据集上施加一个结构，是对数据更高级的抽象，提供了具体的API处理分布式数据，同时使得Spark拥有更广泛的用户群。

DataFrame与RDD相同之处，都是不可变分布式弹性数据集。不同之处在于，DataFrame的数据集都是按指定列存储，即结构化数据，类似于传统数据库中的表。DataFrame的设计是为了让大数据处理起来更容易。DataFrame允许开发者把结构化数据集导入DataFrame，并做了更高层次的抽象；DataFrame提供特定领域的语言(DSL)API来操作你的数据集。

在Spark2.0中，DataFrame API将会和Dataset API合并，统一数据处理API。由于这个统一"有点急"，导致大部分Spark开发者对Dataset的high-level和type-safe API并没有什么概念。



Dataset

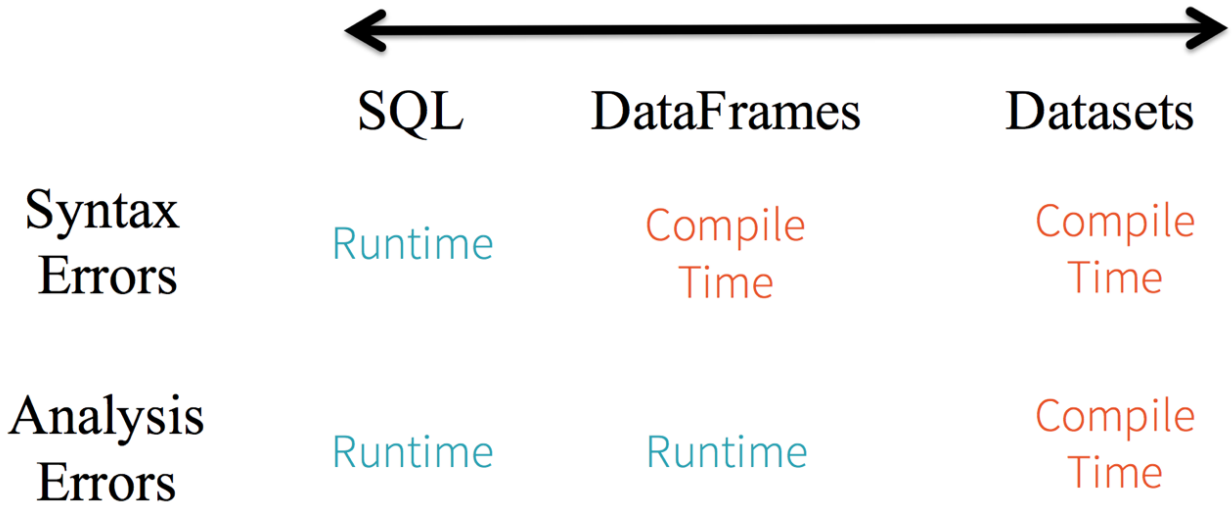
从Spark2.0开始，DataSets扮演了两种不同的角色：强类型API和弱类型API，见下表。从概念上来讲，可以把DataFrame 当作一个泛型对象的集合DataSet[Row], Row是一个弱类型JVM 对象。相对应地，如果JVM对象是通过Scala的case class或者Java class来表示的，Dataset是强类型的。

Typed and Un-typed APIs

Language	Main Abstraction
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python*	DataFrame
R*	DataFrame

Dataset API的优势 对于Spark开发者而言，你将从Spark 2.0的DataFrame和Dataset统一的API获得以下好处：

1，静态类型和运行时类型安全 考虑静态类型和运行时类型安全，SQL有很少的限制而Dataset限制很多。例如，Spark SQL查询语句，你直到运行时才能发现语法错误(syntax error)，代价较大。然后DataFrame和Dataset在编译时就可捕捉到错误，节约开发时间和成本。 Dataset API都是lambda函数和JVM typed object，任何typed-parameters不匹配即会在编译阶段报错。因此使用Dataset节约开发时间。



2，High-level抽象以及结构化和半结构化数据集的自定义视图 DataFrame是Dataset[Row]的集合，把结构化数据集视图转换成半结构化数据集。例如，有个海量IoT设备事件数据集，用JSON格式表示。JSON是一个半结构化数据格式，很适合使用Dataset, 转成强类型的Dataset[DeviceIoTData]。

```
{"device_id": 198164, "device_name": "sensor-pad-198164owomcJZ", "ip": "80.55.20.25", "cca2":
"PL", "cca3": "POL", "cn": "Poland", "latitude": 53.080000, "longitude": 18.620000, "scale":
"Celsius", "temp": 21, "humidity": 65, "battery_level": 8, "c02_level": 1408, "lcd": "red",
"timestamp" :1458081226051}
```

使用Scala为JSON数据DeviceIoTData定义case class。

```
case class DeviceIoTData (battery_level: Long, c02_level: Long, cca2: String, cca3: String, cn:
String, device_id: Long, device_name: String, humidity: Long, ip: String, latitude: Double, lcd:
String, longitude: Double, scale:String, temp: Long, timestamp: Long)
```

紧接着，从JSON文件读取数据

```
// read the json file and create the dataset from the
// case class DeviceIoTData
// ds is now a collection of JVM Scala objects DeviceIoTData
val ds = spark.read.json("/databricks-public-
datasets/data/iot/iot_devices.json").as[DeviceIoTData]
```

上面代码运行时底层会发生下面3件事。 Spark读取JSON文件，推断出其schema，创建一个DataFrame； Spark把数据集转换DataFrame -> Dataset[Row]，泛型Row object，因为这时还不知道其确切类型； Spark进行转换：Dataset[Row] -> Dataset[DeviceIoTData]，DeviceIoTData类的Scala JVM object 我们的大多数人，在操作结构化数据时，都习惯于以列的方式查看处理数据列，或者访问对象的指定列。Dataset 是Dataset[ElementType]类型对象的集合，既可以编译时类型安全，也可以为强类型的JVM对象定义视图。从上面代码获取到的数据可以很简单的展示出来，或者用高层方法处理。

```
> //display Dataset's
  display(ds)
```

▸ (1) Spark Jobs

battery_level	c02_level	cca2	cca3	cn	device_id	device_name	humidity	ip	latitude	lcd	longitude	scale	temp	timestamp
8	868	US	USA	United States	1	meter-gauge-1xbYRYcj	51	68.161.225.1	38	green	-97	Celsius	34	1458444054093
7	1473	NO	NOR	Norway	2	sensor-pad-2n2Pea	70	213.161.254.1	62.47	red	6.15	Celsius	11	1458444054119
2	1556	IT	ITA	Italy	3	device-mac-36TWSKiT	44	88.36.5.1	42.83	red	12.83	Celsius	19	1458444054120
6	1080	US	USA	United States	4	sensor-pad-4mzWkz	32	66.39.173.154	44.06	yellow	-121.32	Celsius	28	1458444054121
4	931	PH	PHL	Philippines	5	therm-stick-5gimpUrBB	62	203.82.41.9	14.58	green	120.97	Celsius	25	1458444054122
3	1210	US	USA	United States	6	sensor-pad-6al7RTAobR	51	204.116.105.67	35.93	yellow	-85.46	Celsius	27	1458444054122
3	1129	CN	CHN	China	7	meter-gauge-7GeDoanM	26	220.173.179.1	22.82	yellow	108.32	Celsius	18	1458444054123
0	1536	JP	JPN	Japan	8	sensor-pad-8xUD6pzsQI	35	210.173.177.1	35.69	red	139.69	Celsius	27	1458444054123
3	807	JP	JPN	Japan	9	device-mac-9GcjZ2pw	85	118.23.68.227	35.69	green	139.69	Celsius	13	1458444054124

Showing the first 1000 rows.

Command took 0.20s

3，简单易用的API 虽然结构化数据会给Spark程序操作数据集带来挺多限制，但它却引进了丰富的语义和易用的特定领域语言。大部分计算可以被Dataset的high-level API所支持。例如，简单的操作agg，select，avg，map，filter或者groupBy即可访问DeviceIoTData类型的Dataset。 使用特定领域语言API进行计算是非常简单的。例如，使用filter()和map()创建另一个Dataset。 把计算过程翻译成领域API比RDD的关系代数式表达式要容易的多。例如：

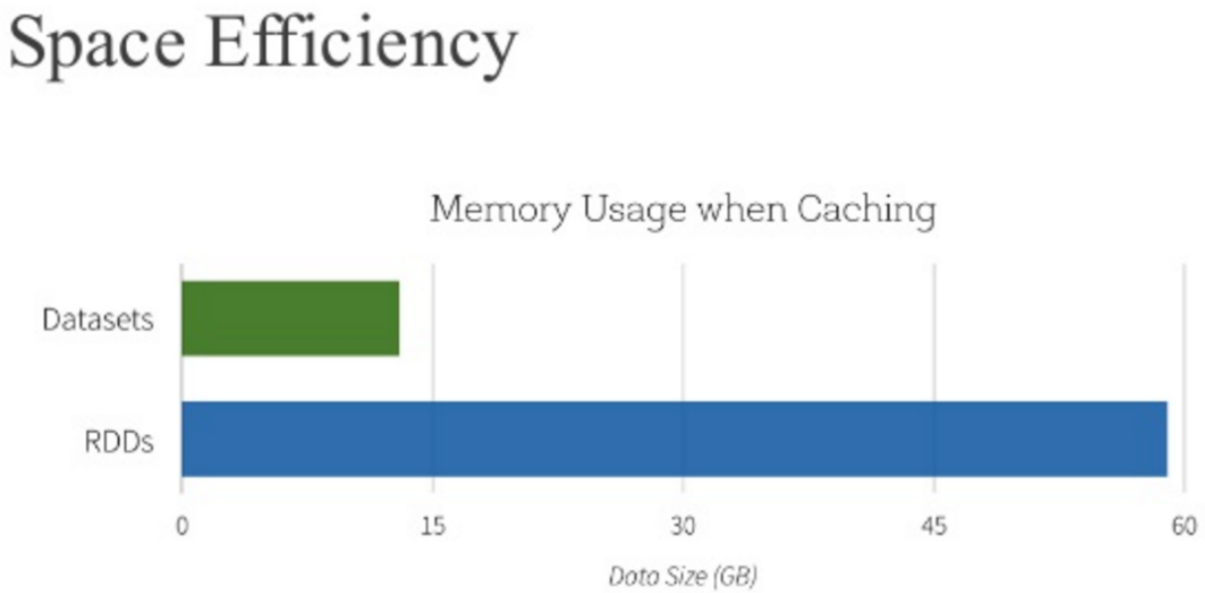
```
// Use filter(), map(), groupBy() country, and compute avg()
// for temperatures and humidity. This operation results in
// another immutable Dataset. The query is simpler to read,
// and expressive

val dsAvgTmp = ds.filter(d => {d.temp > 25}).map(d => (d.temp, d.humidity,
d.cca3)).groupBy($"_3").avg()

//display the resulting dataset
display(dsAvgTmp)
```

4，性能和优化 使用DataFrame和Dataset API获得空间效率和性能优化的两个原因： 首先：因为DataFrame和Dataset是在Spark SQL 引擎上构建的，它会使用Catalyst优化器来生成优化过的逻辑计划和物理查询计划。 R，Java，Scala或者Python的DataFrame/Dataset API，所有的关系型的查询都运行在相同的代码优化器下，代码优化器带来的的是空间和速度的提升。不同的是Dataset[T]强类型API优化数据引擎任务，而弱类型API DataFrame在

交互式分析场景上更快，更合适。



其次，Dataset能使用Encoder映射特定类型的JVM 对象到Tungsten内部内存表示。Tungsten的Encoder可以有效的序列化/反序列化JVM object，生成字节码来提高执行速度。

什么时候使用DataFrame或者Dataset？

- 你想使用丰富的语义，更高层次的抽象，和特定领域语言API，使用DataFrame或者Dataset；
- 你处理的半结构化数据集需要high-level表达，filters，maps，aggregation，average，sum，SQL查询，列式访问和使用lambda函数，那你可以使用DataFrame或者Dataset；
- 想利用编译时严格的type-safety，Catalyst优化和高效的Tungsten的code生成，那你可以使用DataFrame或者Dataset；
- 你想统一和简化API使用跨Spark的Library，那你可以使用DataFrame或者Dataset；
- 如果你是一个R使用者，那你可以使用DataFrame或者Dataset；
- 如果你是一个Python使用者，那你可以使用DataFrame或者Dataset。

你可以无缝地把DataFrame或者Dataset转化成一个RDD，只需简单的调用.rdd：

```
// select specific fields from the Dataset, apply a predicate
// using the where() method, convert to an RDD, and show first 10
// RDD rows
val deviceEventsDS = ds.select($"device_name", $"cca3", $"c02_level").where($"c02_level" > 1300)
// convert to RDDs and take the first 10 rows
val eventsRDD = deviceEventsDS.rdd.take(10)
```

RDD适合需要low-level函数式编程和操作数据集的情况；DataFrame和Dataset适合结构化数据集，使用high-level和特定领域语言(DSL)编程，空间效率高和速度快。

4. 案例练习

sparkSQL版Wordcount（略）

5.读取JDBC数据源

步骤：

```
import java.util.Properties

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

object JdbcDataSource {

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

    val spark = SparkSession.builder().appName("JdbcDataSource")
      .master("local[*]")
      .getOrCreate()

    import spark.implicits._

    //load这个方法会读取真正mysql的数据吗？
    val logs: DataFrame = spark.read.format("jdbc").options(
      Map("url" -> "jdbc:mysql://localhost:3306/bigdata",
        "driver" -> "com.mysql.jdbc.Driver",
        "dbtable" -> "logs",
        "user" -> "root",
        "password" -> "123568")
    ).load()

    //logs.printSchema()


    //logs.show()

//    val filtered: Dataset[Row] = logs.filter(r => {
//      r.getAs[Int]("age") <= 13
//    })
//    filtered.show()

    //lambda表达式
    val r = logs.filter($"age" <= 13)

    //val r = logs.where($"age" <= 13)

    val reslut: DataFrame = r.select($"id", $"name", $"age" * 10 as "age")

    //val props = new Properties()
    //props.put("user","root")
    //props.put("password","123568")
    //reslut.write.mode("ignore").jdbc("jdbc:mysql://localhost:3306/bigdata", "logs1", props)

    //DataFrame保存成text时出错(只能保存一列)
```

```
    //reslut.write.parquet("hdfs://node-4:9000/parquet")


    //reslut.show()

    spark.close()


  }
}
```

6. 读取Json数据源

步骤：

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

object JsonDataSource {

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

    val spark = SparkSession.builder().appName("JdbcDataSource")
      .master("local[*]")
      .getOrCreate()

    import spark.implicits._

    //指定以后读取json类型的数据(有表头)
    val jsons: DataFrame = spark.read.json("/Users/Desktop/json")

    val filtered: DataFrame = jsons.where($"age" <=500)


    filtered.printSchema()

    filtered.show()

    spark.stop()


  }
}
```

7. 读取csv数据源

步骤：

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

object CsvDataSource {

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

    val spark = SparkSession.builder().appName("CsvDataSource")
      .master("local[*]")
      .getOrCreate()

    //指定以后读取json类型的数据
    val csv: DataFrame = spark.read.csv("c://data//t.scv")

    csv.printSchema()

    val pdf: DataFrame = csv.toDF("id", "name", "age")

    pdf.show()

    spark.stop()


  }
}
```

8. 读取parquet数据源

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

object ParquetDataSource {

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

    val spark = SparkSession.builder().appName("ParquetDataSource")
      .master("local[*]")
      .getOrCreate()

    //指定以后读取json类型的数据
    val parquetLine: DataFrame = spark.read.parquet("c://data//t.parquet")
    //val parquetLine: DataFrame = spark.read.format("parquet").load("")

    parquetLine.printSchema()


    //show是Action
```

```
    parquetLine.show()

    spark.stop()


  }
}
```

9.Spark Shell获取MySQL中的数据

1、启动Spark Shell，必须指定mysql连接驱动jar包

```
/usr/local/spark-1.6.1-bin-hadoop2.6/bin/spark-shell\
--master spark://node01:7077\
--jars/usr/local/spark-1.6.1-bin-hadoop2.6/mysql-connector-java-5.1.35-bin.jar \
--driver-class-path/usr/local/spark-1.6.1-bin-hadoop2.6/mysql-connector-java-5.1.35-bin.jar
```

2、从mysql中加载数据

```
val jdbcDF =sqlContext.read.format("jdbc")
    .options(Map("url" ->"jdbc:mysql://node03:3306/bigdata", "driver" ->"com.mysql.jdbc.Driver",
"dbtable" ->"person", "user" -> "root","password" -> "root"))
    .load()
```

3、执行查询

```
jdbcDF.show()
```

10. 将数据写入MySQL中并打包上传任务

```
object InsertData2MySQL {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("InsertData2MySQL").setMaster("local")
    val sc = new SparkContext(conf)
    val sqlContext = new SQLContext(sc)

    // 获取数据
    val linesRDD = sc.textFile("hdfs://node01:9000/test.txt").map(_.split(","))

    // 指定schema信息
    val schema = StructType{
      Array(
        StructField("name", StringType, true),
        StructField("age", IntegerType, true))
    }

    // linesRDD映射到Row
```

```
    val personRDD = linesRDD.map(p => Row(p(0), p(1).toInt))

    // 转换成DataFrame
    val personDF = sqlContext.createDataFrame(personRDD, schema)

    // 封装请求MySQL的配置信息
    val prop = new Properties()
    prop.put("user", "root")
    prop.put("password", "root")
    prop.put("driver", "com.mysql.jdbc.Driver")

    // 把数据写入MySQL
    personDF.write.mode("append").jdbc("jdbc:mysql://node03:3306/bigdata", "person", prop)

    sc.stop()
  }
}
```

用maven将程序打包

将Jar包提交到spark集群

```
/usr/local/spark-1.6.1-bin-hadoop2.6/bin/spark-submit \

--class com.qf.spark.sql.JdbcRDD \

--master spark://node01:7077 \

--jars /usr/local/spark-1.6.1bin-hadoop2.6/mysql-connector-java-5.1.35-bin.jar \

--driver-class-path /usr/local/spark-1.6.1-bin-hadoop2.6/mysql-connector-java-5.1.35-bin.jar \

/root/spark-mvn-1.0-SNAPSHOT.jar
```

11 . Spark sql API 介绍

(略)