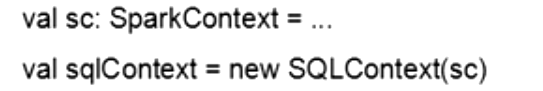
# 1 Spark SQL

## SQLContext

SQLContext是Spark SQL的入口，SQLContext由SparkContext创建。

（注意：SQLContext只支持sql方言，不支持其他hivesql之类的方言）

图1



### 1.1.1 SparkContext，SQLContext，SparkSession的区别

SparkContext是Spark 1.\*中的Spark SQL的入口，必须先创建SparkContext。再用SparkContext创建SQLContext。一个SparkContext对象表示对计算集群的一个连接。

（注意：appName是Spark Web界面展示的Spark应用的名字，一般和类名一样就可以了）

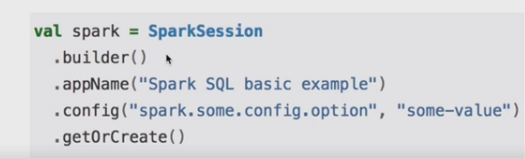
（注意：master是该Spark应用需要连接的Spark集群的master节点的地址，local表示本地运行，如果不写master，默认是local）

图1（先创建SparkContext，再用SparkContext创建SQLContext）



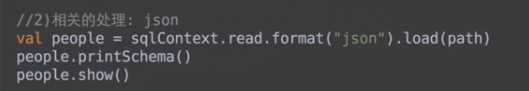
SparkSession是Spark2.\*中的入口。

图2



### 1.1.2 SQLContext读取JSON文件

图1



### 1.1.3 SQLContext和HiveContext的区别

SQLContext中去除了对Hive的支持，主要是为了某些用户引用了Hive的全部依赖后导致依赖冲突而设计的。

如果使用Hive，推荐使用HiveContext；如果不使用Hive，推荐使用SQLContext，因为它不会导致太多的依赖冲突。

## 1.1.4 不同类型的SparkContext

scala开发使用原生的SparkContext；

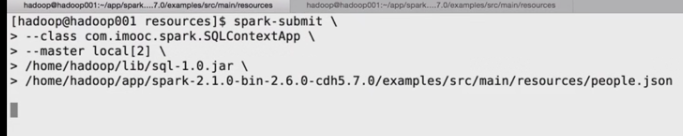
Java开发使用JavaSparkContext；

如果开发Spark SQL程序，使用SQLContext，HiveContext；

如果开发Spark Streaming程序，使用Streaming独有的SparkContext。

## 1.2 服务器上运行Spark SQL的jar包

图1



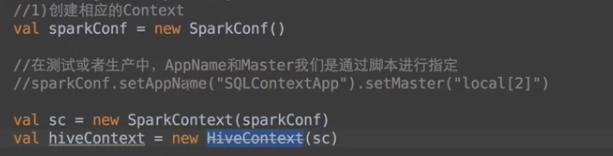
## 1.3 HiveContext

HiveContext是SQLContext的子类。

（注意：使用HiveContext是不需要搭建Hive环境的）

（注意：SQLContext支持的数据源，HiveContext都支持，HiveContext默认的方言是hivesql，可以选择其他方言，比如：sql）

图1



### 1.3.1 HiveContext显示table的数据

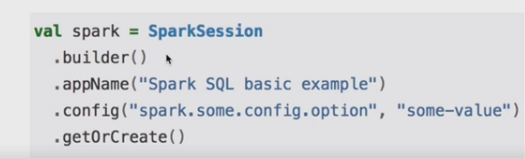
图1



## 1.4 SparkSession

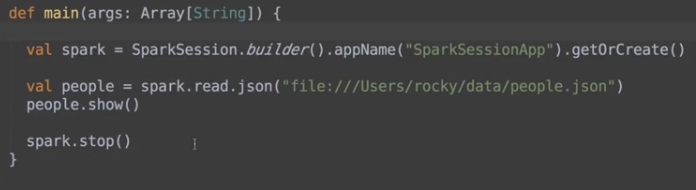
SparkSession是Spark2.\*中的入口。

图1



### 1.4.1 SparkSession读取JSON文件

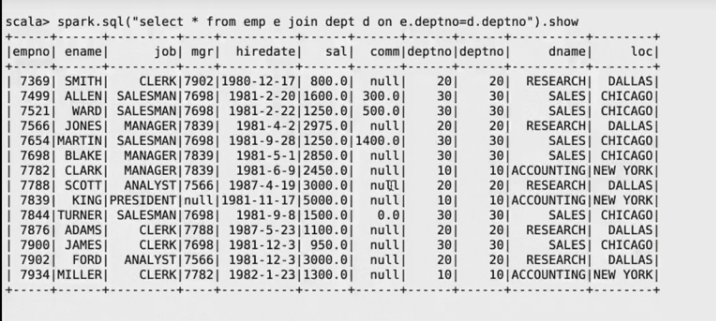
图1



## 1.5 Spark SQL相比Hive的速度

Spark SQL比Hive SQL的执行速度快很多。

图1



## 1.6 Spark thriftserver

启动thriftserver，然后通过客户端（beeline/code）去连接该thriftserver。

图1



图2



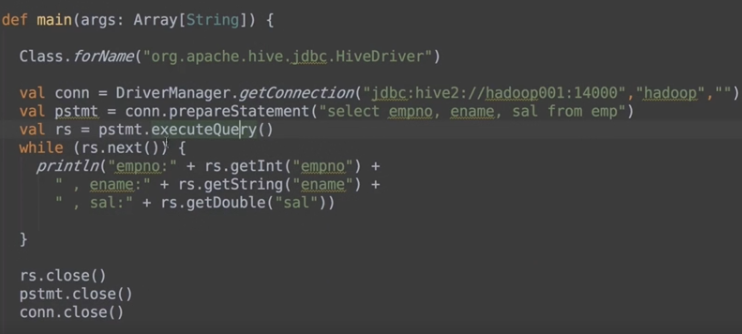
### 1.6.1 thriftserver和spark-shell/spark-sql的区别

1.每一次启动spark-shell或spark-sql都会启动一个Spark Application。

2.启动thriftserver时会启动一个Spark Application，但后面无论多少个客户端连接，永远都是一个Spark Application。

### 1.6.2 JDBC访问thriftserver

图1



## 1.7 DataFrame和DataSet

DataSet是一个分布式数据集，DataFrame是一个以列（列名，列类型，列值）的形式构成的分布式数据集。

（注意：DataFrame类似于关系型数据库的表）

图1



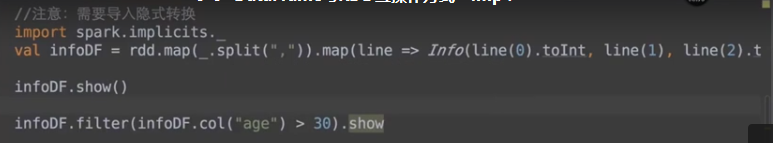
图2（DataFrame的基本API）



### 1.7.1 反射的方式根据RDD获取DataFrame

这种方式必须预先知道schema。

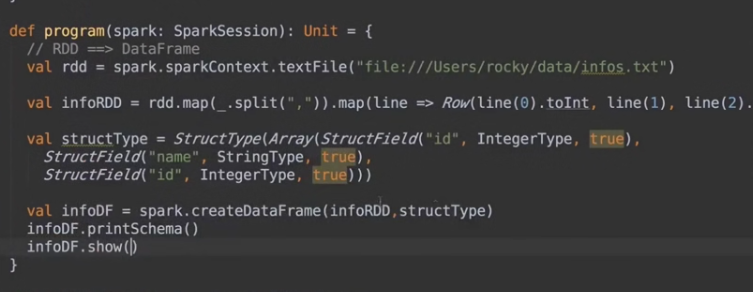
图1



### 1.7.2 structType的方式根据RDD获取DataFrame

这种方式不需要预先知道schema。

图1



### 1.7.3 创建DataFrame

#### 1.7.3.1 read().json() 从JSON文件创建

（注意：在1.3.0版本用的是jsonFile()，在1.5.1用的是read().json()）

（注意：这个json文件不是文件全部内容是一个json结构，而是每一个元素（每一行）是一个json结构）

图1（每一个元素是一个json结构）

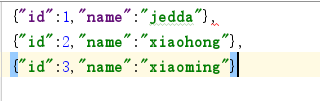
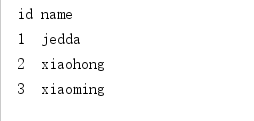


图2



图3



#### 1.7.3.2 sql() 从sql语句创建

用sqlContext的sql()从sql语句创建DataFrame。

图1

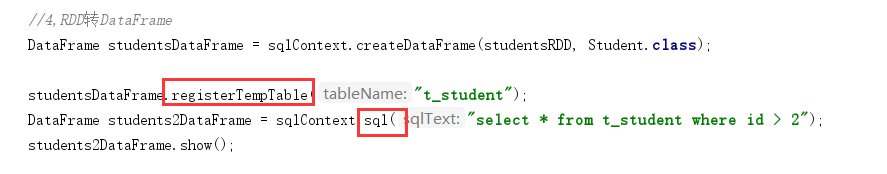
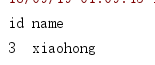


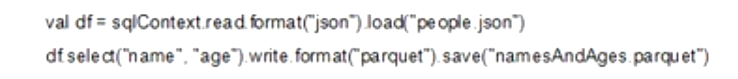
图2



#### 1.7.3.3 read().load() 从外部文件创建

load()可以读取各种类型的文件，不仅仅是json文件（比如：parquet列存储文件）。

图1



#### 1.7.3.4 parquetFile() 从parquet文件创建

图1

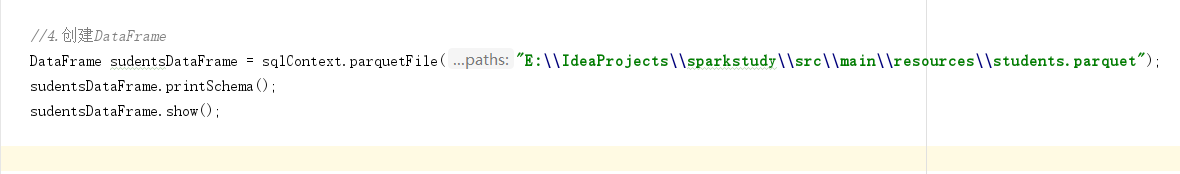
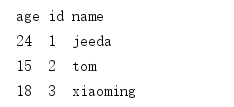
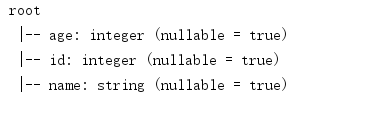


图2



### 1.7.4 DataFrame常用操作

#### 1.7.4.1 show

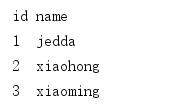
show()打印DataFrame中所有的数据。

（注意：show()类似于select \* from ...）

图1



图2



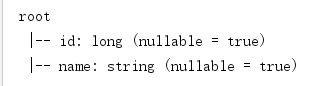
#### 1.7.4.2 printSchema

printSchema()打印DataFrame的元数据（Schema）。

图1



图2



#### 1.7.4.3 select(“xxx”).show()

select(“xxx”).show()查询某列所有的数据。

（注意：类似于select xxx from ...）

图1



图2

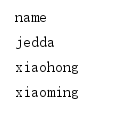
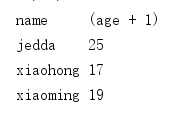


图3



图4



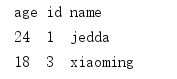
#### 1.7.4.4 filter

过滤。

图1



图2



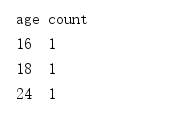
#### 1.7.4.5 groupBy

groupBy根据某一列分组，然后可以进行聚合操作。

图1



图2



#### 1.7.4.6 registerTemplate

registerTemplate将DataFrame注册到一个临时表，后面就可以用sql对这个临时表进行操作。

图1

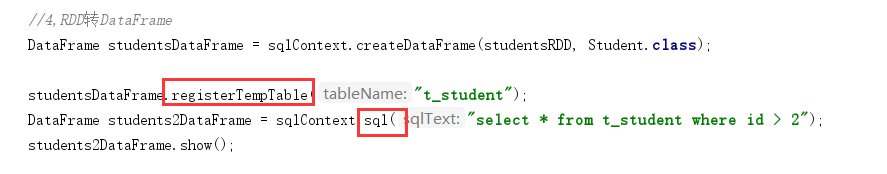


图2

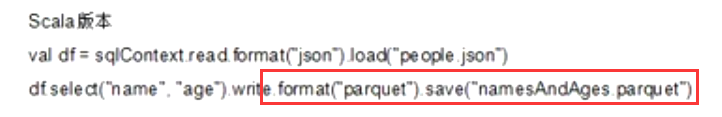


#### 1.7.4.7 write().save()

write().save() 将DataFrame保存到外部文件（可以是json文件，也可以是parquet列存储文件）。

（注意：写进去的是一个目录，目录下有几个文件）

图1



#### 1.7.4.8 saveAsParqueFile()

saveAsParqueFile()将DataFrame保存到parque文件。

（注意：写进去的是一个目录，目录下有几个文件）

图1

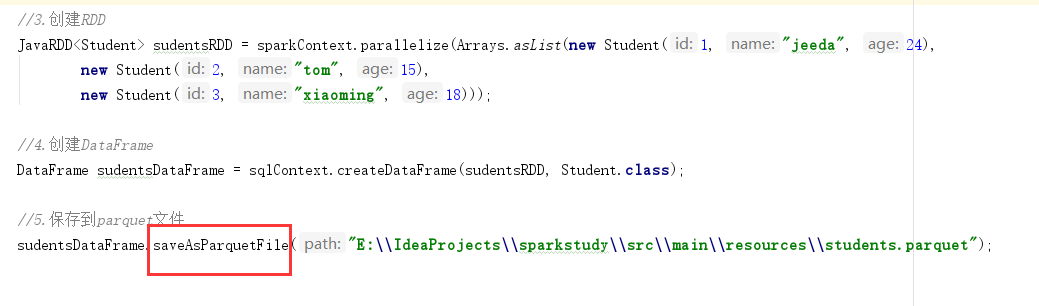
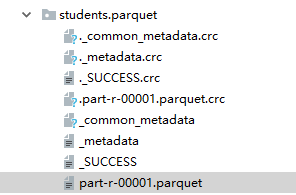


图2



### 1.7.5 RDD转DataFrame

#### 1.7.5.1 反射方式

方式1：通过反射的方式将RDD转换成DataFrame，缺点是RDD一个元素中各个字段的数据类型在整个程序运行中都是固定的，不能根据不同的条件（比如：实时从MySQL读取字段类型的时候才知道字段类型是什么）而改变。

（注意：Java版本支持将含每一个元素是简单类型（不包含List等复杂数据类型，不包含嵌套Java Bean）的JavaBean的RDD转换成DataFrame）

（注意：Java版本的JavaBean必须是public类型）

（注意：Scala版本支持将每一个元素是case class的RDD（允许包含Array，允许包含嵌套case class）转换成DataFrame）

（注意：Scala版本使用反射的方式，将RDD转换成DataFrame时，需要手动导入隐式转换）

（注意：反射方式可以从RDD -> DataFrame，也可以从DataFrame -> RDD ）

Java版本：

图1（RDD -> DataFrame）

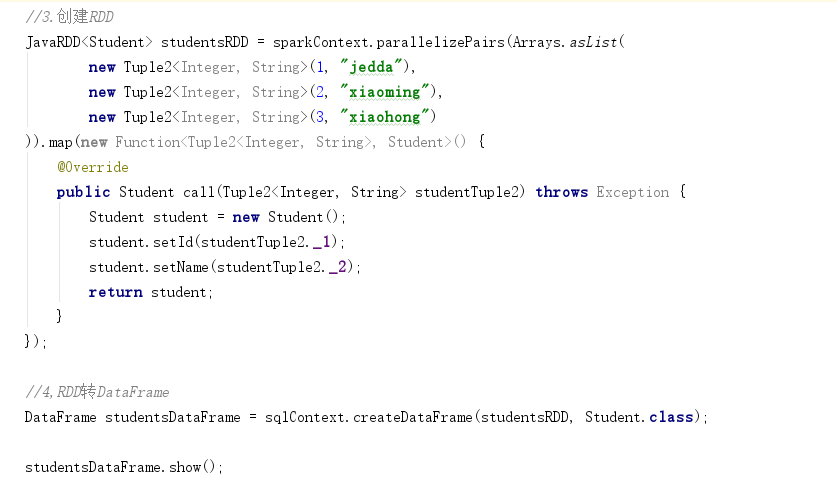


图2

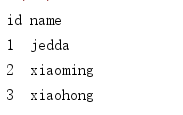


图3（DataFrame -> RDD）

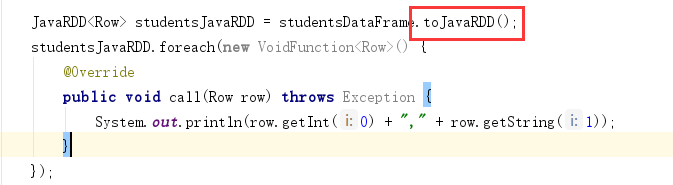
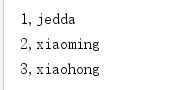


图4



Scala版本：

图1（RDD -> DataFrame）

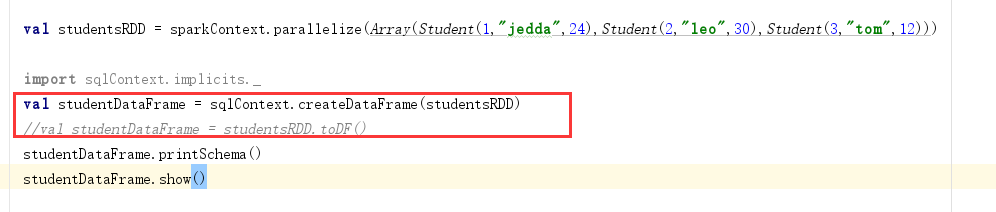


图2

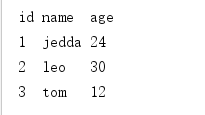


图3（DataFrame -> RDD）

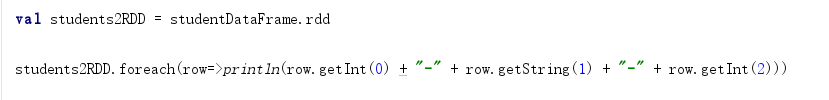
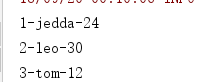


图4



#### 1.7.5.2 动态指定元数据

动态指定元数据来将RDD转换成DataFrame的方式，优点式RDD一个元素中各个字段的类型可以不固定，可以根据不同的条件（比如：实时从MySQL读取字段类型的时候才知道字段类型是什么）而改变。。

（注意：首先需要将普通的RDD转换成RDD[Row]的类型）

Java版本：

图1（RDD -> DataFrame）

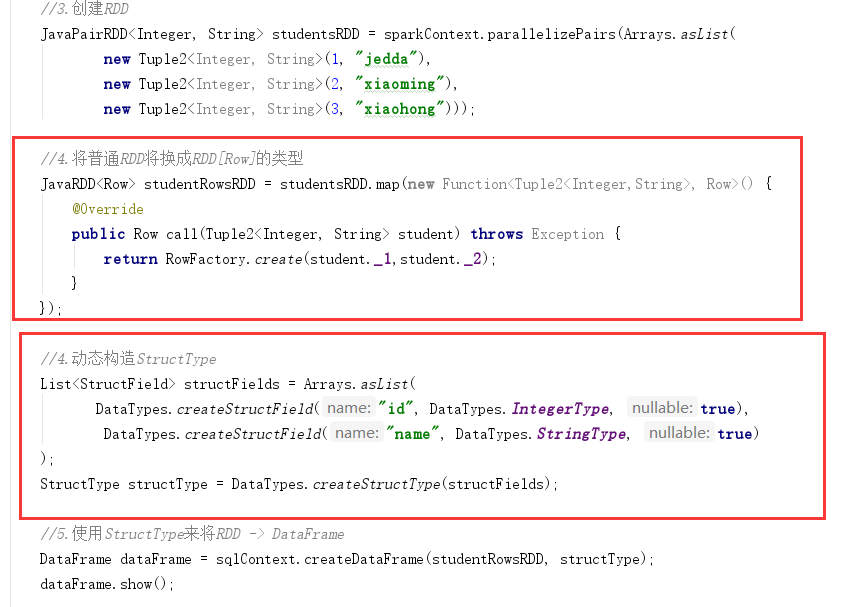
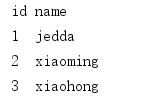


图2



Scala版本：

图1

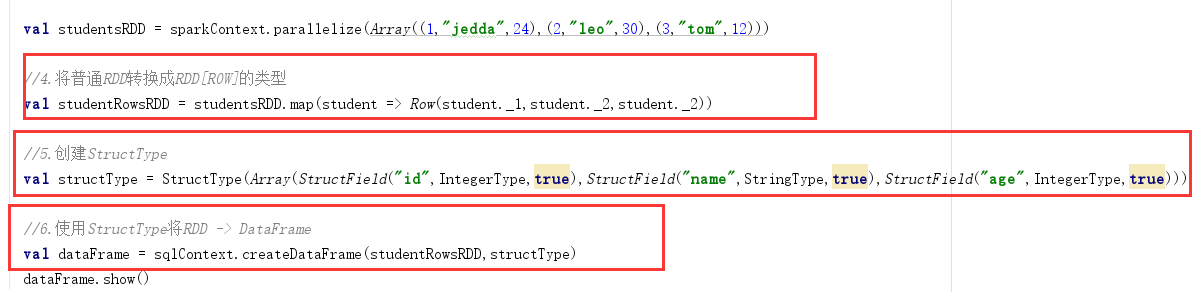
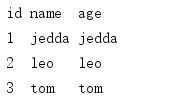


图2



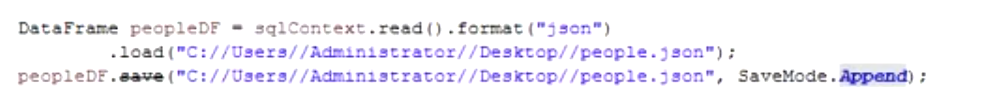
### 1.7.6 saveMode

DataFrame写入文件（save）时的几种模式，默认模式下，如果目标位置已存在数据，则会抛出异常。

图1



图2



### 1.7.7 Parquet数据源自动推断分区列

因为 Parquet数据源是一个目录，该目录如果分区后，每个分区列是一个分区子目录（如图1）。

Spark SQL可以根据分区目录自动推断出分区列的列名和列数据类型。

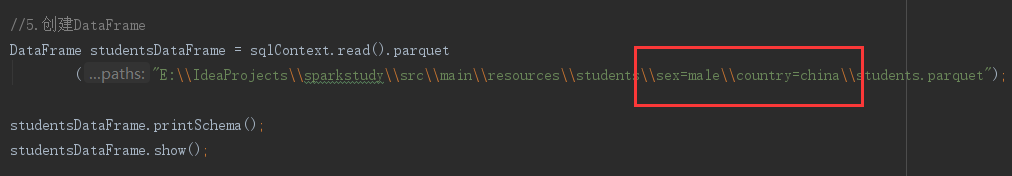
（注意：目前只支持自动推断的列数据类型包括数字和字符串类型）

（注意：如果用户不希望Spark SQL自动推断分区列类型，可以禁止，禁止后分区列类型统一是String）

图1（每个分区列是一个分区子目录）



图2（在本地是没有分区列一说的，所以本地文件会推断失败，但在hdfs上是有分区列的）



### 1.7.8 同一个Parquet目录下不同的元数据合并

在Spark1.5.0以后的版本，默认是关闭Parquet元数据合并的，可以通过图1的2种方式开启。

（注意：合并时，如果没有的列值会是null）

（注意：2个元数据出现列相同时，不会进行值的合并）

图1

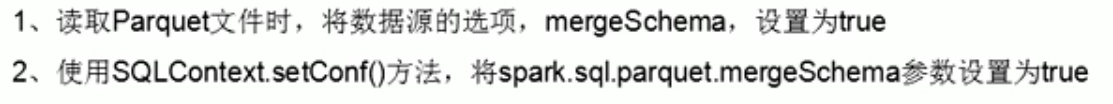


图2

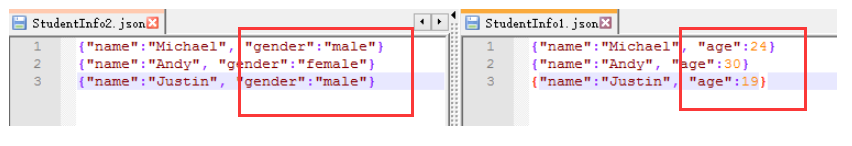


图3

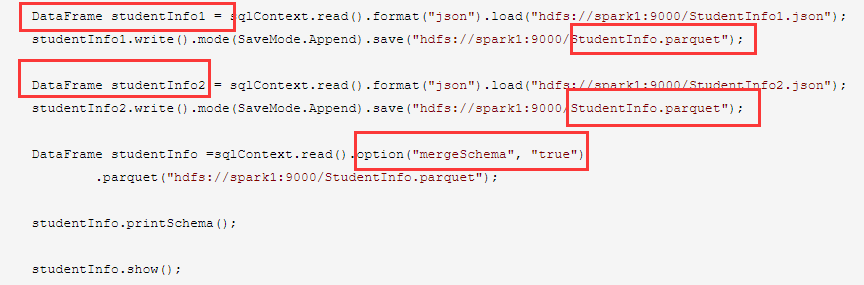
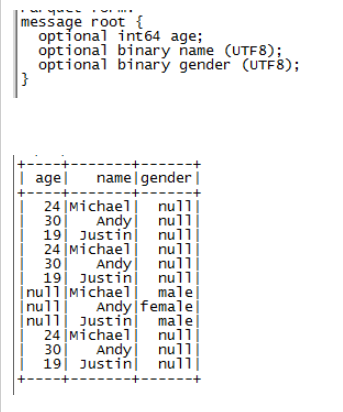


图4



### 1.7.9 2个DataFrame的数据关联查询

将2个DataFrame分别注册到2个临时表，然后用SQLContext的sql方法用sql查询这2个临时表，得到一个关联后的新的DataFrame。

图1

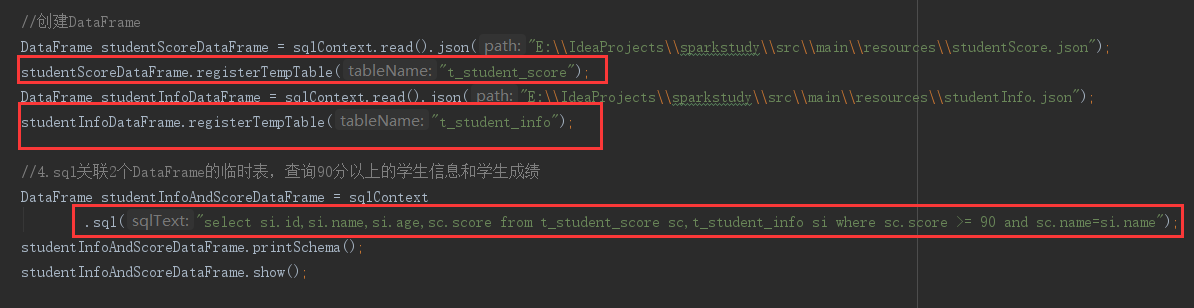
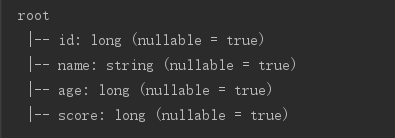
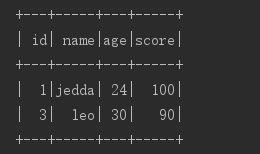


图2





### 1.7.10 DataFrame对Hive的操作

#### 1.7.10.1 saveAsTable

saveAsTable将DataFrame的数据物化到Hive表中，还会在Hive元数据库（比如MySQL）创建表的元数据。Spark程序重启后，物化的Hive表依旧存在。

（注意：registerTemplate创建的只是临时表，不会物化。Spark程序重启后，临时表会被删除）

## 1.8 SchemaRDD

读取数据和执行查询都会返回SchemaRDD，SchemaRDD就是每一个元素都是ROW对象的RDD。SchemaRDD可能在Spark1.3版本以后更名为DataFrame。

图1



### 1.8.1 Row对象

Row对象表示SchemaRDD中的一条记录。

图1



## 1.9 Parquet

Parquet是列式存储格式。

### 1.9.1 列式存储相比行式存储的优点

1.列式存储只加载需要的列，不需要的列可以不加载，降低IO数据量。

2.因为一列的所有数据的数据类型是相同的，所以列式存储可以使用压缩编码来节省存储空间。

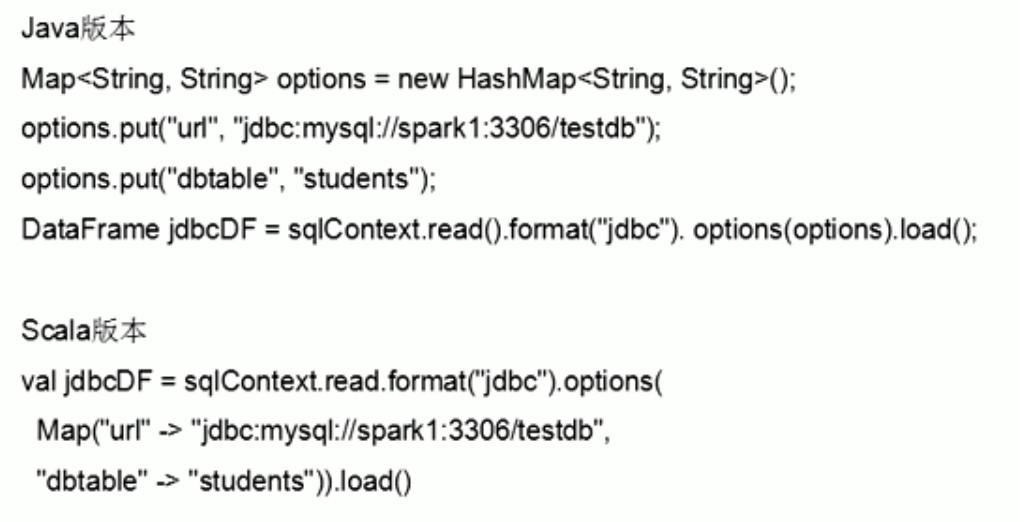
3.列式存储支持向量运算，提高运算性能。

## 1.10 Spark SQL处理关系型数据库的数据

### 1.10.1 read().format(“jdbc”).option(xxx).load()

read().format(“jdbc”).option(xxx).load()从MySQL数据库的表创建DataFrame。

图1



### 1.10.2 write().format(“jdbc”).option(xxx).save()

write().format(“jdbc”).option(xxx).save()将DataFrame的数据写入到MySQL的表中。

## 1.11 DataFrame内置函数

### 1.11.1 聚合函数

#### 1.11.1.1 countDistinct

countDistinct是去重后，再统计个数。

（注意：agg只是聚合的语义，没有分组的语义）

（注意：aggr的第一个参数，必须是groupBy中出现过的字段，并且在aggr中以单引号为前缀（不需要以单引号为后缀）；aggr的第二个参数是内置函数，内置函数的参数也以单引号为前缀）

图1

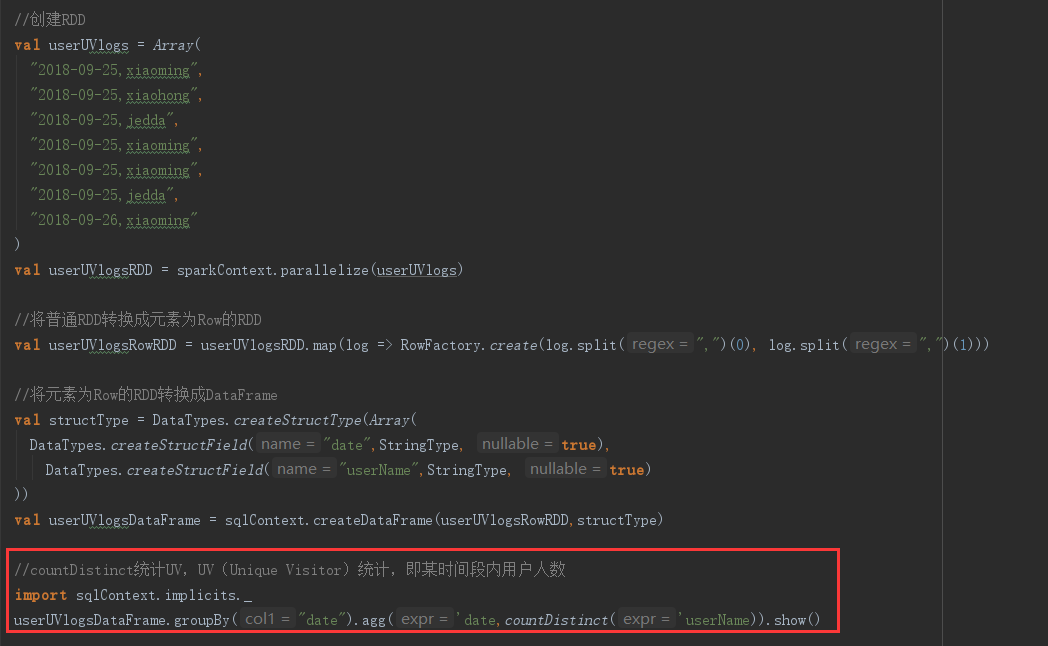


图2

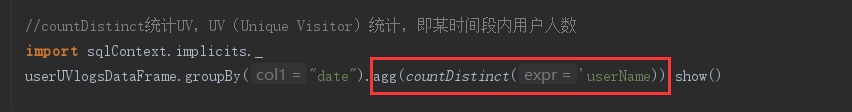
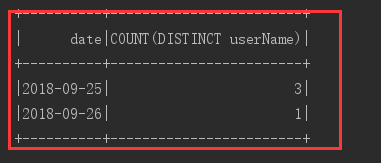


图3



#### 1.11.1.2 sum

sum是求和。

图1

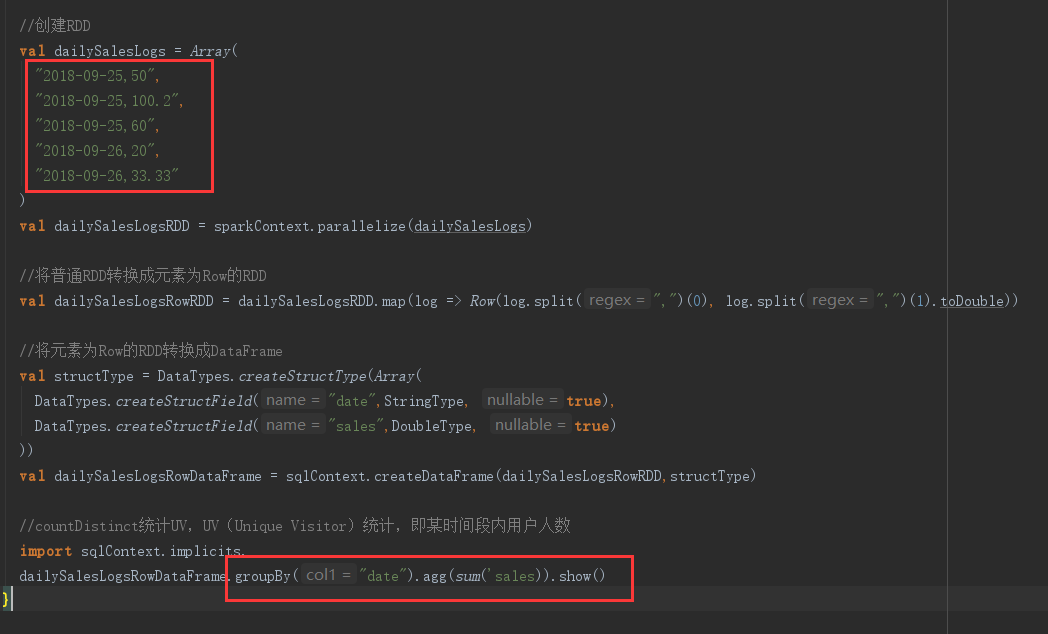


图2



### 1.11.2 窗口（开窗）函数

#### 1.11.2.1 row\_number

row\_number()开窗函数就是对每一行赋予一个行号，行号从1开始。

具体用法：

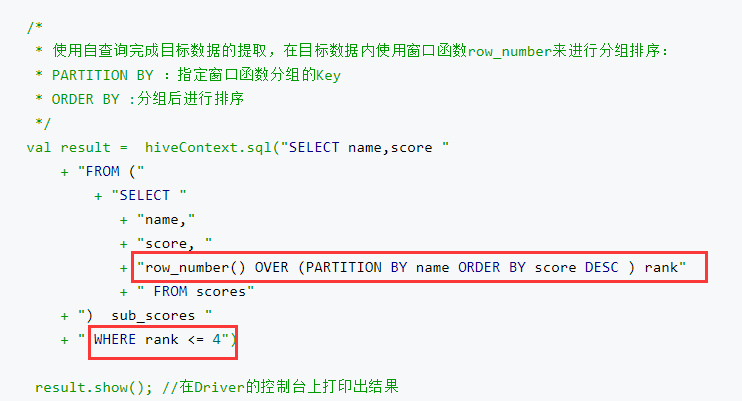
1. 在select子句中可以使用row\_number()开窗函数
2. row\_number()后面紧跟over关键字
3. over后面跟括号，括号里面是partition关键字，partition表示根据哪个字段分组；order by关键字表示组内根据哪个字段排序
4. 组内排序后，row\_number()会为组内每一行赋予一个行号，从1开始

（注意：每组的行号都会重新从1开始）

（注意：之所以外面再套一层select，是因为row\_number()是在select子句中的，而where的执行顺序在select之前，所以如果外面不套一层，是无法先执行select中的row\_number()再执行where的）

（注意：这里要使用HiveContext，因为SQLContext是没有row\_num()这个开窗函数的）

图1



## 1.12 UDF(User Defined Function)用户自定义函数

使用SQLContext的udf的register函数注册自定义函数，第一个参数是函数名，第二个参数是匿名函数。

图1

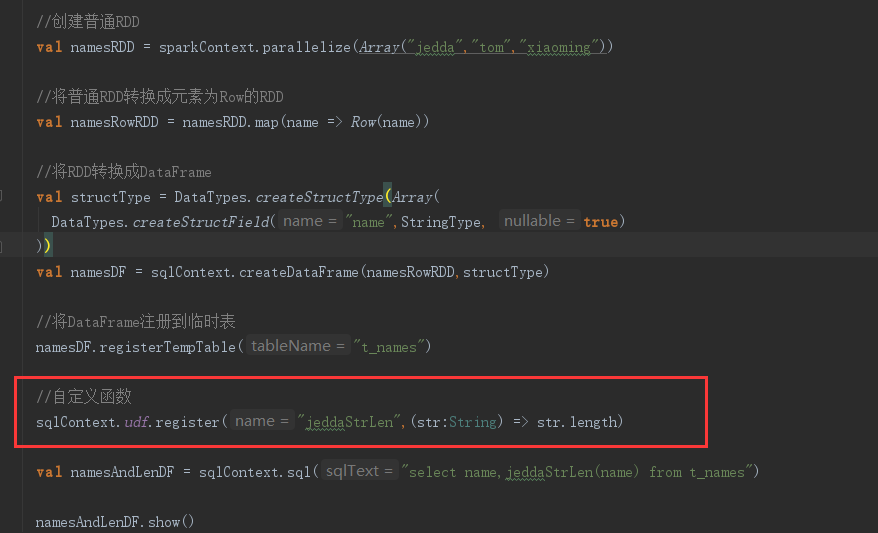
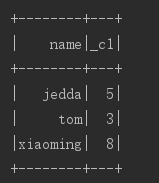


图2



### 1.12.1 UDAF（User Defined Aggregate Function）用户自定义聚合函数

UDF一般是对一行输入，产生一行输出。

UDAF一般是对多行输入，进行聚合操作，然后产生一行输出。

自定义的UDAF继承UserDefinedAggregateFunction，并实现它里面的方法。

# 2 Spark Streaming

Spark Streaming是一个实时流处理框架。

Spark Streaming将一段时间的数据封装成一个batch，然后将这个batch交给Spark Engine去处理这一批次的数据。

图1



## 2.1 实时流处理

### 2.1.1 为什么需要实时流处理

比如对于需求：统计慕课网主站上每个课程的地域信息（由IP可以得到地域信息）。

对于1小时，10分钟，1分钟级别需求或许都可以采用离线计算（mapreduce）的方式，但对于秒级别的需求只能用实时流处理，因为离线计算可能光启动进程就已经超过1秒了。

### 2.1.2 什么时候需要进行实时流处理

1.时效性高：比如，秒级别的需求

2.数据量大

### 2.1.3 实时流处理的特点

1.实时：时效性高

2.流式：数据连绵不断（像滔滔江水连绵不绝），7\*24小时连续运作

### 2.1.4 实时流计算和离线计算的区别

数据来源：

1. 离线计算的数据来源于HDFS上的历史数据，数据量很大。
2. 实时流计算的数据来源于消息队列（比如kafka），数据量一般是实时新增/修改的某一笔增量数据，增量数据很小。

处理过程：

1. 离线计算用mapreduce处理数据。
2. Spark实时流计算用Spark（DStream/SS）处理数据。

处理速度：

1. 离线慢，实时块。

进程：

1. 离线计算map进程和reduce进程用完销毁。
2. Spark Streaming 7\*24小时不停运作。

### 2.1.5 流行的实时流框架

1.Apache Storm：完全实时，来一笔数据就计算。

2.Apache Spark Streaming：Spark Streaming是Spark的一个子模块，不是完全实时，定时间（比如每2分钟，每5分钟）处理N笔数据。

3.IBM Stream。

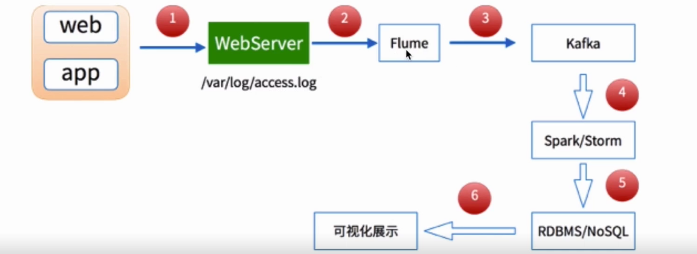
4.Kafka Streaming。

5.Yahoo! S4。

6.Apache Flink。

### 2.1.6 实时流架构

图1



1. 前端的请求到达web服务器（Web Server或Ngnix）后，会产生日志/var/log/access.log。
2. 通过分布式日志收集框架Flume收集web服务器的日志/var/log/access.log。
3. Flume收集日志后，丢到Kafka中。

（注意：其实Flume可以直接丢进Spark/Storm，但高峰期如果直接丢进Spark/Storm可以会导致Spark/Storm崩掉，所以Kakfa起到了缓冲的作用）

1. Spark/Storm处理完数据后，将结果写到关系型数据库（RDBMS）或NoSQL数据库中。
2. 通过可视化展示来观察统计数据是否有问题。

### 2.1.7 Storm和Spark Streaming的比较

1.Storm完全实时，来一条数据，处理一条数据；Spark Streaming准实时，将一个时间段的数据收集起来，这一时间段（这一批次）的数据变成一个RDD，再处理。

2.Storm毫秒级；Spark Streaming秒级，如果一秒的数据要处理5秒（如果RDD的处理比较复杂），那就达到5秒。

3.Storm吞吐量低；Spark Streaming吞吐量（单位时间内处理的数据量）高。

4.Storm支持完善的事务机制；Spark Streaming支持的事务不够完善。

5.Storm健壮性和容错性很强，基于Zookeeper和Acker；Spark Streaming健壮性和容错性一般，基于CheckPoint，WAL。

6.Storm支持动态调整并行度（到中午高峰可以临时从100并行度调高到1000并行度）；Spark Streaming不支持动态调整并行度。

7.Storm要完成复杂功能要自己编码实现；Spark Streaming提供了很多强大的算子。

（注意：总的来说，Spark Streaming只有吞吐量比Storm高，以及提供了强大的算子；但是，Spark Streaming能够和Spark Core，Spark SQL无缝整合，这是一个很大的优点。）

## 2.2 Spark提交任务

### 2.2.1 spark-submit

spark-submit需要指定jar包，适合生产。

### 2.2.2 spark-shell

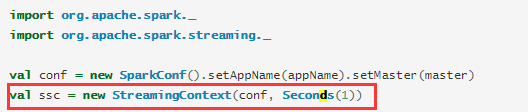
spark-shell可以直接写scala语句，适合开发和测试。

## 2.3 SparkContext

SparkContext是Spark上下文，一个SparkContext对象表示对计算集群的一个连接。

Seconds：是批处理间隔，单位是秒。

图1



## 2.4 StreamingContext

StreamingContext是Spark Streaming上下文，由SparkContext创建。

（注意：有些重载方法可以不创建SparkContext，直接由SparkConf创建，具体使用时看源码）

（注意：第一个参数是一个batch interval，表示每隔多长时间收集数据划分成一个batch（一个RDD））

（注意：一个JVM同时只能有一个StreamingContext启动）

（注意：StreamingContext.stop()在停止StreamingContext的同时，默认会把它里面的SparkContext也停止掉。如果不希望停止SparkContext，即后续可能还用到SQLContext等，则可以用stop(false)，表示停止StreamingContext的同时，不停止SparkContext）

（注意：一个SparkContext可以创建多个StreamingContext，但必须是上一个StreamingContext先stop(false)后，才能创建下一个StreamingContext）

图1

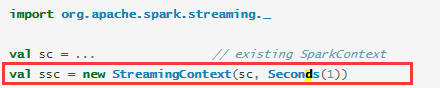
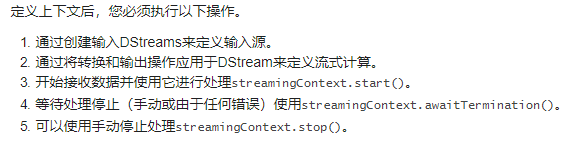


图2（创建StreamingContext后进行的操作）



## 2.5 DStream

DStream（Discretized Stream，离散的数据流）表示连续的数据流。DStream由连续的RDD构成，每个RDD是特定时间间隔的数据集合。

（注意：其实DStream就是全部实时数据，一个RDD就是一个时间段的实时数据（或者说是一个batch的实时数据））

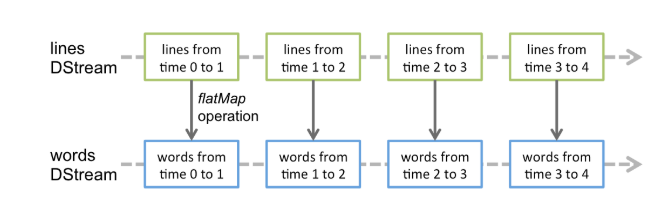
（注意：DStream的任何操作都会转换成底层RDD上的操作）

（注意：同一时间可以创建多个DStream，来并行接收多个数据源的数据。比如：一个DStream接收socket数据源的数据，另一个DStream接收Kafka数据源的数据）

图1



图2



### 2.5.1 DStream的变换（transform）

INPUT DStream可以进行变换，从而修改成不一样的INPUT DStream。DStream的变换操作分为无状态和有状态两种。

无状态：每个批次的处理不依赖之前批次的数据。

有状态：每个批次的处理依赖之前批次的数据。

（注意：虽然看上去变换操作是作用在DStream上，但实际上是作用在每个时间批次的RDD上，所以很多RDD的变换函数可以直接用在DStream上（但不是全部，如果想重用Spark RDD的所有操作，则使用DStream的foreachRDD方法））

（注意：DStream1经过transform操作后，会变成DStream2；DStream1的RDD1也会变成DStream2中的新的RDD1）

图1

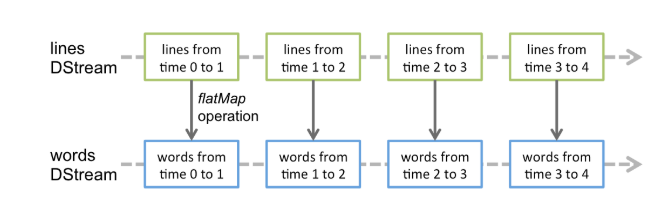
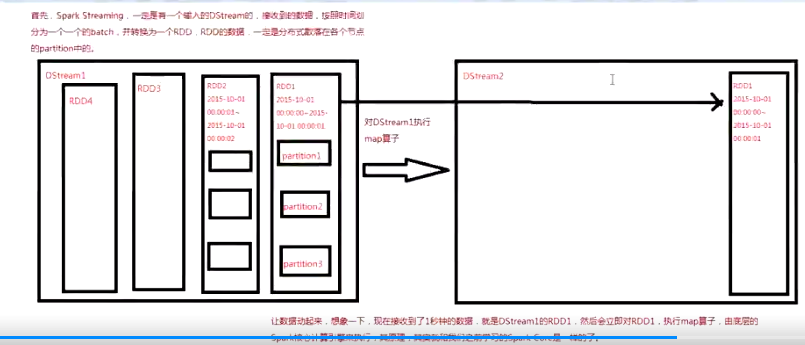


图2



#### 2.5.1.1 map

map：将DStream（所有行）的RDD（一个batch interval的某几行）的每个元素传递给函数func来返回一个新的DStream。

（换句话说，map是一个元素转换成一个元素）

图1



#### 2.5.1.2 flatmap

flatmap：与map类型，但DStream的RDD的每个元素传递给函数func来返回0个或多个输出项。

（换句话说，flatmap是一个元素转换成多个元素）

图1



#### 2.5.1.3 updateStateByKey

当我们需要跨批次维护状态时，需要用到有状态的updateStateByKey，比如：统计每个单词实时出现的总次数。

updateStateByKey：调用该方法的DStream是以(K,V)出现的，让每个key维护一个state，并指定一个更新函数取更新该key的state。

1. 首先，要定义一个state，可以是任意的数据类型。
2. 其次，要定义一个state更新函数，这个函数表示如何根据“旧state和新值”来更新“新state”。

（注意：如果state更新函数返回none，则该key对应的state就会被删除）

（注意：updateStateByKey操作，要求必须开启CheckPoint机制，以保证当内存中的state丢失后，能够从CheckPoint中恢复state）

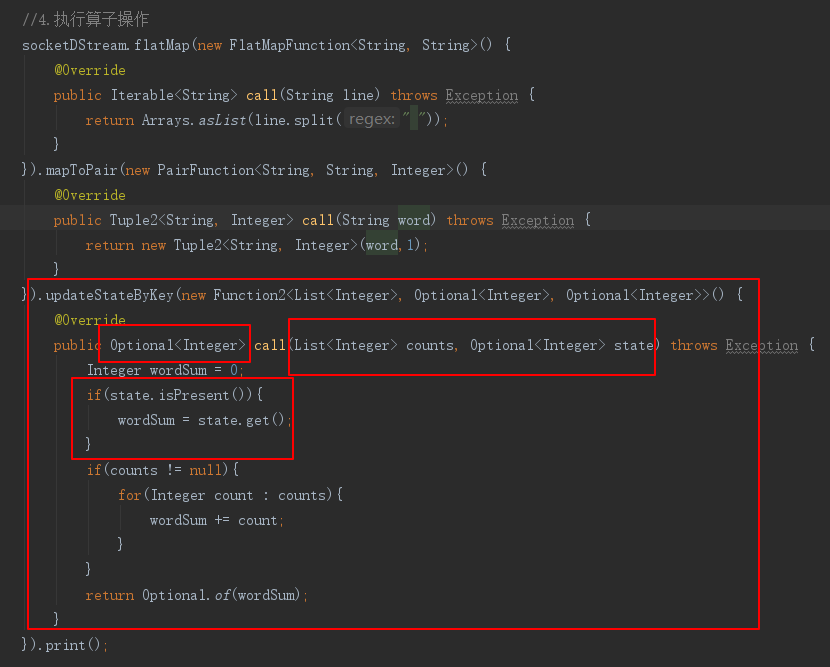
（注意：更新函数的第一个参数是该key的新的一个batch的所有value，第二个参数是该key的旧state，返回值是该key的新的state。因为state可能是不存在的，所以类型是Option）

Java版本：

图1

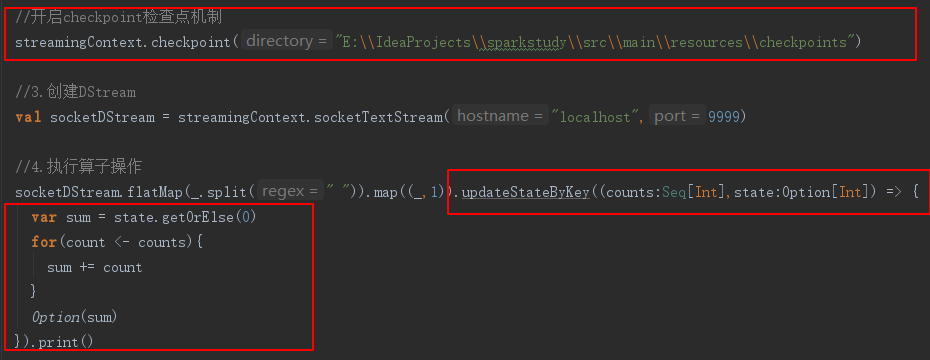


图2



Scala版本：

图3



#### 2.5.1.4 滑动窗口操作

窗口操作设计2个参数：

1. 窗口长度：窗口的持续时间
2. 滑动间隔：执行窗口操作的间隔时间

（举例：如果窗口长度为3秒，滑动间隔为2秒，batch interval为1秒，则第一次的窗口从time1到time3；第二次从time1滑动2秒（不包括time1）到time3，第二次的窗口是tim3到time5）

（每隔xxx秒统计前xxx秒的结果，第一个是滑动间隔，第二个是窗口长度）

（注意：窗口长度和滑动间隔都必须是batch interval的整数倍，否则会报错）

（注意：每次掉落在窗口内的几个RDD的数据，会被聚合到Window DStream的一个RDD）

图1

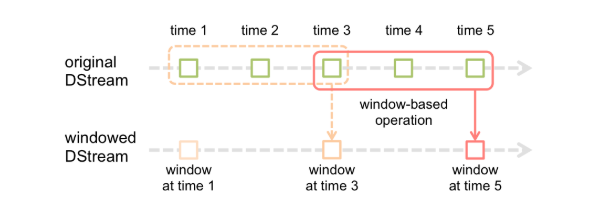


图2（window()返回一个新的DStream，这个DStream包含这个窗口内的所有RDD）



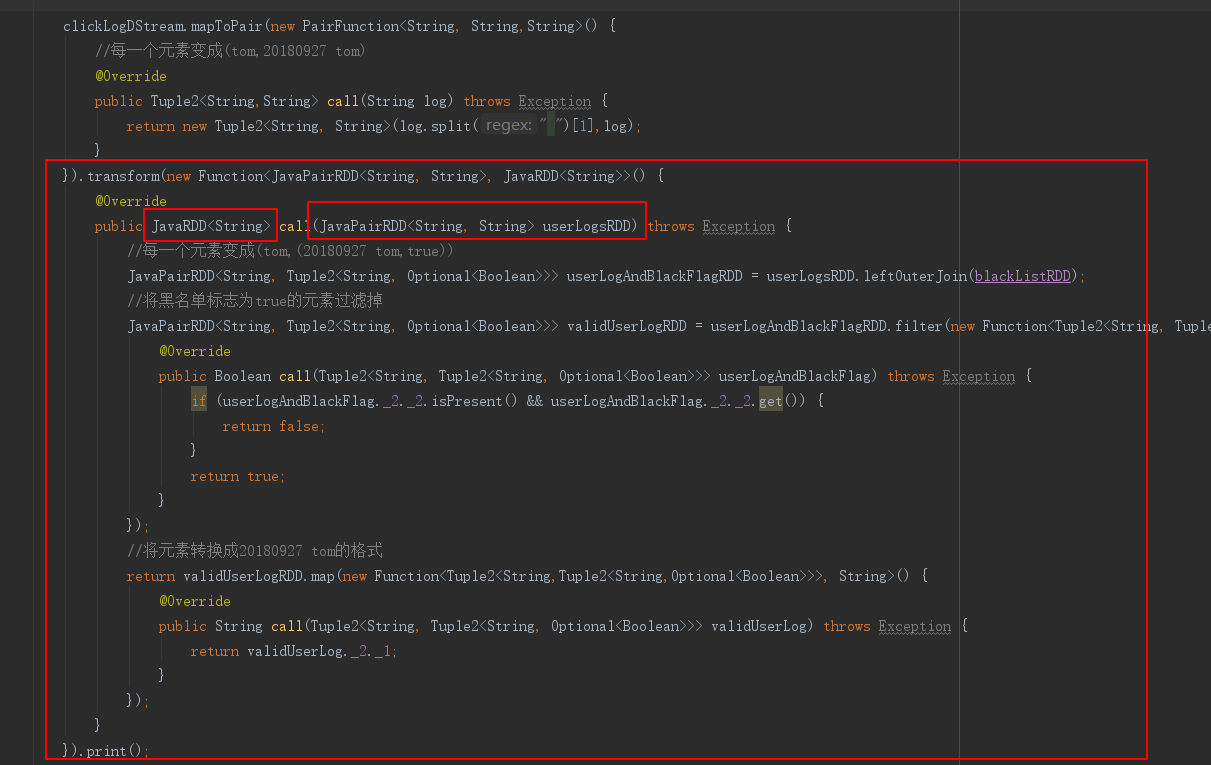
#### 2.5.1.5 transform

transform：将该DStream中的每个RDD都进行一个变换，形成一个新的RDD。

当存在一种业务场景，DStream中的每个RDD都与一个特定的RDD进行join，使用DStream的API是不行的，因为DStream只能join其他DStream，不能join RDD。这时候就要使用transform对DStream中的每个RDD进行转换了。

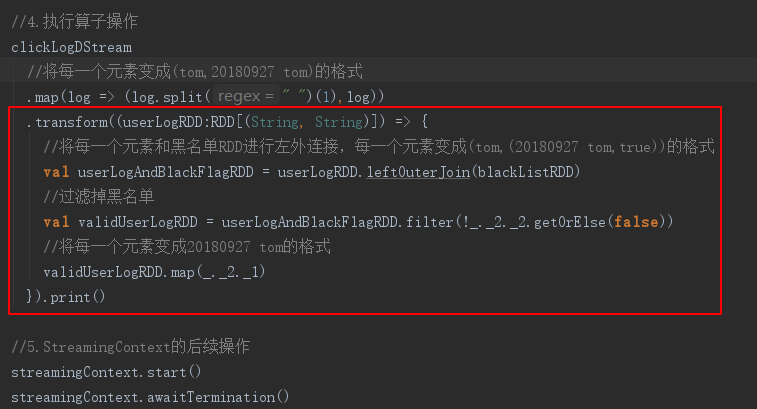
Java版本：

图1



Scala版本：

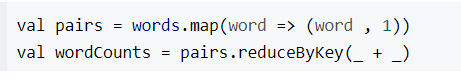
图2



#### 2.5.1.6 reduceByKey

reduceByKey：调用该方法的DStream是以(K,V)出现的，K相同则合并，返回一个(K,V)形式的DStream。K为原来的K，V根据传入的func计算得到。

图1



#### 2.5.1.7 countByValue

把这个函数应用于DStream的每个元素（注意：不是Key/Value对），对元素的值进行分组，返回(元素的值，元素的值的个数)的Key/Value对的新元素。

#### 2.5.1.8 join

#### 2.5.1.9 cogroup

### 2.5.2 DStream的输出操作

输出操作可以将DStream的数据推送到外部系统，比如：数据库或文件系统。

#### 2.5.2.1 foreachRDD

foreachRDD：对于该DStream中的每个RDD执行函数func的操作。

（注意：foreachRDD是对于该DStream中的每个RDD；foreachPartition是对于该RDD中的每个分区（因为RDD是batch interval时间间隔内的数据集，数据量可能会很大，所以RDD内部会有分区）；foreach是对于该分区内的每条记录）

图1



### 2.5.3 DStream和RDD的区别

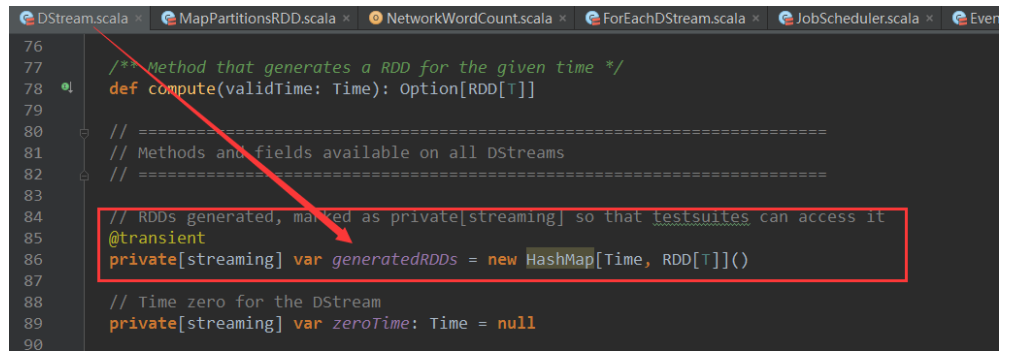
DStream由连续的RDD构成，每个RDD是特定时间间隔的数据集合。

图1



图2

（从源码角度，1个DStream中维护了一个HashMap，key是时间Time，Value是1个RDD）



### 2.5.4 DStream的分类

DStream主要分为3大类：

1. Input DStream
2. Transformed DStream
3. Output DStream

### 2.5.5 DStream的创建

#### 2.5.5.1 socketTextStream

socketTextStream()从指定IP和端口创建一个输入的DStream。

（注意：DStream的元素类型其实就是RDD的元素类型）

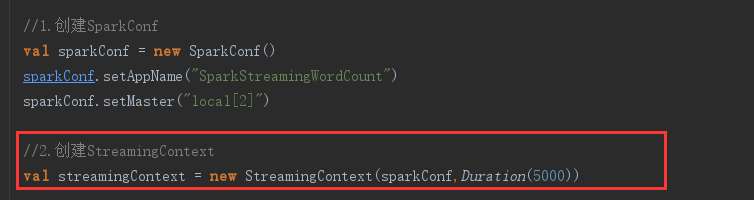
Java版本：

图1



Scala版本：

图2



#### 2.5.5.2 textFileStream

textFileStream()其实就是监控一个文件目录，只要这个文件目录有新文件出现，就实时处理。batch interval就是隔多长时间去扫描这个目录。

（注意：放到该文件目录的文件，必须有同样的格式）

（注意：放入该目录的文件，必须用移动或者重命名的方式，将文件移入目录；文件一旦处理后，就算文件内容再被修改，也不会重新被处理了）

（注意：基于文件的输入DStream是不会关联一个Receiver的，也不会占用一个CPU Core）

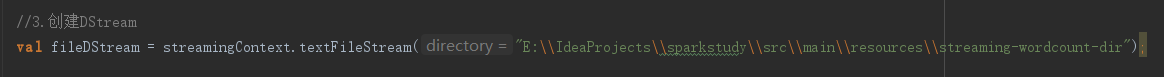
Java版本：

图1



Scala版本：

图2



#### 2.5.5.3 Kafka

##### 2.5.5.3.1 基于Receiver的方式（KafkaUtils.createStream()）

基于Receiver的方式来接收Kafka数据源的数据。这种方式使用Receiver来接收数据，Receiver是使用Kafka的高层次Consumer API来实现的，它把从Kafka获取到的数据存储到Executor的内存中，然后Spark Streaming的job会去处理这些数据。

（注意：如果该Executor内存中的数据在处理之前，该Executor所在的节点宕机了，那么数据就会丢失。这种情况下必须启用Spark Streaming的预写日志机制（Write Ahead Log），该机制会将从Kafka接收到的数据在写入内存之前，会先写入分布式文件系统（比如：HDFS）上的预写日志中。当数据丢失时，就可以从预写日志恢复数据）

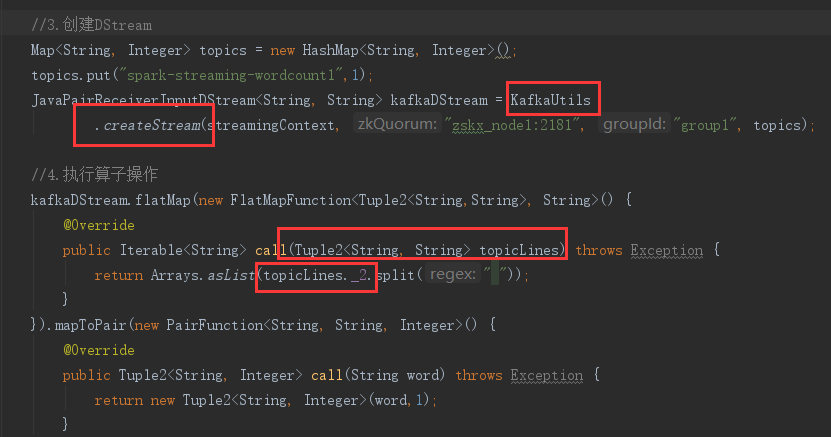
（注意：Kafka的topic的partition，和Spark的partition是没有关系的）

（注意：基于Receiver的方式，如果要同时读取一个topic下的多个partition，则一个partition需要创建一个DStream去读取）

（注意：基于Receiver的方式，因为使用Kafka的高阶Consumer API来消费数据，而Kafka的高阶Consumer API是使用Zookeeper来保存消费过的offset。因为Spark和Zookeeper之间可能是不同步的，所以无法保证数据仅被处理一次，可能数据会被重复处理2次）

Java版本：

图1



##### 2.5.5.3.2 基于Direct的方式

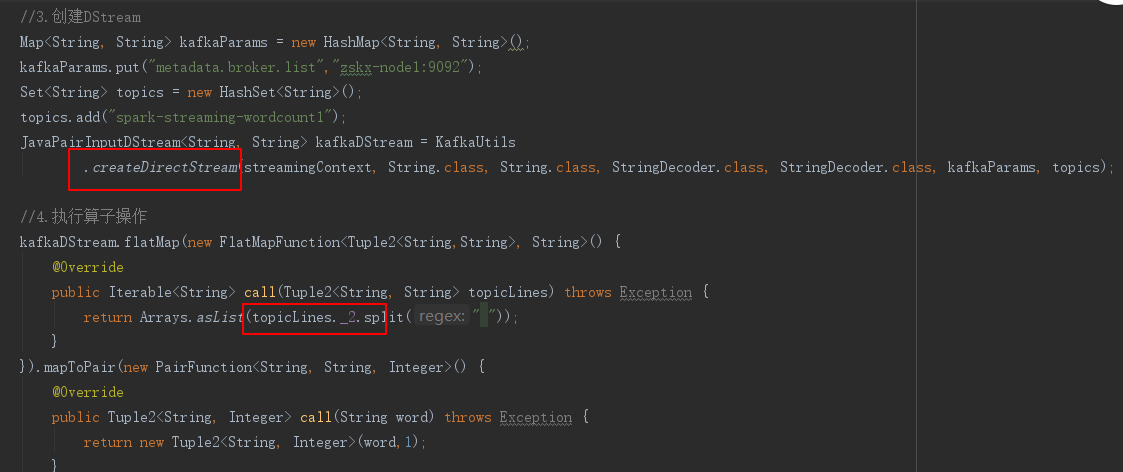
基于Direct的方式，不使用Receiver来接收Kafka的数据。这种方式会周期性去查询Kafka，获取每个topic的每个partition的最新的offset，根据这个offset来定义每个batch的数据的范围。当job启动时，使用Kafka的简单Consumer API来获取指定offset范围的数据。

（注意：基于Direct的方式，如果要同时读取一个topic下的多个partition，不需要为每一个partition创建一个DStream。Spark会为一个partition创建一个RDD partition，从而并行的从多个partition读取数据）

（注意：基于Direct的方式，不需要开启WAL来保证可靠，因为Kafka本身已经实现了可靠的机制，所以只要Kafka本身实现了数据的复制，基于Direct的方式就可以通过Kafka的副本来实现数据的恢复）

（注意：基于Direct的方式，因为Spark Streaming自己负责跟踪消费过的offset，并保存在CheckPoint中，所以可以保证数据仅被处理一次，不会重复处理2次）

图1

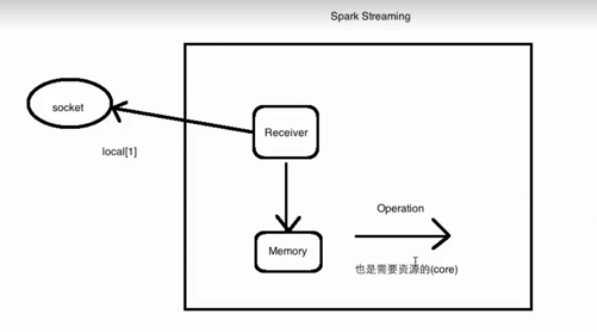


## 2.6 Receiver

每个DStream（除了文件流）都与Receiver相关联，Receiver会从源接收输入数据，并将输入数据存储在Spark的内存中，然后进行处理。

（注意：如果使用基于Receiver的INPUT DStream，不要使用local或local[1]来运行，因为这个单一的线程会被用来运行Receiver，没有剩余的线程来处理接收到的INPUT DStream，所以local[n]的n必须>Receiver的数量）

图1



### 2.6.1 两类内置的数据源支持

1.基本来源：StreamingContext API中直接提供的源。比如：文件系统和套接字连接。

2.高级来源：从额外的实用程序获取。比如：Kafka，Flume，Kinesis等。

3.自定义数据源：用户自定义的数据源。

### 2.6.2 local[2]的必要性

在使用本地模式（local）运行Spark Streaming时，不能使用local或local[1]的方式。因为一个DStream关联的Receiver至少需要1个Executor进程去接收数据，而这个Executor进程至少需要分配1个线程。

此外，还需要至少需要1个Executor进程，分配至少1个线程，去处理数据。

所以，至少要给一个Spark Application分配2个线程。

（注意：local[2]中的2表示给这个Spark Application分配多少个线程）

（注意：一个CPU有多少个核就表示有多少个CPU Core，而且普通线程数量=CPU Core数量。但如果CPU支持超线程技术，则线程数量=CPU Core数量，但一定是成正比的，即CPU Core越多，可支持的线程数量越多）

### 2.6.3 集群CPU Core和Receiver的数量关系

因为一个Receiver会长久的占用一个CPU Core去从对应的数据源接收数据，并且至少留有一个CPU Core去处理数据。

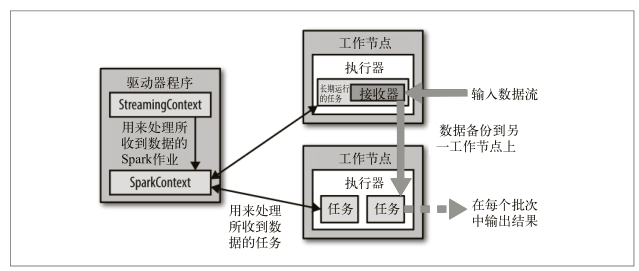
所以，集群总共的CPU Core数量必须 > 全部的Receiver的数量。

而且，分配给一个Spark Application的CPU Core总数必须 > 该Spark Application的全部的Receiver的数量。

（注意：是 > ，不是 >= ）

## 2.7 Spark Streaming组件架构

图1



1. Spark Streaming为输入数据源创建对应的接收器。
2. 接收器以长期运行的任务的方式运行在工作节点的执行器中，负责从输入数据流收集数据。
3. 接收器收集到数据后，会马上将数据复制到另一个工作节点的执行器的内存中，从而来保证容错性。

（注意：是复制，不是移动，所以数据默认情况下会有2份）

1. 驱动器程序中的StreamingContext会周期性地运行Spark作业，去处理执行器内存中缓存下来的数据。

## 2.8 检查点机制

检查点机制会阶段性地把Spark应用数据存储到如HDFS这样的可靠的存储系统中，以便恢复故障时使用。

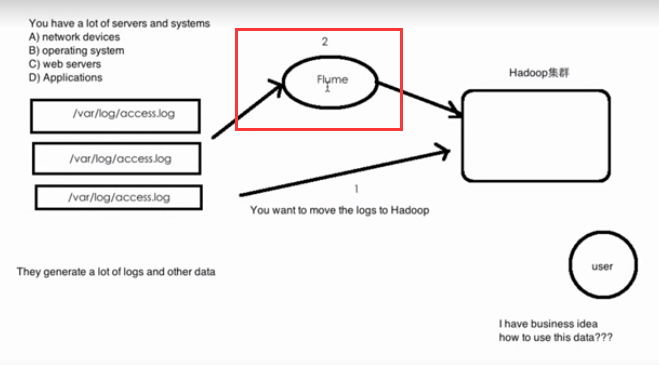
# 3 Flume

Flume是一个分布式日志收集框架。

## 3.1 为什么需要Flume

我们需要从日志中收集有价值的信息，因为日志数据量很大，所以需要将日志文件从web服务器收集到Hadoop集群中进行计算，这个收集过程Flume可以完成。

图1



## 3.2 Flume架构

每一个agent的组成包括：

Source：数据来源

Channel：数据缓存池，当Channel满了才将数据写到Sink中。

Sink：数据目的地

图1（单个agent）

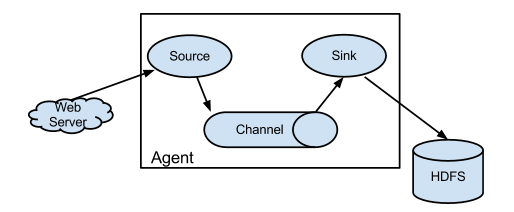


图2（多个agent）

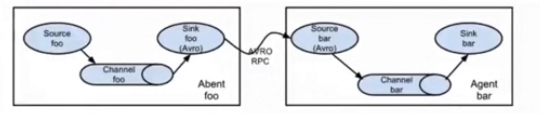


图3（多个agent）

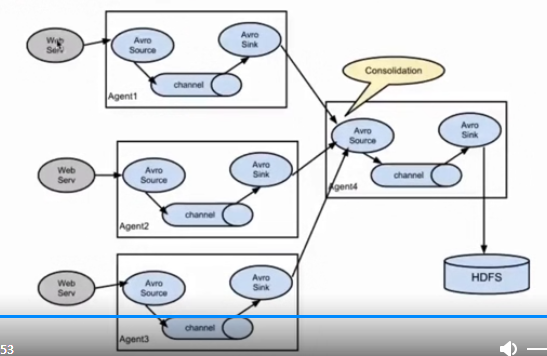
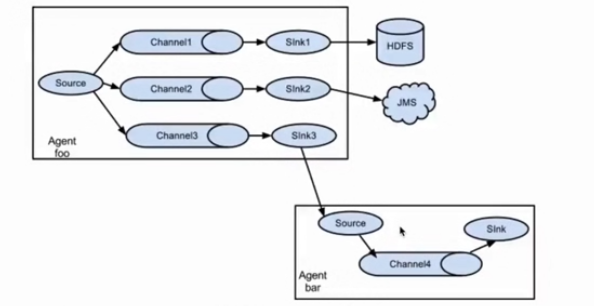


图4（多个Channel，多个Sink）



# 4 Spark

## 4.1 Spark运行模式

### 4.1.1 local

local一般用来开发调试Spark应用程序。

（注意：local和standlone是不一样的，local是本地运行，不会将应用提交到Spark集群；而standlone是会将应用提交到Spark集群运行的）

### 4.1.2 Standlone

Standlone使用Spark自带的资源管理和调度器，运行Spark集群，Spark集群使用Master/Slave结构。

（注意：--master如果不填，默认是Standlone，会提交到Spark集群运行）

### 4.1.3 Apache Mesos

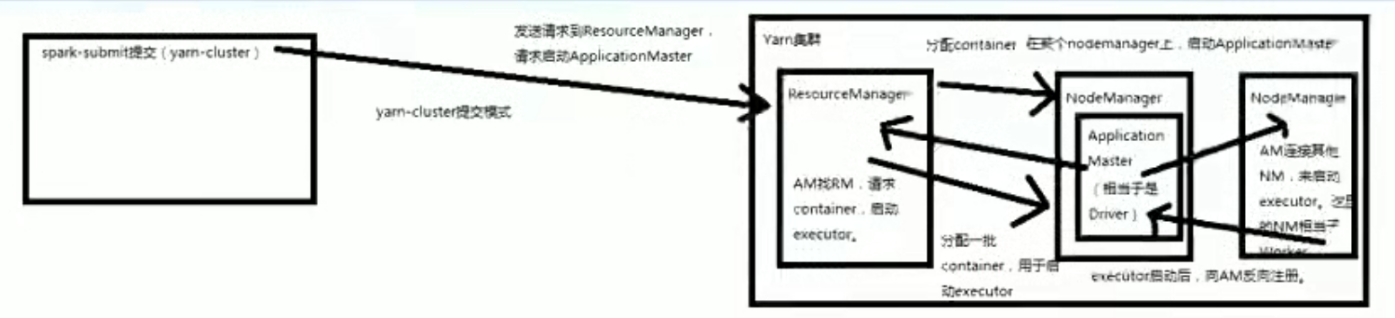
使用著名的Mesos资源管理框架，运行Spark集群。这种模式将资源管理交给Mesos，Spark只负责任务调度和计算。

### 4.1.4 Hadoop YARN

使用 Hadoop的YARN资源管理器，运行Spark集群。这种模式将资源管理交给YARN，Spark只负责任务调度和计算。

#### 4.1.4.1 Yarn cluster（适合生产环境）

图1



1. 客户端（Client，spark-submit命令执行的机器）通过提交Application到Yarn的ResourceManager，向ResourceManager请求启动Application Master。
2. ResourceManager（ResourceManager节点相当于Spark集群的Master节点）会选择一个NodeManager，分配container资源给该NodeManager，在这个NodeManager节点上创建Application Master（Application Master相当于Driver进程）。
3. Application Master会向ResourceManager发送请求，请求获取资源去创建Executor。ResourceManager分配container资源给Application Master，ApplicationMaster会连接其他NodeManager，在这些其他的NodeManager上利用分配的container资源创建executor。

（注意：创建Executor的NodeManager节点相当于Spark集群的Worker节点）

1. executor进程启动后，向ApplicationMaster进行反向注册。
2. Application Master会发送task到充当Worker的NodeManager的executor上执行。

Yarn cluster的优点：

1. 因为每次spark-submit时ResourceManager都会从几台NodeManager中选一台去分配container资源创建Application Master，所以不会导致一台NodeManager总是充当Application Master而造成一台NodeManager的网络带宽过高。

Yarn cluster的缺点：

1.本地spark-submit提交Application后，本地看不到log。只能通过yarn的logs application-id命令来查看log。

#### 4.1.4.2 Yarn client（适合测试，调试）

1.客户端（Client，spark-submit命令执行的机器，在此机器启动Driver进程）通过提交Application到Yarn的ResourceManager，向ResourceManager请求启动ExecutorLaucher（相当于Application Master）。

（注意：Yarn client模式下的ExecutorLaucher类似于Yarn cluster下的Application Master，但它的功能有限，并不相当于Driver进程，executor也不会向ExecutorLaucher进行反向注册）

2.ResourceManager（ResourceManager节点相当于Spark集群的Master节点）会选择一个NodeManager，分配container资源给该NodeManager，在这个NodeManager节点上创建ExecutorLaucher。

3.ExecutorLaucher会向ResourceManager发送请求，请求获取资源去创建Executor。ResourceManager分配container资源给ExecutorLaucher，ExecutorLaucher会连接其他NodeManager，在这些其他的NodeManager上利用分配的container资源创建executor。

（注意：创建Executor的NodeManager节点相当于Spark集群的Worker节点）

4.executor进程启动后，向spark-submit命令执行的机器上的Dirver进程进行反向注册。

5.spark-submit命令执行的机器上的Dirver进程会发送task到充当Worker的NodeManager的executor上执行。

Yarn client的优点：

1.因为Driver在本地，所以log日志都在本地，方便测试和调试。

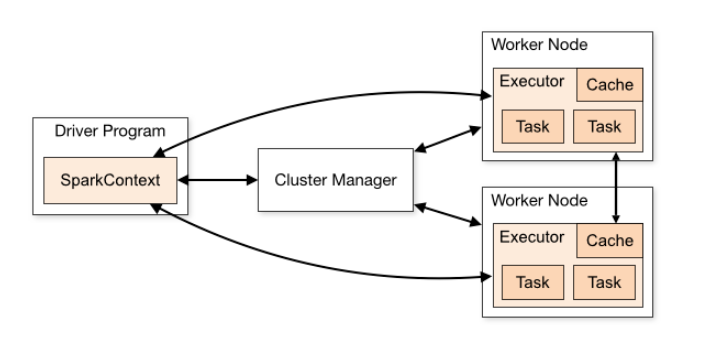
Yarn client的缺点：

1.因为Driver进程在本地，所以本地的Driver需要与Yarn集群进行大量的通信，会占用大量的网络带宽，会被公司的运维警告，因为这可能会导致river进程所在的机器的其他进程占用不到网络带宽而挂掉。

## 4.2 Spark组件架构

原文链接：<https://www.cnblogs.com/miqi1992/p/5621268.html>

图1



Application：Application就是我们编写的应用程序。

Driver Program：每个Spark应用（Spark Application）都由一个驱动程序（Driver Program）来发起集群的所有并行操作。Driver Program这个进程会去运行我们编写的Application应用程序。

（注意：Driver Program是一个Driver进程，它存在于提交Spark应用的机器上。如果是本地开发，Driver进程就在Windows机器，如果在Spark集群的某一个节点上运行spark submit命令，则Driver进程就在该Spark节点上）

Cluster Manager：资源管理器。在Standlone模式下是Master节点，在YARN模式下是ResourceManager。

（注意：资源管理器是可插拔的）

Worker Node：工作节点（包含执行器和缓存，执行器负责执行任务，缓存负责缓存RDD的某个分区）。在Standlone模式下是Slave节点，在YARN模式下是NodeManager，负责启动Executor。

（注意：Worker节点除了会在自己节点启动Worker进程，还会启动Executor进程）

Executor：执行器，在Worker Node上启动，负责计算任务（Task）。

（注意：Executor启动后，会向Driver进行反注册，这样Driver就知道有哪些Executor为它服务了）

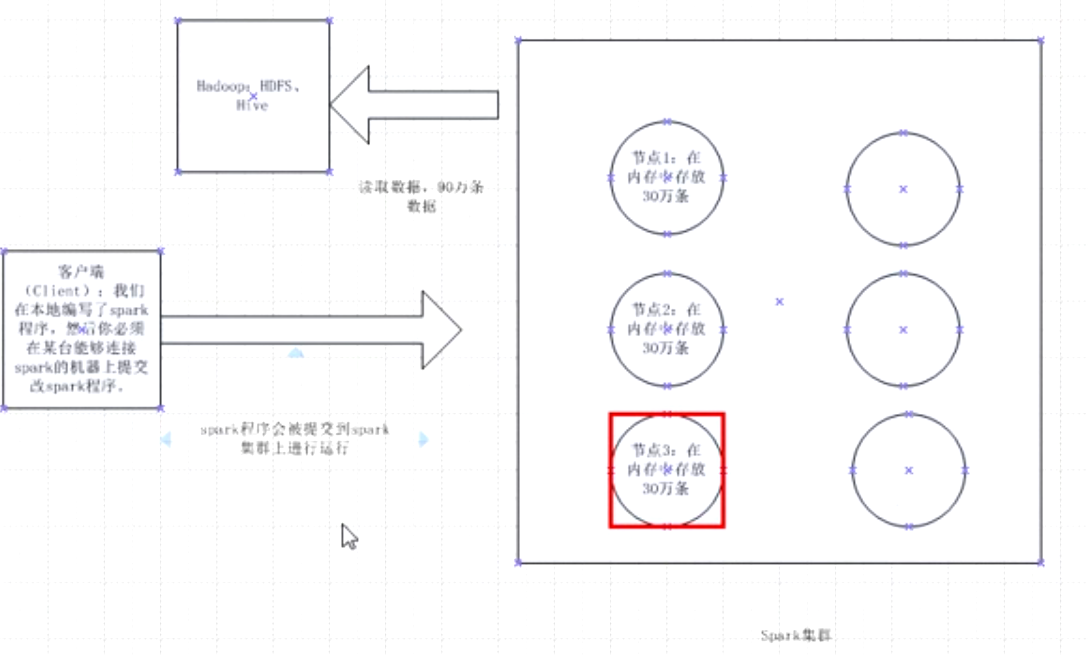
（注意：Executor反注册后，Driver会提交很多Task（算子操作）到Executor上，Executor会启动很多个线程去执行这些Task（算子操作））

SparkContext：整个Spark应用（Spark Application）的上下文，负责控制应用的生命周期。

### 4.2.1 分布式内存存储

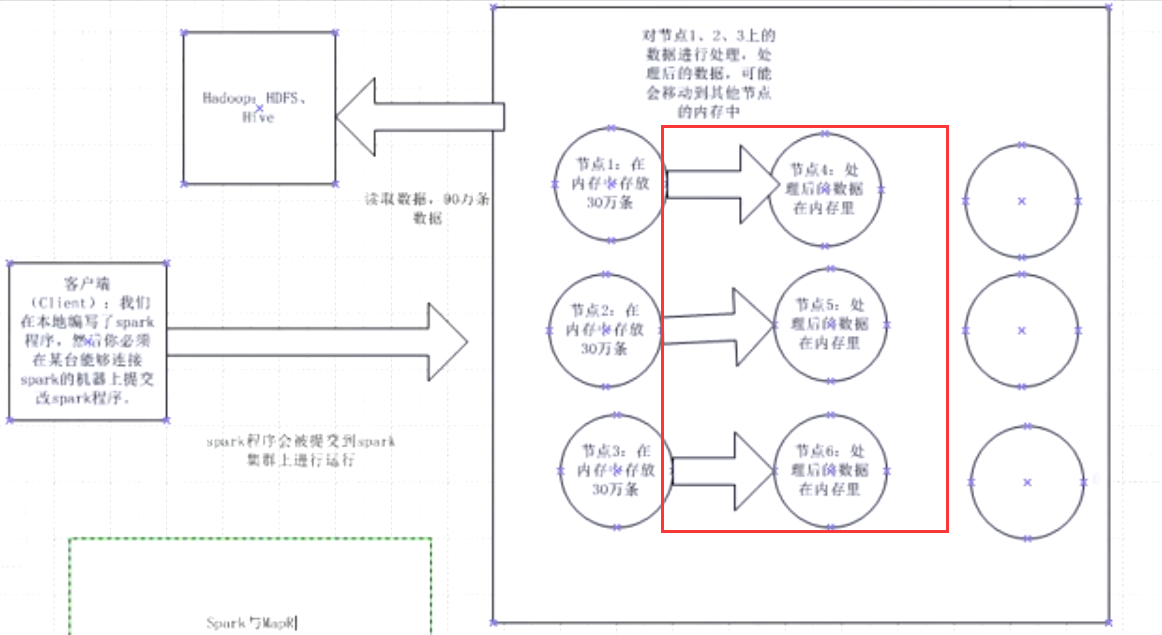
Spark读取的90W数据，不会只存储在一个节点上，而是会存储在多个Spark节点的内存中。

图1



（注意：处理完的数据量不能少于30M，也可能大于30M，处理完的数据可能会移动到其他节点上）

图2



## 4.3 Spark Shell

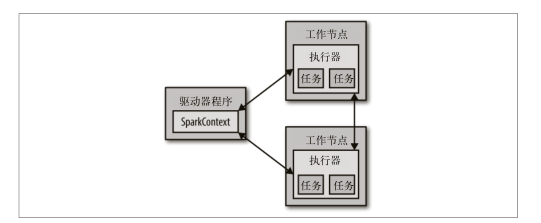
### 4.3 Spark shell和其他shell的不同点

其他shell只能用单机的硬盘或内存操作数据，而Spark shell可以用多机的硬盘或内存操作数据，且分发操作由Spark自动控制完成。

## 4.4 SparkContext

一个SparkContext对象表示对Spark计算集群的一个连接。

图1



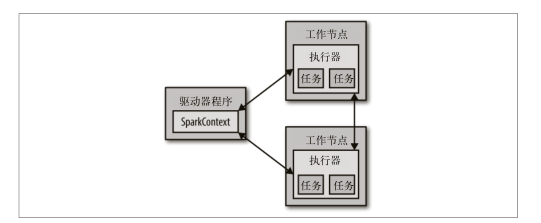
### 4.4.1 Spark驱动程序和SparkContext的关系

每个Spark应用都通过一个驱动器程序来发起集群中的并行操作。驱动器程序包含了main函数。

Spark驱动程序通过创建一个SparkContext来访问Spark计算集群。

（注意：驱动程序就是我们平时Scala/Python/Java写的程序，包含了main函数）

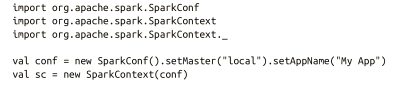
图1



### 4.4.2 SparkConf

可以通过创建一个SparkConf来配置Spark应用，然后基于这个Spark Conf来创建一个SparkContext对象。

图1



## 4.5 RDD

RDD（Resilient Distributed Dataset，弹性分布式数据集）是不可变的分布式数据集合。

### 4.5.1 分区

每个RDD都被分为多个分区，每个分区运行在Spark集群中的不同节点上，这也是为什么RDD叫做分布式数据集的原因。

比如：一个RDD如果有90W数据，这90W数据假设分为了9个分区，每个分区10W数据，每个分区分布在Spark的1个节点上。

（注意：如果每个分区10W数据，但是每个节点的内存只能容纳5W数据，则每个分区的部分数据会写入磁盘上，这种内存和磁盘的切换存储就是弹性存储，这也是为什么RDD叫做弹性数据集的原因）

图1

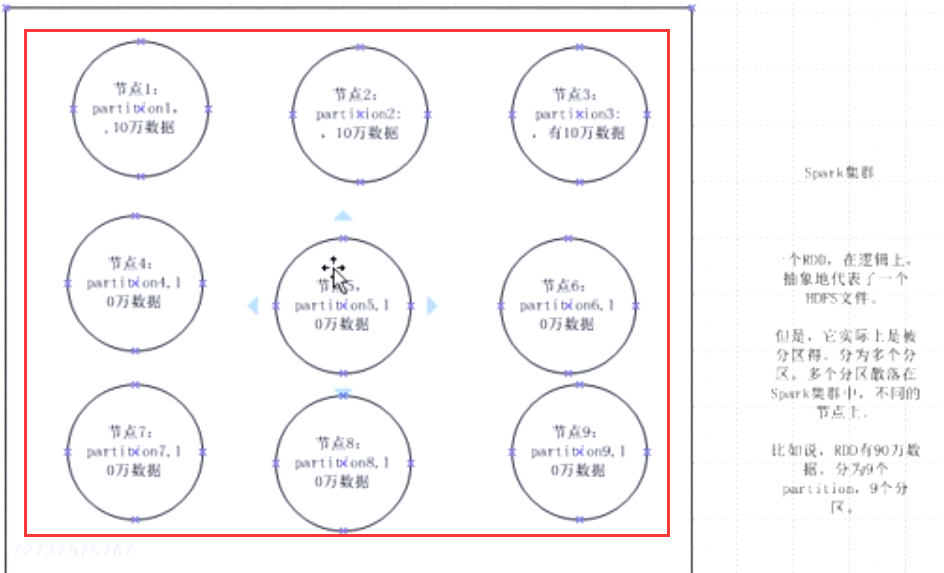


图2（弹性存储，分区部分数据存储在磁盘）

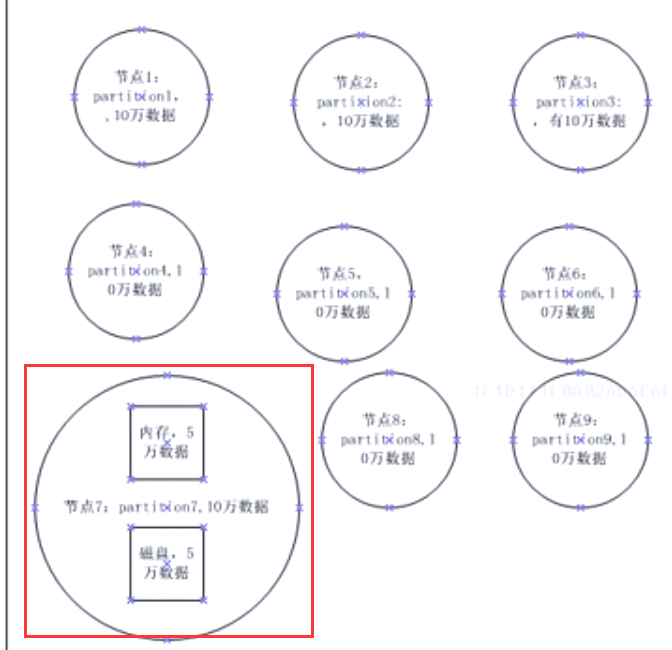
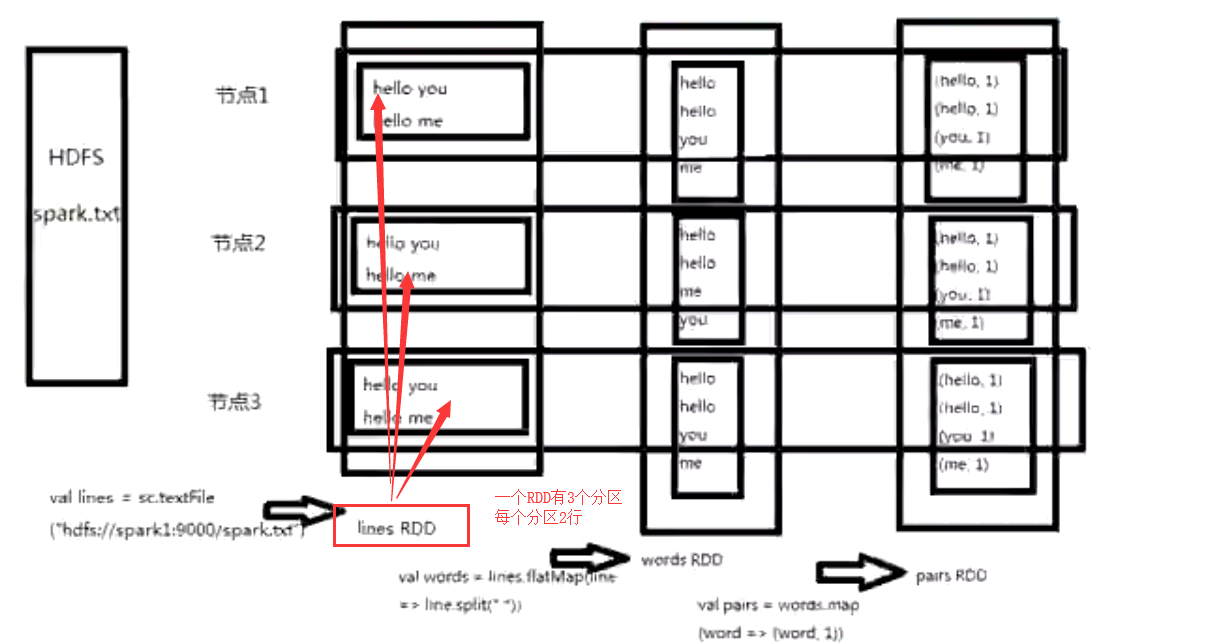


图3（从文本文件读取RDD）



### 4.5.2 创建RDD

法1：读取一个外部数据集

图1



法2：在驱动器程序中分发对象集合，比如：list、set

图2



#### 4.5.2.1 textFile 从文件中创建RDD

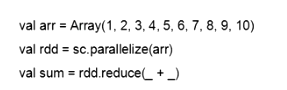
textFile (filePah) 从文件（本地文件或HDFS文件）中创建RDD。

（注意：filePah可以是目录，文件，文件通配符，压缩包等等）

（注意：当使用textFile为hdfs文件创建RDD时，Spark默认为一个block创建一个分区，可以通过第二个参数手动指定分区，但分区数必须>=block数）

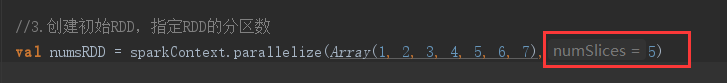
#### 4.5.2.2 parallelize 从程序集合中创建RDD

图1



（注意：parallelize 方法可以指定为该RDD创建多少个分区）

图2（为该RDD创建5个分区）



### 4.5.3 RDD支持的操作

RDD创建后，支持两种类型的操作，转化操作（transformation）和行动操作（action）。

转化操作：由一个RDD生成一个新的RDD。

图1

（转化操作：过滤后，生成一个新的只包含Python单词的字符串的新的RDD）



行动操作：行动操作会对RDD计算得出一个结果，并将这个结果返回到驱动程序中，或将结果存储到外部存储程序（比如：HDFS）中。

（注意：默认情况下，Spark的RDD在每次调用行动操作的时候，都会重新计算）

图2

（行动操作，返回一个结果给驱动程序）



#### 4.5.3.1 转化操作和行动操作的区别

1.转化操作的返回值是RDD，行动操作的返回值是其他类型。

2.转化操作是由一个RDD生成一个新的RDD，行动操作是返回结果给驱动程序或将结果存储到外部存储程序（如HDFS）中。

3.只有转化操作（transformation）是不会执行的，必须有行动操作（action）才会执行。

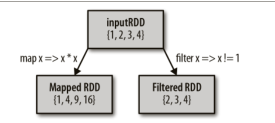
#### 4.5.3.2 转化操作（transformation）

##### 4.5.3.2.1 map

map接收一个函数（x=>x\*x），把这个函数应用于RDD中的每个元素，将函数的返回结果作为新的RDD中对应元素的值。

（注意：map()函数的返回值类型不需要和输入类型一样，比如输入类型是RDD[String]，返回值类型可以是RDD[Double]）

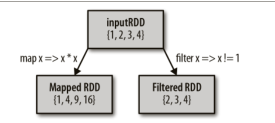
图1



##### 4.5.3.2 filter

filter接收一个函数（x=>x!=1），把这个函数应用于RDD中的每个元素，将RDD中满足该函数（只要那些func返回true的元素）的元素放入新的RDD中。

图1



##### 4.5.3.2.3 flatMap

flatMap接收一个函数（line=>line.split(“ ”)），把这个函数应用于RDD中的每个元素，每个元素返回一个迭代器，但新的RDD不是由迭代器组成的，而是由各迭代器能访问到的所有元素组成的。

图1



##### 4.5.3.2.4 distinct

因为RDD中经常会包含一些重复元素，RDD.distinct()可以对RDD进行去重。

（注意：distinct()的开销很大，因为它需要将所有元素通过网络（因为数据是分布式存储的）进行混洗（shuffle），从而确保每个元素只有一份）

##### 4.5.3.2.5 union，intersaction，subtract，cartesian

union是并集，对2个RDD求并集形成一个新的RDD。

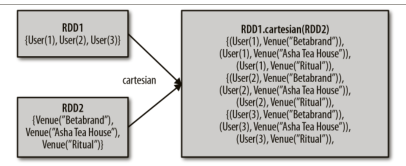
intersaction是交集，对2个RDD求交集形成一个新的RDD。

（注意：intersaction不仅会去掉2个RDD中重复的元素，还会通过混洗去除单个RDD中重复的元素，所以intersaction的性能会比union差很多）

subtract是差集，返回只存在于第一个RDD中，但不存在于第二个RDD中的所有元素组成的新的RDD。

cartesian是笛卡儿积，对2个RDD求笛卡儿积集形成一个新的RDD。

图1（cartesian（笛卡尔积）的结果）



##### 4.5.3.2.9 reduceByKey

把这个函数应用于RDD中的key相同的2个元素，返回一个新元素，reduce是合并是意思，新元素的key是原来的key，value是原来的2个元素的value合并后的value。

（注意：reduceByKey首先会在原分区所在节点根据key进行combine，然后根据key型shuffle，key值相同的shuffle到同一个节点）

（注意：combine之后的分区合起来其实也是一个RDD，但不是程序里面出现的RDD，是隐藏的RDD）

（注意：reduceByKey是transformation操作，而不是action操作，因为reduceByKey算子返回的是一个新的Pair RDD）

图1（在原分区所在节点根据key进行combine）

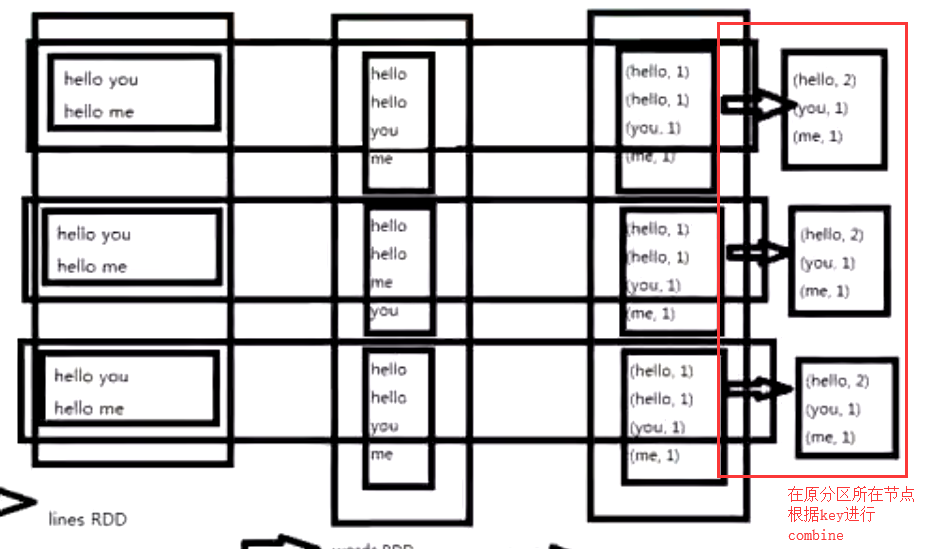
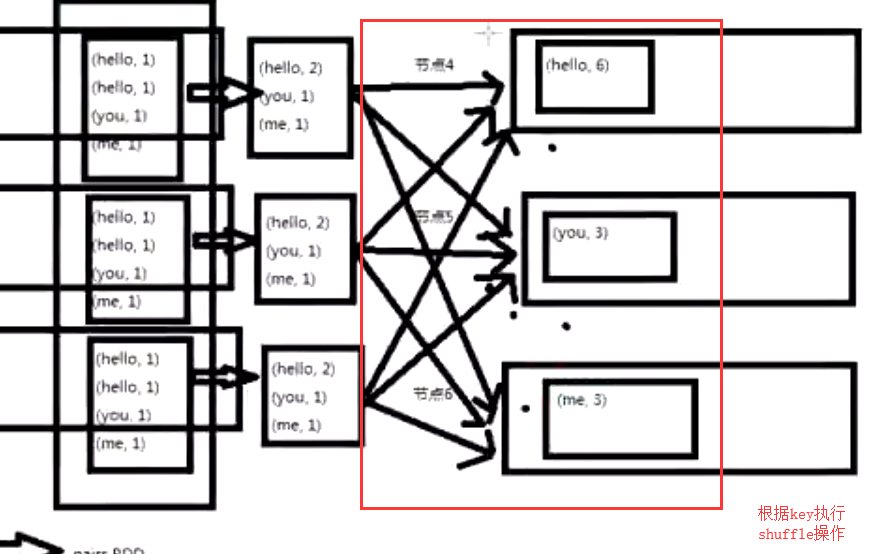


图2



##### 4.5.3.2.10 fold

fold和reduce一样，接收一个函数（(x,y)=>x+y），把这个函数应用于RDD中的2个元素，返回一个新元素。不同的是需要提供一个初始值。

图1



图2



##### 4.5.3.2.11 groupByKey

把这个函数应用于RDD中的key相同的2个元素，返回一个新元素，这个新元素的key就是原来的key，value就是原本的2个元素的value组成的集合。

##### 4.5.3.2.12 sortByKey

把这个函数应用于RDD中的key相同的2个元素，返回一个新元素，新元素的key和value都是原来的，但是在新RDD中根据key进行了排序。

##### 4.5.3.2.13 join

把这个函数应用于2个RDD中的（RDD1中取一个元素RDD v1，然后从RDD2中取所有key=（RDD v1的key）的元素，RDD2中的每一个元素RDD v2和RDD v1组成一个新元素），新元素的key和原来相同，新元素的value是原来的key相同的2个元素的value组成的二元元组。

（注意：其实类似于SQL里面的内连接）

（注意：如果RDD2中有2个以上的元素和RDD v1匹配成功，则每个RDD v2和RDD v1组成一个新元素，最后返回2个以上的新元素）

（注意：join后的结果中，一个key可以有多个新元素）

图1

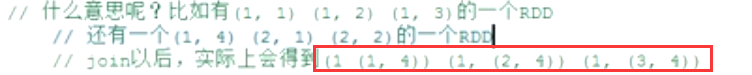


图2

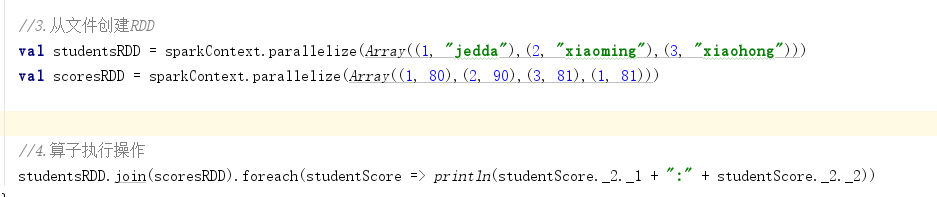
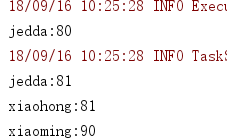


图3



##### 4.5.3.2.14 cogroup

把这个函数应用于2个RDD中的相同key的元素的value聚合到一个二元元组中，该二元元组的第一个位置是第一个RDD的value的可迭代对象，第二个位置是第二个RDD的value的可迭代对象。

（注意：cogroup后的结果中，一个key只有1个新元素）

#### 4.5.3.3 行动操作（action）

一个action操作会触发一个Spark Job执行，然后Driver程序会提交Task到Spark集群中的Executor中执行。

（注意：如果是textFile读取文件，Executor收到Task后，才会去本地文件或HDFS文件读取数据到Task所在的Worker节点的内存中，执行算子操作等等）

##### 4.5.3.3.1 foreach

foreach会对RDD中的每个元素进行操作，但不返回给驱动程序。

（注意：foreach是在远程的Spark集群中执行的，而不是在Driver程序的内存中执行的，所以性能会很快。）

##### 4.5.3.3.2 reduce

reduce接收一个函数（(x,y)=>x+y），把这个函数应用于RDD中的2个元素，返回一个合并后的值。

（注意：reduce要求函数返回值和RDD中的元素类型相同，比如图1中sum的类型和RDD中的元素类型相同）

（注意：因为reduce返回的是一个值而不是一个新的RDD，所以它是action操作而不是transformation操作）

图1（对RDD中所有元素进行求和）



##### 4.5.3.3.3 collect

collect()将Spark集群上的RDD的所有元素复制到驱动器程序所在的单台机器的内存中。

collect的缺点：

1. RDD的全部元素要经过网络传输到达Driver程序，如果数据量很大，网络耗时严重
2. 如果RDD的数据量很大，可能会造成Driver程序出现OOM

图1

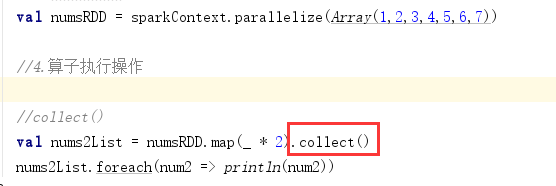
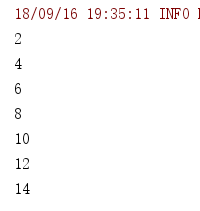


图2



##### 4.5.3.3.4 count

count统计RDD中有多少个元素。

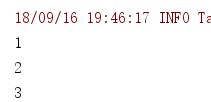
##### 4.5.3.3.5 take

take(n)会取RDD中的前n个元素，并且会尝试访问尽量少的分区。和collect类似，只是collect是获取RDD所有元素到Driver内存，take是取前N个元素到Driver内存。

图1



图2



##### 4.5.3.3.6 saveAsTextFile

把这个函数应用于RDD中的每个元素，将每个元素保存到本地文件或HDFS文件上去。

（注意：saveAsTextFile的参数是一个目录，目录下会生成2个文件，\_SUCCESS表示成功，part-xxxxx是数据）

（注意：如果saveAsTextFile的目录已经存在，会报错说目录已存在）

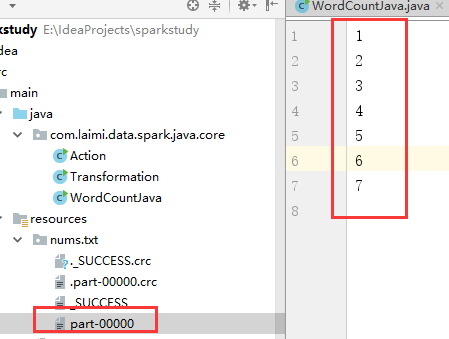
图1



图2



图3



##### 4.5.3.3.7 countByKey

把这个函数应用于RDD中的每个元素，按key进行分组，统计每个分组的元素的总数，返回一个Map类型。

（注意：countByKey返回的不是一个新的RDD）

图1



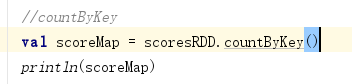


图2



### 4.5.4 Spark惰性计算

惰性计算就是：转化操作并不会计算RDD，只有第一次在某个行动操作中用到这些RDD时，才会真正计算。因为行动操作前可能有很长的转化操作链，当Spark了解完整的转化操作链后，会只计算需要用到的数据。

图1



图2

（当Spark了解完整的转化操作链后，只读取文件的第一行就结束，而不是读取全部行到内存再取第一行）



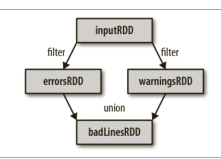
### 4.5.5 谱系图

在进行转化操作时，Spark会使用谱系图来记录这些不同的RDD之间的关系。Spark利用谱系图去计算每个RDD，并且可以根据谱系图去恢复丢失的数据。

图1



图2（谱系图）



### 4.5.6 RDD持久化

当持久化一个RDD时，该RDD所在的Spark集群的各个节点的的各个分区都会保存分区计算结果，然后序列化，将序列化结果保存在JVM堆空间中。

在RDD持久化后，后续如果再用到持久化的结果，则会直接取分区计算结果，而不是重新计算。（比如：读取文件的transformation操作如果持久化后，后续的重复操作会从分区直接取数据，而不是重新读取文件）

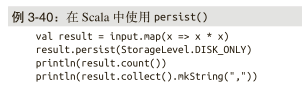
（注意：持久化使用persist()方法；cache()方法底层也是调用persist(MOMERY\_ONLY)，持久化到Spark节点的内存）

（注意：如果创建了一个RDD后，然后单独另起一行执行cahe()或persist()方法，是不会将RDD持久化的，会报错并导致大量文件丢失；cahe()或persist()必须要和transformation或action操作连在一起使用，才会将RDD持久化）

图1（count()和collect()都是行动操作，会引发2个重复的map计算）



图2（缓存结果）



#### 4.5.6.1 RDD不持久化的缺点

RDD如果不持久化，在每次执行action操作时，都要重复执行前面的transformation操作（图1中是textFile，会重复读取2次文件），以及重复执行前面的action操作（图1中是count，会重复count2次）。而对于大数据量而言，每次transformation或action操作可能都要耗费1个小时的时间，会造成重复的时间浪费。

图1



#### 4.5.6.2 RDD持久化策略

（注意：MEMORY\_ONLY是以非序列化的方式存储在JVM内存的；MEMORY\_ONLY\_SER是以序列化的方式存储在JVM内存的）

（注意：MEMORY\_ONLY\_SER因为经过了序列化再存储，拿出来的时候又要反序列化，所以优点是节省内存空间，缺点是增大CPU开销）

图1



图2



#### 4.5.6.2 如何选择RDD持久化策略

1.优先使用MEMORY\_ONLY，因为纯内存，无序列化的速度最快，前提是内存能存储RDD所有的数据。

2.如果内存无法存储RDD所有的数据，则使用MEMORY\_ONLY\_SER，因为序列化后的数据可以进行持久化到磁盘，无法存储的那一部分可以存储在磁盘。而MEMORY\_ONLY无法存储的那一部分因为无序列化无法存储，所以需要重新计算。

3.如果需要快速的失败恢复，则使用后缀为\_2的策略。因为\_2的策略会进行数据的备份，就算失败了，也可以从备份读取数据。

4.能不使用DISK相关的策略，就不使用，有的时候，从磁盘读取数据，还不如重新在内存计算一次。

### 4.5.8 常见大型数据源

#### 4.5.8.1 文本文件

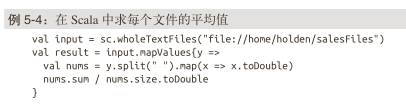
从单个文件读取时，输入的每一行会变成RDD的一个元素。

图1（读取单个文本文件，返回一个RDD[String]）



从一个目录读取多个文件，返回一个Pair RDD[String,String]，key是文件名，value是文件内容。

图2（读取多个文件时，返回一个Pair RDD[String,String]）



### 4.5.9 累加器

累加器就是一个共享变量，可以将工作节点中的值聚合到驱动器程序中。

（注意：工作节点无法读取累加器的值，对于工作节点来说，累加器是只写的）

累加器的用法：

图1

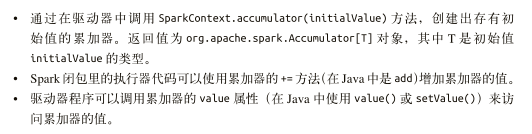


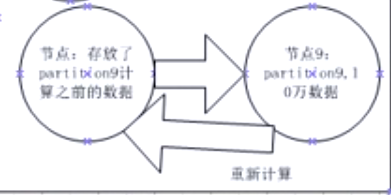
图2（例子）



### 4.5.10 RDD的容错性

如果RDD的分区9的数据因为节点9的宕机而丢失了，Spark会从分区9的数据来源（数据来源可能是分区8的数据，也可能是分区7的数据）重新计算该分区的数据，然后存储在其他节点上。

图1



### 4.5.11 RDD的元素

RDD是数据集，数据集是由很多个元素组成的。

图1（当读取文件时，每一个元素就是文件的一行）



### 4.5.12 Java API的function的参数和泛型理解

Java API中的参数和function其实就是类似于Scala的(xxx):xxx=>{xxx}。泛型可以查看Function的源码，就可以知道每个位置的泛型是用来限制哪个参数或返回值的类型的。

图1

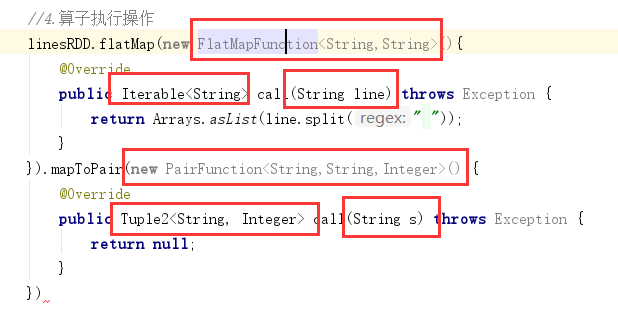
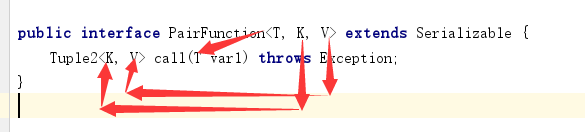


图2



### 4.5.13 共享变量

默认情况下，如果一个算子的函数中用到了某个外部的变量，这个变量的值会被拷贝到每个节点的每个task中（注意：是每个task拷贝一份，不是每个节点），然后每个task就只能操作自己task的那个变量副本，多个task无法共享变量的值。

Spark为此提供了2种共享变量，Broadcast Variable（广播变量）和Accumulator（累加变量）。

#### 4.5.13.1 Broadcast Variable（广播变量）

Broadcast Variable（广播变量）：变量的值会被拷贝到每个节点上（注意：此时是每个节点一份，不是每个task一份）。广播变量的作用是优化性能，减少网络传输和内存消耗。

（注意：因为如果那个变量很大的话，将该变量copy到每个task，网络传输是要消耗时间的，每个task存储那个大变量也是要内存的，所以说Broadcast Variable拷贝到每个节点可以减少网络传输和内存消耗）

图1

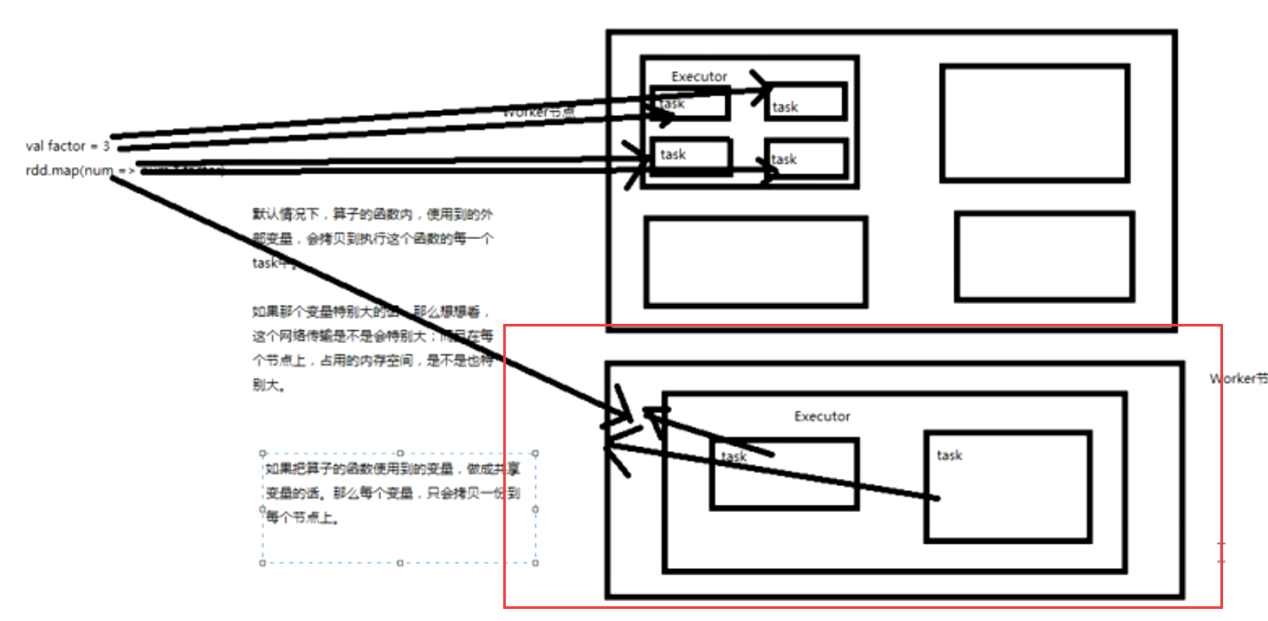
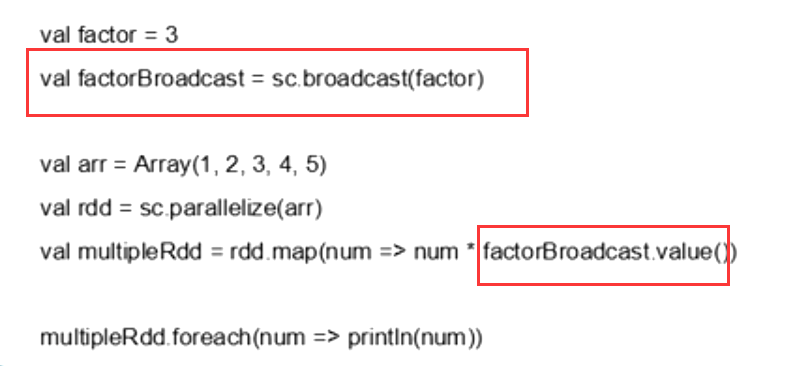


图2



#### 4.5.13.2 Accumulator（累加变量）

Accumulator（累加变量）：累加变量可以让多个task共享变量。

（注意：Accumulator是多个task共享读，是不能写的）

（注意：Driver程序不能读累加变量的值，而只能对这个值进行累加）

图1

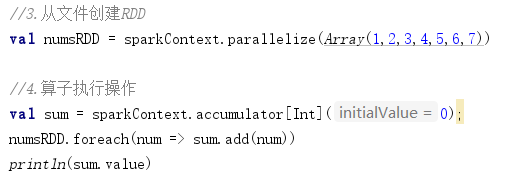


图2



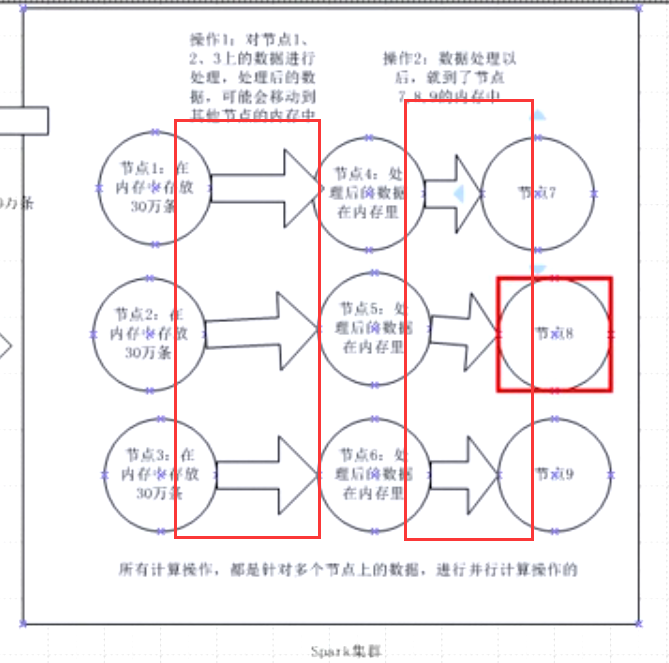
### 4.5.14 二次排序

二次排序需要定义一个自定义的key（Java实现Ordered接口，Scala继承Ordered），然后用sortByKey算子去根据这个自定义的key进行操作。

## 4.6 Spark和MapReduce的区别

1.MapReduce只有map和reduce2个阶段，Spark可以有任意多的阶段。

图1



# 5 Spark内核源码剖析

## SparkContext

Driver进程运行Application程序的第一件事情就是初始化SparkContext。

在Executor反向注册后，Driver程序会结束SparkContext的初始化。

（注意：SparkContext只是一个对象，不是一个进程）

图1



## DAGScheduler

SparkContext初始化时，会去创建DAGScheduler和TaskScheduler。

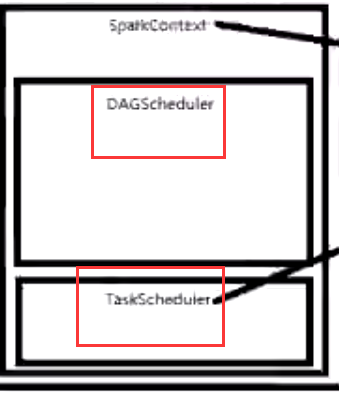
注意：DAGScheduler基于DAGSchedulerEventProcessActors组件（线程）来进行通信。

SparkContext初始化完成后，每一个action操作就会创建一个Job，这个job会提交给DAGScheduler。

DAGScheduler每接收到一个Job，就会将这1个Job划分为多个stage。每一个stage会创建一个TaskSet。

然后DAGScheduler会将每一个TaskSet传给TaskScheduler。

图1



## TaskScheduler

SparkContext初始化时，会去创建DAGScheduler和TaskScheduler。

注意：SparkContext创建TaskScheduler时createTaskScheduler()，会创建三个东西：

1. 创建TaskSchedulerImpl：TaskSchedulerImpl底层实际上基于SparkDeploySchedulerBackended来工作。
2. SparkDeploySchedulerBackended：它负责注册Application到Master节点；Executor的反注册；通过调度算法调度Job的优先级顺序；发送Task到Executor等操作。
3. 创建调度池Schedule Pool，有不同的优先级策略（比如：FIFO先进先出）。

图1



TaskScheduler会对应一个自己的后台进程，然后这个后台进程会去连接Master节点，向Master节点注册Application。

Master收到Application的注册请求后，会利用自己的资源调度算法，通知Spark集群的Worker，让Worker为这一个Application创建1个或多个Executor进程。

TaskScheduler接收到DAGScheduler传过来的每一个TaskSet后，会将每一个TaskSet中的每一个Task提交给已知的1个或多个Executor上去执行。

## Executor

Executor进程启动后，会自己反向注册到TaskScheduler。这样，TaskScheduler就知道自己服务的Application有哪些Executor进程去执行了。

在Executor反向注册后，Driver程序会结束SparkContext的初始化。

Executor接收到TaskScheduler传过来的每一个Task，会用TaskRunner封装每一个Task，然后从线程池里面取出一个线程，去执行这个TaskRunner。

TaskRunner会将我们编写的Application程序的算子，函数，变量等拷贝，反序列化，然后执行它里面的Task。

## Job

SparkContext初始化完成后，每一个action操作就会创建一个Job，这个job会提交给DAGScheduler。

## Task

Task有2种：ShuffleMapTask和ResultTask。只有最后一个stage是ResultTask，其他都是ShuffleMapTask。

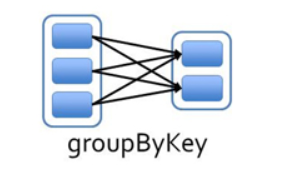
（注意：每一个Task执行自己节点上的，RDD的某一个partition）

## 宽依赖和窄依赖

宽依赖（shuffle dependency，也称wide dependency）：父RDD的一个分区可以被子RDD的多个分区使用。

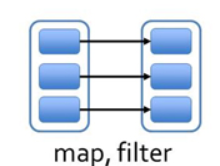
（注意：之所以wide dependency也叫shuffle dependency，因为这本质就是一个shuffle过程。shuffle就是指父RDD的一个分区的一部分会复制到子RDD的一个分区的一部分上，这样就会导致子RDD的一个分区拥有父RDD的多个分区的交叉数据）

图1（宽依赖）



窄依赖（narrow dependency）：父RDD的一个分区只能被最多子RDD的一个分区使用。

图2（窄依赖）



## Spark UI

Spark UI实际上是启动一个Jetty服务器，来显示网页。