# 1 Spark SQL

## 1.1 SQLContext

### 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）

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



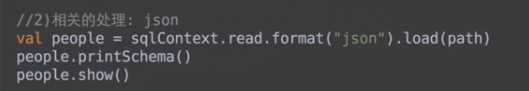
SparkSession是Spark2.\*中的入口。

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；

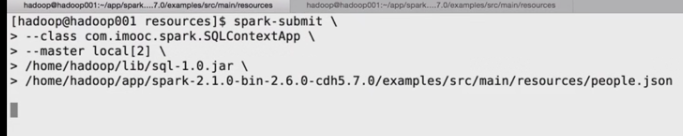
Java开发使用JavaSparkContext；

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

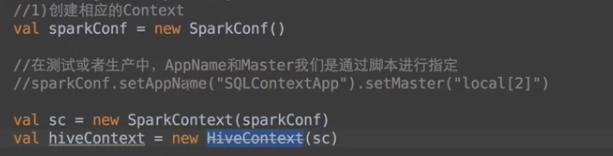


## 1.3 HiveContext

SparkContext是Spark 1.\*中的Spark SQL的入口，必须先创建SparkContext。再用SparkContext创建HiveContext。

（注意：使用HiveContext是不需要搭建Hive环境的）

图1



### 1.3.1 HiveContext显示table的数据

图1



## 1.4 SparkSession

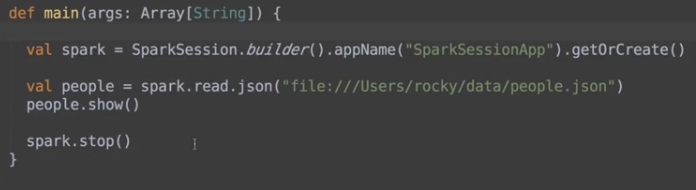
SparkSession是Spark2.\*中的入口。

图1



### 1.4.1 SparkSession读取JSON文件

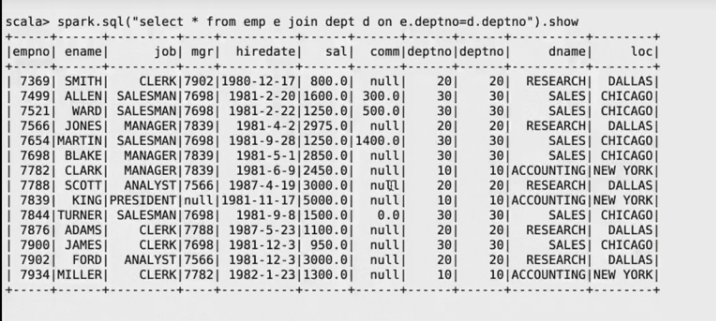
图1



## 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



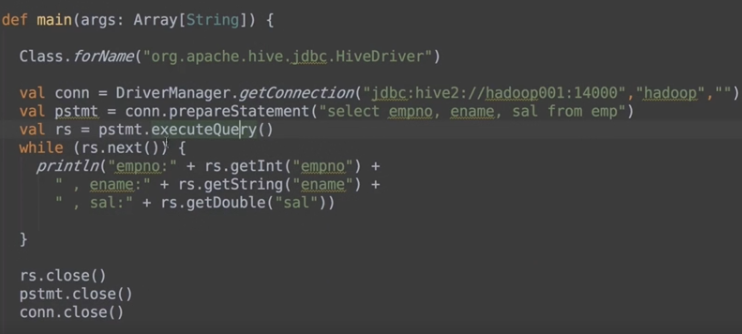
### 1.6.1 thriftserver和spark-shell/spark-sql的区别

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。

图1



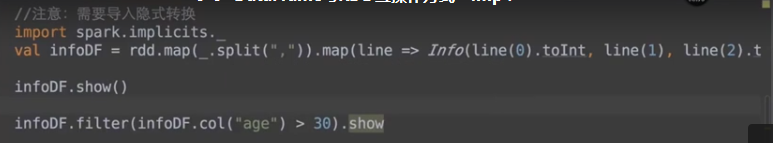
图2（DataFrame的基本API）



### 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。

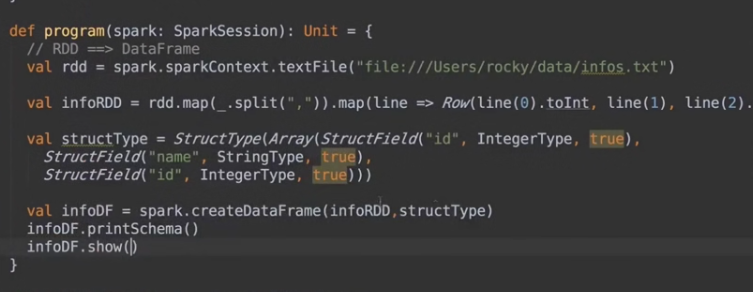
图1



### 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



## 1.8 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



# 2 Spark Streaming

Spark Streaming是一个实时流处理框架。

## 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。

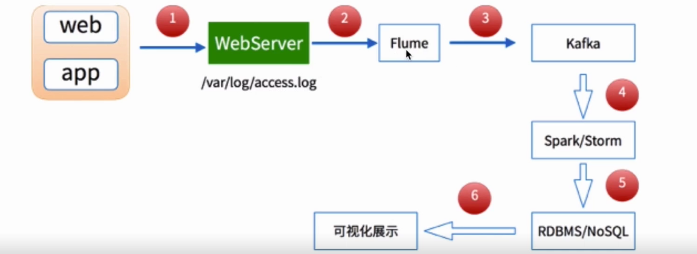
4.Kafka Streaming。

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。

## 2.2 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

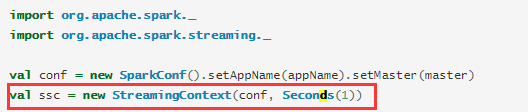
spark-shell可以直接写scala语句，适合开发和测试。

## 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。

图1

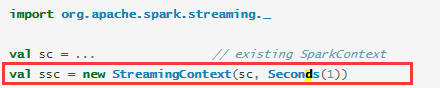
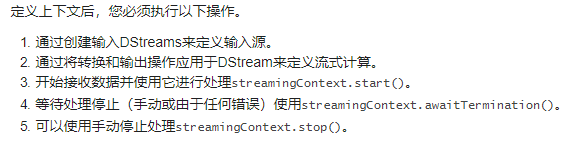


图2（创建StreamingContext后进行的操作）



## 2.5 DStream

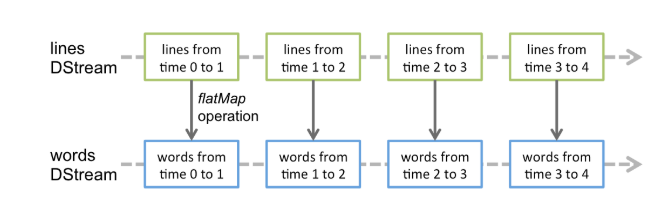
DStream（Discretized Stream，离散的数据流）表示连续的数据流。DStream由连续的RDD构成，每个RDD是特定时间间隔的数据集合。

（注意：DStream的任何操作都会转换成底层RDD上的操作）

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））

#### 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：追踪每个用户最近访问的10个页面。

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）

图1



#### 2.5.1.4 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

（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）

图1

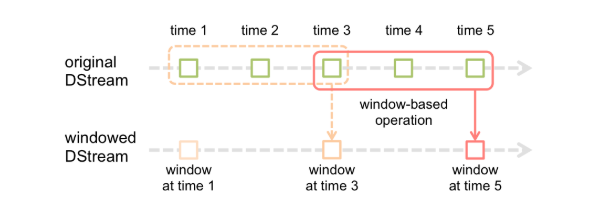
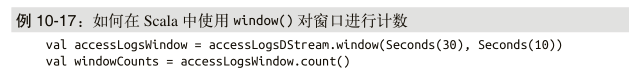


图2（window()返回一个新的DStream，这个DStream包含这个窗口内的所有RDD）



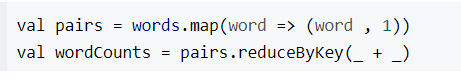
#### 2.5.1.5 transform

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

#### 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



### 2.5.2 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

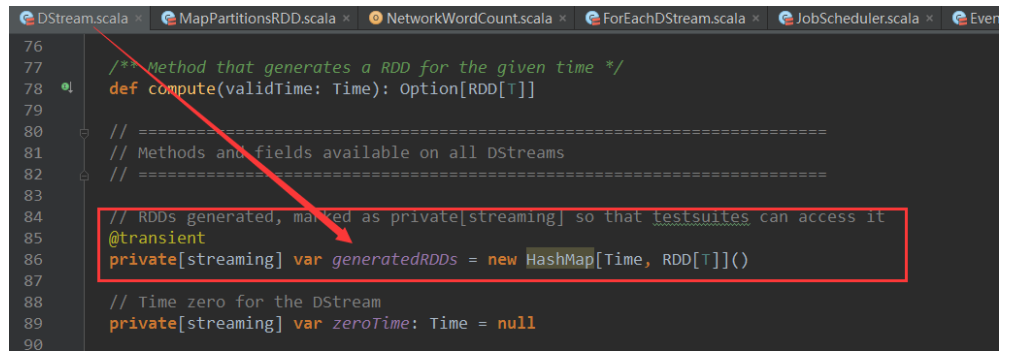
DStream由连续的RDD构成，每个RDD是特定时间间隔的数据集合。

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：

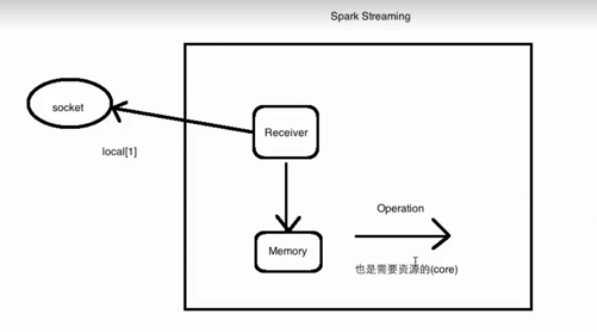
1. Input DStream
2. Transformed DStream
3. Output DStream

## 2.6 Receiver

每个DStream（除了文件流）都与Receiver相关联，Receiver会从源接收INPUT DStream，并将它存储在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



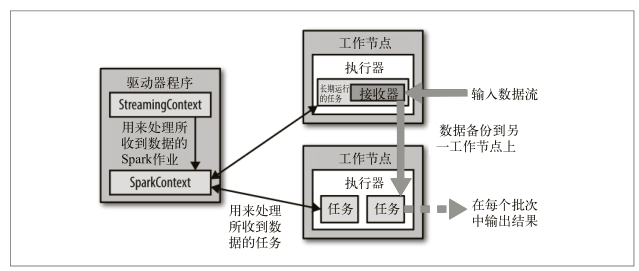
### 2.6.1 两类内置的源

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。

## 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。

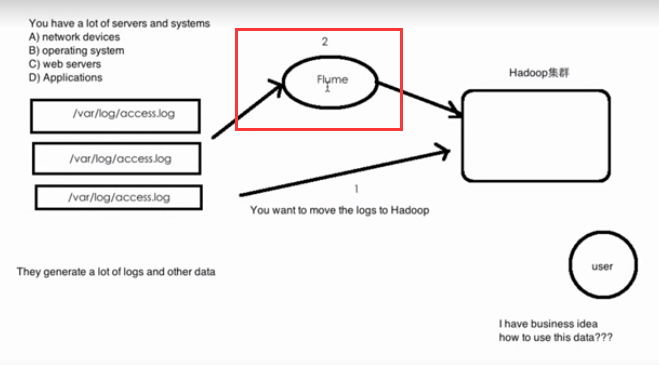
# 3 Flume

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）

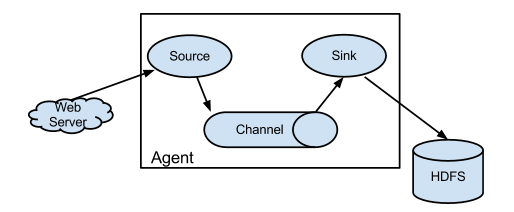


图2（多个agent）

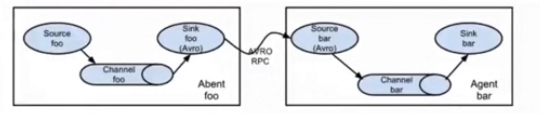


图3（多个agent）

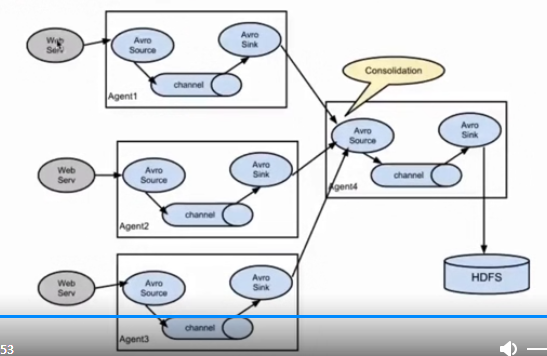
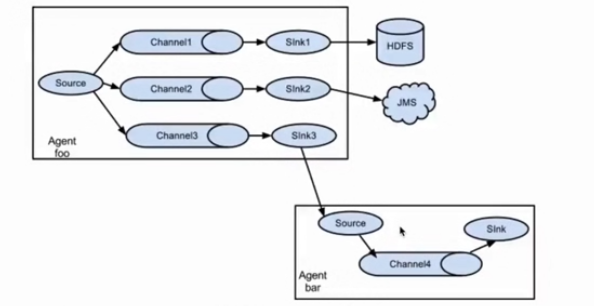


图4（多个Channel，多个Sink）



# 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。

### 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。

### 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。

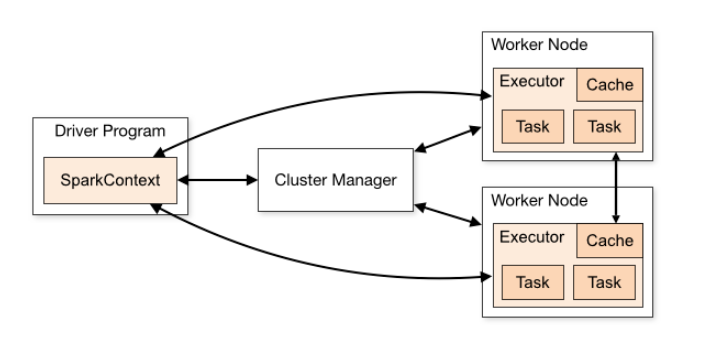
### 4.1.4 Hadoop YARN

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。

## 4.2 Spark组件架构

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

图1



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。

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）。在Standlone模式下是Slave节点，在YARN模式下是NodeManager，负责启动Executor。

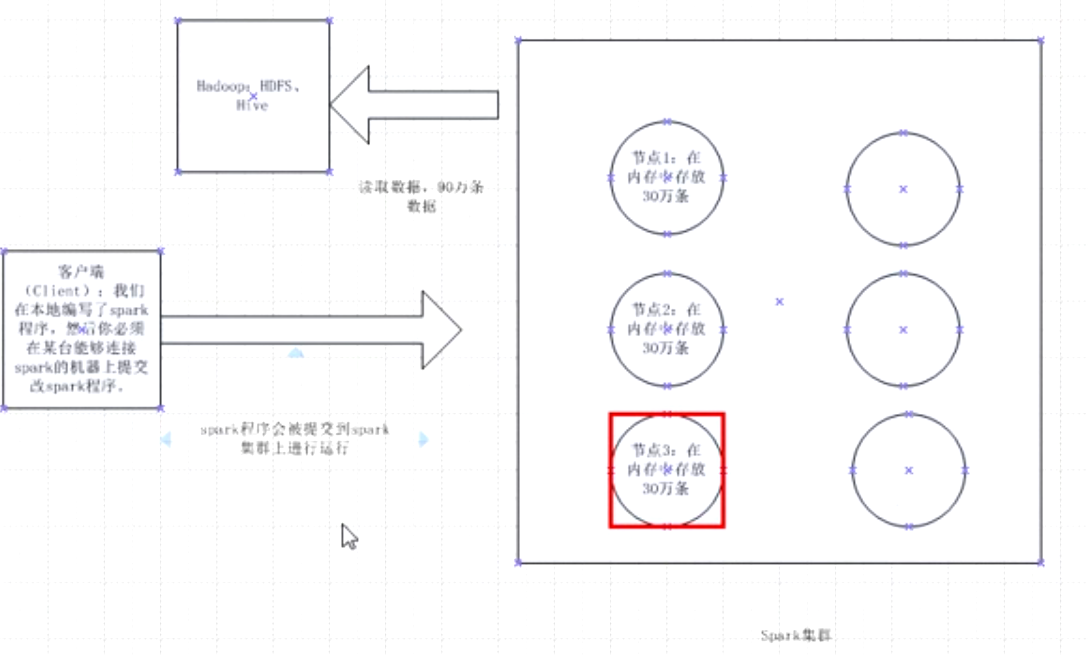
Executor：执行器，在Worker Node上启动，负责计算任务（Task）。

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

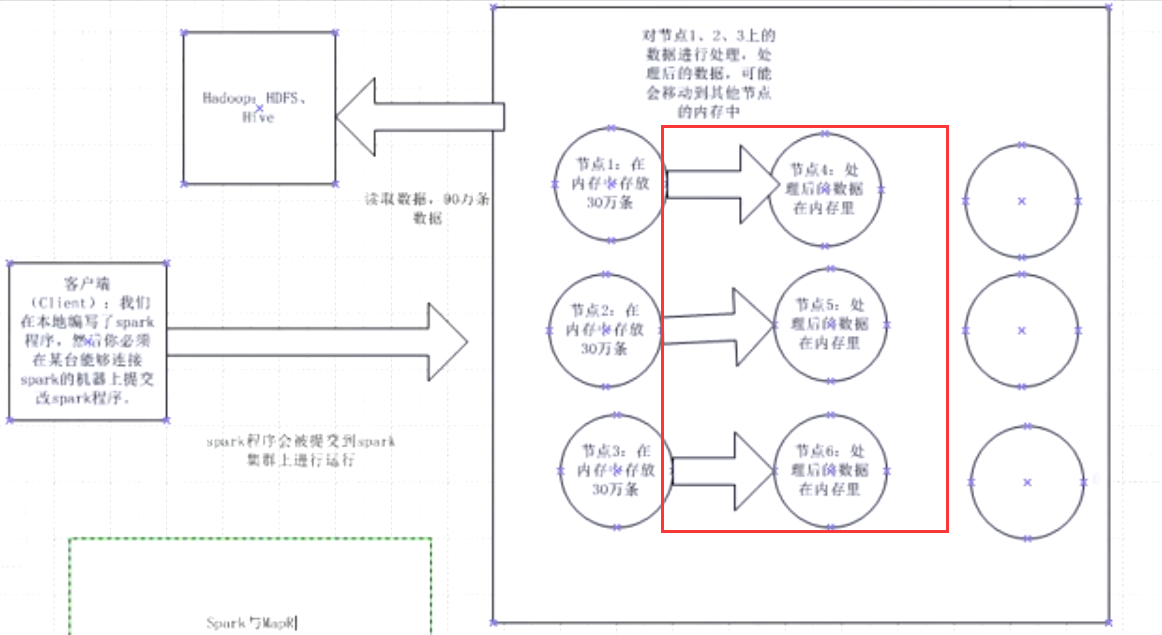
Spark读取的90W数据，不会只存储在一个节点上，而是会存储在多个Spark节点的内存中。

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

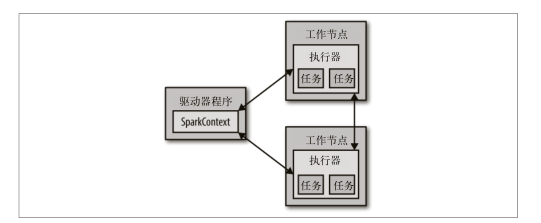
### 4.3 Spark shell和其他shell的不同点

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



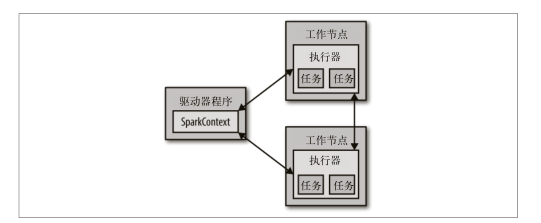
### 4.4.1 Spark驱动程序和SparkContext的关系

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）

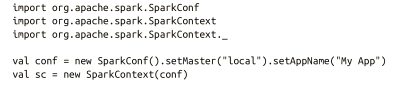
图1



### 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

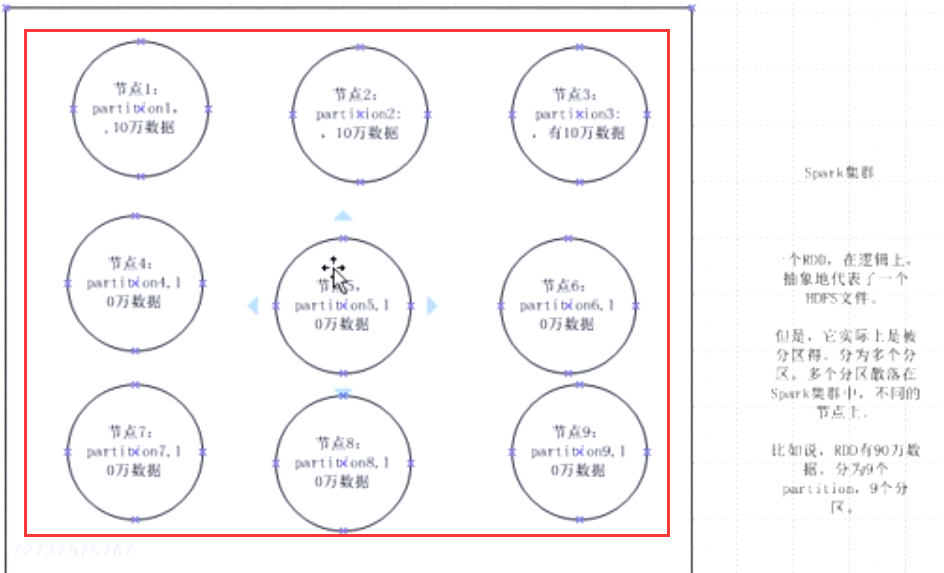
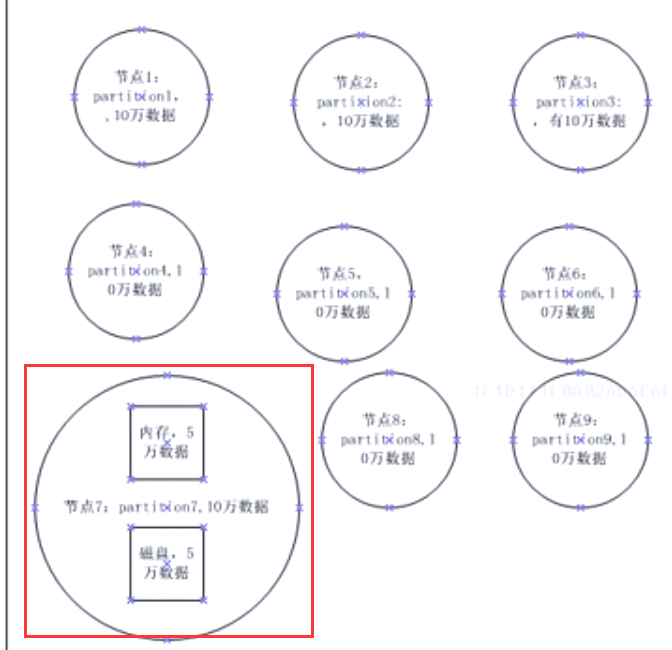


图2（弹性存储，分区部分数据存储在磁盘）



### 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) 从文件中创建RDD。

### 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。

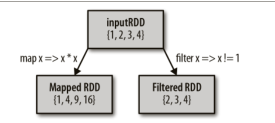
3.只有转化操作（transformation）是不会执行的，必须有行动操作（action）才会执行。

#### 4.5.3.2 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]）

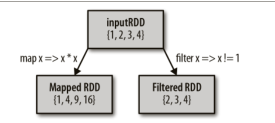
图1



#### 4.5.3.3 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中满足该函数的元素放入新的RDD中。

图1



#### 4.5.3.4 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



#### 4.5.3.5 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）

#### 4.5.3.6 union，intersaction，subtract，cartesian

union是并集，对2个RDD求并集形成一个新的RDD。

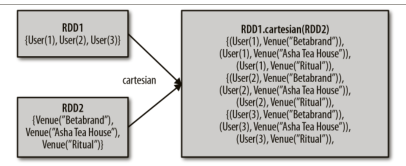
intersaction是交集，对2个RDD求交集形成一个新的RDD。

（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）



#### 4.5.3.7 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，返回一个新元素。

（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）

图1（对RDD中所有元素进行求和）



#### 4.5.3.8 collect

collect()将整个RDD的内容返回，实际上是将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。

#### 4.5.3.9 take

take(n)会取RDD中的n个元素，并且会尝试访问尽量少的分区。

#### 4.5.3.10 foreach

foreach会对RDD中的每个元素进行操作，但不返回给驱动程序。

#### 4.5.3.11 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



### 4.5.4 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

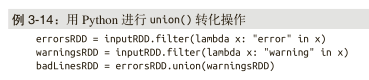
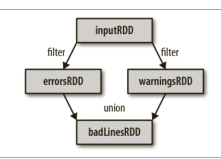


图2（谱系图）



### 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。

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

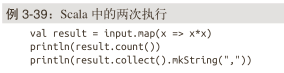
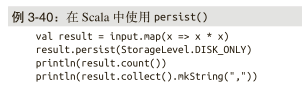


图2（缓存结果）



### 4.5.7 Pair RDD

#### 4.5.7.1 创建Pair RDD

通过map方法传入一个返回类型是键值对的函数。

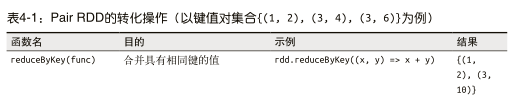
图1



#### 4.5.7.2 reduceByKey

reduce接收一个函数（(x,y)=>x+y），把这个函数应用于Pair RDD中的2个key相等的元素，合并一个新元素。

图1



#### 4.5.7.3 mapValues

mapValues接收一个函数（x=>x+1），把这个函数应用于Pair RDD中的每个元素的value，但不改变每个元素的key。

图1



图2



#### 4.5.7.4 join

join对2个RDD进行内连接，将2个RDD中的2个key相等的元素，合并成一个新元素。

图1



图2



### 4.5.8 常见大型数据源

#### 4.5.8.1 文本文件

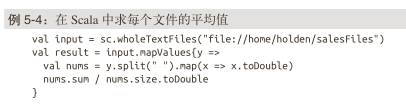
从单个文件读取时，输入的每一行会变成RDD的一个元素。

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

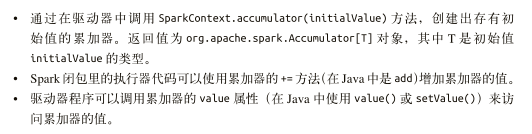


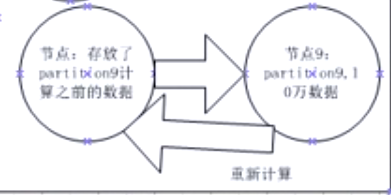
图2（例子）



### 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）



## 4.6 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

