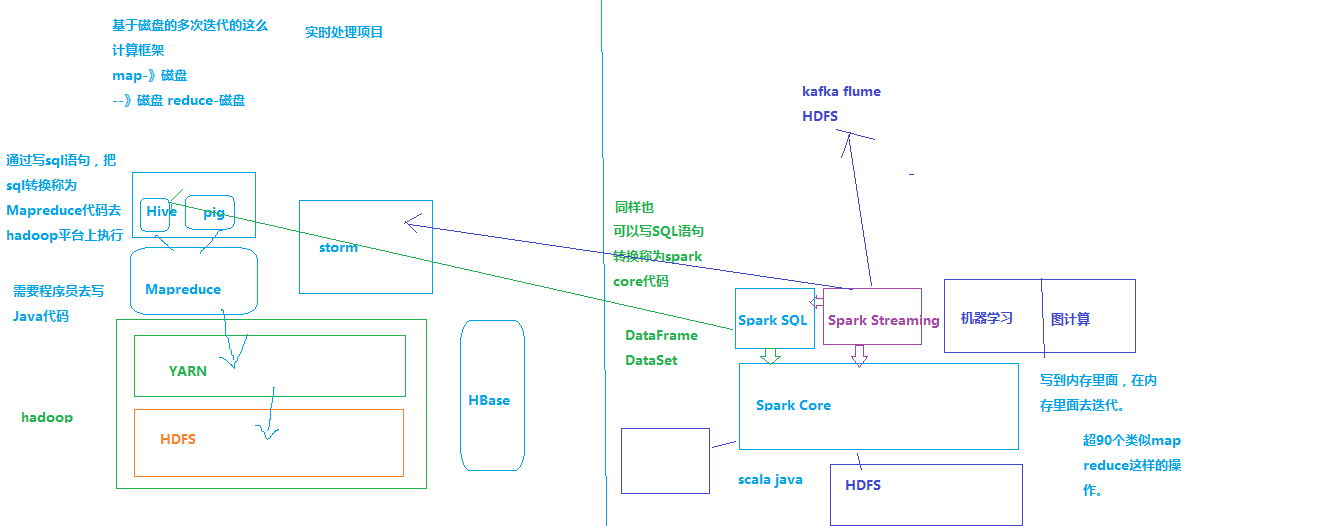
# 1.Spark 内核深度剖析

## 第1课时 Spark概述（四大特性）

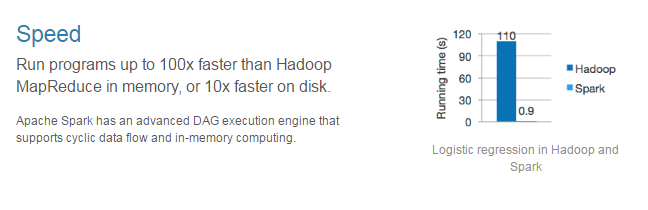
### 1.什么是spark？

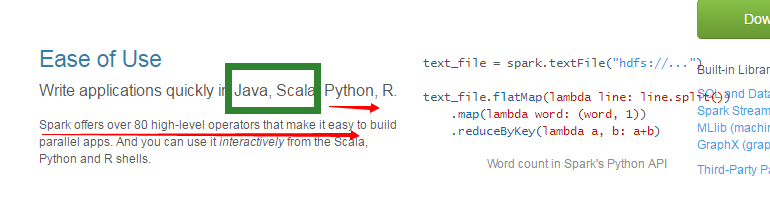
**Apache Spark™** is a fast and general engine for large-scale data processing.

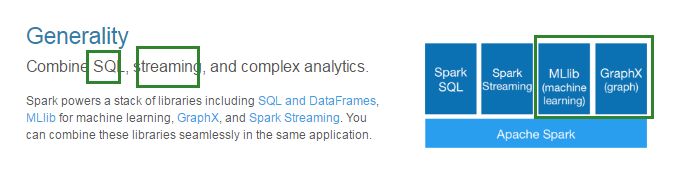
与mapreduce比较：比mapreduce快

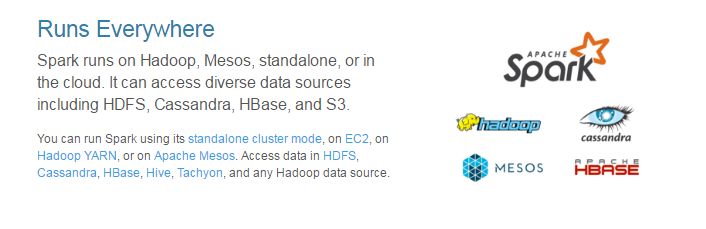


### 2.Spark的四大特性？









Spark 四种部署模式：**hadoop（spark on yarn 用yarn资源管理器来管理spark的资源**

**这也是国内使用最多的模式。）**Mesos（也是一个类似于yarn的资源管理器，但是这个资源管理器，在国内用的不多，大多数还是国外在使用。）**Standalone 模式（spark自己来管理资源，这也是用得比较多的一种模式）**，还有一种模式就是能部署云端。

It can access diverse data sources including HDFS, Cassandra（与hbase类似，国内使用的不多，国外使用的人多）, HBase, and S3. [Hive](http://hive.apache.org/), [Tachyon](http://tachyon-project.org/)（基于内存的分布式的文件系统）

<http://www.alluxio.org/>

## 第2课时 Spark快速使用

**简单使用**

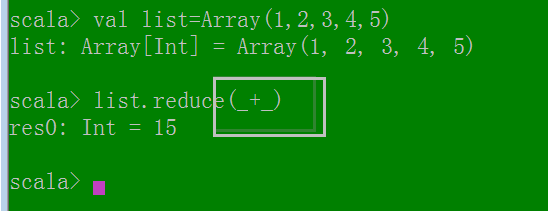
val a=0; 修饰的，这个变量的值不可更改。

var b=0; 修饰的，这个变量值可以更改。

val conf=new SparkConf();s

val sc=new SparkContext(conf);

sc 这个是所有spark程序的入口，



Spark编程：

1：简单，便捷

2：运行的速度很快

## 第3课时 什么是RDD？

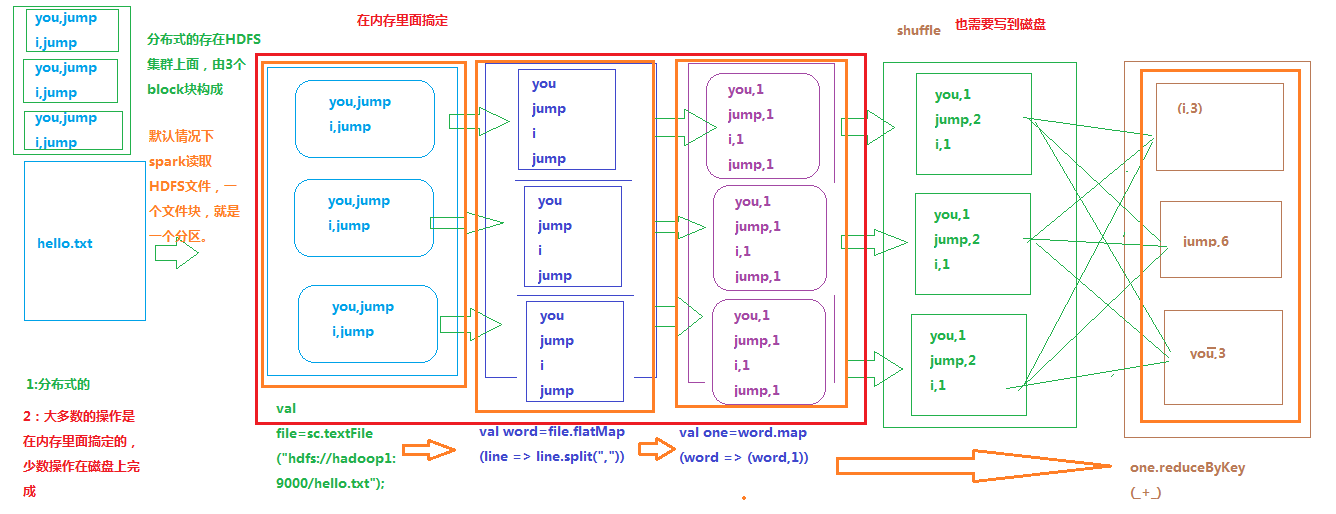
1、RDD是Spark提供的核心抽象，全称为Resillient Distributed Dataset，即**弹性**分布式数据集。

2、RDD在抽象上来说是一种元素集合，包含了数据。它是被分区的，分为多个分区，每个分区分布在集群中的不同节点上，从而让RDD中的数据可以被并行操作。（分布式数据集）

3、RDD通常通过Hadoop上的文件，即HDFS文件或者Hive表，来进行创建；有时也可以通过应用程序中的集合来创建。

4、RDD最重要的特性就是，提供了容错性，可以自动从节点失败中恢复过来。即如果某个节点上的RDD partition，因为节点故障，导致数据丢了，那么RDD会自动通过自己的数据来源重新计算该partition。这一切对使用者是透明的。

5、RDD的数据默认情况下存放在内存中的，但是在内存资源不足时，Spark会自动将RDD数据写入磁盘。（弹性 ==灵活）



## 第4课时 Spark架构

**Driver**

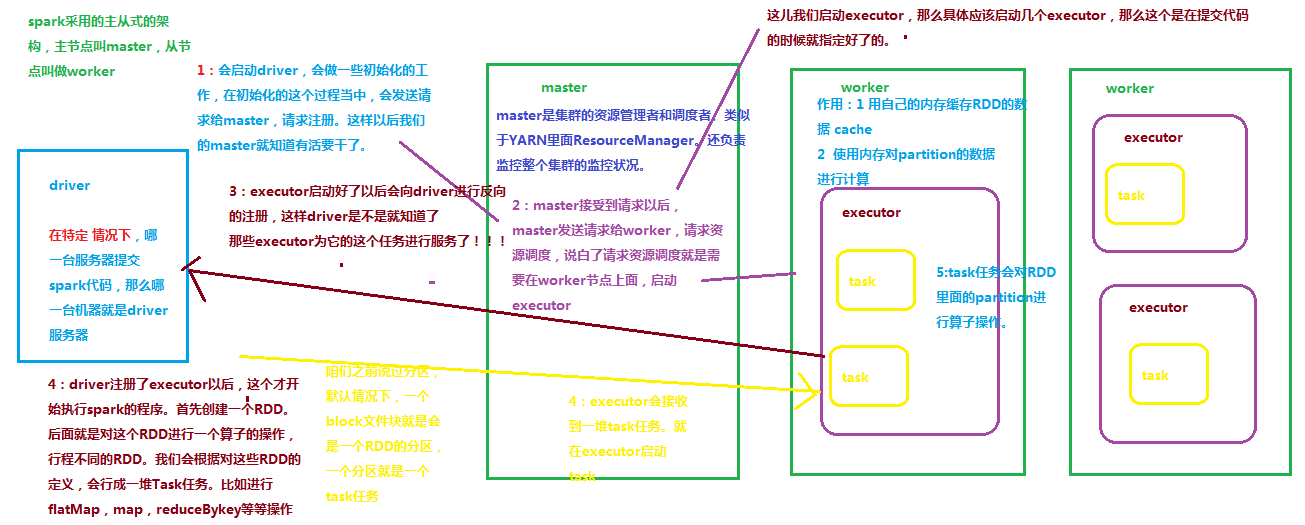
**Master**

**Worker**

**Executor**

**Task**

**Spark --（standalone模式）**



## 第5课时 linux环境准备（虚拟机，linux）

### 1.使用VMWare 虚拟机

如何安装：

解压提供给大家的压缩包

然后一直下一步下一步即可。大约需要5分钟就可安装好。安装好了以后就可以在桌面看导入下的快捷方式：

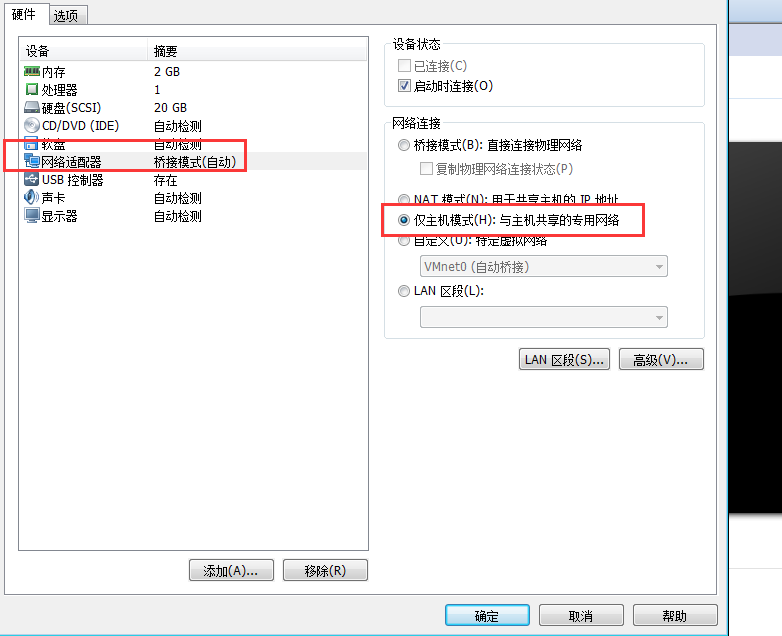


### 2.Linux - centOS，redhat 红帽（64）

提供给大家的linux提前已经格式化好了，大家直接导入虚拟机使用即可。

1：设置一下内存

2：设置一下网络

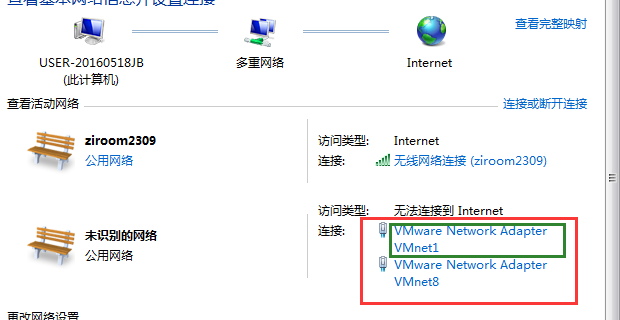


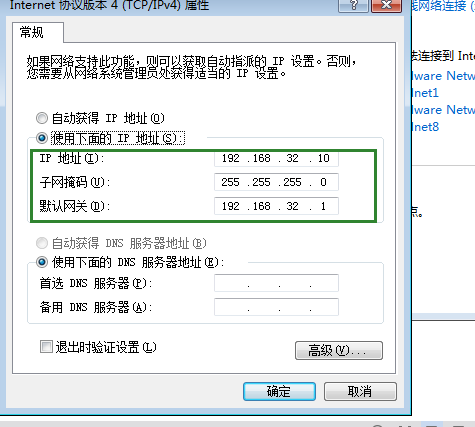
需要我们配置成windows操作系统上CRT或者PTTV直接可以连到虚拟机里面linux上。

需要网络。

需要设置虚拟机的网络和linux的网络

**虚拟机的网络（IP）**



s

**Linux的网络(IP)**

如何去配置？

Linux的IP与虚拟机的IP地址在同一网段即可。

比如虚拟机：192.168.32.10

Linux：192.168.32.110 0-255

**修改主机名**

hosts

**关闭防火墙(hadoop组件之间 spark组件之间 需要通信，端口 )**

service iptables stop

下一次linux重启防火墙也依然关闭

chkconfig iptables off

**给linux做免密码**

1：生产公钥和私钥

ssh-keygen

2： mv id\_rsa.pub authorized\_keys

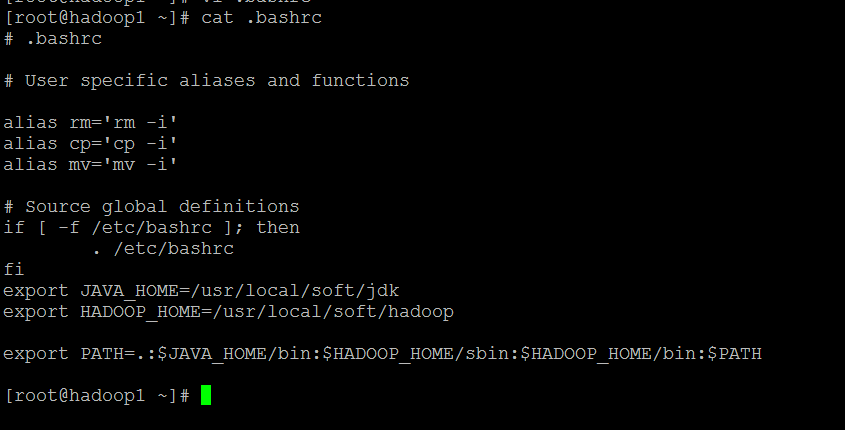
## 第6课时 hadoop环境准备

### Hadoop搭建





设置环境变量



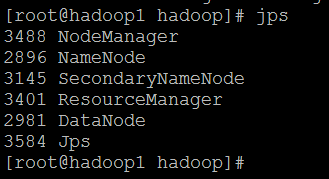
### HDFS



文件系统--启动它--需要格式化

### YARN





## 第7课时 spark环境准备

前提：安装hadoop

scala

Spark

Master和worker在同一台上面，因为这是学习环境。真实生产环境不会让master和worker在同一台机器上面。

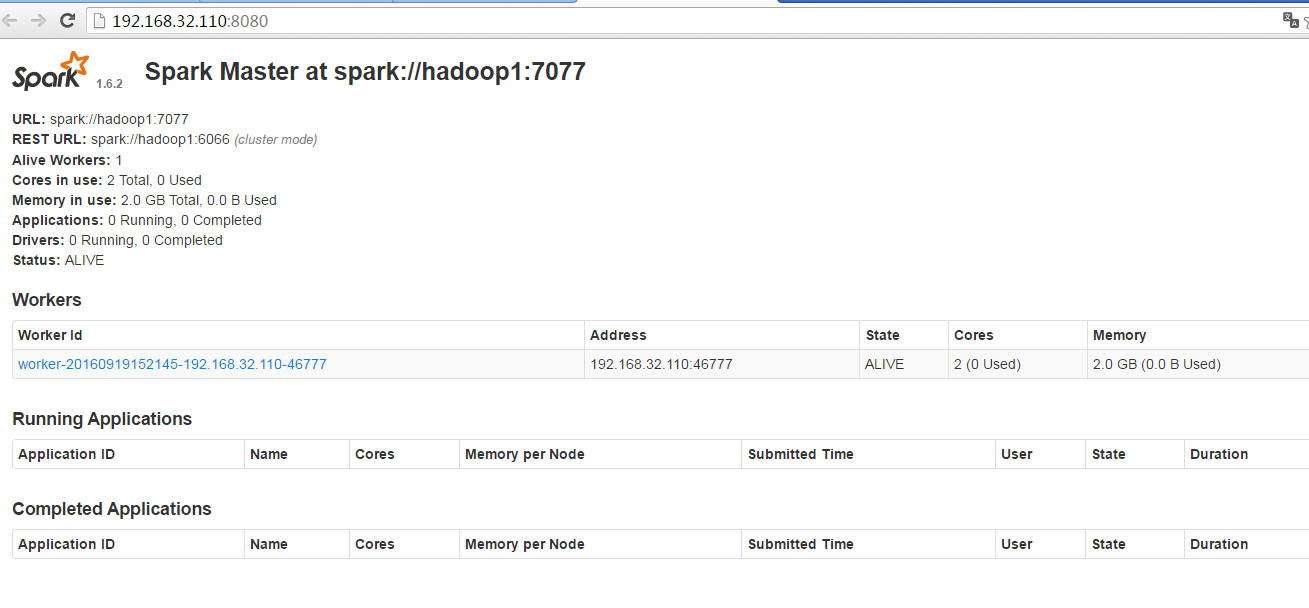
100G 85%

Spark\_env.sh=master worker core memory

Slaves=worker

页面浏览地址

192.168.32.110:8080



## 第8课时 Spark开发环境搭建（java，scala）

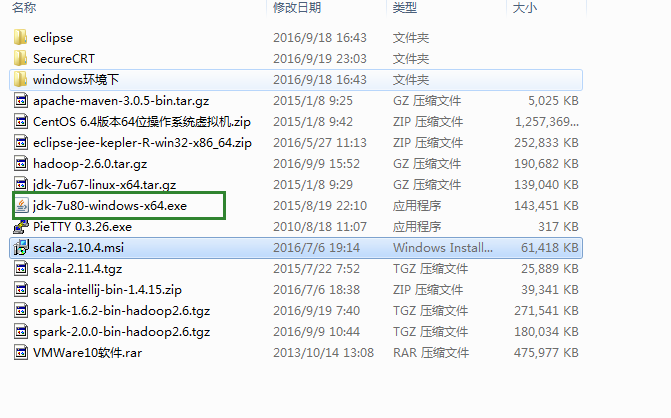
### Java环境

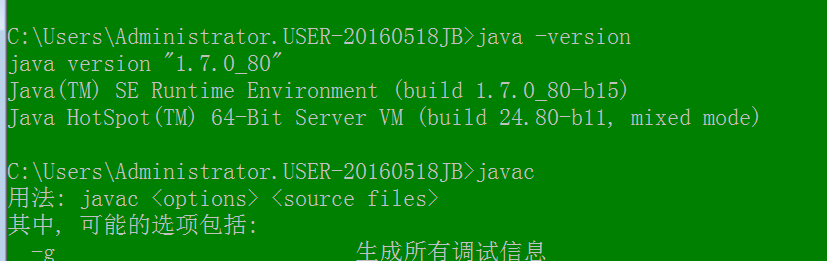
1：安装JDK

2：安装maven

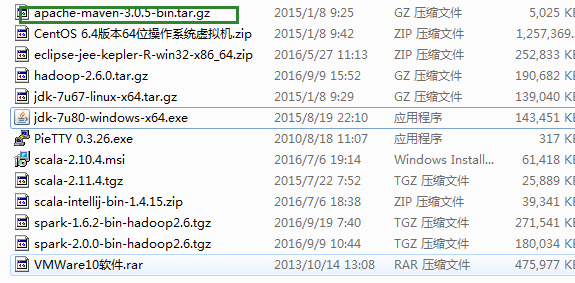
3：安装eclipse

4：配置eclipse maven环境

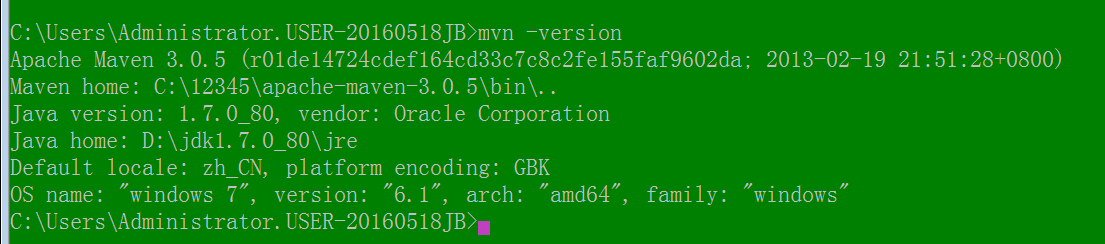




Maven解压出来就可以使用

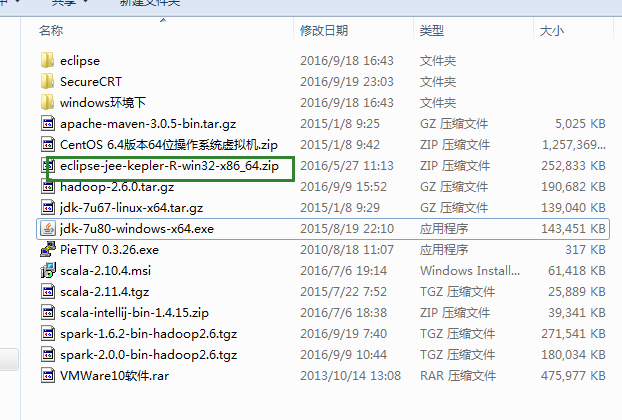


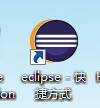
验证maven（maven要安装成功，必须先安装好jdk）



解压出来，点击.exe的软件，直接下一步下一步即可安装好，大约需要5分钟

安装出来这个如果要能正常使用，比如提前安装好jdk。





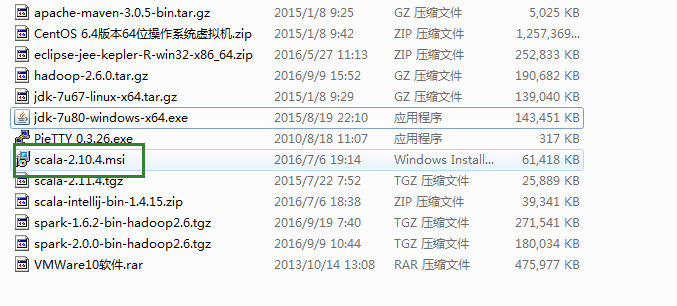
### Scala环境

1：安装scala

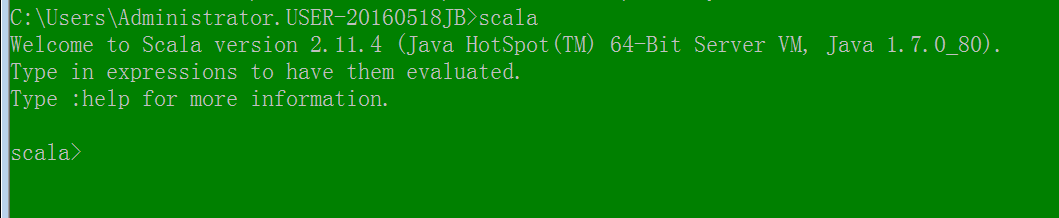
2：安装eclipse

3：配置eclipse maven环境

安装scala之前要安装好JDK，因为scala的代码也是运行在jvm里面的。



直接下一步下一步安装好了，以后，配置一下环境变量，道理跟安装JDK一样。



如果能进来说明scala就安装好了。

配置maven ：跟之前的一模一样

创建一个maven的项目（scala）

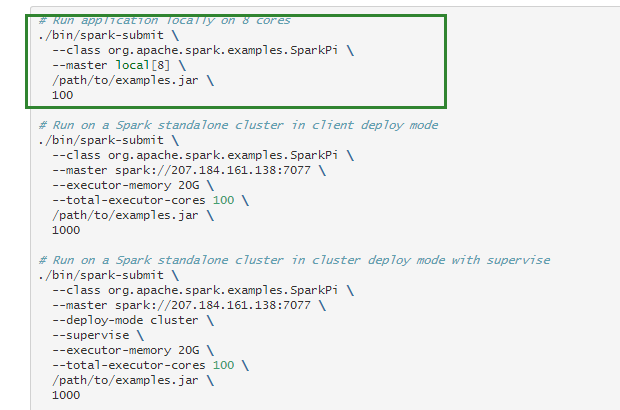
Group id : org.scala-tools.archetypes

Artifact id : scala-archetype-simple

Version : 1.2

## 第9课时 Spark任务提交

### 本地运行spark任务--local



### 集群运行spark任务--standalone



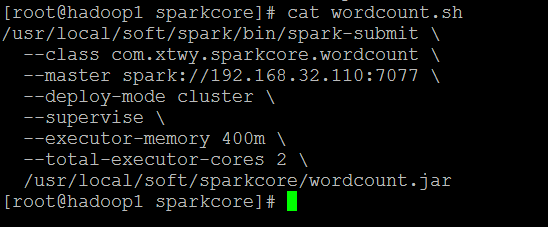
在standalone模式下：

Client：

不指定deploy-mode ,默认就是client模式，也就是哪一台服务器提交spark代码，那么哪一台就是driver服务器。

**Cluster模式：**

我们需要指定，driver服务器并不是提交代码的那一台服务器，而是在提交代码的时候，在worker主机上，随机挑选一台作为driver服务器，那么如果提交10个应用，那么就有可能10台driver服务器。



## 第10课时 Historyserver配置

如果spark记录下了一个作业生命周期内的所有事件，那么就会在该作业执行完成之后，我们进入其web ui时，自动用记录的数据重新绘制作业的web ui。

有3个属性我们可以设置

**spark-defaults.conf**

spark.eventLog.enabled true

spark.eventLog.dir hdfs://192.168.32.110:9000/spark-events

spark.eventLog.compress true

**spark-env.sh**

export SPARK\_HISTORY\_OPTS="-Dspark.history.ui.port=18080 -Dspark.history.retainedApplications=250 -Dspark.history.fs.logDirectory=hdfs://192.168.32.110:9000/spark-events"

务必预先创建好hdfs://192.168.0.103:9000/spark-events目录

而且要注意，spark.eventLog.dir与spark.history.fs.logDirectory指向的必须是同一个目录，spark.eventLog.dir会指定作业事件记录在哪里，spark.history.fs.logDirectory会指定从哪个目录中去读取作业数据

启动HistoryServer: ./sbin/start-history-server.sh

访问地址: 192.168.0.103:18080

## 第11课时 RDD的创建方式

进行Spark核心编程时，首先要做的第一件事，就是创建一个初始的RDD。该RDD中，通常就代表和包含了Spark应用程序的输入源数据。然后在创建了初始的RDD之后，我们接着进行各种算子操作。

大体上有两种方式创建RDD：

### 方式1：读取文件

A:通过读取HDFS上文件，创建RDD。 sc.textFile(“hdfs://”)

B:通过读取本地文件，创建RDD sc.textFile(“”);

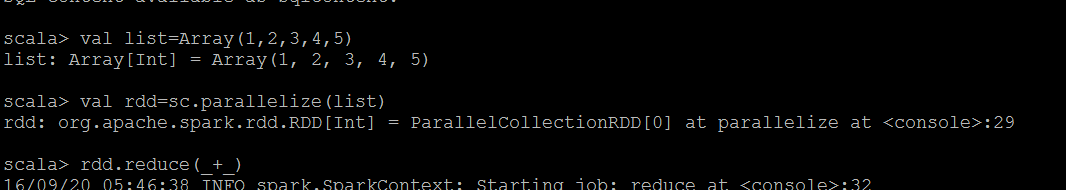
### 方式2：并行化的方式创建RDD

其实这种方式就是通过我们自己去模拟数据

val str=Array(“you jump”,”i jumps”)

val list=Array(1,2,3,4,5,6)

val listrdd=sc.parallelize(list);



其实这种方式就是方便我们去测试跟演示一些功能的时候使用，绝大多数我们使用的是第一种方式创建RDD。

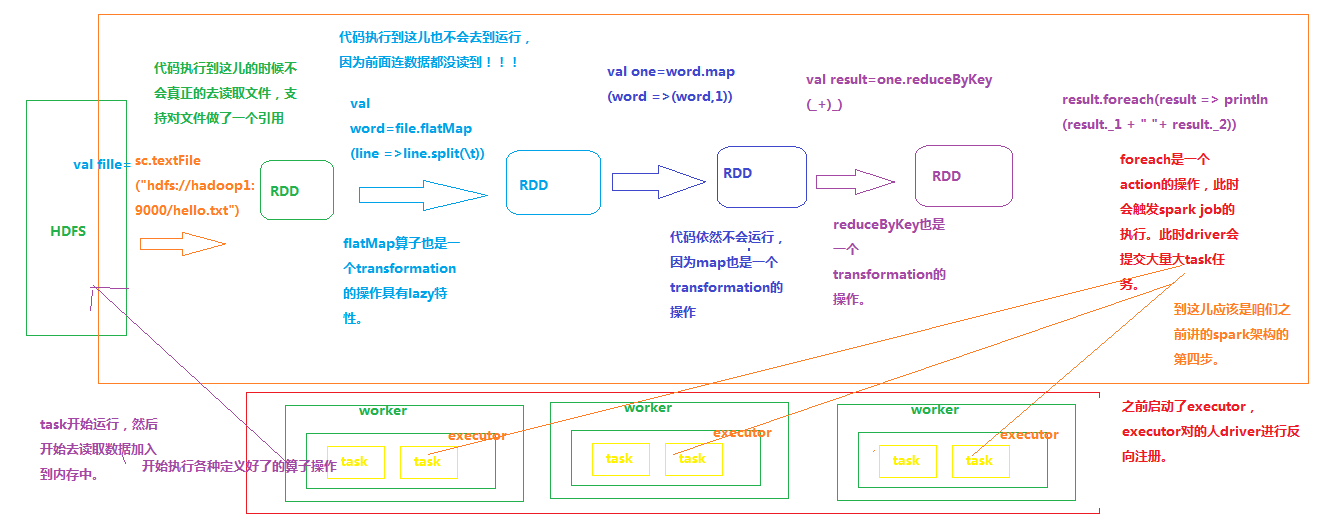
## 第12课时 Transformation和action原理剖析

Spark支持两种RDD操作：transformation和action。transformation操作会针对已有的RDD创建一个新的RDD；而action则主要是对RDD进行最后的操作，比如遍历、reduce、保存到文件等，并可以返回结果给Driver程序。

例如，map就是一种transformation操作，它用于将已有RDD的每个元素传入一个自定义的函数，并获取一个新的元素，然后将所有的新元素组成一个新的RDD。而reduce就是一种action操作，它用于对RDD中的所有元素进行聚合操作，并获取一个最终的结果，然后返回给Driver程序。

transformation的特点就是lazy特性。lazy特性指的是，如果一个spark应用中只定义了transformation操作，那么即使你执行该应用，这些操作也不会执行。也就是说，transformation是不会触发spark程序的执行的，它们只是记录了对RDD所做的操作，但是不会自发的执行。只有当transformation之后，接着执行了一个action操作，那么所有的transformation才会执行。Spark通过这种lazy特性，来进行底层的spark应用执行的优化，避免产生过多中间结果。

action操作执行，会触发一个spark job的运行，从而触发这个action之前所有的transformation的执行。这是action的特性。

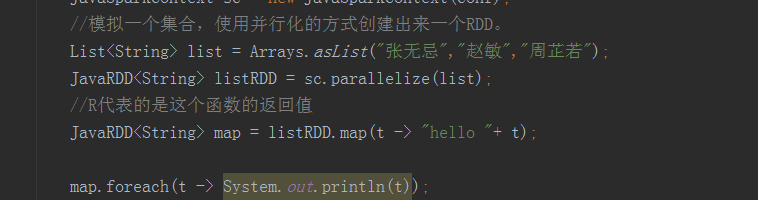


## 第13课时 Transformation和action算子演示

### Java Lambda演示Transformation算子

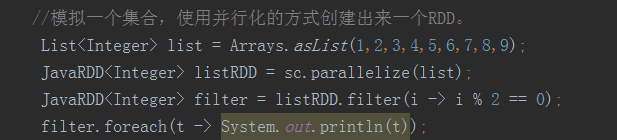
**Map:**

对调用map的RDD数据集中的每个element都使用func，然后返回一个新的RDD,这个返回的数据集是分布式的数据集



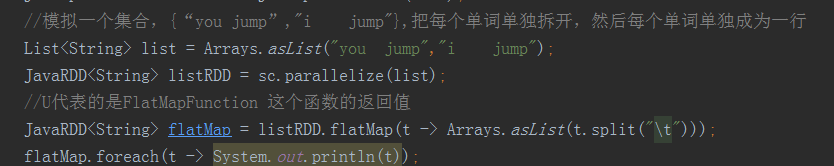
**Filter:**

对调用filter的RDD数据集中的每个元素都使用func，然后返回一个包含使func为true的元素构成的RDD



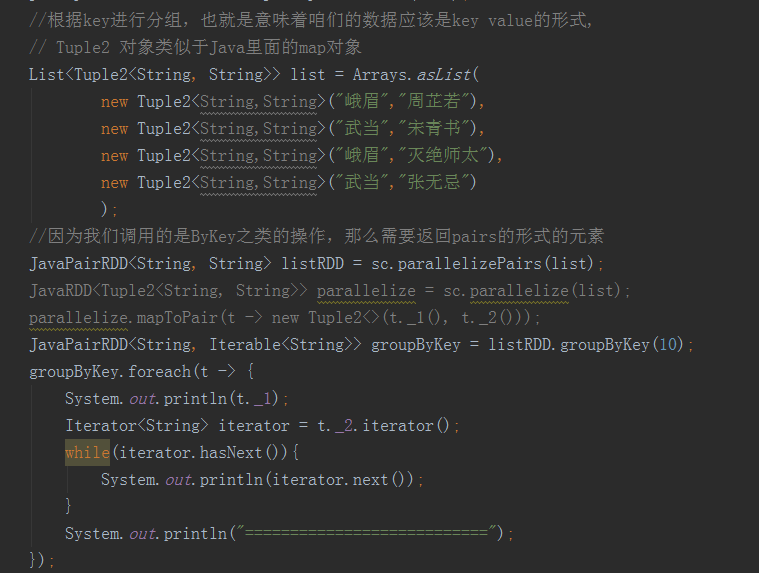
**flatMap:**

  类似于map，但是每一个输入元素，会被映射为0到多个输出元素（因此，func函数的返回值是一个Seq，而不是单一元素）



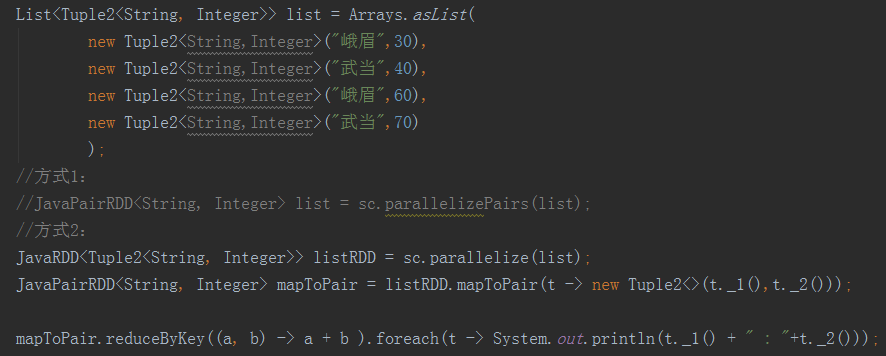
**groupByKey([numTasks])：**

在一个由（K,V）对组成的数据集上调用，返回一个（K，Seq[V])对的数据集。注意：默认情况下，使用8个并行任务进行分组，你可以传入numTask可选参数，根据数据量设置不同数目的Task。



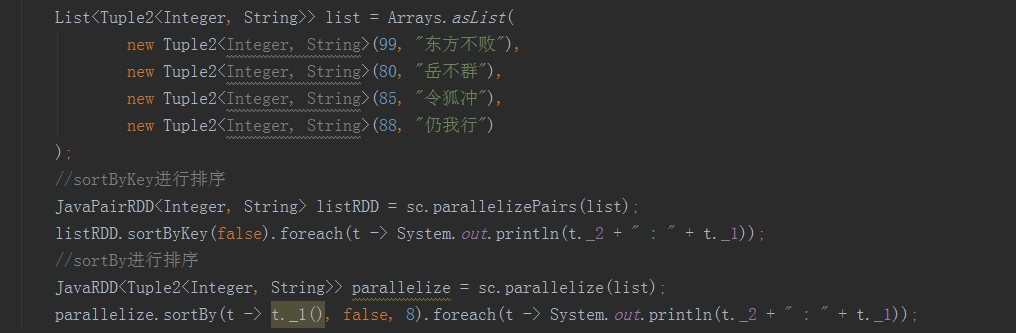
**reduceByKey(func, [numTasks]) :**

在一个（K，V)对的数据集上使用，返回一个（K，V）对的数据集，key相同的值，都被使用指定的reduce函数聚合到一起。和groupbykey类似，任务的个数是可以通过第二个可选参数来配置的。



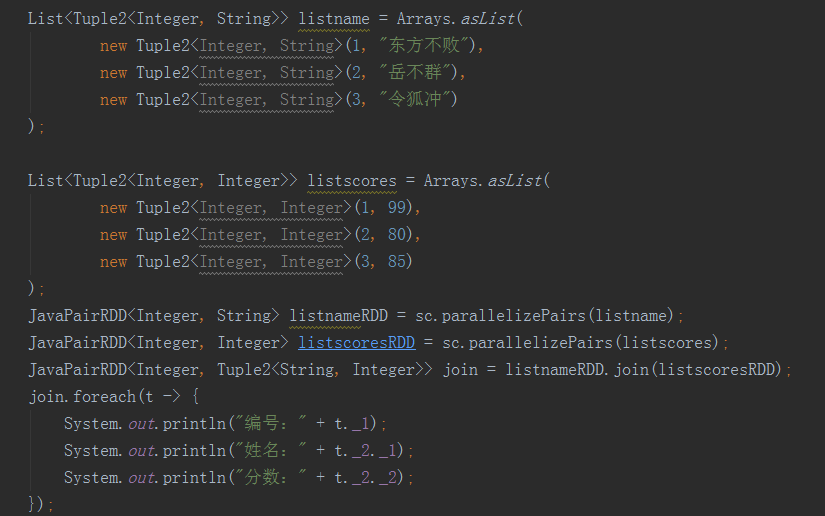
**sortByKey与sortBy：**

sortByKey函数是对PairRDD进行排序;sortBy是对标准RDD进行排序。



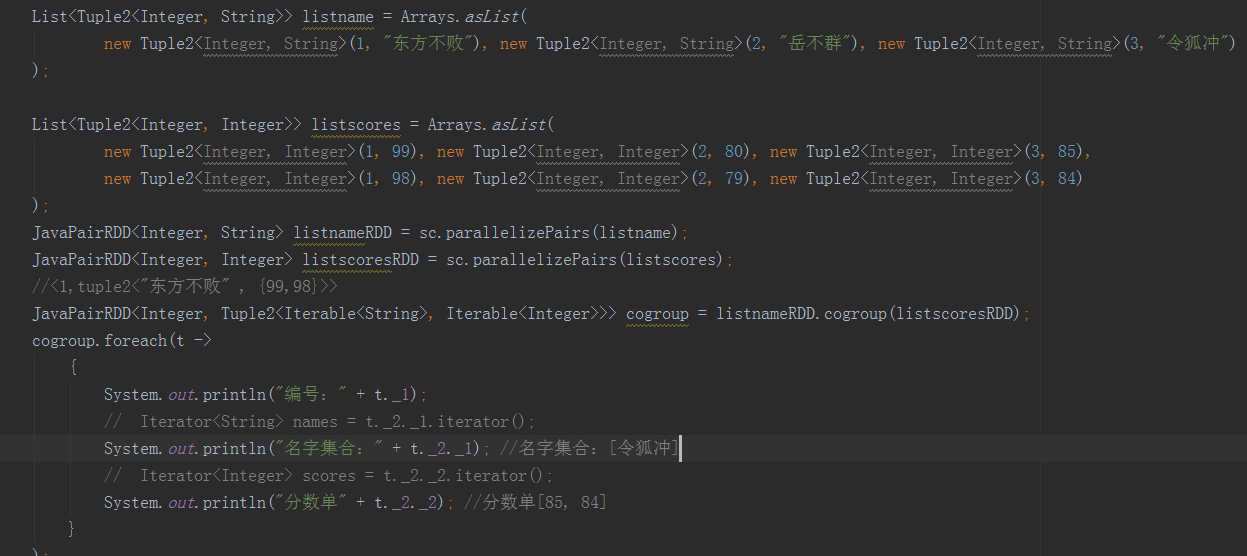
**join(otherDataset, [numTasks]) :**

在类型为（K,V)和（K,W)类型的数据集上调用，返回一个（K,(V,W))对，每个key中的所有元素都在一起的数据集



**groupWith(otherDataset, [numTasks]) :**

在类型为（K,V)和(K,W)类型的数据集上调用，返回一个数据集，组成元素为（K, Seq[V], Seq[W]) Tuples。这个操作在其它框架，称为CoGroup。



**union(otherDataset) :** 返回两个rdd的并集，不去重。

**intersection：**该函数返回两个RDD的交集，并且去重。

**distinct：**rdd元素去重。

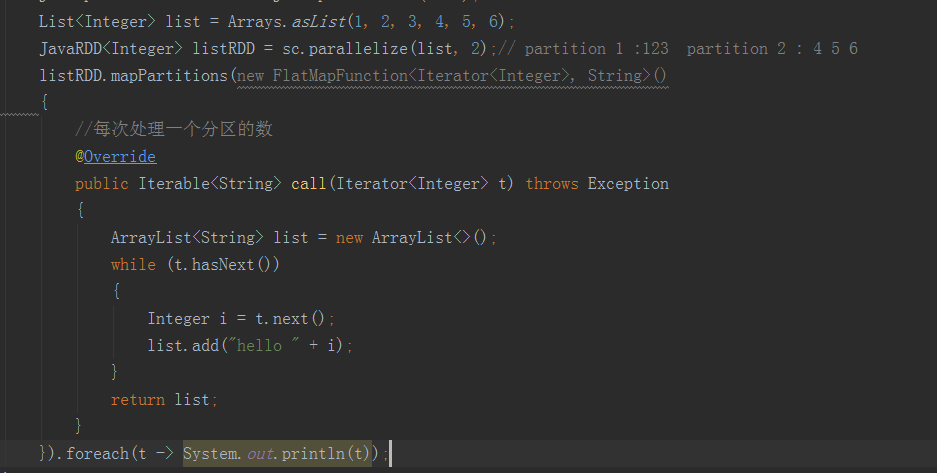
**cartesian(otherDataset) :**

笛卡尔积。在数据集T和U上调用时，返回一个(T，U）对的数据集，所有元素交互进行笛卡尔积。

**mapPartitions:**

该函数和map函数类似，只不过映射函数的参数由RDD中的每一个元素变成了RDD中每一个分区的迭代器。如果在映射的过程中需要频繁创建额外的对象，使用mapPartitions要比map高效的过。

比如，将RDD中的所有数据通过JDBC连接写入数据库，如果使用map函数，可能要为每一个元素都创建一个connection，这样开销很大，如果使用mapPartitions，那么只需要针对每一个分区建立一个connection。



使用mapPartitions时，call方法只会被调用两次，使用map方法时，call方法会被调用6次。

**reparition:**reparition是coalesce shuffle为true的简易实现

**coalesce:**

如果原来有N个partition，需要重新规划承M个partition

1）N < M 需要将shuffle设置为true。

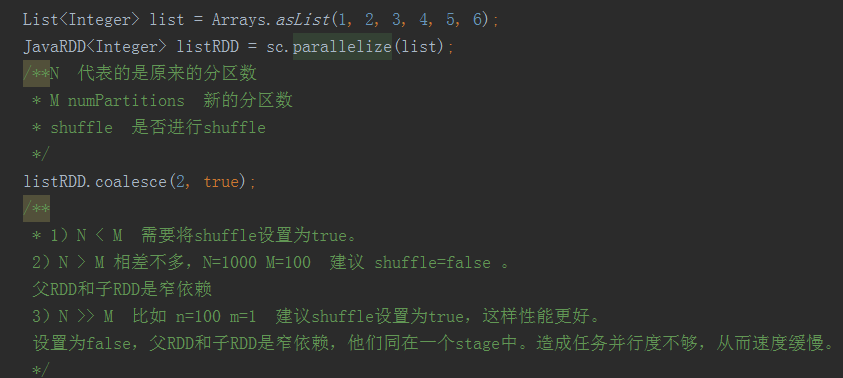
2）N > M 相差不多，N=1000 M=100 建议 shuffle=false 。

父RDD和子RDD是窄依赖

3）N >> M 比如 n=100 m=1 建议shuffle设置为true，这样性能更好。

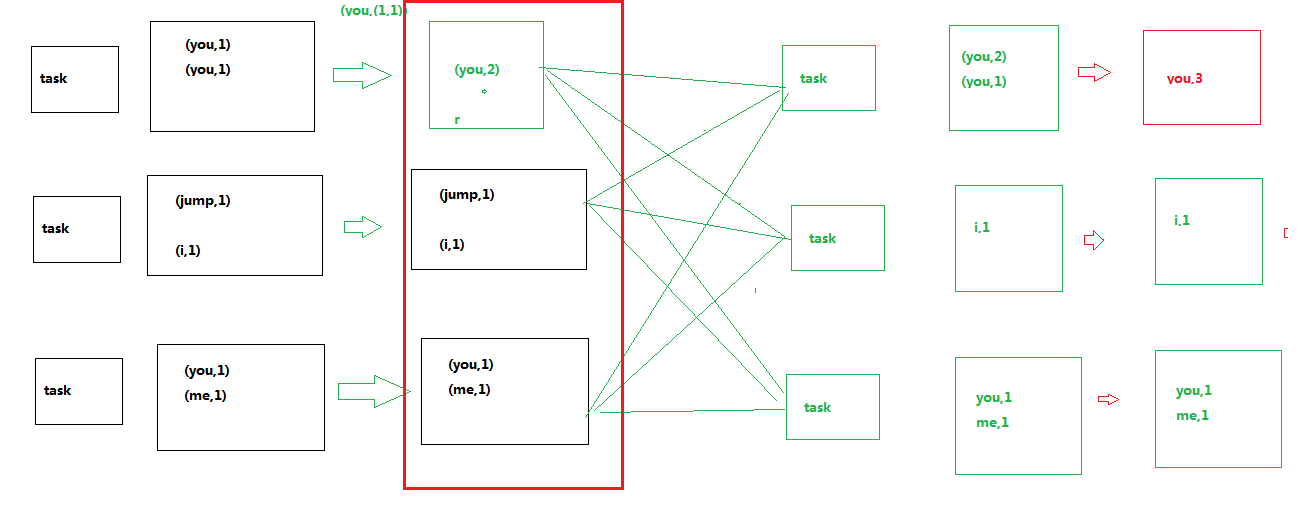
设置为false，父RDD和子RDD是窄依赖，他们同在一个stage中。造成任务并行度不够，从而速度缓慢。

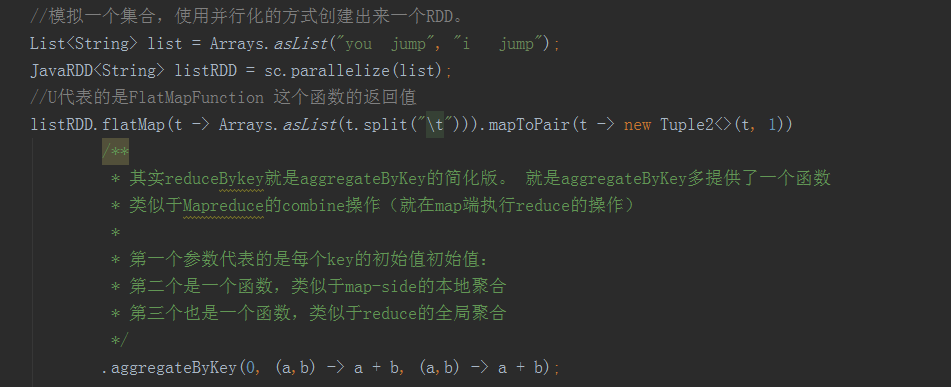
filter 过滤了以后 --partition数据量会减少，可能需要重新分区。



**Sample:**对RDD中的集合内元素进行采样，第一个参数withReplacement是true表示有放回取样，false表示无放回。第二个参数表示比例

**aggregateByKey**

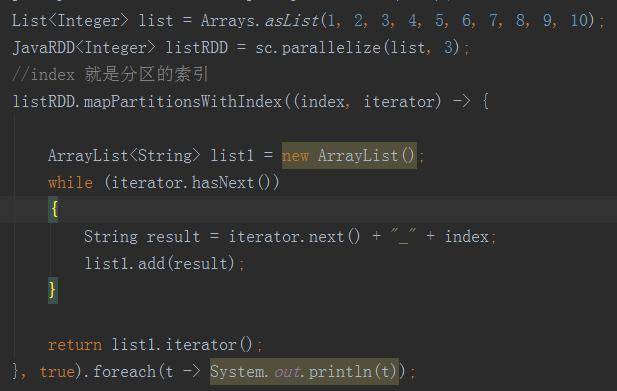




调优算子，比reduceByKey多提供了一个参数，用于map阶段进行本地聚合操作，即上图红框中的部分。在map阶段进行聚合之后，可以使传到reduce端的数据变少，提高性能。

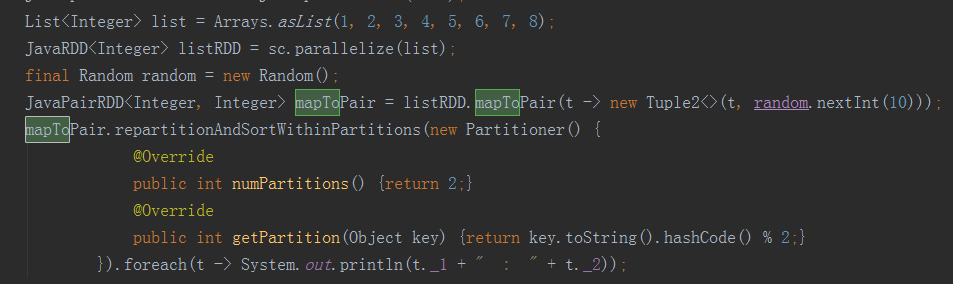
**mapPartitionsWithIndex:**

mapPartitionsWithIndex与mapPartitions基本相同，只是在处理函数的参数是一个二元元组，元组的第一个元素是当前处理的分区的index，元组的第二个元素是当前处理的分区元素组成的Iterator



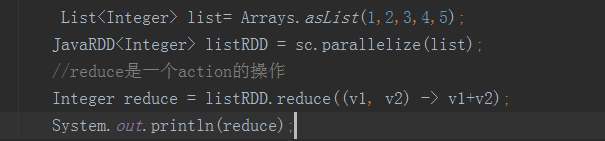
**repartitionAndSortWithinPartitions:**

**该方法依据partitioner对RDD进行分区，并且在每个结果分区中按key进行排序；通过对比sortByKey发现，这种方式比先分区，然后在每个分区中进行排序效率高，这是因为它可以将排序融入到shuffle阶段。**

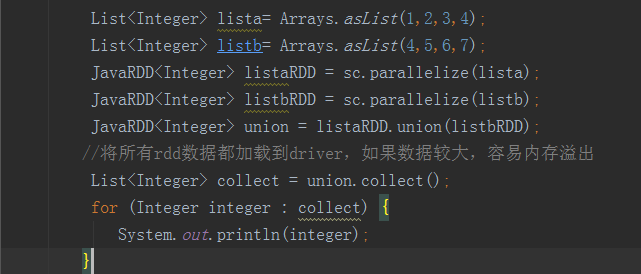


### Java Lambda演示action算子

**reduce:** 聚集，传入的函数是两个参数输入返回一个值。



**collect()：**一般在filter或者足够小的结果的时候，再用collect封装返回一个数组。



**foreach(func):**对dataset中的每个元素都使用func

**count():**返回的是dataset中的element的个数。

**first():**返回的是dataset中的第一个元素。

**take(n):**返回前n个elements。

**takeOrdered(n):**升序排序后返回前n个elements。

**top(n):**降序排序后返回前n个elements。

**saveAsTextFile（path）：**把dataset写到一个textfile中，或者hdfs，或者hdfs支持的文件系统中，Spark把每条记录都转换为一行记录，然后写到file中。如果保存到本地文件夹会报NullPointException。

**saveAsSequenceFile(path):**只能用在key-value对上，然后生成SequenceFile写到本地或者hadoop文件系统

**countByKey()：**返回的是一个map，value是key对应的个数。

**takeSample(withReplacement，num，seed)：**抽样返回一个dataset中的num个元素，随机种子seed。

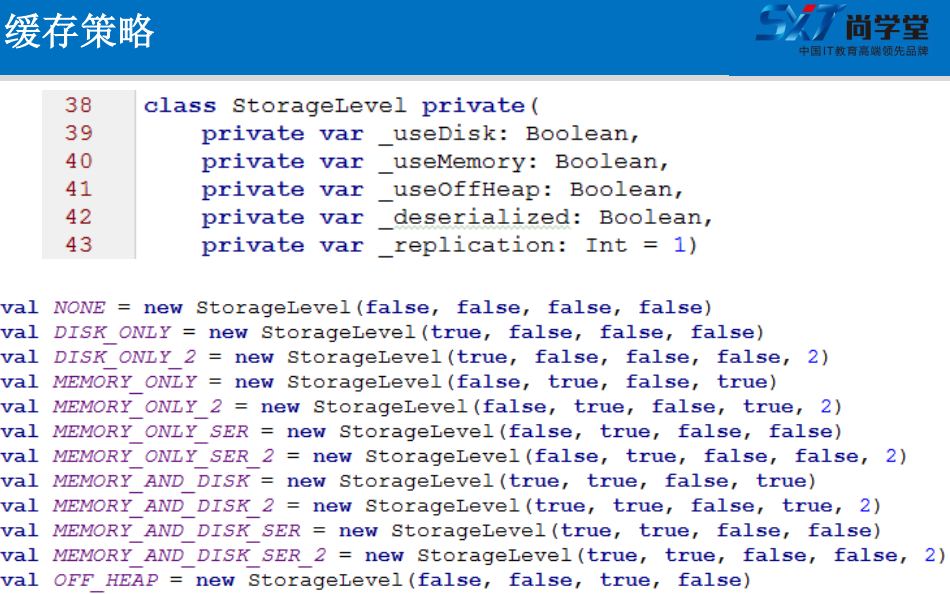
## 第14课时 RDD持久化

### RDD持久化

将数据通过操作持久化（或缓存）在内存中是Spark的重要能力之一。当你缓存了一个RDD，每个节点都缓存了RDD的所有分区。这样就可以在内存中进行计算。这样可以使以后在RDD上的动作更快（通常可以提高10倍）。

可以对希望缓存的RDD通过使用persist或cache方法进行标记。它通过动作操作第一次在RDD上进行计算后，它就会被缓存在节点上的内存中。Spark的缓存具有容错性，如果RDD的某一分区丢失，它会自动使用最初创建RDD时的转换操作进行重新计算。

另外，RDD可以被持久化成不同的级别。比如，可以允许你存储在磁盘，内存，甚至是序列化的Java对象（节省空间），备份在不同的节点上，或者存储在基于内存的文件系统Tachyon上。通过向persist()方法传递StorageLevel对象来设置。cache方法是使用默认级别StorageLevel.MEMORY\_ONLY的方法。



MEMORY\_ONLY:当文件为1G，内存为512M时，只加载512M数据进入内存，另外的数据留在原来的文件中，如hdfs中。加载rdd时，一半数据来自内存，一半数据来自hdfs的原文件中。

MEMORY\_ONLY\_DISK:当文件为1G，内存为512M时，只加载512M数据进入内存，另一半加载到服务器的磁盘中，加载rdd时，一半数据来自内存，一半数据来自服务器磁盘。

**选持久化方案建议：**

1：优先选择MEMORY\_ONLY，如果可以用内存缓存所有的数据，那么也就意味着我的计算是纯内存的计算，速度当然快。

2：MEMORY\_ONLY 缓存不了所有的数据，MEMORY\_ONLY\_SER 把数据实现序列化然后进行存储。这样也是纯内存操作，速度也快，只不过需要耗费一点cpu资源需要反序列化。

3：使用备份级别，如果需要更快的恢复。所有的存储级别都通过重新计算提供了全面的容错性，但是备份级别允许你继续在RDD上执行任务而无需重新计算丢失的分区。

4：能不能使用DISK的，就不使用DISK，有时候从磁盘读，还不如从新计算一次。

### tachyon

1：什么是tachyon

是一个内存分布式文件系统。它是介于计算层和存储层之间的，我们可以简单的理解为存储层在内存内的一个缓存系统。它是一个开源的系统，是一个以JVM为base的系统。

<http://www.alluxio.org/>

2:为什么会出现tachyon

发现已内存去替换磁盘，就可以明显的减小延时，所以涌现出来很多基于内存的计算工具，比较出名的就是咱们的讲的spark这个计算框架。

**Spark使用内存代替磁盘时存在的问题：**

1. spark运行已JVM为基础，所以spark的任务会把数据存入JVM的堆中，随着计算的迭代，JVM堆中存放的数据量迅速增大，对于spark而言，spark的计算引擎和存储引擎处在同一个JVM中，所以会有重复的GC方面的开销。这样就增大了系统的延时。

2.当JVM崩溃时，缓存在JVM堆中的数据也会消失，这个时候spark不得不根据RDD的血缘关系重新计算数据。

3.如果spark需要其他的框架的共享数据，比如就是hadoop的Mapreduce，这个时候就必须通过第三方来共享，比如借助HDFS，那么这样的话，就需要额外的开销，借助的是HDFS，那么就需要磁盘IO的开销。

因为我们基于内存的分布式计算框架有以上的问题，那么就促使了内存分布式文件系统的诞生，比如tachyon。Tachyon可以解决spark的什么问题呢？

如果我们把数据存放到tachyon上面：

1：减少Spark GC的开销。

2：当spark 的JVM崩溃的时候，存放在tachyon上的数据不受影响。

3：spark如果要想跟被的计算工具共享数据，只要通过tachyon的Client就可以做到了。并且延迟远低于HDFS等系统。

## 第15课时 共享变量

Spark执行不少操作时都依赖于闭包函数的调用，此时如果闭包函数使用到了外部变量，会进行一系列操作：

　　1、驱动程序使将闭包中使用变量封装成对象，驱动程序序列化对象，传给worker节点任务；

　　2、worker节点任务接收到对象，执行闭包函数；

由于使用外部变量势必会通过网络、序列化、反序列化，如外部变量过大或过多使用外部变量将会影响Spark程序的性能；

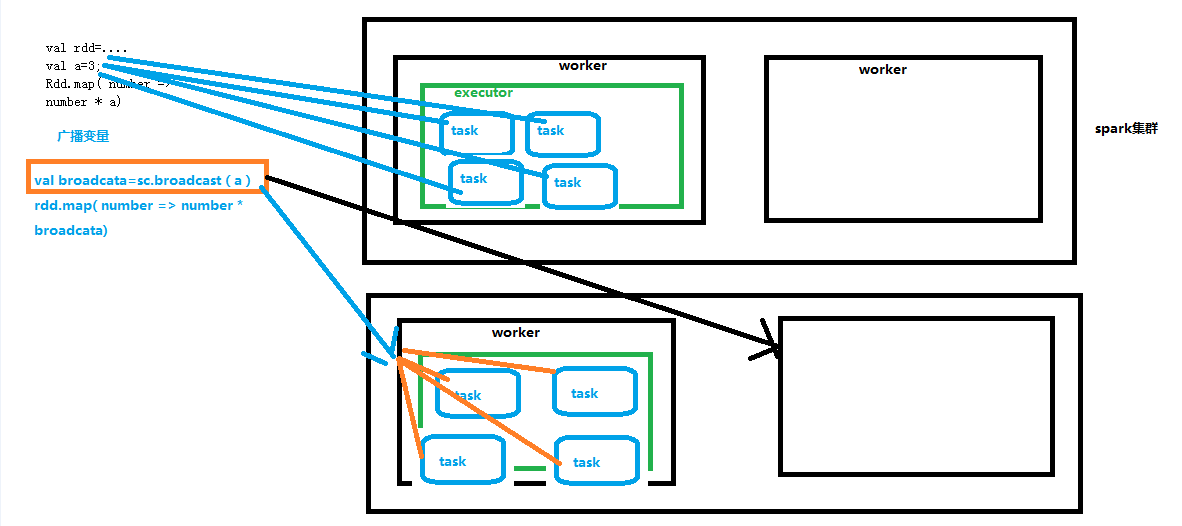
　　Spark提供了两种类型的共享变量（Shared Variables）：广播变量（Broadcast Variables）、累加器（Accumulators ）；

### 广播变量（Broadcast Variables）

　　Spark提供的广播变量可以解决闭包函数引用外部大变量引起的性能问题；广播变量将只读变量缓存在每个worker节点中，Spark使用了高效广播算法分发变量从而提高通信性能；如直接在闭包函数中使用外部变量，该变量会缓存在每个任务（jobTask）中，如果多个任务同时使用了一个大变量势必会影响到程序性能；

　　广播变量：每个worker节点中缓存一个副本，通过高效广播算法提高传输效率，广播变量是只读的；

　　Spark Scala Api与Java Api默认使用了Jdk自带序列化库，通过使用第三方或使用自定义的序列化库还可以进一步提高广播变量的性能；



### 累加器**（Accumulators）**

累加器可以使得worker节点中指定的值聚合到驱动程序中，如统计Spark程序执行过程中的事件总数等；

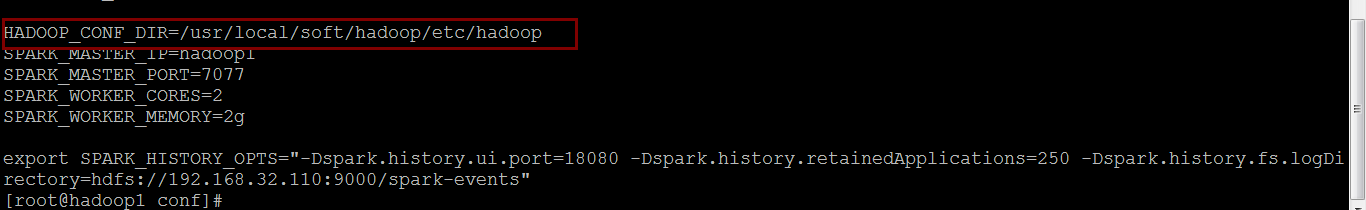
在使用累加器（Accumulators）时需要注意，只有在action操作中才会触发累加器，也就是说上述代码中由于flatMap()为transformation操作，因为Spark惰性特征所以只用当saveAsTextFile() 执行时累加器才会被触发；累加器只有在驱动程序中才可访问，worker节点中的任务不可访问累加器中的值。

## 第16课时 Spark on YARN模式（cluster,client）

Standalone-->master worker -->client cluster

Yarn --client cluster

配置spark on yarn 只需要配置如下参数即可。



使用YARN模式的时候，不需要启动master和worker了。

只需要启动HDFS和YARN即可。

使用如下命令启动HDFS：start-dfs.sh

使用如下命令启动YARN：start-yarn.sh

我们spark的开发跟以前一模一样，没有任何的变换。

我们是需要在提交任务的时候指定一下运行的模式即可。

./bin/spark-submit **\**

--class org.apache.spark.examples.SparkPi **\**

--master yarn **\**

--deploy-mode **cluster** **\**  *# can be client for client mode*

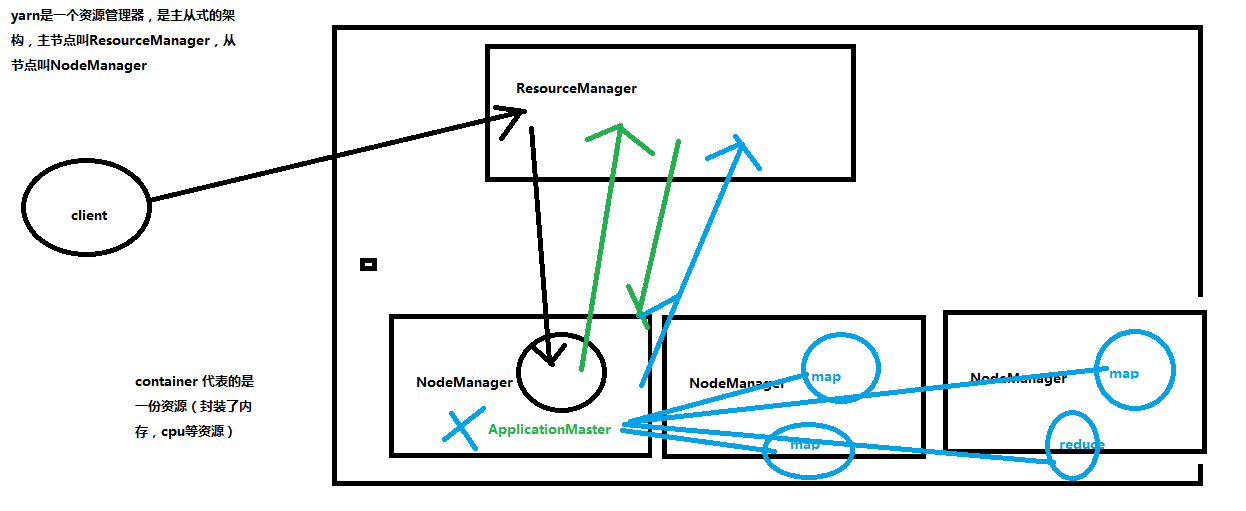
--executor-memory 20G **\**

--num-executors 50 **\**

/path/to/examples.jar **\**

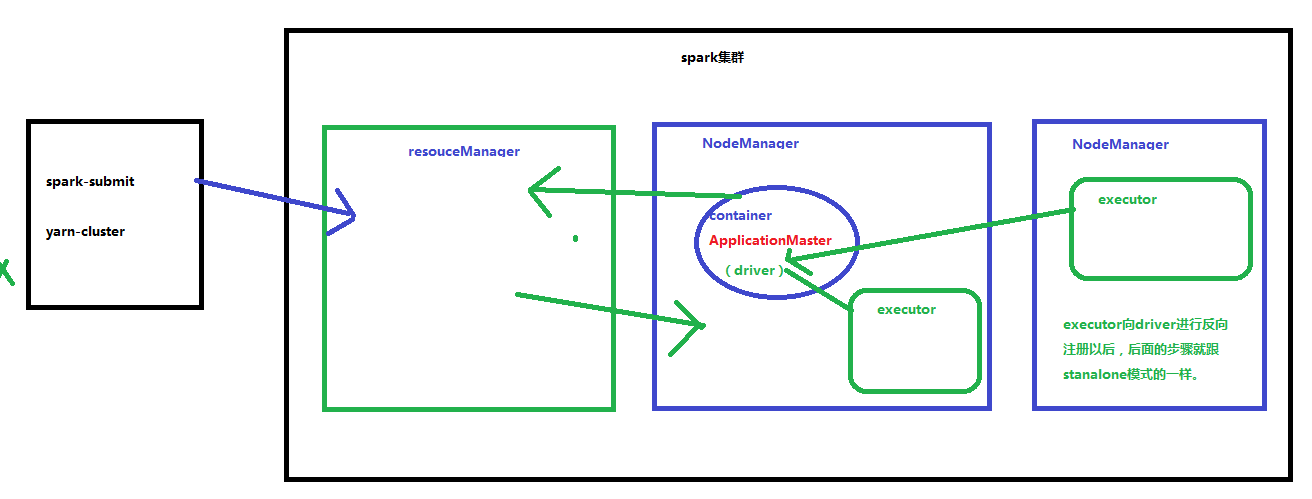
1000

**Mapreduce在YARN上面的运行的详细过程**

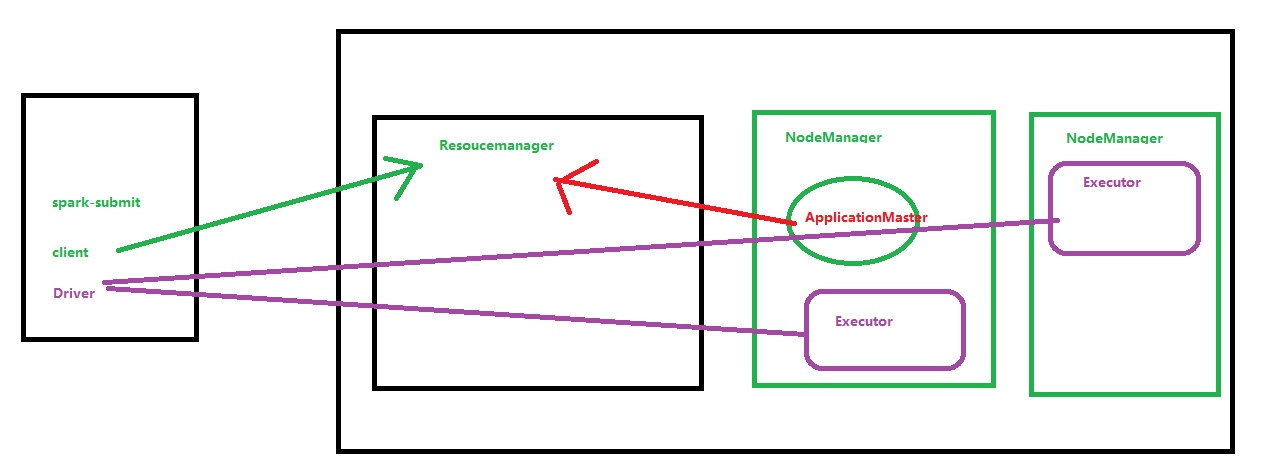


**Spark on yarn**

### Cluster



### Client



**建议：**

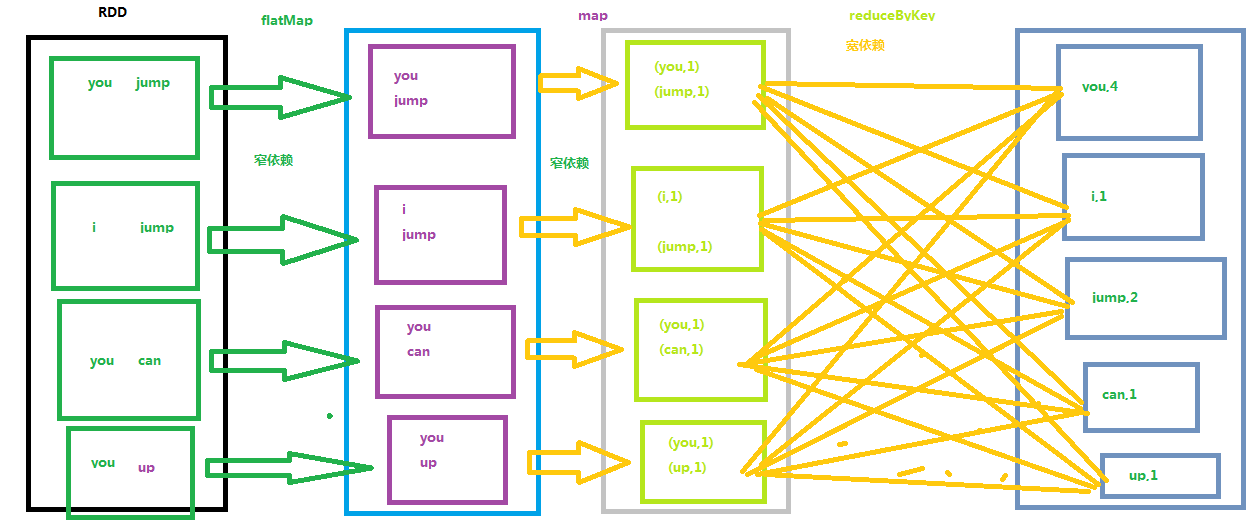
1：调试程序的时候，建议使用client模式。使用client模式的时候打印出来的信息非常

详细，有利于我们调试程序。

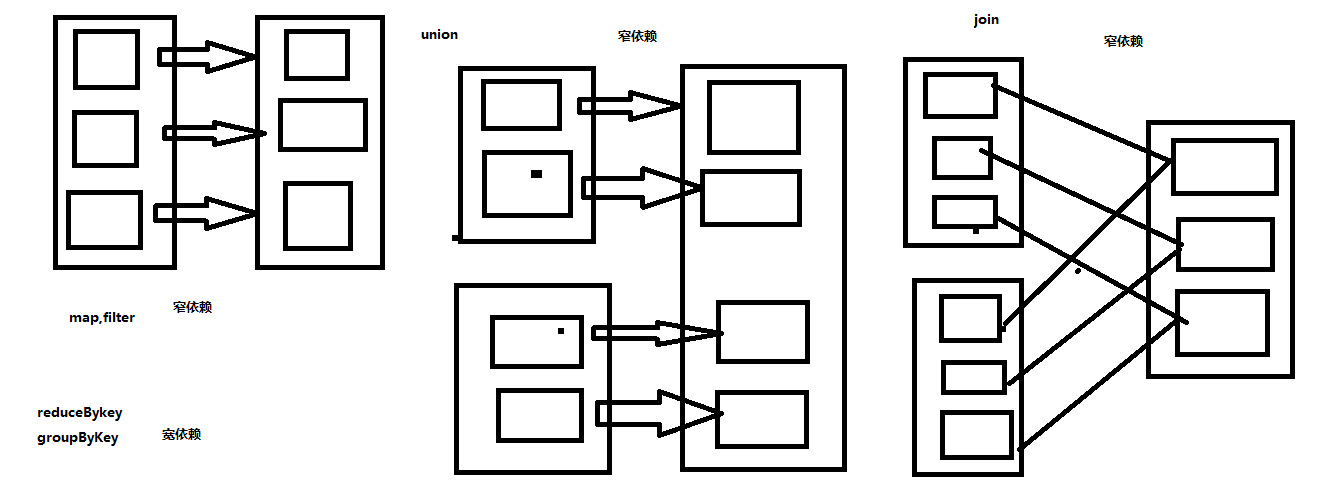
2：如果我们调试完成以后，建议使用cluster模式提交任务，分散driver服务器压力。

## 第17课时 窄依赖和宽依赖

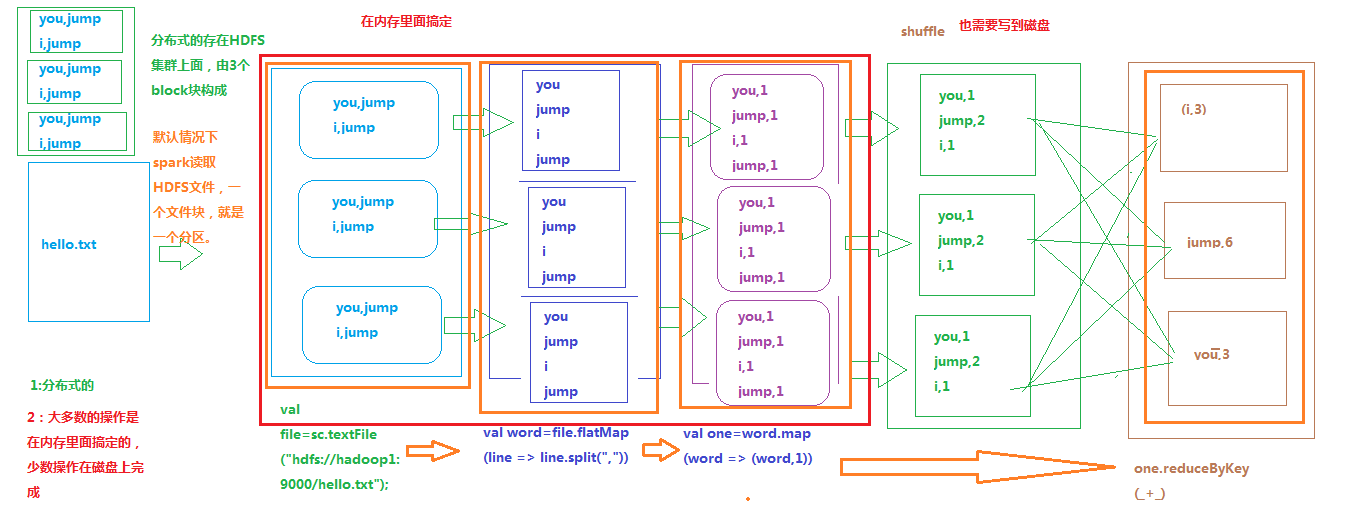
在RDD中将依赖分成了两种类型：窄依赖和宽依赖，**窄依赖是指父RDD的每个分区都只被子RDD一个分区使用**。相应的，那么宽依赖就是指**父RDD的分区被多个子RDD的分区所依赖。**



例子演示

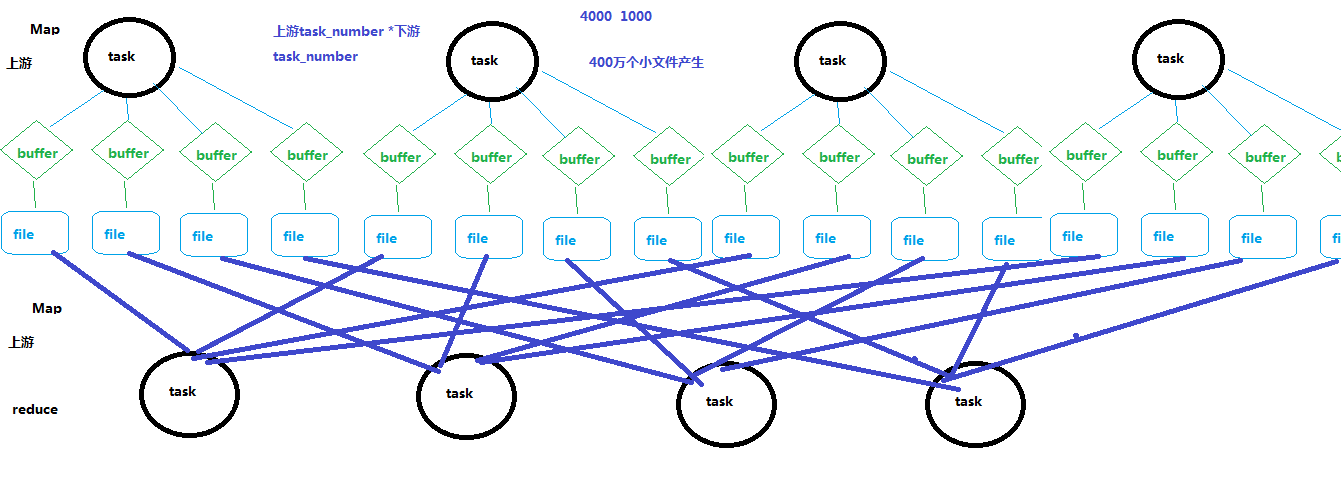


## 第18课时 Shuffle原理剖析



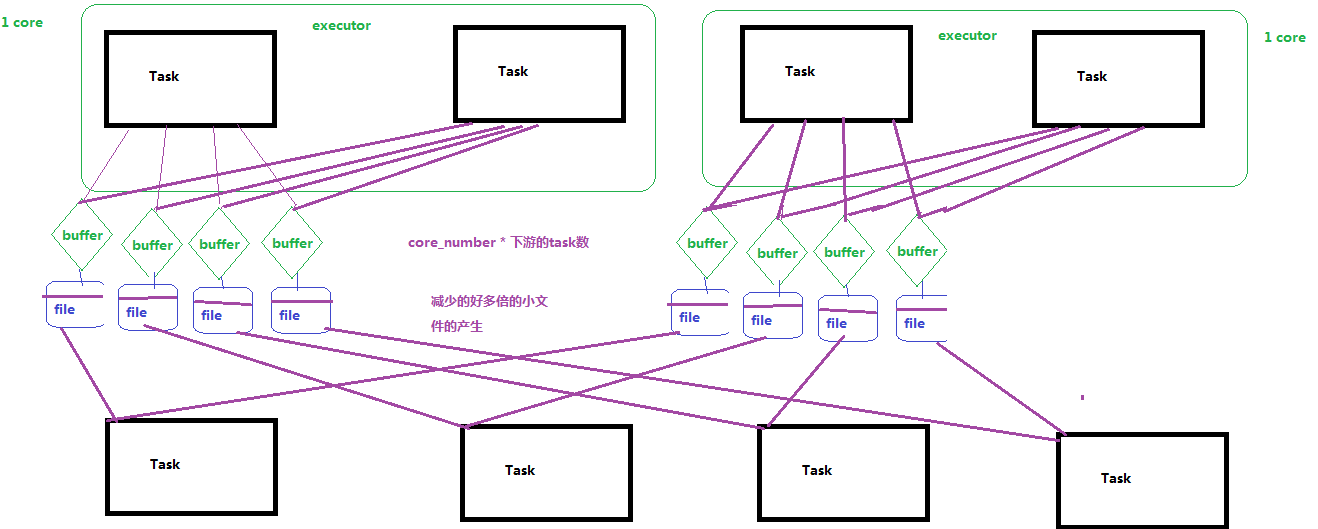
**Shuffle的发展历程**

**未经优化的HashShuffleManager(1.2版本之前)**



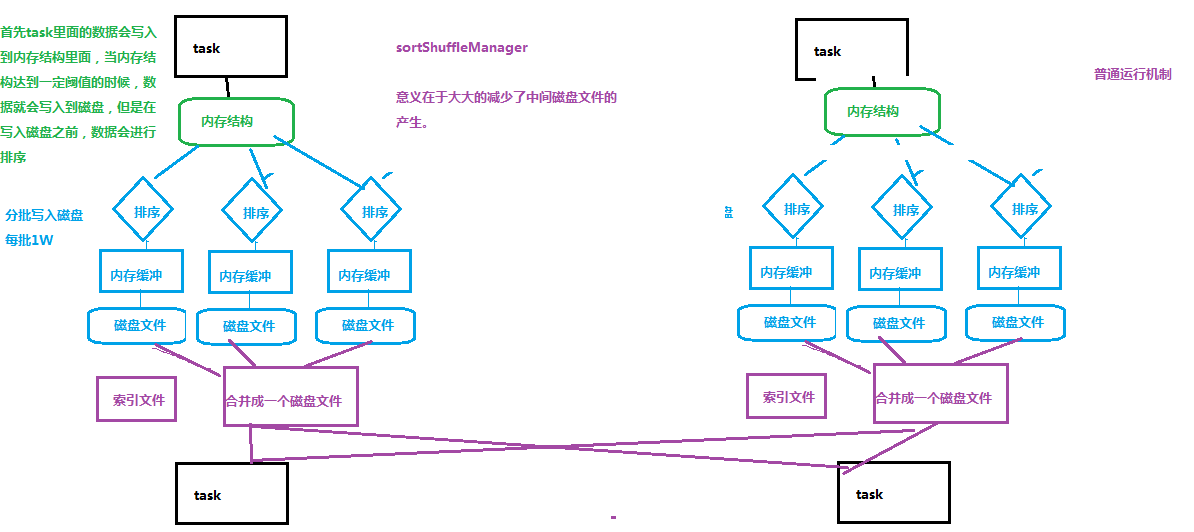
buffer数量与下游task数量有关。缺点是下游的task会为上游的每一个task创建一个file文件。导致shuffle过程中产生大量的磁盘文件。

**经过优化以后的HashShufferManager**



一个核内的task共用一组file。

**SortShuffleManager普通运行机制**



**Shuffle read task < spark.shuffle.sort.bypassMergeThreshold(200)**

