# 1.Spark 内核深度剖析

## 第1课时 Spark概述（四大特性）

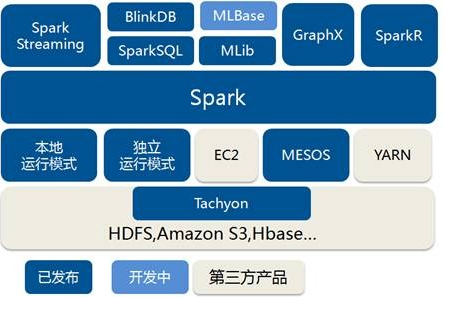
### 1.什么是spark

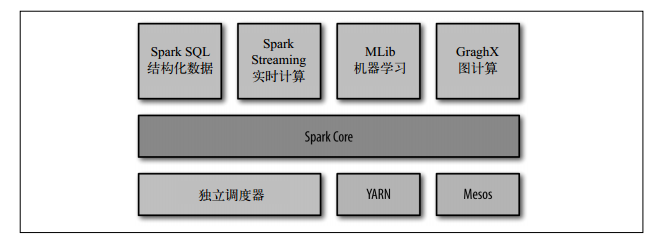
Spark 是一个用来实现快速而通用的集群计算的平台。  
 在速度方面， Spark 扩展了广泛使用的 MapReduce 计算模型，而且高效地支持更多计算模式，包括交互式查询和流处理。 在处理大规模数据集时，速度是非常重要的。速度快就意味着我们可以进行交互式的数据操作， 否则我们每次操作就需要等待数分钟甚至数小时。park 的一个主要特点就是能够在内存中进行计算，因而更快。不过即使是必须在磁盘上进行的复杂计算，Spark 依然比 MapReduce更加高效。Spark拥有DAG执行引擎，支持在内存中对数据进行迭代计算。官方提供的数据表明，如果数据由磁盘读取，速度是Hadoop MapReduce的10倍以上，如果数据从内存中读取，速度可以高达100多倍。  
 总的来说，Spark 适用于各种各样原先需要多种不同的分布式平台的场景，包括批处理、迭代算法、交互式查询、流处理。通过在一个统一的框架下支持这些不同的计算， Spark使我们可以简单而低耗地把各种处理流程整合在一起。而这样的组合，在实际的数据分析过程中是很有意义的。Spark的这种特性还大大减轻了原先需要对各种平台分别管理的负担。

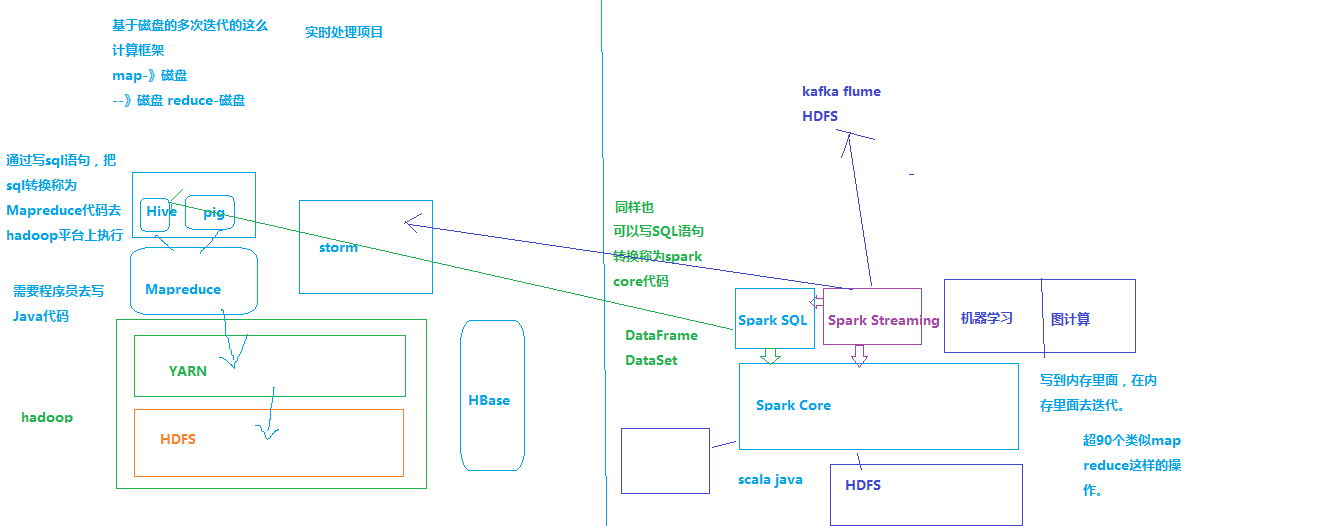
Spark不仅可以将任何Hadoop分布式文件系统（HDFS）上的文件读取为分布式数据集，也可以支持其他支持 Hadoop 接口的系统，比如本地文件、亚马逊 S3、Cassandra、Hive、HBase等。我们需要弄清楚的是，Hadoop并非Spark的必要条件，Spark 支持任何实现了Hadoop接口的存储系统。Spark支持的Hadoop输入格式包括文本文件、SequenceFile、Avro、Parquet等。

### Spark技术栈

Spark 项目包含多个紧密集成的组件。Spark 的核心是一个对由很多计算任务组成的、运行在多个工作机器或者是一个计算集群上的应用进行调度、分发以及监控的计算引擎。





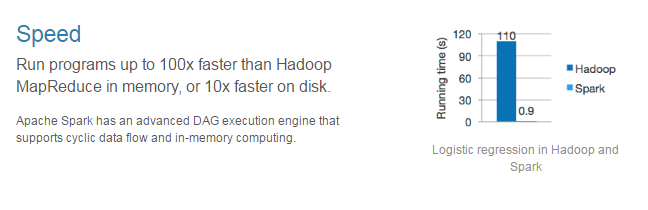


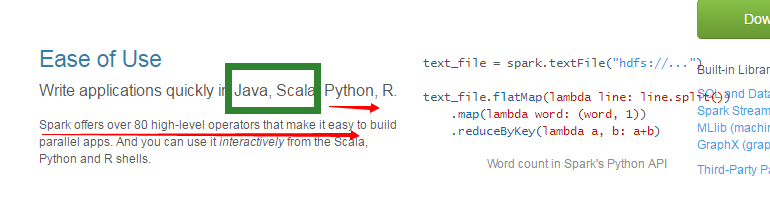
Spark Core  
 Spark Core实现了Spark的基本功能，包含任务调度、内存管理、错误恢复、与存储系统  
交互等模块。提供了有向无环图（DAG）的分布式并行计算框架，并提供Cache机制来支持多次迭代计算或者数据共享，大大减少迭代计算之间读取数据局的开销，这对于需要进行多次迭代的数据挖掘和分析性能有很大提升。Spark Core 中还包含了对弹性分布式数据集（resilient distributed dataset，简称RDD）的API定义。RDD表示分布在多个计算节点上可以并行操作的元素集合，是Spark主要的编程抽象。Spark Core提供了创建和操作这些集合的多个API。

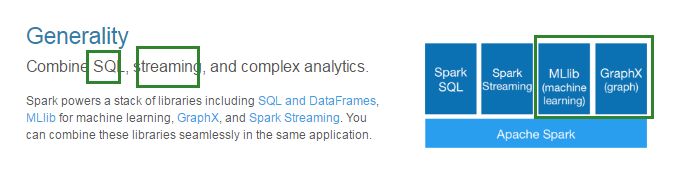
Spark SQL  
 Spark SQL是Spark用来操作结构化数据的程序包。通过Spark SQL，我们可以使用SQL或者Apache Hive版本的SQL方言（HQL）来查询数据。Spark SQL支持多种数据源，比如Hive表、Parquet以及JSON等。除了为Spark提供了一个SQL接口，Spark SQL还支持开发者将SQL和传统的RDD编程的数据操作方式相结合，不论是使用Python、Java还是Scala，开发者都可以在单个的应用中同时使用SQL和复杂的数据分析。

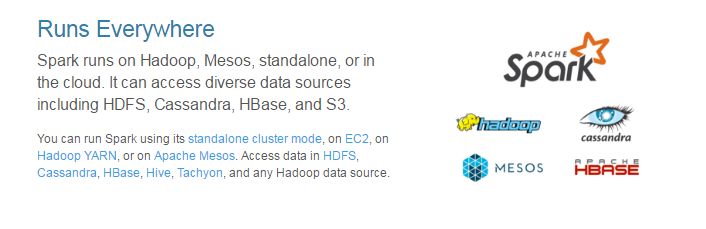
Spark Streaming  
 Spark Streaming是Spark提供的对实时数据进行流式计算的组件。比如生产环境中的网页  
服务器日志，或是网络服务中用户提交的状态更新组成的消息队列，都是数据流。Spark  
Streaming 提供了用来操作数据流的 API，可以对多种数据源（如Kdfka、Flume、Twitter、Zero和TCP 套接字）进行类似Map、Reduce和Join等复杂操作。并且与Spark Core中的RDD API高度对应。这样一来，程序员编写应用时的学习门槛就得以降低，不论是操作内存或硬盘中的数据，还是操作实时数据流，程序员都更能应对自如。

### 3.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**

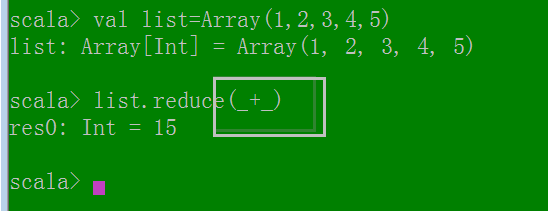
val a=0; 修饰的，这个变量的值不可更改。

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？

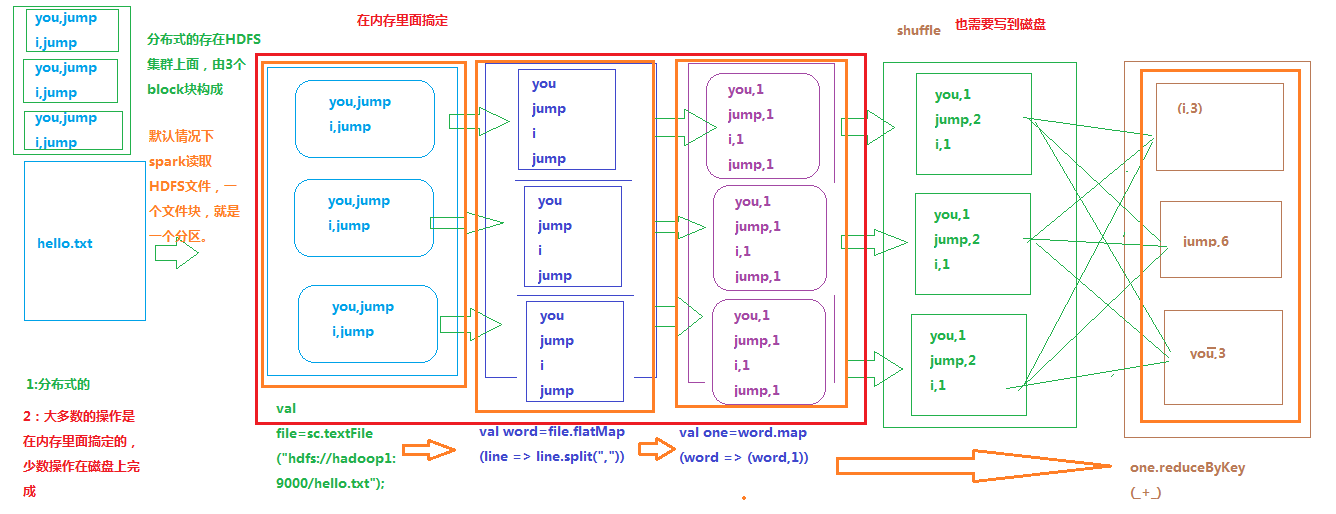
1、RDD是Spark提供的核心抽象，全称为Resillient Distributed Dataset，即**弹性**分布式数据集。

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）

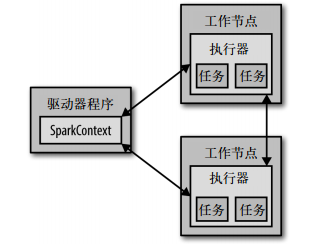


## 第4课时 Spark核心概念与架构

**Spark核心概念：**

1. **Driver、Worker、Executor、Task：**

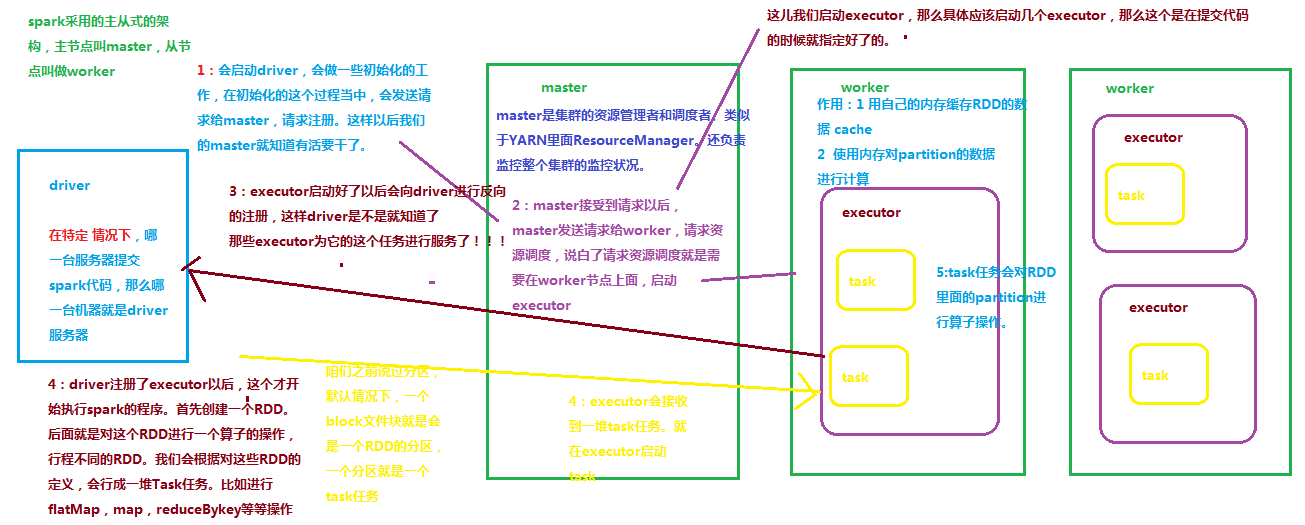
每个 Spark 应用都由一个驱动器程序（ driver program）来发起集群上的各种并行操作。驱动器程序包含应用的 main 函数，并且定义了集群上的分布式数据集，还对这些分布式数据集应用了相关操作。驱动器程序通过一个 SparkContext 对象来访问Spark。这个对象代表对计算集群的一个连接。shell启动时已经自动创建了一个SparkContext对象，是一个叫作sc的变量。一旦有了SparkContext，就可以用它来创建RDD。我们可以在这些行上进行各种操作，比如 count()。要执行这些操作，驱动器程序一般要管理多个执行器（ executor）节点。比如，如果我们在集群上运行 count() 操作，那么不同的节点会统计文件的不同部分的行数。



**Master概念**

Spark API 最神奇的地方就在于像 filter 这样基于函数的操作也会在集群上并行执行。也就是说， Spark 会自动将函数（比如 line.contains("Python")）发到各个执行器节点上。这样，你就可以在单一的驱动器程序中编程，并且让代码自动运行在多个节点上。

**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：

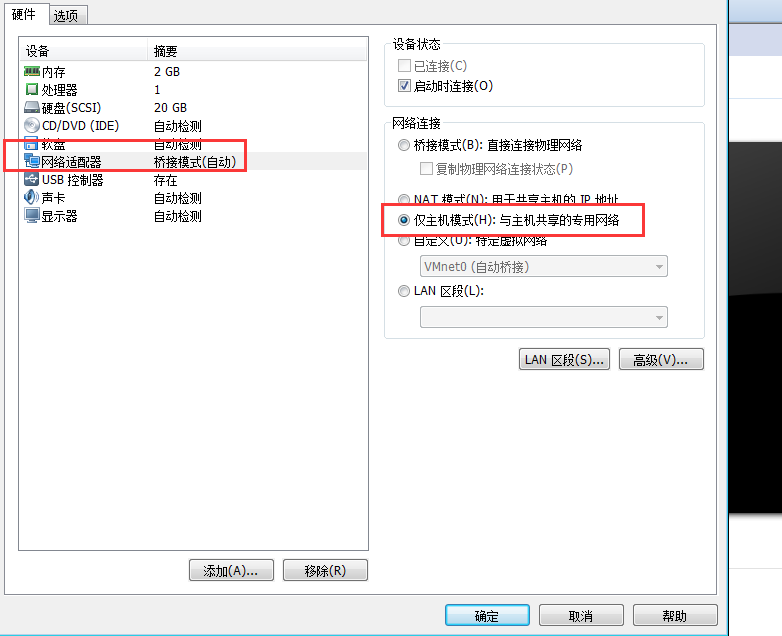


### 2.Linux - centOS，redhat 红帽（64）

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

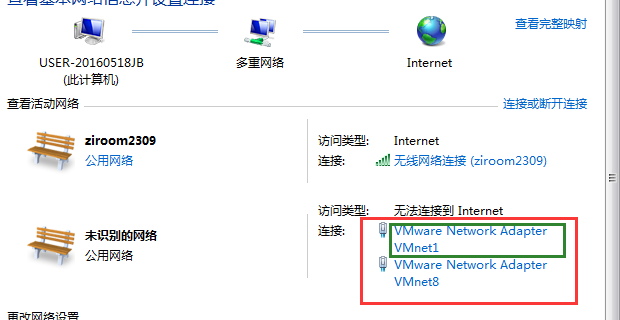


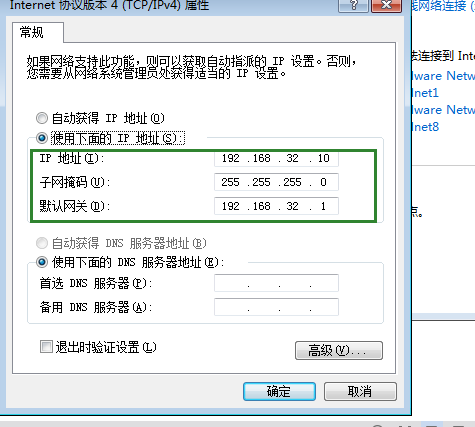
需要我们配置成windows操作系统上CRT或者PTTV直接可以连到虚拟机里面linux上。

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

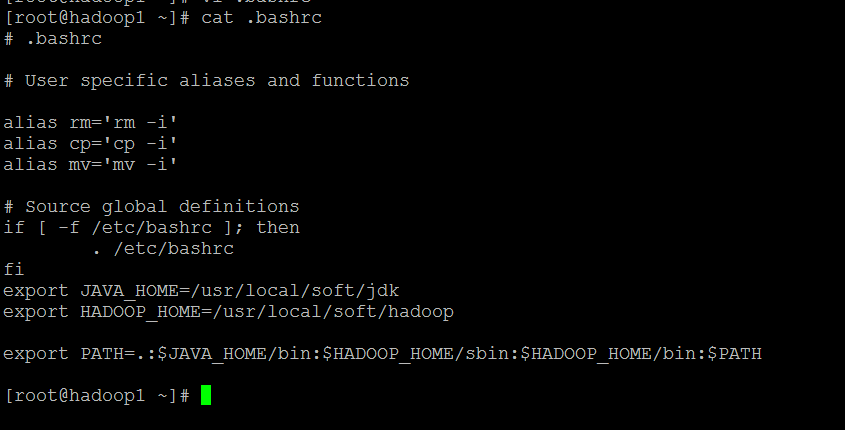
## 第6课时 hadoop环境准备

### Hadoop搭建





设置环境变量



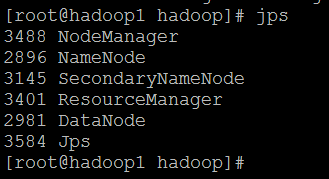
### HDFS



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。

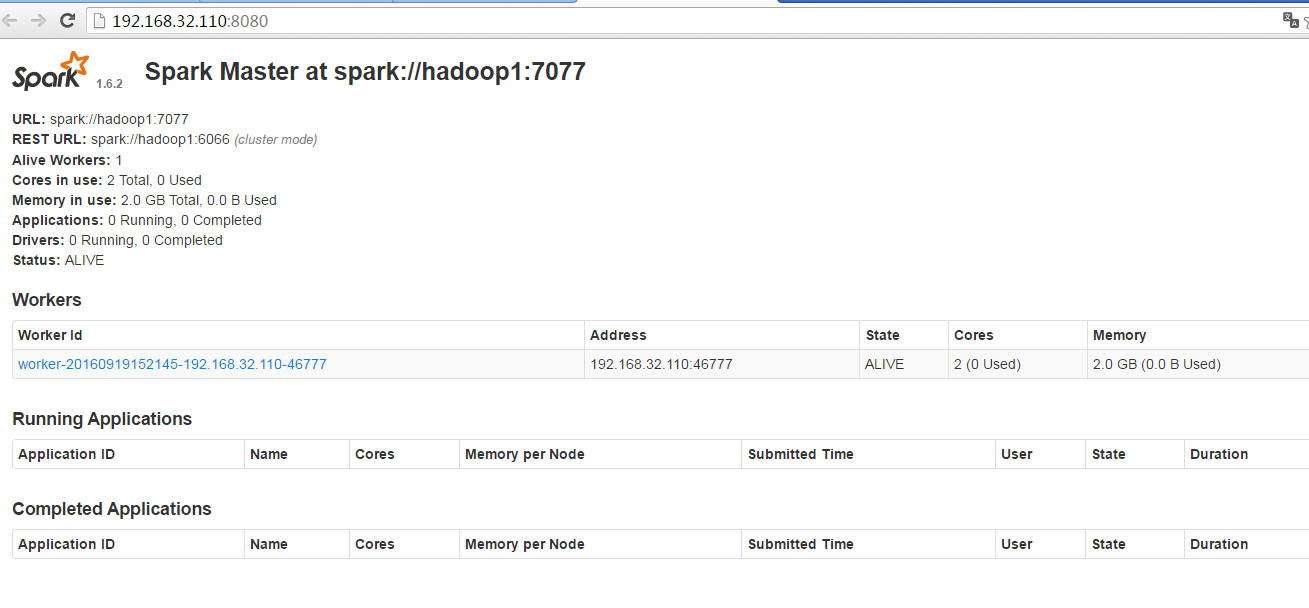
100G 85%

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）

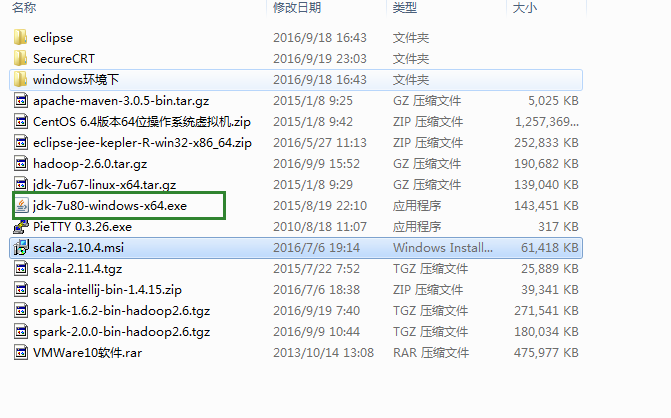
### Java环境

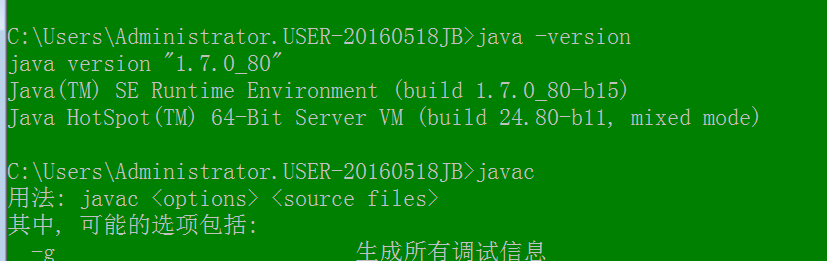
1：安装JDK

2：安装maven

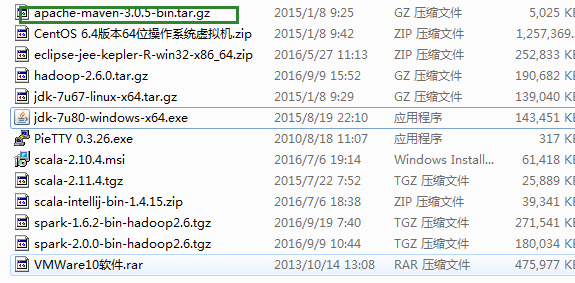
3：安装eclipse

4：配置eclipse maven环境

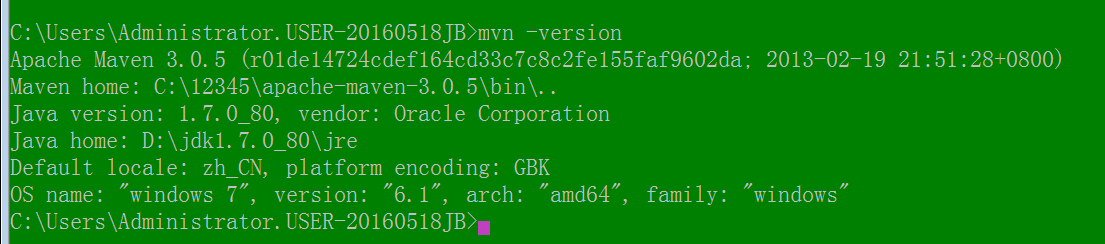




Maven解压出来就可以使用

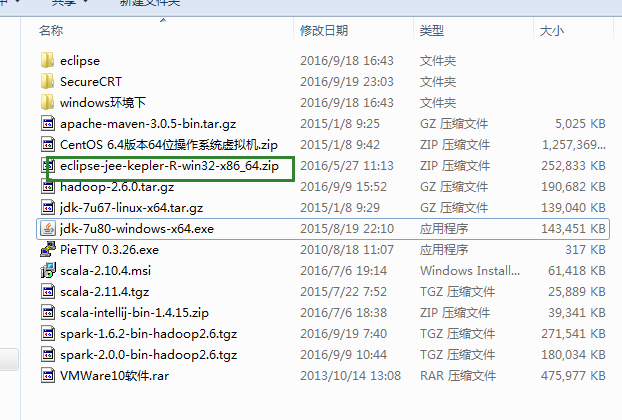


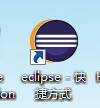
验证maven（maven要安装成功，必须先安装好jdk）



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。





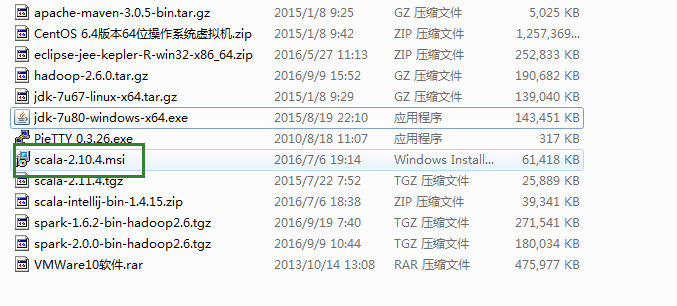
### Scala环境

1：安装scala

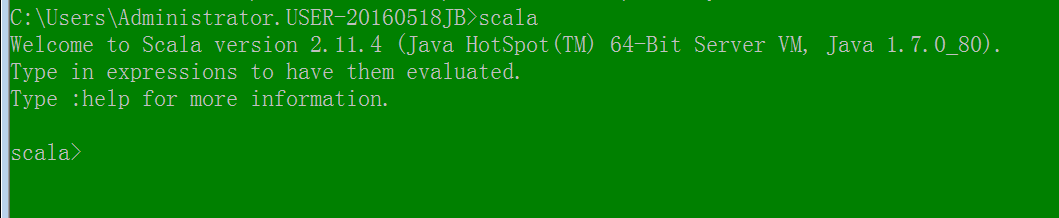
2：安装eclipse

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）

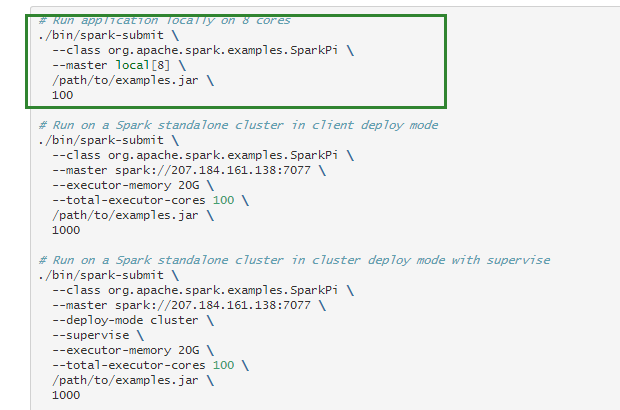
Group id : org.scala-tools.archetypes

Artifact id : scala-archetype-simple

Version : 1.2

## 第9课时 Spark运行模式

### 本地运行spark任务--local



### 集群运行spark任务--standalone



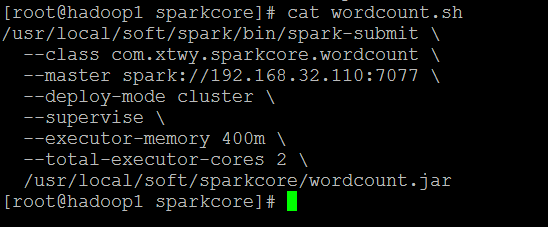
在standalone模式下：

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(“”);

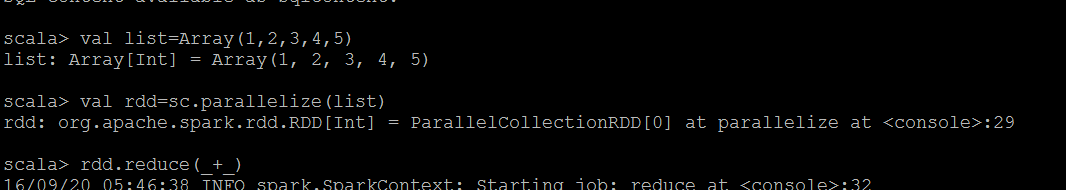
### 方式2：并行化的方式创建RDD

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。

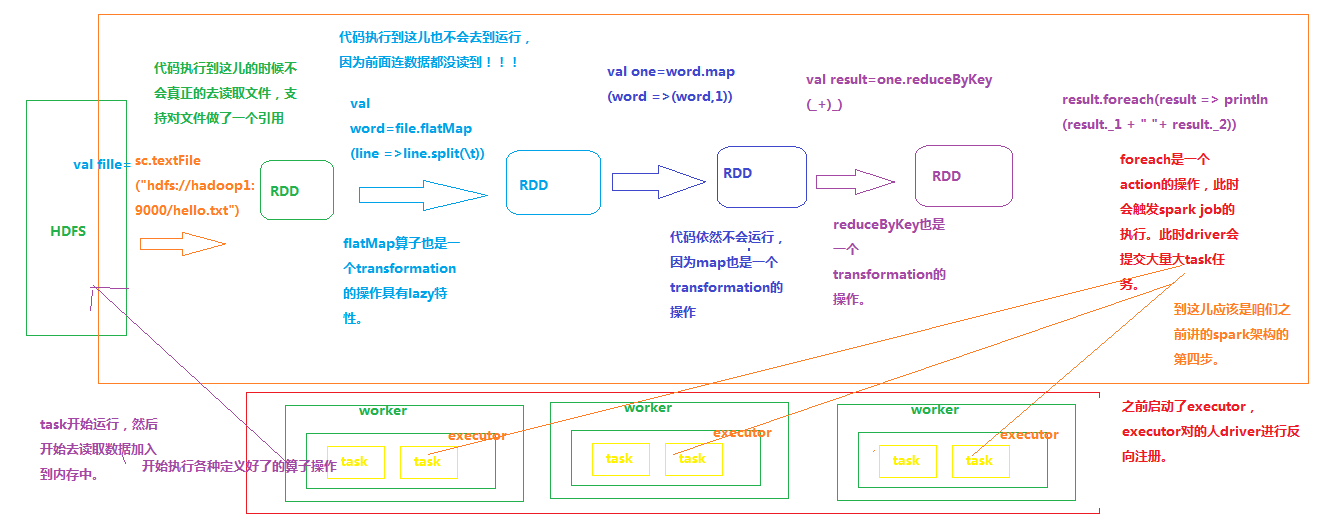
## 第12课时 Transformation和action原理剖析

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。

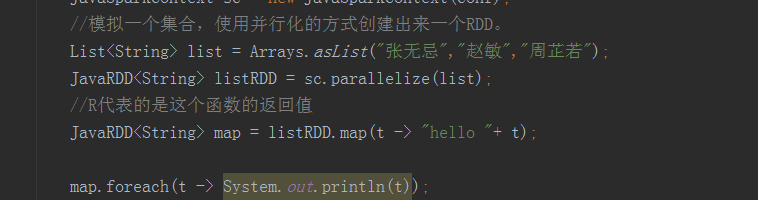


## 第13课时 Transformation和action算子演示

### Java Lambda演示Transformation算子

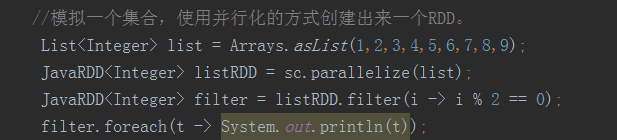
**Map:**

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



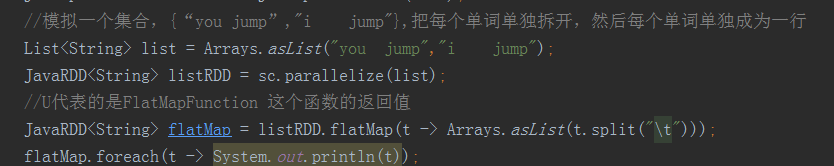
**Filter:**

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



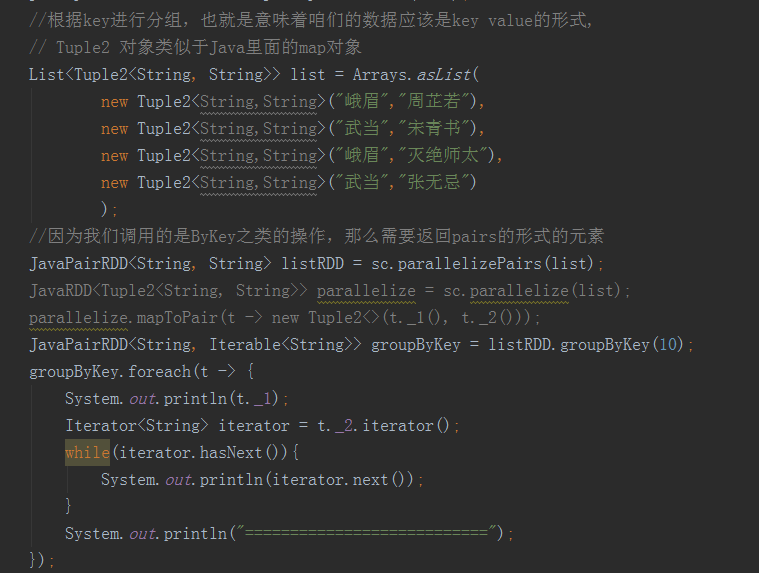
**flatMap:**

  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）



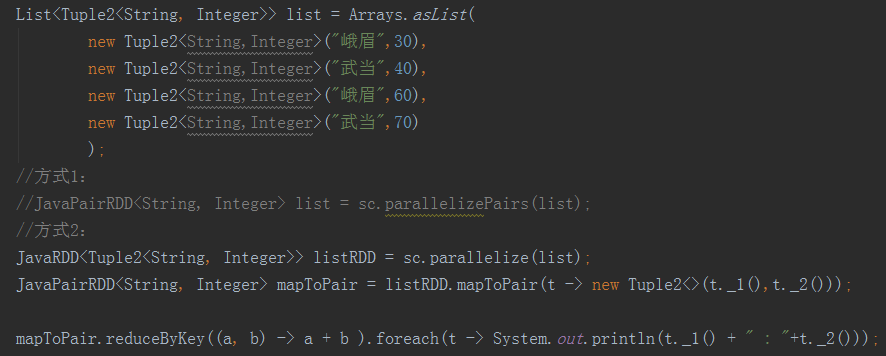
**groupByKey([numTasks])：**

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。



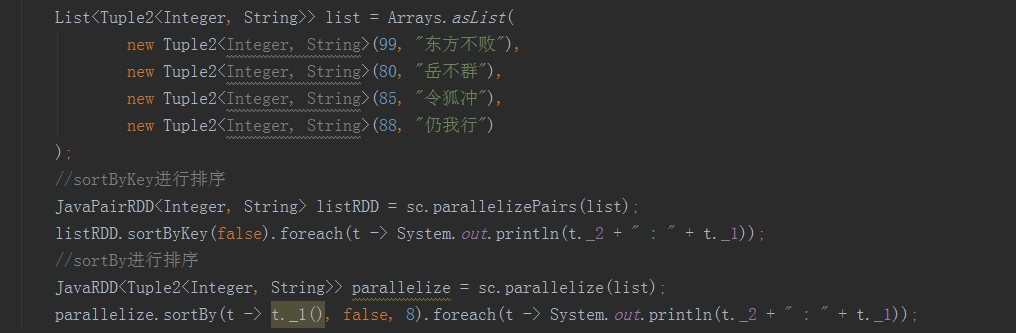
**reduceByKey(func, [numTasks]) :**

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。



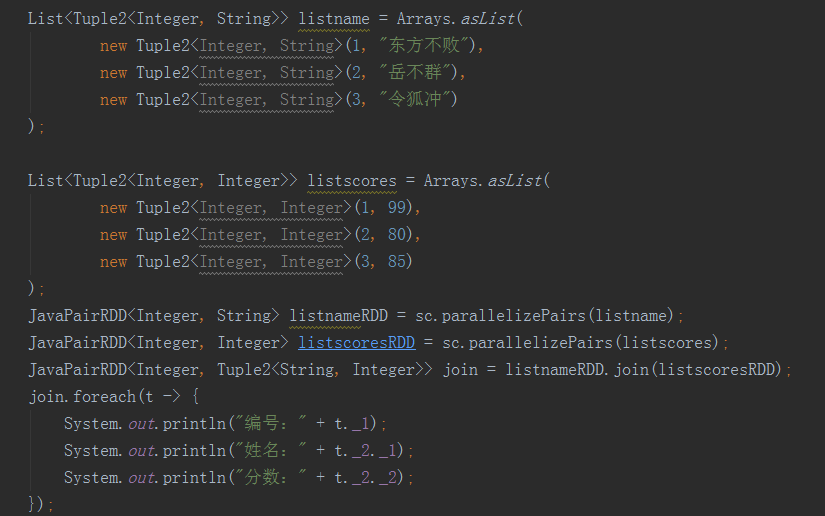
**sortByKey与sortBy：**

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



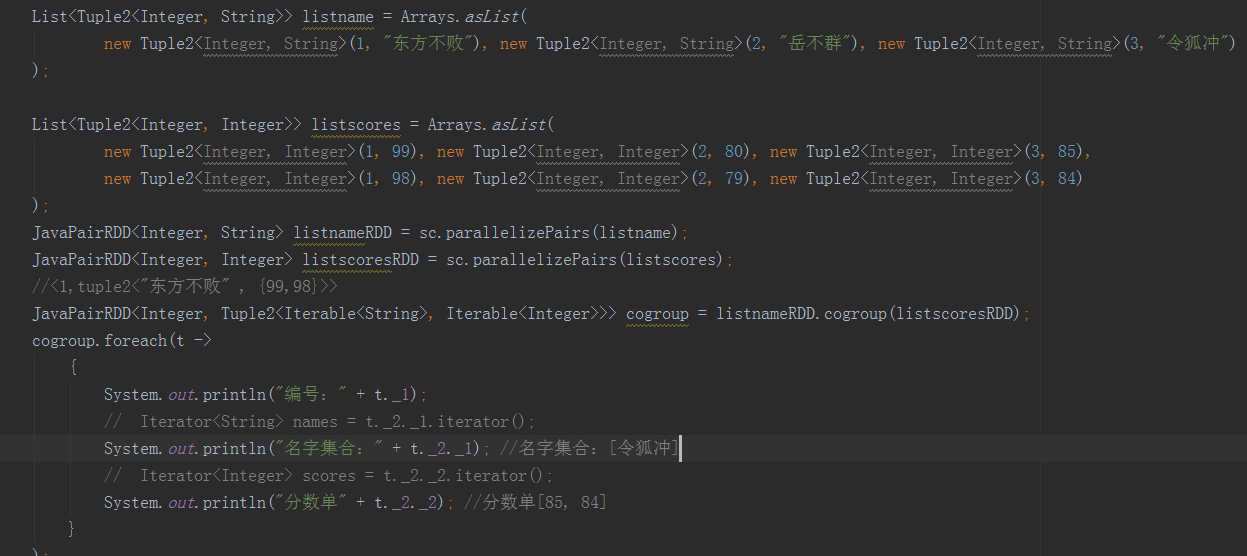
**join(otherDataset, [numTasks]) :**

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。

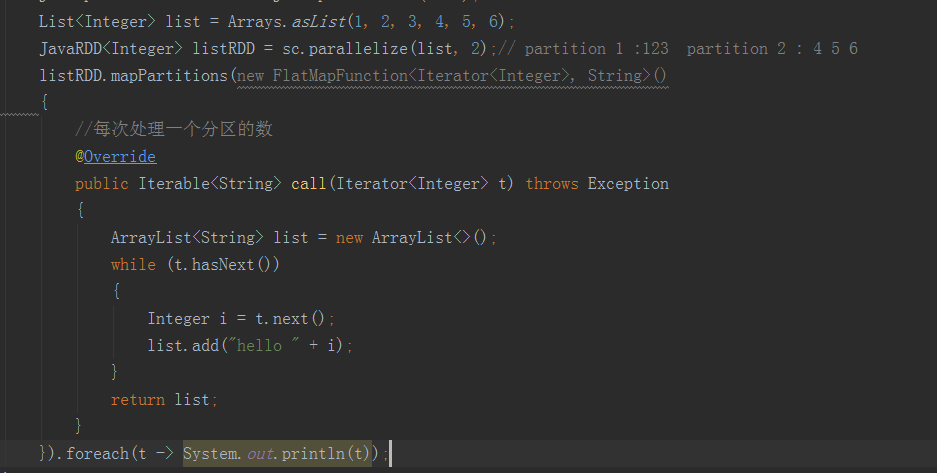
**cartesian(otherDataset) :**

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。

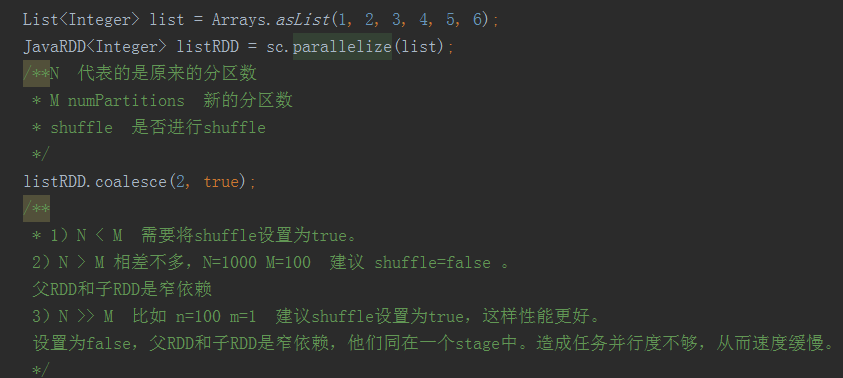
2）N > M 相差不多，N=1000 M=100 建议 shuffle=false 。

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。

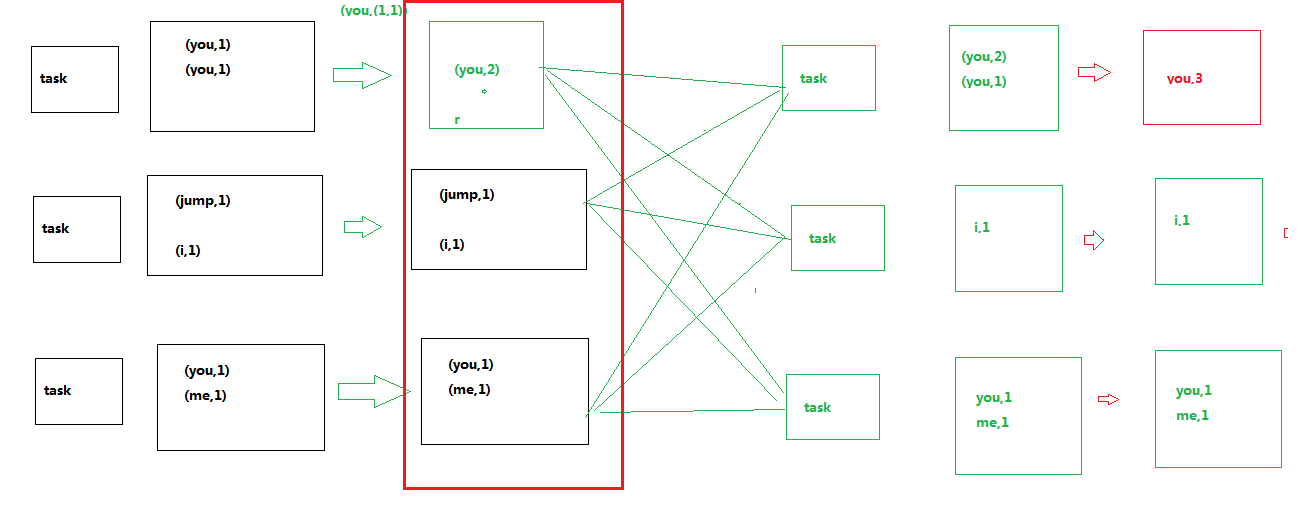
设置为false，父RDD和子RDD是窄依赖，他们同在一个stage中。造成任务并行度不够，从而速度缓慢。

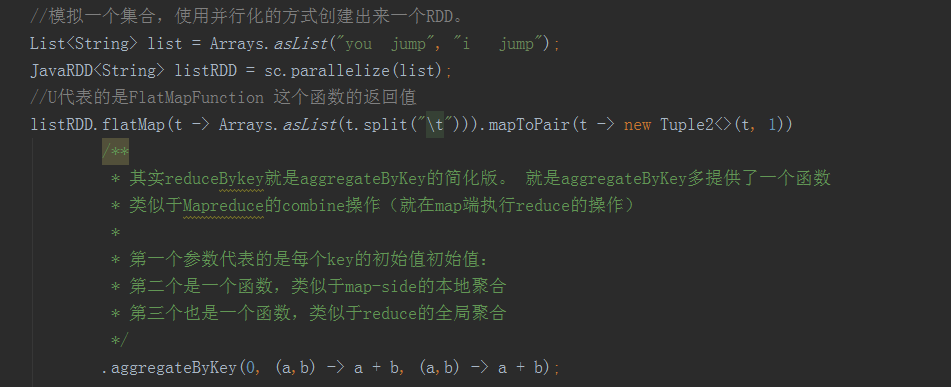
filter 过滤了以后 --partition数据量会减少，可能需要重新分区。



**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**

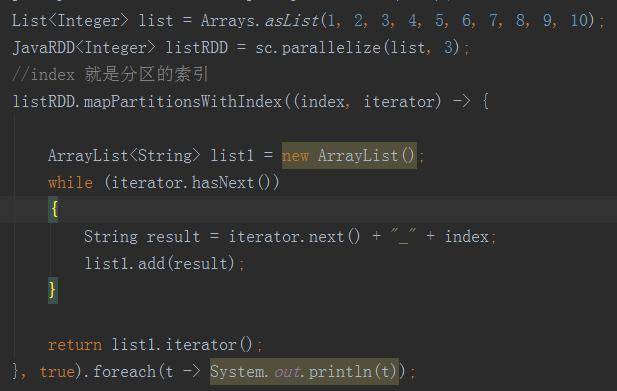




调优算子，比reduceByKey多提供了一个参数，用于map阶段进行本地聚合操作，即上图红框中的部分。在map阶段进行聚合之后，可以使传到reduce端的数据变少，提高性能。

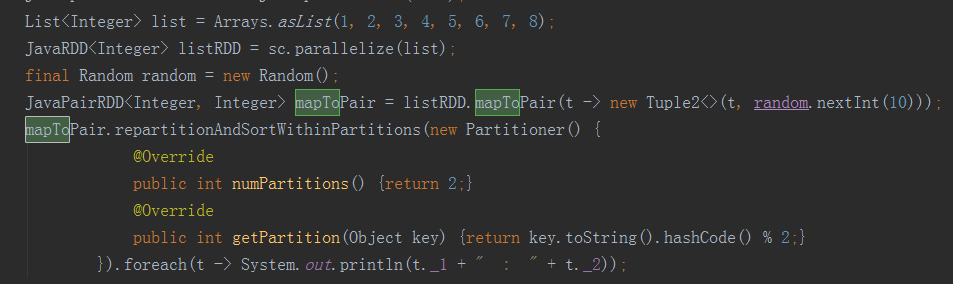
**mapPartitionsWithIndex:**

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



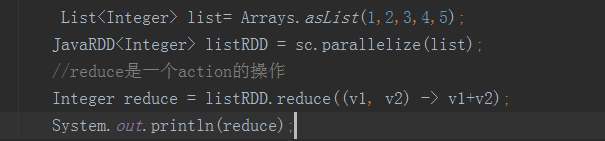
**repartitionAndSortWithinPartitions:**

**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。**

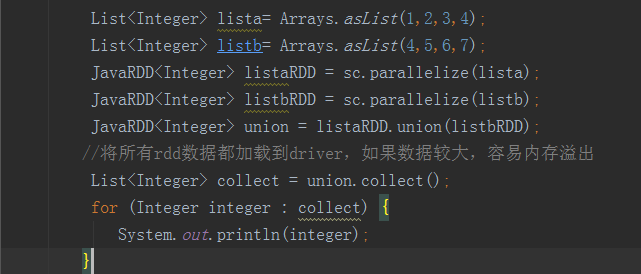


### Java Lambda演示action算子

**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

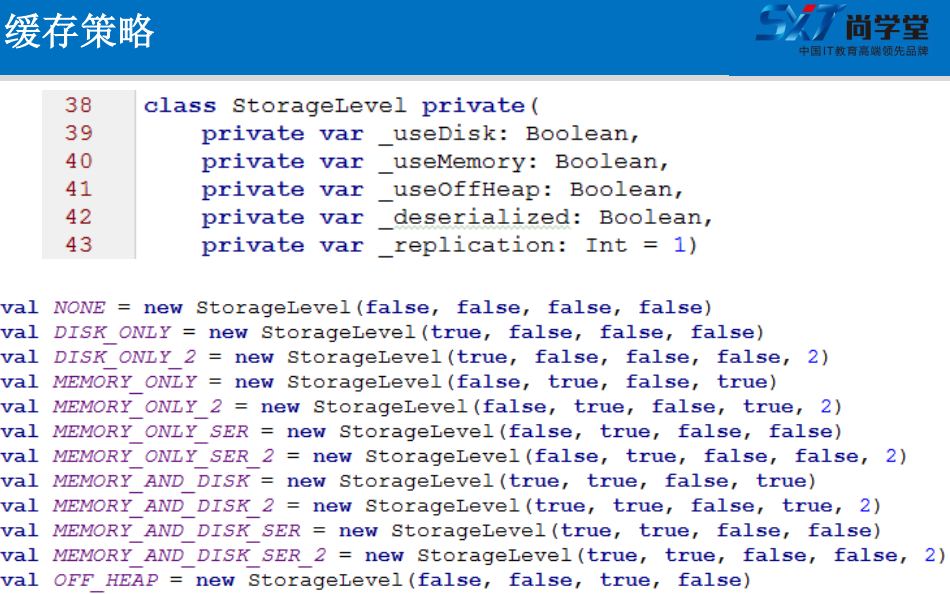
### RDD持久化

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；

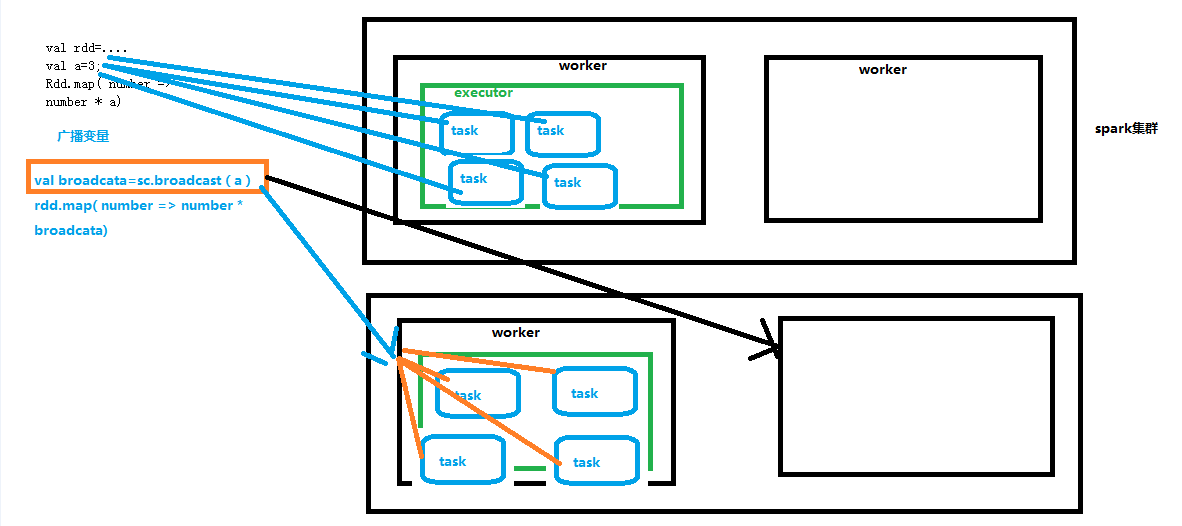
　　Spark提供了两种类型的共享变量（Shared Variables）：广播变量（Broadcast Variables）、累加器（Accumulators ）；

### 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；

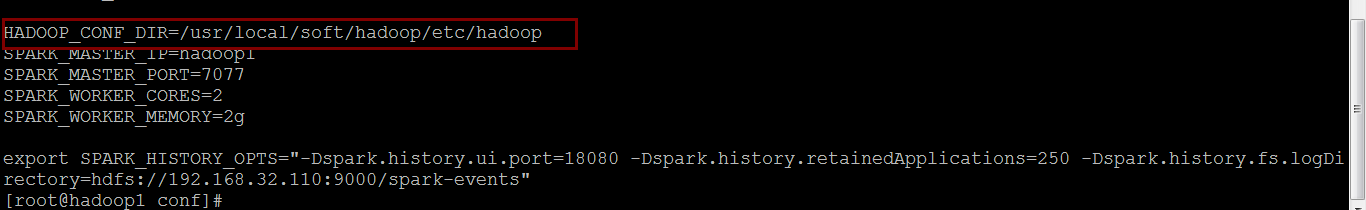
在使用累加器（Accumulators）时需要注意，只有在action操作中才会触发累加器，也就是说上述代码中由于flatMap()为transformation操作，因为Spark惰性特征所以只用当saveAsTextFile() 执行时累加器才会被触发；累加器只有在驱动程序中才可访问，worker节点中的任务不可访问累加器中的值。

## 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*

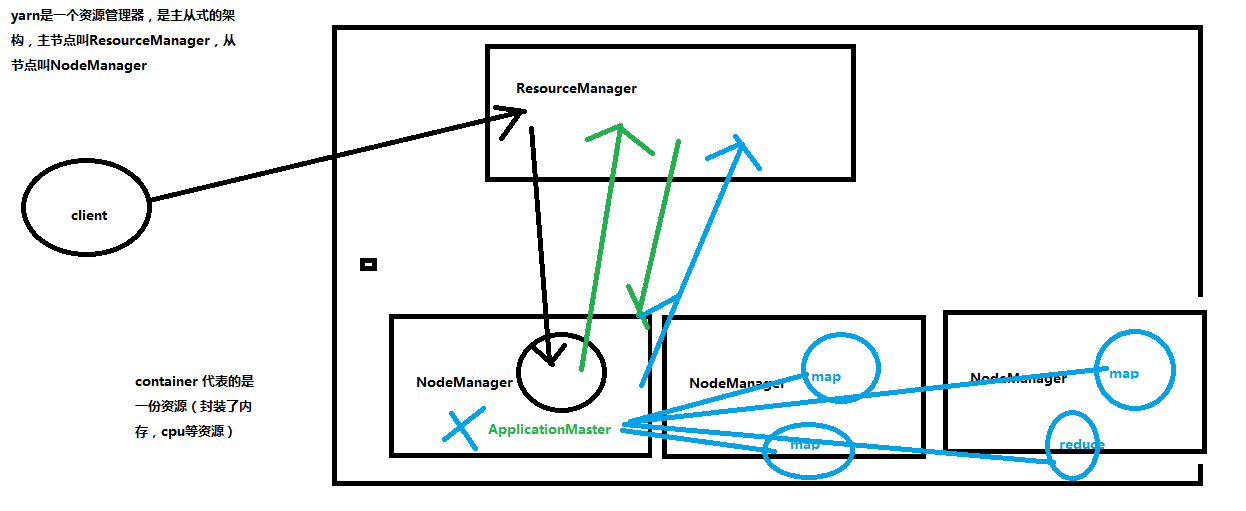
--executor-memory 20G **\**

--num-executors 50 **\**

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

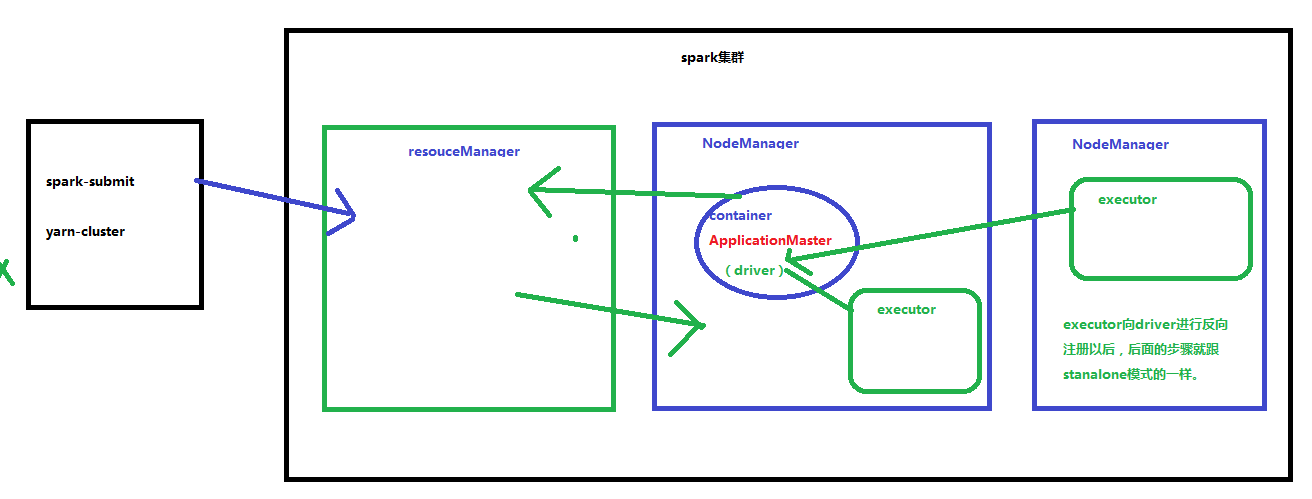
1000

**Mapreduce在YARN上面的运行的详细过程**

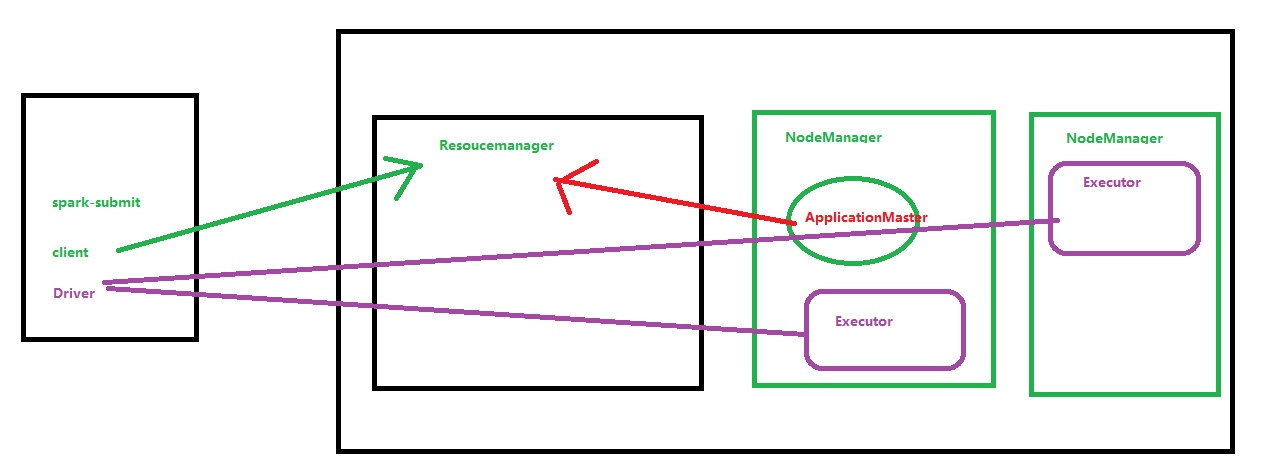


**Spark on yarn**

### Cluster



### Client



**建议：**

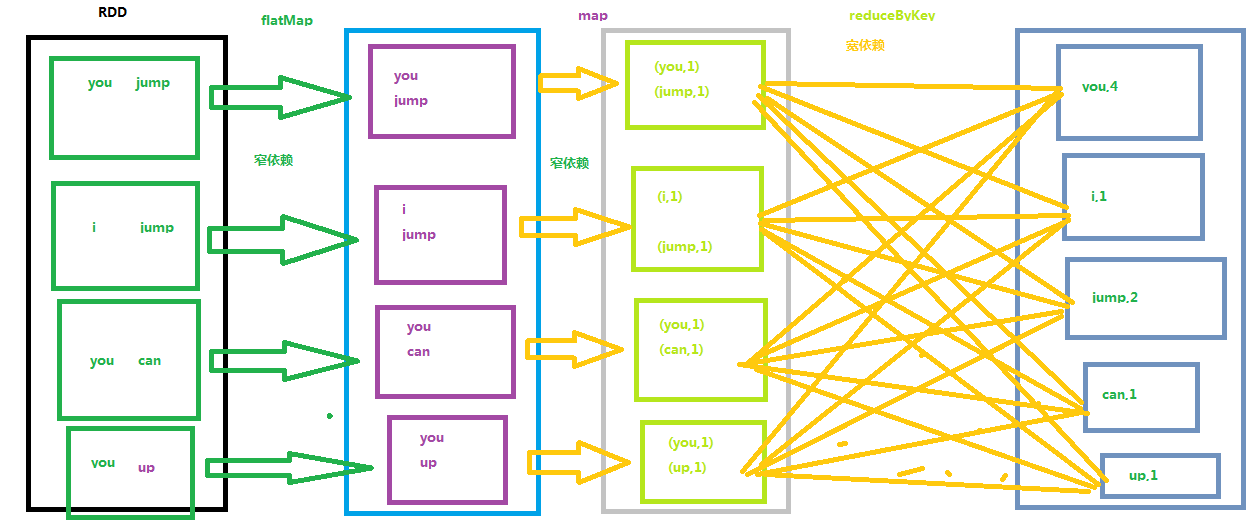
1：调试程序的时候，建议使用client模式。使用client模式的时候打印出来的信息非常

详细，有利于我们调试程序。

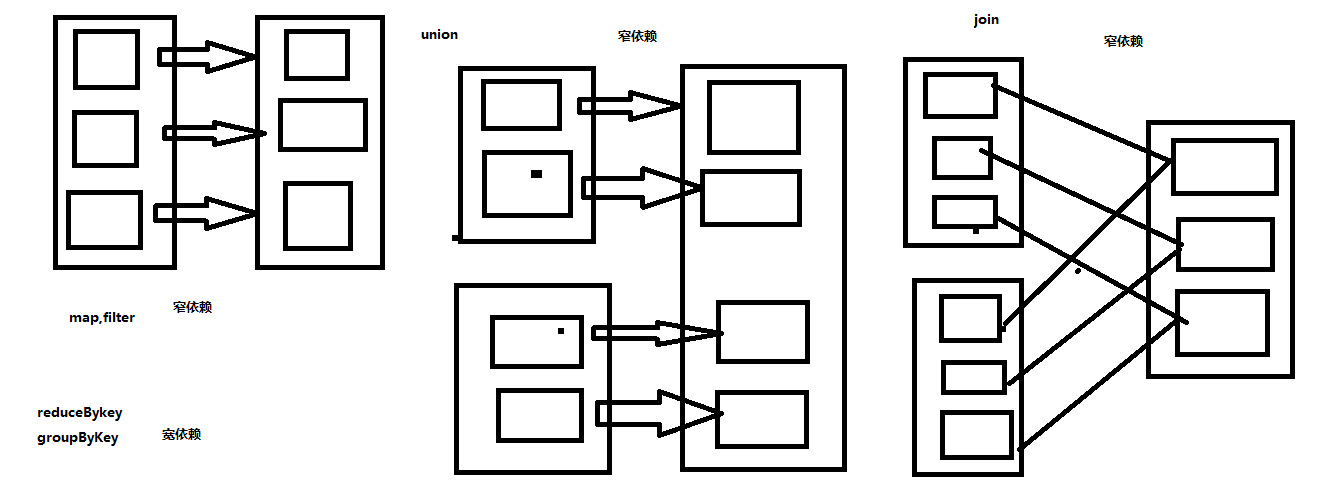
2：如果我们调试完成以后，建议使用cluster模式提交任务，分散driver服务器压力。

## 第17课时 窄依赖和宽依赖

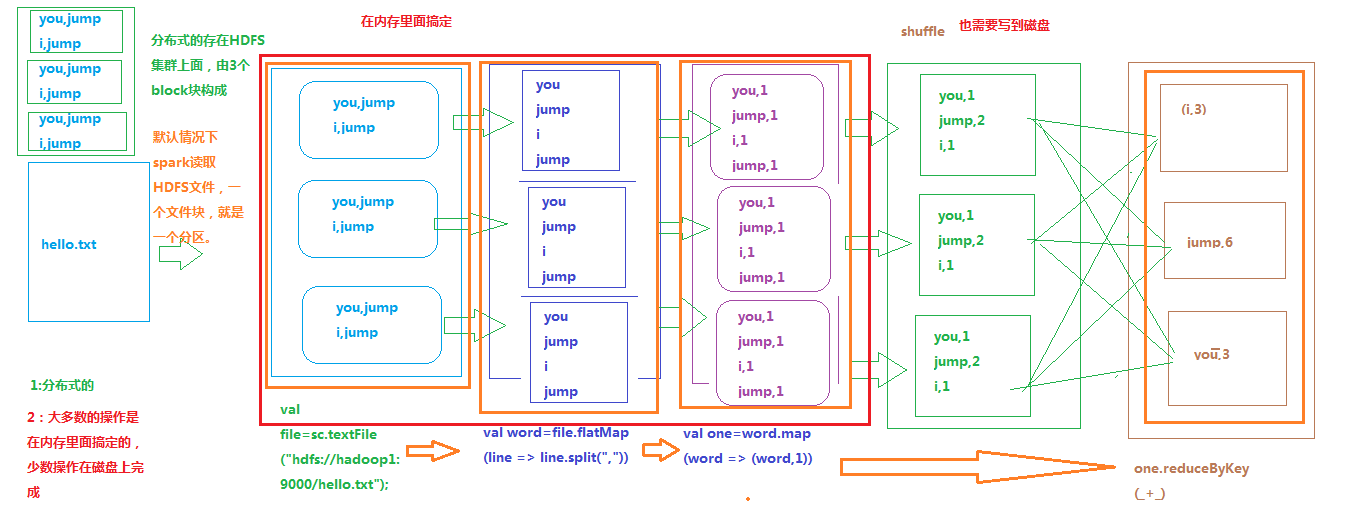
在RDD中将依赖分成了两种类型：窄依赖和宽依赖，**窄依赖是指父RDD的每个分区都只被子RDD一个分区使用**。相应的，那么宽依赖就是指**父RDD的分区被多个子RDD的分区所依赖。**



例子演示

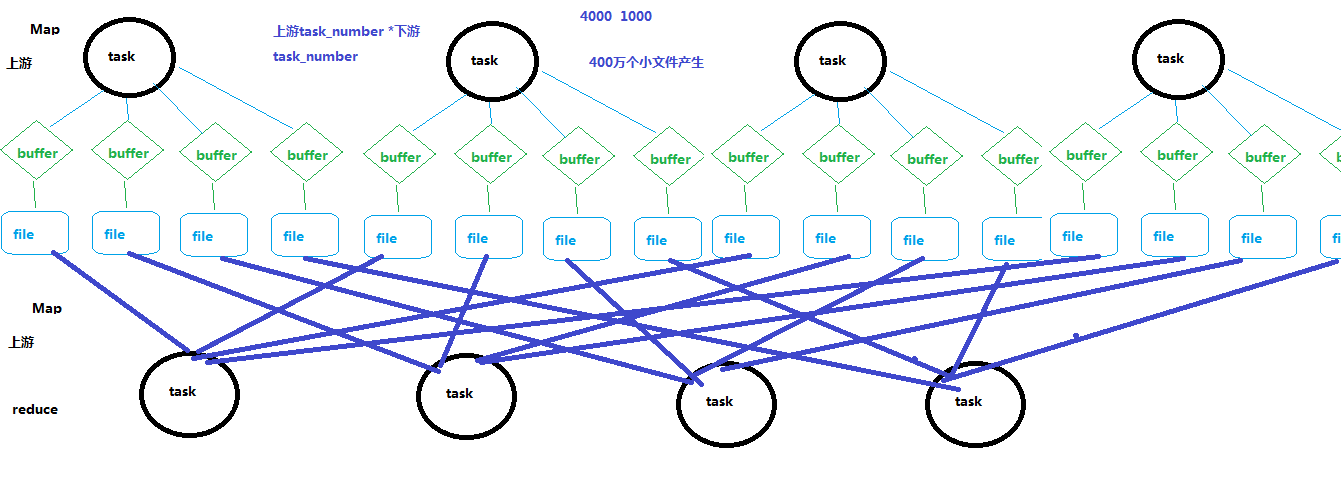


## 第18课时 Shuffle原理剖析



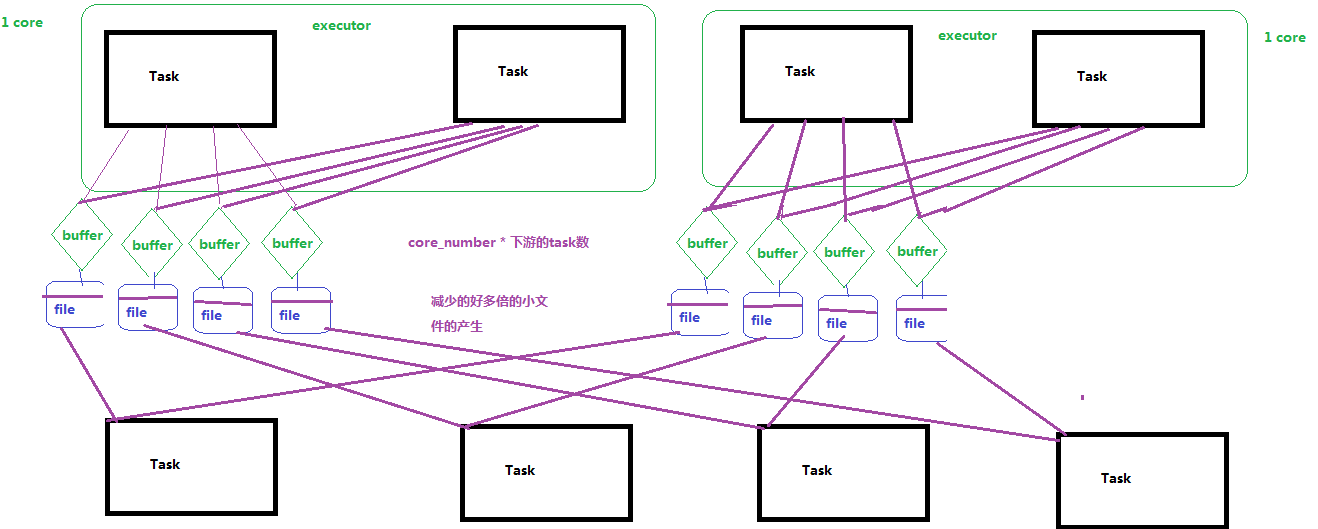
**Shuffle的发展历程**

**未经优化的HashShuffleManager(1.2版本之前)**



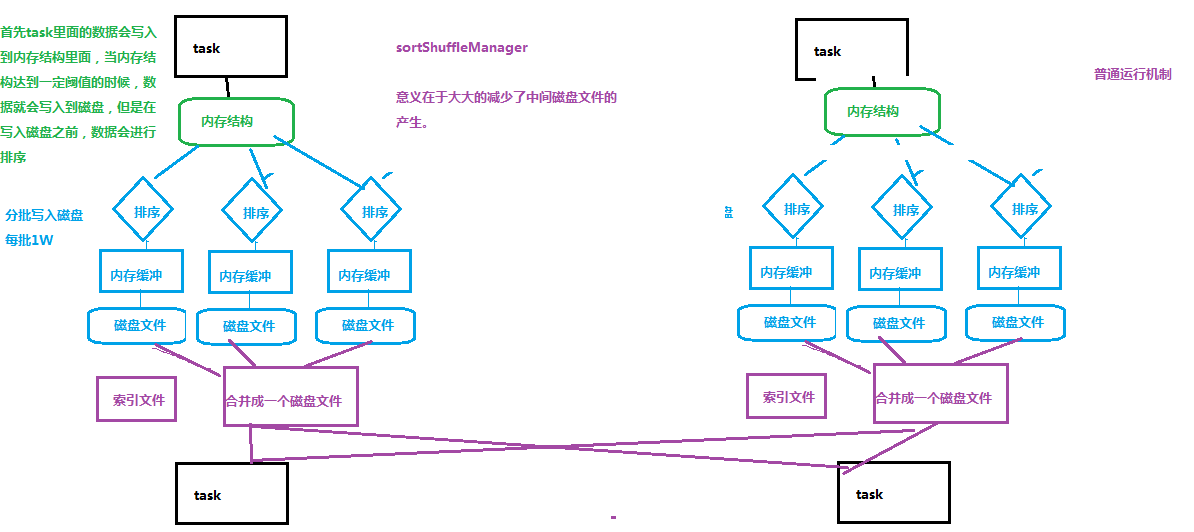
buffer数量与下游task数量有关。缺点是下游的task会为上游的每一个task创建一个file文件。导致shuffle过程中产生大量的磁盘文件。

**经过优化以后的HashShufferManager**

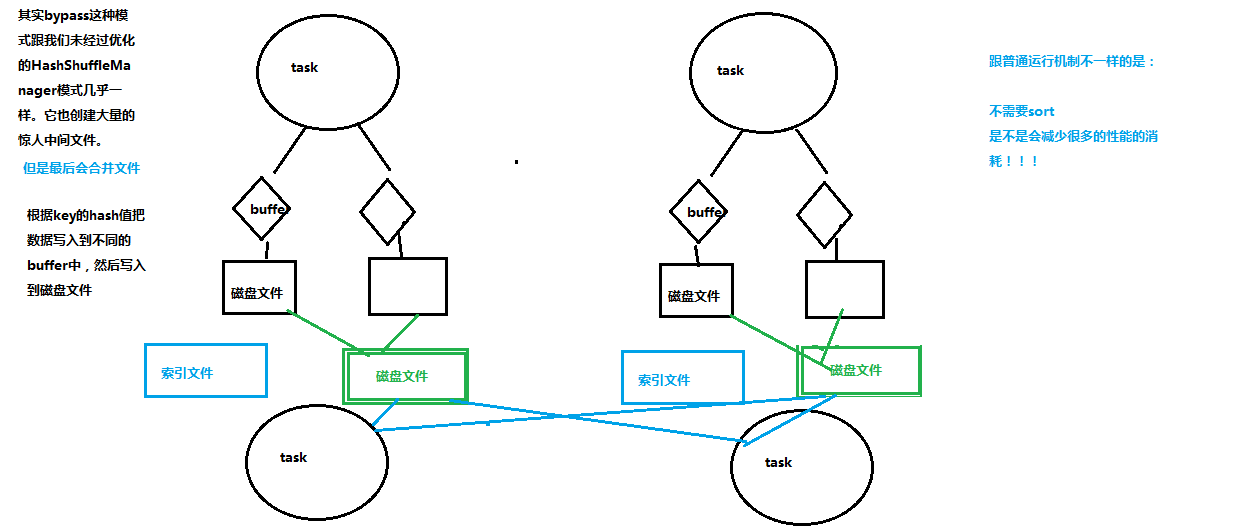


一个核内的task共用一组file。

**SortShuffleManager普通运行机制**



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

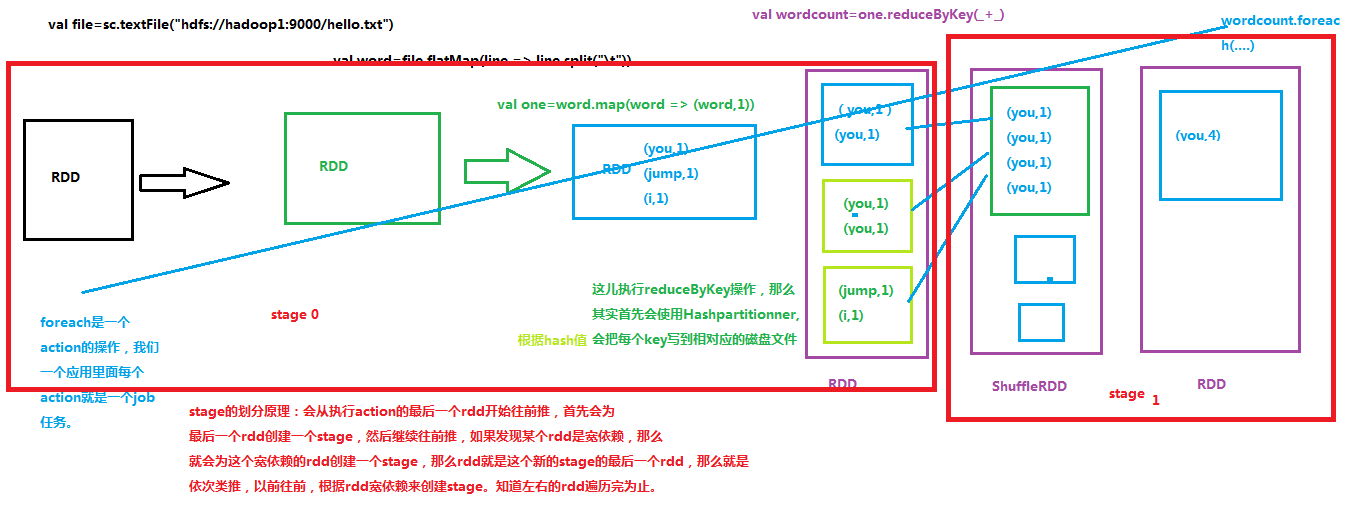


## 第19课时 Stage划分原理剖析

**概念**

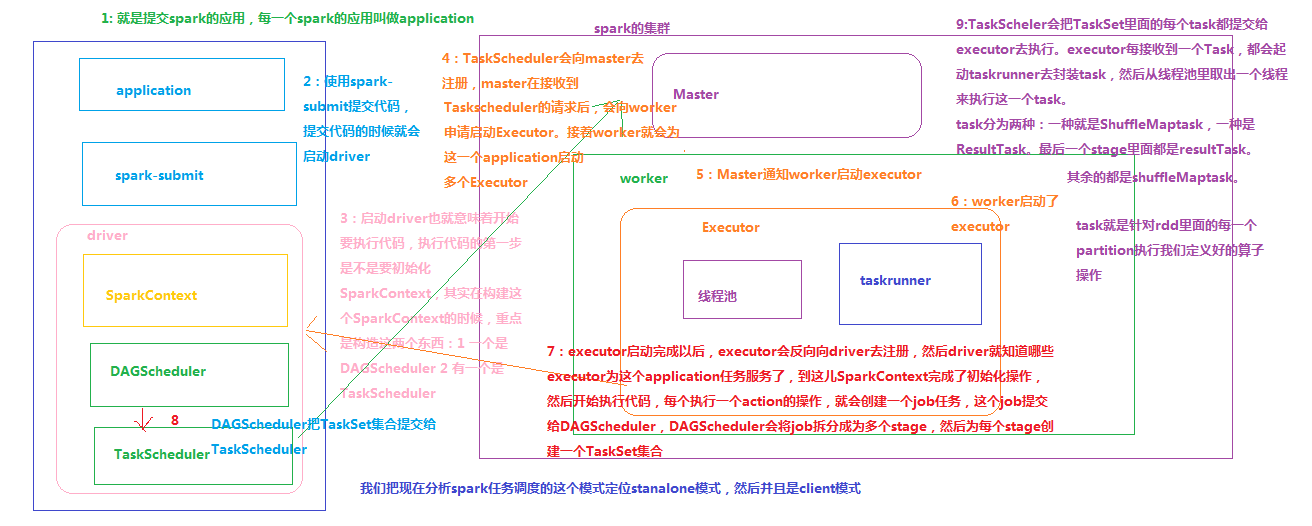
|  |  |
| --- | --- |
| Application | 用户提交的任务，提交到集群上面，那么这个就是一个Application |
| Application jar | A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing  their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries,  however, these will be added at runtime. |
| Driver program | The process running the main() function of the application and creating the SparkContext  （这个driver类似于YARN里面的ApplicationMaster） |
| Cluster manager | An external service for acquiring resources on the cluster (e.g. **standalone** manager, Mesos, **YARN**) |
| Deploy mode | Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the  cluster. In "client" mode, the submitter launches the driver outside of the cluster. |
| Worker node | Any node that can run application code in the cluster |
| Executor | A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage  across them. Each application has its own executors. |
| Task | A unit of work that will be sent to one executor |
| Job | A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action  (e.g.save, collect); you'll see this term used in the driver's logs. |
| Stage | Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and  reduce stages in MapReduce); you'll see this term used in the driver's logs. |

首先我们开发完一个应用以后，把这个应用提交到集群，那么这个应用就叫做Application，这个应用里面我们开发了好多代码，这些代码里面凡是遇到一个action操作，就会产生一个job任务。也就意味着，一个Application有一个或者一个以上的job任务。然后这些job任务划分为不同stage去执行，这个stage里面就是运行不同的task任务。Task计算的就是分区上面的数据。



在调优的时候可以从界面上看到每一个stage的运行情况，需要自己根据代码判断stage的划分，然后对性能差的stage进行代码定位。

## 第20课时 Spark任务调度



# Spark调优

**为什么调优？**

由于Spark的计算本质是基于内存的，所以Spark性能程序的性能可能因为集群中的任何因素出现瓶颈：CPU、网络带宽、或者是内存。如果内存能够容纳得下所有的数据，那么网络传输和通信就会导致性能出现瓶颈。但是如果内存比较紧张，不足以放下所有的数据（比如在针对10亿以上的数据量进行计算时），还是需要对内存的使用进行性能优化的，比如说使用一些手段来减少内存的消耗。

通常情况下来说，如果Spark应用程序计算的数据量比较小，并且你的内存足够使用，那么只要运维可以保障网络通畅，一般是不会有大的性能问题的。但是Spark应用程序的性能问题往往出现在针对大数据量（比如10亿级别）进行计算时出现

大部分Spark作业经过调优以后能进行较高性能跑起来了，但是在不同的生产环境和项目背景下，可能会遇到其他更加棘手的问题（比如各种**数据倾斜**），也可能会遇到更高的性能要求。为了应对这些挑战，需要使用更高级的技巧来处理这类问题。

基于上述背景，所以spark工程师岗位对人员要求高，所以要想成为spark的高手就必须会调优。

**性能优化主要包括的方向：**

**一：开发调优**

**二：数据本地化调优**

**三：数据倾斜调优**

**四：shuffle调优**

**五：JVM调优**

## 第21课时 开发调优

### 1.对多次使用的RDD进行持久化

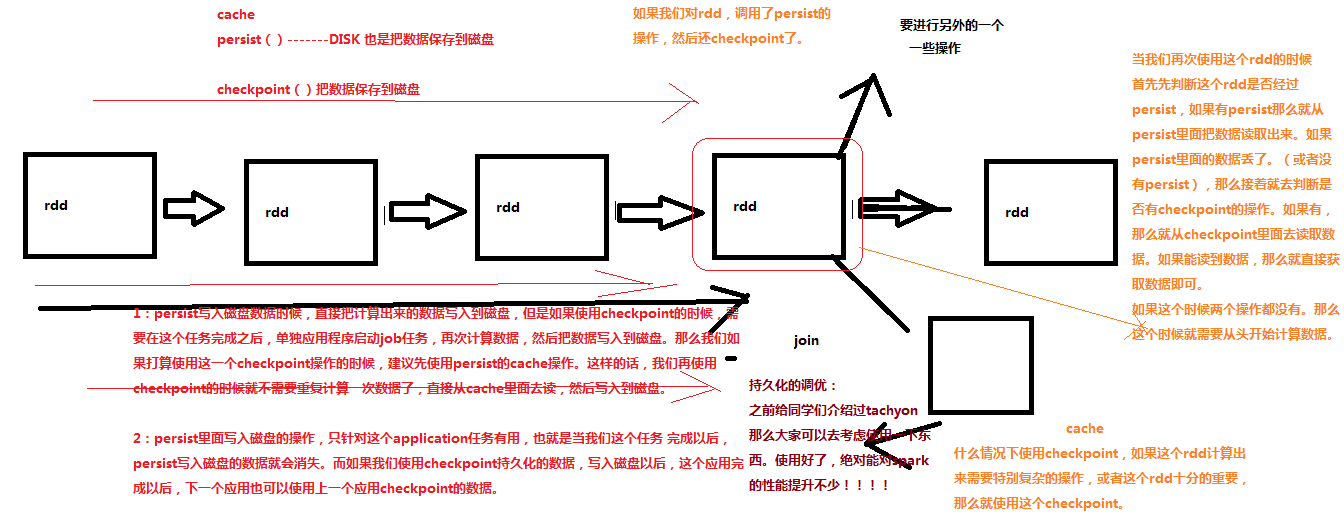
      Spark中对于一个RDD执行多次算子的默认原理是这样的：每次你对一个RDD执行一个算子操作时，都会重新从源头处计算一遍，计算出那个RDD来，然后再对这个RDD执行你的算子操作。这种方式的性能是很差的。

      因此对于这种情况，我们的建议是：**对多次使用的RDD进行持久化。此时Spark就会根据你的持久化策略，将RDD中的数据保存到内存或者磁盘中。以后每次对这个RDD进行算子操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后执行算子，而不会从源头处重新计算一遍这个RDD，再执行算子操作。**

Cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。 也就是说需要 checkpoint 的 RDD 会被计算两次。因此，在使用 rdd.checkpoint() 的时候，建议加上 rdd.cache()， 这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。

**cache 与 checkpoint 的区别？**

rdd.persist(StorageLevel.DISK\_ONLY) 与 checkpoint 也有区别。前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程 CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个 blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动 remove 掉，是一直存在的，也就是说可以被下一个 driver program 使用，而 cached RDD 不能被其他 dirver program 使用。



### 2.避免对相同数据创建重复的RDD

通常来说，我们在开发一个Spark作业时，首先是基于某个数据源（比如Hive表或HDFS文件）创建一个初始的RDD；接着对这个RDD执行某个算子操作，然后得到下一个RDD；以此类推，循环往复，直到计算出最终我们需要的结果。在这个过程中，多个RDD会通过不同的算子操作（比如map、reduce等）串起来，这个“RDD串”，就是RDD lineage，也就是“RDD的血缘关系链”。

      我们在开发过程中要注意：**对于同一份数据，只应该创建一个RDD，不能创建多个RDD来代表同一份数据。**

      一些Spark初学者在刚开始开发Spark作业时，或者是有经验的工程师在开发RDD lineage极其冗长的Spark作业时，可能会忘了自己之前对于某一份数据已经创建过一个RDD了，从而导致对于同一份数据，创建了多个RDD。这就意味着，我们的Spark作业会进行多次重复计算来创建多个代表相同数据的RDD，进而增加了作业的性能开销。

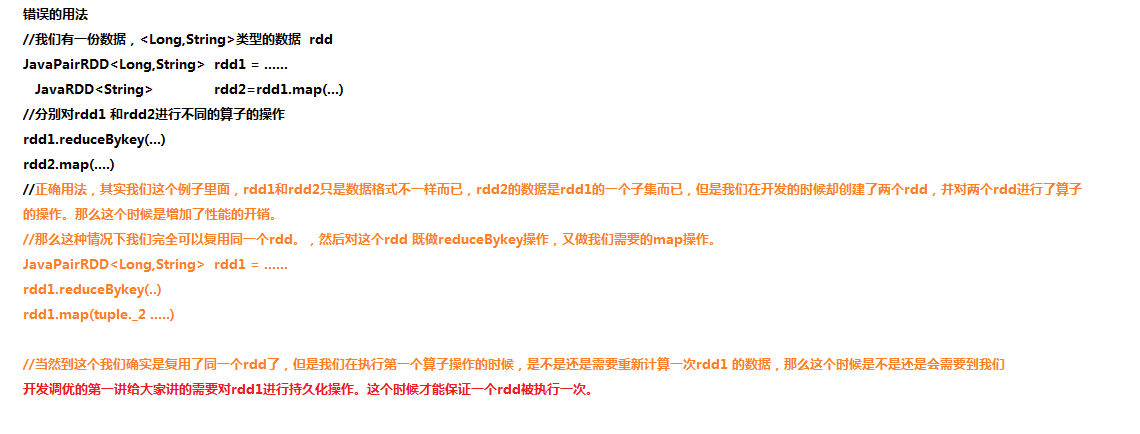
**案例演示**



### 3.尽可能复用同一个RDD

除了要避免在开发过程中对一份完全相同的数据创建多个RDD之外，在对不同的数据执行算子操作时还要尽可能地复用一个RDD。比如说，有一个RDD的数据格式是key-value类型的，另一个rdd是单value类型的，这两个RDD的value数据是完全一样的。那么此时我们可以只使用key-value类型的那个RDD，因为其中已经包含了另一个的数据。对于类似这种**多个RDD的数据有重叠或者包含的情况，我们应该尽量复用一个RDD，这样可以尽可能地减少RDD的数量，从而尽可能减少算子执行的次数。**

案例演示



### 4.尽量避免使用shuffle类算子

     如果有可能的话，要尽量避免使用shuffle类算子。因为Spark作业运行过程中，最消耗性能的地方就是shuffle过程。shuffle过程，简单来说，就是将分布在集群中多个节点上的同一个key，拉取到同一个节点上，进行聚合或join等操作。比如reduceByKey、join等算子，都会触发shuffle操作。

      shuffle过程中，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。因此在shuffle过程中，可能会发生大量的磁盘文件读写的IO操作，以及数据的网络传输操作。磁盘IO和网络数据传输也是shuffle性能较差的主要原因。

      因此在我们的开发过程中，能避免则尽可能避免使用reduceByKey、join、distinct、repartition等会进行shuffle的算子，尽量使用map类的非shuffle算子。这样的话，没有shuffle操作或者仅有较少shuffle操作的Spark作业，可以大大减少性能开销。

**spark中会导致shuffle操作的有以下几种算子**

1、repartition类的操作：比如repartition、repartitionAndSortWithinPartitions、coalesce等

2、byKey类的操作：比如reduceByKey、groupByKey、sortByKey等

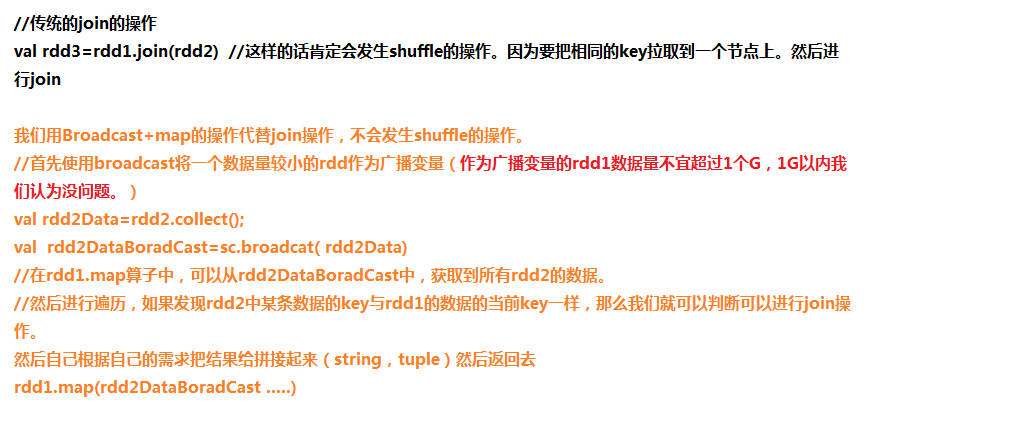
3、join类的操作：比如join、cogroup等

**重分区**: 一般会shuffle，因为需要在整个集群中，对之前所有的分区的数据进行随机，均匀的打乱，然后把数据放入下游新的指定数量的分区内

**byKey类的操作**：因为你要对一个key，进行聚合操作，那么肯定要保证集群中，所有节点上的，相同的key，一定是到同一个节点上进行处理

**join类的操作**：两个rdd进行join，就必须将相同join key的数据，shuffle到同一个节点上，然后进行相同key的两个rdd数据的笛卡尔乘积

**案例演示**

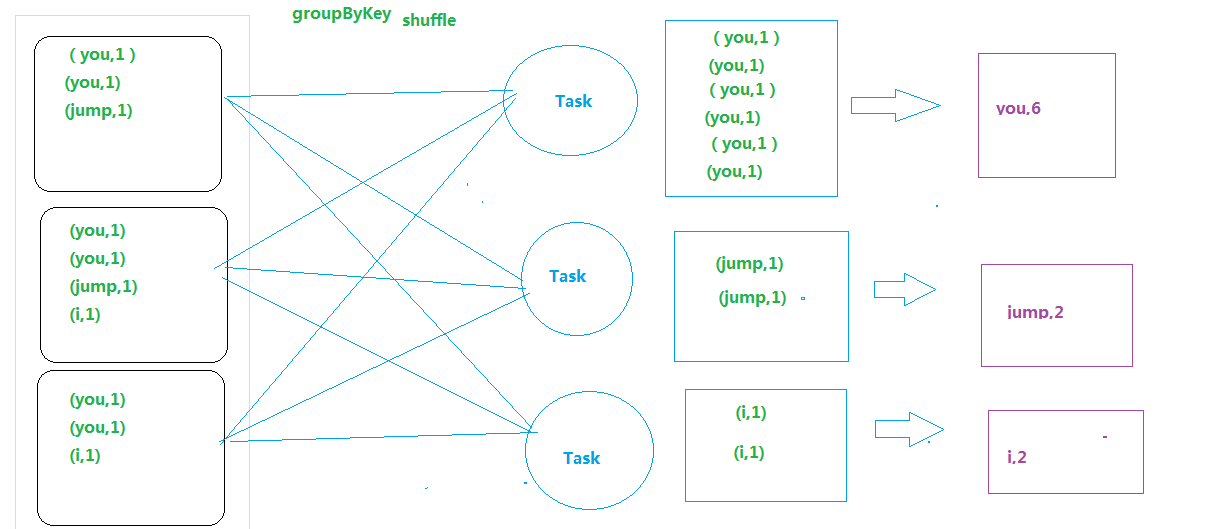


### 5.使用map-side预聚合的shuffle操作

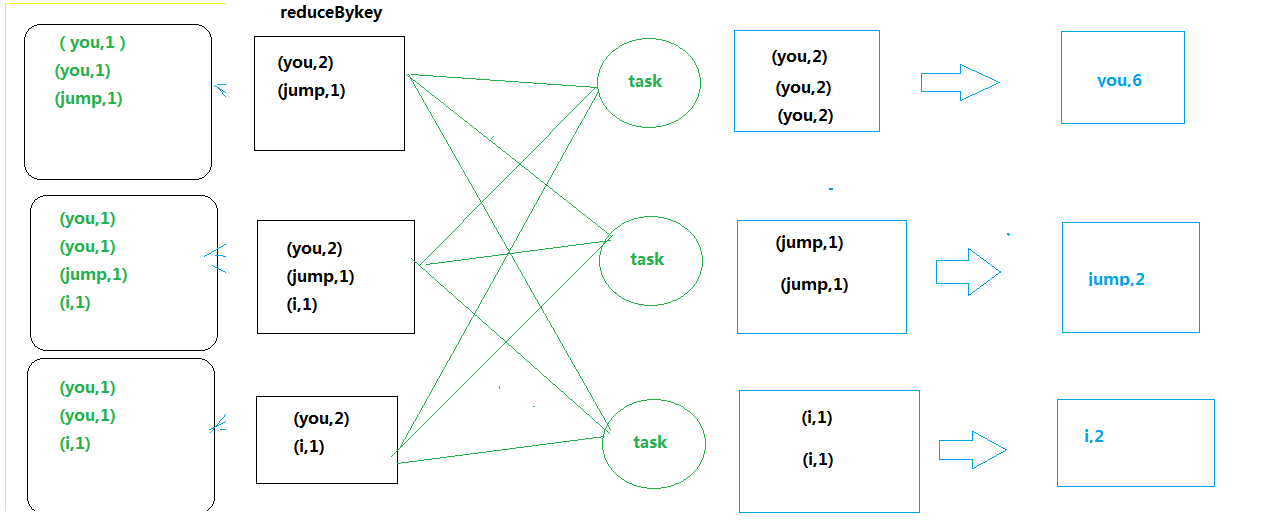
如果因为业务需要，一定要使用shuffle操作，无法用map类的算子来替代，那么尽量使用可以map-side预聚合的算子。所谓的map-side预聚合，说的是在每个节点本地对相同的key进行一次聚合操作，类似于MapReduce中的本地combin。map-side预聚合之后，每个节点本地就只会有一条相同的key，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘IO以及网络传输开销。通常来说，在可能的情况下**，建议使用reduceByKey或者aggregateByKey算子来替代掉groupByKey算子。因为reduceByKey和aggregateByKey算子都会使用用户自定义的函数对每个节点本地的相同key进行预聚合。而groupByKey算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。**

**原理剖析**

groupByKey原理图



reduceByKey原理图



### 6.使用高性能的算子

除了shuffle相关的算子有优化原则之外，其他的算子也都有着相应的优化原则。

**使用reduceByKey/aggregateByKey替代groupByKey**

      详情见：使用map-side预聚合的shuffle操作”。

**使用mapPartitions替代普通map**

      mapPartitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。但是有的时候，使用mapPartitions会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！

**使用foreachPartitions替代foreach**

      原理类似于“使用mapPartitions替代map”，也是一次函数调用处理一个partition的所有数据，而不是一次函数调用处理一条数据。在实践中发现，foreachPartitions类的算子，对性能的提升还是很有帮助的。比如在foreach函数中，将RDD中所有数据写**[MySQL](http://lib.csdn.net/base/14" \o "MySQL知识库" \t "http://blog.csdn.net/u012102306/article/details/_blank)**，那么如果是普通的foreach算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个**[数据库](http://lib.csdn.net/base/14" \o "MySQL知识库" \t "http://blog.csdn.net/u012102306/article/details/_blank)**连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用foreachPartitions算子一次性处理一个partition的数据，那么对于每个partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。实践中发现，对于1万条左右的数据量写MySQL，性能可以提升30%以上。

**使用filter之后进行coalesce操作**

      通常对一个RDD执行filter算子过滤掉RDD中较多数据后（比如30%以上的数据），建议使用coalesce算子，手动减少RDD的partition数量，将RDD中的数据压缩到更少的partition中去。因为filter之后，RDD的每个partition中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个task处理的partition中的数据量并不是很多，有一点资源浪费，而且此时处理的task越多，可能速度反而越慢。因此用coalesce减少partition数量，将RDD中的数据压缩到更少的partition之后，只要使用更少的task即可处理完所有的partition。在某些场景下，对于性能的提升会有一定的帮助。

**使用repartitionAndSortWithinPartitions替代repartition与sort类操作**

      repartitionAndSortWithinPartitions是Spark官网推荐的一个算子，官方建议，如果需要在repartition重分区之后，还要进行排序，建议直接使用repartitionAndSortWithinPartitions算子。因为该算子可以一边进行重分区的shuffle操作，一边进行排序。shuffle与sort两个操作同时进行，比先shuffle再sort来说，性能可能是要高的。

### 7.广播大变量

  有时在开发过程中，会遇到需要在算子函数中使用外部变量的场景（尤其是大变量，比如100M以上的大集合），那么此时就应该使用Spark的广播（Broadcast）功能来提升性能。

      在算子函数中使用到外部变量时，默认情况下，Spark会将该变量复制多个副本，通过网络传输到task中，此时每个task都有一个变量副本。如果变量本身比较大的话（比如100M，甚至1G），那么大量的变量副本在网络中传输的性能开销，以及在各个节点的Executor中占用过多内存导致的频繁GC，都会极大地影响性能。

      因此对于上述情况，如果使用的外部变量比较大，建议使用Spark的广播功能，对该变量进行广播。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，而Executor中的task执行时共享该Executor中的那份变量副本。这样的话，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对Executor内存的占用开销，降低GC的频率。

### 8.使用Kryo优化序列化性能

 在Spark中，主要有三个地方涉及到了序列化：

* 1）在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输（见广播大变量”中的讲解）。
* 2）将自定义的类型作为RDD的泛型类型时（比如JavaRDD，Student是自定义类型），所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现Serializable接口。
* 3）使用可序列化的持久化策略时（比如MEMORY\_ONLY\_SER），Spark会将 RDD中的每个partition都序列化成一个大的字节数组。

      对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库，来优化序列化和反序列化的性能。**Spark默认使用的是[Java](http://lib.csdn.net/base/17" \o "Java EE知识库" \t "http://blog.csdn.net/u012102306/article/details/_blank)的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介绍，Kryo序列化机制比Java序列化机制，性能高10倍左右。**Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说，这种方式比较麻烦。

**案例演示**

SparkConf().set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

Scala版本：

val conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Array(classOf[Counter] ))

val sc = new SparkContext(conf)

Java版本：

SparkConf conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Counter.class)

JavaSparkContext sc = new JavaSparkContext(conf)

如果注册的要序列化的自定义的类型，本身特别大，比如包含了超过100个field。那么就会导致要序列化的对象过大。此时就需要对Kryo本身进行优化。因为Kryo内部的缓存可能不够存放那么大的class对象。此时就需要调用SparkConf.set()方法，设置spark.kryoserializer.buffer.mb参数的值，将其调大。默认情况下它的值是2，就是说最大能缓存2M的对象，然后进行序列化。可以在必要时将其调大。比如设置为10。

### 优化数据结构

Java中，有三种类型比较耗费内存：

1）对象，每个Java对象都有对象头、引用等额外的信息，因此比较占用内存空间。

2）字符串，每个字符串内部都有一个字符数组以及长度等额外信息。

3）集合类型，比如HashMap、LinkedList等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如Map.Entry。

因此Spark官方建议，在Spark编码实现中，特别是对于算子函数中的代码，尽量不要使用上述三种**[数据结构](http://lib.csdn.net/base/31" \o "算法与数据结构知识库" \t "http://blog.csdn.net/u012102306/article/details/_blank)**，尽量使用字符串替代对象，使用原始类型（比如Int、Long）替代字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低GC频率，提升性能。

建议：我们在实际的开发中，要做到如上所述，其实不容易。我们要考虑到代码可维护性，如果一个代码里面，完全没有抽象，全部都是字符串的拼接，对于后面的代码维护和修改难度很大。

所以我们应该考虑在合适的时候采取考虑这样的优化方式，因为我们首先得考虑代码的维护性。

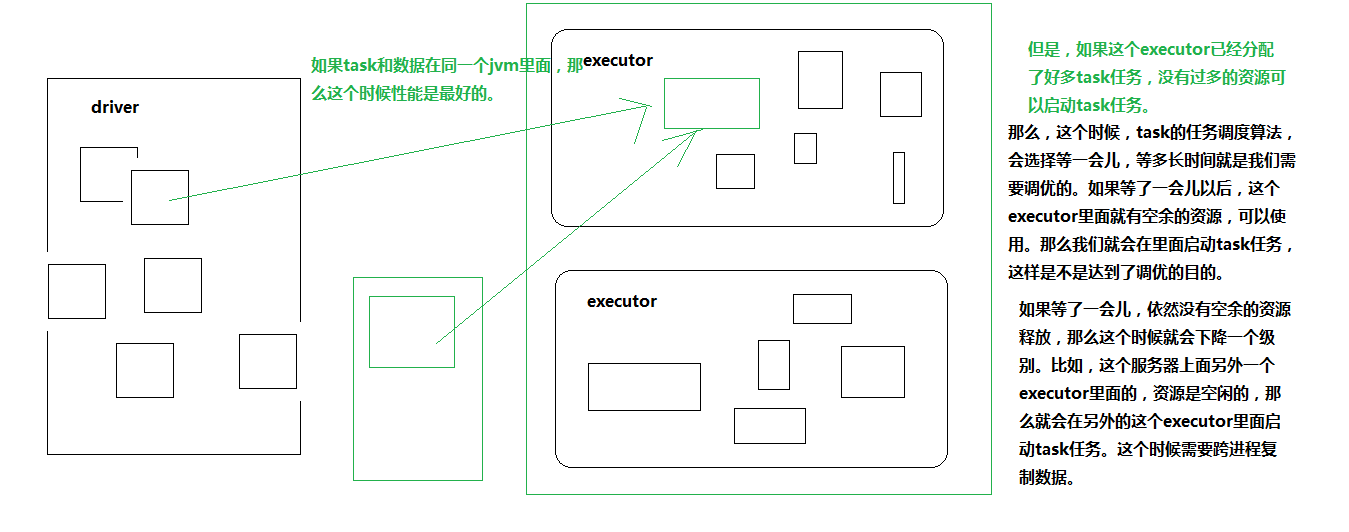
## 第22课时 数据本地化

**数据本地化的背景:**

数据本地化对于Spark Job性能有着巨大的影响。如果数据以及要计算它的代码是在一起的，那么性能当然会非常高。但是，如果数据和计算它的代码是分开的，那么其中之一必须到另外一方的机器上。通常来说，移动代码到其他节点，会比移动数据到代码所在的节点上去，速度要快得多，因为代码比较小。Spark也正是基于这个数据本地化的原则来构建task调度算法的。

数据本地化，指的是，数据离计算它的代码有多近。基于数据距离代码的距离，有几种数据本地化级别：  
1、PROCESS\_LOCAL：数据和计算它的代码在同一个JVM进程中。  
2、NODE\_LOCAL：数据和计算它的代码在一个节点上，但是不在一个进程中，比如在不同的executor进程中，或者是数据在HDFS文件的block中。  
3、NO\_PREF：从任何地方访问数据速度都是一样，不关心数据的位置。  
4、RACK\_LOCAL：数据和计算它的代码在一个机架上。  
5、ANY：数据可能在任意地方，比如其他网络环境内，或者其他机架上。





建议：越往前的级别等待的时间可以调的长一点。

## 第23课时 数据倾斜原理

### 1.数据倾斜原理

**调优概述**

      有的时候，**我们可能会遇到[大数据](http://lib.csdn.net/base/20" \o "Hadoop知识库" \t "http://blog.csdn.net/u012102306/article/details/_blank)计算中一个最棘手的问题——数据倾斜，此时Spark作业的性能会比期望差很多。数据倾斜调优，就是使用各种技术方案解决不同类型的数据倾斜问题，以保证Spark作业的性能。**

**1.1数据倾斜发生时的现象**

绝大多数task执行得都非常快，但个别task执行极慢。比如，总共有1000个task，997个task都在1分钟之内执行完了，但是剩余两三个task却要一两个小时。这种情况很常见。

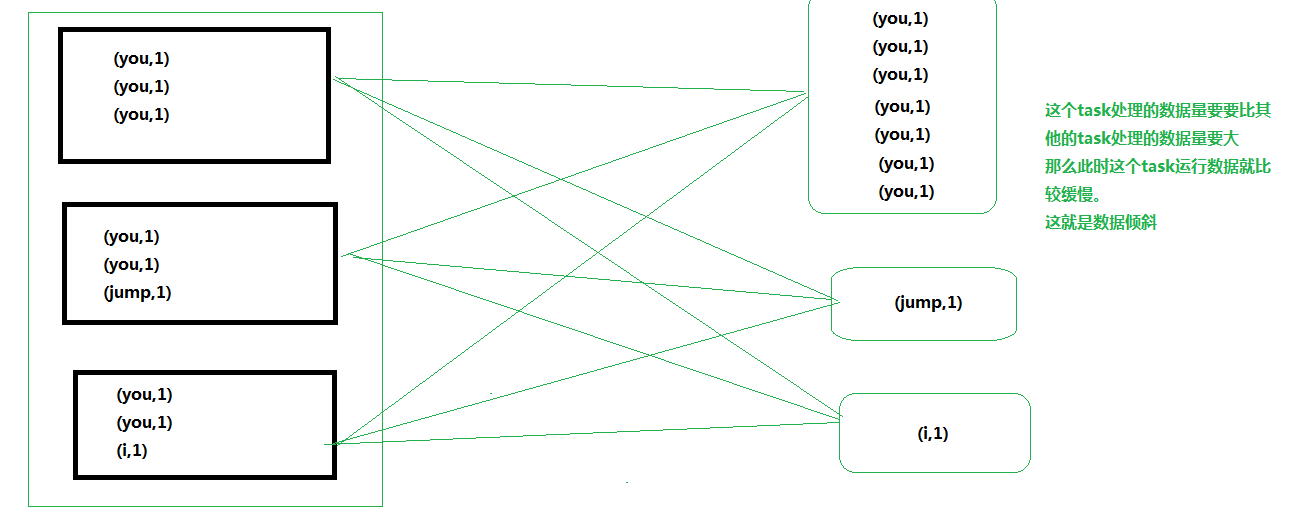
原本能够正常执行的Spark作业，某天突然报出OOM（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

**1.2数据倾斜发生的原理**

      数据倾斜的原理很简单：在进行shuffle的时候，必须将各个节点上相同的key拉取到某个节点上的一个task来进行处理，比如按照key进行聚合或join等操作。此时如果某个key对应的数据量特别大的话，就会发生数据倾斜。比如大部分key对应10条数据，但是个别key却对应了100万条数据，那么大部分task可能就只会分配到10条数据，然后1秒钟就运行完了；但是个别task可能分配到了100万数据，要运行一两个小时。因此，整个Spark作业的运行进度是由运行时间最长的那个task决定的。

      因此出现数据倾斜的时候，Spark作业看起来会运行得非常缓慢，甚至可能因为某个task处理的数据量过大导致内存溢出。

举例



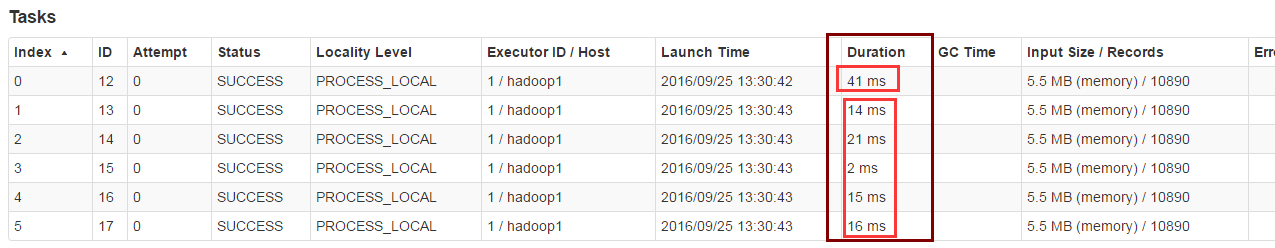
**1.3如何定位发生数据倾斜的代码**

**1）数据倾斜只会发生在shuffle过程中。**这里给大家罗列一些常用的并且可能会触发shuffle操作的算子：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。出现数据倾斜时，可能就是你的代码中使用了这些算子中的某一个所导致的。

**2）通过观察spark UI的界面，定位数据倾斜发生在第几个stage中。**

如果是用yarn-client模式提交，那么本地是直接可以看到log的，可以在log中找到当前运行到了第几个stage；如果是用yarn-cluster模式提交，则可以通过Spark Web UI来查看当前运行到了第几个stage。此外，无论是使用yarn-client模式还是yarn-cluster模式，我们都可以在Spark Web UI上深入看一下当前这个stage各个task分配的数据量，从而进一步确定是不是task分配的数据不均匀导致了数据倾斜。

1. 根据之前学的stage的划分算法定位到极有可能发生数据倾斜的代码



**1.4查看导致数据倾斜的key的分布情况**

1.如果是Spark SQL中的group by、join语句导致的数据倾斜，那么就查询一下SQL中使用的表的key分布情况。

2.如果是对Spark RDD执行shuffle算子导致的数据倾斜，那么可以在Spark作业中加入查看key分布的代码，比如RDD.countByKey()。然后对统计出来的各个key出现的次数，collect/take到客户端打印一下，就可以看到key的分布情况。

val sampledPairs = pairs.sample(false, 0.1)

val sampledWordCounts = sampledPairs.countByKey()

sampledWordCounts.foreach(println(\_))

## 第24课时 数据倾斜解决方案

### 方案一：使用Hive ETL预处理数据

**方案适用场景**：导致数据倾斜的是Hive表。如果该Hive表中的数据本身很不均匀（比如某个key对应了100万数据，其他key才对应了10条数据），而且业务场景需要频繁使用Spark对Hive表执行某个分析操作，那么比较适合使用这种技术方案。

**方案实现思路**：此时可以评估一下**，是否可以通过Hive来进行数据预处理**（即通过Hive ETL预先对数据按照key进行聚合，或者是预先和其他表进行join），然后在Spark作业中针对的数据源就不是原来的Hive表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在Spark作业中也就不需要使用原先的shuffle类算子执行这类操作了。

**方案实现原理**：这种方案从根源上解决了数据倾斜，因为彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以Hive ETL中进行group by或者join等shuffle操作时，还是会出现数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中，避免Spark程序发生数据倾斜而已。

**方案优点**：实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提升。

**方案缺点**：治标不治本，Hive ETL中还是会发生数据倾斜。

**方案实践经验：**在一些**[Java](http://lib.csdn.net/base/17" \o "Java EE知识库" \t "http://blog.csdn.net/u012102306/article/details/_blank)**系统与Spark结合使用的项目中，会出现Java代码频繁调用Spark作业的场景，而且对Spark作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次Java调用Spark作业时，执行速度都会很快，能够提供更好的用户体验。

### 方案二：过滤少数导致倾斜的key

**方案适用场景**：**如果发现导致倾斜的key就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。**比如99%的key就对应10条数据，但是只有一个key对应了100万数据，从而导致了数据倾斜。

**方案实现思路**：如果我们判断那少数几个数据量特别多的key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个key。比如，在Spark SQL中可以使用where子句过滤掉这些key或者在Spark Core中对RDD执行filter算子过滤掉这些key。如果需要每次作业执行时，动态判定哪些key的数据量最多然后再进行过滤，那么可以使用sample算子对RDD进行采样，然后计算出每个key的数量，取数据量最多的key过滤掉即可。

**方案实现原理**：将导致数据倾斜的key给过滤掉之后，这些key就不会参与计算了，自然不可能产生数据倾斜。

**方案优点**：实现简单，而且效果也很好，可以完全规避掉数据倾斜。

**方案缺点**：适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个。

**方案实践经验**：在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天Spark作业在运行的时候突然OOM了，追查之后发现，是Hive表中的某一个key在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个key之后，直接在程序中将那些key给过滤掉。

### 方案三：提高shuffle操作的并行度

**方案适用场景**：**如果我们必须要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。**

**方案实现思路**：在对RDD执行shuffle算子时，给shuffle算子传入一个参数，比如reduceByKey(1000)，该参数就设置了这个shuffle算子执行时shuffle read task的数量。对于Spark SQL中的shuffle类语句，比如group by、join等，需要设置一个参数，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，该值默认是200，对于很多场景来说都有点过小。

**方案实现原理**：增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了。

**方案优点**：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

**方案缺点：**只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。

**方案实践经验**：该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用嘴简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。

原理演示

