

# 目录

简介	1.1
第一章：HDFS	1.2
1.1 HDFS Report分析	1.2.1
第二章：MapReduce	1.3
2.1 详解Shuffle过程	1.3.1
第三章：Yarn	1.4
3.1 Yarn内存分配管理机制及相关参数配置	1.4.1
3.2 Yarn调度器Scheduler详解	1.4.2
第四章：Hadoop	1.5
4.1 GP-Hadoop-Elk的原理区别	1.5.1
第五章：Spark	1.6
5.1 跨集群访问HBase	1.6.1
第六章：Zookeeper	1.7
6.1 Basic Paxos算法	1.7.1
结束	1.8

# Introduction

- 简介
- 1. Hadoop
  - 1.1 HDFS
    - 1.1.1 HDFS Report分析
  - 1.2 MapReduce
  - 1.3 Yarn
  - 1.4 其他
    - 1.4.1 HDFS Report分析
- 2. Spark
  - 2.1 SparkStreaming
    - 2.1.1 跨集群访问HBase
  - 2.2 SparkSql
- 结束

## 1. HDFS

## 1.1 HDFS Report分析方法 -- hdfs dfsadmin

### hdfs dfsadmin -report

```
[root@CNSZ431014 ~]# hdfs dfsadmin -report
Configured Capacity: 74426056298496 (67.69 TB)
Present Capacity: 70124136352340 (63.78 TB)
DFS Remaining: 33715992505225 (30.66 TB)
DFS Used: 36408143847115 (33.11 TB)
DFS Used%: 51.92%
Under replicated blocks: 29904
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0

-----
Live datanodes (6):

Name: 10.14.18.207:25009 (CNSZ431018)
Hostname: CNSZ431018
Rack: /default/rack0
Decommission Status : Normal
Configured Capacity: 17720489594880 (16.12 TB)
DFS Used: 7213844099821 (6.56 TB)
Non DFS Used: 1017260168087 (947.40 GB)
DFS Remaining: 9489385326972 (8.63 TB)
DFS Used%: 40.71%
DFS Remaining%: 53.55%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 18
Last contact: Mon Dec 11 10:59:04 CST 2017

Name: 10.14.18.208:25009 (CNSZ431019)
Hostname: CNSZ431019
Rack: /default/rack0
Decommission Status : Normal
Configured Capacity: 17720489594880 (16.12 TB)
DFS Used: 7133852738211 (6.49 TB)
Non DFS Used: 1016951088951 (947.11 GB)
DFS Remaining: 9569685767718 (8.70 TB)
DFS Used%: 40.26%
DFS Remaining%: 54.00%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 16
Last contact: Mon Dec 11 10:59:04 CST 2017

Name: 10.14.18.206:25009 (CNSZ431017)
Hostname: CNSZ431017
Rack: /default/rack0
Decommission Status : Normal
Configured Capacity: 17720489594880 (16.12 TB)
DFS Used: 6728401972418 (6.12 TB)
Non DFS Used: 1018018554490 (948.10 GB)
DFS Remaining: 9974069067972 (9.07 TB)
DFS Used%: 37.97%
DFS Remaining%: 56.29%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
```

```

Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 19
Last contact: Mon Dec 11 10:59:04 CST 2017

```

```

Name: 10.14.18.203:25009 (CNSZ431014)
Hostname: CNSZ431014
Rack: /default/rack0
Decommission Status : Normal
Configured Capacity: 7088195837952 (6.45 TB)
DFS Used: 5117967231685 (4.65 TB)
Non DFS Used: 416702126051 (388.08 GB)
DFS Remaining: 1553526480216 (1.41 TB)
DFS Used%: 72.20%
DFS Remaining%: 21.92%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 23
Last contact: Mon Dec 11 10:59:04 CST 2017

```

```

Name: 10.14.18.204:25009 (CNSZ431015)
Hostname: CNSZ431015
Rack: /default/rack0
Decommission Status : Normal
Configured Capacity: 7088195837952 (6.45 TB)
DFS Used: 4596038181902 (4.18 TB)
Non DFS Used: 417036831431 (388.40 GB)
DFS Remaining: 2075120824619 (1.89 TB)
DFS Used%: 64.84%
DFS Remaining%: 29.28%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 22
Last contact: Mon Dec 11 10:59:03 CST 2017

```

```

Name: 10.14.18.205:25009 (CNSZ431016)
Hostname: CNSZ431016
Rack: /default/rack0
Decommission Status : Normal
Configured Capacity: 7088195837952 (6.45 TB)
DFS Used: 5618039623078 (5.11 TB)
Non DFS Used: 415951177146 (387.38 GB)
DFS Remaining: 1054205037728 (981.80 GB)
DFS Used%: 79.26%
DFS Remaining%: 14.87%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 28
Last contact: Mon Dec 11 10:59:04 CST 2017

```

## hdbs dfsadmin -safemode 安全模式

NameNode在启动的时候首先进入安全模式，如果datanode丢失的block达到一定的比例（由hdfs-site.xml文件中dfs.safemode.threshold.pct决定，默认0.999f），则系统会一直处于安全模式状态即只读状态；否则没有其他情况影响，一般情况下，系统会自动离开安全模式。

dfs.safemode.threshold.pct 表示HDFS启动的时候，如果DataNode上报的 block 个数0.999倍才可以离开安全模式，否则一直是这种只读状态。如果设为1则hdfs永远是处于SafeMode。

通常两种情况可以离开这处安全模式：

1、修改 dfs.safemode.threshold.pct为一个比较小的值，缺省值是0.999 2、hadoop dfsadmin -safemode leave 命令强制离开

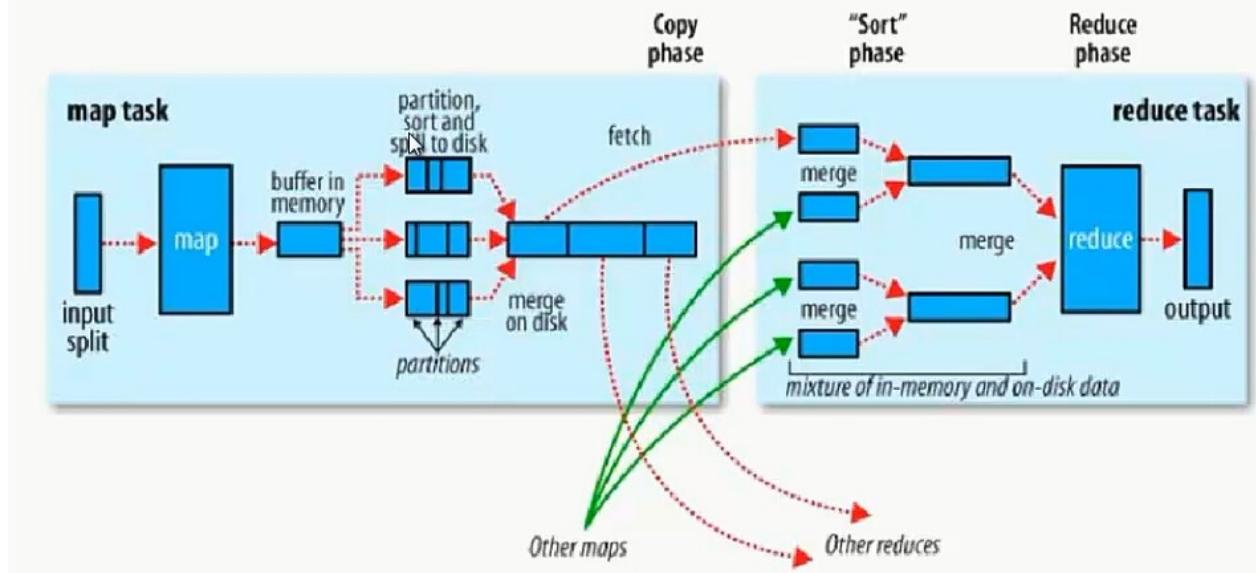
用户可以使用命令行（hdfs dfsadmin -safemode value）做如下的操作：

```
# hdfs dfsadmin -safemode get    ## 返回安全模式是否开启的信息，返回 Safe mode is OFF/OPEN
# hdfs dfsadmin -safemode enter  ## 进入安全模式
# hdfs dfsadmin -safemode leave  ## 强制 NameNode 离开安全模式
# hdfs dfsadmin -safemode wait   ## 等待，一直到安全模式结束
```

## 2. MapReduce

## 2.1 详解Shuffle过程

- Hadoop计算框架Shuffler



Shuffle过程，也称Copy阶段。reduce task从各个map task上远程拷贝一片数据，并针对某一片数据，如果其大小超过一定的阈值，则写到磁盘上，否则直接放到内存中。

官方的Shuffle过程如上图所示，不过细节有错乱，官方图并没有说明partition、sort和combiner具体作用于哪个阶段。

注意：Shuffle过程是贯穿于map和reduce两个过程的！

Hadoop的集群环境，大部分的map task和reduce task是执行在不同的节点上的，那么reduce就要取map的输出结果。那么集群中运行多个Job时，task的正常执行会对集群内部的网络资源消耗严重。虽说这种消耗是正常的，是不可避免的，但是，我们可以采取措施尽可能的减少不必要的网络资源消耗。另一方面，每个节点的内部，相比于内存，磁盘IO对Job完成时间的影响相当的大，。

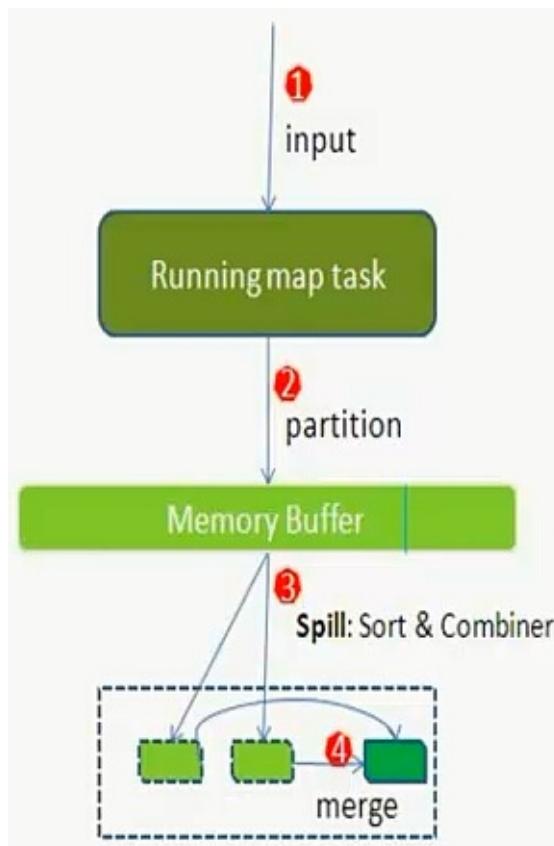
所以：从以上分析，shuffle过程的基本要求：

1. 完整地从map task端拉取数据到reduce task端
2. 在拉取数据的过程中，尽可能地减少网络资源的消耗
3. 尽可能地减少磁盘IO对task执行效率的影响

那么，Shuffle的设计目的就要满足以下条件：

1. 保证拉取数据的完整性
2. 尽可能地减少拉取数据的数据量
3. 尽可能地使用节点的内存而不是磁盘

**map端：**



说明：map节点执行map task任务生成map的输出结果。

shuffle的工作内容：

从运算效率的出发点，map输出结果优先存储在map节点的内存中。每个map task都有一个内存缓冲区，存储着map的输出结果，当缓冲区块满时，需要将缓冲区中的数据以一个临时文件的方式存到磁盘，当整个map task结束后再对磁盘中这个map task所产生的所有临时文件做合并，生成最终的输出文件。最后，等待reduce task来拉取数据。当然，如果map task的结果不大，能够完全存储到内存缓冲区，且未达到内存缓冲区的阀值，那么就不会有写临时文件到磁盘的操作，也不会有后面的合并。

详细过程如下：

1.map task任务执行，输入数据的来源是：HDFS的block。当然在mapreduce概念中，map task读取的是split分片。split与block的对应关系：一对一（默认）。

此处有必要说明一下block与split

block（物理划分）：

文件上传到HDFS，就要划分数据成块，这里的划分属于物理的划分，块的大小可配置（默认：第一代为64M，第二代为128M）可通过dfs.block.size配置。为保证数据的安全，block采用冗余机制：默认为3份，可通过dfs.replication配置。注意：当更改块大小的配置后，新上传的文件的块大小为新配置的值，以前上传的文件的块大小为以前的配置值。

split（逻辑划分）：

Hadoop中split划分属于逻辑上的划分，目的只是为了让map task更好地获取数据。split是通过hadoop中的InputFormat接口中的getSplit（）方法得到的。那么，split的大小具体怎么得到呢？

首先介绍几个数据量：

totalSize：整个mapreduce job所有输入的总大小。注意：基本单位是block个数，而不是Bytes个数。

numSplits：来自job.getNumMapTasks()，即在job启动时用户利用org.apache.hadoop.mapred.JobConf.setNumMapTasks(int n)设置的值，从方法的名称上看，是用于设置map的个数。但是，最终map的个数也就是split的个数并不一定取用户设置的这个值，用户设置的map个数值只是给最终的map个数一个提示，只是一个影响因素，而不是决定因素。

goalSize : totalSize/numSplits，即期望的split的大小，也就是每个mapper处理多少的数据。但是仅仅是期望

minSize : split的最小值，该值可由两个途径设置：1.子类复写函数protected void setMinSplitSize(long minSplitSize)设置。一般情况为1，特殊情况除外 2.配置文件中的mapred.min.split.size设置 最终取两者中的最大值！

最终：split大小的计算原则：finalSplitSize=max(minSize,min(goalSize,blockSize))

那么，map的个数=totalSize/finalSplitSize

注意：新版的API中InputSplit划分算法不再考虑用户设定的Map Task个数，而是用mapred.max.split.size(记为maxSize)代替 即：InputSplit大小的计算公式为：splitSize=max{minSize,min{maxSize,blockSize}}

接下来就简答说说怎么根据业务需求，调整map的个数。当我们用hadoop处理大批量的大数据时，一种最常见的情况就是job启动的mapper数量太多而超出系统限制，导致hadoop抛出异常终止执行。

解决方案：减少mapper的数量！具体如下：

#### 1. 输入文件数量巨大，但不是小文件

这种情况可通过增大每个mapper的inputsize，即增大minSize或者增大blockSize来减少所需的mapper的数量。增大blocksize通常不可行，因为HDFS被hadoop namenode -format之后，blocksize就已经确定了(由格式化时dfs.block.size决定)，如果要更改blocksize，需要重新格式化HDFS，这样当然会丢失已有的数据。所以通常情况下只能增大minSize，即增大mapred.min.split.size的值。

#### 2. 输入文件数量巨大，且都是小文件

所谓小文件，就是单个文件的size小于blockSize。这种情况通过增大mapred.min.split.size不可行，需要使用FileInputFormat衍生的CombineFileInputFormat将多个input path合并成一个InputSplit送给mapper处理，从而减少mapper的数量。增加mapper的数量，可以通过减少每个mapper的输入做到，即减小blockSize或者减少mapred.min.split.size的值。

block与split关系说清楚了，那先说到这里，还是回到shuffle的过程解说中来！

1. map执行后，得到key/value键值对。接下来的问题就是，这些键值对应该交给哪个reduce做？注意：reduce的个数是允许用户在提交job时，通过设置方法设置的！

MapReduce提供partitioner接口解决上述问题。默认操作是：对key hash后再以reduce task数量取模，返回值决定着该键值对应该由哪个reduce处理。这种默认的取模方式只是为了平均reduce的处理能力，防止数据倾斜，保证负载均衡。

如果用户自己对Partition有需求，可以自行定制并设置到job上。

接下来，需要将key/value以及Partition结果都写入到缓冲区，缓冲区的作用：批量收集map结果，减少磁盘IO的影响。

当然，写入之前，这些数据都会被序列化成字节数组。而整个内存缓冲区就是一个字节数组。这个内存缓冲区是有大小限制的，默认100MB。当map task的输出结果很多时，就可能撑爆内存。需将缓冲区的数据临时写入磁盘，然后重新利用这块缓冲区。从内存往磁盘写数据被称为Spill(溢写)，由单独线程完成，不影响往缓冲区写map结果的线程。溢写比例：spill.percent(默认0.8)。当缓冲区的数据达到阀值，溢写线程启动，锁定这80MB的内存，执行溢写过程。剩下的20MB继续写入map task的输出结果。互不干涉！当溢写线程启动后，需要对这80MB空间内的key做排序(Sort)。排序是mapreduce模型的默认行为，也是对序列化的字节做的排序。排序规则：字典排序！

map task的输出结果写入内存后，当溢写线程未启动时，对输出结果并没有做任何的合并。从官方图可以看出，合并是体现在溢写的临时磁盘文件上的，且这种合并是对不同的reduce端的数值做的合并。所以溢写过程一个很重要的细节在于，如果有多个key/value对需要发送到某个reduce端，那么需要将这些键值对拼接到一块，减少与partition相关的索引记录。如果client设置过Combiner，其会将有相同key的key/value对的value加起来，减少溢写到磁盘的数据量。注意：这里的合并并不能保证map结果中所有的相同的key值的键值对的value都合并了，它合并的范围只是这80MB，它能保证的是在每个单独的溢写文件中所有键值对的key值均不相同！

溢写生成的临时文件的个数随着map输出结果的数据量变大而增多，当整个map task完成，内存中的数据也全部溢写到磁盘的一个溢写文件。也就是说，不论任何情况下，溢写过程生成的溢写文件至少有一个！但是最终的文件只能有一个，需要将这些溢写文件归并到一起，称为merge。

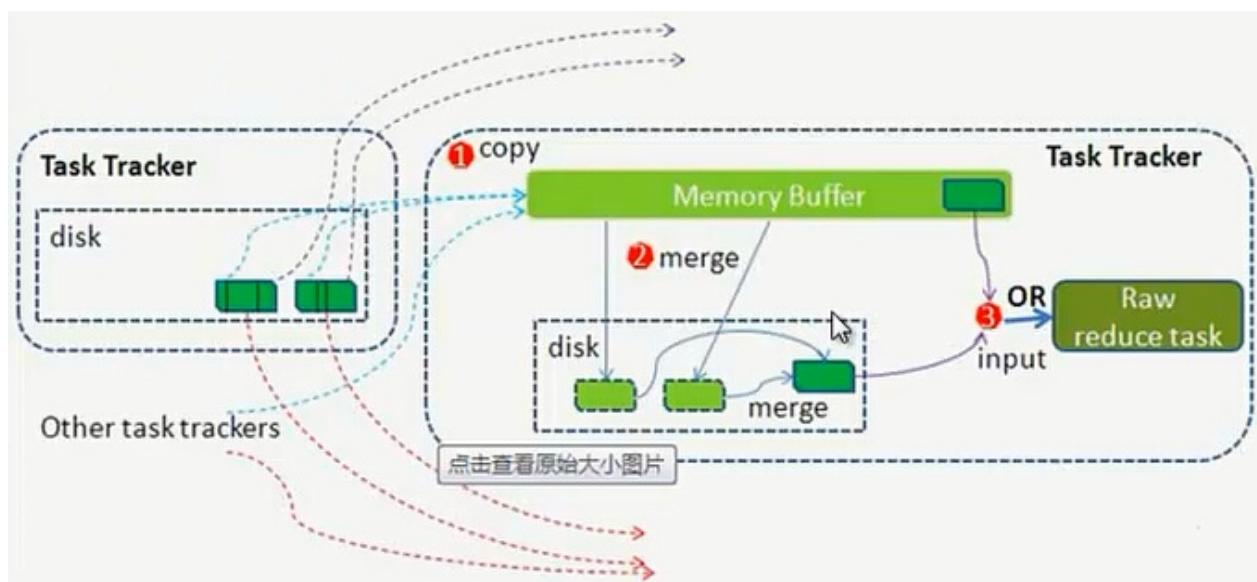
merge是将所有的溢写文件归并到一个文件，结合上面所描述的combiner的作用范围，归并得到的文件内键值对有可能拥有相同的key，这个过程如果client设置过Combiner，也会合并相同的key值的键值对，如果没有，merge得到的就是键值集合，如{"aaa", [5, 8, 2, ...]}

注意：combiner的合理设置可以提高效率，但是如果使用不当会影响效率！

1. 至此，map端的所有工作都已经结束！

## Reduce端：

当mapreduce任务提交后，reduce task就不断通过RPC从JobTracker那里获取map task是否完成的信息，如果获知某台TaskTracker上的map task执行完成，Shuffle的后半段过程就开始启动。其实呢，reduce task在执行之前的工作就是：不断地拉取当前job里每个map task的最终结果，并对不同地方拉取过来的数据不断地做merge，也最终形成一个文件作为reduce task的输入文件。



1.Copy过程，简单地拉取数据。Reduce进程启动一些数据copy线程（Fether），通过HTTP方式请求map task所在的TaskTracker获取map task的输出文件。因为map task早已结束，这些文件就归TaskTracker管理在本地磁盘。

2.Merge过程。这里的merge如map端的merge动作，只是数组中存放的是不同map端copy过来的数值。Copy过来的数据会先放入内存缓冲区中，这里缓冲区的大小要比map端的更为灵活，它是基于JVM的heap size设置，因为shuffler阶段reducer不运行，所以应该把绝大部分的内存都给shuffle用。

merge的三种形式：

内存到内存、内存到磁盘、磁盘到磁盘

默认情况下，第一种形式不启用。当内存中的数据量达到一定的阈值，就启动内存到磁盘的merge。与map端类似，这也是溢写过程，当然如果这里设置了Combiner，也是会启动的，然后在磁盘中生成了众多的溢写文件。第二种merge方式一直在运行，直到没有map端的数据时才结束，然后启动第三种磁盘到磁盘的merge方式生成最终的那个文件。

3.reducer的输入文件。不断地merge后，最后会生成一个“最终文件”。这个最终文件可能在磁盘中也可能在内存中。当然我们希望它在内存中，直接作为reducer的输入，但默认情况下，这个文件是存放于磁盘中的。当reducer的输入文件已定，整个shuffle才最终结束。然后就是reducer执行，把结果存放到HDFS上。

### 3. Yarn

# Yarn内存分配管理机制及相关参数配置

## 一、相关配置情况

关于Yarn内存分配与管理，主要涉及到了ResourceManage、ApplicationMatser、NodeManager这几个概念，相关的优化也要紧紧围绕着这几方面来开展。这里还有一个Container的概念，现在可以先把它理解为运行map/reduce task的容器，后面有详细介绍。

### 1.1 RM的内存资源配置，配置的是资源调度相关

RM1：**yarn.scheduler.minimum-allocation-mb** 分配给AM单个容器可申请的最小内存

RM2：**yarn.scheduler.maximum-allocation-mb** 分配给AM单个容器可申请的最大内存

注：

- 最小值可以计算一个节点最大Container数量
- 一旦设置，不可动态改变

### 1.2 NM的内存资源配置，配置的是硬件资源相关

NM1：**yarn.nodemanager.resource.memory-mb** 节点最大可用内存

NM2：**yarn.nodemanager.vmem-pmem-ratio** 虚拟内存率，默认2.1

注：

- RM1、RM2的值均不能大于NM1的值
- NM1可以计算节点最大Container数量， $\text{max}(\text{Container}) = \text{NM1}/\text{RM1}$
- 一旦设置，不可动态改变

### 1.3 AM内存配置相关参数，配置的是任务相关

AM1：**mapreduce.map.memory.mb** 分配给map Container的内存大小

AM2：**mapreduce.reduce.memory.mb** 分配给reduce Container的内存大小

这两个值应该在RM1和RM2这两个值之间

- AM2的值最好为AM1的两倍
- 这两个值可以在启动时改变

AM3：**mapreduce.map.java.opts** 运行map任务的jvm参数，如-Xmx，-Xms等选项

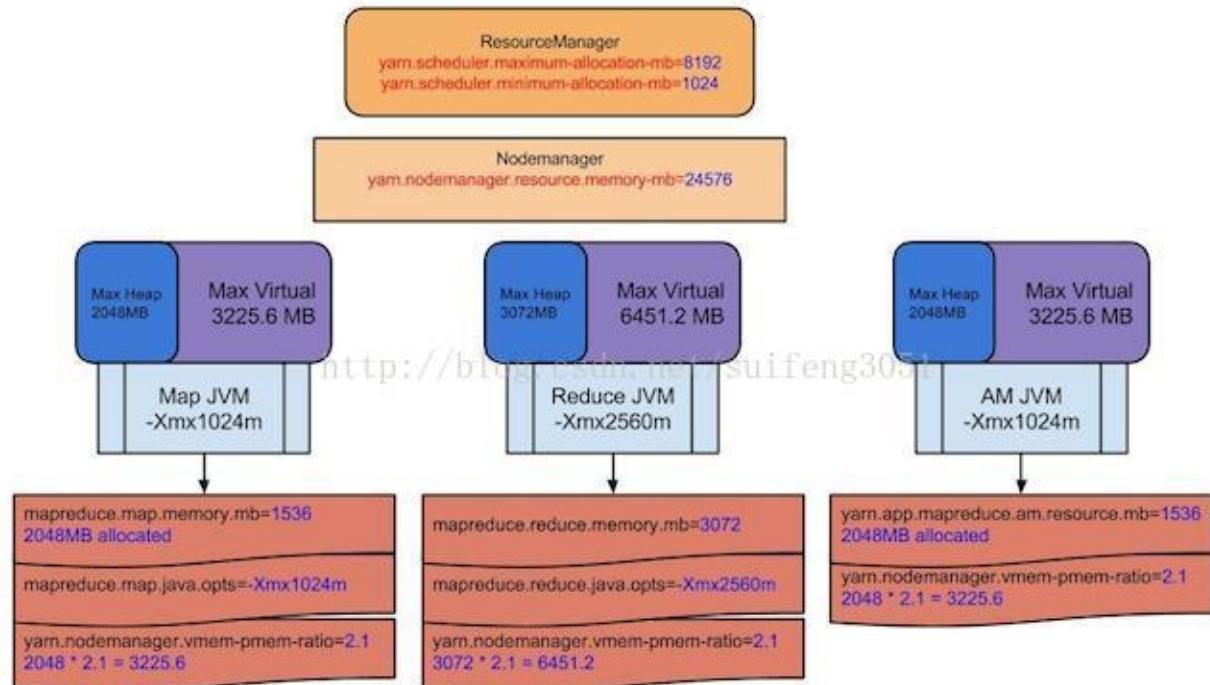
AM4：**mapreduce.reduce.java.opts** 运行reduce任务的jvm参数，如-Xmx，-Xms等选项

注：

- 这两个值应该在AM1和AM2之间

## 二、对于这些配置概念的理解

知道有这些参数，还需理解其如何分配，下面我就一副图让大家更形象的了解各个参数的含义。



如上图所示，先看最下面褐色部分，AM参数`mapreduce.map.memory.mb=1536MB`，表示AM要为map Container申请1536MB资源，但RM实际分配的内存却是2048MB，因为`yarn.scheduler.minimum-allocation-mb=1024MB`，这定义了RM最小要分配1024MB，1536MB超过了这个值，所以实际分配给AM的值为2048MB(这涉及到了规整化因子，关于规整化因子，在本文最后有介绍)。

AM参数`mapreduce.map.java.opts=-Xmx 1024m`，表示运行map任务的jvm内存为1024MB,因为map任务要运行在Container里面，所以这个参数的值略微小于`mapreduce.map.memory.mb=1536MB`这个值。

NM参数`yarn.nodemanager.vmem-pmem-radio=2.1`,这表示NodeManager可以分配给map/reduce Container 2.1倍的虚拟内存，按照上面的配置，实际分配给map Container容器的虚拟内存大小为 $2048 * 2.1 = 3225.6\text{MB}$ ，若实际用到的内存超过这个值，NM就会kill掉这个map Container,任务执行过程就会出现异常。

AM参数`mapreduce.reduce.memory.mb=3072MB`，表示分配给reduce Container的容器大小为3072MB,而map Container的大小分配的是1536MB，从这也看出，reduce Container容器的大小最好是map Container大小的两倍。

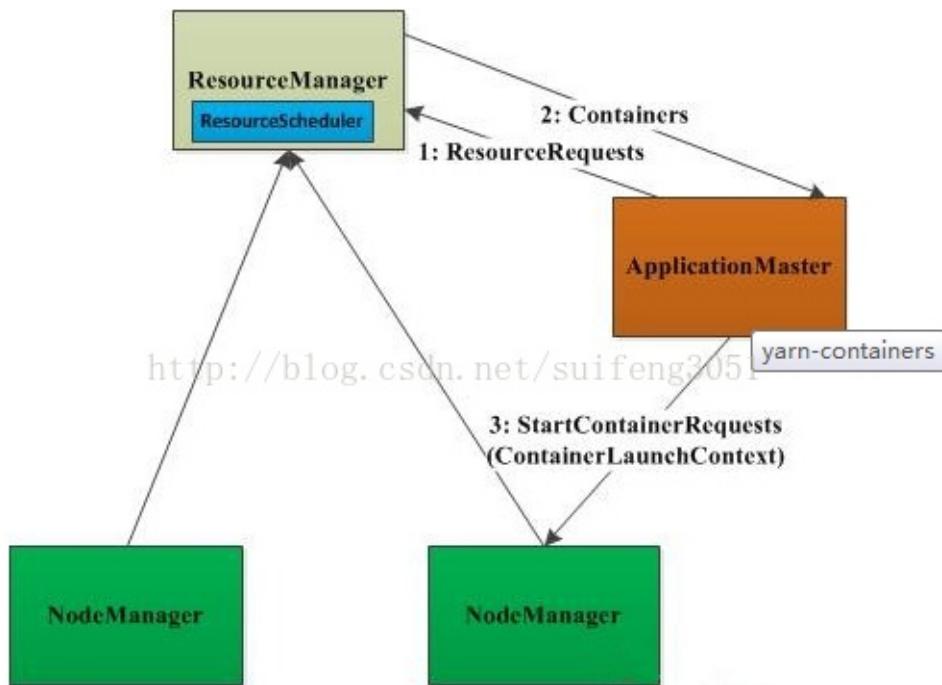
NM参数`yarn.nodemanager.resource.mem.mb=24576MB`,这个值表示节点分配给NodeManager的可用内存，也就是节点用来执行yarn任务的内存大小。这个值要根据实际服务器内存大小来配置，比如我们hadoop集群机器内存是128GB，我们可以分配其中的80%给yarn，也就是102GB。

上图中RM的两个参数分别1024MB和8192MB，分别表示分配给AM map/reduce Container的最大值和最小值。

### 三、关于任务提交过程

#### 3.1 任务提交过程

- 步骤1：用户将应用程序提交到ResourceManager上；
- 步骤2：ResourceManager为应用程序ApplicationMaster申请资源，并与某个NodeManager通信，以启动ApplicationMaster；
- 步骤3：ApplicationMaster与ResourceManager通信，为内部要执行的任务申请资源，一旦得到资源后，将于NodeManager通信，以启动对应的任務。
- 步骤4：所有任务运行完成后，ApplicationMaster向ResourceManager注销，整个应用程序运行结束。



## 3.2 关于Container

(1) Container是YARN中资源的抽象，它封装了某个节点上一定量的资源（CPU和内存两类资源）。它跟Linux Container没有任何关系，仅仅是YARN提出的一个概念（从实现上看，可看做一个可序列化/反序列化的Java类）。

(2) Container由ApplicationMaster向ResourceManager申请的，由ResouceManager中的资源调度器异步分配给ApplicationMaster；

(3) Container的运行是由ApplicationMaster向资源所在的NodeManager发起的，Container运行时需提供内部执行的任务命令（可以使任何命令，比如java、Python、C++进程启动命令均可）以及该命令执行所需的环境变量和外部资源（比如词典文件、可执行文件、jar包等）。

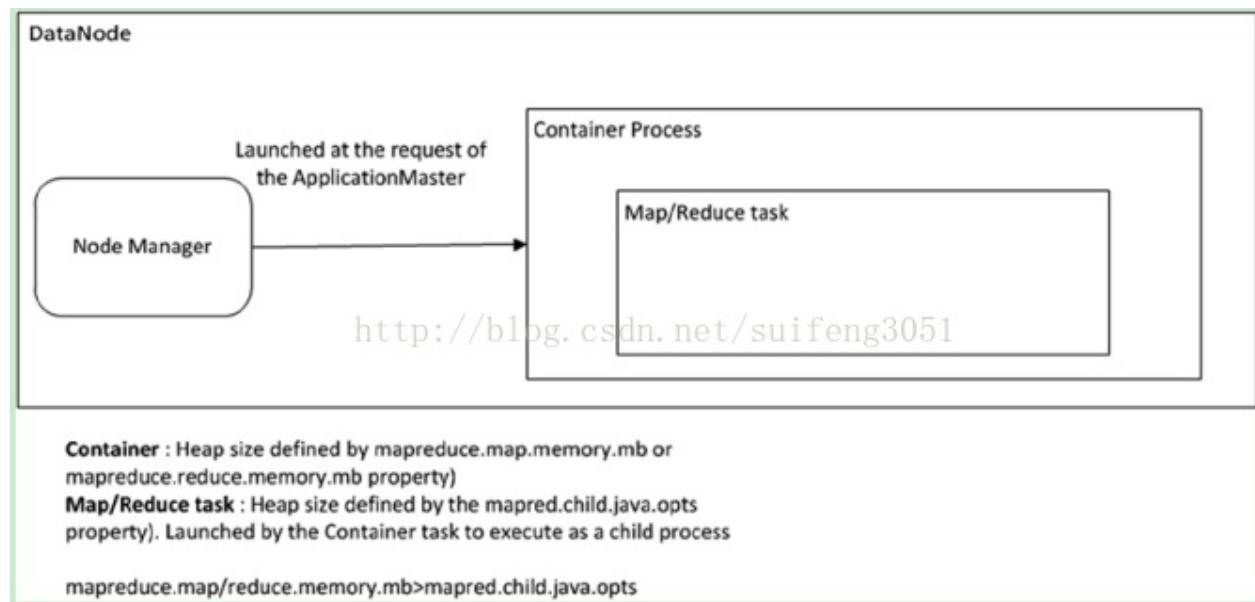
另外，一个应用程序所需的Container分为两大类，如下：

- (a) 运行ApplicationMaster的Container：这是由ResourceManager（向内部的资源调度器）申请和启动的，用户提交应用程序时，可指定唯一的ApplicationMaster所需的资源；
- (b) 运行各类任务的Container：这是由ApplicationMaster向ResourceManager申请的，并由ApplicationMaster与NodeManager通信以启动之。

以上两类Container可能在任意节点上，它们的位置通常而言是随机的，即ApplicationMaster可能与它管理的任务运行在一个节点上。

Container是YARN中最重要的概念之一，懂得该概念对于理解YARN的资源模型至关重要，望大家好好理解。

注意：如下图，map/reduce task是运行在Container之中的，所以上面提到的mapreduce.map(reduce).memory.mb大小都大于mapreduce.map(reduce).java.opts值的大小。



## 四、HDP平台参数调优建议

根据上面介绍的相关知识，我们就可以根据我们的实际情况作出相关参数的设置，当然还需要在运行测试过程中不断检验和调整。以下是hortonworks给出的配置建议：[http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.1.1/bk\\_installing\\_manually\\_book/content/rpm-chap1-11.html](http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.1.1/bk_installing_manually_book/content/rpm-chap1-11.html)

### 4.1 内存分配

Reserved Memory = Reserved for stack memory + Reserved for HBase Memory (If HBase is on the same node) 系统总内存126GB，预留给操作系统24GB，如果有Hbase再预留给Hbase24GB。下面的计算假设Datanode节点部署了Hbase。

### 4.2 containers 计算：

```
MIN_CONTAINER_SIZE = 2048 MB
containers = min (2*CORES, 1.8*DISKS, (Total available RAM) / MIN_CONTAINER_SIZE)
\# of containers = min (2*12, 1.8*12, (78 * 1024) / 2048)
\# of containers = min (24,21.6,39)
\# of containers = 22
```

container 内存计算：

```
RAM-per-container = max(MIN_CONTAINER_SIZE, (Total Available RAM) / containers))
RAM-per-container = max(2048, (78 * 1024) / 22))
RAM-per-container = 3630 MB
```

### 4.3 Yarn 和 Mapreduce 参数配置：

```

yarn.nodemanager.resource.memory-mb = containers * RAM-per-container
yarn.scheduler.minimum-allocation-mb = RAM-per-container
yarn.scheduler.maximum-allocation-mb = containers * RAM-per-container
mapreduce.map.memory.mb          = RAM-per-container
mapreduce.reduce.memory.mb       = 2 * RAM-per-container
mapreduce.map.java.opts         = 0.8 * RAM-per-container
mapreduce.reduce.java.opts      = 0.8 * 2 * RAM-per-container
yarn.nodemanager.resource.memory-mb = 22 * 3630 MB
yarn.scheduler.minimum-allocation-mb = 3630 MB
yarn.scheduler.maximum-allocation-mb = 22 * 3630 MB
mapreduce.map.memory.mb        = 3630 MB
mapreduce.reduce.memory.mb     = 22 * 3630 MB
mapreduce.map.java.opts       = 0.8 * 3630 MB
mapreduce.reduce.java.opts    = 0.8 * 2 * 3630 MB

```

## 附：规整化因子介绍

为了易于管理资源和调度资源，Hadoop YARN内置了资源规整化算法，它规定了最小可申请资源量、最大可申请资源量和资源规整化因子，如果应用程序申请的资源量小于最小可申请资源量，则YARN会将其大小改为最小可申请量，也就是说，应用程序获得资源不会小于自己申请的资源，但也不一定相等；如果应用程序申请的资源量大于最大可申请资源量，则会抛出异常，无法申请成功；规整化因子是用来规整化应用程序资源的，应用程序申请的资源如果不是该因子的整数倍，则将被修改为最小的整数倍对应的值，公式为 $\text{ceil}(a/b)*b$ ，其中a是应用程序申请的资源，b为规整化因子。

比如，在yarn-site.xml中设置，相关参数如下：

```

yarn.scheduler.minimum-allocation-mb : 最小可申请内存量，默认是1024
yarn.scheduler.minimum-allocation-vcores : 最小可申请CPU数，默认是1
yarn.scheduler.maximum-allocation-mb : 最大可申请内存量，默认是8096
yarn.scheduler.maximum-allocation-vcores : 最大可申请CPU数，默认是4

```

对于规整化因子，不同调度器不同，具体如下：

FIFO和Capacity Scheduler，规整化因子等于最小可申请资源量，不可单独配置。

Fair Scheduler：规整化因子通过参数**yarn.scheduler.increment-allocation-mb**和**yarn.scheduler.increment-allocation-vcores**设置，默认是1024和1。

通过以上介绍可知，应用程序申请到资源量可能大于资源申请的资源量，比如YARN的最小可申请资源内存量为1024，规整因子是1024，如果一个应用程序申请1500内存，则会得到2048内存，如果规整因子是512，则得到1536内存。

## Yarn调度器 Scheduler详解

理想情况下，我们应用对Yarn资源的请求应该立刻得到满足，但现实情况资源往往是有限的，特别是在一个很繁忙的集群，一个应用资源的请求经常需要等待一段时间才能的到相应的资源。在Yarn中，负责给应用分配资源的就是Scheduler。其实调度本身就是一个难题，很难找到一个完美的策略可以解决所有的应用场景。为此，Yarn提供了多种调度器和可配置的策略供我们选择。

### 一、调度器的选择

在Yarn中有三种调度器可以选择：FIFO Scheduler，Capacity Scheduler，Fair Scheduler。

FIFO Scheduler把应用按提交的顺序排成一个队列，这是一个先进先出队列，在进行资源分配的时候，先给队列中最头上的应用进行分配资源，待最头上的应用需求满足后再给下一个分配，以此类推。

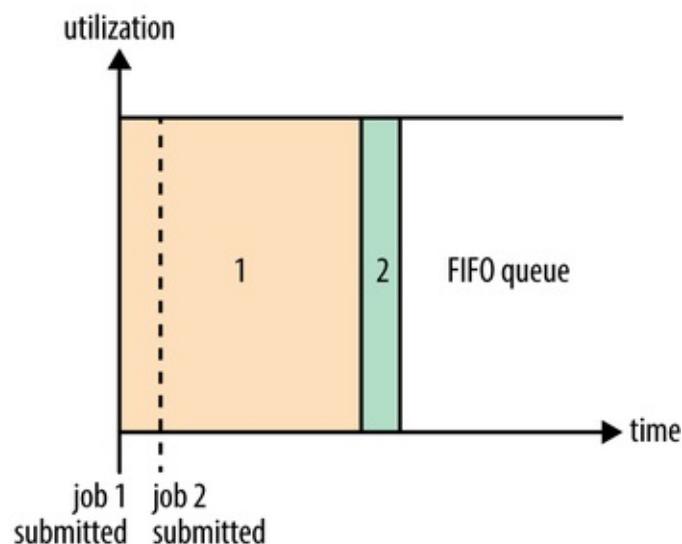
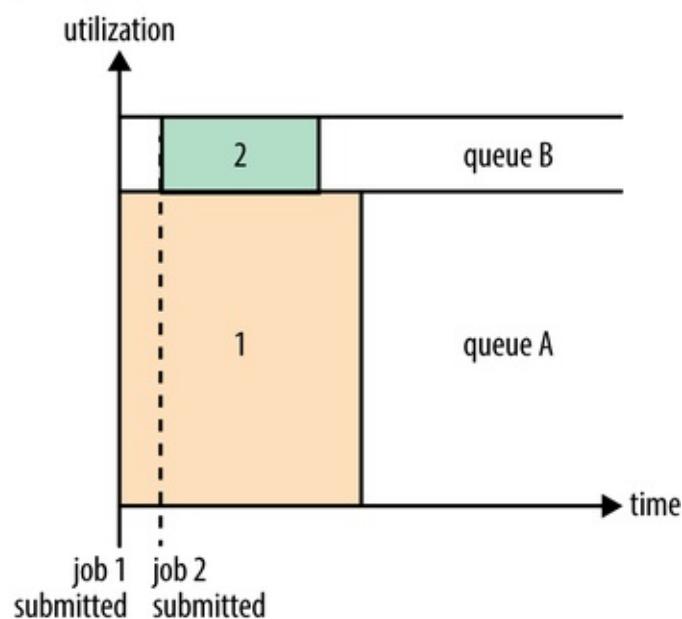
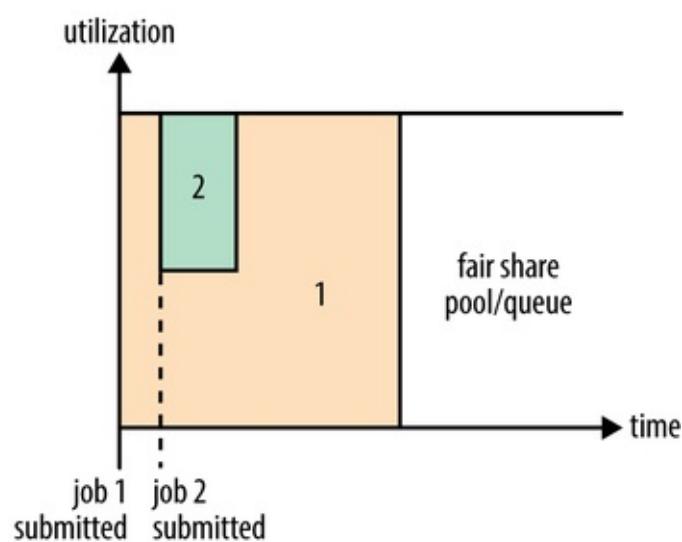
FIFO Scheduler是最简单也是最容易理解的调度器，也不需要任何配置，但它并不适用于共享集群。大的应用可能会占用所有集群资源，这就导致其它应用被阻塞。在共享集群中，更适合采用Capacity Scheduler或Fair Scheduler，这两个调度器都允许大任务和小任务在提交的同时获得一定的系统资源。

下面“Yarn调度器对比图”展示了这几个调度器的区别，从图中可以看出，在FIFO 调度器中，小任务会被大任务阻塞。

而对于Capacity调度器，有一个专门的队列用来运行小任务，但是为小任务专门设置一个队列会预先占用一定的集群资源，这就导致大任务的执行时间会落后于使用FIFO调度器时的时间。

在Fair调度器中，我们不需要预先占用一定的系统资源，Fair调度器会为所有运行的job动态的调整系统资源。如下图所示，当第一个大job提交时，只有这一个job在运行，此时它获得了所有集群资源；当第二个小任务提交后，Fair调度器会分配一半资源给这个小任务，让这两个任务公平的共享集群资源。

需要注意的是，在下图Fair调度器中，从第二个任务提交到获得资源会有一定的延迟，因为它需要等待第一个任务释放占用的Container。小任务执行完成之后也会释放自己占用的资源，大任务又获得了全部的系统资源。最终的效果就是Fair调度器即得到了高的资源利用率又能保证小任务及时完成。

**i. FIFO Scheduler****ii. Capacity Scheduler****iii. Fair Scheduler**

Yarn调度器 对比图:

## 二、Capacity Scheduler（容器调度器）的配置

### 2.1 容器调度介绍

Capacity调度器允许多个组织共享整个集群，每个组织可以获得集群的一部分计算能力。通过为每个组织分配专门的队列，然后再为每个队列分配一定的集群资源，这样整个集群就可以通过设置多个队列的方式给多个组织提供服务了。除此之外，队列内部又可以垂直划分，这样一个组织内部的多个成员就可以共享这个队列资源了，在一个队列内部，资源的调度是采用的是先进先出(FIFO)策略。

通过上面那幅图，我们已经知道一个job可能使用不了整个队列的资源。然而如果这个队列中运行多个job，如果这个队列的资源够用，那么就分配给这些job，如果这个队列的资源不够用了呢？其实Capacity调度器仍可能分配额外的资源给这个队列，这就是“弹性队列”(queue elasticity)的概念。

在正常的操作中，Capacity调度器不会强制释放Container，当一个队列资源不够用时，这个队列只能获得其它队列释放后的Container资源。当然，我们可以为队列设置一个最大资源使用量，以免这个队列过多的占用空闲资源，导致其它队列无法使用这些空闲资源，这就是“弹性队列”需要权衡的地方。

### 2.2 容器调度的配置

假设我们有如下层次的队列：

```

root
└── prod
└── dev
    └── eng
    └── science

```

下面是一个简单的Capacity调度器的配置文件，文件名为capacity-scheduler.xml。在这个配置中，在root队列下面定义了两个子队列prod和dev，分别占40%和60%的容量。需要注意，一个队列的配置是通过属性yarn.scheduler.capacity..指定的，代表的是队列的继承树，如root.prod队列，一般指capacity和maximum-capacity。

我们可以看到，dev队列又被分成了eng和science两个相同容量的子队列。dev的maximum-capacity属性被设置成了75%，所以即使prod队列完全空闲dev也不会占用全部集群资源，也就是说，prod队列仍有25%的可用资源用来应急。我们注意到，eng和science两个队列没有设置maximum-capacity属性，也就是说eng或science队列中的job可能会用到整个dev队列的所有资源（最多为集群的75%）。而类似的，prod由于没有设置maximum-capacity属性，它有可能会占用集群全部资源。

Capacity容器除了可以配置队列及其容量外，我们还可以配置一个用户或应用可以分配的最大资源数量、可以同时运行多少应用、队列的ACL认证等。

### 2.3 队列的设置

关于队列的设置，这取决于我们具体的应用。比如，在MapReduce中，我们可以通过mapreduce.job.queuename属性指定要用的队列。如果队列不存在，我们在提交任务时就会收到错误。如果我们没有定义任何队列，所有的应用将会放在一个default队列中。

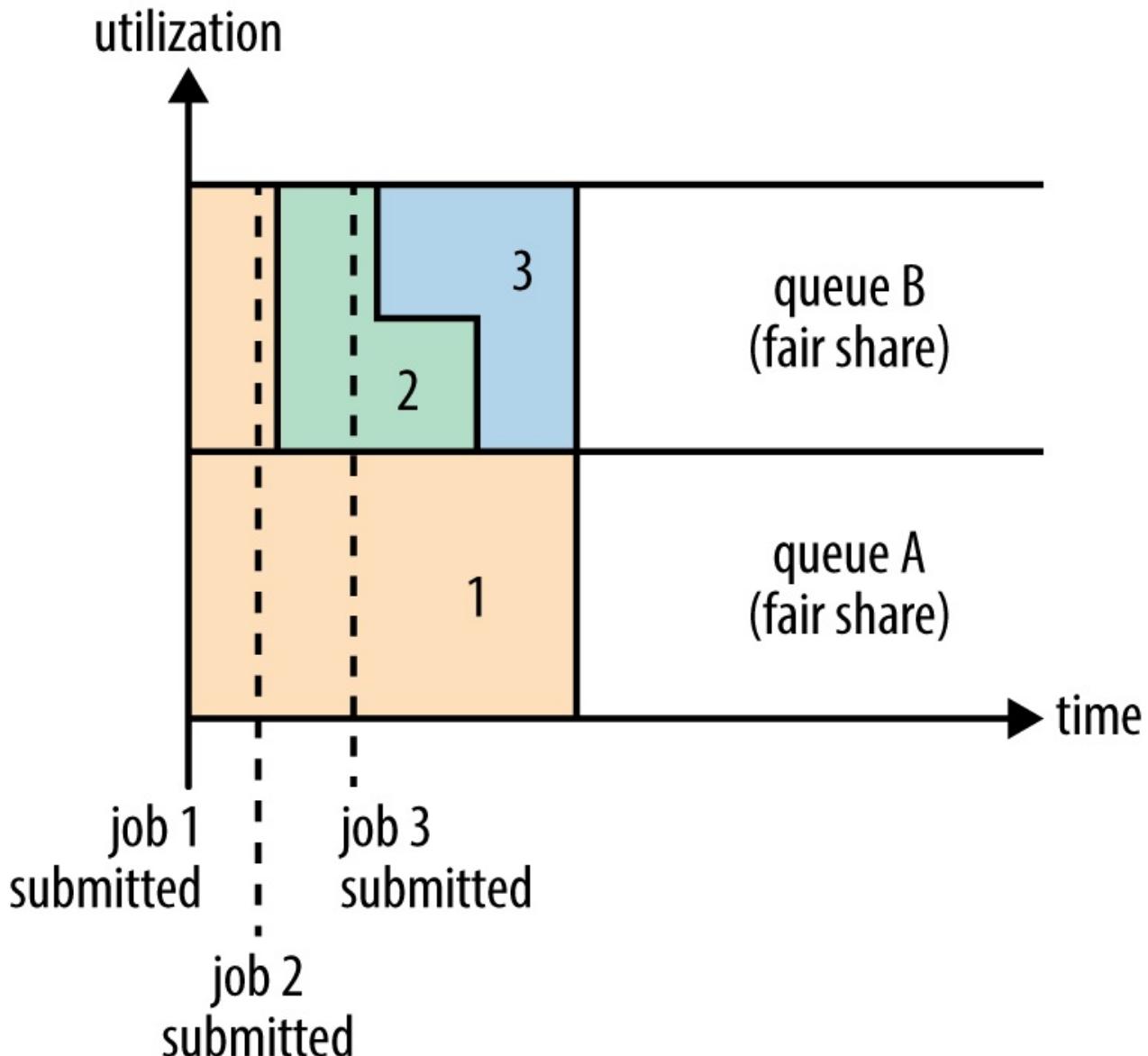
注意：对于Capacity调度器，我们的队列名必须是队列树中的最后一部分，如果我们使用队列树则不会被识别。比如，在上面配置中，我们使用prod和eng作为队列名是可以的，但是如果我们用root.dev.eng或者dev.eng是无效的。

## 三、Fair Scheduler（公平调度器）的配置

### 3.1 公平调度

Fair调度器的设计目标是为所有的应用分配公平的资源（对公平的定义可以通过参数来设置）。在上面的“Yarn调度器对比图”展示了一个队列中两个应用的公平调度；当然，公平调度也可以在多个队列间工作。举个例子，假设有两个用户A和B，他们分别拥有一个队列。当A启动一个job而B没有任务时，A会获得全部集群资源；当B启动一个job后，A的job会继续运

行，不过一会儿之后两个任务会各自获得一半的集群资源。如果此时B再启动第二个job并且其它job还在运行，则它将会和B的第一个job共享B这个队列的资源，也就是B的两个job会用于四分之一的集群资源，而A的job仍然用于集群一半的资源，结果就是资源最终在两个用户之间平等的共享。过程如下图所示：



### 3.2 启用 Fair Scheduler

调度器的使用是通过yarn-site.xml配置文件中的yarn.resourcemanager.scheduler.class参数进行配置的，默认采用Capacity Scheduler调度器。如果我们要使用Fair调度器，需要在这个参数上配置FairScheduler类的全限定名：  
org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler。

### 3.3 队列的配置

Fair调度器的配置文件位于类路径下的fair-scheduler.xml文件中，这个路径可以通过yarn.scheduler.fair.allocation.file属性进行修改。若没有这个配置文件，Fair调度器采用的分配策略，这个策略和3.1节介绍的类似：调度器会在用户提交第一个应用时为其自动创建一个队列，队列的名字就是用户名，所有的应用都会被分配到相应的用户队列中。

我们可以在配置文件中配置每一个队列，并且可以像Capacity 调度器一样分层次配置队列。比如，参考capacity-scheduler.xml来配置fair-scheduler：

```

<?xml version="1.0"?>
<allocations>
  <defaultQueueSchedulingPolicy>fair</defaultQueueSchedulingPolicy>

  <queue name="prod">
    <weight>40</weight>
    <schedulingPolicy>fifo</schedulingPolicy>
  </queue>

  <queue name="dev">
    <weight>60</weight>
    <queue name="eng" />
    <queue name="science" />
  </queue>

  <queuePlacementPolicy>
    <rule name="specified" create="false" />
    <rule name="primaryGroup" create="false" />
    <rule name="default" queue="dev.eng" />
  </queuePlacementPolicy>
</allocations>

```

队列的层次是通过嵌套元素实现的。所有的队列都是root队列的孩子，即使我们没有配到元素里。在这个配置中，我们把dev队列有分成了eng和science两个队列。

Fair调度器中的队列有一个权重属性（这个权重就是对公平的定义），并把这个属性作为公平调度的依据。在这个例子中，当调度器分配集群40:60资源给prod和dev时便视作公平，eng和science队列没有定义权重，则会被平均分配。这里的权重并不是百分比，我们把上面的40和60分别替换成2和3，效果也是一样的。注意，对于在没有配置文件时按用户自动创建的队列，它们仍有权重并且权重值为1。

每个队列内部仍可以有不同的调度策略。队列的默认调度策略可以通过顶级元素进行配置，如果没有配置，默认采用公平调度。

尽管是Fair调度器，其仍支持在队列级别进行FIFO调度。每个队列的调度策略可以被其内部的元素覆盖，在上面这个例子中，prod队列就被指定采用FIFO进行调度，所以，对于提交到prod队列的任务就可以按照FIFO规则顺序的执行了。需要注意，prod和dev之间的调度仍然是公平调度，同样eng和science也是公平调度。

尽管上面的配置中没有展示，每个队列仍可配置最大、最小资源占用数和最大可运行的应用的数量。

## 3.4 队列的设置

Fair调度器采用了一套基于规则的系统来确定应用应该放到哪个队列。在上面的例子中，元素定义了一个规则列表，其中的每个规则会被逐个尝试直到匹配成功。例如，上例第一个规则specified，则会把应用放到它指定的队列中，若这个应用没有指定队列名或队列名不存在，则说明不匹配这个规则，然后尝试下一个规则。primaryGroup规则会尝试把应用放在以用户所在的Unix组名命名的队列中，如果没有这个队列，不创建队列转而尝试下一个规则。当前面所有规则不满足时，则触发default规则，把应用放在dev.eng队列中。

当然，我们可以不配置queuePlacementPolicy规则，调度器则默认采用如下规则：

```

<queuePlacementPolicy>
<rule name="specified" />
<rule name="user" />
</queuePlacementPolicy>

```

上面规则可以归结成一句话，除非队列被准确的定义，否则会以用户名为队列名创建队列。

还有一个简单的配置策略可以使得所有的应用放入同一个队列（default），这样就可以让所有应用之间平等共享集群而不是在用户之间。这个配置的定义如下：

```
<queuePlacementPolicy>
<rule name="default" />
</queuePlacementPolicy>
```

实现上面功能我们还可以不使用配置文件，直接设置yarn.scheduler.fair.user-as-default-queue=false，这样应用便会被放入default队列，而不是各个用户名队列。另外，我们还可以设置yarn.scheduler.fair.allow-undeclared-pools=false，这样用户就无法创建队列了。

### 3.5 抢占（Preemption）

当一个job提交到一个繁忙集群中的空队列时，job并不会马上执行，而是阻塞直到正在运行的job释放系统资源。为了使提交job的执行时间更具预测性（可以设置等待的超时时间），Fair调度器支持抢占。

抢占就是允许调度器杀掉占用超过其应占份额资源队列的containers，这些containers资源便可被分配到应该享有这些份额资源的队列中。需要注意抢占会降低集群的执行效率，因为被终止的containers需要被重新执行。

可以通过设置一个全局的参数yarn.scheduler.fair.preemption=true来启用抢占功能。此外，还有两个参数用来控制抢占的过期时间（这两个参数默认没有配置，需要至少配置一个来允许抢占Container）：

```
- minimum share preemption timeout
- fair share preemption timeout
```

如果队列在minimum share preemption timeout指定的时间内未获得最小的资源保障，调度器就会抢占containers。我们可以通过配置文件中的顶级元素为所有队列配置这个超时时间；我们还可以在元素内配置元素来为某个队列指定超时时间。

与之类似，如果队列在fair share preemption timeout指定时间内未获得平等的资源的一半（这个比例可以配置），调度器则会进行抢占containers。这个超时时间可以通过顶级元素和元素级元素分别配置所有队列和某个队列的超时时间。上面提到的比例可以通过(配置所有队列)和(配置某个队列)进行配置，默认是0.5。

## 4. Hadoop

## GP、Hadoop、Elk的原理区别

### 存储模型

hadoop是hdfs，扩展是通过元数据来做的，中心节点用来存元数据，在加入新的节点的时候，只需要修改元数据就可以了，所以hdfs的扩展能力是受到管理元数据那台机器的性能限制的。

mpp通常采用的是没有中心节点的存储模型，比如hash，你每次增加节点的时候，都需要rehash，这样当规模到了几百台的时候，扩展能力就下来了

### 内存管理方式

mpp内存管理比较精细，他主要的想法是在每个机器上放个数据库，传统数据库的内存管理比较复杂，主要是内外存交互的东西，这样的架构决定了mpp在小数据量的时候，延迟可以做的比较小，但是在大数据量的时候，吞吐量做不上去。

hive的内存管理非常粗放，他后来就是mapreduce的job，mr的job是没有太多精细的内存管理的，他就是拼了命地scan，完了顶多就是个spill，这样的架构导致throughput很大，但是latency很高，当你集群规模很大的时候，你一般会追求很大的throughput。当数据量很大的时候，如果你用mpp那种传统的内存管理的话，大批量的计算反而会慢，而且更加占资源。

### 事务

hive不支持传统意义上的那种高并发的事务

一旦你要上分布式事务，基本上你的可扩展性就上不去了

### failover机制，

hive的failover就是mr的failover，job挂掉了重新换机器跑就完了，但是mpp如果采用传统架构的话，他的计算是要attach到数据节点上去的，如果你规模上去，那么fail的可能性就上去了，这样如果你每次计算都有台机器挂了，你一挂，别人就要等你，而不是换台机器继续跑，那么这个也限制了可扩展性，当然，如果mpp在底层用了统一的存储，完了计算也可以到处转移，再想个办法把中间状态记录下来，也可以扩展（这个实际上就是sparksql）

---

### 扩展性

MPP DB 还是基于原 DB 扩展而来，DB 里面天然追求一致性（Consistency），必然带来分区容错性较差。集群规模变得太大，业务数据太多时，MPP DB 的元数据管理就完全是一个灾难。元数据巨大无比，一旦出错很难恢复，动不动导致毁库。

所以 MPP DB 要在扩展性上有质的提升，要对元数据，以及数据存储有架构上的突破，降低对一致性的要求，这样扩展性才能提升，否则的话很难相信一个 MPP DB 数据库是可以容易扩展的

### 并发

一个查询系统，设计出来就是提供人用的，所以能支持的同时并发越高越好。MPP DB 核心原理是一个大的查询通过分析为一个个子查询，分布到底层的执行，最后再合并结果，说白了就是通过多线程并发来暴力 SCAN 来实现高速。这种暴力 SCAN 的方法，对单个查询来说，动用了整个系统的能力，单个查询比较快，但同时带来用力过猛的问题，整个系统能支持的并发必然不高，从目前实际使用的经验来说，也就支持50~100的并发能力。

---

hadoop 和 mpp 的本质区别是：就是什么时候解决 data locality 的问题 hadoop 的思路是每次计算的时候解决，mpp的思路是加载的时候解决。

从查询引擎看，由于数据库支持索引，查询性能应该优于HADOOP。但是对于PB级别的数据，无法给所有维度的查询建立索引，主要靠全表扫描。因此对于复杂查询，MPP并不比HADOOP特别是现在的SPARK方案体现出优势，而且架不住Hadoop集群机器多。

## 5.1 安全集群模式下，Spark跨集群连接HBase

背景：本端集群的Spark组件需要操作对端集群的HBase组件。Spark程序运行模式为yarn-cluster模式。

前置条件：两个集群已经互信。集群互信的基本条件是，本端集群和对端集群的版本必须完全一致，并且都为安全模式。

跨集群连接HBase的步骤如下：

1. 修改spark客户端的conf目录下的**spark-defaults.conf**，确保

```
spark.hbase.obtainToken.enabled = true
spark.inputFormat.cache.enabled = false
```

这是为了开启Spark on HBase特性，安全模式下必须开启此特性，Spark中才能认证HBase，否则会报出GSSEException：

```
javax.security.sasl.SaslException: GSS initiate failed [Caused by GSSEException: No valid credentials provided (Mechanism level: Failed to find any Kerberos tgt)]
```

通过实际投产情况，发现跨集群连接HBase的情况下，必须是通过修改**spark-defaults.conf**文件来开启Spark on HBase才有效果；如果不修改这个文件，用别的手段去修改参数（比如程序代码中直接指定配置项的值），虽然Spark的Environment信息中显示为开启，但是实际还是无法连接，仍然报出GSSEException。

2. 继续修改**spark-defaults.conf**文件，将**spark.yarn.cluster.driver.extraClassPath**参数值，加入\$PWD作为classpath的一部分，并保证\$PWD在首位。例如，修改前为：

```
spark.yarn.cluster.driver.extraClassPath = /opt/huawei/Bigdata/FusionInsight/spark/cfg:/opt/huawei/Bigdata/FusionInsight/spark/spark/lib/*
```

修改后为：

```
spark.yarn.cluster.driver.extraClassPath = $PWD:/opt/huawei/Bigdata/FusionInsight/spark/cfg:/opt/huawei/Bigdata/FusionInsight/spark/spark/lib/*
```

如果不加入这一步，跨集群情况下，当前FusionInsight版本会存在无法刷新HBase Token导致token过期的问题。因为yarn内部启动container的shell脚本中，定义的classpath的顺序会导致刷新token时，优先读取节点上的配置文件，而不是Spark客户端提交的，导致刷新HBase Token时连接的zookeeper服务不是HBase所在集群的，而变成本集群的了，最终导致无法刷新HBase Token而连接过期。

3. 将spark客户端的conf目录下的**hbase-site.xml**，替换为对端集群的**hbase-site.xml**文件。

4. 在**spark-submit**提交时，通过--files参数将对端集群**hbase-site.xml**文件发送到每个Executor

```
$SPARK_HOME/bin/spark-submit \
--master yarn \
--deploy-mode cluster \
--driver-memory 2G \
--num-executors 5 \
--principal logc \
--keytab /logcAPP/SparkApp/SparkStreamingApp/conf/logc.keytab \
--jars $SPARK_HOME/lib/streamingClient/kafka-clients-0.8.2.1.jar,$SPARK_HOME/lib/streamingClient/kafka_2.10-0.8.2.1.jar,$SPARK_HOME/lib/streamingClient/spark-streaming-kafka_2.10-1.5.1.jar,/logcAPP/SparkApp/SparkStreamingApp/lib/commons-dbcpc2-2.1.1.jar,/logcAPP/SparkApp/SparkStreamingApp/lib/commons-pool2-2.4.2.jar,/logcAPP/SparkApp/SparkStreamingApp/lib/ojdbc7.jar,/logcAPP/SparkApp/SparkStreamingApp/lib/fastjson-1.2.31.jar \
--files /logcAPP/SparkApp/SparkStreamingApp/conf/dbconfig.properties,/logcAPP/SparkApp/SparkStreamingApp/conf/BehaviorTraceInfo.properties,/logcAPP/SparkApp/SparkStreamingApp/conf/hbase-site.xml \
--name BehaviorTraceInfo_New \
--class com.berchina.iec.spark.streaming.BehaviorTraceInfo_New \
/logcAPP/SparkApp/SparkStreamingApp/BehaviorTraceInfo.jar cluster 300
```

5. 程序代码中，在创建**HBase**连接时，将--files参数发送过来的对端**hbase-site.xml**文件作为连接配置，再创建连接：

```
public class HBaseTableFactory {
    private static Configuration conf = null;
    private static Connection conn = null;

    private Table hTable;

    public Table getTable() {
        return hTable;
    }

    static {
        init();
    }

    private static void init() {
        conf = HBaseConfiguration.create();

        conf.addResource(new Path(System.getProperty("user.dir") + File.separator + "hbase-site.xml"));
        System.err.println("配置读取测试：" + conf.get("hbase.zookeeper.quorum"));

        try {
            conn = ConnectionFactory.createConnection(conf);
        } catch (IOException e) {
            System.err.println("ConnectionFactory.createConnection错误：" + e.getMessage());
        }
    }

    public HBaseTableFactory(String tableName) {
        try {
            hTable = conn.getTable(TableName.valueOf(tableName));
        } catch (IOException e) {
            System.err.println("conn.getTable(" + tableName + ")错误：" + e.getMessage());
        }
    }
}
```

注意，yarn-cluster模式下，无需在代码中编写安全认证相关代码。但是如果是长时间运行的Spark Streaming应用，会存在认证过期问题，需要指定principal和keytab参数，使得yarn能去定期认证。关于这一点，在后面的小节中会详细介绍。

## 6.1 Paxos算法

### 背景

在计算机通信理论中，有一个著名的两军问题(two-army problem)，讲述通信的双方通过ACK来达成共识，永远会有一个在途的ACK需要进行确认，因此无法达成共识。

两军问题和Basic Paxos非常相似：

- 1) 通信的各方需要达成共识；
- 2) 通信的各方仅需要达成一个共识；
- 3) 假设的前提是信道不稳定，有丢包、延迟或者重放，但消息不会被篡改。

Basic Paxos最早以希腊议会的背景来讲解，但普通人不理解希腊议会的运作模式，因此看Basic Paxos的论文会比较难理解。

两军问题的背景大家更熟悉，因此尝试用这个背景来演绎一下Basic Paxos。

为了配合Basic Paxos的多数派概念，把两军改为3军；同时假设了将军和参谋的角色。

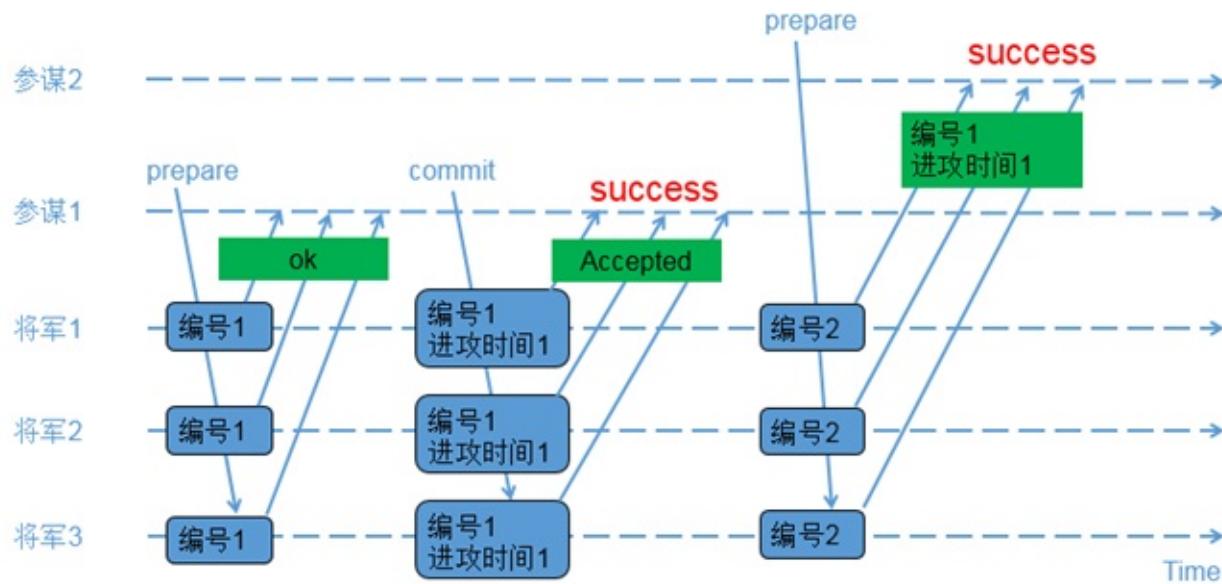
### 假设的3军问题

- 1) 1支红军在山谷里扎营，在周围的山坡上驻扎着3支蓝军；
- 2) 红军比任意1支蓝军都要强大；如果1支蓝军单独作战，红军胜；如果2支或以上蓝军同时进攻，蓝军胜；
- 3) 三支蓝军需要同步他们的进攻时间；但他们唯一的通信媒介是派通信兵步行进入山谷，在那里他们可能被俘虏，从而将信息丢失；或者为了避免被俘虏，可能在山谷停留很长时间；
- 4) 每支军队有1个参谋负责提议进攻时间；每支军队也有1个将军批准参谋提出的进攻时间；很明显，1个参谋提出的进攻时间需要获得至少2个将军的批准才有意义；
- 5) 问题：是否存在一个协议，能够使得蓝军同步他们的进攻时间？

接下来以两个假设的场景来演绎BasicPaxos；参谋和将军需要遵循一些基本的规则：

- 1) 参谋以两阶段提交（prepare/commit）的方式来发起提议，在prepare阶段需要给出一个编号；
- 2) 在prepare阶段产生冲突，将军以编号大小来裁决，编号大的参谋胜出；
- 3) 参谋在prepare阶段如果收到了将军返回的已接受进攻时间，在commit阶段必须使用这个返回的进攻时间；

### 两个参谋先后提议的场景



### 两个参謀交叉提议的场景



- 1) 参謀1发起提议，派通信兵带信给3个将军，内容为 (编号1)；
- 2) 3个将军的情况如下：

- a) 将军1和将军2收到参谋1的提议，将军1和将军2把（编号1）记录下来，如果有其他参谋提出更小的编号，将被拒绝；同时让通信兵带信回去，内容为（ok）；
- b) 负责通知将军3的通信兵被抓，因此将军3没收到参谋1的提议；
- 3) 参谋2在同一时间也发起了提议，派通信兵带信给3个将军，内容为（编号2）；
- 4) 3个将军的情况如下：
  - a) 将军2和将军3收到参谋2的提议，将军2和将军3把（编号2）记录下来，如果有其他参谋提出更小的编号，将被拒绝；同时让通信兵带信回去，内容为（ok）；
  - b) 负责通知将军1的通信兵被抓，因此将军1没收到参谋2的提议；
- 5) 参谋1收到至少2个将军的回复，再次派通信兵带信给有答复的2个将军，内容为（编号1，进攻时间1）；
- 6) 2个将军的情况如下：
  - a) 将军1收到了（编号1，进攻时间1），和自己保存的编号相同，因此把（编号1，进攻时间1）保存下来；同时让通信兵带信回去，内容为（Accepted）；
  - b) 将军2收到了（编号1，进攻时间1），由于（编号1）小于已经保存的（编号2），因此让通信兵带信回去，内容为（Rejected，编号2）；
- 7) 参谋2收到至少2个将军的回复，再次派通信兵带信给有答复的2个将军，内容为（编号2，进攻时间2）；
- 8) 将军2和将军3收到了（编号2，进攻时间2），和自己保存的编号相同，因此把（编号2，进攻时间2）保存下来，同时让通信兵带信回去，内容为（Accepted）；
- 9) 参谋2收到至少2个将军的（Accepted）内容，确认进攻时间已经被多数派接受；
- 10) 参谋1只收到了1个将军的（Accepted）内容，同时收到一个（Rejected，编号2）；参谋1重新发起提议，派通信兵带信给3个将军，内容为（编号3）；
- 11) 3个将军的情况如下：
  - a) 将军1收到参谋1的提议，由于（编号3）大于之前保存的（编号1），因此把（编号3）保存下来；由于将军1已经接受参谋1前一次的提议，因此让通信兵带信回去，内容为（编号1，进攻时间1）；
  - b) 将军2收到参谋1的提议，由于（编号3）大于之前保存的（编号2），因此把（编号3）保存下来；由于将军2已经接受参谋2的提议，因此让通信兵带信回去，内容为（编号2，进攻时间2）；
  - c) 负责通知将军3的通信兵被抓，因此将军3没收到参谋1的提议；
- 12) 参谋1收到了至少2个将军的回复，比较两个回复的编号大小，选择大编号对应的进攻时间作为最新的提议；参谋1再次派通信兵带信给有答复的2个将军，内容为（编号3，进攻时间2）；
- 13) 将军1和将军2收到了（编号3，进攻时间2），和自己保存的编号相同，因此保存（编号3，进攻时间2），同时让通信兵带信回去，内容为（Accepted）；
- 14) 参谋1收到了至少2个将军的（accepted）内容，确认进攻时间已经被多数派接受；

## 小结

BasicPaxos算法难理解，除了讲故事的背景不熟悉之外，还有以下几点：

- 1) 参与的各方并不是要针锋相对，拼个你死我活；而是要合作共赢，最终达成一个共识；当大家讲起投票的时候，往往第一反应是要针锋相对，没想到是要合作共赢；很明显可以想到，在第二个场景下，如果参谋1为了逞英雄，强行要提交他提出的进攻时间1，那么最终是无法达成一个共识的；这里的点就在于参谋1违反了规则，相当于产生了拜占庭错误；
- 2) 常规的通信协议设计，对于写操作，通常都是只返回成功和失败的状态，不会返回更多的东西；但BasicPaxos的prepare和commit，将军除了返回成功还是失败的状态之外，还会把之前已经发生的一些状态带回给参谋，这个和常规的通信协议是不同的；

- 3) 在两军问题的背景下，其实知道进攻时间被至少2个将军接受的是参谋，而不是将军；在“两个参谋交叉提议的场景”下，当参谋1没有做第2次prepare之前，将军1记录的其实是一个错误的进攻时间；理论上来说，任何一个将军在任何一个时刻都无法判断自己不是处在将军1的场景下；因此BasicPaxos在3个蓝军组成的系统中达成了一个共识，但并没有为每个将军明确了共识；
- 4) 本文的两个场景都以“两个参谋”来讲，这里的“两个参谋”可能是真的两个不同的参谋，也可能是同一个参谋因为某种原因先后做了多次提议；对应分布式系统的场景
  - a) 真的有两个并发的client
  - b) 两个client一先一后；第一个client执行到某个步骤因为某种原因停止了；过了一段时间，另外一个client接着操作同一个数据
  - c) 同一个client重试；第一次执行到某一步骤因为某种原因停止了，立即或者稍后进行了重试

结束

---

结束