## big\_day06

Hadoop01

## 大数据的概念

什么是大数据

大数据(big data,mega data)，或称巨量资料。比如TB、PB级别的数据。

目前对于大数据特点的总结是5V特点：

##### Volume(大量)

为了更准确地理解人们现在面临的数据量大小，再来看一组公式：

1024GB=1TB

1024TB=1PB

1024PB=1EB

1024 EB=1ZB

1024ZB=1YB

与海量的数据同时存在的还有越来越快的数据增长速度。根据IDC的统计，全球每年产生的数据达到将近8ZB。

##### Velocity(高速)，持续的高速增长

1. Variety(多样) ，网络日志、视频、图片、地理位置信息
2. Value(价值密度低)，以视频为例，连续不间断监控过程中，可能有用的数据仅仅有一两秒
3. Veracity（真实性）

国内Hadoop的应用情况(见图片Hadoop应用)

百度:数据挖掘与分析 日志分析平台 数据仓库系统 推荐引擎系统 用户行为分析系统

淘宝的五层架构:数据层-计算层-存储层-查询层-应用层

支付宝使用Hadoop 构建了两层数据平台:支撑技术-系统组件

腾讯用Hadoop构建了(台风系统)

中国移动

华为:Hadoop主要做出贡献的公司之一

大数据的价值

第一、计算机科学在大数据出现之前，非常依赖模型以及算法。当数据越来越大时，数据本身（而不是研究数据所使用的算法和模型）保证了数据分析结果的有效性。即便缺乏精准的算法，只要拥有足够多的数据，也能得到接近事实的结论。数据因此而被誉为新的生产力。

##### 当数据达到一定程度时，数据本身就可以说话了。

第二、当数据足够多的时候，不需要了解具体的因果关系就能够得出结论。

第三、由于能够处理多种数据结构，大数据能够在最大程度上利用互联网上记录的人类行为数据进行分析。

第四、从政府或社会角度，大数据时代到来，会催生很多新的就业岗位

所以大数据是一种新的生成力。

大数据是人工智能的基础

大数据是物理网的基础

大数据和区块链

# Hadoop概述



**官方网址：**<http://hadoop.apache.org/>

Apache的Hadoop是一个开源的、可靠的、可扩展的系统架构，可**利用分布式架构来存储海量数据**，以及**实 现分布式的计算**。

Hadoop的两个作用：①存储海量数据 ②计算海量数据

Hadoop允许使用简单的编程模型在计算机集群中对大型数据集进行分布式处理。可以从单个服务器扩展到数 千台机器，每个机器都提供本地计算和存储，而不是依靠硬件来提供高可用性。

此外，Hadoop集群的高可用性也非常良好，因为框架内的机制是可以够自动检测和处理故障。

Hadoop的创始人 Doug Cutting 与 Mike Cafarella

1985年毕业于美国斯坦福大学，他是 Lucene的创始人，同时也是Nutch和Hadoop的联合创立者，先是参与了Lucene的开发，然后是Nutch，然后是Hadoop。

hadoop名字的来源：这个名字不是一个缩写，它是一个虚构的名字。该项目的创建者，Doug Cutting如此解释Hadoop的得名："这个名字是我孩子给一头吃饱了的棕黄色大象命名的。我的命名标准就是简短，容易发音

和拼写，没有太多的意义，并且不会被用于别处。小孩子是这方面的高手。

Hadoop演变

Hadoop起源于Apache Nutch，Nutch是一个开源的java语言来实现的搜索引擎。

2004年，Cutting和同为程序员出身的Mike Cafarella决定开发一款可以代替当时的主流搜索产品的开源搜索引擎，这个项目被命名为Nutch。它是模仿google搜索引擎创立的开源搜索引擎，后归于apache旗下。

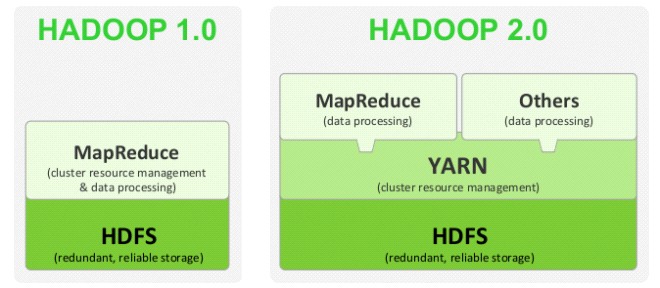
nutch主要完成抓取，提取内容等工作。和solr一样，都是搜索引擎，并且都是基于lucene的。支持全文搜索 和爬虫。

Hadoop版本说明

第一代Hadoop和第二代Hadoop

第一代Hadoop称为Hadoop1.0,分别是0.20.x,0.21.x和0.22.x 第二代Hadoop称为Hadoop2.0,分别是0.23.x和2.x。

2.0版本完全不同于Hadoop 1.0，是一套全新的架构，加入了Yarn资源协调管理框架。

Hadoop1.0和Hadoop2.0简要架构图

Yarn 资源调度框架——>实现对资源的细粒度封装（cpu，内存，带宽）

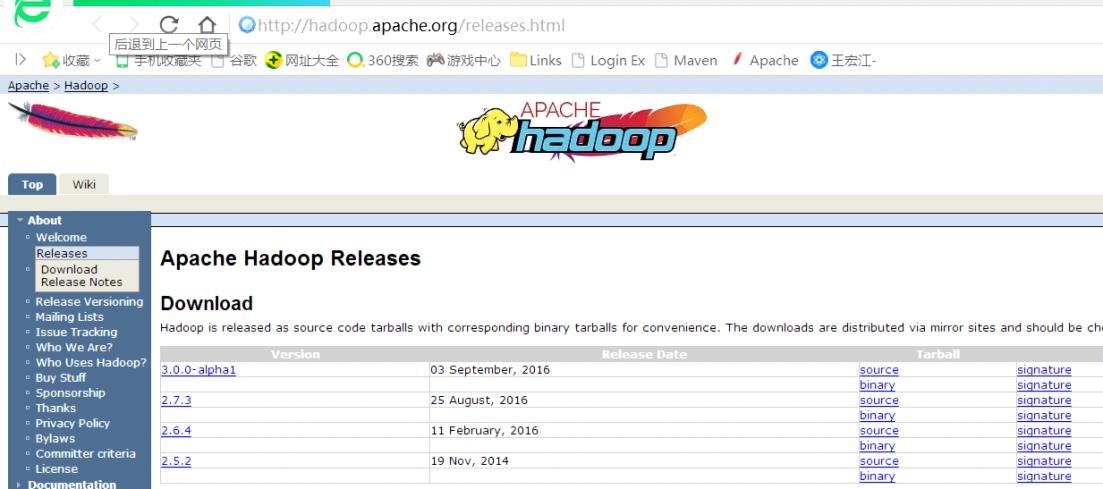
此外，还可以通过yarn协调多种不同计算框架（MR，Spark）

**最新Hadoop3.0** 2017年9月释出的。

### Hadoop伪分布式安装

下载地址：

<http://hadoop.apache.org/releases.html>



注意：

source表示源码

binary表示二级制包（安装包）

安装模式

**单机模式**：不能使用HDFS，只能使用MapReduce,所以单机模式最主要的目的是在本机调试

mapreduce代码

**伪分布式模式**：用多个线程模拟多台真实机器，即模拟真实的分布式环境。 **完全分布式模式**：用多台机器（或启动多个虚拟机）来完成部署集群。

安装步骤：

0.关闭防火墙

**执行：**service iptables stop 这个指令关闭完防火墙后，如果重启，防火墙会重新建立，所以，如果想重启后防火墙还关闭，

**需额外执行：**chkconfig iptables off

1.配置主机名

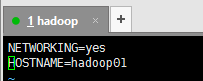
**执行：vim /etc/sysconfig/network**

编辑主机名

注意：主机名里不能有下滑线，或者特殊字符 #$，不然会找不到主机导致无法启动这种方式更改主机名需要重启才能永久生效，因为主机名属于内核参数。

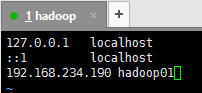
如果不想重启，可以执行：hostname hadoop01。但是这种更改是临时的，重启后会恢复原主机名。

所以可以结合使用。先修改配置文件，然后执行:hostname hadoop01 。可以达到不重启或重启都是主机名都是同一个的目的

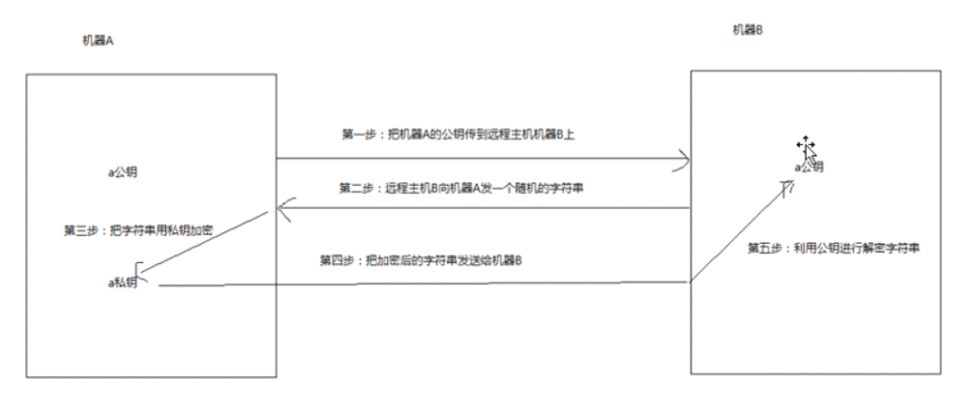


2.配置hosts文件

执行：vim /etc/hosts



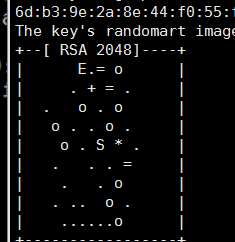
3.配置免秘钥登录



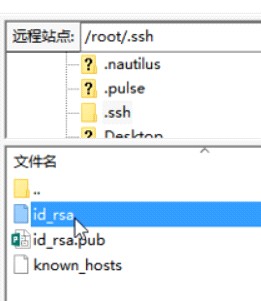
在hadoop01节点执行：

**执行：ssh-keygen**

然后一直回车



生成节点的公钥和私钥，生成的文件会自动放在/root/.ssh目录下

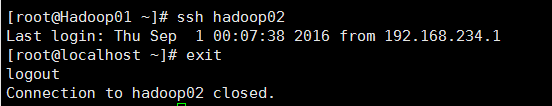


然后把公钥发往远程机器，比如hadoop01向hadoop01发送

**执行：ssh-copy-id root@hadoop01**

此时，hadoop02节点就是把收到的hadoop秘钥保存在

/root/.ssh/authorized\_keys 这个文件里，这个文件相当于访问白名单，凡是在此白明白存储的秘钥对应的机器，登录时都是免密码登录的。

当hadoop01再次通过ssh远程登录hadoop01时，发现不需要输入密码了。

在hadoop02节点执行上述上述步骤，让hadoop02节点连接hadoop01免密码登录

4.配置自己节点登录的免密码登录

如果是单机的伪分布式环境，节点需要登录自己节点，即hadoop01要登录hadoop01

但是此时是需要输入密码的，所以要在hadoop01节点上

**执 行 ：ssh-copy-id root@hadoop01**

5.安装和配置jdk

执行： vi /etc/profile

2）在尾行添加

#set java environment

JAVA\_HOME=/usr/local/src/java/jdk1.7.0\_51

PATH=$JAVA\_HOME/bin:$PATH

CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar export JAVA\_HOME PATH CLASSPATH

保存退出

* 1. source /etc/profile 使更改的配置立即生效
  2. java -version 查看JDK版本信息。如显示1.7.0证明成功。

6.上传和解压hadoop安装包**执 行 ：tar -xvf hadoop……** 目录说明：

bin目录：命令脚本

etc/hadoop:存放hadoop的配置文件

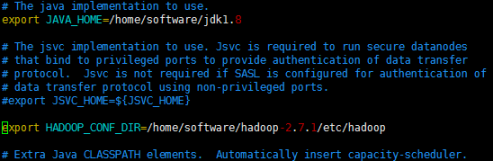
lib目录：hadoop运行的依赖jar包

sbin目录：启动和关闭hadoop等命令都在这里

libexec目录：存放的也是hadoop命令，但一般不常用最常用的就是bin和etc目录

7.修改hadoop的java\_home路径和hadoop\_conf\_dir 路径

/home/software/hadoop-2.7.1/etc/hadoop# vim hadoop-env.sh

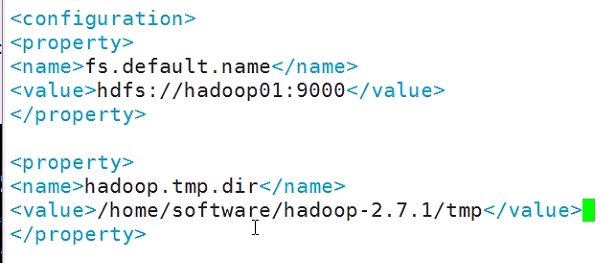


source hadoop-env.sh 让配置立即生效

8.vim core-site.xml

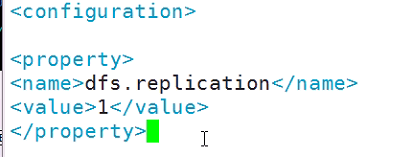
用来指定hdfs的老大，namenode的地址

用来指定hadoop运行时产生文件的存放目录

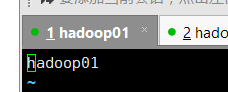


9.vim hdfs-site.xml

<!--指定hdfs保存数据副本的数量，包括自己，默认值是3,如果是伪分布模式，此值是1-->

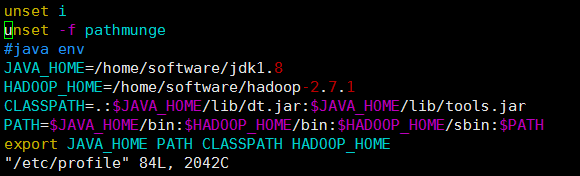


10.配置slaves文件



11.配置hadoop的环境变量

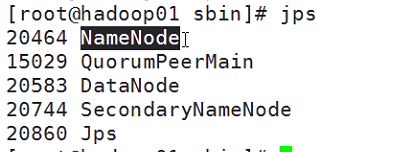
 **配置代码：**



JAVA\_HOME=/home/software/jdk1.8 HADOOP\_HOME=/home/software/hadoop-2.7.1 CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar PATH=$JAVA\_HOME/bin:$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin:$PATH export JAVA\_HOME PATH CLASSPATH HADOOP\_HOME

12格式化namenode

#hadoop namenode -format

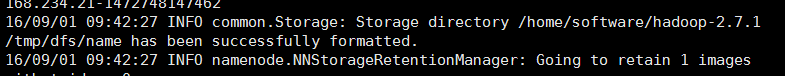


13.为什么要格式化

**执行：hadoop namenode -format**

如果不好使，可以重启linux

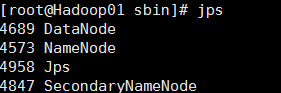
当出现：successfully，证明格式化成功



Hadoop的启动

切换到sbin目录，

**执行：start-dfs.sh 启动hadoop相关的服务**

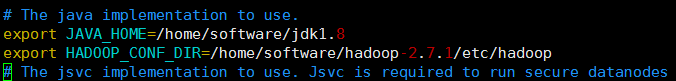


 **注：如果在启动时，报错：**Cannot find configuration directory: /etc/hadoop

**解决办法：**

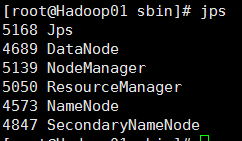
编辑 etc/hadoop下的 hadoop-env.sh 文件，添加如下配置信息：

export HADOOP\_CONF\_DIR=/home/software/hadoop-2.7.1/etc/hadoop



然后执行 source hadoop-env.sh 使配置立即生效

**执行：start-yarn.sh 启动yarn相关的服务**



在浏览器访问：

192.168.234.21:50070来访问hadoop的管理页面

Hadoop-HDFS命令

创建目录:hadoop fs -mkdir /park01

查看目录:hadoop fs -ls /

将操作系统的目录放在HDFS的park目录下:

hadoop fs -put 1.txt /park01

hadoop fs -put hadoop-2.7.1\_64bit.tar.gz /park01

IMG_256

把hdfs文件系统下park目录的文件下载到linux的home目录下

hadoop fs -get /park01/1.txt ./

hadoop fs -rmr /park01

合并:hadoop fs -getmerge /park01 ./5.txt

查看1.txt 这个文件block信息以及机架信息:

hadoop fsck /park01/1.txt -files -blocks -locations

hadoop fsck /park01/hadoop-2.7.1\_64bit.tar.gz -files -blocks -locations

HDFS常用指令

|  |  |
| --- | --- |
| **命令** | **说明** |
| 1.执行：hadoop fs -mkdir /park | 在hdfs 的根目录下，创建 park目录 |
| 2.执行：hadoop fs -ls / | 查看hdfs根目录下有哪些目录 |
| 3.执行：hadoop fs -put /root/1.txt /park | 将linux操作系统root目录下的1.txt放在hdfs的park目录下 |
| 4.执行：hadoop fs -get /park/jdk /home | 把hdfs文件系统下park目录的文件下载到linux的home目录下 |
| 5.执行：hadoop fs -rm /park/文件名 | 删除hdfs 的park目录的指定文件 |
| 6.执行：hadoop fs -rmdir /park | 删除park目录，但是前提目录里没有文件 |
| 7.执行：hadoop fs -rmr /park | 删除park目录，即使目录里有文件 |
| 8.执行：hadoop fs -cat /park/a.txt | 查看park目录下的a.txt文件 |
| 9.执行：hadoop fs -tail /park/a.txt | 查看park目录下a.txt文件末尾的数据 |
| 10.执行：haddop jar xxx.jar | 执行jar包 |
| 11.执行：hadoop fs -cat  /park/result/part-r-00000 | 查看 /park/result/part-r-00000文件的内容 |
| 12.执行：hadoop fs –mv /park02  /park01 | 将HDFS上的park02目录重名为park01命令。 |
| 13.执行：hadoop fs -mv /park02/1.txt  /park01 | 将park02目录下的1.txt移动到/park01目录下 |
| 14.执行：hadoop fs -touchz /park/2.txt | 创建一个空文件 |
| 15.执行：hadoop fs -getmerge /park  /root/tmp | 将park目录下的所有文件合并成一个文件，并下载到linux的root目录下的tmp目录 |
| 16.执行：hadoop dfsadmin -safemode leave 离开安全模式  执行：hadoop dfsadmin -safemode  enter 进入安全模式 | 离开hadoop安全模式  在重新启动HDFS后，会立即进入安全模式，此时不能操作hdfs中的文件，只能查看目录文件名等，读写操 作都不能进行。  namenode启动时，需要载入fsimage文件到内存，同时执行edits文件中各项操作一旦在内存中成功建立文件系统元数据的映射，则创建一个新的fsimage文件（这个步骤不需要SNN的参与）和一个空的编辑文件。 此时namenode文件系统对于客户端来说是只读的。  在此阶段NameNode收集各个DataNode的报告，当数据块达到最小复本数以上时，会被认为是“安全”的， 在一定比例的数据块被确定为安全后，再经过若干时间，安全模式结束  当检测到副本数不足的数据块时，该块会被复制直到到达最小副本数，系统中数据块的位置并不是  namenode维护的，而是以块列表的形式存储在datanode中。  当启动报如下错误时:  org.apache.hadoop.dfs.SafeModeException: Cannot delete /user/hadoop/input. Name node is in safe mode |
| 17.执行：hadoop dfsadmin -rollEdits | 手动执行fsimage文件和Edis文件合并元数据 |
| 18.执行：hadoop dfsadmin -report | 查看存活的datanode节点信息 |
| 19.执行：hadoop fsck /park | 汇报/park目录 健康状况 |
| 20.执行：hadoop fsck /park/1.txt  -files -blocks -locations -racks | 查看1.txt 这个文件block信息以及机架信息 |

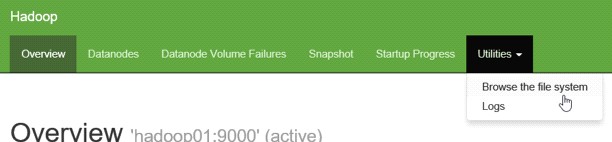
|  |  |
| --- | --- |
|  |  |
| 21.hadoop fs -expunge | 手动清空hdfs回收站 |

其他指令

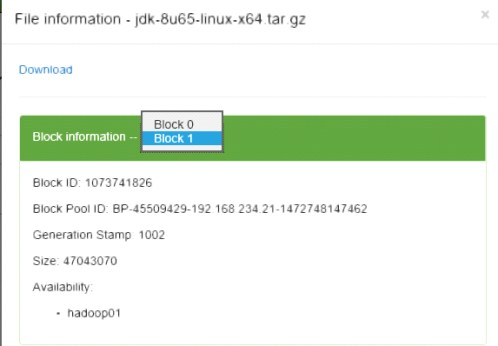
|  |  |
| --- | --- |
| 命令 | 说明 |
| hadoop fs -cp /park01/1.txt  /park02 | 将HDFS上 /park01下的1.txt拷贝一份到 /park02目录下。目标路径可以有多个，用空格隔开，比如：  hadoop fs -cp /park01/1.txt /park02 /park03…… |
| hadoop fs -du /park/1.txt | 查看HDFS上某个文件的大小。也可以查看指定目录，如果是目录的话，则列出目录下所有的文件及其大小，比如：  hadoop fs -du /park |
| hadoop fs -copyFromLocal  /home/1.txt /park01 | 将本地文件1.txt上传到/park01目录下 |
| hadoop fs -copyToLocal  /park01/1.txt /home | 将HDFS上的1.txt 拷贝到本地文件系统 |
| hadoop fs -lsr / | 递归查看指定目录下的所有内容 |

HDFS控制台页面

通过访问：192.168.234.21:50070来查看hdfs系统



将文件上传到hdfs后，如果大于128M,会被切成两块



# HDFS细节

**存储:**

纵向扩展:硬件扩展,存在上限瓶颈.

横向扩展:分布式思想,多态机器

分布式存储的新概念:切块存储

namenode的职责

:1.管理各个DataNode节点

2.管理分布式文件系统的元数据信息

3.通过RPC心跳机制来监控DataNode的状态,对于网络通信,HTTP主外,RPC主内.

4为了确保数据存储的可靠性,采用冗余机制来实现,

注意:副本数量不宜过多(一般3副本策略),会降低整个集群的磁盘利用率(3个的话1/3),

#### HDFS架构图

知识点

1. HDFS Hadoop Distributed File System Hadoop的分布式文件系统，可以存储海量数据（文件， 图片，影音等），实际工作中，一般存储的都是用户的访问日志（.txt）

#### HDFS之所以能够存储海量数据，原因是一个可扩展的分布式架构，硬盘存储空间不够，加服务器即 可。

1. HDFS是基于Google的一篇论文《Google File System》
2. namenode，名字节点。最主要的职责是管理和存储HDFS的元数据信息（比如文件名，文件大小， 文件切块的数量，每个文件块的大小，文件块编号，存储在哪个datanode上），可以通过指令： hadoop fsck /park01/1.txt -files -blocks -locations
3. namenode不存储文件块
4. namenode除了存储元数据信息以外，还会通过RPC心跳机制来管理各个datanode
5. namenode会把元数据信息放在namenode服务器的内存里，目的是供用户快速查询。

#### namenode为了确保元数据存储的可靠性，会将元数据落地，存储的目录路径有core-site.xml里的

hadoop.tmp.dir来指定的。

注意：此属性默认是放在linux的 /tmp 目录，所以在工作中一定要更换此目录。

#### namenode底层是通过两个文件来进行元数据管理：

①Edits文件 当客户端发起写请求时，Edits文件都会进行记录写请求，比如：-mkdir -put -mv

②Fsimage文件 存储元数据信息的

以上这个两个文件会定期做一次合并，合并的目的是确保Fsimage文件里的数据是最新的。合并周期默 认是3600s（1小时）

10.以上这个两个文件可以在配置的元数据目tmp/dfs/name/current找到

#### 11.格式化指令：hadoop namenode -format

它的作用是在元数据目录生成新的Edits和Fsimage文件。

这个指令 在初次Hadoop时需要执行一次。

#### 12.格式化指令很危险，因为它会清空之前所有的元数据。所以在实际工作，初次使用完之后，会通过 配置文件的使得这个指令失效。

13.Edits和Fsimage文件可以通过指令手动合并：hadoop dfsadmin -rollEdits

14.为了当namenode宕机后，SNN能够工作，我们需要SNN具有元数据数据，

所以Hadoop底层机制是让SNN来做元数据的合并工作，通过这机制使得NN和SNN都有元数据信息 了。

注意：以上这个机制存在一个问题，就是可能会造成元数据丢失

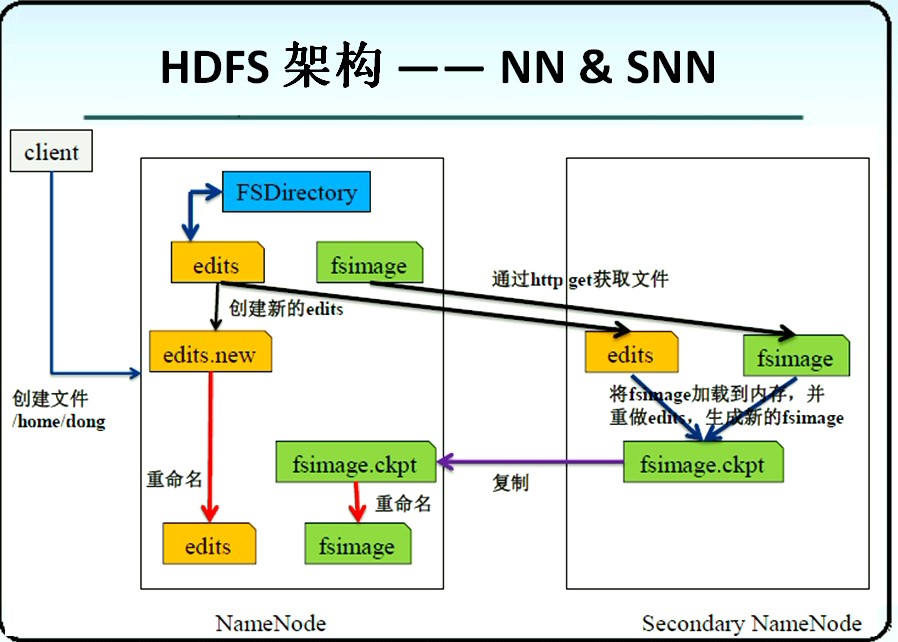
#### 比如：

3：00 Edits和Fsimage 合并一次

3：30 namenode 宕机了

#### 相当于丢失了30分钟之内的元数据信息。

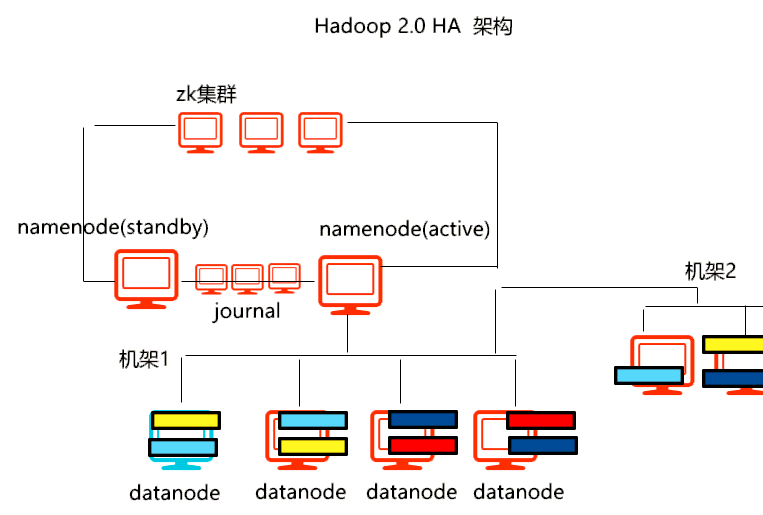
这个机制是Hadoop1.0的机制，是存在问题的，即SNN合并不能达到实时热备，所以会元数据丢失， 即Hadoop1.0的namenode还是会存在单点故障问题。



15.下图是Hadoop2.0 集群的架构图。

Hadoop2.0如果伪分布式模式，会看到SNN，但实际没有作用

Hadoop2.0如果是完全分布式，就没有SNN进程了



16.可以指令单独启停某个进程:

启动：

hadoop-daemon.sh start namenode

#### hadoop-daemon.sh start datanode

hadoop-daemon.sh start secondarynamenode

停止：

hadoop-daemon.sh stop namenode

此外，掌握：start-dfs.sh 启动和HDFS相关的所有进程

17.当namenode挂掉再次启动时，底层会将Edits和Fsimage合并一次。

3：10 namenode 宕机

#### 即3:00 ~3：10会产生一些操作记录，

所以通过这种机制，可以确保namenode宕机再启动之后，Fsimage是最新的 18.当整个HDFS启动（namenode和datanode）时，底层每台datanode都会向namenode汇报自身 的存储状态信息（存储了哪些文件块），namenode收到这写数据后，汇总并检查。检查文件块是否 缺少，数据是否丢失，以及每个文件块的副本数量是否达到要求（集群环境，3副本），如果检查有问

#### 题，HDFS会进入安全模式，在安全模式下，要完成数据的修复、副本的复制。安全模式直到数据恢复 完毕之后自动退出

19.如果HDFS处于安全模式，对外只能提供读服务，不能提供写服务。 可以通过指令：

hadoop dfsadmin -safemode enter hadoop dfsadmin -safemode leave

20.如果是伪分布式模式，在hdfs-site.xml 配置副本数量，只能配置1，因为如果大于1，会使得HDFS 一直安全模式而不退出（因为副本数量一直满足不了要求，就一台服务器，只能存一个副本）

#### datanode，数据节点，专门用来存储和管理文件块。

1. datanode会定期向namenode发送心跳，心跳周期默认是3s

#### 23.配置datanode地址列表是在slaves配置文件里配置的

1. HDFS存储文件的方式是切块存储，

Hadoop 1.0 切块大小64MB

Hadoop 2.0 切块大小128MB

#### 比如：上传来个文件①1.txt 257MB ②2.txt 100MB ,并设定副本数=1

①1.txt 1块：128MB 2块：128MB 3块：1MB

②2.txt 1块：100MB

#### 综上，共4个文件块。

**强调：**切块是以文件为单位的，不同文件的文件块不能共用。

块是多大，在磁盘上就占多大。比如1MB文件块就占1MB，不会浪费127MB的磁盘空间的。

#### HDFS的特点：当文件上传HDFS之后，就不允许修改此文件。所以HDFS的适用场景：once-write- many-read （一次写入，多次读取）

1. HDFS不允许修改数据，但允许追加数据
2. HDFS是否适合存储海量的小文件（比如几kb,几mb)？

#### 不适合，因为每个文件都会占用一条元数据信息，根据经验，一条元数据大约在150字节，即如果由海

量小文件时，会占用大量的namenode服务内存空间。

HDFS 回收站机制

Hadoop回收站trash，默认是关闭的。

修改conf/core-site.xml,增加

 配置示例：

<property>

<name>fs.trash.interval</name>

<value>1440</value>

<description>

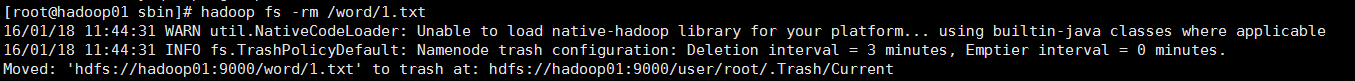
Number of minutes between trash checkpoints. If zero, the trash feature is disab led. </description>

</property>

注：value的时间单位是分钟，如果配置成0,表示不开启HDFS的回收站。

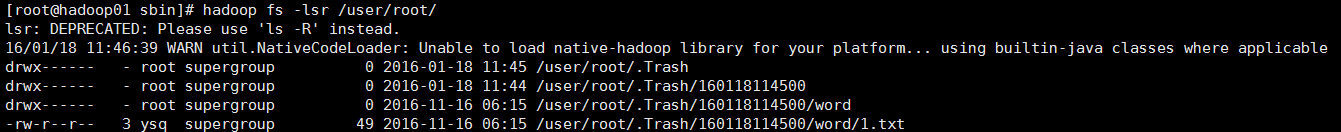
1440=24\*60,表示的一天的回收间隔，即文件在回收站存在一天后，被清空。

启动回收站后，比如我们删除一个文件：



我们可以通过递归查看指令，找到我们要恢复的文件放在回收站的哪个目录下

执 行 ：hadoop fs -lsr /user/root/.Trash



找到文件路径后，如果想恢复，执行hdfs 的mv 指令即可，（mv指令可用于文件的移动）

1. HDFS的守护进程
   1. 主节点：Namenode、SecondaryNamenode
   2. 从节点：Datanode
2. YARN的守护进程
   1. 主节点：ResourceManager
   2. 从节点：NodeManager

## big\_day07

Hadoop02

**Edits和Fsimage机制详解**

**概述**

fsimage镜像文件包含了整个HDFS文件系统的所有目录和文件的indoe（节点）信息，比如：/park01/node，会记录每个节点inodid，以及节点之间父子路径。

以及文件名，文件大小，文件被切成几块，每个数据块描述信息、修改时间、访问时间等；此外还有对目录的修改时间、访问权限控制信息(目录所属用户，所属组等)等。

另外，edits文件主要是在NameNode已经启动情况下对HDFS进行的各种**更新操作**进行记录，比如 ：hadoop fs -mkdir  hadoop fs -delete  hadoop fs -put等。

对于每次事务操作，都会用一个TXID(事务id)来标识，OP\_MKDIR  OP\_DELETE等。

总结：**Edits文件存储的是操作(HDFS客户端执行所有的写操作都会被记录到edits文件中)，而fsimage文件存储的是执行操作后，变化的状态。（元数据）**

**知识点**

1.当执行格式化指令时候,会在指定元数据目录生成dfs/name/current/,最开始只有fsimage,没有edits文件(因为没有启动HDFS)

2.当初次启动HDFS,会生成edits\_inprogress\_0000000000000000089,此文件用于记录事务(写操作)

3.HDFS对于每次写操作,都会用一个事务id(txid)来记录,txid是递增的.

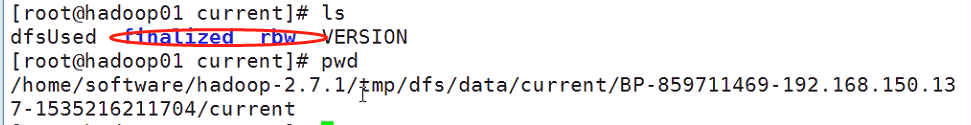
4.edits\_0000000000000000003\_0000000000000000007,数字表示的合并后起始的事务id和终止事务id.

5.seen\_txid存储的当前的事务id,和edits\_inprogress最后的数字一致.

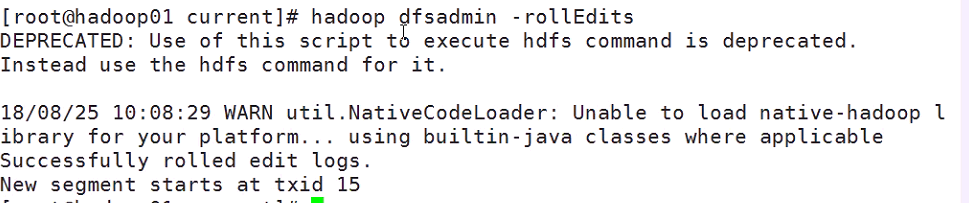
6.datanode存储块的目录路径:

hadoop-2.7.1/tmp/dfs/data/current/BP-859711469-192.168.150.137-1535216211704/current/finalized/subdir0/subdir0

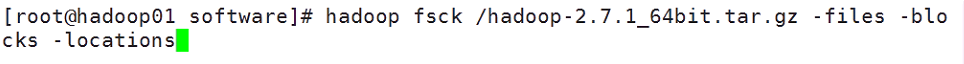
7**.finalized**此目录存储的已经完毕的数据块**,rbw**目录存的是正在写但还未写完的数据块



重新删除tmp,上传文件查看



手动合并命令:



**查看Edits文件和Fsimage文件**

hdfs oev -i edits\_0000000000000000001-0000000000000000003 -o edits.xml

hdfs oiv -i fsimage\_0000000000000000012 -o fsimage.xml -p XML

**HDFS回收站机制**

HDFS 回收站机制

2016年1月18日

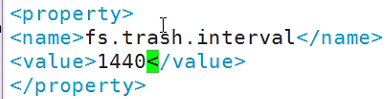
11:49

Hadoop回收站trash，默认是关闭的。

修改conf/core-site.xml,增加

**配置示例：**

/home/software/hadoop-2.7.1/etc/hadoop# vim core-site.xml



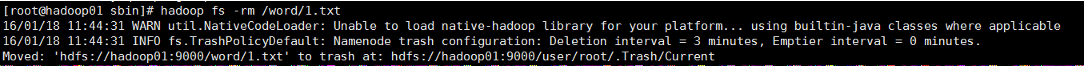
 <description>Number of minutes between trash checkpoints.   If zero, the trash feature is disabled. </description>

注：value的时间单位是分钟，如果配置成0,表示不开启HDFS的回收站。

1440=24\*60(分钟),表示的一天的回收间隔，即文件在回收站存在一天后，被清空。

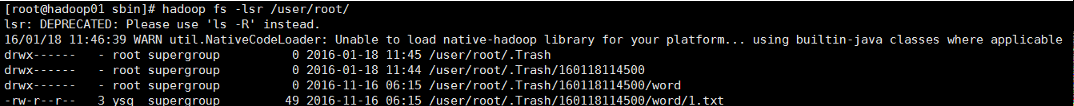
启动回收站后，比如我们删除一个文件：

sbin# hadoop fs -rm /word/1.txt



我们可以通过递归查看指令，找到我们要恢复的文件放在回收站的哪个目录下

执行：hadoop fs -lsr  /user/root/.Trash



 找到文件路径后，如果想恢复，执行hdfs 的mv 指令即可，（mv指令可用于文件的移动）

HDFS API操作

**实现步骤：**

1.创建java工程

2.导入hadoop依赖jar包

**连接namenode以及读取hdfs中指定文件**

@Test

public void testConnectNamenode() throws Exception{

Configuration conf=new Configuration();

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.21:9000"), conf);

InputStream in=fs.open(new Path("/park/1.txt"));

OutputStream out=new FileOutputStream("1.txt");

IOUtils.copyBytes(in, out, conf);

}

**上传文件到hdfs上**

@Test

public void testPut() throws Exception{

Configuration conf=new Configuration();

conf.set("dfs.replication","1");

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.21:9000"),conf,"root");

ByteArrayInputStream in=new ByteArrayInputStream("hello hdfs".getBytes());

OutputStream out=fs.create(new Path("/park/2.txt"));

IOUtils.copyBytes(in, out, conf);

}

**从hdfs上删除文件**

@Test

public void testDelete()throws Exception{

Configuration conf=new Configuration();

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.21:9000"),conf,"root");

//true表示无论目录是否为空，都删除掉。可以删除指定的文件

fs.delete(new Path("/park01"),true);

//false表示只能删除不为空的目录。

fs.delete(new Path("/park01"),false);

fs.close();

}

**在hdfs上创建文件夹**

@Test

public void testMkdir()throws Exception{

Configuration conf=new Configuration();

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.21:9000"),conf,"root");

fs.mkdirs(new Path("/park02"));

}

**查询hdfs指定目录下的文件**

@Test

public void testLs()throws Exception{

Configuration conf=new Configuration();

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.21:9000"),conf,"root");

FileStatus[] ls=fs.listStatus(new Path("/"));

for(FileStatus status:ls){

System.out.println(status);

}

}

**递归查看指定目录下的文件**

@Test

public void testLs()throws Exception{

Configuration conf=new Configuration();

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.214:9000"),conf,"root");

RemoteIterator<LocatedFileStatus> rt=fs.listFiles(new Path("/"), true);

while(rt.hasNext()){

System.out.println(rt.next());

}

}

**重命名**

/\*重命名

 \*/

@Test

public void testCreateNewFile() throws Exception{

Configuration conf=new Configuration();

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.176:9000"),conf,"root");

fs.rename(new Path("/park"), new Path("/park01"));

}

**获取文件的块信息**

@Test

public void testCopyFromLoaclFileSystem() throws Exception{

Configuration conf=new Configuration();

FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.176:9000"),conf,"root");

**BlockLocation[] data=fs.getFileBlockLocations(new Path("/park01/1.txt"),0,Integer.MaxValue);**

for(BlockLocation bl:data){

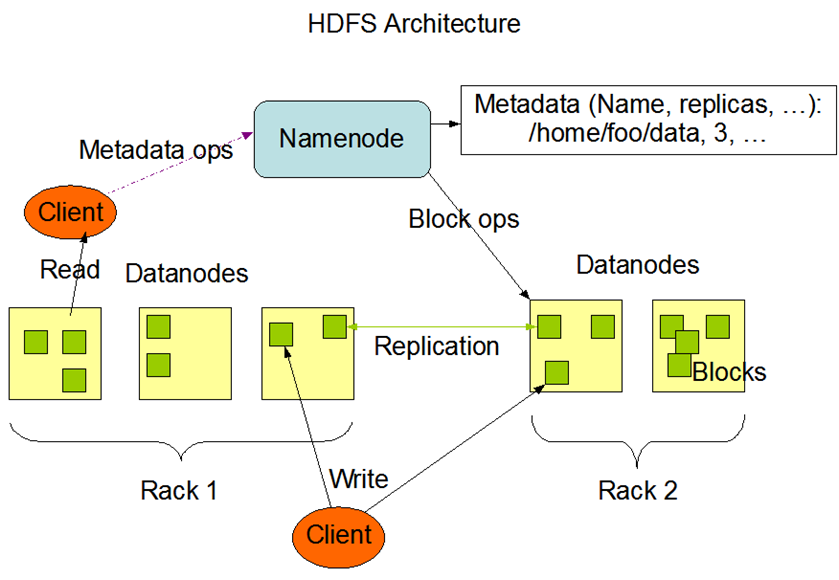
System.out.println(bl);

}

}

HDFS各流程图

**HDFS架构图**



1.namenode。名字节点，最主要的作用是管理元数据

2.Metadata 。元数据信息

3.文件块 Block

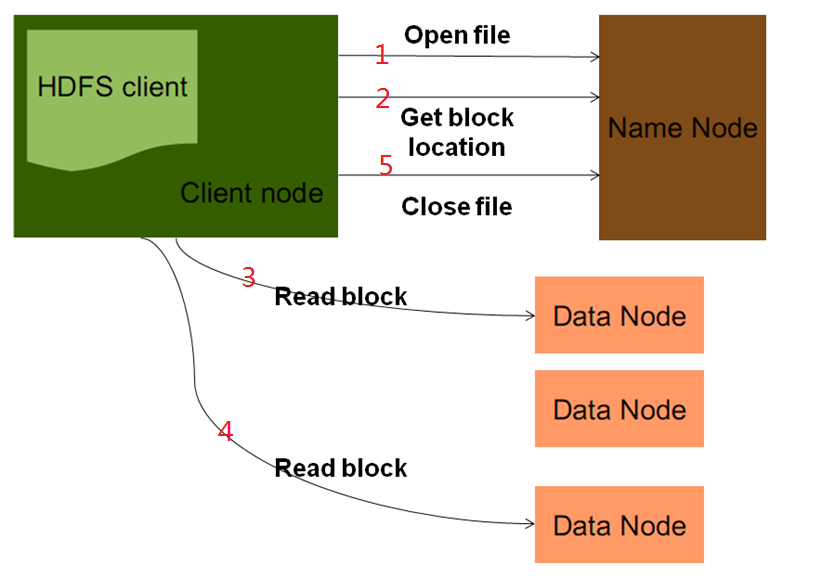
4.datanode。数据节点，用来存储文件块

5.Replication 文件块副本。一般采用的是3副本策略

6.Rack 机架

7.Client 客户端。凡是通过指令或代码操作的一端都属于客户端。

**从HDFS下载文件过程**



1.Client向namenode发起 Open file 请求。目的是获取指定文件的输入流。

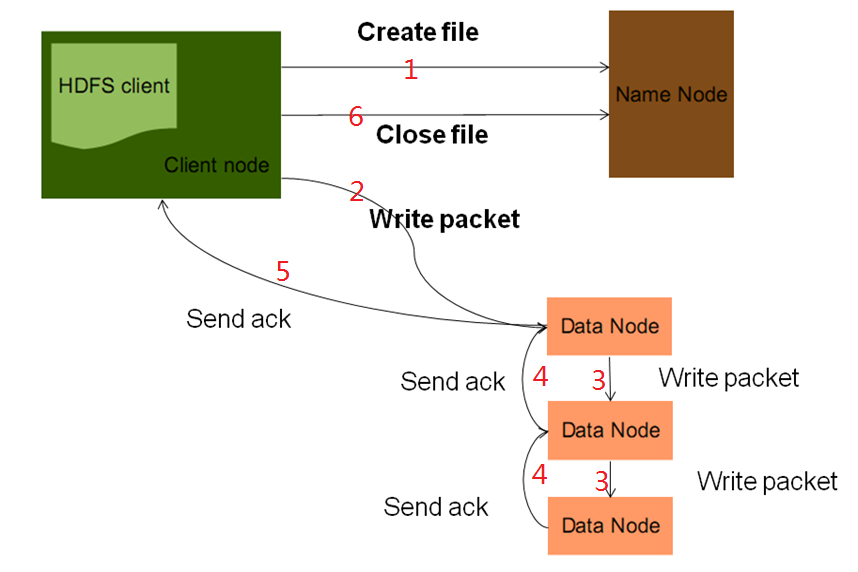
namenode收到请求之后，会检查路径的合法性，此外，还是检查客户端的操作权限。如果检测未通过，则直接报错返回。后续过程不会发生。

2.Client也会向namenode发起：Getblockloaction请求，获取指定文件的元数据信息。如果第一步的检测通过，namenode会将元数据信息封装到输入流里，返回给客户端。

3.4 客户端根据元数据信息，直接去对应的datanode读取文件块，然后下载到本地（创建本地的输出流，然后做流的对接）

5.读完后，关流。

**上传文件到HDFS**



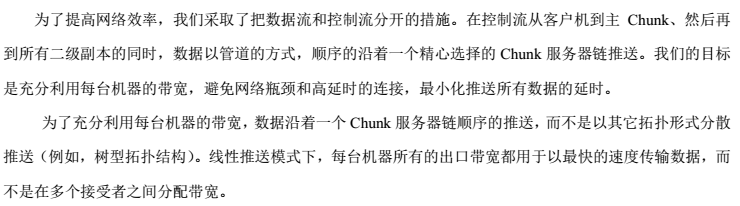
1.Client向namenode发现 Create file请求，目的是获取HDFS文件的输出流。namenode收到请求后，会检测路径的合法性和权限。如果检测未通过，直接报错返回。

如果通过检测，namenode会将文件的切块信息（比如文件被切成几块，每个文件块的副本存在哪台datanode上），然后把这些信息封装到输出流里，返回给客户端。

**所以注意：**文件块的输出（上传）是客户端直接和对应DN交互的，namenode的作用是告诉Client文件块要发送给哪个datanode上。

2.Client通过输出流，发送文件块（底层会将一个文件块打散成一个一个的packet，每个packet的大小=64kb）。这个过程的机制，叫Pipeline（数据流管道机制）

这种机制的目的：



3.4.5 。通过数据流管道机制，实现数据的发送和副本的复制。每台datanode服务器收到数据之后，会向上游反馈ack确认机制。直到第五步的ack发送给Client之后，再发送下一个packet。依次循环，直到所有的数据都复制完毕。此外，在底层传输的过程中，会用到全双工通信。

补充：建议看《Google File System》的3.2节

6.数据上传完之后，关流。

最后的补充：第一步Client获取的输出流，对应的类：DFSOutPutStream。建议阅读源码注释：

 \* DFSOutputStream creates files from a stream of bytes.

 \*

 \* The client application writes data that is cached internally by

 \* this stream. Data is broken up into packets, each packet is

 \* typically 64K in size. A packet comprises of chunks. Each chunk

 \* is typically 512 bytes and has an associated checksum with it.

 \*

 \* When a client application fills up the currentPacket, it is

 \* enqueued into dataQueue.  The DataStreamer thread picks up

 \* packets from the dataQueue, sends it to the first datanode in

 \* the pipeline and moves it from the dataQueue to the ackQueue.

 \* The ResponseProcessor receives acks from the datanodes. When an

 \* successful ack for a packet is received from all datanodes, the

 \* ResponseProcessor removes the corresponding packet from the

 \* ackQueue.

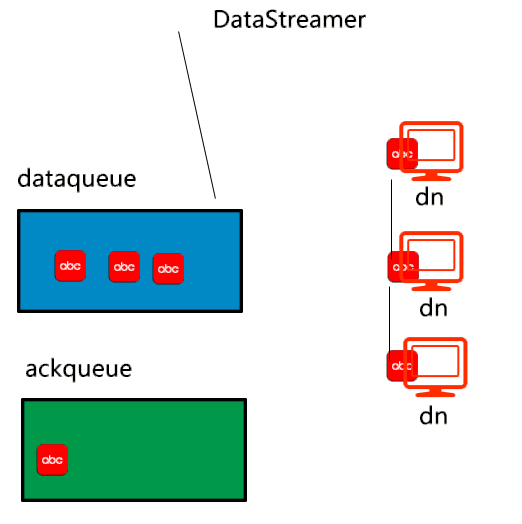
 \*

 \* In case of error, all outstanding packets and moved from

 \* ackQueue. A new pipeline is setup by eliminating the bad

 \* datanode from the original pipeline. The DataStreamer now

 \* starts sending packets from the dataQueue.



**从HDFS删除文件的流程**

1、客户端向namenode发现 删除文件指令，比如：hadoop fs  -rm /park01/1.txt

2、namenode收到请求后，会检查路径的合法性以及权限

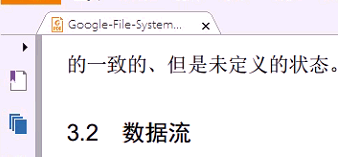
3、如果检测通过，会将对应的文件从元数据中删除。（注意，此时这个文件并没有真正从集群上被删除）

4、每台datanode会定期向namenode发送心跳，会领取删除的指令，找到对应的文件块，进行文件块的删除。

**HDFS各流程图**

上传文件到HDFS

namenode检测权限.路径,分配块编号等,记录edits,分装输出流输出



DFSClient |  Namenode |  Datanode 源码分析顺序图：

**DFSClient**

**|-------ClientProtocol**

**|-------DFSInputStream**

**|-------LocatedBlocks**

**|-------BlockReader**

**|-------DFSInputStream**

**|-------DFSOutputStream**

**|--------Packet**

**|--------pipeline**

**|--------DataStreamer**

**|--------ResponseProcessor**

**Datanode**

**|-------BlockSender**

**|-------DataXceiverServer**

**|-------- BlockReceiver**

**|-------DatanodeProtocol**

**|-------InterDatanodeProtocol**

**|-------ClientDatanodeProtocol**

**Namenode**

**|-------FSNameSystem**

**|-------FSDirectory**

**|-------FsImage**

**|-------LeaseManager**

**|-------HeartbeatManager**

**|-------HeartbeatThread**

**|------Monitor**

**一、DFSClient相关体系**

**DFSClient**

**|-------ClientProtocol**

**|-------DFSInputStream**

**|-------LocatedBlocks**

**|-------BlockReader**

**|-------DFSInputStream**

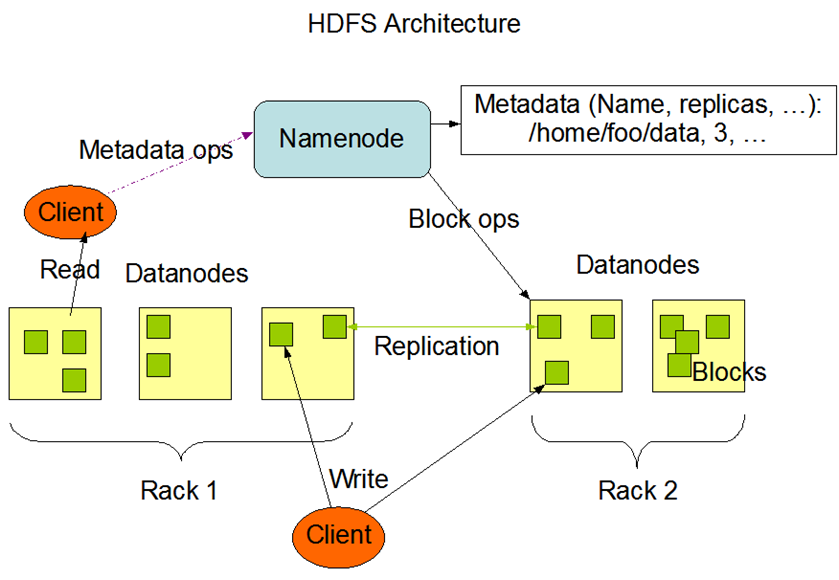
**|-------DFSOutputStream**

**|--------Packet**

**|--------pipeline**

**|--------DataStreamer**

**|--------ResponseProcessor**



**DFSClient类的介绍源码：**

\* DFSClient can connect to a Hadoop Filesystem and

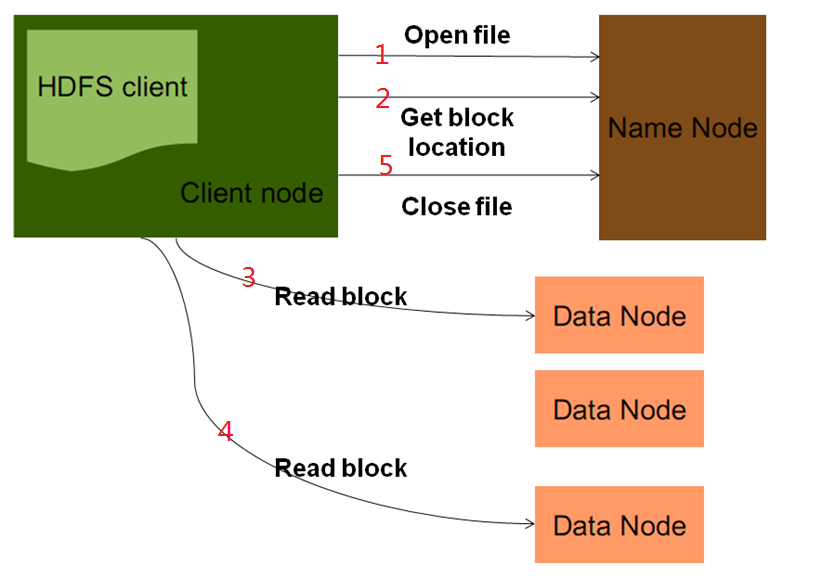
 \* perform basic file tasks.  It uses the ClientProtocol

 \* to communicate with a NameNode daemon, and connects

 \* directly to DataNodes to read/write block data.

DFSClient这个类的作用是用于客户端连接Hadoop的HDFS文件系统，并在HDFS文件系统上做基本的文件操作（任务）。这个类和通过RPC机制和HDFS文件系统通信的，具体的RPC通信协议接口类是：org.apache.hadoop.hdfs.protocol.ClientProtocol。

DFSClient通过PRC和namenode节点和datanode节点进行通信以及实现在datanode上执行读/写 block的操作



**重要方法源码：**

static LocatedBlocks **callGetBlockLocations**(ClientProtocol namenode,

      String src, long start, long length)

      throws IOException {

    try {

      return namenode.getBlockLocations(src, start, length);

    } catch(RemoteException re) {

      throw re.unwrapRemoteException(AccessControlException.class,

                                     FileNotFoundException.class,

                                     UnresolvedPathException.class);

    }

  }

这个函数主要是通过跟namenode的交互，来完成从namenode取得用户请求的file的元数据信息

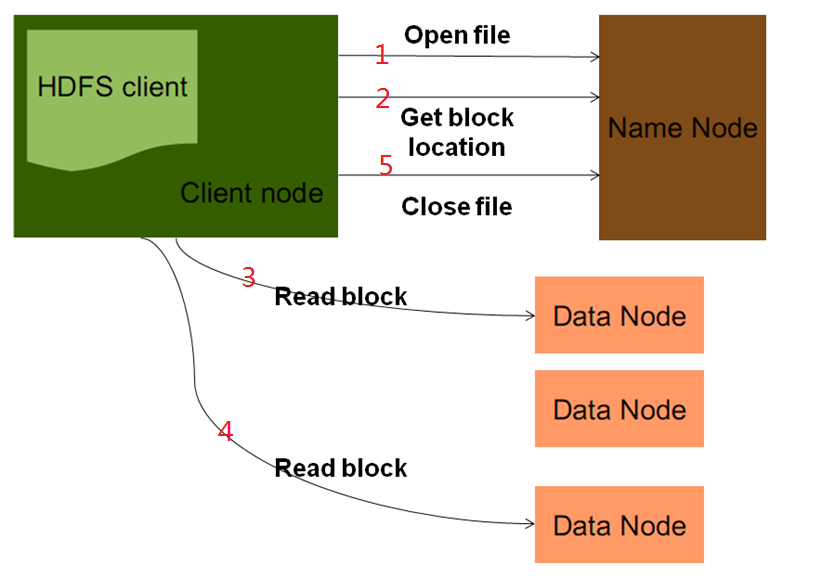
**ClientProtocol介绍**

客户端和namenode进行rpc通信的协议接口，实现类是org.apache.hadoop.hdfs.protocol.ClientProtocol。

DFSClient通过RPC机制和NameNode通信并获得文件的元数据信息（数据的存放位置，校验和，等等一些关键的信息），然后DFSClient再与DataNode通信（集群中的其它机器）通过数据I/O流来获取到指定的文件信息。

**DFSInputStream、BlockReader  介绍**

DFSInputStream是DFSClient用于读取datanode上文件的输入流。当DFSClient向namenode发送读文件请求之后，namenode会将此file的元数据信息返回： LocatedBlocks。这个类，封装了文件块的信息（每个文件块的大小、所在的datanode节点信息）。该输入流根据这些信息找到对应的DataNode，然后调用BlockReader类的read()方法，读取datanode节点上文件块里的内容。



**重要变量源码：**

public class DFSInputStream {

  private final DFSClient dfsClient;

  private BlockReader blockReader = null;

  private LocatedBlocks locatedBlocks = null;

  private DatanodeInfo currentNode = null;

}

**DFSOutputStream、Packet 、DataStreamer、ResponseProcessor介绍**

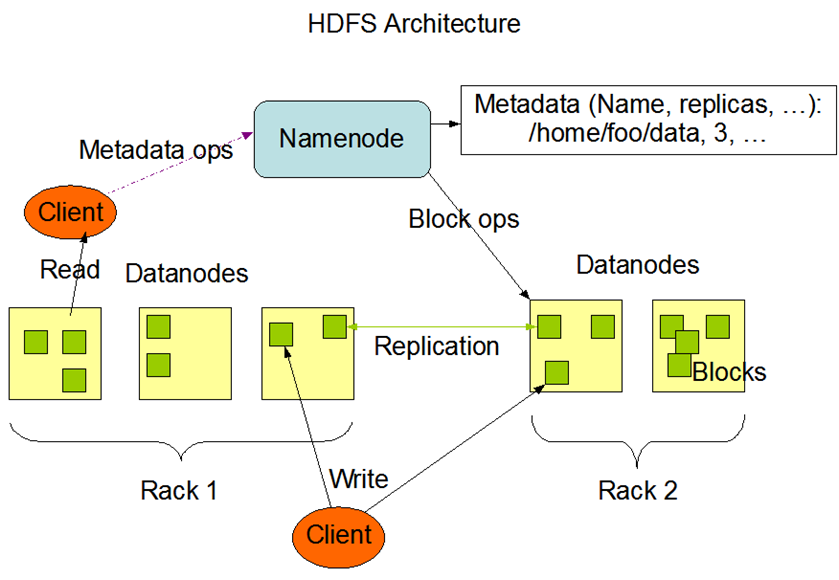
DFSoutputStream类是一个输出流，用于提供客户端将本地文件（在客户端的电脑上存的文件）上传到HDFS上。当客户端发送一个

写请求后（create请求），比如上传一个大文件。首先会通过DFSclient去找namenode,namenode首先会创建对应的目录，然后返回给DSFClient输出流。拿到输出输出流之后，  根据namenode返回的块信息blocksize，将数据变成64kb的packet存储到dataqueue队列里

**DataStreamer**会从dataQueue里取出一个一个的packet进行数据传输，然后形成一个数据流管道 pipeline。然后将

数据流管道输出给管道里的第一个datanode节点。第一个datanode节点将数据流管道信息交给管道中的第二个节点。直到管道里最后一个datanode完成存储并返回ack确认后，通知resoponse线程，然后DataStreamer发送下一个packet。

**DFSOutputStream里有2个队列和2个线程**



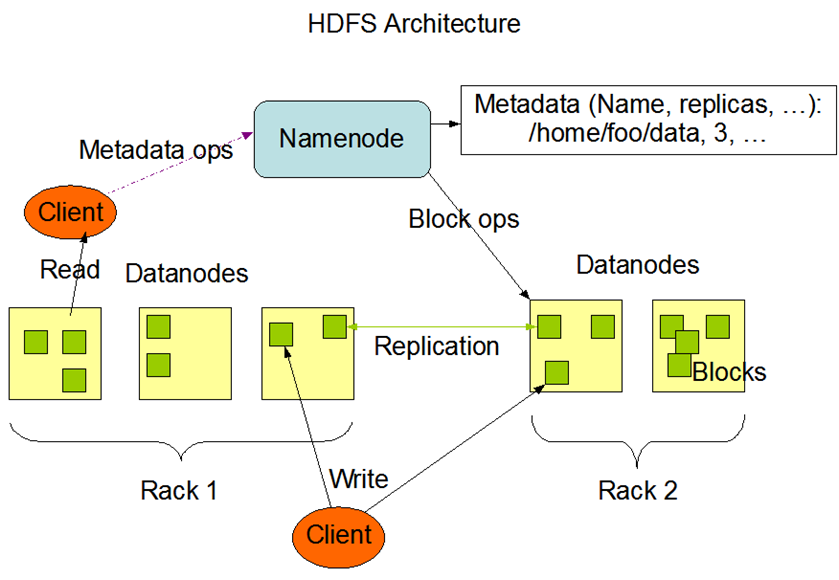
 两个队列相关代码：

dataQueue是数据队列，用于保存等待发送给datanode的数据包

 private final LinkedList<Packet> dataQueue = new LinkedList<Packet>();

ackQueue是确认队列，保存还没有被datanode确认接收的数据包

 private final LinkedList<Packet> ackQueue = new LinkedList<Packet>();



 两个线程：

streamer线程，不停的从dataQueue中取出数据包，发送给datanode

private DataStreamer streamer = new DataStreamer();

response线程，用于接收从datanode返回的反馈信息

private ResponseProcessor response = null;

在向DFSOutputStream中，写入数据（通常是byte数组）的时候，实际的传输过程是：

1、文件数据以字节数组进行传输，每个byte[]被封装成64KB的Packet，然后扔进dataQueue中

2、DataStreamer线程不断的从dataQueue中取出Packet，通过socket发送给datanode（向blockStream写数据）

    发送前，将当前的Packet从dataQueue中移除，并addLast进ackQueue

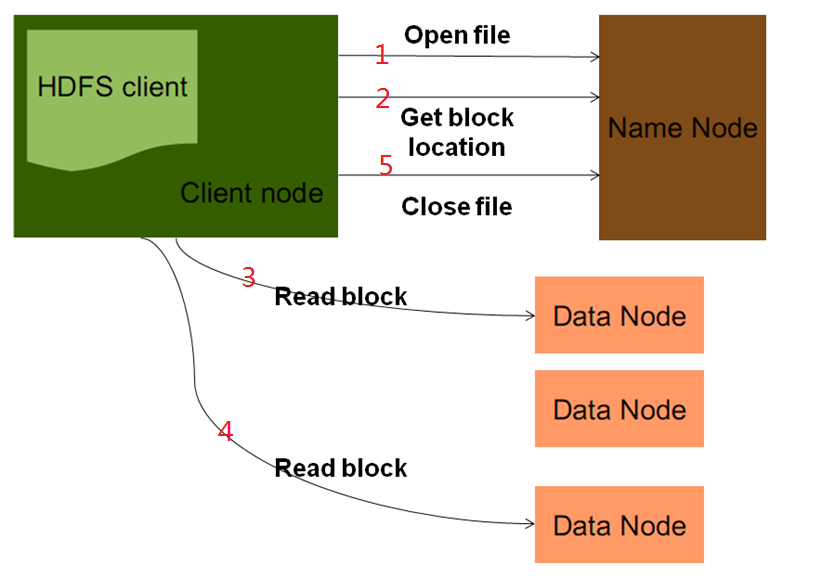
3、ResponseProcessor线程从blockReplyStream中读出从datanode的反馈信息

       反馈信息很简单，就是一个seqno，再加上每个datanode返回的标志（成功标志为  DataTransferProtocol.OP\_STATUS\_SUCCESS）

       通过判断seqno(序列号，每个Packet有一个序列号)，判断datanode是否接收到正确的包。

       只有收到反馈包中的seqno与ackQueue.getFirst()的包seqno相同时，说明正确。否则可能出现了丢包的情况。

4、如果一切OK，则从ackQueue中移出：ackQueue.removeFirst(); 说明这个Packet被datanode成功接收了。



**DataStreamer类重要源码：**

private DataStreamer() {

      isAppend = false;

**stage = BlockConstructionStage.PIPELINE\_SETUP\_CREATE;**

    }

在DatStreamer开始发送block时，创建pipeline数据流管道

**二、Datanode相关体系**

**Datanode**

**|-------BlockSender**

**|-------DataXceiverServer**

**|-------- BlockReceiver**

**|-------DatanodeProtocol**

**|-------InterDatanodeProtocol**

**|-------ClientDatanodeProtocol**

**BlockSender介绍**

1.当用户（客户端）向HDFS读取某一个文件时，客户端会根据数据所在的位置转向到具体的DataNode节点请求对应数据块的数据，此时DataNode节点会用BlockSender向该客户端发送数据；

2.当NameNode节点发现某个Block的副本不足时，它会要求某一个存储了该Block的DataNode节点向其它DataNode节点复制该Block，当然此时仍然会采用流水线的复制方式，只不过数据来源变成了一个DataNode节点；

3.HDFS开了一个调节DataNode负载均衡的工具Balancer，当它发现某一个DataNode节点存储的Block过多时，就会让这个DataNode节点转移一部分Blocks到新添加到集群的DataNode节点或者存储负载轻的DataNode节点上；

**sh start-balancer.sh -t %10**

百分数是磁盘使用偏差率，一般调节的范围在10%~20%间。

4.DataNode会定期利用BlockSender来检查一个Block的数据是否损坏。

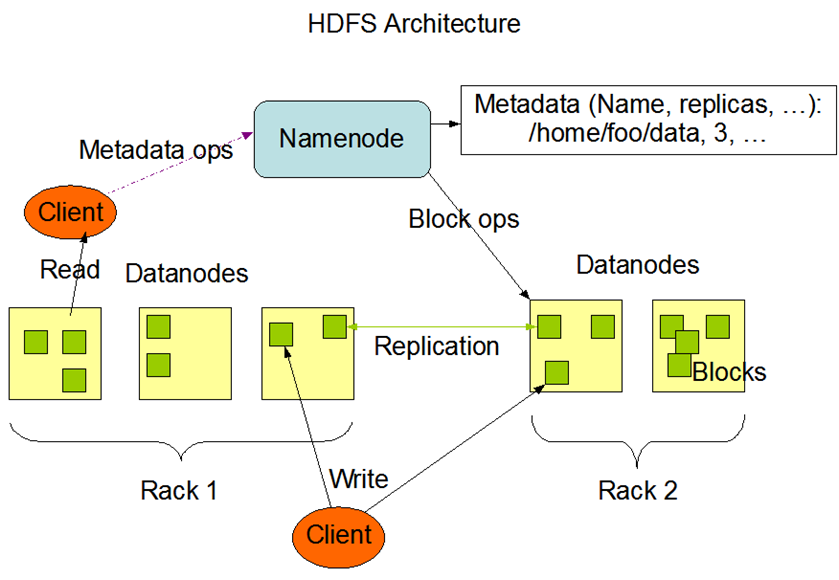
**DataXceiverServer介绍**

datanode端是如何接受传来的数据文件呢？

在datanode类里，有一个线程类：

org.apache.hadoop.hdfs.server.datanode.DataXceiver。每当有client连接到datanode时，datanode会new一个DataXceiver，负责数据的传输工作。

在这个DataXceiver线程类里：



 相关代码：

@Override

public void writeBlock（）{

/\*\* A class that receives a block and writes to its own disk, meanwhile

 \* may copies it to another site. If a throttler is provided,

 \* streaming throttling is also supported.

 \*\*/

 BlockReceiver blockReceiver = null; // responsible for data handling

}

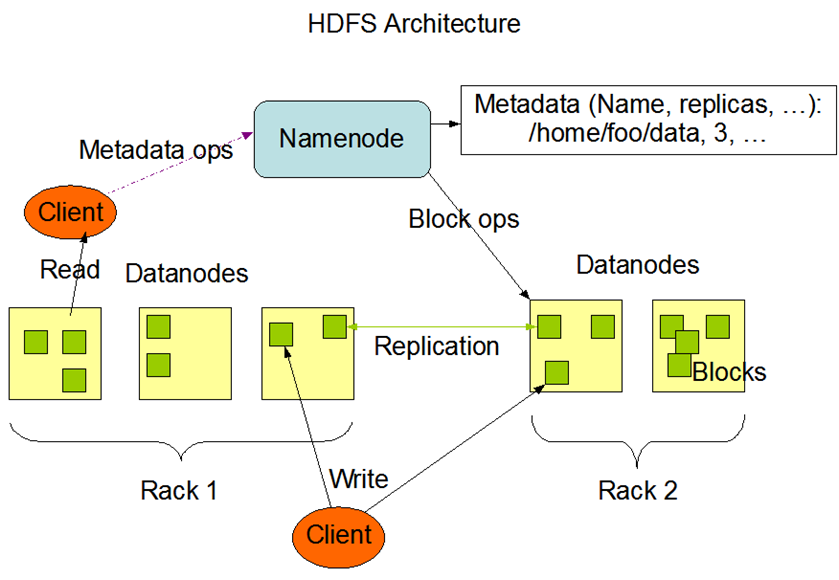
会调用 BlockReceiver 这个类，这个类是用于接收数据并写入本地磁盘，还负责将数据传输给管道里下一个datanode节点。

**DatanodeProtocol介绍**

\* Protocol that a DFS datanode uses to communicate with the NameNode.

 \* It's used to upload current load information and block reports.

这个类是datanode和namenode  实现RPC通信的接口协议，作用是datanode通过RPC机制连接namenode并汇报自身节点状态信息。在这个接口类里，一个非常重要的方法是：



 相关代码：

public HeartbeatResponse sendHeartbeat（）{

}

这个是datanode向namenode发送心跳的方法，datanode周期性通过RPC调用sendHeartbeat向namenode汇报自身的状态。

**三、NameNode相关体系**

**Namenode**

**|-------FSNameSystem**

**|-------DFSConfigKeys**

**|-------FSDirectory**

**|-------FsImage**

**|-------LeaseManager**

**|-------HeartbeatManager**

**|-------HeartbeatThread**

**|------Monitor**

**Namenode介绍**

\* NameNode serves as both directory namespace manager and

 \* "inode table" for the Hadoop DFS.  There is a single NameNode

 \* running in any DFS deployment.  (Well, except when there

 \* is a second backup/failover NameNode, or when using federated NameNodes.)

 \*

 \* The NameNode controls two critical tables:

 \*   1)  filename->blocksequence (namespace)

 \*   2)  block->machinelist ("inodes")

 \*

 \* The first table is stored on disk and is very precious.

 \* The second table is rebuilt every time the NameNode comes up.

 \*

 \* 'NameNode' refers to both this class as well as the 'NameNode server'.

 \* The 'FSNamesystem' class actually performs most of the filesystem

 \* management.

Namenode类是Hadoop DFS文件系统的管理者类，用来管理HDFS的元数据信息。实际控制的是两张表信息，分别是：

1.文件——文件块信息

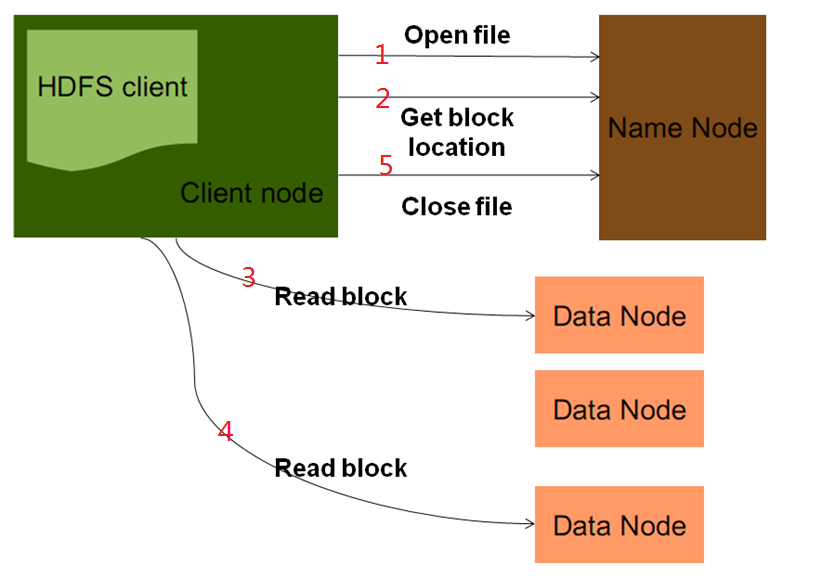
2.文件块信息——存储这个文件块的机器列表信息

第一张表信息存储在namenode节点的磁盘上，并且访问是非常高效的（因为namenode在启动之后，会将数据加载到内存里供快速访问）

第二张表信息，在每次namenode重启工作后，会重新建立。（**这么做目的是为了确保块信息存储的准备性，实现机制时，当namenode重启工作后，每个datanode节点通过rpc心跳向namenode汇报自身存储的文件块信息，然后namenode根据这些信息，重建第二张表信息，在此过程中，HDFS是处于安全模式的，即只能对外提供读服务。当第二表信息重建完后，确认文件块数量正确且都完整，则退出安全模式**）

实际上，namenode更多的是充当一个领导角色或更像是一个协议类，它定义了需要做哪些事，而真正干活的是FSNamesystem这个类。

在这个类里，format()这个方法我们应该会很熟悉，这是namenode的格式化方法



**format()方法：**

/\*\* Format a new filesystem.  Destroys any filesystem that may already

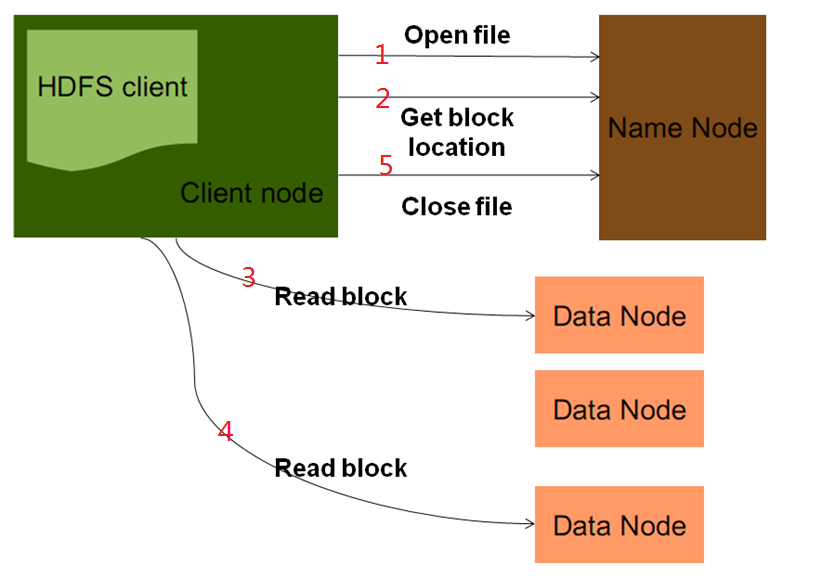
   \* exist at this location.  \*\*/

  public static void format(Configuration conf) throws IOException {

    format(conf, true, true);

  }

格式化方法的定义：生成一个全新的文件系统（DFS文件系统）。并且摧毁已经存在的旧数据信息。



**format()方法源码骨架：**

 private static boolean format(Configuration conf, boolean force boolean isInteractive) throws IOException {

**//调用Fsnamesystem,获取用户配置的元数据信息存放路径。如果不配置，则默认使用linux的/tmp/hadoop/dfs/name这个目录。**

**//但是这个目录是临时目录，非常危险，所以需要更改。**

     Collection<URI> nameDirsToFormat = FSNamesystem.getNamespaceDirs(conf);

     List<URI> editDirsToFormat =  FSNamesystem.getNamespaceEditsDirs(conf);

**//然后调用FSImage类，在对应的目录下，创建新的Fsimage文件和Edits文件**

    FSImage fsImage = new FSImage(conf, nameDirsToFormat, editDirsToFormat);

    fsImage.format(fsn, clusterId);

  }

**FSNameSystem介绍**

FSNameSystem是HDFS文件系统实际执行的核心，提供各种对文件的管理和操作

**FSdirectory介绍**

FSDirectory存储整个文件系统的**目录状态**，保存了文件路径和数据块的映射关系。

**INodeFile和INodeDirectory**

分别表示文件节点和目录节点。

**FSimage介绍**

把文件和目录的元数据信息持久化地存储到fsimage文件中，每次启动时从中将元数据加载到内存中构建目录结构树，之后的操作记录在edits 中

定期将edits与fsimage合并刷到fsimage中

loadFSImage(File curFile)用于从fsimage中读入Namenode持久化的信息。

**HeartbeatManager介绍**

\* Manage the heartbeats received from datanodes.

 \* The datanode list and statistics are synchronized

 \* by the heartbeat manager lock.

namenode通过这个类来管理各个datanode传来的心跳。具体工作的线程是 HeartbeatThread这个线程类

**Monitor介绍**

这是HeartbeatManager类里一个私有线程类，这个类的作用是周期性检测各个Datanode的心跳，

在这个线程类里run方法里，有一个方法比较重要，就是 **heartbeatCheck();**这个方法的作用：

   \* Check if there are any expired heartbeats, and if so,

   \* whether any blocks have to be re-replicated.

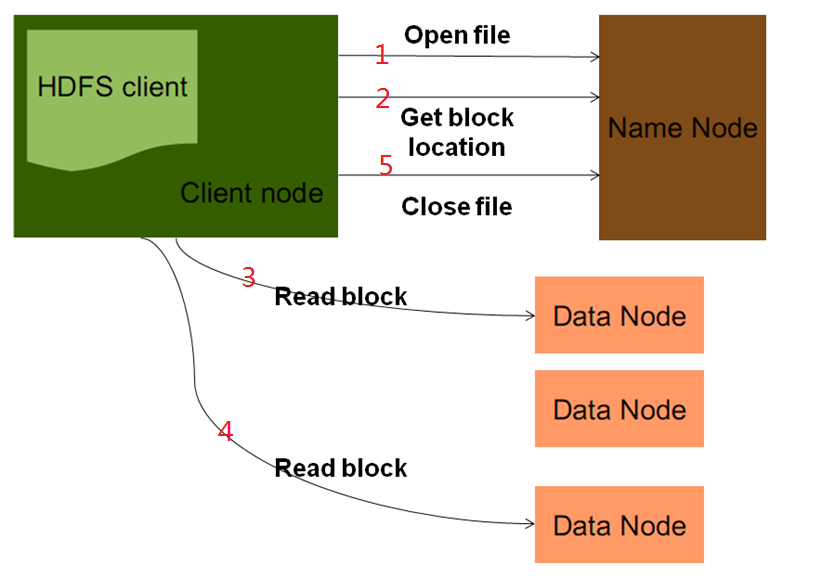
   \* While removing dead datanodes, make sure that only one datanode is marked

   \* dead at a time within the synchronized section. Otherwise, a cascading

   \* effect causes more datanodes to be declared dead.

检查是否有超时的datanode心跳。如果有的话，先判断下在这个死亡节点上是否有数据块需要进行复制。然后对这个datanode进行死亡标记。如果这个节点上有文件需要复制备份，则进行数据备份（满足一个block在HDFS上存3份），复制完后然后再删除这个死掉的datanode节点。从slaves列表中移除

此外，在进行死亡节点数据复制过程中，HDFS对外**不提供写服务**，即客户端此时是不能上传文件到HDFS系统上的。直到数据复制完成（比如一个block达到3份），最后删除此死亡节点后，才对外提供写服务。此过程中，不影响HDFS的查询文件操作



 相关代码：

private class Monitor implements Runnable {

    @Override

    public void run() {

**heartbeatCheck();**

              }

  }

HDFS的租约机制

HDFS的有个内部机制：不允许客户端的并行写。指的是同一时刻内，不允许多个客户端向一个HDFS上写数据。

所以要实现以上的机制，实现思路就是用互斥锁，但是如果底层要是用简单的互斥锁，可能有与网络问题，造成客户端不释放锁，而造成死锁。所以Hadoop为了避免这种情况产生，引入租约机制。

租约锁本质上就是一个带有租期的互斥锁。

Hadoop的思想来自于Google的论文，3.1

Hadoop 租约锁对应的类：org.apache.hadoop.hdfs.server.namenode.LeaseManager.Lease

还有一个租约锁管理者：

org.apache.hadoop.hdfs.server.namenode.LeaseManager

建议掌握租约这种思想

 HDFS特点总结

**HDFS特点**

1、分布式存储架构，支持海量数据存储。（GB、TB、PB级别数据）

2、高容错性，数据块拥有多个副本（副本冗余机制）。副本丢失后，自动恢复。

3、低成本部署，Hadoop可构建在廉价的服务器上。

4、能够检测和快速应对硬件故障，通过RPC心跳机制来实现。

5、简化的一致性模型，这里指的是用户在使用HDFS时，所有关于文件相关的操作，比如文件切块、块的复制、块的存储等细节并不需要去关注，所有的工作都已被框架封装完毕。用户所需要的做的仅仅是将数据上传到HDFS。这大大简化了分布式文件存储操作的难度和管理的复杂度。

6、**HDFS不能做到低延迟的数据访问（毫秒级内给出响应）**。但是Hadoop的优势在于它的高吞吐率（吞吐率指的是：单位时间内产生的数据流）。可以说HDFS的设计是牺牲了低延迟的数据访问，而获取的是数据的高吞吐率。如果要想获取低延迟的数据访问，可以通过Hbase框架来实现。

7、HDFS不许修改数据，所以适用场景是：一次写入，多次读取（once-write-many-read）。注意：HDFS允许追加数据，但不允许修改数据。追加和修改的意义是不同的。

8、HDFS不支持并发写入，一个文件同一个时间只能有一个写入者。

9、HDFS不适合存储海量小文件，因为会浪费namenode服务节点的**内存空间**

**HDFS的租约机制**

HDFS的有个内部机制:不允许客户端的并行写,值得是同一时刻内,不允许多个客户端向一个HDFS上写数据..

Hadoop 的思想来自于Google的论文,3.1.Hadoop租约

**纠删码技术(Erasure Codes)**

是一种数据恢复技术,使用这种技术可以提高磁盘的利用率.

实现数据存储的可靠性

纠删码技术（Erasure Codes ）

2018年8月25日

16:18

**概述**

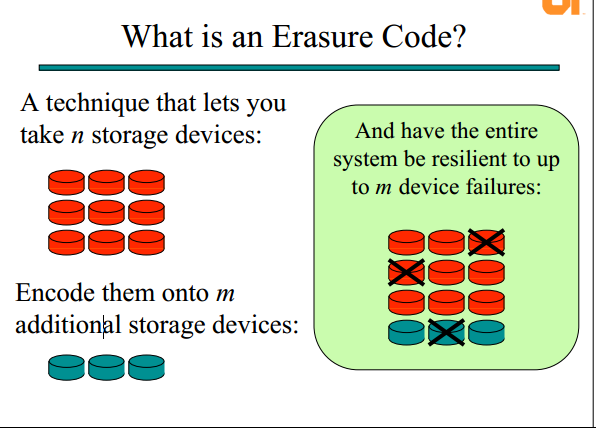
纠删码技术是一种数据恢复技术，使用这种技术可以提高磁盘的利用率。

在存储系统领域，实现数据存储的可靠性的途径：

①副本冗余机制，比如HDFS的3副本策略，磁盘利用率是1/3

②使用纠删码技术，引入数据校验块，通过编码过程让数据校验块和原始数据产生关联关系。则当数据块丢失时，可以进行恢复。

下图，示意使用纠删码，磁盘利用率 ：9/12=75%



三个原始数据块

X=1

Y=2

Z=3

三个数据校验块

X+Y+Z=6

2X+3Y+Z=11

X+2Y+3Z=14

里所码（Reed-Solomon )

**概述**

里所码算法分两个过程：

①编码过程（将原始数据块和数据校验块编码）

②解码过程（数据恢复）

具体看论文

优点：可以提供磁盘利用率，理论上来说，最大接近：n/n+m~100%

磁盘利用率越高，可靠性越低。

缺点：

1.会产生大量的计算量，占用很大带宽

2.数据恢复的代价高，因为需要计算，所以恢复效率较低

3.如果数据块发生了变化，会触发重新的编码。

如果使用纠删码（里所码）技术应用于存储系统，适用在冷数据的数据容灾。

**里所码(Reed-Solomon)**

分为两个过程

优点:

可以提高磁盘利用率,理论上来说,最大接近:n/n+m~100%

磁盘利用率越高,可靠性越低

缺点:

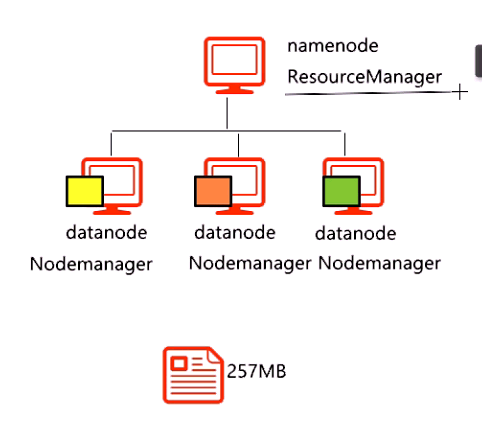
1.会产生大量的计算量,占用很大带宽

2.数据恢复的代价高,因为需要计算,所以恢复效率较低

3.如果数据块发生了变化,

## big\_day08

Hadoop03



MapReduce介绍

**概述**

MapReduce是一个**分布式的计算框架**（编程模型）MR框架对于程序员的最大意义在于，不需要掌握分布式计算编程，不需要考虑分布式编程里可能存在的种种难题，比如任务调度和分配、文件逻辑切块、位置追溯、工作。这样，程序员能够把大部分精力放在核心业务层面上，**大大简化了分布式程序的开发和调试周期**。

**JobTracker / ResourceManager：** 任务调度者，管理多个TaskTracker。ResourceManager是hadoop2.0版本之后引入了yarn，有yarn来管理hadoop之后，jobtracker就被替换成了ResourceManager

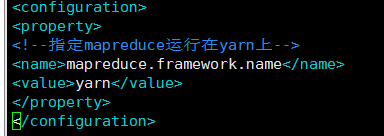
**TaskTracker / NodeManager**：任务执行者

**相关配置**

**1.修改 mapred-site.xml**

**执行：cp mapred-site.xml.template mapred-site.xml**

**配置如下：**



yarn是资源协调工具，

**2.修改yarn-site.xml**

**配置如下：**



**补充：**

在实际工作中，namenode节点和ResourceManager节点不在同一台服务器上。

此外，在slaves文件中，配置的datanode和nodemanger的服务列表，即dn和nm这个两个进程在同一台服务器上。

启动指令：start-yarn.sh

如果要启动HDFS+MapReduce: start-all.sh

MapReduce实现WordCount

**MR入门代码示例**

先单独运行map任务，得到的结果文件：

hello 1

hello 1

hello 1

world 1

1605 1

然后加入reduce任务，查看迭代器的内容

1605 1，

hello 1,1,1,

world 1,

**Map任务代码：**

/\*\*

 \*  HDFS负责存储数据，MR计算框架负责计算HDFS上的数据，比如/park/word.txt。即MR想要计算某个文件，这个文件必须先要在HDFS上进行存储。

 \*  MR计算文件，会对文件进行逻辑切块，每一块是一个InputSpilt，每有一个InputSplit，会启动一个Map任务。逻辑切块的大小是128MB。

 \*  现在假设word.txt正好是128MB，则会有一个InputSpilt》》有一个map任务》》分配给集群里某一个TaskTracker处理。

 \*  如果小于128MB，也是一个InputSpilt》》有一个map任务》》分配给集群里某一个TaskTracker处理。

 \*  如果是129MB，两个InpuSpilt》》两个map任务》》此时根据集群里TaskTracker数量及状态分配map任务，这个工作MR框架会做的。不需要程序员来考虑。

 \*  此外，因为涉及到了逻辑切块，所以涉及到：要处理的文件在HDFS的位置信息（在哪台datanode节点上）、计算start、end位置信息以及位置追溯等问题，统统都交给MR框架来做。不需要考虑。

 \*

 \***MapReudce计算框架抽象出两个任务阶段，Map任务阶段和Reduce任务阶段。**

 \*  Map任务阶段的目标：从文件里，按行读取数据。**一行一行**的进行处理。

 \*  **如何利用MR框架编写Map阶段代码**？

 \*  1.写一个类，继承这个类：org.apache.hadoop.mapreduce.Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>

 \*  2.定义Mapper的4个泛型类型。

 \*  3.重写map()方法，在map方法里编写map任务代码

 \*

 \*  map函数()参数说明：

 \*  之前曾提到：一个InputSpilt会对应一个Map任务，在InputSpilt里，实际就包含着待处理的文件数据。比如现在要处理的word.txt数据如下：

 \*  hello world

 \*  hello hadoop

 \*  hello 1605

 \*

 \*  然后MR框架会按行进行处理，并形成一个key/value对。默认的情况下，key是一个长整型，表示行的行首的偏移量。value是一个文本类型，代表每一行数据。

 \*

 \***key：每一行行首的偏移量（用处不大）**

**\*  value：每一行数据（主要用的是这个）**

 \*  context:我们利用context 输出我们自定义的输出map<E,V>

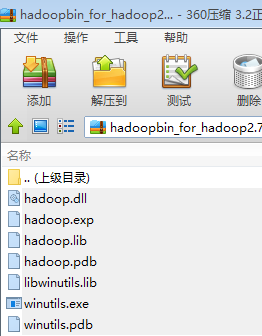
 \*

 \*  最后需要牢记的是：**每有一行数据，就会调用一次map()方法。**

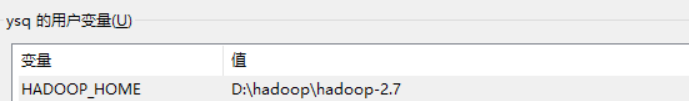
 \*/

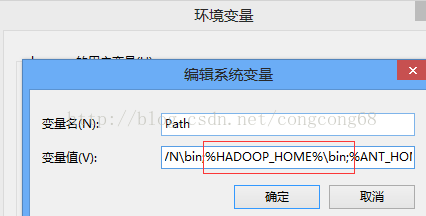
 通过插件实现WordCount

1.将压缩包中的所有内容复制到hadoop的安装目录的bin目录下，以及C盘 windows的System32目录下



8.然后配置变量环境HADOOP\_HOME 和path，如图所示：





3..在hadoop bin目录下，双击执行winutils.exe

4.重启eclipse，然后测试插件是否可用

5.如果不打印日志，引入log4j.properties文件即可

MapReduce入门知识点

2018年8月27日

14:27

**Mapper组件知识点**

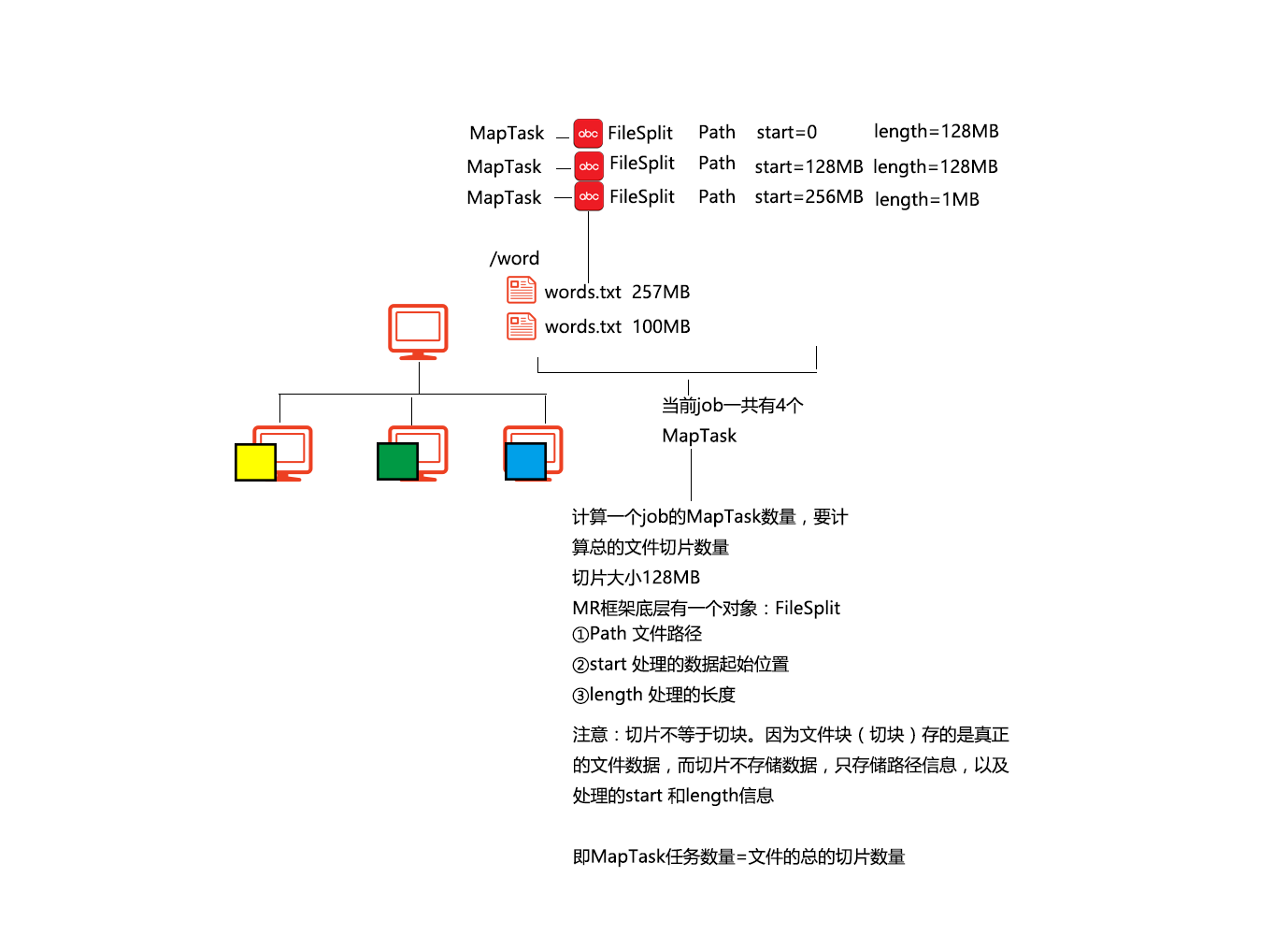
1.Mapper组件开发方式：写一个类，继承Mapper

2.Mapper组件的作用是定义 每一个MapTask具体要怎么处理数据。

比如一个文件，257MB，会生成3个MapTask。即三个MapTask处理逻辑是一样的

只是每个MapTask处理的数据不一样。

**针对2的补充:**



 3.第一个泛型类型：LongWritable，对应的Mapper的输入key。

输入key是每行的行首偏移量

4.第二个泛型类型：Text，对应的Mapper的输入Value。

输入value是每行的内容

5.第三个泛型类型：对应的Mapper的输出key，这是程序员根据业务来定义

6.第四个泛型类型：对应的Mapper的输出value，这是程序员根据业务来定义

7.注意：初学时，第一个和第二个泛型写死。第三个和第四个不固定

8.Writable机制是Hadoop自身的序列化机制，常用的类型：

①LongWritable ②Text(String) ③IntWritable ④NullWritable

9.定义MapTask的任务逻辑是通过重写map()方法来实现的。

读取一行数据就会调用一次此方法，同时会把输入key和输入value传给程序员

10.在实际开发中，最重要的是拿到输入value(每行内容)

11.输出方法：通过context.write(输出key，输出value)

12.开发一个MapReduce程序（job），Mapper可以单独存储，此时，最后的输出的结果文件内容就是Mapper的输出。

13.Reducer组件不能单独存在，因为Redcuer要依赖于Mapper的输出。当引入了Reducer之后，最后输出的结果文件的结果就是Reducer的输出。

**Reduce组件知识点**

1.reduce组件用于接收mapper组件的输出

2.redudce第一个泛型类型是reduce的输入key,需要和mapper的输出key类型一致

3.第二个泛型类型是reduce的输入value，需要和mapper的输出value类型一致

4.第三个泛型类型是reduce的输出key类型，根据具体业务决定

5.第四个泛型类型是reduce的输出value类型，根据具体业务决定

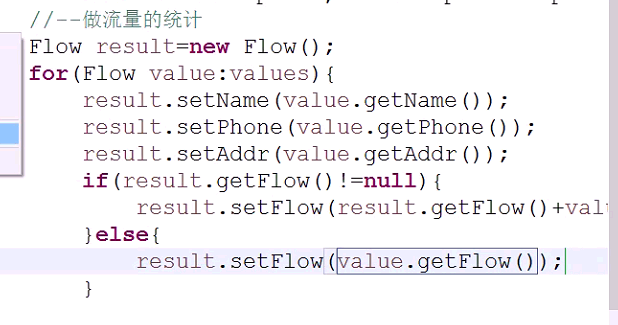
6.reduce收到map的输出，会按相同的key做聚合，形成:key Iterable 形式然后通过reduce方法传给程序员。

7.reduce方法中的Iterable是一次性的，即遍历一次之后，再遍历，里面就没有数据了。所以，在某些业务场景，会涉及到多次操作此迭代器，处理的方法是：①先创建一个List  ②把Iterable装到List ③多次去使用List即可

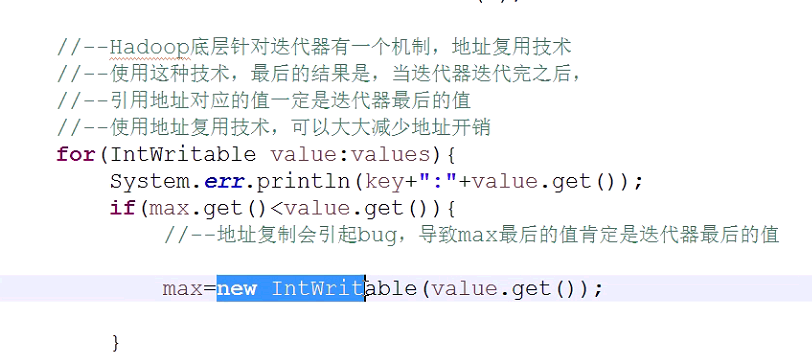
## big\_day09

Hadoop04

如果Flow的属性是Integer修饰,则,result.getFlow可能为null,报错空指针异常



max案例



**序列化机制+FlowCount案例**

由于集群工作过程中，需要用到RPC操作，所以MR处理的对象必须可以进行序列化/反序列操作。Hadoop利用的是avro实现的序列化和反序列，并且在其基础上提供了便捷的API

要序列化的对象必要实现相关的接口：

Writable接口--WritableComparable

**Hadoop提供的Writable类**

LongWritable

IntWritable

Text

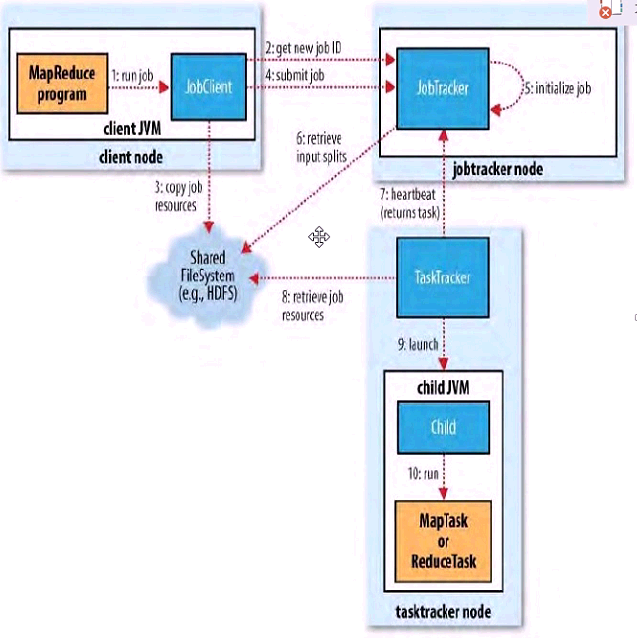
NullWritbale

ByteWritable

DoubleWritabl

**以上这些类型，之所以能够被序列化，就是因为都实现了Writable接口**

job任务执行流程



1.run job阶段。此阶段发生在客户端进程中，底层有一个JobClient类在工作，JobClient在这个阶段的职责是：

①收集整个job的环境信息（比如通过conf设定的参数，还有mapperClass,reducerClass,以及输出kv类型）

②会计算当前job的切片数量（切片不同等切块，用FileSplit：path  start length）

③检测环境信息的合法性，以及输入和输出的路劲合法性。

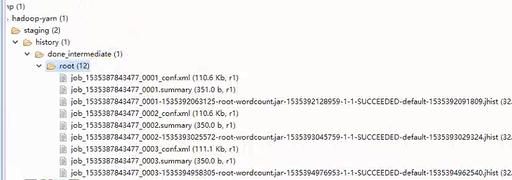
比如输出的kv类型不匹配，输入路径不存在，输出的结果目录已存在，以上都是不合法的，所以就会直接报错返回，不会再继续下一步了。

2.如果第一步的检测通过之后，会去找JobTracker，为当前的job申请jobid，用于标识job。jobid是全局唯一的，目的是管理job，因为整个集群同一时间内可能跑多个job。

3.JobClient收到jobid，就将此job的运算资源（①conf.xml ②summary ③jar 包）

提交到HDFS上，目录路径：eclipse中的/tmp/hadoop-yarn/history/done\_intermediate/root

conf.xml：存储的是job的环境配置信息



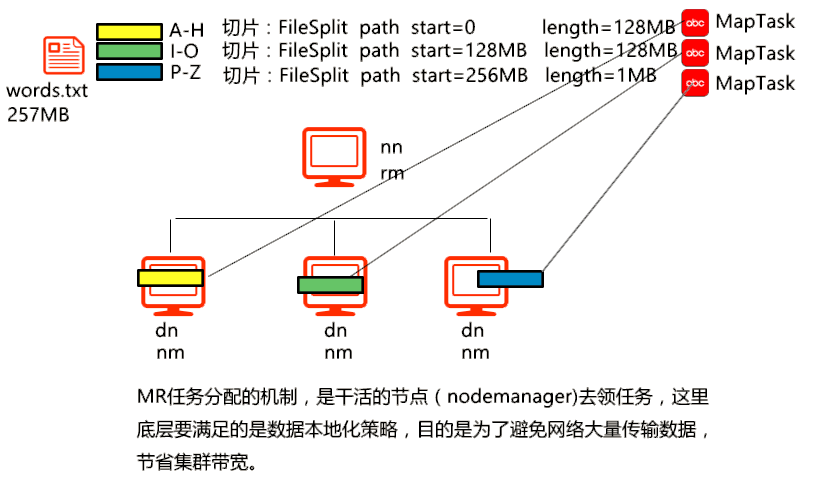
summary内记录的是:jobid，mapTask数量和reduceTask数

jar包:程序员写的代码

4.JobClient 做submit job动作，底层是把第三步的job的资源路径信息告知给JobTracker。

5.6 去HDFS上拿取job的运算资源，然后做job的初始化，查看配置信息，以及拿到job的切片数量（本地目的是获取有几个mapTask）

7.任务的领取，底层要满足数据本地化策略，节省集群的带宽



**补充：**因为MapTask读取文件是按行读取，所以必须要保证读取的是完整一行。底层会发生一个**位置追溯**的动作，此过程不可避免的会发生网络数据的传输，但数据量很小。

8.去HDFS获取job 的运算资源（主要是jar包），然后结合代码来处理数据了。

这里体现了Hadoop的思想：移动的是运算，而不是数据。目的也是节省集群带宽。

9.10启动JVM进程,执行MapTask或ReduceTask。

**补充：**MapTask任务的数量=job的切片数量

ReduceTask任务数量，默认就一个。

ReduceTask数+分区机制

**概述**

1.一个Job的ReduceTask数量，默认就1个。可以通过代码来设置 在driver类中添加代码:

**//--设置reduceTask任务数量，默认1个**

**job.setNumReduceTasks(3); //flow案例中的driver类**

//--设置自定义的分区器，默认用的是HashPartitioner

//--HashPartitioner的作用是按Mapper输出key的hashcode值分区的

//--注意：记住一个顺序：先分区，再按相同的key做聚合

/\*\*

\* 第一个泛型类型：Mapper的输出key

\* 第二个泛型类型：Mapper的输出Value

\*/

public class FlowPartitioner extends **Partitione**r<Text, Flow>{

/\*

\* getPartion方法用于接收Mapper的输出key和输出value值

\* 此外，也会将分区数量（reduceTask)传过来

\* 分区默认是有编号。比如有3个分区，编号：0,1,2

\*/

@Override

public int **getPartition(**Text key, Flow value, int numPartitions) {

if(value.getAddr().equals("bj")){

return 0;}

if(value.getAddr().equals("sh")){

return 1; }

if(value.getAddr().equals("sz")){

return 2;

}else{

//--以上条件都不符合，就发给分区0

return 0; } } }

**自定义分区案例：**

**FlowCountPartition代码：**

/\*\*

 \* 想要实现自定义分区，需要继承Partitioner类，并重写getPartition（）方法。

 \* 根据用户自定义的分区，把输出map的value值（本例中是一个一个的FlowBean对象），根据用户定义的条件放到对应的分区里。

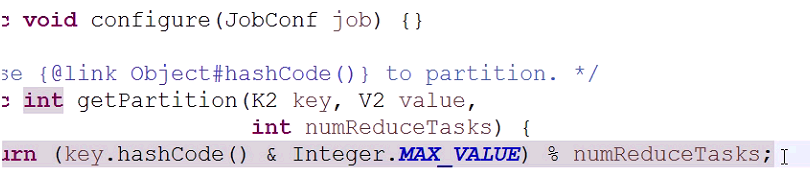
 \* 分完区之后，在每个区里，会按照输出map的key值整理成如下形式，然后发送给reduce端：

 \* 然后reduce端拿到数据后，根据用户的自定义代码进行每个区里的数据合并

2.习惯上，把reduceTask叫做分区，即有几个reduceTask，就有几个分区。

3.Hadoop底层有一个默认的分区器（HashPartitioner），此分区器的作用可以确保相同的Mapper输出key落到同一个分区（reduceTask)里。

算法： return (key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks;



4.最后的结果文件数量=分区（reduceTask）数量，即每个结果文件存储的是对应分区的结果数据。

5.因为底层用的是简单hash算法，所以会产生数据倾斜，有时会产生某个结果文件数据很少或没有的情况。

6.如果最后想多个结果文件的结果合并在一起，

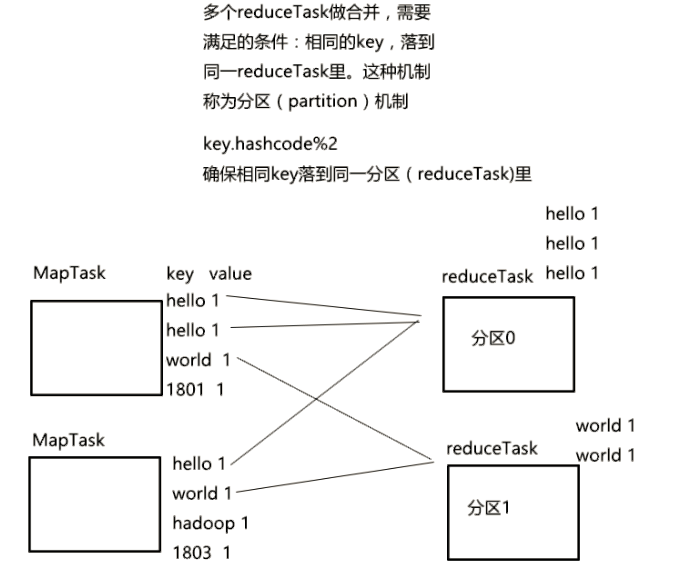
可以通过指令：hadoop  fs  -getmerge  /word/result    ./result.txt。

**movie的**

 //--MR框架会将Mapper的输出key做排序

//--排序的规则是由Key类型在实现WritableComparable接口时的**CompareTo()**决定的

//--注意：框架底层没有按Mapper输出value排序的机制，**只能按输出key排序**



分区案例+自定义分区案例

**分区案例：**

计算每月的利润

1  280

1  560

2  234

2  264

3  873

3  2323

要求：用三个分区来存放三个月的总利润

**实现思路**

输出map的key值为月份数。然后在Driver里设置分区数量（reduceTask数量）为3。Hadoop会根据key的hashcode值来进行分区。

**job.setPartitionerClass(FlowCountPartition.class);**

**job.setNumReduceTasks(4);**

案例——全排序

**需求说明**

有这样一组数字，要求利用3个reduce来处理，并且生成的三个结果文件，是整体有序的。

**源数据**：

82 239 231

23 22 213

123 232 124

213 3434 232

4546 565 123

231 231

2334 231

1123 5656 657

12313 4324 213

123 2 232 32

343 123 4535

12321 3442 453

1233 342 453

1231 322 452

232 343 455

3123 3434 3242

**生成的三个结果文件：**

|  |  |  |
| --- | --- | --- |
| 文件1 | 文件2 | 文件3 |
| 2        1  22        1  23        1  32        1  82        1 | 123        4  124        1  213        3  231        4  232        4  239        1  322        1  342        1  343        2  452        1  453        2  455        1  565        1  657        1 | 1123        1  1231        1  1233        1  2334        1  3123        1  3242        1  3434        2  3442        1  4324        1  4535        1  4546        1  5656        1  12313        1  12321        1 |

**实现思路**

如果是一个reduce，那肯定就是全排序。但如果只单纯指定3个reduce数量，Hadoop会默认根据Key的hash进行分区，这样不会出现三个结果文件是全排序的情况。所以，需要自定义分区，在自定义分区里，根据数据采样边界来对数据进行分区。

**自定义分区代码：**

public class AuthPartitioner extends Partitioner<IntWritable, IntWritable>{

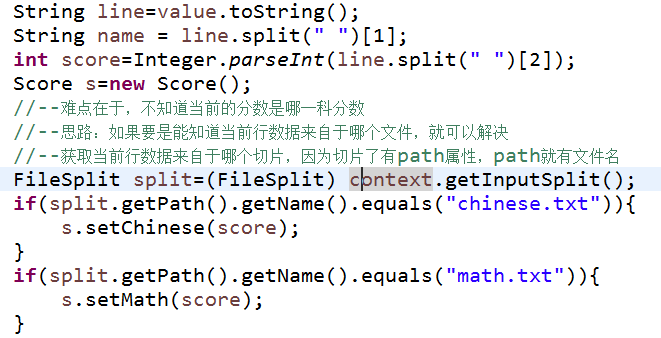
if(key.get()<100){

return 0; }

if(key.get()<1000){

return 1; }

else{return 2; }

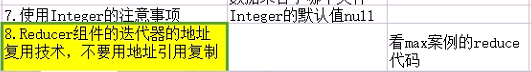


知识点与案例:

|  |  |  |
| --- | --- | --- |
| wordcount |  |  |
| average | 求平均值 |  |
| distinct | 数据去重 | 192.168.234.21 |
| flow | javabean |  |
| max | 年份,最高气温 | 2329999919500515070000 |
| yesun | 爷孙关系 |  |

|  |  |  |
| --- | --- | --- |
| max |  | 地址复用 |
| flow |  |  |
| profit |  | 多个mapper 多个driver处理 |
| movie |  |  |
| totalsort | 排序,分区 |  |
| score |  | 多个格式一致的文件一同处理 |





**切块与切片的区别**

切片:

有一个切片对应一个maptask,split是MapReduce里的概念，是切片的概念，split是逻辑切片 ；

用FileSplit：path  start length,MapTask任务的数量=job的切片数量

切块:

block是hdfs中切块的大小，block是物理切块；

## big\_day10

Hadoop05

潜在好友案例

电信流量案列

IMG_256

**Combiner合并**

合并的目的是减少Reduce端 迭代的次数

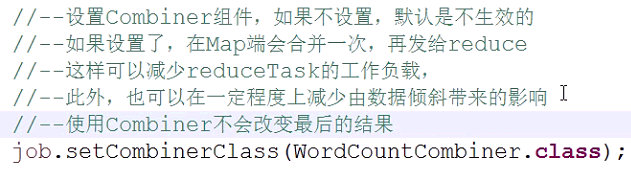
combiner是实现Mapper端进行key的归并，combiner具有类似本地的reduce功能。

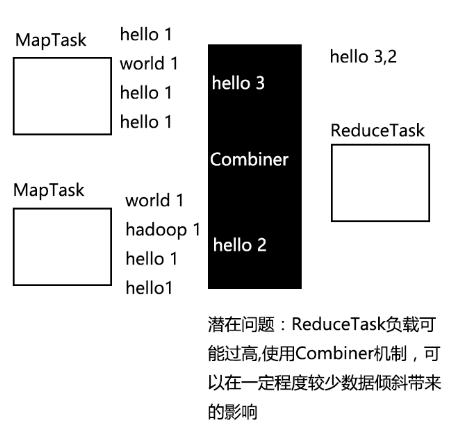
如果不用combiner,那么所有的结果都是reduce完成，效率会很低。使用combiner先做合

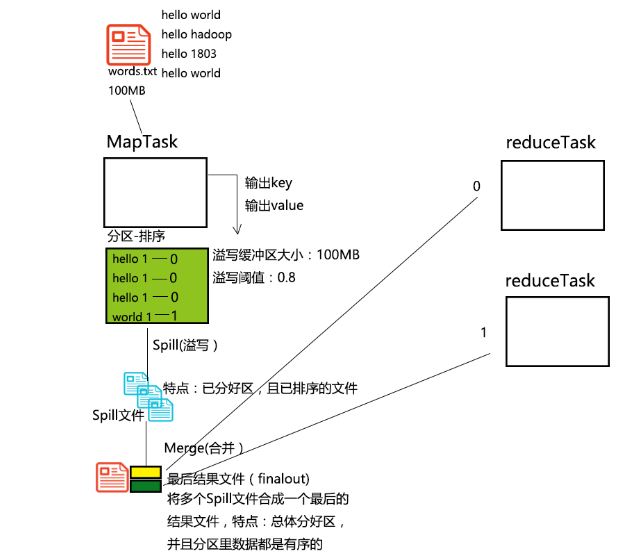
并，然后发往reduce。

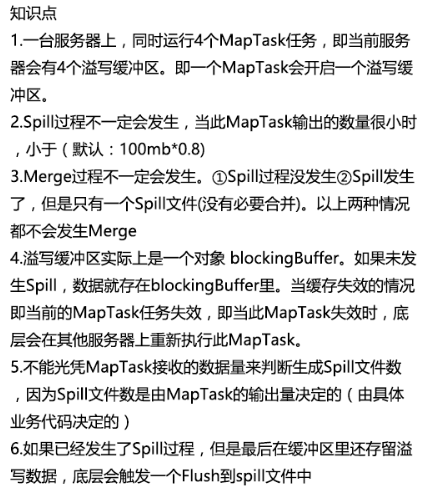
开发一个Combiner组件，就是继承Reducer，

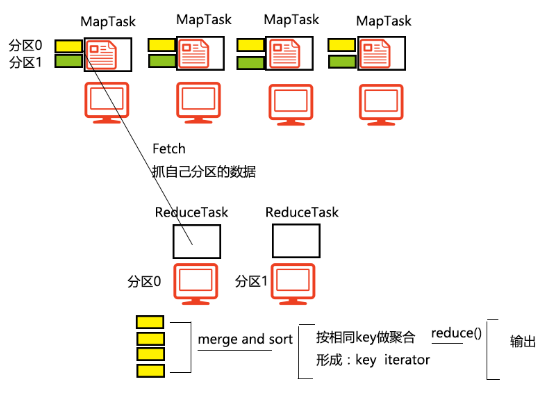
\* 因为此组件的作用就是接受map的输出，然后合并。合并之后，在发给reduce



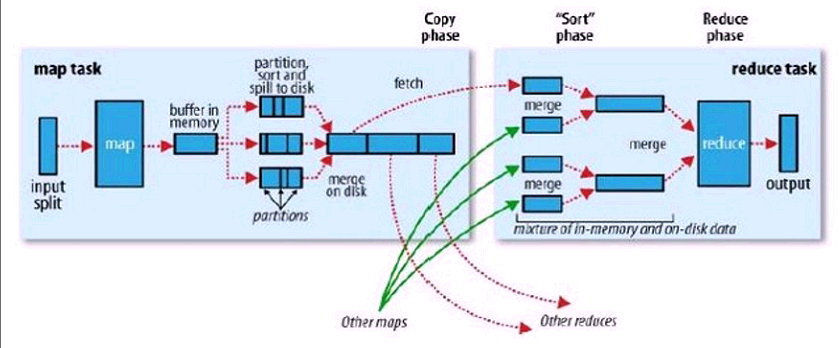


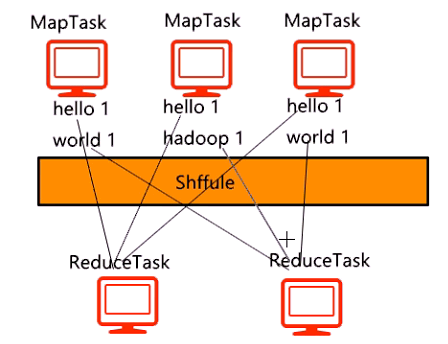






**shuffle 洗牌--mr框架的核心**





**hadoop2.0高可用集群搭建方案::**

Hadoop 的namenode好比人的心脏，非常重要，绝对不可以停止工作，在hadoop1,只有一

个namenode,如果该namenode数据丢失或停止工作，整个集群就不能恢复了。

hadoop2比hadoop1改进的地方：①高可用的解决方案

hadoop2中，有两个namenode(目前只支持两个,注意，不是secondarynamenode）。每

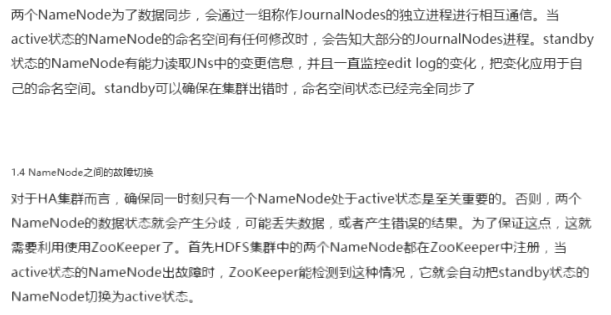
一个都具有相同的职能，一个是active状态，一个是standby待命状态，时刻同步active状态

的namenode数据，实现高可用。

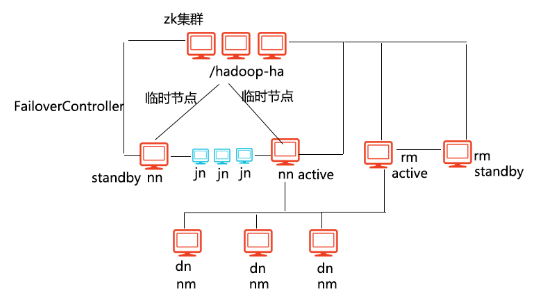
要想实现数据的实现共享，新HDFS采用了一种共享机制，Quorum Journal

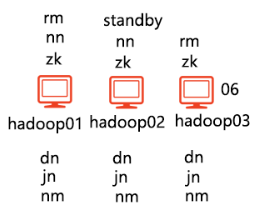
Node（JournalNode）集群。NFS是操作系统层面的，JournalNode是hadoop层面的（主

流做法）



scp -r zookeeper-3.4.7 176.129.8.21:/home/software/





vim /etc/sysconfig/network

hostname hadoop02

vim /etc/hosts

scp /etc/hosts hadoop03:/etc/

免秘钥登录

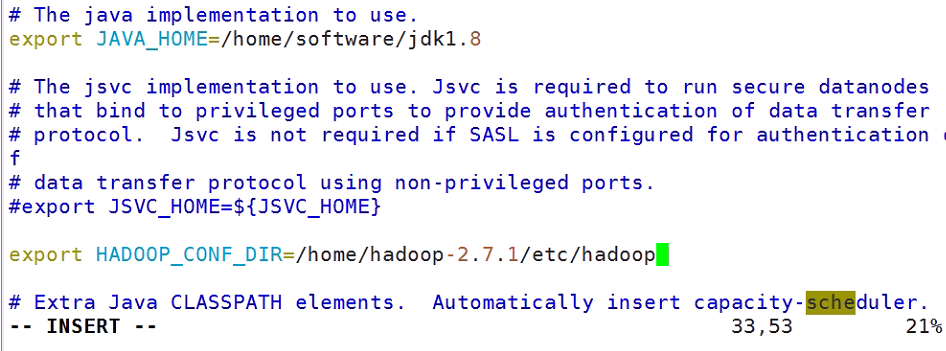
ssh-keygen

ssh-copy-id root@hadoop01/02/03

ssh hadoop02

logout

vim hadoop-env.sh



配置core-site.xml



vim hdfs-site.xml

<!--执行hdfs的nameservice为ns,和core-site.xml保持一致-->

<name>dfs.nameservices</name>

<value>ns</value>

<!--ns下有两个namenode,分别是nn1,nn2-->

<name>dfs.ha.namenodes.ns</name>

<value>nn1,nn2</value>

<!--nn1的RPC通信地址-->

<name>dfs.namenode.rpc-address.ns.nn1</name>

<value>hadoop01:9000</value>

<!--nn1的http通信地址-->

<name>dfs.namenode.http-address.ns.nn1</name>

<value>hadoop01:50070</value>

<!--nn2的RPC通信地址-->

<name>dfs.namenode.rpc-address.ns.nn2</name>

<value>hadoop02:9000</value>

<!--nn2的http通信地址-->

<name>dfs.namenode.http-address.ns.nn2</name>

<value>hadoop02:50070</value>

<!--指定namenode的元数据在JournalNode上的存放位置,这样，namenode2可以从jn集群里获

取

最新的namenode的信息，达到热备的效果-->

<name>dfs.namenode.shared.edits.dir</name>

<value>qjournal://hadoop04:8485;hadoop05:8485;hadoop06:8485/ns</value>

<!--指定JournalNode存放数据的位置-->

<name>dfs.journalnode.edits.dir</name>

<value>/home/software/hadoop-2.7.1/journal</value>

<!--开启namenode故障时自动切换-->

<name>dfs.ha.automatic-failover.enabled</name>

<value>true</value>

<!--配置切换的实现方式-->

<name>dfs.client.failover.proxy.provider.ns</name><value>

org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>

<!--配置隔离机制-->

<name>dfs.ha.fencing.methods</name>

<value>sshfence</value>

<!--配置隔离机制的ssh登录秘钥所在的位置-->

<name>dfs.ha.fencing.ssh.private-key-files</name>

<value>/root/.ssh/id\_rsa</value>

<!--配置namenode数据存放的位置,可以不配置，如果不配置，默认用的是

core-site.xml里配置的hadoop.tmp.dir的路径-->

<name>dfs.namenode.name.dir</name>

<value>file:///home/software/hadoop-2.7.1/tmp/namenode</value>

<!--配置datanode数据存放的位置,可以不配置，如果不配置，默认用的是

core-site.xml里配置的hadoop.tmp.dir的路径-->

<name>dfs.datanode.data.dir</name>

<value>file:///home/software/hadoop-2.7.1/tmp/datanode</value>

<!--配置block副本数量-->

<name>dfs.replication</name>

<value>3</value>

<!--设置hdfs的操作权限，false表示任何用户都可以在hdfs上操作文件-->

<name>dfs.permissions</name>

<value>false</value>

vim mapred-site.xml

<!--指定mapreduce运行在yarn上-->

<name>mapreduce.framework.name</name>

<value>yarn</value>

vim yarn-site.xml

<!-- 开启YARN HA -->

<name>yarn.resourcemanager.ha.enabled</name>

<value>true</value>

<!-- 指定两个resourcemanager的名称 -->

<name>yarn.resourcemanager.ha.rm-ids</name>

<value>rm1,rm2</value>

<!-- 配置rm1，rm2的主机 -->

<name>yarn.resourcemanager.hostname.rm1</name>

<value>hadoop01</value>

<name>yarn.resourcemanager.hostname.rm2</name>

<value>hadoop03</value>

<!--开启yarn恢复机制-->

<name>yarn.resourcemanager.recovery.enabled</name>

<value>true</value>

<!--执行rm恢复机制实现类-->

<name>yarn.resourcemanager.store.class</name>

<value>

org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore</value>

<!-- 配置zookeeper的地址 -->

<name>yarn.resourcemanager.zk-address</name>

<value>hadoop01:2181,hadoop02:2181,hadoop03:2181</value>

<description>For multiple zk services, separate them with comma</description>

<!-- 指定YARN HA的名称 -->

<name>yarn.resourcemanager.cluster-id</name>

<value>yarn-ha</value>

<!--指定yarn的老大 resoucemanager的地址-->

<name>yarn.resourcemanager.hostname</name>

<value>hadoop03</value>

<!--NodeManager获取数据的方式-->

<name>yarn.nodemanager.aux-services</name>

分区 hadoop第五天笔记 的第 25 页

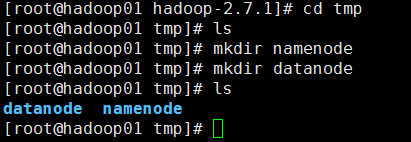
<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

vim slaves:

创建空新文件,

IMG_269



root@hadoop01 home]# scp -r hadoop-2.7.1 hadoop02:/home/

vim /etc/profile

第一台:hdfs zkfc -formatZK

Sucessfully m生成

hadoop-daemon.sh start journalnode 三台一起启动

启动JournalNode jps

第一台的进程

hadoop namenode -format

hadoop-daemon.sh start namenode

把第二台服务器变成standby

bin# hdfs namenode -bootstrapStandby

hadoop-daemon.sh start namenode

三台

hadoop-daemon.sh start datanode

第一二台:

hadoop-daemon.sh start zkfc

打开一二服务器:176.129.8.81:50070

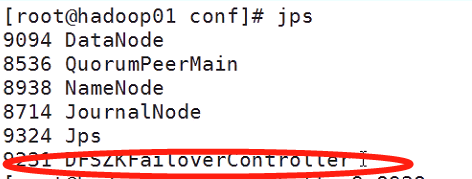


kill 第一台的进程

第二台active

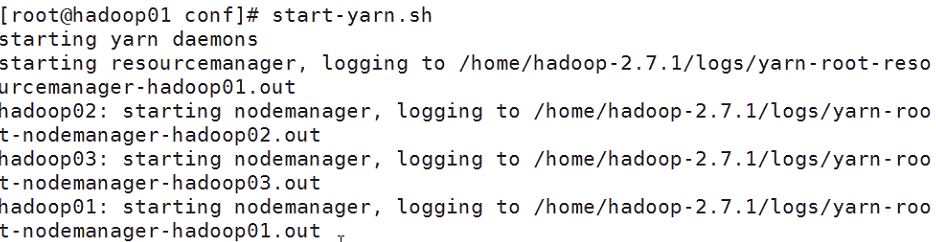
再次启动第一台变成standby

hadoop-daemon.sh start namenode



第一台:start-yarn.sh

]



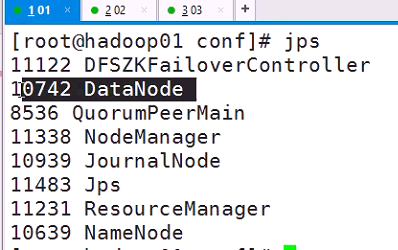
三台都多NodeManage

yarn-daemon.sh start resourcemanager

启动resourcemanager

stop-all.sh 三个一起停掉了

start0all.sh 启动所有集群

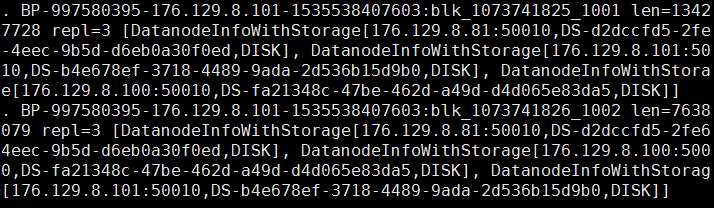


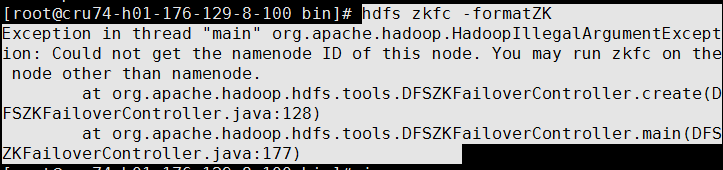
上传文件:

IMG_275

分别客户端,web查看文件命令

# hadoop fsck /hadoop-2.7.1\_64bit.tar.gz -files -blocks -locations





1 81 hadoop0

2 101 克隆 hadoop0

3 100 mysql hadoop03

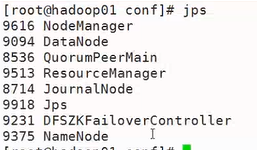
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ip 176.129.8 | zookeeper的myid | zookeeper节点 | hadoopnode节点 | 三个节点共有的 |  |  |
| 81 | 1 | QuorumPeerMain | follow | NodeManager,JournalNode,DataNode | NameNode,DFSZKFailoverController | ResourceManager |
| 101 | 2 | QuorumPeerMain | leader | NodeManager,JournalNode,DataNode | NameNode,DFSZKFailoverController |  |
| 02 | 3:observer | QuorumPeerMain | observer | NodeManager,JournalNode,DataNode |  |  |

NodeManage三个都没开启

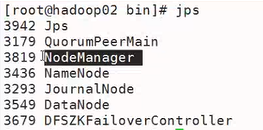
ResourceManage

第三台DataNode

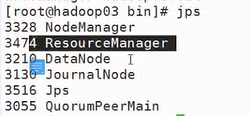
7



6



5



## big\_day11

Hadoop06

**shuffle**

提高性能的办法:

1.调大溢写缓冲区大小

溢写缓冲区大小:100MB

溢写缓冲区,溢写阈值:0.8

环写(环形)缓冲区

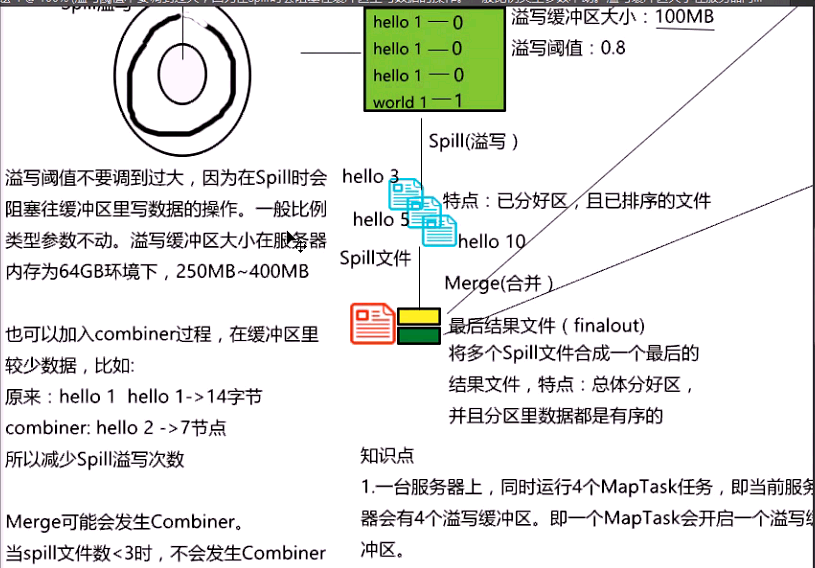
但是.溢写阈值不要过大,因为在spil时会阻塞往缓冲区里协数据的操作,一般比例类型参数不动.溢写缓冲区大小在64GB环境下,250MB-400MB

2.加入combiner的过程,在缓冲区里减少数据,减少溢写次数

Merge可能会发生Combiner,当spil文件数<3时,不会发生combiner,即spil>=3,merge才会发生combiner,merge的combiner会减少网络数据的传输,从而节省网络带宽

combiner在溢写缓冲区里面合并,不一定在merger中合并

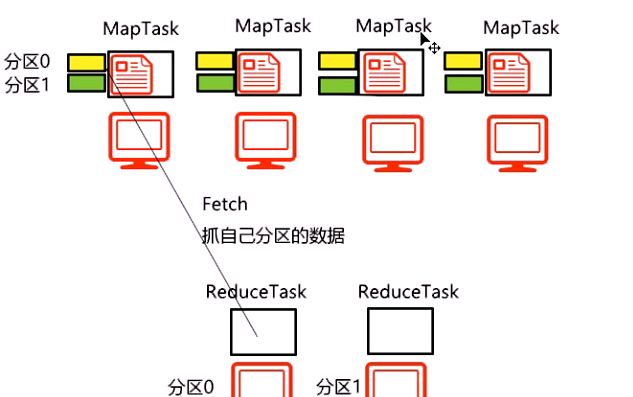
还可以通过配置让maptask进行最后压缩,减少网络传输,节省带宽.(但是解压缩费内存和cpu资源,所以要做一衡量),默认false.

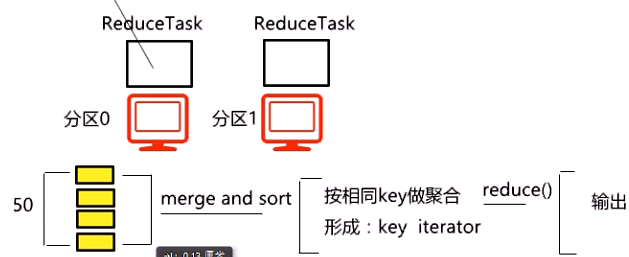


reducetask任务开启的标志是开始fetch(copy远程拷贝),

reducetask默认有一个启动阈值:5%,即如果有100个maptask,有5个完成,就开始工作(fetch).

fetch线程数可以调,默认是5个,一般调节策略,让线程数量等于或接近maptask数量,达到并行抓取的效果.



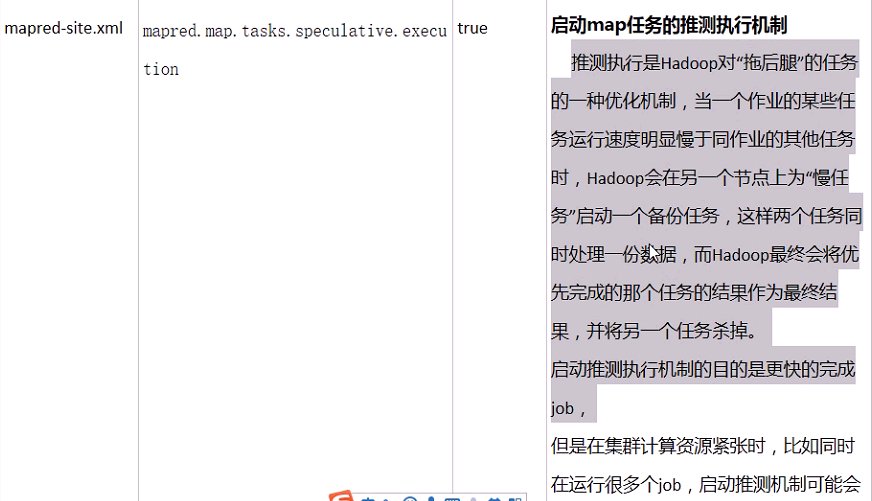


merge有个merge因子,默认为10;



如果是40个,先拿4个合成一个

**常见参数控制+优化策略**



缺点,是耗费CPU的计算资源,可以设为false

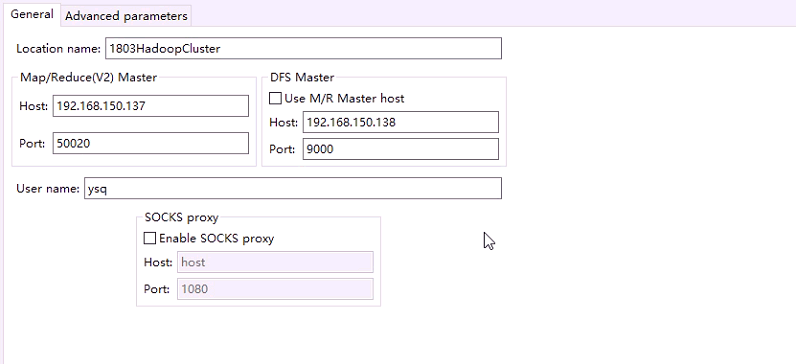
**core-site.xml配置文件详解--手册**

**源码**

JobConf

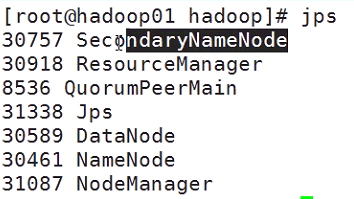


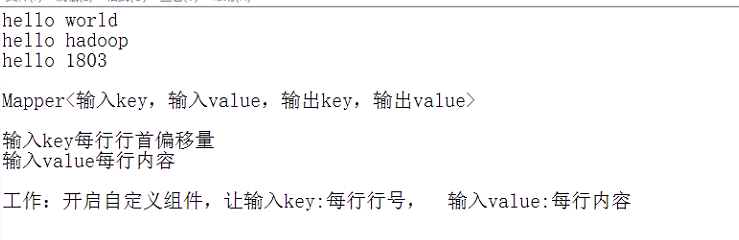
MapTask

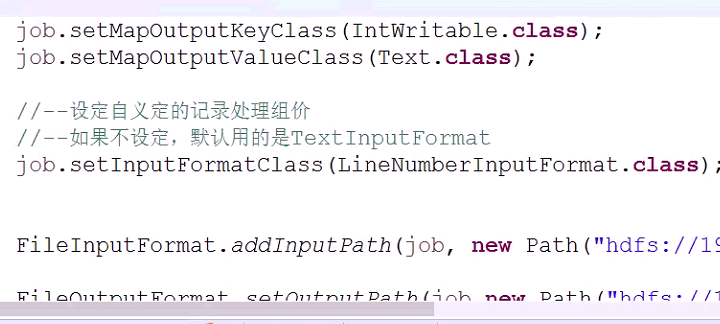


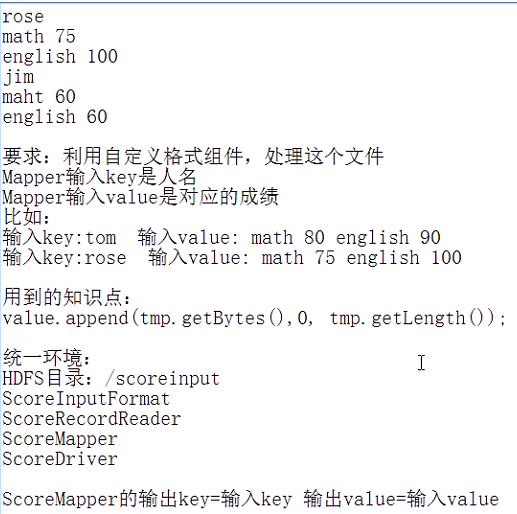
改为伪分布式

IMG_263

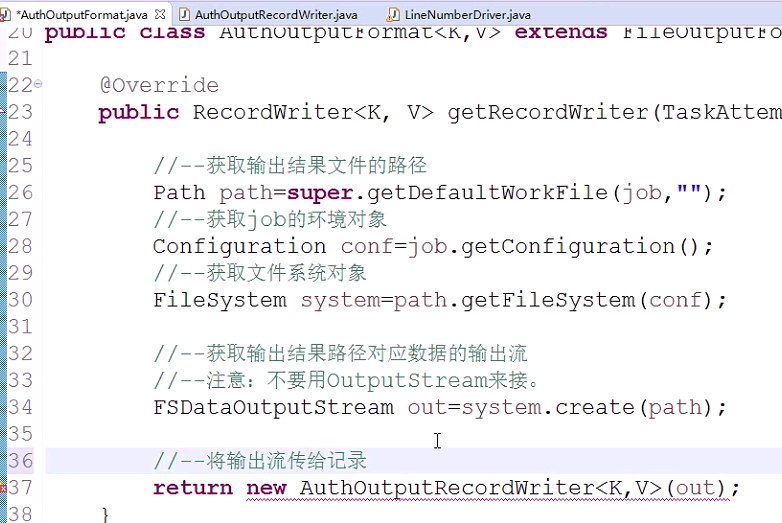


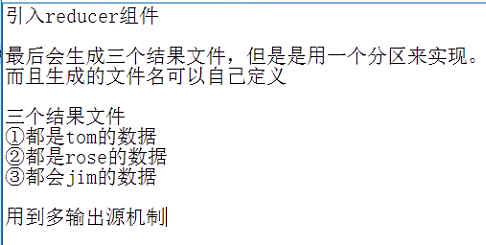




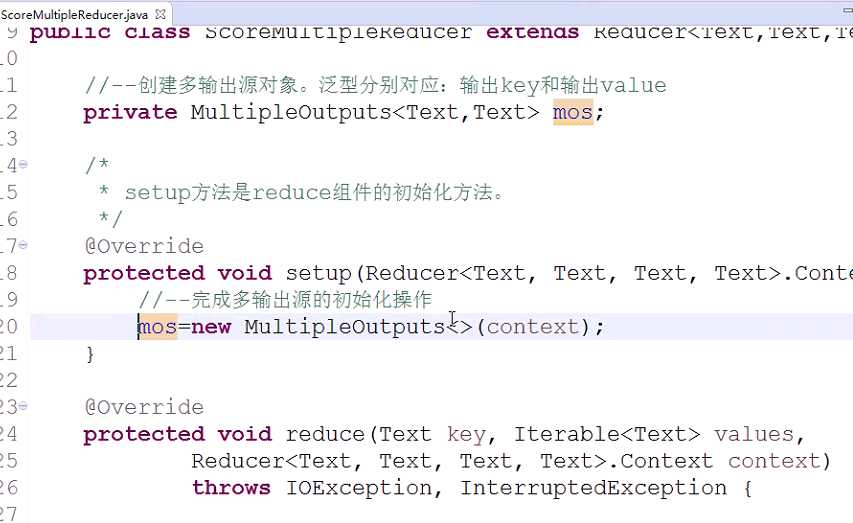


**sorcerer多输入**





**ScoreMultipleReducer**





## big\_day12

Hadoop07

**倒排索引**

**join**

**map side join**

在利用MapReduce做Join操作时，经常会出现数据倾斜的情况，**产生的原因主要来自于业务**，比如Mapper输出的是热销商品的pid，这样会造成某个join操作的reduce收到的数据特别多。

**如何解决数据倾斜**是一个常谈的话题，不同的框架有不同的处理方案，

如果是MR框架的话，我们可以利用DistributedCache（Hadoop内置的分布式缓存机制）来实现。 DistributedCache 是一个提供给Map/Reduce框架的工具，用来缓存指定的文件。当我们使用了这个机制后，MR框架底层会将指定的文件拷贝到slave节点上的缓存中。

使用DistributedCache机制，尤其在做join操作时，可以大大的提高作业的运行效率，并且可以额避免产生数据倾斜。**实现思路**是：

将Join操作中的**小表**进行缓存，这样每个Map Task在执行时，都是可以在Map Task运行所在的节点的缓冲区拿到小表数据，从而在Map阶段就可以完成Join操作。这样一来，就不需要引入Reducer组件，也就不会产生数据倾斜的问题。

**hadoop压缩机制**

**1 gzip压缩**

优点：压缩率比较高，而且压缩/解压速度也比较快；hadoop本身支持，在应用中处理gzip格式的文件就和直接处理文本一样；有hadoop native库；大部分linux系统都自带gzip命令，使用方便。

  缺点：不支持split。

应用场景：当每个文件压缩之后在128M以内的（1个块大小内），都可以考虑用gzip压缩格式。譬如说一天或者一个小时的日志压缩成一个gzip文件，运行mapreduce程序的时候通过多个gzip文件达到并发。hive程序，streaming程序，和java写的mapreduce程序完全和文本处理一样，压缩之后原来的程序不需要做任何修改。

**2 bzip2压缩**

  优点：支持split；具有很高的压缩率，比gzip压缩率都高；hadoop本身支持，但不支持native；在linux系统下自带bzip2命令，使用方便。

  缺点：压缩/解压速度慢；不支持native。

应用场景：适合对速度要求不高，但需要较高的压缩率的时候，可以作为mapreduce作业的输出格式；或者输出之后的数据比较大，处理之后的数据需要压缩存档减少磁盘空间并且以后数据用得比较少的情况；或者对单个很大的文本文件想压缩减少存储空间，同时又需要支持split，而且兼容之前的应用程序（即应用程序不需要修改）的情况。

**hadoop压缩api实现**

**yarn概述**

Apache Hadoop YARN （Yet Another Resource Negotiator，另一种资源协调者）是一种新的 Hadoop 资源管理器，它是一个通用**资源管理**系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在**资源利用率、资源统一管理**和数据共享等方面带来了巨大好处。

**YARN的基本思想**是将JobTracker的两个主要功能（**资源管理**和**作业调度/监控**）分离，主要方法是创建一个全局的ResourceManager（RM）和若干个针对应用程序的ApplicationMaster（AM）。这里的应用程序是指传统的MapReduce作业。

**YARN 分层结构的本质**是 ResourceManager。这个实体控制整个集群并管理应用程序向基础计算资源的分配。ResourceManager 将各个资源部分（计算、内存、带宽等）精心安排给基础 NodeManager（YARN 的每节点代理）。

  ResourceManager 还与 ApplicationMaster 一起分配资源，与 NodeManager 一起启动和监视它们的基础应用程序。在此上下文中，ApplicationMaster 承担了以前的 TaskTracker 的一些角色，ResourceManager 承担了 JobTracker 的角色。

  ApplicationMaster 管理一个在 YARN 内运行的应用程序的每个实例。ApplicationMaster 负责协调来自 ResourceManager 的资源，并通过 NodeManager 监视容器的执行和资源使用（CPU、内存等的资源分配）。

YARN的核心思想

将JobTracker和TaskTacker进行分离，它由下面几大构成组件：

a. 一个全局的资源管理器 ResourceManager

b.ResourceManager的每个节点代理 NodeManager

c. 表示每个应用的 ApplicationMaster

d. 每一个ApplicationMaster拥有多个Container在NodeManager上运行

**YARN的主要架构**

**ResourceManager（RM）**

RM是一个全局的资源管理器，**负责整个系统的资源管理和分配**。它主要由两个组件构成：**调度器（Scheduler）**和**应用程序管理器**（Applications Manager，ASM）。

调度器 调度器根据容量、队列等限制条件（如每个队列分配一定的资源，最多执行一定数量的作业等），将系统中的资源分配给各个正在运行的应用程序。需要注意的是，该调度器是一个“纯调度器”，它不再从事任何与具体应用程序相关的工作，比如不负责监控或者跟踪应用的执行状态等，也不负责重新启动因应用执行失败或者硬件故障而产生的失败任务，这些均交由应用程序相关的ApplicationMaster完成。**调度器仅根据各个应用程序的资源需求进行资源分配，而资源分配单位用一个抽象概念“资源容器”（Resource Container，简称Container）表示**，Container是一个动态资源分配单位，它将内存、CPU资源封装在一起，从而限定每个任务使用的资源量。

应用程序管理器(Applications Manager)负责管理整个系统中所有应用程序，包括应用程序提交、与调度器协商资源以启动ApplicationMaster、监控ApplicationMaster运行状态并在失败时重新启动它等。

**ApplicationMaster（AM）**

用户提交的每个应用程序均包含一个AM，主要功能包括：

与RM调度器协商以获取资源（用Container表示）；

将得到的任务进一步分配给内部的任务(资源的二次分配)；

与NM通信以启动/停止任务；

监控所有任务运行状态，并在任务运行失败时重新为任务申请资源以重启任务。

**NodeManager（NM）**

NM是每个节点上的资源和任务管理器，一方面，它会定时地向RM汇报本节点上的资源使用情况和各个Container的运行状态；另一方面，它接收并处理来自AM的Container启动/停止等各种请求。

**Container**

Container是YARN中的资源抽象，它封装了某个节点上的内存、CPU资源，当AM向RM申请资源时，RM为AM返回的资源便是用Container表示。YARN会为每个任务分配一个Container，且该任务只能使用该Container中描述的资源。

**相关配置（yarn-site.xml）**

|  |  |
| --- | --- |
| 参数 | 默认值 |
| yarn.nodemanager.resource.memory-mb | 8192（MB)（每台服务nm服务器贡献的内存），工作中，要根据服务器的实际内存来调节。  比如服务器内存：64GB。给操作系统留出8G。还需要考虑这个服务器上是否还运行，比如Hbase。给Hbase留出16GB  剩下的40GB留给yarn |
| yarn.nodemanager.resource.cpu-vcores | 8,cpu核数，根据实际情况来配置，有几核就配置几个。 |
| yarn.scheduler.minimum-allocation-mb | 1024（MB），每个Container最小的使用内存量 |
| yarn.scheduler.maximum-allocation-mb | 8192（MB），每个Container最大的使用内存量 |
| yarn.scheduler.minimum-allocation-vcores | 1，每个Container最少的使用核数 |
| yarn.scheduler.maximum-allocation-vcores | 4，每个Container最多使用的核数 |
| mapreduce.map.memory.mb | 1024（MB），每个MapTask运行所有的内存大小。此参数如果在Container的下限和上限之间，就用设置的参数值。  如果不在上限和下限范围，就取下限或上限值 |
| mapreduce.reduce.memory.mb | 1024（MB） |
| yarn.resourcemanager.scheduler.class | org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler  配置Yarn的调度器类型，默认是容器调度器。  另外两种调度器：  ①FIFO 调度器  ②Fair 调度器 |

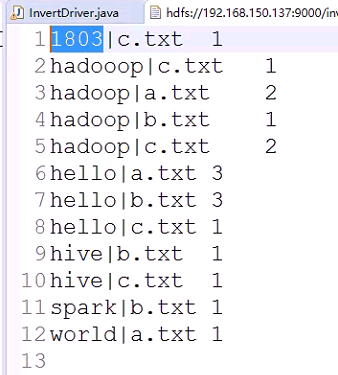
**hadoop2.0jvm重用及调优**

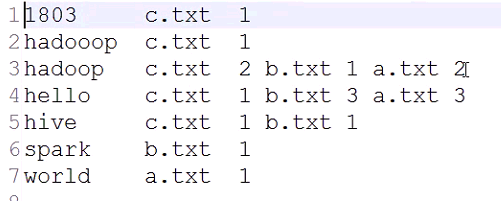
**hadoop小文件的处理方法**

**hadoop生态系统**

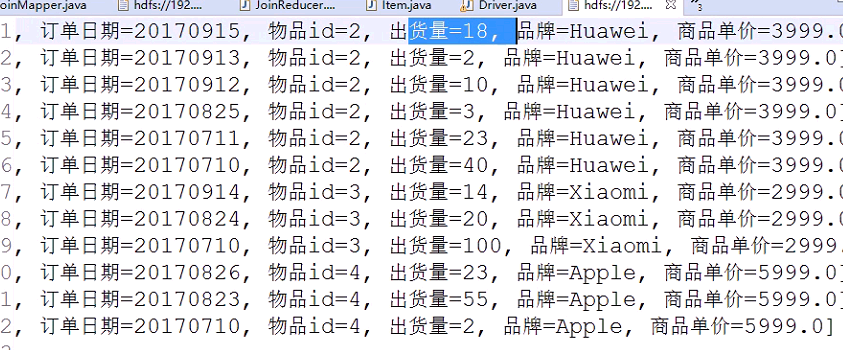
**就业:生产环境的服务器设置**

1.倒排索引Invert





2.Join



怎样解决数据倾斜?

1.combiner

2.map side join





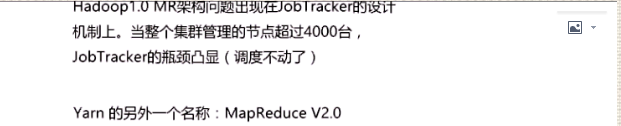
hadoop压缩文件

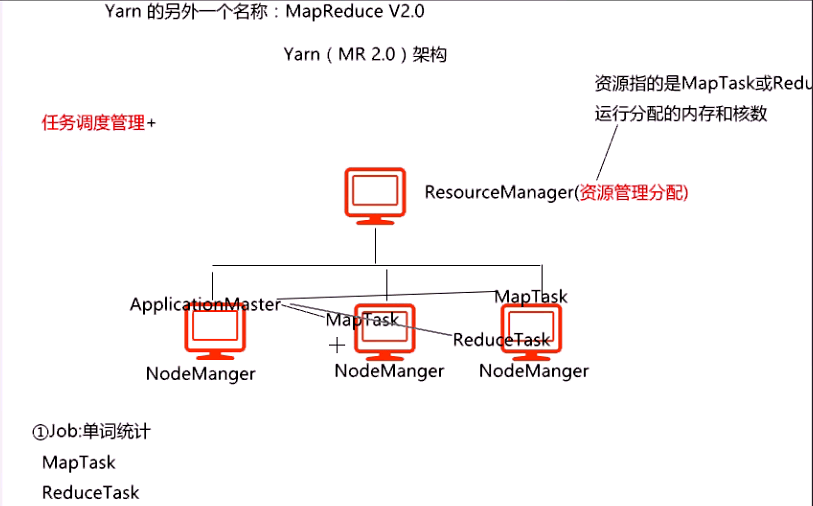
gzip,

bzip

用day03的word测试

**yarn**





<http://176.129.8.81:8088/cluster>