笔记

目录

[1hdfs 4](#_Toc494142453)

[1.1hadoop安装 4](#_Toc494142454)

[1.1.1所需软件 4](#_Toc494142455)

[1.1.2机器部署 4](#_Toc494142456)

[1.1.3创建hadoop用户 4](#_Toc494142457)

[1.1.4jdk、Hadoop配置环境变量 5](#_Toc494142458)

[1.1.5hosts配置 5](#_Toc494142459)

[1.1.6关闭防火墙和安全模式 6](#_Toc494142460)

[1.1.7ssh免密码登录 6](#_Toc494142461)

[1.1.8zookeeper安装 7](#_Toc494142462)

[1.1.9hadoop配置 8](#_Toc494142463)

[1.2hadoop启动 15](#_Toc494142464)

[1.2.1启动zookeeper 15](#_Toc494142465)

[1.2.2格式化namenode 16](#_Toc494142466)

[1.2.3格式化zk 16](#_Toc494142467)

[1.2.4namenode1同步到namenode2 16](#_Toc494142468)

[1.2.5启动namenode、datanode、JournalNode,zkfc 16](#_Toc494142469)

[1.2.6启动yarn 17](#_Toc494142470)

[1.2.7启动historyserver 17](#_Toc494142471)

[1.3hadoop命令 17](#_Toc494142472)

[1.3.1用户命令 17](#_Toc494142473)

[1.3.2管理命令 18](#_Toc494142474)

[1.4hadoop shell命令 18](#_Toc494142475)

[1.4.1基础命令 18](#_Toc494142476)

[1.4.2对象存储 23](#_Toc494142477)

[1.5hdfs命令 24](#_Toc494142478)

[1.5.1用户命令 24](#_Toc494142479)

[1.5.2系统命令 26](#_Toc494142480)

[1.5.3调试命令 31](#_Toc494142481)

[1.6hadoop架构 31](#_Toc494142482)

[2mapreduce 39](#_Toc494142483)

[2.1mr总体执行流程 39](#_Toc494142484)

[2.2mr内部执行流程 41](#_Toc494142485)

[2.3RM的HA机制 45](#_Toc494142486)

[2.3Partitioner 54](#_Toc494142487)

[2.4Combiner 56](#_Toc494142488)

[2.5wordcount 66](#_Toc494142489)

[2.6mapreduce命令 68](#_Toc494142490)

[2.6.1用户命令 68](#_Toc494142491)

[2.6.2系统命令 69](#_Toc494142492)

[2.7yarn命令 69](#_Toc494142493)

[2.7.1用户命令 69](#_Toc494142494)

[2.7.2系统命令 70](#_Toc494142495)

# 1hdfs

## 1.1hadoop安装

### 1.1.1所需软件

|  |  |
| --- | --- |
| 虚拟机 | vmware10.0 |
| 操作系统 | centos6.8 |
| jdk | jdk-8u131-linux-x64.tar.gz |
| hadoop | hadoop-2.8.0.tar.gz |
| zookeeper | zookeeper-3.4.10.tar.gz |

### 1.1.2机器部署

|  |  |  |
| --- | --- | --- |
| hostname | ip | 运行的服务 |
| cenos1 | 192.168.70.131 | NameNode、DFSZKFailoverController(ZKFC) |
| cenos2 | 192.168.70.132 | NameNode、DFSZKFailoverController(ZKFC) |
| cenos3 | 192.168.70.133 | ResourceManager |
| cenos4 | 192.168.70.134 | ResourceManager |
| cenos5 | 192.168.70.135 | DataNode、NodeManager、JournalNode、QuorumPeerMain、 |
| cenos6 | 192.168.70.136 | DataNode、NodeManager、JournalNode、QuorumPeerMain、 |
| cenos7 | 192.168.70.137 | DataNode、NodeManager、JournalNode、QuorumPeerMain、 |

### 1.1.3创建hadoop用户

sudo useradd -m hadoop -s /bin/bash

sudo adduser hadoop sudo

sudo passwd hadoop # 设置hadoop用户密码

### 1.1.4jdk、Hadoop配置环境变量

上传jdk-8u131-linux-x64.tar.gz到/usr/local目录

新建java安装目录：mkdier -p /usr/local/java

解压：tar –zxvf jdk-8u131-linux-x64.tar.gz -C /usr/local/java

上传 hadoop-2.8.0.tar.gz到/usr/local目录

解压：tar –zxvf hadoop-2.8.0.tar.gz

重命名：mv hadoop-2.8.0 hadoop

配置环境变量:

vim /etc/profile

export HADOOP\_HOME=/usr/local/hadoop

export JAVA\_HOME=/usr/local/java/jdk1.8.0\_131

export JRE\_HOME=${JAVA\_HOME}/jre

export CLASSPATH=.:${JAVA\_HOME}/lib:${JRE\_HOME}/lib

export PATH=$PATH:${JAVA\_HOME}/bin:${JRE\_HOME}/bin:${HADOOP\_HOME}/bin:${HADOOP\_HOME}/sbin

使配置生效

source /etc/profile

测试：

java –version

### 1.1.5hosts配置

先在centos1上进行配置，然后scp /etc/hosts hadoop@centos2:/etc/等各个节点

vim /etc/hosts

192.168.70.131 centos1

192.168.70.132 centos2

192.168.70.133 centos3

192.168.70.134 centos4

192.168.70.135 centos5

192.168.70.136 centos6

192.168.70.137 centos7

### 1.1.6关闭防火墙和安全模式

chkconfig iptables off

vim /etc/selinux/config

将SELINUX=enforcing改为SELINUX=disabled，重启

### 1.1.7ssh免密码登录

进入：cd /home/Hadoop/.ssh

执行：ssh-keygen -t rsa 然后三个回车

导入到公钥文件：cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys

分发到需要登陆的机器：

scp ~/.ssh/ authorized\_keys   hadoop@centos2:/home/hadoop/.ssh/ authorized\_keys

在centos2上修改权限：

chmod 700 /home/hadoop/.ssh chmod 600 /home/hadoop/.ssh/authorized\_keys

这样centos1上的hadoop就可以无密码登录cetnos2了

如果centos2还想无密码登录cenos1,需要执行以下操作:

登录centos2,在/home/hadoop/.ssh

执行ssh-keygen -t rsa，三个回车

执行cat ~/.ssh/id\_rsa.pub >> ~/.ssh/authorized\_keys

将公钥复制到centos1,登录centos1，修改权限：

chmod 700 /home/hadoop/.ssh chmod 600 /home/hadoop/.ssh/authorized\_keys

按照这种方法，将centos1-7的id\_rsa.pub全部追加到authorized\_keys中，然后将authorized\_keys分发到centos1-7的/home/hadoop/.ssh目录下，然后修改权限

chmod 700 /home/hadoop/.ssh chmod 600 /home/hadoop/.ssh/authorized\_keys即可

### 1.1.8zookeeper安装

需要安装的机器：centos5、centos6、centos7

上传：zookeeper-3.4.10.tar.gz到/usr/local目录

解压：tar –zxvf zookeeper-3.4.10.tar.gz

重命名：mv zookeeper-3.4.10 zookeeper

修改配置：

cd /usr/local/zookeeper/conf/

cp zoo\_sample.cfg zoo.cfg

vim zoo.cfg

修改以下内容

dataDir=/usr/local/zookeeper/tmp

在最后添加：

server.1=centos5:2888:3888

server.2=centos6:2888:3888

server.3=centos7:2888:3888

保存退出

然后创建一个tmp文件夹:mkdir /usr/local/zookeeper/tmp

再创建一个空文件:touch /usr/local/zookeeper/tmp/myid

最后向该文件写入ID，填入myid文件里

centso5机器：echo 1 > /usr/local/zookeeper/tmp/myid

centso6机器：echo 2 > /usr/local/zookeeper/tmp/myid

centso7机器：echo 3 > /usr/local/zookeeper/tmp/myid

### 1.1.9hadoop配置

新建文件夹：

在cnetos1-4 mdkir /usr/local/Hadoop/tmp/dfs/name

在cnetos5-7 mdkir /usr/local/Hadoop/tmp/dfs/data

vim hadoop-env.sh

JAVA\_HOME=/usr/local/java/jdk1.8.0\_131

vim slaves:

centos3

centos4

centos5

core-site.xml:

<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>

<name>fs.defaultFS</name>

<value>hdfs://ns1</value>

</property>

<property>

<name>hadoop.tmp.dir</name>

<value>/usr/local/hadoop/tmp</value>

</property>

<property>

<name>ha.zookeeper.quorum</name>

<value>centos5:2181,centos6:2181,centos7:2181</value>

</property>

<property>

<name>hadoop.proxyuser.hadoop.hosts</name>

<value>\*</value>

</property>

<property>

<name>hadoop.proxyuser.hadoop.groups</name>

<value>\*</value>

</property>

<property>

<name>hadoop.proxyuser.hue.hosts</name>

<value>\*</value>

</property>

<property>

<name>hadoop.proxyuser.hue.groups</name>

<value>\*</value>

</property>

</configuration>

hdfs-site.xml:

<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>

<name>dfs.nameservices</name>

<value>ns1</value>

</property>

<property>

<name>dfs.ha.namenodes.ns1</name>

<value>nn1,nn2</value>

</property>

<property>

<name>dfs.namenode.rpc-address.ns1.nn1</name>

<value>centos1:9000</value>

</property>

<property>

<name>dfs.namenode.http-address.ns1.nn1</name>

<value>centos1:50070</value>

</property>

<property>

<name>dfs.namenode.rpc-address.ns1.nn2</name>

<value>centos2:9000</value>

</property>

<property>

<name>dfs.namenode.http-address.ns1.nn2</name>

<value>centos2:50070</value>

</property>

<property>

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

<value>qjournal://centos5:8485;centos6:8485;centos7:8485/ns1</value>

</property>

<property>

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

<value>/usr/local/hadoop/journal</value>

</property>

<property>

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

<value>true</value>

</property>

<property>

<name>dfs.client.failover.proxy.provider.ns1</name>

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

</property>

<property>

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

<value>

sshfence

shell(/bin/true)

</value>

</property>

<property>

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

<value>/home/hadoop/.ssh/id\_rsa</value>

</property>

<property>

<name>dfs.ha.fencing.ssh.connect-timeout</name>

<value>30000</value>

</property>

<property>

<name>dfs.webhdfs.enabled</name>

<value>true</value>

</property>

</configuration>

mapred-site.xml

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>

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

<value>yarn</value>

</property>

<property>

<name>mapreduce.jobhistory.address</name>

<value>centos3:10020</value>

</property>

<property>

<name>mapreduce.jobhistory.webapp.address</name>

<value>centos3:19888</value>

</property>

<property>

<name>mapreduce.jobhistory.intermediate-done-dir</name>

<value>/usr/local/hadoop/history/intermediate</value>

</property>

<property>

<name>mapreduce.jobhistory.done-dir</name>

<value>/usr/local/hadoop/history/done</value>

</property>

</configuration>

yarn-site.xml:

<?xml version="1.0"?>

<configuration>

<property>

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

<value>true</value>

</property>

<property>

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

<value>yrc</value>

</property>

<property>

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

<value>rm1,rm2</value>

</property>

<property>

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

<value>centos3</value>

</property>

<property>

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

<value>centos4</value>

</property>

<property>

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

<value>centos5:2181,centos6:2181,centos7:2181</value>

</property>

<property>

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

<value>mapreduce\_shuffle</value>

</property>

</configuration>

## 1.2hadoop启动

### 1.2.1启动zookeeper

在centos5-7上分别执行：

cd /usr/local/zookeeper/bin/

./zkServer.sh start

查看zookeeper的状态：

./zkServer.sh status

三个中有一个leader,2个follwer即启动正常

### 1.2.2格式化namenode

在centos1上执行：

hdfs namenode -format

### 1.2.3格式化zk

在centos1上执行：

hdfs zkfc -formatZK

### 1.2.4namenode1同步到namenode2

在centos2上执行：

hdfs namenode bootstrapstandby

#验证

tmp下生成dfs

### 1.2.5启动namenode、datanode、JournalNode,zkfc

在centos1上：

cd /usr/local/Hadoop/sbin

./start-dfs.sh

在centos2上执行：

cd /usr/local/Hadoop/sbin

./hadoop-daemon.sh start namenode

单个启动：

namenode: ./hadoop-daemon.sh start namenode

datanode: ./hadoop-daemon.sh start datanode

journalnode: ./hadoop-daemon.sh start journalnode zkfc: ./hadoop-daemon.sh start zkfc

### 1.2.6启动yarn

整体启动：

在centos3上：

cd /usr/local/Hadoop/sbin

./start-yarn.sh

在centosos4上：

cd /usr/local/Hadoop/sbin

./start-yarn.sh

单个启动：

cd /usr/local/Hadoop/sbin

resourcemanager: **yarn-daemon.sh start resourcemanager**

### 1.2.7启动historyserver

在centos3 cd /usr/local/Hadoop/sbin

./mr-jobhistory-daemon.sh start historyserver

访问地址：

hdfs:centos1:50070、centos2:50070

resourcemanager:centos3:8088、centos4:8088

jobhistory:centos3:19888

## 1.3hadoop命令

### 1.3.1用户命令

|  |  |
| --- | --- |
| 命令 | 用途 |
| archive | 创建hadoop存档 |
| checknative | 检查Hadoop本机代码的可用性 |
| classpath | 打印获取Hadoop jar和所需库所需的类路径 |
| credential | 管理证书提供者内的凭据，密码和秘密的命令 |
| distcp | 递归复制文件或目录 |
| fs | 查看文件目录 |
| jar | 行一个jar文件 |
| key | 通过KeyProvider管理密钥。有关KeyProvider的详细信息 |
| trace | 查看和修改Hadoop跟踪设置 |
| version | 打印版本 |
| classname | 运行名为classname的类 |

### 1.3.2管理命令

|  |  |
| --- | --- |
| daemonlog | 获取/设置由守护程序中由限定类名称标识的日志的日志级别 |

## 1.4hadoop shell命令

### 1.4.1基础命令

|  |  |  |
| --- | --- | --- |
| 命令 | 用法 | 节食 |
| appendToFile | hadoop fs -appendToFile <localsrc> ... <dst> | 将单个src或多个srcs从本地文件系统附加到目标文件系统。还从stdin读取输入并附加到目标文件系统 |
| cat | hadoop fs -cat [-ignoreCrc] URI [URI ...] | 将源路径复制到stdout |
| checksum | hadoop fs -checksum URI | 返回文件的校验和信息 |
| chgrp | hadoop fs -chgrp [-R] GROUP URI [URI ...] | 更改文件组的关联。用户必须是文件的所有者，否则是超级用户 |
| chmod | hadoop fs -chmod [-R] <MODE [，MODE] ... | OCTALMODE> URI [URI ...] | 更改文件的权限。使用-R，通过目录结构递归地进行更改。用户必须是文件的所有者，否则是超级用户 |
| chown | hadoop fs -chown [-R] [OWNER] [：[GROUP]] URI [URI] | 更改文件的所有者。用户必须是超级用户 |
| copyFromLocal | hadoop fs -copyFromLocal <localsrc> URI | 上传本地文件到hdfs系统 |
| copyToLocal | hadoop fs -copyToLocal [-ignorecrc] [-crc] URI <localdst> | 下载hdfs文件到本地 |
| count | hadoop fs -count [-q] [-h] [-v] [-x] [-t [<storage type>]] [-u] <paths> | 计算与指定文件模式匹配的路径下的目录，文件和字节数。获取配额和用法 |
| cp | hadoop fs -cp [-f] [-p | -p [topax]] URI [URI ...] <dest> | 将文件从源文件复制到目的地。此命令还允许多个源，在这种情况下，目标必须是目录 |
| createSnapshot |  |  |
| deleteSnapshot |  |  |
| df |  | 显示可用空间 |
| du | hadoop fs -du [-s] [-h] [-x] URI [URI ...] | 显示给定目录中包含的文件和目录的大小或文件的长度，以防它只是一个文件 |
| dus | hadoop fs -dus <args> | 弃用 |
| expunge | hadoop fs -expunge | 在垃圾邮件目录下的永久保留阈值以上的检查点中永久删除文件，并创建新的检查点 |
| find | hadoop fs -find <path> ... <expression> ... | 查找与指定表达式匹配的所有文件，并将选定的操作应用到它们。如果没有指定路径，则默认为当前工作目录。如果没有指定表达式，则默认为-print |
| get | hadoop fs -get [-ignorecrc] [-crc] [-p] [-f] <src> <localdst> | 将文件复制到本地文件系统。可能使用-ignorecrc选项复制CRC校验失败的文件 |
| getfacl | hadoop fs -getfacl [-R] <path> | 显示文件和目录的访问控制列表（ACL）。如果一个目录具有默认的ACL，那么getfacl也会显示默认的ACL。 |
| getfattr | hadoop fs -getfattr [-R] -n name | -d [-e en] <path> | 显示文件或目录的扩展属性名称和值 |
| getmerge | hadoop fs -getmerge [-nl] <src> <localdst> | 将源目录和目标文件作为输入，并将src中的文件连接到目标本地文件。可选择-nl可以设置为在每个文件的末尾添加换行符（LF）。-skip-empty-file可以用于在空文件的情况下避免不必要的换行符。 |
| help | hadoop fs -help |  |
| ls | hadoop fs -ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] | 对于文件，ls以文件形式返回stat |
| lsr | hadoop fs -lsr <args> | 弃用 |
| mkdir | hadoop fs -mkdir [-p] <paths> | 以路径uri为参数，创建目录 |
| moveFromLocal | hadoop fs -moveFromLocal <localsrc> <dst> | 移动本地文件到hdfs，本地文件在复制后被删除 |
| moveToLocal | hadoop fs -moveToLocal [-crc] <src> <dst> | 移动hdfs文件到本地，hdfs文件在移动后被删除 |
| mv | hadoop fs -mv URI [URI ...] <dest> | 将文件从源移动到目标。此命令还允许多个源，在这种情况下，目标需要是目录。不允许跨文件系统移动文件 |
| put | hadoop fs -put [-f] [-p] [-l] [-d] [ - | <localsrc1> ..]。<DST> | 将单个src或多个srcs从本地文件系统复制到目标文件系统。如果源设置为“ - ”，则还从stdin读取输入并写入目标文件系统 |
| renameSnapshot |  |  |
| rm | hadoop fs -rm [-f] [-r | -R] [-skipTrash] [-safely] URI [URI ...] | 删除指定为args的文件。  如果垃圾桶已启用，文件系统将删除的文件移动到垃圾桶目录 |
| rmdir | hadoop fs -rmdir [--ignore-fail-on-non-empty] URI [URI ...] | 删除目录 |
| rmr | hadoop fs -rmr [-skipTrash] URI [URI ...] | 递归删除 |
| setfacl | hadoop fs -setfacl [-R] [-b | -k -m | -x <acl\_spec> <path>] | [ - set <acl\_spec> <path>] | 设置文件和目录的访问控制列表（ACL） |
| setfattr | hadoop fs -setfattr -n name [-v value] | -x name <path> | 为文件或目录设置扩展属性名称和值。 |
| setrep | hadoop fs -setrep [-R] [-w] <numReplicas> <path> | 更改文件的复制因子。如果path是一个目录，则命令会递归地更改根据路径的目录树下的所有文件的复制因子 |
| stat | hadoop fs -stat [format] <path> ... |  |
| tail | hadoop fs-tail [-f] URI | 将文件的最后一千字节显示为stdout |
| test | hadoop fs -test - [defsz] URI | 测试文件类型 |
| text | hadoop fs -text <src> | 获取源文件并以文本格式输出文件。允许的格式是zip和TextRecordInputStream |
| touchz | hadoop fs -touchz URI [URI ...] | 创建一个零长度的文件。如果文件存在非零长度，则返回错误 |
| truncate | hadoop fs-truncate [-w] <length> <paths> | 将与指定文件模式匹配的所有文件截断为指定长度 |
| usage | hadoop fs -usage command | 返回单个命令的帮助 |

### 1.4.2对象存储

Hadoop FileSystem外壳可与Object Stores（如Amazon S3，Azure WASB和OpenStack Swift）一起使用。

＃创建一个目录 hadoop fs -mkdir s3a：// bucket / datasets / ＃从集群文件系统上传文件 hadoop fs -put /datasets/example.orc s3a：// bucket / datasets / ＃触摸一个文件 hadoop fs -touchz wasb：//yourcontainer@youraccount.blob.core.windows.net/touched

＃从集群文件系统上传文件 hadoop fs -put -d /datasets/example.orc s3a：// bucket / datasets / ＃从本地文件系统的用户主目录下载文件。 ＃注意它是扩展“〜”的shell，而不是hadoop fs命令 hadoop fs -copyFromLocal -d -f〜/ datasets / devices.orc s3a：// bucket / datasets / ＃从stdin创建一个文件 ＃特殊的“ - ”源意味着“使用stdin” 回声“你好”| hadoop fs -put -d -f - wasb：[//yourcontainer@youraccount.blob.core.windows.net/hello.txt](mailto://yourcontainer@youraccount.blob.core.windows.net/hello.txt)

|  |  |
| --- | --- |
| find | ＃枚举对象存储容器中的所有文件。 hadoop fs -find s3a：// bucket / -print 记得要逃避通配符来阻止shell首先扩展它们 hadoop fs -find s3a：// bucket / datasets / -name \ \*。txt -print |
| rename | hadoop fs -mv s3a：// bucket / datasets s3a：// bucket / historical |
| copy | hadoop fs -cp s3a：// bucket / datasets s3a：// bucket / historical |

## 1.5hdfs命令

### 1.5.1用户命令

|  |  |  |
| --- | --- | --- |
| 命令 | 格式 | 说明 |
| classpath | hdfs classpath [--glob | --jar <path> | -h | --help] | 打印获取Hadoop jar和所需库所需的类路径 |
| dfs | hdfs dfs [COMMAND [COMMAND\_OPTIONS]] | 在Hadoop支持的文件系统上运行文件系统命令 |
| fetchdt | hdfs fetchdt <opts> <token\_file\_path> | 从NameNode获取授权令 |
| fsck | hdfs fsck <path> [-list-corruptfileblocks | [-move | - 删除| -openforwrite] [-files [-blocks [-locations | -racks | -replicaDetails]]] [-includeSnapshots] [-storagepolicies] [-blockId <blk\_Id]] | 运行HDFS文件系统检查实用程序 |
| getconf | hdfs getconf -namenodes-secondaryNameNodes hdfs getconf -backupNodes hdfs getconf -includeFile hdfs getconf -excludeFile hdfs getconf -nnRpcAddresses hdfs getconf -confKey [key] | 从配置目录获取配置信息后处理 |
| groups | hdfs groups [username ...] | 返回给定一个或多个用户名的组信息 |
| lsSnapshottableDir | hdfs lsSnapshottableDir [-help] | 获取快照目录的列表。当以超级用户身份运行时，它将返回所有快照目录。否则返回当前用户拥有的那些目录。 |
| jmxget | hdfs jmxget [-localVM ConnectorURL | -port port | -server mbeanserver | -service service] | 从服务转储JMX信息 |
| oev | hdfs oev [OPTIONS] -i INPUT\_FILE -o OUTPUT\_FILE | Hadoop离线编辑查看器 |
| oiv | hdfs oiv [OPTIONS] -i INPUT\_FILE | Hadoop离线镜像查看器用于较新的镜像文件 |
| oiv\_legacy | hdfs oiv\_legacy [OPTIONS] -i INPUT\_FILE -o OUTPUT\_FILE | 离线镜像查看器，用于旧版本的hadoop |
| snapshotDiff | hdfs snapshotDiff <path> <fromSnapshot> <toSnapshot> | 确定HDFS快照之间的区别 |
| version | hdfs version | 打印版本 |

### 1.5.2系统命令

|  |  |  |
| --- | --- | --- |
| 命令 | 格式 | 说明 |
| balancer | hdfs balancer [-threshold <threshold>] [-policy <policy>] [-exclude [-f <hosts-file> | <comma-separated list of hosts>]] [-include [-f <hosts-file> | <comma-separated list of hosts>]] [-source [-f <hosts-file> | <comma-separated list of hosts>]] [-blockpools <comma-separated list of blockpool ids>] [-idleiterations <idleiterations>] | 运行集群平衡实用程序 |
| cacheadmin | hdfs cacheadmin -addDirective -path <path> -pool <pool-name> [-force] [-replication <replication>] [-ttl <time-to-live>] | hdfs缓存 |
| crypto | hdfs crypto -createZone -keyName <keyName> -path <path> hdfs crypto -listZones hdfs crypto -provisionTrash -path <path> hdfs crypto -help <command-name> | 加密 |
| datanode | datanode [-regular | -rollback | -rollingupgrade rollback] | 运行HDFS datanode |
| dfsadmin | hdfs dfsadmin [GENERIC\_OPTIONS] [-report [-live] [-dead] [-decommissioning]] [-safemode enter | leave | get | wait | forceExit] [-saveNamespace] [-rollEdits] [-restoreFailedStorage true |false |check] [-refreshNodes] [-setQuota <quota> <dirname>...<dirname>] [-clrQuota <dirname>...<dirname>] [-setSpaceQuota <quota> [-storageType <storagetype>] <dirname>...<dirname>] [-clrSpaceQuota [-storageType <storagetype>] <dirname>...<dirname>] [-finalizeUpgrade] [-rollingUpgrade [<query> |<prepare> |<finalize>]] [-metasave filename] [-refreshServiceAcl] [-refreshUserToGroupsMappings] [-refreshSuperUserGroupsConfiguration] [-refreshCallQueue] [-refresh <host:ipc\_port> <key> [arg1..argn]] [-reconfig <datanode |...> <host:ipc\_port> <start |status>] [-printTopology] [-refreshNamenodes datanodehost:port] [-deleteBlockPool datanode-host:port blockpoolId [force]] [-setBalancerBandwidth <bandwidth in bytes per second>] [-getBalancerBandwidth <datanode\_host:ipc\_port>] [-allowSnapshot <snapshotDir>] [-disallowSnapshot <snapshotDir>] [-fetchImage <local directory>] [-shutdownDatanode <datanode\_host:ipc\_port> [upgrade]] [-getDatanodeInfo <datanode\_host:ipc\_port>] [-evictWriters <datanode\_host:ipc\_port>] [-triggerBlockReport [-incremental] <datanode\_host:ipc\_port>] [-help [cmd]] |  |
| haadmin | hdfs haadmin -checkHealth <serviceId> hdfs haadmin -failover [--forcefence] [--forceactive] <serviceId> <serviceId> hdfs haadmin -getServiceState <serviceId> hdfs haadmin -help <command> hdfs haadmin -transitionToActive <serviceId> [--forceactive] hdfs haadmin -transitionToStandby <ser |  |
| journalnode | hdfs journalnode |  |
| mover | hdfs mover [-p <files/dirs> | -f <local file name>] | 运行数据迁移实用程序 |
| namenode | hdfs namenode [-backup] | [-checkpoint] | [-format [-clusterid cid ] [-force] [-nonInteractive] ] | [-upgrade [-clusterid cid] [-renameReserved<k-v pairs>] ] | [-upgradeOnly [-clusterid cid] [-renameReserved<k-v pairs>] ] | [-rollback] | [-rollingUpgrade <rollback |started> ] | [-finalize] | [-importCheckpoint] | [-initializeSharedEdits] | [-bootstrapStandby [-force] [-nonInteractive] [-skipSharedEditsCheck] ] | [-recover [-force] ] | [-metadataVersion ] | 运行namenode |
| nfs3 | hdfs nfs3 | 启动NFS3网关 |
| portmap | hdfs portmap | 启动RPC portmap |
| secondarynamenode | hdfs secondarynamenode [-checkpoint [force]] | [-format] | [-geteditsize] | 运行HDFS辅助namenode |
| storagepolicies | hdfs storagepolicies | 列出所有存储策略 |
| zkfc | hdfs zkfc [-formatZK [-force] [-nonInteractive]] | 启动一个Zookeeper故障转移控制器进程 |

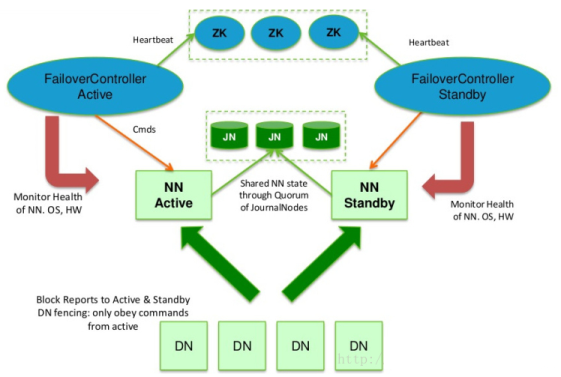
### 1.5.3调试命令

|  |  |  |
| --- | --- | --- |
| 命令 | 格式 | 说明 |
| verifyMeta | hdfs debug verifyMeta -meta <metadata-file> [-block <block-file>] | 验证HDFS元数据和阻止文件 |
| computeMeta | hdfs debug computeMeta -block <block-file> -out <output-metadata-file> | 从块文件计算HDFS元数据 |
| recoverLease | hdfs debug recoverLease -path <path> [-retries <num-retries>] | 在指定的路径上恢复租约 |

## 1.6hadoop架构

#### 1.6.1hdfs架构

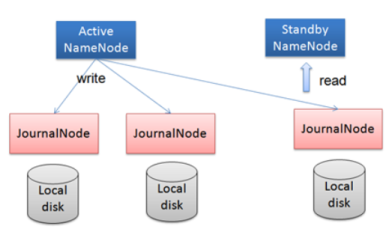
hadoop2.x之后，Clouera提出了QJM/Qurom Journal Manager，这是一个基于Paxos算法实现的HDFS HA方案，它给出了一种较好的解决思路和方案,示意图如下：



基本原理就是用2N+1台 JN 存储EditLog，每次写数据操作有大多数（>=N+1）返回成功时即认为该次写成功，数据不会丢失了。当然这个算法所能容忍的是最多有N台机器挂掉，如果多于N台挂掉，这个算法就失效了。这个原理是基于Paxos算法

在HA架构里面SecondaryNameNode这个冷备角色已经不存在了，为了保持standby NN时时的与主Active NN的元数据保持一致，他们之间交互通过一系列守护的轻量级进程JournalNode

任何修改操作在 Active NN上执行时，JN进程同时也会记录修改log到至少半数以上的JN中，这时 Standby NN 监测到JN 里面的同步log发生变化了会读取 JN 里面的修改log，然后同步到自己的的目录镜像树里面，如下图：



当发生故障时，Active的 NN 挂掉后，Standby NN 会在它成为Active NN 前，读取所有的JN里面的修改日志，这样就能高可靠的保证与挂掉的NN的目录镜像树一致，然后无缝的接替它的职责，维护来自客户端请求，从而达到一个高可用的目的

QJM方式来实现HA的主要优势：

不需要配置额外的高共享存储，降低了复杂度和维护成本

消除spof

系统鲁棒性(Robust:健壮)的程度是可配置

JN不会因为其中一台的延迟而影响整体的延迟，而且也不会因为JN的数量增多而影响性能（因为NN向JN发送日志是并行的）

hadoop2.x ha 详述：

datanode的fencing: 确保只有一个NN能命令DN。HDFS-1972中详细描述了DN如何实现fencing

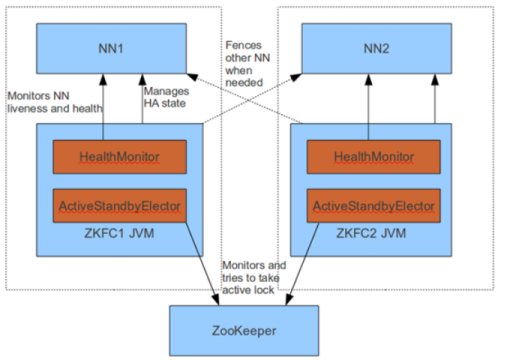
每个NN改变状态的时候，向DN发送自己的状态和一个序列号

DN在运行过程中维护此序列号，当failover时，新的NN在返回DN心跳时会返回自己的active状态和一个更大的序列号。DN接收到这个返回则认为该NN为新的active

如果这时原来的active NN恢复，返回给DN的心跳信息包含active状态和原来的序列号，这时DN就会拒绝这个NN的命令

客户端fencing：确保只有一个NN能响应客户端请求，让访问standby nn的客户端直接失败。在RPC层封装了一层，通过FailoverProxyProvider以重试的方式连接NN。通过若干次连接一个NN失败后尝试连接新的NN，对客户端的影响是重试的时候增加一定的延迟。客户端可以设置重试此时和时间

Hadoop提供了ZKFailoverController角色，部署在每个NameNode的节点上，作为一个deamon进程, 简称zkfc，示例图如下：



FailoverController主要包括三个组件:

HealthMonitor: 监控NameNode是否处于unavailable或unhealthy状态。当前通过RPC调用NN相应的方法完成

ActiveStandbyElector: 管理和监控自己在ZK中的状态

ZKFailoverController 它订阅HealthMonitor 和ActiveStandbyElector 的事件，并管理NameNode的状态

ZKFailoverController主要职责：

健康监测：周期性的向它监控的NN发送健康探测命令，从而来确定某个NameNode是否处于健康状态，如果机器宕机，心跳失败，那么zkfc就会标记它处于一个不健康的状态

会话管理：如果NN是健康的，zkfc就会在zookeeper中保持一个打开的会话，如果NameNode同时还是Active状态的，那么zkfc还会在Zookeeper中占有一个类型为短暂类型的znode，当这个NN挂掉时，这个znode将会被删除，然后备用的NN，将会得到这把锁，升级为主NN，同时标记状态为Active

当宕机的NN新启动时，它会再次注册zookeper，发现已经有znode锁了，便会自动变为Standby状态，如此往复循环，保证高可靠，需要注意，目前仅仅支持最多配置2个NN

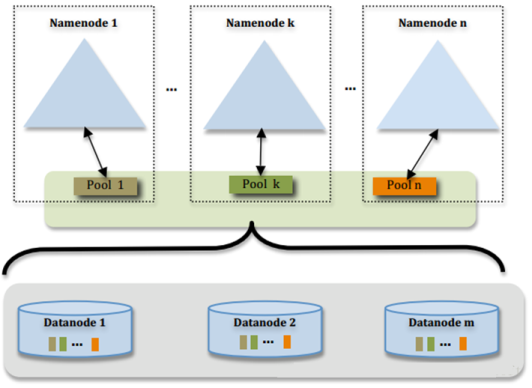
master选举：如上所述，通过在zookeeper中维持一个短暂类型的znode，来实现抢占式的锁机制，从而判断那个NameNode为Active状态

hadoop2.x Federation：

单Active NN的架构使得HDFS在集群扩展性和性能上都有潜在的问题，当集群大到一定程度后，NN进程使用的内存可能会达到上百G，NN成为了性能的瓶颈

常用的估算公式为1G对应1百万个块，按缺省块大小计算的话，大概是64T (这个估算比例是有比较大的富裕的，其实，即使是每个文件只有一个块，所有元数据信息也不会有1KB/block)

为了解决这个问题,Hadoop 2.x提供了HDFS Federation, 示意图如下：



多个NN共用一个集群里的存储资源，每个NN都可以单独对外提供服务

每个NN都会定义一个存储池，有单独的id，每个DN都为所有存储池提供存储

DN会按照存储池id向其对应的NN汇报块信息，同时，DN会向所有NN汇报本地存储可用资源情况

如果需要在客户端方便的访问若干个NN上的资源，可以使用客户端挂载表，把不同的目录映射到不同的NN，但NN上必须存在相应的目录

设计优势：

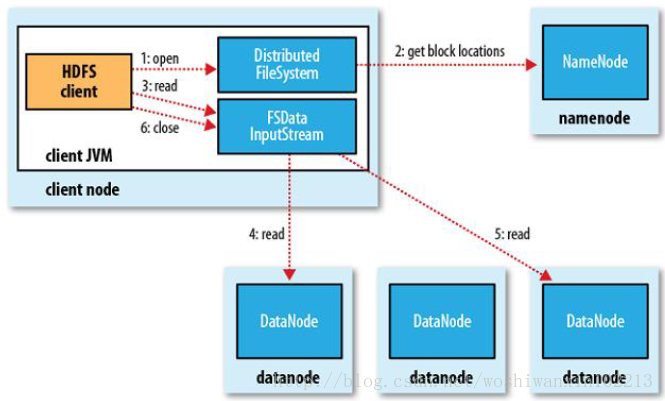
改动最小，向前兼容；现有的NN无需任何配置改动；如果现有的客户端只连某台NN的话，代码和配置也无需改动

分离命名空间管理和块存储管理

客户端挂载表：通过路径自动对应NN、使Federation的配置改动对应用透明

#### 1.6.2hdfs读写流程

hdfs文件读取：

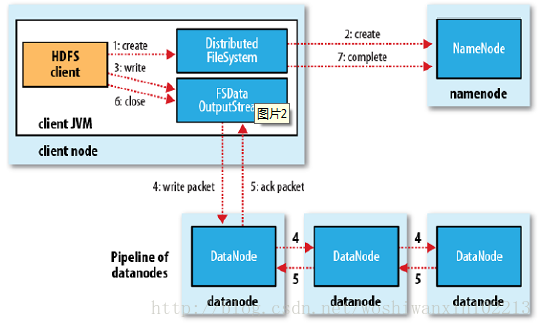


1.首先调用FileSystem对象的open方法，其实是一个DistributedFileSystem的实例2.DistributedFileSystem通过rpc获得文件的第一批个block的locations，同一block按照重复数会返回多个locations，这些locations按照hadoop拓扑结构排序，距离客户端近的排在前面.；  
3.前两步会返回一个FSDataInputStream对象，该对象会被封装成 DFSInputStream对象，DFSInputStream可以方便的管理datanode和namenode数据流。客户端调用read方 法，DFSInputStream最找出离客户端最近的datanode并连接。  
4.数据从datanode源源不断的流向客户端。  
5.如果第一块的数据读完了，就会关闭指向第一块的datanode连接，接着读取下一块。这些操作对客户端来说是透明的，客户端的角度看来只是读一个持续不断的流。  
6.如果第一批block都读完了，DFSInputStream就会去namenode拿下一批blocks的location，然后继续读，如果所有的块都读完，这时就会关闭掉所有的流。

**HDFS读取发生异常处理**  
       如果在读数据的时候，DFSInputStream和datanode的通讯发生异常，就会尝试正在读的block的排第二近的datanode,并且 会记录哪个datanode发生错误，剩余的blocks读的时候就会直接跳过该datanode。DFSInputStream也会检查block数据校验和，如果发现一个坏的block,就会先报告到namenode节点，然后DFSInputStream在其他的datanode上读该block的镜像  
HDFS读操作设计思考

客户端直接连接datanode来检索数据并且namenode来负责为每一个block提供最优的datanode，namenode仅仅处理 block location的请求，这些信息都加载在namenode的内存中，hdfs通过datanode集群可以承受大量客户端的并发访问。

**HDFS文件写入**

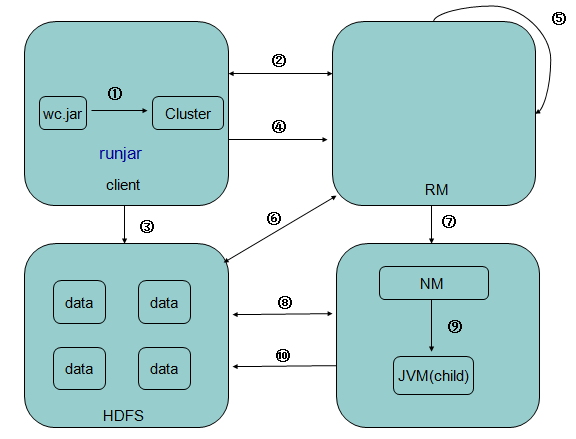


1.客户端通过调用DistributedFileSystem的create方法创建新文件  
2.DistributedFileSystem通过RPC调用namenode去创建一个 没有blocks关联的新文件，创建前，namenode会做各种校验，比如文件是否存在，客户端有无权限去创建等。如果校验通过，namenode就会记录下新文件，否则就会抛出IO异常.  
3.前两步结束后会返回FSDataOutputStream的对象，和读文件的时候相似，FSDataOutputStream被封装成DFSOutputStream，DFSOutputStream可以协调namenode和 datanode。客户端开始写数据到DFSOutputStream,DFSOutputStream会把数据切成一个个小packet，然后排成队列 data quene。  
4.DataStreamer会去处理接受data quene，他先问询namenode这个新的block最适合存储的在哪几个datanode里，比如重复数是3，那么就找到3个最适合的 datanode，把他们排成一个pipeline.DataStreamer把packet按队列输出到管道的第一个datanode中，**第一个 datanode又把packet输出到第二个datanode中**，以此类推。  
5.DFSOutputStream还有一个对列叫ack quene，也是有packet组成，等待datanode的收到响应，当pipeline中的所有datanode都表示已经收到的时候，这时akc quene才会把对应的packet包移除掉。  
6.客户端完成写数据后调用close方法关闭写入流  
7.DataStreamer把剩余得包都刷到pipeline里然后等待ack信息，收到最后一个ack后，通知datanode把文件标示为已完成。

**HDFS文件写入失败**  
       如果在写的过程中某个datanode发生错误，会采取以下几步：  
       1.pipeline被关闭  
       2.为了防止防止丢包ack quene里的packet会同步到data quene  
       3.把产生错误的datanode上当前在写但未完成的block删掉  
       4.block剩下的部分被写到剩下的两个正常的datanode  
       5.namenode找到另外的datanode去创建这个块的副本  
       这些操作对客户端来说是无感知的。  
       (客户端执行write操作后，写完得block才是可见的，正在写的block对客户端是不可见的，只有调用sync方法，客户端才确保该文件被写操 作已经全部完成，当客户端调用close方法时会默认调用sync方法。是否需要手动调用取决你根据程序需要在数据健壮性和吞吐率之间的权衡。)

# 2mapreduce

## 2.1mr总体执行流程



(1).客户端提交一个mr的jar包给JobClient(提交方式：hadoop jar ...)

(2).JobClient通过RPC和ResourceManager进行通信，返回一个存放jar包的地址（HDFS）和jobId

(3).client将jar包写入到HDFS当中(path = hdfs上的地址 + jobId)

(4).开始提交任务(任务的描述信息，不是jar, 包括jobid，jar存放的位置，配置信息等等)

(5)ResourceManager进行初始化任务

(6)读取HDFS上的要处理的文件，开始计算输入分片，每一个分片对应一个MapperTask

(7)NodeManager通过心跳机制**领取任务**（任务的描述信息）

(8).下载所需的jar，配置文件等

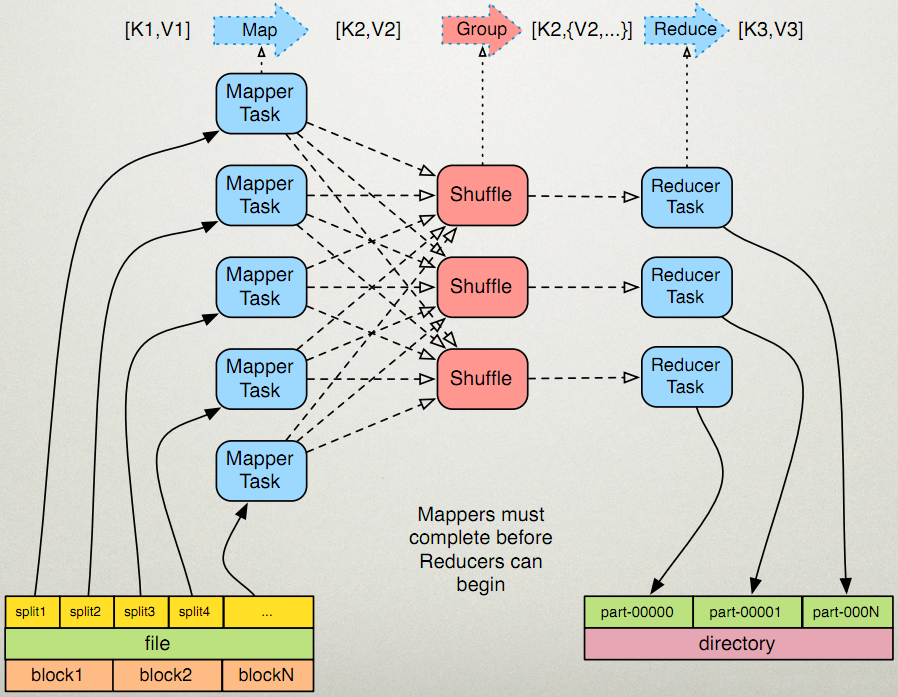
(9). NodeManager启动一个java child子进程，用来执行具体的任务（MapperTask或ReducerTask）

(10).将结果写入到HDFS当中

执行失败

**1.如果task出现问题(map或者reduce)**  
错误可能原因：用户代码出现异常；任务超过mapred.task.timeout指定的时间依然没有返回  
错误处理：  
首先将错误信息写入日志  
然后ResourceManager会调度其他NodeManager来重新执行次任务，如果失败次数超过4次(通过mapred.map.max.attempts和mapred.reduce.max.attempts属性来设置，默认为 则job以失败告终  
如果系统不想以这种方式结束退出，而是想通过Task成功数的百分比来决定job是否通过，则可以指定如下两个属性  
mapred.max.map.failures.percent            map任务最大失败率  
mapred.max.reduce.failures.percent        reduce任务最大失败率  
如果失败比率超过指定的值，则job以失败告终  
**2.如果是NodeManager出现问题**  
判断问题的依据：和ResourceManager不再心跳通信  
ResourceManager将该NodeManager从资源池中移除，以后不在调度它  
**3.ResourceManager出现问题**  
ResourceManager作为系统的单点如果出现问题也是最为严重的问题，系统将处于瘫痪

## 2.2mr内部执行流程

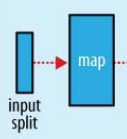


shuffle的详细执行过程



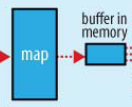
**map过程**

1. 一个inputsplit对应一个map任务，Split与block的对应关系可能是多对一，默认是一对一。



1. 每个map任务都有一个环形内存缓冲区buffer in memory，用于存储任务的输出。默认情况下，缓冲区的大小为100M,此值可以通过io.sort.mb属性来调整，

一旦缓冲区内容达到阈值（io.sort.spill.percent,默认为0.8,80%），一个后台线程便开始把内容溢出到磁盘中。在溢出的过程中，map输出继续被写到缓冲区，但如果在此期间缓冲区被填满，map会阻塞直到写磁盘过程完成。



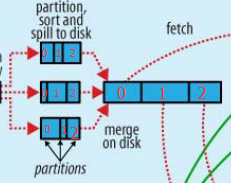
1. 在写磁盘之前，线程首先会根据数据最终要传送到的reducer把数据划分成相应的分区（partition）。系统默认使用自己的**HashPartitioner**进行分区，

public class HashPartitioner<K, V> extends Partitioner<K, V> {  
   
  public int getPartition(K key, V value,  
                          int numReduceTasks) {  
    return (**key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks**;  
  }  
}

比如有3个reducer任务，编号分别是0,1,2。我们通过**HashPartitioner**将输出数据进行编号，分别和reducer的编号相对应，这样就保证了每个reducer可以较为平均的取到任务。当然，partition也可以自己定义。分区完成之后再对每个分区的数据进行排序。如果有一个combiner，它会在排序后的输出上运行。



1. 一旦内存缓冲区达到溢出写出的阈值，就会新建一个溢出写出文件，因此在map任务写完其最后一个输出记录之后，会有几个溢出写出文件。在任务完成之前，溢出文件被合并成一个已分区且已排序的输出文件。配置io.sort.factor控制着一次最多能合并多少流，默认值是10.



**reducer过程-hadoop2**

computer0

ResourceManager

computer3

computer1

computer2

NodeManager

NodeManager

NodeManager

MrAppMaster

2

3

2

4

1、mapper在启动的过程中，会有一个nodemanager启动一个MrAppmaster

2、mapper在处理完成数据之后，会将信息发送MrAppMaster

3、reducer会启动一个线程，定时的从MrAppMaster上获取分配个自己处理的数据信息，比如位置信息

4、reducer在得到自己需要处理的文件信息之后，通过http协议去相应的机器上下载

map的输出数据，放在自己的磁盘上，加载进内存，合并，排序然后输出到hdfs。

**reducer过程-hadoop1**

computer0

JobTracker

3

2

TaskTracker

computer3

TaskTracker

computer2

TaskTracker

computer1

2

1

1

4

1、mapper在处理完成数据之后，会将信息发送自己的上级TaskTracker

2、TaskTracker收到信息之后会发送给自己的上级JobTracker

3、reducer会启动一个线程，跨过他的上级TaskTracker定时的从JobTracker上获取分配个自己处理的数据信息，比如位置信息

4、reducer在得到自己需要处理的文件信息之后，通过htpp协议去相应的机器上下载

map的输出数据，放在自己的磁盘上，加载进内存，合并，排序然后输出到hdfs

## 2.3RM的HA机制

大家都知道在hadoop2中对HDFS的改进很大，实现了NameNode的HA；也增加了ResourceManager。但是ResourceManager也可以实现HA。你没看错，确实是ResourceManager的HA。注意是在Apache Hadoop 2.4.1版本中开始加入的，可不是任意一个版本。

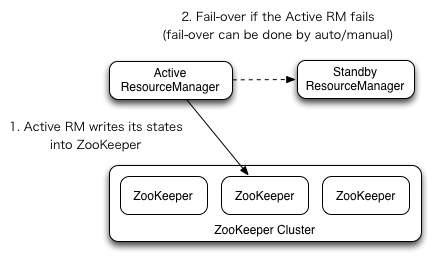
我们不讲单点问题的危害，直接讲如何配置ResourceManager的HA。

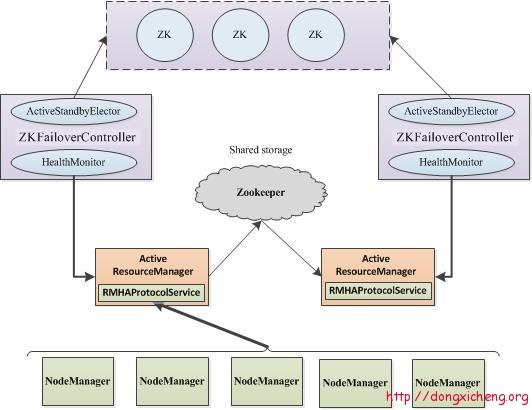
HA的架构

如 果大家理解HDFS的HA，那么ResourceManager的HA与之是相同道理的：也是Active/Standby架构，任意时刻，都一个是 Active，其余处于Standby状态的ResourceManager可以随时转换成Active状态。状态转换可以手工完成，也可以自动完成。手 工完成时通过命令行的管理命令(命令是“yarn rmadmin”)。自动完成是通过配置自动故障转移(automatic-failover)，使用集成的failover-controller完成 状态的自动切换。

自动故障转移是依赖于ZooKeeper集群，依赖ZooKeeper的ActiveStandbyElector会嵌入到ResourceManager中，当Active状态的ResourceManager失效时，处于 Standby状态的ResourceManager就会被选举为Active状态的，实现切换。注意：这里没有ZooKeeperFailoverController进程，这点和HDFS的HA不同。

对于客户端而言，必须知道所有的ResourceManager中。因此，需要在yarn-site.xml中配置所有的ResourceManager。 那么，当一个Active状态的ResourceManager失效时，客户端怎么办哪？客户端会采用轮询机制，轮询配置在yarn-site.xml中 的ResourceManager，直到找到一个active状态的ResourceManager。如果我们想修改这种寻找 ResourceManager的机制，可以继承类 org.apache.hadoop.yarn.client.RMFailoverProxyProvider，实现自己的逻辑。然后把类的名字配置到 yarn-site.xml的配置项yarn.client.failover-proxy-provider中。

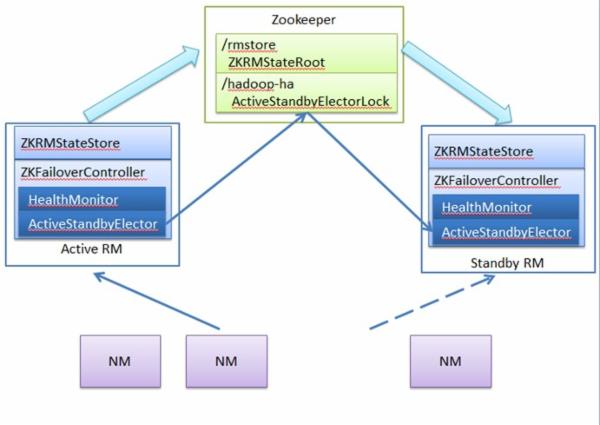




cdh5的RM的HA

直接记录配置 RM HA 的最小需求和配置项。跟NN的HA一样，RM的HA也需要两台机器硬件配置相同，这个没什么可解释的了，当初1代的时候，NN和SNN就必须是一模一样的 硬件配置。就像配置NN的HA一样，RM的HA也需要给出servicename。以下配置是配置RM自动失效恢复的配置项，大概没多少人会用到手动恢复吧，用手动恢复就把zookeeper部分给干掉就行了。

流程：两个RM, 启动的时候都是standby, 进程启动以后状态未被加载, 转换为active后才会加载相应的状态并启动服务. RM的状态通过配置可以存储在zookeeper, HDFS上。Standby转换到active可以通过命令或开启auto failover。

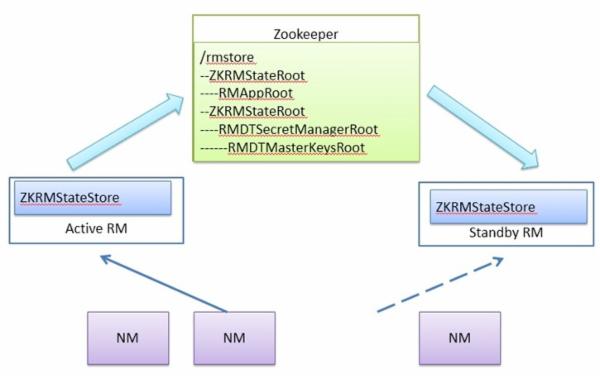


RM 的作业信息存储在ZK的/rmstore下，Active RM向+++这个目录写App信息。

RM启动的时候会通过向ZK的/hadoop-ha目录下写一个Lock文件，写成功则成为Active，否则为Standby，Standby RM会一直监控Lock文件是否存在，如果不存在则会试图去创建，即争取成为Active RM。

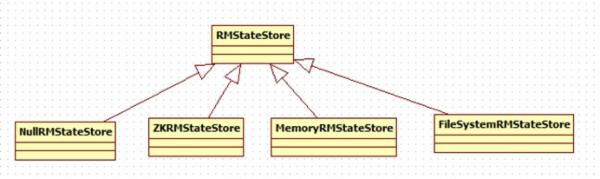
当Active RM挂掉，另外一个Standby RM成功转换为Active RM后，会从/rmstore读取相应的作业信息，重新构建作业的内存信息。然后启动内部服务，开始接收NM的心跳，构建集群资源信息，并接收客户端提交作业的请求等。

社区trunk版本当前也已经支持RM HA，但只支持手动切换，不支持Auto Failover。社区的基本原理和Cloudera RM HA类似，其架构图如下图所示：



对比Cloudera RM HA的架构图，仅少了Auto Failover部分。

服务端RM HA的关键部分主要为RMStateStore和ZKFailoverController。RMStateStore是实现RM状态存储的基类，主要包 括存储和加载RM状态等方法。实现类主要包括FileSystemRMStateStore和ZKRMStateStore。类图如下图所示。

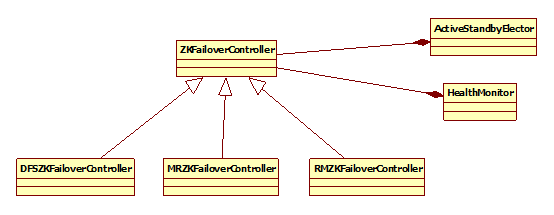


ZKFailoverController中维护着ActiveStandbyElector和HealthMonitor，ActiveStandbyElector主要工作是。

1. 初始化时在ZK上创建一个Lock文件，

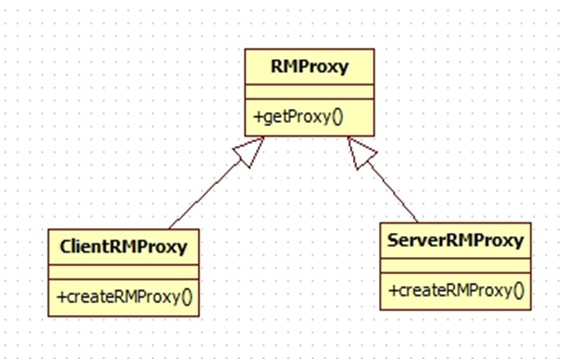
2. Standby RM运行过程中监控ZM上的Lock文件是否存在。

HealthMonitor的主要工作是检查自己（RM）的健康状态，通过HAServiceStatus提供的 getServiceStatus()和monitorHealth()方法，如果自己健康的，则会试图创建Lock文件，按照结果成为active或 standby。下图是ZKFailoverController的类图，图中可以看出，Cloudera的Hadoop版本中，NameNode、 Jobtracker和ResourceManager都采用相同的Auto Failover机制。



客户端的实现机制

在RM HA之前，客户端与RM的通信直接使用ApplicationClientProtocol等协议，增加RM HA后，客户端使用RetryPolicy，它提供了一种重试机制，但一个RM连不上，则会Failover到另外一台RM上。具体的实现方法是采用动态 代理模式，增加RMProxy实现retry方式连接RM。下图是RMProxy的类图。



其中ClientRMProxy，代理ApplicationClientProtocol、 ApplicationMasterProtocol、ResourceManagerAdministrationProtocol，实现 Yarn client、AM与RM的连接。ServerRMProxy提供给NM连接RM使用。代理ResourceTracker。

**配置说明：**

yarn.resourcemanager.zk-address

ZooKeeper服务器的地址（主机：端口号），既用于状态存储也用于内嵌的leader-election。

yarn.resourcemanager.ha.enabled

是否启用RM HA，默认为false（不启用）。

yarn.resourcemanager.ha.rm-ids

RMs的逻辑id列表，用逗号分隔，如：rm1,rm2

yarn.resourcemanager.hostname.rm-id

每个rm-id的主机名，rm-id的值包含在上面的参数值中。

yarn.resourcemanager.ha.id

可选项，用于标识RM。如果设置了，管理员需要确保所有的RMs在配置中都有自己的ID。

yarn.resourcemanager.ha.automatic-failover.enabled

是否启用自动故障转移。默认情况下，在启用HA时，启用自动故障转移。

yarn.resourcemanager.ha.automatic-failover.embedded

启用内置的自动故障转移。默认情况下，在启用HA时，启用内置的自动故障转移。

yarn.resourcemanager.cluster-id

集群的Id，elector使用该值确保RM不会做为其它集群的active。

yarn.client.failover-proxy-provider

Clients, AMs和NMs使用该类故障转移到active RM。

yarn.client.failover-max-attempts

FailoverProxyProvider尝试故障转移的最大次数。

yarn.client.failover-sleep-max-ms

故障转移间的最大休眠时间（单位：毫秒）。

yarn.client.failover-retries

每个尝试连接到RM的重试次数。

yarn.client.failover-retries-on-socket-timeouts

在socket超时时，每个尝试连接到RM的重试次数。

配置示例

下面是RM故障转移的简单配置示例，在该示例中启用了HA，那么也就启用了自动故障转移。

<property>

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

<value>true</value>

</property>

<property>

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

<value>cluster1</value>

</property>

<property>

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

<value>rm1,rm2</value>

</property>

<property>

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

<value>master1</value>

</property>

<property>

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

<value>master2</value>

</property>

<property>

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

<value>zk1:2181,zk2:2181,zk3:2181</value>

</property>

正如上面提到的，hadoop也为管理员提供了CLI的方式管理RM HA，但在没有启用HA的情况下，下面的命令是不可用的：

[hadoop@hadoop ~]$ yarn rmadmin -getServiceState

Cannot run -getServiceState when ResourceManager HA is not enabled

[hadoop@hadoop ~]$ yarn rmadmin -transitionToStandby

Cannot run -transitionToStandby when ResourceManager HA is not enabled

假设已经启用了HA，那么就可以通过CLI的方式查看RM的状态和手动进行故障转移，假设yarn.resourcemanager.ha.rm-ids的值为rm1和rm2：

$ yarn rmadmin -getServiceState rm1 active

$ yarn rmadmin -getServiceState rm2 standby

手动故障转移必须在自动故障转移禁用的前提下执行，否则会出现split-brain的情形或者其它不正确的状态，手动故障转移的命令为：

$ yarn rmadmin -transitionToStandby rm1

|  |  |
| --- | --- |
|  |  |

将配置写入yarn-site.xml里面，然后分别在两台服务器正常启动RM就可以了，就像启动NN HA一样的方式 sudo -u yarn yarn-daemon.sh start resourcemanager

另外，在Hadoop的各种HA中，有个隐藏属性是很多人不知道的，就是强制切换，一般来说，我们通过命令行切换HA，需要去运行

|  |  |
| --- | --- |
|  | sudo -u hdfs hdfs haadmin -transitionToActive/transitionToStandby |

或者

|  |  |
| --- | --- |
|  | sudo -u yarn yarn rmadmin -transitionToActive/transitionToStandby |

但是，这种方式在启用了ZKFC做自动失效恢复的状态下是不允许修改的，提示信息里只说了可以强制执行，但是没有提供命令，其实强制切换主备命令很简单。加个forcemanual就好了。

|  |  |
| --- | --- |
|  | sudo -u hdfs hdfs haadmin -transitionToActive --forcemanual nn1 |

但是这样做的后果是，ZKFC将停止工作，你将不会再有自动故障切换的保障，但是有些时候，这是必须的，特别是有时候，Hadoop的NN在ZKFC正常工 作的情况下，也会出现两个standby，两个standby的问题就在于诸如Hive和Pig这种东西，会直接报一个什么 Operation category READ is not supported in state standby 什么什么的，甚至你看着明明一个是active，一个是standby，也会报这个错误，这时候就必须手动强制切换了，强制切换完以后，别忘了重新启动 ZKFC就好了。这个强制切换的要求就是用户必须没有任何对元数据的操作，这样才能有效的防止脑裂的发生。应该来说，进入安全模式再切换会比较稳妥一些。

补充: Hadoop ResourceManager的实现不像namenode只能有两个做HA，ResourceManager的HA可以多台。

## 2.3Partitioner

在进行MapReduce计算时，有时候需要把最终的输出数据分到不同的文件中，比如按照省份划分的话，需要把同一省份的数据放到一个文 件中；按照性别划分的话，需要把同一性别的数据放到一个文件中。我们知道最终的输出数据是来自于Reducer任务。那么，如果要得到多个文件，意味着有 同样数量的Reducer任务在运行。Reducer任务的数据来自于Mapper任务，也就说Mapper任务要划分数据，对于不同的数据分配给不同的 Reducer任务运行。Mapper任务划分数据的过程就称作Partition。负责实现划分数据的类称作Partitioner。

在我们前面讲过的例子中，始终没有提到分区，那是因为框架内置了分区类，称作HashPartitioner。我们看一下源码

**public class HashPartitioner<K, V> extends Partitioner<K, V> {  
   
  public int getPartition(K key, V value,  
                          int numReduceTasks) {  
    return (key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks;  
  }  
}**

HashPartitioner是处理Mapper任务输出的，getPartition()方法有三个形参，**key、value分别指的是 Mapper任务的输出，numReduceTasks指的是设置的Reducer任务数量，默认值是1。**那么任何整数与1相除的余数肯定是0。也就是说 getPartition(…)方法的返回值总是0。也就是Mapper任务的输出总是送给一个Reducer任务，最终只能输出到一个文件中。

据此分析，如果想要最终输出到多个文件中，在Mapper任务中对数据应该划分到多个区中。那么，我们只需要按照一定的规则让getPartition(…)方法的返回值是0,1,2,3…即可。假设我们手机号码分区，手机号分到一起，小灵通分到一起，那么可以覆盖Partitioner类的getpartition(…)方法，代码如下：

**import** org.apache.hadoop.io.Text;

**import** org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;

//自定义partitioner 需要继承HashPartitioner,重写getPartition方法

**public** **class** MobilPartitioner **extends** HashPartitioner<Text, DataBean> {

@Override

//参数key:map的输出的key，value：map输出的value,numReduceTasks：reducer的数量

**public** **int** getPartition(Text key, DataBean value, **int** numReduceTasks) {

//将key是手机号到分到同一reducer，将小灵通数据分到同一reducer

**final** **int** length = key.toString().length();

**return** length == 11 ? 0 : 1;

}

}

我们通过给自定义partitioner的getPartition给输出的手机号进行分类，如果是11位，也就是手机的话，返回0，否则返回1，这样就将map的输出分配到了2个不同的reducer中。

注意：我们还需要在job中设置自定义的Partitioner和reducer任务的数量

//设置自定义Partitioner

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

//设置reducer的数量

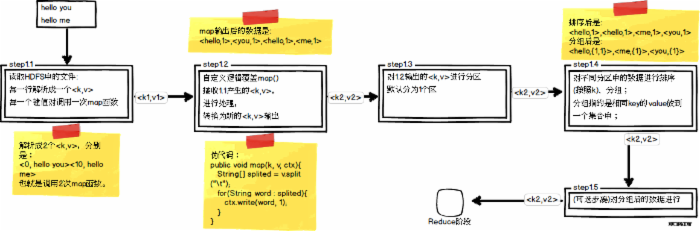
**job.setNumReduceTasks(2);**

## 2.4Combiner

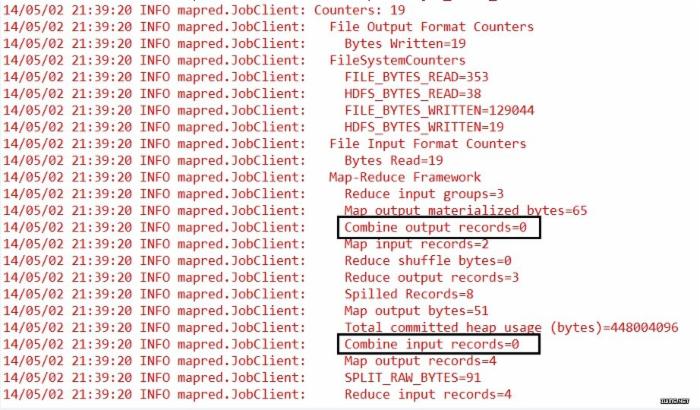
一、Combiner的出现背景

1.1 回顾Map阶段五大步凑

在第四篇博文《初识MapReduce》中，我们认识了MapReduce的八大步凑，其中在Map阶段总共五个步凑，如下图所示：

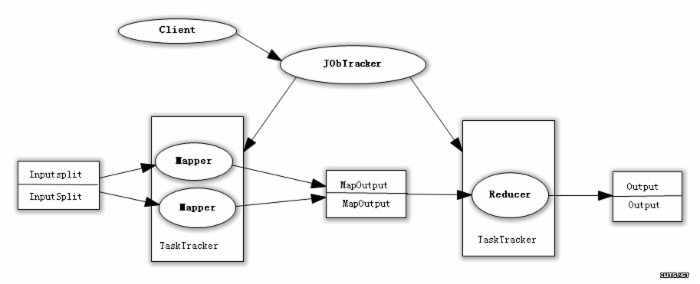


其中，step1.5是一个可选步凑，它就是我们今天需要了解的 **Map规约** 阶段。现在，我们再来看看前一篇博文《计数器与自定义计数器》中的第一张关于计数器的图：

我们可以发现，其中有两个计数器： **Combine output records** 和 **Combine input records** ，他们的计数都是0，这是因为我们在代码中没有进行Map阶段的规约操作。

1.2 为什么需要进行Map规约操作

众所周知，Hadoop框架使用Mapper将数据处理成一个个的<key,value>键值对，在网络节点间对其进行整理(shuffle)，然后使用Reducer处理数据并进行最终输出。



在上述过程中，我们看到至少两个 **性能瓶颈** ：

（1）如果我们有10亿个数据，Mapper会生成10亿个键值对在网络间进行传输，但如果我们只是对数据求最大值，那么很明显的Mapper只需要输出它所知道的最大值即可。这样做不仅可以减轻网络压力，同样也可以大幅度提高程序效率。

总结： 网络带宽严重被占降低程序效率；

（2）假设使用美国专利数据集中的国家一项来阐述 **数据倾斜** 这 个定义，这样的数据远远不是一致性的或者说平衡分布的，由于大多数专利的国家都属于美国，这样不仅Mapper中的键值对、中间阶段(shuffle)的 键值对等，大多数的键值对最终会聚集于一个单一的Reducer之上，压倒这个Reducer，从而大大降低程序的性能。

总结： 单一节点承载过重降低程序性能；

那么，有木有一种方案能够解决这两个问题呢？

二、初步探索Combiner

2.1 Combiner的横空出世

在MapReduce编程模型中，在Mapper和Reducer之间有一个非常重要的组件，它解决了上述的性能瓶颈问题，它就是 **Combiner** 。

PS：

**①与mapper和reducer不同的是，combiner没有默认的实现，需要显式的设置在conf中才有作用。**

**②并不是所有的job都适用combiner，只有操作满足结合律的才可设置combiner。combine操作类似于：opt(opt(1, 2, 3 ), opt(4, 5, 6))。如果opt为求和、求最大值的话，可以使用，但是如果是求中值的话，不适用。**

每一个map都可能会产生大量的 **本地输出** ， **Combiner** 的作用就是对map端的输出 **先做一次合并** ，以 **减少在map和reduce节点之间的数据传输量** ，以 **提高网络IO性能** ，是MapReduce的一种 **优化手段之一** ，其具体的作用如下所述。

（1）Combiner最基本是实现 **本地key的聚合，对map输出的key排序，value进行迭代** 。如下所示：

map: (K1, V1) → list(K2, V2)

combine: (K2, list(V2)) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)

（2）Combiner还有 **本地reduce** 功能（其本质上就是一个reduce），例如Hadoop自带的wordcount的例子和找出value的最大值的程序，combiner和reduce完全一致，如下所示：

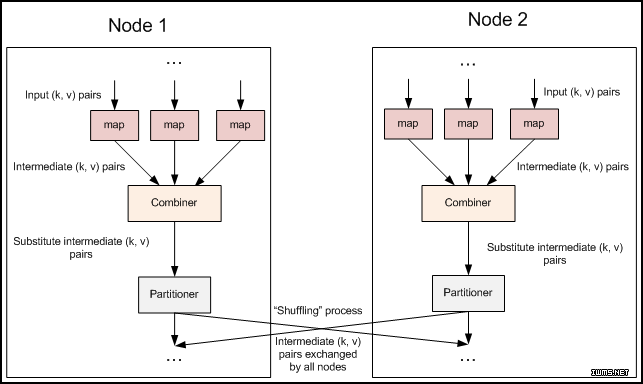
map: (K1, V1) → list(K2, V2)

combine: (K2, list(V2)) → list(K3, V3)

reduce: (K3, list(V3)) → list(K4, V4)

PS： 现在想想，如果在wordcount中不用combiner，那么所有的结果都是 reduce完成，效率会相对低下。使用combiner之后，先完成的map会在本地聚合，提升速度。对于hadoop自带的wordcount的例 子，value就是一个叠加的数字，所以map一结束就可以进行reduce的value叠加，而不必要等到所有的map结束再去进行reduce的 value叠加。

2.2 融合Combiner的MapReduce



前面文章中的代码都忽略了一个可以优化MapReduce作业所使用带宽的步骤— **Combiner** ，它在 **Mapper之后Reducer之前** 运 行。Combiner是可选的，如果这个过程适合于你的作业，Combiner实例会在每一个运行map任务的节点上运行。Combiner会接收特定节 点上的Mapper实例的输出作为输入，接着Combiner的输出会被发送到Reducer那里，而不是发送Mapper的输出。Combiner是一 个 ***“迷你reduce”*** 过程，它只处理 **单台机器** 生成的数据 **。**

2.3 使用MyReducer作为Combiner

在前面文章中的WordCount代码中加入以下一句简单的代码，即可加入Combiner方法：

// 设置Map规约Combiner

job.setCombinerClass(MyReducer.class);

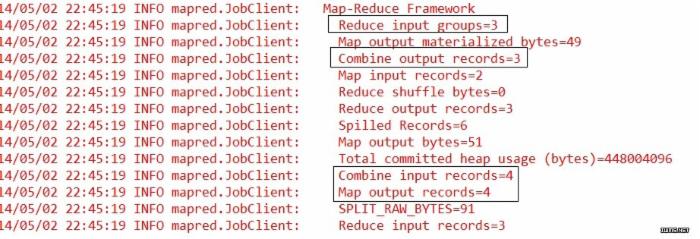
还是以下面的文件内容为例，看看这次计数器会发生怎样的改变？

（1）上传的测试文件的内容

hello edison

hello kevin

（2）调试后的计数器日志信息



可以看到，原本都为0的Combine input records和Combine output records发生了改变。我们可以清楚地看到map的输出和combine的输入统计是一致的，而combine的输出与reduce的输入统计是一样 的。由此可以看出规约操作成功，而且执行在map的最后，reduce之前。

三、自己定义Combiner

为了能够更加清晰的理解Combiner的工作原理，我们自定义一个Combiners类，不再使用MyReduce做为Combiners的类，具体的代码下面一一道来。

3.1 改写Mapper类的map方法

public static class MyMapper extends

Mapper<LongWritable, Text, Text, LongWritable> {

protected void map(LongWritable key, Text value,

Mapper<LongWritable, Text, Text, LongWritable>.Context context)

throws java.io.IOException, InterruptedException {

String line = value.toString();

String[] spilted = line.split(" ");

for (String word : spilted) {

context.write(new Text(word), new LongWritable(1L));

// 为了显示效果而输出Mapper的输出键值对信息

System.out.println("Mapper输出<" + word + "," + 1 + ">");

}

};

}

3.2 改写Reducer类的reduce方法

public static class MyReducer extends

Reducer<Text, LongWritable, Text, LongWritable> {

protected void reduce(Text key,

java.lang.Iterable<LongWritable> values,

Reducer<Text, LongWritable, Text, LongWritable>.Context context)

throws java.io.IOException, InterruptedException {

// 显示次数表示redcue函数被调用了多少次，表示k2有多少个分组

System.out.println("Reducer输入分组<" + key.toString() + ",N(N>=1)>");

long count = 0L;

for (LongWritable value : values) {

count += value.get();

// 显示次数表示输入的k2,v2的键值对数量

System.out.println("Reducer输入键值对<" + key.toString() + ","

+ value.get() + ">");

}

context.write(key, new LongWritable(count));

};

}

3.3 添加MyCombiner类并重写reduce方法

public static class MyCombiner extends

Reducer<Text, LongWritable, Text, LongWritable> {

protected void reduce(

Text key,

java.lang.Iterable<LongWritable> values,

org.apache.hadoop.mapreduce.Reducer<Text, LongWritable, Text, LongWritable>.Context context)

throws java.io.IOException, InterruptedException {

// 显示次数表示规约函数被调用了多少次，表示k2有多少个分组

System.out.println("Combiner输入分组<" + key.toString() + ",N(N>=1)>");

long count = 0L;

for (LongWritable value : values) {

count += value.get();

// 显示次数表示输入的k2,v2的键值对数量

System.out.println("Combiner输入键值对<" + key.toString() + ","

+ value.get() + ">");

}

context.write(key, new LongWritable(count));

// 显示次数表示输出的k2,v2的键值对数量

System.out.println("Combiner输出键值对<" + key.toString() + "," + count

+ ">");

};

}

3.4 添加设置Combiner的代码

// 设置Map规约Combiner

job.setCombinerClass(MyCombiner.class);

3.5 调试运行的控制台输出信息

（1）Mapper

Mapper输出<hello,1>

Mapper输出<edison,1>

Mapper输出<hello,1>

Mapper输出<kevin,1>

（2）Combiner

Combiner输入分组<edison,N(N>=1)>

Combiner输入键值对<edison,1>

Combiner输出键值对<edison,1>

Combiner输入分组<hello,N(N>=1)>

Combiner输入键值对<hello,1>

Combiner输入键值对<hello,1>

Combiner输出键值对<hello,2>

Combiner输入分组<kevin,N(N>=1)>

Combiner输入键值对<kevin,1>

Combiner输出键值对<kevin,1>

这里可以看出，在Combiner中进行了一次本地的Reduce操作，从而简化了远程Reduce节点的归并压力。

（3）Reducer

Reducer输入分组<edison,N(N>=1)>

Reducer输入键值对<edison,1>

Reducer输入分组<hello,N(N>=1)>

Reducer输入键值对<hello,2>

Reducer输入分组<kevin,N(N>=1)>

Reducer输入键值对<kevin,1>

这里可以看出，在对hello的归并上，只进行了一次操作就完成了。

那么，如果我们再来看看不添加Combiner时的控制台输出信息：

（1）Mapper

Mapper输出<hello,1>

Mapper输出<edison,1>

Mapper输出<hello,1>

Mapper输出<kevin,1>

（2）Reducer

Reducer输入分组<edison,N(N>=1)>

Reducer输入键值对<edison,1>

Reducer输入分组<hello,N(N>=1)>

Reducer输入键值对<hello,1>

Reducer输入键值对<hello,1>

Reducer输入分组<kevin,N(N>=1)>

Reducer输入键值对<kevin,1>

可以看出，没有采用Combiner时hello都是由Reducer节点来进行统一的归并，也就是这里为何会有两次hello的输入键值对了。

总结：从控制台的输出信息我们可以发现，其实combine只是把两个相同的hello进行规约，由此输入给 reduce的就变成了<hello,2>。在实际的Hadoop集群操作中，我们是由多台主机一起进行MapReduce的，如果加入规约 操作，每一台主机会在reduce之前进行一次对 **本机数据** 的规约，然后在通过集群进行reduce操作，这样就会大大节省reduce的时间，从而加快Map

## 2.5wordcount

public class WordCount {

public static void main(String[] args) throws Exception {

//配置属性

JobConf conf = new JobConf(WordCount.class);

conf.setJobName("wordcount");

conf.set("fs.defaultFS", "hdfs://192.168.198.130:9000");

//设置job到配置信息

Job job=Job.getInstance(conf);

job.setJarByClass(WordCount.class);

//设置mapper的配置信息:注意输入和输出与reducer的区别

job.setMapperClass(WordCountMapper.class);

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(LongWritable.class);

//设置mapper的输入信息

FileInputFormat.setInputPaths(job,new Path("/word/lamp.txt"));

//设置reducer的执行类

job.setReducerClass(WordCountReducer.class);

//reducer的输入输出信息

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(LongWritable.class);

//设置输出位置

FileOutputFormat.setOutputPath(job, new Path("/wordout"));

//提交作业信息

job.waitForCompletion(true);

}

}

public class WordCountMapper extends Mapper<LongWritable, Text,Text,LongWritable>

{

LongWritable out= new LongWritable(1);

Text text=new Text();

@Override

protected void map(LongWritable key, Text value,

Mapper<LongWritable, Text, Text, LongWritable>.Context context) throws IOException, InterruptedException

{

String line=value.toString();

String words[]=line.split(" ");

for(String s:words)

{

text.set(s);

context.write(text, out);

}

}

}

public class WordCountReducer extends Reducer<Text, LongWritable,Text, LongWritable>

{

LongWritable out= new LongWritable();

Text text=new Text();

@Override

protected void reduce(Text key, Iterable<LongWritable> value,

Reducer<Text, LongWritable, Text, LongWritable>.Context context)

throws IOException, InterruptedException {

//接收数据

//定义一个计数器

long counter=0;

//循环迭代values

for (LongWritable i:value) {

counter+=i.get();

}

text.set(key);

out.set(counter);

//输出

context.write(text, out);

}

}

## 2.6mapreduce命令

### 2.6.1用户命令

|  |  |  |
| --- | --- | --- |
| 命令 | 格式 | 说明 |
| archive |  | 创建hadoop存档 |
| archive-logs |  | 将YARN聚合日志组合到Hadoop存档中以减少HDFS中的文件数量的工具 |
| classpath | yarn classpath [--glob |--jar <path> |-h |--help] | 打印获取Hadoop jar和所需库所需的类路径 |
| distcp |  | 递归复制文件或目录 |
| job |  | 与mapreduce作业交互的命令 |
| pipes | mapred pipes [-conf <path>] [-jobconf <key=value>, <key=value>, ...] [-input <path>] [-output <path>] [-jar <jar file>] [-inputformat <class>] [-map <class>] [-partitioner <class>] [-reduce <class>] [-writer <class>] [-program <executable>] [-reduces <num>] | 管道工作 |
| queue | mapred queue [-list] | [-info <job-queue-name> [-showJobs]] | [-showacls] | 交互并查看作业队列信息 |
| version |  | 打印版本 |

### 2.6.2系统命令

|  |  |  |
| --- | --- | --- |
| 命令 | 格式 | 说明 |
| historyserver | mapred historyserver | 启动JobHistoryServer |
| hsadmin | maped hsadmin [-refreshUserToGroupsMappings] | [-refreshSuperUserGroupsConfiguration] | [-refreshAdminAcls] | [-refreshLoadedJobCache] | [-refreshLogRetentionSettings] | [-refreshJobRetentionSettings] | [-getGroups [username]] | [-help [cmd]] | 运行MapReduce hsadmin客户端以执行JobHistoryServer管理命令 |

## 2.7yarn命令

### 2.7.1用户命令

|  |  |  |
| --- | --- | --- |
| 格式 | 命令 | 说明 |
| application |  | 打印应用程序报告/杀死应用程序 |
| applicationattempt |  | 打印应用程序尝试报告 |
| classpath | yarn classpath [--glob |--jar <path> |-h |--help] | 打印获取Hadoop jar和所需库所需的类路径 |
| container | yarn container [options] | 打印容器报告 |
| jar | yarn jar <jar> [mainClass] args... | 运行一个jar文件。用户可以将YARN代码捆绑在一个jar文件中，并使用此命令执行它 |
| logs | yarn logs -applicationId <application ID> [options] |  |
| node |  | 转储容器日志 |
| queue | yarn queue [options] | 打印队列信息 |
| version |  | 打印Hadoop版本 |

### 2.7.2系统命令

|  |  |  |
| --- | --- | --- |
| 格式 | 命令 | 说明 |
| daemonlog |  | 获取/设置由守护程序中由限定类名称标识的日志的日志级别。 |
| nodemanager |  | 启动NodeManager |
| proxyserver |  | 启动Web代理服务器 |
| resourcemanager |  | 启动ResourceManager |
| rmadmin |  | 运行ResourceManager管理客户端 |
| scmadmin |  | 运行共享缓存管理器管理客户端 |
| sharedcachemanager |  | 启动共享缓存管理器 |
| timelineserver |  | 启动TimeLineServer |