Hadoop分布文件系统源代码详细解析

# 一、基本元数据

下图是HDFS中基本元数据的类结构:



## 1.1 Block

在HDFS中，以Block为单位抽象数据块，其中包括Block、BlockInfo及BlockMap等。

### 1.1.1 Block

Block是HDFS数据块中最基本的抽象，Block实现了Writable接口，可以序列化，同时实现了comparable接口，按照blockId大小排序。

1)成员变量

Block类主要有以下三个字段：1. blockid唯一的标示这个block， 2. numBytes是这个数据块的大小（单位是字节），3. 时间戳（联系GenerationStamp这个类）

2)方法

方法包括3个属性的get/set()，序列化反序列方法，以及一些工具方法（eg : isBlockFilename – 是否是合法的block文件名）

### 1.1.2 BlockInfo

在FB HDFS中有三个类中定义了BlockInfo内部类，分别是BlockMap,Placementmonit

or及DirectoryStripeReader。但是BlockMap中定义的BlockInfo是数据块管理的核心类，其扩展了Block类，下面对这个类进行详细的介绍：

1)成员变量

InodeFile inode:保存了该Block所属的inode，但是在hadoop 2.2后该字段为BlockCollection bc

Object[] triplets:该字段是BlockInfo的核心字段，triplets数组有3\*replication个元素（replication是block副本的个数）。

假设i为第i个保存有该block副本的datanode，那么triplets[3\*i]为第i个datanode的DatanodeDescriptor对象，triplets[3\*i+1]为该datanode上前一个blockInfo对象，triplets[3\*i+2]为该datanode上后一个blockInfo对象。（这么做的原因是更加的节省内存，方便通过datanodeDescriptor找到该节点上的所有blockInfo。如果使用LinkedLists双向链表存储这些数据，每个块副本需要42个字节，如果使用triplets只需要16个字节，同样完成双向链表功能）。

int checksum：

LightWeightGSet.LinkedElement nextLinkedElement:

2)方法

BlockInfo当中的方法大多数维护数据结构triplets，例如增加datanodde，或者修改prev\next blockinfo。

在Hadoop 2.2中增加了BlockInfoConstruction方法，将一个Block转换成正在创建的block。

3) Raid相关成员变量、方法

CheckBlockInfo(b,replication,checkExistence):常规文件用不到这个方法，但是对于hardlink文件及outside of loading，需要使用这个方法。

在这个方法中，如果Block(b)是RaidBlockInfo类，则为blockInfo创建RaidBlockInfo对象。

4) 类RaidBlockInfo

RaidBlockInfo继承了BlockInfo，增加了一个block索引index，该索引用于在blocks array中快速查找。

前面介绍了BlockInfo中triplets字段的使用，其中查找类似于链表，查询速度慢，因此增加了index可以直接定位。

### 1.1.3 BlockMap

BlockMap类的作用是存放Block->BlockInfo(Block元数据)的对应关系。BlocksMap类很简单，但是namenode当中维护block元数据的类，无论是获取某个数据块对应的INode以及数据块所在的datanode，都需要通过BlocksMap类。在系统中BlockInfo信息不断更新中。

1) 成员变量

Gset<Block,BlockInfo> blocks：类中使用了GSet，提供类似映射功能的集合 FSNameSystem ns;

2)方法

BlocksMap当中的方法大多也都是维护blocks这个字段

3) Raid相关成员变量和方法

在前两小节中，已经包含了BlocksMap当中的Raid相关内容

## 1.2 INode

FSDirectory维护HDFS文件系统的目录树结构以及hdfs文件同数据块之间的对应关系。在Linux中inode保存了Linux文件的元信息，如文件类型、所有者标识、文件长度及数据块索引等。HDFS中的Inode抽象了hdfs中的文件和目录，hdfs文件用InodeFile类来表示，hdfs目录用InodeFileDirectory来表示，InodeFileDirectoryWithQuIta表示带文件配额的目录

### 1.2.1 INode

INode是整个INode体系的根接口，保存了HDFS目录和文件的共同属性

1)成员变量

name : 文件\目录所属用户名,文件\目录所属组名等

modificationTime : 上次修改时间

accessTime : 上次访问时间

permission : 访问权限

INodeDirectory parent:父类的inode引用

INodeType:将Inode分为4类，Regular\_inode,HARDLINKED\_INODE,RAIDED\_INOD

E,HARDLINKE\_RAID\_INODE

2) 方法

Inode中存在大量上述属性的get和set方法。当然还有其他的方法，例如判断方法，isDirectory,isQuotaSet,isRoot及isUnderConstruction等

3) raid相关成员变量和方法

在inodeType中增加了RAIDED\_INODE和HARDLINKED\_RAID\_INODE等类型。

4)其他

在INode中实现了内部类DirCounts：用于保存namespace consumed和diskspace consumed参数：

long nsCount;

long dsCount;

### 1.2.2 InodeFile

扩展自Inode，表示一个文件，当中存储了Block信息

1)成员变量

header:文件头，使用这个来保存replication的多少以及数据块的大小（处理类类似于Inode当中的permission字段），前16比特位冗余备份的数目，后48bit为块大小，使用内部类HeaderFormat处理。

storage:存储了文件对应的块列表(blockInfo，数据块的元数据)，重要的值

2)方法

大多数的方法，是对Header和storage的操作

3)Raid相关方法和变量

INodeFile(id,permissions,blklist,replication,modificationTime...,codec):初始化

InodeFile，但是参数中如果RaidCodec不为空，则创建InodeRaidStorage，否则创建InodeRegularStorage。

getBlockReplication(BlockInfo block):storage的类型如果是InodeRaidStorage则特殊处理

RaidBlockInfo getFirstBlockInStripe(Block block,int index):返回storage中第一个stripe中的第一个Block

### 1.2.3 InodeDirectory

其实是一个目录的inode，抽象了hdfs当中的目录

1)成员变量

除了继承inode父类当中的parent 以外，多了一个List类型的字段children，存放目录中所有的孩子节点inode的集合。

List<Inode> children;

在类中定义了ItemCounts内部类，用来保存该目录下blocks、files及directories的数目。

ItemCounts itemCounts;

2)方法

相对于Inode添加了对于children字段的操作，增、删、改、查、例如addchild(),

removeChild(),replaceChild(),以addChild()为例，将Inode节点加入children的list当中：

3)Raid相关的成员变量和方法

目录当中，没有与raid相关的成员变量和方法

### 1.2.4 InodeDirectoryWithQuota

InodeDirectoryWithQuota是InodeDirectory的子类，具有配额限制的目录Inode实现

1)字段

nsQuota表示的是命名空间的配额

nsCount表示命名空间大小

dsQuota表示磁盘空间的配额

diskspace 表示磁盘空间大小

这里的quota都是通过构造方法或者set方法获得的， 而count则是通过inode的抽象方法spaceConsumedInTree获得当前inode占用的命名空间以及磁盘空间

2) 方法

添加了一个verifyQuota方法，如果超出了配额的限制，那么则抛出异常DSQuotaExceededException。同时提供了一些关于配额（quota）的增删改查操作

### 1.2.5 InodeFileUnderConstruction

表示正在处理当中 InodeFile （当HDFS打开一个文件进行写操作时，这个文件就处于UnderConstrution状态），是 InodeFile 的子类（同理的还有BlockinfoUnderConstruction）

1)成员变量

clientName : 写操作租约的持有者

clientMachine : 写操作客户端主机

clientNode : 如果客户端是集群当中的一个节点（例如在文件备份的时候，写操作是由datanode进行的）

2)方法

将一个InodeFileUnderConstruction转换为一个InodeFile，其实就是重新构造了一个inode，然后更改它的accessTime

setLastBlock && removeLastBlock (为什么只能对最后一个block进行操作呢 ? )(问题2)(注意这里还有一个BlockInfoUnderConstruction)

之所以这样是因为HDFS file只支持在文件末尾添加数据

## 1.3 Storage

### 1.3.1 InodeStorage

InodeStorage是一个抽象类，用于保存一个InodeFile中的BlocksInfo信息，这个类是为了支持Raid而出现的。

1) 成员变量

BlockInfo[] blocks:保存InodeFile的blocks信息

2) 方法

类中的方法用于blocks的操作，包括getBlocks,checkBlock,appendBlock,addBlock,

removeBlock等

3)Raid相关方法及变量

为了支持Raid，在该类中增加枚举类型storage\_type：{REGULAR\_STORAGE,RAID\_STORAGE}

INodeRaidStorage convertToRaidStorage(BlockInfo[] partityBlocks,RaidCodec codec,int[] checksums,BlocksMap blockMap，shortReplication,InodeFile inode)，该方法用于将InodeFile转化为Raid类型的File。

### 1.3.2 INodeRaidStorage

1) 成员变量

RaidCodec codec:该类用于保存与Raid操作相关的一些参数，可以当做一个工具类，用于实现InodeStorage的方法

2) 方法

由于常规文件与Raid file格式不同，因此对于Blocks的访问需要处理，在该类中大部分的操作是通过RaidCodec来完成：

例如：getSourceBlocs(){return codec.getSourceBlocks(blocks)}

其他的方法，getLastBlocks,getFileSize,diskspaceConsumed,isSourceBlock,getFirstBl

ockInStripe,convertToRaidStorage等。

3)其他

为了快速查询，在类中实现了RaidBlockInfo，该类扩展了BlockInfo，增加了Block索引。

### 1.3.3 INodeRegularStorage

该类扩展了INodeStorage，用于对BlockInfo[] blocks进行操作。该类中没有增加成员变量，仅是实现抽象类的InodeStorage，这里不再介绍。该Storage的类型为StorageType.

REGULAR\_STORAGE。

## 1.4 DataNodeDescriptor

在Namenode中使用DatanodeDescritor来描述一个datanode节点，DatanodeDescriptor扩展自DatanodeInfo类。

### 1.4.1 DataNodeID

唯一标识Datanode，通过<ip,port>以及StorageID进行标识

1) 成员变量

String name:hostname:portNumber

String storageID: 每个集群StorageID是唯一的

int infoPort: infoServer运行端口

int ipcPort: ipcServer运行端口

2) 方法

所有的方法和上面几种变量操作相关

### 1.4.2 DatanodeInfo

1) 成员变量

扩展自DatanodeID，添加了Datanode的一些简单的状态，如下所示：

private long capacity; // 容量

private long dfsUsed; // 使用的空间

private long remaining; // 剩余空间

private long blockPoolUsed; // 数据块池使用量

private long lastUpdate; // 上次更新时间

private int xceiverCount; // xceiver数量

private String location = NetworkTopology.DEFAULT\_RACK; // 地址

private String softwareVersion; // 软件版本

还有一个比较重要的字段， AdminState，标示当前datanode可能处于的状态。NORMAL – Datanode处于正常服务状态 ； DECOMMISSION\_INPROGRESS – Datanode处于下架状态中 (通过admin命令将指定datanode下架) ； DECOMMISSIONED – 已经下架。

2) 方法

基本上都是以上字段的set/get方法

### 1.4.3 DataNodeDescriptor

1)字段

BlockInfo blockList:描述当前DataNode内的所有数据块

heartbeatedSinceFailover:当NN出现失败时，将这个字段设置为false，当正常接收到blockReport时设置为true

blockContentsStale：namenode出现失败或者正在启动时，datanode可能挂起上一次Namenode发起的删除操作，设置datanode的状态为stale状态。

replicateBlocks ：要在datanode上备份的副本队列。会在心跳回复当中将这个队列带到datanode，datanode接受命令后，进行备份操作

recoverBlocks ：要在datanode上进行恢复的副本队列

invalidateBlocks ：要在datanode上进行删除的副本队列

2) 方法

addBlock()：将指定副本添加到这个datanode的数据块当中，同样的方法还有removeblock，replaceblock()等

updateHeartbeat()：有心跳更新时，HeartbeatMabager调用这个方法更新DatanodeDescriptor中的对应字段。同时将heartbeatedSinceFailover设置为true

markStaleAfterFailover() – 将当前DatanodeDescriptor，设置为stale状态。这个方法是当namenode发生active standby切换时，通过DatanodeManager调用的。

areBlockContentsStale() – 查询当前datanode是否为stale

3) Raid相关成员变量和方法

raidEncodingTasks:该dataNode需要完成的raid encoding task

raidDecodingTasks:datanode需要完成的decoding task

方法中增加了clearRaidEncodingTasks，addRaidEncodingTask，clearRaidDecodingTasks，addDecodingTasks,getRaidCommand。

4) 其他

在DataNodeDescriptor中定义了两个内部类，BlockQueue和BlockTargetPair BlockQueue：用于保存需要进行操作的block，

BlockTargetPair用于保存block->datanodeDescritpor[] targets，block操作的辅助类

# 二、HDFS 数据管理

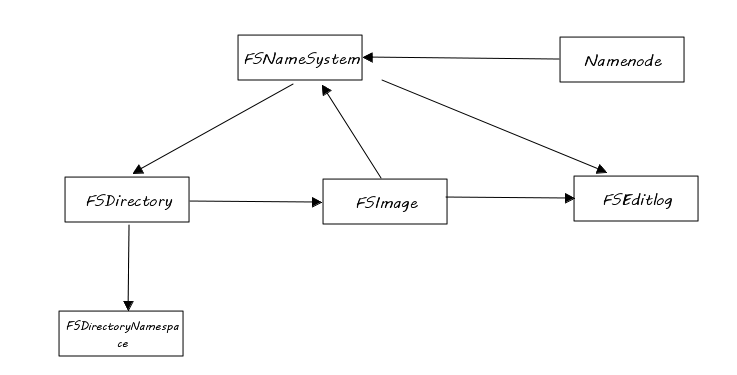
HDFS元数据由Namenode维护，目录和文件在内存当中以一棵树形式的数据结构存储，例如添加和删除文件的时候Namenode会对这个树形结构进行修改。在HDFS中每个文件被拆分成若干数据块，Namenode维护着文件同数据块的对应关系。

上述元数据和数据块的管理是通过NameNode中FsDirectory(org.apache.hadoop.hdf

s.server.namenode)来完成的，FsDirectory保存了目录树的根节点rootDir(INodeDirector

yWithQuota)。

下图是FSNameSystem、FSImage、FSEditlog和FSDirectory之间的关系：



namenode会定期将namespace(文件目录树、文件\目录元信息)保存到fsimage文件当中，如果每次将内存中的元数据保存到fsimage文件当中，将会非常耗费资源。因此namenode将近期进行的操作保存在fseditlog当中，然后定期合并fsimage以及fseditlog文件。

## 2.1 NameNode

前两节介绍了文件系统中的FSNameSystem和FSdirectory，FSnameSystem大部分功能都是为NameNode服务，NameNode是HDFS中文件目录和文件分配的管理者。NameNode中两个重要映射是文件名->数据块（磁盘上），数据块->Datanode列表（通过DataNode上报来构建）。

下面介绍NameNode：Client和DataNode的数据访问都是通过NameNode实现，因此NameNode实现了接口ClientProtocol和DatanodeProtocol

### 2.1.1 成员变量

1) FSNameSystem nameSystem;

2) Server server:RPC服务器实例

3)Thread emptier:处理回收站的线程句柄

4)InetSocketAddress namenodeAdderss:Namenode的地址，包括IP地址和监听端口

### 2.1.2 方法

1) main(String argv[])

main()是Namenode的入口，在方法内调用createNameNode()

2) createNameNode(argv,null)

根据命令行参数，如果是FORMAT或FINALIZE（格式化或终止），调用对应的方法后退出，如果是其他的参数，将创建NameNode，构造函数中进行初始化（initialize()）。

3) initialize()

完成初始化NameNode的成员变量，包括创建RPC服务器，初始化FSnameSystem,启动RPC服务器和回收站线程。

FSNameSystem的初始化包括加载FSImage，设置系统为安全模式，其他各工作和HTTP服务器。

startDNServer()->dnprotocolServer.start()

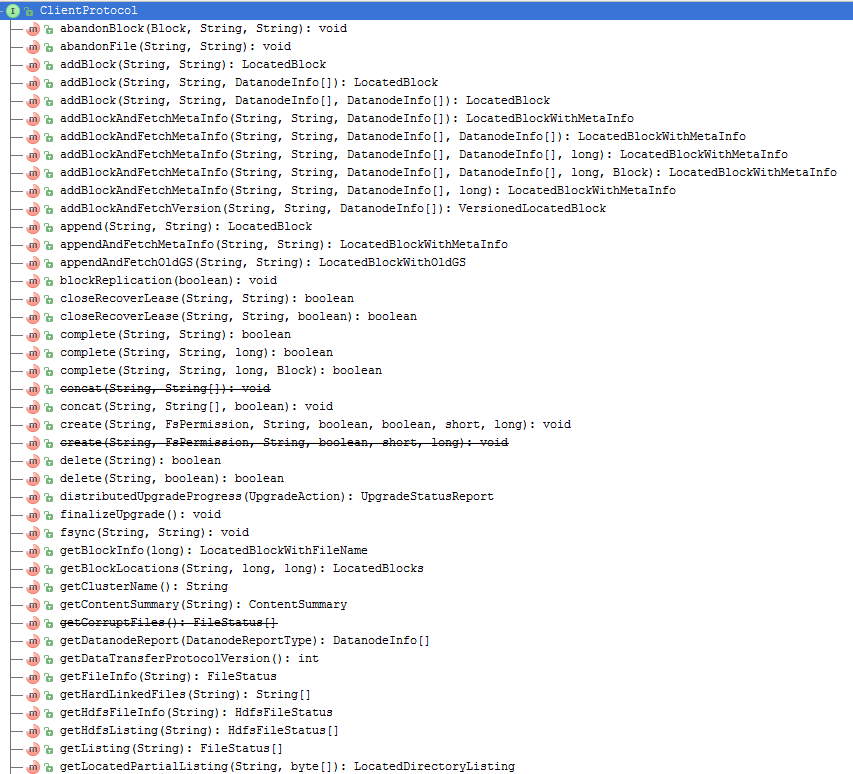
startHTTPserver()->httpServer.start()

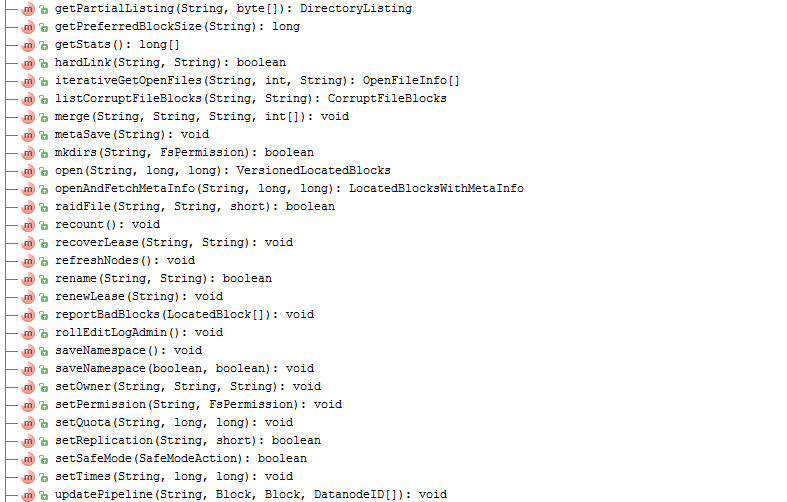
### 2.1.3 Client/Datanode与Namenode的通信

2.1.3.1 ClientProtocol

Namenode实现了ClientProtocol接口，客户端通过这个接口对目录树进行操作，打开和关闭文件等。

接口实现的操作如下：





下面介绍几种核心的方法：

1) LocatedBlocks getBlockLocations(src,long，long):

调用FSNameSystem的同名方法，首先检查权限和设置参数，然后找到src对应的INode，通过INode的getBlocks方法，获取节点的Block列表。最后获取Block对应的Datanode。

2) create(src,FSpermission,clientName,overwrite,createparent,replication,blockSize)

通过FSnameSystem的startFile方法来实现，流程图如下所示：



3) appendFile(src,clientName)

appendFile流程如下图所示：



4)setReplication,setPermission,setOwner,setSafeMode及其他getxxx（）

这些方法都是通过FSnameSystem或者FSDirectory中的方法获取

5)Blocks相关操作

abandonBlock用于放弃一个数据块，普通的文件系统中没有放弃操作，但是在HDFS中出现放弃数据块的原因，在于客户端获取LocatedBlock后，打开一个Block的输出流，由于从DataNode出错到Namenode发现这个信息，需要时间，打开的输出流卡可能出错，这时客户端可以向NameNode请求这个数据块。

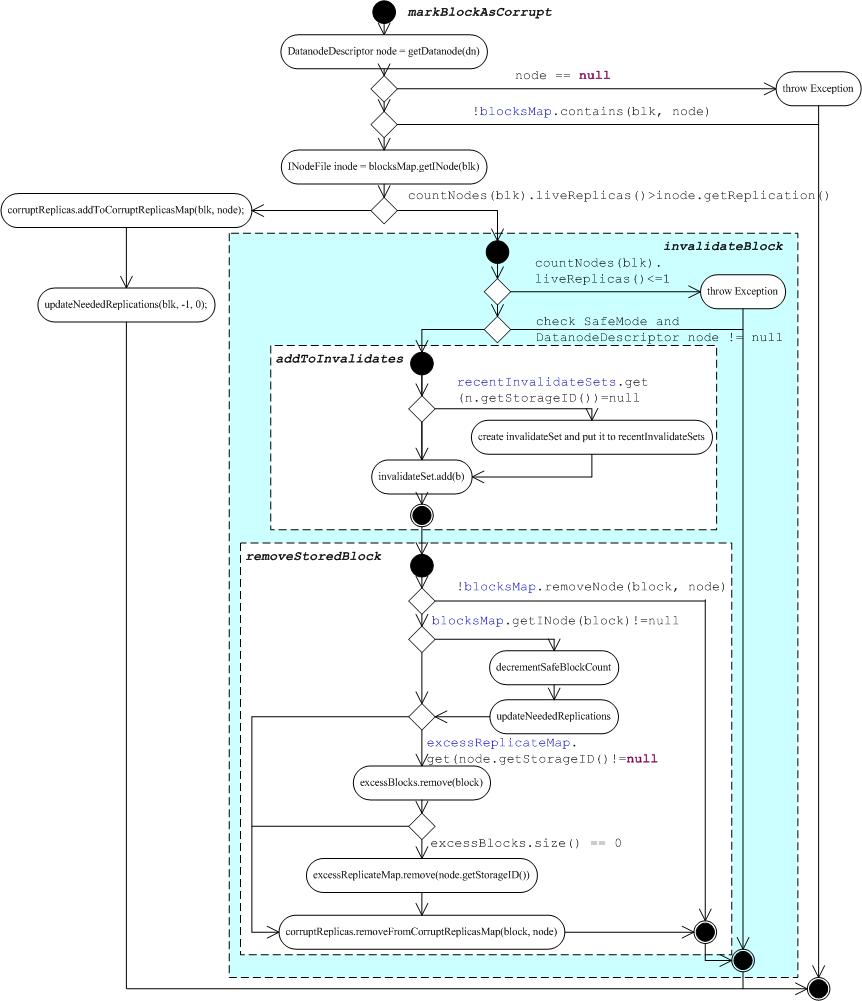
addBlock:如果数据块被写满，客户端可以通过addBlock创建新的数据块，具体的创建工作由FSNamesystem的getAdditionalBlock方法完成

complete:当客户端完成对数据块的写操作后，调用complete完成操作

reportBadBlocks：

removeStoredBlock:从BlocksMap中删除记录removeNode(block,node)，如果系统中有多余数据块等待删除，移除对应记录。删除在CorruptRreplacasMap中的记录。

markBlockAsCorrupt，流程如下图所示：



6)目录树相关操作

下面几个方法都是调用FSdirectory中的同名方法进行目录树的操作。

rename(src,dst):更改文件名

delete(src,recursive):删除文件

mkdir(src,FsPermission):创建文件夹

下面几个方法是系统维护管理的方法：

renewLease(string clientName):LeaseManager.renewLease(holder)

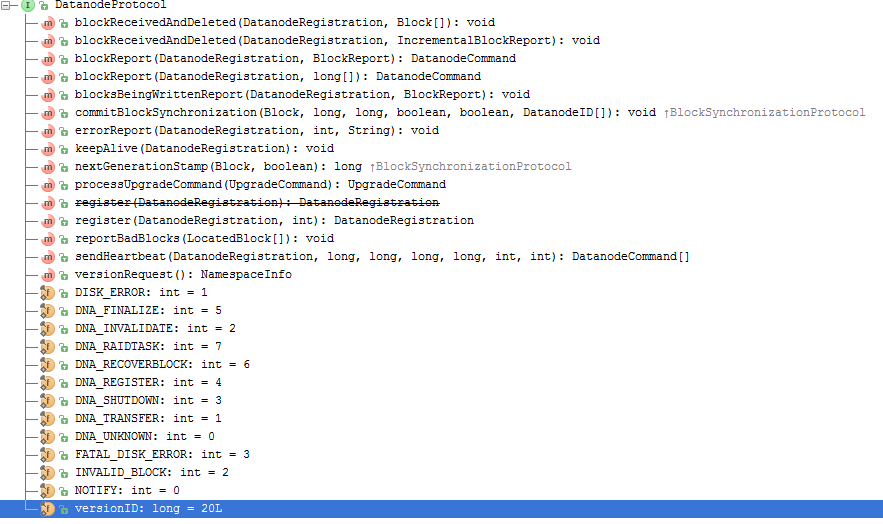
refreshNodes():重新读取datanode在线列表和Datanode离线列表

finalizeUpgrade():确认客户端有超级用户权限后，调用FSImage.finalizeUpgrad()

fsync(src,client): 调用FSdirectory.persistBlocks()

2.1.3.2 datanodeProtocol

datanodeProtocol的接口如下所示：



1)DatanodeRegistration register(DatanodeRegistration nodeReq)

用于Datanode向Namenode登记，输入和输出参数都是DatanodeRegistration，注册过程：更新请求的DatanodeID，从datanodeMap和Host2NodesMap获取DatanodeDescriptor（nodeS及nodeN）,先删除node，然后根据注册信息更新网络拓扑、心跳信息等。

2) DatanodeCommand sendHeartbeat(DatanodeRegistration nodeReg，....)

DataNode发送NameNode的心跳信息，请求的内容是DatanodeRegistraction，应答换成了DatanodeCommand(要重点介绍)。

DatanodeCommand支持的命令有DNA\_TRANSFER(拷贝数据块到其他DataNode)，DNA\_INVALIDATE(删除数据块)，DNA\_SHUTDOWN(关DataNode),DNA\_REGISTER:

Datanode重新注册，DNA\_FINALIZE:提交升级，DNA\_RECOVERBLOCK:恢复数据块

通过Namenode.sendHeartbeat获取命令后，通过processCommand处理cmds，代码如下：

private boolean processCommand(DatanodeCommand cmd, long processStartTime)

throws IOException {

if (cmd == null)

return true;

final BlockCommand bcmd = cmd instanceof BlockCommand? (BlockCommand)cmd: null;

boolean retValue = true;

long startTime = System.currentTimeMillis();

switch(cmd.getAction()) {

case DatanodeProtocol.DNA\_TRANSFER:

// Send a copy of a block to another datanode

transferBlocks(namespaceId,

bcmd.getBlocks(), bcmd.getTargets());

myMetrics.blocksReplicated.inc(bcmd.getBlocks().length);

break;

case DatanodeProtocol.DNA\_INVALIDATE:

//

// Some local block(s) are obsolete and can be

// safely garbage-collected.

//

Block toDelete[] = bcmd.getBlocks();

try {

if (blockScanner != null) {

blockScanner.deleteBlocks(namespaceId, toDelete);

}

data.invalidate(namespaceId, toDelete);

} catch(IOException e) {

checkDiskError();

throw e;

}

myMetrics.blocksRemoved.inc(toDelete.length);

break;

case DatanodeProtocol.DNA\_SHUTDOWN:

// shut down the data node

shouldServiceRun = false;

retValue = false;

break;

case DatanodeProtocol.DNA\_REGISTER:

// namenode requested a registration - at start or if NN lost contact

LOG.info("DatanodeCommand action: DNA\_REGISTER");

if (shouldRun) {

register();

firstBlockReportSent = false;

}

break;

case DatanodeProtocol.DNA\_FINALIZE:

storage.finalizedUpgrade(namespaceId);

break;

case UpgradeCommand.UC\_ACTION\_START\_UPGRADE:

// start distributed upgrade here

processDistributedUpgradeCommand((UpgradeCommand)cmd);

break;

case DatanodeProtocol.DNA\_RECOVERBLOCK:

recoverBlocks(namespaceId, bcmd.getBlocks(), bcmd.getTargets(),

processStartTime);

break;

case DatanodeProtocol.DNA\_RAIDTASK:

processRaidTaskCommand((RaidTaskCommand) cmd);

break;

default:

LOG.warn("Unknown DatanodeCommand action: " + cmd.getAction());

}

long endTime = System.currentTimeMillis();

if (endTime - startTime > 1000) {

LOG.info("processCommand() took " + (endTime - startTime)

+ " msec to process command " + cmd.getAction() + " from " + nnAddr);

} else if (LOG.isDebugEnabled()) {

LOG.debug("processCommand() took " + (endTime - startTime)

+ " msec to process command " + cmd.getAction() + " from " + nnAddr);

}

return retValue;

}

### 2.1.4 Raid相关成员变量与方法

1) String[] NAMESERVICE\_SPECIFIC\_KEYS

增加了DFS\_RAIDNODE\_HTTP\_ADDRESS\_KEY = dfs.raid.http.address(

FSConstants.java中定义)

2)RaidFile及raidFileInternal()(不再详细介绍)

3)public boolean raidFile(RaidFileRequest req)

类中增加了该方法，继承了clientProtocol中的方法

4)sendHeartBeat()

返回RaidCommand，Datanode根据方法调用返回的command，执行processRaidTasksCommand(),InjectionHandler.processEventIO(InjfectionEvent.DATAN

ODE\_PROCESS\_RAID\_TASK)。

要介绍InjectionHandler(单元测试用，JUnit)

## 2.2 FSNameSystem

FSNameSystem是NameNode实际记录信息的地方，HDFS文件系统实际执行的核心，提供各种增删改查文件操作的接口，保存在FSNameSystem中的数据有：

1) 文件名 —> 数据块列表（存放在FSImage和FEditlog）

2) 有效的数据块列表（上述数据的逆关系）

3) 数据块 —>DataNode的映射(只保存在内存中，根据Datanode发过来的信息动态建立)

4)DataNode上的数据块（上述数据的逆关系）

5)最近发送过心跳信息的DataNode(LRU)

### 2.2.1 成员变量

下面介绍几种核心的成员变量：

2.1.1.1 核心管理数据

1)FSDirectory dir: 系统使用的FSDirectory对象

2)BlocksMap blocksMap:Inode和DataNode的映射关系

3)Map<String,DatanodeDescriptor> datanodeMap:保存StorageID->DatanodeDes

criptor的映射，用户保证Datanode使用的Storage的一致性。

4) Map<String, LightWeightHashSet<Block>> recentInvalidateSets:每个DataNode上有效，但是需要删除的数据块（storageID->Treeset<Block>）

5)ArrayList<DatanodeDescriptor> heartbeats:目前活着的Datanode，线程HeartbeatMonitor会定期检查。

6) LeaseManager leaseManager: 租约管理器

7) Host2NodesMap host2DataNodeMap:保存了主机名到DatanodeDescriptor数组的映射

8) NetWorkTopology clusterMap:网络拓扑

1.1.1.2 线程

1) Daemon hbthread:心跳Monitor线程

2) Daemon lmthread: 租约Monitor线程

3) Daemon smmthread:安全模式线程

4) Daemon replthread: 数据块复制线程

5) Daemon raidEncodingRaskThread: raid任务线程

6) Daemon dnthread:用于检测DataNode上的Decommission线程，

### 2.2.2 FSNameSystem层次结构

FSnameSystem是HDFS系统执行的核心，不管是Client还是DN的消息最终会落到FSnameSystem中，对各种服务的请求处理都转交给它完成，这个数据结构维护了Namenode的元数据信息，下图是请求转交关系，它提供了对各种数据结构操作的接口：

上节介绍了几种核心的数据信息，下图示FSNameSystem的层次结构：



由上图可以看出，在内存中存在两种重要的映射表：

1）文件系统的命名空间，主要是保存在FSdirectory中的文件和Block映射关系（黑线）

2）Block和Inode&DataNode的映射关系，保存在FSnameSystem中（红线）

### 2.2.3 与Raid相关成员变量与方法

1) raidFile()

在FSDirectory中的raidFile介绍中已经做了详细的介绍

2)handlerHearbeat()

为了处理raid请求，增加raidCommand的处理：

cmd = nodeinfo.getRaidCommand(ReplicationConfigKeys.raidEncodingTaskLimit,

ReplicationConfigKeys.raidDecodingTaskLimit);

if (cmd != null) {

cmds.add(cmd);

}

在FDNameSystem中成员变量heartbeats用于Datanode与Namenode之间的信息交互，其处理流程如下所示：

1) DataNode向Namenode注册，数据结构为DatanodeRegistration nodeReg

2) namenode根据NodeReg获取datanodeDescriptor nodeinfo

3)从nodeinfo中获取命令，包括leaseRecoverCommand,replicationCommand,invalida

tedBlocks，raidCommand等

4)Namenode根据心跳信息中获取的command，进行下一步的操作

3) 内部类RaidEncodingTaskMonitor

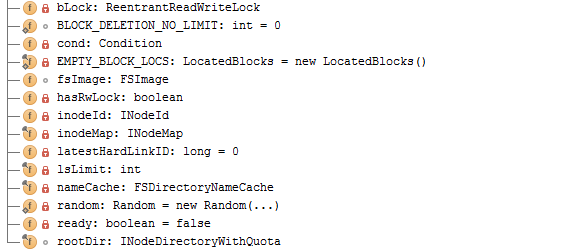
用于将NameSystem中接收的RaidEncodingTasks初步处理，将其放入raidEncodingTasks集合中。

## 2.3 FsDirectory

FsDirectory用于维护整个文件系统的目录树，NameNode中，对于hdfs整个文件系统的名字空间，通过FSDirectory来管理的，在目录树中不管是目录还是文件都被看做是INode。

### 2.3.1 成员变量

FsDirectory的成员变量如下所示：



INodeDirectoryWithQuota rootDir ： 这个是整个文件目录树的根节点，参考Inode章节内容

FSImage fsImage： FSDirectory操作目录树时，可通过FSImage提供的功能记录操作，参考FSEditLog以及FSImage章节内容

INodeMap inodeMap ： 维护Inode同InodeId之间的对应关系

ReentrantReadWriteLock dirLock ： 对于目录树以及inodemap操作的锁

FSNamesystem namesystem ： 整个namenode的门面，这里主要支持一些对于block进行操作的方法。例如addBLock()

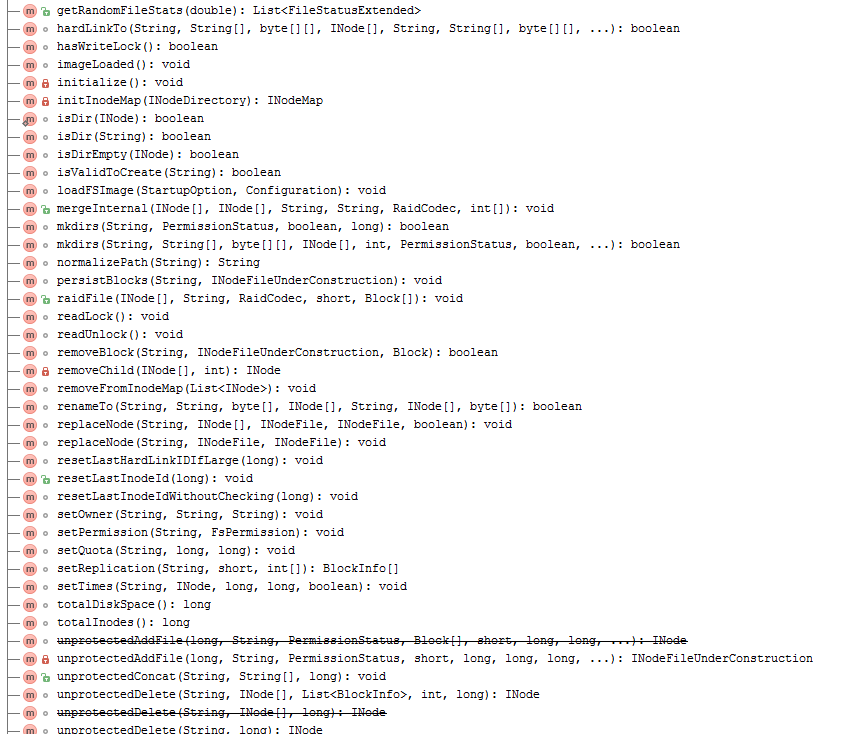
boolean ready ： 当namenode完成对fsimage以及editlog的加载之后，这个类变为true，表明可以对目录树结构进行操作

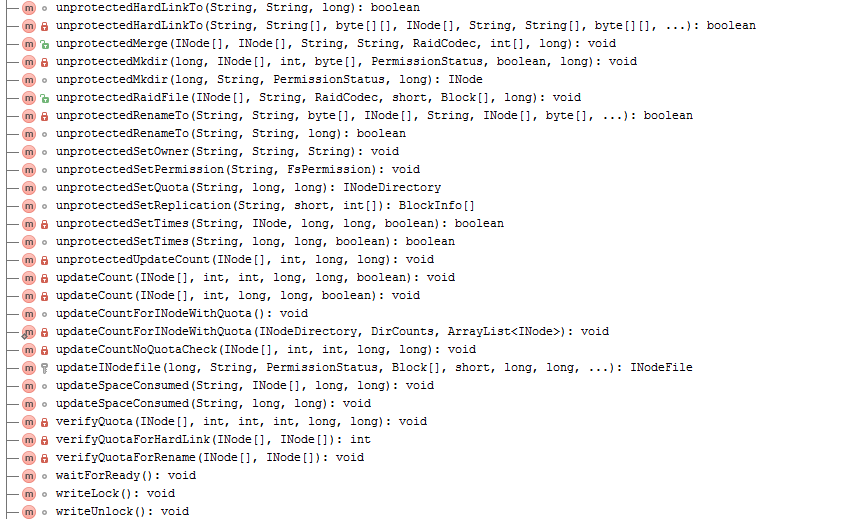
NameCache nameCache – 这里将常用的name缓存下来，以降低byte[]的使用，并降低heap使用

### 2.3.2 FSDirectory方法

方法如下所示：

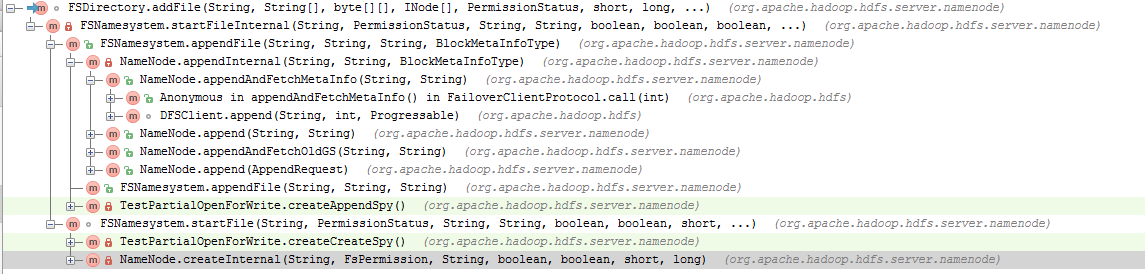






1) loadFSImage用于加载目录树结构，会调用FSImage的方法，完成持久化信息的导入，把成员变量ready置为true，系统调用LoadFSImage是在FSNamesystem.java的initialize方法。

2)addFile用于创建文件或追加数据时创建INodeFileUnderConstruction，下图是它的Call Hierachy图：



addFile首先会试图在系统中创建到文件的路径，如果文件为/home/hadoop/Hadoop.

tar，addFile会调用mkdirs（创建路径为/home/hadoop，这也会涉及到一系列方法），保证文件路径存在，然后创建INodeFileUnderConstruction节点，并把该节点加到目录树中（通过addNode，也是需要调用一系列方法），如果成功，就写操作日志（logOpenFile）。

3) unprotectedAddFile也用于在系统中创建一个目录或文件（非UnderConstruction），如果是文件，还会建立对应的block。FSDirectory中还有好几个unprotected\*方法，它们不检查成员变量ready，不写日志，它们大量用于loadFSEdits中（这个时候ready当然是false，而且因为正在恢复日志，也不需要写日志）。

4) addToParent添加一个INode到目录树中，并返回它的上一级目录，它的实现和unprotectedAddFile是类似的。

5) persistBlocks用于往日志里记录某inode的block信息，其实并没有一个对应于persistBlocks的写日志方法，它用的是logOpenFile。这个大家可以去检查一下logOpenFile记录的信息。closeFile对应了logCloseFile。

6) addBlock和removeBlock对应，用于添加/删除数据块信息，同时它们还需要更新FSNamesystem.java中对应的信息。

7) unprotectedRenameTo和renameTo实现了UNIX的mv命令，主要的功能都在unprotectedRenameTo中完成，复杂的地方在于对各种各样情况的讨论。

8) setReplication和unprotectedSetReplication用于更新数据块的副本数，很简单的方法，注意，改变产生的对数据块的删除/复制是在FSNamesystem.java中实现。

9)setPermission，unprotectedSetPermission，setOwner和unprotectedSetOwner都是简单的方法。

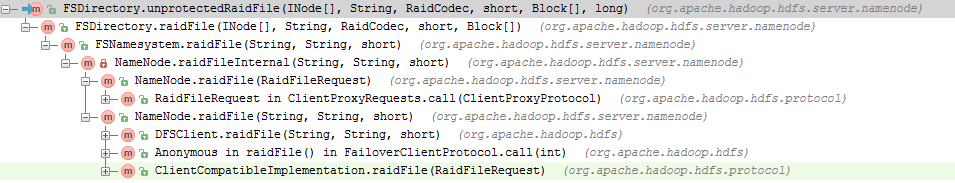
10) Delete和unprotectedDelete又是一对方法，删除如果需要删除数据块，将通过FSNamesystem的removePathAndBlocks进行。

### 2.3.3 Raid相关成员变量与方法

FSdirectory中与raid相关的方法有mergeInteral(),unprotecedMerge(),raidFile(),

UnprotecedRaidFile(),unprotectedsetReplication()，下面进行介绍。

1) unprotectedRaidFile()



当namenode接收到RaidFile请求（server线程或者DFSClient），以DFSclient为例，DFSclient.raidFile(source,codecId,short),参数source为文件路径，codecID为Xor或者RS（对应Codec）。

在方法内部调用namenode.raidfile(source,codeID,short)，该函数进行的调用内部方法raidFileInternal(source,codecID,short)。

namenode.raidFileInternal调用nameSystem.raidFile()，同时nanmenodeMetrics.nu

mraidFielOps.inc()(NameNodeMetrics)。

nameSystem.raidFile():

1) 判断source是否为空

2)获取RaidCodec,getCodec(codeID)

3)调用FSDirectory.getExistingPathInodes(source),从目录结构中查询特定的文件名或目录名，一步步的遍历，通过getchildNode获得是否存在查询的Inode对象，如果存在则返回string数据定义的文件名对应的Inode对象，否则返回null。

4)根据步骤3中返回的INode数组，找到数组中的InodeFile（sourceInodes[length -1]），获取其BlocksInfo[] blocks。

5) 如果sourceInode类型是Raid\_Storage，则判断其是否已经raid化（IndeRaidStorage.getCodec().id.equals(codec.id)）。

没有则RaidCodec检查其block数目足够raid化，(RaidCodec.checkRaidProgress)，同时初始化raidEncodingTasks（RaidBlockInfo set)。

6)如果是Regular\_storage，

分配校验块，

调用FSDirectory.raidFile(sourceInodes,source,codec,expectedSourceRepl,parity

Blocks)

6.1)调用unprotectedRaidFile() getFSnameSystem().blockMap.addInode(rawparityBlocks[i],sINode,CodecParityReplication),将parityBlocks添加入FSnameSystem中的blockMap中

将SInode的storage类型改变成raidStorage

6.2)fsimage.getEditLog().logRaidFile()

使用codec.checkRaidProgress(sourceINode，raidEncodingTasks)，将要raid的sourceInode加入raidEncodingTasks

7)RaidEncodingTaskThread（FSnameSystem中）.处理raidEncodingTasks，将RaidTask发送至对应的DataNode，DataNode调用processRaidTaskCommand进行数据raid化

2)raidFile()

FSNameSystem.addFile(source,codecID,expectedSourceRepl)，调用unprotectedRaidFile()方法。

上面文件raidFile流程图如下所示：

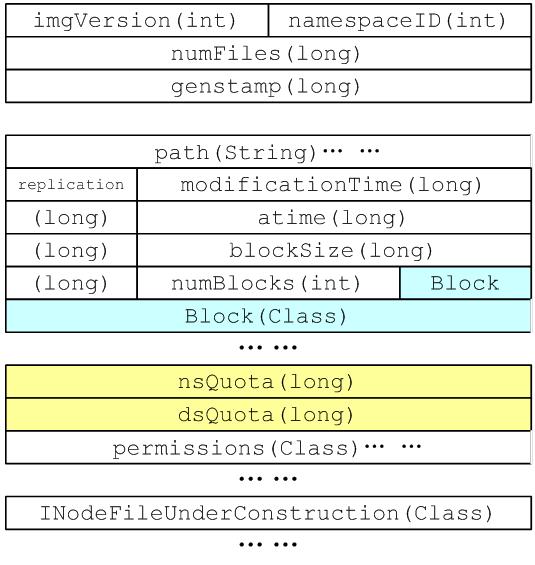


## 2.4 FsImage

文件名->数据块、目录树的更新和文件名->数据块关系的修改都必须持久化，为了保证每次修改都不需要重新保存整个结构，HDFS使用操作日志，保存更新。

与上面相关的是fsimage和FsEditlog，fsimage保存文件系统的目录树，edits是文件数的操作日志，fstime是上一次新打开一个操作日志的时间。下图是FSImage文件的格式：

最开始是版本号，然后是命名空间的ID号，文件个数和最高文件版本号，接下来是文件的信息（文件名、副本数、修改时间、数据块大小、数据块数目等）。如果数据块数目大于0，则接下来是numblocks个数据块，否则是目录。最后是文件访问控制的一些信息。



FsImage的主要操作：

保存namespace，将namenode内存当中的namespace保存到fsimage文件当中。加载fsimage，将磁盘上的fsimage文件保存的namespace加载到namenode内存当中。合并fseditlog以及fsimage。

### 2.4.1 成员变量

1) FSNamesystem nameSystem

需要进行持久化的FSNameSysetm

2)FSEditLog editlog

namenode将近期进行的操作保存在fseditlog当中，然后定期合并fsimage以及fseditlog文件。

3) ImageSet imageSet

提供Namenode上image locations的抽象，提供Image Managers(FSImageManager)工具类

4) NNStorage storage

用于管理Namenode中的storageDirectories

5)NNStorageRetentionManager

用于检查NN中的storage directories，在checkpoints和editlogs强制retention policy

6)SaveNamespaceContext saveNamespaceContext

用于Namespace操作的上下文环境

### 2.4.2 FsImage 方法

1) saveFSImage()

FSImage最重要的方法就是将namenode内存中的元数据保存到fsimage文件中。在这个方法中使用了fsimage工具类，读取和保存数据，并维护了fsimage的文件格式，工具类提供了两个内部类，Saver用于保存fsimage文件，Loader用于加载fsimage文件。

saveFSimage中调用了Saver中的save(outputstream,FSimageCompression,out,

name)，方法中的参数由FSimage类提供。

save方法按照image的格式写入数据，包括以下几个步骤：

(1)获取IO流

(2)写文件头

(3)写压缩信息

(4)保存目录树根节点，这里调用FSImageSerialization.saveInode2Image方法

(5)保存其他节点：调用saveChildern()，将目录树当中的所有节点写入文件（最终调用FSImageSerialization.saveInode2Image），如果子节点是目录，则递归调用saveImage方法。

(6)保存构建中的节点，调用FSnameSystem.saveFilesUnderConstruction方法

2)　LoadFSImage()

从磁盘上读取fsimage文件，并将元数据加载到namenode内存中，这个方法先加载fsImage，然后同editlog合并。 Namenode在初始化时创建FSnameSystem实例，然后通过FSdirectory.LoadFsImage来实现，最终的工作由FSImage.LoadFSImage来完成，加载流程如下：

1. 根据方法参数获取相应的ImageInputStream

2.LoadFSImage(iis,imageFile.getFile)->LoadFSImage(iis)

3.调用FSImageFormat.Loader.load(iis,null),获取输入流及digester

4.loadDatanodes和LoadFileUnderConstrunction及其他的数据信息

5.LoadFSimage会在所有的storage中读取最新的NameNode持久化信息

6.FSeditLogLoader.load方法，将editlog中记录的操作同当前的namespace进行合并

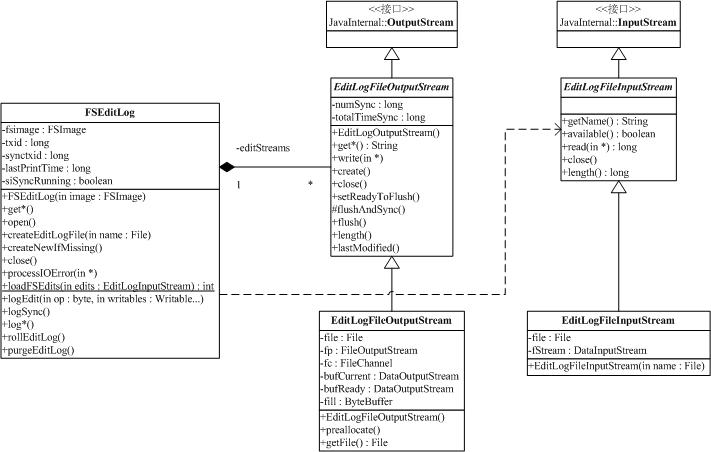
经过LoadFSimage后，内存中的目录树就是最新的。

### 2.4.3 与raid相关的成员变量和方法

FSimage当中没有与raid相关的方法和成员变量

## 2.5 FsEditLog

FSeditLog提供NameNode操作日志和日志文件的相关方法，相关类图如下所示：



FSEditLog依赖的输入/输出流。输入流基本上没有新添加功能；输出流在打开的时候，会写入日志的版本号（最前面的4字节），同时，每次将内存刷到硬盘时，会为日志尾部写入一个特殊的标识（OP\_INVALID）。

### 2.5.1 成员变量

1) NNStorage storage

2)FSnameSystem nameSystem

3)Configuration conf

4) Collection<URI> editDirs

5) EditLogFileInputStream和EditLogFileOutputStream

其中类中有个重要的变量EditDoubleBuffer，要将editlog操作写入editlog文件，首先要将操作写入outputstream的缓冲区当中，然后再flush到磁盘中。EditLogFileOutputStream中的缓冲区用到了一个 比较特殊的树结构，EditsDoubleBuffer。

这个缓冲区数据结构中有两块缓存，数据写入时，数据会先写入这个缓存中的一块缓冲区（另一块缓冲区可能正在进行向磁盘的同步操作-就是将内存中的文件写入磁盘）。这样的设计会保证 进行磁盘同步操作的同时，并不影响数据写入缓存。下面是EditsDoubleBuffer的具体实现。

### 2.5.2 方法

1)create()

create方法构造一个新的、空的editlog文件

public void create() throws IOException {

fc.truncate(0); //将文件大小设置为0

fc.position(0); //通道位置设置为1

writeHeader(doubleBuf.getCurrentBuf()); //向缓存写入文件头

setReadyToFlush(); //将缓冲区中的两块缓存互换

flush();//数据写入磁盘

}

2)write()

向doubleBuf缓存写入editlog操作，这里要注意操作是先写入缓存，然后通过同步操作写入磁盘。

public void write(FSEditLogOp op) throws IOException {

doubleBuf.writeOp(op);// 向doubleBuf写入FSEditLogOp

}

3)setReadyToFlush()

将doubleBuf缓存设置为ready状态 – 其实就是doubleBuf中的两块缓冲区互换，写入数据的缓冲区进行磁盘同步，原先磁盘同步的缓冲区换为写入数据部分。

public void setReadyToFlush() throws IOException {

doubleBuf.setReadyToFlush();

}

4) flushAndSync()

真正进行flush操作，将数据持久化到存储

public void flushAndSync(boolean durable) throws IOException {

// ..

preallocate(); // 如果editlog文件大小不够，则扩充文件大小

doubleBuf.flushTo(fp); //将buf中的数据写入通道

//..

}

5) preallocate()

这里我们看下preallocate()方法的实现

private void preallocate() throws IOException {

long position = fc.position();

long size = fc.size();

int bufSize = doubleBuf.getReadyBuf().getLength();

long need = bufSize - (size - position); //判断需要扩充容量的大小

if (need <= 0) {

return;

}

long oldSize = size;

long total = 0;

long fillCapacity = fill.capacity();

while (need > 0) {

fill.position(0);

IOUtils.writeFully(fc, fill, size); //将填充缓冲区写入通道，但不改变position，也就起到了将通道扩充的作用。

need -= fillCapacity;

size += fillCapacity;

total += fillCapacity;

}

}

7) 其他方法

logOpenFile（OP\_ADD）：申请lease

path(路径)/replication（副本数，文本形式）/modificationTime（修改时间，文本形式）/accessTime（访问时间，文本形式）/preferredBlockSize（块大小，文本形式）/BlockInfo[]（增强的数据块信息，数组）/permissionStatus（访问控制信息）/clientName（客户名）/clientMachine（客户机器名）

logCloseFile（OP\_CLOSE）：归还lease

path/replication/modificationTime/accessTime/preferredBlockSize/BlockInfo[]/permissionStatus

logMkDir（OP\_MKDIR）：创建目录

path/modificationTime/accessTime/permissionStatus

logRename（OP\_RENAME）：改文件名

src（原文件名）/dst（新文件名）/timestamp（时间戳）

logSetReplication（OP\_SET\_REPLICATION）：更改副本数

src/replication

logSetQuota（OP\_SET\_QUOTA）：设置空间额度

path/nsQuota（文件空间额度）/dsQuota（磁盘空间额度）

logSetPermissions（OP\_SET\_PERMISSIONS）：设置文件权限位

src/permissionStatus

logSetOwner（OP\_SET\_OWNER）：设置文件组和主

src/username（所有者）/groupname（所在组）

logDelete（OP\_DELETE）：删除文件

src/timestamp

logGenerationStamp（OP\_SET\_GENSTAMP）：文件版本序列号

genstamp（序列号）

logTimes（OP\_TIMES）：更改文件更新/访问时间

src/modificationTime/accessTime

### 2.5.3 raid相关的成员变量与方法

logRaidFile:未实现

# 三、DataNode

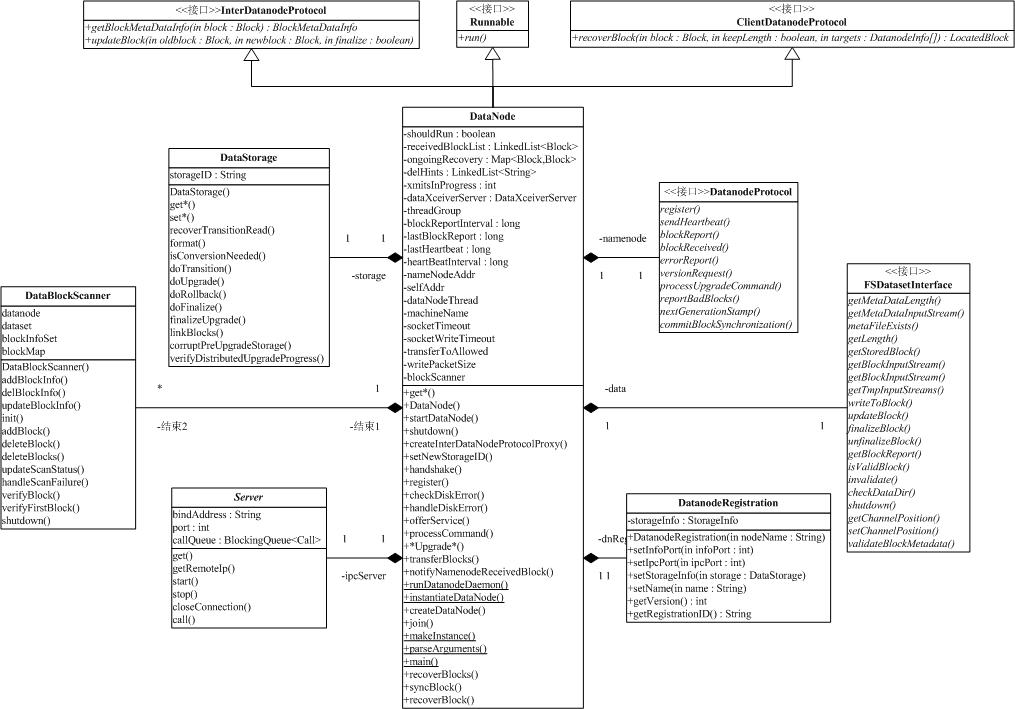
一个典型的HDFS系统包括一个Namenode和多个Datanode，NameNode维护命名空间，而DataNode存储数据块。前面两章介绍了NameNode的命名空间的管理，本章介绍DataNode。

DataNode负责存储数据，一个数据块在多个DataNode中有备份，而一个DataNode对于一个块最多只包含一个备份，所以可以简单的认为DataNode上存放了数据块ID和数据块的内容，以及它们的映射关系。一个HDFS集群可能包含上千个DataNode节点，这些节点DataNode定时和NameNode通信，接收Namenode的指令，但是为了减轻Namenode负担，Nmaenode上并不永久保存DataNode上那些数据块信息，而是通过DataNode启动时的上报，来更新NameNode上的映射表。

DataNode和Namenode建立连接以后，就会不断地和Namenode保存心跳，心跳的返回包含了Namenode对DataNode一些命令，例如删除数据块或者把数据块复制到另一个Datanode（NameNode不会发起DataNode请求）。DataNode当然也作为服务器接受来自客户端的访问，处理数据块读/写请求。DataNode之间还会互相通信，执行数据块复制任务，同时，在客户端写操作的时候，DataNode需要相互配合，保证写操作的一致性。下面对DataNode运行的类进行介绍

## 3.1 DataNode

DataNode的类图如下图所示：

****

DataNode实现了两个通信接口，其中ClientDatanodeProtocol用于和Client交互，InterDatanodeProtocol用于和Datanode通信。

Datanode从main函数开始，调用createDataNode方法，启动Datanode相关线程。CreateDataNode首先调用instantiateDataNode初始化DataNode，然后执runDatanode

Daemon向Namenode注册，注册成功启动DataNode线程。

1) 初始化的过程，读取Datanode需要的配置文件，及配置的storage目录，然后把这两个参数送到makeInstance中，检查参数后调用Datanode构造函数。

2)构造函数调用startDataNode，完成和Datanode相关的初始化工作，首先是初始化一堆的配置参数，Namenode地址，socket参数等，然后向Namenode请求配置信息（Datanodeprotocol,versionRequest），并检查返回的NamespaceInfo和本地的版本是否一致。（initConfig(),registerMXBean()）

3)创建DataXceiveServer，run方法里启动，创建DataBlockScanner（根据需要在offerservice中启动），创建httpservr和ipcserver。(initDataXceiver(),startInfoServer()

,initIpcServer())

4) 检查文件状态，并初始化FSDataset(initialDataSetAndScanner(conf,dataDirs))

在启动Datanode之前，Datanode需要向Namenode注册。注册信息在初始化时已经构造完毕，包括DataXceiverServer端口，ipcServer端口，文件布局版本号等。注册成功后启动Datanode线程。

DataNode run的循环有两种选择，升级/正常工作。offerService（内部类NNOfferservice）中，offerService定时向Namenode发送心跳，报告Block状态变化，报告Datanode现在管理的Block状态。发送心跳和Block状态报告时，Namenode会返回一些命名，Datanode会执行这些命令。

心跳和Block状态报告时Namenode向Datanode发起请求的唯一方法，命令如下：

DNA\_TRANSFER：拷贝数据块到其他DataNode

DNA\_INVALIDATE：删除数据块（简单方法）

DNA\_SHUTDOWN：关闭DataNode（简单方法）

DNA\_REGISTER：DataNode重新注册（简单方法）

DNA\_FINALIZE：提交升级（简单方法）

DNA\_RECOVERBLOCK：恢复数据块

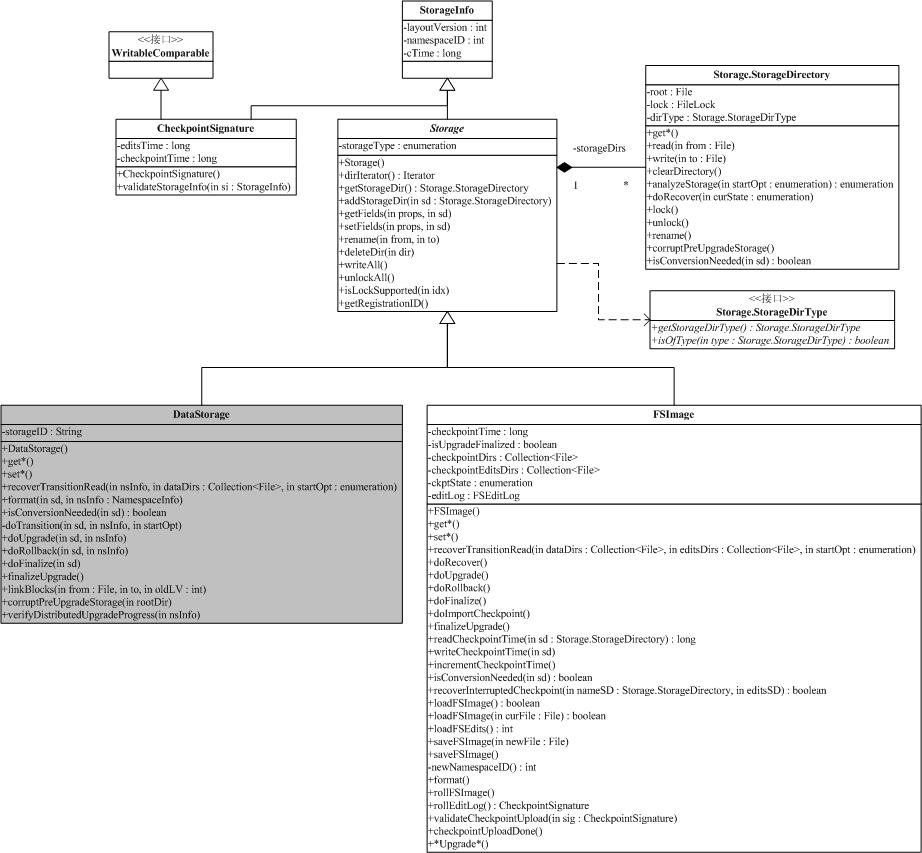
复制数据块到其他Datanode由transferBlocks方法执行，返回的命令包含多个数据块，每个数据块可以包含多个目标地址。transferBlocks为每个block启动一个DataTransfer线程，用于传输数据。DataTransfer是Datanode的一个类。

Block状态变化报告，会使用保存在receivedBlockList和delHints两个列表中的信息。receivedBlockList表明在这个Datanode成功创建的新的数据块，而delHints是可以删除数据块的节点。如在DataXceiver的replaceBlock中调用notifyNamenodeReceivedBlock

(block,sourceID)，这表明Datanode已经从sourceID上接收了一个block，sourceID上对应的Block可以删除。

## 3.2 DataStorage(DataNode节点的数据存储器)

对于每一个DataNode节点而言，都需要本地文件系统来存储与自己相关的数据，任何节点都可以配置多个本地存储路径，每个存储路径被HDFS抽象成一个StorageDirectory对象，但是为了DataNode节点对自己存储路径的管理，HDFS又为每一个DataNode节点设计了一个DataStorage对象，Datanode节点可以利用这个对象对所有的存储路径进行统一管理，下图是Storage相关的类图：



### 3.2.1 StorageInfo

1)成员变量

layoutVersion:版本号，如果Hadoop调整文件结构布局，版本号就会修改。

NamespaceID:Storage的ID

ctime:创建时间

StorageInfo中的方法大多数set/get变量的方法，因此不再介绍

### 3.2.2 Storage

Storage是Hadoop中的最大结构，继承了StorageInfo。Storage可以包含多个根（参考hdfs-site.xml中的配置项dfs.data.dir），这些根通过StorageDirectory（storage内部类）来表示。

3.2.2.1 StageDirectory

1)成员变量

File root； 根目录

final boolean userLock:启用目录锁的标志

FileLock lock:storage Lock

StorageDirType: storage dir type

2)方法

analyzeStore():

根据系统启动时的参数和判断条件，返回Storage Directory现在的状态，状态为StorageState:

public enum StorageState {

NON\_EXISTENT, //目录不存在

NOT\_FORMATTED, //目录存在但未被格式化

COMPLETE\_UPGRADE, //previous.tmp存在，current也存在

UPGRADE\_DONE,//

RECOVER\_UPGRADE,// finalized.tmp存在，current不存在

COMPLETE\_FINALIZE,//finalized.tmp存在，current存在

COMPLETE\_ROLLBACK,//removed.tmp存在，current存在，previous不存在

RECOVER\_ROLLBACK,//removed.tmp ,current不存在，previous存在

COMPLETE\_CHECKPOINT,//lastcheckpoint.tmp存在，current存在

RECOVER\_CHECKPOINT, //lastcheckpoint.tmp存在，current不存在

NORMAL,//普通工作模式

INCONSISTENT;

}

可以根据这些状态，进一步的对系统进行操作，如对系统进行恢复，系统升级等

docover():根据storage的状态，定义如何恢复

3.2.2.2 StorageState及StorageDirType

3.2.2.3 Storage成员变量和方法

1)成员变量

NodeType storageType:使用storage的节点类型

List<StorageDirectory> storageDirs;

2)方法：基本上是对StoregeDirectory的操作

### 3.2.3 DataStorage

DataStorage是Storage的子类，应用于DataNode，Datanode的升级、回滚和提交过程，就是对DataStorage的doUpgrade/doRollBack/doFinzalize分析得到的。

DataStorage提供了Format方法，用于创建Datanode上的Storage，同时利用StorageDirectory，DataStorage管理存储系统的状态。

1) 成员变量

String storageID:storage编号

DataNode, datanode:

Map<Integer,NamespaceSliceStorage> nsStorageMap:namespaceID和NamespaeS

torage的映射表。

2)方法

除了成员变量的set/get方法，下面介绍几种重要的方法：

ArrayList<StorageDirectory> analyzeStorageDirs(NamespaceInfo nsInfo,Collection

<File> dataDirs,StartupOption startOpt):检查路径，并返回StorageDirectory

void recoverTransitionRead(DataNode datanode,NamespaceInfo nsInfo,Collection

<File> dataDirs,StartupOption startOpt):分析storage 目录，并根据需求恢复到先前的状态

还有其他的方法，在再详细阐述

3)执行流程

下面分析DataNode启动时对Storage的操作，流程如下图所示：



1)在启动Datanode节点的时候，会指定若干本地存储路径，DataStorage对每个存储路径构造成一个StorageDirectory

2)分析每个存储路径当前状态

3)对于非正常状态的存储路径进行相应的操作(StorageDirectory中有详细分析)

4)正常状态的路径，进行用于启动节点时的指定操作（备份，升级，回滚及提交等）

5)操作成功，更新每一个存储路径下的版本信息，及对应的Version文件

注意：DataStorage只是对DataNode的存储路径进行管理，并没有对存储路径下中的具体数据文件进行管理

## 3.3 FSDataset

上面介绍Storage，并没有涉及到数据块的Block的操作，所有和数据块相关的操作，都在FSDataset相关的类中进行处理。下图是FSDataSet相关的类图：



在DataNode上可以指定多个Storage来存储数据块，HDFS规定了一个目录能存放的Blocks数目，所有一个Storage上存在多个目录。对应的FSDataset中的FSVolume对应一个Storage，FSDir对应一个目录，所有的FSVolume由FSVolumnSet管理，FSDataset中通过一个FSVolumnSet对象，就可以管理所有的存储空间。

关系如下：



### 3.3.1 DatanodeBlockInfo

DatanodeBlockInfo存放的是Block在文件系统上的信息，保存了Block存放的卷，文件名和detach状态，系统升级时会创建一个snapshot。

1) 成员变量

boolean detached; 文件detached状态

Block block;

BlockDataFile blockDataFile;存放block存放的卷、文件名

2) 方法

detachFile():改变current里的文件，如果不进行detach操作，那么修改的内容就会影响snapshot里的文件，需要将对应的硬链接解除掉。

在临时文件夹里，复制文件，然后将临时文件名改成current里的对应文件

detachBlock()：对硬链接进行操作，并调用detachFile()

3) BlockDataFile

该类保存了一个block块的File、所属的FSVolume及提供文件操作的File流

### 3.3.2 FSdir

FSDir对应着HDFS的一个目录，目录里存放着数据块文件和它的元文件。上节提到一个Storage上存在多个目录，所在的目录，都对应着一个FSDir。

1)成员变量

File dir；FSdir对应的文件

int numBlocks;目录中的blocks数目

FSDir childrenDirs[] ;目录的关系

2) 方法

getBlockInfo:分析目录下的所有数据块文件信息，生成Block对象，存放到一个集合中

FSDir(namespaceiD,dir,volumn)：建立block和DatanodeBlockInfo的关系

### 3.3.3 FSVolume

FSVolume对应着某一个Storage，数据块文件，detach文件和临时文件都是通过FSvolumn来管理的。

1) 成员变量

File currentDir: 对应/storageDirectory/current

FSDataSet dataset;

DF usage; disk space使用情况

DU dfsUsage;FileSystem disk space usge使用情况

ExecutorService nativeIOExecutor;线程，不断更新FSvolumn所在文件系统的剩余容量

### 3.3.4 FSVolumnSet

管理着所有的FSVolumn

1)成员变量

FsVolumnList volumeList;

int curVolume;

ExecutorService scannersExecutor;

2)方法

方法均是对VolumeList的操作

getDFSused：获取FSvolumn的dfs使用情况

getNSUsed:获取NS的使用情况

其他方法不再介绍

### 3.3.5 ActiveFile

保存了一个文件和操作这个文件的线程，线程很可能有多个。ActiveFile的构造函数会自动地把当前线程加入其中

DatanodeBlockInfo datanodeBlockInfo:

BlockCrcUpdater crcUpdater;

List<Thread> threads = new ArrayList<Threads>(2);

### 3.3.6 NamespaceSlice

表示存储在volumn上的namespace，集群中使用同一namespaceID的slice代表一个namespace。

int namespaceId;

FSVolume volume; //属于namespaceSlice的volume

FSDir dataDir;//storageDirectory /current/nsid/current/finalized

File detachDir: 存放finalizde replica的目录

File rbwDir: 存放RBW replica的目录

File tmpDir:存放Tempory replica的目录

NamespaceSliceDU; dfs 使用情况

recoverDetachedBlocks(): 用于恢复detach文件

3.2.7 FSDataset

1)成员变量

FSVolumeSet volumes; FSdataset使用的所有storage

Datanode datanode;

VolumeMap volumeMap;

2)方法

finalizeBlock(Block b):提交通过writeToBlock打开的block，把block从tmp文件夹放到current文件夹

updateBlock(oldBlock,newBlock):调用tryupdateBlock内部方法，更新数据块相关信息，包括元文件盒内存中的映射表。

BlockWriteStreams writeToBlock(block b,boolean isRecovery):得到一个block的输出流，包含数据输出流和元数据输出流。

## 3.4 DataXceiverServer及DataXceiver

前两节介绍了DataNode工作时的文件结构和文件结构在内存中的对应对象，下面分析DataNode上的数据块接受/发送并没有采用RPC机制，DataNode处理数据部分是一种流式机制。因此DataNode为了实现流式处理实现DataXceiverServer和DataXceiver，DataXceiver有两个辅助类：BlockSender和BlockReceiver。

下面是类图：

### 3.4.1 DataXceiverServer

用于接收和发送block数据，监听着client或者其他DataNode的请求，DataXceiverServer每接收到一个请求，就会创建一个DataXceiver来处理该请求。

1)成员变量

ServerSocket ss; datanode负责接收Socket连接请求

DataNode datanode;

Map<socket,socket> childSocket;记录为数据传输打开的所有Sockets

2) 方法

run():

DataXceiverServer继承Runnable的线程类，run是核心方法。每次执行，完成两个工作：

通过ss接口接收socket连接

启动线程DataXceiver处理这个线程（请求的实际处理线程）

kill():

杀死此线程，前提是确保数据节点已经关闭运行，然后关闭ServerSocket

### 3.4.2 DataXceiver

DataXceiver处理client或datanode的请求（DataTransferProtocol接口定义），下面具体介绍：

1)成员变量

Socket s;

String remoteAddress;

String localAddress;

Datanode datanode;

DataXceiverServer dataXceiverServer;

2)方法

DataXceiver是一个线程类，其核心方法：

run()

根据请求类型，进行不同的操作，类型种类如下：

OP\_READ\_BLOCK：读取数据块，调用方法readBlock(in)

OP\_WRITE\_BLOCK:写数据块，方法writeBlock(in)

OP\_REPLACE\_BLOCK:替换数据块，方法replaceBlock(in)

OP\_COPY\_BLOCK:拷贝数据块，方法copyBlock(in)

OP\_BLOCK\_CHECKSUM:读取数据校验码，getBlockChecksum(in)

OP\_READ\_BLOCK\_ACCELERATOR: readBlockAccelerator()

OP\_READ\_META:readMetadata(in ,versionAndopcode)

OP\_APPEND\_BLOCK:appendBlock(in,versionAndopCode)

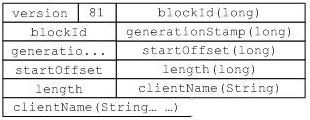
OP\_BLOCK\_CRC:getBlockCrc(in,versionAndopcode)

执行后，将socket请求，从dataXceiverServer.childSockets中移除

readBlock(DataInputStream in,VersionAndOpcode versionAndOpcode,reveiveStart

Time)

DataXceiver读取客户端的版本号并检验，然后再读取一个字节的操作码，并转入相关的子程序进行处理，下图是读数据块时，客户端发送过来的信息：



包括读取的BlockID，时间戳，开始偏移和读取的长度，最后是客户端的名字。根据这些信息，可以创建一个BlockSender。通过BlockSender来读取block信息，下面一节介绍BlockSender和BlockReveiver

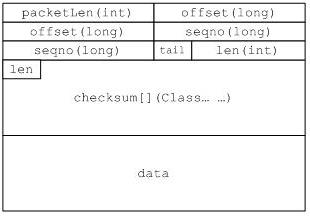
3)BlockSender

BlockSender的构造函数，根据需求，打开相应的数据流。

sendBlock用于发送数据，数据发送包括应答头和后续的数据包，应答头如下（包括DataXceiver中发送成功标识）：

a4dd8fdc-2b3f-34fc-81cc-24676d508728

后面的数据组织成数据包发送，包结构如下：



packetLen：包长度，包括包头  
offset：偏移量

seqno：包序列号

tail：是否是最后一个包

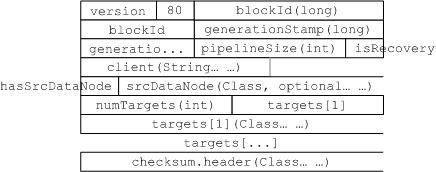
len：数据长度

checksum：检验数据

data：数据块数据

4)BlockReceiver

writeBlock(in)是HDFS中比较复杂的方法，通过BlockRecevier来完成，写数据块的请求如下：

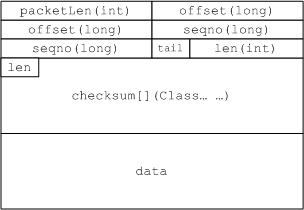


首先是客户端的版本号和一个字节的操作码，然后是BlocKId和generationStamp。pipelinesize是数据流链的长度，isRecovery指示写数据是一次恢复操作，client是请求发起端的名字，如果hasSrcDatanode被设置，表明源节点是个Datanode。numTargets是目标节点的数目，后面跟的是各个目标节点。

根据上面的消息包，创建一个BlockReceiver，接下来创建读写的流（在BlockReciever中创建）



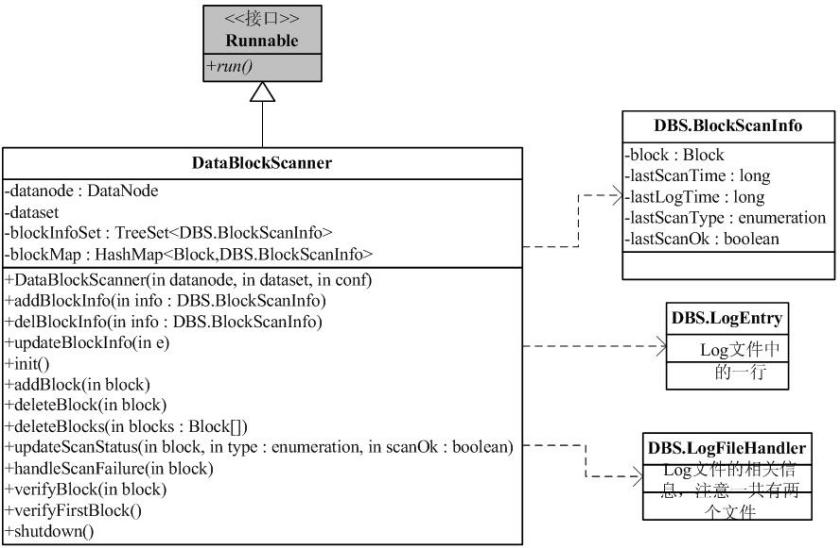
读写流创建后，由recieveBlock启动一个线程，用于处理应答，不断调用receivePacket读取数据。数据以分块的形式传送，格式和读取Block是已有的，如下图：



如果该Datanode处于数据流的中间，该数据包会发送到下一个节点，接下来是处理数据和校验，将数据写到数据文件和数据块元数据文件。

## 3.5 DataBlockScanner

DataBlockScanner用于定时对数据块文件进行校验，类图如下所示：



DataBlockScanner有单独的线程，定时从目前DataNode管理的数据块文件进行校验，最重要的方法是verifyBlock。

blockSender = new BlockSender(namespaceId, block, 0, -1, false,

false, true, datanode);

DataOutputStream out =

new DataOutputStream(new IOUtils.NullOutputStream());

blockSender.sendBlock(out, null, throttler);

校验利用了BlockSender，在发送数据的同时，会对数据进行校验。verifyBlock只需要读一个Block到一个空输出设置，如果异常则校验失败，正常则校验成功。

DataBlockScanner其他的辅助方法用于对DataBlockScanner管理的数据块文件信息进行增加/删除，排序操作。同时校验信息还会保持在storage上，保持在ncp\_block\_verificatin

.log.curr和dncp\_block\_verification.log.prev当中。

# 四、Client

客户端DFSClient和建立在DFSClient基础之上的DistributedFileSystem、DFSAdmin和FsShell，屏蔽了HDFS系统的复杂性，为应用提供了标准的Hadoop文件系统应用程序接口、文件系统Shell和管理工具。

## 4.1 DFSClient

DFSClient实现客户端功能的类，它能够连接到一个Hadoop文件系统并执行基本的文件任务，使用ClientProtocol和Namenode通信，并且使用Socket直接连接到Datanode来完成数据的读/写。

1)成员变量

ClientProtocol namenode;namenode实现clientprotocol接口，适配器模式

ClientProtocol rpcNamenode:

InetSocketAddress nameNodeAddr;

LeaseChecker leaseChecker;

Configuration conf;

SocketFactory socketFactory;

上述几个变量是DFSclient的核心变量，其他变量是一些参数及辅助实例（DFSClientMetrics）等

2)方法

DFSClient中的方法较多，下面介绍几种核心的方法：

1、DFSClient()：构造函数，DFSClient提供构造函数很多，最简单的形式只需要一个保存配置信息的Configuration对象参数。

构造方法的任务有两个：

(1) 读取配置项并初始化一些成员变量

(2) 建立和名字节点的IPC连接

2、DFSClient.close()：用于关闭客户端，设置clientRunning为false，中断DFSClient打开的输入流和输出流，停止租约管理器。

3、对于文件系统，分为文件和目录相关事务，以及文件数据读写两种情况。在DFSClient中，文件实现几乎都是调用namenode上的同名连接方法。

checkopen():判断DFSClient对象是否关闭

mkdirs(src,permission):直接调用namenode.mkdirs(src,permission)，进行目录的创建

delete(src,recursive):

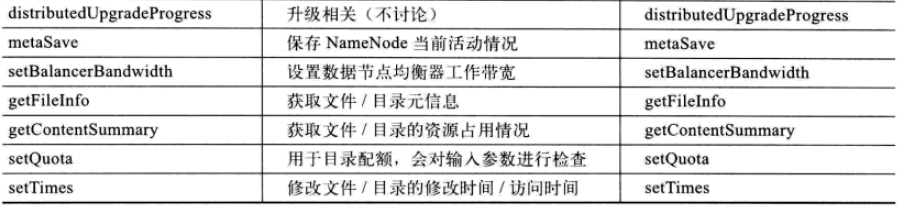
getFileInfo(src):

setOwner(src,userName,groupname):

这些方法都是通过namenode的同名方法来实现。

下图是总结DFSclient中这部分的方法功能和使用ClientProtocol远程方法：

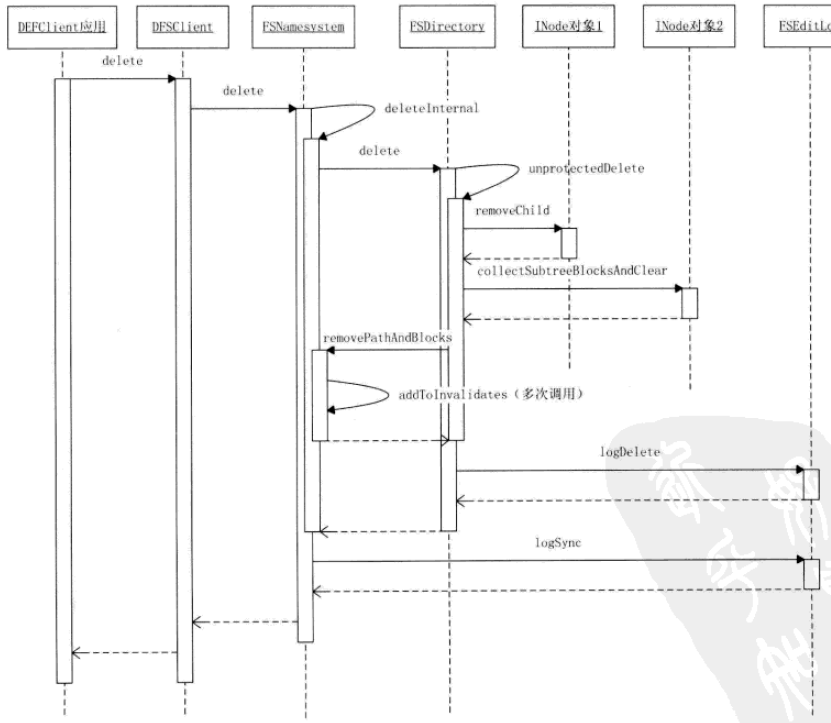




3) 删除、写入和读取文件解析

1.客户端调用DFSClient.delete

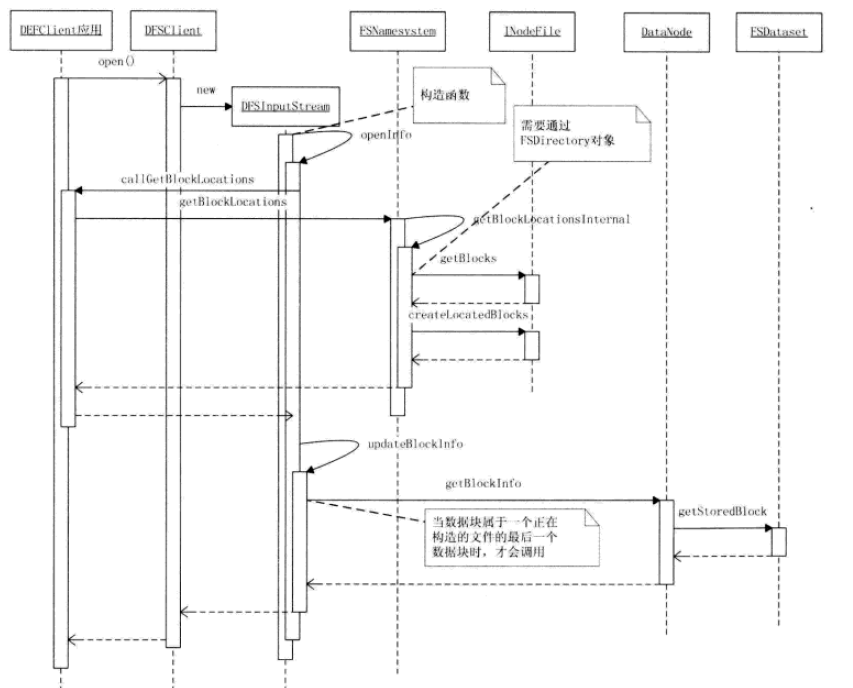
HDFS各个主要模块间的配合如下图所示：



客户端通过远程方法的实现NameNode.delete()，最终调用了FSNamesystem的同名方法。删除操作涉及名字节点第一关系和第二关系，通过FSDirectory对象在目录树中删除目标文件/目录，即图中INode.removeChild()调用，并在返回Inode对象上，通过collectSubtreeBlocksANdClear()方法获取等待删除的数据块，并使用名字节点第二关系removePathAndBlocks()和addToInvalidate()，将上述数据块放入FSNameSystem成员变量recentInvalidateSets中。由于删除操作修改了namenode节点中的信息，需要将操作记录到日志中，所以要经过FSEditLog对象的logDelete()和LogSync()调用。

2.读取数据

流程如下图所示：



1) HDFS打开一个文件，首先在客户端调用DistributedFileSystem.open(Path f,int buffersize)，实现如下：

public FSDataInputStream open(Path f, int bufferSize) throws IOException {

return new DFSClient.DFSDataInputStream(

dfs.open(getPathName(f), bufferSize, verifyChecksum, statistics));

}

其中dfs为DistributedFileSystem的成员变量DFSClient，其open函数别调用，创建一个DFSInputStream并返回。

起始：通过Java程序调用Hadoop提供的FileSystem的API，FileSystem提供各种HDFS操作接口，（DFSClient是DistributedFileSystem和DistribytedRaidFileSystem的成员变量）。FSDataInputStream继承了java.io.DataInputStream接口的一个类，由该类完成数据的读取过程。

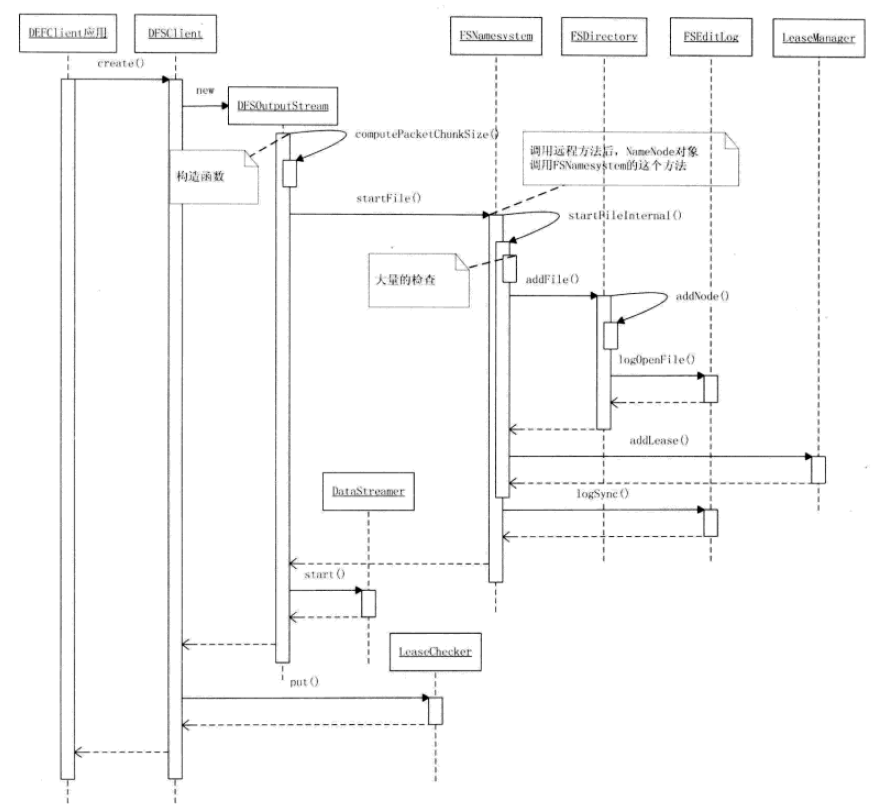
2)DistributedFileSystem通过RPC来调用namenode，以确定文件起始块位置，步骤2，RPC过程通过DFSClient类完成。Namenode.java

3) 步骤2中，返回一个文件的FSDataInputStream对象，客户端对这个输入调用read()方法

4) FSDataInputStream，随即连接距离最近的datanode，对数据流返回调用read方法，可以将数据从datanode传送至客户端。

3.写入文件

流程如下图：



应用通过DFSClient的create()方法创建一个新文件，获得一个输出流，输出流构造时，通过远程调用ClientProtocol.create()在namenode节点上创建文件。Namenode对象使用FSNamesystem.startFile()，最后调用startFileInternal()创建文件，根据输入参数进行检查后，使用FSdirectory.addFile()，最终由addNode()在目录树上添加一个INodeFileUnderConstruction对象，也就创建了文件。接下来，通过addFile()往日志中写入添加文件的记录。应用在获取DFSOutputStream对象后，通过write()输出数据。

## 4.2 DistributedFileSystem

Hadoop抽象了文件系统FileSystem，满足不同应用的访问需求，一个具体文件系统实现是DistributedFileSystem。Hadoop用户使用DistributedFileSystem实例，而不是直接使用DFSclient来执行文件任务，DistributedFileSystem使用DFSClient来处理文件系统任务，通过FileSyste.get()获取文件系统时，返回的是一个DistributedFileSystem。

DistributedFileSystem是适配器模式的一个典型应用，将DFSClient提供的访问HDFS的能力，包装秤FileSystem要求的界面。

1)成员变量

DFSClient dfs：文件访问的核心变量，操作都需要经过DFSclient

Path workingDir:文件系统的当前工作路径

URI uri;

2) 方法

initialize()：用于初始化对象，workingDirs,uri及dfs都在这个方法中被初始化，该方法也需要一个Configuration对象

delete()：删除文件，调用dfsclient的同名方法

open()：打开HDFS文件，返回一个FSDataInputStream对象，用于读取文件数据

create()：创建文件，返回FSDataOutputStream对象，用户创建文件

## 4.3 FsShell

Hadoop文件系统shell命令可以执行其他文件系统中常见的操作，如读取文件、移动文件、创建目录等。

例如，在终端可以通过下面的命令，hadoop fs -help获得shell命令

FSshell是一个Java程序，实现了应用入口的main方法，典型的基于ToolRunner实现的应用。

4.3.1 GenericOptionsParser

用于解释常用的Hadoop命令行选项，根据Hadoop配置configuration对象设置相应的配置项，一般在ToolRunner内部调用GenericOptionsParser。

4.3.2 FsShell支持命令

hadoop shell -help可以看到支持的命令，在FsShell中可以看到同名的方法，这些方法通过DistributedFileSytem中的方法实现

## 4.4 DFSAdmin

DFSAdmin继承自FsShell，实现和FsShell类似，也是通过ToolRunner.run()执行，该方法根据不同的命令调用相应的处理函数。

最终的实现还是通过DistributedFileSystem来完成。

## 4.5 Raid相关方法和变量

DistributedRaidFileSystem和RaidShell是Raid HDFS重要的客户端，将在以后进行详细介绍，这里只是简单的阐述。

### 4.5.1 DistributedRaidFileSystem

DistributedRaidFileSystem继承FilterFileSystem抽象类，FilterFileSystem实现接口FileSystem

FilterFileSystem的成员变量为FileSystem在创建实例是进行赋值，FilterFileSystem方法实现都是调用FileSystem的同名方法，没有进行改变。

1)DistributedRaidFileSystem初始化

根据conf，进行初始化Codec

根据配置文件中的fs.raid.underlyingfs.impl，获取所使用的文件系统，没有配置则使用DistributedFileSystem。

DistributedRaidFileSystem进行文件操作的方法实现，通过调用DistributedFileSystem的方法，同时进行ParityFile的处理。

### 4.5.2 RaidShell

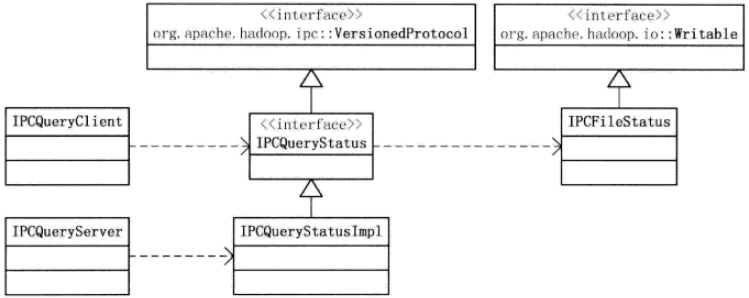
RaidShell和FSshell不同,FsShell直接使用DistributedFilesystem的方法即可，但是DistributedRaidFileSystem并没有提供很多可用的方法，因此RaidShell通过与RaidNode来实现操作，因此过程复杂。

# 五、Hadoop Common

Hadoop作为分布式文件系统，各个实体之间存在着大量的交互过程。RPC(远程过程调用)就是可以允许用户像调用本地方法一样，调用另外一个应用程序提供的服务。所以Hadoop提供了自己的RPC框架，动态代理以及protobuf等基础技术。

## 5.1 RPC使用实例

RPC是允许程序调用位于其他机器上的过程（可以使同一台机器的不同进程），机器A上的进程调用机器B上的进程时，A上的调用进程被挂起，而B上的被调用进程开始执行。调用方使用参数将信息传送给调用方，然后通过传回的结果得到信息。这个过程对于编程人员来说是透明的。Hadoop的远程过程调用没有使用Java RMI，而是实现了自己的节点通信机制，从而精确的控制进程间通信中的连接、超时、缓存等通信细节。下面介绍如何使用Hadoop IPC来构建简单的分布式系统，类图如下所示：



Hadoop IPC必须继承自VersionedProtocol接口：

public interface VersionedProtocol{

public long getProtocolVersion(String protocol,long clientVersion) throws IOException;

}

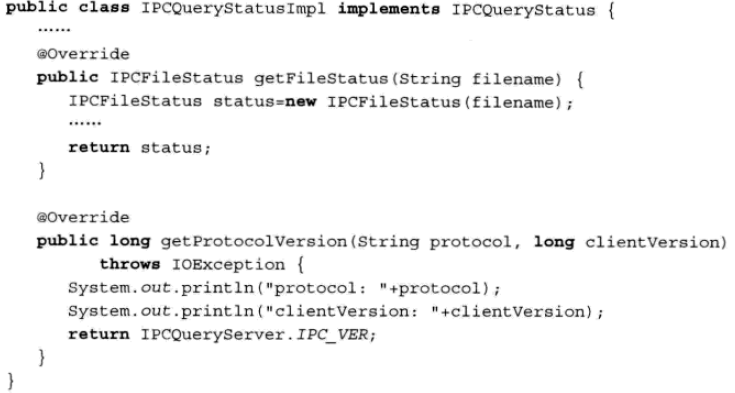
IPCQueryStatus接口定义，如下：

public interface IPCQueryStatus extends VersionedProtocol{

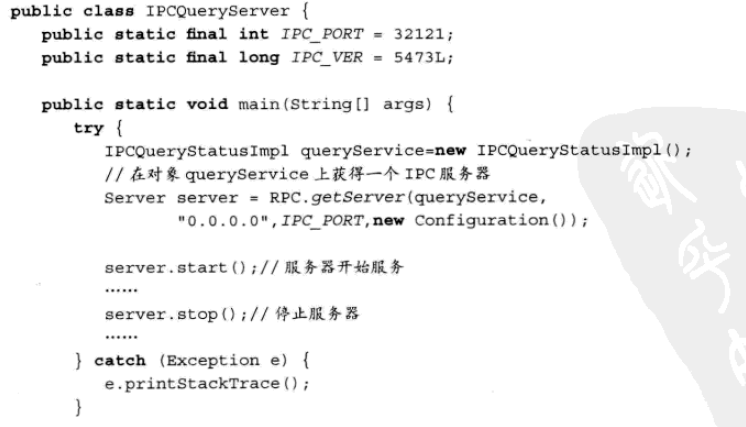
IPCFileStatus getFileStatus(String filename);

}

IPCQueryStatus的一个实现是IPCQueryStatusImpl，代码比较简单，根据getFileStatus的输入参数构造一个IPCFileStatus对象，然后返回该对象。



下面是以实现了IPC接口的IPCQueryStatusImpl类为基础，搭建IPC服务。Hadoop IPC建立服务器的方法非常简单，通过IPC的静态方法getServer()，就可以在一个IPCQuery StatusImpl实例上，获得一个提供IPCQueryStatus接口功能的IPC服务器，服务实例通过start()和stop()方法启动和停止



RPC.getServer共有4个参数，分别是IPC接口的实现对象、监听地址、端口和Hadoop的配置类Configuration的实例。监听地址和端口形成IPC服务器的Socket地址，Conf设置一些IPC服务器的参数。

访问IPC服务器的客户端代码如下：



RPC.getProxy()用于获得一个IPC客户端的实例，需要四个参数，IPC接口的类对象，接口版本，服务器的Socket地址和配置类Configuration对象。

上例Client通过端口调用IPCQueryStatus.getFileStatus("/tmp/testIPC")来查询/tmp/testIPC的状态。

1)创建IPCQueryStatus实例（IPCQueryStatusImpl继承其IPCQueryStatus）queryService

2)使用RPC.getServer(queryService,"0.0.0.0",IPC\_PORT,config)，获取对象queryService的一个IPC服务器，返回一个Server实例，对象至少实现一个IPC接口，可以实现多个IPC就接口，例如Namenode。

3)Client，通过RPC.getProxy(IPCQueryStatus.class,IPCQueryServer.IPC\_VER,addr

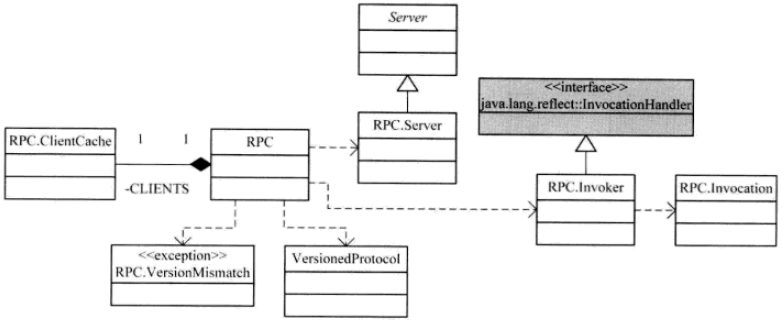
new Configuration)，或得接口IPCQueryStatus的一个实例query

4)通过步骤3中的query实例，调用远程方法，query.getFileStatus(src)

5)使用结束后，停止RPC.stopProxy(query)

## 5.2 RPC 类分析

Hadoop中与客户端、服务器实现的代码主要在Client.java、Server.java和RPC.java三个文件中，类结构如下图所示：



### 5.2.1 VersionedProtocol

在RPC中，在服务器上注册的类必须实现VersionedProtocol接口。该接口有一个get

ProtocolVersion()方法，该方法的参数为接口名称protocol和客户端期望的服务器的版本号clientVersion,方法返回服务器端的接口实现的版本号。

### 5.2.2 Connection

IPC主要有两类流程，分别是连接相关过程和IPC方法调用相关过程，下面介绍IPC连接过程，IPC连接分为Client和Server连接。

#### 5.2.2.1 Client.Connection

Client是IPC客户的抽象，内部类分为IPC连接相关的如Client.connection

Client.ConnectionID等。

IPC客户端在调用服务器之前，需要和服务器建立一个连接，IPC连接建立在TCP连接之上，为了提高通信效率，客户端可以服用服务器连接，通过连接标识ConnectionID区分不同的连接

ConnectionID类中成员变量如下：

1)address:远程服务器的地址，类型是InetSocketAddress，包括主机名和服务器的监听地址

2) UserGroupInformation: 用户和用户组的一些信息

3) Class<?> protocol，保存了IPC接口对应的类对象

ConnectionHeader，与IPC连接的另一个类是连接消息头ConnectionHeader，是客户端与服务器连接建立后交换的第一条信息，携带的内容包括ConnectionID中的用户信息和IPC接口信息。

InetSocketAddress server: 服务器的IP:Port

Socket socket: 客户端与服务器连接socket

DataInputStream in:

DataOutputStream out:

int rpcTimeout: 每个RPC的最大等待时间

HashTable<Integer,Call> calls: 当前正在处理的远程调用

常用方法：

sendParam(call,fastProtocol):调用sendParamInternal，使用DataoutputStream输出数据流（即call数据）

receiverResponse():从DatainputStream输入流中读取相应信息，接收从服务器端发送过来的应答，分别读取call status和响应值value，最后将响应的call删除

addCall(call)，将新创建的call加入Calls中

run()：connection对象，一直运行，查看receiverResponse()

#### 5.2.2.2 Server.Connection

实现的IPC服务器端的抽象类，和Client类似也有对应的IPC连接、远程调用Call相关的类，Server.Connection的成员变量比Client.Connection多。

SocketChannel channel:

volatitle int rpc:当前正在处理的RPC请求量

Socket socket:

String hostAddress:主机名

int remotePort:端口

ConnectionHeader header:

还有一些是配合通道工作的一些缓冲区，如data,dataLengthBuffer，dataLength等，连接状态机及一些检查量。

#### 5.2.2.3 IPC连接的建立

1)client IPC连接

Client与Server建立连接时，通过调用Client.getConnetcion()方法，首先检查HashTable<ConnectionID,Connection> connections中是否有连接（连接复用），如果有复用连接，否则创建新的IPC连接。

getConnection()参数包括ConnectionID，需要远端服务器的地址、用户信息和远程接口信息，最后一个参数是Call，其类型是Client.Call。

setupIOstreams()，通过该方法和服务器建立连接，首先使用Java基本套接字和服务器建立Socket连接，接着调用writeHeader()和IPC服务器进行握手，最后是启动线程，该线程用于在Socket上读取并处理响应数据。

2) server IPC连接

服务器端建立IPC连接的代码分散在Listener和Server，Connection中。

Listener在构造函数中打开服务器的端口，创建Selector并开始监听。Listener.run()实现了NIO的选择器循环，调用选择器的select方法并处理时间, doAccept方法接受客户端的连接请求，注册Socket到选择器并创建Server.Connection对象。

doRead()通过Server.Connection.readAndProcess()读取并处理数据，问下问题通过closeConnection()关闭连接。

Server.Connection中最主要的方法是readAndProcess()，doRead()调用 这个方法读取客户端发送过来的数据并处理。这个方法包括两个步骤：

1)读入ConnectionHeader，保证服务器实现了IPC接口和获取用户信息

2)调用authorize() ，保证用户有相关的访问远程端口

#### 5.2.2.4 数据分帧和读写

IPC连接涉及两个方向的数据分帧，IPC客户端请求用的是显式长度方法，长度是Int。

在IPC客户端，Connection.sendParam()用于将远程调用Call形成消息发送出去。sendParam()方法将要发送的数据写入DataOutputBuffer对象中，同时获取数据的长度和消息内容，然后通过输出流out发送数据。

服务器端接收数据在Connection.readAndProcess()，读取显式长度使用的是缓冲区datalengthBuffer，长度是4字节，即一个整数的长度。

但是服务器到客户端用的定长消息的变种，即利用前面介绍的Writable发送相应，发送Writable时，通信双方不需要定界符，也不需要显式长度，其本质上定长消息，由客户端Writable对象保证读入服务器端写入的所有数据。服务器端往客户端写数据的代码在Server.setupResponse()方法中，客户端读数据代码在Connection.receiveResponse()方法中。

#### 5.2.2.5 连接的维护和关闭

IPC连接长时间没有发生远程调用，客户端会发送一个特殊的心跳信息给服务器，用户维护客户端/服务器间的连接，上次发生IPC通信的时间记录在Connection的成员变量lastActivity中Connection.touch()更新lastActivity当前时间，在setupIOstreams和receiveResponse中被调用，sendping()也会更新lastActivity。

当用户通过Client.stop()关闭整个客户端，或者IPC连接出现错误，或者IPC连接上长时间没有发送IPC调用时，必须关闭连接。IPC连接错误，调用Connection.markClosed()方法设置成员变量shouldCloseConnection，并通知连接执行关闭动作。其他的关闭方法，有Connection.close()等，不再介绍。

### 5.2.3 Call

本小节在IPC连接基础上，调用远程方法。IPC方法调用，在客户端只是比较复杂的Java动态代理应用。在服务器端，由Listener、Handler和Responser配合，完成请求读取，请求处理和请求应答三个步骤。IPC发生时会涉及3个用户抽象调用的类：RPC.Invo

cation,Client.Call和Server.Call。

#### 5.2.3.1 RPC.Invocation

IPC客户端的调用句柄被捕获之后，会生成RPC.Invocation实例，包括调用上下文的方法名称、形式参数列表和实际参数列表。

String methodName;

Class[] parameterClasses;

Object[] parameters;

RPC.Invocation不包含调用上下文的接口名称，连接头中已经包含了接口名称。RPC.Invocation实例被创建后，需要通过IPC连接传送到服务器。这个过程需要添加一些IPC连接处理需要的信息，形成一个Client.Call对象。

#### 5.2.3.2 Client.Call

Client.Call类的成员变量：

int id:标识符

Writable param:RPC.Invocation对象

Writable value:返回值

IOException error:异常返回时的异常

boolean done;该调用是否已经完成

由于连接共享，一个连接上可能有多个IPC调用正在执行

#### 5.2.3.3 Server.Call

Client.Call通过IPC连接发送到Server侧，形成一个Server.Call对象，Server.Call的成员变量和Client.Call类似

int id:标识符

Writable param:RPC.Invocation对象

Connection connection:IPC连接

long timestamp: 用于超时检查

ByteBuffer response;应答

Connection是服务器端的IPC连接抽象，用于发送应答，保存在response的调用结果，需要通过connection发送客户端。

#### 5.2.3.4 RPC调用过程

应用程序调用getFileStatus()，开始了一个客户端远程方法调用的处理过程，流程如下图所示：



由上图，RPC调用分成两个部分，客户端调用和Server端调用：

1) 客户端调用

应用程序调用getFileStatus()(前面讲述的例子)，开始了一个客户端远程方法调用的处理过程。这个过程的入口是RPC.Invoker.invoke()，RPC.invoker类实现了动态调用转换接口，java.lang.refelet.InvocationHandler。

Invoker构造函数中首先通过ClientCache.getClient()方法获得该成员变量，根据method和args参数构造Invocation对象，接着调用Client.call方法发送IPC请求并获取结果，结果通过value.get()返回给调用者， 在Client.call方法中，通过sendParam把调用相关的信息发送给服务器，客户端开始等待服务器发送回来的应答。

2) server端调用

服务端收到sendParam发送过来的参数，通过Listener接收。Listener.doRead()方法读取数据，在Connection.readAndProcess()中恢复数据帧，然后调用processData()方法。构造服务器端Call，将该对象放入callQueue队列中。

Listener和Handler是典型的生产者-消费者，processData()将待处理的数据放入队列中，由Handler调用对象的服务器方法。Handler.run方法中请求通过Server.call()完成服务端方法调用，返回Writable调用结果，Server.call是通过Subject.doAs()完成。

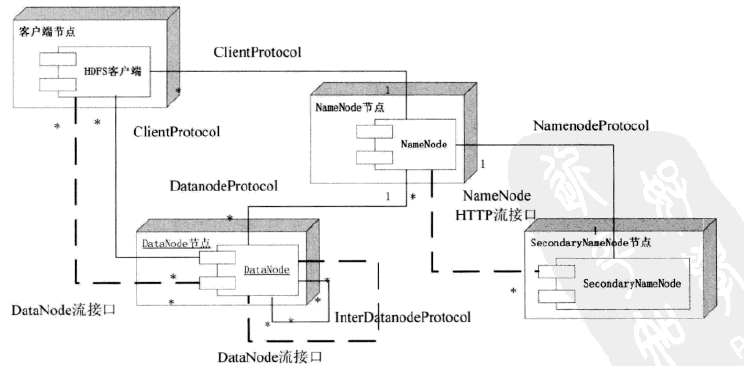
接下来setupResponse方法，将返回结果序列化Call的成员变量response中，完成发送应答前的处理。在RPC.Server.call中，传入的param会被强制转换成RPC.Invocation对象，通过该对象获取Method对象。

Handler通过Responser的doRespond方法，将处理完的结果交给Responder，把远程调用对象放入IPC连接的应答队列中，调用Hanler.processResponse()方法，向客户端发送结果，发送的过程由doAsyncWrite()中的channelWrite()发送至通道中。

## 5.3 HDFS RPC

### 5.3.1 IPC接口

HDFS各个实体间存在着多种信息交互的过程，很多使用Hadoop远程过程调用实现，下图是客户端、服务器各个实体间的接口：



接口分为两类：Hadoop远程过程调用接口和基于TCP或HTTP流式接口。这里仅介绍与IPC相关的接口，接口分为三类：

#### 5.3.1.1 客户端相关接口

ClientProtocol:客户端与Namenode间的接口，是HDFS访问文件系统的入口，客户端通过这个接口访问名字节点，操作文件或目录的元数据信息，读写文件也必须先访问Namenode,接下来再和数据节点进行交互，操作文件数据。另外从Namenode节点获取分布式温恩杰系统的一些整体运行状态信息，也是通过这个接口运行的。

ClientDatanodeProtocol:客户端与数据节点间的接口，用于客户端和数据节点进行交互，客户端和数据节点间的主要交互是通过流接口进行读/写文件数据操作。

#### 5.3.1.2 服务器间接口

包括namenode,secondaryNamenode、Datanode之间存在的IPC调用关系，具体接口包括：

DatanodeProtocol:数据节点与名字节点的接口，在HDFS主从体系结构中，数据节点通过这个接口向Namenode报告一些信息，同步信息到Namenode。同时，该接口的一些方法返回值会带回Namenode的指令，根据这些指令进行数据节点及数据块等操作。

InterDatanodeProtocol:数据节点与数据节点间的接口，数据节点通过这个接口和其他数据节点进行通信，恢复数据块，保证数据的 一致性

NamenodeProtocol:secondaryNamenode，HDFS balancer与Namenode之间的接口。SecondearyNamenode不断获取Namennode某时间点的namespace镜像和变化日志，然后合并为一个新镜像，并将结果返回给Namenode。该接口也为HDFS balancer的正常工作提供一些信息。

#### 5.3.1.3 和安全相关接口

包括RefreshAuthorizationPolicyProtocol和RefreshUserMappingProtocol，Namenode实现了这两个接口。

### 5.3.2 RPC调用流程

Hadoop远程方法调用使得HDFS两个进程实体间能够使用传统Java过程调用中的控制流模型来进行交互，这个交互可以使用Java基本类型数据或使用对象作为调用参数或返回值。

#### 5.3.2.1 Client与Namenode

通过ClientProtocol进行操作，下图是ClientProtocol的方法简介：



分为文件/目录相关事务的方法，写数据，读数据及工具dfsadmin依赖的方法。客户端调用HDFS的FileSystem实例，例如DistributedFileSystem的mkdir方法时， DistributedFileSystem对象通过IPC调用Namenode上的远程方法mkdir，让namenode执行具体的创建子目录操作；在目录树数据结构上的对象位置创建新的目录节点，同时记录这个操作并持久化到日志中，方法执行成功后，mkdir返回true，结束创建过程。步骤如下：

1) Namenode通过RPC.getServer(namenode,addr,port,handlercount,...,getConf)

在Server内注册该Namenode对象

2)DFSClient 通过RPC.getProxy(ClientProtocol.class,version,addr,ugi,conf..)获取namenode实例

3)Client 通过namenode(代理)，调用Namenode.mkdir

4)执行完成后，返回结果给Client

#### 5.3.2.2 Client与Datanode

Datanode实现ClientDatanodeProtocol接口，ClientDatanodeProtocol接口只有recoverBlock、getBlockInfo和getBlockLocalPathInfo三个方法。在一般情况下，客户端和数据节点通过基于流的接口进行交互，较少使用ClientDatanodeProtocol中提供的方法。

客户端在检测到Data异常，主动发起recoverBlock，客户端会通过ClientDatanode

Protocol接口采用RPC调用方式和DataNode通信，流程如下：

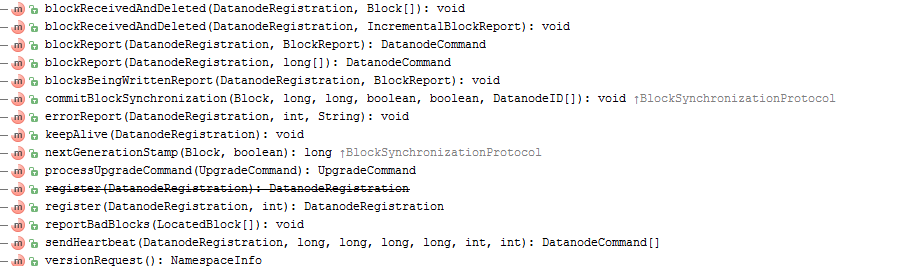
1) DFSclient.create()中，对outputStream进行初始化，调用了DFSoutputStream的构造函数

2)构造函数中，调用processDatanodeError()

3)在方法中，使用BlockRecoveryCoordinator.recoverBlock()，其中变量datanode的获取通过RPC.getProxy(InterDatanodeProtocol.class,versionID,addr,...)等获取

#### 5.3.2.3 Datanode与Namenode

DatanodeProtocol用于Datanode与Namenode之间的通信，数据节点是客户端，Namenode是服务器，方法如下：



Namenode维护整个文件系统的文件目录树及文件对应的数据块列表，这些信息存储在本地文件系统中，Namenode在启动时这些信息都要重建。Datanode在初始化时，会将当期存储的数据块告知Namenode，并且不断更新Namenode中本地数据块变化信息，特别是客户端写数据时引起的数据块变化，并接受来自Namenode的指令，上面的方法就是为了满足这些需求而设计的。

数据节点启动时，会和Namenode节点先进行握手，使用的是DatanodeProtocol接口的无参数方法versionRequest，返回值为NamespaceInfo对象。

注册是数据节点投入正常工作前需要进行的必要步骤，数据节点通过远程方法DatanodeProtocol.register()向Namenode节点注册，输入参数是DatanodeRegistration。

注册成功后，数据节点通过DatanodeProtocol中的blockReport成员方法，上报管理的全部数据块信息，帮助namenode创建数据块和数据节点的映射关系。blockReport返回一个DatanodeCommand对象，使用DatanodeCommand对象通知数据节点执行一些操作，如重新注册、发送心跳或删除本地磁盘上的数据块。

sendHeartbeat除了携带标识身份DatanodeRegistration对象外，还包括数据节点当前运行情况的信息，返回值是一个DatanodeCommand数组。下面先介绍相关的几个类：

1) DatanodeRegistration类和NamespaceInfo类

DatanodeRegistration用于数据节点的注册，继承自DatanodeID。DatanodeRegi

stration新增成员变量storageInfo，保存数据节点的存储系统信息。当Datanode进行注册时，Namenode和datanode需要对StorageInfo这些信息进行检查，保证当前注册的节点是HDFS的一个合法数据节点。

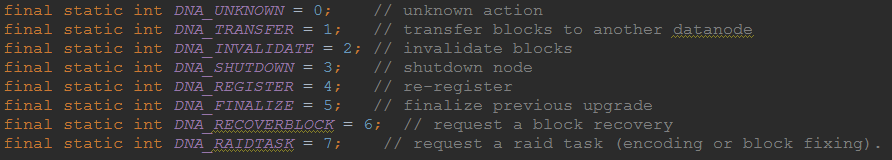
NamespaceInfo继承自storageInfo，包含的信息是整个HDFS集群的信息。

2) DataCommand

DatanodeCommand和其子类如下图所示：



blockReport和sendHeartbeat返回值都是DatanodeCommand，一共8种命令，其中包括DNA\_RAIDTASK命令，如下所示：



BlockComamnd有两个成员变量，分别是blocks[]和targets[][]，targets是datanodeInfo的二维数组，用于数据块复制或数据块恢复中，提供复制或恢复涉及的数据节点信息。

3) register

NameNode在startDNServer中使用dnprotocolServer = RPC.Server()，初始化server。

Datanode在createDataNode()中，runDatanodeDataNode()->NamespaceManager.

startAll()->NamesapceService.start()->调用NsOffferService.run方法。

Datanode内部类NSOfferService， 在方法setupNS()中通过RPC.waitForProxy(...)获取Namenode代理nsNamenode，在run中调用register()完成注册，远程调用nsNamenode.register。

4) Heartbeat

DataNode创建后，在NSofferService.run()中调用offerServices,Namenode的获取过程在上面已经介绍：

offerService中远程调用nsNamenode.sendHeartbeat(nsRegistration,...)，向namenode发送心跳报告，内容是dfs容量、剩余空间和DataXceiverServer数量等，返回信息为DatanodeCommand.

调用processCommand方法处理datanodeCommand，根据command类型，选择要进行的操作。下面以DNA\_RAIDTASK为例，介绍其处理过程：

datanode远程调用Namenode.sendHeartbeat后，调用FSnameSystem.handleHeartb

eat(nodeReg,cap,defUsed,...)

handleHeartbeat()，根据nodeReg信息查看其是否注册，如果没有注册返回DatanodeCommand.REGISTER

注册后，根据nodeReg获取DatanodeDescritpr nodeinfo，更新其信息（datanode信息），getLeaseRecoverCommand,getReplicationCommand,getInvalidateBlocks,getRaidCo

mmand,getDistributedUpgradeCommand以此根据注册信息生成指令，然后返回给Datanode。

datanode根据command类型，进行操作，其中RAID\_COMMAND，调用processRaidTaskCommand(RaidTaskCommand)，(RaidTaskCommand类中有成员变量RaidTask[])。

processRaidTaskCommand():进行encoding或者blockFixing任务，InjectionHandler.

processEventIO。

#### 5.3.2.4 Datanode与Datanode

InterDatanodeProtocol是Datanode与Datanode之间的接口，有以下三个方法：

getBlockMetaDataInfo()，startBlockRecovery()，updateBlock()

startBlockRecovery和updateBlock用于数据恢复，前面介绍ClientDatanodeProtocol接口的时候，数据节点向客户端提供了错误恢复的远程方法，ClientDatanodeProtocol.re

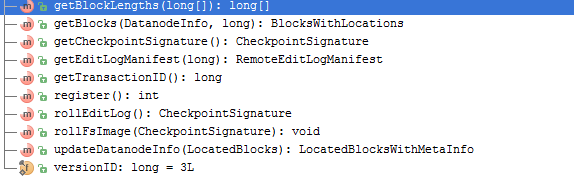
coverBlock()，同样Namenode到datanode的指令DNA\_RECOVERBLOCK也可以发起一个数据块恢复。

数据块的恢复，需要选择一个数据节点，协调其他数据节点进行恢复，该过程使用InterDatanodeProtocol的这两个远程方法，startBlockRecovery获得参与数据块恢复过程中各个数据节点上的数据块信息，返回BlockRecoveryInfo对象，包含数据块Block和标志位wasRecoverdOnStartup。根据返回信息，通过updateBlock方法通知其他各数据节点。

#### 5.3.2.5 Namenode与SecondaryNamenode

Namenode实现接口NamenodeProtocol，调用者有SecondaryNamenode和balancer,

有以下的方法：



balancer使用NamenodeProtocol的getBlocks和getBlockKeys方法，getBlocks获得某一个数据节点上一系列数据块及位置，根据返回值，把数据块从该数据节点移到其他数据节点，达到平衡各个数据节点数据块数量的目的。getBlockKeys方法用于支持这个过程中需要的安全特性。

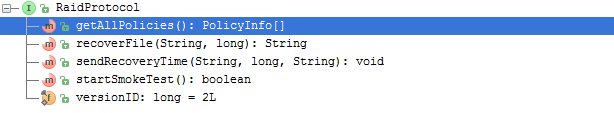
SecondaryNamenode：另外几个方法完成获取HDFS的命名空间镜像和镜像编辑日志，合并得到一个新的镜像，上次新的命名空间镜像到namenode，替换原镜像并清除日志。getE

ditLogSize()获得Namenode编辑日志大小，rollEditLog通知Namenode开始合并过程。合并完成后，SecondaryNamenode通过HTTP接口上次新的元数据镜像，操作完成后，无参的远程方法rollFSImage被调用。

### 5.3.3 Raid相关的RPC调用

#### 5.3.3.1 RaidProtocol

该接口用于User code和RaidNode的通信，其继承了VersionedProtocol。为了扩展Raid功能增加的接口，下图是RaidProtocol的几个方法：



getAllPolicies():获取配置参数

recoverFile(inputPath,corruptOffset):当文件损坏时，包括校验文件，使用该方法进行文件的恢复，移动文件到file.old，并使用RAID subSystem进行恢复（这个方法没有被使用）

sendRecoveryTime:文件不在变动时，将recoveryTime发送到RaidNode，用于数据的收集。

startSmokeTest()：功能测试

#### 5.3.3.2 RaidNode

RaidNode继承RaidProtocol，其成员变量Server server 用于远程RPC调用。通过RPC.getServer(this,addre,port,handlerCount,false,conf) 放入Server中已供调用。

RaidNode的主要功能是接收Client端的RPC请求和调度各守护线程完成数据的raid化和数据修复，parity文件删除等操作。

#### 5.3.3.3 RaidShell与RaidNode

在Raid中只有Raidshell与RaidNode通过RPC调用，DRFS未使用RPC，因此下面仅介绍RaidShell与RaidNode的通信过程，在Raid的几个线程的RPC调用都是通过RaidShell来实现的，这点使与FsShell是不同的。

RaidShell.java的成员变量：RaidProtocol raidNode 及RaidProtocol rpcRaidNode;

用于RPC与RaidNode的通信

rpcRaidNode = createRPCRaidNode()，调RPC.getproxy(RaidProtocol.class,address

,port,...)

raidNode = createRaidNode(rpcRaidNode)

通过RPC调用远程的方法，如下所示：

1、private int showConfig(string cmd,string argv[],startIndex)

PolicyInfo[] all = raidnode.getAllPolicies()

2、private boolean startSmokeTest() throws Exception{

return raidnode.startSmokeTest()

}

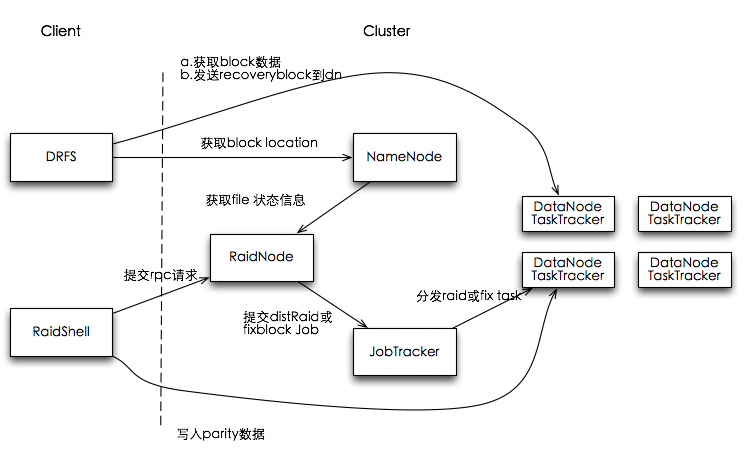
#### 5.3.3.4 线程中的RPC调用

有两个线程涉及到了RaidNode RPC调用，分别是DistBlockIntegrityMonitor和LocalBlockIntegrityMonitor。两个线程中的成员变量：raidnode和rpcraidnode，都是通过RPC.createRaidNode()和RPC.createRPCRaidNode()来构建。

这两个线程是对已经raid化的数据进行检查，检查的内部包括文件的corrupt和decommossion。

# 六、Raid类分析

前面几章分析了HDFS的文件系统、运行机制及为了实现Raid而做的改变，下面分析Raid的各个类，下图是Raid的结构：



Raid文件分为三类，源文件、parity文件和Har文件(归档)。

## 6.1 Raid简介

6.1.1 Client端HDFS RAID

使用场景主要有两个：raid数据管理和raid数据读取

### 6.1.1.1 数据管理

DRFS的管理包括DFS中文件的raid化，查询raid文件的状态，主要是通过HDFS RAID提供的raidshell 工具来完成，raidshell作为Client工具，通过RPC与集群中的RaidNode通信，完成各种管理操作。

RaidNode启动一个RPC Server，接受RPC请求执行相应的命令，对应的协议是RPCp

rotocol。

#### 6.1.1.2 Raid数据读写

使用HDFS RAID client端需要配置fs.hdfs.impl为DistributedRaidFileSystem,DRFS包装了DFS的读请求，调用HDFS Client去Namenode/datanode获取数据，如果返回数据正常则OK，如果读取过程中遇到corrupted data，那么DRFS client截获HDFS Client返回的BlockMissingException，然后接管这个文件的读取流，通过parity file恢复丢失的数据块，然后返回给应用程序。

而且DRFS client在读取到corrupted data block，然后恢复之后只是传给了应用程序，并不会把数据恢复到HDFS中，而BlockFixer和通过RaidShell触发的recoverFile操作会把修复好的块复原到HDFS中。这里涉及的类为DistributedRaidFileSystem,Dsitrited

FileSystem,BlockFixer,RaidShell等，流程如下图所示：

当block读取时发生block丢失或损坏，DRFS会向RaidNode发送RPC对失效的数据进行恢复。

### 6.1.2 RaidNode结构（Server端）

RaidNode是HDFS-RAID中除了NameNode和JobTracker之外的第三个master node，主要接受client端的RPC请求和调度各守护线程完成数据的raid化和数据修复，parity文件修复。有两种实现方式：

1) LocalRaidNode:在RaidNode本地进行parity计算，parity文件的生成是一个计算密集型任务，本地计算有限，该方式的扩展型有限

2) DistributedRaidNode:通过提交mapreduce job来进行parity计算

## 6.2 Raid元数据

### 6.2.1 PolicyInfo

PolicyInfo是Raid 机制的基础

RaidNode在启动后，会生成一个ConfigManager，通过读取raid.xml配置文件，获取用户指定的policy，一般一个policy对应一个文件或者目录，包括这个目录下什么样的文件会触发RAID操作（例如srcReplication是3的文件，modTimeperiod为360000表示最后一次修改时间在1个小时后的文件才被作为Raid的备选）；这些文件采用何种编码方式（xor或者RS）;编码后parity file和meta file要存放几个备份。

#### 6.2.1.1 成员变量

1)Path srcPath ;指定的源文件路径

2)Path fileListPath;包含一系列文件的路径

3)String policyNmae; policy名称

4) String codecId; 使用的codec

5) boolean shouldRaid;

6) Configuration conf;hadoop configuration

7) Properties properties; policy-dependent properties

#### 6.2.1.2 方法

方法中用于上面的成员变量的初始化

### 6.2.2 Codec

这个类用于初始化使用的ErasueCode，Stripe和partity长度，校验文件目录和Codec优先权。

#### 6.2.2.1 成员变量

String jsonStr；用于ErasureCode.init()

String id;code id

int stripeLength; 在一个stripe中有多少blocks

int parityLength; 每个stripe创建多少个校验块

string erasureCodeClass;所使用的ErasureCode类名称

String parityDirectory；校验文件的目录

boolean isDirRaid；file-level raiding还是directory-level raiding

List<Codec> codecs;

Map<String,Codec> idToCodec;

#### 6.2.2.2 方法

void initializeCodec(conf):根据hadoop conf进行初始化，在raid-default.xml中定义了性质raid.codecs.json，使用json格式描述的值。

Codec(JSONobject json):构造函数，根据json中的值进行设置

其他的方法不再介绍，但是在一个定义文件中会定义多个Codecs;

### 6.2.3 StripeStore/StripeInfo

StripeStore用于directory-raid中来存储stripes的block信息。在raid过程中可以将不同的文件放在同一stripes中。StripeStore应该支持以下功能：

给定一个Block,可以查询同一stripes中的其他Block

将包含blocks的stripe存入stripe store中

StripeStore是一个抽象类

#### 6.2.3.1 StripeInfo(StripeStore的内部类)

1）成员变量

Codec codec;

Block block;

List<Block> parityBlocks;

List<Block> srcBlocks;

2) 方法

int getBlockIdxInstripe(Block block):返回block在stripe中的位置，用Idx表示

#### 6.2.3.2 StripeStore

initialize(): stripeStore的初始化

putStripe(codec,parityBlks,srcBlks)：

StripeInfo getStripe(Codec codec,Block block):根据block找到其所属的stripeInfo

### 6.2.4 LocalStripeStore和DBStripeStore

StripeStore是一个抽象类，必须对其进行实现。在FB中有两个实现LocalStripeStore和DBStripeStore。

#### 6.2.4.1 LocalStripeStore

1)内部类LocalStripeInfo

继承了StripeInfo的成员变量，增加了几个方法

getKey(Codec codec,Block blk):返回codec.id+DELIMITER+blk.toString()

getBlock(String str):获取目录下的Blocks

LocalStripeInfo getStripeInfo(File storeDir,String fileName):根据文件StoreDir和fileName获取其StripeInfo：

如果获取的文件太小(文件为Blocks)，parts.length<codec.parityLength，则没有返回值，没有足够的元素用于codec

否则，将block添加进StripeInfo中，将前几个Block，作为parityBlocks,后面的Block作为srcBlocks

2) StripeStore的方法

initialize(conf,createStore,fs):首先获取storeDirHome，在hdfs.raid.local.stripe.dir中定义，如果没有Store则创建该目录

获取storeDirHome下的所有文件的LocalStripeInfo，将StripeInfo添加到blockToStripeStore中。

persistent(LocalStripeInfo,tmpFile): 文件的持久化，createHardLink(tmpFile,f)

createStripeFile(parityBlks,srcBlks)：创建stripeFile

putStripe(codec,parityBlks,srcBlks)：持久化LocalStripeStore

getStripe(codec,blk):根据block信息获取其所属的StripeStore

### 6.2.5 RaidInfo

## 6.3 Raid工具类

### 6.3.1 RaidCodec

该类与Codec有重复的地方

1)成员变量

String id; codec的id号

int numDataBlocks;在一个stripe中data blocks的数目

int numParityBlocks;在一个stripe中parity blocks的数目

int numStripeBlocks;stripe中blocks的总数目

long stripeChunkSize;

short parityReplication;校验块的副本数目

short minSourceReplication;最少的source blocks数目

String erasureCodeClass; erasureCode类的全称

List<RaidCodec> codecs;

List<String,RaidCodec> idToCodec;

2) 方法

initializeCodecs(conf):初始化codec，在Hadoop conf中定义，hdfs.raid.codec.json.

ErasureCode createErasureCode(conf):根据配置文件，生成ErasureCode，根据变量中的erasureCodeclass来生成，使用反射机制。

BlockInfo[] getSourceBlocks(BlockInfo[] blocks):返回raidFile的sourceBlocks，根据raidFile的blocks放置，前numParityBlock block为parityBlock，其余的为sourceBlock。

BlockInfo[] getParityBlocks(BlockInfo[] blocks)

BlockInfo[] convertToRaidStorage(BlockInfo[] partitBlocks,BlockInfo[] blocks,int checksum,BlockMap blocksMpa,replication,InodeFile inode):

根据blocks的个数生成stripe的个数

将parityBlocks的数据复制到newList中

执行blocksMap.unpdateINode(newList[j],new RaidBlckInfo(newList[j],parityReli

cation,j),inode,parityReplication,true)，更新blocksMap中inode的信息，将parityBlocks加入到inode file中。

boolean checkRaidProgress(INodeFile sourceINode,raidEncodingTasks,fs,forceAdd),

count raided file中的可用的parity block的live replicas。如果一个stripe没有足够的parity Block replicas，则执行encoding过程增加parity block

### 6.3.2 ConfigManager

这个类用于将Raid.xml配置文件读入内存，并进行管理。

#### 6.3.2.1 成员变量

1) final long RELOAD\_INTERVAL; config file重新检测间隔

2) long RESCAN\_INTERVAL; 运行配置文件中的policies的时间间隔

3) Configuration conf;hadoop相关配置

4) Thread reloadThread;

5) Collection<policyInfo> allPolicies = new ArrayList<~>()

6) boolean deReload；是否需要重载，当policies变化时重载

#### 6.3.2.2 方法

1) 初始化 ConfigManager()

从hadoop.conf中的raid.config.file获取raid.xml的位置，并根据conf进行初始参数设置。

然后调用reloadConfigs()，文件可以是xml或者json文件，

2) reloadConfigs()

以xml初始化为例，先将xml文件进行解析获取Nodelist，从解析的元素中初始化PolicyInfo。

3) UpdateThread.run

根据设置的时间间隔，进行配置的reload

### 6.3.3 EnCoder

根据source file产生parity file

1) 核心成员变量：

Configuration conf

Codec codec;

ErasureCod code;

StripeStore stripStore;

2) 方法

encodeFile(parityFS,parityFile,numStipes,StripeReader sReader,EncodingCandida

te ec)

encodeFileToStream(StripeReader sReader,....)

### 6.3.4 Decoder

这个类用于从读取包含损坏Block的文件，其中损坏block的恢复通过parity file来实现。

1) 核心成员变量

Codec codec;

ErasureCode code;

StripeStore stripeStore;

ErasureCode[] paralleCode;

2)方法

connectToStore(srcPath):

this.stripeStore=RaidNode.createStripStore(conf,false,fs)

StripeInfo retrieveStripe(Block lostBlock,Path p,....)

从stripeStore中获取lostBlock所在的stripeInfo

recoverBlockToFile():

recover 一个corrupt Block到本地文件

recoverBlockToFileFromStripeInfo()

fixErasedBlock()

### 6.3.5 StripeReader

1)成员变量

Codec codes; 校验编码方式

Configuration conf;

long stripeStartIdx;

long stripeEndIdx;

int bufferSize;

long currentStripeIdx;

2)内部类

StripeInputInfo:

成员变量：InputStream[] inputs;

Path[] srcPaths;

long[] offSets;

LocationPair:

成员变量：int stripeIdx;

int blockIdxInStripe;

List<FileStauts> lfs;

3)子类FileStripeReader和DirectoryStripeReader

FileStripeReader:

成员变量： FileSystem fs;Path srcFile;long blockSize;

方法：

StripeInputInfo getStripeInputs(stripeIdx)

为每个block创建一个InputStream，并根据stripeLength创建一个StripeInputInfo

DirectoryStripeReader:

成员变量：Path srcDir;源目录

List<FileStatus> lfs;

long parityBlockSize;

List<BlockInfo> stripeBlocs; BlockInfo是个内部类，成员变量为 fileIdx,blockID

方法：

getDirLogicalSize(lfs):目录的逻辑大小

getDirPhysicalSize(lfs):目录的物理大小

StripeInputInfo getStripeInputs(stripeIdx):

BlockLocation[] getNextStripeBlockLocations():获取curFileIdx的block信息

### 6.3.6 DistRaid

用于Raid的MapReduce任务

内部类EncodingCandidate:

成员变量： FileStatus srcStat;

long startStripe;

long encodingUnit;

String encodingID;

List<List<Block>> srcStripes；

DistRaid类对strcStripes进行doRaid操作。

## 6.4 RaidNode

### 6.4.1 成员变量

Server server;IPC server

InetSocketAddress serverAddress; rpc server地址

ConfigManager configMr;

Configuration conf;

### 6.4.2 线程

RaidNode会启动多个线程:

BlockFixer线程用于定期扫描看是否有数据数据corrupt然后执行恢复；

TriggerMonitor安装raid.xml配置的各种策略定期检查相应的文件，然后执行encoding操作生成parity file；

PurgeMonitor线程用于删除已经废弃的parity file；

HarMonitor定期把parity file小文件安装Har的形式合并，减少Namenode元数据存储的压力。

BlockIntegrityMonitor:负责通过DFS的fsck来对DRFS中已经raid化的数据进行检查，检查内容包括corrupt和decommosion。一旦检测到这类文件，BlockIntegrityMonitor会通过其维护的CorruptMonitor和DecommossionMonitor的两个线程进行数据的恢复。

有两类Monitor: LocalBlockIntegrityMonitor和DistBlockIntegrityMonitor。

BlockCopier:用于构建decommission文件的workker线程

PlacementMonitor:通过blockMove完成对DRFS中的根据placement策略提供在Datanode之间move block的工具线程。BlockMove通过ClusterInfo线程周期性的获取Live节点的最新topo结构。对于parity Block过于集中的节点，将其分开。

BlockMoveAction：分散的过程主要是为每个的block构建一个BlockMoveAction线程，该线程在所有datanode中除当前block所在的节点外随机选取一个datanode，并选取一个proxysource datanode，proxysource datanode是用于将block复制到datanode的源节点，选取规则是优先选取当前block副本所在dn中与目标datanode所属同一rack的节点，如果没有，则从副本列表中随机选取一个作为源节点。

## 6.5 RaidShell

### 6.5.1 成员变量

RaidProtocol raidnode;raidnode代理类

RaidProtocol rpcRaidnode;

Configuration conf;

其他变量不再介绍

### 6.5.2 方法

createRPCRaidNode(InetSocketAddress,conf,ugi)

createRaidNode() //创建RaidNode代理

根据RaidShell指令的执行，共有执行几种方法：

-showConfig: showConfig()  
 调用raidnode.getAllPolicies()方法，并打印所有PolicyInfo信息。RaidNode实现了RaidProtocal接口，因此会调用RaidNode的getAllPolicies()方法，最终调用configMgr.getAllPolicies()。

-recover: recoverAndPrint()

调用recover(cmd,argv,startindex)

-recoverBlocks:recoverBlocks()

使用fixer = new BlockReconstructor.corruptBlockReconstructor(conf)

fixer.reconstructFile(new Path(path),null)

如果RaidNode.isParityHarPartFile(srcPath)，那么调用processParityHarPartFile()

使用lostBlocksInfile(dfs,uriPath,partFileStat)，获取lostBlocks。

lostBlock = lb.getBlock()

lostBlockOffset = lb.getStartOffset()

localBlockFile = File.createTempFile(lostBlock.getBlockName(),".tmp")

1) RaidNode.isParityHarPartFile()

processParityHarPartBlock():创建encoder,encoder.recoverParityBlockToStream()

2) 需要创建parityFile

processParityFile(srcPath,decoder,context)

decoder.recoverParityBlockToFile(srcFs....)

computeMetadataAndSendReconstructedBlock()

3) 不用connect to stripe store，重建source file

StripeInfo si = decoder.retrieveStripe(lostBlock,.....)

decoder.recover.BlockToFileFromStripeInfo

decoder.recoverBlockTofile()

4) throuth stripe store，重建source file

-raidFile: raidFile()

RaidNode.doRaid()

RaidNode.generateParityFile()

构建stripeReader,encoder.encodeFile(....,stripeReader)

-DistRaidCommand(): distRaid()

DistRaid.startDistRaid()

-FILE\_CHECK\_CMD:fileCheck

FastFileCheck.startFileCheck()

启动mapreduce任务

-fsck:fsck()

-usefullHar: usefulHar()

PurgeMonitor.usefulHar()

-checkFile(): checkFile()

ifFileCorrupt()

-purgeParity: purgeParity()

-checkParity: checkParity()

-findMissingParityFile：findMissingPartityFile()

-verifyParity:verifyParity()

-verfifyFile: verifyFile(argv,i)

-rs\_benchmark: reBeanchmark()

-estimateSaving: estimateSaving()

-smokeTest: startSmokeTest()

启动smokeTest，raidnode.startSmokeTest()

## 6.6 DistributedRaidFileSystem

# 七、其他

Mapreduce中与raid有关的变动（肯定有）