HDFS 源码解析

HDFS是Hadoop的分布式文件系统，作为一个高容错的特点，支持超大文件、检测和快速应对硬件故障、低延迟数据访问等特性。文件系统采用了主从体系结构，主要包括NameNode、DataNode和客户端三个角色。角色之间的访问关系，如下图所示：

下图描述了数据读取过程中客户端与之交互的HDFS、namenode和datanode之间的数据流：



图1 客户端读取HDFS中的数据

如上图所示，文件系统客户端在步骤1中用FileSystem对象的open函数打开文件，文件系统使用RPC通信机制与管理节点通信获取该文件的数据块信息(步骤2)。文件的读取通过FSDataInputstream输入流来完成，在步骤1，2后文件系统会返回一个输入流的对象。在步骤3中，客户端通过输入流对象调用read()方法读取数据，在文件系统中每个文件有多个副本，输入流对象会根据系统信息选择连接距离最近的datanode读取文件。

在Hadoop中，客户端通过文件类的API来读取HDFS中的文件内容，下图是从HDFS文件系统中读取数据的流程图：



图2 客户端将数据写入HDFS

文件系统的写入过程相对复杂，而且时间耗费大。在步骤1，2中，Client通过文件系统提供的API，调用分布式文件系统，通过文件系统对象的create方法启动创建文件过程，要完成的工作包括在命名空间创建文件元数据，并返回一个FSDataOutputStream输出流对象。为了保证文件的可靠性，写入多个副本，在输出流中根据副本数选择多个数据节点构成管线，步骤3通过输出流对象将数据其中的一个数据节点，然后数据节点之间进行数据发送（步骤4）。各数据节点在数据写入完成时向管道返回确认信息，在步骤5中管道确认数据写入完毕后，从队列中删除数据包。客户端完成数据的写入后，通过步骤6，7结束数据写入流程。

一、NameNode主要数据结构及执行流程

Namenode是HDFS的“书记员”，维护着整个文件系统的文件目录树，文件/目录的元信息和文件的数据块索引，即每个文件对应的数据块列表。这些信息以两种形式存储于本地文件系统中，一种是命名空间镜像，另一种是命名空间镜像的编辑日志。Namenode有两个功能，分布式维护hdfs的元数据及数据块和数据节点之间的对应关系。

1）HDFS元数据

元数据包括两个部分：HDFS当中的目录和文件在内存当中是以一棵树的形式存储的，而这个数据结构是由namenode维护的，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。HDFS当中的文件同数据块的对应关系，每个文件被拆分为若干数据块冗余存放的，这个文件同数据块的对应关系是由Namenode来维护的。

2)数据块和数据节点

对应包括两个方向，数据块->数据节点，数据块存放在哪个节点上面。还有另外一个方向是数据节点->数据块，数据节点当中有哪些数据块。

NameNode的设计和实现过程中设计一些处理逻辑，如读操作、写操作、append等操作。通过Namenode，客户端可以获取数据块所在的数据节点信息。NameNode中与数据节点相关的信息（命名空间镜像和编辑日志）不保留在名字节点的本地文件系统中，NameNode每次启动，都会动态的重建这些信息。客户端通过Namenode获取上述信息，然后和Datanode进行交换，读取文件数据。HDFS是为大文件而设计，但存放在HDFS上的文件和传统文件系统类似，也就是将文件分块，然后进行存储。但和传统文件系统不同，在数据节点上，HDFS文件块以Linux文件系统上的普通文件进行保存。 在NameNode中主要的数据结构有FSNameSystem,FSDirectory,BlocksMaph和其他结构，下面进行详细的介绍。

1.1 FSDirectory

在namenode中，对于hdfs整个文件系统的名字空间，也就是以“/”为根的整个目录树的管理，通过FSDirectory来管理。

HDFS整个文件系统的namespace在NameNode的内存中，以一棵树的结构来维护。在HDFS中，不管是目录还是文件，都被看作是INode。如果是目录，则实际的标识是INodeDirectory，如果是文件则为INodeFile，这两个都是INode的派生类。INodeDirectory包含一个成员数组变量List children，如果该目录下有子目录或者文件，该子目录或文件的引用会保存在children列表中。HDFS通过这种方式维持整个文件系统的目录结构。

HDFS会将namespace保存到namenode的本地文件系统的一个叫fsimage的文件中，从该文件中，Namenode每次重启都能将整个hdfs的namespace重构。同时也负责对fsimage相关操作，另外，对HDFS的各种操作，Namenode都会记录相应的操作日志，以便周期性的将该日志与fsimage进行合并，生成新的fsimage。该日志文件也是在Namenode本地文件系统中（editlog文件）。FSDirectory已负责对editlog的相关操作。

FSDirectory类图，下图是类中的成员变量：

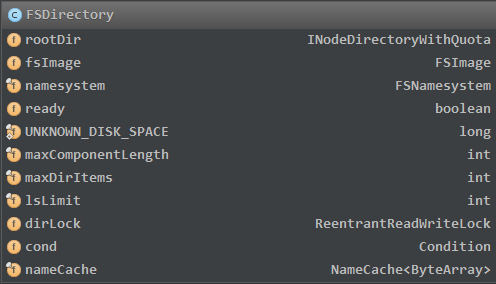


图3 FSDirectory 类图

从图中可以看出，FSDirectory维护的Namespace的根节点（这个根节点是整个文件系统的root,InodeDirectoryWithQuota类型），同时FSDirectory还持有一个FsImage的引用。

1.1.1 INode（抽象类）

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。

INode类如图4所示：

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，Inode抽象类只维护了一个字段，父类的inode引用。

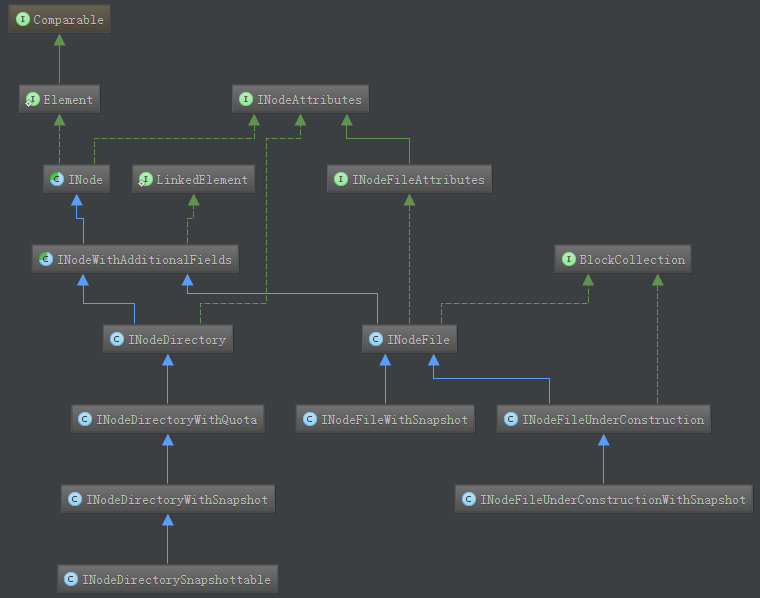


图4 INode类图

1) 类声明

Inode实现了INodeAttributed接口，这个接口包括以下五个字段的get方法：

userName : 文件\目录所属用户名

groupName : 文件\目录所属组名

modificationTime : 上次修改时间

accessTime : 上次访问时间

fsPermission : 访问权限

2) 方法

除了实现INodeAttributed接口当中的方法外，还有INode元信息的get与set接口，例如inode id,name-文件\目录名称，fullpathName-文件\目录的完整路径，parent-文件目录的父节点、nsQuota-命名空间配额、dsQuota-磁盘空间配额。以及判断方法：isFile-是否为文件、isDirectory-是否为目录、isSymlink-是否为符号链接。

这里需要注意的是，Inode类的设计采用了模板模式。方法的定义多为两个，一个作为final的接口，另一层则作为abstract的具体实现，留给子类具体实现。

/\*\* 获取 {@link PermissionStatus} \*/

abstract PermissionStatus getPermissionStatus(Snapshot snapshot);

/\*\* 等同于 getPermissionStatus(null). \*/

final PermissionStatus getPermissionStatus() {

return getPermissionStatus(null);

}

3) 字段

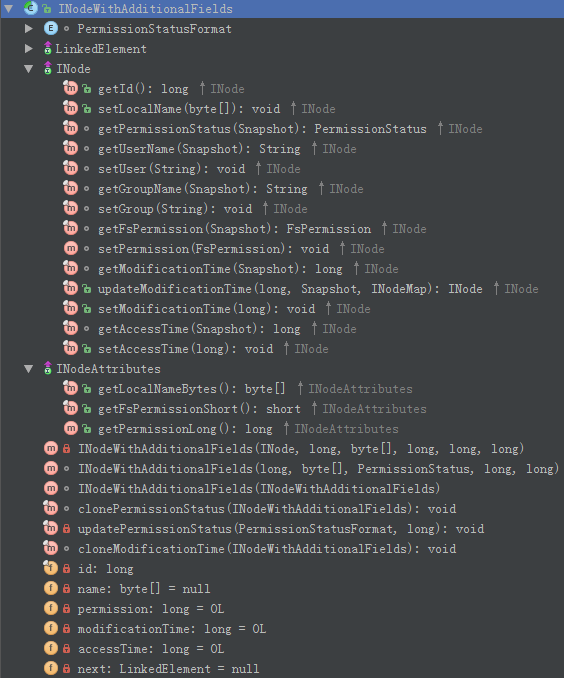
Inode当中只有一个字段，就是parent。表明当前Inode的父节点 – 父节点也是Inode类型。

/\*\* parent is either an {@link INodeDirectory} or an {@link INodeReference}.\*/

private INode parent = null;

1.1.1.1 InodeWithAdditionalFields

在Inode的实现中，只有一个字段parent，其余信息都通过get/set方法。本节讲解的InodeWithAdditionalFields就实现了这些字段- id、name、permission、modificationTime、accessTime，并且覆盖了Inode当中对应的方法。



这里比较有意思的是permission这个字段，配合内部类PermissionStatusFormat内部类，来标识权限。以下是permission字段的详解

对于permission字段我们可以通过linux的权限来理解，主要是包括三个部分：a. 用户 b.用户组 c.权限（0777这种）。

permission字段是一个long类型，其中前16个bit是用来存放mode（就是777那个），后25个bit用来存放group，再后来23个用来存放user。retrieve()以及combine()就是通过移位以及并操作获取和合并指定字段。

static enum PermissionStatusFormat {

MODE(0, 16),

GROUP(MODE.OFFSET + MODE.LENGTH, 25),

USER(GROUP.OFFSET + GROUP.LENGTH, 23);

final int OFFSET;

final int LENGTH; //bit length

final long MASK;

PermissionStatusFormat(int offset, int length) {

OFFSET = offset;

LENGTH = length;

MASK = ((-1L) >>> (64 - LENGTH)) << OFFSET;

}

long retrieve(long record) {

return (record & MASK) >>> OFFSET;

}

long combine(long bits, long record) {

return (record & ~MASK) | (bits << OFFSET);

}

//...

}

HDFS当中，用户名和用户标识的映射、用户组名和用户组名的映射保存在SerialNumberManager这个对象当中。这样不用在inode对象当中保存字符串形式的用户名和用户组名，节省了内存（引用2）。所以Inode当中的抽象方法getName()可以先通过PermissionStatusFormat获得用户的int类型的user id，然后通过SerialNumberManager获取字符串类型的用户名。

final String getUserName(Snapshot snapshot) {

if (snapshot != null) {

return getSnapshotINode(snapshot).getUserName();

}

int n = (int)PermissionStatusFormat.USER.retrieve(permission);

return SerialNumberManager.INSTANCE.getUser(n);

}

1.1.1.2 InodeDirectory

1) 作用

其实就是一个目录的inode，抽象了hdfs当中的目录

2)字段

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。

private List<INode> children = null;

3)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：

private void addChild(final INode node, final int insertionPoint) {

if (children == null) {

children = new ArrayList<INode>(DEFAULT\_FILES\_PER\_DIRECTORY);

}

node.setParent(this);

children.add(-insertionPoint - 1, node);

if (node.getGroupName() == null) {

node.setGroup(getGroupName());

}

}

1.1.1.3 INodeDirectoryWithQuota

1）作用

InodeDirectory的子类，Quota单词的意思就是配额，也就是具有配额限制的目录Inode的实现。

2)字段

从成员变量上面我们就可以看出来：

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

3)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

1.1.1.4 InodeFile

1)作用

扩展自InodeWithAdditionalFields类，表示一个关闭的文件inode，与inodeDirectory对应。这里要特别注意，InodeFile与InodeDirectory的不同在于，InodeFile当中存储了Block信息。

2) 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。

blocks:对应于文件对应的块列表(blockInfo，数据块的元数据)，重要的值

private long header = 0L;

private BlockInfo[] blocks;

3) HeaderFormat

HeaderFormat处理header字段当中的long类型数据，前16比特作为冗余备份数目，后48比特为期望的块大小

static class HeaderFormat {

/\*\* Number of bits for Block size \*/

static final int BLOCKBITS = 48;

/\*\* Header mask 64-bit representation \*/

static final long HEADERMASK = 0xffffL << BLOCKBITS;

static final long MAX\_BLOCK\_SIZE = ~HEADERMASK;

static short getReplication(long header) {

return (short) ((header & HEADERMASK) >> BLOCKBITS);

}

static long combineReplication(long header, short replication) {

if (replication <= 0) {

throw new IllegalArgumentException(

"Unexpected value for the replication: " + replication);

}

return ((long)replication << BLOCKBITS) | (header & MAX\_BLOCK\_SIZE);

}

// ...

}

4)方法

InodeFile 扩展了 INodeFileAttributes fileReplication（文件副本数）以及header（上一节提到的文件头）两个字段的get()方法。

addBlock()/setBlock() : 对于blocks这个数组进行设置

void addBlock(BlockInfo newblock) {

if (this.blocks == null) {

this.setBlocks(new BlockInfo[]{newblock});

} else {

int size = this.blocks.length;

BlockInfo[] newlist = new BlockInfo[size + 1];

System.arraycopy(this.blocks, 0, newlist, 0, size);

newlist[size] = newblock;

this.setBlocks(newlist);

}

}

1.1.1.5 InodeFileUnderConstruction

1) 作用

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）

2) 字段

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）

3) 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

1.1.2 FSImage

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，以防止节点掉电或者进程崩溃。但如果每次都将内存中的元数据保存到fsimage文件当中，将会非常的消耗资源。因此，Namenode将近期进行的操作保存到fseditlog当中，然后定期合并fsimage和fseditlog文件。

FSimage主要进行的操作：

1) 保存namespace，将namenode内存当中的namespace保存到fsimage文件中

2) 加载fsimage，将磁盘上fsimage文件保存的namespace加载到namenode内存当中

3)合并fseditlog以及fsimage

1.1.2.1 保存namespace

将namenode内存中的元数据保存到fsimage文件中，下面saveFSimage方法看起：

SaveFSImage()

void saveFSImage(SaveNamespaceContext context, StorageDirectory sd)

throws IOException {

long txid = context.getTxId(); //获取当前namespace最大txid

File newFile = NNStorage.getStorageFile(sd, NameNodeFile.IMAGE\_NEW, txid); //fsImage文件

File dstFile = NNStorage.getStorageFile(sd, NameNodeFile.IMAGE, txid); //fsImage文件2，这里为什么这么设计？

FSImageFormat.Saver saver = new FSImageFormat.Saver(context); // Saver类负责保存fsImage (收获)

FSImageCompression compression = FSImageCompression.createCompression(conf); // 压缩类

saver.save(newFile, compression); //调用saver保存fsImage

MD5FileUtils.saveMD5File(dstFile, saver.getSavedDigest()); //保存MD5校验值

storage.setMostRecentCheckpointInfo(txid, Time.now()); //保存当前

}

可以看到fsImage保存使用了一个类FSimageFormat，这个类读取和保存fsiamge的工具类，维护了fsimage文件的格式。提供了两个内部类saver保存fsimage文件。

1.1.2.1.1 FSImageFormat.saver

Saver类有三个比较重要的字段，如下代码所示：

//上下文

private final SaveNamespaceContext context;

//保存是否完成

private boolean saved = false;

//文件已写部分的MD5校验值（收获）

private MD5Hash savedDigest;

Saver类最重要的方法，save() 保存fsimage文件

void save(File newFile, FSImageCompression compression) throws IOException {

// ... 准备工作

// 1. 获取写的IO流 - out

MessageDigest digester = MD5Hash.getDigester();

FileOutputStream fout = new FileOutputStream(newFile);

DigestOutputStream fos = new DigestOutputStream(fout, digester);

DataOutputStream out = new DataOutputStream(fos);

try {

// 2. 写fsimage文件的文件

out.writeInt(HdfsConstants.LAYOUT\_VERSION);

out.writeInt(sourceNamesystem.unprotectedGetNamespaceInfo()

.getNamespaceID());

out.writeLong(fsDir.rootDir.numItemsInTree());

out.writeLong(sourceNamesystem.getGenerationStampV1());

out.writeLong(sourceNamesystem.getGenerationStampV2());

out.writeLong(sourceNamesystem.getGenerationStampAtblockIdSwitch());

out.writeLong(sourceNamesystem.getLastAllocatedBlockId());

out.writeLong(context.getTxId());

out.writeLong(sourceNamesystem.getLastInodeId());

sourceNamesystem.getSnapshotManager().write(out);

// 3. 写压缩信息

out = compression.writeHeaderAndWrapStream(fos);

LOG.info("Saving image file " + newFile +

" using " + compression);

// 4. 保存目录树跟节点

saveINode2Image(fsDir.rootDir, out, false, referenceMap, counter);

// 5. 保存其他节点

saveImage(fsDir.rootDir, out, true, counter);

// ... 其他操作

// 6. 保存构建中的节点

sourceNamesystem.saveFilesUnderConstruction(out);

// ... 其他操作

out.flush();

context.checkCancelled();

fout.getChannel().force(true);

} finally {

out.close();

}

// 保存完成

saved = true;

// 设置当前文件的MD5校验

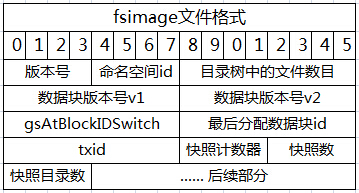
savedDigest = new MD5Hash(digester.digest());

}

我们将这个方法的实现分为6个步骤：

1) 获取IO流 ： 这里注意，我们包装了一个DigestOutputStream，这样就可以一边写入文件，一边更新校验值，很方便。

2) 写文件头 ： fsimage文件的文件头如下图所示，这部分代码写入文件头对应的值。



3) 写压缩信息 ： 将压缩信息写入文件当中。这里要注意的是，如果有压缩，则替换文件的输出流。

4) 保存目录树根节点 ： 将目录树的跟节点写入文件，这里直接调用了FSImageSerialization.saveInode2Image() 方法

5) 保存其他节点 ： 调用saveChildren()将目录树当中所有节点写入文件（最终调用FSImageSerialization.saveINode2Image方法）。如果子节点是目录，则递归调用saveImage方法。注意:这个方法并不将当前节点信息写入fsimage文件。

private void saveImage(INodeDirectory current, DataOutputStream out,

boolean toSaveSubtree, Counter counter) throws IOException {

// 写inode id

out.writeLong(current.getId());

if (!toSaveSubtree) {

return;

}

final ReadOnlyList<INode> children = current.getChildrenList(null);

int dirNum = 0;

List<INodeDirectory> snapshotDirs = null;

if (current instanceof INodeDirectoryWithSnapshot) {

snapshotDirs = new ArrayList<INodeDirectory>();

((INodeDirectoryWithSnapshot) current).getSnapshotDirectory(

snapshotDirs);

dirNum += snapshotDirs.size();

}

// 2.写 INodeDirectorySnapshottable

// Snapshots

if (current instanceof INodeDirectorySnapshottable) {

INodeDirectorySnapshottable snapshottableNode =

(INodeDirectorySnapshottable) current;

SnapshotFSImageFormat.saveSnapshots(snapshottableNode, out);

} else {

out.writeInt(-1); // # of snapshots

}

// 3. 调用saveCildren()写所有的子节点，最终调用saveInode2Image()

dirNum += saveChildren(children, out, counter);

// 4. 写 DirectoryDiff lists

SnapshotFSImageFormat.saveDirectoryDiffList(current, out, referenceMap);

// 如果子节点是目录，则递归调用saveImage()保存子目录

out.writeInt(dirNum); // the number of sub-directories

for(INode child : children) {

if(!child.isDirectory()) {

continue;

}

// make sure we only save the subtree under a reference node once

boolean toSave = child.isReference() ?

referenceMap.toProcessSubtree(child.getId()) : true;

saveImage(child.asDirectory(), out, toSave, counter);

}

// 将snapshot目录保存，同样是递归调用saveImage()

if (snapshotDirs != null) {

for (INodeDirectory subDir : snapshotDirs) {

// make sure we only save the subtree under a reference node once

boolean toSave = subDir.getParentReference() != null ?

referenceMap.toProcessSubtree(subDir.getId()) : true;

saveImage(subDir, out, toSave, counter);

}

}

}

6) 保存构建中的节点 ： 这里调用了FSNameSystem.saveFilesUnderConstruction()方法，在这个方法内又调用了FSImageSerialization.writeINodeUnderConstruction()这个静态方法,非常简单，这里就不赘述了。

1.1.2.1.2 FSImageSerialization

上一节中，保存inode信息用到了FSImageSerialization类，这个类是真正对fsimage文件进行写入操作的类，通过这个类当中的方法可以了解到fsimage文件当中对于不同的类型的Inode文件，是以什么格式存储的。

入口方法：saveInode2Image，下面是实现：

public static void saveINode2Image(INode node, DataOutput out,

boolean writeUnderConstruction, ReferenceMap referenceMap)

throws IOException {

if (node.isReference()) {

writeINodeReference(node.asReference(), out, writeUnderConstruction,

referenceMap);

} else if (node.isDirectory()) {

writeINodeDirectory(node.asDirectory(), out);

} else if (node.isSymlink()) {

writeINodeSymlink(node.asSymlink(), out);

} else if (node.isFile()) {

writeINodeFile(node.asFile(), out, writeUnderConstruction);

}

}

可以看到，这里将传入的节点进行判断，分为：reference,directory,symlink,file等情况，调用对应的方法。当然root根节点是作为directory情况处理。下面这个方法安装fsimage格式写入文件

writeINodeReference()

writeINodeDirectory()

writeINodeSymlink()

writeINodeFile()

1.1.2.2 加载fsimage

从磁盘上读取fsimage文件，并将元数据加载到namenode的内存当中，入口方法为LoadFSiamge()，总体来说进行了两个工作：加载fsImage和合并editlog

boolean loadFSImage(FSNamesystem target, MetaRecoveryContext recovery)

throws IOException {

// 准备工作 ...

// 获取editlog文件io流

initEditLog();

if (LayoutVersion.supports(Feature.TXID\_BASED\_LAYOUT,

getLayoutVersion())) {

long toAtLeastTxId = editLog.isOpenForWrite() ? inspector.getMaxSeenTxId() : 0;

editStreams = editLog.selectInputStreams(

imageFiles.get(0).getCheckpointTxId() + 1,

toAtLeastTxId, recovery, false);

} else {

editStreams = FSImagePreTransactionalStorageInspector

.getEditLogStreams(storage);

}

// ...

// 调用方法loadFSImageFile()，加载fsimage文件

for (int i = 0; i < imageFiles.size(); i++) {

try {

imageFile = imageFiles.get(i);

loadFSImageFile(target, recovery, imageFile);

break;

} catch (IOException ioe) {

LOG.error("Failed to load image from " + imageFile, ioe);

target.clear();

imageFile = null;

}

}

// 如果加载失败，则记录

if (imageFile == null) {

FSEditLog.closeAllStreams(editStreams);

throw new IOException("Failed to load an FSImage file!");

}

// 调用loadEdit文件合并editlog

long txnsAdvanced = loadEdits(editStreams, target, recovery);

needToSave |= needsResaveBasedOnStaleCheckpoint(imageFile.getFile(),

txnsAdvanced);

editLog.setNextTxId(lastAppliedTxId + 1);

return needToSave;

}

加载fsimage主要是方法loadfsimage()当中进行，这个方法调用了FSImageFormat.loader加载到fsimage，同时将加载的md5同md5文件记录的值进行比对，然后更新lastAppliedTxid。加载fsimage的大部分操作都是在loader.load方法当中完成：

这个方法首先读入fsimage当中的文件头，并根据读入的值设置namesystem当中的对应值

加载inode，如下段代码所示

if (LayoutVersion.supports(Feature.FSIMAGE\_NAME\_OPTIMIZATION,

imgVersion)) {

if (supportSnapshot) {

loadLocalNameINodesWithSnapshot(numFiles, in, counter);

} else {

loadLocalNameINodes(numFiles, in, counter);

}

} else {

loadFullNameINodes(numFiles, in, counter);

}

loadFilesUnderConstruction(in, supportSnapshot, counter);

loadLocalNameINodesWithSnapshot()方法首先调用loadRoot()加载根节点，然后调用方法loadDirectoryWithSnapshot()

private void loadDirectoryWithSnapshot(DataInput in, Counter counter)

throws IOException {

// Step 1. 确定parent节点，这里注意是通过id获得的

long inodeId = in.readLong();

final INodeDirectory parent = this.namesystem.dir.getInode(inodeId)

.asDirectory();

// 检查真个subTree是否已经保存

boolean toLoadSubtree = referenceMap.toProcessSubtree(parent.getId());

if (!toLoadSubtree) {

return;

}

// Step 2. 如果父节点支持snapshot，则加载snapshot

int numSnapshots = in.readInt();

if (numSnapshots >= 0) {

final INodeDirectorySnapshottable snapshottableParent

= INodeDirectorySnapshottable.valueOf(parent, parent.getLocalName());

// load snapshots and snapshotQuota

SnapshotFSImageFormat.loadSnapshotList(snapshottableParent,

numSnapshots, in, this);

if (snapshottableParent.getSnapshotQuota() > 0) {

// add the directory to the snapshottable directory list in

// SnapshotManager. Note that we only add root when its snapshot quota

// is positive.

this.namesystem.getSnapshotManager().addSnapshottable(

snapshottableParent);

}

}

// Step 3. 加载所有孩子节点

loadChildren(parent, in, counter);

// Step 4. 加载目录差异列表

SnapshotFSImageFormat.loadDirectoryDiffList(parent, in, this);

// 递归调用加载所有子节点

int numSubTree = in.readInt();

for (int i = 0; i < numSubTree; i++) {

loadDirectoryWithSnapshot(in, counter);

}

}

这里的LoadChildren()节点通过构造Inode，调用addprent方法添加Parent的孩子节点列表，完成Inode tree重建。

private int loadChildren(INodeDirectory parent, DataInput in,

Counter counter) throws IOException {

int numChildren = in.readInt();

for (int i = 0; i < numChildren; i++) {

// load single inode

INode newNode = loadINodeWithLocalName(false, in, true, counter);

addToParent(parent, newNode);

}

return numChildren;

}

加载InodeUnderConstruction,调用方法LoadFileunderConstruction加载构建中的inode，首先从image文件中构造出InodeFIleUnderConstruction，然后将namespace当中对应inode替换为inodeFileUnderConstructin。最后更新租约管理器当中的信息。

private void loadFilesUnderConstruction(DataInput in,

boolean supportSnapshot, Counter counter) throws IOException {

// ...

for (int i = 0; i < size; i++) {

// 读取image文件并重建InodeFileUnderConstrution节点

INodeFileUnderConstruction cons = FSImageSerialization

.readINodeUnderConstruction(in, namesystem, getLayoutVersion());

counter.increment();

// 确认这个文件是否在namespace当中

String path = cons.getLocalName();

final INodesInPath iip = fsDir.getLastINodeInPath(path);

INodeFile oldnode = INodeFile.valueOf(iip.getINode(0), path);

cons.setLocalName(oldnode.getLocalNameBytes());

cons.setParent(oldnode.getParent());

//将inode文件更新为inodeFileUnderConstruction

if (oldnode instanceof INodeFileWithSnapshot) {

cons = new INodeFileUnderConstructionWithSnapshot(cons,

((INodeFileWithSnapshot)oldnode).getDiffs());

}

fsDir.replaceINodeFile(path, oldnode, cons);

// 添加租约信息

namesystem.leaseManager.addLease(cons.getClientName(), path);

}

}

更新安全相关信息

1.1.2.3 加载editlog

将editlog当中记录的操作同当前namespace进行合并，这里调用了FSEditLogLoader.load()。这里进行操作比较冗长，基本归纳为从ediltLog当中读取一个操作，然后调用对应的FSDirectory方法进行更新。

/ 从editlog当中读取一个操作

FSEditLogOp op;

try {

op = in.readOp();

if (op == null) {

break;

}

} catch (Throwable e) {

//...

}

//...

// 在当前FSDirectory当中记载对应的操作

long inodeId = applyEditLogOp(op, fsDir, in.getVersion(), lastInodeId);

if (lastInodeId < inodeId) {

lastInodeId = inodeId;

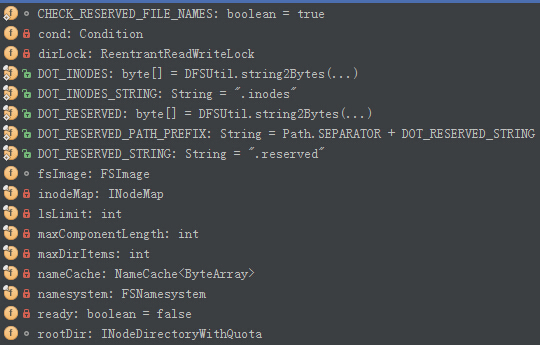
}

这里的applyEditLogOp()方法就是对纯如的操作数做了一个switch，然后进行对应的操作，比较简单，我们就不多叙述了，感兴趣的同学可以自己看代码。

1.1.3 FSdirectory

Namenode一个重要功能，是对HDFS整个文件系统的名字空间，也就是/为根的整个目录树的管理。这个功能通过FSdirectory类来进行的。FSdirectory使用门面设计模式（facade）,为子系统的一组接口提供一个一致的界面，FSdirectory是HDFS设计者为了屏蔽复杂性，针对目录树的管理功能，对外提供的一个门面类。FSDirectory使用了两个重要的类，Inode和FSImage。

1) 字段



INodeDirectoryWithQuota rootDir，这个是整个文件目录树的根节点，

FSImage fsImage：FSdirectory操作目录树时，通过FSImage提供的功能记录操作，参考FSEditlog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) 方法

FSDiretory的方法非常多，主要是封装了对于文件目录树的操作，例如增删改查等。下面以几个方法为例，讲解下提供的操作。

addchild()

主要作用是添加一个child节点，调用InodeDirecotry.addchild()，并且更新quota信息，同时更新InodeMap

private boolean addChild(INodesInPath iip, int pos,

INode child, boolean checkQuota) throws QuotaExceededException {

// ...

// 检查是否有足够的空间

if (checkQuota) {

verifyMaxComponentLength(child.getLocalNameBytes(), inodes, pos);

verifyMaxDirItems(inodes, pos);

}

// 检查Inode的名称

verifyINodeName(child.getLocalNameBytes());

final Quota.Counts counts = child.computeQuotaUsage();

updateCount(iip, pos,

counts.get(Quota.NAMESPACE), counts.get(Quota.DISKSPACE), checkQuota);

final INodeDirectory parent = inodes[pos-1].asDirectory();

boolean added = false;

try {

// 将Inode添加到父节点的孩子节点列表

added = parent.addChild(child, true, iip.getLatestSnapshot(),

inodeMap);

} catch (QuotaExceededException e) {

updateCountNoQuotaCheck(iip, pos,

-counts.get(Quota.NAMESPACE), -counts.get(Quota.DISKSPACE));

throw e;

}

if (!added) {

updateCountNoQuotaCheck(iip, pos,

-counts.get(Quota.NAMESPACE), -counts.get(Quota.DISKSPACE));

} else {

iip.setINode(pos - 1, child.getParent());

addToInodeMap(child);

}

return added;

}

addBlock()

这个方法在指定的Inode上添加一个Block，使用namesystem的BLockManager

BlockInfo addBlock(String path, INodesInPath inodesInPath, Block block,

DatanodeDescriptor targets[]) throws IOException {

waitForReady();

writeLock();

try {

final INodeFileUnderConstruction fileINode =

INodeFileUnderConstruction.valueOf(inodesInPath.getLastINode(), path);

// 检查空间是否够用

updateCount(inodesInPath, 0, fileINode.getBlockDiskspace(), true);

// 构造BLock对象

BlockInfoUnderConstruction blockInfo =

new BlockInfoUnderConstruction(

block,

fileINode.getFileReplication(),

BlockUCState.UNDER\_CONSTRUCTION,

targets);

// 使用BlockManager添加BLock

getBlockManager().addBlockCollection(blockInfo, fileINode);

// 将Block同fileInode关联起来

fileINode.addBlock(blockInfo);

if(NameNode.stateChangeLog.isDebugEnabled()) {

NameNode.stateChangeLog.debug("DIR\* FSDirectory.addBlock: "

+ path + " with " + block

+ " block is added to the in-memory "

+ "file system");

}

return blockInfo;

} finally {

writeUnlock();

}

}

SetOwner()

修改文件的所有者，非常简单，使用editlog记录了setowner的操作

void setOwner(String src, String username, String groupname)

throws FileNotFoundException, UnresolvedLinkException,

QuotaExceededException, SnapshotAccessControlException {

writeLock();

try {

unprotectedSetOwner(src, username, groupname);

} finally {

writeUnlock();

}

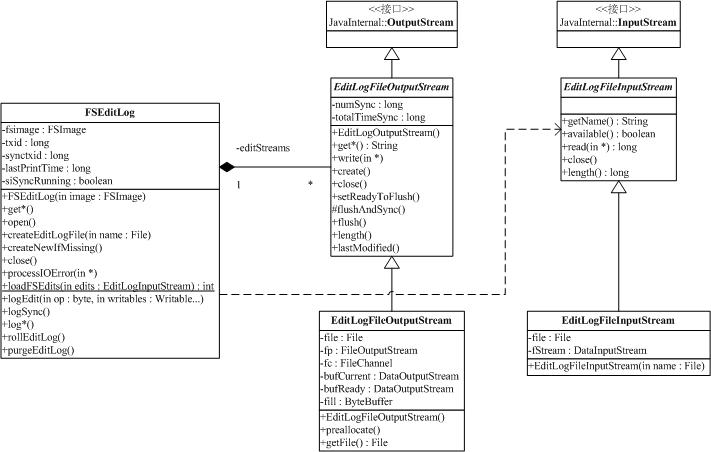
fsImage.getEditLog().logSetOwner(src, username, groupname);

}

整体上来说，FSDirectory的功能实现还是比较简单，都是基于我们已经了解过的知识，其他的方法我们就不再分析了，大家感兴趣可以自己学习一下。

1.1.4 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：



对FSEditLog的读写用到了两个类EditLogFileInputStream和EditLogFileOutputStream

1)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。

在EditsDoubleBuffer中有两块缓冲区，一块缓冲区用来进行写入操作，另一块缓冲区用来同步数据。

private TxnBuffer bufCurrent; // 正在写入的缓冲区

private TxnBuffer bufReady; // 准备好同步的缓冲区

private final int initBufferSize; //缓冲区的大小

setReadyToFlush()

要进行同步操作时，将写入缓冲区改变为同步缓冲区，以同步数据

public void setReadyToFlush() {

assert isFlushed() : "previous data not flushed yet";

TxnBuffer tmp = bufReady;//交换两个缓冲区

bufReady = bufCurrent;

bufCurrent = tmp;

}

flushTo()

同步方法，将同步缓冲区中的数据写入输出流

public void flushTo(OutputStream out) throws IOException {

bufReady.writeTo(out); // 数据写入文件

bufReady.reset(); // 缓冲区当中的数据清空

}

2)EditLogFileOutputstream

这个类的主要作用是将操作写入一个EditLog文件当中。父类是抽象类EditLogOutputStream ，实现了父类当中的抽象方法。

字段：

file：editlog文件

fp：对应editlog文件的输出流

fc：对应editlog输出流的通道

doubleBuf：一个具有两块缓冲区的缓存，数据必须先写入缓存，然后再由缓存同步到磁盘上。（这里的实现请参考小节EditsDoubleBuffer实现）

fill ：用来扩充editlog文件大小的数据块。当要同步操作时，如果editlog文件不够大，则使用fill来扩充editlog。（参考preallocate()方法实现）

private File file;

private FileOutputStream fp;

private FileChannel fc;

private EditsDoubleBuffer doubleBuf;

static ByteBuffer fill = ByteBuffer.allocateDirect(MIN\_PREALLOCATION\_LENGTH);

构造方法：

先看下static的代码段，将fill区用FSEditLogOpCodes.OP\_INVALID字节填满。

static {

fill.position(0);

for (int i = 0; i < fill.capacity(); i++) {

fill.put(FSEditLogOpCodes.OP\_INVALID.getOpCode());

}

}

FSEditLogOpCodes是一个枚举类型，每种情况使用一个Byte表示。

public enum FSEditLogOpCodes {

OP\_INVALID ((byte) -1),

OP\_ADD ((byte) 0),

OP\_RENAME\_OLD ((byte) 1),

//..

}

构造方法比较简单，初始化了所有的字段。

public EditLogFileOutputStream(Configuration conf, File name, int size)

throws IOException {

super();

// ...

file = name;

doubleBuf = new EditsDoubleBuffer(size);

RandomAccessFile rp;

if (shouldSyncWritesAndSkipFsync) {

rp = new RandomAccessFile(name, "rws");

} else {

rp = new RandomAccessFile(name, "rw");

}

fp = new FileOutputStream(rp.getFD()); // open for append

fc = rp.getChannel();

fc.position(fc.size());

}

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

}

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

}

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();

}

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

//..

}

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;

}

}

3)EditLogFileInputStream

EditLogFileInputStream的实现非常简单，都是返回了特定的字段，或者调用了reader的readOp()方法获得下一个operation。这里就不详细分析了，大家可以自己阅读代码。

1.2 BlockManager

Namenode主要维护文件系统的目录树和数据块同数据节点之间的对应关系。上一节介绍了FSDirectory，负责第一个功能的实现，BlockMananger实现数据块以及Datanode的管理。

1.2.1 Block

Block用于HDFS抽象数据块，BlockInfo.BlockMap等是对HDFS的Block抽象相关的类。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。

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（联想GenerationStamp这个类）

private long blockId;

private long numBytes;

private long generationStamp;

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)BlockInfo

扩展了Block，是block的补充说明。BlockInfo保存了该Block所属的inode,也就是bc字段。同时保存了这个block的datanode信息，保存在triplets数组中。

private BlockCollection bc;

private Object[] triplets;

triplets数组是个比较有趣的数据结构。triplets数组有3 \* replication个元素（replication是这个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个副本，同样完成双向链表功能）。

BlockInfo当中的方法大都是维护数据结构triplets – 增加datanode，或者修改prev\next blockinfo。

同时需要注意一下 BlockInfoUnderConstruction()这个方法。将一个完整BlockInfo转换为构建当中的BlockInfoUnderConstruction。

public BlockInfoUnderConstruction convertToBlockUnderConstruction(

BlockUCState s, DatanodeDescriptor[] targets) {

if(isComplete()) {

return new BlockInfoUnderConstruction(

this, getBlockCollection().getBlockReplication(), s, targets);

}

// the block is already under construction

BlockInfoUnderConstruction ucBlock = (BlockInfoUnderConstruction)this;

ucBlock.setBlockUCState(s);

ucBlock.setExpectedLocations(targets);

return ucBlock;

}

2) BlockInfoUnderConstruction

HDFS在加载fsImage时，如果当前加载的文件为正在构建当中的文件-InodeFileUnderConstrution，则将该InodeFiled的最后一个数据块设置为BlockInfoUnderConstrution，表明当前数据块处于正在构建当中。这里要注意InodeFileUnderConstrution除最后一个数据块之外的数据块均为正常的BlockInfo。

int numBlocks = in.readInt();

BlockInfo[] blocks = new BlockInfo[numBlocks];

Block blk = new Block();

int i = 0;

for (; i < numBlocks-1; i++) {

blk.readFields(in);

blocks[i] = new BlockInfo(blk, blockReplication);

}

// 最后一个数据块为 UNDER\_CONSTRUCTION

if(numBlocks > 0) {

blk.readFields(in);

blocks[i] = new BlockInfoUnderConstruction(

blk, blockReplication, BlockUCState.UNDER\_CONSTRUCTION, null);

}

当datanode汇报构建中副本的状态为FINALIZED，并且当前数据块的状态为COMMITTD（参考Block与Replica小节），数据块副本数目大于等于要求的副本数目时，namenode将当前数据块由blockInfoUnderConstrution转变为blockinfo，并将blockInfoz状态设置为COMPLETE。

if(storedBlock.getBlockUCState() == BlockUCState.COMMITTED &&

numLiveReplicas >= minReplication) {

storedBlock = completeBlock((MutableBlockCollection)bc, storedBlock, false);

}

这里还需要注意在BlockInfoUnderConstruction当中记录了所有正在备份的副本。通过数据结构replicas。

private List<ReplicaUnderConstruction> replicas;

3)BlocksMap

类的作用是存放Block->BlockInfo(Block元数据)的对应关系，GSet是一个提供类似映射功能的集合。

private final int capacity;

private volatile GSet<Block, BlockInfo> blocks;

BlocksMap当中的方法大多也都是维护 **blocks** 这个字段。

BlocksMap类实现虽然非常的简单，但确实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。

为什么BlocksMap是Block -> BlockInfo的对应，是因为BlockInfo当中数据节点的信息都是在datanode启动时上报的，而namenode当中关于数据块的信息只有Block类中维护的这么多。所以元数据通过Block -> BlockInfo的对应关系来维护，而BlockInfo的信息是不断更新的(通过datanode上报的信息)。

3) Block与Replica

在HDFS当中Block和replica需要区分，将namenonde数据称为block，datanode中的数据为replica。

block的状态为：

static public enum BlockUCState {

COMPLETE,

UNDER\_CONSTRUCTION,

UNDER\_RECOVERY,

COMMITTED;

}

**COMPLETE** ：block的length和gs不再发生变化，并且namenode已经收到至少有一个datanode报告有finalized状态的replica（datanode上的replica状态发生变化会通过RPC blockReceivedAndDeleted向NN报告）。一个complete的block会保存finalized的replica的locations在namenode的内存中。只有当文件的所有的block都是complete的，该文件才能被close。  
 **UNDER\_CONSTRUCTION** ： 文件被create或者append时，正在被写入的block就处于under\_construction状态。该状态的length和gs都不是finalized的，但是处在该状态的block对于read来说是visible的。  
 **UNDER\_RECOVERY** ： 如果一个file的last block处于under\_construction状态的时候，client异常退出，lease超过softLimit过期，那么该block就需要走下面要说的Lease recovery和Block recovery流程释放lease关闭file。那么正在走Lease recovery和Block recovery流程的block就处于under\_recovery状态。  
 **COMMITTED** ：client端在写文件的时候，每次请求新的block（addBlock RPC请求）或者close文件时，都会顺带把previous block进行commit操作（previous block从under\_construction状态转化成committed状态）。这个时候Client已经把所有的该block的byte都发送给了DN组成的pipeline，已经收到ACK请求。但是NN还没有收到任何一个DN说有finalized replica。

2)replica

replica的状态分为5中状态

static public enum ReplicaState {

/\*\* Replica is finalized. The state when replica is not modified. \*/

FINALIZED(0),

/\*\* Replica is being written to. \*/

RBW(1),

/\*\* Replica is waiting to be recovered. \*/

RWR(2),

/\*\* Replica is under recovery. \*/

RUR(3),

/\*\* Temporary replica: created for replication and relocation only. \*/

TEMPORARY(4);

private int value;

//...

}

FINALIZED ： 副本已完成写操作，不在修改

RBW（ReplicaBeingWritten） ： 刚刚被create或者append的replica，处在write的pipeline中，正在被写入。但是byte还是visible to read的。

RUR（ReplicaUnderRecovery） ： Lease过期之后发生Lease和Block recovery时replica所处的状态。

RWR(ReplicaWaitingToBeRecovered)：如果一个DN挂掉并且重启之后，所有RBW的replica将会转换为RWR。RWR的replica不会出现在pipeline中，结果就是等着Lease recovery恢复。

Temporary(ReplicaInPipeline)：DN之间传输replica（例如cluster rebalance）时，正在传输的就是处在Temporary。和RBW不同的是，它对read不是visible的，DN如果重启直接删除处于Temporary状态的replica。

1.2.2 BlockManager

Namenode当中对于数据块的管理都是通过BlockManager类

1.2.2.1 数据结构

1) block集合：下面这段代码是BlockManager当中存储不同状态Block的集合变量。

final CorruptReplicasMap corruptReplicas = new CorruptReplicasMap();

private final InvalidateBlocks invalidateBlocks;

private final Set<Block> postponedMisreplicatedBlocks = Sets.newHashSet();

public final Map<String, LightWeightLinkedSet<Block>> excessReplicateMap =

new TreeMap<String, LightWeightLinkedSet<Block>>();

public final UnderReplicatedBlocks neededReplications = new UnderReplicatedBlocks();

final PendingReplicationBlocks pendingReplications;

private final Set<Block> postponedMisreplicatedBlocks = Sets.newHashSet();

corruptReplicas ： 损坏的数据块副本（corruptReplica）集合。

excessReplicateMap ： 多余的副本集合。减低文件的备份数会产生多余的副本。用TreeMap存放 StorageId(标识datanode) -> extraBlocks(多余副本集合)的映射。BlockMangaer会定期将这个队列当中的副本加入invalidateBlocks集合。

invalidateBlocks ： 需要删除的block集合。需要删除的block来自corruptReplicas以及excessReplicateMap这两个集合。这里还需要特别注意，加入这个队列当中的Block，在namenode的BlockMap当中是已经不存在了。

neededReplications ： 等待复制的副本，正在等待生成复制请求。数据块复制是为了让数据块的副本数满足文件的副本系数。（引用1）。

pendingReplications ： 已经生成复制请求的数据块副本保存在pendingReplications中，当复制请求生成后，数据块的信息会从neededReplications取出，放入pendingReplications。

postponedMisreplicatedBlocks ： 当namenode发生失败，进行了active与standby切换时。多余的副本不能进行删除，先放入这个数据结构当中，直到这个副本对应数据块的所有副本都进行块汇报。为什么要这样，参考后面的具体分析。

可以看到的这里的很多数据结构都没有简单的使用集合类，而是重新写了类作为容器。下面我们就分析下上面代码当中出现过的容器。

CorruptReplicasMap

损坏的数据块副本（corruptReplica）集合。当datanode发现有损坏的副本时(副本的大小与时间戳同数据块不同)，则会上报blockmanager，将损坏的副本加入这个数据结构当中。这个数据结构存储的是损坏数据块副本的blocks -> datanodedescriptor的映射关系，使用了一个TreeMap（我们知道Block是实现了Comparable接口的，按照BlockId的大小排序）。

private SortedMap<Block, Collection<DatanodeDescriptor>> corruptReplicasMap =

new TreeMap<Block, Collection<DatanodeDescriptor>>();

InvalidateBlocks

需要删除的block集合。使用一个TreeMap存放StorageId(标识datanode) -> invalidateBlocks(多余副本集合)的映射。InvalidateBlocks当中的数据块来自于corruptReplicas以及excessReplicateMap这两个集合。

private final Map<String, LightWeightHashSet<Block>> node2blocks =

new TreeMap<String, LightWeightHashSet<Block>>();

UnderReplicatedBlocks

等待复制的副本。和上面的数据结构相比，UnderReplicatedBlocks实现相对复杂一点，维护了一个优先级队列priorityQueues – 是一个列表，有5个子队列。每一个队列对应一个优先级列表，其中0为最高，5为最低。

private List<LightWeightLinkedSet<Block>> priorityQueues

= new ArrayList<LightWeightLinkedSet<Block>>();

那么如何判断一个副本的优先级呢：

优先级0 ： 需要立刻备份的block。这个block只有一个拷贝，或者这个block没有活着的拷贝并且有一个拷贝所在datanode正在离线中。这些Block很有可能丢失。

优先级1 ： block的副本数极低，当实际副本：期望副本的比例小于1:3时，加入这个队列。

优先级2 ： block正在备份中，但并未达到优先级1队列当中的比例

优先级3 ： block的副本数已经够了，但是block副本的分布不是很好。当一个机架或者交换机宕机很有可能造成block丢失

优先级4 ： 已经损坏的block。这里的策略是将损坏的block放入这个队列当中，但将没有损坏的副本给予更高的优先级。

PendingReplicationBlocks

存放已经生成复制请求的数据块。之所有这个数据结构是因为虽然UnderReplicatedBlocks队列当中已经放入了要复制的数据块，但是在数据块复制过程当中很有可能出现错误。所以将已经生成复制请求的数据块放入的pendingReplicationBlocks中缓存，如果出现复制失败的情况，会重新加入UnderReplicatedBlocks队列。

PendingReplicationBlocks除了保存正在进行备份的数据块外。还更新每个请求发起的时间，同时启动一个线程（timerThread）标识那些已经超时的请求(timeOutItems)。

private Map<Block, PendingBlockInfo> pendingReplications;

private ArrayList<Block> timedOutItems;

Daemon timerThread = null;

这里的PendingBlockInfo类就记录了时间戳以及这个备份这个Block的Datanode

private long timeStamp;

private final List<DatanodeDescriptor> targets;

postponedMisreplicatedBlocks

当namenode发生失败，进行了active与standby切换时。多余的副本不能进行删除，先放入这个数据结构当中，直到这个副本对应数据块的所有副本都进行块汇报。

为什么要这样设计，考虑下面的这个场景：

blockA有两个副本，副本1在datanod1上，副本2在datanode2上。

这时，namenode1发出删除指令，删除datanode1上的副本1

发出删除指令后，namenode1发生错误，切换至namenode2。

这时datanode1并没有进行快汇报，namenode2并不知道datanode1上已经删除了副本1。所以namenode2向datanode2发出删除操作。

datanode2删除复本2，数据块丢失。

为了解决上述的这种问题，当namenode发生active -> standby切换时，会将所有datanode设置为stale状态。当一个datanode为stale状态，那么这个datnode上的所有副本被设置为stale状态，如果一个namenode上的block有至少一个stale状态的副本，那么这个block为stale。对于stale状态的block是可以进行overReplcate操作的，而是放入postponedMisreplicatedBlocks当中。当stale的datanode进行了块汇报操作之后，会重新扫描postponedMisreplicatedBlocks队列当中的数据块。

if (numCurrentReplica > expectedReplication) {

if (num.replicasOnStaleNodes() > 0) {

//副本数多余，并且存在stale的副本，则放入POSTPONE队列

return MisReplicationResult.POSTPONE;

}

processOverReplicatedBlock(block, expectedReplication, null, null);

return MisReplicationResult.OVER\_REPLICATED;

}

2) Block计数

下面这段代码主要是BlockManager当中对于不同状态Block的计数变量。

private volatile long pendingReplicationBlocksCount = 0L;

private volatile long corruptReplicaBlocksCount = 0L;

private volatile long underReplicatedBlocksCount = 0L;

private volatile long scheduledReplicationBlocksCount = 0L;

private AtomicLong excessBlocksCount = new AtomicLong(0L);

private AtomicLong postponedMisreplicatedBlocksCount = new AtomicLong(0L);

pendingReplicationBlocksCount ：正在进行Replication的BLock数量 （对应于pendingReplications）

corruptReplicaBlocksCount ：损坏的Block数量 (对应于corruptReplicas)

underReplicatedBlocksCount ： 需要进行Replication的Block数量 （对应于neededReplications）

excessBlocksCount ：多余的BLock(对应于excessReplicateMap)

postponedMisreplicatedBlocksCount : postponed队列中block数量

scheduledReplicationBlocksCount ： 计算出来安排进行Replication的数量

3) BlocksMap

当然还有最重要的维护Block -> BlockInfo关系的BlocksMap

final BlocksMap blocksMap;

4) 数据块副本状态（TODO构建状态，撤销状态）

在blockmanager当中一个数据块副本可能存在多种状态，并且这些状态之间是可以转移的。但HDFS代码当中并没有使用一个枚举类给出数据块副本的状态，以及状态之间的转移。而是通过集合（例如上一节描述的不同状态的集合），或者不同的类-BlockUnderConstrution&Block，以及副本所存在datanode的状态（有可能是正在撤销的datanode）等等。

我们可以知道下面这五种状态是通过BlockManager当中对应的数据结构来标识的。

正常副本 ： 存在BlocksMap当中的正常的数据块副本

损坏的副本 ： 存在corruptReplicas队列中的损坏的数据块。损坏的副本一般由datanode的数据块扫描器，数据块接收器或者客户端的数据块校验过程发现（引用1）。损坏副本状态都会变为等待删除副本状态。

多余的副本 ： 存放在excessReplicateMap当中的数据块，当前数据块的副本数目多余设置的数目时。这种情况可能发生在更改数据块副本数量之后。多余副本状态会变为等待删除副本状态。

等待删除的副本 ： 存放在invalidateBlocks当中，需要由BlockManager命令对应的datanode删除的副本

等待复制副本 ： 存放在neededReplications当中。还没有生成复制命令。等待复制副本会转移到正在复制副本状态，当发出复制命令之后。

正在复制副本 ： 已生成复制命令，但并不知道复制动作是否成功。如果复制动作成功，那么正在复制副本就会转移至正常副本状态。如果复制失败，则转至等待复制副本状态。

推迟操作的副本 ： 副本数过多，但是存在stale副本，所以推迟处理，等待stale的datanode进行块汇报。

同时我们知道，在HDFS append逻辑时，会出现 正在构建状态 副本， 以及 恢复状态 副本。这些信息保存在租约管理器当中。

正在构建状态副本 ： 当前数据块正在构建当中

恢复状态副本 ： 参考 （http://yanbohappy.sinaapp.com/?p=175）

当datanode被撤销时（当前datanode离线，或者放入exclude文件中时），会存 撤销节点副本 ，这些副本由于节点下架而进入“最终状态”，或者由于节点重新进入工作状态而返回“正常副本”状态。

1.2.2.2 Replication Thread

BlockManager的active方法中启动了两个线程：

public void activate(Configuration conf) {

pendingReplications.start();

datanodeManager.activate(conf);

this.replicationThread.start();

}

1)pendingReplications线程用户处理已经超时的任务，然后放入指定的数据结构中

2) replicationMonitor

replicationThread的实现类是BlockManager当中的内部类ReplicationMonitor，是一个线程类，这个线程类周期性的调用computeDataNodeWork以及processpendingReplications()

public void run() {

while (namesystem.isRunning()) {

try {

computeDatanodeWork();

processPendingReplications();

Thread.sleep(replicationRecheckInterval);

} catch (Throwable t) {

//... 异常处理策略

}

}

}

ComputeDataNodeWork

首先调用needReplications.chooseUnderReplicatedBlocks()方法， 这个方法从needRelications的优先级队列当中选出需要进行备份的副本集合（按照优先级放在不同的集合当中）。之后调用**computeReplicationWorkForBlocks()**方法进行备份操作。

int computeReplicationWork(int blocksToProcess) {

List<List<Block>> blocksToReplicate = null;

namesystem.writeLock();

try {

// Choose the blocks to be replicated

blocksToReplicate = neededReplications

.chooseUnderReplicatedBlocks(blocksToProcess);

} finally {

namesystem.writeUnlock();

}

return computeReplicationWorkForBlocks(blocksToReplicate);

}

这里的**computeReplicationWorkForBlocks()**方法比较简单，就是从needReplications优先级队列当中取出**blockToProcess**个元素，并按照优先级放入**blocksToreplicate**中，这里就不再赘述。

我们重点看下computeReplicationWorkForBlocks()的实现。这里的实现分为三个部分：a.选择源节点 b.选择目标节点 c.进行备份操作

选择源节点 – 选择备份数据的源节点，确定备份数量，添加任务

for (int priority = 0; priority < blocksToReplicate.size(); priority++) {

for (Block block : blocksToReplicate.get(priority)) {

// 通过blocksMap获得该block所属inode

bc = blocksMap.getBlockCollection(block);

// 如果文件正在构建当中，则不可以备份，从备份列表当中删除

if(bc == null || bc instanceof MutableBlockCollection) {

neededReplications.remove(block, priority); // remove from neededReplications

neededReplications.decrementReplicationIndex(priority);

continue;

}

// 需要的副本数

requiredReplication = bc.getBlockReplication();

// 调用chooseSourceDatanode获取拷贝副本的源节点

containingNodes = new ArrayList<DatanodeDescriptor>();

liveReplicaNodes = new ArrayList<DatanodeDescriptor>();

NumberReplicas numReplicas = new NumberReplicas();

srcNode = chooseSourceDatanode(

block, containingNodes, liveReplicaNodes, numReplicas,

priority);

// 当前有效的副本数 = 已经备份的数目 + 正在备份的副本

numEffectiveReplicas = numReplicas.liveReplicas() +

pendingReplications.getNumReplicas(block);

// 如果副本数已经够了，则不用进行备份操作

if (numEffectiveReplicas >= requiredReplication) {

if ( (pendingReplications.getNumReplicas(block) > 0) ||

(blockHasEnoughRacks(block)) ) {

neededReplications.remove(block, priority); // remove from neededReplications

neededReplications.decrementReplicationIndex(priority);

}

}

if (numReplicas.liveReplicas() < requiredReplication) {

additionalReplRequired = requiredReplication

- numEffectiveReplicas;

} else {

additionalReplRequired = 1; // Needed on a new rack

}

// 在工作队列当中添加备份任务，任务会在下次心跳时带回datanode

work.add(new ReplicationWork(block, bc, srcNode,

containingNodes, liveReplicaNodes, additionalReplRequired,

priority));

}

}

在选择源节点操作时，有个非常重要的函数**chooseSourceDatanode()**，这个方法计算出当前block有多少个有效备份、无效备份、以及超出的备份数。同时选择一个源节点作为备份的数据源。这里选择策略很简单，如果当前的节点是DECOMMISSION\_INPROGRESS状态，那么就选择当前节点，因为这个节点没有写操作占用带宽。其他情况时，随机选择一个节点就好。

选择目标节点 – 这里从datanode当中选择出需要进行备份操作的节点

HashMap<Node, Node> excludedNodes

= new HashMap<Node, Node>();

for(ReplicationWork rw : work){

// 将已有节点加入列表

excludedNodes.clear();

for (DatanodeDescriptor dn : rw.containingNodes) {

excludedNodes.put(dn, dn);

}

// 调用chooseTarget()方法，选择目标节点

rw.targets = blockplacement.chooseTarget(rw.bc,

rw.additionalReplRequired, rw.srcNode, rw.liveReplicaNodes,

excludedNodes, rw.block.getNumBytes());

}

这里的**chooseTarget()**方法选择需要进行副本操作的节点。策略是第一个副本放在本机或者同机架的随机datanode上；第二个副本放在不同机架的datanode上；第三个副本放在第二个副本同机架的另一个datanode上。

进行备份操作 – 给datanode发送指令（在下一次心跳中）；将备份节点放入pendingReplications队列；如果副本足够则从neededReplications队列当中移除。

// 将副本加入datanode的备份队列，在下次心跳时，发出备份指令

rw.srcNode.addBlockToBeReplicated(block, targets);

scheduledWork++;

for (DatanodeDescriptor dn : targets) {

dn.incBlocksScheduled();

}

// 加入pendingReplication队列

pendingReplications.increment(block, targets);

if(blockLog.isDebugEnabled()) {

blockLog.debug(

"BLOCK\* block " + block

+ " is moved from neededReplications to pendingReplications");

}

// 如果副本数目足够，则从neededReplication队列中删除

if(numEffectiveReplicas + targets.length >= requiredReplication) {

neededReplications.remove(block, priority); // remove from neededReplications

neededReplications.decrementReplicationIndex(priority);

}

computeInvalidateWork()

选择出需要进行删除操作的副本，并对datanode节点发出删除指令。

这里就是调用了invalidateBlocks.invalidateWork()方法，之后级联调用invalidateWork()，我们只需看下invalidateWork()的实现就好。这个方法很简单，直接从toInvalidate队列当中取出limit个元素，然后发出删除命令。

private synchronized List<Block> invalidateWork(

final String storageId, final DatanodeDescriptor dn) {

final LightWeightHashSet<Block> set = node2blocks.get(storageId);

if (set == null) {

return null;

}

// 每次心跳所能发送的指令是有限制的

final int limit = datanodeManager.blockInvalidateLimit;

final List<Block> toInvalidate = set.pollN(limit);

// 如果当前节点上面没有需要删除的副本，则删除数据结构当中节点的入口

if (set.isEmpty()) {

remove(storageId);

}

// 在datanode当中保存需要删除的副本，在下一次心跳时将删除命令发出

dn.addBlocksToBeInvalidated(toInvalidate);

numBlocks -= toInvalidate.size();

return toInvalidate;

}

processPendingReplications()

processPendingReplications()方法将已经超时的副本重新加入needReplication队列。

这个方法首先调用pendingReplications数据结构取出已经超时的备份请求，然后对每个备份请求判断该请求所在的副本是否还需要进行备份，如果需要备份，则加入neededReplication队列。

private void processPendingReplications() {

Block[] timedOutItems = pendingReplications.getTimedOutBlocks();

if (timedOutItems != null) {

namesystem.writeLock();

try {

for (int i = 0; i < timedOutItems.length; i++) {

NumberReplicas num = countNodes(timedOutItems[i]);

if (isNeededReplication(timedOutItems[i], getReplication(timedOutItems[i]),

num.liveReplicas())) {

neededReplications.add(timedOutItems[i],

num.liveReplicas(),

num.decommissionedReplicas(),

getReplication(timedOutItems[i]));

}

}

} finally {

namesystem.writeUnlock();

}

}

}

1.2.2.3 BlockReport

DataNode会通过心跳携带Blockreport给blockManager，blockmanager根据report当中携带的block状态更新对应的数据结构。

1) processReport入口

processReport接受来自datanode等的汇报，更新machine->blocklist以及blocklist->machine的对应关系。

如果第一次块汇报，则调用processFirstBlockReport方法，否则调用processreport方法。

if (node.numBlocks() == 0) {

// The first block report can be processed a lot more efficiently than

// ordinary block reports. This shortens restart times.

processFirstBlockReport(node, newReport);

} else {

processReport(node, newReport);

}

当出现一次块汇报时，需要判断这个节点是否为stale状态，如果是则需要对postphoneMisreplicationBlock进行重新扫描，删除哪些已经不是stale状态的block

boolean staleBefore = node.areBlockContentsStale();

node.receivedBlockReport();

if (staleBefore && !node.areBlockContentsStale()) {

rescanPostponedMisreplicatedBlocks();

}

2）第一次块汇报

仅当当前块汇报是datanode注册后的第一次块汇报时，我们使用processFirstBlockReport()方法来处理。这里将所有有效的副本加入datanodeDescriptor中（唯一进行的操作），但并不计算一个需要进行删除(toRemove，datanode上不存在，namenode上存在)的列。这里同时忽略了所有无效副本（datanode上存在，namenode的blocksmap当中不存在），将无效副本的处理推迟到了下次操作。

processFirstBlockReport()

我们一段段来看代码，首先获取当前汇报的副本状态（ReplicaState）

Block iblk = itBR.next();

ReplicaState reportedState = itBR.getCurrentReplicaState();

然后对shouldPostponeBlocksFromFuture进行判断，如果当前指令是在standby节点执行，那么可能出现blockReport出现在standby节点接受对应的editlog前，这种情况下standby节点内存中的block信息不是最新的，所以需要将操作延后。

if (shouldPostponeBlocksFromFuture &&

namesystem.isGenStampInFuture(iblk)) {

queueReportedBlock(node, iblk, reportedState,

QUEUE\_REASON\_FUTURE\_GENSTAMP);

continue;

}

如果一个副本是损坏的，那么标示这个损坏的副本（将副本加入corruptReplicas方法），并继续处理。

BlockInfo storedBlock = blocksMap.getStoredBlock(iblk);

if (storedBlock == null) continue;

BlockUCState ucState = storedBlock.getBlockUCState();

BlockToMarkCorrupt c = checkReplicaCorrupt(

iblk, reportedState, storedBlock, ucState, node);

if (c != null) {

if (shouldPostponeBlocksFromFuture) {

// 这里还是要考虑standby情况

queueReportedBlock(node, iblk, reportedState,

QUEUE\_REASON\_CORRUPT\_STATE);

} else {

markBlockAsCorrupt(c, node);

}

continue;

}

如果这个数据块处在正在构建状态，那么将这个副本加入对应的构建中副本队列

if (isBlockUnderConstruction(storedBlock, ucState, reportedState)) {

((BlockInfoUnderConstruction)storedBlock).addReplicaIfNotPresent(

node, iblk, reportedState);

//and fall through to next clause

}

如果副本状态正常，那么在blockmanager的内存中加入这个副本

if (reportedState == ReplicaState.FINALIZED) {

addStoredBlockImmediate(storedBlock, node);

}

注意，这里调用的addStoredBlockImmediate()而不是addStoredBlock()

addStoredBlockImmediate()

addStoredBlockImmediate是addStoredBlock()的快速版本。如果当前namenode是startup safemode时调用。由于是快速版本，所以不用考虑underReplication/overReplication/pendingReplications/corruptReplicas等，也不用记录日志，因为刚启动的namenode有数以万计的block需要处理，快速的将他们加入blockManager的内存是最重要的。

3) 普通块汇报

如果不是第一次块汇报，那么将会调用processReport()方法处理，我们这一节就看下这个方法的实现。

private void processReport(final DatanodeDescriptor node,

final BlockListAsLongs report) throws IOException {

//根据原来的node信息，获取需要更新的不同的队列

Collection<BlockInfo> toAdd = new LinkedList<BlockInfo>();

Collection<Block> toRemove = new LinkedList<Block>();

Collection<Block> toInvalidate = new LinkedList<Block>();

Collection<BlockToMarkCorrupt> toCorrupt = new LinkedList<BlockToMarkCorrupt>();

Collection<StatefulBlockInfo> toUC = new LinkedList<StatefulBlockInfo>();

reportDiff(node, report, toAdd, toRemove, toInvalidate, toCorrupt, toUC);

// 处理不同的队列

for (StatefulBlockInfo b : toUC) {

addStoredBlockUnderConstruction(b.storedBlock, node, b.reportedState);

}

for (Block b : toRemove) {

removeStoredBlock(b, node);

}

for (BlockInfo b : toAdd) {

addStoredBlock(b, node, null, true);

}

for (Block b : toInvalidate) {

blockLog.info("BLOCK\* processReport: "

+ b + " on " + node + " size " + b.getNumBytes()

+ " does not belong to any file");

addToInvalidates(b, node);

}

for (BlockToMarkCorrupt b : toCorrupt) {

markBlockAsCorrupt(b, node);

}

}

这里我们看到，主要通过调用reportDiff()方法，同原有datanode的信息比较，获取对应的队列。可以看到这里有五个队列：

toAdd – 上报的副本是有效的，并且同namenode当中记录的block有相同的时间戳以及长度，那么就加入toAdd队列，加入namenode内存当中

toRemove – DatanodeDescritor中存在，但是块汇报时并没有上报。从DatanodeDescriptor中删除

toInvalidate – BLockManager的BlocksMap中没有对应的block。从datanode上删除

toCorrupt – 副本的gs或者长度不正常。添加到corruptReplicas队列中。这会触发副本的备份操作，同时当副本备份数量足够的时候，则将损坏的副本删除。

toUC – 如果副本对应的block正在构建当中，那么将这个副本添加到blockinfounderconstrution的replicas队列当中。

reportDiff()

下面我们看下reportDiff()方法的实现。reportDiff()首先添加一个分隔节点delimiter到datanodeDescriptor的blockList的头部（datanode -> blockinfo，通过blockInfo中的triplets[]）。然后遍历report当中所有上报的节点，调用 processReportedBlock() 方法计算toAdd, toInvalidate, toCorrupt, toUC队列，将处理完的节点移动到blockList的头。这样delimiter就分隔了上报的节点以及没有上报的节点。最后将没有上报的节点加入toRemove队列当中。

// 这里添加一个分隔Block到list的第一位，将已经上报的block同没有上报的block分割开来。

BlockInfo delimiter = new BlockInfo(new Block(), 1);

boolean added = dn.addBlock(delimiter);

assert added : "Delimiting block cannot be present in the node";

int headIndex = 0; //currently the delimiter is in the head of the list

int curIndex;

if (newReport == null)

newReport = new BlockListAsLongs();

// 遍历处理blockReport当中上报的所有block

BlockReportIterator itBR = newReport.getBlockReportIterator();

while(itBR.hasNext()) {

Block iblk = itBR.next();

ReplicaState iState = itBR.getCurrentReplicaState();

BlockInfo storedBlock = processReportedBlock(dn, iblk, iState,

toAdd, toInvalidate, toCorrupt, toUC);

// 将block放置在datanode的头结点，也是为了将上报的block同没有上报的block分隔开来

if (storedBlock != null && (curIndex = storedBlock.findDatanode(dn)) >= 0) {

headIndex = dn.moveBlockToHead(storedBlock, curIndex, headIndex);

}

}

// 将所有dn中没有上报的节点加入toMove列表

Iterator<? extends Block> it = new DatanodeDescriptor.BlockIterator(

delimiter.getNext(0), dn);

while(it.hasNext())

toRemove.add(it.next());

dn.removeBlock(delimiter);

这里调用了方法processReportedBlock()处理toAdd, toInvalidate, toCorrupt, toUC这几个队列。我们分段看下processReportedBlock()的实现：

如果上报的副本在BlocksMap当中不存在，那么则加入toInvalidate队列中

BlockInfo storedBlock = blocksMap.getStoredBlock(block);

if(storedBlock == null) {

toInvalidate.add(new Block(block));

return null;

}

如果上报副本的GS以及文件长度同BlockManager中记录的不同，则加入toCorrupt队列中

BlockToMarkCorrupt c = checkReplicaCorrupt(

block, reportedState, storedBlock, ucState, dn);

if (c != null) {

if (shouldPostponeBlocksFromFuture) {

// 当前namenode处于standby情况时，推迟处理

queueReportedBlock(dn, storedBlock, reportedState,

QUEUE\_REASON\_CORRUPT\_STATE);

} else {

toCorrupt.add(c);

}

return storedBlock;

}

如果上报副本处于构建状态，则加入toUC队列

if (isBlockUnderConstruction(storedBlock, ucState, reportedState)) {

toUC.add(new StatefulBlockInfo(

(BlockInfoUnderConstruction)storedBlock, reportedState));

return storedBlock;

}

如果上报的副本有效，并且在blockManager的blocksMap当中并没有记录这个副本，则加入toAdd队列

if (reportedState == ReplicaState.FINALIZED

&& storedBlock.findDatanode(dn) < 0) {

toAdd.add(storedBlock);

}

addStoredBlockUnderConstruction()

这个方法非常简单，构造一个ReplicatUnderConstrution，然后添加到blockInfoUnderConstrution的replica队列当中

block.addReplicaIfNotPresent(node, block, reportedState);

if (reportedState == ReplicaState.FINALIZED && block.findDatanode(node) < 0) {

addStoredBlock(block, node, null, true);

}

addStoredBlock()

addStoredBlock不仅仅完成了将副本添加到内存的操作。同时处理副本超出、副本不足、无效副本，以及构建副本提交等功能

代码比较长，我们一段段分析

首先将block添加到内存中，注意：node.addBlock同时修改了block->datanode以及datanode->block的两条映射，通过修改blockInfo的triplets，这个非常的巧妙

// add block to the datanode

boolean added = node.addBlock(storedBlock);

如果Block状态为COMMITTED，则将内存中的Inode以及blocksMap当中保存的blockInfoUnderConstrution转换为blockInfo。如果不是COMMITTED状态，则跳出继续执行

if(storedBlock.getBlockUCState() == BlockUCState.COMMITTED &&

numLiveReplicas >= minReplication) {

storedBlock = completeBlock((MutableBlockCollection)bc, storedBlock, false);

} else if (storedBlock.isComplete()) {

namesystem.incrementSafeBlockCount(numCurrentReplica);

}

if (bc instanceof MutableBlockCollection) {

return storedBlock;

}

如果当前副本数超出了期望副本数或者低于期望，则更新对应的队列

short fileReplication = bc.getBlockReplication();

if (!isNeededReplication(storedBlock, fileReplication, numCurrentReplica)) {

neededReplications.remove(storedBlock, numCurrentReplica,

num.decommissionedReplicas(), fileReplication);

} else {

updateNeededReplications(storedBlock, curReplicaDelta, 0);

}

if (numCurrentReplica > fileReplication) {

processOverReplicatedBlock(storedBlock, fileReplication, node, delNodeHint);

}

如果损坏副本对应的block的副本数已经足够，则将损坏的副本加入invalidate队列，删除该损坏副本。

int corruptReplicasCount = corruptReplica8s.numCorruptReplicas(storedBlock);

int numCorruptNodes = num.corruptReplicas();

if (numCorruptNodes != corruptReplicasCount) {

LOG.warn("Inconsistent number of corrupt replicas for " +

storedBlock + "blockMap has " + numCorruptNodes +

" but corrupt replicas map has " + corruptReplicasCount);

}

if ((corruptReplicasCount > 0) && (numLiveReplicas >= fileReplication))

invalidateCorruptReplicas(storedBlock);

addToInvalidates

这个方法非常的简单，直接加入invalidateBlocks队列

void addToInvalidates(final Block block, final DatanodeInfo datanode) {

invalidateBlocks.add(block, datanode, true);

}

markBlockAsCorrupt

将副本加入corruptReplicas队列中。同时block的副本数如果足够时，则删除dn上的这个副本。如果不够时，则更新需要备份neededReplications队列

private void markBlockAsCorrupt(BlockToMarkCorrupt b,

DatanodeInfo dn) throws IOException {

// ...

node.addBlock(b.stored);

corruptReplicas.addToCorruptReplicasMap(b.corrupted, node, b.reason);

if (countNodes(b.stored).liveReplicas() >= bc.getBlockReplication()) {

invalidateBlock(b, node);

} else if (namesystem.isPopulatingReplQueues()) {

updateNeededReplications(b.stored, -1, 0);

}

}

1.2.2.3 其他方法

NameNodeRpcServer直接调用BlockManager.processReport，

public DatanodeCommand blockReport(DatanodeRegistration nodeReg,

String poolId, StorageBlockReport[] reports) throws IOException {

// ...

namesystem.getBlockManager().processReport(nodeReg, poolId, blist);

// ...

return null;

}

lockReceivedAndDeleted

blockReceivedAndDeleted方法，datanode向namenode汇报近期新添加以及删除的节点

public void processIncrementalBlockReport(final DatanodeID nodeID,

final String poolId, final ReceivedDeletedBlockInfo blockInfos[])

throws IOException {

writeLock();

try {

blockManager.processIncrementalBlockReport(nodeID, poolId, blockInfos);

} finally {

writeUnlock();

}

}

fsNameSystem最用调用了BlockManager的processIncrementalBlockReport()方法。这个方法判断汇报节点的的状态，如果是删除的节点 – 则直接调用removeStoredBlock()，添加则调用addBLock()，如果是正在接受，则调用processAndHandleReportedBlock()方法，注意这里副本的状态是RBW。

for (ReceivedDeletedBlockInfo rdbi : blockInfos) {

switch (rdbi.getStatus()) {

case DELETED\_BLOCK:

removeStoredBlock(rdbi.getBlock(), node);

deleted++;

break;

case RECEIVED\_BLOCK:

addBlock(node, rdbi.getBlock(), rdbi.getDelHints());

received++;

break;

case RECEIVING\_BLOCK:

receiving++;

processAndHandleReportedBlock(node, rdbi.getBlock(),

ReplicaState.RBW, null);

break;

default:

String msg =

"Unknown block status code reported by " + nodeID +

": " + rdbi;

blockLog.warn(msg);

assert false : msg; // if assertions are enabled, throw.

break;

}

removeStoredBlock()内存中将对应的副本删除

public void removeStoredBlock(Block block, DatanodeDescriptor node) {

if (!blocksMap.removeNode(block, node)) {

return;

}

// 这里需要判断下是否有必要备份新的副本

BlockCollection bc = blocksMap.getBlockCollection(block);

if (bc != null) {

namesystem.decrementSafeBlockCount(block);

updateNeededReplications(block, -1, 0);

}

// 我们删除了一个数据块，那么这个数据块一定不会在超额副本的队列中

LightWeightLinkedSet<Block> excessBlocks = excessReplicateMap.get(node

.getStorageID());

if (excessBlocks != null) {

if (excessBlocks.remove(block)) {

excessBlocksCount.decrementAndGet();

if(blockLog.isDebugEnabled()) {

blockLog.debug("BLOCK\* removeStoredBlock: "

+ block + " is removed from excessBlocks");

}

if (excessBlocks.size() == 0) {

excessReplicateMap.remove(node.getStorageID());

}

}

}

// 从损坏副本集合当中删除

corruptReplicas.removeFromCorruptReplicasMap(block, node);

}

}

addNode()是专门用datanode接受了一个新节点时的情况，这里更新了datanode的指令，同时从pendingReplications队列当中取出block（也就是说复制请求完成了）。最后调用processAndHandleReportedBlock()方法更新namenode数据结构

void addBlock(DatanodeDescriptor node, Block block, String delHint)

throws IOException {

// 更改对于datanode的指令

node.decBlocksScheduled();

// 这里获取一个delHintNode，

DatanodeDescriptor delHintNode = null;

if (delHint != null && delHint.length() != 0) {

delHintNode = datanodeManager.getDatanode(delHint);

if (delHintNode == null) {

blockLog.warn("BLOCK\* blockReceived: " + block

+ " is expected to be removed from an unrecorded node " + delHint);

}

}

//

// Modify the blocks->datanode map and node's map.

//

pendingReplications.decrement(block, node);

processAndHandleReportedBlock(node, block, ReplicaState.FINALIZED,

delHintNode);

}

reportBadBlocks

这个方法用于datanode向namenode汇报损坏的数据块.最终调用了blockManager.findAndMarkBlockAsCorrupt()，这个方法上一节当中我们已经讲过了，不在赘述。

1.2.6 Block状态转移

当blockmanager当中一个副本可能存在多种状态，并且这些状态之间是可以转移的。但HDFS代码当中并没有使用一个枚举给出数据块副本的状态以及状态之间的转移。而是通过集合或者不同的类BlockUnderConstruction&Block，以及副本所存在的datanode的状态。

下面的5种状态是通过BlockManager当中对应的数据结构来标识的。

正常副本 ： 存在BlocksMap当中的正常的数据块副本

损坏的副本 ： 存在corruptReplicas队列中的损坏的数据块。损坏的副本一般由datanode的数据块扫描器，数据块接收器或者客户端的数据块校验过程发现（引用1）。损坏副本状态都会变为等待删除副本状态。

多余的副本 ： 存放在excessReplicateMap当中的数据块，当前数据块的副本数目多余设置的数目时。这种情况可能发生在更改数据块副本数量之后。多余副本状态会变为等待删除副本状态。

等待删除的副本 ： 存放在invalidateBlocks当中，需要由BlockManager命令对应的datanode删除的副本

等待复制副本 ： 存放在neededReplications当中。还没有生成复制命令。等待复制副本会转移到正在复制副本状态，当发出复制命令之后。

正在复制副本 ： 已生成复制命令，但并不知道复制动作是否成功。如果复制动作成功，那么正在复制副本就会转移至正常副本状态。如果复制失败，则转至等待复制副本状态。

推迟操作的副本 ： 副本数过多，但是存在stale副本，所以推迟处理，等待stale的datanode进行块汇报。

在HDFS append逻辑时，会出现 正在构建状态 副本， 以及 恢复状态 副本。这些信息保存在租约管理器当中。

正在构建状态副本 ： 当前数据块正在构建当中

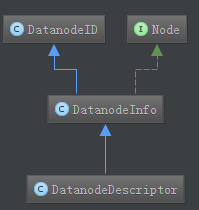
恢复状态副本 ： 参考 （http://yanbohappy.sinaapp.com/?p=175）

当datanode被撤销时（当前datanode离线，或者放入exclude文件中时），会存 撤销节点副本 ，这些副本由于节点下架而进入“最终状态”，或者由于节点重新进入工作状态而返回“正常副本”状态。

1.3 DatanodeDescriptor

介绍Namenode内存当中对于datanode的管理，包括三个部分，在namenode当中使用DatanodeDescriptor类来描述一个Datanode节点。使用HeartbeatManager类管理datanode同namenode之间的心跳。最后使用DatanodeManager提供对于datanode管理的接口。

DatanodeDescriptor扩展自Datanodeinfo类，用于在namenode当中描述一个Datanode信息：



1.3.1 DatanodeId

DatanodeID用于唯一标识Datanode，Datanode通过<ip,port>以及storageID进行标识的，如下代码所示：

private String ipAddr; // IP address

private String hostName; // hostname claimed by datanode

private String peerHostName; // hostname from the actual connection

private String storageID; // unique per cluster storageID

private int xferPort; // data streaming port

private int infoPort; // info server port

private int infoSecurePort; // info server port

private int ipcPort; // IPC server port

1.3.2 DatanodeInfo

扩展自DatanodeID，包含了一些比较简单的状态，如下代码所示：

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。

protected AdminStates adminState;

public enum AdminStates {

NORMAL("In Service"),

DECOMMISSION\_INPROGRESS("Decommission In Progress"),

DECOMMISSIONED("Decommissioned");

final String value;

// ...

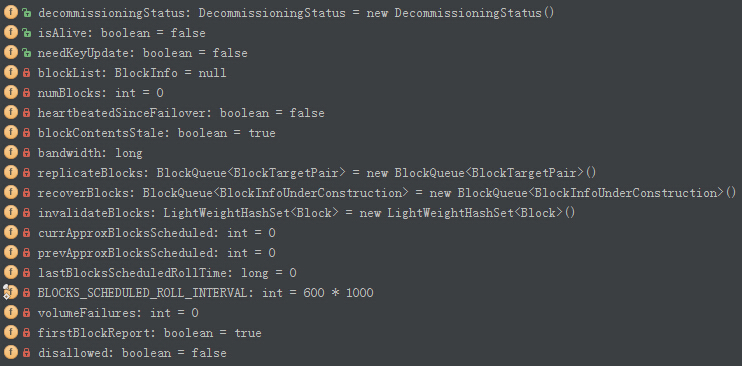
}

1.3.3 DatanodeDescriptor

DatanodeDescriptor是Namenode当中用于描述datanode信息的类，这个类在namenode server侧，而对于client是不可见的。

1) 字段

下图是比较重要的几个字段：



BlockInfo blockList : 用以描述当前这个datanode所有的数据块。回忆BlockInfo类的triplets[]数组当中记录当前datanode包含的前一个数据块以及后一个数据块，这里就利用了这个数据结构，非常巧妙。同时需要注意的是，BlockManager当中将一个副本添加到内存中时，就是通过addBlock()在blockList这个数据结构添加的副本。

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 ： 当namenode出现失败或者正在启动时，datanode可能会挂起上一次namenode发起的删除操作。那么这个时候我们就认为当前datanode为stale状态，直到收到了这个datanode的块汇报。当一个datanode处于stale状态时，那么这个datanode上的所有副本都为stale的。如果一个数据块有至少一个stale副本，那么这个数据块就不可以被删除。（引用1 HDFS-1972）。

考虑下面的情况，当nn发生失败，active和standby切换时。上一次进行invalidate操作的datanode还没有进行块汇报（这里datanode可能完成了删除操作，也有可能没有完成）。那么在这种情况下，原来standby的namenode当中的block信息就很有可能不完整，如果这时候再次对这个数据块发起删除操作，很有可能就丢了这个数据块（例如副本为2时，dn1已经删除了一个副本，但是没有汇报。这时如果用户再次发起一个删除操作，nn并不知道副本数为1了，直接删除，那么就丢失了这个数据块）。

所以现在HDFS的逻辑是当出现active standby切换时，namenode将所有datanode heartbeatedSinceFailover以及blockContentsStale两个字段设置为false和true。然后扫描内存中的所有数据块，如果当前数据块有副本在stale的datanode当中，那么就将这个数据块放入postponedMisreplicatedBlocks队列当中，直到所有stale状态的datanode进行了块汇报。

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

public void updateHeartbeat(long capacity, long dfsUsed, long remaining,

long blockPoolUsed, int xceiverCount, int volFailures) {

setCapacity(capacity);

setRemaining(remaining);

setBlockPoolUsed(blockPoolUsed);

setDfsUsed(dfsUsed);

setXceiverCount(xceiverCount);

setLastUpdate(Time.now());

this.volumeFailures = volFailures;

this.heartbeatedSinceFailover = true;

rollBlocksScheduled(getLastUpdate());

}

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。

public void markStaleAfterFailover() {

heartbeatedSinceFailover = false;

blockContentsStale = true;

}

areBlockContentsStale() – 查询当前datanode是否为stale

public boolean areBlockContentsStale() {

return blockContentsStale;

}

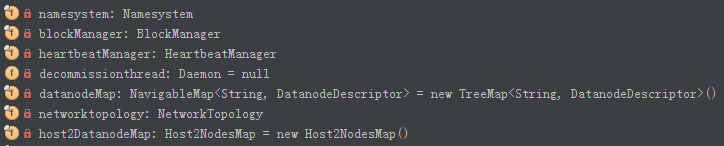
其余的方法都比较简单，这里不再赘述。大家可以自己看看。

1.4 DatanodeManager

DatanodeManager是对Datanode的管理类，重点介绍DatanodeManager维护datanode注册、下架以及心跳的操作

1.4.1 属性

DatanodeManager主要维护了namenode当中注册的datanode信息，以及这些datanode在网络中拓扑结构，如下图所示



datanodeMap维护了一个storageID -> DatanodeDescriptor的映射关系

host2DatanodeMap:维护host->DatanodeDescriptor的映射关系，这里要特别注意，为什么要维护两个映射关系呢，是因为datanode的storageId是有可能发生变化的。一般情况下，这两个映射关系是一致，但是可能会出现一个datanode重新启动，并使用一个新的storageId，这时候就需要对datanodeMap进行更新。这个在后面registerDatanode()方法当中具体说明。

networktopoloy，维护整个网络的托盘结构

decommissionthread:一个线程类周期性的调用checkDecommissionState方法设置datanodeDescriptior的decommission状态

1.4.2 datanode注册

数据节点启动时，需要和名字节点进行握手、注册和数据块上报

1) 握手

握手请求由namenodeRpcServer实现，非常简单，如下所示：

public NamespaceInfo versionRequest() throws IOException {

namesystem.checkSuperuserPrivilege();

return namesystem.getNamespaceInfo();

}

2) 注册

datanode的注册请求会由registerDatanode()方法进行响应。 DatanodeManager会为进行注册的datanode节点分配唯一的storageId作为标识（在datanodemap中作为key，获取DatanodeDescriptor）。要注意的是，数据节点可以重复发送注册信息，并且datanode的  
 storageId是有可能发生改变的（当数据节点上的数据都被擦除时）。

同时在datanode注册时，datanodeManager还需要对datanode是否可以接入进行判断（exclude文件中时）。

datanode注册情况可以基本分类为以下三种（引用1）：

该datanode没有注册过

该datanode注册过，但是这次注册使用了新的storageId，表明该数据节点存储空间已经被清理过了，原有的数据块副本被删除

该datanode注册过，这次是重复注册

在第一小节的介绍当中我们知道datanodemanager当中维护着两个映射关系。这里用nodeS表明从datanodemap中通过StorageId获得的datanodeDescriptor对象。用nodeN表明从host2DatanodeMap中通过host名获得的datanodeDescriptor对象。

if (!hostFileManager.isIncluded(nodeReg)) {

throw new DisallowedDatanodeException(nodeReg);

} // 判断是否在exclude当中

DatanodeDescriptor nodeS = datanodeMap.get(nodeReg.getStorageID());

DatanodeDescriptor nodeN = host2DatanodeMap.getDatanodeByXferAddr(

nodeReg.getIpAddr(), nodeReg.getXferPort());

那么对应于上述三种情况：

nodeS == null && nodeN == null， 第一次注册。这里产生新的storageId，更新datanodemap以及host2DatanodeMap，更新网络拓扑，检查下架等。

// 第一次注册

if ("".equals(nodeReg.getStorageID())) {

// 设置新的storageId

nodeReg.setStorageID(newStorageID());

// ...

}

DatanodeDescriptor nodeDescr

= new DatanodeDescriptor(nodeReg, NetworkTopology.DEFAULT\_RACK);

boolean success = false;

try {

// 更细网络拓扑

nodeDescr.setNetworkLocation(resolveNetworkLocation(nodeDescr));

networktopology.add(nodeDescr);

nodeDescr.setSoftwareVersion(nodeReg.getSoftwareVersion());

// 添加到datanodemap以及host2DatanodeMap

addDatanode(nodeDescr);

// 检查是否下架

checkDecommissioning(nodeDescr);

// 在heartbeatManager中添加该数据节点

heartbeatManager.addDatanode(nodeDescr);

success = true;

incrementVersionCount(nodeReg.getSoftwareVersion());

} finally {

//...

}

nodeN !=null && nodeN != nodeS ，使用新的storageId注册，也就是说原先在datanode上的数据都无效了。所以需要清理namenode数据。后续处理同第一种情况相似。

if (nodeN != null && nodeN != nodeS) {

// 删除一个DatanodeDescriptor

removeDatanode(nodeN);

// 从datanodemap以及host2DatanodeMap删除

wipeDatanode(nodeN);

nodeN = null;

}

这里调用了两个删除方法，removeDatanode(nodeN)以及wipeDatanode(nodeN)。其中removeDatanode主要删除了namenode内存中其他数据结构中的datanodeDescriptor，而wipeDatanode()则是datanode内部的两个数据结构datanodemap以及host2DatanodeMap。参考以下代码

private void wipeDatanode(final DatanodeID node) {

final String key = node.getStorageID();

synchronized (datanodeMap) {

host2DatanodeMap.remove(datanodeMap.remove(key));

}

}

private void removeDatanode(DatanodeDescriptor nodeInfo) {

assert namesystem.hasWriteLock();

heartbeatManager.removeDatanode(nodeInfo);

blockManager.removeBlocksAssociatedTo(nodeInfo);

networktopology.remove(nodeInfo);

decrementVersionCount(nodeInfo.getSoftwareVersion());

namesystem.checkSafeMode();

}

nodeS != null 除去上述情况，刷新注册

// 更新网络拓扑，节点等信息...

getNetworkTopology().remove(nodeS);

if(shouldCountVersion(nodeS)) {

decrementVersionCount(nodeS.getSoftwareVersion());

}

nodeS.updateRegInfo(nodeReg);

nodeS.setSoftwareVersion(nodeReg.getSoftwareVersion());

nodeS.setDisallowed(false);

nodeS.setNetworkLocation(resolveNetworkLocation(nodeS));

getNetworkTopology().add(nodeS);

heartbeatManager.register(nodeS);

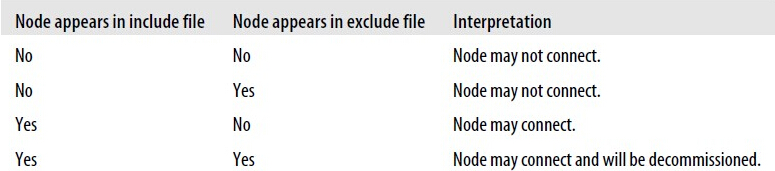
incrementVersionCount(nodeS.getSoftwareVersion());

checkDecommissioning(nodeS);

1.4.3 下架,decommision

可以动态的向HDFS集群中添加以及删除datanode节点。HDFS提供了dfs.hosts文件（又称include文件）以及dfs.hosts.exclude文件。用户可以讲想要上架以及下架的datanode节点名写入这两个文件。然后调用dfsadmin refreshNodes命令，namenode就会开始下架datanode。被下架的数据块会复制到集群的其他datanode。在这个过程中，datanode就处于“正在下架状态”，数据块复制完成后就会转移到“已下架”状态。（引用1）

节点的状态有四种



refreshNodes命令最终会调用DatanodeManager.refreshNodes()

private void refreshDatanodes() {

for(DatanodeDescriptor node : datanodeMap.values()) {

// Check if not include.

if (!hostFileManager.isIncluded(node)) {

node.setDisallowed(true); // 设置为不可以同namenode通信

} else {

if (hostFileManager.isExcluded(node)) {

startDecommission(node); // 开始下架.

} else {

stopDecommission(node); // 结束下架

}

}

}

}

我们在DatanodeDescriptor章节当中已经介绍过，dd当中有一个字段标识当前datanode的状态 – adminState。如果adminState==null，则表明当前datanode为正常状态。如果为AdminStates.DECOMMISSION\_INPROGRESS，则表明datanode为正在下架状态。如果为AdminStates.DECOMMISSIONED，则表明datanode已经下架了。这里要特别注意，对于DatanodeDescriptor这个字段的修改是通过heartBeatManager的方法修改的。

下startDecommission以及stopDecommission两个方法

private void startDecommission(DatanodeDescriptor node) {

// 判断当前节点不在decommission相关状态

if (!node.isDecommissionInProgress() && !node.isDecommissioned()) {

// 设置datanodeDescriptor adminState状态

heartbeatManager.startDecommission(node);

node.decommissioningStatus.setStartTime(now());

// 检查当前的decommission操作是否完成

checkDecommissionState(node);

}

}

void stopDecommission(DatanodeDescriptor node) {

if (node.isDecommissionInProgress() || node.isDecommissioned()) {

// 设置datanodeDescriptor adminState状态

heartbeatManager.stopDecommission(node);

if (node.isAlive) {

// 由于节点重新上架，需要对超出备份数目的节点判断。并进行删除操作

blockManager.processOverReplicatedBlocksOnReCommission(node);

}

}

}

checkDecommissionState

前面已经给出了概念，当下架节点当中的所有数据块移动和备份到其他的datanode时，该节点的状态才能从DECOMMISSION\_INPROGRESS -> DECOMMISSIONED。这里判断节点是否完成decommission操作的方法为  
checkDecommissionState，这个方法非常的重要。这个小节了解下这个方法的实现。

boolean checkDecommissionState(DatanodeDescriptor node) {

if (node.isDecommissionInProgress()) {

if (!blockManager.isReplicationInProgress(node)) {

node.setDecommissioned();

LOG.info("Decommission complete for " + node);

}

}

return node.isDecommissioned();

}

如上代码所示，其实是调用了blockManager.isReplicationInProgress()。这个方法就是判断出，当前datanode节点上的所有数据块是否有足够的备份数。如果所有数据块都得到了足够的副本数，那么就完成了下架操作。这里要特别注意最后一个判断，当节点被撤销时，有新的数据块添加到了datanode。datanode通过blockReport上报给namenode，这时由于节点已经下架，就需要将这个新添加的block加入needReplication队列进行备份操作。另一个进入备份队列的机会是，对副本计数时，decommission节点上的副本并不算live副本，这时会将下架的datanode上的副本加入needReplication队列。

boolean isReplicationInProgress(DatanodeDescriptor srcNode) {

boolean status = false;

int underReplicatedBlocks = 0;

int decommissionOnlyReplicas = 0;

int underReplicatedInOpenFiles = 0;

final Iterator<? extends Block> it = srcNode.getBlockIterator();

while(it.hasNext()) {

final Block block = it.next();

BlockCollection bc = blocksMap.getBlockCollection(block);

if (bc != null) {

NumberReplicas num = countNodes(block);

int curReplicas = num.liveReplicas();

int curExpectedReplicas = getReplication(block);

if (isNeededReplication(block, curExpectedReplicas, curReplicas)) {

if (curExpectedReplicas > curReplicas) {

if (!status) {

status = true;

logBlockReplicationInfo(block, srcNode, num);

}

underReplicatedBlocks++;

if ((curReplicas == 0) && (num.decommissionedReplicas() > 0)) {

decommissionOnlyReplicas++;

}

if (bc instanceof MutableBlockCollection) {

underReplicatedInOpenFiles++;

}

}

if (!neededReplications.contains(block) &&

pendingReplications.getNumReplicas(block) == 0) {

neededReplications.add(block,

curReplicas,

num.decommissionedReplicas(),

curExpectedReplicas);

}

}

}

}

srcNode.decommissioningStatus.set(underReplicatedBlocks,

decommissionOnlyReplicas,

underReplicatedInOpenFiles);

return status;

}

Monitor:

由于decommission的时间比较长，使用Monitro来监控decommission操作，

由下代码所知，其实还是周期性的调用了checkDecommissionState()方法。

private void check() {

final DatanodeManager dm = blockmanager.getDatanodeManager();

int count = 0;

for(Map.Entry<String, DatanodeDescriptor> entry

: dm.getDatanodeCyclicIteration(firstkey)) {

final DatanodeDescriptor d = entry.getValue();

firstkey = entry.getKey();

if (d.isDecommissionInProgress()) {

try {

dm.checkDecommissionState(d);

} catch(Exception e) {

LOG.warn("entry=" + entry, e);

}

if (++count == numNodesPerCheck) {

return;

}

}

}

}

1.4.4 心跳处理

这个方法三步操作：

1. 判断是否为死的节点和没有注册的节点  
2. 调用HeartBeatManager更新Namenode负载信息，以及datanode的负载信息。同时更新上次汇报时间  
3. 返回这次心跳需要带回的操作

synchronized (heartbeatManager) {

synchronized (datanodeMap) {

DatanodeDescriptor nodeinfo = null;

try {

nodeinfo = getDatanode(nodeReg);

} catch(UnregisteredNodeException e) {

return new DatanodeCommand[]{RegisterCommand.REGISTER};

}

// Check if this datanode should actually be shutdown instead.

if (nodeinfo != null && nodeinfo.isDisallowed()) {

setDatanodeDead(nodeinfo);

throw new DisallowedDatanodeException(nodeinfo);

}

if (nodeinfo == null || !nodeinfo.isAlive) {

return new DatanodeCommand[]{RegisterCommand.REGISTER};

}

heartbeatManager.updateHeartbeat(nodeinfo, capacity, dfsUsed,

remaining, blockPoolUsed, xceiverCount, failedVolumes);

// If we are in safemode, do not send back any recovery / replication

// requests. Don't even drain the existing queue of work.

if(namesystem.isInSafeMode()) {

return new DatanodeCommand[0];

}

//check lease recovery

BlockInfoUnderConstruction[] blocks = nodeinfo

.getLeaseRecoveryCommand(Integer.MAX\_VALUE);

if (blocks != null) {

BlockRecoveryCommand brCommand = new BlockRecoveryCommand(

blocks.length);

for (BlockInfoUnderConstruction b : blocks) {

DatanodeDescriptor[] expectedLocations = b.getExpectedLocations();

// Skip stale nodes during recovery - not heart beated for some time (30s by default).

List<DatanodeDescriptor> recoveryLocations =

new ArrayList<DatanodeDescriptor>(expectedLocations.length);

for (int i = 0; i < expectedLocations.length; i++) {

if (!expectedLocations[i].isStale(this.staleInterval)) {

recoveryLocations.add(expectedLocations[i]);

}

}

// If we only get 1 replica after eliminating stale nodes, then choose all

// replicas for recovery and let the primary data node handle failures.

if (recoveryLocations.size() > 1) {

if (recoveryLocations.size() != expectedLocations.length) {

LOG.info("Skipped stale nodes for recovery : " +

(expectedLocations.length - recoveryLocations.size()));

}

brCommand.add(new RecoveringBlock(

new ExtendedBlock(blockPoolId, b),

recoveryLocations.toArray(new DatanodeDescriptor[recoveryLocations.size()]),

b.getBlockRecoveryId()));

} else {

// If too many replicas are stale, then choose all replicas to participate

// in block recovery.

brCommand.add(new RecoveringBlock(

new ExtendedBlock(blockPoolId, b),

expectedLocations,

b.getBlockRecoveryId()));

}

}

return new DatanodeCommand[] { brCommand };

}

final List<DatanodeCommand> cmds = new ArrayList<DatanodeCommand>();

//check pending replication

List<BlockTargetPair> pendingList = nodeinfo.getReplicationCommand(

maxTransfers);

if (pendingList != null) {

cmds.add(new BlockCommand(DatanodeProtocol.DNA\_TRANSFER, blockPoolId,

pendingList));

}

//check block invalidation

Block[] blks = nodeinfo.getInvalidateBlocks(blockInvalidateLimit);

if (blks != null) {

cmds.add(new BlockCommand(DatanodeProtocol.DNA\_INVALIDATE,

blockPoolId, blks));

}

blockManager.addKeyUpdateCommand(cmds, nodeinfo);

// check for balancer bandwidth update

if (nodeinfo.getBalancerBandwidth() > 0) {

cmds.add(new BalancerBandwidthCommand(nodeinfo.getBalancerBandwidth()));

// set back to 0 to indicate that datanode has been sent the new value

nodeinfo.setBalancerBandwidth(0);

}

if (!cmds.isEmpty()) {

return cmds.toArray(new DatanodeCommand[cmds.size()]);

}

}

}

return new DatanodeCommand[0];

1.5 HeartBeatManager

在DatanodeDescriptor当中，有两个字段负责心跳以及stale，HeartbeatManager专门负责track各个datanode心跳，以及设置对应状态的类。

在心跳当中维护了整个系统的负载信息，同时负责更新整个系统的负载信息

1.5.1 字段

如下代码所示，HeartBeatManager当中有四个字段，datanodes保存了一个当前所有活着的dn，heartbeatRecheckInterval则为心跳检查间隔， stat记录了所有数据节点的基本信息，heartbeatThread为检查线程。

private final List<DatanodeDescriptor> datanodes = new ArrayList<DatanodeDescriptor>();

/\*\* Statistics, which are synchronized by the heartbeat manager lock. \*/

private final Stats stats = new Stats();

/\*\* The time period to check for expired datanodes \*/

private final long heartbeatRecheckInterval;

/\*\* Heartbeat monitor thread \*/

private final Daemon heartbeatThread = new Daemon(new Monitor());

1.5.2 heartbeatThread

run()

每隔heartbeatrecheckinterval，调用一次heartbeatCheck方法：

public void run() {

while(namesystem.isRunning()) {

try {

final long now = Time.now();

if (lastHeartbeatCheck + heartbeatRecheckInterval < now) {

heartbeatCheck();

lastHeartbeatCheck = now;

}

// ...

} catch (Exception e) {

LOG.error("Exception while checking heartbeat", e);

}

try {

Thread.sleep(5000); // 5 seconds

} catch (InterruptedException ie) {

}

}

}

heartbeatcheck()

这个方法会检查是否有dead的数据节点 – 也就是datanode在heartbeatRecheckInterval时间内没有发送心跳。同时判断当前节点是否为stale节点 – 存在一个stale超时时间，超过这个时间没有发送心跳，就判断为stale节点。如果是有dead节点，则从datanodemanager当中移除。

boolean allAlive = false;

while (!allAlive) {

DatanodeID dead = null;

int numOfStaleNodes = 0;

synchronized(this) {

// 判断心跳是否已经超时

for (DatanodeDescriptor d : datanodes) {

if (dead == null && dm.isDatanodeDead(d)) {

stats.incrExpiredHeartbeats();

dead = d;

}

// 判断当前节点是否为stale，这里也是判断超时，用stale超时时间

if (d.isStale(dm.getStaleInterval())) {

numOfStaleNodes++;

}

}

// 设置stale dn的数目

dm.setNumStaleNodes(numOfStaleNodes);

}

allAlive = dead == null;

if (!allAlive) {

// 删除死亡的dn

namesystem.writeLock();

try {

if (namesystem.isInStartupSafeMode()) {

return;

}

synchronized(this) {

dm.removeDeadDatanode(dead);

}

} finally {

namesystem.writeUnlock();

}

}

}

二、HDFS 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。

2.1 protobuf

Client同Namenode之间的接口ClientProtol是最常用的接口，Hadoop底层序列化使用的是protobuf，并且Hadoop自己实现了基于protobuf的rpc。Protobuf为了解决XML(序列化)的性能缺点。例如模块A发送订单信息给B，通讯方式使用socket，假设订单包括如下属性：  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－  
　　时间：time（用整数表示）  
　　客户id：userid（用整数表示）  
　　交易金额：price（用浮点数表示）  
　　交易的描述：desc（用字符串表示）  
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

使用protobuf实现，首先写一个proto文件，添加一个名为order的message接口，用来描述通讯协议中的结构化数据，文件内容大致如下：

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

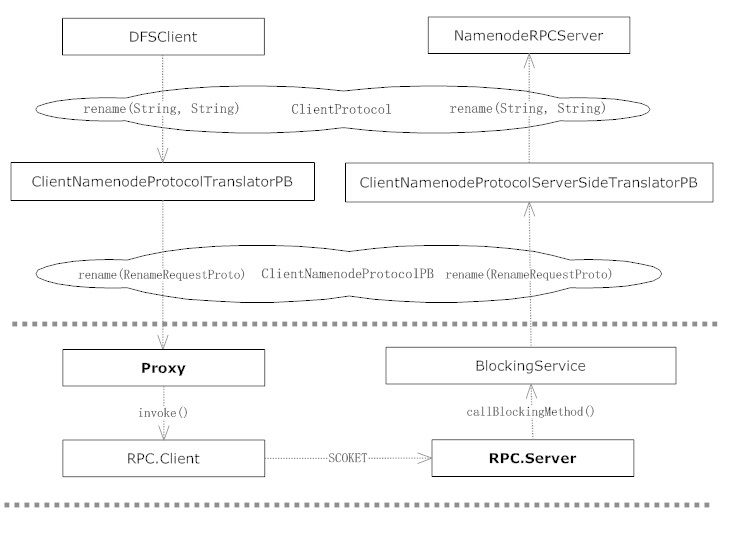
message Order{ required int32 time = 1;  
  required int32 userid = 2;  
  required float price = 3;  
  optional string desc = 4;  
}

－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

使用protobuf内置的编译器编译该proto，让它生成c++语言的订单包装类，调用某种socket的通讯库把序列化之后的字符串发送出去。接收方通过网络通讯库接收数据，存放到某字符串。ProtocolBuffer是Google的一种数据交换的格式，它独立于语言，独立于平台。

ClientProtocal使用适配器设计模式，HDFS构造了类ClientNameNodeProtocalTrans

latorPB作为Client侧的适配器类，将给予Probobuf的接口转换为HDFS的接口，同理使用类ClientNamenodeprotocoalServerSideTranslatorPB作为Server侧的接口

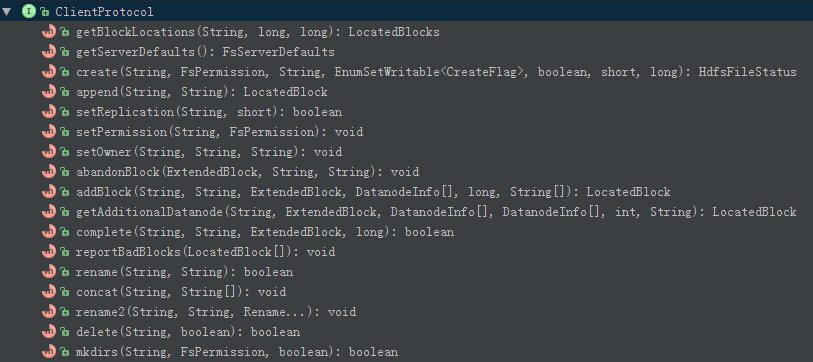


2.1.1 ClientProtocal

主要包括DFSClient同NameNode之间交互的所有接口方法（例如，rename,mkdir etc），这些方法是两者之间交互的RPC协议，这个类将DFSClient和Namenode同RPC的序列化隔绝了起来。

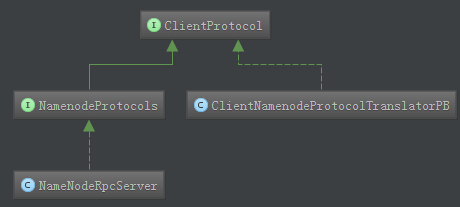
1) 接口

ClientProtocal中包含很多方法，例如rename,delete,mkdir等常用的DFSClient与Namenode交互的方法，如下图所示：



2) 实现

通过下图，ClientProtocol主要有两个重要的实现：



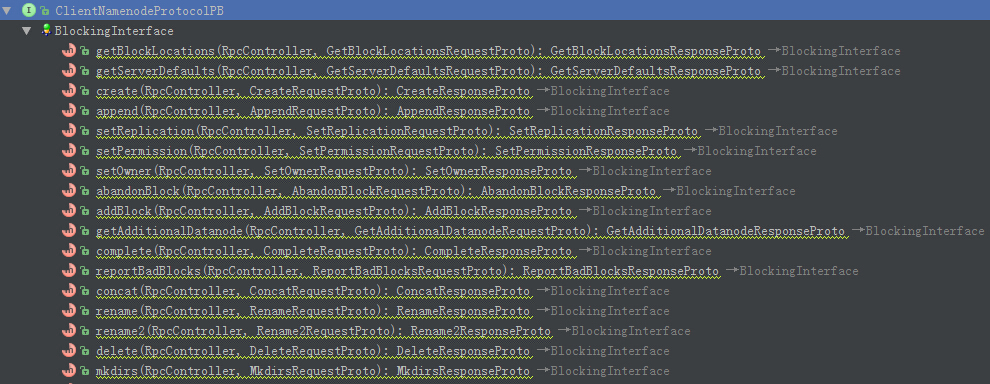
ClientNamenodeProtolTranslatorPB，这个类是整个RPC请求Client侧最重要的类，将Client端的请求参数封装成PB形式，然后通过代理类(实现ClientNamenodeProtocolPB接口)发送出去。

NamenodeRpcServer，真正的namenode侧对clientProtocal中调用进行响应的实现，例如修改Namenode目录树、块内容等。

2.1.1.1 ClientNamenodeProtocolPB

Client接口是Namenode同DFSClient之间的接口，但是真正在RPC调用过程中，接口方法的参数都是需要进行序列化（hdfs使用了protobuf作为序列化工具），需要对参数进行包装。

ClientNamenodeProtocolPB接口，将ClientProtocol方法当中的参数使用Protobuf进行了包装，使得这些参数可以通过网络传输，通过protobuf进行序列化/解序列化。



所有的方法都有两个RpcController，其实都是MethodNameRequestProto这种命名方式的参数，这个类包装了对应方法的参数，例如clientprotocal中的rename(src & dst)，在这里就是用RenameRequestProtol封装可以protobuf序列化的类，同时返回一个renameResponseProto封装返回结果。

那么这些类的申明都是在哪里呢 – ClientNamenodeProtocol.proto文件。这个文件是ClientNamenodeProtocol的protobuf的定义文件。然后会调用protoc.exe产生对应的java类。这个文件主要分为两个部分 ：

调用过程中的请求参数 和 响应参数的申明 – 也就是上面描述的RenameRequestProto以及RenameResponseProto的申明（注意这两个类是写好申明之后，执行protoc.exe动态生成的）

message RenameRequestProto {

required string src = 1;

required string dst = 2;

}

message RenameResponseProto {

required bool result = 1;

}

对应于protoBuf的接口定义 – 特别注意ProtoBuf是只有序列化的实现，并没有RPC的实现， 所以这里只是给出的接口 – 真正的实现是由Hadoop自己做的（第四讲当中会提到的RPC、Server、Client类）

service ClientNamenodeProtocol {

rpc getBlockLocations(GetBlockLocationsRequestProto)

returns(GetBlockLocationsResponseProto);

rpc getServerDefaults(GetServerDefaultsRequestProto)

returns(GetServerDefaultsResponseProto);

rpc create(CreateRequestProto)returns(CreateResponseProto);

rpc append(AppendRequestProto) returns(AppendResponseProto);

rpc setReplication(SetReplicationRequestProto)

returns(SetReplicationResponseProto);

rpc setPermission(SetPermissionRequestProto)

returns(SetPermissionResponseProto);

// ...

}

写好这个文件之后，就可以通过命令编译生成类 – ClientNamenodeProtocolProtos。要使用这个类作为接口有两种方式：

由于这个类实现了Service的接口 ，同时所有ClientProtocol的方法，在这个类当中都有abstract的方法，那么实现类只需要extends ClientNamenodeProtocol即可，之后就可以调用Servicee接口中的方法

要注意ClientNamenodeProtocol的newReflectiveserver(Interface), 以及newReflectiveBlockingService(BlockingInterface)， 这里传入的BlockingInterface以及Interface都是ClientNamenodeProtocol当中定义的同接口。也就是使用时只需要传入一个实现了ClientNamenodeProtocol.BlockingInterface的引用即可(正好联想到ClientNameNodeServerSideTranslatorPB)，然后调用callBlockingMethod()方法即可串联整个类。

public static com.google.protobuf.BlockingService

newReflectiveBlockingService(final BlockingInterface impl) {

return new com.google.protobuf.BlockingService() {

public final com.google.protobuf.Descriptors.ServiceDescriptor

getDescriptorForType() {

return getDescriptor();

}

public final com.google.protobuf.Message callBlockingMethod(

com.google.protobuf.Descriptors.MethodDescriptor method,

com.google.protobuf.RpcController controller,

com.google.protobuf.Message request)

throws com.google.protobuf.ServiceException {

if (method.getService() != getDescriptor()) {

throw new java.lang.IllegalArgumentException(

"Service.callBlockingMethod() given method descriptor for " +

"wrong service type.");

}

switch(method.getIndex()) {

case 0:

return impl.getBlockLocations(controller, (org.apache.hadoop.hdfs.protocol.proto.ClientNamenodeProtocolProtos.GetBlockLocationsRequestProto)request);

case 1:

return impl.getServerDefaults(controller, (org.apache.hadoop.hdfs.protocol.proto.ClientNamenodeProtocolProtos.GetServerDefaultsRequestProto)request);

case 2:

return impl.create(controller, (org.apache.hadoop.hdfs.protocol.proto.ClientNamenodeProtocolProtos.CreateRequestProto)request);

case 3:

//....

}

}

}

2.1.1.2 ClientNamenodeProtocolTranslatorPB

在DFSClient当中持有这样的一个引用，类似于设计模式当中的适配器模式，将ClientProtocol的请求转化为ClientNameProtocolPB这个接口的请求。

public class ClientNamenodeProtocolTranslatorPB implements

ProtocolMetaInterface, ClientProtocol, Closeable, ProtocolTranslator {

// 真正底层调用类

final private ClientNamenodeProtocolPB rpcProxy;

// ...

}

以rename作为例子，主要就是两部操作，构建protobuf的参数，然后调用Impl的指定方法，返回结果，类似于桥梁的作用：

public boolean rename(String src, String dst) throws UnresolvedLinkException,

IOException {

// 构建pb参数

RenameRequestProto req = RenameRequestProto.newBuilder()

.setSrc(src)

.setDst(dst).build();

try {

// 调用底层impl对应方法，返回结果

return rpcProxy.rename(null, req).getResult();

} catch (ServiceException e) {

throw ProtobufHelper.getRemoteException(e);

}

}

2.1.1.3 ClientNamenodeprotocolServersideTranslatorPB

ClientNamenodeProtocolServerSideTranslatorPB就是server侧的适配器。Client侧的适配器是从ClientProtocol的请求转化为ClientNamenodeProtocolPB，那么可以知道Server侧的适配器一定是ClientNamenodeProtocolPB转化为ClientProtocol。

public class ClientNamenodeProtocolServerSideTranslatorPB implements

ClientNamenodeProtocolPB {

// 底层实现类

final private ClientProtocol server;

// ...

}

方法的逆向：

public RenameResponseProto rename(RpcController controller,

RenameRequestProto req) throws ServiceException {

try {

boolean result = server.rename(req.getSrc(), req.getDst());

return RenameResponseProto.newBuilder().setResult(result).build();

} catch (IOException e) {

throw new ServiceException(e);

}

}

2.2 Client获取Proxy

下面介绍client侧RPC调用的流程，下面是DFSClient的rename方法：

public void rename(String src, String dst, Options.Rename... options)

throws IOException {

checkOpen();

try {

namenode.rename2(src, dst, options);

} catch(RemoteException re) {

// ...

}

}

调用了rename2()的方法，这里的namenode其实就是一个clientprotocol的引用。通过proxyInfo类来获取这个引用。这里的proxyInfo是通过NamenodeProxies工厂类产生的。

inal ClientProtocol namenode;

// 构造方法

this.namenode = proxyInfo.getProxy();

// 获取proxyInfo引用

proxyInfo = NameNodeProxies.createProxy(conf, nameNodeUri,ClientProtocol.class);

namenode这里的应用是一个代理类，将clientProtocal的方法调用前转到RPC.client上，通过Socket将请求远程发送给RPC.server。

2.2.1 NameNodeProxies

在Hadoop2.x当中引入了NameNode的HA机制，但是对于Client来说是透明的，也就是Client并不知道请求是发送到了哪个Namenode。所以这里将Proxy的创建进行了封装，NameNodeProxie就是这个作用，会根据配置文件创建 HA的clientProxy 或者 非HA的clientProxy。参考下段代码

public static <T> ProxyAndInfo<T> createProxy(Configuration conf,

URI nameNodeUri, Class<T> xface) throws IOException {

Class<FailoverProxyProvider<T>> failoverProxyProviderClass =

getFailoverProxyProviderClass(conf, nameNodeUri, xface);

if (failoverProxyProviderClass == null) {

// 非HA情况

return createNonHAProxy(conf, NameNode.getAddress(nameNodeUri), xface,

UserGroupInformation.getCurrentUser(), true);

} else {

// HA情况

FailoverProxyProvider<T> failoverProxyProvider = NameNodeProxies

.createFailoverProxyProvider(conf, failoverProxyProviderClass, xface,

nameNodeUri);

Conf config = new Conf(conf);

T proxy = (T) RetryProxy.create(xface, failoverProxyProvider, RetryPolicies

.failoverOnNetworkException(RetryPolicies.TRY\_ONCE\_THEN\_FAIL,

config.maxFailoverAttempts, config.failoverSleepBaseMillis,

config.failoverSleepMaxMillis));

Text dtService = HAUtil.buildTokenServiceForLogicalUri(nameNodeUri);

return new ProxyAndInfo<T>(proxy, dtService);

}

}

下面我们就分析HA以及非HA的proxy构造流程

HA模式

这里看到HA模式是调用了RetryProxy.creat()方法。这里的RetryProxy是一个工厂类。我们直接看工厂方法

public static <T> Object create(Class<T> iface,

FailoverProxyProvider<T> proxyProvider, RetryPolicy retryPolicy) {

// 直接调用java动态代理，返回Proxy

return Proxy.newProxyInstance(

proxyProvider.getInterface().getClassLoader(),

new Class<?>[] { iface },

new RetryInvocationHandler<T>(proxyProvider, retryPolicy)

);

}

这里的工厂方法是直接通过JAVA动态代理构造，那么我们知道，最终所有的方法都会通过RetryInvocationHandler的invoke()方法进行代理。那么下面我们就分析一下invoke()方法

首先获得RetryPolicy，这里默认的RetryPolicy是failoverOnNetworkException

RetryInvocationHandler.invoke()方法通过反射调用method

Object ret = invokeMethod(method, args);

这里的invokeMethod其实就是在currentProxy上面进行调用 – currentProxy是泛型T类型 – 也就是接口ClientProtocolPB类型。代码如下所示

protected Object invokeMethod(Method method, Object[] args) throws Throwable {

try {

if (!method.isAccessible()) {

method.setAccessible(true);

}

// 注意，这里的currentProxy会发生切换

return method.invoke(currentProxy, args);

} catch (InvocationTargetException e) {

throw e.getCause();

}

}

注 ： 这里currentProxy就是当前使用的底层代理，当Namenode发生主从切换后，这个currentProxy也会发生相应的变化。

this.currentProxy = proxyProvider.getProxy();

之后就是在catch端处理异常情况

通过annotation判断这个操作是否是Idempotent (幂等,也就是执行多次是没有问题的，例如ClientProtocl当中的setReplication())以及AtMostOnce(Server端对于多个这个请求，只执行一次，例如CLientProtocol当中的rename())，（收获1，如何使用Annotation。）

// 判断方法是否是幂等的

boolean isIdempotentOrAtMostOnce = proxyProvider.getInterface()

.getMethod(method.getName(), method.getParameterTypes())

.isAnnotationPresent(Idempotent.class);

if (!isIdempotentOrAtMostOnce) {

isIdempotentOrAtMostOnce = proxyProvider.getInterface()

.getMethod(method.getName(), method.getParameterTypes())

.isAnnotationPresent(AtMostOnce.class);

}

// 在RetryPolicy上分析需要进行的retry动作

RetryAction action = policy.shouldRetry(e, retries++,

invocationFailoverCount, isIdempotentOrAtMostOnce);

之后调用policy.shouldRetry() (我们这里是FailoverOnNetworkExceptionRetry，通过RetryPolicies这个工厂类获得)，这里我们分析FailoverOnNetworkExceptionRetry.shouldRetry()方法

public RetryAction shouldRetry(Exception e, int retries,

int failovers, boolean isIdempotentOrAtMostOnce) throws Exception {

// 已经超出了最大充实次数

if (failovers >= maxFailovers) {

return new RetryAction(RetryAction.RetryDecision.FAIL, 0,

"failovers (" + failovers + ") exceeded maximum allowed ("

+ maxFailovers + ")");

}

// 判断异常类型

if (e instanceof ConnectException ||

e instanceof NoRouteToHostException ||

e instanceof UnknownHostException ||

e instanceof StandbyException ||

e instanceof ConnectTimeoutException ||

isWrappedStandbyException(e)) {

return new RetryAction(

RetryAction.RetryDecision.FAILOVER\_AND\_RETRY,

// retry immediately if this is our first failover, sleep otherwise

failovers == 0 ? 0 :

calculateExponentialTime(delayMillis, failovers, maxDelayBase));

} else if (e instanceof SocketException ||

(e instanceof IOException && !(e instanceof RemoteException))) {

if (isIdempotentOrAtMostOnce) {

return RetryAction.FAILOVER\_AND\_RETRY;

} else {

return new RetryAction(RetryAction.RetryDecision.FAIL, 0,

"the invoked method is not idempotent, and unable to determine " +

"whether it was invoked");

}

} else {

return fallbackPolicy.shouldRetry(e, retries, failovers,

isIdempotentOrAtMostOnce);

}

}

如果失败的次数已经超过最大的次数，那么就返回一个RetryAction.RetryDecision.FAIL的RetryAction.

如果抛出的异常是ConnectionException、NoRouteToHostException、UnKnownHostException、StandbyException、RemoteException中的一个，说明底层代理在RPC过程中Active NN连不上或者宕机或者已经发生主从切换了，那么就需要返回一个RetryAction.RetryDecision.FAILOVER\_AND\_RETRY的RetryAction，需要执行performFailover()操作，然后用另外一个NN的底层代理重试。

如果抛出的异常是SocketException、 IOException或者其他非RemoteException的异常，那么就无法判断这个RPC命令到底是不是执行成功了。可能是本地的Socket或者IO出问题，也可能是NN端的Socket或者IO问题。那就进行进一步的判断：如果被调用的方法是idempotent的，也就是多次执行是没有副作用的，那么就连接另外的一个底层代理重试；否则直接返回RetryAction.RetryDecision.FAIL。

通过ecpetion的情况，获得了对应的RetryAction之后，就会在proxyProvider上面调用performFailover()方法更新currentProxy，然后继续循环。

if (action.action == RetryAction.RetryDecision.FAILOVER\_AND\_RETRY) {

synchronized (proxyProvider) {

if (invocationAttemptFailoverCount == proxyProviderFailoverCount) {

proxyProvider.performFailover(currentProxy);

proxyProviderFailoverCount++;

currentProxy = proxyProvider.getProxy();

} else {

LOG.warn()

}

}

当需要进行重连的时候呢，则先执行performFailover方法，也非常简单，就是将currentProxyIndex进行一下变化。

public synchronized void performFailover(T currentProxy) {

currentProxyIndex = (currentProxyIndex + 1) % proxies.size();

}

非HA模式

这里方法的入口就是NameNodeProxies当中的createNonHAProxy()方法了。这里就是对xface也就是接口进行一个判断，然后返回对应proxy就好。

public static <T> ProxyAndInfo<T> createNonHAProxy(

Configuration conf, InetSocketAddress nnAddr, Class<T> xface,

UserGroupInformation ugi, boolean withRetries) throws IOException {

Text dtService = SecurityUtil.buildTokenService(nnAddr);

T proxy;

// 当前接口是ClientProtocol，调用对应的方法

if (xface == ClientProtocol.class) {

proxy = (T) createNNProxyWithClientProtocol(nnAddr, conf, ugi,

withRetries);

} else if (xface == JournalProtocol.class) {

proxy = (T) createNNProxyWithJournalProtocol(nnAddr, conf, ugi);

}

// ...

}

由于我们一直是以ClientProtocol作为例子，这里我们就看下createNNProxyWithClientProtocol()方法的实现

createNNProxyWithClientProtocol()

首先设置ProtocolEngine为ProtobufRpcEngine

RPC.setProtocolEngine(conf, ClientNamenodeProtocolPB.class, ProtobufRpcEngine.class);

之后产生一个ClientNamenodeProtocolPB类型的Proxy

ClientNamenodeProtocolPB proxy = RPC.getProtocolProxy(

ClientNamenodeProtocolPB.class, version, address, ugi, conf,

NetUtils.getDefaultSocketFactory(conf),

org.apache.hadoop.ipc.Client.getTimeout(conf), defaultPolicy)

.getProxy();

由于这个方法调用了RPC.getProtocoProxy()，我们下节重点讲解这个方法

RPC.getProtocolProxy()

getProtocolProxy()首先获得一个protocolEngine的实例，这个protocolEngine作为一个抽象类，表明一个特定接口对应的序列化引擎。

return getProtocolEngine(protocol,conf).getProxy(protocol, clientVersion,

addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy);

我们还是以ClientProtocol为例，由于ClientProtocol使用了protobuf作为序列化工具，所以在createNNProxyWithClientProtocol()的第一条语句就是注册ClientProtocol的序列化引擎为ProtobufRpcEngine。

static synchronized RpcEngine getProtocolEngine(Class<?> protocol,

Configuration conf) {

RpcEngine engine = PROTOCOL\_ENGINES.get(protocol);

// 通过反射创建rpcEngine实例

if (engine == null) {

Class<?> impl = conf.getClass(ENGINE\_PROP+"."+protocol.getName(),

WritableRpcEngine.class);

engine = (RpcEngine)ReflectionUtils.newInstance(impl, conf);

PROTOCOL\_ENGINES.put(protocol, engine);

}

return engine;

}

我们知道，这里的engine在最开始就设置为ProtobufRpcEngine，那么构造这个engine的proxy是什么呢 ？这里学习一下ProtoBufRpcEngine.getProxy()

public <T> ProtocolProxy<T> getProxy(Class<T> protocol, long clientVersion,

InetSocketAddress addr, UserGroupInformation ticket, Configuration conf,

SocketFactory factory, int rpcTimeout, RetryPolicy connectionRetryPolicy

) throws IOException {

// Invoker负责拦截所有请求到invoke()方法

final Invoker invoker = new Invoker(protocol, addr, ticket, conf, factory,

rpcTimeout, connectionRetryPolicy);

// 调用java动态代理，返回proxy

return new ProtocolProxy<T>(protocol, (T) Proxy.newProxyInstance(

protocol.getClassLoader(), new Class[]{protocol}, invoker), false);

}

从上面的代码当中我们知道，使用java的动态代理，所有的请求都会被Invoker类响应。那么这里我们学习一下Invoker的Invoke方法实现。Invoke主要做了三件事情 – a. 构造请求头 b.通过RPC.Client类发送请求 c.获取响应，返回

public Object invoke(Object proxy, Method method, Object[] args)

throws ServiceException {

// 通过上一节的学习，我们知道，pb接口的参数只有两个 - RpcController + Message

if (args.length != 2) {

throw new ServiceException("Too many parameters for request. Method: ["

+ method.getName() + "]" + ", Expected: 2, Actual: "

+ args.length);

}

if (args[1] == null) {

throw new ServiceException("null param while calling Method: ["

+ method.getName() + "]");

}

// 构造请求头域 - 标明在什么接口上调用什么方法

RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method);

if (LOG.isTraceEnabled()) {

LOG.trace(Thread.currentThread().getId() + ": Call -> " +

remoteId + ": " + method.getName() +

" {" + TextFormat.shortDebugString((Message) args[1]) + "}");

}

// 真正的参数，例如RenameRequestProto

Message theRequest = (Message) args[1];

final RpcResponseWrapper val;

try {

// 调用RPC.Client发送请求

val = (RpcResponseWrapper) client.call(RPC.RpcKind.RPC\_PROTOCOL\_BUFFER,

new RpcRequestWrapper(rpcRequestHeader, theRequest), remoteId);

} catch (Throwable e) {

// ...

throw new ServiceException(e);

}

// ...

Message prototype = null;

try {

// 获取返回参数类型 - RenameResponseProto

prototype = getReturnProtoType(method);

} catch (Exception e) {

throw new ServiceException(e);

}

Message returnMessage;

try {

// 获取返回参数

returnMessage = prototype.newBuilderForType()

.mergeFrom(val.theResponseRead).build();

// ...

} catch (Throwable e) {

throw new ServiceException(e);

}

// 返回结果

return returnMessage;

}

这里我们还要学习一下RequestHeaderProto是如何定义的，如下代码所示。主要是告诉RPC.Server，我们在哪一个接口的上，调用了什么方法。

message RequestHeaderProto {

// 调用的方法名

required string methodName = 1;

// 在什么protocol上调用

required string declaringClassProtocolName = 2;

// 版本号

required uint64 clientProtocolVersion = 3;

}

2.3 NamenodeRpcServer

我们知道Server的启动主要是在Namenode这个类当中，这里我们看下Namenode类代码。namenode当中有个rpcServer的引用 – rpcServer，然后将这个rpcServer启动起来。

初始化 ： 在initialize()方法中，NameNodeRpcServer rpcServer = createRpcServer(conf);

启动 ： startCommonServices()方法中，rpcServer.start();

然后只需要分析NameNodeRpcServer即可

初始化

构造底层接口实现。在[《RPC-ClientProtocol》](http://xupeng.sinaapp.com/rpc-clientprotocol/)章当中我们知道，真正在HDFS内存中执行RPC调用的响应类是NameNodeRpcServer – 也就是这里的**this**引用。而底层通过protobuf调用的类是一个**BlockingService**类型的类。而将这两个类适配起来的类是 –**ClientNamenodeProtocolServerSideTranslatorPB**

ClientNamenodeProtocolServerSideTranslatorPB

clientProtocolServerTranslator =

new ClientNamenodeProtocolServerSideTranslatorPB(this);

BlockingService clientNNPbService = ClientNamenodeProtocol.

newReflectiveBlockingService(clientProtocolServerTranslator);

构造RPC.Server。通过RPC.Builder构造一个RPC.Server的引用。这的内容我们在下一节重点分析

this.serviceRpcServer = new RPC.Builder(conf)

.setProtocol(

org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)

.setInstance(clientNNPbService)

.setBindAddress(bindHost)

.setPort(serviceRpcAddr.getPort()).setNumHandlers(serviceHandlerCount)

.setVerbose(false)

.setSecretManager(namesystem.getDelegationTokenSecretManager())

.build();

启动Server

void start() {

clientRpcServer.start();

if (serviceRpcServer != null) {

serviceRpcServer.start();

}

}

构造clientRpcServer

如下代码所示，这里最重要的是设置了Protocol为ClientNamenodeProtocolPB，同时impl设置为clientNNPBService，也就是上面一节分析得出的clientNNPbService(其中有个callBlockingMethod)方法

this.serviceRpcServer = new RPC.Builder(conf)

.setProtocol(

org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)

.setInstance(clientNNPbService)

.setBindAddress(bindHost)

.setPort(serviceRpcAddr.getPort()).setNumHandlers(serviceHandlerCount)

.setVerbose(false)

.setSecretManager(namesystem.getDelegationTokenSecretManager())

.build();

我们看下build()方法的实现，这里依然是使用**ProtocolEngine.getServer()**方法来获取Server类

public Server build() throws IOException, HadoopIllegalArgumentException {

if (this.conf == null) {

throw new HadoopIllegalArgumentException("conf is not set");

}

if (this.protocol == null) {

throw new HadoopIllegalArgumentException("protocol is not set");

}

if (this.instance == null) {

throw new HadoopIllegalArgumentException("instance is not set");

}

return getProtocolEngine(this.protocol, this.conf).getServer(

this.protocol, this.instance, this.bindAddress, this.port,

this.numHandlers, this.numReaders, this.queueSizePerHandler,

this.verbose, this.conf, this.secretManager, this.portRangeConfig);

}

我们继续学习ProtocolEngine.getServer() – 这里是ProtobufRPCEngine的实现，注意这里构造的是ProtobufRPCEngine的内部类Server。

public RPC.Server getServer(Class<?> protocol, Object protocolImpl,

String bindAddress, int port, int numHandlers, int numReaders,

int queueSizePerHandler, boolean verbose, Configuration conf,

SecretManager<? extends TokenIdentifier> secretManager,

String portRangeConfig)

throws IOException {

return new Server(protocol, protocolImpl, conf, bindAddress, port,

numHandlers, numReaders, queueSizePerHandler, verbose, secretManager,

portRangeConfig);

}

ProtobufRpcEngine.Server

看下Server这个内部类的构造方法。要特别注意构造方法的最后一段，注册了这个接口类 –**protocolClass** 以及 实现类 – **protocolImpl** 的映射关系。这样未来Client的请求到达时，就可以通过这个Mapping获得具体的实现类。这非常重要。

public Server(Class<?> protocolClass, Object protocolImpl,

Configuration conf, String bindAddress, int port, int numHandlers,

int numReaders, int queueSizePerHandler, boolean verbose,

SecretManager<? extends TokenIdentifier> secretManager,

String portRangeConfig)

throws IOException {

super(bindAddress, port, null, numHandlers,

numReaders, queueSizePerHandler, conf, classNameBase(protocolImpl

.getClass().getName()), secretManager, portRangeConfig);

this.verbose = verbose;

registerProtocolAndImpl(RPC.RpcKind.RPC\_PROTOCOL\_BUFFER, protocolClass,

protocolImpl);

}

Server内部类还有一个非常重要的功能是实现了一个ProtoBufRpcInvoker类。这个类会在RPC.Server响应RPC请求时调用Invoker的call()。这里我们重点看下call()方法。

public Writable call(RPC.Server server, String protocol,

Writable writableRequest, long receiveTime) throws Exception {

// 获取rpc调用头

RpcRequestWrapper request = (RpcRequestWrapper) writableRequest;

RequestHeaderProto rpcRequest = request.requestHeader;

// 获得调用的接口名，方法名，版本号

String methodName = rpcRequest.getMethodName();

String protoName = rpcRequest.getDeclaringClassProtocolName();

long clientVersion = rpcRequest.getClientProtocolVersion();

if (server.verbose)

LOG.info("Call: protocol=" + protocol + ", method=" + methodName);

// 获得该接口在Server侧对应的实现类

ProtoClassProtoImpl protocolImpl = getProtocolImpl(server, protoName,

clientVersion);

BlockingService service = (BlockingService) protocolImpl.protocolImpl;

MethodDescriptor methodDescriptor = service.getDescriptorForType()

.findMethodByName(methodName);

if (methodDescriptor == null) {

String msg = "Unknown method " + methodName + " called on " + protocol

+ " protocol.";

LOG.warn(msg);

throw new RpcNoSuchMethodException(msg);

}

// 反序列话参数

Message prototype = service.getRequestPrototype(methodDescriptor);

Message param = prototype.newBuilderForType()

.mergeFrom(request.theRequestRead).build();

Message result;

try {

// ...

// 在实现类上调用callBlockingMethod方法，级联适配调用到NamenodeRpcServer

result = service.callBlockingMethod(methodDescriptor, null, param);

// ...

} catch (ServiceException e) {

throw (Exception) e.getCause();

} catch (Exception e) {

throw e;

}

return new RpcResponseWrapper(result);

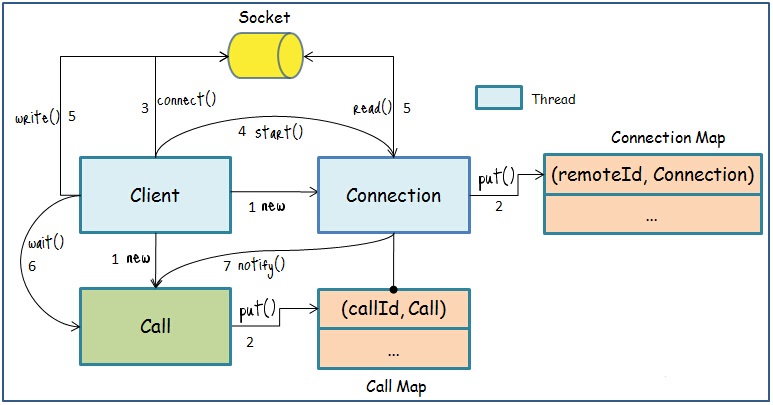
}

2.4 RPC.Client

proxy会调用RPC.client.call方法向Server发送一个RPC请求，RPC.Client接收调用，传入一个Writable的参数，调用远程的方法，返回一个Writable的结果。

Client发送接受流程，只有一个入口，就是call方法，程序调用call方法远程调用server上的方法，传入一个Writable参数，返回一个Writable参数，如果远端server出现异常，则也在call当中抛出：

整个Client的工作流可以参考如下图：



1) Client线程将输入参数值封装成Call,Call在原来的基础上增加了RPC完成标识和RPC返回值信息。随后，创建一个Connection

2)用ConnectionId作为key，将新建的Connection放入Client的Connection Map，以callId作为key，将之前的Call放入Connection的Call Map

3)通过socket建立连接，并发送RPC Header和Protocal Header

4)启动Connection线程

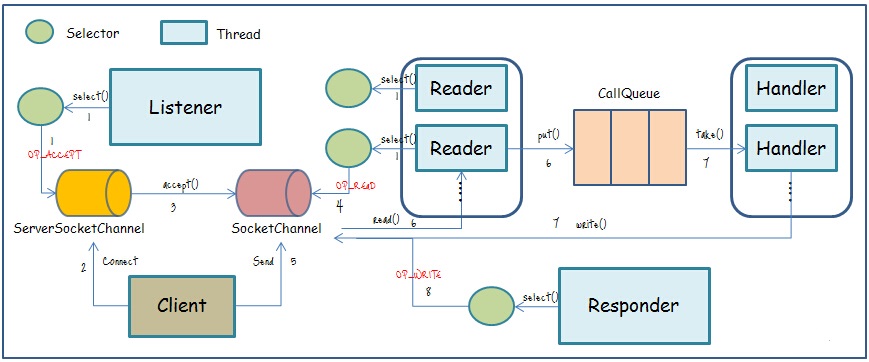
5)Connection线程监听socket，等待server response；同时，Client线程发送RPC Parameters

6)Client等待Call返回，call.wait()

7)Connection线程收到response，唤醒Client线程读取Call返回值，call.notify()

2.5 RPC.server

当中用到了Java的并发，NIO,多线程技术，下面是流程图：



1) Listener线程的selector在ServerSocketChannel上注册OP\_ACCEPT事件，并且创建Reader线程池。每个Reader的selector此时并不监听任何Channel；

2)Client发送TCP连接，触发Listener的selector唤醒Listener线程；

3)Listener调用ServerSocketChannel.accept()创建一个新的SocketChannel；

4) Listener从Reader线程池中挑选一个，将选中Reader的selector在新建的 SocketChannel上注册OP\_READ事件；

5)Client发送TCP数据包，触发Reader的selector唤醒Reader线程；

6)Reader从SocketChannel中读取数据，封装成Call，然后放入CallQueue；

7)最初，Handler线程池中的线程都block在CallQueue（调用BlockingQueue.take()）上，当有Call被放入后，其中一个Handler线程被唤醒，然后根据Call的信息调用对应的实现方法。随后，Handler尝试将Response写入SocketChannel；

8)如果Handler发现无法将Response完全写入SocketChannel，将让Responder的selector注册OP\_WRITE事件。当socket恢复正常，可以继续尝试了，Responder将被唤醒，继续做尝试。当然，如果一个Call Response在一定事件内都无法被写入，会被Responder移除。