Cumulus 代码解析

Dislab, Computer Science, Nanjing University
Vither Chien
zhinhai@gmail.com

忙里不得闲，写个报告吧，名字就叫 Cumulus 代码解析。

写流程前序

先谈写流程，Cumulus，而非 Nimbus，代号你们大概懂的，新老架构而已。不喜欢 Cumulus，架构很垃圾，但是也是自己架构的。Nimbus 一直没有时间实现，后面我会大概写写 Nimbus 的架构。

注意整个过程，这个系统一直都是支持很多种文件系统的，也就是我们要修改的大部分都是多态继承中走的，中间的很多过程都是为了支持这样的设计，才会显得有点累赘，有点多余的。

在 common 包里的 org.apache.hadoop.fs 子包中有 FsShell.java 文件，命令行中的-put、-get 等命令就是这个 java 文件首先处理的，这个是整个 Client 端的 shell 命令入口。

```
/**
 * main() has some simple utility methods
 */
public static void main(String argv[]) throws Exception {
  FsShell shell = new FsShell();
  int res;
  try {
    res = ToolRunner.run(shell, argv);
  } finally {
    shell.close();
  }
  System.exit(res);
}
```

根据不同的 argv 系统会调用不同的函数，在设计模式中类似翻译者模式。上面的 ToolRunner 调用了一圈，会调用 FsShell 类中的 run 函数（多态，不解释）。我们看代码，根据不同的命令行输入，该函数把这些命令行翻译成要执行的相应操作。下面我们一步步追踪 put 命令的执行情况。这就是我们最为关注的写流程。

```
if ("-put".equals(cmd) || "-copyFromLocal".equals(cmd)) {
  Path[] srcs = new Path[argv.length-2];
  for (int j=0 ; i < argv.length-1 ;)
    srcs[j++] = new Path(argv[i++]);
  copyFromLocal(srcs, argv[i++]);
} else if ("-moveFromLocal".equals(cmd)) {
  Path[] srcs = new Path[argv.length-2];
  for (int j=0 ; i < argv.length-1 ;)
    srcs[j++] = new Path(argv[i++]);
  moveFromLocal(srcs, argv[i++]);
```

接下来调用 moveFromLocal 函数，这个函数主要分流一下 stdin 的情况，标准输入，别的情况继续往下走。

```
void copyFromLocal(Path[] srcs, String dstf) throws IOException {
  Path dstPath = new Path(dstf);
  FileSystem dstFs = dstPath.getFileSystem(getConf());
  if (srcs.length == 1 && srcs[0].toString().equals("-"))
    copyFromStdin(dstPath, dstFs);
  else
    dstFs.copyFromLocalFile(false, false, srcs, dstPath);
}
```

接下来我们会追踪到 FileSystem.java 中，这是个文件系统基类，从名字就可以看出来。

```
public void copyFromLocalFile(boolean delSrc, boolean overwrite, Path src, Path dst) throws
IOException {
  Configuration conf = getConf();
  FileUtil.copy(getLocal(conf), src, this, dst, delSrc, overwrite, conf);
}
```

接下来直接到 FileUtil.java 中的 copy 函数，这个也是很大统一的那种。

```
public static boolean copy(FileSystem srcFS, Path src, FileSystem dstFS, Path dst, boolean
deleteSource, boolean overwrite, Configuration conf) throws IOException {
  FileStatus fileStatus = srcFS.getFileStatus(src);
  return copy(srcFS, fileStatus, dstFS, dst, deleteSource, overwrite, conf);
}
```

接下来调用 FileUtil.java 中的 copy 重载的函数，参数类型是就把第二个类型变成了 Path[]，这个函数会把 Path[]中的每个单个文件分解成一个个的单个文件，然后每个文件跑接下来的流程 copy 函数。

接下来就是我们逻辑上要敲的第一行代码的地方了：在 copy(FileSystem srcFS, FileStatus srcStatus, FileSystem dstFS, Path dst, boolean deleteSource, boolean overwrite, Configuration conf)函数中添加我们要的文件长度，这个位置不是太早，太早的话我们要更早的打出分支，更不是很好的支持多态，但是也不能太晚，走出这个位置，再通过 FileSystem 类来获得文件长度就有点困难了，所以我们这里获得文件长度，接下来对于相同操作，我们选择完成重载函数，然后调用我们旁路流程。

```
try {
  long fileSize = srcFS.getFileStatus(src).getLen();
  in = srcFS.open(src);
  out = dstFS.create(dst, fileSize, overwrite);
  IOUtils.copyBytes(in, out, conf, true);
}
```

上面代码 create 函数会向 namenode 进行存储申请，然后 copyBytes 函数会根据 namenode 的反馈信息（信息的 RPC，在下面）进行实际数据传输与写入。接下来我们分两部分对代码进行分析。

Client 端申请文件写流程

处理过程中，程序会判断源路径是目录还是文件（本例是文件），然后对源文件建立输入流（in），目标文件路径建立输入流（out），再调用 IOUtils.copyBytes()函数写入数据。

然后我们就会对每个与原流程对应的操作都会重载一个带有 long fileSize 的重载函数。这个操作充斥了整个 common 项目。先走 create 函数分支。

又回到了 Filesystem.java，很多 create 函数的重载版本，对于每一个，我们都要完成一个带有 fileSize 的重载版本，整个过程不是很复杂，最后调用一个 abstract 的 create 函数，然后整个 common 项目到此为止(我们还需要对 common 中的每个 FileSystem 的子类实现一个带有 fileSize 的 create 方法，这是 java 的抽象方法的要求，函数体只需要简单的 return null;)。HDFS 中实现了一个 DistributeFileSystem 类，该类实现了这样的一个 create 操作。接下来我们进入 HDFS。

HDFS 中的 org.apache.hadoop.hdfs 包中的 DistributedFileSystem.java 文件是我们处理 HDFS 的切入点。重载这里的 create 函数，模仿这里的做法，只是简单的加入 fileSize。

```
@Override
public FSDataOutputStream create(Path f, long fileSize, FsPermission permission, boolean
overwrite, int bufferSize, Progressable progress) throws IOException{
    statistics.incrementWriteOps(1);
    return new FSDataOutputStream(dfs.create(getPathName(f), fileSize, permission,
    overwrite ? EnumSet.of(CreateFlag.OVERWRITE) : EnumSet.of(CreateFlag.CREATE),
    progress, bufferSize), statistics);
}
```

接着就到了 DFSClient.java 文件，这部分的重载略有点复杂度，主要是要把 namenode 的 create 方法重载出来一个返回类型是 long 的函数版本，用来计算对于这样的 fileSize 应该用多大的 blocksize 来对文件进行切片，然后再创建 DFSOutputStream 类对象（构造函数中有 blocksize 域），原来的方法是在 DFSOutputStream 的构造函数中进行对 namenode create 方法进行 RPC，我们重载的方式是更早的进行 RPC，得到 blocksize，然后再调用一个简单的构造函数，创建 DFSOutputStream 对象。在这以后的过程中我们需要知道 packetsize，所以我们在这里传进去一个 long 类型的参数 packetsize。

```
public OutputStream create(String src, long fileSize, FsPermission permission,
EnumSet<CreateFlag> flag, boolean createParent, Progressable progress, int buffersize) throws
IOException {
    /* Check for parameters is omitted here. */
    long blockSize;
    try {
        blockSize = namenode.create(src, fileSize, writePacketSize, masked, this.clientName,
        new EnumSetWritable<CreateFlag>(flag), createParent);
    } catch(RemoteException re) {
        /* throw Exceptions; */
    }
    OutputStream result = new DFSOutputStream(this, src, masked,  flag, createParent,
    progress, blockSize, buffersize, conf.getInt(DFSConfigKeys.DFS_BYTES_PER_CHECKSUM_KEY,
    DFSConfigKeys.DFS_BYTES_PER_CHECKSUM_DEFAULT));
    leasechecker.put(src, result);
    return result;
}
```

这里先对 namenode 进行 RPC 调用，然后创建的 DFSOutputStream 跟

```
    out = dstFS.create(dst, fileSize, overwrite);
    IOUtils.copyBytes(in, out, conf, true);
```

这块代码段有很大的相似之处，因为在 DistributedFileSystem 中，就是 DFSOutputStream 负

责的 copyBytes 操作。这是后话。

接下来我们在 namenode 中分析我们的 PRC 中使用的 create 函数，在这之前我们需要在 namenode 的接口类 ClientProtocol.java 中添加这个重载版本的 create 方法的声明。然后跳转到 org.apache.hadoop.hdfs.server.namenode 包，如法炮制。

```
public long create(String src, long fileSize, int packetSize, FsPermission masked, String
clientName, EnumSetWritable<CreateFlag> flag, boolean createParent) throws IOException {
    String clientMachine = getClientMachine();
    /* Check for parameters is omitted here. */
    long blockSize = namesystem.startFile(src, fileSize, packetSize, new
    PermissionStatus(UserGroupInformation.getCurrentUser().getShortUserName(),
    null, masked), clientName, clientMachine, flag.get(), createParent);
    myMetrics.numFilesCreated.inc();
    myMetrics.numCreateFileOps.inc();
    return blockSize;
}
```

然后就是重载 namesystem 中的 starFile 方法，然后根据原来的该方法的写法，我们写我们的重载版本，调用 startFileInternal 的重载方法。

```
long startFile(String src, long fileSize, long packetSize,PermissionStatus permissions, String holder,
String clientMachine,EnumSet<CreateFlag> flag, boolean createParent)throws Exceptions…
{
    long blockSize = startFileInternal(src, fileSize, packetSize,permissions, holder, clientMachine,
    flag, createParent);getEditLog().logSync();
        …
    return blockSize;
}
```

startFileInternal 是我们表现大手笔的地方了，首先一下常规检查，然后调用 addFile 函数，该函数根据文件长度和 packetsize 生成合适的编码矩阵，计算 blockSize，计算最后一个切片的大小 tailSize，然后调用 chooseRandom 函数选择合适个数的数据节点作为文件的存储位置，最后再对文件申请 blockID header，然后对每个原始 block 根据序来生成合适的 ID，再根据编码矩阵，和 chosenDN，生成每一个 LocatedBlockWithCodingFactor 类，该类继承了 LocatedBlock 类，并加进去了编码因子（从编码矩阵获得），这个信息会暂存到一个 map 中，供下面 Client 端与 Datanode 交互时使用，整个文件信息会写在 INodeFile 中，表现为 INodeFileUnderConstruction，等待确认。之后 startFileInternal 函数返回这个切片过程的 size，共 DFSClient 端使用。整个过程甚为复杂！

```
private long startFileInternal(String src, long fileSize, long packetSize, PermissionStatus
permissions, String holder, String clientMachine,EnumSet<CreateFlag> flag, boolean
createParent) throws Exceptions…{
    INodeFileUnderConstruction newNode = null;
    writeLock();
    try {
        boolean overwrite = flag.contains(CreateFlag.OVERWRITE);
        …
        boolean pathExists = dir.exists(src);
        …
```

```
        if (isPermissionEnabled) {
            if (overwrite && pathExists) {
                checkPathAccess(src, FsAction.WRITE);
            }
            else {
                checkAncestorAccess(src, FsAction.WRITE);
            }
        }
        /* Check for parameters is omitted here. */
        try {
           DatanodeDescriptor clientNode =
           host2DataNodeMap.getDatanodeByHost(clientMachine);
           checkFsObjectLimit();
           long genstamp = nextGenerationStamp();
           newNode = dir.addFile(src, fileSize, packetSize, permissions,holder, clientMachine,
    clientNode, genstamp);
           if (newNode == null) {
               throw new IOException("…");
           }
        leaseManager.addLease(newNode.getClientName(), src);
        …
        catch (IOException ie) {… throw ie;}}
        finally {writeUnlock();}
        return newNode.getBlockSize();
}
```

下面是我们的核心，FSDirectory.java 中的 addFile 函数，红色部分是我们的重点，可以说是重中之重！基本全红了，慢慢看吧，这块是核心中的核心。标记为红色的代码部分我会重点的分析一下。

```
INodeFileUnderConstruction addFile(String path, long fileSize, long packetSize, PermissionStatus
permissions, String clientName, String clientMachine, DatanodeDescriptor clientNode,
long generationStamp) throws IOException {
    waitForReady();
    long modTime = now();
    CodingMatrix codingMatrix = new CodingMatrix(fileSize);
    long blockSize = fileSize % (codingMatrix.getRow()*packetSize) == 0 ? fileSize /
    codingMatrix.getRow() : (fileSize / (codingMatrix.getRow()*packetSize) + 1) * packetSize;
    long tailSize = fileSize - blockSize * (codingMatrix.getRow() - 1);
    …
    DatanodeDescriptor chosenNodes[] = getBlockManager().replicator.chooseRandom(path,
    codingMatrix.getColumn(), clientNode, blockSize);
    List<DatanodeDescriptor> chosen = Arrays.asList(chosenNodes);
    INodeFileUnderConstruction newNode = new INodeFileUnderConstruction( permissions,
    codingMatrix, blockSize, tailSize, modTime, clientName, clientMachine, clientNode);
    writeLock();
```

```
try {
        newNode = addNode(path, newNode, UNKNOWN_DISK_SPACE, false);
        }
finally {
        writeUnlock();
        }
long idHeader = getBlockManager().getNextBlockID(path, codingMatrix.getRow());
long MASK = 0xffffL;
dHeader &= (~MASK);
long fileLength = newNode.computeContentSummary().getLength();
for(int i = 0; i < codingMatrix.getRow(); i++){
        getFSNamesystem().checkFsObjectLimit();
        Block newBlock = new Block(idHeader | (1 << i), 0,
        getFSNamesystem().getGenerationStamp());
        int[] codingFactor = codingMatrix.getCodingFactorList(i);
        byte[] s = codingMatrix.getRowArray(i);
        DatanodeDescriptor targets[] = chooseDataNodes(s, chosenNodes);
        writeLock();
        try {
                INode[] pathINodes = getExistingPathINodes(path);
                int inodesLen = pathINodes.length;
                getFSNamesystem().checkLease(path, clientName, pathINodes[inodesLen - 1]);
                newBlock = addBlock(path, pathINodes, newBlock, targets);
                for (DatanodeDescriptor dn : targets) {
                        dn.incBlocksScheduled();
                }
        }
        finally {
                writeUnlock();
        }
        LocatedBlock b = new LocatedBlock(newBlock, targets, fileLength);
        LocatedBlockWithCodingFactor bf =new LocatedBlockWithCodingFactor(b,
        codingFactor);getBlockManager().enQueue(bf);
}
INode[] inodes = getExistingPathINodes(path);
INodeFileUnderConstruction fileINode =
(INodeFileUnderConstruction)inodes[inodes.length-1];
DatanodeDescriptor[] target = new DatanodeDescriptor[1];
for(int j = 0; j <codingMatrix.getColumn(); j++){
        Block u = new Block(idHeader | (codingMatrix.getLastID(j)), 0,
        getFSNamesystem().getGenerationStamp());target[0] = chosenNodes[j];
        updateCount(inodes, inodes.length-1, 0,
        fileINode.getBlockSize()*fileINode.getReplication(), true);
        BlockInfoUnderConstruction uInfo = new BlockInfoUnderConstruction(u,
```

```
            fileINode.getReplication(), BlockUCState.UNDER_CONSTRUCTION, target);
            getBlockManager().addINode(uInfo, fileINode);
            fileINode.addBlock(uInfo);
        }
        fsImage.getEditLog().logOpenFile(path, newNode);
        return newNode;
}
```

首先是 codingMatrix，通过 fileSize 来调用构造函数，这也是我们费尽苦力把 fileSize 传递过来的一个重要原因，当然，我们还接下来计算 blockSize，然后通过 RPC 返回给 Client 端，这是 Client 端利用这个参数的意义，两端都需要，但是意义不一样吧。接下来的两行分别计算 blockSize 和 tailSize（这个参数重要用来存到 INode 中去的，这样每个文件才能知道它自身的大小。）

接下来就是 chooseRandom 函数了，这个函数也是花费了很大的精力才做出来的，它是一个远亲吧，离的看代码吧，难度倒不是很大，在你能看到了实现代码之后。它的声明在 BlockPlacementPolicy 类中，这是一个虚基类，有唯一的实现就是 BlockPlacementPolicyDefault 类，所以，实现的时候先在虚基类中加一个声明，然后在实现类中做实现。模仿一下原来的类似的一个函数就是了，我们只是简单的跟原来的那个函数一样，调用类似的 API 就可以搞定，慢慢调试下。以后做负载均衡，这个地方有挖掘的潜力，不过要调用 common 中的 API 或者再写别的东西。

接下来再回来 addFile，接下来就是注册该文件，然后注册每个编码后的 block 信息，之前的流程的注册 block 信息是后面来的，是每一个 block 在写 DataNode 之前来申请 ID，申请存储节点，然后写 INode 索引，获得 LocatedBlock，然后再写 DataNode，但是我们每个文件的 Block 之间现在有关系了，他们的 DataNode 不再是没有联系的了，而是根据 codingMatrix 来写的，所以我们需要在一个地方暂存一些信息，这里的做法是暂存 LocatedBlock，我们也可以选择暂存 chosenDataNode。

ID 的生成大概变化了一下原版的做法，原先是寻找下一个没有被用到的 long 类型数字，我们寻找的是下一个没有被用到的 long 类型数字的头 48bit，这是我们的 ID header，每个文件有唯一的 header，每个属于这个文件的 Block 的 ID 是在这个 header 的基础上尾部 16 个 bit 做下区分。

接下来就是对付每个编码后的 block 的 INode 和 LocatedBlock 信息了，我们对这个 LocatedBlock 做了一下加成，就是加了一个编码域，我们选择用继承的方式而不是组合的方式，这是一种选择方式而已。

编码矩阵类的实现见 CodingMatrix.java，这个类在 org.apache.hadoop.hdfs.server.namenod 包中，记得实现序列化，序列化的实现方式可以参考该类。首先实现 Writable 接口，实现 readFields、write 方法，一般会依赖另一个方法，就是该类从 DataInput 中进行的构造函数。以后有需要，可以参考，还有一个地方，就是 LocatedBlockWithCodingFactor 的序列化方法，它是继承了一个原本序列化的类，子类的序列化方式参考 LocatedBlockWithCodingFactor 就可以了。

INodeFile 这部分是非常复杂的，反反复复的进行了很多的设计，都不够很完美，最后由我和陈正亮确定了一个 128bits 的 header，然后存储 blockSize，tailSize，matrix。别的东西不变，当然了这部分的 get 和 set 函数要修改下，慢慢写。

至此，写流程部分的 Client 与 namenode 部分基本结束，这部分结束之后 Client 端通过 RPC 获得了 blockSize，iNodefile 中存储了要存储文件的 INodeFileUnderConstruction 对象，等待确认，一个 map 中存储了该文件每个 Block 的 LocatedBlockWithCodingFactor 信息，通过

该信息我们能够确定每个 block 将被发往哪些 DataNode。

Client 与数据准备过程

接下来我们分析另一部分，就是 Client 与 DataNode 交互，并写入文件，然后 DataNode 再与 NameNode 交互，进行确认，完成整个写流程。

```
public static void copyBytes(InputStream in, OutputStream out, Configuration conf, boolean
close) throws IOException {
    copyBytes(in, out, conf.getInt("io.file.buffer.size", 4096),  close);
}
```

返回 common 项目中 org.apache.hadoop.fs 包中的 FileUtil.java 中的 IOUtils.copyBytes(in, out, conf, true)语句，IOUtils 调用重载的 copyBytes()函数，并在 FSOutputSummer 类中生成带 checksum 的 writeChecksumChunk()函数，由于继承关系，最终还是调用 DFSClient 类中的 DFSOutputStream 类中的 writeChunk()函数。接下来我们跑到了 HDFS 项目中的 DFSOutputStream.java，继续我们的分析。

DFSOutputStream 类继承 FSOutputSummer 类，故由 DFSOutputStream 类来实现 writeChunk()函数，DistributedFileSystem 类的 create()方法返回是一个包含了 DFSOutputStream 类的 FSDataOutputStream 类型。接着继续讨论 DFSOutputStream 类的 writeChunk()函数。首先介绍下写入数据流程，这又是一个超百行代码的函数，并且是同步方法。首先，参数检查，然后装 packet（Block 细分为 packet 然后一个个 packet 写过去的），packet 分头部，数据段，校验码段三部分，分别写，写好的 packet 放在一个发送队列里等待，最后一个 packet 发送完毕后再发送一个空的，作为同步或者 ACK 用吧。

writeChunk 函数有一个很隐蔽的 bug，就是原来有的 bug，就是 chunksPerPacket 这个参数，它是表明这个 packet 应该有多少个 chunks 的一个表征，由 computerPacketChunkSize 函数来负责计算，主要的思想就是如果还有很多的数据，那么这个数值就应该是 Max 的，负责就是把剩下所有的数据都给装进去就 OK 了，这是它的思想，但是该函数的实现不是这样子的，它有一个小的 bug 就是在是用上次计算的结果来作为这次的 chunksPerPacket，其实我们应该以当前的数据量来计算这个值的，而不是以上次的数据量来计算这个值，chunksPerPacket 在各个 block 之间还是共享的，也就是上个 block 走完了，剩余数据量为零了，下个 block 的第一个 packet 的 chunksPerPacket 是 1（chunksPerPacket 返回最小值为 1），然后我们在 DataNode 端根据这个东西进行的编码就会很麻烦。这个在此记录一下。

```
private void computePacketChunkSize(int psize, int csize) {
    int chunkSize = csize + checksum.getChecksumSize();
    int n = PacketHeader.PKT_HEADER_LEN;
    chunksPerPacket = Math.max((psize - n + chunkSize-1)/chunkSize, 1);
    packetSize = n + chunkSize*chunksPerPacket;
    …
    }
```

我们接下来在分析这个处理队列的这个方法，分析它是怎么把这些 packet 发送过去的。dataQueue 是由 DFSOutputStream 来维护的，实际的发送操作是这个类下面的子类 DataStreamer 来做的（DFSOutputStream 下面有一个 steamer 成员，就是 DataStreamer 类型的），主要的数据发送就是由这个成员来做的。所以我们的分析直接到了 DataStreamer，这是一个守护进程，它 extends 了 Daemon，构造函数首先是初始化了自己的成员变量，然后 run 函数开始执行，这是一个近两百行代码的 run 函数，所以要有耐心。首先我们要知道这

个类不仅仅是为了我们的流程而或者，它还有自己的用途，比如心跳包等。我们只分析跟我们写流程相关的代码段。

DFSOutputStream 中维护了两个重要的队列：

private LinkedList<Packet> dataQueue = new LinkedList<Packet>(); 和 private LinkedList<Packet> ackQueue = new LinkedList<Packet>(); 负责保证以上流水线持续进行。writeChunk()会先检查 data queue 是否已满，若满就等到有足够的空间来创建队列中下一个 packet。若 currentPacket 为空，则创建一个 new Packet，并将该 packet 写上 checksum、数据、序号等信息；接着检查该 packet 是否已满，若满则放入 data queue，并唤醒等待 data queue 的传输线程，即 DataStreamer。DataStreamer 的 run()函数来进行接下来的处理。函数会先判断队列里有无 packet，没有则等待 1s；若有则得到队列里的第一个 packet，然后由 nextBlockOutPutStream() 函数首先通过调用我们在 namenode 刚才放在 map 中的 LocatedBlockWithCodingFactor 来获得这个 block 的存储信息，主要包括编码因子，包括即将被发往的 DataNode 等信息。然后建立相应的 pipeline，主要包括写入流和返回 ack 流。

nodes[0]是获取 pipeline 中的第一个 datanode，接着写入指令（指令头、操作类型、stamp 等信息）。

Client 端这部分是非常复杂的，慢慢熟悉下就好。不过这部分更改的不多，所以我不准备在这里过多的说明什么。流程还是那样的，就是多了一个编码因子，还有我们的获得 LocatedBlock 信息的方式有一个改变，原来的是要为每一个 block 分配 LocatedBlock 信息，但是由于我们文件内部各个 block 之间的关联性，我们已经在文件申请写的时候把这一部分给做好了并且存放在一个 map 中，所以我们在这里获得 LocatedBlock 只需要去 map 中找就是了。

DataNode 写入磁盘并确认流程

接下来我们分析下 DataNode 端怎么处理，怎么接收，编码，确认。

org.apche.hadoop.hdfs.server.datanode 包中的 DataNode.java，当然本例中是写入操作的命令（DataTransferProtocol.OP_WRITE_BLOCK）。datanode 的 DataXceiver 是一个进程，持续监听各种命令。服务会监听到该命令，然后进入 writeBlock()函数的处理。首先我们来看一下 dataNode 的 main 函数，这是一个 command 设计模式的体现，dataNode 就一直在等各种 command，然后执行相应的操作。

```
public static void main(String args[]) {
    secureMain(args, null);
}
public static void secureMain(String args[], SecureResources resources) {
    try {
        StringUtils.startupShutdownMessage(DataNode.class, args, LOG);
        DataNode datanode = createDataNode(args, null, resources);
        if (datanode != null)
            datanode.join();
        } catch (Throwable e) {
            LOG.error(StringUtils.stringifyException(e));
            System.exit(-1);
        }
}
```

这是一个进程，我们再来找它的 run 函数。

```
public void run() {
    LOG.info(dnRegistration + "In DataNode.run, data = " + data);
    dataXceiverServer.start();
    ipcServer.start();
    while (shouldRun) {
        try {
            startDistributedUpgradeIfNeeded();
            offerService();
        } catch (Exception ex) {
            LOG.error("Exception: " + StringUtils.stringifyException(ex));
            if (shouldRun) {
            try {
                Thread.sleep(5000);
            }
            catch (InterruptedException ie) { }
            }
        }
    }
    LOG.info(dnRegistration + ":Finishing DataNode in: "+data);
    shutdown();
}
```

这里的主要先启动了 dataXceiverServer，然后该进程启动监听，Datanode 的服务大部分由 offerService()提供。

```
public void offerService() throws Exception {
    while (shouldRun) {
        try {
            long startTime = now();
            if (startTime - lastHeartbeat > heartBeatInterval) {
               lastHeartbeat = startTime;
               DatanodeCommand[] cmds = namenode.sendHeartbeat(dnRegistration,
               data.getCapacity(), data.getDfsUsed(), data.getRemaining(),
               xmitsInProgress.get(), getXceiverCount(), data.getNumFailedVolumes());
               myMetrics.heartbeats.inc(now() - startTime);
               if (!processCommand(cmds))
                    continue;
            }
            reportReceivedBlocks();
            DatanodeCommand cmd = blockReport();
            processCommand(cmd);
            …
            long waitTime = heartBeatInterval - (System.currentTimeMillis() - lastHeartbeat);
            synchronized(receivedBlockList) {
               if (waitTime > 0 && receivedBlockList.size() == 0) {
```

```
                try {
                receivedBlockList.wait(waitTime);
                } catch (InterruptedException ie) { }
              }
            }
        } catch(RemoteException re) {
        …
        } catch (IOException e) {…}
    }
}
```

offerService 主要提供三种服务，分别用红色标识出，第一种是发送心跳包、第二种就是 reportReceivedBlocks 函数，它报告在这一个时间间隔中受到的 block 给 namenode，然后 namenode 进行相应的确认操作。第三种就是处理来自外界的命令，比如写文件块，读文件块等命令。后两种服务我们都需要做修改，我们首先谈下 processCommand(cmd)函数。

在命令 case 为 DatanodeProtocol.DNA_TRANSFER 的情况下，调用 transferBlocks(Block blocks[], DatanodeInfo xferTargets[][])函数，然后分解为单个处理版本的 transferBlock(Block block, DatanodeInfo xferTargets[])函数，该函数在做了很多的参数检查之后执行

Daemon(new DataTransfer(xferTargets, block, this)).start();

上述语句先 new 了一个 DataTransfer 对象，然后再包装为 Daemon，然后启动这个守护进程。我们接下来我们看下 DataTransfer 的 run 函数。它做了 Blocksender，然后利用它进行发送。

继续找到 Blocksender，其中有一个 sendBlock 函数，这个是它的核心。这个是它的发送。

那么它的 receiver 呢，在 DataXceiver 中，我们找到 run 函数，核心的一条语句就是 processOp(op, in)，我们继续追踪这个。它是 DataTransferProtocol 中的 Receiver 子类的方法，各种 case，我们最关心 opWriterBlock 函数，继续追踪，就到了我们要写代码的核心。

opWriterBlock(DataInoutStream in)这个函数主要就是读去数据流，然后反序列化构造各种参数，然后再调用 protected abstract void opWriteBlock(DataInputStream in, Block blk, int pipelineSize, BlockConstructionStage stage, long newGs, long minBytesRcvd, long maxBytesRcvd, String client, DatanodeInfo src, DatanodeInfo[] targets, int factor, int[] factors, Token<BlockTokenIdentifier> blockToken)函数，注意这是个虚方法，实现的地方在 DataXceiver 中，我们找到该方法。

这里面的核心就是创建了一个 BlockReceiver 对象，然后由这个对象进行实际的数据接收，然后再把这些数据传递的发送给 pipeline 中的后面节点，这是由 sender 来做的，我们接下来重点分析一下 BlockReceiver。

首先我们需要保存各个 block 之间的信息，以处理编码，这里我们主要是通过 replicaInfo 来做的，replicaInfo 保存了当前参与编码的 block 的信息，然后将它们进行相应操作。然后在 replicaInfo 中保存了一个 PacketList，为缓存数据，每次 block 数据来的时候，如果是新，就建立 replicaInfo，否则，就根据编码信息与缓存进行编码，然后调用写磁盘函数写磁盘。最后调用 Datanode 在每一定的间隔会把这个时段收到的 block 上传给 Namenode 确认，我们简单的由于流程的不一致，这方面也要做一定的修改，之后整个系统的写流程已经表现的 very perfectly。