# hdfs总结

## NN的启动过程

1. Namenode如果是正常启动（非format），则会调用initialize方法，在这个方法里完成初始化和启动的全部工作:
   1. initialize方法首先进行一些权限方面的配置，然后初始化**this**.namesystem = **new** FSNamesystem(**this**, conf);大部分的启动工作都在FSNamesystem内部完成。
   2. FSNamesystem内部也有一个initialize方法，在这个initialize方法中，会初始化整个系统的大部分模块：
      1. 初始化**this**.dir = **new** FSDirectory(**this**, conf);
      2. 加载保存的文件系统镜像**this**.dir.loadFSImage（下面会具体讲到）
      3. 初始化安全模式**this**.safeMode = **new** SafeModeInfo(conf);然后调用setBlockTotal方法设置safeMode的blockTotal属性（因为此时已经从image和edits中恢复了文件系统的目录结构，所以可以获得系统内的所有block数目）
         1. SafeModeInfo负责处理安全模式的细节，setBlockTotal方法除了设置safemode的总block数目外，还会调用SafeModeInfo的checkMode方法
         2. checkMode会检查block数目是否达到配置的阈值（默认为0.999），如果没有达到会马上进入安全模式，并启动一个SafeModeMonitor的线程
         3. SafeModeMonitor负责每1秒检查一次block数目是否达到阈值（其实是等待DN向NN报到的过程），达到了则调用SafeModeInfo的leave方法退出安全模式（达到阈值之后会等待extension时间，默认为30秒）
         4. leave方法先调用FSNamesystem的processMisReplicatedBlocks方法处理所有block，然后销毁safeMode
         5. processMisReplicatedBlocks遍历blocksMap中所有block，找出：
            1. 副本不足的，添加到neededReplications
            2. 副本多余的，添加到recentInvalidateSets
            3. 系统不能识别的block（即block对应的文件不存在），添加到recentInvalidateSets
      4. 初始化几个线程，用于监控系统状态，包括：
         1. 心跳监控：HeartbeatMonitor,定期检查heartbeats中的DN是否过期
         2. 租约监控：LeaseManager.Monitor
         3. 副本状态监控：ReplicationMonitor，定期检查neededReplications和recentInvalidateSets，从中选取要添加和要减少的副本，并执行副本操作
            1. 对于neededReplications中，需要增加副本数的block，查询到合适的DN后添加到pendingReplications并从neededReplications移除（最后其实是添加到DatanodeDescriptor的replicateBlocks队列中，在下次心跳时会被发送到对应的DN）
            2. 选取DN，使用ReplicationTargetChooser的chooseTarget方法
            3. 对于recentInvalidateSets中需要删除block的所有DN，查找出本次应该执行的DN，添加到DatanodeDescriptor的invalidateBlocks列表中
            4. HeartbeatMonitor还会定期监控那些超时没有完成的副本请求，从pendingReplications中删除并再移动回neededReplications中
            5. 上面两种副本变更都会在下次心跳中发送给DN处理
         4. 正在处理中的副本监控：PendingReplicationMonitor，定期检查pendingReplications，选出超时的，添加到timedOutItems
         5. Datanode监控：DecommissionManager.Monitor，定期检查decommissioning node的状态，调用FSNamesystem的checkDecommissionStateInternal方法
      5. 另外还有几个数据结构，用于维护系统状态：
         1. 整个文件系统的逻辑层次：FSDirectory dir
         2. Block->BlcokInfo的映射：BlocksMap blocksMap（BlcokInfo包含了INode, datanodes, self ref等信息）
         3. 废弃的block信息Block -> TreeSet<DatanodeDescriptor>：CorruptReplicasMap corruptReplicas
         4. datanode到其上block的映射StorageID -> DatanodeDescriptor：NavigableMap<String, DatanodeDescriptor> datanodeMap（DatanodeDescriptor中存储了BlcokInfo，而通过BlcokInfo可以索引到datanode上所有的block）
         5. 废弃的block信息，StorageID -> ArrayList<Block>：Map<String, Collection<Block>> recentInvalidateSets
         6. 多余的block信息，StorageID -> TreeSet<Block>：Map<String, Collection<Block>> excessReplicateMap
         7. 心跳信息：ArrayList<DatanodeDescriptor> heartbeats
         8. 需要更多副本的block集合：UnderReplicatedBlocks neededReplications
         9. 正在执行副本备份的block集合：PendingReplicationBlocks pendingReplications
         10. 租约集合：LeaseManager leaseManager
         11. host到datanode的映射信息：Host2NodesMap host2DataNodeMap（因为一个host可能被映射到多个datanode）
   3. FSNamesystem初始化结束后，Namenode初始化rpc server：**this**.server = RPC.*getServer*。然后启动http和rpc服务器startHttpServer(conf);和**this**.server.start();
   4. 最后，会启动垃圾回收器：startTrashEmptier(conf);
2. FSDirectory，存储filesystem directory state，实时维护了filename->blockset的映射，并记录日志到硬盘。
   1. 启动NN时，在FSNamesystem的初始化方法中会调用loadFSImage来加载整个文件系统的镜像，实际是先后调用FSImage的loadFSImage和loadFSEdits两个方法
   2. FSImage的loadFSImage方法从Image文件中加载整个文件系统的镜像（因为配置项dfs.name.dir中的每个目录都保存了镜像文件，所以要先选取一个最新的），镜像文件存储格式为：



共有numFiles个文件，循环遍历每个文件，把读取的文件添加到系统的FSDirectory数据结构中。（上图中的block只包含了blockid，而没有包含block的位置信息，block到底存储在哪个DN上，要等待DN的报告）

* 1. FSImage的loadFSEdits方法实际是调用FSEditLog的*loadFSEdits*方法，这个方法循环读取edits文件，每次先读取操作符，直到读取到了*OP\_INVALID*，表示文件的结束。对于不同的操作符，执行不同的操作，最终都把日志中的变更应用到之前加载到内存中的文件系统上。具体参考FSEditlog.doc。

1. 所有的block信息都在BlocksMap中维护，其内部类BlockInfo维护了每个block的元数据信息，包括：
   1. 该block所属的文件InodeFile
   2. 存储该block的DN信息，存储在一个三元组上Object[] triplets;该三元组的结构如下图



每个存储该block的DN在triplets中占用三个位置，所以，对于副本数为n的block，triplets数组的长度就是3\*n。

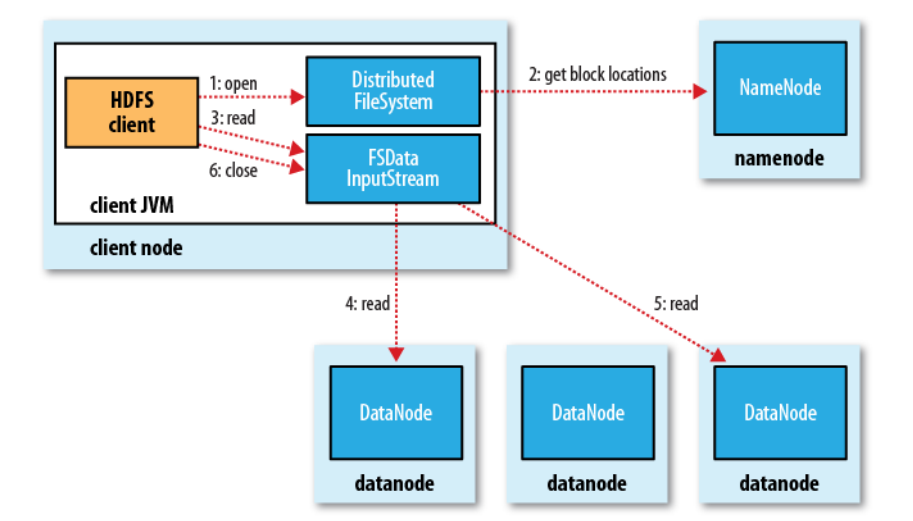
* + 1. 第一个位置指向DN对应的DatanodeDescriptor
    2. 第二个位置指向DN中存储的前一个block对应的BlockInfo
    3. 第三个位置指向DN中存储的后一个block对应的BlockInfo
    4. 每个DN在NN的数据结构中，对应一个DatanodeDescriptor，而存储在这个DN上的block列表信息并没有直接存储在DatanodeDescriptor中，DatanodeDescriptor中只存储了位于block链表最前端的一个block（BlockInfo），然后通过这个BlockInfo维护整个链表。上图中假设DatanodeDescriptor1的链表首部就是该block，所以prev1为null，prev2则指向链表中的下一个block
  1. 一个文件的逻辑结构图为：

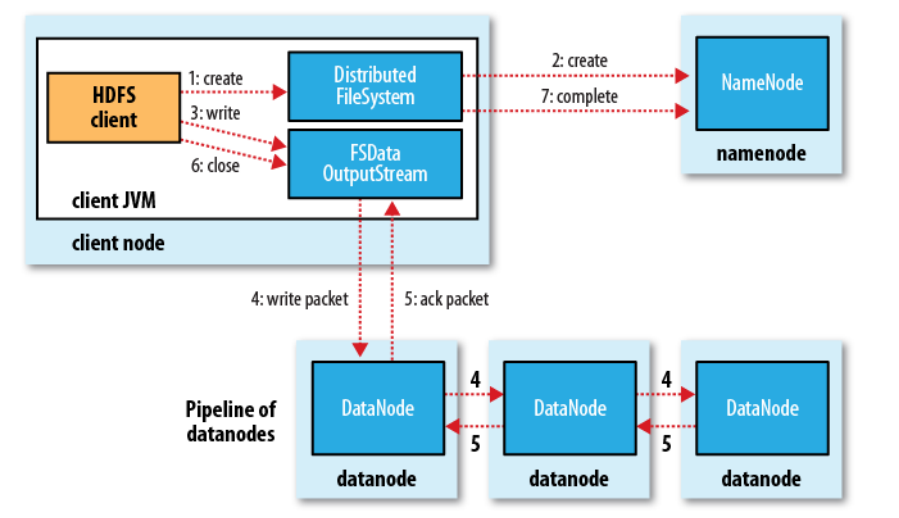


## DN的启动过程

1. DN的启动过程主要在startDataNode方法中完成
   1. 初始化storage = **new** DataStorage(); DataStorage表示DN存储目录的一些信息，负责升降级等操作
   2. 构造namenode的proxy，即建立到namenode的连接：**this**.namenode
   3. 获取data目录信息（可能会执行事务恢复等操作，主要是系统升级方面的）：storage.recoverTransitionRead(nsInfo, dataDirs, startOpt);
   4. 初始化daatnode的内部数据结构：**this**.data = **new** FSDataset(storage, conf); FSDataset负责block在DN上的实际存储，下面会详细分析。
   5. 初始化用于接收block的线程组：**this**.threadGroup = **new** ThreadGroup("dataXceiverServer");和用于接收数据的server：**this**.dataXceiverServer = **new** Daemon(threadGroup, **new** DataXceiverServer(ss, conf, **this**));
   6. 初始化block扫描器，用于定期的检查block：blockScanner = **new** DataBlockScanner(**this**, (FSDataset)data, conf);
   7. 初始化**this**.infoServer（即HttpServer）
   8. 初始化ipcServer = RPC.*getServer*（用于被client或其他datanode调用）
2. DN初始化完成后，会作为一个线程一直运行，在其run方法中，循环调用offerService方法，为NN提供服务
   1. offerService循环执行下面所有的操作：
      1. 远程调用NN的sendHeartbeat方法，发送心跳信息（此处并不包括block信息），并从NN接收返回的指令，然后处理这些指令processCommand(cmds)
      2. 如果receivedBlockList不为空，则说明有新收到的block要报告给NN，调用NN的blockReceived方法，然后清空receivedBlockList和delHints（receivedBlockList中的每个block都对应delHints的一条信息）
      3. 如果距离上次block report已经过了blockReportInterval（配置项dfs.blockreport.intervalMsec定义，默认是1小时），则要发送block report到NN（block的实体文件就是以blockid为根据命名）：
         1. 如果block report还没准备好，则调用FSDataset的requestAsyncBlockReport方法，要求AsyncBlockReport开始扫描磁盘（这次循环就此结束，等到block report准备好，可能要过很多次循环）
         2. 等到block report准备好了：
            1. 调用FSDataset的retrieveAsyncBlockReport（每次调用这个方法之前，必须先调用requestAsyncBlockReport）方法获取最终的block数组：Block[] bReport（虽然AsyncBlockReport获取的是粗略的block信息，但是此处获取的是精确的block信息，因为retrieveAsyncBlockReport方法中会调用reconcileRoughBlockScan重新构建block信息）
            2. 调用NN的blockReport方法发送report
      4. 如果DataBlockScanner还没有开始运行，则新建一个线程运行之（这一步应该可以放在循环之外吧？）
      5. 如果receivedBlockList此时为空，且还没有超过heartBeatInterval，则睡眠等待receivedBlockList.wait(waitTime);（由此可以看出，只要有新的block收到，马上就会开始下一轮心跳，而不必等待心跳间隔）
3. FSDataset继承自FSDatasetInterface，管理block的在DN上的存储。
   1. 构造函数中：
      1. 根据配置文件中定义的存储目录，初始化FSVolume数组（平时配置的时候只有一个，生成环境可以配置多个）
      2. 然后把FSVolume数组填充到FSVolumeSet，并调用volumes.getVolumeMap(volumeMap);获取所有block的信息，填充到volumeMap中
      3. 初始化asyncBlockReport = **new** AsyncBlockReport(**this**);并启动这个线程，异步的执行block报告
      4. 初始化asyncDiskService = **new** FSDatasetAsyncDiskService(roots);负责异步执行一些磁盘操作
   2. FSVolumeSet维护了FSVolume数组，大部分操作都是遍历FSVolume数组执行
   3. 每个FSVolume对应配置项dfs.data.dir（和dfs.name.dir不同，这里如果配置多个位置，并不是每个位置都写一份数据，而是选择其中一个位置写）中的一个存储位置
      1. 维护了四个目录，tmpDir，blocksBeingWritten，detachDir，currentDir，分别对应了存储目录中的四个文件夹
      2. 构造函数：
         1. 初始化**this**.dataDir = **new** FSDir(currentDir); FSDir中维护了整个存储目录中的block信息
         2. 初始化上面提到的四个目录，其中需要注意：
            1. tmpDir目录下的所有文件全部删除
            2. blocksBeingWritten，如果配置项dfs.support.append为true，即支持append操作，则调用recoverBlocksBeingWritten进行恢复，否则，删除所有文件
         3. 初始化DF和DU
      3. FSVolume的操作主要围绕着FSDir进行
   4. FSDir负责维护FSVolume对应的文件夹下的所有block信息。
      1. FSDir是一个树结构，可能包含子节点FSDir children[];
      2. 构造函数中，递归的构建整个文件树，构建完成时，就能统计出该目录下的所有block
      3. 添加block时，先检查当前目录，如果超限了（配置项dfs.datanode.numblocks定义了每个文件夹中最多block数目，默认64），则添加到子目录中（递归执行）
      4. block实体文件的文件名
         1. blk\_-8036292694616114566，存储blokc的实际数据
         2. blk\_-8036292694616114566\_1014.meta，存储元数据信息，包括版本号和checksum。其中的1014为file的generation stamp，全局唯一，由NN生成，每个文件1个，所以该文件的所有block的generation stamp相同
   5. AsyncBlockReport负责定期的扫描磁盘上的block信息，因为不加任何锁，可能出现同步问题，所有这里获得的只是粗略的block信息
   6. FSDatasetAsyncDiskService维护了多个线程池（每个FSVolume对应一个），负责异步的执行一些磁盘操作（其实就是使用线程池运行Runnable），典型的用法为删除block的异步执行。
4. DataBlockScanner维护DN上所有的block：
   1. 内部有两个数据结构，存储了所有的block信息：
      1. TreeSet<BlockScanInfo> blockInfoSet;
      2. HashMap<Block, BlockScanInfo> blockMap;
   2. DataBlockScanner是一个Runnable，在DN上作为线程运行，run方法：
      1. 调用init初始化所有block信息
      2. 定期执行verifyFirstBlock方法，检查排在blockInfoSet第一位的的block的正确性
   3. init负责初始化：
      1. 调用FSDataset的getBlockReport方法查询所有的block：Block arr[]，然后随进排序
      2. 为每个block构建BlockScanInfo，并添加到blockInfoSet和blockMap中
      3. 并打开一个日志用于记录scan情况
5. DataXceiverServer负责处理block的传输，包括DN到DN，和client到DN。NN启动时作为一个线程启动。
   1. 这个server启动后，监听tcp端口，一旦接收到连接后，就托管给DataXceiver
   2. DataXceiver是一个Runnable，处理一个block的输入输出流，从输入流中读取操作符，根据操作符执行相应的动作：
      1. *OP\_READ\_BLOCK*: 读取readBlock(in);
      2. *OP\_WRITE\_BLOCK*: 写：writeBlock(in);
      3. *OP\_REPLACE\_BLOCK*: replaceBlock(in);
      4. *OP\_COPY\_BLOCK*：copyBlock(in);
      5. *OP\_BLOCK\_CHECKSUM*: getBlockChecksum(in);
      6. 1,2,5主要用于和client交互，3,4主要用DN之间的数据流动（balance）
   3. 发送数据使用BlockSender（用于读取本地数据），请求协议见下图，第一个图为客户端的读请求协议，第二个为DN的返回数据，block是以分包的形式发送出去。

读数据逻辑图：

* 1. 接收数据使用BlockReceiver（用于接受数据存储到本地），协议和读数据类似。更复杂的一点是，接受数据时，可能需要构建一个传输链（pipeline），如图：
     1. 传输链上每个上游DN都等待其下游DN的写完成回复，然后再向自己的上游DN返回。
     2. 这种机制对应了写副本的需求，当所有数据都写副本成功后才会向client返回成功的信号，只要有一个DN出错，就不能成功返回，参考DFSClient.java的错误处理processDatanodeError方法，出现错误后，会尝试调用DN的recoverBlock方法恢复错误，处理完错误后只要有1个副本写成功，就可以成功返回给客户端（配置项dfs.replication.min控制这个值，安全模式也是用这个值判断）。（详细分析见DataXceiver.doc）
     3. 每个DN上完成写block后，都会调用DN的notifyNamenodeReceivedBlock方法(大部分NN在writeBlock方法中调用，只有最上游的DN中才会在PacketResponder中调用)，这个方法不会立刻报告给NN，而是添加到receivedBlockList中（其实添加完成后，马上就会唤醒offerService，参考2.a.5）

## Client

1. 客户端操作dfs时，一般是先获取到DistributedFileSystem的实例（使用FileSystem的get方法获取），而DistributedFileSystem的大部分操作是封装了DFSClient
2. DFSClient负责处理客户端到hdfs连接的大部分细节。
   1. 构造函数中会初始化到NN的连接，RPC协议为ClientProtocol，ClientProtocol定义了所有客户端和NN交互的方法
   2. 读文件：
      1. 先调用DistributedFileSystem的open方法打开文件，实际是调用DFSClient的open方法打开一个DFSInputStream，然后包装成DFSDataInputStream返回给上层。
      2. DFSDataInputStream继承自FSDataInputStream（暴露给用户的最上层数据流，用户直接操作这个数据流，参考上面DN的读数据逻辑图），DFSDataInputStream的所有操作都是包装DFSInputStream
      3. DFSClient的open方法简单的新建一个DFSInputStream，向NN查询元数据，建立到DN的连接等底层细节，都在DFSInputStream中实现
      4. DFSInputStream继承自FSInputStream：
         1. 构造方法中，调用openInfo方法从NN获取文件的位置信息
         2. openInfo方法调用DFSClient的*callGetBlockLocations*方法从NN获取文件的LocatedBlocks信息（这里会根据配置项dfs.read.prefetch.size决定要预读取几个block的位置，默认是10个）并赋值给locatedBlocks
         3. 实际读取数据时，是调用DFSInputStream的read方法，在此之前，并没有真正的连接到DN，而只是从NN获取元数据。
            1. read方法先定位到要读取的偏移位置所在的DN，如果pos > blockEnd，即表示要读取的位置已经不在当前block内，需要重新获取新的block（blockEnd初始时为-1）：currentNode = blockSeekTo(pos);
            2. blockSeekTo方法负责查找当前应该连接到的DN并建立到DN的连接，获取数据流DataInputStream：

先调用getBlockAt获取当前读取位置所在的block，getBlockAt方法从locatedBlocks中查找之前从NN获取的block列表中是否包含了希望读取的位置，如果没有包含，则会再次调用*callGetBlockLocations*方法从NN获取，并把获取结果缓存到locatedBlocks中

然后调用chooseDataNode从选定的block中选择一个存储位置DN（因为从NN返回的block信息中的DN列表已经是排好序的，chooseDataNode只是简单的选择最前面的DN，并保证DN不在该客户端之前的黑名单deadNodes中）

先尝试从本地读取，即不需要使用tcp连接的情况：初始化blockReader = *getLocalBlockReader*该功能默认关闭，通过配置项dfs.client.read.shortcircuit控制

如果不从本地读取，则需要使用tcp连接：

建立到DN的socket连接NetUtils.*connect*(s, targetAddr, socketTimeout);

初始化blockReader = BlockReader.*newBlockReader*

其实最后都是使用blockReader进行数据的底层读取

* + - * 1. 然后调用readBuffer读取数据，实际是调用BlockReader的read和readChunk读取数据。
      1. BlockReader和DN的BlockSender对应，负责从socket读取数据，在内部把socket的流包装成DataOutputStream和DataInputStream。初始化时，会写连接的头部信息，参考上面的协议图。获取它继承自FSInputChecker，在底层支持支持checksum的检查。

参考下面的类图，读取HDFS时，用户可见的是FSDataInputStream包装的FSInputStream，但是实际使用的是DFSDataInputStream包装的DFSInputStream。然后和DN实际交互的是BlockReader。



* 1. 写文件：
     1. 先调用DistributedFileSystem的create方法打开文件（实际是调用DFSClient的create方法打开一个DFSOutputStream，然后包装成FSDataOutputStream返回给上层）。获取到输出流后，可以开始写数据，写数据在最底层，其实是调用DFSOutputStream的writeChunk方法
     2. FSDataOutputStream（暴露给用户的最上层数据流，用户直接操作这个数据流，参考上面DN的读数据逻辑图）的所有操作都是包装DFSOutputStream
     3. DFSClient的create方法简单的新建一个DFSOutputStream
     4. DFSOutputStream继承自FSOutputSummer，在底层支持checksum。
        1. 构造函数中：
           1. 初始化DataChecksum
           2. 调用NN的create方法新建指定的文件的元数据信息
           3. 最后启动DataStreamer：streamer.start();
        2. DFSOutputStream和DataStreamer配合，DFSOutputStream负责写数据到队列，DataStreamer负责从队列中取packet，发送到DN，并监控DN的返回（通过dataQueue和ackQueue两个队列进行交互）
        3. DFSOutputStream的writeChunk方法负责写packet到队列LinkedList<Packet> dataQueue中：
           1. currentPacket指向当前正在写的packet，每次调用writeChunk方法是都先向currentPacket中写checksum，再写数据（每次写一个chunk，chunk的大小为512字节）
           2. 当currentPacket被chunk填充满时，就调用enqueueCurrentPacket方法把currentPacket填充到dataQueue中
        4. DataStreamer是个线程，一直等待dataQueue中被填充
           1. 一旦dataQueue有数据，则马上取队列中的第一个packet
           2. 如果需要建立到DN的连接，则调用nextBlockOutputStream获取到DN连接（新建文件，或者需要新的block时，都会调用这个方法），并新建一个这个pipeline的ResponseProcessor，然后启动之，等待返回信息

nextBlockOutputStream方法先调用locateFollowingBlock从NN获取应该连接到的DN列表（pipeline），然后调用createBlockOutputStream简历到pipeline中第一个DN的数据流

locateFollowingBlock调用NN的addBlock方法获取一个LocatedBlock（包含了用于写block到DN pipeline的所有信息）

createBlockOutputStream负责建立到pipeline的连接（协议参考DataXceiver的writeBlock方法）。最后获取的流赋值给blockStream

* + - * 1. 从dataQueue中移除第一个packet，添加到ackQueue的中队列尾，并把这个packet的数据，通过blockStream（即上面打开的到pipeline首部DN的流）写给pipeline
        2. 如果当前处理的packet是最后一个packet，则一直等待ackQueue
    1. DFSOutputStream写数据的方式：
       1. 数据被划分为packet，packet默认大小为64k
       2. packet又划分为chunk，每个chunk大小为512字节，每个chunk都对应一个checksum
       3. 数据发送方式：
          1. 当currentPacket被数据填充满，则被添加到dataQueue
          2. DataStreamer监听dataQueue，从dataQueue取出packet，发送到pipeline中的第一个DN，然后放入到ackQueue
          3. ResponseProcessor接收DN的ack，收到成功的ack后，从ackQueue中移除对应的packet
       4. 如果发生错误：
          1. 所有发出的packet都从ackQueue中删除，添加回dataQueue的首部
          2. 把坏点从原先的pipeline中移除，构成了新的pipeline，然后建立到主DN的rpc连接，调用DN的recoverBlock方法。
          3. DataStreamer重新发送dataQueue中的packet
          4. hdfs可以接受不满足副本要求的情况，只要能满足最低副本数，就能成功返回给client，且这个过程对client是透明的，参考上面DN的最后部分