

# Edits和Fsimage机制详解

2018年8月24日 19:01

## 概述

fsimage镜像文件包含了整个HDFS文件系统的所有目录和文件的inode（节点）信息，比如：`/park01/node`，会记录每个节点inodeid，以及节点之间父子路径。

以及文件名，文件大小，文件被切成几块，每个数据块描述信息、修改时间、访问时间等；此外还有对目录的修改时间、访问权限控制信息(目录所属用户，所在组等)等。

另外，edits文件主要是在NameNode已经启动情况下对HDFS进行的各种**更新操作**进行记录，比如：

`hadoop fs -mkdir` `hadoop fs -delete` `hadoop fs -put`等。

对于每次事务操作，都会用一个TXID来标识，`OP_MKDIR` `OP_DELETE`等。

总结：Edits文件存储的操作，而fsimage文件存储的是执行操作后，变化的状态。（元数据）

HDFS客户端执行所有的写操作都会被记录到edits文件中。

## 知识点

1.当执行格式化指令时候，会在指定元数据目录生成 `dfs/name/current/`

最开始只有fsimage，没有edits文件(因为没有启动HDFS)

2.当初次启动HDFS，会生成`edits_inprogress_00000000000000000001`，此文件用于记录事务（写操作）

3.HDFS对于每次写操作，都会用一个事务ID(TXID)来记录，TXID是递增的。

4.`edits_00000000000000000003-00000000000000000007`，数字表示的合并后起始的事务id和终止事务id

5.`seen_txid` 存储的当前的事务id，和`edits_inprogress`最后的数字一致

6.datanode存储块的目录路

径：/tmp/dfs/data/current/BP-859711469-192.168.150.137-1535216211704/current/finalized/subdir0/subdir0

7. **finalized**此目录存储的已经存储完毕的数据块，rbw目录存的是正在写但还未写完的数据块

### 查看Edits文件和Fsimage文件

```
hdfs oev -i edits_00000000000000000001-00000000000000000003 -o edits.xml
```

```
hdfs oiv -i fsimage_00000000000000000012 -o fsimage.xml -p XML
```

# HDFS 回收站机制

2016年1月18日 11:49

Hadoop回收站trash，默认是关闭的。

修改conf/core-site.xml, 增加

## 配置示例：

```
<property>
  <name>fs.trash.interval</name>
  <value>1440</value>
  <description>
    Number of minutes between trash checkpoints. If zero, the trash feature is disabled.
  </description>
</property>
```

注：value的时间单位是分钟，如果配置成0,表示不开启HDFS的回收站。

1440=24\*60,表示的一天的回收间隔，即文件在回收站存在一天后，被清空。

启动回收站后，比如我们删除一个文件：

```
[root@hadoop01 sbin]# hadoop fs -rm /word/1.txt
16/01/18 11:44:31 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
16/01/18 11:44:31 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval = 3 minutes, Emptier interval = 0 minutes.
Moved: 'hdfs://hadoop01:9000/word/1.txt' to trash at: hdfs://hadoop01:9000/user/root/.Trash/Current
```

我们可以通过递归查看指令，找到我们要恢复的文件放在回收站的哪个目录下

执行：hadoop fs -lsr /user/root/.Trash

```
[root@hadoop01 sbin]# hadoop fs -lsr /user/root/
lsr: DEPRECATED: Please use 'ls -R' instead.
16/01/18 11:46:39 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
drwx----- - root supergroup          0 2016-01-18 11:45 /user/root/.Trash
drwx----- - root supergroup          0 2016-01-18 11:44 /user/root/.Trash/160118114500
drwx----- - root supergroup          0 2016-11-16 06:15 /user/root/.Trash/160118114500/word
-rw-r--r--  3 ysq supergroup         49 2016-11-16 06:15 /user/root/.Trash/160118114500/word/1.txt
```

找到文件路径后，如果想恢复，执行hdfs的mv指令即可，（mv指令可用于文件的移动）

# HDFS API操作

## 实现步骤：

- 1.创建java工程
- 2.导入hadoop依赖jar包

## 连接namenode以及读取hdfs中指定文件

@Test

```
public void testConnectNamenode() throws Exception{

    Configuration conf=new Configuration();

    FileSystem fs=FileSystem.get(new URI("hdfs://192.168.234.21:9000"),
    conf);

    InputStream in=fs.open(new Path("/park/1.txt"));

    OutputStream out=new FileOutputStream("1.txt");

    IOUtils.copyBytes(in, out, conf);

}
```

## 上传文件到hdfs上

@Test

```
public void testPut() throws Exception{

    Configuration conf=new Configuration();

    conf.set("dfs.replication","1");

    FileSystem fs=FileSystem.get(new
    URI("hdfs://192.168.234.21:9000"), conf, "root");

    ByteArrayInputStream in=new ByteArrayInputStream("hello
```

```

        hdfs".getBytes());

        OutputStream out=fs.create(new Path("/park/2.txt"));

        IOUtils.copyBytes(in, out, conf);

    }

```

## 从hdfs上删除文件

```

@Test

    public void testDelete() throws Exception{

        Configuration conf=new Configuration();

        FileSystem fs=FileSystem.get(new

        URI("hdfs://192.168.234.21:9000"), conf, "root");

        //true表示无论目录是否为空，都删除掉。可以删除指定的文件

        fs.delete(new Path("/park01"), true);

        //false表示只能删除不为空的目录。

        fs.delete(new Path("/park01"), false);

        fs.close();

    }

```

## 在hdfs上创建文件夹

```

@Test

    public void testMkdir() throws Exception{

        Configuration conf=new Configuration();

        FileSystem fs=FileSystem.get(new

        URI("hdfs://192.168.234.21:9000"), conf, "root");

        fs.mkdirs(new Path("/park02"));
    }

```

```
}
```

## 查询hdfs指定目录下的文件

```
@Test
```

```
public void testLs() throws Exception{

    Configuration conf=new Configuration();

    FileSystem fs=FileSystem.get(new

    URI("hdfs://192.168.234.21:9000"), conf, "root");

    FileStatus[] ls=fs.listStatus(new Path("/"));

    for(FileStatus status:ls){

        System.out.println(status);

    }

}
```

## 递归查看指定目录下的文件

```
@Test
```

```
public void testLs() throws Exception{

    Configuration conf=new Configuration();

    FileSystem fs=FileSystem.get(new

    URI("hdfs://192.168.234.214:9000"), conf, "root");

    RemoteIterator<LocatedFileStatus> rt=fs.listFiles(new Path("/"), true);

    while(rt.hasNext()){

        System.out.println(rt.next());

    }

}
```

## 重命名

```
/*重命名

*/

@Test

public void testCreateNewFile() throws Exception{

    Configuration conf=new Configuration();

    FileSystem fs=FileSystem.get(new

    URI("hdfs://192.168.234.176:9000"),conf,"root");

    fs.rename(new Path("/park"), new Path("/park01"));

}
```

## 获取文件的块信息

```
@Test

public void testCopyFromLoaclFileSystem() throws Exception{

    Configuration conf=new Configuration();

    FileSystem fs=FileSystem.get(new

    URI("hdfs://192.168.234.176:9000"),conf,"root");

    BlockLocation[] data=fs.getFileBlockLocations(new

    Path("/park01/1.txt"), 0, Integer.MaxValue);

    for(BlockLocation bl:data){

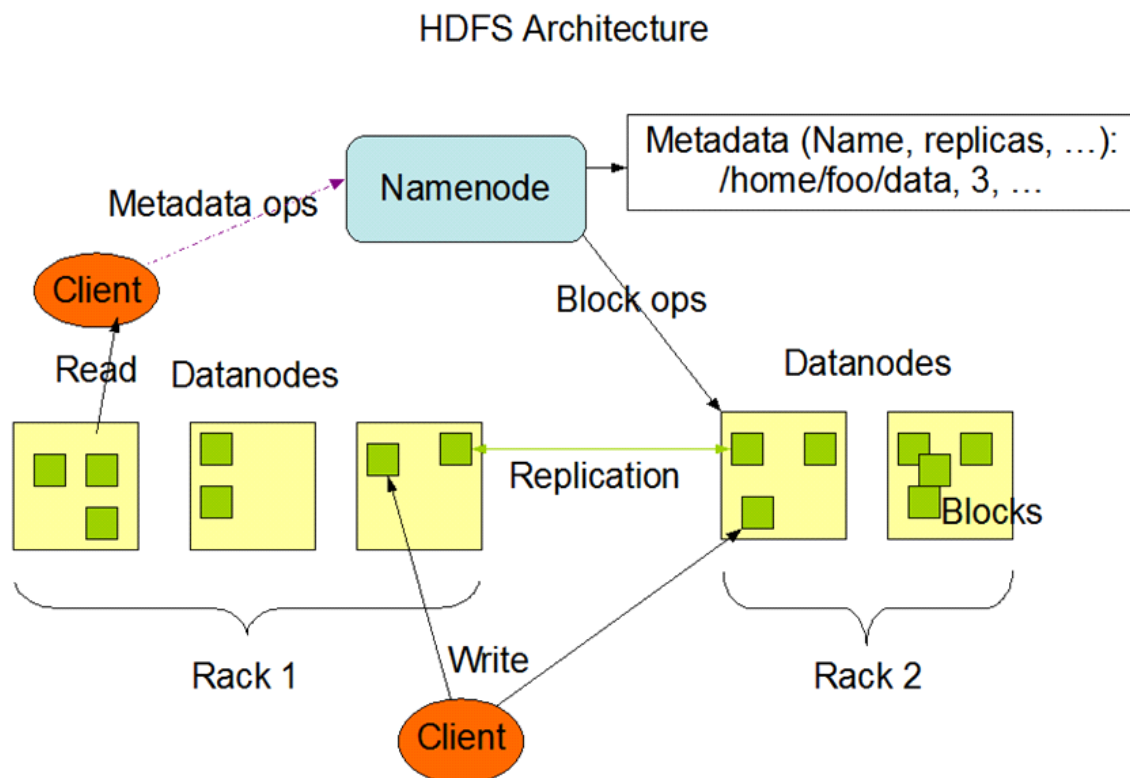
        System.out.println(bl);

    }

}
```

# HDFS各流程图

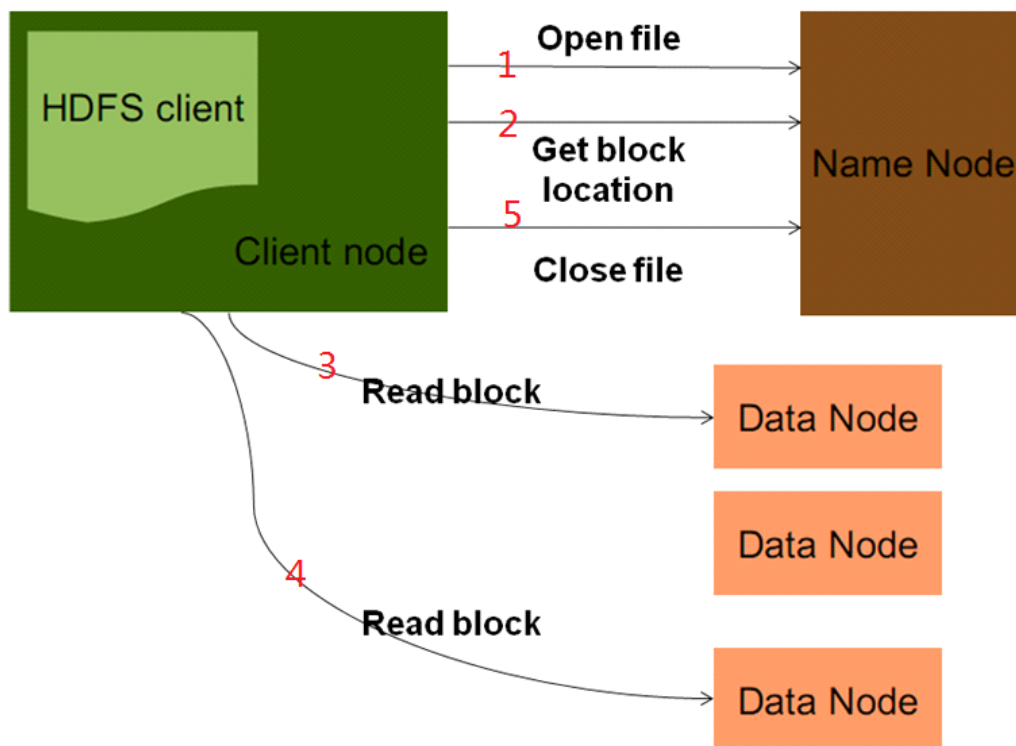
## HDFS架构图



- 1.namenode。名字节点，最主要的作用是管理元数据
- 2.Metadata 。元数据信息
- 3.文件块 Block
- 4.datanode。数据节点，用来存储文件块
- 5.Replication 文件块副本。一般采用的是3副本策略
- 6.Rack 机架
- 7.Client 客户端。凡是通过指令或代码操作的一端都属于客户端。

## 从HDFS下载文件过程





1.Client向namenode发起 Open file 请求。目的是获取指定文件的输入流。

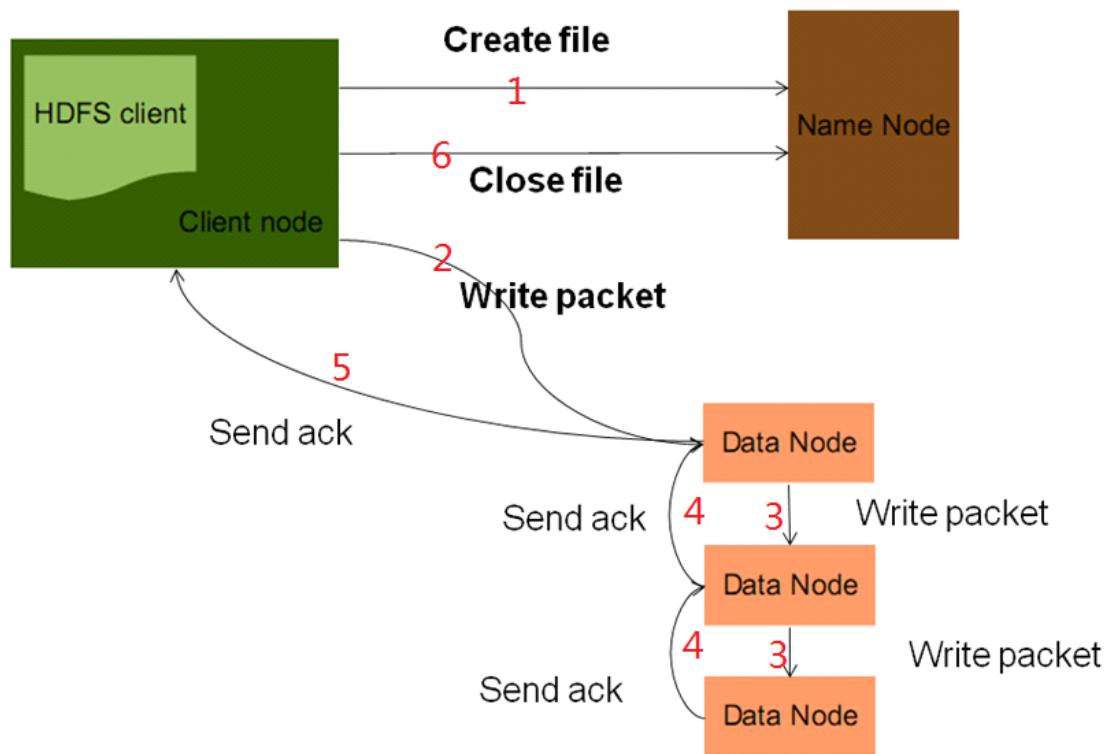
namenode收到请求之后，会检查路径的合法性，此外，还是检查客户端的操作权限。如果检测未通过，则直接报错返回。后续过程不会发生。

2.Client也会向namenode发起：Getblockloaction请求，获取指定文件的元数据信息。如果第一步的检测通过，namenode会将元数据信息封装到输入流里，返回给客户端。

3.4 客户端根据元数据信息，直接去对应的datanode读取文件块，然后下载到本地（创建本地的输出流，然后做流的对接）

5.读完后，关流。

[上传文件到HDFS](#)



1.Client向namenode发现 Create file请求，目的是获取HDFS文件的输出流。namenode收到请求后，会检测路径的合法性和权限。如果检测未通过，直接报错返回。

如果通过检测，namenode会将文件的切块信息（比如文件被切成几块，每个文件块的副本存在哪台datanode上），然后把这些信息封装到输出流里，返回给客户端。

**所以注意：**文件块的输出（上传）是客户端直接和对应DN交互的，namenode的作用是告诉Client文件块要发送给哪个datanode上。

2.Client通过输出流，发送文件块（底层会将一个文件块打散成一个一个的packet，每个packet的大小=64kb）。这个过程的机制，叫Pipeline（数据流管道机制）

这种机制的目的：

为了提高网络效率，我们采取了把数据流和控制流分开的措施。在控制流从客户机到主 Chunk、然后再到所有二级副本的同时，数据以管道的方式，顺序的沿着一个精心选择的 Chunk 服务器链推送。我们的目标是充分利用每台机器的带宽，避免网络瓶颈和高延时的连接，最小化推送所有数据的延时。

为了充分利用每台机器的带宽，数据沿着一个 Chunk 服务器链顺序的推送，而不是以其它拓扑形式分散推送（例如，树型拓扑结构）。线性推送模式下，每台机器所有的出口带宽都用于以最快的速度传输数据，而不是在多个接受者之间分配带宽。

3.4.5。通过数据流管道机制，实现数据的发送和副本的复制。每台datanode服务器收到数据之后，会向上游反馈ack确认机制。直到第五步的ack发送给Client之后，再发送下一个packet。依次循环，直到所有的数据都复制完毕。此外，在底层传输的过程中，会用到全双工通信。

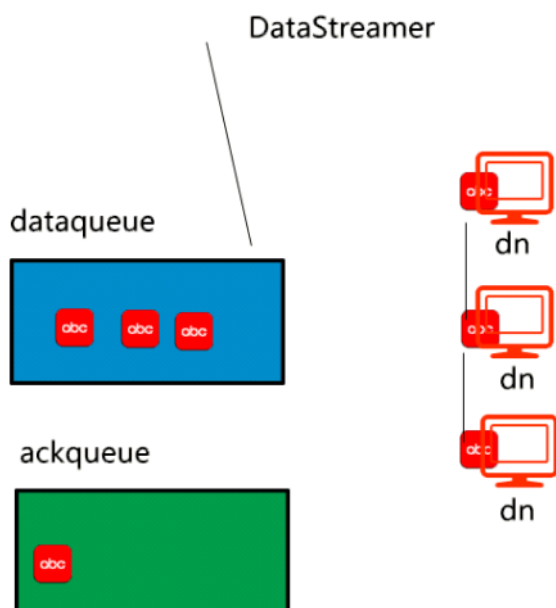
补充：建议看《Google File System》的3.2节

6.数据上传完之后，关流。

最后的补充：第一步Client获取的输出流，对应的类：DFSOutputStream。建议阅读源码注

释：

```
* DFSOutputStream creates files from a stream of bytes.  
*  
* The client application writes data that is cached internally by  
* this stream. Data is broken up into packets, each packet is  
* typically 64K in size. A packet comprises of chunks. Each chunk  
* is typically 512 bytes and has an associated checksum with it.  
*  
* When a client application fills up the currentPacket, it is  
* enqueued into dataQueue. The DataStreamer thread picks up  
* packets from the dataQueue, sends it to the first datanode in  
* the pipeline and moves it from the dataQueue to the ackQueue.  
* The ResponseProcessor receives acks from the datanodes. When an  
* successful ack for a packet is received from all datanodes, the  
* ResponseProcessor removes the corresponding packet from the  
* ackQueue.  
*  
* In case of error, all outstanding packets are moved from  
* ackQueue. A new pipeline is setup by eliminating the bad  
* datanode from the original pipeline. The DataStreamer now  
* starts sending packets from the dataQueue.
```



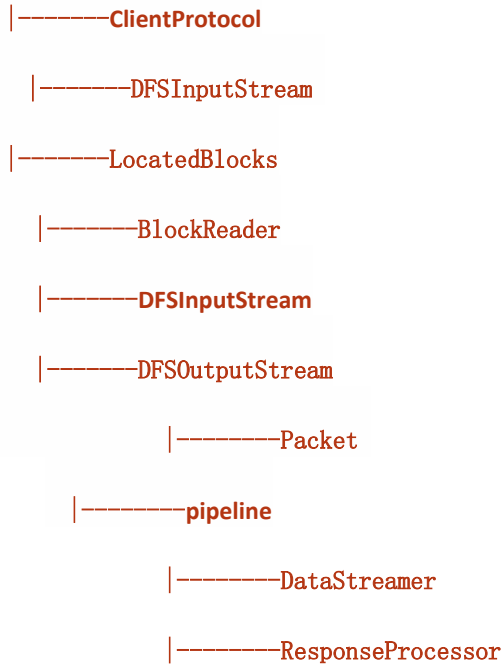
## 从HDFS删除文件的流程

- 1、客户端向namenode发现 删除文件指令，比如：`hadoop fs -rm /park01/1.txt`
- 2、namenode收到请求后，会检查路径的合法性以及权限
- 3、如果检测通过，会将对应的文件从元数据中删除。（注意，此时这个文件并没有真正从集群上被删除）
- 4、每台datanode会定期向namenode发送心跳，会领取删除的指令，找到对应的文件块，进行文件块的删除。

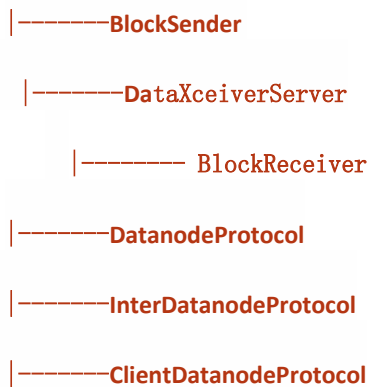
# HDFS相关源码剖析

DFSClient | Namenode | Datanode 源码分析顺序图：

## DFSClient



## Datanode



## Namenode



```
|-----HeartbeatThread
|-----Monitor
```

## 一、DFSClient相关体系

### DFSClient

```
|-----ClientProtocol
|-----DFSInputStream
|-----LocatedBlocks
|-----BlockReader
|-----DFSInputStream

|-----DFSOutputStream
|-----Packet
|-----pipeline
|-----DataStreamer
|-----ResponseProcessor
```



### DFSClient类的介绍源码：

- \* DFSClient can connect to a Hadoop Filesystem and
- \* perform basic file tasks. It uses the ClientProtocol
- \* to communicate with a NameNode daemon, and connects
- \* directly to DataNodes to read/write block data.

DFSClient这个类的作用是用于客户端连接Hadoop的HDFS文件系统，并在HDFS文件系统上做基本的文件操作（任务）。这个类和通过RPC机制和HDFS文件系统通信的，具体的RPC通信协议接口类是：

org.apache.hadoop.hdfs.protocol.ClientProtocol。

DFSClient通过RPC和namenode节点和datanode节点进行通信以及实现在datanode上执行读/写 block的操作

### 重要方法源码：

```
static LocatedBlocks callGetBlockLocations(ClientProtocol namenode,
                                           String src, long start, long length)
    throws IOException {
    try {
        return namenode.getBlockLocations(src, start, length);
    } catch (RemoteException re) {
        throw re.unwrapRemoteException(AccessControlException.class,
                                         FileNotFoundException.class,
                                         UnresolvedPathException.class);
    }
}
```

这个函数主要是通过跟namenode的交互，来完成从namenode取得用户请求的file的元数据信息

### ClientProtocol介绍

客户端和namenode进行rpc通信的协议接口，实现类是org.apache.hadoop.hdfs.protocol.ClientProtocol。

DFSClient通过RPC机制和NameNode通信并获得文件的元数据信息（数据的存放位置，校验和，等等一些关键的信息），然后DFSClient再与DataNode通信（集群中的其它机器）通过数据I/O流来获取到指定的文件信息。

### DFSInputStream、BlockReader 介绍

DFSInputStream是DFSClient用于读取datanode上文件的输入流。当DFSClient向namenode发送读文件请求之后，namenode会将此file的元数据信息返回：LocatedBlocks。这个类，封装了文件块的信息（每个文件块的大小、所在的datanode节点信息）。该输入流根据这些信息找到对应的DataNode，然后调用BlockReader类的read()方法，读取datanode节点上文件块里的内容。

## 重要变量源码：

```
public class DFSInputStream {  
  
    private final DFSClient dfsClient;  
  
    private BlockReader blockReader = null;  
  
    private LocatedBlocks locatedBlocks = null;  
  
    private DatanodeInfo currentNode = null;  
  
}
```

## DFSOutputStream、Packet、DataStreamer、ResponseProcessor介绍

DFSOutputStream类是一个输出流，用于提供客户端将本地文件（在客户端的电脑上存的文件）上传到HDFS上。当客户端发送一个

写请求后（create请求），比如上传一个大文件。首先会通过DFSClient去找namenode,namenode首先会创建对应的目录，然后返回给DFSClient输出流。拿到输出流之后，根据namenode返回的块信息blocksize，将数据变成64kb的packet存储到dataqueue队列里

DataStreamer会从dataQueue里取出一个一个的packet进行数据传输，然后形成一个数据流管道 pipeline。然后将数据流管道输出给管道里的第一个datanode节点。第一个datanode节点将数据流管道信息交给管道中的第二个节点。直到管道里最后一个datanode完成存储并返回ack确认后，通知response线程，然后DataStreamer发送下一个packet。

## DFSOutputStream里有2个队列和2个线程

### 两个队列相关代码：

dataQueue是数据队列，用于保存等待发送给datanode的数据包

```
private final LinkedList<Packet> dataQueue = new LinkedList<Packet>();
```

ackQueue是确认队列，保存还没有被datanode确认接收的数据包

```
private final LinkedList<Packet> ackQueue = new LinkedList<Packet>();
```

### 两个线程：

streamer线程，不停的从dataQueue中取出数据包，发送给datanode

```
private DataStreamer streamer = new DataStreamer();
```

response线程，用于接收从datanode返回的反馈信息



```
private ResponseProcessor response = null;
```

在向DFSOutputStream中，写入数据（通常是byte数组）的时候，实际的传输过程是：

- 1、文件数据以字节数组进行传输，每个byte[]被封装成64KB的Packet，然后扔进dataQueue中
- 2、DataStreamer线程不断的从dataQueue中取出Packet，通过socket发送给datanode（向blockStream写数据）  
发送前，将当前的Packet从dataQueue中移除，并addLast进ackQueue

- 3、ResponseProcessor线程从blockReplyStream中读出从datanode的反馈信息

反馈信息很简单，就是一个seqno，再加上每个datanode返回的标志（成功标志为

DataTransferProtocol.OP\_STATUS\_SUCCESS）

通过判断seqno（序列号，每个Packet有一个序列号），判断datanode是否接收到正确的包。

只有收到反馈包中的seqno与ackQueue.getFirst()的包seqno相同时，说明正确。否则可能出现了丢包的情况。

- 4、如果一切OK，则从ackQueue中移出：ackQueue.removeFirst(); 说明这个Packet被datanode成功接收了。



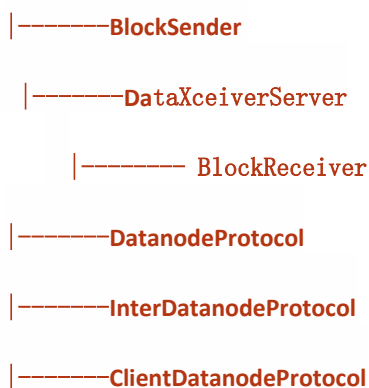
#### DataStreamer类重要源码：

```
private DataStreamer() {  
  
    isAppend = false;  
  
    stage = BlockConstructionStage.PIPELINE_SETUP_CREATE;  
  
}
```

在DataStreamer开始发送block时，创建pipeline数据流管道

## 二、Datanode相关体系

### Datanode



## BlockSender介绍

1. 当用户（客户端）向HDFS读取某一个文件时，客户端会根据数据所在的位置转向到具体的DataNode节点请求对应数据块的数据，此时DataNode节点会用BlockSender向该客户端发送数据；
2. 当NameNode节点发现某个Block的副本不足时，它会要求某一个存储了该Block的DataNode节点向其它DataNode节点复制该Block，当然此时仍然会采用流水线的复制方式，只不过数据来源变成了一个DataNode节点；
3. HDFS开了一个调节DataNode负载均衡的工具Balancer，当它发现某一个DataNode节点存储的Block过多时，就会让这个DataNode节点转移一部分Blocks到新添加到集群的DataNode节点或者存储负载轻的DataNode节点上；

**sh start-balancer.sh -t %10**

百分数是磁盘使用偏差率，一般调节的范围在10%~20%间。

4. DataNode会定期利用BlockSender来检查一个Block的数据是否损坏。

## DataXceiverServer介绍

datanode端是如何接受传来的数据文件呢？

在datanode类里，有一个线程类：

org.apache.hadoop.hdfs.server.datanode.DataXceiver。每当有client连接到datanode时，datanode会new一个DataXceiver，负责数据的传输工作。

在这个DataXceiver线程类里：



相关代码：

```
@Override
public void writeBlock ( ){

/** A class that receives a block and writes to its own disk, meanwhile
 * may copies it to another site. If a throttler is provided,
 * streaming throttling is also supported.
 */

BlockReceiver blockReceiver = null; // responsible for data handling
```

```
}
```

会调用 BlockReceiver 这个类，这个类是用于接收数据并写入本地磁盘，还负责将数据传输给管道里下一个datanode节点。

### DatanodeProtocol介绍

\* Protocol that a DFS datanode uses to communicate with the NameNode.

\* It's used to upload current load information and block reports.

这个类是datanode和namenode 实现RPC通信的接口协议，作用是datanode通过RPC机制连接namenode并汇报自身节点状态信息。在这个接口类里，一个非常重要的方法是：



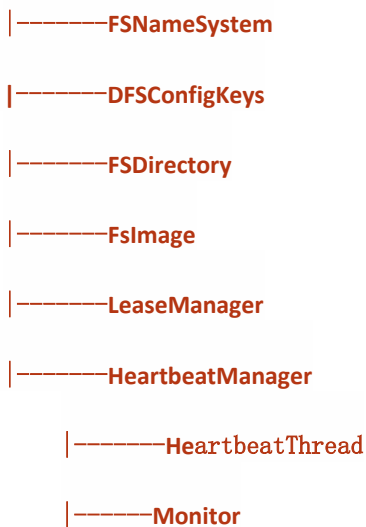
相关代码：

```
public HeartbeatResponse sendHeartbeat ( ) {  
  
}
```

这个是datanode向namenode发送心跳的方法，datanode周期性通过RPC调用sendHeartbeat向namenode汇报自身的状态。

## 三、NameNode相关体系

### Namenode



### Namenode介绍

- \* NameNode serves as both directory namespace manager and
- \* "inode table" for the Hadoop DFS. There is a single NameNode
- \* running in any DFS deployment. (Well, except when there
- \* is a second backup/failover NameNode, or when using federated NameNodes.)
- \*
- \* The NameNode controls two critical tables:
- \* 1) filename->blocksequence (namespace)
- \* 2) block->machinelist ("inodes")
- \*
- \* The first table is stored on disk and is very precious.
- \* The second table is rebuilt every time the NameNode comes up.
- \*
- \* 'NameNode' refers to both this class as well as the 'NameNode server'.
- \* The 'FSNamesystem' class actually performs most of the filesystem
- \* management.

Namenode类是Hadoop DFS文件系统的管理者类，用来管理HDFS的元数据信息。实际控制的是两张表信息，分别是：

1.文件——文件块信息

2.文件块信息——存储这个文件块的机器列表信息

第一张表信息存储在namenode节点的磁盘上，并且访问是非常高效的（因为namenode在启动之后，会将数据加载到内存里供快速访问）

第二张表信息，在每次namenode重启工作后，会重新建立。（这么做目的是为了确保块信息存储的准备性，实现机制时，当namenode重启工作后，每个datanode节点通过rpc心跳向namenode汇报自身存储的文件块信息，然后namenode根据这些信息，重建第二张表信息，在此过程中，HDFS是处于安全模式的，即只能对外提供读服务。当第二表信息重建完后，确认文件块数量正确且都完整，则退出安全模式）

实际上，namenode更多的是充当一个领导角色或更像是一个协议类，它定义了需要做哪些事，而真正干活的是FSNamesystem这个类。

在这个类里，format()这个方法我们应该会很熟悉，这是namenode的格式化方法

### format()方法：

```
/** Format a new filesystem.  Destroys any filesystem that may already
 * exist at this location.  */
public static void format(Configuration conf) throws IOException {
    format(conf, true, true);
}
```

格式化方法的定义：生成一个全新的文件系统（DFS文件系统）。并且摧毁已经存在的旧数据信息。

### format()方法源码骨架：

```
private static boolean format(Configuration conf, boolean force boolean isInteractive) throws IOException {
    //调用FsNamesystem,获取用户配置的元数据信息存放路径。如果不配置，则默认使用linux的/tmp/hadoop/dfs/name这个目
    录。
    //但是这个目录是临时目录，非常危险，所以需要更改。

    Collection<URI> nameDirsToFormat = FSNamesystem.getNamespaceDirs(conf);

    List<URI> editDirsToFormat = FSNamesystem.getNamespaceEditsDirs(conf);

    //然后调用FSImage类，在对应的目录下，创建新的Fsimage文件和Edits文件

    FSImage fsImage = new FSImage(conf, nameDirsToFormat, editDirsToFormat);

    fsImage.format(fsn, clusterId);
}
```

## FSNameSystem介绍

FSNameSystem是HDFS文件系统实际执行的核心，提供各种对文件的管理和操作

## FSdirectory介绍

FSDirectory存储整个文件系统的**目录状态**，保存了文件路径和数据块的映射关系。

### INodeFile和INodeDirectory

分别表示文件节点和目录节点。

### FSimage介绍

把文件和目录的元数据信息持久化地存储到fsimage文件中，每次启动时从中将元数据加载到内存中构建目录结构树，之后的操作记录在edits 中

定期将edits与fsimage合并刷到fsimage中

loadFSImage(File curFile)用于从fsimage中读入Namenode持久化的信息。

### HeartbeatManager介绍

- \* Manage the heartbeats received from datanodes.
- \* The datanode list and statistics are synchronized
- \* by the heartbeat manager lock.

namenode通过这个类来管理各个datanode传来的心跳。具体工作的线程是 HeartbeatThread这个线程类

### Monitor介绍

这是HeartbeatManager类里一个私有线程类，这个类的作用是周期性检测各个Datanode的心跳，


在这个线程类里run方法里，有一个方法比较重要，就是 heartbeatCheck() ;这个方法的作用：

- \* Check if there are any expired heartbeats, and if so,
  - \* whether any blocks have to be re-replicated.
- \* While removing dead datanodes, make sure that only one datanode is marked
- \* dead at a time within the synchronized section. Otherwise, a cascading
- \* effect causes more datanodes to be declared dead.

检查是否有超时的datanode心跳。如果有的话，先判断下在这个死亡节点上是否有数据块需要进行复制。然后对这个

datanode进行死亡标记。如果这个节点上有文件需要复制备份，则进行数据备份（满足一个block在HDFS上存3份），复制完后然后再删除这个死掉的datanode节点。从slaves列表中移除

此外，在进行死亡节点数据复制过程中，HDFS对外**不提供写服务**，即客户端此时是不能上传文件到HDFS系统上的。直到数据复制完成（比如一个block达到3份），最后删除此死亡节点后，才对外提供写服务。此过程中，不影响HDFS的查询文件操作

 相关代码：

```
private class Monitor implements Runnable {

    @Override

    public void run() {

        heartbeatCheck();

    }

}
```

# HDFS的租约机制

2018年8月25日 15:34

HDFS的有个内部机制：不允许客户端的并行写。指的是同一时刻内，不允许多个客户端向一个HDFS上写数据。

所以要实现以上的机制，实现思路就是用互斥锁，但是如果底层要是用简单的互斥锁，可能有与网络问题，造成客户端不释放锁，而造成死锁。所以Hadoop为了避免这种情况产生，引入租约机制。

租约锁本质上就是一个带有租期的互斥锁。

Hadoop的思想来自于Google的论文，3.1

Hadoop 租约锁对应的类：`org.apache.hadoop.hdfs.server.namenode.LeaseManager.Lease`

还有一个租约锁管理者：

`org.apache.hadoop.hdfs.server.namenode.LeaseManager`

建议掌握租约这种思想



# HDFS特点总结

## HDFS特点

- 1、分布式存储架构，支持海量数据存储。（GB、TB、PB级别数据）
- 2、高容错性，数据块拥有多个副本（副本冗余机制）。副本丢失后，自动恢复。
- 3、低成本部署，Hadoop可构建在廉价的服务器上。
- 4、能够检测和快速应对硬件故障，通过RPC心跳机制来实现。
- 5、简化的一致性模型，这里指的是用户在使用HDFS时，所有关于文件相关的操作，比如文件切块、块的复制、块的存储等细节并不需要去关注，所有的工作都已被框架封装完毕。用户所需要的做的仅仅是将数据上传到HDFS。这大大简化了分布式文件存储操作的难度和管理的复杂度。
- 6、**HDFS不能做到低延迟的数据访问（毫秒级内给出响应）**。但是Hadoop的优势在于它的高吞吐率（吞吐率指的是：单位时间内产生的数据流）。可以说HDFS的设计是牺牲了低延迟的数据访问，而获取的是数据的高吞吐率。如果要想获取低延迟的数据访问，可以通过Hbase框架来实现。
- 7、HDFS不许修改数据，所以适用场景是：一次写入，多次读取（once-write-many-read）。注意：HDFS允许追加数据，但不允许修改数据。追加和修改的意义是不同的。
- 8、HDFS不支持并发写入，一个文件同一个时间只能有一个写入者。
- 9、HDFS不适合存储海量小文件，因为会浪费namenode服务节点的**内存空间**

# 纠删码技术 ( Erasure Codes )

2018年8月25日 16:18

## 概述

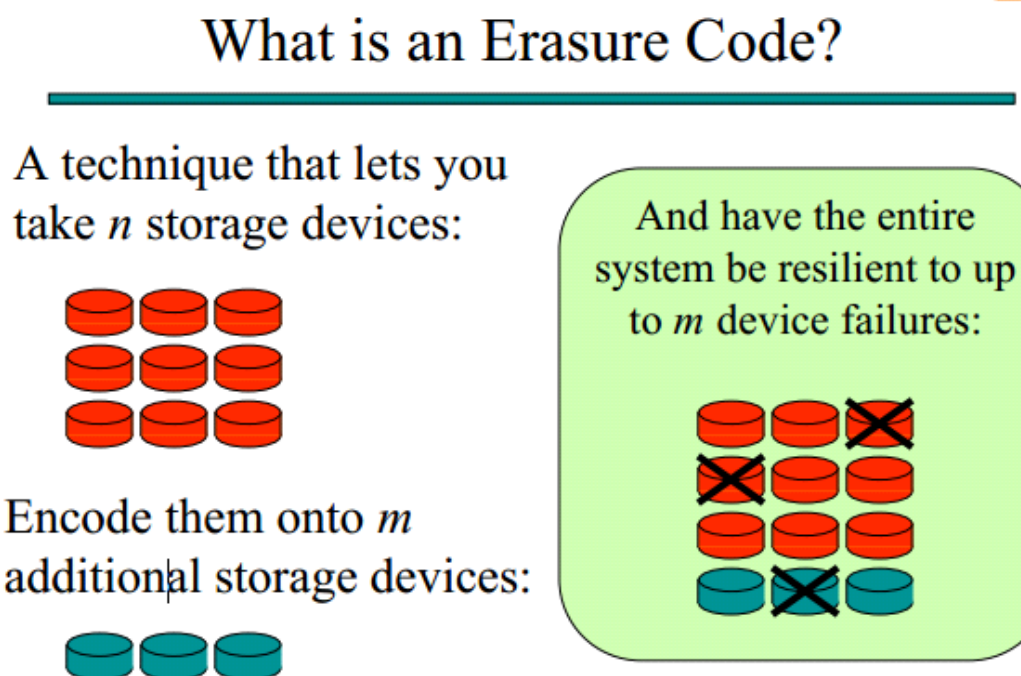
纠删码技术是一种数据恢复技术，使用这种技术可以提高磁盘的利用率。

在存储系统领域，实现数据存储的可靠性的途径：

①副本冗余机制，比如HDFS的3副本策略，磁盘利用率是1/3

②使用纠删码技术，引入数据校验块，通过编码过程让数据校验块和原始数据产生关联关系。则当数据块丢失时，可以进行恢复。

下图，示意使用纠删码，磁盘利用率：9/12=75%



三个原始数据块

$X=1$

$Y=2$

$Z=3$

三个数据校验块

$X+Y+Z=6$

$2X+3Y+Z=11$

$x+2Y+3Z=14$

# 里所码 ( Reed-Solomon )

2018年8月25日 16:31

## 概述

里所码算法分两个过程：

①编码过程（将原始数据块和数据校验块编码）

②解码过程（数据恢复）

具体看论文

优点：可以提供磁盘利用率，理论上来说，最大接近： $n/n+m \sim 100\%$

磁盘利用率越高，可靠性越低。

缺点：

- 1.会产生大量的计算量，占用很大带宽
- 2.数据恢复的代价高，因为需要计算，所以恢复效率较低
- 3.如果数据块发生了变化，会触发重新的编码。

如果使用纠删码（里所码）技术应用于存储系统，适用在冷数据的数据容灾。