Hadoop 分布式文件系统源代码详细解析

——主要介绍与Hadoop 1.0相比有重大变化的地方

# 1、基本元数据

下图是hadoop 2.0中HDFS中的基本元数据的类结构：

 与Hadoop 1.0相比，2.0中元数据中增加了DatanodeStorageInfo，ExtendedBlock，BlockCollection等类。Hadoop1.0中一个Datanode对应一个Storage，但是在Hadoop2.0中可以有多个Storage，Storage由dfs.datanode.data.dir来定义。

## 1.1 DatanodeStorageInfo

在Datanode中有一个或者多个Storage，在Datanode中Storage用DatanodeStorage及DatanodeStorageInfo来表示。DatanodeStorageInfo的成员变量：

*DatanodeDescriptor dn; 所属的datanode节点*

*String storageID;//每个datanode都有多个DataStorage，指定其ID号。*

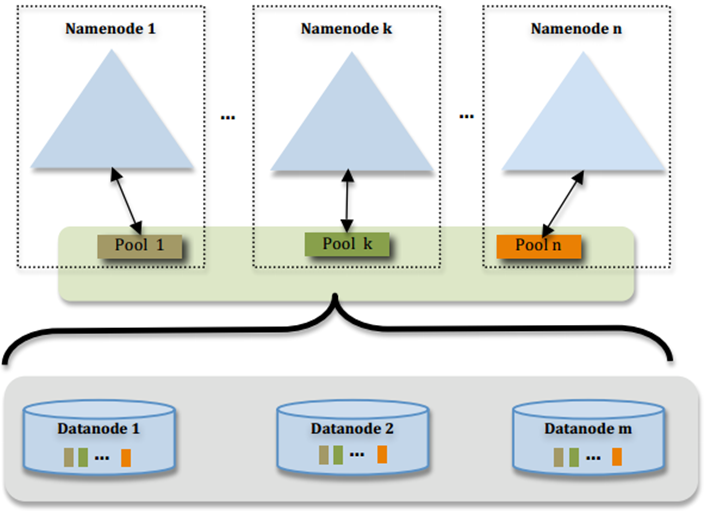
*StrorageType storageType;*

*State state;*

*BlockInfo blockList://该DataStorage内的Block链表*

## 1.2 ExtendedBlock

ExtendedBlock继承了Block类，增加了poolID的属性，Hadoop 2.0中增加了Namenode federation的运行模式，如下图所示：



采用这种模式，可以使HDFS的运行效率更高，文件的处理可以实现高并行化。在同一集群中Namenode共用datanode节点，但是Namenode只能管理本节点的数据，因此引入了Pool的概念。datanode在发送心跳信息时，namenode只能识别和处理自己pool内的数据。

## 1.3 BlockCollection

block Manager使用接口BlockCollection中提供的几个方法来对Block进行管理。InodeFile继承了该接口，在INodeFile的成员变量BlockInfo[] blocks用于保存文件的Blocks的信息。BlockCollection提供的方法：

*getLastBlock():return blocks[blocks.length-1]，返回blocks的最后一个Block对象*

*numBlocks()：返回InodeFile共有多个Block*

*getBlocks()：获取InodeFile的所有Blocks*

*其他的方法不再介绍*

## 1.4 InodesInPath

这个类用于根据指定的路径获取包含的Inodes信息，成员变量：

*Snapshot snapshot; 对于snapshot路径，要对该成员变量进行赋值*

*Inode[] inodes; 给定路径下的Inode*

核心方法：

*InodesInPath resolve(InodeDirectory startingDir,components,numOfInodes,resolvelink)*

*该方法用于从给定的InodeDirectory返回InodesInPath*

# 2、Namenode管理类

## 2.1 BlockManager(Block的管理)

BlockManager主要完成数据块和Dattanode的对应关系的管理，涉及的类包括Block,

BlockInfo和BlockMap（这几个类不再介绍）。

在hadoop0.20及hadoop1.0中对Block的操作都是直接对BlockMap等直接操作，但是在hadoop 2.0中药通过BlockManager来完成。

1) 成员变量（较多，只介绍重要的变量）

* *corruptReplicas:损坏的数据块副本集合，当datanode发现有损坏的副本时，上报给BlockManager，将损坏的副本加入这个数据结构当中。（存储损坏数据块副本blocks->*
* *datanodedescriptor的映射关系）*
* *excessReplicateMap:多余的副本集合*
* *invalidateBlocks:需要删除的Block集合，使用一个TreeMap存放StorageID（标识datanode）->invalidateBlocks的集合的映射。*
* *neededReplications:等待复制的副本*
* *pendingReplications:已经生成复制请求的数据块副本保存在pendingReplications*
* *postponedMisreplicatedBlocks:当namenode发生失败，进行active和standby切换，多余的副本不能删除，先放入这个数据结构当中。*
* *BlocksMap blocksMap；这是Namenode中的重要类，维护着block->blockInfo的关系*
* *DatanodeManager datanodeManager;*
* *HeartbeatManager heartbeatManager;*

## 2.2 DatanodeManager

DatanodeManager主要维护了在namenode当中注册的datanode信息，以及这些datanode在网络中的拓扑结构。DatanodeManager在FSnamesystem中的使用都是通过BlockManage来调用，DatanodeManager是blockManager的一个成员变量。以下是成员变量：

* *datanodeMap:维护了一个storageID->datanodeDescriptor的映射关系*
* *host2DatanodeMap:维护host->DatanodeDescriptor的映射关系，dataNode重新启动，是使用一个新的storageID，需要对datanodeMap进行更新。以 networktopology：维护整个网络的拓扑结构。以上几个成员变量在hadoop0.20是直接维护在FSnameSystem中。*
* *decommissionthread:一个线程了周期性的调用checkdeommissionState方法设置datanodeDescriptor的decommission状态。*

## 2.3 HeartBeatManager

用于负责track各个datanode心跳，以及设置对应状态。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。HearbeatManager当中有四个字段：

* *List<DatanodeDescriptor> datanodes;保存当前所有或者的datanode*
* *long heartbeatRecheckInterval;心跳检查间隔*
* *Stats stats;记录了所有数据节点的基本信息*
* *Daemon heartbeatThread;检查线程*

# 3、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，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，会非常消耗资源。因此namenode会将近期进行的操作保存在fseditlog当中。然后定期的合并fsimage以及fseditlog当中。FSImage的主要操作有：

* *保存namespace,将namenode内存当中的namespace保存到fsimage文件中*
* *加载fsimage，将磁盘上的fsimage文件保存的namespace加载到namenode内存当中*
* *合并fsediltlog以及fsimage*

## 3.1 保存fsimage

fsimage的保存使用了一个类FSImageFormat，这个类是读取和保存fsImage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，Loader-加载fsimage文件。

### 3.1.1 Saver类

三个比较重要的字段：

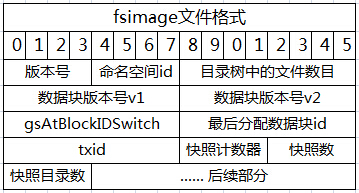
* *private final SaveNamespaceContext context;//上下文*
* *private boolean saved = false;//保存是否完成*
* *private MD4Hash savedDigest;// 文件已写部分的MD5校验值*

其中最重要的方法：

方法save(File newFile,FSImageCompression compression)：保存fsimage文件

1)获取写的IO流out,

2)写fsimage文件头，格式如下所示：



*out.writeInt(HdfsConstants.LAYOUT\_VERSION)*

*out.writeInt(sourceNameSystem.unprotectedGetNamespaceInfo.getNamespaceID());*

*......*

*out.writeLong(sourceNamesytem.getLastInodeId());*

*sourceNameSystem.getSnapshotManager().write(out)*

3) 写压缩信息

4)保存目录树根节点及其它的节点

saveINode2Image(fsDir.rootDir,out,false,feferenceMap,counter)

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

sourceNameSystem.saveFileUnderConstruction(out)

5)其他操作及设置文件的MD5校验

第4步中保存目录树根节点，saveInode2Image调用了FSImageSerialization.saveIno

de2Image()方法，其他节点的存储调用saveChildren()将目录树当中的所有节点写入文件（最用调用saveINode2Image），子节点是目录，则调用saveImage方法。

### 3.1.2 FSImageSerialization

这个类是静态类，用于将FSDirectory中的INode写入FSImage中。

saveINode2Image(INode node,DataOutputStream out):根据Inode的类型，进行不同的写入操作

writeINodeReference(node.asReference,out,writeUnderConstruction,referenceMap): 如果Inode是个引用，则调用这个方法

writeINodeDirectory(node.asDirecotry,out):将目录写入fsimage

writeINodeSymlink(node.asSymlink,out)

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

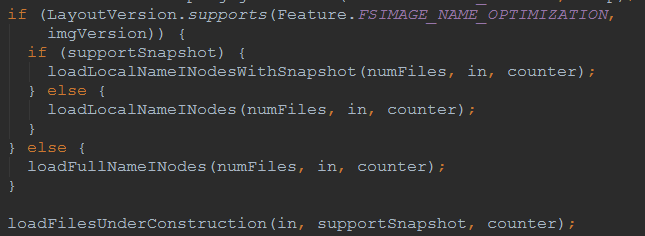
## 3.2 加载fsimage

从磁盘上读取fsimage文件，并将元数据加载到namenode的内存中，调用FSimage.loadFSimage(File,expectedMd2,FSnameSystem target,recovery)来完成操作。加载fsImage操作主要是在方法loadFSImage()当中，这个方法调用了FSImageFormat.Loa

der加载fsimage。加载fsimage的大部分操作都是在Loader.load()方法当中完成：

1) 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，包括imageversion,namespaceID,generation stamp等，根据上面定义的文件头的定义读取。

2) 加载inode，代码如下图：



loadLocalNameInodeWithSnapshot()方法先调用LoadRoot()加载根节点，然后调用方法loadDirectoryWithSnapshot()，迭代的加载其余Inode。

loadLocalNameInode(numFiles,DataInput in,Counter)方法，先加载根节点，然后调用loadDirectory(in,counter)加载其余INode。

3) 加载InondeUnderConstruction，调用方法loadFilesUnderConstruction()加载构建中的inode，首先从image文件中构造出InodeFileUnderConstruction，然后将namespace当中对应的Inode替换为inodeFileUnderConstruction，最后更新租约管理器当中的信息。

## 3.3 加载editlog

将editlog当中记录的操作同当前namespace进行合并，这里调用FSEditlogLoader.load()，基本归纳为从editlog当中读取一个操作，然后调用对应的FSDirectory进行更新。下面是FSImage.loadEdits(editStreams)源码：

*FSEditLogLoader loader = new FSEditLogLoader(namesystem);*

*// Load latest edits*

*for (EditLogInputStream editIn : editStreams) {*

*FLOG.info("Load Image: Reading edits: " + editIn + " last applied txid#: "*

*+ lastAppliedTxId);*

*numLoaded += loader.loadFSEdits(editIn, lastAppliedTxId);*

*lastAppliedTxId = loader.getLastAppliedTxId();*

*}*

*editLog.setLastWrittenTxId(lastAppliedTxId);*

*FLOG.info("Load Image: Number of edit transactions loaded: "*

*+ numLoaded + " last applied txid: " + lastAppliedTxId);*

*// update the counts*

*namesystem.dir.updateCountForINodeWithQuota();*

1) 初始化FSEditlogLoader

2) 获取最新的操作

3) 调用对应的FSdirectory方法进行更新

## 3.4 Image File visit

下面介绍Image File特殊处理的几个类，ImageLoader /ImageLoaderCurrent

及ImageLoaderVistor/ImageVisitor/TextWriterImageVisitor

### 3.4.1 ImageLoader

这个接口用于给定的ImageVisitor进行Hadoop FSImage文件的处理，在接口中定义了两个方法：

void loadImage(DataInputStream in,ImageVisitor v,boolean enumerateBlock)，参数in 是FSImage File的输入流，v是用在FSImage文件上的visit(访问)，enumerateBlock这个标记用于指示visitor是否要访问文件的每个blocks。

boolean canLoadVersion(int version): processor是否可以处理给定版本的FSImage文件。

在ImageLoader中定义了内部类LoaderFactory，用于返回ImageLoader实例，由于java接口中不能定义静态方法，所以采用了这个方式。

ImageLoader getLoader(int version)：由于在Hadoop2.0中只有ImageLoader

Current继承了这个接口，因此getLoader方法仅返回ImageLoaderCurrent实例。

### 3.4.2 ImageLoaderCurrent

这个类继承了ImageLoader接口，使用提供的ImageVisitor来处理Hadoop FSImage 文件，在以下列举的element 调用visitor。

重要的成员变量：

int[] versions; 如果verisons中没有相应的值，则不能被处理。

下面是实现ImageLoader接口的两个方法：

loadImage():

1) 调用ImageVisitor.start()及visitEnclosingElement(ImageElement.FS\_IAMGE)进行初始化工作

2)从fsimage文件中读取image文件对应的各字段值，并visit

### 3.4.3 ImageVisitor

在ImageVisitor定义了内部类，ImageElement(enum)定义了在FSImage可能会遇到的结构化元素，ImageVisitor对这些元素进行处理。

ImageVisitor是个接口类，主要的方法如下：

start(): 在访问fsimage 结构之前，进行初始化工作

finish()：访问后，进行任何必要的操作

visit(ImageElement,String/value/long):使用特定的值来访问no enclosing element

visitEnclosingElement(ImageElement):使用另外一个Elment来访问元素

其他方法不再介绍

### 3.4.4 TextWriterImageVistor/LsImageVisitor

这两个类是对接口ImageVisitor的实现，TextWriterImageVisitor实现接口ImageVisitor，并输出到text file中，在这个类中创建了DataoutputStream文件输出流。

核心的类是LsImageVisitor，这个类展示了namespace中block，类似于输出命令ls/lsr。类的成员变量如下： private int numBlocks;

*private String perms;*

*private String replication;*

*private String username;*

*private String group;*

*private long filesize;*

*private String modTime;*

*private String path;*

*private String linkTarget;*

*private String type;*

*private boolean inInode = false;*

核心方法：

*visit(ImageElement element,long value/String value):visit方法实质是对类的成员变量进行赋值，*

*Switch(Element)*

*case INODE\_PATH:*

*path = value;*

*case REPLICATION:*

*replication = value;*

.

但是switch并没有处理所有的element。如果element为INODE，则newline()

printLine():输出类的成员变量，调用out.write()

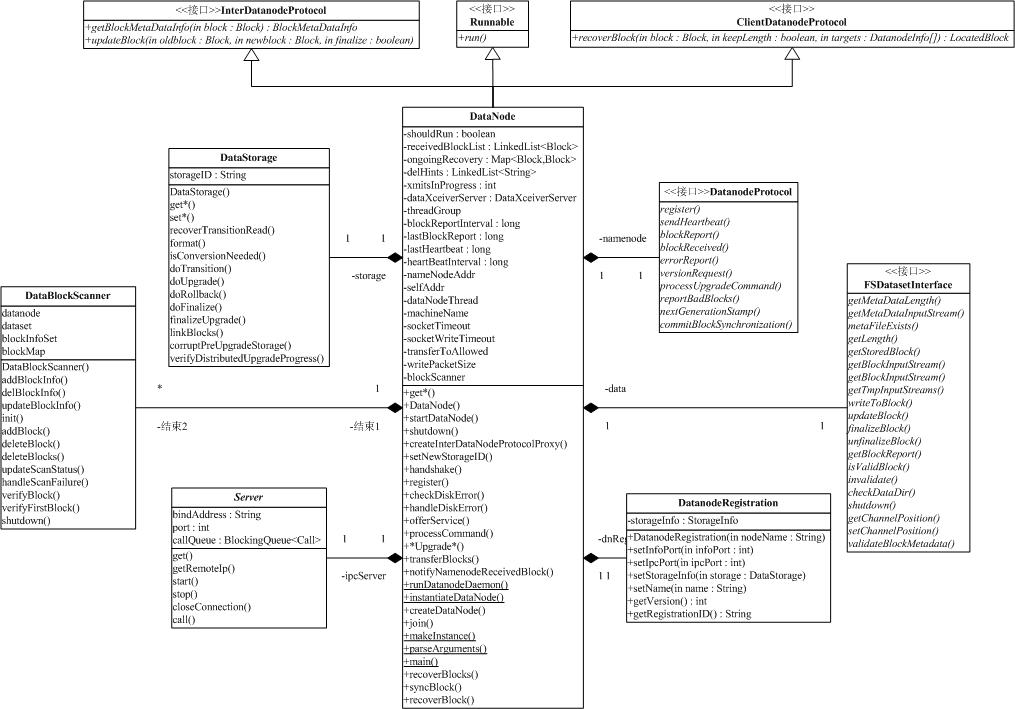
# 4、Datanode

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

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

## 4.1 DataNode类图

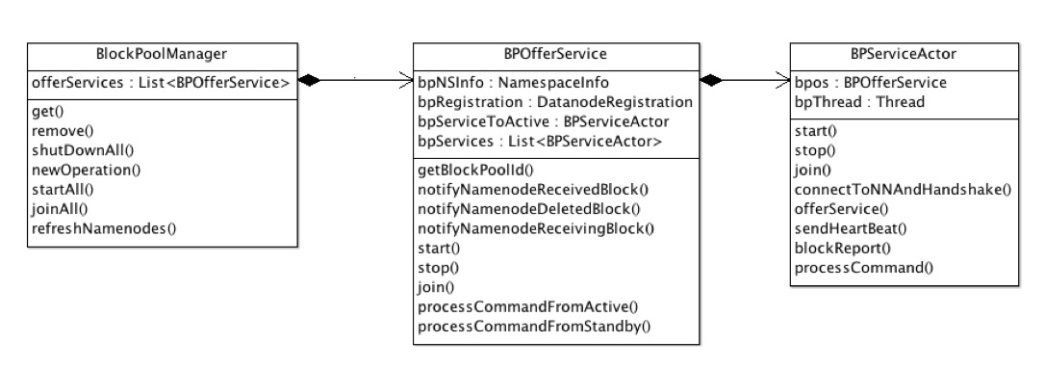
如下图所示：



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

### 成员变量BlockPoolManager,blockPoolManager

BlockPoolManager用于管理datanode中的BPOfferService实体，创建、移除、启动、停止及关闭BPOfferService。



BlockPoolManager: 对外提供管理BlockPool的API，其它模块对BlockPool的操作都通过它来完成，每个datanode中都有一个它的实例，对应namespaceManager。

BPOfferService: 封装了对单个Namespace下单个BlockPool相关的功能，由外部分线程驱动，每个Namespace下每个BlockPool都包含一个它的实例

BPServiceActor: 封装了单个BlockPool针对单台namenode的相关逻辑，包括connect()->handshake()->register()->heartbeat()->reportblock()一系列操作，自带线程驱动，每个BPServiceActor是一个单独的线程。

成员变量：

Map<String,BPOfferserice> bpByNameserviceId;

Map<String,BPOfferservice> bpByBlockPoolId;

List<BPOffserService> offerService; //每个Namenode对应一个BPOfferService，但是Datanode上有多个BPOfferService

DataNode dn;

BlockManager提供的操作，均是对这几个成员变量的管理

### BPOfferService

在DN上一个namespace或者block-pool对应一个BPOfferService。接收active及standby 的Namenode传送过来的心跳信息。

NamespaceInfo bnNSInfo; //该bpOffserService注册的namespace信息

DatanodeRegistration bpRegistration; 该blockpool的注册信息

Datanode dn; 所属的datanode

BPServiceActor ; //active或者standby namenode执行的线程

List<BPServiceActor> bpService; //这个nameservice中namenodes对应的所有actors。

### BPServiceActor

该线程用于和Namenode注册前的握手信息，注册到namenode，向namenode周期性发送注册信息及处理从namenode接收的命令

BPOfferService bpos; //相关的BPOfferService

long lastBlockReport,lastDeleteReport,resetBlockReportTime,lastCacheReport,

lastHeartbeat;

DatanodeProtocolClientSideTranslatorPB bpNamenode;

Datanode dn;

DatanodeRegistration bpRegistration;