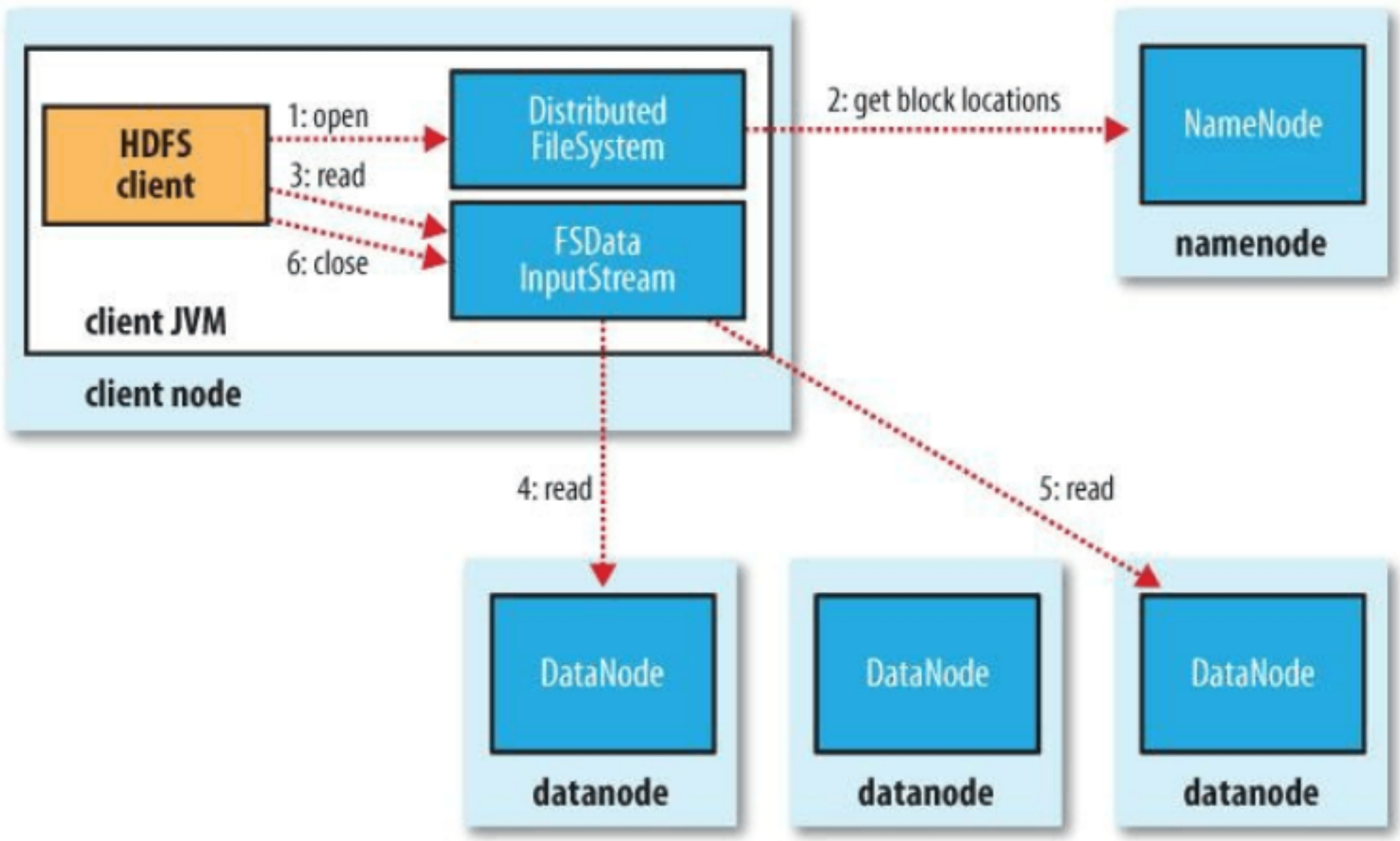
HDFS 读文件解析

下图描述了在文件读过程中，client、NameNode和DataNode三者之间是如何互动的。



1. client调用 get方法得到 HDFS文件系统的一个实例（DistributedFileSystem）。然后调用它的 open方法。

2. DistributedFileSystem通过 RPC远程调用 NameNode决定文件文件的block的位置信息。对于每一个 bolck，NameNode返回 block所在的DataNode（包括副本）的地址。 DistributedFileSystem 返回FSDataInputStream给 client用来读数据。 FSDataInputStream封装了DFSInputStream用于管理 NameNode和 DataNode的IO。

3. client调用 FSDataInputStream的read方法。

4. DFSInputStream保存了 block块所在的 DataNode的地址信息。DFSInputStream连接第一个 block的 DataNode，read block数据，传回给client。

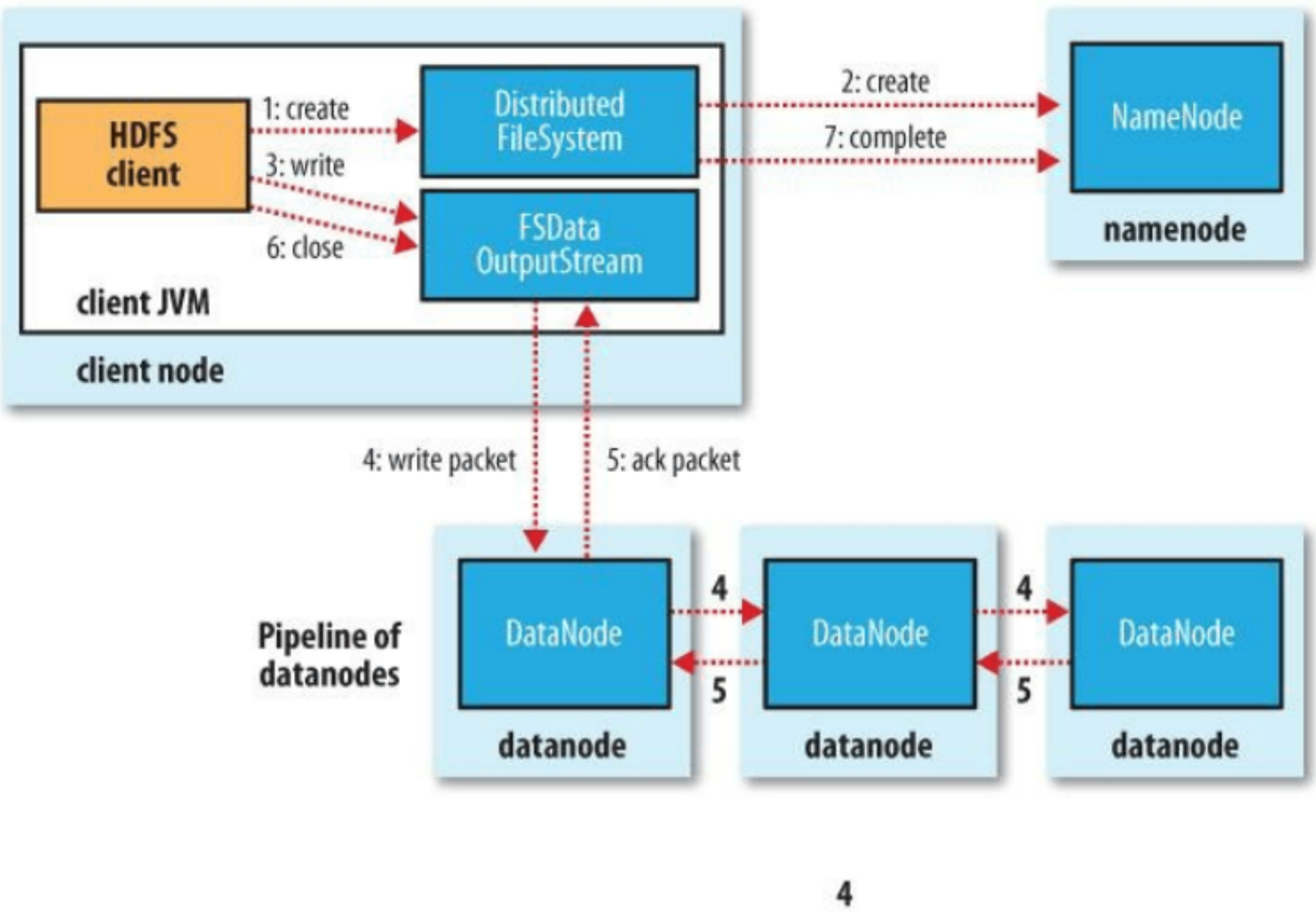
5. 当第一个 block 读完， DFSInputStream关掉与这个 DataNode的连接。然后开始第二个 block。

6. 当client 读结束，调用 FSDataInputStream的close方法。

在读的过程中，如果 client和一个 datanode通讯时出错，他会连接副本所在的 datanode。这种 client直接连接 datanode读取数据的设计方法使HDFS可以同时相应很多 client的同时并发。因为数据流量均匀的分布在所有的 datanode上， NameNode只负责 block的位置信息请求。

HDFS 写文件解析

我们看一下创建文件，写文件最后关闭文件的流程。如下图：



4

1. client通过调用 DistributedFileSystem 的create方法来创建文件。

2. DistributedFileSystem通过 RPC调用 NameNode在文　件系统的名字空间里创建一个文件，这个时候还没有任何　block 信息。DistributedFileSystem 返回 FSDataOutputStream给 client。FSDataOutputStream封装了一个 DFSOutputStream来处理与 datanodes和namenode之间的通讯。

3. 当client 写一个 block数据的时候，　DFSOutputStream把数据分成很多packet。FSDataOutputStream询问 namenode挑选存储这个 block以及它的副本的 datanode列表。这个 datanode列表组成了一个管道，在上图管道由三个 datanode组成（备份参数是 3），这三个 datanode的选择有一定的副本放置策略，详细请看下一篇。

4. FSDataOutputStream把packet写进管道的第一个 datanode，然后管道把 packet转发给第二个 datanode，这样一直转发到最后一个 datanode。

5. 只有当管道里所有 datanode都返回写入成功，这个 packet才算写成功，发送应答给 FSDataOutputStream。开始下一个 packet。

如果某个 datanode写失败了，会产生如下步骤，但是这些对　client是透

明的。

1) 管道关闭。

2) 正常的 datanode上正在写的 block会有一个新 ID （需要和 namenode通信）。这样失败的 datanode上的那个不完整的 block在上报心跳的时候会被删掉。

3) 失败的 datanode会被移出管道。 block中剩余的 packet继续写入管道的其他两个 datanode

4) namenode会标记这个 block的副本个数少于指定值。 block的副本会稍后在另一个 datanode创建。

5) 有些时候多个 datanode会失败。只要 dfs.replication.min(缺省是 1)个 datanode成功了，整个写入过程就算成功。缺少的副本会异步的恢复。

6. 当client完成了写所有 block的数据后，调用 FSDataOutputStream的close方法关闭文件。

7. FSDataOutputStream通知 namenode写文件结束。

HDFS 文件创建流程

文件夹的创建是一个相对简单的过程，主要是通过 FileSystem中的 mkdirs（）方法，这个方法在 DFSClient 实例中调用同名方法 mkdirs（），通过 Hadoop本身的 RPC机制调用 Namenode的 mkdirs（）方法，最终这个调用 PUSH到FSNameSystem的 mkdirsInternal 方法，这个方法主要就是检验访问权限，最后通过 FSDirectory 的unprotectedMkdir （）方法，构建一个 INodeDirectory 实例添加到文件系统的目录树中。

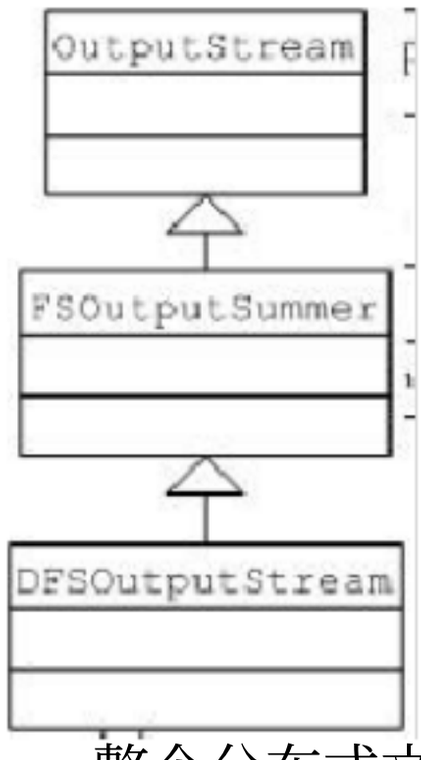
文件节点的创建与添加相对比较麻烦，主要步骤如下：

FileSystem的 create方法返回一个很重要的类 FSDataOutputStream，这一点也比较好理解，就像 java中的文件流一样，创建一个文件写入流对文件内容进行追加，首先我们看文件创建阶段 namenode主要做了什么事情（权限验证以及租约验证这些前面都已经有提到，下面的内容就会掠过这一部分）

DfSOutputStream在实例构建时，通过 Hadoop本身的 RPC机制调用 Namenode的 create方法，最终这个调用 PUSH到 FSNameSystem的 StartFileInternal 方法，需要做权限验证，租约检验等工作，这个方法主要作用就是创建一个 INodeFileUnderConstruction 实例（上面已经提过，文件写入过程中都会有一个 INodeFileUnderConstruction 与这个文件对应），这个实例最后通过 FSDirectoty 的addNode()方法添加到文件系统目录数中，这个时候文件创建操作就算完成了重要的第一步，文件系统中已经有了这个文件的记录。

下面就涉及到文件的写入操作（相当复杂的部分）

这个时候就需要用到返回的 DfSOutputStream对象。这部分太复杂了，我们先分析一些基本模块儿，逐步吃透这部分的实现。



整个分布式文件系统中网络通讯部分分为两类：

1.命令类调用（这部分通过 HADOOP 的RPC机制进行支持）

2.流式数据传输（这部分通过 HADOOP 的流式数据传输协议支持）

为了保证数据的正确性， hadoop在多个关键处理单元做了数据检验操作，在流式数据网络传输部分通过校验和保证数据传输正常。

Client 在DfSOutputStream 对象调用 write 方法时，系统并不会马上把数据写入 SOCKET中，而是逐个构建 Package并将这些 Package加入一个队列。

在DfSOutputStream 对象构建时，系统通过 Hadoop本身的 RPC机制 调用Namenode的create 方法后，会启动一个后台线程 streamer .start(); 这个线程的主要目的就是将上述的 package 队列写入SOCKET中。

右图为 DfSOutputStream 的继承关系

其中 FSOutputSummer这个类其实质是一个 decorator 设计模式的实现，主要的目的就是在 OutputStream 的 void write(byte b[], int off, int len) 方法中增加一些功能，上文已经提过，文件数据传输的同时，系统会在传输的数据中增加检验和数据，系统收到数据后对数据进行校验，保证数据传输的正确性 ，但是用户在对文件输出流进行操作的时候并不需要关注校验和数据，用户只需要不断的调用 write 方法在目标文件中追加数据。

注：我们通常可以使用继承来实现功能的拓展 ,如果这些需要拓展的功能的种类很繁多 ,那么势必生成很多子类 ,增加系统的复杂性 ,同时 ,使用继承实现功能拓展 ,我们必须可预见这些拓展功能 ,这些功能是编译时就确定了 ,是静态的 .使用 Decorator的理由是 :这些功能需要由用户动态决定加入的方式和时机 .Decorator提供了 "即插即用 "的方法 ,在运行期间决定何时增加何种功能 .

我们看一下 FSOutputSummer中write(byte b[], int off, int len) 的实现

public synchronized void write(byte b[], int off, int len) throws IOException

{

if (off < 0 || len < 0 || off > b.length - len) {

throw new ArrayIndexOutOfBoundsException();

}

for (int n=0;n<len;n+=write1(b, off+n, len-n)) { }

}

可以看到不断调用 write1方法，保证数据发送的完整性。

那么 write1方法又做了什么事情呢， write1将用户需要写入的数据流首先写到自己的 BUFFER中，达到一定数量（基本是一个 chunk的大小）后进行 CheckSum方法调用得到一段数据的校验和，然后通过 writeChecksumChunk 这个方法将数据以及该部分数据的校验和，按照一定格式要求一并写入 Stream。

writeChecksumChunk 这个方法的主要作用就是将用户写入的数据以及该部分数据的校验和做为参数调用 writeChunk () 方法，这个方法是一个虚方法，真正的实现在 DFSOutputStream 这个类中，这也合情合理，本身 FSOutputSummer 这个类的作用仅仅是在输出流中增加校验和数据，至于数据是如何进行传输的是通过 DFSOutputStream 来实现的。

那么接下来需要说明的就是 DFSOutputStream 的writeChunk 这个方法了。

HDFS流式数据网络传输的基本单位有哪些呢？

chunk->package->block

我们上文已经提过：等用户写入的数据达到一定数量（ 基本是一个chunk的大小）后就会对这段数据取校验和。一定数量的 chunk就会组成一个package, 这个package就是最终进行网络传输的基本单元, datanode收到package后， 将这些 package组合起来最终得到一个 block 。

我们接下来通过实际主要的代码了解这部分功能的实现：

currentPacket这个对象初始化的时候就是 null, 第一次写入数据时这个判断成立

if (currentPacket == null)

{

currentPacket = new Packet(packetSize, chunksPerPacket,

```
bytesCurBlock);
...

//下面开始构建 package包。
//在package包中增加一个 chunk, 首先添加这个 chunk所包含数据的checksum
currentPacket.writeChecksum(checksum, 0, cklen);
currentPacket.writeData(b, offset, len); //然后添加这个 chunk所包含的数据
currentPacket.numChunks++; //增加这个 package所包含的 chunk个数
bytesCurBlock += len; // 当前已经写入的 byte个数
// If packet is full, enqueue it for transmission
//如果这个 package已经达到一定的 chunk数量 ,准备实际的传输操作
if (currentPacket.numChunks == currentPacket.maxChunks || bytesCurBlock == blockSize)
{
.....
if (bytesCurBlock == blockSize) // 如果用户写入的数据, 已经达到一个block缺省大小（ 64M）
{
//设置当前的 package是某一个 block的最后一个 package
currentPacket.lastPacketInBlock = true;
//清除一些变量的值
bytesCurBlock = 0;
lastFlushOffset = -1;
}
//这三段代码是关键的一部分代码, 将已经构建完成的 package写入一个 dataQueue队列, 由另一个线程（就是我们开始提到的：启动一个后台线程 streamer.start();这个线程的主要目的就是将上述的 package队列写入 SOCKET中）从该队列中不断取出 package, 进行实际的网络传输
dataQueue.addLast(currentPacket); //产生 event, 进而通知并唤醒等待线程
dataQueue.notifyAll(); // 这一步也很重要 ,设置 currentPacket为空, 表示这个 package已经满了, 需要 new一个新的 package继续接收用户后面进一步需要写入的数据。
currentPacket = null;
// If this was the first write after reopening a file, then
```

```
        // the above write filled up any partial chunk. Tell the summer to generate
full
        // crc chunks from now on.
        if (appendChunk)
        {
        appendChunk = false;
        resetChecksumChunk(bytesPerChecksum);
        }
        int psize = Math.min((int) (blockSize - bytesCurBlock),writePacketSize);
        computePacketChunkSize(psize, bytesPerChecksum);
        }
```

computePacketChunkSize这个方法的主要作用是计算两个参数：

1.chunksPerPacket

接下来的 package需要承载多少个 chunk;因为最后一个 package承载的chunk个数与文件大小也有关系。

2.packetSize

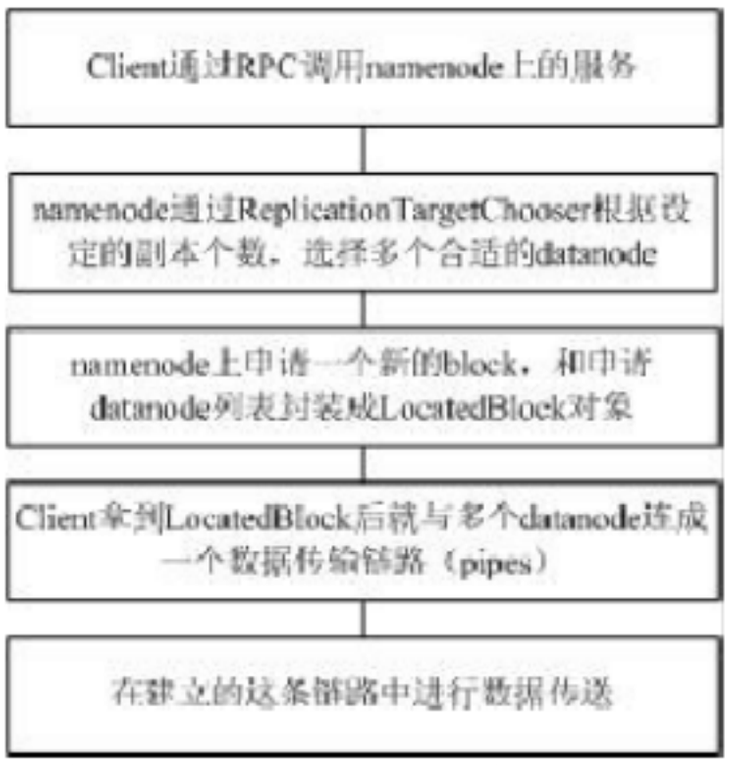
接下来的 package的大小。

以上两个参数与判断是否需要 new一个新的 PACKAGE 很有关系。

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

可以看到构建的 package不断添加到 dataQueue这个队列，streamer.start()这个线程从中弹出 package进行实际网络传输操作。

下面就涉及到比较复杂的网络传输协议部分。



我们先看一下这部分的流程：

1.上面已经讲过 ,开始的一步就是客户端调用 create方法，在 namenode上的目录树中注册一个 INodeFileUnderConstruction 节点 ,并得到一个DfSOutputStream。

2.用户得到这个 outputStream后就可以进行写入操作，用户写入的数据就不断构建成 package写入 dataQueue这个队列。

3.streamer.start()这个线程从 dataQueue队列中取出 package进行实际网络传输操作。

下面的网络传输流程为关键流程：

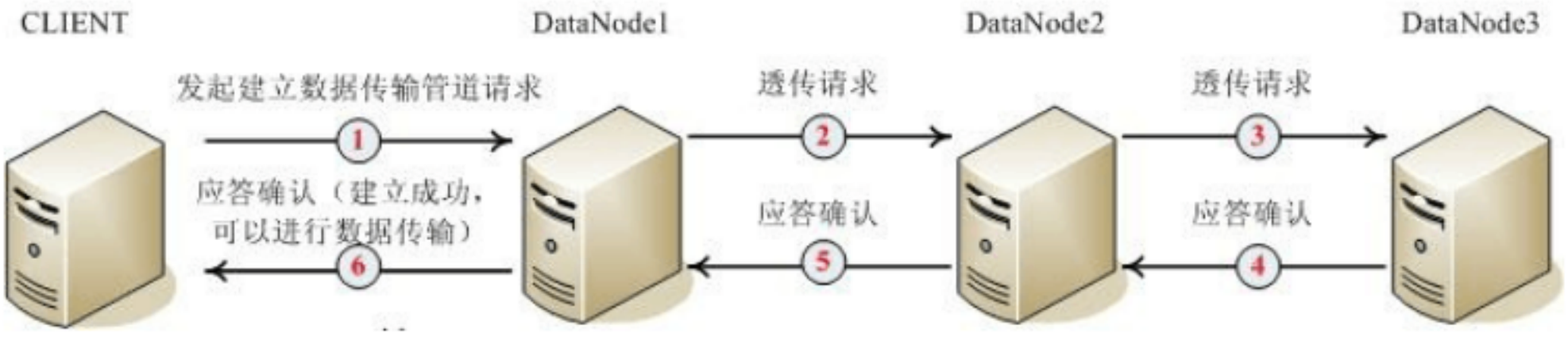
4. streamer是一个 DataStreamer的实例，这是一个线程实例。大家知道HDFS中的文件数据会分成很多 64M 大小的 block,所以在 HDFS中保存文件数据第一步就是在 namenode上申请一个特殊的 blockID （当然还是通过 RPC调用的方式）。右图为 文件写入流程

4. HDFS 文件数据流写入传输协议

FileSystem是一个提供给用户对文件系统进行访问的抽象类，访问HDFS的具体实现类为 DistributedFileSystem,用户对 HDFS的所有操作就是通过这个类的具体实例完成的。在HDFS写入文件之前需要 首先在Namenode创建并注册一个 INodeFileUnderConstruction，在向文件进行数据追加时，会针对这一文件逐个向系统中追加 BLOCK，如果是第一次写入（或者上一个 BLOCK 已经写满），需要在 Namenode中注册一个新的BLOCK， 通过 RPC调用 Namenode的具体实例返回 LocatedBlock 对象，除了新申请的 block这个 LocatedBlock 对象还包含多个 datanodeinfo信息，表示这个 block块儿需要传输的哪些 datanode上进行保存。

下面我们详细看一下数据传输的具体流程以及传输协议的详细内容：

第一步将返回的多个 datanode逐个建立链接形成数据传输链，如下图所示（以三份数据拷贝为例）：



当流程到达第 6步的时候这个管道才算建立成功，每一步都是与下一个节点建立链接，发送管道创建协议包。

管道建立协议：

4字节 数据传输协议版本号（ 0.19.1版本的 hadoop这个字段为14）

4字节 操作码（追加文件时为： OP_WRITE_BLOCK ）

8字节 Block的BLOCKID 字段

8字节 Block的时间戳字段

4字节 建立的传输链中一共有多少台 datanode

Boolean 表示是否为恢复数据请求（ recoverblock操作）

Text Client名称字串，格式为字串长度（这个采取了压缩形式） +字串内容

Boolean 表示是否传输客户端信息

DatanodeInfo 如果上面字段为 true，才会有该字段，表示 client信

息

Int 表示数据链中后续节点个数（每过一个节点，这个字段就减少一）

DatanodeInfo[] 表示数据链中后续节点信息，上面字节是多少就读多少个

DatanodeInfo信息，传给后面的节点做下一节点的链接

Checksum byte Checksum的类型，可以根据该字段实例化具体checksum类

Int 对多少字节进行取 checksum操作

Client发出上面的请求后就等待应答，应答的协议格式如下：

Text Client名称字串，格式为字串长度（这个采取了压缩形式）+字串内容

返回就是一个字串信息，如果字串为空表示后续节点所有链路都链接成功，如果不为空该字段保存的就是出错的 datanode信息，格式如下：name:port。

针对于一个 block传输操作创建数据链路成功以后，就可以进行实际数据传输了。后面的数据就是一个接一个的 Package包。

4字节 Package数据长度

4字节 package在这个 block中的偏移量

8字节 一个序列号

Byte 这个package是否是这个 block中最后一个包

4字节 实际的数据长度，除去 checksum。

CheckSum数据 缺省的 CheckSum为CRC32，（缺省每个 chunk的checksum占四个字节）

实际 block数据 有多少写多少

数据传输过程中一定需要耗时，如何知道数据传输链中的 datanode都是正常工作的呢？以及 datanode成功收到 package包以后是如何应答吗？这部分协议相关字段如下：

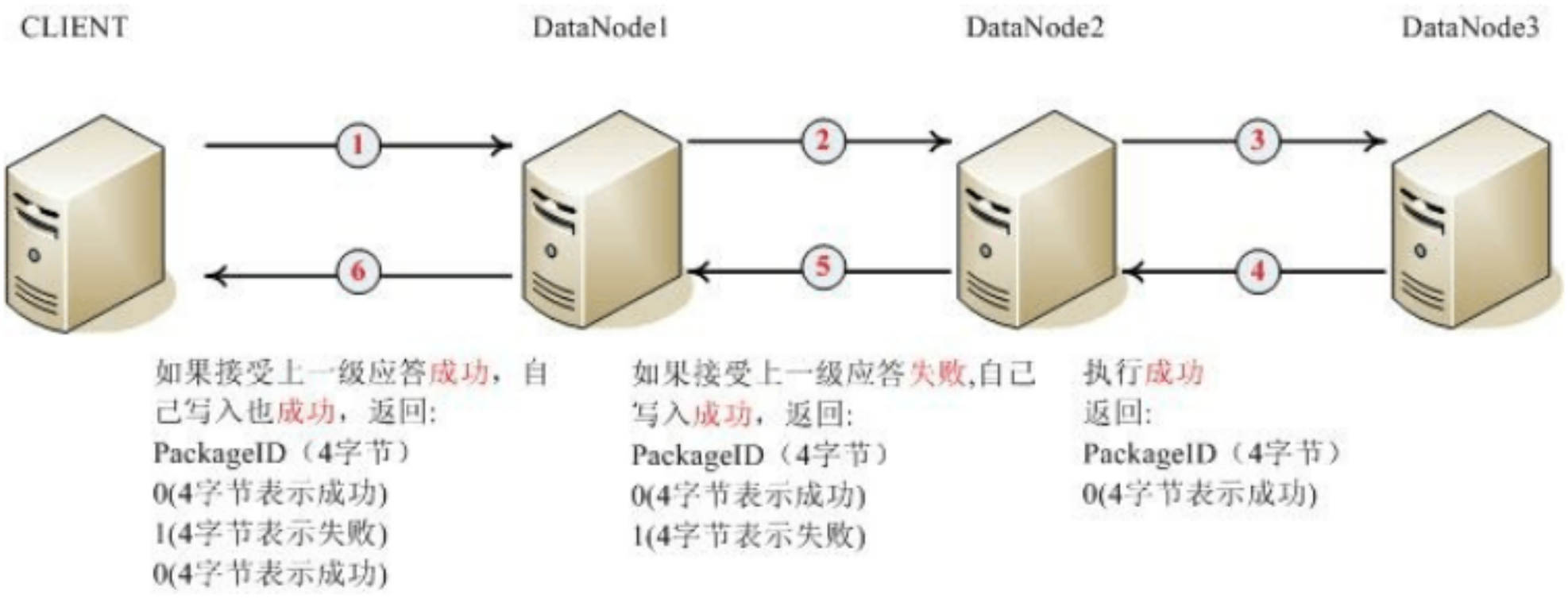
如果收到下一个 datanode返回 4字节（ -1），表示心跳正常

4字节 opCode(-1,表示心跳正常)

如果收到下一个 datanode返回 4字节（不等于 -1，也不等于 -2）

4字节　　packageID(向上一级返回接受到的　packageID)
4字节　　OP_STATUS_SUCCESS = 0; OP_STATUS_ERROR = 1;
0表示接受成功，　1表示接受失败。

这个字节根据数据链中的位置可能会收到多个，详细见下图：



client最终可以根据返回的　PackageID后的多个状态字节得知哪台机器可能出现问题，从上面的流程可以看到：　　datanode3可能已经写入成功，datanode2由于没有收到　datanode3的应答，故而认为　datanode3接受package失败，这个状态一直透传到　client。

也有可能 datanode3节点的确失效，　datanode2也会收不到　datanode3的应答响应。

DFSClient 从Namenode取得需要读取的文件对应的　LocatedBlocks 信息以后，就会按照　block 的顺序与 datanode建立链接并发送读取　block 数据的请求。我们看一下这部分的协议格式：

4字节	数据传输协议版本号（　0.19.1　版本的 hadoop 这个字段为 14）
4字节	操作码（读取文件时为：　OP_READ_BLOCK, 对应81）
8字节	Block　的BLOCKID 字段
8字节	Block　的时间戳字段
8字节	读取在 block　数据文件开始的偏移量
8字节	一共读取多少字节
Text	Client　名称字串，格式为字串长度（　这个采取了压缩形式 ）+字串内容

DFSClient 发送读取数据块儿请求完成以后，首先等待 datanode的应答， datanode应答协议格式如下：

4字节	OP_STATUS_SUCCESS=0表示链接建立成功 OP_STATUS_ERROR=1表示链接建立失败

接下来 Datanode 就开始进行数据传送，具体数据格式如下：

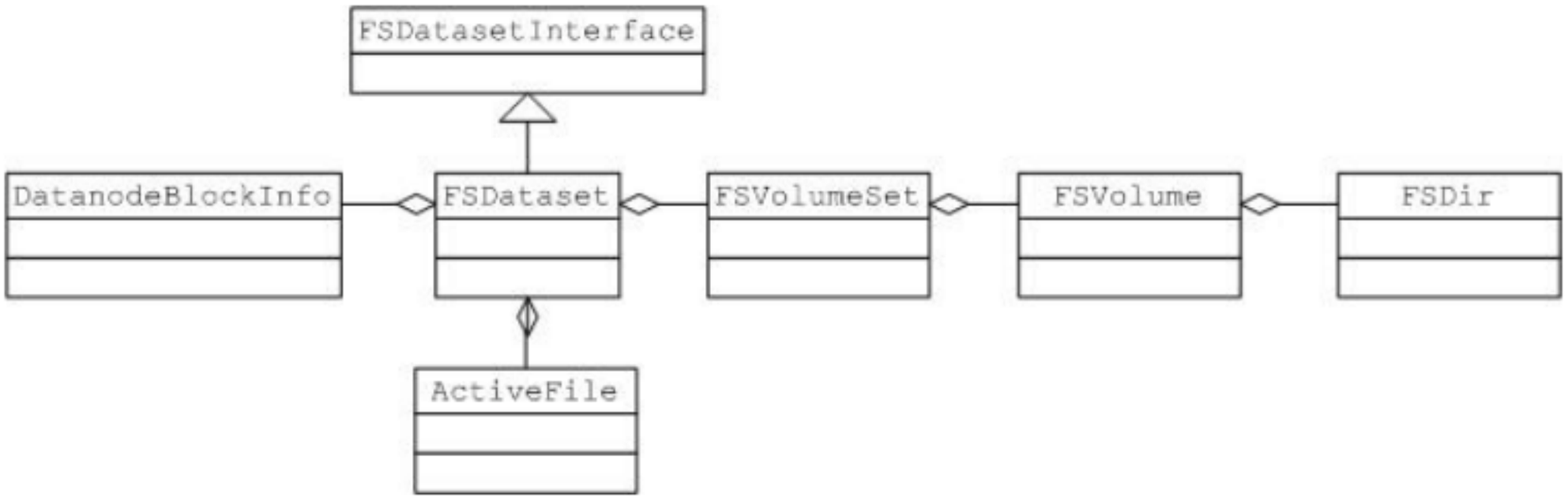
数据校验相关信息	Byte	Chunksum 类型
Int bytesPerChecksum	每次 checksum 对应的字节数	
8字节	Offset	（读取一个 block 开始的偏移量） （这8字节有些情况没有）
1.DFSClient	从Datanode 读取 block 数据时，这个字段是必须有的。	
2.Datanode	之间进行 block 互相拷贝时（ balance 需要，或者 block 块儿没有达到副本个数要求），这个字段是不存在的。	
4字节	packetLen,package 长度	
8字节	chunk 的偏移量（用户读取文件时，很可能偏移一个位置，偏移的位置很可能不在一个 chunk 的结束位置，因为 checksum 是按照 chunk 来计算的，所以 hadoop 会将这部分偏移量的数据多传送给 client ， client 需要将这部分数据丢弃）	
8字节	Seqno	序列号
Byte	是否是 block 数据块的最后一个 package	
4字节	传输的 block 中包含的数据大小	
CheckSum 数据	缺省的 CheckSum 为CRC32，（缺省每个 chunk 的checksum 占四个字节）	
实际 block 数据	有多少写多少	

HDFS-Datanode　　关于 block　文件的管理

数据文件真正存储的地方是在　datanode，当用户需要填充文件中某一个block 的实际数据内容时，就需要连接到　datanode进行实际的 block 写入操作，下面我们看一下　datanode如何管理 block ，以及如何存储 block 。

Datanode是通过文件存储 block 数据的 ，datanode中有一个FSDatasetInterface　　接口，这个接口的主要作用就是对　block 对应的实际数据文件进行操作。

首先我们看一下这部分涉及的主要类关系：



FSDataset　实现了接口 FSDatasetInterface　　，主要承载 block 对应的文件操作。

FSDataset　有几个重要的属性：

```
FSVolumeSet volumes;
private HashMap<Block, ActiveFile> ongoingCreates
= new HashMap<Block, ActiveFile>();
private HashMap<Block, DatanodeBlockInfo>
volumeMap = null;
```

Datanode　可以配置多个目录来存储　block　对应的文件，配置的每一个目录就对应一个　FSVolume ，每一个 FSVolume 对应的目录中会有一个current　　目录，这个目录进行实际文件存放，同时系统会在这个目录下自行创建很多子文件夹（每个文件夹存放的文件个数是有限制的）。每个子文件夹就会对应一个　FSDir　对象。

我么先看一个 FSVolume 对应目录的文件结构：

```
distribute-hadoop-boss@hadoop172-16-218-109:/data1/dfs/data> pwd
/data1/dfs/data
distribute-hadoop-boss@hadoop172-16-218-109:/data1/dfs/data> ls
current  detach  in_use.lock  storage  tmp
```

/data1/dfs/data/　　这个目录是我们配置的一个存储　block 对应的文件

的目录，也就是对应一个 FSVolume，其中 current 目录是我们真正存放 block 数据块文件的地方，其中还有一个 tmp目录，这个目录主要临时存放一些正在写入的 block 数据文件，成功写入完成后这个临时文件就会从tmp目录移动到 current 目录。

看一下我们的配置文件内容

```
<name>dfs.data.dir</name>
<value>${hadoop.tmp.dir}/dfs/data,/data1/dfs/data/,/data2/dfs/data/,/data3/dfs/data/,/data4/dfs/data/
```

可以看到我们配置了 /data1/dfs/data/ 目录做为我们数据存储目录之一。

下面我们看一下实际的 /data1/dfs/data/current 目录下会有那些内容

```
distribute-hadoop-boss@hadoop172-16-218-109:/data5/dfs/data/current> pwd
/data5/dfs/data/current
distribute-hadoop-boss@hadoop172-16-218-109:/data5/dfs/data/current> ls
blk_1030194901617285606              blk_4219793740419837O_1762289.meta    blk_740594592431508994              subdir26
blk_1030194901617285606_1806929.meta blk_4259974877543818638               blk_740594592431508994_1766500.meta subdir27
blk_110104886973365714               blk_4259974877543818638_1767059.meta  blk_-760626345059408128             subdir28
blk_110104886973365714_1764609.meta  blk_-4307442546244024103              blk_-760626345059408128_1766058.meta subdir29
blk_1178145829636463169              blk_-4307442546244024103_1763658.meta blk_7642699599886492103             subdir3
blk_1178145829636463169_1761567.meta blk_4343206839744485609              blk_7642699599886492103_1764887.meta subdir30
blk_1369158373250385052              blk_4343206839744485609_1807185.meta  blk_7664613894483835213             subdir31
blk_1369158373250385052_1761861.meta blk_-4366482584752032752             blk_7664613894483835213_1772377.meta subdir32
blk_1625773782971755689              blk_-4366482584752032752_1763123.meta blk_-7841220916945562710            subdir33
blk_1625773782971755689_1765202.meta blk_438317301428857249               blk_-7841220916945562710_1762937.meta subdir34
blk_-1762529950983705120             blk_438317301428857249_1762817.meta   blk_7916633398131063228             subdir35
```

可以看到这个目录主要存放了 block 数据文件，block 数据文件的命名规则是 blk_$BLOCKID$, 还有就是 block 文件的元数据文件，元数据文件的命名规则是： blk_$BLOCKID$_$ 时间戳 $, 同时大家可以看到很多累似 subdir00 命名的文件夹，这些文件夹其中存放的也是 block 数据文件及其元数据文件， hadoop 规定每个目录存放的 block 数据文件个数是有限制的，达到限制之后就会新建 sub 子目录进行存放，这些 sub 子目录包括 current 目录，都会和一个 FSDir 对象对应。

我们来看一下 FSDir 对象，这个对象和文件系统中的目录是很相近的概念，他的主要作用就是管理一个目录下的所有与 block 相关的文件。

我们先看一个这个类的主要属性：

```
File dir;  //                这个对象对应的文件目录对象。
FSDir children[]; //                这个目录下的子文件夹对象
```

这个对象在进行实例化构造的过程就会遍历文件夹下的文件，判断哪些是目录，然后生成相应的 FSDir 添加到 children 数组中。

主要方法：

```
void getVolumeMap(HashMap<Block,
DatanodeBlockInfo> volumeMap,FSVolume volume)
```

这个方法的主要目的就是遍历整个目录，得到所有 block 文件列表，并添加所有 block 的记录到 FSDataset 的volumeMap 属性中，参数中的 volumeMap 传入的正是 FSDataset 的volumeMap 属性，这个对

象保存 block 与DatanodeBlockInfo 的映射关系，便于通过 block 查询具体的 block 文件信息。

datanodeBlockInfo 对象主要保存了 block 属于哪一个 FSVolume ，以及 block 块实际的存放文件是哪个。

```
{
    if (children != null)
    {
        for (int i = 0; i < children.length; i++)
        {
            children[i].getVolumeMap(volumeMap, volume);//       遍历子文件夹
        }
    }
    File blockFiles[] = dir.listFiles();
    for (int i = 0; i < blockFiles.length; i++)
    {
    //  判断文件是否是一个 block ，主要是通过文件名来判断的, block 的文件名的命名特征是 "blk_$BLOCKID"
        if (Block.isBlockFilename(blockFiles[i]))
        {
            long genStamp = getGenerationStampFromFile(blockFiles,
            blockFiles[i]);
            volumeMap.put(new Block(blockFiles[i], blockFiles[i].length(), genStamp), new DatanodeBlockInfo(volume,blockFiles[i]));
        }
    }
}
```

Volumes 相对来说比较简单，他就是一个 FSVolume 集合，并封装了对所有 volume 的操作，比如： getBlockReport() 这个方法，就是得到所有的 block 列表，上报给 namenode 。有了 volumes 这个集合对象，类似这样的操作就可以封装起来，方便后续操作。

ongoingCreates

这个对象主要存储了正在创建的 block 列表，这个对象中的 block 表示用户正在进行该 block 文件数据上传操作，这个对象中包含 ActiveFile 对象的实例，我们先看一下 ActiveFile 对象，这个对象

中有两个关键属性： 1. 对象对应的数据文件。 2. 正在操作这个文件的线程列表。

```
final File file;
final List<Thread> threads = new ArrayList<Thread>(2);
```

保存线程列表的主要目的是：在进行 block 文件写入操作时，如果 datanode 收到了对这个 block 进行 recoverblock 的请求后，需要先 interrupt 所有正在写入这个 block 文件的线程。

我们先看一下文件数据的接受流程。

接下来我们会详细介绍 datanode 接收数据的协议，下面会提到 Datanode 构建一个 BlockReceiver 实例，进行实际数据的接受操作。

BlockReceiver 在构建实例过程中会首先通过如下方法：streams = datanode.data.writeToBlock(block, isRecovery); 打开 block 数据写入 datanode 本地文件的通道，以便 datanode 接收到 block 数据块儿内容以后将数据内容写入磁盘。第一步我们可能需要了解 BlockWriteStreams 这个对象，因为 writeToBlock 会返回 BlockWriteStreams 对象实例。

```
static class BlockWriteStreams
{
    OutputStream dataOut;
    OutputStream checksumOut;
    BlockWriteStreams(OutputStream dOut, OutputStream cOut) {
      dataOut = dOut;
      checksumOut = cOut;
}
```

我们可以看到这个对象包含两个重要的属性： dataOut ， checksumOut 。从字面意思我们也可以揣测到，这两个 outputstream 对象，一个用来写入 block 数据，一个用来写入 block 数据中的 checksum 数据。

接下来我们详细分析一下这个方法 ：

```
public BlockWriteStreams writeToBlock(Block b, boolean isRecovery)
{
    .....// 前面主要做一些合法性判断。
    if (!isRecovery)
```

```
    {
        v = volumes.getNextVolume(blockSize);
    //  这是很重要的方法，主要是先创建一个临时文件存储上传的数
据，等一个完成的 block 文件写入完成以后，再将这个文件以及元数据
文件移动到正式文件目录。
        f = createTmpFile(v, b);
        volumeMap.put(b, new DatanodeBlockInfo(v));
        ...
        }
    ...
    return createBlockWriteStreams(f, metafile);
}
```

主要是创建一个临时文件和一个存放元数据信息的临时文件，然后打开两个文件将返回的 OutputStream 返回给前端。

HDFS的Java 访问接口

得到 filesystem 的实例，有两个静态方法可以得到 filesystem 接口的实例

```
public static FileSystem get(Configuration conf)
throws IOException
public static FileSystem get(URI uri, Configuration
conf) throws IOException
```

第一个方法得到缺省的文件系统，具体由配置文件的 fs.default.name 属性决定。

第二个方法由 URI 的前缀决定是哪个文件系统（参考前一篇文章），如果 URI 没有前缀也是得到缺省的文件系统。

使用 open 方法得到文件的输入流

有了 filesystem 的实例，我们就可以调用 open 方法得到某个文件的输入流了

```
public FSDataInputStream open(Path f) throws
IOException
public abstract FSDataInputStream open(Path f, int
bufferSize) throws IOException
```

第一个方法用缺省的 buffer 大小 4K

使用 FSDataInputStream 读数据

FSDataInputStream 派生自 java.io.DataInputStream ，支持随机读取。

```
public class FSDataInputStream extends
DataInputStream
    implements Seekable, PositionedReadable {
    void seek(long pos) throws IOException;
    public int read(long position, byte[] buffer,
int offset, int length) throws IOException;
}
```

使用 FSDataOutputStream 写数据

filesystem 接口创建文件的接口

```
public FSDataOutputStream create(Path f) throws
IOException
```

还可以 append 到一个已经存在的文件

```
public FSDataOutputStream append(Path f) throws
IOException
```

public class FSDataOutputStream extends
DataOutputStream implements Syncable {
   public void write(int b) throws IOException;
   public void write(byte b[], int off, int len)
throws IOException;
}
filesystem　　的文件系统操作方法

public boolean mkdirs(Path f) throws IOException ;
public void copyToLocalFile(boolean delSrc, Path
src, Path dst) throws IOException;
public abstract boolean delete(Path f, boolean
recursive) throws IOException;
public abstract boolean rename(Path src, Path dst)
throws IOException;
public abstract FileStatus[] listStatus(Path f)
throws IOException;

hadoop 文件读写

hadoop 提供了文件读写的多种 java 接口形式，下面一一叙述。

1.A simple way to read Data from a hadoop URL

一个最简单的方法是用 java.net.URL 类建立一个流读取 hadoop 中的文件。实现的代码如下：

```
InputStream in = null;
try{
    in = new URL("hdfs://host:9000/hadoop-dfs-path");
    //process in
}finally
{
    IOUtils.closeStream(in);
}
```

这个读写方法简单，但是有一个局限性就是要求 JAVA 能识别 Hadoop 的 hdfs URL Schema ，类似于 hdfs://host:9000/hadoop-dfs-path 这样的 URL 。当然可以通过传入 FsUrlStreamHandlerFactory 的一个实例给 URL 类的一个方法 setURLStreamHandlerFactory 来实现。但是这个方法在一个 JVM 只能被调用一次（ once per JVM ），因此需要在静态块中执行。这种方法的局限性在于，如果你的一段程序比如非你所控制的第三方的组件使用了 setURLStreamHandlerFactory ，则你的程序将不能再使用这个方法来实现从 hadoop 中读出数据。随后将讨论解决这一问题。

IOUtils 类的 copyBytes 方法实现的是从输入（ in）到输出（ System.out ）的拷贝，并给出了一个 4096bit 大小的缓冲区， false 表示结束时并不关闭输入流。

2.Reading Data Using the FileSystem API

正如前文所言，在程序中利用 URLStreamHandlerFactory 存在着一些局限，因此，可以采用 FileSystem API 来打开一个文件输入流。

在 Hadoop 文件系统中 Path 类的实例用来表示一个文件，字面上看起来类似于本地文件系统的路径，当然你可以把 Path 看做 Hadoop fileSystem 的 URI ，例如上文中在我机器中使用的 hdfs://192.168.2.38:9000/user/had/conf/core-site.xml （其实我是把 Hadoop 的配置文件全部 put 到 hdfs 上去了）。

在 hadoop 中两个静态方法可以获得 FileSystem 实例：

(1)public static FileSystem get(configuration conf) throws IOException

(2)public static FileSystem get(URI url,Configration conf) throws IOException

Configuration 对象封装了一个客户端或服务器端的配置，默认地从配置文件中读取，比如 conf/core-site.xml.

(1) 返回的是默认的 Hdfs 文件系统（ conf/core-site.xml 中所描述的文件系统或默认的本地文件系统）

(2) 采用给定的 URI 描述和权限载入文件系统，若不指定 URI ，则返回默认的文件系统。

对于得到的 FileSystem 实例，可以调用方法 open(), 从文件中获得输入流。

```
public FSDataInputStream open(Path f)throws IOException
public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

第一个 open 方法的默认缓冲区大小是 4K。

```
public static viod main(String[] args) throws Exception{
    String uri = arg[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri),conf);
    InputStream in = null;
    try
    {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in,System.out,4096,false);
```

```
    }finally
    {
      IOUtils.closeStream(in);
    }
  }
```

3. FSDataInputStream

第2部分所描述的 open （）方法实际上返回了一个 FSDataInputStream 实例而不仅仅是一个标准的 java.io 类， FSDataInputStream 继承自 DataInputStream ，支持随机读取，因此你可以读取流的任意部分。

· FSDataInputStream 的定义：

```
package org.apache.hadoop.fs;
public class FSDataInputStream extends DataInputStream implements
Seekable,PositionedReadable{
//implements elided
}
```

seekable 接口的实现，其中的方法 seek （）允许定位到文件中的任一位置， getPos （）方法更是能轻易地得到目前偏移位置。

```
public interface Seekable{
  void seek(long pos) throws IOException;
  long getPos() throws IOException;
  boolean seekToNewSource(long targetPos) throws IOException;
}
```

seek 方法的调用必须小心，超越文件的 length 则可能导致 IOException 的异常。
与java.io.InputStream 中的 skip （）方法不同， seek （）可以任意地前后移动到某一位置。
seekToNewSource() 方法一般被 writers 所用，用来寻找数据的另一个 copy 并在此 copy 中定位到 targetPos. 非常重要的一句话，原文： This is used internally in HDFS to provide a reliable input stream of data to the client in the face of datanode failure.

利用第三部分的描述，我们可以扩充第二部分的程序，在读出文件后，定位到第 10 个字节，再输出一次。

输入文件：文件是 MapReduce 任务的数据的初始存储地。正常情况下，输入文件一般是存在 HDFS 里。这些文件的格式可以是任意的；我们可以使用基于行的日志文件，也可以使用二进制格式，多行输入记录或其它一些格式。这些文件会很大—数十 G或更大。

输入格式： InputFormat 类定义了如何分割和读取输入文件，它提供有下面的几个功能：

1. 选择作为输入的文件或对象；
2. 定义把文件划分到任务的 InputSplits ；
3. 为RecordReader 读取文件提供了一个工厂方法；

Hadoop 自带了好几个输入格式。其中有一个抽象类叫 FileInputFormat ，所有操作文件的 InputFormat 类都是从它那里继承功能和属性。当开启 Hadoop 作业时， FileInputFormat 会得到一个路径参数，这个路径内包含了所需要处理的文件， FileInputFormat 会读取这个文件夹内的所有文件（译注：默认不包括子文件夹内的），然后它会把这些文件拆分成一个或多个的 InputSplit 。你可以通过 JobConf 对象的 setInputFormat() 方法来设定应用到你的作业输入文件上的输入格式。下表给出了一些标准的输入格式：

输入格式	描述	键	值
TextInputFormat	默认格式，读取文件的行	行的字节偏移量	行的内容
KeyValueInputFormat	把行解析为键值对	首个 tab 字符前的所有字符	行剩下的内容
SequenceFileInputFormat	Hadoop 定义的高性能二进制格式	用户自定义	用户自定义

表4.1MapReduce 提供的输入格式

默认的输入格式是 TextInputFormat ，它把输入文件每一行作为单独的一个记录，但不做解析处理。这对那些没有被格式化的数据或是基于行的记录来说是很有用的，比如日志文件。更有趣的一个输入格式是KeyValueInputFormat ，这个格式也是把输入文件每一行作为单独

的一个记录。然而不同的是 TextInputFormat 把整个文件行当做值数据， KeyValueInputFormat 则是通过搜寻 tab 字符来把行拆分为键值对。这在把一个 MapReduce 的作业输出作为下一个作业的输入时显得特别有用，因为默认输出格式（下面有更详细的描述）正是按KeyValueInputFormat 格式输出数据。最后来讲讲 SequenceFileInputFormat ，它会读取特殊的特定于 Hadoop 的二进制文件，这些文件包含了很多能让 Hadoop 的mapper 快速读取数据的特性。 Sequence 文件是块压缩的并提供了对几种数据类型（不仅仅是文本类型）直接的序列化与反序列化操作。 Squence 文件可以作为MapReduce 任务的输出数据，并且用它做一个 MapReduce 作业到另一个作业的中间数据是很高效的。