

## [Hadoop 源代码分析 \(一\)](#)

关键字: 分布式 云计算

Google 的核心竞争技术是它的计算平台。Google 的大牛们用了下面 5 篇文章，介绍了它们的计算设施。

GoogleCluster : <http://research.google.com/archive/googlecluster.html>

Chubby : <http://labs.google.com/papers/chubby.html>

GFS : <http://labs.google.com/papers/gfs.html>

BigTable : <http://labs.google.com/papers/bigtable.html>

MapReduce : <http://labs.google.com/papers/mapreduce.html>

很快，Apache 上就出现了一个类似的解决方案，目前它们都属于 Apache 的 Hadoop 项目，对应的分别是：

Chubby-->ZooKeeper

GFS-->HDFS

BigTable-->HBase

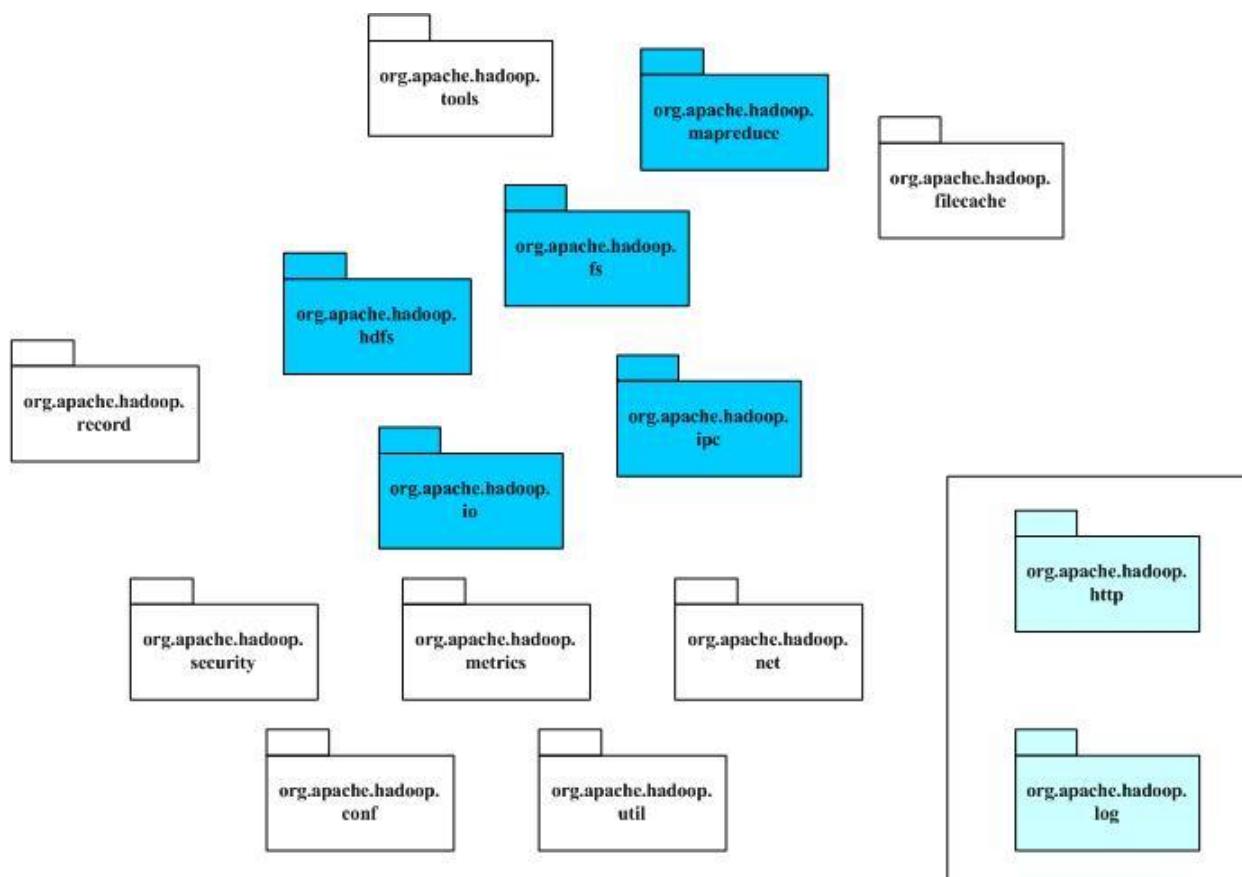
MapReduce-->Hadoop

目前，基于类似思想的 Open Source 项目还很多，如 Facebook 用于用户分析的 Hive。

HDFS 作为一个分布式文件系统，是所有这些项目的基础。分析好 HDFS，有利于了解其他系统。由于 Hadoop 的 HDFS 和 MapReduce 是同一个项目，我们就把他们放在一块，进行分析。

下图是 MapReduce 整个项目的顶层包图和他们的依赖关系。Hadoop 包之间的依赖关系比较复杂，原因是 HDFS 提供了一个分布式文件系统，该系统提供 API，可以屏蔽本地文件系统和分布式文件系统，甚至象 Amazon S3 这样的在线存储系统。这就造成了分布式文件系统的实现，或者是分布式文件系统的底层的实现，依赖于某些貌似高层的功能。功能的相互引用，造成了蜘蛛网型的依赖关系。一个典型的例子就是包 conf，conf 用于读取系统配置，它依赖于 fs，主要是读取配置文件的时候，需要使用文件系统，而部分的文件系统的功能，在包 fs 中被抽象了。

Hadoop 的关键部分集中于图中蓝色部分，这也是我们考察的重点。



Package	Dependences
tool	mapreduce, fs, hdfs, ipc, io, security, conf, util
<b>mapreduce</b>	filecache, <b>fs</b> , <b>hdfs</b> , <b>ipc</b> , <b>io</b> , net, metrics, security, conf, util
filecache	fs, conf, util
<b>fs</b>	<b>hdfs</b> , <b>ipc</b> , io, net, metrics, security, conf, util
<b>hdfs</b>	fs, <b>ipc</b> , io, http, net, metrics, security, conf, util
<b>ipc</b>	<b>io</b> , net, metrics, security, conf, util
<b>io</b>	<b>ipc</b> , fs, conf, util
net	ipc, fs, conf, util
security	io, conf, util
conf	fs, io, util
metrics	util
util	<b>mapred</b> , fs, io, conf
record	io.Writable*
http	log, conf, util
log	util

## Hadoop 源代码分析 (二)

下面给出了 Hadoop 的包的功能分析。

Package	Dependences
tool	提供一些命令行工具，如 DistCp , archive
<b>mapreduce</b>	Hadoop 的 Map/Reduce 实现
filecache	提供 HDFS 文件的本地缓存，用于加快 Map/Reduce 的数据访问速度
<b>fs</b>	文件系统的抽象，可以理解为支持多种文件系统实现的统一文件访问接口
<b>hdfs</b>	HDFS , Hadoop 的分布式文件系统实现
<b>ipc</b>	一个简单的 IPC 的实现，依赖于 io 提供的编解码功能 参 考： <a href="http://zhangyu8374.javaeye.com/blog/86306">http://zhangyu8374.javaeye.com/blog/86306</a>

io	表示层。将各种数据编码/解码，方便于在网络上传输
net	封装部分网络功能，如 DNS，socket
security	用户和用户组信息
conf	系统的配置参数
metrics	系统统计数据的收集，属于网管范畴
util	工具类
record	根据 DDL（数据描述语言）自动生成他们的编解码函数，目前可以提供 C++ 和 Java
http	基于 Jetty 的 HTTP Servlet，用户通过浏览器可以观察文件系统的一些状态信息和日志
log	提供 HTTP 访问日志的 HTTP Servlet

### Hadoop 源代码分析（三）

由于 Hadoop 的 MapReduce 和 HDFS 都有通信的需求，需要对通信的对象进行序列化。Hadoop 并没有采用 Java 的序列化，而是引入了它自己的系统。

org.apache.hadoop.io 中定义了大量的可序列化对象，他们都实现了 Writable 接口。实现了 Writable 接口的一个典型例子如下：

Java 代码

```

1. public class MyWritable implements Writable {
2.     // Some data
3.     private int counter;
4.     private long timestamp;
5.
6.     public void write(DataOutput out) throws IOException {
7.         out.writeInt(counter);
8.         out.writeLong(timestamp);
9.     }
10.
11.    public void readFields(DataInput in) throws IOException {
12.        counter = in.readInt();
13.        timestamp = in.readLong();
14.    }
15.
```

```
16.     public static MyWritable read(DataInput in) throws IOException {  
17.         MyWritable w = new MyWritable();  
18.         w.readFields(in);  
19.         return w;  
20.     }  
21. }
```

其中的 write 和 readFields 分别实现了把对象序列化和反序列化的功能，是 Writable 接口定义的两个方法。下图给出了庞大的 org.apache.hadoop.io 中对象的关系。

这里，我把 ObjectWritable 标为红色，是因为相对于其他对象，它有不同的地位。当我们讨论 Hadoop 的 RPC 时，我们会提到 RPC 上交换的信息，必须是 Java 的基本类型，String 和 Writable 接口的实现类，以及元素为以上类型的数组。

ObjectWritable 对象保存了一个可以在 RPC 上传输的对象和对象的类型信息。这样，我们就有了一个万能的，可以用于客户端/服务器间传输的 Writable 对象。例如，我们要把上面例子中的对象作为 RPC 请求，需要根据 MyWritable 创建一个 ObjectWritable，ObjectWritable 往流里会写如下信息

对象类名长度，对象类名，对象自己的串行化结果

这样，到了对端，ObjectWritable 可以根据对象类名创建对应的对象，并解串行。应该注意到，ObjectWritable 依赖于 WritableFactories，那存储了 Writable 子类对应的工厂。我们需要把 MyWritable 的工厂，保存在 WritableFactories 中（通过 WritableFactories.setFactory）。

## Hadoop 源代码分析（五）

介绍完 org.apache.hadoop.io 以后，我们开始来分析 org.apache.hadoop.rpc。RPC 采用客户机/服务器模式。请求程序就

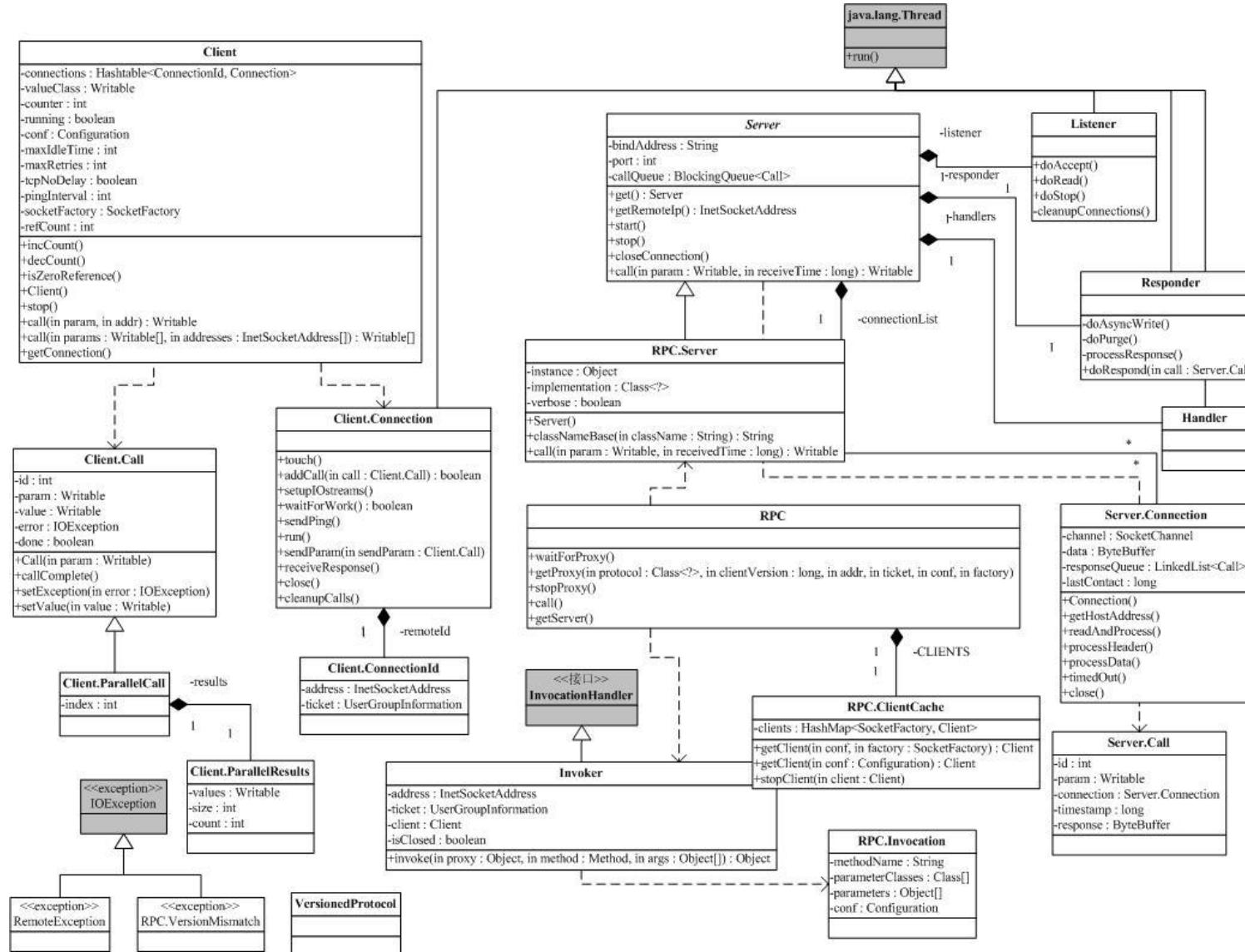
是一个客户机，而服务提供程序就是一个服务器。当我们讨论 HDFS 的，通信可能发生在：

- Client-NameNode 之间，其中 NameNode 是服务器
- Client-DataNode 之间，其中 DataNode 是服务器
- DataNode-NameNode 之间，其中 NameNode 是服务器
- DataNode-DataNode 之间，其中某一个 DataNode 是服务器，另一个是客户端

如果我们考虑 Hadoop 的 Map/Reduce 以后，这些系统间的通信就更复杂了。为了解决这些客户机/服务器之间的通信，Hadoop 引入了一个 RPC 框架。该 RPC 框架利用的 Java 的反射能力，避免了某些 RPC 解决方案中需要根据某种接口语言(如 CORBA 的 IDL )生成存根和框架的问题。但是，该 RPC 框架要求调用的参数和返回结果必须是 Java 的基本类型，String 和 Writable 接口的实现类，以及元素为以上类型的数组。同时，接口方法应该只抛出 IOException 异常。（参考自 <http://zhangyu8374.javaeye.com/blog/86306>）

既然是 RPC，当然就有客户端和服务器，当然，org.apache.hadoop.rpc 也就有了类 Client 和类 Server。但是类 Server 是一个抽象类，类 RPC 封装了 Server，利用反射，把某个对象的方法开放出来，变成 RPC 中的服务器。

下图是 org.apache.hadoop.rpc 的类图。

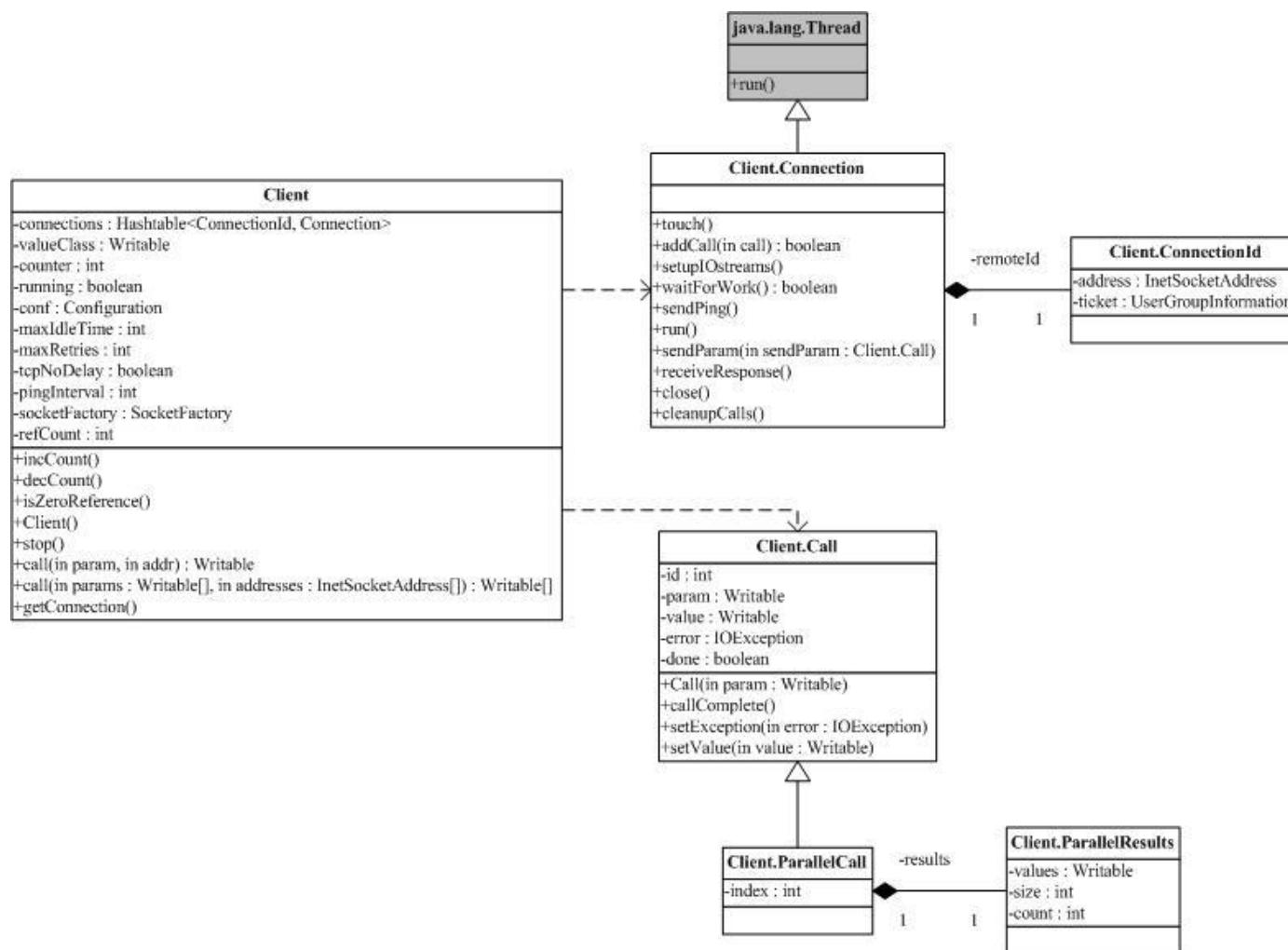


## Hadoop 源代码分析（六）

既然是 RPC，自然就有客户端和服务器，当然，org.apache.hadoop.rpc 也就有了类 Client 和类 Server。在这里我们来仔细考察 org.apache.hadoop.rpc.Client。下面的图包含了 org.apache.hadoop.rpc.Client 中的关键类和关键方法。

由于 Client 可能和多个 Server 通信，典型的一次 HDFS 读，需要和 NameNode 打交道，也需要和某个/某些 DataNode 通信。这就意味着某一个 Client 需要维护多个连接。同时，为了减少不必要的连接，现在 Client 的做法是拿 ConnectionId（图

中最右侧)来做为 Connection 的 ID。ConnectionId 包括一个 InetSocketAddress (IP 地址+端口号或主机名+端口号) 对象和一个用户信息对象。这就是说,同一个用户到同一个 InetSocketAddress 的通信将共享同一个连接。



连接被封装在类 Client.Connection 中,所有的 RPC 调用,都是通过 Connection,进行通信。一个 RPC 调用,自然有输入参数,输出参数和可能的异常,同时,为了区分在同一个 Connection 上的不同调用,每个调用都有唯一的 id。调用是否结束也需要一个标记,所有的这些都体现在对象 Client.Call 中。Connection 对象通过一个 Hash 表,维护在这个连接上的所有 Call:

#### Java 代码

1. private Hashtable<Integer, Call> calls = new Hashtable<Integer, Call>();

一个 RPC 调用通过 addCall ,把请求加到 Connection 里。为了能够在这个框架上传输 Java 的基本类型, String 和 Writable 接口的实现类,以及元素为以上类型的数组,我们一般把 Call 需要的参数打包成为 ObjectWritable 对象。

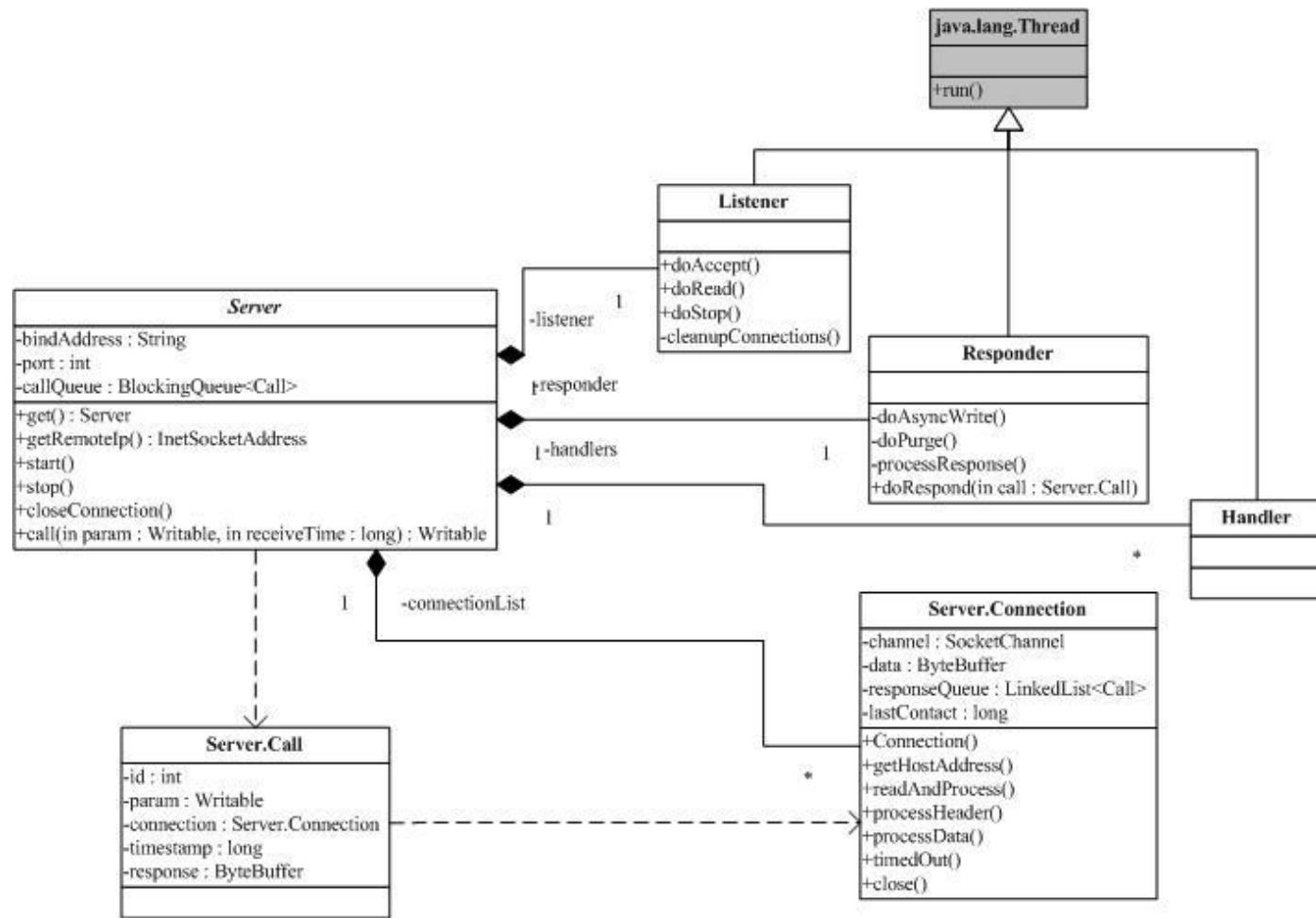
Client.Connection 会通过 socket 连接服务器,连接成功后回校验客户端/服务器的版本号( Client.ConnectionwriteHeader() 方法),校验成功后就可以通过 Writable 对象来进行请求的发送/应答了。注意,每个 Client.Connection 会起一个线程,不断去读取 socket ,并将收到的结果解包,找出对应的 Call ,设置 Call 并通知结果已经获取。

Call 使用 Object 的 wait 和 notify ,把 RPC 上的异步消息交互转成同步调用。

还有一点需要注意,一个 Client 会有多个 Client.Connection ,这是一个很自然的结果。

## [Hadoop 源代码分析 \(七\)](#)

聊完了 Client 聊 Server ,按惯例,先把类图贴出来。



需要注意的是，这里的 Server 类是个抽象类，唯一抽象的地方，就是

#### Java 代码

1. public abstract Writable call(Writable param, long receiveTime) throws IOException;

这表明，Server 提供了一个架子，Server 的具体功能，需要具体类来完成。而具体类，当然就是实现 call 方法。

我们先来分析 Server.Call，和 Client.Call 类似，Server.Call 包含了一次请求，其中，id 和 param 的含义和 Client.Call 是一致的。不同点在后面三个属性，connection 是该 Call 来自的连接，当然，当请求处理结束时，相应地结果会通过相同的 connection，发送给客户端。属性 timestamp 是请求到达的时间戳，如果请求很长时间没被处理，对应的连接会被关闭，客户端也就知道出错了。最后的 response 是请求处理的结果，可能是一个 Writable 的串行化结果，也可能一个异常的串行化结果。

Server.Connection 维护了一个来自客户端的 socket 连接。它处理版本校验，读取请求并把请求发送到请求处理线程，接收处理结果并把结果发送给客户端。

Hadoop 的 Server 采用了 Java 的 NIO，这样的话就不需要为每一个 socket 连接建立一个线程，读取 socket 上的数据。在 Server 中，只需要一个线程，就可以 accept 新的连接请求和读取 socket 上的数据，这个线程，就是上面图里的 Listener。

请求处理线程一般有多个，它们都是 Server.Handle 类的实例。它们的 run 方法循环地取出一个 Server.Call，调用 Server.call 方法，搜集结果并串行化，然后将结果放入 Responder 队列中。

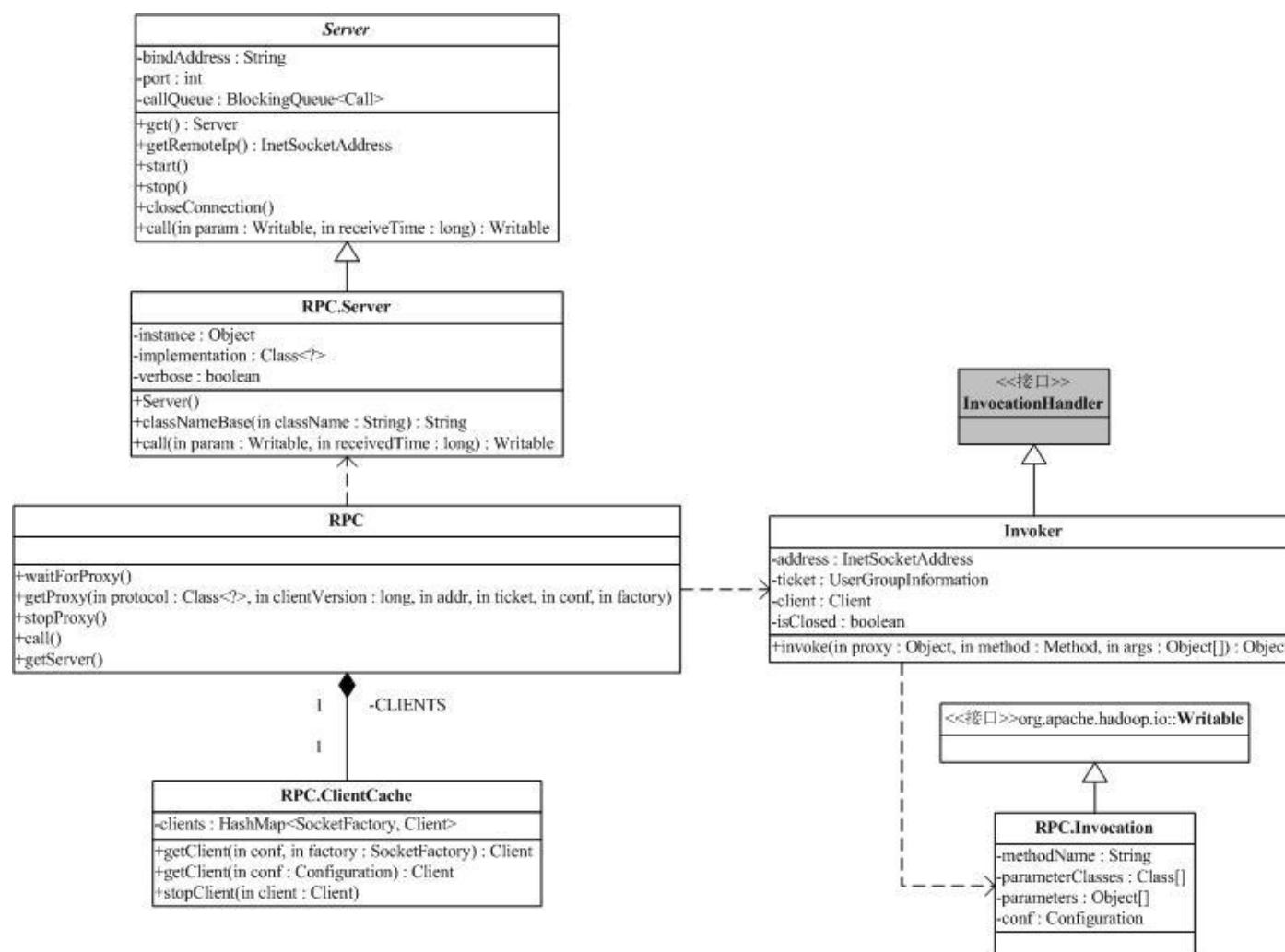
对于处理完的请求，需要将结果写回去，同样，利用 NIO，只需要一个线程，相关的逻辑在 Responder 里。

## Hadoop 源代码分析（八）

（注：本节需要用到一些 Java 反射的背景）

有了 Client 和 Server，很自然就能 RPC 啦。下面轮到 RPC.java 啦。

一般来说，分布式对象一般都会要求根据接口生成存根和框架。如 CORBA，可以通过 IDL，生成存根和框架。但是，在 org.apache.hadoop.rpc，我们就不需要这样的步骤了。上类图。



为了分析 Invoker，我们需要介绍一些 Java 反射实现 Dynamic Proxy 的背景。

Dynamic Proxy 是由两个 class 实现的：`java.lang.reflect.Proxy` 和 `java.lang.reflect.InvocationHandler`，后者是一个接口。

所谓 Dynamic Proxy 是这样一种 class：它是在运行时生成的 class，在生成它时你必须提供一组 interface 给它，然后该 class 就宣称它实现了这些 interface。

这个 Dynamic Proxy 其实就是一个典型的 Proxy 模式，它不会替你作实质性的工作，在生成它的实例时你必须提供一个 handler，由它接管实际的工作。这个 handler，在 Hadoop 的 RPC 中，就是 Invoker 对象。

我们可以简单地理解：就是你可以通过一个接口来生成一个类，这个类上的所有方法调用，都会传递到你生成类时传递的 InvocationHandler 实现中。

在 Hadoop 的 RPC 中，Invoker 实现了 InvocationHandler 的 invoke 方法（ invoke 方法也是 InvocationHandler 的唯一方法）。Invoker 会把所有跟这次调用相关的调用方法名，参数类型列表，参数列表打包，然后利用前面我们分析过的 Client，通过 socket 传递到服务器端。就是说，你在 proxy 类上的任何调用，都通过 Client 发送到远方的服务器上。

Invoker 使用 Invocation。Invocation 封装了一个远程调用的所有相关信息，它的主要属性有：methodName，调用方法名，parameterClasses，调用方法参数的类型列表和 parameters，调用方法参数。注意，它实现了 Writable 接口，可以串行化。

RPC.Server 实现了 org.apache.hadoop.ipc.Server，你可以把一个对象，通过 RPC，升级成为一个服务器。服务器接收到的请求（通过 Invocation），解串行化以后，就变成了方法名，方法参数列表和参数列表。利用 Java 反射，我们就可以调用对应的对象的方法。调用的结果再通过 socket，返回给客户端，客户端把结果解包后，就可以返回给 Dynamic Proxy 的使用者了。

## Hadoop 源代码分析（九）

一个典型的 HDFS 系统包括一个 NameNode 和多个 DataNode。NameNode 维护名字空间；而 DataNode 存储数据块。

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。所以我们可以简单地认为 DataNode 上存了数据块 ID 和数据块内容，以及他们的映射关系。

一个 HDFS 集群可能包含上千 DataNode 节点，这些 DataNode 定时和 NameNode 通信，接受 NameNode 的指令。为了减轻 NameNode 的负担，NameNode 上并不永久保存那个 DataNode 上有那些数据块的信息，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 保持心跳。心跳的返回其还包含了 NameNode 对 DataNode 的一些命令，如删除数据库或者是把数据块复制到另一个 DataNode。应该注意的是：NameNode 不会发起到 DataNode 的请求，在这个通信过程中，它们是严格的客户端/服务器架构。

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 之间还会相互通信，执行数据块复制任务，同时，在客户端做写操作的时候，DataNode 需要相互配合，保证写操作的一致性。

下面我们就来具体分析一下 DataNode 的实现。DataNode 的实现包括两部分，一部分是对本地数据块的管理，另一部分，就是和其他的实体打交道。我们先来看本地数据块管理部分。

安装 Hadoop 的时候，我们会指定对应的数据块存放目录，当我们检查数据块存放目录目录时，我们回发现下面有个叫 dfs 的目录，所有的数据就存放在 dfs/data 里面。

Name	Size	Date
current		2008-11-27 15:41:00
detach		2008-11-14 10:28:00
tmp		2008-11-27 15:41:00
in_use.lock	0	2008-11-14 10:28:00
storage	157	2008-11-14 10:28:00

其中有两个文件，storage 里存的东西是一些出错信息，貌似是版本不对...云云。in\_use.lock 是一个空文件，它的作用是如果需要对整个系统做排斥操作，应用应该获取它上面的一个锁。

接下来是 3 个目录，current 存的是当前有效的数据块，detach 存的是快照（snapshot，目前没有实现），tmp 保存的是一些操作需要的临时数据块。

但我们进入 current 目录以后，就会发现有一系列的数据块文件和数据块元数据文件。同时还有一些子目录，它们的名字是 subdir0 到 subdir63，子目录下也有数据块文件和数据块元数据。这是因为 HDFS 限定了每个目录存放数据块文件的数量，多了以后会创建子目录来保存。

数据块文件显然保存了 HDFS 中的数据，数据块最大可以到 64M。每个数据块文件都会有对应的数据块元数据文件。里面存放的是数据块的校验信息。下面是数据块文件名和它的元数据文件名的例子：

blk\_3148782637964391313

blk\_3148782637964391313\_242812.meta

上面的例子中，3148782637964391313 是数据块的 ID 号，242812 是数据块的版本号，用于一致性检查。

在 current 目录下还有下面几个文件：

VERSION，保存了一些文件系统的元信息。

dncp\_block\_verification.log.curr 和 dncp\_block\_verification.log.prev，它记录了一些 DataNode 对文件系定时统做一致性检查需要的信息。

## Hadoop 源代码分析 (一零)

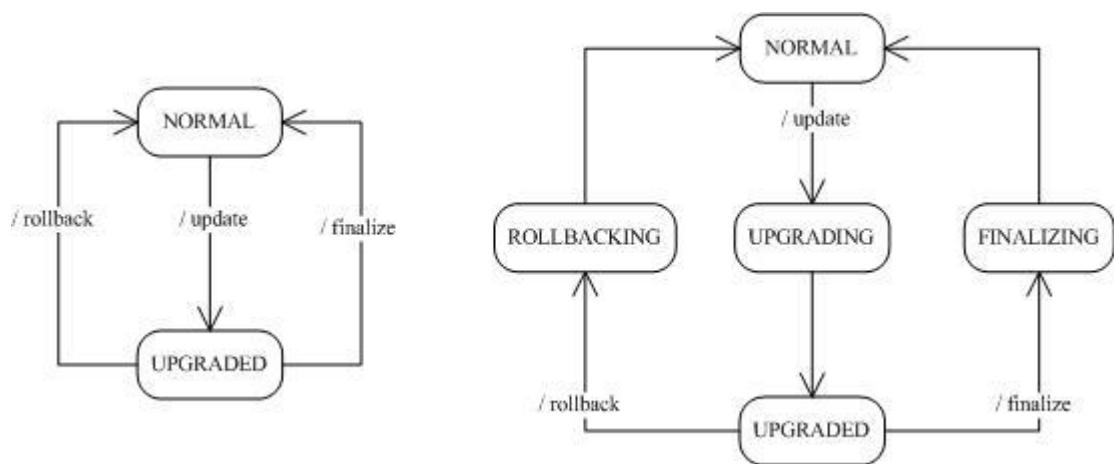
在继续分析 DataNode 之前，我们有必要看一下系统的工作状态。启动 HDFS 的时候，我们可以选择以下启动参数：

- FORMAT("-format")：格式化系统
- REGULAR("-regular")：正常启动
- UPGRADE("-upgrade")：升级
- ROLLBACK("-rollback")：回滚
- FINALIZE("-finalize")：提交
- IMPORT("-importCheckpoint")：从 Checkpoint 恢复。

作为一个大型的分布式系统，Hadoop 内部实现了一套升级机制 ([http://wiki.apache.org/hadoop/Hadoop\\_Upgrade](http://wiki.apache.org/hadoop/Hadoop_Upgrade))。upgrade 参数就是为了这个目的而存在的，当然，升级可能成功，也可能失败。如果失败了，那就用 rollback 进行回滚；如果过了一段时间，系统运行正常，那就可以通过 finalize，正式提交这次升级(跟数据库有点像啊)。

importCheckpoint 选项用于 NameNode 发生故障后，从某个检查点恢复。

有了上面的描述，我们得到下面左边的状态图：



大家应该注意到，上面的升级/回滚/提交都不可能一下就搞定，就是说，系统故障时，它可能处于上面右边状态中的某一个。特别是分布式的各个节点上，甚至可能出现某些节点已经升级成功，但有些节点可能处于中间状态的情况，所以 Hadoop 采用类似于数据库事务的升级机制也就不是很奇怪。

大家先理解一下上面的状态图，它是下面我们要介绍 DataNode 存储的基础。

## [Hadoop 源代码分析（一）](#)

我们来看一下升级/回滚/提交时的 DataNode 上会发生什么（在类 DataStorage 中实现）。

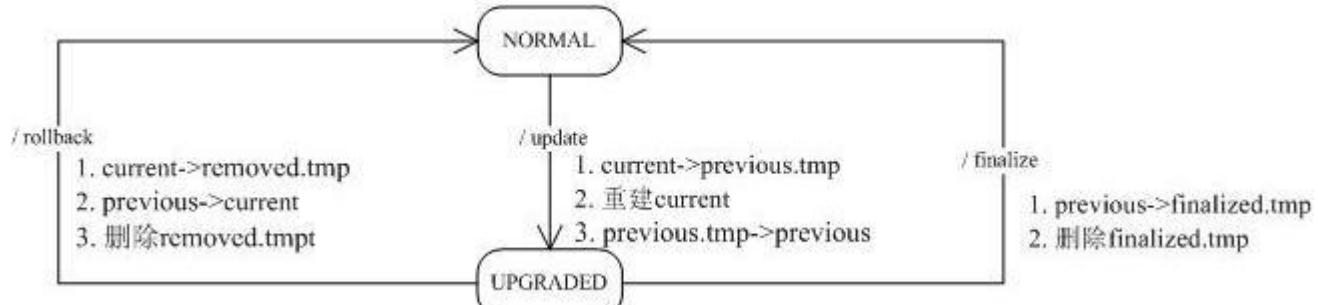
前面我们提到过 VERSION 文件，它保存了一些文件系统的元信息，这个文件在系统升级时，会发生对应的变化。

升级时，NameNode 会将新的版本号，通过 DataNode 的登录应答返回。DataNode 收到以后，会将当前的数据块文件目录改名，从 current 改名为 previous.tmp，建立一个 snapshot，然后重建 current 目录。重建包括重建 VERSION 文件，重建对应的子目录，然后建立数据块文件和数据块元数据文件到 previous.tmp 的硬连接。建立硬连接意味着在系统中只保留一份数据块文件和数据块元数据文件，current 和 previous.tmp 中的相应文件，在存储中，只保留一份。当所有的这些工作完成以后，会在 current 里写入新的 VERSION 文件，并将 previous.tmp 目录改名为 previous，完成升级。

了解了升级的过程以后，回滚就相对简单。因为说有的旧版本信息都保存在 previous 目录里。回滚首先将 current 目录改名为 removed.tmp，然后将 previous 目录改名为 current，最后删除 removed.tmp 目录。

提交的过程，就是将上面的 previous 目录改名为 finalized.tmp，然后启动一个线程，将该目录删除。

下图给出了上面的过程：

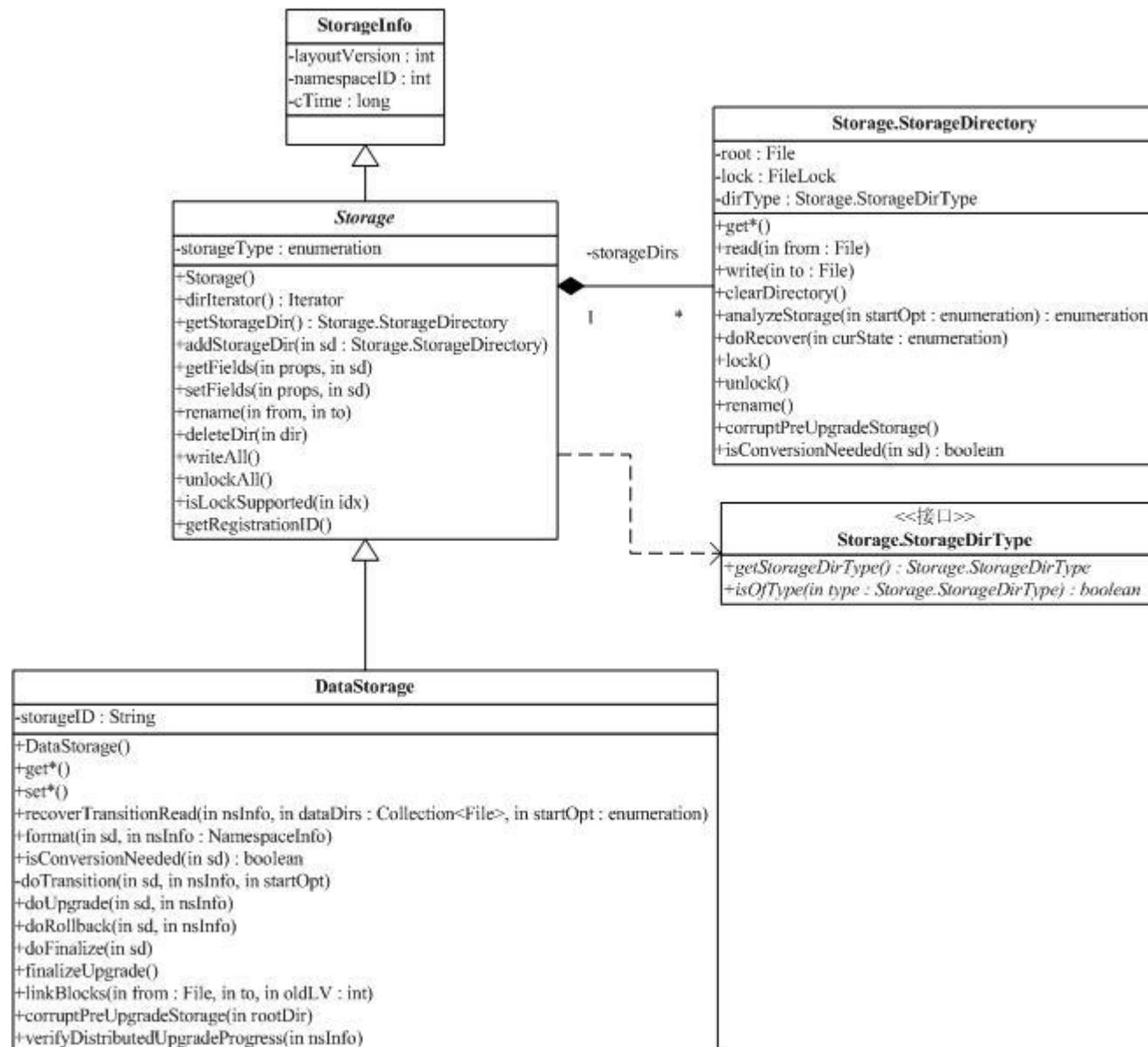


需要注意的是，HDFS 的升级，往往只是支持从某一个特点的老版本升级到当前版本。回滚时能够恢复到的版本，也是 previous 中记录的版本。

下面我们继续分析 DataNode。

文字分析完 DataNode 存储在文件上的数据以后，我们来看一下运行时对应的数据结构。从大到小，Hadoop 中最大的结构是 Storage，最小的结构，在 DataNode 上是 block。

类 Storage 保存了和存储相关的信息，它继承了 StorageInfo，应用于 DataNode 的 DataStorage，则继承了 Storage，总体类图如下：



StorageInfo 包含了 3 个字段，分别是 layoutVersion：版本号，如果 Hadoop 调整文件结构布局，版本号就会修改，这样可以保证文件结构和应用一致。namespaceID 是 Storage 的 ID，cTime，creation time。

和 StorageInfo 相比，Storage 就是个大家伙了。

Storage 可以包含多个根（参考配置项 dfs.data.dir 的说明），这些根通过 Storage 的内部类 StorageDirectory 来表示。StorageDirectory 中最重要的方法是 analyzeStorage，它将根据系统启动时的参数和我们上面提到的一些判断条件，返回系统现在的状态。StorageDirectory 可能处于以下的某一个状态（与系统的工作状态一定的对应）：

NON\_EXISTENT：指定的目录不存在；

NOT\_FORMATTED：指定的目录存在但未被格式化；

COMPLETE\_UPGRADE：previous.tmp 存在，current 也存在

RECOVER\_UPGRADE：previous.tmp 存在，current 不存在

COMPLETE\_FINALIZE：finalized.tmp 存在，current 也存在

COMPLETE\_ROLLBACK : removed.tmp 存在 , current 也存在 , previous 不存在

RECOVER\_ROLLBACK : removed.tmp 存在 , current 不存在 , previous 存在

COMPLETE\_CHECKPOINT : lastcheckpoint.tmp 存在 , current 也存在

RECOVER\_CHECKPOINT : lastcheckpoint.tmp 存在 , current 不存在

NORMAL : 普通工作模式。

StorageDirectory 处于某些状态是通过发生对应状态改变需要的工作文件夹和正常工作的 current 夹来进行判断。状态改变需要的工作文件夹包括 :

previous : 用于升级后保存以前版本的文件

previous.tmp : 用于升级过程中保存以前版本的文件

removed.tmp : 用于回滚过程中保存文件

finalized.tmp : 用于提交过程中保存文件

lastcheckpoint.tmp : 应用于从 NameNode 中 , 导入一个检查点

previous.checkpoint : 应用于从 NameNode 中 , 结束导入一个检查点

有了这些状态 , 就可以对系统进行恢复 ( 通过方法 doRecover ) 。恢复的动作如下 ( 结合上面的状态转移图 ) :

COMPLETE\_UPGRADE : mv previous.tmp -> previous

RECOVER\_UPGRADE : mv previous.tmp -> current

COMPLETE\_FINALIZE : rm finalized.tmp

COMPLETE\_ROLLBACK : rm removed.tmp

RECOVER\_ROLLBACK : mv removed.tmp -> current

COMPLETE\_CHECKPOINT : mv lastcheckpoint.tmp -> previous.checkpoint

RECOVER\_CHECKPOINT : mv lastcheckpoint.tmp -> current

我们以 RECOVER\_UPGRADE 为例 , 分析一下。根据升级的过程 ,

1. current->previous.tmp

2. 重建 current

### 3. previous.tmp->previous

当我们发现 previous.tmp 存在，current 不存在，我们知道只需要将 previous.tmp 改为 current，就能恢复到未升级时的状态。

StorageDirectory 还管理着文件系统的元信息，就是我们上面提过 StorageInfo 信息，当然，StorageDirectory 还保存每个具体用途自己的信息。这些信息，其实都存储在 VERSION 文件中，StorageDirectory 中的 read/write 方法，就是用于对这个文件进行读/写。下面是某一个 DataNode 的 VERSION 文件的例子：

### 配置文件代码

1. #Fri Nov 14 10:27:35 CST 2008
2. namespaceID=1950997968
3. storageID=DS-697414267-127.0.0.1-50010-1226629655026
4. cTime=0
5. storageType=DATA\_NODE
6. layoutVersion=-16

对 StorageDirectory 的排他操作需要锁，还记得我们在分析系统目录时提到的 in\_use.lock 文件吗？它就是用来给整个系统加/解锁用的。StorageDirectory 提供了对应的 lock 和 unlock 方法。

分析完 StorageDirectory 以后，Storage 类就很简单了。基本上都是对一系列 StorageDirectory 的操作，同时 Storage 提供一些辅助方法。

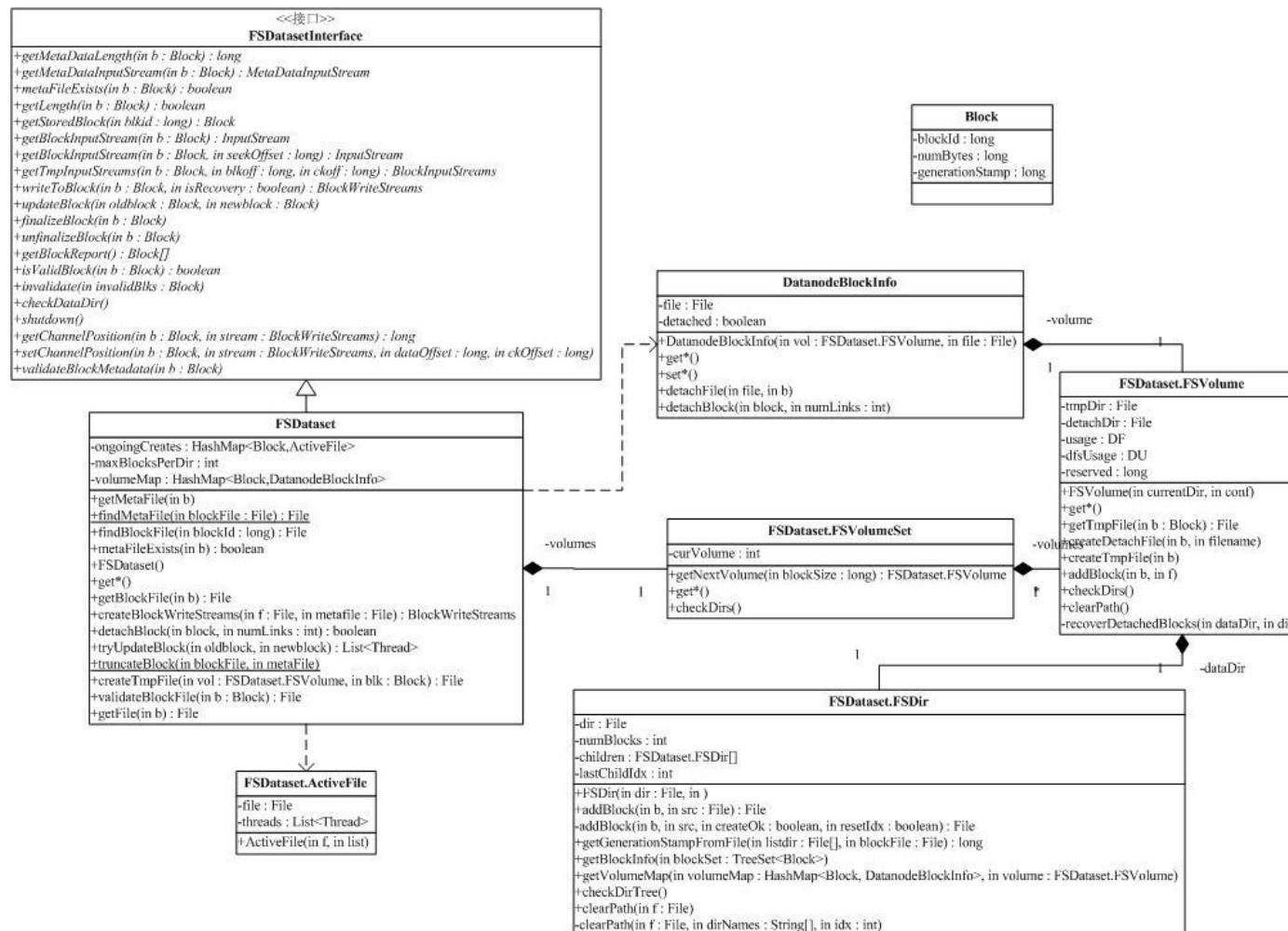
DataStorage 是 Storage 的子类，专门应用于 DataNode。上面我们对 DataNode 的升级/回滚/提交过程，就是对 DataStorage 的 doUpgrade/doRollback/doFinalize 分析得到的。

DataStorage 提供了 format 方法，用于创建 DataNode 上的 Storage，同时，利用 StorageDirectory，DataStorage 管理存储系统的状态。

## [Hadoop 源代码分析（一二）](#)

分析完 Storage 相关的类以后，我们来看下一个家伙，FSDataSet 相关的类。

上面介绍 Storage 时，我们并没有涉及到数据块 Block 的操作，所有和数据块相关的操作，都在 FSDataset 相关的类中进行处理。下面是类图：



Block 是对一个数据块的抽象，通过前面的讨论我们知道一个 Block 对应着两个文件，其中一个存数据，一个存校验信息，如下：

blk\_3148782637964391313

blk\_3148782637964391313\_242812.meta

上面的信息中，blockId 是 3148782637964391313，242812 是数据块的版本号，当然，系统还会保存数据块的大小，在类中是属性 numBytes。Block 提供了一系列的方法来操作对象的属性。

DatanodeBlockInfo 存放的是 Block 在文件系统上的信息。它保存了 Block 存放的卷 ( FSVolume )，文件名和 detach 状态。这里有必要解释一下 detach 状态：我们前面分析过，系统在升级时会创建一个 snapshot，snapshot 的文件和 current 里的数据块文件和数据块元文件是通过硬链接，指向了相同的内容。当我们需要改变 current 里的文件时，如果不进行 detach 操作，那么，修改的内容就会影响 snapshot 里的文件，这时，我们需要将对应的硬链接解除掉。方法很简单，就是在临时文件夹里 复制文件 然后将临时文件改名成为 current 里的对应文件，这样的话，current 里的文件和 snapshot 里的文件就 detach 了。这样的技术，也叫 **copy-on-write**，是一种有效提高系统性能的方法。DatanodeBlockInfo 中的 detachBlock，能够对 Block 对应的数据文件和元数据文件进行 detach 操作。

介绍完类 Block 和 DatanodeBlockInfo 后，我们来看 FSVolumeSet，FSVolume 和 FSDir。我们知道在一个 DataNode 上可以指定多个 Storage 来存储数据块，由于 HDFS 规定了一个目录能存放 Block 的数目，所以一个 Storage 上存在多个目录。

对应的，FSDataset 中用 FSVolume 来对应一个 Storage，FSDir 对应一个目录，所有的 FSVolume 由 FSVolumeSet 管理，FSDataset 中通过一个 FSVolumeSet 对象，就可以管理它的所有存储空间。

FSDir 对应着 HDFS 中的一个目录，目录里存放着数据块文件和它的元文件。FSDir 的一个重要的操作，就是在添加一个 Block 时，根据需要有时会扩展目录结构，上面提过，一个 Storage 上存在多个目录，所有的目录，都对应着一个 FSDir，目录的关系，也由 FSDir 保存。FSDir 的 getBlockInfo 方法分析目录下的所有数据块文件信息，生成 Block 对象，存放到一个集合中。getVolumeMap 方法能，则会建立 Block 和 DatanodeBlockInfo 的关系。以上两个方法，用于系统启动时搜集所有的数据块信息，便于后面快速访问。

FSVolume 对应着是某一个 Storage。数据块文件，detach 文件和临时文件都是通过 FSVolume 来管理的，这个其实很自然，在同一个存储系统上移动文件，往往只需要修改文件存储信息，不需要搬数据。FSVolume 有一个 recoverDetachedBlocks 的方法，用于恢复 detach 文件。和 Storage 的状态管理一样，detach 文件有可能在复制文件时系统崩溃，需要对 detach 的操作进行回复。FSVolume 还会启动一个线程，不断更新 FSVolume 所在文件系统的剩余容量。创建 Block 的时候，系统会根据各个 FSVolume 的容量，来确认 Block 的存放位置。

FSVolumeSet 就不讨论了，它管理着所有的 FSVolume。

HDFS 中，对一个 chunk 的写会使文件处于活跃状态，FSDataset 中引入了类 ActiveFile。ActiveFile 对象保存了一个文件，和操作这个文件的线程。注意，线程有可能有多个。ActiveFile 的构造函数会自动地把当前线程加入其中。

有了上面的基础，我们可以开始分析 FSDataset。FSDataset 实现了接口 FSDatasetInterface。FSDatasetInterface 是 DataNode 对底层存储的抽象。

下面给出了 FSDataset 的关键成员变量：

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

其中，volumes 就是 FSDataset 使用的所有 Storage，ongoingCreates 是 Block 到 ActiveFile 的映射，也就是说，说有正在创建的 Block，都会记录在 ongoingCreates 里。

下面我们讨论 FSDataset 中的方法。

```
public long getMetaDataLength(Block b) throws IOException;
```

得到一个 block 的元数据长度。通过 block 的 ID，找对应的元数据文件，返回文件长度。

```
public MetaDataInputStream getMetaDataInputStream(Block b) throws IOException;
```

得到一个 block 的元数据输入流。通过 block 的 ID，找对应的元数据文件，在上面打开输入流。下面对于类似的简单方法，我们就不再仔细讨论了。

```
public boolean metaFileExists(Block b) throws IOException;
```

判断 block 的元数据的元数据文件是否存在。简单方法。

```
public long getLength(Block b) throws IOException;
```

block 的长度。简单方法。

```
public Block getStoredBlock(long blkid) throws IOException;
```

通过 Block 的 ID , 找到对应的 Block。简单方法。

```
public InputStream getBlockInputStream(Block b) throws IOException;
```

```
public InputStream getBlockInputStream(Block b, long seekOffset) throws IOException;
```

得到 Block 数据的输入流。简单方法。

```
public BlockInputStreams getTmpInputStreams(Block b, long blkoff, long ckoff) throws IOException;
```

得到 Block 的临时输入流。注意 , 临时输入流是指对应的文件处于 tmp 目录中。新创建块时 , 块数据应该写在 tmp 目录中 , 直到写操作成功 , 文件才会被移动到 current 目录中 , 如果失败 , 就不会影响 current 目录了。简单方法。

```
public BlockWriteStreams writeToBlock(Block b, boolean isRecovery) throws IOException;
```

得到一个 block 的输出流。BlockWriteStreams 既包含了数据输出流 , 也包含了元数据 ( 校验文件 ) 输出流 , 这是一个相当复杂的方法。

参数 isRecovery 说明这次写是不是对以前失败的写的一次恢复操作。我们先看正常的写操作流程 : 首先 , 如果输入的 block 是个正常的数据块 , 或当前的 block 已经有线程在写 , writeToBlock 会抛出一个异常。否则 , 将创建相应的临时数据文件和临时元数据文件 , 并把相关信息 , 创建一个 ActiveFile 对象 , 记录到 ongoingCreates 中 , 并创建返回的 BlockWriteStreams 。前面我们已经提过 , 建立新的 ActiveFile 时 , 当前线程会自动保存在 ActiveFile 的 threads 中。

我们以 blk\_3148782637964391313 为例 , 当 DataNode 需要为 Block ID 为 3148782637964391313 创建写流时 , DataNode 创建文件 tmpblk\_3148782637964391313 做为临时数据文件 , 对应的 meta 文件是 tmpblk\_3148782637964391313\_XXXXXX.meta 。其中 XXXXXX 是版本号。

isRecovery 为 true 时 , 表明我们需要从某一次不成功的写中恢复 , 流程相对于正常流程复杂。如果不成功的写是由于提交 ( 参考 finalizeBlock 方法 ) 后的确认信息没有收到 , 先创建一个 detached 文件 ( 备份 ) 。接着 , writeToBlock 检查是否有还有对文件写的线程 , 如果有 , 则通过线程的 interrupt 方法 , 强制结束线程。这就是说 , 如果有线程还在写对应的文件块 , 该线程将被终止。同时 , 从 ongoingCreates 中移除对应的信息。接下来将根据临时文件是否存在 , 创建 / 复用临时数据文件和临时数据元文件。后续操作就和正常流程一样 , 根据相关信息 , 创建一个 ActiveFile 对象 , 记录到 ongoingCreates 中.....

由于这块涉及了一些 HDFS 写文件时的策略 , 以后我们还会继续讨论这个话题。

```
public void updateBlock(Block oldblock, Block newblock) throws IOException;
```

更新一个 block。这也是一个相当复杂的方法。

updateBlock 的最外层是一个死循环，循环的结束条件，是没有任何和这个数据块相关的写线程。每次循环，updateBlock 都会去调用一个叫 tryUpdateBlock 的内部方法。tryUpdateBlock 发现已经没有线程在写这个块，就会跟新和这个数据块相关的信息，包括元文件和内存中的映射表 volumeMap。如果 tryUpdateBlock 发现还有活跃的线程和该块关联，那么，updateBlock 会试图结束该线程，并等在 join 上等待。

```
public void finalizeBlock(Block b) throws IOException;
```

提交（或叫：结束 finalize）通过 writeToBlock 打开的 block，这意味着写过程没有出错，可以正式把 Block 从 tmp 文件夹放到 current 文件夹。

在 FSDataset 中，finalizeBlock 将从 ongoingCreates 中删除对应的 block，同时将 block 对应的 DatanodeBlockInfo，放入 volumeMap 中。我们还是以 blk\_3148782637964391313 为例，当 DataNode 提交 Block ID 为 3148782637964391313 数据块文件时，DataNode 将把 tmpblk\_3148782637964391313 移到 current 下某一个目录，以 subdir12 为例，这是 tmpblk\_3148782637964391313 将会挪到 current/subdir12blk\_3148782637964391313。对应的 meta 文件也在目录 current/subdir12 下。

```
public void unfinalizeBlock(Block b) throws IOException;
```

取消通过 writeToBlock 打开的 block，与 finalizeBlock 方法作用相反。简单方法。

```
public boolean isValidBlock(Block b);
```

该 Block 是否有效。简单方法。

```
public void invalidate(Block invalidBlks[]) throws IOException;
```

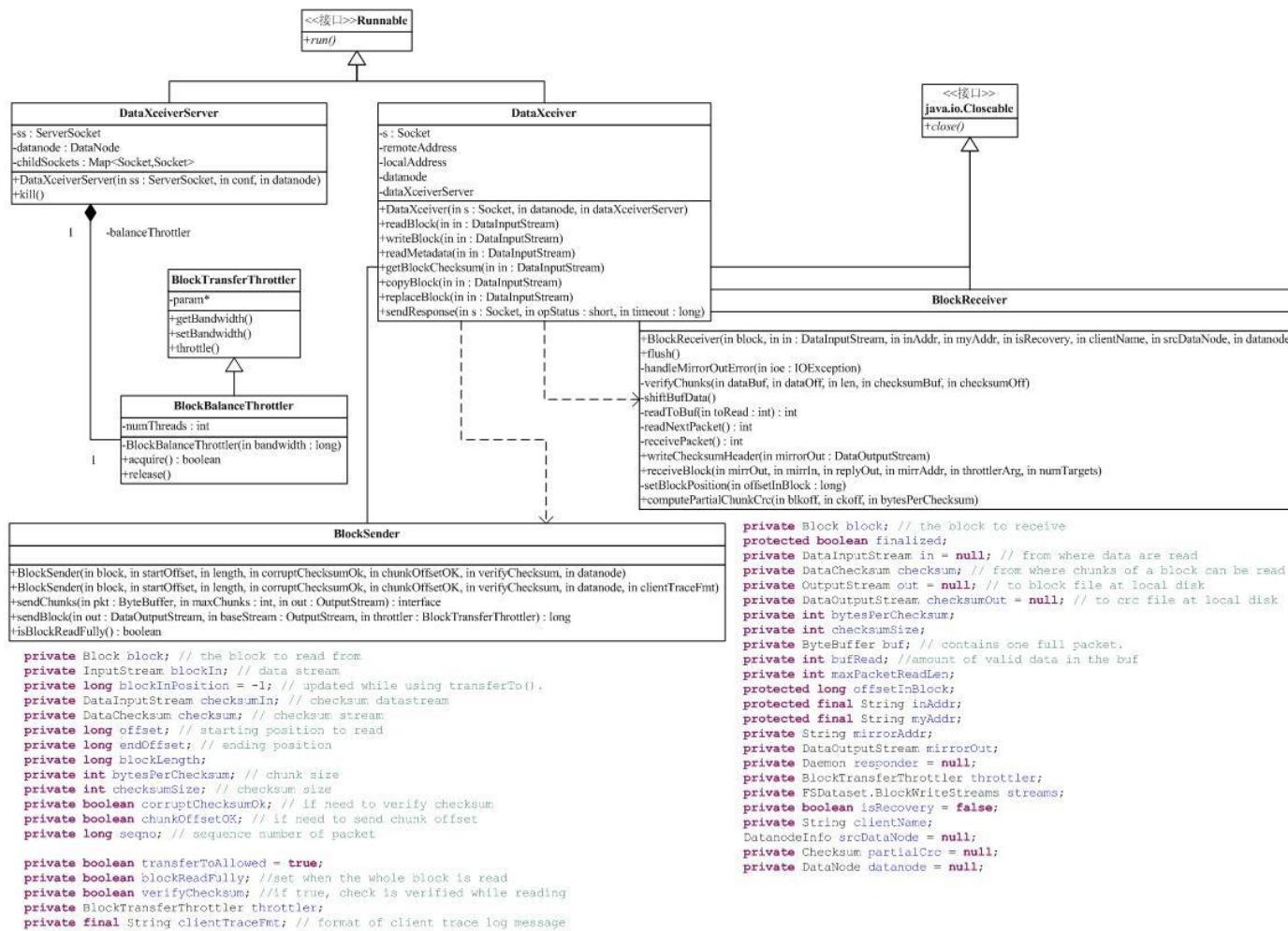
使 block 变为无效。简单方法。

```
public void validateBlockMetadata(Block b) throws IOException;
```

检查 block 的有效性。简单方法。

## Hadoop 源代码分析 (一三)

通过上面的一系列介绍，我们知道了 DataNode 工作时的文件结构和文件结构在内存中的对应对象。下面我们可以来开始分析 DataNode 上的动态行为。首先我们来分析 DataXceiverServer 和 DataXceiver。DataNode 上数据块的接受/发送并没有采用我们前面介绍的 RPC 机制，原因很简单，RPC 是一个命令式的接口，而 DataNode 处理数据部分，往往是一种流式机制。DataXceiverServer 和 DataXceiver 就是这个机制的实现。其中，DataXceiver 还依赖于两个辅助类：BlockSender 和 BlockReceiver。下面是类图：



(为了简单起见，BlockSender 和 BlockReceiver 的成员变量没有进入 UML 模型中 )

DataXceiverServer 很简单，它打开一个端口，然后每接收到一个连接，就创建一个 DataXceiver，服务于该连接，并记录该连接的 socket，对应的实现在 DataXceiverServer 的 run 方法里。当系统关闭时，DataXceiverServer 将关闭监听的 socket 和所有 DataXceiver 的 socket，这样就导致了 DataXceiver 出错并结束线程。

DataXceiver 才是真正干活的地方，目前，DataXceiver 支持的操作总共有六条，分别是：

`OP_WRITE_BLOCK(80)` : 写数据块

`OP_READ_BLOCK(81)` : 读数据块

`OP_READ_METADATA(82)` : 读数据块元文件

`OP_REPLACE_BLOCK(83)` : 替换一个数据块

`OP_COPY_BLOCK(84)` : 拷贝一个数据块

`OP_BLOCK_CHECKSUM(85)` : 读数据块检验码

DataXceiver 首先读取客户端的版本号并检验，然后再读取一个字节的操作码，并转入相关的子程序进行处理。我们先看一下读数据块的过程吧。

首先看输入，下图是读数据块时，客户端发送过来的信息：

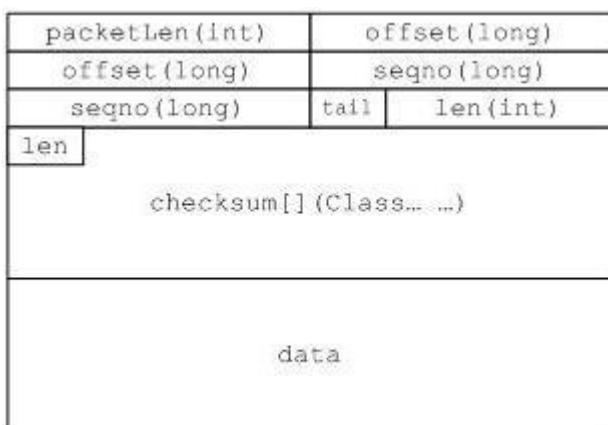
version	81	blockId(long)
blockId		generationStamp(long)
generatio...		startOffset(long)
startOffset		length(long)
length		clientName(String)
clientName	(String... ...)	

包括了要读取的 Block 的 ID , 时间戳 , 开始偏移和读取的长度 , 最后是客户端的名字 (貌似只是在写日志的时候用到了)。根据上面的信息 , 我们可以创建一个 BlockSender , 如果 BlockSender 没有出错 , 返回客户端一个正确指示后 , 否则 , 返回错误码。成功创建 BlockSender 以后 , 就可以开始通过 BlockSender.sendBlock 发送数据。

下面我们就来分析 BlockSender。BlockSender 的构造函数看似很复杂 , 其实就是根据需求 (特别是在处理 checksum 上 , 因为 checksum 是基于块的 ) , 打开相应的数据流。close() 用于释放各种资源 , 如已经打开的数据流。sendBlock 用于发送数据 , 数据发送包括应答头和后续的数据包。应答头如下 ( 包含 DataXceiver 中发送的成功标识 ) :

0(成功)	checksum.header(Class... ...)	offset(long, 可能)
-------	-------------------------------	------------------

然后后面的数据就组织成数据包来发送 , 包结构如下 :



各个字段含义 :

packetLen : 包长度 , 包括包头

offset : 偏移量

seqno : 包序列号

tail : 是否是最后一个包

len : 数据长度

checksum : 检验数据

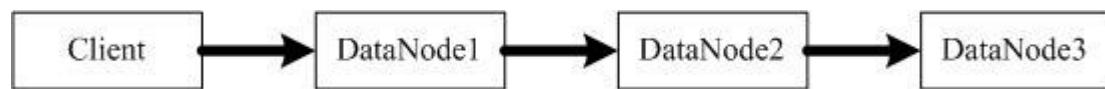
data : 数据块数据

需要注意的 , 在写数据前 , BlockSender 会校验数据 , 保证数据包中的 checksum 和数据的一致性。同时 , 如果数据出错 , 将会有 ChecksumException 抛出。

数据传输结束的标志 , 是一个 packetLen 长度为 0 的包。客户端可以返回一个两字节的应答 *OP\_STATUS\_CHECKSUM\_OK(5)*

[Hadoop 源代码分析 \(一四\)](#)

继续 DataXceiver 分析，下一块硬骨头是写数据块。HDFS 的写数据操作，比读数据复杂 N 多倍。读数据的时候，只需要在多个数据块文件的选一个读，就可以了；但是，写数据需要同时写到多个数据块文件上，这就比较复杂了。HDFS 实现了 Google 写文件时的机制，如下图：



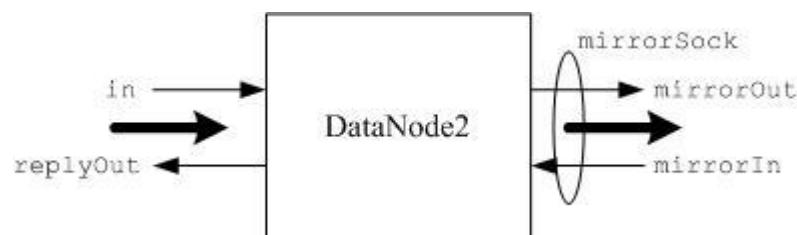
数据流从客户端开始，流经一系列的节点，到达最后一个 DataNode。图中的所有 DataNode 只需要写一次硬盘，DataNode1 和 DataNode2 会将从 socket 上接受到的数据，直接写到到下个节点的 socket 上。

我们来看一下写数据块的请求。

version	80	blockId(long)
blockId		generationStamp(long)
generatio...		pipelineSize(int) isRecovery
client(String... ...)		
hasSrcDataNode	srcDataNode(Class, optional... ...)	
numTargets(int)		targets[1]
	targets[1] (Class... ...)	
	targets[...]	
	checksum.header(Class... ...)	

首先是客户端的版本号和一个字节的操作码，接下来是我们熟悉的 blockId 和 generationStamp。参数 pipelineSize 是整个数据流链的长度，以上面为例，pipelineSize=3。isRecovery 指示这次写是否是一次恢复操作，还记得我们在讨论 FSDataset.writeToBlock 时的那个参数吗？isRecovery 来自客户端。client 是客户端的名字，就是发起请求的节点名，**需要特别注意的是，如果是从 NameNode 来的复制请求，client 为空**。hasSrcDataNode 是一个标志位，如果被设置，表明源节点是个 DataNode，接下来读取的数据就是 DataNode 的信息。numTargets 是目标节点的数目，包括当前节点，以上面的图为例，DataNode1 上这个参数值为 3，到了 DataNode3，就只有 1 了。targets 包含了目标节点的相关信息，根据这些信息，就可以创建到它们上面的 socket 连接。targets 后跟着的是校验头。

writeBlock 最开始是处理上面提到的消息包，然后创建一个 BlockReceiver。接下来就是创建一堆用于读写的流，如下图（图中除了 in 外，都是在 writeBlock 中创建，这个图还不涉及在 BlockReceiver 对本地文件读写的流）：

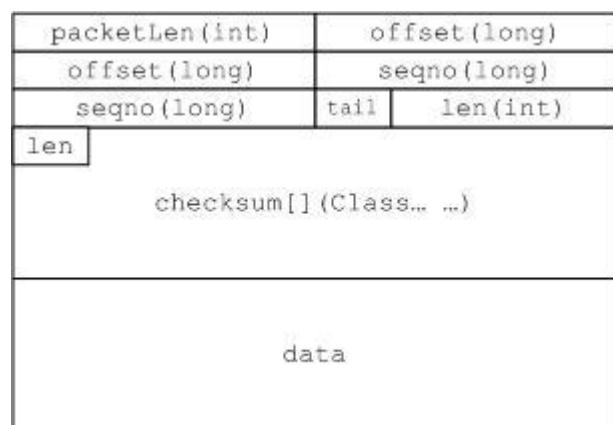


在进行实际的数据写之前，上面的这些流会被建立起来（也就是说，DataNode1 到 DataNode3 都可写以后，才开始处理写数据）。如果其中某一个点出错了，那么，出错的节点名会通过 mirrorIn 发送回来，一直沿着这条链，传播到客户端。

如果一切正常，那么，BlockReceiver.receiveBlock 就开始干活了。

BlockReceiver 的构造函数会创建写数据块和校验数据的输出流。剩下的就交给 receiveBlock 这个家伙了。首先 receiveBlock 会再启动一个线程(一般来说 ,BlockReceiver 就跑在它自己的线程上 ),用于处理应答( 内部类 PacketResponder 定义了该线程 )，然后就不断调用 receivePacket 读数据。

数据是以分块的形式传送，格式和读 Block 的时候是一样的。如下图 ( 很奇怪，为啥不抽象为类 ) :



注意：如果当前 DataNode 处于数据流的中间，该数据包会发送到下一个节点。

接下来的处理，就是处理数据和校验，并分别写到数据块文件和数据块元数据文件。如果出错，抛出的异常会导致 receiveBlock 关闭相关的输出流，并终止传输。注意，数据校验出错还会上报到 NameNode 上。

PacketResponder 用于处理应答。也就是上面讲的 mirrorIn 和 replyOut。PacketResponder 里有一个队列 ackQueue， receivePacket 每收到一个包，都会往队列里添加一项。PacketResponder 的 run 方法，根据工作的 DataNode 所处的位置，行为不一样。

最后一个 DataNode 由于没有后续节点，PacketResponder 的 ackQueue 每收到一项，表明对应的数据块已经处理完毕，那么就可以发送成功应答。如果该应答是最后一个包的，PacketResponder 会关闭相关的输出流，并提交（前面讲 FSDataset 时后我们讨论过的 finalizeBlock 方法）。

如果 DataNode 有后续节点，那么，它必须等到后续节点的成功应答，才可以发送应答到它前面的节点。

PacketResponder 的 run 方法还引入了心跳机制，用于检测连接是否还存在。

注意 所有改变 DataNode 的操作，需要把信息更新到 NameNode 上，这是通过 DataNode.notifyNamenodeReceivedBlock 方法，然后通过 DataNode 统一发送到 NameNode 上。

## [Hadoop 源代码分析 \(一五\)](#)

DataXceiver 支持的的 6 条操作，我们已经分析完最重要的两条。剩下的分别是：

*OP\_READ\_METADATA* (82) : 读数据块元文件

*OP\_REPLACE\_BLOCK* (83) : 替换一个数据块

*OP\_COPY\_BLOCK* (84) : 拷贝一个数据块

## *OP\_BLOCK\_CHECKSUM*(85) : 读数据块检验码

我们逐个讨论。

读数据块元文件的请求如图 ( 操作码 82 ) :

version	82	blockId(long)
blockId		generationStamp(long)
generatio...		

version	83	blockId(long)
blockId		generationStamp(long)
generatio...		sourceID(String)
sourceID(String... ...)		proxySource
proxySource(Class... ...)		

version	84	blockId(long)
blockId		generationStamp(long)
generatio...		

version	85	blockId(long)
blockId		generationStamp(long)
generatio...		

应答很简单，应答码 ( 如 *OP\_STATUS\_SUCCESS* ) , 文件长度 ( int ) , 数据。

拷贝数据块和替换数据块是一对相对应操作。

替换数据块的请求如图 ( 操作码 83 ) 。这个比起上面的读数据块元文件请求，有点复杂。替换一个数据块是系统平衡操作的一部分，用于接收一个数据块。它和普通的数据块写的差别是，它只发生在两个节点上，一个写，一个读，而不需要建立数据链。我们可以比较一下它们在创建 *BlockReceiver* 对象时的差别：

## Java 代码

1. `blockReceiver = new BlockReceiver(block, proxyReply,`
2.       `proxySock.getRemoteSocketAddress().toString(),`
3.       `proxySock.getLocalSocketAddress().toString(),`
4.       `false, "", null, datanode); //OP_REPLACE_BLOCK`
5. `blockReceiver = new BlockReceiver(block, in,`
6.       `s.getRemoteSocketAddress().toString(),`
7.       `s.getLocalSocketAddress().toString(),`
8.       `isRecovery, client, srcDataNode, datanode); //OP_WRITE_BLOCK`

首先，proxyReply 和 in 不一样，这是因为发起请求的节点和提供数据的节点并不是同一个。写数据块发起请求方也提供数据，替换数据块请求方不提供数据，而是提供了一个数据源（ proxySource 参数），由 replaceBlock 发起一个拷贝数据块的请求，建立数据源。对于拷贝数据块操作，isRecovery=false，client=""，srcDataNode=null。注意，我们在分析 BlockReceiver 是，讨论过 client="" 的情况，就是应用于这种场景。

在创建 BlockReceiver 对象前，需要利用下面介绍的拷贝数据块的请求建立到数据源的 socket 连接并发送拷贝数据块请求。然后通过 BlockReceiver.receiveBlock 接收数据。任务成功后将结果通知 notifyNamenodeReceivedBlock。

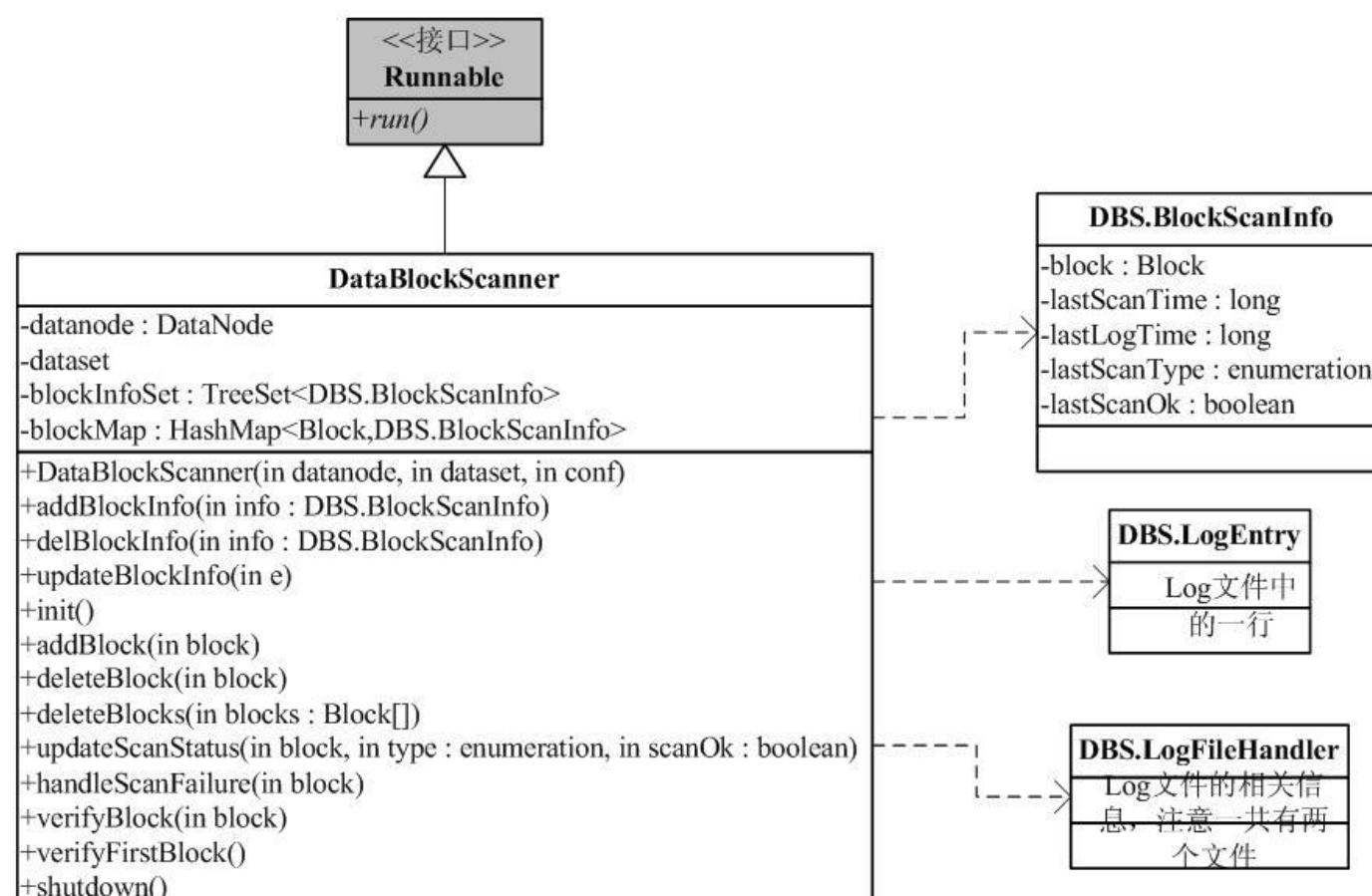
拷贝数据块的请求如图（操作码 84）。和读数据块操作请求类似，但是读取的是整个数据块，所以少了很多参数。

读数据块检验码的请求如图（操作码 85）。它能够读取某个数据块的检验码和的 MD5 结果，实现的方法很简单。

## Hadoop 源代码分析（一六）

通过上面的讨论，DataNode 上的读/写流程已经基本清楚了。我们来看下一个非主流流程，

DataBlockScanner 用于定时对数据块文件进行校验。类图如下：



DataBlockScanner 拥有它单独的线程，能定时地从目前 DataNode 管理的数据块文件进行校验。其实最重要的方法就是 verifyBlock，我们来看这个方法最关键的地方：

### Java 代码

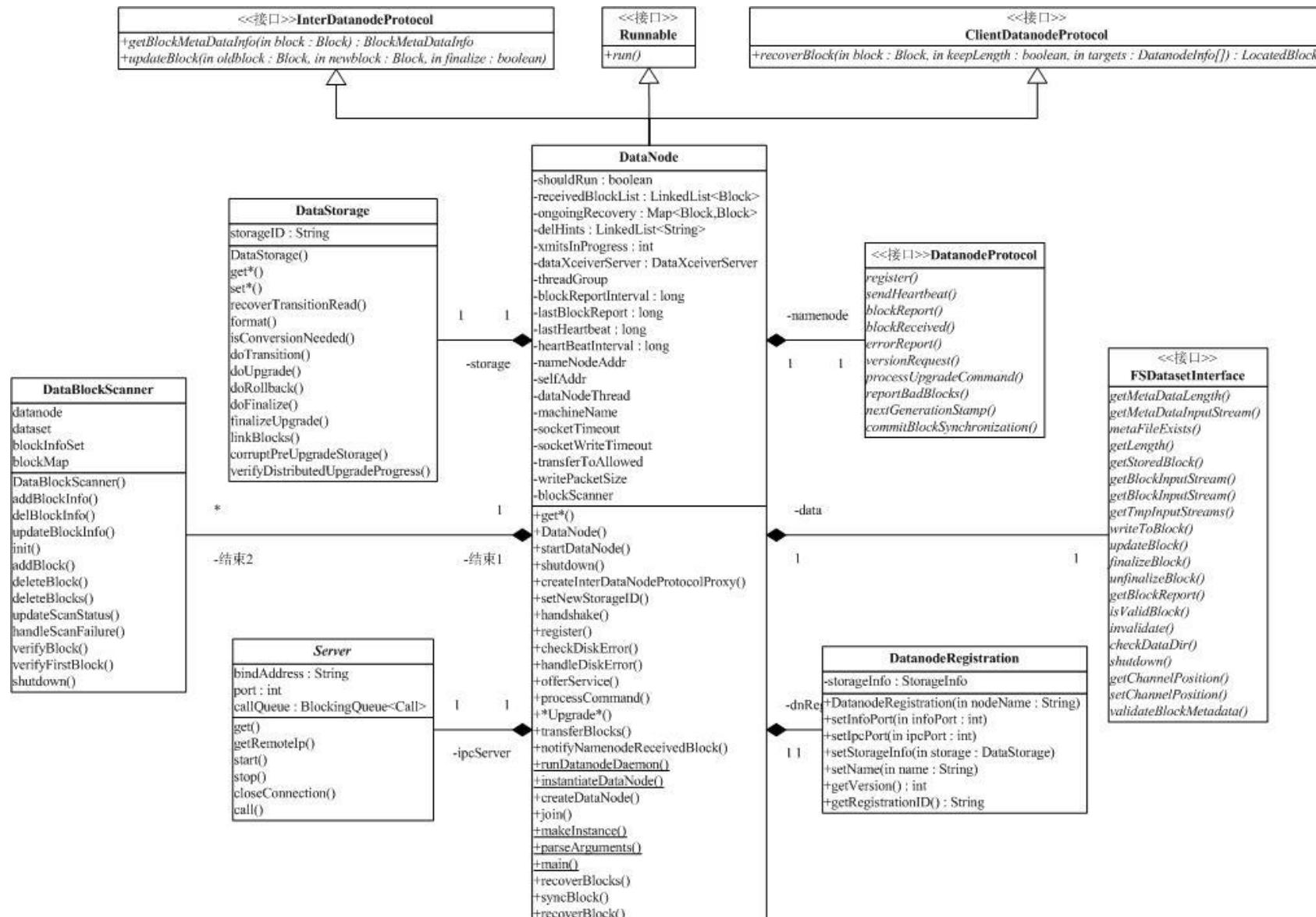
1. blockSender = new BlockSender(block, 0, -1, false, false, true, datanode);
2. DataOutputStream out = new DataOutputStream(new IOUtils.NullOutputStream());
3. blockSender.sendBlock(out, null, throttler);

校验利用了 BlockSender，因为我们知道 BlockSender 中，发送数据的同时，会对数据进行校验。verifyBlock 只需要读一个 Block 到一个空输出设备（NullOutputStream），如果有异常，那么校验失败，如果正常，校验成功。

DataBlockScanner 其他的辅助方法用于对 DataBlockScanner 管理的数据块文件信息进行增加/删除，排序操作。同时，校验的信息还会保持在 Storage 上，保存在 dncp\_block\_verification.log.curr 和 dncp\_block\_verification.log.prev 中。

## Hadoop 源代码分析 (一七)

周围的障碍扫清以后，我们可以开始分析类 DataNode。类图如下：



**public class DataNode extends Configured**

**implements InterDataNodeProtocol, ClientDataNodeProtocol, FSConstants, Runnable**

上面给出了 DataNode 的继承关系，我们发现，DataNode 实现了两个通信接口，其中 ClientDataNodeProtocol 是用于和 Client 交互的，InterDataNodeProtocol，就是我们前面提到的 DataNode 间的通信接口。ipcServer（类图的左下方）是 DataNode 的一个成员变量，它启动了一个 IPC 服务，这样，DataNode 就能提供 ClientDataNodeProtocol 和 InterDataNodeProtocol 的能力了。

我们从 main 函数开始吧。这个函数很简单，调用了 createDataNode 的方法，然后就等着 DataNode 的线程结束。

createDataNode 首先调用 instantiateDataNode 初始化 DataNode，然后执行 runDatanodeDaemon。

runDatanodeDaemon 会向 NameNode 注册，如果成功，才启动 DataNode 线程，DataNode 就开始干活了。

初始化 DataNode 的方法 instantiateDataNode 会读取 DataNode 需要的配置文件，同时读取配置的 storage 目录（可能有多个，看 storage 的讨论部分），然后把这两参数送到 makeInstance 中，makeInstance 会先检查目录（存在，是目录，可读，可写），然后调用：

```
new DataNode(conf, dirs);
```

接下来控制流就到了构造函数上。构造函数调用 startDataNode，完成和 DataNode 相关的初始化工作（注意，DataNode 工作线程不在这个函数里启动）。首先是初始化一堆的配置参数，什么 NameNode 地址，socket 参数等等。然后，向 NameNode 请求配置信息（DatanodeProtocol.versionRequest），并检查返回的 NamespaceInfo 和本地的版本是否一致。

正常情况的下一步是检查文件系统的状态并做必要的恢复，初始化 FSDataset（到这个时候，上面图中 storage 和 data 成员变量已经初始化）。

然后，找一个端口并创建 DataXceiverServer（run 方法里启动），创建 DataBlockScanner（根据需要在 offerService 中启动，只启动一次），创建 DataNode 上的 HttpServer，启动 ipcServer。这样就结束了 DataNode 相关的初始化工作。

在启动 DataNode 工作线程前，DataNode 需要向 NameNode 注册。注册信息在初始化的时候已经构造完毕，包括 DataXceiverServer 端口，ipcServer 端口，文件布局版本号等重要信息。注册成功后就可以启动 DataNode 线程。

DataNode 的 run 方法 循环里有两种选择，升级（暂时不讨论）/正常工作。我们来看正常工作的 offerService 方法。offerService 也是个循环，在循环里，offerService 会定时向 NameNode 发送心跳，报告系统中 Block 状态的变化，报告 DataNode 现在管理的 Block 状态。发送心跳和 Block 状态报告时，NameNode 会返回一些命令，DataNode 将执行这些命令。

心跳的处理比较简单，以 heartBeatInterval 间隔发送。

Block 状态变化报告，会利用保存在 receivedBlockList 和 delHints 两个列表中的信息。receivedBlockList 表明在这个 DataNode 成功创建的新的数据块，而 delHints，是可以删除该数据块的节点。如在 DataXceiver 的 replaceBlock 中，有调用：

```
datanode.notifyNamenodeReceivedBlock(block, sourceID)
```

这表明，DataNode 已经从 sourceID 上接收了一个 Block，sourceID 上对应的 Block 可以删除了（这个场景出现在当系统需要做负载均衡时，Block 在 DataNode 之间拷贝）。

Block 状态变化报告通过 NameNode.blockReceived 来报告。

Block 状态报告也比较简单，以 blockReportInterval 间隔发送。

心跳和 Block 状态报告可以返回命令，这也是 NameNode 先 DataNode 发起请求的唯一方法。我们来看一下都有那些命令：

*DNA\_TRANSFER*：拷贝数据块到其他 DataNode

*DNA\_INVALIDATE*：删除数据块（简单方法）

*DNA\_SHUTDOWN*：关闭 DataNode（简单方法）

*DNA\_REGISTER*：DataNode 重新注册（简单方法）

*DNA\_FINALIZE*：提交升级（简单方法）

*DNA\_RECOVERBLOCK*：恢复数据块

拷贝数据块到其他 DataNode 由 transferBlocks 方法执行。注意，返回的命令可以包含多个数据块，每一个数据块可以包含多个目标地址。transferBlocks 方法将为每一个 Block 启动一个 DataTransfer 线程，用于传输数据。

DataTransfer 是一个 DataNode 的内部类，它利用我们前面介绍的 OP\_WRITE\_BLOCK 写数据块操作，发送数据到多个目标上面。

恢复数据块和 NameNode 的租约（lease）恢复有关，我们后面再讨论。

## Hadoop 源代码分析（一八）

DataNode 的介绍基本告一段落。我们开始来分析 NameNode。相比于 DataNode，NameNode 比较复杂。系统中只有一个 NameNode，作为系统文件目录的管理者和“inode 表”（熟悉 UNIX 的同学们应该了解 inode）。为了高可用性，系统中还存在着从 NameNode。

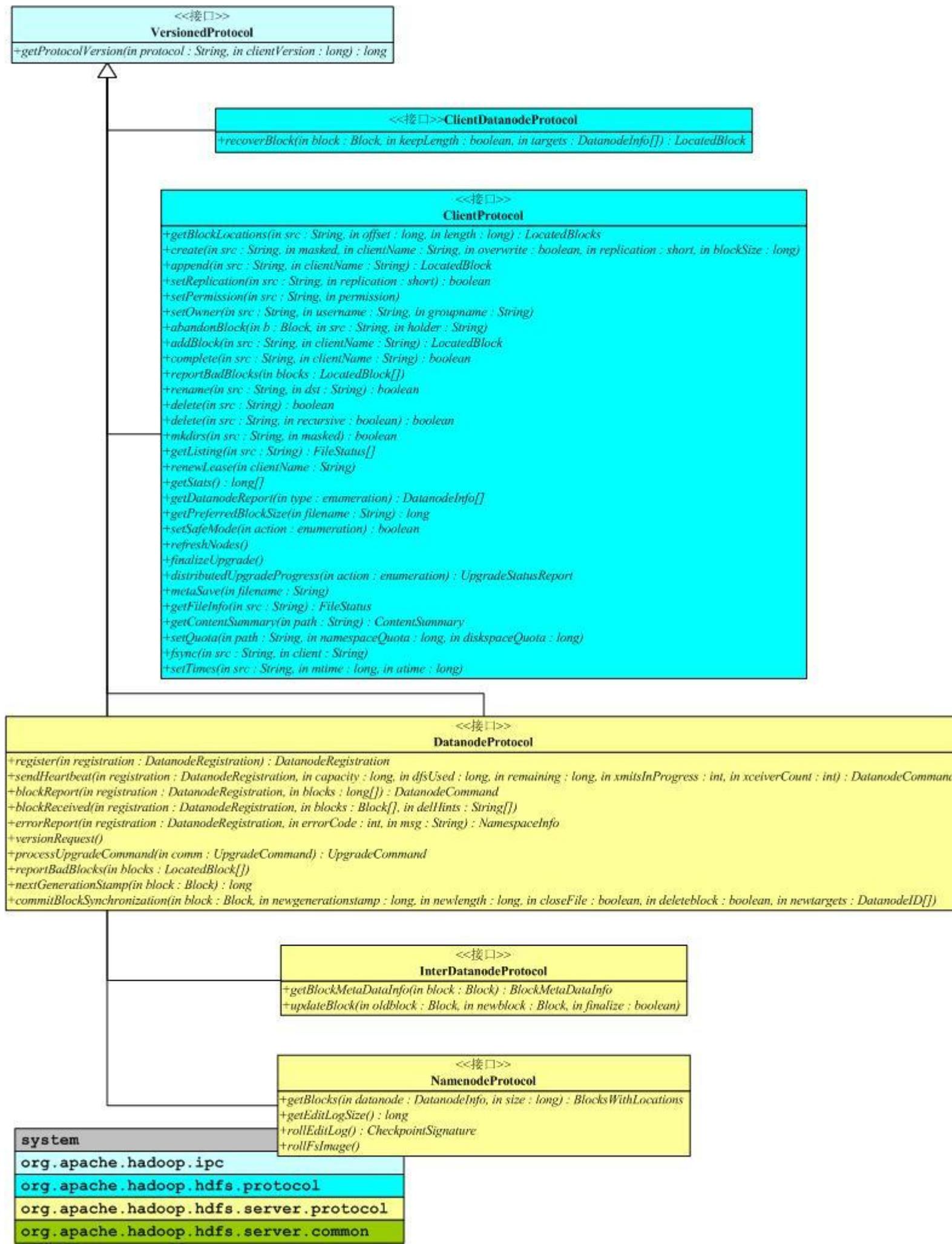
先前我们分析 DataNode 的时候，关注的是数据块。NameNode 作为 HDFS 中文件目录和文件分配的管理者，它保存的最重要信息，就是下面两个映射：

文件名à数据块

数据块àDataNode 列表

其中，文件名à数据块保存在磁盘上（持久化）；但 NameNode 上不保存数据块àDataNode 列表，该列表是通过 DataNode 上报建立起来的。

下图包含了 NameNode 和 DataNode 往外暴露的接口，其中，DataNode 实现了 InterDatanodeProtocol 和 ClientDatanodeProtocol，剩下的，由 NameNode 实现。

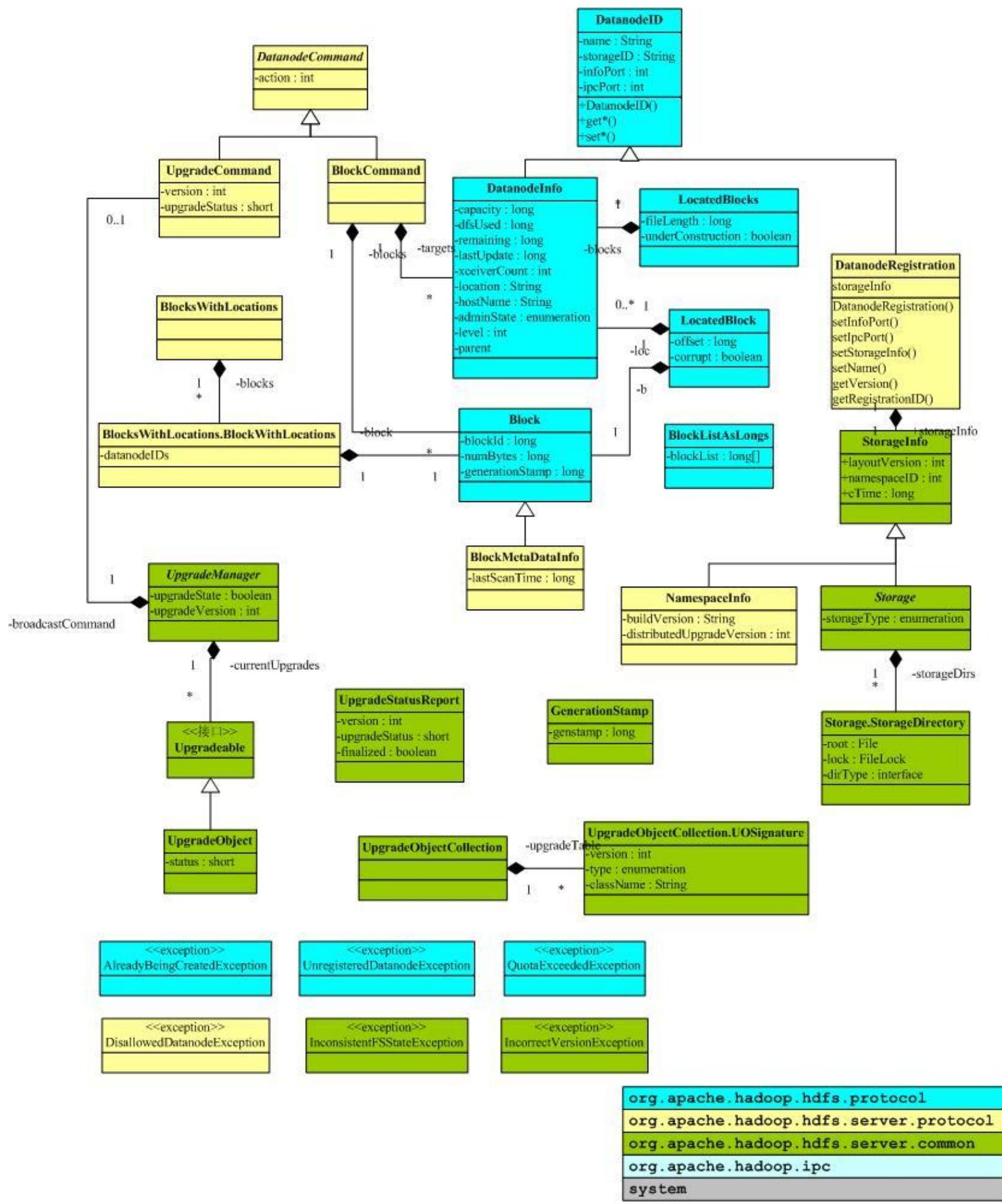


ClientProtocol 提供给客户端，用于访问 NameNode。它包含了文件角度上的 HDFS 功能。和 GFS 一样，HDFS 不提供 POSIX 形式的接口，而是使用了一个私有接口。一般来说，程序员通过 org.apache.hadoop.fs.FileSystem 来和 HDFS 打交道，不需要直接使用该接口。

DatanodeProtocol：用于 DataNode 向 NameNode 通信，我们已经在 DataNode 的分析过程中，了解部分接口，包括：register，用于 DataNode 注册；sendHeartbeat/blockReport/blockReceived，用于 DataNode 的 offerService 方法中；errorReport 我们没有讨论，它用于向 NameNode 报告一个错误的 Block，用于 BlockReceiver 和 DataBlockScanner；nextGenerationStamp 和 commitBlockSynchronization 用于 lease 管理，我们在后面讨论到 lease 时，会统一说明。

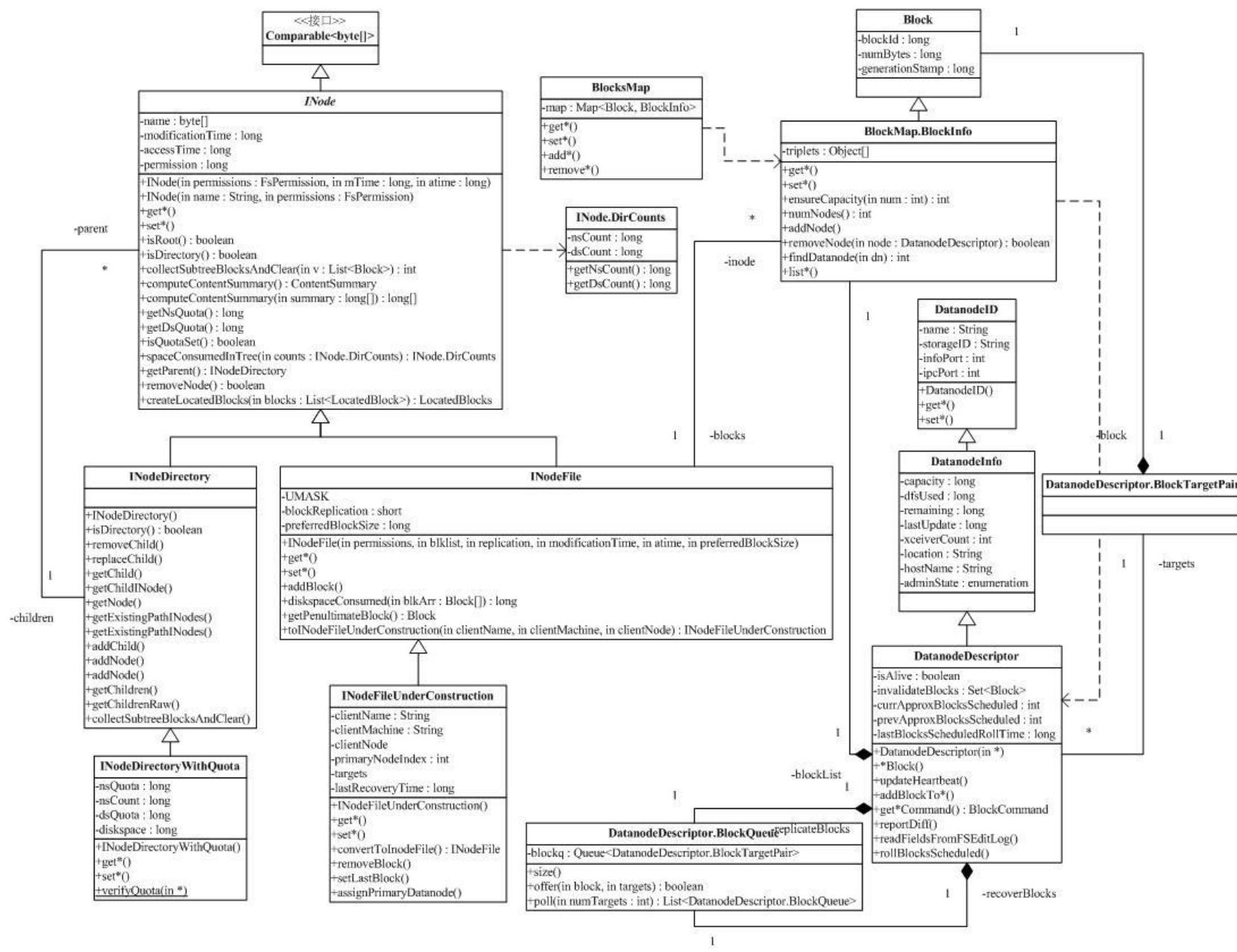
NamenodeProtocol 用于从 NameNode 到 NameNode 的通信。

下图补充了接口里使用的数据的关系。



## Hadoop 源代码分析（一九）

我们先分析 `INode*`, 类 `INode*` 抽象了文件层次结构。如果我们将文件系统进行面向对象的抽象，一定会得到和下面一样类似的结构图（类 `INode*`）：



**INode** 是一个抽象类，它的两个子类，分别对应着目录( **INodeDirectory** )和文件( **INodeFile** )。 **INodeDirectoryWithQuota**，如它的名字隐含的，是带了容量限制的目录。 **INodeFileUnderConstruction**，抽象了正在构造的文件，当我们需要在 HDFS 中创建文件的时候，由于创建过程比较长，目录系统会维护对应的信息。

**INode** 中的成员变量有：name，目录/文件名；modificationTime 和 accessTime 是最后的修改时间和访问时间；parent 指向了父目录；permission 是访问权限。HDFS 采用了和 UNIX/Linux 类似的访问控制机制。系统维护了一个类似于 UNIX 系统的组表 ( group ) 和用户表 ( user )，并给每一个组和用户一个 ID，permission 在 **INode** 中是 long 型，它同时包含了组和用户信息。

**INode** 中存在大量的 get 和 set 方法，当然是对上面提到的属性的操作。导出属性，比较重要的有：

`collectSubtreeBlocksAndClear`，用于收集这个 **INode** 所有后继中的 Block；`computeContentSummary` 用于递归计算 **INode** 包含的一些相关信息，如文件数，目录数，占用磁盘空间。

**INodeDirectory** 是 HDFS 管理的目录的抽象，它最重要的成员变量是：

```
private List<INode> children;
```

就是这个目录下的所有目录/文件集合。**INodeDirectory** 也是有大量的 get 和 set 方法，都很简单。**INodeDirectoryWithQuota** 进一步加强了 **INodeDirectory**，限制了 **INodeDirectory** 可以使用的空间（包括 NameSpace 和磁盘空间）。

**INodeFile** 是 HDFS 中的文件，最重要的成员变量是：

```
protected BlockInfo blocks[] = null;
```

这是这个文件对应的 Block 列表，BlockInfo 增强了 Block 类。

INodeFileUnderConstruction 保存了正在构造的文件的一些信息，包括 clientName，这是目前拥有租约的节点名（创建文件时，只有一个节点拥有租约，其他节点配合这个节点工作）。clientMachine 是构造该文件的客户端名称，如果构造请求由 DataNode 发起，clientNode 会保持相应的信息，targets 保存了配合构造文件的所有节点。

上面描述了 INode\*类的关系。下面我们顺便考察一下一些 NameNode 上的数据类。

BlocksMap 保存了 Block 和它在 NameNode 上一些相关的信息。其核心是一个 map : Map<Block, BlockInfo>。BlockInfo 扩展了 Block，保存了该 Block 归属的 INodeFile 和 DatanodeDescriptor，同时还包括了它的前继和后继 Block。有了 BlocksMap，就可以通过 Block 找对应的文件和这个 Block 存放的 DataNode 的相关信息。

接下来我们来分析类 Datanode\*。DatanodeInfo 和 DatanodeID 都定义在包 org.apache.hadoop.hdfs.protocol。DatanodeDescriptor 是 DatanodeInfo 的子类，包含了 NameNode 需要的附加信息。DatanodeID 只包含了一些配置信息，DatanodeInfo 增加了一些动态信息，DatanodeDescriptor 更进一步，包含了 DataNode 上一些 Block 的动态信息。DatanodeDescriptor 包含了内部类 BlockTargetPair，它保存 Block 和对应 DatanodeDescriptor 的关联，BlockQueue 是 BlockTargetPair 队列。

DatanodeDescriptor 包含了两个 BlockQueue，分别记录了该 DataNode 上正在复制（replicateBlocks）和 Lease 恢复（recoverBlocks）的 Block。同时还有一个 Block 集合，保存的是该 DataNode 上已经失效的 Block。DatanodeDescriptor 提供一系列方法，用于操作上面保存的队列和集合。也提供 get\*Command 方法，用于生成发送到 DataNode 的命令。

当 NameNode 收到 DataNode 对现在管理的 Block 状态的汇报时，会调用 reportDiff，找出和现在 NameNode 上的信息差别，以供后续处理用。

readFieldsFromFSEditLog 方法用于从日志中恢复 DatanodeDescriptor。

## Hadoop 源代码分析（二零）

前面我们提过关系：文件名à数据块持久化在磁盘上，所有对目录树的更新和文件名à数据块关系的修改，都必须能够持久化。为了保证每次修改不需要从新保存整个结构，HDFS 使用操作日志，保存更新。

现在我们可以得到 NameNode 需要存储在 Disk 上的信息了，包括：

```
[hadoop@localhost dfs]$ ls -R name
```

```
name:
```

```
current  image  in_use.lock
```

```
name/current:
```

```
edits  fsimage  fstime  VERSION
```

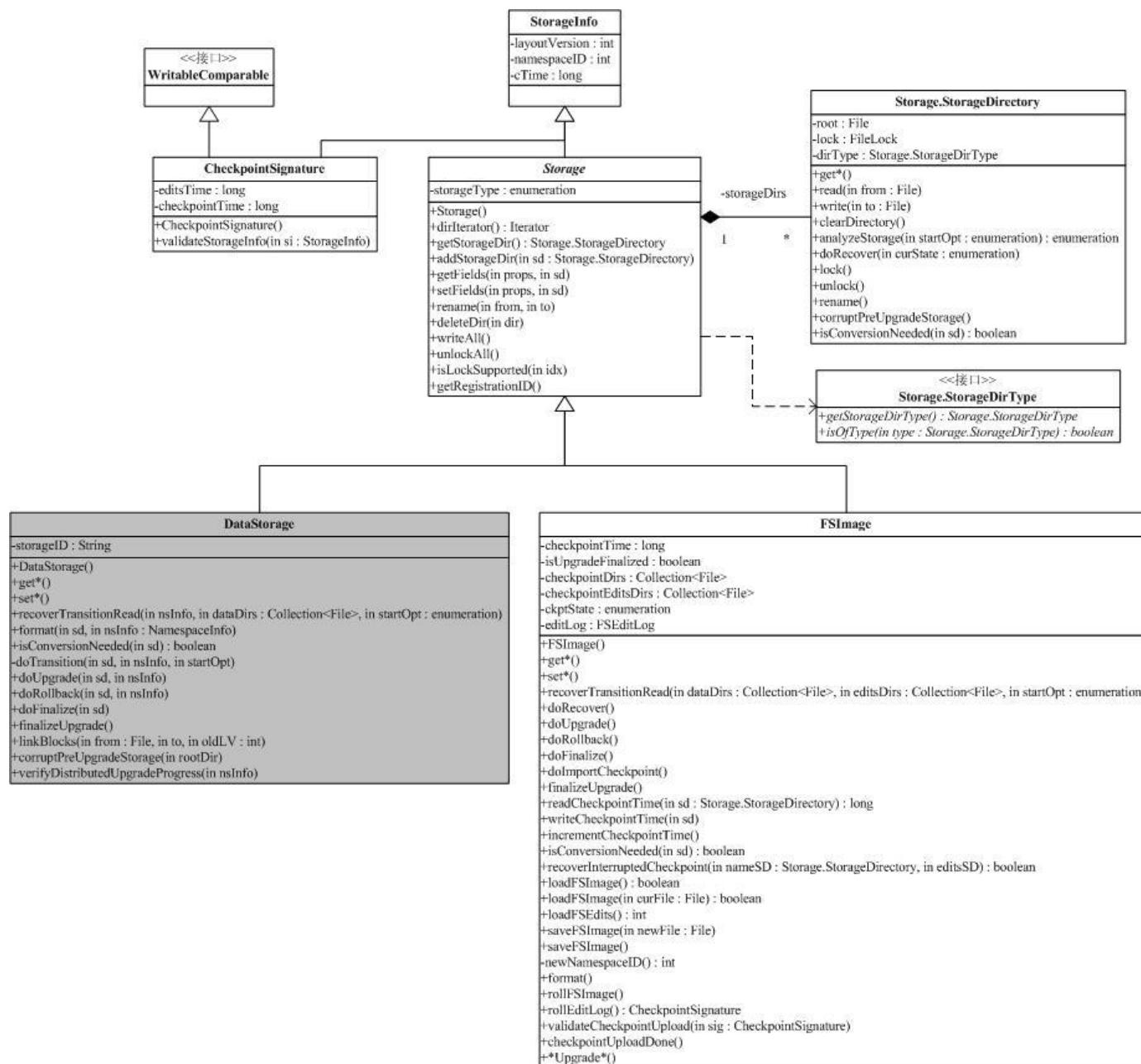
name/image:

fsimage

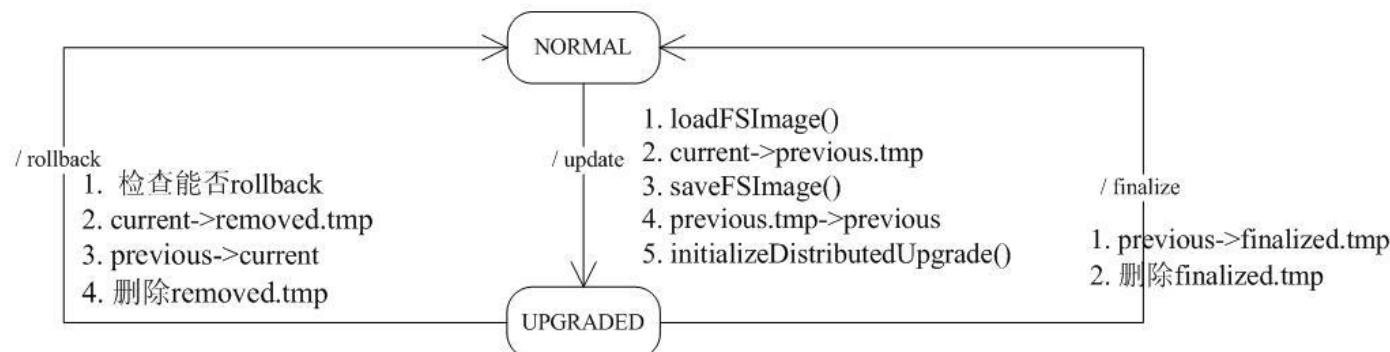
in\_use.lock 的功能和 DataNode 的一致。fsimage 保存的是文件系统的目录树， edits 则是文件树上的操作日志，fstime 是上一次新打开一个操作日志的时间 ( long 型 ) 。

image/fsimage 是一个保护文件，防止 0.13 以前的版本启动 ( 0.13 以前版本将 fsimage 存放在 name/image 目录下，如果用 0.13 版本启动，显然在读 fsimage 会出错 ) 。

我们可以开始讨论 FSImage 了，类 FSImage 如下图：



分析 FSImage，不免要跟 DataStorage 做比较（上图也保留了类 DataStorage）。前面我们已经分析过 DataStorage 的状态变化，包括升级/回滚/提交，FSImage 也有类似的升级/回滚/提交动作，而且这部分的行为和 DataStorage 是比较一致，如下状态转移图。图中 update 方法和 DataStorage 的差别比较大，是因为处理数据库和处理文件系统名字空间不一样，其他的地方都比较一致。FSImage 也能够管理多个 Storage，而且还能区分 Storage 为 IMAGE(目录结构)/EDITS ( 日志 ) /IMAGE\_AND\_EDITS ( 前面两种的组合 ) 。



我们可以看到，FSImage 和 DataStorage 都有 recoverTransitionRead 方法。FSImage 的 recoverTransitionRead 方法主要步骤是检查系统一致性 ( analyzeStorage ) 并尝试恢复，初始化新的 storage ，然后根据启动 NameNode 的参数，做升级 / 回滚等操作。

FSImage 需要支持参数 -importCheckpoint , 该参数用于在某一个 checkpoint 目录里加载 HDFS 的目录信息，并更新到当前系统，该参数的主要功能在方法 doImportCheckpoint 中。该方法很简单，通过读取配置的 checkpoint 目录来加载 fsimage 文件和日志文件，然后利用 saveFSImage ( 下面讨论 ) 保存到当前的工作目录，完成导入。

loadFSImage(File curFile) 用于在 fsimage 中读入 NameNode 持久化的信息，是 FSImage 中最重要的方法之一，该文件的结构如下：

imgVersion(int)	namespaceID(int)	
numFiles(long)		
genstamp(long)		
path(String) ... ...		
replication	modificationTime(long)	
(long)	atime(long)	
(long)	blockSize(long)	
(long)	numBlocks(int)	Block
Block(Class)		
...		
nsQuota(long)		
dsQuota(long)		
permissions(Class) ... ...		
...		
INodeFileUnderConstruction(Class)		
...		

最开始是版本号（注意，各版本文件布局不一样，文中分析的样本是 0.17 的），然后是命名空间的 ID 号，文件个数和最高文件版本号（就是说，下一次产生文件版本号的初始值）。接下来就是文件的信息啦，首先是文件名，然后是该文件的副本数，接下来是修改时间/访问时间，数据块大小，数据块数目。数据块数目如果大于 0，表明这是个文件，那么接下来就是 numBlocks 个数据块（浅蓝），如果数据块数目等于 0，那该条目是目录，接下来是应用于该目录的 quota。最后是访问控制的一些信息。文件信息一共有 numFiles 个，接下来是处于构造状态的文件的信息。（有些版本可能还会保留 DataNode 的信息，但 0.17 已经不保存这样的信息啦）。loadFSImage(File curFile)的对应方法是 saveFSImage(File newFile)，FSImage 中还有一系列的方法（大概 7，8 个）用于配合这两个方法工作，我们就不再深入讨论了。

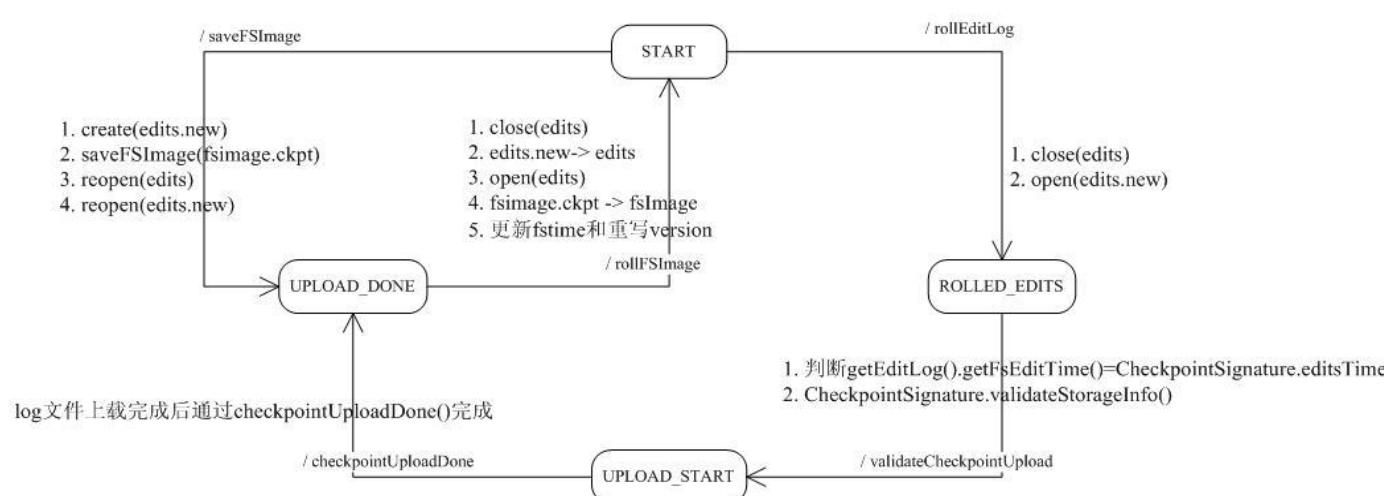
loadFSEdits(StorageDirectory sd)用于加载日志文件，并把日志文件记录的内容应用到 NameNode，loadFSEdits 只是简单地调用 FSEditLog 中对应的方法。

loadFSImage()和 saveFSImage()是另外一对重要的方法。

loadFSImage()会在所有的 Storage 中，读取最新的 NameNode 持久化信息，并应用相应的日志，当 loadFSImage()调用返回以后，内存中的目录树就是最新的。loadFSImage()会返回一个标记，如果 Storage 中有任何和内存中最终目录树中不一致的 Image（最常见的情况是日志文件不为空，那么，内存中的 Image 应该是 Storage 的 Image 加上日志，当然还有其它情况），那么，该标记为 true。

saveFSImage()的功能正好相反，它将内存中的目录树持久化，很自然，目录树持久化后就可以把日志清空。saveFSImage()会创建 edits.new，并把当前内存中的目录树持久化到 fsimage.ckpt（fsimage 现在还存在），然后重新打开日志文件 edits 和 edits.new，这会导致日志文件 edits 和 edits.new 被清空。最后，saveFSImage()调用 rollFSImage()方法。

rollFSImage()上来就把所有的 edits.new 都改为 edits（经过了方法 saveFSImage，它们都已经为空），然后再把 fsimage.ckpt 改为 fsimage。如下图：



为了防止误调用 rollFSImage()，系统引入了状态 CheckpointStates.UPLOAD\_DONE。

有了上面的状态转移图，我们就很好理解方法 recoverInterruptedCheckpoint 了。

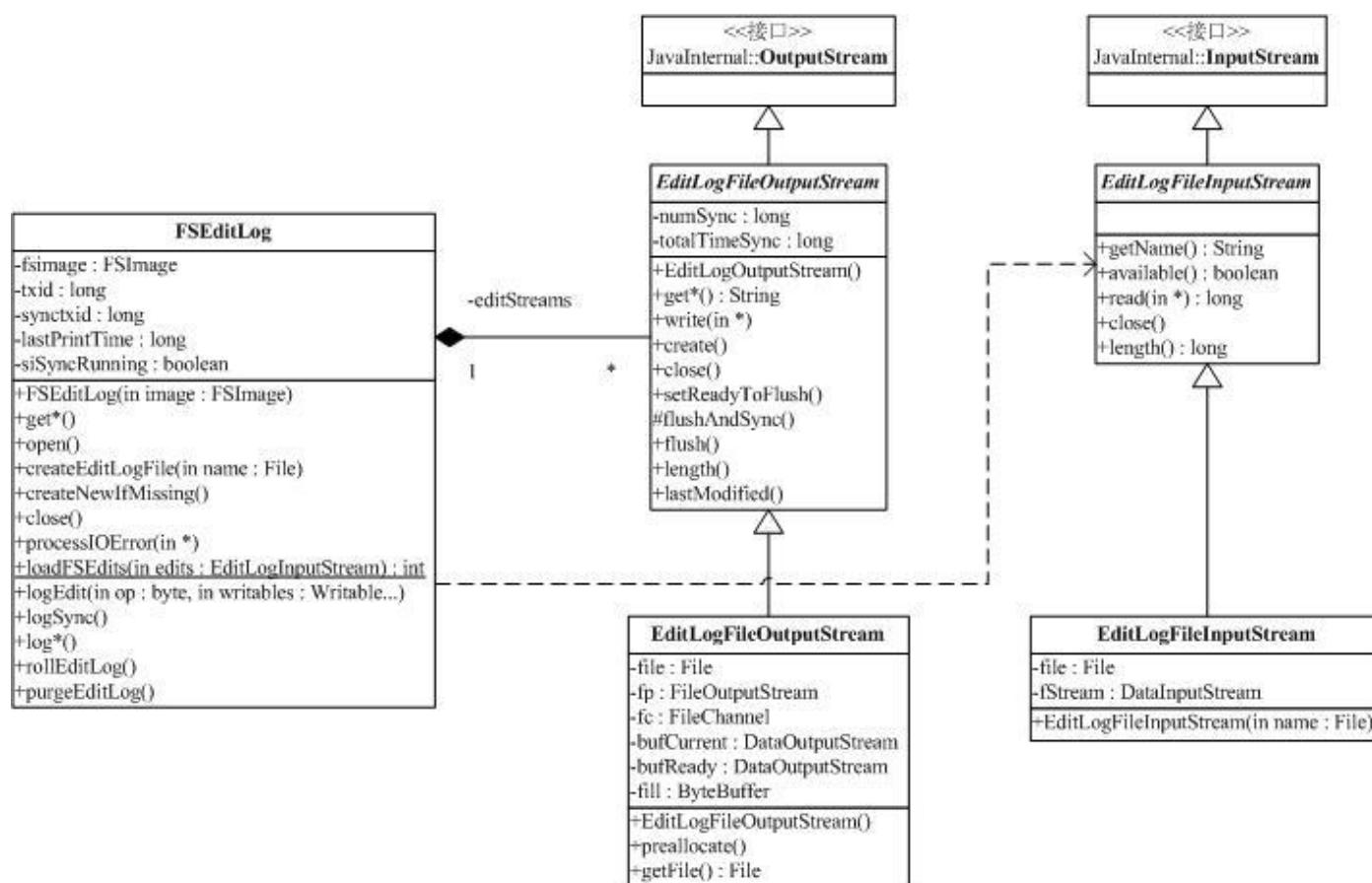
图中存在另一条路径，应用于 GetImageServlet 中。GetImageServlet 是和从 NameNode 进行文件通信的接口，这个场景留到我们分析从 NameNode 时再进行分析。

最后我们分析一下和检查点相关的一个类，rollFSImage()会返回这个类的一个实例。CheckpointSignature 用于标识一个日志的检查点，它是 StorageInfo 的子类，同时实现了 WritableComparable 接口，除了 StorageInfo 的信息，它还包括了两个属性：editsTime 和 checkpointTime。editsTime 是日志的最后修改时间，checkpointTime 是日志建立时间。在和从 NameNode 节点的通信中，需要用 CheckpointSignature，来保证从 NameNode 获得的日志是最新的。

## [Hadoop 源代码分析（二一）](#)

不好意思，突然间需要忙项目的其他事情了，更新有点慢下来，争取月底搞定 HDFS 吧。

我们来分析 FSEditLog.java，该类提供了 NameNode 操作日志和日志文件的相关方法，相关类图如下：



首先是 FSEditLog 依赖的输入/输出流。输入流基本上没有新添加功能；输出流在打开的时候，会写入日志的版本号（最前面的 4 字节），同时，每次将内存刷到硬盘时，会为日志尾部写入一个特殊的标识（OP\_INVALID）。

FSEditLog 有打开/关闭的方法，它们都是很简单的方法，就是关闭的时候，要等待所有正在写日志的操作都完成写以后，才能关闭。processIOError 用于处理 IO 出错，一般这会导致对于的 Storage 的日志文件被关闭（还记得 loadFSImage 要找出最后写的日志文件吧，这也是提高系统可靠性的一个方法），如果系统再也找不到可用的日志文件，NameNode 将会退出。

loadFSEdits 是个大家伙，它读取日志文件，并把日志应用到内存中的目录结构中。这家伙大是因为它需要处理所有类型的日志记录，其实就一大 case 语句。logEdit 的作用和 loadFSEdits 相反，它向日志文件中写入日志记录。我们来分析一下什么操作需要写 log，还有就是需要 log 那些参数：

**logOpenFile ( OP\_ADD ) : 申请 lease**

path(路径)/replication ( 副本数 , 文本形式 ) /modificationTime ( 修改时间 , 文本形式 ) /accessTime ( 访问时间 , 文本形式 ) /preferredBlockSize ( 块大小 , 文本形式 ) /BlockInfo[] ( 增强的数据块信息 , 数组 ) /permissionStatus ( 访问控制信息 ) /clientName ( 客户名 ) /clientMachine ( 客户机器名 )

#### **logCloseFile ( OP\_CLOSE ) : 归还 lease**

path/replication/modificationTime/accessTime/preferredBlockSize/BlockInfo[]/permissionStatus

#### **logMkDir ( OP\_MKDIR ) : 创建目录**

path/modificationTime/accessTime/permissionStatus

#### **logRename ( OP\_RENAME ) : 改文件名**

src ( 原文件名 ) /dst ( 新文件名 ) /timestamp ( 时间戳 )

#### **logSetReplication ( OP\_SET\_REPLICATION ) : 更改副本数**

src/replication

#### **logSetQuota ( OP\_SET\_QUOTA ) : 设置空间额度**

path/nsQuota ( 文件空间额度 ) /dsQuota ( 磁盘空间额度 )

#### **logSetPermissions ( OP\_SET\_PERMISSIONS ) : 设置文件权限位**

src/permissionStatus

#### **logSetOwner ( OP\_SET\_OWNER ) : 设置文件组和主**

src/username ( 所有者 ) /groupname ( 所在组 )

#### **logDelete ( OP\_DELETE ) : 删除文件**

src/timestamp

#### **logGenerationStamp ( OP\_SET\_GENSTAMP ) : 文件版本序列号**

genstamp ( 序列号 )

#### **logTimes ( OP\_TIMES ) : 更改文件更新/访问时间**

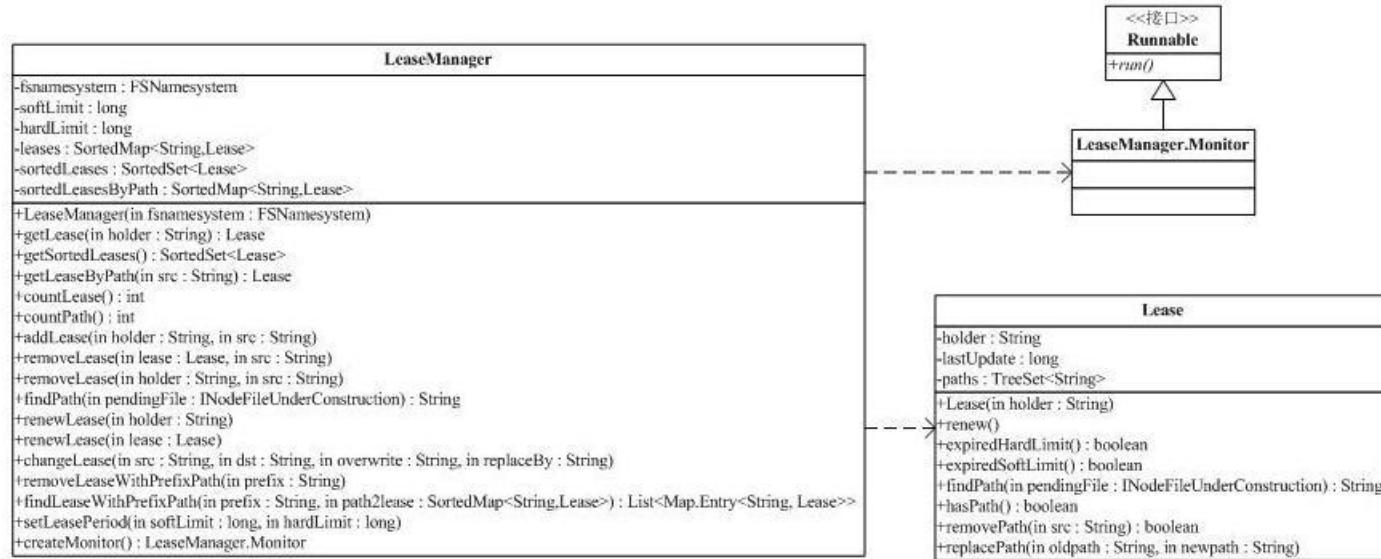
src/modificationTime/accessTime

通过上面的分析，我们应该清楚日志文件里记录了那些信息。

rollEditLog()我们在前面已经提到过（配合 saveFSImage 和 rollFSImage），它用于关闭 edits，打开日志到 edits.new。purgeEditLog()的作用正好相反，它删除老的 edits 文件，然后把 edits.new 改名为 edits。这也是 Hadoop 在做更新修改时经常采用的策略。

## Hadoop 源代码分析 (二二)

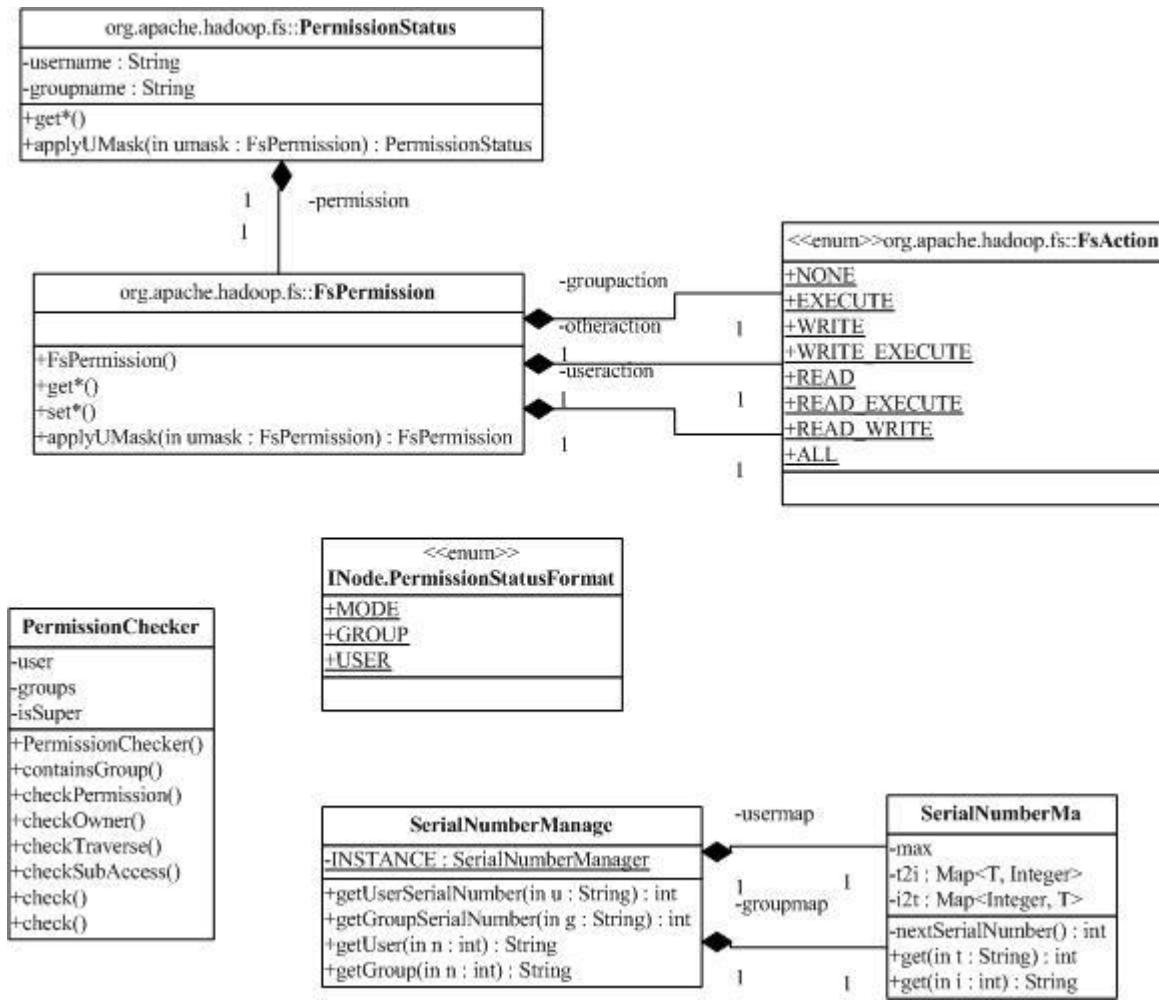
我们开始对租约 Lease 进行分析，下面是类图。Lease 可以认为是一个文件写锁，当客户端需要写文件的时候，它需要申请一个 Lease，NameNode 负责记录那个文件上有 Lease，Lease 的客户是谁，超时时间（分布式处理的一种常用技术）等，所有这些工作由下面 3 个类完成。至于租约过期 NameNode 需要采取什么动作，并不是这部分 code 要完成的功能。



LeaseManager (左) 管理着系统中的所有 Lease (右)，同时，LeaseManager 有一个线程 Monitor，用于检查是否有 Lease 到期。

一个租约由一个 holder ( 客户端名 ) , lastUpdate ( 上次更新时间 ) 和 paths ( 该客户端操作的文件集合 ) 构成。了解了这些属性，相关的方法就很好理解了。LeaseManager 的方法也就很好理解，就是对 Lease 进行操作。注意，LeaseManager 的 addLease 并没有检查文件上是否已经有 Lease，这个是由 LeaseManager 的**调用者来保证的**，这使 LeaseManager 跟简单。内部类 Monitor 通过对 Lease 的最后跟新时间来检测 Lease 是否过期，如果过期，简单调用 FSNamesystem 的 internalReleaseLease 方法。

这部分的代码比我想象的简单，主要是大部分的一致性逻辑都存在于 LeaseManager 的使用者。在开始分析 FSNamesystem.java 这个 4.5k 多行的庞然大物之前，我们继续来扫除外围的障碍。下面是关于访问控制的一些类：



Hadoop 文件保护采用的 UNIX 的机制，文件用户分文件属主、文件组和其他用户，权限读，写和执行（ FsAction 中抽象了所有组合）。

我们先分析包 org.apache.hadoop.fs.permission 的几个类吧。 FsAction 抽象了操作权限， FsPermission 记录了某文件/路径的允许情况，分文件属主、文件组和其他用户，同时提供了一系列的转换方法， applyUMask 用于去掉某些权限，如某些操作需要去掉文件的写权限，那么可以通过该方法，生成对应的去掉写权限的 FsPermission 对象。 PermissionStatus 用于描述一个文件的文件属主、文件组和它的 FsPermission 。

INode 在保存 PermissionStatus 时，用了不同的方法，它用一个 long 变量，和 SerialNumberManager 配合，保存了 PermissionStatus 的所有信息。

SerialNumberManager 保存了文件主和文件主号，用户组和用户组号的对应关系。注意，在持久化信息 FSImage 中，不保存文件主号和用户组号，它们只是 SerialNumberManager 分配的，只保存在内存的信息。通过 SerialNumberManager 得到某文件主的文件主号时，如果找不到文件主号，会往对应关系中添加一条记录。

INode 的 long 变量作为一个位串，分组保存了 FsPermission ( MODE ) ，文件主号 ( USER ) 和用户组号 ( GROUP ) 。

PermissionChecker 用于权限检查。

### [Hadoop 源代码分析 \( 二三 \)](#)

下面我们来分析 FSDirectory 。其实分析 FSDirectory 最好的地方，应该是介绍完 INode\* 以后， FSDirectory 在 INode\* 的基础上，保存了 HDFS 的文件目录状态。系统加载 FSImage 时， FSImage 会在 FSDirectory 对象上重建文件目录状态， HDFS 文件目录状态的变化，也由 FSDirectory 写日志，同时，它保存了文件名à数据块的映射关系。

FSDirectory 只有很少的成员变量，如下：

```

final FSNamesystem namesystem;
final INodeDirectoryWithQuota rootDir;
FSImage fsImage;
boolean ready = false;

```

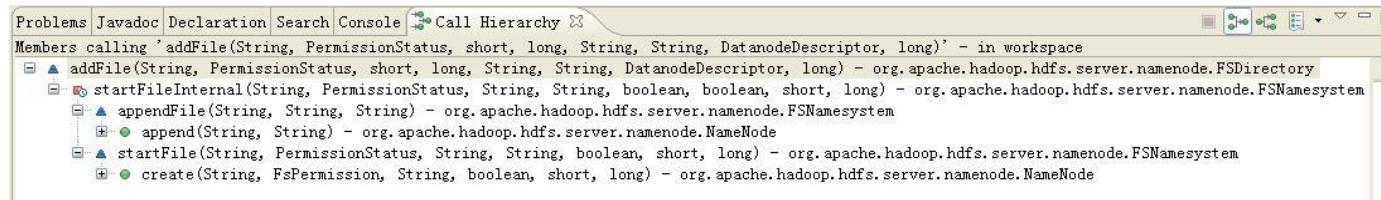
其中，namesystem，fsImage 是指向 FSNamesystem 对象和 FSImage 对象的引用，rootDir 是文件系统的根，ready 初值为 false，当系统成功加载 FSImage 以后，ready 会变成 true，FSDirectory 的使用者就可以调用其它 FSDirectory 功能了。

FSDirectory 中剩下的，就是一堆的方法（我们不讨论和 MBean 相关的类，方法和过程）。

FSDirectory
-namespace -rootDir -fsImage -ready  +FSDirectory(in fsImage : FSImage, in ns : FSNamesystem, in conf : Configuration) +loadFSImage(in dataDirs : Collection<File>, in editsDirs : Collection<File>, in startOpt : enumeration) +close() +waitForReady() +addFile(in path, in permissions, in replication, in preferredBlockSize, in clientName : String, in clientMachine : String, in clientNode : DatanodeDescriptor, in generationStamp) : INodeFileUnderConstruction +unprotectedAddFile(in path, in permissions, in replication, in modificationTime, in atime, in preferredBlockSize) : INode +addToParent(in src, in parentINode : INodeDirectory, in permissions, in blocks, in replication, in modificationTime, in atime, in nsQuota, in dsQuota, in preferredBlockSize) : INodeDirectory +addBlock(in path, in inodes : INode[], in block : Block) : Block +persistBlocks(in path, in file : INodeFileUnderConstruction) +closeFile(in path, in file : INodeFile) +removeBlock(in path, in fileNode : INodeFileUnderConstruction, in block) : boolean +renameTo(in src : String, in dst : String) : boolean +unprotectedRenameTo(in src, in dst, in timestamp) : boolean +setReplication(in src, in replication, in oldReplication : int[]) : Block[] +unprotectedSetReplication(in src, in replication, in oldReplication) : Block[] +getPreferredBlockSize(in filename : String) : long +exists() : boolean +setPermission(in src, in permission) +unprotectedSetPermission(in src, in permission) +setOwner(in src : String, in username : String, in groupname : String) +unprotectedSetOwner(in src, in username, in groupname) +delete(in src) : INode +isDirEmpty(in src : String) : boolean +unprotectedDelete(in src, in modificationTime : long) : INode +replaceNode(in path, in oldnode : INodeFile, in newnode : INodeFile) +replaceNode(in path, in oldnode, in newnode, in updateDiskSpace : boolean) +getListing(in src) : FileStatus[] +getFileInfo(in src) : FileStatus +getFileBlocks(in src) : Block[] +getNode(in src) : INodeFile +getExistingPathINodes(in path) : INode[] +isValidToCreate(in src) : boolean +isDir(in src) : boolean +updateSpaceConsumed(in path, in nsDelta : long, in dsDelta : long) +updateCount(in inodes : INode[], in numOfNodes : int, in nsDelta, in dsDelta) +getFullPathName(in inodes : INode[], in pos : int) : String +mkdirs(in src, in permissions, in inheritPermission : boolean, in now : long) : boolean +unprotectedMkdir(in src, in permissions, in timestamp : long) : INode +unprotectedMkdir(in inodes : INode[], in pos : int, in name : byte[], in permission, in inheritPermission, in timestamp) +addNode(in src, in child : T, in childDiskSpace : long, in inheritPermission) : <T extends INode> T +addChild(in pathComponents : INode[], in pos : int, in child : T, in inheritPermission) : <T extends INode> T +addChild(in pathComponents, in pos, in child, in childDiskSpace : long, in inheritPermission) : <T extends INode> T +removeChild(in pathComponents, in pos) : INode +normalizePath(in src) : String +getContentSummary(in src) : ContentSummary +updateCountForINodeWithQuota() +updateCountForINodeWithQuota(in dir : INodeDirectory, in counts : INode.DirCounts, in nodesInPath : ArrayList<INode>) +unprotectedSetQuota(in src, in nsQuota, in dsQuota) : INodeDirectory +setQuota(in src, in nsQuota, in dsQuota) +totalINodes() : long +setTimes(in src : String, in node : INodeFile, in mtime : long, in atime : long, in force : boolean) +unprotectedSetTimes(in src, in mtime, in atime, in force : boolean) : boolean +unprotectedSetTimes(in src, in node : INodeFile, in mtime, in atime, in force) : boolean +createFileStatus(in path, in node : INode) : FileStatus

loadFSImage 用于加载目录树结构，它会去调用 FSImage 的方法，完成持久化信息的导入以后，它会把成员变量 ready 置为 true。系统调用 loadFSImage 是在 FSNamesystem.java 的 initialize 方法，那是系统初始化重要的一步。

addFile 用于创建文件或追加数据时创建 INodeFileUnderConstruction，下图是它的 Call Hierarchy 图：



addFile 首先会试图在系统中创建到文件的路径，如果文件为 /home/hadoop/Hadoop.tar，addFile 会调用 mkdirs（创建路径为 /home/hadoop，这也会涉及到一系列方法），保证文件路径存在，然后创建 INodeFileUnderConstruction 节点，并把该节点加到目录树中（通过 addNode，也是需要调用一系列方法），如果成功，就写操作日志（logOpenFile）。

unprotectedAddFile 也用于在系统中创建一个目录或文件（非 UnderConstruction），如果是文件，还会建立对应的 block。FSDirectory 中还有好几个 unprotected\* 方法，它们不检查成员变量 ready，不写日志，它们大量用于 loadFSEdits 中（这个时候 ready 当然是 false，而且因为正在恢复日志，也不需要写日志）。

addToParent 添加一个 INode 到目录树中，并返回它的上一级目录，它的实现和 unprotectedAddFile 是类似的。

persistBlocks 比较有意思，用于往日志里记录某 inode 的 block 信息，其实并没有一个对应于 persistBlocks 的写日志方法，它用的是 logOpenFile。这个大家可以去检查一下 logOpenFile 记录的信息。closeFile 对应了 logCloseFile。

addBlock 和 removeBlock 对应，用于添加/删除数据块信息，同时它们还需要更新 FSNamesystem.java 中对应的信息。

unprotectedRenameTo 和 renameTo 实现了 UNIX 的 mv 命令，主要的功能都在 unprotectedRenameTo 中完成，复杂的地方在于对各种各样情况的讨论。

setReplication 和 unprotectedSetReplication 用于更新数据块的副本数，很简单的方法，注意，改变产生的对数据块的删除/复制是在 FSNamesystem.java 中实现。

setPermission，unprotectedSetPermission，setOwner 和 unprotectedSetOwner 都是简单的方法。

Delete 和 unprotectedDelete 又是一对方法，删除如果需要删除数据块，将通过 FSNamesystem 的 removePathAndBlocks 进行。

.....(后续的方法和前面介绍的，都比较类似，都是一些过程性的东西，就不再讨论了)

## Hadoop 源代码分析（二四）

下面轮到 FSNamesystem 出场了。FSNamesystem.java 一共有 4573 行，而整个 namenode 目录下所有的 Java 程序总共也只有 16876 行，把 FSNamesystem 搞定了，NameNode 也就基本搞定。

FSNamesystem 是 NameNode 实际记录信息的地方，保存在 FSNamesystem 中的数据有：

文件名à数据块列表（存放在 FSImage 和日志中）

合法的数据块列表（上面关系的逆关系）

数据块àDataNode（只保存在内存中，根据 DataNode 发过来的信息动态建立）

DataNode 上保存的数据块（上面关系的逆关系）

最近发送过心跳信息的 DataNode（LRU）

我们先来分析 FSNamesystem 的成员变量。

```
private boolean isPermissionEnabled;
```

是否打开权限检查，可以通过配置项 dfs.permissions 来设置。

```
private UserGroupInformation fsOwner;
```

本地文件的用户文件属主和文件组，可以通过 hadoop.job.ugi 设置，如果没有设置，那么将使用启动 HDFS 的用户（通过 whoami 获得）和该用户所在的组（通过 groups 获得）作为值。

```
private String supergroup;
```

对应配置项 dfs.permissions.supergroup，应用在 defaultPermission 中，是系统的超级组。

```
private PermissionStatus defaultPermission;
```

缺省权限，缺省用户为 fsOwner，缺省用户组为 supergroup，缺省权限为 0777，可以通过 dfs.upgrade.permission 修改。

```
private long capacityTotal, capacityUsed, capacityRemaining;
```

系统总容量/已使用容量/剩余容量

```
private int totalLoad = 0;
```

系统总连接数，根据 DataNode 心跳信息跟新。

```
private long pendingReplicationBlocksCount, underReplicatedBlocksCount, scheduledReplicationBlocksCount;
```

分别是成员变量 pendingReplications（正在复制的数据块），neededReplications（需要复制的数据块）的大小，  
scheduledReplicationBlocksCount 是当前正在处理的复制工作数目。

```
public FSDirectory dir;
```

指向系统使用的 FSDirectory 对象。

```
BlocksMap blocksMap = new BlocksMap();
```

保存数据块到 INode 和 DataNode 的映射关系

```
public CorruptReplicasMap corruptReplicas = new CorruptReplicasMap();
```

保存损坏（如：校验没通过）的数据块到对应 DataNode 的关系，CorruptReplicasMap 类图如下，类只有一个成员变量，保存 Block 到一个 DatanodeDescriptor 的集合的映射和这个映射上的一系列操作：

CorruptReplicasMap
-corruptReplicasMap : <Block, Collection<DatanodeDescriptor>>
+addToCorruptReplicasMap()
+removeFromCorruptReplicasMap(in blk)
+removeFromCorruptReplicasMap(in blk, in datanode)
+getNodes()
+isReplicaCorrupt()
+numCorruptReplicas()

Map<String, DatanodeDescriptor> datanodeMap = new TreeMap<String, DatanodeDescriptor>();

保存了 StorageID DatanodeDescriptor 的映射，用于保证 DataNode 使用的 Storage 的一致性。

**private Map<String, Collection<Block>> recentInvalidateSets**

保存了每个 DataNode 上无效但还存在的数据块 ( StorageID ArrayList<Block> ) 。

Map<String, Collection<Block>> recentInvalidateSets

保存了每个 DataNode 上有效 ,但需要删除的数据块( StorageID TreeSet<Block> ) ,这种情况可能发生在 DataNode 故障后恢复后 ,上面的数据块在系统中副本数太多 ,需要删除一些数据块。

HttpServer infoServer;

int infoPort;

Date startTime;

用于内部信息传输的 HTTP 请求服务器( Servlet 的容器 )。现在有 /fsck ,/getImage ,/listPaths/\* ,/data/\* 和 /fileChecksum/\* ,我们后面还会继续讨论。

ArrayList<DatanodeDescriptor> heartbeats;

所有目前活着的 DataNode ,线程 HeartbeatMonitor 会定期检查。

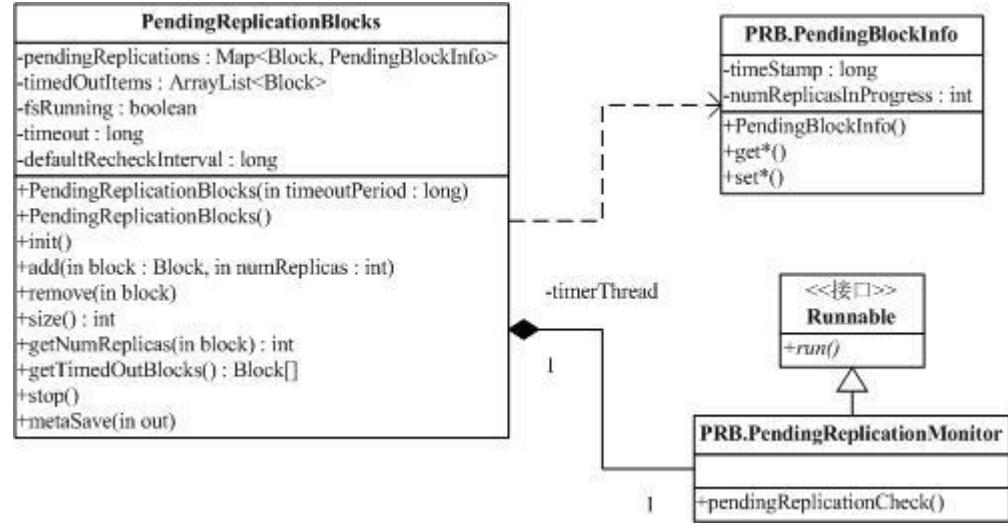
**private UnderReplicatedBlocks neededReplications**

需要进行复制的数据块。 UnderReplicatedBlocks 的类图如下 ,它其实是一个数组 ,数组的下标是优先级 ( 0 的优先级最高 ,如果数据块只有一个副本 ,它的优先级是 0 ) ,数组的内容是一个 Block 集合。 UnderReplicatedBlocks 提供一些方法 ,对 Block 进行增加 ,修改 ,查找和删除。

UnderReplicatedBlocks
-priorityQueues : List<TreeSet<Block>>
+UnderReplicatedBlocks()
+clear()
+size() : int
+contains(in block : Block) : boolean
+getPriority(in block : Block, in curReplicas : int, in decommissionedReplicas : int, in expectedReplicas : int) : int
+add(in block, in curReplicas, in decommissionedReplicas, in expectedReplicas) : boolean
+remove(in block, in oldReplicas, in decommissionedReplicas, in oldExpectedReplicas) : boolean
+remove(in block, in priLevel : int) : boolean
+update(in block, in curReplicas, in decommissionedReplicas, in curExpectedReplicas : int, in curReplicasDelta : int, in expectedReplicasDelta : int)

**private PendingReplicationBlocks pendingReplications;**

保存正在复制的数据块的相关信息。 PendingReplicationBlocks 的类图如下 :



其中，pendingReplications 保存了所有正在进行复制的数据块，使用 Map 是需要一些附加的信息 PendingBlockInfo。这些信息包括时间戳，用于检测是否已经超时，和现在进行复制的数目 numReplicasInProgress。timedOutItems 是超时的复制项，超时的复制项在 FSNamesystem 的 processPendingReplications 方法中被删除，并从新复制。timerThread 是用于检测复制超时的线程的句柄，对应的线程是 PendingReplicationMonitor 的一个实例，它的 run 方法每隔一段会检查是否有超时的复制项，如果有，将该数据块加到 timedOutItems 中。Timeout 是 run 方法的检查间隔，defaultRecheckInterval 是缺省值。PendingReplicationBlocks 和 PendingBlockInfo 的方法都很简单。

```
public LeaseManager leaseManager = new LeaseManager(this);
```

租约管理器。

## [Hadoop 源代码分析 \(二五\)](#)

继续对 FSNamesystem 进行分析。

```
Daemon hbthread = null; // HeartbeatMonitor thread
public Daemon lmthread = null; // LeaseMonitor thread
Daemon smmthread = null; // SafeModeMonitor thread
public Daemon replthread = null; // Replication thread
```

NameNode 上的线程，分别对应 DataNode 心跳检查，租约检查，安全模式检查和数据块复制，我们会在后面介绍这些线程对应的功能。

```
volatile boolean fsRunning = true;
```

```
long systemStart = 0;
```

系统运行标志和系统启动时间。

接下来是一堆系统的参数，比方说系统每个 DataNode 节点允许的最大数据块数，心跳检查间隔时间等... ...

```
// The maximum number of replicates we should allow for a single block  
private int maxReplication;  
// How many outgoing replication streams a given node should have at one time  
private int maxReplicationStreams;  
// MIN_REPLICATION is how many copies we need in place or else we disallow the write  
private int minReplication;  
// Default replication  
private int defaultReplication;  
// heartbeatRecheckInterval is how often namenode checks for expired datanodes  
private long heartbeatRecheckInterval;  
// heartbeatExpireInterval is how long namenode waits for datanode to report  
// heartbeat  
private long heartbeatExpireInterval;  
//replicationRecheckInterval is how often namenode checks for new replication work  
private long replicationRecheckInterval;  
//decommissionRecheckInterval is how often namenode checks if a node has finished decommission  
private long decommissionRecheckInterval;  
// default block size of a file  
private long defaultBlockSize = 0;  
  
private int replIndex = 0;
```

和 neededReplications 配合，记录下一个进行复制的数据块位置。

```
public static FSNamesystem fsNamesystemObject,
```

哈哈，不用介绍了，还是 **static** 的。

```
private String localMachine;
```

```
private int port;
```

本机名字和 RPC 端口。

```
private SafeModeInfo safeMode; // safe mode information
```

记录安全模式的相关信息。

安全模式是这样一种状态，系统处于这个状态时，不接受任何对名字空间的修改，同时也不会对数据块进行复制或删除数据块。

NameNode 启动的时候会自动进入安全模式，同时也可手工进入（不会自动离开）。系统启动以后，DataNode 会报告目前它拥有的数据块的信息，当系统接收到的 Block 信息到达一定门槛，同时每个 Block 都有 dfs.replication.min 个副本后，系统等待一段时间后就离开安全模式。这个门槛定义的参数包括：

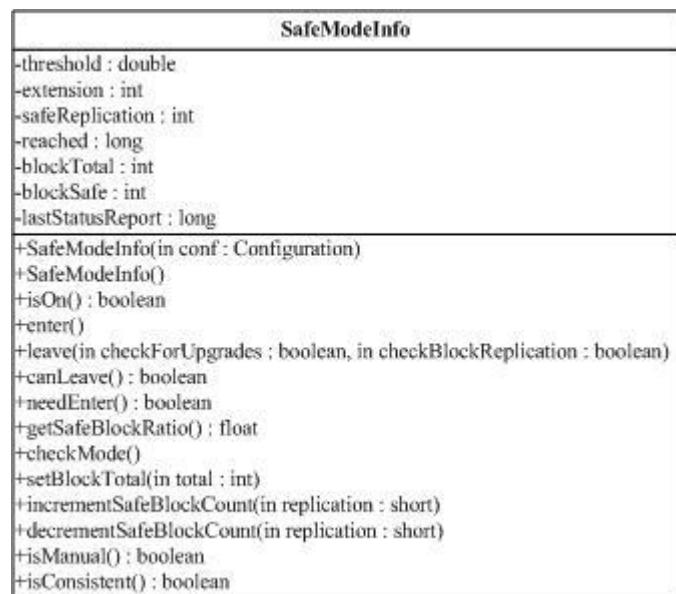
dfs.safemode.threshold.pct : 接受到的 Block 的比例，缺省为 95%，就是说，必须 DataNode

报告的数据块数目占总数的 95%，才到达门槛；

dfs.replication.min：缺省为 1，即每个副本都存在系统中；

dfs.replication.min：等待时间，缺省为 0，单位秒。

SafeModeInfo 的类图如下：



threshold，extension 和 safeReplication 保存的是上面说的 3 个参数。Reached 等于-1 表明安全模式是关闭的，0 表示安全模式打开但是系统还没达到 threshold。blockTotal 是计算 threshold 时的分母，blockSafe 是分子，lastStatusReport 用于控制写日志的间隔。

SafeModeInfo(Configuration conf) 使用配置文件的参数，是 NameNode 正常启动时使用的构造函数，SafeModeInfo() 中，this.threshold = 1.5f 使得系统用于处于安全模式。

enter() 使系统进入安全模式，leave() 会使系统离开安全模式，canLeave() 用于检查是否能离开安全模式而 needEnter()，则判断是否应该进入安全模式。checkMode() 检查系统状态，如果必要，则进入安全模式。其他的方法都比较简单，大多为对成员变量的访问。

讨论完类 SafeModeInfo，我们来分析一下 SafeModeMonitor，它用于定期检查系统是否能够离开安全模式（smmthread 就是它的一个实例）。系统离开安全模式后，smmthread 会被重新赋值为 null。

## [Hadoop 源代码分析（二六）](#)

(没想到需要分页啦)

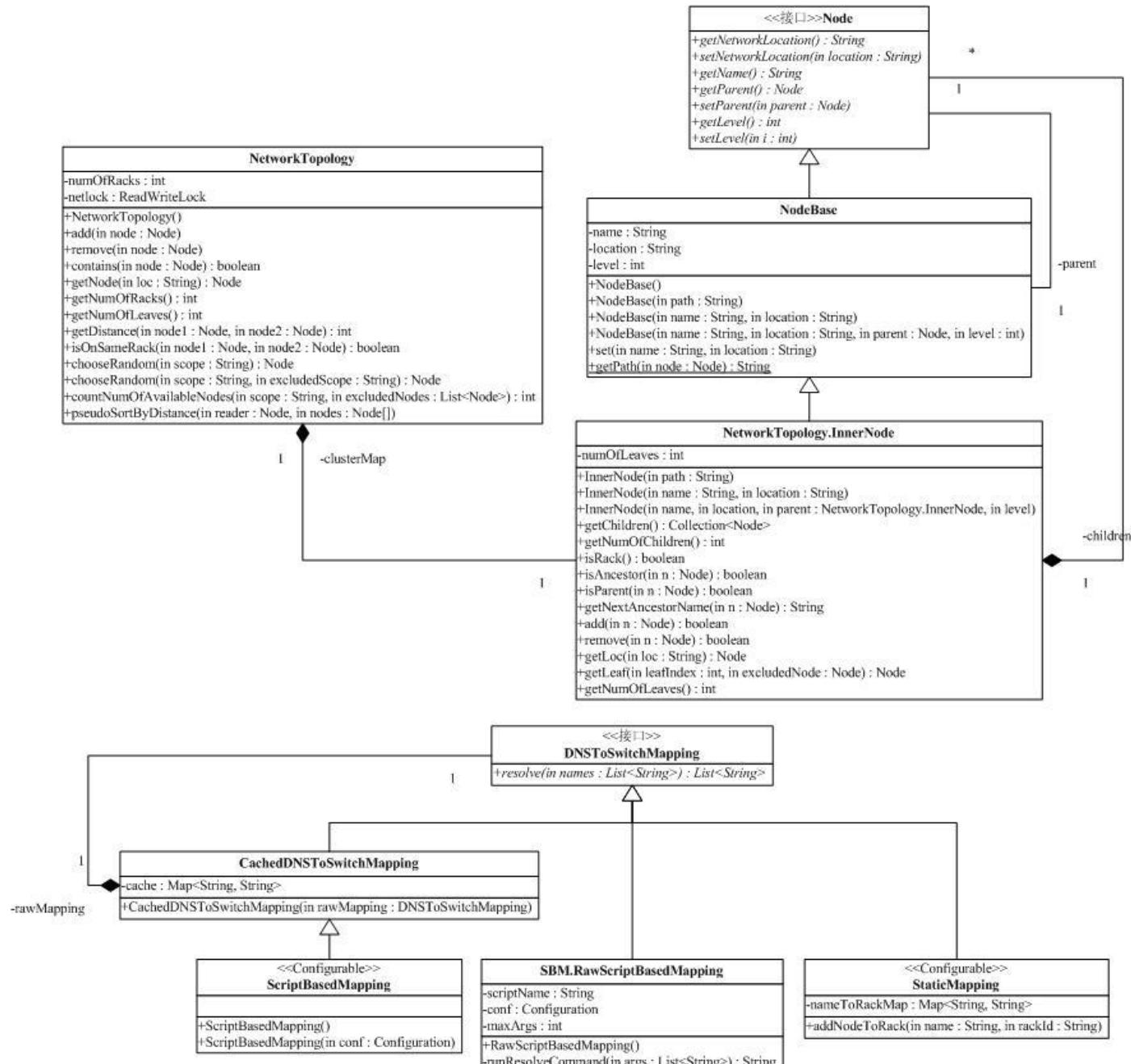
**private Host2NodesMap host2DataNodeMap = new Host2NodesMap();**

保存了主机名（String）到 DatanodeDescriptor 数组的映射（Host2NodesMap 唯一的成员变量为 HashMap<String,DatanodeDescriptor[]> map，它的方法都是对这个 map 进行操作）。

**NetworkTopology clusterMap = new NetworkTopology();**

**private DNSToSwitchMapping dnsToSwitchMapping;**

定义了 HDFS 的网络拓扑，网络拓扑对应选择数据块副本的位置很重要。如在一个层次型的网络中，接到同一个交换机的两个节点间的网络速度，会比跨越多个交换机的两个节点间的速度快，但是，如果某交换机故障，那么它对接到它上面的两个节点会同时有影响，但跨越多个交换机的两个节点，这种影响会小得多。下面是 NetworkTopology 相关的类图：



Hadoop 实现了一个树状的拓扑结构抽象，其中，Node 接口，定义了网络节点的一些方法，NodeBase 是 Node 的一个实现，提供了叶子节点的一些方法（明显它没有子节点），而 InnerNode 则实现了树的内部节点，如果我们考虑一个网络部署的话，那么叶子节点是服务器，而 InnerNode 则是服务器所在的机架或交换机或路由器。Node 提供了对网络位置信息（采用类似文件树的方式），节点名称和 Node 所在的树的深度的方法。NodeBase 提供了一个简单的实现。InnerNode 是 NetworkTopology 的内部类，对比 NodeBase，它的 children 保存了所有的子节点，这样的话，就可以构造一个拓扑树。这棵树的叶子可能是服务器，也可能是机架，内部则是机架或者是路由器等设备，InnerNode 提供了一系列的方法区分处理这些信息。

NetworkTopology 的 add 方法和 remove 用于在拓扑结构中加入节点和删除节点，同时也给出一些 get\* 方法，用于获取一些对象内部的信息，如 getDistance，可以获取两个节点的距离，而 isOnSameRack 可以判断两个节点是否处于同一个机架。chooseRandom 有两个实现，用于在一定范围内（另一个还有一个排除选项）随机选取一个节点。chooseRandom 在选择数据块副本位置的时候调用。

DNSToSwitchMapping 配合上面 NetworkTopology，用于确定某一个节点的网络位置信息，它的唯一方法，可以通过一系列机器的名字找出它们对应的网络位置信息。目前有支持两种方法，一是通过命令行方式，将节点名作为输入，输出为网络位置信息（RawScriptBasedMapping 执行命令 CachedDNSToSwitchMapping 缓存结果），还有一种就是利用配置参数 hadoop.configured.node.mapping 静态配置（StaticMapping）。

```
ReplicationTargetChooser replicator;
```

用于为数据块备份选择目标，例如，用户写文件时，需要选择一些 DataNode，作为数据块的存放位置，这时候就利用它来选择目标地址。chooseTarget 是 ReplicationTargetChooser 中最重要的方法，它通过内部的一个 NetworkTopology 对象，计算出一个 DatanodeDescriptor 数组，该数组就是选定的 DataNode，同时，顺序就是最佳的数据流顺序（还记得我们讨论 DataXceiver 些数据的那个图吗？）。

```
private HostsFileReader hostsReader;
```

保存了系统中允许/不允许连接到 NameNode 的机器列表。

```
private Daemon dnthread = null;
```

线程句柄，该线程用于检测 DataNode 上的 Decommission 进程。例如，某节点被列入到不允许连接到 NameNode 的机器列表中（HostsFileReader），那么，该节点会进入 Decommission 状态，它上面的数据块会被复制到其它节点，复制结束后机器进入 DatanodeInfo.AdminStates.DECOMMISSIONED，这台机器就可以从 HDFS 中撤掉。

```
private long maxFsObjects = 0; // maximum number of fs objects
```

系统能拥有的 INode 最大数（配置项 dfs.max.objects，0 为无限制）。

```
private final GenerationStamp generationStamp = new GenerationStamp();
```

系统的时间戳生产器。

```
private int blockInvalidateLimit = FSConstants.BLOCK_INVALIDATE_CHUNK;
```

发送给 DataNode 删除数据块消息中，能包含的最大数据块数。比方说，如果某 DataNode 上有 250 个 Block 需要被删除，而这个参数是 100，那么一共会有 3 条删除数据块消息消息，前面两条包含了 100 个数据块，最后一条是 50 个。

```
private long accessTimePrecision = 0;
```

用于控制文件的 access 时间的精度，也就是说，小于这个精度的两次对文件访问，后面的那次就不做记录了。

## [Hadoop 源代码分析（二七）](#)

我们接下来分析 NameNode.java 的成员变量，然后两个类综合起来，分析它提供的接口，并配合说明接口上请求对应的处理流程。

前面已经介绍过了，NameNode 实现了接口 ClientProtocol，DatanodeProtocol 和 NamenodeProtocol，分别提供给客户端/DataNode/从 NameNode 访问。由于 NameNode 的大部分功能在类 FSNamesystem 中实现，那么 NameNode.java 的成员变量就很少了。

```
public FSNamesystem namesystem;
```

指向 FSNamesystem 对象。

```
private Server server;
```

NameNode 的 RPC 服务器实例。

```
private Thread emptier;
```

处理回收站的线程句柄。

```
private int handlerCount = 2;
```

还记得我们分析 RPC 的服务器时提到的服务器请求处理线程( Server.Handle )吗 ? 这个参数给出了 server 中服务器请求处理线程的数目 , 对应配置参数为 dfs.namenode.handler.count.

```
private boolean supportAppends = true;
```

是否支持 append 操作 , 对应配置参数为 dfs.support.append.

```
private InetSocketAddress nameNodeAddress = null;
```

NameNode 的地址 , 包括 IP 地址和监听端口。

下面我们来看 NameNode 的启动过程。 main 方法是系统的入口 , 它会调用 createNameNode 创建 NameNode 实例。 createNameNode 分析命令行参数 , 如果是 FORMAT 或 FINALIZE , 调用对应的方法后退出 , 如果是其他的参数 , 将创建 NameNode 对象。 NameNode 的构造函数会调用 initialize , 初始化 NameNode 的成员变量 , 包括创建 RPC 服务器 , 初始化 FSNamesystem , 启动 RPC 服务器和回收站线程。

FSNamesystem 的构造函数会调用 initialize 方法 , 去初始化上面我们分析过的一堆成员变量。几个重要的步骤包括加载 FSImage , 设置系统为安全模式 , 启动各个工作线程和 HTTP 服务器。系统的一些参数是在 setConfigurationParameters 中初始化的 , 其中一些值的计算比较麻烦 , 而且也可能被其它部分的 code 引用的 , 就独立出来了 , 如 getNamespaceDirs 和 getNamespaceEditsDirs。 initialize 对应的是 close 方法 , 很简单 , 主要是停止 initialize 中启动的线程。

对应于 initialize 方法 , NameNode 也提供了对应的 stop 方法 , 用于初始化时出错系统能正确地退出。

NameNode 的 format 和 finalize 操作 , 都是先构造 FSNamesystem , 然后利用 FSNamesystem 的 FSImage 提供的对应方法完成的。我们在分析 FSImage.java 时 , 已经了解了这部分的功能。

## Hadoop 源代码分析 (二八)

万事俱备 , 我们可以来分析 NameNode 上的流程啦。

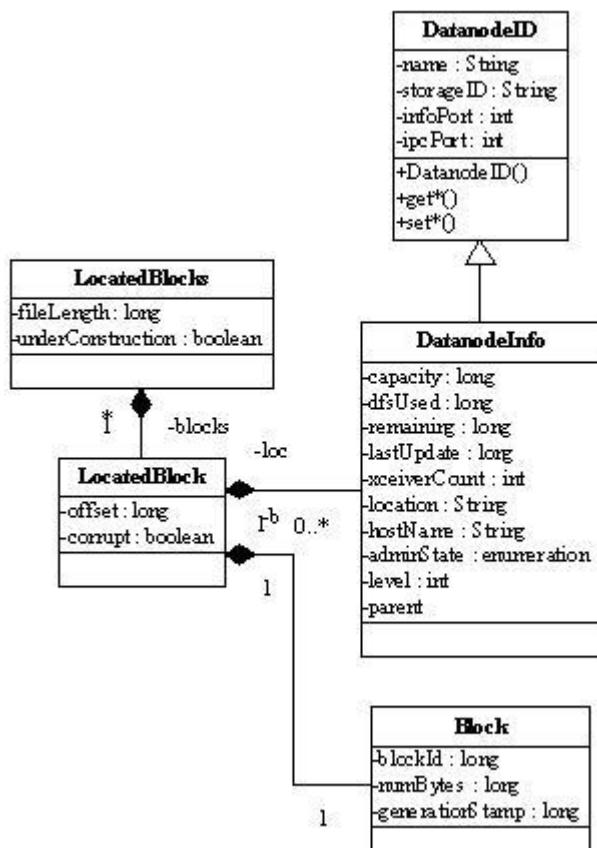
```

<<接口>>
ClientProtocol
+getBlockLocations(in src : String, in offset : long, in length : long) : LocatedBlocks
+create(in src : String, in masked, in clientName : String, in overwrite : boolean, in replication : short, in blockSize : long)
+append(in src : String, in clientName : String) : LocatedBlock
+setReplication(in src : String, in replication : short) : boolean
+setPermission(in src : String, in permission)
+setOwner(in src : String, in username : String, in groupname : String)
+abandonBlock(in b : Block, in src : String, in holder : String)
+addBlock(in src : String, in clientName : String) : LocatedBlock
+complete(in src : String, in clientName : String) : boolean
+reportBadBlocks(in blocks : LocatedBlock[])
+rename(in src : String, in dst : String) : boolean
+delete(in src : String) : boolean
+delete(in src : String, in recursive : boolean) : boolean
+mkdirs(in src : String, in masked) : boolean
+getListing(in src : String) : FileStatus[]
+renewLease(in clientName : String)
+getStats() : long[]
+getDatanodeReport(in type : enumeration) : DatanodeInfo[]
+getPreferredBlockSize(in filename : String) : long
+setSafeMode(in action : enumeration) : boolean
+refreshNodes()
+finalizeUpgrade()
+distributedUpgradeProgress(in action : enumeration) : UpgradeStatusReport
+metaSave(in filename : String)
+getFileInfo(in src : String) : FileStatus
+getContentSummary(in path : String) : ContentSummary
+setQuota(in path : String, in nameSpaceQuota : long, in diskSpaceQuota : long)
+fsync(in src : String, in client : String)
+setTimes(in src : String, in mtime : long, in atime : long)

```

首先我们来看 NameNode 上实现的 ClientProtocol，客户端通过这个接口，可以对目录树进行操作，打开/关闭文件等。

getBlockLocations 用于确定文件内容的位置，它的输入参数为：文件名，偏移量，长度，返回值是一个 LocatedBlocks 对象（如下图），它携带的信息很多，大部分字段我们以前都讨论过。



getBlockLocations 直接调用 NameSystem 的同名方法。NameSystem 中这样的方法首先会检查权限和对参数进行检查（如偏移量和长度要大于 0），然后再调用实际的方法。找 LocatedBlocks 先找 src 对应的 INode，然后通过 INode 的 getBlocks

方法，可以拿到该节点的 Block 列表，如果返回为空，表明该 INode 不是文件，返回 null；如果 Block 列表长度为 0，以空的 Block 数组构造返回的 LocatedBlocks。

如果 Block 数组不为空，则通过请求的偏移量和长度，就可以把这个区间涉及的 Block 找出来，对于每一个 block，执行：

通过 BlocksMap 我们可以找到它存在于几个 DataNode 上（BlocksMap.numNodes 方法）；

计算包含该数据块但数据块是坏的 DataNode 的数目（通过 NameSystem.countNodes 方法，间接访问 CorruptReplicasMap 中的信息）；

计算坏数据块的数目（CorruptReplicasMap.numCorruptReplicas 方法，应该和上面的数相等）；

通过上面的计算，我们得到现在还 OK 的数据块数目；

从 BlocksMap 中找出所有 OK 的数据块对应的 DatanodeDescriptor( DatanodeInfo 的父类)；

创建对应的 LocatedBlock。

收集到每个数据块的 LocatedBlock 信息后，很自然就能构造 LocatedBlocks 对象。getBlockLocations 其实只是一个读的方法，请求到了 NameNode 以后只需要查表就行了。

create 方法，该方法用于在目录树上创建文件（创建目录使用 mkdir），需要的参数比较多，包括文件名，权限，客户端名，是否覆盖已存在文件，副本数和块大小。NameNode 的 create 调用 NameSystem 的 startFile 方法（startFile 需要的参数 clientMachine 从线程局部变量获取）。

startFile 方法先调用 startFileInternal 完成操作，然后调用 logSync，等待日志写完后才返回。

startFileInternal 不但服务于 startFile，也被 appendFile 调用（通过参数 append 区分）。方法的最开始是一堆检查，包括：安全模式，文件名 src 是否正确，权限，租约，replication 参数，overwrite 参数（对 append 操作是判断 src 指向是否存在并且是文件）。租约检查很简单，如果通过 FSDirectory.getFileINode(src) 得到的文件是出于构造状态，表明有客户正在操作该文件，这时会抛出异常 AlreadyBeingCreatedException。

如果对于创建操作，会通过 FSDirectory 的 addFile 往目录树上添加一个文件并在租约管理器里添加一条记录。

对于 append 操作，执行的是构造一个新的 INodeFileUnderConstruction 并替换原有的节点，然后在租约管理器里添加一条记录。

总的来说，最简单的 create 流程就是在目录树上创建一个 INodeFileUnderConstruction 对象并往租约管理器里添加一条记录。

我们顺便分析一下 append 吧 ,它的返回值是 LocatedBlock ,比起 getBlockLocations ,它只需要返回数组的一项。appendFile 是 NameSystem 的实现方法 ,它首先调用上面讨论的 startFileInternal 方法 (已经在租约管理器里添加了一条记录 ) 然后写日志。然后寻找对应文件 INodeFile 中记录的最后一个 block ,并通过 BlocksMap.getStoredBlock()方法得到 BlockInfo ,然后再从 BlocksMap 中获得所有的 DatanodeDescriptor ,就可以构造 LocatedBlock 了。需要注意的 ,如果该 Block 在需要被复制的集合 ( UnderReplicatedBlocks ) 中 ,移除它。

如果文件刚被创建或者是最后一个数据块已经写满 ,那么 append 会返回 null ,这是客户端需要使用 addBlock ,为文件添加数据块。

## [Hadoop 源代码分析 \(二九\)](#)

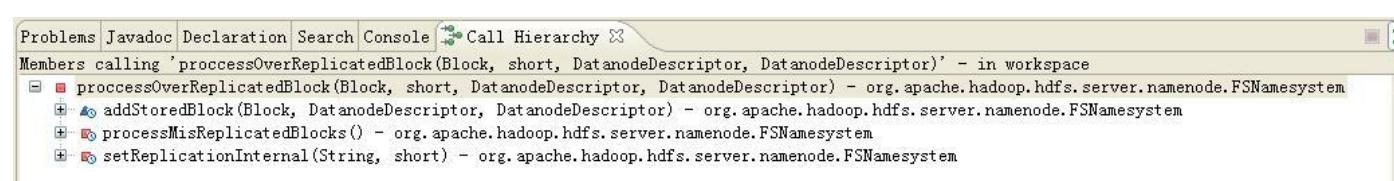
```
public boolean setReplication(String src,  
                               short replication  
) throws IOException;
```

setReplication ,设置文件 src 的副本数为 replication ,返回值为 boolean ,在 FSNameSystem 中 ,调用方法 setReplicationInternal ,然后写日志。

setReplicationInternal 上来自然是检查参数了 ,然后通过 FSDirectory 的 setReplication ,设置新的副本数 ,并获取老的副本数。根据新旧数 ,决定删除/复制数据块。

增加副本数通过调用 updateNeededReplications ,为了获取 UnderReplicatedBlocks. update 需要的参数 ,FSNameSystem 提供了内部方法 countNodes 和 getReplication ,获得对应的数值 (这两个函数都很简单) 。

processOverReplicatedBlock 用于减少副本数 ,它被多个方法调用 :



主要参数有 block ,副本数 ,目标 DataNode ,源 DataNode ( 用于删除 ) 。 processOverReplicatedBlock 首先找出 block 所在的 ,处于非 Decommission 状态的 DataNode 的信息 ,然后调用 chooseExcessReplicates 。 chooseExcessReplicates 执行 :

按机架位置 ,对 DatanodeDescriptor 进行分组 ;

将 DataNode 分为两个集合 ,分别是一个机架包含一个以上的数据块的和剩余的 ;

选择可以删除的数据块 ( 顺序是 : 源 DataNode ,同一个机架上的 ,剩余的 ) ,把它加到 recentInvalidateSets 中。

```
public void setPermission(String src, FsPermission permission  
    ) throws IOException;
```

setPermission，用于设置文件的访问权限。非常简单，首先检查是否有权限，然后调用 FSDirectory.setPermission 修改文件访问权限。

```
public void setOwner(String src, String username, String groupname
```

```
    ) throws IOException;
```

```
public void setTimes(String src, long mtime, long atime) throws IOException;
```

```
public void setQuota(String path, long namespaceQuota, long diskspaceQuota)
```

```
    throws IOException;
```

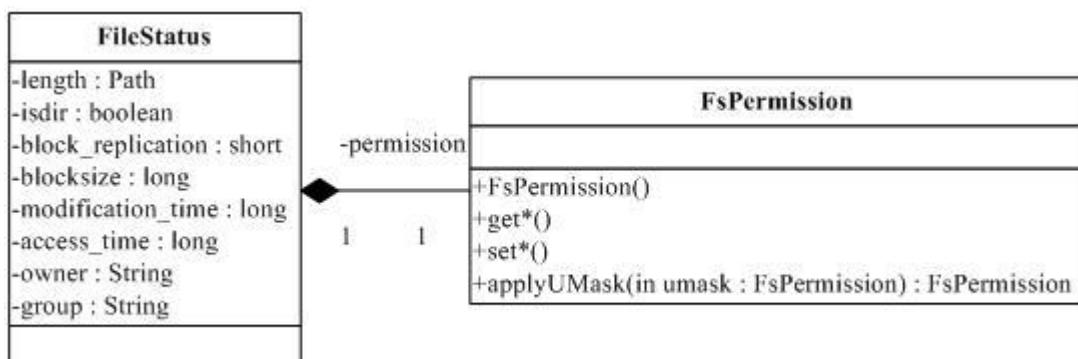
setOwner，设置文件的文件主和文件组，setTimes，设置文件的访问时间和修改时间，setQuota,设置某路径的空间限额和空间额度，和 setPermission 类似，调用 FSDirectory 的对应方法，简单。

```
public boolean setSafeMode(FSConstants.SafeModeAction action) throws IOException;
```

前面我们已经介绍了 NameNode 的安全模式，客户端通过上面的方法，可以让 NameNode 进入 ( SAFEMODE\_ENTER ) / 退出 ( SAFEMODE\_LEAVE ) 安全模式或查询 ( SAFEMODE\_GET ) 状态。FSNamesystem 的 setSafeMode 处理这个命令，对于进入安全模式的请求，如果系统现在不处于安全模式，那么创建一个 SafeModeInfo 对象（创建的这个对象有别于启动时创建的那个 SafeModeInfo，它不会自动退出，因为 threshold=1.5f），这标志着系统进入安全模式。退出安全模式很简单，将 safeMode 赋空就可以啦。

```
public FileStatus[] getListing(String src) throws IOException;
```

分析完 set\*以后，我们来看 get\*。getListing 对应于 UNIX 系统的 ls 命令，返回值是 FileStatus 数组，FileStatus 的类图如下，它其实给出了文件的详细信息，如大小，文件主等等。其实，这些信息都存在 INode\*中，我们只需要把这些信息搬到 FileStatus 中就 OK 啦。FSNamesystem 和 FSDirectory 中都有同名方法，真正干活的地方在 FSDirectory 中。getListing 不需要写日志。



```
public long[] getStats() throws IOException;
```

getStats 得到的是文件系统的信息，UNIX 对应命令为 du，它的实现更简单，所有的信息都存放在 FSNamesystem 对象里。

```
public DatanodeInfo[] getDatanodeReport(FSConstants.DatanodeReportType type)  
throws IOException;
```

getDatanodeReport 获取当前 DataNode 的状态，可能的选项有 DatanodeReportType.ALL, IVE 和 DEAD。FSNamesystem 的同名方法调用 getDatanodeListForReport，通过 HostsFileReader 读取对应信息。

```
public long getPreferredBlockSize(String filename) throws IOException;
```

getPreferredBlockSize，返回 INodeFile.preferredBlockSize，数据块大小。

```
public FileStatus getFileInfo(String src) throws IOException;
```

和 getListing 类似，不再分析。

```
public ContentSummary getContentSummary(String path) throws IOException;
```

得到文件树的一些信息，如下图：

ContentSummary
+length() : long +fileCount() : long +directoryCount() : long +quota() : long +spaceConsumed() : long +spaceQuota() : long

```
public void metaSave(String filename) throws IOException;
```

这个也很简单，它把系统的 metadata 输出/添加到指定文件上（NameNode 所在的文件系统）。

### [Hadoop 源代码分析 \(三零\)](#)

软柿子都捏完了，我们开始啃硬骨头。前面已经分析过 getBlockLocations, create, append, setReplication, setPermission 和 setOwner，接下来我们继续回来讨论和文件内容相关的操作。

```
public void abandonBlock(Block b, String src, String holder  
) throws IOException;
```

abandonBlock 用于放弃一个数据块。普通的文件系统中并没有“放弃”操作，HDFS 出现放弃数据块的原因，如下图所示。当客户端通过其他操作（如下面要介绍的 addBlock 方法）获取 LocatedBlock 后，可以打开到一个 block 的输出流，由于从 DataNode 出错到 NameNode 发现这个信息，需要有一段时间（NameNode 长时间收到 DataNode 心跳），打开输出流可能出错，这时客户端可以向 NameNode 请求放弃这个数据块。



abandonBlock 的处理不是很复杂，首先检查租约（调用 checkLease 方法。block 对应的文件存在，文件处于构造状态，租约拥有者匹配），如果通过检查，调用 FSDirectory 的 removeBlock，从 INodeFileUnderConstruction/BlocksMap/CorruptReplicasMap 中删除 block，然后通过 logOpenFile() 记录变化（logOpenFile 真是万能啊）。

**public LocatedBlock addBlock(String src, String clientName) throws IOException;**

写 HDFS 的文件时，如果数据块被写满，客户端可以通过 addBlock 创建新的数据块。具体的创建工作由 FSNamesystem 的 getAdditionalBlock 方法完成，当然上来就是一通检查（是否安全模式，命名/存储空间限额，租约，数据块副本数，保证 DataNode 已经上报数据块状态），然后通过 ReplicationTargetChooser，选择复制的目标（如果目标数不够副本数，又是一个异常），然后，就可以分配数据块了。allocateBlock 创建一个新的 Block 对象，然后调用 addBlock，检查参数后把数据块加到 BlocksMap 对象和对应的 INodeFile 对象中。allocateBlock 返回后，getAdditionalBlock 还会继续更新一些需要记录的信息，最后返回一个新构造的 LocatedBlock。

**public boolean complete(String src, String clientName) throws IOException;**

当客户端完成对数据块的写操作后，调用 complete 完成写操作。方法 complete 如果返回是 false，那么，客户端需要继续调用 complete 方法。

FSNamesystem 的同名方法调用 completeFileInternal，它会：

检查环境；

获取 src 对应的 INode；

如果 INode 存在，并且处于构造状态，获取数据块；

如果获取数据块返回空 ,返回结果 CompleteFileStatus.OPERATION\_FAILED ,FSNamesystem 的 complete 会抛异常返回 ;

如果上报文件完成的 DataNode 数不够系统最小的副本数 , 返回 STILL\_WAITING ;

调用 finalizeINodeFileUnderConstruction ;

返回成功 COMPLETE\_SUCCESS

其中 , 对 finalizeINodeFileUnderConstruction 的处理包括 :

释放租约 ;

将对应的 INodeFileUnderConstruction 对象转换为 INodeFile 对象 , 并在 FSDirectory 进行替换 ;

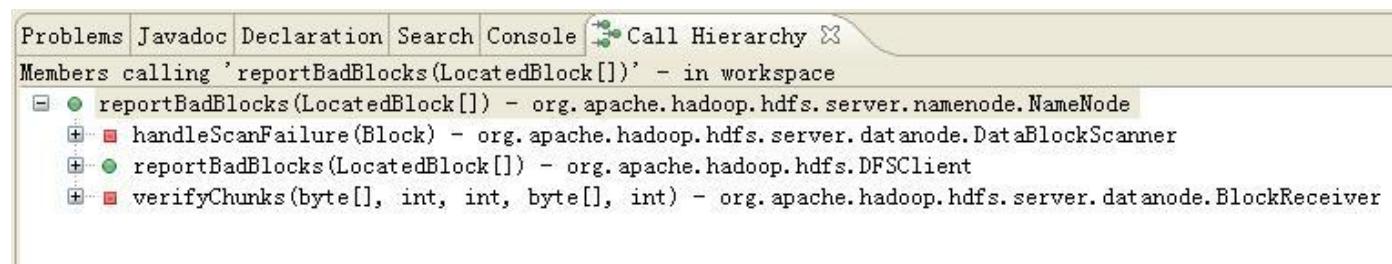
调用 FSDirectory.closeFile 关闭文件 , 其中会写日志 logCloseFile(path, file)。

检查副本数 , 如果副本数小于 INodeFile 中的目标数 , 那么添加数据块复制任务。

我们可以看到 , complete 一个文件还是比较复杂的 , 需要释放很多的资源。

**public void reportBadBlocks(LocatedBlock[] blocks) throws IOException;**

调用 reportBadBlocks 的地方比较多 , 客户端可能调用 , DataNode 上也可能调用。



由于上报的是个数组 , reportBadBlocks 会循环处理 , 调用 FSNamesystem 的 markBlockAsCorrupt 方法。

markBlockAsCorrupt 方法需要两个参数 , blk ( 数据块 ) 和 dn ( 所在的 DataNode 信息 ) 。如果系统目前副本数大于要求 , 那么直接调用 invalidateBlock 方法。

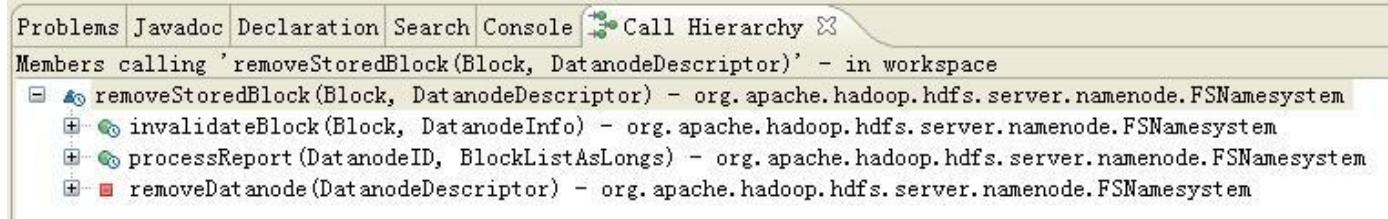
方法 invalidateBlock 很简单 , 在检查完系统环境以后 , 先调用 addToInvalidates 方法往 FSNamesystem.recentInvalidateSets 添加一项 , 然后调用 removeStoredBlock 方法。

removeStoredBlock 被多个方法调用 , 它会执行 :

从 BlocksMap 中删除记录 removeNode(block, node) ;

如果目前系统中还有其他副本，调用 decrementSafeBlockCount ( 可能的调整安全模式参数 ) 和 updateNeededReplications ( 跟新可能存在的 block 复制信息，例如，现在系统中需要复制 1 个数据块，那么更新后，需要复制 2 个数据块 ) ；

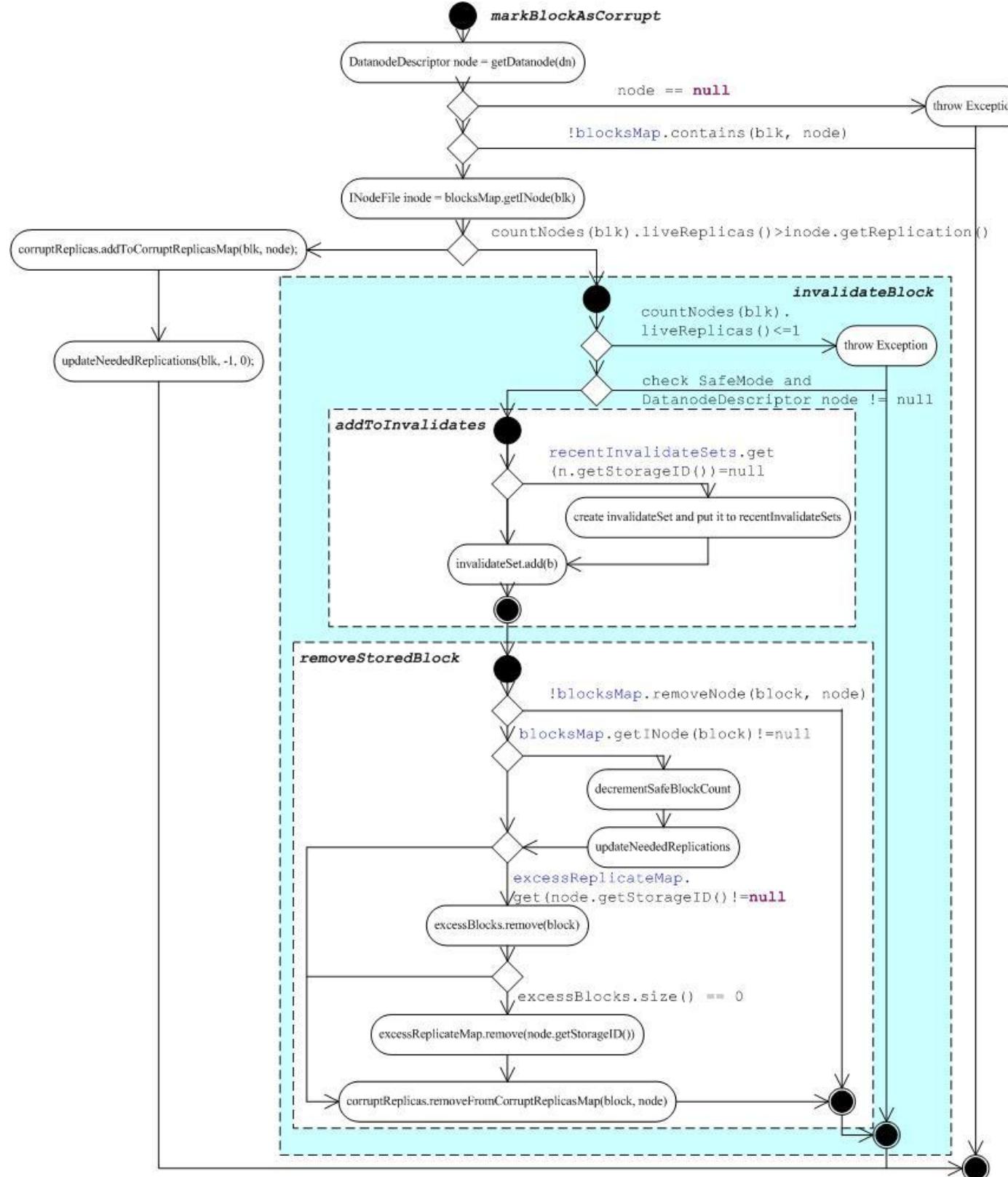
如果目前系统中有多余数据块等待删除 ( 在 excessReplicateMap 中 ) ，那么移除对应记录；  
删除在 CorruptReplicasMap 中的记录 ( 可能有 ) 。



removeStoredBlock 其实也是涉及了多处表操作，包括 BlocksMap , excessReplicateMap 和 CorruptReplicasMap。

我们回到 markBlockAsCorrupt , 如果系统目前副本数小于要求，那么很显然，我们需要对数据块进行复制。首先将现在的数据块加入到 CorruptReplicasMap 中，然后调用 updateNeededReplications , 跟新复制信息。

markBlockAsCorrupt 这个流程太复杂了，我们还是画个图吧：



## [Hadoop 源代码分析 \(三一\)](#)

下面是和目录树相关的方法。

```
public boolean rename(String src, String dst) throws IOException;
```

更改文件名。调用 FSNamesystem 的 renameTo , 干活的是 renameToInternal , 最终调用 FSDirectory 的 renameTo 方法 , 如果成功 , 更新租约的文件名 , 如下 :

```
changeLease(src, dst, dinfo);
```

```
public boolean delete(String src) throws IOException;
```

```
public boolean delete(String src, boolean recursive) throws IOException;
```

第一个已经废弃不用 , 使用第二个方法。

最终使用 deleteInternal , 该方法调用 FSDirectory.delete()。

```
public boolean mkdirs(String src, FsPermission masked) throws IOException;
```

在做完一系列检查以后，调用 FSDirectory.mkdirs()。

```
public FileStatus[] getListing(String src) throws IOException;
```

前面我们已经讨论了。

下面是其它和系统维护管理的方法。

```
public void renewLease(String clientName) throws IOException;
```

就是调用了一下 leaseManager.renewLease(holder)，没有其他的事情需要做，简单。

```
public void refreshNodes() throws IOException;
```

还记得我们前面分析过 NameNode 上有个 DataNode 在线列表和 DataNode 离线列表吗，这个命令可以让 NameNode 重新读这两个文件。当然，根据前后 DataNode 的状态，一共有 4 种情况，其中有 3 种需要修改。

对于从工作状态变为离线的，需要将上面的 DataNode 复制到其他的 DataNode，需要调用 updateNeededReplications 方法（前面我们已经讨论过这个方法了）。

对于从离线变为工作的 DataNode，只需要改变一下状态。

```
public void finalizeUpgrade() throws IOException;
```

finalize 一个升级，确认客户端有超级用户权限以后，调用 FSImage.finalizeUpgrade()。

```
public void fsync(String src, String client) throws IOException;
```

将文件信息持久化。在检查租约信息后，调用 FSDirectory 的 persistBlocks，将文件的原信息通过 logOpenFile(path, file) 写日志。

## Hadoop 源代码分析 (三二)

搞定 ClientProtocol，接下来是 DatanodeProtocol 部分。接口如下：

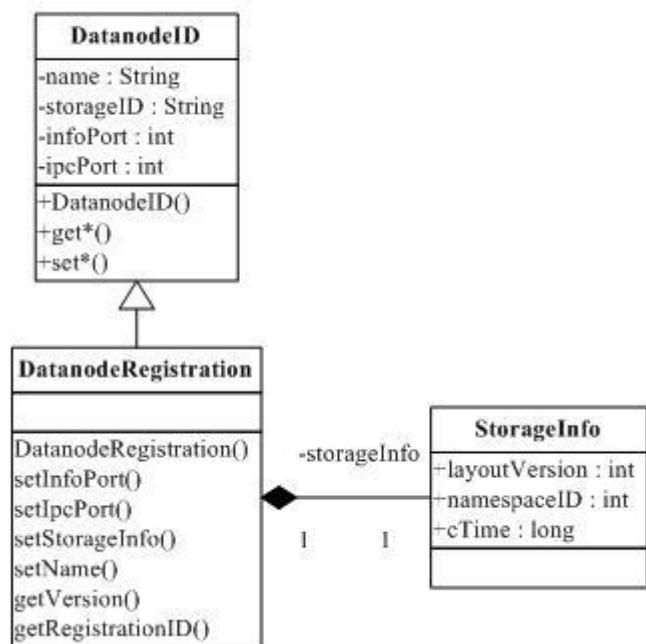
```
<<接口>>
DatanodeProtocol
+register(in registration : DatanodeRegistration) : DatanodeRegistration
+sendHeartbeat(in registration : DatanodeRegistration, in capacity : long, in dfsUsed : long, in remaining : long, in xmitsInProgress : int, in xceiverCount : int) : DatanodeCommand
+blockReport(in registration : DatanodeRegistration, in blocks : long[]) : DatanodeCommand
+blockReceived(in registration : DatanodeRegistration, in blocks : Block[], in delHints : String[])
+errorReport(in registration : DatanodeRegistration, in errorCode : int, in msg : String) : NamespaceInfo
+versionRequest()
+processUpgradeCommand(in comm : UpgradeCommand) : UpgradeCommand
+reportBadBlocks(in blocks : LocatedBlock[])
+nextGenerationStamp(in block : Block) : long
+commitBlockSynchronization(in block : Block, in newgenerationstamp : long, in newlength : long, in closeFile : boolean, in deleteblock : boolean, in newtargets : DatanodeID[])
```

```

public DatanodeRegistration register(DatanodeRegistration nodeReg
) throws IOException

```

用于 DataNode 向 NameNode 登记。输入和输出参数都是 DatanodeRegistration , 类图如下 :



前面讨论 DataNode 的时候 , 我们已经讲过了 DataNode 的注册过程 , 我们来看 NameNode 的过程。下面是主要步骤 :

检查该 DataNode 是否能接入到 NameNode ;

准备应答 , 更新请求的 DatanodeID ;

从 datanodeMap ( 保存了 StorageID à DatanodeDescriptor 的映射 , 用于保证 DataNode 使用的 Storage 的一致性 ) 得到对应的 DatanodeDescriptor , 为 nodeS ;

从 Host2NodesMap ( 主机名到 DatanodeDescriptor 数组的映射 ) 中获取 DatanodeDescriptor , 为 nodeN ;

如果 `nodeN!=null` 同时 `nodeS!=nodeN` ( 后面的条件表明表明 DataNode 上使用的 Storage 发生变化 ) , 那么我们需要先在系统中删除 nodeN ( `removeDatanode` , 下面再讨论 ) , 并在 Host2NodesMap 中删除 nodeN ;

如果 nodeS 存在 , 表明前面已经注册过 , 则 :

1. 更新网络拓扑 ( 保存在 NetworkTopology ) , 首先在 NetworkTopology 中删除 nodeS , 然后跟新 nodeS 的相关信息 , 调用 `resolveNetworkLocation` , 获得 nodeS 的位置 , 并从新加到 NetworkTopology 里 ;
2. 更新心跳信息 ( `register` 也是心跳 ) ;

如果 nodeS 不存在，表明这是一个新注册的 DataNode，执行

1. 如果注册信息的 storageID 为空，表明这是一个全新的 DataNode，分配 storageID；
2. 创建 DatanodeDescriptor，调用 resolveNetworkLocation，获得位置信息；
3. 调用 unprotectedAddDatanode（后面分析）添加节点；
4. 添加节点到 NetworkTopology 中；
5. 添加到心跳数组中。

上面的过程，我们遗留了两个方法没分析，removeDatanode 的流程如下：

更新系统状态，包括 capacityTotal，capacityUsed，capacityRemaining 和 totalLoad；

从心跳数组中删除节点，并标记节点 isAlive 属性为 false；

从 BlocksMap 中删除这个节点上的所有 block，用了（三零）分析到的 removeStoredBlock 方法；

调用 unprotectedAddDatanode；

从 NetworkTopology 中删除节点信息。

unprotectedAddDatanode 很简单，它只是更新了 Host2NodesMap 的信息。

### [Hadoop 源代码分析（三三）](#)

下面来看一个家伙：

```
public DatanodeCommand sendHeartbeat(DatanodeRegistration nodeReg,  
                                      long capacity,  
                                      long dfsUsed,  
                                      long remaining,  
                                      int xmitsInProgress,  
                                      int xceiverCount) throws IOException
```

DataNode 发送到 NameNode 的心跳信息。细心的人会发现，请求的内容还是 DatanodeRegistration，应答换成 DatanodeCommand 了。 DatanodeCommand 类图如下：

前面介绍 DataNode 时，已经分析过了 DatanodeCommand 支持的命令：

DNA\_TRANSFER：拷贝数据块到其他 DataNode

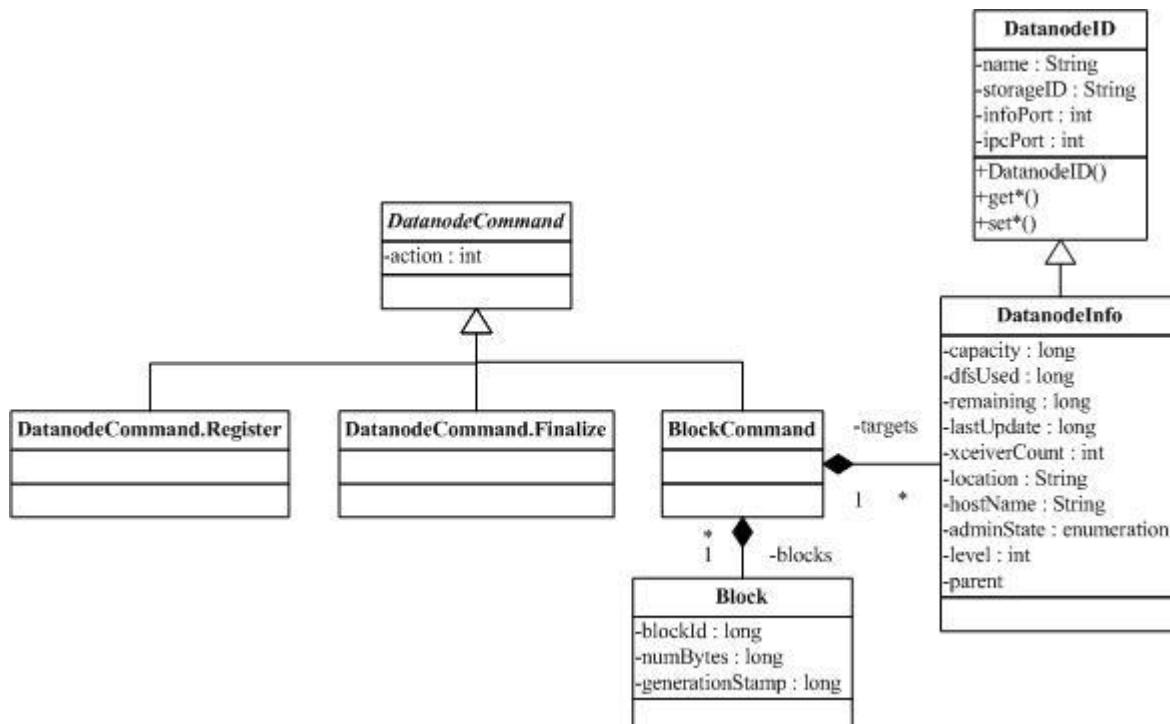
DNA\_INVALIDATE：删除数据块

*DNA\_SHUTDOWN* : 关闭 DataNode

*DNA\_REGISTER* : DataNode 重新注册

*DNA\_FINALIZE* : 提交升级

*DNA\_RECOVERBLOCK* : 恢复数据块



有了上面这些基础，我们来看 FSNamesystem.handleHeartbeat 的处理过程：

调用 `getDatanode` 方法找对应的 `DatanodeDescriptor`，保存于变量 `nodeinfo`（可能为 null）中，如果现有 `NameNode` 上记录的 `StorageID` 和请求的不一样，返回 `DatanodeCommand.REGISTER`，让 `DataNode` 重新注册。

如果发现当前节点需要关闭（已经 `isDecommissioned`），抛异常 `DisallowedDatanodeException`。

`nodeinfo` 是空或者现在状态不是活的，返回 `DatanodeCommand.REGISTER`，让 `DataNode` 重新注册。

更新系统的状态，包括 `capacityTotal`，`capacityUsed`，`capacityRemaining` 和 `totalLoad`；

接下来按顺序看有没有可能的恢复数据块/拷贝数据块到其他 `DataNode`/删除数据块/升级命令（不讨论）。一次返回只能有一条命令，按上面优先顺序。

下面分析应答的命令是如何构造的。

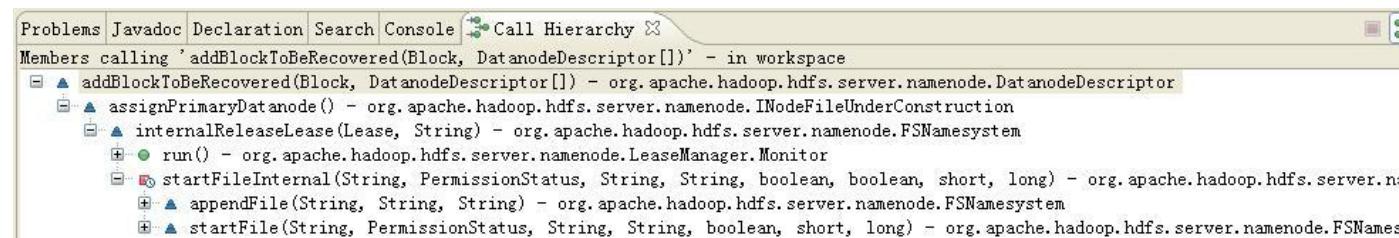
首先是 `DNA_RECOVERBLOCK`（恢复数据块），那是个非常长的流程，同时需要回去讨论 `DataNode` 上的一些功能，我们在后面介绍它。

对于 DNA\_TRANSFER (拷贝数据块到其他 DataNode) , 从 DatanodeDescriptor.replicateBlocks 中取出尽可能多的项目，放到 BlockCommand 中。在 DataNode 中，命令由 transferBlocks 执行，前面我们已经分析过啦。

删除数据块 DNA\_INVALIDATE 也很简单，从 DatanodeDescriptor.invalidateBlocks 中获取尽可能多的项目，放到 BlockCommand 中，DataNode 中的动作，我们也分析过。

我们来讨论 DNA\_RECOVERBLOCK (恢复数据块)，在讨论 DataNode 的过程中，我们没有讲这个命令是用来干什么的，还有它在 DataNode 上的处理流程，是好好分析分析这个流程的时候了。DNA\_RECOVERBLOCK 命令通过 DatanodeDescriptor.getLeaseRecoveryCommand 获取，获取过程很简单，将 DatanodeDescriptor 对象中队列 recoverBlocks 的所有内容取出，放入 BlockCommand 的 Block 中，设置 BlockCommand 为 DNA\_RECOVERBLOCK，就 OK 了。

关键是，这个队列里的信息是用来干什么的。我们先来看那些操作会向这个队列加东西，调用关系图如下：



租约有两个超时时间，一个被称为软超时（1分钟），另一个是硬超时（1小时）。如果租约软超时，那么就会触发 internalReleaseLease 方法，如下：

**void internalReleaseLease(Lease lease, String src) throws IOException**

该方法执行：

检查 src 对应的 INodeFile，如果不存在，不处于构造状态，返回；

文件处于构造状态，而文件目标 DataNode 为空，而且没有数据块，则 finalize 该文件（该过程在 completeFileInternal 中已经讨论过，租约在过程中被释放），并返回；

文件处于构造状态，而文件目标 DataNode 为空，数据块非空，则将最后一个数据块存放的 DataNode 目标取出（在 BlocksMap 中），然后设置为文件现在的目标 DataNode；

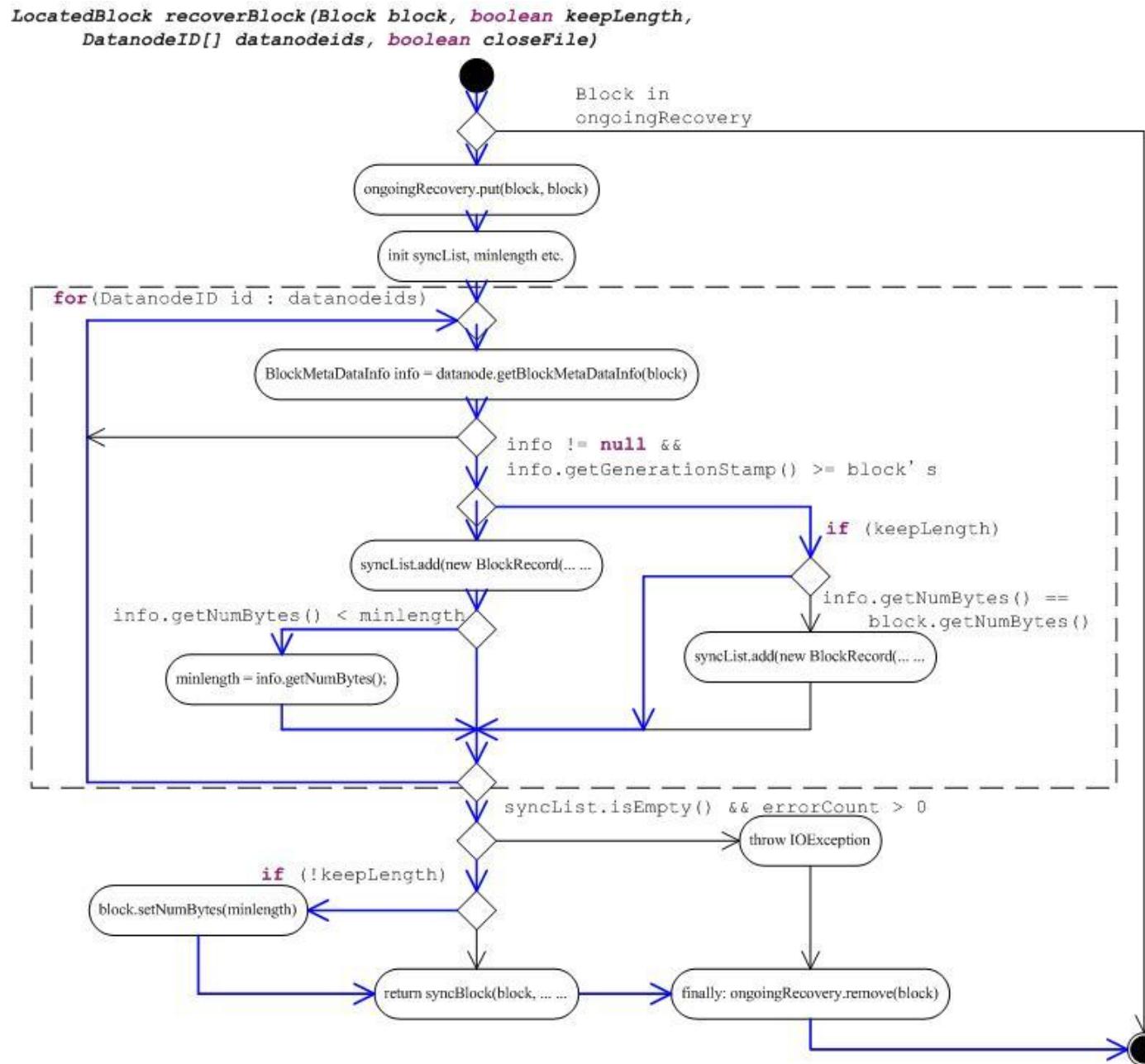
调用 `INodeFileUnderConstruction.assignPrimaryDatanode` , 该过程会挑选一个目前还活着的 DataNode , 作为租约的主节点 , 并把`<block , block 目标 DataNode 数组>`加到该 DataNode 的 `recoverBlocks` 队列中 ;

更新租约。

上面分析了租约软超时的情况下 NameNode 发生租约恢复的过程。DataNode 上收到这个命令后 , 将会启动一个新的线程 , 该线程为每个 Block 调用 `recoverBlock` 方法 : `recoverBlock(blocks[i], false, targets[i], true)`。

```
private LocatedBlock recoverBlock(Block block, boolean keepLength,
DatanodeID[] datanodeids, boolean closeFile) throws IOException
```

它的流程并不复杂 , 但是分支很多 , 如下图 ( 蓝线是上面输入 , 没有异常走的流程 ) :



首先是判断进来的 Block 是否在 `ongoingRecovery` 中 , 如果存在 , 返回 , 不存在 , 加到 `ongoingRecovery` 中。

接下来是个循环( 框内部分是循环体 , 奇怪 , 没找到表示循环的符号 ) , 对每一个 DataNode , 获取 Block 的 `BlockMetaDataInfo` ( 下面还会分析 ) , 这需要调用到 DataNode 间通信的接口上的方法 `getBlockMetaDataInfo` 。然后分情况看要不要把信息保存下来 ( 图中间的几个判断 ) , 其中包括要进行同步的节点。

根据参数 , 更新数据块信息 , 然后调用 `syncBlock` 并返回 `syncBlock` 生产的 `LocatedBlock`。

上面的这一圈，对于我们这个输入常数来说，就是把 Block 的长度，更新成为拥有最新时间戳的最小长度值，并得到要更新的节点列表，然后调用 syncBlock 更新各节点。

getBlockMetaDataInfo 用于获取 Block 的 BlockMetaDataInfo，包括 Block 的 generationStamp，最后校验时间，同时它还会检查数据块文件的元信息，如果出错，会抛出异常。

syncBlock 定义如下：

```
private LocatedBlock syncBlock(Block block, List<BlockRecord> syncList,  
    boolean closeFile)
```

它的流程是：

如果 syncList 为空，通过 commitBlockSynchronization 向 NameNode 提交这次恢复；

syncList 不为空，那么先 NameNode 申请一个新的 Stamp，并根据上面得到的长度，构造一个新的数据块信息 newblock；

对于没一个 syncList 中的 DataNode，调用它们上面的 updateBlock，更新信息；更新信息如果返回 OK，记录下来；

如果更新了信息的 DataNode 不为空，调用 commitBlockSynchronization 提交这次恢复；并生成 LocatedBlock；

如果更新的 DataNode 为空，抛异常。

通过 syncBlock，所有需要恢复的 DataNode 上的 Block 信息都被更新。

DataNode 上的 updateBlock 方法我们前面已经介绍了，就不再分析。

下面我们来看 NameNode 的 commitBlockSynchronization 方法，它在上面的过程中用于提交数据块恢复：

```
public void commitBlockSynchronization(Block block,  
    long newgenerationstamp, long newlength,  
    boolean closeFile, boolean deleteblock, DatanodeID[] newtargets  
)
```

参数分别是 block，数据块；newgenerationstamp，新的时间戳；newlength，新长度；closeFile，是否关闭文件；deleteblock，是否删除文件；newtargets，新的目标列表。

上面的两次调用，输入参数分别是：

```
commitBlockSynchronization(block, 0, 0, closeFile, true, DatanodeID.EMPTY_ARRAY);
```

```
commitBlockSynchronization(block, newblock.getGenerationStamp(), newblock.getNumBytes(), closeFile, false,
```

```
nlist);
```

处理流程是：

参数检查；

获取对应的文件，记为 pendingFile；

BlocksMap 中删除老的信息；

如果 deleteblock 为 true，从 pendingFile 删除 Block 记录；

否则，更新 Block 的信息；

如果不关闭文件，那么写日志保存更新，返回；

关闭文件的话，调用 finalizeINodeFileUnderConstruction。

这块比较复杂，不仅涉及了 NameNode 和 DataNode 间的通信，而且还存在对于 DataNode 和 DataNode 间的通信（DataNode 间的通信就只支持这两个方法，如下图）。后面介绍 DFSClient 的时候，我们还会再回来分析它的功能，以获取全面的理解。



## Hadoop 源代码分析（三四）

继续对 NameNode 实现的接口做分析。

```
public DatanodeCommand blockReport(DatanodeRegistration nodeReg,
                                    long[] blocks) throws IOException
```

DataNode 向 NameNode 报告它拥有的所有数据块，其中，参数 blocks 包含了数组化以后数据块的信息。

FSNamesystem.processReport 处理这个请求。一番检查以后，调用 DatanodeDescriptor 的 reportDiff，将上报的数据块分成三组，分别是：

删除：其它情况；

加入：BlocksMap 中有数据块，但目前的 DatanodeDescriptor 上没有对应信息；

使无效：BlocksMap 中没有找到数据块。

对于删除的数据块，调用 removeStoredBlock，这个方法我们前面已经分析过啦。

对应需要加入的数据块，调用 addStoredBlock 方法，处理流程如下：

从 BlocksMap 获取现在的信息，记为 storedBlock；如果为空，返回；

记录 block 和 DatanodeDescriptor 的关系；

新旧数据块记录不是同一个（我们这个流程是肯定不是啦）：

1. 如果现有数据块长度为 0，更新为上报的 block 的值；
2. 如果现有数据块长度比新上报的长，invalidateBlock（前面分析过，很简单的一个方法）当前数据块；
3. 如果现有数据块长度比新上报的小，那么会删除所有老的数据块（还是通过 invalidateBlock），并更新 BlocksMap 中数据块的大小信息；
4. 跟新可用存储空间等信息；

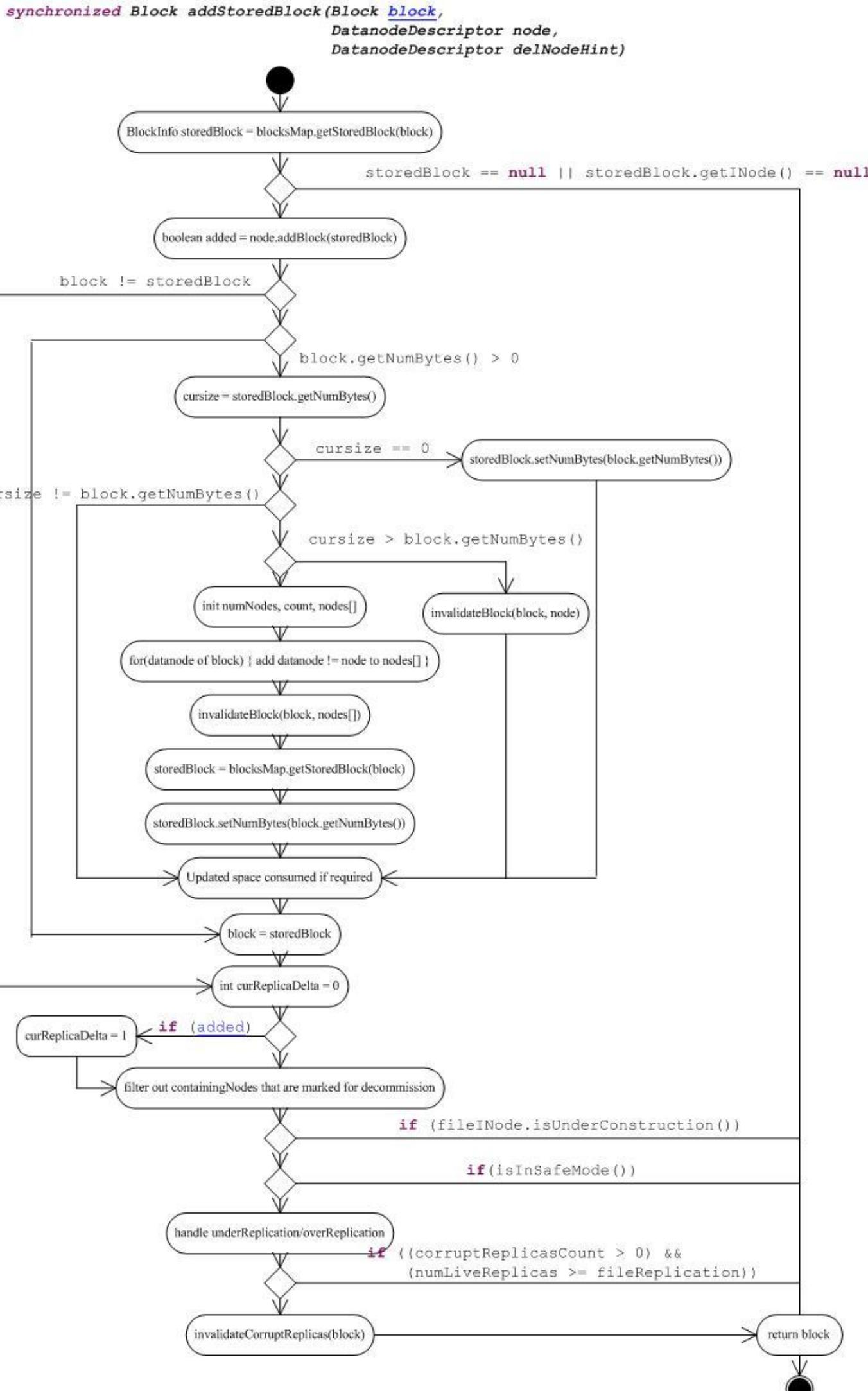
根据情况确定数据块需要复制的数目和目前副本数；

如果文件处于构建状态或系统现在是安全模式，返回；

处理当前副本数和文件的目标副本数不一致的情况；

如果当前副本数大于系统设定门限，开始删除标记为无效的数据块。

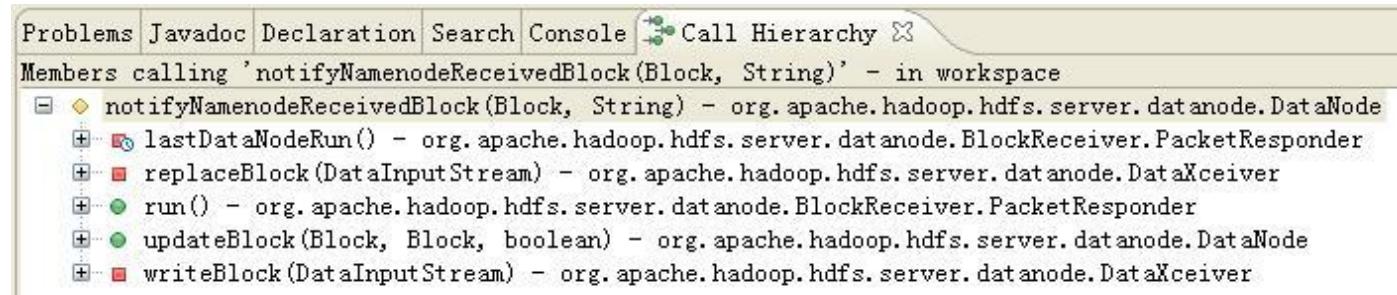
还是给个流程图吧：



对于标记为使无效的数据块，调用 `addToInvalidates` 方法，很简单的方法，直接加到 `FSNamesystem` 的成员变量 `recentInvalidateSets` 中。

```
public void blockReceived(DatanodeRegistration registration,  
Block blocks[],  
String[] delHints)
```

DataNode 可以通过 `blockReceived`，向 NameNode 报告它最近接受到的数据块，同时给出如果数据块副本数太多时，可以删除数据块的节点（参数 `delHints`）。在 DataNode 中，这个信息是通过方法 `notifyNamenodeReceivedBlock`，记录到对应的列表中。



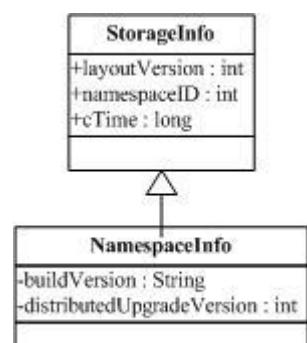
NameNode 上的处理不算复杂，对输入参数进行检查以后，调用上面分析的 `addStoredBlock` 方法。然后在 `PendingReplicationBlocks` 对象中删除相应的 block。

```
public void errorReport(DatanodeRegistration registration,
                        int errorCode,
                        String msg)
```

向 NameNode 报告 DataNode 上的一个错误，如果错误是硬盘错，会删除该 DataNode，其它情况只是简单地记录收到一条出错信息。

```
public NamespaceInfo versionRequest() throws IOException;
```

从 NameNode 上获取 NamespaceInfo，该信息用于构造 DataNode 上的 DataStorage。



```
UpgradeCommand processUpgradeCommand(UpgradeCommand comm) throws IOException;
```

我们不讨论。

```
public void reportBadBlocks(LocatedBlock[] blocks) throws IOException
```

报告错误的数据块。NameNode 会循环调用 FSNamesystem 的 markBlockAsCorrupt 方法。处理流程不是很复杂，找对应的 INodeFile，如果副本数够，那么调用 invalidateBlock，使该 DataNode 上的 Block 无效；如果副本数不够，加 Block 到 CorruptReplicasMap 中，然后准备对好数据块进行复制。

目前为止，我们已经完成了 NameNode 上的 ClientProtocol 和 DatanodeProtocol 的分析了，NamenodeProtocol 我们在理解从 NameNode 的时候，才会进行分析。

### Hadoop 源代码分析（三五）

除了对外提供的接口，NameNode 上还有一系列的线程，不断检查系统的状态，下面是这些线程的功能分析。

在 NameNode 中，定义了如下线程：

```
Daemon hbthread = null; // HeartbeatMonitor thread  
public Daemon lmthread = null; // LeaseMonitor thread  
Daemon smmthread = null; // SafeModeMonitor thread  
public Daemon replthread = null; // Replication thread  
private Daemon dnthread = null;
```

PendingReplicationBlocks 中也有一个线程：

```
Daemon timerThread = null;
```

NameNode 内嵌的 HTTP 服务器中自然也有线程，这块我们就不分析啦。

```
HttpServer infoServer;
```

心跳线程用于对 DataNode 的心态进行检查，以间隔 heartbeatRecheckInterval 运行 heartbeatCheck 方法。如果在一定时间内没收到 DataNode 的心跳信息，我们就认为该节点已经死掉，调用 removeDatanode ( 前面分析过 ) 将 DataNode 标记为无效。

租约 lmthread 用于检查租约的硬超时，如果租约硬超时，调用前面分析过的 internalReleaseLease，释放租约。

smmthread 运行的 SafeModeMonitor 我们前面已经分析过了。

replthread 运行 ReplicationMonitor，这个线程会定期调用 computeDatanodeWork 和 processPendingReplications。

computeDatanodeWork 会执行 computeDataNodeWork 或 computeInvalidateWork。computeDataNodeWork 从 neededReplications 中扫描，取出需要复制的项，然后：

检查文件不存在或者处于构造状态；如果是，从队列中删除复制项，退出对复制项的处理( 接着处理下一个 )；

得到当前数据块副本数并选择复制的源 DataNode，如果空，退出对复制项的处理；

再次检查副本数（很可能有 DataNode 从故障中恢复），如果发现不需要复制，从队列中删除复制项，退出对复制项的处理；

选择复制的目标，如果目标空，退出对复制项的处理；

将复制的信息（数据块和目标 DataNode）加入到源目标 DataNode 中；在目标 DataNode 中记录复制请求；

从队列中将复制项移动到 pendingReplications。

可见，这个方法执行后，复制项从 neededReplications 挪到 pendingReplications 中。DataNode 在某次心跳的应答中，可以拿到相应的信息，执行复制操作。

computeInvalidateWork 当然是用于删除无效的数据块。它的主要工作在 invalidateWorkForOneNode 中完成。和上面 computeDatanodeWork 类似，不过它的处理更简单，将 recentInvalidateSets 的数据通过 DatanodeDescriptor.addBlocksToBeInvalidated 挪到 DataNode 中。

dnthread 执行的是 DecommissionedMonitor，它的 run 方法周期调用 decommissionedDatanodeCheck，再到 checkDecommissionStateInternal，定期将完成 Decommission 任务的 DataNode 状态从 DECOMMISSION\_INPROGRESS 改为 DECOMMISSIONED。

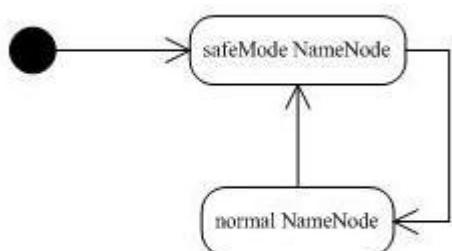
PendingReplicationMonitor 中的线程用于对处在等待复制状态的数据块进行检查。如果发现长时间该数据块没被复制，那么会将它挪到 timedOutItems 中。请参考 PendingReplicationBlocks 的讨论。

infoServer 的相关线程我们就不分析了，它们都用于处理 HTTP 请求。

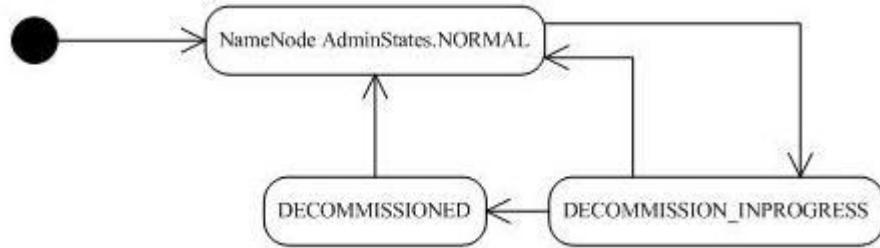
上面已经总结了 NameNode 上的一些为特殊任务启动的线程，除了这些线程，NameNode 上还运行着 RPC 服务器的相关线程，具体可以看前面章节。

在我们开始分析 Secondary NameNode 前，我们给出了以 NameNode 上一些状态转移图，大家可以通过这个图，更好理解 NameNode。

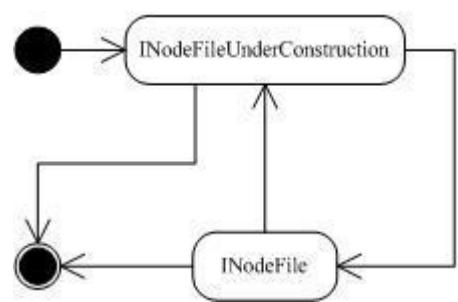
NameNode：



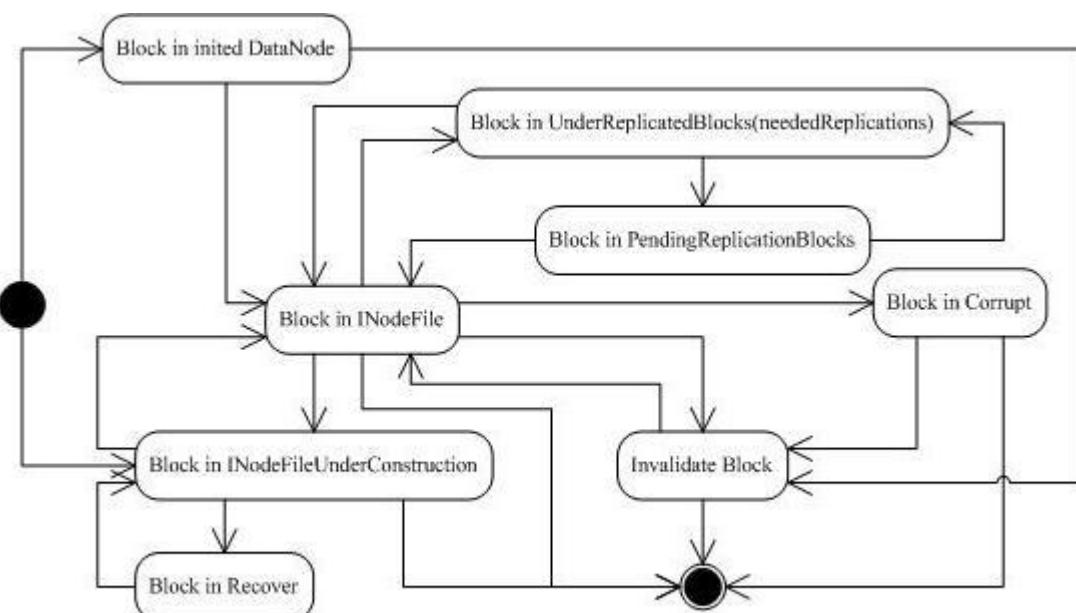
DataNode：



文件：



Block，比较复杂：



上面的图不是很严格，只是用于帮助大家理解 NameNode 对 Block 复杂的处理过程。

稍微说明一下，“Block in initied DataNode” 表明这个数据块在一个刚初始化的 DataNode 上。“Block in INodeFile” 是数据块属于某个文件，“Block in INodeFileUnderConstruction” 表明这数据块属于一个正在构建的文件，当然，处于这个状态的 Block 可能因为租约恢复而转移到“Block in Recover”。右上方描述了需要复制的数据块的状态，

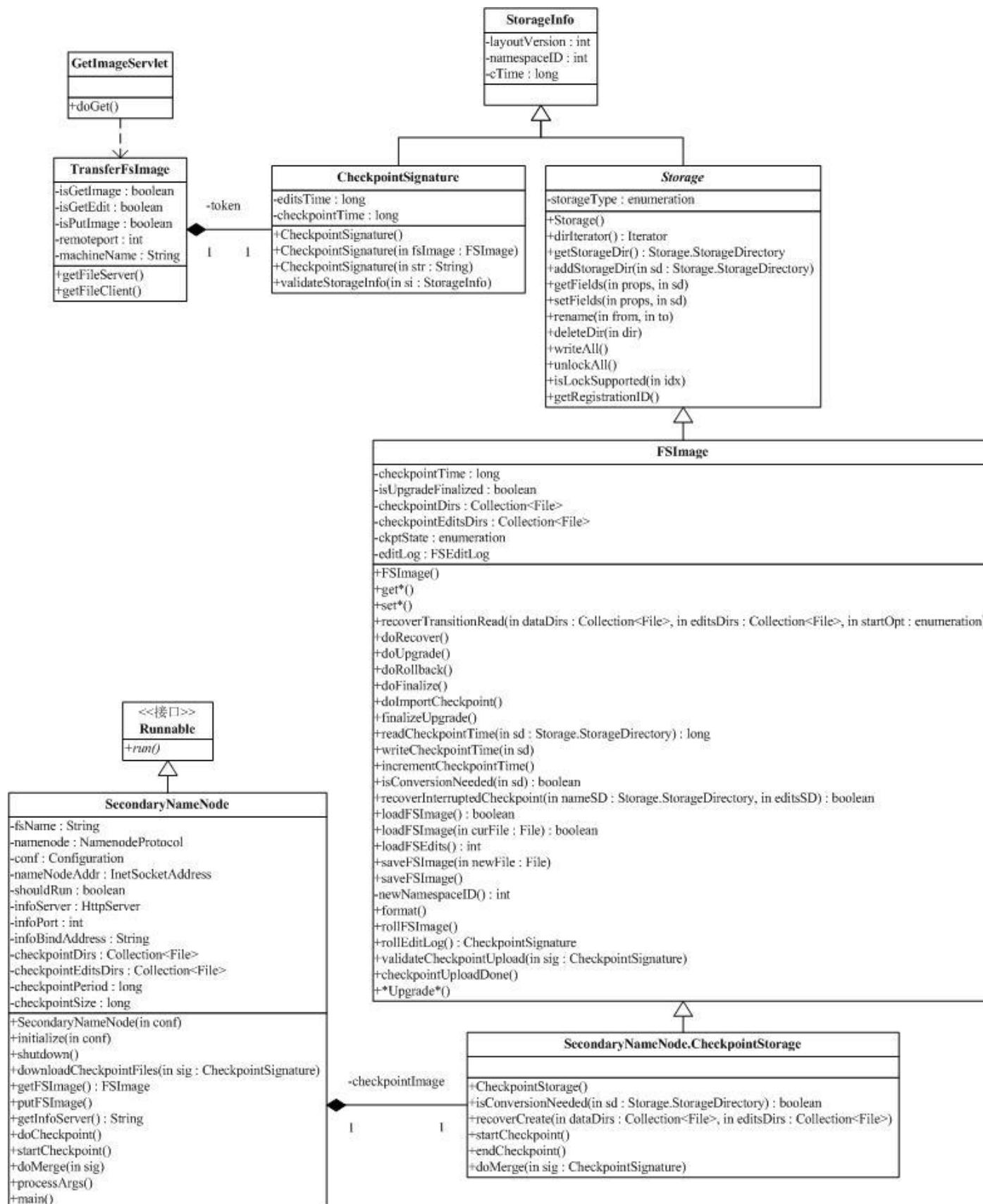
UnderReplicatedBlocks 和 PendingReplicationBlocks 的区别在于 Block 是否被插入到某一个 DatanodeDescriptor 中。

Corrupt 和 Invalidate 的就好理解啦。

## Hadoop 源代码分析 (三六)

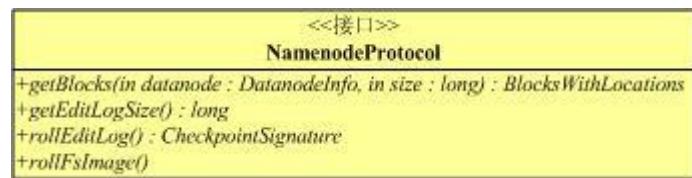
转战进入 Secondary NameNode ,前面的分析我们有事也把它称为从 NameNode ,从 NameNode 在 HDFS 里是个小配角。

跟 Secondary NameNode 有关的类不是很多 ,如下图 :



首先要讨论的是 NameNode 和 Secondary NameNode 间的通信。NameNode 上实现了接口 NamenodeProtocol ( 如下图 ) ,就是用于 NameNode 和 Secondary NameNode 间的命令通信。

NameNode 和 Secondary NameNode 间数据的通信，使用的是 HTTP 协议，HTTP 的容器用的是 jetty，TransferFsImage 是文件传输的辅助类。



GetImageServlet 的 doGet 方法目前支持取 FSImage(getimage) , 取日志(getedit)和存 FSImage(putimage)。例如：

<http://localhost:50070/getimage?getimage>

可以获取 FSImage。

<http://localhost:50070/getimage?getedit>

可以获取日志文件。

保存 FSImage 需要更多的参数，它的流程很好玩，Secondary NameNode 发送一个 HTTP 请求到 NameNode，启动 NameNode 上一个 HTTP 客户端到 Secondary NameNode 上去下载 FSImage，下载需要的一些信息都放在从 NameNode 的 HTTP 请求中。

我们先来考察 Secondary NameNode 持久化保存的信息：

```
[hadoop@localhost namesecondary]$ ls -R
.:
current  image  in_use.lock  previous.checkpoint

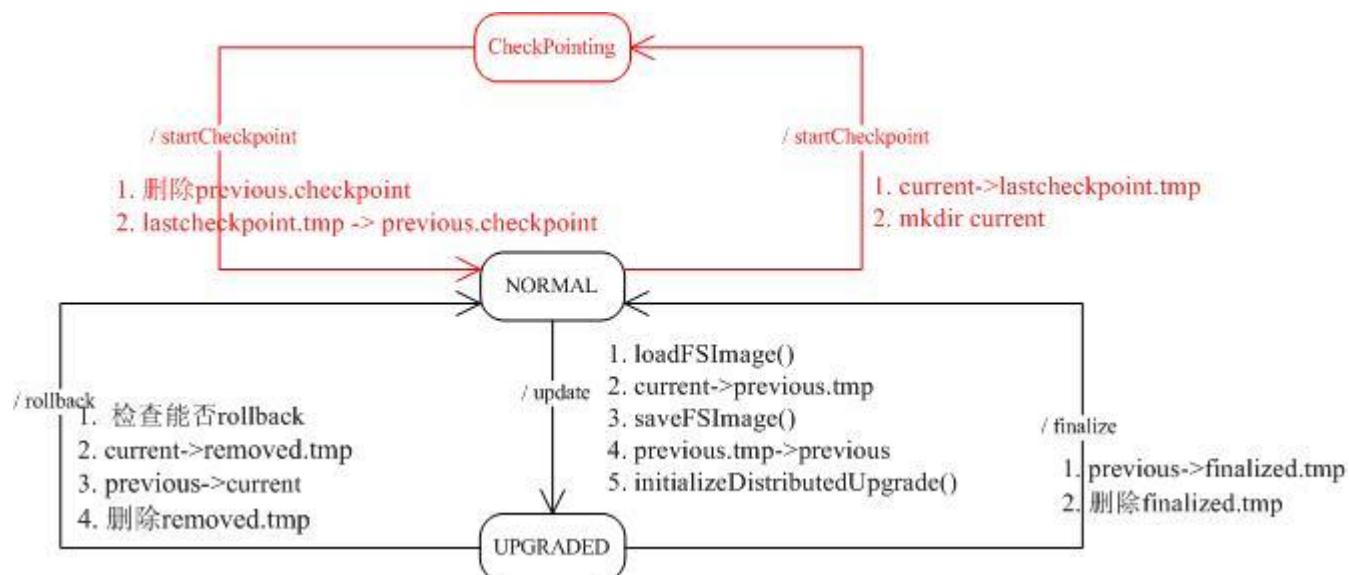
./current:
edits  fsimage  fstime  VERSION

./image:
fsimage

./previous.checkpoint:
edits  fsimage  fstime  VERSION
```

in\_use.lock 的用法和前面 NameNode , DataNode 的是一样的。对比 NameNode 保存的信息，我们可以发现 Secondary NameNode 上保存多了一个 previous.checkpoint。CheckpointStorage 就是应用于 Secondary NameNode 的存储类，它继承自 FSImage，只添加了很少的方法。

previous.checkpoint 目录保存了上一个 checkpoint 的信息( current 里的永远是最新的 ) ,临时目录用于创建新 checkpoint ,成功后 ,老的 checkpoint 保存在 previous.checkpoint 目录中。状态图如下 ( 基类 FSImage 用的是黑色 ) :



至于上面目录下文件的内容 , 和 FSImage 是一样的。

CheckpointStorage 除了上面图中的 startCheckpoint 和 endCheckpoint 方法 ( 上图给出了正常流程 ) , 还有 :

```
void recoverCreate(Collection<File> dataDirs,  
Collection<File> editsDirs) throws IOException
```

和 FSImage.coverTransitionRead 类似 , 用于分析现有目录 , 创建目录 ( 如果不存在 ) 并从可能的错误中恢复。

```
private void doMerge(CheckpointSignature sig) throws IOException
```

doMerge 被类 SecondaryNameNode 的同名方法调用 , 我们后面再分析。

## [Hadoop 源代码分析 \( 三七 \)](#)

Secondary NameNode 的成员变量很少 , 主要的有 :

```
private CheckpointStorage checkpointImage;
```

Secondary NameNode 使用的 Storage

```
private NamenodeProtocol namenode;
```

和 NameNode 通信的接口

```
private HttpServer infoServer;
```

传输文件用的 HTTP 服务器

main 方法是 Secondary NameNode 的入口 , 它最终启动线程 , 执行 SecondaryNameNode 的 run 。启动前的对 SecondaryNameNode 的构造过程也很简单 , 主要是创建和 NameNode 通信的接口和启动 HTTP 服务器。

SecondaryNameNode 的 run 方法每隔一段时间执行 doCheckpoint()，从 NameNode 的主要工作都在这一个方法里。这个方法，总的来说，会从 NameNode 上取下 FSImage 和日志，然后再本地合并，再上传回 NameNode。这个过程结束后，从 NameNode 上保持了 NameNode 上持久化信息的一个备份，同时，NameNode 上已经完成合并到 FSImage 的日志可以抛弃，一箭双雕。

具体的流程是：

- 1：调用 startCheckpoint，为接下来的工作准备空间。startCheckpoint 会在内部做一系列的检查，然后调用 CheckpointStorage 的 startCheckpoint 方法，创建目录。
- 2：调用 namenode 的 rollEditLog 方法，开始一次新的检查点过程。调用会返回一个 CheckpointSignature（检查点签名），在上传合并完的 FSImage 时，会使用这个签名。

Namenode 的 rollEditLog 方法最终调用的是 FSImage 的同名方法，前面提到过这个方法，作用是关闭往 edits 上写日志，打开日志到 edits.new。明显，在 Secondary NameNode 下载 fsimage 和日志的时候，对命名空间的修改，将保持在 edits.new 的日志中。

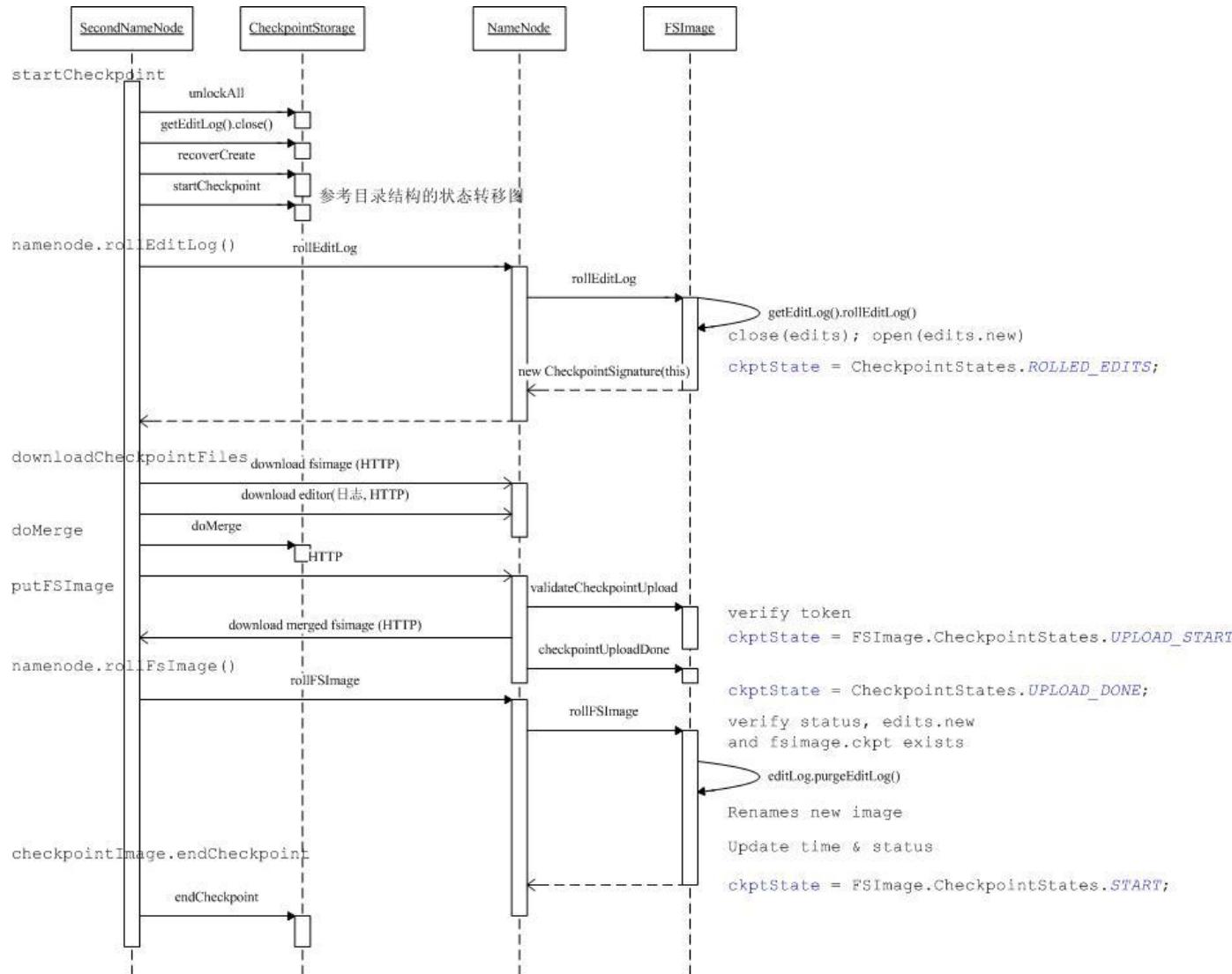
注意，如果 FSImage 这时候的状态（看下面的状态机，前面出现过一次）不是出于 CheckpointStates.ROLLED\_EDITS，将抛异常结束这个过程。

- 3：通过 downloadCheckpointFiles 下载 fsimage 和日志，并设置本地检查点状态为 CheckpointStates.UPLOAD\_DONE。
- 4：合并日志的内容到 fsimage 中。过程很简单，CheckpointStorage 利用继承自 FSImage 的 loadFSImage 加载 fsimage，loadFSEdits 应用日志，然后通过 saveFSImage 保存。很明显，现在保存在硬盘上的 fsimage 是合并日志的内容以后的文件。
- 5：使用 putFSImage 上传合并日志后的 fsimage（让 NameNode 通过 HTTP 到从 NameNode 取文件）。这个过程中，NameNode 会：

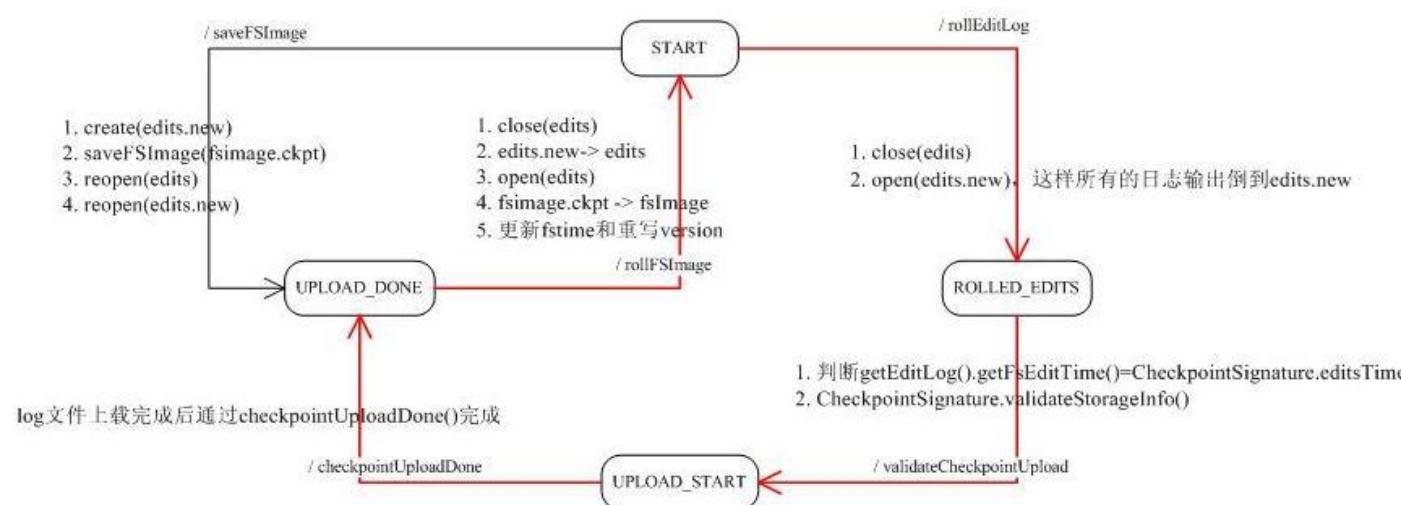
调用 NameNode 的 FSImage.validateCheckpointUpload，检查现在的状态；  
利用 HTTP，从 Secondary NameNode 获取新的 fsimage；  
更新结束后设置新状态。

- 6：调用 NameNode 的 rollFsImage，最终调用 FSImage 的 rollFsImage 方法，前面我们已经分析过了。
- 7：调用本地 endCheckpoint 方法，结束一次 doCheckpoint 流程。

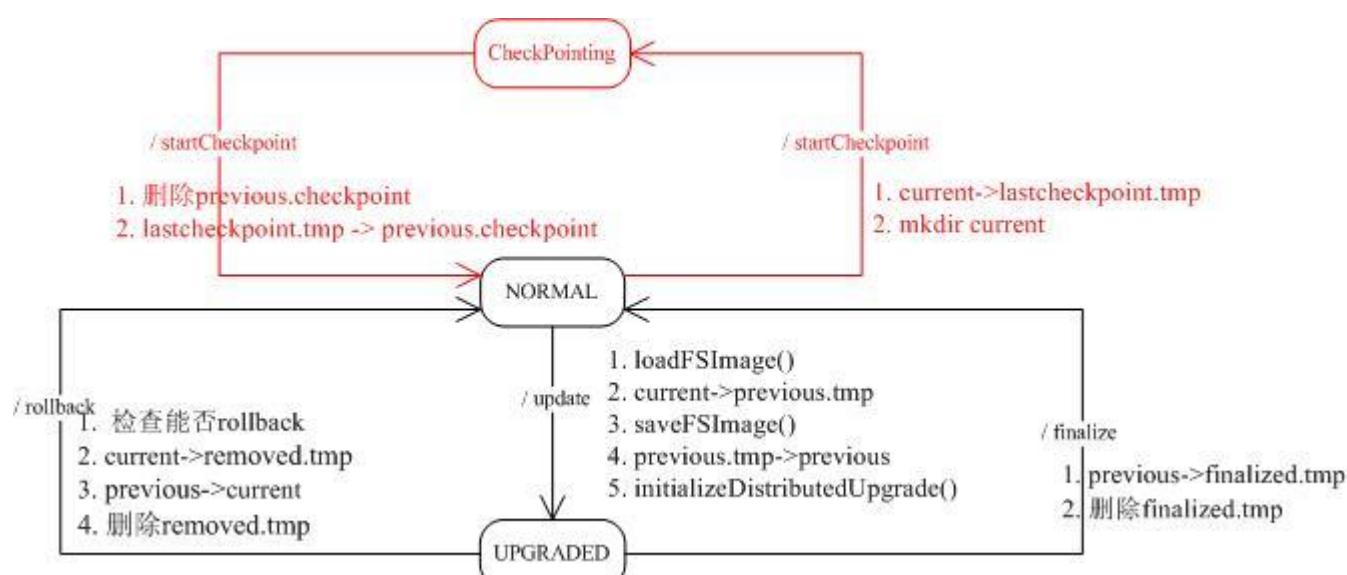
其实前面在分析 FSImage 的时候，我们在不了解 Secondary NameNode 的情况下，分析了很多和 Checkpoint 相关的方法，现在我们终于可以有一个比较统一的了解了，下面给出 NameNode 和 Secondary NameNode 的存储系统在这个流程中的状态转移图，方便大家理解。



图中右侧的状态转移图：



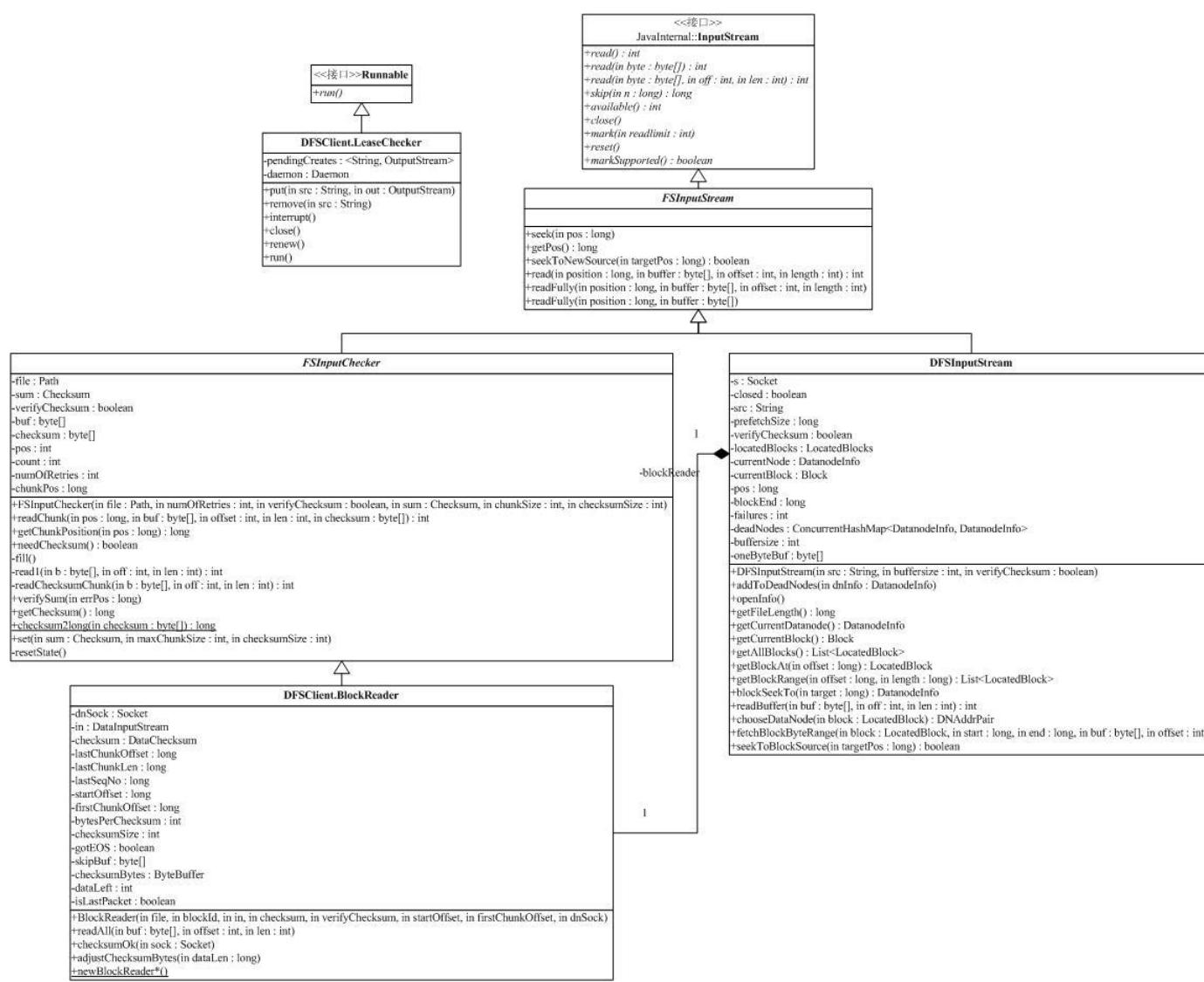
文件系统上的目录的变化（三六中出现）：



## Hadoop 源代码分析 (三八)

我们可以开始从系统的外部来了解 HDFS 了，DFSClient 提供了连接到 HDFS 系统并执行文件操作的基本功能。DFSClient 也是个大家伙，我们先分析它的一些内部类。我们先看 LeaseChecker。租约是客户端对文件写操作时需要获取的一个凭证，前面分析 NameNode 时，已经了解了租约，INodeFileUnderConstruction 的关系，INodeFileUnderConstruction 只有在文件写的时候存在。客户端的租约管理很简单，包括了增加的 put 和删除的 remove 方法，run 方法会定期执行，并通过 ClientProtocol 的 renewLease，自动延长租约。

接下来我们来分析内部为文件读引入的类。



InputStream 是系统的虚类，提供了 3 个 read 方法，一个 skip ( 跳过数据 ) 方法，一个 available 方法 ( 目前流中可读的字节数 )，一个 close 方法和几个在输入流中做标记的方法 ( mark : 标记，reset : 回到标记点和 markSupported : 能力查询 )。

FSInputStream 也是一个虚类，它将接口 Seekable 和 PositionedReadable 混插到类中。Seekable 提供了可以在流中定位的能力 ( seek , newPos 和 seekToNewSource )，而 PositionedReadable 提高了从某个位置开始读的方法 ( 一个 read 方法和两个 readFully 方法 )。

FSInputChecker 在 FSInputStream 的基础上，加入了 HDFS 中需要的校验功能。校验在 readChecksumChunk 中实现，并在内部的 read1 方法中调用。所有的 read 调用，最终都是使用 read1 读数据并做校验。如果校验出错，抛出异常 ChecksumException。

有了支持校验功能的输入流，就可以开始构建基于 Block 的输入流了。我们先回顾前面提到的读数据块的请求协议：

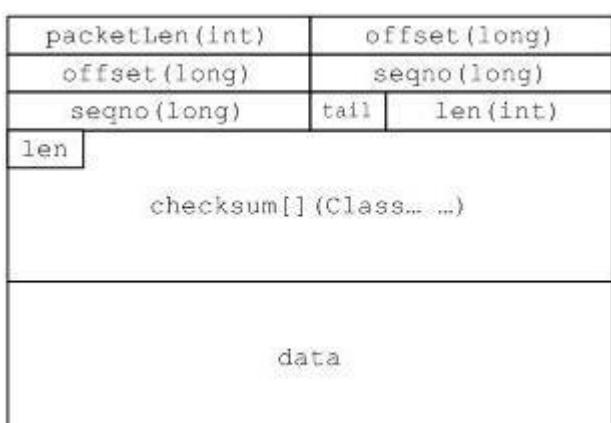
version	81	blockId(long)
blockId		generationStamp(long)
generatio...		startOffset(long)
startOffset		length(long)
length		clientName(String)
clientName	(String... ...)	

然后我们来分析一下创建 BlockReader 需要的参数，newBlockReader 最复杂的请求如下：

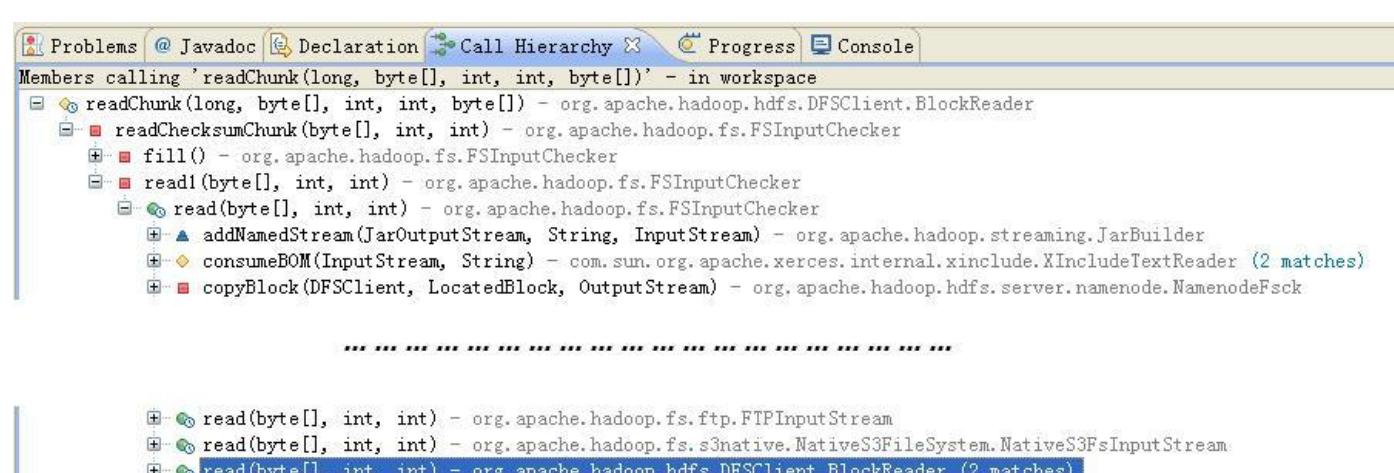
```
public static BlockReader newBlockReader( Socket sock, String file,
                                         long blockId,
                                         long genStamp,
                                         long startOffset, long len,
                                         int bufferSize, boolean verifyChecksum,
                                         String clientName)
                                         throws IOException
```

其中，sock 为到 DataNode 的 socket 连接，file 是文件名（只是用于日志输出），其它的参数含义都很清楚，和协议基本是一一对应的。该方法会和 DataNode 进行对话，发送上面的读数据块的请求，处理应答并构造 BlockReader 对象（BlockReader 的构造函数基本上只有赋值操作）。

BlockReader 的 readChunk 用于处理 DataNode 送过来的数据，格式前面我们已经讨论过了，如下图。



读数据用的 read，会调用父类 FSInputChecker 的 read，最后调用 readChunk，如下：



read 如果发现读到正确的校验码，则用过 checksumOk 方法，向 DataNode 发送成功应答。

BlockReader 的主要流程就介绍完了，接下来分析 DFSInputStream，它封装了 DFSClient 读文件内容的功能。在它的内部，不但要处理和 NameNode 的通信，同时通过 BlockReader，处理和 DataNode 的交互。

DFSInputStream 记录 Block 的成员变量是：

```
private LocatedBlocks locatedBlocks = null;
```

它不但保持了文件对应的 Block 序列，还保持了管理 Block 的 DataNode 的信息，是 DFSInputStream 中最重要的成员变量。

DFSInputStream 的构造函数，通过类内部的 openInfo 方法，获取这个变量的值。openInfo 间接调用了 NameNode 的 getBlockLocations，获取 LocatedBlocks。

DFSInputStream 中处理数据块位置的还有下面一些函数：

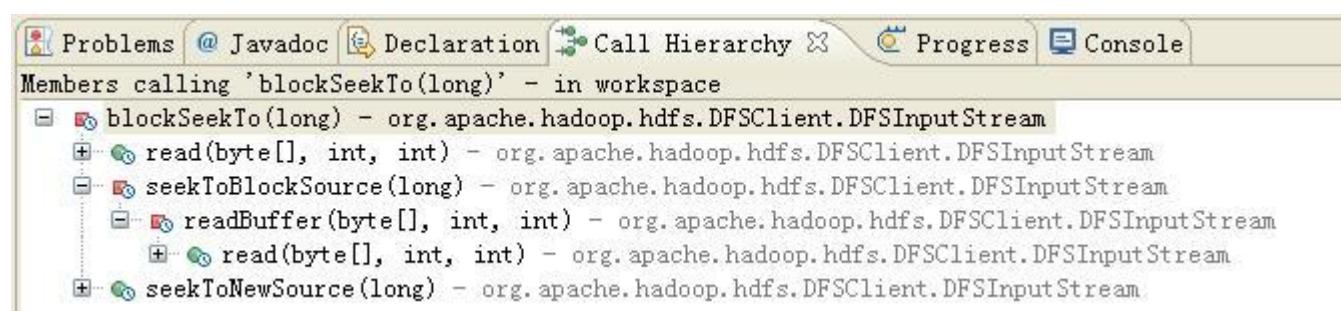
```
synchronized List<LocatedBlock> getAllBlocks() throws IOException
```

```
private LocatedBlock getBlockAt(long offset) throws IOException
```

```
private synchronized List<LocatedBlock> getBlockRange(long offset,  
                                                 long length)
```

```
private synchronized DatanodeInfo blockSeekTo(long target) throws IOException
```

它们的功能都很清楚，需要注意的是他们处理过程中可能会调用再次调用 NameNode 的 getBlockLocations，使得流程比较复杂。blockSeekTo 还会创建对应的 BlockReader 对象，它被几个重要的方法调用（如下图）。在打开到 DataNode 之前，blockSeekTo 会调用 chooseDataNode，选择一个现在活着的 DataNode。



通过上面的分析，我们已经知道了在什么时候会连接 NameNode，什么时候会打开到 DataNode 的连接。下面我们来看读数据。read 方法定义如下：

```
public int read(long position, byte[] buffer, int offset, int length)
```

该方法会从流的 position 位置开始，读取最多 length 个 byte 到 buffer 中 offset 开始的空间中。参数检测完以后，通过 getBlockRange 获取要读取的数据块对应的 block 范围，然后，利用 fetchBlockByteRange 方法，读取需要的数据。

fetchBlockByteRange 从某一个数据块中读取一段数据，定义如下：

```
private void fetchBlockByteRange(LocatedBlock block, long start,  
                                long end, byte[] buf, int offset)
```

由于读取的内容都在一个数据块内部，这个方法会创建 BlockReader，然后利用 BlockReader 的 readAll 方法，读取数据。读的过程中如果发生校验错，那么，还会通过 reportBadBlocks，向 NameNode 报告校验错。

另一个读方法是：

```
public synchronized int read(byte buf[], int off, int len) throws IOException
```

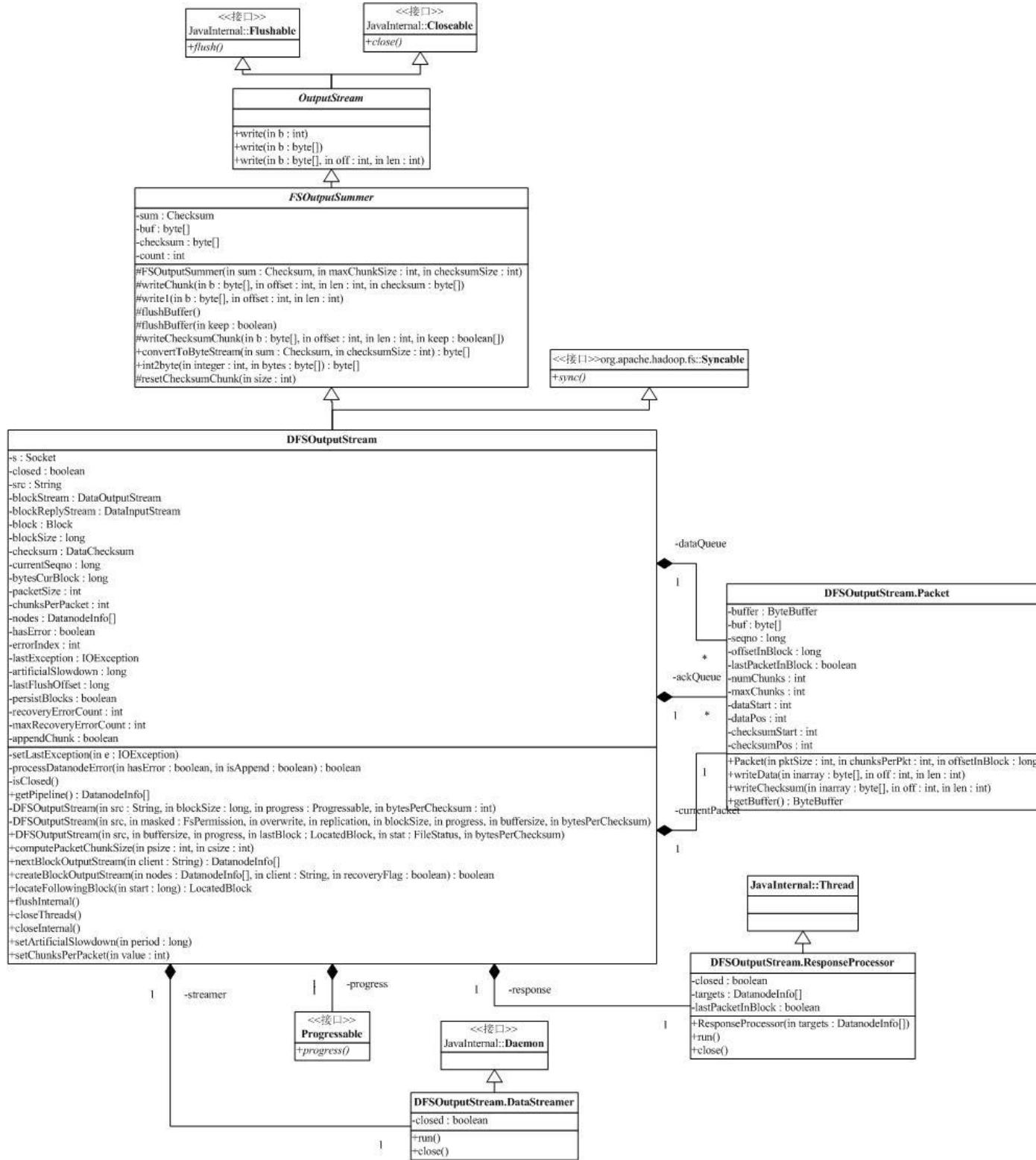
它在流的当前位置( 可以通过 seek 方法调整 )读取数据。首先它会判断当前流的位置，如果已经越过了对象现在的 blockReader 能读取的范围（当上次 read 读到数据块的尾部时，会发生这种情况），那么通过 blockSeekTo 打开到下一个数据块的 blockReader。然后，read 在当前的这个数据块中通过 readBuffer 读数据。主要，这个 read 方法只在一块数据块中读取数据，就是说，如果还有空间可以存放数据但已经到了数据块的尾部，它不会打开到下一个数据块的 BlockReader 继续读，而是返回，返回值包含了以读取数据的长度。

DFSDataInputStream 是一个 Wrapper(DFSInputStream)，我们就不再讨论了。

## Hadoop 源代码分析（三九）

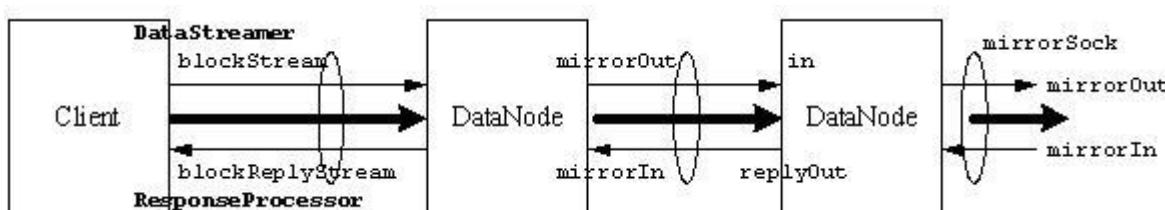
接下来当然是分析输出流了。

处于继承体系的最上方是 OutputStream，它实现了 Closeable ( 方法 close ) 和 Flushable ( 方法 flush ) 接口，提供了 3 个不同形式的 write 方法，这些方法的含义都很清楚。接下来的是 FSOOutputSummer，它引入了 HDFS 写数据时需要的计算校验和的功能。FSOutputSummer 的 write 方法会调用 write1，write1 中计算校验和并将用户输入的数据拷贝到对象的缓冲区中，缓冲区满了以后会调用 flushBuffer，flushBuffer 最终调用还是虚方法的 writeChunk，这个时候，缓冲区对应的校验和缓冲区对的内容都已经准备好了。通过这个类，HDFS 可以把一个流转换成为 DataNode 数据接口上的包格式（前面我们讨论过这个包的格式，如下）。



`DFSOutputStream` 继承自 `FSOutputSummer`，是一个非常复杂的类，它包含了几何内部类。我们先分析 `Packet`，其实它对应了上面的数据包，有了上面的图，这个类就很好理解了，它的成员变量和上面数据块包含的信息基本一一对应。构造函数需要的参数有 `pktSize`，包的大小，`chunksPerPkt`，chunk 的数目（chunk 是一个校验单元）和该包在 Block 中的偏移量 `offsetInBlock`。`writeData` 和 `writeChecksum` 用于往缓冲区里写数据/校验和。`getBuffer` 用户获得整个包，包括包头和数据。

`DataStreamer` 和 `ResponseProcessor` 用于写包/读应答，和我们前面讨论 `DataNode` 的 Pipe 写时类似，客户端写数据也需要两个线程，下图扩展了我们在讨论 `DataNode` 处理写时的示意图，包含了客户端：



DataStreamer 启动后进入一个循环，在没有错误和关闭标记为 false 的情况下，该循环首先调用 processDatanodeError，处理可能的 IO 错误，这个过程比较复杂，我们在后面再讨论。

接着 DataStreamer 会在 dataQueue（数据队列）上等待，直到有数据出现在队列上。DataStreamer 获取一个数据包，然后判断到 DataNode 的连接是否是打开的，如果不是，通过 DFSOutputStream.nextBlockOutputStream 打开到 DataNode 的连接，并启动 ResponseProcessor 线程。

DataNode 的连接准备好以后，DataStreamer 获取数据包缓冲区，然后将数据包从 dataQueue 队列挪到 ackQueue 队列，最后通过 blockStream，写数据。如果数据包是最后一个，那么，DataStreamer 将会写一个长度域为 0 的包，指示 DataNode 数据传输结束。

DataStreamer 的循环在最后一个数据包写出去以后，会等待直到 ackQueue 队列为空（表明所有的应答已经被接收），然后做清理动作（包括关闭 socket 连接，ResponseProcessor 线程等），退出线程。

ResponseProcessor 相对来说比较简单，就是等待来自 DataNode 的应答。如果是成功的应答，则删除在 ackQueue 的包，如果有错误，那么，记录出错的 DataNode，并设置标志位。

## [Hadoop 源代码分析（四零）](#)

有了上面的基础，我们可以来解剖 DFSOutputStream 了。先看构造函数：

```
private DFSOutputStream(String src, long blockSize, Progressable progress,  
    int bytesPerChecksum) throws IOException
```

```
DFSOutputStream(String src, FsPermission masked, boolean overwrite,  
    short replication, long blockSize, Progressable progress,  
    int buffersize, int bytesPerChecksum) throws IOException
```

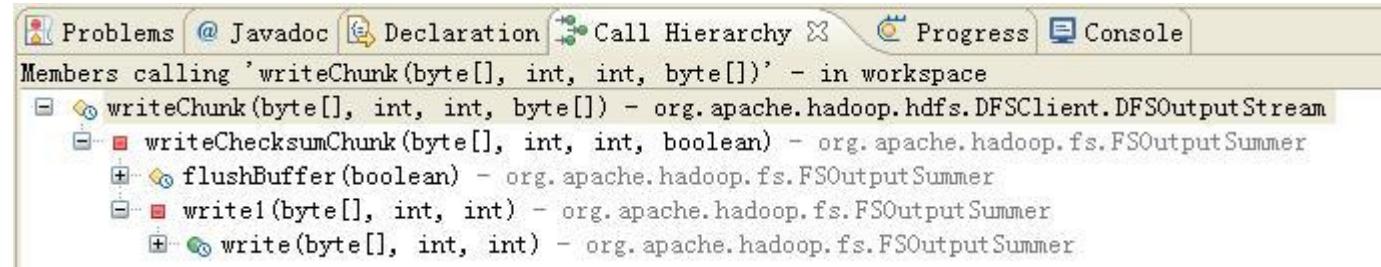
```
DFSOutputStream(String src, int buffersize, Progressable progress,  
    LocatedBlock lastBlock, FileStatus stat,  
    int bytesPerChecksum) throws IOException {
```

这些构造函数的参数主要有：文件名 src；进度回调函数 progress（预留接口，目前未使用）；数据块大小 blockSize；Block 副本数 replication；每个校验 chunk 的大小 bytesPerChecksum；文件权限 masked；是否覆盖原文件标记 overwrite；文件状态信息 stat；文件的最后一个 Block 信息 lastBlock；buffersize（？未见引用）。

后面两个构造函数会调用第一个构造函数，这个函数会调用父类的构造函数，并设置对象的 src，blockSize，progress 和 checksum 属性。

第二个构造函数会调用 namenode.create 方法，在文件空间中建立文件，并启动 DataStreamer，它被 DFSClient 的 create 方法调用。第三个构造函数被 DFSClient 的 append 方法调用，显然，这种情况比价复杂，文件拥有一些数据块，添加数据往往添加在最后的数据块上。同时，append 方法调用时，Client 已经知道了最后一个 Block 的信息和文件的一些信息，如 FileStatus 中包含的 Block 大小，文件权限位等等。结合这些信息，构造函数需要计算并设置一些对象成员变量的值，并试图从可能的错误中恢复（调用 processDatanodeError），最后启动 DataStreamer。

我们先看正常流程，前面已经分析过，通过 FSOutputSummer，HDFS 客户端能将流转换成 package，这个包是通过 writeChunk，发送出去的，下面是它们的调用关系。



在检查完一系列的状态以后，writeChunk 先等待，直到 dataQueue 中未发送的包小于门限值。如果现在没有可用的 Packet 对象，则创建一个 Packet 对象，往 Packet 中写数据，包括校验值和数据。如果数据包被写满，那么，将它放入发送队列 dataQueue 中。writeChunk 的过程比较简单，这里的写入，也只是把数据写到本地队列，等待 DataStreamer 发送，没有实际写到 DataNode 上。

createBlockOutputStream 用于建立到第一个 DataNode 的连接，它的声明如下：

```
private boolean createBlockOutputStream(DatanodeInfo[] nodes, String client,
                                         boolean recoveryFlag)
```

nodes 是所有接收数据的 DataNode 列表，client 就是客户端名称，recoveryFlag 指示是否是为错误恢复建立的连接。createBlockOutputStream 很简单，打开到第一个 DataNode 的连接，然后发送下面格式的数据包，并等待来自 DataNode 的 Ack。如果出错，记录出错的 DataNode 在 nodes 中的位置，设置 errorIndex 并返回 false。

version	80	blockId(long)
blockId		generationStamp(long)
generatio...	pipelineSize(int)	isRecovery
client(String... ...)		
hasSrcDataNode	srcDataNode(Class<?>, optional... ...)	
numTargets(int)		targets[1]
	targets[1] (Class<?> ...)	
	targets[...]	
	checksum.header(Class<?> ...)	

当 recoveryFlag 指示为真时，意味着这次写是一次恢复操作，对于 DataNode 来说，这意味着为写准备的临时文件（在 tmp 目录中）可能已经存在，需要进行一些特殊处理，具体请看 FSDataset 的实现。

当 Client 写数据需要一个新的 Block 的时候，可以调用 nextBlockOutputStream 方法。

```
private DatanodeInfo[] nextBlockOutputStream(String client) throws IOException
```

这个方法的实现很简单，首先调用 locateFollowingBlock（包含了重试和出错处理），通过 namenode.addBlock 获取一个新的数据块，返回的是 DatanodeInfo 列表，有了这个列表，就可以建立写数据的 pipe 了。下一个大动作就是调用上面的 createBlockOutputStream，建立到 DataNode 的连接了。

有了上面的准备，我们来分析 processDataNodeError，它的主要流程是：

参数检查；

关闭可能还打开着的 blockStream 和 blockReplyStream；

将未收到应答的数据块（在 ackQueue 中）挪到 dataQueue 中；

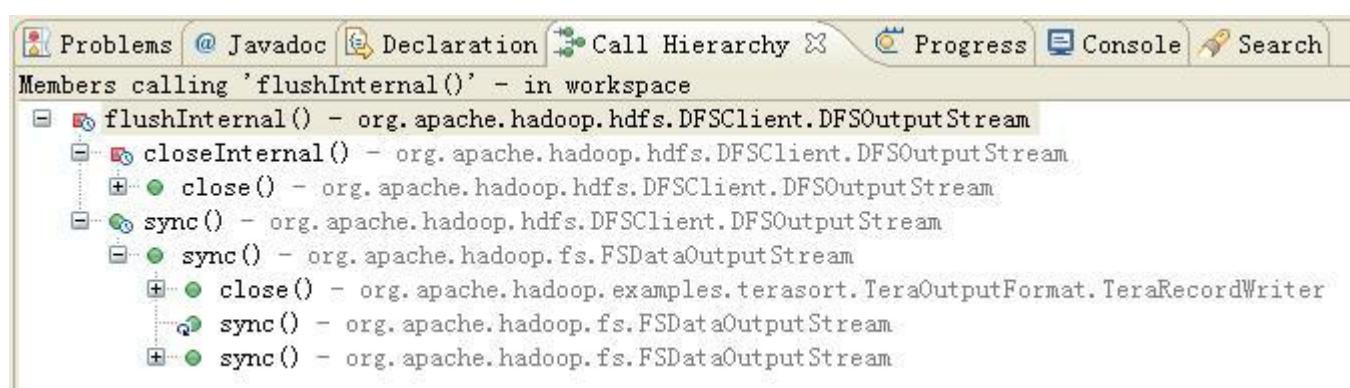
循环执行：

1. 计算目前还活着的 DataNode 列表；
2. 选择一个主 DataNode，通过 DataNode RPC 的 recoverBlock 方法启动它上面的恢复过程；
3. 处理可能的出错；
4. 处理恢复后 Block 可能的变化（如 Stamp 变化）；
5. 调用 createBlockOutputStream 到 DataNode 的连接。

启动 ResponseProcessor。

这个过程涉及了 DataNode 上的 recoverBlock 方法和 createBlockOutputStream 中可能的 Block 恢复，是一个相当耗资源的方法，当系统出错的概率比较小，而且数据块上能恢复的数据很多（平均 32M），还是值得这样做的。

写的流程就分析到着，接下来我们来看流的关闭，这个过程也涉及了一系列的方法，它们的调用关系如下：



flushInternal 会一直等待到发送队列（包括可能的 currentPacket）和应答队列都为空，这意味着数据都被 DataNode 顺利接收。

sync 作用和 UNIX 的 sync 类似，将写入数据持久化。它首先调用父类的 flushBuffer 方法，将可能还没拷贝到 DFSOutputStream 的数据拷贝回来，然后调用 flushInternal，等待所有的数据都写完。然后调用 namenode.fsync，持久化命名空间上的数据。

closeInternal 比较复杂一点，它首先调用父类的 flushBuffer 方法，将可能还没拷贝到 DFSOutputStream 的数据拷贝回来，然后调用 flushInternal，等待所有的数据都写完。接着结束两个工作线程，关闭 socket，最后调用 amenode.complete，通知 NameNode 结束一次写操作。close 方法先调用 closeInternal，然后再本地的 leasechecker 中移除对应的信息。

## Hadoop 源代码分析（四一）

前面分析的 DFSClient 内部类，占据了这个类的实现部分的 2/3，我们来看剩下部分。

DFSClient 的成员变量不多，而且大部分是系统的缺省配置参数，其中比较重要的是到 NameNode 的 RPC 客户端：

```
public final ClientProtocol namenode;  
final private ClientProtocol rpcNamenode;
```

它们的差别是 namenode 在 rpcNamenode 的基础上，增加了失败重试功能。DFSClient 中提供可各种构造它们的 static 函数，createClientDatanodeProtocolProxy 用于生成到 DataNode 的 RPC 客户端。

DFSClient 的构造函数也比价简单，就是初始化成员变量，close 用于关闭 DFSClient。

下面的功能，DFSClient 只是简单地调用 NameNode 的对应方法（加一些简单的检查），就不罗嗦了：

```
setReplication/rename/delete/exists (通过 getFileInfo 的返回值是否为空判断)  
/listPaths/getFileInfo/setPermission/setOwner/getDiskStatus/totalRawCapacity/totalRawUsed/datanodeReport/set  
SafeMode/refreshNodes/metaSave/finalizeUpgrade/mkdirs/getContentSummary/setQuota/setTimes
```

DFSClient 提供了各种 create 方法，它们最后都是构造一个 OutputStream，并将文件名和生成的 OutputStream 加到 leasechecker，完成创建动作。

append 操作是通过 namenode.append，获取最后的 Block 信息，然后构造一个 OutputStream，并将文件名和生成的 OutputStream 加到 leasechecker，完成 append 动作。

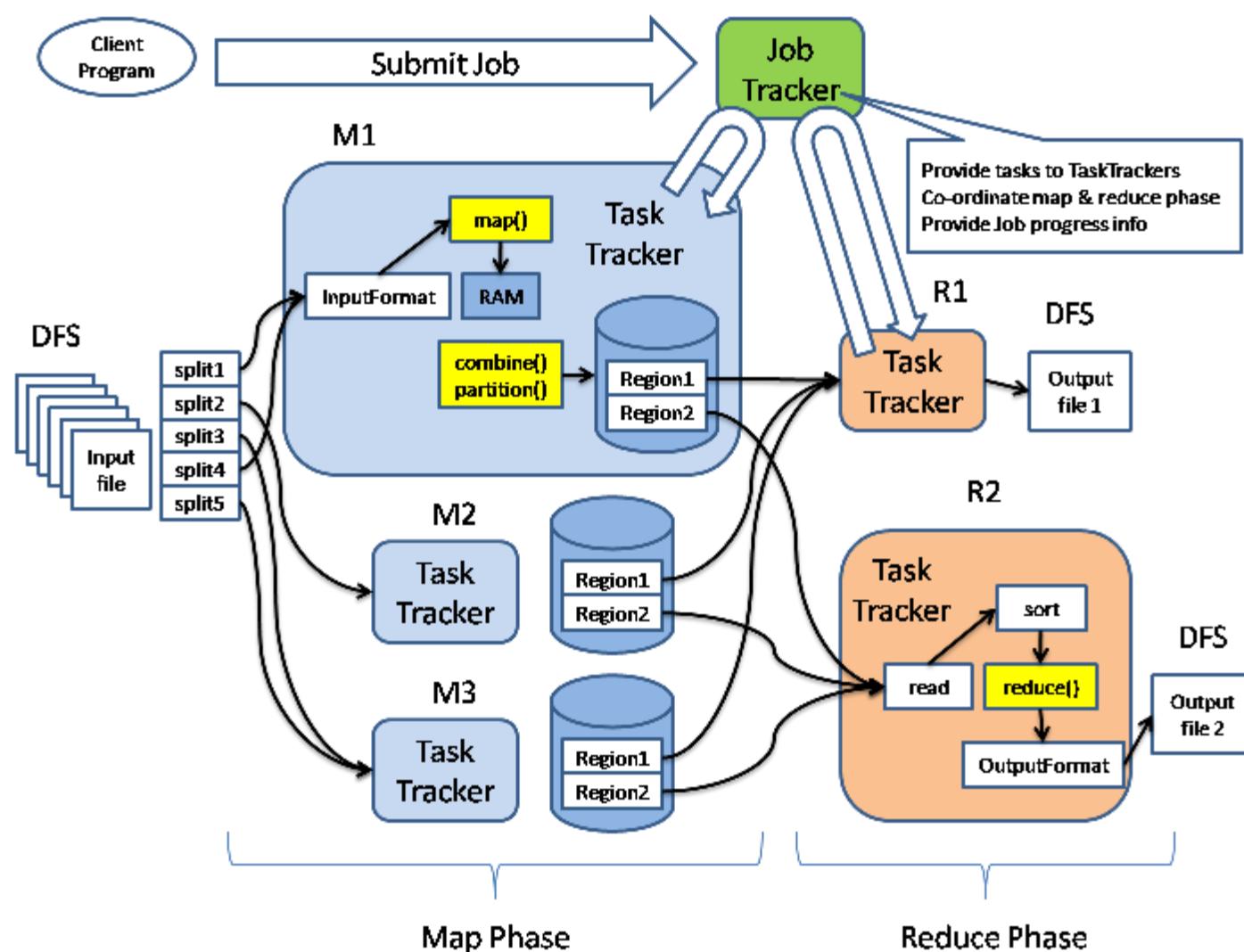
getFileChecksum 用于获取文件的校验信息，它在得到数据块的位置信息后利用 DataNode 提供的 OP\_BLOCK\_CHECKSUM 操作，获取需要的数据，并综合起来。过程简单，方法主要是在处理 OP\_BLOCK\_CHECKSUM 需要交换的数据包。

DFSClient 内部还有一些其它的辅助方法，都比较简单，就不再分析了。

## Hadoop 源代码分析（MapReduce 概论）

大家都熟悉文件系统，在对 HDFS 进行分析前，我们并没有花很多的时间去介绍 HDFS 的背景，毕竟大家对文件系统的还是有一定的理解的，而且也有很好的文档。在分析 Hadoop 的 MapReduce 部分前，我们还是先了解系统是如何工作的，然后再

进入我们的分析部分。下面的图来自 <http://horicky.blogspot.com/2008/11/hadoop-mapreduce-implementation.html>，是我看到的讲 MapReduce 最好的图。



以 Hadoop 带的 wordcount 为例子（下面是启动行）：

```
hadoop jar hadoop-0.19.0-examples.jar wordcount /usr/input /usr/output
```

用户提交一个任务以后，该任务由 JobTracker 协调，先执行 Map 阶段（图中 M1，M2 和 M3），然后执行 Reduce 阶段（图中 R1 和 R2）。Map 阶段和 Reduce 阶段动作都受 TaskTracker 监控，并运行在独立于 TaskTracker 的 Java 虚拟机中。

我们的输入和输出都是 HDFS 上的目录（如上图所示）。输入由 InputFormat 接口描述，它的实现如 ASCII 文件，JDBC 数据库等，分别处理对于的数据源，并提供了数据的一些特征。通过 InputFormat 实现，可以获取 InputSplit 接口的实现，这个实现用于对数据进行划分（图中的 split1 到 split5，就是划分以后的结果），同时从 InputFormat 也可以获取 RecordReader 接口的实现，并从输入中生成<k,v>对。有了<k,v>，就可以开始做 map 操作了。

map 操作通过 context.collect（最终通过 OutputCollector. collect）将结果写到 context 中。当 Mapper 的输出被收集后，它们会被 Partitioner 类以指定的方式区分地写出到输出文件里。我们可以为 Mapper 提供 Combiner，在 Mapper 输出它的<k,v>时，键值对不会被马上写到输出里，他们会被收集在 list 里（一个 key 值一个 list），当写入一定数量的键值对时，这部分缓冲会被 Combiner 中进行合并，然后再输出到 Partitioner 中（图中 M1 的黄颜色部分对应着 Combiner 和 Partitioner）。

Map 的动作做完以后，进入 Reduce 阶段。这个阶段分 3 个步骤：混洗（Shuffle），排序（sort）和 reduce。

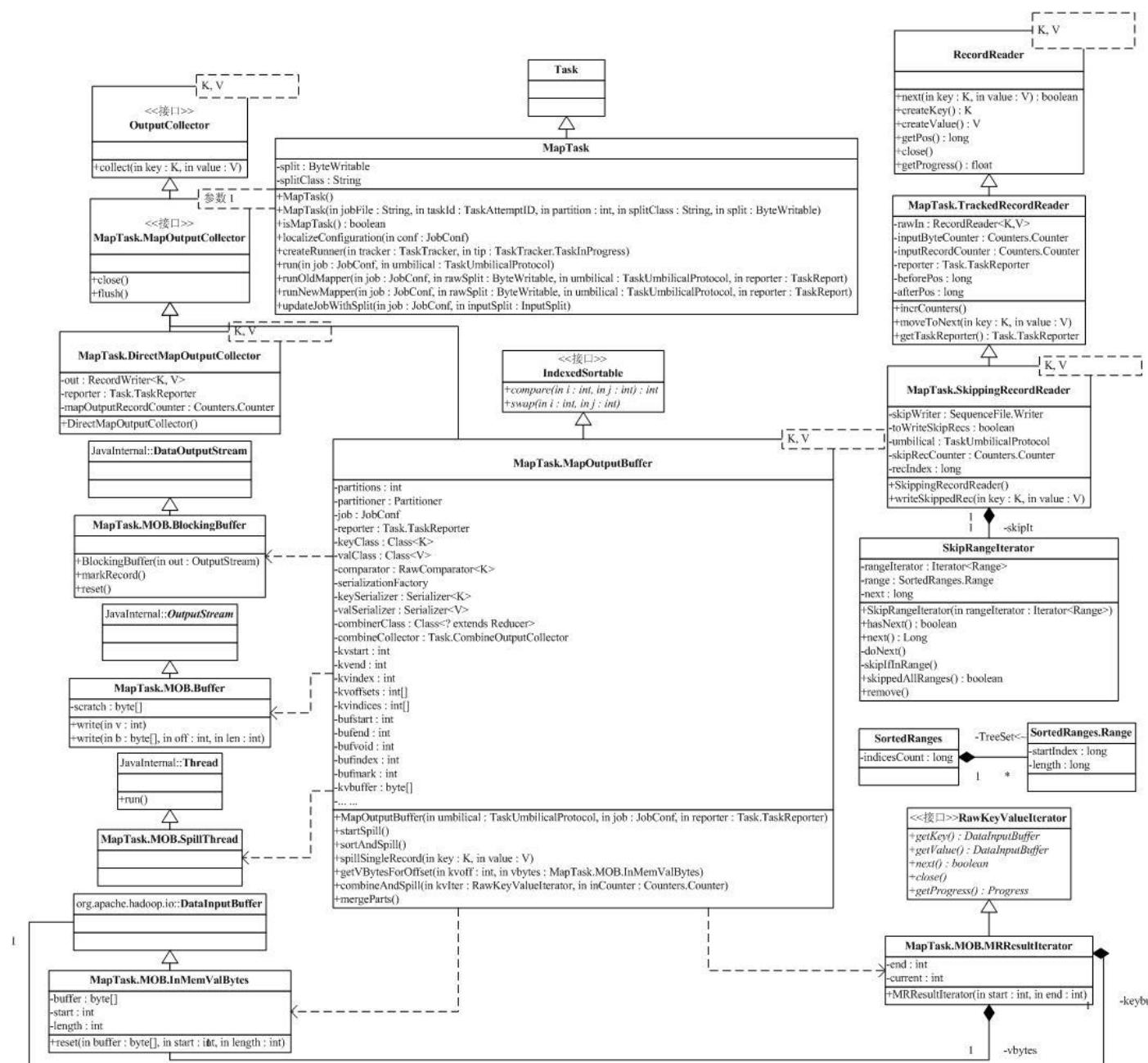
混洗阶段，Hadoop 的 MapReduce 框架会根据 Map 结果中的 key，将相关的结果传输到某一个 Reducer 上（多个 Mapper 产生的同一个 key 的中间结果分布在不同的机器上，这一步结束后，他们传输都到了处理这个 key 的 Reducer 的机器上）。这个步骤中的文件传输使用了 HTTP 协议。

排序和混洗是一块进行的，这个阶段将来自不同 Mapper 具有相同 key 值的<key,value>对合并到一起。

Reduce 阶段，上面通过 Shuffle 和 sort 后得到的<key, (list of values)>会送到 Reducer. reduce 方法中处理，输出的结果通过 OutputFormat，输出到 DFS 中。

## Hadoop 源代码分析 ( MapTask )

接下来我们来分析 Task 的两个子类，MapTask 和 ReduceTask。MapTask 的相关类图如下：



MapTask 其实不是很复杂，复杂的是支持 MapTask 工作的一些辅助类。MapTask 的成员变量少，只有 split 和 splitClass。我们知道，Map 的输入是 split，是原始数据的一个切分，这个切分由 org.apache.hadoop.mapred.InputSplit 的子类具体描述（前面我们是通过 org.apache.hadoop.mapreduce.InputSplit 介绍了 InputSplit，它们对外的 API 是一样的）。splitClass 是 InputSplit 子类的类名，通过它，我们可以利用 Java 的反射机制，创建出 InputSplit 子类。而 split 是一个 BytesWritable，它是 InputSplit 子类串行化以后的结果，再通过 InputSplit 子类的 readFields 方法，我们可以回复出对应的 InputSplit 对象。

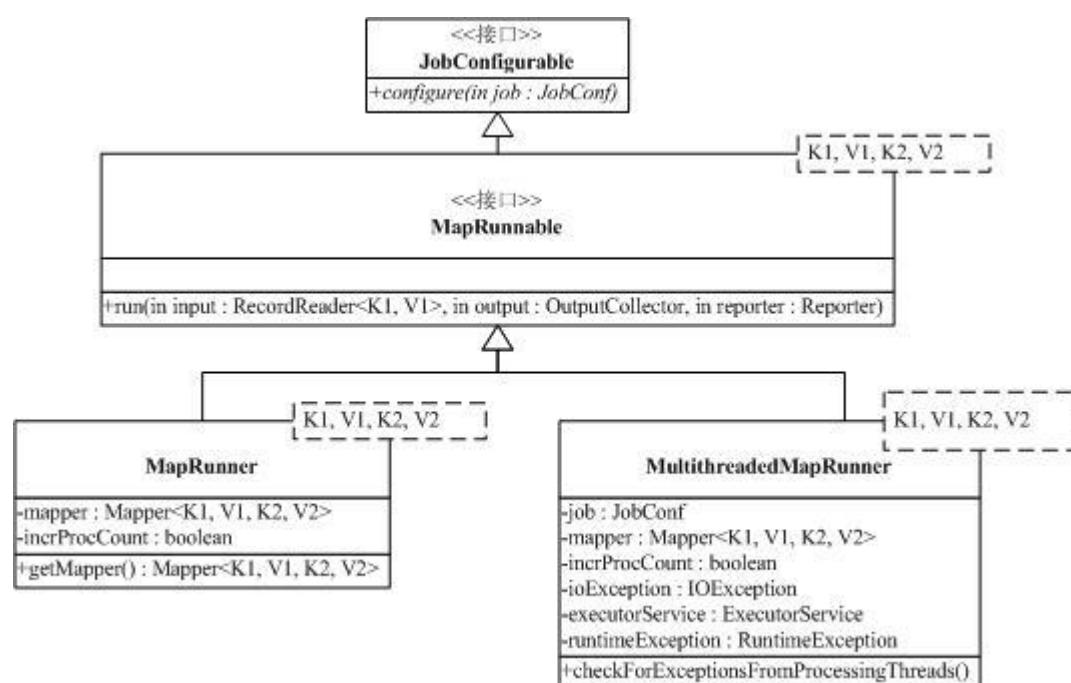
MapTask 最重要的方法是 run。 run 方法相当简单，配置完系统的 TaskReporter 后，就根据情况执行 runJobCleanupTask , runJobSetupTask , runTaskCleanupTask 或执行 Mapper。由于 MapReduce 现在有两套 API , MapTask 需要支持这两套 API ,使得 MapTask 执行 Mapper 分为 runNewMapper 和 runOldMapper ,run\*Mapper 后 ,MapTask 会调用父类的 done 方法。

接下来我们来分析 runOldMapper ,最开始部分是构造 Mapper 处理的 InputSplit ,更新 Task 的配置 然后就开始创建 Mapper 的 RecordReader , rawIn 是原始输入 ,然后分正常 ( 使用 TrackedRecordReader ,后面讨论 ) 和跳过部分记录 ( 使用 SkippingRecordReader ,后面讨论 ) 两种情况 ,构造对应的真正输入 in。

跳过部分记录是 Map 的一种出错恢复策略 ,我们知道 , MapReduce 处理的数据集合非常大 ,而有些任务对一部分出错的数据不进行处理 ,对结果的影响很小 ( 如大数据集合的一些统计量 ) ,那么 ,一小部分的数据出错导致已处理的大量结果无效 ,是得不偿失的 ,跳过这部分记录 ,成了 Mapper 的一种选择。

Mapper 的输出 ,是通过 MapOutputCollector 进行的 ,也分两种情况 ,如果没有 Reducer ,那么 ,用 DirectMapOutputCollector ( 后面讨论 ) ,否则 ,用 MapOutputBuffer ( 后面讨论 ) 。

构造完 Mapper 的输入输出 ,通过构造配置文件中配置的 MapRunnable ,就可以执行 Mapper 了。目前系统有两个 MapRunnable : MapRunner 和 MultithreadedMapRunner ,如下图。



原有 API 在这块的处理上和新 API 有很大的不一样。接口 MapRunnable 是原有 API 中 Mapper 的执行器 , run 方法就是用于执行用户的 Mapper。 MapRunner 是单线程执行器 ,相当简单 ,首先 ,当 MapTask 调用 :

```

MapRunnable<INKEY,INVALUE,OUTKEY,OUTVALUE> runner =
    ReflectionUtils.newInstance(job.getMapRunnerClass(), job);

```

MapRunner 的 configure 会在 newInstance 的最后被调用 ,configure 执行的过程中 ,对应的 Mapper 会通过反射机制构造出来。

MapRunner 的 run 方法，会先创建对应的 key , value 对象，然后，对 InputSplit 的每一对<key , value>，调用 Mapper 的 map 方法，循环结束后，Mapper 对应的清理方法会被调用。我们需要注意，key , value 对象在 run 方法中是被重复使用的，就是说，每次传入 Mapper 的 map 方法的 key , value 都是同一个对象，只不过是里面的内容变了，对象并没有变。如果你需要保留 key , value 的内容，需要实现 clone 机制，克隆出对象的一个新备份。

相对于新 API 的多线程执行器，老 API 的 MultithreadedMapRunner 就比较复杂了，总体来说，就是通过阻塞队列配合 Java 的多线程执行器，将<key , value>分发到多个线程中去处理。需要注意的是，在这个过程中，这些线程共享一个 Mapper 实例，如果 Mapper 有共享的资源，需要有一定的保护机制。

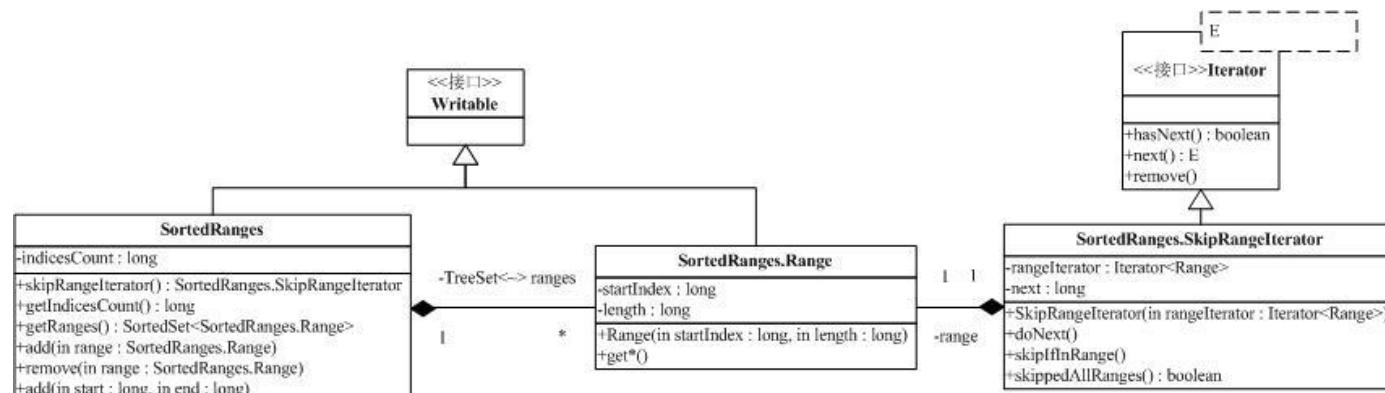
runNewMapper 用于执行新版本的 Mapper，比 runOldMapper 稍微复杂，我们就不再讨论了。

## [Hadoop 源代码分析 \( MapTask 辅助类 I \)](#)

MapTask 的辅助类主要针对 Mapper 的输入和输出。首先我们来看 MapTask 中用的的 Mapper 输入，在类图中，这部分位于右上角。

MapTask.TrackedRecordReader 是一个 Wrapper，在原有输入 RecordReader 的基础上，添加了收集上报统计数据的功能。

MapTask.SkippingRecordReader 也是一个 Wrapper，它在 MapTask.TrackedRecordReader 的基础上，添加了忽略部分输入的功能。在分析 MapTask.SkippingRecordReader 之前，我们先看一下类 SortedRanges 和它相关的类。



类 `SortedRanges.Ranges` 表示了一个范围，以开始位置和范围长度（这样的话就可以表示长度为 0 的范围）来表示一个范围，并提供了一系列的范围操作方法。注意，方法 `getEndIndex` 得到的右端点并不包含在范围内( 应理解为开区间 )。`SortedRanges` 包含了一系列不重叠的范围，为了保证包含的范围不重叠，在 `add` 方法和 `remove` 方法上需要做一些处理，保证不重叠的约束。`SkipRangeIterator` 是访问 `SortedRanges` 包含的 Ranges 的迭代器。

MapTask.SkippingRecordReader 的实现很简单，因为要忽略的输入都保持在 `SortedRanges.Ranges`，只需要在 `next` 方法中，判断目前范围时候落在 `SortedRanges.Ranges` 中，如果是，忽略，并将忽略的记录写文件（可配置）

NewTrackingRecordReader 和 NewOutputCollector 被新 API 使用，我们不分析。

MapTask 的输出辅助类都继承自 `MapOutputCollector`，它只是在 `OutputCollector` 的基础上添加了 `close` 和 `flush` 方法。

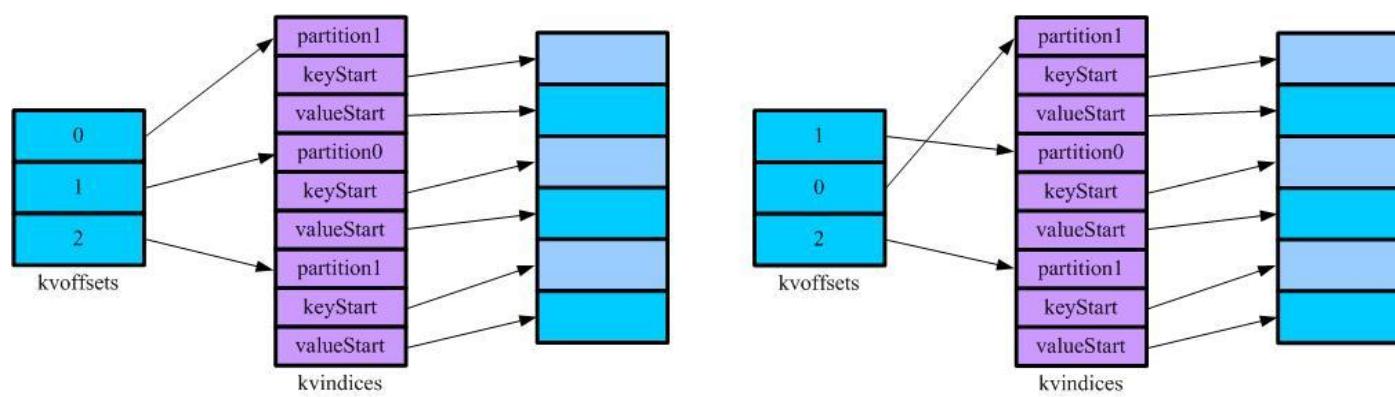
DirectMapOutputCollector 用在 Reducer 的数目为 0，就是不需要 Reduce 阶段的时候。它是直接通过

```
out = job.getOutputFormat().getRecordWriter(fs, job, finalName, reporter);
```

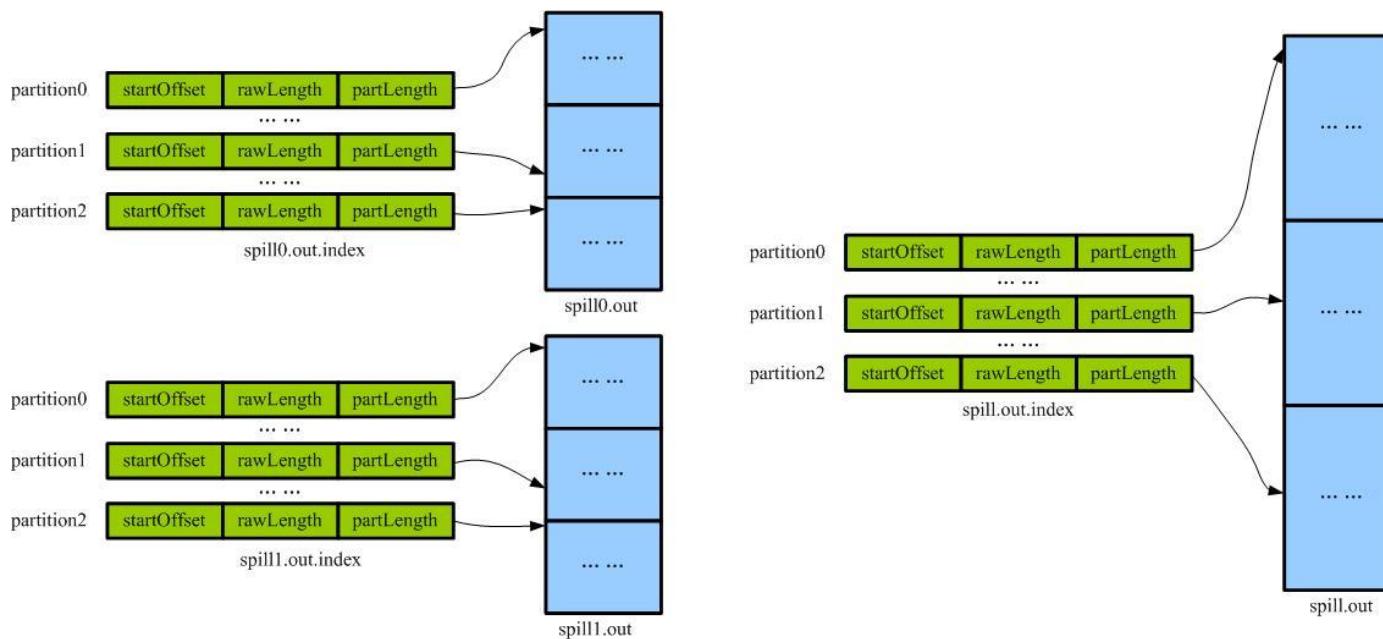
得到对应的 RecordWriter , collect 直接到 RecordWriter 上。

如果 Mapper 后续有 reduce 任务 , 系统会使用 MapOutputBuffer 做为输出 , 这是个比较复杂的类 , 有 1k 行左右的代码。

我们知道 ,Mapper 是通过 OutputCollector 将 Map 的结果输出 , 输出的量很大 ,Hadoop 的机制是通过一个 circle buffer 收集 Mapper 的输出 , 到了  $io.sort.mb * percent$  量的时候 , 就 spill 到 disk , 如下图。图中出现了两个数组和一个缓冲区 ,kvindices 保持了记录所属的 ( Reduce ) 分区 , key 在缓冲区开始的位置和 value 在缓冲区开始的位置 , 通过 kvindices , 我们可以在缓冲区中找到对应的记录。kvoffsets 用于在缓冲区满的时候对 kvindices 的 partition 进行排序 , 排完序的结果将输出到输出到本地磁盘上 , 其中索引 ( kvindices ) 保持在  $spill\{spill\#}.out.index$  中 , 数据保存在  $spill\{spill\#}.out$  中。



当 Mapper 任务结束后 , 有可能会出现多个 spill 文件 , 这些文件会做一个归并排序 , 形成 Mapper 的一个输出 ( spill.out 和 spill.out.index ) , 如下图 :



这个输出是按 partition 排序的 , 这样的话 , Mapper 的输出被分段 , Reducer 要获取的就是 spill.out 中的一段。 ( 注意 , 内存和硬盘上的索引结构不一样 )

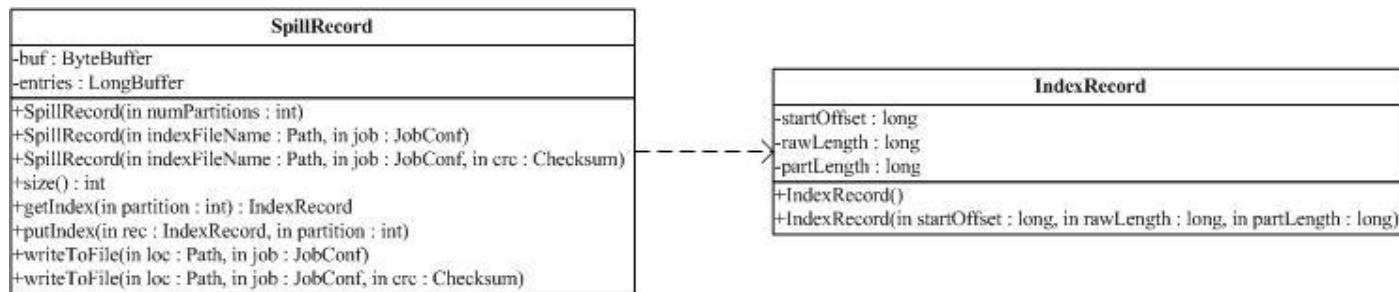
( 感谢彭帅的 Hadoop Map Stage 流程分析 <http://www.cnblogs.com/OnlyXP/archive/2009/05/25/1488811.html> )

## Hadoop 源代码分析 ( MapTask 辅助类 , II )

有了上面 Mapper 输出的内存存储结构和硬盘存储结构讨论，我们来仔细分析 MapOutputBuffer 的流程。

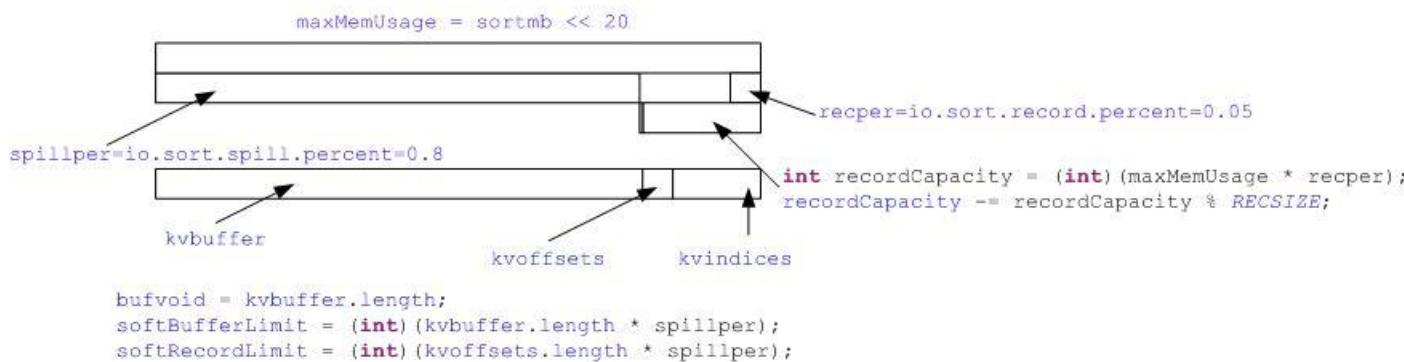
首先是成员变量。最先初始化的是作业配置 job 和统计功能 reporter。通过配置，MapOutputBuffer 可以获取本地文件系统 ( localFs 和 rfs )，Reducer 的数目和 Partitioner。

SpillRecord 是文件 spill.out{spill 号}.index 在内存中的对应抽象（内存数据和文件数据就差最后的校验和），该文件保持了一系列的 IndexRecord，如下图：



IndexRecord 有 3 个字段，分别是 startOffset : 记录偏移量，rawLength : 初始长度，partLength : 实际长度（可能有压缩）。SpillRecord 保持了一系列的 IndexRecord，并提供方法用于添加记录（没有删除记录的操作，因为不需要），获取记录，写文件，读文件（通过构造函数）。

接下来是一些和输出缓存区 kvbuffer，缓存区记录索引 kvindices 和缓存区记录索引排序工作数组 kvoffsets 相关的处理，下面的图有助于说明这段代码。



这部分依赖于 3 个配置参数，io.sort.spill.percent 是 kvbuffer，kvindices 和 kvoffsets 的总大小（以 M 为单位，缺省是 100，就是 100M，这一部分是 MapOutputBuffer 中占用存储最多的）。io.sort.record.percent 是 kvindices 和 kvoffsets 占用的空间比例（缺省是 0.05）。前面的分析我们已经知道 kvindices 和 kvoffsets 如果记录数是 N 的话，它占用的空间是  $4N \times 4\text{bytes}$ ，根据这个关系和 io.sort.record.percent 的值，我们可以计算出 kvindices 和 kvoffsets 最多能有多少个记录，并分配相应的空间。参数 io.sort.spill.percent 指示当输出缓冲区或 kvindices 和 kvoffsets 记录数量到达对应的占用率的时候，会启动 spill，将内存缓冲区的记录存放到硬盘上，softBufferLimit 和 softRecordLimit 为对应的字节数。

值对<key, value>输出到缓冲区是通过 Serializer 串行化的，这部分的初始化跟在上面输出缓存后面。接下来是一些计数器和可能的数据压缩处理器的初始化，可能的 Combiner 和 combiner 工作的一些配置。

最后是启动 spillThread，该 Thread 会检查内存中的输出缓存区，在满足一定条件的时候将缓冲区中的内容 spill 到硬盘上。这是一个标准的生产者-消费者模型，MapTask 的 collect 方法是生产者，spillThread 是消费者，它们之间同步是通过 spillLock (ReentrantLock) 和 spillLock 上的两个条件变量 (spillDone 和 spillReady) 完成的。

先看生产者，MapOutputBuffer.collect 的主要流程是：

报告进度和参数检测 ( $\langle K, V \rangle$  符合 Mapper 的输出约定)；

spillLock.lock()，进入临界区；

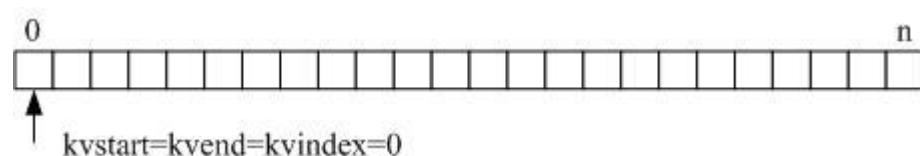
如果达到 spill 条件，设置变量并通过 spillReady.signal()，通知 spillThread；并等待 spill 结束（通过 spillDone.await() 等待）；

spillLock.unlock()；

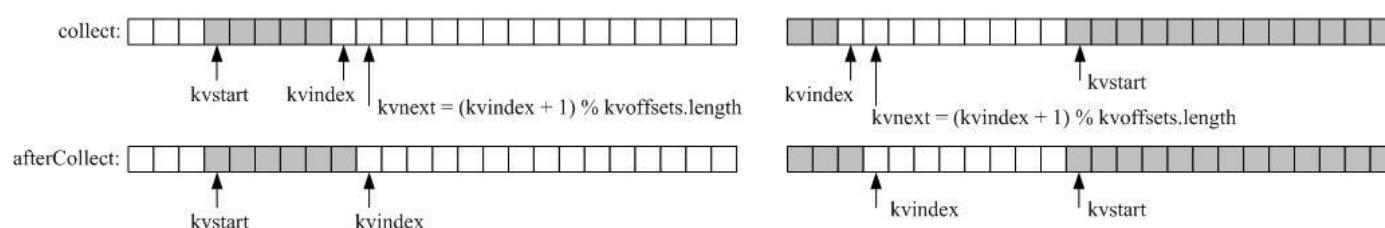
输出 key, value 并更新 kvindices 和 kvoffsets（注意，方法 collect 是 synchronized，key 和 value 各自输出，它们也会占用连续的输出缓冲区）；

kvstart, kvend 和 kvindex 三个变量在判断是否需要 spill 和 spill 是否结束的过程中很重要，kvstart 是有效记录开始的下标，kvindex 是下一个可做记录的位置，kvend 的作用比较特殊，它在一般情况下  $kvstart == kvend$ ，但开始 spill 的时候它会被赋值为 kvindex 的值，spill 结束时，它的值会被赋给 kvstart，这时候  $kvstart == kvend$ 。这就是说，如果  $kvstart \neq kvend$ ，系统正在 spill，否则， $kvstart == kvend$ ，系统处于普通工作状态。其实在代码中，我们可以看到很多  $kvstart == kvend$  的判断。

下面我们分情况，讨论 kvstart, kvend 和 kvindex 的配合。初始化的时候，它们都被赋值 0。

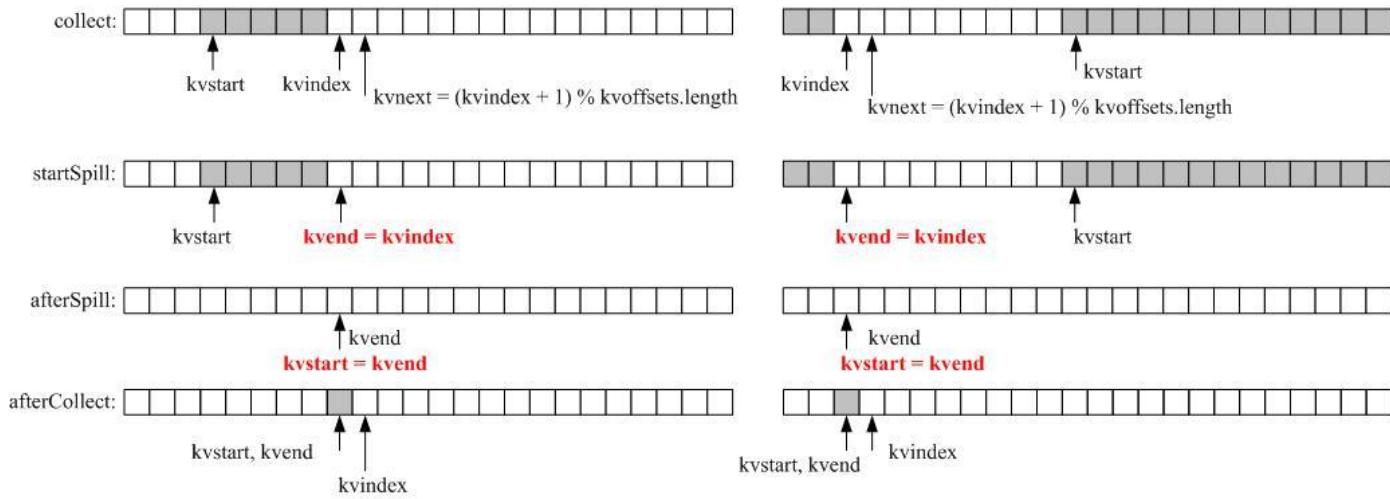


下图给出了一个没有 spill 的记录添加过程：



注意 kvindex 和 kvnext 的关系，取模实现了循环缓冲区

如果在添加记录的过程中，出现 spill（多种条件），那么，主要的过程如下：



首先还是计算 `kvnext`，主要，这个时候 `kvend==kvstart`（图中没有画出来）。如果 spill 条件满足，那么，`kvindex` 的值会赋给 `kvend`（这是 `kvend` 不等于 `kvstart`），从 `kvstart` 和 `kvend` 的大小关系，我们可以知道记录位于数组的那一部分（左边是 `kvstart<kvend` 的情况，右边是另外的情况）。Spill 结束的时候，`kvend` 值会被赋给 `kvstart`，`kvend==kvstart` 又重新满足，同时，我们可以发现 `kvindex` 在这个过程中没有变化，新的记录还是写在 `kvindex` 指向的位置，然后，`kvindex=kvnext`，`kvindex` 移到下一个可用位置。

大家体会一下上面的过程，特别是 `kvstart`，`kvend` 和 `kvindex` 的配合，其实，`<key, value>` 对输出使用的缓冲区，也有类似的过程。

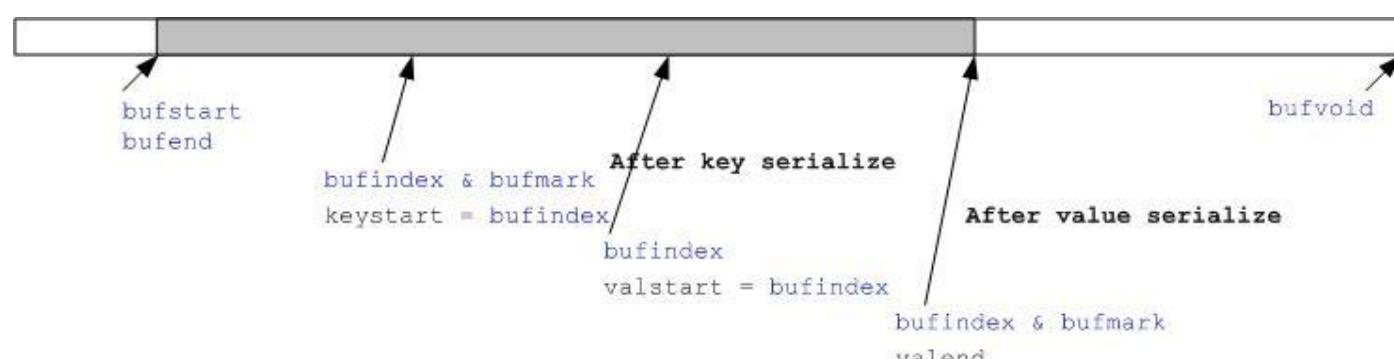
Collect 在处理 `<key, value>` 输出时，会处理一个 `MapBufferTooSmallException`，这是 `value` 的串行化结果太大，不能一次放入缓冲区的指示，这种情况下我们需要调用 `spillSingleRecord`，特殊处理。

### [Hadoop 源代码分析 \( MapTask 辅助类 , III \)](#)

接下来讨论的是 `key, value` 的输出，这部分比较复杂，不过有了前面 `kvstart`，`kvend` 和 `kvindex` 配合的分析，有利于我们理解这部分的代码。

输出缓冲区中，和 `kvstart`，`kvend` 和 `kvindex` 对应的是 `bufstart`，`bufend` 和 `bufmark`。这部分还涉及到变量 `bufvoid`，用于表明实际使用的缓冲区结尾（见后面 `BlockingBuffer.reset` 分析），和变量 `bufmark`，用于标记记录的结尾。这部分代码需要 `bufmark`，是因为 `key` 或 `value` 的输出是变长的，（前面元信息记录大小是常量，就不需要这样的变量）。

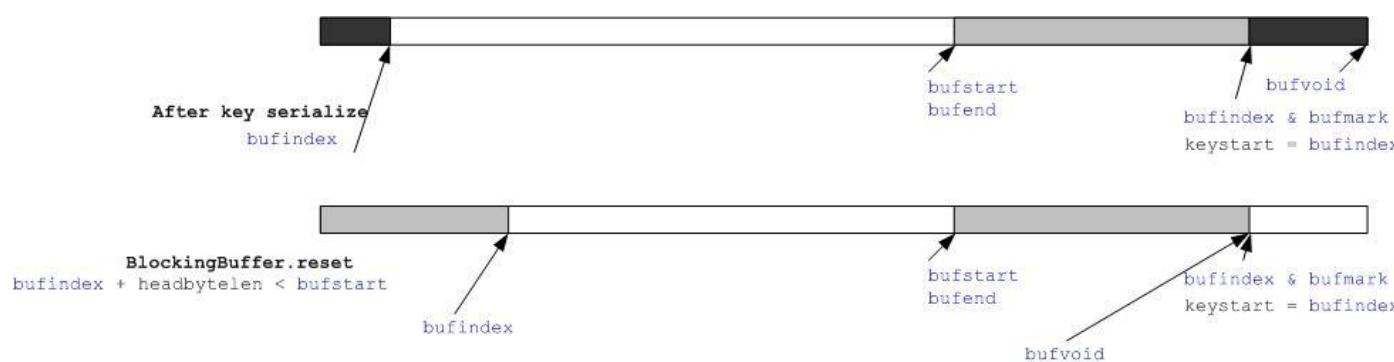
最好的情况是缓冲区没有翻转和 `value` 串行化结果很小，如下图：



先对 key 串行化，然后对 value 做串行化，临时变量 keystart , valstart 和 valend 分别记录了 key 结果的开始位置，value 结果的开始位置和 value 结果的结束位置。

串行化过程中，往缓冲区写是最终调用了 Buffer.write 方法，我们后面再分析。

如果 key 串行化后出现  $\text{bufindex} < \text{keystart}$  那么会调用 BlockingBuffer 的 reset 方法。原因是在 spill 的过程中需要对  $\langle \text{key}, \text{value} \rangle$  排序，这种情况下，传递给 RawComparator 的必须是连续的二进制缓冲区，通过 BlockingBuffer.reset 方法，解决这个问题。下图解释了如何解决这个问题：



当发现 key 的串行化结果出现不连续的情况时，我们会把 bufvoid 设置为 bufmark，见缓冲区开始部分往后挪，然后将原来位于 bufmark 到 bufvoid 出的结果，拷到缓冲区开始处，这样的话，key 串行化的结果就连续存放在缓冲区的最开始处。

上面的调整有一个条件，就是 bufstart 前面的缓冲区能够放下整个 key 串行化的结果，如果不能，处理的方式是将 bufindex 置 0，然后调用 BlockingBuffer 内部的 out 的 write 方法直接输出，这实际调用了 Buffer.write 方法，会启动 spill 过程，最终我们会成功写入 key 串行化的结果。

下面我们看 write 方法。key , value 串行化过程中，往缓冲区写数据是最终调用了 Buffer.write 方法，又是一个复杂的方法。

do-while 循环，直到我们有足够的空间可以写数据（包括缓冲区和 kvindices 和 kvoffsets）

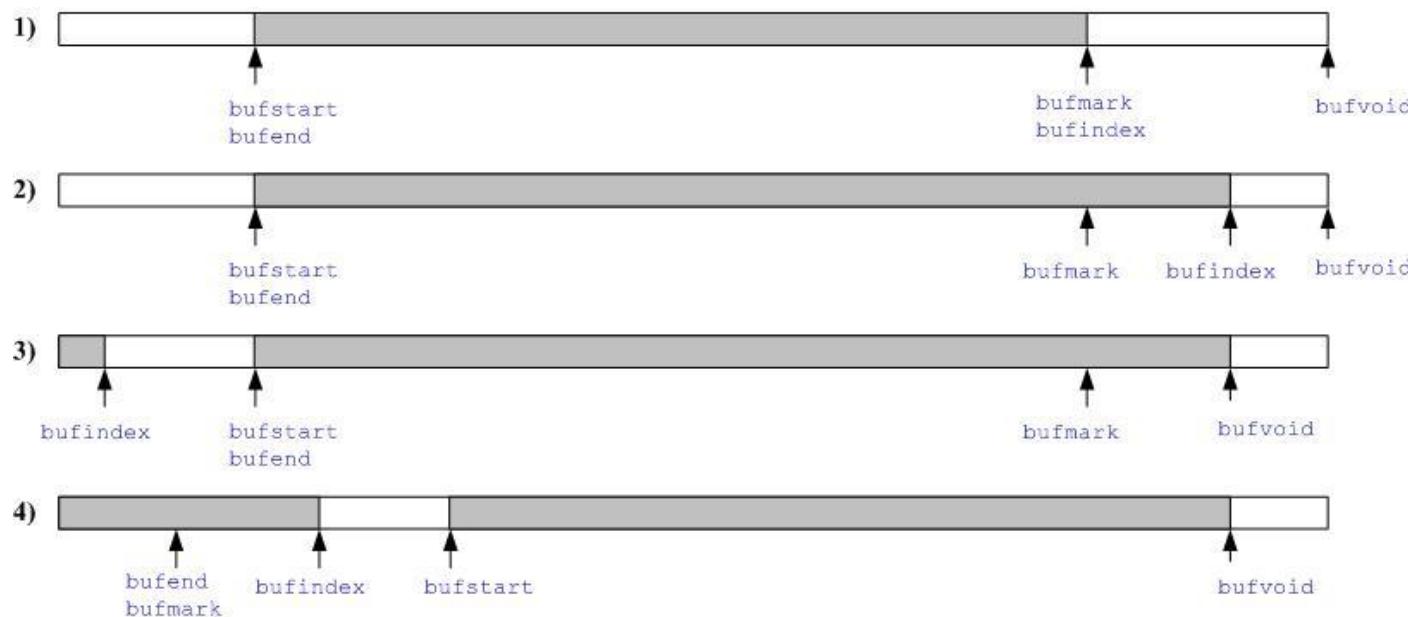
首先我们计算缓冲区连续写是否写满标志 buffull 和缓冲区非连续情况下有足够写空间标志 wrap（这个实在拗口），见下面的讨论；条件 ( buffull && !wrap ) 用于判断目前有没有足够的写空间；

在 spill 没启动的情况下 ( kvstart == kvend )，分两种情况，如果数组中有记录 (kvend != kvindex) ，那么，根据需要（目前输出空间不足或记录数达到 spill 条件）启动 spill 过程；否则，如果空间还是不够 ( buffull && !wrap )，表明这个记录非常大，以至于我们的内存缓冲区不能容下这么大的数据量，抛 MapBufferTooSmallException 异常；

如果空间不足同时 spill 在运行，等待 spillDone；

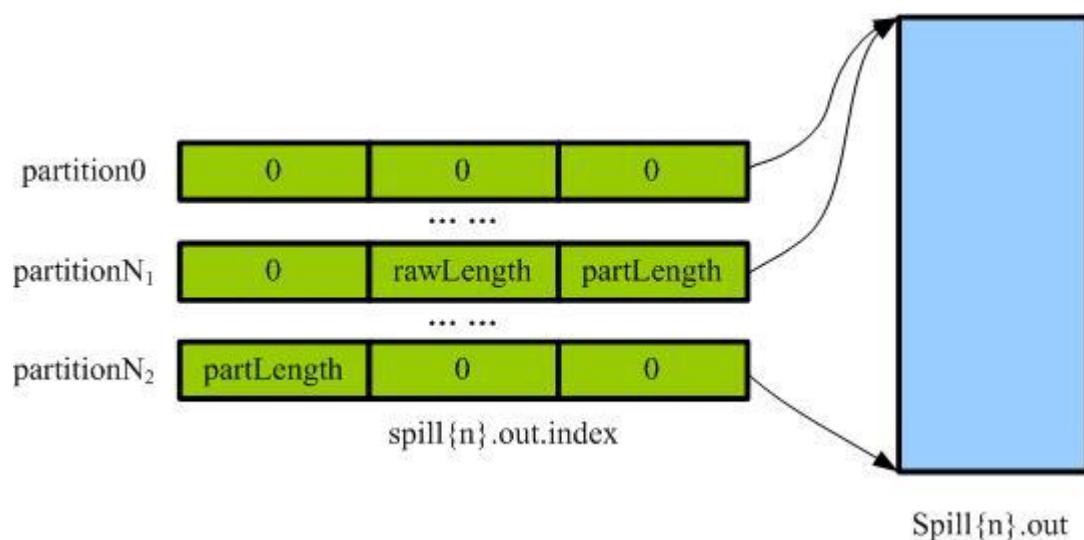
写数据，注意，如果 buffull，则写数据会不连续，则写满剩余缓冲区，然后设置 bufindex=0，并从 bufindex 处接着写。否则，就是从 bufindex 处开始写。

下图给出了缓冲区连续写是否写满标志 buffull 和缓冲区非连续情况下有足够写空间标志 wrap 计算的几种可能：



情况 1 和情况 2 中，buffull 判断为从 bufindex 到 bufvoid 是否有足够的空间容纳写的内容，wrap 是图中白颜色部分的空间是否比输入大，如果是，wrap 为 true；情况 3 和情况 4 中，buffull 判断 bufindex 到 bufstart 的空间是否满足条件，而 wrap 肯定是 false。明显，条件 (buffull && !wrap) 满足时，目前的空间不够一次写。

接下来我们来看 spillSingleRecord，只是用于写放不进内存缓冲区的<key, value>对。过程很流水，首先是创建 SpillRecord 记录，输出文件和 IndexRecord 记录，然后循环，构造 SpillRecord 并在恰当的时候输出记录（如下图），最后输出 spill{n}.index 文件。



前面我们提过 spillThread，在这个系统中它是消费者，这个消费者相当简单，需要 spill 时调用函数 sortAndSpill，进行 spill。sortAndSpill 和 spillSingleRecord 类似，函数的开始也是创建 SpillRecord 记录，输出文件和 IndexRecord 记录，然后，需要在 kvoffsets 上做排序，排完序后顺序访问 kvoffsets，也就是按 partition 顺序访问记录。

按 partition 循环处理排完序的数组，如果没有 combiner，则直接输出记录，否则，调用 combineAndSpill，先做 combin 然后输出。循环的最后记录 IndexRecord 到 SpillRecord。

sortAndSpill 最后是输出 spill{n}.index 文件。

combineAndSpill 比价简单，我们就不分析了。

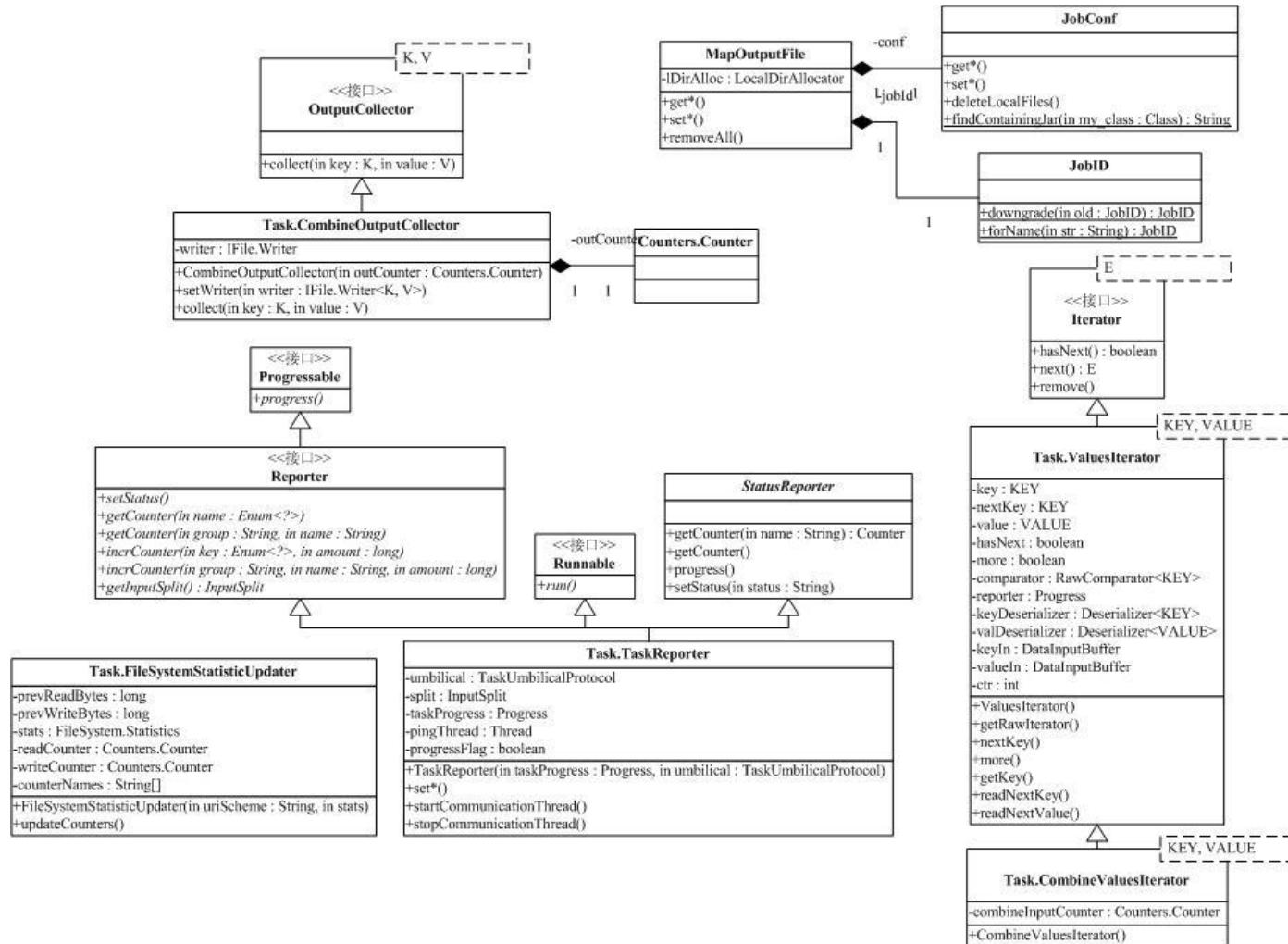
BlockingBuffer 中最后要分析的方法是 flush 方法。调用 flush 方法，意味着 Mapper 的结果都已经 collect 了，需要对缓冲区做一些最后的清理，并合并 spill{n} 文件产生最后的输出。

缓冲区处理部分很简单，先等待可能的 spill 过程完成，然后判断缓冲区是否为空，如果不是，则调用 sortAndSpill，做最后的 spill，然后结束 spill 线程。

flush 合并 spill{n} 文件是通过 mergeParts 方法。如果 Mapper 最后只有一个 spill{n} 文件，简单修改该文件的文件名就可以。如果 Mapper 没有任何输出，那么我们需要创建哑输出（dummy files）。如果 spill{n} 文件多于 1 个，那么按 partition 循环处理所有文件，将处于处理 partition 的记录输出。处理 partition 的过程中可能还会再次调用 combineAndSpill，最记录再做一次 combination，其中还涉及到工具类 Merger，我们就不再深入研究了。

## [Hadoop 源代码分析（Task 的内部类和辅助类）](#)

从前面的图中，我们可以发现 Task 有很多内部类，并拥有大量类成员变量，这些类配合 Task 完成相关的工作，如下图。



MapOutputFile 管理着 Mapper 的输出文件，它提供了一系列 get 方法，用于获取 Mapper 需要的各种文件，这些文件都存放在一个目录下面。

我们假设传入 MapOutputFile 的 JobID 为 job\_200707121733\_0003，TaskID 为 task\_200707121733\_0003\_m\_000005。MapOutputFile 的根为

{mapred.local.dir}/taskTracker/jobcache/{jobid}/{taskid}/output

在下面的讨论中，我们把上面的路径记为{MapOutputFileRoot}

以上面 JobID 和 TaskID 为例，我们有：

{mapred.local.dir}/taskTracker/jobcache/job\_200707121733\_0003/task\_200707121733\_0003\_m\_000005/output

需要注意的是，{mapred.local.dir}可以包含一系列的路径，那么，Hadoop 会在这些根路径下找一个满足要求的目录，建立所需的文件。MapOutputFile 的方法有两种，结尾带 ForWrite 和不带 ForWrite，带 ForWrite 用于创建文件，它需要一个文件大小作为参数，用于检查磁盘空间。不带 ForWrite 用于获取以建立的文件。

getOutputFile : 文件名为{MapOutputFileRoot}/file.out ;

getOutputIndexFile : 文件名为{MapOutputFileRoot}/file.out.index

getSpillFile : 文件名为{MapOutputFileRoot}/spill{spillNumber}.out

getSpillIndexFile : 文件名为{MapOutputFileRoot}/spill{spillNumber}.out.index

以上四个方法用于 Task 子类 MapTask 中；

getInputFile : 文件名为{MapOutputFileRoot}/map\_{mapId}.out

用于 ReduceTask 中。我们到使用到他们的地方再介绍相应的应用场景。

介绍完临时文件管理以后，我们来看 Task.CombineOutputCollector，它继承自 org.apache.hadoop.mapred.OutputCollector，很简单，只是一个 OutputCollector 到 IFile.Writer 的 Adapter，活都让 IFile.Writer 干了。

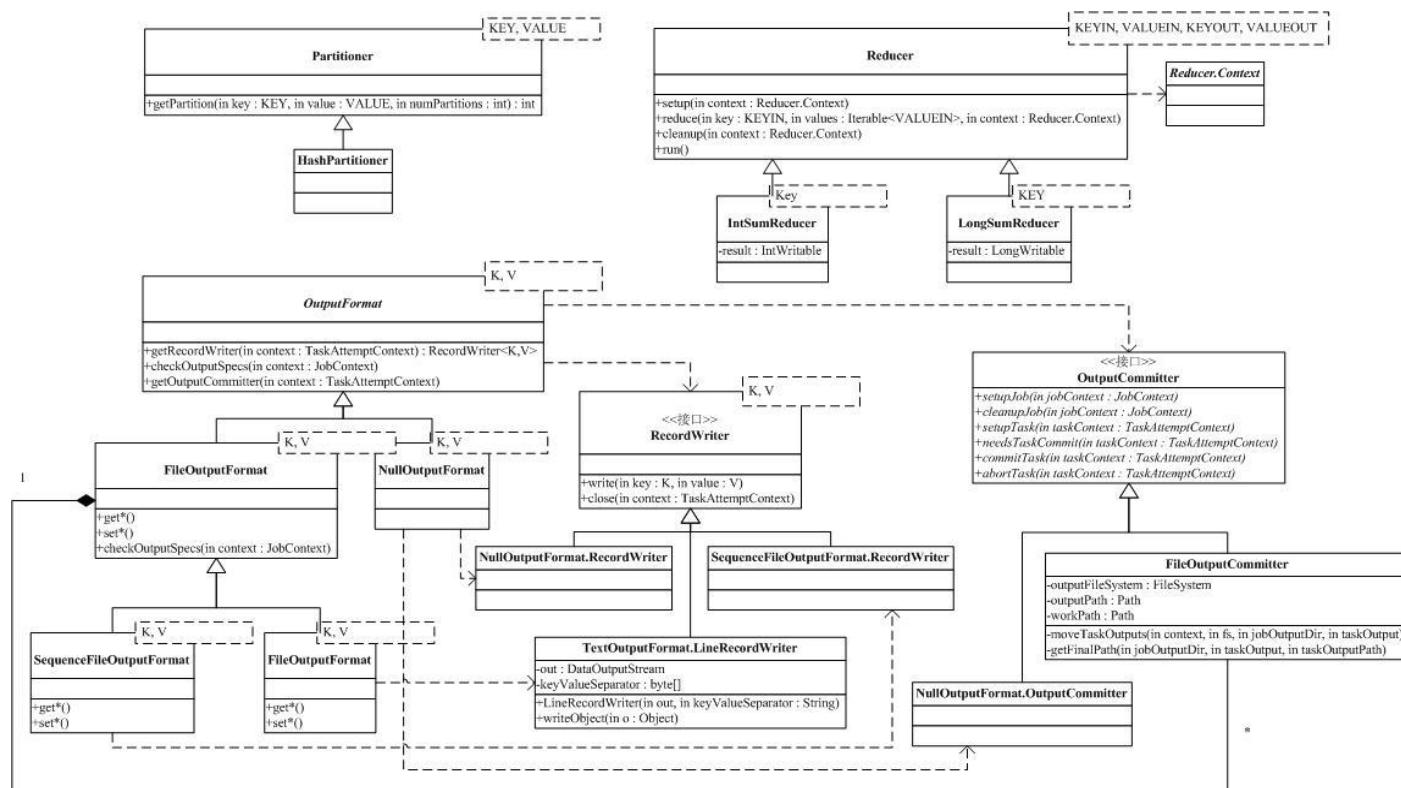
ValuesIterator 用于从 RawKeyValueIterator ( Key , Value 都是 DataInputStream , ValuesIterator 要求该输入已经排序 ) 中获取符合 RawComparator<KEY> comparator 的值的迭代器。它在 Task 中有一个简单子类，CombineValuesIterator。

Task.TaskReporter 用于向 JobTracker 提交计数器报告和状态报告，它实现了计数器报告 Reporter 和状态报告 StatusReporter。为了不影响主线程的工作，TaskReporter 有一个独立的线程，该线程通过 TaskUmbilicalProtocol 接口，利用 Hadoop 的 RPC 机制，向 JobTracker 报告 Task 执行情况。

FileSystemStatisticUpdater 用于记录对文件系统的对/写操作字节数，是个简单的工具类。

[\*\*Hadoop 源代码分析 \( mapreduce.lib.partition/reduce/output \)\*\*](#)

Map 的结果，会通过 partition 分发到 Reducer 上，Reducer 做完 Reduce 操作后，通过 OutputFormat，进行输出，下面我们就来分析参与这个过程的类。



Mapper 的结果，可能送到可能的 Combiner 做合并，Combiner 在系统中并没有自己的基类，而是用 Reducer 作为 Combiner 的基类，他们对外的功能是一样的，只是使用的位置和使用时的上下文不太一样而已。

Mapper 最终处理的结果对<key, value>，是需要送到 Reducer 去合并的，合并的时候，有相同 key 的键/值对会送到同一个 Reducer 那，哪个 key 到哪个 Reducer 的分配过程，是由 Partitioner 规定的，它只有一个方法，输入是 Map 的结果对<key, value>和 Reducer 的数目，输出则是分配的 Reducer ( 整数编号 )。系统缺省的 Partitioner 是 HashPartitioner，它以 key 的 Hash 值对 Reducer 的数目取模，得到对应的 Reducer。

Reducer 是所有用户定制 Reducer 类的基类，和 Mapper 类似，它也有 setup, reduce, cleanup 和 run 方法，其中 setup 和 cleanup 含义和 Mapper 相同，reduce 是真正合并 Mapper 结果的地方，它的输入是 key 和这个 key 对应的所有 value 的一个迭代器，同时还包括 Reducer 的上下文。系统中定义了两个非常简单的 Reducer，IntSumReducer 和 LongSumReducer，分别用于对整形/长整型的 value 求和。

Reduce 的结果，通过 Reducer.Context 的方法 collect 输出到文件中，和输入类似，Hadoop 引入了 OutputFormat。OutputFormat 依赖两个辅助接口：RecordWriter 和 OutputCommitter，来处理输出。RecordWriter 提供了 write 方法，用于输出<key, value>和 close 方法，用于关闭对应的输出。OutputCommitter 提供了一系列方法，用户通过实现这些方法，可以定制 OutputFormat 生存期某些阶段需要的特殊操作。我们在 TaskInputOutputContext 中讨论过这些方法（明显，TaskInputOutputContext 是 OutputFormat 和 Reducer 间的桥梁）。

OutputFormat 和 RecordWriter 分别对应着 InputFormat 和 RecordReader，系统提供了空输出 NullOutputFormat( 什么结果都不输出，NullOutputFormat.RecordWriter 只是示例，系统中没有定义 )，LazyOutputFormat ( 没在类图中出现，不分析 )，FilterOutputFormat ( 不分析 ) 和基于文件 FileOutputFormat 的 SequenceFileOutputFormat 和 TextOutputFormat 输出。

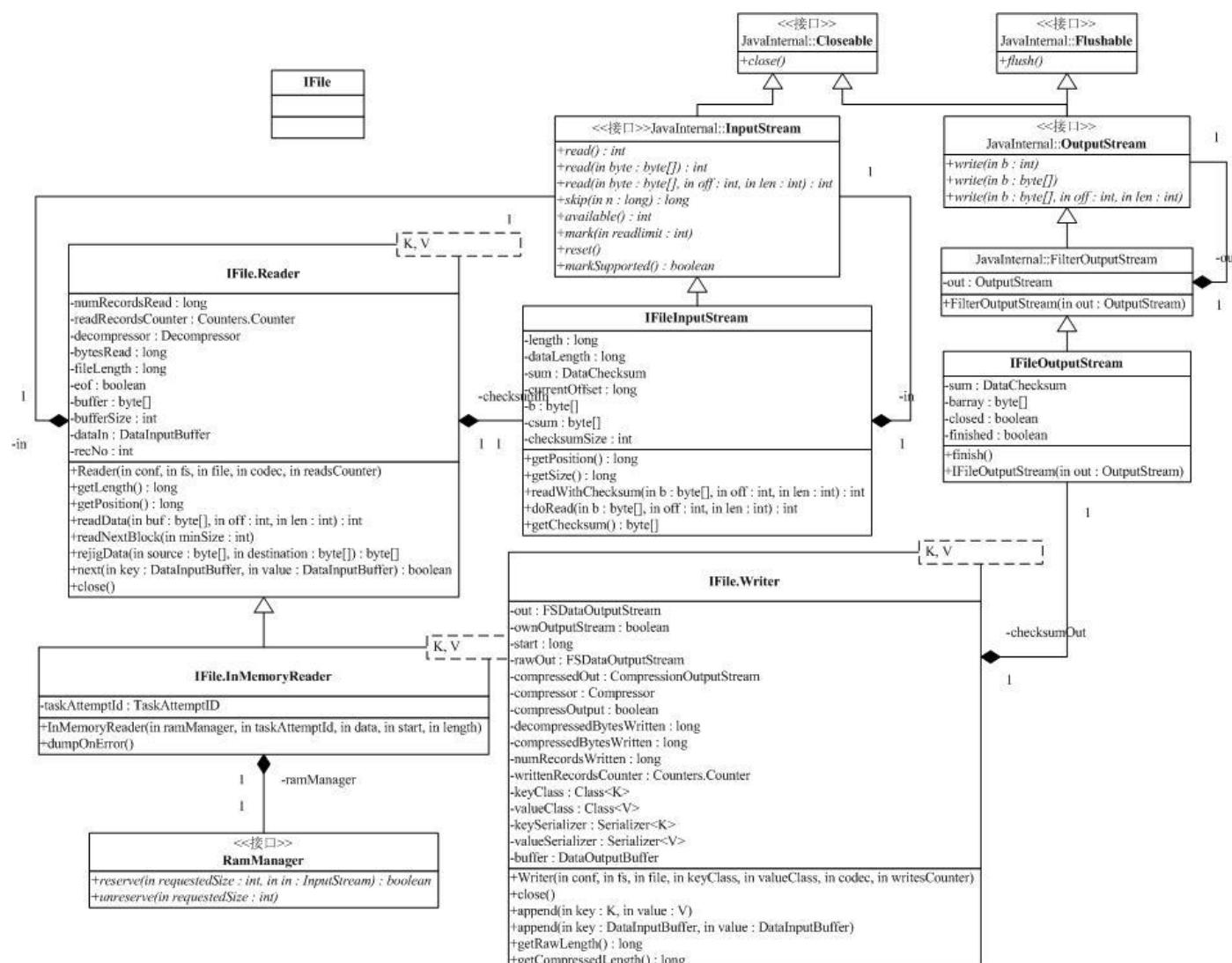
基于文件的输出 `FileOutputFormat` 利用了一些配置项配合工作，包括 `mapred.output.compress`：是否压缩；`mapred.output.compression.codec`：压缩方法；`mapred.output.dir`：输出路径；`mapred.work.output.dir`：输出工作路径。`FileOutputFormat` 还依赖于 `FileOutputCommitter`，通过 `FileOutputCommitter` 提供一些和 Job，Task 相关的临时文件管理功能。如 `FileOutputCommitter` 的 `setupJob`，会在输出路径下创建一个名为 `_temporary` 的临时目录，`cleanupJob` 则会删除这个目录。

`SequenceFileOutputFormat` 输出和 `TextOutputFormat` 输出分别对应输入的 `SequenceFileInputFormat` 和 `TextInputFormat`，我们就不再详细分析啦。

## Hadoop 源代码分析 ( IFile )

Mapper 的输出，在发送到 Reducer 前是存放在本地文件系统的，`IFile` 提供了对 Mapper 输出的管理。我们已经知道，Mapper 的输出是 `<Key, Value>` 对，`IFile` 以记录 `<key-len, value-len, key, value>` 的形式存放了这些数据。为了保存键值对的边界，很自然 `IFile` 需要保存 `key-len` 和 `value-len`。

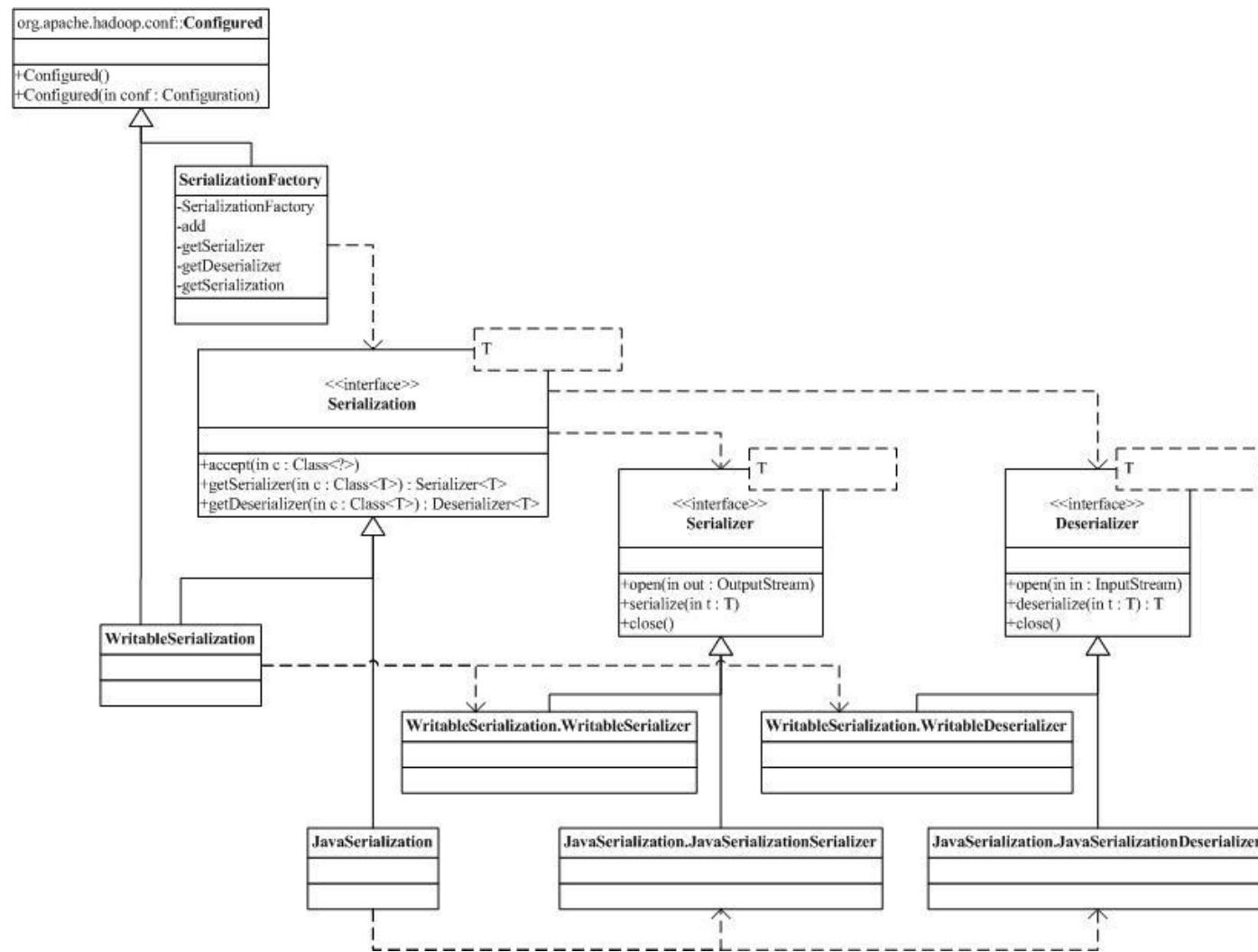
和 `IFile` 相关的类图如下：



其中，文件流形式的输入和输出是由 `IFileInputStream` 和 `IFileOutputStream` 抽象。以记录形式的读/写操作由 `IFile.Reader/IFile.Writer` 提供，`IFile.InMemoryReader` 用于读取存在于内存中的 `IFile` 文件格式数据。

我们以输出为例，来分析这部分的实现。首先是下图的和序列化反序列化相关的 `Serialization/Deserializer`，这部分的 code 是在包 `org.apache.hadoop.io.serializer`。序列化由 `Serializer` 抽象，通过 `Serializer` 的实现，用户可以利用 `serialize` 方法把对象序列化到通过 `open` 方法打开的输出流里。`Deserializer` 提供的是相反的过程，对应的方法是 `deserialize`。

`hadoop.io.serializer` 中还实现了配合工作的 `Serialization` 和对应的工厂 `SerializationFactory`。两个具体的实现是 `WritableSerialization` 和 `JavaSerialization`，分别对应了 `Writable` 的序列化反序列化和 Java 本身带的序列化反序列化。



有了 Serializer/Deserializer，我们来分析 IFile.Writer。Writer 的构造函数是：

```
public Writer(Configuration conf, FSDataOutputStream out,  
Class<K> keyClass, Class<V> valueClass,  
CompressionCodec codec, Counters.Counter writesCounter)
```

conf，配置参数，out 是 Writer 的输出，keyClass 和 valueClass 是输出的 Key，Value 的 class 属性，codec 是对输出进行压缩的方法，参数 writesCounter 用于对输出字节数进行统计的 Counters.Counter。通过这些参数，我们可以构造我们使用的支持压缩功能的输出流（类成员 out，类成员 rawOut 保存了构造函数传入的 out），相关的计数器，还有就是 Key，Value 的 Serializer 方法。

Writer 最主要的方法是 append 方法（居然不是 write 方法，呵呵），有两种形式：

```
public void append(K key, V value) throws IOException {  
  
public void append(DataInputBuffer key, DataInputBuffer value)
```

append(K key, V value)的主要过程是检查参数，然后将 key 和 value 序列化到 DataOutputBuffer 中，并获取序列化后的长度，最后把长度(2个)和 DataOutputBuffer 中的结果写到输出，并复位 DataOutputBuffer 和计数。append(DataInputBuffer key, DataInputBuffer value)处理过程也比较类似，就不再分析了。

`close` 方法中需要注意的是，我们需要标记文件尾，或者是流结束。目前是通过写 2 个值为 `EOF_MARKER` 的长度来做标记。

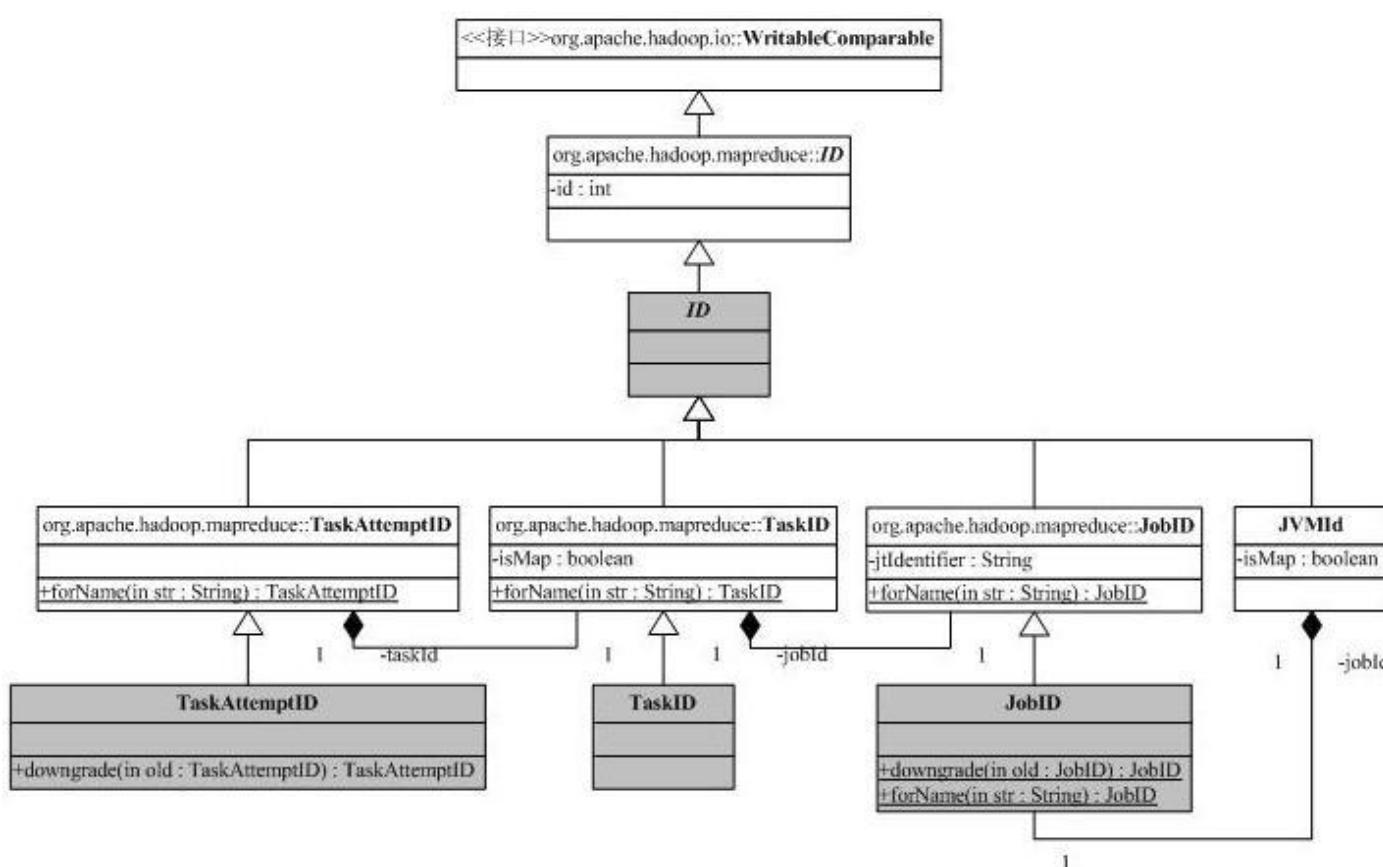
IFileOutputStream 是用于配合 Writer 的输出流，它会在 IFiles 的最后添加校验数据。当 Writer 调用 IFileOutputStream 的 write 操作时，IFileOutputStream 计算并保持校验和，流被 close 的时候，校验结果会写到对应文件的文件尾。实际上存放在磁盘上的文件是一系列的<key-len, value-len, key, value>记录和校验结果。

Reader 的相关过程，我们就不再分析了。

## [Hadoop 源代码分析 \(\\*IDs 类和\\*Context 类\)](#)

我们开始来分析 Hadoop MapReduce 的内部的运行机制。用户向 Hadoop 提交 Job ( 作业 )，作业在 JobTracker 对象的控制下执行。Job 被分解成为 Task( 任务 )，分发到集群中，在 TaskTracker 的控制下运行。Task 包括 MapTask 和 ReduceTask，是 MapReduce 的 Map 操作和 Reduce 操作执行的地方。这中任务分布的方法比较类似于 HDFS 中 NameNode 和 DataNode 的分工，NameNode 对应的是 JobTracker，DataNode 对应的是 TaskTracker。JobTracker，TaskTracker 和 MapReduce 的客户端通过 RPC 通信，具体可以参考 HDFS 部分的分析。

我们先来分析一些辅助类，首先是和 ID 有关的类，ID 的继承树如下：



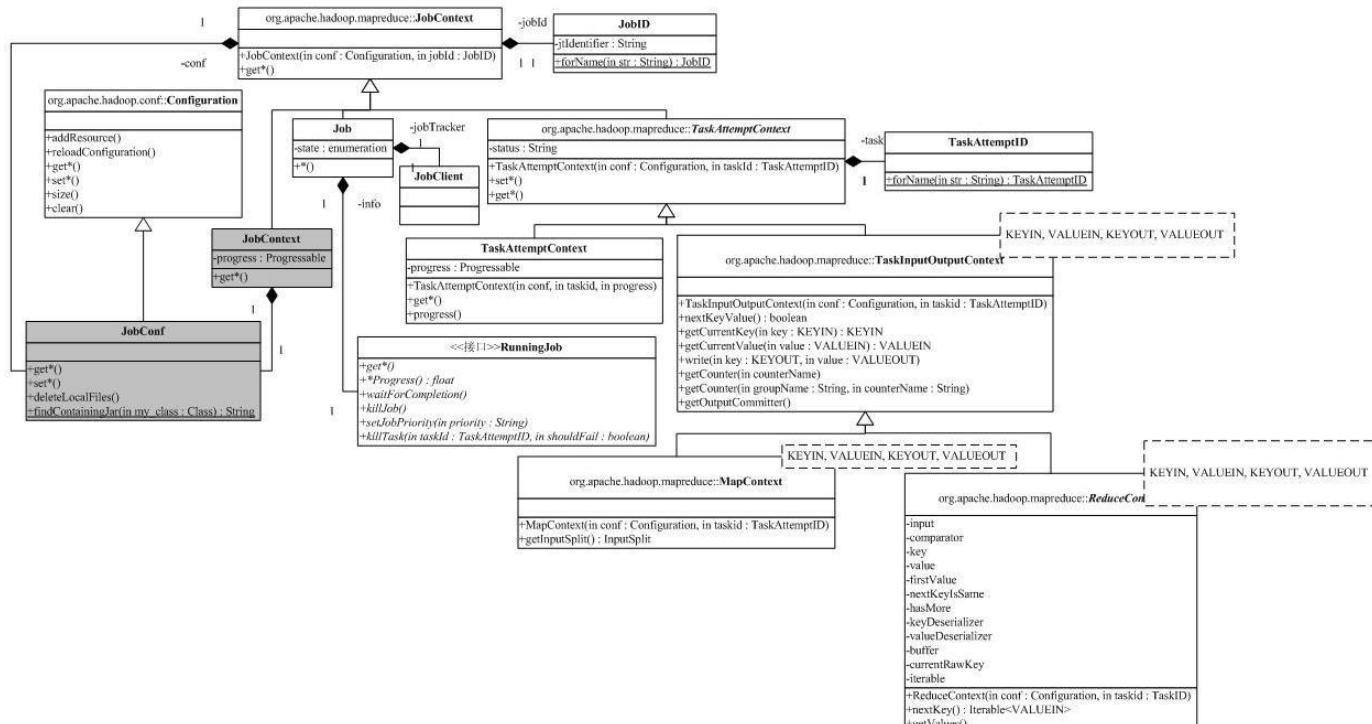
这张图可以看出现在 Hadoop 的 org.apache.hadoop.mapred 向 org.apache.hadoop.mapreduce 迁移带来的一些问题，其中灰色是标注为@Deprecated 的。ID 携带一个整型，实现了 WritableComparable 接口，这表明它可以比较，而且可以被 Hadoop 的 io 机制串行化/解串行化（必须实现 compareTo/readFields/write 方法）。JobID 是系统分配给作业的唯一标识符，它的 toString 结果是 job\_<jobtrackerID>\_<jobNumber>。例子：job\_200707121733\_0003 表明这是 jobtracker 200707121733（利用 jobtracker 的开始时间作为 ID）的第 3 号作业。

作业分成任务执行，任务号 TaskID 包含了它所属的作业 ID，同时也有任务 ID，同时还保持了这是否是一个 Map 任务（成员变量 isMap）。任务号的字符串表示为 task\_<jobtrackerID>\_<jobNumber>\_[m|r]\_<taskNumber>，如 task\_200707121733\_0003\_m\_00005 表示作业 200707121733\_0003 的 00005 号任务，改任务是一个 Map 任务。

一个任务有可能有多个执行（错误恢复/消除 Stragglers 等），所以必须区分任务的多个执行，这是通过类 TaskAttemptID 来完成，它在任务号的基础上添加了尝试号。一个任务尝试号的例子是 attempt\_200707121733\_0003\_m\_000005\_0，它是任务 task\_200707121733\_0003\_m\_000005 的第 0 号尝试。

JVMId 用于管理任务执行过程中的 Java 虚拟机，我们后面再讨论。

为了使 Job 和 Task 工作，Hadoop 提供了一系列的上下文，这些上下文保存了 Job 和 Task 工作的信息。



处于继承树的最上方是 org.apache.hadoop.mapreduce.JobContext，前面我们已经介绍过了，它提供了 Job 的一些只读属性，两个成员变量，一个保存了 JobID，另一个类型为 JobConf，JobContext 中除了 JobID 外，其它的信息都保持在 JobConf 中。它定义了如下配置项：

`mapreduce.inputformat.class` : InputFormat 的实现

`mapreduce.map.class` : Mapper 的实现

`mapreduce.combine.class`: Reducer 的实现

`mapreduce.reduce.class` : Reducer 的实现

`mapreduce.outputformat.class`: OutputFormat 的实现

`mapreduce.partitioner.class`: Partitioner 的实现

同时，它提供方法，使得通过类名，利用 Java 反射提供的 `Class.forName` 方法，获得类对应的 Class。

`org.apache.hadoop.mapred` 的 `JobContext` 对象比 `org.apache.hadoop.mapreduce.JobContext` 多了成员变量 `progress`，用于获取进度信息，它类型为 `JobConf` 成员 `job` 指向 `mapreduce.JobContext` 对应的成员，没有添加任何新功能。

JobConf 继承自 Configuration，保持了 MapReduce 执行需要的一些配置信息，它管理着 46 个配置参数，包括上面 mapreduce 配置项对应的老版本形式，如 mapreduce.map.class 对应 mapred.mapper.class。这些配置项我们在使用到它们的时候再介绍。

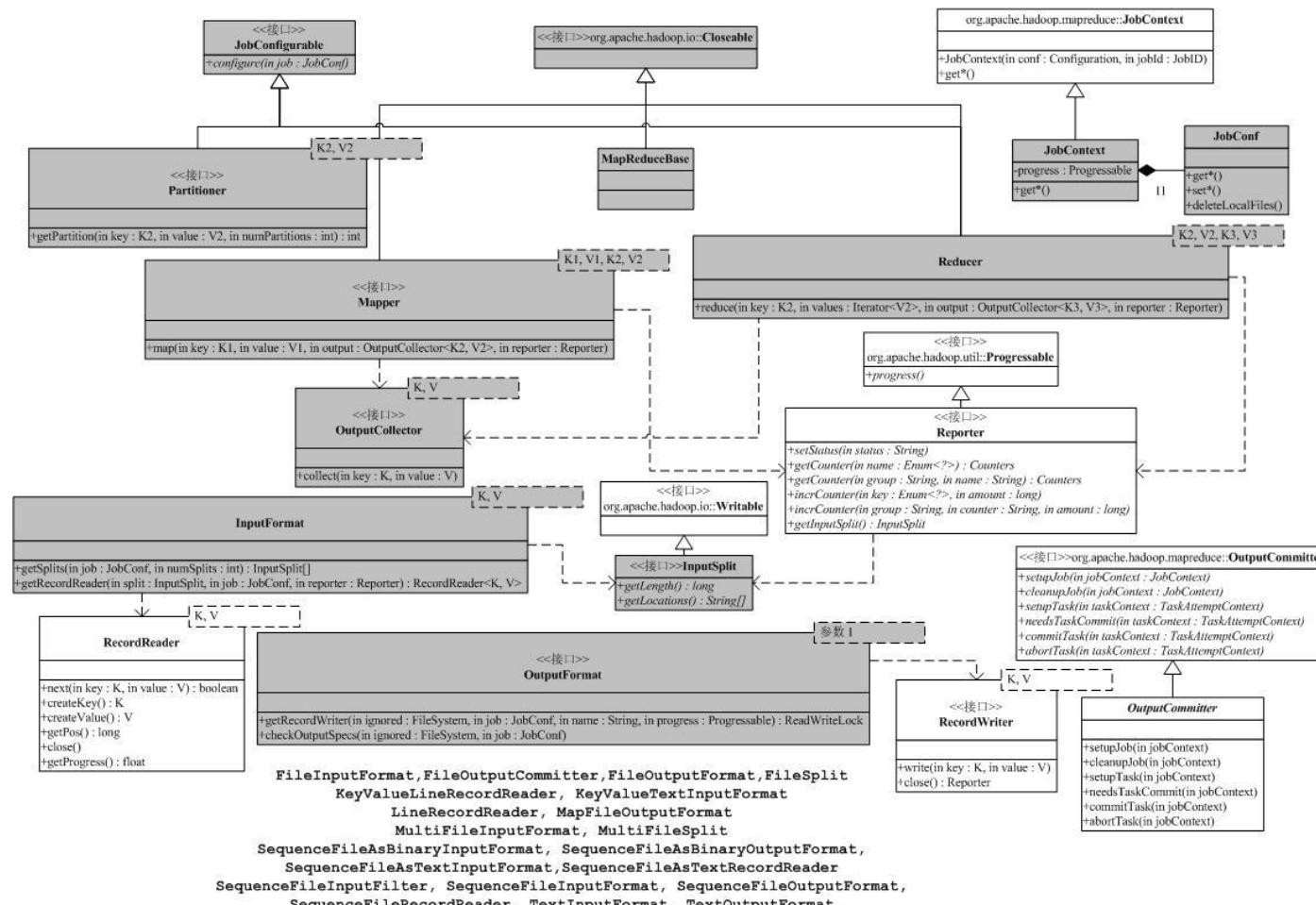
org.apache.hadoop.mapreduce.JobContext 的子类 Job 前面也已经介绍了，后面在讨论系统的动态行为时，再回来看它。

TaskAttemptContext 用于任务的执行，它引入了标识任务执行的 TaskAttemptID 和任务状态 status，并提供新的访问接口。org.apache.hadoop.mapred 的 TaskAttemptContext 继承自 mapreduce 的对应版本，只是增加了记录进度的 progress。

TaskInputOutputContext 和它的子类都在包 org.apache.hadoop.mapreduce 中，前面已经分析过了，我们就不再罗嗦。

## [Hadoop 源代码分析（包 hadoop.mapred 中的 MapReduce 接口）](#)

前面已经完成了对 org.apache.hadoop.mapreduce 的分析，这个包提供了 Hadoop MapReduce 部分的应用 API，用于用户实现自己的 MapReduce 应用。但这些接口是给未来的 MapReduce 应用的，目前 MapReduce 框架还是使用老系统（参考补丁 HADOOP-1230）。下面我们来分析 org.apache.hadoop.mapred，首先还是从 mapred 的 MapReduce 框架开始分析，下面的类图（灰色部分为标记为@Deprecated 的类/接口）：



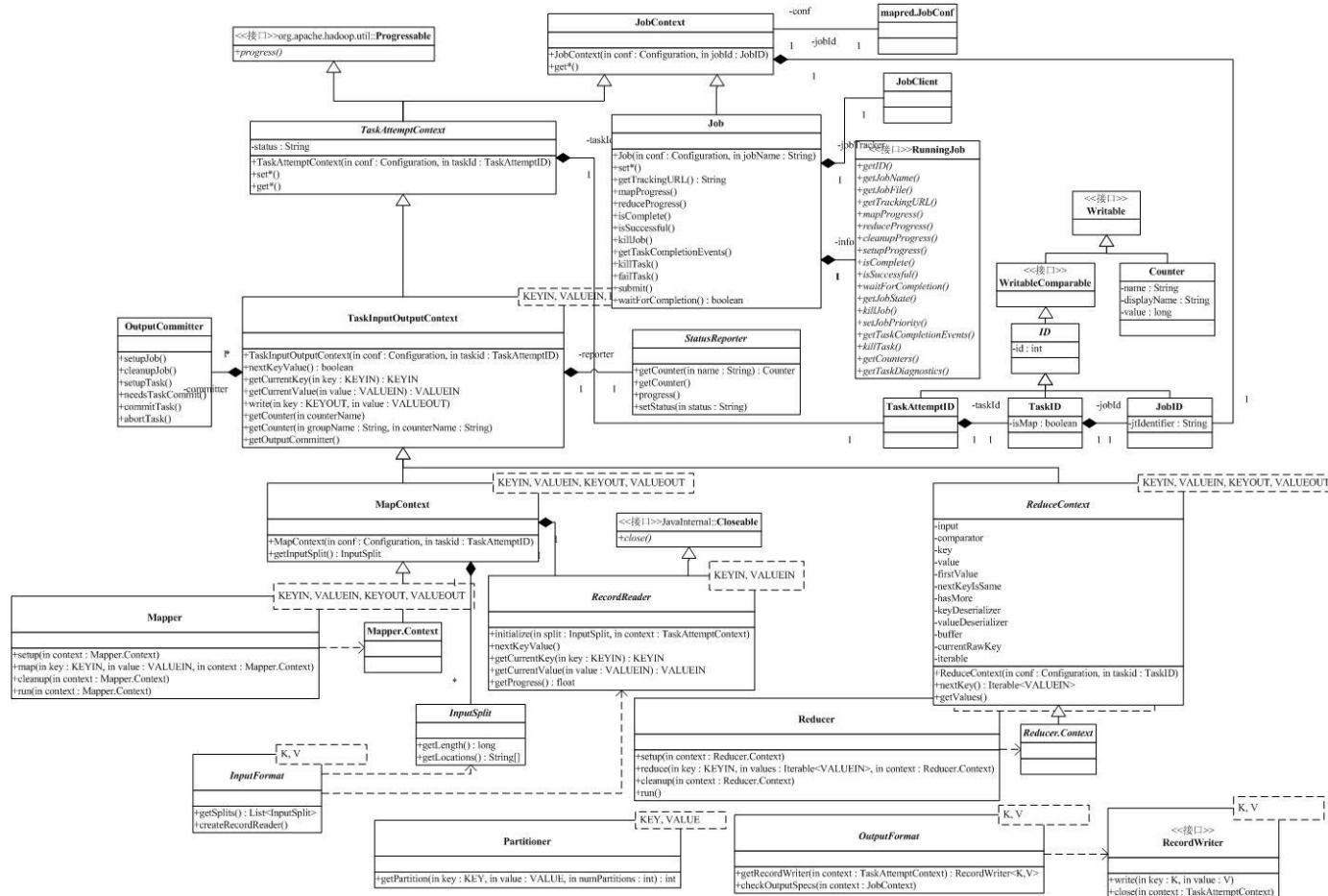
我们把包 mapreduce 的类图附在下面，对比一下，我们就会发现，org.apache.hadoop.mapred 中的 MapReduce API 相对来说很简单，主要是少了和 Context 相关的类，那么，好多在 mapreduce 中通过 context 来完成的工作，就需要通过参数来传递，如 Map 中的输出，老版本是：

```
output.collect(key, result); // output' s type is: OutputCollector
```

新版本是：

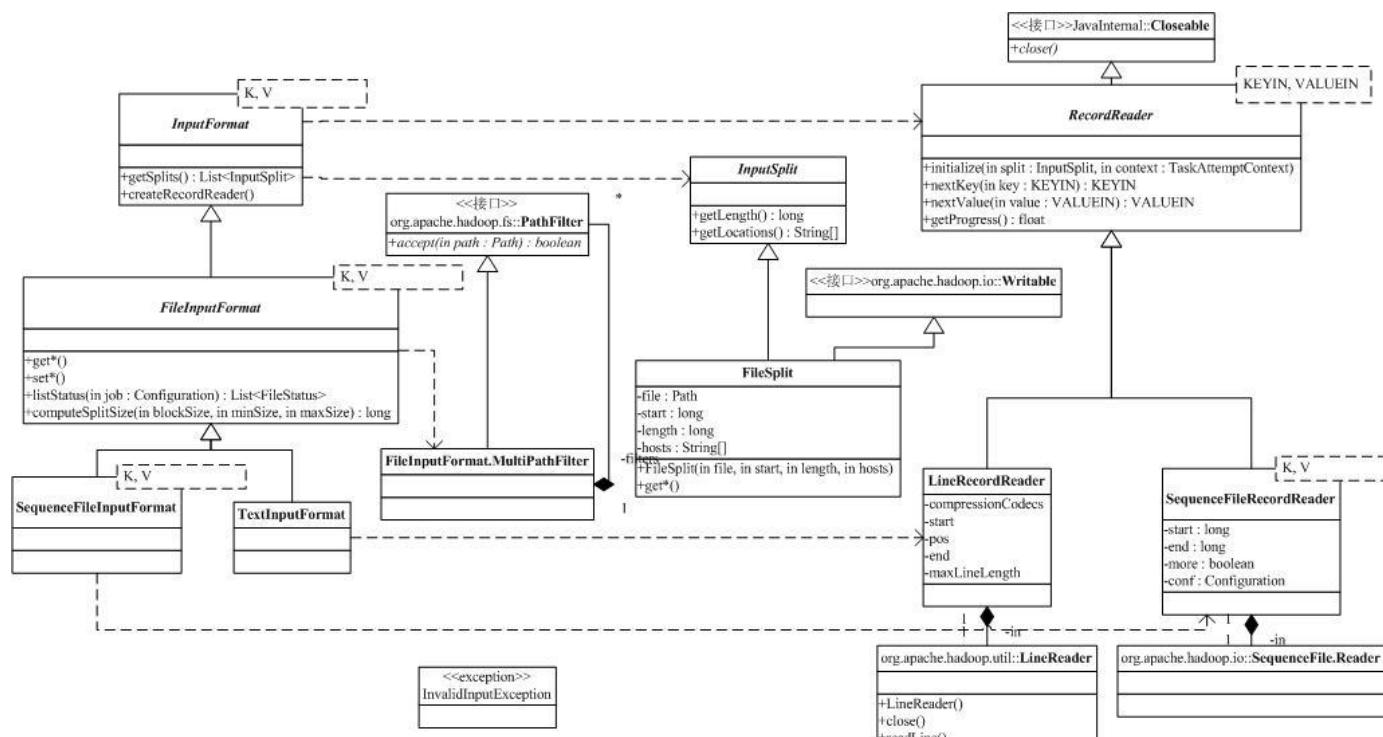
```
context.write(key, result); // output's type is: Context
```

它们分别使用 OutputCollector 和 Mapper.Context 来输出 map 的结果，显然，原有 OutputCollector 的新 API 中就不再需要。总体来说，老版本的 API 比较简单，MapReduce 过程中关键的对象都有，但可扩展性不是很强。同时，老版中提供的辅助类也很多，我们前面分析的 FileOutputFormat，也有对应的实现，我们就不再讨论了。



## Hadoop 源代码分析 (包 mapreduce.lib.input)

接下来我们按照 MapReduce 过程中数据流动的顺序，来分解 org.apache.hadoop.mapreduce.lib.\* 的相关内容，并介绍对应该的功能。首先是 input 部分，它实现了 MapReduce 的数据输入部分。类图如下：



类图的右上角是 InputFormat，它描述了一个 MapReduce Job 的输入，通过 InputFormat，Hadoop 可以：

检查 MapReduce 输入数据的正确性；

将输入数据切分为逻辑块 InputSplit，这些块会分配给 Mapper；

提供一个 RecordReader 实现，Mapper 用该实现从 InputSplit 中读取输入的<K,V>对。

在 org.apache.hadoop.mapreduce.lib.input 中，Hadoop 为所有基于文件的 InputFormat 提供了一个虚基类 FileInputFormat。下面几个参数可以用于配置 FileInputFormat：

mapred.input.pathFilter.class：输入文件过滤器，通过过滤器的文件才会加入 InputFormat；

mapred.min.split.size：最小的划分大小；

mapred.max.split.size：最大的划分大小；

mapred.input.dir：输入路径，用逗号做分割。

类中比较重要的方法有：

**protected** List<FileStatus> listStatus(Configuration job)

递归获取输入数据目录中的所有文件（包括文件信息），输入的 job 是系统运行的配置 Configuration，包含了上面我们提到的参数。

**public** List<InputSplit> getSplits(JobContext context)

将输入划分为 InputSplit，包含两个循环，第一个循环处理所有的文件，对于每一个文件，根据输入的划分最大/最小值，循环得到文件上的划分。注意，划分不会跨越文件。

FileInputFormat 没有实现 InputFormat 的 createRecordReader 方法。

FileInputFormat 有两个子类，SequenceFileInputFormat 是 Hadoop 定义的一种二进制形式存放的键/值文件（参考 <http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/io/SequenceFile.html>），它有自己定义的文件布局。由于它有特殊的扩展名，所以 SequenceFileInputFormat 重载了 listStatus，同时，它实现了 createRecordReader，返回一个 SequenceFileRecordReader 对象。TextInputFormat 处理的是文本文件，createRecordReader 返回的是 LineRecordReader 的实例。这两个类都没有重载 FileInputFormat 的 getSplits 方法，那么，在他们对于的 RecordReader 中，必须考虑 FileInputFormat 对输入的划分方式。

FileInputFormat 的 getSplits，返回的是 FileSplit。这是一个很简单的类，包含的属性（文件名，起始偏移量，划分的长度和可能的目标机器）已经足以说明这个类的功能。

RecordReader 用于在划分中读取<Key,Value>对。RecordReader 有五个虚方法，分别是：

initialize : 初始化 , 输入参数包括该 Reader 工作的数据划分 InputSplit 和 Job 的上下文 context ;

nextKey : 得到输入的下一个 Key , 如果数据划分已经没有新的记录 , 返回空 ;

nextValue : 得到 Key 对应的 Value , 必须在调用 nextKey 后调用 ;

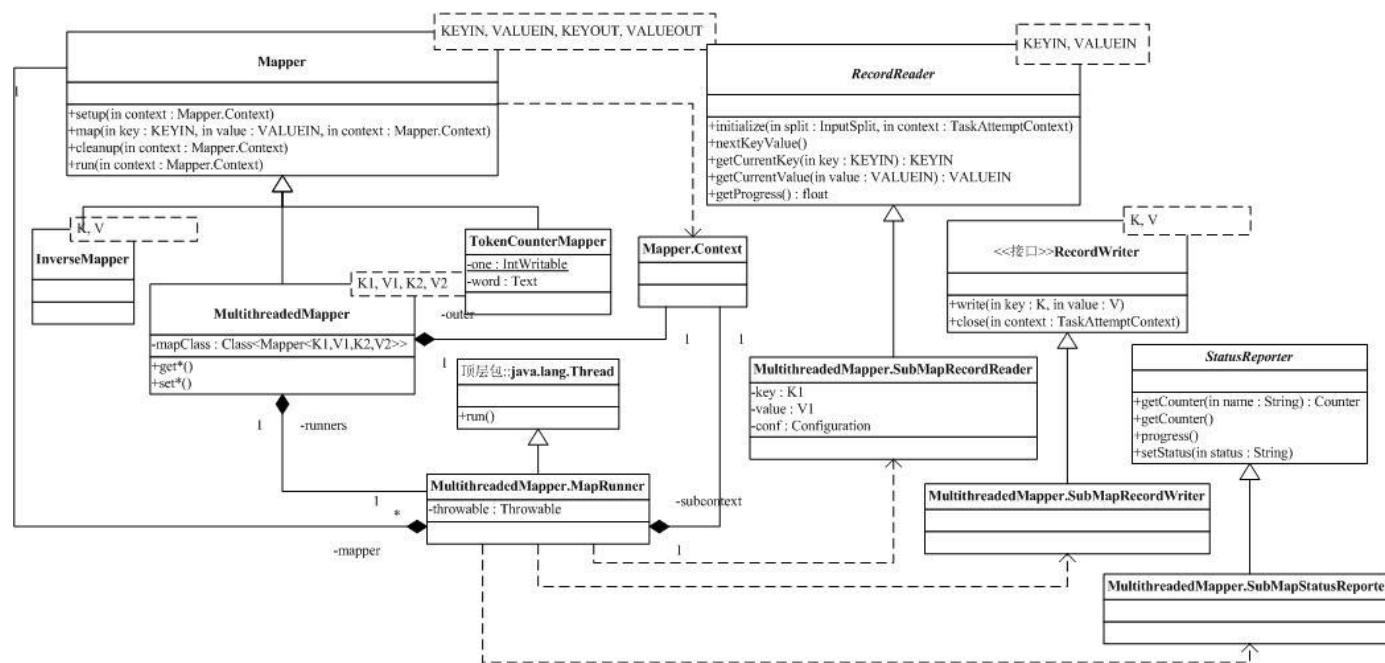
getProgress : 得到现在的进度 ;

close , 来自 java.io 的 Closeable 接口 , 用于清理 RecordReader.

我们以 LineRecordReader 为例 , 来分析 RecordReader 的构成。前面我们已经分析过 FileInputFormat 对文件的划分了 , 划分完的 Split 包括了文件名 , 起始偏移量 , 划分的长度。由于文件是文本文件 , LineRecordReader 的初始化方法 initialize 会创建一个基于行的读取对象 LineReader ( 定义在 org.apache.hadoop.util 中 , 我们就不分析啦 ) , 然后跳过输入的最开始的部分 ( 只在 Split 的起始偏移量不为 0 的情况下进行 , 这时最开始的部分可能是上一个 Split 的最后一行的一部分 ) 。 nextKey 的处理很简单 , 它使用当前的偏移量作为 Key , nextValue 当然就是偏移量开始的那一行了 ( 如果行很长 , 可能出现截断 ) 。进度 getProgress 和 close 都很简单。

## [Hadoop 源代码分析 \( 包 mapreduce.lib.map \)](#)

Hadoop 的 MapReduce 框架中 , Map 动作通过 Mapper 类来抽象。一般来说 , 我们会实现自己特殊的 Mapper , 并注册到系统中 , 执行时 , 我们的 Mapper 会被 MapReduce 框架调用。Mapper 类很简单 , 包括一个内部类和四个方法 , 静态结构图如下 :



内部类 Context 继承自 MapContext , 并没有引入任何新的方法。

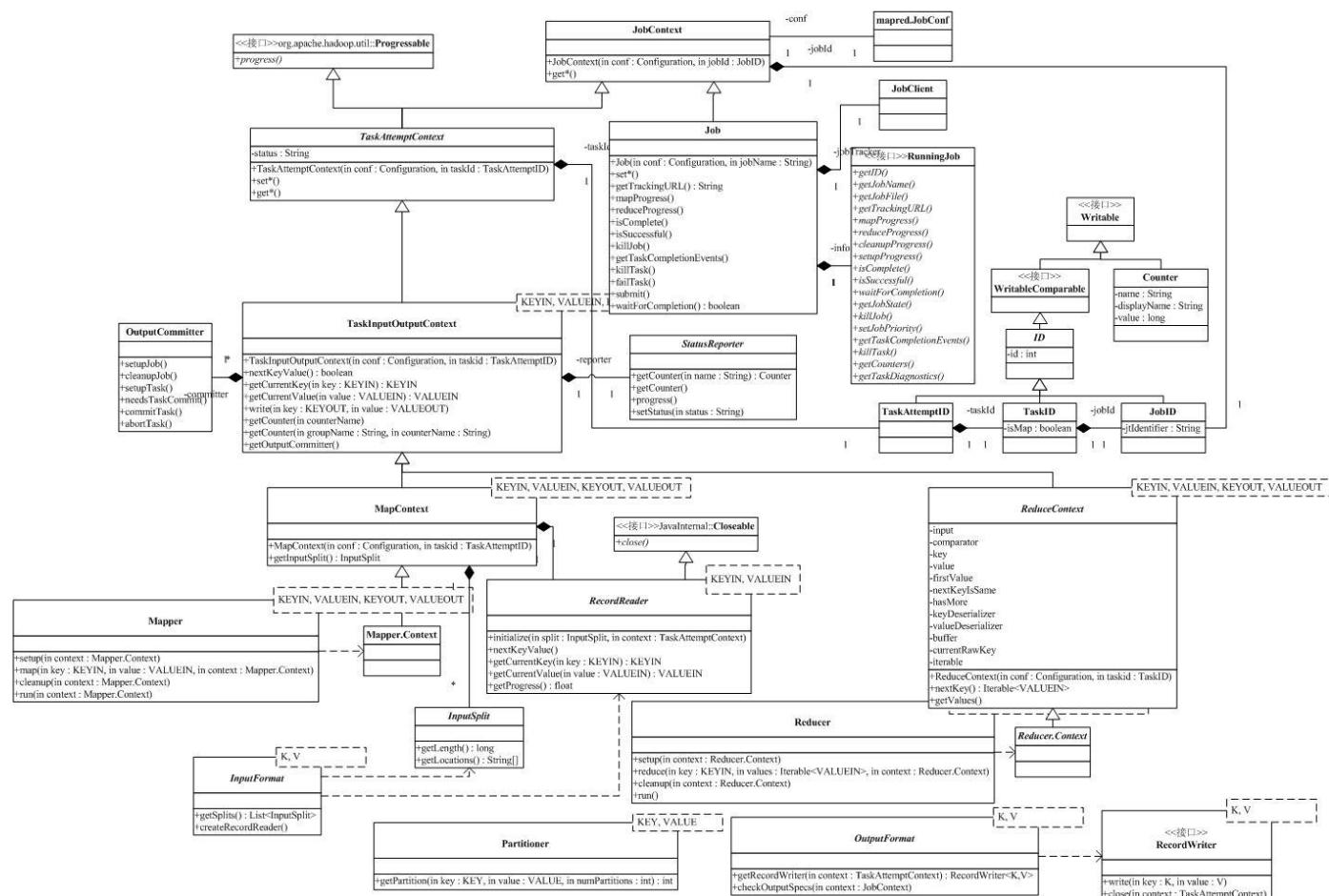
Mapper 的四个方法是 setup , map , cleanup 和 run。其中 , setup 和 cleanup 用于管理 Mapper 生命周期中的资源 , setup 在完成 Mapper 构造 , 即将开始执行 map 动作前调用 , cleanup 则在所有的 map 动作完成后被调用。方法 map 用于对一次输入的 key/value 对进行 map 动作。 run 方法执行了上面描述的过程 , 它调用 setup , 让后迭代所有的 key/value 对 , 进行 map , 最后调用 cleanup。

org.apache.hadoop.mapreduce.lib.map 中实现了 Mapper 的三个子类，分别是 InverseMapper( 将输入 <key, value> map 为输出 <value, key> ) , MultithreadedMapper ( 多线程执行 map 方法 ) 和 TokenCounterMapper ( 对输入的 value 分解为 token 并计数 ) 。其中最复杂的是 MultithreadedMapper ，我们就以它为例，来分析 Mapper 的实现。

MultithreadedMapper 会启动多个线程执行另一个 Mapper 的 map 方法，它会启动 mapred.map.multithreadedrunner.threads( 配置项 ) 个线程执行 Mapper : mapred.map.multithreadedrunner.class( 配置项 ) 。 MultithreadedMapper 重写了基类 Mapper 的 run 方法，启动 N 个线程（对应的类为 MapRunner ）执行 mapred.map.multithreadedrunner.class( 我们称为目标 Mapper ) 的 run 方法（就是说，目标 Mapper 的 setup 和 cleanup 会被执行多次）。目标 Mapper 共享同一份 InputSplit ，这就意味着，对 InputSplit 的数据读必须线程安全。为此， MultithreadedMapper 引入了内部类 SubMapRecordReader , SubMapRecordWriter , SubMapStatusReporter ，分别继承自 RecordReader , RecordWriter 和 StatusReporter ，它们通过互斥访问 MultithreadedMapper 的 Mapper.Context ，实现了对同一份 InputSplit 的线程安全访问，为 Mapper 提供所需的 Context 。这些类的实现方法都很简单。

## [Hadoop 源代码分析 \( 包 org.apache.hadoop.mapreduce \)](#)

有了前一节的分析，我们来看一下具体的接口，它们都处于包 org.apache.hadoop.mapreduce 中。



上面的图中，类可以分为 4 种。右上角的是从 Writeable 继承的，和 Counter ( 还有 CounterGroup 和 Counters ，也在这个包中，并没有出现在上面的图里 ) 和 ID 相关的类，它们保持 MapReduce 过程中需要的一些计数器和标识；中间大部分是和 Context 相关的 \*Context 类，它为 Mapper 和 Reducer 提供了相关的上下文；关于 Map 和 Reduce ，对应的类是 Mapper , Reducer 和描述他们的 Job ( 在 Hadoop 中一次计算任务称之为一个 job ，下面的分析中，中文为 “作业” ，相应的 task 我们称为 “任务” ) ；图中其他类是配合 Mapper 和 Reduce 工作的一些辅助类。

如果你熟悉 `HTTPServlet`，那就能很轻松地理解 Hadoop 采用的结构，把整个 Hadoop 看作是容器，那么 Mapper 和 Reduce 就是容器里的组件，`*Context` 保存了组件的一些配置信息，同时也是和容器通信的机制。

和 ID 相关的类我们就不再讨论了。我们先看 `JobContext`，它位于 `*Context` 继承树的最上方，为 Job 提供一些只读的信息，如 Job 的 ID，名称等。下面的信息是 MapReduce 过程中一些较关键的定制信息：

( 来自 <http://www.ibm.com/developerworks/cn/opensource/os-cn-hadoop2/index.html> ) :

参数	作用	缺省值	其它实现
<b>InputFormat</b>	将输入的数据集切割成小数 <code>TextInputFormat</code> 据集 <code>InputSplits</code> , 每一个 (针对文本文件, 按行将文本文件切割成 <code>InputSplit</code> 将由一个 <code>InputSplits</code> , 并用 <code>Mapper</code> 负责处理。此外 <code>LineRecordReader</code> 将 <code>InputSplit</code> 解析成 <code>&lt;key,value&gt;</code> 对, <code>key</code> 是行在文件中的位置, <code>value</code> 是文件中的一行) 将一个 <code>InputSplit</code> 解析成 <code>&lt;key,value&gt;</code> 对提供给 <code>map</code> 函数。		<code>SequenceFileInputFormat</code>
<b>OutputFormat</b>	提供一个 <code>RecordWriter</code> 的 <code>TextOutputFormat</code> 实现, 负责输出最终结果 (用 <code>LineRecordWriter</code> 将最终结果写成纯文件文件, 每个 <code>&lt;key,value&gt;</code> 对一行, <code>key</code> 和 <code>value</code> 之间用 tab 分隔)		<code>SequenceFileOutputFormat</code>
<b>OutputKeyClass</b>	输出的最终结果中 <code>key</code> 的 <code>LongWritable</code> 类型		
<b>OutputValueClass</b>	输出的最终结果中 <code>value</code> <code>Text</code> 的类型		
<b>MapperClass</b>	Mapper 类, 实现 <code>map</code> 函数 <code>IdentityMapper</code> 数, 完成输入的 (将输入的 <code>&lt;key,value&gt;</code> 原封不动的 <code>LogRegexMapper</code> , <code>&lt;key,value&gt;</code> 到中间结果的输出为中间结果) 映射		<code>LongSumReducer</code> , <code>LogRegexMapper</code> , <code>InverseMapper</code>
<b>CombinerClass</b>	实现 <code>combine</code> 函数, 将中间结果中的重复 <code>key</code> 做合并 (不对中间结果中的重复 <code>key</code> 做合并) 并		
<b>ReducerClass</b>	Reducer 类, 实现 <code>reduce</code> <code>IdentityReducer</code> 函数, 对中间结果做合并, (将中间结果直接输出为最终结果)		<code>AccumulatingReducer</code> , <code>LongSumReducer</code>

---

形成最终结果

<b>InputPath</b>	设定 job 的输入目录, job null 运行时会处理输入目录下的 所有文件	
<b>OutputPath</b>	设定 job 的输出目录 , job null 的最终结果会写入输出目录 下	
<b>MapOutputKeyClass</b>	设定 map 函数输出的中间 如果 用户 没有 设定 的话 , 使用 结果中 key 的类型 OutputKeyClass	
<b>MapOutputValueClass</b>	设定 map 函数输出的中间 如果 用户 没有 设定 的话 , 使用 结果中 value 的类型 OutputValuesClass	
<b>OutputKeyComparator</b>	对结果中的 key 进行排序 WritableComparable 时的使用的比较器	
<b>PartitionerClass</b>	对中间结果的 key 排序后 , HashPartitioner 用此 Partition 函数将其划 (使用 Hash 函数做 partition) 分为 R 份 , 每份由一个 Reducer 负责处理。	KeyFieldBasedPartitioner PipesPartitioner

Job 继承自 JobContext , 提供了一系列的 set 方法 , 用于设置 Job 的一些属性 ( Job 更新属性 , JobContext 读属性 ) , 同时 , Job 还提供了一些对 Job 进行控制的方法 , 如下 :

mapProgress : map 的进度 ( 0—1.0 ) ;  
reduceProgress : reduce 的进度 ( 0—1.0 ) ;  
isComplete : 作业是否已经完成 ;  
isSuccessful : 作业是否成功 ;  
killJob : 结束一个在运行中的作业 ;  
getTaskCompletionEvents : 得到任务完成的应答 ( 成功/失败 ) ;  
killTask : 结束某一个任务 ;