

第 1 章 HDFS

HDFS (Hadoop 分布式文件系统) 是运行在通用硬件上的分布式文件系统。HDFS 提供了一个高度容错性和高吞吐量的海量数据存储解决方案。HDFS 已经在各种大型在线服务和大型存储系统中得到广泛应用, 已经成为各大网站等在线服务公司的海量存储事实标准, 多年来为网站客户提供了可靠、高效的服务。本章将对 HDFS 进行一个提纲挈领的介绍。

1.1 HDFS 概述

HDFS (Hadoop Distributed File System, Hadoop 分布式文件系统) 最开始是作为 Apache Nutch 搜索引擎项目的基础架构而开发的, 是 Apache Hadoop Core 项目的一部分。HDFS 被设计为可以运行在通用硬件 (commodity hardware) 上、提供流式数据操作、能够处理超大文件的分布式文件系统。HDFS 具有高度容错、高吞吐量、容易扩展、高可靠性等特征, 为大型数据集的处理提供了一个强有力的工具。

1.1.1 HDFS 体系结构

HDFS 是一个主/从 (Master/Slave) 体系结构的分布式系统, 如图 1-1 所示, HDFS 集群拥有一个 Namenode 和一些 Datanode, 用户可以通过 HDFS 客户端同 Namenode 和 Datanodes 交互以访问文件系统。

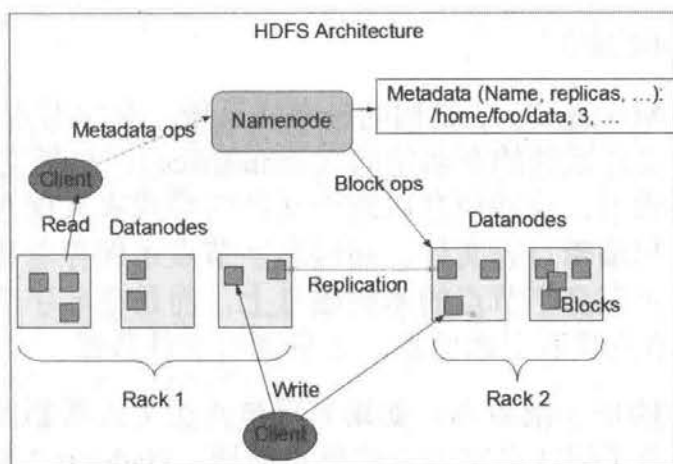


图 1-1 HDFS 体系结构示意图

在 HDFS 中，Namenode 是 HDFS 的 Master 节点，负责管理文件系统的命名空间(namespace)，以及数据块到具体 Datanode 节点的映射等信息。集群中的 Datanode 一般是一个节点一个，负责管理它所在节点上的存储。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组 Datanode 上，Datanode 会以本地文件的形式保存这些数据块以及数据块的校验信息。

用户能够通过 HDFS 客户端发起读写 HDFS 文件的请求，同时还能通过 HDFS 客户端执行文件系统的命名空间操作，比如打开、关闭、重命名文件或目录。Namenode 会响应这些请求，更改命名空间以及数据块的映射信息，然后指导 Datanode 处理文件 HDFS 客户端的读写请求。

1.1.2 HDFS 基本概念

了解了 HDFS 整体架构后，我们再介绍 HDFS 中几个比较重要的概念。

1. 数据块 (Block)

HDFS 中数据块的概念与大部分 Linux 文件系统(ext2、ext3)的数据块概念相同，HDFS 文件是以数据块的形式存储的，数据块是 HDFS 文件处理的最小单元。由于 HDFS 文件往往比较大，同时为了最小化寻址开销，所以 HDFS 数据块也更大，默认是 128MB。HDFS 数据块会以文件的形式存储在数据节点的磁盘上。

在 HDFS 中，所有文件都会被切分成若干个数据块分布在数据节点上存储。同时由于 HDFS 会将同一个数据块冗余备份保存到不同的数据节点上(一个数据块默认保存 3 份)，所以数据块的一个副本丢失了并不会影响这个数据块的访问。

在 HDFS 的读和写操作中，数据块都是最小单元。在读操作中，HDFS 客户端会首先到名字节点查找 HDFS 文件包含的数据块的位置信息，然后根据数据块的位置信息从数据节点读取数据。而在写操作中，HDFS 客户端也会首先从名字节点申请新的数据块，然后根据新申请数据块的位置信息建立数据流管道写数据。

2. 名字节点 (Namenode)

HDFS 是一个典型的 Master/Slave 结构的分布式系统，名字节点是 HDFS 主/从结构中的主节点。名字节点管理着文件系统的命名空间(namespace)，包括文件系统目录树、文件/目录信息以及文件的数据块索引，这些信息以两个文件的形式永久保存在名字节点的本地磁盘上，即命名空间镜像文件和编辑日志文件。同时名字节点还保存着数据块与数据节点的对应关系，这部分数据并不保存在名字节点的本地磁盘上，而是在名字节点启动时动态构建的。HDFS 客户端会通过名字节点获取上述信息，之后读写文件数据。

名字节点是 HDFS 中的单一故障点，如果名字节点丢失元数据或者损坏，文件系统将出现错误，甚至无法使用。为了解决名字节点的单点问题，Hadoop 2.X 版本引入了名字节点高可用性(HA)的支持。在 HA 实现中，同一个 HDFS 集群中会配置两个名字节点——活动名

字节点和备用名字节点。活动名字节点的内存元数据与备用名字节点是完全同步的，那么在活动名字节点发生故障而停止服务时，备用名字节点可以立即切换为活动状态，而不影响 HDFS 集群的服务。

名字节点的内存除了保存文件系统的命名空间外，还保存了文件系统中所有数据块与数据节点的对应关系，这意味着如果集群中文件数量过多时，名字节点的内存将成为限制系统横向扩展的瓶颈。为了解决这个问题，Hadoop 2.X 版本引入了联邦 HDFS 机制（HDFS Federation）。联邦 HDFS 机制允许添加名字节点以实现命名空间的扩展，其中每个名字节点都管理文件系统命名空间中的一部分，是一个独立的命名空间卷（namespace volume）。命名空间卷之间是相互独立的，两两之间并不相互通信，甚至其中一个名字节点失效了也不会影响由其他名字节点维护的命名空间的可用性。例如，一个名字节点可能管理/user 目录下的所有文件，而另一个名字节点可能管理/share 目录下的所有文件，这两个名字节点独立运行，互不影响。

HDFS 名字节点的相关内容我们会在第 3 章中介绍，包括 HDFS 元数据的管理，以及名字节点的 HA 机制等内容。而对于联邦 HDFS 机制，由于更改较多的是 Datanode 部分的代码，所以这部分内容我们会在第 4 章中介绍，请读者参考对应章节学习名字节点的实现。

3. 数据节点（Datanode）

数据节点是 HDFS 中的从节点，数据节点会根据 HDFS 客户端请求或者 Namenode 调度将新的数据块写入本地存储，或者读出本地存储上保存的数据块。

数据节点作为 HDFS 中的从节点，会不断地向名字节点发送心跳、数据块汇报以及缓存汇报，名字节点会通过心跳、数据块汇报以及缓存汇报的响应向数据节点发送指令，数据节点会执行这些指令，例如创建、删除或者复制数据等。

HDFS 数据节点的相关内容我们会在第 4 章中介绍。

4. 客户端

HDFS 提供了多种客户端接口供应用程序以及用户使用，包括命令行接口、浏览器接口以及代码 API 接口。用户通过这些接口可以很方便地使用 HDFS，而不需要考虑 HDFS 的实现细节。

而这些 HDFS 客户端接口的实现都是建立在 DFSCClient 类的基础上的，DFSCClient 类封装了客户端与 HDFS 其他节点间的复杂交互，我们会在第 5 章中介绍 HDFS 客户端的相关内容。

5. HDFS 通信协议

HDFS 作为一个分布式文件系统，它的某些流程是非常复杂的（例如读、写文件等典型流程），常常涉及数据节点、名字节点和客户端三者之间的配合、相互调用才能实现。为了降低节点间代码的耦合性，提高单个节点代码的内聚性，HDFS 将这些节点间的调用抽象成不同的接口。

HDFS 节点间的接口主要有两种类型。

- Hadoop RPC 接口：HDFS 中基于 Hadoop RPC 框架实现的接口。
- 流式接口：HDFS 中基于 TCP 或者 HTTP 实现的接口。

由于 HDFS 通信协议部分内容比较多，我们会单独在本章的 HDFS 通信协议小节中介绍这两部分内容。

1.2 HDFS 通信协议

HDFS 通信协议抽象了 HDFS 各个节点之间的调用接口。由于本书后续章节只涉及各个节点的具体实现，所以这里将 HDFS 通信协议作为一个小节放在本章，做一个整体性的介绍。如果读者在后续章节中遇到通信协议相关的内容，可以查询本节的介绍。

1.2.1 Hadoop RPC 接口

Hadoop RPC 调用使得 HDFS 进程能够像本地调用一样调用另一个进程中的方法，并且可以传递 Java 基本类型或者自定义类作为参数，同时接收返回值。如果远程进程在调用过程中出现异常，本地进程也会收到对应的异常。目前 Hadoop RPC 调用是基于 Protobuf 实现的，我们会在第 2 章中介绍底层的具体实现，本节主要介绍 Hadoop RPC 接口的定义。Hadoop RPC 接口主要定义在 `org.apache.hadoop.hdfs.protocol` 包和 `org.apache.hadoop.hdfs.server.protocol` 包中，包括以下几个接口。

- **ClientProtocol**: `ClientProtocol` 定义了客户端与名字节点间的接口，这个接口定义的方法非常多，客户端对文件系统的所有操作都需要通过这个接口，同时客户端读、写文件等操作也需要先通过这个接口与 `Namenode` 协商之后，再进行数据块的读出和写入操作。
- **ClientDatanodeProtocol**: 客户端与数据节点间的接口。`ClientDatanodeProtocol` 中定义的方法主要是用于客户端获取数据节点信息时调用，而真正的数据读写交互则是通过流式接口进行的。
- **DatanodeProtocol**: 数据节点通过这个接口与名字节点通信，同时名字节点会通过这个接口中方法的返回值向数据节点下发指令。注意，这是名字节点与数据节点通信的唯一方式。这个接口非常重要，数据节点会通过这个接口向名字节点注册、汇报数据块的全量以及增量的存储情况。同时，名字节点也会通过这个接口中方法的返回值，将名字节点指令带回该数据块，根据这些指令，数据节点会执行数据块的复制、删除以及恢复操作。
- **InterDatanodeProtocol**: 数据节点与数据节点间的接口，数据节点会通过这个接口和其他数据节点通信。这个接口主要用于数据块的恢复操作，以及同步数据节点上存储的数据块副本的信息。
- **NamenodeProtocol**: 第二名字节点与名字节点间的接口。由于 Hadoop 2.X 中引入了

HA 机制，检查点操作也不再由第二名字节点执行了，所以 NamenodeProtocol 我们就不详细介绍了。

- 其他接口：主要包括安全相关接口（RefreshAuthorizationPolicyProtocol、RefreshUserMappingsProtocol）、HA 相关接口（HAResourceProtocol）等。HA 相关接口的实现和定义我们将在第3章的 HA 小节中介绍。

下面我们重点介绍 ClientProtocol、ClientDatanodeProtocol、DatanodeProtocol、InterDatanodeProtocol 和 NamenodeProtocol 等接口的定义。

1. ClientProtocol

ClientProtocol 定义了所有由客户端发起的、由 Namenode 响应的操作。这个接口非常大，有 80 多个方法，我们把这个接口中的方法分为如下几类。

- HDFS 文件读相关的操作。
- HDFS 文件写以及追加写的相关操作。
- 管理 HDFS 命名空间（namespace）的相关操作。
- 系统问题与管理相关的操作。
- 快照相关的操作。
- 缓存相关的操作。
- 其他操作。

HDFS 文件读操作、HDFS 文件写与追加写操作，以及命名空间的管理操作，这三个部分都可以在 FileSystem 类中找到对应的方法，这些方法都是用来支持 Hadoop 文件系统实现的。对于系统问题与管理相关的操作，则是由 DFSAdmin 这个工具类发起的，其中的方法是用于支持管理员配置和管理 HDFS 的。而快照和缓存则都是 Hadoop2.X 中引入的新特性，ClientProtocol 中也有对应的方法用于支持这两个新特性。当然，ClientProtocol 中还包括安全、XAttr 等方法，这部分不是重点，我们就不再详细介绍了。

（1）读数据相关方法

ClientProtocol 中与客户端读取文件相关的方法主要有两个：getBlockLocations() 和 reportBadBlocks()。

客户端会调用 ClientProtocol.getBlockLocations() 方法获取 HDFS 文件指定范围内所有数据块的位置信息。这个方法参数是 HDFS 文件的文件名以及读取范围，返回值是文件指定范围内所有数据块的文件名以及它们的位置信息，使用 LocatedBlocks 对象封装。每个数据块的位置信息指的是存储这个数据块副本的所有 Datanode 的信息，这些 Datanode 会以与当前客户端的距离远近排序。客户端读取数据时，会首先调用 getBlockLocations() 方法获取 HDFS 文件的所有数据块的位置信息，然后客户端会根据这些位置信息从数据节点读取数据块。ClientProtocol.getBlockLocations() 方法的定义如下：

```
public LocatedBlocks getBlockLocations(String src,
                                       long offset,
                                       long length)
```

```
throws AccessControlException, FileNotFoundException,  
UnresolvedLinkException, IOException;
```

客户端会调用 `ClientProtocol.reportBadBlocks()` 方法向 Namenode 汇报错误的数据块。当客户端从数据节点读取数据块且发现数据块的校验和并不正确时，就会调用这个方法向 Namenode 汇报这个错误的数据块信息。`ClientProtocol.reportBadBlocks()` 方法的定义如下：

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

(2) 写/追加写数据相关方法

在 HDFS 客户端操作中最重要的一部分就是写入一个新的 HDFS 文件，或者打开一个已有的 HDFS 文件并执行追加写操作。`ClientProtocol` 中定义了 8 个方法支持 HDFS 文件的写操作：`create()`、`append()`、`addBlock()`、`complete()`、`abandonBlock()`、`getAdditionalDataNodes()`、`updateBlockForPipeline()` 和 `updatePipeline()`。

`create()` 方法用于在 HDFS 的文件系统目录树中创建一个新的空文件，创建的路径由 `src` 参数指定。这个空文件创建后对于其他的客户端是“可读”的，但是这些客户端不能删除、重命名或者移动这个文件，直到这个文件被关闭或者租约过期。客户端写一个新的文件时，会首先调用 `create()` 方法在文件系统目录树中创建一个空文件，然后调用 `addBlock()` 方法获取存储文件数据的数据块的位置信息，最后客户端就可以根据位置信息建立数据流管道，向数据节点写入数据了。`create()` 方法的定义如下：

```
public HdfsFileStatus create(String src, FsPermission masked,  
    String clientName, EnumSetWritable<CreateFlag> flag,  
    boolean createParent, short replication, long blockSize,  
    CryptoProtocolVersion[] supportedVersions)  
    throws..., IOException;
```

`append()` 方法用于打开一个已有的文件，如果这个文件的最后一个数据块没有写满，则返回这个数据块的位置信息（使用 `LocatedBlock` 对象封装）；如果这个文件的最后一个数据块正好写满，则创建一个新的数据块并添加到这个文件中，然后返回这个新添加的数据块的位置信息。客户端追加写一个已有文件时，会先调用 `append()` 方法获取最后一个可写数据块的位置信息，然后建立数据流管道，并向数据节点写入追加的数据。如果客户端将这个数据块写满，与 `create()` 方法一样，客户端会调用 `addBlock()` 方法获取新的数据块。

```
public LocatedBlock append(String src, String clientName)  
    throws AccessControlException, DSQuotaExceededException,  
    FileNotFoundException, SafeModeException, UnresolvedLinkException,  
    SnapshotAccessControlException, IOException;
```

客户端调用 `addBlock()` 方法向指定文件添加一个新的数据块，并获取存储这个数据块副本的所有数据节点的位置信息（使用 `LocatedBlock` 对象封装）。要特别注意的是，调用 `addBlock()` 方法时还要传入上一个数据块的引用。`Namenode` 在分配新的数据块时，会顺便提交上一个数据块，这里 `previous` 参数就是上一个数据块的引用。`excludeNodes` 参数则是数据节点的黑名单，保存了客户端无法连接的一些数据节点，建议 `Namenode` 在分配保存数据块副本的数据节点时不要考虑这些节点。`favorableNodes` 参数则是客户端所希望的保存数据块副本的数据节点。

点的列表。客户端调用 `addBlock()` 方法获取新的数据块的位置信息后，会建立到这些数据节点的数据流管道，并通过数据流管道将数据写入数据节点。`addBlock()` 方法的定义如下：

```
public LocatedBlock addBlock(String src, String clientName,
    ExtendedBlock previous, DatanodeInfo[] excludeNodes, long fileId,
    String[] favoredNodes)
    throws AccessControlException, FileNotFoundException,
    NotReplicatedYetException, SafeModeException, UnresolvedLinkException,
    IOException;
```

当客户端完成了整个文件的写入操作后，会调用 `complete()` 方法通知 Namenode。这个操作会提交新写入 HDFS 文件的所有数据块，当这些数据块的副本数量满足系统配置的最小副本系数（默认值为 1），也就是该文件的所有数据块至少有一个有效副本时，`complete()` 方法会返回 `true`，这时 Namenode 中文件的状态也会从构建中状态转换为正常状态；否则，`complete()` 会返回 `false`，客户端就需要重复调用 `complete()` 操作，直至该方法返回 `true`。

```
public boolean complete(String src, String clientName,
    ExtendedBlock last, long fileId)
    throws AccessControlException, FileNotFoundException, SafeModeException,
    UnresolvedLinkException, IOException;
```

上面描述的 5 个方法都是在正常流程时，客户端写文件必须调用的方法。但是对于一个分布式系统来说，写流程中涉及的任何一个节点都有可能出现故障。出现故障的情况也是需要考虑在内的，所以 `ClientProtocol` 定义了 `abandonBlock()`、`getAdditionalDatanode()`、`updateBlockForPipeline()` 以及 `updatePipeline()` 等方法，用于在异常情况下进行恢复操作。

客户端调用 `abandonBlock()` 方法放弃一个新申请的数据块。考虑下面这种情况：当客户端获取了一个新申请的数据块，发现无法建立到存储这个数据块副本的某些数据节点的连接时，会调用 `abandonBlock()` 方法通知名字节点放弃这个数据块，之后客户端会再次调用 `addBlock()` 方法获取新的数据块，并在传入参数时将无法连接的数据节点放入 `excludeNodes` 参数列表中，以避免 Namenode 将数据块的副本分配到该节点上，造成客户端再次无法连接这个节点的情况。

```
public void abandonBlock(ExtendedBlock b, long fileId,
    String src, String holder)
    throws AccessControlException, FileNotFoundException,
    UnresolvedLinkException, IOException;
```

`abandonBlock()` 方法用于处理客户端建立数据流管道时数据节点出现故障的情况。那么，如果客户端已经成功建立了数据流管道，在客户端写某个数据块时，存储这个数据块副本的某个数据节点出现了错误该如何处理呢？这个操作就比较复杂了，客户端首先会调用 `getAdditionalDatanode()` 方法向 Namenode 申请一个新的 Datanode 来替代出现故障的 Datanode。然后客户端会调用 `updateBlockForPipeline()` 方法向 Namenode 申请为这个数据块分配新的时间戳，这样故障节点上的没能写完整的数据块的时间戳就会过期，在后续的块汇报操作中会被删除。最后客户端就可以使用新的时间戳建立新的数据流管道，来执行对数据块的写操作了。数据流管道建立成功后，客户端还需要调用 `updatePipeline()` 方法更新 Namenode 中当前数据块的数据流管道信息。至此，一个完整的恢复操作结束。

上面我们描述的都是写数据操作时数据节点发生故障的情况，包括了数据流管道建立时以及建立后数据节点发生故障的情况。在写数据的过程中，Client 节点也有可能在任意时刻发生故障，为了预防这种情况，对于任意一个 Client 打开的文件都需要 Client 定期调用 `ClientProtocol.renewLease()` 方法更新租约（关于租约请参考第 3 章中租约相关小节）。如果 Namenode 长时间没有收到 Client 的租约更新消息，就会认为 Client 发生故障，这时就会触发一次租约恢复操作，关闭文件并且同步所有数据节点上这个文件数据块的状态，确保 HDFS 系统中这个文件是正确且一致保存的。

如果在写操作时，名字节点发生故障该如何处理呢？这就要涉及 HDFS 的 HA 架构了，请读者参考第 3 章的 HA 部分。

（3）命名空间管理的相关方法

`ClientProtocol` 中有很重要的一部分操作是对 Namenode 命名空间的修改。我们知道 `FileSystem` 类也定义了对文件系统命名空间修改操作的 API（`FileSystem` 类抽象了一个文件系统对外提供的 API 接口），HDFS 则满足 `FileSystem` 类抽象的所有方法。表 1-1 总结了 `FileSystem` API 与 `ClientProtocol` 接口的对应关系。

表 1-1 `FileSystem` API 与 `ClientProtocol` 接口的对应关系

Hadoop FileSystem 操作	ClientProtocol 对应的接口	描 述
<code>FileSystem.rename2()</code>	<code>rename2()</code>	更改文件/目录名称
<code>FileSystem.concat()</code>	<code>concat()</code>	将两个已有文件拼接成一个
<code>FileSystem.delete()</code>	<code>delete()</code>	从文件系统中删除指定文件或者目录
<code>FileSystem.mkdirs()</code>	<code>mkdirs()</code>	以指定名称和权限在文件系统中创建目录
<code>FileSystem.listStatus()</code>	<code>getListing()</code>	读取一个指定目录下的所有项目
<code>FileSystem.set* ()</code>	<code>setPermission()</code> <code>setOwner()</code> <code>setTimes()</code> <code>setReplication()</code>	修改文件属性。分别用于修改文件权限、文件主/组、文件修改时间/访问时间以及文件的副本系数
<code>FileSystem.listCorruptFileBlocks()</code>	<code>listCorruptFileBlocks()</code>	获取文件系统中损坏文件的一部分，如果想要获取文件系统中所有损坏的文件，则循环调用这个方法
<code>FileSystem.getFileStatus</code>	<code>getFileInfo()</code>	获取文件/目录的属性
<code>FileSystem.getFileLinkStatus</code>	<code>getFileLinkStatus()</code>	获取文件/目录的属性，如果文件指向一个符号链接，则返回这个符号链接的信息
<code>DistributedFileSystem.isFileClosed</code>	<code>isFileClosed()</code>	判断指定文件是否关闭了
<code>FileSystem.getContentSummary</code>	<code>getContentSummary()</code>	获取文件/目录使用的存储空间信息
<code>FileSystem.createSymlink()</code>	<code>createSymlink()</code>	对于已经存在的文件创建符号链接
<code>resolveLink()</code>	<code>getLinkTarget()</code>	获取指定符号链接指向目标

通过表 1-1 我们可以看出，`ClientProtocol` 中涉及的命名空间管理的方法都有与之对应的 HDFS 文件系统 API，且方法的名称和参数很相近。在这里我们以 `setReplication()` 为例，

setReplication()方法在 ClientProtocol 中的定义是:

```
public boolean setReplication(String src, short replication)
    throws AccessControlException, DSQuotaExceededException,
    FileNotFoundException, SafeModeException, UnresolvedLinkException,
    SnapshotAccessControlException, IOException;
```

在 FileSystem 接口中的定义是:

```
public boolean setReplication(Path src, short replication)
    throws IOException {
    return true;
}
```

可以看到, setReplication()方法在 FileSystem 和 ClientProtocol 中定义的方法名及参数都很接近, 大部分情况下 ClientProtocol 接口方法定义的参数更多, 可以很好地支持 FileSystemAPI 定义的操作。

(4) 系统问题与管理操作

ClientProtocol 中另一个重要的部分就是支持 DFSAdmin 工具的接口方法, DFSAdmin 是供 HDFS 管理员管理 HDFS 集群的命令行工具。一个典型的 dfsadmin 命令如下所示, 管理员可以添加不同的参数以触发 HDFS 进行相应的操作。

```
hdfs dfsadmin[参数]
```

表 1-2 给出了 ClientProtocol 中定义的接口方法与 dfsadmin 命令参数之间的对应关系, 我们会重点讲解几个比较重要的方法。

表 1-2 ClientProtocol 中定义的接口方法与 dfsadmin 命令参数之间的对应关系

ClientProtocol 接口	dfsadmin 命令参数
getStatus()	用于获取文件系统状态信息, 包括磁盘使用情况、复制数据块的数量、损坏数据块的数量、丢失数据块的数量等。对应于 dfsadmin 命令`-report`选项
getDatanodeReport()	获取集群中存活的、死亡的或者所有的数据节点信息。对应于 dfsadmin 命令`-report`选项
getDatanodeStorageReport()	获取数据节点上所有存储的信息
setSafeMode()	用于进入、离开安全模式, 或者获取当前安全模式的状态。这个方法我们会在第 3 章的安全模式小节中详细介绍。对应于 dfsadmin 命令`-safemode`选项
saveNamespace()	将 Namenode 内存中的命名空间保存至新的 fsimage 中, 并且重置 editlog。对应于 dfsadmin 命令`-saveNamespace`选项。注意, 执行这个操作要求必须是处于安全模式中
rollEdits()	重置 editlog, 也就是关闭当前正在写入的 editlog 文件, 开启一个新的 editlog 文件。对应于 dfsadmin 命令`-rollEdits`选项。注意, 执行这个操作要求必须是处于安全模式中
restoreFailedStorage()	用于当失败的(failed)存储变得可用时, 设置是否对这个存储上保存的副本进行恢复操作。对应于 dfsadmin 命令`-restoreFailedStorage`选项
refreshNodes()	触发 Namenode 重新读取 include/exclude 文件。对应于 dfsadmin 命令`-refreshNodes`选项
finalizeUpgrade()	提交 Namenode 的升级操作。对应于 dfsadmin 命令`-finalizeUpgrade`选项
rollingUpgrade()	触发 Namenode 进行升级操作。对应于 dfsadmin 命令`-rollingUpgrade`选项

续表

ClientProtocol 接口	dfsadmin 命令参数
metaSave()	将 Namenode 中主要的数据结构保存到指定文件中, 包括同 Namenode 心跳过的 Datanode、等待复制的数据块、等待删除的数据块、当前正在复制的数据块等信息。对应于 dfsadmin 命令`-metasave`选项
setBalancerBandwidth()	更改 Datanode 在进行数据块平衡操作时所占用的带宽。调用这个命令设置的带宽值会覆盖 dfs.balance.bandwidthPerSec 配置项配置的带宽值。对应于 dfsadmin 命令`-setBalancerBandwidth`选项
setQuota()	设置目录中的文件/目录的数量配额, 以及文件大小的配额。对应于 dfsadmin 命令`-setQuota`、`-clrQuota`、`-setSpaceQuota`和`-clrSpaceQuota`选项, 这4个选项底层都是通过 setQuota()触发 Namenode 操作的

首先看一下 setSafeMode()方法, 这里涉及一个非常重要的概念——安全模式。安全模式是 Namenode 的一种状态, 处于安全模式中的 Namenode 不接受客户端对命名空间的修改操作, 整个命名空间都处于只读状态。同时, Namenode 也不会向 Datanode 下发任何数据块的复制、删除指令。管理员可以通过 dfsadmin setSafemode 命令触发 Namenode 进入或者退出安全模式, 同时还可以使用这个命令查询安全模式的状态。需要注意的是, 刚刚启动的 Namenode 会直接自动进入安全模式, 当 Namenode 中保存的满足最小副本系数的数据块达到一定的比例时, Namenode 会自动退出安全模式。而对于用户通过 dfsAdmin 方式触发 Namenode 进入安全模式的情况, 则只能由管理员手动关闭安全模式, Namenode 不可以自动退出。dfsadmin setSafemode 命令正是通过调用 ClientProtocol.setSafeMode()方法实现的。setSafeMode()方法的定义如下:

```
public boolean setSafeMode(HdfsConstants.SafeModeAction action, boolean isChecked)
    throws IOException;
```

了解了安全模式之后, 我们来看看必须在安全模式中才能进行的两个操作。`-saveNamespace`用于将整个命名空间保存到新的 fsimage 文件中, 并且重置 editlog 文件; 而`-rollEdits`则会触发重置 editlog 文件的操作, 关闭当前正在写入的 editlog 文件, 开启一个新的 editlog 文件 (fsimage 与 editlog 文件请参考第3章的 fsimage 小节)。

refreshNodes()方法会触发 Namenode 刷新数据节点列表。管理员可以通过 include 文件指定可以连接到 Namenode 的数据节点列表, 通过 exclude 文件指定不能连接到 Namenode 的数据节点列表。每当管理员修改了这两个配置文件后, 都需要通过`-refreshNodes`选项触发 Namenode 刷新数据节点列表, 这个操作会造成 Namenode 从文件系统中移除已有的数据节点, 或者添加新的数据节点 (请参考第3章的数据节点管理小节)。

finalizeUpgrade()和 rollingUpgrade()操作都是与 Namenode 升级相关的, 管理员可以通过`-rollingUpgrade`选项触发 Namenode 进行升级操作。当 Namenode 成功地执行了升级操作后, 管理员可以通过`-finalizeUpgrade`提交升级操作, 提交升级操作会删除升级操作创建的一些临时目录, 提交升级操作之后就不可以再回滚了 (请参考第4章的 Storage 小节)。

对于其他方法, 请读者参考表 1-2 中的说明, 这里不再详细介绍了。

(5) 快照相关操作

Hadoop 2.X 添加了新的快照特性，用户可以为 HDFS 的任意路径创建快照。快照保存了一个时间点上 HDFS 某个路径中所有数据的拷贝，快照可以将失效的集群回滚到之前一个正常的时间点上。用户可以通过 `hdfs dfs` 命令执行创建、删除以及重命名快照等操作，ClientProtocol 也定义了对应的方法来支持快照命令。

需要特别注意的是，在创建快照之前，必须先通过 `hdfs dfsadmin -allowSnapshot` 命令开启目录的快照功能，否则不可以在该目录上创建快照。表 1-3 给出了快照操作与 ClientProtocol 中相关方法的对应关系，请读者参考（快照相关内容我们会在第 3 章的快照小节中介绍）。

表 1-3 快照操作与 ClientProtocol 中相关方法的对应关系

方法名	作用	对应的命令
createSnapshot()	创建快照	`hdfs dfs -createSnapshot`
deleteSnapshot()	删除快照	`hdfs dfs -deleteSnapshot`
renameSnapshot()	重命名快照	`hdfs dfs -renameSnapshot <path><oldName> <newName>`
allowSnapshot()	开启指定目录的快照功能。一个目录必须在开启快照功能之后才可以添加快照	`hdfs dfsadmin -allowSnapshot <path>`
disallowSnapshot()	关闭指定目录的快照功能	`hdfs dfs -deleteSnapshot <path><snapshotName>`
getSnapshotDiffReport()	获取两个快照间的不同	`hdfs snapshotDiff <path><fromSnapshot> <toSnapshot>`

(6) 缓存相关操作

HDFS 2.3 版本添加了集中式缓存管理（HDFS Centralized Cache Management）功能。用户可以指定一些经常被使用的数据或者高优先级任务对应的数据，让它们常驻内存而不被淘汰到磁盘上，这对于提升 Hadoop 系统和上层应用的执行效率与实时性有很大的帮助。

这里涉及两个概念。

- cache directive: 表示要被缓存到内存的文件或者目录。
- cache pool: 用于管理一系列的 cache directive，类似于命名空间。同时使用 UNIX 风格的文件读、写、执行权限管理机制。

表 1-4 总结了缓存相关命令与 ClientProtocol 方法之间的对应关系，请读者参考（缓存相关内容我们会在第 3 章的缓存管理小节中介绍）。

表 1-4 缓存相关命令与 ClientProtocol 方法之间的对应关系

方法名	作用	对应的命令
addCacheDirective()	添加一个缓存	`hdfs cacheadmin -addDirective -path <path> -pool <pool-name> [-force] [-replication <replication>] [-ttl <time-to-live>]`

续表

方法名	作 用	对应的命令
modifyCacheDirective()	修改缓存	-modifyDirective
removeCacheDirective()	删除缓存	`hdfs cacheadmin -removeDirective <id>`
listCacheDirectives()	列出指定路径下的所有缓存	`hdfs cacheadmin -listDirectives [-stats] [-path <path>] [-pool <pool>]`
addCachePool()	添加一个缓存池	`hdfs cacheadmin -addPool <name> [-owner <owner>] [-group <group>] [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>`
modifyCachePool()	修改已有缓存池的元数据	`hdfs cacheadmin -modifyPool <name> [-owner <owner>] [-group <group>] [-mode <mode>] [-limit <limit>] [-maxTtl <maxTtl>]`
removeCachePool()	删除缓存池	`hdfs cacheadmin -removePool <name>`
listCachePools()	列出已有缓存池的信息，包括用户名、用户组、权限等	`hdfs cacheadmin -listPools [-stats] [<name>]`

(7) 其他操作

安全相关以及 XAttr 相关命令，主要都是增加、删除以及 List 操作，这里就不再详细介绍了

2. ClientDatanodeProtocol

ClientDatanodeProtocol 定义了 Client 与 Datanode 之间的接口。相比 ClientProtocol，ClientDatanodeProtocol 的定义简单很多，如图 1-2 所示。

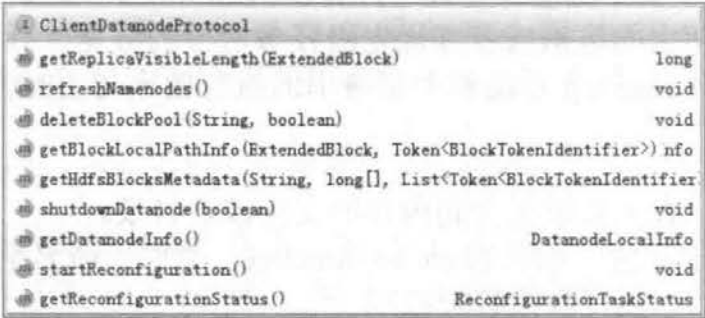


图 1-2 ClientDatanodeProtocol 定义

ClientDatanodeProtocol 中定义的接口可以分为两部分：一部分是支持 HDFS 文件读取操作的，例如 getReplicaVisibleLength()以及 getBlockLocalPathInfo(); 另一部分是支持 DFSAdmin 中与数据节点管理相关的命令。下面我们就看一下图 1-2 中所示 9 个方法的具体定义。

(1) getReplicaVisibleLength()

客户端会调用 getReplicaVisibleLength()方法从数据节点获取某个数据块副本真实的数据长度。当客户端读取一个 HDFS 文件时，需要获取这个文件对应的所有数据块的长度，用于

建立数据块的输入流，然后读取数据。但是 Namenode 元数据中文件的最后一个数据块长度与 Datanode 实际存储的可能不一致，所以客户端在创建输入流时就需要调用 `getReplicaVisibleLength()` 方法从 Datanode 获取这个数据块的真实长度。

(2) `getBlockLocalPathInfo()`

HDFS 对于本地读取，也就是 Client 和保存该数据块的 Datanode 在同一台物理机器上时，是有很多优化的。Client 会调用 `ClientProtocol.getBlockLocalPathInfo()` 方法获取指定数据块文件以及数据块校验文件在当前节点上的本地路径，然后利用这个本地路径执行本地读取操作，而不是通过流式接口执行远程读取，这样也就大大优化了读取的性能。

在 HDFS 2.6 版本中，客户端会通过调用 `DataTransferProtocol` 接口从数据节点获取数据块文件的文件描述符，然后打开并读取文件以实现短路读操作，而不是通过 `ClientDatanodeProtocol` 接口。客户端的短路读操作请参考第 5 章的文件短路读操作小节。

(3) `refreshNamenodes()`

在用户管理员命令中有一个 ``hdfs dfsadmin datanodehost:port`` 命令，用于触发指定的 Datanode 重新加载配置文件，停止服务那些已经从配置文件中删除的块池 (blockPool)，开始服务新添加的块池。块池的概念请参考第 4 章的 Datanode 逻辑结构小节。

这条命令底层就是由 `ClientDatanodeProtocol.refreshNamenodes()` 方法实现的，客户端会通过这个接口触发对应的 Datanode 执行操作。

(4) `deleteBlockPool()`

在用户管理员命令中还有一个与块池管理相关的 ``hdfs dfsadmin-deleteBlockPool datanode-host:port blockpoolId [force]`` 命令，用于从指定 Datanode 删除 `blockpoolId` 对应的块池，如果 `force` 参数被设置了，那么无论这个块池目录中有没有数据都会被强制删除；否则，只有这个块池目录为空的情况下才会被删除。需要注意的是，如果 Datanode 还在服务这个块池，这个命令的执行将会失败。要停止一个数据节点服务指定的块池，需要调用上面提到的 `refreshNamenodes()` 方法。

`deleteBlockPool()` 方法有两个参数，其中 `blockpoolId` 用于设置要被删除的块池 ID；`force` 用于设置是否强制删除。

```
void deleteBlockPool(String bpid, boolean force) throws IOException;
```

(5) `getHdfsBlocksMetadata()`

`getHdfsBlocksMetadata()` 方法主要用于获取数据块是存储在指定 Datanode 的哪个卷 (volume) 上的，这个方法主要是为了支持 `DistributedFileSystem.getFileBlockStorageLocations()` 方法。关于卷 (volume) 的定义请参考第 4 章。

(6) `shutdownDatanode()`

`shutdownDatanode()` 方法用于关闭一个数据节点，这个方法主要是为了支持管理命令 ``hdfs`

`dfsadmin-shutdownDatanode <datanode_host:ipc_port> [upgrade]`。

(7) `getDatanodeInfo()`

`getDatanodeInfo()`方法用于获取指定 Datanode 的信息，这里的信息包括 Datanode 运行的 HDFS 版本、Datanode 配置的 HDFS 版本，以及 Datanode 的启动时间。对应于管理命令 `'hdfs dfsadmin-getDatanodeInfo'`。

(8) `startReconfiguration()`

`startReconfiguration()`方法用于触发 Datanode 异步地从磁盘重新加载配置，并且应用该配置。这个方法用于支持管理命令 `'hdfs dfsadmin-getDatanodeInfo-reconfigstart'`。

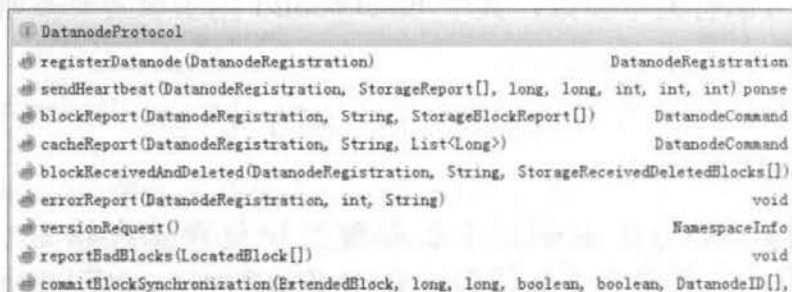
(9) `getReconfigurationStatus()`

我们知道 `startReconfiguration()`方法是异步地加载配置操作，所以 HDFS 提供了 `getReconfigurationStatus()`方法用于查询上一次触发的重新加载配置操作的运行情况。对应于管理命令 `'hdfs dfsadmin-getDatanodeInfo-reconfigstartstatus'`。我们可以看到 `getReconfigurationStatus()`方法和 `startReconfiguration()`方法对应的管理命令是一样的，只不过参数不同，一个是 `start`，一个是 `status`。

3. DatanodeProtocol

`ClientProtocol` 和 `DatanodeProtocol` 都是由客户端发起调用的接口，下面我们介绍服务器间的接口。`DatanodeProtocol` 是 Datanode 与 Namenode 间的接口，Datanode 会使用这个接口与 Namenode 握手、注册、发送心跳、进行全量以及增量的数据块汇报。Namenode 会在 Datanode 的心跳响应中携带名字节点指令，Datanode 收到名字节点指令之后会执行对应的操作。要特别注意的是，Namenode 向 Datanode 下发名字节点指令是没有任何其他接口的，只会通过 `DatanodeProtocol` 的返回值来下发命令。

`DatanodeProtocol` 定义的方法如图 1-3 所示，我们可以将 `DatanodeProtocol` 定义的方法分为三种类型：Datanode 启动相关、心跳相关以及数据块读写相关。下面将会分别介绍这三种类型的方法，以及 Namenode 向 Datanode 下发的指令。



DatanodeProtocol	
<code>registerDatanode(DatanodeRegistration)</code>	<code>DatanodeRegistration</code>
<code>sendHeartbeat(DatanodeRegistration, StorageReport[], long, long, int, int, int)</code>	<code>pong</code>
<code>blockReport(DatanodeRegistration, String, StorageBlockReport[])</code>	<code>DatanodeCommand</code>
<code>cacheReport(DatanodeRegistration, String, List<Long>)</code>	<code>DatanodeCommand</code>
<code>blockReceivedAndDeleted(DatanodeRegistration, String, StorageReceivedDeletedBlocks[])</code>	
<code>errorReport(DatanodeRegistration, int, String)</code>	<code>void</code>
<code>versionRequest()</code>	<code>NamespaceInfo</code>
<code>reportBadBlocks(LocatedBlock[])</code>	<code>void</code>
<code>commitBlockSynchronization(ExtendedBlock, long, long, boolean, boolean, DatanodeID[])</code>	

图 1-3 DatanodeProtocol 定义

(1) Datanode 启动相关方法

一个完整的 Datanode 启动操作会与 Namenode 进行 4 次交互，也就是调用 4 次

DatanodeProtocol 定义的方法。首先调用 `versionRequest()` 与 Namenode 进行握手操作，然后调用 `registerDatanode()` 向 Namenode 注册当前的 Datanode，接着调用 `blockReport()` 汇报 Datanode 上存储的所有数据块，最后调用 `cacheReport()` 汇报 Datanode 缓存的所有数据块。

我们首先看一下 `versionRequest()` 方法。Datanode 启动时会首先调用 `versionRequest()` 方法与 Namenode 进行握手。这个方法的返回值是一个 `NamespaceInfo` 对象，`NamespaceInfo` 对象会封装当前 HDFS 集群的命名空间信息，包括存储系统的布局版本号 (`layoutversion`)、当前的命名空间的 ID (`namespaceId`)、集群 ID (`clusterId`)、文件系统的创建时间 (`ctime`)、构建时的 HDFS 版本号 (`buildVersion`)、块池 ID (`blockpoolId`)、当前的软件版本号 (`softwareVersion`) 等。Datanode 获取到 `NamespaceInfo` 对象后，就会比较 Datanode 当前的 HDFS 版本号和 Namenode 的 HDFS 版本号，如果 Datanode 版本与 Namenode 版本不能协同工作，则抛出异常，Datanode 也就无法注册到该 Namenode 上。如果当前 Datanode 上已经有了文件存储的目录，那么 Datanode 还会检查 Datanode 存储上的块池 ID、文件系统 ID 以及集群 ID 与 Namenode 返回的是否一致。

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

成功进行握手操作后，Datanode 会调用 `ClientProtocol.registerDatanode()` 方法向 Namenode 注册当前的 Datanode，这个方法的参数是一个 `DatanodeRegistration` 对象，它封装了 `DatanodeID`、Datanode 的存储系统的布局版本号 (`layoutversion`)、当前命名空间的 ID (`namespaceId`)、集群 ID (`clusterId`)、文件系统的创建时间 (`ctime`) 以及 Datanode 当前的软件版本号 (`softwareVersion`)。名字节点会判断 Datanode 的软件版本号与 Namenode 的软件版本号是否兼容，如果兼容则进行注册操作，并返回一个 `DatanodeRegistration` 对象供 Datanode 后续处理逻辑使用。

```
public DatanodeRegistration registerDatanode(DatanodeRegistration registration
) throws IOException;
```

Datanode 成功向 Namenode 注册之后，Datanode 会通过调用 `DatanodeProtocol.blockReport()` 方法向 Namenode 上报它管理的所有数据块的信息。这个方法需要三个参数：`DatanodeRegistration` 用于标识当前的 Datanode；`poolId` 用于标识数据块所在的块池 ID；`reports` 是一个 `StorageBlockReport` 对象的数组，每个 `StorageBlockReport` 对象都用于记录 Datanode 上一个存储空间存储的数据块。这里需要特别注意的是，上报的数据块是以长整型数组保存的，每个已经提交的数据块 (`finalized`) 以 3 个长整型来表示，每个构建中的数据块 (`under-construction`) 以 4 个长整型来表示。之所以不使用 `ExtendedBlock` 对象保存上报的数据块，是因为这样可以减少 `blockReport()` 操作所使用的内存，Namenode 接收到消息时，不需要创建大量的 `ExtendedBlock` 对象，只需要不断地从长整型数组中提取数据块即可。

```
public DatanodeCommand blockReport(DatanodeRegistration registration,
String poolId, StorageBlockReport[] reports) throws IOException;
```

Namenode 接收到 `blockReport()` 请求之后，会根据 Datanode 上报的数据块存储情况建立数据块与数据节点之间的对应关系。同时，Namenode 会在 `blockReport()` 的响应中携带名字节点指令，通知数据节点进行重新注册、发送心跳、备份或者删除 Datanode 本地磁盘上数据块

副本的操作。这些名字节点指令都是以 `DatanodeCommand` 对象封装的，我们会在 `DatanodeCommand` 小节详细介绍名字节点指令以及 `DatanodeCommand` 对象。

`blockReport()`方法只在 `Datanode` 启动时以及指定间隔时执行一次。在这里间隔是由 `dfs.blockreport.intervalMsec` 参数配置的，默认是 6 小时执行一次。`cacheReport()`方法与 `blockReport()`方法是完全一致的，只不过汇报的是当前 `Datanode` 上缓存的所有数据块。`cacheReport()`方法的定义如下：

```
public DatanodeCommand cacheReport(DatanodeRegistration registration,
    String poolId, List<Long> blockIds) throws IOException;
```

(2) 心跳相关方法

我们知道分布式系统的节点之间大多采用心跳来维护节点的健康状态。HDFS 也是一样，`Datanode` 会定期（由 `dfs.heartbeat.interval` 配置项配置，默认是 3 秒）向 `Namenode` 发送心跳，如果 `Namenode` 长时间没有接到 `Datanode` 发送的心跳，则 `Namenode` 会认为该 `Datanode` 失效。

`ClientProtocol.sendHeartbeat()`方法就是用于心跳汇报的接口，除了携带标识 `Datanode` 身份的 `DatanodeRegistration` 对象外，还包括数据节点上所有存储的状态、缓存的状态、正在写文件数据的连接数、读写数据使用的线程数等。

`sendHeartbeat()`会返回一个 `HeartbeatResponse` 对象，这个对象包含了 `Namenode` 向 `Datanode` 发送的名字节点指令，以及当前 `Namenode` 的 HA 状态。需要特别注意的是，在开启了 HA 的 HDFS 集群中，`Datanode` 是需要同时向 `Active Namenode` 以及 `Standby Namenode` 发送心跳的，不过只有 `ActiveNamenode` 才能向 `Datanode` 下发名字节点指令。

```
public HeartbeatResponse sendHeartbeat(DatanodeRegistration registration,
    StorageReport[] reports,
    long dnCacheCapacity,
    long dnCacheUsed,
    int xmitsInProgress,
    int xceiverCount,
    int failedVolumes) throws IOException;
```

(3) 数据块读写相关方法

上面两个小节我们介绍了 `Datanode` 启动时以及发送心跳时与 `Namenode` 交互的方法。这一小节将介绍 `Datanode` 在进行数据块读写操作时与 `Namenode` 交互的方法，包括 `DatanodeProtocol` 中的 `reportBadBlocks()`、`blockReceivedAndDeleted()`以及 `commitBlockSynchronization()`方法。下面我们依次介绍这三个方法的定义及使用。

`reportBadBlocks()`与 `ClientProtocol.reportBadBlocks()`方法很类似，`Datanode` 会调用这个方法向 `Namenode` 汇报损坏的数据块。`Datanode` 会在三种情况下调用这个方法：`DataBlockScanner` 线程定期扫描数据节点上存储的数据块，发现数据块的校验出现错误时；数据流管道写数据时，`Datanode` 接受了一个新的数据块，进行数据块校验操作出现错误时；进行数据块复制操作（`DataTransfer`），`Datanode` 读取本地存储的数据块时，发现本地数据块副本的长度小于 `Namenode` 记录的长度，则认为该数据块已经无效，会调用 `reportBadBlocks()`方法。

reportBadBlocks()方法的参数是 LocatedBlock 对象, 这个对象描述了出现错误数据块的位置, Namenode 收到 reportBadBlocks()请求后, 会下发数据块副本删除指令删除错误的数据块。

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

Datanode 会定期 (默认是 5 分钟, 不可以配置) 调用 blockReceivedAndDeleted()方法向 Namenode 汇报 Datanode 新接受的数据块或者删除的数据块。Datanode 接受一个数据块, 可能是因为 Client 写入了新的数据块, 或者从别的 Datanode 上复制一个数据块到当前 Datanode。Datanode 删除一个数据块, 则有可能是因为该数据块的副本数量过多, Namenode 向当前 Datanode 下发了删除数据块副本的指令。我们可以把 blockReceivedAndDeleted()方法理解为 blockReport()的增量汇报, 这个方法的参数包括 DatanodeRegistration 对象、增量汇报数据块所在的块池 ID, 以及 StorageReceivedDeletedBlocks 对象的数组, 这里的 StorageReceived DeletedBlocks 对象封装了 Datanode 的一个数据存储上新添加以及删除的数据块集合。Namenode 接受了这个请求之后, 会更新它内存中数据块与数据节点的对应关系。

```
public void blockReceivedAndDeleted(DatanodeRegistration registration,
    String poolId,
    StorageReceivedDeletedBlocks[] rcvdAndDeletedBlocks)
    throws IOException;
```

DatanodeProtocol 中与数据块读写相关的最后一个方法是 commitBlockSynchronization(), 这个方法用于在租约恢复操作时同步数据块的状态。在租约恢复操作时, 主数据节点完成所有租约恢复协调操作后调用 commitBlockSynchronization()方法同步 Datanode 和 Namenode 上数据块的状态, 所以 commitBlockSynchronization()方法包含了大量的参数。对于租约恢复, 我们会在第 3 章的租约管理小节以及第 4 章的文件系统数据集小节中介绍, 请读者参考这两个章节内容。

```
public void commitBlockSynchronization(ExtendedBlock block,
    long newgenerationstamp, long newlength,
    boolean closeFile, boolean deleteblock, DatanodeID[] newtargets,
    String[] newtargetstorages) throws IOException;
```

(4) 其他方法

DataProtocol 中最后一个方法是 errorReport(), 该方法用于向名字节点上报运行过程中发生的一些状况, 如磁盘不可用等, 这个方法在调试时非常有用。由于这个方法并不涉及具体的逻辑, 我们就不再详细介绍了。

(5) DatanodeCommand

通过前面几个小节的介绍我们知道, sendHeartbeat()、blockReport()以及 cacheReport()方法的返回值都会携带 Namenode 向 Datanode 下发的名字节点指令。在 HDFS 中, 使用 DatanodeCommand 类描述 Namenode 向 Datanode 发出的名字节点指令。DatanodeCommand 类以及它的子类结构如图 1-4 所示。

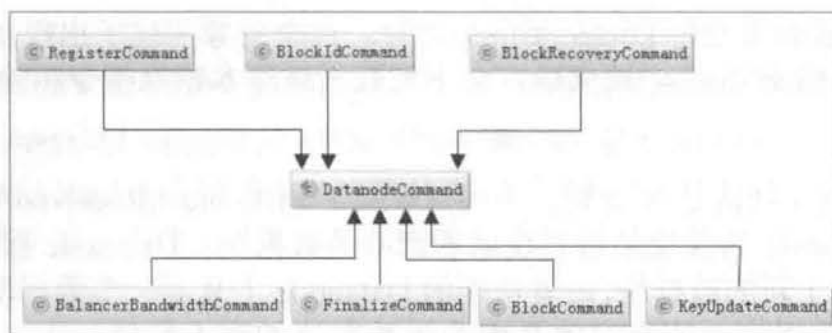


图 1-4 DatanodeCommand 类结构

DatanodeCommand 是所有名字节点的基类，它一共有 7 个子类，但在 DatanodeProtocol 中一共定义了 10 个名字节点指令，每个指令都有一个唯一的编号与之对应，代码如下：

```

public interface DatanodeProtocol {
    // ...
    final static int DNA_UNKNOWN = 0;    // 未定义
    final static int DNA_TRANSFER = 1;    // 数据块复制
    final static int DNA_INVALIDATE = 2;  // 数据块删除
    final static int DNA_SHUTDOWN = 3;    // 关闭数据节点
    final static int DNA_REGISTER = 4;    // 重新注册数据节点
    final static int DNA_FINALIZE = 5;    // 提交上一次升级
    final static int DNA_RECOVERBLOCK = 6; // 数据块恢复
    final static int DNA_ACCESSKEYUPDATE = 7; // 安全相关
    final static int DNA_BALANCERBANDWIDTHUPDATE = 8; // 更新平衡器带宽
    final static int DNA_CACHE = 9;      // 缓存数据块
    final static int DNA_UNCACHE = 10;   // 解缓存数据块
    // ...
}

```

可以看到，DatanodeProtocol 中定义的指令类型数量和 DatanodeCommand 子类的数量并不一致。这是因为指令编号 DNA_SHUTDOWN 已经废弃不用了，Datanode 接收到 DNA_SHUTDOWN 指令后会直接抛出 UnsupportedOperationException 异常。关闭 Datanode 是通过调用 ClientDatanodeProtocol.shutdownDatanode() 方法来触发的。

这里同时要注意的是，DNA_TRANSFER、DNA_RECOVERBLOCK 以及 DNA_INVALIDATE 都是通过 BlockCommand 子类来封装的，只不过参数不同。DNA_TRANSFER 指令用于触发数据节点的数据块复制操作，当 HDFS 系统中某个数据块的副本数小于配置的副本系数时，Namenode 会通过 DNA_TRANSFER 指令通知某个拥有这个数据块副本的 Datanode 将该数据块复制到其他数据节点上。DNA_INVALIDATE 用于通知 Datanode 删除数据节点上的指定数据块，这是因为 Namenode 发现了某个数据块的副本数已经超过了配置的副本系数，这时 Namenode 会通知某个数据节点删除这个数据节点上多余的数据块副本。当客户端在写文件时发生异常退出，会造成数据流管道中不同数据节点上数据块状态的不一致，这时 Namenode 会从数据流管道中选出一个数据节点作为主恢复节点，协调数据流管道中的其他数据节点进行租约恢复操作，以同步这个数据块的状态。此时 Namenode 就会向这个数据节点下发 DNA_RECOVERBLOCK 指令，通知数据节点开始租约恢复操作。

至于 `DatanodeCommand` 中其他类的作用都比较简单，这里就不再单独介绍了。

4. InterDatanodeProtocol

介绍完了 `Datanode` 与 `Namenode` 之间的接口，我们来介绍 `InterDatanodeProtocol`——`Datanode` 与 `Datanode` 之间的接口。`InterDatanodeProtocol` 接口主要用于租约恢复操作，如图 1-5 所示，`InterDatanodeProtocol` 只有 `initReplicaRecovery()` 和 `updateReplicaUnderRecovery()` 两个方法。

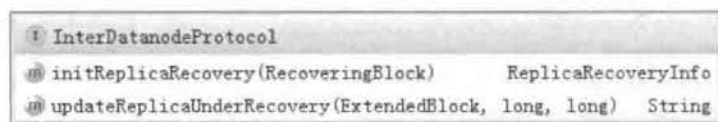


图 1-5 `InterDatanodeProtocol` 接口

客户端打开一个文件进行写操作时，首先要获取这个文件的租约，并且还需要定期更新租约。当 `Namenode` 的租约监控线程发现某个 HDFS 文件租约长期没有更新时，就会认为写这个文件的客户端发生异常，这时 `Namenode` 就需要触发租约恢复操作——同步数据流管道中所有 `Datanode` 上该文件数据块的状态，并强制关闭这个文件。

租约恢复的控制并不是由 `Namenode` 负责的，而是 `Namenode` 从数据流管道中选出一个主恢复节点，然后通过下发 `DatanodeCommand` 的恢复指令触发这个数据节点控制租约恢复操作，也就是由这个主恢复节点协调整个租约恢复操作的过程。主恢复节点会调用 `InterDatanodeProtocol` 接口来指挥数据流管道的其他数据节点进行租约恢复。租约恢复操作其实很简单，就是将数据流管道中所有数据节点上保存的同一个数据块状态（时间戳和数据块长度）同步一致。当成功完成租约恢复后，主恢复节点会调用 `DatanodeProtocol.commitBlockSynchronization()` 方法同步名字节点上该数据块的时间戳和数据块长度，保持名字节点和数据节点的一致。

由于数据流管道中同一个数据块状态（长度和时间戳）在不同的 `Datanode` 上可能是不一致的，所以主恢复节点会首先调用 `InterDatanodeProtocol.initReplicaRecovery()` 方法获取数据流管道中所有数据节点上保存的指定数据块的状态，这里的数据块状态使用 `ReplicaRecoveryInfo` 类封装。主恢复节点会根据收集到的这些状态，确定一个当前数据块的新长度，并且使用 `Namenode` 下发的 `recoverId` 作为数据块的新时间戳。

```
ReplicaRecoveryInfo initReplicaRecovery(RecoveringBlock rBlock)
throws IOException;
```

主恢复节点计算出数据块的新长度后，就会调用 `InterDatanodeProtocol.updateReplicaUnderRecovery()` 方法将数据流管道中所有节点上该数据块的长度同步为新的长度，将数据块的时间戳同步为新的时间戳。

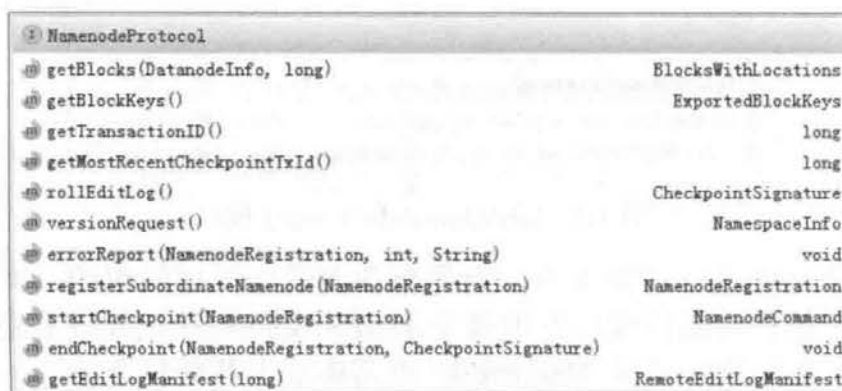
```
String updateReplicaUnderRecovery(ExtendedBlock oldBlock, long recoveryId,
    long newLength) throws IOException;
```

当完成了所有的同步操作后，主恢复节点就可以调用 `DatanodeProtocol.commitBlockSynchronization()` 将 `Namenode` 上该数据块的长度和时间戳同步为新的长度和时间戳，这样

Datanode 和 Namenode 的数据也就一致了。

5. NamenodeProtocol

NamenodeProtocol 定义了第二名字节点与名字节点之间的接口。由于在 Hadoop 2.X 架构中第二名字节点的功能已经完全被 Standby 节点所取代，这个接口我们就不再详细介绍了。图 1-6 给出了 NamenodeProtocol 定义的所有方法，读者可以参考这个图读取相关的代码。



NamenodeProtocol	
getBlocks(DatanodeInfo, long)	BlocksWithLocations
getBlockKeys()	ExportedBlockKeys
getTransactionID()	long
getMostRecentCheckpointTxId()	long
rollEditLog()	CheckpointSignature
versionRequest()	NamespaceInfo
errorReport(NamenodeRegistration, int, String)	void
registerSubordinateNamenode(NamenodeRegistration)	NamenodeRegistration
startCheckpoint(NamenodeRegistration)	NamenodeCommand
endCheckpoint(NamenodeRegistration, CheckpointSignature)	void
getEditLogManifest(long)	RemoteEditLogManifest

图 1-6 NamenodeProtocol 定义

1.2.2 流式接口

HDFS 除了定义 RPC 调用接口外，还定义了流式接口，流式接口是 HDFS 中基于 TCP 或者 HTTP 实现的接口。在 HDFS 中，流式接口包括了基于 TCP 的 DataTransferProtocol 接口，以及 HA 架构中 Active Namenode 和 Standby Namenode 之间的 HTTP 接口。

1. DataTransferProtocol

DataTransferProtocol 是用来描述写入或者读出 Datanode 上数据的基于 TCP 的流式接口，HDFS 客户端与数据节点以及数据节点与数据节点之间的数据块传输就是基于 DataTransferProtocol 接口实现的。HDFS 没有采用 Hadoop RPC 来实现 HDFS 文件的读写功能，是因为 Hadoop RPC 框架的效率目前还不足以支撑超大文件的读写，而使用基于 TCP 的流式接口有利于批量处理数据，同时提高了数据的吞吐量

图 1-7 给出了 DataTransferProtocol 的方法定义，其中最重要的方法就是 readBlock()和 writeBlock()。

- readBlock(): 从当前 Datanode 读取指定的数据块。
- writeBlock(): 将指定数据块写入数据流管道 (pipeLine) 中。
- transferBlock(): 将指定数据块复制 (transfer) 到另一个 Datanode 上。数据块复制操作是指数据流管道中的数据节点出现故障，需要用新的数据节点替换异常的数据节点时，DFSClient 会调用这个方法将数据流管道中异常数据节点上已经写入的数据块复制到新添加的数据节点上。

```

1 DataTransferProtocol
2 readBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, long, long, boolean, Caching) void
3 writeBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, DatanodeInfo[], StorageType[], DatanodeInfo[]) void
4 transferBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, DatanodeInfo[], StorageType[]) void
5 requestShortCircuitFds(ExtendedBlock, Token<BlockTokenIdentifier>, Slot[]) void
6 releaseShortCircuitFds(Slot[]) void
7 requestShortCircuitShm(String) void
8 replaceBlock(ExtendedBlock, StorageType, Token<BlockTokenIdentifier>, String, DatanodeInfo[]) void
9 copyBlock(ExtendedBlock, Token<BlockTokenIdentifier>, String, DatanodeInfo[]) void
10 blockChecksum(ExtendedBlock, Token<BlockTokenIdentifier>, String) void

```

图 1-7 DataTransferProtocol 接口

- **replaceBlock()**: 将从源 Datanode 复制来的数据块写入本地 Datanode。写成功后通知 NameNode，并且删除源 Datanode 上的数据块。这个方法主要用在数据块平衡操作 (balancing) 的场景下。
- **copyBlock()**: 复制当前 Datanode 上的数据块。这个方法主要用在数据块平衡操作的场景下。
- **blockChecksum()**: 获取指定数据块的校验值。
- **requestShortCircuitFds()**: 获取一个短路 (short circuit) 读取数据块的文件描述符 (请参考第 5 章的文件短路读操作小节)。
- **releaseShortCircuitFds()**: 释放一个短路读取数据块的文件描述符。
- **requestShortCircuitShm()**: 获取保存短路读取数据块的共享内存。

DataTransferProtocol 接口调用并没有使用 Hadoop RPC 框架提供的功能，而是定义了用于发送 DataTransferProtocol 请求的 Sender 类，以及用于响应 DataTransferProtocol 请求的 Receiver 类，Sender 类和 Receiver 类都实现了 DataTransferProtocol 接口。图 1-8 给出了 DataTransferProtocol 接口的一个调用示例。我们假设 DFSCClient 发起了一个 DataTransferProtocol.readBlock() 操作，那么 DFSCClient 会调用 Sender 将这个请求序列化，并传输给远端的 Receiver。远端的 Receiver 接收到这个请求后，会反序列化请求，然后调用代码执行读取操作。

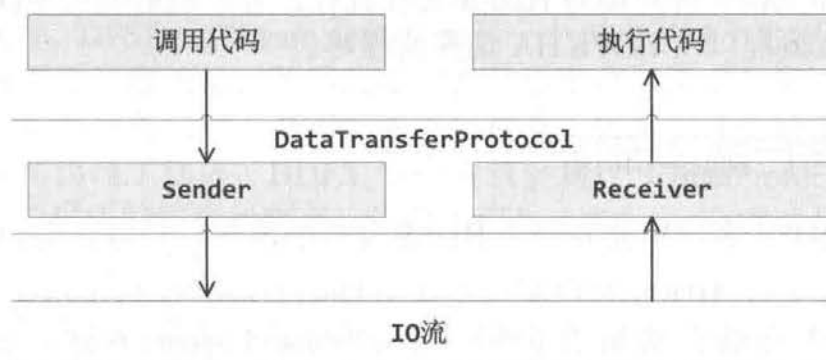


图 1-8 DataTransferProtocol 调用示例

DataTransferProtocol 接口的具体实现我们会在第 4 章的流式接口小节中详细介绍，请读者参考该小节内容学习。

2. Active Namenode 和 Standby Namenode 间的 HTTP 接口

Namenode 会定期将文件系统的命名空间（文件目录树、文件/目录元信息）保存到一个名叫 `fsimage` 的文件中，以防止 Namenode 掉电或者进程崩溃。但如果 Namenode 实时地将内存中的命名空间同步到 `fsimage` 文件中，将会非常地消耗资源且造成 Namenode 运行缓慢。所以 Namenode 会先将命名空间的修改操作保存在 `editlog` 文件中，然后定期合并 `fsimage` 和 `editlog` 文件。

合并 `fsimage` 和 `editlog` 文件是非常耗费资源的，所以在 Hadoop 2.X 版本之前，HDFS 引入了一个第二名字节点专门负责合并 `fsimage` 和 `editlog` 文件。而在 Hadoop 2.X 版本中，由于 Standby Namenode 会不断地将读入的 `editlog` 文件与当前的命名空间合并，从而始终保持着一个最新版本的命名空间，所以 Standby Namenode 只需定期将自己的命名空间写入一个新的 `fsimage` 文件，并通过 HTTP 协议将这个 `fsimage` 文件传回 Active Namenode 即可。

Active Namenode 和 Standby Namenode 之间的 HTTP 接口就是用来传输这个新的 `fsimage` 文件的。Standby Namenode 成功地将自己的命名空间写入新的 `fsimage` 文件后，就会向 Active Namenode 的 `ImageServlet` 发送 HTTP GET 请求 `/getimage?putimage=1`。这个请求的 URL 中包括了新的 `fsimage` 文件的事务 ID，以及 Standby Namenode 用于下载的端口和 IP 地址。Active Namenode 接收到这个请求后，会根据 Standby Namenode 提供的信息向 Standby Namenode 的 `ImageServlet` 发起 HTTP GET 请求以下载 `fsimage` 文件。

Active Namenode 和 Standby Namenode 的内容请读者参考第 3 章的 `FSImage` 以及 HA 小节。

1.3 HDFS 主要流程

在介绍完 HDFS 各种节点间的接口后，这一节我们将重点介绍 HDFS 的几个典型流程：客户端读 HDFS 文件流程、客户端写 HDFS 文件流程、客户端追加写 HDFS 文件流程、数据节点与名字节点交互流程以及 HDFS HA 切换流程等。

1.3.1 HDFS 客户端读流程

图 1-9 给出了 HDFS 客户端读取一个 HDFS 文件的流程，可以分为如下几个步骤。

- 打开 HDFS 文件：HDFS 客户端首先调用 `DistributedFileSystem.open()` 方法打开 HDFS 文件，这个方法在底层会调用 `ClientProtocol.open()` 方法，该方法会返回一个 `HdfsDataInputStream` 对象用于读取数据块。`HdfsDataInputStream` 其实是一个 `DFSInputStream` 的装饰类，真正进行数据块读取操作的是 `DFSInputStream` 对象。
- 从 Namenode 获取 Datanode 地址：在 `DFSInputStream` 的构造方法中，会调用 `ClientProtocol.getBlockLocations()` 方法向名字节点获取该 HDFS 文件起始位置数据块的位置信息。Namenode 返回的数据块的存储位置是按照与客户端的距离远近排序

的，所以 `DFSInputStream` 可以选择一个最优的 `Datanode` 节点，然后与这个节点建立数据连接读取数据块。

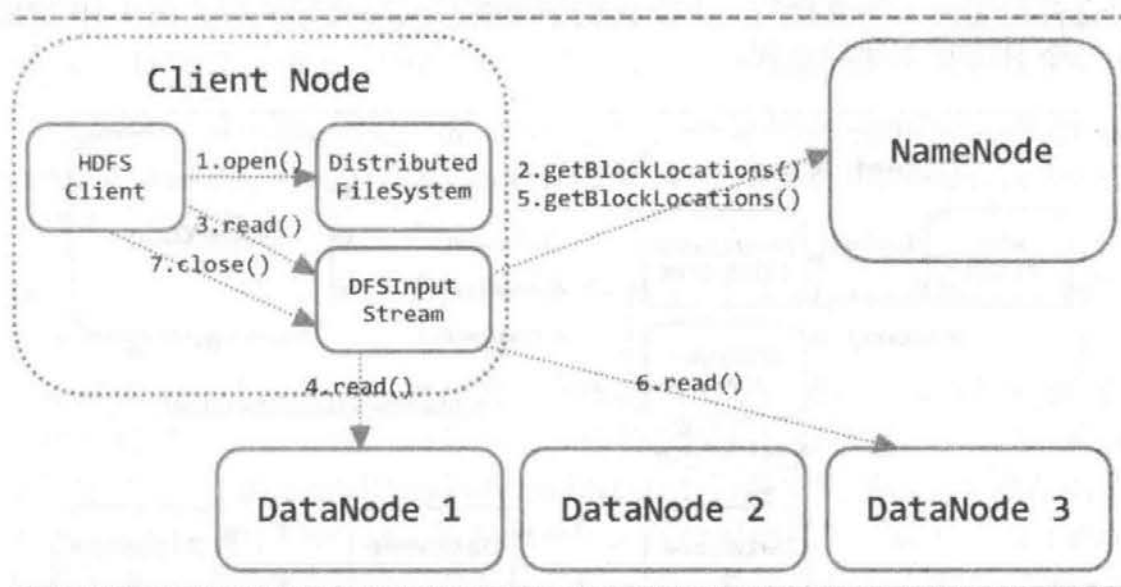


图 1-9 HDFS 读流程

- 连接到 `Datanode` 读取数据块：HDFS 客户端通过调用 `DFSInputStream.read()` 方法从这个最优的 `Datanode` 读取数据块，数据会以数据包（packet）为单位从数据节点通过流式接口传送到客户端。当达到一个数据块的末尾时，`DFSInputStream` 就会再次调用 `ClientProtocol.getBlockLocations()` 获取文件下一个数据块的位置信息，并建立和这个新的数据块的最优节点之间的连接，然后 HDFS 客户端就可以继续读取数据块了。
- 关闭输入流：当客户端成功完成文件读取后，会通过 `HdfsDataInputStream.close()` 方法关闭输入流。

客户端读取数据块时，很有可能存储这个数据块的数据节点出现异常，也就是无法读取数据。出现这种情况时，`DFSInputStream` 会切换到另一个保存了这个数据块副本的数据节点，然后读取数据。同时需要注意的是，数据块的应答包中不仅包含了数据，还包含了校验值。HDFS 客户端接收到数据应答包时，会对数据进行校验，如果出现校验错误，也就是数据节点上的这个数据块副本出现了损坏，HDFS 客户端就会通过 `ClientProtocol.reportBadBlocks()` 向 `Namenode` 汇报这个损坏的数据块副本，同时 `DFSInputStream` 会尝试从其他的数据节点读取这个数据块。

HDFS 客户端发起读操作请求以及 `HdfsDataInputStream` 输入流的实现会在第 5 章的文件读操作与输入流小节中介绍，`Datanode` 响应读数据请求的实现会在第 4 章的流式接口小节中介绍，`Namenode` 响应 HDFS 客户端 RPC 请求的实现会在第 3 章的 `ClientProtocol` 实现小节中介绍。

1.3.2 HDFS 客户端写流程

在介绍完客户端读文件流程后，下面我们学习客户端写文件流程。图 1-10 给出了 HDFS 客户端写入一个 HDFS 文件的流程。

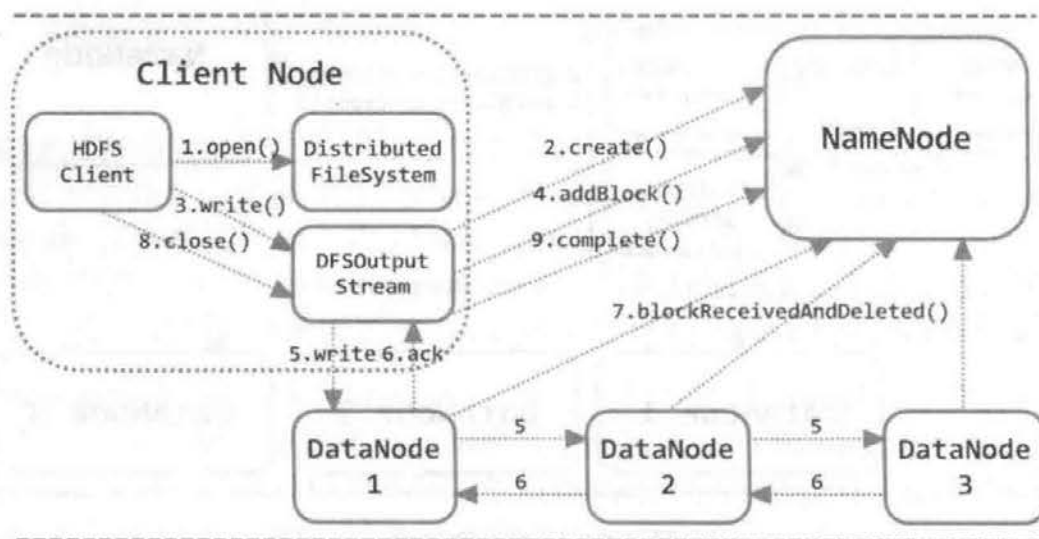


图 1-10 HDFS 写流程

- **创建文件:** HDFS 客户端写一个新的文件时，会首先调用 `DistributedFileSystem.create()` 方法在 HDFS 文件系统中创建一个新的空文件。这个方法在底层会通过调用 `ClientProtocol.create()` 方法通知 Namenode 执行对应的操作，Namenode 会首先在文件系统目录树中的指定路径下添加一个新的文件，然后将创建新文件的操作记录到 `editlog` 中。完成 `ClientProtocol.create()` 调用后，`DistributedFileSystem.create()` 方法就会返回一个 `HdfsDataOutputStream` 对象，这个对象在底层包装了一个 `DFSOutputStream` 对象，真正执行写数据操作的其实是 `DFSOutputStream` 对象。
- **建立数据流管道:** 获取了 `DFSOutputStream` 对象后，HDFS 客户端就可以调用 `DFSOutputStream.write()` 方法来写数据了。由于 `DistributedFileSystem.create()` 方法只是在文件系统目录树中创建了一个空文件，并没有申请任何数据块，所以 `DFSOutputStream` 会首先调用 `ClientProtocol.addBlock()` 向 Namenode 申请一个新的空数据块，`addBlock()` 方法会返回一个 `LocatedBlock` 对象，这个对象保存了存储这个数据块的所有数据节点的位置信息。获得了数据流管道中所有数据节点的信息后，`DFSOutputStream` 就可以建立数据流管道写数据块了。
- **通过数据流管道写入数据:** 成功地建立数据流管道后，HDFS 客户端就可以向数据流管道写数据了。写入 `DFSOutputStream` 中的数据会先被缓存在数据流中，之后这些数据会被切分成一个个数据包 (packet) 通过数据流管道发送到所有数据节点。这里的每个数据包都会按照图 1-10 所示，通过数据流管道依次写入数据节点的本地存储。每个数据包都有个确认包，确认包会逆序通过数据流管道回到输出流。输出流在确认了所有数据节点已经写入这个数据包之后，就会从对应的缓存队列删除这个数据包。当客户端写满一个数据块之后，会调用 `addBlock()` 申请一个新的数据块，

然后循环执行上述操作。

- 关闭输入流并提交文件：当 HDFS 客户端完成了整个文件中所有数据块的写操作之后，就可以调用 `close()` 方法关闭输出流，并调用 `ClientProtocol.complete()` 方法通知 Namenode 提交这个文件中的所有数据块，也就完成了整个文件的写入流程。

对于 Datanode，当 Datanode 成功地接受一个新的数据块时，Datanode 会通过 `DatanodeProtocol.blockReceivedAndDeleted()` 方法向 Namenode 汇报，Namenode 会更新内存中的数据块与数据节点的对应关系。

如果客户端在写文件时，数据流管道中的数据节点出现故障，则输出流会进行如下操作来进行故障恢复。

- 输出流中缓存的没有确认的数据包会重新加入发送队列，这种机制确保了数据节点出现故障时不会丢失任何数据，所有的数据都是经过确认的。但是输出流会通过调用 `ClientProtocol.updateBlockForPipeline()` 方法为数据块申请一个新的时间戳，然后使用这个新的时间戳重新建立数据流管道。这种机制保证了故障 Datanode 上的数据块的时间戳会过期，然后在故障恢复之后，由于数据块的时间戳与 Namenode 元数据中的不匹配而被删除，保证了集群中所有数据块的正确性。
- 故障数据节点会从输入流管道中删除，然后输出流会通过调用 `ClientProtocol.getAdditionalDatanode()` 方法通知 Namenode 分配新的数据节点到数据流管道中。接下来输出流会将新分配的 Datanode 添加到数据流管道中，并使用新的时间戳重新建立数据流管道。由于新添加的数据节点上并没有存储这个新的数据块，这时 HDFS 客户端会通过 `DataTransferProtocol` 通知数据流管道中的一个 Datanode 复制这个数据块到新的 Datanode 上。
- 数据流管道重新建立之后，输出流会调用 `ClientProtocol.updatePipeline()` 更新 Namenode 中的元数据。至此，一个完整的故障恢复流程就完成了，客户端可以正常完成后续的写操作了。

HDFS 客户端发起写操作请求以及 `HdfsDataOutputStream` 输出流的实现会在第 5 章的文件写操作与输出流小节中介绍，Datanode 响应写数据请求的实现会在第 4 章的流式接口小节中介绍，Namenode 响应 HDFS 客户端 RPC 请求的实现会在第 3 章的 `ClientProtocol` 实现小节中介绍。

1.3.3 HDFS 客户端追加写流程

客户端除了可以执行写入新的 HDFS 文件操作外，还可以打开一个已有文件并执行追加写操作。

- 打开已有的 HDFS 文件：客户端调用 `DistributedFileSystem.append()` 方法打开一个已有的 HDFS 文件，`append()` 方法首先会调用 `ClientProtocol.append()` 方法获取文件最后一个数据块的位置信息，如果文件的最后一个数据块已经写满则返回 `null`。然后 `append()` 方法会调用 `DFSOutputStream.newStreamForAppend()` 方法创建到这个数据块

的 `DFSOutputStream` 输出流对象，获取文件租约，并将新构建的 `DFSOutputStream` 方法包装为 `HdfsDataOutputStream` 对象，最后返回。

- 建立数据流管道：`DFSOutputStream` 类的构造方法会判断文件最后一个数据块是否已经写满，如果没有写满，则根据 `ClientProtocol.append()` 方法返回的该数据块的位置信息建立到该数据块的数据流管道；如果写满了，则调用 `ClientProtocol.addBlock()` 向 `Namenode` 申请一个新的空数据块之后建立数据流管道。
- 通过数据流管道写入数据：成功地建立数据流管道后，HDFS 客户端就可以向数据流管道写数据了，这部分内容与上节描述的写 HDFS 文件流程类似。
- 关闭输入流并提交文件：与上节描述的写 HDFS 文件流程类似，当 HDFS 客户端完成了追加写操作后，需要调用 `close()` 方法关闭输出流，并调用 `ClientProtocol.complete()` 方法通知 `Namenode` 提交这个文件中的所有数据块。

可以看到，HDFS 客户端追加写流程与写流程是很类似的，只不过在初始建立数据流管道时有些不同，请读者们注意。HDFS 客户端发起追加写操作请求的部分会在第 5 章的文件写操作与输出流小节中介绍，`Datanode` 响应写数据请求的实现会在第 4 章的流式接口小节中介绍，`Namenode` 响应 HDFS 客户端 RPC 请求的实现会在第 3 章的 `ClientProtocol` 实现小节中介绍。

1.3.4 Datanode 启动、心跳以及执行名字节点指令流程

上面介绍的三个流程主要是 HDFS 客户端发起的流程，这一节我们介绍 `Datanode` 与 `Namenode` 的交互流程，如图 1-11 所示，包括 `Datanode` 启动时的注册流程、心跳流程、数据块汇报以及增量汇报等流程。

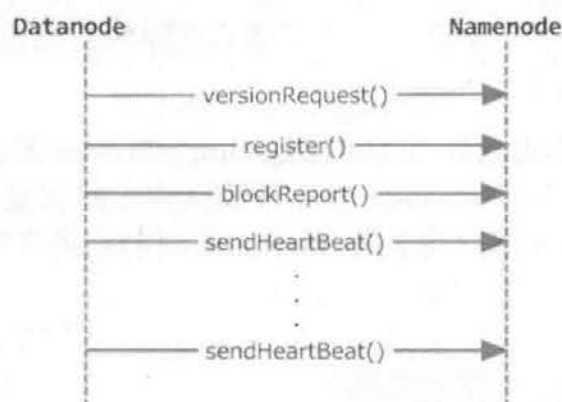


图 1-11 Datanode 启动、心跳以及执行名字节点指令流程

`Datanode` 启动后与 `Namenode` 的交互主要包括三个部分：①握手；②注册；③块汇报以及缓存汇报。

- `Datanode` 启动时会首先通过 `DatanodeProtocol.versionRequest()` 获取 `Namenode` 的版本号以及存储信息等，然后 `Datanode` 会对 `Namenode` 的当前软件版本号和 `Datanode` 的当前软件、版本号进行比较，确保它们是一致的。

- 成功地完成握手操作后, Datanode 会通过 `DatanodeProtocol.register()` 方法向 Namenode 注册。Namenode 接收到注册请求后, 会判断当前 Datanode 的配置是否属于这个集群, 它们之间的版本号是否一致。
- 注册成功之后, Datanode 就需要将本地存储的所有数据块以及缓存的数据块上报到 Namenode, Namenode 会利用这些信息重新建立内存中数据块与 Datanode 之间的对应关系。

至此, Datanode 就完成了启动的所有操作, 之后就可以正常对外服务了。

Datanode 成功启动之后, 需要定期向 Namenode 发送心跳, 让 Namenode 知道当前 Datanode 处于活动状态能够对外服务。Namenode 会在 Datanode 的心跳响应中携带名字节点指令, 指导 Datanode 进行数据块的复制、删除以及恢复等操作, 具体可以执行哪些指令请参考本章的 `DatanodeCommand` 小节。

当 Datanode 成功地添加了一个新的数据块或者删除了一个已有的数据块时, 需要通过 `DatanodeProtocol.blockReceivedAndDeleted()` 方法向 Namenode 汇报。Namenode 接收到这个汇报之后, 会更新 Namenode 内存中数据块与数据节点之间的对应关系。

Datanode 向 Namenode 发起交互请求的内容会在第4章的 `BlockPoolManager` 小节中介绍, 而 Namenode 对 Datanode 请求的响应则会在第3章的数据块管理以及数据节点管理小节中介绍。

1.3.5 HA 切换流程

在 Hadoop 2.X 之前, Namenode 是 HDFS 集群中可能发生单点故障的节点, 即每个 HDFS 集群中只有一个 Namenode, 一旦这个节点不可用, 整个 HDFS 集群就将处于不可用状态。

HDFS 的高可用 (High Availability, HA) 方案就是为了解决上述问题而产生的。图 1-12 给出了一个 HA HDFS 集群, 在 HA HDFS 集群中会同时运行两个 Namenode, 一个作为活动的 (Active) Namenode, 一个作为备份的 (Standby) Namenode。Active Namenode 的命名空间与 Standby Namenode 是实时同步的, 所以当 Active Namenode 发生故障而停止服务时, Standby Namenode 可以立即切换为活动状态, 而不影响 HDFS 集群的服务。

为了使 Standby 节点与 Active 节点的状态能够同步一致, 就要求两个 Namenode 的命名空间一致并且数据块与数据节点之间的对应关系一致。对于命名空间的一致性, 两个节点都需要与一组独立运行的节点 (JournalNodes, JNS) 通信, 当 Active Namenode 执行了修改命名空间的操作时, 它会定期将执行的操作记录在 `editlog` 中, 并写入 JNS 的多数节点中。而 Standby Namenode 会一直监听 JNS 上 `editlog` 的变化, 如果发现 `editlog` 有改动, Standby Namenode 就会读取 `editlog` 并与当前的命名空间合并。当发生了错误切换时, Standby 节点会先保证已经从 JNS 上读取了所有的 `editlog` 并与命名空间合并, 然后才会从 Standby 状态切换为 Active 状态。通过这种机制, 保证了 Active Namenode 与 Standby Namenode 之间命名空间状态的一致性。而对于数据块与数据节点对应关系的一致性, 则要求 HDFS 集群中的所有

Datanode 同时向这两个 Namenode 发送心跳以及块汇报信息, 这样 Active Namenode 和 Standby Namenode 的数据块与数据节点之间的对应关系也就完全同步了。一旦发生故障, 就可以马上切换, 也就是热备。

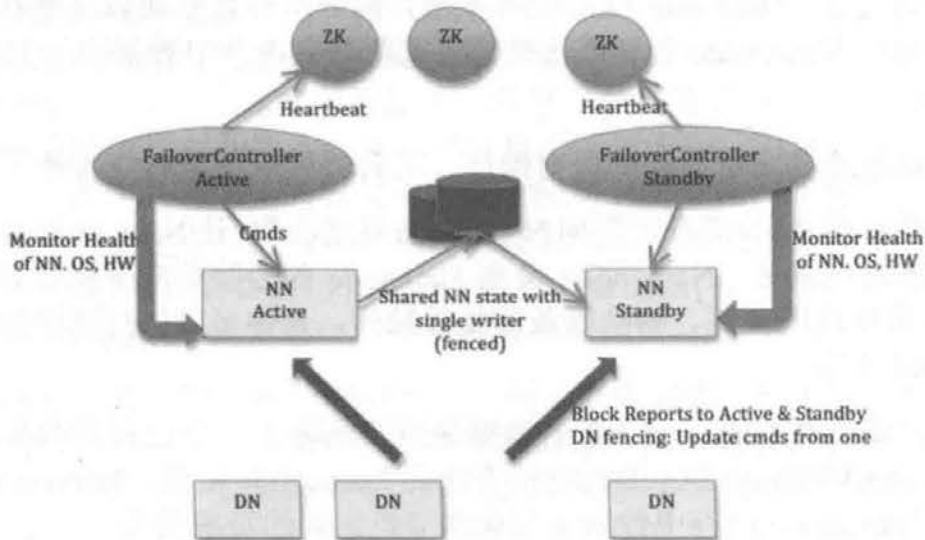


图 1-12 HA 切换流程

HDFS 提供了 HA 管理命令 (DFSHAAdmin) 使得管理员可以手动执行主备切换, 同时还提供了自动 Failover 机制, 该机制依赖于两个新增的网元: 一个是 ZooKeeper 集群; 一个是 ZKFailoverController (org.apache.hadoop.ha.ZKFailoverController)。ZKFailoverController 会实时监控 Namenode 的 HA 状态, 如果 Active Namenode 处于不可服务状态, 那么 ZKFailoverController 会自动触发主备切换操作, 无须管理员执行任何命令。

我们会在第 3 章的 HDFS High Availability 小节中重点介绍 HDFS HA 架构的实现, 请读者参考该小节内容。

第 2 章 Hadoop RPC

Hadoop 作为分布式存储系统，各个节点之间的通信和交互是必不可少的，所以需要实现一套节点间的通信交互机制。RPC（Remote Procedure Call Protocol，远程过程调用协议）允许本地程序像调用本地方法一样调用远程机器上应用程序提供的服务，所以 Hadoop 实现了一套自己的 RPC 框架。HadoopRPC 框架并没有使用 JDK 自带的 RMI（Remote Method Invocation，远程方法调用），而是基于 IPC（Inter-Process Communications，进程间通信）模型实现了一套高效的轻量级 RPC 框架，这套 RPC 框架底层采用了 JavaNIO、Java 动态代理以及 protobuf 等基础技术。

本章将介绍 HadoopRPC 框架的实现细节，但由于篇幅原因不会介绍底层依赖的 NIO、动态代理以及 protobuf 等技术，请读者自行查阅这部分内容。

2.1 概述

在介绍 Hadoop RPC 实现细节之前，有必要简要地介绍一下 Hadoop RPC 框架的总体结构以及使用方式。

2.1.1 RPC 框架概述

RPC（Remote Procedure Call Protocol，远程过程调用协议）是一种通过网络调用远程计算机服务的协议。RPC 协议假定存在某些网络传输协议，如 TCP 或 UDP，RPC 会使用这些协议传递 RPC 请求以及响应信息。RPC 协议使得分布式程序的开发更加容易，因此很受欢迎。

RPC 采用客户端/服务器模式，请求程序就是一个客户端，而服务提供程序就是一个服务器。客户端首先会发送一个有参数的调用请求到服务器，然后等待服务器发回响应信息。在服务器端，服务提供程序会保持睡眠状态直到有调用请求到达为止。当一个调用请求到达后，服务提供程序会执行调用请求，计算结果，向客户端发送响应信息，然后等待下一个调用请求。最后，客户端成功地接收服务器发回的响应信息，一个远程调用结束。

图 2-1 给出了 RPC 框架的结构，主要包括以下几部分。

- 通信模块：传输 RPC 请求和响应的网络通信模块，可以基于 TCP 协议，也可以基于 UDP 协议，可以是同步的，也可以是异步的。

- 客户端 Stub 程序：服务器和客户端都包括 Stub 程序。在客户端，Stub 程序表现得就像本地程序一样，但底层却会将调用请求和参数序列化并通过通信模块发送给服务器。之后 Stub 程序会等待服务器的响应信息，将响应信息反序列化并返回给请求程序。
- 服务器端 Stub 程序：在服务器端，Stub 程序会将远程客户端发送的调用请求和参数反序列化，根据调用信息触发对应的服务程序，然后将服务程序返回的响应信息序列化并发回客户端。
- 请求程序：请求程序会像调用本地方法一样调用客户端 Stub 程序，然后接收 Stub 程序返回的响应信息。
- 服务程序：服务器会接收来自 Stub 程序的调用请求，执行对应的逻辑并返回执行结果。

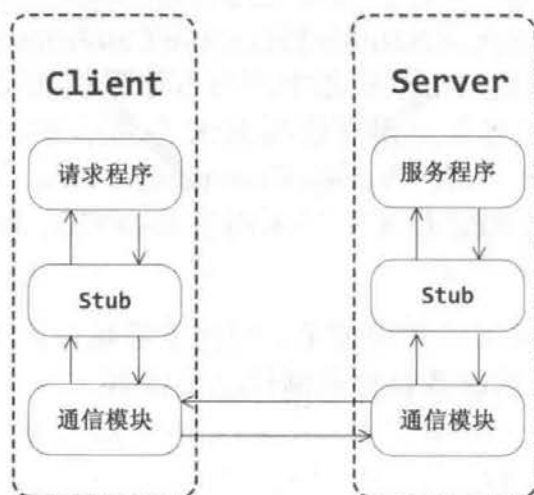


图 2-1 RPC 框架结构图

2.1.2 Hadoop RPC 框架概述

Hadoop RPC 框架与其他 RPC 框架一样实现了图 2-1 所示的结构，本节介绍 RPC 框架中各个模块在 Hadoop RPC 中的实现。

1. 通信模块

Hadoop 实现了 `org.apache.hadoop.ipc.Client` 类（之后简称为 Client 类）以及 `org.apache.hadoop.ipc.Server` 类（之后简称为 Server 类）提供的基于 TCP/IP Socket 的网络通信功能，客户端可以通过 Client 类将序列化的请求发送到远程服务器，服务器会通过 Server 类接收来自客户端的请求。

客户端在发送请求到远程服务器前需要先将请求序列化，然后调用 `Client.call()` 方法发送这个请求到远程服务器。为了使 RPC 机制更加健壮，Hadoop RPC 允许客户端配置使用不同的序列化框架序列化 RPC 请求（例如 `protobuf`、`avro`），这就要求 `Client.call()` 方法的定义更加通用，也就是 `Client.call()` 方法可以发送任意序列化框架产生的 RPC 请求。所以 `Client.call()` 方法的定义如下代码所示，`rpcKind` 参数用于描述 RPC 请求的序列化工具类型，`rpcRequest` 参数则用于记录序列化后的 RPC 请求。这里需要注意的是，`rpcRequest` 是 `Writable` 类型的

(Hadoop 框架自己定义的序列化类型), 这就要求客户端 Stub 程序将 RPC 请求序列化后包装成 Writable 类型, 所以 WritableRpcEngine 定义了 Invocation 类包装 Writable 序列化的 RPC 请求, 而 ProtobufRpcEngine 则定义了 RpcRequestWrapper 包装 protobuf 序列化的 RPC 请求。这里的 Client 类、WritableRpcEngine 类以及 ProtobufRpcEngine 类的实现会在 Hadoop RPC 实现小节中介绍。

```
public Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,
    ConnectionId remoteId, AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    // ...
}
```

再看服务器端, 为了提高性能, Server 类采用了 Java NIO 提供的基于 Reactor 设计模式的事件驱动 I/O 模型, 当 Server 完整地从网络接收一个 RPC 请求后, 会调用 call() 方法响应这个请求, 如下代码所示, Server.call() 方法的定义与 Client.call() 很相似。Server 的实现也会在 Hadoop RPC 实现小节中介绍。

```
public abstract Writable call(RPC.RpcKind rpcKind, String protocol,
    Writable param, long receiveTime) throws Exception;
```

2. 客户端 Stub 程序

客户端的 Stub 可以看作是一个代理对象, 它会将请求程序的 RPC 调用序列化, 并调用 Client.call() 方法将这个请求发送给远程服务器, 这些实现对于客户端调用程序是完全透明的。

客户端 Stub 程序做的第一件事情就是将 RPC 请求序列化, Hadoop 2.X 默认使用 protobuf 作为序列化工具, 当然 Hadoop RPC 框架也支持其他的序列化框架。Hadoop 定义了 RpcEngine 接口抽象使用不同序列化框架的 RPC 引擎, 如图 2-2 所示, RpcEngine 接口包括两个重要的方法。

- **getProxy():** 客户端会调用 RpcEngine.getProxy() 方法获取一个本地接口的代理对象, 然后在这个代理对象上调用本地接口的方法。getProxy() 方法的实现采用了 Java 动态代理机制, 客户端调用程序在代理对象上的调用会由一个 RpcInvocationHandler (java.lang.reflect.InvocationHandler 的子类, 在 RpcEngine 的实现类中定义) 对象处理, 这个 RpcInvocationHandler 会将请求序列化 (使用 RpcEngine 实现类定义的序列化方式) 并调用 Client.call() 方法将请求发送到远程服务器。当远程服务器发回响应信息后, RpcInvocationHandler 会将响应信息反序列化并返回给调用程序, 这一切通过 Java 动态代理机制对于调用程序是完全透明的, 就像本地调用一样。
- **getServer():** 该方法用于产生一个 RPC Server 对象, 服务器会启动这个 Server 对象监听从客户端发来的请求。成功从网络接收请求数据后, Server 对象会调用 RpcInvoker (在 RpcEngine 的实现类中定义) 对象处理这个请求。

RpcEngine 目前有两个子类, 其中 WritableRpcEngine 用于描述使用 Hadoop 自带的 Writable 作为序列化工具的 RPC 引擎; ProtobufRpcEngine 用于描述使用 protobuf 作为序列化工具的 RPC 引擎。如图 2-3 所示, WritableRpcEngine 和 ProtobufRpcEngine 都定义了若干内部类, 在

这里我们简单介绍一下这些类的作用，在 Hadoop RPC 实现小节中会详细介绍这些类之间的关系以及代码实现细节。

```

+ RpcEngine
+ getProxy(Class<T>, long, InetAddress, UserGroupInformation, Configuration, SocketFactory, int, RetryPolicy) ProtocolProxy<T>
+ getProxy(Class<T>, long, InetAddress, UserGroupInformation, Configuration, SocketFactory, int, RetryPolicy, AtomicBoolean) LocalProxy<T>
+ getServer(Class<?>, Object, String, int, int, int, int, boolean, Configuration, SecretManager<? extends TokenIdentifier>, String) Server
+ getProtocolMetaInfoProxy(ConnectionId, Configuration, SocketFactory) ProtocolProxy<ProtocolMetaInfoPB>

```

图 2-2 RpcEngine 接口定义

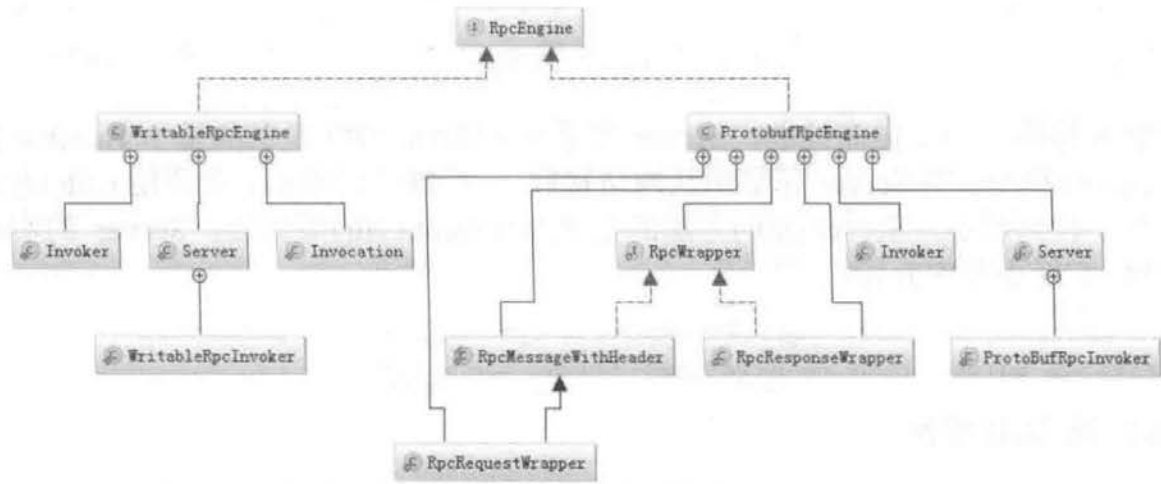


图 2-3 RpcEngine 继承结构图

(1) WritableRpcEngine

- **Invoker:** WritableRpcEngine.Invoker 是 InvocationHandler 的子类 (Java 动态代理框架中的处理类)。客户端首先会调用 WritableRpcEngine.getProxy() 获取一个本地接口 (例如 ClientProtocol) 的代理对象，然后在这个代理对象上调用本地接口的方法。通过 Java 动态代理机制的处理，这个调用会被 WritableRpcEngine.Invoker 类的 invoke() 方法响应。WritableRpcEngine.Invoker.invoke() 方法会使用 Writable 序列化框架将 RPC 请求以及参数序列化，然后构造一个 Invocation 对象 (WritableRpcEngine 的子类，实现了 Writable 接口) 包装序列化的 RPC 请求以及参数，最后调用 Client.call() 方法将这个 Invocation 对象发送到远程服务器，并等待远程服务器的响应信息。成功获取了服务器发回的响应信息后，Invoker 会将响应信息反序列化并返回给客户端。
- **Invocation:** 在通信模块小节中我们介绍了 Client.call() 方法只能发送 Writable 类型的 RPC 请求，这样定义是为了让发送接口更通用，不同框架序列化的请求只需包装成一个 Writable 对象就可以通过 Client 发送。这里的 Invocation 内部类就是为了包装 Writable 的 RPC 请求，它保存了客户端在什么接口上调用什么方法，以及这个方法的参数等调用信息。Invoker 构造了这个对象之后，就会调用 Client.call() 方法将这个对象发送到远程服务器。
- **Server:** WritableRpcEngine.Server 是 org.apache.hadoop.ipc.Server 类的子类，服务器代码会调用 WritableRpcEngine.getServer() 方法获取一个 WritableRpcEngine.Server 对象，WritableRpcEngine.Server 类继承了的 Server 大部分方法，它会在 Socket 上监听

RPC 请求，并调用 `WritableRpcInvoker` 类的 `call()` 方法响应这个请求。

- **WritableRpcInvoker:** `WritableRpcInvoker` 是 `WritableRpcEngine.Server` 的内部类，它实现了 `RpcInvoker` 接口，用于响应远程客户端的请求。`WritableRpcInvoker` 会先使用 `Writable` 序列化工具反序列化请求信息以及请求参数，然后根据请求信息反射调用服务程序，并将响应结果包装成一个 `Writable` 对象返回。

(2) ProtobufRpcEngine

- **Invoker:** 与 `WritableRpcEngine.Invoker` 类相同，都是用于处理客户端发送 RPC 请求的。不同处有 4 点：①RPC 请求信息以及请求参数的序列化使用了 `protobuf` 工具；②使用 `RPCRequestWrapper` 类包装 RPC 请求（包括请求信息以及请求参数）；③使用 `RPCResponseWrapper` 类包装服务器发回的 RPC 响应信息；④使用 `protobuf` 反序列化远程服务器发回的 RPC 响应信息。
- **RpcRequestWrapper:** 我们知道 `Client.call()` 方法只接受 `Writable` 类型的 RPC 请求，而 `ProtobufRpcEngine.Invoker` 是使用 `protobuf` 序列化 RPC 请求的，这里就需要一个包装类 `RpcRequestWrapper` 将 `protobuf` 格式的请求包装成 `Writable` 类型。`RpcRequestWrapper` 保存了两部分信息，这两部分信息都是使用 `protobuf` 序列化的——①RPC 请求头：RPC 请求头记录了客户端在什么接口上调用了什么方法；②请求参数：方法调用的参数。`Invoker` 类构造了 `RpcRequestWrapper` 对象后，会调用 `Client.call()` 方法将这个请求发送到远程服务器。
- **RpcResponseWrapper:** `Client.call()` 方法返回的响应信息也是 `Writable` 类型的，但是服务器端使用了 `protobuf` 序列化 RPC 响应信息，这就需要定义一个包装类 `RpcResponseWrapper` 将这个 `protobuf` 序列化的响应信息包装成 `Writable` 类型。
- **Server:** 与 `WritableRpcEngine.Server` 类的功能类似，它会在 `Socket` 上监听 RPC 请求，并调用 `ProtoBufRpcInvoker` 类的 `call()` 方法响应这个请求。
- **ProtoBufRpcInvoker:** 与 `WritableRpcInvoker` 类的功能类似，用于响应远程客户端的请求。不同之处有如下几点：①使用了 `protobuf` 这个序列化工具；②使用了 `RPCRequestWrapper` 类包装 RPC 请求（包括请求信息以及请求参数）；③使用了 `RPCResponseWrapper` 类包装服务器发回的 RPC 响应信息。

了解了客户端 `Stub` 程序的功能，我们再来学习客户端调用程序获得 `Stub` 引用的流程。客户端会调用 `RPC.getProtocolProxy()` 方法获取某个本地接口（例如 `ClientProtocol`）的代理对象，之后调用程序就可以在该代理对象上调用本地接口的方法了。如下代码所示，`RPC.getProtocolProxy()` 方法会首先调用 `getProtocolEngine()` 获取当前 RPC 类的序列化引擎（可能是 `WritableRpcEngine` 或者 `ProtobufRpcEngine`），然后调用 `RpcEngine.getProxy()` 方法获取代理对象。`RpcEngine.getProxy()` 方法的具体实现我们将在客户端获取 `Proxy` 对象小节中介绍。

```
public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,
                                                    long clientVersion,
                                                    InetAddress addr,
                                                    UserGroupInformation ticket,
```

```
Configuration conf,
SocketFactory factory,
int rpcTimeout,
RetryPolicy connectionRetryPolicy,
AtomicBoolean fallbackToSimpleAuth)
throws IOException {
return getProtocolEngine(protocol, conf).getProxy(protocol, clientVersion,
addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy,
fallbackToSimpleAuth);
}
```

至此，与客户端 Stub 相关的内容就介绍完了。图 2-4 给出了调用程序通过 RPC.getProxy() 获取一个 TestProtocol 接口的代理对象，然后在代理对象上调用 TestProtocol.test() 方法的流程。这个流程的实现细节将会在客户端获取 Proxy 对象小节中介绍。

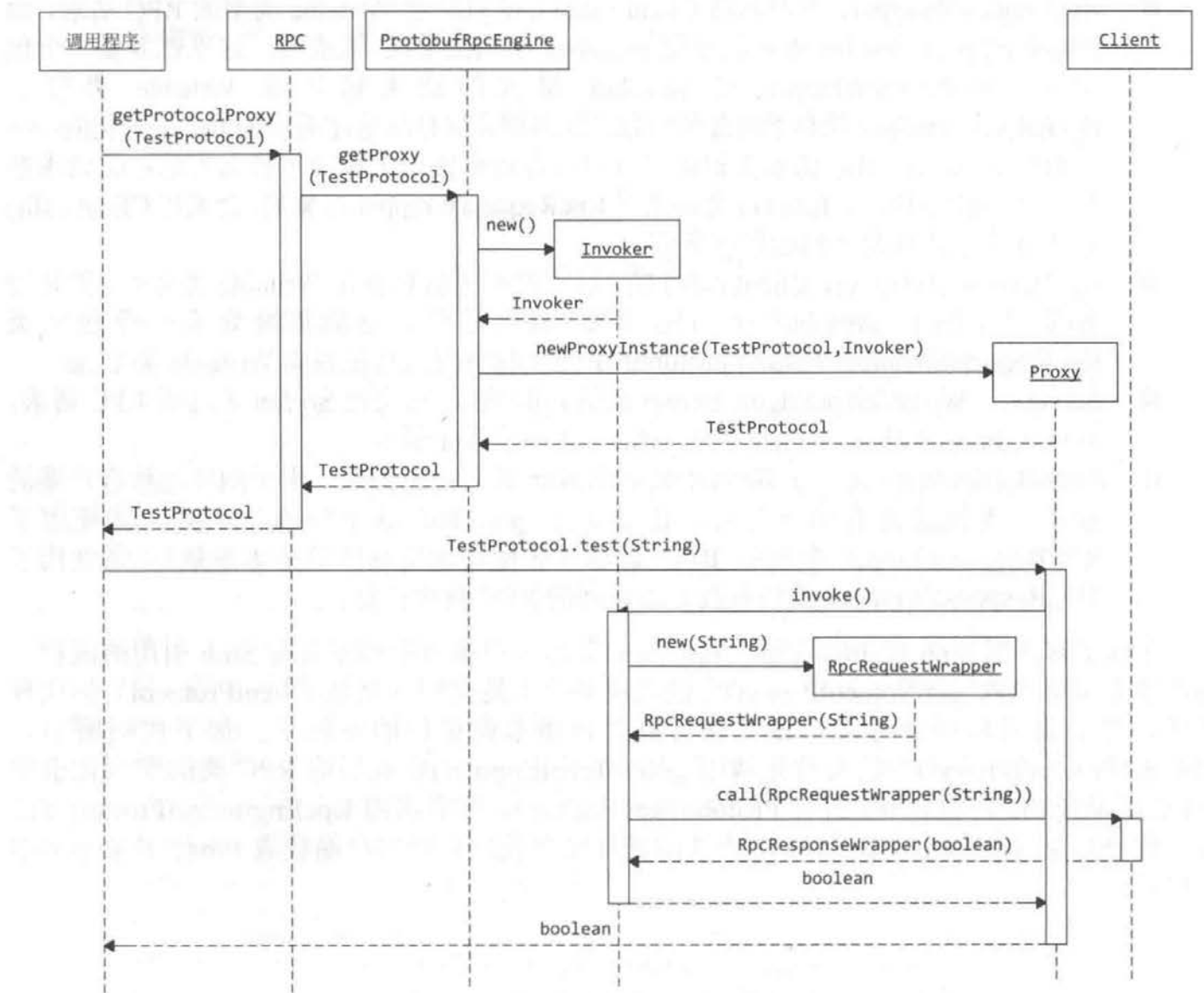


图 2-4 客户端调用流程

3. 服务器端 Stub 程序

服务器端 Stub 程序会将通信模块接收的数据反序列化，然后调用服务程序对应的方法响应这个 RPC 请求。在通信模块小节中我们介绍了 Server 类是服务器端从网络接收 RPC 请求的类，成功地接收一个 RPC 请求后，Server 会调用 call() 方法响应这个请求。

RPC.Server.call() 实现了父类 Server 的抽象方法 call()。RPC.Server.call() 的代码如下所示，它首先调用 getRpcInvoker() 获取一个 RpcInvoker 对象，然后调用 RpcInvoker.call() 方法响应这个 RPC 请求，这个 RpcInvoker 对象可以理解为服务器端 Stub 程序的实现。

```
public Writable call(RPC.RpcKind rpcKind, String protocol,
    Writable rpcRequest, long receiveTime) throws Exception {
    return getRpcInvoker(rpcKind).call(this, protocol, rpcRequest,
        receiveTime);
}
```

如图 2-3 所示，不同的 RpcEngine 会实现自己的 Server 对象 (RPC.Server 的子类)，Server 对象又会实现一个内部的 RpcInvoker 对象，例如 ProtobufRpcEngine 实现了内部类 Server，而 Server 类又定义了自己的 ProtoBufRpcInvoker 类。这个 ProtoBufRpcInvoker 对象会使用 protobuf 反序列化 RPC 请求 (RpcRequestWrapper 类型)，然后调用服务程序响应这个请求，最后 ProtoBufRpcInvoker 会将响应消息序列化包装成一个 RpcResponseWrapper 对象并返回。整个服务器端 Stub 处理 RPC 请求的过程如图 2-5 所示。

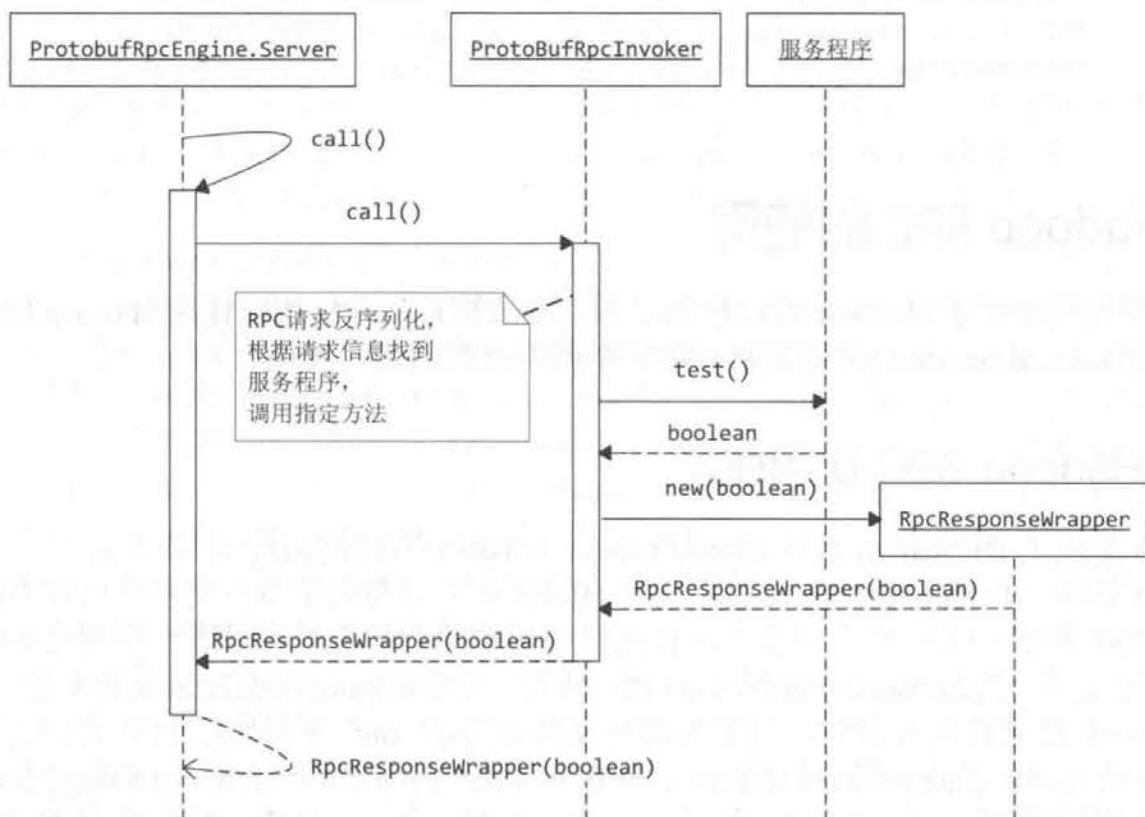


图 2-5 服务器端响应流程

那么，服务器端如何获取这个 Server 对象呢？服务器端会调用 `RPC.Builder.build()` 方法获取 Server 对象的引用，`build()` 方法是一个工厂方法，它首先检查工厂的参数是否设置了，然后会调用 `getProtocolEngine()` 方法获取当前 RPC 类配置的 `RpcEngine`，最后调用 `RpcEngine.getServer()` 获取使用特定序列化框架的 Server 类，例如 `ProtobufRpcEngine.getServer()` 会返回 `ProtobufRpcEngine.Server` 类，`ProtobufRpcEngine.Server` 就会使用 `ProtoBufRpcInvoker` 作为服务器的 Stub 代理程序。`RpcEngine.getServer()` 方法的实现请参考服务器获取 Server 对象小节。

```
public Server build() throws IOException, HadoopIllegalArgumentException {
    // 检查参数
    if (this.conf == null) {
        throw new HadoopIllegalArgumentException("conf is not set");
    }
    if (this.protocol == null) {
        throw new HadoopIllegalArgumentException("protocol is not set");
    }
    if (this.instance == null) {
        throw new HadoopIllegalArgumentException("instance is not set");
    }

    // 调用 RpcEngine.getServer() 获取 Server 对象
    return getProtocolEngine(this.protocol, this.conf).getServer(
        this.protocol, this.instance, this.bindAddress, this.port,
        this.numHandlers, this.numReaders, this.queueSizePerHandler,
        this.verbose, this.conf, this.secretManager, this.portRangeConfig);
}
```

2.2 Hadoop RPC 的使用

上一节我们介绍了 Hadoop RPC 框架结构，本节则重点介绍如何使用 Hadoop RPC，同时会以 `ClientProtocol.rename()` 调用为例介绍整个调用流程的代码实现。

2.2.1 Hadoop RPC 使用概述

图 2-6 给出了 `DFSClient` 调用 `ClientProtocol.rename()` 方法的流程图。我们首先看一下 RPC 协议的定义部分。`ClientProtocol` 协议定义了 HDFS 客户端与名字节点交互的所有方法，但是 `ClientProtocol` 协议中方法的参数是无法在网络中传输的，需要对参数进行序列化操作，所以 HDFS 又定义了 `ClientNamenodeProtocolPB` 协议。`ClientNamenodeProtocolPB` 协议包含了 `ClientProtocol` 定义的所有方法，但是参数却是使用 `protobuf` 序列化后的格式。这里还是以 `rename()` 方法为例，`ClientNamenodeProtocolPB` 将 `ClientProtocol` 中 `rename(String, String)` 方法的两个参数抽象成一个 `RenameRequestProto` 对象，`rename()` 方法的签名也就变成了 `rename(RenameRequestProto)`。这里的 `RenameRequestProto` 对象是通过 `protobuf` 序列化后的对

象，是可以在网络上传输的对象。

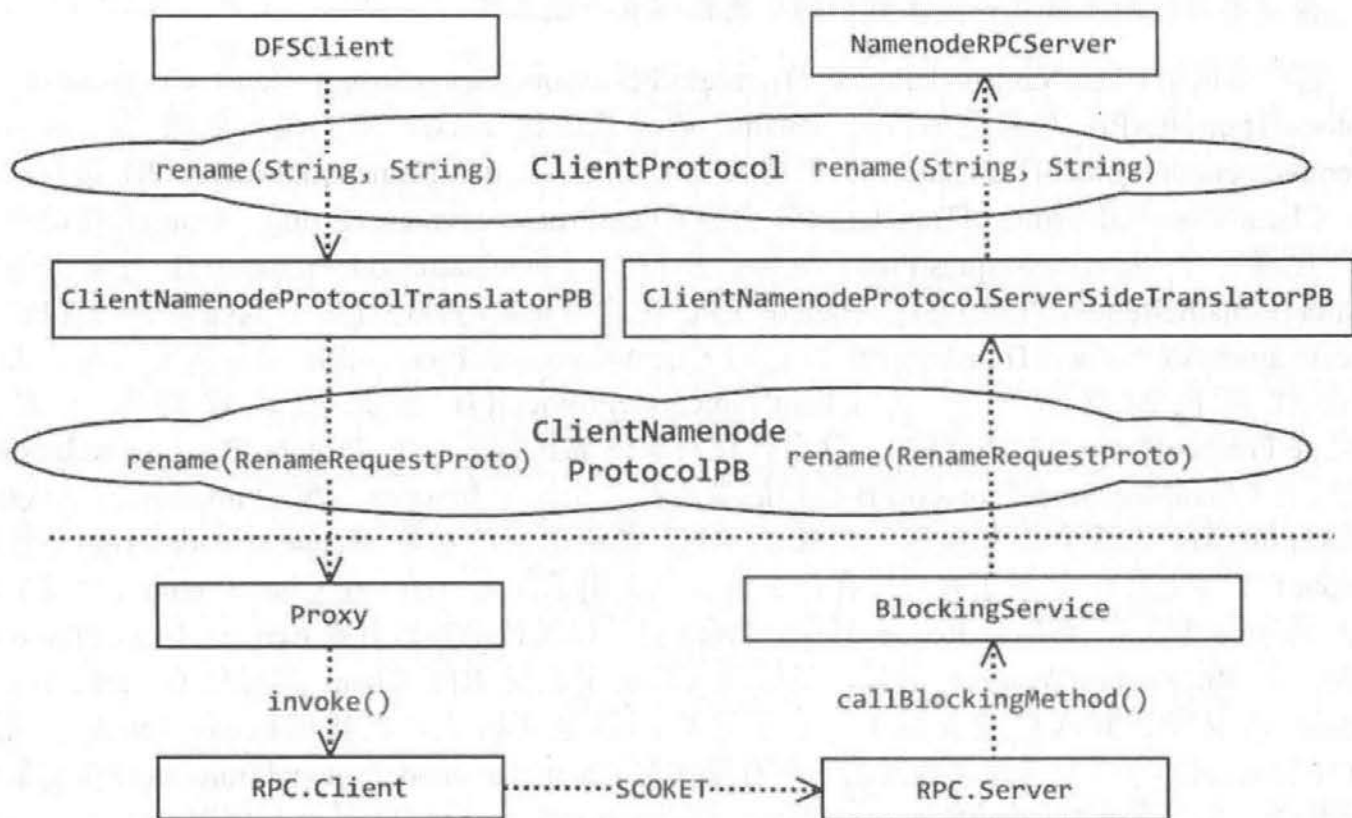


图 2-6 ClientProtocol.rename()调用流程图

了解了这两个接口之后，我们再来看 rename()调用就比较简单了。为了将不可以序列化的 ClientProtocol 接口调用转换为可以序列化的 ClientNameNodeProtocolPB 接口调用，HDFS 引入了两个适配器类（使用适配器模式的类）进行接口适配。

- ClientNameNodeProtocolTranslatorPB 类：作为 Client 侧的适配器类，实现了 ClientProtocol 接口，它内部拥有一个实现了 ClientNameNodeProtocolPB 接口的对象，可以将 ClientProtocol 调用适配成 ClientNameNodeProtocolPB 调用。以 rename()调用为例，ClientNameNodeProtocolTranslatorPB 将 rename(String, String)调用中的两个 String 参数序列化成一个 RenameRequestProto 对象，然后调用 ClientNameNodeProtocolPB 对象的 rename(RenameRequestProto)方法，这样就完成了 ClientProtocol 接口到 ClientNameNodeProtocolPB 接口的适配。
- ClientNameNodeProtocolServerSideTranslatorPB 类：作为 Server 侧的适配器类，实现了 ClientNameNodeProtocolPB 接口，同时内部拥有一个实现了 ClientProtocol 接口的对象（NameNodeRpcServer），可以将 ClientNameNodeProtocolPB 调用适配成 ClientProtocol 调用。还是以 rename()调用为例，ClientNameNodeProtocolServerSideTranslatorPB 将 rename(RenameRequestProto)调用的 RenameRequestProto 对象反序列化成两个 String 对象，之后调用 NameNodeRpcServer（实现了 ClientProtocol）类的 rename(String, String)方法执行重命名操作。

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

客户端调用 `ClientNamenodeProtocolTranslatorPB.rename(String,String)` 方法后, `ClientNamenodeProtocolTranslatorPB` 对象是如何将 `rename` 请求发送到 Server 端的呢? 如图 2-7 所示, `ClientNamenodeProtocolTranslatorPB` 类持有一个实现了 `ClientNamenodeProtocolPB` 接口的对象, `ClientNamenodeProtocolTranslatorPB` 会将 `ClientProtocol.rename(String, String)` 调用的参数序列化成一个 `RenameRequestProto` 对象, 然后在 `ClientNamenodeProtocolPB` 对象上调用 `rename(RenameRequestProto)` 方法。Hadoop RPC 在这里非常巧妙地使用了 Java 动态代理机制, `ClientNamenodeProtocolTranslatorPB` 持有的 `ClientNamenodeProtocolPB` 对象其实是通过 Java 动态代理机制获取的一个 `ClientNamenodeProtocolPB` 接口的代理对象 (调用 `RPC.getProtocolProxy()` 方法获取), 这个代理对象内部封装了一个 `ProtobufRpcEngine.Invoker` 对象。对 `ClientNamenodeProtocolPB` 接口的调用都会由这个 `Invoker` 对象的 `invoke()` 方法代理, `Invoker.invoke()` 方法会首先构造一个描述 RPC 调用信息的对象 `RequestHeaderProto` (使用 `protobuf` 序列化的), 记录了客户端在什么协议上调用了什么方法 (在 `ClientProtocol` 协议上调用了 `rename` 方法), 然后将 `RequestHeaderProto` 对象以及调用参数对象 `RenameRequestProto` 包装成一个 `RpcRequestWrapper` 对象, 最后就可以调用底层 `RPC.Client` 类提供的 `call()` 方法将 `rename` 请求发送到远程服务器了。由于这里巧妙地使用了动态代理机制, `Invoker` 调用 `RPC.Client.call()` 方法发送请求到远端的所有操作对 `ClientNamenodeProtocolTranslatorPB` 对象都是透明的, 客户端调用 `ClientNamenodeProtocolTranslatorPB.rename()` 方法就与调用本地方法一样。

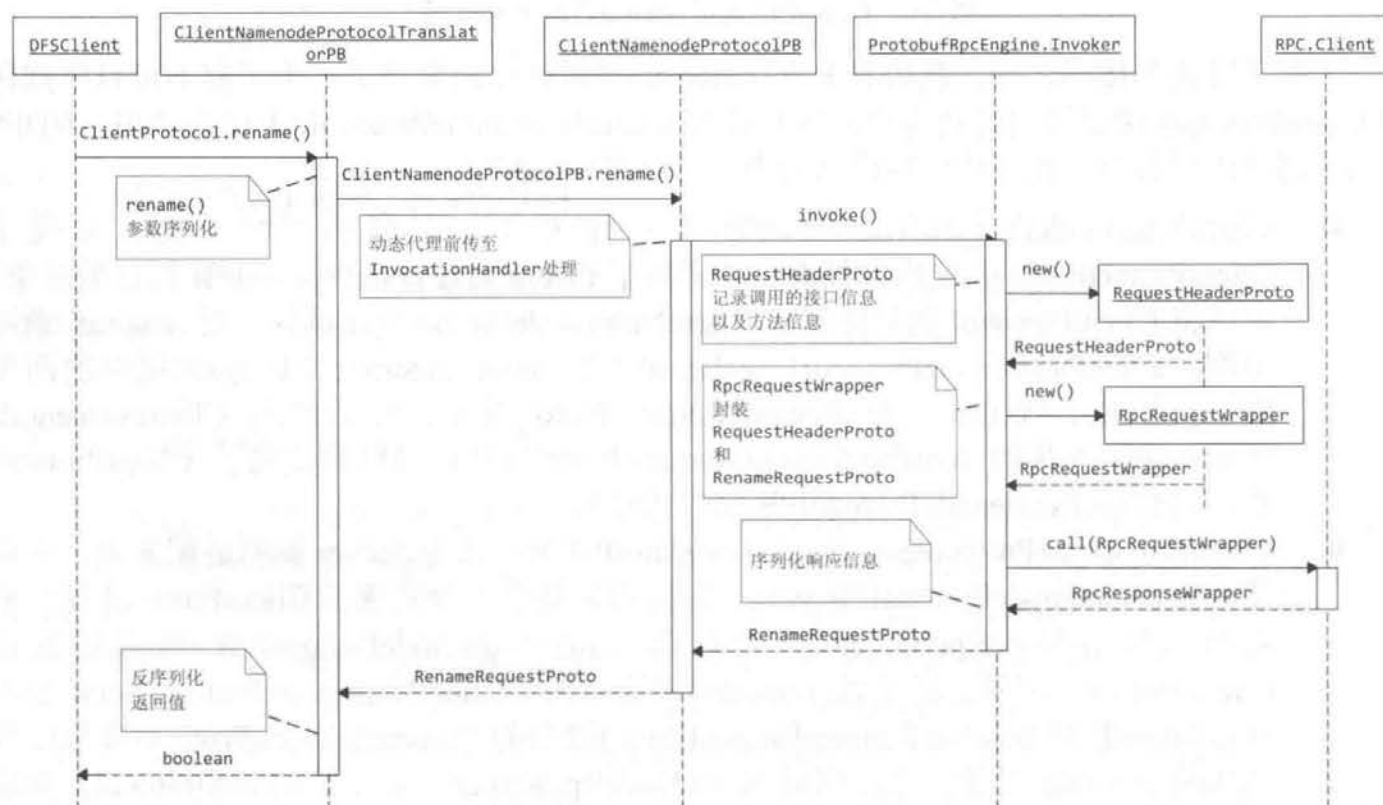


图 2-7 客户端发送 rename 请求流程

那么,当 rename 请求到达服务器端之后,服务器端是如何响应的呢?如图 2-8 所示,NameNodeRpcServer 会启动一个 RPC Server 监听来自客户端的所有 RPC 请求,当 RPC Server 在网络上监听到一个 RPC 请求时,它会从网络中解析这个请求,然后构造一个 ProtoBufRpcInvoker 对象来处理这个请求。ProtoBufRpcInvoker 对象会将请求数据反序列化,解析出调用信息和调用参数。之后 ProtoBufRpcInvoker 对象会根据调用信息中的调用接口信息(在这个例子中是 ClientProtocol)查找实现了 ClientProtocol 接口的 BlockingService 对象,这个 BlockingService 对象持有一个实现了 ClientNamenodeProtocolPB 接口的 ClientNamenodeProtocolServerSideTranslatorPB 对象,用于响应 ClientNamenodeProtocolPB 接口上的调用。获取了 BlockingService 对象后,ProtoBufRpcInvoker 利用调用信息中的调用方法信息(在这个例子中是 rename)和调用参数对象(RenameRequestProto)调用 BlockingService.callBlockingMethod()方法响应 RPC 请求。callBlockingMethod()方法根据调用方法信息判断这是一个 ClientNamenodeProtocolPB.rename()调用,它会在 ClientNamenodeProtocolServerSideTranslatorPB 对象上调用 rename(RenameRequestProto)方法响应,ClientNamenodeProtocolServerSideTranslatorPB.rename()方法会将 RenameRequestProto 参数反序列化成两个 String 参数,然后在自己持有的实现了 ClientProtocol 接口的 NameNodeRpcServer 上调用 rename(String,String)方法响应,NameNodeRpcServer.rename()方法会在 Namenode 的命名空间中更改指定 HDFS 文件的名称,最后返回响应信息。ClientNamenodeProtocolServerSideTranslatorPB 接收到 NameNodeRpcServer 的响应信息后会将这个响应包装成一个 protobuf 序列化的 RenameResponseProto 对象,然后返回到 ProtoBufRpcInvoker 对象。ProtoBufRpcInvoker 接收到 RenameResponseProto 这个响应对象后,由于 ProtoBufRpcInvoker.call()方法的返回值定义是 Writable 类型的,所以 ProtoBufRpcInvoker 会构造一个 RpcResponseWrapper 对象包装 RenameResponseProto,然后将这个对象返回给 RPC Client。

学习了 ClientProtocol.rename()调用的流程后,我们可以将 Hadoop RPC 框架的使用抽象为如下几个步骤。

- 定义 RPC 协议:RPC 协议是客户端和服务端之间 RPC 调用的接口,只有定义了 RPC 协议,客户端才知道服务器对外提供了哪些服务。这里以 ClientProtocol 为例,ClientProtocol 定义了 Namenode 服务器与 HDFS 客户端之间的接口,HDFS 客户端调用 ClientProtocol.rename()方法,Namenode 服务器就会更改指定 HDFS 文件的文件名。
- 实现 RPC 协议:服务器端的服务程序需要实现 RPC 协议,当 RPC 调用通过网络到达服务器时,实现了 RPC 协议的服务程序会响应这个 RPC 调用。还是以 ClientProtocol 为例,Namenode 端的 NameNodeRpcServer 类实现了 ClientProtocol 协议,当 ClientProtocol RPC 请求到达 Namenode 时,会由 NameNodeRpcServer 类响应这个 RPC 请求。
- 客户端获取代理对象:客户端需要调用 RPC.getProtocolProxy()方法获取一个 RPC 协议的代理对象,之后客户端调用程序就可以在这个代理对象上调用 RPC 协议的方法,通过 Java 动态代理机制,这个 RPC 请求会由一个 InvocationHandler 代理对象处理,InvocationHandler 对象会将 RPC 调用信息和调用参数序列化,最后通过调用 Client.call()方法将这个请求发送到远程服务器。

- 服务器构造并启动 RPC Server: 服务器需要调用 `RPC.Builder.build()` 方法构造一个 `Server` 对象, 然后调用 `Server.start()` 方法启动这个 `Server` 对象响应来自客户端的 RPC 请求。例如对于 `NameNodeRpcServer` 类, 它会构造两个 `Server` 对象分别响应来自 HDFS 客户端和数据节点的 RPC 请求。

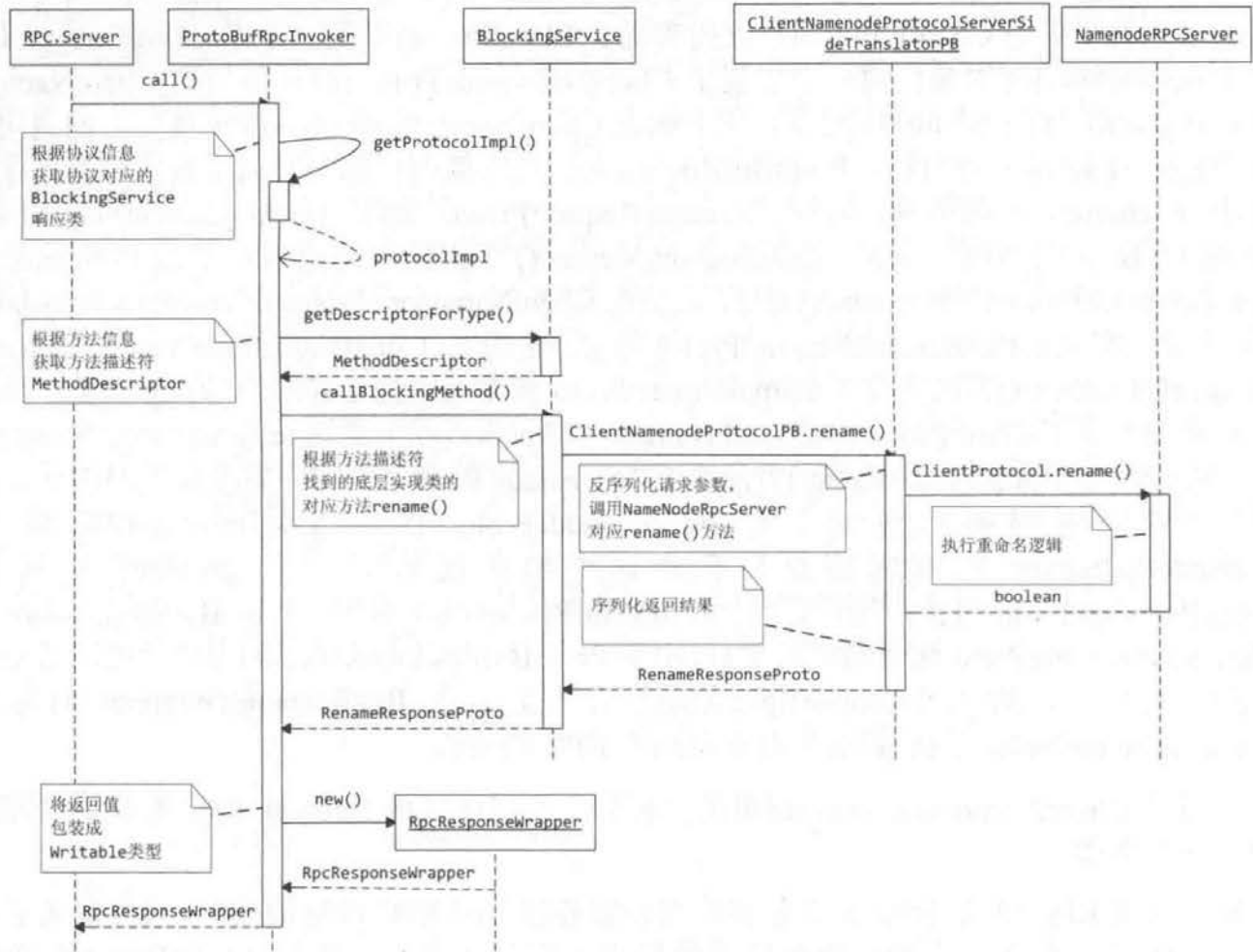


图 2-8 客户端响应 rename 请求流程

2.2.2 定义 RPC 协议

在第 1 章的 HDFS 概述小节中我们介绍了 HDFS 为各个节点之间抽象了不同的接口, 也就是 RPC 协议, 例如 `ClientProtocol` 抽象了客户端与名字节点之间的通信协议, 而 `DatanodeProtocol` 则抽象了数据节点与名字节点之间的通信协议。通过上一节的介绍, 我们知道定义 RPC 协议是使用 Hadoop RPC 框架的第一步, 而对于 HDFS 中的所有 RPC 协议, 都会存在一个如图 2-6 所示的协议栈。例如对于 `DatanodeProtocol`, 存在对应的支持序列化的接口 `DatanodeProtocolPB`, 同时还存在 `DatanodeProtocolClientSideTranslatorPB` 以及 `DatanodeProtocolServerSideTranslatorPB` 两个类用于进行协议适配, 读者对比理解即可。本节还是以 `ClientProtocol` 作为例子来介绍这些支持类的代码实现。

1. ClientProtocol 协议

ClientProtocol 协议定义了 HDFS 客户端与名字节点交互的所有接口方法（例如 rename()、mkdir()等），ClientProtocol 定义的具体方法我们在 HDFS 概述小节中已经介绍过了，本节主要介绍 ClientProtocol 的子类。ClientProtocol 的类结构如图 2-9 所示，定义了两个子类。

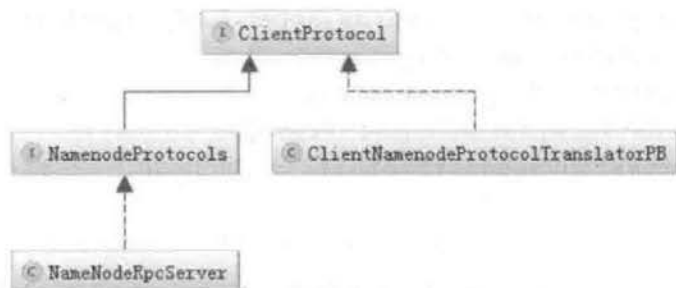


图 2-9 ClientProtocol 继承结构图

- ClientNamenodeProtocolTranslatorPB——这个类是 RPC 客户端侧最重要的类之一，它将客户端的请求参数封装成可以序列化的 protobuf 格式，然后通过代理类（实现 ClientNamenodeProtocolPB 接口）发送出去。
- NameNodeRpcServer——Namenode 侧响应 ClientProtocol 调用的类，它会执行 HDFS 操作并将操作结果返回。

2. ClientNamenodeProtocolPB 协议

我们知道 ClientProtocol 是 HDFS 客户端与名字节点之间的接口，但是在 RPC 调用中，RPC 方法的参数和返回值都必须是可序列化的，所以 HDFS 定义了一个新的 ClientNamenodeProtocolPB 协议对 ClientProtocol 协议中方法的参数和返回值进行包装。ClientNamenodeProtocolPB 协议定义的方法与 ClientProtocol 一样，但是参数和返回值却是使用 protobuf 序列化后的格式，这样 ClientNamenodeProtocolPB 请求就可以通过网络传输。为了便于理解，大家可以把 ClientNamenodeProtocolPB 理解为使用 protobuf 定义参数和返回值的 ClientProtocol 协议。

那么如何定义 ClientNamenodeProtocolPB 协议呢？Hadoop RPC 在这里使用了 protobuf 工具。使用 protobuf 工具的第一步是定义一个 ClientNamenodeProtocol.proto 文件，这个文件中包含了 RPC 协议 ClientNamenodeProtocol 的定义，ClientNamenodeProtocol 是 protobuf 产生的支持 ClientProtocol RPC 调用的协议接口类，ClientNamenodeProtocolPB 是 ClientNamenodeProtocol.BlockingInterface 接口的子类。ClientNamenodeProtocol.proto 文件中 ClientNamenodeProtocol 协议的定义如下代码所示，可以看到 ClientNamenodeProtocol 就是将 ClientProtocol 的参数全部抽象为一个 *RequestProto 对象，而将返回值抽象为一个 *ResponseProto 对象。这里还是以 rename() 方法为例，ClientNamenodeProtocol.rename() 方法的参数是 RenameRequestProto，而返回值是 RenameResponseProto。RenameRequestProto 和 RenameResponseProto 是 protobuf 定义的用于封装参数以及返回值的类，它们的结构也是在 ClientNamenodeProtocol.proto 文件中定义的。

```

service ClientNamenodeProtocol {
  rpc getBlockLocations(GetBlockLocationsRequestProto)
    returns (GetBlockLocationsResponseProto);
  rpc getServerDefaults(GetServerDefaultsRequestProto)
    returns (GetServerDefaultsResponseProto);
  rpc create(CreateRequestProto) returns (CreateResponseProto);
  rpc append(AppendRequestProto) returns (AppendResponseProto);
  rpc setReplication(SetReplicationRequestProto)
    returns (SetReplicationResponseProto);
  rpc setPermission(SetPermissionRequestProto)
    returns (SetPermissionResponseProto);
  // ...
}

```

ClientNamenodeProtocol.proto 文件中定义的第二部分是调用过程中的请求参数和响应参数的声明，也就是上面描述的 RenameRequestProto 和 RenameResponseProto 的声明（注意这两个类是写好声明之后，执行 protoc.exe 动态生成的），RenameRequestProto 和 RenameResponseProto 对象都是可以通过 protobuf 工具序列化的。如下代码所示，RenameRequestProto 的定义中包含了 ClientProtocol.rename() 方法的两个 String 类型的参数 (src&dst)，RenameResponseProto 则包含了 ClientProtocol.rename() 方法的 bool 类型的返回值（重命名操作的执行结果）。

```

message RenameRequestProto {
  required string src = 1; // rename() 方法的第一个参数 src
  required string dst = 2; // rename() 方法的第二个参数 dst
}

message RenameResponseProto {
  required bool result = 1; // 封装返回结果，布尔类型
}

```

在 ClientNamenodeProtocol.proto 文件中定义了 ClientNamenodeProtocol 后，就可以使用 protobuf 工具生成 ClientNamenodeProtocol 类了，生成的 ClientNamenodeProtocol.BlockingInterface 接口的代码如下所示。我们注意到 BlockingInterface 接口中定义的所有方法都只有两个参数。

- controller 参数：这个参数没有用到，方法调用时设置为 Null。
- request 参数：封装原有方法调用的参数，例如 ClientProtocol.rename(String, String) 方法有两个 String 类型的参数 (src & dst)，在 ClientNamenodeProtocol.BlockingInterface 接口中这两个参数是用 RenameRequestProto 封装的，RenameRequestProto 是可以通过 protobuf 序列化的类，结构是在 ClientNamenodeProtocol.proto 文件中定义的。

```

public static abstract class ClientNamenodeProtocol
  implements com.google.protobuf.Service {
  protected ClientNamenodeProtocol() {}
  // ...
  public interface BlockingInterface {
    public RenameResponseProto rename(

```

```

        com.google.protobuf.RpcController controller,
        RenameRequestProto request)
        throws com.google.protobuf.ServiceException;

    public CompleteResponseProto complete(
        com.google.protobuf.RpcController controller,
        CompleteRequestProto request)
        throws com.google.protobuf.ServiceException;

    public ReportBadBlocksResponseProto reportBadBlocks(
        com.google.protobuf.RpcController controller,
        ReportBadBlocksRequestProto request)
        throws com.google.protobuf.ServiceException;

    public ConcatResponseProto concat(
        com.google.protobuf.RpcController controller,
        ConcatRequestProto request)
        throws com.google.protobuf.ServiceException;
    // ...
}

```

ClientNamenodeProtocolPB 接口是 ClientNamenodeProtocol.BlockingInterface 接口的子类，它完全继承了 ClientNamenodeProtocol.BlockingInterface 接口中方法的定义。读者可能会问，这里直接使用 ClientNamenodeProtocol 作为序列化协议就可以啊，为什么还要再定义一个 ClientNamenodeProtocolPB 协议呢？之所以使用 ClientNamenodeProtocolPB 作为序列化协议类是为了添加 Annotation 支持，因为 ClientNamenodeProtocol 的代码是 protobuf 自动生成的，不可以添加 Annotation 支持。ClientNamenodeProtocolPB 接口的定义如下：

```

@interfaceAudience.Private
@interfaceStability.Stable
@KerberosInfo(
    serverPrincipal = DFSConfigKeys.DFS_NAMENODE_KERBEROS_PRINCIPAL_KEY)
@TokenInfo(DelegationTokenSelector.class)
@ProtocolInfo(protocolName = HdfsConstants.CLIENT_NAMENODE_PROTOCOL_NAME,
    protocolVersion = 1)
// ClientNamenodeProtocolPB 是 ClientNamenodeProtocol.BlockingInterface 的子类
// ClientNamenodeProtocol 则是由 protobuf 工具生成的 RPC 调用接口类
public interface ClientNamenodeProtocolPB extends
    ClientNamenodeProtocol.BlockingInterface {
}

```

3. ClientNamenodeProtocolTranslatorPB 类

如图 2-6 所示，DFSClient 会持有一个 ClientNamenodeProtocolTranslatorPB 类的引用，用于将 ClientProtocol 协议上的请求适配成 ClientNamenodeProtocolPB 协议的请求。如下代码所示，ClientNamenodeProtocolTranslatorPB 类实现了 ClientProtocol 接口，并持有一个实现了

ClientNamenodeProtocolPB 接口的 rpcProxy 对象。

```
// ClientNamenodeProtocolTranslatorPB 实现了 ClientProtocol 接口
public class ClientNamenodeProtocolTranslatorPB implements
    ProtocolMetaInterface, ClientProtocol, Closeable, ProtocolTranslator {
    // 持有一个实现 ClientNamenodeProtocolPB 接口的 rpcProxy 对象
    final private ClientNamenodeProtocolPB rpcProxy;
    // ...
}
```

ClientNamenodeProtocolTranslatorPB 会将 ClientProtocol 请求的参数序列化，然后调用 rpcProxy 对象（实现了 ClientNamenodeProtocolPB 接口）上的对应方法，这样就完成了适配操作。我们还是以 rename() 方法调用作为例子，如下代码所示，ClientNamenodeProtocolTranslatorPB 执行了两个操作来完成 ClientProtocol.rename() 调用的适配功能。

- 将 ClientProtocol.rename() 调用的两个 String 类型的参数封装成 RenameRequestProto 对象。
- 调用 ClientNamenodeProtocolPB.rename() 方法，然后反序列化调用结果并返回。

```
public boolean rename(String src, String dst) throws UnresolvedLinkException,
    IOException {
    // 构建 pb 参数
    RenameRequestProto req = RenameRequestProto.newBuilder()
        .setSrc(src)
        .setDst(dst).build();
    try {
        // 调用底层 impl 对应方法，返回结果
        return rpcProxy.rename(null, req).getResult();
    } catch (ServiceException e) {
        throw ProtobufHelper.getRemoteException(e);
    }
}
```

至于这里的 rpcProxy 对象（实现了 ClientNamenodeProtocolPB 接口），则是使用 Java 动态代理机制获取的 ClientNamenodeProtocolPB 接口的代理对象。在这个对象上的调用会由 ProtobufRpcEngine.Invoker 对象代理，这个 Invoker 对象的 invoke() 方法会调用底层的 RPC.Client 类提供的 call() 方法将 ClientNamenodeProtocolPB.rename() 请求发送到远程服务器，并等待远程服务器返回响应信息。获取 ClientNamenodeProtocolPB 接口代理对象的流程我们将在客户端获取 Proxy 对象小节中详细介绍。

4. ClientNamenodeProtocolServerSideTranslatorPB 类

ClientNamenodeProtocolTranslatorPB 类是 RPC Client 侧的适配器类，ClientNamenodeProtocolServerSideTranslatorPB 则是 RPC Server 侧的适配器类。Client 侧的适配器用于将 ClientProtocol 的请求转换为 ClientNamenodeProtocolPB 请求，那么 Server 侧的适配器则是将 ClientNamenodeProtocolPB 请求转换为 ClientProtocol 请求。如下代码所示，ClientNamenodeProtocolServerSideTranslatorPB 实现了 ClientNamenodeProtocolPB 接口，底层持有一个实现了

ClientProtocol 接口的 server 对象。

```
public class ClientNameNodeProtocolServerSideTranslatorPB implements
    ClientNameNodeProtocolPB {
    // 底层实现类
    final private ClientProtocol server;
    // ...
}
```

ClientNameNodeProtocolServerSideTranslatorPB 只需将 ClientNameNodeProtocolPB 请求的参数反序列化, 然后在 ClientProtocol 服务对象 server 上调用对应的方法即可完成适配工作。这里还是以 rename() 方法为例, ClientNameNodeProtocolServerSideTranslatorPB.rename() 会首先反序列化请求参数 RenameRequestProto 对象, 然后调用 server 对象的 rename() 方法执行重命名操作, 最后将返回值序列化为 RenameResponseProto 对象并返回。

```
public RenameResponseProto rename(RpcController controller,
    RenameRequestProto req) throws ServiceException {
    try {
        boolean result = server.rename(req.getSrc(), req.getDst());
        return RenameResponseProto.newBuilder().setResult(result).build();
    } catch (IOException e) {
        throw new ServiceException(e);
    }
}
```

ClientNameNodeProtocolPB 持有的 server 对象实际上是 NameNodeRpcServer 的引用, 我们知道 NameNodeRpcServer 是 ClientProtocol 的子类, 它会响应 ClientProtocol 的请求, 修改名字节点的命名空间, 执行对应的 HDFS 逻辑。

2.2.3 客户端获取 Proxy 对象

我们知道 DFSClient 会持有一个 ClientProtocol 对象向 NameNode 发送请求, 在 Hadoop RPC 框架概述小节中我们介绍了 DFSClient 持有的 ClientProtocol 对象其实是 ClientNameNodeProtocolTranslatorPB 的实例 (ClientNameNodeProtocolTranslatorPB 实现了 ClientProtocol 接口)。在上一节中我们介绍了 ClientNameNodeProtocolTranslatorPB 对象会持有一个实现了 ClientNameNodeProtocolPB 接口的对象, 这个对象是通过 Java 动态代理机制获取的 ClientNameNodeProtocolPB 接口的代理对象。

接下来 ClientNameNodeProtocolTranslatorPB 对象会将 ClientProtocol 调用的参数使用 protobuf 序列化, 然后调用 ClientNameNodeProtocolPB 接口代理类的对应方法将这个请求发送出去。在 ClientNameNodeProtocolPB 代理对象上的调用都会由一个 ProtobufRpcEngine.Invoker 对象代理, Invoker 对象会构造请求头 (在什么接口上调用什么方法), 然后调用 RPC.Client.call() 方法将请求头以及序列化之后的参数发送到服务器。本节就重点介绍 DFSClient 获取 ClientNameNodeProtocolTranslatorPB 对象, 以及 ClientNameNodeProtocolTranslatorPB 对象获取 ClientNameNodeProtocolPB 代理对象的流程。

我们先从一个简单的 ClientRPC 请求 (DFSCClient.rename()) 开始分析, 如下代码所示, DFSCClient 会在 namenode 这个引用上调用 rename2() 方法。

```
public void rename(String src, String dst, Options.Rename... options)
    throws IOException {
    checkOpen();
    try {
        namenode.rename2(src, dst, options);
    } catch (RemoteException re) {
        // ...
    }
}
```

这里的 namenode 字段保存了一个实现了 ClientProtocol 接口的对象, DFSCClient 通过 ProxyInfo 类来获取这个对象的引用, 而这个 ProxyInfo 对象则是通过调用 NameNodeProxies.createProxy() 方法产生的。

```
final ClientProtocol namenode;
// 构造方法
this.namenode = proxyInfo.getProxy();
// 获取 proxyInfo 引用
proxyInfo = NameNodeProxies.createProxy(conf, nameNodeUri, ClientProtocol.class);
```

我们知道 ClientProtocol 对象定义了客户端与名字节点之间的所有接口, 而 Hadoop 2.X 引入了 Namenode 的 HA 机制, 也就是说, HDFS 集群中会存在两个 Namenode 实例, 同一时间 DFSCClient 只会将 ClientProtocol RPC 请求发送给集群中的 Active Namenode。而当集群发生错误切换时, DFSCClient 又会将请求发送给新的 Active Namenode, 这些实现对于 DFSCClient 来说是透明的, DFSCClient 并不知道 ClientProtocol RPC 请求发送到了哪个 Namenode, 它只需在 ClientProtocol 对象上发起 RPC 调用即可。NameNodeProxies.createProxy() 方法就是用于创建支持 HA 机制的 ClientProtocol 代理对象的, 它会根据配置文件判断当前 HDFS 集群是否处于 HA 模式。对于处于 HA 模式的情况, createProxy() 方法会调用 createFailoverProxyProvider() 方法创建支持 HA 机制的 ClientProtocol 对象; 而对于非 HA 模式的情况, createProxy() 方法则会调用 createNonHAProxy() 方法创建普通的 ClientProtocol 对象。NameNodeProxies.createProxy() 方法的代码如下:

```
public static <T> ProxyAndInfo<T> createProxy(Configuration conf,
    URI nameNodeUri, Class<T> xface) throws IOException {
    Class<FailoverProxyProvider<T>> failoverProxyProviderClass =
        getFailoverProxyProviderClass(conf, nameNodeUri, xface);

    if (failoverProxyProviderClass == null) {
        // 非 HA 情况
        return createNonHAProxy(conf, NameNode.getAddress(nameNodeUri), xface,
            UserGroupInformation.getCurrentUser(), true);
    } else {
        // HA 情况
        FailoverProxyProvider<T> failoverProxyProvider = NameNodeProxies
```

```

        .createFailoverProxyProvider(conf, failoverProxyProviderClass, xface,
            nameNodeUri);
    Conf config = new Conf(conf);
    T proxy = (T) RetryProxy.create(xface, failoverProxyProvider, RetryPolicies
        .failoverOnNetworkException(RetryPolicies.TRY_ONCE_THEN_FAIL,
            config.maxFailoverAttempts, config.failoverSleepBaseMillis,
            config.failoverSleepMaxMillis));

    Text dtService = HAUtil.buildTokenServiceForLogicalUri(nameNodeUri);
    return new ProxyAndInfo<T>(proxy, dtService);
}
}

```

下面我们就分析 HA 以及非 HA 情况时的 ClientProtocol 对象的构造流程。

1. 非 HA 模式

非 HA 模式的入口方法是 NameNodeProxies.createNonHAProxy(), 它会对 xface 参数也就是 RPC 接口进行判断, 然后构造并返回实现了这个接口的对象。对于获取 ClientProtocol 代理的情况, createNonHAProxy 会调用 createNNProxyWithClientProtocol() 方法创建实现了 ClientProtocol 接口的 ClientNamenodeProtocolTranslatorPB 对象。createNonHAProxy() 方法的代码如下:

```

public static <T> ProxyAndInfo<T> createNonHAProxy(
    Configuration conf, InetSocketAddress nnAddr, Class<T> xface,
    UserGroupInformation ugi, boolean withRetries) throws IOException {
    Text dtService = SecurityUtil.buildTokenService(nnAddr);

    T proxy;
    // 当前接口是 ClientProtocol, 调用对应的方法
    if (xface == ClientProtocol.class) {
        proxy = (T) createNNProxyWithClientProtocol(nnAddr, conf, ugi,
            withRetries);
    } else if (xface == JournalProtocol.class) { // 当前接口是 JournalProtocol
        proxy = (T) createNNProxyWithJournalProtocol(nnAddr, conf, ugi);
    }
    // 其他协议接口 ...
}

```

由于我们一直是以 ClientProtocol 为例子的, 这里我们就看一下 createNNProxyWithClientProtocol() 方法的实现。createNNProxyWithClientProtocol() 方法首先会设置 RPC.protocolEngine 字段为 ProtobufRpcEngine, RPC.protocolEngine 字段用于指定当前 RPC 调用使用什么序列化方式, 这里配置的 ProtobufRpcEngine 类就定义了当前 RPC 调用是使用 protobuf 作为序列化引擎的。之后 createNNProxyWithClientProtocol() 方法会调用 RPC.getProtocolProxy() 方法获取 ClientNamenodeProtocolPB 协议的代理对象, 然后构造 ClientNamenodeProtocolTranslatorPB 对象并返回。

createNNProxyWithClientProtocol()方法的代码如下:

```
private static ClientProtocol createNNProxyWithClientProtocol(
    InetAddress address, Configuration conf, UserGroupInformation ugi,
    boolean withRetries, AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    // 设置 protocolEngine
    RPC.setProtocolEngine(conf, ClientNamenodeProtocolPB.class, ProtobufRpcEngine.class);
    // ...
    // 构造 ClientNamenodeProtocolPB 代理对象
    ClientNamenodeProtocolPB proxy = RPC.getProtocolProxy(
        ClientNamenodeProtocolPB.class, version, address, ugi, conf,
        NetUtils.getDefaultSocketFactory(conf),
        org.apache.hadoop.ipc.Client.getTimeout(conf), defaultPolicy,
        fallbackToSimpleAuth).getProxy();
    // ...
    // 构造 ClientNamenodeProtocolTranslatorPB 对象并返回
    // 注意 ClientNamenodeProtocolTranslatorPB 会持有一个 ClientNamenodeProtocolPB 对象
    return new ClientNamenodeProtocolTranslatorPB(proxy);
}
```

图 2-10 总结了 createNNProxyWithClientProtocol()调用的全部流程。

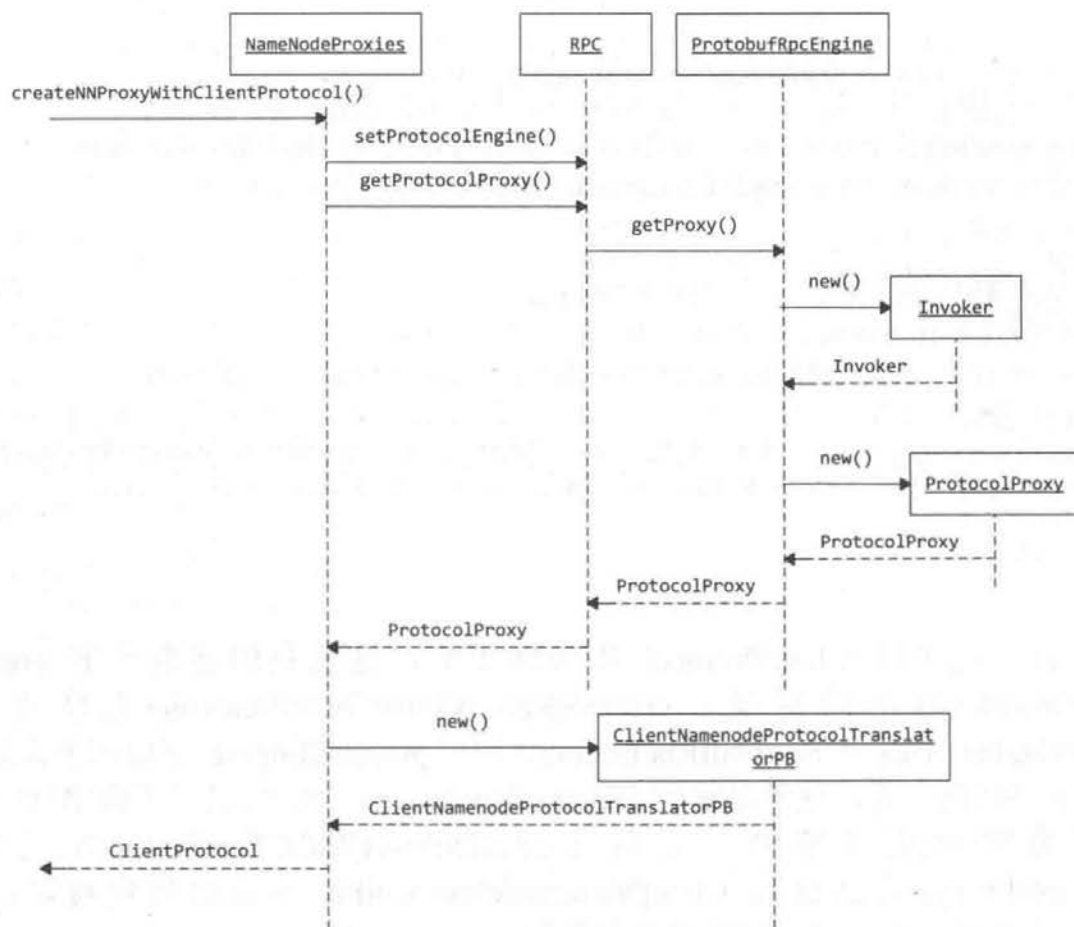


图 2-10 createNNProxyWithClientProtocol()调用流程图

通过调用 `createNNProxyWithClientProtocol()` 方法，我们就成功地获取了一个实现了 `ClientProtocol` 接口的 `ClientNamenodeProtocolTranslatorPB` 对象。下面我们重点看一下获取 `ClientNamenodeProtocolPB` 代理对象的 `RPC.getProtocolProxy()` 方法的实现，这个方法是 `RPC` 类中最重要的一个方法，用于获取一个指定 `RPC` 接口的代理对象。`getProtocolProxy()` 会首先调用 `getProtocolEngine()` 方法获取当前 `RPC` 定义的 `protocolEngine` 对象的实例，然后在这个 `protocolEngine` 对象上调用 `getProxy()` 获取使用特定序列化方式的接口代理对象。`getProtocolProxy()` 方法的代码如下：

```
public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,
                                                    long clientVersion,
                                                    InetSocketAddress addr,
                                                    UserGroupInformation ticket,
                                                    Configuration conf,
                                                    SocketFactory factory,
                                                    int rpcTimeout,
                                                    RetryPolicy connectionRetryPolicy,
                                                    AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    return getProtocolEngine(protocol, conf).getProxy(protocol, clientVersion,
                                                    addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy,
                                                    fallbackToSimpleAuth);
}
```

我们看一下 `getProtocolEngine()` 方法的实现。还是以 `ClientProtocol` 为例，由于 `createNNProxyWithClientProtocol()` 的第一条语句就是注册 `ClientProtocol` 的序列化引擎为 `ProtobufRpcEngine`，所以 `ClientProtocol` 使用 `protobuf` 作为序列化工具，那么这里的 `getProtocolEngine()` 方法返回的就是一个 `ProtobufRpcEngine` 的实例。

```
static synchronized RpcEngine getProtocolEngine(Class<?> protocol,
                                                Configuration conf) {
    RpcEngine engine = PROTOCOL_ENGINES.get(protocol);
    // 通过反射创建 rpcEngine 实例
    if (engine == null) {
        Class<?> impl = conf.getClass(ENGINE_PROP+"."+protocol.getName(),
                                    WritableRpcEngine.class);
        engine = (RpcEngine)ReflectionUtils.newInstance(impl, conf);
        PROTOCOL_ENGINES.put(protocol, engine);
    }
    return engine;
}
```

可以看到，真正构造代理对象的方法其实是 `RpcEngine.getProxy()` 方法，这里的 `RpcEngine` 是一个抽象类，它只定义了接口，具体的实现留给子类去做。`RpcEngine` 有两个重要的子类：`WritableRpcEngine` 用于描述使用 `HadoopWritable` 序列化机制的 `RPC` 引擎；而 `ProtobufRpcEngine` 则用于描述使用 `protobuf` 序列化机制的 `RPC` 引擎。之所以这样定义，是为了使 `Hadoop RPC` 可以方便地支持多种 `RPC` 序列化方式，同时在切换 `RPC` 引擎时并不需要更

改源码，只需要通过配置更改 RPC 的 RpcEngine 对象即可。我们知道 ClientProtocol 默认是使用 ProtobufRpcEngine 的，所以我们来看一下 ProtobufRpcEngine.getProxy()方法的实现。如下代码所示，getProxy()方法首先会构造一个实现了 InvocationHandler 接口的 invoker 对象（动态代理机制中的 InvocationHandler 对象会在 invoke()方法中代理所有目标接口上的调用，用户可以在 invoke()方法中添加代理操作），这个 invoker 对象是 ProtobufRpcEngine.Invoker 类型的。构造了这个对象之后，getProxy()就可以调用 Proxy.newProxyInstance()方法构造动态代理对象，然后将这个对象封装在 ProtocolProxy 对象中并返回。

```
public <T> ProtocolProxy<T> getProxy(Class<T> protocol, long clientVersion,
    InetAddress addr, UserGroupInformation ticket, Configuration conf,
    SocketFactory factory, int rpcTimeout, RetryPolicy connectionRetryPolicy,
    AtomicBoolean fallbackToSimpleAuth) throws IOException {
    // 首先构造 InvocationHandler 对象
    final Invoker invoker = new Invoker(protocol, addr, ticket, conf, factory,
        rpcTimeout, connectionRetryPolicy, fallbackToSimpleAuth);
    // 然后调用 Proxy.newProxyInstance() 获取动态代理对象，并通过 ProtocolProxy 返回
    return new ProtocolProxy<T>(protocol, (T) Proxy.newProxyInstance(
        protocol.getClassLoader(), new Class[]{protocol}, invoker), false);
}
```

由于 Java 动态代理机制决定了在代理对象上的所有调用都会由 InvocationHandler 对象的 invoke()方法代理，所以所有在 ClientNamenodeProtocolPB 代理对象上的调用都会由这个 ProtobufRpcEngine.Invoker 对象的 invoker()方法代理，ProtobufRpcEngine.Invoker.invoker()方法主要做了三件事情：①构造请求头域，使用 protobuf 将请求头序列化，这个请求头域记录了当前 RPC 调用是什么接口的什么方法上的调用；②通过 RPC.Client 类发送请求头以及序列化好的请求参数。请求参数是在 ClientNamenodeProtocolPB 调用时就已经序列化好的，调用 Client.call()方法时，需要将请求头以及请求参数使用一个 RpcRequestWrapper 对象封装；③获取响应信息，序列化响应信息并返回。ProtobufRpcEngine.Invoker 的 invoke()方法如下：

```
public Object invoke(Object proxy, Method method, Object[] args)
    throws ServiceException {
    // 通过上一节的学习，我们知道 pb 接口的参数只有两个，即 RpcController + Message
    if (args.length != 2) {
        throw new ServiceException("Too many parameters for request. Method: ["
            + method.getName() + "]" + ", Expected: 2, Actual: "
            + args.length);
    }
    if (args[1] == null) {
        throw new ServiceException("null param while calling Method: ["
            + method.getName() + "]" + ", Expected: 2, Actual: "
            + args.length);
    }

    // 构造请求头域，标明在什么接口上调用什么方法
    RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method);
    // 获取请求调用的参数，例如 RenameRequestProto
    Message theRequest = (Message) args[1];
```

```

final RpcResponseWrapper val;
try {
    // 调用 RPC.Client 发送请求
    val = (RpcResponseWrapper) client.call(RPC.RpcKind.RPC_PROTOCOL_BUFFER,
        new RpcRequestWrapper(rpcRequestHeader, theRequest), remoteId);

} catch (Throwable e) {
    // ...
    throw new ServiceException(e);
}

// ...

Message prototype = null;
try {
    // 获取返回参数类型, RenameResponseProto
    prototype = getReturnProtoType(method);
} catch (Exception e) {
    throw new ServiceException(e);
}
Message returnMessage;
try {
    // 序列化响应信息并返回
    returnMessage = prototype.newBuilderForType()
        .mergeFrom(val.theResponseRead).build();
    // ...
} catch (Throwable e) {
    throw new ServiceException(e);
}
// 返回结果
return returnMessage;
}

```

2. HA 模式

对于 HA 模式, NameNodeProxies 调用 RetryProxy.create() 方法构造实现了 RPC 协议的对象, 这里的 RetryProxy 是一个工厂类, 它会构造支持 HA 模式的协议对象, 这个协议对象会首先尝试连接 HDFS 中的 ActiveNamenode, 如果连接失败则会重试, 如果重试达到一定的次数, 则会切换到 HDFS 集群中的 StandbyNamenode。我们直接看一下 RetryProxy 的工厂方法。

```

public static <T> Object create(Class<T> iface,
    FailoverProxyProvider<T> proxyProvider, RetryPolicy retryPolicy) {
    // 直接调用 Java 动态代理构造方法, 返回代理对象
    return Proxy.newProxyInstance(
        proxyProvider.getInterface().getClassLoader(),
        new Class<?>[] { iface },
        new RetryInvocationHandler<T>(proxyProvider, retryPolicy)
    );
}

```

RetryProxy 的工厂方法 RetryProxy()调用了 Java 动态代理方法 Proxy.newProxyInstance(), 那么由上面的代码我们知道, 在协议的代理对象上的调用都会由 RetryInvocationHandler 的 invoke()方法代理(Java 动态代理定义的, 在代理对象上的调用会由 InvocationHandler 的 invoke()方法统一代理)。下面我们就分析一下 RetryInvocationHandler.invoke()方法的实现。invoke()方法的实现主要分为如下几步。

(1) 首先获取 RetryPolicy 对象, RetryPolicy 定义了出现调用错误时的重试逻辑。这里默认的 RetryPolicy 是 failoverOnNetworkException。

(2) 然后 RetryInvocationHandler.invoke()方法通过反射调用 method 对象描述的方法。

```
Object ret = invokeMethod(method, args);
```

invokeMethod()方法其实就是在 currentProxy 对象上调用 method 参数描述的方法——currentProxy 是泛型 T 类型——也就是接口 ClientProtocol 类型。invokeMethod()方法的代码如下:

```
protected Object invokeMethod(Method method, Object[] args) throws Throwable {
    try {
        if (!method.isAccessible()) {
            method.setAccessible(true);
        }
        // 注意, 这里的 currentProxy 会发生切换
        return method.invoke(currentProxy, args);
    } catch (InvocationTargetException e) {
        throw e.getCause();
    }
}
```

currentProxy 是实现了 ClientProtocol 协议的对象, 它是通过调用 FailoverProxyProvider.getProxy()获得的。当 Namenode 发生主从切换后, currentProxy 字段会被赋值为新的 Active Namenode 对应的 ClientProtocol 的引用, 之后在 ClientProtocol 上的调用也就会发送到新的 Active Namenode。

```
this.currentProxy = proxyProvider.getProxy();
```

(3) 如果完成上述操作后没有抛出异常, 也就是说, 客户端成功地将请求发送到了 Active Namenode 服务器。如果抛出异常, 则说明远程调用出现了错误, 这部分代码在 catch 段处理。

- 通过 annotation 判断这个操作是否是 idempotent (幂等的, 也就是执行多次是没有问题的, 例如 ClientProtocol 中的 setReplication()), 对于幂等的操作可以再次调用。之后调用 RetryPolicy.ShouldRetry()方法分析如何处理这个错误。

```
// 判断方法是否是幂等的
```

```
boolean isIdempotentOrAtMostOnce = proxyProvider.getInterface()
    .getMethod(method.getName(), method.getParameterTypes())
    .isAnnotationPresent(Idempotent.class);
if (!isIdempotentOrAtMostOnce) {
```

```

isIdempotentOrAtMostOnce = proxyProvider.getInterface()
    .getMethod(method.getName(), method.getParameterTypes())
    .isAnnotationPresent(AtMostOnce.class);
}

```

// 在 RetryPolicy 上分析需要进行的 retry 动作

```

RetryAction action = policy.shouldRetry(e, retries++,
    invocationFailoverCount, isIdempotentOrAtMostOnce);

```

■ 之后调用 `policy.shouldRetry()` 判断是否需要执行重试操作（这里的 `RetryPolicy` 是默认值 `FailoverOnNetworkExceptionRetry`），这里我们分析 `FailoverOnNetworkExceptionRetry.shouldRetry()` 方法的逻辑。

- 如果失败的次数已经超过最大的次数，就返回一个 `RetryAction.RetryDecision.FAIL` 的 `RetryAction` 表明调用失败。
- 如果抛出的异常是 `ConnectionException`、`NoRouteToHostException`、`UnknownHostException`、`StandbyException`、`RemoteException` 中的一个，则说明底层的协议代理对象无法连接到 `ActiveNamenode`，或者 `ActiveNamenode` 宕机，或者 HDFS 集群已经发生主从切换了。在这些情况下，就需要返回一个 `RetryAction.RetryDecision.FAILOVER_AND_RETRY` 的 `RetryAction`，表明需要执行 `performFailover()` 操作更新 `Active Namenode` 的引用。
- 如果抛出的异常是 `SocketException`、`IOException` 或者其他非 `RemoteException` 的异常，则无法判断这个 RPC 命令到底是不是执行成功了。可能是本地的 `Socket` 或者 `IO` 问题，也可能是 `Namenode` 端的 `Socket` 或者 `IO` 问题。这时就进行进一步的判断：如果被调用的方法是 `idempotent`，也就是多次执行是没有副作用的，那么就连接另外一个底层代理重试；否则直接返回 `RetryAction.RetryDecision.FAIL` 表明调用失败。

```

public RetryAction shouldRetry(Exception e, int retries,
    int failovers, boolean isIdempotentOrAtMostOnce) throws Exception {

```

// 已经超出了最大重试次数

```

if (failovers >= maxFailovers) {
    return new RetryAction(RetryAction.RetryDecision.FAIL, 0,
        "failovers (" + failovers + ") exceeded maximum allowed ("
        + maxFailovers + ")");
}

```

// 判断异常类型

```

if (e instanceof ConnectException ||
    e instanceof NoRouteToHostException ||
    e instanceof UnknownHostException ||
    e instanceof StandbyException ||
    e instanceof ConnectTimeoutException ||
    isWrappedStandbyException(e)) {
    return new RetryAction(
        RetryAction.RetryDecision.FAILOVER_AND_RETRY,

```

```

        // retry immediately if this is our first failover, sleep otherwise
        failovers == 0 ? 0 :
            calculateExponentialTime(delayMillis, failovers, maxDelayBase));
    } else if (e instanceof SocketException ||
        (e instanceof IOException && !(e instanceof RemoteException))) {
        if (isIdempotentOrAtMostOnce) {
            return RetryAction.FAILOVER_AND_RETRY;
        } else {
            return new RetryAction(RetryAction.RetryDecision.FAIL, 0,
                "the invoked method is not idempotent, and unable to determine " +
                "whether it was invoked");
        }
    } else {
        return fallbackPolicy.shouldRetry(e, retries, failovers,
            isIdempotentOrAtMostOnce);
    }
}

```

- 通过异常情况，获得了对应的 `RetryAction` 之后，就会在 `proxyProvider` 上调用 `performFailover()` 方法更新 `currentProxy`。如果是 HA 配置，那么在 Namenode 主从切换后，`performFailover()` 会更新 `currentProxy` 到新的 Active Namenode，然后继续循环，这样在 `currentProxy` 上的调用就可以发送到新的 Active Namenode 了。

```

if (action.action == RetryAction.RetryDecision.FAILOVER_AND_RETRY) {
    synchronized (proxyProvider) {
        if (invocationAttemptFailoverCount == proxyProviderFailoverCount) {
            proxyProvider.performFailover(currentProxy);
            proxyProviderFailoverCount++;
            currentProxy = proxyProvider.getProxy();
        } else {
            LOG.warn();
        }
    }
}

```

2.2.4 服务器获取 Server 对象

在 Hadoop RPC 使用概述小节中我们介绍了 RPC 服务器需要构造一个 Server 对象，这个 Server 对象用于监听并响应来自 RPC 客户端的请求。例如对于 Namenode，它会构造两个 Server 对象分别响应来自 HDFS 客户端和 Datanode 的 RPC 请求。本节就以 Namenode 构造 Server 的流程为例，介绍 RPC 服务器端获取 Server 对象的代码实现。

1. 构造 NameNodeRpcServer

Namenode 定义了 `NameNodeRpcServer` 类响应来自 HDFS 集群中其他节点的 RPC 请求，如图 2-11 所示，`NameNodeRpcServer` 实现了包括 `ClientProtocol`、`NamenodeProtocol`、`DatanodeProtocol` 以及 `HAServiceProtocol` 在内的所有需要与 Namenode 交互的 RPC 协议接口。

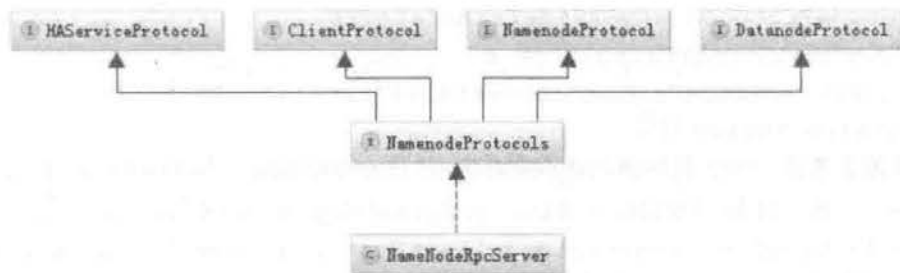


图 2-11 NameNodeRpcServer 实现的接口

NameNode 会在它的初始化方法 `initialize()` 中调用 `createRpcServer()` 创建 `NameNodeRpcServer` 对象的实例，`createRpcServer()` 方法会直接调用 `NameNodeRpcServer` 的构造方法。我们首先看一下这个构造方法。由于 `NameNodeRpcServer` 实现了多个接口，造成它的构造方法非常长，这里只列出了与 `ClientProtocol` 相关的代码，以简化篇幅。

`NameNodeRpcServer` 的构造方法首先设置了 RPC 类的序列化引擎为 `protobuf`，然后构造了两个 `RPC.Server` 对象：`clientRpcServer` 用于响应来自 HDFS 客户端的 RPC 请求；`serviceRpcServer` 则用于响应来自 `Datanode` 的 RPC 请求。`clientRpcServer` 和 `serviceRpcServer` 的构造方法很类似，都是先调用 `RPC.build()` 方法获取一个 `RPC.Server` 对象，然后调用 `DFSUtil.addPBProtocol()` 方法在新获取的 `RPC.Server` 对象上添加 `*ProtocolPB` 协议与 `BlockingService` 对象之间的映射关系，例如 `ClientNamenodeProtocolPB` 协议会由 `BlockingService` 对象 `clientNNPbService` 处理。之所以配置这个映射关系，是因为当 `RPC.Server` 监听到网络上的 RPC 请求后，它会首先提取出 RPC 请求的请求头域，解析出这次 RPC 请求是在什么接口（接口信息）的什么方法（方法信息）上调用的，然后根据接口信息以及上述配置的映射关系提取出执行响应操作的 `BlockingService` 对象，最后根据方法信息调用 `BlockingService.callBlockingMethod()` 方法响应这个 RPC 调用。所以 `*ProtocolPB` 协议接口与 `BlockingService` 对象之间的映射关系保证了 RPC 请求到达 Server 后，Server 可以找到正确的响应类来执行相应操作。`NameNodeRpcServer` 的构造方法代码如下：

```

public NameNodeRpcServer(Configuration conf, NameNode nn)
    throws IOException {
    this.nn = nn;
    this.namesystem = nn.getNamesystem();
    this.metrics = NameNode.getNameNodeMetrics();

    int handlerCount =
        conf.getInt(DFS_NAMENODE_HANDLER_COUNT_KEY,
            DFS_NAMENODE_HANDLER_COUNT_DEFAULT);

    // 设置 RPC 引擎为 protobuf
    RPC.setProtocolEngine(conf, ClientNamenodeProtocolPB.class,
        ProtobufRpcEngine.class);

    // 构造 ClientNamenodeProtocolServerSideTranslatorPB 对象
    // 这个对象用于适配 ClientProtocolPB 到 ClientProtocol 接口的转换
  
```

Hadoop 2.X HDFS 源码剖析

```
ClientNamenodeProtocolServerSideTranslatorPB
    clientProtocolServerTranslator =
        new ClientNamenodeProtocolServerSideTranslatorPB(this);
// 构造 BlockingService 对象
// 这个对象用于将 Server 提取出的请求前转到 clientProtocolServerTranslator 对象
BlockingService clientNNPbService = ClientNamenodeProtocol.
    newReflectiveBlockingService(clientProtocolServerTranslator);
// ...其他接口同 ClientProtocol 类似
WritableRpcEngine.ensureInitialized();

// 初始化 serviceRpcServer 对象, 这个对象用于响应来自 Datanode 的请求
InetSocketAddress serviceRpcAddr = nn.getServiceRpcServerAddress(conf);
if (serviceRpcAddr != null) {
    String bindHost = nn.getServiceRpcServerBindHost(conf);
    if (bindHost == null) {
        bindHost = serviceRpcAddr.getHostName();
    }
    int serviceHandlerCount =
        conf.getInt(DFS_NAMENODE_SERVICE_HANDLER_COUNT_KEY,
            DFS_NAMENODE_SERVICE_HANDLER_COUNT_DEFAULT);

    // 构造 serviceRpcServer 对象, 并配置 ClientProtocolPB 的响应类为 clientNNPbService
    this.serviceRpcServer = new RPC.Builder(conf)
        .setProtocol(
            org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
        .setInstance(clientNNPbService)
        .setBindAddress(bindHost)
        .setPort(serviceRpcAddr.getPort()).setNumHandlers(serviceHandlerCount)
        .setVerbose(false)
        .setSecretManager(namesystem.getDelegationTokenSecretManager())
        .build();

    // 注册 NamenodeRPCServer 实现的所有接口
    DFSUtil.addPBProtocol(conf, HATServiceProtocolPB.class, haPbService,
        serviceRpcServer);
    DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class, NNPbService,
        serviceRpcServer);
    // ...添加其他协议

    // ...更新端口号
} else {
    serviceRpcServer = null;
    serviceRPCAddress = null;
}
InetSocketAddress rpcAddr = nn.getRpcServerAddress(conf);
String bindHost = nn.getRpcServerBindHost(conf);
```

```

if (bindHost == null) {
    bindHost = rpcAddr.getHostName();
}

// 构造 clientRpcServer, 用于响应来自 HDFS 客户端的 RPC 请求
this.clientRpcServer = new RPC.Builder(conf)
    .setProtocol(
        org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
    .setInstance(clientNNPbService).setBindAddress(bindHost)
    .setPort(rpcAddr.getPort()).setNumHandlers(handlerCount)
    .setVerbose(false)
    .setSecretManager(namesystem.getDelegationTokenSecretManager()).build();

// 注册 NamenodeRPCServer 实现的所有接口
DFSUtil.addPBProtocol(conf, HATServiceProtocolPB.class, haPbService,
    clientRpcServer);
DFSUtil.addPBProtocol(conf, NamenodeProtocolPB.class, NNPbService,
    clientRpcServer);
// ...添加其他协议

// ...安全相关、更新端口号、配置异常处理机制
}

```

图 2-12 总结了 NameNodeRpcServer 构造方法中 serviceRpcServer 的简化版构造流程, 整个流程可以分为两个部分: ①获取响应 ClientNamenodeProtocolPB 请求的 BlockingService 对象 (其他协议的处理流程与 ClientNamenodeProtocolPB 类似); ②构造 RPC.Server。这两部分内容我们会在后面两个小节中分别进行介绍。

2. 获取 BlockingService 对象

在 NameNodeRpcServer 的构造方法中, ClientNamenodeProtocolPB 协议对应的 BlockingService 实现类是通过调用 ClientNamenodeProtocol.newReflectiveBlockingService(BlockingInterfaceImpl)方法获取的, 而 newReflectiveBlockingService()方法的 impl 参数则是实现了 ClientNamenodeProtocolPB 接口的 ClientNamenodeProtocolServerSideTranslatorPB 对象, 这个对象是直接通过它的构造方法构造的。在 ClientNamenodeProtocolServerSideTranslatorPB 类小节中我们介绍了 ClientNamenodeProtocolServerSideTranslatorPB 对象是将 ClientNamenodeProtocolPB 接口调用适配成 ClientProtocol 调用的适配器类, 它内部会持有一个实现了 ClientProtocol 接口的对象, 将 ClientNamenodeProtocolPB 调用的参数反序列化之后, 它会调用 ClientProtocol 对象的对应方法执行 RPC 操作。通过 NameNodeRpcServer 构造方法的代码我们知道, ClientNamenodeProtocolServerSideTranslatorPB 中持有的 ClientProtocol 接口对象其实就是 NameNodeRpcServer, NameNodeRpcServer 实现了 ClientProtocol 接口, 是 Namenode RPC 服务真正的实现类。

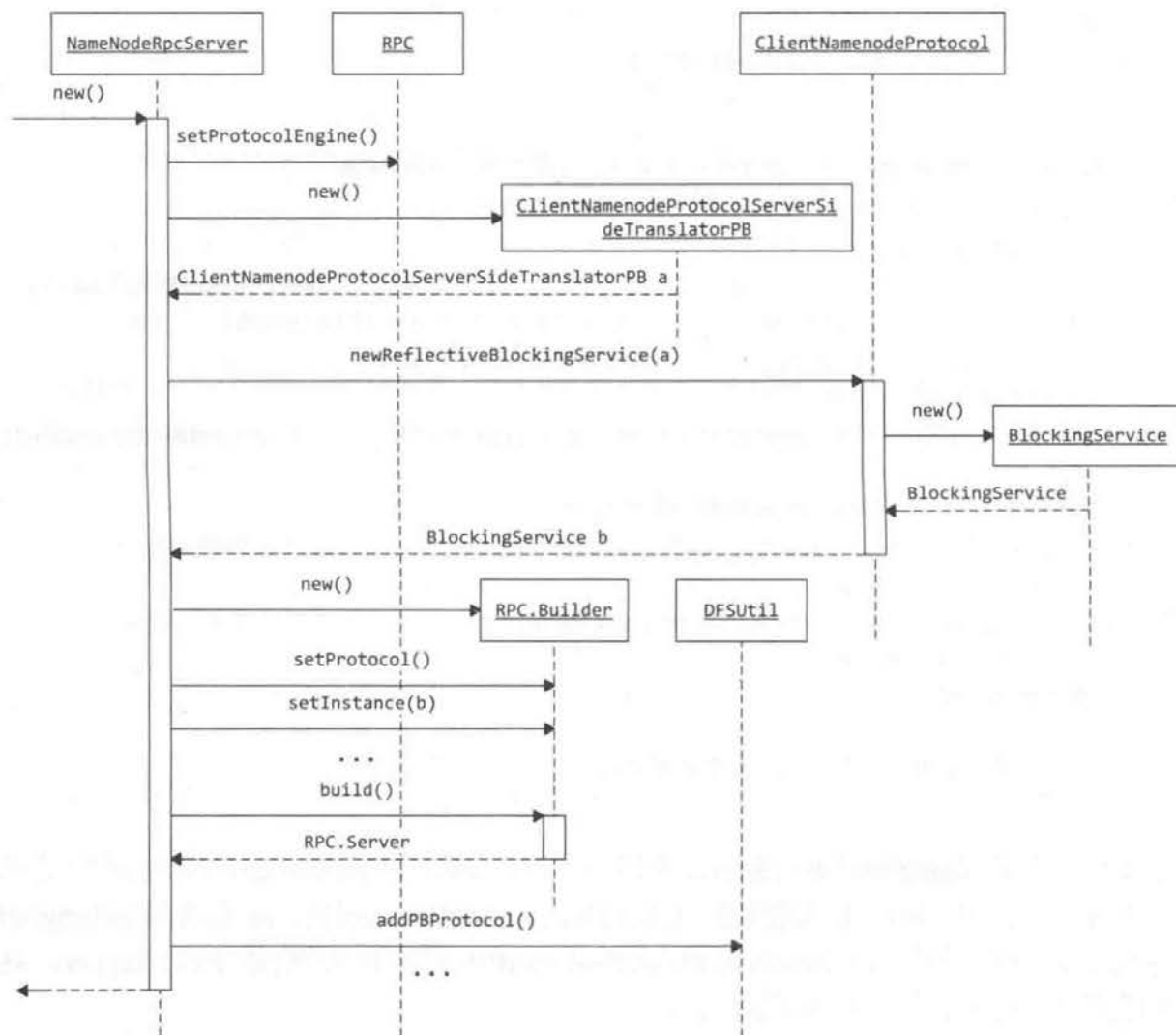


图 2-12 NameNodeRpcServer 构造方法调用流程图

```

public NameNodeRpcServer(Configuration conf, NameNode nn)
    throws IOException {
    // ...

    // 构造 ClientNamenodeProtocolServerSideTranslatorPB 对象
    // 这个对象用于适配 ClientProtocolPB 到 ClientProtocol 接口的转换
    ClientNamenodeProtocolServerSideTranslatorPB
    clientProtocolServerTranslator =
        new ClientNamenodeProtocolServerSideTranslatorPB(this);
    // 构造 BlockingService 对象
    // 这个对象用于将 Server 提取出的请求前转到 clientProtocolServerTranslator 对象
    BlockingService clientNNPbService = ClientNamenodeProtocol.
        newReflectiveBlockingService(clientProtocolServerTranslator);
    // ...
}

```

ClientNamenodeProtocol.newReflectiveBlockingService()方法的定义如下代码所示，它会构

造一个匿名的 `BlockingService` 对象并返回，这个匿名的 `BlockingService` 对象定义了一个 `callBlockingMethod()` 方法。`callBlockingMethod()` 方法接受三个参数：`method` 参数描述了当前 RPC 调用的方法信息；`controller` 参数在这里默认为 `null`，不使用；`request` 参数记录了 RPC 调用的参数信息。`callBlockingMethod()` 方法会根据 `method` 参数记录的调用方法信息，在 `impl` 引用上调用对应的方法。这里的 `impl` 引用是 `ClientNamenodeProtocolServerSideTranslatorPB` 类型的，`ClientNamenodeProtocolServerSideTranslatorPB` 会将 `ClientNamenodeProtocolPB` 调用的参数反序列化，然后前转到 `NamenodeRpcServer` 对象上执行 RPC 操作。这样，`Server` 对象监听到 RPC 请求后，只需通过请求头域中的接口信息获取对应的 `BlockingService` 对象，然后在这个 `BlockingService` 对象上调用 `callBlockingMethod()` 就可以触发 `NameNodeRpcServer` 对象响应这个 RPC 请求了。

```
public static com.google.protobuf.BlockingService
    newReflectiveBlockingService(final BlockingInterface impl) {
    return new com.google.protobuf.BlockingService() {
        public final com.google.protobuf.Descriptors.ServiceDescriptor
            getDescriptorForType() {
            return getDescriptor();
        }
    }

    public final com.google.protobuf.Message callBlockingMethod(
        com.google.protobuf.Descriptors.MethodDescriptor method,
        com.google.protobuf.RpcController controller,
        com.google.protobuf.Message request)
        throws com.google.protobuf.ServiceException {
        if (method.getService() != getDescriptor()) {
            throw new java.lang.IllegalArgumentException(
                "Service.callBlockingMethod() given method descriptor for " +
                "wrong service type.");
        }
        switch(method.getIndex()) {
            case 0:
                return impl.getBlockLocations(controller, (GetBlockLocationsRequestProto)
request);
            case 1:
                return impl.getServerDefaults(controller, (GetServerDefaultsRequestProto)
request);
            case 2:
                // ...
            default:
                throw new java.lang.AssertionError("Can't get here.");
        }
    }
    // ...
};
}
```

3. 构造 Server 对象

NameNodeRpcServer 的构造方法调用了 RPC.Server.build()方法构造 Server 对象, build()方法的代码如下所示, 它设置了 protocol 参数为 ClientNamenodeProtocolPB.class, 同时设置了 impl 参数为 clientNNPbService。这两个参数设置了 ClientNamenodeProtocolPB 协议与它的 BlockingService 响应类的对应关系, 也就是说, ClientNamenodeProtocolPB 的调用会由 BlockingService 对象 clientNNPbService 来响应。本节我们就来学习 build()方法的实现, 也就是 RPC Server 对象的构造。

```
this.serviceRpcServer = new RPC.Builder(conf)
    .setProtocol(
        org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)
    .setInstance(clientNNPbService)
    .setBindAddress(bindHost)
    .setPort(serviceRpcAddr.getPort()).setNumHandlers(serviceHandlerCount)
    .setVerbose(false)
    .setSecretManager(namesystem.getDelegationTokenSecretManager())
    .build();
```

build()方法首先调用 getProtocolEngine()获取当前 RPC 类配置的 RpcEngine 对象, 在 NameNodeRpcServer 的构造方法中已经将当前 RPC 类的 RpcEngine 对象设置为 ProtobufRpcEngine 了。获取了 ProtobufRpcEngine 对象之后, build()方法会在 ProtobufRpcEngine 对象上调用 getServer()方法获取一个 RPC Server 对象的引用。

```
public Server build() throws IOException, HadoopIllegalArgumentException {
    if (this.conf == null) {
        throw new HadoopIllegalArgumentException("conf is not set");
    }
    if (this.protocol == null) {
        throw new HadoopIllegalArgumentException("protocol is not set");
    }
    if (this.instance == null) {
        throw new HadoopIllegalArgumentException("instance is not set");
    }

    return getProtocolEngine(this.protocol, this.conf).getServer(
        this.protocol, this.instance, this.bindAddress, this.port,
        this.numHandlers, this.numReaders, this.queueSizePerHandler,
        this.verbose, this.conf, this.secretManager, this.portRangeConfig);
}
```

在客户端获取 Proxy 对象小节中我们已经介绍了, 客户端获取 Proxy 对象是通过调用 RpcEngine.getProxy()方法实现的, 对于不同的 RpcEngine 获取的 Proxy 对象是不同的。对于底层来说, 就是 RPC 请求所使用的序列化工具是不同的。同理, 对于 RPC.Server 来说, 不同的 RpcEngine 构造 RPC.Server 对象也是使用不同的反序列化工具, ProtobufRpcEngine.getServer()会返回使用 protobuf 作为反序列化工具的服务器, 这个 RPC.Server 对象是 ProtobufRpcEngine 的内部类 Server (RPC.Server 的子类), getServer()方法会构造这个对象并返回。

```

public RPC.Server getServer(Class<?> protocol, Object protocolImpl,
    String bindAddress, int port, int numHandlers, int numReaders,
    int queueSizePerHandler, boolean verbose, Configuration conf,
    SecretManager<? extends TokenIdentifier> secretManager,
    String portRangeConfig)
    throws IOException {
    return new Server(protocol, protocolImpl, conf, bindAddress, port,
        numHandlers, numReaders, queueSizePerHandler, verbose, secretManager,
        portRangeConfig);
}

```

下面我们看一下 `ProtobufRpcEngine.Server` 的构造方法。`ProtobufRpcEngine.Server` 是 `RPC.Server` 的子类，它的构造方法首先调用父类 `RPC.Server` 的构造方法（`RPC.Server` 类的实现我们将在下一节中介绍），之后构造方法会调用 `registerProtocolAndImpl()` 方法注册接口类 `protocolClass` 和实现类 `protocolImpl` 的映射关系。这样，当客户端的 RPC 请求到达时，就可以通过这个映射关系获得具体的实现类了。

```

public Server(Class<?> protocolClass, Object protocolImpl,
    Configuration conf, String bindAddress, int port, int numHandlers,
    int numReaders, int queueSizePerHandler, boolean verbose,
    SecretManager<? extends TokenIdentifier> secretManager,
    String portRangeConfig)
    throws IOException {
    super(bindAddress, port, null, numHandlers,
        numReaders, queueSizePerHandler, conf, classNameBase(protocolImpl
            .getClass().getName()), secretManager, portRangeConfig);
    this.verbose = verbose;
    registerProtocolAndImpl(RPC.RpcKind.RPC_PROTOCOL_BUFFER, protocolClass,
        protocolImpl);
}

```

`ProtobufRpcEngine.Server` 类最重要的部分就是实现了一个 `ProtoBufRpcInvoker` 类，当 `RPC.Server` 类解析来自网络的 RPC 请求后，会调用 `ProtoBufRpcInvoker.call()` 方法响应这个请求。这里我们重点看一下 `call()` 方法的实现。`call()` 方法首先会从请求头中提取出 RPC 调用的接口名和方法名等信息，然后根据调用的接口信息获取对应的 `BlockingService` 对象，再根据调用的方法信息在 `BlockingService` 对象上调用 `callBlockingMethod()` 方法并将调用前转到 `ClientNamenodeProtocolServerSideTranslatorPB` 对象上，最终这个请求会由 `NameNodeRpcServer` 响应。

```

public Writable call(RPC.Server server, String protocol,
    Writable writableRequest, long receiveTime) throws Exception {
    // 获取 rpc 调用头
    RpcRequestWrapper request = (RpcRequestWrapper) writableRequest;
    RequestHeaderProto rpcRequest = request.requestHeader;
    // 获得调用的接口名、方法名、版本号
    String methodName = rpcRequest.getMethodName();
    String protoName = rpcRequest.getDeclaringClassProtocolName();
    long clientVersion = rpcRequest.getClientProtocolVersion();
}

```

Hadoop 2.X HDFS 源码剖析

```
// 获得该接口在 Server 侧对应的实现类
ProtoClassProtoImpl protocolImpl = getProtocolImpl(server, protoName,
    clientVersion);
BlockingService service = (BlockingService) protocolImpl.protocolImpl;
MethodDescriptor methodDescriptor = service.getDescriptorForType()
    .findMethodByName(methodName);
if (methodDescriptor == null) {
    String msg = "Unknown method " + methodName + " called on " + protocol
        + " protocol.";
    LOG.warn(msg);
    throw new RpcNoSuchMethodException(msg);
}
// 获取调用的方法描述符以及调用参数
Message prototype = service.getRequestPrototype(methodDescriptor);
Message param = prototype.newBuilderForType()
    .mergeFrom(request.theRequestRead).build();

Message result;
try {
    // ...
    // 在实现类上调用 callBlockingMethod 方法，级联适配调用到 NameNodeRpcServer
    result = service.callBlockingMethod(methodDescriptor, null, param);
    // ...
} catch (ServiceException e) {
    throw (Exception) e.getCause();
} catch (Exception e) {
    throw e;
}
return new RpcResponseWrapper(result);
}
```

通过对 `call()` 方法的分析我们知道，客户端在什么接口上调用什么方法是在请求头 `requestHeader` 中保存的，`ProtoBufRpcInvoker` 获取了接口信息之后，会调用 `getProtocolImpl()` 方法通过接口名获取对应的实现类。这个映射信息是在 `RPC.Server` 构造时创建的，同时也可以 在 `NameNodeRpcServer` 的构造方法中调用 `DFSUtil.addPBProtocol()` 方法添加。`getProtocolImpl()` 方法的实现如下：

```
private static ProtoClassProtoImpl getProtocolImpl(RPC.Server server,
    String protoName, long clientVersion) throws RpcServerException {
    ProtoNameVer pv = new ProtoNameVer(protoName, clientVersion);
    // 从 RPC.Server 的 ProtocolImplMap 对象中获取接口信息对应的实现类
    ProtoClassProtoImpl impl =
        server.getProtocolImplMap(RPC.RpcKind.RPC_PROTOCOL_BUFFER).get(pv);
    if (impl == null) { //
        VerProtocolImpl highest =
            server.getHighestSupportedProtocol(RPC.RpcKind.RPC_PROTOCOL_BUFFER,
                protoName);
        // 如果不存在实现类，则抛出异常
    }
}
```

```

    if (highest == null) {
        throw new RpcNoSuchProtocolException(
            "Unknown protocol: " + protoName);
    }
    // 如果 RPC 版本不匹配, 则抛出异常
    throw new RPC.VersionMismatch(protoName, clientVersion,
        highest.version);
}
// 返回实现类
return impl;
}

```

通过接口信息获取对应的 `BlockingService` 实现对象后, `call()` 方法会在这个 `BlockingService` 对象上调用 `BlockingService.callBlockingMethod()` 方法, 这部分内容我们在获取 `BlockingService` 对象小节中已经介绍过了。至此, 一个完整的 `Server` 对象构造以及响应 RPC 请求的流程就介绍完了。

2.3 Hadoop RPC 实现

Hadoop RPC 框架主要由三个类组成: `RPC`、`Client` 和 `Server` 类。`RPC` 类用于对外提供一个使用 Hadoop RPC 框架的接口, `Client` 类用于实现 RPC 客户端功能, `Server` 类则用于实现 RPC 服务器端功能。本节分别介绍这三个类的代码实现。

2.3.1 RPC 类实现

`RPC` 类为使用 Hadoop RPC 框架的代码提供了一个统一的接口, 同时隐藏了底层 RPC 通信的实现细节, 方便了用户的使用。

如图 2-13 所示, 客户端调用程序可以通过调用 `RPC` 类提供的 `waitForProxy()` 和 `getProxy()` 方法获取指定 RPC 协议的代理对象, 之后 `RPC` 客户端就可以调用代理对象的方法发送 RPC 请求到服务器了。`getProxy()` 方法的实现我们在客户端获取 `Proxy` 对象小节中已经介绍过了, 这里不再重复介绍。

而在服务器侧, 服务程序会调用 `RPC` 内部类 `Builder.build()` 方法构造一个 `RPC.Server` 类, 然后调用 `RPC.Server.start()` 方法启动 `Server` 对象监听并响应 RPC 请求。这里的 `RPC.Builder` 内部类是 `RPC` 定义的用来构造 `RPC.Server` 对象的工厂类, 用户可以调用 `Builder.set*()` 方法对 `RPC.Server` 对象进行配置, 之后调用 `build()` 方法构造这个 `RPC.Server` 对象。`RPC.Server` 内部类则是 `Server` 的子类, 它将监听到的 RPC 请求委托给了 `RPC` 内部接口 `RpcInvoker` 的子类处理, 当 `RPC.Server` 监听到一个 RPC 请求后, 它会调用 `RpcInvoker.call()` 方法处理这个请求。这部分内容我们在服务器获取 `Server` 对象小节中也介绍过了, 这里不再重复介绍。

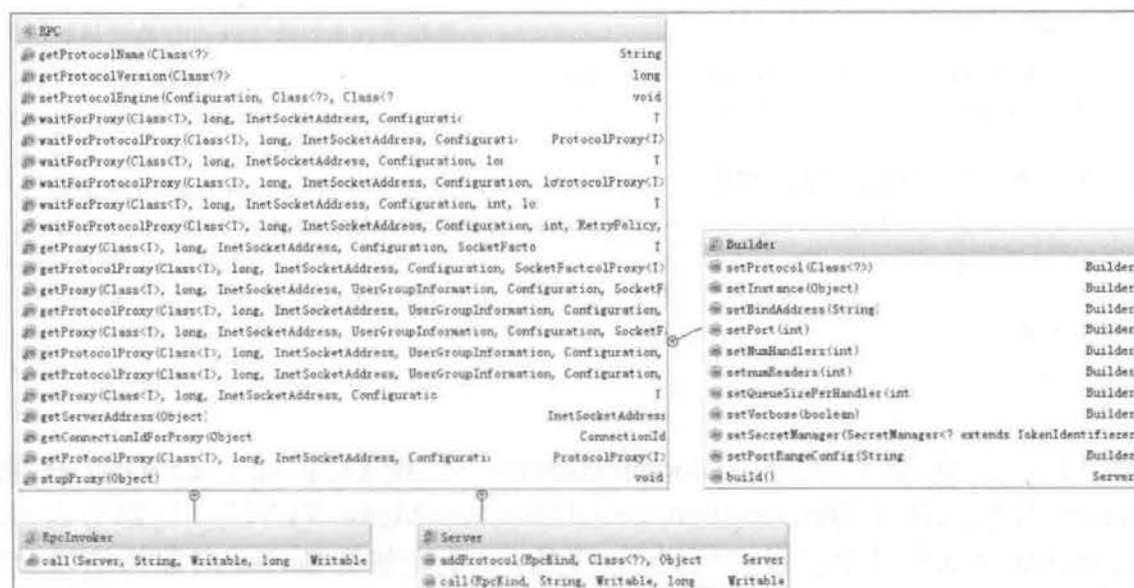


图 2-13 RPC 类结构图

RPC 类还提供了 `setProtocolEngine()` 方法用于配置 RPC 框架当前使用的序列化引擎，以及 `getProtocolEngine()` 方法用于获取序列化引擎对象。在客户端获取 Proxy 对象以及服务器获取 Server 对象小节中我们介绍了 `RPC.getProtocolProxy()` 以及 `RPC.Builder.build()` 方法都会先调用 `getProtocolEngine()` 获取当前 RPC 配置的序列化引擎，然后调用 `RpcEngine.getProxy()` 以及 `RpcEngine.getServer()` 方法执行构造 Proxy 对象和 Server 对象操作。这里以 `getProtocolProxy()` 方法为例，`RPC.Builder.build()` 方法就不再贴出代码了。

```

public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,
                                                    long clientVersion,
                                                    InetSocketAddress addr,
                                                    UserGroupInformation ticket,
                                                    Configuration conf,
                                                    SocketFactory factory,
                                                    int rpcTimeout,
                                                    RetryPolicy connectionRetryPolicy,
                                                    AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    // 首先调用 getProtocolEngine() 方法获取 RPC 引擎
    return getProtocolEngine(protocol, conf).getProxy(protocol, clientVersion,
                                                    addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy,
                                                    fallbackToSimpleAuth);
}
  
```

2.3.2 Client 类实现

在 Hadoop RPC 的使用小节中我们介绍了 `DFSClient` 会获取一个 `ClientProtocolPB` 协议的代理对象，并在这个代理对象上调用 RPC 方法，代理对象会调用 `RPC.Client.call()` 方法将序列化之后的 RPC 请求发送到服务器。本节就介绍 Client 类的实现。

1. Client 发送请求与接收响应流程

Client 类只有一个入口，就是 `call()` 方法。代理类会调用 `Client.call()` 方法将 RPC 请求发送到远程服务器，然后等待远程服务器的响应。如果远程服务器响应请求时出现异常，则在 `call()` 方法中抛出异常。`Client.call()` 方法发送请求与接收响应的流程如图 2-14 所示，分为以下几步。

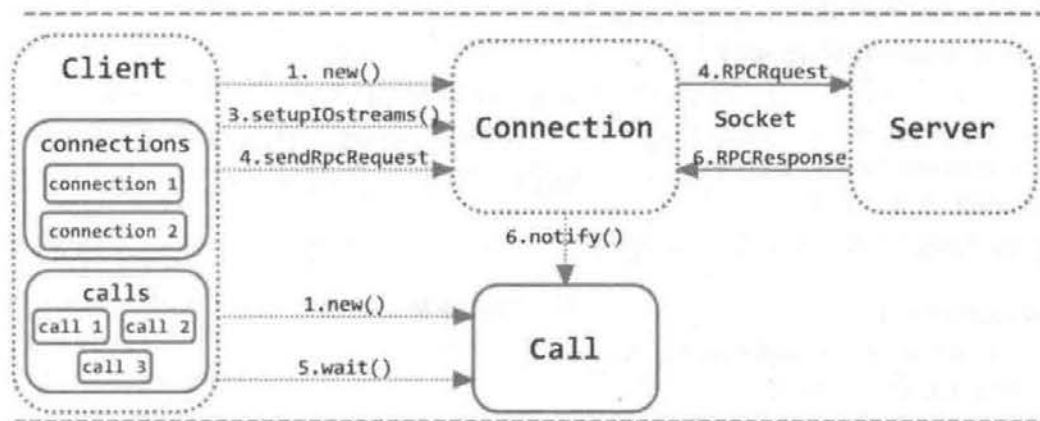


图 2-14 `Client.call()` 方法执行流程图

- `Client.call()` 方法将 RPC 请求封装成一个 `Call` 对象，`Call` 对象中保存了 RPC 调用的完成标志、返回值信息以及异常信息；随后，`Client.call()` 方法会创建一个 `Connection` 对象，`Connection` 对象用于管理 `Client` 与 `Server` 的 `Socket` 连接。
- 用 `ConnectionId` 作为 key，将新建的 `Connection` 对象放入 `Client.connections` 字段中保存（对于 `Connection` 对象，由于涉及了与 `Server` 建立 `Socket` 连接，会比较耗费资源，所以 `Client` 类使用一个 `HashTable` 对象 `connections` 保存那些没有过期的 `Connection`，如果可以复用，则复用这些 `Connection` 对象）；以 `callId` 作为 key，将构造的 `Call` 对象放入 `Connection.calls` 字段中保存。
- `Client.call()` 方法调用 `Connection.setupIOStreams()` 方法建立与 `Server` 的 `Socket` 连接。`setupIOStreams()` 方法还会启动 `Connection` 线程，`Connection` 线程会监听 `Socket` 并读取 `Server` 发回的响应信息。
- `Client.call()` 方法调用 `Connection.sendRpcRequest()` 方法发送 RPC 请求到 `Server`。
- `Client.call()` 方法调用 `Call.wait()` 在 `Call` 对象上等待，等待 `Server` 发回响应信息。。
- `Connection` 线程收到 `Server` 发回的响应信息，根据响应消息中携带的信息找到对应的 `Call` 对象，然后设置 `Call` 对象的返回值字段，并调用 `call.notify()` 唤醒调用 `Client.call()` 方法的线程读取 `Call` 对象的返回值。

`Client.call()` 方法的代码如下：

```

public Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,
    ConnectionId remoteId, int serviceClass,
    AtomicBoolean fallbackToSimpleAuth) throws IOException {
    // 构造 Call 对象
    final Call call = createCall(rpcKind, rpcRequest);
    // 构造 Connection 对象
    Connection connection = getConnection(remoteId, call, serviceClass,

```

Hadoop 2.X HDFS 源码剖析

```
        fallbackToSimpleAuth);
    try {
        connection.sendRpcRequest(call);           // 发送 RPC 请求
    } catch (RejectedExecutionException e) {
        throw new IOException("connection has been closed", e);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        throw new IOException(e);
    }

    boolean interrupted = false;
    synchronized (call) {
        while (!call.done) {
            try {
                call.wait();                       // 等待 RPC 响应
            } catch (InterruptedException ie) {
                interrupted = true;
            }
        }

        if (interrupted) {
            Thread.currentThread().interrupt();
        }

        if (call.error != null) { // 发送线程被唤醒，但是服务器处理 RPC 请求时出现异常
            // 从 Call 对象中获取异常，并抛出
            if (call.error instanceof RemoteException) {
                call.error.fillInStackTrace();
                throw call.error;
            } else {
                InetSocketAddress address = connection.getRemoteAddress();
                throw NetUtils.wrapException(address.getHostName(),
                    address.getPort(),
                    NetUtils.getHostname(),
                    0,
                    call.error);
            }
        } else {
            // 服务器成功发回响应信息，返回 RPC 响应
            return call.getRpcResponse();
        }
    }
}
```

可以看到 RPC.Client 的实现涉及很多内部类，例如 Call、ConnectionId、Connection 等，我们首先介绍一下这些内部类的实现，然后再详细介绍发送请求以及接收响应这两个流程的实现。

2. 内部类——Call

RPC.Client 中发送请求和接收响应是由两个独立的线程进行的，发送请求线程就是调用 Client.call()方法的线程，而接收响应线程则是 call()启动的 Connection 线程。那么这两个线程是如何同步 Server 发回的响应信息的呢？这里就使用了 Call 类。举个例子，线程 1 调用 Client.call()发送 RPC 请求到 Server，然后在这个请求对应的 Call 对象上调用 Call.wait()方法等待 Server 发回响应信息。当线程 2 从 Server 接收了响应信息后，会设置 Call.rpcResponse 字段保存响应信息，然后调用 Call.notify()方法唤醒线程 1。线程 1 被唤醒后，会取出 Call.rpcResponse 字段中记录的 Server 发回的响应信息并返回。Call 对象巧妙地同步了 RPC 请求的发送线程以及 RPC 响应的接收线程，值得我们积累和学习。

Call 对象标识了一个 RPC 请求，它定义了如下字段用于记录一个 RPC 请求的信息。

```
final int id;           // RPC 请求的 id
final int retry;        // 重试次数
final Writable rpcRequest; // 序列化的 RPC 请求
Writable rpcResponse;   // 序列化的 RPC 响应，如果调用发生错误，则这个字段为空
IOException error;      // 如果发生错误，则保存远程的异常
final RPC.RpcKind rpcKind; // Rpc 引擎类型
boolean done;           // 调用操作是否完成
```

下面我们看一下 Call 的方法实现。当 Server 成功地执行了 RPC 调用，并发回响应到接收线程后，接收线程会调用 Call.setRpcResponse()方法保存 Server 发回的响应信息。如果 Server 在执行 RPC 调用时出现异常，则接收线程会调用 Call.setException()方法保存异常信息。要特别注意的是，setRpcResponse()以及 setException()都会唤醒在 Call 对象上等待的请求发送线程。

```
public synchronized void setException(IOException error) {
    this.error = error; // 保存异常信息
    callComplete();    // 调用 callComplete()方法唤醒在 Call 对象上等待的线程
}

public synchronized void setRpcResponse(Writable rpcResponse) {
    this.rpcResponse = rpcResponse; // 保存响应信息
    callComplete();    // 调用 callComplete()方法唤醒在 Call 对象上等待的线程
}

protected synchronized void callComplete() {
    this.done = true; // 设置 done 字段为 true，表明当前请求
    notify();        // 唤醒在 Call 对象上等待的线程
}
```

3. 内部类——Connection

内部类 Connection 是一个线程类，它提供了建立 Client 到 Server 的 Socket 连接、发送 RPC 请求以及读取 RPC 响应信息等功能。Connection 的字段多是与网络连接相关的，如 Socket 输入输出流、超时时间、重发次数等。Connection 类定义的字段如下：

```
private InetAddress server; // Server IP 端口
private final ConnectionId remoteId; // connectionId 唯一标识一个 Connection
// ...
private Socket socket = null; // 到 Server 的 Socket 连接
private DataInputStream in;
private DataOutputStream out;
private int rpcTimeout;
private int maxIdleTime; // 最长空闲时间

private Hashtable<Integer, Call> calls = new Hashtable<Integer, Call>(); // 使用这个
Connection 对象发送的请求
private AtomicBoolean shouldCloseConnection = new AtomicBoolean(); // 是否关闭这个连接
private IOException closeException; // 导致连接关闭的异常
```

- 网络相关字段：与 Server 进行网络通信的所有字段，例如 socket、inputStream、outputStream、maxIdleTime、tcpNoDelay 等。
- calls 字段：一个 Connection 对象往往对应于到一个 Server 的 Socket 连接，在这个 Server 上的所有 RPC 调用是可以共用这个 Connection 对象的，所以 Connection 定义了 calls 字段保存这些调用对应的 Call 对象。
- shouldCloseConnection 字段：是否关闭当前 Connection 对象的标识符。

对于 Connection 类的分析我们还是从入口方法切入，RPC.Client.call()方法会首先调用 Connection.getConnection()方法获取一个 Connection 对象。getConnection()方法首先尝试从 RPC.Client.connections 字段中提取缓存的 Connection 对象。RPC.Client.connections 字段是 Hashtable(ConnectionId -> Connection)类型的，用于将已经成功建立 Socket 连接的 Connection 对象缓存起来，这是因为 Connection 类中与 Server 建立 Socket 连接的过程非常耗费资源，所以 Client 类使用 connections 字段保存那些没有过期的 Connection 对象，如果可以复用，则复用这些 Connection 对象。如果 RPC.Client.connections 字段没有缓存 Connection 对象，则 getConnection()方法会直接调用 Connection 的构造方法创建新的 Connection 对象，并将新构造的 Connection 对象放入 RPC.Client.connections 字段中保存。成功地获取了 Connection 对象后，getConnection()方法会调用 addCall()方法将待发送的 RPC 请求对象 Call 添加到这个 Connection 的请求队列 calls 当中，然后调用 setupIOstreams()方法初始化到 Server 的 Socket 连接并获取 IO 流。getConnection()方法的代码如下：

```
private Connection getConnection(ConnectionId remoteId,
    Call call, int serviceClass, AtomicBoolean fallbackToSimpleAuth)
    throws IOException {
    if (!running.get()) {
        throw new IOException("The client is stopped");
    }
    Connection connection;
    do {
        synchronized (connections) {
            // 首先尝试从 Client.connections 队列中获取 Connection 对象
            connection = connections.get(remoteId);
            if (connection == null) {
```

```

        // 如果 connections 队列中没有保存, 则构造新的对象
        connection = new Connection(remoteId, serviceClass);
        connections.put(remoteId, connection);
    }
}
// 将待发送请求对应的 Call 对象放入 Connection.calls 队列
} while (!connection.addCall(call));

// 调用 setupIOstreams() 方法, 初始化 Connection 对象并获取 IO 流
connection.setupIOstreams(fallbackToSimpleAuth);
return connection;
}

```

Connection 的构造方法以及 setupIOstreams() 方法的实现我们在下面两个小节中分析。

(1) Connection 构造方法

Connection 类的构造方法比较简单, 它会根据传入的 ConnectionId 对 Connection 对象中的字段赋值, 同时由于 Connection 还是一个线程类, 构造方法还会将当前线程设置为精灵线程。

(2) setupIOstreams() 方法

可以看到 Connection 类的构造方法并没有建立到远程服务器的 Socket 连接, 这个部分是由 setupIOstreams() 方法执行的。setupIOstreams() 方法除了连接到远程服务器并建立 IO 流外, 还会向服务器发送一个连接头, 然后启动 Connection 线程监听 Socket 输入流并等待服务器返回 RPC 响应。由于 setupIOstreams() 方法很长, 我们将其拆分成几个部分进行介绍。

- 首先调用 setupConnection() 建立到 Server 的 Socket 连接, 并且在这个 Socket 连接上获得 InputStream 和 OutputStream 对象。
- 调用 writeConnectionHeader() 方法在连接建立时发送连接头域。
- 将 inputStream 和 outputStream 装饰成 DataInputStream 和 DataOutputStream 类型, 方便以后读写。
- 调用 writeConnectionContext() 方法写入连接上下文。
- 调用 touch() 方法更新上次活跃时间。
- 调用 start() 方法启动 Connection 线程监听并接收 Server 发回的响应信息。

```

private synchronized void setupIOstreams(
    AtomicBoolean fallbackToSimpleAuth) {
    try {
        short numRetries = 0;
        Random rand = null;
        while (true) {
            // 1. 调用 setupConnection() 方法建立到远程服务器的连接
            setupConnection();
            InputStream inStream = NetUtils.getInputStream(socket);
            OutputStream outStream = NetUtils.getOutputStream(socket);

```

```
// 2. 发送连接头域
writeConnectionHeader(outStream);

// 3. 包装输入输出流
if (doPing) {
    inStream = new PingInputStream(inStream);
}
this.in = new DataInputStream(new BufferedInputStream(inStream));
if (!(outStream instanceof BufferedOutputStream)) {
    outStream = new BufferedOutputStream(outStream);
}
this.out = new DataOutputStream(outStream);

// 4. 写入连接上下文头域
writeConnectionContext(remoteId, authMethod);

// 5. 更新上次活跃时间
touch();

// 6. 启动 Connection 线程
start();
return;
}
} catch (Throwable t) {
    if (t instanceof IOException) {
        markClosed((IOException)t);
    } else {
        markClosed(new IOException("Couldn't set up IO streams", t));
    }
    close();
}
}
```

(3) 发送请求——Connection.sendRpcRequest()

RPC 发送请求线程会调用 Connection.sendRpcRequest()方法发送 RPC 请求到 Server, 这里要特别注意, 这个方法不是由 Connection 线程调用的, 而是由发起 RPC 请求的线程调用的。我们知道客户端获取的协议代理类会将请求元数据 (保存调用的接口信息和方法信息) 以及请求参数封装到一个 Writable 对象中, 然后调用 Client.call()方法将这个 Writable 类型的请求发送到 Server 端。sendRpcRequest()方法还会发送一个 RPC 请求头给 Server, 这个 RPC 请求头保存了本次 RPC 请求的序列化类型、请求 id、请求重试次数等信息。Server 在处理这次 RPC 请求时会根据 RPC 请求头域中的信息配置请求处理的流程, 同时在发回 RPC 响应时携带请求 id 等信息。Connection 对象收到响应信息后, 会提取出请求 id, 然后根据请求 id 查找本次响应对应的 Call 对象, 最后执行保存响应信息的操作。sendRpcRequest()方法的执行流程如下:

```
public void sendRpcRequest(final Call call)
    throws InterruptedException, IOException {
```

```

if (shouldCloseConnection.get()) {
    return;
}

// 先构造RPC请求头
final DataOutputBuffer d = new DataOutputBuffer();
RpcRequestHeaderProto header = ProtoUtil.makeRpcRequestHeader(
    call.rpcKind, OperationProto.RPC_FINAL_PACKET, call.id, call.retry,
    clientId);
// 将RPC请求头写入输出流
header.writeDelimitedTo(d);
// 将RPC请求（包括请求元数据和请求参数）写入输出流
call.rpcRequest.write(d);

// 这里使用线程池将请求发送出去，请求包括三个部分：① 长度；② RPC请求头；③ RPC请求（包括请求元
// 数据以及请求参数）
synchronized (sendRpcRequestLock) {
    Future<?> senderFuture = SEND_PARAMS_EXECUTOR.submit(new Runnable() {
        @Override
        public void run() {
            try {
                synchronized (Connection.this.out) {
                    if (shouldCloseConnection.get()) {
                        return;
                    }

                    byte[] data = d.getData();
                    int totalLength = d.getLength();
                    out.writeInt(totalLength); // 总长度
                    out.write(data, 0, totalLength); // RPC请求头 + RPC请求（请求元数据+参数）
                    out.flush();
                }
            } catch (IOException e) {
                // 如果发生发送异常，则直接关闭连接
                markClosed(e);
            } finally {
                // 之前申请的buffer给关闭了，比较优雅
                IOUtils.closeStream(d);
            }
        }
    });

    // 获取执行结果
    try {
        senderFuture.get();
    } catch (ExecutionException e) {
        Throwable cause = e.getCause();
        // 如果有异常则直接抛出
    }
}

```

```

    if (cause instanceof RuntimeException) {
        throw (RuntimeException) cause;
    } else {
        throw new RuntimeException("unexpected checked exception", cause);
    }
}
}
}

```

Client 向 Server 发送的一个完整的 RPC 请求格式如图 2-15 所示。

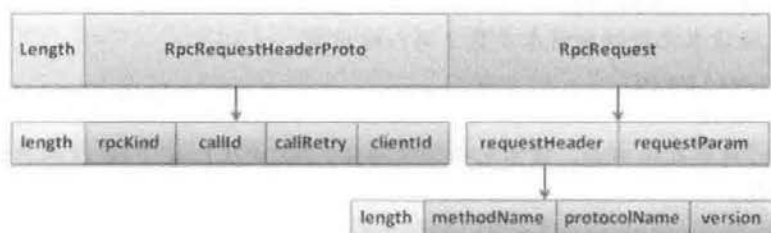


图 2-15 RPC 请求格式

- **length**: 每个 protobuf 类型的数据都包含一个 **length** 字段，这是因为，在 HDFS 写入操作时，使用了 `writeDelimitedTo()` 方法。这个方法会先写入数据的 **length**，然后再写入数据。
- **RpcRequestHeaderProto**: RPC 调用头域，保存了 **callId**、**clientId**、**rpcKind** 等重要信息。服务器发回的响应消息中会带回 **clientId**、**callId** 等信息，用于提取 **call**、鉴权等。
- **RpcRequest**: 要特别注意，这里的 **RpcRequest** 是在 `ProtobufRpcEngine.Invoker.invoke()` 方法中构造的 **RpcRequestWrapper** 类。其中包括两个部分。
 - **requestHeader**: 请求元信息，在什么接口上调用什么方法。例如在 **ClientProtocol** 接口上调用了 `rename()` 方法。
 - **requestParam**: 请求参数，使用 **protobuf** 包装的，例如 `rename()` 请求的 **RenameRequestProto** 参数。

(4) 接收响应——`Connection.run()`

Connection 线程负责监听并接收从 **Server** 发回的 RPC 响应。我们看一下 `Connection.run()` 方法的实现。`Connection.run()` 方法会调用 `waitForWork()` 等待执行读取操作，等待结束后调用 `receiveRpcResponse()` 方法接收 RPC 响应。

```

public void run() {
    try {
        // 调用 waitForWork() 等待
        while (waitForWork()) {
            // 接收响应
            receiveRpcResponse();
        }
    } catch (Throwable t) {
        // 异常都在 receiveRpcResponse 中捕获，这里捕获未知的异常
    }
}

```

```

        markClosed(new IOException("Error reading responses", t));
    }
    // 关闭连接
    close();
}

```

run()方法调用了 waitForWork()方法等待读取操作，waitForWork()方法会判断当前 Connection 的 calls 队列当中是否有 Call 对象，如果没有则等待 maxTimeOut 时长（calls 队列没有对象则表明 Connection 空闲，没有待发送请求）。等待之后如果还没有数据，则返回 false，并且关闭 Connection 对象。

```

private synchronized boolean waitForWork() {
    if (calls.isEmpty() && !shouldCloseConnection.get() && running.get()) {
        long timeout = maxIdleTime -
            (Time.now() - lastActivity.get());
        if (timeout > 0) {
            try {
                wait(timeout);
            } catch (InterruptedException e) {}
        }
    }

    if (!calls.isEmpty() && !shouldCloseConnection.get() && running.get()) {
        return true;
    } else if (shouldCloseConnection.get()) {
        return false;
    } else if (calls.isEmpty()) { // 空闲的连接，则关闭
        markClosed(null);
        return false;
    } else { // 仍有等待处理的请求，但是连接被关闭了
        markClosed((IOException) new IOException().initCause(
            new InterruptedException()));
        return false;
    }
}

```

等待结束后，调用 receiveRpcResponse()方法接收 RPC 响应。receiveRpcResponse()方法会从输入流中读取序列化对象 RpcResponseHeaderProto，然后根据 RpcResponseHeaderProto 中记录的 callid 字段获取对应的 Call 的对象。接下来 receiveRpcResponse()方法会从输入流中读取响应消息，然后调用 Call.setRpcResponse()将响应消息保存在 Call 对象中。如果服务器在处理 RPC 请求时抛出异常，则 receiveRpcResponse()会从输入流中读取异常信息，并构造异常对象，然后调用 Call.setException()将异常保存在 Call 对象中。

```

private void receiveRpcResponse() {
    // ...

    try {
        int totalLen = in.readInt();
    }
}

```

```
RpcResponseHeaderProto header =
    RpcResponseHeaderProto.parseDelimitedFrom(in);
checkResponse(header);

int headerLen = header.getSerializedSize();
headerLen += CodedOutputStream.computeRawVarint32Size(headerLen);

int callId = header.getCallId();
Call call = calls.get(callId);
RpcStatusProto status = header.getStatus();

// 如果调用成功, 则读取响应消息, 在 call 实例中设置
if (status == RpcStatusProto.SUCCESS) {
    Writable value = ReflectionUtils.newInstance(valueClass, conf);
    value.readFields(in);           // 读取响应消息
    calls.remove(callId);
    call.setRpcResponse(value);

    // ....
}
} else { // RPC 调用失败
    if (totalLen != headerLen) {
        throw new RpcClientException(
            "RPC response length mismatch on rpc error");
    }

    // 取出响应中的异常消息
    final String exceptionClassName = header.hasExceptionClassName() ?
        header.getExceptionClassName() :
            "ServerDidNotSetExceptionClassName";
    final String errorMsg = header.hasErrorMsg() ?
        header.getErrorMsg() : "ServerDidNotSetErrorMsg";
    final RpcErrorCodeProto erCode =
        (header.hasErrorDetail() ? header.getErrorDetail() : null);
    // 构造异常
    RemoteException re =
    ( (erCode == null) ?
        new RemoteException(exceptionClassName, errorMsg) :
        new RemoteException(exceptionClassName, errorMsg, erCode));
    // 在 Call 对象中设置异常
    if (status == RpcStatusProto.ERROR) {
        calls.remove(callId);
        call.setException(re);
    } else if (status == RpcStatusProto.FATAL) {
        // Close the connection
        markClosed(re);
    }
}
```

```

    }
    } catch (IOException e) {
        markClosed(e);
    }
}

```

保存好响应消息或者异常之后,在 Call 对象上等待的发送请求线程会被唤醒,Client.call()方法会从 Call 对象中取出保存的响应消息并返回,如果 Call 对象中保存的是异常,则直接抛出异常。Client.call()方法中与响应消息处理相关的代码如下:

```

public Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,
    ConnectionId remoteId, int serviceClass,
    AtomicBoolean fallbackToSimpleAuth) throws IOException {

```

```

    // 发送请求

```

```

    //

```

```

    boolean interrupted = false;

```

```

    synchronized (call) {

```

```

        while (!call.done) {

```

```

            try {

```

```

                call.wait();

```

```

                // 等待结果

```

```

            } catch (InterruptedException ie) {

```

```

                interrupted = true;

```

```

            }

```

```

    // 提取出 Call 对象中保存的异常,直接抛出

```

```

    if (call.error != null) {

```

```

        if (call.error instanceof RemoteException) {

```

```

            call.error.fillInStackTrace();

```

```

            throw call.error;

```

```

        } else {

```

```

            InetAddress address = connection.getRemoteAddress();

```

```

            throw NetUtils.wrapException(address.getHostAddress(),

```

```

                address.getPort(),

```

```

                NetUtils.getHostname(),

```

```

                0,

```

```

                call.error);

```

```

        }

```

```

    } else { // 如果成功执行,则返回 Call 对象中保存的 RPC 响应消息

```

```

        return call.getRpcResponse();

```

```

    }

```

```

}

```

至此,一个 Client 发送 RPC 请求以及接收 RPC 响应的流程我们就介绍完了。

2.3.3 Server 类实现

在 RPC 类实现小节中我们介绍了 RPC 服务器端代码获取 Server 对象的流程，服务器端代码获取了 Server 对象后，会启动这个 Server 对象监听网络上的 RPC 请求并触发响应操作。本节就介绍 Server 类的具体实现。

为了提高性能，Server 类采用了很多技术来提高并发能力，包括线程池、JavaNIO 提供的 Reactor 模式等，其中 Reactor 模式贯穿了整个 Server 的设计。

1. Reactor 模式

RPC 服务器端代码的处理流程与所有网络程序服务器端的处理流程类似，都分为 5 个步骤：①读取请求；②反序列化请求；③处理请求；④序列化响应；⑤发回响应。对于网络服务器端程序来说，如果对每个请求都构造一个线程响应，那么在负载增加时性能会下降得很快；而如果只用少量线程响应，又会在 IO 阻塞时造成响应流程停滞、吞吐率降低。所以为了解决上述问题，Reactor 模式出现了。

Reactor 模式是一种广泛应用在服务器端的设计模式，也是一种基于事件驱动的设计模式。Reactor 模式的处理流程是：应用程序向一个中间人注册 IO 事件，当中间人监听到这个 IO 事件发生后，会通知并唤醒应用程序处理这个事件。这里的中间人其实是一个不断等待和循环的线程，它接受所有应用程序的注册，并检查应用程序注册的 IO 事件是否就绪，如果就绪了则通知应用程序进行处理。

一个简单的基于 Reactor 模式的网络服务器设计如图 2-16 所示，包括 reactor、acceptor、以及 handler 等模块。reactor 负责监听所有的 IO 事件，当检测到一个新的 IO 事件发生时，reactor 会唤醒这个事件对应的模块处理。acceptor 则负责响应 Socket 连接请求事件，acceptor 会接收请求建立连接，之后构造 handler 对象。handler 对象则负责向 reactor 注册 IO 读事件，然后从网络上读取请求并执行对应的业务逻辑，最后发回响应。使用 Reactor 模式的服务器响应客户端请求的流程可以分为如下几个步骤。

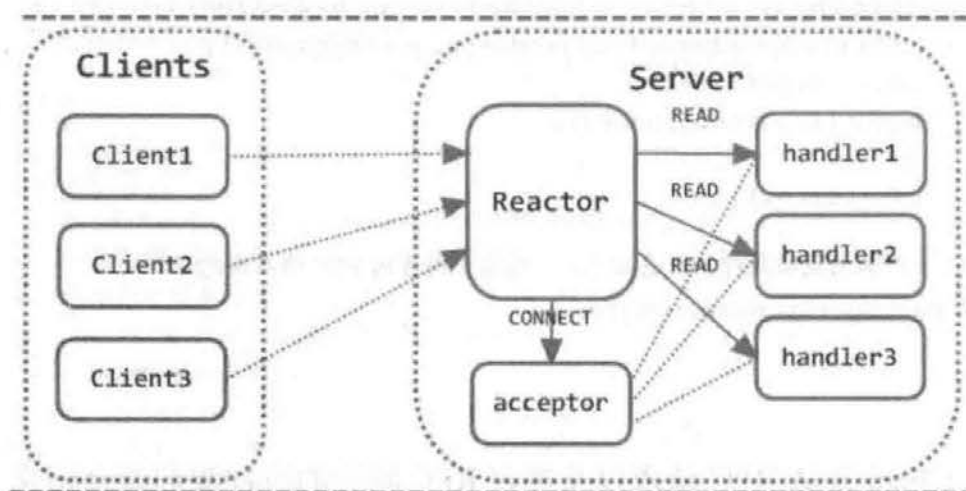


图 2-16 基于 Reactor 模式的网络服务器设计

- 客户端发送 Socket 连接请求到服务器，服务器端的 reactor 对象监听到了这个 IO 请求，由于 acceptor 对象在 reactor 对象上注册了 Socket 连接请求的 IO 事件，所以 reactor 会触发 acceptor 对象响应 Socket 连接请求。
- acceptor 对象会接收来自客户端的 Socket 连接请求，并为这个连接创建一个 handler 对象，handler 对象的构造方法会在 reactor 对象上注册 IO 读事件。
- 客户端在连接建立成功之后，会通过 Socket 发送 RPC 请求。RPC 请求到达 reactor 后，会由 reactor 对象分发（dispatch）到对应的 handler 对象处理。
- handler 对象会从网络上读取 RPC 请求，然后反序列化请求并执行对应的逻辑，最后将响应消息序列化并通过 Socket 发回客户端。至此，一个完整的 RPC 请求流程就结束了。

采用了基于事件驱动模式的 Reactor 结构，服务器只有在指定的 IO 事件发生时才会调用 acceptor 以及 handler 对象提供的方法执行业务逻辑，避免了在 IO 上无谓的阻塞，也就提高了服务器的效率。但是由于上述设计中服务器端只有一个线程，所以就要求 handler 中读取请求、执行请求以及发送响应的流程必须能够迅速处理完，如果在任意一个环节阻塞了，则整个服务器逻辑全部阻塞。所以需要进一步改进架构，也就是使用多线程处理业务逻辑。

对于 handler 处理 RPC 请求的 5 个步骤，我们可以将占用时长较长的读取请求部分以及业务逻辑处理部分交给两个独立的线程池处理。图 2-17 给出了使用多线程的 Reactor 模式的网络服务器结构，readers 线程池中包含若干个执行读取 RPC 请求任务的 Reader 线程，它们会在 Reactor 上注册读取 RPC 请求的 IO 事件，然后从网络中读取 RPC 请求，并将 RPC 请求封装在一个 Call 对象中，最后将 Call 对象放入共享消息队列 MQ 中。而 handlers 线程池则包含若干个处理业务逻辑的 Handler 线程，它们会不断地从共享消息队列 MQ 中取出 RPC 请求，然后执行业务逻辑并向客户端发回响应。这种结构保证了 IO 事件的监听和分发，RPC 请求的读取和响应是在不同的线程中执行的，大大提高了服务器的并发性能。

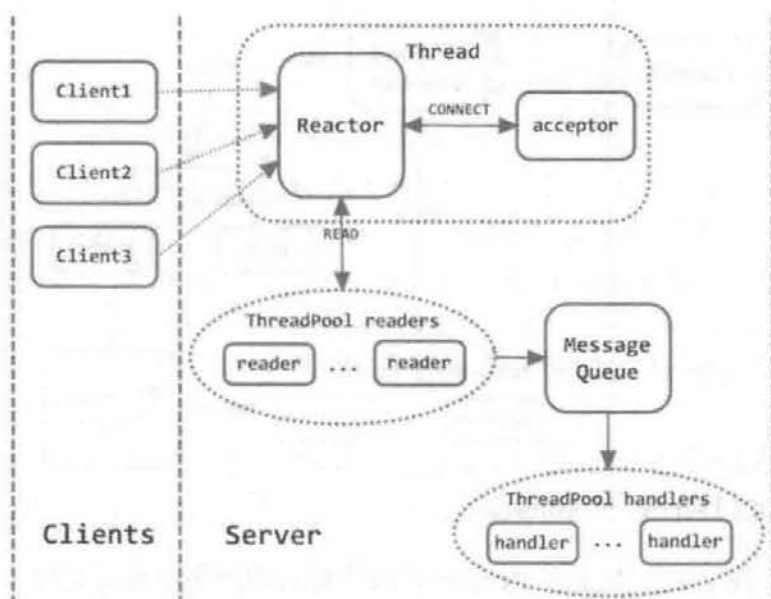


图 2-17 基于多线程 Reactor 模式的网络服务器结构

采用了多线程的 Reactor 模式后, IO 事件的监听、RPC 请求的读取和处理就可以并发地进行了, 但是对于像 Namenode 这种分布式集群中的 Master 节点来说, 同一时间可能有非常多的 Socket 连接请求以及 RPC 请求到达, 这就可能造成 Reactor 在处理 and 分发这些 IO 事件时出现拥塞, 导致服务器整体性能降低。所以我们可以将图 2-17 中的一个 Reactor 对象扩展成多个 Reactor, 它们分别用于并发地监听不同的 IO 事件, 这样也就提高了 IO 事件的处理效率, 同时提高了 RPC 服务器的性能。

图 2-18 给出了使用多个 Reactor 的服务器结构, 这里的主 reactor 负责监听 Socket 连接事件, readReactor 负责监听 IO 读事件, respondSelector 负责监听 IO 写事件。由于同一时间到达 RPC 服务器的 RPC 请求可能很多, 也会造成一个 readReactor 要同时处理多个 IO 读事件的分发, 当系统负载达到一定量时, readReactor 就有可能成为瓶颈。所以我们可以构造多个 readReactor 对象, 不同的 Reader 线程会根据一定的逻辑到不同的 readReactor 上注册 IO 读事件。当 acceptor 建立了 Socket 连接后, 会从 readers 线程池中取出一个 Reader 线程触发读取 RPC 请求的流程。Reader 线程会根据一定的逻辑选出一个 readReactor 对象并在这个 readReactor 对象上注册读取 RPC 请求的 IO 事件, 之后就会由该 readReactor 在网络上监听是否有 RPC 请求到达, 并触发 Reader 线程读取流程了。当 Handler 成功地处理了一个 RPC 请求后, 它会向 respondSelector 注册写 RPC 响应 IO 事件, 当 Socket 输出流管道可以写数据时, Sender 类就可以将响应信息发回客户端了。

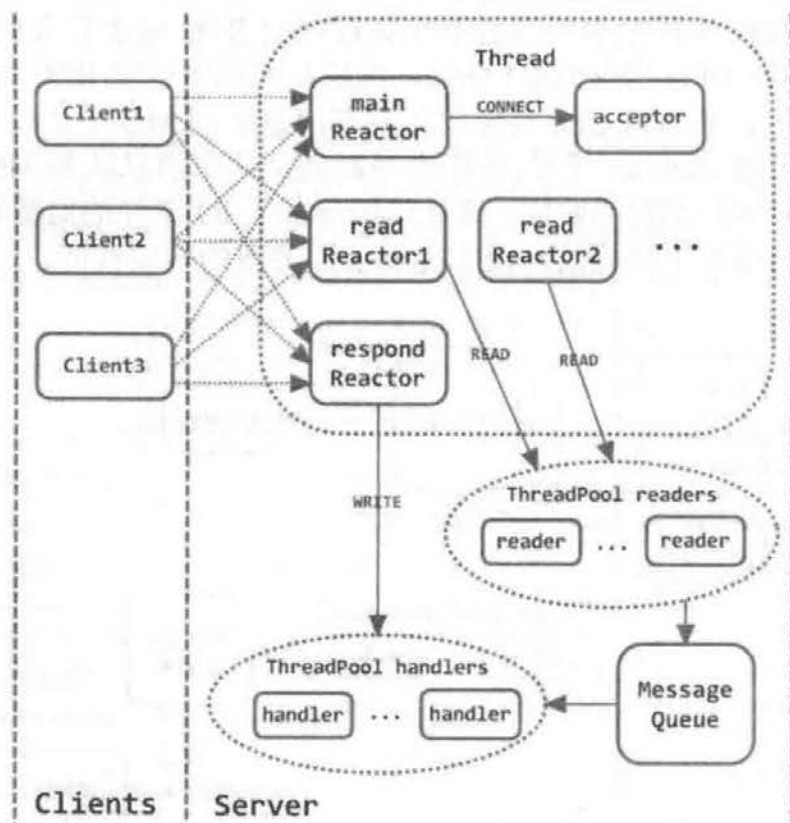


图 2-18 基于多 Reactor 多线程模式的网络服务器结构

Server 类正是采用了多线程加多 Reactor 的设计, 我们在下一小节中介绍 Server 类的结构。

2. Server 类设计

Server 类的设计结构基本与图 2-18 类似，是一个典型的多线程加多 Reactor 的网络服务器结构。Server 定义了如下几个内部类，我们可以对比 Reactor 结构中的模块来理解。

- **Listener:** 类似于 Reactor 模式中的 mainReactor。Listener 对象中存在一个 Selector 对象 acceptSelector，负责监听来自客户端的 Socket 连接请求。当 acceptSelector 监听到连接请求后，Listener 对象会初始化这个连接，之后采用轮询的方式从 readers 线程池中选出一个 Reader 线程处理 RPC 请求的读取操作。
- **Reader:** 与 Reactor 模式中的 Reader 线程相同，用于读取 RPC 请求。Reader 线程类中存在一个 Selector 对象 readSelector，类似于 Reactor 模式中的 readReactor，这个对象用于监听网络中是否有可以读取的 RPC 请求。当 readSelector 监听到有可读的 RPC 请求后，会唤醒 Reader 线程读取这个请求，并将请求封装在一个 Call 对象中，然后将这个 Call 对象放入共享队列 CallQueue 中。
- **Handler:** 与 Reactor 模式中的 Handler 类似，用于处理 RPC 请求并发回响应。Handler 对象会从 CallQueue 中不停地取出 RPC 请求，然后执行 RPC 请求对应的本地函数，最后封装响应并将响应发回客户端。为了能够并发地处理 RPC 请求，Server 中会存在多个 Handler 对象。
- **Responder:** 用于向客户端发送 RPC 响应，读者可能会问，在 Handler 中不是已经发送 RPC 响应了吗？为什么还需要再实现一个 Responder 类？这是因为，在响应很大或者网络条件不佳等情况下，Handler 线程很难将完整的响应发回客户端，这就会造成 Handler 线程阻塞，从而影响 RPC 请求的处理效率。所以 Handler 在没能够将完整的 RPC 响应发回客户端时，会在 Responder 内部的 respondSelector 上注册一个写响应事件，这里的 respondSelector 与 Reactor 模式的 respondSelector 概念相同，当 respondSelector 监听到网络情况具备写响应的条件时，会通知 Responder 将剩余响应发回客户端。

了解了 Server 类的设计后，我们再来学习 Server 处理客户端 RPC 请求的流程。如图 2-19 所示，Server 类处理 RPC 请求的流程可以分为如下几个步骤。

- Listener 线程的 acceptSelector 在 ServerSocketChannel 上注册 OP_ACCEPT 事件，并且创建 readers 线程池。每个 Reader 的 readSelector 此时并不监听任何 Channel。
- Client 发送 Socket 连接请求，触发 Listener 的 acceptSelector 唤醒 Listener 线程。
- Listener 调用 ServerSocketChannel.accept() 创建一个新的 SocketChannel。
- Listener 从 readers 线程池中挑选一个线程，并在 Reader 的 readSelector 上注册 OP_READ 事件。
- Client 发送 RPC 请求数据包，触发 Reader 的 selector 唤醒 Reader 线程。
- Reader 从 SocketChannel 中读取数据，封装成 Call 对象，然后放入共享队列 CallQueue 中。
- 最初，handlers 线程池中的线程都在 CallQueue（调用 BlockingQueue.take()）上阻塞，当有 Call 对象被放入后，其中一个 Handler 线程被唤醒，然后根据 Call 对象的信息

调用 `BlockingService` 对象的 `callBlockingMethod()` 方法。随后, `Handler` 尝试将响应写入 `SocketChannel`。

- 如果 `Handler` 发现无法将响应完全写入 `SocketChannel`, 将在 `Responder` 的 `respondSelector` 上注册 `OP_WRITE` 事件。当 `Socket` 恢复正常时, `Responder` 将被唤醒, 继续写响应。当然, 如果一个 `Call` 响应在一定时间内都无法被写入, 则会被 `Responder` 移除。

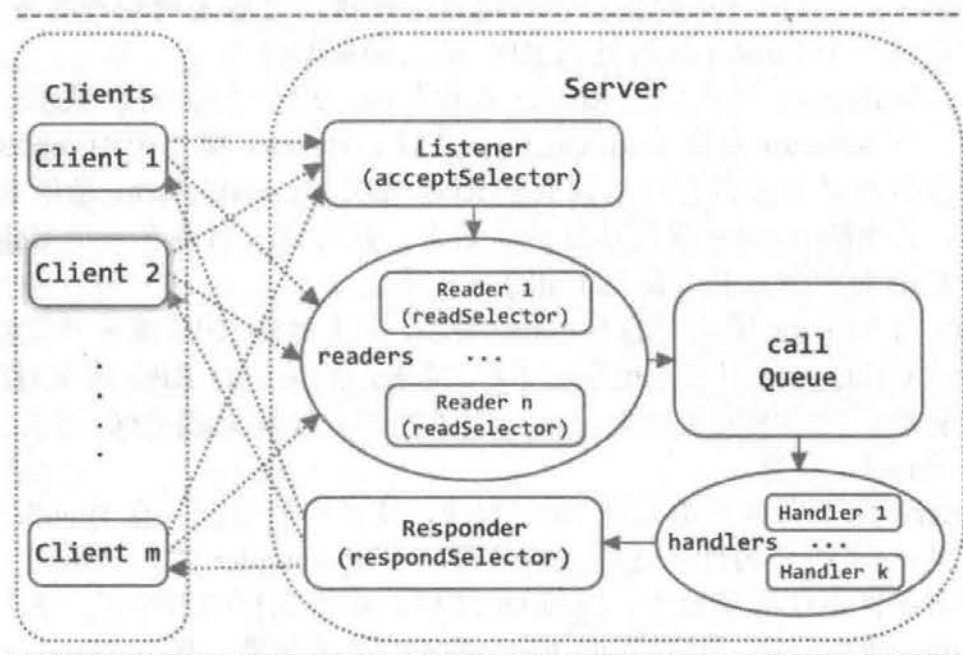


图 2-19 Server 处理客户端 RPC 请求流程图

了解了 `Server` 类的设计以及处理 RPC 请求的流程后, 再来学习 `Server` 类的代码实现就比较简单了, 下面就介绍 `Server` 类的具体实现。

3. Server 类实现

在上一节中我们介绍了 `Server` 类的设计, 包括 `Listener`、`Reader`、`Handler`、`Responder` 类的作用以及相互之间的关系, 本节则从代码角度来分析这几个类的实现。

(1) 内部类 `Listener`

`Listener` 是一个线程类, 整个 `Server` 中只会有一个 `Listener` 线程, 用于监听来自客户端的 `Socket` 连接请求。对于每一个新到达的 `Socket` 连接请求, `Listener` 都会从 `readers` 线程池选择一个 `Reader` 线程来处理。

`Listener` 类中定义了一个 `Selector` 对象, 负责监听 `SelectionKey.OP_ACCEPT` 事件, `Listener` 线程的 `run()` 方法会循环判断是否监听到了 `OP_ACCEPT` 事件, 也就是是否有新的 `Socket` 连接请求到达, 如果有则调用 `doAccept()` 方法响应。 `Listener.run()` 方法的代码如下:

```
public void run() {
    while (running) {
```

```

SelectionKey key = null;
try {
    getSelector().select();
    // 循环判断是否有新的连接建立请求
    Iterator<SelectionKey> iter = getSelector().selectedKeys().iterator();
    while (iter.hasNext()) {
        key = iter.next();
        iter.remove();
        try {
            if (key.isValid()) {
                if (key.isAcceptable())
                    // 如果有, 则调用 doAccept() 方法响应
                    doAccept(key);
            }
        } catch (IOException e) {
        }
        key = null;
    }
} catch (OutOfMemoryError e) {
    // 这里可能出现内存溢出的情况, 要特别注意
    closeCurrentConnection(key, e);
    cleanupConnections(true);
    try { Thread.sleep(60000); } catch (Exception ie) {}
} catch (Exception e) {
    // 捕获到其他异常, 也关闭当前连接
    closeCurrentConnection(key, e);
}
cleanupConnections(false);
}

// running == false 时, 关闭 Listener 线程
// ...
}

```

下面看一下 `doAccept()` 方法的实现。`doAccept()` 方法会接收来自客户端的 Socket 连接请求并初始化 Socket 连接。之后 `doAccept()` 方法会从 `readers` 线程池中选出一个 `Reader` 线程读取来自这个客户端的 RPC 请求。每个 `Reader` 线程都会有一个自己的 `readSelector`, 用于监听是否有新的 RPC 请求到达。所以 `doAccept()` 方法在建立连接并选出 `Reader` 对象后, 会在这个 `Reader` 对象的 `readSelector` 上注册 `OP_READ` 事件。那么这里就有一个问题了, `Reader` 对象在被通知时是怎么知道从哪个 Socket 输入流上读取数据呢? 这里就用到了 `Connection` 类, `Connection` 类封装了 `Server` 与 `Client` 之间的 Socket 连接, `doAccept()` 方法会通过 `SelectionKey` 将新构造的 `Connection` 对象传给 `Reader`, 这样 `Reader` 线程在被唤醒时就可以通过 `Connection` 对象读取 RPC 请求了。`doAccept()` 方法的代码如下:

```

void doAccept(SelectionKey key) throws IOException, OutOfMemoryError {
    // 接收请求, 建立连接
    Connection c = null;

```

```
ServerSocketChannel server = (ServerSocketChannel) key.channel();
SocketChannel channel;
while ((channel = server.accept()) != null) {
    channel.configureBlocking(false);
    channel.socket().setTcpNoDelay(tcpNoDelay);

    // 从 readers 线程池中取出一个 Reader 线程
    Reader reader = getReader();
    try {
        // 唤醒处于等待状态的 readSelector
        reader.startAdd();
        // 注册 IO 读事件
        SelectionKey readKey = reader.registerChannel(channel);
        // 构造 Connection 对象, 添加到 readKey 的附件传递给 Reader 对象
        c = new Connection(readKey, channel, Time.now());
        readKey.attach(c);
        synchronized (connectionList) {
            connectionList.add(numConnections, c);
            numConnections++;
        }
    } finally {
        reader.finishAdd();
    }
}
```

这里的 Reader 其实就是一个独立的线程, 专门负责读操作。我们接下来看 Reader 类的实现。

(2) 内部类 Reader (done)

Reader 也是一个线程类, 每个 Reader 线程都会负责读取若干个客户端连接发来的 RPC 请求。而在 Server 类中会存在多个 Reader 线程构成一个 readers 线程池, readers 线程池并发地读取 RPC 请求, 提高了 Server 处理 RPC 请求的速率。Reader 类定义了自己的 readSelector 字段, 用于监听 SelectionKey.OP_READ 事件。Reader 类还定义了 adding 字段标识是否有任务正在添加到 Reader 线程。

```
private volatile boolean adding = false;
private final Selector readSelector;
```

Reader 线程的主循环则是在 doRunLoop() 方法中实现的, doRunLoop() 方法会监听当前 Reader 对象负责的所有客户端连接中是否有新的 RPC 请求到达, 如果有则读取这些请求, 然后将成功读取的请求用一个 Call 对象封装, 最后放入 callQueue 中等待 Handler 线程处理。doRunLoop() 方法的代码如下:

```
private synchronized void doRunLoop() {
    while (running) {
        SelectionKey key = null;
```

```

try {
    readSelector.select();
    // 有任务添加时等待; 在任务添加完成之后会被唤醒
    while (adding) {
        this.wait(1000);
    }

    // 在当前的 readSelector 上等待可读事件, 也就是有客户端 RPC 请求到达
    Iterator<SelectionKey> iter = readSelector.selectedKeys().iterator();
    while (iter.hasNext()) {
        key = iter.next();
        iter.remove();
        if (key.isValid()) {
            if (key.isReadable()) {
                // 有可读事件时, 调用 doRead() 方法处理
                doRead(key);
            }
        }
        key = null;
    }
} catch (InterruptedException e) {
    if (running) { // 出现异常, 则记录在日志中
        LOG.info(getName() + " unexpectedly interrupted", e);
    }
} catch (IOException ex) {
    LOG.error("Error in Reader", ex);
}
}
}

```

doRead()方法负责读取 RPC 请求, 虽然 readSelector 监听到了 RPC 请求的可读事件, 但是 doRead()方法此时并不知道这个 RPC 请求是由哪个客户端发送来的, 所以 doRead()方法首先会调用 SelectionKey.attachment()方法获取 Listener 对象构造的 Connection 对象, Connection 对象中封装了 Server 与 Client 之间的网络连接, 之后 doRead()方法只需调用 Connection.readAndProcess()方法就可以读取 RPC 请求了, 这里的设计非常的巧妙。

```

void doRead(SelectionKey key) throws InterruptedException {
    int count = 0;
    // 通过 SelectionKey 获取 Connection 对象
    Connection c = (Connection)key.attachment();
    if (c == null) {
        return;
    }
    c.setLastContact(Time.now());

    try {
        count = c.readAndProcess(); // 调用 Connection.readAndProcess 处理读取请求
    } catch (InterruptedException ieo) {
    }
}

```

```
        throw ieo;
    } catch (Exception e) {
        count = -1;
    }
    if (count < 0) {
        closeConnection(c);
        c = null;
    }
    else {
        c.setLastContact(Time.now());
    }
}
```

下面我们看一下 Connection 类的实现。

(3) 内部类 Connection (done)

Connection 类维护了 Server 与 Client 之间的 Socket 连接。Reader 线程会调用 readAndProcess()方法从 IO 流中读取一个 RPC 请求。

readAndProcess()方法会首先从 Socket 流中读取连接头域 (connectionHeader)，然后读取一个完整的 RPC 请求，最后调用 processOneRpc()方法处理这个 RPC 请求。processOneRpc()方法会读取出 RPC 请求头域，然后调用 processRpcRequest()处理 RPC 请求体。这里特别注意，如果在处理过程中抛出了异常，则直接通过 Socket 返回 RPC 响应 (带有 Server 异常信息的响应)。processOneRpc()方法的代码如下：

```
private void processOneRpc(byte[] buf)
    throws IOException, WrappedRpcServerException, InterruptedException {
    int callId = -1;
    int retry = RpcConstants.INVALID_RETRY_COUNT;
    try {
        final DataInputStream dis =
            new DataInputStream(new ByteArrayInputStream(buf));
        // 解析出 RPC 请求头域
        final RpcRequestHeaderProto header =
            decodeProtobufFromStream(RpcRequestHeaderProto.newBuilder(), dis);
        callId = header.getCallId(); // 从 RPC 请求头域中提取出 callId
        retry = header.getRetryCount(); // 从 RPC 请求头域中提取出重试次数
        checkRpcHeaders(header);

        // 处理 RPC 请求头域异常的情况
        if (callId < 0) { during connection setup
            processRpcOutOfBandRequest(header, dis);
        } else if (!connectionContextRead) {
            throw new WrappedRpcServerException(
                RpcErrorCodeProto.FATAL_INVALID_RPC_HEADER,
                "Connection context not established");
        } else {
            // 如果 RPC 请求头域正常，则直接调用 processRpcRequest 处理 RPC 请求体
        }
    }
}
```

```

        processRpcRequest(header, dis);
    }
} catch (WrappedRpcServerException wrse) { // 直接发回异常, 通知 Client
    Throwable ioe = wrse.getCause();
    final Call call = new Call(callId, retry, null, this);
    setupResponse(authFailedResponse, call,
        RpcStatusProto.FATAL, wrse.getRpcErrorCodeProto(), null,
        ioe.getClass().getName(), ioe.getMessage());
    // 通过 Socket 返回这个带有异常信息的 RPC 响应
    responder.doRespond(call);
    throw wrse;
}
}
}

```

对于一个正常的 RPC 请求, `processOneRpc()` 方法会调用 `processRpcRequest()` 方法处理。`processRpcRequest()` 会从输入流中解析出完整的请求对象 (包括请求元数据以及请求参数), 然后根据 RPC 请求头的信息 (包括 `callId`) 构造 `Call` 对象 (`Call` 对象保存了这次调用的所有信息), 最后将这个 `Call` 对象放入 `callQueue` 队列中保存, 等待 `Handler` 线程处理。

```

private void processRpcRequest(RpcRequestHeaderProto header,
    DataInputStream dis) throws WrappedRpcServerException,
    InterruptedException {

    // ...
    Writable rpcRequest;
    try { // 读取 RPC 请求体
        rpcRequest = ReflectionUtils.newInstance(rpcRequestClass, conf);
        rpcRequest.readFields(dis);
    } catch (Throwable t) { // includes runtime exception from newInstance
        // 出现异常则直接抛出, 在上一层捕获异常
        throw new WrappedRpcServerException(
            RpcErrorCodeProto.FATAL_DESERIALIZING_REQUEST, err);
    }

    // 构造 Call 对象封装 RPC 请求信息
    Call call = new Call(header.getCallId(), header.getRetryCount(),
        rpcRequest, this, ProtoUtil.convert(header.getRpcKind()), header
            .getClientId().toByteArray());
    // 将 Call 对象放入 callQueue 中, 等待 Handler 处理
    callQueue.put(call);
    incRpcCount();
}

```

下面我们就学习 `Handler` 线程是如何处理这个 `Call` 对象的。

(4) 内部类 `Handler` (done)

`Handler` 类也是一个线程类, 负责执行 RPC 请求对应的本地函数, 然后将结果发回客户端。在 `Server` 类中会有多个 `Handler` 线程, 它们并发地处理 RPC 请求。

Handler 线程类的主方法会循环从共享队列 callQueue 中取出待处理的 Call 对象，然后调用 Server.call()方法执行 RPC 调用对应的本地函数，如果在调用过程中发生异常，则将异常信息保存下来。接下来 Handler 会调用 setupResponse()方法构造 RPC 响应，并调用 responder.doRespond()方法将响应发回。

```
while (running) {
    try {
        // ...
        // 从 callQueue 中取出请求
        final Call call = callQueue.take();
        try {
            if (call.connection.user == null) {
                // 通过 call()发起本地调用，并返回结果
                value = call(call.rpcKind, call.connection.protocolName, call.rpcRequest,
                    call.timestamp);
            }
            // ...
        } catch (Throwable e) {
            // ...
            // 如果在调用过程中发生异常，则将异常信息保存下来
            if (e instanceof RpcServerException) {
                RpcServerException rse = ((RpcServerException)e);
                returnStatus = rse.getRpcStatusProto();
                detailedErr = rse.getRpcErrorCodeProto();
            } else {
                returnStatus = RpcStatusProto.ERROR;
                detailedErr = RpcErrorCodeProto.ERROR_APPLICATION;
            }
            errorClass = e.getClass().getName();
            error = StringUtils.stringifyException(e);
            String exceptionHdr = errorClass + ": ";
            if (error.startsWith(exceptionHdr)) {
                error = error.substring(exceptionHdr.length());
            }
        }
        CurCall.set(null);
        synchronized (call.connection.responseQueue) {
            // 构造 RPC 响应，如果调用正常就返回结果，有异常则返回异常信息
            setupResponse(buf, call, returnStatus, detailedErr,
                value, errorClass, error);

            // 调用 responder.doRespond()返回响应
            responder.doRespond(call);
        }
    } catch (InterruptedException e) {
        // ...
    } catch (Exception e) {
        // ...
    }
}
```

`Server.call()`方法执行本地函数的部分我们在服务器获取 `Server` 对象小节中已经介绍过了，这里不再重复介绍。

(5) 内部类 `Responder` (done)

内部类 `Responder` 也是一个线程类，`Server` 端仅有一个 `Responder` 对象，`Responder` 内部包含一个 `Selector` 对象 `responseSelector`，用于监听 `SelectionKey.OP_WRITE` 事件。当网络环境不佳或者响应信息太大时，`Handler` 线程可能无法发送完整的响应信息到客户端，这时 `Handler` 会在 `Responder.responseSelector` 上注册 `SelectionKey.OP_WRITE` 事件，`responseSelector` 会循环监听网络环境是否具备发送数据的条件，之后 `responseSelector` 会触发 `Responder` 线程发送未完成的响应结果到客户端。由于篇幅原因，这里不再贴出 `Responder` 类的代码实现，感兴趣的读者可以自行阅读。

第3章 Namenode（名字节点）

HDFS 集群是以 Master/Slave 模式运行的，主要有两类节点：Namenode（名字节点）和 Datanode（数据节点）。Namenode 是 HDFS 中的主节点，本章介绍 Namenode 的实现。

对于 Namenode 的分析，我们分为以下几个部分来介绍。

- 文件系统目录树管理：HDFS 的目录和文件在内存中是以一棵树的形式存储的，这个目录树结构是由 Namenode 维护的，Namenode 会修改这个树形结构以对外提供添加和删除文件等操作功能。文件系统目录树上的节点还保存了 HDFS 文件与数据块的对应关系，我们知道 HDFS 中的每个文件都是被拆分成若干数据块冗余存放的，文件与数据块的对应关系也是由 Namenode 维护的。
- 数据块以及数据节点管理：HDFS 中的数据块是冗余备份在集群中的数据节点上的，所以 Namenode 还需要维护数据块与数据节点之间的对应关系。这里的对应关系包括两个部分：①数据块存放在哪些数据节点上；②一个数据节点上保存了哪些数据块。
- 租约管理：租约是 Namenode 给予租约持有者（LeaseHolder，一般是 HDFS 客户端）在规定时间内拥有文件权限（写文件）的合同，Namenode 会执行租约的发放、回收、检查以及恢复等操作。
- 缓存管理：Hadoop 2.3.0 版本新增了集中式缓存管理功能（Centralized Cache Management），允许用户将一些文件和目录保存到 HDFS 缓存中。HDFS 的集中式缓存是由分布在 Datanode 上的堆外内存组成的，并且由 Namenode 统一管理。
- FSNamesystem：Namenode 涉及很多 HDFS 的处理逻辑，例如读文件、写文件、追加写文件等，Namenode 的 FSNamesystem 类是管理这些逻辑的门面类，也是 Namenode 中最重要的一个类。
- Namenode 的启动和停止：Hadoop 2.X 实现中提供了 HA 功能，HA 集群中会存在两种状态的 Namenode：Active Namenode 作为服务节点，Standby Namenode 作为热备节点。Namenode 在启动时会先进入安全模式，在安全模式中的 Namenode 不会接受客户端对命名空间的修改，Namenode 成功启动后会离开安全模式进入 Standby 状态。

下面我们分别介绍上述几个部分在 Namenode 中的实现。

3.1 文件系统目录树

Namenode 最重要的两个功能之一就是维护文件系统的命名空间（namesystem）。HDFS

文件系统的命名空间(namespace)是以“/”为根的整个目录树,是通过 FSDirectory 类来管理的。

HDFS 文件系统的命名空间在 Namenode 的内存中是以一颗树的结构来存储的。在 HDFS 中,不管是目录还是文件,在文件系统目录树中都被看作是一个 INode 节点。如果是目录,则其对应的类为 INodeDirectory;如果是文件,则其对应的类为 INodeFile。INodeDirectory 以及 INodeFile 类都是 INode 的派生类。INodeDirectory 中包含一个成员集合变量 children,如果该目录下有子目录或者文件,其子目录或文件的 INode 引用就会被保存在 children 集合中。HDFS 就是通过这种方式来维护整个文件系统的目录结构的。

HDFS 会将命名空间保存到 Namenode 的本地文件系统上一个叫 fsimage(命名空间镜像)的文件中。利用这个文件,Namenode 每次重启时都能将整个 HDFS 的命名空间重构,fsimage 文件的操作由 FSImage 类负责。另外,对 HDFS 的各种操作,Namenode 都会在操作日志(editlog)中进行记录,以便周期性地将该日志与 fsimage 进行合并生成新的 fsimage。该日志文件也在 Namenode 的本地文件系统中保存,叫 editlog 文件,editlog 的相关操作由 FSEditLog 类管理。

如图 3-1 所示,FSDirectory 维护着文件系统目录树的根节点(这个根节点是整个文件系统的 root,是一个 INodeDirectory 类型),本章就以 INode 作为分析入口。

FSDirectory	
① CHECK_RESERVED_FILE_NAMES	boolean
② DOT_RESERVED_STRING	String
③ DOT_RESERVED_PATH_PREFIX	String
④ DOT_RESERVED	byte[]
⑤ RAW_STRING	String
⑥ RAW	byte[]
⑦ DOT_INODES_STRING	String
⑧ DOT_INODES	byte[]
⑨ KEYID_XATTR	XAttr
⑩ UNREADABLE_BY_SUPERUSER_XATTR	XAttr
⑪ rootDir	INodeDirectory
⑫ namespace	FSNamesystem
⑬ skipQuotaCheck	boolean
⑭ maxComponentLength	int
⑮ maxDirItems	int
⑯ lslimit	int
⑰ contentCountLimit	int
⑱ inodeMap	INodeMap
⑲ yieldCount	long
⑳ inodeXattrLimit	int
㉑ dirLock	ReentrantReadWriteLock
㉒ ezManager	EncryptionOneManager
㉓ nameCache	NameCache<ByteArray>

图 3-1 FSDirectory 字段列表

3.1.1 INode 相关类

看到 INode,我们可以马上联想到 Linux 的 inode,即索引节点。索引节点保存了 Linux 文件的元信息,如文件类型与权限、所有者标识和以字节为单位的文件长度等。在索引节点的后半部分,则存放着数据块索引,也就是文件或目录数据在磁盘上的位置。

HDFS 就是借鉴了 Linux 的 inode,将 HDFS 中文件和目录的抽象类命名为 INode。在 HDFS 的文件系统目录树中,不管是目录还是文件,都会被看作是一个 INode 类的引用。INode 类

是一个抽象类，是 `InodeDirectory` 和 `InodeFile` 的父类。如果是目录，则其实际类为 `InodeDirectory`；如果是文件，则其对应的类为 `InodeFile`。`InodeDirectory` 中包含一个成员集合变量 `children`，如果该目录下有子目录或者文件，其子目录或文件的引用就会保存在 `children` 集合中。

HDFS 中 `Inode` 继承关系图如图 3-2 所示，我们依次介绍 `Inode` 相关的所有类。

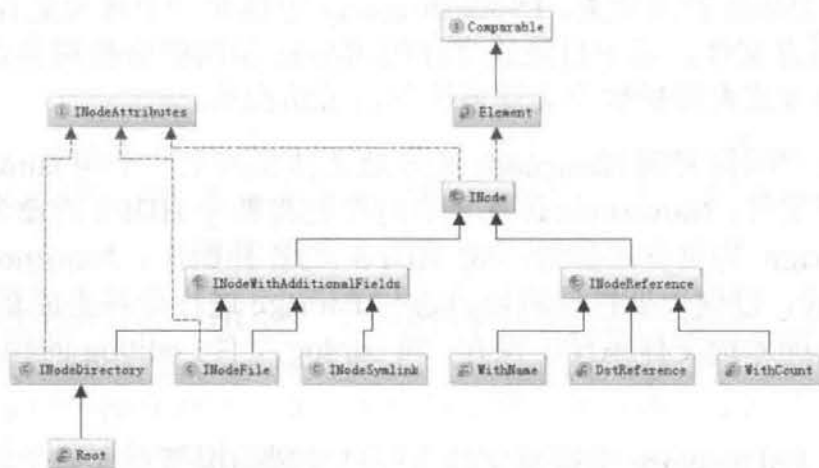


图 3-2 Inode 继承关系图

1. Inode 抽象类

`Inode` 类是整个 `Inode` 体系的根接口，它是一个抽象类，保存了 HDFS 目录和文件的所有共同属性，包括当前节点的父节点的 `Inode` 对象的引用（只能是 `InodeDirectory` 类或者 `InodeReference` 类）、文件/目录名、用户组、访问权限、最后修改时间、上次访问时间、完整路径名、文件扩展属性等。

(1) Inode 类方法

如图 3-2 所示，`Inode` 类实现了 `InodeAttributes` 接口。这个接口包括以下 7 个字段的 `get` 方法。

- `userName`: 文件/目录所属用户名。
- `groupName`: 文件/目录所属组名。
- `fsPermission`: 文件/目录访问权限。
- `aclFeature`: 安全相关。
- `modificationTime`: 文件/目录上次修改时间。
- `accessTime`: 文件/目录上次访问时间。
- `XAttrFeature`: 当前文件/目录的扩展属性 (`ExtendedAttributes`)。

文件系统扩展属性是目前流行的 POSIX 系统中文件系统具有的一项特殊功能，可以给文件、文件夹添加额外的 `key/value` 键值对，键和值都是字符串并且有一定长度的限制。文件系统扩展属性使得现有的文件系统得以支持在原始设计中未提供的功能。

INode 抽象类除了实现 INodeAttributes 接口中的方法外, 还定义了 INode 元信息的 get 与 set 接口方法。INode 元信息包括如下一些信息。

- id: INode 的 id。
- name: 文件/目录的名称。
- fullPathName: 文件/目录的完整路径。
- parent: 文件/目录的父节点。

同时 INode 还提供了如下几个基本的判断方法。

- isFile(): 判断是否为文件。
- isDirectory(): 判断是否为目录。
- isSymlink(): 判断是否为符号链接。
- isRoot(): 判断是否为文件系统目录树的根节点。

这里需要注意的是, INode 类的设计采用了模板模式。INode 类定义的方法多为两个, 其中一个为 final 的接口方法, 用于规范接口的调用; 另一个则是 abstract 的抽象方法, 抽象方法留给子类具体实现。INode 类实现的 INodeAttributes 接口中定义的方法就是采用了这种模式, 将 userName 等字段的定义留给子类实现, 这里我们以 setUser() 方法为例。

```
// 抽象方法, 具体实现留给子类
abstract void setUser(String user);

// 模板方法, 是 final 的, 不可以继承, 供接口调用
final INode setUser(String user, int latestSnapshotId)
    throws QuotaExceededException {
    recordModification(latestSnapshotId);
    setUser(user);
    return this;
}
```

(2) INode 类字段

INode 类中只有一个字段, 就是 parent, 表明当前 INode 的父目录。HDFS 中除了根目录外, 其他所有的文件与目录都存在一个父目录。注意, 父目录的类型只能是 INodeDirectory 类或者 INodeReference 类之一。

```
private INode parent = null;
```

2. INodeWithAdditionalFields 类

上一节介绍了 INode 这个根接口类, 但是 INode 类只定义了一个字段 parent, 其余字段的值都是通过抽象的 get() 方法获得的, 并且留给了子类来定义。如图 3-3 所示, INodeWithAdditionalFields 类就定义了这些字段——id、name、permission、modificationTime、accessTime 等, 并且覆盖了 INode 中对应的抽象方法。

INodeWithAdditionalFields	
id	long
name	byte[]
permission	long
modificationTime	long
accessTime	long
next	LinkedElement
EMPTY_FEATURE	Feature[]
features	Feature[]

图 3-3 INodeWithAdditionalFields 类字段

这里比较有意思的是 permission 字段和 features 字段, 我们重点讲一下这两个字段。

(1) permission 字段

permission 字段主要包括 3 个部分的信息：用户信息、用户组信息和权限信息。

permission 字段是 long 类型的，其中前 16 个比特用来存放文件模式标识（mode，类似于 Linux 中的 777），中间 25 个比特用来存放用户组标识（group），最后 23 个比特用来存放用户名标识（user）。枚举类 `PermissionStatusFormat` 就是用来解析以及处理 permission 字段的工具类，提供了获取 permission 字段中 mode、group、user 部分对应的文件模式、用户组名以及用户名的方法。在 HDFS 中，用户名和用户标识的对应关系、用户组名和用户组标识的对应关系都保存在 `SerialNumberManager` 类中。通过 `SerialNumberManager` 类，名字节点不必在 `INode` 对象中保存字符串形式的用户名和用户组名，只需将整型的用户名标识和用户组名标识放入 permission 字段中即可。这种设计节省了 `INode` 对象对内存的占用，是一个非常巧妙的优化。

`PermissionStatusFormat` 类的代码如下所示，可以看到 `PermissionStatusFormat` 枚举类使用到了 `LongBitFormat` 类来存储与操作底层的 permission 字段。`LongBitFormat` 的 `combine()` 方法和 `retrieve()` 方法主要是对底层的 permission 字段进行掩码操作、移位操作以及逻辑与操作，比较简单，这里就不详细叙述了，感兴趣的读者可以自己学习。

```
static enum PermissionStatusFormat {
    MODE(null, 16),
    GROUP(MODE.BITS, 25),
    USER(GROUP.BITS, 23);

    final LongBitFormat BITS;

    private PermissionStatusFormat(LongBitFormat previous, int length) {
        BITS = new LongBitFormat(name(), previous, length, 0);
    }

    // 提取最后 23 个比特的 user 信息，并通过 SerialNumberManager 获取用户名
    static String getUser(long permission) {
        // 首先获取 permission 中最后 23 个比特的用户标识
        final int n = (int)USER.BITS.retrieve(permission);
        // 通过 SerialNumberManager 类获取用户标识对应的用户名
        return SerialNumberManager.INSTANCE.getUser(n);
    }

    // 提取中间 25 个比特的 group 信息，并通过 SerialNumberManager 获取用户组名
    static String getGroup(long permission) {
        // 首先获取 permission 中间 25 个比特的用户组标识
        final int n = (int)GROUP.BITS.retrieve(permission);
        // 使用 SerialNumberManager 获取用户组标识对应的用户组名
        return SerialNumberManager.INSTANCE.getGroup(n);
    }

    // 提取前 16 个比特的 mode 信息
    static short getMode(long permission) {
```

```

    return (short)MODE.BITS.retrieve(permission);
}

// 将一个 PermissionStatus 类, 转换成 long 类型的 permission 信息
static long toLong(PermissionStatus ps) {
    long permission = 0L;
    final int user = SerialNumberManager.INSTANCE.getUserSerialNumber(
        ps.getUserName());
    permission = USER.BITS.combine(user, permission);
    final int group = SerialNumberManager.INSTANCE.getGroupSerialNumber(
        ps.getGroupName());
    permission = GROUP.BITS.combine(group, permission);
    final int mode = ps.getPermission().toShort();
    permission = MODE.BITS.combine(mode, permission);
    return permission;
}
}

```

(2) features 字段

在 HDFS 2.2 版本前, INode 的子类还包括 InodeDirectoryWithQuota 和 InodeFileUnder Constrution, 这两个类分别用于描述带有磁盘配额的目录以及正在进行写操作的文件。但在 HDFS 2.6 版本中, 这两个子类从 INode 的类图中消失了。这是因为 HDFS 2.6 版本的代码将磁盘配额、正在构建(Under Constrution)、快照(Snapshot)等功能抽象成 INode 的特性(Feature), 可以将特性添加到 INode 上, 使得该 INode 具备该特性的功能。

Feature 的实现细节我们在 Feature 相关类小节中再做介绍, 这里先回到 INodeWithAdditionalFields 类, INodeWithAdditionalFields.features 字段就是用来保存当前 INode 拥有哪些特性的字段, 它是一个 Feature 类型的数组, 默认值是空数组 EMPTY_FEATURE。

```

private static final Feature[] EMPTY_FEATURE = new Feature[0];
protected Feature[] features = EMPTY_FEATURE;

```

INodeWithAdditionalFields 提供了向 INode 添加、删除以及查询特性的方法, 这些方法的底层还是对 features 数组的操作。以 addFeature()方法为例, addFeature()方法用于向当前 INode 节点添加一个新的特性, addFeature()方法向 features 数组中添加了一个新的 Feature 元素, 代码如下:

```

protected void addFeature(Feature f) {
    int size = features.length;
    Feature[] arr = new Feature[size + 1]; // 申请一个更大的数组
    if (size != 0) {
        System.arraycopy(features, 0, arr, 0, size); // 将原 features 数组拷贝到现在数组中
    }
    arr[size] = f; // 将新的 Feature 对象添加到数组中
    features = arr; // 用新数组替换 features 引用
}

```

3. INodeDirectory 类

INodeDirectory 抽象了 HDFS 文件系统中的目录，目录是文件系统中的虚拟容器，里面保存了一组文件和其他一些目录。在 INodeDirectory 的实现中，添加了成员变量 children，用来保存目录中所有子目录项的 INode 对象。

```
// 使用一个 children 字段保存该目录中所有孩子节点的 INode 对象
private List<INode> children = null;
```

如图 3-4 所示，在 HDFS 2.6 版本的 INodeDirectory 代码中添加了 Feature 相关、Snapshot 相关的方法，我们可以将 INodeDirectory 中的方法分为如下几类。

- children 字段的增、删、改、查方法：用于向当前目录添加、删除、替换、查找子目录项等操作，包括 addChild()、removeChild()、replaceChild()、getChild()、clearChildren()、cleanSubtreeRecursively()等方法。
- 特性 (Feature) 相关方法：用于向当前 INodeDirectory 添加新的 Feature 对象，以及获取指定 Feature 对象的方法，包括 addDirectoryWithQuotaFeature()、getDirectoryWithQuotaFeature()、addSnapshottableFeature()、getDirectorySnapshottableFeature()、addSnapshotFeature()、getDirectoryWithSnapshotFeature()等方法。
- 快照 (Snapshot) 相关方法：用于向当前目录添加、删除或者更改快照等操作，包括 isSnapshottable()、getSnapshot()、setSnapshotQuota()、addSnapshot()、removeSnapshot()等方法。



图 3-4 INodeDirectory 类结构

(1) 子目录项相关方法

INodeDirectory 作为目录容器,最主要的功能就是维护目录中保存的文件以及子目录,也就是维护 children 字段。INodeDirectory 提供了创建子目录项、删除子目录项以及查询子目录项等方法,这些方法的底层实现都是操作 children 字段,这个小节我们就学习关于 children 字段的操作方法。

需要特别注意的是,在 HDFS 2.6 版本的代码中引入了快照 (Snapshot) 特性。当 HDFS 管理员在当前 INodeDirectory 上建立了快照之后,任何对于子目录项的操作都需要在快照中进行记录。这里我们先介绍没有建立快照的子目录项操作方法,对于有快照情况下的对应方法我们在 SnapshotFeature 实现小节中介绍。

没有开启快照功能的 INodeDirectory 中子目录项的操作方法都比较简单,就是操作 children 这个集合。我们以 addChild()和 removeChild()两个方法为例,其他方法请读者自行参考源代码。addChild()方法首先在 children 列表中找到插入位置,然后将 INode 对象插入到 children 集合中。removeChild()方法也是一样的,首先找到指定 INode 对象在 children 集合中的位置,然后删除。addChild()和 removeChild()方法的代码如下:

```
public boolean addChild(INode node) {
    // 首先找到 INode 节点在 children 列表中的位置
    final int low = searchChildren(node.getLocalNameBytes());
    if (low >= 0) {
        return false;
    }
    // 调用 addChild()方法将 INode 节点插入到 children 列表的 low 位置
    addChild(node, low);
    return true;
}
```

```
public boolean removeChild(final INode child) {
    // 找到 INode 节点在 children 列表中的位置
    final int i = searchChildren(child.getLocalNameBytes());
    if (i < 0) {
        return false;
    }
    // 从 children 列表中删除
    final INode removed = children.remove(i);
    Preconditions.checkState(removed == child);
    return true;
}
```

(2) 特性相关方法

在 InodeWithAdditionalFields 类小节中,我们已经介绍过可以向 INode 添加特性 (Feature)。对于 INodeDirectory,也就是 HDFS 目录来讲,我们可以对目录添加磁盘配额特性 (DirectoryWithQuotaFeature) 以及快照特性 (DirectoryWithSnapshotFeature&DirectorySnapshottableFeature)。添加特性方法的实现也比较简单,直接构造特性对应的 Feature 对象,

然后调用 `INodeWithAdditionalFields.addFeature()` 方法将构造的 `Feature` 对象添加到 `features` 集合中即可。查找 `Feature` 对象，则直接调用 `INodeWithAdditionalFields.getFeature()` 从 `features` 集合中找到对应的 `Feature` 对象返回即可。这里以 `addDirectoryWithQuotaFeature()` 方法和 `getDirectoryWithQuotaFeature()` 方法为例，其他方法请读者自行参考源代码。

```
// 向当前目录添加磁盘配额特性
DirectoryWithQuotaFeature addDirectoryWithQuotaFeature(
    long nsQuota, long dsQuota) {
    // 构造 DirectoryWithQuotaFeature 对象，调用 addFeature() 方法添加到 features 集合中
    final DirectoryWithQuotaFeature quota = new DirectoryWithQuotaFeature(
        nsQuota, dsQuota);
    addFeature(quota);
    return quota;
}

// 获取当前目录磁盘配额特性对应的 DirectoryWithQuotaFeature 对象
public final DirectoryWithQuotaFeature getDirectoryWithQuotaFeature() {
    // 调用 getDirectoryWithQuotaFeature() 从 features 集合中查找 Feature 对象
    return getFeature(DirectoryWithQuotaFeature.class);
}
```

(3) 快照相关方法

在 HDFS 2.6 版本中，我们可以对文件系统中的任意目录建立快照，之后任何改变这个目录或者目录中子目录项的操作都会被记录在快照中，以便于日后恢复。`INodeDirectory` 中另一部分功能就是快照相关方法，主要包括 `addSnapshot()`、`getDiffs()` 等方法的实现。这一部分内容我们将在 `SnapshotFeature` 实现小节中介绍。

4. INodeFile 类

在文件系统目录树中，使用 `INodeFile` 类抽象一个 HDFS 文件，`INodeFile` 类继承自 `INodeWithAdditionalFields` 类。

`INodeFile` 类中保存了 HDFS 文件最重要的两个信息：文件头 `header` 字段和文件对应的数据块信息 `blocks` 字段。`header` 字段保存了当前文件有多少个副本，以及文件数据块的大小（`header` 字段的处理类似于 `INode` 中的 `permission` 字段，前 4 个比特用于保存存储策略，中间 12 个比特用于保存文件备份系数，后 48 个比特用于保存数据块大小。使用内部类 `HeaderFormat` 处理）；`blocks` 字段是一个 `BlockInfo` 类型的数组，保存了当前文件对应的所有数据块信息。`INodeFile` 定义的 `header` 字段和 `blocks` 字段的代码如下：

```
private long header = 0L; // 文件头信息
private BlockInfo[] blocks; // 文件数据块信息
```

`INodeFile` 有一个内部类 `HeaderFormat`，用于处理 `INodeFile.header` 字段。它的实现很简单，就是从 `header` 这个 `long` 类型字段中提取出前 4 个比特的存储策略信息、中间 12 个比特的文件备份系数信息，以及后 48 个比特的数据块大小信息。

blocks 字段是一个 BlockInfo 类型的数组, BlockInfo 类继承自 Block 类, 它保存了数据块与文件、数据块与数据节点的对应关系。从 BlockInfo 对象可以获得数据块所属的文件, 即文件的 INodeFile 对象, 也可以获得保存数据块副本的所有数据节点的信息, 在数据块管理小节中还会详细地介绍 BlockInfo 类的实现。

INodeFile 中的方法可以分为以下几个部分。

- 构建 (Under Construction) 特性相关方法: 当 HDFS 客户端写文件时, 该文件就处于构建状态。在早先的 HDFS 版本中, 处于构建状态中的文件是用 INodeFileUnderConstruction 类表示的; 在 HDFS 2.6 版本中, 则通过在 INodeFile 中添加 FileUnderConstructionFeature 特性来表示文件处于构建状态。这类方法包括 getFileUnderConstructionFeature()、isUnderConstruction()、toUnderConstruction()、toCompleteFile()、removeLastBlock()、setLastBlock() 等。这部分代码的实现, 将在 FileUnderConstructionFeature 实现小节中进行重点介绍。
- 快照 (Snapshot) 特性相关方法: 对处于快照中的文件进行修改时, HDFS 会首先向这个文件添加 FileWithSnapshotFeature 特性, 表明这个文件在快照中。这里的方法包括 addSnapshotFeature()、getFileWithSnapshotFeature()、recordModification()、getDiffs() 等。这部分代码的实现, 将在 SnapshotFeature 实现小节中介绍。
- 其他方法: 主要包括获取和修改 header 字段信息的方法, 以及对 blocks 数组字段进行操作的方法。这部分代码的实现比较简单, 我们就不再赘述了。

5. INodeReference

当 HDFS 文件/目录处于某个快照中, 并且这个文件/目录被重命名或者移动到其他路径时, 该文件/目录就会存在多条访问路径。INodeReference 及其子类就是为了解决这个问题而产生的。

这里我们举一个例子, 如图 3-5 所示, /abc 是 HDFS 文件系统中的普通目录, 管理员为 /abc 目录建立了一个快照 s0, /abc 目录下有一个文件 foo。根据快照功能的定义, 用户可以通过路径 /abc/foo 以及 /abc/snapshot/s0/foo 访问 foo 文件。

当用户将 /abc/foo 文件重命名为 /xyz/bar 时, 通过快照路径 /abc/snapshot/s0/foo 将无法访问 foo 文件, 如图 3-6 所示, 这种情况是不符合快照规范的。

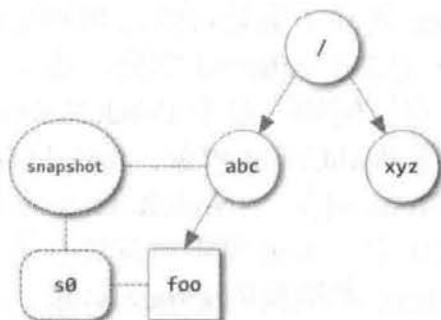


图 3-5 带有快照的文件

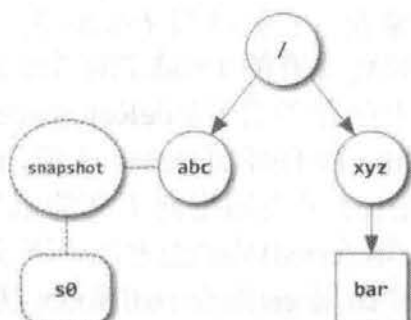


图 3-6 重命名后的文件

为了解决这个问题，HDFS 引入了 `INodeReference` 类。图 3-7 给出了 `INodeReference` 及其子类的继承关系图，这里的 `WithName`、`WithCount`、`DstReference` 都是 `INodeReference` 的子类，同时也是 `INodeReference` 的内部类。`WithName` 对象用于替代重命名操作前源路径中的 `INode` 对象，`DstReference` 对象则用于替代重命名操作后目标路径中的 `INode` 对象。`WithName` 和 `DstReference` 共同指向了一个 `WithCount` 对象，`WithCount` 对象则指向了文件系统目录树中真正的 `INode` 对象。

图 3-8 给出了使用 `INodeReference` 后的文件目录树，当进行重命名操作时，`Namenode` 会在 `/abc` 目录下添加 `INodeReference.WithName` 节点替代重命名前的 `foo` 节点，在 `/xyz` 目录下添加 `INodeReference.DstReference` 节点替代重命名后的 `bar` 节点。`INodeReference.WithName` 以及 `INodeReference.DstReference` 则共同指向一个 `INodeReference.withCount` 节点，`INodeReference.withCount` 节点指向真实的 `INode` 节点 `bar`。这样，无论用户是通过 `/xyz/bar` 路径还是 `/abc/snapshot/s0/foo` 快照路径访问文件，都可以通过获取到 `withCountReference` 对象的引用找到真正的 `INode` 节点 `bar`，也就解决了这个问题。

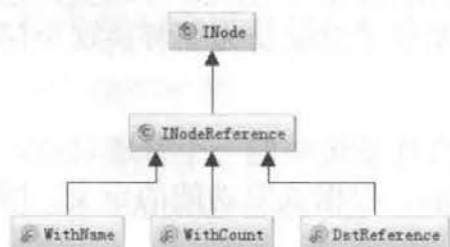


图 3-7 INodeReference 继承关系图

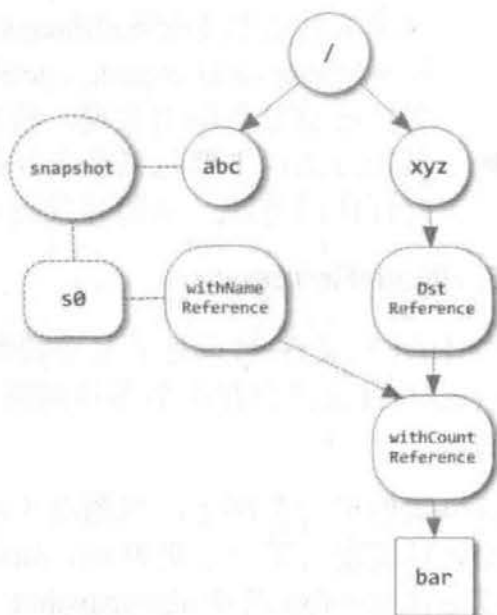


图 3-8 使用 INodeReference 后的文件目录树

了解了 `INodeReference` 的概念后，我们再来看 `INodeReference` 的代码实现。`INodeReference` 是一个抽象类，它扩展自 `INode` 类，所以 `INodeReference` 及其子类是可以添加到文件系统目录树中以替代原有的 `INodeFile` 节点的。`INodeReference` 定义了 `referred` 字段，这个字段非常重要，用于保存当前 `INodeReference` 类指向的 `INode` 节点。例如，对于 `INodeReference` 的子类 `WithName` 和 `DstReference` 来说，`referred` 字段就指向了 `WithCount` 对象；而对于 `WithCount` 来说，`referred` 字段则指向了文件系统目录树中真正的 `INode` 对象。`INodeReference` 抽象类还定义了 `getReferredINode()` 方法，在文件系统目录树的操作中，如果判断当前节点是一个引用节点，则会调用 `getReferredINode()` 方法获取 `INodeReference` 类指向的 `INode` 对象。

```

public abstract class INodeReference extends INode {
    private INode referred; // 指向的 INode 节点
}
  
```

```

public INodeReference(INode parent, INode referred) {
    super(parent);
    this.referred = referred;
}
public final INode getReferredINode() { // 获取指向的 INode 节点
    return referred;
}
public final void setReferredINode(INode referred) {
    this.referred = referred;
}
//...
}

```

了解了 INodeReference 抽象类的代码实现后,我们再来看 WithCount 类的实现。WithCount 类定义了一个集合字段 withNameList 用于保存所有指向这个 WithCount 对象的 WithName 对象的集合。WithCount 类还定义了 addReference()方法,任何指向 WithCount 对象的 WithName 对象以及 DstReference 对象都需要调用这个方法添加指向关系。对于指向这个 WithCount 对象的 DstReference 对象,addReference()方法会将这个对象设置为自己的父 INode 节点(通过 INode.parent 字段);而对于 WithName 对象,addReference()方法则将这个对象放入 withNameList 集合中保存。

```

public static class WithCount extends INodeReference {
    // 保存所有指向这个 WithCount 对象的 WithName 对象的集合
    private final List<WithName> withNameList = new ArrayList<WithName>();

    public WithCount(INodeReference parent, INode referred) {
        super(parent, referred); // 调用父类的构造方法,指向文件系统目录树中的 INode
        Preconditions.checkArgument(!referred.isReference());
        referred.setParentReference(this); // 设置真实 INode 的父节点为当前 WithCount 对象
    }

    public void addReference(INodeReference ref) {
        if (ref instanceof WithName) { // 如果是 WithName 对象,则加入 withNameList
            WithName refWithName = (WithName) ref;
            int i = Collections.binarySearch(withNameList, refWithName,
                WITHNAME_COMPARATOR);
            Preconditions.checkState(i < 0);
            withNameList.add(-i - 1, refWithName);
        } else if (ref instanceof DstReference) { // 如果是 DstReference 对象,则设置为父节点
            setParentReference(ref);
        }
    }
    // ...
}

```

了解了 WithCount 对象的实现后,我们再来看 WithName 和 DstReference,这两个类的实现都比较简单。WithName 类定义了 name 字段用于保存重命名前文件的名称,同时定义了 lastSnapshotId 字段用于保存 WithName 对象构造时源路径的快照版本号。DstReference 类的实

现就更简单了，只定义了一个 `dstSnapshotId` 字段用于保存重命名操作前目标路径的最新快照的版本号。`WithName` 和 `DstReference` 在构造时都会调用父类的构造方法指向 `WithCount` 对象，同时还会调用 `WithCount.addReference()` 方法配置 `WithCount` 对象。

```
public static class WithName extends INodeReference {
    private final byte[] name;    // 重命名前的文件名
    private final int lastSnapshotId;
    public WithName(INodeDirectory parent, WithCount referred, byte[] name,
        int lastSnapshotId) {
        super(parent, referred); // 调用父类构造方法，指向 WithCount 节点
        this.name = name;
        this.lastSnapshotId = lastSnapshotId;
        referred.addReference(this); // 调用 WithCount.addReference()
    }
    // ...
}

public static class DstReference extends INodeReference {
    private final int dstSnapshotId;
    public DstReference(INodeDirectory parent, WithCount referred,
        final int dstSnapshotId) {
        super(parent, referred); // 调用父类构造方法，指向 WithCount 节点
        this.dstSnapshotId = dstSnapshotId;
        referred.addReference(this); // 调用 WithCount.addReference()
    }
    // ...
}
```

了解了 `INodeReference` 的代码实现后，我们再来看 `Namenode` 重命名操作的代码实现。建立 `INodeReference` 节点操作的入口是 `FsDirectory.RenameOperation`，只有在进行 `rename` 操作时，也就是将一个快照中的 `INode` 节点重命名为另一个路径下的 `INode` 节点时才有可能创建 `INodeReference` 节点。这里我们看一下 `RenameOperation` 的具体操作。首先看一下 `RenameOperation` 的构造方法，构造方法首先判断 `Rename` 操作的源节点是否在快照中，如果在快照中，则调用 `INodeDirectory.replaceChild4ReferenceWithName()` 方法构造 `INodeReference` `WithName` 对象，并将 `INodeDirectory` 中的 `srcINode` 对象全部替换为该对象（需要特别注意的是，如果 `srcINode` 存在于快照 `diff` 对象的 `c-list` 列表中，也是需要替换的）。

```
private RenameOperation(String src, String dst, INodesInPath srcIIP, INodesInPath
dstIIP)
    throws QuotaExceededException {
    this.srcIIP = srcIIP;
    this.dstIIP = dstIIP;
    this.src = src;
    this.dst = dst;
    srcChild = srcIIP.getLastINode();
    srcChildName = srcChild.getLocalNameBytes();
    isSrcInSnapshot = srcChild.isInLatestSnapshot(
```

```

        srcIIP.getLatestSnapshotId());
srcChildIsReference = srcChild.isReference();
srcParent = srcIIP.getInode(-2).asDirectory();

if (isSrcInSnapshot) {
    srcChild.recordModification(srcIIP.getLatestSnapshotId());
}

srcRefDstSnapshot = srcChildIsReference ? srcChild.asReference()
    .getDstSnapshotId() : Snapshot.CURRENT_STATE_ID;
oldSrcCounts = Quota.Counts.newInstance();
// 如果源节点在快照中
if (isSrcInSnapshot) {
    // 调用 replaceChild4ReferenceWithName() 方法构造 WithName 和 WithCount 对象
    final INodeReference.WithName withName = srcIIP.getInode(-2).asDirectory()
        .replaceChild4ReferenceWithName(srcChild, srcIIP.getLatestSnapshotId());
    withCount = (INodeReference.WithCount) withName.getReferredINode();
    srcChild = withName;
    srcIIP.setLastINode(srcChild);
    withCount.getReferredINode().computeQuotaUsage(oldSrcCounts, true);
} else if (srcChildIsReference) {
    withCount = (WithCount) srcChild.asReference().getReferredINode();
} else {
    withCount = null; // 否则将 withCount 设置为 null, 也就是普通的重命名操作
}
}

```

完成了对 withCount 的构造之后, 就可以调用 addSourceToDestination() 方法将源节点或者 dst 节点添加到目标路径了。这里分为两种情况: withCount==null, 也就是普通的重命名操作, 则不需要使用 INodeReference 机制, 在这种情况下直接将源 INode 节点添加到目标路径即可; withCount!=null, 也就是需要使用 INodeReference 机制的情况, 这里则构造 DstReference 对象, 然后将这个 DstReference 对象添加到目标路径即可。这里注意, DstReference 对象的构造方法会将 DstReference.referred 字段设置为 WithCount 对象, 然后设置 WithCount 对象的父节点为当前 DstReference 对象。

```

boolean addSourceToDestination() {
    final INode dstParent = dstIIP.getInode(-2);
    srcChild = srcIIP.getLastINode();
    final byte[] dstChildName = dstIIP.getLastLocalName();
    final INode toDst;
    if (withCount == null) {
        srcChild.setLocalName(dstChildName); // 普通情况, 则直接添加源节点
        toDst = srcChild;
    } else { // 要使用 INodeReference 机制的情况, 则构造 DstReference 对象
        withCount.getReferredINode().setLocalName(dstChildName);
        int dstSnapshotId = dstIIP.getLatestSnapshotId();
        toDst = new INodeReference.DstReference(
            dstParent.asDirectory(), withCount, dstSnapshotId);
    }
}

```

```

    }
    return addLastINodeNoQuotaCheck(dstIIP, toDst); // 将 toDst 节点添加到目标路径
}

```

3.1.2 Feature 相关类

在 HDFS 2.6 版本中引入了特性 (Feature) 这个概念, HDFS 定义了 `INode.Feature` 接口抽象 `INode` 特性的根接口, 如图 3-9 所示, `INode` 的所有特性都实现了这个接口, 每个特性都对应一个 `Feature` 子类, 包括:

- `DirectoryWithSnapshotFeature`: 带有快照的目录特性。
- `DirectorySnapshottableFeature`: 可以添加快照的目录特性。
- `FileWithSnapshotFeature`: 带有快照的文件特性。
- `DirectoryWithQuotaFeature`: 支持磁盘配额的目录特性。
- `FileUnderConstructionFeature`: 正在构建的文件特性。
- `XAttrFeature`: 支持文件系统扩展属性的特性。
- `AclFeature`: 安全特性, 我们暂不介绍。

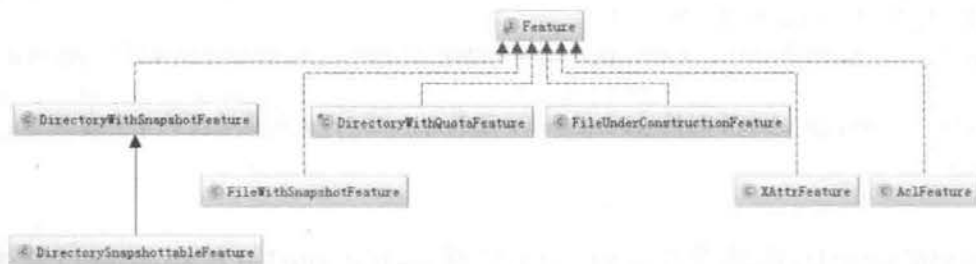


图 3-9 Feature 接口继承关系图

本节就介绍快照特性对应的 `SnapshotFeature` 类, 以及构建中特性对应的 `FileUnderConstructionFeature` 类。

1. SnapshotFeature 实现

HDFS 2.6 版本发布了快照功能, HDFS 管理员可以对 HDFS 中的任意目录建立快照 (Snapshot)。快照是一个文件系统, 或者是文件系统中某个目录在某一时刻的镜像。在目标目录的快照创建之后, 不论目标目录发生任何变化, 或者目标目录中保存的子目录项发生任何变化, 都可以通过快照找回快照建立时目标目录的所有文件以及目录结构。本节我们就学习 HDFS 快照功能的实现。

在为目标目录创建快照之前, 需要先执行 “`hdfs dfsadmin -allowSnapshot`” 命令启用目标目录的快照功能。在目标目录启用快照功能后, 该目录并不会自动执行快照保存操作, 管理员还需要执行 “`hdfs dfs -createSnapshot`” 命令创建快照。需要注意的是, 同一个目录可以创建多个快照, 不同的快照通过名字来区分。

我们首先以一个例子来分析整个快照功能执行的流程。如图 3-10 所示, 左边的树是一棵

普通的文件系统目录树。当管理员执行“`hdfs dfsadmin-allowSnapshot`”命令在目录 a 上开启快照功能后，HDFS 会创建一个 `DirectorySnapshottableFeature` 对象，然后将这个新创建的 Feature 对象添加到目录 a 对应的 `INodeDirectory` 对象的 features 集合中，如右图所示。

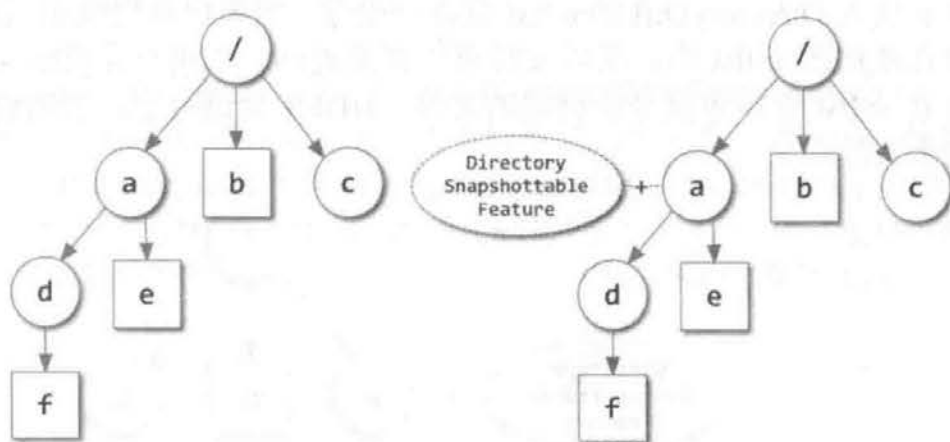


图 3-10 开启快照功能

成功开启目录 a 的快照功能后，管理员就可以执行“`hdfs dfs-createSnapshot`”命令在目录 a 下创建一个快照 s1。如图 3-11 所示，HDFS 会创建一个 `DirectoryDiff` 对象记录 s1 快照创建之后目录 a 上执行的所有操作，并将这个新创建的 `DirectoryDiff` 对象添加到 `DirectorySnapshottableFeature` 的 `DirectoryDiffList` 对象中保存。`DirectoryDiff` 会通过持有一个 `ChildrenDiff` 对象记录目录 a 的子目录项的变化情况，`ChildrenDiff` 对象的 c-list 集合保存了快照创建之后目录 a 下所有新添加的文件或者目录，`ChildrenDiff` 对象的 d-list 集合则保存了快照创建之后从目录 a 删除的文件或者目录。

如图 3-12 所示，成功创建快照 s1 后，当我们从目录 a 删除文件 e 时，文件 e 并不会直接从 `INodeDirectory` 中完全删除，而是暂时保存在快照 s1 对应的 `DirectoryDiff` 对象的 d-list 集合中。

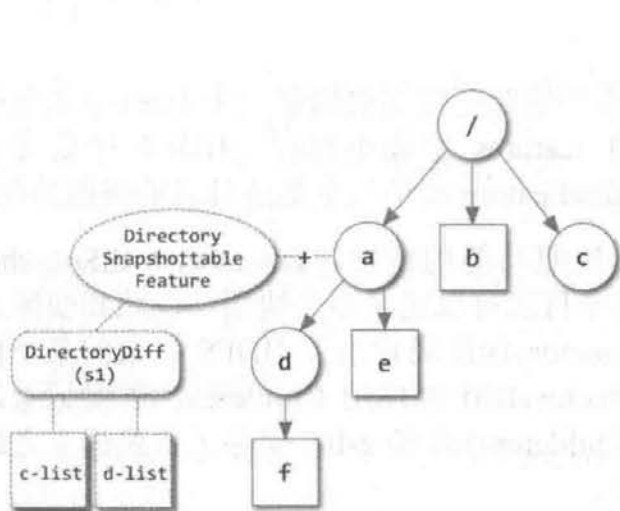


图 3-11 建立快照 s1

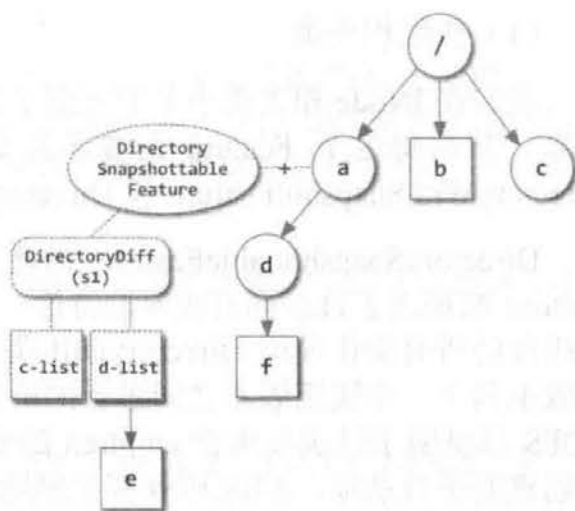


图 3-12 删除文件 e

删除文件 e 之后，我们在目录 a 下再创建一个快照 s2，然后向目录 a 添加一个文件 g。如

图 3-13 所示，HDFS 会在 `DirectorySnapshottableFeature` 对象上添加一个新的 `DirectoryDiff` 对象记录快照 s2 创建之后在目录 a 上进行的所有操作，这样快照 s1 对应的 `DirectoryDiff` 对象就记录了快照 s1 和快照 s2 之间目录 a 上执行的操作。成功创建 `DirectoryDiff` 对象之后，HDFS 将新添加的文件 g 放入 `DirectoryDiff` 的 c-list 集合中保存。当用户在快照 s1 上检索文件 e 时，由于文件 e 保存在快照的 d-list 中，所以文件可以正常返回。当用户在快照 s2 上检索文件 g 时，由于文件 g 在 c-list 中，也就是新创建的文件，HDFS 会返回空，表明快照 s2 创建时 a 目录下并没有这个文件。

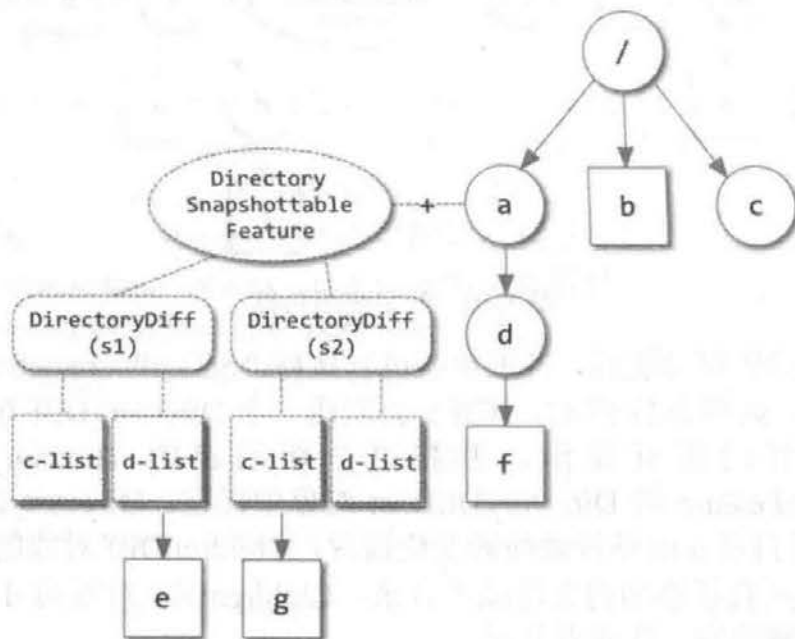


图 3-13 建立快照 s2

至此，一个完整的快照流程就介绍完了。在下面的两个小节中我们介绍实现快照功能的类，以及快照流程的代码实现。

(1) 快照相关类

我们在 `Inode` 相关类小节中介绍了，为 `Inode` 添加快照特性需要创建一个 `Feature` 类型的对象，然后将这个 `Feature` 对象添加到 `Inode` 的 `features` 字段中保存。HDFS 中定义了 `DirectoryWithSnapshotFeature` 和 `DirectorySnapshottableFeature` 两个类来描述目录的快照特性。

`DirectorySnapshottableFeature` 描述了目录开启快照功能的特性，`DirectoryWithSnapshotFeature` 则描述了目录拥有快照的特性。HDFS 的单个目录可以定义多个快照，快照和快照之间进行的所有操作使用 `DirectoryDiff` 类来记录。`DirectoryDiff` 类记录了 HDFS 目录的某个快照版本和上一个快照版本之间进行的所有操作，`DirectoryDiff` 类使用 `ChildrenDiff` 类记录该 HDFS 目录的子目录项集合 `children` 的变化情况，`ChildrenDiff` 的 c-list 保存了快照建立之后新创建的子目录项，d-list 则保存了删除的子目录项。

由于一个 HDFS 目录可以拥有多个快照，单个快照和上一个快照之间进行的操作是通过一个 `DirectoryDiff` 对象来记录的，所以 HDFS 目录需要持有多个 `DirectoryDiff` 对象来记录所

有快照之间进行的操作。HDFS 定义了 `DirectoryDiffList` 类来保存这些 `DirectoryDiff` 对象, `DirectoryDiffList` 底层持有一个 `DirectoryDiff` 的集合, 集合中的所有 `DirectoryDiff` 元素是按照快照 ID 进行排序的, 并且每个 `DirectoryDiff` 对象都持有下一个 `DirectoryDiff` 对象的引用, 类似于链表的功能。

快照功能涉及的所有类如图 3-14 所示, `DirectoryWithSnapshotFeature` 类通过持有一个 `DirectoryDiffList` 对象记录了 HDFS 目录中所有快照之间进行的全部操作; 而 `DirectoryDiffList` 底层则保存了多个 `DirectoryDiff` 对象, 每个 `DirectoryDiff` 对象记录了目录的一个快照版本和上一个快照版本之间进行的操作; `DirectoryDiff` 底层则通过持有一个 `ChildrenDiff` 对象来记录目录中所有子目录项的变化操作。下面我们将一一介绍这些类的具体实现。

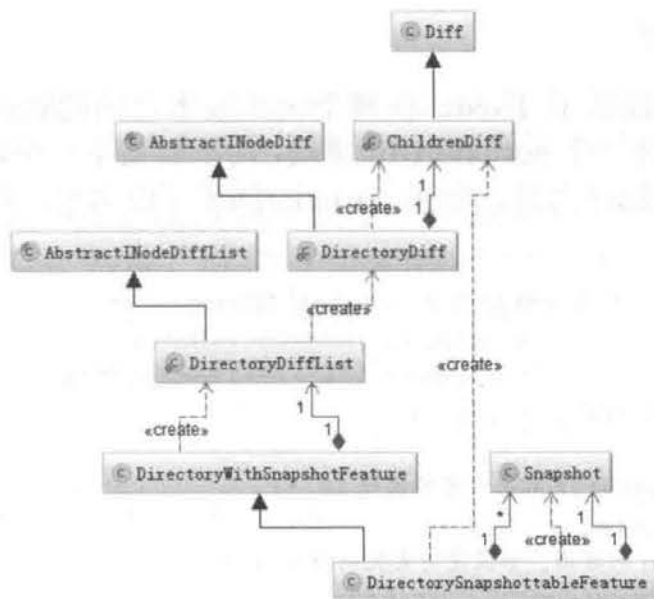


图 3-14 快照功能涉及的类

Diff 类

我们首先看一下类图中的第一个类 `Diff`。`Diff` 类用来描述一个集合的当前状态和上一个状态之间的不同。`Diff` 类中有两个列表: `c-list` 记录了集合在两个状态之间新创建的元素; `d-list` 记录了集合在两个状态之间删除的元素。使用这两个列表, `Diff` 类可以记录集合上一个状态与当前状态之间所有新创建、删除与修改的元素。

`Diff` 类的算法很简单: 对于新创建的元素, 放入 `c-list` 中; 对于新删除的元素, 放入 `d-list` 中; 对于修改的元素, 则同时放入 `c-list` 和 `d-list` 中。

ChildrenDiff 类

`ChildrenDiff` 类用于描述 `InodeDirectory` 的子目录项集合 `children` 字段的当前状态与上一个快照版本状态之间的差异。

`ChildrenDiff` 扩展自 `Diff` 类, 从类定义代码我们可以看出, `ChildrenDiff` 描述的是 `Inode` 集合状态间的差异, 而 `InodeDirectory` 的 `children` 字段正好是一个 `List<Inode>` 类型, 也就是

目录中所有子目录项（包括文件和子目录）的 `INode` 集合。由此我们可以知道，`ChildrenDiff` 的 `c-list`（创建列表）和 `d-list`（删除列表）存放的就是 `INodeDirectory` 在当前状态和上一个快照状态之间新创建的子目录项以及删除的子目录项。`ChildrenDiff` 类的定义如下：

```
static class ChildrenDiff extends Diff<byte[], INode> {
    ChildrenDiff() {}

    private ChildrenDiff(final List<INode> created, final List<INode> deleted) {
        super(created, deleted);
    }
    ...
}
```

AbstractINodeDiff 类

`AbstractINodeDiff` 类抽象了 `INode` 在两个快照版本之间的状态差异。如下代码所示，`AbstractINodeDiff` 类中包含一个 `posteriorDiff` 字段，用于链接下一个快照版本与当前快照版本之间差异的 `AbstractINodeDiff` 对象，如果 `posteriorDiff` 字段为空，则表明 `INode` 的当前状态为最新状态。

```
abstract class AbstractINodeDiff<N extends INode,
                                A extends INodeAttributes,
                                D extends AbstractINodeDiff<N, A, D>>
    implements Comparable<Integer> {

    /** 当前 AbstractINodeDiff 对象对应的快照 ID */
    private int snapshotId;
    /** 快照所对应的 INode 的数据，如果没有变化则为 null */
    A snapshotINode;
    /** 下一个快照版本与当前快照版本之间差异的 AbstractINodeDiff */
    private D posteriorDiff;

    // .....
}
```

DirectoryDiff 类

`DirectoryDiff` 是 `DirectoryWithSnapshotFeature` 的内部类，它扩展自 `AbstractINodeDiff` 类。`AbstractINodeDiff` 类用于抽象 `INode` 在两个快照版本之间的状态差异，由此可以知道 `DirectoryDiff` 就是用于描述 `INodeDirectory` 在两个快照版本之间的状态差异的。

如下代码所示，`DirectoryDiff` 类通过持有一个 `ChildrenDiff` 对象来记录 `INodeDirectory` 的当前快照与上一个快照间的子目录项集合的变化情况，也就是 `children` 字段的变化情况。

```
public static class DirectoryDiff extends
    AbstractINodeDiff<INodeDirectory, INodeDirectoryAttributes, DirectoryDiff> {
    /** 快照创建时，INodeDirectory 子目录项的数目 */
    private final int childrenSize;
    /** 子目录项 diff */
    ...
}
```

```

private final ChildrenDiff diff;
private boolean isSnapshotRoot = false;

private DirectoryDiff(int snapshotId, INodeDirectory dir) {
    super(snapshotId, null, null);
    // 获取快照建立时, 子目录项的个数
    this.childrenSize = dir.getChildrenList(Snapshot.CURRENT_STATE_ID).size();
    // 构造一个新的 ChildrenDiff 对象, 记录 INodeDirectory.children 字段的变化
    this.diff = new ChildrenDiff();
}
// ...
}

```

AbstractINodeDiffList 类

AbstractINodeDiffList 维护了一个 AbstractINodeDiff 的集合列表, 用于保存一个 INode 拥有的所有 AbstractINodeDiff 对象, 也就是记录了 INode 所有快照间差异的列表。

如下代码所示, AbstractINodeDiffList 定义了一个 diffs 集合字段, diffs 保存了 INode 所有快照间的 AbstractINodeDiff 对象, 所有 AbstractINodeDiff 对象是按照快照 ID 排序的。AbstractINodeDiffList 的方法主要用于维护 diffs 这个数据结构。

```

abstract class AbstractINodeDiffList<N extends INode,
                                   A extends INodeAttributes,
                                   D extends AbstractINodeDiff<N, A, D>>
    implements Iterable<D> {
    /** 快照的 Diff 列表, 这个列表按照快照 ID 排序 */
    private final List<D> diffs = new ArrayList<D>();
}

```

DirectoryDiffList 类

DirectoryDiffList 扩展自 AbstractINodeDiffList, 维护了 INodeDirectory 所有快照间差异的集合, 也就是 DirectoryDiff 对象的集合。DirectoryDiffList 对象的定义代码如下:

```

public static class DirectoryDiffList
    extends AbstractINodeDiffList<INodeDirectory, INodeDirectoryAttributes, DirectoryDiff>

```

DirectoryWithSnapshotFeature 类

DirectoryWithSnapshotFeature 类用于存储和处理指定目录下建立的所有快照, 也就是记录目录快照之间进行的添加、删除、修改目录项的操作。需要注意的是, DirectoryDiff、DirectoryDiffList 以及 ChildrenDiff 类都是 DirectoryWithSnapshotFeature 类的内部类。

DirectoryWithSnapshotFeature 类持有一个 DirectoryDiffList 类型的字段 diffs, 用于保存当前目录以及它的子目录项在不同快照间的所有差异。DirectoryWithSnapshotFeature 类的构造方法也非常简单, 就是判断传入的 DirectoryDiffList 是否为 null, 不为 null 则构造一个新的 DirectoryDiffList。

```

private final DirectoryDiffList diffs;

```

```

public DirectoryWithSnapshotFeature(DirectoryDiffList diffs) {
    this.diffs = diffs != null ? diffs : new DirectoryDiffList();
}

```

在老版本的 HDFS 代码中，向 `INodeDirectory` 中添加、删除以及修改子目录项的操作只需要更改 `INodeDirectory.children` 字段即可。添加了快照机制之后，对于子目录项的修改都需要由 `DirectoryWithSnapshotFeature` 代理。考虑下面这种情况，HDFS 管理员为目录 `/tmp/dir` 建立快照 `s0`，则 `/tmp/dir` 对应的 `INodeDirectory` 会首先被添加 `DirectoryWithSnapshotFeature` 特性，并在 `DirectoryWithSnapshotFeature.directoryDiffList` 字段中添加一个当前快照 `s0` 对应的 `DirectoryDiff` 对象。这个对象用于保存 `s0` 建立之后对 `/tmp/dir` 目录的添加、删除、修改子目录项等操作。所以 `INodeDirectory` 中所有对于 `children` 字段的修改方法都会由 `DirectoryWithSnapshotFeature` 中对应的方法代理，比如 `addChild()`、`removeChild()` 方法等。

通过前面章节的介绍我们知道，`DirectoryDiff` 对象会将当前目录中新创建的子目录项放入 `DirectoryDiff.childrenDiff` 字段的 `c-list` 中，而将删除的子目录项放入 `DirectoryDiff.childrenDiff` 字段的 `d-list` 中。还是继续刚才的例子，当管理员成功为 `/tmp/dir` 建立了一个名为 `s0` 的快照后，用户向 `/tmp/dir` 目录中新写入一个 `syslog1` 文件时，这个文件会被添加到 `s0` 快照对应的 `DirectoryDiff` 对象的 `c-list` 中。当用户通过 `s0` 快照路径查找 `syslog1` 文件时，由于 `syslog1` 文件在 `DirectoryDiff` 的 `c-list` 中，也就是该文件是快照建立之后新创建的文件，HDFS 会返回未找到该文件。当用户正常查找 `syslog1` 文件而不通过快照路径查找时，HDFS 将会直接返回 `syslog1` 文件。删除操作与添加操作同理，这里就不再详细解释了。

了解清楚了流程之后，再看 `DirectoryWithSnapshotFeature` 的 `addChild()` 以及 `removeChild()` 方法就比较简单了。这里以 `addChild()` 方法为例，`addChild()` 方法首先从 `DirectoryDiffList` 中获取当前快照对应的 `DirectoryDiff` 对象，然后获取这个 `DirectoryDiff` 对象中保存的 `ChildrenDiff` 对象。之后调用 `ChildrenDiff.create()` 方法将这个新添加的 `INode` 对象放入 `ChildrenDiff` 的 `c-list` 中。成功在 `DirectoryDiff` 对象中记录修改操作后，`DirectoryWithSnapshotFeature` 的代理工作就完成了，这时候 `addChild()` 方法会调用 `INodeDirectory.add()` 方法，将新创建的 `INode` 对象放入 `INodeDirectory.children` 字段中。

```

public boolean addChild(INodeDirectory parent, INode inode,
    boolean setModTime, int latestSnapshotId) throws QuotaExceededException {
    // 根据 latestSnapshotId 获取 DirectoryDiff 的 ChildrenDiff 字段
    ChildrenDiff diff = diffs.checkAndAddLatestSnapshotDiff(latestSnapshotId,
        parent).diff;
    // 在 ChildrenDiff 的 c-list 中添加新创建的 INode 对象
    int undoInfo = diff.create(inode);

    // 调用 INodeDirectory.add() 方法将新创建的 INode 对象加入 INodeDirectory.children 字段
    final boolean added = parent.addChild(inode, setModTime,
        Snapshot.CURRENT_STATE_ID);
    if (!added) {
        diff.undoCreate(inode, undoInfo);
    }
    return added;
}

```

除了添加和删除子目录项外, 获取一个目录在指定快照状态下的所有子目录项也是一个很重要的操作。还是考虑刚才的例子, HDFS 目录/tmp/dir 下有三个文件 a、b、c, 这时建立快照 s0, 然后删除文件 a、b, 再建立快照 s1。当我们通过 s1 快照获取 dir 目录的子目录项时, 应当返回文件 c。通过 s0 快照获取时, 应该返回文件 a、b、c。这个逻辑是由 getChildrenList() 方法实现的, getChildrenList() 方法首先从 DirectoryDiffList 中获取指定快照 ID 对应的 DirectoryDiff 对象, 然后在该 DirectoryDiff 对象上调用 getChildrenList() 方法。

```
public ReadOnlyList<INode> getChildrenList(INodeDirectory currentINode,
    final int snapshotId) {
    // 获取快照对应的 DirectoryDiff 对象
    final DirectoryDiff diff = diffs.getDiffById(snapshotId);
    // 调用 DirectoryDiff.getChildrenList() 方法获取该快照下的所有子目录项
    return diff != null ? diff.getChildrenList(currentINode) : currentINode
        .getChildrenList(Snapshot.CURRENT_STATE_ID);
}
```

DirectoryDiff.getChildrenList() 方法会将目录的指定快照状态与当前状态间的所有 ChildrenDiff 合并成一个 ChildrenDiff 对象 combined, 然后调用 combined.apply2Current() 方法通过目录当前状态以及 combined 逆推出快照建立时目录的状态, 最后返回。

```
private ReadOnlyList<INode> getChildrenList(final INodeDirectory currentDir) {
    return new ReadOnlyList<INode>() {
        // ...
        private List<INode> initChildren() {
            if (children == null) {
                // 将指定快照状态与当前目录状态间的所有操作合并成一个 ChildrenDiff 对象
                final ChildrenDiff combined = new ChildrenDiff();
                for (DirectoryDiff d = DirectoryDiff.this; d != null;
                    d = d.getPosterior()) {
                    combined.combinePosterior(d.diff, null);
                }
                // 在 combined 对象上调用 apply2Current() 方法逆推出快照建立时的目录状态
                children = combined.apply2Current(ReadOnlyList.Util.asList(
                    currentDir.getChildrenList(Snapshot.CURRENT_STATE_ID)));
            }
            // 返回快照建立时目录中的所有子目录项
            return children;
        }
        // ...
    };
}
```

了解了 DirectoryWithSnapshotFeature.getChildrenList() 方法的实现后, 我们再学习 DirectoryWithSnapshotFeature.getChild() 方法, 这个方法用于获取当前目录在指定快照状态下的一个 INode 对象。getChild() 方法首先获取 snapshotId 对应的 DirectoryDiff 对象, 然后在 DirectoryDiff 对象上调用 getChild() 方法获取这个快照下的指定 INode 对象。

```

public INode getChild(INodeDirectory currentINode, byte[] name,
    int snapshotId) {
    final DirectoryDiff diff = diffs.getDiffById(snapshotId);
    return diff != null ? diff.getChild(name, true, currentINode)
        : currentINode.getChild(name, Snapshot.CURRENT_STATE_ID);
}

```

DirectoryDiff.getChild()方法会从目录的快照版本向后遍历，依次获得快照间保存差异操作的 DirectoryDiff 对象（DirectoryDiff 的 posteriorDiff 字段保存了下一个快照版本的 DirectoryDiff 对象的引用），然后在 DirectoryDiff 对象的 diff 字段（ChildreDiff）上调用 accessPrevious()方法判断差异操作中是否保存了要查找的 INode 对象，如果保存了则返回这个 INode 对象。如果在快照版本与当前版本之间的所有 DirectoryDiff 对象中都没有这个 INode 的信息，则调用 INodeDirectory.getChild()方法从当前状态的目录中获取这个 INode 对象并返回。

```

INode getChild(byte[] name, boolean checkPosterior,
    INodeDirectory currentDir) {
    for(DirectoryDiff d = this; ; d = d.getPosterior()) {
        // 调用 ChildrenDiff.accessPrevious()方法获取 INode 信息
        final Container<INode> returned = d.diff.accessPrevious(name);
        if (returned != null) {
            // 如果在两个版本间的 DirectoryDiff 中记录了这个 INode 对象，则返回
            return returned.getElement();
        } else if (!checkPosterior) {
            return null;
        } else if (d.getPosterior() == null) {
            // 所有快照版本间的 DirectoryDiff 对象中都没有记录，则在当前目录上查找
            return currentDir.getChild(name, Snapshot.CURRENT_STATE_ID);
        }
    }
}

```

DirectorySnapshottableFeature 类

DirectorySnapshottableFeature 类用于描述开启快照功能的目录，这个类是 DirectoryWithSnapshotFeature 的子类，相对于父类它添加了一些描述在这个目录下创建的所有快照的字段。

如图 3-15 所示，DirectorySnapshottableFeature 中最重要的字段就是 snapshotsByNames，它是一个 Snapshot 类的集合，用于保存当前目录下创建的所有快照。Snapshot 类用于描述一个新创建的快照，包括了快照 ID、快照根节点、快照名等信息。

DirectorySnapshottableFeature 定义的方法也多是处理 snapshotsByNames 这个字段的，方法的作用通过方法名就可以看出来，实现也比较简单。这里我们重点介绍一下 computeDiff()方法，其他方法比较简单，就不再详细介绍了。

DirectorySnapshottableFeature	
SNAPSHOT_LIMIT	int
snapshotsByNames	List<Snapshot>
snapshotQuota	int
getHunSnapshots()	int
searchSnapshot(byte[])	int
getSnapshot(byte[])	Snapshot
getSnapshotById(int)	Snapshot
getSnapshotList()	ReadOnlyList<Snapshot>
renameSnapshot(String, String, String)	void
getSnapshotQuota()	int
setSnapshotQuota(int)	void
addSnapshot(Snapshot)	void
addSnapshot(INodeDirectory, int, String)	Snapshot
removeSnapshot(INodeDirectory, String, BlocksMapUpdateInfo, List<INode>) upshot	
computeContentSummary(INodeDirectory, ContentSummaryComputationContext) context	
computeDiff(INodeDirectory, String, String)	SnapshotDiffInfo
getSnapshotByName(INodeDirectory, String)	Snapshot
computeDiffRecursively(INodeDirectory, INode, List<byte[]>, SnapshotDiffInfo) i	
findRenameTargetPath(INodeDirectory, WithName, int)	byte[]
toString()	String
dumpTreeRecursively(INodeDirectory, PrintWriter, StringBuilder, int)	void

图 3-15 DirectorySnapshottableFeature 类结构

computeDiff()方法用于计算指定目录的两个快照版本之间的差异，它会返回一个 SnapshotDiffInfo 类记录哪些子目录项发生了变化、哪些子目录项被重命名、发生变化子目录项的 ChildrenDiff 对象的引用等信息。computeDiff()方法首先在 snapshotsByNames 集合中查找两个快照的 Snapshot 对象的引用，然后调用 computeDiffRecursively()方法递归计算两个快照之间的差异。

```
SnapshotDiffInfo computeDiff(final INodeDirectory snapshotRoot,
    final String from, final String to) throws SnapshotException {
    // 从 snapshotsByNames 集合中获取两个快照的引用
    Snapshot fromSnapshot = getSnapshotByName(snapshotRoot, from);
    Snapshot toSnapshot = getSnapshotByName(snapshotRoot, to);

    if (from.equals(to)) {
        return null;
    }
    // 构造 SnapshotDiffInfo 对象，用于保存两个快照之间的差异
    SnapshotDiffInfo diffs = new SnapshotDiffInfo(snapshotRoot, fromSnapshot,
        toSnapshot);
    //调用 computeDiffRecursively(), 将结果放入 SnapshotDiffInfo 对象中
    computeDiffRecursively(snapshotRoot, snapshotRoot, new ArrayList<byte[]>(),
        diffs);
    //返回 diffs
    return diffs;
}
```

computeDiffRecursively()方法首先判断当前目录或者文件是否发生了改变，如果发生了改变，则记录到 SnapshotDiffInfo 中。判断完当前目录后，递归调用判断当前目录的子目录项的改变情况，这里需要注意，对于在新版本的快照中已经删除的目录，不需要递归判断该目录的子目录项。那么对于目录来说，如何计算两个快照版本之间发生的变化呢？这里调用了

DirectoryWithSnapshotFeature.computeDiffBetweenSnapshots() 方法，这个方法会首先从 DirectoryWithSnapshotFeature 的 DirectoryDiffList 中取出两个 Snapshot 状态间的所有 ChildrenDiff 对象，然后调用 Diff.combinePosterior() 方法将所有的 ChildrenDiff 对象合并为一个 ChildrenDiff 对象 diff，这个 diff 对象汇总了两个快照版本之间进行的所有操作。

```
private void computeDiffRecursively(final INodeDirectory snapshotRoot,
    INode node, List<byte[]> parentPath, SnapshotDiffInfo diffReport) {
    // 获取两个快照的 Snapshot 对象引用
    final Snapshot earlierSnapshot = diffReport.isFromEarlier() ?
        diffReport.getFrom() : diffReport.getTo();
    final Snapshot laterSnapshot = diffReport.isFromEarlier() ?
        diffReport.getTo() : diffReport.getFrom();
    byte[][] relativePath = parentPath.toArray(new byte[parentPath.size()][]); // 当前路径
    if (node.isDirectory()) { // 如果当前节点是目录
        final ChildrenDiff diff = new ChildrenDiff(); // diff 用来保存两个快照间的操作
        INodeDirectory dir = node.asDirectory();
        DirectoryWithSnapshotFeature sf = dir.getDirectoryWithSnapshotFeature();
        if (sf != null) {
            // 调用 computeDiffBetweenSnapshots() 方法合并两个快照间的 diff
            boolean change = sf.computeDiffBetweenSnapshots(earlierSnapshot,
                laterSnapshot, diff, dir);
            if (change) {
                // 如果当前目录在两个快照版本间发生了变化，则写入 diffReport
                diffReport.addDirDiff(dir, relativePath, diff);
            }
        }
        // 调用 getChildrenList() 获取 earlierSnapshot 快照建立时的当前目录的所有子目录项
        ReadOnlyList<INode> children = dir.getChildrenList(earlierSnapshot
            .getId());
        for (INode child : children) { // 遍历所有的子目录项
            final byte[] name = child.getLocalNameBytes();
            boolean toProcess = diff.searchIndex(ListType.DELETED, name) < 0;
            // 如果该子目录项在 laterSnapshot 中已经被删除，则不用递归处理
            if (!toProcess && child instanceof INodeReference.WithName) {
                // 如果发生了 rename 操作，则在 diffReport 中记录 rename 操作
                byte[][] renameTargetPath = findRenameTargetPath(
                    snapshotRoot, (WithName) child,
                    laterSnapshot == null ? Snapshot.CURRENT_STATE_ID :
                        laterSnapshot.getId());
                if (renameTargetPath != null) {
                    toProcess = true;
                    diffReport.setRenameTarget(child.getId(), renameTargetPath);
                }
            }
            // 否则递归处理该子目录项
            if (toProcess) {
                parentPath.add(name);
                computeDiffRecursively(snapshotRoot, child, parentPath, diffReport);
            }
        }
    }
}
```

```

        parentPath.remove(parentPath.size() - 1);
    }
}
} else if (node.isFile() && node.asFile().isWithSnapshot()) {
    // 对于文件, 如果发生改变, 也记录在 diffReport 中
    INodeFile file = node.asFile();
    boolean change = file.getFileWithSnapshotFeature()
        .changedBetweenSnapshots(file, earlierSnapshot, laterSnapshot);
    if (change) {
        diffReport.addFileDiff(file, relativePath);
    }
}
}
}

```

(2) 快照流程的代码实现

了解了实现快照功能的类之后, 我们再来学习快照相关流程的代码实现, 包括: 启用目录的快照功能、添加一个快照、在有快照的目录下添加和删除文件, 以及查询快照中文件等流程。

启用快照功能

用户对 HDFS 目录创建快照前, 需要首先启用目标目录的快照功能 (通过执行 “hdfs dfsadmin-allowSnapshot” 命令)。只有启用快照功能后, 用户才可以手动在目标目录下创建一个快照 (通过执行 “hdfs dfs-createSnapshot” 命令)。

启用快照功能是通过在目标目录对应的 INodeDirectory 对象上添加一个 DirectorySnapshottableFeature 特性对象实现的, 由 INodeDirectory.addSnapshottableFeature() 方法执行具体逻辑。addSnapshottableFeature() 方法会首先构造 DirectorySnapshottableFeature 对象, 然后调用 addFeature() 方法 (参考 InodeWithAdditionalFiles 类小节) 将 Feature 对象加入 INodeDirectory 的 features 字段中保存。addSnapshottableFeature() 方法的代码如下:

```

public void addSnapshottableFeature() {
    DirectoryWithSnapshotFeature s = this.getDirectoryWithSnapshotFeature();
    // 构造 DirectorySnapshottableFeature 特性
    final DirectorySnapshottableFeature snapshottable =
        new DirectorySnapshottableFeature(s);
    if (s != null) {
        this.removeFeature(s);
    }
    // 将 DirectorySnapshottableFeature 特性添加至 INodeDirectory 的特性列表中
    this.addFeature(snapshottable);
}

```

添加一个快照

开启目录的快照功能后, HDFS 管理员就可以通过执行 “hdfs dfs-createSnapshot” 命令添加一个快照了。添加快照的流程是由 INodeDirectory.addSnapshot() 方法执行的。

`INodeDirectory.addSnapshot()`方法会首先获取当前目录的 `DirectorySnapshottableFeature` 对象, 然后在这个对象上调用 `addSnapshot()`方法添加一个快照到当前目录。

```
public Snapshot addSnapshot(int id, String name) throws SnapshotException,
    QuotaExceededException {
    // 调用 DirectorySnapshottableFeature.addSnapshot() 方法添加一个快照到当前目录
    return getDirectorySnapshottableFeature().addSnapshot(this, id, name);
}
```

`DirectorySnapshottableFeature.addSnapshot()`方法会首先判断当前目录创建的快照数目是否超过了配额, 然后构建 `Snapshot` 对象放入 `DirectorySnapshottableFeature.snapshotsByNames` 集合中, 最后构建当前快照对应的 `DirectoryDiff` 对象放入 `DirectoryDiffList` 中。这里的 `Snapshot` 对象是用来描述快照信息的, 主要保存快照 ID、快照的目录以及快照名等信息。

```
public Snapshot addSnapshot(INodeDirectory snapshotRoot, int id, String name)
    throws SnapshotException, QuotaExceededException {
    // 检查目录的快照数量是否超出了配额
    final int n = getNumSnapshots();
    if (n + 1 > snapshotQuota) {
        throw new SnapshotException("...");
    }
    // 创建 Snapshot 对象
    final Snapshot s = new Snapshot(id, name, snapshotRoot);
    final byte[] nameBytes = s.getRoot().getLocalNameBytes();
    final int i = searchSnapshot(nameBytes);
    if (i >= 0) {
        throw new SnapshotException("..."); // 已经存在同名的快照, 抛出异常
    }

    // 创建当前快照对应的 DirectoryDiff 对象, 并将该对象放入 DirectoryDiffList 中
    final DirectoryDiff d = getDiffs().addDiff(id, snapshotRoot);
    d.setSnapshotRoot(s.getRoot());
    // 将 Snapshot 对象放入 snapshotsByNames 集合中
    snapshotsByNames.add(-i - 1, s);

    // 更新 modificationTime
    final long now = Time.now();
    snapshotRoot.updateModificationTime(now, Snapshot.CURRENT_STATE_ID);
    s.getRoot().setModificationTime(now, Snapshot.CURRENT_STATE_ID);
    return s;
}
```

添加和删除文件

建立快照之后, 向指定的 `INodeDirectory` 中添加和删除文件就不再是简单修改 `INodeDirectory.children` 字段了, 这里我们以 `INodeDirectory.addChild()`方法为例, 讲解快照建立之后的 `INodeDirectory` 是如何添加和删除子目录项的。

`addChild()`方法首先判断添加的子目录项是否已经在当前的 `INodeDirectory` 中, 然后判断

当前 `INodeDirectory` 对象是否在最新的快照中, 如果是则获取 `DirectoryWithSnapshotFeature` 对象, 然后在这个对象上调用 `addChild()` 方法 (参考 `DirectoryWithSnapshotFeature` 类小节)。如果不在最新的快照中, 则调用朴素的 `addChild()` 方法, 直接将 `INode` 节点加入 `INodeDirectory` 的 `children` 集合中。

```
public boolean addChild(INode node, final boolean setModTime,
    final int latestSnapshotId) throws QuotaExceededException {
    final int low = searchChildren(node.getLocalNameBytes());
    if (low >= 0) {
        return false; // 如果要添加的 INode 节点已经在目录中, 则返回
    }

    if (isInLatestSnapshot(latestSnapshotId)) {
        // 获取 DirectoryWithSnapshotFeature 对象
        DirectoryWithSnapshotFeature sf = this.getDirectoryWithSnapshotFeature();
        if (sf == null) {
            sf = this.addSnapshotFeature(null);
        }
        // 在 DirectoryWithSnapshotFeature 上调用 addChild() 方法
        return sf.addChild(this, node, setModTime, latestSnapshotId);
    }
    // 将要添加的 INode 加入 children 字段中保存
    addChild(node, low);
    if (setModTime) {
        // 更新父目录的修改时间
        updateModificationTime(node.getModificationTime(), latestSnapshotId);
    }
    return true;
}

public boolean addChild(INode node) {
    // 查找 INode 节点的插入位置
    final int low = searchChildren(node.getLocalNameBytes());
    if (low >= 0) {
        return false;
    }
    // 调用 addChild() 方法插入 INode 到 children 字段中保存
    addChild(node, low);
    return true;
}

private void addChild(final INode node, final int insertionPoint) {
    if (children == null) {
        children = new ArrayList<INode>(DEFAULT_FILES_PER_DIRECTORY);
    }
    node.setParent(this);
    // 插入 INode 到 children 集合的指定位置
    children.add(-insertionPoint - 1, node);
}
```

```

    if (node.getGroupName() == null) {
        node.setGroup(getGroupName());
    }
}

```

查询文件

在目录上添加了快照功能之后，获取 `INodeDirectory` 的子目录项就变得复杂了，也就是获取目录在指定快照版本中的某个子目录项。我们参考 `INodeDirectory.getChild()` 方法的实现，`getChild()` 方法首先获取目标目录的 `DirectoryWithSnapshotFeature` 对象，然后调用 `DirectoryWithSnapshotFeature.getChild()` 方法获取指定快照版本中的子目录项。`DirectoryWithSnapshotFeature.getChild()` 方法我们在 `DirectoryWithSnapshotFeature` 类小节中已经介绍过了，请读者参考。

```

public INode getChild(byte[] name, int snapshotId) {
    DirectoryWithSnapshotFeature sf;
    if (snapshotId == Snapshot.CURRENT_STATE_ID ||
        (sf = getDirectoryWithSnapshotFeature()) == null) {
        ReadOnlyList<INode> c = getCurrentChildrenList();
        final int i = ReadOnlyList.Util.binarySearch(c, name);
        return i < 0 ? null : c.get(i);
    }

    return sf.getChild(this, name, snapshotId);
}

```

至此，HDFS 新引入的快照功能就介绍完了。下面我们学习构建（UnderConstruction）特性的实现。

2. FileUnderConstructionFeature 实现

`FileUnderConstructionFeature` 的字面意思就是处于构建状态的特性，构建状态描述的是当客户端为写或者追加写（append）数据打开 HDFS 文件时，文件所处的状态就是构建状态。在 HDFS 2.6 版本之前，使用 `INodeFileUnderConstruction` 类描述处于构建状态的文件；2.6 版本添加 Feature 体系后，则使用 `FileUnderConstructionFeature` 类描述处于构建状态的文件。当客户端打开一个文件进行写或者追加写操作前，会首先调用 `INodeFile.toUnderConstruction()` 方法将该文件转变为构建状态。我们首先看一下 `toUnderConstruction()` 方法的实现。

为了保留原有接口的兼容性，HDFS 并没有将这个方法的命名为 `add*Feature()` 这种形式，而是保留了 `toUnderConstruction()` 这个方法名。`toUnderConstruction()` 方法的实现与 `SnapshotFeature` 是类似的，它首先构造 `FileUnderConstructionFeature` 对象，然后调用 `addFeature()` 将这个 Feature 对象添加到当前的 `INode` 节点上，最后返回 `INodeFile` 的引用。同理，如果将文件从构建状态转变为正常状态，则只需将 `FileUnderConstructionFeature` 对象从 `INode` 中移除即可，对应于方法 `toCompleteFile()`。`toUnderConstruction()` 方法的代码如下：

```

INodeFile toUnderConstruction(String clientName, String clientMachine) {
    Preconditions.checkState(!isUnderConstruction(),

```

```

    "file is already under construction");
// 构造 FileUnderConstructionFeature 对象
FileUnderConstructionFeature uc = new FileUnderConstructionFeature(
    clientName, clientMachine);
// 添加到 INode 的 features 字段中保存
addFeature(uc);
return this;
}

```

HDFS 写文件和追加写 (append) 文件的流程是非常复杂的, 所以 FileUnderConstructionFeature 中需要记录一些与写操作相关的属性, 包括:

- **clientName**: 发起文件写操作的客户端名称, 这个属性也用于租约管理功能。在 HDFS 中, 租约是名字节点维护的给予客户端在一定期限内可以进行文件写操作的权限的合同, 我们会在租约管理小节中介绍。
- **clientMachine**: 客户端所在主机。

如图 3-16 所示, FileUnderConstructionFeature 定义的方法包括了 clientName 和 clientMachine 两个属性的 get/set 方法, updateLengthOfLastBlock() 方法用于更新文件最后一个正在写的数据块的长度, cleanZeroSizeBlock() 方法则用于删除快照中文件的最后一个长度为 0 的数据块。



图 3-16 FileUnderConstructionFeature 类结构图

3.1.3 FSEditLog 类

在 Namenode 中, 命名空间 (namespace, 指文件系统中的目录树、文件元数据等信息) 是被全部缓存在内存中的, 一旦 Namenode 重启或者宕机, 内存中的所有数据将会全部丢失, 所以必须要有一种机制能够将整个命名空间持久化保存, 并且能在 Namenode 重启时重建命名空间。

目前 Namenode 的实现是将命名空间信息记录在一个叫作 fsimage (命名空间镜像) 的二进制文件中, fsimage 将文件系统目录树中的每个文件或者目录的信息保存为一条记录, 每条记录中包括了该文件 (或目录) 的名称、大小、用户、用户组、修改时间、创建时间等信息。Namenode 重启时, 会读取这个 fsimage 文件来重构命名空间。但是 fsimage 始终是磁盘上的一个文件, 不可能时时刻刻都跟 Namenode 内存中的数据结构保持同步, 并且 fsimage 文件一般都很大 (GB 级别的很常见), 如果所有的更新操作都实时地写入 fsimage 文件, 则会导致 Namenode 运行得十分缓慢, 所以 HDFS 每过一段时间才更新一次 fsimage 文件。

那么问题来了, 在一个新 fsimage 文件和上一个 fsimage 文件中进行的 Namenode 操作保存在哪里呢? HDFS 将这些操作记录在 editlog (编辑日志) 文件中, editlog 是一个日志文件, HDFS 客户端执行的所有写操作首先会被记录到 editlog 文件中。HDFS 会定期地将 editlog 文件与 fsimage 文件进行合并, 以保持 fsimage 跟 Namenode 内存中记录的命名空间完全同步。本节和下一节就重点介绍 editlog 以及 fsimage 的实现。

在 HDFS 源码中, 使用 FSEditLog 类来管理 editlog 文件。和 fsimage 文件不同, editlog 文件会随着 Namenode 的运行实时更新, 所以 FSEditLog 类的实现依赖于底层的输入流和输出流, 同时 FSEditLog 类还需要对外提供大量的 log*() 方法用于记录命名空间的修改操作。本节首先介绍 FSEditLog 类使用到的 transactionId 机制, 然后介绍 FSEditLog 类的状态机以及对外提供的方法, 最后介绍 editlog 文件的输入流和输出流。

1. transactionId 机制

HDFS 的 editlog 文件可以存放在多种容器中, 比如文件系统 (FileJournalManager 类管理)、共享 NFS (BackupJournalManager 类管理)、Bookkeeper (BookkeeperJournalManager 类管理) 等。而管理这些不同容器内文件的方法也有很多种, 目前 HDFS 采用的是基于 transactionId 的日志管理方法。

这里我们以 Namenode 元数据文件夹为例。图 3-17 显示的是在 Namenode 元数据文件夹中运行 tree 命令的结果, Namenode 元数据文件夹是存放 fsimage 文件和 editlog 文件的文件夹, 由 hdfs-site.xml 配置文件的 dfs.namenode.name.dir 配置项配置。在这个示例中, 元数据文件夹是 data/dfs/name 目录。

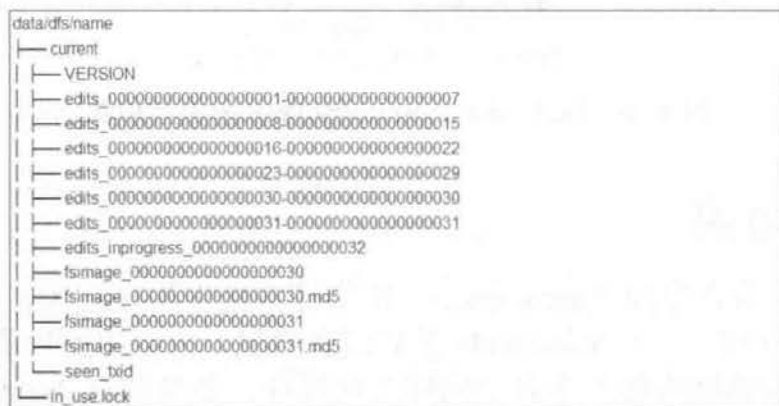


图 3-17 Namenode 元数据文件夹结构

这里我们首先了解一下 transactionId 的概念。TransactionId 与客户端每次发起的 RPC 操作相关, 当客户端发起一次 RPC 请求对 Namenode 的命名空间修改后, Namenode 就会在 editlog 中发起一个新的 transaction 用于记录这次操作, 每个 transaction 会用一个唯一的 transactionId 标识。理解清楚了 transactionId 的概念之后, 我们看一下这个文件夹中每个文件都有什么作用。

- edits_start transaction id-end transaction id: edits 文件就是我们描述的 editlog 文件, edits 文件中存放的是客户端执行的所有更新命名空间的操作。每个 edits 文件都包含了文件名中 start transaction id - end transaction id 之间的所有事务。这里以 edits_000*01 -

000*07 为例, 这个文件记录了 transaction id 在 1 和 7 之间的所有事务 (transaction)。

- **edits_inprogress_start transaction id:** 正在进行处理的 editlog。所有从 start transaction id 开始的新的修改操作都会记录在这个文件中, 直到 HDFS 重置 (roll) 这个日志文件。重置操作会将 inprogress 文件关闭, 并将 inprogress 文件改名为正常的 editlog 文件 (如上一项所示), 同时还会打开一个新的 inprogress 文件, 记录正在进行的事务。例如当前文件夹中的 edits_inprogress_0000000000000000032 文件, 这个文件记录了所有 transaction id 大于 32 的新开始的事务, 我们将这个事务区间称为一个日志段落 (segment)。Namenode 元数据文件夹中存在这个文件有两种可能, 要么是 Active Namenode 正在写入数据, 要么是前一个 Namenode 没有正确地关闭。
- **fsimage_end transaction id:** fsimage 文件是 Hadoop 文件系统元数据的一个永久性的检查点, 包含 Hadoop 文件系统中 end transaction id 前的完整的 HDFS 命名空间元数据镜像, 也就是 HDFS 所有目录和文件对应的 INode 的序列化信息。以当前文件夹为例, fsimage_000*31 就是 fsimage_000*30 与 edits_000*31-000*31 合并后的镜像文件, 保存了 transaction id 小于 32 的 HDFS 命名空间的元数据。每个 fsimage 文件还有一个对应的 md5 文件, 用来确保 fsimage 文件的正确性, 以防止磁盘异常发生。
- **seen_txid:** 这个文件中保存了上一个检查点 (checkpoint) (合并 edits 和 fsimage 文件) 以及编辑日志重置 (editlog roll) (持久化当前的 inprogress 文件并且创建一个新的 inprogress 文件) 时最新的事务 id (transaction id)。要特别注意的是, 这个事务 id 并不是 Namenode 内存中最新的事务 id, 因为 seen_txid 只在检查点操作以及编辑日志重置操作时更新。这个文件的作用在于 Namenode 启动时, 可以利用这个文件判断是否有 edits 文件丢失。例如, Namenode 使用不同的目录保存 fsimage 以及 edits 文件, 如果保存 edits 的目录内容丢失, Namenode 将会使用上一个检查点保存的 fsimage 启动, 那么上一个检查点之后的所有事务都会丢失。为了防止发生这种状况, Namenode 启动时会检查 seen_txid 并确保内存中加载的事务 id 至少超过 seen_txid; 否则 Namenode 将终止启动操作。

2. FSEditLog 状态机

介绍完 transactionId 机制后, 我们来学习 FSEditLog 类的实现。FSEditLog 类被设计成一个状态机, 用内部类 FSEditLog.State 描述。FSEditLog 有以下 5 个状态。

- **UNINITIALIZED:** editlog 的初始状态。
- **BETWEEN_LOG_SEGMENTS:** editlog 的前一个 segment 已经关闭, 新的还没开始。
- **IN_SEGMENT:** editlog 处于可写状态。
- **OPEN_FOR_READING:** editlog 处于可读状态。
- **CLOSED:** editlog 处于关闭状态。

对于非 HA 机制的情况, FSEditLog 应该开始于 UNINITIALIZED 或者 CLOSED 状态 (因为在构造 FSEditLog 对象时, FSEditLog 的成员变量 state 默认为 State.UNINITIALIZED)。FSEditLog 初始化完成之后进入 BETWEEN_LOG_SEGMENTS 状态, 表示前一个 segment 已经关闭, 新的还没开始, 日志已经做好了准备。当打开日志服务时, 改变 FSEditLog 状态为

IN_SEGMENT 状态，表示可以写 editlog 文件了。

对于 HA 机制的情况，FSEditLog 同样应该开始于 UNINITIALIZED 或者 CLOSED 状态，但是在完成初始化后 FSEditLog 并不进入 BETWEEN_LOG_SEGMENTS 状态，而是进入 OPEN_FOR_READING 状态（因为目前 Namenode 启动时都是以 Standby 模式启动的，然后通过 DFSHAAdmin 发送命令把其中一个 Standby NameNode 转换成 Active Namenode）。

下面我们看一下 FSEditLog 的状态转移图，如图 3-18 所示。图中所示的方法会在后面的小节中详细介绍。

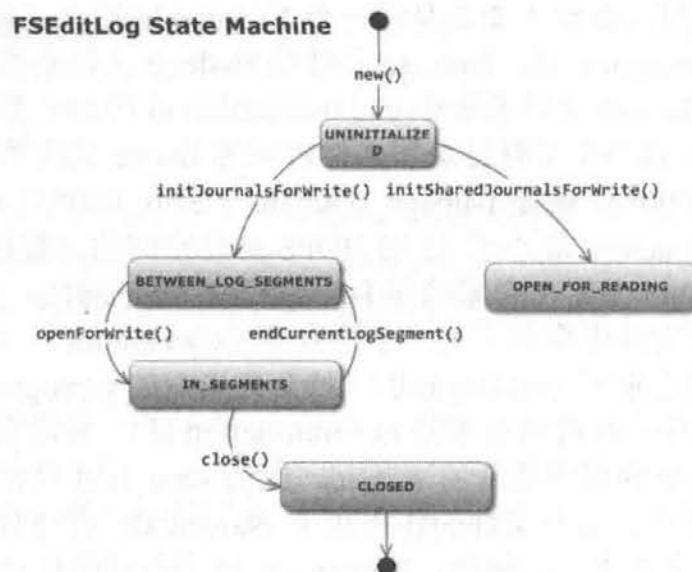


图 3-18 FSEditLog 的状态转移图

(1) initJournalsForWrite()

initJournalsForWrite()方法是 FSEditLog 的 public 方法，调用这个方法会将 FSEditLog 从 UNINITIALIZED 状态转换为 BETWEEN_LOG_SEGMENTS 状态。

```

public synchronized void initJournalsForWrite() {
    Preconditions.checkState(state == State.UNINITIALIZED ||
        state == State.CLOSED, "Unexpected state: %s", state);
    // 检查之前的状态
    initJournals(this.editsDirs); // 调用 initJournals() 方法
    state = State.BETWEEN_LOG_SEGMENTS; // 状态转换为 BETWEEN_LOG_SEGMENTS
}
  
```

initJournalsForWrite()方法调用了 initJournals()方法，initJournals()方法会根据传入的 dirs 变量（保存的是 editlog 文件的存储位置，都是 URI）初始化 journalSet 字段（JournalManager 对象的集合）。初始化之后，FSEditLog 就可以调用 journalSet 对象的方法向多个日志存储位置写 editlog 文件了。

JournalManager 类是负责在特定存储目录上持久化 editlog 文件的类，它的 format()方法负责格式化底层存储，startLogSegment()方法负责从指定事务 id 开始记录一个操作的段落，

finalizeLogSegment()方法负责完成指定事务 id 区间的写操作。这里之所以抽象这个接口，是因为 Namenode 可能将 editlog 文件持久化到不同类型的存储上，也就需要不同类型的 JournalManager 来管理，所以需要定义一个抽象的接口。如图 3-19 所示，JournalManager 有多个子类，普通的文件系统由 FileJournalManager 类管理、共享 NFS 由 BackupJournalManager 类管理、Bookkeeper 由 BookkeeperJournalManager 类管理、Quorum 集群则由 QuorumJournalManager 类管理。

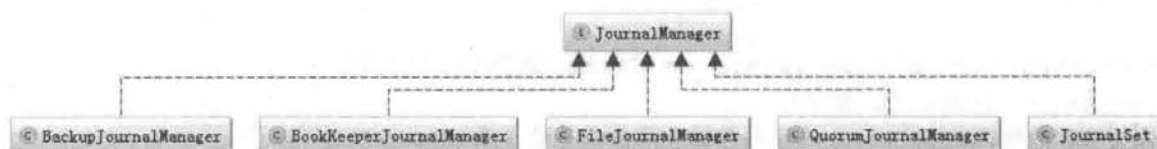


图 3-19 JournalManager 继承关系图

了解了 JournalManager 后，我们看一下 initJournals()方法的实现，代码如下：

```

private synchronized void initJournals(List<URI> dirs) {
    int minimumRedundantJournals = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_KEY,
        DFSConfigKeys.DFS_NAMENODE_EDITS_DIR_MINIMUM_DEFAULT);

    // 初始化 journalSet 集合，存放存储路径对应的所有 JournalManager 对象
    journalSet = new JournalSet(minimumRedundantJournals);

    // 根据传入的 URI 获取对应的 JournalManager 对象
    for (URI u : dirs) {
        boolean required = FSNamesystem.getRequiredNamespaceEditsDirs(conf)
            .contains(u);
        if (u.getScheme().equals(NNStorage.LOCAL_URI_SCHEME)) {
            StorageDirectory sd = storage.getStorageDirectory(u);
            if (sd != null) {
                // 本地 URI，则加入 FileJournalManager 即可
                journalSet.add(new FileJournalManager(conf, sd, storage), required);
            }
        } else {
            // 否则根据 URI 创建对应的 JournalManager 对象，并放入 journalSet 中保存
            journalSet.add(createJournal(u), required);
        }
    }
}

```

(2) initSharedJournalsForRead()

initSharedJournalsForRead()方法是 FSEditLog 的 public 方法，用在 HA 情况下。调用这个方法会将 FSEditLog 从 UNINITIALIZED 状态转换为 OPEN_FOR_READING 状态。

与 initJournalsForWrite()方法相同，initSharedJournalsForRead()方法也调用了 initJournals()方法执行初始化操作，只不过 editlog 文件的存储位置不同，在 HA 的情况下，editlog 文件的

存储目录为共享存储目录，这个共享存储目录由 Active Namenode 和 Standby Namenode 共享读取。initSharedJournalsForRead()方法的代码如下：

```
public synchronized void initSharedJournalsForRead() {
    if (state == State.OPEN_FOR_READING) {
        LOG.warn("Initializing shared journals for READ, already open for READ",
            new Exception());
        return;
    }
    Preconditions.checkState(state == State.UNINITIALIZED ||
        state == State.CLOSED);
    // 对于 HA 的情况，editlog 的日志存储目录为共享的目录 sharedEditsDirs
    initJournals(this.sharedEditsDirs);
    state = State.OPEN_FOR_READING;
}
```

(3) openForWrite()

openForWrite()方法用于初始化 editlog 文件的输出流，并且打开第一个日志段落 (log segment)。在非 HA 机制下，调用这个方法会完成 BETWEEN_LOG_SEGMENTS 状态到 IN_SEGMENT 状态的转换。本节结合下面的目录结构说明 openForWrite()方法的实现。

```
data/dfs/name
├── current
│   ├── VERSION
│   ├── edits_00000000000000000001-00000000000000000007
│   ├── edits_00000000000000000008-00000000000000000015
│   ├── edits_00000000000000000016-00000000000000000022
│   ├── edits_00000000000000000023-00000000000000000029
│   ├── edits_00000000000000000030-00000000000000000030
│   ├── edits_00000000000000000031-00000000000000000031
│   ├── fsimage_00000000000000000030
│   ├── fsimage_00000000000000000030.md5
│   ├── fsimage_00000000000000000031
│   ├── fsimage_00000000000000000031.md5
│   └── seen_txid
└── in_use.lock
```

如图 3-20 所示，openForWrite()方法的执行分为这样几个部分，分别调用了三个不同的方法。

- getLastWrittenTxId()——查找已经写到 editlog 日志文件中的最新的 transactionId，对应图 3-20 中的情况，返回的是 31。
- journalSet.selectInputStreams()——传入了参数 segmentTxId，这个参数会作为这次操作的 transactionId，值为 editlog 已经记录的最新的 transactionId 加 1 (这里是 31+1=32)。selectInputStreams()方法会判断有没有一个以 segmentTxId (32) 开始的日志 (从上面的例子来看，是没有的)，如果没有则表示当前 transactionId 的值选择正确，可以打开新的 editlog 文件记录以 segmentTxId 开始的日志段落。如果方法找到了包含这

个 transactionId 的 editlog 文件, 则表示出现了两个日志 transactionId 交叉的情况, 抛出异常。

- startLogSegment()——开始记录 transactionId 为 32 的日志段落, 新建 edits_inprogress_32 文件。同时将 FSEditlog 的状态转变为 IN_SEGMENT。

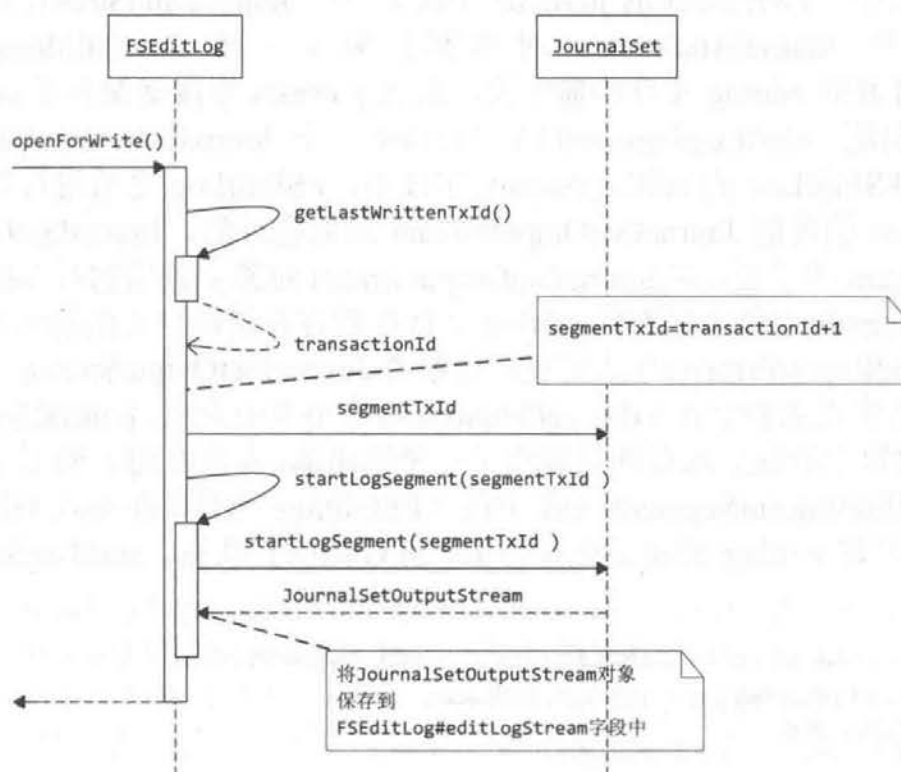


图 3-20 openForWrite()方法调用流程

`openForWrite()`方法的代码如下:

```

synchronized void openForWrite() throws IOException {
    Preconditions.checkState(state == State.BETWEEN_LOG_SEGMENTS,
        "Bad state: %s", state);

    // 返回最后一个写入 log 的 transactionId+1, 作为本次操作的 transactionId
    long segmentTxId = getLastWrittenTxId() + 1;

    // 这里判断, 有没有包含这个新的 segmentTxId 的 editlog 文件, 如果有则抛出异常
    List<EditLogInputStream> streams = new ArrayList<EditLogInputStream>();
    journalSet.selectInputStreams(streams, segmentTxId, true, true);
    if (!streams.isEmpty()) {
        // ...
        throw new IllegalStateException(error);
    }

    // 调用 startLogSegment() 方法
}

```