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。

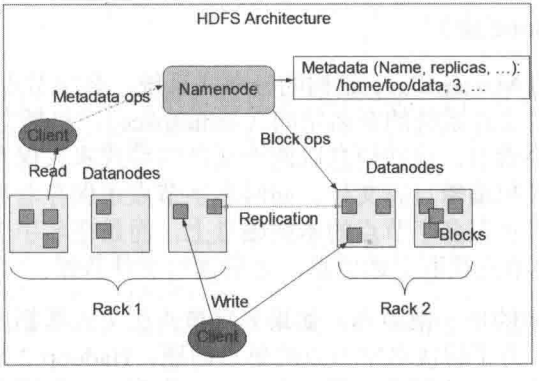

图 1-1　HDFS 体系结构示意图

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 DFSClient 类的基础上的，DFSClient 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 Hadoop2.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（HAServiceProtocol）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。这个方法的参数是 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()、getAddtionnalDatanodes()、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。favoredNodes 参数则是客户端所希望的保存数据块副本的数据节

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()、getAdditionnalDatanode()、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在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
FileSystem.rename2()	rename2()	更改文件/目录名称
FileSystem.concat()	concat()	将两个已有文件拼接成一个
FileSystem.delete()	delete()	从文件系统中删除指定文件或者目录
FileSystem.mkdirs()	mkdirs()	以指定名称和权限在文件系统中创建目录
FileSystem.listStatus()	getListing()	读取一个指定目录下的所有项目
FileSystem.set* ()	setPermission() setOwner() setTimes() setReplication()	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
FileSystem. listCorruptFileBlocks()	listCorruptFileBlocks()	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
FileSystem.getFileStatus	getFileInfo()	获取文件/目录的属性
FilSystem.getFileLinkStatus	getFileLinkStatus()	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
DistributedFileSystem.isFileClosed	isFileClosed()	判断指定文件是否关闭了
FilSystem.getContentSummary	getContentSummary()	获取文件/目录使用的存储空间信息
FileSystem.createSymlink()	createSymlink()	对于已经存在的文件创建符号链接
resolveLink()	getLinkTarget()	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（fsiamge 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。

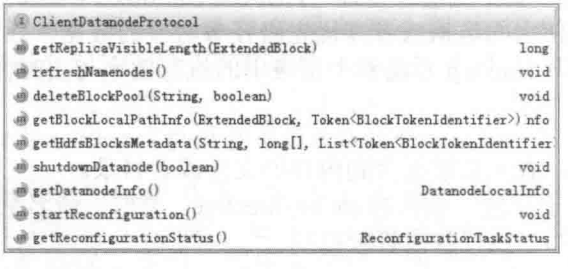

图 1-2　ClientDatanodeProtocol 定义

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 ClientDatanodeProtoco 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 dfsadmindatanodehost: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。

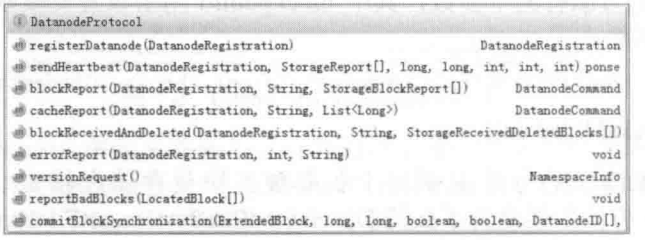

图 1-3 DatanodeProtocol 定义

（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：Datanode Registration 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。