北京航空航天大學
BEIHANG UNIVERSITY

2018-2019 学年第二学期
《软件体系结构》课程作业

HDFS 体系结构分析

姓　名：	高松		学　号：	ZY1806406
姓　名：	李佳磊		学　号：	ZY1806407
姓　名：	李琪		学　号：	ZY1806704

时间：　2019 年 5 月 30 日

目 录

一、项目简介.....1
1.1 基本介绍.....1
1.2 架构介绍.....1
1.3 HDFS 特性.....2
1.4 HDFS 功能介绍.....3
1.5 项目代码行数统计.....4
1.6 项目中包和类的数量.....4
二、需求分析.....5
2.1 项目功能需求.....5
2.1.1 透明性.....5
2.1.6 安全性.....6
2.2 主要质量属性.....7
2.2.1 性能.....7
2.2.2 可用性.....7
2.2.3 可修改性.....7
2.2.4 安全性.....8
2.2.5 可测试性.....8
2.2.6 易用性.....9
三、软件体系结构设计.....10
3.1 HDFS 核心内容.....10
3.1.1、Block.....11
3.1.2、Namenode.....12
3.1.3、DataNode.....20
3.2 HDFS 通信协议.....25
3.2.1 Hadoop RPC 接口.....26
3.2.2 RPC 框架.....26
3.2.3 RPC 的实现.....28
3.3 HDFS 客户端.....47
3.3.1 HDFS 客户端文件层次结构.....47
3.3.2 DFSOutputstream.java 逆向工程.....48
3.3.3 DFSClient 实现.....49
3.3.4 文件读操作与输入流.....53
3.3.5 文件写操作与输出流.....55
3.4 NFS 模块介绍.....61
3.5 RBF 模块介绍.....63
四、设计特色分析.....65
4.1 高容错性.....65
4.1.1 设计原理.....66
4.1.2 优势与需求.....68
4.2 读写流程.....69
4.2.1 写流程.....69
4.2.2 读流程.....70

4.2.3 优势与需求.....70
4.3 缺点.....70
五、组内分工情况.....72
六、参考文献.....74
七、附件：体系结构的分解过程.....75

一、项目简介

1.1 基本介绍

Hadoop 分布式文件系统（HDFS）是一种分布式文件系统，设计用于在商用硬件集群上运行，用于存储具有流数据访问模式的非常大的文件。它与现有的分布式文件系统有许多相似之处。但是，与其他分布式文件系统的差异很大。HDFS 具有高度容错能力，旨在部署在低成本硬件上。HDFS 提供对应用程序数据的高吞吐量访问，适用于具有大型数据集的应用程序。HDFS 放宽了一些 POSIX 要求，以实现对文件系统数据的流式访问。HDFS 最初是作为 Apache Nutch 网络搜索引擎项目的基础设施而构建的。HDFS 是 Apache Hadoop Core 项目的一部分。

1.2 架构介绍

HDFS 具有主/从架构。HDFS 分别存储文件系统元数据和应用程序数据。HDFS 将元数据存储在名为 NameNode 的专用服务器上。应用程序数据存储在名为 DataNodes 的其他服务器上。NameNode 是一个主服务器，用于管理文件系统命名空间并管理客户端对文件的访问。此外，还有许多 DataNode，通常是群集中每个节点一个，用于管理连接到它们运行的节点的存储。HDFS 公开文件系统命名空间，并允许用户数据存储在文件中。

在内部，文件被分成一个或多个块，这些块存储在一组 DataNode 中。NameNode 执行文件系统命名空间操作，如打开，关闭和重命名文件和目录。它还确定了块到 DataNode 的映射。DataNode 负责提供来自文件系统客户端的读写请求。DataNode 还根据 NameNode 的指令执行块创建，删除和复制。

所有服务器都是完全连接的，并使用基于 TCP 的协议相互通信。与其他 filsystem 不同，HDFS 中的 DataNode 不依赖于 RAID 等数据保护机制来使数据持久。相反，文件内容将复制到多个 DataNode 上以确保可靠性。在确保数据持

久性的同时，该策略还具有额外的优势，即数据传输带宽成倍增加，并且在所需数据附近定位计算的机会更多。

1.3 HDFS 特性

我将从以下几点介绍 HDFS 的特性

（i）、数据的种类和数量：

当我们谈论 HDFS 的时候，我们谈论的是存储巨大的数据，即 TB 级和 PB 级的数据和不同类型的数据。所以，您可以将任何类型的数据存储到 HDFS 中，无论是结构化的，非结构化的还是半结构化的。

（ii）、可靠性和容错性：

当您将数据存储在 HDFS 上时，它会将给定的数据内部分割为数据块，并以分布的方式将其存储在 Hadoop 集群中。关于哪个数据块位于哪个数据节点上的信息被记录在元数据中。NameNode 管理元数据， DataNode 负责存储数据，NameNode 也复制数据，即维护数据的多个副本。数据的这种复制使得 HDFS 非常可靠和容错。因此，即使任何节点失败，我们也可以从驻留在其他数据节点上的副本中检索数据。默认情况下，复制因子为 3.因此，如果将 1 GB 的文件存储在 HDFS 中，则最终将占用 3 GB 的空间。名称节点定期更新元数据并保持复制因子一致。

（iii）、数据完整性：

数据完整性将讨论存储在我的 HDFS 中的数据是否正确。HDFS 不断检查存储的数据的完整性与其校验和。如果发现任何错误，它会向名称节点报告。然后，名称节点创建额外的新副本，因此删除损坏的副本。

（iv）、高吞吐量：

吞吐量是单位时间内完成的工作量。它讨论了如何从文件系统访问数据的速度。基本上，它给你一个关于系统性能的见解。正如你在上面的例子中看到的那样，我们共用十台机器来增强计算。在那里我们能够将处理时间从 43 分钟缩短到只有 4.3 分钟，因为所有的机器都在并行工作。因此，通过并行处理数据，能大大减少了处理时间，从而实现了高吞吐量。

（v）、数据局部性：

数据局部性讨论的是将处理单元移动到数据而不是数据到处理单元。在我们的传统系统中，我们曾经把数据带到应用层，然后进行处理。但是现在，由于数据的体系结构和庞大的数据量，把数据带到应用层会使网络性能显着降低。因此，在 HDFS 中，我们将计算部分带到数据所在的数据节点。因此，你不移动数据，你正在把程序或处理部分的数据。

1.4 HDFS 功能介绍

Hadoop 不需要昂贵，高度可靠的硬件。它被设计为在商用硬件集群（可从多个供应商处获得的常用硬件）上运行，对于这些集群，集群中节点故障的可能性很高，至少对于大型集群而言。HDFS 旨在在面对此类故障时继续工作而不会对用户造成明显的中断。

HDFS 有很多功能：

- 通过检测故障并应用快速自动恢复来实现容错
- 通过 MapReduce 流进行数据访问
- 简单而强大的一致性模型
- 处理逻辑接近数据，而不是接近处理逻辑的数据
- 跨异构商品硬件和操作系统的可移植性
- 可扩展性，可靠地存储和处理大量数据
- 通过在商品个人计算机集群之间分配数据和处理来实现经济
- 通过分布数据和逻辑以在数据所在的节点上并行处理它来提高效率
- 通过自动维护多个数据副本并在发生故障时自动重新部署处理逻辑来实现可靠性

其具体实现如下所示：

- （i）HDFS 集群分为两大角色：NameNode、DataNode
- （ii）NameNode 负责管理整个文件系统的元数据
- （iii）DataNode 负责管理用户的文件数据块
- （iv）文件会按照固定的大小（blocksize）切成若干块后分布式存储在若干台 datanode 上
- （v）每一个文件块可以有多个副本，并存放在不同的 datanode 上

- （vi）Datanode 会定期向 Namenode 汇报自身所保存的文件 block 信息，而 namenode 则会负责保持文件的副本数量
- （vii）HDFS 的内部工作机制对客户端保持透明，客户端请求访问 HDFS 都是通过向 namenode 申请来进行

1.5 项目代码行数统计

Language	files	blank	comment	code
Java	2068	65195	123828	416546
XML	51	2840	2555	97541
JSON	4	4	0	9876
C	50	1263	1984	8993
CSS	9	189	118	8567
Javascript	19	867	483	3519
Protocol Buffers	24	788	1578	3080
Maven	7	48	100	1801
C/C++ Header	27	391	1896	1471
HTML	15	133	222	1461
Bourne Shell	18	208	437	613
DOS Batch	6	63	0	360
Bourne Again Shell	2	29	46	260
CMake	5	40	100	224
Velocity Template Language	1	49	11	137
XSLT	2	7	22	57
SUM:	2308	72114	133380	554506

HDFS 整个工程共有代码 554506 行，其中以 Java 为主，占 416546 行。

1.6 项目中包和类的数量

HDFS 共包含 6 个包，分别是：
hadoop-hdfs
Hadoop-hdfs-client
Hadoop-hdfs-httpfs
Hadoop-hdfs-native-client
Hadoop-hdfs-nfs
Hadoop-hdfs-rbs
共包含 2068 个类。

二、需求分析

2.1 项目功能需求

分布式文件系统的设计需求一般是：透明性、并发控制、可伸缩性、容错以及安全需求等。

2.1.1 透明性

首先是透明性，按照开放分布式处理的标准确定有 8 种透明性：访问的透明性、位置的透明性、并发透明性、复制透明性、故障透明性、移动透明性、性能透明性和伸缩透明性。对于分布式文件系统，最重要的是要达到 5 个透明性要求：

1）访问的透明性：用户能通过相同的操作来访问本地文件和远程文件资源。HDFS 可以做到这一点，若 HDFS 设置成本地文件系统，而非分布式，那么读写分布式 HDFS 的程序可以不用修改地读写本地文件，只要修改配置文件。可见，HDFS 提供的访问的透明性是不完全的，它是构建于 java 之上，不能像 NFS 或者 AFS 那样去修改 unix 内核，同时将本地文件和远程文件以一致的方式处理。

2）位置的透明性：使用单一的文件命名空间，在不改变路径名的前提下，文件或者文件集合可以被重定位。HDFS 集群只有一个 Namenode 来负责文件系统命名空间的管理，文件的 block 可以重新分布复制，block 可以增加或者减少副本，副本可以跨机架存储，而这一切对客户端都是透明的。

3）移动的透明性，这一点与位置的透明性类似，HDFS 中的文件经常由于节点的失效、增加或者 replication 因子的改变或者重新均衡等进行着复制或者移动，而客户端和客户端程序并不需要改变什么，Namenode 的 edits 日志文件记录着这些变更。

4）性能的透明性和伸缩的透明性：HDFS 的目标就是构建在大规模廉价机器上的分布式文件系统集群，可伸缩性毋庸置疑。

2.1.2 并发控制

并发控制是指客户端对于文件的读写不应该影响其他客户端对同一个文件的读写。要想实现近似原生文件系统的单个文件拷贝语义，分布式文件系统需要做出复杂的交互，例如采用时间戳，或者类似回调承诺（类似服务器到客户端的 RPC 回调，在文件更新的时候；回调有两种状态：有效或者取消。客户端通过检查回调承诺的状态，来判断服务器上的文件是否被更新过）。HDFS 并没有这样做，它的机制非常简单，任何时间都只允许一个写的客户端，文件经创建并写入之后不再改变，它的模型是 write-one-read-many，一次写，多次读。这与它的应用场合是一致，HDFS 的文件大小通常是兆至 T 级的，这些数据不会经常修改，最经常的是被顺序读并处理，随机读很少，因此 HDFS 非常适合 MapReduce 框架或者 web crawler 应用。HDFS 文件的大小也决定了它的客户端不能像某些分布式文件系统那样缓存常用到的几百个文件。

2.1.3 文件复制

文件复制是指一个文件可以表示为其内容在不同位置的多个拷贝。这样做带来了两个好处：访问同个文件时可以从多个服务器中获取从而改善服务的伸缩性，另外就是提高了容错能力，某个副本损坏了，仍然可以从其他服务器节点获取该文件。HDFS 文件的 block 为了容错都将被备份，根据配置的 replication 因子来，默认是 3。副本的存放策略也是很有讲究，

一个放在本地机架的节点，一个放在同一机架的另一节点，另一个放在其他机架上。这样可以较大限度地防止因故障导致的副本的丢失。不仅如此，HDFS 读文件的时候也将优先选择从同一机架乃至同一数据中心的节点上读取 block。

2.1.4 硬件和操作系统的异构性

由于构建在 java 平台上，HDFS 的跨平台能力毋庸置疑，得益于 java 平台已经封装好的文件 IO 系统，HDFS 可以在不同的操作系统和计算机上实现同样的客户端和服务端程序。

2.1.5 容错能力

在分布式文件系统中，尽量保证文件服务在客户端或者服务端出现问题的时候能正常使用是非常重要的。HDFS 的容错能力大致可以分为两个方面：文件系统的容错性以及 Hadoop 本身的容错能力。文件系统的容错性通过这么几个手段：

1)在 Namenode 和 Datanode 之间维持心跳检测，当由于网络故障之类的原因，导致 Datanode 发出的心跳包没有被 Namenode 正常收到的时候，Namenode 就不会将任何新的 IO 操作派发给那个 Datanode，该 Datanode 上的数据被认为是无效的，因此 Namenode 会检测是否有文件 block 的副本数目小于设置值，如果小于就自动开始复制新的副本并分发到其他 Datanode 节点。

2）检测文件 block 的完整性，HDFS 会记录每个新创建的文件的所有 block 的校验和。当以后检索这些文件的时候，从某个节点获取 block，会首先确认校验和是否一致，如果不一致，会从其他 Datanode 节点上获取该 block 的副本。

3）集群的负载均衡，由于节点的失效或者增加，可能导致数据分布的不均匀，当某个 Datanode 节点的空闲空间大于一个临界值的时候，HDFS 会自动从其他 Datanode 迁移数据过来。

4）Namenode 上的 fsimage 和 edits 日志文件是 HDFS 的核心数据结构，如果这些文件损坏了，HDFS 将失效。因而，Namenode 可以配置成支持维护多个 FsImage 和 Editlog 的拷贝。任何对 FsImage 或者 Editlog 的修改，都将同步到它们的副本上。它总是选取最近的一致的 FsImage 和 Editlog 使用。Namenode 在 HDFS 是单点存在，如果 Namenode 所在的机器错误，手工的干预是必须的。

5)文件的删除，删除并不是马上从 Namenode 移出 namespace，而是放在/trash 目录随时可恢复，直到超过设置时间才被正式移除。

若是 Hadoop 本身的容错性，Hadoop 支持升级和回滚，当升级 Hadoop 软件时出现 bug 或者不兼容现象，可以通过回滚恢复到老的 Hadoop 版本。

2.1.6 安全性

HDFS 的安全性是比较弱的，只有简单的与 unix 文件系统类似的文件许可控制，未来版本会实现类似 NFS 的 kerberos 验证系统。

HDFS 作为通用的分布式文件系统并不适合，它在并发控制、缓存一致性以及小文件读写的效率上是比较弱的。但是它有自己明确的设计目标，那就是支持大的数据文件（兆至 T 级），并且这些文件以顺序读为主，以文件读的高吞吐量为目标，并且与 MapReduce 框架紧密结合。

2.2 主要质量属性

HDFS 系统的质量属性主要分为：可用性，可修改性，性能，安全性，可测试性，易用性。下面将根据这 6 个质量属性对 HDFS 进行描述。

2.2.1 性能

性能与时间有关，事件发生时系统对其做出的反应，对应有海量用户同时使用 HDFS 系统作为场景进行描述。

场景	海量用户同时使用 HDFS 系统
刺激源	用户
刺激	疯狂使用系统
制品	系统（用户界面）
环境	正常操作
响应	大量的操作同时被处理
响应度量	每个操作平均等待时间为 3s

2.2.2 可用性

可用性指在系统在发生故障时的运行时间和系统正常运行时间比例。针对海量用户同时使用 HDFS 系统造成系统崩溃的场景进行描述。

场景	海量用户同时使用 HDFS 系统
刺激源	海量用户
刺激	用户在同一时间访问量过大，造成系统崩溃
制品	处理系统崩溃的处理器
环境	正常操作
响应	页面无法加载
响应度量	提示请重试，短时间内恢复系统正常运行

2.2.3 可修改性

可修改性要关注的点在于：可以修改什么？何时进行变更以及由于谁进行变更。

可修改性的一个场景用户修改 HDFS 系统上的个人信息，主要针对这个场景进行描述。

场景	用户修改信息

刺激源	用户
刺激	修改个人信息
制品	个人设置页面
环境	运行时
响应	对一条信息的信息进行修改时，不会影响其他资料及功能
响应度量	不影响其他资料及正常浏览等功能

2.2.4 安全性

安全性指衡量系统在向合法用户提供服务的同时，阻止非授权使用的能力。在 HDFS 系统中出现的安全性问题就在于黑客的恶意攻击，以致于导致系统的崩溃和瘫痪。

场景	黑客试图非法入侵系统后台，获取用户信息
刺激源	黑客（非授权用户）
刺激	试图采用非法手段来入侵 HDFS 系统后台以获取信息
制品	系统中的数据
环境	在线环境中
响应	对访问用户进行验证，阻止不正当的用户访问数据
响应度量	查到非法入侵时在 1 秒以内做出反应，进行阻拦处理，保护数据安全性

2.2.5 可测试性

可测试性是指通过测试来揭示软件缺陷的容易程度。对应场景为 HDFS 系统在上线初进行系统整体的测试。

场景	内测用户使用不正确的用户名密码来登录系统
刺激源	系统内测用户
刺激	内测阶段时间测试登录系统，输入错误密码
制品	完整应用
环境	完成时
响应	密码错误的情况下无法登录
响应度量	错误密码账户 100%不能完成登陆

2.2.6 易用性

易用性关注的是对用户来说完成某个期望任务的容易程度和系统所提供的用户支持的种类。对应 HDFS 系统中的一个场景为用户在登录时，会显示记住用户名和密码的选项。

场景	用户在登录时，会显示记住用户名和密码的选项
刺激源	用户
刺激	用户使用更加便捷
制品	HDFS 系统
环境	正常运行时
响应	显示记住用户名和密码的选项
响应度量	下次登录时自动填充用户名和密码

三、软件体系结构设计

3.1 HDFS 核心内容

在代码中HDFS总类中包括的包有net，protool，protocolPB，qjournal，security，server，tools，util，web几个包，其中各包功能如下：

Net：主要包括的是DFS网络拓扑以及Tcp等与网络有关的类。

Protocol：主要包含的类是有关数据的一些配置。

ProtocolPB ：主要包含了PB文件相关的类，定义了message以及service，Security：主要包含的是安全有关的内容。

Tools：包含的是相关工具。

Util：是相关使用时需要的类。

而最重要的是server，server中最重要的是blockmanagement，datanode，namenode，下面将重点进行介绍。如图3-1所示为HDFS核心功能的架构图。

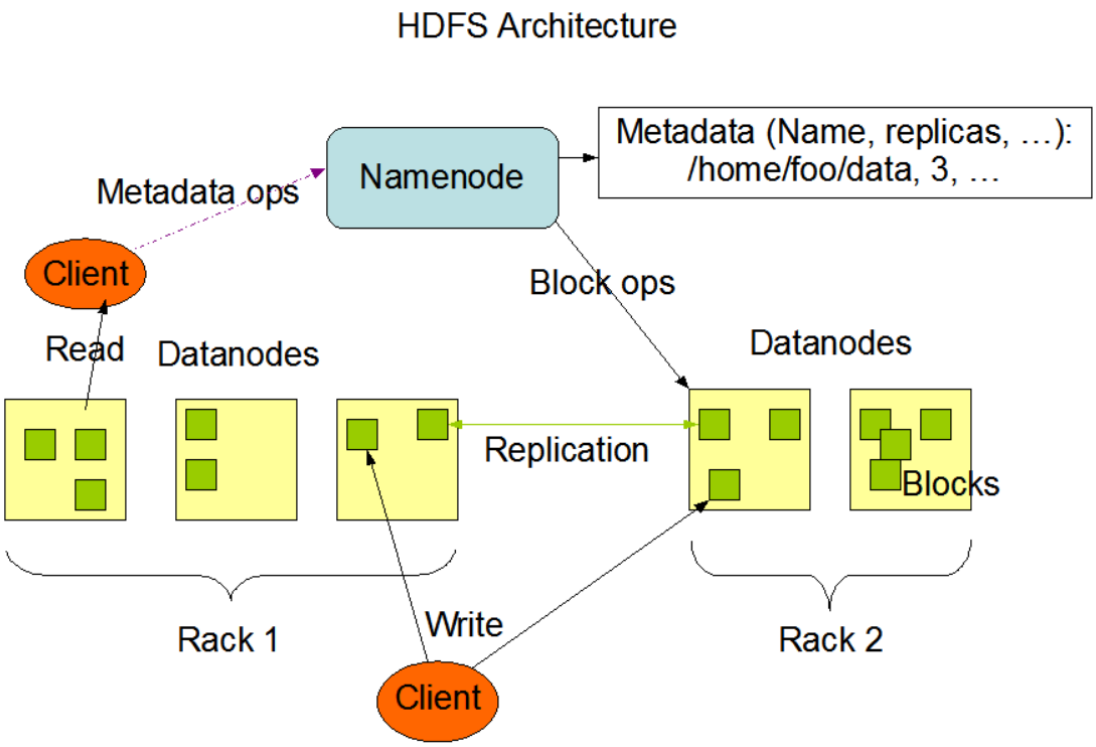

图 3-1 HDFS 架构图

3.1.1、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 会将同一个数据块元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。如下类图所示，在代码中通过BlockInfo存储与块有关的信息，通过BlockManager对块进行管理，如进行块与块之间的连接，对块进行复制、修复，维护块的安全，建立数据流管道等。如图3-2所示为block的相关类图。

图 3-2 BlockInfo 类图

3.1.2、Namenode

HDFS 集群有两种类型的节点以主-工模式运行：名称节点（主节点）和多个数据节点（工作者）。namenode 管理文件系统命名空间。它维护文件系统树以及树中所有文件和目录的元数据。此信息以两个文件的形式持久存储在本地磁盘上：命名空间映像和编辑日志。namenode 还知道给定文件的所有块所

在的 datanode; 但是，它不会持久存储块位置，因为此信息是在系统启动时从数据节点重建的。

文件和目录由NameInode在inode上表示。Inode记录属性，如权限，修改和访问时间，命名空间和磁盘空间配额。文件内容被拆分为大块（通常为128兆字节，但用户可以逐个文件选择），并且文件的每个块都在多个DataNode上独立复制。NameNode维护命名空间树以及块到DataNode的映射。

如果没有namenode，则无法使用文件系统。事实上，如果运行namenode的机器被删除，文件系统上的所有文件都将丢失，因为无法知道如何从数据节点上的块重建文件。因此，重要的是使namenode适应失败。Namenode的主要功能有以下几种：

（i）、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。

（ii）、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。

（iii）、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。

（iv）、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。

以下做详细描述：

（i）文件系统目标树的管理

HDFS 文件系统的命名空间在Namenode 的内存中是以一棵树的结构来存储的。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。

图 3-3 FSDirectory部分类图

HDFS 会将命名空间保存到Namenode 的本地文件系统上一个叫fsimage C 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fsi mage 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。如下图所示， 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）,如图3-3为其类图的部分截图。

HDFS 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IN ode 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。INodeDirectory 抽象了HDFS文件系统中的目录，目录是文件系统中的一个虚拟容器， 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。如图3-4为其类图：

图 3-4 INodeDirectory 类图

（ii）数据块以及数据节点管理管理

Namenode 通过BlocksMap 维护了数据块副本与数据节点之间的对应关系。在HDFS 运行时，一个数据块副本可以存在很多不同的状态，系统发生异常或者用户执行特定操作时都会对数据块副本状态产生影响。本节就介绍Namenode 上的数据块副本状态，以及保存这些不同状态数据块副本的数据结构。在HDFS 源码中并没有使用一个枚举类给出数据块副本的状态定义以及状态之间的转移操作，而是通过BlockManager 中的数据结构、不同的数据块副本类（例如BlockUnderC onstruction 和Block）以及副本所在Datanode 的状态（Datanode 处于撤销状态）来记录数据块副本的状态。如图3-5所示为BlockManager类图

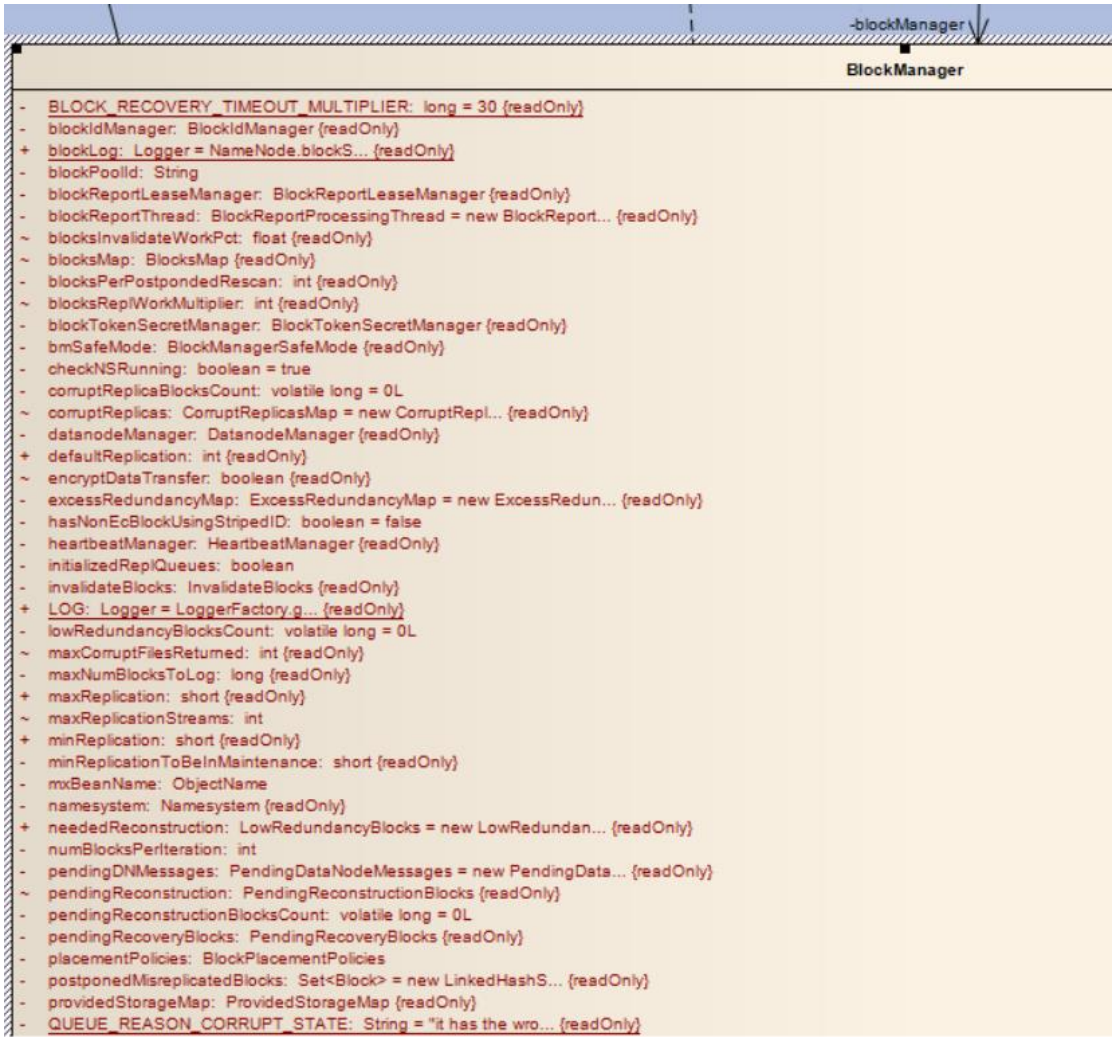

图 3-5 BlockManager 类图

（iii）租约管理

在HDFS 中，客户端写文件时需要先从租约管理器（LeaseManager ）申请一个租约，成功申请租约之后客户端就成为了租约持有者， 也就拥有了对该HDFS 文件的独占权限，其他客户端在该租约有效时无法打开这个HDFS 文件进行操作。Namenode 的租约管理器保存了HDFS 文件与租约、租约与租约持有者的对应关系， 租约管理器还会定期检查它维护的所有租约是否过期。租约管理器会强制收回过期的租约，所以租约持有者需要定期更新租约(renew），维护对该文件的独占锁定。当客户端完成了对文件的写操作，关闭文件时，必须在租约管理器中释放租约。

如当客户端创建文件和追加写文件时，FSNamesystem.startFileIntemal()以及appendFileintemal()方法都会调用LeaseManager.addLease()为该客户端在HDFS 文件上添加一个租约。addLease()方法有两个参数，其中holder 参数保存租约的持有者信息， src 参数则保存创建或者追加写文件的路径，这两个参数分别对应于ClientProtocol.create()或者append()方法中的clientName 和src 参数。addLease()方法的实现也非常简单，先是通过getLease()方法构造租约，然后在LeaseManager 定义的保存租约的数据结构中添加这个租约的信息。代码实例如下。

```
synchronized Lease addLease (String holder, String src) {
Lease lease= getLease(holder）;
      if (lease == null) {
            lease= new Lease(holder); ／／构造Lease 对象
            leases . put(holder, lease); ／／在LeaseManager.leases 字段中添加Lease 对象
            sortedLeases.add(lease); ／／在LeaseManager.sortedLease 字段中添加Lease 对象
) else {
      renewLease(lease);
      }
sorte dLeasesByPa th. put ( src, lease) ; ／／在LeaseManager . sortedLeasesByPath 字段中添加
      Lease 对象
      lease.paths.add(src);
      return lease;}
```

（iv）缓存管理

HDFS 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。

添加集中式缓存功能的HDFS 集群具有以下显著的优势。

（1）、阻止了频繁使用的数据从内存中清除。

（2）、因为集中式缓存是由Namenode 统一管理的，所以HDFS 客户端可以根据数据块的缓存情况调度任务，从而提高了数据块的读性能。

（3）、数据块被Datanode 缓存后，客户端就可以使用一个新的更高效的零拷贝机制读取数据块。因为数据块被缓存时已经执行了校验操作，所以使用零拷贝读取数据块的客户端不会有读取开销。

图 3-6 CacheManager 类图

CacheManager 类是Namenode 管理集中式缓存的核心组件，它管理着分布在HDFS 集群中Datanode 上的所有缓存数据块，同时负责响应“ hdfs cach eadmin ”命令或者HDFS API 发送的缓存管理命令。通过图3-6所示类图可以看出， CacheManager 定义了以下字段。

directivesByld ：以缓存指令id 为key ，保存所有的缓存指令。

directivesByPath ：以路径为key ，保存所有的缓存指令。

cachePools ：以缓存池名称为key ，保存所有的缓存池。

monitor: CacheReplicationMonitor 对象，负责扫描命名空间和活跃的缓存指令，以确定需要缓存或删除缓存的数据块，并向Datanode 分配缓存任务。

3.1.3、DataNode

Datanode是文件系统的主力，它以存储数据块（Block）的形式保存HDFS文件，同时Datanode 还会响应HDFS客户端读、写数据块的请求。Datanode会周期性地向Namenode 上报心跳信息、数据块汇报信息（BlockReport）、缓存数据块汇报信息（ CacheReport ）以及增量数据块块汇报信息。Nam enode 会根据块汇报的内容，修改Namenode 的命名空间（Namesp ace），同时向Datanode返回名字节点指令。Datanode 会响应Namenode 返回的名字节点指令，如创建、删除和复制数据块指令等。下面结合类图对Datanode的结构进行介绍。

（i）、Datanode存储

在Datanode的源码中，DataStorage类提供了管理与组织Datanode磁盘存储空间，以及管理Datanode 存储空间生命周期（包括支持升级、回滚、提交等操作，维护存储空间的状态机等〉等功能。Datanode 最重要的功能就是管理磁盘上存储的HDFS 数据块。Datanode 将这个管理功能切分为两个部分：①管理与组织磁盘存储目录（由dfs.data .dir 指定），如current、previous、detach、tmp 等，这个功能由DataStorage 类实现： ②管理与组织数据块及其元数据文件，这个功能主要由FsDatasetlmpl 相关类实现。

数据节点存储DataStorage是抽象类Storage的子类，而抽象类Storage又继承自StorageInfo。和DataStorage同级的FSImage类主要用于组织NameNode

的磁盘数据，FSImage的子类CheckpointStorage，则管理SecondaryNameNode使用的文件结构。

StorageInfo包含了三个重要的共有的属性，包括HDFS存储系统信息结构版本号layoutVersion、存储系统标识namespaceID和存储系统创建时间cTime。在StorageInfo的基础上，抽象类Storage可以管理多个目录，存储目录的实现类为StorageDirectory，它是Storage的内部类，提供了在存储目录上的一些通用操作。

DataStorage扩展了Storage，专注于数据节点存储空间的生命周期管理，其代码可以分为两个部分：升级相关和升级无关的。在数据节点第一次启动时，会调用DataStorage.format()创建存储目录结构，通过删除原有目录及数据，并重新创建目录，然后将VERSION文件的属性赋值，并持久化到磁盘中。

如图3-7所示是DataStorage的类图：

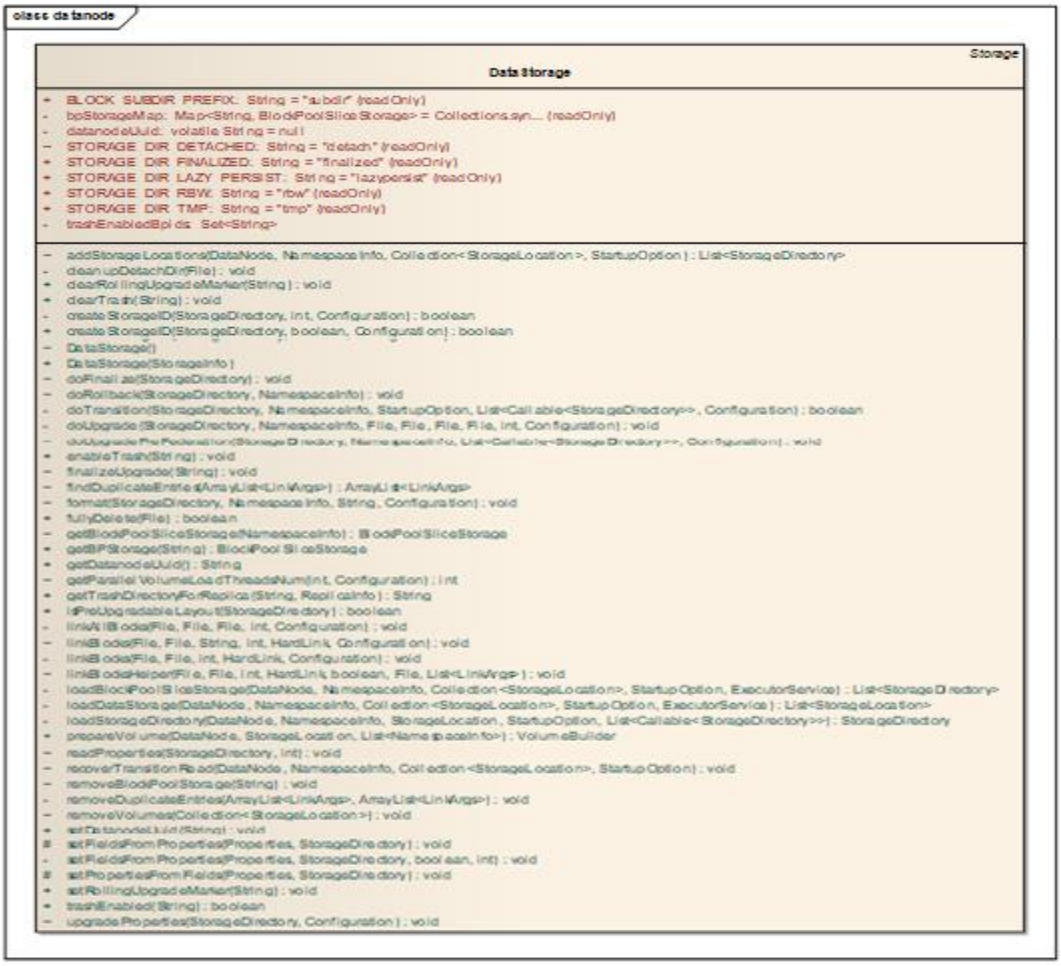

图 3-7 DataStorage 类图

（ii）、管理与操作数据块

Datanode 可以配置多个存储目录保存数据块文件（Datanode 的多个存储目录存储的数据块并不相同，并且不同的存储目录可以是异构的，这样的设计可以提高数据块IO的吞吐率），所以Datanode 将底层数据块的管理抽象为多个层次，并定义不同的类来实现各个层次上数据块的管理。如图3-8所示。

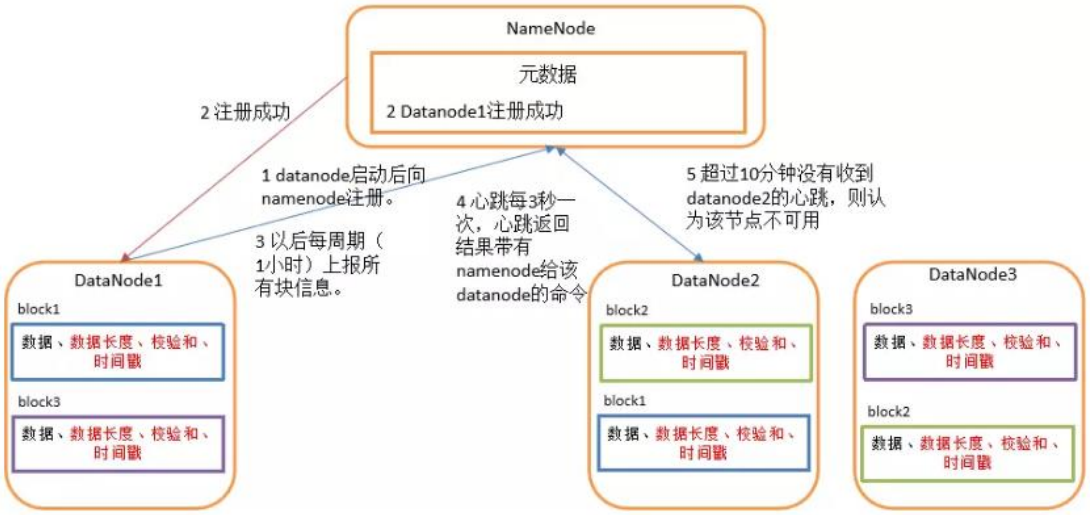

图 3-8　NameNode与DataNode交互图

由图 3-8 可知，它们在被告知（通过客户端或名称节点）时存储和检索块，并且它们定期向名称节点报告它们存储的块列表。DataNode 上的每个块副本由本地本机文件系统中的两个文件表示。第一个文件包含数据本身，第二个文件记录块的元数据，包括数据的校验和和生成标记。数据文件的大小等于块的实际长度，并且不需要额外的空间来将其四舍五入到传统文件系统中的标称块大小。因此，如果块是半满的，则它只需要本地驱动器上整个块的一半空间。

在启动期间，每个 DataNode 都连接到 NameNode 并执行握手。握手的目的是验证命名空间 ID 和 DataNode 的软件版本。如果其中任何一个与 NameNode 的匹配，则 DataNode 会自动关闭。握手后，DataNode 向 NameNode 注册。DataNode 持久存储其唯一的存储 ID。存储 ID 是 DataNode 的内部标符，即使使用不同的 IP 地址或端口重新启动，也可以识别它。存储 ID 在第一次向 NameNode 注册时分配给 DataNode，之后永远不会更改。

DataNode 通过发送块报告来识别其拥有的块副本到 NameNode。块报告包含块 ID，生成戳记以及服务器托管的每个块副本的长度。在 DataNode 注册后立即发送第一个块报告。每小时发送一次后续块报告，并为 NameNode 提供块副本在群集上的位置的最新视图。

在正常操作期间，DataNode 将心跳发送到 NameNode 以确认 DataNode 正在运行，并且它所托管的块副本可用。默认心跳间隔为 3 秒。如果 NameNode 在十分钟内没有从 DataNode 接收到心跳，则 NameNode 会认为 DataNode 已停止服务，并且该 DataNode 托管的块副本不可用。然后，NameNode 计划在其他 DataNode 上创建这些块的新副本。来自 DataNode 的心跳还包含有关总存储容量，正在使用的存储容量以及当前正在进行的数据传输的数量的信息。这些统计信息用于 NameNode 的块分配和负载平衡决策。

NameNode 不直接向 DataNode 发送请求。它使用对心跳的回复来向 DataNode 发送指令。这些指令包括将块复制到其他节点，删除本地块副本，重新注册和发送立即块报告以及关闭节点的命令。

（iii）、流式接口

JDK 基本套接字提供了 java.net.Socket 和 java.net.ServerSocket 类，在服务器端构造 ServerSocket 对象，并将该对象绑定到某空闲端口上，然后调用 accept()方法监听此端口的入站连接。当客户端连接到服务器时，accept()方法会返回一个 Socket 对象，服务器使用该 Socket 对象与客户端进行交互，直到一方关闭为止

而以上的 ServerSocket 和 Socket 就对应着流式接口中的 DataXceiverServer 和 DataXceiver，它们分别实现了对 ServerSocket 和 Socket 的封装。并采用了一客户一线程（DataXceiver）的方式，满足了数据节点流式接口批量读写数据、高吞吐量的特殊要求。

DataXceiverServer 和 DataXceiver 两个类主要用来是实现流式接口，DataXceiverServer 包含的成员变量有 childSockets 和 maxXceiverCount,和 ss，其中 childSockets 中包含了所有打开的用于数据传输的 Socket，这些 Socket 由 DataXceiver 对象进行维护，maxXceiverCount：数据节点流式接口能够支持的

最大客户端数，由配置项${dfs.datanode.max.xceivers}指定，默认值 256。在一个繁忙的集群中，应该适当提高该值。ss 用于监听客户端入站连接的 ServerSocket 对象。

DataXceiverServer 在 run()方法中实现了 ServerSocket 的 accept()循环，也就是说 DataXceiverServer 用于监听来自客户端或其他 DataNode 的请求。每当阻塞方法 accept()返回新的请求时，DataXceiverServer 会创建一个新的 DataXceiver 线程对象，实现一对一的客户服务。其类图如图 3-9 所示。

代码如下：

```
public void run() {
    while (datanode.shouldRun) {
        try {
            Socket s = ss.accept();
            s.setTcpNoDelay(true);
            new Daemon(datanode.threadGroup,
                    new DataXceiver(s, datanode, this)).start();
        } catch (SocketTimeoutException ignored) {
            // wake up to see if should continue to run
        } catch (AsynchronousCloseException ace) {
            datanode.shouldRun = false;
        } catch (IOException ie) {
        } catch (Throwable te) {
            datanode.shouldRun = false;
        }
    }
    try {
        ss.close();
    } catch (IOException ie) {
    }
}
```

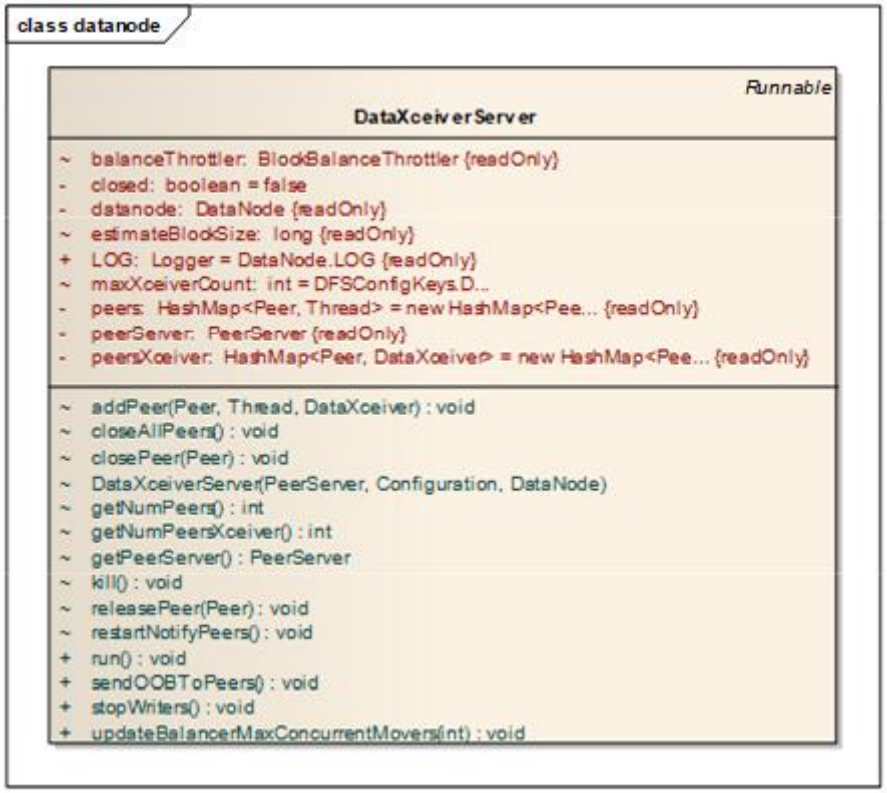

图 3-9 DataXceiverServer 类图

DataXceiver 中包含成员变量 ss，用来实现与客户端一对一交互的 Socket 对象和 DataXceiver.run()方法，用来实现了管理每个实际 Socket 请求的输入输出数据流，其执行流程如下：

（1）、打开输入流，并读取数据流中请求帧的第一个字段进行版本号检查；

（2）、检查该请求是否超出数据节点的支撑能力，以确保数据节点的服务质量；

（3）、读入请求码，并根据请求码调用相应的方法；

3.2 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,所以需要一套节点间的通信交互机制。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。

3.2.1 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。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: 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。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 。

其他接口：主要包括安全相关接口（ RefreshAuthorizationPolicyProtocol 、RefreshUserMappingsProtocol ）、HA 相关接口（ HAServiceProtocol ）等。

3.2.2 RPC 框架

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。

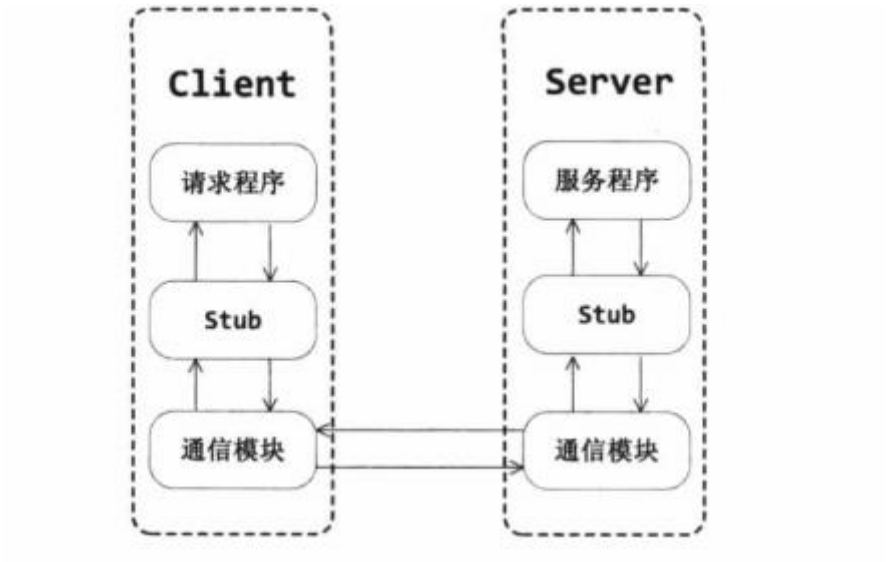

图 3-10　RPC 框架结构图

图 3-10 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。

3.2.3 RPC 的实现

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。

（i）、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。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。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.Sever 类，然后调用 RPC.Server.start()方法启动 Sever 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.Sever 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。

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。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。

```
public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol ,
```

```
                    long clientVersiori ,
throws IOException {
                    InetSocketAddress addr ,
                    UserGroupinformation ticket ,
                    Conf 工 guration conf ,
                    SocketFactory factory ,
                    int rpcTimeout ,
                    RetryPolicy connectionRetryPolicy ,
                    AtomicBoolean fallbackToSimpleAuth)
                    return getProtocolEngine(protoco l , conf) .getProxy(protocol ,
                    clientVersion ,
                    addr , ticket , conf, factory , rpcT 工 meout ,
                    connectionRetryPolicy ,
                    fallbackToSimpleAuth) ;
}
```

（ii）、Client 类实现

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 3-11 所示，分为以下几步。

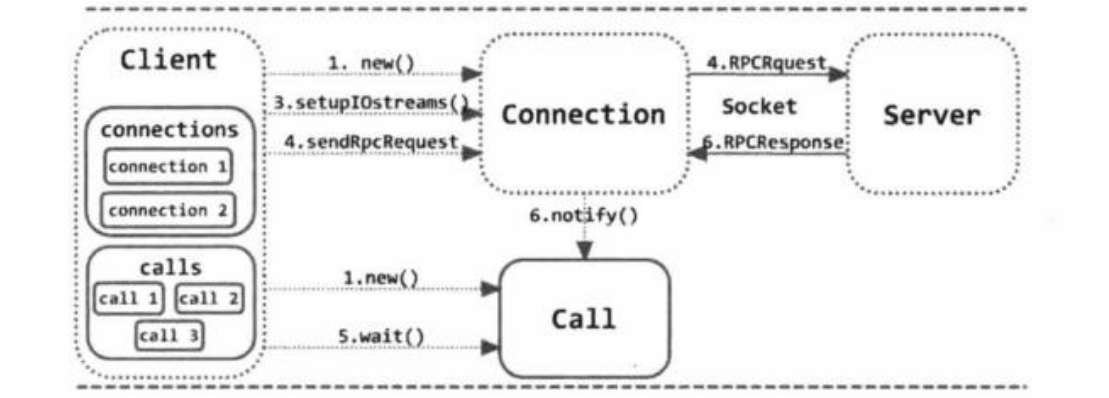

图 3-11 Client.call（）方法执行流图

• Client.call（）方法将 RPC 请求封装成一个 Call 对象，Call 对象中保存 RPC 调用的完成标志、返回值信息以及异常信息；随后，Client.cal（）方法会创建一个 Connection 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 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 f allbackToSimpleAuth) throws I OException {
   / / 构造 Call 对象
   final Call call= createCall(rpcKind, rpcRequest);
   / / 构造 Conne c tion 对象
   Connection connection = getConnection(remoteid, call , serviceClass ,
         fallbackToSimpleAuth) ;
      try {
         connection . sendRpcRequest(call) ;
```

```
} catch ( RejectedExecot.ioRException e) (
/ / 发送 RPC 请求
    throw new IOException (" connection has been closed", e);
) catch (InterruptedException e) {
    Thread . currentThread() .interrupt();
    throw new IOException (e) ;
    boolean interrupted = false ;
    synchro 口 l zed (call) {
whi.le ( 1cal1 . done) {
    try {
        call.wait() ;
        ) catch (InterruptedException ie) {
    interrupted = true ;
if (interrupted) {
    Thread.currentThread() . interrupt() ;
    / / 等待 RPC 响应
if（ cal l. error ! = null) { / / 友送线程被唤醒，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 = coŋnection . getRemoteAddress();
    throw NetUtils . wrapException(address . getHostName() ,
    address.getPort() ,
    ) else {
        NetUtils . getHostname () ,0
    call . error);
```

```
/ / 服务器成功发回响应信息 ， 返回 RP C 响应
        return call . getRpcResponse() ;
        }
      }
  }
```

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 Clientl.call（）方法的 线程，而接收响应线程则是 call（）启动的 Connection 线程。这里使用了 Call 类。举个例子，线程 l 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 l 被唤醒后，会取出 Call.rpcResponse 字段中记录的 Server 发田的响应信息并返回。

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

```
        this.done = true;  / / 设置 done 字段为 tr.ue ，表明 当前请求
        notify();  / / 唤醒在 Call 对象上等待的线程
    }
```

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。

```
  private InetSocketAddress 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  工  nt 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
```

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 CConnectionid -> 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 Sever 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 Ca ll 添加到这个 Connection 的请求队列 calls 当中，然后调用 setupIOs 位 earns（）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 工 s stopped” );
Connection connection ;
do {
synchronized (connections) {
／／首先尝试从 Client .connecti ons 队列 中获取 Connection 对象
connection= connections . get(remoteid) ;
工 f (connection == null) {
第 2 章 Hadoop RPC
／／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
) while (! connection .addCall(call));
／／ 调用 setupIOstrea ms （）方法，初始化 Connection 对象并获取 IO 流
connection.setupIOstreams(fallbackToSimpleAuth) ;
return connection ;
}
```

（iii）、 Sever 类实现

Sever 类部分结构图如图 3-12 所示。

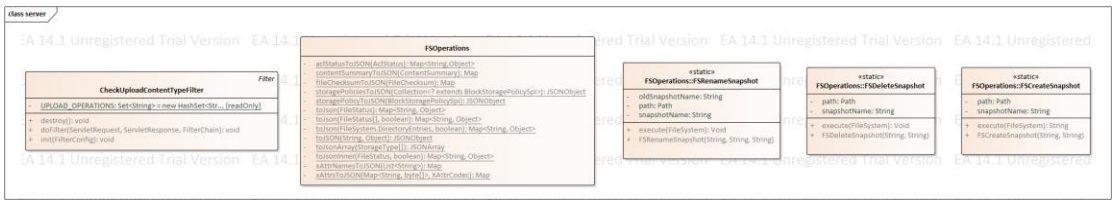

图 3-12　Sever 类结构图

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。

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 3-13 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 。

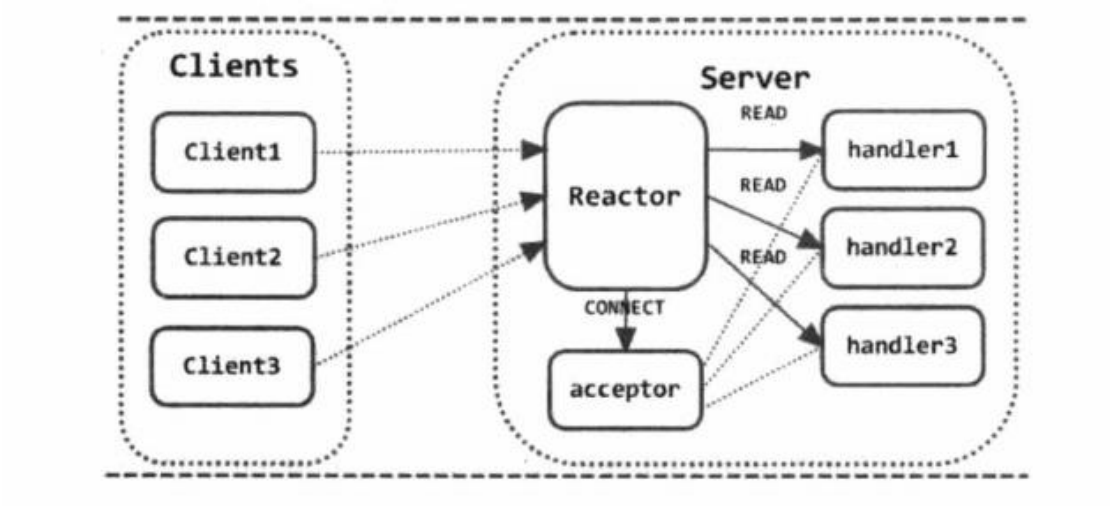

图 3-13 基于 Reactor 模式的网络服务器设计

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 Socekt 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 3-14 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 。

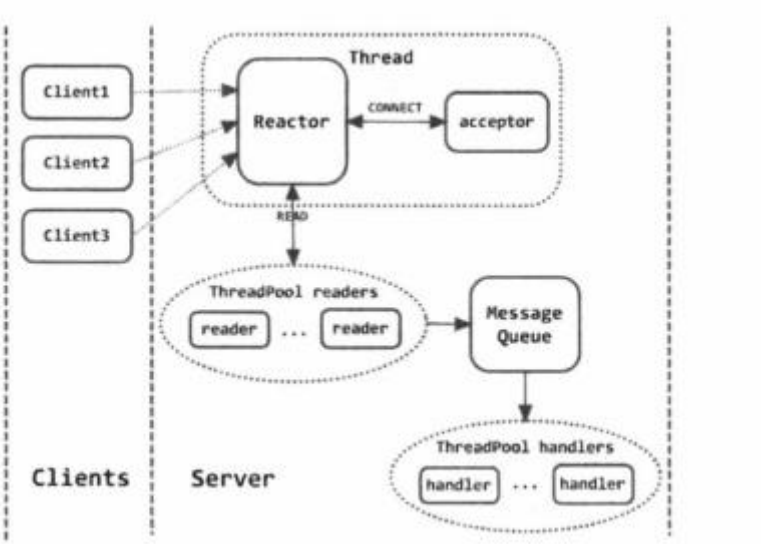

图 3-14 基于多线程 Reactor 模式的网络服务器结构

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 线程池中选择一个 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 = i ter . 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 ， 妥特别 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 doAcceptO 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 Se 凹 er 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 = (Servers 。 cketChannel) key . channel();
  SocketChannel channel ;
  while ( (channel = server . accept () ) != null) {
      channel . configureBlocking(false);
      channel . socket() . setTcpNoDelay(tcpNoDelay) ;
      / / 从 readers 线希呈池中取出一个 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。

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 Sever 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 synchron 工 zed 飞 roid 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(lOOO);
            / / 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> i ter ~ readSelector . selectedKeys () .
iterator() ;
            while (iter . hasNext()) {
                key = i ter. next() ;
                iter.remove() ;
                if (key . isVal 工 d()){
                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()) ;
        }
     }
```

内部类-----Connection (done)

Connection 类维护了 Sever 与 Client 之间的 Socket 连接。Reader 线程会调用 read.AndProcess（）方法从 IO 流中读取一个 RPC 请求。read.AndProcess （）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。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 Sever 异常信息的响应）。 processOneRpc（）方法的代码如下：

```
private void processOneRpc (byte [ J 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 call Id
    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 WrappedRpcServerExcepti on (
            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 = Ref lectionUt 工 ls . newinstance(rpcRequestClass , conf) ;
    rpcRequest.readFields(dis);
    ) catch (Throwable t) {  / / 工 nclu des 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() ;
        }
```

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 call Queue 中取出 待处理的 Call 对象，然后调用 Sever. 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 call Queue 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 . connect 工 on.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(exCept 工 onHdr)) {
        error = error substr 工 ng(exceptionHdr . length());
        CurCall .set(null);
        synchronized (call.connection . responseQueue) {
      / / 构造 RPC 响应，如采调用正常就返回结采，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) {
      //…
    }
}
```

3.3 HDFS 客户端

HDFS 客户端作为与人进行交互的终端，HDFS 为客户端提供了三个客户端接口，分别为：DistributedFileSystem、FsShell 和 DFSAdmin。其中 DistributedFileSystem 为用户开发基于 HDFS 的应用程序提供了 API；FsShell 工具使用户可以通过 HDFS Shell 命令执行常见的文件系统操作，比如创建文件、删除文件等；DFSAdmin 可以帮助系统管理员管理 HDFS，比如执行升级、管理安全模式等操作。

上文提到的三个接口在作用时都要直接或间接的获取 DFSClient 对象的引用，然后通过 DFSClient 中的接口方法对 HDFS 进行管理和操作。所以 HDFS 与客户端的交互都要经过 DFSClient 类进行交互，因此 DFSClient 在有关客户端的相关操作中具有比较重要的地位，下面就 DFSClient 的实现做详细分析。

3.3.1 HDFS 客户端文件层次结构

根据源码分析，在客户端模块里，主要的类结构层次如 3-15 图所示：

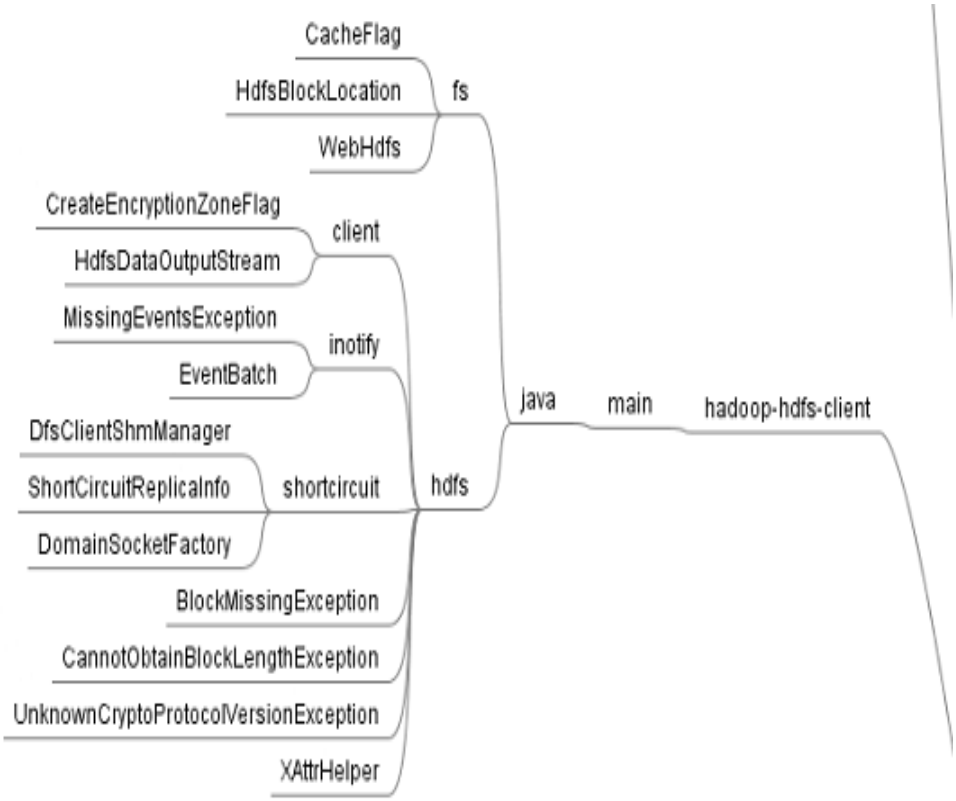

图 3-15 Client 文件层次结构图

其中，各个层次的解释如下：

Fs:此包内为文件系统，包含一般文件的一些相关操作。

Hdfs:此包为分布式文件系统，包含了客户端模块关于分布式文件系统操作的相关类。

在两个包下分别存放着很多类，现就其中重点的类进行介绍：

HdfsDataInputStream.java：是进行分布式文件写入文件的相关操作，符合写操作的相关设计。

HdfsDataOutputStream.java：是进行分布式文件系统输出文件的相关操作，符合读操作的相关设计。

DFSClient.java：是客户端分布式文件系统中比较重要的一个类，外界与系统进行相关交互都要经过该类才能实现相关操作。

BlockReader.java：块读取类，该类将文件分为固定的大小，并按照块进行读取。

DFSOutputStream.java：该类是在 DFSClient 之前，控制客户端与外界进行交互的文件组织方式与传输方式的类。

3.3.2 DFSOutputstream.java 逆向工程

通过逆向工程软件 Enterprise Architect 对 hadoop 源码进行分析获得如下 DFSOut putstream 有关的类图。如图 3-16 所示，为 DFSoutputstream 的类图，从图中可以看到该类存在较多的属性与方法，涉及客户端与外界进行交互的输出流的相关操作。

图 3-16 DFSOutputstream 类图

3.3.3 DFSClient 实现

DFSClient 是一个真正实现了分布式文件系统客户端功能的类，是用户使用 HDFS 各项功能的起点。DFSClient 会连接到 HDFS ，对外提供管理文件或目

录、读写文件以及管理与配置 HDFS 系统等功能。

对于管理文件或目录以及管理与配置 HDFS 系统这两个功能， DFSClient 并不需要与 Datanode 交互，而是直接通过远程接口 ClientProtocol 调用 Namenode 提供的服务即可。而对于文件读写功能，DFSClient 除了需要调用 ClientProtocol 与 Namenode 交互外，还需要通过流式接口 DataTransferProtocol 与 Datanode 交互传输数据。

DFSClient 对外提供了很多接口，大致可以分为以下几类：

（i）、DFSClient 的构造方法

（ii）、管理与配置文件系统相关的方法

（iii）、操作 HDFS 文件与目录的方法

（iv）、读写 HDFS 文件的方法

DFSClient 接口代码分析

（i)、DFSClient 构造方法

通过分析源码，可以看出在 DFSClient 类中共设置了五种构造方法。这些构造方法主要实现了两个功能，分别如下所示：

读入配置文件，并初始化以下成员变量。

conf: HDFS 配置信息。

stats: Client 状态统计信息，包括 Client 读、写字节数等。

dtpReplaceDatanodeOnFailure ：当 Client 读写数据时， 如果 Datanode 出现故障，是否进行 Datanode 替换的策略。

locallnterfaceAddrs ：本地接口地址。

readahead ：预读取字节数。

readDropBehind ：读取数据后，是否立即从操作系统缓冲区中删除。

writeDropBehind ：写数据后，是否立即从操作系统缓冲区中删除。

hedgedReadThresholdMillis: hedgedReadThresholdMillis 字段保存了触发“ hedgedread“机制的时长。“ hedgedread“模式是 Hadoop 2.4 中引入的新特性， 当 Client 发现一个数据块读取操作太慢时（读取时长超过 hedgedReadThresho ldMillis ）,那么 Client 会启动另一个并发操作读取数据块的另一个副本，之后 Client 会返回先完成读取副本的数据。

获取 Namenode 的 RPCProxy 引用,供 DFSClient 远程调用 NamenodeRPC 方法。

构造方法代码示例

```
 @Deprecated
  public DFSClient(Configuration conf) throws IOException {
    this(DFSUtilClient.getNNAddress(conf), conf);
  }

  public DFSClient(InetSocketAddress address, Configuration conf)
      throws IOException {
    this(DFSUtilClient.getNNUri(address), conf);
  }

  /**
   * Same as this(nameNodeUri, conf, null);
   * @see #DFSClient(URI, Configuration, FileSystem.Statistics)
   */
  public DFSClient(URI nameNodeUri, Configuration conf) throws
IOException {
    this(nameNodeUri, conf, null);
  }

  /**
   * Same as this(nameNodeUri, null, conf, stats);
   * @see #DFSClient(URI, ClientProtocol, Configuration,
FileSystem.Statistics)
   */
  public DFSClient(URI nameNodeUri, Configuration conf,
      FileSystem.Statistics stats) throws IOException {
    this(nameNodeUri, null, conf, stats);
```

```
  }
@VisibleForTesting
  public DFSClient(URI nameNodeUri, ClientProtocol rpcNamenode,
      Configuration conf, FileSystem.Statistics stats) throws
IOException {
    // Copy only the required DFSClient configuration
```

（ii）、管理与配置文件系统相关的方法

HDFS 管理员通过 DFSAdmin 工具管理与配置 HDFS ，DFSAdmin 也是通过获取 DistributedFileSystem 对象的引用， 然后进一步调用 DFSClient 类提供的方法来执行管理与配置操作的。此处以其中的一个方法 DFSClient.rollEdits()为例进行分析。如图 3-17 所示为 rollEdits()方法的调用序列图。该方法主要是用来滚动编辑日志的，从序列图中可以看到，DFSAdmin 可以调用该方法进行自我的日志编辑，另外若需要对 Namenode 进行操作，需要通过获取 DistributedFileSystem 对象的引用，然后才能通过调用 DFSClient 提供的方法对 Namenode 进行相关操作。

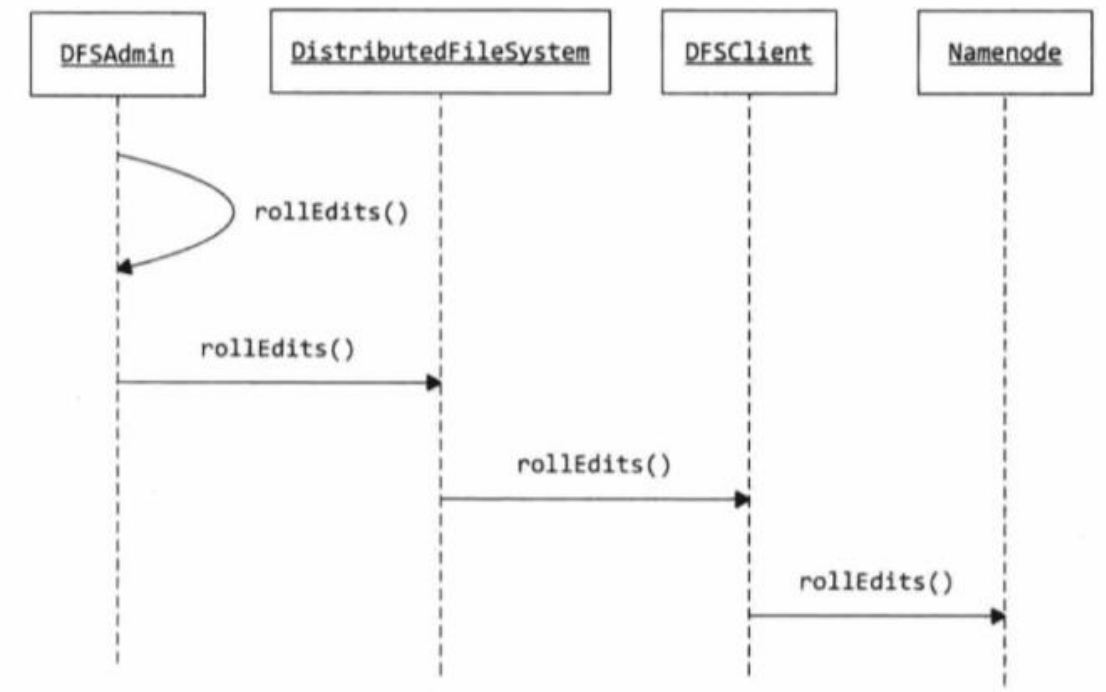

图 3-17 rollEdits()方法的调用序列图

该方法的具体实现如下，从代码中可以看到在进行 Namenode 相关操作时调

用了 namenode.rollEdits()方法，并进行了异常处理。

rollEdits 方法的代码实现

```
    long rollEdits() throws IOException {
        checkOpen();
        try (TraceScope ignored = tracer.newScope("rollEdits")) {
          return namenode.rollEdits();
        } catch (RemoteException re) {
          throw
re.unwrapRemoteException(AccessControlException.class);
        }
      }
```

DFSAdmin 类中管理与配置文件系统的方法流程大多与 roll Edits（）类似，它们直接调用了 DFSClient 对应的方法，再由 DFSClient 调用 ClientProtocol 对应的方法。

（iii）、操作 HDFS 文件与目录的方法

除了管理与配置 HDFS 文件系统外，DFSClient 的另一个重要功能就是操作 HDFS 文件与目录，例如 setPermission（）、rename（）、getFilelnfo（）、delete（）等对文件或目录树的增、删、改、查等操作。

这些方法的实现也比较简单，它们都是首先调用 checkOpen（）检查 DFSClient 的运行情况，然后调用 ClientProtocol 对应的 RPC 方法，触发 Namenode 更改文件系统目录树。

（iv）、读写 HDFS 文件的方法

DFSClient 中剩余的方法主要是对文件读写操作的支持，将在以下几小结重点介绍。

3.3.4 文件读操作与输入流

HDFS 目前实现的读操作有三个层次，分别是网络读、短路读以及零拷贝读，它们的读取效率依次递增。

网络读：网络读是最基本的一种 HDFS 读， DFSClient 和 Datanode 通过建立

Socket 连接传输数据。

短路读：当 DFSClient 和保存目标数据块的 Datanode 在同一个物理节点上时，DFSClient 可以直接打开数据块副本文件读取数据，而不需要 Datanode 进程的转发。

零拷贝读：当DFSClient 和缓存目标数据块的Datanode 在同一个物理节点上时，DFSClient 可以通过零拷贝的方式读取该数据块，大大提高了效率。而且即使在读取过程中该数据块被 Datanode 从缓存中移出了，读取操作也可以退化成本地短路读,非常方便。HdfsDataln putStream.read（）方法就实现了上面描述的层次读取。该方法首先会调用 HasEnhancedByteBufferAccess.read（）方法尝试进行零拷贝读取，如果当前配置不支持零拷贝读取模式，则抛出异常，然后调用 ByteBufferUtil.fallbackRead（）静态方法退
化成短路读或者网络读。HdfsDatalnputStream.read（）方法的调用流程如图 3-18 所示。

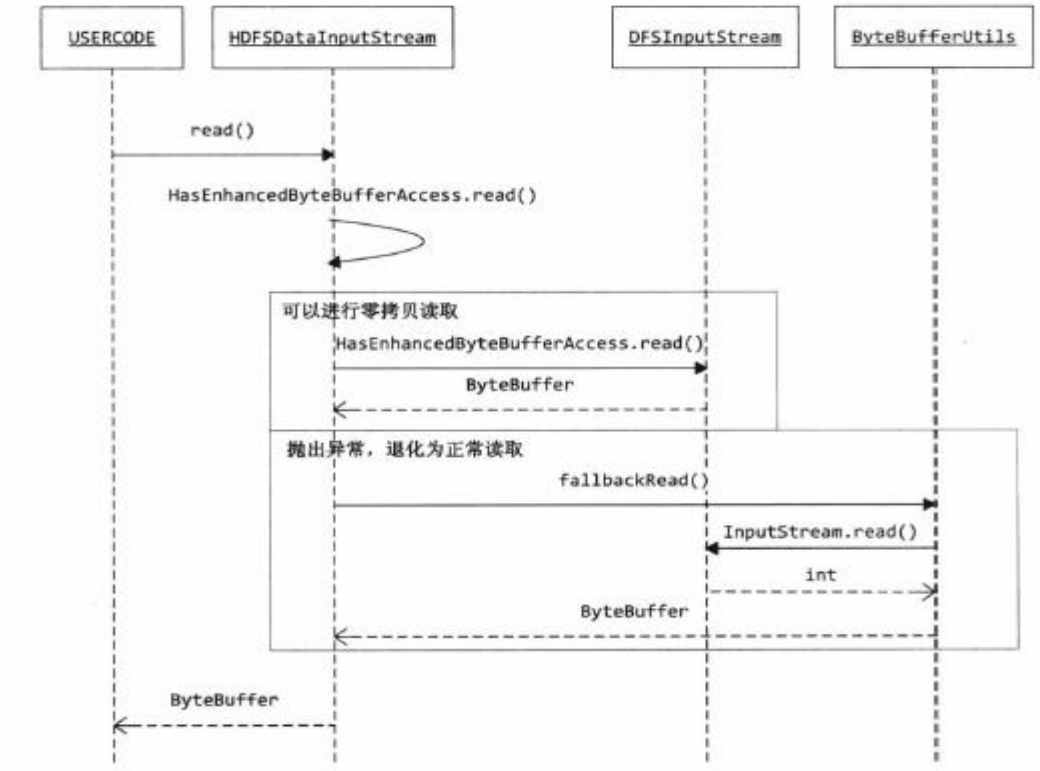

图 3-18 HdfsDatalnputStream.read 方法的调用序列图

HdfsDatalnputStream 实现了 HasEnhancedByteBufferAccess.read（）方法以及 InputStream. read（）方法，这两个方法的实现都是通过调用底层包装类 DFSinputStre am 对应的方法执行的。

HasEnhancedByteBufferAccess.read（）方法定义了零拷贝读取的实现，而 lnputStream.read（）方法则定义了短路读和网络读的实现。

3.3.5 文件写操作与输出流

客户端的写操作需要涉及创建文件、写操作、追加和关闭输出流等操作。

创建文件

客户端在执行文件写操作前， 首先需要调用 DistributedFileSystem . create（）创建一个空的 HDFS 文件， 并且获取这个 HDFS 文件的输出流 HdfsDataOutputStream 对象。成功获取到输出流对象后，客户端就可以在输出流 HdfsDataOutputStream 对象上调用 write()方法执行写操作了。下面结合代码具体分析一下。DistributedFileSystem . create（）方法

用户代码创建一个新文件时，会首先调用 DistributedFileS ystem.create（）方法创建一个空文件，然后通过 create（）方法返回的 HdfsDataOutputStream 输出流写文件。

如下代码所示，这里 DistributedFileSystem.create（）方法直接调用了 DFSClien t.creat 方法，并将返回的 DFSOutputStream 包装成一个 HdfsDataOutputStream 输出流。

```
 public HdfsDataOutputStream create(final Path f,
      final FsPermission permission, final boolean overwrite,
      final int bufferSize, final short replication, final long
blockSize,
      final Progressable progress, final InetSocketAddress[]
favoredNodes)
      throws IOException {
    statistics.incrementWriteOps(1);
    storageStatistics.incrementOpCounter(OpType.CREATE);
    Path absF = fixRelativePart(f);
    return new FileSystemLinkResolver<HdfsDataOutputStream>() {
      @Override
```

```
      public HdfsDataOutputStream doCall(final Path p) throws
IOException {
        final DFSOutputStream out = dfs.create(getPathName(f),
permission,
            overwrite ? EnumSet.of(CreateFlag.CREATE,
CreateFlag.OVERWRITE)
                : EnumSet.of(CreateFlag.CREATE),
            true, replication, blockSize, progress, bufferSize, null,
            favoredNodes);
        return dfs.createWrappedOutputStream(out, statistics);
      }
      @Override
      public HdfsDataOutputStream next(final FileSystem fs, final
Path p)
          throws IOException {
        if (fs instanceof DistributedFileSystem) {
          DistributedFileSystem myDfs = (DistributedFileSystem)fs;
          return myDfs.create(p, permission, overwrite, bufferSize,
replication,
              blockSize, progress, favoredNodes);
        }
        throw new UnsupportedOperationException("Cannot create with"
+
            " favoredNodes through a symlink to a non-
DistributedFileSystem: "
            + f + " -> " + p);
      }
    }.resolve(this, absF);
  }
```

写操作

当用户代码通过 DistributedFileSystem . create（）方法创建了一个新文件，并获取了 DFSOutputStream 输出流对象之后,就可以在输出流对象上调用 write（）方法写数据了。DFSOutpuStream 类结构如图 3-19 所示。

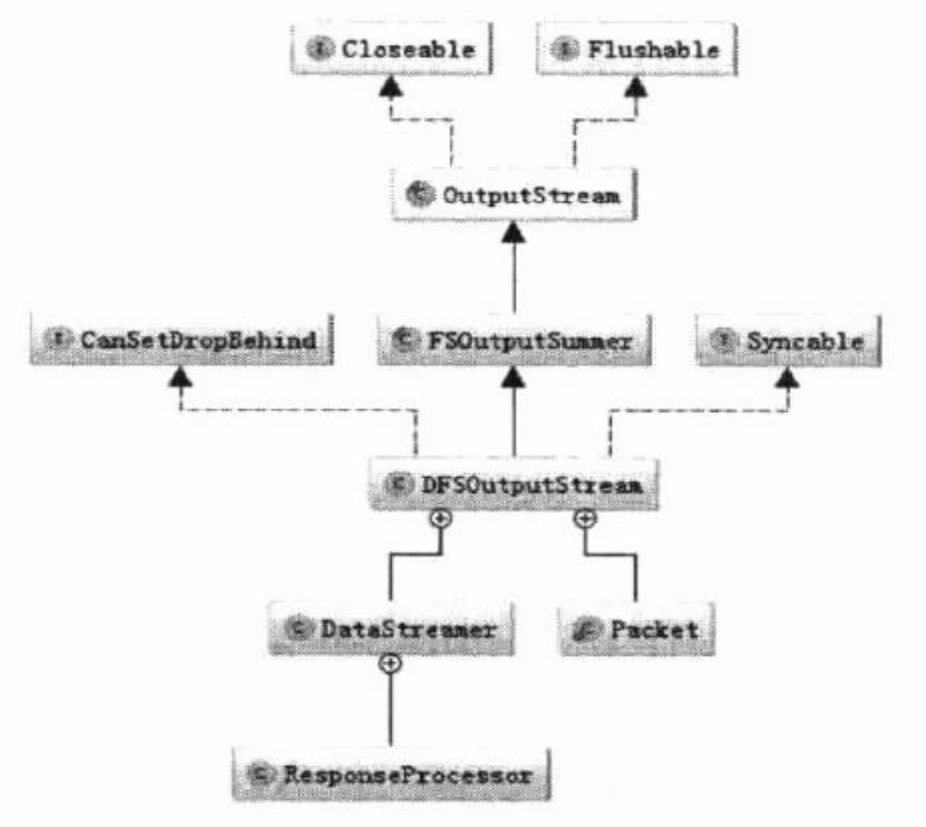

图 3-19 DFSOutputstrem 类的结构（摘自网络）

DFSOutputStrearn 扩展自抽象类 FSOutputSummer, FSOutputSummer 在 OutputStream 的基础上提供了写数据并计算校验和的功能，DFSOutputStream . write（）方法的实现就继承自 FSOutputSummer 类。

DFSOutputStream 中使用 Packet 类来封装一个数据包。每个数据包中都包含若干个校验块，以及校验块对应的校验和。一个完整的数据包结构如图 3-20 所示。首先是数据包包头，记录了数据包的概要属性信息，然后是校验和数据， 最后是校验块数据。Packet 类提供了 writeData（）以及 writeChecksum（）方法向数据块中写入校验块数据以及校验和。

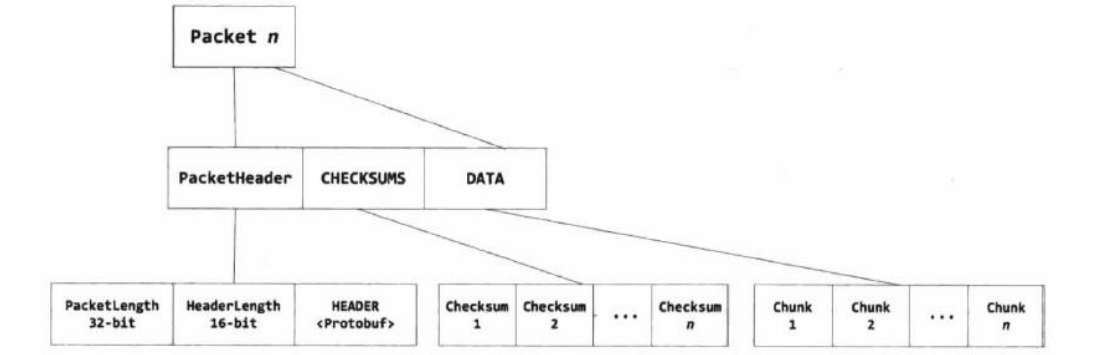

图 3-20 数据包结构

DFSOutputStream. write（）方法继承自 FSOutputSummer. write（）方法，用于向数据流管道中写入指定大小的数据以及校验和， 是客户端写数据操作的入口。write（）方法会循环调用 writel（）方法每次发送一个校验块数据，直到所有数据发送完毕。write（）方法的调用流程如图 3-21 所示。

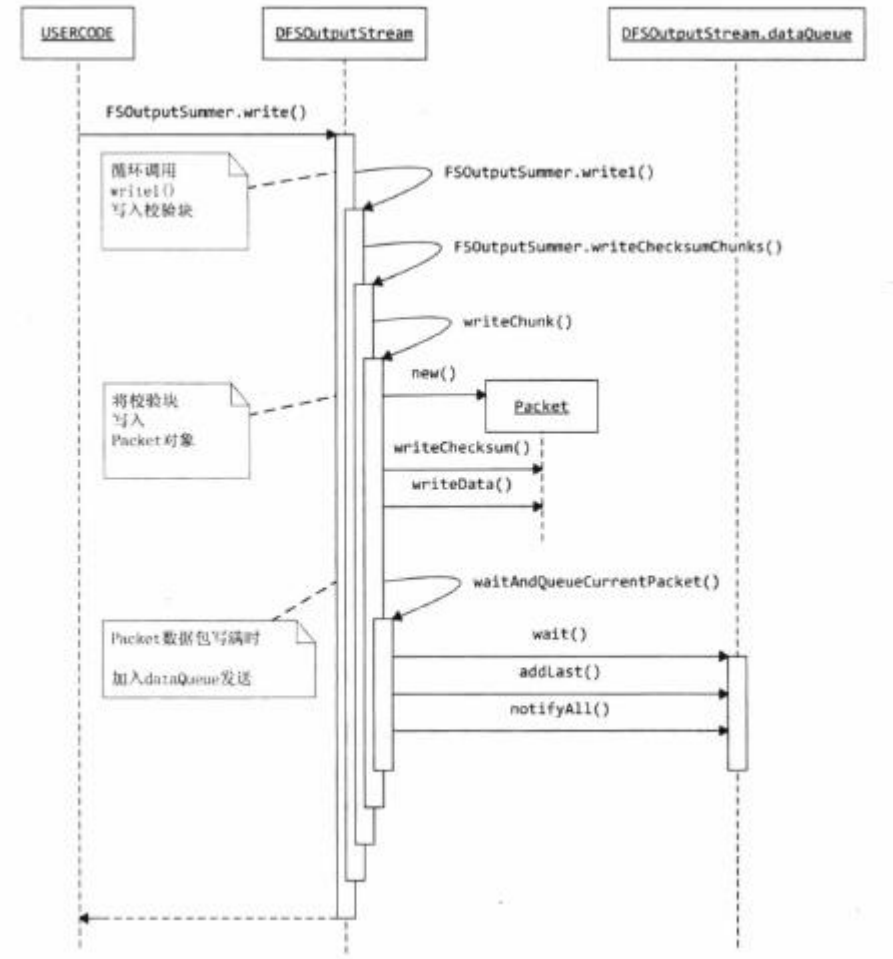

图 3-21 write（）方法的调用流程

追加

客户端除了可以执行写新文件的操作外，还可以打开一个已有的文件并执行追加写操作。

DistributedFileSystem. append()方法就是用于打开一个已有的 HDFS 文件，并获取追加写操作。

如图 3-22 所示，DistributedFileSystem.append（）方法调用了 DFSClient.callAppend（） 方法获取输出流对象。callAppend（）方法首先通过 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 callAppend（）方法会调用 DFSOutputStream.newStreamForAppend()方法创建到文件最后一个数据块的输出流对象。获取文件租约，并将新构建的 DFSOutputStream 包装为 HdfsDataOutputStream 对象，然后返回。

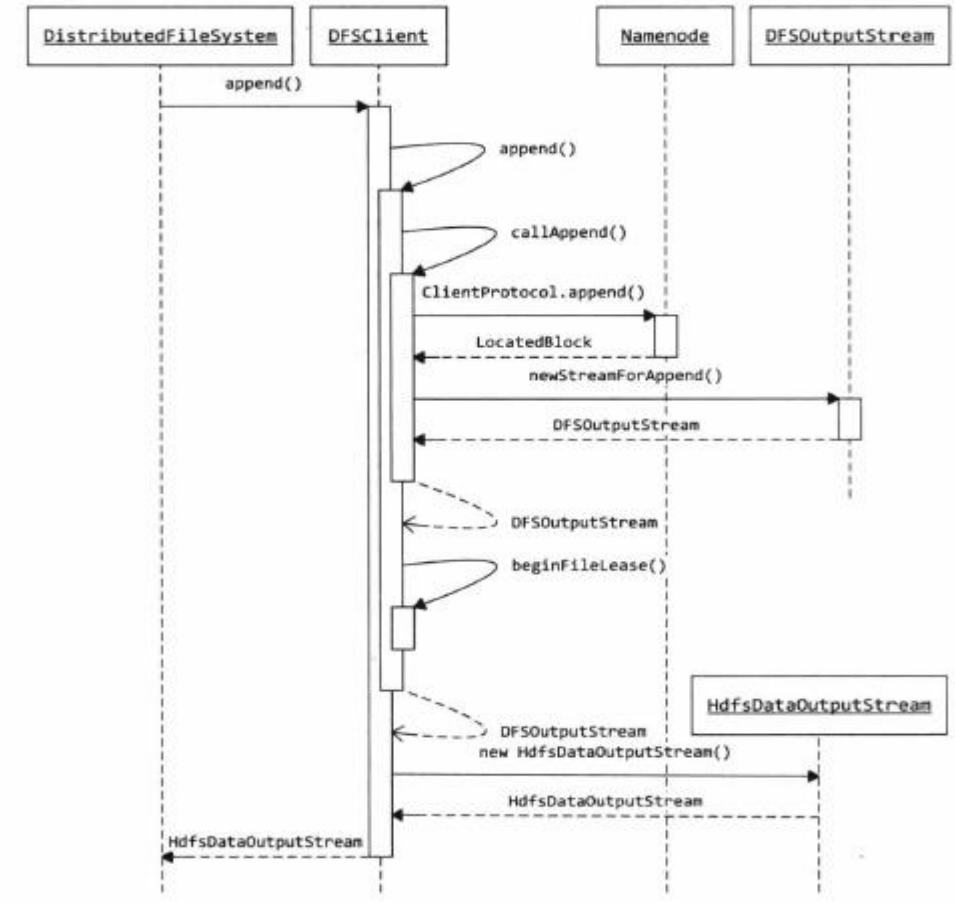

图 3-22 DistributedFileSystem.append（）序列图

关闭输出流

DFSOutputStream . close（） 方法实现了 DFSOutputStream 的关闭操作，

图 3-23 给出了 close()方法的调用流程。

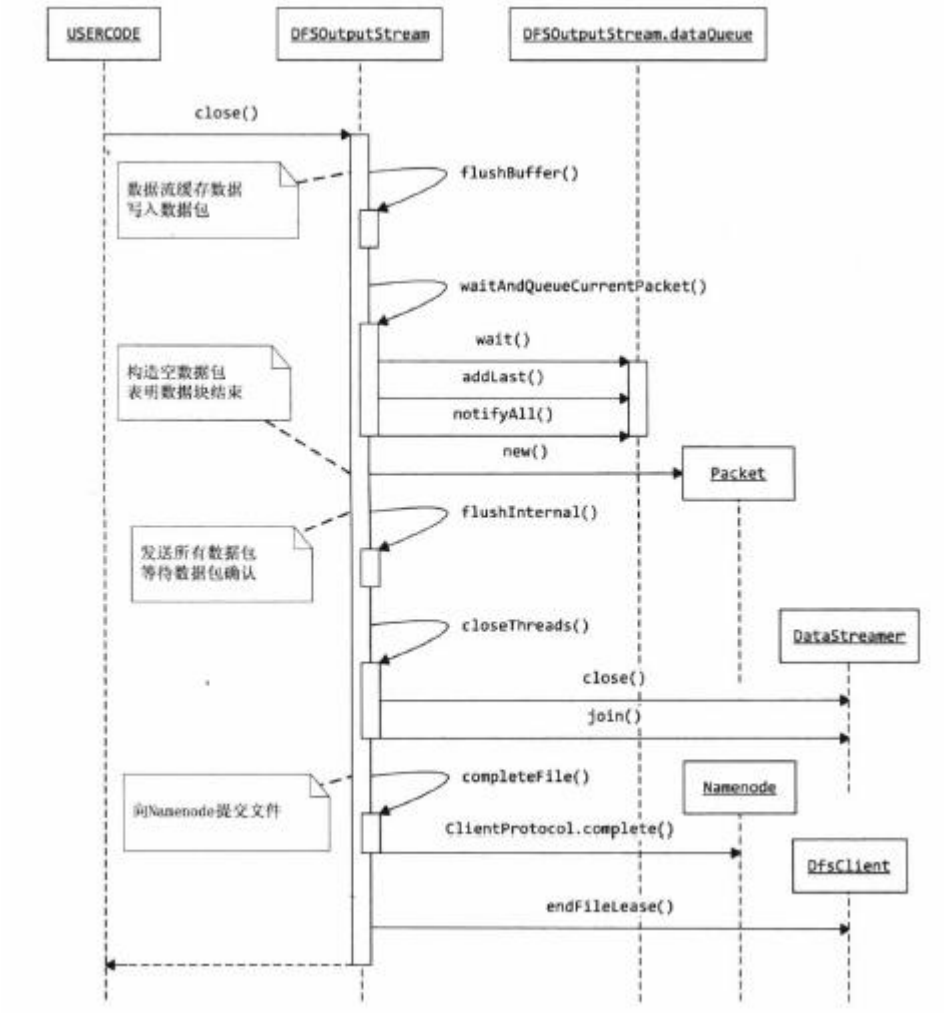

图 3-23 DFSOutputStream.close（）的调用流程图

DFSOutputStream.close（）方法首先调用 flushBuffer（）将输出流中缓存的数据写入数据包，然后将数据流中没有发送的数据包放入 dataQueue 队列中，最后构造一个新的空数据包用于标识数据块已经全部写完。close（）方法调用 flushlnternal （）方法确认所有数据包已经成功地写入数据流管道后，就可以调用 closeThreads （）关闭 DataStreamer 线程了。之后 close （）方法会调用 completeFile （）方法向 Namenode 提交这个文件， completeFile （）方法底层调用了 ClientProtocol.complete （）方法。最后 close（）方法会释放当前文件的租约。

3.4 NFS 模块介绍

NFS 就是 Network File System 的缩写，它最大的功能就是可以通过网络，让不同的机器、不同的操作系统可以共享彼此的文件。如图 3-24 所示为 NFS 的类图。

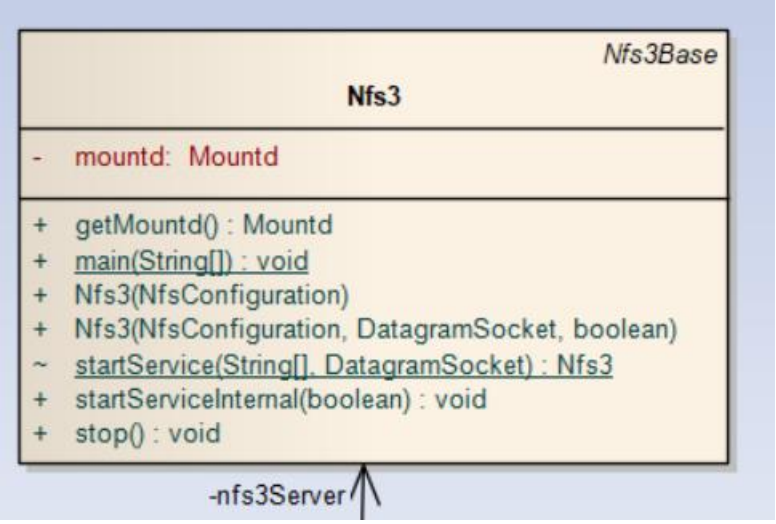

图 3-24 NFS 类图

NFS 服务器可以让 PC 将网络中的 NFS 服务器共享的目录挂载到本地端的文件系统中，而在本地端的系统中来看，那个远程主机的目录就好像是自己的一个磁盘分区一样，在使用上相当便利；

其中，主要模块是 RpcProgramNfs3 模块，通过如下类图可以看出，nfs 在 RPC 框架下与 httpServer,NfsMetrics,DFSClientCache,WriteManager 相关联，表明它可以通过 http 协议进行远程访问，也可以用 writemanager 进行本地写入和 DFSClientCache 进行缓存和读取。如图 3-25 为 NF 类间关系图。

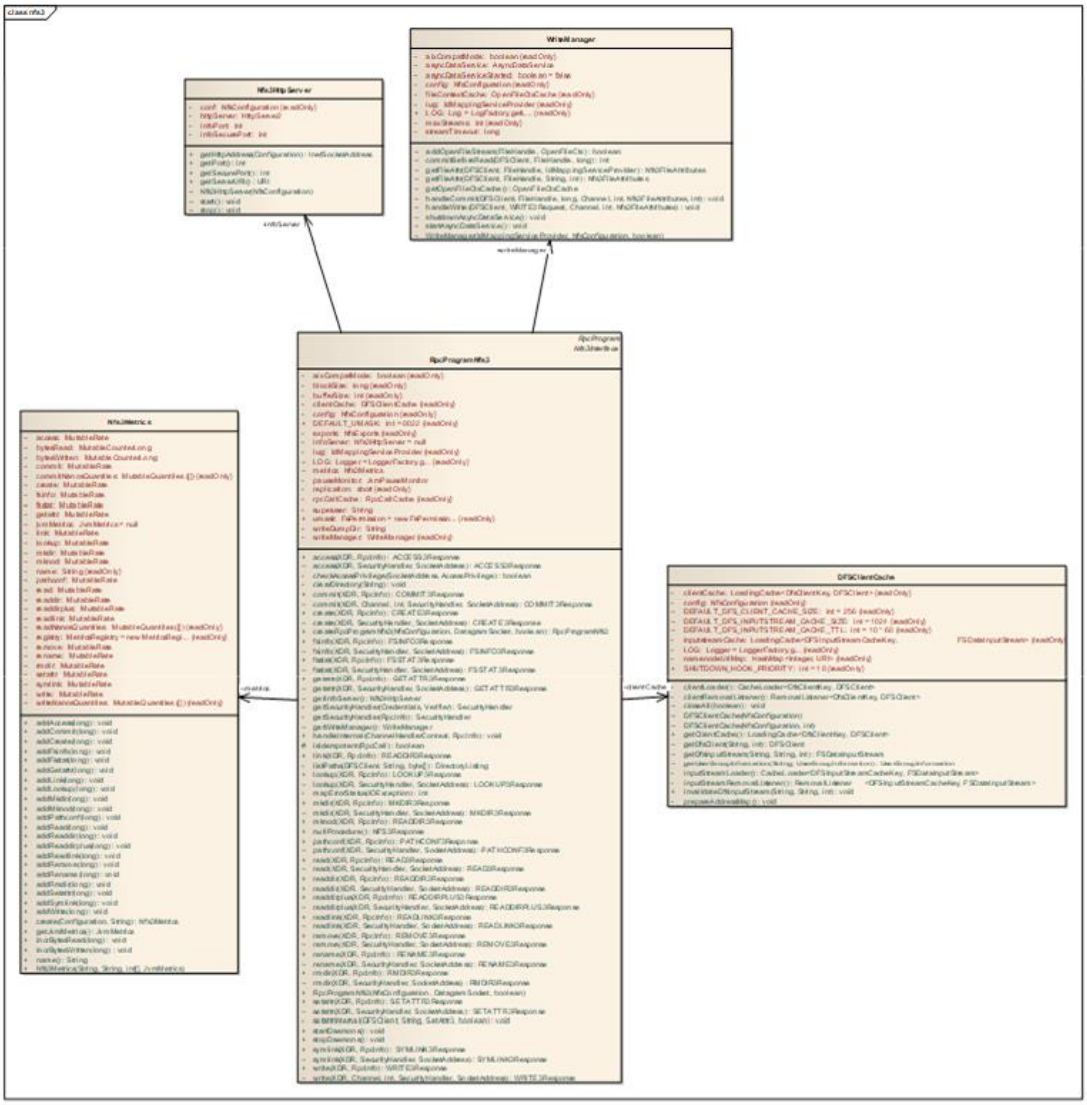

图 3-25 类间关系图

若使用 hdfs 中的 nfs，数据访问路径如下图所示，用户或程序通过 Linux 自带的 nfs client 访问 hdfs nfs 服务，然后再由 nfs 网关作为 hdfs 的客户端访问 hdfs。

在图 3-26 中，中间的节点就是 nfs 代理服务器（hdfs nfs proxy）或 nfs 网关（hdfs nfs gateway）。蓝色代表该模块是一个进程或服务，绿色代表该模块是一个库。图中还画了两条虚线，下、上线分别表示操作系统级别和分布式操作系统(hadpp)级别的内核态与用户态分界

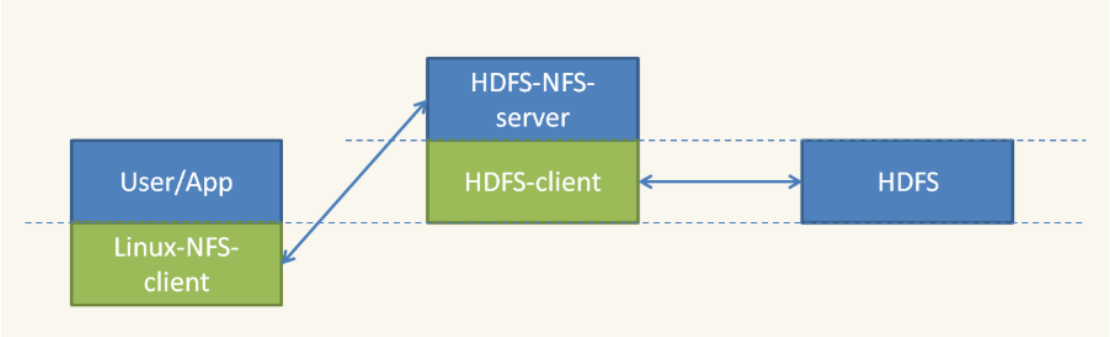

图 3-26 节点关系图

3.5 RBF 模块介绍

Rbf 是指基于路由的 federation，其提出的背景：

（i）、namenode 单点且元数据受限于 namenode 的内存，且 namenode 没有办法水平扩展

（ii）、不支持多租户公用一个 hdfs

其解决的问题有：

（i）、多个 namenode 尽管他们之间是相互独立，这样就水平扩展了 namenode，使得元数据的规模不受限制

（ii）、每个 namenode 与所有的 datanode 交互这样就形成了读写负载

（iii）、每个 namenode 代表一个 ClusterID 从而可以支持多租户

图 3-27 是它的架构图：

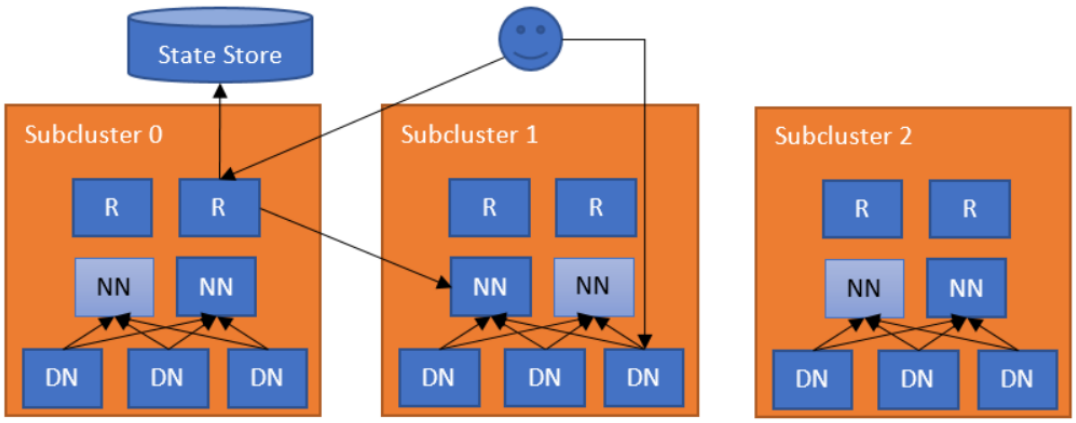

图 3-27 架构图

NameNodes 具有可伸缩性限制，因为由 inode（文件和目录）和文件块组成的元数据开销，Datanode 心跳的数量以及 HDFS RPC 客户端请求的数量。常见

的解决方案是将文件系统拆分为较小的子集群 HDFS Federation，并提供联合视图 ViewF。问题是如何维护子群集的拆分（例如，命名空间分区），这会强制用户连接到多个子群集并管理文件夹/文件的分配。

图 3-28 分别是 Rbf 中的路由器部分与客户端部分的类图，可以看出，在 HDFS 系统中，路由器监视本地 NameNode 并将状态检测到状态存储。当常规 DFS 客户端联系任何路由器以访问联合文件系统中的文件时，路由器会检查状态存储中的挂载表（即本地缓存）以找出包含该文件的子集群。然后，它检查 State Store 中的 Membership 表（即本地缓存），以查找负责子集群的 NameNode。在识别出正确的 NameNode 之后，路由器代理请求。客户端直接访问 Datanodes。

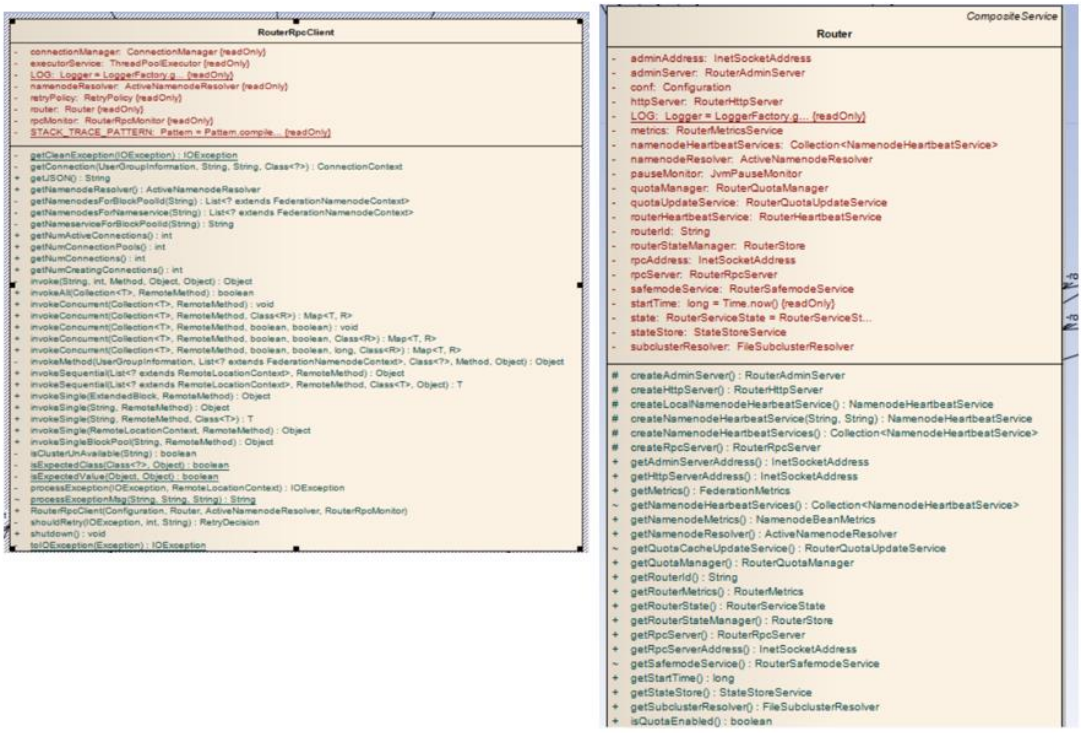

图 3-28 类图示例

四、设计特色分析

HDFS 是 Hadoop 下的分布式文件系统模块，它很好的处理了大规模文件如何进行分布式存储的问题。HDFS 分布式文件系统相比于其它的文件系统来说具有很多优点，如高容错性、适合批处理、适合大数据处理、可构建在廉价的机器上等。这些设计上的特别考虑，为 HDFS 分布式文件系统很好的服务于 Hadoop 的相关功能提供了帮助，使 HDFS 成为目前 Hadoop 架构中无可替代的关键一个子系统部分。

HDFS 分布式文件系统相较于其它一般的文件系统，在设计上有很多可供借鉴的部分，下面就这些部分展开分析。

4.1 高容错性

坦白来说，分布式系统在本质上是不可靠的，这时就需要很多机制来维持分布式系统的可用性，而高容错性的设计正是为了更好的达到可用性的目标。在一个大的集群系统中，有些节点会发生宕机，这些节点出现了故障，对于整个系统来说会受到一定的影响。高容错性的设计就是要尽量将类似的这种影响降到最低，以便维持系统的可持续运行。

对于一个好的容错性设计可以很好的解决以下问题：（i）、当有很多用户需要连接服务器进行相关文件操作时，由于这是一个分布式的大型系统，某些节点出现异常是不可避免的。这时，就需要启用容错机制来处理突发的事件，维持服务器端系统的可用性。所以高容错性的设计侧面的使系统达到了高可用的目标。（ii）、分布式系统中涉及大量的设备，无论是磁盘还是路由，从概率学的角度来看，随着系统规模的增大，部分失败不可以忽略不计。而容错机制的设计可以大大降低系统的维护成本。（iii）、对于 Hadoop 这样的大型系统，由于处理的数据规模巨大，采用分布式系统的设计是必然的。由于系统处理的数据规模都是比较大的，所以处理一个作业可能需要很长时间。如果没有一个很好的高容错性的设计，系统可能根本无法处理完一个作业。

所以高容错性的设计对于 Hadoop 这样的大型分布式系统来说是必不可少的。

针对 Hadoop 系统相关操作中可能遇到的问题，架构师们特别对 HDFS 创新性

的设计了较高的容错机制，其设计原理也是 HDFS 设计中的一亮点所在。主要原理如下文所述。

4.1.1 设计原理

在 HDFS 中主要通过三个方面来保证系统的高容错性，即（i）、系统由数百或数千个服务器机器组成，每个服务器机器储存文件系统数据的一部分；（ii）、数据自动保存多个副本；（iii）、副本丢失后检测故障快速，自动恢复。其中（i）和（ii）的设计思路在一般的分布式系统中较为常见，重点在（iii）的故障检测方法，较为新颖，采用多种检测方式结合，从根本切中问题，达到了较快的检测速度，从而能够及时响应故障，做出应对措施。

系统故障是不可避免的，如何做到故障之后的数据恢复和容错处理是至关重要的。对于 HDFS 来说主要存在三类故障分别为：节点失败、网络故障和数据损坏。对于这三类故障,HDFS 给出了很好的故障检测办法，可以较快速的检测到故障问题，进而快速制定解决方案，保证整个大系统的正常运行。

（i）、节点失败检测

在 HDFS 中主要存在两种节点,即 Namenode 和 Datanode 两种,其中 Namenode 是整个 HDFS 中的唯一单点，如果 Namenode 发生故障，那整个集群都会宕掉，较容易检测的到，所以这里主要针对 Datanode 节点进行检测。

在 Datanode 节点检测过程中采用了类似人体心跳的心跳检测机制。每一个 Datanode 以固定周期向 Namenode 发送心跳信号，通过这种方式告诉 Namenode 节点这些 Datanode 是正常工作的。如果在一定较长的时间内没有收到数据节点发来的心跳信号，这时系统就会认为该数据节点出现了故障。但是，这种情况发生除了可能是节点出现了故障还有另一种原因，那就是可能网络发生了故障，所以这时需要继续进行网络通信方面的故障检测。为了更好的理解节点检测的过程，

特从网上查找了整个节点检测的形象描述，如图 4-1 所示。

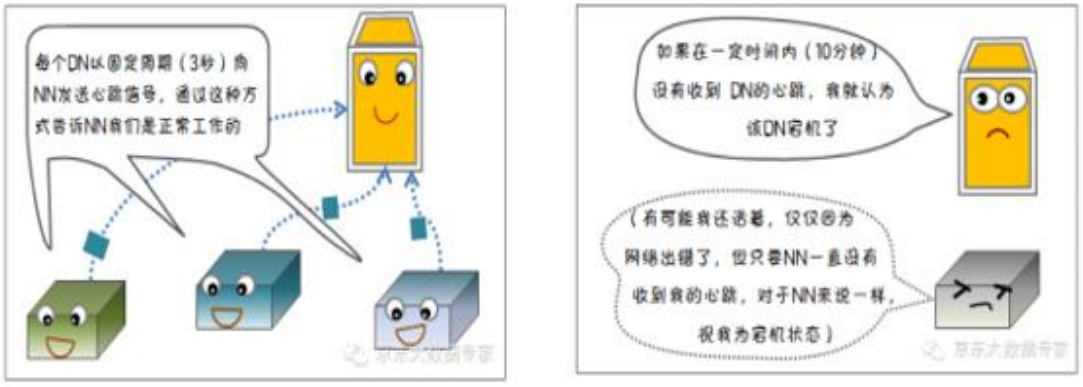

图 4-1 节点故障检测（摘自网络）

（ii）、网络故障检测

在进行了节点故障检测之后，由于无法排除是节点本身出现了故障还是网络传输方面出现了问题，所以需要接着进行网络故障检测。

在网络故障检测中采用了类似网络传输中三次握手原则中用到的确认码机制。只要用一方发送了数据，接收方就会返回给一个确认码。如果没有收到确认码，发送方会重新发送几次相同的数据，若经过几次重试后，还是没有收到确认码，那么这时发送方会认为发送方节点宕掉了或者网络存在错误。同样给出形象的描述，如图 4-2 所示。

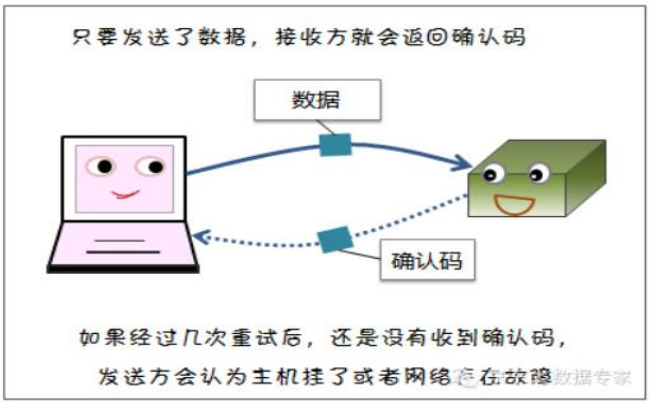

图 4-2 网络故障检测（摘自网络）

（iii）、数据错误检测

另外可能会发生数据损坏的情况，所以数据错误检测也是故障检测中需要注意到的一个关键。在传输数据的时候，同时会发送总和校验码，并且总和校验码会和数据同时存储的硬盘上。所有的数据节点都会定期向名字节点发送数据块的

存储情况，在发送数据块报告前，数据节点会先检查总和校验码是否正确，如果数据存在错误就不发送该数据块的信息。当名字节点发现收到的数据块存在缺失时，就认为有一个数据发生了损坏，从而达到数据错误检测的效果。形象描述如图 4-3 所示。

HDFS 存储理念是以最少的钱买最烂的机器并实现最安全、难度高的分布式文件系统（高容错性低成本），从上可以看出，HDFS 认为机器故障是种常态，所以在设计时充分考虑到单个机器故障，单个磁盘故障，单个文件丢失等情况。

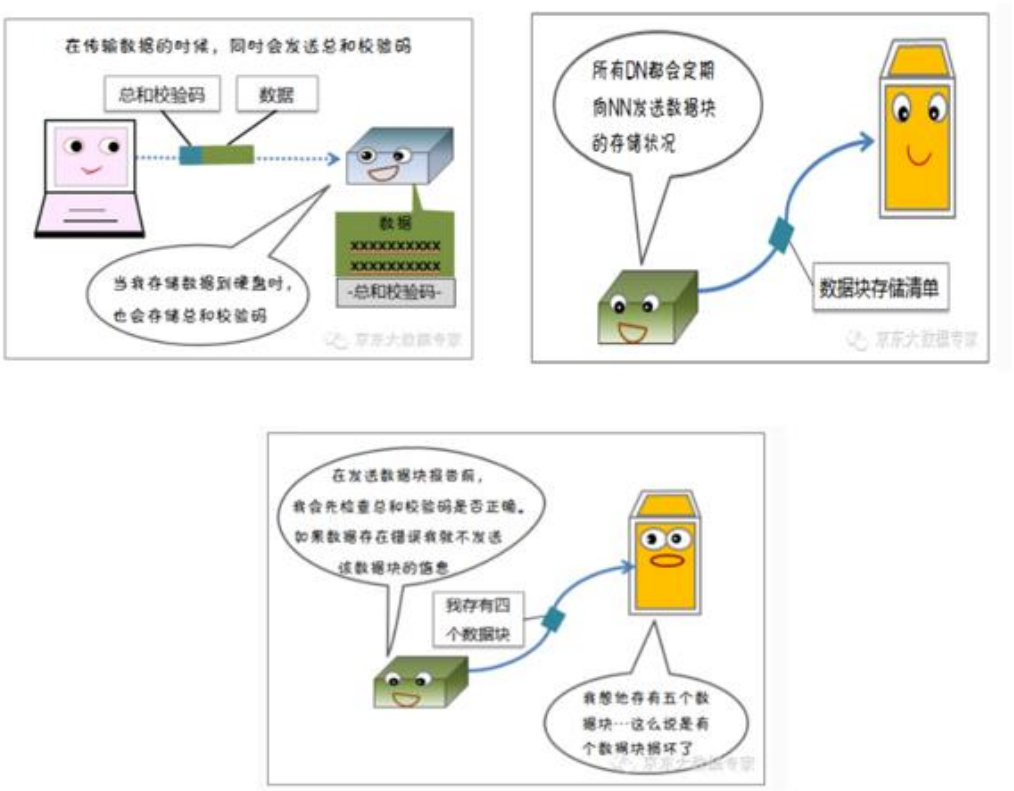

图 4-3 数据错误检测（摘自网络）

4.1.2 优势与需求

HDFS 高容错性的设计很好的解决了大规模数据文件在分布式中的存储问题，减少了整个系统全部宕掉的概率，提高了系统的稳定性。Hadoop 是大数据时代兴起后被提及最多的一个框架，而 HDFS 作为 Hadoop 中数据存储的一个模块，在框架使用中需要很好的稳定性与可用性，HDFS 的设计很好的适应了大规模数据存储的需求，相比于之前传统的分布式系统具有更好的优势。

4.2 读写流程

HDFS 分布式文件系统的读写流程较为复杂，但很好的处理了大规模数据如何读取的问题，具有很好的借鉴意义。

4.2.1 写流程

根据图 4-4 进行写流程的分析。按照图中的标号，（i）、首先创建 filesystem 和 namenode。（ii）、然后返回到客户端，传送写命令给 outputstream 流，告诉它此时需要往数据节点中写数据。（iii）、outputstream 会把数据分成块，写入到数据队列。数据队列由数据流来进行读取，并通知数据节点分配空间存储数据块。按照流水的方式，先将数据块发送给第一个数据节点，然后第一个数据节点将其发送给第二个数据节点，按照这样的规则一步一步的向下传。（iv）、每一个数据节点还会一步一步的往发送端返回确认信息。（v）、当全部发送完毕后，关闭数据流。（vi）、如果数据节点在写入的过程中失败，则关闭传送流水线，将确认队列中的数据块放入数据队列的开始，当前的数据块在已经写入的数据节点中被元数据节点赋予新的标示，则错误节点重启后能够察觉其数据块是过时的，会被删除。失败的数据节点从传送流水线中移除，另外的数据块则写入传送流水线中的另外两个数据节点。元数据节点则被通知此数据块是复制块数不足，将来会再创建第三份备份。

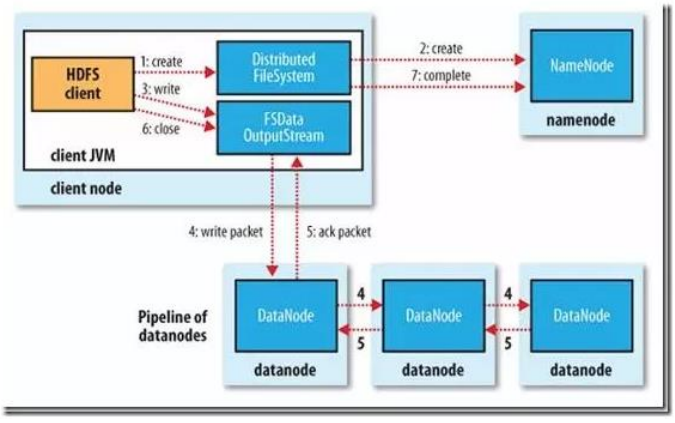

图 4-4 写流程

4.2.2 读流程

根据图 4-5 进行读流程的分析。读流程相比于写流程来说简单了许多，（i）、同样先要打开 filesystem，并获取命名节点的地址。（ii）、FileSystem 返回 InputStream 给客户端，用来读取数据，客户端调用 stream 的 read()函数开始读取数据。（iii）、InputStream 连接保存此文件第一个数据块的最近的数据节点，data 从数据节点读到客户端(client)。（iv）、当此数据块读取完毕时，InputStream 关闭和此数据节点的连接，然后连接此文件下一个数据块的最近的数据节点。（v）、最后关闭数据流。（vi）、在读取数据的过程中，如果客户端在与数据节点通信出现错误，则尝试连接包含此数据块的下一个数据节点。

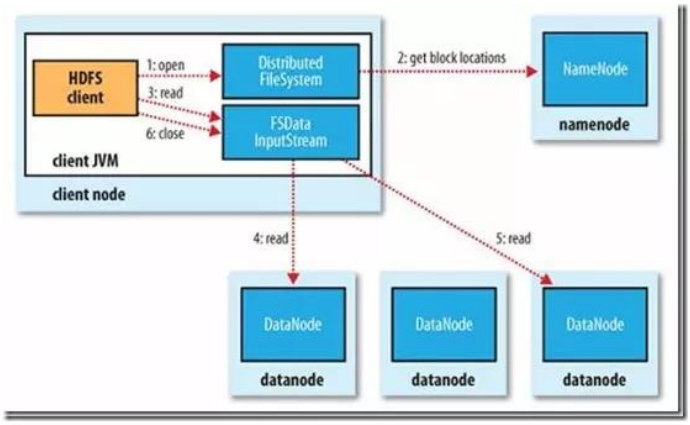

图 4-5 读流程

4.2.3 优势与需求

HDFS 的设计思想“一次写入，多次读取”，一个数据集一旦由数据源生成，就会被复制分发到不同的存储节点中，然后响应各种各样的数据分析任务请求。HDFS 处理的应用一般是批处理，而不是用户交互式处理，注重的是数据的吞吐量而不是数据的访问速度。流式存储的设计可以提高吞吐量，更好的满足 Hadoop 分布式系统的需求。另外采用流式读取的方式而不是随机读取，可以实现多线程复制备份的需求，更好的实现高容错性的要求。

4.3 缺点

什么东西都不是万能的，满足了一方面需求另一方面的问题就会凸显出

来，所以很多情况下的很多问题的研究就是一个取折中的过程。HDFS 系统的设计同样也不例外，它也存在着一些问题：

（i）、不适合低延迟数据访问：HDFS 是为了处理大型数据集，主要是为了达到高的数据吞吐量而设计，这就可能以高延迟作为代价。HDFS 无法做到 10 毫秒以下的响应时间，但可以通过 HBASE 来弥补这一个缺点。

（ii）、无法高效存储大量小文件：namenode 节点在内存中存储整个文件系统的元数据，因此文件的数量就会受到限制，每个文件的元数据大约 150 字节。

（iii）、不支持多用户写入及任意修改文件：不支持多用户对同一文件进行操作，而且写操作只能在文件末尾完成，即追加操作。

五、组内分工情况

学号	名字	工作内容
ZY1806407	李佳磊	1、通过查找资料与分析代码，编写了 3.3 HDFS 客户端的体系结构分析部分。 2、通过分析对比 hdfs 与之前分布式文件系统对比，编写了第四章设计特色分析部分，提出了两点主要的特色，并分析了 hdfs 存在的缺点。 3、编写第六章参考文献；整合大家的内容，为图片添加序号；为全文排版。
ZY1806406	高松	1、利用 Enterprise Architect 对源代码进行逆向工程，生成类图 2、编写第一章项目简介部分 3、软件体系结构设计中 3.1 节的 HDFS 核心内容，包括 Block 、Namenode、以及 DataNode 的结合代码类图的分析和结构介绍，以及 3.4 和 3.5 节对于 HDFS 中的 Nfs 和 Rbf 模块的介绍

ZY1806704	李琪	1、主要负责查找资料，整个项目的需求调研与分析，编写第二章需求分析部分 2、详细研究并撰写了HDFS体系结构中通信协议的部分，编写了3.2节内容 3、对源码进行分析，了解整体体系结构并使用 freeMind 软件以思维导图的形式表现出来。

六、参考文献

[1] 徐鹏.Hadoop 2.X HDFS 源码剖析[M].北京:电子工业出版社,2016.3.
[2] Tom Wbite.Hadoop 权威指南(第 3 版)[M].北京:清华大学出版社,2010.5.
[3] APACHE.hadoop3.1.2 官方文档[EB/OL].2019.1.
[4] Owen O'Malley, Kan Zhang, Sanjay Radia, Ram Marti, and Christopher Harrell.Hadoop Security Design[M].Yahoo,2009.10.
[5] Corejavaguru.HDFS Architecture[EB/OL].
[6] 百度百科.HDFS[EB/OL]. https://baike.baidu.com/item/hdfs/4836121?fr=aladdin,2019.
[7] 刘超.初步掌握 HDFS 的架构及原理[EB/OL]. https://www.cnblogs.com/codeOfLife/p/5375120.html,2016.4.
[8] 刘超.熟练掌握 HDFS 的 JAVA API 接口访问[EB/OL]. https://www.cnblogs.com/codeOfLife/p/5396624.html,2016.4.
[9] Lust-Ring.深入理解 HDFS：Hadoop 分布式文件系统[EB/OL]. https://blog.csdn.net/bingduanlbd/article/details/51914550#t24,2016.7.
[10] 朱培.HDFS 基本原理及数据存取实战[EB/OL]. https://www.cnblogs.com/sdksdk0/p/5585047.html,2016.6.
[11] 漂泊的胡萝卜.HDFS[EB/OL]. https://www.jianshu.com/p/f1e785fffd4d,2018.9.
[12] Easen.Cai.HDFS 知识点总结[EB/OL]. https://www.cnblogs.com/caiyisen/p/7395843.html,2017.7.
[13] 渐行渐远的记忆.Hdfs 详解[EB/OL]. https://www.cnblogs.com/growth-hong/p/6396332.html,2017.2.
[14] Wxquare.认识 HDFS 分布式文件系统[EB/OL]. https://www.cnblogs.com/wxquare/p/4846438.html,2015.9.
[15] 何石.一篇文看懂 Hadoop[EB/OL]. https://www.cnblogs.com/shijiaoyun/p/5778025.html,2016.8.
[16] APACHE.Hadoop 快速入门[EB/OL]. http://hadoop.apache.org/docs/r1.0.4/cn/quickstart.html,2018.9.
[17] W3Cschool.hadoop 教程[EB/OL]. https://www.w3cschool.cn/hadoop/,2018.4.
[18] Dong Cutting.Hadoop 入门概念[EB/OL]. https://www.cnblogs.com/xiaolong1032/p/4345795.html,2015.3.
[19] Master-dragon.hadoop 各种概念整理[EB/OL]. https://blog.csdn.net/qq_26437925/article/details/78467216,2017.11.
[20] 似水流年.史上最详细的 Hadoop 环境搭建[EB/OL]. https://blog.csdn.net/hliq5399/article/details/78193113,2017.10.

七、附件：体系结构的分解过程

如图所示，为使用 freeMind 软件导出的 hdfs 部分的思维导图，由于思维导图过大，.mm 后缀的原图放于作业包中，若有需要可以查看。