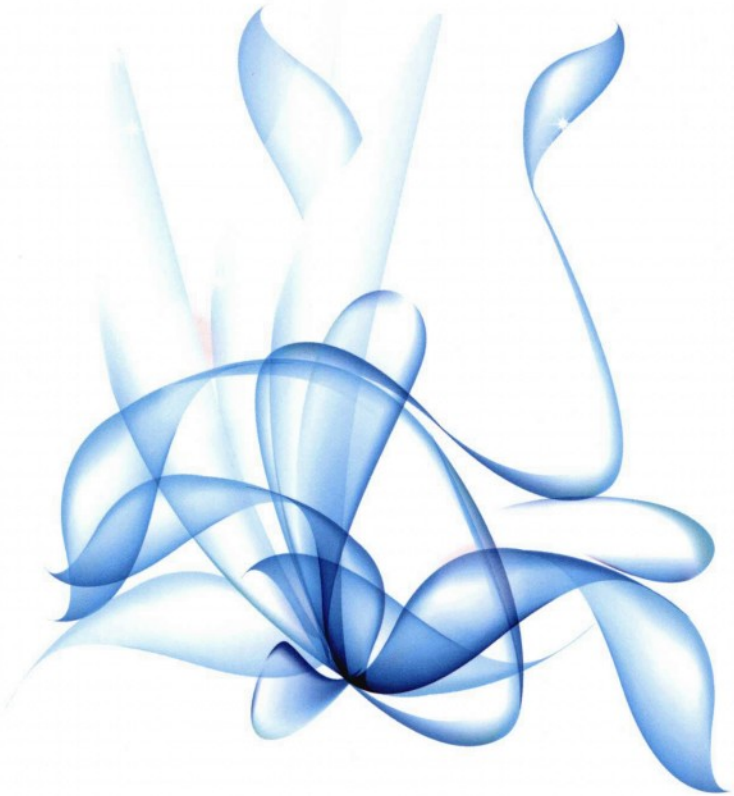
腾讯资深Hadoop技术专家撰写，EasyHadoop和51CTO等专业技术社区联袂推荐！

**从源代码角度深入分析Common和HDFS的架构设计与实现原理，为Hadoop的优化、定制和扩展提供原理性指导。**

**从源代码中参透分布式技术精髓与分布式系统设计的优秀思想和方法。**

华章科技 HZ BOOKS

大数据 技术丛书



Hadoop Internals: in-depth study of Common and HDFS

# Hadoop技术内幕

## 深入解析Hadoop Common和HDFS架构设计与实现原理

蔡斌 陈湘萍◎著

机械工业出版社
China Machine Press

EasyHadoop
技术丛书

51CTO.com
技术成就梦想

联袂推荐

经过几年的快速发展，Hadoop已经成为大数据时代存储和处理海量数据的事实标准，具有高可靠性、高扩展性、高效性和高容错性等许多优点，在学术界备受关注，在业界则更受欢迎。Hadoop的功能非常强大，与之相伴的是使用上的复杂性，如何才能更好地使用Hadoop，更好地对Hadoop进行性能优化、扩展和定制？阅读和研究它的源代码是一种不错的选择。“Hadoop技术内幕”这两本书根据读者的需求，对Common、HDFS和MapReduce等核心技术的架构设计和实现原理进行了深入且详细的分析，对于想了解Hadoop工作机制的读者来说，这两本书应该是十分难得的，强烈推荐。

—— EasyHadoop开源技术社区（www.easyhadoop.com） 致力于Hadoop的易用和普及

Hadoop是近两年来技术领域最热门的话题，从媒体界，到学术界，再到企业界，都给予了高度关注。在企业界，Hadoop的应用已经非常广泛，不仅在IT领域被广泛使用，而且在金融、电信、生物、医药、制造等传统领域也十分受欢迎，有海量数据需要处理的地方，就能用上Hadoop。尤其是在IT领域，从国外的Google、Yahoo、Microsoft、Oracle、Facebook，到国内的阿里巴巴、淘宝、腾讯、百度、新浪、搜狐，都在Hadoop的研发和应用上投入了大量的人力和资金，并获得了丰厚的收益。随着Hadoop越来越普及，用户遇到的问题也会越来越多，越来越复杂。要解决实际生产环境中可能遇到的复杂问题，掌握Hadoop本身的设计和实现原理是根本之道。“Hadoop技术内幕”这两本书就本着这个目标出发，从源代码的角度对Hadoop的工作机制进行了深入的分析，并对实践中可能会遇到的各种问题给出了解决方案。

—— 童小军 资深Hadoop技术专家、企业级大数据解决方案RedHadoop创始人

## 本书主要内容：

- Hadoop的技术优势和整个生态系统；
- Hadoop源代码的获取和阅读环境的搭建；
- Hadoop配置信息的处理；
- 面向海量数据处理的序列化和压缩机制；
- Hadoop的远程过程调用；
- Hadoop抽象文件系统和部分具体文件系统；
- HDFS的总体结构和主要流程；
- 数据节点、名字节点和HDFS客户端的实现。

上架指导：计算机/程序设计
ISBN 978-7-111-41766-8
9 787111 417668 >
定价：89.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站：www.hzbook.com
网上购书：www.china-pub.com

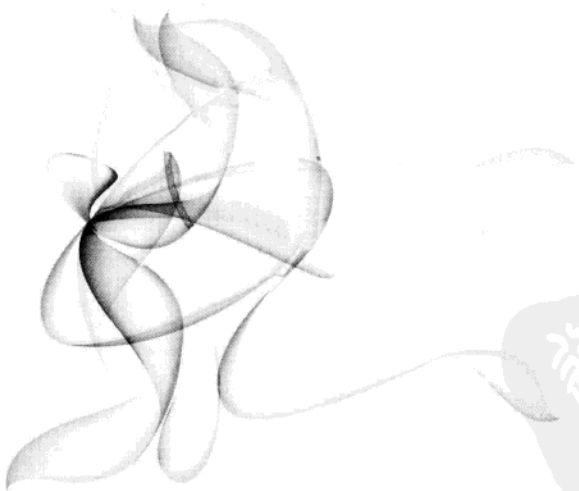
技术丛书

Hadoop Internals: in-depth study of Common and HDFS

# Hadoop技术内幕

## 深入解析Hadoop Common和HDFS架构设计与实现原理

蔡斌 陈湘萍◎著



机械工业出版社
China Machine Press

**图书在版编目（CIP）数据**

Hadoop 技术内幕：深入解析 Hadoop Common 和 HDFS 架构设计与实现原理 / 蔡斌，陈湘萍著 . —北京：机械工业出版社，2013.3

ISBN 978-7-111-41766-8

I. H… II. ①蔡… ②陈… III. ①数据处理软件 ②分布式文件系统 IV. ① TP274 ② TP316

中国版本图书馆 CIP 数据核字（2013）第 047263 号

**版权所有 · 侵权必究**

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

“Hadoop 技术内幕”共两册，分别从源代码的角度对“Common+HDFS”和 MapReduce 的架构设计与实现原理进行了极为详细的分析。本书由腾讯数据平台的资深 Hadoop 专家、X-RIME 的作者亲自执笔，对 Common 和 HDFS 的源代码进行了分析，旨在为 Hadoop 的优化、定制和扩展提供原理性的指导。除此之外，本书还从源代码实现中对分布式技术的精髓、分布式系统设计的优秀思想和方法，以及 Java 语言的编码技巧、编程规范和对设计模式的精妙运用进行了总结和分析，对提高读者的分布式技术能力和 Java 编程能力都非常有帮助。本书适合 Hadoop 的二次开发人员、应用开发工程师、运维工程师阅读。

全书共 9 章，分为三部分：第一部分（第 1 章）主要介绍了 Hadoop 源代码的获取和源代码阅读环境的搭建；第二部分（第 2 ~ 5 章）对 Hadoop 公共工具 Common 的架构设计和实现原理进行了深入分析，包含 Hadoop 的配置信息处理、面向海量数据处理的序列化和压缩机制、Hadoop 的远程过程调用，以及满足 Hadoop 上各类应用访问数据的 Hadoop 抽象文件系统和部分具体文件系统等内容；第三部分（第 6 ~ 9 章）对 Hadoop 的分布式文件系统 HDFS 的架构设计和实现原理进行了详细的分析，这部分内容采用了总分总的结构，第 6 章对 HDFS 的各个实体和实体间接口进行了分析；第 7 章和第 8 章分别详细地研究了数据节点和名字节点的实现原理，并通过第 9 章对客户端的解析，回顾了 HDFS 各节点间的配合，完整地介绍了一个大规模数据存储系统的实现。

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：朱秀英

北京市荣盛彩色印刷有限公司印刷

2013 年 4 月第 1 版第 1 次印刷

186mm × 240 mm · 32.75 印张

标准书号：ISBN 978-7-111-41766-8

定 价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991 88361066

购书热线：（010）68326294 88379649 68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

# 前　言

## 为什么写本书

互联网使得信息的采集、传播速度和规模达到空前的水平，实现了全球的信息共享与交互，它已经成为信息社会必不可少的基础设施，同时也带来了多方面的新挑战。2003 年，Google 发表了《Google File System》，介绍了 Google 海量数据处理使用的文件系统，使互联网时代的数据存储发生了革命性的变化。而 Doug Cutting 等人在 Nutch 项目上应用 GFS 和 MapReduce 思想，并演化为 Hadoop 项目，经过多年的发展，最终形成了包含多个相关项目的软件生态系统，开创了海量数据处理的新局面。

Hadoop 正是为了解决互联网时代的海量数据存储和处理而设计、开发的。简单地讲，Hadoop 是一个可以更容易开发和并行处理大规模数据的分布式计算平台，它的主要特点是：扩展能力强、成本低、高效率、可靠。目前，Hadoop 的用户已经从传统的互联网公司，扩展到科学计算、电信行业、电力行业、生物行业以及金融公司，并得到越来越广泛的应用。

Hadoop 作为一个优秀的开源项目，提供了一些文档和所有的源代码，但是，对于很多开发人员，仅仅通过一些简单的例子或教程学习使用 Hadoop 的基本功能是远远不够的。同时，随着云计算和大数据的发展，产业界正在经历一次重大变革，特别是基于云计算的海量数据处理，改变着我们思考的方式和习惯，开发者们越来越有必要去了解 Hadoop 的架构与设计原理。

本书从源代码的层面上对 Hadoop 的公共工具 Common 和 Hadoop 的分布式文件系统 HDFS 进行了介绍，帮助广大开发者从架构与设计原理的角度去理解 Hadoop，从而为更好

地使用和扩展 Hadoop 打下坚实的基础。同时，Hadoop 是一个使用 Java 语言实现的优秀系统，从事 Java 和分布式计算相关技术的开发者们能从它的源码实现中看到许多优秀的设计思想、对各种设计模式的灵活运用、语言的使用技巧以及编程规范等。这些都有助于加深开发者们对 Java 相关技术，尤其是 Hadoop 的理解，从而提高自己的开发水平，拓展自己的技术视野，为工作带来帮助。

## 读者对象

❑Hadoop 开发人员

对这部分读者来说，本书的内容能够帮助他们加深对 Hadoop 的理解，通过全面了解 Hadoop，特别是 HDFS 的实现原理，为进一步优化、定制和扩展 Hadoop 提供坚实基础。

❑学习分布式技术的读者

Hadoop 是一个得到广泛应用的大型分布式系统，开放的源代码中包含了大量分布式系统设计原理和实现，读者可以通过本书，充分学习、体验和实践分布式技术。

❑学习 Java 语言的中高级读者

Hadoop 使用 Java 语言实现，它充分利用了 Java 的语言特性，并使用了大量的标准库和开源工具，很多功能的设计和实现非常优秀，是极佳的学习 Java 技术的参考资料。

## 本书的主要内容

本书主要分为三个部分。

第一部分（第 1 章）对如何建立 Hadoop 的开发、分析环境做了简单的介绍。对于 Hadoop 这样复杂、庞大的项目，一个好的开发环境可以让读者事半功倍地学习、研究源代码。

第二部分（第 2~5 章）主要对 Hadoop 公共工具 Common 的实现进行研究。分别介绍了 Hadoop 的配置系统、面向海量数据处理的序列化和压缩机制、Hadoop 使用的远程过程调用，以及满足 Hadoop 上各类应用访问数据的 Hadoop 抽象文件系统和部分具体文件系统。

第三部分（第 6~9 章）对 Hadoop 分布式文件系统进行了详细的分析。这部分内容采用总 - 分 - 总的结构，第 6 章介绍了 HDFS 各个实体和实体间接口，第 7 章和第 8 章分别详细地研究了数据节点和名字节点的实现原理，第 9 章通过对客户端的解析，回顾 HDFS 各节点间的配合，完整地介绍了一个大规模数据存储系统的实现。

通过本书，读者不仅能全面了解 Hadoop 的优秀架构和设计思想，而且还能从 Hadoop，特别是 HDFS 的实现源码中一窥 Java 开发的精髓和分布式系统的精要。

## 勘误和支持

由于作者的水平有限，编写时间跨度较长，同时开源软件的演化较快，书中难免会出现

一些错误或者不准确的地方，恳请读者批评指正。如果大家有和本书相关的内容需要探讨，或有更多的宝贵意见，欢迎通过 caibinbupt@qq.com 和我们联系，希望能结识更多的朋友，大家共同进步。书中的源代码文件可以从华章网站[⊖]下载。

## 致谢

感谢机械工业出版社华章公司的编辑杨福川和白宇，杨老师的耐心和支持让本书最终得以出版，白老师的很多建议使本书的可读性更强。

感谢腾讯数据平台部的张文郁、赵重庆和徐钊，作为本书的第一批读者和 Hadoop 专家，他们的反馈意见让本书增色不少。

感谢和我们一起工作、研究和应用 Hadoop 的腾讯数据平台部，以及 IBM 中国研究中心和中山大学的领导和同事们，本书的很多内容是对实际项目的总结。

最后，作者向支持本书写作的家人深表谢意，感谢他们的耐心和理解。



⊖ 参见华章网站 www.hzbook.com——编辑注

# 目　录

前　言

## 第一部分　环境准备

**第 1 章　源代码环境准备** / 2

1.1　什么是 Hadoop / 2

1.1.1　Hadoop 简史 / 2

1.1.2　Hadoop 的优势 / 3

1.1.3　Hadoop 生态系统 / 4

1.2　准备源代码阅读环境 / 8

1.2.1　安装与配置 JDK / 8

1.2.2　安装 Eclipse / 9

1.2.3　安装辅助工具 Ant/ 12

1.2.4　安装类 UNIX Shell 环境 Cygwin / 13

1.3　准备 Hadoop 源代码 / 15

1.3.1　下载 Hadoop / 15

1.3.2　创建 Eclipse 项目 / 16

1.3.3 Hadoop 源代码组织 / 18
1.4 小结 / 19

## 第二部分 Common 的实现

**第 2 章 Hadoop 配置信息处理** / 22
2.1 配置文件简介 / 22
2.1.1 Windows 操作系统的配置文件 / 22
2.1.2 Java 配置文件 / 23
2.2 Hadoop Configuration 详解 / 24
2.2.1 Hadoop 配置文件的格式 / 24
2.2.2 Configuration 的成员变量 / 26
2.2.3 资源加载 / 27
2.2.4 使用 get* 和 set* 访问 / 设置配置项 / 32
2.3 Configurable 接口 / 34
2.4 小结 / 35

**第 3 章 序列化与压缩** / 36
3.1 序列化 / 36
3.1.1 Java 内建序列化机制 / 36
3.1.2 Hadoop 序列化机制 / 38
3.1.3 Hadoop 序列化机制的特征 / 39
3.1.4 Hadoop Writable 机制 / 39
3.1.5 典型的 Writable 类详解 / 41
3.1.6 Hadoop 序列化框架 / 48
3.2 压缩 / 49
3.2.1 Hadoop 压缩简介 / 50
3.2.2 Hadoop 压缩 API 应用实例 / 51
3.2.3 Hadoop 压缩框架 / 52
3.2.4 Java 本地方法 / 61
3.2.5 支持 Snappy 压缩 / 65
3.3 小结 / 69

**第 4 章　Hadoop 远程过程调用** / 70

4.1　远程过程调用基础知识 / 70

4.1.1　RPC 原理 / 70

4.1.2　RPC 机制的实现 / 72

4.1.3　Java 远程方法调用 / 73

4.2　Java 动态代理 / 78

4.2.1　创建代理接口 / 78

4.2.2　调用转发 / 80

4.2.3　动态代理实例 / 81

4.3　Java NIO/ 84

4.3.1　Java 基本套接字 / 84

4.3.2　Java NIO 基础 / 86

4.3.3　Java NIO 实例：回显服务器 / 93

4.4　Hadoop 中的远程过程调用 / 96

4.4.1　利用 Hadoop IPC 构建简单的分布式系统 / 96

4.4.2　Hadoop IPC 的代码结构 / 100

4.5　Hadoop IPC 连接相关过程 / 104

4.5.1　IPC 连接成员变量 / 104

4.5.2　建立 IPC 连接 / 106

4.5.3　数据分帧和读写 / 111

4.5.4　维护 IPC 连接 / 114

4.5.5　关闭 IPC 连接 / 116

4.6　Hadoop IPC 方法调用相关过程 / 118

4.6.1　Java 接口与接口体 / 119

4.6.2　IPC 方法调用成员变量 / 121

4.6.3　客户端方法调用过程 / 123

4.6.4　服务器端方法调用过程 / 126

4.7　Hadoop IPC 上的其他辅助过程 / 135

4.7.1　RPC.getProxy() 和 RPC.stopProxy() / 136

4.7.2　RPC.getServer() 和 Server 的启停 / 138

4.8　小结 / 141

**第 5 章　Hadoop 文件系统** / 142

5.1　文件系统 / 142

5.1.1 文件系统的用户界面 / 142
5.1.2 文件系统的实现 / 145
5.1.3 文件系统的保护控制 / 147
5.2 Linux 文件系统 / 150
5.2.1 Linux 本地文件系统 / 150
5.2.2 虚拟文件系统 / 153
5.2.3 Linux 文件保护机制 / 154
5.2.4 Linux 文件系统 API/ 155
5.3 分布式文件系统 / 159
5.3.1 分布式文件系统的特性 / 159
5.3.2 基本 NFS 体系结构 / 160
5.3.3 NFS 支持的文件操作 / 160
5.4 Java 文件系统 / 162
5.4.1 Java 文件系统 API / 162
5.4.2 URI 和 URL / 164
5.4.3 Java 输入 / 输出流 / 166
5.4.4 随机存取文件 / 169
5.5 Hadoop 抽象文件系统 / 170
5.5.1 Hadoop 文件系统 API / 170
5.5.2 Hadoop 输入 / 输出流 / 175
5.5.3 Hadoop 文件系统中的权限 / 179
5.5.4 抽象文件系统中的静态方法 / 180
5.5.5 Hadoop 文件系统中的协议处理器 / 184
5.6 Hadoop 具体文件系统 / 188
5.6.1 FileSystem 层次结构 / 189
5.6.2 RawLocalFileSystem 的实现 / 191
5.6.3 ChecksumFileSystem 的实现 / 196
5.6.4 RawInMemoryFileSystem 的实现 / 210
5.7 小结 / 213

**第三部分 Hadoop 分布式文件系统**

**第 6 章 HDFS 概述** / 216

6.1 初识 HDFS / 216

6.1.1 HDFS 主要特性 / 216
6.1.2 HDFS 体系结构 / 217
6.1.3 HDFS 源代码结构 / 221
6.2 基于远程过程调用的接口 / 223
6.2.1 与客户端相关的接口 / 224
6.2.2 HDFS 各服务器间的接口 / 236
6.3 非远程过程调用接口 / 244
6.3.1 数据节点上的非 IPC 接口 / 245
6.3.2 名字节点和第二名字节点上的非 IPC 接口 / 252
6.4 HDFS 主要流程 / 254
6.4.1 客户端到名字节点的文件与目录操作 / 254
6.4.2 客户端读文件 / 256
6.4.3 客户端写文件 / 257
6.4.4 数据节点的启动和心跳 / 258
6.4.5 第二名字节点合并元数据 / 259
6.5 小结 / 261

**第 7 章 数据节点实现** / 263

7.1 数据块存储 / 263
7.1.1 数据节点的磁盘目录文件结构 / 263
7.1.2 数据节点存储的实现 / 266
7.1.3 数据节点升级 / 269
7.1.4 文件系统数据集的工作机制 / 276
7.2 流式接口的实现 / 285
7.2.1 DataXceiverServer 和 DataXceiver / 286
7.2.2 读数据 / 289
7.2.3 写数据 / 298
7.2.4 数据块替换、数据块拷贝和读数据块检验信息 / 313
7.3 作为整体的数据节点 / 314
7.3.1 数据节点和名字节点的交互 / 314
7.3.2 数据块扫描器 / 319
7.3.3 数据节点的启停 / 321
7.4 小结 / 326

**第 8 章　名字节点实现** / 327

8.1　文件系统的目录树 / 327

8.1.1　从 i-node 到 INode/ 327

8.1.2　命名空间镜像和编辑日志 / 333

8.1.3　第二名字节点 / 351

8.1.4　FSDirectory 的实现 / 361

8.2　数据块和数据节点管理 / 365

8.2.1　数据结构 / 366

8.2.2　数据节点管理 / 378

8.2.3　数据块管理 / 392

8.3　远程接口 ClientProtocol 的实现 / 412

8.3.1　文件和目录相关事务 / 412

8.3.2　读数据使用的方法 / 415

8.3.3　写数据使用的方法 / 419

8.3.4　工具 dfsadmin 依赖的方法 / 443

8.4　名字节点的启动和停止 / 444

8.4.1　安全模式 / 444

8.4.2　名字节点的启动 / 449

8.4.3　名字节点的停止 / 454

8.5　小结 / 454

**第 9 章　HDFS 客户端** / 455

9.1　认识 DFSClient / 455

9.1.1　DFSClient 的构造和关闭 / 455

9.1.2　文件和目录、系统管理相关事务 / 457

9.1.3　删除 HDFS 文件 / 目录的流程 / 459

9.2　输入流 / 461

9.2.1　读数据前的准备：打开文件 / 463

9.2.2　读数据 / 465

9.2.3　关闭输入流 / 475

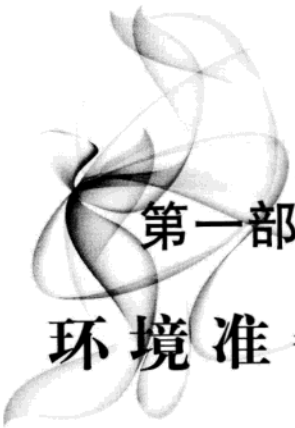
9.2.4　读取 HDFS 文件数据的流程 / 475

9.3　输出流 / 478

9.3.1　写数据前的准备：创建文件 / 481

9.3.2　写数据：数据流管道的建立 / 482

9.3.3 写数据：数据包的发送 / 486
9.3.4 写数据：数据流管道出错处理 / 493
9.3.5 写数据：租约更新 / 496
9.3.6 写数据：DFSOutputStream.sync() 的作用 / 497
9.3.7 关闭输出流 / 499
9.3.8 向 HDFS 文件写入数据的流程 / 500
9.4 DistributedFileSystem 的实现 / 506
9.5 HDFS 常用工具 / 508
9.5.1 FsShell / 508
9.5.2 DFSAdmin / 510
9.6 小结 / 511



# 第一部分

# 环境准备

**本部分内容**

- 源代码环境准备

# 第 1 章　源代码环境准备

数据！数据！数据！

今天，我们正被数据包围。全球 43 亿部电话、20 亿位互联网用户每秒都在不断地产生大量数据，人们发送短信给朋友、上传视频、用手机拍照、更新社交网站的信息、转发微博、点击广告等，使得机器产生和保留了越来越多的数据。数据的指数级增长对处于市场领导地位的互联网公司，如 Facebook、谷歌、雅虎、亚马逊、腾讯等提出了挑战。它们需要对 TB 级别和 PB 级别的数据进行分析处理，以发现哪些网站更受欢迎，哪些商品更具有吸引力，哪些广告更吸引用户。传统的工具对于处理如此规模的数据集越来越无能为力。

现在，Hadoop 应运而生，庞大的信息流有了新的处理平台。

## 1.1　什么是 Hadoop

Hadoop 是 Apache 基金会下的一个开源分布式计算平台，以 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）和 MapReduce 分布式计算框架为核心，为用户提供了底层细节透明的分布式基础设施。HDFS 的高容错性、高伸缩性等优点，允许用户将 Hadoop 部署在廉价的硬件上，构建分布式系统；MapReduce 分布式计算计算框架则允许用户在不了解分布式系统底层细节的情况下开发并行、分布的应用程序，充分利用大规模的计算资源，解决传统高性能单机无法解决的大数据处理问题。

Apache Hadoop 是目前分析海量数据的首选工具。

### 1.1.1　Hadoop 简史

谈到 Hadoop 的历史，就不得不提到 Lucene 和 Nutch。Hadoop 开始时是 Nutch 的一个子项目，而 Nutch 又是 Apache Lucene 的子项目。这 3 个项目都是由 Doug Cutting 创立，每个项目在逻辑上都是前一个项目的演进。

Lucene 是引擎开发工具包，提供了一个纯 Java 的高性能全文索引，它可以方便地嵌入各种实际应用中实现全文搜索 / 索引功能。Nutch 项目开始于 2002 年，是以 Lucene 为基础实现的搜索引擎应用。Lucene 为 Nutch 提供了文本搜索和索引的 API，Nutch 不光有搜索功能，还有数据抓取的功能。

但很快，Doug Cutting 和 Mike Calarella（Hadoop 和 Nutch 的另一位创始人）就意识到，他们的架构无法扩展以支持拥有数十亿网页的网络。这个时候，Google 的研究人员在 2003 年的 ACM SOSP（Symposium on Operating Systems Principles）会议上发表的描述 Google 分布式文件系统（简称 GFS）的论文及时地为他们提供了帮助。GFS 或类似的系统可以解决他

们在网络抓取和索引过程中产生的大量文件存储需求。于是，在 2004 年，他们开始写 GFS 的一个开源实现，即 Nutch 分布式文件系统（NDFS）。

2004 年，在 OSDI（Operating Systems Design and Implementation）会议上，Google 发表了论文，向全世界介绍了 MapReduce。2005 年初，Nutch 的开发者在 Nutch 上有了一个可工作的 MapReduce 应用，到当年的年中，所有主要的 Nutch 算法被迁移到 MapReduce 和 NDFS 上。

在 Nutch0.8.0 版本之前，Hadoop 还属于 Nutch 的一部分，而从 Nutch0.8.0 开始，Doug Cutting 等人将其中实现的 NDFS 和 MapReduce 剥离出来成立了一个新的开源项目，这就是 Hadoop。同时，对比以前的 Nutch 版本，Nutch0.8.0 在架构上有了根本性的变化，它完全构建在 Hadoop 的基础之上了。这个时候，已经是 2006 年 2 月，大约在同一时间，Doug Cutting 加入雅虎，Yahoo 投入了专门的团队和资源将 Hadoop 发展成一个可在网络上运行的系统。

值得一提的是 Hadoop 名字的来源。

为软件项目命名时，Doug Cutting 似乎总会得到家人的启发。Lucene 是他妻子的中间名，也是她外祖母的名字。他的儿子在咿呀学语时，总把所有用于吃饭的词叫成 Nutch。Doug Cutting 如此解释 Hadoop 的得名："这是我的孩子给一头吃饱了的棕黄色大象起的名字。我的命名标准就是简短，容易发音和拼写，没有太多的意义，并且不会被用于别处。小孩子是这方面的高手，Googol 就是由小孩命名的。"

2008 年 1 月，Hadoop 已成为 Apache 顶级项目，证明它是成功的。通过这次机会，Hadoop 成功地被雅虎之外的很多公司应用，如 Facebook、纽约时报等。特别是纽约时报，它使用运行在亚马逊的 EC2 云计算上的 Hadoop，将 4TB 的报纸扫描文档压缩，转换为用于 Web 的 PDF 文档，这个过程历时不到 24 小时，使用 100 台机器运行，这成为 Hadoop 一个良好的宣传范例。

2008 年 2 月，雅虎宣布其索引网页的生产系统采用了在 10 000 多个核的 Linux 集群上运行的 Hadoop。Hadoop 真正达到了万维网的规模。2008 年 4 月，在一个 900 节点的 Hadoop 集群上，雅虎的研究人员运行 1TB 的 Jim Gray 基准排序，只用了 209 秒，而到了 2009 年 4 月，在一个 1400 节点的集群上对 500GB 数据进行排序，只用了 59 秒，这显示了 Hadoop 强大的计算能力。

2008 年开始，Hadoop 迈向主流，开始了它的爆发式发展，出现了大量的相关项目，如 2008 年的 HBase、ZooKeeper 和 Mahout，2009 年的 Pig、Hive 等。同时，还出现了像 Cloudera（成立于 2008 年）和 Hortonworks（以雅虎的 Hadoop 业务部门为基础成立的公司）这样的专注于 Hadoop 的公司。

经过多年的发展，Hadoop 已经从初出茅庐的小象变身为行业巨人。

### 1.1.2　Hadoop 的优势

将 Hadoop 运用于海量数据处理，主要有如下几个优势：

- 方便：Hadoop 可以运行在一般商业机器构成的大型集群上，或者是亚马逊弹性计算云（Amazon EC2）等云计算服务上。
- 弹性：Hadoop 通过增加集群节点，可以线性地扩展以处理更大的数据集。同时，在集群负载下降时，也可以减少节点，以高效使用计算资源。
- 健壮：Hadoop 在设计之初，就将故障检测和自动恢复作为一个设计目标，它可以从容处理通用计算平台上出现的硬件失效的情况。
- 简单：Hadoop 允许用户快速编写出高效的并行分布代码。

由于 Hadoop 具有上述优势，使得 Hadoop 在学术界和工业界都大受欢迎。今天，Hadoop 已经成为许多公司和大学基础计算平台的一部分。学术界如内布拉斯加大学通过使用 Hadoop，支持紧凑型 μ 子螺旋形磁谱仪实验数据的保存和计算；加州大学伯克利分校则对 Hadoop 进行研究，以提高其整体性能；在国内，中国科学院计算技术研究所在 Hadoop 上开展了数据挖掘和地理信息处理等的研究。在工业界，Hadoop 已经成为很多互联网公司基础计算平台的一个核心部分，如雅虎、Facebook、腾讯等；传统行业，如传媒、电信、金融，也在使用这个系统，进行数据存储与处理。

如今，Hadoop 分布式计算基础架构这把“大伞”下，已经包含了多个子项目。而海量数据处理也迅速成为许多程序员需要掌握的一项重要技能。

### 1.1.3 Hadoop 生态系统

经过几年的快速发展，Hadoop 现在已经发展成为包含多个相关项目的软件生态系统。狭义的 Hadoop 核心只包括 Hadoop Common、Hadoop HDFS 和 Hadoop MapReduce 三个子项目，但和 Hadoop 核心密切相关的，还包括 Avro、ZooKeeper、Hive、Pig 和 HBase 等项目，构建在这些项目之上的，面向具体领域、应用的 Mahout、X-Rime、Crossbow 和 Ivory 等项目，以及 Chukwa、Flume、Sqoop、Oozie 和 Karmasphere 等数据交换、工作流和开发环境这样的外围支撑系统。它们提供了互补性的服务，共同提供了一个海量数据处理的软件生态系统，Hadoop 生态系统如图 1-1 所示。

图 1-1 Hadoop 生态系统

下面详细介绍生态系统的组成。

1. Hadoop Common

从 Hadoop 0.20 版本开始，原来 Hadoop 项目的 Core 部分更名为 Hadoop Common。Common 为 Hadoop 的其他项目提供了一些常用工具，主要包括系统配置工具 Configuration、远程过程调用 RPC、序列化机制和 Hadoop 抽象文件系统 FileSystem 等。它们为在通用硬件上搭建云计算环境提供基本的服务，并为运行在该平台上的软件开发提供了所需的 API。

2. Avro

Avro 由 Doug Cutting 牵头开发，是一个数据序列化系统。类似于其他序列化机制，Avro 可以将数据结构或者对象转换成便于存储和传输的格式，其设计目标是用于支持数据密集型应用，适合大规模数据的存储与交换。Avro 提供了丰富的数据结构类型、快速可压缩的二进制数据格式、存储持久性数据的文件集、远程调用 RPC 和简单动态语言集成等功能。

3. ZooKeeper

在分布式系统中如何就某个值（决议）达成一致，是一个十分重要的基础问题。ZooKeeper 作为一个分布式的服务框架，解决了分布式计算中的一致性问题。在此基础上，ZooKeeper 可用于处理分布式应用中经常遇到的一些数据管理问题，如统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。ZooKeeper 常作为其他 Hadoop 相关项目的主要组件，发挥着越来越重要的作用。

4. HDFS

HDFS（Hadoop Distributed File System，Hadoop 分布式文件系统）是 Hadoop 体系中数据存储管理的基础。它是一个高度容错的系统，能检测和应对硬件故障，用于在低成本的通用硬件上运行。HDFS 简化了文件的一致性模型，通过流式数据访问，提供高吞吐量应用程序数据访问功能，适合带有大型数据集的应用程序。

5. MapReduce

MapReduce 是一种计算模型，用以进行大数据量的计算。Hadoop 的 MapReduce 实现，和 Common、HDFS 一起，构成了 Hadoop 发展初期的三个组件。MapReduce 将应用划分为 Map 和 Reduce 两个步骤，其中 Map 对数据集上的独立元素进行指定的操作，生成键 - 值对形式中间结果。Reduce 则对中间结果中相同“键”的所有“值”进行规约，以得到最终结果。MapReduce 这样的功能划分，非常适合在大量计算机组成的分布式并行环境里进行数据处理。

6. HBase

Google 发表了 BigTable 系统论文后，开源社区就开始在 HDFS 上构建相应的实现 HBase。HBase 是一个针对结构化数据的可伸缩、高可靠、高性能、分布式和面向列的动态模式数据库。和传统关系数据库不同，HBase 采用了 BigTable 的数据模型：增强的稀疏排序映射表（Key/Value），其中，键由行关键字、列关键字和时间戳构成。HBase 提供了对大规

模数据的随机、实时读写访问，同时，HBase 中保存的数据可以使用 MapReduce 来处理，它将数据存储和并行计算完美地结合在一起。

7. Hive

Hive 是 Hadoop 中的一个重要子项目，最早由 Facebook 设计，是建立在 Hadoop 基础上的数据仓库架构，它为数据仓库的管理提供了许多功能，包括：数据 ETL（抽取、转换和加载）工具、数据存储管理和大型数据集的查询和分析能力。Hive 提供的是一种结构化数据的机制，定义了类似于传统关系数据库中的类 SQL 语言：Hive QL，通过该查询语言，数据分析人员可以很方便地运行数据分析业务。

8. Pig

Pig 运行在 Hadoop 上，是对大型数据集进行分析和评估的平台。它简化了使用 Hadoop 进行数据分析的要求，提供了一个高层次的、面向领域的抽象语言：Pig Latin。通过 Pig Latin，数据工程师可以将复杂且相互关联的数据分析任务编码为 Pig 操作上的数据流脚本，通过将该脚本转换为 MapReduce 任务链，在 Hadoop 上执行。和 Hive 一样，Pig 降低了对大型数据集进行分析和评估的门槛。

9. Mahout

Mahout 起源于 2008 年，最初是 Apache Lucent 的子项目，它在极短的时间内取得了长足的发展，现在是 Apache 的顶级项目。Mahout 的主要目标是创建一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序。Mahout 现在已经包含了聚类、分类、推荐引擎（协同过滤）和频繁集挖掘等广泛使用的数据挖掘方法。除了算法，Mahout 还包含数据的输入 / 输出工具、与其他存储系统（如数据库、MongoDB 或 Cassandra）集成等数据挖掘支持架构。

10. X-RIME

X-RIME 是一个开源的社会网络分析工具，它提供了一套基于 Hadoop 的大规模社会网络 / 复杂网络分析工具包。X-RIME 在 MapReduce 的框架上对十几种社会网络分析算法进行了并行化与分布式化，从而实现了对互联网级大规模社会网络 / 复杂网络的分析。它包括 HDFS 存储系统上的一套适合大规模社会网络分析的数据模型、基于 MapReduce 实现的一系列社会网络分析分布式并行算法和 X-RIME 处理模型，即 X-RIME 工具链等三部分。

11. Crossbow

Crossbow 是在 Bowtie 和 SOAPsnp 基础上，结合 Hadoop 的可扩展工具，该工具能够充分利用集群进行生物计算。其中，Bowtie 是一个快速、高效的基因短序列拼接至模板基因组工具；SOAPsnp 则是一个重测序一致性序列建造程序。它们在复杂遗传病和肿瘤易感的基因定位，到群体和进化遗传学研究中发挥着重要的作用。Crossbow 利用了 Hadoop Stream，将 Bowtie、SOAPsnp 上的计算任务分布到 Hadoop 集群中，满足了新一代基因测序技术带来的海量数据存储及计算分析要求。

### 12. Chukwa

Chukwa 是开源的数据收集系统，用于监控大规模分布式系统（2000+ 以上的节点，系统每天产生的监控数据量在 T 级别）。它构建在 Hadoop 的 HDFS 和 MapReduce 基础之上，继承了 Hadoop 的可伸缩性和鲁棒性。Chukwa 包含一个强大和灵活的工具集，提供了数据的生成、收集、排序、去重、分析和展示等一系列功能，是 Hadoop 使用者、集群运营人员和管理人员的必备工具。

### 13. Flume

Flume 是 Cloudera 开发维护的分布式、可靠、高可用的日志收集系统。它将数据从产生、传输、处理并最终写入目标的路径的过程抽象为数据流，在具体的数据流中，数据源支持在 Flume 中定制数据发送方，从而支持收集各种不同协议数据。同时，Flume 数据流提供对日志数据进行简单处理的能力，如过滤、格式转换等。此外，Flume 还具有能够将日志写往各种数据目标（可定制）的能力。总的来说，Flume 是一个可扩展、适合复杂环境的海量日志收集系统。

### 14. Sqoop

Sqoop 是 SQL-to-Hadoop 的缩写，是 Hadoop 的周边工具，它的主要作用是在结构化数据存储与 Hadoop 之间进行数据交换。Sqoop 可以将一个关系型数据库（例如 MySQL、Oracle、PostgreSQL 等）中的数据导入 Hadoop 的 HDFS、Hive 中，也可以将 HDFS、Hive 中的数据导入关系型数据库中。Sqoop 充分利用了 Hadoop 的优点，整个数据导入导出过程都是用 MapReduce 实现并行化，同时，该过程中的大部分步骤自动执行，非常方便。

### 15. Oozie

在 Hadoop 中执行数据处理工作，有时候需要把多个作业连接到一起，才能达到最终目的。针对上述需求，Yahoo 开发了开源工作流引擎 Oozie，用于管理和协调多个运行在 Hadoop 平台上的作业。在 Oozie 中，计算作业被抽象为动作，控制流节点则用于构建动作间的依赖关系，它们一起组成一个有向无环的工作流，描述了一项完整的数据处理工作。Oozie 工作流系统可以提高数据处理流程的柔性，改善 Hadoop 集群的效率，并降低开发和运营人员的工作量。

### 16. Karmasphere

Karmasphere 包括 Karmasphere Analyst 和 Karmasphere Studio。其中，Analyst 提供了访问保存在 Hadoop 里面的结构化和非结构化数据的能力，用户可以运用 SQL 或其他语言，进行即时查询并做进一步的分析。Studio 则是基于 NetBeans 的 MapReduce 集成开发环境，开发人员可以利用它方便快速地创建基于 Hadoop 的 MapReduce 应用。同时，该工具还提供了一些可视化工具，用于监控任务的执行，显示任务间的输入输出和交互等。需要注意的是，在上面提及的这些项目中，Karmasphere 是唯一不开源的工具。

正是这些项目的发展，带来了廉价的处理大数据的能力，让 Hadoop 成为大数据行业发展背后的驱动力。如今，Hadoop 已成为分布式大数据处理事实上的标准。

## 1.2 准备源代码阅读环境

在研究一个开源项目之前，都需要安装与配置基本的开发环境和源代码的阅读环境。这一系列内容包括：安装与配置 JDK、安装开发调试 IDE、安装与配置相关辅助工具等。

### 1.2.1 安装与配置 JDK

在分析 Hadoop 的源代码前，需要做一些准备工作，其中搭建 Java 环境是必不可少的。Hadoop 的运行环境要求 Java 1.6 以上的版本。打开 http://www.oracle.com/technetwork/java/javase/downloads/index.html 页面，可以下载最新的 JDK 安装程序，下载页面如图 1-2 所示。

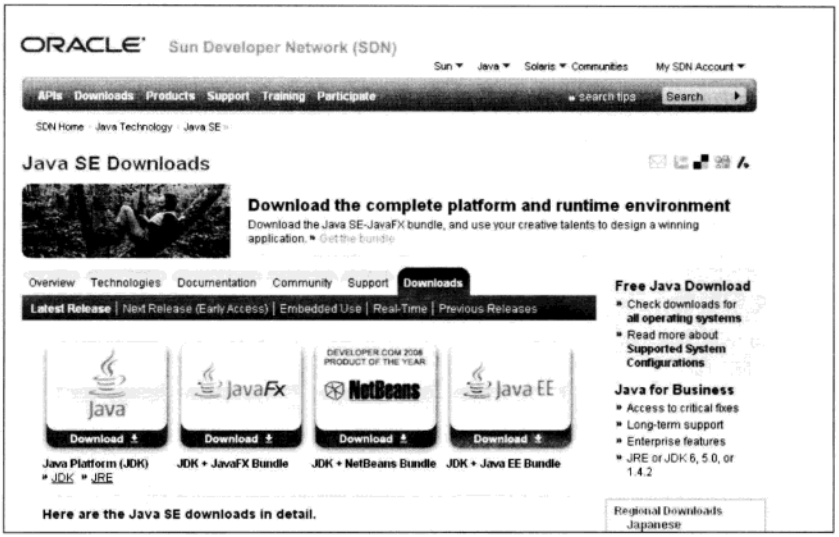



图 1-2 JDK 下载首页

安装完后，要检查 JDK 是否配置正确。

某些第三方的程序会把自己的 JDK 路径加到系统 PATH 环境变量中，这样，即便安装最新版本的 JDK，系统还是会使用第三方程序所带的 JDK。在 Windows 环境中，需要正确配置的 Java 运行时环境变量有 JAVA_HOME、CLASSPATH 和 PATH 等。

方便起见，我们往往为操作系统本身指定一个系统级别的环境变量。例如，Windows 平台上的系统环境变量可以在“系统属性”的“高级”选项卡中找到，可在其中配置 JAVA_HOME、PATH 和 CLASSPATH 值。图 1-3 是 Windows XP 操作系统中为系统添加 JAVA_HOME 环境变量的例子。

安装并配置完成后，可以在命令行窗口中输入“java -version”命令检测当前的 JDK 运行版本。如果配置完全正确，会显示当前客户端的 JRE 运行版本，如图 1-4 所示。

图 1-3　Windows XP 上添加 JAVA_HOME 环境变量

```
java version "1.6.0_21"
Java(TM) SE Runtime Environment (build 1.6.0_21-b07)
Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
```

图 1-4　JDK 安装成功

## 1.2.2　安装 Eclipse

在成功安装和配置 JDK 后，还需要安装进行 Java 开发调试的 IDE（Integrated Development Environment，集成开发环境），因为一个好的开发环境和源代码阅读环境可以使工作效率事半功倍。目前比较常用的 Java 开发 IDE 主要有 Eclipse 和 NetBeans 等，读者可以任意选择自己习惯的 IDE 作为开发工具。本书以 Eclipse 集成开发环境为例，着重介绍在 Eclipse 中开发与调试源码的方法。读者也可以举一反三，在其他 IDE 中做相应的尝试。

Eclipse 是一个界面友好的开源 IDE，并支持成千上万种不同的插件，为代码分析和源码调试提供了极大的便利。可以在 Eclipse 官方网站（http://www.eclipse.org/downloads/）找到 Eclipse 的各个版本（对 Hadoop 源码进行分析，只需要下载 Eclipse IDE for Java SE Developers）并下载安装。Eclipse 下载页面如图 1-5 所示。Eclipse 是基于 Java 的绿色软件，解压下载得到 ZIP 包后就能直接使用。关于 Eclipse 的基本使用已超出了本书的范围，因此下面仅向读者简要介绍如何使用 Eclipse 进行一些基本的源代码分析工作。

**1. 定位某个类、方法和属性**

在分析源代码的过程中，有时候需要快速定位光标位置的某个类、方法和属性，在 Eclipse 中可通过按 F3 键，方便地查看类、方法和变量的声明和定义的源代码。

有时候在查看一些在 JDK 库中声明 / 定义的类、方法和变量的源代码时，打开的却是相应的 CLASS 文件（字节码），为此 Eclipse 提供了一个功能，把字节码和源代码关联起来，这样，就可以查看（提供源代码）第三方库的实现了。

图 1-5 Eclipse 下载页

Eclipse 打开字节码文件时，可以单击“Attach Source”按钮进行字节码和源代码关联，如图 1-6 所示。

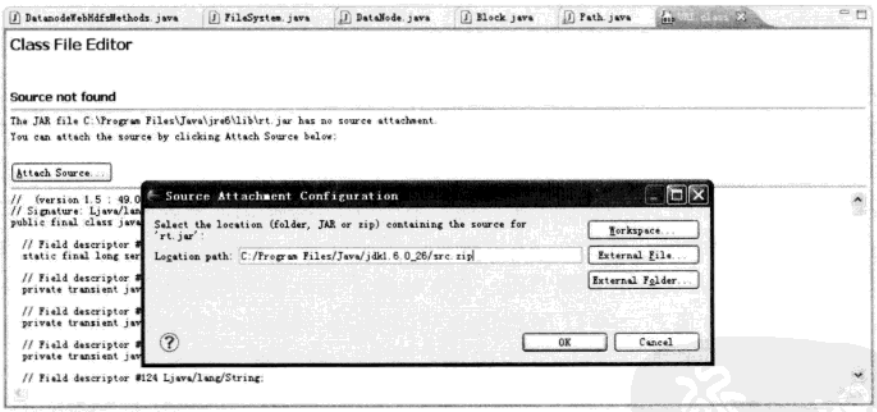



图 1-6 字节码和源代码关联

在查看 java.net.URL 时，Eclipse 提示代码关联，将 JDK 中附带的 JDK 源文件压缩包（在安装目录下可以找到，名字是“src.zip”）绑定到“rt.jar”，以后，只要访问该 JAR 包中的字节码文件，Eclipse 就会自动显示相应的源代码文件。

其他第三方 Java 插件的源代码文件的载入方法类似。

### 2. 根据类名查找相应的类

如果知道希望在编辑器中打开的 Java 类的名称，则找到并打开它的最简单的方法是使用

快捷键 Ctrl+Shift+T（或者单击 Navigate → Open Type）打开 Open Type 窗口，在该窗口中输入名称，Eclipse 将显示可以找到的匹配类型列表。图 1-7 显示了 Hadoop 1.0 中名字包含“HDFS”的所有类。

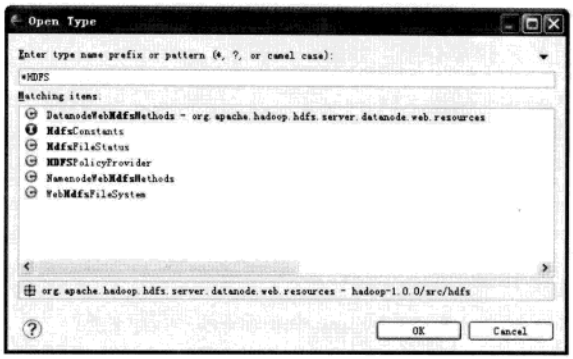



图 1-7　在 Eclipse 中查看名字包含 HDFS 的所有类

**注意**　除了输入完整的类名之外，还可以使用“*”和“?”通配符来分别匹配“任何”或“单个”字符。

### 3. 查看类的继承结构

Java 是面向对象的程序设计语言，继承是面向对象的三大特性之一，了解类、接口在继承关系上的位置，可以更好地了解代码的工作原理。选中某个类并使用 Ctrl + T 快捷键（或单击 Navigate → Quick Type Hierarchy）可以显示类型层次结构。

层次结构将显示所选元素的子类型。如图 1-8 所示，该列表显示已知的所有 org.apache.hadoop.fs.FileSystem 子类。

#### 4. 分析 Java 方法的调用关系

在 Eclipse 中可以分析 Java 方法的调用关系，具体做法如下：在代码区中选择相应的方法定义，然后用鼠标右键选取 Open Call Hierarchy 项或者使用快捷键 Ctrl+Alt+H，则可以在 Call Hierarchy 视图中看到方法的调用关系，该视图还提供了一层一层的方法调用追溯功能，对查找方法的相互调用关系非常有用，如图 1-9 所示。

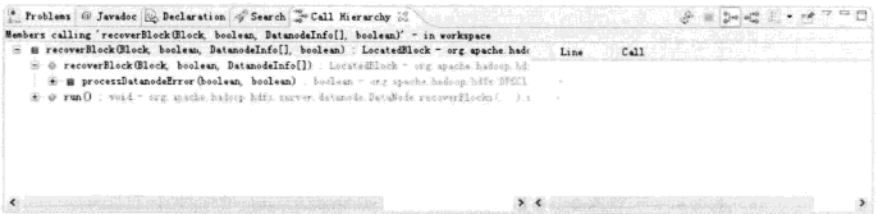



图 1-9 在 Eclipse 中查看方法的调用关系

**注意** 快捷键是日常开发调试中最为便捷的技巧。Eclipse 中的快捷键也可谓是博大精深，这里不一一列举。读者可以在实际开发中不断摸索并牢记这些快捷键，因为它们也是日常开发中必不可少的内容。读者也可参照 Eclipse 中的这些快捷键，在其他 IDE 中找到相应的快捷键设置。

### 1.2.3 安装辅助工具 Ant

在安装和配置了 JDK 和 Eclipse 后，为了编译 Hadoop，还需要安装辅助工具 Ant。

对 Hadoop 这样复杂的项目进行构建，不是仅仅将 Java 源文件编译并打包这么简单，项目中使用到的各种资源都需要得到合理的安排，如有些文件需要拷贝到指定位置，有些类需要放入某个 JAR 归档文件，而另外一些类则需要放入另外一个 JAR 归档文件等，这些工作如果全部由手工执行，项目的构建部署将会变得非常困难，而且难免出错。Ant 是针对这些问题推出的构建工具，在 Java 的项目中得到了最广泛的使用。

Ant 跨平台、可扩展，而且运行高效，使用 Ant，开发人员只需要编写一个基于 XML 的配置文件（文件名一般为 build.xml），定义各种构建任务，如复制文件、编译 Java 源文件、打包 JAR 归档文件等，以及这些构建任务间的依赖关系，如构建任务“打包 JAR 归档文件”需要依赖另外一个构建任务“编译 Java 源文件”。Ant 会根据这个文件中的依赖关系和构建任务，对项目进行构建、打包甚至部署。

和 Hadoop 一样，Ant 也是 Apache 基金会支持的项目，可以在 http://ant.apache.org/bindownload.cgi 下载，下载页面如图 1-10 所示。

和 Eclipse 类似，Ant 也是绿色软件，不需要安装，解压缩下载的文件后需要做一些配置，用户需要添加环境变量 ANT_HOME（指向 Ant 的根目录），并修改环境变量 PATH（在 Windows 环境下，添加 %ANT_HOME%\bin 到 PATH 中）。安装并配置完成后，可以在命令

行窗口中输入“ant -version”命令来检测 Ant 是否被正确设置。

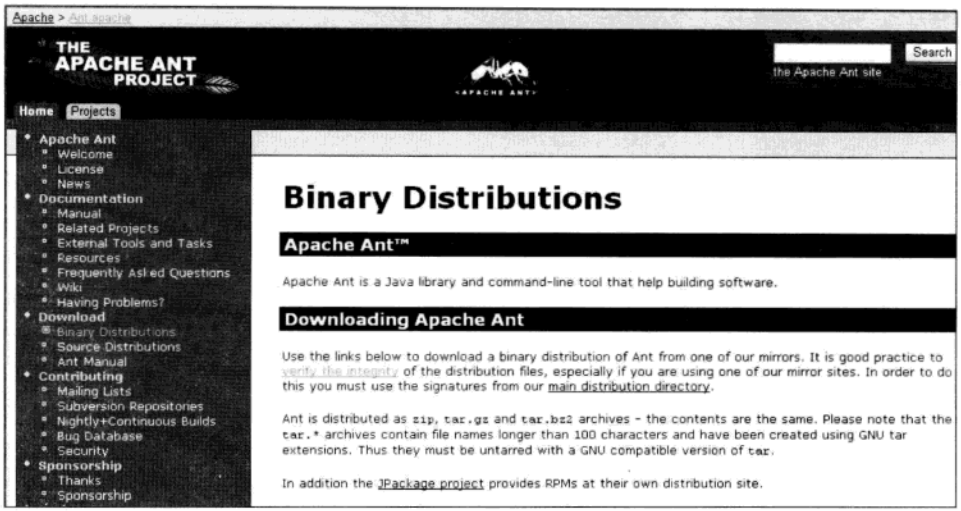



图 1-10　Apache Ant 下载页面

Hadoop 的 Ant 还使用了一个工具：Apache Ivy，它是 Ant 的一个子项目，用于管理项目的外部构建依赖项。外部构建依赖项是指软件开发项目的构建需要依靠来自其他项目的源代码或 JAR 归档文件，例如，Hadoop 项目就依靠 log4j 作为日志记录工具，这些外部依赖项使得构建软件变得复杂。对于小项目而言，一种简单可行的方法是将其依赖的全部项目（JAR 文件）放入一个目录（一般是 lib）中，但当项目变得庞大以后，这种方式就会显得很笨拙。Apache 的另外一个构建工具 Maven 中，引入了 JAR 文件公共存储库的概念，通过外部依赖项声明和公开的公共存储库（通过 HTTP 协议）访问，自动查找外部依赖项并下载，以满足构建时的依赖需要。

Ivy 提供了 Ant 环境下最一致、可重复、易于维护的方法，来管理项目的所有构建依赖项。和 Ant 类似，Ivy 也需要开发人员编写一个 XML 形式的配置文件（一般文件名为 ivy.xml），列举项目的所有依赖项；另外还要编写一个 ivysettings.xml 文件（可以随意为此文件命名），用于配置下载依赖关系的 JAR 文件的存储库。通过 Ant 的两个 Ivy 任务 ivy:settings 和 ivy:retrieve，就可以自动查找依赖项并下载对应的 JAR 文件。

### 1.2.4　安装类 UNIX Shell 环境 Cygwin

对于在 Windows 上工作的读者，还需要准备类 UNIX Shell 环境的 Cygwin。

**注意**　在 Linux 等类 UNIX 系统中进行 Hadoop 代码分析、构建的读者可以略过这一节。

Cygwin 是用于 Windows 的类 UNIX Shell 环境，由两个组件组成：UNIX API 库（它模

拟 UNIX 操作系统提供的许多特性），以及在此基础上的 Bash Shell 改写版本和许多 UNIX 实用程序，它们一起提供了大家熟悉的 UNIX 命令行界面。

Cygwin 的安装程序 setup.exe 是一个标准的 Windows 程序，通过它可以安装或重新安装软件，以及添加、修改或升级 Cygwin 组件。其下载页面为 http://cygwin.com/index.html，如图 1-11 所示。

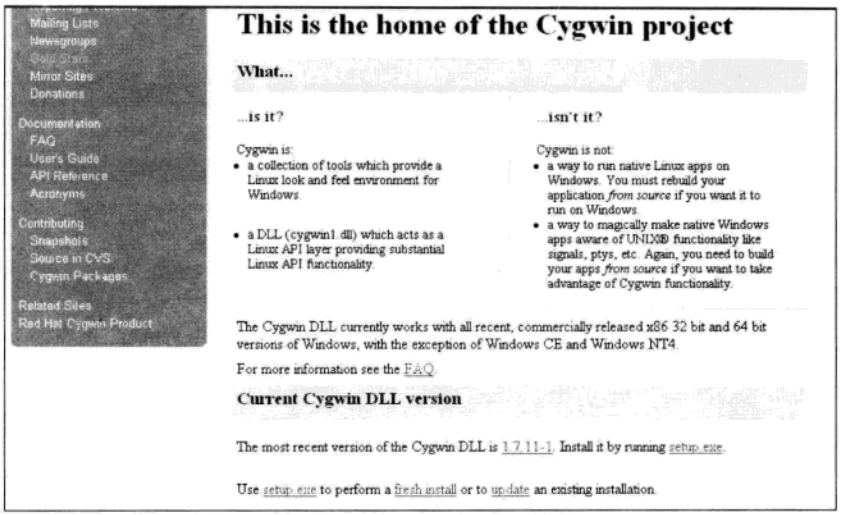



图 1-11 Cygwin 下载页面

执行安装程序 setup.exe，并在安装程序的步骤 4（Cygwin Setup – Select Package）中选择 UNIX 的在线编辑器 sed，如图 1-12 所示（可以利用 Search 输入框快速找到 sed）。

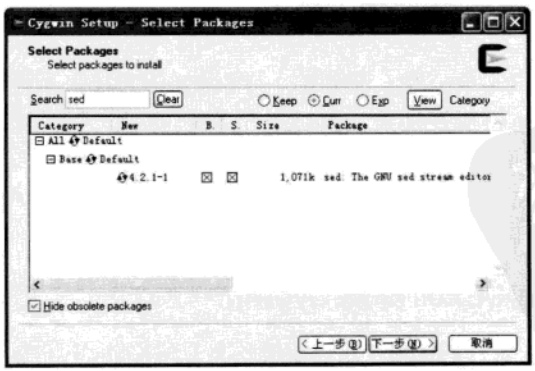



图 1-12 Cygwin 中选择在线编辑器 sed

在安装 sed 时，setup.exe 会自动安装它依赖的包。在 Cygwin 中，可用的包超过 1000 个，所以只需选择需要的类别和包，以后随时可以通过再次运行 setup.exe，添加整个类别或单独

的包。在 Windows 下构建 Hadoop，只需要文本处理工具 sed。

安装完成后，使用 Start 菜单或双击 Cygwin 图标启动 Cygwin。可以在 Shell 环境中执行“ant -version | sed "s/version/Version/g"”命令验证 Cygwin 环境，如图 1-13 所示。

图 1-13　Cygwin 安装验证

成功安装 JDK、Eclipse、Ant 和 Cygwin 之后，就可以开始准备 Hadoop 源代码分析的 Eclipse 环境了。

## 1.3　准备 Hadoop 源代码

在 Hadoop 的官方网站（http://hadoop.apache.org/）中，可以找到 Hadoop 项目相关的信息，如图 1-14 所示。

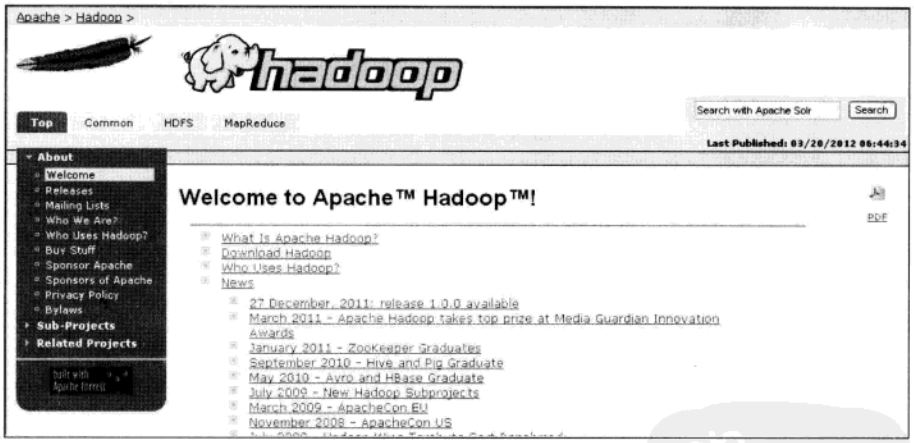



图 1-14　Apache Hadoop 官方网站

### 1.3.1　下载 Hadoop

前面在介绍 Hadoop 生态系统的时候，已经了解到 Hadoop 发展初期的系统中包括 Common（开始使用的名称是 Core）、HDFS 和 MapReduce 三部分，现在这些子系统都已经独立，成为 Apache 的子项目。但在 Hadoop 1.0 的发行包中，Common、HDFS 和 MapReduce 还是打包在一起，我们只需要下载一个 hadoop-1.0.0.tar.gz 包即可。注意，Hadoop 官方也提供 Subversion（SVN）方式的代码下载，SVN 地址为 http://svn.apache.org/repos/asf/hadoop/common/tags/release-0.1.0/。

熟悉 Subversion 的读者，也可以通过该地址下载 Hadoop1.0 版本代码，该 Tag 也包含了上述三部分的代码。

Apache 提供了大量镜像网站，供大家下载它的软件和源码，上面提到的 hadoop-1.0.0.tar.gz 的一个下载地址为 http://apache.etoak.com/hadoop/common/hadoop-1.0.0，如图 1-15 所示。

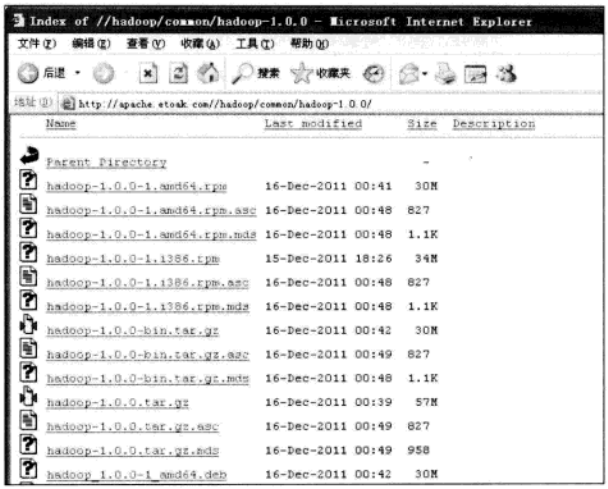



图 1-15 Apache Hadoop 1.0 的一个下载页

该地址包含了 Hadoop 1.0 的多种发行方式，如 64 位系统上的 hadoop-1.0.0-1.adm64.rpm、不包含源代码的发行包 hadoop-1.0.0.bin.tar.gz 等。下载的 hadoop-1.0.0.tar.gz 是包括源代码的 Hadoop 发行包。

## 1.3.2 创建 Eclipse 项目

解压下载的 hadoop-1.0.0.tar.gz 包，假设解压后 Hadoop 的根目录是 E:\hadoop-1.0.0，启动 Cygwin，进入项目的根目录，我们开始将代码导入 Eclipse。Hadoop 的 Ant 配置文件 build.xml 中提供了 eclipse 任务，该任务可以为 Hadoop 代码生成 Eclipse 项目文件，免去创建 Eclipse 项目所需的大量配置工作。只需在 Cygwin 下简单地执行“ant eclipse”命令即可，如图 1-16 所示。

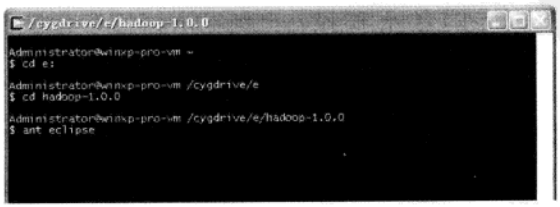



图 1-16 创建 Eclipse 项目文件

**注意**　该过程需要使用 UNIX 的在线编辑器 sed，所以一定要在 Cygwin 环境里执行上述命令，否则会出错。

命令运行结束后，就可以在 Eclipse 中创建项目了。打开 Eclipse 的 File → New → Java Project，创建一个新的 Java 项目，选择项目的位置为 Hadoop 的根目录，即 E:\hadoop-1.0.0，然后单击“Finish”按钮，就完成了 Eclipse 项目的创建，如图 1-17 所示。

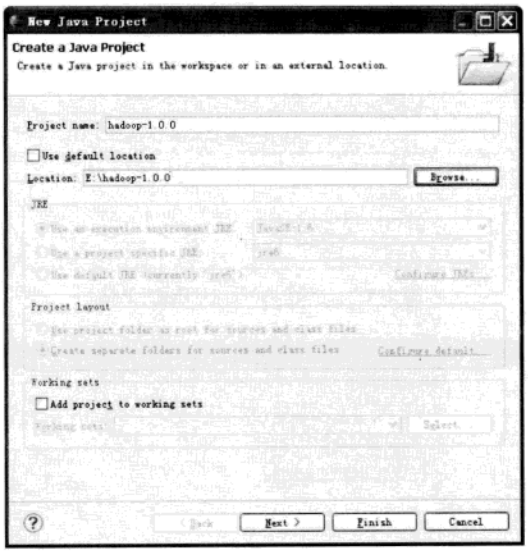



图 1-17　创建 Eclipse 项目

完成上述工作以后，Eclipse 提示一个错误：“Unbound classpath variable: 'ANT_HOME/lib/ant.jar' in project 'hadoop-1.0.0'”。

显然，我们需要设置系统的 ANT_HOME 变量，让 Eclipse 能够找到编译源码需要的 Ant 库，选中项目，然后打开 Eclipse 的 Project → Properties → Java Build Path，在 Libraries 页编辑（单击“Edit”按钮）出错的项：ANT_HOME/lib/ant.jar，创建变量 ANT_HOME（在接下来第一个对话框里单击“Varliable”，第二个对话框里单击“New”按钮），其值为 Ant 的安装目录，如图 1-18 所示。

由于本书只分析 Common 和 HDFS 两个模块，在 Project → Properties → Java Build Path 的 Source 页只保留两个目录，分别是 core 和 hdfs，如图 1-19 所示。

完成上述操作以后，创建 Eclipse 项目的任务就完成了。

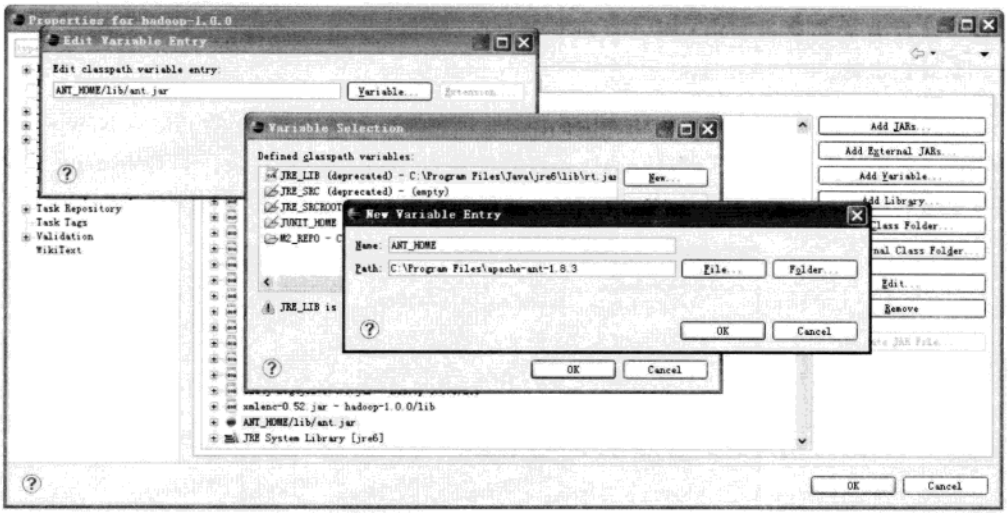



图 1-18 创建 ANT_HOME 变量

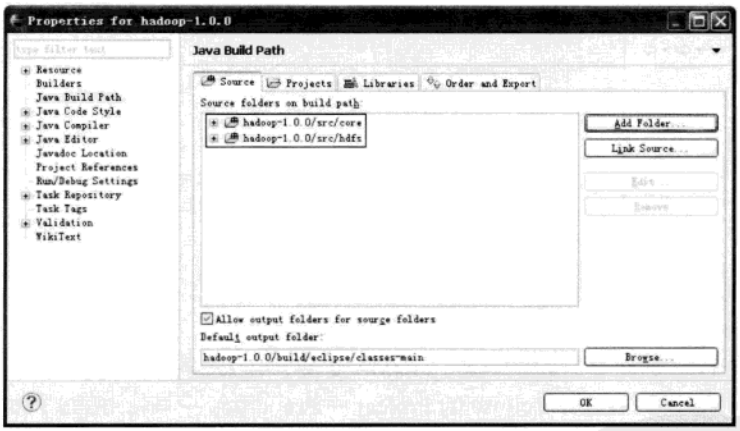



图 1-19 保留 core 和 hdfs 两个源码目录

### 1.3.3 Hadoop 源代码组织

打开已经解压的 Hadoop 1.0 源代码，进入 src 目录，该目录包含了 Hadoop 中所有的代码，如图 1-20 所示。

前面已经提到过，Hadoop 1.0 的发行包中，Common、HDFS 和 MapReduce 三个模块还是打包在一起的，它们的实现分别位于 core、hdfs 和 mapred 子目录下。源代码目录 src 下还有若干值得关注的子目录，具体如下。

❑tools：包含 Hadoop 的一些实用工具的实现，如存档文件 har、分布式拷贝工具

distcp、MapReduce 执行情况分析工具 rumen 等。

- benchmarks：包含对 Hadoop 进行性能测试的两个工具 gridmix 和 gridmix2，通过这些工具，可以测试 Hadoop 集群的一些性能指标。
- c++：需要提及的是 libhdfs，它通过 Java 的 C 语言库界面，实现了一套访问 HDFS 的 C 接口。
- examples：为开发人员提供了一些使用 Hadoop 的例子，不过这些例子只涉及 MapReduce 的 API，本书中不会讨论这部分内容。
- contrib：是 contribution 的缩写，包含大量 Hadoop 辅助模块的实现，如在亚马逊弹性计算云上部署、运行 Hadoop 所需的脚本就在 contrib\ec2 目录下。
- test：包含项目的单元测试用例，在该目录中能找到 Common、HDFS 和 MapReduce 等模块的单元测试代码。

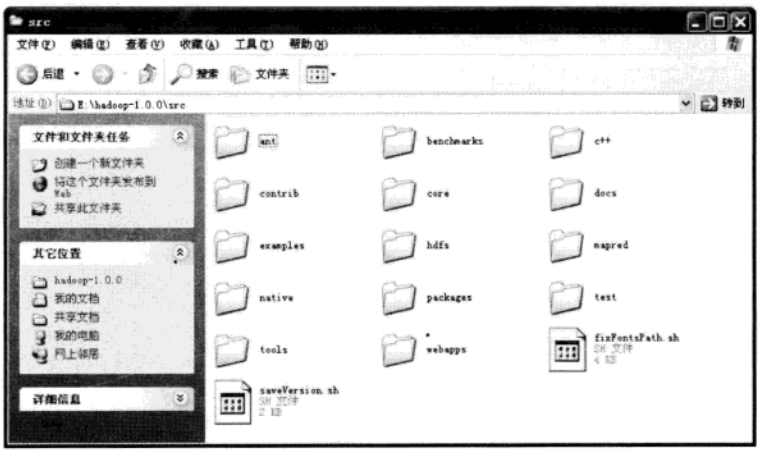



图 1-20　Hadoop 源码组织

## 1.4　小结

大数据以及相关的概念、技术是业界和学界最近关注的热点内容，Hadoop 在其中扮演了十分重要的角色。本节首先对 Hadoop 进行了简单的介绍，展示了蓬勃发展的 Hadoop 生态系统和 Hadoop 的简单历史。并在此基础上，向读者介绍了阅读分析 Hadoop 所必需的开发环境的搭建过程，包括：安装与配置 JDK、安装与配置 Eclipse、安装与配置辅助工具的工作。最后，在上述环境中，从零开始建立了一个包含 Hadoop Common 和 HDFS 的源码环境，为进一步学习 Hadoop 源代码做好准备。

# 第二部分

# Common的实现

## 本部分内容

- Hadoop配置信息处理
- 序列化与压缩
- Hadoop远程过程调用
- Hadoop文件系统

# 第 2 章　Hadoop 配置信息处理

任何一个复杂的软件系统，为了提高其适应性和扩展性，一般都会有一个配置模块或配置系统，作为其扩展、定制的手段和方式。Hadoop 使用配置文件将系统中的重要属性以文件的形式进行持久化，使得这些属性可以被重启后的进程或者不同的进程使用。

## 2.1　配置文件简介

配置文件是一个灵活系统不可缺少的一部分，虽然配置文件非常重要，但却没有标准。本节我们来了解 Windows 操作系统和 Java 环境中的配置文件。

### 2.1.1　Windows 操作系统的配置文件

Windows 系统广泛使用一种特殊化的 ASCII 文件（以“ini”为文件扩展名）作为它的主要配置文件标准。下面是 INI 文件的片段：

```
; 最后修改时间：2012.10.12
[owner]
name=John Doe
organization=Acme Widgets Inc.

[database]
server=192.0.2.62 ; 使用 IP 地址，在域名解析不能使用时还能正常工作
port=143
file="payroll.dat"

[ftp]
```

该文件也称为初始化文件（Initialization File，它的扩展名就是 initialization 的前三个字母）或概要文件（profile），应用程序可以拥有自己的配置文件，存储应用的设置信息，也可以访问 Windows 的基本系统配置文件 win.ini 中存储的配置信息。INI 文件将配置信息分为“节”，节标题放在方括号中。如上面例子中的 [database]，就是 database 节的节标题。节用于对配置数据做一个归类，每一个节可以包含一些与之相关的“项”（ENTRY），并通过等号对其进行赋值（VALUE）。一般的形式如下：

```
[SECTION]
ENTRY=VALUE
```

其中 VALUE 值可以有两种类型：数型或字符串。上面给出的 INI 文件片段中，database 节中包含 3 个项，分别是 server、port 和 file。其中，配置项 port 可以以数型的形式读取。

INI 文件中的注释以分号开始，到行尾结束。

Windows 操作系统同时还提供了一些 API，用来对配置文件进行读、写。如使用 GetProfileString() 函数可以从配置文件 win.ini 中获取字符串型配置，使用 GetPrivateProfileInt() 函数可以从私有的配置文件中读取一个配置整数型项。该函数的原型如下：

```
UINT WINAPI GetPrivateProfileInt(
    __in  LPCTSTR lpAppName,
    __in  LPCTSTR lpKeyName,
    __in  INT nDefault,
    __in  LPCTSTR lpFileName
);
```

其中，参数 LPCTSTR lpFileName 是 INI 文件的文件名，LPCTSTR lpAppName 和 LPCTSTR lpKeyName 分别是上述的“节”和“项”，INT nDefault 是默认值，也就是说，如果在配置文件中找不到配置信息，就返回该默认值。

## 2.1.2　Java 配置文件

JDK 提供了 java.util.Properties 类，用于处理简单的配置文件。Properties 很早就被引入到 Java 的类库中，并且一直没有什么变化。它继承自 Hashtable，如图 2-1 所示，表示了一个持久的属性集，该集可保存在流中或从流中加载。属性列表中每个键及其对应值都是字符串类型。

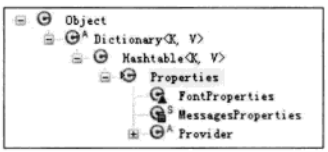



图 2-1　Properties 的继承关系

相对于 INI 文件，Properties 处理的配置文件格式非常简单，它只支持键 – 值对，等号“=”左边为键，右边为值。形式如下：

```
ENTRY=VALUE
```

由于 Properties 基于 Hashtable，它并不能支持 INI 文件中的“节”，对配置项进行分类。

java.util.Properties 中用于处理属性列表的主要方法如下，其中，getProperty() 用于在属性列表中获取指定键（参数 key）对应的属性，它有两个形式，一个不提供默认值，另一个可以提供默认值。Properties.setProperty() 用于在属性列表中设置 / 更新属性值。相关代码如下：

```
// 用指定的键在此属性列表中搜索属性
public String getProperty(String key)

// 功能同上，参数 defaultValue 提供了默认值
public String getProperty(String key, String defaultValue)

// 最终调用 Hashtable 的方法 put
public synchronized Object setProperty(String key, String value)
```

Properties 中的属性通过 load() 方法加载，该方法从输入流中读取键 – 值对，而 store() 方法则将 Properties 表中的属性列表写入输出流。使用输入流和输出流，Properties 对象不但可以保存在文件中，而且还可以保存在其他支持流的系统中，如 Web 服务器。J2SE 1.5 版本以后，Properties 中的数据也可以以 XML 格式保存，对应的加载和写出方法是

loadFromXML() 和 storeToXML()。

下面是以 XML 格式存在的 Properties 配置文件的例子。

```
<?xml?version="1.0"?encoding="UTF-8"?>
<!DOCTYPE?properties?SYSTEM
       "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>Hi</comment>
    <entry?key="foo">bar</entry>
    <entry?key="fu">baz</entry>
</properties>
```

由于 java.util.Properties 提供的能力有限，Java 社区中出现了大量的配置信息读 / 写方案，其中比较有名的是 Apache Jakarta Commons 工具集中提供的 Commons Configuration。Commons Configuration 中的 PropertiesConfiguration 类提供了丰富的访问配置参数的方法。Commons Configuration 支持文本、XML 配置文件格式；支持加载多个配置文件；支持分层或多级的配置；同时提供对单值或多值配置参数的基于类型的访问。应该说，Commons Configuration 是一个功能强大的配置文件处理工具。

## 2.2 Hadoop Configuration 详解

Hadoop 没有使用 java.util.Properties 管理配置文件，也没有使用 Apache Jakarta Commons Configuration 管理配置文件，而是使用了一套独有的配置文件管理系统，并提供自己的 API，即使用 org.apache.hadoop.conf.Configuration 处理配置信息。

### 2.2.1 Hadoop 配置文件的格式

Hadoop 配置文件采用 XML 格式，下面是 Hadoop 配置文件的一个例子：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
   <property>
      <name>io.sort.factor</name>
      <value>10</value>
      <description>The number of streams to merge at once while sorting
      files.  This determines the number of open file handles.</description>
   </property>

   <property>
      <name>dfs.name.dir</name>
      <value>${hadoop.tmp.dir}/dfs/name</value>
      <description>Determines where on the local filesystem the DFS name
      nodeshould store the name table(fsimage).  ……</description>
   </property>
```

```
    <property>
      <name>dfs.web.ugi</name>
      <value>webuser,webgroup</value>
      <final>true</final>
      <description>The user account used by the web interface.
      Syntax: USERNAME,GROUP1,GROUP2, ……</description>
    </property>
</configuration>
```

Hadoop 配置文件的根元素是 configuration，一般只包含子元素 property。每一个 property 元素就是一个配置项，配置文件不支持分层或分级。每个配置项一般包括配置属性的名称 name、值 value 和一个关于配置项的描述 description；元素 final 和 Java 中的关键字 final 类似，意味着这个配置项是“固定不变的”。final 一般不出现，但在合并资源的时候，可以防止配置项的值被覆盖。

在上面的示例文件中，配置项 dfs.web.ugi 的值是“webuser,webgroup”，它是一个 final 配置项；从 description 看，这个配置项配置了 Hadoop Web 界面的用户账号，包括用户名和用户组信息。这些信息可以通过 Configuration 类提供的方法访问。

在 Configuration 中，每个属性都是 String 类型的，但是值类型可能是以下多种类型，包括 Java 中的基本类型，如 boolean（getBoolean）、int（getInt）、long（getLong）、float（getFloat），也可以是其他类型，如 String（get）、java.io.File（getFile）、String 数组（getStrings）等。以上面的配置文件为例，getInt("io.sort.factor") 将返回整数 10；而 getStrings("dfs.web.ugi") 返回一个字符串数组，该数组有两个元素，分别是 webuser 和 webgroup。

合并资源指将多个配置文件合并，产生一个配置。如果有两个配置文件，也就是两个资源，如 core-default.xml 和 core-site.xml，通过 Configuration 类的 loadResources() 方法，把它们合并成一个配置。代码如下：

```
Configurationconf = new Configuration();
conf.addResource("core-default.xml");
conf.addResource("core-site.xml");
```

如果这两个配置资源都包含了相同的配置项，而且前一个资源的配置项没有标记为 final，那么，后面的配置将覆盖前面的配置。上面的例子中，core-site.xml 中的配置将覆盖 core-default.xml 中的同名配置。如果在第一个资源（core-default.xml）中某配置项被标记为 final，那么，在加载第二个资源的时候，会有警告提示。

Hadoop 配置系统还有一个很重要的功能，就是属性扩展。如配置项 dfs.name.dir 的值是 ${hadoop.tmp.dir}/dfs/name，其中，${hadoop.tmp.dir} 会使用 Configuration 中的相应属性值进行扩展。如果 hadoop.tmp.dir 的值是“/data”，那么扩展后的 dfs.name.dir 的值就是“/data/dfs/name”。

使用 Configuration 类的一般过程是：构造 Configuration 对象，并通过类的 addResource() 方法添加需要加载的资源；然后就可以使用 get* 方法和 set* 方法访问 / 设置配置项，资源会在第一次使用的时候自动加载到对象中。

## 2.2.2 Configuration 的成员变量

org.apache.hadoop.conf.Configuration 类图如图 2-2 所示。

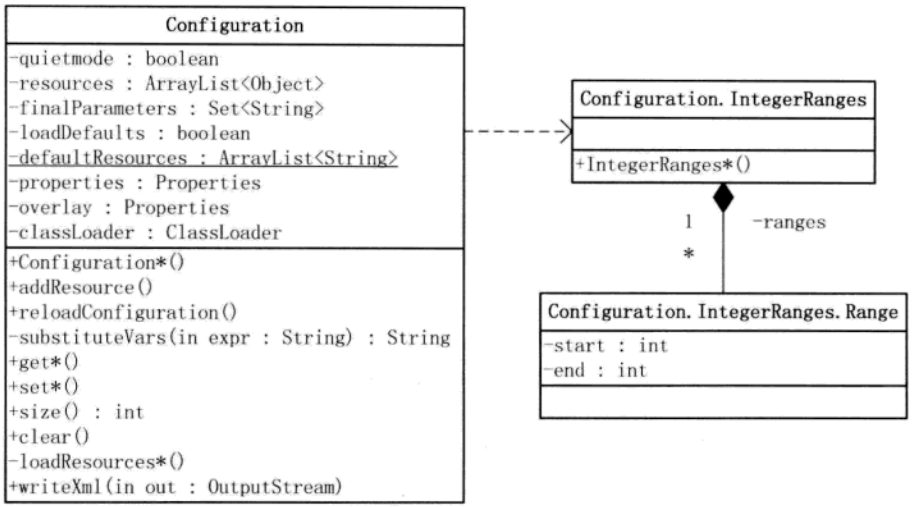



图 2-2 Configuration 类图

从类图可以看到，Configuration 有 7 个主要的非静态成员变量。

布尔变量 quietmode，用来设置加载配置的模式。如果 quietmode 为 true（默认值），则在加载解析配置文件的过程中，不输出日志信息。quietmode 只是一个方便开发人员调试的变量。

数组 resources 保存了所有通过 addResource() 方法添加 Configuration 对象的资源。Configuration.addResource() 有如下 4 种形式：

```
public void addResource(InputStream in)
public void addResource(Path file)
public void addResource(String name)   //CLASSPATH 资源
public void addResource(URL url)
```

也就是说，用户可以添加如下形式的资源：

❑一个已经打开的输入流 InputStream；

❑Hadoop 文件路径 org.apache.hadoop.fs.Path 形式（后面会讨论 Path 类）的资源，如 hdfs:// www.example.com:54300/conf/core-default.xml；

❑URL，如 http://www.example.com/core-default.xml；

❑CLASSPATH 资源（String 形式），前面提到的“core-default.xml”就是这种形式。

布尔变量 loadDefaults 用于确定是否加载默认资源，这些默认资源保存在 defaultResources 中。注意，defaultResources 是个静态成员变量，通过方法 addDefaultResource() 可以添加系统的默认资源。在 HDFS 中，会把 hdfs-default.xml 和 hdfs-site.xml 作为默认资源，并通过

addDefaultResource() 保存在成员变量 defaultResources 中；在 MapReduce 中，默认资源是 mapred-default.xml 和 mapred-site.xml。如 HDFS 的 DataNode 中，就有下面的代码，加载上述两个默认资源：

```
// 下面的代码来自 org.apache.hadoop.hdfs.server.datanode.DataNode
static{
Configuration.addDefaultResource("hdfs-default.xml");
Configuration.addDefaultResource("hdfs-site.xml");
}
```

properties、overlay 和 finalParameters 都是和配置项相关的成员变量。其中，properties 和 overlay 的类型都是前面介绍过的 java.util.Properties。Hadoop 配置文件解析后的键 – 值对，都存放在 properties 中。变量 finalParameters 的类型是 Set<String>，用来保存所有在配置文件中已经被声明为 final 的键 – 值对的键，如前面配置文件例子中的键“dfs.web.ugi”。变量 overlay 用于记录通过 set() 方式改变的配置项。也就是说，出现在 overlay 中的键 – 值对是应用设置的，而不是通过对配置资源解析得到的。

Configuration 中最后一个重要的成员变量是 classLoader，这是一个类加载器变量，可以通过它来加载指定类，也可以通过它加载相关的资源。上面提到 addResource() 可以通过字符串方式加载 CLASSPATH 资源，它其实通过 Configuration 中的 getResource() 将字符串转换成 URL 资源，相关代码如下：

```
public URL getResource(String name) {
return classLoader.getResource(name);
}
```

其中，getResource() 用于根据资源的名称查找相应的资源，并返回读取资源的 URL 对象。

**注意**　这里的资源，指的是可以通过类代码以与代码基无关的方式访问的一些数据，如图像、声音、文本等，不是前面提到的配置资源。

了解了 Configuration 各成员变量的具体含义，Configuration 类的其他部分就比较容易理解了，它们都是为了操作这些变量而实现的解析、设置、获取方法。

### 2.2.3　资源加载

资源通过对象的 addResource() 方法或类的静态 addDefaultResource() 方法（设置了 loadDefaults 标志）添加到 Configuration 对象中，添加的资源并不会立即被加载，只是通过 reloadConfiguration() 方法清空 properties 和 finalParameters。相关代码如下：

```
public void addResource(String name) { // 以 CLASSPATH 资源为例
   addResourceObject(name);
}

private synchronized void addResourceObject(Object resource) {
   resources.add(resource);// 添加到成员变量 resources 中
```

```
    reloadConfiguration();
}

public synchronized void reloadConfiguration() {
    properties = null;// 会触发资源的重新加载
    finalParameters.clear();
}
```

静态方法 addDefaultResource() 也能清空 Configuration 对象中的数据（非静态成员变量），这是通过类的静态成员 REGISTRY 作为媒介进行的。

静态成员 REGISTRY 记录了系统中所有的 Configuration 对象，所以，addDefaultResource() 被调用时，遍历 REGISTRY 中的元素并在元素（即 Configuration 对象）上调用 reloadConfiguration() 方法，即可触发资源的重新加载，相关代码如下：

```
public static synchronized void addDefaultResource(String name) {
    if(!defaultResources.contains(name)) {
        defaultResources.add(name);
        for(Configuration conf : REGISTRY.keySet()) {
            if(conf.loadDefaults) {
                conf.reloadConfiguration(); // 触发资源的重新加载
            }
        }
    }
}
```

成员变量 properties 中的数据，直到需要的时候才会加载进来。在 getProps() 方法中，如果发现 properties 为空，将触发 loadResources() 方法加载配置资源。这里其实采用了延迟加载的设计模式，当真正需要配置数据的时候，才开始分析配置文件。相关代码如下：

```
private synchronized Properties getProps() {
    if (properties == null) {
        properties = new Properties();
        loadResources(properties, resources, quietmode);
        ......
    }
}
```

Hadoop 的配置文件都是 XML 形式，JAXP（Java API for XML Processing）是一种稳定、可靠的 XML 处理 API，支持 SAX（Simple API for XML）和 DOM（Document Object Model）两种 XML 处理方法。

SAX 提供了一种流式的、事件驱动的 XML 处理方式，但编写处理逻辑比较复杂，比较适合处理大的 XML 文件。

DOM 和 SAX 不同，其工作方式是：首先将 XML 文档一次性装入内存；然后根据文档中定义的元素和属性在内存中创建一个“树形结构”，也就是一个文档对象模型，将文档对象化，文档中每个节点对应着模型中一个对象；然后使用对象提供的编程接口，访问 XML 文档进而操作 XML 文档。由于 Hadoop 的配置文件都是很小的文件，因此 Configuration 使

用 DOM 处理 XML。

首先分析 DOM 加载部分的代码：

```
private void loadResource(Properties properties,
      Object name, boolean quiet) {
   try {
      // 得到用于创建 DOM 解析器的工厂
      DocumentBuilderFactorydocBuilderFactory
            = DocumentBuilderFactory.newInstance();

      // 忽略 XML 中的注释
      docBuilderFactory.setIgnoringComments(true);
      // 提供对 XML 名称空间的支持
      docBuilderFactory.setNamespaceAware(true);
      try {
         // 设置 XInclude 处理状态为 true，即允许 XInclude 机制
         docBuilderFactory.setXIncludeAware(true);
      } catch (UnsupportedOperationException e) {
         ……
      }

      // 获取解析 XML 的 DocumentBuilder 对象
      DocumentBuilder builder = docBuilderFactory.newDocumentBuilder();
      Document doc = null;
      Element root = null;

      // 根据不同资源，做预处理并调用相应形式的 DocumentBuilder.parse
      if (name instanceof URL) {// 资源是 URL 形式
         ……
         doc = builder.parse(url.toString());
         ……
      } else if (name instanceof String) {//CLASSPATH 资源
         ……
      } else if (name instanceof Path) {// 资源是 Hadoop Path 形式的
         ……
      } else if (name instanceof InputStream) {//InputStream
         ……
      } else if (name instanceof Element) {// 处理 configuration 子元素
         root = (Element)name;
      }

      if (doc == null && root == null) {
         if (quiet)
            return;
         throw new RuntimeException(name + " not found");
   }
   ……
```

一般的 JAXP 处理都是从工厂开始，通过调用 DocumentBuilderFactory 的 newInstance() 方法，获得用于创建 DOM 解析器的工厂。这里并没有创建出 DOM 解析器，只是获

得一个用于创建DOM解析器的工厂，接下来需要对上述newInstance()方法得到的docBuilderFactory对象进行一些设置，才能进一步通过DocumentBuilderFactory，得到DOM解析器对象builder。

针对DocumentBuilderFactory对象进行的主要设置包括：

- 忽略XML文档中的注释；
- 支持XML空间；
- 支持XML的包含机制（XInclude）。

XInclude机制允许将XML文档分解为多个可管理的块，然后将一个或多个较小的文档组装成一个大型文档。也就是说，Hadoop的一个配置文件中，可以利用XInclude机制将其他配置文件包含进来一并处理，下面是一个例子：

```
<configuration xmlns:xi="http://www.w3.org/2001/XInclude">
   ......
   <xi:include href="conf4performance.xml"/>
   ......
</configuration>
```

通过XInclude机制，把配置文件conf4performance.xml嵌入到当前配置文件，这种方法更有利于对配置文件进行模块化管理，同时就不需要再使用Configuration.addResource()方法加载资源conf4performance.xml了。

设置完DocumentBuilderFactory对象以后，通过docBuilderFactory.newDocumentBuilder()获得了DocumentBuilder对象，用于从各种输入源解析XML。在loadResource()中，需要根据Configuration支持的4种资源分别进行处理，不过这4种情况最终都调用DocumentBuilder.parse()函数，返回一个DOM解析结果。

如果输入是一个DOM的子元素，那么将解析结果设置为输入元素。这是为了处理下面出现的元素configuration包含configuration子节点的特殊情况。

成员函数loadResource的第二部分代码，就是根据DOM的解析结果设置Configuration的成员变量properties和finalParameters。

在确认XML的根节点是configuration以后，获取根节点的所有子节点并对所有子节点进行处理。这里需要注意，元素configuration的子节点可以是configuration，也可以是properties。如果是configuration，则递归调用loadResource()，在loadResource()的处理过程中，子节点会被作为根节点得到继续的处理。

如果是property子节点，那么试图获取property的子元素name、value和final。在成功获得name和value的值后，根据情况设置对象的成员变量properties和finalParameters。相关代码如下：

```
if (root == null) {
   root = doc.getDocumentElement();
}
// 根节点应该是 configuration
```

```
if (!"configuration".equals(root.getTagName()))
   LOG.fatal("bad conf file: top-level element not <configuration>");

//获取根节点的所有子节点
NodeList props = root.getChildNodes();
for (int i = 0; i <props.getLength(); i++) {
   Node propNode = props.item(i);
   if (!(propNode instanceof Element))
      continue;  //如果子节点不是 Element，忽略

   Element prop = (Element)propNode;
   if ("configuration".equals(prop.getTagName())) {
      //如果子节点是 configuration，递归调用 loadResource 进行处理
      //这意味着 configuration 的子节点可以是 configuration
      loadResource(properties, prop, quiet);
      continue;
   }

   //子节点是 property
   if (!"property".equals(prop.getTagName()))
      LOG.warn("bad conf file: element not <property>");

   NodeList fields = prop.getChildNodes();
      String attr = null;
      String value = null;
      boolean finalParameter = false;

      //查找 name、value 和 final 的值
      for (int j = 0; j <fields.getLength(); j++) {
         Node fieldNode = fields.item(j);
         if (!(fieldNode instanceof Element))
            continue;
         Element field = (Element)fieldNode;
         if ("name".equals(field.getTagName()) &&field.hasChildNodes())
            attr = ((Text)field.getFirstChild()).getData().trim();
         if ("value".equals(field.getTagName()) &&field.hasChildNodes())
            value = ((Text)field.getFirstChild()).getData();
         if ("final".equals(field.getTagName()) &&field.hasChildNodes())
            finalParameter =
               "true".equals(((Text)field.getFirstChild()).getData());
      }

      if (attr != null && value != null) {
         //如果属性已经标志为 'final'，忽略
         if (!finalParameters.contains(attr)) {
            //添加键 - 值对到 properties 中
            properties.setProperty(attr, value);
            if (finalParameter) {
               //该属性标志为 'final'，添加 name 到 finalParameters 中
               finalParameters.add(attr);
```

```
                }
            }
            ......
        }
    }
    // 处理异常
    ......
}
```

### 2.2.4 使用 get* 和 set* 访问 / 设置配置项

1. get*

get* 一共代表 21 个方法，它们用于在 Configuration 对象中获取相应的配置信息。这些配置信息可以是 boolean（getBoolean）、int（getInt）、long（getLong）等基本类型，也可以是其他一些 Hadoop 常用类型，如类的信息（getClassByName、getClasses、getClass）、String 数组（getStringCollection、getStrings）、URL（getResource）等。这些方法里最重要的是 get() 方法，它根据配置项的键获取对应的值，如果键不存在，则返回默认值 defaultValue。其他的方法都会依赖于 Configuration.get()，并在 get() 的基础上做进一步处理。get() 方法如下：

```
public String get(String name, String defaultValue)
```

Configuration.get() 会调用 Configuration 的私有方法 substituteVars()，该方法会完成配置的属性扩展。属性扩展是指配置项的值包含 ${key} 这种格式的变量，这些变量会被自动替换成相应的值。也就是说，${key} 会被替换成以 key 为键的配置项的值。注意，如果 ${key} 替换后，得到的配置项值仍然包含变量，这个过程会继续进行，直到替换后的值中不再出现变量为止。

substituteVars 的工作依赖于正则表达式：

```
varPat: \$\{[^\}\$ ]+\}
```

由于“$”、左花括号“{”、右花括号“}”都是正则表达式中的保留字，因此需要通过“\”进行转义。正则表达式 varPat 中，“\$\{”部分用于匹配 ${key} 中的 key 前面的“${”，最后的“\}”部分匹配属性扩展项的右花括号“}”，中间部分“[^\}\$ ]+”用于匹配属性扩展键，它使用了两个正则表达式规则：

- ❑[^ ] 规则，通过 [^ ] 包含一系列的字符，使表达式匹配这一系列字符以外的任意一个字符。也就是说，“[^\}\$ ]”将匹配除了“}”、“$”和空格以外的所有字符。注意，$ 后面还包含了一个空格，这个看不见的空格，是通过空格的 Unicode 字符 \u0020 添加到表达式中的。
- ❑+ 是一个修饰匹配次数的特殊符号，通过该符号保证了“+”前面的表达式“[^\}\$ ]”至少出现 1 次。

通过正则表达式“\$\{[^\}\$ ]+\}”，可以在输入字符串里找出需要进行属性扩展的地方，

并通过字符串替换，进行属性扩展。

前面提过，如果一次属性扩展完成以后，得到的表达式里仍然包含可扩展的变量，那么，substituteVars() 需要再次进行属性扩展。考虑下面的情况：

属性扩展 ${key1} 的结果包含属性扩展 ${key2}，而对 ${key2} 进行属性扩展后，产生了一个包含 ${key1} 的新结果，这会导致属性扩展进入死循环，没办法停止。

针对这种可能发生的情况，substituteVars() 中使用了一个非常简单而又有效的策略，即属性扩展只能进行一定的次数（20 次，通过 Configuration 的静态成员变量 MAX_SUBST 定义），避免出现上面分析的属性扩展死循环。

最后一点需要注意的是，substituteVars() 中进行的属性扩展，不但可以使用保存在 Configuration 对象中的键 – 值对，而且还可以使用 Java 虚拟机的系统属性。如系统属性 user.home 包含了当前用户的主目录，如果用户有一个配置项需要使用这个信息，可以通过属性扩展 ${user.home}，来获得对应的系统属性值。而且，Java 命令行可以通过“-D<name>=<value>”的方式定义系统属性。这就提供了一个通过命令行，覆盖或者设置 Hadoop 运行时配置信息的方法。在 substituteVars() 中，属性扩展优先使用系统属性，然后才是 Configuration 对象中保存的键 – 值对。具体代码如下：

```
// 正则表达式对象，包含正则表达式 \$\{[^\}\$ ]+\}
// 注意，u0020 前面只有一个"\"，转义发生在 Java 里，不在正则表达式里
private static Pattern varPat =
      Pattern.compile("\\$\\{[^\\}\\$\u0020]+\\}");

// 最多做 20 次属性扩展
private static int MAX_SUBST = 20;

private String substituteVars(String expr) {
   if (expr == null) {
      return null;
   }
   Matcher match = varPat.matcher("");
   String eval = expr;

   // 循环，最多做 MAX_SUBST 次属性扩展
   for(int s=0; s<MAX_SUBST; s++) {
      match.reset(eval);

      if (!match.find()) {
         return eval;  // 什么都没有找到，返回
      }
      String var = match.group();
      var = var.substring(2, var.length()-1); // 获得属性扩展的键
      String val = null;
      try {
         // 看看系统属性里有没有 var 对应的 val
         // 这一步保证了我们首先使用系统属性做属性扩展
         val = System.getProperty(var);
```

```
        } catch(SecurityException se) {
            LOG.warn("Unexpected SecurityException in Configuration", se);
        }

        if (val == null) {
            // 看看 Configuration 保存的键 - 值对里有没有 var 对应的 val
            val = getRaw(var);
        }

        if (val == null) {
            // 属性扩展中的 var 没有绑定，不做扩展，返回
            return eval;
        }

        // 替换 ${……}，完成属性扩展
        eval = eval.substring(0,match.start())
                +val+eval.substring(match.end());
    }

    // 属性扩展次数过多，抛异常
    throw new IllegalStateException(……);
}
```

2. set*

相对于 get* 来说，set* 的大多数方法都很简单，这些方法对输入进行类型转换等处理后，最终都调用了下面的 Configuration.set() 方法：

```
public String set(String name, String value)
```

对比相对复杂的 Configuration.get()，成员函数 set() 只是简单地调用了成员变量 properties 和 overlay 的 setProperty() 方法，保存传入的键 – 值对。

## 2.3 Configurable 接口

Configurable 是一个很简单的接口，也位于 org.apache.hadoop.conf 包中，其类图如图 2-3 所示。

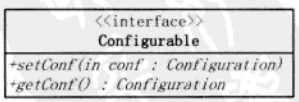



图 2-3 Configurable 类图

从字面理解，Configurable 的含义是可配置的，如果一个类实现了 Configurable 接口，意味着这个类是可配置的。也就是说，可以通过为这个类的对象传入一个 Configuration 实例，提供对象工作需要的一些配置信息。Hadoop 的代码中有大量的类实现了 Configurable 接口，如 org.apache.hadoop.mapred.SequenceFileInputFilter.RegexFilter。RegexFilter 对象工作时，需要提供一个正则表达式，用于过滤读取的记录。由于 RegexFilter 的父类 Filter 中实现的 Configurable 接口，RegexFilter 可以在它的 setConf() 方法中，使用 Configuration.get() 方法获取以字符串传入的正则表达式，并初始化成员变量 p。相关代码如

下：

```
public void setConf(Configuration conf) {
   // 在 conf 中获取键为 "sequencefile.filter.regex" (FILTER_REGEX) 的配置项
   String regex = conf.get(FILTER_REGEX);

   if (regex==null)
      throw new RuntimeException(FILTER_REGEX + "not set");
   this.p = Pattern.compile(regex);
   this.conf = conf;
}
```

Configurable.setConf() 方法何时被调用呢？一般来说，对象创建以后，就应该使用 setConf() 方法，为对象提供进一步的初始化工作。为了简化对象创建和调用 setConf() 方法这两个连续的步骤，org.apache.hadoop.util.ReflectionUtils 中提供了静态方法 newInstance()，代码如下：

```
public static <T> T newInstance(Class<T>theClass, Configuration conf)
```

方法 newInstance() 利用 Java 反射机制，根据对象类型信息（参数 theClass），创建一个新的相应类型的对象，然后调用 ReflectionUtils 中的另一个静态方法 setConf() 配置对象，代码如下：

```
public static void setConf(Object theObject, Configuration conf) {
   if(conf != null) {
      // 传入的对象实现了 Configurable 接口
      if(theObject instanceof Configurable) {
         // 调用对象的 setConf 方法，传入 Configuration 对象
         ((Configurable) theObject).setConf(conf);
      }
      setJobConf(theObject, conf);
   }
}
```

在 setConf() 中，如果对象实现了 Configurable 接口，那么对象的 setConf() 方法会被调用，并根据 Configuration 类的实例 conf 进一步初始化对象。

## 2.4　小结

配置系统是复杂软件必不可少的一部分，作为 Hadoop Common 部分介绍的第一个组件，org.apache.hadoop.conf.Configuration 在 Hadoop 各个子项目中发挥着重要的作用。本章从 Windows 和基于 Java Properties 配置文件开始，分析了 Hadoop 使用的基于健－值对构成的、结构相对简单的 XML 配置文件，以及相应的处理类 Configuration，特别是 Configuration 类中的资源加载、资源合并和属性扩展等比较重要的处理过程。

Hadoop 配置信息处理是学习 Hadoop 源代码的一个很好的起点。

# 第3章　序列化与压缩

传统的计算机系统通过I/O操作与外界进行交流，Hadoop的I/O由传统的I/O系统发展而来，但又有些不同，Hadoop需要处理P、T级别的数据，所以在org.apache.hadoop.io包中包含了一些面向海量数据处理的基本输入输出工具，本章会对其中的序列化和压缩进行研究。

## 3.1　序列化

对象的序列化（Serialization）用于将对象编码成一个字节流，以及从字节流中重新构建对象。“将一个对象编码成一个字节流”称为序列化该对象（Serializing）；相反的处理过程称为反序列化（Deserializing）。

序列化有三种主要的用途：

- 作为一种持久化格式：一个对象被序列化以后，它的编码可以被存储到磁盘上，供以后反序列化用。
- 作为一种通信数据格式：序列化结果可以从一个正在运行的虚拟机，通过网络被传递到另一个虚拟机上。
- 作为一种拷贝、克隆（clone）机制：将对象序列化到内存的缓存区中，然后通过反序列化，可以得到一个对已存对象进行深拷贝的新对象。

在分布式数据处理中，主要使用上面提到的前两种功能：数据持久化和通信数据格式。

在分析Hadoop的序列化机制前，先介绍一下Java内建的序列化机制。

### 3.1.1　Java内建序列化机制

Java序列化机制将对象转换为连续的byte数据，这些数据可以在日后还原为原先的对象状态，该机制还能自动处理不同操作系统上的差异，在Windows系统上序列化的Java对象，可以在UNIX系统上被重建出来，不需要担心不同机器上的数据表示方法，也不需要担心字节排列次序，如大端（big endian）、小端（little endian）或其他细节。

在Java中，使一个类的实例可被序列化非常简单，只需要在类声明中加入implements Serializable即可。Serializable接口是一个标志，不具有任何成员函数，其定义如下：

```
public interface Serializable {
}
```

Serializable接口没有任何方法，所以不需要对类进行修改，Block类通过声明它实现了Serializable接口，立即可以获得Java提供的序列化功能。代码如下：

```
public class Block implements Writable, Comparable<Block>, Serializable
```

由于序列化主要应用在与 I/O 相关的一些操作上，其实现是通过一对输入 / 输出流来实现的。如果想对某个对象执行序列化动作，可以在某种 OutputStream 对象（后面还会讨论 Java 的流）的基础上创建一个对象流 ObjectOutputStream 对象，然后调用 writeObject() 就可达到目的。

writeObject() 方法负责写入实现了 Serializable 接口对象的状态信息，输出数据将被送至该 OutputStream。多个对象的序列化可以在 ObjectOutputStream 对象上多次调用 writeObject()，分别写入这些对象。下面是序列化一个 Block 对象的例子：

```
Block block1=new Block(7806259420524417791L, 39447755L, 56736651L);
Block block2=new Block(5547099594945187683L, 67108864L, 56736828L);
……
ByteArrayOutputStream out=new ByteArrayOutputStream();

// 在 ByteArrayOutputStream 的基础上创建 ObjectOutputStream
ObjectOutputStream objOut=new ObjectOutputStream(out);

// 对 block 进行序列化
objOut.writeObject(block1);
```

对于 Java 基本类型的序列化，ObjectOutputStream 提供了 writeBoolean()、writeByte() 等方法。

输入过程类似，将 InputStream 包装在 ObjectInputStream 中并调用 readObject()，该方法返回一个指向向上转型后的 Object 的引用，通过向下转型，就可以得到正确结果。读取对象时，必须要小心地跟踪存储的对象的数量、顺序以及它们的类型。

Java 的序列化机制非常“聪明”，JavaDoc 中对 ObjectOutputStream 的 writeObject() 方法的说明是：“……这个对象的类、类签名、类的所有非暂态和非静态成员的值，以及它所有的父类都要被写入”，序列化机制会自动访问对象的父类，以保证对象内容的一致性。同时，序列化机制不仅存储对象在内存中的原始数据，还会追踪通过该对象可以到达的其他对象的内部数据，并描述所有这些对象是如何被链接起来的。对于复杂的情形，Java 序列化机制也能应付自如：在输出 objectA 和 objectB 时，不会重复保存对象的序列化结果（如 objectC，即 objectC 只被序列化一次）；对于循环引用的对象，序列化也不会陷入死循环（如图 3-1 右图的情形）。

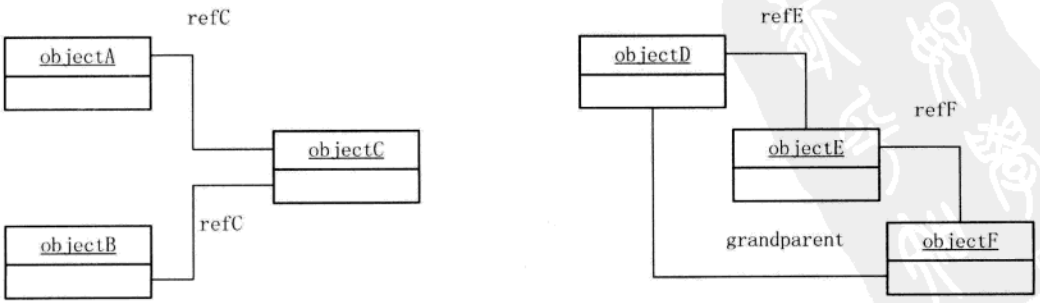



图 3-1　复杂的序列化例子

但是，序列化以后的对象在尺寸上有点过于充实了，以 Block 类为例，它只包含 3 个长整数，但是它的序列化结果竟然有 112 字节，而 BlockMetaDataInfo 其实只多了一个 long 型的成员变量，输出结果已经膨胀到 190 字节。包含 3 个长整数的 Block 对象的序列化结果如下：

```
AC ED 00 05 73 72 00 1C  6F 72 67 2E 68 61 64 6F    ....sr.. org.hado
6F 70 69 6E 74 65 72 6E  61 6C 2E 73 65 72 2E 42    opintern al.ser.B
6C 6F 63 6B E7 80 E3 D3  A6 B6 22 53 02 00 03 4A    lock.... ..."S...J
00 07 62 6C 6F 63 6B 49  64 4A 00 0F 67 65 6E 65    ..blockI dJ..gene
72 61 74 69 6F 6E 53 74  61 6D 70 4A 00 08 6E 75    rationSt ampJ..nu
6D 42 79 74 65 73 78 70  6C 55 67 95 68 E7 92 FF    mBytesxp lUg.h...
00 00 00 00 03 61 BB 8B  00 00 00 00 02 59 EC CB    .....a.. .....Y..
```

仔细看 Block 的输出会发现，序列化的结果中包含了大量与类相关的信息。Java 的序列过程在《Java Object Serialization Specification》中规范，以 Block 为例，其结果的前两个字节是魔数（Magic Number）“AC ED”；后续两个字节是序列化格式的版本号，现在使用的版本号是 5；接下来是类的描述信息，包括类的版本 ID、是否实现 writeObject() 和 readObject() 方法等信息，对于拥有超类的类（如 BlockMetaDataInfo），超类的信息也会递归地被保存下来；这些信息都写入 OutputStream 对象后，接下来才是对象的数据。在这个过程中，序列化输出中保存了大量的附加信息，导致序列化结果膨胀，对于需要保存和处理大规模数据的 Hadoop 来说，需要一个新的序列化机制。

由于篇幅的关系，不再详细讨论 Java 的序列化机制，有兴趣的读者可以参考《Java Object Serialization Specification》。

### 3.1.2 Hadoop 序列化机制

和 Java 序列化机制不同（在对象流 ObjectOutputStream 对象上调用 writeObject() 方法），Hadoop 的序列化机制通过调用对象的 write() 方法（它带有一个类型为 DataOutput 的参数），将对象序列化到流中。反序列化的过程也是类似，通过对象的 readFields()，从流中读取数据。值得一提的是，Java 序列化机制中，反序列化过程会不断地创建新的对象，但在 Hadoop 的序列化机制的反序列化过程中，用户可以复用对象：如，在 Block 的某个对象上反复调用 readFields()，可以在同一个对象上得到多个反序列化的结果，而不是多个反序列化的结果对象（对象被复用了），这减少了 Java 对象的分配和回收，提高了应用的效率。

```
public static void main(String[] args) {
   try {
      Block block1=new Block(7806259420524417791L, 39447755L, 56736651L);
      ……
      ByteArrayOutputStream bout = new ByteArrayOutputStream();
      DataOutputStream dout=new DataOutputStream(bout);
      block1.write(dout); // 序列化对象到输出流 dout 中
      dout.close();
      System.out.println(……);
      SerializationExample.print16(out.toByteArray(), bout.size());
   }
```

```
    ......
}
```

由于 Block 对象序列化时只输出了 3 个长整数，block1 的序列化结果一共有 24 字节，如下所示。和 Java 的序列化机制的输出结果对比，Hadoop 的序列化结果紧凑而且快速。

```
AC ED 00 05 73 72 00 28  6F 72 67 2E 68 61 64 6F    ....sr.( org.hado
6F 70 69 6E 74 65 72 6E                             opintern
```

### 3.1.3　Hadoop 序列化机制的特征

对于处理大规模数据的 Hadoop 平台，其序列化机制需要具有如下特征：

- ❑紧凑：由于带宽是 Hadoop 集群中最稀缺的资源，一个紧凑的序列化机制可以充分利用数据中心的带宽。
- ❑快速：在进程间通信（包括 MapReduce 过程中涉及的数据交互）时会大量使用序列化机制，因此，必须尽量减少序列化和反序列化的开销。
- ❑可扩展：随着系统的发展，系统间通信的协议会升级，类的定义会发生变化，序列化机制需要支持这些升级和变化。
- ❑互操作：可以支持不同开发语言间的通信，如 C++ 和 Java 间的通信。这样的通信，可以通过文件（需要精心设计文件的格式）或者后面介绍的 IPC 机制实现。

Java 的序列化机制虽然强大，却不符合上面这些要求。Java Serialization 将每个对象的类名写入输出流中，这导致了 Java 序列化对象需要占用比原对象更多的存储空间。同时，为了减少数据量，同一个类的对象的序列化结果只输出一份元数据，并通过某种形式的引用，来共享元数据。引用导致对序列化后的流进行处理的时候，需要保持一些状态。想象如下一种场景，在一个上百 G 的文件中，反序列化某个对象，需要访问文件中前面的某一个元数据，这将导致这个文件不能切割，并通过 MapReduce 来处理。同时，Java 序列化会不断地创建新的对象，对于 MapReduce 应用来说，不能重用对象，在已有对象上进行反序列化操作，而是不断创建反序列化的各种类型记录，这会带来大量的系统开销。

### 3.1.4　Hadoop Writable 机制

为了支持以上这些特性，Hadoop 引入 org.apache.hadoop.io.Writable 接口，作为所有可序列化对象必须实现的接口，其类图如图 3-2 所示。

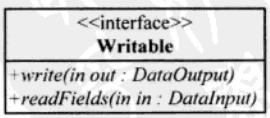



图 3-2　Writable 类图

Writable 机制紧凑、快速（但不容易扩展到 Java 以外的语言，如 C、Python 等）。和 java.io.Serializable 不同，Writable 接口不是一个说明性接口，它包含两个方法：

```
public interface Writable {
   /**
    * 输出（序列化）对象到流中
    * @param out DataOuput 流，序列化的结果保存在流中
    * @throws IOException
```

```
     */
    void write(DataOutput out) throws IOException;

    /**
     * 从流中读取（反序列化）对象
     * 为了效率，请尽可能复用现有的对象
     * @param in DataInput 流，从该流中读取数据
     * @throws IOException
     */
    void readFields(DataInput in) throws IOException;
}
```

Writable.write() 方法用于将对象状态写入二进制的 DataOutput 中，反序列化的过程由 readFields() 从 DataInput 流中读取状态完成。下面是一个例子：

```
public class Block implements Writable, Comparable<Block>, Serializable {
    ......
    private long blockId;
    private long numBytes;
    private long generationStamp;
    ......
    public void write(DataOutput out) throws IOException {
        out.writeLong(blockId);
        out.writeLong(numBytes);
        out.writeLong(generationStamp);
    }

    public void readFields(DataInput in) throws IOException {
        this.blockId = in.readLong();
        this.numBytes = in.readLong();
        this.generationStamp = in.readLong();
        if (numBytes < 0) {
            throw new IOException("Unexpected block size: " + numBytes);
        }
    }
    ......
}
```

这个例子使用的是前面分析 Java 序列化机制的 Block 类，Block 实现了 Writable 接口，即需要实现 write() 方法和 readFields() 方法，这两个方法的实现都很简单：Block 有三个成员变量，write() 方法简单地把这三个变量写入流中，而 readFields() 则从流中依次读入这些数据，并做必要的检查。

Hadoop 序列化机制中还包括另外几个重要接口：WritableComparable、RawComparator 和 WritableComparator。

WritableComparable，顾名思义，它提供类型比较的能力，这对 MapReduce 至关重要。该接口继承自 Writable 接口和 Comparable 接口，其中 Comparable 用于进行类型比较。ByteWritable、IntWritable、DoubleWritable 等 Java 基本类型对应的 Writable 类型，都继承自 WritableComparable。

效率在 Hadoop 中非常重要，因此 HadoopI/O 包中提供了具有高效比较能力的 RawComparator 接口。RawComparator 和 WritableComparable 类图如图 3-3 所示。

RawComparator 接口允许执行者比较流中读取的未被反序列化为对象的记录，从而省去了创建对象的所有开销。其中，compare() 比较时需要的两个参数所对应的记录位于字节数组 b1 和 b2 的指定开始位置 s1 和 s1，记录长度为 l1 和 l2，代码如下：

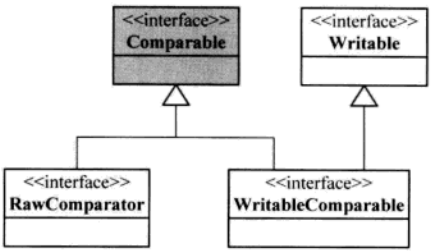



图 3-3　RawComparator 和 WritableComparable 类图

```
public interface RawComparator<T>extends
Comparator<T> {
   public int compare(byte[] b1, int s1, int
   l1, byte[] b2, int s2, int l2);
}
```

以 IntWritable 为例，它的 RawComparator 实现中（WritableComparator 是一个辅助类，实现了 RawComparator 接口），compare() 方法通过 readInt() 直接在字节数组中读入需要比较的两个整数，然后输出 Comparable 接口要求的比较结果。值得注意的是，该过程中 compare() 方法避免使用 IntWritable 对象，从而避免了不必要的对象分配。相关代码如下：

```
public static class Comparator extends WritableComparator {
   ......
   public int compare(byte[] b1, int s1, int l1,
                      byte[] b2, int s2, int l2) {
      int thisValue = readInt(b1, s1);
      int thatValue = readInt(b2, s2);
      return (thisValue<thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
   }
   ......
}
```

WritableComparator 是 RawComparator 对 WritableComparable 类的一个通用实现。它提供两个主要功能。首先，提供了一个 RawComparator 的 compare() 默认实现，该实现从数据流中反序列化要进行比较的对象，然后调用对象的 compare() 方法进行比较（这些对象都是 Comparable 的）。其次，它充当了 RawComparator 实例的一个工厂方法，例如，可以通过如下代码获得 IntWritable 的 RawComparator：

```
RawComparator<IntWritable>comparator=
   WritableComparator.get(IntWritable.class);
```

## 3.1.5　典型的 Writable 类详解

Hadoop 将很多 Writable 类归入 org.apache.hadoop.io 包中，类图如图 3-4 所示。

在这些类中，比较重要的有 Java 基本类、Text、Writable 集合、ObjectWritable 等，本节重点介绍 Java 基本类和 ObjectWritable 的实现。

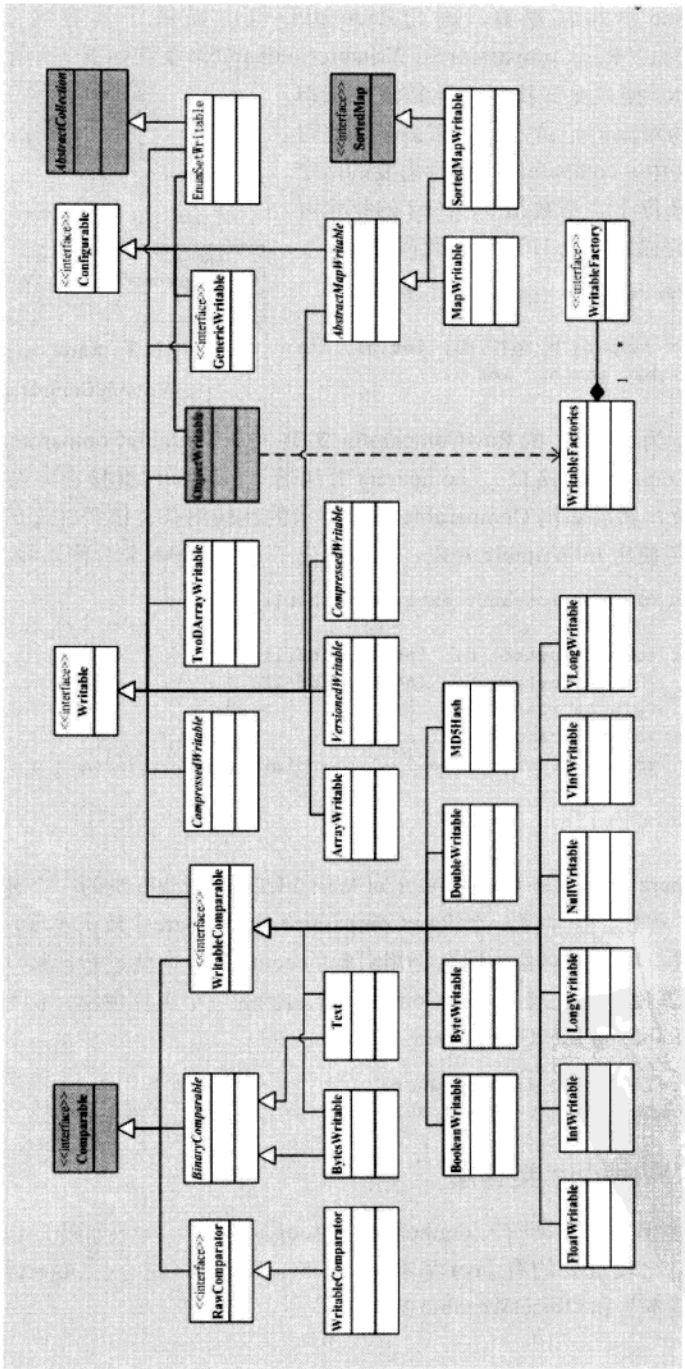



图 3-4 Writable 的子类

### 1. Java 基本类型的 Writable 封装

目前 Java 基本类型对应的 Writable 封装如表 3-1 所示。所有这些 Writable 类都继承自 WritableComparable。也就是说，它们是可比较的。同时，它们都有 get() 和 set() 方法，用于获得和设置封装的值。

表 3-1　Java 基本类型对应的 Writable 封装[⊖]

| Java 基本类型 | Writable | 序列化后长度 |
|---|---|---|
| 布尔型（boolean） | BooleanWritable | 1 |
| 字节型（byte） | ByteWritable | 1 |
| 整型（int） | IntWritable<br>VIntWritable | 4<br>1~5 |
| 浮点型（float） | FloatWritable | 4 |
| 长整型（long） | LongWritable<br>VLongWritable | 8<br>1~9 |
| 双精度浮点型（double） | DoubleWritable | 8 |

在表 3-1 中，对整型（int 和 long）进行编码的时候，有固定长度格式（IntWritable 和 LongWritable）和可变长度格式（VIntWritable 和 VLongWritable）两种选择。固定长度格式的整型，序列化后的数据是定长的，而可变长度格式则使用一种比较灵活的编码方式，对于数值比较小的整型，它们往往比较节省空间。同时，由于 VIntWritable 和 VLongWritable 的编码规则是一样的，所以 VIntWritable 的输出可以用 VLongWritable 读入。下面以 VIntWritable 为例，说明 Writable 的 Java 基本类封装实现。代码如下：

```
public class VIntWritable implements WritableComparable {
   private int value;
   ......
   // 设置 VIntWritable 的值
   public void set(int value) { this.value = value; }

   // 获取 VIntWritable 的值
   public int get() { return value; }

   public void readFields(DataInput in) throws IOException {
      value = WritableUtils.readVInt(in);
   }

   public void write(DataOutput out) throws IOException {
      WritableUtils.writeVInt(out, value);
   }
   ......
}
```

首先，每个 Java 基本类型的 Writable 封装，其类的内部都包含一个对应基本类型的

⊖ Java 的基本类型 short 和 char 并没有对应的 Writable 类，它们可以保持在 IntWritable 中。

成员变量 value，get() 和 set() 方法就是用来对该变量进行取值 / 赋值操作的。而 Writable 接口要求的 readFields() 和 write() 方法，VIntWritable 则是通过调用 Writable 工具类中提供的 readVInt() 和 writeVInt() 读 / 写数据。方法 readVInt() 和 writeVInt() 的实现也只是简单调用了 readVLong() 和 writeVLong()，所以，通过 writeVInt() 写的数据自然可以通过 readVLong() 读入。

writeVLong () 方法实现了对整型数值的变长编码，它的编码规则如下：

如果输入的整数大于或等于 -112 同时小于或等于 127，那么编码需要 1 字节；否则，序列化结果的第一个字节，保存了输入整数的符号和后续编码的字节数。符号和后续字节数依据下面的编码规则（又一个规则）：

- 如果是正数，则编码值范围落在 -113 和 -120 间（闭区间），后续字节数可以通过 -(v+112) 计算。
- 如果是负数，则编码值范围落在 -121 和 -128 间（闭区间），后续字节数可以通过 -(v+120) 计算。

后续编码将高位在前，写入输入的整数（除去前面全 0 字节）。代码如下：

```
public final class WritableUtils  {
   public stati cvoid writeVInt(DataOutput stream, int i) throws IOException
   {
      writeVLong(stream, i);
   }

   /**
    * @param stream 保存系列化结果输出流
    * @param i 被序列化的整数
    * @throws java.io.IOException
    */
   public static void writeVLong(DataOutput stream, long i) throws……
   {
      // 处于 [-112, 127] 的整数
      if (i >= -112 && i <= 127) {
         stream.writeByte((byte)i);
         return;
      }

      // 计算情况 2 的第一个字节
      int len = -112;
      if (i < 0) {
         i ^= -1L;
         len = -120;
      }

      long tmp = i;
      while (tmp != 0) {
         tmp = tmp >> 8;
         len--;
```

```
        }

        stream.writeByte((byte)len);

        len = (len < -120) ? -(len + 120) : -(len + 112);

        //输出后续字节
        for (int idx = len; idx != 0; idx--) {
           int shiftbits = (idx - 1) * 8;
           long mask = 0xFFL << shiftbits;
           stream.writeByte((byte)((i & mask) >> shiftbits));
        }
     }
  }
```

**注意** 本书附带代码对 writeVLong() 的输出结果做了一些分析，有兴趣的读者可以运行 org.hadoopinternal.ser.VLongShow，分析一些典型整数的 writeVLong() 输出结果。

### 2. ObjectWritable 类的实现

针对 Java 基本类型、字符串、枚举、Writable、空值、Writable 的其他子类，ObjectWritable 提供了一个封装，适用于字段需要使用多种类型。ObjectWritable 可应用于 Hadoop 远程过程调用（将在第 4 章介绍）中参数的序列化和反序列化；ObjectWritable 的另一个典型应用是在需要序列化不同类型的对象到某一个字段，如在一个 SequenceFile 的值中保存不同类型的对象（如 LongWritable 值或 Text 值）时，可以将该值声明为 ObjectWritable。

ObjectWritable 的实现比较冗长，需要根据可能被封装在 ObjectWritable 中的各种对象进行不同的处理。ObjectWritable 有三个成员变量，包括被封装的对象实例 instance、该对象运行时类的 Class 对象和 Configuration 对象。

ObjectWritable 的 write 方法调用的是静态方法 ObjectWritable.writeObject()，该方法可以往 DataOutput 接口中写入各种 Java 对象。

writeObject() 方法先输出对象的类名（通过对象对应的 Class 对象的 getName() 方法获得），然后根据传入对象的类型，分情况系列化对象到输出流中，也就是说，对象通过该方法输出对象的类名，对象序列化结果对到输出流中。在 ObjectWritable.writeObject() 的逻辑中，需要分别处理 null、Java 数组、字符串 String、Java 基本类型、枚举和 Writable 的子类 6 种情况，由于类的继承，处理 Writable 时，序列化的结果包含对象类名，对象实际类名和对象序列化结果三部分。

为什么需要对象实际类名呢？根据 Java 的单根继承规则，ObjectWritable 中传入的 declaredClass，可以是传入 instance 对象对应的类的类对象，也可以是 instance 对象的父类的类对象。但是，在序列化和反序列化的时候，往往不能使用父类的序列化方法（如 write 方法）来序列化子类对象，所以，在序列化结果中必须记住对象实际类名。相关代码如下：

```
public class ObjectWritable implements Writable, Configurable {
```

```
private Class declaredClass;// 保存于 ObjectWritable 的对象对应的类对象
private Object instance;// 被保留的对象
private Configuration conf;

public ObjectWritable() {}

public ObjectWritable(Object instance) {
   set(instance);
}

public ObjectWritable(Class declaredClass, Object instance) {
   this.declaredClass = declaredClass;
   this.instance = instance;
}
……
public void readFields(DataInput in) throws IOException {
   readObject(in, this, this.conf);
}

public void write(DataOutput out) throws IOException {
   writeObject(out, instance, declaredClass, conf);
}
……
public static void writeObject(DataOutput out, Object instance,
      Class declaredClass,Configuration conf) throws……{

   if (instance == null) {// 空
      instance = new NullInstance(declaredClass, conf);
      declaredClass = Writable.class;
   }

   // 写出 declaredClass 的规范名
   UTF8. -iteString(out, declaredClass.getName());

   if (declaredClass.isArray()) {// 数组
      ……
   } else if (declaredClass == String.class) {// 字符串
      ……
   } else if (declaredClass.isPrimitive()) {// 基本类型
      if (declaredClass == Boolean.TYPE) {        //boolean
         out.writeBoolean(((Boolean)instance).booleanValue());
      } else if (declaredClass == Character.TYPE) { //char
         ……
      }
   } else if (declaredClass.isEnum()) {// 枚举类型
      ……
   } else if (Writable.class.isAssignableFrom(declaredClass)) {
      //Writable 的子类
      UTF8.writeString(out, instance.getClass().getName());
      ((Writable)instance).write(out);
```

```
        } else {
            ......
    }

    public static Object readObject(DataInput in,
                    ObjectWritable objectWritable, Configuration conf){
        ......
        Class instanceClass = null;
        ......
        Writable writable = WritableFactories.newInstance(instanceClass,
              conf);
        writable.readFields(in);
        instance = writable;
        ......
    }
}
```

和输出对应，ObjectWritable 的 readFields() 方法调用的是静态方法 ObjectWritable.readObject ()，该方法的实现和 writeObject() 类似，唯一值得研究的是 Writable 对象处理部分，readObject () 方法依赖于 WritableFactories 类。WritableFactories 类允许非公有的 Writable 子类定义一个对象工厂，由该工厂创建 Writable 对象，如在上面的 readObject () 代码中，通过 WritableFactories 的静态方法 newInstance()，可以创建类型为 instanceClass 的 Writable 子对象。相关代码如下：

```
public class WritableFactories {
    // 保存了类型和 WritableFactory 工厂的对应关系
    private static final HashMap<Class, WritableFactory>CLASS_TO_FACTORY
        = new HashMap<Class, WritableFactory>();
    ......
    public static Writable newInstance(Class<? extends Writable> c,
                                       Configuration conf) {
        WritableFactory factory = WritableFactories.getFactory(c);
        if (factory != null) {
            Writable result = factory.newInstance();
            if (result instanceof Configurable) {
                ((Configurable) result).setConf(conf);
            }
            return result;
        } else {
            // 采用传统的反射工具 ReflectionUtils，创建对象
            return ReflectionUtils.newInstance(c, conf);
        }
    }
}
```

WritableFactories.newInstance() 方法根据输入的类型查找对应的 WritableFactory 工厂对象，然后调用该对象的 newInstance() 创建对象，如果该对象是可配置的，newInstance() 还会通过对象的 setConf() 方法配置对象。

WritableFactories 提供注册机制，使得这些 Writable 子类可以将该工厂登记到 WritableFactories 的静态成员变量 CLASS_TO_FACTORY 中。下面是一个典型的 WritableFactory 工厂实现，来自于 HDFS 的数据块 Block。其中，WritableFactories.setFactory() 需要两个参数，分别是注册类对应的类对象和能够构造注册类的 WritableFactory 接口的实现，在下面的代码里，WritableFactory 的实现是一个匿名类，其 newInstance() 方法会创建一个新的 Block 对象。

```
public class Block implements Writable, Comparable<Block> {
   static {
      WritableFactories.setFactory
         (Block.class,// 类对象
         new WritableFactory() {// 对应类的 WritableFactory 实现
            public Writable newInstance() { return new Block(); }
         });
   }
   ......
}
```

ObjectWritable 作为一种通用机制，相当浪费资源，它需要为每一个输出写入封装类型的名字。如果类型的数量不是很多，而且可以事先知道，则可以使用一个静态类型数组来提高效率，并使用数组索引作为类型的序列化引用。GenericWritable 就是因为这个目的被引入 org.apache.hadoop.io 包中，由于篇幅关系，不再详细介绍，有兴趣的读者可以继续分析 GenericWritable 的源代码。

### 3.1.6 Hadoop 序列化框架

大部分的 MapReduce 程序都使用 Writable 键－值对作为输入和输出，但这并不是 Hadoop 的 API 指定的，其他序列化机制也能和 Hadoop 配合，并应用于 MapReduce 中。

目前，除了前面介绍过的 Java 序列化机制和 Hadoop 使用的 Writable 机制，还流行其他序列化框架，如 Hadoop Avro、Apache Thrift 和 Google Protocol Buffer。[⊖]

- ❑Avro 是一个数据序列化系统，用于支持大批量数据交换的应用。它的主要特点有：支持二进制序列化方式，可以便捷、快速地处理大量数据；动态语言友好，Avro 提供的机制使动态语言可以方便地处理 Avro 数据。
- ❑Thrift 是一个可伸缩的、跨语言的服务开发框架，由 Facebook 贡献给开源社区，是 Facebook 的核心框架之一。基于 Thrift 的跨平台能力封装的 Hadoop 文件系统 Thrift API（参考 contrib 的 thriftfs 模块），提供了不同开发语言开发的系统访问 HDFS 的能力。
- ❑Google Protocol Buffer 是 Google 内部的混合语言数据标准，提供了一种轻便高效的结构化数据存储格式。目前，Protocol Buffers 提供了 C++、Java、Python 三种语言的 API，广泛应用于 Google 内部的通信协议、数据存储等领域中。

Hadoop 提供了一个简单的序列化框架 API，用于集成各种序列化实现，该框架由 Serialization 实现（在 org.apache.hadoop.io.serializer 包中）。

⊖ 项目地址分别是 http://avro.apache.org/、http://thrift.apache.org、http://code.google.com/apis/protocolbuffers/。

Serialization 是一个接口，使用抽象工厂的设计模式，提供了一系列和序列化相关并相互依赖对象的接口。通过 Serialization 应用可以获得类型的 Serializer 实例，即将一个对象转换为一个字节流的实现实例；Deserializer 实例和 Serializer 实例相反，它用于将字节流转为一个对象。很明显，Serializer 和 Deserializer 相互依赖，所以必须通过抽象工厂 Serialization，才能获得对应的实现。相关代码如下：

```
public interface Serialization<T> {

   // 客户端用于判断序列化实现是否支持该类对象
   boolean accept(Class<?> c);

   // 获得用于序列化对象的 Serializer 实现
   Serializer<T> getSerializer(Class<T> c);

   // 获得用于反序列化对象的 Deserializer 实现
   Deserializer<T> getDeserializer(Class<T> c);
}
```

如果需要使用 Serializer 来执行序列化，一般需要通过 open() 方法打开 Serializer，open() 方法传入一个底层的流对象，然后就可以使用 serialize() 方法序列化对象到底层的流中。最后序列化结束时，通过 close() 方法关闭 Serializer。Serializer 接口的相关代码如下：

```
public interface Serializer<T> {
   // 为输出（序列化）对象做准备
   void open(OutputStream out) throws IOException;

   // 将对象序列化到底层的流中
   void serialize(T t) throws IOException;

   // 序列化结束，清理
   void close() throws IOException;
}
```

Hadoop 目前支持两个 Serialization 实现，分别是支持 Writable 机制的 WritableSerialization 和支持 Java 序列化的 JavaSerialization。通过 JavaSerialization 可以在 MapReduce 程序中方便地使用标准的 Java 类型，如 int 或 String，但如同前面所分析的，Java 的 Object Serialization 不如 Hadoop 的序列化机制有效，非特殊情况不要轻易尝试。

## 3.2　压缩

一般来说，计算机处理的数据都存在一些冗余度，同时数据中间，尤其是相邻数据间存在着相关性，所以可以通过一些有别于原始编码的特殊编码方式来保存数据，使数据占用的存储空间比较小，这个过程一般叫压缩。和压缩对应的概念是解压缩，就是将被压缩的数据从特殊编码方式还原为原始数据的过程。

压缩广泛应用于海量数据处理中，对数据文件进行压缩，可以有效减少存储文件所需的

空间，并加快数据在网络上或者到磁盘上的传输速度。在 Hadoop 中，压缩应用于文件存储、Map 阶段到 Reduce 阶段的数据交换（需要打开相关的选项）等情景。

数据压缩的方式非常多，不同特点的数据有不同的数据压缩方式：如对声音和图像等特殊数据的压缩，就可以采用有损的压缩方法，允许压缩过程中损失一定的信息，换取比较大的压缩比；而对音乐数据的压缩，由于数据有自己比较特殊的编码方式，因此也可以采用一些针对这些特殊编码的专用数据压缩算法。

### 3.2.1 Hadoop 压缩简介

Hadoop 作为一个较通用的海量数据处理平台，在使用压缩方式方面，主要考虑压缩速度和压缩文件的可分割性。

所有的压缩算法都会考虑时间和空间的权衡，更快的压缩和解压缩速度通常会耗费更多的空间（压缩比较低）。例如，通过 gzip 命令压缩数据时，用户可以设置不同的选项来选择速度优先或空间优先，选项 -1 表示优先考虑速度，选项 -9 表示空间最优，可以获得最大的压缩比。需要注意的是，有些压缩算法的压缩和解压缩速度会有比较大的差别：gzip 和 zip 是通用的压缩工具，在时间 / 空间处理上相对平衡，gzip2 压缩比 gzip 和 zip 更有效，但速度较慢，而且 bzip2 的解压缩速度快于它的压缩速度。

当使用 MapReduce 处理压缩文件时，需要考虑压缩文件的可分割性。考虑我们需要对保持在 HDFS 上的一个大小为 1GB 的文本文件进行处理，当前 HDFS 的数据块大小为 64MB 的情况下，该文件被存储为 16 块，对应的 MapReduce 作业将会将该文件分为 16 个输入分片，提供给 16 个独立的 Map 任务进行处理。但如果该文件是一个 gzip 格式的压缩文件（大小不变），这时，MapReduce 作业不能够将该文件分为 16 个分片，因为不可能从 gzip 数据流中的某个点开始，进行数据解压。但是，如果该文件是一个 bzip2 格式的压缩文件，那么，MapReduce 作业可以通过 bzip2 格式压缩文件中的块，将输入划分为若干输入分片，并从块开始处开始解压缩数据。bzip2 格式压缩文件中，块与块间提供了一个 48 位的同步标记，因此，bzip2 支持数据分割。

表 3-2 列出了一些可以用于 Hadoop 的常见压缩格式以及特性。

**表 3-2 Hadoop 支持的压缩格式**

| 压缩格式 | UNIX 工具 | 算法 | 文件扩展名 | 支持多文件 | 可分割 |
|---|---|---|---|---|---|
| DEFLATE | 无 | DEFLATE | .deflate | 否 | 否 |
| gzip | gzip | DEFLATE | .gz | 否 | 否 |
| zip | zip | DEFLATE | .zip | 是 | 是 |
| bzip | bzip2 | bzip2 | .bz2 | 否 | 是 |
| LZO | lzop | LZO | .lzo | 否 | 否 |

为了支持多种压缩解压缩算法，Hadoop 引入了编码 / 解码器。与 Hadoop 序列化框架类似，编码 / 解码器也是使用抽象工厂的设计模式。目前，Hadoop 支持的编码 / 解码器如表 3-3

所示。

表 3-3　压缩算法及其编码 / 解码器

| 压缩格式 | 对应的编码 / 解码器 |
| --- | --- |
| DEFLATE | org.apache.hadoop.io.compress.DefaultCodec |
| gzip | org.apache.hadoop.io.compress.GzipCodec |
| bzip | org.apache.hadoop.io.compress.BZip2Codec |
| Snappy | org.apache.hadoop.io.compress.SnappyCodec ⊖ |

同一个压缩方法对应的压缩、解压缩相关工具，都可以通过相应的编码 / 解码器获得。

### 3.2.2　Hadoop 压缩 API 应用实例

本节介绍使用编码 / 解码器的典型实例（代码在 org.hadoopinternal.compress 包中）。其中，compress() 方法接受一个字符串参数，用于指定编码 / 解码器，并用对应的压缩算法对文本文件 README.txt 进行压缩。字符串参数使用 Java 的反射机制创建对应的编码 / 解码器对象，通过 CompressionCodec 对象，进一步使用它的 createOutputStream() 方法构造一个 CompressionOutputStream 流，未压缩的数据通过 IOUtils.copyBytes() 方法，从输入文件流中复制写入 CompressionOutputStream 流，最终以压缩格式写入底层的输出流中。

在本实例中，底层使用的是文件输出流 FileOutputStream，它关联文件的文件名，是在原有文件名的基础上添加压缩算法相应的扩展名生成。该扩展名可以通过 CompressionCodec 对象的 getDefaultExtension() 方法获得。相关代码如下：

```
public static void compress(String method) throws…… {
   File fileIn = new File("README.txt");

   //输入流
   InputStream in =  new FileInputStream(fileIn);

   Class<?> codecClass = Class.forName(method);

   Configuration conf = new Configuration();
   //通过名称找对应的编码 / 解码器
   CompressionCodec codec = (CompressionCodec)
         ReflectionUtils.newInstance(codecClass, conf);

   File fileOut = new File("README.txt"+codec.getDefaultExtension());
   fileOut.delete();
   //文件输出流
   OutputStream out =  new FileOutputStream(fileOut);

   //通过编码 / 解码器创建对应的输出流
   CompressionOutputStream cout =
```

⊖ Snappy 是 Google 在 Zippy 上提供的一个压缩库，后面会介绍如何引入一个压缩库到 Hadoop 中。

```
        codec.createOutputStream(out);

    // 压缩
    IOUtils.copyBytes(in, cout, 4096, false);

    in.close();
    cout.close();
}
```

需要解压缩文件时，通常通过其扩展名来推断它对应的编码 / 解码器，进而用相应的解码流对数据进行解码，如扩展名为 gz 的文件可以使用 GzipCodec 阅读。每个压缩格式的扩展名请参考表 3-3。

CompressionCodecFactory 提供了 getCodec() 方法，用于将文件扩展名映射到对应的编码 / 解码器，如下面的例子。有了 CompressionCodec 对象，就可以使用和压缩类似的过程，通过对象的 createInputStream() 方法获得 CompressionInputStream 对象，解码数据。相关代码如下：

```
public static void decompress(File file) throws IOException {
    Configuration conf = new Configuration();
    CompressionCodecFactory factory = new CompressionCodecFactory(conf);

    // 通过文件扩展名获得相应的编码 / 解码器
    CompressionCodec codec = factory.getCodec(new Path(file.getName()));

    if( codec == null ) {
        System.out.println("Cannot find codec for file "+file);
        return;
    }

    File fileOut = new File(file.getName()+".txt");

    // 通过编码 / 解码器创建对应的输入流
    InputStream in = codec.createInputStream( new FileInputStream(file) );
    ......
}
```

### 3.2.3 Hadoop 压缩框架

Hadoop 通过以编码 / 解码器为基础的抽象工厂方法，提供了一个可扩展的框架，支持多种压缩方法。下面就来研究 Hadoop 压缩框架的实现。

#### 1. 编码 / 解码器

前面已经提过，CompressionCodec 接口实现了编码 / 解码器，使用的是抽象工厂的设计模式。CompressionCodec 提供了一系列方法，用于创建特定压缩算法的相关设施，其类图如图 3-5 所示。

CompressionCodec 中的方法很对称，一个压缩功能总对应着一个解压缩功能。其中，与

压缩有关的方法包括：

- createOutputStream() 用于通过底层输出流创建对应压缩算法的压缩流，重载的 createOutputStream() 方法可使用压缩器创建压缩流；
- createCompressor() 方法用于创建压缩算法对应的压缩器。后续会继续介绍压缩流 CompressionOutputStream 和压缩器 Compressor。解压缩也有对应的方法和类。

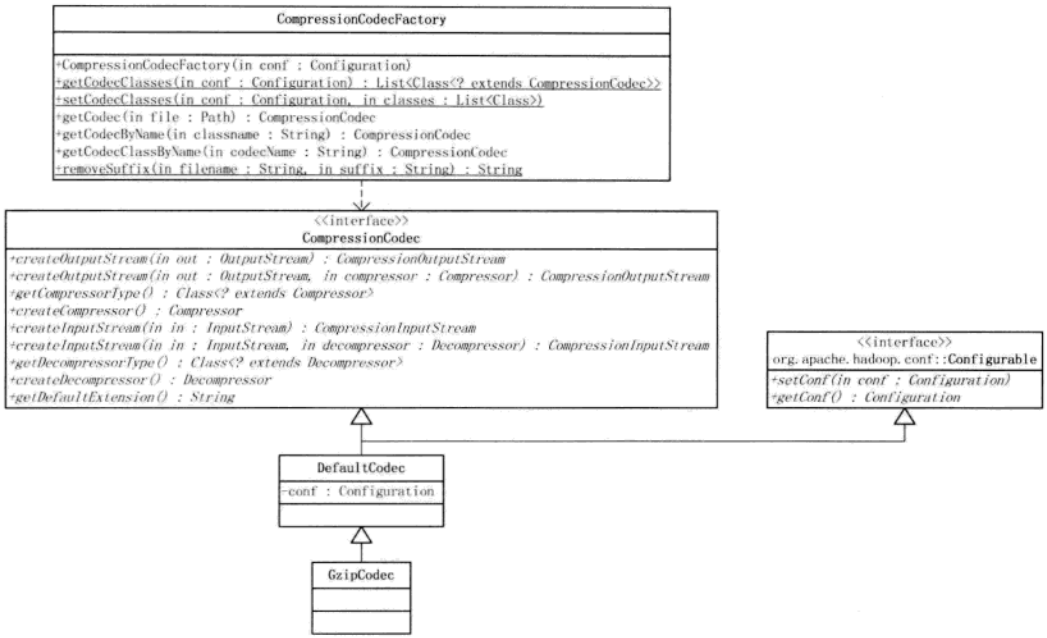



图 3-5　CompressionCodec 及其相关类

CompressionCodec 中还提供了获取对应文件扩展名的方法 getDefaultExtension()，如对于 org.apache.hadoop.io.compress.BZip2Codec，该方法返回字符串“.bz2”，注意字符串的第一个字符。相关代码如下：

```
public interface CompressionCodec {

    // 在底层输出流 out 的基础上创建对应压缩算法的压缩流 CompressionOutputStream 对象
    CompressionOutputStream createOutputStream(OutputStream out)……

    // 使用压缩器 compressor，在底层输出流 out 的基础上创建对应的压缩流
    CompressionOutputStream createOutputStream(OutputStream out,
                                               Compressor compressor) ……
    ……
    // 创建压缩算法对应的压缩器
    Compressor createCompressor();

    // 在底层输入流 in 的基础上创建对应压缩算法的解压缩流 CompressionInputStream 对象
```

```
    CompressionInputStream createInputStream(InputStream in) ……
    ……

    // 获得压缩算法对应的文件扩展名
    String getDefaultExtension();
}
```

CompressionCodecFactory 是 Hadoop 压缩框架中的另一个类，它应用了工厂方法，使用者可以通过它提供的方法获得 CompressionCodec。

---

**注意** 抽象工厂方法和工厂方法这两个设计模式有很大的区别，抽象工厂方法用于创建一系列相关或互相依赖的对象，如 CompressionCodec 可以获得和某一个压缩算法相关的对象，包括压缩流和解压缩流等。而工厂方法（严格来说，CompressionCodecFactory 是参数化工厂方法），用于创建多种产品，如通过 CompressionCodecFactory 的 getCodec() 方法，可以创建 GzipCodec 对象或 BZip2Codec 对象。

---

在前面的实例中已经使用过 getCodec() 方法，为某一个压缩文件寻找对应的 CompressionCodec。为了分析该方法，需要了解 CompressionCodec 类中保存文件扩展名和 CompressionCodec 映射关系的成员变量 codecs。

codecs 是一个有序映射表，即它本身是一个 Map，同时它对 Map 的键排序，下面是 codecs 中保存的一个可能的映射关系：

```
{
    2zb.: org.apache.hadoop.io.compress.BZip2Codec,
    etalfed.: org.apache.hadoop.io.compress.DeflateCodec,
    yppans.: org.apache.hadoop.io.compress.SnappyCodec,
    zg.: org.apache.hadoop.io.compress.GzipCodec
}
```

可以看到，Map 中的键是排序的。

getCodec() 方法的输入是 Path 对象，保存着文件路径，如实例中的“README.txt.bz2”。

首先通过获取 Path 对象对应的文件名并逆转该字符串得到“2zb.txt.EMDAER”，然后通过有序映射 SortedMap 的 headMap() 方法，查找最接近上述逆转字符串的有序映射的部分视图，如输入“2zb.txt.EMDAER”的查找结果 subMap，只包含“2zb.”对应的那个键 – 值对，如果输入是“zg.txt.EMDAER”，则 subMap 会包含成员变量 codecs 中保存的所有键 – 值对。

然后，简单地获取 subMap 最后一个元素的键，如果该键是逆转文件名的前缀，那么就找到了文件对应的编码 / 解码器，否则返回空。实现代码如下：

```
public class CompressionCodecFactory {
    ……
    // 该有序映射保存了逆转文件后缀（包括后缀前的“.”）到 CompressionCodec 的映射
    // 通过逆转文件后缀，我们可以找到最长匹配后缀
    private SortedMap<String, CompressionCodec> codecs = null;
    ……
    public CompressionCodec getCodec(Path file) {
```

```
            CompressionCodec result = null;
        if (codecs != null) {
            String filename = file.getName();
            //逆转字符串
            String reversedFilename = new
                StringBuffer(filename).reverse().toString();
            SortedMap<String, CompressionCodec> subMap =
                codecs.headMap(reversedFilename);
            if (!subMap.isEmpty()) {
                String potentialSuffix = subMap.lastKey();
                if (reversedFilename.startsWith(potentialSuffix)) {
                    result = codecs.get(potentialSuffix);
                }
            }
        }
        return result;
    }
}
```

CompressionCodecFactory.getCodec() 方法的代码看似复杂，但通过灵活使用有序映射 SortedMap，实现其实还是非常简单的。

### 2. 压缩器和解压器

压缩器（Compressor）和解压器（Decompressor）是 Hadoop 压缩框架中的一对重要概念。

Compressor 可以插入压缩输出流的实现中，提供具体的压缩功能；相反，Decompressor 提供具体的解压功能并插入 CompressionInputStream 中。Compressor 和 Decompressor 的这种设计，最初是在 Java 的 zlib 压缩程序库中引入的，对应的实现分别是 java.util.zip.Deflater 和 java.util.zip.Inflater。下面以 Compressor 为例介绍这对组件。

Compressor 的用法相对复杂，请参考 org.hadoopinternal.compress.CompressDemo 的 compressor() 方法。Compressor 通过 setInput() 方法接收数据到内部缓冲区，自然可以多次调用 setInput() 方法，但内部缓冲区总是会被写满。如何判断压缩器内部缓冲区是否已满呢？可以通过 needsInput() 的返回值，如果是 false，表明缓冲区已经满，这时必须通过 compress() 方法获取压缩后的数据，释放缓冲区空间。

为了提高压缩效率，并不是每次用户调用 setInput() 方法，压缩器就会立即工作，所以，为了通知压缩器所有数据已经写入，必须使用 finish() 方法。finish() 调用结束后，压缩器缓冲区中保持的已经压缩的数据，可以继续通过 compress() 方法获得。至于要判断压缩器中是否还有未读取的压缩数据，则需要利用 finished() 方法来判断。

---

**注意**　finished() 和 finish() 的作用不同，finish() 结束数据输入的过程，而 finished() 返回 false，表明压缩器中还有未读取的压缩数据，可以继续通过 compress() 方法读取。

---

使用 Compressor 的一个典型实例如下：

```
public static void compressor() throws ClassNotFoundException, IOException
```

```
{
   // 读入被压缩的内容
   File fileIn = new File("README.txt");
   InputStream in =  new FileInputStream(fileIn);
   int datalength=in.available();
   byte[] inbuf = new byte[datalength];
   in.read(inbuf, 0, datalength);
   in.close();

   // 长度受限制的输出缓冲区，用于说明 finished() 方法
   byte[] outbuf = new byte[compressorOutputBufferSize];

   Compressor compressor=new BuiltInZlibDeflater();// 构造压缩器

   int step=100;// 一些计数器
   int inputPos=0;
   int putcount=0;
   int getcount=0;
   int compressedlen=0;

   while(inputPos < datalength) {
      // 进行多次 setInput()
      int len=(datalength-inputPos>=step)? step:datalength-inputPos;
      compressor.setInput(inbuf, inputPos, len );
      putcount++;

      while (!compressor.needsInput()) {
         compressedlen=compressor.compress(outbuf, 0, ……);
         if(compressedlen>0) {
            getcount++;  // 能读到数据
         }
      }  // end of while (!compressor.needsInput())
      inputPos+=step;
   }

   compressor.finish();

   while(!compressor.finished()) {  // 压缩器中有数据
      getcount++;
      compressor.compress(outbuf, 0, compressorOutputBufferSize);
   }

   System.out.println("Compress "+compressor.getBytesRead()  // 输出信息
                      +" bytes into "+compressor.getBytesWritten());
   System.out.println("put "+putcount+" times and get "+getcount+" times");

   compressor.end();// 停止
}
```

以上代码实现了 setInput()、needsInput()、finish()、compress() 和 finished() 的配合过程。

将输入 inbuf 分成几个部分，通过 setInput() 方法送入压缩器，而在 finish() 调用结束后，通过 finished() 循序判断压缩器是否还有未读取的数据，并使用 compress() 方法获取数据。

在压缩的过程中，Compressor 可以通过 getBytesRead() 和 getBytesWritten() 方法获得 Compressor 输入未压缩字节的总数和输出压缩字节的总数，如实例中最后一行的输出语句。Compressor 和 Decompressor 的类图如图 3-6 所示。

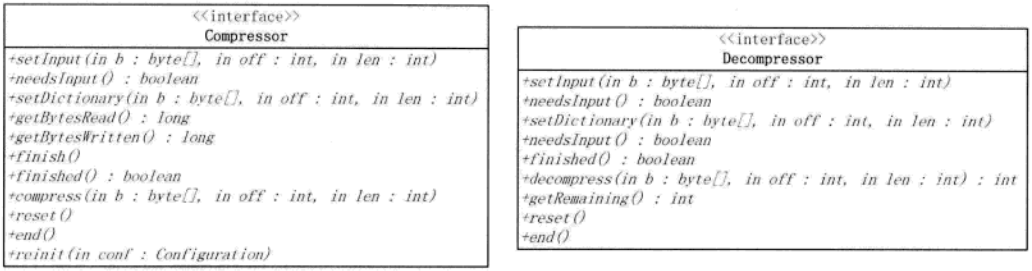



图 3-6　Compressor 和 Decompressor 的类图

Compressor.end() 方法用于关闭解压缩器并放弃所有未处理的输入；reset() 方法用于重置压缩器，以处理新的输入数据集合；reinit() 方法更进一步允许使用 Hadoop 的配置系统，重置并重新配置压缩器。

限于篇幅，这里就不再探讨解压器 Decompressor 了。

### 3. 压缩流和解压缩流

Java 最初版本的输入 / 输出系统是基于流的，流抽象了任何有能力产出数据的数据源，或者是有能力接收数据的接收端。一般来说，通过设计模式装饰，可以为流添加一些额外的功能，如前面提及的序列化流 ObjectInputStream 和 ObjectOutputStream。

压缩流（CompressionOutputStream）和解压缩流（CompressionInputStream）是 Hadoop 压缩框架中的另一对重要概念，它提供了基于流的压缩解压缩能力。如图 3-7 所示是从 java.io.InputStream 和 java.io.OutputStream 开始的类图。

这里只分析和压缩相关的代码，即 CompressionOutputStream 及其子类。

OutputStream 是一个抽象类，提供了进行流输出的基本方法，它包含三个 write 成员函数，分别用于往流中写入一个字节、一个字节数组或一个字节数组的一部分（需要提供起始偏移量和长度）。

---

**注意**　流实现中一般需要支持的 close() 和 flush() 方法，是 java.io 包中的相应接口的成员函数，不是 OutputStream 的成员函数。

---

CompressionOutputStream 继承自 OutputStream，也是个抽象类。如前面提到的 ObjectOutputStream、CompressionOutputStream 为其他流添加了附加额外的压缩功能，其他流保存在类的成员变量 out 中，并在构造的时候被赋值。

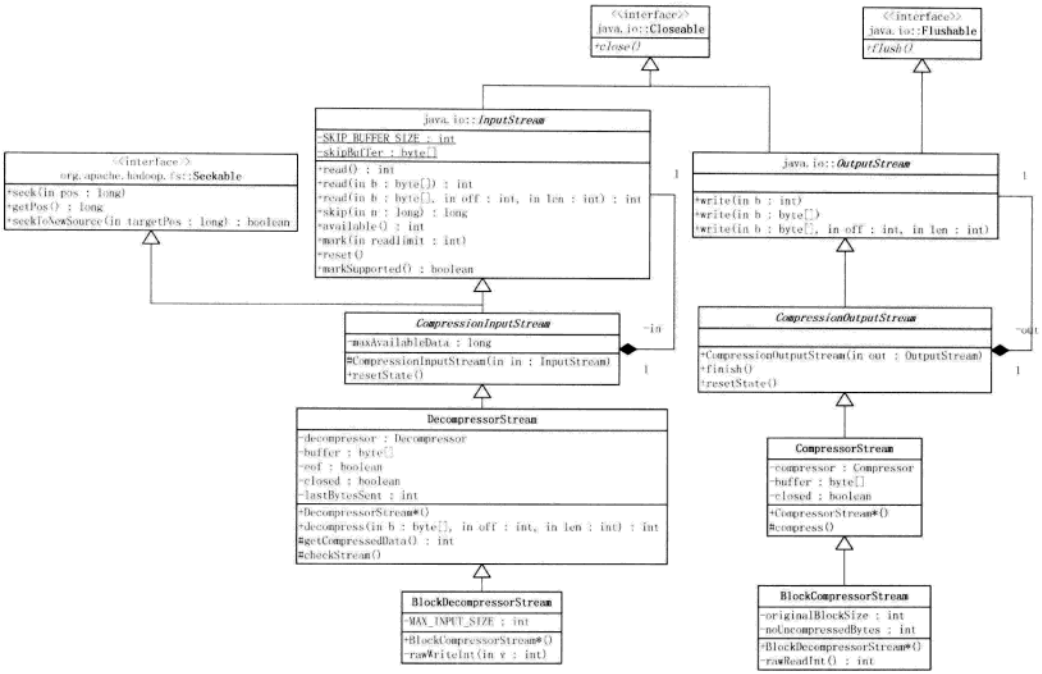



图 3-7 压缩流 / 解压缩流类图

CompressionOutputStream 实现了 OutputStream 的 close() 方法和 flush() 方法，但用于输出数据的 write() 方法、用于结束压缩过程并将输入写到底层流的 finish() 方法和重置压缩状态的 resetState() 方法还是抽象方法，需要 CompressionOutputStream 的子类实现。相关代码如下：

```
public abstract class CompressionOutputStream extends OutputStream {
    // 输出压缩结果的流
    protected final OutputStream out;

    // 构造函数
    protected CompressionOutputStream(OutputStream out) {
        this.out = out;
    }

    public void close() throws IOException {
        finish();
        out.close();
    }

    public void flush() throws IOException {
        out.flush();
    }
```

```
    public abstract void write(byte[] b, int off, int len) throws IOException;

    public abstract void finish() throws IOException;

    public abstract void resetState() throws IOException;
}
```

CompressionOutputStream 规定了压缩流的对外接口，如果已经有了一个压缩器的实现，能否提供一个通用的、使用压缩器的压缩流实现呢？答案是肯定的，CompressorStream 使用压缩器实现了一个通用的压缩流，其主要代码如下：

```
public class CompressorStream extends CompressionOutputStream {
    protected Compressor compressor;
    protected byte[] buffer;
    protected boolean closed = false;

    //构造函数
    public CompressorStream(OutputStream out,
            Compressor compressor, int bufferSize) {
        super(out);
        ……//参数检查，略
        this.compressor = compressor;
        buffer = new byte[bufferSize];
    }
    ……
    public void write(byte[] b, int off, int len) throws IOException {
        //参数检查，略
        ……
        compressor.setInput(b, off, len);
        while (!compressor.needsInput()) {
            compress();
        }
    }

    protected void compress() throws IOException {
        int len = compressor.compress(buffer, 0, buffer.length);
        if (len > 0) {
            out.write(buffer, 0, len);
        }
    }

    //结束输入
    public void finish() throws IOException {
        if (!compressor.finished()) {
            compressor.finish();
            while (!compressor.finished()) {
                compress();
            }
        }
    }
```

```
    ......
    // 关闭流
    public void close() throws IOException {
        if (!closed) {
            finish();// 结束压缩
            out.close();// 关闭底层流
            closed = true;
        }
    }
    ......
}
```

CompressorStream 提供了几个不同的构造函数，用于初始化相关的成员变量。上述代码片段中保留了参数最多的构造函数，其中，CompressorStream 需要的底层输出流 out 和压缩时使用的压缩器，都作为参数传入构造函数。另一个参数是 CompressorStream 工作时使用的缓冲区 buffer 的大小，构造时会利用这个参数分配该缓冲区。

CompressorStream.write() 方法用于将待压缩的数据写入流中。待压缩的数据在进行一番检查后，最终调用压缩器的 setInput() 方法进入压缩器。setInput() 方法调用结束后，通过 Compressor.needsInput() 判断是否需要调用 compress() 方法，获取压缩后的输出数据。上一节已经讨论了这个问题，如果内部缓冲区已满，则需要通过 compress() 方法提取数据，提取后的数据直接通过底层流的 write() 方法输出。

当 finish() 被调用（往往是 CompressorStream 被关闭），这时 CompressorStream 流调用压缩器的 finish() 方法通知输入已经结束，然后进入另一个循环，该循环不断读取压缩器中未读取的数据，然后输出到底层流 out 中。

CompressorStream 中的其他方法，如 resetState() 和 close() 都比较简单，不再一一介绍了。

CompressorStream 利用压缩器 Compressor 实现了一个通用的压缩流，在 Hadoop 中引入一个新的压缩算法，如果没有特殊的考虑，一般只需要实现相关的压缩器和解压器，然后通过 CompressorStream 和 DecompressorStream，就实现相关压缩算法的输入 / 输出流了。

CompressorStream 的实现并不复杂，只需要注意压缩器几个方法间的配合，图 3-8 给出了这些方法的一个典型调用顺序，供读者参考。

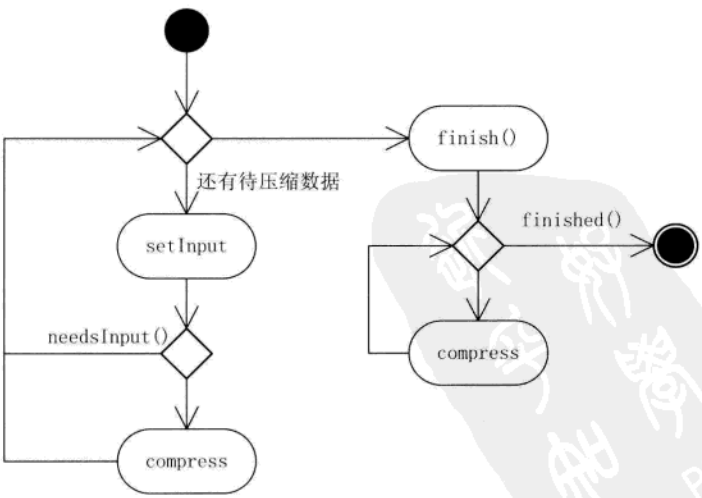



图 3-8 压缩器方法的典型调用顺序

### 3.2.4　Java 本地方法

数据压缩往往是计算密集型的操作，考虑到性能，建议使用本地库（Native Library）来压缩和解压。在某个测试中，与 Java 实现的内置 gzip 压缩相比，使用本地 gzip 压缩库可以将解压时间减少 50%，而压缩时间大概减少 10%。

Hadoop 的 DEFLATE、gzip 和 Snappy 都支持算法的本地实现，其中 Apache 发行版中还包含了 DEFLATE 和 gzip 的 32 位和 64 位 Linux 本地压缩库（Cloudera 发行版还包括 Snappy 压缩方法）。默认情况下，Hadoop 会在它运行的平台上查找本地库。

假设有一个 C 函数，它实现了某些功能，同时因为某种原因（如效率），使得用户不希望用 Java 语言重新实现该功能，那么 Java 本地方法（Native Method）就是一个不错的选择。Java 提供了一些钩子函数，使得调用本地方法成为可能，同时，JDK 也提供了一些工具，协助用户减轻编程负担。不熟悉 C 语言的读者可以略过本部分的内容。

Java 语言中的关键字 native 用于表示某个方法为本地方法，显然，本地方法是类的成员方法。下面是一个本地方法的例子，代码片段来自 Cloudera 的 Snappy 压缩实现，在 org.apache.hadoop.io.compress.snappy 包中[⊖]。其中，静态方法 initIDs() 和方法 compressBytesDirect() 用关键字 native 修饰，表明这是一个 Java 本地方法。相关代码如下：

```
public class SnappyCompressor implements Compressor {
   ......
   private native static void initIDs();

   private native int compressBytesDirect();
}
```

实际上，如果什么都不做也可以编译这个类，但是当使用这个类的时候，Java 虚拟机会告诉你无法找到上述两个方法。要想实现这两个本地方法，一般需要如下三个步骤：

1）为方法生成一个在 Java 调用和实际 C 函数间转换的 C 存根；

2）建立一个共享库并导出该存根；

3）使用 System.loadLibrary() 方法通知 Java 运行环境加载共享库。

JDK 为 C 存根的生成提供了实用程序 javah，以上面 SnappyCompressor 为例，可以在 build/classes 目录下执行如下命令：

```
javah org.apache.hadoop.io.compress.snappy.SnappyCompressor
```

系统会生成一个头文件 org_apache_hadoop_io_compress_snappy_SnappyCompressor.h。该文件包含上述两个本地方法相应的声明：

- Java_org_apache_hadoop_io_compress_snappy_SnappyCompressor_initIDs（下面以 Java_…_initIDs 代替）
- Java_org_apache_hadoop_io_compress_snappy_SnappyCompressor_compressBytesDirect（下面以 Java_…_ compressBytesDirect 代替）

⊖ 可以在 Cloudera 网站（www.cloudera.com）下载这部分实现的完整代码。

这两个声明遵从了 Java 本地方法的命名规则，以 Java 起首，然后是类的名称，最后是本地方法的方法名。声明中的 JNIEXPORT 和 JNICALL 表明了这两个方法会被 JNI 调用。

上述第一个声明对应方法 Java_…_initIDs，由于是一个静态方法，它的参数包括类型为 JNIEnv 的指针，用于和 JVM 通信。JNIEnv 提供了大量的函数，可以执行类和对象的相关方法，也可以访问对象的成员变量或类的静态变量（对于 Java_…_initIDs，只能访问类的静态变量）。参数 jclass 提供了引用静态方法对应类的机制，而 Java_…_compressBytesDirect 中的 jobject，则可以理解为 this 引用。这两个参数大量应用于 JNI 提供的 C API 中。

头文件 org_apache_hadoop_io_compress_snappy_SnappyCompressor.h 代码如下：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include<jni.h>
/* Header for class org_apache_hadoop_io_compress_snappy_SnappyCompressor */

#ifndef _Included_org_apache_hadoop_io_compress_snappy_SnappyCompressor
#define _Included_org_apache_hadoop_io_compress_snappy_SnappyCompressor
#ifdef __cplusplus
extern "C" {
#endif
#undef
  org_apache_hadoop_io_compress_snappy_SnappyCompressor_DEFAULT_DIRECT_BUFFER_SIZE
#define
  org_apache_hadoop_io_compress_snappy_SnappyCompressor_DEFAULT_DIRECT_BUFFER_
  SIZE 65536L
/*
 * Class:     org_apache_hadoop_io_compress_snappy_SnappyCompressor
 * Method:    initIDs
 * Signature: ()V
 */
  JNIEXPORT void JNICALL Java_org_apache_hadoop_io_compress_snappy_SnappyCompressor_initIDs
    (JNIEnv *, jclass);

/*
 * Class:     org_apache_hadoop_io_compress_snappy_SnappyCompressor
 * Method:    compressBytesDirect
 * Signature: ()I
 */
  JNIEXPORT jint JNICALL Java_org_apache_hadoop_io_compress_snappy_
  SnappyCompressor_compressBytesDirect
    (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

有了上述头文件，就可以实现 Java_…_initIDs 和 Java_…_ compressBytesDirect 方法，它们的实现在目录 src/native/src/org/apache/hadoop/io/compress/snappy 下，压缩部分对应的 C 源代码是 SnappyCompressor.c，在这里只介绍 Java_…_compressBytesDirect 的实现，代码如下：

```
……
static jfieldID SnappyCompressor_clazz;
……
static jfieldID SnappyCompressor_directBufferSize;
……
static snappy_status (*dlsym_snappy_compress)(constchar*, size_t, char*, size_t*);
……
JNIEXPORT jint JNICALL Java_org_apache_hadoop_io_compress_snappy_SnappyCompressor_
  compressBytesDirect(JNIEnv *env, jobject thisj)
{
   // 获得 SnappyCompressor 的相关成员变量
   jobject clazz = (*env)->GetStaticObjectField
      (env, thisj, SnappyCompressor_clazz);
   jobject uncompressed_direct_buf = (*env)->GetObjectField
      (env, thisj, SnappyCompressor_uncompressedDirectBuf);
   jint uncompressed_direct_buf_len = (*env)->GetIntField
      (env, thisj, SnappyCompressor_uncompressedDirectBufLen);
   jobject compressed_direct_buf = (*env)->GetObjectField
      (env, thisj, SnappyCompressor_compressedDirectBuf);
   jint compressed_direct_buf_len = (*env)->GetIntField
      (env, thisj, SnappyCompressor_directBufferSize);

   // 获得未压缩数据缓冲区
   LOCK_CLASS(env, clazz, "SnappyCompressor");
   const char* uncompressed_bytes = (const char*)
      (*env)->GetDirectBufferAddress(env, uncompressed_direct_buf);
   UNLOCK_CLASS(env, clazz, "SnappyCompressor");

   if (uncompressed_bytes == 0) {
      return (jint)0;
   }

   // 获得保存压缩结果的缓冲区
   ……

   // 进行数据压缩
   snappy_status ret = dlsym_snappy_compress(uncompressed_bytes,
                                    uncompressed_direct_buf_len,
                                    compressed_bytes,
                                    &compressed_direct_buf_len);
   // 处理返回结果
   if (ret != SNAPPY_OK){
      THROW(env, "Ljava/lang/InternalError",
            "Could not compress data. Buffer length is too small.");
   }

   (*env)->SetIntField
            (env, thisj, SnappyCompressor_uncompressedDirectBufLen, 0);

   return (jint)compressed_direct_buf_len;
}
```

在介绍 Java_…_ compressBytesDirect 的实现前，先来研究几个 JNI C 提供的方法。

JNIEnv 提供了 C 代码和 Java 虚拟机通信的环境，Java_…_compressBytesDirect 方法执行过程中需要获得 SnappyCompressor 类中的一些成员变量，就需要使用 JNIEnv 提供的方法。

GetObjectField() 函数可用于获得对象的一个域，在上述代码中，使用该方法获得了保存待压缩数据缓冲区和压缩数据写入的缓冲区的成员变量，即 SnappyCompressor 的成员变量 uncompressedDirectBuf 和 compressedDirectBuf，接着使用 JNIEnv 的 GetDirectBufferAddress() 方法获得缓冲区的地址，这样，就可以直接访问数据缓冲区中的数据了。

JNIEnv 还提供 GetIntField() 函数，可用于得到 Java 对象的整型成员变量，而 SetIntField() 函数则相反，它设置 Java 对象的整型成员变量的值。

了解了这些 JNI 方法以后，Java_…_compressBytesDirect 的实现就非常好理解了，它将未压缩的数据进行压缩，首先当然是获得相应的缓冲区了，通过 GetObjectField() 和 GetIntField() 获取输入缓冲区及其大小、输出缓冲区及其大小后，就可以调用 Snappy 本地库提供的压缩方法。

调用 Snappy 压缩算法是通过函数指针 dlsym_snappy_compress 进行的，该指针在 Java_…_initIDs 中初始化，对应的是 Snappy 库 libsnappy 中的 snappy_compress() 方法。相关代码如下：

```
JNIEXPORT void JNICALL
  Java_org_apache_hadoop_io_compress_snappy_SnappyCompressor_initIDs
(JNIEnv *env, jclass clazz){
   ......
   LOAD_DYNAMIC_SYMBOL(dlsym_snappy_compress, env,
                       libsnappy, "snappy_compress");
   ......
}
```

snappy_compress() 方法是 Google 提供的 Snappy 压缩库中的方法，它需要输入缓冲区及其大小，输出缓冲区及其大小 4 个参数。注意，输出缓冲区大小参数是一个指针，压缩结果的大小通过该指针指向的整数返回，即它既充当输入值，也是返回值。相关代码如下：

```
snappy_status snappy_compress(const char* input,
                              size_t input_length,
                              char* compressed,
                              size_t* compressed_length);
```

Java_…_ compressBytesDirect 中对 snappy_compress() 的调用结束后，还需要进行返回值检查，如果该值不为 0，则通过 THROW() 抛出异常，否则返回压缩后数据的长度。

实现了 SnappyCompressor.c 以后，作为本地方法开发的后续步骤，还需要使用 C 的编译器、链接器编译相关的文件并连接成动态库（Cloudera 发行版中，SnappyCompressor.c 和其他本地库打包成 libhadoop.so，当然，运行时还需要将 Snappy 的动态库加载进来）。在运行时，需要将包含该动态库的路径放入系统属性 java.library.path 中，这样，Java 虚拟机才能找到对应的动态库。最后，Java 应用需要显式通知 Java 运行环境加载相关的动态库（如加载

Snappy 的动态库），可用如下代码（细节请参考类 LoadSnappy 的实现）：

```
public class LoadSnappy {
   try {
      System.loadLibrary("snappy");
      LOG.warn("Snappy native library is available");
      AVAILABLE = true;
   } catch (UnsatisfiedLinkError ex) {
      //NOP
   }
   ......
```

System.loadLibrary() 方法会用在 java.library.path 指定的路径下，寻找并加载附加的动态库，如上述调用，可以加载 Snappy 压缩需要的 libsnappy.so 库。

### 3.2.5　支持 Snappy 压缩

Snappy ⊖的前身是 Zippy，虽然只是一个数据压缩库，却被 Google 用于许多内部项目，如 BigTable、MapReduce 等。Google 表示该算法库针对性能做了调整，针对 64 位 x86 处理器进行了优化，并在英特尔酷睿 i7 处理器单一核心上实现了至少每秒 250MB 的压缩性能和每秒 500MB 的解压缩性能。Snappy 在 Google 的生产环境中经过了 PB 级数据压缩的考验，并使用 New BSD 协议开源。

本节不介绍 Snappy 压缩算法是如何实现的，而是在前面已有的基础上，介绍如何在 Hadoop 提供的压缩框架下集成新的压缩算法。本节只介绍和压缩相关的实现，将涉及 Cloudera 发行版的 org.apache.hadoop.io.compress.snappy 包下的代码和 org.apache.hadoop.io.compress.SnappyCodec 类。

org.apache.hadoop.io.compress.snappy 包括支持 Snappy 的压缩器 SnappyCompressor 和解压器 SnappyDecompressor。LoadSnappy 类用于判断 Snappy 的本地库是否可用，如果可用，则通过 System.loadLibrary() 加载本地库（上一节分析过这部分代码）。

SnappyCompressor 实现了 Compressor 接口，是这一节的重点。前面提过，压缩器的一般用法是循环调用 setInput()、finish() 和 compress() 三个方法对数据进行压缩。在分析这些方法前，了解 SnappyCompressor 的主要成员变量，如下所示：

```
public class SnappyCompressor implements Compressor {
   ......
   private int directBufferSize;
   private Buffer compressedDirectBuf = null;  // 输出（压缩）数据缓冲区
   private int uncompressedDirectBufLen;
   private Buffer uncompressedDirectBuf = null;  // 输入数据缓冲区
   private byte[] userBuf = null;
   private int userBufOff = 0, userBufLen = 0;
   private boolean finish, finished;
```

⊖ 项目地址：http://code.google.com/p/snappy/。

```
    private long bytesRead = 0L;  // 计数器，供 getBytesRead() 使用
    private long bytesWritten = 0L;  // 计数器，供 getBytesWritten() 使用
    ......
}
```

SnappyCompressor 的主要属性有 compressedDirectBuf 和 uncompressedDirectBuf，分别用于保存压缩前后的数据，类型都是 Buffer。缓冲区 Buffer 代表一个有限容量的容器，是 Java NIO（新输入 / 输出系统）中的重要概念，和基于流的 Java IO 不同，缓冲区可以用于输入，也可以用于输出。为了支持这些特性，缓冲区会维持一些标记，记录目前缓冲区中的数据存放情况（第 4 章详细介绍 Buffer，读者可以参考 Java NIO 的内容）。

成员变量 userBuf、userBufOff 和 userBufLen 用于保存通过 setInput() 设置的，但超过压缩器工作空间 uncompressedDirectBuf 剩余可用空间的数据。后面在分析 setInput() 方法的时候，可以看到这三个变量是如何使用的。

在分析压缩器 / 解压器和压缩流 / 解压缩流时，一直强调 Compressor 的 setInput()、needsInput()、finish()、finished() 和 compress() 5 个方法间的配合，那么为什么需要这样的配合呢？让我们先从 setInput() 开始了解这些方法的实现。

### 1. setInput()

setInput() 方法为压缩器提供数据，在做了一番输入数据的合法性检查后，先将 finished 标志位置为 false，并尝试将输入数据复制到内部缓冲区中。如果内部缓存器剩余空间不够大，那么，压缩器将“借用”输入数据对应的缓冲区，即利用 userBuf、userBufOff 和 userBufLen 记录输入的数据。否则，setInput() 复制数据到 uncompressedDirectBuf 中。

需要注意的是，当“借用”发生时，使用的是引用，即数据并没有发生实际的复制，用户不能随便修改传入的数据。同时，缓冲区只能借用一次，用户如果再次调用 setInput()，将会替换原来保存的相关信息，造成数据错误。相关代码如下：

```
public synchronized void setInput(byte[] b, int off, int len) {
    ......
    finished = false;

    if (len > uncompressedDirectBuf.remaining()) {
        // 借用外部缓冲区，这个时候 needsInput 为 false
        this.userBuf = b;
        this.userBufOff = off;
        this.userBufLen = len;
    } else {
        ((ByteBuffer) uncompressedDirectBuf).put(b, off, len);
        uncompressedDirectBufLen = uncompressedDirectBuf.position();
    }

    bytesRead += len;
}
```

setInput() 借用外部缓冲区后就不能再接收数据，这时，用户调用 needsInput() 将返回

false，就可以获知这个信息。

### 2. needsInput()

needsInput() 方法返回 false 有三种情况：输出缓冲区（即保持压缩结果的缓冲区）有未读取的数据、输入缓冲区没有空间，以及压缩器已经借用外部缓冲区。这时，用户需要通过 compress() 方法取走已经压缩的数据，直到 needsInput() 返回 true，才可再次通过 setInput() 方法添加待压缩数据。相关代码如下：

```
public synchronized boolean needsInput() {
    return !(compressedDirectBuf.remaining() > 0
        || uncompressedDirectBuf.remaining() == 0 || userBufLen > 0);
}
```

### 3. compress()

compress() 方法用于获取压缩后的数据，它需要处理 needsInput() 返回 false 的几种情况。

如果压缩数据缓冲区有数据，即 compressedDirectBuf 中还有数据，则读取这部分数据，并返回。

如果该缓冲区为空，则需要压缩数据。首先清理 compressedDirectBuf，这个清理（即 clear() 调用和 limit() 调用）是一个典型的 Buffer 操作，具体函数的意义在第 4 章会讲。待压缩的数据有两个来源，输入缓冲区 uncompressedDirectBuf 或者“借用”的数据缓冲区。

如果输入缓冲区没有数据，那待压缩数据可能（可以在没有任何带压缩数据的情况下调用 compress() 方法）在“借用”的数据缓冲区里，这时使用 setInputFromSavedData() 方法复制“借用”数据缓冲区中的数据到 uncompressedDirectBuf 中。setInputFromSavedData() 函数调用结束后，待压缩数据缓冲区里还没有数据，则设置 finished 标记位，并返回 0，表明压缩数据已经读完。

uncompressedDirectBuf 中的数据，利用前面已经介绍过的 native 方法 compressBytesDirect() 进行压缩，压缩后的数据保存在 compressedDirectBuf 中。由于待压缩数据缓冲区和压缩数据缓冲区的大小是一样的，所以 uncompressedDirectBuf 中的数据是一次被处理完的。compressBytesDirect() 调用结束后，需要再次设置缓冲区的标记，并根据情况复制数据到 compress() 的参数 b 提供的缓冲区中。相关代码如下：

```
public synchronized int compress(byte[] b, int off, int len)
    ......
    // 是否还有未取走的已经压缩的数据
    int n = compressedDirectBuf.remaining();
    if (n > 0) {
        n = Math.min(n, len);
        ((ByteBuffer) compressedDirectBuf).get(b, off, n);
        bytesWritten += n;
        return n;
    }
```

```
    // 清理压缩数据缓冲区
    compressedDirectBuf.clear();
    compressedDirectBuf.limit(0);
    if (0 == uncompressedDirectBuf.position()) {
       // 输入数据缓冲区没有数据
       setInputFromSavedData();
       if (0 == uncompressedDirectBuf.position()) {
          // 真的没数据，设置标记位，并返回
          finished = true;
          return 0;
       }
    }

    // 压缩数据
    n = compressBytesDirect();
    compressedDirectBuf.limit(n);
    uncompressedDirectBuf.clear();

    // 本地方法以及处理完所有的数据，设置 finished 标志位
    if (0 == userBufLen) {
       finished = true;
    }

    n = Math.min(n, len);
    bytesWritten += n;
    ((ByteBuffer) compressedDirectBuf).get(b, off, n);

    return n;
}
```

### 4. finished()

最后要分析的成员函数是 finished()。如图 3-8 所示，finished() 返回 true，表明压缩过程已经结束，压缩过程结束其实包含了多个条件，包括 finish 标志位和 finished 标志位必须都为 true，以及 compressedDirectBuf 中没有未取走的数据。其中 finish 为 true，表示用户确认已经完成数据的输入过程，finished 表明压缩器中没有待压缩的数据，这三个条件缺一不可。相关代码如下：

```
public synchronized boolean finished() {
   // Check if all uncompressed data has been consumed
   return (finish && finished && compressedDirectBuf.remaining() == 0);
}
```

SnappyCompressor 中的其他方法都相对简单，不再一一介绍了。通过对 SnappyCompressor 的实现分析，我们了解了为何压缩器要求图 3-8 那样的方法配合过程，有助于读者在 Hadoop 中引入一些新的压缩算法。

## 3.3　小结

本章涉及了 org.apache.hadoop.io 包下最重要的两部分内容：序列化和压缩。

序列化广泛应用于分布式数据处理中，是交换数据必备的能力。Hadoop 没有使用 Java 内建的序列化机制，而是引入了紧凑、快速、轻便和可扩展的 Writable 接口。Writable 接口通过 write() 和 readFields() 方法声明了序列化和反序列化的功能。在此基础上，分析了 Writable 的一些典型子类的实现，包括 Java 基本类型对应的 Writable 封装和 ObjectWritable，它们为用户使用 Hadoop 提供了很多方便。

压缩是 org.apache.hadoop.io 包中实现的另一个重要功能，Hadoop 必须支持多种压缩算法，如何灵活地支持这些算法呢？ Hadoop 实现了压缩框架，包括编码 / 解码器及其工厂、压缩器 / 解压器以及压缩流 / 解压缩流三种组件，它们相互配合满足了用户对压缩功能的需求。最后，以 Cloudera 发行版中 Snappy 压缩功能的实现为例，介绍了在压缩框架里如何集成新的压缩算法并支持本地库，以提高压缩效率。

# 第 4 章　Hadoop 远程过程调用

作为典型的分布式系统，Hadoop 中各个实体间存在着大量的交互，远程过程调用让用户可以像调用本地方法一样调用另外一个应用程序提供的服务，而不必设计和开发相关的信息发送、处理和接收等具体代码，是一种重要的分布式计算技术，它提高了程序的互操作性，在 Hadoop 的实现中得到广泛的应用。

## 4.1　远程过程调用基础知识

很多分布式系统，如浏览器和服务器间的通信，需要在不同的实体中显式交换消息，处理诸如消息的编解码、消息的发送和接收等非常具体的任务。如何隐藏底层通信的细节，是实现分布式系统访问透明性的重要议题之一。

远程过程调用（Remote Procedure Call，RPC）由 Birrell 和 Nelson 于 1984 年引入分布式计算领域，是解决上述访问透明性的一套精妙方案。随着 RPC 的发展，出现了大量的相关技术，如 XML-RPC、JSON-RPC（JavaScript Object NotationRPC），CORBA（Common Object Request Broker Architecture，公共对象请求代理体系结构）和 RMI（Remote Method Invocation，远程方法调用）等。本章讨论的 Hadoop IPC（Inter-Process Communication，进程间通信）属于 RPC 的一种比较简单的实现。

### 4.1.1　RPC 原理

简要地说，RPC 就是允许程序调用位于其他机器上的过程（也可以是同一台机器的不同进程）。当机器 A 上的进程调用机器 B 上的进程时，A 上的调用进程被挂起，而 B 上的被调用进程开始执行。调用方使用参数将信息传送给被调用方，然后通过传回的结果得到信息。在这个过程中，A 是 RPC 客户，B 是 RPC 服务器。同时，编程人员看不到任何消息的传递，其行为如同一个过程到另一个过程的调用一样。

图 4-1 是一个常规的过程调用，概括了进程的控制流同步模型。

传统的过程调用中，主程序将参数压入栈内并调用过程，这时候主程序停止执行并开始执行相应的过程。被调用的过程从栈中获取参数，然后执行过程函数；执行完毕后，将返回参数入栈（或者保存在寄存器里），并将控制权交还给调用方。调用方获取返回参数，并继续执行。

RPC 跨越了不同的进程（theClient 和 theServer），其调用示例如图 4-2 所示。

当 theClient 需要调用一个远程过程的时候，通过 theClient 将参数打包成为一个消息，并附加被调用过程的名字（指明要调用服务器上的哪一个过程），然后发送消息到服务器。theClient 发送完消息以后，必须等待服务器的应答，这个时候，执行流是空闲的。

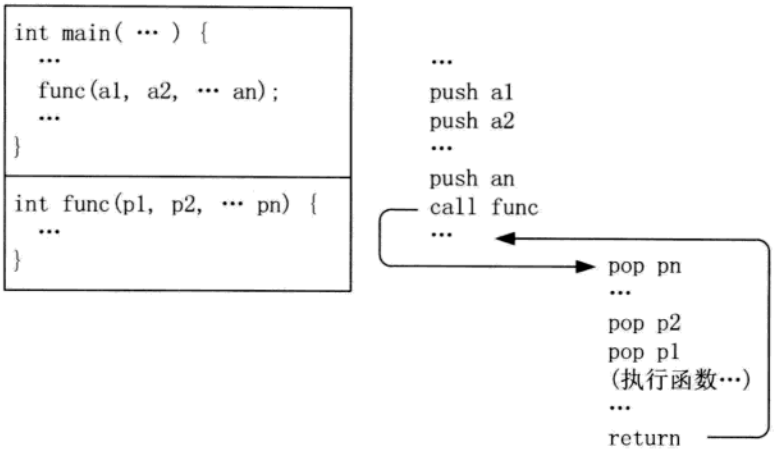



图 4-1　常规过程调用示例

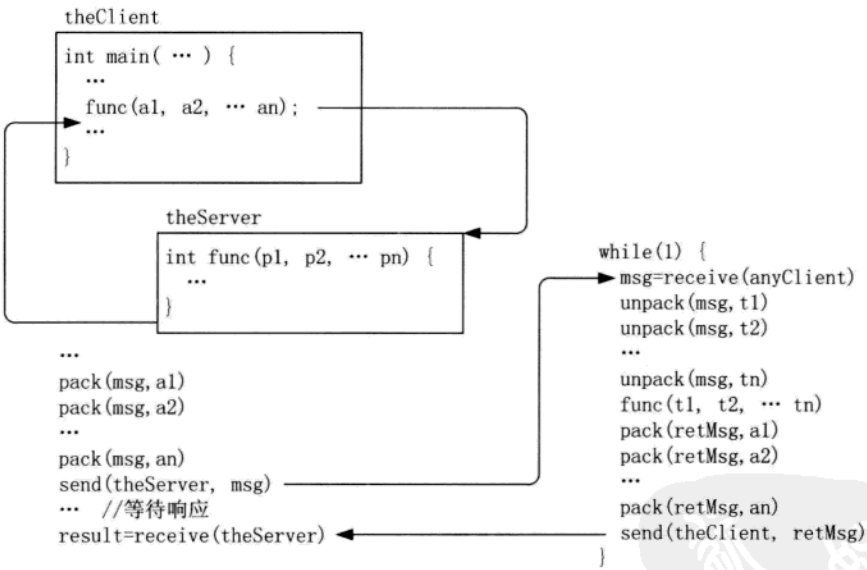



图 4-2　RPC 调用示例

RPC 的 Server 运行时会阻塞在接收消息的调用上，当接到客户端的请求后，它会解包以获取请求参数，这类似传统过程调用中，被调用函数从栈中接收参数，然后确定调用过程的名字并调用相应过程。调用结束后，返回值通过主程序打包并发送回客户端，通知客户端调用结束。

### 4.1.2 RPC 机制的实现

RPC 机制使两个进程间能够使用传统过程调用中的控制流模型来进行交互。RPC 看似简单，但在实现的时候，存在很多的具体问题，例如如何在系统有差异的情况下进行参数和结果的传递，如果通信双方的某一个机器崩溃了，该如何解决等。为了保证 RPC 具有与本地调用相应的形式，实现 RPC 需要关注下面三个问题：

- RPC 的语法应该与高级程序设计语言中的本地过程调用的语法有相同的外观；
- RPC 和本地调用的语义尽可能地类似，如参数传递中包含的语义；
- RPC 的接收者（也就是服务器上业务代码的执行）应该如同一个传统的调用。

RPC 的实现一般采用图 4-3 的形式。

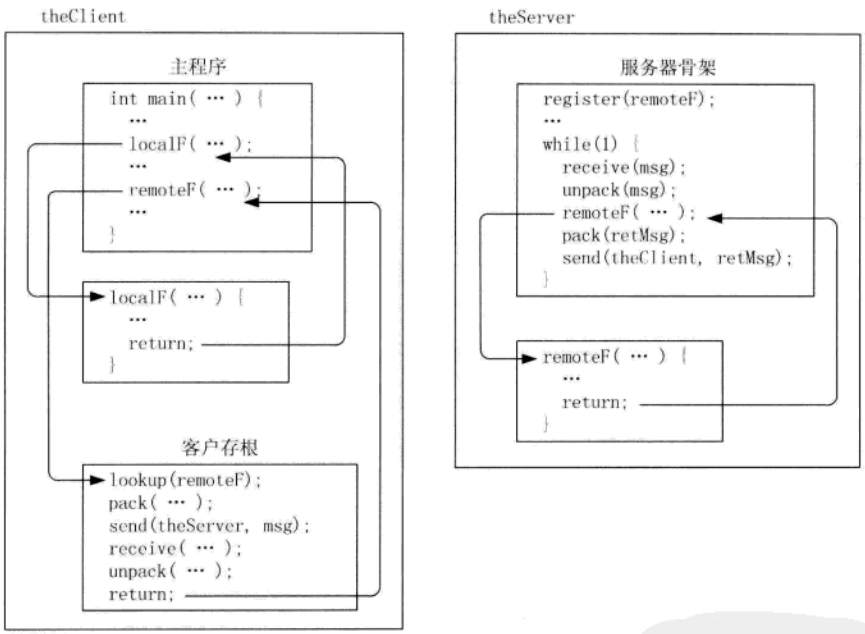



图 4-3 RPC 的实现

RPC 引入客户存根（Client Stub）和服务器骨架（Server Skeleton）解决了前面提及的问题。细心的读者可以发现，如果调用方法的接口确定，那么图 4-2 中 RPC 调用过程的很多步骤都可以自动生成，生成的这些步骤正是客户存根和服务器骨架的重要组成部分。

**注意** 篇幅限制，我们不讨论 RPC 中的服务器注册（register）和客户对服务器定位（lookup）的过程，感兴趣的读者可以参考分布式计算的相关资料。

客户端应用使用相同的方式调用普通方法 localF 和远程方法 remoteF。也就是说，都需要进行相同的参数入栈，并执行对应的子程序。唯一不同的是，remoteF 的客户存根只是简

单地将参数打包成一个消息，并请求将此消息发送到服务器。然后，存根通过调用 receive 过程，随即阻塞自己，直到收到响应消息。

另一端的服务器骨架是客户存根的等价物，它也是一段代码，用来将通过网络输入的请求转换为本地调用。服务器骨架先调用 receive，然后阻塞，等待消息输入。收到消息后，服务器骨架将参数从消息中提取出来，然后以常规方式调用服务器上的相应过程，即图中的 remoteF 过程。从 remoteF 的角度看，过程就好像是由普通客户直接调用一样，参数和返回地址都位于栈中。remoteF 执行完成，结果返回到服务器骨架，由骨架打包成消息发送（调用 send）给客户存根。服务器骨架再次调用 receive，等待下一条消息。

客户端的 receive 随着消息的到达而返回，客户存根检查消息并解包返回参数，通过 return 将结果传递到主程序中。调用者在 remoteF 结束后重新获得控制权，它唯一知道的就是已经得到了所要的数据，它并不清楚操作是通过本地方法进行的，还是远程完成的。

上述 RPC 实现的精彩之处在于：客户方可以简单地忽略不需要关心的内容。客户只需要调用一个普通的、本地的过程，就可以访问远程服务，不需要调用 send 或 receive 这些底层操作，所有的消息传递细节都隐藏在 RPC 的库过程中，就如同传统库中隐藏了执行实际系统调用的细节一样。应该说，RPC 的实现基本上解决了我们关注的三个问题。

前面已经提到，如果调用接口确定，上述过程中的很多组件都可以自动生成，RPC 通过接口定义语言（Interface Definition Language，IDL）描述调用接口的相关信息。IDL 文件可以包含类型定义、常量声明、进行参数传递时需要的一些其他信息和注释。

IDL 文件编写完以后，通过调用 IDL 编译器编译，一般会输出以下文件：

❑头文件（C 语言，包括一些函数的定义）

❑客户存根

❑服务器骨架

应用开发人员编写客户端和服务器代码，编译、连接（客户代码和客户存根目标文件和运行库连接，服务器代码和服务器骨架目标文件和运行库连接），就可以得到客户程序和服务器程序，完成一个包含 RPC 的应用程序的开发。

### 4.1.3　Java 远程方法调用

Java 远程方法调用（Remote Method Invocation，RMI）是 Java 的一个核心 API 和类库，允许一个 Java 虚拟机上运行的 Java 程序调用不同虚拟机上运行的对象中的方法，即使这两个虚拟机运行于物理隔离的不同主机上。在某种程度上，RMI 可以看成 RPC 的 Java 升级版。

和 RPC 一样，包含 RMI 的 Java 应用程序通常包括服务器程序和客户端程序。典型的服务器应用程序将创建多个远程对象（Remote Object），使这些对象能够被客户端引用，并等待客户端调用远程对象的方法；而典型的客户端程序则从服务器中得到一个或多个远程对象的引用，然后调用位于服务器端的远程对象方法。在这个过程中，RMI 提供了和 RPC 中类似的、标准的 Stub/Skeleton 机制。

Java RMI 和 RPC 工作原理非常类似，Stub 代表可以被客户端引用的远程对象，位于客

户端，并保存着远程对象的接口和方法列表。客户端应用调用远程对象时，Stub 将调用请求，通过 RMI 的基础结构转发到远程对象上。接收到调用请求时，服务器端的 Skeleton 对象处理有关调用“远方”对象中的所有细节，并调用 Skeleton 对象。

**注意** Java 远程方法调用依赖于前面介绍过的 Java 对象序列化机制，它将调用的参数和返回值（Java 基本类型或者对象）序列化并在网络中传递。

### 1. Java 远程方法调用实例

前面已经介绍了大量的 RPC、RMI 的背景，下面通过一个简单的实例，详细介绍 Java 远程方法调用，具体的实现请参考包 org.hadoopinternal.rmi 下的相关代码。

远程对象是指这样一类对象：除了对象本身所在的虚拟机，其他虚拟机也可以调用此对象的方法，而且这些虚拟机一般运行在不同的计算机上。每个远程对象都实现了一个或多个远程接口（Remote Interface），远程接口声明了可以由外部系统调用的远程对象方法。

Java RMI 的开发往往从远程接口的定义开始，本例使用的远程接口如下：

```
import java.rmi.Remote;
import java.rmi.RemoteException;

// 远程接口必须声明为 public
// 远程接口必须继承自 java.rmi.Remote
public interface RMIQueryStatus extends Remote {

   // 远程接口中的函数必须将 java.rmi.RemoteException 声明于其 throws 子句内
   RMIFileStatus getFileStatus(String filename)
      throws RemoteException;
}
```

RMIQueryStatus 接口中的 getFileStatus() 方法接受一个字符串参数 filename，返回一个 RMIQueryStatus 对象，这个方法简单地模拟了文件系统中对文件元信息的查询功能。

RMIQueryStatus 是一个典型的 RMI 远程接口，它遵循以下规则：

- 远程接口必须声明为 public，否则客户端试着装载“实现远程接口”的远端对象时，会收到错误的消息。
- 远程接口必须继承自 java.rmi.Remote。
- 远程接口中的每一个方法，除了自定义的异常之外，必须将 java.rmi.RemoteException 声明于其 throws 子句中。
- 在远程方法声明中，作为参数或者返回值的远程对象，或者包含在其他非远程对象中的远程对象，必须声明为其对应的远程接口，而不是实际的实现类。

**注意** 最后一条规则并没有在例子中体现，getFileStatus() 的返回值 RMIFileStatus 是个简单的 Java 类，并不包含任何远程对象，读者如果需要了解详情，可参考其他 Java 远程方法调用资料。

有了远程接口 RMIQueryStatus，就可以定义远程对象类了。远程对象类继承自 UnicastRemoteObject，并实现远程接口 RMIQueryStatus。在例子中，RMIQueryStatusImpl 类是远程对象类，这个类可以拥有额外的成员函数，但只有远程接口中的函数才可以被 RMI 客户端访问。

远程接口中的 getFileStatus() 函数非常简单，它根据输入参数构造一个 RMIFileStatus 对象，然后返回该对象。从下面的代码可以看到，远程接口中函数的实现和普通的成员函数并没有太大的区别（除了它可能抛出 java.rmi.RemoteException 异常以外）。相关代码如下：

```
//RMIQueryStatusImpl 继承自 UnicastRemoteObject，并实现了 RMIQueryStatus 接口
public class RMIQueryStatusImpl
      extends UnicastRemoteObject implements RMIQueryStatus {
   ……
   @Override
   public RMIFileStatus getFileStatus(String filename) throws ……{
      RMIFileStatus status=new RMIFileStatus(filename);
      ……
      return status;
   }
   ……
}
```

客户端如果要访问一个远程对象，必须获得一个本地的存根对象，如何获得本地存根对象呢？ Java 引入了 RMI 远端对象注册点（registry）解决了这个问题。

RMI 服务器创建远程对象类的对象，并将这些对象绑定到注册服务上；RMI 客户程序则通过注册点，利用名字查找远程对象，并构造出相应的本地存根对象。然后，RMI 客户程序就可以调用远程方法了。

RMI 远端对象注册点 rmiregistry 是 Java 自带的一个工具，该应用程序位于 Java 运行环境的 bin 目录下。通过下面的命令启动 Java RMI 注册服务（使用默认端口）：

```
rmiregistry
```

然后启动服务器 RMIQueryStatusServer，该服务器的 main 方法先创建 RMIQueryStatusImpl 对象，并与刚才启动的对象注册点对话，将远端对象绑定到名字“rmi://yourserver.com:12090/query”上（应用这个例子的时候，请将这个 URI 的主机部分改为相应的主机名）。代码如下：

```
public static void main(String[] args) {
   try {
      // 创建 RMIQueryStatusImpl 对象
      RMIQueryStatusImpl queryService=new RMIQueryStatusImpl();

      LocateRegistry.createRegistry(12090);

      // 绑定远端对象到名字
      Naming.rebind(RMI_URL, queryService);
```

```
            System.out.println("Server ready");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

虽然 main() 方法的执行很快就结束了，但服务器还会继续运行，不过，用户需要写的服务器代码就只有这么多。

现在我们可以讨论客户端了。

在普通 Java 对象调用一个方法前，它需要这个方法所属对象的一个引用。客户端对象在调用远程方法之前，则需要被调用方法所属对象的一个远程引用。客户端 RMIQueryStatusClient 从远端对象注册点中，通过 Naming.lookup() 获取远程引用。一旦获取对象引用，并恢复了对象类型，客户端就可以使用此引用，调用远程对象的远程方法，即 getFileStatus() 方法。相关代码如下：

```
try {
    // 创建 RMIQueryStatusImpl 对象
    RMIQueryStatus query=
            (RMIQueryStatus)Naming.lookup(RMIQueryStatusServer.RMI_URL);

    // 调用远程方法：该调用如同调用本地方法
    RMIFileStatus status=query.getFileStatus("/tmp/testRMI");
    System.out.println(status);
} catch (RemoteException e) {
    // 远程方法和普通方法的一个差别：远程方法可能抛出 RemoteException 异常
    e.printStackTrace();
}
```

应该说，调用远程方法和调用本地方法有着近乎相同的“感观”，一个主要的差别在于需要通过命名服务获得远程对象的引用；另外，和本地方法调用不一样，远程方法可能抛出 RemoteException 异常，客户端程序需要处理该异常。

### 2. Java 远程方法调用实例类图详解

上面的实例虽然简单，但是涉及了大量的 Java 远程方法调用使用细节。图 4-4 给出了实例中各对象的关系，其中包括 java.rmi 和 java.io 中的相关类和接口。仔细分析实例相关代码，并没有涉及远程方法调用中的客户端存根和服务器骨架等基础设施，其实这些都隐藏在相关类和接口中。

图 4-4 中最上面是接口 java.io.Serialization 和 java.rmi.Remote，其中 java.io.Serialization 用于对调用参数和返回值进行序列化和反序列化。RMIQueryStatus.getFileStatus() 方法的返回值 RMIFileStatus 需要从服务器传输到客户端，那么，RMIFileStatus 类就必须实现 java.io.Serialization。java.rmi.Remote 和 java.io.Serialization 一样，没有声明任何方法，它只用于标识作为远程对象的对象。

抽象类 RemoteObject 是作用于远程对象的特殊版本的 java.lang.Object，提供了对远程对象有意义的 toString()、hashCode() 等方法。

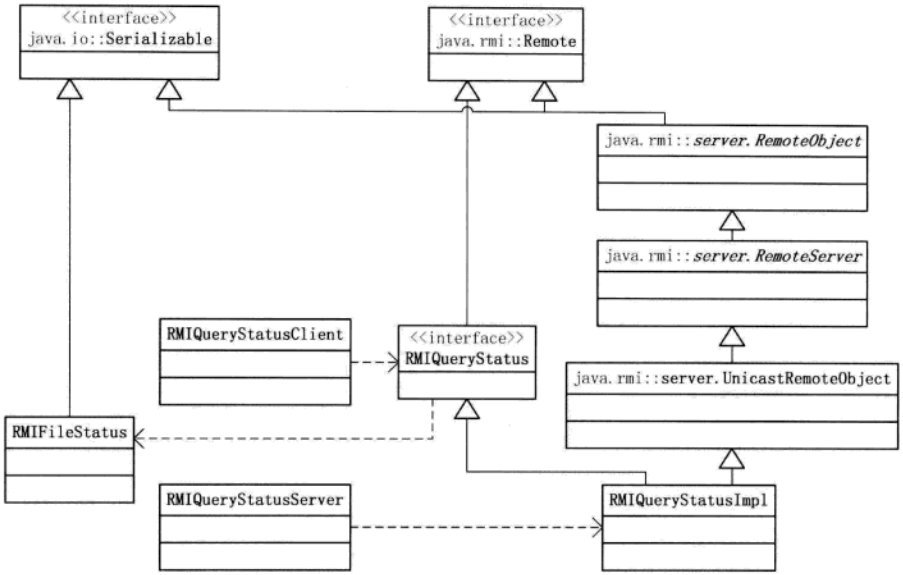



图 4-4　Java RMI 实例类图

RemoteServer 扩展了 RemoteObject 并提供了大多数服务器对象需要的工具方法。UnicastRemoteObject 是 RemoteServer 的一个具体子类，对一个远程对象类，最简单的方法就是继承 UnicastRemoteObject，如本节的例子。UnicastRemoteObject 类实现了处理一般远程对象的任务，如序列化和反序列化参数及返回值，使用 TCP 和客户端通信等功能。通过继承 UnicastRemoteObject，RMIQueryStatusImpl 获得了远程对象类中需要的 RMI 骨架功能。

客户端 RMIQueryStatusClient 的工作依赖于 RMI 存根，这个存根是通过 Java 的代理机制 java.lang.reflect.Proxy（下一节介绍）动态生成的。存根包含了 Remote 接口的信息，在例子中，这个信息就是：远程对象 RMIQueryStatusImpl 中提供了远程方法 getFileStatus()。在一些特殊情况下，开发人员需要自己生成存根，与 RPC 提供 IDL 和 IDL 编译器类似，JDK 提供 rmic 工具辅助用户开发。该工具可以根据 RMIQueryStatusImpl 的字节码生成相应的存根：RMIQueryStatusImpl_Stub.class。如果使用 rmic 的 -keep 选项，生成器将会保持中间结果 RMIQueryStatusImpl_Stub.java，感兴趣的读者可以分析它的代码，看一个典型的 RMI 存根都包含哪些成员。由于充分使用了 Java 的面向对象技术和反射能力，相对于 RPC，Java RMI 的开发过程显得非常简单。

**注意**　JDK 5.0 版本以前，用户需要通过 rmic 工具生成客户端存根，在更早的 Java 版本 JDK 1.1 中，服务器骨架也必须由 rmic 产生。

## 4.2 Java 动态代理

Hadoop 远程过程调用实现使用 Java 动态代理和新输入 / 输出系统（New Input/Output，NIO）。本节介绍 Java 动态代理相关内容。

Java 动态代理类位于 java.lang.reflect 包下，主要包括 java.lang.reflect.Proxy 和 java.lang.reflect.InvocationHandler。

### 4.2.1 创建代理接口

从包和类名可以知道，java.lang.reflect.Proxy 和 Java 反射有关，同时也和代理模式有关。代理（Proxy）在《现代汉语词典》里的解释是：受当事人委托，代表他进行某种活动，如贸易、诉讼、纳税、签订合同等。如果把这个解释应用到面向对象领域，应该是：代理对象，代表另一个对象（目标对象，Target），执行相关活动。也就是说，在实际应用中使用代理对象作为中介，代替目标对象。

在 Java 中，代理对象往往实现和目标对象一致的接口，并作为目标对象的代替，接收对象用户（Client）的调用，并将全部或部分调用转发给目标对象，如图 4-5 所示。在这个过程中，实现代理接口和调用转发，是代理对象必须完成的两个重要任务，缺一不可。

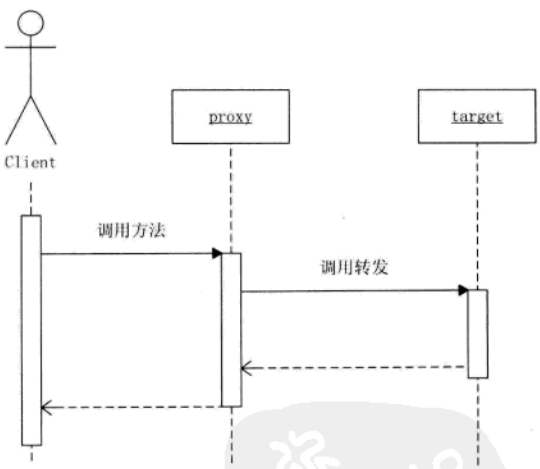



图 4-5 代理序列图

在完成上述两个任务的同时，代理对象可以在调用转发前或者调用转发后执行一些功能，这些功能可以很简单，如输出日志、实施访问控制等，也可以非常复杂，包括访问数据库，或者加载远程资源等。也正是这些在调用前后执行的附加逻辑，体现了代理对象的价值。

java.lang.reflect.Proxy 中的另一个关键字是 reflect（反射），反射提供了一个非常丰富而精心设计的工具集，以便编写能够动态操纵 Java 代码的程序。在前面 Java 配置管理和 Hadoop 的序列化机制中已经看到了反射的巨大威力。java.lang.reflect.Proxy 提供了用于创建动态代理类和对象的静态方法。也就是说，通过 java.lang.reflect.Proxy 可以动态地创建某个接口实现，如图 4-5 中的 proxy 对象。代理对象的另一个重要任务是调用转发，一般由 java.lang.reflect.InvocationHandler 接口的实例完成，接口实例处理接口上的方法调用并返回结果。

Proxy 类提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。

java.lang.reflect.Proxy 中比较重要的方法有：

```
public class Proxy implements java.io.Serializable {
   public static Class<?> getProxyClass(ClassLoader loader,
                                        Class<?>... interfaces)
   ......
   public static Object newProxyInstance(ClassLoader loader,
                                         Class<?>[] interfaces,
                                         InvocationHandler h)
   ......
   public static boolean isProxyClass(Class<?> cl)
   ......
   public static InvocationHandler getInvocationHandler(Object proxy)
   ......
}
```

其中，Proxy 的静态方法 getProxyClass() 用于获得代理类的 java.lang.Class 对象。该代理类将定义在指定的类加载器（参数 loader）中，并将实现参数 interfaces 指定的所有接口。注意，这个类只会创建一次，如果再次传入相同的 loader 和 interfaces 给 newProxyInstance() 方法，获得的也只是第一次调用创建的那个 java.lang.Class 对象。

通过 Proxy.getProxyClass() 获得的代理类都包含一个构造函数，该构造函数需要一个 java.lang.reflect.InvocationHandler（后面会介绍这个接口）的实例（参数 h）。也就是说，假设 Proxy.getProxyClass() 获得的代理类的名字是 $Proxy1，那么它有如下构造函数（注意，下面代码只用于说明如何通过代理类构造相应的代理对象，并不是真实的 Java 代码）：

```
public class $Proxy1 {
   public $Proxy1(InvocationHandler h)
   ......
}
```

通过 Java 的反射机制就可以利用代理类获得相应的代理对象，代码如下：

```
Class cl = getProxyClass(loader, interfaces);   // 获取代理类

// 获得构造函数
Constructor cons = cl.getConstructor(
      new Class[] {InvocationHandler.class });

Object proxy = cons.newInstance(new Object[] { h }); // 创建对象
```

其中，cl 是代理类的 java.lang.Class 对象，通过该对象可以获得代理类的构造方法 cons，最终，利用 cons.newInstance() 创建了一个代理对象 proxy。对 Java 反射不熟悉的读者，请参考相关资料，如 JDK 文档中 java.lang.reflect.Proxy 部分。

上述创建代理对象的三步调用，可以利用 Proxy.newProxyInstance() 一次完成。在 Hadoop IPC 中，一般用 newProxyInstance() 返回一个或多个指定接口（通过参数 interfaces）的代理类实例，在该接口上的方法调用，将指派到指定的调用处理程序上，即 InvocationHandler 的实例 h 上。也就是说，在代理对象上的方法调用，都会转发给实现了 InvocationHandler 接口的对象 h。

此外，Proxy 类还提供以下方法：

- isProxyClass()：判断某 java.lang.Class 对象是否是代理类。
- getInvocationHandler()：获取代理实例对应的调用处理程序（即构造代理实例时传入的 InvocationHandler 实例）。

### 4.2.2 调用转发

前面提过，代理对象有两个重要任务，其中，创建接口实现由 java.lang.reflect.Proxy 完成，而代理的另一个任务调用转发则通过 java.lang.reflect.InvocationHandler 的实例完成。

InvocationHandler 的实例也叫调用句柄实例。InvocationHandler 是一个接口，invoke() 是这个接口的唯一方法，当用户调用 newProxyInstance() 返回的对象上的方法时，该对象将方法调用转发给调用句柄实例，即调用 InvocationHandler 实例的 invoke() 方法。相关代码如下：

```
public interface InvocationHandler {
   public Object invoke(Object proxy, Method method, Object[] args)
          throws Throwable;
}
```

传递给 invoke() 方法的 3 个参数分别是：

- proxy：代理对象本身。
- method：用户调用的代理对象上的方法。
- args：传递给该方法的参数。

而 invoke() 的返回值将作为代理对象方法的调用结果返回给用户。其中，参数 method 的类型是 java.lang.reflect.Method。Method 类也是 Java 反射中的一个重要抽象，它提供了关于类或接口上某个方法以及如何访问该方法的信息。通过 java.lang.reflect.Method 的 invoke() 方法，可以在某指定对象上调用对象的方法。方法声明如下：

```
public Object invoke(Object obj, Object... args)
       throws IllegalAccessException, IllegalArgumentException,
       InvocationTargetException
```

假设目标对象是 target，那么最简单的调用转发实现如下：

```
public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
       return method.invoke(target, args );
}
```

以上代码通过 Method 对象的 invoke() 方法，在目标对象 target 上使用相同的参数 args，调用了同一个方法（它们都实现了同一个接口），并将结果返回给调用者。

更复杂的 invoke() 实现，可以包含调用前或调用后的参数处理；将调用转发给不同的对象，或者是多个对象等。图 4-6 总结了上述调用转发过程中重要对象的配合。

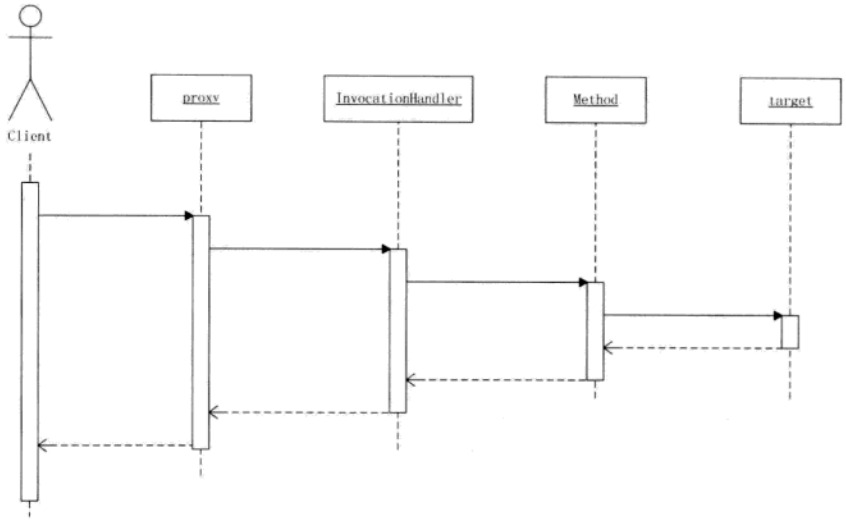



图 4-6　Java 动态代理序列图

### 4.2.3　动态代理实例

介绍了 Java 动态代理的基本知识后，下面通过例子进一步讨论代理对象，具体代码请参考 org.hadoopinternal.dynamicproxy。dynamicproxy 实例中的主要类和接口如图 4-7 所示。

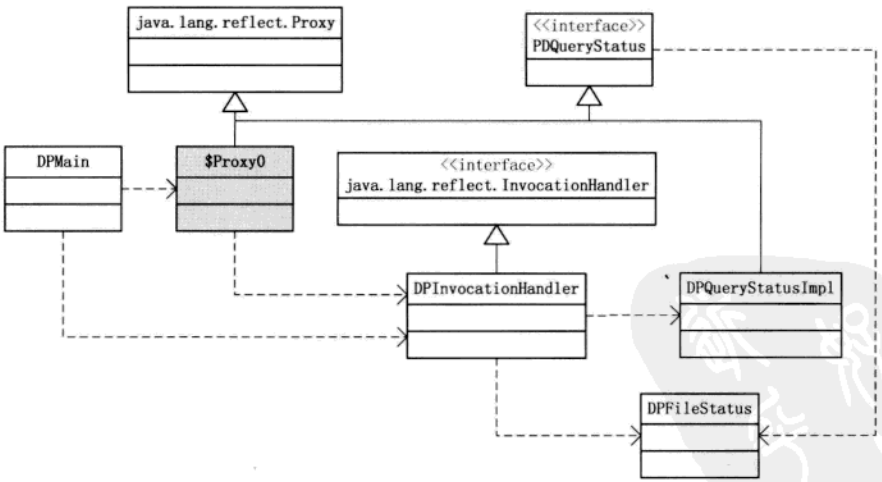



图 4-7　org.hadoopinternal.dynamicproxy 类图

Java 动态代理中大量使用了 Java 接口，与 RMI 的例子类似，dynamicproxy 中使用 PDQueryStatus 接口，它包含一个非常简单的 getFileStatus() 方法，该方法接受一个字符串参数，返回一个类型为 DPFileStatus 的 Java 对象。和远程方法调用不同，对 PDQueryStatus 接

口，Java 动态代理机制没有额外的要求。相关代码如下：

```
public interface PDQueryStatus {
    DPFileStatus getFileStatus(String filename);
}
```

DPQueryStatusImpl 简单地实现了 PDQueryStatus 接口，同时，DPQueryStatusImpl 的实例也是这个范例的目标对象。通过图 4-7 可以发现，和 DPQueryStatusImpl 有联系的是 DPInvocationHandler。DPInvocationHandler 继承自 InvocationHandler，也就是说，例子中调用转发由 DPInvocationHandler 实现。

DPInvocationHandler 类的唯一成员变量是目标对象 dpqs，它在类构造的时候由调用者传入。相关代码如下：

```
public class DPInvocationHandler implements InvocationHandler {
    private DPQueryStatusImpl dpqs; // 目标对象

    // 构造函数，传入目标对象
    public DPInvocationHandler(DPQueryStatusImpl dpqs) {
        this.dpqs = dpqs;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        Object ret = null;

        // 实现了附加的功能，往控制台输出调用参数的 String 表示
        String msg1 = MessageFormat.format("Calling method {0}({1})",
                                           method.getName(),
                                           Arrays.toString(args));
        System.out.println(msg1);

        // 调用转发
        ret = method.invoke(dpqs, args);

        // 其他附加功能
        ......
        return ret;
    }
}
```

DPInvocationHandler 中最重要的方法是 invoke()，它来自 InvocationHandler 接口。如果代理对象的客户调用 getFileStatus() 方法，DPInvocationHandler.invoke() 将会被调用，传入 invoke() 方法的 method 对象，就是 PDQueryStatus.getFileStatus() 方法对应的 java.lang.reflect.Method 实例；传入的 args 数组只有一个元素，该元素是 getFileStatus 的唯一参数 filename。

在 DPInvocationHandler.invoke 中，完成代理对象调用转发的是 method.invoke：

```
ret = method.invoke(dpqs, args);
```

当 method 是方法 PDQueryStatus.getFileStatus() 时，其效果相当于：

```
ret = dpqs.getFileStatus(……);
```

值得注意的是，如果 PDQueryStatus 接口有多个方法，对不同方法的调用，代理对象最终调用的都是 DPInvocationHandler.invoke() 方法。通过 method 参数，可以在 invoke() 的处理过程中获知代理对象的客户到底调用了哪一个方法。

在 DPInvocationHandler.invoke() 的 method.invoke 调用前后，实现了一些附加功能，例如在控制台上输出调用参数和返回对象的 String 表示。当然，也可以实现一些更复杂的功能，如根据参数 method 的不同值，调用不同的目标对象等。

讨论完代理的调用转发功能后，再来分析代理对象的创建过程，这部分的代码在 org.hadoopinternal.dynamicproxy.DPMain() 中，静态方法 create() 用于创建一个到 DPQueryStatusImpl 实例（也就是目标对象）的代理对象。相关代码如下：

```
public static PDQueryStatus create(DPQueryStatusImpl dpqs) {
   // 调用 newProxyInstance 需要的 Class<?>[] 参数
   Class<?>[] interfaces=new Class[] { PDQueryStatus.class };

   // 调用 newProxyInstance 需要的 DPInvocationHandler 参数
   DPInvocationHandler handler = new DPInvocationHandler(dpqs);

   Object result
           =Proxy.newProxyInstance(dpqs.getClass().getClassLoader(),
                   interfaces, handler);
   ……
   return (PDQueryStatus) result;
}
```

在 DPMain.create() 方法中，首先为调用 java.lang.reflect.Proxy.newProxyInstance() 准备所需的参数，其中，interfaces 是动态代理类需要实现的代理接口，它是一个 Class 数组（Class<?>[]），也就是说，代理对象可以同时实现一系列接口。

本例中，代理对象只实现了一个接口，即 PDQueryStatus。变量 handler 的类型就是前面介绍过的 DPInvocationHandler，它是 newProxyInstance() 需要的调用处理程序，实现了代理对象的调用转发功能。准备完参数以后，通过 Proxy.newProxyInstance() 创建了一个实现指定接口代理 PDQueryStatus 并带有指定调用处理程序 DPInvocationHandler 的代理实例，最后，通过强制转换操作把它转换成 PDQueryStatus。

在 DPMain.main() 中，变量 query 是一个代理对象，实现了 PDQueryStatus 接口，可以在 query 上调用 PDQueryStatus 的 getFileStatus() 方法，如同调用一个普通对象的方法。同时，可以观察到，这个调用会转发到 DPInvocationHandler 的 invoke() 方法，最终调用 DPQueryStatusImpl 的同名方法。相关代码如下：

```
public static void main(String[] args) {
   try {
```

```
            PDQueryStatus query = DPMain.create(new DPQueryStatusImpl());
            DPFileStatus status = query.getFileStatus("/tmp/testDP");
            System.out.println(status);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

读者可能会好奇，query 变量的类型是什么？通过 Java 的反射机制可以了解到，query 的类型是 $Proxy0（也可能是 $Proxy1、$Proxy2，它们总是以 $Proxy 开头），它继承自 java.lang.reflect.Proxy，是一个 public、final 的类，上述信息也体现在图 4-7 中。

## 4.3 Java NIO

本节介绍一些 Java NIO 的背景知识，熟悉 NIO 的读者可以跳过这部分内容。

### 4.3.1 Java 基本套接字

Java 的最初版本中，只提供了基于流的基本套接字（socket）实现，socket 是两台主机间的一个连接，可以进行 7 项基本操作：

- ❑连接远程机器。
- ❑发送数据。
- ❑接收数据。
- ❑关闭连接。
- ❑绑定端口。
- ❑监听入站数据。
- ❑在所绑定端口上接受来自远程机器的连接。

其中，前四项用于客户端，后六项用于服务器，最后三项只有服务器才需要，即等待客户端的连接，这些操作通过 ServerSocket 类实现。

Java 程序一般以下面的顺序使用客户端 socket：

1）通过构造函数构造一个新 Socket 对象；

2）将 socket 连接到远程主机；

3）连接建立以后，本地和远程主机从 socket 取得输出流和输入流，使用这两个流彼此发送数据，两台主机都可以同时发送和接收数据；

4）数据传输完毕后，关闭连接。

上面讨论的这些过程，对应的客户端 socket 方法如下（这里只给出了 java.net.Socket 中很少的方法，大量其他方法请参考 JDK 的相关文档）：

```
public class Socket {
    ......
    // 构造函数
```

```
    public Socket()
    ……
    // 连接到远程主机
    public void connect(SocketAddress endpoint)
    ……
    // 获得输入流
    public InputStream getInputStream()
    ……
    // 获得输出流
    public OutputStream getOutputStream()
    ……
    // 关闭连接
    public synchronized void close()
}
```

值得注意的是，getOutputStream() 方法返回一个原始的 OutputStream，用于将应用程序的数据写入 socket 的另一端；getInputStream() 方法得到的 InputStream，用于读取 socket 另一端发送过来的数据。Java 的 OutputStream 和 InputStream 没有提供异步 I/O 操作的能力，也就是说，在 OutputStream 上的写操作会阻塞直到数据被成功写入，而 InputStream 上的读操作也会阻塞，直到有数据可读。

对于接受连接的服务器，Java 提供了表示服务器 socket 的 ServerSocket 类。ServerSocket 运行于服务器上，监听入站的 TCP 连接。socket 服务器监听服务器机器的某个端口，当有客户端尝试连接此端口时，服务器被唤醒，经过协商后，返回一个表示两台主机间 socket 的 java.net.Socket 对象。综上所述，服务器程序的基本生命周期如下：

1）使用一个 ServerSocket 构造函数构造对象；

2）将 ServerSocket 对象绑定（bind）在某一端口上；

3）使用 accept() 方法监听此端口的入站连接，accept() 会一直阻塞，直到客户端尝试进行连接，这时 accept() 将返回一个 java.net.Socket 对象；

4）服务器使用 Socket 对象和客户端进行交互，直到关闭连接；

5）服务器或客户端关闭连接；

6）服务器返回步骤 3（accept() 方法处），等待下一次连接。

其中，服务器上使用 java.net.Socket 与客户端交互，也是通过 getOutputStream() 和 getInputStream() 得到的相应的流对象进行的；关闭连接则通过 close() 方法，这些和客户端上使用 Socket 实例是一致的。主要差别在于 ServerSocket 的 bind() 方法和 accept() 方法，它们的定义如下。其中，accept() 方法没有提供异步实现，对它的调用会一直阻塞，直到有客户端连接。

```
public class ServerSocket{
    ……
    // 构造函数
    public ServerSocket()
    ……
    // 绑定 ServerSocket 到特定端口
```

```
    public void bind(SocketAddress endpoint)
    ……
    //接受连接
    public Socket accept()
    ……
}
```

图 4-8 给出了客户端、服务器各个方法间的配合。

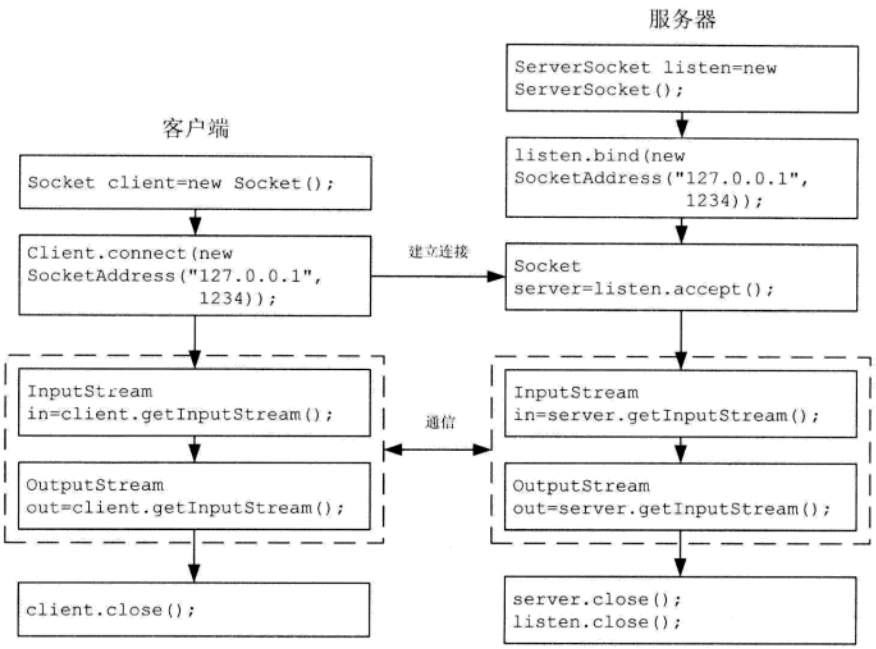



图 4-8 Java 基本套接字 API 的配合

通过 java.net.Socket 和 java.net.ServerSocket 可以实现较简单的、小规模的网络应用。

### 4.3.2 Java NIO 基础

当我们需要实现同时处理上千个客户请求的服务器时，Java 基本套接字会产生一些问题：由于 OutputStream 的 write() 方法、InputStream 的 read() 方法和 ServerSocket 的 accept() 方法[⊖]都是阻塞方法，往往需要采用一个客户对应一个线程的服务器系统，虽然使用线程池在某种程度上能够节省部分系统开销，但对于生存期很长的协议来说，大量的闲置客户端限制了系统可以同时服务的客户端总数。为了解决这样的问题，JDK 1.4 中引入了 Java 新输入/输出系统（New Input/Output，NIO）。

⊖ 其实可以通过 ServerSocket 的 setSoTimeout() 方法为 accept() 操作设置超时，如果调用 accept() 时，有一个新连接请求在等待，accept() 将立即返回，否则，accept() 将等待到有新的请求到来，或者是计时器超时。

非阻塞是 NIO 实现的重要功能之一，为了实现非阻塞，NIO 引入了选择器（Selector）和通道（Channel）的概念。通道表示到实体，如硬件设备、文件、网络套接字或可以执行一个或多个不同 I/O 操作（如读取或写入）的程序组件的开放连接。一些通道，如文件、网络套接字，允许选择器对通道进行轮询。也就是说，通道能够注册一个选择器实例，通过该实例的 select() 方法，用户可以询问“在（一个或一组）通道中，哪一个是当前需要的服务（即被读、写或接受）”。在一个准备好的通道上进行相应的 I/O 操作，就不需要等待，也就不会阻塞了。

### 1. 缓冲区

NIO 中一个主要的特性是 java.nio.Buffer。缓冲区（Buffer）提供了一个比流抽象的、更高效和可预测的 I/O。Buffer 代表了一个有限容量的容器——其本质是一个数组，通道 Channel 使用 Buffer 实例来传递数据。

从编程的角度看，流和通道的差别在于流是基于字节的，而通道是基于块的（也就是 Buffer）。流设计为一个字节接着一个字节、按顺序提供数据，虽然基于性能考虑，流也可以传输字节数组，但最基本的流概念要求一次传送一个字节。通道不同，通道使用的是缓冲区中的数据块，即 Buffer，在读写通道数据时，这些数据必须存储在缓冲区中，一次读写一个缓冲区的数据。

基本上 Buffer 是一个数组，但是在读写数据时，Buffer 具有一些内部状态，用来跟踪缓冲区的当前位置。每个 Buffer 都包含下面 4 个索引：

- ❑capacity：缓冲区的元素总数（容量），通过 Buffer.capacity() 可以获取该索引。缓冲区元素总数是不可修改的。
- ❑position：缓冲区的位置，是下一个要读取或写入的元素的索引，该位置可以由 position() 方法和 position(int) 方法获取和设置。
- ❑limit：缓冲区的限制，即第一个不应该读取或写入的元素的索引，limit() 方法和 limit(int) 方法可用于获取和设置缓冲区的限制。
- ❑mark：缓冲区的位置设置标记，通过 mark() 方法设置一个位置，然后利用 reset() 方法可以将 position 重置为 mark() 方法设置的位置。

Buffer 的上述 4 个索引遵守以下不变式：

$$0 \leqslant \text{mark} \leqslant \text{position} \leqslant \text{limit} \leqslant \text{capacity}$$

缓冲区限制必须小于或等于缓冲区的容量，同时，缓冲区的位置也必须小于或等于缓冲区的限制，它们之间的距离就是这个缓冲区可以读取 / 写入的元素数（可通过 Buffer.remaining() 方法获得该值），mark 是缓冲区在某一个时刻设置的位置标记，它必然要小于或等于缓冲区位置。

在了解了缓冲区索引以后，以 Hadoop IPC 中用到的 ByteBuffer 为例，介绍缓冲区的一些典型 API。ByteBuffer 对象的创建一般用工厂方法：

```
public static ByteBuffer allocate(int capacity)
```

```
public static ByteBuffer wrap(byte[] array)
public static ByteBuffer wrap(byte[] array, int offset, int length)
```

其中，allocate() 用于直接创建缓冲区，而 wrap() 是在某个字节数组上创建缓冲区，该数组也叫底层实现数组，对该数组的数据进行修改，都将改变缓冲区中的数据。

创建缓冲区时，缓冲区索引会被初始化，其中，allocate() 创建的缓冲区，position 等于 0，而 limit 则等于 capacity；通过 wrap() 的第二种形式创建的缓冲区，position 等于输入的 offset，而 limit 则是 offset+length 的结果；wrap() 第一种形式相当于调用了 wrap(array,0,array.length)，即 position 的值为 0，缓冲区的限制 limit 等于字节数组的长度。

有了缓冲区，就可以用它来存放数据了。作为数据的“容器”，缓冲区可以用于输入、也可以用于输出，这一点就和流不一样了，流只能向一个方向传递数据。使用 put() 方法可以将数据放入缓冲区，使用 get() 方法则可以从缓冲区中获取数据。

ByteBuffer 提供两种类型的 put()/get() 方法：基于相对位置和基于绝对位置。

基于相对位置就是基于目前缓冲区位置 position 的当前值，从“下一个”位置读取或存放数据，并为 position 增加适当的值。也就是说，每次调用 put() 方法，都是在 position 后面追加数据（并移动 position 的位置），而每次调用 get() 方法，都是在读 position 后面的数据（并移动 position 的位置）。需要注意的是，部分数据的 get()/put() 是不允许的。以写入为例，如果要写入的数据量超过当前缓冲区允许写入的数据量（可通过 Buffer.remaining() 方法获得该值），则所有的数据都不会写入缓冲区，position 的位置不变，put() 方法抛出 BufferOverflowException 异常。绝对位置的 put()/get() 方法，必须提供写入 / 读出的位置，该方式的读写操作，不改变 position 的值。相关代码如下：

```
// 相对位置
public byte get()
public ByteBuffer get(byte[] dst)
public ByteBuffer get(byte[] dst, int offset, int length)
public ByteBuffer put(byte b)
public final ByteBuffer put(byte[] src)
public ByteBuffer put(byte[] src, int offset, int length)
public ByteBuffer put(ByteBuffer src)

// 绝对位置
public byte get(int index)
public ByteBuffer put(int index, byte b)
```

在使用缓冲区进行数据输入 / 输出时，必须确认缓冲区的 position、limit 处于正确的位置，java.nio.Buffer 提供了一些工具方法，帮助用户设定 position、limit 的位置。

假设某缓冲区目前处于某状态，这个时候 position 不等于 0，limit 也不等于 capacity；如果现在想将数据读入 / 写入到这个缓冲区中，首先要先对该缓冲区进行清空，通过 clear() 方法，缓冲区的 poistion 被设置为 0，limit 设置为 capacity，这样，缓冲区准备好接收新数据。后续的 put()/read() 调用，将数据从第一个元素开始填入缓冲区，最多直到填满该缓冲区，达到 limit 位置（等于 capacity）。

flip() 方法用于将缓冲区准备为数据传出状态，该方法将 limit 设置为 position 后，将 position 设置为 0。后续的 get()/write() 方法将从缓冲区的第一个元素开始传出数据，直到 limit 位置。通过 flip() 方法和 get()/write() 方法配合，可以将前面利用 put()/read() 方法放入缓冲区的所有数据读出（一直读到 limit）。

rewind() 方法将 position 设置为 0，但不改变 limit 的值，如果需要多次读取缓冲区里的数据，可以在两次读取间使用 rewind() 方法。

clear()、flip()、rewind() 和 compact() 方法对索引 position、limt 的影响如图 4-9 所示。

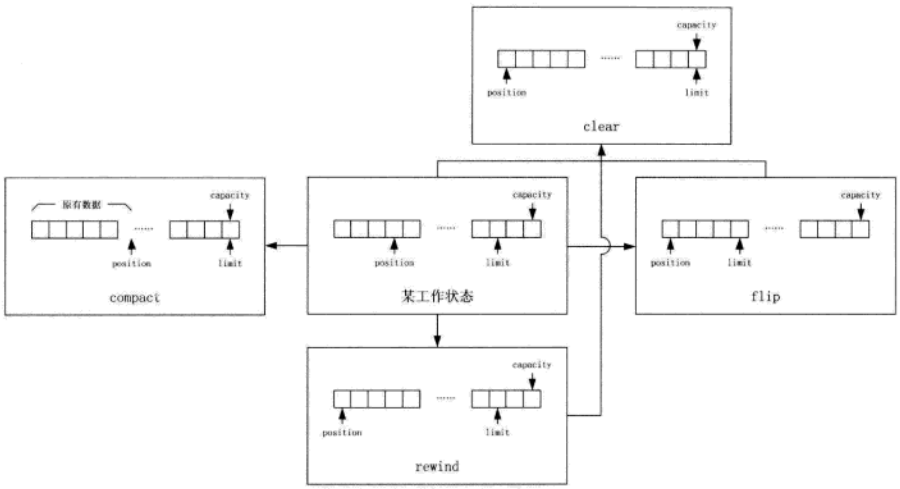



图 4-9　clear()、flip()、rewind() 和 compact() 方法对索引 position、limt 的影响

compact() 方法将 position 和 limit 间的数据复制到缓冲区的开始位置，为后续的 put()/read() 调用让出空间。调用结束后，poistion 的值被设置为数据的长度，也就是原来的 limit 减去 position 的值，而 limit 则设置为 capacity。和 clear()、flip() 等方法不同，compact() 不但改变了 position 和 limit 的位置，还改变了缓冲区中的数据。

compact() 主要用于在缓冲区中还有未写出的数据时，为读入数据准备空间：即在 write() 方法调用后和添加新数据的 read() 方法前调用 compact() 方法，将未写出的“剩余”数据移动到缓冲区前面，为后面 read() 方法提供释放空间。

在图 4-9 中，假设 write() 方法调用后，缓冲区处于“某工作状态”，这时，position 到 limit 间的数据为未写出的“剩余”数据，而 limit 到 capacity 的空间则是 read() 方法可以使用的空间，通过 compact() 操作，position 到 limit 间的数据被挪到缓冲区的前面，position 的位置也被设置为“剩余”数据长度。接下来，开发人员就可以直接调用 read()/put() 方法，从 position 位置开始放入数据。该数据和原来的“剩余”数据一起，构成了连续的可用数据。

Buffer 还支持一些其他功能，如直接缓冲区（direct buffer）、Java 基本类型的 put()/get()、缓冲区共享、复制、透视、字符编码转换等，由于在 Hadoop IPC 上没有用到这些功

能，这里就不再进行分析了，有兴趣的读者可以参考相关文档。

**2. 通道**

通道（Channel）是 NIO 中的另一个重要概念，如前文所述，Channel 实例代表一个和设备的连接，通过它可以进行输入输出操作。

Channel 的思想和前面介绍过的普通套接字非常类似，对于 TCP 协议来说，有 ServerSocketChannel 和 SocketChannel 对应于前面介绍的 ServerSocket 和 Socket。针对其他设备还有其他类型的 Channel，如文件设备 FileChannel。下面通过 API 来了解 ServerSocketChannel 和 SocketChannel 与传统 Socket 的 4 个区别。

（1）对象的创建

ServerSocketChannel 和 SocketChannel 通过工厂方法创建，相关代码如下：

```
public static SocketChannel open()
public static SocketChannel open(SocketAddress remote)
public static ServerSocketChannel open()
```

SocketChannel 创建后，可以通过 connect() 连接到远程机器，通过 close() 关闭连接，这些操作和 Socket 的没有什么差别。

（2）数据的读写

和通过 Socket 获取输入 / 输出流进行读写不同，SocketChannel 读写数据时，使用前面介绍过的 Buffer 作为参数。下面是 SocketChannel 读写数据的 read()/write() 方法声明，最基本的读写操作都需要一个 ByteBuffer 作为参数：

```
public int read(ByteBuffer dst)
public long read(ByteBuffer[] dsts, int offset, int length)
public final long read(ByteBuffer[] dsts)
public int write(ByteBuffer src)
public long write(ByteBuffer[] srcs, int offset, int length)
public final long write(ByteBuffer[] srcs)
```

ServerSocketChannel 也提供了和 ServerSocket 类似的 accept()、close() 等方法，与 ServeSocket 有细微差别的地方是：accept() 方法返回 SocketChannel 对象，而不是 Socket 对象；另外，ServerSocketChannel 不提供 bind() 方法，绑定 Socket 到某端口需要使用底层的 Socket 的 bind() 方法，即 ServerSocket.socket().bind(……)。相关代码如下：

```
public SocketChannel accept()
public ServerSocket socket()
```

（3）是否能工作在非阻塞状态

相对普通套接字，结合 Buffer 使用阻塞式 Channel 几乎没有什么优点。一般情况下，需要把 Channel 设置成非阻塞式，方法如下。

```
public SelectableChannel configureBlocking(boolean block)
public boolean isBlocking()
```

非阻塞式 SocketChannel 和阻塞式 Socket 有一些不同，如非阻塞的 SocketChannel.

connect() 方法会立即返回，用户必须通过 isConnected() 判断连接是否已经建立，或者通过 finishConnect() 方法在非阻塞套接字上阻塞等待连接成功；非阻塞的 read()，在 Socket 上没有数据的时候，立即返回（返回值为 0），不会等待；非阻塞的 accept()，如果没有等待的连接，将返回 null。相关方法定义如下：

```
public boolean isConnected()
public boolean finishConnect()
```

（4）和选择器配合工作

ServiceSocketChannel/SocketChannel 和基本套接字的最后一个差别是它们能够和选择器（Selector）配合工作，避免非阻塞式 I/O 操作中很浪费资源的“忙等”方法。如在 HDFS 中，同时有上千个 DataNode 连接到 NameNode 服务器，但在任何时刻，服务器都只有少量的（甚至可能没有）请求需要处理，这就需要一种方法阻塞等待，直到至少一个 Channel 可以进行 I/O 操作，并指出是哪个通道。NIO 的选择器就是为这个功能设计的，一个 Selector 实例可以同时检查（也可以等待）一组通道的 I/O 状态。

### 3. 选择器

选择器（Selector）的使用方法：通过静态的工厂方法创建 Selector 实例，通过 Channel 的注册方法，将 Selector 实例注册到想要监控的 Channel 实例上，最后调用选择器的 select() 方法。该方法会阻塞等待，直到有一个或多个通道准备好 I/O 操作或超时。select() 方法将返回可进行 I/O 操作的通道数量。现在，在一个单独的线程中，就可以检查多个通道是否可以进行 I/O 操作，不需要为每一个通道都准备一个线程了。

可以通过如下方法打开和关闭 Selector：

```
public static Selector open()
public boolean isOpen()
public abstract void close()
```

获得 Selector 实例后，就可以将选择器和通道进行关联，这个关联是多对多的关联，关联的信息保存在 java.nio.channels.SelectionKey 实例中。选择器注册标记 SelectionKey 维护了一个通道上感兴趣的事件类型信息，包括 4 种类型：

- OP_READ（通道上有数据可读）
- OP_WRITE（通道已经可写）
- OP_CONNECT（通道连接已建立）
- OP_ACCEPT（通道上有连接请求）

这些类型都是 SelectionKey 类中定义的常量，并通过位图（bitmap）保存在 SelectionKey 实例中，通过 interestOps() 方法获取 / 设置这些标志位，相关代码如下：

```
public int interestOps()
public SelectionKey interestOps(int ops)
```

ServerSocketChannel/SocketChannel 和基本套接字与 Selector 配合工作相关的 API 如下：

```
public SelectionKey register(Selector sel, int ops)  // 注册
// 带附件注册
public SelectionKey register(Selector sel, int ops, Object att)
public SelectionKey keyFor(Selector sel)  // 根据选择器，查找对应的 SelectionKey
publicboo lean isRegistered()  // 判断通道是否已经注册
```

调用通道的 register() 方法，可以将一个选择器注册到该通道，注册过程需要提供位图，指定该通道上的初始兴趣事件集，三参数的 register() 方法可以为选择器注册标记提供附件 att，附件可以为后续通道上的 I/O 操作提供附加参数。register() 方法返回代表通道和给定选择器之间关联的 Selector 注册标记实例，下面代码（从 Hadoop IPC 服务器代码中摘出）在选择器实例 selector 上注册了一个 SocketChannel 对象，支持读操作：

```
SelectionKey readKey = channel.register(selector, SelectionKey.OP_READ)
```

SelectionKey 关联的选择器实例和 Channel 实例可以通过键上的 selector() 方法和 channel() 方法获取。SelectionKey.cancel() 方法用于注销该键。注销的键将从选择器中移除，其关联的通道也将不再监听（除非它又重新注册），相关代码如下：

```
Selector selector()
SelectableChannel channel()
void cancel()
```

一般来说，选择器上会注册多个通道，这些通道通过关联的 SelectionKey 指定它们感兴趣的 I/O 操作集，接下来可以开始等待通道 I/O 事件了，这一般通过选择器来完成。

选择器提供了三种形式的 select() 方法，其返回值都是整型，表明已注册通道上发生了感兴趣 I/O 事件的个数：

```
public int select()// 阻塞等待，直到一个注册通道有感兴趣的操作就绪
public int select(long timeout)// 等待一段时间，或一个注册通道有感兴趣的操作就绪
public int selectNow()// 非阻塞版本
```

如果发现 select() 的返回值大于 0，表明有需要处理的 I/O 事件发生，选择器会将对应的通道放入已选键集（Selected-key set）中，通过 Selector.selectedKeys() 可以获得该集合，并在上面迭代处理关联通道上的 I/O 操作。API 如下：

```
public Set<SelectionKey> keys()//Selector 上已注册的所有键
public Set<SelectionKey> selectedKeys()// 已选键集
```

以下片段是 Hadoop IPC 服务器代码的一部分，是应用 NIO 选择器的典型实现，代码如下：

```
Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
while (iter.hasNext()) {
   key = iter.next();
   iter.remove();
   ……
   if (key.isAcceptable())doAccept(key);
   else if (key.isReadable()) doRead(key);
   ……
}
```

以上代码通过 selectedKeys() 方法获取可进行 I/O 操作的通道，通过遍历通道，利用 SelectionKey 判断通道上等待的操作（如 isAcceptable() 表明有一个入站请求），并作相应的处理。selectedKeys() 方法返回的键集是可修改的，实际上，两次调用 select() 之间，必须“手工”将其清空（代码片段中的 iter.remove() 调用）。SelectionKey 上有如下判断通道上等待的操作方法：

```
public int readyOps()
public boolean isReadable()
public boolean isWritable()
public boolean isConnectable()
public boolean isAcceptable()
```

SelectionKey 还提供了一个非常有用的功能：通道附件。

一般来说，通道准备好 I/O 操作时，通常还需要额外的信息来处理请求，如在 Hadoop IPC 的处理过程中，需要保持部分已经接收的数据，直到收到一个完整的 IPC 请求。这些额外的信息构成了通道的状态，通道附件，使得我们很容易保持这些信息。

通过通道附件可以将处理通道上请求的一些上下文信息，作为附件保存在 SelectionKey 对象里。SelectionKey 中与通道附件相关的方法有三个，除了前面介绍的三参数 register() 方法，还有下面两个操作：

```
public Object attach(Object ob)
public Object attachment()
```

### 4.3.3　Java NIO 实例：回显服务器

Java NIO 包括缓冲区（Buffer）、通道（Channel）和选择器（Selector）三大类抽象，涉及的内容较多，下面以最简单的回显服务器为例（参考 org.hadoopinternal.nio），介绍这些对象间如何相互配合，异步完成 I/O 操作。

回显服务器简单地将 Socket 上接收的数据发送回客户端，用户可以通过下面的命令，体验回显服务器（黑体字是输入，Ctrl+] 用于退出）：

```
C:\Documents and Settings\Administrator>telnet 127.0.0.1 12112
aabbccaaCtrl+]
欢迎使用 Microsoft Telnet Client
Microsoft Telnet>quit
Escape 字符是 'CTRL+]'
```

例子包括 NIOServer 和 NIOServerConnection 两个类。NIOServer 只包含 main() 方法，使用选择器在唯一的线程中处理各种 I/O 请求；NIOServerConnection 代表回显服务器客户端与服务器间的连接，保存了一些和连接相关的状态。

NIOServer.main() 的开始部分创建了一个工作于异步模式的 ServerSocketChannel 对象，并注册到选择器 selector 中，该 ServerSocketChannel 对象感兴趣的操作是通道上有连接请求，即 SelectionKey.OP_ACCEPT。相关代码如下：

```
//打开一个选择器
Selector selector = Selector.open();

//打开一个 ServerSocketChannel
ServerSocketChannel listenChannel = ServerSocketChannel.open();

listenChannel.configureBlocking(false);     //配置为异步模式

//绑定到 TCP 端口上，注意，ServerSocketChannel 不提供 bind 方法，
//需要使用 ServerSocketChannel 内部的 socket 对象对应的 bind 方法
listenChannel.socket().bind(new InetSocketAddress(PORT));

listenChannel.register(selector, SelectionKey.OP_ACCEPT);    //注册
```

接下来进入服务器循环，调用 Selector.select() 方法等待 I/O 事件。如果 select() 方法的返回值是 0，表明没有事件发生。代码如下：

```
while(true) {
   if(selector.select(TIMEOUT)==0) {
       System.out.print(".");
       continue;
   }

   //获取已选键集
   Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
   while( iter.hasNext() ) {//对于已选键集中的每一个键
       SelectionKey key = iter.next();
       iter.remove();//将已选键从键集中移除
       ......
   }
}
```

如果 select() 方法返回值大于 0，表明有 I/O 事件等待处理，则获取已选键集并对已选键集中的每一个键进行处理。如果事件是“通道上有连接请求”，相应的处理是通过 accept() 操作获得 SocketChannel 对象，并配置该对象为异步工作模式、注册到选择器中（注册事件为通道可读 SelectionKey.OP_READ），根据注册得到的 SelectionKey 对象构造连接对象，并将对象作为 SelectionKey 对象的附件，关联到 SelectionKey 对象上。如果事件是“通道上有数据可读”或“通道已经可写”，那么获取通道附件并调用相应的方法处理事件。代码如下：

```
if( key.isAcceptable() ) {
   SocketChannel channel=listenChannel.accept();
   channel.configureBlocking(false);//配置为异步模式
   SelectionKey connkey=channel.register(selector,//注册
                                         SelectionKey.OP_READ );
   //构造连接对象
   NIOServerConnection conn=new NIOServerConnection(connkey);
   connkey.attach(conn);//连接对象作为 SelectionKey 对象的附件
}
```

```
if( key.isReadable() ) {
   NIOServerConnection conn=(NIOServerConnection) key.attachment();
   conn.handleRead(); // 处理通道上有数据可读
}

// 调用 isValid(), 保证通道有效（即未关闭）
if( key.isValid() && key.isWritable() ) {
   NIOServerConnection conn=(NIOServerConnection) key.attachment();
   conn.handleWrite(); // 回显，往客户端写数据
}
```

总的来说，使用 Selector 的步骤如下：

1）创建一个 Selector 实例；

2）将该实例注册到各种通道，指定每个通道上感兴趣的 I/O 操作；

3）重复执行（选择器循环）：

a）调用一种 select() 方法；

b）获取已选键集；

c）对于已选键集中的每一个键：

i. 将已选键从键集中移除；

ii. 获取信道，并从键中获取附件（如果需要）；

iii. 确定准备就绪的操作并执行；对于 accept 操作获得的 SocketChannel 对象，需将信道设置为非阻塞模式，并将其注册到选择器中；

iv. 根据需要，修改键的兴趣操作集。

NIOServerConnection 是一个使用缓冲区的典型，主要代码在方法 handleRead() 和 handleWrite() 中。

NIOServerConnection.handleRead() 通过 SocketChannel 的 read() 方法读取数据，当 read() 方法返回 -1 时，表明对方已经关闭 Socket，相应地，服务器端也要关闭 SocketChannel；否则，设置感兴趣的 I/O 事件为读 / 写，即 SelectionKey.OP_READ 和 SelectionKey.OP_WRITE。

handleWrite() 方法在 SocketChannel 可写（Socket 为什么会不可写，请参考相关的书籍）的情况下将数据写回客户端。主要步骤：通过 flip() 方法将缓冲区由读模式转为写模式，然后调用 SocketChannel 的 write() 方法写数据。如果缓冲区中已经没有数据，则设置感兴趣的 I/O 事件为读：SelectionKey.OP_READ，最后调用 ByteBuffer.compact() 方法，一方面将缓冲区由写模式转为读模式，另一方面为读入数据保留更多的空间。相关代码如下：

```
public void handleRead() throws IOException {
   long bytesRead=channel.read(buffer);

   if(bytesRead==-1) {    // 可以关闭 socket
       channel.close();
   } else {
       key.interestOps( SelectionKey.OP_READ | SelectionKey.OP_WRITE );
```

```
        }
    }

    public void handleWrite() throws IOException {
        buffer.flip();
        channel.write(buffer);

        if(!buffer.hasRemaining()) {  //缓冲区中已经没有数据
            key.interestOps( SelectionKey.OP_READ );
        }

        buffer.compact();
    }
```

读者可以体会一下 handleRead() 和 handleWrite() 中对缓存区的使用，特别值得注意的是，在连续多次调用 handleRead()，或有足够的数据可写情况下连续多次调用 handleRead() 方法时，缓冲区工作状态的变化。还需要注意的是，缓冲区不是线程安全的，如果在多线程的环境中，需要根据实际情况做一些保护。不过，本例中使用了一个线程，而且只使用了一个线程，就完成了 Java 基本套接字中需要多线程完成的任务。

## 4.4 Hadoop 中的远程过程调用

Hadoop 没有使用前面介绍的 Java RMI，而是实现了一套自己独有的节点间通信机制，理由和 Hadoop 使用 Writable 形式的序列化机制类似，有效的 IPC（Inter-Process Communication，进程间通信）对于 Hadoop 来说是至关重要的，Hadoop 需要精确控制进程间通信中比如连接、超时、缓存等通信细节，显然，Java RMI 达不到这些需求，Hadoop 进程间通信机制包括上一章分析过的，结合数据输出流（DataOutputStream）和数据输入流（DataInputStream）的 Writable 序列化机制，以及下面将要介绍的，一个简洁的、低消耗的远程过程调用机制。

### 4.4.1 利用 Hadoop IPC 构建简单的分布式系统

在分析 Hadoop 远程过程调用实现之前，利用 Hadoop IPC 机制构建一个简单的分布式系统。该系统的源代码在包 org.hadoopinternal.ipc 中，类图如图 4-10 所示。

细心的读者会发现，这个类图和 Java RMI 实例的类图（图 4-4）非常类似，IPCQueryStatus 接口定义了 IPC 服务器对外提供的功能，它使用 IPCFileStatus 作为 getFileStatus() 方法的返回结果；IPCQueryStatusImpl 实现了接口中定义的功能，IPCQueryServer 利用 RMIQueryStatusImpl 的实例，构造并运行 IPC 服务器；IPCQueryStatusClient 作为客户端，使用 IPC 访问服务器提供的 RMIQueryStatus 功能。

Java RMI 的开发从远程接口的定义开始，远程接口必须继承自 java.rmi.Remote；在 Hadoop 远程过程调用中，也是通过一个 IPC 接口开始进行开发。Hadoop IPC 接口必须继承

自 org.apache.hadoop.ipc.VersionedProtocol 接口，代码如下：

```
public interface VersionedProtocol {
   //返回协议接口的版本
   //输入参数是协议接口的类名和客户端的版本号，输出是服务器的协议接口版本号
   public long getProtocolVersion(String protocol,
                                  long clientVersion) throws IOException;
}
```

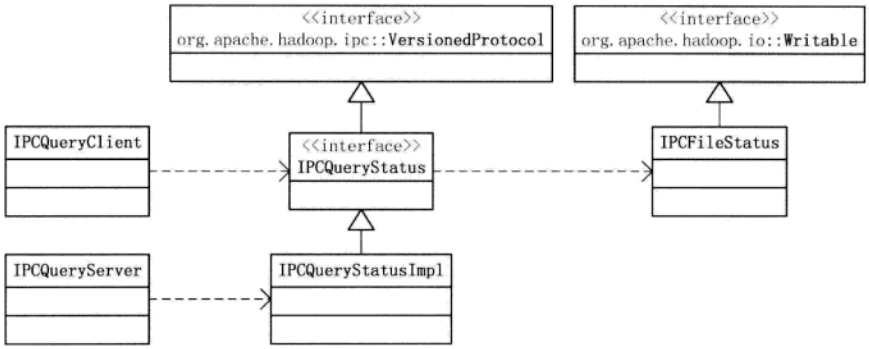



图 4-10　org.hadoopinternal.ipc 的类图

和 java.rmi.Remote 不同，VersionedProtocol 不是一个声明性接口，它有一个 getProtocolVersion() 方法，实现该接口的类都必须实现这个方法。getProtocolVersion() 有两个参数，分别是协议接口对应的接口名称 protocol 和客户端期望的服务的版本号 clientVersion，方法返回服务器端的接口实现的版本号。在建立 IPC 时，getProtocolVersion() 用于检查通信的双方，保证它们使用了相同版本的接口。

IPCQueryStatus 接口定义如下：

```
//IPC 接口必须继承自 VersionedProtocol
public interface IPCQueryStatus extends VersionedProtocol {
   IPCFileStatus getFileStatus(String filename);
}
```

IPCQueryStatus 接口的功能和 RMIQueryStatus 一样，都模拟了文件系统中对文件元信息的查询功能，IPCQueryStatus 中的唯一方法 getFileStatus() 接受一个字符串参数 filename，返回一个 IPCFileStatus 对象。其中，IPCFileStatus 对象会从 IPC 服务器传输到客户端，它必须实现接口 Writable。在这一点上，Hadoop IPC 和 Java RMI 也比较类似，都要求需要在网络上传输的对象实现某种序列化机制。

IPCQueryStatus 的一个实现是 IPCQueryStatusImpl（下面的讨论中，实现 IPC 接口的对象称为 IPC 接口实现类），它的代码非常简单。getFileStatus() 根据输入参数构造一个 IPCFileStatus 对象，然后返回该对象。大家可以发现，和 Java RMI 类似，Hadoop 中 IPC 接口的实现中的方法与普通方法没有什么差别。代码如下：

```
public class IPCQueryStatusImpl implements IPCQueryStatus {
    ……
    @Override
    public IPCFileStatus getFileStatus(String filename) {
        IPCFileStatus status=new IPCFileStatus(filename);
        ……
        return status;
    }

    @Override
    public long getProtocolVersion(String protocol, long clientVersion)
            throws IOException {
        System.out.println("protocol: "+protocol);
        System.out.println("clientVersion: "+clientVersion);
        return IPCQueryServer.IPC_VER;
    }
}
```

前面已经介绍过了，getProtocolVersion() 用于建立 IPC 时的版本检查，在 IPCQueryStatusImpl 中，只需简单返回一个代表接口版本的常量。在一些比较复杂的实现中，如 org.apache.hadoop.hdfs.server.namenode.NameNode 中，由于 NameNode 类实现了多个 IPC 接口，它的 getProtocolVersion 实现需要根据 IPC 客户端发过来的接口名称 protocol，确定并返回相应的服务器端版本号。

下面以实现了 IPC 接口的 IPCQueryStatusImpl 类为基础，开始搭建 IPC 服务器。和 Java RMI 相比，Hadoop IPC 建立服务器的方法非常简单，通过 org.hadoopinternal.ipc.IPC 的静态方法 getServer()，就可以在一个 IPCQueryStatusImpl 实例上，获得一个提供 IPCQueryStatus 接口功能的 IPC 服务器。一旦获得服务器实例，就可以通过它的 start() 方法和 stop() 方法启动和停止服务器。具体代码如下：

```
public class IPCQueryServer {
    public static final int IPC_PORT = 32121;
    public static final long IPC_VER = 5473L;

    public static void main(String[] args) {
        try {
            IPCQueryStatusImpl queryService=new IPCQueryStatusImpl();
            // 在对象 queryService 上获得一个 IPC 服务器
            Server server = RPC.getServer(queryService,
                    "0.0.0.0",IPC_PORT,new Configuration());

            server.start();// 服务器开始服务
            ……
            server.stop();// 停止服务器
            ……
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
    }
}
```

RPC.getServer() 一共需要 4 个参数，它们分别是 IPC 接口的实现对象、监听地址、端口和 Hadoop 配置类 Configuration 的实例。其中，实现对象最少实现一个 IPC 接口，实现对象也可以实现多个 IPC 接口，如 Hadoop 的 NameNode（类 org.apache.hadoop.hdfs.server.namenode.NameNode）就实现了多个 IPC 接口，客户端可以访问这些接口中的任意一个。监听地址（在上面的例子中是“0.0.0.0”，表示在所有的地址上监听）和端口（常量 IPCQueryServer. IPC_PORT）一起，形成了 IPC 服务器的 Socket 地址。配置类 Configuration 的实例可用于设置一些 IPC 服务器的参数，不过在这个例子中，只使用了 IPC 服务器的默认配置。

访问 IPC 服务器的客户端代码如下。

```
public class IPCQueryClient {
   public static void main(String[] args) {
      try {
         // 为 RPC.getProxy 准备 Socket 地址
         InetSocketAddress addr=new InetSocketAddress("localhost",
               IPCQueryServer.IPC_PORT);

         // 通过创建 IPC 客户端，获得接口 IPCQueryStatus 的一个实例
         IPCQueryStatus query=(IPCQueryStatus)
               RPC.getProxy(IPCQueryStatus.class,// 接口的类对象
                     IPCQueryServer.IPC_VER,// 接口版本
                     addr,// 服务器地址
                     new Configuration());

         // 调用远程方法：该调用如同调用本地方法
         IPCFileStatus status=query.getFileStatus("/tmp/testIPC");
         System.out.println(status);

         // 停止客户端
         RPC.stopProxy(query);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

RPC.getProxy() 用于获得一个 IPC 客户端接口实例，它需要 4 个参数：

- IPC 接口的类对象：可以通过 IPC 接口的静态成员 class 直接获得。接口的静态成员 class 保存了该接口的 java.lang.Class 实例，java.lang.Class 类的实例表示正在运行的 Java 应用程序中的类和接口，提供了一系列与 Java 反射相关的重要功能。
- 接口版本：在 Hadoop 这样复杂的系统中，接口会根据需求不断地进行升级，形成多个版本的 IPC 接口。如果 IPC 客户端和服务器使用同一个 IPC 接口的不同版本进行通信，结果将是灾难性的，所以在建立 IPC 时，需要对 IPC 的双方进行版本检查。

❑服务器的 Socket 地址：用于建立 IPC 的底层 TCP 连接。

❑配置类 Configuration 对象：用于定制 IPC 客户端参数。

客户端通过 RPC.getProxy() 就可以获得一个 IPC 客户端对象，通过这个客户端对象进行远程过程调用，调用会转发到服务器端执行，但调用者在使用这个客户端对象时，不会感到它和普通对象有什么差别。例如，例子中对 IPCQueryStatus 的实例 query 调用 getFileStatus() 方法，将会发送到 IPC 服务器上执行，但用户在使用 query 时，不需要区别它是一个远程过程调用的客户端对象，还是一个普通对象。可以说，Hadoop IPC 实现了远程过程调用和本地调用的语义尽可能地类似的基本要求。

在使用这个例子的时候可以发现，IPCQueryStatusImpl.getProtocolVersion() 方法会在 RPC.getProxy 返回前被调用。也就是说，在创建客户端对象的时候，客户端存根会利用 RPC.getProxy() 方法的第二个参数（接口版本），检查服务器端返回的版本信息，如果版本不匹配，会抛出下面的异常：

```
org.apache.hadoop.ipc.RPC$VersionMismatch: Protocol org.hadoopinternal.ipc.
  IPCQueryStatus version mismatch. (client = 5474, server = 5473)
  at org.apache.hadoop.ipc.RPC.getProxy(RPC.java:364)
  at org.apache.hadoop.ipc.RPC.getProxy(RPC.java:346)
  at org.apache.hadoop.ipc.RPC.getProxy(RPC.java:383)
  at org.hadoopinternal.ipc.IPCQueryClient.main(IPCQueryClient.java:12)
```

## 4.4.2 Hadoop IPC 的代码结构

Hadoop 中与 IPC 相关的代码都在 org.apache.hadoop.ipc 包中，共 7 个文件，包括远程过程调用实现需要的辅助类：

❑RemoteException：远程异常，应用于 IPC 客户端，表示远程过程调用中的错误。

❑Status.java：Status 是一个枚举类，定义了远程过程调用的返回结果，包括 SUCCESS（成功）、ERROR（一般错误）和 FATAL（致命错误）三种情况。

❑VersionedProtocol 接口：前面已经介绍过，Hadoop IPC 的远程接口都扩展自 VersionedProtocol。

❑ConnectionHeader.java：IPC 客户端与服务器建立连接时发送的消息头。

与客户端、服务器实现相关的代码主要在 Client.java、Server.java 和 RPC.java 三个文件中。

1）Client.java 包含了与 IPC 客户端相关的代码，如图 4-11 所示。

Client 是对 IPC 客户的抽象，它包含的内部类可以分为与 IPC 连接相关的，如 Client.Connection 和 Client.ConnectionId 等；和远程调用 Call 相关的如 Client.Call、Client.ParallelCall 等（后面会详细介绍 IPC 连接、远程调用 Call 的相关概念）。

2）Server.java 实现了一个 IPC 服务器端的抽象类。如图 4-12 所示。

和 Client.java 类似，Server.java 也有对应的 IPC 连接、远程调用 Call 相关的类，即 Server.Connection 与 Server.Call。Server.java 中，对远程调用 Call 的处理，由 Listener、Handler 和 Responder 配合完成，它们都继承自 java.lang.Thread，在各自的线程中运行。

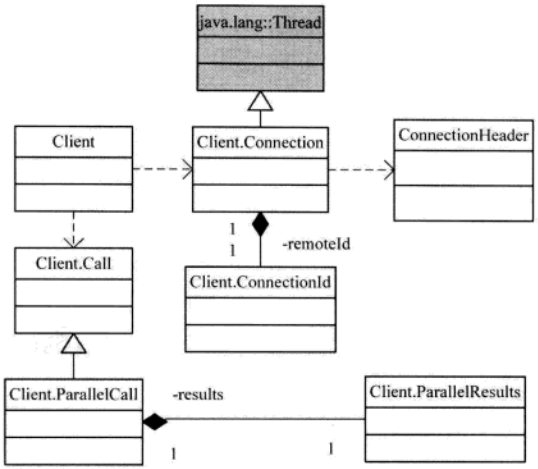



图 4-11　Client.java 中包含的主要类

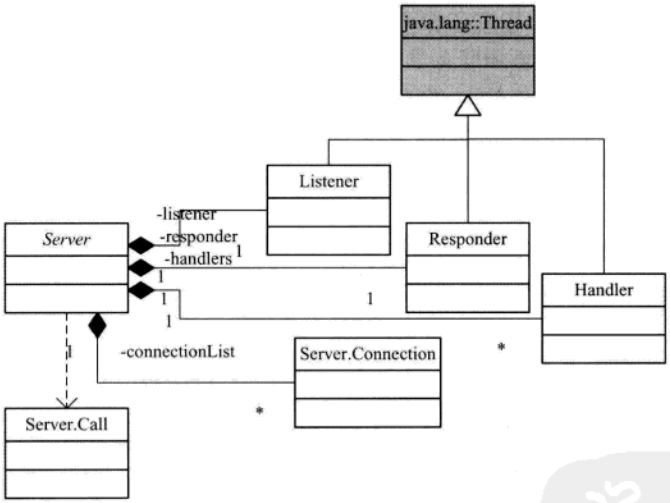



图 4-12　Server.java 的类结构图

3）RPC.java 在 Client.java 和 Server.java 基础上实现了 Hadoop IPC。如图 4-13 所示。RPC.java 中实现的功能主要分为两部分：

- 与客户端相关的功能包括 RPC.ClientCache、RPC.Invoker 和 RPC.Invocation，其中 RPC.Invoker 继承自前面介绍的调用句柄 java.lang.reflect.InvocationHandler。
- 与服务器相关的内部类只有 RPC.Server，它是 IPC 服务器抽象类（定义在 Server.java 中）的一个具体子类。

应该说，Hadoop IPC 的核心代码就在 Client.java、Server.java 和 RPC.java 等三个文件中，

下面分析三个类结构图中的一些比较重要的类。

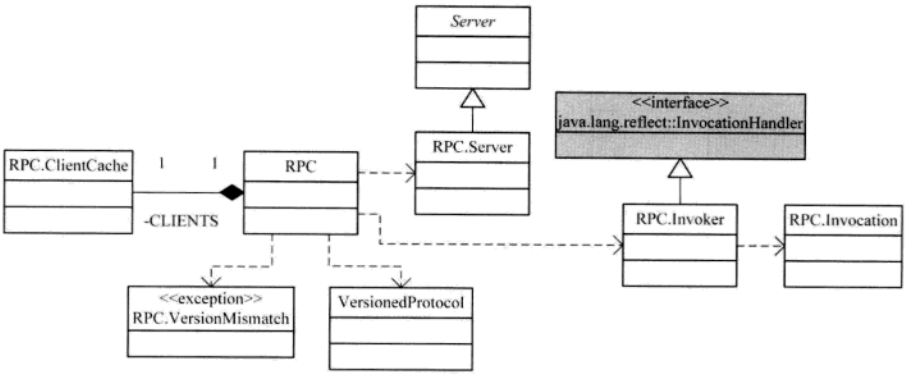



图 4-13 RPC.java 的类结构

### 1. Connection

IPC 连接（Connection）是 IPC 客户端和服务器关系的一个抽象，一个 IPC 客户端在调用服务器上的方法前，需要和服务器建立一个连接。由于客户端对连接的抽象与服务器端对连接的抽象不太一样，连接必须区分为客户端连接类 Client.Connection 和服务器连接类 Server.Connection，它们各自提供了客户端和服务器管理连接需要的相关信息和方法。Hadoop 的远程过程调用使用 TCP 协议进行通信，IPC 连接建立在 TCP 连接之上。

（1）ConnectionId

为了提高通信效率，客户端会复用到服务器的连接，并通过连接标识 ConnectionId 区分不同的连接。ConnectionId 类图如图 4-14 所示。

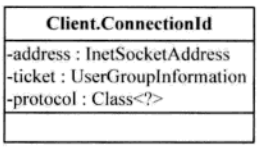



图 4-14 ConnectionId 类图

- address 是远端服务器的地址，类型是 InetSocketAddress，也就包括了主机名和服务器的监听地址。
- ticket 的类型是 UserGroupInformation。UserGroupInformation 在 org.apache.hadoop.security 包中定义，由类名即可知道，它包含了用户和用户所在用户组的一些信息。
- protocol 属性的类型是 Class<?>，它保存的是 IPC 接口对应的类对象（注意：Class 类的实例表示正在运行的 Java 应用程序中的类和接口）。

在上述三个成员变量都相等的情况下，ConnectionId 相等。连接复用指具有相同 ConnectionId 的多个 IPC 客户端共享同一个 IPC 连接。如果 ConnectionId 中的成员变量有不同（如用或不用 ticket），创建到同一 IPC 服务器上同一 IPC 接口的两个客户端，则这两个客户端会分别使用不同的 IPC 连接。连接复用一方面可以减少 Hadoop IPC 服务器 / 客户端上资源的占用，同时节省了复用连接客户端建立 IPC 连接的时间。

（2）ConnectionHeader

与 IPC 连接相关的另一个类是连接消息头 ConnectionHeader，它在 ConnectionHeader.

java 中定义。ConnectionHeader 是客户端与服务器间 TCP 连接建立后交换的第一条消息，携带的内容包括 ConnectionId 中的用户信息和 IPC 接口信息（ConnectionHeader 当然不需要携带服务器地址），这两个域用于检查服务器是否实现了 IPC 接口，并确认客户端有权利使用这个接口。

2. Call

在客户端和服务器建立连接以后，客户端就可以对外提供远程过程调用服务了。远程过程调用是对 IPC 接口上方法的调用，它在 Hadoop 源代码中的抽象就是远程调用 Call。和 Connection 类似，客户端和服务器分别定义了各自的内部远程调用类，分别是 Client.Call 和 Server.Call。通过对 Call 的抽象，客户端对远程服务器的过程调用，就可以通过 IPC 客户端和服务器间的连接，发送到服务器，处理后再返回到客户端。

Client.java 中还定义了 ParallelCall，它是 Client.Call 的子类。从 ParallelCall 的类名可以知道，它用于并行调用多个远程 IPC 服务器上的方法。Client.ParallelCall 用于并行向多个 IPC 服务器发送一组方法调用请求，并等待所有这些请求都得到响应的应答，或者超时。对于 IPC 的使用者来说，它相当于同时调用多个 IPC 服务器上的同一个方法，并得到所有服务器的应答。

Client.ParallelResults 是一个辅助类，用在上述过程中，保存已经返回的部分结果。

远程调用 Call 对象包括 Client.Call 对象和 Server.Call 对象，是在处理 IPC 客户端接口实例上的一个调用而产生的，存在于 IPC 客户端（存根）和 IPC 服务器（骨架）中的实体。Client.Call 对象通过 IPC 连接发送到服务器以后，自然会构造相应的 Server.Call 对象，那么，Client.Call 是在什么时候产生的？它又是如何产生的呢？

用户在 IPC 客户端接口实例上的一个方法调用，最终都会产生相应的 Client.Call 对象，而从方法调用到 Client.Call 对象的产生，首先会使用定义于 RPC.Java 中的调用句柄 Invoker。根据前面介绍的 Java 动态代理机制，所有在 IPC 客户端接口实例上的调用都被调用句柄 Invoker 捕获了。

RPC.Invoker 继承自 java.lang.reflect.InvocationHandler，在它的 invoke() 方法中，首先会根据方法 invoke() 的输入参数 method（被调用方法的 java.lang.reflect.Method 对象）和 args（调用参数），生成 RPC.Invocation 对象。为了区分 Client.Call 和 RPC.Invocation（Invocation 和 Call 的中文翻译都是调用），后续讨论将 Invocation 对象称为 Invoker 调用对象。RPC.Invocation 对象生成以后，Invoker 调用 Client 的 call() 方法，该方法会利用 RPC.Invocation 对象创建远程调用 Client.Call 对象，将该对象序列化并通过 IPC 连接，发送到服务器。接下来，Client.call 会等待，直到接收服务器端发送回来的远程调用结果才返回。Client.call 返回后，Invoker() 继续执行，并最终返回远程过程调用的结果。

应该说，Hadoop IPC 的实现，是利用 Java 动态代理的能力，捕获调用并将调用发送到服务器端进行处理。在运用 Java 动态代理的功能上，Hadoop IPC 只比前面介绍的动态代理的例子稍微复杂，但却实现了一个有效的远程过程调用系统，可见 Java.lang.reflect 的强大威力。

### 3. Server 处理器

Server 处理器指的是监听器（Listener）、处理器（Handler）和应答器（Responder），三者配合在 IPC 服务器上完成客户端的远程调用的处理。

其中，Listener 监听客户端发来的连接请求和连接建立后的数据请求，并调用服务器端连接对象上的相关方法。连接对象的主要工作是接收来自客户端的远程调用请求帧，反序列化以后，这些调用请求会放入一个阻塞队列中，由 Handler 处理。

Handler 根据远程调用 Call 中的上下文，调用对应的 IPC 接口实现类，完成服务器端的过程调用，调用结果经过序列化后，在连接的应答队列为空时，返回给客户端。

当客户端比较忙时（这时应答队列不为空），Handler 将调用结果放入响应队列，并由 Responder 通过 IPC 连接发送回客户端。也就是说，Handler 和 Responder 都可以发送调用结果。

## 4.5 Hadoop IPC 连接相关过程

Hadoop IPC 的实现主要有两类流程，分别是 IPC 连接相关过程和 IPC 方法调用相关过程，本节先介绍与 IPC 连接的相关过程。

IPC 连接是客户端和服务器关系的一个抽象，它的实现包括连接建立、连接上的数据读写、连接维护和连接关闭四个流程。

### 4.5.1 IPC 连接成员变量

前面介绍 IPC 连接的时候提到 IPC 连接是 IPC 客户端和服务器关系的一个抽象，它建立在 TCP 连接上，分别由客户端连接类 Client.Connection 和服务器连接类 Server.Connection 抽象。在介绍 IPC 连接的具体流程前，先来分析 Client.Connection 和 Server.Connection 的主要成员变量。

#### 1. Client.Connection 的成员变量

Client.Connection 的成员变量分为三部分：与 TCP 相关的成员变量、与 IPC 连接相关的成员变量和与远程调用的成员变量，代码如下：

```
private InetSocketAddress server; //IPC 服务器地址
private ConnectionHeader header; // 连接消息头
private ConnectionId remoteId; //IPC 连接标识

private Socket socket = null; //TCP 连接的 Socket 对象
private DataInputStream in;
private DataOutputStream out;

// 当前正在处理的远程调用
private Hashtable<Integer, Call> calls = new Hashtable<Integer, Call>();

//IPC 连接的最后一次通信时间
```

```
private AtomicLong lastActivity = new AtomicLong();

// 连接关闭标记
private AtomicBoolean shouldCloseConnection = new AtomicBoolean();

private IOException closeException; // 导致 IPC 连接关闭的异常
```

与 TCP 相关的成员变量包括：IPC 服务器的地址 server（类 InetSocketAddress 包含了建立 TCP 连接需要的 IP 地址 / 主机名和端口号信息）、TCP 连接对应的 Socket 对象 socket、以及 socket 对象上的输入流对象 in 和输出流对象 out，这部分成员变量的使用相对比简单。

与 IPC 连接相关的成员变量包括：相对不变的连接消息头 header、连接标识 header，它们在连接对象初始化的时候就已经初始化，后续 Client.Connection 工作时，也不会对它们进行修改。连接标识和连接消息头的作用前面都已经介绍过了，连接标识用于 IPC 连接复用，连接消息头是 Socket 连接建立后交换的第一条消息。在维护 IPC 连接和关闭 IPC 连接过程中，需要使用到成员变量 lastActivity、shouldCloseConnection 和 closeException。lastActivity 记录最后一次 I/O 发生的时间，如果 IPC 连接上长时间没有数据交换，即没有 I/O 发生，那么 IPC 连接会进行相应的维护工作。当需要关闭 IPC 连接时，通过置位 AtomicBoolean 型的标记 shouldCloseConnection，可以停止 Client.Connection 线程的工作（注意，Client.Connection 继承自 java.lang.Thread）。IPC 连接可以是正常关闭，也可以是因为某些异常导致异常关闭，当 IPC 连接异常关闭时，导致关闭的异常原因，会记录在 closeException 中。

与远程调用相关的变量只包含一个成员变量 calls，它保存目前 IPC 连接上的所有远程调用。成员变量 calls 是一个哈希表，其中键的类型是整形，是每个 Client.Call 对象都包含的，在一个 IPC 客户端里唯一的整数标识，哈希表的值就是 Client.Call 对象本身，通过这个标识可以很快在服务器端发送过来的应答里找出对应的 Client.Call 对象。

### 2. Server.Connection 的成员变量

Server.Connection 的成员变量比 Client.Connection 多：

```
private boolean rpcHeaderRead = false;// 状态，是否已经读入 RPC 版本号
private boolean headerRead = false;// 状态，是否已经读入连接消息头

private SocketChannel channel;
private ByteBuffer data;
private ByteBuffer dataLengthBuffer;
private LinkedList<Call> responseQueue;
private volatile int rpcCount = 0; // 当前正在处理的 RPC 请求量
private long lastContact;
private int dataLength;
private Socket socket;

// 客户端的主机名和端口
private String hostAddress;
private int remotePort;
```

```
ConnectionHeader header = new ConnectionHeader();
Class<?> protocol;

Subject user = null;

// 一个假的远程调用，作为鉴权失败时的应答
// (Fake 'call' for failed authorization response)
private final int AUTHROIZATION_FAILED_CALLID = -1;
private final Call authFailedCall =
       new Call(AUTHROIZATION_FAILED_CALLID, null, null);
private ByteArrayOutputStream authFailedResponse =
       new ByteArrayOutputStream();
```

和 Client.Connection 类似，Server.Connection 的变量也可以分为与 TCP 相关、与连接相关和与远程调用相关等三类。

与 TCP 相关的主要是 Socket 通道 channel 和配合通道工作的一些缓冲区，如 data、dataLengthBuffer 等，以及配合缓冲区工作的 dataLength 等。

与连接相关的成员变量比较多，包括连接状态管理的 versionRead、headerRead，它们的初始值都为 false。连接建立的时候，需要对客户端做一些检查，然后才允许客户端发送 IPC 请求，这样的检查包括版本检查，即检查客户端使用的 IPC 版本（注意，这里的版本是 IPC 实现的版本，不是具体 IPC 接口的版本）和服务器端的是一样的，以及连接头检查，主要检查服务器是否实现了客户端需要的 IPC 接口，还有就是客户端用户有权限使用这些接口。通过了这些检查后，也就是 versionRead、headerRead 的值都为 true 时，连接才会进入处理 IPC 调用的状态。连接建立时检查的同时会保存一些工作时需要的信息，如连接头 header（Client.Connection 也保持了这个信息）、该 IPC 连接上的远程接口 protocol、客户端用户 user 等，这些也都作为 Server.Connection 的成员变量。authFailedCall 和 authFailedResponse 用于用户鉴权 / 授权失败后对客户端的应答，具体使用方法，请参考后面 IPC 连接建立过程的分析。和连接相关的最后一个变量是 lastContact，它保持了最后一次接收到客户端数据的时间，如果客户端长时间没有发送数据，那么，服务器将会关闭连接，以释放相关资源。

与远程调用相关的成员变量包括：当前连接上的 IPC 调用应答队列 responseQueue 和该连接上正在处理的 RPC 数 rpcCount。

## 4.5.2 建立 IPC 连接

前面提到过 IPC 连接复用，这部分的代码在 org.apache.hadoop.ipc.Client 中。Client 类的成员 connections 保存 ConnectionId 到 Connection 的映射，Client 需要使用连接时，都会调用 getConnection() 方法。该方法先检查 connections 中是否有满足条件的 IPC 连接，如果有，复用连接，否则才会创建新的 IPC 连接。

getConnection() 的参数包括构造 ConnectionId 需要的远端服务器的地址、用户信息和远程接口信息，最后一个参数是 call，其类型是 Client.Call。这表明，连接是在需要的时候，也就是有 IPC 调用发生的时候才建立的。相关代码如下：

```
// org.apache.hadoop.ipc.Client 的代码片段
private Hashtable<ConnectionId, Connection> connections =
      new Hashtable<ConnectionId, Connection>();
......
private Connection getConnection(InetSocketAddress addr,
                                 Class<?> protocol,
                                 UserGroupInformation ticket,
                                 Call call)
                                 throws IOException {
   ......
   do {
      synchronized (connections) {
         connection = connections.get(remoteId);
         if (connection == null) {
            connection = new Connection(remoteId);
            connections.put(remoteId, connection);
         }
      }
   } while (!connection.addCall(call));
   ......
   connection.setupIOstreams();
}
```

getConnection() 方法主要部分在 do…while 循环中，循环内实现了连接复用。比较难理解的地方是 do…while 循环的结束条件，这需要结合 Client.Connection 的 addCall() 方法才能分析清楚。

addCall() 方法的作用是将一个 IPC 调用放入 IPC 连接中，如果连接的成员变量 shouldCloseConnection 为 true，addCall() 会返回 false，也就防止将一个 IPC 调用放入一个正处于关闭过程中的连接。IPC 连接可以在多个地方触发，进入关闭过程，但直到 Connection.close() 方法中，ConnectionId 和连接的映射关系才会被删除。连接被删除后，getConnection() 才有机会构造出一个新的连接，并将调用 addCall() 时传入的 IPC 调用放入新的 IPC 连接中。所以，getConnection() 的 do…while 循环结束条件，保证调用者能得到一个打开的、正常工作的 IPC 连接，不会获得一个即将关闭、处于清除阶段的连接。相关代码如下：

```
// org.apache.hadoop.ipc.Client.Connection 的代码片段
private class Connection extends Thread {
   ......
   public Connection(ConnectionId remoteId) throws IOException {
      ......
   }
   ......
   private synchronized void setupIOstreams() {
      if (socket != null || shouldCloseConnection.get()) {
         return;
      }
      ......
```

```
            writeHeader();

            // 更新连接维护时间
            touch();

            // 启动接受线程
            start();
            ......
        }
        ......
        private synchronized boolean addCall(Call call) {
            if (shouldCloseConnection.get())
                return false;
            calls.put(call.id, call);
            notify();
            return true;
        }
        ......
    }
```

getConnection() 获取的 Client.Connection 对象，需要通过 setupIOstreams() 方法和服务器建立连接。该方法首先使用 Java 基本套接字和服务器建立 Socket 连接。建立 Socket 连接如果失败，setupIOstreams() 会进行一定次数的重试，重试次数由 Client 的成员变量 maxRetries 指定，可通过配置项 ${ipc.client.connect.max.retries} 设置。另一个需要注意的地方是，Connection 使用的 Socket 连接设置了 TcpNoDelay 标志，禁用 TCP 的 Nagle 算法[⊖]，关闭 Socket 底层的缓冲，确保数据及时发送，配置项 ${ipc.client.tcpnodelay} 用于打开 / 关闭 TcpNoDelay 标志。

基础的 Socket 连接建立以后，setupIOstreams() 接着调用 writeHeader() 和 IPC 服务器进行握手。writeHeader() 方法写到服务器的内容包括：

❑IPC 连接魔数（“hrpc”，作用类似于文件魔数）。

❑协议版本号（目前是 4）。

❑ConnectionHeader（包括 Writable 序列化后的长度和内容）。

服务器就是根据这些内容，对客户端进行协议版本检查、接口检查和权限检查。writeHeader() 中发送 ConnectionHeader 部分，其实相当于发起了一次 RPC 调用，后面分析到方法调用处理过程的时候再来解释为什么要这样做。

writeHeader() 调用结束后，setupIOstreams() 调用 touch() 记录最后一次 I/O 发生的时间，并启动线程，该线程用于在 Socket 上读取并处理响应数据。

在远程过程调用的另一边，服务器端建立 IPC 连接的代码分散在 Listener 和 Server.Connection 中。

Listener 基于 Java NIO 开发，是一个标准的 NIO 应用。Listener 在它的构造函数中打开

⊖ Nagle 算法通过减少 TCP 连接上必须发送的包的个数来增加网络软件系统的效率，具体算法和应用场景请参考 TCP/IP 相关图书。

服务器的端口，创建 Selector 并开始监听。需要注意参数 backlogLength，它由配置项 ${ipc.server.listen.queue.size} 指定，backlogLength 是调用 ServerSocket.bind() 时可以额外提供的一个参数，用于指定在监听端口上排队的请求的最大长度，队列满了以后到达的客户端连接请求，会被拒绝。

Listener.run() 实现了 NIO 中的选择器循环，即调用选择器的 select() 方法并处理事件，它的处理过程和前面介绍的 NIO 实例有点差别，这里只通过 doRead() 方法处理 OP_READ，以及通过 doAccept() 方法处理 OP_ACCEPT 事件，不关注通道的 OP_WRITE 事件。

doAccept() 方法接受客户端的连接请求、注册 Socket 到选择器并创建 Server.Connection 对象。

doRead() 方法通过 Server.Connection.readAndProcess() 读取并处理数据，如果处理中出现问题，doRead() 会通过 Server.closeConnection() 关闭连接。doRead() 方法还会更新连接的成员变量 lastContact（最后一次接收到客户端数据的时间），在服务器端维护 IPC 连接。相关代码如下：

```
public void run() {
   ......
   while (running) {
      SelectionKey key = null;
      try {
         selector.select();// 下面是典型的选择器代码
         Iterator<SelectionKey> iter = selector.selectedKeys().iterator();
         while (iter.hasNext()) {
            key = iter.next();
            iter.remove();
            try {
               if (key.isValid()) {
                  if (key.isAcceptable())
                     doAccept(key);// 处理 acctpt
                  else if (key.isReadable())
                     doRead(key);// 处理 read
               }
            } catch (IOException e) {
            } // catch 里什么事情都没做
            key = null;
         } //while (iter.hasNext())
      } catch (OutOfMemoryError e) {
         ......// 应对没有内存的情况
         closeCurrentConnection(key, e);
         cleanupConnections(true);// 后面讨论关闭连接时再介绍
         try { Thread.sleep(60000); } catch (Exception ie) {}
      } catch (InterruptedException e) {
         ......// 其他异常情况
      } catch (Exception e) {
         closeCurrentConnection(key, e);
      }
```

```
            cleanupConnections(false);
        }
        ......
    }
```

Server.Connection 中最主要的方法是 readAndProcess()，doRead() 调用这个方法读取客户端发送过来的数据并处理。

在介绍 Server.Connection 成员变量的时候提到 Server.Connection 是有状态的，连接状态由 rpcHeaderRead 和 headerRead 管理。按照 Client.Connection.writeHeader() 发送的握手信息，如果顺利读取到 IPC 连接魔数和协议版本号并完成版本检查，则设置 rpcHeaderRead 为 true 然后进入连接头检查，连接头检查通过后，headerRead 置为 true，服务器开始处理客户端的 IPC 请求。

在版本检查阶段，readAndProcess() 先读 IPC 连接魔数和协议版本号，IPC 连接魔数一共 4 字节，正好是缓冲区 dataLengthBuffer 的长度（这是精心设计的），协议版本号 1 字节，通过 versionBuffer 缓冲区读入。数据读入以后，立即比较客户端发送过来的 IPC 连接魔数和协议版本号和服务器是否一致，如果一致，设置 versionRead 为 true；否则向客户端返回 -1 表示失败。相关代码如下：

```
public int readAndProcess() throws IOException, InterruptedException {
    ......
    if (!versionRead) {
        ......
        int version = versionBuffer.get(0);
        dataLengthBuffer.flip();
        if (!HEADER.equals(dataLengthBuffer) || version != CURRENT_VERSION)
        {
            ......
            return -1;   // 比较 IPC 连接魔数和协议版本号失败
        }
        ......
    }
    ......
    if (data.remaining() == 0) {
        ......
        if (headerRead) {
            ......
        } else {
            processHeader();// 处理连接头
            headerRead = true;
            data = null;

            try { // 对连接进行鉴权
                authorize(user, header);
                ......
            } catch (AuthorizationException ae) {
                authFailedCall.connection = this;
                setupResponse(authFailedResponse, authFailedCall,
```

```
                        Status.FATAL, null,
                        ae.getClass().getName(), ae.getMessage());
                responder.doRespond(authFailedCall);
                return -1;// 鉴权失败关闭连接
            }
        }
        ......
    }
}
......
private void processHeader() throws IOException {
    DataInputStream in =
        new DataInputStream(new ByteArrayInputStream(data.array()));
    header.readFields(in);
    try {
        String protocolClassName = header.getProtocol();
        if (protocolClassName != null) {
            protocol = getProtocolClass(header.getProtocol(), conf);
        }
    } catch (ClassNotFoundException cnfe) {
        throw new IOException("Unknown protocol: " + header.getProtocol());
    }
    ..... // 获取用户信息
}
```

对连接头 ConnectionHeader 进行处理的逻辑在 readAndProcess() 的底部，包含两个步骤：

1）第一个步骤是调用读入 ConnectionHeader，保证服务器实现了 IPC 接口和获取用户信息。

这部分逻辑实现在 processHeader()。需要注意的是，getProtocolClass() 其实只是判断 Server 实现了 ConnectionHeader 中要求的 IPC 接口，不做其他检查，如服务器是否实现相同的接口协议、协议版本是否一致等。getProtocolClass() 的处理结果保存在类型为 Class<?> 的成员变量 protocol 中，供后续方法调用时使用。

2）第二个步骤是调用 authorize()，保证用户有相关的权限访问远程接口。

Service.authorize() 是类的虚方法，具体的鉴权策略由子类实现。鉴权失败时 authorize() 抛出异常 AuthorizationException，这个异常通过 setupResponse() 打包成一个正常 IPC 调用应答，通过 responder.doRespond 发送回客户端处理。和版本检查失败一样，鉴权失败也会关闭连接。

### 4.5.3　数据分帧和读写

TCP 通信是基于字节流的，没有消息边界的概念，因此，有了 IPC 连接以后，就必须结合 IPC 连接上的消息，考虑数据的帧结构。一般来说，有多种技术使通信中的接收者能够准确地找到消息的结束位置：

- 定长消息：通信双方发送的消息长度是固定的，接收者只需要简单地将数据读入对应的缓冲区中，就可以获得消息。

- 基于定界符：消息的结束由唯一标记指出，消息发送者在传输完数据后，添加一个特殊的字节序列。这个特殊的标记不能在传输的数据中出现，接收者简单地扫描输入信息并查找定界符，并将定位符前面的数据形成消息交给上层应用。
- 显式长度：在具体消息前面附加一个固定大小的字段，指示该消息包含多少字节。接收者首先以定长消息的方式接受长度字段，然后根据这个长度接收消息。

IPC 连接涉及两个方向的数据分帧，IPC 客户端发送请求用的是“显式长度”方法，长度是 int 类型。在 IPC 客户端中，Connection.sendParam() 用于将远程调用 Call 形成消息发送出去。sendParam() 方法首先将要发送的数据写入 DataOutputBuffer 对象中，以同时获取数据长度和消息内容，然后通过输出流 out 发送数据。代码如下：

```
public void sendParam(Call call) {
   ......
   DataOutputBuffer d=null;
   try {
      synchronized (this.out) {
         ......
         // 将远程调用 Call 串行化到一个 DataOutputBuffer 中
         d = new DataOutputBuffer();
         d.writeInt(call.id);
         call.param.write(d);
         byte[] data = d.getData();
         int dataLength = d.getLength();
         out.writeInt(dataLength);// 输出数据长度
         out.write(data, 0, dataLength);// 输出数据
         out.flush();
      }
   }
   ......
}
```

服务器端接收数据在 Connection. readAndProcess()，上一节已经分析过这个方法的部分代码。与数据分帧相关的代码如下（代码可以分为两部分：读取“显式长度”和读取消息）：

```
public int readAndProcess() throws IOException, InterruptedException {
   while (true) {
      // 最多读入一个 RPC 的数据
      int count = -1;
      if (dataLengthBuffer.remaining() > 0) {
         // 读入“显式长度”，dataLengthBuffer 一共 4 字节
         count = channelRead(channel, dataLengthBuffer);
         if (count < 0 || dataLengthBuffer.remaining() > 0)
            return count;  //“显式长度”还没有读完，或者读过程中出错
      }

      if (!versionRead) {  // 处理握手过程中的版本信息
         ......
      }
```

```
            if (data == null) {
                dataLengthBuffer.flip();
                dataLength = dataLengthBuffer.getInt();// 数据长度已读取

                if (dataLength == Client.PING_CALL_ID) {// 心跳消息
                    dataLengthBuffer.clear();
                    return 0;  //ping message
                }
                data = ByteBuffer.allocate(dataLength); // 为读取数据分配缓冲区
                incRpcCount();  // Increment the rpc count
            }

            count = channelRead(channel, data);

            if (data.remaining() == 0) {// 读到一个完整的消息
                dataLengthBuffer.clear();
                data.flip();
                if (headerRead) {
                    processData();// 处理数据
                    data = null;
                    return count;
                } else {
                    ......
                }
            }
            return count;
        }
    }
```

读取“显式长度”使用的是缓冲区 dataLengthBuffer，它的长度是 4 字节，即一个整数的长度。readAndProcess() 方法的第一部分代码先将数据读入 dataLengthBuffer，直到读满；接着通过缓冲区的 getInt() 方法得到消息长度，并分配读取消息的缓冲区 data。其中，消息长度为 Client.PING_CALL_ID 的是心跳消息，下一节会介绍这个消息的用途。

代码中间部分的 channelRead() 调用，会将通道数据读入缓冲区 data，直到缓存区剩余空间为 0，这时，服务器已经完整地读入一条消息，可以进行后续的处理了。根据是否读入 ConnectionHeader，处理消息逻辑执行不同的分支。如果连接头已经读入，表明这里读到是一个 IPC 请求，调用 processData() 方法处理该消息。

通过这部分代码的分析，读者也应该了解到，为什么 IPC 连接魔数是“hrpc”，它正好是 4 字节，可以放入缓冲区 dataLengthBuffer 中；同时，为什么在写入 ConnectionHeader 对象时，要先写入它的长度，因为在服务器端，它被当成一个远程调用请求接收（只是在后续处理上有差别）。

IPC 客户端到服务器通信用的是“显式长度”方法，而服务器到客户端用的是“定长消息”的变种：即利用前面介绍的序列化机制 Writable 发送响应。使用 Writable 时，通信双方不需要定界符，也不需要显式长度，其本质是一种“定长消息”，由客户端 Writable 对象保证读入服务器端写入的所有数据。服务器端往客户端写数据的代码在 Server.setupResponse()

方法中，客户端读数据的代码在 Connection.receiveResponse() 方法中。相对于客户端到服务器通信来说，这部分的代码比较简单，就不分析了。

### 4.5.4 维护 IPC 连接

IPC 连接上如果长时间没有发生远程调用，客户端会发送一个特殊的心跳消息给服务器，用于维护客户端 / 服务器间的连接。其中，上次发生 IPC 通信的时间记录在 Connection 的成员变量 lastActivity 中，图 4-15 中是代码中用到该变量的地方。

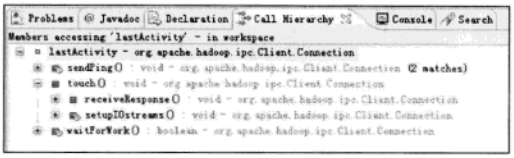



图 4-15 代码中用到 lastActivity 变量的地方

Connection.touch() 方法更新 lastActivity 为当前时间，它在 setupIOstreams() 和 receiveResponse() 中被调用。setupIOstreams() 方法的功能已经分析过了，receiveResponse() 用于处理服务器发送回来的数据，也就是说，收到服务器发送回来的消息，lastActivity 就会被更新。如果当前时间和 lastActivity 的差超过某个值（保存在 Client.pingInterval，可以通过配置项 ${ipc.ping.interval} 进行配置，默认是 1 分钟），意味着需要发送心跳消息。发送心跳消息使用 sendPing() 方法，代码如下：

```
private synchronized void sendPing() throws IOException {
   long curTime = System.currentTimeMillis();
   if ( curTime - lastActivity.get() >= pingInterval) {
      lastActivity.set(curTime);
      synchronized (out) {
         out.writeInt(PING_CALL_ID);// 发送 -1
         out.flush();
      }
   }
}
```

当前时间和上次心跳时间的差大于 Client.pingInterval 时，sendPing() 方法会更新 lastActivity，同时通过和服务器端的输出流 out 输出整数 PING_CALL_ID（值为 -1），并将缓存区的数据刷出。

需要说明的是，对输出流 out 的操作使用了 synchronized 同步，这保证了数据的帧结构不会被破坏。sendPing() 方法只被 PingInputStream 的 handleTimeout() 方法调用，而 handleTimeout() 又只被 PingInputStream 的 read() 方法调用。PingInputStream 继承自 FilterInputStream，FilterInputStream 提供了在某一特定流的基础上提供附加功能的能力。具体到 PingInputStream 类，它在原有 Socket 提供的输入流 SocketInputStream 上添加心跳检查的能力，代码如下：

```
private synchronized void setupIOstreams(){
    ......
    this.socket.setSoTimeout(pingInterval);
    ......
}

private class PingInputStream extends FilterInputStream {
    // 构造函数，需要一个 InputStream
    protected PingInputStream(InputStream in) {
        super(in);
    }

    // 处理超时，如果客户端处于运行状态，则调用 sendPing() 方法
    private void handleTimeout(SocketTimeoutException e) throws…… {
        if (shouldCloseConnection.get() || !running.get()) {
            throw e;
        } else {
            sendPing();
        }
    }

    public int read() throws IOException {
        do {
            try {
                return super.read();
            } catch (SocketTimeoutException e) {
                handleTimeout(e);// 处理 SocketTimeoutException 异常
            }
        } while (true);
    }
    ......
}
```

如果 IPC 客户端和服务器有通信的话，IPC 客户端会通过 PingInputStream.read() 读取数据，最终调用 Connection.receiveResponse() 更新最后一次 I/O 发生的时间 lastActivity；否则，super.read()（在 PingInputStream 的 read 方法中）会抛出 SocketTimeoutException 异常，该异常表示读取套接字时发生超时，这时就可以调用 handleTimeout()，处理（可能的）超时。

SocketInputStream.read() 方法的超时时间在 setupIOstreams() 中通过 Socket 的 setSoTimeout() 方法设置，值是 Client.pingInterval。

handleTimeout() 很简单，如果客户端处于正常工作状态，调用 sendPing() 方法判断是否需要发送心跳消息。正是通过 PingInputStream 的巧妙设计，IPC 客户端不需要启动额外的线程，而是使用读 Socket 数据的那个线程，实现了维护 IPC 连接的心跳检查。

服务器端的连接维护相对简单，当接收到客户端的数据时，服务器连接也会更新它的成员变量 lastContact（如下）：

```
private synchronized void doRead() throws IOException {
    ......
```

```
        c.setLastContact(System.currentTimeMillis());
        ......
    }

    public void setLastContact(long lastContact) {
        this.lastContact = lastContact;
    }
```

后续在 readAndProcess() 处理数据时，服务器端为“长度”是 PING_CALL_ID 的请求相应做了特殊的处理：简单地清理长度缓冲区 dataLengthBuffer，为再次读入一个正常的消息长度做准备。

### 4.5.5 关闭 IPC 连接

当用户通过 Client.stop() 关闭整个客户端，或者 IPC 连接上出现错误，或者 IPC 连接上长时间没有发生 IPC 调用时，必须关闭连接。

用户通过 Client.stop() 方法关闭整个客户端，在 stop() 方法中，首先设置成员变量 running 为 false，这标识客户端停止运行，进入清理阶段；接着，通过线程的 interrupt() 方法唤醒各个 IPC 连接线程并执行关闭连接操作，最后，Client.stop() 会等待，直到所有连接清理结束。

IPC 连接上出现错误时，调用 Connection.markClosed() 方法设置成员变量 shouldCloseConnection，并通知连接执行关闭动作。这个方法需要一个类型为 IOException 的参数，该参数包含连接关闭的原因。markClosed() 方法被多个方法调用，如图 4-16 所示。

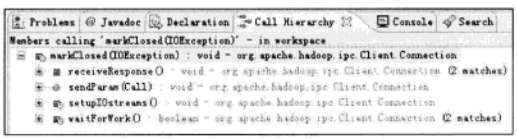



图 4-16 调用 markClosed() 的方法

IPC 连接上长时间没有发生 IPC 调用的判断在 waitForWork() 中，代码如下：

```
//waitForWork() 如果返回 false，会导致 IPC 连接关闭
private synchronized boolean waitForWork() {
    // 注意 if 语句的判断条件，与的条件包括：1）目前没有正在处理的远程调用；
    //2）连接不需要关闭；3）客户端还处于运行状态
    if (calls.isEmpty() && !shouldCloseConnection.get()  && running.get())
    {
        long timeout = maxIdleTime-
            (System.currentTimeMillis()-lastActivity.get());// 等待时间

        if (timeout>0) {
            try {
                // 通过 wait 等待，可能被到达的数据打断，
                // 也可能被 Client.stop() 打断或超时
                wait(timeout);
```

```
            } catch (InterruptedException e) {}
        }
    }

    if (!calls.isEmpty() && !shouldCloseConnection.get() && running.get())
    {
        return true;// 需要等待连接上返回的远程调用结果
    } else if (shouldCloseConnection.get()) {
        return false;//shouldCloseConnection 置位，关闭连接
    } else if (calls.isEmpty()) {
        markClosed(null);
        return false;// 连接长时间处于空闲状态，关闭
    } else {
        markClosed((IOException)new IOException().initCause(
            new InterruptedException()));
        return false;// 连接上还有未结束的远程调用，客户端被打断，记录原因
    }
}
.....
public void run() {
    .....
    while (waitForWork()) {
        receiveResponse();// 如果连接上还有数据要处理，处理数据
    }

    close();// 关闭连接
    .....
}
```

waitForWork() 方法如果返回 false，run() 方法将会从 while 循环中返回，并执行 close() 方法关闭连接。在 waitForWork() 中，如果目前没有正在处理的远程调用，同时 shouldCloseConnection 没被置位，且客户端的 running 标志位为 true 时，计算等待时间，并调用 wait() 方法等待。

下面几种情况会结束在 wait() 方法上的等待，包括：

❑发生一次远程调用，也就是说，需要等待连接上返回的远程调用结果；

❑markClosed() 被调用，shouldCloseConnection 为 true，可以关闭连接；

❑wait() 方法超时，这意味着 IPC 连接长时间处于空闲状态；

❑Client.stop() 关闭整个客户端，这时 running 被设置为 false，也可以关闭连接。

wait() 方法结束后需要对结束的原因进行判断，并分别处理。只有在连接正常（条件：!shouldCloseConnection），客户端处于工作状态（条件：running）和有远程调用正在处理的情况下，waitForWork() 才会返回 true，这时，run() 方法调用 receiveResponse() 等待服务器返回数据。其他情况，waitForWork() 返回 false，关闭连接。需要注意的是，只有按照 waitForWork() 中 else…if 子句的顺序，才能区分出 shouldCloseConnection 为 true、IPC 连接长时间空闲和客户端被关闭等三种连接关闭情况。

Connection.close() 方法相对简单，在连接列表中移除当前连接后，关闭输入 / 输出流，并调用 Connection.cleanupCalls() 方法清理未完成的远程调用。相关代码如下：

```
private void cleanupCalls() {
   Iterator<Entry<Integer, Call>> itor = calls.entrySet().iterator() ;
   while (itor.hasNext()) {
      Call c = itor.next().getValue();
      c.setException(closeException); // local exception
      itor.remove();
   }
}
```

Connection.cleanupCalls() 通过为当前连接上未完成的远程调用 Call 设置异常，结束远程调用。

服务器端也可能因为多种原因发起连接关闭操作，如图 4-17 所示。

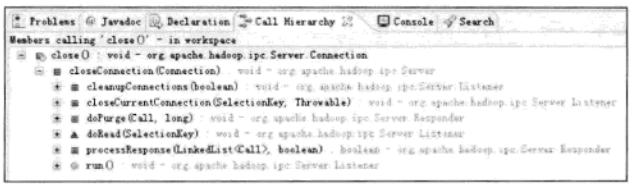



图 4-17 连接关闭操作的原因

连接关闭操作的原因包括：

- 服务器负载太高，出现 OutOfMemoryError 异常；
- 长时间没有收到客户端发送过来的心跳消息；
- 服务器响应在很长时间内不能发送给客户端；
- 客户端主动关闭连接；
- 处理服务器响应时出错；
- 服务器关闭。

当服务器负载太高（出现 OutOfMemoryError 异常）或心跳检查失败（在 Listener 的主循序中）时，cleanupConnections() 被调用，清理连接以释放资源。该方法每次只处理服务器当前 IPC 连接的一部分，以避免阻塞 Listener 的选择器循环。对于待清理的连接，cleanupConnections() 简单地调用 Server.closeConnection() 方法，在服务器连接列表中移除相关记录并调用服务器连接 Connection 的 close() 方法清理资源、关闭通道（服务器端用的是 NIO 的通道）、关闭 Socket。

其他连接关闭的原因在后面 IPC 方法调用处理过程中再逐一介绍。

## 4.6 Hadoop IPC 方法调用相关过程

研究了 IPC 连接的相关过程后，本节分析建立在 IPC 连接基础上的远程方法调用过程。

IPC 方法调用，在客户端只是一个比较复杂的 Java 动态代理应用；在服务器端，由 Listener、Handler 和 Responder 配合，完成请求读取、请求处理和请求应答三个步骤。

远程过程调用是对 IPC 接口上方法的调用，在继续讨论远程过程调用之前，有必要仔细回顾与 Java 接口和 Java 接口方法相关的基础知识。

### 4.6.1　Java 接口与接口体

Java 接口[⊖]是 Java 中重要的语言结构，一般包含一组方法声明（没有实现的方法）和一些常量。Java 接口用于描述类具有什么功能，而不需要给出功能的具体实现。Java 类可以实现一个或者多个接口，以提供 Java 接口中描述的功能。如 Writable 接口描述了 Hadoop 的序列化功能，WritableComparable 接口则继承自 Writable 接口和 java.lang.Comparable 接口，WritableComparable 接口同时描述了 Hadoop 的序列化功能和对象顺序的自然比较功能。

Writable 和 java.lang.Comparable 都是很简单的接口，而 IPC 可能发生在所有符合规范的 Java 接口上，什么是符合规范的 Java 接口呢？ Java 规范中，接口声明规范如下[⊖]：

```
NormalInterfaceDeclaration:
   InterfaceModifiers_opt interface Identifier TypeParameters_opt
      ExtendsInterfaces_opt InterfaceBody

InterfaceModifiers:
   InterfaceModifier
   InterfaceModifiers InterfaceModifier

InterfaceModifier: one of
   Annotation public protected private
   abstract static strictfp

ExtendsInterfaces:
   extends InterfaceType
   ExtendsInterfaces, InterfaceType

InterfaceType:
   TypeDeclSpecifier TypeArguments_opt

InterfaceBody:
{ InterfaceMemberDeclarations_opt }
```

从上面的接口声明规范可以看出：出现在关键字 interface 前面的是可选接口修饰符 InterfaceModifiers，可以是 public、protected、private、abstract、static 或 strictfp。关键字 interface 后跟着的是接口的名称 Identifier，如果接口声明了一个或多个类型变量（TypeParameters，通过尖括号进行界定），这个接口是泛型接口，如 java.lang.Comparable

⊖ Java 中接口有两种含义：Java 接口，Java 语言中存在的结构，有特定的语法和结构；接口，一个类所具有的方法的特征集合，是一种逻辑上的抽象。前者叫做“Java 接口”，后者叫做“接口”，本节讨论的是“Java 接口”。

⊖ 由于篇幅关系，不会完整讨论 Java 接口的所有声明。

接口就是一个泛型接口。接口声明如果提供了一个 extends 子句，也就是规范中的 ExtendsInterfaces 部分，表明被声明的接口扩展了其他接口，从而继承了其他每个接口的成员类型、方法和常量。接口声明的最后一部分是必选的接口体，接口体声明接口成员。

接口体可以包含 4 种成员，包括：

❑常量声明（ConstantDeclaration）

❑抽象方法声明（AbstractMethodDeclaration）

❑成员类声明（ClassDeclaration）

❑成员接口声明（InterfaceDeclaration）

其中，成员类声明是封闭在某一个类或接口里面的类，成员接口是封闭在某一个类或接口里的接口。值得关注的是，成员接口也可以作为 Hadoop IPC 接口应用于远程方法调用中。相关规范如下：

```
InterfaceBody:
   { InterfaceMemberDeclarations_opt }

InterfaceMemberDeclarations:
   InterfaceMemberDeclaration
   InterfaceMemberDeclarations InterfaceMemberDeclaration

InterfaceMemberDeclaration:
   ConstantDeclaration
   AbstractMethodDeclaration
   ClassDeclaration
   InterfaceDeclaration

AbstractMethodDeclaration:
   AbstractMethodModifiers_opt TypeParameters_opt ResultType
   MethodDeclarator Throws_opt ;

AbstractMethodModifiers:
   AbstractMethodModifier
   AbstractMethodModifiers AbstractMethodModifier

AbstractMethodModifier: one of
   Annotation public abstract

ResultType:
   Type
   void

MethodDeclarator:
   Identifier ( FormalParameterList_opt )
```

接口体的 4 种成员中最关心的是抽象方法，它是远程方法调用中发生调用的目标。抽象方法的规范从 AbstractMethodDeclaration 开始，包括：

❑public 或者 abstract：可选的修饰符。

❑TypeParameters：可选的，应用于泛型接口的类型变量。

❑ResultType：必选方法结果，结果可以是空，也可以是其他返回值类型。

❑MethodDeclarator：必选的方法声明。

❑Throws：可选的 throws 子句。

其中，方法声明 MethodDeclarator 包括方法名称 Identifier 和可选的方法形式参数列表 FormalParameterList[⊖]。由于存在方法重载，单独通过方法名称 Identifier 不能够确定一个方法，必须由方法名称结合方法的形式参数列表，才能唯一确定我们调用的方法。

通过上面对接口定义规范的讨论，可以明确发生方法调用时的上下文信息，包括：

❑接口名称：被调用方法所在接口的规范名。接口的规范名可以通过调用接口的 Class 对象的 getName() 方法获得，在前面给出的 IPC 实例中，IPC 接口 IPCQueryStatus 的规范名是：org.hadoopinternal.ipc.IPCQueryStatus。

❑方法名称：远程方法调用调用的方法名，如 getFileStatus。

❑形式参数列表：被调用方法的形参类型列表。由于存在方法重载，只有结合方法名称和参数列表，才能唯一确定一个方法。

❑实际参数列表：调用方法时传入的实际参数列表。

远程过程调用发生时，通过 Java 的反射机制，可以获得上述信息，这些信息需要传输到服务器端，并被服务器上的实际方法调用使用。

方法调用结束返回时，根据方法的定义可能出现以下两种情况：

❑正常返回，返回可以带返回值，也可以不带返回值；

❑有异常发生，即方法执行过程中抛出异常。

当设计服务器端到客户端的应答消息时，必须考虑上述两种情况。

## 4.6.2　IPC 方法调用成员变量

根据上述讨论，IPC 发生时将会涉及 3 个用于抽象调用的类：RPC.Invocation、Client.Call 和 Server.Call，其类图如图 4-18 所示。

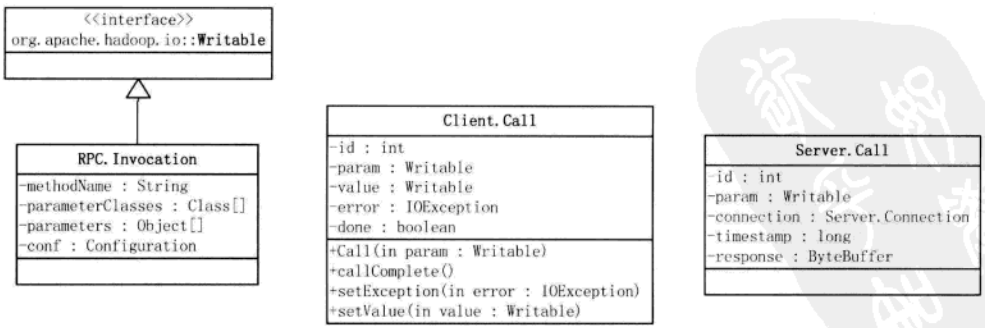



图 4-18　RPC.Invocation、Client.Call 和 Server.Call 类图

⊖ FormalParameterList 在 Java 语言规范中有详细介绍，请参考相关资料。

### 1. RPC.Invocation

当用户在IPC客户端接口实例上的调用被调用句柄捕获以后，会生成一个RPC.Invocation实例，它包含了前面讨论的方法调用上下文中的方法名称、形式参数列表和实际参数列表。在类中，它们对应的成员变量分别是：methodName、parameterClasses和parameters。

细心的读者会发现，RPC.Invocation不包含方法调用上下文中的接口名称，回忆前面建立IPC连接的过程，连接头中ConnectionHeader已经包含了接口名称，所以，在IPC连接上进行方法调用时，就不需要再提供发生方法调用对应的接口名称了。RPC.Invocation实现了Writable接口，可以使用Hadoop的序列化机制进行序列化。

RPC.Invocation实例创建后，需要通过IPC连接传送到服务器。这个过程中需要添加一些IPC连接处理需要的信息，形成一个Client.Call对象。

### 2. Client.Call

Client.Call类的成员变量有：

```
private class Call {
   int id; //标识符
   Writable param;//RPC.Invocation对象
   Writable value;//返回值
   IOException error;//异常返回时的异常
   boolean done; //该调用是否已经完成
   ……
```

由于连接共享，一个连接上可能有多个IPC调用正在执行，为了区分不同的IPC调用，连接为每一个Client.Call对象分配了标识符，即成员变量ID。接下来的成员保存的是远程过程调用的参数param，它就是上面分析的RPC.Invocation对象。

变量value和error分别用于调用结束时的两种情况：如果是正常返回，将返回值赋予value；如果是异常返回，将异常包装在一个RemoteException中，赋予error。

布尔变量done置位表示IPC连接上的远程调用已经完成，这时，成员变量value和error必有一个不是null，即使方法没有返回值，即方法的ResultType为void，变量value中还是会保存一个NullWritable对象。

### 3. Server.Call

Client.Call通过IPC连接发送到Server侧，Server侧会形成一个Server.Call对象，Server.Call的成员变量和Client.Call类似：

```
private static class Call {
   private int id; //标识符
   private Writable param;//RPC.Invocation对象
   private Connection connection; //IPC连接
   private long timestamp;//用于超时检查
   private ByteBuffer response; //应答，可能是正常返回值，也可能是异常
   ……
```

其中，Server.Call 的 id 和 param 的含义和 Client.Call 一样。Server.Call 中的应答，不管是方法执行成功，还是异常，都把序列化后的结果保存在 response 中，变量 response 还扮演着类似 Client.Call.done 的角色，如果为空，表明方法调用还没有结束。

Server.Call 与 Client.Call 的主要差别在于变量 connection 和 timestamp。这里的 connection 是服务器端的 IPC 连接抽象，用于发送应答，即保存在 response 的调用结果，需要通过 connection 发送会客户端。成员变量 timestamp 用于超时检查，如果超过一定时间请求没有被处理，或者应答没有被发生给客户端，该应答会被清理掉。超时检查和应答清理，在后面的 Responder 分析中有详细介绍。由于有两个清理时机，所以，接收到请求时，timestamp 会被更新为当前时间，调用结束后，timestamp 会再次被更新。

### 4.6.3　客户端方法调用过程

在 4.4.1 节的例子中，应用程序调用 getFileStatus() 时，开始了一个客户端远程方法调用的处理过程，代码如下：

```
IPCQueryStatus query=(IPCQueryStatus) RPC.getProxy(IPCQueryStatus.class,
       IPCQueryServer.IPC_VER,addr,new Configuration());
IPCFileStatus status=query.getFileStatus("/tmp/testIPC");
```

这个过程的入口在 RPC.Invoker.invoke()，RPC.Invoker 类实现了前面分析过的动态代理调用转发接口 java.lang.reflect.InvocationHandler，invoke() 方法是这个接口的唯一方法。RPC.Invoker 很简单，最重要的成员变量是 client，其类型为 org.apache.hadoop.ipc.Client，即 RPC 客户端，如代码所示，在构造函数中对象通过 ClientCache.getClient() 方法获得该成员变量。代码如下：

```
private static class Invoker implements InvocationHandler {
   ......
   private Client client;
   ......
   public Invoker(InetSocketAddress address, UserGroupInformation ticket,
                  Configuration conf, SocketFactory factory) {
      ......
      this.client = CLIENTS.getClient(conf, factory);
   }

   public Object invoke(Object proxy, Method method, Object[] args)
                        throws Throwable {
      ......
      ObjectWritable value = (ObjectWritable) client.call(
            new Invocation(method, args),address,
            method.getDeclaringClass(),ticket);
      ......
      return value.get();
   }
```

```
    /* 关闭 IPC 客户端，简单调用 ClientCache 的静态方法 stopClient()*/
    synchronized private void close() {
        ......
    }
}
```

RPC.Invoker 最重要的方法就是 invoke()，前面介绍 InvocationHandler 时已经分析了 proxy、method 和 args 这三个参数的含义。在 Invoker.invoke 中，首先根据 method 和 args 参数构造 Invocation 对象，接着调用 Client.call() 方法发送 IPC 请求并获取结果，其结果通过 value.get() 返回给调用者。

注意：invoke 的返回值包裹在 ObjectWritable 中，这样，既可以返回 null，也可以返回 Java 基本类型，还可以返回其他继承自 Writable 的类型；如果调用出现异常，那么异常在 Client.call() 中抛出，Invoker.invoke() 不需要做任何后续处理。

Client.call() 方法代码如下：

```
public Writable call(Writable param, InetSocketAddress addr,
                     Class<?> protocol, UserGroupInformation ticket)
                           throws InterruptedException, IOException {
    Call call = new Call(param);
    Connection connection = getConnection(addr, protocol, ticket, call);
    connection.sendParam(call);// 在 IPC 连接上发送调用的相关信息
    boolean interrupted = false;
    synchronized (call) {
        while (!call.done) {
        try {
            call.wait();// 等待结果
        } catch (InterruptedException ie) {
            interrupted = true;// 远程调用被打断
        }
    }

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

    if (call.error != null) {
        if (call.error instanceof RemoteException) {
            call.error.fillInStackTrace();
            throw call.error;// 远程调用异常返回，抛异常给本地调用者
        } else {// 本地处理出现的异常
            throw wrapException(addr, call.error);
        }
    } else {
        return call.value; // 调用正常结束，返回结果
    }
}
```

以上代码首先根据输入参数 param 构造一个 Client.Call 对象，其中 param 就是包含方法调

用的方法名和参数的 Invocation 对象。接着，利用 getConnection() 方法获取 IPC 连接，然后把 Client.Call 对象中和调用相关的信息发送出去，sendParam() 方法已经在 4.4.3 节中介绍过。

参数发送给服务器以后，客户端开始等待服务器发送回来的应答，通过典型的 wait() 循环[⊖]实现。当接收到应答（包括正常返回或异常返回）或者 IPC 客户端发现错误中断等待过程后，循环结束，接下来就是根据不同的情况，处理异常或方法调用返回。

Client.Call 对象中的 call.wait() 等待，必然有对应的 call.notify()。在 Call.callComplete() 方法中，布尔变量 Call.done 标志位被置位，即远程调用结果已经准备好，同时，callComplete() 调用 notify()，通知 Client.call() 方法停止等待。调用 Call.callComplete() 的方法如图 4-19 所示。

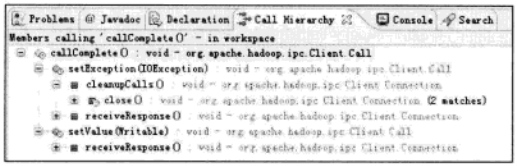



图 4-19　Call.callComplete() 被调用的方法

Call.callComplete() 被 Call.setValue() 调用，用于设置正常的返回值；被 Call.setException() 调用，用于设置异常返回。通过上面的调用树读者可以发现，Call.setException() 被 Connection.receiveResponse() 调用，也被 Connection.cleanupCalls() 调用。

在分析 IPC 连接关闭的时候已经知道，在关闭时包含了清理未完成的远程调用的动作，该动作就是：调用 setException()，为连接上的每一个 Client.Call 设置一个 IOException。

Call.callComplete() 其他可能的调用均来自 Connection.receiveResponse()，如方法名 receiveResponse() 用于接收服务器发送的调用处理结果。相关代码如下：

```
private void receiveResponse() {
   ......
   try {
      int id = in.readInt();// 获取调用 ID
      ......
      Call call = calls.remove(id);

      int state = in.readInt();// 远程调用的处理结果
      if (state == Status.SUCCESS.state) { // 成功
         Writable value = ReflectionUtils.newInstance(valueClass, conf);
         value.readFields(in); // 读取返回值
         call.setValue(value);
      } else if (state == Status.ERROR.state) { // 异常
         call.setException(
               new RemoteException(WritableUtils.readString(in),
               WritableUtils.readString(in)));
      } else if (state == Status.FATAL.state) { // 致命错误
```

⊖ 为什么要用 wait() 循环，请参考 Java 多线程编程的相关资料。

```
            markClosed(new RemoteException(WritableUtils.readString(in),
                  WritableUtils.readString(in)));// 关闭连接
         }
      } catch (IOException e) {
         markClosed(e);
      }
   }
```

前面在介绍IPC连接上的数据读写时，由于涉及应答的细节，没有详细分析receiveResponse()。其实这个方法的处理过程相当简单。首先获取应答对应的Client.Call标识符从而找到对应的Client.Call对象，然后读取IPC的处理状态。

状态有三种：如果是成功应答，创建返回值对象（继承自Writable）并完成反序列化，通过setValue()方法设置Client.Call的返回值；如果是错误（ERROR）或致命错误（FATAL），对应的处理是设置Client.Call异常返回。其中，致命错误调用markClosed()方法后，不但会为Call设置错误码，还会关闭连接。

Hadoop IPC客户端的处理，其实只比动态代理实例稍微复杂：代理对象上的调用被InvocationHandler捕获后，请求被打包并通过IPC连接发送到服务器上，客户端等待并在服务器的处理应答到达后，根据应答生成并返回调用结果。其中，IPC上的调用是一个同步操作，也就是说，调用者的线程会一直等到调用结束，才会开始后续的处理；而网络的处理是异步的，请求发送后不需要等待应答。客户端通过Java的wait()/notify()机制简单地解决了异步的网络处理和同步的IPC调用的差异。

### 4.6.4 服务器端方法调用过程

Hadoop IPC客户端的请求发送到服务器以后，首先由Listener接收。

#### 1. Listener的工作过程

Listener主要运行NIO选择器循环，并在Listener.doRead()方法中读取数据，在Connection.readAndProcess()中恢复数据帧，然后调用processData()。如图4-20所示。

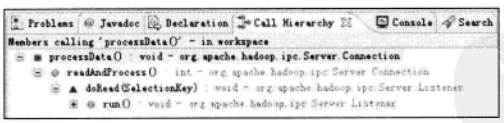



图4-20 Listener的工作过程

processData()处理一帧数据，它从数据帧中获取调用的标识符，并反序列化调用参数，构造服务器端Call对象，然后将该对象放入callQueue队列中。代码如下：

```
private void processData() throws IOException, InterruptedException {
   DataInputStream dis =
      new DataInputStream(new ByteArrayInputStream(data.array()));
   int id = dis.readInt();// 调用标识符
   ......
```

```
        Writable param = ReflectionUtils.newInstance(paramClass, conf);
        param.readFields(dis); //RPC.Invocation 对象

        Call call = new Call(id, param, this);
        callQueue.put(call); // 将调用对象放入队列中
    }
```

阻塞队列 callQueue 定义于 Server 类中，是 Listener 和 Handler 的边界。Listener 和 Handler 是典型的生产者 - 消费者，processData() 将待处理的数据放入队列中，由 Handler（消费者）调用对应的服务器方法。

### 2. Handler 的工作过程

Handler 的主要工作都在 run() 方法中完成。在方法的主循环中，每次循环处理一个请求，请求通过调用 Server 的抽象方法 call() 完成服务端方法调用。

抽象方法 call() 一共需要三个参数，前面两个参数提供了方法调用需要的上下文信息，其中接口名称由参数 protocol 提供，在建立 IPC 连接的时候，通信的双方已经交换了这个信息。其他信息，包括方法名称、形式参数列表和实际参数列表都序列化在请求参数 param 中。

抽象方法 Server.call() 通过 Writable 返回调用的结果，或者通过 IOException 告知调用过程中发生了异常。看以下代码：

```
// 抽象的 call() 方法
public abstract Writable call(Class<?> protocol,
      Writable param, long receiveTime)throws IOException;
......
public void run() {
   ......
   ByteArrayOutputStream buf = new ByteArrayOutputStream(10240);
   while (running) {
      try {
         final Call call = callQueue.take(); // 获取一个远程调用请求
         ......
         String errorClass = null;
         String error = null;
         Writable value = null;
         ......
         try {
            // 通过 Subject.doAs() 调用方法
            value = Subject.doAs(call.connection.user,
                  new PrivilegedExceptionAction<Writable>() {
                     @Override
                     public Writable run() throws Exception {
                        // 方法调用
                        return call(call.connection.protocol,
                                    call.param, call.timestamp);
                     }
                  }
               );
```

```
                } catch (PrivilegedActionException pae) {
                    Exception e = pae.getException();
                    errorClass = e.getClass().getName();
                    error = StringUtils.stringifyException(e);
                } catch (Throwable e) {
                    errorClass = e.getClass().getName();
                    error = StringUtils.stringifyException(e);
                }
                ......
                setupResponse(buf, call,
                        (error == null) ? Status.SUCCESS : Status.ERROR,
                        value, errorClass, error);
                responder.doRespond(call);
            } catch (InterruptedException e) {
                ......
            } catch (Exception e) {
                ......
            }
        }
        ......
    }
```

上述代码中，对抽象方法 Server.call() 的调用是通过 Subject.doAs() 完成的。Subject.doAs() 是 Java 鉴权与授权服务（Java Authentication and Authorization Service，JAAS）中的方法。JAAS 包含两部分内容：

- ❑鉴权（Authentication）：对用户身份进行验证。一般来说，就是要求用户需要登录系统，如登录 Linux 主机一样。
- ❑授权（Authorization）：授权用户执行某些特定的操作，如用户是否能够删除某个目录下的文件，就是授权系统需要完成的工作。

Subject.doAs() 中的 Subject 在 JAAS 中用于表示某一项（如一个人）的一组相关信息，此类信息一般包括 Subject 的身份，以及与安全相关的属性（如密码和加密密钥）。doAs() 方法为某一项 Subject 执行相应的动作 / 代码，如果调用者，也就是 Subject 对象不具有调用此方法的权限，则会抛出 SecurityException。以上面的代码段为例，如果用户 call.connection.user 没有执行某个远程过程调用的权限，第二个 catch 语句会捕获 SecurityException 异常。至于 Subject 对象是否有调用特定方法的权限，JAAS 提供了灵活的机制，系统管理员可以根据需要，动态地调整策略。

最后要注意的是，如果 run() 调用的方法中抛出异常（非授权检查抛出的异常），那么这个异常会包裹在 PrivilegedActionException 中。Subject.doAs() 的调用者需要通过 PrivilegedActionException 的 getException() 方法得到真正发生的异常，如上述 PrivilegedActionException 捕获后的处理代码。

抽象方法 Server.call() 调用结束后，可能返回一个 Writable 对象，也可能抛出异常，抛出的异常会被捕获并被设置到相关变量中。接下来的 setupResponse() 方法（该方法很简单，不再分析）用于将返回结果序列化到 Call 的成员变量 response 中，完成发送应答前的处理。

在实际运行的 IPC 服务器中，抽象方法 Server.call() 的实现在 RPC.Server 中，代码如下。

```
private Object instance;
……
public Writable call(Class<?> protocol, Writable param,
      long receivedTime) throws IOException {
   try {
      Invocation call = (Invocation) param;
      ……
      Method method = protocol.getMethod(call.getMethodName(),
            call.getParameterClasses());
      method.setAccessible(true);
      ……
      // 在 IPC 服务器实现对象 instance 上调用相应的方法
      Object value = method.invoke(instance, call.getParameters());
      ……
      return new ObjectWritable(method.getReturnType(), value);
   } catch (InvocationTargetException e) {
      Throwable target = e.getTargetException();
      if (target instanceof IOException) {
         throw (IOException) target;
      } else {
         // 用 IOException 包装异常对象
         IOException ioe = new IOException(target.toString());
         ioe.setStackTrace(target.getStackTrace());
         throw ioe;
      }
   } catch (Throwable e) {
      ……
   }
}
```

在 RPC.Server.call() 中，传入的 param 会被强制转换成 RPC.Invocation 对象，并通过该对象包含的方法名称（getMethodName() 方法）和形式参数列表（getParameterClasses() 方法）获取 Method 对象。前面介绍 Java 动态代理的时候，曾经提过 Method 类提供了关于类或接口上某个方法以及如何访问该方法的信息，在 RPC.Server 的 call() 方法中，最终是通过 Method 类的 invoke() 方法，调用 IPC 服务器实现对象 instance 上对应的方法，完成 Hadoop 远程过程调用。

对于调用的结果，如同前面方法相关的讨论，需要分两种情况：如果调用正常结束，则将结果放入一个 ObjectWritable 对象中，并返回；否则，由于 Server.call() 方法只能抛出 IOException，必须将所有非 IOException 的异常，包装在一个新构造的 IOException 中，并抛出该新 IOException 对象。

有了对 Handler 工作过程的了解，前面 IPC 连接建立时对客户端的鉴权代码也随着清晰起来。IPC 的 TCP 连接建立后，需要进行握手和连接头检查。其中，连接头检查需要对连接进行鉴权（代码如下），鉴权失败时，会给客户端发送一个类似过程调用的假的应答，该应答的状态是致命错误（FATAL），客户端在处理致命错误时，会关闭 IPC 连接。其实，FATAL 状态只

应用于这种场景，真正处理 IPC 的时候，只使用了 ERROR 状态。相关代码如下：

```
public int readAndProcess() throws IOException, InterruptedException {
   ......
   try{// 对连接进行鉴权
      authorize(user, header);
      ......
      } catch (AuthorizationException ae) { // 鉴权失败
         authFailedCall.connection = this;
         setupResponse(authFailedResponse, authFailedCall, Status.FATAL,
               null,ae.getClass().getName(), ae.getMessage());
         responder.doRespond(authFailedCall);
         return -1;// 关闭连接
      }
   }
}
```

### 3. Responder 的工作过程

Handler 通过调用 Responder 的 doRespond() 方法，将处理完的结果交给 Responder。为什么不在处理器处理完 IPC 请求后，就将结果发送给客户端呢？一般情况下，远程过程调用使用 Handler 的线程执行被调用的过程。也就是说，Handler 所在的线程其实是共享资源，对共享资源占用的时间越短越好；另一方面，网络通信的时间是不确定的，如果客户端比较忙的话，响应时间会比较长（相对于过程调用）。综合考虑这些情况，Hadoop IPC 服务器引入了专门的应答器发送 IPC 的处理结果。

往 IPC 连接发送应答都必须通过 Handler 的 doRespond() 方法，该方法很简单，把应答对应的远程调用对象放入 IPC 连接的应答队列里。将应答放入队列后，这里做了一个特殊的处理，如果 IPC 连接的应答队列只有一个元素，那么立即调用 Handler 的 processResponse() 方法，向客户端发送结果。这是一个提高服务器性能的优化，当应答队列只有一个元素的时候，表明对应的 IPC 连接比较空闲，这时候直接调用 processResponse() 发送应答，可以避免从 Handler 的处理线程到 Responder 处理线程的切换开销。代码如下：

```
void doRespond(Call call) throws IOException {
   synchronized (call.connection.responseQueue) {
      call.connection.responseQueue.addLast(call);
      if (call.connection.responseQueue.size() == 1) { // 答队列只有一个元素
         processResponse(call.connection.responseQueue, true);
      }
   }
}
```

Responder 运行在自己的线程里，其处理线程实现了和 Listener 类似的 NIO 选择器循环，Listener 监听了 OP_READ 和 OP_ACCEPT 事件，自然，Responder 感兴趣的事件应该是 OP_WRITE，即通道可写。Responder 的选择器循环实现在 Responder.run() 中，代码如下：

```
public void run() {
   ......
```

```
        while (running) {
            try {
                waitPending();// 等待通道登记
                writeSelector.select(PURGE_INTERVAL);// 等待通道可写
                Iterator<SelectionKey> iter =
                      writeSelector.selectedKeys().iterator();
                while (iter.hasNext()) {
                    SelectionKey key = iter.next();
                    iter.remove();
                    try {
                        if (key.isValid() && key.isWritable()) {
                            doAsyncWrite(key); // 输出远程调用结果
                        }
                    } catch (IOException e) {
                        ......
                    }
                }
                ......
            } //try
            ......
        }
    }

    private void doAsyncWrite(SelectionKey key) throws IOException {
        Call call = (Call)key.attachment();
        ......
        synchronized(call.connection.responseQueue) {
            if (processResponse(call.connection.responseQueue, false)) {
                try {
                    key.interestOps(0);
                } catch (CancelledKeyException e) {
                    ......
                }
            }
        }
    }
```

其中 waitPending() 方法保证对成员变量 writeSelector 的 select() 操作和 register() 操作不会相互影响，当可以安全进行 select() 操作时，run() 主循环调用选择器的 select() 方法，并在可写的通道上调用 doAsyncWrite()，进行写操作。

成员函数 doAsyncWrite() 对输入的数据进行一些必要的检查以后，调用 processResponse() 写数据。如果 processResponse() 返回 true，表明通道上没有等待的数据，这时使用参数 0 调用 SelectionKey.interestOps() 方法，清除兴趣操作集，也就是清除对 OP_WRITE 的监听。

processResponse() 是应答器最终调用 channelWrite() 发送数据的地方。这个方法比较长，关键点有两个：

❑processResponse() 可能在 Handler 中被调用，也可能在 Responder 中被调用，需要区

别对待。该方法在 Handler 中被调用时，参数 inHandler 为 true；同时，responseQueue 中只有一个等待被发送的远程调用，如果 inHandler 为 false，则 responseQueue 可以有一个，也可以有多个未处理调用。

❑processResponse() 返回 true 时，通道上没有需要发送的数据且是在 Handler 中被调用时，不需要处理返回值。

processResponse() 方法的代码如下：

```
private boolean processResponse(LinkedList<Call> responseQueue,
      boolean inHandler) throws IOException {
   boolean error = true;
   boolean done = false; // 该通道上有没有数据要发送
   int numElements = 0;
   Call call = null;
   try {
      synchronized (responseQueue) {
         numElements = responseQueue.size();
         // 如果通道上没有可写的数据，返回
         ......
         // 获取第一个应答
         call = responseQueue.removeFirst();
         SocketChannel channel = call.connection.channel;
         ......
         // 异步写尽可能多的数据
         int numBytes = channelWrite(channel, call.response);
         if (numBytes < 0) {
            return true;
         }
         if (!call.response.hasRemaining()) {
            // 应答数据已经写完
            call.connection.decRpcCount();
            if (numElements == 1) {
               done = true;// 该通道上没有需要写的数据
            } else {
               done = false;// 通道上还有数据要发送
            }
            ......
         } else {
            // 应答数据没有写完，插入队列头，等待再次发送
            call.connection.responseQueue.addFirst(call);

            if (inHandler) {  // 我们不在 Responder 线程，在 Handler 线程
               // 更新写应答的时间
               call.timestamp = System.currentTimeMillis();

               // 成员变量 pending++，该变量表示现在有多少个线程在进行通道注册
               incPending();
               try {
                  // 唤醒在 select() 方法上等待的 Responder 线程，
```

```
                    // 这样我们才能够调用 channel.register() 方法
                    writeSelector.wakeup();
                    channel.register(writeSelector,
                                     SelectionKey.OP_WRITE,
                                     call);
                } catch (ClosedChannelException e) {
                    //ClosedChannelException 异常，通道可能在其他地方被关闭了
                    done = true;
                } finally {
                    // 成员变量 pending--，该变量表示现在有多少个线程在进行通道注册
                    decPending();
                }
            }
            ......
        }
        error = false;// 没有错误
    }
} finally {
    if (error && call != null) {
        // 出错，关闭连接，连接关闭后当然通道上也就没有数据可写了。
        done = true;
        closeConnection(call.connection);.
    }
}
return done;
}

private synchronized void incPending() {
    pending++;
}

private synchronized void decPending() {
    pending--;
    notify();
}

private synchronized void waitPending() throws InterruptedException {
    while (pending > 0) {
        wait();
    }
}
```

以上方法中使用的变量主要有返回值 done，发送队列长度 numElements 和当前处理应答 call，它是队列的第一个元素。在对变量做一些必要的检查后，processResponse() 调用 Server 的静态辅助方法 channelWrite()，往通道中写入尽可能多的数据。如果这次写操作成功写完所有数据（缓冲区的 hasRemaining() 方法返回 false），则根据队列长度设置返回值 done。如果写操作没有完成，则将当前处理应答 call 重新放入队列头，等待通道再次可用时，继续写出数据。

如果 processResponse() 是在 Handler 中被调用，同时写操作没有完成，那么必然还没有写出去的数据必须由应答器继续处理，这时如何通知应答器呢？

首先调用 incPending() 方法，然后通过选择器的 wakeup() 方法唤醒可能处于 select 中等待的 Responder 选择器，并将通道注册到该选择器上（当前 call 作为 SelectionKey 附件）。这样，应答器就会开始监听连接对应的通道的可写事件，并在通道可写的时候发送数据。如果连接上的所有数据都写完了，processResponse() 会返回 true，并在 doAsyncWrite() 中清除对 OP_WRITE 事件的监听。Handler 中调用 processResponse() 的最后一步处理是在完成通道注册后调用 decPending()，这样可能导致应答器的主线程在 waitPending() 中返回并进入 select() 方法。正是 incPending()、decPending() 和 waitPending() 的相互配合，保证了多线程共享资源 writeSelector 的互斥访问。

前面已经介绍了 Responder 的 run() 循环的上半段代码，循环的下半段代码如下：

```
public void run() {
   ......
   while (running) {
      try {
         ......
         long now = System.currentTimeMillis();
         if (now < lastPurgeTime + PURGE_INTERVAL) {
            continue;
         }
         lastPurgeTime = now;
         // 已经超过了清理时间，可以检查未处理的 call
         ArrayList<Call> calls;

         // 通过 SelectionKey 附件，从 writeSelector 中获得注册过的（未关闭）连接
         synchronized (writeSelector.keys()) {
            calls = new ArrayList<Call>(writeSelector.keys().size());
            iter = writeSelector.keys().iterator();
            while (iter.hasNext()) {
               SelectionKey key = iter.next();
               Call call = (Call)key.attachment();
               if (call!= null&&key.channel()== call.connection.channel)
               {
                  calls.add(call);
               }
            }
         }

         for(Call call : calls) {
            try {
               doPurge(call, now);// 调用 doPurge 进行清理
            } catch (IOException e) {
               ......
            }
         }
```

```
            } catch (OutOfMemoryError e) {
                ......
            } catch (Exception e) {
                ......
            }
        }
    }

    private void doPurge(Call call, long now) throws IOException {
        LinkedList<Call> responseQueue = call.connection.responseQueue;
        synchronized (responseQueue) {
            Iterator<Call> iter = responseQueue.listIterator(0);
            while (iter.hasNext()) {
                call = iter.next();
                if (now > call.timestamp + PURGE_INTERVAL) {
                    closeConnection(call.connection);
                    break;
                }
            }
        }
    }
```

清理工作的时间间隔是 15 分钟（不可配置），也就是说，如果某个连接上有 15 分钟前的应答未发送，这个连接将被清除。

在对检查间隔进行确认以后，应答器通过选择器的 keys() 方法，获取所有注册过的 SelectionKey，并通过选择器注册标记附件获得 Call（为了顺藤摸瓜得到相应的 IPC 连接），然后在这些 Server.Call 上调用 doPurge() 方法。doPurge() 方法遍历连接上所有的调用，如果发现调用的处理时间戳（注意，这时该调用可能还未进入处理器处理，也可能是已经处理过的调用，但结果还没发送出去）和当前时间差大于 15 分钟，则调用 closeConnection() 方法，直接关闭连接。

Responder 是 IPC 服务器中比较特殊的设计，它提高了 IPC 服务器的处理能力和网络的使用率，同时又提供了异步操作中必需的超时检查，是一个非常有针对性的优化。

Responder 和其他 Server 处理器，包括 Listener 和 Handler 一起配合，完成 IPC 的服务器端处理：Listener 接受连接请求和接收数据；Handler 完成实际的过程调用，并在通道空闲时发送应答；Responder 处理网络忙时的应答发送，并对连接进行必要的超时检查。IPC 服务器处理器间分工明确并相互配合，是一个高性能 TCP 服务器的典型例子。

## 4.7　Hadoop IPC 上的其他辅助过程

通过上面两小节对 IPC 连接和 IPC 方法调用相关过程的介绍，读者应该基本了解了过程调用发生后的相关处理过程，这一小节继续介绍 Hadoop IPC 的一些辅助过程。

### 4.7.1 RPC.getProxy() 和 RPC.stopProxy()

在 IPC 发生之前，客户端需要通过 RPC.getProxy() 获得一个 IPC 接口实例，当不需要该接口实例时，必须通过 RPC.stopProxy() 方法释放相关的资源，如以下代码所示：

```
// 为 RPC.getProxy 准备 Socket 地址
InetSocketAddress addr=new InetSocketAddress("localhost",
      IPCQueryServer.IPC_PORT);

// 通过创建 IPC 客户端，获得接口 IPCQueryStatus 的一个实例
IPCQueryStatus query=
      (IPCQueryStatus) RPC.getProxy(IPCQueryStatus.class, // 接口的类对象
                                    IPCQueryServer.IPC_VER,// 接口版本
                                    addr,// 服务器地址
                                    new Configuration());
.....
// 停止客户端
RPC.stopProxy(query);
```

在 RPC.java 中一共有 3 种 RPC.getProxy() 方法，其中，最复杂也正实现该方法的是具有 6 个参数的 getProxy() 方法。这些参数的含义是：

- ❑protocol：IPC 接口对应的 Class 对象，一般通过接口的静态成员 class 获得，如 IPCQueryStatus.class。
- ❑clientVersion：IPC 客户端要求的 IPC 接口版本号，作用类型于 Java 序列化框架中的序列版本，由于 IPC 接口会不断演化，接口版本号用于保证 IPC 客户端使用的是它期望的那个版本的服务器的服务。
- ❑addr：IPC 服务器的地址，包括服务器 IP 地址和端口信息。
- ❑ticket：用户信息，主要包含了用户的鉴权信息。
- ❑conf：IPC 客户端可能使用的配置参数。
- ❑factory：创建套接字时使用的套接字工厂。套接字工厂可用于创建应用了不同策略的套接字，如带有安全性或支持防火墙隧道的套接字。

通过 SocketFactory 的静态方法 getDefault() 可以获取系统默认的套接字工厂，相关代码如下：

```
public static VersionedProtocol getProxy(Class<?> protocol,
      long clientVersion, InetSocketAddress addr,
      UserGroupInformation ticket, Configuration conf,
      SocketFactory factory) throws IOException {
   VersionedProtocol proxy = (VersionedProtocol) Proxy.newProxyInstance(
         protocol.getClassLoader(), new Class[] { protocol },
         new Invoker(addr, ticket, conf, factory));

   long serverVersion = proxy.getProtocolVersion(protocol.getName(),
         clientVersion);
```

```
    if (serverVersion == clientVersion) {
      return proxy;
    } else {
      throw new VersionMismatch(protocol.getName(), clientVersion,
                                serverVersion);
    }
  }
```

getProxy() 首先通过 java.lang.reflect.Proxy 的 newProxyInstance() 方法，直接获得动态代理类的一个实例，其中，调用句柄使用的是 RPC.Invoker 类，它的构造函数会试图复用或者创建一个 Client 对象，相关内容请参考客户端方法调用的相关分析。

前面提到，Hadoop IPC 接口必须继承自 VersionedProtocol 接口，该接口包含了 getProtocolVersion() 方法，客户端调用该方法，可以获得服务器上实现的接口版本。在前面的例子中，并没有显式调用该方法的代码，版本检查其实在 getProxy() 中实现，即在获得动态代理后，getProxy() 会立即调用服务器端的 getProtocolVersion() 方法。

getProtocolVersion() 需要接口的规范名和期望版本号作为参数，服务器端会返回一个版本号，为什么需要给服务器提供规范名和期望版本号呢？

下面来自 HDFS 名字节点的代码说明了为什么需要给出规范名。由于 NameNode 实现了多个 IPC 接口，所以必须根据客户端发送过来的不同接口规范名确定返回的版本号，说明如下：

```
public class NameNode implements ClientProtocol, DatanodeProtocol,
                          NamenodeProtocol, FSConstants,
                          RefreshAuthorizationPolicyProtocol {
  ......
  public long getProtocolVersion(String protocol,
                                 long clientVersion) throws IOException {
    if (protocol.equals(ClientProtocol.class.getName())) {
      return ClientProtocol.versionID;
    } else if (protocol.equals(DatanodeProtocol.class.getName())){
      return DatanodeProtocol.versionID;
    } else if (protocol.equals(NamenodeProtocol.class.getName())){
      ......
    }
  }
  ......
}
```

在上面的例子中，并没有用到客户端发送过来的 clientVersion 参数，但是，如果服务器可以支持多个版本，就需要根据 clientVersion 参数，返回相应的值。如下面代码展示的场景，服务器支持两个版本——ClientProtocol.versionID 和之前的一个版本：

```
public long getProtocolVersion(String protocol,
                               long clientVersion) throws IOException {
  if (clientVersion== ClientProtocol.versionID-1) {
    return clientVersion;
  }
  return ClientProtocol.versionID;
```

```
}
```

与 getProxy() 相比，stopProxy() 相对简单，代码如下：

```
public static void stopProxy(VersionedProtocol proxy) {
   if (proxy != null) {
      ((Invoker) Proxy.getInvocationHandler(proxy)).close();
   }
}
```

下面的代码是通过获得动态代理中绑定的 RPC.Invoker 对象并调用它的 close() 方法，进而关闭对应的客户端：

```
private static class Invoker implements InvocationHandler {
   ......
   synchronized private void close() {
      if (!isClosed) {
         isClosed = true;
         CLIENTS.stopClient(client);
      }
   }
}
```

## 4.7.2 RPC.getServer() 和 Server 的启停

客户端通过 RPC.getProxy() 获得 IPC 接口实例，通过 RPC.stopProxy() 释放接口实例。和客户端不同，服务器通过 RPC.getServer() 获得服务器对象之后，需要显式调用服务器对象的 start() 方法启动服务器，停止服务器则通过对应的 stop() 方法，实例中的相关代码如下：

```
// 远程过程调用的实际调用目标
IPCQueryStatusImpl queryService=new IPCQueryStatusImpl();

// 在对象 queryService 上获得一个 IPC 服务器
Server server = RPC.getServer(queryService,"0.0.0.0",IPC_PORT,
      new Configuration());

// 服务器开始服务
server.start();
......
// 停止服务器
server.stop();
```

RPC.getServer() 有两个形式，其中参数较少的形式会提供一些默认值，并调用另一种形式；第二种形式有 6 个参数，这些参数的具体意义见下面代码中的注释。需要说明的是，bindAddress 用于指定服务器绑定的 IP 地址，在多 IP 的主机上，可以将 IPC 服务器绑定在某些特定的 IP 地址上；numHandlers 用于指定服务器的 Handler 处理器的线程数。

在一个 IPC 服务器里，Listener 和 Responder 只使用一个线程，但 Handler 会启动多个线程，这些线程作为消费者，在请求队列 callQueue 上等待并处理 Listener 生产的调用请求。代码如下：

```
public static Server getServer(final Object instance,
      final String bindAddress, final int port, Configuration conf)
         throws IOException {
   return getServer(instance, bindAddress, port, 1, false, conf);
}

/** 获取一个RPC服务器
 * @param instance IPC方法调用的目标
 * @param bindAddress 服务器绑定的IP地址
 * @param port 监听端口
 * @param numHandlers 处理器线程的个数
 * @param verbose 是否打开调用方法日志
 * @param conf 配置Server对象的配置参数
 */
public static Server getServer(final Object instance,
      final String bindAddress,final int port,final int numHandlers,
      final boolean verbose, Configuration conf) throws IOException {
   return new Server(instance, conf, bindAddress,
                     port, numHandlers, verbose);
}

public Server(Object instance, Configuration conf,
      String bindAddress, int port) throws IOException {
   this(instance, conf,  bindAddress, port, 1, false);
}

public Server(Object instance, Configuration conf, String bindAddress,
      int port,int numHandlers,boolean verbose) throws IOException
{
   super(bindAddress, port, Invocation.class, numHandlers, conf,
         classNameBase(instance.getClass().getName()));
   this.instance = instance;
   this.verbose = verbose;
   this.authorize = conf.getBoolean(
         ServiceAuthorizationManager.SERVICE_AUTHORIZATION_CONFIG,
         false);
}
```

RPC.Server 的构造方法在调用了父类的构造方法后，简单地初始化了一些特有的成员变量。RPC.Server 继承自 org.apache.hadoop.ipc.Server，其构造方法相对复杂。同样，Server 的构造方法也需要初始化一系列变量，有的通过输入的参数初始化，有的则通过配置系统，如连接空闲时间 maxIdleTime、空闲连接清理时一次最多清理连接数 maxConnectionsToNuke（一次处理过多的连接会占用太多的资源）、thresholdIdleConnections（连接数过了此阈值才会进行空闲连接清理）和 tcpNoDelay（是否禁用 TCP 的 Nagle 算法）等；同时，服务器运行时需要的 Listener 和 Responder 对象，也在构造的时候就创建了。代码如下：

```
protected Server(String bindAddress, int port,
      Class<? extends Writable> paramClass, int handlerCount,
      Configuration conf, String serverName) throws IOException
```

```
{
    ......
    this.paramClass = paramClass;
    this.handlerCount = handlerCount;
    this.socketSendBufferSize = 0;
    this.maxQueueSize = handlerCount * MAX_QUEUE_SIZE_PER_HANDLER;
    this.callQueue = new LinkedBlockingQueue<Call>(maxQueueSize);
    this.maxIdleTime =
          2*conf.getInt("ipc.client.connection.maxidletime", 1000);
    this.maxConnectionsToNuke = conf.getInt("ipc.client.kill.max", 10);
    this.thresholdIdleConnections
          = conf.getInt("ipc.client.idlethreshold", 4000);

    // 创建 Listener
    listener = new Listener();
    ......
    this.tcpNoDelay = conf.getBoolean("ipc.server.tcpnodelay", false);

    // 创建 Responder
    responder = new Responder();
}
```

RPC.Server 并没有覆盖父类的启动方法 Server.start()，从下面的代码可见，服务器 start() 方法调用后，所有的线程才开始工作：

```
public synchronized void start() throws IOException {
    responder.start();
    listener.start();
    handlers = new Handler[handlerCount];

    for (int i = 0; i < handlerCount; i++) {
        handlers[i] = new Handler(i);
        handlers[i].start();
    }
}
```

和启动服务器对应的是关闭服务器方法 Server.stop()，该方法先设置标志位 running 为 false，然后通过线程的 interrupt() 方法中断线程。监听器的线程还需要做些清理工作，包括唤醒选择器和关闭监听的 Socket 通道（在 Listener.doStop() 中实现）。实现代码如下：

```
public synchronized void stop() {
    LOG.info("Stopping server on " + port);
    running = false;
    if (handlers != null) {
        for (int i = 0; i < handlerCount; i++) {
            if (handlers[i] != null) {
                handlers[i].interrupt();
            }
        }
    }
    listener.interrupt();
```

```
        listener.doStop();
        responder.interrupt();
        notifyAll();
        ......
    }
```

## 4.8　小结

本节在介绍了远程过程调用、Java 动态代理和 Java NIO 的大量背景知识基础上，分析了 Hadoop 远程过程调用的实现。

远程过程调用自从 1984 年引入分布式计算领域以后，得到了广泛的研究和应用，涌现出如 XML-RPC、Java RMI、CORBA 和 Web Service 等相关技术。这些技术的具体实现，一般都使用客户存根和服务器骨架来简化开发流程，使得在客户端访问远程过程和本地过程调用有着类似的外观，而服务器上的被调用代码又如同传统调用。和上一章讨论的序列化一样，Hadoop 也没有使用 Java 的远程方法调用，而是在 Java 提供的一些工具上实现了自己的 IPC 机制。主要的工具包括用于实现接口的动态代理和用于调用转发的调用句柄，同时，由于 IPC 服务器的高负载，我们也简要介绍了 Hadoop IPC 使用的具有大量非阻塞特征的 Java NIO 系统。

Hadoop IPC 实现的代码并不多，主要过程分为两大类：IPC 连接相关和 IPC 方法调用相关。其中，IPC 连接是客户端和服务器关系的一个抽象，包括连接建立、连接上的数据读写、连接维护和连接关闭 4 个流程。IPC 方法调用在客户端只是一个比较复杂的、应用 Java 动态代理的实现，而在客户端，则是由 Listener、Handler 和 Responder 配合，完成请求读取、请求处理和请求应答等步骤。正是通过 IPC 连接和方法调用相互配合，Hadoop 实现了一个简单、但又能精确控制通信细节的客户存根和服务器骨架，为 Hadoop 其他组件相互通信、配合工作提供了坚实基础。

# 第 5 章　Hadoop 文件系统

文件系统是计算机系统处理信息的一种重要方法。从 20 世纪 60 年代开始，文件系统就一直在学界和业界得到广泛的研究、开发和应用，经历了计算机发展的各个时代，并得到长足的发展。Hadoop 文件系统，包括 Hadoop 抽象文件系统以及基于该抽象文件系统的大量具体文件系统，满足了构建在 Hadoop 上的各类应用的各种数据访问需求，是文件系统发展的新阶段。

## 5.1　文件系统

早在 1965 年开发的 Multics（UNIX 的前身）就详细地设计了文件系统，这使得文件系统成为多用户单节点操作系统的重要组成部分。最初的文件系统用于解决信息的长期存储，并达到如下要求：

- ❑能够存储大量的信息。
- ❑使用信息的应用终止时，信息必须保存下来。
- ❑多个应用可以并发地存储信息。

解决这些问题的通常方法，是把信息以一种单元，即“文件”（file）的形式存储在磁盘或者其他外部介质上，一个文件是一个命名的、存储在设备上的信息的线性字节流。文件在需要的时候可以读取这些信息或者写入新的信息。存储在文件中的信息必须是永久的，也就是说，它不会因为应用的创建 / 终止受到影响。只有当用户显式地删除它时，文件才会消失。对文件的管理，包括了文件的结构以及命名、存取、使用、保护和实现，称为文件系统。

从用户的观点来看，文件系统中最重要的方面是文件系统如何呈现在他们面前。即一个文件由什么组成、文件如何命名、如何保护文件，以及对文件可以进行哪些操作等。而对文件系统的设计者来说，他们还需要关注如何记录文件的相关信息、如何组织存储区等问题。

### 5.1.1　文件系统的用户界面

下面详细讨论文件系统的用户界面，也就是从用户的角度来研究文件，即文件是怎么使用的、有些什么特征。

文件作为一种抽象机制，最重要的特征就是命名方法。各种系统的文件命名规则略有不同，但一般都支持由一定长度字符串作为文件名。应用创建文件的时候要指定文件名，文件在应用结束后仍然存在，其他应用可以使用该文件名对它进行操作。大部分的文件系统不关心文件里保存的数据，把文件内容作为无结构的字节序列保存。也有一些文件系统支持结构化文件，以记录为单位组织信息，免去文件系统使用者将“原始的”字节流转换成记录流的

麻烦。除了文件名和数据，文件系统会赋予文件其他的信息，比如文件的创建日期、文件长度、创建信息、引用计数等。我们把这些额外的项称为文件属性，有时也称为元数据。

**注意**　本书不讨论文件系统中的特殊设备文件、硬链接文件、符号链接文件等，忽略文件系统中对这些功能的支持。本书提到的文件都是常规文件。

### 1. 常用文件操作

大部分文件系统支持的与文件相关的常用系统调用如下：

- ❑创建文件：创建没有任何数据的文件，该调用的目的是声明文件存在，并且设置一些属性。
- ❑删除文件：当不再需要文件时，必须删除它以释放存储空间。
- ❑打开文件：在使用文件之前要先打开文件，打开文件后，文件系统会在内存中保留和打开文件相关的一些信息，有时还会创建对文件存取的相关机制。
- ❑关闭文件：当存取结束后，通过关闭文件，可以把在内存中保留的一些信息写回文件系统，并释放打开文件需要占用的资源。
- ❑读文件：从文件中读取数据，一般来说，该操作从当前位置的文件中将数据读取到一个缓冲区中。
- ❑写文件：往文件中写入数据。
- ❑追加数据：该方法是写文件的限制形式，它只能在文件末尾添加数据。
- ❑读记录：在结构化文件中读取记录。
- ❑写记录：在结构化文件中写入记录。
- ❑删除记录：在结构化文件中删除记录。
- ❑移动当前位置：当需要指定从哪里读写数据的时候，使用该调用可以把当前位置移动到文件中的特定位置。
- ❑获取文件属性：读取文件的某个属性，如文件的最后修改时间。
- ❑设置文件属性：有些属性是用户可以修改的，在文件创建之后，能够修改它们，设置文件属性就是用于这个目的。
- ❑文件改名：用户改变现有文件的文件名。

### 2. 目录与目录树

目录是文件系统中另一个重要的概念。一个“目录”或“文件夹”就是一个虚拟容器，它里面保存着一组文件和其他一些目录。一个典型的文件系统可能会包含成千上万的目录，多个文件通过存储在一个目录中，可以达到有组织地存储文件的目的。同时，在一个目录中可以保持另外的目录（称为子目录），这些目录和文件构成了一个层次结构（即目录树），如图 5-1 所示。

使用目录树组织文件系统时，需要通过某种方法指明文件名。路径名描述了怎样在一个文件系统中确认一个文件的位置，它是一个分量名序列，各分量名之间用分隔符（在 Linux

和 Hadoop 分布式文件系统中，分隔符是斜杠符）隔开。分量名是一个字符序列，它指明一个被唯一地包含在前缀（父目录）分量中的名字。一个完整的路径名由一个分隔符（为了方便讨论，后面采用斜杠符“/”作为分隔符）开始，并且指明一个文件，这个文件可以从文件系统的根（没有祖先的目录）开始，沿着该路径名的后继分量名所在的那个分支游历文件树而找到。因此，路径名“/etc/passwd”和“/usr/src/cmd/date.c”都是图 5-1 树中文件。

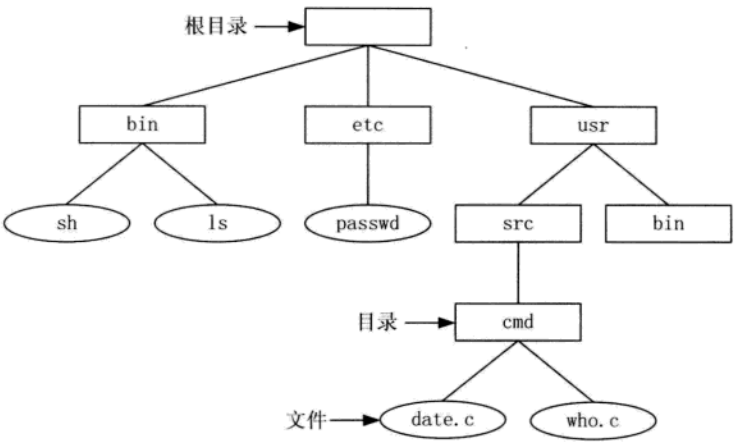



图 5-1 文件目录树示例

以上提到的两个路径名都是从根目录开始，由根目录到文件的路径组成，这样的路径是绝对路径。相对路径是另外一种指明文件的方法，需要和工作目录（也称当前目录）一起使用。用户可以指定一个目录作为当前的工作目录。这时，所有的路径名如果不是从根目录开始，都是相对于工作目录的。例如，当前工作目录是“/usr/alice”，那么绝对路径为“/usr/alice/mailbox”的文件，可以简单地用“mailbox”引用。

文件系统中用于管理目录的系统调用差别比较大，下面是 UNIX 系统的例子：

- ❑创建目录：创建目录，新创建的目录会包含两个特殊的目录项，其中“.”指当前目录，“..”指向父目录。
- ❑删除目录：删除目录。
- ❑打开目录：目录内容可以被读取。例如，为了列出目录中的所有文件 / 目录，程序必须先打开该目录，让后读取其中所有文件 / 目录项。
- ❑关闭目录：读目录结束以后，应该关闭被打开的目录。
- ❑读目录：返回打开目录的下一个目录项。
- ❑目录改名：在很多方面，目录和文件相似。文件可以换名，目录也一样。

### 3. 存储媒体上的文件系统

现代计算机系统常常配置有多个存储设备，并且大容量磁盘常常被分成逻辑上不同的存储设备，文件管理器可以实现存储在不同设备上的目录层次，并将这些不同的层次组合成单

个系统层次，如软盘，CD-ROM 等可移动媒体上的信息就被组织成单根的树，通过根就可以遍历这个层次结构。上述存储媒体上的具有单个根目录的文件层次集合也称为文件系统，文件系统是一个非常有用的管理单元。注意，这里的“文件系统”和本章讨论的 Hadoop 文件系统，意义不相同，请读者区分。

UNIX 系统中提供的文件系统操作包括：

- 加载文件系统：将一个文件系统添加到一个已经存在的目录中，被安装的文件系统的根取代基文件系统的目录。
- 卸载文件系统：将某一个已经加载的文件系统卸载。

加载文件系统操作，就是将某个文件系统“嫁接”到另一个文件系统的目录层次中。例如，有两个不同的文件系统，分别是文件系统 1 和文件系统 2，它们是存储在不同介质上（如两个不同的磁盘）的单根目录树。将文件系统 2 加载到文件系统 1 的“/home”目录下，就可以通过“/home/user1”访问文件系统 2 原有目录结构下“/user1”的目录内容了，如图 5-2 所示。

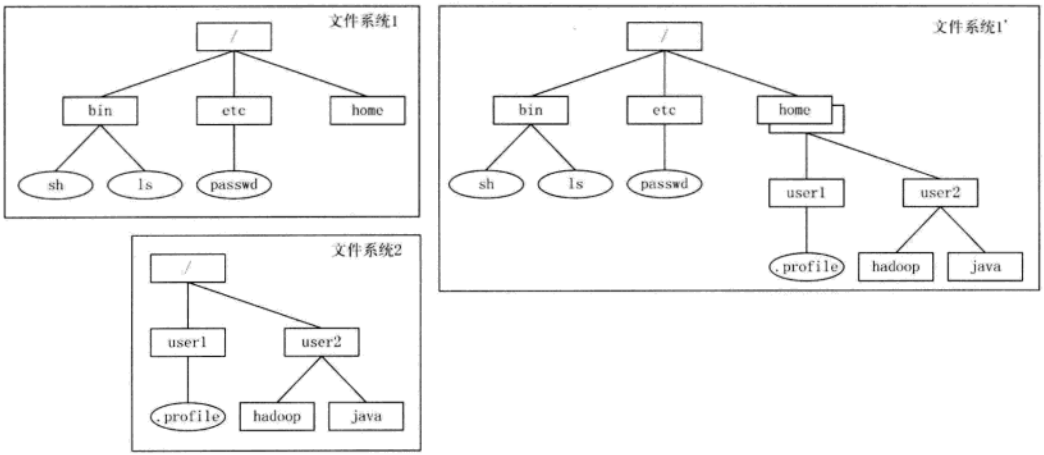



图 5-2　具有单个根目录的文件系统

## 5.1.2　文件系统的实现

现在，让我们从用户角度转到实现者角度来研究文件系统。用户关心的是文件是如何命名的，可以进行哪些操作，目录树是什么样的以及类似的问题。而实现者感兴趣的是文件和目录是如何存储的，存储空间是怎样管理的和系统是如何有效而可靠地工作。

### 1. 块管理

文件的物理结构指文件在存储设备（如磁带、磁盘、光盘）上的存取方式。为了便于管理，设备往往将存储空间组织成为具有一定结构的存储单位。

以磁盘为例，磁盘在逻辑上会划分为磁道、柱面和扇区，扇区是磁盘的读写单位，也是磁

盘读写时候的最小寻址单位，一个扇区一般是 512 字节（自 2009 年底开始，磁盘制造商开始引入使用 4096 字节扇区的磁盘）。讨论文件的物理结构时，需要考虑存储设备的存储结构。

块管理用于记录存储块和文件的关联关系，对于随机存储设备（如磁盘）而言，一般有如下三种方法来实现块管理。

（1）连续分配

最简单的物理结构是连续分配，连续分配将文件中的 $N$ 个逻辑块映射到 $N$ 个地址连续的物理块上。以磁盘为例，如果扇区的大小是 512 字节，50KB 的文件需要分配连续的 100 个扇区。这种方案简单、性能好，允许驱动器花较少的时间对整个文件进行读取和写入。

不足之处也很明显，首先很难预先知道文件的大小，文件系统也就无法确定要保留多少存储空间；其次，这种分配方案很容易产生存储空间碎片，原本可以使用的存储空间被浪费了，同时，只要文件系统打算给一个文件分配 $N$ 块，则存储设备中必须找到 $N$ 个连续的物理块。

（2）链接表

存储文件的第二种方式是为每一个文件构造存储空间的链接表，在每个存储单元的特定部位，保存下一个存储单元的位置，每个逻辑块都包含一个带连接的头，用来指向下一个逻辑块对应的物理块的地址。和连续分配方案不同，这种方法能利用系统的每一个磁盘块，同时，目录项中只需要保存第一个存储单元的地址，文件的其他块可以根据这个地址来查找。链接分配方案顺序读取文件非常方便，但是随机存储效率比较低。如果应用需要读取文件块 3 中的内容，它必须依次读取前面的 3 个文件块，才能定位到文件块 3，如图 5-3 所示。

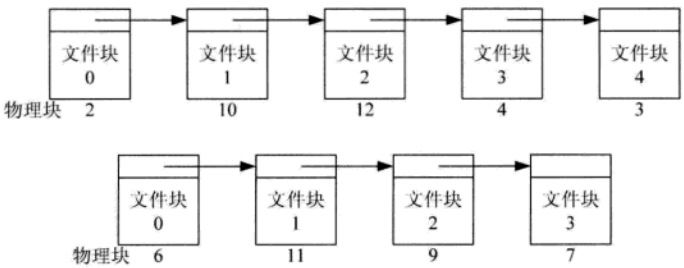



图 5-3 按磁盘块链接表来存储文件

（3）索引链式表

为了克服链接表分配的不足，取出每个磁盘块的指针字段，把它放到一张索引表中，就形成了使用索引链式表分配。如图 5-4 所示，表的序号为物理块号，每个表项存放链接指针，指向属于同一文件的下一物理块，文件的第一块地址存放在目录项中，计算机引导以后，这张表会保持在内存中，不需要访问磁盘。该方法的缺点是需要把整个链表存放在内存中。MS-DOS 采用的就是这种文件物理结构。

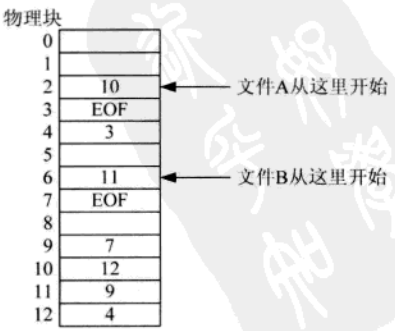



图 5-4 按索引链式表存储文件

### 2. 目录管理

目录作为文件和子目录的容器，其数据由一组结构化的记录组成，每个记录描述了集合中的一个文件或者子目录。记录提供足够的信息，允许文件管理器确定文件的所有已知特征，例如，名字、长度、建立时间、最后访问时间、所有者等。目录的实现，通常是在存储设备中，将目录作为特殊文件（往往是结构化文件，支持基于记录的操作）保存，或保存在某一个特定的存储区域中。以 MS-DOS 目录项为例，它的目录项有 32 字节，如图 5-5 所示。

目录项中包含了文件名、扩展名等属性，特别需要注意的是，通过第一块块号，以及前面介绍的索引链式表，可以找到文件的所有块。

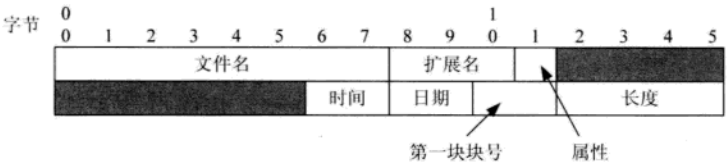



图 5-5　MS-DOS 目录项

### 3. 存储媒体上的文件系统管理

计算机启动的时候，需要标识系统引导盘，获得主引导记录，在某些系统上（如 Linux），还需要定义包含根目录的逻辑设备。同时，当可移动媒体被连接到相应的存储设备时，它的文件系统被“安装”到某一个目录层次中。这些操作都是针对存储媒体上的文件系统的操作，涉及了文件系统的物理结构。

存储媒体上的文件系统的物理结构和目录类似，是保存在某一位置的一个结构化记录。以 CD-ROM 使用的 ISO 9660 文件系统为例，该文件系统存放在卷（volume）中，其信息保存在一个 2048 字节的块中，包含了如文件系统魔数、版本号、系统标识等信息。文件管理器通过这些信息，使用文件管理器的 ISO 9660 处理子系统，就可以访问 CD-ROM 上的文件、目录。

### 4. 存储空间管理

在传统文件系统的物理实现中，最后一个重要问题是存储空间管理，也就是对存储介质上未使用空间的管理，当创建文件、写入文件或删除文件时，都涉及存储空间的管理。目前，存储空间管理主要的方法有空闲文件目录、空闲块链、位示图等。以空闲块链为例，当文件系统采用链式表进行块管理时，可以将空闲的块组成空闲块链，保存在一个特殊的文件中。

由于在 Hadoop 具体文件系统的实现，特别是 HDFS 的实现中存储空间管理的特殊性，就不再对这个问题进行深入讨论了。

## 5.1.3　文件系统的保护控制

文件系统解决了信息的长期存储和多个应用存储、访问信息的问题，但也带来了文件的共享、保护和保密三个不同而又相互联系的问题。

文件的共享是指不同的用户共同使用一个文件；文件的保护指文件本身需要防止文件的拥有者或其他用户破坏文件内容；文件保密指未经用户拥有者允许，任何用户不得访问该文件。这三个问题实际上是一个用户对文件的使用权限，即读、写、执行的许可权问题。每次访问被保护资源时，系统通过存储控制验证模块做出相应的决定，这个过程一般包括如下 3 步：

1）审定用户的存取权限；

2）比较用户权限和本次存取要求是否一致；

3）将存取要求和被访问文件的保密性比较，看是否有冲突。

用于验证用户存取操作一般有 4 个方式：口令、密码、存取控制矩阵、存取控制表和权限表。

### 1. 口令

用户创建文件时，可以为文件提供口令，并记录在文件说明信息中。当任意一个用户想使用该文件时，都必须首先提供口令，只有用户提供的口令和文件说明信息中的口令一致的时候，才允许存取。显然，口令只有设置者自己知道，若允许其他用户使用自己的文件，口令设置者可以将口令赋予其他用户。

通过这样的方法，可以简单地实现文件的共享和保密，而且口令较为简单，占用的内存单元以及验证口令所费时间都将非常少。不过，相对来说，口令方式保密性较差，口令一旦被人掌握，就可以获得和文件拥有者同样的权利而没有任何等级差别。同时，修改文件的口令，也需要通知所有授权用户。

### 2. 密码

防止文件泄密以及控制存取访问的另一种方法是密码。密码方式是指在用户创建源文件、写入存储设备前，通过特定的算法和加密密钥对文件内容进行编码加密，读取文件时，必须提供相应的解密密钥，进行解密。只有用户给定的密钥和加密时的密钥一致，才能读取被加密的文件。

和口令不同，加密方式具有保密性强的优点，同时密钥不需要保存在文件系统中，而是由用户自己掌握。但是，由于编码解码工作需要耗费大量的处理时间，因此，加密技术是以牺牲系统开销为代价的。口令方式中有时也采用一定的加密技术，加密口令方式可以对口令本身进行加密，加密后的口令仍然存放在文件说明信息中，但加密口令方式不对文件内容进行加密，保密性弱。

### 3. 存取控制矩阵

存取控制矩阵以二维矩阵的方式进行存储控制，其中一维是所有用户，另一维是所有文件，矩阵的每一个元素都表示该元素对应的用户对文件的权限，如表 5-1 所示。其中，读、写、执行分别表示为 R、W 和 X。

表 5-1　存取控制矩阵

| | 用户 / 文件 | Alice | Bin | Bob | Tom | …… | …… |
|---|---|---|---|---|---|---|---|
| 1 | /bin/datetime | RWX | X | RWX | X | | |
| 2 | /bin/showdata | RW | | | X | | |
| 3 | /home/alice/bin/lc | | R | WX | | | |
| 4 | /home/bin/README | | RW | | | | |

存储控制验证模块在处理用户的存取需求时，通过查找矩阵中该用户和该文件所对应元素的值，当该元素表明有访问许可的时候，才能进行具体的访问。如第 1 行，Alice 的权限为 RWX，表明 Alice 可以对文件“/bin/datetime”进行读、写和执行操作；用户 Bin 只拥有权限 X，所以 Bin 只能执行文件“/bin/datetime”，不能进行其他操作。

存取控制矩阵是一种很自然的方法，可以对系统中所有文件的存取权限进行完整有效的控制。但是当系统中用户和文件数较多时，存储和管理这样的矩阵需要比较大的系统开销，用户访问一个文件，都需要对矩阵的一行进行搜索，访问效率很低。

**4. 存取控制表和权限表**

实际上我们很少存储像表 5-1 那样的矩阵，这个矩阵非常大，并且有很多空项，大多数用户只存取很少的文件，因此，可以使用两种方法：按行或按列来存储矩阵，而且只存取非空元素。

第一种存储技术按行来保存，每一个文件被赋予一张排序的表，其中列出可以访问该文件的用户，以及如何进行访问，这种类型表称为存取控制表，表 5-2 给出了表 5-1 对应的存取控制表。

表 5-2　存取控制表

| 文件 | 访问控制表 |
|---|---|
| /bin/datetime | (Alice, RWX), (Bin, X), (Bob, RWX), (Tom, X) |
| /bin/showdata | (Alice, RW), (Tom, X) |
| /home/alice/bin/lc | (Bin, R), (Bob, WX) |
| /home/bin/README | (Bin, RW) |

权限表中每个访问控制表（Access Control List，ACL）都指定了用户对文件允许的访问（读、写、执行）。如文件“/bin/showdata”，Alice 具有读、写权限，而 Tom 只能执行该文件，其他用户都不能访问该文件。UNIX/Linux 系统采用类似的方案，下一节将会讨论 UNIX/Linux 的保护控制机制。

如果按照列来保存存取控制矩阵，为每一个用户保存一张可以访问的文件表及其操作，就形成了权限表，其中每一项叫权限，用户 Bin 的权限表如表 5-3 所示。

表 5-3 用户 Bin 的权限表

| 编 号 | 权 限 | 对象（文件） |
|---|---|---|
| 0 | X | 指向 /bin/datetime 的指针 |
| 1 | R | 指向 /home/alice/bin/lc 的指针 |
| 2 | RW | 指向 /home/bin/README 的指针 |

上表中，权限域给出了对象（文件）允许执行的合法操作，通过对象域的指针，指向被保护的对象。由表 5-3 可知，用户 Bin 可以执行文件“/bin/datetime”，可以对文件“/home/bin/README”进行读写。

## 5.2 Linux 文件系统

前面介绍了文件系统的基本原理，本节通过 Linux 文件系统进一步深入分析文件系统的具体设计方法。Linux 的本地文件系统包括：

- Ext2（Second Extended Filesystem，第二扩展文件系统）
- Ext3（ThirdExtended Filesystem，第三扩展文件系统）

它们是 UNIX 文件系统的一种快速、稳定的实现。同时，为了支持多种类文件系统，如网络文件系统、Windows 文件系统等，Linux 还引入了虚拟文件系统（Virtual File System，VFS），应用可以利用标准的 UNIX 文件系统调用，对不同种类的文件系统进行操作。

### 5.2.1 Linux 本地文件系统

首先研究 Linux 的本地文件系统，i-node 或索引节点是 Linux/UNIX 文件系统中最有名也是最重要的概念，它存储了文件和目录的元数据（HDFS 中对应的结构分别是 INodeFile 和 INodeDirectory，它们是 INode 的子类，借鉴了 i-node 的命名）。所有索引节点大小相同都是 128 字节，这样，如果磁盘的块（块由多个连续的扇区组成）大小为 1024 字节，它可以包含 8 个索引节点。

当 i-node 用于保存文件时，使用一种变种的索引分配方案，典型结构如图 5-6 所示。

基本的索引节点开始部分保存了文件的一些元信息，如文件类型与权限（最前面的 2 字节）、所有者标识（接下来的 2 字节）、以字节为单位的文件长度（4 字节，严格地说，是文件长度的低 32 位）等信息。索引节点后半部分是文件所在数据块的索引，开始的 12 个块地址（每个地址需要占用 4 字节）存放在 i-node 内，这样，对于小文件，所需信息都在 i-node 中。如果（假设）文件管理器的块大小是 4KB，那么这 12 个索引能表示的最大文件是 48KB。对于稍大的文件，在 i-node 中有一个称为一次间接块的索引，索引指向的块以连续的方式存放了数据块的索引，也就是文件内容对应的块地址。对于大于 48KB 的文件（假设块是 4KB），部分数据的访问，需要访问一次索引节点和一次间接块，才能定位到数据所在的块。类似的，更大的文件可以通过二次间接块、三次间接块保存数据块的索引。通过

间接块技术，i-node 很好地实现了在支持大文件的同时，保证了小文件的访问效率。下面是 Linux Ext2 文件系统 i-node 定义的源代码，供大家参考。

```
ext2_inode {
    __u16       i_mode;                 /* 文件类型和访问权限 */
    __u16       i_uid;                  /* 拥有者标识符 */
    __u32       i_size;                 /* 以字节为单位的文件长度 */
    __u32       i_atime;                /* 最后一次访问文件时间 */
    __u32       i_ctime;                /* 索引节点最后改变时间 */
    __u32       i_mtime;                /* 文件内容最后改变时间 */
    __u32       i_dtime;                /* 文件删除时间 */
    __u16       i_gid;                  /* 用户组标识符 */
    __u16       i_links_count;          /* 硬链接计数器 */
    __u32       i_blocks;               /* 文件的数据块数 */
    __u32       i_flags;                /* 文件标志 */
    union {
        ......
    } osd1;                             /* 特定操作系统信息 */
    __u32       i_block[EXT2_N_BLOCKS]; /* 指向数据块的指针 */
    __u32       i_generation;           /* 文件版本 (for NFS) */
    __u32       i_file_acl;             /* 文件访问控制列表 */
    __u32       i_dir_acl;              /* 目录访问控制列表 */
    __u32       i_faddr;                /* 片的地址 */
    union {
        ......
    } osd2;                             /* 特定操作系统信息 */
};
```

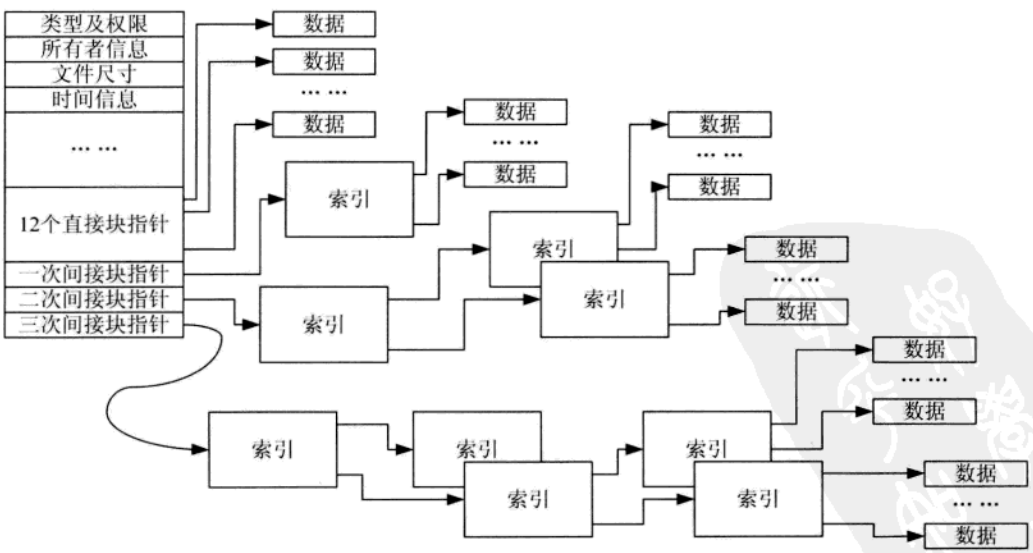



图 5-6　文件的 i-node 典型结构

文件的元信息都存放在文件的 i-node 中，目录项被保存在为目录分配的块中，它和一个正常的文件很相似，只是它的 i-node 上的类型和普通文件不一样。前面介绍索引节点时提过，索引节点最开始的 2 字节保存了文件类型与权限，普通文件类型的类型值为 1，目录为 2。目录的内容，即目录包含的下一级目录项，可以是目录，也可以是文件，被组织成目录项记录保持在数据块中，通过数据块的索引在目录项的 i-node 中管理。

由于目录项的元信息和普通文件一样存放在 i-node 中，所以，目录项记录包含的信息很少，图 5-7 给出了新版本目录项记录（Linux 目录项有两种格式）中保存的内容和数据布局。其中，最前面的 4 字节是目录项对应的 i-node 号，接下来是本目录项的长度（注意，记录长度出现在记录的第二个字段，而且包含 i-node 号的那 4 字节），接下来是文件名长度，由于它占 1 字节，所以，Linux 文件的文件 / 目录名最长是 255 字节。

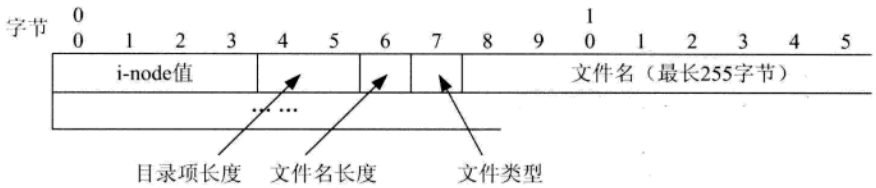



图 5-7 Linux 目录项

一个目录下的所有目录项组织起来，作为数据存放在数据块中。其中，每个目录的前两个目录项分别是当前目录“.”和父目录“..”。打开文件时，文件管理器根据文件名，找到它所在的磁盘块。

例如，要查找路径名“/home/alice/data.txt”的文件，首先文件管理器要找到根目录，Linux 的根目录总是位于系统的 2 号 i-node，通过 2 号 i-node 可以找到根目录目录项所在的数据块，在该块的内容中找路径的第一部分：“home”，从而获得目录“/home”的 i- 节点号，由 i-node 号可以很直接找到对应的 i-node，然后通过这个 i-node 找到对应的数据块，就可以开始查找目录“/home/alice”了。利用相同的流程，最终获得“/home/alice/data.txt”的 i-node 号。打开这个文件时，对应的 i-node 会被读入内存，如图 5-8 所示。

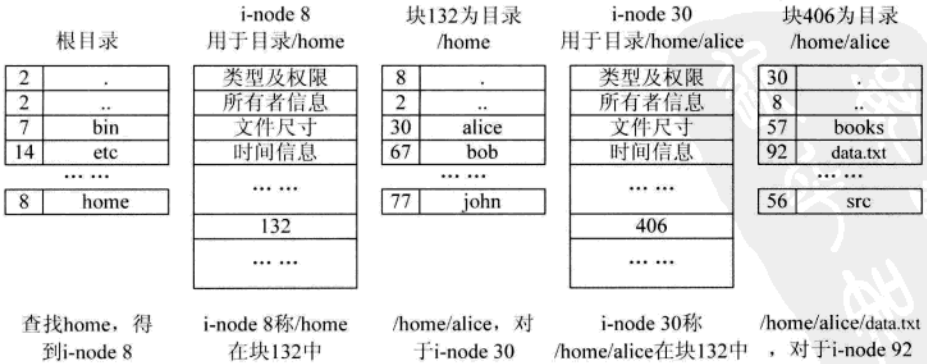



图 5-8 查找 /home/alice/data.txt

了解了 i-node 和数据块以后，进一步分析 ExtX 的逻辑磁盘结构，如图 5-9 所示。

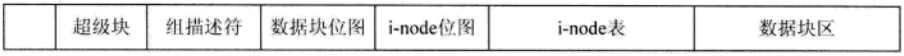



图 5-9　Ext2/Ext3 逻辑磁盘结构

其中，超级块保存了这个逻辑磁盘结构的一些信息，包括块大小、总块数、总 i-node 数等。数据块位图和 i-node 位图分别记录了 i-node 表和数据块的使用情况，这样的组织有利于 i-node 和数据块的分配，接下来才是前面介绍的 i-node 表和数据块区，如果知道一个 i-node 号，由于 i-node 的大小都是一样的，根据 i-node 表的起始位置，很快就可以定位到对应的 i-node，对于数据块，定位的机制也是一样的。

通过 Linux 的 mount 命令安装文件系统时，文件管理器会读入超级块，这样就可以获得整个 ExtX 文件系统的信息，进而挂载整个文件系统。超级块的格式属于存储媒体上的文件系统的物理结构研究的范围。

## 5.2.2　虚拟文件系统

随着需求的增长，要求操作系统能够支持不同类型的文件系统，如在 Linux 环境中访问 NTFS（New Technology File System，新技术文件系统）。为了实现这个功能，Linux 内核使用了虚拟文件系统（Virtual File System，VFS），它是内核中的一个软件层，为上层应用提供文件系统接口，该接口隐藏了底层各种文件系统的具体细节。基于 VFS 的文件管理器如图 5-10 所示。

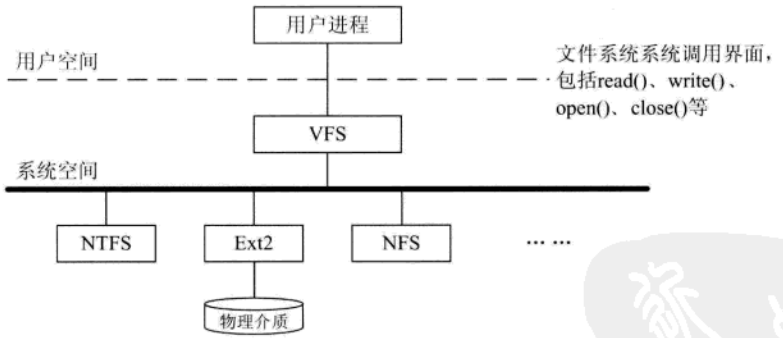



图 5-10　基于 VFS 的文件管理器

VFS 是由面向对象的思想发展起来的，提供了一个抽象基类，由这个基类派生的子类支持具体的文件系统。真正的文件系统，如 Ext2/Ext3、NFS（网络文件系统）、ISO 9660 等，在 VFS 提供的统一接口下工作。当应用程序对文件系统进行操作时，内核文件系统首先调用 VFS 的相应函数，处理与文件系统无关的操作，然后再调用真正文件系统中对应的函数，处理与设备相关的操作。

VFS 提供的模型源于 UNIX 风格的文件系统：文件、目录项、索引节点和超级块。像

FAT（File Allocation Table）、NTFS 这样非 UNIX 风格的文件系统，为了在 Linux 中工作，它们必须经过封装提供一个符合这些概念的界面。如 FAT，它本来就不支持索引节点，但具体支持 FAT 的文件管理器，必须在内存中，根据 FAT 文件系统的结构，构造出对应的索引节点，就像它本身就支持索引节点一样。但即使如此，Linux 仍然可以支持差异很大的文件系统。

VFS 在现代文件管理器中获得巨大的成功，并为实现远程文件服务器提供了基础。

### 5.2.3 Linux 文件保护机制

Linux 文件系统采用一种改进的存取控制表来实现文件系统的保护控制。首先将文件的潜在用户分为以下 3 种：

❑作为文件所有者的用户。

❑同组用户、不包括所有者。

❑所有剩下的用户（其他）。

每个文件都保存了文件所有者和文件所有者所在用户组的信息。同时，Linux 将文件的访问权限分为读、写和执行，每组用户都有这几种权限。通过将文件潜在用户分类和权限相结合，Linux 实现了对文件的保护，这样，每个文件只需要 9 位信息，就实现了存取控制表，该方案虽然没有上面讨论的存取控制表那么全面，但实用、简单而且方便。

对于文件的读、写和执行权限，具体内容为：

❑r（read）：可以读取文件的内容。

❑w（write）：可以编辑、修改文件的内容。

❑x（execute）：该文件可以被执行。

需要注意的是，文件的这些权限都是针对文件的内容，与文件本身没有任何关系。即便是对文件有“rwx”权限，用户也不一定可以修改文件名或删除文件。

对于目录，如果将目录下的所有文件 / 子目录看成是目录的内容，其读、写和执行权限具体内容为：

❑r（read）：可以读取文件夹内容列表，但如果用户没有“x”权限，就只能看到文件名而无法查看其他内容（大小、权限等）。

❑w（write）：用户具有“w”权限，就可以修改目录的内容，即文件夹记录列表，前提是用户拥有“x”权限，可以进入这个目录。修改目录的内容，也就是“w”权限，包括：建立新的文件或文件夹、删除已存在的文件或文件夹、对已存在的文件或文件夹改名和更改目录内文件或文件夹的位置。

❑x（execute）：可以进入该文件夹，没有“x”权限便无法执行该目录下的任何命令。

需要注意的是，对文件 / 子目录的改名与删除，系统检查文件父目录的“w”权限，而和文件本身的权限没有关系。当用户改名或删除文件 / 子目录时，执行的是对它上一级目录的“w”操作，因此，删除文件 / 子目录不需要考虑被删除文件 / 子目录自身的权限设置。

如表 5-4 所示为 Linux 存取控制表。

表 5-4　Linux 的存取控制表

| 文件 / 目录 | 文件所有者 | 文件所有者所在的用户组 | 访问控制 |
|---|---|---|---|
| /home/bin/README | Bin | user | rw------- |
| /home/alice/bin/lc | Bin | user | r---wx--- |
| /bin/datetime | root | root | rwxr-xr-x |
| /bin | root | root | rwxr-xr-x |

对于文件“/home/bin/README”，只有用户 Bin 才能够进行读写，其他用户不能进行任何读、写和执行操作。

对于文件“/home/alice/bin/lc”，文件所有者 Bin 能读取文件，不能写和执行；处于 user 组的其他用户具有写和执行的权限，其他用户没有任何权限。

对于文件“/bin/datetime”，系统中所有的用户都具有读和执行权限，但只有 root 用户才能对文件内容进行修改。

**注意**　上面关于文件“/bin/datetime”的执行权限，严格来说，还需要考虑到目录“/bin”的执行权限。

最后分析目录“/bin”，该目录的访问控制位是“rwxr-xr-x”，除了 root 用户对目录具有写权限以外，其他用户都不能对“/bin”目录进行修改。也就是说，只有 root 用户才能在“/bin”目录下创建文件 / 子目录或删除文件 / 子目录，或者对文件 / 子目录进行改名等。非 root 用户由于只有“r-x”权限，他们能够在“/bin”下列目录，或执行文件（命令），但不能修改目录内容。

### 5.2.4　Linux 文件系统 API

Linux 文件系统 API 分为两类：

- 文件 I/O 函数：用于对文件数据读写。
- 文件 / 目录 API：主要是针对文件、目录属性的操作和目录树的操作，不涉及文件数据读写。

表 5-5 总结了 Linux 文件系统的相关 API。

表 5-5　Linux 文件操作（主要）

| 操　作 | 描　述 |
|---|---|
| open | 打开一个文件 |
| read | 读取文件中包含的数据 |
| write | 向文件中写入数据 |
| close | 关闭一个文件 |
| lseek | 改变文件读写位置 |
| stat | 获取文件 / 目录的属性 |

（续）

| 操　作 | 描　述 |
|---|---|
| chmod/chmod/utime | 修改文件属性 |
| create | 创建一个文件 |
| remove | 从文件系统中删除一个文件 |
| rename | 更改文件名 |
| mkdir | 在给定目录下创建一个子目录 |
| rmdir | 从一个目录中删除一个空的子目录 |
| readdir | 读取一个目录下的项目 |
| getcwd/getwd | 返回当前工作目录 |
| chdir | 更改当前工作目录 |

### 1. 文件 I/O 函数

文件 I/O 函数包括 5 个函数：open()、read()、write()、close() 和 lseek()。应用读写文件之前，必须首先打开文件，这个操作使得文件管理器为需要读写的指定文件做准备。

系统调用 open() 打开文件，形式如下：

```
intfid=open("/home/alice/data.txt", flag)
```

其中第一个参数规定了要打开文件的路径名，flag 是一个开关，如 flag 中 O_RDONLY 被设置，表明这次打开是为读打开的。当文件被打开时，文件管理器通过前面介绍的步骤，在目录树中查找文件，并进行检查以确保进程被授权访问文件。文件管理器还会对 open() 操作检查其他约束，如文件的读 / 写锁。在确认了进程可以访问文件后，系统会在特定的进程打开文件表中创建表项，该表项由调用返回的整数值来表示，该表项指向打开文件表中被称为文件结构的另外一个表项，文件结构中保存了文件读写位置等信息，如图 5-11 所示。

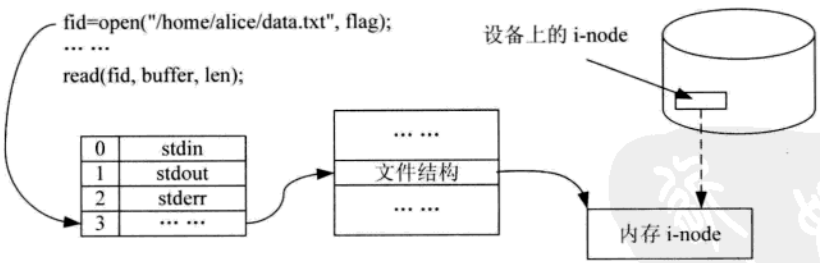



图 5-11　打开文件“/home/alice/data.txt”

如果有两个不同的进程打开了同一个文件，如“/home/alice/data.txt”，那么每个进程都有它们自己的文件读写位置。打开文件的同时，文件的 i-node 也会被加载到主存，文件结构表中有指针指向内存中的 i-node 拷贝。特定文件的 i-node 拷贝只有一份，也就是说，两个不同的进程打开同一个文件，它们各自的打开文件表，指向了同一个 i-node 拷贝。

文件打开后，就可以进行读写操作了。以 read() 为例（限于篇幅，不介绍 write() 的过

程），该函数的语法格式如下：

```
char buffer[1024];
int number=read(fid, buffer, 1024)
```

fid 是由 open() 打开的文件描述符，buffer 是用户进程空间中的一个数据结构的地址，read() 的第三个参数是该数据结构的大小，或用户要读的字节数。该调用结束后，buffer 中存放所读的文件数据，返回值 number 是实际读取数据的字节数。

文件管理器接到 read() 调用时，根据 fid 找到对应的文件结构项和内存 i-node。接着，读请求被分成几个段，每段对应于设备上的一块。例如，当前文件位置为 1600 字节，要读取的数据长度为 1K 字节，设备块大小为 1K 字节，那么读请求将分为两个部分，分别为 1600 ～ 2023 字节和 2024 ～ 2623 字节。通过 i-node 的直接块指针，可以找到相应的块号，在不考虑文件管理器高速缓存的情况下，内核向设备管理系统发出两个读操作请求。设备管理系统读操作返回后，再将数据拷贝到用户提供的缓存区，然后文件管理器送出响应消息，告知拷贝字节数，调用结束。

每个打开文件对应的打开文件表中保存了文件读写位置，通常读写操作都是从这个位置开始，通过 lseek() 可以改变文件读写位置。

写文件结束以后，可以通过 close() 方法关闭文件，释放相关资源。文件管理器如果更新了内存中的 i-node（如在文件中追加数据改变了文件长度），在文件关闭或应用程序发出 sync() 命令后，内存中的 i-node 会被更新到设备上。

---

**注意**　上述过程是一个简化的 Linux 文件读操作的过程，由于 VFS 的引入，实际上在 Linux 上读取一个 Ext2 文件的过程，要比上述过程复杂得多。

---

### 2. 文件 / 目录函数

（1）文件属性的操作

文件 / 目录 API 中，处理文件属性可以通过 stat() 函数获得，该函数的原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char* pathname, struct stat* buf)
```

给定一个文件 pathname，stat() 函数通过填充 buf 指向的一个结构，返回与此文件相关的信息。其 Linux 源代码如下：

```
struct stat {
   dev_t     st_dev;     /* 文件所在的设备 ID */
   ino_t     st_ino;     /* i-node 号 */
   mode_t    st_mode;    /* 访问权限 */
   nlink_t   st_nlink;   /* 硬链接数 */
   uid_t     st_uid;     /* 文件所有者的用户 ID */
   gid_t     st_gid;     /* 文件所有者的组 ID */
   dev_t     st_rdev;    /* 设备（应用的特殊文件） */
```

```
    off_t     st_size;    /* 文件的大小，以 byte 为单位 */
    blksize_t st_blksize; /* 用于文件 IO 的块大小 */
    blkcnt_t  st_blocks;  /* 已经分配的文件块数 */
    time_t    st_atime;   /* 最后访问时间 */
    time_t    st_mtime;   /* 最后修改时间 */
    time_t    st_ctime;   /* 元数据最后修改时间 */
};
```

结构 stat 包含了文件类型、访问权限、i-node 号、用户 ID，用户组 ID、文件大小、文件最后访问 / 修改 / 属性修改时间等信息。通过访问 stat 结构，可以获取文件 / 目录的元数据。

对于文件属性的修改，Linux 提供了大量的方法，如使用 chmod() 更改现存文件的存取许可；使用 chown() 可对文件的用户 ID 和用户组 ID 进行修改；用 utime() 可以修改文件存取时间和修改时间。具体方法定义如下：

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
int chown(const char *path, uid_t owner, gid_t group);
int utime(const char *filename, const struct utimbuf *buf);
```

（2）目录树的操作

目录树的操作函数包括：create() 函数用于创建一个文件（也可以用 open() 来创建文件）；remove() 函数用于删除一个文件；文件或目录的改名用 rename() 函数；mkdir() 和 rmdir() 分别用于创建和删除目录。同时，Linux 包含了一套读目录的相关例程，包括如下方法：

```
#include <sys/types.h>
#include <dirent.h>

DIR*opendir(const char* pathname);
struct dirent*readdir(DIR* dp);
void rewinddir(DIR* dp);
void seekdir(DIR* dp, long int loc);
int closedir(DIR* dp);
```

如果把 Linux 目录看成是结构化文件，显然这套例程使用了和前面介绍的读文件类似的 open()、read()、close() 模式，对应方法分别是 opendir()，readdir() 和 closedir()。其中，readdir() 返回一个结构，该结构包含了目录项中的 i-node 号和文件名，如果读到目录尾，该函数返回 NULL。通过多次调用 readdir() 函数，可以遍历一个目录下的所有目录项。

方法 seekdir() 和 rewinddir() 类似于 lseek()，用于改变当前 readdir() 的读位置。

（3）进程工作目录操作

最后介绍的是和进程工作目录相关的 API，通过下面的系统调用，进程可以设置 / 获取它的工作目录。工作目录和相对路径配合工作，用于在目录树上定位文件。

```
#include <unistd.h>
char *getcwd(char *buf,size_t size);
char *getwd(char *pathname);
int chdir(const char *path);
```

## 5.3　分布式文件系统

分布式文件系统是常规单机文件系统的发展，其目的是允许物理上分散的计算机用户使用公共的文件系统，共享数据和存储空间。

### 5.3.1　分布式文件系统的特性

分布式文件系统研发开始于 20 世纪 70 年代，作为分布式系统的重要子系统，它需要解决一些典型的分布式服务的需求和困难，如：

- ❑访问透明性：文件系统的用户不需要了解文件的分布性，用户通过一个统一的文件操作集来访问本地或远程文件。最好能做到访问本地的程序不需要修改，就可以访问分布式文件系统中的文件。
- ❑位置透明性：客户程序可以使用单一的文件名空间。在不改变路径名的情况下，多个文件或文件组可以被重新定位。客户程序在任意时刻执行时都使用同样的命名空间。
- ❑移动透明性：文件被移动时，客户程序不必要改变，多个文件或文件组可以被系统管理员移动或自动移动。
- ❑性能透明性：分布式文件系统应该保证系统负载在一定范围内变化时，客户程序可以获得和常规文件系统差不多的性能。
- ❑伸缩透明性：文件系统可以不断地扩充，以满足负载和规模的增长。当服务负载和用户数增加时，能用简单的方法增加资源。
- ❑复制透明性：使用资源的多个实例，即一个文件可以表现为在不同位置文件内容的多个副本，来增加可靠性和性能，客户程序和应用程序员不需要了解副本的存在。
- ❑故障透明性：屏蔽错误，不论硬件故障或是软件组件故障，允许客户程序和用户完成他们的任务。
- ❑并发透明性：几个进程能并发地对共享资源进行操作而互不干扰。一个客户改变文件的操作不应该影响其他客户访问或改变同一文件的操作。

除了上述特性，分布式文件系统还需要关注数据完整性、安全性和系统的异构等问题。分布式文件系统通过多种组织结构，实现以上部分或全部特性。

早期的大多数分布式文件系统是遵循传统的客户机 - 服务器体系结构构造，如 Sun Microsystems 的网络文件系统（Network File System，NFS）、卡内基梅隆大学的 AFS（Andrew File System）；更松散的解决方案采用基于集群的分布式文件系统，典型的例子是本书的主题，Hadoop 的 HDFS 和 Google 分布式文件系统 GFS（Google File System）；基于 P2P（Peer to Peer）的文件系统是完全分散式的解决方案，这类系统的示例是麻省理工学院

的 Ivy，它构建在 Chord DHT（Distributed Hash Table）的基础上。

### 5.3.2 基本 NFS 体系结构

NFS 是分布式文件系统的一个典型示例，它的结构可以增强，用于服务器集群，形成 HDFS、GFS 这样的系统。

NFS 最初由 SUN 公司开发，并运行在 SUN UNIX 工作站上。NFS 背后的基本思想是：每个文件服务器都提供其本地文件系统的一个标准化视图。换句话说，无论本地文件系统是如何实现的，每个 NFS 服务器通过一个通信协议，支持相同的模型，该协议允许客户以完全相同的方法访问存储在一个服务器上的文件，从而允许客户进程共享公用的文件系统。用于 UNIX 系统的基本 NFS 体系结构如下图。其中，VFS 是虚拟文件系统，NFS 客户端作为 VFS 的一个具体的文件系统。当应用程序对文件进行操作时，内核处理完与文件系统无关的操作后，调用 NFS 客户，该组件负责处理对存储在远程服务器上的文件的访问。

在 NFS 中，所有的客户－服务器通信都是通过远程过程调用（Remote Process Call，RPC）完成的。在服务器端可以看到相似的组织方式，NFS 服务器负责处理传入的客户请求，并把这些请求转换为常规的 VFS 文件操作，随后将这些操作传递给 VFS 层，VFS 负责对实际的文件系统进行操作。这种模式的优点使 NFS 很大程度上独立于本地文件系统，从原理上讲，它不关心客户端或服务器端的操作系统实现。

基于 UNIX 系统的基本 NFS 体系结构如图 5-12 所示。

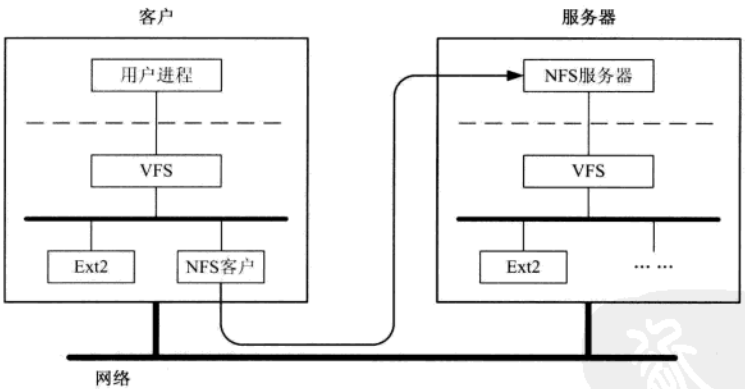



图 5-12 基于 UNIX 系统的基本 NFS 体系结构

NFS 提供的文件系统模型几乎与基于 Linux 的系统所提供的文件系统模型一致：文件解释为字节序列，被组织到目录树中；文件具有文件名，并通过一种类似 Linux 的文件句柄访问。也就是说，访问一个文件，客户需要通过文件名获得对应的句柄。

### 5.3.3 NFS 支持的文件操作

表 5-6 给出了 NFS 版本 3 和版本 4 支持的一些文件操作。

表 5-6　NFS 支持的文件操作

| 操　作 | 版本 3 | 版本 4 | 描　述 |
|---|---|---|---|
| lookup | 有 | 有 | 根据文件名查找文件 |
| open | 无 | 有 | 打开一个文件 |
| read | 有 | 有 | 读取文件中包含的数据 |
| write | 有 | 有 | 向文件中写入数据 |
| close | 无 | 有 | 关闭一个文件 |
| getattr | 有 | 有 | 获取文件的属性值 |
| setattr | 有 | 有 | 修改文件的一个或多个属性 |
| create | 有 | 无 | 创建一个文件 |
| create | 无 | 有 | 创建一个特殊文件，如目录 |
| remove | 有 | 有 | 从文件系统中删除一个文件 |
| rename | 有 | 有 | 更改文件名 |
| mkdir | 有 | 无 | 在给定目录下创建一个子目录 |
| rmdir | 有 | 无 | 从一个目录中删除一个空的子目录 |
| readdir | 有 | 有 | 读取一个目录下的项目 |

表中有许多我们熟悉的操作，如 mkdir（NFS 4 中的 create）、rename、remove、rmdir（在 NFS 4 中通过 remove 实现）、getattr、setattr 等。lookup 操作用于查找给定路径的文件，并返回相应的句柄；readdir 用于读取目录中的项目，该操作返回 < 文件名，文件句柄 > 列表。

NFS 版本 3 和版本 4 最大的差别是，NFS 3 服务器是无状态的，而 NFS 4 通过引入租约（Lease），实现带状态的服务器，引入 open 和 close 也是为了适应这一变化。通过租约，NFS 4 提供了简单的文件锁定机制（HDFS 目前还不提供这样的能力）和有限的权限委托（GFS 实现了部分这样的能力，HDFS 不提供）。在 NFS 4 中，打开一个不存在的文件会创建相应的文件，所以不需要 create 操作。

NFS 中从文件读取数据和向文件写入数据的操作时通过 read 和 write 进行，原型如下：

```
read(fh, offset, count) → attr, data
write(fh, offset, count, data) → attr
```

客户指定要读取的偏移量和字节数，read 操作返回已被读取的实际字节数，以及附加的状态信息（例如是否到达文件末尾），写操作客户需要指定写入的开始位置、字节数和数据。

NFS 部分实现了前面提到的分布式文件系统的设计目标。

NFS 通过对访问接口的抽象，结合 VFS，实现访问透明性。

NFS 位置透明性是通过将远程文件系统加载（mount）到客户端的本地命名空间实现，由于远程文件系统加载需要在客户端上进行操作，因此 NFS 不能完全达到移动透明性。

NFS 性能透明性和伸缩透明性可以通过增加 NFS 服务器的处理器、磁盘等在一定的范围内得到提高，但单一服务器的设计还是有一定的极限。

NFS 只支持只读文件在多个服务器上的复制，并没有完全实现复制透明性。

故障透明性是通过 NFS 协议的无状态和幂等（NFS 3）实现的，服务器失效以后，它提

供的服务会被挂起，但一旦服务器重启，客户端就可以从服务被打断的那一点继续执行。

至于并发透明性，NFS 提供了满足大部分应用需求的更新行为。

## 5.4 Java 文件系统

处理文件、目录以及读写文件数据是程序设计语言的重要组成部分，也是从用户角度对文件系统在另一个层次进行抽象，从而设计一个令人满意的输入 / 输出系统，是件极其艰巨的任务。Java 与文件系统交互的类库可以分为两部分：

❑用于处理文件和目录的相关事务；

❑用于读写文件数据。

### 5.4.1 Java 文件系统 API

Java 在 java.io.File 类中封装了对用户机器文件系统进行操作的功能，类名 File 很容易造成误解，你可能认为它只用于处理文件，但是 File 类其实包含了对用户机器的文件系统进行操作的功能。例如，可通过 File 类获得文件上次修改的时间、移动文件，也可以删除目录、移动目录。

File 类提供了一系列的构造函数，其中最简单的构造函数是为 File 对象提供一个（完整的）文件名。如果文件名不是以绝对路径提供，那么 Java 使用当前工作目录，作为相对路径分析的起点。下面的代码创建一个文件对象，引用 Java 虚拟机工作目录（也就是启动 Java 虚拟机时系统所在的目录）下的文件“test.txt”：

```
File f=new File("test.txt");
```

如果文件不存在，该构造方法也不会创建文件 test.txt。通过调用 File.createNewFile() 方法或某一个流类的构造函数可以创建文件。另一方面，一旦有了 File 对象，通过 exists() 方法可以判断文件是否存在。

File 对象可以用于表示路径，通过使用 isDirectory() 和 isFile()，可以判断文件对象表示的是文件还是路径。下面的代码输出两个“true”（在 Windows 和 Linux 环境下）。如果目录不存在，通过 File.mkdir() 可以创建相应的目录：

```
File f=new File("/");
System.out.println(f.exists());
System.out.println(f.isDirectory());
```

File 对象如果是目录，使用 list() 方法可获得该目录下所有文件名的数组，使用 isDirectory() 方法和 list() 方法就可以遍历目录树。

File 类中对目录树进行操作的方法如下，按顺序分别用于创建文件，删除文件或目录，虚拟机退出时删除文件或目录，创建目录和修改文件名（移动文件）：

```
public boolean createNewFile()
public boolean delete()
```

```
public void deleteOnExit()
public boolean mkdir()
public boolean renameTo(File dest)
```

File 对象还提供了大量的方法，用于访问文件 / 目录的元数据。例如，canRead() 方法可用于判断文件 / 目录能否由当前应用程序读取；getCannonicalFile() 返回一个包含文件规范路径名的 File 对象。文件 / 目录的规范路径名是绝对路径名，并且是唯一的，其中，路径名中多余的名称（如“.”目录）会被删除，并使用正确的目录分隔符。File 类的其他方法请参考 Java API 的相关文档。

需要注意的是，和 Linux 不同，Java 并没有提供设置 / 获取工作目录的 API。访问系统属性“user.dir”可以获得进程工作目录，代码如下：

```
String currentdir = System.getProperty("user.dir");
```

Java 没有提供任何方法设置进程的工作目录。

File 类的实现是通过 java.io.FileSystem 进行的，FileSystem 是 java.io 包的私有抽象类，也就是说，只有处于 java.io 包的类才可以访问 FileSystem。File 和 FileSystem 的类图如图 5-13 所示。

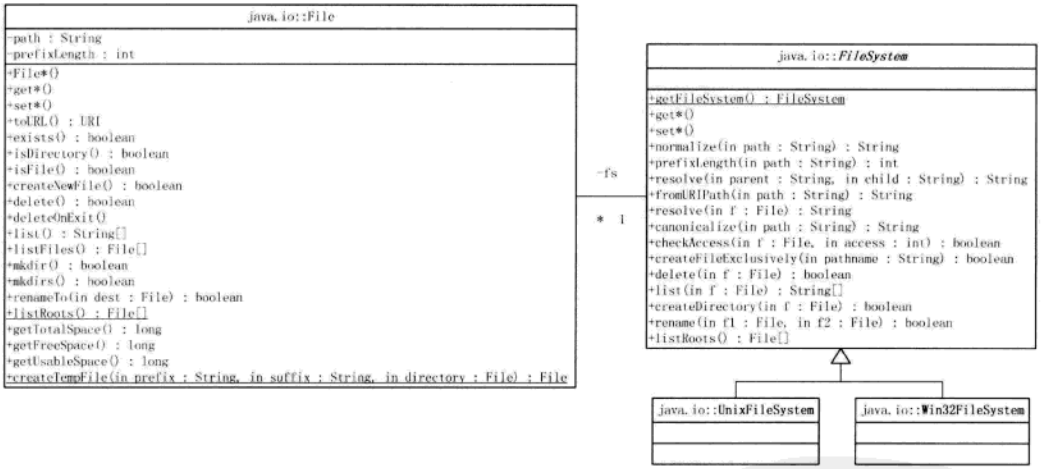



图 5-13　File 和 FileSystem 的类图

仔细分析 FileSystem 类，可以发现它有大量的方法名和 File 是一样或者类似的，因为大量的 File 方法是通过 FileSystem 实现的。以 File 的 renameTo() 方法为例，实现方式如下：

```
public boolean renameTo(File dest) {
   SecurityManager security = System.getSecurityManager();
   if (security != null) {
      security.checkWrite(path);
      security.checkWrite(dest.path);
   }
   return fs.rename(this, dest);
}
```

在进行必要的安全检查以后，File 类的 renameTo() 方法通过调用 FileSystem 的 rename() 方法对文件名进行修改。其中，fs 是 File 类中的静态成员变量，代码如下：

```
static private FileSystem fs = FileSystem.getFileSystem();
```

FileSystem 是一个抽象类，在 Windows 系统上，它的实现是子类 Win32FileSystem，FileSystem.rename() 最终通过 Win32FileSystem 中的原生（native）方法，使用操作系统提供的 API，完成文件改名操作。

表 5-7 总结了 Java 文件操作。

**表 5-7 Java 文件操作（部分）**

| Java 操作 | Linux 操作 | 描 述 |
|---|---|---|
| URL.openStream 或通过流的构造函数 | open | 打开一个文件 |
| InputStream.read | read | 读取文件中包含的数据 |
| OutputStream.write | write | 向文件中写入数据 |
| InputStream.close<br>OutputStream.close | close | 关闭一个文件 |
| RandomAccessFile.seek | lseek | 改变文件读写位置 |
| File.get*，如 isFile | stat | 获取文件 / 目录的属性 |
| File.set*，如 setReadOnly | chmod/chmod/utime | 修改文件属性 |
| File.createNewFile | create | 创建一个文件 |
| File.delete | remove | 从文件系统中删除一个文件 |
| File.renameTo | rename | 更改文件 / 目录名 |
| File.mkdir | mkdir | 在给定目录下创建一个子目录 |
| File.delete | rmdir | 从一个目录中删除一个空的子目录 |
| File.list | readdir | 读取一个目录下的项目 |
| | getcwd/getwd | 返回当前工作目录 |
| | chdir | 更改当前工作目录 |

## 5.4.2 URI 和 URL

如图 5-13 所示，File 类提供了 toURI() 操作，可以把文件对象转换成对应的 URI：

```
public URI toURI()
```

URI（Uniform Resource Identifier，统一资源标识符）以特定的语法标识一个资源的字符串。所标识的资源可以是文件，如上面 File 类的例子，也可以是一个电子邮件地址、一本书或其他东西。绝对 URI 由 URI 模式和模式特有部分组成，它们之间由冒号隔开，格式为：

```
模式：模式特有部分 # 片段 ([<scheme>:]<scheme-specific-part>[#<fragment>])
```

模式特有部分的语法依赖于所使用的模式，比较常用的模式有：

- file：本地磁盘上的文件，上面 toURI() 例子的一个可能输出为“file:/E:/”。
- http：使用超文本传输协议的万维网服务器，例如“http://www.hzbook.com”。

❑ftp：FTP 服务器。

❑mailto：电子邮件地址。

❑telnet：与基于 Telnet 协议的服务的连接。

HDFS 中还使用了 hdfs、har、s3、kfs 等非标准自定义模式，分别对应不同的资源。

URI 中的模式特有部分没有特定的语法，但是很多的模式都定义了层次式的模式特有部分，如 http、ftp、hdfs 等。层次式的模式形式如下：

```
//授权机构/路径?查询 ([//<authority>]<path>[?<query>] )
```

以 URI“http://www.ietf.org/rfc/rfc2396.txt”为例（该 URI 对应的 RFC 定义了 URI），它具有模式 http 和授权机构 www.ietf.org。其中，授权机构表示位于 www.ietf.org 的服务器负责将路径“/rfc/rfc2396.txt”映射到资源。这个 URI 没有查询部分。另一个 URI“mailto:hzsj@hzbook.com”的模式是 mailto，模式特有部分是 hzsj@hzbook.com。显然，它的模式特有部分就不是上面讨论的层次式模式。

目前，大部分的 URI 使用 Internet 主机作为授权机构，在这种情况下，授权机构部分还可以包括可选的用户名和端口：

```
用户信息@主机:端口 ( [<userInfo>@]<host>[:<port>] )
```

例如，URI“ftp://user1:password1@example.com:83/hadoopInternal/chart2.pdf”的授权机构是 user1:password1@example.com:83。用户信息是 user1:password1，其中 user1 是用户名，password1 是用户 user1 对应的密码；主机为 example.com；端口是 83（FTP 的默认端口是 21）。它的模式是 ftp，路径是“/hadoopInternal/chart2.pdf”。

路径（包括开头的 /）是授权机构用来确定所标识资源的字符串，不同授权机构可以将不同的路径指向不同的资源。例如，授权机构是 www.hadoopInternal.org 时，路径“index.html”可能表示一种意思，而授权机构是 www.ietf.org 时，路径“index.html”可能表示另一种意思。路径可以是分层的，如前面提到的“/rfc/rfc2396.txt”。

在 Java 中，java.net.URI 类是对 URI 的抽象，如图 5-14 所示。

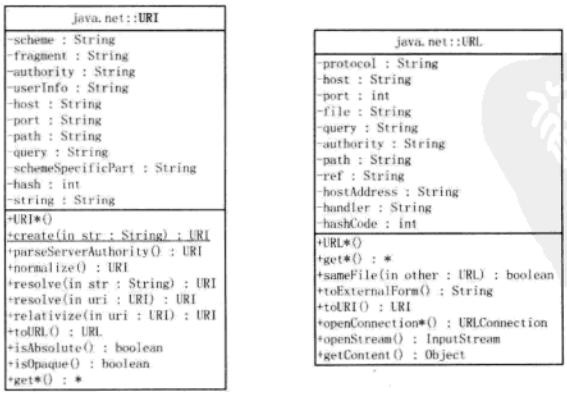



图 5-14　URI 和 URL 类图

URI 提供了若干种构造函数创建 URI 对象。以最简单的构造函数为例：

```
public URI(String str)
```

该构造函数接收一个字符串表示的 URI，并进行解析。如果输入参数 str 为“ftp://user1:password1@example.com:83/hadoopInternal/chart2.pdf”，可以通过 URI 类的 get*() 方法获得 URI 相应的各部分。如 scheme 为 ftp，schemeSpecificPart 为 //user1:password1@example.com:83/hadoopInternal/chart2.pdf，userInfo 为 user1:password1。

有两种类型的 URI：

❑URL（Uniform Resource Locator，统一资源定位符）

❑URN（Uniform Resource Name，统一资源名）

URL 是指向 Internet 上位于某个位置的某个资源，例如“http://www.ietf.org/rfc/rfc2396.txt”是 RFC 2396 的 URL 之一。

URN 用于标识资源本身，不关注资源的位置，也不限于表示 Internet 资源，例如本书的一个 URN 是“urn:isbn:0123456789”。

URL 用于标识 Internet 上资源的位置，可以指定用于访问服务器的协议（如 FTP、HTTP）、服务器的名称和文件在此服务器上的位置。典型的 URL，如上面提到的“http://www.ietf.org/rfc/rfc2396.txt”指明了服务器 www.ietf.org 的目录 rfc 下有一个名字为“rfc2396.txt”的文件，此文件可以通过 HTTP 协议进行访问。URL 的语法是：

```
协议 :// 用户信息 @ 主机名 : 端口 / 路径 ? 查询 # 片段
( [<protocol>:] [//[<userInfo>@]<hostname>[:<port>]]<path>[?<query>][#<fragment>] )
```

其中，协议相当于 URI 中的 scheme。主机名提供所需资源的服务器名，也可以是服务器的 IP 地址。用户信息和端口是可选的。路径和查询一起形成了 URL 的文件部分，其中路径指定了服务器上的某个文件（这里的文件需要和 URL 的文件部分区分），查询提供了获取文件的附加参数。最后，片段指向远程资源的某个部分。如果远程资源是 HTML，那么片段标识符将指定该 HTML 文档中的一个锚（anchor）。

URL 的实现是 java.net.URL 类，它和 URI 类似，提供了一系列构造方法和读取 URL 属性的 get*() 方法。由于 URL 指向的是 Internet 上的实际资源，java.net.URL 类中提供了打开到相应资源连接的机制。如 URL.openStream() 可以获得一个输入流 InputStream，用于读取资源：

```
public final InputStream openStream()
```

### 5.4.3 Java 输入 / 输出流

和前面介绍的 Linux 文件系统 API 不同，Java 的 I/O 系统是一个典型的基于流的系统，流代表任何有能力产出数据的数据源，或者是有能力接收数据的接收端。流的源和端可以是文件，也可以是内存块、网络，甚至是声卡等设备（这里不讨论 Java NIO）。HDFS 中，文件数据的输入 / 输出也实现为流的形式。

一个可以读取字节序列的对象称为输入流，一个可以写入字节序列的对象称为输出流，分别用抽象类 InputStream 和 OutputStream 进行封装，它们的类图如图 5-15 所示。

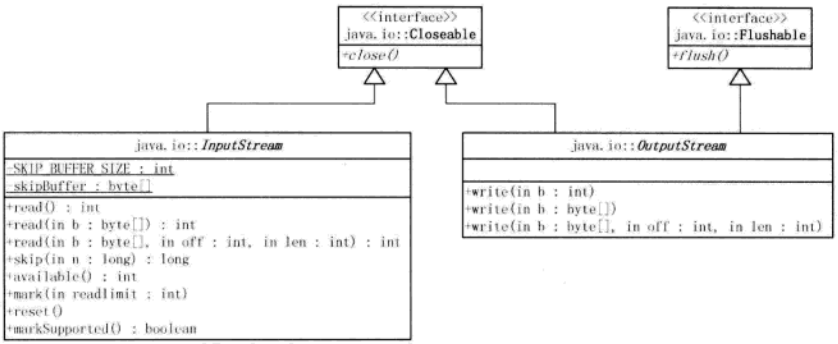



图 5-15　InputStream 和 OutputStream 类图

### 1. InputStream 和 OutputStream

InputStream 类有一个抽象的方法：

```
public abstract int read() throws IOException;
```

该方法读取 1 字节的数据，作为一个 0 ～ 255 的 int 返回；如果已经读到输入源的末尾，将返回 -1。具体的输入流类开发者通过覆盖这个方法处理特定的媒介。例如，在 FileInputStream 中，该方法一般使用一个本地方法从文件中读取数据；而在 ByteArrayInputStream 中，则通过纯 Java 代码从数据中复制 1 字节。

在最简单的 read() 方法基础上，InputStream 还提供非抽象的 read() 方法重载，用于读取 1 字节数组，默认实现是通过多次调用抽象的 read() 方法读取一批数据。

同样，OutputStream 类定义了抽象方法 write()，将 1 字节写到指定的输出位置：

```
public abstract void write(int b) throws IOException;
```

和 InputStream 类似，OutputStream 也提供批量的数据写出操作。无论是 read() 方法还是 write() 方法，都会阻塞一个线程直到字节被真正地读入或者写出，如果流不能立即被读取或者写入，Java 会挂起发起这个调用的线程。通过 available() 方法可以检查目前流中可读取的字节数，在调用 read() 方法前检查流中是否有数据可读，可以避免线程被阻塞。InputStream 通过 skip() 方法，支持在流中跳过 n 字节。InputStream 还可以支持在输入流的当前位置放置一个标记（不是所有的流都支持这个功能，可以通过 markSupported() 进行判断），然后通过 rese() 返回该位置。

OutputStream 中需要注意 flush()，该方法用于刷新输出流，即将输出缓冲区中的所有数据发送往目的地。

当完成对一个流的读取或写入以后，用户需要调用 close() 方法关闭流，这样可以释放流所占用的资源，关闭一个输出流，一般会自动调用 flush() 方法。

### 2. FileInputStream 和 DataInputStream

Java 提供了许多由基本 InputStream 和 OutputStream 类派生的流类，这些类可以被划分为两类：一些流（例如 FileInputStream 和 ByteArrayInputStream）可以从文件或者其他地方接收字节；另一些流（如 DataInputStream），可以将流中的字节组合成更加有用的数据类型。

将一个已经存在的流传递给另一个流的构造函数，可以将两种流结合起来，形成一个过滤器流（filtered stream）。图 5-16 分别给出了这两类流的具体例子。

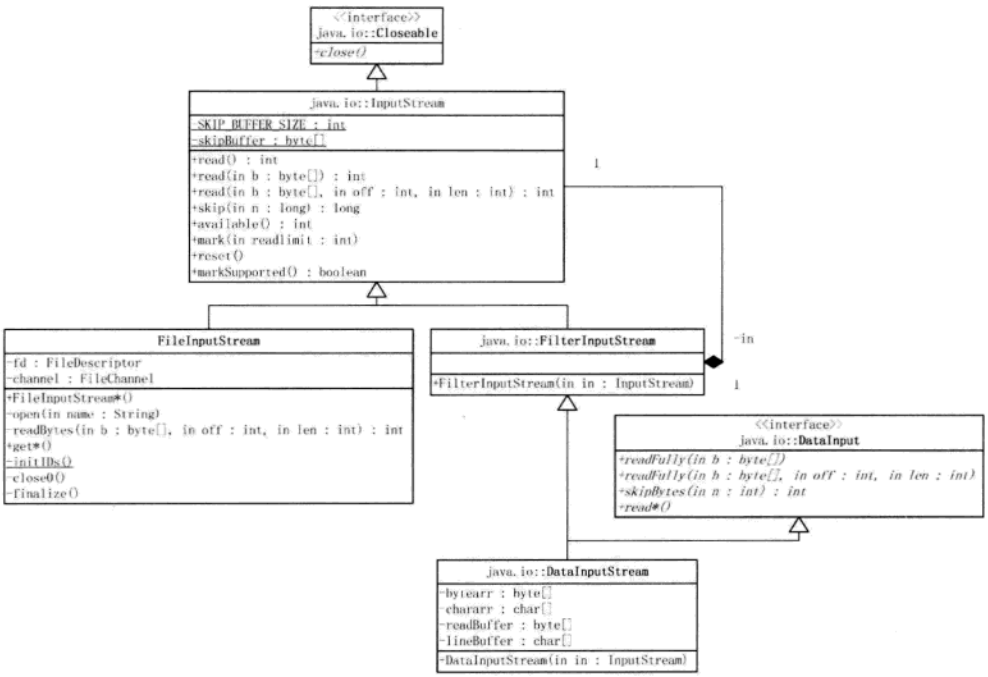



图 5-16 FileInputStream 和 DataInputStream 类图

其中，FileInputStream 在字节层次上读写文件，而输入数据流 DataInputStream 继承自 FilterInputStream，提供读取 Java 基本类型的一系列方法。FilterInputStream 中通过类成员变量 in 保存另一个输入流。

下面的代码演示了如何通过 DataInputStream 在文件中读取 Java 的基本类型：

```
FileInputStream fin=new FileInputStream("block.data");
DataInputStream din=new DataInputStream(fin);
long l=din.readLong();
```

首先，通过构造 FileInputStream 创建了在文件 block.data 上的一个输入流。然后在构造 din 的时候，将 fin 作为构造函数的参数，就这样将 fin 和 din 组合成一个过滤流，如图 5-17 所示。

上面代码的最后部分，通过 DataInputStream 提供的 readLong() 方法，在文件中读入一个长整数。注意，如果传入 DataInputStream 的是一个从 Socket 对象得到的输入流（通过 Socket.getInputStream() 方法可以获得该输入流），其实是在 TCP 连接上读取一个长整数。

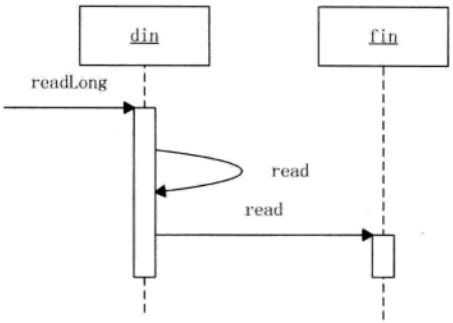


图 5-17　过滤流的序列

Hadoop 中，经常需要向流写入计算结果，或者从流中读取结果。数据流 DataOutputStream 和 DataInputStream 支持写入和读取所有 Java 基本类型的方法，当需要在流中写入一个数值、字符、布尔值或字符串时，可以使用下面 DataOutput 接口提供的方法之一。对于输入流 DataInputStream，则实现相应的接口 DataInput。

```
void write(int b)
void write(byte b[])
void write(byte b[], int off, int len)
void writeBoolean(boolean v)
void writeByte(int v)
void writeShort(int v)
void writeChar(int v)
void writeInt(int v)
void writeLong(long v)
void writeFloat(float v)
void writeDouble(double v)
void writeBytes(String s)
void writeChars(String s)
void writeUTF(String s)
```

数据流广泛使用于 Hadoop 的实现中，如序列化机制 Writable。

### 5.4.4　随机存取文件

与 Linux 文件 API lseek() 对应的 Java 操作是 RandomAccessFile.seek()，而不是流的某一个具体方法。由于流是对数据源的抽象，大部分数据源是不支持随机存取功能的，如来自网络的数据流，为此，Java 引入了 RandomAccessFile 类，用于在（磁盘）文件的任何位置查找或者写入数据。

RandomAccessFile 的实例支持对随机访问文件的读取和写入，通过如下的构造函数，可以创建 RandomAccessFile 对象，其中，对应的文件可以通过指定名称或 File 对象传入，参数 mode 用于指定打开方式，如只读打开（“r”）或者读写打开（“rw”）。方法声明如下：

```
public RandomAccessFile(String name, String mode)
public RandomAccessFile(File file, String mode)
```

RandomAccessFile 提供了一个文件指针，如果把随机访问文件比喻为存储在文件系统中

的一个大型 byte 数组，那文件指针就是数组的光标或索引。输入操作从文件指针开始读取字节，并随着对字节的读取而向前移动此文件指针。输出操作从文件指针开始写入字节，并随着对字节的写入移动文件指针。

通过 getFilePointer() 方法可以返回文件指针的当前位置；而 seek 方法将文件指针设置在文件内部的任何字节位置。需要注意的是，这两个方法都被声明为 native 方法。代码如下：

```
public native long getFilePointer() throws IOException;
public native void seek(long pos) throws IOException;
```

从随机存取文件中读取数据，可以使用与 DataInputStream 对象相同的方法，如 readInt() 和 readChar()；类似的，写入数据有 writeInt() 和 writeChar()。

RandomAccessFile 实现了 DataInputStream 实现的 DataInput 接口，也实现了 DataOutputStream 实现的 DataOutput。

## 5.5 Hadoop 抽象文件系统

为了提供对不同数据访问的一致接口，Hadoop 借鉴了 Linux 虚拟文件系统的概念，引入了 Hadoop 抽象文件系统，并在 Hadoop 抽象文件系统的基础上，提供了大量的具体文件系统的实现，满足构建于 Hadoop 上应用的各种数据访问需求。

通过 Hadoop 抽象文件系统，MapReduce 目前可以运行在基于 HDFS 的集群上，也可以运行在基于 Amazon S3 的云计算环境里。

### 5.5.1 Hadoop 文件系统 API

前面分析 Java 文件 API 的时候了解过 java.io.FileSystem，它是 java.io 包的私有抽象类，不能被包以外的类引用。也就是说，不能像类似于 UnixFileSystem 或 Win32FileSystem 那样，通过继承 java.io.FileSystem，实现一个和 Java 文件系统兼容的，又能访问如 HDFS 的子类[⊖]。

为了解决这个问题，Hadoop 特别提供了一个抽象的文件系统，HDFS 只是这个抽象文件系统的一个具体实现。Hadoop 文件系统抽象类 org.apache.hadoop.fs.FileSystem 的类图如图 5-18 所示。

与 Linux 和 Java 文件 API 类似，Hadoop 抽象文件系统的方法可以分为两部分：一部分用于处理文件和目录的相关事务；另一部分用于读写文件数据。表 5-8 总结了 Hadoop 抽象文件系统的文件操作与 Java、Linux 的对应关系。

⊖ 在 JSR 203: More New I/O APIs for the Java Platform ("NIO.2") 中，除了支持默认文件系统外，还支持可插式文件系统。从长远来看，这个接口可以代替 Hadoop 的抽象类 org.apache.hadoop.fs.FileSystem。JSR 203 可以在 http://jcp.org/en/jsr/detail?id=203 找到。

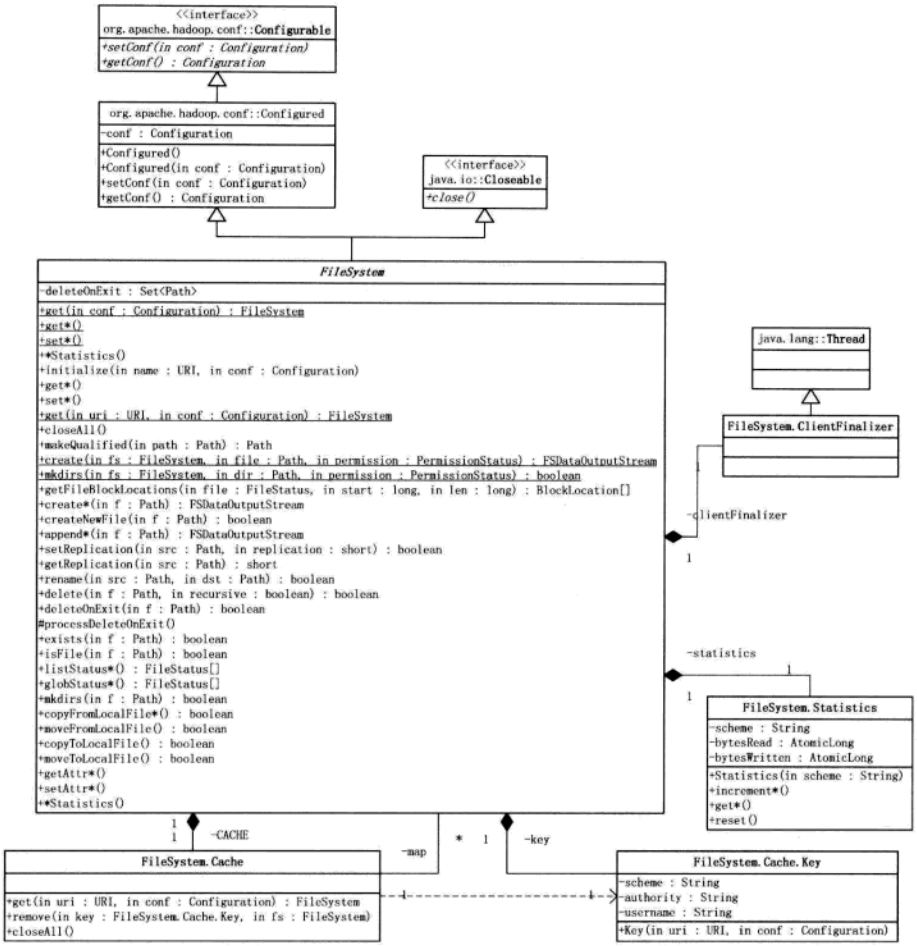



图 5-18　org.apache.hadoop.fs.FileSystem 类图

表 5-8　Hadoop 抽象文件系统文件操作（部分）

| HadoopFileSystem 操作 | Java 操作 | Linux 操作 | 描　　述 |
|---|---|---|---|
| URL.openStream<br>FileSystem.open<br>FileSystem.create<br>FileSystem.append | URL.openStream | open | 打开一个文件 |
| FSDataInputStream.read | InputStream.read | read | 读取文件中包含的数据 |
| FSDataOutputStream.write | OutputStream.write | write | 向文件中写入数据 |
| FSDataOutputStream.close<br>FSDataInputStream.close | InputStream.close<br>OutputStream.close | close | 关闭一个文件 |
| FSDataInputStream.seek | RandomAccessFile.seek | lseek | 改变文件读写位置 |

（续）

| HadoopFileSystem 操作 | Java 操作 | Linux 操作 | 描　述 |
|---|---|---|---|
| FileSystem.getFileStatus FileSystem.get* | File.get* | stat | 获取文件 / 目录的属性 |
| FileSystem.set* | File.set* | chmod 等 | 修改文件属性 |
| FileSystem.createNewFile | File.createNewFile | create | 创建一个文件 |
| FileSystem.delete | File.delete | remove | 从文件系统中删除一个文件 |
| FileSystem.rename | File.renameTo | rename | 更改文件 / 目录名 |
| FileSystem.mkdirs | File.mkdir | mkdir | 在给定目录下创建一个子目录 |
| FileSystem.delete | File.delete | rmdir | 从一个目录中删除一个空的子目录 |
| FileSystem.listStatus | File.list | readdir | 读取一个目录下的项目 |
| FileSystem. setWorkingDirectory | | getcwd/getwd | 返回当前工作目录 |
| FileSystem. setWorkingDirectory | | chdir | 更改当前工作目录 |

通过上表对比可以发现，在处理文件和目录相关事务方面，Hadoop 抽象文件系统和 Java 文件 API 有大量相似之处，如删除文件、文件改名、创建子目录等。不同之处是调用的格式有所不同，以删除文件为例，在 Java 中，通过 File 的无参数方法删除文件：

```
public boolean delete()
```

在 Hadoop 抽象文件系统中，可以使用 FileSystem.delete() 删除文件。其中，参数 f 是要删除的文件，它的类型是 Path：

```
public abstract boolean delete(Path f, boolean recursive)
```

org.apache.hadoop.fs.Path 是一个很简单的类，它在 java.net.URI 的基础上抽象了文件系统中的路径。需要注意的是，Java 中没有对应的 Path 类。但是，在 Java 中，java.io.File 提供了 toURL() 方法，用于获得对应的使用 file 模式的 URL。而从一个使用 file 模式的 URL 对象，可以通过 URLDecoder.decode() 构造对应的 File 对象。但是，Java 的 File 类和 URL 类分别抽象了不同的事物，它们都具有各自的不同的内部状态和行为。

这一部分的 API 中，Hadoop FileSystem 和 Java 文件 API 比较大的差别，是获取文件或目录属性的方法。通过 FileSystem.getFileStatus() 方法，Hadoop 抽象文件系统可以一次获得文件 / 目录的所有属性，这些属性被保存在类 FileStatus 中，如图 5-19 所示。

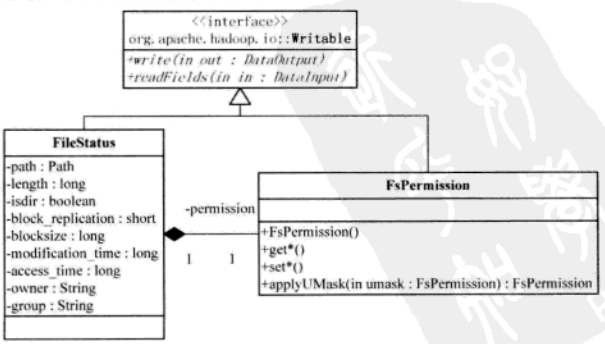



图 5-19　FileStatus 类图

通过 FileStatus 可以获得的信息包括：文件路径 path、文件长度

length、是否是目录 isdir、副本数 block_replication（这是为了 HDFS 而准备的特殊参数）、块大小 blocksize（为了 HDFS 准备的特殊参数）、最后修改时间 modification_time、最后访问时间 access_time、许可信息 permission、文件所有者 owner、用户组 group。具体代码如下：

```
public class FileStatus implements Writable, Comparable {

   private Path path;
   private long length;
   private boolean isdir;
   private short block_replication;
   private long blocksize;
   private long modification_time;
   private long access_time;
   private FsPermission permission;
   private String owner;
   private String group;
   ……
}
```

FileStatus 实现了 Writable 接口，这就是说，FileStatus 可以被序列化后在网络上传输。同时，一次性将文件的所有属性读出并返回到客户端，可以减少在分布式系统中进行网络传输的次数，应该说，这是一个相当有针对性的设计。

出现在 FileSystem 中的，但在 Java 文件 API 中找不到对应的方法有 getContentSummary()、setReplication() 和 getReplication()，其声明如下：

```
public boolean setReplication(String src, short replication)
      throws IOException;
……
@Deprecated
public short getReplication(Path src) throws IOException
……
public ContentSummary getContentSummary(String path)
      throws IOException;
```

FileSystem.getContentSummary() 提供了类似 Linux 命令 du、df 提供的功能。du 表示“disk usage”，它会报告特定的文件和每个子目录所使用的磁盘空间大小；命令 df 则是“diskfree”的缩写，用于显示文件系统上已用的和可用的磁盘空间的大小。du、df 是 Linux 中查看磁盘和文件系统状态的重要工具。

getContentSummary() 方法的输入是一个文件或目录的路径，输出是该文件或目录的一些存储空间信息，这些信息定义在 ContentSummary，如图 5-20 所示，包括文件大小、文件数、目录数、文件配额，已使用空间和已使用文件配额等。

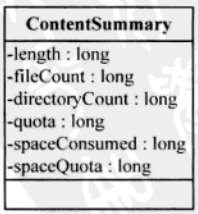



图 5-20　ContentSummary 中包含的内容

getContentSummary() 方法不是 FileSystem 的抽象方法，它可以在 getFileStatus() 的基础上，计算出 ContentSummary 中包含

的信息。

setReplication() 方法用于设置文件系统上的文件副本数，getReplication() 方法则可获得文件的副本数。这是两个有 HDFS 特色的方法。

实现一个 Hadoop 具体文件系统，需要实现哪些功能呢？下面整理了 org.apache.hadoop.fs.FileSystem 中的抽象方法：

```
// 获得文件系统 URI
public abstract URI getUri();

// 为读打开一个文件，并返回一个输入流，下一节讨论
public abstract FSDataInputStream open(Path f, int bufferSize)
        throws IOException;

// 创建一个文件，并返回一个输出流，下一节讨论
public abstract FSDataOutputStream create(Path f,
        FsPermission permission,
        boolean overwrite,
        int bufferSize,
        short replication,
        long blockSize,
        Progressable progress) throws IOException;

// 在一个已经存在的文件中追加数据
public abstract FSDataOutputStream append(Path f, int bufferSize,
        Progressable progress) throws IOException;

// 修改文件名或目录名
public abstract boolean rename(Path src, Path dst) throws IOException;

// 删除文件
public abstract boolean delete(Path f) throws IOException;
public abstract boolean delete(Path f, boolean recursive)
        throws IOException;

// 如果 Path 是一个目录，读取一个目录下的所有项目和项目属性；
// 如果 Path 是一个文件，获取文件属性
public abstract FileStatus[] listStatus(Path f) throws IOException;

// 设置当前的工作目录
public abstract void setWorkingDirectory(Path new_dir);

// 获取当前的工作目录
public abstract Path getWorkingDirectory();

// 如果 Path 是一个文件，获取文件属性
public abstract boolean mkdirs(Path f, FsPermission permission)
        throws IOException;

// 获取文件或目录的属性
```

```
public abstract FileStatus getFileStatus(Path f) throws IOException;
```

上述抽象方法基本覆盖了表 5-8 中用于处理文件和目录相关事务的主要操作，实现一个具体的文件系统，至少需要实现上面这些抽象方法。另一方面，Hadoop 抽象文件系统基于以上方法，提供了一些工具方法，以方便用户调用。如 listStatus() 方法，org.apache.hadoop.fs.FileSystem 就提供了另外三种形式，由于这些方法的实现都非常简单，不再一一介绍了。

## 5.5.2　Hadoop 输入 / 输出流

上一节已经介绍了 Hadoop 抽象文件系统处理文件和目录相关事务的 API，和 Java 文件系统类似，Hadoop 抽象文件系统也是使用流机制进行文件的读写。Hadoop 抽象文件系统中，用于读文件数据的流是 FSDataInputStream，它是一个抽象类，对应地，写文件通过抽象类 FSDataOutputStream 实现。

### 1. FSDataInputStream

输入流 FSDataInputStream 通过 FileSystem.open() 方法打开，输出流 FSDataOutputStream 可以通过创建文件的 FileSystem.create() 打开，或通过在已有文件后添加数据的 FileSystem.append() 打开。

FSDataInputStream 的类图如图 5-21 所示。它继承了 DataInputStream 类，并实现了 Seekable、PositionedReadable 和 java.io.Closeable 接口（没有体现在类图中）。其中，DataInputStream 前面已经讨论过，它提供了读取 Java 基本类型的方法族。

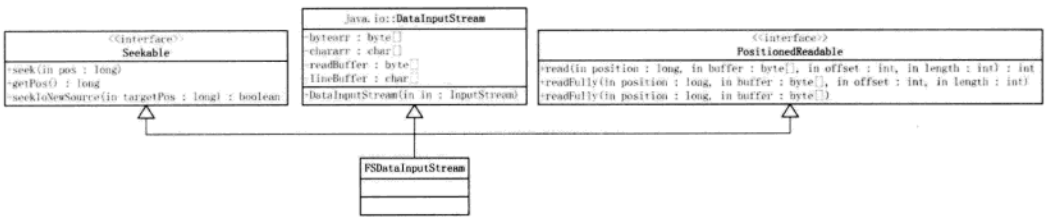



图 5-21　FSDataInputStream 类结构

Seekable 接口提供在（文件）流中进行随机存取的方法，其功能类似于 RandomAccessFile 中的 getFilePointer() 方法（Seekable.getPos() 方法）和 seek() 方法，它提供了某种随机定位文件读取位置的能力。

Seekable 接口中的 seekToNewSource() 是一个比较特殊的方法，当文件数据有多个副本的时候，如在 HDFS 中，seekToNewSource() 可用于重新选择一个副本。代码如下：

```
/** 接口，用于支持在流中定位 */
public interface Seekable {
    // 将当前偏移量设置到参数位置，下次读数据将从该位置开始
    void seek(long pos) throws IOException;

    // 得到当前偏移量
    long getPos() throws IOException;
```

```
    // 重新选择一个副本
    boolean seekToNewSource(long targetPos) throws IOException;
}
```

FSDataInputStream 实现的另一个接口是 PositionedReadable，它提供了从流中某一个位置开始读数据的一系列方法，这些方法的第一个参数 position，用于指定流中的一个位置。其中，read() 和 readFully() 的差别在于：一个试图读取指定长度的数据，另一个读取指定长度的数据，直到读满缓冲区或流结束。接口如下：

```
// 接口，用于在流中进行定位读
public interface PositionedReadable {
    // 从指定位置开始，读最多指定长度的数据到 buffer 中 offset 开始的缓冲区中
    // 注意，该函数不改变读流的当前位置，同时，它是线程安全的
    public int read(long position, byte[] buffer, int offset, int length)
            throws IOException;

    // 从指定位置开始，读指定长度的数据到 buffer 中 offset 开始的缓冲区中
    public void readFully(long position, byte[] buffer, int offset,
            int length) throws IOException;

    public void readFully(long position, byte[] buffer) throws IOException;
}
```

如果流已经实现了 Seekable 接口，还需要实现 PositionedReadable 接口吗？答案是肯定的，注意，PositionedReadable 中的 3 个读方法，都不会改变流的当前位置，而且还是线程安全的。当然，你可以通过一些同步机制，通过 Seekable 接口实现 PositionedReadable 接口中的方法。

### 2. FSInputStream

在 org.apache.hadoop.fs 包中，还包括了抽象类 FSInputStream。Seekable 接口和 PositionedReadable 中的方法都成为这个类的抽象方法。

在 FSInputStream 类中，通过 Seekable 接口的 seek() 方法实现了 PositionedReadable 接口中的 read() 方法。过程很简单，在进行该 read() 操作前，利用 Seekable.getPos() 保存当前读的位置，然后，调用 Seekable.seek() 设置读位置 position 并通过 InputStream.read() 读取数据，读取完数据后，再次调用 Seekable.seek() 恢复 InputStream.read() 前的位置。

注意，由于 PositionedReadable.read() 是线程安全的，所以该方法需要借助于 synchronized (this)，保证方法被调用时，其他方法不会被调用，也就保证不会有其他线程改变 Seekable.getPos() 保存的当前读位置，这样，最后的位置恢复才是正确并有意义的。相关代码如下：

```
public abstract class FSInputStream extends InputStream
    implements Seekable, PositionedReadable {
    ......
    public int read(long position, byte[] buffer, int offset, int length)
            throws IOException {
        synchronized (this) {
```

```
            long oldPos = getPos(); //保存当前位置
            int nread = -1;
            try {
               seek(position); //移动读数据的位置
               nread = read(buffer, offset, length);
            } finally {
               seek(oldPos); //返回前面保存的位置
            }
            return nread;
         }
      }
      ……
   }
```

FSInputStream 有 10 多个子类，它们都是具体文件系统实现的具体输入流，如用于读取 HDFS 文件的 org.apache.hadoop.hdfs.DFSInputStream，在适当的时候，会介绍这些具体输入流的实现。

**注意**　Hadoop 中并没有和 FSInputStream 对应的 FSOutputStream。这部分的 Hadoop 代码不是很一致。

### 3. FSDataOutputStream

FSDataOutputStream 用于写数据，和 FSDataInputStream 类似，它继承自 DataOutputStream，提供 writeInt() 和 writeChar() 等方法。类结构如图 5-22 所示。

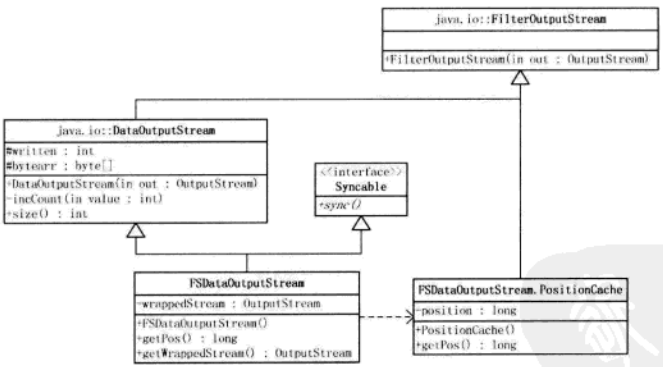



图 5-22　FSDataInputStream 类结构

相对于 FSDataInputStream，FSDataOutputStream 比较简单，它没有实现 Seekable 接口，也就是说，Hadoop 文件系统不支持随机写，用户不能在文件中重新定位写位置，并通过写数据来覆盖文件原有的内容。但用户可以通过 getPos() 方法获得当前流的写位置，为了实现 getPos() 方法，FSDataOutputStream 定义了内部类 PositionCache，该类继承自 FilterOutputStream，并通过重载 write() 方法跟踪目前流的写位置。

PositionCache 是一个典型的过滤器流，在基础的流功能上添加了 getPos() 方法，同时，利用 FileSystem.Statistics 实现了文件系统读写的一些统计。而 FSDataOutputStream 的构造函数也很干脆，直接生成 PositionCache 对象并调用父类构造方法。代码如下：

```
public class FSDataOutputStream extends DataOutputStream
                                implements Syncable {
   private OutputStream wrappedStream;

   private static class PositionCache extends FilterOutputStream {
      private FileSystem.Statistics statistics;
      long position; // 当前流的写位置

      public PositionCache(OutputStream out,
                           FileSystem.Statistics stats,
                           long pos) throws IOException {
      super(out);
      statistics = stats;
      position = pos;
   }

   public void write(int b) throws IOException {
      out.write(b);
      position++;// 更新当前位置
      if (statistics != null) {
            statistics.incrementBytesWritten(1);// 更新文件统计值
         }
      }
      ……
      public long getPos() throws IOException {
         return position;// 返回当前流的写位置
      }
      ……
   }
   ……
   public FSDataOutputStream(OutputStream out,…….Statistics stats,
         long startPosition) throws IOException {
      super(new PositionCache(out, stats, startPosition));
      wrappedStream = out;
   }
   ……
}
```

FSDataOutputStream 实现了 Syncable 接口，该接口只有一个函数 sync()，其目的和 Linux 中系统调用 sync() 类似，用于将流中保存的数据同步到设备中。

细心的读者会发现，FSDataInputStream 继承的 DataInputStream，和 FSDataOutputStream 继承的 DataOutputStream，都只是对其他流进行增强。通过对 FSDataInputStream 和 FSDataOutputStream 构造函数的分析也可以发现，它们需要依赖于具体的、能够接收或发送数据的流来构造相应的对象。传递给这些构造函数的输入 / 输出流，一般由具体的文件系统

实现来提供。代码如下：

```
public FSDataInputStream(InputStream in)
public FSDataOutputStream(OutputStream out)
public FSDataOutputStream(OutputStream out,
                          FileSystem.Statistics stats)
public FSDataOutputStream(OutputStream out,
                          FileSystem.Statistics stats,
                          long startPosition)
```

**注意**　代码中“FSDataOutputStream”表示这个方法后面会废弃不用。

## 5.5.3　Hadoop 文件系统中的权限

Hadoop 分布式文件系统实现了一个和 Linux 系统类似的权限模型。

每个文件和目录都有一个所有者（owner）和一个组（group），在 FileStatus 类中，可以通过 getOwner() 和 getGroup() 获得相应的属性。

图 5-23 给出了 Hadoop 文件系统中与权限相关的类。

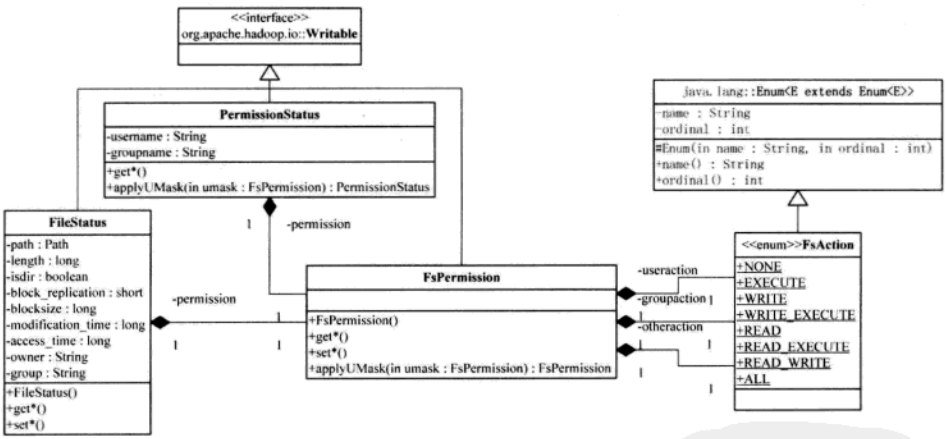



图 5-23　Hadoop 文件系统中与权限相关的类

用户身份机制对 Hadoop 文件系统来说只是外部特性。目前，Hadoop 并不提供创建用户身份、用户组或处理用户凭证等功能。也正是这个原因，图 5-23 中，FileStatus 类使用了字符串 owner 和 group，保存了文件所有者和文件所在用户组的信息，代码如下：

```
public class FileStatus implements Writable, Comparable {
    ......
    private FsPermission permission;
    private String owner; // 文件所有者
    private String group; // 文件所在用户组
    ......
}
```

而在 Linux 系统中，文件或目录元信息通过整型的用户 ID 和组 ID，保存文件所有者和文件所在用户组。Linux 系统的实现方式比较节省存储空间，但需要增加用户 ID 到用户名、组 ID 到组的映射关系，并提供命令维护该关系。

FsPermission 保存了 FileStatus 对应文件的权限，也是采用 Linux 的表示和显示习惯，包括使用八进制数来表示权限。当新建一个文件或目录时，它的所有者是客户进程的用户，它所属组是父目录的组。文件或目录的文件所有者、文件组和其他用户的权限信息，可以保存在 FsPermission 对象中。应该说，FsPermission 是一个简单的包装类，PermissionStatus 则在 FsPermission 的基础上包含了文件的文件所有者和文件所在用户组的信息，其实现也很简单。

和 Linux 系统一样，文件或目录对其所有者、同组的其他用户以及所有其他用户可以分别有着不同的读、写和执行权限，也就是前面讨论的 rwx 权限。在 Hadoop 中，rwx 权限通过枚举 FsAction 定义。关键字 enum 是在 Java 1.5 中引入的新特性，通过该关键字，用户可以方便地定义一个常量集合。对于枚举类型，Java 内部将其转换为 java.lang.Enum 的子类（如图 5-23 所示），并提供了一些方法。其中，成员变量 ordinal 用于返回枚举常量的序数，也就是它在枚举声明中的位置，初始常量序数为 0。FsAction 中权限常量的定义利用了枚举常量序数，如 FsAction.NONE，它是这个枚举类型的第一个枚举值，它的序数为 0，正好也是对应 Linux 形式权限的表示 000，而 WRITE_EXECUTE，其表示为 011，序数为 3。

FsAction 还提供了 implies() 方法，用于判断权限 1 是否隐含权限 2，如权限 FsAction.WRITE_EXECUTE 隐含了权限 FsAction.WRITE 和 FsAction.EXECUTE，但不隐含权限 FsAction.READ_EXECUTE。利用这些权限的枚举常量序数，可以很容易实现 implies() 方法。FsAction 类的主要代码如下：

```
public enum FsAction {
   //POSIX 风格的权限
   NONE("---"),
   EXECUTE("--x"),
   WRITE("-w-"),
   WRITE_EXECUTE("-wx"),
   READ("r--"),
   READ_EXECUTE("r-x"),
   READ_WRITE("rw-"),
   ALL("rwx");
   ……
   public boolean implies(FsAction that) {
      if (that != null) {
         return (ordinal() & that.ordinal()) == that.ordinal();
      }
      return false;
   }
}
```

## 5.5.4 抽象文件系统中的静态方法

org.apache.hadoop.fs.FileSystem 中提供的一系列抽象方法定义了 Hadoop 抽象文件系统

的系统界面，即一个文件系统需要提供的服务。FileSystem 中还有另外一类值得研究的方法，这些方法都是静态方法，它们提供了获得具体文件系统的实例，或者关闭文件系统等重要方法；也提供如 create() 打开文件、mkdir() 创建目录等工具方法；文件系统的一些统计数据，如文件读操作数、读字节数、写操作数、写字节数，也是通过静态方法提供的。

FileSystem 中获得具体文件系统主要通过 get() 方法，它有多种重载形式，下面是一个例子（在 org.hadoopinternal.fs.RamFileSystemDemo 的 main() 方法中）：

```
......
URI uri = URI.create("ramfs://demo");
Configuration conf = new Configuration();
InMemoryFileSystem inMemFs
      =  (InMemoryFileSystem)FileSystem.get(uri, conf);
......
```

实际上，get() 是典型的工厂模式实现，用于创建多种“产品”，在 FileSystem 中，这些产品就是具体的文件系统，如 Hadoop 分布式文件系统、内存文件系统等。以上述代码为例，get() 方法需要一个 URI，文件系统根据这个 URI 的模式“ramfs”，了解到用户需要打开一个内存文件系统，则创建或返回对应的文件系统。getLocal() 用于获取本地文件系统，是 get() 的一个工具方法，用法类似于 get()。下面的代码来自 org.hadoopinternal.fs.LocalFileSystemDemo：

```
......
Configuration conf = new Configuration();
LocalFileSystem localFs = FileSystem.getLocal(conf);
......
```

有了 get() 方法，getLocal() 的实现非常简单，其中，常量 LocalFileSystem.NAME 的值是“file:///”：

```
public static LocalFileSystem getLocal(Configuration conf)
      throws IOException {
   return (LocalFileSystem)get(LocalFileSystem.NAME, conf);
}
```

几种形式的 FileSystem.get() 最终调用下面的 get() 方法，它的输入参数包括要打开文件系统的 URI 和打开时使用的配置：

```
public static FileSystem get(URI uri, Configuration conf) throws IOException
{
   String scheme = uri.getScheme();// 获得 URI 模式
   String authority = uri.getAuthority();// 鉴权信息

   if (scheme == null) {//URI 模式为空，返回默认文件系统
      return get(conf);
   }
   ......
   String disableCacheName // 是否使用被 Cache 的文件系统
         = String.format("fs.%s.impl.disable.cache", scheme);
```

```
    if (conf.getBoolean(disableCacheName, false)) {
      return createFileSystem(uri, conf);// 创建对应文件系统
    }

    return CACHE.get(uri, conf);// 在 Cache 中获取对应文件系统
  }
```

该方法通过 getScheme() 获得文件系统的 URI 模式，如果模式为空，则返回默认文件系统。默认文件系统的 URI 模式可以通过配置项 ${fs.default.name} 进行配置，该配置项的默认值是 file:///，该默认值和常量 LocalFileSystem.NAME 相同，即默认文件系统的默认配置是本地文件系统 LocalFileSystem。

FileSystem.get() 处理过程中另一个可配置并需要判断的条件是：是否使用 FileSystem.CACHE 中保存的、已经缓存的文件系统。打开一个文件系统是一个比较耗费资源的操作，如打开 HDFS，需要和名字节点建立 IPC 通信，所以，共享文件系统是一个不错的优化。但是，由于文件系统间是相互共享的，应用不小心关闭共享的文件系统，将会影响其他使用者。所以，Hadoop 引入了一系列的配置项 ${fs.%s.impl.disable.cache}，其中，%s 在使用过程中，会被替换成相应的 URI 模式。如果该配置项为 true（默认值为 false），则每次创建一个新的具体文件系统实例；否则，使用共享的文件系统实例。以 HDFS 为例，它的模式为 hdfs，对应的配置项就是 ${fs.hdfs.impl.disable.cache}，该配置项决定，如果 FileSystem.get() 获取的是（可共享的）HDFS 文件系统，方法返回一个新的 HDFS 文件系统实例，或者是共享 HDFS 文件系统实例。

如果不共享文件实例，需要通过 FileSystem 的私有方法 createFileSystem() 创建相应的文件系统。代码如下：

```
  private static FileSystem createFileSystem(URI uri,Configuration conf)
          throws IOException {
    // 获取文件系统实现的 Class 类对象
    Class<?> clazz = conf.getClass("fs." + uri.getScheme() + ".impl", null);
    ......
    if (clazz == null) {
      thrownew IOException("No FileSystem for scheme: " + uri.getScheme());
    }

    // 通过反射创建对象，并初始化文件系统
    FileSystem fs = (FileSystem)ReflectionUtils.newInstance(clazz, conf);
    fs.initialize(uri, conf);
    return fs;
  }
```

FileSystem.createFileSystem() 其实很简单，通过 Configuration.getClass() 获得具体文件系统实现对应的 Class 类对象，并进一步创建文件系统对象。再次以 HDFS 为例，它的 URI 模式为 hdfs，对应配置项名字为 fs.hdfs.impl，一般来说，该配置项的值为 org.apache.hadoop.hdfs.DistributedFileSystem，根据这个值，Configuration.getClass() 返回它对应的 Class 类对象，保存在局部变量 clazz 中。然后，createFileSystem() 用反射工具类 ReflectionUtils

中的 newInstance() 方法创建具体文件系统对象，即 DistributedFileSystem 实例。如同第 3 章所分析的，ReflectionUtils.newInstance() 创建对象后，如果对象实现了 Configurable 接口，会调用该接口的 setConf() 方法配置对象，使用的是 newInstance() 的参数 conf 中保存的配置信息。随后，具体文件系统的 initialize() 方法被调用，完成构造后的初始化工作。该方法默认实现是构造用于统计的 FileSystem.Statistics 对象，该对象会保存在文件系统上的相关操作的计数。

回到 FileSystem.get() 方法的分析，如果方法允许共享文件系统实例，则调用 CACHE.get() 获取文件系统实例。FileSystem 的内部类 Cache 保存了已经打开的、可以共享的文件系统，其实现不是很复杂。唯一需要注意的是可共享的条件，这个条件体现在 FileSystem.CACHE.Key 中（在讨论 Hadoop IPC 的时候也有一个类似的结构），代码如下：

```
static class Key {
   final String scheme;
   final String authority;
   final long unique;
   final UserGroupInformation ugi;
   ……
}
```

也就是说，只有在上述这 4 个值都相等的情况下，文件实例才会被共享。其中，schema 是 URI 模式，这个很自然要相等；其次是 URI 的 authority 部分，用相同授权打开的具体文件系统才能共享，用户可以打开 URI 模式相同，但有不同授权的具体文件系统；unique 是一个整数，如果没有特殊情况，默认为 0；类型为 UserGroupInformation 的变量 ugi，则保存了打开具体文件系统的本地用户的信息，也就是说，即使其他 3 部分都相等，不同本地用户打开的具体文件系统也是不能共享的。FileSystem.CACHE.Key 通过这 4 个字段，明确地定义了可共享具体文件系统的条件。

其中，unique 字段比较特殊，它提供了一种机制，如果用户需要创建其他 3 个字段都相同，但又不共享的具体文件系统时，可以使用该字段，以作区分。FileSystem.CACHE.getUnique() 实现了这个机制，通过它调用者可以得到一个被 Cache 管理的，且不被共享的具体文件系统，代码如下。其中 getInternal() 会调用前面介绍的 createFileSystem() 函数创建具体文件系统：

```
static class Cache {
   private final ClientFinalizer clientFinalizer = new ClientFinalizer();
   ……
   /** 用于 unique 字段的自增变量 */
   private static AtomicLong unique = new AtomicLong(1);
   ……
   /** 得到一个被 Cache 管理的，不被共享的具体文件系统 */
   synchronized FileSystem getUnique(URI uri, Configuration conf)
         throws IOException{
      Key key = new Key(uri, conf, unique.getAndIncrement());
      return getInternal(uri, conf, key);
```

```
    }
}
```

FileSystem.newInstance() 调用 FileSystem.CACHE.getUnique()，它和 FileSystem.get() 类似，只不过 newInstance() 返回的，永远是不被共享的，但又被 FileSystem.CACHE 对象管理的文件系统。读者可能会问，为什么文件系统要被 FileSystem.CACHE 管理呢？简单地说，FileSystem.CACHE 对象提供了清理机制，如果用户打开文件系统后不关闭，FileSystem.CACHE 会利用 Runtime.addShutdownHook() 机制，在 Java 虚拟机退出时，关闭被打开的文件系统。这样，就不会出现因为文件系统没有关闭导致的数据丢失、数据不一致的情况。这部分功能的相关代码在 FileSystem.CACHE.getInternal() 中，由于篇幅的原因，就不再继续探讨了。另外，由于 FileSystem.newInstance() 方法永远都返回一个不被共享的具体文件系统实例，在抽象文件系统的 close() 方法中，会调用 FileSystem.CACHE.remove()，在 CACHE 对象中移除对应的文件系统，避免 Java 虚拟机退出时再次被关闭，代码如下：

```
public void close() throws IOException {
    processDeleteOnExit();
    CACHE.remove(this.key, this);// 从 CACHE 中删除文件系统实例
    ……
}
```

总的来说，利用 Java 的反射机制，FileSystem.get() 和 FileSystem.newInstance() 的实现过程还是比较简单并且具有一定的可扩展性的。抽象文件系统 FileSystem 中的其他静态方法，包括工具方法和对文件系统度量信息的收集等实现都不复杂，也就不再赘述。

### 5.5.5 Hadoop 文件系统中的协议处理器

前面在介绍 Java 的 URL 类时提到，通过 URL.openStream() 可以获得到某一个资源的输入流，在下面的例子中，可以打开 HDFS 上的某个文件的输入流，然后读取数据：

```
InputStream in=null;
try {
    FsUrlStreamHandlerFactory factory =
          new org.apache.hadoop.fs.FsUrlStreamHandlerFactory();
    // 安装协议处理系统
    java.net.URL.setURLStreamHandlerFactory(factory);

    in = new URL("hdfs://example:port/path/file").openStream();
    ……// 后续处理
} finally {
    IOUtils.closeStream(in);
}
```

如何通过 URL 顺利打开一个特定 URL 模式的输入 / 输出流，并将这个机制和 Hadoop FileSystem 结合起来，是本节要研究的内容。它和在构建可扩展浏览器的体系结构时，如何在浏览器中处理新的协议面临的问题是一样的。用户如果在支持 hdfs 模式的浏览器地址栏输入 URL“hdfs://example:port/path/media.rmvb”后，浏览器应能正确播放存放在 HDFS 上

的流媒体文件“media.rmvb”（当然，目前没有浏览器支持 hdfs 模式）。如何支持上述功能，Java 语言的网络库设计时，将该过程分成两部分的内容：协议处理和内容处理。

协议处理涉及客户端和服务器间的交互，以 HDFS 模式为例，它必须能够与 NameNode 和 DataNode 通信，以获取数据；内容处理则涉及将协议处理过程中获得的内容进行展现的各个方面，如播放 rmvb 格式的媒体。

Java 将协议处理任务分成几个部分，java.net 包中的 4 个类 URL、URLStreamHandler、URLConnection 和 URLStreamHandlerFactory 一起实现了协议处理器机制。具体类参见图 5-24。

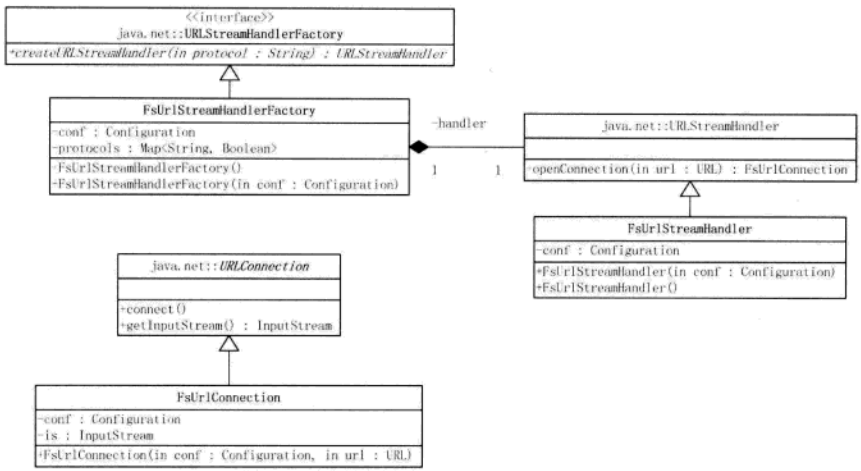



图 5-24　URL 协议处理器

其中，URL 是这 4 个类中唯一的具体类，而 URLStreamHandler 和 URLConnection 是抽象类，URLStreamHandlerFactory 则是接口。为了提供上面提到的通过 hdfs 模式的 URL 打开输入 / 输出流，必须编写 URLStreamHandler 和 URLConnection 的具体子类；同时，为了使用这些子类，开发人员还必须编写一个实现了 URLStreamHandlerFactory 接口的类。在 Hadoop 文件系统中，它们分别是 FsUrlStreamHandler、FsUrlConnection 和 FsUrlStreamHandlerFactory。

假设实现了上述 3 个类，通过诸如“hdfs://example:port/……”的 URL 地址构造 URL 对象，并调用 URL.openStream() 获得相应的 InputStream 对象时，到底发生了什么？由于我们构造的 URL 模式是 hdfs，在 URL 的构造函数中，该模式会被传递到通过 URL.setURLStreamHandlerFactory() 注册的，实现了 URLStreamHandlerFactory 接口的具体对象（在 Hadoop 文件系统中，就是 FsUrlStreamHandlerFactory 的实例），此工厂的任务就是接受协议 hdfs，并为该协议找到适当的 URLStreamHandler，如 FsUrlStreamHandler，创建该流处理器的对象并保持在 URL 对象的内部成员变量中。

**注意** 每个应用程序至多有一个 URLStreamHandlerFactory，一旦安装，试图安装另一个时会抛出一个 Error。所以，如同本节开始的例子，用户必须手动安装 FsUrlStreamHandlerFactory，Hadoop 默认不提供 URL.openStream() 支持具体文件系统模式的功能。

URL 对象有了流处理器，就可以要求流处理器解析 URL 字符串，创建一个知道如何使用此协议和服务器通信的 URLConnection 子类，在 Hadoop 中，该子类为 FsUrlConnection。所以，URLStreamHandler 和 URLConnection 的具体子类总是成对出现。在一些比较复杂的场景里，流处理器必须解析 URL，以实现该协议定义的新的 URL 格式。

调用 getInputStream() 时，实际调用的是 URLConnection 中的同名方法，由该方法返回能获得 URL 数据所需的输入流。也就是说，在上面的例子中，FsUrlConnection.getInputStream() 会被调用，并打开到对应 URL 的一个输入流。至此，构造 URL 并打开到对应资源的过程结束。

了解了一般性过程后，我们来分析具体代码。

首先是 FsUrlStreamHandlerFactory，它根据输入的 URL 模式返回相应的 URLStreamHandler。在 Hadoop 中，只实现了一个 FsUrlStreamHandler 子类，这个子类采用了 FileSystem.get() 实现中的机制，非常灵活，能够根据系统的设置支持实现了 Hadoop 抽象文件系统，并在配置系统中进行了配置的多个具体文件系统 URL 模式。所以，不需要为每一个具体文件系统编写对应的 URLStreamHandler 子类，FsUrlStreamHandlerFactory 和 FsUrlStreamHandler 能够根据配置和 URL 模式做出正确的判断。

FsUrlStreamHandlerFactory 的成员变量 handler 在构造时，就被赋予一个 FsUrlStreamHandler 实例，成员变量 protocols 则是一个映射表，这也是一个类似于 FileSystem.CACHE 的优化，保存了是否支持特定 URL 模式的判断结果。

FsUrlStreamHandlerFactory 有两个构造函数，它们的差别在于：其中一个可以提供默认的 Configuration 对象。由于 Configuration 有延迟加载（具体请参考第二章的相关内容），所以，在构造函数中调用 this.conf.getClass()，以确保 Configuration 对象加载了配置项，然后利用该 Configuration 对象构造一个万能的 FsUrlStreamHandler 对象。代码如下：

```
public class FsUrlStreamHandlerFactory implements
        URLStreamHandlerFactory {
   //Hadoop Configuration，用于初始化 FsUrlStreamHandler 对象 .
   private Configuration conf;

   // 是否支持 URL 模式
   private Map<String, Boolean> protocols = new HashMap<String, Boolean>();

   // 工厂使用的 URLStreamHandler 实例
   private java.net.URLStreamHandler handler;
   ......
   public FsUrlStreamHandlerFactory(Configuration conf) {
      this.conf = new Configuration(conf);
      // 加载配置文件，注意，Configuration 实现了延迟加载技术
```

```
        this.conf.getClass("fs.file.impl", null);
        this.handler = new FsUrlStreamHandler(this.conf);
    }

    public java.net.URLStreamHandler createURLStreamHandler(String
            protocol) {
        if (!protocols.containsKey(protocol)) {
            boolean known =
                    (conf.getClass("fs." + protocol + ".impl", null) != null);
            protocols.put(protocol, known);
        }
        if (protocols.get(protocol)) {
            return handler;
        } else {
            // 不知道如何处理这个协议，让虚拟机继续处理
            return null;
        }
    }
}
```

当 URL 对象构造时，需要根据 URL 模式调用 FsUrlStreamHandlerFactory.createURL-StreamHandler()，上面的例子中，URL 模式 hdfs（以字符串的形式）会作为参数，传递给该方法，其处理过程是：首先，如果前面没处理过这个 URL 模式，需要判断目前系统的配置是否支持该 URL。这个判断是通过 Configuration.getClass() 来完成的（上一节已经分析过类似代码的工作原理），判断的结果会保存在成员变量 protocol 中。有了这个判断结果，后续的处理就简单了，从 protocols 中获取 URL 是否被支持的判断，如果是，返回那个万能的 FsUrlStreamHandler 对象；否则，返回 null，由 Java 虚拟机继续处理。一般来说，如果传入的是“http://……”，这一步便会返回 null，并由虚拟机返回 URL 模式为 http 的对应 java.net.URLStreamHandler 实现对象。

FsUrlStreamHandler 的实现更简单，主要方法是 openConnection()，它返回一个新的 FsUrlConnection 对象。和 FsUrlStreamHandler 类似，FsUrlConnection 也可以处理多种模式，相关代码如下：

```
class FsUrlStreamHandler extends URLStreamHandler {
    ......
    private Configuration conf;
    ......
    FsUrlStreamHandler(Configuration conf) {
        this.conf = conf;
    }
    ......
    @Override
    protected FsUrlConnection openConnection(URL url) throws IOException {
        return new FsUrlConnection(conf, url);
    }
}
```

FsUrlConnection 的 getInputStream() 在 URL.openStream() 的实现逻辑中被调用，返

回对应 URL 的输入流。getInputStream() 方法在成员变量 is 为空的时候，会调用 connect() 方法创建 InputStream 对象；否则，返回已经打开的输入流。connect() 使用上一节分析的 FileSystem.get() 获得 URL 对应的具体文件系统，并调用该具体文件系统的 open() 方法，获得输入流对象，代码如下：

```
class FsUrlConnection extends URLConnection {

   private Configuration conf;
   private InputStream is;

   FsUrlConnection(Configuration conf, URL url) {
      super(url);
      this.conf = conf;
   }

   @Override
   public void connect() throws IOException { //连接到URL指定的资源
      try {
         //打开文件系统
         FileSystem fs = FileSystem.get(url.toURI(), conf);
         //通过文件系统获得输入流
         is = fs.open(new Path(url.getPath()));
      } catch (URISyntaxException e) {
         throw new IOException(e.toString());
      }
   }

   //返回InputStream对象
   public InputStream getInputStream() throws IOException {
      if (is == null) {
         connect();
      }
      return is;
   }
}
```

构造一个 URL 对象，并通过 URL.openStream() 获得对应的输入流是一个很简单的过程，但 Java 的开发者把这个过程进行分解，将不同的责任分配到不同的类去实现，并通过这些类提供一个可扩展的处理框架，这背后的思想值得我们仔细体会。

## 5.6 Hadoop 具体文件系统

构建在 Hadoop 上的应用，特别是 MapReduce 应用，可能处理位于不同环境中的数据，这些数据典型是存放在分布式文件系统 HDFS 上，但有些时候也可能保存在网络储存服务上，如 Amazon 的 S3，并通过简单 S3 或本地 S3 具体文件系统访问。当然，用户也可以在 Hadoop 抽象文件系统的基础上，实现针对特定网络储存服务的具体文件系统。

### 5.6.1　FileSystem 层次结构

Hadoop 的发行包已经提供大量具体文件系统的实现，满足用户各种数据访问需求。Hadoop 项目中，FileSystem 子类以及它们之间的继承关系如图 5-25 所示。

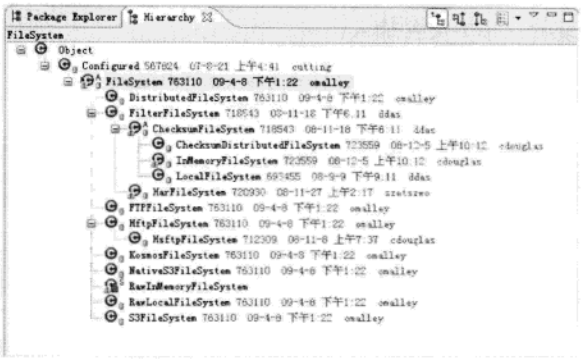



图 5-25　Hadoop 文件系统的具体实现

并不是所有的子类都继承自 org.apache.hadoop.fs.FileSystem，但 Hadoop 具体文件系统还是形成一个以 FileSystem 为根的层次结构。其中，需要关注的新层次是 FilterFileSystem 和 ChecksumFileSystem。

和前面研究过的过滤器流（FilterInputStream 和 FilterOutputStream）类似，FilterFileSystem 用于在某已有文件系统的基础上添加新的功能。FilterFileSystem 部分源代码如下，其中，成员变量 fs 保存了被装饰的原始文件系统，它一般在构造的时候就被赋值：

```
public class FilterFileSyst emextends FileSystem {
protected FileSystem fs;  //被装饰的文件系统
……
public FilterFileSystem(FileSystem fs) {
   this.fs = fs;
   this.statistics = fs.statistics;
}
……
```

FilterFileSystem 有两个子类：

- ChecksumFileSystem：用于在原始文件系统上添加校验和功能。数据在不可靠的设备存放或传输时可能会损坏，如果读数据时产生的校验和与原有校验和不一致，数据会被认为是损坏的。为了发现损坏数据，ChecksumFileSystem 会为文件保存对应的校验信息文件（在写文件的时候生成），并在读文件的时候进行校验。ChecksumFileSystem 有 3 个子类，分别在 Hadoop 分布式文件系统、本地文件系统和内存文件系统。
- HarFileSystem：保存大量的（小）文件会消耗 HDFS 名字节点的大量内存，并对 HDFS 的性能造成影响。Hadoop Archives 或 HAR 文件系统，通过将（小）文件归档（类似于 UNIX 系统中的归档命令 tar），形成一些大文件，更高效地将大量（小）

文件存放在原始文件系统中（注意，HAR 文件也可以保存在其他基础文件系统中）。Hadoop Archives 将文件的目录信息和文件数据的起始位置、长度保存在索引文件中。索引文件和归档存放的数据一起，保存在原始文件系统上。HarFileSystem 实现了 Hadoop 抽象文件系统，使得构建于 Hadoop MapReduce 的应用可以访问 Hadoop 归档文件，处理大量的（小）文件输入。

目前，Hadoop 实现的具体文件系统主要如表 5-9 所示。

**表 5-9 Hadoop 实现的具体文件系统**

| 文件系统 | URI 模式 | 实 现 | 描 述 |
|---|---|---|---|
| 本地 | file | fs.LocalFileSystem<br>fs.RawLocalFileSystem | 其中 RawLocalFileSystem 是不带校验的本地文件系统；LocalFileSystem 是 ChecksumFileSystem 的子类，是支持数据检验的本地文件系统 |
| HDFS | hdfs | hdfs.DistributedFileSystem | Hadoop 分布式文件系统 |
| HFTP | hftp | hdfs.HftpFileSystem | 在 HTTP 上提供的对 HDFS 做只读访问的文件系统。注意，虽然它的名称为 hftp，但它和 FTP（文件传输协议）没有任何关系。HFTP 通常和 distcp 命令结合，在不同版本的 HDFS 集群间复制数据 |
| HSFTP | hsftp | hdfs.HsftpFileSystem | 在 HTTPS 上提供的对 HDFS 做只读访问的文件系统。实现上，HSFTP 是 HFTP 的子类，HSFTP 使用 HTTPS 链接，提供更多的安全配置。同样，HSFTP 也和 FTP 没有关系 |
| HAR | har | fs.HarFileSystem | 存档文件系统 |
| KFS | kfs | fs.kfs.KosmosFileSystem | 类似于 HDFS 或 Google File System，由 kosmix 公司开发，用 C++ 编写的分布式文件系统，源代码可以在 http://kosmosfs.sourceforge.net/ 下载 |
| FTP | hdfs | fs.ftp.FTPFileSystem | 由 FTP 协议支持的文件系统 |
| 简单 S3 | s3n | fs.s3native.NativeS3FileSystem | Amazon S3 是一个网络储存服务，NativeS3FileSystem 可用于读写存放在 S3 上的数据⊖。Hadoop 能运行在 Amazon 提供的虚拟化集群上（Amazon EC2：Amazon Elastic Comput Cloud），在 EC2 的环境下，Hadoop MapReduce 处理的数据一般都保存在 Amazon S3 上 |
| 本地 S3 | s3 | fs.s3.S3FileSystem | 由 Amazon S3 支持的文件系统，采用和 Hadoop 类似的块格式保存文件，突破 Amazon S3 存储的 5GB 文件大小限制，可在 Amazon S3 上保持大于 5GB 的大数据。本地 S3 的主要用法和简单 S3 一样，为运行在 EC2 的 Hadoop MapReduce 应用提供数据。本地 S3 的实现机制和 HAR 类似 |
| 内存 | ramfs | fs.RawInMemoryFileSystem<br>fs.InMemoryFileSystem | 在内存中建立文件系统，提供一个比本地文件系统速度更快的 Hadoop 具体文件系统。内存文件系统在对 Hadoop 的 API 进行单元测试时，可以充当测试桩 |

⊖ 关于 Amazon S3 请参考 http://wiki.apache.org/hadoop/AmazonS3。

上述功能各异的 Hadoop 具体文件系统，一方面丰富了 Hadoop 的应用环境；另一方面充分证明了 Hadoop 抽象文件系统设计的成功。

## 5.6.2　RawLocalFileSystem 的实现

本地文件系统 RawLocalFileSystem 用于访问本地文件系统。由于 Java 已经提供了大量访问本地文件系统的方法，包括前面介绍的 java.io.File 类和输入 / 输出流，RawLocalFileSystem 使用适配器模式实现 Hadoop 抽象文件系统 FileSystem。

如前面分析，Java 文件系统界面包括两部分：用于处理文件和目录相关事务的 Java.io.File 和用于读写文件数据的输入 / 输出流。与此对应，java.io.File 实现的功能都定义在 Hadoop 抽象文件系统 FileSystem 中，读写文件，在 Hadoop 中由输入流 FSDataInputStream 和输出流 FSDataOutputStream 完成。

**1. 文件元数据和目录相关的操作**

在 RawLocalFileSystem 中，与文件元数据和目录相关的操作，都是通过适配到 java.io.File 的对应 API 完成的，适配过程相对简单，下面以 mkdirs() 和 listStatus() 为例介绍 RawLocalFileSystem 的实现。

（1）mkdirs() 方法

mkdirs() 方法用于在本地文件系统上创建目录，对应的 java.io.File 方法是 mkdir()，RawLocalFileSystem. mkdirs() 带一个参数，即被创建的目录路径，而 File.mkdir() 是无参数方法，创建目录的路径，做为 File 对象的成员变量，在构造对象时就已经传入。

RawLocalFileSystem.mkdirs() 需要的唯一参数 f 的类型是 Path，Path 在 java.net.URI 的基础上抽象了 Hadoop 文件系统中的文件或目录。org.apache.hadoop.fs.Path 类支持本地文件系统路径，如“file:///dir1/dir2/file1”，也支持 Hadoop 分布式文件系统路径，如“hdfs://example:port/dir1/dir2”，同时，还支持内存文件系统路径，如“ramfs://demo/file1”。应该说，Path 是为了支持多文件系统而引入的文件 / 目录机制，在 Java 那样的只支持原生文件系统的 API 中，不需要这样复杂的抽象。

本地文件系统的 mkdirs()，或者说大部分 Hadoop 具体文件系统的 mkdirs() 方法，支持递归地创建目录。也就是说，对于被创建目录“/dir1/dir2/dir3”，如果“/dir1/dir2”不存在，mkdirs() 会创建“/dir1/dir2”；同理，如果还没有目录“/dir1”，也会自动地创建该目录。mkdirs() 返回 true 时，操作成功，否则返回 false。相关代码如下：

```
public class RawLocalFileSystem extends FileSystem {
   ......
   public File pathToFile(Path path) {
      checkPath(path);
      if (!path.isAbsolute()) {
         path = new Path(getWorkingDirectory(), path);
      }
      return new File(path.toUri().getPath());
```

```
    }
    ......
    public boolean mkdirs(Path f) throws IOException {
        Path parent = f.getParent();
        File p2f = pathToFile(f);
        // 如果父目录非空，试图先创建父目录
        // 通过 File 创建目录，并判断成功创建目录
        return (parent == null || mkdirs(parent)) &&
                (p2f.mkdir() || p2f.isDirectory());
    }
    ......
}
```

RawLocalFileSystem.mkdirs() 的实现很简单，首先获取父目录，为可能的递归创建父目录做准备，同时调用 pathToFile() 方法，将 Path 对象转换成 File 对象。

RawLocalFileSystem.pathToFile() 首先调用 FileSystem.checkPath() 检查路径的合法性。该方法保证了用户不会在 RawLocalFileSystem 中创建“hdfs://example:port/dir1/dir2”这样的目录。接下来，pathToFile() 判断传入的路径是相对路径还是绝对路径，如果是相对路径，则转换为绝对路径。最后，利用 Path.toUri()，先将 Path 转为 URI 对象，然后通过 URI 的 getPath()，获得目录的绝对路径，并构造对应的 File 对象。

mkdirs() 方法递归创建目录部分的代码在 return 中，这个布尔表达式相对复杂，它充分使用了布尔表达式短路求值的功能。

---

**提示** 布尔表达式短路求值是指在计算布尔表达式过程中，如果表达式的结果已经可以确定是真或假，求值过程便告终止。

---

当执行 mkdirs() 方法的 return 语句表达式时，会先执行运算符“&&”的前半部分，这又是一个布尔表达式，所以，当 parent 为空时，表达式“||”的值已经为 true，表达式“||”后半部分的 mkdirs()，由于表达式短路求值，就不会被调用。所以，只有当 parent 不为空，才会创建父目录。

如果父目录创建失败，则第一个“||”子表达式会失败，并导致整个“&&”表达式失败，“&&”表达式的后半部分也就不会执行了。

表达式“&&”的后一部分也是一个“||”子表达式，它会先执行 p2f.mkdir() 创建目录，如果创建成功，则“||”子表达式的求值过程结束，整个表达式成功返回。否则，即 p2f.mkdir() 失败，要判断失败原因，如果 p2f 是目录，即 p2f.isDirectory() 返回 true，这时，待创建的目录存在，mkdir() 可以成功返回。这意味着，通过 mkdir() 重复创建已经存在的目录，mkdir() 返回 true，mkdir() 是一个幂等操作。如果 p2f.isDirectory() 返回 false，表明传入的 Path 对象对应的目标不是一个目录（它应该是一个文件），显然，由于存在同名文件，用户不能创建 Path 对象指定的目录。

RawLocalFileSystem.mkdirs() 的这个表达式充分利用了布尔表达式的短路求值，实现了各种情况下创建目录需要执行的不同逻辑，简洁而且巧妙。

（2）listStatus() 方法

RawLocalFileSystem 中要介绍的第二个方法是 listStatus()。

在目录上调用 listStatus()，可以获得一个目录下所有项目（包括文件和子目录）的元信息；在文件上调用 listStatus()，可以获得文件的元信息。listStatus() 方法的返回值是 FileStatus，它包含了文件 / 目录的大量相关信息。

RawLocalFileSystem 定义了自己的 RawLocalFileStatus，如图 5-26 所示。

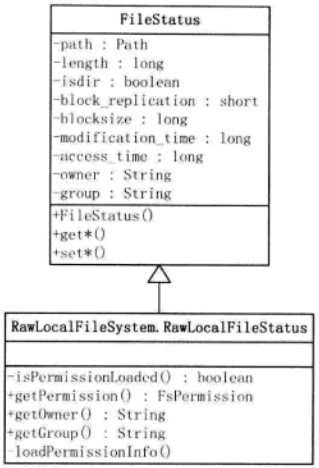



图 5-26　RawLocalFileStatus 类图

对比 FileStatus，RawLocalFileStatus 特有的处理逻辑主要在构造函数和文件权限处理部分。

构造函数通过传入的 java.io.File 对象获取文件长度、是否是目录、最后修改时间等属性。Hadoop 文件系统特有的副本数（恒为 1）、文件块大小（可以通过 fs.local.block.size 进行配置）等，则赋予特定值。代码如下：

```
static class RawLocalFileStatus extends FileStatus {
   private boolean isPermissionLoaded() {
      return !super.getOwner().equals("");
   }

   RawLocalFileStatus(File f, long defaultBlockSize, FileSystem fs) {
      super(f.length(), f.isDirectory(), 1, defaultBlockSize,
            f.lastModified(), new Path(f.getPath()).makeQualified(fs));
   }
   ......
   // 通过 "ls -ld" 命令获得文件的文件主、文件所在组和权限位
   private void loadPermissionInfo() {
      IOException e = null;
      try {
         StringTokenizer t = new StringTokenizer(
            execCommand(new File(getPath().toUri()),
                        Shell.getGET_PERMISSION_COMMAND()));
      ......
   }
   ......
}
```

获得本地文件的权限是一项非常耗资源的工作，文件权限处理部分涉及的方法包括：isPermissionLoaded()、getPermission()、getOwner()、getGroup() 和 loadPermissionInfo()。

loadPermissionInfo() 通过执行 Linux 的 Shell 命令 “ls –ld”，并解析该命令执行的结果以获得文件的权限。execCommand() 方法会创建一个新的 Java 虚拟机进程，然后在该虚拟机进程中执行上述 Shell 命令，执行 Shell 命令本身并不复杂，但在 Linux 系统中，创建一个新的 Java 虚拟机进程会消耗大量的资源。为了避免重复加载文件的权限，RawLocalFileStatus.

isPermissionLoaded() 用于判断文件的权限信息是否已经加载，采用的方法非常简单，加载了权限的 RawLocalFileStatus 对象，其 owner 非空。

RawLocalFileSystem.listStatus() 的实现不是很复杂，它也是通过 pathToFile() 获得 File 对象，如果 File 是文件，直接创建一个包含该文件元信息的 RawLocalFileStatus 元素的数组，并返回该数组；如果 File 是目录，则通过 File.list() 获得目录的所有子元素，遍历这些子元素并获取相应的 RawLocalFileStatus 对象，得到返回结果。代码如下：

```
public FileStatus[] listStatus(Path f) throws IOException {
   File localf = pathToFile(f);
   FileStatus[] results;
   ……
   if (localf.isFile()) { //f 是文件
      return new FileStatus[] {
            new RawLocalFileStatus(localf,
            getDefaultBlockSize(), this) };
   }

   String[] names = localf.list();//f 是目录
   ……// 处理 names 为空的情况
   results = new FileStatus[names.length];
   for (int i = 0; i < names.length; i++) {
      results[i] = getFileStatus(new Path(f, names[i]));
   }
   return results;
}
```

### 2. 读写文件

LocalFSFileInputStream 和 LocalFSFileOutputStream 用于读写 RawLocalFileSystem 上的文件。本节只介绍 LocalFSFileInputStream 的实现，它继承自 FSInputStream，并通过 FileInputStream 完成具体的文件读操作。

FileInputStream 从 InputStream 中派生，从名称开头的 File 就可以知道，FileInputStream 用于处理简单的文件输入操作。建立一个 FileInputStream 对象时，需要一个类型是 java.io.File 的参数，用于打开指定文件。FileInputStream 打开以后，就可以使用 InputStream 中定义的 read() 方法读取数据。LocalFSFileInputStream，也是通过适配到 FileInputStream，实现 FSInputStream 中定义的功能。它的类图如图 5-27 所示。

在阅读 LocalFSFileInputStream 的代码前，先分析 TrackingFileInputStream，它继承自 FileInputStream 并改写了所有的 read() 方法，以提供文件读取字节数的统计功能。TrackingFileInputStream 是一个典型的修饰器模式应用。

LocalFSFileInputStream 的代码分析主要关注 Seekable 和 PositionedReadable 接口相关操作的实现。Seekable 接口提供在文件流中进行随机存取的方法，而 PositionedReadable 则提供了从流中某一个位置开始读数据的一系列方法。由于 FSInputStream 中已经提供了基于 InputStream.read() 和 Seekable.seek() 的 PositionedReadable 实现，LocalFSFileInputStream 其

实只需要实现 Seekable 接口中的方法。Seekable 接口需要记住当前读取的数据在文件中的位置，在 LocalFSFileInputStream 中，该位置保持在成员变量 position 中。

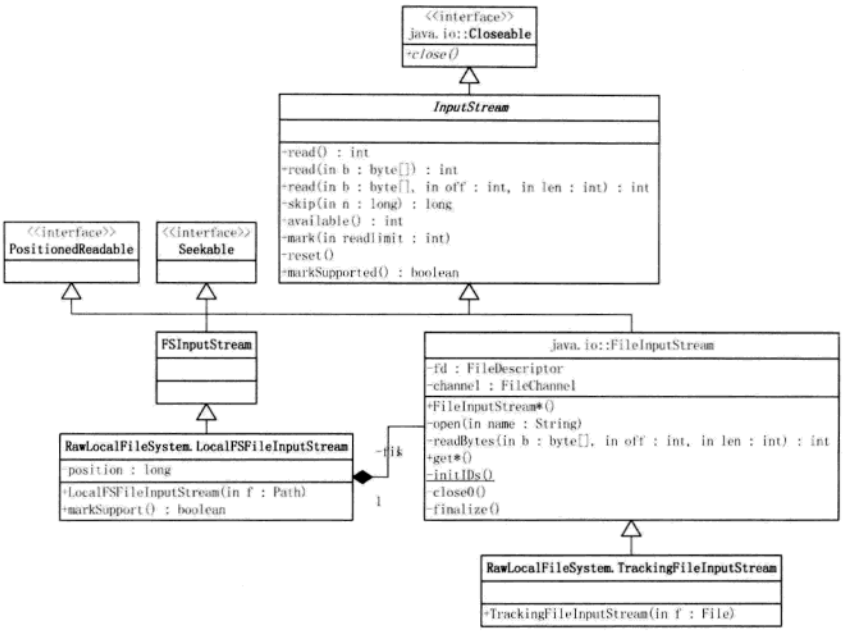



图 5-27　LocalFSFileInputStream 类图

Seekable.seek() 方法的实现使用了输入流关联的文件通道提供的能力。文件通道 FileChannel 是 Java NIO 中引入的，用于读取、写入、映射和操作文件的通道。FileChannel 保存了文件的当前位置，并提供了查询和修改当前位置的方法。LocalFSFileInputStream.seek() 的实现很简单：获得 FileInputStream 对应的 FileChannel，并调用文件通道上修改文件当前位置的方法 position()，同时更新成员变量 position。调用 read() 时需要随时更新 position，以保证 getPos() 能返回正确的值。至于 seekToNewSource()，由于本地文件系统没有这样的功能，简单返回失败即可。相关代码如下：

```
class LocalFSFileInputStream extends FSInputStream {
   FileInputStream fis;
   private long position;
   ……
   public void seek(long pos) throws IOException { //移动文件当前位置
      fis.getChannel().position(pos);
      this.position = pos;
   }

   public long getPos() throws IOException {
      return this.position;
   }
```

```
    public boolean seekToNewSource(long targetPos) throws IOException {
        // 不支持，简单返回 false
        return false;
    }

    public int available() throws IOException { return fis.available(); }
    public void close() throws IOException { fis.close(); }
    public boolean markSupport() { return false; }

    public int read() throws IOException {
        try {
            int value = fis.read();
            if (value >= 0) {
                this.position++; // 更新文件当前位置
            }
            return value;
        }
        ......
    }
    ......
}
```

LocalFSFileInputStream 中的其他方法实现，如 available() 和 close()，只需要调用 FileInputStream 的对应方法，不需要任何新的修饰。由于 FileInputStream 已经提供了读取本地文件数据的大部分功能，LocalFSFileInputStream 的实现非常简单易懂。

### 5.6.3 ChecksumFileSystem 的实现

作为用户，我们希望存放在文件系统中的数据在存储的时候不会出现损坏，也希望数据在进行网络传输的时候不会引入错误。Hadoop 针对海量数据进行设计，管理的数据量非常大，即使单个数据发生错误的概率很小，但保存的信息出现损坏的可能性还是很高的。为了在已有文件系统的基础上，保证文件的数据完整性，Hadoop 提供了抽象的 ChecksumFileSystem 文件系统，它的具体子类有本地文件系统 LocalFileSystem、内存文件系统 InMemoryFileSystem 等，而 Hadoop 分布式文件系统的实现，也使用了 ChecksumFileSystem 中的一些能力。

#### 1. 数据完整性和 CRC

对数据差错进行校验的研究已经有很长的历史了，一般来说，减少数据差错的手段有两种：一方面是提高设备的性能和质量，如磁盘的质量或网络通信的质量，以减少数据发生错误的概率；另一方面，通过采用差错控制，利用某个手段去发现并纠正错误。

发现差错甚至能纠正差错的常用方法是对数据进行适当的编码，即给原始数据加上一些冗余数据，并使冗余数据和原始数据之间具备某种关系，然后将这两种数据通过一个不可靠的信道传输（磁盘可以看成是一种特殊的信道），数据可能在传输过程中被损坏，信道的另一方接收到数据后，通过检验原始数据和冗余数据间的关系是否符合原来建立的那个关系，

进行数据完整性检查。上述冗余数据和原始数据间的关系，可分为两种，能校验差错的编码称为检错码，可以纠正错误的编码称为纠错码，本书只涉及检错码。

需要注意的一点是，原始数据和冗余数据都有可能出错，有可能出现冗余数据出现错误但原始数据是正确的情况。一般来说，由于冗余数据远小于原始数据，这种情况发生的可能性不大。

常用的错误检测方法有奇偶校验、校验和，以及循环冗余校验等。在 Hadoop 中，一般使用 CRC-32（Cyclic Redundancy Check，循环冗余校验）进行错误检测，它可用于计算任意长度的 32 位整数序列的校验和。循环冗余校验的基本思想是：将任何一个二进制位串看成是一个多项式，并通过多项式运算，先对多项式进行乘法，然后再根据校验值的位数，除以某特定多项式，求出原始数据对应的冗余数据，有兴趣的读者可以参考计算机网络或通信原理的相关图书。应该说，循环冗余校验的检错能力，在上述 3 种校验方法中是最好的。

Hadoop 中，CRC-32 的实现在 org.apache.hadoop.util.PureJavaCrc32，CRC-32 的细节不是本书讨论的重点。站在使用者的角度看，我们关心的是如果有一批数据，如何获得这批数据的 CRC-32 检验结果。

### 2. Checksum 接口

Checksum 接口定义在 java.util.zip 包中，用于表示数据校验和，类图如图 5-28 所示。

Checksum 有 4 个方法：

- update() 用于为校验和算法提供数据，它有两个重载形式。为 Checksum 提供一定的数据后，可以通过 getValue() 获得这批数据的校验和。
- getValue() 方法的返回值类型是 long，它是一个 64 位的整数，调用者需要根据校验和算法的定义，从返回值中获取结果。以 CRC-32 为例，CRC-32 的校验和输出是 32 位的，所以将 getValue() 的返回值强制转换成 int 类型，即可得到对应的校验和。
- reset() 方法复位 Checksum 对象，为新的输入做准备。

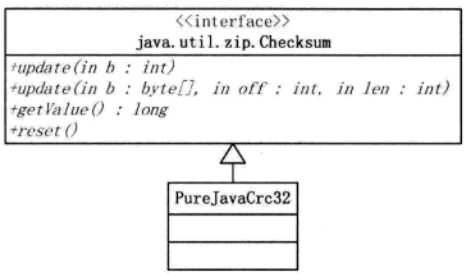



图 5-28　Checksum 类图

Java 中，Checksum 是为了处理标准的 zip 和 gzip 文件而引入到 java.util.zip 包中，其具体实现类是 java.util.zip.CRC32，它使用 Java 本地方法实现了 update() 方法。实际上，该实现的的效率不高，Hadoop 的开发者使用纯 Java 重新实现了 java.util.zip.CRC32，就是前面提到的 PureJavaCrc32，它的性能比 Java 类库中提供的版本高 1.8 到 10 倍。

使用 Checksum 类的一个例子是 org.hadoopinternal.fs.LocalFileSystemDemo 的 crcDemo() 方法，通过 PureJavaCrc32.update() 写入 512 字节的数据，然后通过 getValue() 获得这些输入的 CRC-32 校验结果，其十六进制表示是 FC152A68。代码如下：

```
public static int crc_block_len=512;
```

```
public static void crcDemo() {
    Checksum crc32=new PureJavaCrc32();

    String data="testingtesting";
    byte[] buf=data.getBytes();

    int buflen=buf.length;
    for(int ii=0; ii<crc_block_len/buflen; ii++) {
        crc32.update(buf, 0, buflen); // 调用 update() 输入数据
    }
    crc32.update(buf, 0, crc_block_len%buflen);

    // 获得结果并输出
    System.out.println("CRC-32 checksum is: "+crc32.getValue()
            +" Hex is:"+Long.toHexString(crc32.getValue()).toUpperCase());
}
```

### 3. ChecksumFileSystem 的实现

ChecksumFileSystem，顾名思义，就是提供了校验和（Checksum）能力，具有数据完整性保证的文件系统。数据校验能力其实是在基本具体文件系统上提供的新功能，其工作离不开原来的具体文件系统，所以，ChecksumFileSystem 继承自 FilterFileSystem。

在 ChecksumFileSystem 中，如果有文件“example.txt”，那么该文件的 CRC-32 校验信息保存在文件“.example.txt.crc”中，即在文件名前加“.”。大多数操作系统下，如 Windows 和 Linux 中，在文件名前加点为隐藏文件。Linux 操作系统下，用“ls”命令无法列出隐藏文件，必须使用参数“-a”才能查看。

ChecksumFileSystem.getChecksumFile() 用于获得一个文件对应的 CRC-32 校验信息文件，转换方法有如上述规则。isChecksumFile() 方法也运用该规则，判断某指定文件是否是校验信息文件。代码如下：

```
public abstract class ChecksumFileSystem extends FilterFileSystem {
    ......
    public Path getChecksumFile(Path file) {
        return new Path(file.getParent(), "." + file.getName() + ".crc");
    }

    public static boolean isChecksumFile(Path file) {
        String name = file.getName();
        return name.startsWith(".") && name.endsWith(".crc");
    }
}
```

**注意** ChecksumFileSystem 引入了一个规则，即校验信息文件的文件名是对应文件的文件名前面加“.”，后面加扩展名“.crc”构成，用户不应该在 ChecksumFileSystem 中破坏这个规则。

（1）文件和目录相关事务

ChecksumFileSystem 中，文件和目录相关事务的相关方法实现，大部分都在处理 CRC-32 校验信息文件，或者说，是在维护文件和 CRC-32 校验信息文件的上述关系。

以文件改名方法 rename() 为例，如果修改了一个数据文件的文件名，对应的 CRC-32 校验信息文件文件名也必须修改。ChecksumFileSystem.rename() 首先判断需要修改名字的是否是目录，如果是目录，由于不存在校验文件，直接调用底层文件系统的改名方法改名。如果是个文件，则先修改数据文件的文件名，操作成功后才进行可能的校验信息文件文件名修改。这个过程分为几步，首先是判断校验信息文件是否存在，如果不存在（在 ChecksumFileSystem 中，这是允许的），则不需要任何操作。如果校验信息文件存在，则要判断修改名字操作的目标 dst 是不是目录，如果是目录，rename() 其实是把文件移动到目标目录，则需要将校验信息文件也移动到目录 dst 中；如果目标是文件，则将原校验信息文件改名为新目标文件对应的，通过 ChecksumFileSystem.getChecksumFile() 获得校验信息文件名。代码如下：

```
public boolean rename(Path src, Path dst) throws IOException {
   if (fs.isDirectory(src)) {
      return fs.rename(src, dst);
   } else {
      boolean value = fs.rename(src, dst);// 修改数据文件文件名
      if (!value)
         return false;

      Path checkFile = getChecksumFile(src);
      if (fs.exists(checkFile)) { // 修改校验信息文件文件名
         if (fs.isDirectory(dst)) {
            value = fs.rename(checkFile, dst);
         } else {
            value = fs.rename(checkFile, getChecksumFile(dst));
         }
      }
   }
}
```

（2）写文件

与 rename()、delete() 等方法相比，在 ChecksumFileSystem 上读写文件数据是一件相对复杂的事情。其难度在于必须维护字节流上的数据读写和基于块的校验和的关系。一般来说，对于数据的每 ${io.bytes.per.checksum} 字节，都会创建一个单独的校验和。该配置项的默认值是 512，而 CRC-32 校验和的输出是 4 字节（4Bytes，32bits）。也就是说，ChecksumFileSystem 上的数据文件会逻辑上以 512 字节进行分块，然后每个块计算出一个 4 字节的校验和，存放在校验信息文件中，数据块校验带来的存储开销小于 1%。

ChecksumFileSystem 中，ChecksumFSOutputSummer 在基本具体文件系统输出流的基础上，实现数据文件和校验信息文件的数据输出。ChecksumFSOutputSummer 继承自 org.apache.hadoop.fs.FSOutputSummer，它是一个抽象类，提供在流上计算校验和并输出的能力，

图 5-29 是它们的关系图。

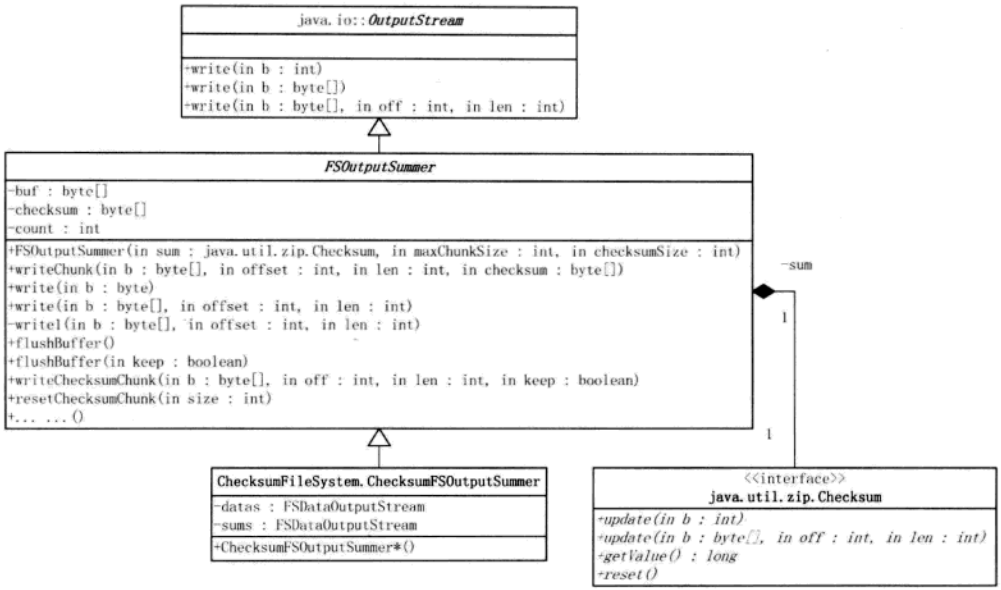



图 5-29 ChecksumFSOutputSummer 类图

FSOutputSummer 有 4 个成员变量，包括：用于计算校验和的 sum 变量（类型为 Checksum）、用于暂时保存输出数据的缓冲区 buf、校验和结果缓冲区 checksum，以及 buf 已使用空间计数 count。相关代码如下：

```
abstract public class FSOutputSummer extends OutputStream {
   private Checksum sum;// 用于计算校验和的 Checksum
   private byte buf[];// 输出数据（未计算校验和）
   private byte checksum[];// 校验和缓冲区
   private int count;
   ......
}
```

FSOutputSummer 中 write() 方法有两种形式，下面以缓冲区写形式的 write() 为例分析写数据的实现。它的主要过程是一个循环，循环地调用内部方法 write1() 写数据，直到所有的数据都写入流中。FSOutputSummer.write1() 每次都尽可能地写出数据，即尽可能将缓冲区 buf 写满。

方法 write1() 的前半部分代码实现了一个“完美”情况下的写处理流程，这时它将读入缓冲区 buf 长度的数据，首先它计算这部分数据的校验和，并通过纯虚方法 writeChunk() 写出数据。

纯虚方法 writeChunk() 由 FSOutputSummer 的子类实现，它的实现一般需要写出缓冲区中的文件数据和这部分数据的校验和两部分内容。

为什么 write1() 要做这样的特殊处理呢？在这个“完美”情况下，它避免了将数据拷贝到缓存区中，减少内存操作，提高了系统的效率。FSOutputSummer.write1() 的另一部分代码不复杂，拷贝尽可能多的数据到缓冲区，如果缓冲区满，则通过 flushBuffer() 刷出数据。代码如下：

```
protected abstract void writeChunk(byte[] b, int offset, int len,
      byte[] checksum)throws IOException;
......
public synchronized void write(byte b[], int off, int len)
      throws IOException {
   ......
   for (int n=0;n<len;n+=write1(b, off+n, len-n)) {
   }
}

// 尽可能地写出数据，并（根据情况）输出到底层的输出流
private int write1(byte b[], int off, int len) throws IOException {
   if(count==0 && len>=buf.length) {
      // 本地缓冲区为空而且输出数据大于一个校验块，可以计算出校验和
      final int length = buf.length;
      sum.update(b, off, length); // 计算检验和
      writeChecksumChunk(b, off, length, false); // 输出数据
      return length;
   }

   // 将数据读入对象的内部缓存区
   int bytesToCopy = buf.length-count; //buf 还能保持多少数据
   bytesToCopy = (len<bytesToCopy) ? len : bytesToCopy; // 读入数据
   sum.update(b, off, bytesToCopy); // 读入数据计算校验和
   System.arraycopy(b, off, buf, count, bytesToCopy);
   count += bytesToCopy;
   if (count == buf.length) {
      flushBuffer();// 缓冲区满，已经读入一个校验块，可以输出
   }
   return bytesToCopy;
}
......
protected synchronized void flushBuffer(boolean keep) throws IOException {
   if (count != 0) {
      int chunkLen = count;
      count = 0;
      writeChecksumChunk(buf, 0, chunkLen, keep);
      ......
   }
}

private void writeChecksumChunk(byte b[], int off, int len, boolean keep)
      throws IOException {
   int tempChecksum = (int)sum.getValue();
   if (!keep) {
```

```
            sum.reset(); // 复位 Checksum 对象
        }
        int2byte(tempChecksum, checksum);
        writeChunk(b, off, len, checksum); // 输出数据
    }
```

FSOutputSummer.flushBuffer() 用于写出缓冲区，它最终通过 writeChecksumChunk() 方法调用纯虚方法 writeChunk() 输出数据。在 writeChecksumChunk() 中，FSOutputSummer 还需要根据情况复位 Checksum 对象，为下一块数据的输入做准备。

ChecksumFileSystem 使用的输出流是 ChecksumFSOutputSummer，它是 FSOutputSummer 子类，相当简单，毕竟其父类已经实现了很多的功能。主要代码如下：

```
private static class ChecksumFSOutputSummer extends FSOutputSummer {
    private FSDataOutputStream datas;
    private FSDataOutputStream sums;
    private static final float CHKSUM_AS_FRACTION = 0.01f;
    ......
    public ChecksumFSOutputSummer(ChecksumFileSystem fs,Path file,
            boolean overwrite,int bufferSize,short replication,
            long blockSize,Progressable progress)throws IOException{
        super(new PureJavaCrc32(), fs.getBytesPerSum(), 4);
        int bytesPerSum = fs.getBytesPerSum();
        this.datas = fs.getRawFileSystem().create(file,
                overwrite, bufferSize, replication, blockSize, progress);
        int sumBufferSize = fs.getSumBufferSize(bytesPerSum, bufferSize);
        this.sums = fs.getRawFileSystem().create(fs.getChecksumFile(file),
                true, sumBufferSize, replication, blockSize);
        sums.write(CHECKSUM_VERSION, 0, CHECKSUM_VERSION.length);
        sums.writeInt(bytesPerSum);
    }
    ......
    protected void writeChunk(byte[] b, int offset, int len, byte[] checksum)
                throws IOException {
        datas.write(b, offset, len); // 输出文件数据
        sums.write(checksum); // 输出校验
    }
}
```

ChecksumFSOutputSummer 类的实现有两个关键点，一个是构造函数，另一个就是 FSOutputSummer 的纯虚方法 writeChunk()。

构造函数需要初始化几个成员变量：用 PureJavaCrc32 对象初始化父类的 Checksum 成员变量、设定数据块大小（默认值是 512 字节）和校验和结果的大小（4 字节，CRC-32 的结果）。然后，创建两个输出流，分别用于输出文件数据和输出文件数据的校验和，最后，在输出校验和的流上写入校验文件的文件头，包括校验方法和进行校验的数据块大小，按照默认配置，分别是“crc\0”（\0 表示是数值 0，不是字符 0）和 512 字节。对比构造函数，writeChunk() 函数很简单，它将数据分别写入在构造函数中打开的两个输出流。

在 org.hadoopinternal.fs.LocalFileSystemDemo 实例中，打开本地文件系统 LocalFileSystem

（一个 ChecksumFileSystem 的子类），并通过该 LocalFileSystem 实例的输出流写了一批数据（往文件中写入字符串“testingtesting”，一共写 256 次），可得到如下校验文件的内容（在 LocalFileSystemDemo 的 PrintCRCFile() 函数中输出）：

```
63 72 63 00 00 00 02 00   FC 15 2A 68 AE B1 64 2D    crc..... ..*h..d-
85 9B B4 9E 18 DF 9B 8F   99 8D 95 6C 47 B9 2A 7D    ........ ...lG.*}
DC 72 80 73                                          .r.s
```

如同前面的分析，该文件的前面 8 字节包含的信息有：检验文件文件头“crc\0”，数据文件每 512（0x00000200）字节产生一个校验和。接下来的文件内容是第一个 512 字节数据产生的检验和“FC152A68”，在“数据完整性和 CRC”讨论过程中已经了解过这个结果。

总的来说，FSOutputSummer 抽象了大部分和数据分块、计算检验和相关功能，只需要实现抽象方法 writeChunk()，就可以得到一个具体子类。如 ChecksumFSOutputSummer 就简单地实现了父类的 writeChunk() 方法，将数据和数据的检验和写入具体文件系统，非常简洁。

（3）读文件

ChecksumFSOutputSummer 将数据和数据的 CRC-32 校验信息分别写入数据文件和校验信息文件。读文件的时候，也必须从数据文件和校验信息文件中分别读入数据，并根据校验信息对读入的数据进行检查，以判断数据是否正确。

和写文件对偶，读文件的流程涉及 ChecksumFSInputChecker 和它的父类 FSInputChecker，它们分别对应于 ChecksumFSOutputSummer 和 FSOutputSummer。从类图上看，它们的结构也很相似，如图 5-30 所示。

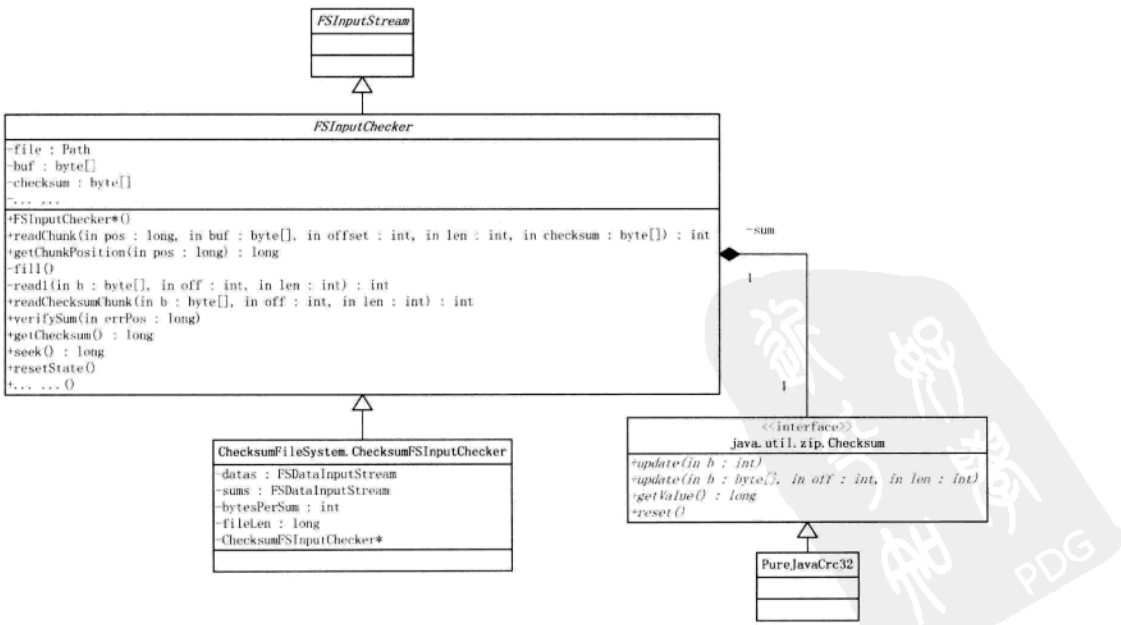



图 5-30　ChecksumFSInputChecker 类图

FSInputChecker 也来自于 org.apache.hadoop.fs 包，为数据输入提供了检验能力。它的成员变量远多于 FSOutputSummer，在 FSOutputSummer 中已经出现的、功能类似的有：类型为 Checksum 的成员变量 sum、用于暂时保存输入数据的缓冲区 buf、校验和缓冲区 checksum，以及 buf 已使用空间计数 count。FSInputChecker 特有的包括已读数据位置 pos、出错重试次数 numOfRetries 和当前数据块位置 chunkPos。

其中 pos 将输入数据缓冲区的数据分为两部分：下标位于 pos 位置前面（不包括 pos）的数据，是已经被应用读取的数据；下标位于 pos 和 count 间的数据，是 FSInputChecker 缓冲的、未被应用取走的数据。当前数据块位置 chunkPos 记录了下一次读取数据所在的数据块的起始位置。声明如下：

```
abstract public class FSInputChecker extends InputStream {
    ……
    private Checksum sum;
    private boolean verifyChecksum = true;
    private byte[] buf;
    private byte[] checksum;
    private int pos;
    private int count;
    private int numOfRetries;
    private long chunkPos = 0;// 已经读入的数据的位置
    ……
}
```

从 read() 开始分析 FSInputChecker，和 FSOutputSummer.write() 类似，它循环调用内部的 read1() 方法直到读满缓冲区或者读到文件尾。read1() 为了提高效率，在能够避免内存拷贝的情况下，会通过 readChecksumChunk() 直接读一整块数据到用户缓冲区。避免内存拷贝的条件是：

1）当前 FSInputChecker 没有可读取的数据，即 buf 为空，这就不需要拷贝 FSInputChecker 中缓冲的输入数据到用户缓冲区了。

2）用户提供的缓冲区能够读满一整块数据，也即不需要在对象内部暂存数据。

如果不满足上述条件 1，即 FSOutputSummer 对象有缓冲数据，则拷贝尽可能多的数据到输入缓冲区 b 中；否则，先通过 fill() 方法，读入一整块数据，再将读入的部分数据拷贝到缓冲区 b，这种情况和上面不满足条件 1 的区别在于，需要通过 fill() 准备数据，数据准备完毕后，两种情况的处理过程是一样的、共享的。

另外，为了避免在对象内部的数据拷贝，FSInputChecker 使用了一个缓冲区 buf 上的两个变量 pos 和 count，用于记录缓冲区里数据的使用情况，而不是直接使用 java.nio.Buffer，也是基于效率的考虑。代码如下：

```
public synchronized int read(byte[] b, int off, int len) throws……{
    ……
    int n = 0;
    for (;;) {
        int nread = read1(b, off + n, len - n); //循环读数据
```

```
        if (nread <= 0)
            return (n == 0) ? nread : n;
        n += nread;
        if (n >= len)
        return n;
    }
}

private int read1(byte b[], int off, int len) throws IOException {
    int avail = count-pos;
    if( avail <= 0 ) {
        if(len>=buf.length) {
            // 读一块数据到用户缓冲区，避免数据拷贝
            int nread = readChecksumChunk(b, off, len);
            return nread;
        } else {
            // 读一整块数据到 FSInputChecker 的缓冲区
            fill();
            if( count <= 0 ) {
                return -1;
            } else {
                avail = count;
            }
        }
    }

    // 拷贝数据，从内部的缓冲区 buf 复制到用户缓冲区 b 中
    int cnt = (avail < len) ? avail : len;
    System.arraycopy(buf, pos, b, off, cnt);
    pos += cnt;
    return cnt;
}
```

FSInputChecker.fill() 很简单，但 readChecksumChunk() 就复杂了，它在读数据发生校验错误的时候，如果有可读的副本，它会试图从副本中读取数据，重试次数由成员变量 numOfRetries 规定。readChecksumChunk() 方法的开始会先复位缓冲区，并对方法的一些成员变量进行初始化。在 do-while 循环中，readChecksumChunk() 先通过抽象方法 readChunk() 读入数据和数据的校验和，然后利用成员变量 sum 计算输入数据的校验和，并通过 verifySum() 方法（简单方法）进行对比，如果校验和对比失败，即读入的数据校验和与读入数据通过 Checksum 对象 sum 计算出来的校验和不相等，verifySum() 方法便会抛出异常。

再次强调，verifySum() 方法不能区分是检验数据有错误还是原始数据有错误，只要是上述两个校验和不相等，都认为校验失败。否则，更新成员变量 chunkPos，为下次读做准备，同时设置变量 retry 为 false，退出 do-while 循环，结束读取数据的过程。

当 verifySum() 抛出异常后，会被 catch 子句捕获。该子句的处理过程是这样的，如果已经进行了多次尝试并都失败了，那么重新抛出异常；否则，通过调用流的 seekToNewSource() 方法打开另外一个副本，并将读取流的位置设置为 chunkPos，为下一次读做准备。如果

seekToNewSource() 返回 false，也就是说没有新的副本可以使用，没法继续进行尝试，这时也抛出被捕获的异常，通知应用检验出错。代码如下：

```
abstract protected int readChunk(long pos, byte[] buf, int offset, int len,
      byte[] checksum) throws IOException;
......
private void fill(  ) throws IOException {
   assert(pos>=count);
   // 调用 readChecksumChunk，读入数据到内部缓冲区
   count = readChecksumChunk(buf, 0, buf.length);
}
......
private int readChecksumChunk(byte b[], int off, int len)throws……{
   count = pos = 0;// 缓冲区复位

   int read = 0;
   boolean retry = true;
   int retriesLeft = numOfRetries;
   do {
      retriesLeft--;

      try {
         // 我们需要成员变量 chunkPos 的信息，才能进行重试
         read = readChunk(chunkPos, b, off, len, checksum);
         if( read > 0 ) {
            if( needChecksum() ) {
               sum.update(b, off, read);
               verifySum(chunkPos);  // 进行校验，出错会抛异常
            }
            chunkPos += read;
         }
         retry = false;
      } catch (ChecksumException ce) {
         ......
         if (retriesLeft == 0) {
            throw ce;
         }

         // 试图找一个新的副本
         if (seekToNewSource(chunkPos)) {
            seek(chunkPos);// 找一个新的副本，重新读取数据
         } else {
            throw ce;// 找新的副本失败，抛出异常
         }
      }
   } while (retry);
   return read;
}
```

在转入 ChecksumFSInputChecker 的分析前，我们先看一下 FSInputChecker.seek() 方法。

函数 seek() 来自 org.apache.hadoop.fs.Seekable 接口，提供了在输入流中进行定位的能力。FSInputChecker 的输入是基于块的，所以 seek() 必须将在字节流上的偏移量转换成基于块的读入操作。具体过程是：如果要定位的数据正好处于当前缓冲区，那么 seek() 会进行一个优化（方法开始的那部分代码），这时候不需要进行真正的数据读取，只需要简单地改变下次读数据的指针 pos 到相应位置即可。如果 seek() 移动读位置到一个新的校验块，则必须重置当前对象状态，调用虚方法 getChunkPosition() 定位对应的块位置，接着使用工具方法 readFully() 读取数据，并抛弃从块开始位置到目标位置的那部分数据，这相当于在块内部将下次读数据的指针移动到 seek() 的输入参数 pos 指向的位置。至此，定位完成。代码如下：

```
public synchronized void seek(long pos) throws IOException {
    ……
    // 优化：如果要定位的数据块正好是当前块
    long start = chunkPos - this.count;
    if( pos>=start && pos<chunkPos) {
        this.pos = (int)(pos-start);// 简单移动下次读数据的指针到相应位置
        return;
    }

    resetState();// 重置当前状态

    chunkPos = getChunkPosition(pos);// 对应块的位置

    // 移动 pos 到指定位置
    int delta = (int)(pos - chunkPos);
    if( delta > 0) {
        readFully(this, new byte[delta], 0, delta);
    }
}
```

ChecksumFSInputChecker 继承自 FSInputChecker，对比 ChecksumFSOutputSummer，它比较复杂，但也较 FSInputChecker 简单。ChecksumFSInputChecker 的主要成员变量有：创建对象对应的文件系统实例，保存该信息，主要是为了在 ChecksumFSInputChecker 发现读入数据有错误时，通知文件系统进行相关的处理；成员变量 datas 和 sums，是相应的原始数据文件和冗余数据文件的输入流；整型的成员变量 bytesPerSum 保存了每个数据块的大小，即每 bytesPerSum 字节的原始数据形成一个数据块，然后计算出校验和。

在构造函数中，ChecksumFSInputChecker 打开输入流 datas 和 sums 后，还需要进行一系列的检查和配置，才能完成对象的创建。这个过程，首先是在输入流 sums 上读取校验文件的文件头，FSOutputSummer 在写校验文件时，前面 4 字节是“crc\0”，这是校验文件的魔数（Magic Number），如果读入的文件魔数不正确，则抛出异常 IOException，在后面的异常处理 catch 子句中，会禁用校验功能。这是一种保护机制，即根据 ChecksumFileSystem 规则得到的检验信息文件，其实不是 FSOutputSummer 创建的，那么，ChecksumFSInputChecker 就不能影响对原有数据文件的读取，即不能根据该检验信息文件对数据文件进行校验。在正常流程中，接下来构造函数需要在流 sums 上继续读入一个整数，这就是数据块大小

bytesPerSum 的值。代码如下：

```
public ChecksumFSInputChecker(ChecksumFileSystem fs, Path file,
      int bufferSize)throws IOException {
   super( file, fs.getFileStatus(file).getReplication() );
   this.datas = fs.getRawFileSystem().open(file, bufferSize);
   this.fs = fs;
   Path sumFile = fs.getChecksumFile(file);

   try {
      int sumBufferSize
            = fs.getSumBufferSize(fs.getBytesPerSum(), bufferSize);
      sums = fs.getRawFileSystem().open(sumFile, sumBufferSize);

      byte[] version = new byte[CHECKSUM_VERSION.length];
      sums.readFully(version);
      if (!Arrays.equals(version, CHECKSUM_VERSION))//检查文件魔数
         throw new IOException("Not a checksum file: "+sumFile);

      this.bytesPerSum = sums.readInt(); //读入bytesPerSum
      set(fs.verifyChecksum, new PureJavaCrc32(), bytesPerSum, 4);
   } catch (FileNotFoundException e) {//校验文件不存在，关闭校验功能
      set(fs.verifyChecksum, null, 1, 0);
   } catch (IOException e) {
      ......
      set(fs.verifyChecksum, null, 1, 0);//校验文件魔数不正确，关闭校验功能
   }
}
```

FSInputChecker 中的纯虚函数在 ChecksumFSInputChecker 的实现如下：

```
protected long getChunkPosition( long dataPos ) {
   return dataPos/bytesPerSum*bytesPerSum;
}
......
protected int readChunk(long pos, byte[] buf, int offset, int len,
      byte[] checksum) throws IOException {
   boolean eof = false;
   if(needChecksum()) {
      try {
         long checksumPos = getChecksumFilePos(pos);
         if(checksumPos != sums.getPos()) { // 当前读取数据位置不在边界上
            sums.seek(checksumPos);
         }
         sums.readFully(checksum);
      } catch (EOFException e) {
         eof = true;
      }
      len = bytesPerSum;
   }
```

```
    if(pos != datas.getPos()) {
       datas.seek(pos);
    }
    int nread = readFully(datas, buf, offset, len);
    if( eof && nread > 0) {
       throw new ChecksumException("Checksum error: "+file+" at "+pos, pos);
    }
    return nread;
}
```

这两个方法都很简单。getChunkPosition() 返回包含文件中某一位置对应块的起始地址。readChunk() 用于读入一块原始数据及其对应的校验数据，读入的过程非常相似，如果流的当前读取位置和要读入数据的位置不一致，则通过 seek() 方法移动读位置，接着利用工具方法 readFully() 读入数据并保证读入要求的所有数据。

在本节的最后，给出一个例子，让读者观察在校验出错，ChecksumFSInputChecker 抛出异常时，ChecksumFSInputChecker 和 FSInputChecker 中各个方法的调用关系。

在 org.hadoopinternal.fs.LocalFileSystemDemo 的 LFSCRCErrorDemo() 方法中，通过 ChecksumFSOutputSummer 创建文件 LocalFileDemo.txt，接下来修改了该原始数据文件对应的 CRC-32 校验文件的内容，将校验数据的第一位从 FC 修改为 00。也就是说，原来第一个数据块的校验码是“FC152A68”，修改后变成“00152A68”，这必然触发一个校验错误，并抛出如下 ChecksumException 异常。

```
INFO fs.FSInputChecker: Found checksum error: b[0, 512]=74657374696e6774657374696
  e677465......
org........ChecksumException: Checksum error: LocalFileDemo.txt at 0
   at org........FSInputChecker.verifySum(FSInputChecker.java:277)
   at org........FSInputChecker.readChecksumChunk(FSInputChecker.java:241)
   at org.apache.hadoop.fs.FSInputChecker.fill(FSInputChecker.java:176)
   at org.apache.hadoop.fs.FSInputChecker.read1(FSInputChecker.java:193)
   at org.apache.hadoop.fs.FSInputChecker.read(FSInputChecker.java:158)
   at java.io.DataInputStream.read(Unknown Source)
   at org.hadoopinter.......LFSCRCErrorDemo(LocalFileSystemDemo.java:162)
   at org.hadoopinter.......main(LocalFileSystemDemo.java:174)
......
```

ChecksumException 异常时输出的调用栈，正好说明了这一节分析的 FSInputChecker 读数据过程。当应用通过 DataInputStream 对象试图读入数据时，由于 DataInputStream 其实是过滤器流，需要从被包装的流中去读取数据，即它调用了 FSInputChecker.read() 方法。

FSInputChecker.read() 调用内部的 read1()，由于这个时候还没有读入任何数据，readChecksumChunk() 会被方法 read1() 调用。

在通过 readChunk() 读入原始数据和校验数据后，readChecksumChunk() 使用 verifySum() 进行数据校验。由于 CRC-32 校验文件的校验结果被修改，所以通过原始数据计算得到的校验和结果“FC152A68”显然和修改后的“00152A68”不相等，verifySum() 抛出 ChecksumException 异常。

数据完整性是海量数据处理中一个重要问题，不但 Hadoop Common 中的文件系统提供了 ChecksumFileSystem，在 Hadoop 分布式文件系统中也实现了类似的机制。正因为如此重要，在这一小节中，和以往对输入 / 输出流的讨论不同，我们详细地介绍了写文件和读文件的完整过程，让读者有一个更全面的了解。

### 5.6.4 RawInMemoryFileSystem 的实现

从实现者的角度上看，RawLocalFileSystem 和 ChecksumFileSystem 并没有实现一个真正的文件系统，它们都必须依赖某一个已有的文件系统。在 org.apache.hadoop.fs 包中，Hadoop 的开发者提供了一个不依赖任何文件系统，但实现又十分简单的文件系统：内存文件系统 RawInMemoryFileSystem。内存文件系统提供的功能不如文件系统发展早期那些系统提供的能力强，但通过它，还是可以了解一些文件系统实现的内貌。

在内存中实现文件系统是一个很自然的想法：应用开发人员熟悉文件系统，同时，内存提供了远比磁盘高的访问速度。所以，当需要对 Hadoop 的 API 进行单元测试时，利用内存文件系统充当测试桩，是一个不错的选择。

内存文件系统不依赖其他文件系统，它部分实现了抽象类 org.apache.hadoop.fs.FileSystem 中要求的 API 及简单的输入 / 输出流。

RawInMemoryFileSystem 是 InMemoryFileSystem 的内部类，而 InMemoryFileSystem 是 ChecksumFileSystem 的子类，它有能力对保存在文件系统上的数据进行完整性检查。InMemoryFileSystem 已经被注解（Annotation）为 @Deprecated，也就是说，不应该继续使用内存文件系统了，但作为文件系统的简单实现，InMemoryFileSystem 还是值得分析的。

InMemoryFileSystem 十分简陋，原因是：首先它不支持目录，不能在内存文件系统中创建目录；其次，InMemoryFileSystem 的实现要求使用者预先知道要写入的文件的大小，并通知文件系统保留对应的空间，并保存校验数据；同时，在内存文件系统中，用户保存的所有文件长度的和，不能超过文件系统的预留空间。

图 5-31 是 RawInMemoryFileSystem 的类图，它实现了抽象类 FileSystem 中的所有抽象方法，同时，和 RawLocalFileSystem 类似，它有自己的继承自 FileStatus 的 InMemoryFileStatus，用于保存文件属性。

前面已经提到，InMemoryFileSystem 不支持目录，这意味着所有在内存文件系统中保存的文件，都处在同一级目录，也就是“根目录”下。所以，在 InMemoryFileSystem 中，这些文件对应的内容以及预留空间的大小，都保持在 FileAttributes 中。

FileAttributes 非常简单，预留空间的大小保存在成员变量 size 中，而文件的内容，会放在字节数组 data 中。代码如下：

```
private static class FileAttributes {
   private byte[] data; // 文件数据
   private int size;

   public FileAttributes(int size) {
```

```
        this.size = size;
        this.data = new byte[size];
    }
}
```

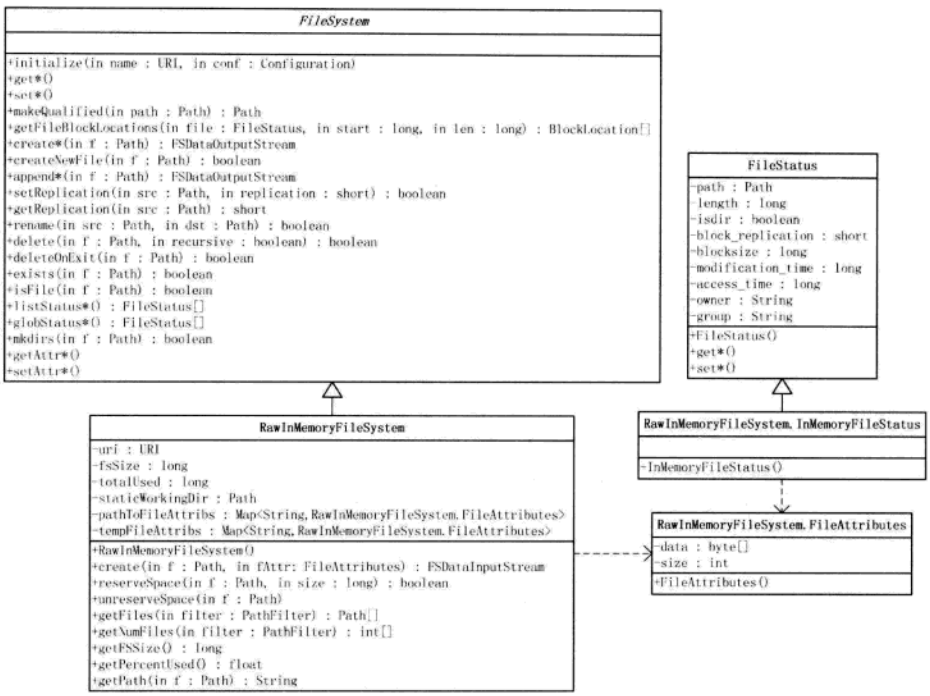



图 5-31　RawInMemoryFileSystem 类图

RawInMemoryFileSystem 的成员变量不多，包括文件系统预留空间 fsSize 和目前已使用空间 totalUsed，它们都是长整型。

前面已经提到，内存文件系统中没有目录，所有文件就都以映射表的形式保存在系统中。映射表的键是文件名，映射表对应的值是 FileAttributes 对象，它包含了预留空间和该空间大小的信息。这样的映射表有两个，一个用于保存正在产生中的文件，即已经为文件预留空间或者应用正在写文件的文件，保存在 tempFileAttribs 中，如果文件正常关闭，那么文件名和它的 FileAttributes 对象，会从 tempFileAttribs 移动到 pathToFileAttribs 中。这样，文件的文件名、文件长度和文件的内容，就都保存在内存中。代码如下：

```
private static class RawInMemoryFileSystem extends FileSystem {
    ......
    private long fsSize;
    private volatile long totalUsed;
    ......
    // 已经在 RawInMemoryFileSystem 对象中成功关闭了的文件
```

```
    private Map<String, FileAttributes> pathToFileAttribs =
            new HashMap<String, FileAttributes>();

    // 这个映射表保存已经分配了空间，但还没有写入数据或正在写入数据的文件
    // 如果文件被成功关闭，记录会移到映射表 pathToFileAttribs
    private Map<String, FileAttributes> tempFileAttribs =
            new HashMap<String, FileAttributes>();
    ......
}
```

FileSystem.rename() 用于修改文件系统中的文件名字。RawInMemoryFileSystem 的 rename() 方法的实现如下。

首先，判断目标文件是否存在，如果目标文件存在，改名操作抛出异常。这一点和普通的文件系统不太一样，即移动文件时不会自动覆盖目标文件。如果目标不存在，在映射表 pathToFileAttribs 中删除被改名的文件，然后，重新用新文件名插入到 pathToFileAttribs 中，便完成 rename() 操作。代码如下：

```
public boolean rename(Path src, Path dst) throws IOException {
    synchronized (this) {
        if (exists(dst)) { // 目标存在
            throw new IOException ("Path " + dst + " already exists");
        }
        FileAttributes fAttr = pathToFileAttribs.remove(getPath(src));
        if (fAttr == null) return false;
        pathToFileAttribs.put(getPath(dst), fAttr);
        return true;
    }
}
```

rename() 方法的逻辑很有代表性，RawInMemoryFileSystem 中其他文件和目录相关事务方法的实现都和 rename() 类似，实现涉及映射表 tempFileAttribs 和映射表 pathToFileAttribs。

InMemoryInputStream 和 InMemoryOutputStream 是 RawInMemoryFileSystem 提供的输入/输出流。当用户通过 open() 方法打开文件时，文件系统会返回输入流 FSDataInputStream，对于内存文件系统，open() 方法通过创建输入流 InMemoryInputStream，并利用该流构造 FSDataInputStream，获得打开文件后返回的流对象。代码如下：

```
private class InMemoryInputStream extends FSInputStream {
    private DataInputBuffer din = new DataInputBuffer();
    private FileAttributes fAttr;

    public InMemoryInputStream(Path f) throws IOException {
        synchronized (RawInMemoryFileSystem.this) {
            // 获取文件的 FileAttributes 对象
            fAttr = pathToFileAttribs.get(getPath(f));
            if (fAttr == null) {
                throw new FileNotFoundException("File "+f+" does not exist");
            }
            din.reset(fAttr.data, 0, fAttr.size); // 设置文件数据
```

```
        }
    }
    ......
    public int read() throws IOException {
        return din.read(); // 从 DataInputBuffer 中读取数据
    }
    ......
}

public FSDataInputStream open(Path f, int bufferSize) throws IOException {
    return new FSDataInputStream(new InMemoryInputStream(f));
}
```

InMemoryInputStream 的构造函数需要被打开文件的路径，通过这个参数在映射表 pathToFileAttribs 查找对应的 FileAttributes 对象。如果找不到 FileAttributes 对象，抛出 FileNotFoundException 异常，表明要打开的文件不存在。如果文件存在，文件的内容在 FileAttributes 对象的成员变量 data 中，文件长度在成员变量 size 中，根据这些信息，设置 InMemoryInputStream 的成员变量 din，其类型是 DataInputBuffer。

DataInputBuffer 是一个工具类，它的实现在 org.apache.hadoop.io 包中，是 DataInputStream 的子类，其主要的功能是将字节数据（如包含内存文件内容的 FileAttributes 对象成员变量 data）转换成一个 DataInputStream，然后就可以在流上通过 read() 读取数据了。

RawInMemoryFileSystem 十分简单，却包含了文件系统的基本功能。从实现者角度看，它通过内存实现了文件内容的存储和系统存储空间管理这两个在文件系统中非常重要的功能。遗憾的是，内存文件系统并没有实现目录管理相关的内容，是一个原始、简陋的系统。读者可以在目前的基础上，扩展 RawInMemoryFileSystem 支持目录操作，继续完善内存文件系统。

## 5.7　小结

本章从文件系统的一般特征开始，分别从用户角度和系统角度介绍了文件的用户界面、内部实现和保护控制。并以 Linux 本地文件系统和分布式文件系统 NFS 为例，具体分析了文件系统的实现，涉及 Linux 文件系统中的索引节点、VFS、保护机制、基于方法的应用程序编程接口和网络文件系统的文件系统模型、对分布式服务服务透明性的支持等经典设计。

设计一个令人满意的文件系统界面是件极其艰巨的任务，幸好，Java 文件系统界面为 Hadoop 提供了一个优秀的参考。Java 与文件系统相关的类库分为处理文件和目录相关事务和基于流的文件数据读写等两部分。Hadoop 抽象文件系统遵循了 Java 的设计准则，在 FileSystem 中提供了文件 / 目录 API，而文件读写则留给了相关的输入 / 输出流。同时，为支持应用访问不同的具体文件系统实现，Hadoop FileSystem 提供了打开、关闭文件系统的方法，并协助用户管理打开的文件系统，而且 Hadoop 文件系统还利用协议处理器，提供了和 Java URL 进行无缝集成的能力。

为了保证 Hadoop 应用可以访问不同环境中的数据，Hadoop 实现提供了大量的具体文件系统实现，我们研究了通过适配器模式实现本地文件系统访问的 RawLocalFileSystem、通过修饰器模式提供数据完整性检查的 ChecksumFileSystem，并解剖了一个简陋的文件系统 RawInMemoryFileSystem，了解了具体文件系统的各种实现方法。

严格地说，文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型。本章对文件系统进行了详细、深入的讨论，为后续对 Hadoop 分布式文件系统的研究提供了一个坚实的基础。

# 第三部分

# Hadoop 分布式文件系统

**本部分内容**

- HDFS 概述
- 数据节点实现
- 名字节点实现
- HDFS 客户端

# 第 6 章　HDFS 概述

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台，用户可以在 Hadoop 上管理、开发和运行处理大规模数据的应用，其中，Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）扮演了非常基础的作用，它以文件系统的形式为应用提供海量数据存储服务。

## 6.1　初识 HDFS

HDFS 作为一个分布式文件系统，具有高容错的特点，它可以部署在廉价的通用硬件上，提供高吞吐率的数据访问，适合那些需要处理海量数据集的应用程序。HDFS 没有遵循可移植操作系统接口（Portable Operating System Interface，POSIX）的要求，不支持“ls”或“cp”这样的标准 UNIX 命令，也不支持如 fopen() 和 fread() 这样的文件读写方法，而是提供了一套特有的、基于 Hadoop 抽象文件系统的 API，支持以流的形式访问文件系统中的数据。

### 6.1.1　HDFS 主要特性

HDFS 的主要特性包括：

- ❑支持超大文件。超大文件在这里指的是几百 MB、几百 GB 甚至几 TB 大小的文件，一般来说，一个 Hadoop 文件系统会存储 T（1TB=1024GB）、P（1P=1024T）级别的数据。Hadoop 需要能够支持这种级别的大文件。
- ❑检测和快速应对硬件故障。在大量通用硬件平台上构建集群时，故障，特别是硬件故障是常见的问题。一般的 HDFS 系统是由数百台甚至上千台存储着数据文件的服务器组成，这么多的服务器意味着高故障率。因此，故障检测和自动恢复是 HDFS 的一个设计目标。
- ❑流式数据访问。HDFS 处理的数据规模都比较大，应用一次需要访问大量的数据。同时，这些应用一般是批量处理，而不是用户交互式处理。HDFS 使应用程序能够以流的形式访问数据集，注重的是数据的吞吐量，而不是数据访问的速度。
- ❑简化的一致性模型。大部分的 HDFS 程序操作文件时需要一次写入，多次读取。在 HDFS 中，一个文件一旦经过创建、写入、关闭后，一般就不需要修改了。这样简单的一致性模型，有利于提供高吞吐量的数据访问模型。

正是由于以上的设计目标，HDFS 并不适合如下应用：

- ❑低延迟数据访问。低延迟数据，如和用户进行交互的应用，需要数据在毫秒或秒的范围内得到响应。由于 Hadoop 针对高数据吞吐量做了优化，而牺牲了获取数据的延迟，对于低延迟访问，可以考虑使用 HBase 或 Cassandra。

- 大量的小文件。HDFS 支持超大文件，是通过将数据分布在数据节点（DataNode），并将文件的元数据保存在名字节点（NameNode）上。名字节点的内存大小，决定了 HDFS 文件系统可保存的文件数量，虽然现在的系统内存都比较大，但大量的小文件还是会影响名字节点的性能。
- 多用户写入文件、修改文件。HDFS 中的文件只能有一个写入者，而且写操作总是在文件末。它不支持多个写入者，也不支持在数据写入后，在文件的任意位置进行修改。

总之，HDFS 是为以流式数据访问模式存储超大文件而设计的文件系统，并在普通商用硬件集群上运行。

## 6.1.2　HDFS 体系结构

为了支持流式数据访问和存储超大文件，HDFS 引入了一些比较特殊的设计，在一个全配置的集群上，“运行 HDFS”意味着在网络分布的不同服务器上运行一些守护进程（daemon），这些进程有各自的特殊角色，并相互配合，一起形成一个分布式文件系统。

HDFS 采用了主从（Master/Slave）体系结构，名字节点 NameNode、数据节点 DataNode 和客户端 Client 是 HDFS 中 3 个重要的角色。

在一个 HDFS 中，有一个名字节点和一个第二名字节点，典型的集群有几十到几百个数据节点，规模大的系统可以有上千、甚至几千个数据节点；而客户端，一般情况下，比数据节点的个数还多。名字节点和第二名字节点、数据节点和客户端的关系如图 6-1 所示。

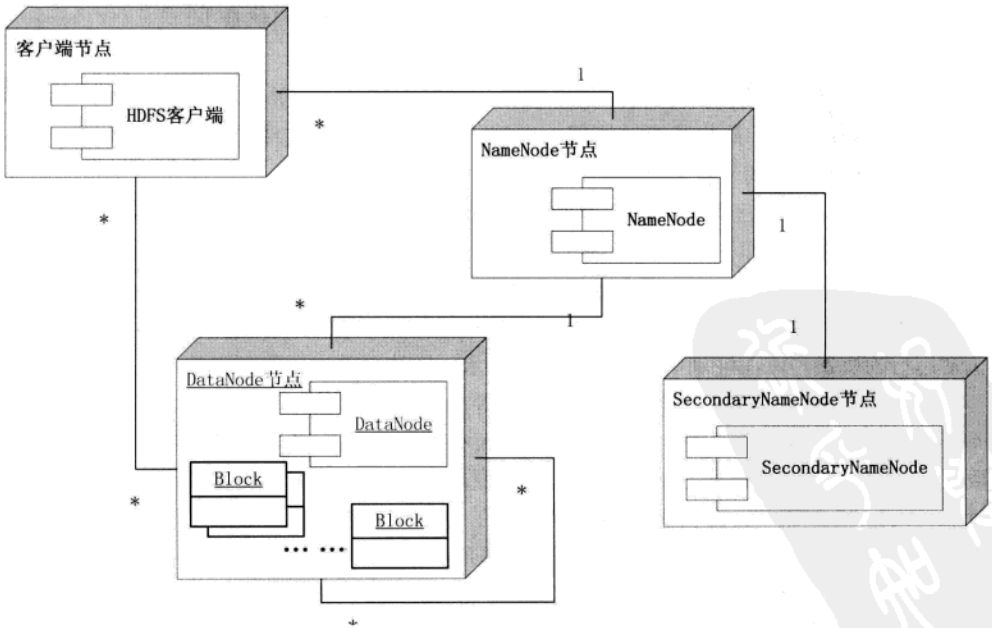



图 6-1　HDFS 体系结构

名字节点可以看作是分布式文件系统中的管理者，它负责管理文件系统命名空间、集群配置和数据块复制等。

数据节点是文件存储的基本单元，它以数据块的形式保存了 HDFS 中文件的内容和数据块的数据校验信息。

客户端和名字节点、数据节点通信，访问 HDFS 文件系统，操作文件。

**1. 数据块**

在介绍上述各实体之前，首先了解一下 HDFS 中的重要概念：数据块（Block）。

在讨论文件系统的时候，特别是在分析文件系统的实现时，我们知道，为了便于管理，设备往往将存储空间组织成为具有一定结构的存储单位。如磁盘，文件是以块的形式存储在磁盘中，块的大小代表系统读 / 写操作的最小单位；在 Linux 的 Ext3 文件系统中，块大小默认为 4096 字节。文件系统通过一个块大小的整数倍的数据块，来使用磁盘。磁盘上的数据块管理属于文件系统实现的内部细节，对于通过系统调用读写文件的用户来说，是透明的。

HDFS 也有块的概念，不过是更大的单元，默认 HDFS 数据块大小是 64MB。和普通文件系统类似，HDFS 上的文件也进行分块，块作为单独的存储单元，以 Linux 上普通文件的形式保存在数据节点的文件系统中。数据块是 HDFS 的文件存储处理的单元。

HDFS 是针对大文件设计的分布式系统，使用数据块带来了很多的好处，具体如下：

1）HDFS 可以保存比存储节点单一磁盘大的文件。

文件块可以保存在不同的磁盘上。其实，在 HDFS 中，文件数据可以存放在集群上的任何一个磁盘上，不需要保存在同一个磁盘上，或同一个机器的不同磁盘上。

2）简化了存储子系统。

简单化是所有系统的追求，特别是在故障种类繁多的分布式系统，将管理“块”和管理“文件”的功能区分开，简化了存储管理，也消除了分布式管理文件元数据的复杂性。

3）方便容错，有利于数据复制。

在 HDFS 中，为了应对损坏的块以及磁盘、机器故障，数据块会在不同的机器上进行复制（一般副本数为 3，即一份数据保存在 3 个不同的地方），如果一个数据块副本丢失或者损坏了，系统会在其他地方读取副本，这个过程对用户来说是透明的，它实现了分布式系统中的位置透明性和故障透明性。同时，一个因损坏或机器故障而丢失的块会从其他地方复制到某一个正常运行的机器，以保证副本数目恢复到正常水平。该正常水平的副本数，也称副本系数。

HDFS 的数据块比前面讨论过的磁盘块大得多，一个典型 HDFS 系统中，磁盘块的大小为 64MB，也有使用 128MB 和 256MB 数据块大小的集群。为什么在 HDFS 中要使用这么大的数据块呢？原因和在磁盘上使用大磁盘块的原理是一样的。在普通文件系统中使用较大的磁盘块，可以减少管理数据块需要的开销，如在 Linux 中可以减少保存在 i-node 中磁盘地址表中的信息链的长度；同时，在对文件进行读写时，可以减少寻址开销，即磁盘定位数据块的次数。HDFS 中使用大数据块，可以减少名字节点上管理文件和数据块关系的开销，同时，对数据块进行读写时，可以有效地减少建立网络连接需要的成本。

### 2. 名字节点和第二名字节点

名字节点（NameNode）是 HDFS 主从结构中主节点上运行的主要进程，它指导主从结构中的从节点，数据节点（DataNode）执行底层的 I/O 任务。

名字节点是 HDFS 的书记员，维护着整个文件系统的文件目录树，文件 / 目录的元信息和文件的数据块索引，即每个文件对应的数据块列表（后面的讨论中，上述关系也称名字节点第一关系）。这些信息以两种形式存储在本地文件系统中：一种是命名空间镜像（File System Image，FSImage，也称文件系统镜像），另一种是命名空间镜像的编辑日志（Edit Log）。

命名空间镜像保存着某一特定时刻 HDFS 的目录树、元信息和数据块索引等信息，后续对这些信息的改动，则保存在编辑日志中，它们一起提供了一个完整的名字节点第一关系。

同时，通过名字节点，客户端还可以了解到数据块所在的数据节点信息。需要注意的是，名字节点中与数据节点相关的信息不保留在名字节点的本地文件系统中，也就是上面提到的命名空间镜像和编辑日志中，名字节点每次启动时，都会动态地重建这些信息，这些信息构成了名字节点第二关系。运行时，客户端通过名字节点获取上述信息，然后和数据节点进行交互，读写文件数据。

另外，名字节点还能获取 HDFS 整体运行状态的一些信息，如系统的可用空间、已经使用的空间、各数据节点的当前状态等。

第二名字节点（Secondary NameNode，SNN）是用于定期合并命名空间镜像和镜像编辑日志的辅助守护进程。和名字节点一样，每个集群都有一个第二名字节点，在大规模部署的条件下，一般第二名字节点也独自占用一台服务器。

第二名字节点和名字节点的区别在于它不接收或记录 HDFS 的任何实时变化，而只是根据集群配置的时间间隔，不停地获取 HDFS 某一个时间点的命名空间镜像和镜像的编辑日志，合并得到一个新的命名空间镜像。该新镜像会上传到名字节点，替换原有的命名空间镜像，并清空上述日志。应该说，第二名字节点配合名字节点，为名字节点上的名字节点第一关系提供了一个简单的检查点（Checkpoint）机制，并避免出现编辑日志过大，导致名字节点启动时间过长的问题。

如前面所述，名字节点是 HDFS 集群中的单一故障点，通过第二名字节点的检查点，可以减少停机的时间并减低名字节点元数据丢失的风险。但是，第二名字节点不支持名字节点的故障自动恢复，名字节点失效处理需要人工干预。

### 3. 数据节点

HDFS 集群上的从节点都会驻留一个数据节点的守护进程，来执行分布式文件系统中最忙碌的部分：将 HDFS 数据块写到 Linux 本地文件系统的实际文件中，或者从这些实际文件读取数据块。

虽然 HDFS 是为大文件设计，但存放在 HDFS 上的文件和传统文件系统类似，也是将文件分块，然后进行存储。但和传统文件系统不同，在数据节点上，HDFS 文件块（也就是数据块）以 Linux 文件系统上的普通文件进行保存。客户端进行文件内容操作时，先由名字节

点告知客户端每个数据块驻留在哪个数据节点，然后客户端直接与数据节点守护进程进行通信，处理与数据块对应的本地文件。同时，数据节点会和其他数据节点进行通信，复制数据块，保证数据的冗余性。

数据节点作为从节点，会不断地向名字节点报告。初始化时，每个数据节点将当前存储的数据块告知名字节点。后续数据节点工作过程中，数据节点仍会不断地更新名字节点，为之提供本地修改的相关信息，并接受来自名字节点的指令，创建、移动或者删除本地磁盘上的数据块。

图 6-2 说明了名字节点和数据节点的角色。

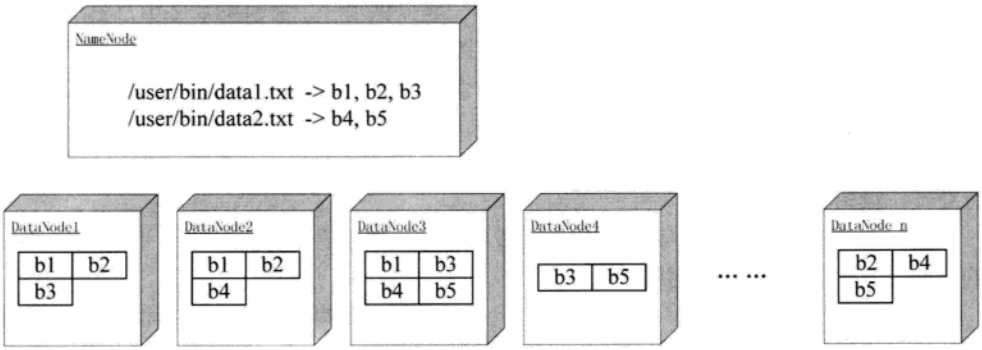



图 6-2 名字节点上的元数据和数据节点上的数据块

图中显示两个数据文件，它们都位于“/usr/bin”目录下。其中，“data1.txt”文件有 3 个数据块，表示为 b1、b2 和 b3，“data1.txt”文件由数据块 b4 和 b5 组成。这两个文件的内容分散在几个数据节点上。示例中，每个数据块都有 3 个副本。如数据块 1（属于“data1.txt”文件）的 3 个副本分布于 DataNode1、DataNode2 和 DataNode3 上，当这些数据节点中任意一个崩溃或者无法通过网络访问时，可以通过其他节点访问“data1.txt”文件。

**4. 客户端**

客户端是用户和 HDFS 进行交互的手段，HDFS 提供了各种各样的客户端，包括命令行接口、Java API、Thrift 接口、C 语言库、用户空间文件系统（Filesystem in Userspace，FUSE）等。

虽然 Hadoop 不是一个 POSIX 文件系统，不支持“ls”和“cp”这样的命令，但 Hadoop 提供了一套和 Linux 文件命令类似的命令行工具，使熟悉 Linux 文件系统的用户，可以很快使用该工具，对 HDFS 进行操作。例如，下面的命令可以在 HDFS 文件系统上创建一个目录：

```
hadoop fs -mkdir testDIR
```

通过命令行工具，可以进行一些典型的文件操作，如读文件、创建文件路径、移动文件（包括文件改名）、删除文件、列出文件列表等，同时，命令行工具也提供了本地文件和 HDFS 交互的能力，可以通过下面的命令，将本地文件上传到 HDFS：

```
hadoop fs -copyFromLocal testInput/hello.txt /user/alice/in/hello.txt
```

命令行工具提供了访问 HDFS 的基本能力，HDFS 的 Java API 提供了更进一步的功能，目前，所有访问 HDFS 的接口都是基于 Java API，包括上面介绍的命令行工具。HDFS 的 Java API 实现了上一章介绍的 Hadoop 抽象文件系统，包括 DistributedFileSystem 和对应的输入 / 输出流。

DistributedFileSystem 继承自 org.apache.hadoop.fs.FileSystem，实现了 Hadoop 文件系统界面，提供了处理 HDFS 文件和目录的相关事务；而 DFSDataInputStream 和 DFSDataOutputStream 分别实现了 FSDataInputStream 和 FSDataOutputStream，提供了读写 HDFS 文件的输入 / 输出流。

下面是一个非常简单的 HDFS Java API 的例子：

```
Path inPath  = new Path("hdfs://<ip>:<port>/user/alice/in/hello.txt");

FileSystem hdfs = FileSystem.get(inPath.toUri(), conf);

FSDataOutputStream fout = hdfs.create(inPath);

String data="testingtesting";

for(int ii=0;ii<256;ii++) {
   fout.write(data.getBytes());
}
fout.close();

FileStatus stat = hdfs.getFileStatus(inPath);
System.out.println("Replication number of file "+inPath
      +" is "+stat.getReplication());

hdfs.delete(inPath);
```

在上面的代码中，通过 FileSystem.get() 获取的文件系统是 DistributedFileSystem 实例，而通过 FileSystem.create() 创建的输出流，则是一个 DFSDataOutputStream 实例。方法 getFileStatus() 和 delete() 用于获得 HDFS 上文件状态信息和删除文件，它们都是标准的 Hadoop 文件系统方法。

HDFS 正是通过 Java 的客户端，屏蔽了访问 HDFS 的各种各样细节，用户通过标准的 Hadoop 文件接口，就可以访问复杂的 HDFS，而不需要考虑与名字节点、数据节点等节点的交互细节，降低了 Hadoop 应用开发的难度，也证明了 Hadoop 抽象文件系统的适用性。

此外，HDFS 的 Thrift 接口、C 语言库和 FUSE 等模块，它们和 HDFS 的命令行一样，都是在 Java API 的基础上开发的，用于访问 HDFS 的接口，不再详细介绍。

### 6.1.3　HDFS 源代码结构

了解了 HDFS 体系结构中的名字节点、数据节点和客户端以后，我们来分析 HDFS 实现

的源代码结构。HDFS 源代码都在 org.apache.hadoop.hdfs 包下，其结构如图 6-3 所示。

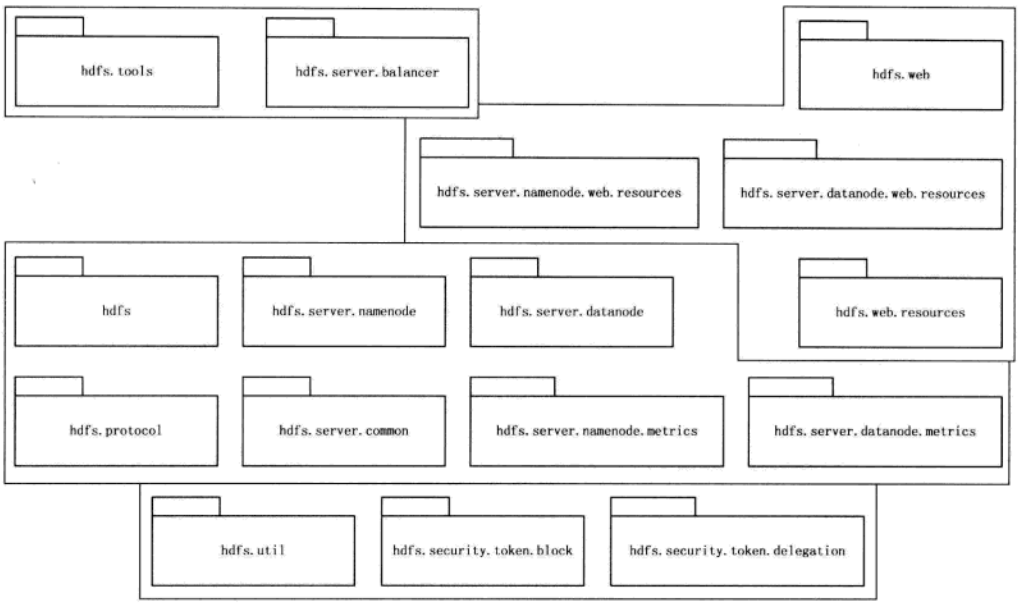



图 6-3 HDFS 实现的包结构

HDFS 的源代码分布在 16 个目录下，它们可以分为如下四类。

**1. 基础包**

包括工具和安全包。其中，hdfs.util 包含了一些 HDFS 实现需要的辅助数据结构；hdfs.security.token.block 和 hdfs.security.token.delegation 结合 Hadoop 的安全框架，提供了安全访问 HDFS 的机制。该安全特性最先是由 Yahoo 开发的，集成了企业广泛应用的 Kerberos 标准，使得用户可以在一个集群管理各类商业敏感数据。

**2. HDFS 实体实现包**

这是代码分析的重点，包含 7 个包：

hdfs.server.common 包含了一些名字节点和数据节点共享的功能，如系统升级、存储空间信息等。

hdfs.protocol 提供了 HDFS 各个实体间通过 IPC 交互的接口。

hdfs.server.namenode、hdfs.server.datanode 和 hdfs 分别包含了名字节点、数据节点和客户端的实现。上述代码是 HDFS 代码分析的重点。

hdfs.server.namenode.metrics 和 hdfs.server.datanode.metrics 实现了名字节点和数据节点上度量数据的收集功能。度量数据包括名字节点进程和数据节点进程上事件的计数，例如数

据节点上就可以收集到写入字节数、被复制的块的数量等信息。

### 3. 应用包

包括 hdfs.tools 和 hdfs.server.balancer，这两个包提供查询 HDFS 状态信息工具 dfsadmin、文件系统检查工具 fsck 和 HDFS 均衡器 balancer（通过 start-balancer.sh 启动）的实现。

### 4. WebHDFS 相关包

包括 hdfs.web.resources、hdfs.server.namenode.web.resources、hdfs.server.datanode.web.resources 和 hdfs.web 共 4 个包。

WebHDFS 是 HDFS 1.0 中引入的新功能，它提供了一个完整的、通过 HTTP 访问 HDFS 的机制。对比只读的 hftp 文件系统，WebHDFS 提供了 HTTP 上读写 HDFS 的能力，并在此基础上实现了访问 HDFS 的 C 客户端和用户空间文件系统（FUSE）。

---

**注意**　本书不对 WebHDFS 进行分析，对这个特性实现感兴趣的读者，可以参考 Apache 的 HDFS-2316 ⊖补丁的有关讨论。

---

## 6.2　基于远程过程调用的接口

接口是软件系统不同组成部分衔接的约定，一个良好的接口设计可以降低系统各部分的相互依赖，提高组成单元的内聚性，降低组成单元间的耦合程度，从而提高系统的维护性和扩展性。对于 HDFS 这样一个复杂系统，接口也是观察系统工作的一个出发点。

HDFS 的体系结构包括了名字节点、数据节点和客户端 3 个主要角色，它们间有两种主要的通信接口：

- Hadoop 远程过程调用接口。
- 基于 TCP 或 HTTP 的流式接口。

本节先分析 HDFS 各节点间的 IPC 接口。

使用 Eclipse 右键菜单中的 open type hierarchy/F4，可以获取 HDFS 各节点间的 IPC 接口，如图 6-4 所示。

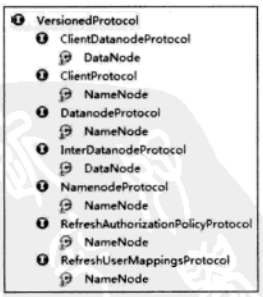



图 6-4　HDFS 各节点间的 IPC 接口

这些接口可以分为三大类。

（1）客户端相关的接口

定义在 org.apache.hadoop.hdfs.protocol 包中。具体接口包括：

- ClientProtocol：客户端与名字节点间的接口。这是一个非常重要的接口，是 HDFS 客户访问文件系统的入口。客户端通过这个接口访问名字节点，操作文件或目录的元数据信息，

⊖ WebHDFS：a complete FileSystem implementation for accessing HDFS over HTTP，https://issues.apache.org/jira/browse/HDFS-2316。

读写文件也必须先访问名字节点，接下来再和数据节点进行交互，操作文件数据。另外，从名字节点能获取分布式文件系统的一些整体运行状态信息，也是通过这个接口进行的。

- ClientDatanodeProtocol：客户端与数据节点间的接口。用于客户端和数据节点进行交互，这个接口用得比较少，客户端和数据节点间的主要交互是通过流接口进行读/写文件数据的操作。错误发生时，客户端需要数据节点配合进行恢复，或者当客户端进行本地文件读[⊖]优化时，需要通过 IPC 接口获取一些信息。

（2）服务器间的接口

包括名字节点、第二名字节点、数据节点间存在的 IPC 调用关系，其接口定义在 org.apache.hadoop.hdfs.server.protocol 包中。具体接口包括：

- DatanodeProtocol：数据节点与名字节点间的接口，这是另外一个重要的接口。在 HDFS 的主从（Master/Slave）体系结构中，数据节点作为从节点不断地通过这个接口向主节点名字节点报告一些信息，同步信息到名字节点；同时，该接口的一些方法，方法的返回值会带回名字节点指令，根据这些指令，数据节点或移动、或删除、或恢复本地磁盘上的数据块，或者执行其他操作。
- InterDatanodeProtocol：数据节点与数据节点间的接口。数据节点通过这个接口，和其他数据节点进行通信，恢复数据块，保证数据的一致性。
- NamenodeProtocol：第二名字节点、HDFS 均衡器与名字节点间的接口。第二名字节点会不停地获取名字节点上某一个时间点的命名空间镜像和镜像的变化日志，然后合并得到一个新的镜像，并将该结果发送回名字节点，在这个过程，名字节点会通过这个接口，配合第二名字节点完成元数据的合并。该接口也为 HDFS 均衡器 balancer 的正常工作提供一些信息，具体就不展开了。

（3）和安全相关的接口

分别在 org.apache.hadoop.security.authorize 包和 org.apache.hadoop.security 包中，包括 RefreshAuthorizationPolicyProtocol 和 RefreshUserMappingsProtocol。名字节点实现了这两个接口，但本书不涉及 Hadoop 的安全机制，就不进一步介绍了。

### 6.2.1 与客户端相关的接口

与客户端相关的接口包括 ClientProtocol 和 ClientDatanodeProtocol。

Hadoop 远程方法调用使得 HDFS 的两个进程实体间能够使用传统 Java 过程调用中的控制流模型来进行交互，这个交互的过程可以使用 Java 基本类型的数据，也可以使用对象作为调用参数或返回值，如果处理过程中出现异常，服务器也可以抛出异常，IPC 的调用者会收到对应的异常。在进一步讨论接口前，先来分析出现在接口上的数据类型。

---

⊖ 指的是当读某一数据块的客户端，如某个 Mapper 正好位于提供数据的数据节点主机时，客户端可以直接通过本地文件系统读数据块的优化。这个时候，数据不再需要由数据节点读取，再通过 Socket 传送到客户端，减少了交互的环节。

客户端相关接口上的类不但为 ClientProtocol 和 ClientDatanodeProtocol 服务，同时也应用于服务器间接口的交互，包含很多 HDFS 中基本概念的抽象。图 6-5 是相关的类图。

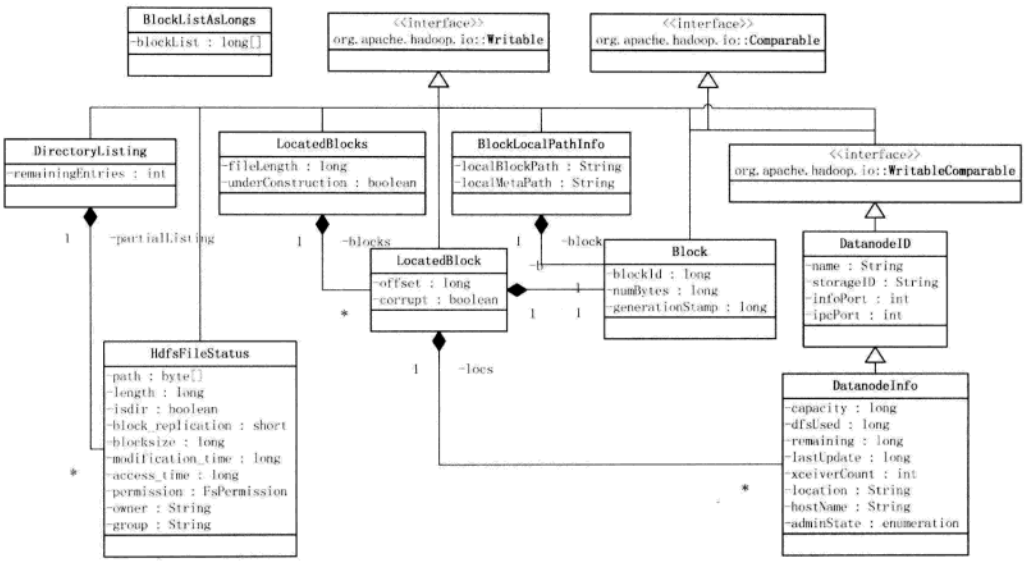



图 6-5　ClientProtocol 和 ClientDatanodeProtocol 中的类

类图中的类可以分为 3 种：和数据块相关、和数据节点相关、和 HDFS 文件元数据相关。

（1）数据块相关

通过前面的分析已经了解了，HDFS 将文件内容分为数据块（典型配置是 64MB），多副本（典型配置是 3 个副本）保存在数据节点上。

数据块在 HDFS 中的抽象是 org.apache.hadoop.hdfs.protocol.Block，它包含 3 个成员变量，都是长整型：

❑blockId：数据块的唯一标识，即数据块的 ID 号。

❑numBytes：数据块包含的文件数据大小。

❑generationStamp：数据块的版本号，或数据块时间戳。

需要特别说明的是属性 generationStamp，每次对数据块做改动后，版本号都会修改。版本号用于数据一致性检查，例如当成功在数据块后面添加数据后，就会修改数据块的版本号。具有相同数据块标识，但版本号不一致的数据块，最少有一个是或都是无效的、需要被删除的数据块。

Block 类的实现包含了多个构造方法和对上述字段操作的一些方法，并实现了 Writable 和 Comparable 接口，这些方法都很简单。需要注意的是，Block 类还维护了一个“关联”影子成员变量：数据块名。如果某个数据块的 ID 号是 <id>，则它的数据块名为 blk_<id>。例如，假设某一个数据块的 ID 号是 3148782637964391313，那么其数据块名为 blk_3148782637964391313。数据块名应用于数据节点中，作为数据块保存在 Linux 文件系统上的文件名的一部分，Block.

getBlockName()、Block.isBlockFilename()、Block.filename2id() 等方法都与数据块名相关。

Block 是大量和数据块相关的类的基础，在客户端接口上，这样的类有 LocatedBlock、LocatedBlocks 和 BlockLocalPathInfo。

顾名思义，LocatedBlock 就是已经确认了存储位置的数据块。其成员变量除了对应的数据块 b 外，还包括数据块在对应文件中的偏移量 offset、数据块所在的数据节点信息 locs 和数据块是否损坏标志 corrupt 等。locs 是一个类型为 DatanodeInfo 的数组，包含了所有可用的数据块位置，损坏的数据块对应的数据节点信息，则不会出现在数组里。获得数据块的位置信息，是读写数据块的前提条件。上述成员变量定义如下：

```
public class LocatedBlock implements Writable {
    ......
    private Block b;
    private long offset;// 数据块在对应文件中的偏移量
    private DatanodeInfo[] locs;// 保存数据块的数据节点
    private boolean corrupt;// 数据块是否已经损坏
    private Token<BlockTokenIdentifier> blockToken
          = new Token<BlockTokenIdentifier>();// 安全相关
    ......
}
```

LocatedBlocks 可用于一次定位多个数据块，它包含了一系列的 LocatedBlock 对象。LocatedBlocks 的成员变量还包括对应文件长度 fileLength 和该文件是否处于构建状态标志 underConstruction。在 HDFS 中，某文件处于构建状态，是指某客户端正在写文件，这时候，LocatedBlocks 对象中获得的文件长度 fileLength，后续可能会发生变化。

BlockLocalPathInfo 应用于 ClientDatanodeProtocol 接口中，用于 HDFS 读文件的数据节点本地读优化[⊖]。当客户端发现它和它要读取的数据块正好位于同一台主机上的时候，它可以不通过数据节点读数据块，而是直接读取本地文件，以获取数据块的内容。这大大降低了对应数据节点的负载，对于有大量本地读的 MapReduce 应用，是一个十分有意义的优化。

当客户端执行本地读优化时，需要通过 ClientDatanodeProtocol 获取数据块对应的本地文件，数据节点通过 BlockLocalPathInfo 返回本地文件的路径，保存在成员变量 localBlockPath 中。成员变量 localMetaPath 是数据块校验信息文件的路径，该文件保存了数据块的校验信息。有了这两个路径，客户端就可以直接读本地文件，而不需要再通过网络访问数据节点，获取文件数据。

（2）DatanodeID 和 DatanodeInfo

图 6-5 中，另一组类和数据节点相关，它们是 DatanodeID 和 DatanodeInfo。DatanodeInfo 继承自 DatanodeID，在 DatanodeID 的基础上，提供了数据节点上的一些度量信息。

数据节点标识 DatanodeID 用于在 HDFS 集群中确定一个数据节点，它的成员变量包括：

❑name：数据节点使用 IP 套接字地址作为它的名字，name 是一个字符串，可以是 IP

⊖ HDFS-2246 是本地读优化对应的 JIRA，提供了这个优化的更多信息，地址：https://issues.apache.org/jira/browse/HDFS-2246。

地址、端口号对，也可以是主机名、端口号对，其中，端口是数据节点的流接口地址，后续我们会继续讨论数据节点的流接口。

- storageID：数据节点的存储标识，当数据节点用不同的存储标识在名字节点上注册时，名字节点通过这个标识，了解到这是一个被重新使用的数据节点。
- infoPort：数据节点 WWW 服务器的监听端口，通过这个端口可以使用 HTTP/HTTPS 协议访问数据节点。
- ipcPort：数据节点 IPC 服务器监听端口，对于客户端，该端口提供了 ClientDatanodeProtocol 接口中定义的服务。

从 DatanodeID 中可以获得数据节点的主机地址，同时还可以了解到数据节点的流接口端口、WWW 服务端口和 IPC 服务端口 3 个重要端口的信息。

对比 DatanodeID，DatanodeInfo 提供的附加状态信息包括：容量、已经使用容量、剩余容量、状态更新时间、流接口服务线程数、数据节点在集群中的位置、数据节点节点状态等信息。通过这些信息可以分析数据节点的负载状态，应用于资源调度、数据节点服务选择等场景中。

（3）HdfsFileStatus 和 DirectoryListing

出现在 ClientProtocol 接口上的最后一组类是 HdfsFileStatus 和 DirectoryListing。HdfsFileStatus 保存了 HDFS 文件 / 目录的属性，DirectoryListing 用于一次返回一个目录下的多个文件 / 子目录的属性，它们都用于实现 FileSystem.getFileStatus() 方法族。

### 1. ClientProtocol

客户端和名字节点间的通信协议定义在 ClientProtocol 接口，一共包含 34 个方法，参考图 6-6。这些方法可以分两组，一组和实现前面介绍的 Hadoop 文件系统相关，另一组用于查询、设置 HDFS 的状态，多功能工具 dfsadmin 的实现依赖这组接口。

ClientProtocol 是一个冗长的接口，包含了大量的方法。ClientProtocol 提供的能力可以分两大类：

- 用于实现 Hadoop 文件系统相关功能的能力；
- 用于对 HDFS 状态进行查询、设置的能力。

其中，第一部分又可以细分为 3 类，包括文件 / 目录元数据部分的操作、写文件相关的操作和读文件相关的操作。

ClientProtocol 中包含的元数据操作接口函数，和 Hadoop 抽象文件系统中相应的方法有着对应关系，方法虽多，但却简单；客户端读文件时，名字节点配合的工作比较少，只涉及 getBlockLocations() 和 reportBadBlocks() 两个方法；最复杂的部分出现在写文件（包括创建文件），一共涉及 8 个 ClientProtocol 成员函数，包括：用于打开文件的 create() 和 append()、用于追加数据块的 addBlock() 和 abandonBlock()、用于持久化文件信息的 fsync()、用于关闭文件的 complete() 和与租约相关的 renewLease() 和 recoverLease()。

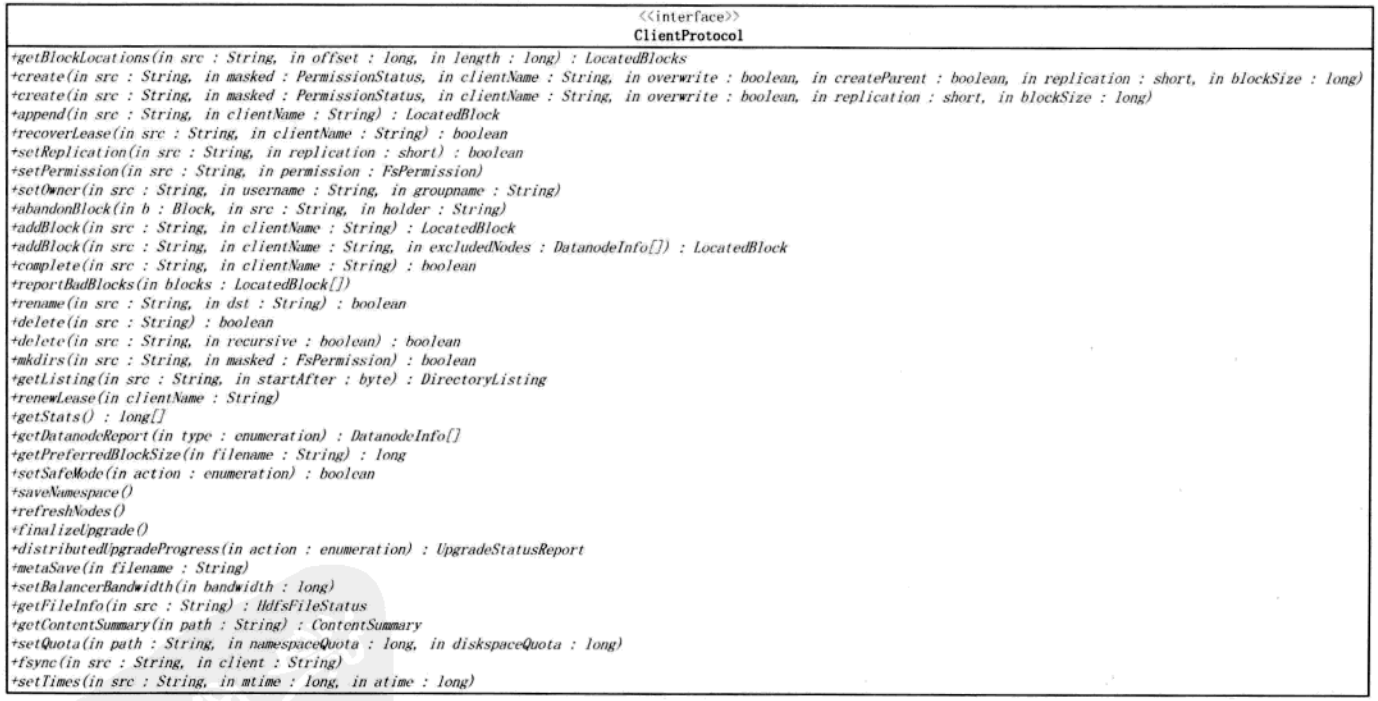



图 6-6 ClientProtocol 接口

用于管理 HDFS 状态，实现多功能工具 dfsadmin 的一共有 7 个接口方法，和实现客户端 API 部分的方法相比较，这 7 个方法虽然意义明确，也较简单，涉及管理名字节点的很多概念，包括安全模式、委托节点和撤销节点、升级等，后续会逐步展开介绍相关的细节。

（1）文件 / 目录相关事务的方法

表 6-1 中展示了 HDFS 客户端实现文件系统相关 API 时，需要使用到的 ClientProtocol 方法。

**表 6-1　ClientProtocol 接口上的操作**

| Hadoop FileSystem 操作 | ClientProtocol 接口上的操作 | 描　述 |
|---|---|---|
| URL.openStream<br>FileSystem.open<br>FileSystem.create<br>FileSystem.append | create<br>append<br>getBlockLocations | 打开一个文件。可以通过创建文件，即 create()，或是在已有文件上追加（append()），为写操作打开文件。也可以通过 open()，为读操作打开文件，这时需要通过 getBlockLocations() 获取数据块的保存位置 |
| FSDataInputStream.read | getBlockLocations<br>reportBadBlocks | 从文件中读数据。读数据的 read() 操作会转换成对数据块的读操作，需要通过 getBlockLocations() 获得数据块的保存位置。读数据发现数据有错误的时候，通过 reportBadBlocks()，报告给名字节点 |
| FSDataOutputStream.write | addBlock<br>abandonBlock | 向文件中写入数据。写数据过程中，如果需要新的数据块，则必须通过 addBlock()，向名字节点申请。写数据出现错误时，客户端可以通过 abandonBlock()，放弃申请到的数据块 |
| FSDataInputStream.close<br>FSDataOutputStream.close | fsync<br>complete | 关闭一个文件。关闭文件时，先调用 fsync()，持久化数据，然后调用 complete() 方法，通知名字节点，关闭文件 |
| FSDataInputStream.seek | getBlockLocations | 改变文件读写位置，有时需要通过 getBlockLocations()，重新定位数据块位置 |
| FileSystem.getFileStatus<br>FileSystem.getContentSummary<br>FileSystem.get* | getFileInfo<br>getContentSummary | getFileInfo() 用于获取文件 / 目录的属性；getContentSummary() 可获得文件 / 目录使用的存储空间信息 |
| FileSystem.set* | setPermission<br>setOwner<br>setTimes<br>setReplication | 修改文件属性。setPermission()、setOwner()、setTimes() 和 setReplication() 分别用于修改文件的权限、文件主 / 组、文件的修改时间 / 访问时间和文件的副本数 |
| FileSystem.createNewFile | N/A | 创建一个文件。在 fs.FileSystem 中，它的默认实现是：通过 create() 创建文件，然后立即通过输出流的 close() 方法关闭文件，从而达到创建一个新的空文件的目的 |
| FileSystem.delete | delete | 从文件系统中删除一个文件 / 目录，同名方法 |
| FileSystem.rename | rename | 更改文件名，同名方法 |
| FileSystem.mkdirs | mkdirs | 在给定目录下创建一个子目录，同名方法 |
| FileSystem.listStatus | getListing | 读取一个目录下的项目 |
| FileSystem.<br>setWorkingDirectory | N/A | |
| FileSystem.<br>setWorkingDirectory | N/A | |

对于大多数文件 / 目录操作，HDFS 客户端提供的 API 和 ClientProtocol 中的方法基本是对应的，以修改文件 / 目录名为例，在 Hadoop 抽象文件系统中，它的定义是：

```
/**
 * 将 src 的名字改为 dst.
 */
public abstract boolean rename(Path src, Path dst) throws IOException;
```

在 ClientProtocol 中对应的方法为：

```
/**
 * 修改文件系统中某一文件或目录的名字，其中参数 src 是原文件或目录的名字，dst 是新
 * 名字。如果操作成功，返回 true，否则，返回 false（被修改的名字已经存在或目标名字
 * 已经存在）。当新名字不合法时，抛出 IOException 异常，如果修改后结果违反文件配额，
 * 则抛出 QuotaExceededException 异常
 */
public boolean rename(String src, String dst) throws IOException;
```

可以发现，这两个方法基本没有什么差别。ClientProtocol 接口中，类似的方法还有 setPermission()、setOwner()、setTimes()、setReplication()、delete()、getContentSummary() 等，而获取文件目录的属性的 getListing()、getFileInfo()，也和 Hadoop 抽象文件系统中相应的方法差别不大。

大多数处理文件和目录相关事务的操作，ClientProtocol 都提供了和 Hadoop FileSystem 中对应 API 类似的方法，便于客户端的实现。但 ClientProtocol 中用于支持读写文件数据相关的方法，包括打开文件、输入 / 输出流和关闭文件等操作，就和 HDFS 的特性、体系结构密切相关了。

Hadoop 抽象文件系统中提供了 create() 方法，用于创建新文件并获得输出流；或者通过 append() 方法，在一个已有文件上获得输出流；而 open() 方法则用于打开文件并构造输入流。在 ClientProtocol 中，虽然也有同名的 create() 和 append() 方法，但它们只是实现 FileSystem 中同名方法的某一个步骤。

（2）写数据需要的方法

ClientProtocol.create() 在 HDFS 目录树中创建一个新文件，它有两种形式，都需要大量的参数。按照函数注释的说明，create() 方法会在 HDFS 目录树中创建一个新的空文件，其路径由参数 src 提供。注意，名字节点没有“当前路径”的概念，传入的路径参数必须是绝对路径。通过 ClientProtocol.create() 创建文件后，该文件对其他用户可见可读。create() 方法的声明如下：

```
public void create(String src,
                   FsPermission masked,
                   String clientName,
                   boolean overwrite,
                   boolean createParent,
                   short replication,
                   long blockSize) throws IOException;
.....
```

```
public void create(String src,
                   FsPermission masked,
                   String clientName,
                   boolean overwrite,
                   short replication,
                   long blockSize) throws IOException;
```

ClientProtocol.create() 除了文件路径 src 外，还需要 6 个参数，其中，masked 携带了文件的权限，replication 和 blockSize 则是被创建文件的数据块副本数和数据块大小，这些都是会持久化到命名空间镜像的文件元数据，特别是 replication 和 blockSize，它们是 HDFS 特有的。标志位 overwrite 和 createParent 用于控制是否覆盖已经存在的文件和是否（递归）创建不存在的父目录。需要特别关注的参数是客户端名字 clientName，名字节点需要通过 clientName，维护和这个客户端关联的一些状态信息，后续还会继续讨论这些状态信息。

与 create() 比，ClientProtocol.append() 就简单得多了。追加操作不需要创建文件，也就不需要携带大量创建文件时需要的元数据和标志位。这两个操作还有一个不同点，append() 会返回一个 LocatedBlock 对象。通过这个返回值，客户端获得追加的文件数据应该写往哪个数据块以及数据块的位置信息，接下来就可以联系数据节点，开始追加数据。代码如下：

```
public LocatedBlock append(String src, String clientName)
      throws IOException;
```

了解了 append() 方法后，读者可能会问，通过 create() 创建文件后，数据应该写到什么地方呢？ ClientProtocol 的成员函数 addBlock() 用于为当前客户端打开的文件添加一个数据块。当用户创建一个新文件，或者写满一个数据块后，客户端就会调用这个方法，向名字节点申请一个新的数据块。

addBlock() 方法有两种形式，它们都有参数 src 和 clientName，分别是对应的文件名和客户端名字。在各个远程调用之间，Hadoop IPC 不会在服务器端保存会话信息，所以，调用 addBlock() 时，客户端还是需要提供上述上下文信息。当前版本的 ClientProtocol 建议使用 3 个参数的 addBlock() 方法，附加的 excludedNodes 参数可用于通知名字节点，在选择数据目标时排除某些数据节点。和 append() 方法一样，addBlock() 返回一个 LocatedBlock 对象，提供了写数据时需要的相关信息。addBlock() 方法有两种形式，代码如下：

```
//@deprecated use the 3-arg form below
public LocatedBlock addBlock(String src, String clientName)
      throws IOException;
.....
public LocatedBlock addBlock(String src, String clientName,
      DatanodeInfo[] excludedNodes) throws IOException;
.....
public void abandonBlock(Block b, String src, String holder)
      throws IOException;
```

addBlock() 方法是我们介绍的，ClientProtocol 接口中与数据块相关的第一个方法，其他数据块方法包括 abandonBlock()、getBlockLocations() 和 reportBadBlocks() 等。

abandonBlock() 方法用于放弃一个数据块。这并不是一个常见的操作，当客户端联系不上名字节点通过 addBlock() 返回的数据节点时，需要通过 abandonBlock() 方法，明确放弃这个数据块，并重新申请新的数据块。下面是一个使用 abandonBlock() 方法的典型场景：

假设客户端通过 addBlock()，向名字节点申请了一个新数据块，同时，假设申请的文件副本数是 1，这时，addBlock() 返回的 LocatedBlock 对象会提供一个数据节点的地址，用于存放该数据块。如果在 addBlock() 返回后，开始写数据前，对应的数据节点因为某种原因崩溃了，客户端联系不上这个节点，在这种情况下，客户端就需要通过 abandonBlock() 通知名字节点，放弃此数据块，并再次调用 addBlock() 方法申请新的数据块。在申请新数据块时，需要将原有数据节点的信息放入 addBlock() 的 excludedNodes 参数，保证名字节点不会再次选中并返回这个已经崩溃的节点。

---

**注意** *数据节点崩溃后，名字节点不会立即感知到变化，而是在一定的超时时间后，才能做出数据节点不存在的判断，从这个角度看，abandonBlock() 也是 HDFS 中一个很有特色的设计。*

---

在 ClientProtocol 接口中，和写文件直接相关的还有两个成员函数：fsync() 和 complete()。调用 ClientProtocol 接口的 fsync() 方法，名字节点会持久化对应文件的相关信息，包括文件的元数据、文件包含的数据块、当前打开这个文件的客户端的标识等。

熟悉 Linux 的读者，应该知道 Linux 文件 API 也提供了同名方法，但这两个方法差异迥大。在 Linux 中，大多数磁盘 I/O 操作都会通过内核缓冲区，如果缓冲区尚未写满，则不会发生实际的 I/O 操作。这种机制叫延迟写，它的优点是减少了磁盘 I/O 操作的次数，但缺点是一旦系统出现故障，可能会丢失那些在缓冲区但未写进磁盘的文件数据。

Linux 的 fsync() 用于确保被修改过的文件内容会持久化到磁盘，可用于实现数据库事务这样有数据一致性要求的应用。ClientProtocol.fsync() 不具备 Linux fsync() 那样的功能，它只保证名字节点上对文件元数据的修改会被保存到磁盘，不保证写到数据节点上的数据的持久化。它的声明如下：

```
public void fsync(String src, String client) throws IOException;
......
public boolean complete(String src, String clientName)
      throws IOException;
```

无论是通过 create() 创建的文件还是通过 append() 打开的文件，在写数据结束后，都应该将其关闭，complete() 用于实现输出流的 close() 方法。和 fsync() 类似，complete() 方法只和名字节点通信，告知客户端已经完成写文件的操作。

从 create() 方法开始，我们依次介绍了 create()、append()、addBlock()、abandonBlock()、fsync() 和 complete() 六个和客户端写数据相关的 ClientProtocol 接口函数，勾画了在客户端和名字节点工作正常的情况下，写数据时名字节点需要向客户端提供的功能。但是，如果客户端或名字节点中的某一个实体崩溃了，怎么能够从错误中恢复过来呢？

租约是 HDFS 实现中的一个重要概念，用于解决上述问题。

**注意**　在名字节点，租约记录了客户端为写数据而打开文件的信息，也可以理解为名字节点将写特定文件的权限授权给该客户端，客户端通过 renewLease() 定时更新它的租约。这个方法很简单，只需要一个参数，即客户端的名称。

客户端调用 ClientProtocol.renewLease()，相当于往名字节点发送心跳信息，如果名字节点长时间没有收到客户端的租约更新，就可以认为客户端已经崩溃，名字节点会试图关闭文件，以防止资源泄漏。

如果客户端从崩溃中恢复，并试图继续未完成的写文件操作，这时候，recoverLease() 就派上用场了，如同方法名，它用于恢复租约。ClientProtocol.recoverLease() 比 renewLease() 多一个参数，字符串 src 指定了需要进行租约恢复的文件路径，如果方法返回 true，表明这个文件已经被成功关闭，客户端可以通过 append() 打开文件，继续写数据。

如果名字节点出现问题，发生崩溃，怎么恢复名字节点上的租约信息呢？客户端创建文件或通过追加打开文件时，名字节点都会将变化记录到命名空间镜像的编辑日志中，所以，名字节点从错误中恢复时，它可以根据日志，恢复名字节点上的租约信息。

HDFS 客户端在写文件时，涉及系统的多个节点，这些节点都有可能发生故障，这使得故障恢复需要考虑多种场景，租约很好地解决了这些问题。在后面的章节，会更深入地讨论并研究租约。

（3）读数据使用的方法

前面提及，ClientProtocol 中和数据块直接相关的方法有 4 个，已经讨论过的有 addBlock() 和 abandonBlock()，用于实现写数据的输出流，另外两个数据块相关方法：getBlockLocations() 和 reportBadBlocks()，应用于 HDFS 客户端实现文件读，声明如下：

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

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

getBlockLocations() 方法使用文件名、数据起始地址和长度 3 个参数，获得一个 LocatedBlocks 对象。我们知道，LocatedBlocks 用于一次定位多个数据块，函数的返回值包含了一系列的 LocatedBlock 对象，客户端通过这些对象就可以访问数据节点，获得文件数据。接口成员函数 getBlockLocations() 一次返回多个已经定位的数据块，可以有效降低名字节点的负载，在分布式系统中，这是一种比较常见的优化手段。

reportBadBlocks() 也用于 HDFS 输入流的实现中，当发现输入数据有错误时，客户端通过这个方法向名字节点进行汇报。输入流使用的对数据进行检查的方法，就是第 5 章中介绍的数据校验技术。数据节点按数据块组织数据，如果客户端发现读入数据有校验错误，意味着该数据块已经损坏，不能再使用，所以 reportBadBlocks() 不需要上报错误发生的具体位置，只需向名字节点告知出错的数据块及所在的数据节点即可。

（4）工具 dfsadmin 依赖的方法

ClientProtocol 接口中剩余的方法用于实现多功能工具 dfsadmin，该工具主要由 HDFS

管理员使用。

成员函数 setQuota() 用于实现与配额相关的功能。HDFS 提供了两种配额：

❑目录配额设置了目录树中文件或者目录的数量限制，有助于防止用户创建大量的小文件；

❑空间配额设置了可在目录树中存储的文件大小限制。

调用 setQuota() 时需要指明路径、目录配额（类型为 long）和空间配额（类型为 long）三个参数。对于配额，可以进行维持（原有配额不变）、设置、取消和修改四种操作。setQuota() 如何在一个长整型的参数里，实现这三种操作呢？

处理目录配额或空间配额参数时，有两个特殊的值：

❑FSConstants.QUOTA_DONT_SET：用于维持原有配额，值是 Long.MAX_VALUE。

❑FSConstants.QUOTA_RESET：用于取消配额，值为 –1。

其他值（从 –1 到 Long.MAX_VALUE 的开区间）则用于设置或修改配额。代码如下：

```
public interface FSConstants {
    ……
    // 特殊的配额值
    public static final long QUOTA_DONT_SET = Long.MAX_VALUE;
    public static final long QUOTA_RESET = -1L;
    ……
}

public interface ClientProtocol{
    ……
    public void setQuota(String path, long namespaceQuota,
                long diskspaceQuota)throws IOException;
    ……
}
```

方法 setSafeMode() 涉及 HDFS 的一个概念：安全模式。

安全模式是 HDFS 的一种工作状态，处于这种工作状态的 HDFS，只向客户端提供文件的只读视图，不接受对名字空间的更改，包括创建文件、删除目录等操作；同时，名字节点也不会进行数据块的复制或删除，如副本数小于正常水平的数据块复制。如果要查看名字节点是否处于安全状态，可以通过工具 dfsadmin 的 safemode 命令，获得名字节点的工作状态。同时，通过 safemode 命令，管理员可以让名字节点进入或者退出安全状态。该命令的实现，就是通过 ClientProtocol.setSafeMode() 完成的。代码如下：

```
public interface FSConstants {
    ……
    // 安全模式动作
    public enum SafeModeAction{ SAFEMODE_LEAVE, SAFEMODE_ENTER,
            SAFEMODE_GET; }
    ……
}

public interface ClientProtocol{
    ……
```

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

setSafeMode() 方法需要一个枚举参数 action，其枚举类型定义在 FSConstants.SafeModeAction 中，包括三个可能的值：SAFEMODE_LEAVE 和 SAFEMODE_ENTER 分别用于使名字节点退出、进入安全模式，枚举值 SAFEMODE_GET 则用于状态查询。

在安全模式下，dfsadmin 的管理命令 saveNamespace 将会把当前内存中的文件系统镜像保持到一个新的命名空间镜像（fsimage）中，并重置日志文件。

另一个类似的方法是 metaSave()，用于将名字节点中的主要数据结构保存到 Hadoop 日志目录的指定文件（metaSave() 函数的唯一参数 filename）中。该文件包含的信息有：名字节点收到的数据节点的心跳信号、等待被复制的数据块、正在被复制的数据块、等待被删除的数据块等。这些信息有利于管理员了解系统的健康情况。和 saveNamespace() 不同，调用 metaSave() 时，不要求系统处于安全模式中。

顾名思义，ClientProtocol.refreshNodes() 用于刷新节点，接受名字节点管理的只有数据节点，可见，该方法用于更新数据节点的设置。作为 Hadoop 集群的管理员，可能随时需要添加或者移除节点。添加节点或委托节点，可以为增加集群的存储容量；相反，如果要缩小（Scale in）集群或者某些节点频繁发生错误，可以移除节点（撤销节点）。refreshNodes() 不带参数，它所需要的信息保存在两个文件中：include 文件（一般在配置目录下）用于指定可以连接到名字节点的数据节点列表，exclude 文件指定要移除的数据节点。

ClientProtocol 中的最后两个成员方法 distributedUpgradeProgress() 和 finalizeUpgrade() 用于升级系统，其中 distributedUpgradeProgress() 用于查询升级的进度，finalizeUpgrade() 则用于提交升级。Hadoop 对自身的兼容性要求非常高，不同版本间不提供相互的兼容性，而系统升级又涉及大量的节点和数据，一个合理的操作步骤就非常有必要了。通过“-upgrade”选项启动 HDFS 可以开始一次系统升级，由于启动系统的升级不需要通过 dfsadmin 工具，在 ClientProtocol 没有对应的接口函数，但升级启动以后，便可以通过 distributedUpgradeProgress() 方法使用不同的参数，获得当前系统的升级状态、升级状态的细节、或者进行强制升级操作。

finalizeUpgrade() 方法用于提交 HDFS 的升级操作，只有通过提交，才能最终完成一次 HDFS 升级。finalizeUpgrade() 方法会让数据节点和名字节点执行一些升级后的清理操作，清理操作完成后，升级过程就不可以回滚了，至此，HDFS 升级成功完成。

总的来说，ClientProtocol 提供的能力可以分以下两大类：用于实现 Hadoop 文件系统相关功能的能力和用于对 HDFS 状态进行查询、设置的能力。

其中，第一部分又可以细分为三类，包括文件 / 目录元数据部分的操作、写文件相关的操作和读文件相关的操作。

ClientProtocol 中包含的元数据操作接口函数，和 Hadoop 抽象文件系统中相应的方法有着对应关系，方法虽多，但却简单；客户端读文件时，名字节点配合的工作比较少，只

涉及 getBlockLocations() 和 reportBadBlocks() 两个方法；最复杂的部分出现在写文件，它（包括创建文件）一共涉及了 8 个 ClientProtocol 成员函数，包括用于打开文件的 create() 和 append()，用于追加数据块的 addBlock() 和 abandonBlock()，用于持久化文件信息的 fsync()，用于关闭文件的 complete() 和与租约相关的 renewLease() 和 recoverLease()。

用于管理 HDFS 状态，实现多功能工具 dfsadmin 的接口方法一共有 7 个，和实现客户端 API 部分的方法相比较，这 7 个方法虽然意义明确，也较简单，但涉及了管理名字节点的很多概念，包括安全模式、委托节点和撤销节点、升级等，后续会逐步展开相关的细节。

2. ClientDatanodeProtocol

和 ClientProtocol 相比，ClientDatanodeProtocol 接口相当简单，如图 6-7 所示，它只有三个方法：recoverBlock()、getBlockInfo() 和 getBlockLocalPathInfo()。

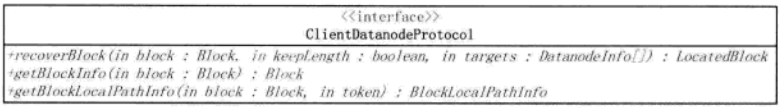



图 6-7 ClientDatanodeProtocol 接口

recoverBlock() 方法应用于 HDFS 的客户端 DFSClient 的输出流 DFSOutputStream 中，客户端往数据节点输出数据的过程中，如果某个副本所在的数据节点出现错误，客户端就会尝试进行数据块恢复，这时候需要调用 recoverBlock()，从正常工作的数据节点中找到恢复点，然后才能继续输出数据。本书的后面将会讨论输出流 DFSOutputStream 中的错误恢复机制，到时再来仔细研究这个接口方法的具体使用方法。

getBlockInfo() 和 HDFS 的文件一致性模型相关。文件一致性模型描述了对文件进行读写时数据的可见性，在 POSIX 系统中，如 Linux，进程往某一个文件里写数据，另一个进程立即就可以读到这些数据。在 HDFS 中，设计者简化了数据的一致性问题，和 POSIX 系统不同，某个进程已经写入的文件数据，另一个进程不一定能立即读到。getBlockInfo() 返回数据节点上指定数据块的信息，它被 DFSInputStream 的 updateBlockInfo() 使用。当 DFSInputStream 对象读取一个被打开的，其他进程正在写入数据的文件数据块时，可以通过这个方法，“感知”到数据的变化，读到最新的数据。getBlockInfo() 方法的输入是 Block 对象，输出是（可能）更新过的，反映数据变化的新 Block 对象。

如前面讨论过的，BlockLocalPathInfo 类应用于本地读优化，执行本地读的客户端通过 getBlockLocalPathInfo() 成员函数，获得某个数据块对应的数据块文件及数据块校验信息文件的本地路径，然后就可以做进一步的本地读操作了。

在一般情况下，客户端和数据节点主要通过基于流的接口进行交互，较少使用 ClientDatanodeProtocol 中提供的方法。

## 6.2.2 HDFS 各服务器间的接口

HDFS 各服务器间的接口一共有三个，它们是 DatanodeProtocol、InterDatanodeProtocol

和 NamenodeProtocol，这些接口都定义在 org.apache.hadoop.hdfs.server.protocol 包中。

### 1. DatanodeProtocol

继 ClientProtocol 接口后，我们接触到的另一个重要接口是 DatanodeProtocol，用于数据节点和名字节点间的通信，数据节点是客户端，名字节点是服务器，如图 6-8 所示。

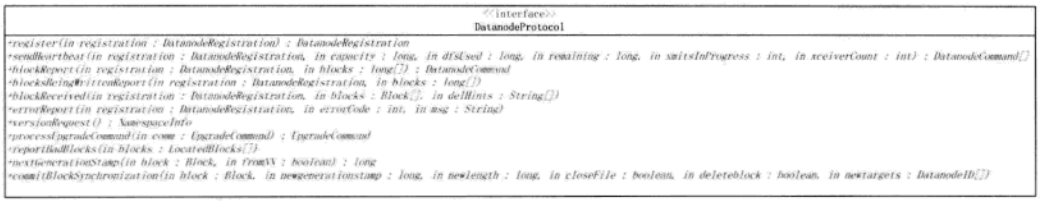



图 6-8　DatanodeProtocol 接口

前面在讨论 HDFS 的体系结构时提到，名字节点维护着整个文件系统的文件目录树，以及每个文件对应的数据块列表，这些信息存储在本地文件系统中。但 HDFS 工作时需要的数据块所在的数据节点位置信息，则在名字节点每次启动时都会重建。另一方面，数据节点在初始化时，就会将当前存储的数据块告知名字节点，后续工作过程中，数据节点仍会不断地更新名字节点，为之提供本地数据块的变化信息，特别是客户端写数据时引起的数据块变化，并接受来自名字节点的指令，创建、移动或者删除本地磁盘上的数据块。DatanodeProtocol 接口中的 11 个方法，就是为了满足上面这些需求而设计的。

（1）DatanodeRegistration 类和 NamespaceInfo 类

在具体介绍这些方法之前，先来分析 DatanodeRegistration 类和 NamespaceInfo 类，如图 6-9 所示。

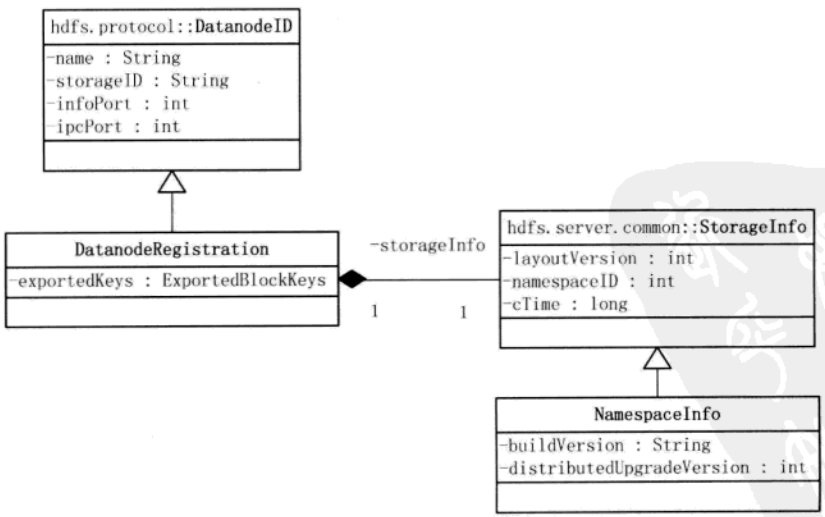



图 6-9　DatanodeRegistration 和 NamespaceInfo

DatanodeRegistration 被大量的 DatanodeProtocol 方法使用。由类的名字可知，DatanodeRegistration 用于数据节点注册。它继承自 DatanodeID，在介绍 ClientProtocol 时，我们已经知道 DatanodeID 可用于在 HDFS 中确定一个数据节点，由此可见，在数据节点注册的时候还必须包含其他的信息，即 DatanodeRegistration 在 DatanodeID 基础上新增加的成员变量 exportedKeys 和 storageInfo，其中 exportedKeys 用于 HDFS 的安全特性，我们不再分析，类型为 StorageInfo 的另一个变量保存了数据节点的存储系统信息。它包括三个成员变量：

❑layoutVersion：是一个负整数，保存了 HDFS 存储系统信息结构的版本号。

❑namespaceID：存储系统的唯一标识符。

❑cTime：该存储系统信息的创建时间。

当数据节点注册时，名字节点和数据节点需要对这些信息，即 StorageInfo 包含的信息进行检查，以保证当前注册的节点是 HDFS 的一个合法数据节点，而不是一个属于其他集群的节点，或者曾经属于集群，但未进行必要升级的节点，以保证存储系统的一致性。

---

**注意** StorageInfo 并不在接口 DatanodeProtocol 所在的包中，而是定义在包 org.apache.hadoop.hdfs.server.common 中，它被数据节点、名字节点和第二名字节点等多个实体使用，是 HDFS 实现的一个重要概念，后续的代码分析还会涉及到 StorageInfo。

---

NamespaceInfo 继承自 StorageInfo，除了 StorageInfo 中的 layoutVersion、namespaceID 和 cTime 属性，它引入了 buildVersion 和 distributedUpgradeVersion。成员变量 buildVersion 保存了系统构建的（subversion）版本号，distributedUpgradeVersion 则用于数据节点升级前进行的版本检查。注意，NamespaceInfo 中包含的信息是整个 HDFS 集群的信息，和具体的数据节点没有关系。

（2）握手、注册、数据块上报和心跳

数据节点启动时，会和名字节点先进行握手，使用的是 DatanodeProtocol 接口的无参数方法 versionRequest()，其返回值为 NamespaceInfo 对象。注意，数据节点启动，可以是一般启动，也可以是为了升级而启动，如果是为了升级而启动，则不会有后续的数据节点注册。同时，握手时会检查名字节点和数据节点的 buildVersion，即构建版本号。HDFS 的版本检查是非常严格的，构建版本号不同会导致数据节点退出。

注册是数据节点投入正常工作前需要进行的必要步骤，数据节点通过远程方法 DatanodeProtocol.register() 向名字节点注册，其输入参数是一个 DatanodeRegistration 对象，如上面所述，该对象提供了进行注册的数据节点的节点标识 DatanodeID 和存储系统信息 StorageInfo，名字节点的返回值也是一个 DatanodeRegistration 对象，携带了对应的信息，供 DataNode 后续处理逻辑使用。

注册成功后，数据节点通过 DatanodeProtocol 中的 blockReport() 成员方法，上报它所管理的全部数据块信息，帮助名字节点建立数据块和数据节点的映射关系。blockReport() 需要两个参数，第一个参数是一个 DatanodeRegistration 对象，用于标识数据节点的身份（Hadoop IPC 不支持会话，数据节点调用名字节点的远程方法时，需要提供自己的身份）。

第二个参数是一个长整型数组，包含了数据块的信息。细心的读者可能会问，这里上报的为什么不是一个 Block 数组呢？我们知道，Block 对象有三个长整型成员变量，把它们放在一个长整型数组里，名字节点接收到信息时，就不需要创建大量的 Block 对象，而只需要使用一个工作对象，不断从数组里提取出数据块，即可以处理 blockReport() 上报的所有信息，减少了名字节点的负担。blockReport() 的返回值是一个 DatanodeCommand 对象，它带回了名字节点的指令。名字节点使用 DatanodeCommand 对象通知数据节点执行一些操作，如重新注册、发送心跳或者删除本地磁盘上的数据块等，在本小节的后面，会集中讨论 DatanodeCommand 以及它的子类。

由于一个数据节点上管理的数据块比较多，blockReport() 一般只在数据节点启动的时候使用。一般情况下，数据节点通过心跳向名字节点报告自己的状态，如果名字节点长时间接收不到数据节点的心跳，它会认为该数据节点已经失效。

sendHeartbeat() 除了携带标识身份的 DatanodeRegistration 对象外，还包括数据节点当前运行情况的信息。这些信息有数据节点的容量、已经使用的容量、剩余容量、正在写文件数据的连接数、读写数据使用的线程数等，这些数据基本概括了数据节点的负载情况。sendHeartbeat() 方法的返回值是一个 DatanodeCommand 数组，它带回来一系列的名字节点指令。

（3）名字节点指令

名字节点指令 DatanodeCommand 和它的子类如图 6-10 所示。

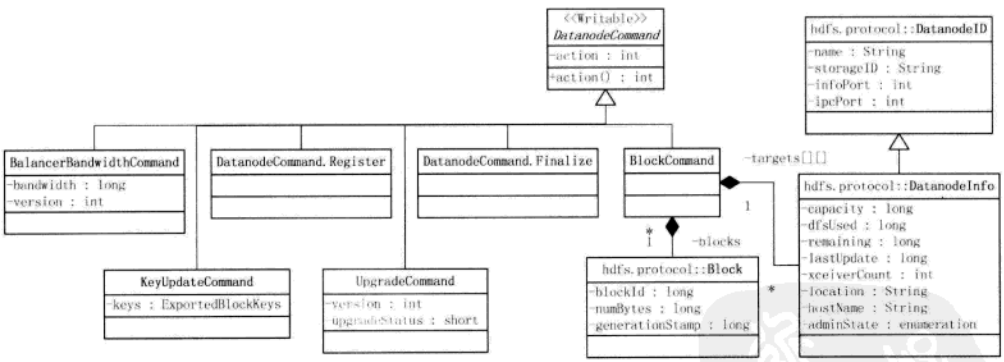



图 6-10　DatanodeCommand 和它的子类

blockReport() 和 sendHeartbeat() 的返回值都和 DatanodeCommand 有关，DatanodeCommand 是所有名字节点对数据节点指令的基类。它一共有 6 个子类，代表 6 类不同的操作，但名字节点到数据节点一共有 9 种命令，每个命令都会有一个编号，作为常量定义在 DatanodeProtocol 中[⊖]。代码如下：

⊖ 既然有 DatanodeCommand 的相关子类，为何又引入命令编号？这是 Hadoop 开发过程中遗留的设计不一致问题，也包含着 Hadoop 的一些历史痕迹。

```
public interface DatanodeProtocol extends VersionedProtocol {
    ......
    final static int DNA_UNKNOWN = 0;// 未定义
    final static intDNA_TRANSFER = 1;// 数据块复制
    final static intDNA_INVALIDATE = 2; // 数据块删除
    final static intDNA_SHUTDOWN = 3;// 关闭数据节点
    final static intDNA_REGISTER = 4;// 数据节点从新注册
    final static intDNA_FINALIZE = 5;// 提交上一次升级
    final static intDNA_RECOVERBLOCK = 6;// 数据块恢复
    final static intDNA_ACCESSKEYUPDATE = 7;// 安全相关
    final static intDNA_BALANCERBANDWIDTHUPDATE = 8;// 更新平衡器可用带宽
    ......
}
```

DatanodeCommand 中包含的内部类 Register（使用编号 DNA_REGISTER）和 Finalize（编号 DNA_FINALIZE），分别用于让数据节点再次发送注册信息和进行升级提交，它们都不需要携带附加信息。在实际运营的系统中，只有 blockReport() 可能会返回 DatanodeCommand.Finalize 对象。

和升级相关的另一条命令是 UpgradeCommand，它用于启动一次升级，其命令编号是 DNA_UNKNOWN。由于数据节点在处理这条命令时，不会处于一般的正常工作状态，所以，它通过子类，而不是命令编号来区分命令。

命令编号 DNA_SHUTDOWN 已经废弃不用，如果名字节点需要让数据节点停止服务，如当运维人员撤销某一个节点时，名字节点通过抛出异常 DisallowedDatanodeException 来通知数据节点，而不是通过某个命令编号为 DNA_SHUTDOWN 的 DatanodeCommand 子类来下发这个指令。DNA_ACCESSKEYUPDATE 以及 DatanodeCommand 的子类 KeyUpdateCommand 和安全相关，就不再进行讨论了。

DNA_BALANCERBANDWIDTHUPDATE 用于设置均衡器工作时可使用的带宽。均衡器用于平衡数据块在 HDFS 各个数据节点中的分布，避免系统出现文件数据分布不均匀，进而影响系统资源利用的情况。DNA_BALANCERBANDWIDTHUPDATE 对应的类 BalancerBandwidthCommand 可用于设置均衡器可使用的最大带宽，避免均衡器占用太多的数据节点带宽而影响系统工作。

命令编号 DNA_TRANSFER、DNA_INVALIDATE 和 DNA_RECOVERBLOCK 是和数据块相关的三条操作，它们都使用 BlockCommand 携带命令执行过程中需要的信息。

BlockCommand 有两个成员变量，分别是 blocks 和 targets。blocks 是数组，包含了该 BlockCommand 命令具体涉及的数据块，targets 是一个类型为 DatanodeInfo 的二维数组，用于数据块复制或数据块恢复中，提供了复制或恢复涉及的数据节点信息。

```
public class BlockCommand extends DatanodeCommand {
    Block blocks[];
    DatanodeInfo targets[][];
    ......
}
```

如命令编号的名字，DNA_TRANSFER 用于数据块复制，当系统中某数据块副本数小于设置值时，名字节点会通知某一个拥有该数据块的数据节点将数据块复制到其他数据节点。当数据块副本数大于副本系数时，DNA_INVALIDATE 可以删除多余的数据块，释放存储空间。名字节点还可以通知数据节点进行数据块恢复，当客户端在写数据过程中异常退出时，数据节点上同一数据块的不同副本数据可能处于不一致的状态，DNA_RECOVERBLOCK 指令选择某一个数据节点，称为主数据节点，协调其他数据节点，将这些副本恢复到一致的状态。HDFS 在数据块恢复过程中采取了将各个数据块恢复到它们中的最小长度的策略。数据块恢复配合租约恢复，是 HDFS 故障处理中的重要机制。

上面介绍的这些 DatanodeCommand，除了和升级相关的两个指令：DatanodeCommand.Finalize 和 UpgradeCommand，其他指令都由 IPC 调用 sendHeartbeat() 的返回结果携带。

（4）数据块操作相关方法 reportBadBlocks()

前面已经了解了 DatanodeProtocol 的 register()、sendHeartbeat()、blockReport() 和 versionRequest() 4 个方法，除了 sendHeartbeat()，其他 3 个方法在数据节点启动时就会使用，它们与数据节点上的数据读写没有直接关系。接下来介绍的则是和数据块密切相关，由数据块读、写或者恢复触发的 DatanodeProtocol 成员方法，它们包括 blocksBeingWrittenReport()、blockReceived()、reportBadBlocks()、nextGenerationStamp() 和 commitBlockSynchronization()。

上述这些方法中，只有 reportBadBlocks() 与数据块读相关。数据块保存在数据节点上，可能由于各种原因，导致数据块损坏。HDFS 使用我们介绍过的循环冗余校验（CRC-32），进行错误检测。HDFS 会在 3 种情况下检验校验和，分别是数据节点接收数据后，存储数据前；（某些情况下）客户端读取数据节点上的数据时；数据节点上的 DataBlockScanner 扫描线程定期扫描数据块。当发现数据块有问题，也就是校验出错时，就会通过 reportBadBlocks() 上报给名字节点。上报时，提供给名字节点的信息是一组 LocatedBlock 对象。LocatedBlock 提供了数据块及其位置信息，名字节点根据这些信息，移除发生错误的数据块，以保证整个系统数据的正确性。

数据节点使用 DatanodeProtocol.blockReceived() 向名字节点报告它已经完整地接收了一些数据块，接收的来源可以是客户端，即用户往 HDFS 上写文件时获得的数据块，也可以是其他的数据节点，由数据块复制命令 DNA_TRANSFER 或者平衡器 Balancer 触发的数据复制。在后续的讨论中，我们将该报告称为向名字节点提交数据块。远程方法 blockReceived() 的第一个参数标识了提交数据块的数据节点的 DatanodeRegistration 对象，接下来是一个 Block 数组，包含了已接收数据块的信息，blockReceived() 会一次上报多个已接收数据块的信息。最后一个参数 delHints，由平衡器使用，代码如下：

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

虽然 DatanodeProtocol 接口中有多个方法与数据块写相关，但在一个正常的流程中，数据节点只需要使用 blockReceived()，向名字节点确认数据块接收的情况，而不会涉及其他接

口方法，总的来说，blockReceived() 是一个比较“忙”的 DatanodeProtocol 方法。

方法 blocksBeingWrittenReport()、nextGenerationStamp() 和 commitBlock-Synchronization() 都和系统的异常处理相关。当系统支持 append，即往 HDFS 文件追加数据时，数据节点会在注册后，调用 DatanodeProtocol.blocksBeingWrittenReport()，向名字节点报告当前正处于写状态的数据块信息，以帮助名字节点进行租约恢复。

当名字节点进行租约恢复，通过 DNA_RECOVERBLOCK 指令下发通知，由数据节点配合进行数据块恢复，或者客户端调用 ClientDatanodeProtocol 接口上的 recoverBlock() 方法，处理写数据过程中发生的数据节点错误，启动数据块恢复时，都需要使用到 DatanodeProtocol 接口上的 nextGenerationStamp() 方法和 commitBlockSynchronization() 方法。

成员函数 nextGenerationStamp() 用于向名字节点申请一个新的数据块版本号，我们知道，数据块在系统里有唯一标识，即数据块的 ID 号，数据块恢复过程中，需要一个新的数据块版本号，用于区分同一数据块修复前后的差别。

数据块恢复完成后，无论结果是成功还是失败，都需要通过 commitBlockSynchronization() 告知数据块恢复的执行情况。该远程方法需要大量的参数，包括：进行恢复的数据块；通过 nextGenerationStamp() 申请的该数据块的新版本号；数据块恢复后的新长度；所属文件是否由名字节点关闭；是否删除名字节点上的数据块信息和成功参与数据块恢复的数据节点列表。由参数可知，执行数据块恢复后，数据节点需要通过这个方法更新名字节点上的一系列信息，它的声明如下：

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

commitBlockSynchronization() 等远程过程调用是名字节点和数据节点相互配合，进行故障恢复过程中涉及的重要方法。HDFS 故障恢复是我们后面会讨论的一个重要议题，在这里就不再深入研究相关的细节了。

（5）errorReport() 和 processUpgradeCommand()

DatanodeProtocol 接口中还没有分析的方法有 errorReport() 和 processUpgradeCommand()。其中，errorReport() 用于数据节点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、等待被复制的数据块不存在等，供开发人员调试或者运维人员了解数据节点上发生的特殊事件。这个方法再次体现了名字节点在 HDFS 中的中心地位。

如同方法名所示，processUpgradeCommand() 和系统升级有关。注意，这是数据节点到名字节点方向的远程过程调用，所以，它用于向名字节点报告升级的状态。

DatanodeProtocol 接口充分体现了 HDFS 体系中数据节点和名字节点的关系。名字节点是 HDFS 中的管理者，数据节点接受名字节点的管理，注册、心跳和数据块上报等接口方法，用于维护它们间的主从结构。另一方面，数据节点以数据块保存着 HDFS 上的所有文件数据，数据块的变化，包括客户端写数据引起的变动或者系统故障恢复过程中发生的修改，都需要及时和名字节点同步，DatanodeProtocol 接口中有接近一半的方法与此相关。

DatanodeProtocol 接口还有一个比较特殊的地方，它的部分方法会以返回值的方式，带来名字节点到数据节点的指令，这样的指令一共有 9 个，数据节点通过执行这些命令，配合名字节点，维护整个 HDFS 系统的正常工作状态。

### 2. InterDatanodeProtocol

InterDatanodeProtocol 有三个方法，如图 6-11 所示。

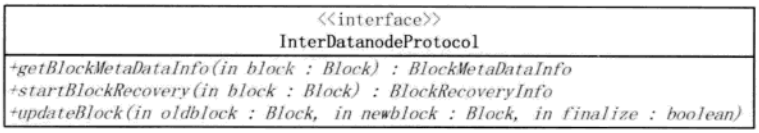



图 6-11　InterDatanodeProtocol 接口

其中，getBlockMetaDataInfo() 并没有被其他的 HDFS 代码使用，该方法可用于实现 HDFS 的维护工具。startBlockRecovery() 和 updateBlock() 用于数据块恢复。前面在讲 ClientDatanodeProtocol 接口的时候，就知道数据节点向客户端提供了错误恢复的远程方法：ClientDatanodeProtocol.recoverBlock()。它用于对客户端写数据过程中出现的故障进行错误恢复；名字节点到数据节点的指令 DNA_RECOVERBLOCK 也可发起一个数据块恢复。

为 IPC 建立连接是一个开销比较大的操作，在正常的数据节点写操作过程中，节点间不需要通过远程方法调用进行交互。但故障发生后，参与到这个写操作的数据节点中，有一个数据节点会被选择出来，协调其他数据节点，进行故障恢复，该过程需要使用到 InterDatanodeProtocol 中的这两个远程方法。

InterDatanodeProtocol.startBlockRecovery() 用于获得参与到数据块恢复过程的各个数据节点上的数据块信息，它的返回值是一个 BlockRecoveryInfo 对象（代码如下），包含了数据块 block 和标志位 wasRecoveredOnStartup。处于协调者地位的数据节点，根据 BlockRecoveryInfo 对象中的这些信息，计算出数据块恢复后数据块的新长度，和名字节点交换一些信息后，通过 updateBlock() 方法通知其他数据节点，将各个节点上的待恢复数据块更新到该长度，并更新数据块的版本号。

```
public class BlockRecoveryInfo implements Writable {
   private Block block;
   private boolean wasRecoveredOnStartup;
   ……
}
```

updateBlock() 携带了数据块恢复前后的信息，即 oldblock 和 newblock 对象，最后一个类型为布尔的参数 finalize，用于通知执行操作的数据节点在完成更新后，是否提交（finalize）数据块。

InterDatanodeProtocol 是一个非常简单的接口，只用于处理异常情况，后续的详细代码解读会再次考察这些远程方法的配合，以构建一个强壮的分布式文件系统。

### 3. NamenodeProtocol

NamenodeProtocol 是本节讨论的最后一个接口，如图 6-12 所示。名字节点实现了该远程接口，调用者有两个，一个是第二名字节点，另一个是 HDFS 工具：均衡器 balancer。

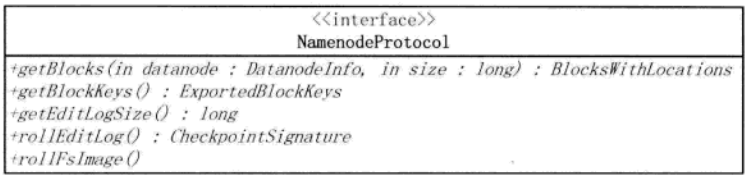



图 6-12 NamenodeProtocol 接口

均衡器使用了 NamenodeProtocol 的 getBlocks() 和 getBlockKeys() 方法，getBlocks() 可获得某一个数据节点（通过类型为 DatanodeInfo 的 datanode 参数）上的一系列数据块及位置，根据这些返回值，均衡器可以把数据块从该数据节点移动到其他数据节点，达到平衡各个数据节点数据块数量的目的；getBlockKeys() 方法用于支持这个过程中需要的安全特性。

NamenodeProtocol 中的另外三个方法相互配合，完成第二名字节点的功能：获取 HDFS 的命名空间镜像和镜像编辑日志，合并得到一个新的镜像，上传新命名空间镜像到名字节点，替换原有镜像并清空镜像编辑日志。

NamenodeProtocol.getEditLogSize() 可获得名字节点上编辑日志的大小，如果日志很小，第二名字节点就不需要合并镜像数据和编辑日志，即使名字节点发生故障也可以很快就恢复过来。第二名字节点通过 rollEditLog() 方法通知名字节点开始一次合并过程。这时，名字节点会停止使用当前的编辑日志，并启用新的日志文件，以方便第二名字节点通过基于 HTTP 的流式接口，获取待合并的命名空间镜像和镜像编辑日志。rollEditLog() 返回一个 CheckpointSignature 对象，该对象可以标识一次合并点，借用数据库中的概念，合并点也叫检查点。合并操作完成后，第二名字节点通过 HTTP 接口上传新的元数据镜像，操作完成后，无参的远程方法 rollFsImage() 被调用，名字节点进行一些必要的处理，最终完成一次元数据合并。

总的来说，NamenodeProtocol 接口是我们讨论的这些接口中，目标清晰而功能简单的一个接口。

## 6.3 非远程过程调用接口

对网络文件系统（NFS）比较熟悉的读者应该知道，SUN 公司是在远程过程调用（RPC）之上开发的 NFS，它的所有文件操作，包括文件 / 目录 API 和用于读写文件数据，都通过远程过程调用实现。客户使用本地操作系统提供的系统调用访问文件系统，当虚拟文件系统（VFS）发现系统调用需要访问 NFS 时，如在远程服务器上创建目录或对文件进行读操作，虚拟文件系统会将操作传递给 NFS 客户组件，由该组件通过 RPC 访问相应的 NFS 服务器。

从前面的分析，特别是对 ClientProtocol 的分析中，我们知道，HDFS 的文件和目录相关事务部分，遵循了 NFS 的实现思路，文件 / 目录 API 利用 Hadoop 远程过程调用，发送到名字节点上去执行。但对文件数据的读写，HDFS 采取了和网络文件系统截然不同的实现方式。

HDFS 没有采用远程过程调用实现对文件的读写，原因非常简单，HDFS 需要支撑超大文件，基于 Hadoop IPC 实现文件读写，效率达不到系统的设计要求。同时，HDFS 提供了对数据的流式访问，使用基于 TCP 的流式数据访问接口，有利于批量处理数据，提高数据的吞吐量。

在介绍 NamenodeProtocol 接口时，我们知道第二名字节点会根据一定的策略合并名字节点上的命名空间镜像和镜像编辑日志，以减少名字节点从故障中恢复的时间，但 NamenodeProtocol 并没有获取原始镜像数据和编辑日志的远程方法，也没有上传合并后新命名空间镜像的远程方法，上述两个过程，是通过名字节点提供的，基于 HTTP 的流式接口进行的。第二名字节点利用名字节点的内建 HTTP 服务器，使用 HTTP 的 GET 操作获取数据，即原始命名空间镜像和编辑日志，合并操作完成后，通知名字节点（利用 HTTP 协议），由名字节点使用 HTTP GET，从第二名字节点下载新命名空间镜像。名字节点和第二名字节点间的这个接口也是非 IPC 接口，是本节要介绍的另一个流式接口。

### 6.3.1　数据节点上的非 IPC 接口

数据节点提供对 HDFS 文件内容，也就是数据块的读写功能：将 HDFS 文件数据写到 Linux 本地文件系统的文件中，或者从这些本地文件中读取 HDFS 文件数据。读写的对外接口是基于 TCP 的非 IPC 接口。除了数据块读写，数据节点还提供数据块替换、数据块拷贝和数据块检验信息读等基于 TCP 的接口。也就是说，数据节点通过流式接口，一共提供了 5 种操作，这些操作都有相应的请求帧结构和操作码。

数据节点的流式接口操作码定义在 DataTransferProtocol 接口中，由于这个接口会被客户端使用，它定义在 org.apache.hadoop.hdfs.protocol 包中。根据 DataTransferProtocol 的代码可以知道，写数据块时会使用操作码 80，代码如下：

```
public interface DataTransferProtocol {
  //接口版本号，目前是第 18 个版本
  public static final int DATA_TRANSFER_VERSION = 17;

  public static final byte OP_WRITE_BLOCK = (byte) 80;
  public static final byte OP_READ_BLOCK = (byte) 81;
  /**
   * @deprecated OP_READ_METADATA 已经不再支持
   */
  @Deprecated public static final byte OP_READ_METADATA = (byte) 82;
  public static final byte OP_REPLACE_BLOCK = (byte) 83;
  public static final byte OP_COPY_BLOCK = (byte) 84;
  public static final byte OP_BLOCK_CHECKSUM = (byte) 85;
  ……
```

```
    public static final int OP_STATUS_SUCCESS = 0;
    public static final int OP_STATUS_ERROR = 1;
    public static final int OP_STATUS_ERROR_CHECKSUM = 2;
    public static final int OP_STATUS_ERROR_INVALID = 3;
    public static final int OP_STATUS_ERROR_EXISTS = 4;
    public static final int OP_STATUS_ERROR_ACCESS_TOKEN = 5;
    public static final int OP_STATUS_CHECKSUM_OK = 6;
    ……
}
```

和 Hadoop IPC 中的版本检查机制类似，数据节点上的这些接口也有一个共同的版本号 DATA_TRANSFER_VERSION，当前值是 17。同时，DataTransferProtocol 中还定义了多个操作的应答码，它们都是以 OP_STATUS 开始的常量。一个操作成功的应答码是 OP_STATUS_SUCCESS，又如在读操作中，如果启用 Hadoop 安全机制，同时用户又没有权限读取目标数据块，数据节点会给客户端发送 OP_STATUS_ERROR_ACCESS_TOKEN 应答码。应答码往往是应答的第一个字段。

### 1. 读数据

读数据就是从数据节点的某个数据块中读取一段文件数据，它的操作码是 81。当客户端需要读数据时，它通过和数据节点的 TCP 连接，发送请求，由于 TCP 是基于字节流的，没有消息边界的概念，所以，必须在流上定义一个数据帧并通过该数据帧交互信息。图 6-13 是客户端读数据时发送的请求。

图 6-13 数据节点读请求

请求的最前面两个字段，分别是接口版本号和操作码，版本号确保通信双方对数据帧的理解是一致的，紧接着的一个字节操作码表明操作的目的，这两个域会出现在所有的数据节点流接口的请求中。读数据的操作码是 81。

请求接下来是数据块 ID 和数据块版本号，根据这两个参数，数据节点可以确定操作的目标数据块，如同普通的文件读操作需要指定请求数据的开始位置和需要的数据长度，读数据块也需要提供偏移量和数据长度。通过上述 4 个参数，发起读请求的客户端明确了这次请求希望获得的数据。

客户端名字是一个字符串，只用于日志输出中，访问令牌则是执行 Hadoop 安全检查的需要，后续不再对这两个字段进行研究了。

读请求的响应也有一定的帧结构，首先是应答头，接下来是一系列的数据包。

应答头的第一个域是操作的返回码，OP_STATUS_SUCCESS（值为 0）代表了一次成

功的操作，应答头的其他部分和数据校验有关。我们知道，为了保证数据的完整性，HDFS 为每个数据块保持了相应的校验信息，校验基于块，块大小默认值是 512 字节，即从数据块头开始，每 512 字节就会产生一个 4 字节的校验和。和 ChecksumFileSystem 中的实现一致，读 / 写数据块时，也需要维护基于流的数据读写和基于块的校验和的关系，所以，在读请求的应答头中，需要告知请求方校验信息的配置信息。这些信息包括：数据检验的方式（如 CRC-32 检验），检验方式的编码定义在 org.apache.hadoop.util.DataChecksum 中，目前支持 CRC-32 校验和空校验（就是不进行数据校验）两种方式，CRC-32 的编码是 CHECKSUM_CRC32，值是 1；接下来的字段给出了每个校验块的大小，一般来说 CRC-32 的校验块大小是 512 字节；应答头的最后是一个可选的偏移量，该偏移量给出了应答数据在数据块中的实际开始位置。考虑到校验和计算，数据节点在某些条件下，会给客户端多发送一些数据。假设要读某个数据块第 60 字节开始的 100 个字节，并假设该数据块的校验块大小是 512 字节，显然，上述数据位于这个数据块的第一个校验块中。这个时候，即使只需要读 100 字节的数据，但数据节点还是需要将前面的 60 字节数据发送给客户端，以保证客户端能够对数据进行检验。在上述条件下，读请求的偏移量，即 startOffset 的值为 60，但应答头的这个偏移量（offset）的值为 0。数据节点读应答的应答头如图 6-14 所示。

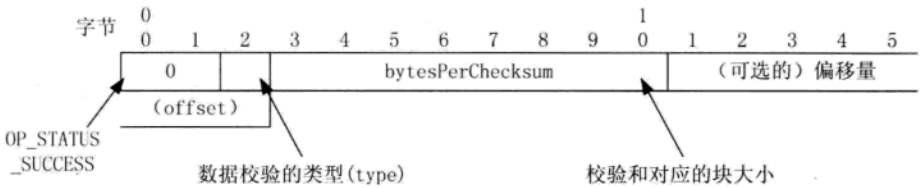



图 6-14　数据节点读应答的应答头

紧接着读操作的应答头是一个或多个数据包，数据包包含了实际的文件数据。值得一提的是，数据包不但用于数据块读，也用于数据块写等与数据块内容操作相关的地方，如图 6-15 所示。

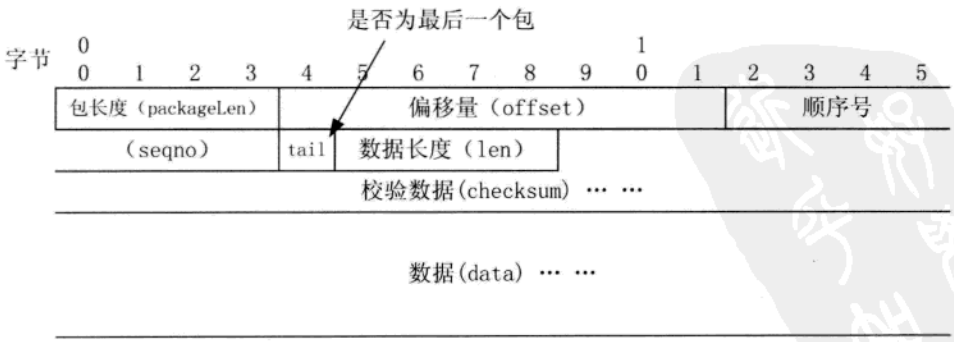



图 6-15　数据节点读应答的数据包

很明显，在读取大量数据的情况下，一个读操作将会产生多个数据应答包，应答包的第

一个字段是包长度，请求者可以通过这个字段，在 TCP 流上进行数据分帧，获得一个完整的数据应答包。偏移量则是这个数据包的起始位置，其实它和应答头的偏移量字段意义是一样的，即数据在数据块中的偏移量，应答头的偏移量应该等于第一个数据包的偏移量。以上面读数据的假设为例，第一个（其实也只有一个）应答数据包中，偏移量字段的值为 0。接下来的 tail 是一个标志位，如果这个包是读操作应答的最后一个包，则 tail 的值是 1，否则是 0。前面提过写数据时，数据也是通过相同格式的结构，发往数据节点，这个时候，tail 表示客户端已经发送完所有的数据。最后一个字段是应答包中包含的数据的长度。

数据应答包中包含校验数据和文件数据，根据数据包中的最后一个字段：数据长度字段和应答头中的校验信息（校验块大小），请求方可以计算出校验数据的大小，并进一步区分校验数据和文件数据。

从数据节点上读取数据时，使用的是基于 TCP 上的流接口。该接口包含三种数据帧，分别是读请求帧、读应答的应答头和数据包。读请求帧包含了在数据节点上定位一段数据所需要的信息，即数据块 ID、数据块版本号、数据开始位置和所需数据的长度，以及执行读操作时需要的一些附加信息。读请求的应答包含一个应答头和一个或多个数据应答包，如同 ChecksumFileSystem 为引入数据完整性而带来的一系列问题，读请求的应答也需要处理数据及其校验信息的对应关系。

### 2. 写数据

写数据操作的复杂程度远超读数据操作，该操作用于往数据节点上的某一数据块上追加数据，其操作码是 80。在介绍写数据前，先来考察 HDFS 写数据的数据流管道。

数据流管道在 Google 实现它们的分布式文件系统（GFS）时就已经引入，其目的是：在写一份数据的多个副本时，可以充分利用集群中每一台机器的带宽，避免网络瓶颈和高延时的连接，最小化推送所有数据的延迟。Hadoop 文件系统也实现了数据流管道，如图 6-16 所示。

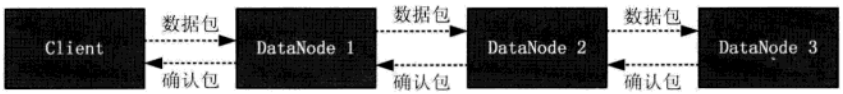



图 6-16 数据流管道

假设目前客户端写数据的文件副本数是 3，也就是说，在该 HDFS 集群上，一共有三个数据节点会保存这份数据的三个副本，客户端在发送数据时，不是同时往三个数据节点上写数据，而是将数据发送往第一个数据节点，然后，第一个数据节点在本地保存数据，同时推送数据到数据节点 2，随后照这样进行，直到管道中的最后一个数据节点。确认包由最后一个数据节点产生，并逆流往客户端方向回送，沿途的数据节点在确认本地写成功后，才往上游传递应答。相对于客户端往多个不同的数据节点同时写数据的方式，处于数据流管道上的每个节点都承担了写数据过程中的部分网络流量，降低了客户端发送多份数据对网络的冲击。

当客户端发送写请求时，它如何支持数据流管道呢？

和读请求类似，写请求的最前面两个字段，分别是接口版本号和操作码。写数据时也需

要指定数据块，所以，请求接下来的两个字段和读请求一致，分别是数据块标识和版本号。但和读请求不同的是，写数据请求没有偏移量字段，也就是说，用户只能往数据块后面添加数据，不能修改已经写入的文件内容。字段 pipeline 指定了整个数据流管道的大小，以图 6-16 的管道为例，该值为 3。标志位 isRecovery 指示这个写操作是不是错误恢复过程中的一部分。如果数据源是某个客户端，则接下来请求会携带客户端的名字，如果数据源是某个数据节点，则客户端字段为空，同时标志位 hasSrcDataNode 置位，在这种情况下，请求中紧接着是源数据节点的信息字段 srcDataNode，该信息就是源数据节点的 DatanodeInfo 对象的序列化结果。写请求数据源是某一个数据节点，表明该数据节点在执行数据复制。注意，如果是客户端发起的请求，hasSrcDataNode 标志位的值为 false，这时，请求中就不会包含可选字段 srcDataNode 了。

数据节点写请求如图 6-17 所示。

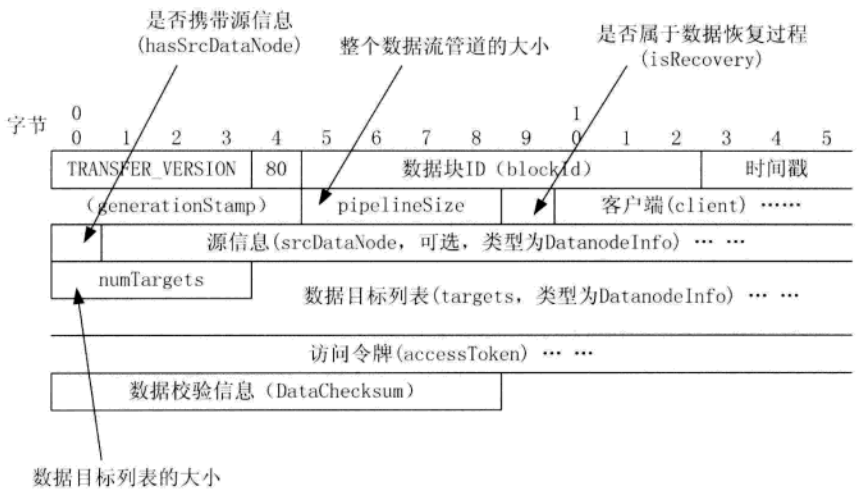



图 6-17　数据节点写请求

为了实现数据流管道功能，写请求包括 numTargets 和 targets 字段，其中 targets 是数据目标列表，numTargets 是这个列表的大小。如果 numTargets 的值为零，表明当前数据节点是数据流管道中的最后一个节点；如果 numTargets 值大于 0，那么，数据目标列表中的第一项，就是数据流管道中，位于当前数据节点后面的下游数据推送目标。还是以前面的管道为例，客户端和数据节点 1、数据节点 2 都会向它们的下游节点发送写请求，客户端发送的请求中，numTargets 值为 2，数据目标列表包含了数据节点 2 和数据节点 3。当数据节点 1 为了建立数据管道而发送写请求时，它知道它的后续节点是数据节点 2，同时它会要求数据节点 2 建立到数据节点 3 的通道，这时，数据节点 1 发送的写请求中，numTargets 值为 1。数据节点 3 由于位于管道的末端，它会接收到一个 numTargets 值为 0 的请求。就这样，根据 numTargets 和 targets 中的信息，连接数据流管道上各个节点的 TCP 连接被建立起来，为后

续写数据准备好通道。

写请求的访问令牌用于 HDFS 的安全特性中，我们就不再深入研究了。数据校验信息是请求的最后一个字段，这个字段也出现在数据节点读应答的应答头中（参见图 6-15），并有着相同的功能，用于进行数据校验能力协商。DataChecksum 序列化后一共 9 个字节，包含了两个字段，即数据检验的方式（前面已经提过，CRC-32 的编码是 1）和校验块的大小，根据 DataChecksum 中的这些信息，数据节点为读取并校验客户端发送过来的数据做好准备。

写请求有相应的应答，正常情况下，这个应答会从管道的最后一个数据节点开始，往客户端方向发送，管道上的每一个节点都会等待这个应答，收到应答后，才会开始接收数据，也就是说，客户端会等待这个应答，然后才开始发送数据。应答包结构如下图，只有两个字段：返回码和附加信息，当返回码是 OP_STATUS_ERROR 时，附加信息提供了流中第一个出错的数据节点地址信息。数据节点写请求应答如图 6-18 所示。

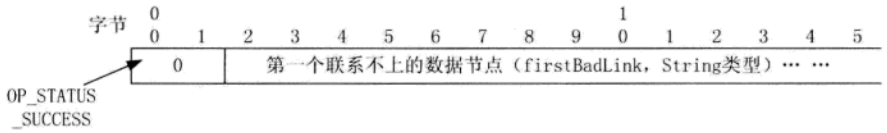



图 6-18 数据节点写请求应答

写请求发送后，请求方会接着发送一个或多个写数据包，发送数据块的具体内容。写数据包采用了和数据节点读应答中的数据包一样的格式，具体字段及字段含义，请参考前文。

客户端通过数据流管道发送数据，管道上的数据节点在接收数据并写磁盘后，需要给上游节点发送确认包，以清除缓冲区中的内容。确认包从最后一个数据节点发出，逆流而上，直达数据源。在写数据的过程中，一般会产生多个确认包。确认包是 DataTransferProtocol.PipelineAck 对象的序列化结果。如图 6-19 所示。

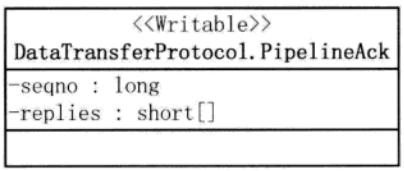



图 6-19 确认包 DataTransferProtocol.PipelineAck

PipelineAck 类包含两个字段，其中 seqno 是该确认对应的写请求数据包，也就是数据包中的顺序号。另外一个字段 replies 是一个数组，它包含了管道上的各个节点对写请求数据包的处理结果，如果数据节点顺利地将数据写入磁盘，则结果为 DataTransferProtocol.OP_STATUS_SUCCESS，否则为 DataTransferProtocol.OP_STATUS_ERROR。随着确认包逆流而上，字段 replies 的大小会不断增长。

数据流管道是 HDFS 实现中针对海量数据写操作的一个优化，在进行实际的写操作前，位于管道上的节点，根据上游节点发送过来的写请求，建立管道，并应用于后续的具体数据写操作中。写数据时，数据流管道上传输的写数据数据包，它复用了读操作的数据格式。每个数据包都有对应的应答包，以保证数据成功到达各个节点，同时，数据源也能根据应答包，了解各个节点上的操作结果。

### 3. 其他接口

读写操作接口是 HDFS 数据节点上最重要的流式接口，除了这两个接口，还有数据块替换、数据块拷贝和读数据块检验信息等基于 TCP 流的接口。

均衡器使用数据块替换和数据块拷贝接口，在数据节点间移动数据块，以平衡数据节点的负载，如图 6-20 所示。

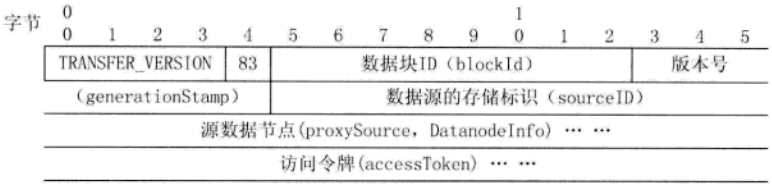



图 6-20　数据块替换请求

数据块替换用于将某个数据节点上的数据块移动到另外一个数据节点，移动，意味着源数据节点上的数据块在成功复制到目标数据节点后，会被删除，以维护一致的副本系数。该请求由均衡器发往目标数据节点，操作码是 83，格式如上图。替换的数据块由数据块标识和数据块版本号一起指定。数据源的存储标识是一个字符串，在数据块替换操作成功以后，执行操作的节点使用这个标识，通过 DatanodeProtocol.blockReceived() 通知名字节点，删除名字节点上源数据节点拥有的，被替换数据块的相应记录，并进一步通过名字节点指令，删除源数据节点上的数据块。接下来的请求字段是源数据节点，其类型是 DatanodeInfo，它包含了源数据节点的信息。

数据节点执行数据块替换请求的过程中，将源数据节点上的数据复制到目标数据节点，是由数据块拷贝完成的。和数据块替换请求略有不同，从结构上看，数据块拷贝的请求非常简单。在接口版本号和操作码后，就只有用于定位数据块的数据块 ID 和数据块版本号，以及用于安全机制的访问令牌等三个字段。数据块拷贝，本质就是一个读操作，由于拷贝针对整个数据块，所以也就不需要前面读请求请求帧的一些字段。同时，数据块拷贝的应答，采用了和读操作应答一样的流程，即应答包含一个应答头和一个或多个数据应答包，如图 6-21 所示。

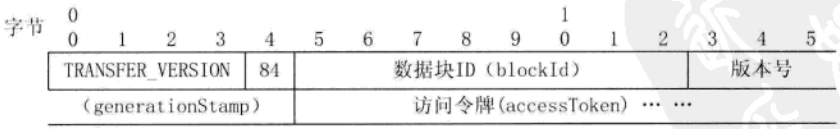



图 6-21　数据块拷贝请求

最后要介绍的数据节点非 IPC 接口用于读数据块检验信息，它的操作码是 85。该接口用于读取整个数据块校验信息的摘要，如图 6-22 所示。

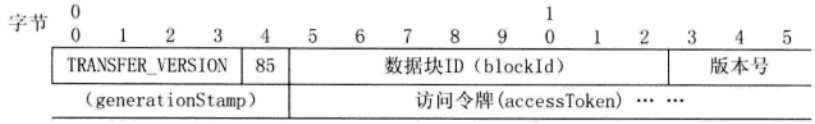



图 6-22 读数据块检验信息请求

**注意** 85 操作不是读取数据块校验文件的内容，它只是获取整个校验文件的 MD5 摘要。MD5（Message Digest Algorithm MD5，消息摘要算法第 5 版）是广泛应用于计算机安全领域的一种哈希函数，用以提供消息的完整性保护。例如，服务器预先提供一个文件的 MD5 校验和，用户下载完文件以后，用 MD5 算法计算下载文件的 MD5 校验和，然后通过检查这两个校验和是否一致，就能判断下载的文件是否完整。

读数据块检验信息的请求，除了操作码和数据块拷贝不一样以外，其他字段一致。该请求的应答如图 6-23 所示，包括三个字段，字段的含义非常容易理解，其中，校验文件的 MD5 摘要一共是 16 字节。

图 6-23 检验信息应答

细心的读者会发现，和其他基于流的接口不太一样，读数据块检验信息的请求信息和应答信息都很小，似乎用 Hadoop 远程过程调用实现更为合适。为什么不使用 IPC 来提供 85 接口呢？原因主要有：读检验信息和其他流接口一样，都是从数据节点上读取数据，它们有着类似的处理过程；另外，数据节点提供给客户端的 ClientDatanodeProtocol 远程接口，只在比较特殊的情况下，即写数据出现问题需要进行恢复时，或者客户端读取处于一个主机的数据节点上的数据块时，才会被使用，为了获取检验信息查询而建立 / 拆除 IPC 连接，性能上反而不如流式接口。

有了读写操作接口的基础，读者很容易就可以理解数据块替换、数据块拷贝和读数据块检验信息等操作的请求和应答。这些接口用于各种不同的 HDFS 应用中，是系统健康、正确运行的重要保证。

## 6.3.2 名字节点和第二名字节点上的非 IPC 接口

名字节点和第二名字节点上并没有保存 HDFS 的文件内容，为什么它们也需要非 IPC 接口呢？它们的非 IPC 接口的作用是什么？

我们知道，名字节点上保存着整个文件系统的文件目录树，以及文件的数据块索引。为

了保证名字节点重启后不会丢失数据，这些信息会持久化到磁盘上，并保存在一个名字为 fsimage 的文件中，对目录树的修改，不可能每次都全量更新 fsimage 文件，而是保存在编辑日志 edits 里面。通过 fsimage 和 edits，名字节点重启后就可以恢复整个目录树及文件的数据块索引的信息。随着系统的工作，编辑日志 edits 会一直增长，如果名字节点重启，将需要花很长的时间来运行编辑日志中的每个操作。为了合并 fsimage 和 edits 以创建一个新的 fsimage，HDFS 引入了第二名字节点。第二名字节点不接收也不记录 HDFS 的任何实时变化，而只是根据集群配置，不停地获取 HDFS 某一个时间点的 fsimage 和 edits，合并得到一个新的命名空间镜像。名字节点会获取该新映像，并替换原有的 fsimage，清空上述 edits 文件。上述过程中，第二名字节点需要从名字节点上获取原始的文件系统镜像和编辑日志，同时，合并后的文件系统镜像，需要由名字节点从第二名字节点上取回。由于 fsimage 和 edits 比较大，上述交互过程并没有采用 Hadoop IPC，同时，也没有使用数据节点的基于 TCP 的机制，而是使用了基于 HTTP 的流接口。

超文本传送协议（HTTP）是一个基于 TCP 的应用层协议。它是万维网协会（World Wide Web Consortium，W3C）和 Internet 工作小组（Internet Engineering Task Force，IETF）合作的结果，由一系列的 RFC（Request For Comments）定义，其中最著名的就是 RFC 2616，该 RFC 定义了目前被广泛使用的 HTTP 1.1 协议。HTTP 采用了典型的客户端 / 服务器模式，简单、灵活、无状态并可以携带元信息，这些特点使得 HTTP 被广泛应用于互联网中，是 WWW 浏览、面向服务的体系结构（Service-Oriented Architecture，SOA）的基础。HDFS 1.0 中的 WebHDFS，也充分利用 HTTP，在 HTTP 协议上提供了一个完整的、支持文件数据读写的访问 HDFS 机制。

当 HTTP 客户向服务器请求服务时，需要传送请求的方法和路径，同时协议允许通过 HTTP 头携带附加的信息（元信息）和数据。常用的 HTTP 请求方法有 GET、HEAD、PUT、POST、DELETE 等，名字节点和第二名字节点各自为对方提供了 GET 方法。在 HTTP 协议中，GET 用于取回指定路径上的资源，是最常用的方法。HTTP 的请求路径，使用了前面已经介绍过的统一资源定位符 URL。HTTP 应答包括应答头、数据和数据的元信息（也是通过 HTTP 头携带）。

第二名字节点获取命名空间镜像和编辑日志时，使用了 HTTP 的 GET 方法，其中，命名空间镜像的 HTTP 地址是：

```
http://< 名字节点地址 >:< 名字节点 HTTP 端口 >/getimage?getimage=1
```

编辑日志的 HTTP 地址是：

```
http://< 名字节点地址 >:< 名字节点 HTTP 端口 >/getimage?getedit=1
```

合并后的新命名空间镜像，名字节点通过如下地址获取：

```
http://< 第二名字节点地址 >:< 第二名字节点 HTTP 端口 >/getimage?getimage=1
```

合并命名空间镜像和编辑日志，生成新的命名空间镜像需要一段时间，名字节点什么时候可以通过 HTTP 协议来下载新的 fsimage 呢？答案是第二名字节点在数据准备好的时候，

通过发送一个 HTTP 请求，通知名字节点。该请求的一般形式如下：

```
http://< 名字节点地址 >:< 名字节点 HTTP 端口 >/getimage?putimage=1&port=< 第二名字节点 HTTP
端口 >&machine=< 第二名字节点地址 >&token=<CheckpointSignature 信息 >
```

注意，这个请求只是一个通知，不会传输大量的数据，但 Hadoop 的开发者并没有使用 IPC 机制，原因是名字节点在处理这个请求的过程中，会立即通过 HTTP 请求，获取第二名字节点上的新命名空间镜像，并在下载结束后才响应 HTTP 请求。通过前面的 IPC 代码分析，我们知道，远程过程调用适用于短而快的请求，不合适上述这种执行时间比较长的操作。

由于 Java API 已经提供了对 HTTP 协议的封装，使得名字节点、第二名字节点上下载数据的过程显得非常简单。节点上的 HTTP 客户端使用了协议处理器，处理模式为 HTTP 的 URL，最后获取输入流，通过“流”这个大家熟悉的机制获取数据。代码如下：

```
……
URL url = new URL(str.toString());
……
URLConnection connection = url.openConnection();
InputStream stream = connection.getInputStream();
……
```

Hadoop 的各个节点都内嵌了 Jetty，这是一个轻量级、基于标准、开源、并且具有丰富功能的 Web 容器，它实现了 Java 的 HTTP Servlet 规范。HTTP Servlet 是 Java Web 技术的核心基础，它简化了 Web 应用的开发和部署，开发人员只需要关注具体的业务逻辑，而把协议处理等细节交给 HTTP Servlet 的运行环境：Web 容器。名字节点和第二名字节点提供的数据下载功能，就是通过运行在 Jetty 的 HTTP Servlet 实现的。该 Servlet 通过上述 HTTP 地址中的参数，如“getimage=1”等，将对应的文件数据通过 HTTP 协议，发送到 HTTP 客户端。

名字节点和第二名字节点的分工，决定了它们之间涉及了命名空间镜像和镜像编辑日志的数据交换，这些数据交换和数据块读写一样，不适用 Hadoop 的远程过程调用机制。和前面介绍的数据节点上的非 IPC 接口不同，名字节点和第二名字节点的命名空间镜像和编辑日志传输，不会出现像数据流管道那样复杂的通信场景，所以，它们间的非 IPC 接口并没有基于 TCP 设计新的协议，而是采用了成熟的 HTTP 协议，并充分利用节点内置的 Jetty 服务器。

## 6.4 HDFS 主要流程

本节介绍 5 个典型的 HDFS 流程：客户端到名字节点的元数据操作、客户端读文件、客户端写文件、数据节点到名字节点的注册和心跳，以及第二名字节点合并元数据。这些流程充分体现了 HDFS 实体间 IPC 接口和流式接口的配合。

### 6.4.1 客户端到名字节点的文件与目录操作

客户端有到名字节点的大量元数据操作，如更改文件名（rename）、在给定目录下创建一个子目录（mkdir）等，这些操作一般只涉及客户端和名字节点的交互，通过远程接口

ClientProtocol 进行。

以创建子目录为例，如图 6-24 所示。

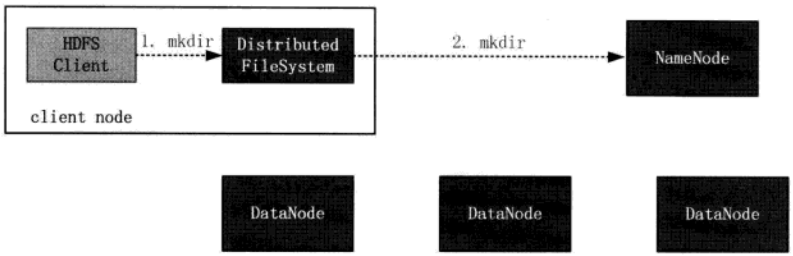



图 6-24　客户端到名字节点的创建目录操作

当客户端调用 HDFS 的 FileSystem 实例，也就是 DistributedFileSystem 的 mkdir() 方法时（图 6-24 步骤 1），DistributedFileSystem 对象通过 IPC 调用名字节点上的远程方法 mkdir()，让名字节点执行具体的创建子目录操作：在目录树数据结构上的对应位置创建新的目录节点，同时记录这个操作并持久化到日志中，方法执行成功后，mkdir() 返回 true，结束创建过程。期间，客户端和名字节点都不需要和数据节点交互。

一些更为复杂的操作，如使用 DistributedFileSystem.setReplication() 增加文件的副本数，再如通过 DistributedFileSystem.delete() 删除 HDFS 上的文件，都需要数据节点配合执行一些动作。

以客户端删除 HDFS 文件为例（图 6-25），操作在名字节点上执行完毕后，数据节点上存放文件内容的数据块也必须删除。但是，名字节点执行 delete() 方法时，它只标记操作涉及的需要被删除的数据块（当然，也会记录 delete 操作并持久化到日志），而不会主动联系保存这些数据块的数据节点，立即删除数据。当保存着这些数据块的数据节点向名字节点发送心跳时，即图 6-25 步骤 3，在心跳的应答里，名字节点会通过 DatanodeCommand 命令数据节点删除数据。在这个过程中，读者需要注意两个要点：被删除文件的数据，也就是该文件对应的数据块，在删除操作完成后的一段时间以后，才会被真正删除；名字节点和数据节点间永远维持着简单的主从关系，名字节点不会向数据节点发起任何 IPC 调用，数据节点需要配合名字节点执行的操作，都是通过数据节点心跳应答中携带的 DatanodeCommand 返回。

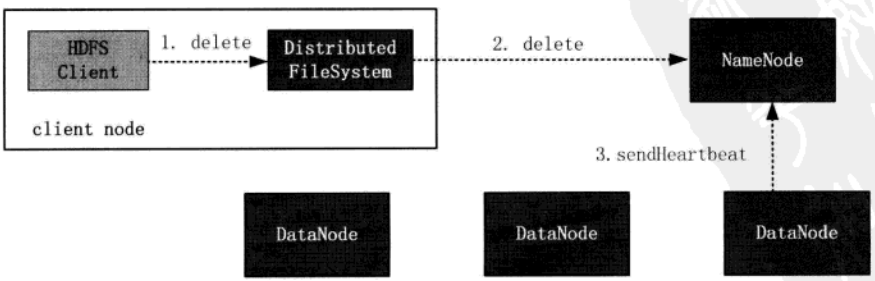



图 6-25　客户端到名字节点的删除文件操作

### 6.4.2 客户端读文件

图 6-26 显示了在读取 HDFS 上的文件时，客户端、名字节点和数据节点间发生的一些事件以及事件的顺序。

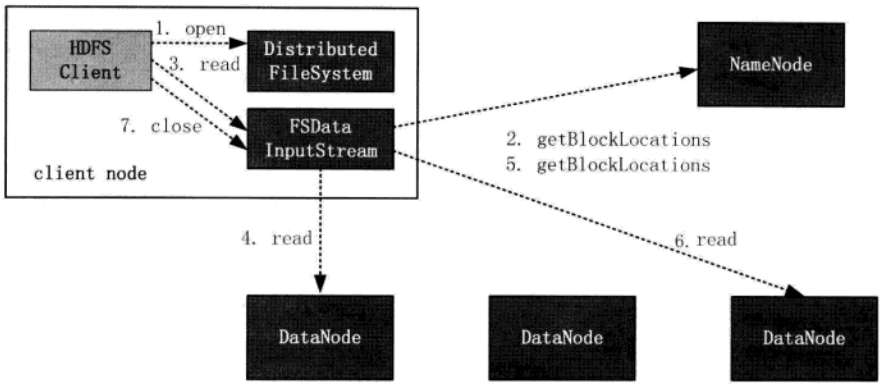



图 6-26 客户端读文件

客户端通过 FileSystem.open() 打开文件，对应的 HDFS 具体文件系统，DistributedFileSystem 创建输出流 FSDataInputStream，返回给客户端，客户端使用这个输入流读取数据。如第 5 章讨论的，FSDataInputStream 需要和具体的输入流结合，一起形成过滤器流（filtered stream）向外提供服务。

对 HDFS 来说，具体的输入流是 DFSInputStream。在 DFSInputStream 的构造函数中，输出流实例通过 ClientProtocol.getBlockLocations() 远程接口调用名字节点，以确定文件开始部分数据块的保存位置，即图 6-26 中的步骤 2。对于文件中的每个块，名字节点返回保存着该块副本的数据节点地址。注意，这些数据节点根据它们与客户端的距离（利用了网络的拓扑信息），进行了简单的排序。

客户端调用 FSDataInputStream.read() 方法读取文件数据时，DFSInputStream 对象会通过和数据节点间的“读数据”流接口，和最近的数据节点建立联系。客户端反复调用 read() 方法，数据会通过数据节点和客户端连接上的数据包返回客户端。当到达块的末端时，DFSInputStream 会关闭和数据节点间的连接，并通过 getBlockLocations() 远程方法获得保存着下一个数据块的数据节点信息（严格说，在对象没有缓存该数据块的位置时，才会使用这个远程方法），即图 6-26 中的步骤 5，然后继续寻找最佳数据节点，再次通过数据节点的读数据接口，获得数据（图 6-26 中的步骤 6）。

另外，由于 ClientProtocol.getBlockLocations() 不会一次返回文件的所有数据块信息，DFSInputStream 可能需要多次使用该远程方法，检索下一组数据块的位置信息。对于客户端来说，它读取的是一个连续的数据流，上面分析的联系不同数据节点、定位一组数据块位置的过程，对它来说都是透明的。当客户端完成数据读取任务后，通过 FSDataInputStream.close() 关闭输入流（图 6-26 中的步骤 6）。

在客户端读取文件时，如果数据节点发生了错误，如节点停机或者网络出现故障，那么客户端会尝试下一个数据块位置。同时，它也会记住出现故障的那个数据节点，不会再进行徒劳无益的尝试。读数据的应答包中，不但包含了数据，还包含了数据的校验和，客户端会检查数据的一致性，如果发现有校验错误，也就是说数据块已经损坏，它会将这个信息报告给名字节点，同时，尝试从别的数据节点中读取另外一个副本的文件内容。由客户端在读数据时进行数据完整性检查，可以降低数据节点的负载，均衡各节点的计算能力。

由客户端直接联系名字节点，检索数据存放位置，并由名字节点安排数据节点读取顺序，这样的设计还有一个好处是，能够将读取文件引起的数据传输，分散到集群的各个数据节点，HDFS 可以支持大量的并发客户端。同时，名字节点只处理数据块定位请求，不提供数据，否则，随着客户端数量的增长，名字节点会迅速成为系统的“瓶颈”。

### 6.4.3　客户端写文件

即使不考虑数据节点出错后的故障处理，文件写入也是 HDFS 中最复杂的流程。本节以创建一个新文件并向文件中写入数据，然后关闭文件为例，分析客户端写文件时系统各节点的配合，如图 6-27 所示。

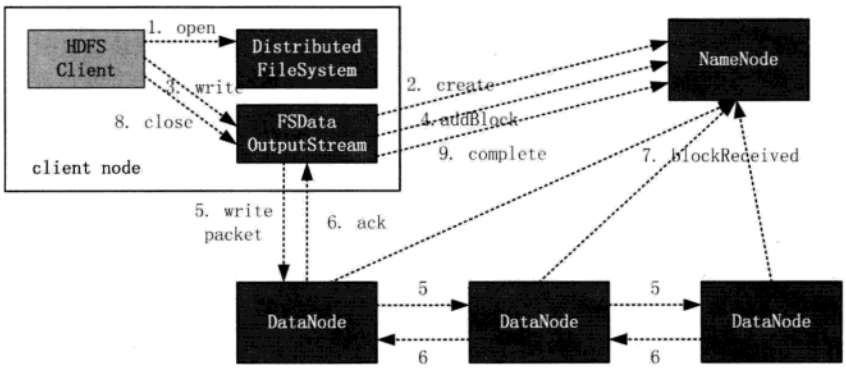



图 6-27　客户端写文件

客户端调用 DistributedFileSystem 的 create() 方法创建文件（图 6-27 中的步骤 1），这时，DistributedFileSystem 创建 DFSOutputStream，并由远程过程调用，让名字节点执行同名方法，在文件系统的命名空间中创建一个新文件。名字节点创建新文件时，需要执行各种各样的检查，如名字节点处于正常工作状态，被创建的文件不存在，客户端有在父目录中创建文件的权限等。这些检查都通过以后，名字节点会构造一个新文件，并记录创建操作到编辑日志 edits 中。远程方法调用结束后，DistributedFileSystem 将该 DFSOutputStream 对象包裹在 FSDataOutputStream 实例中，返回给客户端。

在图 6-27 中的步骤 3 客户端写入数据时，由于 create() 调用创建了一个空文件，所以，DFSOutputStream 实例首先需要向名字节点申请数据块，addBlock() 方法成功执行后，返回一个 LocatedBlock 对象。该对象包含了新数据块的数据块标识和版本号，同时，它的成员变

量 LocatedBlock.locs 提供了数据流管道的信息，通过上述信息，DFSOutputStream 就可以和数据节点联系，通过写数据接口建立数据流管道。客户端写入 FSDataOutputStream 流中的数据，被分成一个一个的文件包，放入 DFSOutputStream 对象的内部队列。该队列中的文件包最后打包成数据包，发往数据流管道，并按照前面讨论的方式，流经管道上的各个数据节点，并持久化。确认包（图 6-27 中的步骤 6）逆流而上，从数据流管道依次发往客户端，当客户端收到应答时，它将对应的包从内部队列移除。

DFSOutputStream 在写完一个数据块后，数据流管道上的节点，会通过和名字节点的 DatanodeProtocol 远程接口的 blockReceived() 方法，向名字节点提交数据块。如果数据队列中还有等待输出的数据，DFSOutputStream 对象需要再次调用 addBlock() 方法，为文件添加新的数据块。

客户端完成数据的写入后，调用 close() 方法关闭流，如图 6-27 中的步骤 8。关闭意味着客户端不会再往流中写入数据，所以，当 DFSOutputStream 数据队列中的文件包都收到应答后，就可以使用 ClientProtocol.complete() 方法通知名字节点关闭文件，完成一次正常的写文件流程。

如果在文件数据写入期间，数据节点发生故障，则会执行下面的操作（注意，这些操作对于写入数据的客户端而言是透明的）：首先，数据流管道会被关闭，已经发送到管道但还没有收到确认的文件包，会重新添加到 DFSOutputStream 的输出队列，这样保证了无论数据流管道中哪个数据节点故障，都不会丢数据。当前正常工作的数据节点上的数据块会被赋予一个新的版本号，并通知名字节点，这样，失败的数据节点从故障恢复过来以后，上面只有部分数据的数据块会因为数据块版本号和名字节点保存的版本号不匹配而删除。然后，在数据流管道中删除错误数据节点并重新建立管道，并继续写数据到正常工作的数据节点。文件关闭后，名字节点会发现该数据块的副本数没有达到要求，会选择一个新的数据节点并复制数据块，创建新的副本。数据节点故障只会影响一个数据块的写操作，后续数据块写入不会受到影响。

在数据块写入过程中，可能出现多于一个的数据节点出现故障的情况，这时，只要数据流管道中的数据节点数满足配置项 ${dfs.replication.min} 的值（默认值是 1），就认为写操作是成功的。后续这个数据块会被复制，直到满足文件的副本系数的要求。

### 6.4.4 数据节点的启动和心跳

本节讨论数据节点和名字节点间的交互，如图 6-28 所示。其中包括数据节点从启动到进入正常工作状态的注册，数据块上报，以及正常工作过程中的心跳等与名字节点相关的远程调用。这部分虽然只涉及 DatanodeProtocol 接口，但有助于读者更进一步理解数据节点和名字节点的关系。

正常启动数据节点或者为升级而启动数据节点，都会向名字节点发送远程调用 versionRequest()，进行必要的版本检查。这里的版本检查，只涉及构建版本号，保证它们间的 HDFS 版本是一致的。正常启动的数据节点，在版本检查结束后，在图 6-28 中的步骤 2 会接着发送远程调用 register()，向名字节点注册。DatanodeProtocol.register() 的主要工作也是检查，通过检查确认该数据节点是名字节点管理集群的成员。也就是说，用户不能将某一

个集群的数据节点，直接注册到另一个集群的名字节点，这保证了整个系统的数据一致性。注册成功后，数据节点会将它管理的所有数据块信息，通过 blockReport() 方法上报到名字节点，帮助名字节点建立 HDFS 文件数据块到数据节点的映射关系。这一步操作完成后，数据节点才正式提供服务。

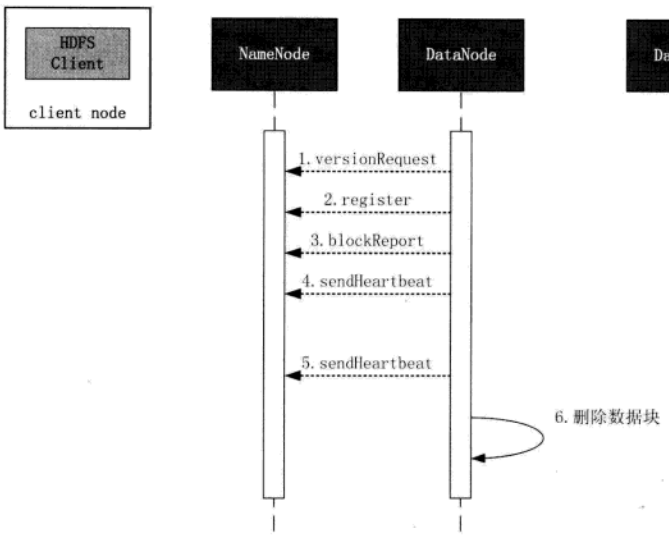



图 6-28　数据节点和名字节点间的交互

由于名字节点和数据节点间存在着主从关系，数据节点需要每隔一段时间发送心跳到名字节点（图 6-28 中的步骤 4、步骤 5），如果名字节点长时间接收不到数据节点的心跳，它会认为该数据节点已经失效。名字节点如果有一些需要数据节点配合的动作，则会通过方法 sendHeartbeat() 返回。该返回值是一个 DatanodeCommand 数组，它带回来一系列名字节点指令。继续上面提到的客户端删除 HDFS 文件例子，操作在名字节点上执行完毕后，被删除文件的数据块会被标记，如果保存这些数据块的数据节点向名字节点发送心跳，则在返回的 DatanodeCommand 数组里，有对应的命令编号为 DNA_INVALIDATE 的名字节点指令。数据节点执行指令，删除数据块，释放存储空间。注意，数据节点删除数据块时，并不知道、也不需要知道该数据块是因为文件被删除，还是均衡器移动数据块后导致副本数升高，或者是用户降低文件的副本系数触发的数据块删除，它只需要简单地执行名字节点指令即可。

应该说，数据节点和名字节点间的交互非常简单，大部分都是数据节点到名字节点的心跳，但考虑到一个规模的 HDFS 集群，一个名字节点会管理上千个数据节点，这样的设计也就非常自然。

### 6.4.5　第二名字节点合并元数据

客户端对 HDFS 的文件系统目录树进行修改时，名字节点都会在编辑日志里写下记录，

以保证系统出现故障后，能够根据这些日志进行恢复。日志会随时间不断地增长，意味着如果系统重启后，需要进行日志恢复的时间会很长。为了避免这种情况发生，HDFS 引入了检查点机制，命名空间镜像（fsimage）文件就是文件系统元数据的持久性检查点，和编辑日志不同，它不能在每次对文件系统元数据进行修改后都进行更新。在一个比较大的运营集群中，fsimage 文件可以有 GB 的大小，所以命名空间镜像加上编辑日志，为名字节点的元数据提供了安全保障。如果名字节点重新启动，元数据可以通过从磁盘中读入命名空间镜像，恢复到某一个检查点，然后再执行检查点后记录的编辑日志，进行重建。一般来说，fsimage 中包括了序列化后的文件系统目录和文件的 i-node。和 Linux 的 i-node 相似，HDFS 索引节点表征一个文件或者目录的元数据信息，以及文件的副本数、修改和访问时间等信息。

上述机制有一个问题，Hadoop 文件系统的编辑日志会随时间不断增长，如果 HDFS 重新启动，名字节点将花很长时间执行编辑日志中的每一个操作，在这个时间内，文件系统是不可用的，减少恢复时间，有利于提供系统的可用性。为了解决这个问题，Hadoop 引入了第二名字节点，它的唯一工作就是辅助名字节点，合并 fsimage 和编辑日志。第二名字节点合并元数据的具体流程如图 6-29 所示。过程涉及远程接口 NamenodeProtocol 和名字节点、第二名字节点上的非 IPC 接口。

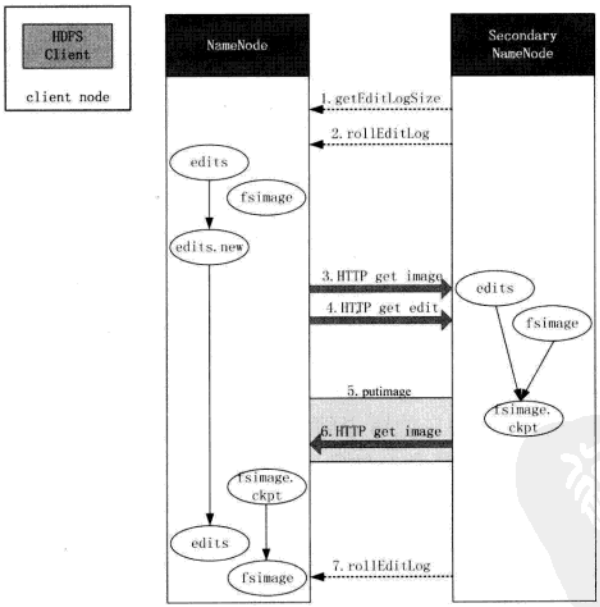



图 6-29 命名空间镜像检查点的产生

该过程由第二名字节点发起，首先通过远程方法 NamenodeProtocol.getEditLogSize() 获得名字节点上编辑日志的大小。如果日志很小，第二名字节点就不需要合并元数据镜像和编辑日志，否则，继续通过该远程接口上的 rollEditLog()，启动一次检查点过程。这时，名字节点需要创建一个新的编辑日志 edit.new，后续对文件系统元数据的改动，都会记录到这

个新的日志里。而原有的命名空间镜像和编辑日志，则由第二名字节点，通过前面分析的 HTTP 接口（图 6-29 中步骤 3 和步骤 4），读取到本地，并在内存中进行合并。合并的结果输出为 fsimage.ckpt，然后，第二名字节点再次发起 HTTP 请求，通知名字节点数据已经准备好。名字节点通过图 6-29 中步骤 6 的 HTTP GET 请求，下载 fsimage.ckpt。名字节点下载新命名空间镜像结束后，第二名字节点通过 NamenodeProtocol.rollFsImage()，完成这次检查点。名字节点在处理该远程方法时，会用 fsimage.ckpt 覆盖原来的 fsimage，形成新的命名空间镜像，同时将新的编辑日志 edit.new 改名为 edit，注意，该编辑日志中的内容包含了在新元数据镜像的基础上，对文件系统的修改。

## 6.5　小结

作为支撑海量数据处理的文件系统，HDFS 需要提供与普通文件系统不一样的特性，正是支持超大文件、检测和快速恢复硬件故障、流式数据访问和简化的一致性模型这些设计目标，决定了 HDFS 采用了主从式体系结构，系统的功能分布在名字节点和第二名字节点、数据节点和客户端等实体中，它们相互配合，为海量数据处理提供底层支持。

接口是观察复杂系统工作的一个很好的出发点，HDFS 是采用多项分布式技术实现的文件系统，HDFS 各个实体间存在着多种信息交互的过程，这些交互过程，有些使用 Hadoop 远程过程调用实现，充分利用了 IPC 机制特有的，和本地过程调用相同的语法外观，在需要交换大量数据的场景中，则使用基于 TCP 或者基于 HTTP 的流式接口。从另一个角度看，这些接口有些和客户端相关，有些是 HDFS 各个服务器间，也就是 HDFS 内部实体间的交互。HDFS 各个实体间的接口如图 6-30 所示。

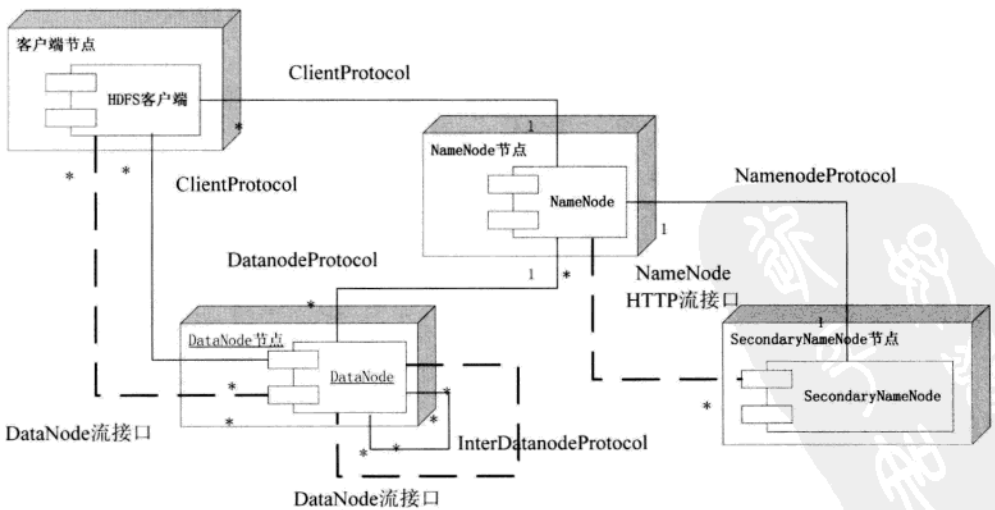



图 6-30　HDFS 各个实体间的接口

基于 IPC 的，和客户端相关的接口包括 ClientProtocol、ClientDatanodeProtocol，分别用于客户端和名字节点、客户端和数据节点的通信。HDFS 内部实体使用 Hadoop 远程过程调用实现的信息交换包括数据节点和名字节点、数据节点和数据节点、第二名字节点和名字节点间的交互。流式接口主要用于大数据传输，包括了客户端和数据节点、数据节点和数据节点间的基于 TCP 的，与数据块操作相关的接口，以及用于第二名字节点合并命名空间镜像和镜像编辑日志，基于 HTTP 协议的接口。

本节的最后，给出 5 个典型的 HDFS 流程，这些流程充分体现了 HDFS 实体间 IPC 接口和流式接口间的配合，了解这些流程，可为后续深入 HDFS 各个节点进行详细代码分析打下坚实基础。

# 第 7 章　数据节点实现

有了第 6 章的基础，我们可以继续来深入了解 Hadoop 分布式文件系统各个实体内部的工作原理。首先是数据节点，它以数据块的形式在本地 Linux 文件系统上保存了 HDFS 文件的内容，并对外提供文件数据访问功能。客户端读写 HDFS 文件时，必须根据名字节点提供的信息，进一步和数据节点交互；同时，数据节点还必须接受名字节点的管理，执行名字节点指令，并上报名字节点感兴趣的事件，以保证文件系统稳定、可靠、高效地运行。

## 7.1　数据块存储

第 5 章曾经简单地讨论过文件系统的物理实现，对一个不定长度的文件进行分块，并将块保存在存储设备上，是一种非常典型的文件系统实现方法。HDFS 也采用了这样的方法，文件数据分块，保存在 HDFS 的存储设备——数据节点上。数据节点将数据块以 Linux 文件的形式保存在节点的存储系统上。数据节点的第一个基本功能，就是管理这些保存在 Linux 文件系统中的数据块。

### 7.1.1　数据节点的磁盘目录文件结构

在 HDFS 中，名字节点、数据节点和第二名字节点都需要在磁盘上组织、存储持久化数据，它们会在磁盘上维护一定的文件结构。在第一次启动 HDFS 集群前，需要通过如下命令对名字节点进行格式化，让名字节点建立对应的文件结构：

```
bin/hadoop namenode -format
```

由于需要在 HDFS 集群运行时动态添加或删除数据节点，所以，数据节点和名字节点不一样，它们不需要进行格式化，而是在第一次启动的时候，创建存储目录。另外，数据节点可以管理多个数据目录，被管理的目录通过配置项 ${dfs.data.dir} 指定，如果该配置项的值为“/data/datanode,/data2/datanode”，则数据节点会管理这两个目录，并把它们作为数据块的保存目录。

---

**注意**　为了讨论方便，后续使用 ${dfs.data.dir} 来指代某个目录的根，如 ${dfs.data.dir}/current 代表数据节点管理目录下的 current 子目录。读者需要注意，这里的 current 包含了多个目录。

---

#### 1. ${dfs.data.dir} 目录

${dfs.data.dir} 目录下一般有 4 个目录和 2 个文件，如图 7-1 所示。

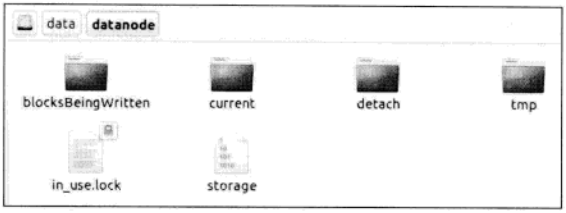



图 7-1 数据节点根目录结构

其中各个目录的作用如下：

- ❑blocksBeingWritten：由名字可以知道，该文件夹保存着当前正在“写”的数据块。和位于“tmp”目录下保存的正在“写”的数据块相比，差别在于“blocksBeingWritten”中的数据块写操作由客户端发起。
- ❑current：数据节点管理的最重要目录，它保存着已经写入 HDFS 文件系统的数据块，也就是写操作已经结束的已“提交”数据块。该目录还包含一些系统工作时需要的文件。
- ❑detach：用于配合数据节点升级，供数据块分离操作保存临时工作文件。
- ❑tmp：该文件夹也保存着当前正在“写”的数据块，这里的写操作是由数据块复制引发的，另一个数据节点正在发送数据到数据块中。

上述目录其实也隐含了数据块在数据节点上可能存在的状态，以上述目录名作为状态，可以得到如图 7-2 所示的状态图。

数据节点上的数据块最开始会处于“blocksBeingWritten”（由客户端写入创建）或“tmp”（由数据块复制写入创建）状态，当数据写入顺利结束后，提交数据块，它们的状态会迁移到“current”。已经提交的数据块可以重新被打开，追加数据，这时，它的状态会返回到“blocksBeingWritten”。处于上述三个状态的数据块都可能被删除，并转移到“最终状态”。

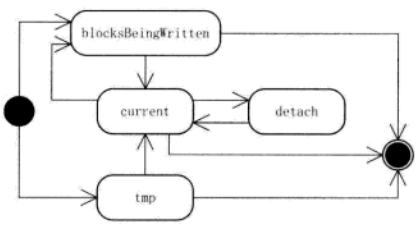



图 7-2 数据节点上数据块的状态

状态“detach”比较特殊，它用于配合数据节点升级，是数据块可能存在的一个临时、短暂的状态。

${dfs.data.dir} 目录下还有两个文件，其中，“storage”保存着如下一段提示信息（注意，该文件的最开始保存着一个二进制整数，所以它是一个二进制文件，不是文本文件）：

```
protected void writeCorruptedData(RandomAccessFile file) throws IOException
{
   final String messageForPreUpgradeVersion =
         "\nThis file is INTENTIONALLY CORRUPTED so that versions\n"
         + "of Hadoop prior to 0.13 (which are incompatible\n"
         + "with this directory layout) will fail to start.\n";
   ......
}
```

由这段信息可知，0.13 版本以前的 Hadoop，使用“storage”作为数据块的保存目录，和现在的目录结构不兼容。这是一种值得学习的技巧，可以防止过旧的 Hadoop 版本在新的目录结构上成功启动，损坏系统。

${dfs.data.dir} 目录下的另一个文件是“in_use.lock”，顾名思义，它表明目录已经被使用，实现了一种“锁”机制。如果停止数据节点，该文件会消失，通过文件“in_use.lock”，数据节点可以保证独自占用该目录，防止两个数据节点（当然，可以在一个节点上启动归属不同 HDFS 集群的多个数据节点）实例共享一个目录，造成混乱。

### 2. ${dfs.data.dir}/current 目录

${dfs.data.dir}/current 目录是 ${dfs.data.dir} 下唯一带目录结构的子目录，其他三个（blocksBeingWritten、detach 和 tmp）都没有子目录。“current”目录结构如图 7-3 所示。

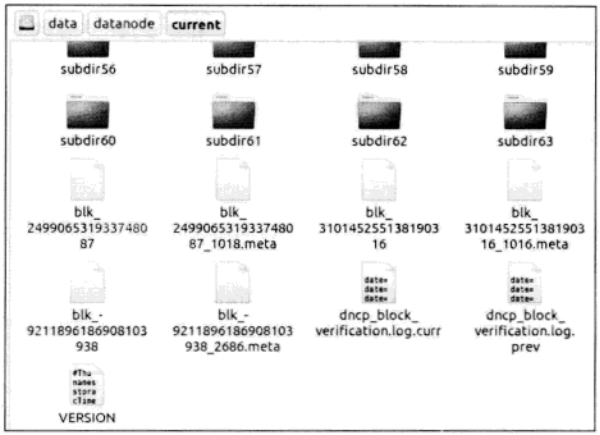



图 7-3　current 目录结构

从图 7-3 中了解到，“current”中既包含目录，也包含文件。其中，大部分文件都以 blk_ 作为前缀，这些文件有两种类型：

❑HDFS 数据块，用来保存 HDFS 文件的内容，如图中的文件“blk_249906531933748087”。

❑使用 meta 后缀标识的校验信息文件，用来保存数据块的校验信息。

和 ChecksumFileSystem 文件系统不同，数据节点中，数据块的校验信息文件不是隐藏文件，但它们的文件内容格式是一样的，都由包含校验配置信息的文件头与一系列校验信息组成。图中，文件“blk_249906531933748087”对应的校验信息文件是“blk_249906531933748087.meta”。

当目录中存储的数据块增加到一定规模时，数据节点会创建一个新的目录，用于保存新的块及元数据。目录块中的块数目达到 64 时（由配置项 ${dfs.datanode.numblocks} 指定），便会创建子目录（如图中的目录“subdir56”），并形成一个更宽的目录树结构。同时，同一父目录下最多会创建 64 个子目录，也就是说，默认配置下，一个目录下最多只有 64 个数据

块（128 个文件）和 64 个目录。通过这两个手段，数据节点既保证了目录的深度不会太深，影响了检索文件的性能，同时避免了某个目录保存了大量数据块，确保每个目录中的文件块数是可控的⊖。

${dfs.data.dir}/current 目录中还有三个特殊的文件，“VERSION”文件是一个 Java 属性文件，包含了运行的 HDFS 版本信息。另外两个文件“dncp_block_verification.log.curr”和“dncp_block_verification.log.prev”，则是数据块扫描器工作时需要的文件，在后续的讨论中会详细分析这些文件。

## 7.1.2 数据节点存储的实现

根据软件开发的一般原则，数据节点的业务逻辑不会在数据节点的文件结构上直接操作，当需要对磁盘上的数据进行操作时，业务逻辑只需要调用管理这些文件结构的对象提供的服务即可。数据节点的文件结构管理包括两部分内容，数据（节点）存储 DataStorage 和文件系统数据集 FSDataset。

HDFS 的服务器实体，名字节点、数据节点和第二名字节点都需要使用磁盘保存数据。所以，在 org.apache.hadoop.hdfs.server.common 包中定义了一些基础类，它们抽象了磁盘上的数据组织，如图 7-4 所示。

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。在这个继承体系中，和 DataStorage 同级的 FSImage 类用于组织名字节点的磁盘数据，FSImage 的子类 CheckpointStorage，则管理着第二名字节点使用的文件结构。CheckpointStorage 和 FSImage 的继承关系，体现了名字节点和第二名字节点的密切联系。

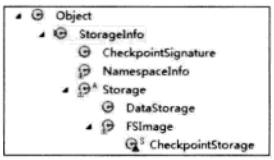


图 7-4 StorageInfo 及其子类

图 7-4 中的另外两个类：NamespaceInfo 和 CheckpointSignature，出现在 HDFS 节点间通信的远程接口中，它们分别应用于数据节点和名字节点通信的 DatanodeProtocol 接口和第二名字节点和名字节点通信的 NamenodeProtocol 接口。如第 6 章所分析的，NamespaceInfo 包含 HDFS 集群存储的一些信息，CheckpointSignature 对象则用于标识名字节点元数据的检查点。另外，StorageInfo 包含的三个字段的含义，分别是 HDFS 存储系统信息结构的版本号 layoutVersion、存储系统标识 namespaceID 和存储系统创建时间 cTime。在 ${dfs.data.dir}/current 目录中，文件“VERSION”保存着这些信息，下面是一个典型的“VERSION”文件包含的内容：

```
#Thu Apr 12 15:59:32 CST 2012
namespaceID=1970677358
storageID=DS-1822991357-10.131.7.79-50010-1332733318628
cTime=0
storageType=DATA_NODE
```

⊖ 根据 Google 工程师的测试结果，在 Linux 的 Ext3 文件系统上，保存了 1000 个文件的目录和保存了 10 000 个文件的目录中打开文件消耗的时间差别不大。

```
layoutVersion=-32
```

“VERSION”是一个典型的 Java 属性文件，数据节点的“VERSION”文件除了上面提到的三个属性外，还有 storageType 和 storageID 属性。

storageType 属性的意义一目了然，它表明这是一个数据节点数据存储目录；storageID 用于名字节点标识数据节点，一个集群的不同数据节点拥有不同的 storageID。

storageID 属性保存在 DataStorage 类中，图 7-5 是 DataStorage 类的完整类图。

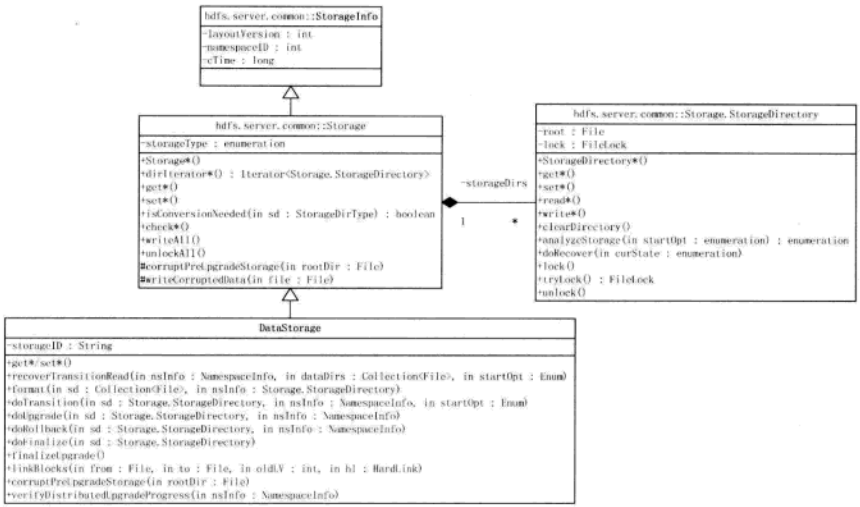



图 7-5　DataStorage 类图

为了讨论的完整性，从 StorageInfo 开始分析图 7-5。

StorageInfo 是一个非常简单的类，包含三个属性和相应的 get/set 方法。它的抽象子类 Storage 也定义在 org.apache.hadoop.hdfs.server.common 包中，为数据节点、名字节点等提供通用的存储服务。Storage 可以管理多个目录，存储目录 StorageDirectory 是 Storage 的内部类。提供了存储目录上的一些通用操作，它们的实现都很简单，值得分析的是 StorageDirectory 的成员变量和 getVersionFile()、tryLock() 等方法。

成员变量 StorageDirectory.root 保存着存储目录的根，dirType 保存着该目录对应的类型。需要注意的是类型为 java.nio.channels.FileLock 的成员变量 lock。FileLock，顾名思义，就是文件锁。Java 的文件锁，要么独占，要么共享。在这里，StorageDirectory 使用的是独占文件锁，对 lock 的加锁代码在 tryLock() 中。

数据节点的文件结构中，当数据节点运行时，${dfs.data.dir} 下会有一个名为“in_use.lock”的文件，就是由 tryLock() 方法创建并上锁的。注意，“创建并上锁”两个操作缺一不可，如果只是创建文件但不加锁，不能防止用户对文件的误操作，如删除文件或移动文件造成“in_use.lock”文件丢失，导致 StorageDirectory.tryLock() 判断逻辑失效。通过 java.nio.channels.FileChannel.tryLock() 方法，StorageDirectory 的 tryLock() 方法获得了文件的独占

锁定，可以避免上述问题，并通过数据节点的实现逻辑，保证对 StorageDirectory 对象指向目录的独占使用。同时，“in_use.lock”文件会在数据节点退出时删除，对应的实现代码是 lockF.deleteOnExit()。

deleteOnExit() 方法由 java.io.File 类提供，当虚拟机退出时，调用了该方法的文件会被虚拟机自动删除。当然，如果 tryLock() 加锁失败，deleteOnExit() 方法自然也不会起作用。相关代码如下：

```
public class StorageDirectory {
   File              root;// 目录的根
   FileLock          lock;// 文件锁
   StorageDirType dirType;// 存储空间的类型
   ……
   public File getVersionFile() {
      return new File(new File(root, STORAGE_DIR_CURRENT),
            STORAGE_FILE_VERSION);
   }
   ……
   FileLock tryLock() throws IOException {
      File lockF = new File(root, STORAGE_FILE_LOCK);
      lockF.deleteOnExit(); //退出时删除文件“in_use.lock”
      RandomAccessFile file = new RandomAccessFile(lockF, "rws");
      FileLock res = null;
      try {
         res = file.getChannel().tryLock();
      }catch(OverlappingFileLockException oe) {
         file.close();
         return null;
      }
      ……
   }
   ……
}
```

StorageDirectory 还提供了一系列的 get*() 方法，如获取存储目录中的“VERSION”文件的 File 对象，可以调用 getVersionFile() 方法。StorageDirectory.analyzeStorage() 和 StorageDirectory.doRecover() 涉及系统升级，会在相关章节解析其原理。

Storage 管理着一个或多个 StorageDirectory 对象，所以 Storage 类中的方法都很简单，或者是在 StorageDirectory 基础上提供的整体操作，或者是对保存的 StorageDirectory 对象进行管理。Storage 是个抽象类，有两个抽象方法，它们和系统升级相关，将在下一节介绍。

和 Storage 一样，DataStorage 的代码可以分为两部分，升级相关代码和升级无关代码。与升级无关的代码很少，需要关注的是 DataStorage 定义的三个常量，对照数据节点的文件结构，读者自然能明白这些常量的用途。代码如下：

```
public class DataStorage extends Storage {
   // 常量
   final static String BLOCK_SUBDIR_PREFIX = "subdir";
```

```
    final static String BLOCK_FILE_PREFIX = "blk_";
    final static String COPY_FILE_PREFIX = "dncp_";

    private String storageID;
    ......
    void format(StorageDirectory sd,
                NamespaceInfo nsInfo) throws IOException {
       sd.clearDirectory(); // 创建目录
       this.layoutVersion = FSConstants.LAYOUT_VERSION;
       this.namespaceID = nsInfo.getNamespaceID();
       this.cTime = 0;
       // store storageID as it currently is
       sd.write();
    }
    ......
}
```

数据节点在第一次启动的时候，会调用 DataStorage.format() 创建存储目录结果。

format() 方法很简单，通过 StorageDirectory.clearDirectory() 删除原有目录及数据并重新创建目录，然后为“VERSION”文件中的属性赋值并将其持久化到磁盘。format() 的第一个参数类型是 StorageDirectory，如果数据节点管理着多个目录，这个方法会被调用多次，在不同的目录下创建节点文件结构；第二个参数 nsInfo 是从名字节点返回的 NamespaceInfo 对象，携带了存储系统标识 namespaceID 等信息，该标识最终存放在“VERSION”文件中。注意，由于“VERSION”文件保存在 current 目录中，保存文件的同时会创建对应的目录。

### 7.1.3　数据节点升级

上一节将 DataStorage 的功能分为升级相关和升级无关两个部分有点勉强，其实，DataStorage 的主要功能是对存储空间的生存期进行管理，通过 DataStorage.format() 创建存储目录，或者利用 DataStorage.doUpgrade() 进行升级，都是存储空间生存期管理的一部分。对于数据节点，存储空间生存期管理的关注点还是系统升级。

#### 1. 升级、升级提交和升级回滚

Hadoop 实现了严格的版本兼容性要求，只有来自相同发布版本的组件才能保证相互的兼容性。大型分布式系统的升级需要一个周密的计划，必须考虑到诸如：持久化的信息是否改变？如果数据布局改变，如何自动地将原有数据和元数据迁移到新版本格式？升级过程中如果出现错误，如何保证数据不丢失？升级出现问题时，怎么才能够回滚升级前的状态？

如果文件系统的布局不需要改变，集群的升级变得非常简单，关闭旧进程，升级配置文件，然后启动新版本的守护进程，客户端使用新的库，就可以完成升级。同时，升级回滚也很简单，使用旧版本和旧配置文件，重启系统即可。如果持久化的信息在上述过程中需要改变到新的格式，特别是考虑到可能的回滚，升级的过程就变得复杂。

HDFS 升级时，需要复制以前版本的元数据（对名字节点）和数据（对数据节点）。在数据节点上，升级并不需要两倍的集群存储空间，数据节点使用了 Linux 文件系统的硬链

接，保留了对同一个数据块的两个引用，即当前版本和以前版本。通过这样的技术，就可以在需要的时候，轻松回滚到以前版本的文件系统。注意，在已升级系统上对数据的修改，升级回滚结束后将会消失。

（1）升级

为简化实现，HDFS 最多保留前一版本的文件系统，没有多版本的升级、回滚机制。同时，引入升级提交机制（通过 hadoop dfsadmin-finalizeUpgrade 可提交一次升级），该机制用于删除以前的版本，所以在升级提交后，就不能回滚到以前的版本了。综上所述，数据节点（和名字节点）的升级过程采用了如图 7-6 所示的状态机。其中，“NORMAL1”是升级前版本，“NORMAL2”是升级后的新版本。

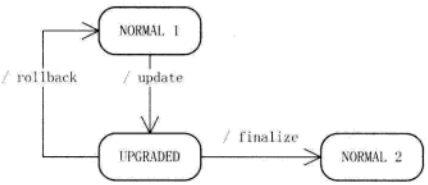


图 7-6 数据节点的升级

数据节点升级的实现在 DataStorage.doUpgrade() 方法中。其中，升级过程涉及如下几个目录：

- curDir：当前版本目录，通过 StorageDirectory.getCurrentDir() 获得，目录名为“current”。
- prevDir：上一版本目录，目录名为“previous”，可通过 StorageDirectory 的 getPreviousDir() 方法得到；这里的“上一版本”指的是升级前的版本。
- tmpDir：上一版本临时目录，即目录 ${dfs.data.dir}/previous.tmp。

doUpgrade() 方法的主要流程是：首先确保上述涉及的工作目录处于正常状态。如检查“current”目录是否存在，如果不存在，也就不需要升级了（“current”目录在数据节点格式化的时候创建）；如果“previous”目录存在，删除该目录。注意，该删除操作相当于提交了上一次升级，同时保证了 HDFS 最多保留前一版本数据的要求；保证“tmpDir”目录不存在。应该说，在（通过硬链接）移动数据前，“previous”目录和“previous.tmp”都是不存在的。

支持升级回滚，就必须保留升级前的数据，在数据节点，就是保存数据块以及数据块的校验信息文件。在 doUpgrade() 中，保留升级前数据是通过建立文件的硬链接实现的。

---

**注意** *硬链接是一种特殊的文件系统机制，它允许一个文件可以有多个名称。当一个文件有多个名称时，删除其中的一个名称，并不会删除文件数据，只有所有的文件名称被删除后，文件系统才会真正删除文件数据。*

---

DataStorage.doUpgrade() 保留升级前数据的动作一共有两步：首先将“current”目录名改为“previous.tmp”，然后，调用 linkBlocks()，在新创建的“current”目录下，建立到“previous.tmp”目录中数据块和数据块校验信息文件的硬链接。DataStorage.linkBlocks() 执行结束后，doUpgrade() 方法还需要在“current”目录下创建新版本的“VERSION”文件，最后，将“previous.tmp”目录名改为“previous”，完成升级。这个时候，数据节点上的存储空间会有“previous”和“current”两个目录，而且，“previous”和“current”包含了同

样的数据块和数据块校验信息文件，但它们有各自的“VERSION”文件。另外，升级过程中需要的“previous.tmp”目录已经消失。代码如下：

```
void doUpgrade(StorageDirectory sd,NamespaceInfo nsInfo)
      throws IOException {
   ……
   File curDir = sd.getCurrentDir();
   File prevDir = sd.getPreviousDir();
   assert curDir.exists() : "Current directory must exist.";
   // 删除可能存在的previous 目录
   if (prevDir.exists())
      deleteDir(prevDir);
   File tmpDir = sd.getPreviousTmp();
   assert !tmpDir.exists() : "previous.tmp directory must not exist.";
   // 目录改名，current ->previous.tmp
   rename(curDir, tmpDir);
   // 为数据块和数据块校验信息文件建立硬链接
   linkBlocks(tmpDir, curDir, this.getLayoutVersion(), hardLink);
   // 写新的VERSION 文件
   this.layoutVersion = FSConstants.LAYOUT_VERSION;
   assert this.namespaceID == nsInfo.getNamespaceID() :
         "Data-node and name-node layout versions must be the same.";
   this.cTime = nsInfo.getCTime();
   sd.write();
   // 目录改名，previous.tmp -> previous
   rename(tmpDir, prevDir);
   ……
}
```

有了 doUpgrade() 流程的基础，升级回滚和升级提交的实现就很好理解了。

（2）升级回滚

升级回滚 doRollback() 的实现有如下要点：

1）在完成对各个工作目录状态的检查后，需要保证升级能够回滚到正确的版本上去。这个步骤是通过比较保存在“previous”目录下“VERSION”文件中的 HDFS 存储系统信息结构的版本号 layoutVersion 和存储系统创建时间 cTime 和回滚后相应的 layoutVersion 和 cTime 来做判断。

2）由于“previous”目录保存了升级前的所有数据，所以，doRollback() 其实只需要简单地将“previous”目录改名成“current”，就可以完成回滚。但由于现在版本的工作目录，“current”目录存在，所以采用了三步操作完成改名动作：先将“current”目录名改为“removed.tmp”，然后将“previous”目录名修改为“current”，最后删除“removed.tmp”目录。代码如下：

```
void doRollback( StorageDirectory sd, NamespaceInfo nsInfo)
      throws IOException {
   ……// 检查工作目录处于正常状态
   DataStorage prevInfo = new DataStorage();
```

```
    StorageDirectory prevSD = prevInfo.new StorageDirectory(sd.getRoot());
    prevSD.read(prevSD.getPreviousVersionFile());
    if (!(prevInfo.getLayoutVersion() >= FSConstants.LAYOUT_VERSION
          && prevInfo.getCTime() <= nsInfo.getCTime()))  // 不能回滚
       throw new InconsistentFSStateException……
    ……
    // 目录改名，current -> removed.tmp
    rename(curDir, tmpDir);
    // 目录改名，previous ->current
    rename(prevDir, curDir);
    // 删除目录 removed.tmp
    deleteDir(tmpDir);
    ……
}
```

（3）升级提交

升级提交 doFinalize() 更简单，它只需要将“previous”目录删除即可。但实际上，DataStorage.doFinalize() 需要将“previous”目录名改为“finalized.tmp”，然后删除“finalized.tmp”来删除目录，而不是直接删除“previous”来提交升级。请参考下面的代码：

```
void doFinalize(StorageDirectory sd) throws IOException {
   …… // 检查工作目录处于正常状态
   final File tmpDir = sd.getFinalizedTmp();
   // 目录改名，previous -> finalized.tmp
   rename(prevDir, tmpDir);

   // 通过一个线程来删除 finalized.tmp 目录
   new Daemon(new Runnable() {
      public void run() {
         try {
            deleteDir(tmpDir);
         }
         ……
      }
      ……
   }).start();
}
```

### 2. 存储空间状态

也许读者会有疑问，为什么在 doFinalize() 中不能直接删除“previous”目录？同样，在 doRollback() 和 doUpgrade() 中，都存在“removed.tmp”和“previous.tmp”这样的临时目录，它们的作用是什么？

图 7-6 中，升级、升级提交或升级回滚都需要进行一定的操作。在系统升级中，doUpgrade() 方法需要建立数据块文件的硬链接，在执行这个任务的中间，如果设备出现故障（如停电），那么，存储空间就可能处于某一个中间状态。引入上述这些临时目录，系统就能够判断目前 doUpgrade() 处于什么阶段，并采取相应的应对措施。

还是以 doUpgrade() 为例，假设现在不使用“previous.tmp”，而是直接在新建的

“current”目录上创建到“previous”目录中文件的硬链接，这个过程中间系统发生错误，这时“current”目录里的数据就和“previous”目录中的不一致，升级失败。但是，当数据节点再次启动时，它并不能根据“current”目录和“previous”目录的不一致，判断在升级的过程中，系统是否发生过错误。因为升级成功的数据节点，运行一段时间以后，它的“current”目录和“previous”目录一般也是不一致的。而且，数据节点不能区分这两个“不一致”，也就无法判断系统是否发生过错误。引入临时的“previous.tmp”目录后，数据节点启动时，如果发现它的存储空间中有该临时目录，就可以准确判断出上次的升级操作没有正常结束。

考虑到升级、升级提交和升级回滚中可能出现的故障，并结合其他可能的生存期状态，Storage 通过枚举 StorageState 完整地定义了存储空间的可能状态。代码如下：

```
public abstract class Storage extends StorageInfo {
    ......
    public enum StorageState {
        NON_EXISTENT,
        NOT_FORMATTED,
        COMPLETE_UPGRADE,
        RECOVER_UPGRADE,
        COMPLETE_FINALIZE,
        COMPLETE_ROLLBACK,
        RECOVER_ROLLBACK,
        COMPLETE_CHECKPOINT,
        RECOVER_CHECKPOINT,
        NORMAL;
    }
    ......
}

public interface HdfsConstants {
    ......
    // 启动选项
    static public enum StartupOption{
        FORMAT  ("-format"),
        REGULAR ("-regular"),
        UPGRADE ("-upgrade"),
        ROLLBACK("-rollback"),
        FINALIZE("-finalize"),
        IMPORT  ("-importCheckpoint");
        ......
    }
    ......
}
```

注意，Storage 状态和当前准备施加在存储空间的动作相关。这些动作以枚举的形式，定义在 HdfsConstants 接口中，如下所示：

❑FORMAT("-format")：格式化系统。

❑REGULAR("-regular")：正常启动 HDFS。
❑UPGRADE("-upgrade")：升级系统。
❑ROLLBACK("-rollback")：从升级中回滚到前一个版本。
❑FINALIZE("-finalize")：提交一次升级。
❑IMPORT("-importCheckpoint")：从名字节点的一个检查点恢复。

---

**注意** 了解 HDFS 的读者应该对这些启动选项很熟悉，如 upgrade 用于发起（存储空间的）一次升级（importCheckpoint 选项应用于名字节点中，用于恢复命名空间到某一个检查点，将在第 8 章讨论）。

---

为什么要结合启动选项来讨论存储空间的状态呢？以存储空间对应的目录不存在为例，如果系统以“格式化系统”选项启动，这时存储空间的状态是 NOT_FORMATTED，即没有格式化，如果系统以其他选项（如 REGULAR 或 UPGRADE）启动，那么存储空间状态应该是“目录不存在”。

Storage 定义的十种状态如下：

❑NON_EXISTENT：以非 FORMAT 选项启动时，目录不存在，或者目录不可写，或者配置项对应的路径是个文件，而非目录等，都可以认为存储空间不存在。
❑NOT_FORMATTED：以 FORMAT 选项启动，同时目录不存在，这时存储空间状态为没有被格式化。
❑COMPLETE_UPGRADE：“previous.tmp”目录存在，同时“current”目录下的“VERSION”文件存在，可完成升级状态。
❑RECOVER_UPGRADE：“previous.tmp”目录存在，“current/VERSION”文件不存在，存储空间应该从升级过程中恢复。
❑COMPLETE_FINALIZE：存在“finalized.tmp”目录，存储空间可以继续执行提交升级。
❑COMPLETE_ROLLBACK：存在“previous.tmp”目录，同时“current/VERSION”文件存在，可以继续执行提交回滚。
❑RECOVER_ROLLBACK：存在“previous.tmp”目录，“current”目录下“VERSION”文件不存在，需要中断升级提交，恢复到上一个状态。
❑COMPLETE_CHECKPOINT：应用于名字节点，用于导入元数据检查点，存储空间可完成检查点导入。
❑RECOVER_CHECKPOINT：应用于名字节点，用于导入元数据检查点，需要恢复导入检查点前状态。
❑NORMAL：目录处于正常工作状态。

图 7-7 说明了上述状态以及状态间的关系。

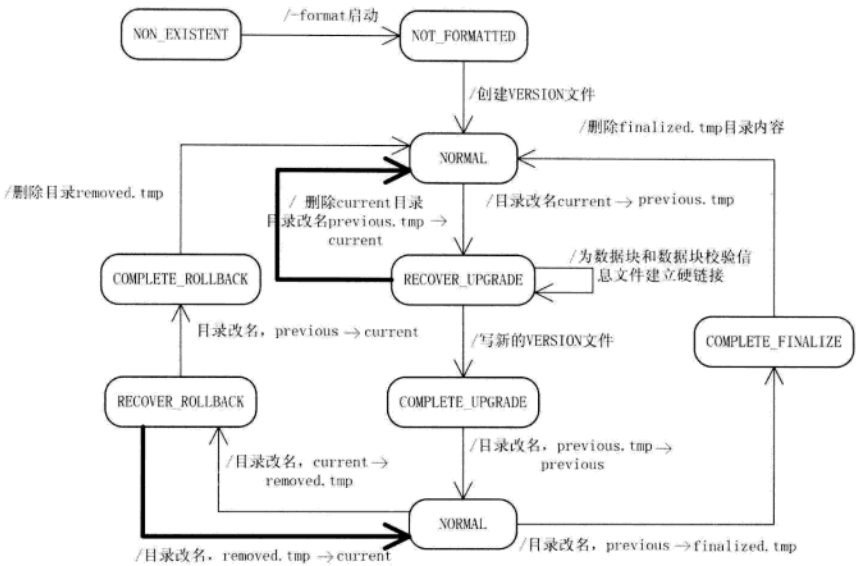



图 7-7　数据节点 Storage 状态机

可以注意到，图中存在两个 NORMAL 状态。Storage 的 NORMAL 状态，表明存储空间里不存在扩展名为“tmp”的文件。图 7-7 下面的 NORMAL 状态在图 7-6 中的对应状态是 UPGRADED。这时，存储空间同时包含了当前版本和以前版本的数据。Storage.analyzeStorage() 方法能够根据上述状态存在的条件，判断出目前存储空间的状态，限于篇幅就不再仔细分析该方法，从图 7-7 所示的状态间转移的动作，读者不难理解 analyzeStorage() 中状态的判断条件。

Storage.analyzeStorage() 的输出结果被 Storage.doRecover() 使用，该方法根据存储空间的当前状态，执行相应的动作，恢复存储空间状态到 Storage 的 NORMAL 状态。

以 COMPLETE_UPGRADE 和 RECOVER_UPGRADE 两个状态为例。如果当前状态是 COMPLETE_UPGRADE，则只需要将“previous.tmp”目录名改为“previous”，就可以恢复为 NORMAL 状态，这个时候，存储空间里会保存当前版本和以前版本的数据。

如果现在的状态是 RECOVER_UPGRADE，那么就必须删除存储空间里的“current”目录，并将“previous.tmp”目录名改为“current”，以恢复到升级前，存储空间只保存一个版本的数据的 NORMAL 状态。代码如下：

```
public void doRecover(StorageState curState) throws IOException {
   File curDir = getCurrentDir();
   String rootPath = root.getCanonicalPath();
   switch(curState) {
   case COMPLETE_UPGRADE:
      ......
      // 目录改名，previous.tmp -> previous
```

```
            rename(getPreviousTmp(), getPreviousDir());
            return;
        case RECOVER_UPGRADE:
            ......
            if (curDir.exists())
                deleteDir(curDir);// 删除目录 current
            // 目录改名, previous.tmp -> current
            rename(getPreviousTmp(), curDir);
            return;          ......
        }
        ......
    }
```

Storage 和 DataStorage 一起提供了一个完美的 HDFS 数据节点升级机制，简化了大型分布式系统的运维。它们不但解决了系统升级过程中数据格式转换、升级回滚等常见问题、并保证了这个过程中不会丢失数据；同时，它们的设计，特别是 Storage 状态机，以及配合状态机工作的临时文件，提供了一个完备的升级方法，在升级过程或者回滚过程中的任意一个步骤出现错误，都可以通过状态机，恢复到正常状态。HDFS 数据节点升级的 Storage 和 DataStorage 实现，非常具有借鉴意义。

## 7.1.4 文件系统数据集的工作机制

DataStorage 专注于数据节点存储空间的生存期管理，在存储空间上，与数据节点逻辑密切相关的存储服务，如创建数据块文件、维护数据块文件和数据块校验信息文件的关系等，则实现在文件系统数据集 FSDataset 中。

### 1. FSDatasetInterface 接口

FSDataset 继承自 FSDatasetInterface 接口，而 FSDatasetInterface 又继承自 FSDatasetMBean，如图 7-8 所示。

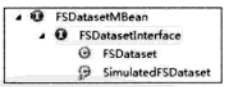



图 7-8 FSDatasetInterface 接口继承关系

FSDatasetMBean 是符合 Java 管理扩展（Java Management eXtensions，JMX）规范中的可管理资源 MBean（Managed Bean）接口。JMX[⊖]是 SUN 公司提出的一套管理框架。它定义了完整的框架体系、设计模式、API 接口、基于网络的管理和监控服务等，通过这些可以对 Java 程序进行管理控制。以 FSDatasetMBean 为例，它提供了获取数据节点管理的存储空间信息的一些方法，如 getDfsUsed() 方法返回这个数据节点上 HDFS 已经使用的存储空间大小。JMX 可以使用 FSDatasetMBean，以标准的方式管理数据节点的存储空间。

FSDatasetInterface 继承自 FSDatasetMBean，通过引入 FSDatasetInterface 接口，方便了 HDFS 的测试，SimulatedFSDataset 也实现了 FSDatasetInterface，作为数据节点存储服务的一个测试桩，在 HDFS 的单元测试中得到广泛的应用。

⊖ 本书中不讨论和 JMX 相关的 Common、HDFS 代码。

FSDatasetInterface 接口如图 7-9 所示。

如图 7-9 所示，FSDatasetInterface 接口的方法可以分为以下三类：

- 数据块相关的方法：FSDataset 管理了数据节点上的数据块，大量的 FSDataset 方法和数据块相关，如创建数据块、打开数据块上的输入、输出流、提交（finalize）数据块等。
- 数据块校验信息文件相关：包括维护数据块和校验信息文件关系，获取校验信息文件输入流等方法。
- 其他：包含 FSDataset 健康检查、关闭 FSDataset 的 shutdown() 方法等。

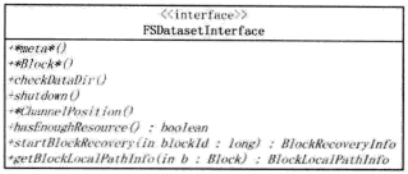



图 7-9　FSDatasetInterface 接口

### 2. FSDataset 的成员变量

文件数据集 FSDataset 实现了 FSDatasetInterface 接口，为了讨论 FSDataset 的工作原理，必须先分析 FSDataset 的几个内部类。

传统文件系统的实现，文件的物理结构，特别是对文件内容的组织，是一个重要的议题。如在随机存储设备（如磁盘）上，一般有连续分配、链接表和使用索引的链式表分配等数据组织方式。HDFS 的文件内容组织比较简单，它的数据块就是数据节点文件系统的普通文件，所以，讨论文件系统的物理结构时，不需要过多考虑存储设备的存储结构，需要考虑的是如何方便、快捷地管理数据节点文件系统上的这些普通文件。

FSDataset 借鉴了 Linux 逻辑卷管理（Logic Volumn Management，LVM）的一些思想。LVM 使系统管理员可以更方便地分配存储空间，它的出现，改变了磁盘空间的管理理念。传统的文件系统是基于分区的，一个文件系统对应一个分区，这种方式直观，但不灵活，在实际应用中，会出现分区不平衡，存储空间和硬件能力得不到充分利用等问题。

Linux 的 LVM 引入了物理卷、卷组和逻辑卷的存储概念：

- 物理卷（Physical Volumn）：典型的物理卷是磁盘分区，也可以是整个磁盘或者是其他块设备（如磁盘阵列）。
- 卷组（Volumn Group）：卷组建立在物理卷之上，是由一个或者多个物理卷所组成的存储器池。
- 逻辑卷（Logic Volumn）：相当于非 LVM 系统中的分区，它建立在卷组的基础上，是一个标准的块设备，可以在逻辑卷上建立文件系统。

LVM 在逻辑和物理存储之间映射数据，允许数据保存在多个物理设备上。采用 LVM 后，磁盘的多个分区由 LVM 统一管理，可以根据需要加入和移走分区，以扩大或缩小卷组的可用容量；同时，文件系统建立在逻辑卷上，可以在卷组容量范围内根据需要改变大小，还可以跨磁盘分区。在创建逻辑卷时，采用条带化（Striping）方式，可以将数据放在多个磁盘上，从而提高了读写数据的速度。

FSDataset 没有使用 Linux LVM 的方式管理它的存储空间，但是，FSDataset 借鉴了

LVM ⊖的一些概念，可以管理多个数据目录。一般情况下，这些不同的数据目录配置在不同的物理设备上，以提高磁盘的数据吞吐量。

文件数据集将它管理的存储空间分为三个级别，分别用FSDir、FSVolume和FSVolumeSet进行抽象。在分析数据节点文件结构时，我们已经知道，数据节点可以管理多个数据目录，而数据目录下包含已经提交数据块的“current”目录中，既包含目录，也包含文件。在“current”目录中引入目录层次，既保证每个目录不会包含太多的数据块文件，也保证了目录的深度不会太深，以获得平衡的检索性能。FSDataset的内部类如图7-10所示。

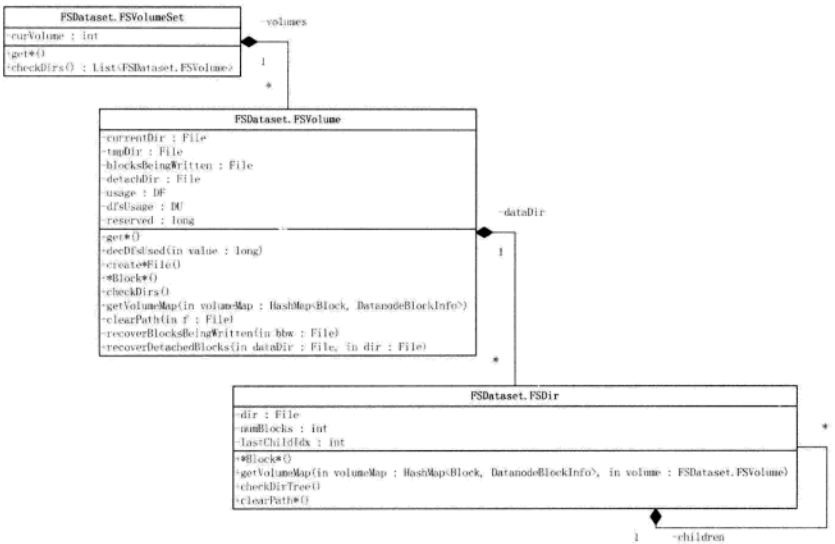



图7-10 FSDataset的内部类

FSDir对象表示了“current”目录下的子目录，其成员变量children包含了目录下的所有子目录，形成目录树。FSVolume是数据目录配置项${dfs.data.dir}中的一项，数据节点可以管理一个或多个数据目录，系统中也就存在一个或多个FSVolume对象。这些FSVolume对象由FSVolumeSet管理。FSDir、FSVolume和FSVolumeSet中的大部分方法都很简单。

由于有上述内部类存在，FSDataset的成员变量不多，比较重要的有volumes，它管理数据节点的所有存储空间。成员ongoingCreates和volumeMap是两个HashMap，其中，ongoingCreates保存着当前正在进行写操作的数据块和对应文件的映射，volumeMap则保存着已经提交的数据块和对应文件的映射。代码如下：

```
public class FSDataset implements FSConstants, FSDatasetInterface {
    ……
    FSVolumeSet volumes;
    private HashMap<Block,ActiveFile> ongoingCreates = new……;
```

⊖ 后续HDFS版本规划中有类似于Linux逻辑卷管理的机制。

```
    ……
    HashMap<Block,DatanodeBlockInfo> volumeMap = new……;
    ……
    FSDatasetAsyncDiskService asyncDiskService;// 后面讨论
    ……
}
```

成员变量 ongoingCreates 和 volumeMap 在 FSDataset 的实现中扮演着重要角色。其中，ongoingCreates 是数据块 Block 对象到 ActiveFile 对象的映射。ActiveFile，顾名思义，就是“活跃文件”，它保存着处于写状态的数据块的一些附加信息，包括：数据块对应的文件 file、当前操作该文件的线程 threads 和当前数据块可见长度 visibleLength 等。声明如下：

```
static class ActiveFile {
    final File file;
    final List<Thread> threads = new ArrayList<Thread>(2);
    private volatile long visibleLength;

    // 涉及较多的细节，我们不讨论 DataNode 重启时的数据块恢复
    final boolean wasRecoveredOnStartup;
    ……
}
```

成员变量 volumeMap 保持着所有数据块在数据节点上的一些附加信息，字段包含的信息如代码中注释，如下：

```
class DatanodeBlockInfo {
    private FSVolume volume;// 数据块保存的数据目录
    private Filefile;// 对应的文件
    private boolean detached;// 是否写时复制
    ……
}
```

### 3. 打开数据块和提交数据块

打开数据块和提交数据块是文件数据集 FSDataset 中最重要的两个方法。

写数据块时，数据节点会调用 FSDatasetInterface.writeToBlock() 方法创建 BlockWriteStreams 对象。BlockWriteStreams 是一个包装器，包含到数据块文件及其校验信息文件的两个输出流，调用者后续可使用这两个输出流，写数据和校验信息。代码如下：

```
public BlockWriteStreams writeToBlock(Block b, boolean isRecovery,
      boolean replicationRequest) throws IOException {
    // 主要步骤 1：数据块有效？
    if (isValidBlock(b)) {
        if (!isRecovery) { // 如果不是为恢复写数据块，抛出异常
            throw new BlockAlreadyExistsException(……);
        }

        // 如果数据块已经提交，即数据节点已经成功接收到所有写数据的数据包并发送 ack，
        // 但客户端因为某种原因没有收到 ack，重新打开连接并重发数据
        // 或者有到该数据块的 append（追加）操作
```

```
    detachBlock(b, 1);
}
long blockSize = b.getNumBytes();// 数据块的当前大小

File f = null;
List<Thread> threads = null;
synchronized (this) {
    // 主要步骤 2：数据块是否已经处于创建过程中？
    ActiveFile activeFile = ongoingCreates.get(b);
    if (activeFile != null) {
        f = activeFile.file;
        threads = activeFile.threads;

        if (!isRecovery) {
            throw new BlockAlreadyExistsException(……);
        } else {
            for (Thread thread:threads) {
                thread.interrupt();
            }
        }
        ongoingCreates.remove(b); // 移除数据块和 ActiveFile 对象的关系
    }

    // 主要步骤 3：创建文件并做记录
    FSVolume v = null;
    if (!isRecovery) {
        v = volumes.getNextVolume(blockSize);
        // 在指定的 volumn 中创建临时文件，存放该数据块
        f = createTmpFile(v, b, replicationRequest);
    } else if (f != null) {
        // 文件存在，复用临时文件，保存数据块
        v = volumeMap.get(b).getVolume();
        volumeMap.put(b, new DatanodeBlockInfo(v, f));
    } else {
        // 为追加操作重新打开数据块
        v = volumeMap.get(b).getVolume();
        f = createTmpFile(v, b, replicationRequest);
        File blkfile = getBlockFile(b);
        File oldmeta = getMetaFile(b);
        File newmeta = getMetaFile(f, b);

        // 校验信息文件文件改名
        if (!oldmeta.renameTo(newmeta)) {
            throw new IOException(……);
        }

        // 数据块文件改名，相当于移动到 tmp 目录下
        if (!blkfile.renameTo(f)) {
            ……
        }
```

```
        }
        if (f == null) {
            throw new IOException(……);
        }
        // 如果是一个复制请求，该数据块不是一个永久数据块，可能在数据节点重启时被删除
        // 如果是一个写或追加操作，则是一个有效数据块
        if (replicationRequest) {
            volumeMap.put(b, new DatanodeBlockInfo(v));
        } else {
            volumeMap.put(b, new DatanodeBlockInfo(v, f));
        }
        ongoingCreates.put(b, new ActiveFile(f, threads));
    }

    // 主要步骤 2：等待线程结束
    try {
        if (threads != null) {
            for (Thread thread:threads) {
                thread.join();
            }
        }
    } ……

    File metafile = getMetaFile(f, b);
    ……
    return createBlockWriteStreams( f , metafile);
}
```

writeToBlock() 方法需要处理数据块创建、追加、恢复（通过参数 isRecovery 判断）、复制（通过参数 replicationRequest 判断）四种情况，代码逻辑比较复杂。该方法通过 isValidBlock() 判断数据块是否有效，首先要明确在什么情况下数据块有效。有效的数据块必需满足以下条件：

❑对应的数据块文件存在（可能因设备损坏导致 volumeMap 中保存的数据块无效）。

❑该数据块已经提交，也就是当前数据块上没有写的动作。

如果数据块有效，即数据块文件存在，那么这个 writeToBlock() 调用可能由追加操作（append）触发，也可能是客户端进行错误恢复的一个步骤。在这两种情况下，均需要调用 detachBlock() 方法，进行必要的数据块分离操作。

什么是数据块分离操作呢？数据节点在升级过程未提交或回滚前，使用 Linux 文件系统的硬链接，保留了对同一个数据块的两个引用（分别位于“previous”目录和“current”目录）。当进行升级回滚，恢复升级前版本时，必须保证在已升级系统上对数据的修改也能回滚。在数据块文件存在硬链接的情况下对它进行修改，将会同时影响两个引用，也就影响到上一个版本的数据，即对“current”目录中的数据块文件进行修改，会同时修改“previous”目录中的对应文件。这时，需要为将进行修改的数据块文件复制一份数据，也就是去除硬链接，保证后续的数据块修改不会影响前一个版本的数据，如图 7-11 所示。

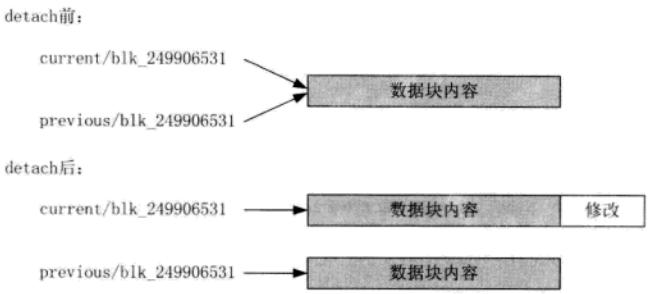



图 7-11 数据块分离操作

**注意** 数据块分离时，需要进行数据块文件复制，复制过程中的临时文件会保存在${dfs.data.dir}/detach 目录中，复制结束后，该临时文件才被移动到目标目录中。“detach”目录的作用和升级过程中的临时目录作用是一样的，用于防止分离操作过程中数据节点崩溃导致的数据不一致问题。

writeToBlock() 方法的第二个检查，是判断数据块是否处于写入过程中，并根据情况做相应的处理。如果数据块保存在映射 ongoingCreates 中，表明它正处于写入状态，这时，用户不能再创建数据块（在代码中，将抛出异常 BlockAlreadyExistsException），只能进行数据块恢复（通过请求参数 isRecovery 判断）。数据块恢复一般由某一次不成功的写操作触发，由于底层协议（主要是 TCP 协议）和系统实现的限制，和那个不成功写操作的相关资源可能还被占用中，如上次写操作的工作线程还在 Socket 上进行读操作，等待数据，这个时候需要通过线程中断，停止上述线程并释放资源。在方法的后半部分，writeToBlock() 会等待被中断的线程结束，以保证调用返回以后，数据块恢复过程能正确地写数据，不会出现两个线程写一个文件的情况。

完成上述步骤以后，FSDataset.writeToBlock() 开始为数据块创建文件，并在映射表中记录相关的信息。这部分代码同样需要根据“是否是数据块恢复”的上下文，分类处理。如果是新创建的数据块，则先分配 FSVolume，然后在该数据目录中创建临时文件。通过 FSVolume 的顺序分配，FSDataset 实现了类似 Linux LVM 的功能，采用类似条带化的方法，在不同的 FSVolume 中创建新的临时文件，来充分利用多个磁盘的吞吐能力。

使用 createTmpFile() 方法创建临时文件时，将根据是否是数据块复制请求，分别将临时文件创建在目录 ${dfs.data.dir}/tmp（数据块复制）或 ${dfs.data.dir}/blocksBeingWritten 下，代码如下：

```
File createTmpFile(Block b, boolean replicationRequest)
      throws IOException {
   File f = null;
   if (!replicationRequest) {
      f = new File(blocksBeingWritten, b.getBlockName());
   } else {
```

```
            f = new File(tmpDir, b.getBlockName());
        }
        return createTmpFile(b, f);
    }
```

如果是为数据恢复而调用 writeToBlock() 方法，这时需要判断被恢复的数据块文件是否还处于写入状态。如果是，即 ongoingCreates 存在对应的记录，则继续使用该文件；如果数据块已经不处于写入状态，则需要为写入重新打开数据块。这里的打开数据块和文件系统中的打开文件不同：数据节点将处于写入状态的数据块对应的文件保存在 ${dfs.data.dir}/blocksBeingWritten 目录下，打开数据块，需要将数据块对应的文件创建或移动到上述目录中。

在 FSDataset.writeToBlock() 方法的最后，打开数据块的相关信息会保持到 volumeMap 和 ongoingCreates 中，并创建包含到打开数据块对应的文件和校验信息文件输出流的 BlockWriteStreams 对象。

和 writeToBlock() 方法对应的是 finalizeBlock() 方法，它用于提交一个被打开的数据块，其作用类似于关闭一个打开的文件。提交数据块的过程非常简单，获得该数据块保存在 ongoingCreates 和 volumeMap 中的信息，根据这些信息，通过 FSVolume 的 addBlock() 方法，最终调用 FSDir.addBlock()，将数据文件和校验信息文件移动到“current”的某个子目录下。然后更新 volumeMap 中保存的信息，并删除在 ongoingCreates 中的记录。代码如下：

```
private synchronized void finalizeBlockInternal(Block b,
                boolean reFinalizeOk) throws IOException {
    ActiveFile activeFile = ongoingCreates.get(b);
    ......
    File f = activeFile.file;
    ......
    FSVolume v = volumeMap.get(b).getVolume();
    ......
    File dest = null;
    dest = v.addBlock(b, f); //移动文件
    volumeMap.put(b, new DatanodeBlockInfo(v, dest));
    ongoingCreates.remove(b);
}
```

**注意**　我们把远程接口 DatanodeProtocol 的 blockReceived() 方法调用称为向名字节点提交数据块，而这里的提交数据块，显然用于向文件数据集 FSDataset 提交数据块，请读者注意区分。

FSDir 对象代表了“current”目录树中的某一个子目录，每个 FSDir 对象最多保存 ${dfs.datanode.numblocks} 个数据块文件，后续的文件会保持在该目录的子目录中，以保证文件检索的性能。FSDir.addBlock() 有两个重载实现。下面代码中的 addBlock()，在实现了文件移动的同时，当目录的数据块文件数达到限制时，会扩展目录，形成一个更宽的目录树结构：

```
private File addBlock(Block b, File src, boolean createOk,
                      boolean resetIdx) throws IOException {
   // 目录中还能保存数据块文件
   if (numBlocks < maxBlocksPerDir) {
      File dest = new File(dir, b.getBlockName());
      File metaData = getMetaFile(src, b);
      File newmeta = getMetaFile(dest, b);
      // 移动文件
      if (!metaData.renameTo(newmeta) || !src.renameTo(dest)) {
         ......
      }
      ......
      numBlocks += 1;
      return dest;
   }

   // 需要在子目录中保存数据块文件
   ......
   if (lastChildIdx >= 0 && children != null) {
      // 检查子目录是否有空间保存数据块
      for (int i = 0; i < children.length; i++) {
         int idx = (lastChildIdx + i) % children.length;
         File file = children[idx].addBlock(b, src, false, resetIdx);
         if (file != null) {
            lastChildIdx = idx;// 子目录有空间
            return file;
         }
      }
      lastChildIdx = -1;
   }
   ......
   // 需要创建子目录
   if (children == null || children.length == 0) {
      children = new FSDir[maxBlocksPerDir];
      for (int idx = 0; idx < maxBlocksPerDir; idx++) {
         children[idx] = new FSDir(new File(dir,
               DataStorage.BLOCK_SUBDIR_PREFIX + idx));
      }
   }
   // 在新创建的子目录中保存数据块文件
   lastChildIdx = random.nextInt(children.length);
   return children[lastChildIdx].addBlock(b, src, true, false);
}
```

#### 4. 其他要点

FSDataset.unfinalizeBlock() 和 FSDataset.invalidate() 都用于删除数据块，它们的应用场景不同，unfinalizeBlock() 用于删除复制不成功的数据块（该方法很简单，不再分析）；invalidate() 用于删除已经提交的数据块。FSDataset.invalidate() 可以一次性删除比较多的数据块，这样的删除操作会比较耗时间，为了减少对调用线程的占用，invalidate() 使用了异步

磁盘操作服务 FSDatasetAsyncDiskService（FSDataset 的成员变量 asyncDiskService 是该类的实例），降低方法的响应时间。这是一种非常典型的提高系统性能的方法，Java 提供了线程池执行器（ThreadPoolExecutor），可以很容易地实现这样的异步服务，在另外的线程中执行对时间不敏感的操作。由于篇幅限制，不详细分析 FSDatasetAsyncDiskService 的具体实现。FSDataset.invalidate() 代码如下：

```
public void invalidate(Block invalidBlks[]) throws IOException {
   boolean error = false;
   for (int i = 0; i < invalidBlks.length; i++) {
      ......
      // 异步方式删除文件，不要占用太多的线程时间
      asyncDiskService.deleteAsync(v, f, metaFile, dfsBytes,
            invalidBlks[i].toString());
   }
   ......
}
```

FSDataset.updateBlock()、FSDataset.truncateBlock() 和 FSDataset.startBlockRecovery() 等方法与数据块恢复相关，后面探讨这部分内容的时候再来分析它们的实现。

总的来说，FSDataset 是一个比较复杂的类，一方面它隐藏了数据节点存储空间复杂的结构；另一方面它需要根据数据节点上层逻辑对数据块的可能操作，维护数据块的正确状态。图 7-12 在图 7-2 的基础上，添加了引起状态转移的 FSDataset 方法，希望能够帮助读者更好地理解 FSDataset 的实现。

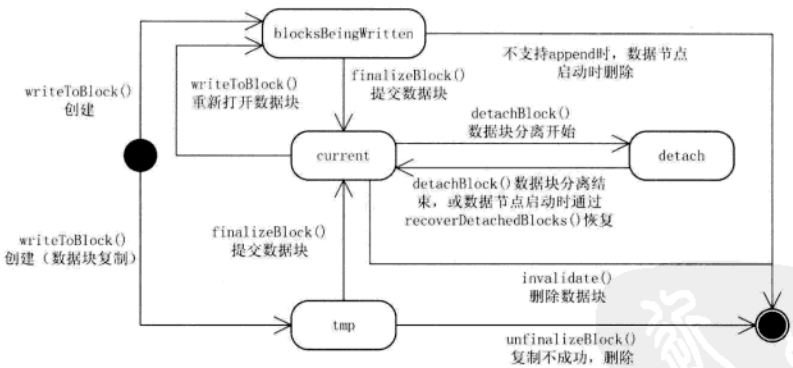



图 7-12　数据节点上数据块的状态及状态转移

## 7.2　流式接口的实现

数据节点通过数据节点存储 DataStorage 和文件系统数据集 FSDataset，将数据块的物理存储抽象为对象上的服务，流式接口就是构建在这个服务之上的、数据节点的另一个基本功能。

为了保证 HDFS 的设计目标，提供高吞吐的数据访问，数据节点使用基于 TCP 流的数

据访问接口，实现 HDFS 文件的读写。第 6 章已经分析了数据节点对外的流式接口，包括读数据、写数据、数据块替换、数据块拷贝和读数据块检验信息等，其中，数据读写的接口最为重要。本节考察数据节点如何实现上述接口。

### 7.2.1 DataXceiverServer 和 DataXceiver

数据节点的流式接口实现是典型的 TCP 服务器。回顾 Java 基本套接字的功能，JDK 为基本套接字准备了 java.net.Socket 和 java.net.ServerSocket，其中 ServerSocket 可在特定端口接收客户的连接请求。Java 程序一般通过构造 ServerSocket 对象，并将该对象绑定（bind）到某空闲端口，然后通过 accept() 方法监听此端口的入站连接。

客户端连接到服务器时，accept() 方法返回一个 Socket 对象，服务器使用该 Socket 对象和客户端进行交互，直到一方关闭连接。数据节点流式接口作为一个 TCP 服务器，以标准的方式实现了以上步骤。

在 DataNode.startDataNode() 方法中，数据节点创建 ServerSocket 对象并绑定到监听地址的监听端口上，监听地址由配置项 ${dfs.datanode.address}（在老版本的 Hadoop 中是 ${dfs.datanode.bindAddress}）指定，监听端口由 ${dfs.datanode.port} 配置。接下来，数据节点调用了 ServerSocket.setReceiveBufferSize() 方法，它设置 Socket 接收缓存区的大小为 128KB（默认值一般为 8KB），这是一个比较重要的参数，数据节点需要提供高吞吐的数据服务，也就需要比较大的接收缓存区。这个缓冲区大小设置适用于所有从 accept() 返回的 Socket 对象。

在 startDataNode() 中，还会为数据节点的流式接口服务线程建立线程组，创建 DataXceiverServer 服务器，并将该线程组的线程设置为守护线程。这里涉及了 Java 线程中的两个概念，线程组和守护线程。

线程组 ThreadGroup 表示一个线程的集合，通过线程组，Java 允许同时对一组线程进行操作。如通过 ThreadGroup.interrupt() 方法可以中断线程组中的所有线程，通过 setDaemon(true) 设置组中线程为守护线程等。

Java 线程分为用户线程和守护线程两类，守护线程是一种“在后台提供通用性支持”的线程，比如垃圾回收线程。它与用户线程的唯一差别是当 Java 虚拟机中所有的线程都是守护线程的时候，虚拟机就可以退出了；如果还有一个或以上的用户线程，虚拟机不会退出。流式接口服务线程所在的线程组设置为守护线程，简化了数据节点对这些线程的管理。如代码所示：

```
void startDataNode(……) throws IOException {
   ……
   ServerSocket ss;
   if(secureResources == null) {
      ss = (socketWriteTimeout > 0) ?
            ServerSocketChannel.open().socket() : new ServerSocket();
      Server.bind(ss, socAddr, 0);
   }
   ……
```

```
    ss.setReceiveBufferSize(DEFAULT_DATA_SOCKET_SIZE);
    tmpPort = ss.getLocalPort();
    ……
    this.threadGroup = new ThreadGroup("dataXceiverServer");
    this.dataXceiverServer = new Daemon(threadGroup,
          new DataXceiverServer(ss, conf, this));
    this.threadGroup.setDaemon(true); // 设置线程组中线程为守护线程
……
}
```

DataNode.startDataNode() 创建的 DataXceiverServer 实现了 accept() 循环，它的实现有如下要点。

成员变量 childSockets 包含了所有打开的用于数据传输的 Socket，这些 Socket 被 DataXceiver 对象使用；成员变量 maxXceiverCount，是数据节点流式接口能够支持的最大客户数，它由配置项 ${dfs.datanode.max.xcievers} 指定，默认值是 256，在一个繁忙的集群上，应该适当提高该数值。

DataXceiverServer.run() 的 accept() 调用会阻塞等待客户端的连接，如果有新的服务请求，服务器会创建一个新的线程，即创建一个 DataXceiver 对象，服务客户。这里，DataXceiverServer 使用了一客户一线程的模式，为每一个连接创建一个新线程，并由该线程和客户交互，DataXceiverServer 的主循环只是简单地通过 accept() 监听连接请求。这种模式，非常适合数据节点的流式接口，有利于批量处理数据，提高数据的吞吐量。

accept() 方法包含在 try/catch 语句块中，其中，捕获异常一共分 4 种情况，执行不同的逻辑。需要注意的是异常 AsynchronousCloseException，当通过 DataXceiverServer.kill() 关闭服务器时，通常都会触发这个异常。一般来说，结束 DataXceiverServer 服务器的方法是给标志位 datanode.shouldRun 赋值 false，run() 方法的循环会检查这个标记位，但即使设置了标志位，如果线程处于 accept() 方法中，还是会阻塞一段比较长的时间。这时，通过 kill() 方法关闭监听的 Socket，可以立即中断 accept() 方法的等待，让它抛出 AsynchronousCloseException 异常，迅速结束 DataXceiverServer 的服务。DataXceiverServer.kill() 在关闭监听 Socket 后，还会循环关闭已经打开的，被 DataXceiver 对象使用的 Socket 对象，以迅速关闭整个数据节点流式接口。代码如下：

```
class DataXceiverServer implements Runnable, FSConstants {
    ……
    // 保存了所有打开的用于数据传输的 Socket
    Map<Socket, Socket> childSockets =
          Collections.synchronizedMap(new HashMap<Socket, Socket>());
    ……
    int maxXceiverCount = MAX_XCEIVER_COUNT;
    ……
    DataXceiverServer(ServerSocket ss, Configuration conf, ……) {
        ……
        this.maxXceiverCount =
              conf.getInt("dfs.datanode.max.xcievers",……);
```

```
        ......
    }
    ......
    public void run() {
        while (datanode.shouldRun) {
            try {
                Socket s = ss.accept();
                new Daemon(datanode.threadGroup,
                           new DataXceiver(s, datanode, this)).start();
            } catch (SocketTimeoutException ignored) {
                // 可以忽略
            } catch (AsynchronousCloseException ace) {
                //Socket 被其他线程关闭，退出
                datanode.shouldRun = false;
            } catch (IOException ie) {
                // 忽略
            } catch (Throwable te) {
                // 其他情况，退出
                datanode.shouldRun = false;
            }
        }
        try {
            ss.close();
        }
        ......
    }

    void kill() {
        // 调用 kill 前，必须将 shouldRun 设置为 false
        assert datanode.shouldRun == false : " ......";
        try {
            this.ss.close();
        }
        ......
        // 关闭所有打开的，用于数据传输的 Socket
        synchronized (childSockets) {
            for (Iterator<Socket> it = ......) {
                Socket thissock = it.next();
                try {
                    thissock.close();
                }
                ......
            }
        }
    }
}
```

DataXceiverServer 只处理客户端的连接请求，实际的请求处理和数据交换都交由 DataXceiver 处理。DataXceiver 对象拥有自己独立的线程，该 DataXceiver 对象和它拥有的线程只处理一个客户端请求。DataXceiver 实现了 Runnable 接口，在它的 run() 方法里，

DataXceiver 会执行一些流式接口共有的操作，然后根据请求码分别调用不同的 DataXceiver 成员方法。run() 方法代码如下：

```
public void run() {
   DataInputStream in = null;
   try {
      in = new DataInputStream(new BufferedInputStream(……);
      short version = in.readShort();
      if (version != DataTransferProtocol.DATA_TRANSFER_VERSION) {
         throw new IOException("Version Mismatch");
      }
      ……
      byte op = in.readByte();
      //是否超过我们能支持的最大客户数
      int curXceiverCount = datanode.getXceiverCount();
      if (curXceiverCount > dataXceiverServer.maxXceiverCount) {
         throw new IOException(……);//超过，抛出异常
      }
      ……
      switch (op) {
      case DataTransferProtocol.OP_READ_BLOCK:
         readBlock(in);
         ……
         break;
   case DataTransferProtocol.OP_WRITE_BLOCK:
      ……
   } finally {
      ……
      IOUtils.closeStream(in);
      IOUtils.closeSocket(s);
      dataXceiverServer.childSockets.remove(s);
   }
}
```

DataXceiver.run() 首先创建输入流，然后进行版本检查。第 6 章分析了数据节点的流式接口，所有的请求帧，第一个字段都是版本号，所以，DataXceiver 能够在 run() 方法中统一处理请求的版本号。版本检查失败会抛出异常，并执行最后的清理工作：关闭输入流和 Socket。方法 run() 进行的第二项检查是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。经过这两项检查后，DataXceiver.run() 读入请求码，并根据请求码，调用相应的方法，如读数据块请求会由 readBlock() 方法进行后续处理。

DataXceiverServer 和 DataXceiver 实现了数据节点流式接口，它们采用一客户一线程的方式，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。

## 7.2.2　读数据

### 1. 准备和清理

客户端读取 HDFS 文件是通过操作码为 81 的流式接口进行的。读请求中包含如下字段：

❑blockId（数据块 ID）：要读取的数据块标识，数据节点通过它定位数据块。

❑generationStamp（数据块版本号）：用于进行版本检查，防止读取错误的数据。

❑startOffset（偏移量）：要读取数据位于数据块中的位置。

❑length（数据长度）：客户端需要读取的数据长度。

❑clientName（客户端名字）：发起读请求的客户端名字。

❑accessToken（访问令牌）：与安全特性相关，不讨论。

DataXceiver.readBlock() 给出了数据节点读数据流式接口实现的框架。方法的开始部分，会通过 Socket 连接的输入流，读取上述请求信息并构造一个数据块发送器 BlockSender 对象，然后，通过该对象发送数据，数据发送完毕后，方法执行一些清理工作。

BlockSender 的构造函数会进行一系列的检查，这些检查都通过以后，才会成功创建对象，否则，通过异常通知 readBlock() 方法，并由该方法返回错误码给客户端，并结束这次请求。如果 readBlock() 方法成功构造了 BlockSender 对象，它会给请求者返回一个成功应答，然后由 BlockSender.sendBlock() 发送客户端要读取的内容，代码如下：

```
private void readBlock(DataInputStream in) throws IOException {
   ......
   BlockSender blockSender = null;
   try {
      try {
         blockSender = new BlockSender(block, startOffset, length, true,
               true, false, datanode, clientTraceFmt);
      } catch (IOException e) {// 构造 BlockSender 时出现问题，读请求有错误
         out.writeShort(DataTransferProtocol.OP_STATUS_ERROR);
         throw e;
      }

      out.writeShort(DataTransferProtocol.OP_STATUS_SUCCESS); // 成功应答

      long read=blockSender.sendBlock(out, baseStream, null);// 发送数据

      if (blockSender.isBlockReadFully()) {
         // 客户端的校验是否成功？这样一个可选的客户端响应
         try {
            if(in.readShort()==Data……Protocol.OP_STATUS_CHECKSUM_OK
                  && datanode.blockScanner != null) {
               datanode.blockScanner.verifiedByClient(block);
            }
         } catch (IOException ignored) {
         }
   ......
}
```

readBlock() 方法还有一个需要提及的地方，在上述代码的最后部分，如果客户端成功读取并校验数据，会发送一个附加的响应码 OP_STATUS_CHECKSUM_OK，通知数据节点。如果数据节点发送了完整的一个数据块，这时，数据节点可以根据这个响应码，通知数据块

扫描器，让扫描器标记该数据块为客户端校验成功。数据节点使用数据块扫描器定期扫描数据块，以期尽快发现数据块错误，保证节点保存数据的正确性。数据块扫描需要读入数据块的数据和校验信息文件，并做检查，是一个比较耗资源的过程，如果客户端已经进行了这样的校验，数据节点就可以省略重复的工作，以减轻系统负载。

**注意**　客户端只上报成功校验的信息，如果读取过程出现错误，客户端不需要通知，也不允许通知数据节点，而是汇报给名字节点，再由名字节点决定下一步的动作。

数据块发送器完成读数据请求的大部分工作，包括：准备、发送读请求应答头、发送应答数据包和清理等。

准备工作主要由 BlockSender 的构造函数完成，在为一系列成员变量赋值后，构造函数开始准备数据块的校验信息，打开校验信息文件，并从文件中获取校验方法、校验块大小（它们保存在校验信息文件的头部），涉及的 BlockSender 构造函数代码如下：

```
BlockSender(Block block, long startOffset, long length,
        boolean corruptChecksumOk, boolean chunkOffsetOK,
        boolean verifyChecksum, DataNode datanode, String clientTraceFmt)
        throws IOException {
    ......
    endOffset = blockLength;
    ......
    // 应答数据在数据块中的开始位置
    offset = (startOffset - (startOffset % bytesPerChecksum));
    if (length >= 0) {
        long tmpLen = startOffset + length;
        if (tmpLen % bytesPerChecksum != 0) {
            // 用户读取数据的结束位置
            tmpLen += (bytesPerChecksum - tmpLen % bytesPerChecksum);
        }
        if (tmpLen < endOffset) {
            endOffset = tmpLen;
        }
    }

    // 设置读校验信息文件的位置信息
    if (offset > 0) {
        long checksumSkip = (offset / bytesPerChecksum) * checksumSize;
        if (checksumSkip > 0) {
            // 跳过不需要的部分
            IOUtils.skipFully(checksumIn, checksumSkip);
        }
    }
    ......
    // 打开数据块文件的输入流
    blockIn = datanode.data.getBlockInputStream(block, offset);
    ......
}
```

这部分代码决定了我们需要从数据块文件和校验信息文件中读入哪些数据。以数据块文件为例，读请求中提供了偏移量 startOffset 和数据长度 length 两个参数，但由于校验信息是按块组织的，为了让客户端能够进行数据校验，必须返回包含用户读取数据的所有块，如图 7-13 所示。

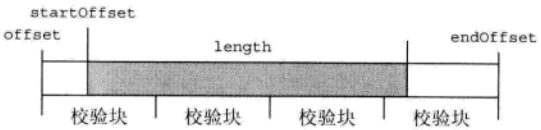


图 7-13 校验块读取范围计算

用户读取数据从位置 startOffset 开始，但它往往不会落在某一个校验块的起始，需要根据 startOffset 计算读取开始位置 offset。同样，用户读取数据的结束位置也不一定和某一个校验块的结束对齐，BlockSender 需要计算发送给用户数据的结束点 endOffset。这里还有一个异常情况需要处理，即用户请求读取的数据结束点（startOffset+length）越过了数据块的尾部，这时，只能将数据结束点设置为数据块的长度 blockLength。

相应的，校验信息文件也需要计算读取的范围，请参照代码注释。完成上述计算后，BlockSender 的构造函数就可以通过 FSDataset 提供的方法打开数据块文件的输入流。至此，读数据的准备工作已经完成。

数据块发送器的清理工作由 close() 完成，它的实现非常简单：关闭可能打开的到数据块文件的输入流和校验信息文件的输入流。

### 2.“零拷贝”数据传输

数据节点是一个 I/O 密集型 Java 应用，为了充分利用 Java NIO 带来的性能提升，BlockSender 支持两种数据发送方式：普通方式和 NIO 方式。普通方式使用基于 Java 流的 API，实现数据节点“数据读”流式接口，NIO 方式则利用了 Java NIO 中的 transferTo() 方法，以零拷贝的数据传输高效地实现了相同的接口。

客户端读取数据块数据时，数据节点上执行的主要操作是：将数据从保存在磁盘上的文件中读出，然后发送到客户端的套接字上。这种实现很简单，但效率其实非常低：首先应用通过 Linux 内核读出文件数据，这些数据需要跨越内核，拷贝到实现为 Java 应用的数据节点中，然后，数据节点需要再次将数据推送回内核，通过 Linux 内核写往套接字。实际上，应用程序在这里担当了一个不怎么高效的中介角色，将磁盘文件的数据转入套接字中。如果能够消除不必要的数据拷贝，要求内核直接将数据从磁盘文件拷贝到套接字，而无需通过应用程序，这将大大地提高应用程序的性能，而且还减少了内核与用户模式的上下文切换。后面的这种实现称为“零拷贝”，Java 通过 java.nio.channels.FileChannel 中的 transferTo() 方法，在 Linux（和 UNIX）系统上支持零拷贝 I/O 操作，transferTo() 方法用于直接将字节从它被调用的通道[⊖]上传输到另外一个可写字节通道上，数据无需经过应用程序。

图 7-14 对比了这两种实现方式的差别。

⊖ 在第 4 章中我们知道，通道表示到实体，如硬件设备、文件、网络套接字或可以执行一个或多个不同 I/O 操作（如读取或写入）的程序组件的开放的连接。

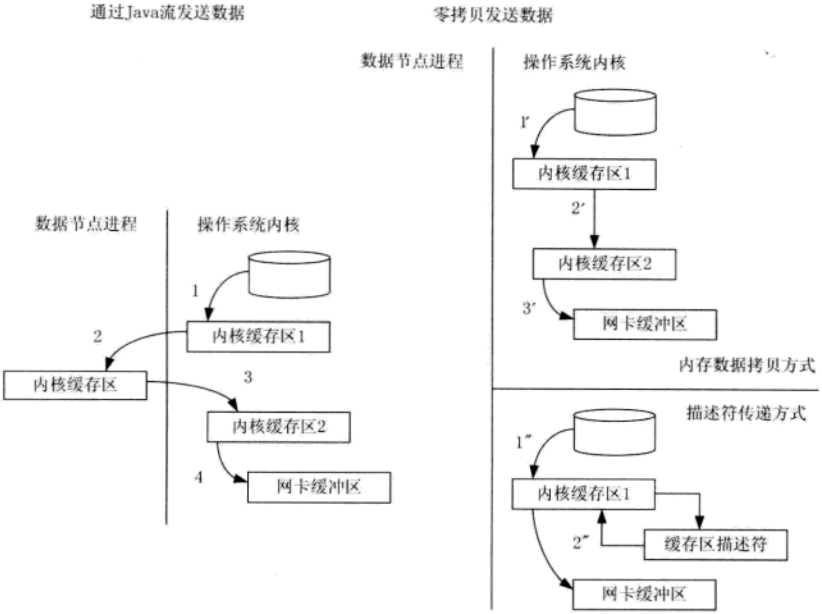



图 7-14　客户端读数据块数据时的缓冲区拷贝

图的左边是传统的实现，通过输入流，数据节点将磁盘中的数据读入到用户缓存区。其中，从磁盘读取数据需要使用到内核缓冲区，读方法会将数据从磁盘中读入内核缓冲区（步骤 1），然后从内核缓冲区中拷贝到用户缓冲区中（步骤 2）。这样一次读数据操作，需要引发一次从用户模式到内核模式，和一次内核模式到用户模式的上下文切换。另外，上述涉及的数据拷贝，步骤 1 发生在外设（磁盘）和内存间，一般由直接内存存取 DMA（Direct Memory Access）引擎执行，步骤 2 的数据拷贝发生在内存中，由 CPU 执行。调用结束后，数据节点会将数据通过 SocketOutputStream 的 write() 方法发送往客户端。

该调用引发了从用户模式到内核模式的上下文切换，数据被第三次拷贝，并被再次放置在内核缓冲区中。这次拷贝和步骤 2 的拷贝一样，由 CPU 执行。注意，该缓冲区和第一次从磁盘中读数据时使用的缓冲区不同，它与目标套接字相关联。写操作返回，触发第四次上下文切换，系统再次从内核模式切换回用户模式。同时，第四次数据拷贝，即内核缓冲区的数据到网卡拷贝，会独立地、异步地发生（由 DMA 引擎执行，所以是一个独立于 CPU，和 CPU 异步的数据拷贝）。最终，网卡上的数据被发送到客户端，由客户端网卡的接收缓冲区接收。

再次检查上述场景，读者会注意到第二次和第三次数据拷贝其实是多余的。应用程序只是缓存数据并将数据传回到套接字，没有其他的作用，数据可以直接从读取缓冲区传输到套接字缓冲区。FileChannel.transferTo() 方法实现这个操作，它的声明如下：

```
public abstract long transferTo(long position, long count,
```

```
        WritableByteChannel target)throws IOException;

    public void transferToFully(FileChannel fileCh, long position, int count)
        throws IOException;
```

该方法将读取通道对象所在的文件中给定位置（由参数 position 指定）处开始的 count 个字节，并将其写入目标通道 target 中。transferToFully() 则是 SocketOutputStream 中的方法，它可用于将文件通道 fileCh 中，由位置 position 开始的 count 个字节发送到 SocketOutputStream 对象中，数据节点的实现中，使用了 SocketOutputStream.transferToFully() 方法。

图 7-14 的右侧是使用零拷贝时系统的情景，分两种情况。右上方描述了需要一次内存拷贝的情形，transferTo() 方法让 DMA 引擎将文件内容拷贝到一个读取缓冲区，然后由内核将数据拷贝到与输出套接字相关联的内核缓冲区，即步骤 2'。对比传统方法，这里的改进包括：控制流上下文切换的次数从 4 次减少到了 2 次；数据复制的次数从 4 次减少到 3 次（其中只有一次涉及了 CPU）。如果底层网络接口卡支持收集操作，就可以进一步减少内核的数据复制，实现图右下角的描述符传递方式。支持收集操作的网络接口，不要求待传输的数据连续存储，数据可以分散存储在不同的位置上。这样一来，从文件中读出的数据就根本不需要被拷贝到 Socket 缓冲区中，而只是需要将缓冲区描述符（缓冲区中建立的与数据包相关结构）传递到网络协议栈中去，然后通过 DMA 收集拷贝功能将所有的数据结合成一个网络数据包，发送到网络上。描述符传递方式也就是 Linux 中众所周知的零拷贝技术，这种方法不但减少了因读写操作带来的多次上下文切换开销，同时也减少了处理过程中需要的数据副本个数，而且 CPU 和传输的数据没有关联，不需要参与到数据拷贝的过程中去。对于用户应用程序来说，代码没有任何改变，还是调用 FileChannel.transferTo() 方法。

BlockSender 使用了 NIO 的 transferTo() 方法，“零拷贝”进行数据高效传输，使得数据块的数据不经过数据节点，带来的一个问题是：数据节点失去了在客户端读取数据的过程中对数据进行校验的能力。所以，除了上述方法，BlockSender 也支持结合数据校验的数据传输，它被应用于数据块扫描器中。另一个解决方案是让客户端对数据进行校验，并上报校验的结果，在 DataXceiver.readBlock() 的清理动作中，数据节点会接收客户端的附加响应码，以获取客户端的校验结果。

### 3. 数据发送

读请求的应答包括应答头和多个应答数据包，主要实现在 BlockSender 的 sendBlock() 方法和 sendChunks() 方法中。

读应答的应答头包括数据校验的类型、校验块大小和一个可选的偏移量，与校验相关的信息已经在数据块发送器的构造函数中，从校验信息文件中获得，现在只需要将这些信息发送到客户端即可。应答头中偏移量是一个可选参数，BlockSender 不但被用于支持客户端读数据，也用于数据块复制中。数据块复制，由于是对整个数据块进行的操作，也就不需要提供偏移量。对于一般的客户端读取数据，偏移量是个必选参数。BlockSender.sendBlock() 的

第一部分代码，就是用于发送上述应答头。

BlockSender.sendBlock() 的第二部分代码，是根据系统的缓冲区大小配置，计算一次能够发送多少校验块的数据，并分配工作缓冲区。如上一小节讨论，sendBlock() 支持两种数据发送方式，我们以“零拷贝”方式，展开后续的分析。

数据节点读应答的数据包包头包括：

- packageLen（包长度）：从数据长度字段开始的包长度，包括数据长度字段（4 字节）、校验数据和数据块数据的长度。
- offset（偏移量）：应答数据位于数据块中的起始位置。
- seqno（顺序号）：该数据包在应答中的顺序号。
- tail（最后应答包标志）：1 字节，该数据包是否是应答的最后一个应答包。
- length（数据长度）：包中包含的实际数据长度。

缓冲区大小 pktSize 的初值是 DataNode.PKT_HEADER_LEN（除数据长度以外其他字段的长度和）和 length 字段大小（即 SIZE_OF_INTEGER）的和。无论是否使用“零拷贝”的数据传输方式，应答包中还必须包含校验数据，所以，缓冲区中必然也需要包含这部分内容。需要解释的是：如同普通发送方式那样，我们还需要在缓冲区中为数据块内容预留空间，以便在数据块内容发生变化的时候，重新计算校验和。数据块内容变化其实是一种竞争条件，出现在某客户端在写数据的同时，另一个客户端在读取数据[⊖]，这时，由于数据块文件和校验信息文件的写入有先后顺序，在校验信息还没完全写入的情况下，根据校验信息文件中的数据对数据块文件进行校验，会出现错误。为了避免这种误判，当发现有可能存在上述竞争条件时，将根据数据块文件中的数据，为应答数据包计算校验信息。

通过上面的分析，我们了解了缓冲区中可能包含的内容，下面的代码也就很好理解了。其中，BUFFER_SIZE 由配置项 ${io.file.buffer.size} 指定，默认值是 4096，即 4KB 字节：

```
int maxChunksPerPacket;
int pktSize = DataNode.PKT_HEADER_LEN + SIZE_OF_INTEGER;

if (transferToAllowed && !verifyChecksum
        && baseStream instanceof SocketOutputStream
        && blockIn instanceof FileInputStream) {
    ......
    maxChunksPerPacket = (Math.max(BUFFER_SIZE,
            MIN_BUFFER_WITH_TRANSFERTO)
            + bytesPerChecksum - 1)/ bytesPerChecksum;

    // 如上面讨论，缓冲区需要能够执行一次普通传输
    pktSize += (bytesPerChecksum + checksumSize)* maxChunksPerPacket;
}
ByteBuffer pktBuf = ByteBuffer.allocate(pktSize);  // 分配空间
```

接下来，sendBlock() 循环调用 sendChunks() 方法发送应答数据包。

⊖ 读者如果想进一步了解社区对这个问题的讨论，可以参考 JIRA HDFS 1057，地址 https://issues.apache.org/jira/browse/HDFS-1057。

BlockSender.sendChunks() 比较简单，实现分为两部分：计算并将应答包的头部，即上述包长度、偏移量等五个字段写入缓冲区，并获取校验数据；接下来，sendChunks() 将根据情况处理数据块数据并发送缓冲区，代码如下：

```
private int sendChunks(ByteBuffer pkt, int maxChunks, OutputStream out)
      throws IOException {
   try {
      // 使用 transferTo() 需要获得 Socket 的输出流和输入文件通道
      SocketOutputStream sockOut = (SocketOutputStream) out;
      FileChannel fileChannel = ((FileInputStream) blockIn).getChannel();

      if (memoizedBlock.hasBlockChanged(len)) {
         // 文件发生变化，假定出现读写竞争
         fileChannel.position(blockInPosition);
         IOUtils.readFileChannelFully(fileChannel, buf, dataOff, len);

         ChecksumUtil.updateChunkChecksum(buf, checksumOff, dataOff,
               len, checksum);
         sockOut.write(buf, 0, dataOff + len);
      } else {
         // 写数据包头和校验信息
         sockOut.write(buf, 0, dataOff);
         // 使用 transferToFully() "零拷贝" 发送数据
         sockOut.transferToFully(fileChannel, blockInPosition, len);
      }

      blockInPosition += len;

   } catch (IOException e) {
   ......
}
```

上面的代码是使用“零拷贝”方式发送数据的 sendChunks() 分支，它考虑上面提到的数据块读写竞争的情况：如果发现有竞争存在，即 memoizedBlock.hasBlockChanged() 方法返回真，则在数据缓冲区中，使用 ChecksumUtil.updateChunkChecksum() 重新计算数据的校验和，然后发送数据；如果数据块没有发生变化，在往输出流中写入数据包头和校验信息后，使用 SocketOutputStream.transferToFully() “零拷贝”发送数据块数据。

BlockSender.sendBlock() 在所有应答数据包发送完毕以后，往客户端的输出流中写入 0，以结束一次读操作。

#### 4. 读写数据的速度控制

在数据节点上，系统的磁盘 I/O 和网络 I/O 往往会成为系统的瓶颈。客户端读数据的时候需要从磁盘获取数据、数据节点间根据名字节点指令进行的数据块复制也需要从磁盘读入数据，后面将要介绍的数据块扫描器在周期性地验证数据节点存储的数据块时，也需要读磁盘上的数据。另一方面，数据节点在客户端读写数据块时，均需要通过网络传输数据；HDFS 系统的正常工作和负载均衡，有时需要在数据节点间移动数据块。这些各不相同的任

务，它们共享磁盘和网络带宽，所以，在必要的时候，必须对它们使用磁盘和网络进行一定的速度控制，特别是数据块扫描器和节点间数据块的移动，不能影响系统的对外服务，以保证系统的吞吐量和延迟。节流器 BlockTransferThrottler 正是为了这样的目的，引入到数据节点中，它提供了一个非常简单的流控机制，并应用于数据节点的多个地方。

BlockTransferThrottler 的使用很简单，成功构造对象后，在需要执行速度控制的地方，简单调用 BlockTransferThrottler.throttle() 即可。以 BlockSender.sendChunks() 为例，如果对象工作时，节流器对象不为空，就会调用节流器对象的 throttle() 方法，对数据发送限制速度。

```
private int sendChunks(ByteBuffer pkt, int maxChunks, OutputStream out)
      throws IOException {
   ......
   if (throttler != null) {
      throttler.throttle(packetLen); // 调用节流器对象的 throttle() 方法
   }

   return len;
}
```

节流器的实现非常简单，类图如图 7-15 所示。成员变量 period（单位：毫秒）和 bytesPerPeriod 一起定义了节流器的流速，通过公式“(1000×bytesPerPeriod)/period”，就可以计算出每秒通过节流器的数据量。

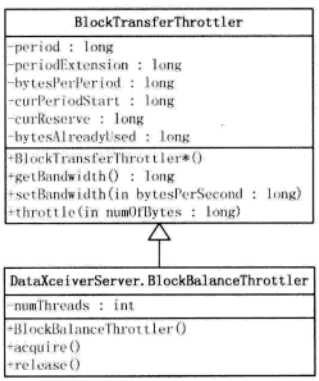



图 7-15　客户端读数据块数据时的缓冲区拷贝

BlockTransferThrottler 中最重要的 throttle() 方法代码如下：

```
synchronized void throttle(long numOfBytes) {
   ......
   curReserve -= numOfBytes;
   ......
   while (curReserve <= 0) {
      long now = System.currentTimeMillis();
      long curPeriodEnd = curPeriodStart + period;

      if ( now < curPeriodEnd ) {
         // 等待检查周期结束，curReserve 在结束时会增加
         try {
            wait( curPeriodEnd - now );
         } catch (InterruptedException ignored) {}
      } else if ( now <  (curPeriodStart + periodExtension)) {
         curPeriodStart = curPeriodEnd;
         curReserve += bytesPerPeriod;  // 相当于增加剩余请求量
      } else {
         // 很长时间没有使用节流器，重新启动节流器
         ......
      }
   }
   ......
}
```

其中，成员变量 curReserve 保存着当前周期还能使用的剩余请求量。如果当前请求量比剩余请求量小，显然，throttle() 中的 while 循环条件不成立，方法立即返回，调用者可以立即发送 / 接收数据。如果剩余请求量不能满足当前请求量，则节流器会循环等待检查周期结束，并在每个周期结束时，增加相应的剩余请求量，直到当前请求量得到满足，才退出循环并返回。注意，所有的 BlockTransferThrottler 方法都是 synchronized 方法，它可以被多个线程使用，而且不需要额外的同步操作。

应该说，BlockTransferThrottler 实现的速度控制方法非常简陋，它只保证请求者发送 / 接收数据在某一个时间段的平均速度不超过指定阈值，但是不能保证请求者在指定的时间段内均匀地发送 / 接收数据。经过一段时间等待的请求者，在 throttle() 调用结束后，往往会突然产生大量的 I/O 操作，影响其他共享了磁盘和网络带宽的任务。

在 DataXceiverServer 中还定义了另外一个节流器 BlockBalanceThrottler，它在控制 I/O 速度的同时，还控制了共享节流器的实例数，保证了这些实例运行时能得到的带宽。BlockBalanceThrottler 专门为平衡器 Balancer 设计，目前，它允许的最大实例数是 5，该常量定义在 Balancer.MAX_NUM_CONCURRENT_MOVES 中。也就是说，在一个数据节点上，包括发送数据和接收数据，平衡器 Balancer 最多能拥有五个工作任务。BlockBalanceThrottler 对实例数的控制是通过 acquire() 和 release() 方法对实现的，而对速度的控制，还是通过 BlockTransferThrottler.throttle() 方法执行。

### 7.2.3 写数据

#### 1. 准备和清理

流式接口的写数据实现远比读数据复杂。客户端写 HDFS 文件数据的操作码为 80，请求包含如下主要字段：

- ❑blockId（数据块 ID）：写数据的数据块标识，数据节点通过它定位数据块。
- ❑generationStamp（版本号）：数据块的版本号，用于进行版本检查。
- ❑pipelineSize（数据流管道的大小）：参与到写过程的所有数据节点的个数。
- ❑isRecovery（是否是数据恢复过程）：这个写操作是不是错误恢复过程中的一部分。
- ❑clientName（客户端名字）：发起写请求的客户端名字，可能为空。
- ❑hasSrcDataNode（源信息标记）：写请求是否携带源信息，如果是 true，则包含源信息。
- ❑srcDataNode（源信息，可选）：类型为 DatanodeInfo，包含发起写请求的数据节点信息。
- ❑numTargets（数据目标列表大小）：当前数据节点还有多少个下游数据推送目标。
- ❑targets（数据目标列表）：当前数据节点的下游数据推送目标列表。
- ❑accessToken（访问令牌）：与安全特性相关，不讨论。
- ❑checksum（数据校验信息）：类型为 DataChecksum，包含了后续写数据数据包的校验方式。

和 DataXceiver.readBlock() 类似，上述字段在 writeBlock() 入口中读取，并保存在对应

的方法变量中，然后，构造数据块接收器 BlockReceiver 对象，在 BlockReceiver 的构造函数中，会为写数据块和校验信息文件打开输出数据流，使用的是 FSDataset.writeToBlock() 方法，在完成一系列检查后，它返回到数据块文件和校验文件的输出流。这部分代码的异常处理值得一提，当打开数据流失败并抛出 IOException 后，catch 子句先执行资源清理，然后会检查这个异常是不是因为磁盘错误导致的。检测硬件故障是 Hadoop 分布式文件系统的一个主要特性，异常处理中的这段代码，能尽快地发现故障，代码如下：

```
BlockReceiver(Block block, DataInputStream in, String inAddr,
      String myAddr, boolean isRecovery, String clientName,
      DatanodeInfo srcDataNode, DataNode datanode) throws IOException{
   ......
   try {
      streams = datanode.data.writeToBlock(block, isRecovery,
            clientName == null || clientName.length() == 0);
      this.finalized = false;
      if (streams != null) {
         this.out = streams.dataOut;
         this.checksumOut = new DataOutputStream(……);
         // 如果是添加操作，将数据块从数据块扫描器中移除
         if (datanode.blockScanner != null && isRecovery) {
            datanode.blockScanner.deleteBlock(block);
         }
      }
   } catch (IOException ioe) {
      IOUtils.closeStream(this);
      cleanupBlock();
      // 是否是磁盘错误
      IOException cause = FSDataset.getCauseIfDiskError(ioe);
      ......
      if (cause != null) {
         ioe = cause;
         datanode.checkDiskError(ioe); // 检查磁盘是否出错
      }
      ......
   }
}
```

接下来，DataXceiver.readBlock() 将根据请求信息建立数据流管道。

数据流管道中，顺流的是 HDFS 的文件数据（如图 7-16 中粗箭头方向），而写操作的确认包会逆流而上，所以，这里需要两个 Socket 对象。其中，对象 s 用于和管道上游通信，它的输入和输出流分别是 in 和 replyOut；往下游的 Socket 对象是 mirrorSock，关联了输出流 mirrorOut 和输入流 mirrorIn。

如果当前数据节点不是数据管道的最末端，readBlock() 方法就会使用数据目标列表的第一项，建立到下一个数据节点的 Socket 连接，连接建立后，通过输出流 mirrorOut，往下一个数据节点发起写请求，除了数据目标列表大小和数据目标列表字段会相应变化以外，其他字段和从上游读到的请求信息是一致的（严格地说，这里不涉及请求的数据校验信息，即上

面讨论的接收发送字段只包含请求前面的 10 个字段）。

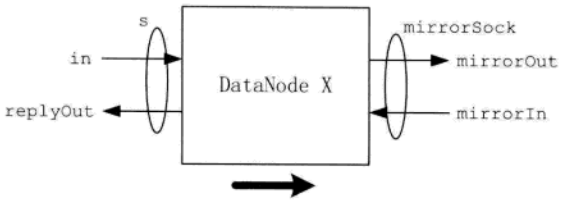



图 7-16 数据流管道中的 Socket 对象和流对象

往下一个数据节点的写请求发送以后，writeBlock() 会等待请求的应答，这是一个同步的过程。由于数据流中会有多个数据节点，所以建立数据流管道会花比较长的时间，这也是 HDFS 不适合用于低延迟数据访问场景的原因之一。第 6 章的讨论已经给出了应答包含的内容：返回码和附加信息，当返回码是 DataTransferProtocol.OP_STATUS_ERROR，即出错时，附加信息提供了流中第一个出错的数据节点地址信息（主机名和端口）。在 writeBlock() 方法中，相关的代码分布在多个处理步骤上，请参见如下片段。方法变量 mirrorInStatus 和 firstBadLink 用于保存返回码和附加信息，请注意它们的初值。writeBlock() 通过从流 mirrorIn 中读取返回码和附加信息，等待下游节点的应答。

对于某一个数据节点，往上游发送响应的时机有两个：

- 第一个是建立管道过程中，到下游的连接出现问题，在 IOException 的异常处理中，往上游发送出错应答，附加信息包含的出错点自然就是下一个数据节点。
- 另一个发送响应的地方是为写进行的准备工作完成后发送的应答，对于数据流中间节点，准备工作完成意味着从流 mirrorIn 中读到下游的应答；对于数据流末端的数据节点，只要 BlockReceiver 对象成功构造，准备工作就完成了。通过变量 mirrorInStatus 和 firstBadLink 的初值，writeBlock() 可以用统一的方式发送请求响应。

另一个要点是构造数据块接收器对象出现错误时，抛出异常的处理。这个异常不会被上面的异常处理捕获（它在对应 try 子句的外面），而是由更外层的异常处理接收，并在最后利用 finally 子句，直接关闭到上游的 Socket 连接，上游数据节点由于在流 mirrorIn 上等待应答，readShort() 方法这个时候会抛出 IOException 异常，从而判断下游数据节点出现问题。这种策略，使上游节点以一致的方式，处理建立数据流管道过程中的错误，但代价是只能了解到这个过程出现了错误，应答中没有附加的信息，提供详细的出错原因。代码如下：

```
private void writeBlock(DataInputStream in) throws IOException {
    ……
    String firstBadLink = ""; //应答附加信息，mirrorInStatus 是返回码
    short mirrorInStatus = (short) DataTransferProtocol.OP_STATUS_SUCCESS;
    try {
        blockReceiver = new BlockReceiver(……);
        ……// 下面代码位于 "if (targets.length > 0) {" 中
        try {
            // 前面有大量的代码，完成了上述往下一个数据节点写请求的任务
```

```
            blockReceiver.writeChecksumHeader(mirrorOut);   // 写数据校验信息
            mirrorOut.flush();// 数据刷出

            // 如果不是复制请求（即从客户端发起的写请求），从下游读取应答
            if (client.length() != 0) {
                mirrorInStatus = mirrorIn.readShort();
                firstBadLink = Text.readString(mirrorIn);
                ......
            }
        } catch (IOException e) {
            if (client.length() != 0) {
                // 在往下一个数据节点写请求的过程中出错，向上游写应答
                replyOut.writeShort((short).....OP_STATUS_ERROR);
                Text.writeString(replyOut, mirrorNode);
                replyOut.flush();
            }
            ......
        }

        if (client.length() != 0) {
            ......
            // 顺利完成各类初始化任务，写应答
            replyOut.writeShort(mirrorInStatus);
            Text.writeString(replyOut, firstBadLink);
            replyOut.flush();
        }
    ......
}
```

DataXceiver 委托 BlockReceiver.receiveBlock() 处理写数据的数据包，成功处理完这些数据包以后，接下来的清理工作有：调用 DataNode.notifyNamenodeReceivedBlock() 通知名字节点，也就是说，写数据操作成功执行后需要告知名字节点，名字节点会有一些后续处理，这一点是读数据过程中没有的（如果是客户端写数据，这个通知由下面介绍的 PacketResponder 线程完成，这里只适用于数据块复制请求场景）。最后，writeBlock() 会关闭到上下游的输入 / 输出流，完成一次写数据请求。

### 2. PacketResponder 线程

当 BlockReceiver 处理客户端的写数据请求时，方法 receiveBlock() 接收数据包，校验数据并保存到本地的数据块文件和校验信息文件中，如果节点处于数据流管道的中间，它还需要向下一个数据节点转发数据包。同时，数据节点还需要从下游接收数据包确认，并向上游转发。这里，涉及了图 7-15 中两个 Socket 输入流（即 in 和 mirrorIn）的读操作，为此，数据块接收器引入了 PacketResponder 线程，它和 BlockReceiver 所在的线程（下面称为 BlockReceiver 线程）一起工作，分别用于从下游接收应答和从上游接收数据。为什么需要两个线程呢？我们知道，从输入流中读取数据，如果流中有可读的数据，立即读取，如果没有，则会阻塞等待。如果只使用一个线程，轮流读取两个输入流，就会在这两个输入流间

引入耦合。客户端如果长时间不往数据节点发送数据，那么，就很可能阻塞了下游确认的接收；另一个极端是，虽然客户端往数据节点写入大量的数据，但由于处理过程正在等待mirrorIn 的输入，也就没有机会进行处理，从而影响了数据的吞吐。

PacketResponder 线程将两个输入流的处理过程分开，该线程从下游数据节点接收确认，并在合适的时候，往上游发送。这里的“合适”包括两个条件：

❑当前数据节点已经顺利处理完该数据包。

❑（数据节点处于管道的中间时）当前数据节点收到下游数据节点的数据包确认。

这两个条件都满足，意味着当前数据节点和数据流管道后续数据节点都完成了对某个数据包的处理。

由于当前节点由 BlockReceiver 线程处理数据包，所以，它必须将处理结果通过某种机制，通知到 PacketResponder 线程，并由 PacketResponder 线程进行进一步的处理。

理解了上述条件，PacketResponder 的实现就很好理解，代码如下：

```
class PacketResponder implements Runnable, FSConstants {
   //BlockReceiver 线程已经处理完的数据包信息
   private LinkedList<Packet> ackQueue = new LinkedList<Packet>();
   ......
   synchronized void enqueue(long seqno, boolean lastPacketInBlock) {
      if (running) {
         ......
         ackQueue.addLast(new Packet(seqno, lastPacketInBlock));
         notifyAll();
      }
   }
   ......
}
```

PacketResponder 中的成员变量 ackQueue，保存了 BlockReceiver 线程已经处理的写请求数据包。BlockReceiver .receivePacket() 方法每处理完一个数据包，就通过 PacketResponder.enqueue() 将对应信息（定义在内部类 BlockReceiver.Packet 中，包括数据包的序列号和是否是最后一个数据包两个字段）放入队列 ackQueue 中，队列 ackQueue 中的信息由 PacketResponder 的 run() 方法处理，这是一个典型的生产者 - 消费者模型。

PacketResponder.run() 是一个大方法，它的处理过程明显地分为两个部分：等待上述两个条件满足，以及条件满足后的处理。其中，第一部分需要通过 Java 的同步工具 wait()，等待 ackQueue 中的数据，这里 wait() 等待的是 enqueue() 方法中 notifyAll() 的通知。如果 ackQueue 有数据，则获取第一个记录，接下来，如果当前数据节点位于数据流管道的中间，那么，在流 mirrorIn 上读取下游的确认，如果顺利读取到下游的响应，表明第一步处理已经完成。代码如下：

```
public void run() {
   ......
   while (running && datanode.shouldRun && !lastPacketInBlock) {
      try {
```

```
        ……
        try {
            /* PipelineAck.UNKOWN_SEQNO 的值是 -2，这是一个特殊的值，用于标识
             * 数据节点从下游读取应答出现错误的情况，这个时候，我们不知道正确的应答
             * 是什么，所以使用 UNKOWN_SEQNO。
             */
            long expected = PipelineAck.UNKOWN_SEQNO;
            long seqno = PipelineAck.UNKOWN_SEQNO;

            PipelineAck ack = new PipelineAck();  // 第 6 章已经给出它的类图
            boolean localMirrorError = mirrorError;
            try {
                Packet pkt = null;
                synchronized (this) {
                    // 等待 BlockReceiver 线程完成数据包的处理
                    while (running &&……&& ackQueue.size() == 0) {
                        ……
                        wait();
                    }
                    ……
                    pkt = ackQueue.removeFirst(); // 获取第一个已处理包的信息
                    expected = pkt.seqno;
                    notifyAll();
                }
                //（如果当前数据节点不是流的最后一个节点）在输入流上等待应答
                if (numTargets > 0 && !localMirrorError) {
                    // 读入应答
                    ack.readFields(mirrorIn);
                    ……
                }
    ……
}
```

如果处理的是整个写请求最后一个数据包的确认，这时，需要执行如下附加步骤：关闭 PacketResponder 所属的数据块接收器对象，设置数据块长度，使用 FSDataset.finalizeBlock() 方法提交数据块，最后利用 notifyNamenodeReceivedBlock() 通知名字节点，本节点完成了一个数据块的接收。和读数据不同，客户端写数据的主要清理工作是由 packetResponder 线程完成，上述步骤提交了写请求，并释放处理过程中的相关资源。代码如下：

```
……
receiver.close();
block.setNumBytes(receiver.offsetInBlock);
datanode.data.finalizeBlock(block);
datanode.notifyNamenodeReceivedBlock(block,DataNode.EMPTY_DEL_HINT);
……
```

无论当前处理的是否是最后一个数据包，也无论当前数据节点是否是管道的最后一个节点，确认包都需要往上游发送，对应的处理过程在 PacketResponder.run() 的尾部，代码相对简单，构造确认包 PipelineAck 对象并设置相关属性，并通过输出流 replyOut，往上游节点

发送。代码如下：

```
public void run() {
   ……
   // 构造消息
   short[] replies = null;
   if (mirrorError) { // 有错误
      replies = new short[2];
      replies[0] = DataTransferProtocol.OP_STATUS_SUCCESS;
      replies[1] = DataTransferProtocol.OP_STATUS_ERROR;
   } else { // 构造成功应答
      short ackLen = numTargets == 0 ? 0 : ack.getNumOfReplies();
      replies = new short[1 + ackLen];
      replies[0] = DataTransferProtocol.OP_STATUS_SUCCESS;
      for (int i = 0; i < ackLen; i++) {
      replies[i + 1] = ack.getReply(i); // 加入下游应答
   }
}
PipelineAck replyAck = new PipelineAck(expected, replies);

replyAck.write(replyOut);// 往上游发送
replyOut.flush();
……
```

当客户端最终接收到确认包时，它可以断定数据流管道上的所有数据节点已经接收到对应的数据包。

### 3. 数据接收

客户端写往数据节点的数据由 BlockReceiver.receiveBlock() 接收，和 BlockSender 类似，receiveBlock() 将大部分处理工作交给 receivePacket()。代码如下：

```
void receiveBlock(……)throws IOException {
   ……
   if (clientName.length() > 0) {  // 是客户端发起的写请求，启动 PacketResponder
      responder = new Daemon(datanode.threadGroup,
            new PacketResponder(this, block, mirrIn, replyOut,
                  numTargets, Thread.currentThread()));
      responder.start(); // 创建并启动 PacketResponder
   }

   while (receivePacket() > 0) {
   }
   ……
}

private int receivePacket() throws IOException {
   int payloadLen = readNextPacket();
   ……
}
```

receivePacket() 方法一开始就调用 readNextPacket()，通过该方法最少读入一个写请求数据包。读入的数据放在类型为 ByteBuffer 的缓冲区 buf 中。ByteBuffer 是 java.nio.Buffer 的子类，Buffer（第 5 章介绍）代表一个有限容量的容器，是 Java NIO 的重要概念，为基于块的通道 I/O 提供了必要的基础设施。

readNextPacket() 的实现比较琐碎，需要保证缓冲区剩余空间可以读入整个数据包，必须考虑数据包的长度（必要时需要重新分配缓冲区）、并根据需要调整数据在缓冲区中的位置。由于 readNextPacket() 一次读入一个完整的数据包，receivePacket() 的后续处理就比较简单了。

BlockReceiver.receivePacket() 从缓冲区中读入数据包的包头信息，并赋值到对象的成员变量，接下来的准备动作是：调用 setBlockPosition() 方法，设置写数据时的文件位置。代码如下：

```
private int receivePacket() throws IOException {
    ......
    // 从缓冲区中读入数据包的包头信息
    ......
    setBlockPosition(offsetInBlock);
    ......
}
```

我们知道，写数据的时候其实需要同时写两个文件，数据块文件和它的校验信息文件，BlockReceiver.setBlockPosition() 会设置这两个文件的写位置，为后续持久化数据做准备。设置数据块文件的写位置比较简单，但校验信息文件的就相对复杂了，这又是一个和分块校验相关的调整。校验信息的计算需要数据以块的形式组织，但用户以流的形式写数据，如果写（追加）数据的开始点落在某个校验块的中间，如图 7-17 所示，需要在确定校验信息文件写位置的时候，计算不完整数据校验块（图中的检验块 n）的校验信息，为后续写校验数据做准备。

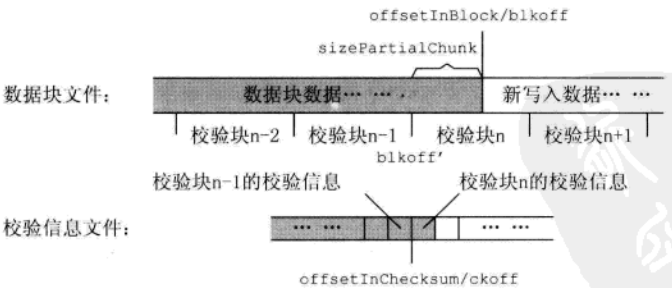



图 7-17　写数据时不完整数据校验块校验信息计算

setBlockPosition() 方法只有一个参数 offsetInBlock，即数据块文件的写位置，如果该位置不处于校验块的边界，就会调用 computePartialChunkCrc() 计算上述校验信息。该方法需要三个参数，进入方法时，前面两个参数 blkoff、ckoff 的值如上图，第三个参数

bytesPerChecksum 是校验块的大小。根据这些参数，首先可以计算出不完整数据块的长度 sizePartialChunk，然后，分配缓冲区读入数据，计算这些数据的校验和。校验和的结果保存在 BlockReceiver 的成员变量 partialCrc 中，后面会和新写入的数据一起，计算出新的校验和的值，并保持在校验信息文件的当前位置。注意，对于数据块文件，新写入的数据会追加在数据块文件的文件尾，但校验信息文件，如果对应的是一个不完整数据校验块，那么，文件的最后部分（从 ckoff 开始的地方）会被更新，而不能简单地直接进行追加。代码如下：

```
private void setBlockPosition(long offsetInBlock) throws IOException {
    ......
    if (datanode.data.getChannelPosition(block, streams)==offsetInBlock) {
        return; //如果当前写位置和打开文件位置一致，表明我们已经调整过了
    }
    ......
    if (offsetInBlock % bytesPerChecksum != 0) {
        ......
        computePartialChunkCrc(offsetInBlock, offsetInChecksum,
                bytesPerChecksum);
    }
    // 设置数据块文件和校验信息文件的写位置
    datanode.data.setChannelPosition(block, streams, offsetInBlock,
            offsetInChecksum);
    ......
}

private void computePartialChunkCrc(long blkoff, long ckoff,
        int bytesPerChecksum) throws IOException {
    int sizePartialChunk = (int) (blkoff % bytesPerChecksum);
    int checksumSize = checksum.getChecksumSize();
    blkoff = blkoff - sizePartialChunk;
    ......// 分配缓冲区并读入数据
    partialCrc = new CRC32();//partialCrc 的类型为 Checksum
    partialCrc.update(buf, 0, sizePartialChunk);
    ......
}
```

上述准备工作完成后，receivePacket() 把请求的数据包发送到下游数据节点，通过这种方式，写数据的数据包就会流经数据流管道上的所有数据节点。代码如下：

```
private int receivePacket() throws IOException {
    ......
    // 发送数据包到下游数据节点
    if (mirrorOut != null && !mirrorError) {
        try {
            mirrorOut.write(buf.array(), buf.position(), buf.remaining());
            mirrorOut.flush();
        ......
    }
    ......
}
```

接下来要提及的是数据长度为 0 的数据包，这是一种特殊的数据包，当客户端长时间没有输出数据时，就会发送这样的心跳包，用于维护客户端和数据流管道上的各个数据节点的 TCP 连接。代码如下：

```
private int receivePacket() throws IOException {
   ......
   if (len == 0) {//接收到一个心跳包
      LOG.debug("Receiving empty packet for block " + block);
   } else {
      ......
   }
}
```

receivePacket() 方法的后续工作，就是将请求包中的数据和校验和写文件，代码如下：

```
private int receivePacket() throws IOException {
   ......
   //是最后一个数据节点，对数据进行校验检查
   if (mirrorOut == null || clientName.length() == 0) {
      verifyChunks(pktBuf, dataOff, len, pktBuf, checksumOff);
   }

   try {
      if (!finalized) {
         out.write(pktBuf, dataOff, len);//写文件数据
         ......
         partialCrc.update(pktBuf, dataOff, len);
         ......
         checksumOut.write(buf);//写校验数据
         ......
      }
   }

   if (responder != null) {
         ((PacketResponder) responder.getRunnable()).enqueue(seqno,
            lastPacketInBlock);
   }
}
```

主要关注点包括：客户端写数据的时候，只有最后一个数据节点需要通过 verifyChunks() 方法对数据进行校验，数据流管道中的其他节点不需要执行这项检查。校验发现数据有问题，会上报名字节点，并抛出异常通知调用者 BlockReceiver.receivePacket()；数据包如果成功接收并顺利写入磁盘后，则需要将处理的结果，通过 PacketResponder.enqueue() 方法，发送给 PacketResponder 线程，并由该线程继续后面的处理。

BlockReceiver.receiveBlock() 循环调用 receivePacket()，处理这次写请求的所有数据包。当所有的数据接收完毕后，它进行下述清理工作：发送一个空的数据包到下游节点，通知后面的数据节点这次写操作结束；等待 PacketResponder 线程结束，线程结束，表明所有的响应已经发送给上游节点，同时，本节点的处理结果也已经通知名字节点；关闭文件，并更新

数据块文件的长度，然后，通过 FSDataset.finalizeBlock() 提交数据块。至此，写数据请求处理完毕。

#### 4. 处理结果上报名字节点

和读操作不同的是，写数据对 HDFS 文件的数据块进行了修改，所以，操作的处理结果需要上报到名字节点，名字节点后续会进行一些登记、清理工作。写数据操作的处理结果，很自然地分为成功和失败两种情况，在前面的讨论中我们知道，对于客户端写数据的场景，当 PacketResponder 顺利发送完最后一个数据包确认，并向 FSDataset 提交数据块以后，通过 DataNode.notifyNamenodeReceivedBlock() 通知名字节点，确认写数据块操作成功。失败的情况可能发生在 BlockReceiver.verifyChunks() 中，当对输入数据进行检验时，发现数据校验和和请求包的校验和不一致时，立即通知名字节点数据块发生损坏。

由于数据节点上的写操作比较频繁，为了减少到名字节点的数据块提交请求量，数据节点会将多个提交合并成一个请求。所以，notifyNamenodeReceivedBlock() 只是简单地在请求队列中插入记录。

下面的代码中，receivedBlockList 和 delHints 是确认信息队列，分别保存着提交时需要的信息：其中，receivedBlockList 保存着完成写操作的数据块信息；而 delHints 队列保存着相应的删除提示信息。在客户端写操作中，数据块提交时不需要删除提示，参数使用空串常量 DataNode.EMPTY_DEL_HINT。但是在数据块替换（下一节介绍）请求处理中，这个字段保存着替换数据块的源数据节点信息，替换操作执行成功后，数据节点可以根据这个字段，通过名字节点指令删除源数据节点上的数据块，以保证各数据节点的平衡工作。代码如下：

```
public class DataNode extends Configured implements InterDatanodeProtocol,
      ClientDatanodeProtocol, FSConstants, Runnable, DataNodeMXBean {
   ......
   private LinkedList<Block> receivedBlockList = new LinkedList<Block>();
   ......
   private LinkedList<String> delHints = new LinkedList<String>();
   ......
   protected void notifyNamenodeReceivedBlock(Block block,String delHint){
      ......
      synchronized (receivedBlockList) {
         synchronized (delHints) {
            receivedBlockList.add(block);
            delHints.add(delHint);
            receivedBlockList.notifyAll();
         }
      }
   }
```

上述两个队列中的信息由数据节点的服务线程读出，并形成 DatanodeProtocol.blockReceived() 需要的数组形式，通过 Hadoop 远程过程调用提交到名字节点。其中，blockReceived() 的第一个参数 dnRegistration 保存着当前数据节点标识，这样，名字节点就可以了解到数据块已经成功保存在哪个数据节点上。代码如下：

```
public class DataNode extends Configured implements InterDatanodeProtocol,
        ClientDatanodeProtocol, FSConstants, Runnable, DataNodeMXBean {
    ......
    public void offerService() throws Exception { // 在数据节点服务线程中执行
        ......
            Block[] blockArray = null;
            String[] delHintArray = null;
            ......//receivedBlockList 和 delHints 多线程共享，同步保护
            blockArray = receivedBlockList.toArray(new Block[numBlocks]);
            delHintArray = delHints.toArray(new String[numBlocks]);
            namenode.blockReceived(dnRegistration, blockArray, delHintArray);
            ......// 删除队列 receivedBlockList 和 delHints 中已经上报的数据项
        ......
```

数据检验出现错误是比较少见的情况，所以它的处理比较直接，校验出现问题，verifyChunks() 立即调用远程方法 DatanodeProtocol.reportBadBlocks() 通知名字节点删除数据块，并抛出异常，通知调用者。主要代码如下：

```
private void verifyChunks(byte[] dataBuf, int dataOff, int len,
                byte[] checksumBuf, int checksumOff) throws IOException {
    while (len > 0) {
        ......
        if (!checksum.compare(checksumBuf, checksumOff)) {  // 校验出现问题
            ......
            datanode.namenode.reportBadBlocks(new LocatedBlock[] { lb });
            ......
            throw new IOException(......)
            ......
```

### 5. 数据块恢复

DataXceiver 和 BlockReceiver 的实现中，并没有对写数据过程中出现的异常情况进行过多的处理，一般来说，关闭到上游节点的 Socket 连接，由上游节点检测错误并发送携带错误信息的确认包，是标准的处理方法。这样的设计，简化了数据节点写请求处理的实现，但把故障恢复工作转移给了客户端或者名字节点，当它们发现某次写操作出现错误以后，该怎么处理呢？本小节将以客户端检测到异常，主动发起一次数据块恢复为例，分析数据节点上的数据块恢复流程。

第 6 章介绍 HDFS 各个节点的 IPC 接口时了解到，数据节点为客户端提供了远程接口 ClientDatanodeProtocol，客户端写数据出现错误后，使用这个接口的 recoverBlock() 方法进行数据块恢复。客户端会在参与到写请求的数据节点中，选取一个节点作为恢复的主数据节点，建立到该节点的 IPC 连接，并调用 recoverBlock()。

ClientDatanodeProtocol.recoverBlock() 方法的声明如下：

```
LocatedBlock recoverBlock(Block block, boolean keepLength,
      DatanodeInfo[] targets) throws IOException;
```

该方法包括三个参数：

❑block：自然携带了被恢复数据块的信息。

❑keepLength：布尔型，用于恢复策略的选择。如果为 true，则只恢复长度和传入数据块长度相同的数据块（当然，这些数据块位于多个数据节点上）；否则，由主数据节点获取参与到恢复过程的各个数据节点上的数据块长度，计算最小值，并将这些节点上的数据块截断到该值；

❑targets：参与恢复过程的数据节点列表。方法的返回值类型是 LocatedBlock，通过 LocatedBlock 对象，客户端可以再次建立数据流管道，重新开始写数据。

无论 keepLength 的值是什么，数据块恢复的主数据节点都需要和 targets 列表中的数据节点联系，并使用远程接口 InterDatanodeProtocol 中的 startBlockRecovery() 和 updateBlock() 方法，应该说，InterDatanodeProtocol 接口的作用就是用于数据块恢复。其中，startBlockRecovery() 用于获得参与到数据块恢复过程的各个数据节点上的数据块信息，然后，主数据节点根据策略，通过 updateBlock() 通知各数据节点，将数据块恢复到一致状态。代码如下：

```
public interface InterDatanodeProtocol extends VersionedProtocol {
   ……
   BlockRecoveryInfo startBlockRecovery(Block block) throws IOException;

   void updateBlock(Block oldblock, Block newblock, boolean finalize)
         throws IOException;
}
```

DataNode 类实现了上述两个远程接口。其中，recoverBlock() 方法的具体实现在同名的私有方法 recoverBlock() 中。首先，主数据节点需要获得参与恢复过程的各个数据节点上数据块的信息。它通过循环，创建到其他数据节点的 InterDatanodeProtocol 远程接口实例，并调用接口上的 startBlockRecovery() 方法获得数据块的恢复信息，将这些信息保存到列表 blockRecords 中。代码如下：

```
private LocatedBlock recoverBlock(Block block, boolean keepLength,
      DatanodeInfo[] targets, boolean closeFile) throws IOException {
   ……
   // 恢复前后数据节点所在的系统没有重启的数据节点数量
   int rbwCount = 0;
   // 恢复前后数据节点发生重启，EXT3 日志重放，导致文件被截断的数据节点数量，
   // 后续不讨论这个特殊的场景
   int rwrCount = 0;

   List<BlockRecord> blockRecords = new ArrayList<BlockRecord>();
   for (DatanodeID id : datanodeids) {
      try {
         InterDatanodeProtocol datanode = dnRegistration.equals(id) ? this
               : DataNode.createInterDataNodeProtocolProxy(id,……);
         BlockRecoveryInfo info = datanode.startBlockRecovery(block);
         ……  // 处理 info 的异常情况
         blockRecords.add(new BlockRecord(id, datanode, info));
```

```
    ……
}
```

远程方法 startBlockRecovery() 简单地调用了 FSDataset 的同名方法。这两个方法的返回值都是 BlockRecoveryInfo 对象，它包含了数据块信息和标志位 wasRecoveredOnStartup，其中，数据块信息可以通过调用 FSDataset.getStoredBlock() 获取。接下来，FSDataset.startBlockRecovery() 会进入一个循环，通过 FSDataset.interruptAndJoinThreads() 中断数据块上可能的写操作线程。有了上面 DataXceiver 和 BlockReceiver 的分析，这段代码就比较好理解了：错误恢复时，该数据块的写数据线程，也就是 BlockReceiver 线程可能还没有结束，startBlockRecovery() 必须中断该线程并等待线程退出，保证后续的数据块恢复过程不会受 BlockReceiver 线程的影响。代码如下：

```
public BlockRecoveryInfo startBlockRecovery(long blockId)throws……{
    Block stored = getStoredBlock(blockId);
    ……
    while (true) {
        ……
        List<Thread> activeThreads = getActiveThreads(stored);
        if (activeThreads == null)
            break;
        if (interruptAndJoinThreads(activeThreads))
            break;
    }

    synchronized (this) {
        ActiveFile activeFile = ongoingCreates.get(stored);
        boolean isRecovery = (activeFile != null)
                && activeFile.wasRecoveredOnStartup;
        BlockRecoveryInfo info = new BlockRecoveryInfo(stored, isRecovery);
        ……
        validateBlockMetadata(stored);
        return info;
    }
}
```

FSDataset.startBlockRecovery() 接下来会计算 BlockRecoveryInfo 对象的 wasRecoveredOnStartup 字段，这个字段和 Linux Ext3 文件系统日志相关，Linux 系统重启，文件系统恢复过程中，非正常关闭的处于写入过程中的文件可能会被截断，wasRecoveredOnStartup 用于标记这种情况，这个场景的细节部分，后续就不再展开讨论了。

处于恢复状态的数据块文件和校验文件可能处于不一致的状态，startBlockRecovery() 返回前，会调用 validateBlockMetadata() 方法对数据块进行一次校验，校验通过后才会返回数据块的 BlockRecoveryInfo 对象。

数据块恢复的主数据节点通过远程方法 startBlockRecovery() 收集了参与到过程中的所有数据节点上的数据块恢复信息后，根据 keepLength 标志，计算出实际需要参与到恢复过程的数据节点列表和恢复后的数据块长度，然后，调用 syncBlock() 方法同步这些数据块。

下面以上述列表 syncList 不为空的情况，介绍数据块同步。在这种情况下，syncBlock() 方法的主要流程是这样的：使用远程方法 DatanodeProtocol.nextGenerationStamp() 向数据节点申请一个新的数据块版本号，作为恢复后数据块的版本号。这个新版本号可以防止出现故障的数据节点重新启动后，上报过时的数据块。接下来，使用恢复后的数据块长度和新数据块版本号，构造新的数据块信息 newblock，再次使用远程接口 InterDatanodeProtocol，调用 syncBlock() 方法同步这些数据节点上的数据块。同步操作结束后，通过远程方法 commitBlockSynchronization() 向名字节点报告这次数据块恢复的结果。

注意，syncList 为空，意味着没有一个数据节点保存着对应的数据块，恢复的主数据节点也需要通知名字节点（也是使用 commitBlockSynchronization() 方法）删除数据块。代码如下：

```
private LocatedBlock syncBlock(Block block, List<BlockRecord> syncList,
      DatanodeInfo[] targets, boolean closeFile) throws IOException {
   …… //syncList 为空，没有一个数据节点有数据块的数据，通知名字节点删除数据块

   long generationstamp = namenode.nextGenerationStamp(block, closeFile);
   Block newblock = new Block(block.getBlockId(),
         block.getNumBytes(), generationstamp);

   for (BlockRecord r : syncList) {
      try {
         r.datanode.updateBlock(r.info.getBlock(), newblock, closeFile);
               successList.add(r.id);
      ……
   }

   if (!successList.isEmpty()) {
      DatanodeID[] nlist = successList.toArray(
            new DatanodeID[successList.size()]);
      namenode.commitBlockSynchronization(block,…….get……Stamp(),
            newblock.getNumBytes(),closeFile, false, nlist);
      ……
}
```

同步数据块的远程方法 updateBlock() 主要功能的实现也是通过 FSDataset.updateBlock() 完成的，FSDataset.updateBlock() 又调用 tryUpdateBlock()，最终，对数据块长度的修改是由静态方法 FSDataset.truncateBlock() 完成的。其中，值得注意的要点有：FSDataset.tryUpdateBlock() 可能会被 updateBlock() 方法调用多次，除了最后一次调用，其他调用都用于获得数据块上可能的写线程，并关闭它们；方法 truncateBlock() 由于对数据块文件进行了截断，需要重新计算校验信息文件，而 tryUpdateBlock() 会在截断动作完成后，通过 validateBlockMetadata()，再次对数据进行检查。

上述几个方法代码冗长，但实现逻辑并不复杂，这里就不再给出详细的代码清单了。

syncBlock() 方法的返回值就是远程方法 recoverBlock() 的返回值，当 recoverBlock() 返回时，客户端了解：

❑恢复数据块的大小和新的版本号。

❑成功进行了数据块恢复的数据节点列表。

根据这些信息，客户端可以重新发起一次写操作，建立数据流管道，从上次的故障点开始，继续写数据。

### 7.2.4　数据块替换、数据块拷贝和读数据块检验信息

我们已经用了大量篇幅介绍数据节点流式接口中最重要的数据块读写接口，数据节点上其他的流式接口还有数据块替换、数据块拷贝和读数据块检验信息，它们分别实现在 DataXceiver 的 replaceBlock()、copyBlock() 和 getBlockChecksum() 中。

数据块替换 replaceBlock() 从某个目标数据节点接收一个完整的数据块，操作成功以后，名字节点就可以删除数据源上的数据块，以释放空间，达到均衡负载的目标。数据块替换的实现依赖于数据块拷贝，它通过向源数据节点发送数据块拷贝请求，获取数据。replaceBlock() 方法比较简单，读入请求后，建立一个到源数据节点的 Socket 连接，并作为客户端发起一个数据块拷贝请求；接下来，节点利用前面已经研究过的 BlockReceiver 接收数据块；数据块接收完毕后，通知名字节点。代码如下：

```
private void replaceBlock(DataInputStream in) throws IOException {
    …… //读入请求的相关信息
    ……
    try {
        //建立到源数据节点的 Socket 连接
        ……
        proxySock = datanode.newSocket();
        NetUtils.connect(proxySock, proxyAddr, datanode.socketTimeout);
        ……
        OutputStream baseStream = NetUtils.getOutputStream(proxySock,
              datanode.socketWriteTimeout);
        proxyOut = new DataOutputStream(new BufferedOutputStream(
              baseStream, SMALL_BUFFER_SIZE));

        /* 发送数据块拷贝请求 */
        proxyOut.writeShort(DataTransferProtocol.DATA_TRANSFER_VERSION);
        proxyOut.writeByte(DataTransferProtocol.OP_COPY_BLOCK); //操作码
        ……
        blockReceiver = new BlockReceiver(block, proxyReply,
              proxySock.getRemoteSocketAddress().toString(),
              proxySock.getLocalSocketAddress().toString(), false,
              "", null, datanode);

        //接收数据块
        blockReceiver.receiveBlock(null, null, null, null,
              dataXceiverServer.balanceThrottler, -1);

        //通知名字节点
```

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

同样，数据块拷贝依赖于BlockSender，从本地磁盘读取数据并发送给执行replaceBlock()方法的数据节点。应该说，BlockSender和BlockReceiver都有一些针对数据块拷贝和数据块替换的特殊处理，不过，这些处理都比客户端发起的数据块读写的处理简单。例如，由于移动数据块是针对整个数据块的操作，在这种情况下，BlockSender不需要计算数据的读取范围，直接从数据块文件的开始发送数据即可。

限于篇幅，这里就不再对数据块拷贝和读数据块检验信息的操作进行深入的探讨，感兴趣的读者可以阅读代码，并寻找它们与数据块读写操作的共同点和差异点，以更好地理解数据节点流式接口的实现。

## 7.3 作为整体的数据节点

数据节点通过数据节点存储和文件系统数据集，管理着保存在Linux文件系统上的数据块，通过流式接口提供数据块的读、写、替换、复制和检验信息获取等功能。建立在上述基础上的数据节点，还需要维护和名字节点的关系，周期性地检查数据块，并作为一个整体保证文件系统的正常工作。

### 7.3.1 数据节点和名字节点的交互

数据节点提供了对HDFS文件数据的读写支持，但是，客户端访问HDFS文件，在读数据时，必须了解文件对应的数据块的存储位置；在写数据时，需要名字节点提供目标数据节点列表，这些操作的实现，都离不开数据节点和名字节点的配合。

数据节点和名字节点间通过IPC接口DatanodeProtocol进行通信，数据节点是客户端，名字节点是服务器。这个接口一共有11个方法，在对写数据块的流程进行分析时，已经涉及了这个接口中的远程方法blockReceived()、reportBadBlocks()、commitBlockSynchronization()和nextGenerationStamp()。其中，前两个方法用于告知名字节点写数据的处理结果，后两个方法直接应用于数据块恢复流程中。

下面考察DatanodeProtocol接口剩余的主要方法，它们用于数据节点到名字节点的握手、注册、数据块上报和心跳。

#### 1. 握手、注册、数据块上报和心跳

无论数据节点是以“-regular”参数启动或者“-rollback”启动，DataNode.startDataNode()方法都会通过handshake()间接调用DatanodeProtocol.versionRequest()，获取名字节点的NamespaceInfo。NamespaceInfo包含了一些存储管理相关的信息，数据节点的后续处理，如版本检查、注册，都需要使用NamespaceInfo中的信息。例如，在handshake()中，数据节点会保证它的构建信息和存储系统结构版本号和名字节点的一致，代码如下：

```
private NamespaceInfo handshake() throws IOException {
    NamespaceInfo nsInfo = new NamespaceInfo();
    while (shouldRun) {
        try {
            nsInfo = namenode.versionRequest(); //调用远程方法
            break;
        ......
    }
    ......
    if (!nsInfo.getBuildVersion().equals(Storage.getBuildVersion())) {
        ......
    }
    assert FSConstants.LAYOUT_VERSION == nsInfo.getLayoutVersion() :.....
    ......
}
```

正常启动数据节点，在 handshake() 调用后，数据节点必须通过远程方法 register() 向名字节点注册，注册的输入和输出都是 DatanodeRegistration 对象，即数据节点的成员变量 dnRegistration。DataNode.startDataNode() 方法会根据数据节点的执行环境，构造对象。代码如下：

```
void startDataNode(Configuration conf, AbstractList<File> dataDirs,
      SecureResources resources) throws IOException {
    ......
    storage = new DataStorage();
    // 构造注册信息
    this.dnRegistration = new DatanodeRegistration(machineName + ":"
          + tmpPort);
    ......
    storage.recoverTransitionRead(nsInfo, dataDirs, startOpt);
    this.dnRegistration.setStorageInfo(storage);
    ......
    this.dnRegistration.setName(machineName + ":" + tmpPort);
    ......
    this.dnRegistration.setInfoPort(this.infoServer.getPort());
    ......
    dnRegistration.setIpcPort(ipcServer.getListenerAddress().getPort());
```

DataNode 的同名方法 register() 调用名字节点提供的注册远程方法，并处理应答。这个方法有几个需要注意的地方。首先，以“-regular”参数启动数据节点时，节点存储 DataStorage 还可能处于未初始化状态（数据节点的第一次启动），这个时候，数据节点存储的标识，即“VERSION”文件的 storageID 属性还没有初始化，方法需要调用 setNewStorageID() 方法构造一个数据节点标识。接下来，数据节点循环地发送注册信息，直到注册成功。

数据节点顺利注册后，在 register() 方法中，还需要根据目前数据节点的配置情况执行一些后续处理，主要包括：可能的节点存储 DataStorage 初始化，数据节点注册完成后，名字节点会返回系统统一存储标识等创建“VERSION”文件的必需信息，帮助数据节点完成节点

存储的初始化工作。至此，我们清楚数据节点“VERSION”文件中各个属性的来源，它们大部分来自名字节点，是整个 HDFS 集群的统一属性，如 namespaceID 和 layoutVersion 等；有一些是数据节点自己产生的，包括 storageID 和 storageType。

如果系统支持 HDFS 的 Append 操作，那么 register() 方法必须使用远程方法 blocksBeingWrittenReport()，报告数据节点上处于客户端写状态的数据块信息。这些数据块对应的数据文件保存在 ${dfs.data.dir}/blocksBeingWritten 目录下（请参考 FSDataset 的讨论），它保存着客户端发起的写操作创建的数据块，但这些数据块还没有提交。名字节点收到这些信息后，会根据情况，对数据块进行恢复或者删除。代码如下：

```
private void register() throws IOException {
   if (dnRegistration.getStorageID().equals("")) {
      setNewStorageID(dnRegistration); // 构造一个新的数据节点标识
   }
   while (shouldRun) {
      try {
         dnRegistration.name = machineName + ":"
               + dnRegistration.getPort(); // 注册信息
         dnRegistration = namenode.register(dnRegistration);
         break;
      ......
   }
   ......
   if (storage.getStorageID().equals("")) {
      storage.setStorageID(dnRegistration.getStorageID());
      storage.writeAll();  // 完成数据节点存储初始化
      ......
   if (supportAppends) {
      Block[] bbwReport = data.getBlocksBeingWrittenReport();
      long[] blocksBeingWritten = BlockListAsLongs
            .convertToArrayLongs(bbwReport);
      namenode.blocksBeingWrittenReport(dnRegistration,
            blocksBeingWritten);
   }
   scheduleBlockReport(initialBlockReportDelay);
}
```

DataNode.register() 的最后，调用了 scheduleBlockReport()，在随机等待一小段时间后，DataNode 的主循环 offerService() 方法就会使用 blockReport() 上报数据节点上现有的数据块信息。

名字节点保存并持久化了整个文件系统的文件目录树以及文件的数据块索引，但名字节点不持久化数据块的保存位置。HDFS 启动时，数据节点需要报告它上面保存的数据块信息，帮助名字节点建立数据块和保存数据块的数据节点的对应关系。这个操作会定时执行，主要流程如下面代码。通过 FSDataset 的 getBlockReport() 方法，DataNode.offerService() 获得当前数据节点上所有数据块的列表，然后将这些数据块信息序列化成一个长整型数组，发送该数组到名字节点。远程方法调用结束后，名字节点返回一个名字节点指令，数据节点随后执

行该指令。需要强调的是：数据块上报 blockReport() 的定期执行，为数据节点和名字节点之间数据的一致性提供了重要的保证。

```
public class DataNode implements……, DatanodeProtocol,…… {
        ……
        lastHeartbeat = startTime;
        DatanodeCommand[] cmds = namenode.sendHeartbeat(
                  dnRegistration, // 数据节点的标记
                  data.getCapacity(), // 数据节点的存储容量
                  data.getDfsUsed(), // 目前已经使用的容量
                  data.getRemaining(), // 剩余容量
                  xmitsInProgress.get(), // 正在进行数据块拷贝的线程数
                  getXceiverCount()); //DataXceiverServer 中的服务线程数
        ……
        if (!processCommand(cmds))
           continue;
     }
     ……
     // 每隔一段时间，数据节点会上报它管理的所有数据块
     if (startTime - lastBlockReport > blockReportInterval) {
        ……
        Block[] bReport = data.getBlockReport();
        ……
        DatanodeCommand cmd = namenode.blockReport(dnRegistration,
              BlockListAsLongs.convertToArrayLongs(bReport));
        ……
        processCommand(cmd);
   }
……
```

DataNode.offerService() 循环中另一个和名字节点的重要交互是心跳，名字节点根据数据节点的定期心跳，判断数据节点是否正常工作。心跳上报过程中，数据节点会发送能够描述当前节点负载的一些信息，如数据节点存储容量、目前已使用容量等，名字节点根据这些信息估计数据节点的工作状态，均衡各个节点的负载。远程方法 sendHeartbeat() 执行结束，会携带名字节点到数据节点的指令，数据节点执行这些指令，保证 HDFS 系统的健康、稳定运行。

### 2. 名字节点指令的执行

名字节点通过 IPC（主要是心跳）调用返回值，通知数据节点执行名字节点指令，这些指令最后都由 DataNode.processCommand() 方法处理。方法的主体是一个 case 语句，根据命令编号执行不同的方法。如命令 DNA_REGISTER 就是通过调用前面已经研究过的 register() 方法实现的，代码如下：

```
private boolean processCommand(DatanodeCommand cmd) throws IOException {
   ……
   final BlockCommand bcmd = cmd instanceof BlockCommand ?
         (BlockCommand) cmd: null;
```

```
switch (cmd.getAction()) {
case DatanodeProtocol.DNA_TRANSFER:
   // 拷贝数据块到其他节点
   transferBlocks(bcmd.getBlocks(), bcmd.getTargets());
   ……
   break;
case DatanodeProtocol.DNA_INVALIDATE:
   Block toDelete[] = bcmd.getBlocks();
   try {
      if (blockScanner != null) {
         blockScanner.deleteBlocks(toDelete);
      }
      data.invalidate(toDelete);
   }
   ……
   break;
case DatanodeProtocol.DNA_SHUTDOWN:// 已经废弃不用
……
case DatanodeProtocol.DNA_REGISTER:
   if (shouldRun) {
      register();// 再次注册，注意，register() 会触发 blockReport()
   }
   break;
case DatanodeProtocol.DNA_RECOVERBLOCK:
   recoverBlocks(bcmd.getBlocks(), bcmd.getTargets());
   break;
……
```

命令 DNA_TRANSFER、DNA_INVALIDATE 和 DNA_RECOVERBLOCK 与数据块相关，其中，DNA_INVALIDATE 指令的实现较为简单，顺序在数据块扫描器和文件系统数据集对象中删除数据块，FSDataset.invalidate() 通过异步磁盘操作服务 FSDatasetAsyncDiskService 删除 Linux 文件系统上的数据块文件和校验信息文件，降低了 processCommand() 的执行时间。

名字节点指令 DNA_RECOVERBLOCK 用于恢复数据块，这是由名字节点发起的数据块恢复，作为整个系统故障恢复的重要措施。指令 DNA_RECOVERBLOCK 可以恢复客户端永久崩溃形成的，还处于写状态的数据块。由名字节点触发的数据块恢复，恢复策略永远是截断，即将数据块恢复到参与到过程的各数据节点上的数据块副本长度的最小值；同时，恢复过程结束后，主数据节点通知名字节点，关闭它上面还处于打开状态的文件。

当名字节点发现某数据块当前副本数小于目标值时，名字节点利用 DNA_TRANSFER 命令，通知某一个拥有该数据块的数据节点将数据块复制到其他数据节点。

**注意** 这里复制的目标节点可以有多个，同时，虽然数据节点流式接口提供了数据块替换和数据块拷贝操作，但那是给平衡器 Balancer 使用的，数据块复制指令不依赖于这两个接口。

DNA_TRANSFER 命令的实现 transferBlocks()，会创建一个 DataNode.DataTransfer 对象，该对象拥有自己的线程，并利用这个线程发起到目标数据节点的写数据操作。在数据流

管道建立以后，DataTransfer 线程使用 BlockSender 往管道发送数据块数据，复制数据块到多个数据节点。限于篇幅，就不再给出 DataNode.DataTransfer() 实现的详细分析，感兴趣的读者可以阅读相关的代码。

## 7.3.2　数据块扫描器

每个数据节点都会执行一个数据块扫描器 DataBlockScanner，它周期性地验证节点所存储的数据块，通过 DataBlockScanner 数据节点可以尽早发现有问题的数据块，并汇报给数据节点。

数据块扫描器是数据节点中比较独立的一个模块，如图 7-18 所示，包括扫描器的主要实现 DataBlockScanner 和辅助类 BlockScanInfo、LogEntry 和 LogFileHandler 等。

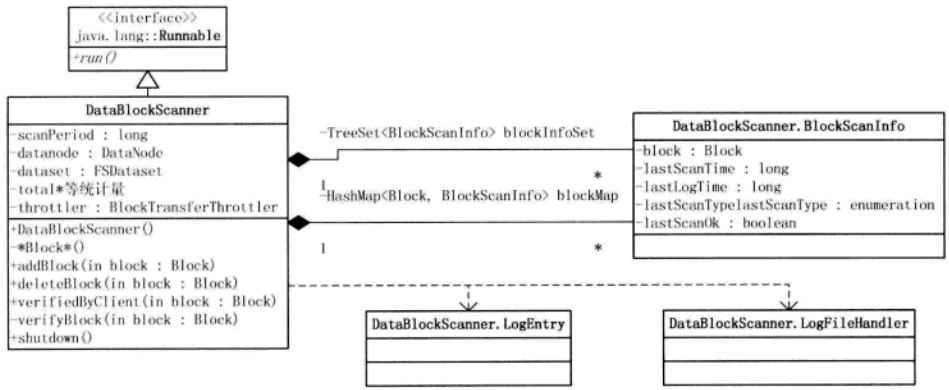



图 7-18　数据块扫描器类图

先从辅助类开始研究数据块扫描器的实现。BlockScanInfo 保存的数据块扫描相关信息包括：

❑block：数据块。

❑lastScanTime：最后一次扫描的时间。

❑lastLogTime：最后写日志的时间。

❑lastScanType：扫描的类型，定义在枚举 DataBlockScanner.ScanType 中。

❑lastScanOk：扫描结果，true 表示通过了扫描，false 表明扫描失败。

其中，扫描的类型包括：

❑NONE：还没有执行过扫描。

❑REMOTE_READ：最后一次扫描结果是由客户端读产生。

❑VERIFICATION_SCAN：扫描结果由数据块扫描器产生。

前面介绍客户端读数据的过程时分析过，客户端成功读取了一个完整数据块（包括校验）后，会发送一个附加的响应码，通知数据节点校验成功，这个信息会更新在 DataBlockScanner 的记录中，对应的类型就是 REMOTE_READ。

BlockScanInfo 对象是可比较的，比较方法 compareTo() 比较 lastScanTime 的大小，按时间先后排序（lastScanTime 相同时比较数据块，注意，数据块是可比较的），这样，保存着所有待扫描数据块信息的成员变量 DataBlockScanner.blockInfoSet 中的元素，也就根据扫描时间排序。

辅助类 LogEntry 和 LogFileHandler 和扫描器日志文件相关。由于数据块每三周才被扫描一次（定义在常量 DataBlockScanner.DEFAULT_SCAN_PERIOD_HOURS 中），将扫描信息保存在文件里，以防止数据节点重启后丢失，就很有必要了。扫描器日志文件保存在 ${dfs.data.dir}/current/dncp_block_verification.log.curr 和 ${dfs.data.dir}/current/dncp_block_verification.log.prev 中，下面是日志记录的例子：

```
date="2012-04-17 11:59:40,323" time="1334635180323" genstamp="1018"\
   id="2539209299677295944"
.....
date="2012-04-20 15:13:36,198" time="1334906016198" genstamp="2872"\
   id="6943529172561203359"
```

记录表明，标识为 2539209299677295944 的数据块（数据块版本是 1018），在 2012 年 04 月 17 日的 11 点 59 分 40 秒扫描过。扫描器启动时，会加载保存在上述两个文件中的扫描信息，扫描的结果，包括客户端上报的结果或数据块扫描产生的结果，都会保存在日志文件中。

数据块扫描器 DataBlockScanner 通过 verifyBlock()，对数据块进行校验。这个方法不算复杂，检查由 BlockSender 执行，通过 BlockSender 将数据发生到一个空的数据流中。在这个场景中，不会使用“零拷贝”数据传输机制，这时，BlockSender 会在读取数据的时候进行校验，并将数据写入 NullOutputStream 对象中。根据 BlockSender.sendBlock() 是否抛出异常，即可得到扫描的结果，并调用 updateScanStatus() 更新 DataBlockScanner 对象的状态并写日志。如果两次检查，数据块都发生校验错误，verifyBlock () 使用 handleScanFailure() 通知名字节点，报告数据块错误。代码如下：

```
private void verifyBlock(Block block) {
   BlockSender blockSender = null;

   /* 这个讨论很有意思，如果出错，在进行第二次检查的时候，如何清空 Linux 内核缓冲区
    * In case of failure, attempt to read second time to reduce
    * transient errors. How do we flush block data from kernel
    * buffers before the second read?
    */
   for (int i=0; i<2; i++) {// 如果第一次检查失败，会再做一次检查
      boolean second = (i > 0);
      try {
         adjustThrottler(); // 调整节流器参数
         blockSender = new BlockSender(block, 0, -1, false,
               false, true, datanode);
         DataOutputStream out =
               new DataOutputStream(new IOUtils.NullOutputStream());
         blockSender.sendBlock(out, null, throttler);
```

```
            ......
            updateScanStatus(block, ScanType.VERIFICATION_SCAN, true);
            return;
        } catch (IOException e) {
            ......
            updateScanStatus(block, ScanType.VERIFICATION_SCAN, false);
            // 数据块不存在（出现竞争条件），不是出错
            if ( dataset.getFile(block) == null ) {
                ......
            if (second) {
                ......
                handleScanFailure(block);
        ......
```

DataBlockScanner.verifyBlock() 巧妙地使用了 BlockSender 实现了数据校验的功能，整个逻辑简单、清晰。

在 DataNode.offerService() 中，数据节点会（定期）启动数据块扫描器检查数据块，代码如下：

```
public class DataNode extends Configured implements InterDatanodeProtocol,
        ClientDatanodeProtocol, FSConstants, Runnable, DataNodeMXBean {
    ......
    public void offerService() throws Exception {// 在数据节点服务线程中执行
        ......
            // 启动扫描器
            if (blockScanner != null && blockScannerThread == null
                  && upgradeManager.isUpgradeCompleted()) {
                blockScannerThread = new Daemon(blockScanner);
                blockScannerThread.start();
            }
```

数据块扫描器拥有自己的线程，DataBlockScanner.run() 会隔一段时间检查 DataBlockScanner.blockInfoSet 中的第一个记录，并记录结果。由于 blockInfoSet 中保存的记录按数据块最后一次扫描的时间 lastScanTime 排序，所以，第一个记录对应的数据块就是最长时间没有检查的数据块。

### 7.3.3　数据节点的启停

万事俱备，我们来启动数据节点。

#### 1. 数据节点的启动

DataNode.main() 是数据节点的入口方法，但它只完成了一件事情，即调用 secureMain() 方法。

secureMain() 方法通过 createDataNode() 创建并启动数据节点，然后通过 DataNode 对象的 join()，等待数据节点停止运行。代码如下：

```
public static void main(String args[]) {
```

```
    secureMain(args, null);
}

public static void secureMain(String[] args, SecureResources resources) {
    try {
        ......
        DataNode datanode = createDataNode(args, null, resources);
        if (datanode != null)
            datanode.join();
    ......
}

public static DataNode createDataNode(String args[], Configuration conf,
        SecureResources resources) throws IOException {
    DataNode dn = instantiateDataNode(args, conf, resources);
    runDatanodeDaemon(dn);
    return dn;
}
```

DataNode.createDataNode() 方法又是通过另外一个方法创建数据节点的实例，然后，方法调用 runDatanodeDaemon() 执行数据节点的主线程。

回到数据节点实例的创建，createDataNode() 调用了 instantiateDataNode() 方法，静态方法 instantiateDataNode() 对输入参数进行了一些简单检查后，接下来调用 makeInstance() 方法，在 makeInstance() 中，保证至少有一个数据节点存储目录可用的情况下构造数据节点。

图 7-19 给出了这个繁琐的调用关系。这种设计保证了每个方法只需要完成一个简单而具体的任务，同时，也提供了通过不同方式启动数据节点的选择，如支持或禁止安全特性，会使用不同的调用链。

图 7-19 数据节点启动时的调用链

数据节点的构造函数很简单，主要的初始化工作由 startDataNode() 完成。这个方法依次完成如下工作：

1）获取节点的名字和名字节点的地址，读入一些运行时需要的配置项。

2）构造注册需要的 DatanodeRegistration 对象。

3）建立到数据节点的 IPC 连接并握手，即调用名字节点上的远程方法 handshake()。

4）执行可能的存储空间状态恢复，构造数据节点使用的 FSDataset 对象。

5）创建流接口服务器 DataXceiverServer。

6）创建数据块扫描器 DataBlockScanner。

7）创建数据节点上的 HTTP 服务器，该服务器提供了数据节点的运行状态信息[⊖]。

8）创建数据节点 IPC 服务器，对外提供 ClientDatanodeProtocol 和 InterDatanodeProtocol 远程接口服务。

可见，在 startDataNode() 方法中，已经对数据节点上的主要模块进行了初始化，但模块提供的服务都还没有投入正式使用。接下来，DataNode.createDataNode() 会调用 runDatanodeDaemon() 方法，首先通过前面介绍的 DataNode.register() 向名字节点注册当前数据节点，成功注册后，执行数据节点的服务主线程。代码如下：

```
public static void runDatanodeDaemon(DataNode dn) throws IOException {
   if (dn != null) {
      dn.register();// 数据节点注册
      dn.dataNodeThread = new Thread(dn, dnThreadName);
      dn.dataNodeThread.setDaemon(true); // 为单元测试准备
      dn.dataNodeThread.start();
   }
}
```

DataNode 类实现了 Runnable 接口，runDatanodeDaemon() 在数据节点服务主线程对象上调用 start() 方法，线程执行 DataNode.run() 方法，run() 方法入口处就启动了流式接口和 IPC 接口的服务，这个时候，数据节点处于正常服务状态，可以接收外部请求。代码如下：

```
public class DataNode extends Configured implements InterDatanodeProtocol,
      ClientDatanodeProtocol, FSConstants, Runnable, DataNodeMXBean {
   ......
   public void run() {
      ......
      // 启动数据节点对外服务，包括流式接口和 IPC 接口
      dataXceiverServer.start();
      ipcServer.start();

      while (shouldRun) {
         try {
            startDistributedUpgradeIfNeeded();
            offerService();
      ......
   }
}
```

DataNode.run() 循环调用 offerService() 方法，而 offerService() 其实也是一个循环，在这个循环里会执行如下操作：

1）发送到名字节点的心跳，并执行（可能的）名字节点指令。

2）通过 blockReceived() 上报数据节点上接收到的数据块。

3）根据需要调用远程接口 DatanodeProtocol.blockReport()，报告数据节点目前保存的数据块信息。

---

⊖ 如果打开 WebHDFS，这个时候会创建 WebHDFS 工作时需要的环境。

4）根据需要启动数据块扫描器 DataBlockScanner。

这些操作的细节已经分别在前面的章节中研究过了，不再重复。

### 2. 数据节点的停止

相对于启动，数据节点停止不需要做太多工作。DataNode.shutdown() 用于停止数据节点，它的调用时机如下（见图 7-20）：

1）DataNode 构造失败，startDataNode() 方法抛出 IOException 异常。

2）数据节点服务主线程捕获如下异常：UnregisteredDatanodeException（节点未注册）、DisallowedDatanodeException（节点被撤销）和 IncorrectVersionException（节点版本不正确）。

3）收到名字节点指令 DatanodeProtocol.DNA_SHUTDOWN。现在不通过这种方式撤销节点，而是使用上面提到的 DisallowedDatanodeException 异常。

4）数据节点服务主线程退出前，即 run() 返回前。

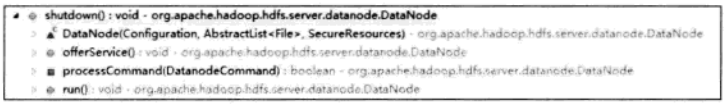



图 7-20 DataNode.shutdown() 的调用者

DataNode.shutdown() 的代码如下：

```
public void shutdown() {
   ......
   if (infoServer != null) {
      try {
         infoServer.stop();
      }
      ......
   }
   if (ipcServer != null) {
      ipcServer.stop();
   }
   this.shouldRun = false;
   if (dataXceiverServer != null) {
       ((DataXceiverServer)
         this.dataXceiverServer.getRunnable()).kill();
      this.dataXceiverServer.interrupt();

      // 等待所有流式接口服务线程退出
      if (this.threadGroup != null) {
         while (true) {
            this.threadGroup.interrupt();
            ......
            if (this.threadGroup.activeCount() == 0) {
               break;
            }
            ......
```

```
                Thread.sleep(1000);
                ......
            }
            try {
                this.dataXceiverServer.join();
                ......
        }

        RPC.stopProxy(namenode); // 结束到名字节点的 IPC 客户端
        ......
        if (blockScannerThread != null) {
            blockScannerThread.interrupt();
            ......
        }
        if (storage != null) {
            ......
            this.storage.unlockAll();
            ......
        }
        if (dataNodeThread != null) {
            dataNodeThread.interrupt();
            ......
        }
        if (data != null) {
            data.shutdown();
        }
        ......
    }
```

通过这个方法回顾数据节点中的关键组件。首先停止的是数据节点上的 HTTP 服务器和提供 ClientDatanodeProtocol 和 InterDatanodeProtocol 远程接口的 IPC 服务器。由于 DataNode 构造过程失败也会调用 shutdown() 方法，所以，所有的资源释放都需要判断相应的对象是否存在。上述两个服务器停止后，数据节点的成员变量 shouldRun 被设置为 false。

**注意**　volatile 变量是 Java 提供的一种同步机制，它可以确保其他线程获得变量的最新值。变量 shouldRun，顾名思义，即“应该运行”，设置该值为 false，表示数据节点应该停止运行了。

这里的 shouldRun 是一种典型的 volatile 变量用法：检查某个状态标记以判断是否退出循环。在 DataNode.offerService()、DataNode.run()、DataXceiverServer.run()、PacketResponder.run()、DataBlockScanner.run() 等大量方法中，实现逻辑里的循环都会判断这个标记，以尽快在数据节点停止时，退出循环。

接下来要停止的是流式接口的相关服务。由于流式接口的监听线程和服务线程都可能涉及 Socket 上的阻塞操作（即 accept()、read() 和 write() 操作），所以，shutdown() 调用 DataXceiverServer 的 kill() 方法，强行关闭套接字，套接字上的阻塞操作异常返回，并在接下来的循环（上述操作一般处于某一个循环中）中，根据 shouldRun 标记结束循环并退出线程。

其他的清理工作还有：

❑关闭数据节点到名字节点的 IPC 客户端，关闭数据块扫描器。

❑对文件“in_use.lock”解锁并删除文件。

❑中断数据节点服务线程，关闭数据节点拥有的 FSDataset 对象。

shutdown() 方法执行结束后，数据节点的服务线程退出，secureMain() 方法的 join() 调用返回，接着，DataNode.secureMain() 执行它的 finally 语句，在日志系统里打印“Exiting Datanode”信息后，数据节点结束运行并退出。代码如下：

```
public static void secureMain(String[] args, SecureResources resources) {
   try {
      ......
      if (datanode != null)
         datanode.join();
   }
   ......
   finally {
      LOG.info("Exiting Datanode");
      // 一些和磁盘相关的异常情况和安全特性要求通过调用 System.exit() 退出。
      System.exit(0);
   }
}
```

## 7.4 小结

文件数据的保存是文件系统物理实现中十分重要的问题，与传统文件系统类似，HDFS 将不定长度的文件分块，然后保存在数据节点上；和传统文件系统不同的是，这些分块的数据，组织成数据块，以 Linux 文件系统文件的形式，保存在数据节点上。

本章采用自下而上的方法，从数据块存储开始介绍数据节点。数据节点在磁盘上维护了一定的文件结构，这些文件结构由数据节点的 DataStorage 对象和 FSDataset 对象共同管理。其中，Storage 类和 DataStorage 类一起提供了数据节点升级机制；而 FSDataset 类则为数据节点应用逻辑提供了数据块操作服务，并维护数据块的状态。构建在存储服务上的主要是数据节点的流式接口，接口主要提供对 HDFS 文件内容，也就是数据块的读写功能，这是数据节点最重要的能力。其中，读数据的要点包括数据发送范围的确定和“零拷贝”数据传输；而写数据接口则有一个远比读数据复杂的流程，需要建立数据流管道，并在管道的两个方向上分别处理请求和确认，同时，系统出现异常时，还需要多个数据节点配合进行数据块恢复。在这些模块之上，作为 HDFS 系统整体的一部分，数据节点在工作时，需要和名字节点进行交互，如注册、心跳、上报接收到的数据块等，同时，数据节点也接受并执行名字节点指令，和名字节点一起维护 HDFS 集群的全局视图和健康运行。

数据节点是我们研究的第一个 HDFS 实体，通过本章的阅读，相信读者能对数据节点的实现细节了如指掌。

# 第 8 章　名字节点实现

名字节点在 Hadoop 分布式文件系统中虽然只有一个实例，却是最复杂的一个实体。名字节点维护着 HDFS 文件系统中两个最重要的关系：

❑HDFS 文件系统的文件目录树，以及文件的数据块索引，即每个文件对应的数据块列表。

❑数据块和数据节点的对应关系，即某一数据块保存在哪些数据节点的信息。

其中，HDFS 的目录树、元信息和数据块索引等信息会持久化到磁盘上，保存在命名空间镜像和编辑日志中。数据块和数据节点的对应关系则在名字节点启动后，由数据节点上报，动态建立。在上述关系的基础上，名字节点管理数据节点，接收数据节点的注册、心跳、数据块提交等信息上报，发送数据块复制、删除、恢复等名字节点指令；同时，名字节点还为客户端对文件系统目录树的操作和对文件数据读写、对 HDFS 系统进行管理提供支持。

本章采用和第 7 章类似的结构，自下而上，从局部到整体，分析名字节点的工作机制。

## 8.1　文件系统的目录树

从实现者的角度看，文件系统的物理实现主要考虑文件和目录如何存储，存储空间怎样管理和系统如何有效而可靠地工作。第 7 章已经讨论了 HDFS 的文件数据存储，也分析了数据节点上的存储空间管理，现在，继续研究同属于物理实现的文件和目录信息组织。

后面的讨论中，会把与目录树相关，即 HDFS 目录树、文件 / 目录元信息和数据块索引间的关系称为名字节点第一关系，数据块和数据节点的对应关系称为名字节点第二关系。

### 8.1.1　从 i-node 到 INode

在学习 Hadoop 文件系统时，介绍了 Linux 文件系统中的 i-node，即索引节点。索引节点保存 Linux 文件的一些元信息，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 i-node 的后半部分，则存放着数据块索引，也就是文件或目录数据的位置。

根据存放数据的大小，数据块索引可分为 i-node 内索引、一次间接块、二次间接块等。对于文件，数据块索引指向的就是文件数据分块存储的位置，通过这些索引信息，就可以读取或者写入文件数据；对于目录而言，目录项被保存在为目录分配的块中。

上述 i-node 实现，充分考虑了需要将 i-node 保存在块设备上这个因素，所以引入了很多定长的记录和结构。如果不需要考虑 i-node 的保存问题，或者只将 i-node 保存在内存中，可以对 i-node 做哪些改进呢？请大家带着这个问题，一起来分析 HDFS 的 INode 实现。

Linux 的 i-node 是如此的有影响，以至于在名字节点中，对文件和目录的抽象使用 INode 对类进行命名。名字节点中，名字中带 INode 的类包括 INode、INodeDirectory、

INodeFile 和 INodeFileUnderConstruction 等，它们的关系如图 8-1 所示。

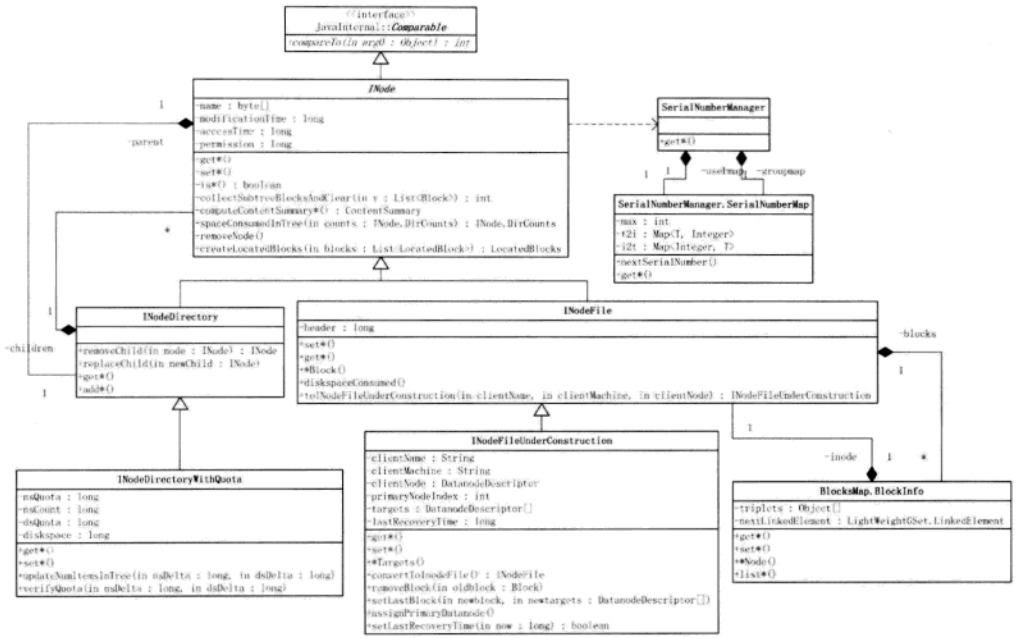



图 8-1 INode* 类图

INode 是一个抽象类，它是 INodeDirectory 和 INodeFile 的父类，INodeDirectory 自然代表了 HDFS 中的目录，而 INodeFile 则抽象了 HDFS 中的文件。INodeDirectory 有一个子类 INodeDirectoryWithQuota，如类的名字所示，就是带配额的目录。INodeFile 的子类 INodeFileUnderConstruction，是这个继承树上的一个“异类”，它代表了一个为写而打开的文件。

### 1. INode

INode 作为这棵继承树的根，保存文件和目录共有的属性，包括文件 / 目录名 name、父目录 parent、最后修改时间 modificationTime、最后访问时间 accessTime 和同时保存访问权限、文件主标识符、文件所在用户组（下面简称用户组）标识符的 permission。和 Linux 的 i-node 比，HDFS 的 INode 不需要支持硬链接、也不需要支持索引节点最后修改时间等功能，所以它保存的属性比 i-node 少。

INode 的属性中需要注意的是 permission，其类型是长整型，即它有 64 字节，保存文件 / 目录的三个属性。如何将这三个属性保存在一个变量中呢？ INode 将 permission 的 64 字节分成三段，分别用于保存访问权限、文件主标识符和用户组标识符，并巧妙地利用了 Java 的枚举，建立长整型上的分段操作，实现上述三个文件属性的操作。代码如下：

```
private static enum PermissionStatusFormat {
```

```
    MODE(0, 16),
    GROUP(MODE.OFFSET + MODE.LENGTH, 25),
    USER(GROUP.OFFSET + GROUP.LENGTH, 23);

    final int OFFSET;
    final int LENGTH; // 数据的位数
    final long MASK;

    PermissionStatusFormat(int offset, int length) {
        OFFSET = offset;
        LENGTH = length;
        MASK = ((-1L) >>> (64 - LENGTH)) << OFFSET;
    }

    long retrieve(long record) {
        return (record & MASK) >>> OFFSET;
    }

    long combine(long bits, long record) {
        return (record & ~MASK) | (bits << OFFSET);
    }
}

/** 获得用户名 */
public String getUserName() {
    int n = (int)PermissionStatusFormat.USER.retrieve(permission);
    return SerialNumberManager.INSTANCE.getUser(n);
}
```

枚举 PermissionStatusFormat 有三个值，MODE、GROUP 和 USER，分别用于处理访问权限、用户组标识符和文件主标识符。上述三个枚举值[⊖]创建时，都会调用 PermissionStatusFormat 的构造函数，构造函数需要两个参数，分别是枚举值对应属性在长整型 permission 中的偏移量和长度。

以访问权限 MODE 为例，它传入构造函数的偏移量为 0，长度为 16，它会使用长整型 permission 中的 0 ～ 15 位保存访问权限。对应的，用户组标识符保存在成员变量 permission 的 16 ～ 40 位，文件主标识符则是 41 ～ 63 位。了解了上述构造函数参数的含义，就不难理解 PermissionStatusFormat 的成员变量和成员函数的作用了。下面，通过和文件主标识符相关的 INode 方法实现，研究如何使用 PermissionStatusFormat。

INode.getUserName() 用于获得该 INode 的文件主，由于文件所有者标识符保存在 permission 的 41 ～ 63 位，使用枚举 USER.retrieve()，它会把成员变量 permission 和掩码 USER.MASK 进行与运算，然后右移，获得标识符的值；然后在保存标识符和用户名对应关系的 SerialNumberManager 实例中进行查找，以得到字符串形式的用户名。

---

⊖ 实际上 enum 声明定义的类型，如 PermissionStatusFormat，是 Java 类库中 Enum 类的子类 (java.lang.Enum<E>)。枚举值是这个类的公有静态常量（public static final），枚举类型的构造方法在构造枚举值的时候被调用。

INode.setUser() 用于设置节点的用户名，它的实现也利用了 PermissionStatusFormat，使用 USER.combine() 设置文件所有者标识符对应位，详细的代码就不列出来了。

HDFS 中，用户名和用户标识的映射、用户组名和用户组标识符的映射保存在 SerialNumberManager 对象中。通过 SerialNumberManager，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，节省了对象对内存的占用。

INode 中的方法大多比较简单，提供了对成员变量的访问 / 设置能力。这些方法中，isRoot() 值得一提，该方法用于判断当前节点是否是目录树的根，目录树的根是 HDFS 中最重要的一个目录，所有目录都由根目录衍生。如果 INode 的成员属性 name 长度为 0，我们约定这是 HDFS 的根节点，INode.isRoot() 返回真，否则，该节点便不是 HDFS 的根。代码如下：

```
boolean isRoot() {
   return name.length == 0;
}
```

### 2. INodeDirectory 和 INodeDirectoryWithQuota

Linux 是使用 C 语言开发的，缺少继承机制，所以，在 i-node 的第一个字段 i_mode 中，保存了文件类型和访问权限字段，后续处理逻辑必须根据 i-node 的文件类型，使用不同的处理逻辑。但在 HDFS 的实现中，利用 Java 的继承机制，目录和文件分别作为 INode 的子类，并以多态实现了文件和目录的特有操作。

INodeDirectory 抽象了 HDFS 中的目录，目录是文件系统中的重要概念，“目录”是一个虚拟容器，里面保存一组文件和其他一些目录。除了根目录，HDFS 中的文件 / 目录都属于某一个“目录”容器，类型为 INodeDirectory 的 INode 成员变量 parent，保存着文件 / 目录的父目录。

目录作为容器，在 INodeDirectory 实现中的体现是成员变量 children，它是一个保存 INode 的列表。INodeDirectory 中的大部分方法都是在操作这个列表，如创建子目录项、查询或遍历子目录项、替换子目录项等，它们的实现都比较简单，下面是几个有代表性的操作：

```
class INodeDirectory extends INode {
   ......
   private List<INode> children;// 目录项
   ......
   INode removeChild(INode node) {
      assert children != null;
      int low = Collections.binarySearch(children, node.name);
      if (low >= 0) {
         return children.remove(low);
      }
      ......
}

abstract class INode implements Comparable<byte[]> {
   ......
   boolean removeNode() {
```

```
        if (parent == null) {// 没有父节点，返回
            return false;
        } else {
            parent.removeChild(this);
            parent = null;// 保护
            return true;
        }
    }
    ......
}
```

INodeDirectory.removeChild() 用于在目录中删除一项，它的实现非常简单，在列表 children 中查找输入的 INode，如果找到了，就在列表中删除相应的项。

INode.removeNode()，就是通过调用父目录对象的 removeChild() 方法，在目录树中删除自身代表的节点。目录的 removeChild() 方法只是简单地将列表 children 中的项删除，这个项可以是一个文件、也可以是一个空目录，甚至是一个目录树。

为什么不需要递归地执行删除操作？如果删除的是文件，怎么才能删除文件拥有的数据块？ collectSubtreeBlocksAndClear() 方法是 INode 的抽象方法，它会返回 INode 所在子目录树中所有文件拥有的数据块。在调用节点删除操作前，名字节点的处理逻辑（实现在类 FSDirectory 中，将会在本节中介绍）会使用这个方法，收集目录树拥有的所有数据块。当成员变量 children 为空，即目录是空目录的情况下，方法直接返回；否则，循环调用目录所管理的目录项的同名方法，收集文件或者子目录中文件的数据块。

collectSubtreeBlocksAndClear() 方法的实现是典型的通过递归遍历目录树模式，INodeDirectory.spaceConsumedInTree() 也使用相同的机制，计算目录占用的空间。代码如下：

```
int collectSubtreeBlocksAndClear(List<Block> v) {
    int total = 1;
    if (children == null) {
        return total;
    }
    for (INode child : children) { //遍历目录项
        total += child.collectSubtreeBlocksAndClear(v);
    }
    ......
}
```

INodeDirectory 有一个子类 INodeDirectoryWithQuota，用于实现 HDFS 的配额机制。HDFS 允许管理员为每个目录设置配额。配额有两种：

- 节点配额：用于限制目录下的名字数量，如果创建文件或目录时超过了该配额，操作会失败。这个配额用于控制用户对名字节点资源的占用，保存在成员变量 nsQuota 中。
- 空间配额：限制存储在目录树中的所有文件的总规模，空间配额保证用户不会过多占用数据节点的资源，该配额由 dsQuota 变量保存。

HDFS 的 dfsadmin 工具提供了修改目录配额的命令，该命令会修改 INodeDirectoryWithQuota 对象相应的成员变量。方法 INodeDirectoryWithQuota.verifyQuota() 则用于检测对目录树的更

新是否满足设定的配额，如果不满足，该方法会抛出异常 NSQuotaExceededException（节点配额）或 DSQuotaExceededException（空间配额）。和 INodeDirectory 一样，INodeDirectoryWithQuota 的成员方法也很简单，感兴趣的读者可以很容易理解它的实现。

### 3. INodeFile 和 INodeFileUnderConstruction

名字节点中，文件由 INodeFile 抽象，它也是 INode 的子类。

INodeFile 包含了两个文件特有的属性：header 和 blocks。变量 header 使用了和 INode.permission 一样的方法，在一个长整型变量里保存了文件的副本系数和文件数据块的大小，它的高 16 字节存放着副本系数，低 48 位存放了数据块大小。数组 blocks 保存文件拥有的数据块，数组元素类型是 BlockInfo。

BlockInfo 继承自 Block，是 org.apache.hadoop.hdfs.server.namenode.BlocksMap 的静态内部类，包含着丰富的信息。从 BlockInfo 对象中可以获得数据块所属的文件，即文件的 INodeFile 对象，也可以获得数据块保存在哪些数据节点的信息。BlockInfo 保存数据块和文件、数据块和数据节点两个关系，是本章开始给出的名字节点中两个最重要关系之间的桥梁。后面还会详细讨论 BlockInfo。

INodeFile 实现的要点如下：

```
class INodeFile extends INode {
   ......
   protected long header;
   protected BlockInfo blocks[] = null;
   ......
   int collectSubtreeBlocksAndClear(List<Block> v) {
      parent = null;
      for (Block blk : blocks) {
         v.add(blk);
      }
      blocks = null;
      return 1;
   }
```

前面已经给出了抽象方法 INode.collectSubtreeBlocksAndClear() 在 INodeDirectory 中的实现，为了讨论的完整性，看一下它在 INodeFile 中的实现。对于 INodeFile 对象，以它为根的子目录树所包含的数据块，就是文件本身拥有的数据块，所以，方法的实现是将成员变量 blocks 的所有元素都加入到列表 v（作为返回值）中，并清理相关的变量。INodeFile 的方法大部分和 collectSubtreeBlocksAndClear () 一样，非常简单。

INodeFileUnderConstruction 是 INodeFile 的子类。按字面的理解，INodeFile-UnderConstruction 就是处于构建状态的文件索引节点，当客户端为写（或追加）数据打开 HDFS 文件时，该文件就处于构建状态，在 HDFS 的目录树中，相应的节点就是一个 INodeFileUnderConstruction 对象。

在 HDFS 中，往文件里添加数据是一件比较复杂的事情，这一点不但体现在第 7 章对写数据流式接口的讨论中，也体现在 INodeFileUnderConstruction 等名字节点相关的实现中。

INodeFileUnderConstruction 的成员变量声明如下：

```
class INodeFileUnderConstruction extends INodeFile {
   String clientName;// 写文件的客户端，也是这个文件租约的文件所有者

   private final String clientMachine; // 客户端所在的主机
   // 如果客户端运行在集群内的某个数据节点上，数据节点的信息
   private final DatanodeDescriptor clientNode;

   private int primaryNodeIndex = -1; // 租约恢复时的主数据节点

   // 最后一个数据块的数据流管道成员
   private DatanodeDescriptor[] targets = null;

   // 租约恢复的开始时间
   private long lastRecoveryTime = 0;
   ……
```

INodeFileUnderConstruction 为写数据保存的信息包括：

- clientName：发起文件写的客户端名称，这个属性也用于租约管理中，在 HDFS 中，租约（后面还有详细讲解）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。
- clientMachine：客户端所在的主机。
- clientNode：如果客户端运行在集群内的某一个数据节点上时，对应的数据节点信息，DatanodeDescriptor 是名字节点内部使用的，用于保存数据节点信息的类，它继承自 DatanodeInfo。
- targets：最后一个数据块的数据流管道成员，即当前参与到写数据的数据节点列表。
- primaryNodeIndex 和 lastRecoveryTime：都用于名字节点发起的数据块恢复，由名字节点发起的数据块恢复也叫租约恢复，这两个变量分别保存恢复时的主数据节点索引和恢复开始时间。

和大部分 INode 的子类类似，INodeFileUnderConstruction 中的方法很平凡，以设置或者获取对象属性为主。需要注意的是，INodeFileUnderConstruction.convertToInodeFile() 和 INode.toINodeFileUnderConstruction() 方法用于 INodeFileUnderConstruction 和 INode 对象的相互转换。

总的来说，INode 及其子类并没有实现太多和名字节点逻辑相关的功能，只是简单地在内存中保存 HDFS 目录树的属性，并提供对这些属性进行存取的方法。

### 8.1.2　命名空间镜像和编辑日志

内存中的 HDFS 目录树及文件 / 目录元信息由 INode 及其子类保存，如果节点掉电或进程崩溃，数据将不再存在，因此，必须将上述信息保存到磁盘。命名空间镜像保存某一个时刻目录树的信息，它在 HDFS 中实现为类 FSImage，是内存元数据和磁盘元数据间的桥梁。对内存目录树的修改，也必须同步到磁盘元数据上，但每次修改都将内存元数据导出到磁

盘，显然是不现实的，为此，名字节点引入了镜像编辑日志，将改动保存在日志中，形成了命名空间镜像和编辑日志的备份机制，其中，命名空间镜像是某一个时间点内存元数据的真实组织情况，而编辑日志则记录了该时刻以后所有元数据操作。本节先对上述机制的实现进行探讨。

### 1. 名字节点的磁盘目录文件结构

命名空间镜像和编辑日志保存在磁盘上，名字节点就必须在磁盘上维护相应的文件结构，以支持 FSImage 对象的工作。启动 HDFS 集群前对名字节点进行格式化，就会建立名字节点文件结构。和数据节点不同，名字节点允许将命名空间镜像和编辑日志保存在不同的目录项中，也就是说，名字节点管理的目录可以分为只保存命名空间镜像（由配置项 ${dfs.name.dir} 指定）、只保存编辑日志（由配置项 ${dfs.name.edits.dir} 指定）和同时保存命名空间镜像和编辑日志（在两个配置项中同时指定）三种情况。和数据节点类似的是，上述两个配置项都可以指定多个目录。在不指定配置项 ${dfs.name.edits.dir} 的情况下，编辑日志也会存放在 ${dfs.name.dir} 中。一般的 HDFS 部署都不会特殊指定 ${dfs.name.edits.dir}，后续讨论就不考虑命名空间镜像和编辑日志分别存放的情况了。

${dfs.name.dir} 目录下一般有三个目录和一个文件，其中文件是“in_use.lock”，它的功能和数据节点中的同名文件是一样的，保证名字节点独占使用该目录，如图 8-2 所示。名字节点格式化后只会产生“current”和“image”目录，节点启动并运行一段时间后，才会建立“previous.checkpoint”目录。

名字节点把命名空间镜像和编辑日志保存在“current”目录下，如图 8-3 所示。

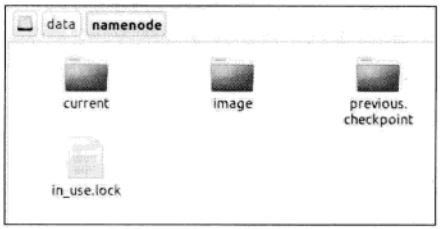



图 8-2 名字节点根目录结构

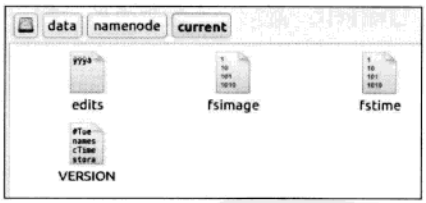



图 8-3 目录“current”的结构

通常，这个目录有 4 个文件，它们分别是：

- fsimage：元数据镜像文件。
- edits：日志文件，和元数据镜像文件一起，提供了一个完整的 HDFS 目录树及元信息。
- fstime：保存了最近一次检查点的时间，检查点一般由第二名字节点产生，是一次 fsimage 和对应 edits 合并的结果。
- VERSION：和数据节点类似，该文件保存了名字节点存储的一些属性。

分析数据节点的时候，已经给过“VERSION”文件的内容，下面是名字节点的属性，意义都很清楚。存储系统标识 namespaceID 是集群全局唯一，由名字节点在格式化的时候创建的标识，数据节点的 namespaceID 必须和名字节点保持一致：

```
#Thu Apr 12 15:55:42 CST 2012
namespaceID=1970677358
cTime=0
storageType=NAME_NODE
layoutVersion=-32
```

${dfs.name.dir}/previous.checkpoint 保存名字节点的上一次检查点，它的目录结构和“current”目录是一致的。${dfs.name.dir}/image 是 0.13 及以前版本保存“fsimage”文件的地方，该目录下的“fsimage”文件和数据节点的 ${dfs.data.dir}/storage 文件有一样的作用，防止不兼容当前目录结构的名字节点的误启动。

### 2. FSImage 和 FSEditLog

数据节点中，对节点存储空间的管理，分别由数据节点存储 DataStorage 和文件系统数据集 FSDataset 实现，DataStorage 主要实现了存储空间的状态管理，而 FSDataset 则提供了数据节点逻辑所需的数据块存储服务。

和数据节点不同，名字节点的存储空间管理由 FSImage 类和 FSEditLog 类一起完成。其中，命名空间镜像 FSImage 起主导作用，它管理着存储空间的生存期，同时，也负责命名空间镜像的保存和加载，还需要和第二名字节点合作，执行检查点过程。FSImage 的类图如图 8-3 所示。它继承自 Storage，在第 7 章已经介绍过 org.apache.hadoop.hdfs.server.common 包中和节点存储相关的类，Storage 处于这个开始于 StorageInfo 的继承层次的中间，提供了管理多个存储目录（StorageDirectory）的能力，数据节点存储 DataStorage 也继承自 Storage。它们都利用了 Storage 提供的方法管理名字节点的目录结构，如图 8-4 所示。

编辑日志 FSEditLog 记录了对元数据的修改，图 8-5 是它的类图。和 FSImage 不同，编辑日志随着名字节点的运行不断产生，所以 FSEditLog 依赖于输出流，使用输出流记录目录树的变化。与输出流对应的是输入流，用于读取持久化在磁盘上的编辑日志。FSEditLog 为记录元数据变化提供了大量的 log*() 方法，如文件改名，就可以通过 logRename() 记录到编辑日志中。

### 3. 命名空间镜像的保存

命名空间镜像存储了某一个时刻名字节点内存元数据的信息，包括前面分析过的 INode 信息，正在写入的文件信息（租约和 INodeFileUnderConstruction 对象）和其他一些状态信息。FSImage.saveFSImage() 会将当前时刻的命名空间镜像，保存到参数 newFile 指定的文件中，代码如下：

```
void saveFSImage(File newFile) throws IOException {
   ……
   // 根据输入文件创建输出流
   DataOutputStream out = new DataOutputStream(
         new BufferedOutputStream(new FileOutputStream(newFile)));
   try {
      // 保存命名空间镜像的文件头
      out.writeInt(FSConstants.LAYOUT_VERSION);
      out.writeInt(namespaceID);
      out.writeLong(fsDir.rootDir.numItemsInTree());
```

```
            out.writeLong(fsNamesys.getGenerationStamp());

            byte[] byteStore = new byte[4 * FSConstants.MAX_PATH_LENGTH];
            ByteBuffer strbuf = ByteBuffer.wrap(byteStore);
            saveINode2Image(strbuf, fsDir.rootDir, out);// 保存根节点
            saveImage(strbuf, 0, fsDir.rootDir, out);// 保存其他节点
            fsNamesys.saveFilesUnderConstruction(out);// 保存构建中的节点
            fsNamesys.saveSecretManagerState(out);// 安全信息
            strbuf = null;
        } finally {
            out.close();
        }
        ......
    }
```

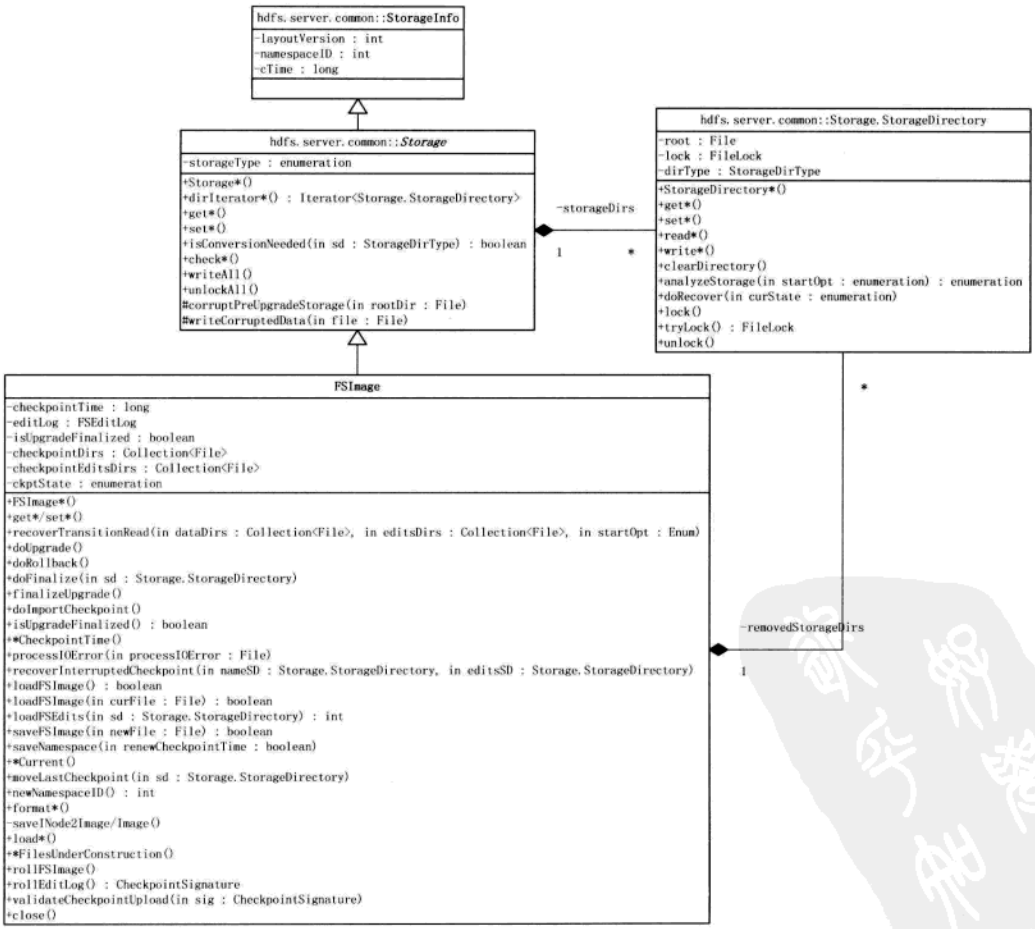



图 8-4 FSImage 类图

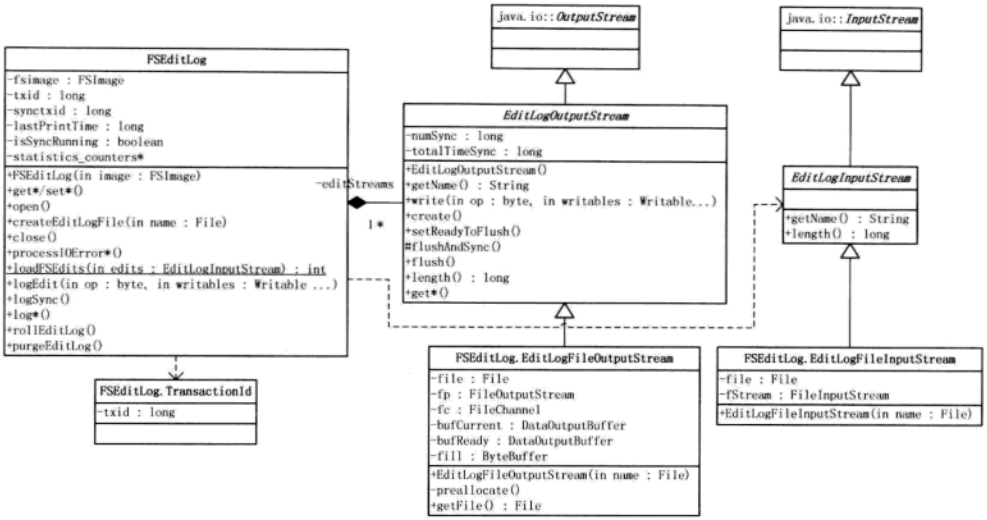



图 8-5　FSEditLog 及其附属类

成员函数 saveFSImage() 很简单，首先输出镜像文件的文件头，如图 8-6 所示，包括命名空间镜像格式的版本号、存储系统标识、目录树包含的节点数和当前已用的数据块版本号。

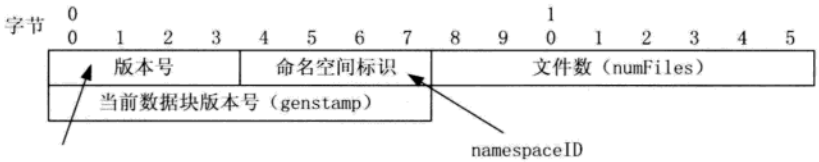



图 8-6　命名空间镜像的文件头

接下来，使用静态方法 saveINode2Image() 输出根节点，使用 saveImage() 保存目录树中的其他节点。根节点是名字节点管理的特殊 INode，它的成员属性 INode.name 长度为 0，所以必须做特殊处理。

FSImage.saveImage() 将目录树 current 下的所有 INode 信息（不包括 current）输出到输出流 out 中。该方法的前两个参数携带了父节点的绝对路径，路径保存在 parentPrefix 中，长度为 prefixLength。由于 HDFS 文件 / 目录的绝对路径不能超过 8000 字节（定义在 FSConstants.MAX_PATH_LENGTH 中）[⊖]，所以缓冲区 parentPrefix 的大小是 31.25KB（4×8000）。如果有一个目录“/foo/bar”，在输出目录“bar”的 INode 时，它的父目录“/foo”会保存在 parentPrefix 中，当输出“/foo/bar”下的 INode 时，parentPrefix 中保存的内容会变成“/foo/bar”。了解这一点，有利于理解 saveImage() 的代码。

假设当前调用的 current 参数值是目录“/foo”，saveImage() 会先循环输出它的所有子节点：

⊖　目录深度最多 1000 层（定义在 FSConstants.MAX_PATH_DEPTH）。

1）设置缓冲区位置，操作完成后缓冲区中的内容为“/foo”。

2）将当前目录通过 ByteBuffer.put() 追加到缓冲区中，这时，缓冲区中保存的是“/foo/bar”。

3）调用 saveINode2Image() 输出节点信息。

如果“/foo”下有多个输出项，在下一个循环的初始又会通过设置缓冲区位置，把上一次循环添加的“/bar”从缓冲区中抹去，让缓冲区中的内容又变成“/foo”。saveImage() 方法灵活地利用缓存区当前位置 position，将 INode 节点中保存的相对路径转换成绝对路径。

子节点输出完成后，saveImage() 会通过递归调用，输出 current 目录下的子目录，这个过程，也需要使用缓冲区 parentPrefix，准备好路径。saveImage() 代码如下：

```
private static void saveImage(ByteBuffer parentPrefix, int prefixLength,
      INodeDirectory current, DataOutputStream out) throws IOException {
   int newPrefixLength = prefixLength;
   if (current.getChildrenRaw() == null)
      return;// 空目录

   for (INode child : current.getChildren()) {// 输出当前节点的所有子节点
      parentPrefix.position(prefixLength);
      parentPrefix.put(PATH_SEPARATOR).put(child.getLocalNameBytes());
      saveINode2Image(parentPrefix, child, out);
   }
   for (INode child : current.getChildren()) {// 子节点是目录，输出该目录
      if (!child.isDirectory())
         continue;// 文件，忽略
      parentPrefix.position(prefixLength);

      // 准备参数
      parentPrefix.put(PATH_SEPARATOR).put(child.getLocalNameBytes());
      newPrefixLength = parentPrefix.position();

      // 递归调用
      saveImage(parentPrefix, newPrefixLength,
            (INodeDirectory) child, out);
   }
   parentPrefix.position(prefixLength);
}
```

saveINode2Image() 方法用于输出一个 INode 对象，由上面的讨论可知，方法的输入参数 name 保存了输出节点的绝对路径，在 saveINode2Image() 方法的入口处，就会往输出流中写入这个路径。所以，在命名空间镜像中，所有的 INode 信息保存的都是绝对路径。代码如下：

```
private static void saveINode2Image(ByteBuffer name, INode node,
              DataOutputStream out) throws IOException {
   int nameLen = name.position();
   out.writeShort(nameLen);
   out.write(name.array(), name.arrayOffset(), nameLen);
   if (!node.isDirectory()) { // 输出文件 INode
      INodeFile fileINode = (INodeFile) node;
      out.writeShort(fileINode.getReplication());
```

```
            ......
            Block[] blocks = fileINode.getBlocks();
            out.writeInt(blocks.length);
            for (Block blk : blocks)
               blk.write(out);
            FILE_PERM.fromShort(fileINode.getFsPermissionShort());
            PermissionStatus.write(out, fileINode.getUserName(),
                  fileINode.getGroupName(), FILE_PERM);
         } else { //输出目录 INode
            ......
            out.writeInt(-1); // 数据块数，-1 代表这是目录
            out.writeLong(node.getNsQuota());
            ......
         }
      };
```

通过 saveINode2Image() 输出的文件信息结构如图 8-7 所示，其中包含了 INodeFile 对象的所有重要信息。

图 8-7　命名空间镜像中的 INodeFile

**注意**　在 INode 中，用户名和用户组名以整数的形式保存在 INode.permission 中，但在命名空间镜像中，用户名和用户组名以字符串的形式保存。

目录由于不包含数据块，所以采用如图 8-8 所示的结构。其中，原来 INodeFile 中的数据块数字段，在 INodeDirectory 中被设置为特殊值 –1，读命名空间镜像时，就是根据这个字段来判断当前的输入是文件还是目录。

FSImage.saveImage() 利用上述两个方法输出目录树以后，会将当前系统中为写打开的文件，即 INodeFileUnderConstruction 对应的文件也写到命名空间镜像中。

FSNamesystem.saveFilesUnderConstruction() 方法会遍历租约管理器中保存的租约，并使用 FSImage.writeINodeUnderConstruction() 输出处于构建状态的文件索引节点。虽然后面才会研究租约管理器的实现原理，但下面的代码很好理解：

```
void saveFilesUnderConstruction(DataOutputStream out) throws IOException {
   synchronized (leaseManager) {
      out.writeInt(leaseManager.countPath());// 租约管理器中管理的项目数
      for (Lease lease : leaseManager.getSortedLeases()) {
         for(String path : lease.getPaths()) {
            INode node = dir.getFileINode(path);
            ……// 确认文件处于构建过程
            INodeFileUnderConstruction cons = (……) node;
            FSImage.writeINodeUnderConstruction(out, cons, path);
         }
      }
   }
}
```

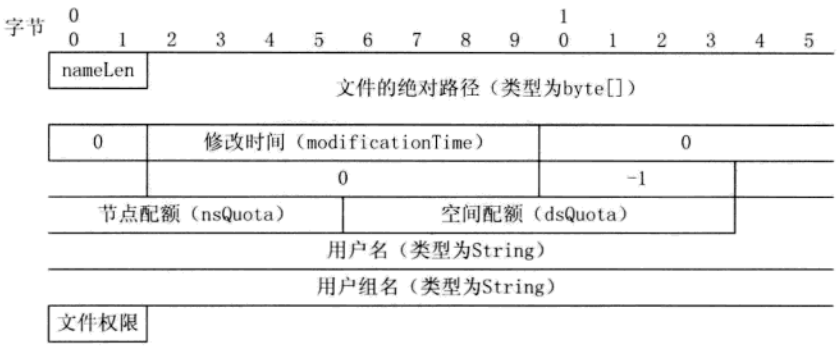



图 8-8 命名空间镜像中的 INodeDirectory

FSImage.writeINodeUnderConstruction() 会将租约记录输出，如图 8-9 所示。

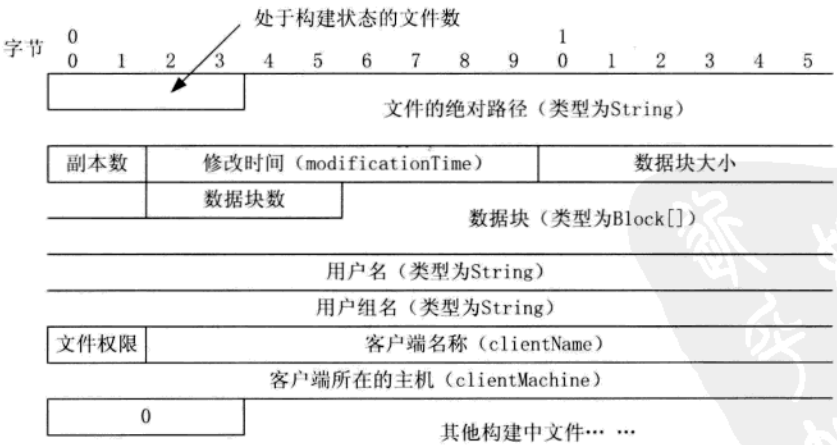



图 8-9 命名空间镜像中的 INodeFileUnderConstruction

命名空间镜像最后一部分内容与安全相关，即调用 FSNamesystem.saveSecretManagerState()

输出的内容，这里就不进行分析了。

至此，命名空间镜像中包含的内容，包括文件头部、目录树信息、租约以及安全管理器状态四部分的机构已经介绍完毕。Hadoop 提供了一个离线命名空间镜像查看器（Offline Image Viewer），该工具可用于对 fsimage 文件进行分析，其实现在 org.apache.hadoop.hdfs.tools.offlineImageViewer 包中，感兴趣的读者可以分析这个工具的代码，以更好地理解 FSImage 的文件格式。

#### 4. 编辑日志数据保存

命名空间镜像 FSImage 作为磁盘上的文件，很难和名字节点内存中的元数据时时刻刻保持一致，为了提高元数据的可靠性，HDFS 将对元数据的修改保存在编辑日志中，编辑日志和命名空间镜像一起，确定了当前时刻文件系统的元数据。

日志是一种支持数据恢复的基础技术，它以一种安全的方式记录系统变更历史，在邮件系统、文件系统和数据库中得到广泛的应用。HDFS 的编辑日志，是一个典型的日志技术范例。当 HDFS 运行时，需要在日志中记录每一个修改名字节点第一关系的事件，所以，可以把日志抽象成一个只允许添加数据的输出流。EditLogOutputStream 抽象了日志输出流，它的一个子类 EditLogFileOutputStream，用于将日志写往磁盘文件，磁盘提供了非易失存储，也有成熟的故障恢复方案（如独立磁盘冗余阵列 RAID），是常用的日志存储设备。使用输出流 EditLogOutputStream 构建日志服务的另一个好处，是可以很容易地将日志传送到其他存储系统，如在 HDFS 的新高可用性方案，就使用了 EditLogOutputStream 的另一个子类 EditLogBackupOutputStream，将日志通过网络直接写往备份名字节点[⊖]。

在图 8-5 中，编辑日志输出流 EditLogOutputStream 继承自输出流 OutputStream，并为写日志数据添加了一些特有的方法。如带变长参数的 write() 方法，就在输出流的基础上，提供了基于记录的日志输出功能。从 write() 方法的参数看，HDFS 的镜像编辑日志包括操作的操作码和操作参数两部分内容，操作码占用一个字节，操作参数长度可变，并采用 Hadoop 特有的序列化机制编码。代码如下：

```
abstract class EditLogOutputStream extends OutputStream {
    ......
    abstract void write(byte op, Writable ... writables) throws IOException;
    ......
    abstract void create() throws IOException;// 创建日志文件
    ......
    abstract protected void flushAndSync() throws IOException;

    public void flush() throws IOException {// 刷出数据
        ......// 记录一下统计信息
        flushAndSync();
        ......
    }
```

⊖ 请参考 HADOOP-4539 的讨论 https://issues.apache.org/jira/browse/HADOOP-4539。

```
    abstract long length() throws IOException;// 获取当前编辑日志的大小
    ……
}
```

编辑日志文件输出流 EditLogFileOutputStream 实现了 EditLogOutputStream，本节以它为例，继续分析编辑日志输出流。

EditLogFileOutputStream 拥有两个工作缓冲区：

❑bufCurrent：日志写入缓冲区。

❑bufReady：写文件缓冲区。

通过 write() 输出的日志记录会写到缓冲区 bufCurrent 中，当 bufCurrent 中的内容需要写往文件时，EditLogFileOutputStream 会交换两个缓冲区，原来的日志写入缓冲区变成了写文件缓冲区，而原来的文件缓冲区则变成日志写入缓冲区。

和一般的输出流不一样，EditLogFileOutputStream 对象构造后，还需要调用对象的 create() 方法，才能开始往输出流中写数据。EditLogFileOutputStream.create() 通过日志输出文件对应的文件通道 fc，将文件清空，并在缓冲区 bufCurrent 中写入版本号，然后调用 setReadyToFlush() 和 flush() 将数据输出到文件中。

一般来说，往日志文件中写入数据需要按照先 setReadyToFlush() 后 flush() 的调用顺序进行，在 setReadyToFlush() 中，方法会在缓冲区 bufCurrent 的尾部写入日志文件结束标识 OP_INVALID，然后交换 bufCurrent 和 bufReady 指向的缓冲区。交换后，bufCurrent 的缓冲区为空，可以立即用于写入日志，bufReady 则保存要写入文件的日志内容。EditLogFileOutputStream 没有覆盖父类的 flush() 实现，所以，写入文件的具体逻辑是由 EditLogFileOutputStream.flushAndSync() 执行。涉及代码如下：

```
static private class EditLogFileOutputStream extends EditLogOutputStream {
    ……
    private FileChannel fc;// 输出文件对应的文件通道
    private DataOutputBuffer bufCurrent;// 日志写入缓冲区
    private DataOutputBuffer bufReady;// 日志输出缓冲区
    ……
    public void write(int b) throws IOException {
        bufCurrent.write(b);
     }

    void write(byte op, Writable... writables) throws IOException {
        write(op); // 输出操作码
        for (Writable w : writables) {// 输出操作参数
            w.write(bufCurrent);
        }
    }
    ……
    void create() throws IOException {
        fc.truncate(0);// 清空文件现有的内容，fc 是文件的文件通道对象
        fc.position(0);
        bufCurrent.writeInt(FSConstants.LAYOUT_VERSION);
```

```
        setReadyToFlush();
        flush();// 调用了 EditLogFileOutputStream 的 flush() 方法
    }
    ......
    void setReadyToFlush() throws IOException {
        assert bufReady.size() == 0 : "previous data is not flushed yet";
        write(OP_INVALID);// 写入日志文件结束标识 OP_INVALID
        DataOutputBuffer tmp = bufReady;
        bufReady = bufCurrent;
        bufCurrent = tmp;
    }

    protected void flushAndSync() throws IOException {
        ......
        bufReady.writeTo(fp);// 写数据到文件中
        bufReady.reset();// 清空缓冲区
        fc.force(false);// 持久化日志数据
        fc.position(fc.position() - 1);// 忽略日志文件结束标识，为下一次写做准备
    }
    ......
}
```

成员函数 flushAndSync() 将数据写入文件后，清空缓冲区，并通过 FileChannel.force() 方法将日志数据持久化到磁盘中；最后，flushAndSync() 还会设置下一次执行写操作时的文件位置，以便覆盖日志文件结束标识。

如同方法名，flushAndSync() 实现了输出内存缓冲区中的日志数据并持久化到磁盘等两个目标：

❑flush：刷新输出流并强制写出所有缓冲的输出数据。

❑sync：强制将刷新数据写入包含该文件的存储设备中。

在 EditLogFileOutputStream 中，flush 只是将输出流中缓存的数据传递给操作系统，并不保证将这些数据实际写入到物理设备中，它们往往只保存在操作系统的内核缓冲区中。sync 则会强制所有内核缓冲区与基础设备同步，FileChannel.force() 方法返回时，与文件相关的内核缓冲区的所有修改，将持久化到磁盘中。

通过对 EditLogFileOutputStream 的分析可了解到，磁盘日志文件由文件头、多条日志记录和文件尾组成。文件头保存，也只保存了日志格式的版本号（FSConstants.LAYOUT_VERSION），文件尾是一个简单的 OP_INVALID 标志。

方法族 FSEditLog.log*() 用于往日志中写入一次名字节点第一关系修改操作，以 logOpenFile() 为例，当客户端打开一个文件时，日志中会写入文件的绝对路径、副本数、修改时间、访问时间、数据块大小、数据块列表、权限（包括用户名、用户组名和访问权限）、客户端名称、客户端所在的主机等字段。其中，副本数、修改时间等整型（包括 long、short 等）属性，会被转换成 UTF8 对象（使用 UTF8 编码保存的字符串，UTF8 类实现了 WritableComparable 接口），记录在日志文件中。代码如下：

```
public void logOpenFile(String path, INodeFileUnderConstruction newNode)
      throws IOException {
   UTF8 nameReplicationPair[] = new UTF8[] {new UTF8(path),
         FSEditLog.toLogReplication(newNode.getReplication()),
         FSEditLog.toLogLong(newNode.getModificationTime()),
         FSEditLog.toLogLong(newNode.getAccessTime()),
         FSEditLog.toLogLong(newNode.getPreferredBlockSize())};
   logEdit(OP_ADD,
         new ArrayWritable(UTF8.class, nameReplicationPair),
         new ArrayWritable(Block.class, newNode.getBlocks()),
         newNode.getPermissionStatus(),
         new UTF8(newNode.getClientName()),
         new UTF8(newNode.getClientMachine()));
}

static private UTF8 toLogLong(long timestamp) {
   return new UTF8(Long.toString(timestamp));
}
```

HDFS 运行过程中记录的修改操作如表 8-1 所示。

**表 8-1 HDFS 运行过程中记录的操作**

| 常量名 / 操作码 | 值 | 操 作 | 记录内容 |
|---|---|---|---|
| OP_INVALID | –1 | 无效 / 结束 | N/A |
| OP_ADD | 0 | 创建文件 | 文件的绝对路径、副本数、修改时间、访问时间、数据块大小、数据块列表、权限（用户名、用户组名和访问权限）、客户端名称、客户端所在的主机 |
| OP_RENAME | 1 | 改名 | 原目标绝对路径，修改后绝对路径，修改时间 |
| OP_DELETE | 2 | 删除 | 被删除目标绝对路径，删除时间 |
| OP_MKDIR | 3 | 创建目录 | 目录的绝对路径、修改时间、访问时间、权限（用户名、用户组名和访问权限） |
| OP_SET_REPLICATION | 4 | 设置文件副本数 | 文件的绝对路径、副本数 |
| OP_DATANODE_ADD | 5 | 增加数据节点 | N/A，不再使用 |
| OP_DATANODE_REMOVE | 6 | 删除数据节点 | N/A，不再使用 |
| OP_SET_PERMISSIONS | 7 | 设置权限 | 目标的绝对路径、新权限（用户名、用户组名和访问权限） |
| OP_SET_OWNER | 8 | 设置文件主 | 目标的绝对路径、用户名、用户组名 |
| OP_CLOSE | 9 | 关闭文件 | 文件的绝对路径、副本数、修改时间、访问时间、数据块大小、数据块列表、权限（用户名、用户组名和访问权限） |
| OP_SET_GENSTAMP | 10 | 设置数据块版本号 | 新数据块版本号 |
| OP_SET_NS_QUOTA | 11 | 设置节点配额 | N/A，不再使用 |
| OP_CLEAR_NS_QUOTA | 12 | 清除节点配额 | N/A，不再使用 |
| OP_TIMES | 13 | 设置访问或修改时间 | 目标的绝对路径、修改时间、访问时间 |
| OP_SET_QUOTA | 14 | 设置配额 | 目录的绝对路径、节点配额、空间配额 |

FSEditLog.log*() 调用 logEdit() 输出日志记录的具体内容，方法不复杂，通过循环往所有的日志输出流中输出日志记录以后，获得一个新的日志交易标识，并记录在 ThreadLocal（线程局部变量）对象 myTransactionId 中。

注意，logEdit() 是 synchronized 方法，这保证 FSEditLog.log*() 通过 logEdit() 输出日志时，日志记录的内容不会相互影响；同时，也保证了 myTransactionId 中保存的日志交易标识（在 FSEditLog 对象）是唯一并递增的。

ThreadLocal 类用于保存线程私有的状态信息，每一个线程都可以独立地通过 set() 方法修改自己拥有的那个线程局部变量的值，而不会和其他线程的同名对象冲突。FSEditLog.logEdit() 方法将日志交易标识保存在线程局部变量 FSEditLog.myTransactionId 中，在下次调用 logEdit() 前，当前线程拥有的日志交易标识不变。参考下面的代码：

```
// 用于保存当前线程的最近一次日志交易标识
private static final ThreadLocal<TransactionId>myTransactionId = new
      ThreadLocal<TransactionId>() {
         protected synchronized TransactionId initialValue() {
            return new TransactionId(Long.MAX_VALUE);
         }
      };

synchronized void logEdit(byte op, Writable... writables) {
   ......
   for (int idx = 0; idx < editStreams.size(); idx++) {
      EditLogOutputStream eStream = editStreams.get(idx);
      try {
         eStream.write(op, writables);// 输出日志记录
      } catch (IOException ie) {
      ......
   }
   txid++;// 获得一个新的日志交易标识并记录
   TransactionId id = myTransactionId.get();
   id.txid = txid;
   ...... // 记录一些统计量
}
```

接下来的问题是，保存在 myTransactionId 中的这个标识用途是什么？

FSEditLog.logEdit() 通过 write() 往 EditLogOutputStream 写入了一条日志记录，按前面的讨论，这条记录往往会保存在流的输出缓冲区中，直到调用者使用 EditLogOutputStream.flush() 才会实际写入到输出目标中。显然，logEdit() 并没有将缓冲区中的数据刷出，所以在一般情况下，调用 logEdit() 方法后，就会使用 FSEditLog.logSync() 同步日志修改。logSync() 是一个比较复杂的方法，其中，FSEditLog 的成员变量 synctxid 保存了已经同步的日志交易标识，如果当前有线程在执行日志同步操作，布尔变量 FSEditLog.isSyncRunning 为 true。实现代码如下：

```
public void logSync() throws IOException {
   ......
```

```
    long mytxid = myTransactionId.get().txid;
    ......
    synchronized (this) {
        ......
        //有其他线程在执行日志同步操作
        while (mytxid > synctxid && isSyncRunning) {
            try {
                wait(1000);
            } catch (InterruptedException ie) {
            }
        }

        // 日志已经被其他线程同步，返回
        if (mytxid <= synctxid) {
            ......
            return;
        }

        // 同步由当前线程执行，记录相关信息
        syncStart = txid;
        isSyncRunning = true;

        // 交换缓冲区
        for (int idx = 0; idx < numEditStreams; idx++) {
            editStreams.get(idx).setReadyToFlush();
        }
    }

    // 执行日志同步操作
    long start = FSNamesystem.now();
    for (int idx = 0; idx < numEditStreams; idx++) {
        EditLogOutputStream eStream = editStreams.get(idx);
        try {
            eStream.flush();
        } catch (IOException ie) {
            ......
        }
    }

    synchronized (this) {
        ......
        synctxid = syncStart;
        isSyncRunning = false;
        this.notifyAll();
    }
    ......
}
```

获取当前线程的交易标识后，logSync() 的处理逻辑可以分为三个步骤：

1）方法的第一个同步代码块为同步日志记录做准备。

名字节点是一个多线程 Java 应用，准备动作的第一步是保证没有其他线程在执行日志记录同步工作，如果当前交易标识大于已处理交易标识，同时标志位 isSyncRunning 为 true，线程会等待一段时间。等待结束时，如果已处理交易标识比当前交易标识还大，表明需要同步的当前线程日志记录已经被其他线程的 logSync() 调用同步，方法返回；否则，当前线程需要执行日志记录同步操作，在记录相关信息后，调用 EditLogOutputStream.setReadyToFlush()，交换输出流的日志写入缓冲区和写文件缓冲区。

2）准备工作完成后，logSync() 调用所有输出流的 flush() 方法，刷新并同步日志记录。

3）最后，logSync() 设置 FSEditLog 的成员变量 synctxid 和 isSyncRunning，结束这次 logSync() 调用并为下次调用做好准备。

以一个不规范的序列图（图 8-10）为例，继续分析多线程情况下 logEdit() 和 logSync() 的配合。

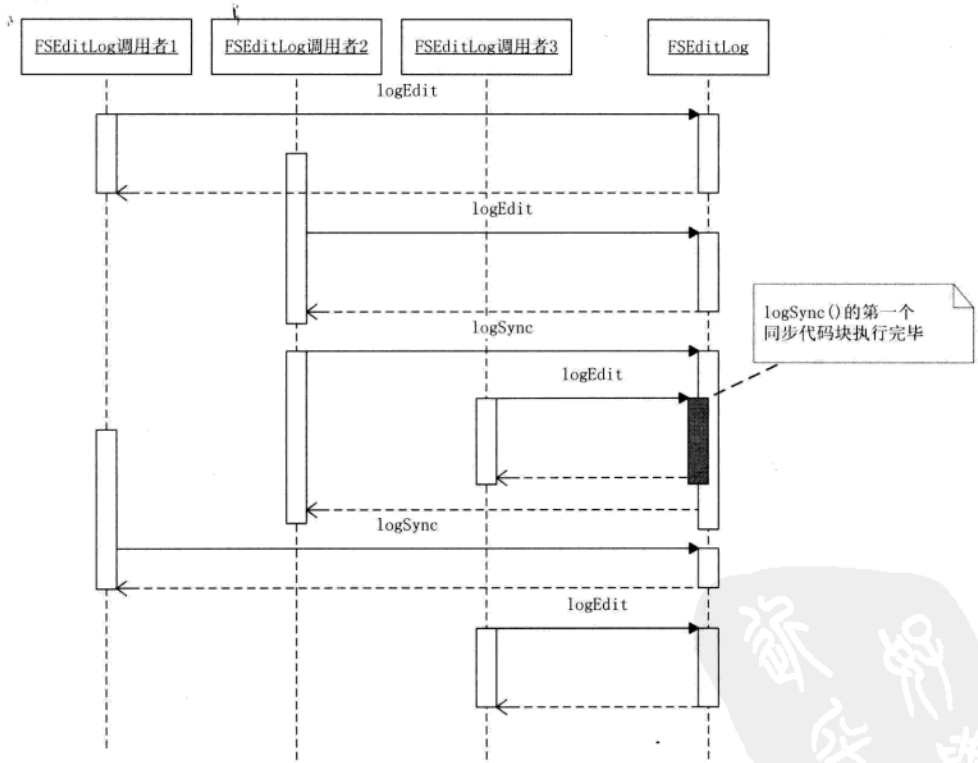



图 8-10　logEdit() 和 logSync() 的配合

假设有三个线程使用 logEdit() 和 logSync() 方法对，输出并同步它们对 HDFS 元数据的修改。调用者 1 在执行了方法 logEdit() 以后，由于线程调度的原因，等待了一段时间，在此期间，调用者 2 连续执行了 logEdit() 和 logSync() 方法。注意，由于 logEdit() 是一个同步方法，所以，即使调用者 1 和调用者 2 并发执行，但在某一个时刻，只有一个线程能执行

FSEditLog.logEdit()。

调用者 2 的 logEdit() 执行结束后，可以假设调用者 1 的日志交易标识为 N+1，那么，调用者 2 的标识为 N+2。调用者 2 开始执行 logSync() 方法的第一个 synchronized 代码块时，如果调用者 3 调用了 logEdit()，由于 logEdit() 是 synchronized 方法，同时，logSync() 的第一个同步块是针对 FSEditLog 对象（即 this）的，调用者 3 的 logEdit() 调用会等待调用者 2 完成 logSync() 的第一个同步调用块的处理逻辑后才开始执行。

显然，调用者 2 的日志交易标识（保存在 myTransactionId 中）比 synctxid（已经同步的日志交易标识）中保存的标识大，调用者 2 的 logSync() 需要执行同步操作，在记录相关信息后，logSync() 交换输出流中的缓冲区，然后退出 synchronized 代码块。

处于等待状态的调用者 3，这个时候就可以执行 logEdit() 往日志写入缓冲区中写数据，同时，调用者 2 的 logSync()，则使用写文件缓冲区，同步日志记录。

细心的读者可能已经发现，EditLogFileOutputStream.setReadyToFlush() 方法不是同步方法，不能保证写数据（使用 EditLogFileOutputStream.write() 方法）或同步数据（EditLogFileOutputStream.flushAndSync() 方法）时，其他线程不会调用 setReadyToFlush()，交换对象的两个工作缓冲区。不过，通过 FSEditLog 对象上的同步，包括 logSync() 中的同步块或者 logEdit() 这样的同步方法，保证了对 EditLogFileOutputStream 的这些可能造成缓冲区竞争的调用以合理的顺序执行，也就保证了缓冲区的正确使用。同样，EditLogFileOutputStream 也不保证多个 write() 调用顺序执行，而是让同步方法 logEdit() 保证多个线程顺序写入日志记录。

当调用者 2 的 logSync() 执行磁盘同步时，如果调用者 1 被调度并开始执行，但是由于标志位 isSyncRunning 现在值为 true，调用者 1 会执行 wait() 方法等待 isSyncRunning 复位，在调用者 2 执行 logSync() 的最后一个 synchronized 代码块结束后（isSyncRunning 复位），该等待结束。这个时候，synctxid 的值为 N+2，由于调用者 1 的线程局部变量 myTransactionId 中保存的值为 N+1，小于 synctxid 现在的值，所以调用者 1 不需要执行磁盘同步操作，它写入的日志记录，已经随调用者 2 的 logSync() 调用的执行同步到物理设备上，调用者 1 的 logSync() 可以返回。

上述过程还有一个要点，通过 isSyncRunning 标志和临时变量 syncStart，保证了调用者 1 在日志记录同步完成后才有机会退出等待，避免出现日志记录尚未完成同步而 logSync() 返回的情况。

由于 EditLogFileOutputStream.flushAndSync() 必然触发磁盘写操作，是一个非常耗时间的操作，所以，HDFS 的开发者在 EditLogFileOutputStream 中引入了两个缓冲区，并利用 FSEditLog.logEdit() 和 FSEditLog.logSync()，让输出日志记录到缓冲区和同步缓冲区到磁盘这两个看似应该放在一起，顺序执行的操作分离，并允许它们并发执行，大大提高了名字节点的吞吐量，是一个十分巧妙的设计，值得读者好好研究。

**5. 读取命名空间镜像和编辑日志数据**

FSImage.loadFSImage() 读取命名空间镜像，并将它包含的元数据添加 / 更新到内存元数

据中，FSEditLog.loadFSEdits() 则加载编辑日志，重放并应用日志记录的操作，以获得元数据在某个时刻的状态。上述两个方法，不但应用于名字节点，也为第二名字节点所使用。

loadFSImage() 比 saveFSImage() 复杂，包含了大量的判断语句，这些判断语句都是为了兼容以前版本的命名空间镜像引入的。以文件系统的目录树项目数为例，当前版本（版本号是 -32，由于是负数，版本号越低版本越高）使用长整型保存，而在版本 -16 以前（包括 -16），使用的是整型保存，所以，在读取命名空间镜像时，需要根据版本号做不同的处理。代码如下：

```
boolean loadFSImage(File curFile) throws IOException {
    ……
    FSNamesystem fsNamesys = FSNamesystem.getFSNamesystem();
    FSDirectory fsDir = fsNamesys.dir;
    ……
    DataInputStream in = new DataInputStream(new……(new……(curFile)));
    try {
        ……
        int imgVersion = in.readInt();// 读入 image 版本号
        this.namespaceID = in.readInt()// 读入 namespaceID

        long numFiles;// 目录树项目数
        if (imgVersion <= -16) {
            numFiles = in.readLong();
        } else {
            numFiles = in.readInt();
        }
        ……
        for (long i = 0; i < numFiles; i++) {
            ……
            parentINode = fsDir.addToParent(path, parentINode, permissions,
                  blocks, replication, modificationTime, atime, nsQuota,
                  dsQuota, blockSize);// 将读入的 INode 添加到目录树中
        }
        ……
        this.loadFilesUnderConstruction(imgVersion, in, fsNamesys);
        this.loadSecretManagerState(imgVersion, in, fsNamesys);
        ……
```

上面的 loadFSImage() 方法中，for 语句包含的代码用于读入文件 / 目录的 INode 信息，并通过 FSNamesystem.addToParent() 方法，将节点添加到目录树中。该方法会根据传入的参数，构造相应的 INodeDirectoryWithQuota 或 INodeDirectory 或 INodeFile 对象，并将对象插入到目录树的相应位置中。

命名空间镜像尾部保存的租约信息和安全相关信息，分别由 loadFilesUnderConstruction() 和 loadSecretManagerState() 读入。租约信息读入时，需要更新目录树上的信息，将原来的 INodeFile 对象替换成 INodeFileUnderConstruction 对象；同时，也需要更新租约管理器中的信息，如下面代码。这两个方法的其他细节就不再讨论了。

```
private void loadFilesUnderConstruction(int version, DataInputStream in,
      FSNamesystem fs) throws IOException {
   ……
   for (int i = 0; i < size; i++) {
      // 读入 INodeFileUnderConstruction 信息
      INodeFileUnderConstruction cons = readINodeUnderConstruction(in);
      …… //检查
      INodeFile oldnode = (INodeFile) old;
      fsDir.replaceNode(path, oldnode, cons);
      fs.leaseManager.addLease(cons.clientName, path);
   }
}
```

通过 loadFSImage() 读取命名空间镜像后，内存中的名字节点第一关系信息只包含了保存镜像的那一个时刻的内容，还需要加载后续对元数据的修改，即编辑日志中的内容，才能完全恢复元数据。FSEditLog.loadFSEdits() 用于加载并应用日志，它需要一个 EditLogInputStream 实例，即 EditLogFileInputStream 对象。和复杂的 EditLogFileOutputStream 相比，EditLogFileInputStream 是一个平凡的类，与流相关的操作都由关联的文件输入流完成。方法 loadFSEdits () 冗长，大部分的实现用于根据日志记录的操作码和操作参数，调用 FSDirectory 中的对应方法，修改内存元信息。

以文件 / 目录改名为例，相关逻辑在读入日志记录中的原目标绝对路径（保存在变量 s 中）、修改后绝对路径（保存在变量 d 中）和修改时间（timestamp）三个参数后，调用 FSDirectory.unprotectedRenameTo() 方法修改目录树。由于文件改名会影响到租约管理器，所以还需要通知管理器更新信息。代码如下：

```
static int loadFSEdits(EditLogInputStream edits) throws IOException {
   ……
   case OP_RENAME: {// 请大家忽略对齐
      ……
      int length = in.readInt();
      if (length != 3) {
         throw new IOException("Incorrect data format. Mkdir operation.");
      }

      // 读入改名操作的日志记录内容
      String s = FSImage.readString(in);
      String d = FSImage.readString(in);
      timestamp = readLong(in);

      HdfsFileStatus dinfo = fsDir.getFileInfo(d);
      fsDir.unprotectedRenameTo(s, d, timestamp); // 改名
      fsNamesys.changeLease(s, d, dinfo); // 修改租约管理器中的记录
      break;
   }
   ……
```

注意，和 loadFSImage() 一样，FSEditLog.loadFSEdits () 也需要向下兼容，但总的来说，

它们的实现都不算复杂。

### 8.1.3　第二名字节点

通过 FSImage.saveFSImage() 输出命名空间镜像是一个非常消耗资源的操作，在名字节点上周期性使用该方法保存元数据，需要让名字节点处于只读状态，这将严重影响运行在 HDFS 上的业务。如果只使用编辑日志，记录文件系统上元数据的变化，随着系统的运行，日志文件将变得越来越大，虽然在名字节点运行期间这不会对系统造成影响，但如果名字节点重新启动，就需要花很长的时间执行 FSEditLog.loadFSEdits()，影响名字节点的可用性。解决的方案是运行一个第二名字节点，由它定期地获取并合并名字节点上的命名空间镜像和编辑日志，生成新的命名空间镜像（又称元数据检查点），然后上传并替换名字节点原有镜像，清空编辑日志。

在第 6 章已经给出过检查点产生的流程，该过程实现在方法 SecondaryNameNode.doCheckpoint() 中。代码如下：

```
void doCheckpoint() throws IOException {
    startCheckpoint(); // 第二名字节点初始化工作区

    // 通知名字节点进行 edits 滚动，返回值用于合并后的检查点上传
    CheckpointSignature sig = (CheckpointSignature) namenode.rollEditLog();

    downloadCheckpointFiles(sig); // 获取命名空间镜像和编辑日志
    doMerge(sig); // 合并，产生新的命名空间镜像

    putFSImage(sig); // 上传新的命名空间镜像到名字节点
    namenode.rollFsImage(); // 通知名字节点，命名空间镜像上传完毕

    checkpointImage.endCheckpoint(); // 清理本地工作区
}
```

对上述过程进行展开，第二名字节点工作时，命名空间镜像合并前后的数据都保存在磁盘上，和其他节点一样，在磁盘上也维护着一定的目录文件结构，所以，doCheckpoint() 需要先保证工作区处于恰当的状态。接下来，第二名字节点调用远程方法 NamenodeProtocol.rollEditLog()，让名字节点进行准备，这时，名字节点会关闭到编辑日志文件“edits”的输出，后续的新日志记录写往“edits.new”文件。

接下来，第二名字节点通过 HTTP 协议下载名字节点上的命名空间镜像“fsimage”和编辑日志“edits”到本地工作区，并进行合并；合并后的内存元数据会保存到第二名字节点的磁盘上，形成新的镜像文件；然后通过 putFSImage() 方法，使用 HTTP 协议上传新命名空间镜像到名字节点。新命名空间镜像成功上传后，第二名字节点再次通过远程接口 NamenodeProtocol，使用 rollFsImage() 方法通知名字节点。这时，名字节点会使用新命名空间镜像作为当前镜像，并将“edits.new”改名为“edits”，这样，新命名空间镜像和编辑日志“edits”，又一起定义了当前内存元数据的真实组织情况。

最后，第二名字节点复位工作区，完成一次检查点处理。

类图如图 8-11 所示，SecondaryNameNode 非常简单，doCheckpoint() 成员函数已经使用了 SecondaryNameNode 类中的主要方法，另一方面，FSImage 已经实现了第二名字节点合并命名空间镜像和编辑日志的功能，并被 CheckpointStorage 复用，这也减低了 SecondaryNameNode 的复杂性。

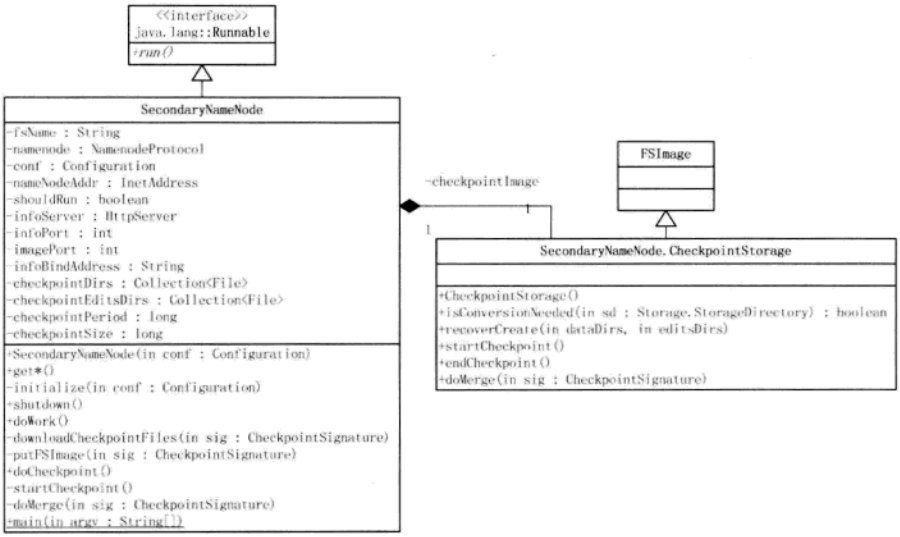



图 8-11 SecondaryNameNode 类图

### 1. 检查点的产生：第二名字节点上的工作

再次按照 SecondaryNameNode.doCheckpoint() 中的顺序来研究第二名字节点如何产生检查点。第一个被调用的方法是 startCheckpoint()，用于初始化工作区，这个函数只有 4 行代码，调用的都是 CheckpointStorage 中的方法。代码如下：

```
private void startCheckpoint() throws IOException {
    checkpointImage.unlockAll();
    checkpointImage.getEditLog().close();
    checkpointImage.recoverCreate
    (checkpointDirs, checkpointEditsDirs);
    checkpointImage.startCheckpoint();
}
```

关键点在后面两个方法调用上，recoverCreate() 用于检验第二名字节点管理的磁盘目录文件结构，并执行可能的文件结构恢复。和名字节点升级一样，产生检查点的过程会使用临时目录，既保存数据，又维护状态。recoverCreate() 会保证目录处于图 8-12 的状态。

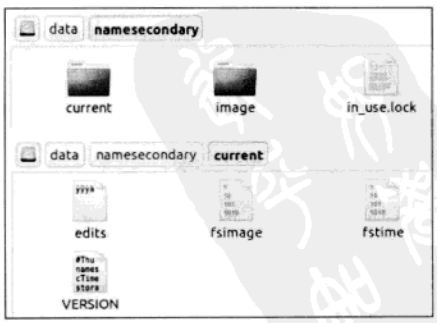



图 8-12 第二名字节点根目录结构和目录“current”的结构

这个目录结构和名字节点正常工作时的状态基本一致。这很正常，CheckpointStorage 类就是继承自 FSImage；同时，如果名字节点发生严重的故障并且没有用于恢复的备份，可以拷贝第二名字节点的“current”目录中的数据，并通过“-importCheckpoint”选项启动名字节点，恢复名字节点上的元数据到某一个检查点。

接下来执行的 CheckpointStorage.startCheckpoint() 方法，通过 FSImage.moveCurrent() 创建命名空间镜像合并时需要的目录：即将“current”目录名改为“lastcheckpoint.tmp”，并创建一个新的“current”目录。代码如下：

```
protected void moveCurrent(StorageDirectory sd) throws IOException {
   File curDir = sd.getCurrentDir(); //“current”目录
   File tmpCkptDir = sd.getLastCheckpointTmp(); //“目录 lastcheckpoint.tmp”
   if (sd.getVersionFile().exists()) {
      assert curDir.exists() : curDir + " directory must exist.";
      assert !tmpCkptDir.exists() : tmpCkptDir + " ……";
      rename(curDir, tmpCkptDir);// 目录改名
   }

   if (!curDir.exists() && !curDir.mkdir())// 重新创建 current 目录
      throw new IOException("Cannot create directory " + curDir);
}
```

该调用结束后，第二名字节点上的根目录结构如图 8-13 所示。

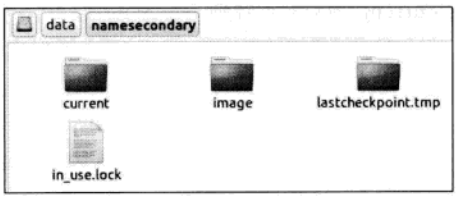



图 8-13　产生检查点时的第二名字节点根目录结构

如果大家在第二名字节点的磁盘目录文件结构中发现“lastcheckpoint.tmp”目录，表明第二名字节点正在创建检查点。上述准备工作完成后，接下来 SecondaryNameNode.doCheckpoint() 将调用名字节点的 rollEditLog() 远程方法（下一小节研究），这个时候名字节点上会发生什么，这里就不展开讨论了。

downloadCheckpointFiles() 方法通过名字节点和第二名字节点间的 HTTP 接口，下载名字节点上的“fsimage”和“edits”文件（参考下面的 URL 格式）到“current”目录中：

```
http://< 名字节点地址 >:< 名字节点的 HTTP 端口 >/getimage?getimage=1
http://< 名字节点地址 >:< 名字节点的 HTTP 端口 >/getimage?getedit=1
```

该方法使用了 TransferFsImage.getFileClient()，TransferFsImage 是一个辅助类，它的 getFileClient() 方法提供了通过 GET 请求，获取 HTTP 服务器上的一个文件并保存在本地文件系统的功能；getFileServer() 被 GetImageServlet.doGet() 使用，它将本地文件发送到某一个 HTTP 输出流上。文件下载结束后，当前 CheckpointStorage 对象的状态会从 CheckpointStates.START 改为 CheckpointStates.UPLOAD_DONE。

命名空间镜像和编辑日志两个文件下载完毕，就可以进行元数据合并了。接下来执行 SecondaryNameNode.doMerge()，它调用的还是 CheckpointStorage 中的同名方法，在

对相关参数进行必要的检查后，通过已经分析过的 FSImage.loadFSImage() 和 FSEditLog.loadFSEdits() 方法，在内存中建立检查点。第二名字节点需要在内存中保存命名空间镜像的全部内容，所以它对内存的要求和名字节点一样高。代码如下：

```
private void doMerge(CheckpointSignature sig) throws IOException {
   getEditLog().open();
   ......
   loadFSImage(FSImage.getImageFile(sdName, NameNodeFile.IMAGE));
   loadFSEdits(sdEdits);
   sig.validateStorageInfo(this);
   saveNamespace(false);
}
```

FSImage.saveNamespace() 持久化内存元数据到磁盘中，创建命名空间镜像，即检查点。这个方法关键步骤如下：

1）关闭日志输出流（注意，这个方法同时也被名字节点使用，第二名字节点不关注日志输出流），记录这次检查点的产生时间。

2）目录改名，将“current”目录名改为“lastcheckpoint.tmp”。

3）建立新的“current”目录，并使用 FSImage.saveCurrent() 输出命名空间镜像，即检查点，saveCurrent() 的最后会写“VERSION”文件和“fstime”文件。

4）再次调用 saveCurrent() 方法创建新的日志输出流（所以，大家会在第二名字节点的“current”目录下看到一个空的“edit”文件，大小为 4 字节）。

5）将目录“lastcheckpoint.tmp”名改为“previous.checkpoint”。

6）设置检查点状态为 CheckpointStates.UPLOAD_DONE。

在 saveNamespace() 方法中再次使用了通过不同的目录保存存储空间状态的技巧，保证 saveCurrent() 方法出错时，系统还处于可恢复的状态。

SecondaryNameNode.putFSImage() 上传新的命名空间镜像到名字节点。虽然 putFSImage() 也使用了辅助类 TransferFsImage.getFileClient() 方法，但在 HTTP 请求的 URL 中携带了参数 putimage。这是一个特殊的请求，名字节点处理这个请求时，会使用 TransferFsImage.getFileClient() 创建一个到第二名字节点的 HTTP 连接，获取“fsimage”文件。也就是说，这个 putFSImage() 调用，会分别在第二名字节点和名字节点上创建一个 HTTP 客户端，只有名字节点发起的 HTTP 连接被真正用于下载文件，第二名字节点的 HTTP 请求只用于通知名字节点，命名空间镜像已经准备好。这两个 HTTP 请求的 URL 如下：

```
http://<名字节点地址>:<名字节点的HTTP端口>/getimage?putimage=1&port=<第二名字节点的
  HTTP端口>&machine=<第二名字节点地址>&token=-32:1128241228:0:1341886363000:1341886
  060817

http://<第二名字节点地址>:<第二名字节点的HTTP端口>/getimage? getimage=1
```

putFSImage() 方法正常返回时，名字节点已经下载了最新的命名空间镜像，接下来，第二名字节点使用 rollFsImage() 让名字节点替换命名空间镜像。最后，第二名字节点清理工作区（实现在 FSImage.moveLastCheckpoint() 中），将“previous.checkpoint”目录删除，并将

“lastcheckpoint.tmp”目录名改为“previous.checkpoint”。

图 8-14 总结了 SecondaryNameNode.doCheckpoint() 工作过程中，第二名字节点执行的主要操作和磁盘目录文件结构的变化。

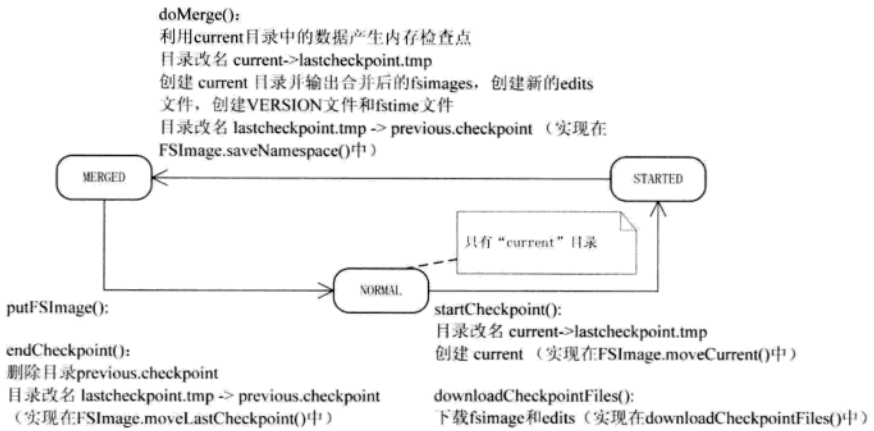



图 8-14　产生检查点时的第二名字节点根目录结构

和名字节点、数据节点不一样，由于复用了 FSImage 中的一些方法，第二名字节点没有非常清晰的节点存储状态，甚至出现了修改不存在目录的目录名的操作（endCheckpoint() 中对“lastcheckpoint.tmp”的修改）。但是，对临时目录“lastcheckpoint.tmp”和“previous.checkpoint”的使用，符合 FSImage 中的节点存储状态管理（后面会介绍），所以，当第二名字节点出现错误时，还是能根据这些临时目录，恢复到正常的工作状态。

### 2. 检查点的产生：名字节点的配合

第二名字节点产生检查点并应用于名字节点的过程中，除了必要的文件传输（提供原始的命名空间镜像和镜像编辑日志和获取检查点）以外，名字节点还提供了两个远程方法配合第二名字节点的流程，它们具体的实现都在类 FSImage 中。

❑rollEditLog()：用于第二名字节点下载命名空间镜像和编辑日志前。

❑NamenodeProtocol.rollFsImage()：用于第二名字节点上载新命名空间镜像后。

前面探讨过让名字节点产生检查点的可能性，名字节点保存命名空间镜像时，只能处于只读状态，即不能对目录树进行修改，以保证镜像中数据的一致性。引入第二名字节点后，在第二名字节点合并命名空间镜像和编辑日志时，名字节点上的元数据还是会发生变化，如何处理这些变化呢？ FSImage.rollEditLog() 的实现提供了答案：

```
CheckpointSignature rollEditLog() throws IOException {
   getEditLog().rollEditLog();
   ckptState = CheckpointStates.ROLLED_EDITS;
   return new CheckpointSignature(this);
}
```

FSImage.rollEditLog() 会调用 FSEditLog 的 rollEditLog() 方法，它会关闭当前的日志输出，即停止往“edits”中写入日志记录，然后打开新的日志文件“edits.new”，后续对名字节点的元数据修改，都会写入新日志文件中。注意，如果文件“edits.new”存在，rollEditLog() 会立即返回，这时，多次调用 rollEditLog()，系统的日志输出状态都不会改变。代码如下：

```
synchronized void rollEditLog() throws IOException {
   if (existsNew()) {
      ……
      return; //“edits.new”文件存在，不需要执行任何操作
   }

   close(); //关闭当前日志输出

   //打开到“edits.new”的日志输出
   for (Iterator<StorageDirectory> it = ……;it.hasNext();) {
      StorageDirectory sd = it.next();
      try {
         EditLogFileOutputStream eStream =
               new EditLogFileOutputStream(getEditNewFile(sd));
         eStream.create();
         editStreams.add(eStream);
      } catch (IOException e) {
         ……
      }
   }
}
```

接下来，FSImage.rollEditLog() 将名字节点的检查点状态修改为 CheckpointStates.ROLLED_EDITS，然后返回代表当前元数据状态的 CheckpointSignature 对象。CheckpointSignature 提供了检查点签名机制，签名可以唯一确定一个检查点，当然也就可以区分不同的检查点。

检查点签名除了 StorageInfo（CheckpointSignature 继承自 StorageInfo）中包含的存储系统信息结构版本号、存储系统标识和存储系统创建时间外，还包括检查点时间和日志时间。检查点时间即“fsimage”文件的版本号，它会一直递增，并保存在文件 ${dfs.name.dir}/current/fstime 中；日志时间以“edits”文件的修改时间为准，由于前面已经调用了 FSEditLog.rollEditLog()，所以在创建检查点过程顺利结束前，签名保持不变。

通过上述五个字段可以全局（不同集群间和同一集群内）确定一个检查点。第二名字节点必须保存获得的检查点签名，在上传检查点时，HTTP 请求 URL 的 token 参数中携带了该签名，以保证数据的一致性。

由上面讨论可知，当 ${dfs.name.dir}/current 目录下存在“edits.new”文件时，一般情况下，第二名字节点正在生成检查点。如图 8-15 所示。

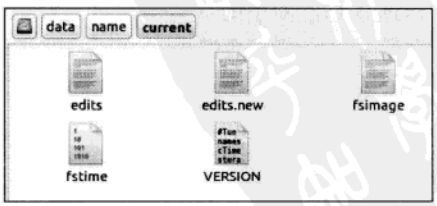



图 8-15 “current”目录下的日志文件“edits.new”

第二名字节点上载新命名空间镜像到名字节点后，使用远程方法 rollFsImage() 让名字节点更新检查点。

首先是状态检查，包括检查点状态必须是 CheckpointStates.UPLOAD_DONE，同时，文件系统的新镜像（“fsimage.ckpt”）存在，当前使用编辑日志文件“edits.new”保存日志等。

接下来，调用 FSEditLog.purgeEditLog()。purgeEditLog() 和 rollEditLog() 的功能正好相反，关闭往“edits.new”的日志输出后，将日志文件“edits.new”改名为“edits”，并重新打开到“edits”的日志。然后，新镜像“fsimage.ckpt”改名为“fsimage”，并输出新的“fstime”文件和“VERSION”文件。这个时候，名字节点“current”目录下保存的“fsimage”文件，已经包含了上一个检查点后的更新，即上一个检查点的命名空间镜像和该检查点上的编辑日志合并后的结果。当前日志文件（原来的“edits.new”）中包含的日志记录，应用到当前命名空间镜像后，就是当前名字节点上元数据的最新状态。最后，rollFSImage() 将检查点状态修改为 FSImage.CheckpointStates.START，结束名字节点上的检查点过程。代码如下：

```
void rollFSImage() throws IOException {
   if (ckptState != CheckpointStates.UPLOAD_DONE) {
      throw new IOException(……);
   }
   // 检查命名空间镜像存储和编辑日志存储处于正确的状态，
   // 即使用“edits.new”输出编辑日志，同时，命名空间镜像的名字是“fsimage.ckpt”
   ……

   editLog.purgeEditLog(); // “edits.new”改名为“edits”

   // 修改新命名空间镜像的名字
   for (Iterator<StorageDirectory> it = ……; it.hasNext();) {
      StorageDirectory sd = it.next();
      File ckpt = getImageFile(sd, NameNodeFile.IMAGE_NEW);
      File curFile = getImageFile(sd, NameNodeFile.IMAGE);
      if (!ckpt.renameTo(curFile)) {
         ……// 处理出错的情况
      }
   }

   // 输出新的“fstime”文件和“VERSION”文件
   ……

   ckptState = FSImage.CheckpointStates.START;
}
```

FSEditLog.purgeEditLog() 实现很简单，注意，它是一个 synchronized 方法，这个方法执行期间，日志记录可保存在缓冲区 bufCurrent 中，但不会写日志文件。代码如下：

```
synchronized void purgeEditLog() throws IOException {
   if (!existsNew()) {
      ……// 异常情况，“edits.new”不存在
```

```
        }
        close();// 关闭到“edits.new”的日志输出

        // 将“edits.new”改名为“edits”
        for (Iterator<StorageDirectory> it = ……; it.hasNext();) {
            StorageDirectory sd = it.next();
            if (!getEditNewFile(sd).renameTo(getEditFile(sd))) {
            }
        }

        open();// 重新打开编辑日志
    }
```

FSImage.rollEditLog() 将检查点的状态设置为 CheckpointStates.ROLLED_EDITS，但在 FSImage.rollFsImage() 方法的入口，名字节点需要判断当前检查点状态为 CheckpointStates.UPLOAD_DONE，而第二名字节点会顺序调用上述两个远程方法，是什么时候，FSImage 的检查点状态发生了改变？其实，在这两次交互间，第二名字节点需要通过 SecondaryNameNode.putFSImage() 上传新的命名空间镜像。名字节点顺利通过 HTTP 下载新镜像后，就会修改检查点状态。代码如下：

```
public class GetImageServlet extends HttpServlet {
    ……
    public void doGet(final HttpServletRequest request,
            final HttpServletResponse response)
            throws ServletException, IOException {
        ……    // 请大家忽略这段代码的缩进，把涉及权限控制的逻辑去除掉
        else if (ff.putImage()) {
            // 发起一个到第二名字节点的 HTTP GET 请求，下载新的命名空间镜像
            nnImage.validateCheckpointUpload(ff.getToken());
            …… {
                TransferFsImage.getFileClient(ff.getInfoServer(),
                        "getimage=1",nnImage.getFsImageNameCheckpoint());
                return null;
            }
            nnImage.checkpointUploadDone();
        }
    ……
```

处理第二名字节点上载检查点请求的 GetImageServlet.doGet() 方法，先通过 FSImage.validateCheckpointUpload() 校验检查点签名，在这个方法中，校验通过后，会将 FSImage 对象的检查点状态修改为 CheckpointStates.UPLOAD_START，即下载开始状态。

接下来，doGet() 通过 TransferFsImage.getFileClient() 从第二名字节点下载新的命名空间镜像，并保存到文件 ${dfs.name.dir}/current/fsimage.ckpt 中。最后执行的 FSImage.checkpointUploadDone()，修改 FSImage 的状态为 CheckpointStates.UPLOAD_DONE，表明检查点已经下载完毕，可以进行后续的 FSImage.rollFsImage() 操作了。

综合上面的讨论，检查点的状态转移和转移过程中涉及的主要操作如图 8-16 所示。

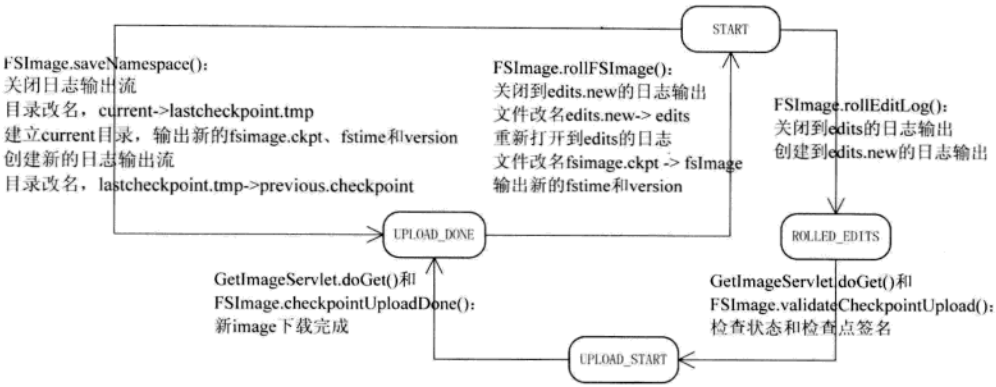



图 8-16　FSImage 的检查点转移

图中有我们介绍过的 FSImage.saveNamespace()，如图 8-17 所示。但这个方法并没有出现在名字节点的检查点过程中，而是出现在第二名字节点的流程中。在名字节点中，这个方法被其他多个 FSImage 成员函数使用，如名字节点启动时，会根据命名空间镜像和编辑日志，重构内存中的 HDFS 目录树及目录 / 文件的元信息，然后，内存中的元数据会通过 saveNamespace() 持久化到磁盘上，这相当于生成了一个检查点，也会触发图 8-16 中的检查点过程。

图 8-17　FSImage.saveNamespace() 的调用者

在名字节点产生检查点的过程中，FSImage 会创建不同的工作目录，保存状态，以便于系统出错时，名字节点可以执行相应的恢复过程。图 8-18 分析了名字节点存储空间的可能状态以及状态转移时使用的方法，供大家深入研究代码时使用。

和数据节点 Storage 状态机相比，最大差别在检查点部分（图的上半部分），涉及名字节点升级的状态和状态转移，由于数据节点也有类似过程，就不再详细讲解了。

### 3. 第二名字节点的启动

第二名字节点是 HDFS 集群中最简单的实体，它的启动过程也很简单。如果带参数“-checkpoint”或“-geteditsize”启动 SecondaryNameNode.main()，第二名字节点执行相应的功能后，就会退出。参数“-checkpoint”可以让名字节点执行一次检查点过程，即调用一次 SecondaryNameNode.doCheckpoint()。使用“-geteditsize”启动第二名字节点，可获得名字节点当前编辑日志的大小（通过和名字节点间的 IPC 接口）。代码如下：

```
publicstatic void main(String[] argv) throws Exception {
    ……
```

```
        Configuration tconf = new Configuration();
        if (argv.length >= 1) {
            SecondaryNameNode secondary = new SecondaryNameNode(tconf);
            int ret = secondary.processArgs(argv);
            System.exit(ret);
        }

        // 创建一个一直执行的 Deamon
        Daemon checkpointThread = new Daemon(new SecondaryNameNode(tconf));
        checkpointThread.start();
    }
```

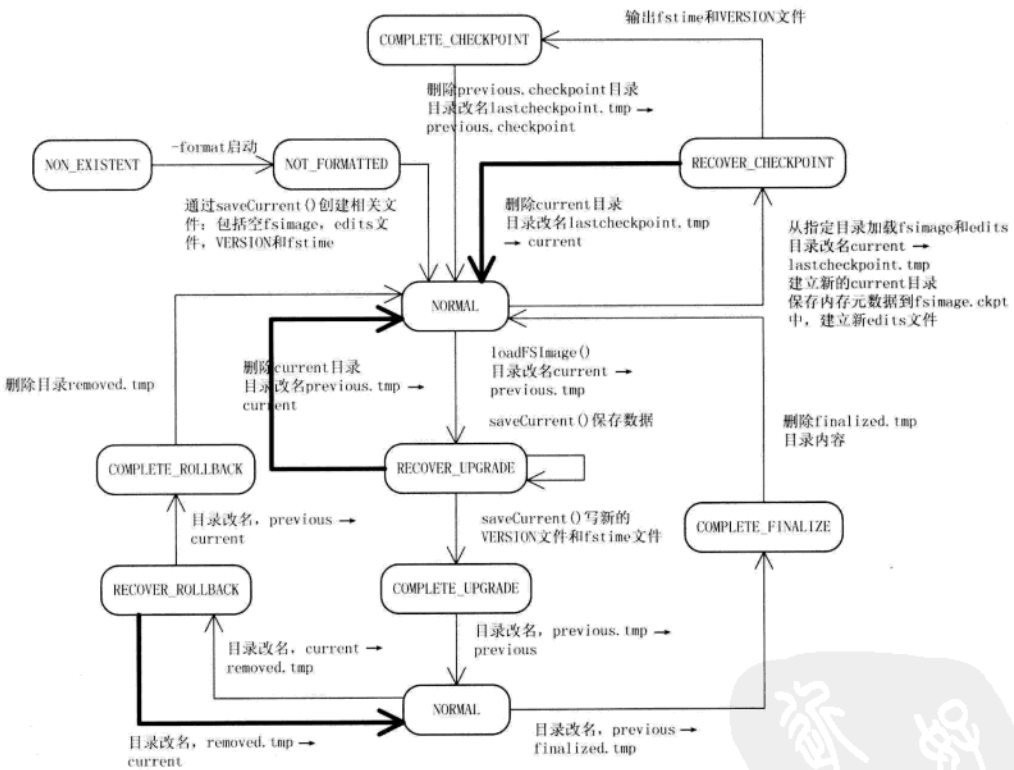



图 8-18 名字节点 Storage 状态机

比较常见的情景是不带参数启动第二名字节点，这时，它会定期产生检查点。main() 方法在创建 SecondaryNameNode 对象时，会通过成员函数 initialize() 读入配置并创建工作时需要的 CheckpointStorage 对象、HTTP 服务器和访问远程接口 NamenodeProtocol 的客户端。

配置参数 ${fs.checkpoint.period} 和 ${fs.checkpoint.size} 决定了检查点执行的时间表。第二名字节点每小时产生一个检查点（前一个参数，默认值是 1 小时），如果编辑日志已经达到 64MB（由后面的参数决定），则每隔 5 分钟检查一次。

SecondaryNameNode.run() 调用 doWork() 执行第二名字节点的主循环。主循环很简单，就是执行上面的策略，根据两个配置参数决定是否调用 doCheckpoint() 方法，其中所需的名字节点编辑日志大小是通过远程调用 getEditLogSize() 获得的。代码如下：

```
public void doWork() {
   long period = 5 * 60; //5 分钟
   long lastCheckpointTime = 0;
   if (checkpointPeriod < period) {
      period = checkpointPeriod;
   }

   while (shouldRun) {
      try {
         Thread.sleep(1000 * period);
      } catch (InterruptedException ie) {
      ......
      try {
         ......
         long now = System.currentTimeMillis();
         long size = namenode.getEditLogSize(); // 获得名字节点编辑日志大小
         if (size >= checkpointSize
               || now >= lastCheckpointTime + 1000 * checkpointPeriod) {
            doCheckpoint();
            lastCheckpointTime = now;
         }
      }
      ......
   }
}
```

停止第二名字节点时不需要执行任何附加的逻辑，shutdown() 方法只在构造 SecondaryNameNode 对象，调用 initialize() 失败时使用。

### 8.1.4　FSDirectory 的实现

门面（Facade）模式是一种结构模式，其意图是为子系统中的一组接口提供一个一致的界面，使得使用这一个子系统变得更加容易。名字节点的业务逻辑对目录树操作，会涉及目录树节点的增、删、查、改，有些操作需要记录到编辑日志中，同时，系统启动时，还需要处理命名空间镜像和编辑日志，这些涉及目录树元数据的操作，需要和多个对象打交道才能达到目的。为了屏蔽子系统的复杂性，让名字节点的其他逻辑更好地操作目录树，减低其他子系统和名字节点第一关系子系统的耦合，HDFS 的设计者引入了 FSDirectory 作为门面，为复杂的第一关系子系统提供一个简单的接口。名字节点的其他子系统使用 FSDirectory 操作目录树。而 FSDirectory 知晓这些涉及目录树的操作如何分派到子系统的各个对象中，并实现相应的处理逻辑。

FSDirectory 拥有大量的方法列表，却只有少量的成员变量，其中重要的有：

```
class FSDirectory implements FSConstants, Closeable {
   final FSNamesystem namesystem;
   final INodeDirectoryWithQuota rootDir;
   FSImage fsImage;
   private boolean ready = false;
   ……
}
```

❑namesystem：类型为 FSNamesystem，FSNamesystem 是一个非常复杂的类，它是整个名字节点的门面，FSDirectory 使用这个成员变量，访问数据块管理的相关功能。

❑rootDir：整个文件系统的根目录，它的类型为 INodeDirectoryWithQuota。

❑fsImage：为 FSDirectory 提供了命名空间镜像和编辑日志的相关功能。

❑ready：布尔变量，当 FSDirectory.loadFSImage() 完成命名空间镜像和编辑日志加载后，变量修改为 true，这时，可以对目录树进行操作。

FSDirectory 有 60 多个成员函数，基本上远程接口 ClientProtocol 中的方法，都能在 FSDirectory 找到对应。下面以几个典型的 FSDirectory 方法为例，对它进行讲解。

### 1. addToParent() 方法

FSImage.loadFSImage() 在读入命名空间镜像，处理 INode 信息时，每读取一条记录，都会调用 FSDirectory.addToParent() 往目录树中添加对应的 INode。这个方法有大量的参数，根据这些参数，addToParent() 构造目录或文件，并使用 INodeDirectory.addToParent() 将节点添加到它的父节点（参数 parentINode）中。

注意，如果被添加的是一个文件，那么必须使用 FSNamesystem 的成员变量 blocksMap 的 addINode() 方法，添加这个文件的每一个数据块到名字节点第二关系管理系统的数据块映射对象中。FSDirectory 大量的方法都会涉及数据块和数据节点间的关系维护机制，下一节会正式对它进行研究，相关代码如下：

```
INodeDirectory addToParent(String src, INodeDirectory parentINode,
      PermissionStatus permissions, Block[] blocks,
      short replication,long modificationTime, long atime,
      long nsQuota, long dsQuota,long preferredBlockSize) {
   // 创建节点
   INode newNode;
   if (blocks == null) {// 输入信息是一个目录
      ……// 根据情况创建 INodeDirectoryWithQuota 或 INodeDirectory
   } else {// 输入信息是一个文件
      newNode = new INodeFile(permissions, blocks.length, replication,
            modificationTime, atime, preferredBlockSize);
   }
   // 将节点加到它的父节点
   INodeDirectory newParent = null;
   synchronized (rootDir) {
      try {
         newParent = rootDir.addToParent(src, newNode,
               parentINode, false);
```

```
            ......
        } catch (FileNotFoundException e) {
            ......
        }
        ......
        if (blocks != null) {
            int nrBlocks = blocks.length;
            // 文件数据块映射
            INodeFile newF = (INodeFile) newNode;
            for (int i = 0; i < nrBlocks; i++) {
                newF.setBlock(i,
                        namesystem.blocksMap.addINode(blocks[i], newF));
            }
        }
    }
    return newParent;
}
```

### 2. delete() 方法

INodeDirectory.delete() 可删除目录树上的一项，它的逻辑很简单，调用 unprotectedDelete() 方法执行实际的删除操作，并往日志文件中记录该操作。对目录树修改的大部分方法都有对应的 unprotected*() 方法，这些方法操作目录树时不会记录日志，所以，也应用于 FSDirectory.loadFSImage() 的日志恢复过程中。代码如下：

```
boolean delete(String src) {
    ......
    long now = FSNamesystem.now();
    int filesRemoved = unprotectedDelete(src, now);
    ......
    fsImage.getEditLog().logDelete(src, now);
    ......
}
```

成员函数 unprotectedDelete() 先将路径规范化，然后调用 INodeDirectory.getExistingPathINodes() 获得从根节点到被删除节点经过的各级 INode，也就获得了被删除文件 / 目录的父节点（该数组的倒数第二项，第一项是被删除节点）。然后在父节点上调用 removeChild() 方法删除 INode 对象，更新父节点的“修改时间”属性，接下来就是使用前面介绍过的 INode.collectSubtreeBlocksAndClear()，获得被删除节点为根的子目录树下所有文件拥有的数据块，通过 FSNamesystem.removePathAndBlocks() 方法，删除所有的数据块和可能的租约。代码如下：

```
int unprotectedDelete(String src, long modificationTime) {
    src = normalizePath(src);// 规范化路径

    synchronized (rootDir) {
        INode[] inodes = rootDir.getExistingPathINodes(src);
        INode targetNode = inodes[inodes.length - 1];

        if (targetNode == null) { // 目录不存在
```

```
            ......
        } else if (inodes.length == 1) { // 目录是根
            ......
        } else {
            try {
                // 删除子节点
                removeChild(inodes, inodes.length - 1);
                // 更新父节点的“修改时间”属性
                inodes[inodes.length - 2]
                        .setModificationTime(modificationTime);
                // 获取该 INode 下所有的数据块，并删除
                ArrayList<Block> v = new ArrayList<Block>();
                int filesRemoved = targetNode
                        .collectSubtreeBlocksAndClear(v);
                namesystem.removePathAndBlocks(src, v);
                ......
                return filesRemoved;
            } catch (IOException e) {
    ......
}

private INode removeChild(INode[] pathComponents, int pos) {
    INode removedNode = ((INodeDirectory)
            pathComponents[pos - 1]).removeChild(pathComponents[pos]);
    ......
}
```

### 3. getFileInfo() 方法

INodeDirectory.delete() 的删除操作实现不算复杂，但也涉及了数据块和数据节点间的关系维护机制。相对于 delete()，getFileInfo() 是一个很简单的方法，它被用于实现远程方法 ClientProtocol.getFileInfo()。和 unprotectedDelete () 方法一样，它也需要对路径进行规范化，然后获得路径对应的 INode，调用 createFileStatus() 构造并返回所需的 HdfsFileStatus 对象。代码如下：

```
HdfsFileStatus getFileInfo(String src) {
    String srcs = normalizePath(src);
    synchronized (rootDir) {
        INode targetNode = rootDir.getNode(srcs);
        if (targetNode == null) {
            return null;
        } else {
            return createFileStatus(HdfsFileStatus.EMPTY_NAME, targetNode);
        }
    }
}

private static HdfsFileStatus createFileStatus(byte[] path, INode node) {
    return new HdfsFileStatus(node.isDirectory() ?
            0 : ((INodeFile) node).computeContentSummary().getLength(),
            node.isDirectory(),……, path);
}
```

### 4. setOwner() 方法

setOwner() 方法用于修改文件主标识符和用户组标识符，这个方法不涉及数据块，它的实现非常简单，如下：

```
void setOwner(String src, String username, String groupname)
                throws IOException {
   unprotectedSetOwner(src, username, groupname);
   fsImage.getEditLog().logSetOwner(src, username, groupname);
}

void unprotectedSetOwner(String src, String username, String groupname)
                throws FileNotFoundException {
   synchronized (rootDir) {
      INode inode = rootDir.getNode(src);
      if (inode == null) //找不到文件
         throw new FileNotFoundException("File does not exist: " + src);
      if (username != null) {
         inode.setUser(username); //修改文件主标识符
      }
      if (groupname != null) {
         inode.setGroup(groupname); //修改用户组标识符
      }
   }
}
```

**注意**　上述四个例子，分别在HDFS的目录树上进行增、删、查、改等典型操作，INodeDirectory提供的目录树子系统外部接口，基本都可以归为上述四类操作。由于操作种类多，INodeDirectory也就拥有了大量的方法，但这些方法都很简单，有些方法会涉及数据块和数据节点的对应关系，如delete()，在必要的时候，会继续对它们进行分析。

## 8.2　数据块和数据节点管理

名字节点维护着HDFS文件系统中两个最重要的关系：

❑HDFS文件系统的文件目录树，以及文件的数据块索引，即每个文件对应的数据块列表；

❑数据块和数据节点的对应关系，即某一数据块保存在那些数据节点的信息。

上一节已经介绍了与目录树相关的名字节点第一关系的管理。现在，转入第二个关系，开始研究数据块和数据节点的对应关系。这个关系和第一个关系不同，“fsimage”文件不会记录数据块保存在那些数据节点的信息，而是在数据节点加入集群时，由数据节点提供它所包含的块列表动态建立起来的。和第一关系相同的是，随着系统的运行，数据块和数据节点都会发生变化，上述对应关系也就会不断地发生变化，名字节点必需跟踪、维护这个关系，以保证系统的正常工作。

### 8.2.1 数据结构

和名字节点第一关系相比，数据块和数据节点的对应关系更复杂，包含更多的状态，也就需要更多的类来抽象相关的信息。接下来从 INodeFile 出发，逐步考察名字节点第二关系中的这些数据结构。INodeFile 是一个很好的开始点，它通过成员变量 blocks 保存文件拥有的数据块，自然也就和名字节点第二关系发生关联。

注意，INodeFile.blocks 也是名字节点第一关系中唯一涉及第二关系的类成员变量。它的类型是 BlockInfo，BlockInfo 是 BlocksMap 的内部类。

#### 1. BlocksMap 和 DatanodeDescriptor

BlocksMap（数据块映射）管理着名字节点上数据块的元数据，包括数据块所属的 INode 和那些数据节点保存数据块等信息。也就是说，如果要定位某个数据块在哪些数据节点上，只需要访问名字节点的 BlocksMap 对象。

DatanodeDescriptor（数据节点描述符）是名字节点中对数据节点的抽象，它继承自 org.apache.hadoop.hdfs.protocol.DatanodeInfo，添加了大量名字节点工作时需要的数据节点信息。

BlocksMap 虽然保存名字节点第二关系中最重要的数据，但它的实现并不复杂：BlocksMap 使用 org.apache.hadoop.hdfs.util.GSet，一个比较特殊的集合类型，存放（对象 blocks）名字节点管理的所有数据块。GSet 特殊的地方在于它是个集合，但提供了类似映射的功能，如 blocks.get()，可以根据数据块对象获得它对应的 BlockInfo 对象。

BlockInfo（数据块信息）是 BlocksMap 的内部类，也是 Block 的子类，它增加了数据块所属的 INode 信息和保存该数据块的数据节点信息。INode 信息保存在 BlockInfo.inode 中，但数据块所在数据节点的 DatanodeDescriptor 对象的保存就比较特殊了，它保存在类型为 Object 的数组 triplets 中，而不是保存在一个 DatanodeDescriptor 数组中。而且数组 triplets 除了保存数据节点信息（第 i 个数据节点的信息保存在元素 triplets[3*i] 中），还以双向链表的形式保存该数据节点上其他两个数据块的数据块信息对象，即 triplets[3*i+1] 保存了当前数据块的前一个数据块的 BlockInfo，triplets[3*i+2] 保存了后面一个数据块的 BlockInfo。也就是说，沿着 triplets[3*i+1] 或 triplets[3*i+2]，可以遍历某个数据节点拥有的所有数据块的 BlockInfo 信息。代码如下：

```
class BlocksMap {
   static class BlockInfo extends Block
         implements LightWeightGSet.LinkedElement {
      private INodeFile          inode;
      ......
      //triplets[3*i] 保存 DatanodeDescriptor 对象
      //triplets[3*i+1] 和 triplets[3*i+2] 分别保存同一数据节点上的其他
      // 两个（前、后）数据块的 BlockInfo，通过这种方法构成了一个链表
      private Object[] triplets;
      ......
   }
   ......
```

```
    private GSet<Block, BlockInfo> blocks;
    ……
}

// 它的一个实现是 org.apache.hadoop.hdfs.util.LightWeightGSet
// 标准的链地址法（拉链法）哈希表实现
public interface GSet<K, E extends K>extends Iterable<E> {
    int size();// 集合大小
    ……
    E get(K key);
    ……
}
```

图 8-19 是 BlocksMap 的类图，无论是 BlocksMap 还是内部类 BlockInfo，它们的方法实现都很简单。BlockInfo 除了访问成员变量的 get*()/set*() 外，需要关注的就是处理数据块双向链表的方法，这些方法以 list 开头，实现都不是很复杂。

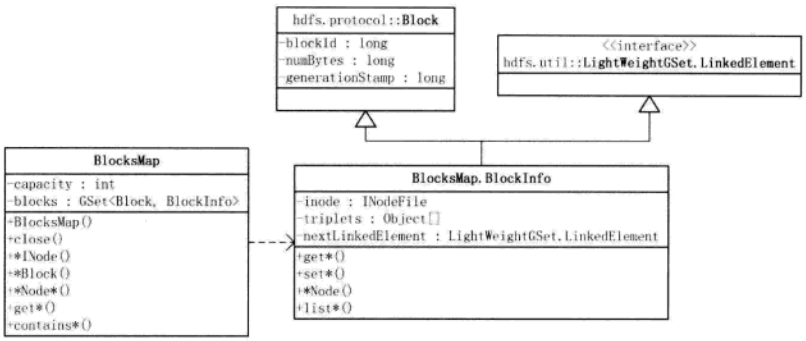



图 8-19　BlocksMap 的类图

BlockInfo 的 triplets[3*i] 保存数据节点信息，类型为 DatanodeDescriptor，它是 DatanodeInfo 的子类，而 DatanodeInfo 又是 DatanodeID 的子类，同时，DatanodeDescriptor 也拥有一些内部类，形成了一个略复杂的类图，如图 8-20 所示。在第 6 章已经接触过 DatanodeInfo，通过它可以定位一个数据节点，并了解该节点的一些状态，在数据节点和名字节点、客户端和名字节点的 IPC 接口中都有 DatanodeInfo 的应用。

和 BlockInfo 一样，DatanodeDescriptor 添加了一些名字节点工作时需要的数据。这些数据可以分为如下三类：

（1）数据节点的状态

包括 isAlive（具体含义请参考代码注释）、firstBlockReport 和 decommissioningStatus、currApproxBlocksScheduled 等，其中 decommissioningStatus（类型为 DatanodeDescriptor.DecommissioningStatus）比较特殊，它只在节点处于撤销状态时使用；currApproxBlocksScheduled 等几个成员变量用于估计数据节点的负载，为写文件操作分配数据块或进行数据块复制时，会根据节点负载信息，优先选择比较空闲的节点作为目标。

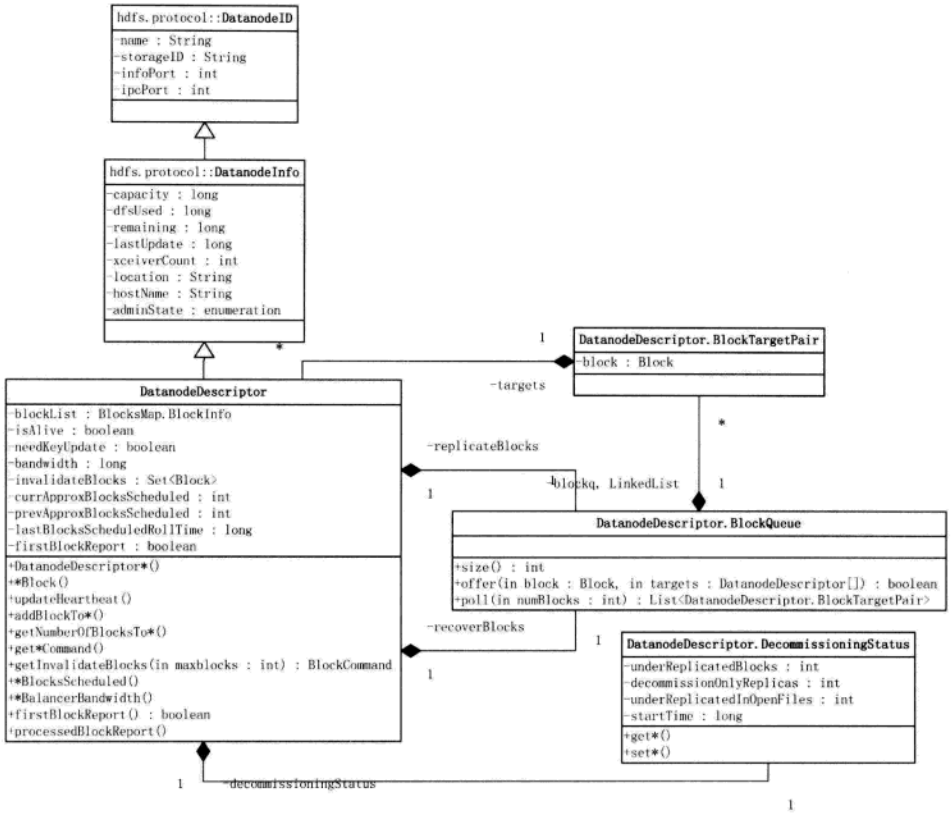



图 8-20 DatanodeDescriptor 的类图

（2）用于产生到该数据节点的名字节点指令

包括 bandwidth、replicateBlocks、recoverBlocks 和 invalidateBlocks 四个成员变量，它们分别用于产生均衡器工作带宽更新（DNA_BALANCERBANDWIDTHUPDATE）、数据块复制（DNA_TRANSFER）、数据块恢复（DNA_RECOVERBLOCK）和数据块删除（DNA_INVALIDATE）的名字节点指令，以数据块删除为例，成员变量 invalidateBlocks 保存了这个数据节点上等待删除的数据块，在下次心跳中，名字节点（可能）会产生一个 DNA_INVALIDATE 操作，让数据节点删除集合 invalidateBlocks 中的一些数据块。成员变量 replicateBlocks 和 recoverBlocks 的类型是数据块队列 BlockQueue，它的元素类型是 BlockTargetPair。如类名所示，BlockTargetPair 包含了数据块和目标数据节点（列表）两项信息，对于复制，目标数据节点是复制的目标节点，对于数据块恢复，目标数据节点则是参与恢复过程的数据节点。

（3）成员变量 blockList

它是该数据节点保存的数据块队列的头节点，队列元素类型为 BlocksMap.BlockInfo。

上述变量的声明如下：

```
public class DatanodeDescriptor extends DatanodeInfo {
   //保存了撤销节点的一些状态信息。只能在节点处于撤销状态时使用
   DecommissioningStatus decommissioningStatus =
         new DecommissioningStatus();

   //数据块和目标数据节点
   public static class BlockTargetPair {
      public final Block block;// 数据块
      public final DatanodeDescriptor[] targets;// 目标数据节点
      ……
   }

   private static class BlockQueue {
      private final Queue<BlockTargetPair> blockq =
            new LinkedList<BlockTargetPair>();
      ……
   }

   private volatile BlockInfo blockList = null;
   protected boolean isAlive = false;// 节点是否失效（活着）
   ……
   // 数据节点的 ${dfs.balance.bandwidthPerSec} 参数，可以通过下面的命令设置
   //"dfsadmin -setBalanacerBandwidth <newbandwidth>",
   // 并用 DatanodeCommand 更新到数据节点，更新命令发出后，bandwidth 会被设置为 0
   private long bandwidth;

   //该数据节点上等待复制的数据块
   private BlockQueue replicateBlocks = new BlockQueue();
   //该数据节点上等待租约恢复的数据块
   private BlockQueue recoverBlocks = new BlockQueue();
   //该数据节点上等待被删除的数据块
   private Set<Block> invalidateBlocks = new TreeSet<Block>();

   //下面变量用于估计往这个数据节点进行写操作的数据块数目（实际值比这个略高）
   private int currApproxBlocksScheduled = 0;
   private int prevApproxBlocksScheduled = 0;
   private long lastBlocksScheduledRollTime = 0;
   private static final int BLOCKS_SCHEDULED_ROLL_INTERVAL = 600*1000;

   //是否发送了第一个数据块上报（远程调用）
   private boolean firstBlockReport = true;

   class DecommissioningStatus {
      int underReplicatedBlocks;
      int decommissionOnlyReplicas;
      int underReplicatedInOpenFiles;
      long startTime;
      ……
```

```
    }
    ……
}
```

明确了上述变量的意义，DatanodeDescriptor 成员方法的实现也就很好理解了。下面介绍两个典型方法。

当数据节点成功接收到一个数据块后，会通过远程方法 blockReceived() 报告名字节点。名字节点自然需要把它添加/更新到对应数据节点的 DatanodeDescriptor 对象中，使用的就是 addBlock() 方法。它首先调用 BlockInfo.addNode() 将对象自己，也就是 DatanodeDescriptor 添加到数据块所属数据节点列表中；接着，又调用 BlockInfo 的 listInsert() 方法，将数据块插入到数据节点管理的数据块列表中。DatanodeDescriptor.addBlock() 方法充分体现了 BlocksMap 和 DatanodeDescriptor 间的复杂关系，代码如下：

```
public class DatanodeDescriptor extends DatanodeInfo {
    ……
    boolean addBlock(BlockInfo b) {
        if(!b.addNode(this))
            return false;
        blockList = b.listInsert(blockList, this); //加到列表中
        return true;
    }
    ……
}

class BlocksMap {
    static class BlockInfo extends…… {
        ……
        boolean addNode(DatanodeDescriptor node) {
            if(findDatanode(node) >= 0) // 节点已经存在于数据块所在数据节点列表中
                return false;
            int lastNode = ensureCapacity(1); // 在 triplets 中找到一个空闲位置
            setDatanode(lastNode, node);//triplets[3*lastNode]
            setNext(lastNode, null);//triplets[3*lastNode+2]
            setPrevious(lastNode, null);//triplets[3*lastNode+1]
            return true;
        }
        ……
        BlockInfo listInsert(BlockInfo head, DatanodeDescriptor dn) {
            ……
            this.setPrevious(dnIndex, null);
            this.setNext(dnIndex, head);
            if(head != null)
                head.setPrevious(head.findDatanode(dn), this);
            return this;
        }
        ……
    }
    ……
}
```

DatanodeDescriptor.getReplicationCommand() 用于产生一条数据块复制指令，它在队列 replicateBlocks 中获取最多 maxTransfers 个元素，然后根据这些元素构造 BlockCommand 对象。这个方法很简单，在 DatanodeDescriptor 中，类似的方法还有 getLeaseRecoveryCommand()、getInvalidateBlocks() 等，相关代码如下：

```
public class DatanodeDescriptor extends DatanodeInfo {
   ……
   BlockCommand getReplicationCommand(int maxTransfers) {
      //replicateBlocks 中的前面 maxTransfers 项（最多）
      List<BlockTargetPair> blocktargetlist
            = replicateBlocks.poll(maxTransfers);
      return blocktargetlist == null? null:
         new BlockCommand(DatanodeProtocol.DNA_TRANSFER, blocktargetlist);
   }
   ……
}
```

### 2. 数据块副本状态

BlocksMap 和 DatanodeDescriptor 一起保存名字节点第二关系，即数据块和数据节点的对应关系，但名字节点上对数据块的管理，还需要其他一些数据结构的支持。本节研究名字节点上的数据块副本状态（准确讲，是针对特定数据块副本的当前操作），和管理副本状态时使用的类。同时，开始涉及 org.apache.hadoop.hdfs.server.namenode.FSNamesystem 的实现原理分析，这个类是整个名字节点的门面（Facade），拥有冗长的代码和众多的成员变量和成员方法，如它的类型为 FSDirectory 的成员变量 FSNamesystem.dir，就用于操作 HDFS 的目录树。

当系统以完美状态、在没有任何故障情况下工作时，数据块副本随着客户端写文件数据产生，并在文件删除时被删除。但集群在工作时，特别是上千台服务器组成的系统工作时，会有比较高的故障率，这时会出现各种各样的意外，如：

- ❑数据节点在执行写操作的过程中崩溃，破坏了数据流管道，客户端进行数据块恢复。
- ❑客户端写数据过程中崩溃，长时间没有恢复，名字节点进行租约恢复（对数据节点来说，是一个名字节点发起的数据块恢复）。
- ❑数据节点磁盘损坏，导致上面保存的数据块副本永久丢失。
- ❑数据节点出现故障，然后恢复。

上述情况都会对数据节点上的数据块副本状态产生影响，如数据块恢复就是数据块副本处于的一种状态（或施加在副本上的当前操作）。有些时候，一个正常的操作也会引起数据块副本的状态变化，如降低某个文件的副本数会导致一些数据块副本被删除。

下面是 FSNamesystem 中涉及数据块管理的成员变量：

```
class FSNamesystem implements…… {
   ……
   // 等待复制的数据块副本数
   volatile long pendingReplicationBlocksCount = 0L;
```

```
    // 已经损坏的数据块副本数
    volatile long corruptReplicaBlocksCount = 0L;
    // 等待复制的数据块副本数
    volatile long underReplicatedBlocksCount = 0L;
    // 进入复制调度的数据块副本数
    volatile long scheduledReplicationBlocksCount = 0L;
    // 多余数据块副本数
    volatile long excessBlocksCount = 0L;
    // 等待删除数据块副本数
    volatile long pendingDeletionBlocksCount = 0L;
    ……
    // 名字节点的 BlocksMap
    final BlocksMap blocksMap = new BlocksMap(DEFAULT_INITIAL_MAP_CAPACITY,
          DEFAULT_MAP_LOAD_FACTOR);

    // 已经损坏的数据块副本
    public CorruptReplicasMap corruptReplicas = new CorruptReplicasMap();

    // 提供了数据节点存储标识 StorageID 到 DatanodeDescriptor 的映射
    NavigableMap<String, DatanodeDescriptor> datanodeMap =
          new TreeMap<String, DatanodeDescriptor>();

    // 数据节点到该节点上等待删除的数据块副本的映射
    private Map<String, Collection<Block>> recentInvalidateSets =
          new TreeMap<String, Collection<Block>>();

    // 数据节点到该节点上多余数据块副本的映射
    Map<String, Collection<Block>> excessReplicateMap =
          new TreeMap<String, Collection<Block>>();
    ……

    // 等待复制，准备生成复制请求的数据块副本
    private UnderReplicatedBlocks neededReplications =
          new UnderReplicatedBlocks();

    // 已经生成复制请求的数据块副本
    private PendingReplicationBlocks pendingReplications;

    // 租约管理器
    public LeaseManager leaseManager = new LeaseManager(this);
}
```

基本上对特定数据块副本的下一步操作，FSNamesystem 都有一个成员变量保存相应的数据块和执行操作时需要的附加信息，有些还有计数器存放等待操作的数据块数目，具体内容如下：

- ❑corruptReplicas：保存损坏的数据块副本，如数据节点的数据块扫描器发现的错误数据块副本。
- ❑recentInvalidateSets：无效数据块副本，即等待删除的数据块副本，文件删除时，该文件拥有的数据块副本都会变成无效，一般来说，损坏的数据块副本也是无效的。

❑excessReplicateMap：多余副本，减低文件的副本数（可通过“hadoop fs –setrep”命令）会产生多余副本，名字节点启动时也可能发现多余副本，多余副本由名字节点在数据块的多个副本中选择得到。

❑neededReplications：等待复制，准备生成复制请求的数据块副本，数据块复制是为了让数据块的副本数满足文件的副本系数。

❑pendingReplications：已经生成复制请求的数据块副本保存在 pendingReplications 中，也就是说，当复制请求生成后，数据块的信息会从 neededReplications 取出，放入 pendingReplications 中。

❑leaseManager：租约管理器，间接保存了处于构建状态或恢复状态的数据块的信息。

为了方便后续讨论，上面代码包含了 blocksMap 和 datanodeMap 两个成员变量。blocksMap 是名字节点中唯一的 BlocksMap 实例；datanodeMap 保存名字节点当前管理的所有数据节点，并提供了通过数据节点存储标识快速查找对应 DatanodeDescriptor 对象的能力。

上面列表中的 recentInvalidateSets 和 excessReplicateMap 的类型比较简单，是一个数据节点存储标识到一组数据块对象（Collection<Block>）的映射。实现中，recentInvalidateSets 使用 java.util.HashSet，excessReplicateMap 则使用 java.util.TreeSet，作为具体的映射实现。其他变量如 corruptReplicas、neededReplications 和 pendingReplications 都定义了对应的类，保存需要的数据（下一节讨论 LeaseManager）。

FSNamesystem.corruptReplicas 的类型是 CorruptReplicasMap，保存数据节点发现的损坏数据块副本。如图 8-21 所示，由于数据块往往有多个副本，只有该数据块的所有副本都损坏了，系统才认为数据块损坏，当部分副本出现损坏时，名字节点会进行数据块复制，复制完好的数据块副本直到数据块副本数恢复正常。CorruptReplicasMap 用于配合上述过程，它通过成员变量 corruptReplicasMap，记录了损坏副本以及副本所在的（一组）数据节点，当发现新的损坏副本时，通过 addToCorruptReplicasMap() 方法添加副本信息到 CorruptReplicasMap 对象中；当损坏数据块从数据节点上删除时，则调用 removeFromCorruptReplicasMap() 删除记录。

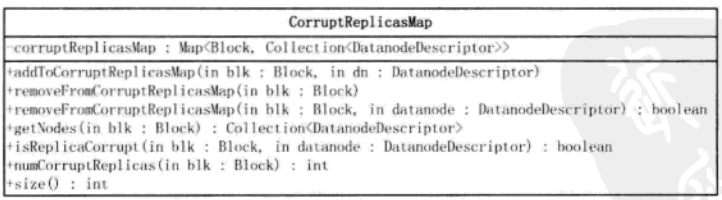



图 8-21　CorruptReplicasMap 的类图

和 CorruptReplicasMap 相比，UnderReplicatedBlocks 的实现比较复杂，如图 8-22 所示。

等待复制的数据块信息保存在变量 priorityQueues 中。priorityQueues 是一个列表，有三个类型为 TreeSet 的元素，对应不同的等待优先级，等待复制集合 TreeSet 的元素是 Block，也就是说，priorityQueues 保存了三个数据块集合。

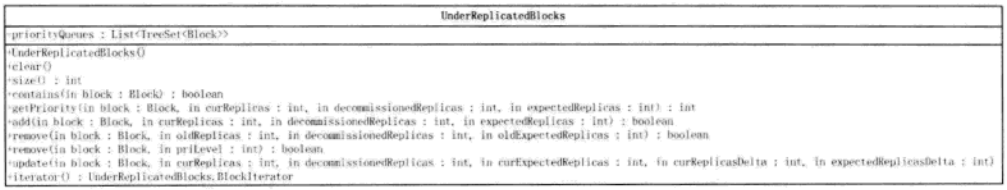



图 8-22 UnderReplicatedBlocks 的类图

待复制的数据块的优先级如何确定呢？当复制源节点是一个等待撤销的数据节点或者数据块只剩一个副本的情况下，优先级最高；副本数不到期望值的 1/3 时，优先级次之，其他情况下，优先级最低。优先级高的数据块最先得到复制。上述优先级计算实现在 UnderReplicatedBlocks 的私有方法 getPriority() 中，往对象中增加、修改复制项时，都需要调用这个方法。代码如下：

```
class UnderReplicatedBlocks implements Iterable<Block> {
   static final int LEVEL = 3;
   private List<TreeSet<Block>> priorityQueues =
         new ArrayList<TreeSet<Block>>();
   ……
   private int getPriority(Block block, int curReplicas,
         int decommissionedReplicas, int expectedReplicas) {
      if (curReplicas < 0 || curReplicas >= expectedReplicas) {
         return LEVEL; // 异常情况
      } else if (curReplicas == 0) {
         if (decommissionedReplicas > 0) {// 处理撤销数据节点的特殊情况
            return 0;
         }
         return 2;// 保持在队列中
      } else if (curReplicas == 1) {
         return 0; // 副本数只有一个，优先级最高
      } else if (curReplicas * 3 < expectedReplicas) {
         return 1;// 副本数不到副本系数的 1/3
      } else {
         return 2;
      }
   }
   ……
}
```

等待复制的数据块从 UnderReplicatedBlocks 对象中读取出来以后，会根据集群数据节点负载等情况，生成复制请求，并将请求放入源数据节点中，即保存在该节点的 DatanodeDescriptor 对象的成员变量 replicateBlocks 中。注意，这个复制请求可能因为多种原因执行失败，所以，名字节点会将复制请求保存在 FSNamesystem.pendingReplications 中，如图 8-23 所示。

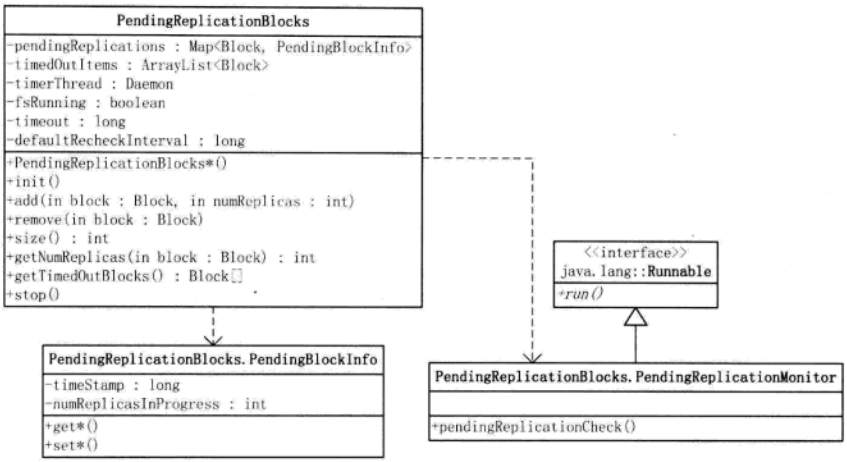



图 8-23　PendingReplicationBlocks 的类图

PendingReplicationBlocks 将这些请求信息保存在成员变量 pendingReplications 中，它是一个数据块到数据块复制信息 PendingBlockInfo 的映射。PendingBlockInfo 中保存的信息包括复制请求的产生时间（timeStamp）和复制副本数（numReplicasInProgress）。如果数据节点的复制操作成功结束，保存在 PendingReplicationBlocks 中的复制请求会被删除，否则，请求会在定期检查过程中被放入成员变量 timedOutItems。定期检查的检查时间由 defaultRecheckInterval 和 timeout 共同决定，它们的默认值都是五分钟（timeout 可调）。其中 defaultRecheckInterval 是对象线程（实现在内部类 PendingReplicationMonitor 中）执行检查方法 pendingReplicationCheck() 的间隔时间，当复制请求的产生时间和当前时间间隔大于 timeout，就会放入成员变量 timedOutItems 中。所以，在默认情况下，如果复制操作没有成功执行，在 5~10 分钟后，这些操作会进入 timedOutItems，最终又被插入到 UnderReplicatedBlocks 对象中，以重新产生复制请求。通过这样的手段，名字节点可以防止数据块复制请求执行失败造成的复制请求丢失问题。上述 PendingReplicationBlocks 的主要成员变量定义如下：

```
class PendingReplicationBlocks {
   private Map<Block, PendingBlockInfo> pendingReplications;
   private ArrayList<Block> timedOutItems;
   Daemon timerThread = null;
   private volatile boolean fsRunning = true;
   private long timeout = 5 * 60 * 1000;
   private long defaultRecheckInterval = 5 * 60 * 1000;
   ……
   static class PendingBlockInfo {
      private long timeStamp;
      private int numReplicasInProgress;
      ……
```

```
        }
        ......
    }
```

图 8-24 是一个不严格的名字节点数据块副本状态转移图，它总结了这一小节的内容。

其中，“复制中”、“等待复制”、“损坏”和“等待删除”等状态已经体现在 corruptReplicas、recentInvalidateSets、excessReplicateMap、neededReplications 和 pendingReplications 等 FSNamesystem 成员变量中。状态“正在写数据”和“恢复”间接保存在租约管理器中。“撤销节点副本”是指位于正在撤销或被撤销数据节点上的副本，可通过副本所在的数据节点状态确定。而“等待数据节点信息”是名字节点刚启动时数据块副本的一种特殊状态，在这种状态下，文件系统中包含的数据块已经通过命名空间镜像和编辑日志加载到内存中，但还不能定位到这些数据块所在的数据节点信息。

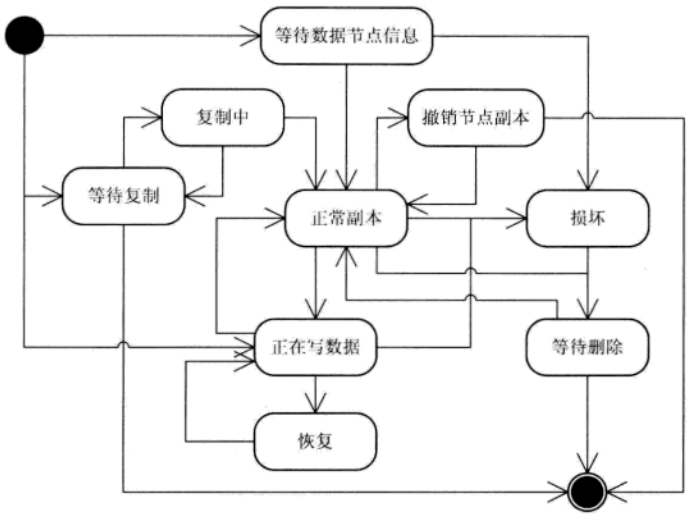



图 8-24 名字节点数据块副本状态转移图

除了上述“等待数据节点信息”状态外，名字节点上的数据块副本由写操作或者复制请求产生，复制结束或者写数据结束后，数据块副本会处于“正常副本”状态。当然，写数据过程中发现数据错误，会导致副本被标记为损坏，并被删除。“正常副本”到“损坏”的转移，一般由数据节点的数据块扫描器、数据块接收器或客户端等的数据校验过程发现，上报到名字节点而触发。正常的文件删除，或者数据节点的数据块上报处理过程中发现的多余数据块，会使数据块副本从“正常副本”转移到“等待删除”状态，并最终通过名字节点指令，从数据节点中删除。当数据节点进入“正在撤销”或“已撤销”状态时，保存在该数据节点的数据块副本进入“撤销节点副本”状态，这些副本或由于节点下架而进入“最终状态”，或由于节点重新进入正常工作状态而返回“正常副本”状态。为了不使状态图过于复杂，这里其实简化了多个可能到“撤销节点副本”的状态转移，如“损坏”到“撤销节点副

本”的状态转移。

由于数据节点的波动（如重启），图中还存在着两个比较特殊的状态转移：由于副本数超过副本系数而进入“等待删除”状态的一些副本，会有机会返回到“正常副本”状态；另外，“等待复制”状态的副本，可能会被直接删除。

后面讨论数据块副本管理的相关过程时，还会继续涉及数据块副本的状态转移。

如果有一个数据块，怎么知道它的副本的情况呢？方法 FSNamesystem.countNodes() 提供了这样的信息，它返回一个 NumberReplicas 对象，保存了处于不同状态的副本数。NumberReplicas 各个字段的含义，请参考下面代码的注释：

```
static class NumberReplicas {
   private int liveReplicas;// “正常副本”副本数
   private int decommissionedReplicas;// 位于撤销数据节点的副本数
   private int corruptReplicas;// 损坏的副本数
   private int excessReplicas;// 多余副本数
   ……
}

private NumberReplicas countNodes(Block b,
      Iterator<DatanodeDescriptor> nodeIter) {
   int count = 0;
   int live = 0;
   int corrupt = 0;
   int excess = 0;
   Collection<DatanodeDescriptor> nodesCorrupt
         = corruptReplicas.getNodes(b);
   while (nodeIter.hasNext()) {
      DatanodeDescriptor node = nodeIter.next();
      if ((nodesCorrupt != null) && (nodesCorrupt.contains(node))) {
         corrupt++; // 已经损坏的副本
      } else if (node.isDecommissionInProgress()
            || node.isDecommissioned()) {
         count++; // 位于撤销数据节点的副本数
      } else {
         Collection<Block> blocksExcess =
               excessReplicateMap.get(node.getStorageID());
         if (blocksExcess != null && blocksExcess.contains(b)) {
            excess++; // 多余副本数
         } else {
            live++; // “正常副本”
         }
      }
   }
   return new NumberReplicas(live, count, corrupt, excess);
}
```

FSNamesystem.countNodes() 根据 corruptReplicas、excessReplicateMap 等对象中保存的信息，进行简单的统计即可得到上述信息，具体代码请读者自行分析。

## 8.2.2 数据节点管理

数据节点启动时，会密集和名字节点通信，握手、注册并进行数据块上报，然后定期发送心跳信息，维护和名字节点的联系。本节讲解名字节点如何管理数据节点，包括：添加和撤销数据节点，数据节点启动时名字节点上执行的流程，心跳处理和名字节点的指令如何产生并下发等内容。

### 1. 添加和撤销数据节点

HDFS 在需要增加集群容量时，可以动态地往集群添加新的数据节点。相反，如果希望缩小集群的规模，那么需要撤销已存在的数据节点。如果一个数据节点频繁地发生故障或者运行缓慢，也可以通过撤销操作，将节点下架。上述操作让 HDFS 有了一定的“弹性”，可根据应用规模进行扩展或收缩。但在实现上述功能时，需要保证名字节点对能够连接到名字节点的数据节点进行明确的管理，以保证数据节点受集群控制，也可以防止配置错误的数据节点误接入名字节点。

最简单的添加数据节点方法是根据集群情况、安装并设置新节点的配置文件，然后手动启动 DataNode 的守护进程（bin/hadoopdatanode），它会自动联系 NameNode 并加入集群。但 HDFS 提供了更明确的 dfs.hosts 文件（由配置项 ${dfs.hosts} 指定，又称 include 文件）和 dfs.hosts.exclude 文件（由配置项 ${dfs.hosts.exclude} 指定，简称 exclude 文件），对能够连接到名字节点的数据节点进行更明确的管理，以保证数据节点受集群控制，也可以防止配置错误的数据节点接入名字节点。

include 文件指定了可以连接到名字节点的数据节点列表，文件保存在名字节点本地文件系统上，一行表示一个数据节点。

exclude 文件用于指定不能连接到名字节点的数据节点列表，也用于撤销数据节点，该文件格式和 include 文件一致。

上述两个文件，按照表 8-2 中的规则决定对节点采取的动作。

**表 8-2 节点采取动作的规则**

| include 文件中包含 | exclude 文件中包含 | 策略 |
|---|---|---|
| 否 | 否 | 节点可以连接 |
| 否 | 是 | 节点不可以连接 |
| 是 | 否 | 节点可以连接 |
| 是 | 是 | 节点撤销 |

一个未定义或空的 include 文件意味着所有的节点都包含在其中。有了 include 文件后，在添加数据节点时，需要在 include 文件中添加对应的记录并通过多任务工具 dfsadmin 的 refreshNodes 命令（hadoop dfsadmin -refreshNodes），刷新名字节点信息，然后才可以启动数据节点。

撤销节点通过 exclude 文件，将要撤销的节点增加到文件中，然后还是执行上面的

dfsadmin refreshNodes 命令，名字节点就会开始撤销数据节点。被撤销节点上的数据块会复制到集群的其他数据节点，这个过程中，数据节点处于“正在撤销”状态，数据复制完成后才会转移到“已撤销”，这个时候就可以关闭相应的数据节点了。

命令 refreshNodes 其实使用远程接口 ClientProtocol.refreshNodes()，通知名字节点更新 include 和 exclude 文件。该远程方法的主要实现逻辑在 FSNamesystem.refreshNodes() 中，代码如下：

```
public void refreshNodes(Configuration conf) throws IOException {
   ……
   // 更新文件名
   hostsReader.updateFileNames(conf.get("dfs.hosts",""),
         conf.get("dfs.hosts.exclude", ""));
   hostsReader.refresh();// 读取文件
   synchronized (this) {
      for (Iterator<DatanodeDescriptor>
               it = datanodeMap.values().iterator(); it.hasNext();) {
         DatanodeDescriptor node = it.next();
         if (!inHostsList(node, null)) {// 不在 include 文件中
            // 直接设置为已撤销状态，不会拷贝节点上的数据块，所以，撤销节点时，要先
            // 在 exclude 文件中添加记录，撤销结束后再删除 include 中的记录
            node.setDecommissioned();
         } else {
            if (inExcludedHostsList(node, null)) {
               if (!node.isDecommissionInProgress() &&
                     !node.isDecommissioned()) {
                  // 开始撤销节点，会拷贝节点上的数据块
                  startDecommission(node);
               }
            } else {
               if (node.isDecommissionInProgress() ||
                     node.isDecommissioned()) {
                  stopDecommission(node);// 停止撤销过程
               }
            }
         }
      }
   }
}
```

文件 include 和 exclude 中包含的列表信息保存在 FSNamesystem 的成员变量 hostsReader 中（类型为 org.apache.hadoop.util.HostsFileReader），读入信息后，方法遍历名字节点当前管理的数据节点，并根据前面表格中的规则设置节点的状态，状态保存在 DatanodeInfo.adminState 中。如 startDecommission() 方法就会通过 DatanodeDescriptor 的同名方法，将数据节点的状态设置为 AdminStates.DECOMMISSION_INPROGRESS。然后，调用 checkDecommissionStateInternal()，把这个数据节点拥有的数据块加入到 FSNamesystem 的成员变量 neededReplications 中。代码如下：

```
private void startDecommission (DatanodeDescriptor node)
      throws IOException {
   if (!node.isDecommissionInProgress() && !node.isDecommissioned()) {
      ......
      node.startDecommission();  // 设置节点状态
      node.decommissioningStatus.setStartTime(now());
      checkDecommissionStateInternal(node);
   }
}
```

refreshNodes() 方法返回后，被撤销的节点往往处于 DECOMMISSION_INPROGRESS 状态（正在撤销）中，什么时候节点状态才会变成已撤销呢？名字节点中引入数据节点撤销管理器 DecommissionManager，跟踪节点的状态。FSNamesystem 使用一个线程执行 DecommissionManager 内部类 Monitor 的 run() 方法，通过 Monitor.check() 方法定期检查处于“正在撤销”状态的数据节点上数据块复制的进度。代码如下：

```
private void check() {//DecommissionManager 的内部类 Monitor 的成员函数
   ......
   if (d.isDecommissionInProgress()) {
      try {
         fsnamesystem.checkDecommissionStateInternal(d);
      } catch(Exception e) {
   ......
}
```

FSNamesystem.checkDecommissionStateInternal() 被 Monitor.check() 和 startDecommission() 调用，它的实现很简洁，使用 isReplicationInProgress() 检查数据节点的撤销进度，并根据检查结果设置数据节点的状态。

isReplicationInProgress() 的实现就有点复杂，它需要遍历数据节点 srcNode 上保存的所有数据块，通过 FSNamesystem.countNodes() 了解数据块现在的状态，并根据当前处于“正常副本”的副本数和文件的期望副本数，判断数据块的复制是否结束。isReplicationInProgress() 方法还会计算只存在于当前撤销节点的数据块副本数（参考代码中的条件）和处于“正在写数据”状态的数据块副本数。代码如下：

```
private boolean isReplicationInProgress(DatanodeDescriptor srcNode) {
   boolean status = false;
   int underReplicatedBlocks = 0; // 处于复制状态的数据块（复制还没有结束）
   int decommissionOnlyReplicas = 0; // 只存在于当前撤销节点的数据块副本数
   int underReplicatedInOpenFiles = 0; // “正在写数据”状态的数据块副本数

   for(final Iterator<Block> i=srcNode.getBlockIterator();i.hasNext();) {
      final Block block = i.next();
      INode fileINode = blocksMap.getINode(block);

      if (fileINode != null) {
         NumberReplicas num = countNodes(block);
         int curReplicas=num.liveReplicas(); // 处于“正常副本”状态的副本数
```

```
            int curExpectedReplicas = getReplication(block); //期望副本数
            if (curExpectedReplicas > curReplicas) {
               if (!status) {
                  status = true; //还有数据块需要复制
                  ......
               }
               underReplicatedBlocks++;
               if ((curReplicas == 0)&&(num.decommissionedReplicas()>0)) {
                  decommissionOnlyReplicas++; //只有当前节点拥有
               }
               if (fileINode.isUnderConstruction()) {
                  underReplicatedInOpenFiles++; //写过程还没有结束
               }

               if (!neededReplications.contains(block) &&
                     pendingReplications.getNumReplicas(block) == 0) {
                  //startDecommission()调用，或者写过程刚刚结束，都会执行这段逻辑
                  neededReplications.add(block,curReplicas,
                        num.decommissionedReplicas(),
                        curExpectedReplicas);
               }
            }
         }
      }
      srcNode.decommissioningStatus.set(underReplicatedBlocks,
            decommissionOnlyReplicas, underReplicatedInOpenFiles);

      return status;
   }
```

isReplicationInProgress() 方法还有一个要点：在最后一个条件判断中，当 FSNamesystem.startDecommission() 调用 isReplicationInProgress() 时，节点刚进入撤销状态，由于 NumberReplicas.liveReplicas() 不计算位于撤销数据节点的副本，所以数据块的期望副本数一般会比 liveReplicas 的值大，这时，撤销节点上的数据块会加入 neededReplications 队列，并被复制。数据块副本进入复制队列的另一个机会是：撤销节点上某个数据块写操作结束，数据块提交到名字节点，在 Monitor.check() 过程中，这样的数据块会被发现并加入到 neededReplications 中。所以，只有当撤销节点上没有正在写的数据块，并且所有的数据块都被复制的情况下，isReplicationInProgress 才会返回 false，节点状态才可以修改为“已撤销”。

和添加数据节点比，由于需要等待该数据节点上写数据流程执行结束，并转移数据块，撤销节点的过程略显复杂，但上述流程，保证了即使文件只有一个副本，也不会因数据节点撤销而导致数据丢失。

### 2. 数据节点的启动

添加数据节点并启动节点时，执行的流程和正常的数据节点启动是一样的。按第 7 章的分析，数据节点启动时，需要和名字节点进行握手、注册和数据块上报。如果系统支持 Append 操作，还需要上报处于客户端写状态的数据块信息。本节先关注前三个远程调用，

即远程接口 DatanodeProtocol 的方法 versionRequest()、register() 和 blockReport() 在名字节点上的实现。

（1）握手

这些方法中，versionRequest() 最简单，调用 FSNamesystem.getNamespaceInfo() 并返回方法的返回结果。代码如下：

```
// 实现在 org.apache.hadoop.hdfs.server.namenode.NameNode 中
public NamespaceInfo versionRequest() throws IOException {
   return namesystem.getNamespaceInfo();
}

// 实现在 org.apache.hadoop.hdfs.server.namenode.FSNamesystem 中
synchronized NamespaceInfo getNamespaceInfo() {
   return new NamespaceInfo(dir.fsImage.getNamespaceID(),
         dir.fsImage.getCTime(),getDistributedUpgradeVersion());
}
```

（2）注册

远程方法 register() 的主要处理逻辑在 FSNamesystem.registerDatanode() 中，只要注意到数据节点可以重复发送注册信息，这个方法的实现就比较好理解。

首先，registerDatanode() 需要为注册节点生成数据节点标识，名字节点不能完全信任数据节点发送过来的信息，它需要根据实际情况更新注册信息中携带的数据节点标识，然后，使用这个标识进行后续的处理。这一部分的代码还会判断是否允许该数据节点连接到名字节点，判断的依据是上一节介绍的 include 文件和 exclude 文件规则，由 verifyNodeRegistration() 方法执行。如果该数据节点不允许连接，简单抛出 DisallowedDatanodeException 异常即可。

接下来，名字节点需要根据不同情况，对注册请求进行不同的处理。可能出现的情况如下：

- ❑该数据节点没有注册过。
- ❑数据节点注册过，这次注册是重复注册。
- ❑数据节点注册过，但这次注册使用了新的数据节点存储标识（storageID），表明该数据节点存储空间已经被清理过了，原有的数据块副本已经被删除。

我们知道，datanodeMap 是数据节点存储标识到 DatanodeDescriptor 的映射，为了区分上述情形，名字节点引入了另外一个映射 Host2NodesMap，提供了从服务器（名称）到该服务器上启动的数据节点的 DatanodeDescriptor（可能有多个）的映射。Host2NodesMap 的成员变量声明如下：

```
class Host2NodesMap {
   private HashMap<String, DatanodeDescriptor[]> map =
         new HashMap<String, DatanodeDescriptor[]>();
   ......
}
```

通过存储标识在 datanodeMap 中获取的数据节点描述符对象为 nodeS，根据服务器名

称和端口号在 host2DataNodeMap（Host2NodesMap 实例）中获取的描述符为 nodeN，如果 nodeN 不为空，同时 nodeN 和 nodeS 不相等，则属于上面讨论的情况三，即数据节点使用新的存储标识进行注册。这时，nodeN 即是原有老数据节点的标识，利用这个标识，通过 FSNamesystem 的 removeDatanode() 和 wipeDatanode() 方法，清理原有节点在名字节点中保存的信息，并将 nodeN 设置为空，后续的处理，就和情况一的一致了。在排除了情况三以后，数据节点否是重复注册（情况二），只需要根据 nodeS 是否为空，即可判断。代码如下：

```
public synchronized void registerDatanode(DatanodeRegistration nodeReg)
      throws IOException {
   String dnAddress = Server.getRemoteAddress();// 获得数据节点的地址
   if (dnAddress == null) {
      dnAddress = nodeReg.getHost();
   }

   // 该数据节点是否允许连接到这个名字节点
   if (!verifyNodeRegistration(nodeReg, dnAddress)) {
      throw new DisallowedDatanodeException(nodeReg);
   }

   String hostName = nodeReg.getHost();
   // 使用 ip:port 更新数据节点标识
   DatanodeID dnReg = new DatanodeID(dnAddress + ":" + nodeReg.getPort(),
         nodeReg.getStorageID(),……);
   nodeReg.updateRegInfo(dnReg);
   ……
   DatanodeDescriptor nodeS = datanodeMap.get(nodeReg.getStorageID());
   DatanodeDescriptor nodeN =
         host2DataNodeMap.getDatanodeByName(nodeReg.getName());

   if (nodeN != null && nodeN != nodeS) { // 数据节点使用新的存储标识进行注册
      ……
      removeDatanode(nodeN);
      wipeDatanode(nodeN);
      nodeN = null;
   }

   if (nodeS != null) { // 重复注册
      ……
      ……// 更新节点的网络拓扑位置
      ……// （可能的）更新心跳信息
      return;
   }

   if (nodeReg.getStorageID().equals("")) {// 数据节点从来都没有注册过
      nodeReg.storageID = newStorageID();// 分配新的 storageID
      ……
   }

   // 登记新的数据节点
```

```
    DatanodeDescriptor nodeDescr = new DatanodeDescriptor(nodeReg,
          NetworkTopology.DEFAULT_RACK, hostName);
    resolveNetworkLocation(nodeDescr);
    unprotectedAddDatanode(nodeDescr);
    clusterMap.add(nodeDescr);

    // 加入到心跳检查列表中，注册信息相当于心跳
    synchronized(heartbeats) {
       heartbeats.add(nodeDescr);
       nodeDescr.isAlive = true;
    }
    return;
}
```

对重复注册的数据节点，由于名字节点已经拥有该节点的信息，所以处理逻辑相对简单：更新节点在网络拓扑中的位置和（可能的）心跳信息。如果注册的是新数据节点，则需要分配数据节点存储标识（如果节点从没注册过）、创建数据节点描述符、获得节点的网络拓扑位置、添加节点到 datanodeMap、host2DataNodeMap（通过 unprotectedAddDatanode() 方法）和心跳信息列表 heartbeats 中。

注册新节点时需要进行一系列的簿记工作。从名字节点中移除数据节点时，可使用方法 removeDatanode() 和 wipeDatanode()，这两个方法除了在 datanodeMap 等对象中移除相应的记录外，还需要更新名字节点第二关系，由于 removeStoredBlock() 方法会移除保存在 BlocksMap 对象中的一些数据块副本，往往会改变该数据节点管理的数据块的副本的状态（影响副本不限于当前数据节点），如触发数据块复制工作、删除无效数据块副本 recentInvalidateSets 中和这个数据节点相关的记录等。代码如下：

```
private void removeDatanode(DatanodeDescriptor nodeInfo) {
   synchronized (heartbeats) {// 与心跳相关的记录
      if (nodeInfo.isAlive) {
         updateStats(nodeInfo, false); // 更新集群的一些统计信息
         heartbeats.remove(nodeInfo); // 在心跳列表中移除记录
         nodeInfo.isAlive = false;
      }
   }

   // 在名字节点第二关系中移除数据节点的数据块副本记录
   for (Iterator<Block> it=nodeInfo.getBlockIterator(); it.hasNext();) {
      removeStoredBlock(it.next(), nodeInfo);
   }
   unprotectedRemoveDatanode(nodeInfo);
   clusterMap.remove(nodeInfo);// 在集群网络拓扑中删除节点
}

void unprotectedRemoveDatanode(DatanodeDescriptor nodeDescr) {
   nodeDescr.resetBlocks();// 重置 DatanodeDescriptor 对象中的信息

   // 移除 recentInvalidateSets 中的记录
```

```
        removeFromInvalidates(nodeDescr.getStorageID());
        ……
    }

    void wipeDatanode(DatanodeID nodeID) throws IOException {
        String key = nodeID.getStorageID();
        host2DataNodeMap.remove(datanodeMap.remove(key));// 移除记录
        ……
    }
```

（3）数据块上报

成功注册的数据节点，接下来会进行数据块上报，向名字节点提供它管理的数据块信息，该请求的主要处理实现在 FSNamesystem.processReport() 中。这个方法有两个关注点：

首先是异常 DisallowedDatanodeException，processReport() 通过 shouldNodeShutdown()，检查该数据节点的状态是否为 AdminStates.DECOMMISSIONED，如果是，表明该节点不允许连接到名字节点，通过抛异常，通知数据节点停止工作。

另一个关注点是通过 DatanodeDescriptor.reportDiff()，将该数据节点管理的数据块副本，根据集群当前情况，进行分类，并添加到不同的副本状态管理对象中去。如方法变量 toInvalidate 中就保存了要删除的数据块副本，这些副本最终通过 addToInvalidates() 方法，添加到 FSNamesystem 的成员变量 recentInvalidateSets 中。下一节会详细讲解这些涉及数据块副本状态的方法，在此就不再赘述了。

上述关注点涉及代码如下：

```
    public synchronized void processReport(DatanodeID nodeID,
            BlockListAsLongs newReport) throws IOException {
        ……
        DatanodeDescriptor node = getDatanode(nodeID);
        ……
        if (shouldNodeShutdown(node)) {
            setDatanodeDead(node);
            throw new DisallowedDatanodeException(node);
        }
        ……
        // 根据当前数据块副本的情况，增加、移除或者删除数据块副本
        Collection<Block> toAdd = new LinkedList<Block>();
        Collection<Block> toRemove = new LinkedList<Block>();
        Collection<Block> toInvalidate = new LinkedList<Block>();
        node.reportDiff(blocksMap, newReport, toAdd, toRemove, toInvalidate);

        for (Block b : toRemove) {
            removeStoredBlock(b, node); // 移除数据块副本
        }
        for (Block b : toAdd) {
            addStoredBlock(b, node, null); // 添加数据块副本
        }
        for (Block b : toInvalidate) {
            ……
```

```
            addToInvalidates(b, node); // 删除数据块副本
        }
        ……
    }
```

### 3. 心跳

在 DataNode.offerService() 中，数据节点利用循环向名字节点发送心跳信息，维护它们间的关系，上报负载信息并获取名字节点指令。由上述描述可知，名字节点中和数据节点心跳相关的代码可以分为两部分，心跳信息处理和心跳检查。

（1）心跳信息处理

FSNamesystem.handleHeartbeat() 被 NameNode.sendHeartbeat() 调用，分三个步骤处理心跳信息。

1）首先是对发送请求的数据节点进行检查，判断该节点是否能连接到名字节点（前面介绍的 processReport() 也有这样的机制）。同时也判断数据节点是否已经注册过，未注册的数据节点会收到 DatanodeCommand.REGISTER 指令，这时，节点要重新注册并上报数据块信息。

2）名字节点利用心跳消息中的负载信息，更新整个 HDFS 系统的负载信息。两次 updateStats() 调用，通过先在计数器里减去上一次更新值，然后再加上这次上报的值，得到整个系统新的负载值。DatanodeDescriptor.updateHeartbeat() 不但更新节点负载，同时也更新了节点的心跳时间。

3）名字节点会为这个数据节点产生名字节点指令，并通过远程调用的返回值返回。其实步骤 1）中的 DatanodeCommand.REGISTER，就是一种名字节点指令。

handleHeartbeat() 方法一般通过 DatanodeDescriptor 的对应方法产生名字节点指令，下面代码中的删除数据块副本指令，就是通过 getInvalidateBlocks() 方法获得的。代码如下：

```
DatanodeCommand[] handleHeartbeat(DatanodeRegistration nodeReg,
      long capacity, long dfsUsed, long remaining,
      int xceiverCount, int xmitsInProgress) throws IOException {
   DatanodeCommand cmd = null;
   synchronized (heartbeats) {
      synchronized (datanodeMap) {
         DatanodeDescriptor nodeinfo = null;
         try {
            nodeinfo = getDatanode(nodeReg);
         } catch(UnregisteredDatanodeException e) {
            return new DatanodeCommand[]{DatanodeCommand.REGISTER};
         }
         ……// 检查数据节点是否应该被关闭，和 processReport() 中的代码类似
         if (nodeinfo == null || !nodeinfo.isAlive) {// 重新注册
            return new DatanodeCommand[]{DatanodeCommand.REGISTER};
         }
```

```
            updateStats(nodeinfo, false);
            nodeinfo.updateHeartbeat(capacity, dfsUsed,
                  remaining, xceiverCount);
            updateStats(nodeinfo, true);
            ......
            ArrayList<DatanodeCommand> cmds =
                  new ArrayList<DatanodeCommand>();
            ......
            // 数据块副本删除命令
            cmd = nodeinfo.getInvalidateBlocks(blockInvalidateLimit);
            if (cmd != null) {
               cmds.add(cmd);
            }
            ......
            if (!cmds.isEmpty()) {
               return cmds.toArray(new DatanodeCommand[cmds.size()]);
            }
         }
      }
      ......
   }
```

生成名字节点指令所需的数据都存放在DatanodeDescriptor对象中，所以getInvalidateBlocks()的实现只需要两行代码，从成员变量invalidateBlocks中获取要删除的副本列表，并根据列表创建删除命令。代码如下：

```
BlockCommand getInvalidateBlocks(int maxblocks) {
   Block[] deleteList = getBlockArray(invalidateBlocks, maxblocks);
   return deleteList == null? null:
         new BlockCommand(DatanodeProtocol.DNA_INVALIDATE, deleteList);
}
```

（2）心跳检查

心跳信息的处理由远程方法sendHeartbeat()实现。FSNamesystem中和心跳相关的另一部分代码是心跳检查，它拥有自己的线程，并通过heartbeatCheck()执行检查逻辑。心跳检查的间隔保存在成员变量heartbeatRecheckInterval中，默认值是5分钟，可以通过配置项${heartbeat.recheck.interval}重新指定。代码如下：

```
class HeartbeatMonitor implements Runnable {//FSNamesystem的内部类
   private long lastHeartbeatCheck;
   ......
   public void run() {
      while (fsRunning) {
         try {
            long now = now();
            if (lastHeartbeatCheck + heartbeatRecheckInterval < now) {
               heartbeatCheck();
               lastHeartbeatCheck = now;
            }
```

```
    }……// 异常处理代码等
    try {
       Thread.sleep(5000);//5 秒
    }
    ……
```

在方法 heartbeatCheck() 中，如果发现节点长时间[⊖]没有上报心跳（在 isDatanodeDead() 中实现）即可判断节点已不能正常工作，这时，调用 removeDatanode() 方法移除数据节点。

最简单的 heartbeatCheck() 实现方式，是通过一个循环语句依次检查名字节点的状态并对故障节点进行处理。但 heartbeatCheck() 没有采用这样的实现方式，而是采取了先寻找可能出现故障的数据节点（需要一个循环语句），随后处理故障节点，再寻找下一个可能故障节点的形式（需要另外一个循环语句，即外围的 while 语句）实现心跳检查。原因主要是：由于故障处理需要对 heartbeats 和 datanodeMap 等进行同步操作，这时，心跳处理函数 handleHeartbeat() 就不能对它们进行更新，在系统负载重，或者机架故障造成数据节点大量故障时，会造成误判断。所以，heartbeatCheck() 使用了故障检查和故障处理分离的方法，尽量避免上述情况。

这样带来的问题是：故障处理前，需要再次确认故障状态，所以在调用 removeDatanode() 前，heartbeatCheck() 还需要再次使用 isDatanodeDead() 确认故障。相关代码如下：

```
void heartbeatCheck() {
   ……
   boolean allAlive = false;
   while (!allAlive) {
      boolean foundDead = false;
      DatanodeID nodeID = null;

      // 确定第一个故障的节点
      synchronized(heartbeats) {
         for (Iterator<DatanodeDescriptor> it = heartbeats.iterator();
               it.hasNext();) {
            DatanodeDescriptor nodeInfo = it.next();
            if (isDatanodeDead(nodeInfo)) {
               foundDead = true;
               nodeID = nodeInfo;
               break;
            }
         }
      }

      // 删除数据节点
      if (foundDead) {
         // synchronized (this、heartbeats、datanodeMap)
```

⊖ 和心跳相关的另一个配置项是 ${dfs.heartbeat.interval}，它是数据节点发送心跳的最小间隔，默认值是 3 秒。名字节点中，使用 ${dfs.heartbeat.interval}×10 + ${heartbeat.recheck.interval}×2，作为判断数据节点故障的阈值。

```
            ......
            DatanodeDescriptor nodeInfo = null;
            try {
               nodeInfo = getDatanode(nodeID);
            } ..... //异常处理代码
            if (nodeInfo != null
                  && isDatanodeDead(nodeInfo)) {// 再次判断节点状态
               ......
               removeDatanode(nodeInfo);// 移除数据节点
            }
            ......
         }
         allAlive = !foundDead;
      }
   }
```

在复杂的名字节点实现中，心跳接收和心跳检查的代码相对简单，但心跳检查中的故障检测和故障处理分离，体现了 HDFS 设计者对复杂分布式系统的把握能力。

#### 4. DatanodeProtocol 中的其他远程方法

我们已经研究了 DatanodeProtocol 中的 versionRequest()、register()、blockReport() 和 sendHeartbeat() 四个方法。该接口中，还有一类方法与数据节点上数据块副本状态的变化相关，如 blockReceived()、reportBadBlocks() 等，后面的章节将结合客户端的远程接口 ClientProtocol 对文件读写的支持，一并介绍。

在第 7 章介绍数据节点的过程中，并没有分析远程方法 DatanodeProtocol.errorReport()，它用于数据节点向名字节点上报运行过程中发生的一些状况。读写数据块时，数据节点的磁盘不可用；数据节点和名字节点握手时，（系统构建）版本号不一致；数据复制时，被复制的数据块不存在等三种情况发生时，数据节点都会将错误信息上报到名字节点。如图 8-25 所示。

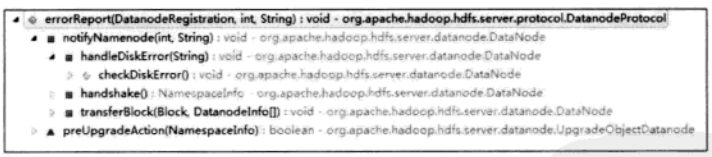



图 8-25　数据节点使用 errorReport() 上报错误

接收到数据节点上报的错误时，NameNode.errorReport() 方法会将信息输出到日志，以方便运维人员发现错误。严重的错误，方法会采取相应的处理措施，如数据节点的所有磁盘都不可用时，errorReport() 会使用 FSNamesystem.removeDatanode()，删除数据节点，避免影响集群的工作。代码如下：

```
public void errorReport(DatanodeRegistration nodeReg,
      int errorCode,String msg) throws IOException {
   String dnName = (nodeReg == null ?
         "unknown DataNode" : nodeReg.getName());
   LOG.info("Error report from " + dnName + ": " + msg); // 记录信息到日志中
```

```
        if (errorCode == DatanodeProtocol.NOTIFY) {
            return;
        }
        ......
        } else if (errorCode == DatanodeProtocol.FATAL_DISK_ERROR) {
            namesystem.removeDatanode(nodeReg);
        }
    }
```

DatanodeProtocol.processUpgradeCommand() 和升级相关，限于篇幅就不再研究了。

### 5. 感知网络布局

随着 HDFS 集群变大，节点将分布在多个机架中，这个时候，集群的网络拓扑结构会开始影响系统的可靠性和性能。一般来说，节点发生故障的概率比机架大，而机架发生故障的概率比整个集群所有节点都出现问题（如停电）大，如果能够感知网络拓扑，就可以安排数据块副本的位置，以提高抗节点 / 机架故障的能力。

另一方面，同一机架中，节点间的带宽资源，会比跨机架间的资源丰富，特别是在处理大规模数据时，了解网络布局，可更好地利用数据的局部性。当集群部署在多个机架上时，一个简单的启发式策略是将数据块副本放在彼此独立的机架上，以减少潜在的数据丢失。HDFS 的标准副本系数是 3，在写入一个数据块时的默认放置策略是：如果执行写操作的客户端是 Hadoop 集群的一部分，第一个副本应该放在客户端所在的数据节点中，否则，随机选择一个放置位置。第二个副本随机放置在与第一个副本不同的机架中，为了充分利用机架的带宽，第三个副本放在与第二个副本相同机架的不同节点上，请参考下面图 8-26。如果副本数大于 3，后续的副本在数据节点上随机放置。目前，这样的数据块放置策略实现在名字节点中，HDFS 后续版本会提供可插拔的框架[⊖]，让用户更灵活地控制数据块放置。

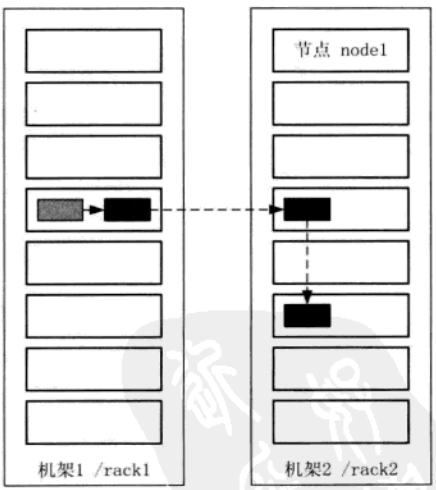


图 8-26 副本的放置

以图 8-26 为例，在 Hadoop 中，假定集群为层次式，或者把网络的拓扑结构看成是一棵树，图中的两个机架，机架位置分别是“/rack1”和“/rack2”。机架 2 下面某台主机的位置就是“/rack2/node1”，通过这种形式，可以为集群中的每一个机器定义一个位置。显然，这些位置信息构成了以“/”为根的一棵树。有了这棵树，就可以通过简单的方式确定节点间的网络距离。树上两个节点间的距离，就是从某一个节点到另一个节点经过的边的数量。以节点“/rack1/node1”为基准，它到节点“/rack1/node1”，也就是它自身，由于不需要经过树上的任何边，所以距离为 0。该

⊖ 具体可参考 JIRA HDFS-385（https://issues.apache.org/jira/browse/HDFS-385）。

节点到同一个机架上的另外一个节点，如“/rack1/node8”的距离为 2，因为需要通过节点“/rack1”。显然，节点“/rack1/node1”与节点“/rack2/node1”的距离是 4。

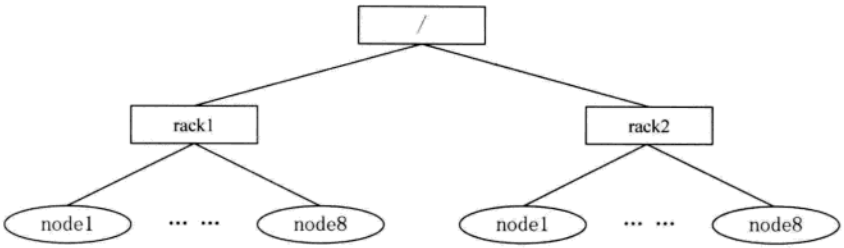



图 8-27　节点间的网络距离

注意，上述结构不限于机架 / 主机两层结构，Hadoop 支持多层的网络拓扑树。实现中，类 NetworkTopology 用于保存上述树状结构，树上的节点则用接口 Node 抽象。在 HDFS 中，DatanodeInfo、DatanodeDescriptor 的父类实现了 Node 接口。代码如下：

```
public interface Node {
   public String getNetworkLocation();// 活动网络的拓扑位置
   ……
   public Node getParent();// 得到当前节点的父节点
   ……
}

public class DatanodeInfo extends DatanodeID implements Node {
   ……
}
```

下面的代码给出了 NetworkTopology 中几个代表性的方法，它们的实现都不是很复杂，就不再深入分析了。其中，当客户端读数据时，pseudoSortByDistance() 用于对多个数据节点做近似排序，以期望客户端能访问到离它最近的数据节点。代码如下：

```
public class NetworkTopology {
   ……
   public int getDistance(Node node1, Node node2) {
      …… // 实现上述网络距离计算
   }

   // 判断两个节点属于同一个机架
   public boolean isOnSameRack( Node node1,  Node node2) {
      ……
      netlock.readLock().lock();
      try {
         return node1.getParent()==node2.getParent();
      } finally {
         netlock.readLock().unlock();
      }
   }
```

```
    // 该方法将 nodes 中，和 reader 同一节点或同一机架放在最前面，它并不对 nodes 排序
    // 只是尽量找一个离 reader 最近的节点，并把它交互到 nodes[0] 上
    public void pseudoSortByDistance( Node reader, Node[] nodes ) {
        int tempIndex = 0;
        if (reader != null ) {
            int localRackNode = -1;
            for(int i=0; i<nodes.length; i++) {// 扫描数组 nodes
                if(tempIndex == 0 && reader == nodes[i]) {// 同一节点
                    ……// 交换该节点到到 nodes[0]，并做记录
                    break;
                }else if(localRackNode == -1 &&
                        isOnSameRack(reader, nodes[i])) { // 同一机架
                    localRackNode = i; // 记录位置，为下面的交换做准备
                    if(tempIndex != 0 ) break;
                }
            }
            …… // 如果 reader 不在 nodes 中，但 nodes 中找到位于同一机架的节点，交换
        }
        ……
    }
    ……
}
```

最后一个问题是，集群中的节点如何知道它所处的位置呢？这需要管理员来告诉它。DNSToSwitchMapping 接口用于进行主机到网络位置的转换，它能够将一批节点（保存在参数 names 中）的 IP 地址，转换成网络位置字符串。注意，这个接口定义在 org.apache.hadoop.net 中，并不是 HDFS 专有的接口，它也被 Hadoop 的 MapReduce 框架使用。定义如下：

```
package org.apache.hadoop.net;
……
public interface DNSToSwitchMapping {
    public List<String> resolve(List<String> names);
}
```

配置项 ${topology.node.switch.mapping.impl} 用于指定 DNSToSwitchMapping 的一个实现，Hadoop 提供的 ScriptBasedMapping，可以通过运行一个 Shell 脚本，完成上述 IP 地址到网络位置的转换。有兴趣的读者可以分析它的代码，和 DNSToSwitchMapping 一样，它的实现也在包 org.apache.hadoop.net 中。

## 8.2.3 数据块管理

名字节点第二关系的管理包括数据节点管理和数据块管理，本节在数据节点管理机制的基础上，继续研究和数据块管理相关的 FSNamesystem 逻辑。

### 1. 添加数据块副本

FSNamesystem.addStoredBlock() 是一个“大”方法，用于在 BlockMap 中添加 / 更新数据节点 node 上的数据块副本 block。在分析数据节点的实现时，我们知道，不管是客户端

写数据或者是数据块复制，操作成功后都会使用远程方法 DatanodeProtocol.blockReceived() 向名字节点汇报，NameNode.blockReceived() 就会调用这个方法，将数据块副本和数据节点信息更新到 BlocksMap 对象中。如果该数据块副本是由数据块复制触发，通过参数 delNodeHint 还可以删除源节点上的数据块副本。使用 addStoredBlock() 方法的另一个场景是数据节点进行数据块上报，FSNamesystem.processReport() 通过它，更新数据节点保存的数据块副本信息到 FSNamesystem.blocksMap 中。

这个方法比较复杂[⊖]，将分段介绍，入口部分的代码如下：

```
synchronized Block addStoredBlock(Block block, DatanodeDescriptor node,
      DatanodeDescriptor delNodeHint) {
   BlockInfo storedBlock = blocksMap.getStoredBlock(block);
   if (storedBlock == null) { // 在 blocksMap 中，没有发现该数据块
      // 不带数据块版本号，在 blocksMap 中再次查找数据块
      storedBlock = blocksMap.getStoredBlockWithoutMatchingGS(block);

      if (storedBlock == null) {
         return rejectAddStoredBlock(block, node,
               "Block not in blockMap with any generation stamp");
      }
      INodeFile inode = storedBlock.getINode();
      if (inode == null) {
         return rejectAddStoredBlock(block, node,"……");
      }

      boolean reportedOldGS = block.getGenerationStamp()
            <storedBlock.getGenerationStamp();// 上报的是一个旧副本
      boolean reportedNewGS = block.getGenerationStamp()
            > storedBlock.getGenerationStamp();// 上报的是一个新副本
      // 所属文件是否处于构建状态
      boolean underConstruction = inode.isUnderConstruction();
      boolean isLastBlock = inode.getLastBlock() != null// 是否是最后一个块
            && inode.getLastBlock().getBlockId() == block.getBlockId();

      // 这是一个“旧”数据块，而且它不是某个处于构建状态文件的最后一个数据块
      if (reportedOldGS && !(underConstruction && isLastBlock)) {
         return rejectAddStoredBlock(block, node,"……");
      }

      // 它是某个处于构建状态文件的最后一个数据块，在 INodeFileUnderConstruction
      // 记录这个数据块，并等待数据块恢复过程的 commitBlockSynchronization()
      if (underConstruction && isLastBlock
            && (reportedOldGS || reportedNewGS)) {
         ……
         ((INodeFileUnderConstruction) inode).addTarget(node);
```

⊖ 这部分代码分析会涉及名字节点上已打开的文件，即正在被“写”的文件（它在目录树中的对象是 INodeFileUnderConstruction 实例），也会使用删除数据块或者数据块复制等名字节点第二关系的功能，这些操作的处理流程会在后续的分析中介绍。

```
            return block;
        }
    }

    INodeFile fileINode = storedBlock.getINode();
    if (fileINode == null) {
        return rejectAddStoredBlock(block, node, "……");
    }
    assert storedBlock != null : "Block must be stored by now";
```

很自然，addStoredBlock() 需要对上报的数据块进行检查，如果在 FSNamesystem.blocksMap 中查找不到该数据块，将不带数据块版本号，在 blocksMap 中再次进行查找。由数据节点的代码分析可知，数据块恢复过程中，数据块的版本号会发生变化，这时，故障节点上不参与数据块恢复的数据块版本号会维持不变，集群会出现同一个数据块的不同副本拥有不同版本号的情况。在 blocksMap 中进行两次查找，都找不到数据块（对应的文件已经被删除），或者查找到的数据块并不属于任何文件（inode 为空），则通过 rejectAddStoredBlock() 方法删除该副本。

接下来检查数据块是否已经失效，它需要当前数据块副本是否属于某个处于构建状态文件的最后一个数据块的信息。如果不是，显然，由于版本号不匹配，这是一个失效的副本，则删除之。否则，将数据节点加入到 INodeFileUnderConstruction 对象的当前写数据块目标中，同时可以断定，该数据块处于数据块恢复过程中。这种情况下，通过 addStoredBlock() 添加的这个数据块，现在还不能判断是否有效，需要等待数据块恢复的结果，才能做进一步的处理。

上述代码中，数据节点上报的无效数据块副本都是通过 rejectAddStoredBlock() 方法删除的。这个方法在输出日志后，使用 addToInvalidates()，将数据块副本加入 recentInvalidateSets 中。代码如下：

```
private Block rejectAddStoredBlock(Block block,
        DatanodeDescriptor node,String msg) {
    …… //输出日志
    addToInvalidates(block, node);
    return block;
}
```

通过上面检查的数据块副本，就可以加入到数据节点描述符对象中。下面语句执行后，数据块既保存在 blocksMap 中，也保存在数据节点的 DatanodeDescriptor 对象中。该语句完成了 FSNamesystem.addStoredBlock() 的最基本工作：

```
boolean added = node.addBlock(storedBlock);
```

新添加的副本会影响属于同一数据块的其他副本，addStoredBlock() 后面的代码就用于处理各种可能的影响。

首先，addStoredBlock() 判断这个数据块是否属于某个处于构建状态文件的最后一个数据块，并将结果保存在 blockUnderConstruction 中。代码如下：

```
boolean blockUnderConstruction = false;
if (fileINode.isUnderConstruction()) {
   NodeFileUnderConstruction cons =
         (INodeFileUnderConstruction) fileINode;
   Block last = fileINode.getLastBlock();
   ......
   blockUnderConstruction = last.equals(storedBlock);
}
```

数据节点提交数据块（包含数据块复制）的处理结果时，blockReceived() 需要做一些附加处理。我们知道，数据块上报也会调用 blockReceived() 方法，和数据节点提交数据块的差别，在于前面一种情况，传入方法的数据块（block）对象，和 blocksMap 中保存的数据块（即变量 storedBlock），它们引用的是相同的对象。也就是说，如果“block== storedBlock”，那么 blockReceived() 正处于处理数据块上报的逻辑中[⊖]。注意，这里使用的不是 Block.equals()，而是（不）等号，即比较的是对象的引用是否相等，而不是对象的内容是否相等。

和数据块上报不一样，（追加）写操作可能会改变数据块的长度，显然需要根据数据块副本的长度区分处理。如果新数据块副本比现有副本的长度还小，显然新数据块副本无效，通过 markBlockAsCorrupt()，将其标记为“损坏”（后面会分析这个方法），如果新副本比较大，使用新副本的信息，并将其他副本标记为“损坏”。后续的 HDFS 版本中，已经放弃了这种通过副本大小决定副本有效性的方法[⊜]，我们就不再探究这里面的细节。不过，在分析 HDFS 文件的追加操作时，会给出系统中出现上述不同长度数据块副本的原因。代码如下：

```
if (block != storedBlock) {
   // 数据节点提交数据块的特有逻辑
   if (block.getNumBytes() >= 0) {
      long cursize = storedBlock.getNumBytes();
      INodeFile file = storedBlock.getINode();
      if (cursize == 0) {
         storedBlock.setNumBytes(block.getNumBytes());
      } else if (cursize != block.getNumBytes()) {
         ......
         try {
            if (cursize > block.getNumBytes()
                  && !blockUnderConstruction){
               ......
               // 新数据块副本比现有副本的长度还小
               markBlockAsCorrupt(block, node);
            } else {
               // 新副本长度比现有副本的长度长
               if (!blockUnderConstruction) {
                  // 将原来存在的数据块标记为损坏
                  ..... // 找到保存原有数据块的数据节点，保存在 nodes[] 中
```

⊖ 前面分析过 FSNamesystem.processReport()，但并没有研究它调用的 DatanodeDescriptor.reportDiff() 方法，后面了解了 reportDiff() 的实现后，读者就会更清楚这个判断为什么成立。

⊜ 这里的实现是基于下面的假设，即数据块的副本新长度应该比旧长度长。后续的版本中，已经进行了修改，感兴趣的读者可以参考 JIRA HDFS-145https://issues.apache.org/jira/browse/HDFS-145。

```
                    for (int j = 0; j < count; j++) {
                        ......
                        markBlockAsCorrupt(block, nodes[j]);
                    }
                }
                storedBlock.setNumBytes(block.getNumBytes());
            }
        } catch (IOException e) {
            ......
        }
    }
    // 必要时，更新系统占用空间信息
    ......
  }
  block = storedBlock;
}
assert storedBlock == block : "Block must be stored by now";
```

除了上面针对数据块副本长度，更准确说是副本有效性的处理，更新的数据块副本会影响同一数据块的其他副本的状态，相关逻辑如下：

```
int curReplicaDelta = 0;

if (added) {
  curReplicaDelta = 1;
}
......
NumberReplicas num = countNodes(storedBlock);
int numLiveReplicas = num.liveReplicas();
int numCurrentReplica = numLiveReplicas
      + pendingReplications.getNumReplicas(block);

// 更新安全模式需要的计数器
incrementSafeBlockCount(numCurrentReplica);

if (blockUnderConstruction) { // 文件还没关闭
  INodeFileUnderConstruction cons =
        (INodeFileUnderConstruction) fileINode;
  cons.addTarget(node);
  return block;
}

// 安全模式，名字节点启动中，不需要处理副本数和期望值不一致的情况
if (isInSafeMode())
  return block;

// 开始处理副本数和期望值不一致的情况
short fileReplication = fileINode.getReplication();
if (numCurrentReplica >= fileReplication) { // 副本数多于（等于）期望数
      neededReplications.remove(block, numCurrentReplica,
            num.decommissionedReplicas, fileReplication);
```

```
} else { // 副本数小于期望数，不过需要更新待复制的副本数数目
   updateNeededReplications(block, curReplicaDelta, 0);
}

if (numCurrentReplica > fileReplication) { // 移除多余副本
   processOverReplicatedBlock(block, fileReplication,
         node, delNodeHint);
}

// 副本数已经到达期望值，删除损坏的副本
int corruptReplicasCount =corruptReplicas.numCorruptReplicas(block);
int numCorruptNodes = num.corruptReplicas();
.....
if ((corruptReplicasCount > 0) && (numLiveReplicas >= fileReplication))
   invalidateCorruptReplicas(block);
return block;
}
```

首先，addStoredBlock() 需要获得现在数据块副本的情况，通过 countNodes() 得到“正常副本”副本数 numLiveReplicas，当前副本数（包括等待复制）numCurrentReplica；当然，如果前面顺利地添加了当前副本，则副本的增量 curReplicaDelta 赋值为 1。在忽略几种特殊的，不需要关注副本状态的例外后，addStoredBlock() 需要根据当前副本数和副本系数，分情况进行处理。

副本数多于或者等于副本系数时，显然可以移除可能存在的，处于等待复制队列 neededReplications 中的副本，停止复制请求；反之，通过 updateNeededReplications() 更新等待复制队列中保存的，等待复制的数据块副本数目。

对于有多余副本的情况，addStoredBlock() 方法使用参数 delNodeHint 调用 processOverReplicatedBlock()。由于只添加了一个数据块副本，所以往往会将数据节点 delNodeHint 上的那个副本删除。在方法的最后，“正常副本”副本数大于等于副本系数时，就可以删除所有损坏的数据块副本，以释放数据节点空间。

FSNamesystem.addStoredBlock() 的功能貌似简单，但却是一个非常复杂的方法，我们花了大量的篇幅介绍这个方法，以期获得处理数据块，更准确地说，是涉及名字节点第二关系数据块处理中的一些要点，如需要区分处于构建状态文件和一般文件的不同情况；数据块副本的变化，包括版本号的变化、数据长度的变化等情形；单一副本的变化，会影响同一数据块其他副本的状态等。这些，都值得读者好好体会。

### 2. 删除数据块副本

数据块副本的删除包括如下 3 种情况：

- ❑数据块副本所属的文件被删除，副本也就相应地删除掉。
- ❑数据块副本数多于副本系数，多余的副本会被删除。
- ❑数据块副本已经损坏，也需要删除损坏的副本。

（1）删除文件拥有的数据块副本

当删除目录上的一个文件时（实现在 FSDirectory.unprotectedDelete() 中），文件拥有的数据块通过 FSNamesystem.removePathAndBlocks() 删除。这个方法很简单，对于被删除的每个数据块，首先将它从 blocksMap 中删除；由于数据块被删除，即使有副本损坏，也都不需要再进行数据块复制了，所以，删除 corruptReplicas 中可能存在的副本损坏记录；最后，调用 addToInvalidates() 在所有拥有副本的数据节点上删除数据块。

addToInvalidates() 方法有两个重载实现，下面代码中的这个实现用于获取数据块所在的数据节点，并调用另一个 addToInvalidates()，最终使用 addToInvalidatesNoLog()，删除数据块副本。addToInvalidatesNoLog() 方法简单地将待删除的数据块副本信息添加到 FSNamesystem 的成员变量 recentInvalidateSets 中。上述 3 个方法代码如下：

```
void removePathAndBlocks(String src, List<Block> blocks) throws IOException
{
   leaseManager.removeLeaseWithPrefixPath(src); //租约相关，后续章节讨论
   for (Block b : blocks) {
      blocksMap.removeINode(b);
      corruptReplicas.removeFromCorruptReplicasMap(b); //删除记录
      addToInvalidates(b);
   }
}

private void addToInvalidates(Block b) {
   for (Iterator<DatanodeDescriptor> it = blocksMap.nodeIterator(b);
         it.hasNext();) {
      DatanodeDescriptor node = it.next();
      addToInvalidates(b, node);  //输出日志后调用 addToInvalidatesNoLog()
   }
}

void addToInvalidatesNoLog(Block b, DatanodeInfo n) {
   Collection<Block> invalidateSet =
         recentInvalidateSets.get(n.getStorageID());
   if (invalidateSet == null) {
      invalidateSet = new HashSet<Block>();
      recentInvalidateSets.put(n.getStorageID(), invalidateSet);
   }
   if (invalidateSet.add(b)) {
      pendingDeletionBlocksCount++;
   }
}
```

（2）多余副本删除

addStoredBlock() 方法最后阶段的处理中，移除多余副本使用了 processOverReplicatedBlock() 方法。这个方法其实只是为 chooseExcessReplicates() 准备数据，由于 blocksMap 保存着数据块的所有副本所在的数据节点，方法需要遍历这些数据节点，当节点同时满足下面 3 个条件时，才把它标记为“等待删除数据块”的候选数据节点，并保存在变量 nonExcess 中。

❑该节点信息不在 excessReplicateMap 中。
❑该节点不是一个处于“正在撤销”或“已撤销”的数据节点。
❑该节点保存的副本已经损坏。
processOverReplicatedBlock() 方法的代码如下：

```
private void processOverReplicatedBlock(Block block, short replication,
      DatanodeDescriptor addedNode, DatanodeDescriptor delNodeHint) {
   ……
   Collection<……> nonExcess = new ArrayList<DatanodeDescriptor>();
   Collection<……> corruptNodes = corruptReplicas.getNodes(block);
   for (Iterator<DatanodeDescriptor> it = blocksMap.nodeIterator(block);
         it.hasNext();) {
      DatanodeDescriptor cur = it.next();
      Collection<Block> excessBlocks =
            excessReplicateMap.get(cur.getStorageID());
      if (excessBlocks == null || !excessBlocks.contains(block)) {
         if (!cur.isDecommissionInProgress()&& !cur.isDecommissioned()) {
            if (corruptNodes == null || !corruptNodes.contains(cur)) {
               nonExcess.add(cur);
            }
         }
      }
   }
   chooseExcessReplicates(nonExcess, block, replication,
         addedNode, delNodeHint);
}

void chooseExcessReplicates(Collection<DatanodeDescriptor> nonExcess,
      Block b, short replication, DatanodeDescriptor addedNode,
      DatanodeDescriptor delNodeHint) {
   //rackMap 保存了机架到数据节点的映射
   HashMap<String, ArrayList<DatanodeDescriptor>> rackMap =
         new HashMap<String, ArrayList<DatanodeDescriptor>>();
   ……// 获取数据节点的机架信息，并保存在 rackMap 中

   // 将数据节点划分成两个集合，priSet 是一个机架上包含多个副本的数据节点集合
   // 其他节点保存在 remains 中
   ArrayList<DatanodeDescriptor> priSet = new……;
   ArrayList<DatanodeDescriptor> remains = new……;
   ……

   // 循环选择删除副本，首先选择节点 delNodeHint，接下来选择 priSet 中的数据节点
   // 最后选择 remains 中存储空间剩余最少的节点
   boolean firstOne = true;
   while (nonExcess.size() - replication > 0) {
      DatanodeInfo cur = null;
      ……
      Collection<Block> excessBlocks =
            excessReplicateMap.get(cur.getStorageID());
```

```
            if (excessBlocks == null) {
                excessBlocks = new TreeSet<Block>();
                excessReplicateMap.put(cur.getStorageID(), excessBlocks);
            }
            if (excessBlocks.add(b)) { //将副本信息加入 excessReplicateMap 中
                excessBlocksCount++;
                ……
            }

            addToInvalidatesNoLog(b, cur); //删除副本
            ……
        }
    }
```

FSNamesystem.chooseExcessReplicates() 是一个冗长的方法，其目标是在上述候选数据节点中选择“正确”的节点，执行删除操作。在 8.2.2 节中了解到：数据块副本的放置是有讲究的，尽量不在同一个机架上放置相同的副本；同时，要尽量删除比较“忙”，即剩余空间少的数据节点上的副本。

chooseExcessReplicates() 方法就是使用上述原则，选择并删除多余数据块副本。这个方法要点如下：机架的信息通过 DatanodeDescriptor.getNetworkLocation() 获得，并保持在 rackMap 中；节点的剩余空间可以通过 DatanodeDescriptor 的 getRemaining() 方法得到。待删除副本选定以后，会添加到 FSNamesystem.excessBlocks 中，并通过 addToInvalidatesNoLog()，删除副本，即将副本信息记录到 recentInvalidateSets 中。

对比 processOverReplicatedBlock() 和 removePathAndBlocks()，这两个方法中，删除的数据块副本都记录在 recentInvalidateSets 中，但是 processOverReplicatedBlock() 还会将删除的多余副本信息保存在 excessBlocks 中，即 recentInvalidateSets 和 excessBlocks 都保留了被删除副本的信息。保存在 excessBlocks 中的信息什么时候删除呢？这个问题留到 FSNamesystem.removeStoredBlock() 的分析中解答。

（3）损坏副本删除

最后讨论的情况是损坏副本的删除。客户端读文件或者数据节点上的数据块扫描器都可能会发现损坏的副本，它们会将检测结果上报到名字节点，最终由 markBlockAsCorrupt() 进行处理。

markBlockAsCorrupt() 方法不是很复杂，和大多数数据块管理方法类似，它使用 datanodeMap（通过 getDatanode() 方法）和 blocksMap 中的信息，并根据数据块目前的具体情况进行处理。例如，对于损坏的数据块副本，其归属文件已经删除，则利用 addToInvalidates() 删除副本。

为什么会出现上述副本损坏，归属文件却已经删除的情况呢？HDFS 是一个典型的分布式系统，文件删除后，数据块并不会立即从数据节点上删除，而需要等待某一个名字节点指令，如果这个时候数据块扫描器正好发现副本损坏，并上报名字节点，就可以形成上述特殊情况，名字节点的处理逻辑必须考虑这些偶尔出现的场景，才能保证系统的鲁棒性。

一般情况下，损坏副本需要记录在FSNamesystem的成员变量corruptReplicas中，如果当前数据块副本比期望值高，直接删除该副本即可，否则，还需要使用updateNeededReplications()复制数据块，以保证系统中数据块副本的数量。代码如下：

```
public synchronized void markBlockAsCorrupt(Block blk, DatanodeInfo dn)
      throws IOException {
   DatanodeDescriptor node = getDatanode(dn);
   ......
   final BlockInfo storedBlockInfo = blocksMap.getStoredBlock(blk);
   if (storedBlockInfo == null) {
      ......
   } else {
      INodeFile inode = storedBlockInfo.getINode();
      if (inode == null) { // 没有找到对应的文件，直接删除
         ......
         addToInvalidates(storedBlockInfo, node);
         return;
      }
      // 添加信息到 corruptReplicas 中
      corruptReplicas.addToCorruptReplicasMap(storedBlockInfo, node);
      if (countNodes(storedBlockInfo).liveReplicas()
            > inode.getReplication()) {
         // 数据块副本多于副本系数，立即删除
         invalidateBlock(storedBlockInfo, node);
      } else {
         // 需要复制数据块副本
         updateNeededReplications(storedBlockInfo, -1, 0);
      }
   }
}
```

markBlockAsCorrupt()方法调用另外一个简单的方法invalidateBlock()，删除有其他多余副本的，损坏的数据块副本。invalidateBlock()方法的主要逻辑依赖于addToInvalidates()和removeStoredBlock()[⊖]两个方法。前面已经研究过addToInvalidates()，知道它用于删除数据节点上的数据块副本，另外一个方法removeStoredBlock()的作用是什么呢？请看以下代码：

```
private synchronized void invalidateBlock(Block blk, DatanodeInfo dn)
      throws IOException {
   ......
   DatanodeDescriptor node = getDatanode(dn);
   ......
   int count = countNodes(blk).liveReplicas();
   if (count > 1) { // 有足够多的副本，删除
      addToInvalidates(blk, dn);
      removeStoredBlock(blk, node);
      ......
   }
}
```

⊖ addStoredBlock()方法中调用的invalidateCorruptReplicas()，最终也会调用removeStoredBlock()方法。

```
synchronized void removeStoredBlock(Block block, DatanodeDescriptor node){
   ......
   if (!blocksMap.removeNode(block, node)) {
      ......
      return;
   }

   INode fileINode = blocksMap.getINode(block);
   if (fileINode != null) {
      decrementSafeBlockCount(block); // 更新安全模式计数器
      updateNeededReplications(block, -1, 0);
   }

   Collection<Block> excessBlocks =
         excessReplicateMap.get(node.getStorageID());
   if (excessBlocks != null) {
      if (excessBlocks.remove(block)) {
         excessBlocksCount--;
         ......
         if (excessBlocks.size() == 0) {
            excessReplicateMap.remove(node.getStorageID());
         }
      }
   }

   corruptReplicas.removeFromCorruptReplicasMap(block, node);
}
```

FSNamesystem.removeStoredBlock() 的作用和 addStoredBlock() 相反，它也会对 blocksMap 进行修改，删除数据块副本的信息。该方法的入口处就调用了 BlocksMap.removeNode()，删除 blocksMap 和数据节点对应 DatanodeDescriptor 对象中的数据块副本信息。

和 addStoredBlock() 类似，修改了上述信息后，还可能需要更新名字节点上其他数据块状态相关的数据结构，包括：

- 多余副本：数据块副本被删除，显然数据块对应的多余副本数应该减一。
- 损坏的数据块副本：如果被删除的数据块副本已经标记为损坏，通过 CorruptReplicasMap 的 removeFromCorruptReplicasMap() 方法移除。
- 等待复制队列：如果当前数据块有复制任务，更新任务信息。

removeStoredBlock() 方法中还有一个值得研究的地方，它删除了 excessBlocks 中的副本信息，该信息是在 processOverReplicatedBlock() 的处理过程中，由 chooseExcessReplicates() 添加的。FSNamesystem.excessBlocks 保存的多余数据块副本也会保存在 recentInvalidateSets 中，并由数据节点删除。接下来的问题是，数据节点在删除了数据块的多余副本后，怎么通知名字节点并删除 excessBlocks 中的对应记录？

数据节点和名字节点间的 DatanodeProtocol 接口并没有上报副本删除结果的远程方法，副本的删除，其实是通过数据节点上定期执行的数据块上报 blockReport() 发现的。名

字节点在处理数据块上报的过程中，就可以发现数据节点上哪些数据块被删除，并调用 removeStoredBlock() 删除名字节点上还保存的数据块副本信息（实现在 FSNamesystem.processReport() 中，前面已经研究过这个方法），并由该方法最终删除 excessBlocks 中的副本信息。

到此为止，我们已经了解了三种数据块副本删除的场景，也分析了多个相关的 FSNamesystem 方法。这些方法，共性多于个性，差别在于不同场景需要执行的特有操作，共同的特点在于它们都会涉及数据块其他副本的状态。另一个共性是：待删除的数据块最终都放入 FSNamesystem 的成员变量 recentInvalidateSets 中，但 recentInvalidateSets 只保存了副本删除请求，遗留的问题是：这些请求什么时候进入 DatanodeDescriptor 的成员变量 invalidateBlocks 中，并最终通过名字节点指令在数据节点得到执行，下一节将讨论这些问题。

### 3. 数据块复制

数据块复制是 HDFS 正常运行的保证，也体现了 HDFS 故障检测和自动恢复的特性。

前面介绍的名字节点第二关系上的多个流程都会触发数据块复制，如数据节点撤销、数据块副本损坏等，这些流程一般通过 FSNamesystem.updateNeededReplications() 方法更新数据块复制信息，剩下的工作由名字节点的数据块副本复制线程完成。

更新数据块复制信息最终使用的是 UnderReplicatedBlocks.update() 方法，这个方法的参数比较多，除了待复制的数据块 block 外，其他信息用于计算复制的优先级，优先级计算只需要四个参数，但 update() 方法需要计算更新前后的优先级，并根据优先级调整数据块所属的优先级集合。update() 方法的主要代码如下：

```
synchronized void update(Block block, int curReplicas,
      int decommissionedReplicas, int curExpectedReplicas,
      int curReplicasDelta, int expectedReplicasDelta) {
   int oldReplicas = curReplicas - curReplicasDelta;
   int oldExpectedReplicas = curExpectedReplicas - expectedReplicasDelta;
   int curPri = getPriority(block, curReplicas,
         decommissionedReplicas, curExpectedReplicas);
   int oldPri = getPriority(block, oldReplicas,
         decommissionedReplicas, oldExpectedReplicas);
   ......
   if (oldPri != LEVEL&& oldPri != curPri) { //优先级发生变化
      remove(block, oldPri); //移除
   }
   if (curPri != LEVEL && priorityQueues.get(curPri).add(block)) { //加入
      ..... //输出日志
   }
}
```

数据块副本复制线程实现在 FSNamesystem 的内部类 ReplicationMonitor 中。在 FSNamesystem 处于运行的情况下，它会隔一段时间调用 computeDatanodeWork() 和 processPendingReplications() 两个方法。间隔时间保存在变量 replicationRecheckInterval 中，可由配置项 ${dfs.replication.interval} 指定，默认值为 3 秒。

在具体分析 run() 调用的两个函数前，交代一下 ReplicationMonitor 使用的两个队列。ReplicateQueueProcessingStats 和 InvalidateQueueProcessingStats 都继承自 QueueProcessingStatistics，它们的共同特征是拥有多项任务，但服务器每次处理的任务数有限制，需要多次循环才能完成所有任务。QueueProcessingStatistics 为这样的任务添加了一些度量信息，而 ReplicationMonitor 的两个内部类，分别包装了 neededReplications 和 recentInvalidateSets。代码如下：

```
class ReplicationMonitor implements Runnable {
   static final int INVALIDATE_WORK_PCT_PER_ITERATION = 32;
   static final float REPLICATION_WORK_MULTIPLIER_PER_ITERATION = 2;
   ReplicateQueueProcessingStats replicateQueueStats =
         new ReplicateQueueProcessingStats();
   InvalidateQueueProcessingStats invalidateQueueStats =
         new InvalidateQueueProcessingStats();

   public void run() {
      while (fsRunning) {
         try {
            computeDatanodeWork();
            processPendingReplications();
            Thread.sleep(replicationRecheckInterval);
         } catch (InterruptedException ie) {
         ......
      }
   }
}
```

computeDatanodeWork() 方法用于计算数据节点工作，这里的工作指的是数据块副本删除和副本复制，它们具体实现在 computeInvalidateWork() 和 computeReplicationWork() 中，并被 computeDatanodeWork() 调用。所以，数据块副本删除不需要一个单独的线程，而是和数据块副本复制一起共用一个线程。

用于副本删除的方法 computeInvalidateWork()，最多删除 nodesToProcess 个节点上的待删除数据块副本。在随机选取了 nodesToProcess 个数据节点后，方法调用 invalidateWorkForOneNode() 进行针对数据节点的操作。代码如下：

```
int computeInvalidateWork(int nodesToProcess) {
   int numOfNodes = 0;
   ArrayList<String> keyArray;
   synchronized (this) {
      numOfNodes = recentInvalidateSets.size();
      // 得到 recentInvalidateSets 中的所有键
      keyArray = new ArrayList<String>(recentInvalidateSets.keySet());
   }
   nodesToProcess = Math.min(numOfNodes, nodesToProcess);
   ..... // 随机选择 nodesToProcess 个节点，并将它们放在数组 keyArray 的最前面
   int blockCnt = 0;
   for (int nodeCnt = 0; nodeCnt < nodesToProcess; nodeCnt++) {
```

```
            blockCnt += invalidateWorkForOneNode(keyArray.get(nodeCnt));
        }
        return blockCnt;
    }

    private synchronized int invalidateWorkForOneNode(String nodeId) {
        …… // 处理安全模式、找不到对应数据节点的特殊情况
        Collection<Block> invalidateSet = recentInvalidateSets.get(nodeId);
        …… //invalidateSet 为空，返回

        ArrayList<Block> blocksToInvalidate =
              new ArrayList<Block>(blockInvalidateLimit);
        Iterator<Block> it = invalidateSet.iterator();
        for (int blkCount = 0; blkCount < blockInvalidateLimit && it.hasNext();
              blkCount++) { // 在数据节点上一次最多删除 blockInvalidateLimit 个副本
            blocksToInvalidate.add(it.next());
            it.remove();
        }

        if (!it.hasNext()) { // 该数据节点上已经没有待删除的副本
            recentInvalidateSets.remove(nodeId);
        }

        dn.addBlocksToBeInvalidated(blocksToInvalidate);
        ……
        pendingDeletionBlocksCount -= blocksToInvalidate.size();
        return blocksToInvalidate.size();
    }
```

FSNamesystem.invalidateWorkForOneNode() 也很简单，其工作就是将数据节点上已经失效的数据块，最多选取 blockInvalidateLimit 项，放入 DatanodeDescriptor 的成员变量 invalidateBlocks 中，等待下次心跳，通过名字节点指令下发。

细心的读者会问：数据块删除如果执行失败了怎么办？名字节点上不再拥有副本的记录，而数据节点上的数据块依然存在。不用担心，数据节点长时间没有心跳，意味着它已经故障，重新启动后，该节点会再次上报它管理的所有数据块信息；另外，数据节点会定期进行数据块上报，因为各种原因没有成功删除的副本，通过这样的机会再次进入 recentInvalidateSets 中，并被删除。

回到数据块复制，computeReplicationWork() 的作用和 computeInvalidateWork() 类似，通过 chooseUnderReplicatedBlocks() 选择待复制副本，然后使用 computeReplicationWorkForBlock() 生成复制请求。代码如下：

```
    private int computeReplicationWork(int blocksToProcess)
          throws IOException {
        ……
        // 选择复制的数据块副本
        List<List<Block>> blocksToReplicate =
              chooseUnderReplicatedBlocks(blocksToProcess);
```

```
        //生成复制请求
        int scheduledReplicationCount = 0;
        for (int i = 0; i < blocksToReplicate.size(); i++) {
            for (Block block : blocksToReplicate.get(i)) {
                if (computeReplicationWorkForBlock(block, i)) {
                    scheduledReplicationCount++;
                }
            }
        }
        return scheduledReplicationCount;
    }
```

数据块副本复制请求是有优先级的，chooseUnderReplicatedBlocks() 根据优先级，最多选择 blocksToProcess 个请求，交由 computeReplicationWorkForBlock 处理。方法 computeReplicationWorkForBlock() 看起来比较“长”，也包含了一些冗余的代码。但它的逻辑可以清楚地分为参数检查、选择复制的源数据节点、目标数据节点和生成复制请求等四部分。

参数或者方法执行时需要进行的检查包括：

- 如果数据块不属于任何一个文件或者属于一个已经打开的文件（当复制请求进入队列后，文件被再次打开）时，不需要进行复制。
- 选择数据源或数据目标时，没有找到合适的数据节点。
- 如果当前副本数满足条件，也不需要复制，这里的“当前副本数”包括正在进行复制的数据块副本，虽然复制工作还没有完成，但名字节点乐观地认为复制会成功。

再次强调，名字节点中使用了大量线程，这些线程会修改名字节点上的各类信息，并形成各种各样的临界区，如上面的打开文件检查，或者副本数检查等，都体现了名字节点的复杂性。在 computeReplicationWorkForBlock() 方法的两个同步块中，做了很多相同的检查，如文件是否存在，也是为了防止两个同步块执行的中间，目录树元数据被修改的情况，是合理而且必要的。

在 computeReplicationWorkForBlock() 中，选择复制的源数据节点和目标数据节点分别通过 FSNamesystem.chooseSourceDatanode() 方法和 ReplicationTargetChooser.chooseTarget() 方法进行，成功选择了源和目标后，和数据块删除类似，将请求放入源数据节点的 DatanodeDescriptor 对象中，等待生成名字节点指令。但和删除操作不同的是，进入名字节点的复制请求，必须放入 pendingReplications 中，以防止复制失败。FSNamesystem.neededReplications 记录的复制请求，只有在复制成功后，数据节点中的副本数达到副本系数才会被删除，否则，复制请求还需要保存在 neededReplications 中，以等待下一次复制的机会。代码如下：

```
    boolean computeReplicationWorkForBlock(Block block, int priority) {
        int requiredReplication, numEffectiveReplicas;
        List<DatanodeDescriptor> containingNodes;
        DatanodeDescriptor srcNode;

        synchronized (this) {
```

```
    synchronized (neededReplications) {
        INodeFile fileINode = blocksMap.getINode(block);
        ……
        // 如果数据块不属于任何一个文件或者属于一个被打开的文件，删除复制请求并返回
        requiredReplication = fileINode.getReplication();

        // 选择复制的源
        containingNodes = new ArrayList<DatanodeDescriptor>();
        NumberReplicas numReplicas = new NumberReplicas();
        srcNode = chooseSourceDatanode(block, containingNodes,
              numReplicas);
        ……// 更新计数器和处理没有数据源的情况，没有数据源时，方法返回

        // 理目前有“已经生成复制请求”数据块的复制请求
        numEffectiveReplicas = numReplicas.liveReplicas()
              + pendingReplications.getNumReplicas(block);
        if (numEffectiveReplicas >= requiredReplication) {
            …… // 不需要复制，从 neededReplications 删除复制请求并返回
        }
    }
}

// 选择复制的目标
DatanodeDescriptor targets[]
    = replicator.chooseTarget(requiredReplication -
          numEffectiveReplicas, srcNode, containingNodes, null,
          block.getNumBytes());
…… // 选择复制目标失败，返回

synchronized (this) {
    synchronized (neededReplications) {
        INodeFile fileINode = blocksMap.getINode(block);
        // 再次使用 synchronized(this)，也就需要再次检查数据块
        // 是否不属于任何一个文件或者属于一个被打开的文件
        ……
        // 副本数也可能发生变化，需要再次获得副本数
        requiredReplication = fileINode.getReplication();
        // 再次处理有“已经生成复制请求”数据块的特殊情况，如果不需要复制，返回
        ……

        // 添加复制请求到源数据节点
        srcNode.addBlockToBeReplicated(block, targets);

        for (DatanodeDescriptor dn : targets) {
            dn.incBlocksScheduled();
        }

        // 在 pendingReplications 中做记录，即“已经生成复制请求”了
        pendingReplications.add(block, targets.length);
        ……
```

```
            // 如果已经达到数据块副本系数，从 neededReplications 中移除记录
            if (numEffectiveReplicas + targets.length >= requiredReplication)
            {
               neededReplications.remove(block, priority);
               replIndex--;
            }
            ......
         }
      }
      return true;
   }
```

复制的源数据节点选择实现在 chooseSourceDatanode() 中，限于篇幅的原因，就不具体讨论它的实现。chooseSourceDatanode() 会在当前包含数据块副本的节点中，通过如下原则选择一个数据节点：

- ❑如果副本所属节点处于“正在撤销”，由于上面写数据的请求很少（只有撤销前发起的请求），优先选择；
- ❑接下来选择的是“不忙”的数据节点，即数据节点上的复制请求较少（即 DatanodeDescriptor.replicateBlocks 值的大小）的节点。

很自然，“已撤销”的数据节点不可能被选择为数据源。chooseSourceDatanode() 方法还会将包含数据块副本的数据节点信息保存在参数 containingNodes 中，供 chooseTarget() 使用。它的声明如下：

```
private DatanodeDescriptor chooseSourceDatanode(Block block,
      List<DatanodeDescriptor> containingNodes, NumberReplicas numReplicas)
{
   ......
}
```

ReplicationTargetChooser.chooseTarget() 用于选择一个或多个数据块复制目标，它利用集群的网络布局信息，根据如下原则，进行选择：

- ❑第一个复制目标尽量选择和源在同一个机架上的另一个节点；
- ❑第二个目标必须选择其他机架上的另一个节点；
- ❑第三个复制节点（这种情况比较少出现），如果前面两个目标在同一个机架上，显然应该避开这个机架；否则，选择源机架上的另一个数据节点，作为目标。

这里的原则和新数据块的副本位置安排略有不同。ReplicationTargetChooser 的实现不算复杂，就不再分析它的原理了。

将选择数据块复制目标实现为类，而不是 FSNamesystem 的成员方法的这个设计，为以后替换复制目标选择策略，提供了基础。

数据块复制成功的信息，由数据节点通过 DatanodeProtocol.blockReceived() 上报到名字节点，在该远程方法的实现中，会删除 pendingReplications 中保存的复制请求。代码如下：

```
public synchronized void blockReceived(DatanodeID nodeID, Block block,
      String delHint) throws IOException {
```

```
    ......
    pendingReplications.remove(block);
    ......
}
```

复制失败了该怎么处理呢？如添加复制请求到源数据节点后，该节点发生故障，没有收到名字节点指令，这时，就需要对 pendingReplications 中保存的信息进行定期的检查，以发现可能的复制失败。ReplicationMonitor.run() 定期调用 processPendingReplications()，该方法会获取长时间没有执行完毕的复制请求，并将它们重新加入到 neededReplications 中，以再次产生复制请求。代码如下：

```
void processPendingReplications() {
    // 长时间没有执行完毕的复制请求
    Block[] timedOutItems = pendingReplications.getTimedOutBlocks();
    if (timedOutItems != null) {
        synchronized (this) {
            for (int i = 0; i < timedOutItems.length; i++) {
                NumberReplicas num = countNodes(timedOutItems[i]);
                neededReplications.add(timedOutItems[i], // 重新添加复制请求
                        num.liveReplicas(),num.decommissionedReplicas(),
                        getReplication(timedOutItems[i]));
            }
        }
    }
}
```

### 4. 其他

数据块副本的添加、删除和复制，是名字节点上数据块管理的主要内容。除了上面介绍的方法，这一小节会给出 FSNamesystem 中和数据块副本相关的一些较重要的方法。

客户端关闭为写打开的文件时，需要对该文件当前拥有的副本数进行检查。由第 7 章的内容可知，在写数据时，数据流管道上的数据节点可能出现故障并触发数据块恢复，成功的恢复会找回一些副本，但这个时候，该数据块的副本数会小于副本系数，所以，在关闭文件时，需要使用 checkReplicationFactor() 判断文件是否有需要进行复制的数据块，并在必要的时候产生复制请求。代码如下：

```
private void checkReplicationFactor(INodeFile file) {
    int numExpectedReplicas = file.getReplication();
    Block[] pendingBlocks = file.getBlocks(); // 文件拥有的数据块
    int nrBlocks = pendingBlocks.length;
    for (int i = 0; i < nrBlocks; i++) { // 逐块检查
        NumberReplicas number = countNodes(pendingBlocks[i]);
        if (number.liveReplicas() < numExpectedReplicas) {
            neededReplications.add(pendingBlocks[i], // 需要复制
                    number.liveReplicas(), number.decommissionedReplicas,
                    numExpectedReplicas);
        }
    }
}
```

下面介绍 DatanodeDescriptor.reportDiff() 方法，数据节点启动时，名字节点所做的配合过程中，FSNamesystem.processReport() 调用这个方法获取该数据节点数据块的变化，并根据不同的情况分别进行处理。reportDiff() 方法能够找出的变化包括移除、添加和删除三种。其中，需要删除的数据块副本包含多种情况，如 blocksMap 中已经不存在记录，或者所属的文件已经删除（找不到所属 INode），或者时间戳不匹配等，都需要将副本删除。在排除了等待删除副本的情况后，需要移除的副本，就是这次上报中不包含，而存在于当前数据节点中的副本，添加副本和移除副本正好相反，它出现在 newReport 中，但名字节点中的 DatanodeDescriptor 对象并不管理该数据块副本。不属于上述情况的，就是没有变化的数据块。

在处理的过程中，reportDiff() 方法引入了一个特殊的数据块 deilimiter，作为标志添加到该数据节点管理的数据块列表的最前端。然后，遍历新上报的数据块列表，对于需要删除或者添加的副本，直接加入 toInvalidate 和 toAdd 两个集合中。否则，将数据块移动到 deilimiter 标志前，循环结束后，deilimiter 标志后的就是等待移除的数据块了。具体代码如下：

```
void reportDiff(BlocksMap blocksMap,
      BlockListAsLongs newReport, //新上报的数据块列表
      Collection<Block> toAdd, //应该添加的副本
      Collection<Block> toRemove, //移除
      Collection<Block> toInvalidate) { //删除
   //添加一个特殊的数据块 deilimiter 到 DatanodeDescriptor 中
   //该数据块用于分割原有数据块和新添加数据块
   BlockInfo delimiter = new BlockInfo(new Block(), 1); //特殊数据块
   boolean added = this.addBlock(delimiter);
   assert added : "Delimiting block cannot be present in the node";
   ……  //处理 newReport 为空的特殊状况

   //扫描上报的数据块列表
   Block iblk = new Block(); //工作 Block 对象，将 newReport 中的信息转换为对象
   Block oblk = new Block();
   for (int i = 0; i < newReport.getNumberOfBlocks(); ++i) {
      iblk.set(newReport.getBlockId(i), newReport.getBlockLen(i),
            newReport.getBlockGenStamp(i)); //获得对象
      BlockInfo storedBlock = blocksMap.getStoredBlock(iblk);
      if(storedBlock == null) {
         oblk.set(newReport.getBlockId(i), newReport.getBlockLen(i),
               GenerationStamp.WILDCARD_STAMP); //不带时间戳查找
         storedBlock = blocksMap.getStoredBlock(oblk);
         if (storedBlock != null && storedBlock.getINode() != null &&
               (storedBlock.getGenerationStamp() <= iblk.get……() ||
               storedBlock.getINode().isUnderConstruction())) {
            //不用执行任何操作
         } else {
            storedBlock = null;
         }
      }
      if(storedBlock == null) { //没有找到信息，删除
         toInvalidate.add(new Block(iblk));
```

```
            continue;
        }
        if(storedBlock.findDatanode(this) < 0) {
            if (storedBlock.getNumBytes() != iblk.getNumBytes()) {
                toAdd.add(new Block(iblk)); //添加
            } else {
                toAdd.add(storedBlock);
            }
            continue;
        }
        // 将数据块移动到列表的前面，不用处理
        this.moveBlockToHead(storedBlock);
    }
    // 这次上报中没有出现的数据块，即数据节点上已经没有，而 DatanodeDescriptor 中
    // 还保存的副本
    Iterator<Block> it = new BlockIterator(delimiter.getNext(0), this);
    while(it.hasNext()) {
        BlockInfo storedBlock = (BlockInfo)it.next();
        INodeFile file = storedBlock.getINode();
        if (file == null || !file.isUnderConstruction()) {
            toRemove.add(storedBlock);
        }
    }
    this.removeBlock(delimiter); // 删除特殊数据块
}
```

FSNamesystem.processReport() 获取这些分类信息后，通过 removeStoredBlock() 移除数据块副本，待添加副本则使用 addStoredBlock ()，更新相关数据结构；最后，通过 addToInvalidates() 删除无效副本。注意，这个过程的顺序不能改变。

至此，名字节点第二关系中，数据块删除，特别是多余副本删除涉及的方法，已经全部介绍完毕。让我们再来回顾整个过程：

1）当发现系统中有多余副本时，通过 processOverReplicatedBlock()，记录信息到 FSNamesystem 的成员变量 recentInvalidateSets 和 excessBlocks 中。

2）ReplicationMonitor.run() 会定期将 recentInvalidateSets 中的数据块，放入对应数据节点的描述符对象的 invalidateBlocks 集合中，这时，recentInvalidateSets 中的记录已经被移除。

3）在数据节点的心跳上报处理中，生成名字节点指令。

4）数据节点执行名字节点指令，但删除数据块的结果，并没有立即返回给名字节点。

5）数据节点周期性地汇报它拥有的数据块，通过 reportDiff()，名字节点发现该数据节点上的一些数据块已经不存在了，调用 removeStoredBlock()，该方法会删除 excessBlocks 中的信息。

数据块管理的这些流程涉及了名字节点第二关系中的主要数据结构，包括 BlocksMap、DatanodeDescriptor 和多个管理数据块副本状态的 FSNamesystem 成员变量。和名字节点第一关系的实现对比，这部分内容繁杂，流程性的东西较多，处理过程中会涉及多个数据结构，缺乏名字节点第一关系实现中的层次结构，略显混乱。但只要大家从副本状态图出发，

了解副本各状态的含义和状态间转换的条件，配合数据节点上需要执行的流程，这些数据块管理方法的实现还是清晰且完备的。

## 8.3 远程接口 ClientProtocol 的实现

如下面代码，名字节点实现了多个远程接口，除了与权限相关的 RefreshAuthorization PolicyProtocol 和 RefreshUserMappingsProtocol，这一章已经分析过 NamenodeProtocol 和 DatanodeProtocol 中大部分方法的实现细节，也介绍了名字节点第一关系和第二关系的工作机制，本节研究 ClientProtocol 的实现原理。

```
public class NameNode implements ClientProtocol, DatanodeProtocol,
      NamenodeProtocol, FSConstants, RefreshAuthorizationPolicyProtocol,
      RefreshUserMappingsProtocol {
   ......
```

在第 6 章中，将 ClientProtocol 中的方法分为两组，实现 Hadoop 文件系统的远程方法和实现多功能工具 dfsadmin 的远程方法。实现文件系统的远程方法可以进一步区分，将读写文件数据和其他操作分开，本节按照上述归类，对名字节点的 ClientProtocol 实现进行研究。

### 8.3.1 文件和目录相关事务

在 Java 系统中，对文件数据的读写是通过输入 / 输出流实现的，其他的文件管理功能由 java.io.File 封装。在 Hadoop 文件系统中，这部分的功能由 FileSystem 的抽象方法定义，并由具体文件系统实现。

**1. 几个典型方法的实现**

本节会分析几个典型的 ClientProtocol 远程方法实现，它们分别在 HDFS 的目录树上进行增、删、查、改等操作。

ClientProtocol.mkdirs() 可在目录树中创建一个子目录。NameNode.mkdirs() 实现了远程接口方法，但主要的创建目录工作由 FSNamesystem.mkdirs() 和 FSNamesystem.mkdirsInternal() 完成。mkdirsInternal() 方法在目录树上执行实际的创建动作前需要进行以下检查：

- ❑用户有访问路径的权限（checkTraverse() 方法）。
- ❑通过 dir.isDir() 方法判断目录不存在，如果目录已经存在，可以直接返回。
- ❑名字节点不处于安全模式（后面还会介绍安全模式），可通过 isInSafeMode() 判断。
- ❑检查 DFSUtil.isValidName() 的结果，保证输入的目录名合法。
- ❑用户在将创建目录的父目录上有写权限，由 checkAncestorAccess() 判断。
- ❑利用 checkFsObjectLimit()，检查创建目录后是否会达到系统容量的极限。系统容量极限保存在 FSNamesystem.maxFsObjects 中，可由 ${dfs.max.objects} 设置，默认值为 0，即没有极限。进行上述检查可以保证 HDFS 目录数中 INode 的数量和数据块的数

量和小于极限值。

完成上述六项检查后，mkdirsInternal() 才调用 FSDirectory.mkdirs()，在目录树的对应位置添加一个新的 INodeDirectory 对象。方法 mkdirs() 在 mkdirsInternal() 成功返回后，会通过 FSEditLog.logSync()，保证创建目录的日志记录持久化后才返回。代码如下：

```
public boolean mkdirs(String src, PermissionStatus permissions)
      throws IOException {
   boolean status = mkdirsInternal(src, permissions);
   getEditLog().logSync();
   ……
   return status;
}

private synchronized boolean mkdirsInternal(String src,
      PermissionStatus permissions) throws IOException {
   ……
   if (isPermissionEnabled) {
      checkTraverse(src); //是否有权限
   }
   if (dir.isDir(src)) {
      return true; //目录已经存在，返回 true（即使目录并没有被创建）
   }
   if (isInSafeMode()) //是否是安全模式
      throw new SafeModeException("Cannot create directory " + ……);
   if (!DFSUtil.isValidName(src)) { //是否是一个合法的名字
      throw new IOException("Invalid directory name: " + src);
   }
   if (isPermissionEnabled) { //是否是写权限
      checkAncestorAccess(src, FsAction.WRITE);
   }

   checkFsObjectLimit(); //是否达到系统容量的极限

   if (!dir.mkdirs(src, permissions, false, now())) { //创建对象
      throw new IOException("Invalid directory name: " + src);
   }
   return true;
}
```

在分析 FSDirectory 时讲解过 delete() 如何在目录树上删除项目。FSNamesystem.deleteInternal() 在进行一番检查后，调用 FSDirectory.delete()[⊖]，删除以路径 src 为根的子目录树。和 FSNamesystem.mkdirs() 类似，由于操作修改了目录树，需要等待日志持久化。

如果被删除的子目录树中有文件，它们拥有的数据块已经在 FSDirectory.delete() 调用中，通过 FSNamesystem.removePathAndBlocks() 进入无效数据块副本 invalidateSet 中。deleteInternal() 方法也就不需要关注这些数据块副本。代码如下：

⊖ delete() 方法有两个重载形式，分别是 delete(String src) 和 delete(String src, boolean recursive)。前者已经不建议使用。

```
public boolean delete(String src, boolean recursive) throws IOException {
   //递归删除条件 recursive 满足否?
   if ((!recursive) && (!dir.isDirEmpty(src))) {
      throw new IOException(src + " is non empty");
   }
   boolean status = deleteInternal(src, true);
   getEditLog().logSync();
   ……
   return status;
}

synchronized boolean deleteInternal(String src,
      boolean enforcePermission) throws IOException {
   ……
   if (isInSafeMode()) //是否是安全模式
      throw new SafeModeException("Cannot delete " + src, safeMode);
   if (enforcePermission && isPermissionEnabled) {
      checkPermission(src, false, null,//是否有写权限
            FsAction.WRITE, null, FsAction.ALL);
   }

   return dir.delete(src);
}
```

远程方法 getFileInfo() 用于获取文件 / 目录属性，由于它不对目录树进行修改，所以，在权限检查后就可以直接调用 FSDirectory 的同名方法，并返回该方法的返回值。代码如下：

```
HdfsFileStatus getFileInfo(String src) throws IOException {
   if (isPermissionEnabled) {
      checkTraverse(src);
   }
   return dir.getFileInfo(src);
}
```

修改文件 / 目录的文件主标识符和用户组标识符使用的是 setOwner()，它的实现如下：

```
public void setOwner(String src, String username, String group)
      throws IOException {
   synchronized (this) {
      if (isInSafeMode()) //是否是安全模式
         throw new SafeModeException("Cannot set owner for " + ……);
      FSPermissionChecker pc = checkOwner(src);
      if (!pc.isSuper) {
         …… //用户权限检查
         if (group != null && !pc.containsGroup(group)) { //组权限检查
            throw new AccessControlException("User does not belong……");
         }
      }
      dir.setOwner(src, username, group);
   }
   getEditLog().logSync();
```

```
    ......
}
```

### 2. 其他方法

上一节已经分析了 mkdirs()、delete()、getFileInfo() 和 setOwner() 四个 ClientProtocol 远程方法的实现，接口中，和文件 / 目录 API 相关的还有其他十个方法，下面简单介绍它们的实现要点。

ClientProtocol.setReplication() 设置文件的数据块副本系数，由于新副本系数可以高于当前副本系数，也可以低于当前系数，所以，需要根据情况调用 updateNeededReplications() 复制副本，或者 processOverReplicatedBlock() 删除多余副本。

setPermission()、setQuota()、setTimes() 等方法的实现和 setOwner() 类似，用于修改文件权限、目录配额和文件的修改时间 / 访问时间。其中，在权限检查打开的情况下，setQuota() 的调用者需要有超级用户的权限，具体的修改动作由 FSDirectory 的成员函数 setQuota() 和 unprotectedSetQuota() 执行。

FSNamesystem.renameToInternal() 用于修改文件名，在进行一番检查后，通过 FSDirectory.renameTo() 修改目录树。如果改动名字的文件已经为写打开，而且没有关闭，还需要修改租约管理器中保存的信息，也就是说，修改文件名不会影响数据节点正在写入的文件内容。

getListing()、getStats()、getDatanodeReport()、getPreferredBlockSize() 和 getContentSummary() 分别用于获取具体目录下的目录项、文件系统的统计信息、数据节点的状态信息、文件的副本系数和文件 / 目录的资源占用情况，它们都不对文件目录树进行任何修改，所以，这些方法的实现和 getFileInfo() 区别不大，通过 FSNamesystem 中的相关方法获取结果并返回。

这部分代码都遵循一个模式，在必要的检查后进行具体的操作，检查包括调用者是否有权限、文件系统是否处于安全模式、是否满足配额限制等。在复杂的名字节点的实现中，这部分的内容简单而且好理解。

## 8.3.2　读数据使用的方法

和上一节讨论的那些远程方法不同，名字节点需要为文件读写准备多个远程接口，如客户端读文件时，就会使用 getBlockLocations() 和 reportBadBlocks() 方法。

客户端在和数据节点建立流式接口的 TCP 连接，读取文件数据前需要定位数据的位置，使用的就是远程方法 getBlockLocations()。它的返回值 LocatedBlocks 对象，包含了一系列的 LocatedBlock 实例，通过这些信息，客户端就知道需要到哪些数据节点上去获取数据。代码如下：

```
public LocatedBlocks getBlockLocations(String src, long offset, long length)
      throws IOException {
  myMetrics.incrNumGetBlockLocations(); // 更新度量信息
```

```
    return namesystem.getBlockLocations(getClientMachine(),
        src, offset, length);
}
```

NameNode.getBlockLocations() 调用的其实还是 FSNamesystem 中的同名方法，但这两个方法的参数不一样，NameNode 中的方法会调用 getClientMachine() 获取客户端地址，后续的处理逻辑中，会使用这个地址对 LocatedBlocks 对象中的数据节点列表进行排序，保证客户端能访问离它位置最近的数据块副本。

FSNamesystem.getBlockLocations() 有三个重载的方法，下面给出了其中的两个。

第一个 getBlockLocations() 会对即将返回的数据节点列表进行整理，尽量找一个离客户端网络距离近的节点放在数组的开始，它使用了前面分析过的 NetworkTopology.pseudoSortByDistance() 方法。第二个 getBlockLocations() 方法增加了权限检查和参数检查。相关代码如下：

```
LocatedBlocks getBlockLocations(String clientMachine, String src,
      long offset, long length) throws IOException {
   LocatedBlocks blocks = getBlockLocations(src, offset, length,
         true, true);
   if (blocks != null) {
      // 对数据块所在的数据节点进行排序
      DatanodeDescriptor client =
            host2DataNodeMap.getDatanodeByHost(clientMachine);
      for (LocatedBlock b : blocks.getLocatedBlocks()) {
         clusterMap.pseudoSortByDistance(client, b.getLocations());
      }
   }
   return blocks;
}

public LocatedBlocks getBlockLocations(String src, long offset, long length,
      boolean doAccessTime, boolean needBlockToken) throws IOException {
   if (isPermissionEnabled) {// 检查读权限
      checkPathAccess(src, FsAction.READ);
   }
   ..... // 检查输出参数 offset 和 length
   final LocatedBlocks ret = getBlockLocationsInternal(src, offset,
         length, Integer.MAX_VALUE, doAccessTime, needBlockToken);
   .....
   return ret;
}
```

FSNamesystem.getBlockLocationsInternal() 才是真正获取返回 LocatedBlocks 对象的地方。这个方法出乎意外地复杂，方法入口部分代码如下：

```
private synchronized LocatedBlocks getBlockLocationsInternal(String src,
      long offset,long length,int nrBlocksToReturn,
      boolean doAccessTime,boolean needBlockToken)throws IOException {
   INodeFile inode = dir.getFileINode(src);
```

```
if(inode == null) { // 文件不存在
   return null;
}
if (doAccessTime && isAccessTimeSupported()) {
   dir.setTimes(src, inode, -1, now(), false);
} // 更新文件的访问时间
Block[] blocks = inode.getBlocks();
…… // 处理空文件的情况，会提前返回
List<LocatedBlock> results;
results = new ArrayList<LocatedBlock>(blocks.length);
```

按照惯例，方法入口处处理各种特殊情况，如要读取的文件不存在或文件为空等，上面代码中，方法还调用了 FSDirectory.setTimes()，更新文件的访问时间。

接下来的代码（如下），根据读取数据起始位置偏移量 offset，计算开始的数据块，并保存在变量 curBlk 中。当然，如果起始位置比文件还大，方法就可以返回了，返回值为空。

```
int curBlk = 0; // 偏移量 offset 的数据所在的数据块
long curPos = 0, blkSize = 0;
int nrBlocks = (blocks[0].getNumBytes() == 0) ? 0 : blocks.length;
for (curBlk = 0; curBlk < nrBlocks; curBlk++) {
   blkSize = blocks[curBlk].getNumBytes();
   assert blkSize > 0 : "Block of size 0";
   if (curPos + blkSize > offset) {
      break;
   }
   curPos += blkSize;
}

if (nrBlocks > 0 && curBlk == nrBlocks)// 起始位置大于文件长度
   return null;
```

FSNamesystem.getBlockLocationsInternal() 方法最复杂的逻辑出现在下面，其目的是找出读取数据范围内的所有可用数据块副本，并生成 LocatedBlocks 对象。如果数据块是正在构造文件的最后一个块，我们认为当前参与数据流管道的各个数据节点都是可用节点；其他情况，需先通过名字节点第二关系中的方法，获得包含数据块副本的节点数、损坏的副本数等信息，判断是否有可用的“正常副本”。如果有，找出这些副本所在的数据节点并保存在 machineSet 中；如果没有，machineSet 则包含了拥有该数据块的所有节点，同时，LocatedBlock 对象的标志位 blockCorrupt 会设置为 true。代码如下：

```
long endOff = offset + length;
do {
   // 包含副本的数据节点数
   int numNodes = blocksMap.numNodes(blocks[curBlk]);
   // 副本损坏的数据节点数
   int numCorruptNodes = countNodes(blocks[curBlk]).corruptReplicas();
   ……
   DatanodeDescriptor[] machineSet = null;
   boolean blockCorrupt = false;
```

```
    if (inode.isUnderConstruction() && curBlk == blocks.length - 1
        && blocksMap.numNodes(blocks[curBlk]) == 0) {
      // 处理还没有提交，客户端正在写数据的数据块
      INodeFileUnderConstruction cons
            = (INodeFileUnderConstruction)inode;
      machineSet = cons.getTargets();
      blockCorrupt = false;
    } else {
       blockCorrupt = (numCorruptNodes == numNodes);// 数据块全部损坏?
      int numMachineSet = blockCorrupt ? numNodes :
            (numNodes - numCorruptNodes);
       machineSet = new DatanodeDescriptor[numMachineSet];
      if (numMachineSet > 0) {
        numNodes = 0;
        for(Iterator<DatanodeDescriptor> it =
          blocksMap.nodeIterator(blocks[curBlk]); it.hasNext();)
        {
          DatanodeDescriptor dn = it.next();
          boolean replicaCorrupt =
            corruptReplicas.isReplicaCorrupt(blocks[curBlk],dn);
          if (blockCorrupt || (!blockCorrupt && !replicaCorrupt))
            machineSet[numNodes++] = dn;
        }
      }
    }

      LocatedBlock b = new LocatedBlock(blocks[curBlk], machineSet,
            curPos, blockCorrupt);
      ..... // 与安全相关的处理
      results.add(b);
      curPos += blocks[curBlk].getNumBytes();
      curBlk++;
    } while (curPos < endOff
          && curBlk < blocks.length
          && results.size() < nrBlocksToReturn);

    return inode.createLocatedBlocks(results);
  }
```

通过 getBlockLocations() 的返回值，客户端就可以和数据节点联系，并通过数据节点的流式接口，读取文件内容。

和客户端读文件相关的另一个方法是 ClientProtocol.reportBadBlocks()，当客户端读取数据发现错误，即对数据进行校验失败时，通过该方法，将情况汇报给名字节点。这个方法也存在于远程接口 DatanodeProtocol 中，它们的方法签名是一致的，也就是说，当 IPC 请求发送到名字节点时，它们调用的是相同的 NameNode.reportBadBlocks() 方法，不需要区分这是一个 ClientProtocol 接口上的远程调用，还是 DatanodeProtocol 上的调用。代码如下：

```
public void reportBadBlocks(LocatedBlock[] blocks) throws IOException {
  for (int i = 0; i < blocks.length; i++) {
```

```
            Block blk = blocks[i].getBlock(); // 无效的数据块
            DatanodeInfo[] nodes = blocks[i].getLocations();
            for (int j = 0; j < nodes.length; j++) {
                DatanodeInfo dn = nodes[j];  // 副本所在的数据节点
                namesystem.markBlockAsCorrupt(blk, dn);
            }
        }
    }
```

NameNode.reportBadBlocks() 调用 markBlockAsCorrupt() 标记损坏副本，后续流程交给名字节点第二关系中的相关逻辑执行。

### 8.3.3　写数据使用的方法

写数据远比读数据复杂，名字节点不但需要处理来自 ClientProtocol 的请求，而且还需要处理 DatanodeProtocol 上的远程调用，同时，还要维护客户端的写文件租约，是 Hadoop 文件系统中最复杂的流程。

#### 1. 写文件：打开文件

在第 6 章中已经简单地介绍了 ClientProtocol 中和写文件有关的远程方法，它们是 create()、append()、addBlock()、abandonBlock()、fsync()、complete()、renewLease() 和 recoverLease()。数据节点和名字节点间的 DatanodeProtocol 远程接口，有 blocksBeingWrittenReport()、blockReceived()、nextGenerationStamp() 和 commitBlockSynchronization() 参与到写文件的过程中。这是 HDFS 中涉及远程方法最多、最复杂的一个过程。

我们从打开文件开始研究名字节点对 HDFS 文件写操作的支持。create() 创建并打开一个空文件，而 append() 则用于在一个已经存在的文件上追加数据。

NameNode.create() 其实调用了 FSNamesystem.startFile()，在目录树中创建文件。代码如下：

```
public void create(String src, FsPermission masked, String clientName,
        boolean overwrite, boolean createParent, short replication,
        long blockSize) throws IOException {
    String clientMachine = getClientMachine();// 发起请求的机器
    ……
    if (!checkPathLength(src)) { // 检查文件长度
        throw new IOException("……");
    }
    namesystem.startFile(……);
    ……
}
```

如下面代码，startFile() 方法的主要工作由 startFileInternal() 完成，和其他修改目录树的 FSNamesystem 方法类似，调用结束后，需要通过 logSync() 同步日志。

```
void startFile(String src, PermissionStatus permissions, String holder,
        String clientMachine, boolean overwrite, boolean createParent,
```

```
        short replication, long blockSize) throws IOException {
    startFileInternal(……);
    getEditLog().logSync();
    ……
}
```

注意，方法 startFileInternal() 也被 ClientProtocol.append() 的实现使用。和 startFile() 一样，这个方法有很长的参数表，参数的具体含义如下：打开（创建 / 追加）文件的路径保存在字符串 src 中；类型为 PermissionStatus 的参数 permissions 提供了（创建的）新文件的访问权限；holder 是客户端标识，用于租约管理中；而客户端节点名 clientMachine 只用于日志输出，startFileInternal() 的业务逻辑不会使用到该信息。接下来是三个标志位：overwrite 用于指示如果原文件已经存在，是否覆盖该文件，明显，overwrite 只用于创建文件的场景；append 为 true，表明是 ClientProtocol.append() 在调用方法；标志位 createParent 可决定如果文件的父目录不存在，是否创建目录。最后两个参数，分别是文件副本系数 replication 和文件的数据块大小 blockSize。

startFileInternal() 方法的关键逻辑是在目录树指定位置插入一个新的 INodeFileUnderConstruction 节点（文件创建）或将目录树原有的 INodeFile 转变成构建状态节点，然后在租约管理器中添加记录。

但在执行上述逻辑之前，startFileInternal() 需要进行大量的检查，包括：

- ❑文件系统不能处于安全模式，安全模式只提供 HDFS 的只读视图。
- ❑通过 DFSUtil.isValidName() 判断文件名是否合法。
- ❑如果创建 / 打开文件的目标 src 在目录树中已经存在，它不能是一个目录。
- ❑操作权限检查，对于创建操作，用户在父目录上有写权限；对于追加操作，或者是覆盖创建操作（文件存在时），用户在该文件上必须有写权限。
- ❑标志位 createParent 为 false 时，需要判断待创建文件的父目录是否存在。
- ❑调用 FSNamesystem.recoverLeaseInternal()，判断文件是否已经被其他客户端打开，防止同时有多个客户端写同一个文件。
- ❑通过 verifyReplication() 检查等待打开的文件的副本数是否落在有效范围内。
- ❑如果是通过 append() 打开文件，需要保证文件已经存在并且不是目录。
- ❑如果是文件创建，待创建的文件不应该存在，当然，如果允许覆盖创建（标志位 overwrite 的值为 true），则删除原文件。

startFileInternal() 方法代码如下：

```
private synchronized void startFileInternal(String src,
      PermissionStatus permissions, String holder, String clientMachine,
      boolean overwrite, boolean append, boolean createParent,
      short replication, long blockSize) throws IOException {
    ……
    if (isInSafeMode()) //安全模式
      throw new SafeModeException("Cannot create file" + src, safeMode);
    if (!DFSUtil.isValidName(src)) { //文件名是否合法
```

```
    throw new IOException("Invalid file name: " + src);
}
boolean pathExists = dir.exists(src);
if (pathExists && dir.isDir(src)) {// 目标存在时，不能是一个目录
    throw new IOException("……");
}
if (isPermissionEnabled) {// 权限检查
    ……
}
if (!createParent) {// 父节点存在
    verifyParentDir(src);
}

try {
    INode myFile = dir.getFileINode(src);

    // 租约检查，文件没有被其他客户端打开
    recoverLeaseInternal(myFile, src, holder, clientMachine, false);

    try {
        verifyReplication(src, replication, clientMachine);// 副本数合法
    } catch (IOException e) {
        throw new IOException("failed to create " + e.getMessage());
    }
    if (append) {
        if (myFile == null) {// 文件存在
            throw new FileNotFoundException(……);
        } else if (myFile.isDirectory()) {// 是目录，抛异常
            throw new IOException(……);
        }
    } else if (!dir.isValidToCreate(src)) {// 文件存在
        if (overwrite) {
            delete(src, true);// 删除文件
    } else {
        throw new IOException(……);
    }
}

DatanodeDescriptor clientNode =
        host2DataNodeMap.getDatanodeByHost(clientMachine);

if (append) {
    INodeFile node = (INodeFile) myFile;
    INodeFileUnderConstruction cons =
              new INodeFileUnderConstruction(……);
        dir.replaceNode(src, node, cons);
        leaseManager.addLease(cons.clientName, src);
    } else {
        checkFsObjectLimit();// 是否达到系统容量
        long genstamp = nextGenerationStamp();
```

```
            INodeFileUnderConstruction newNode = dir.addFile(……);
            if (newNode == null) {
               throw new IOException(……);
            }
            leaseManager.addLease(newNode.clientName, src);
            ……
         }
      } catch (IOException ie) {
         ……
      }
   }
```

在完成了上述多项检查后，对于 append() 操作，需要将原有目录树上的 INodeFile 对象，替换为 INodeFileUnderConstruction 对象，然后通过 LeaseManager.addLease() 在租约管理器中添加一条记录。

如果是创建操作，通过 FSDirectory.addFile()，往目录树中添加新文件的 INodeFileUnderConstruction 对象并在日志中记录文件打开操作，当然，也需要添加记录到租约管理器。注意，添加文件时获取的新版本号 genstamp，在 addFile() 中并没有使用到，这是 HDFS 代码演进过程中遗留下的问题。

create() 调用的 FSDirectory.addFile() 方法代码如下：

```
INodeFileUnderConstruction addFile(……)throws IOException {
   ……
   //默认创建父目录
   if (!mkdirs(new Path(path).getParent().toString(),……)) {
      return null;
   }
   INodeFileUnderConstruction newNode
         = new INodeFileUnderConstruction(……);//创建新节点
   synchronized (rootDir) {
      newNode = addNode(path, newNode, -1, false); //在目录树上插入节点
   }
   …… //处理 newNode 为空的情况后返回
   fsImage.getEditLog().logOpenFile(path, newNode); //记录日志
   ……
   return newNode;
}
```

ClientProtocol.append() 也调用 startFileInternal()，但和 create() 不同，追加操作需要返回 LocatedBlock 对象，客户端通过这个对象建立数据流管道，往数据块中追加数据。所以，appendFile() 在调用了 startFileInternal() 方法后，剩下的工作就是构造返回的 LocatedBlock 对象。

这个过程涉及了名字节点第二关系，因此有点繁琐。首先执行的操作是再次检查租约，以防止出现 startFileInternal() 调用结束后文件被删除的特殊情况。如果文件的最后一个数据块已经被写满，append() 将返回 null；否则，在 blocksMap 中，将当前保存该数据块的数据节点找出来，保持在数组 targets 中。然后，在上述数据节点的描述符中移除该数据块

的信息，并把 targets 数组保存在已经打开的目录树节点中，即 INodeFileUnderConstruction 对象中。前面解读方法 getBlockLocationsInternal() 时，如果客户端要读取的是正在构造文件的最后一个数据块，则不能从 blocksMap 获取数据块的位置信息，而需要使用 INodeFileUnderConstruction 实例中的数据流管道成员，原因也在此。

根据上述信息和当前数据块的剩余可写空间，即可构造返回的 LocatedBlock 对象。

appendFile() 方法代码如下：

```
LocatedBlock appendFile(String src, String holder, String clientMachine)
      throws IOException {
   if (supportAppends == false) { //HDFS 是否支持追加操作，可以通过配置指定
      throw new IOException(……);
   }
   startFileInternal(src, null, holder, clientMachine, false, true, false,
         (short) maxReplication, (long) 0);
    getEditLog().logSync();

    LocatedBlock lb = null;// 返回的 LocatedBlock 对象
   synchronized (this) {
      // 再次检查租约
      INodeFileUnderConstruction file = checkLease(src, holder);

      Block[] blocks = file.getBlocks();
      if (blocks != null && blocks.length > 0) {
         Block last = blocks[blocks.length - 1]; // 最后一个数据块
         BlockInfo storedBlock = blocksMap.getStoredBlock(last);

         if (file.getPreferredBlockSize() > storedBlock.getNumBytes()) {
            // 最后的数据块还有空间
            long fileLength = file.computeContentSummary().getLength();
            DatanodeDescriptor[] targets =
                  new DatanodeDescriptor[blocksMap.numNodes(last)];
            Iterator<DatanodeDescriptor> it
                  = blocksMap.nodeIterator(last);
            for (int i = 0; it != null && it.hasNext(); i++) {
               targets[i] = it.next(); // 找出所有拥有该数据块的数据节点
            }
            for (int i = 0; i < targets.length; i++) {
               targets[i].removeBlock(storedBlock); // 移除数据块信息
            }
            file.setLastBlock(storedBlock, targets);

            lb = new LocatedBlock(last, targets,
                  fileLength - storedBlock.getNumBytes());
            ……
            updateNeededReplications(last, 0, 0); // 停止副本上的复制任务

            for (DatanodeDescriptor dd : targets) { // 停止删除多余数据副本
               String datanodeId = dd.getStorageID();
```

```
                Collection<Block> v =
                    recentInvalidateSets.get(datanodeId);
                if (v != null && v.remove(last)) {
                    if (v.isEmpty()) {
                        recentInvalidateSets.remove(datanodeId);
                    }
                    pendingDeletionBlocksCount--;
                }
            }
        }
    }
    ......
    return lb;
}
```

appendFile() 方法后半部分的代码用于处理追加操作对名字节点第二关系的影响。首先停止打开数据块上可能的复制，即使该数据块副本数不足。当然，如果有打开数据块的副本删除请求，也需要取消。

添加数据块副本的 FSNamesystem.addStoredBlock() 方法中，有一个通过副本大小决定副本有效性的方法，即处理数据节点提交数据块（包含数据块复制的情形）时，要比较上报的新数据块副本和 blocksMap 中保存的现有副本的大小，大者保留，并删除小的副本。采用这样的策略的原因是：appendFile() 虽然会停止打开副本上的复制任务，但对于已经在执行的复制任务，名字节点没有方法终止。复制任务成功完成后，会上报结果，而追加操作提交数据块的结果，也是通过 DatanodeProtocol.blockReceived() 通知到名字节点。这种情况下，就会出现往名字节点第二关系中添加不同大小副本的情况，显然，这里应该采用大者保留的策略，即使追加操作没有往文件中添加任何数据，上述策略也能正常工作。

无论是创建文件或者追加文件，远程方法调用成功结束后，名字节点上下面的判断为真：

❑该文件的目录树节点类型为 INodeFileUnderConstruction。

❑租约管理器中有该打开文件的记录。

由 appendFile() 的处理逻辑可知，如果打开文件当前正在写数据，则这些数据块都保存在 INodeFileUnderConstruction 对象中。在数据块提交前，名字节点第二关系中的数据节点描述符、无效数据块副本集合和等待复制请求队列都不保存这些数据块副本的信息，但数据块映射中还是保存了数据块的信息，只不过不能通过数据块映射，获取副本的位置信息。

### 2. 写文件：数据块提交、添加 / 放弃数据块

获得 ClientProtocol.append() 返回的非空 LocatedBlock 对象后，客户端就可以根据对象中的字段，利用数据节点的流式接口写数据了。数据节点成功接收到一个数据块后，必须使用远程方法 ClientProtocol.blockReceived()，向名字节点提交数据块。为减少到名字节点的请求量，数据节点将多个提交合并成一个请求。NameNode.blockReceived() 通过循环，为每一个提交调用一次 FSNamesystem 中的同名方法。这个方法其实是一个名字节点第二关系方法，由于它也被数据块复制请求使用，所以方法不会涉及打开文件的

INodeFileUnderConstruction 对象，也不会访问租约管理器。

由于添加数据块所需的逻辑都实现在 addStoredBlock() 中，blockReceived() 也就变得很简洁，调用 addStoredBlock() 方法，完成数据块提交。同时，这个方法具有典型的名字节点第二关系方法特征，它会判断数据节点当前的状态，也会修改 pendingReplications 中的信息。代码如下：

```
public synchronized void blockReceived(DatanodeID nodeID, Block block,
      String delHint) throws IOException {
   DatanodeDescriptor node = getDatanode(nodeID);
   …… // 处理找不到数据节点的特殊情况
   if (shouldNodeShutdown(node)) {// 数据节点应该关闭?
      setDatanodeDead(node);
      throw new DisallowedDatanodeException(node);
   }

   // 获取添加数据块时需要的 delHintNode 信息
   DatanodeDescriptor delHintNode = null;
   if (delHint != null && delHint.length() != 0) {
      delHintNode = datanodeMap.get(delHint);
      ……
   }

   pendingReplications.remove(block);
   addStoredBlock(block, node, delHintNode);
   ……
}
```

数据块提交意味着对某一个数据块副本的写操作已经结束，副本的位置信息进入名字节点第二关系。如果客户端还希望继续往文件中写入数据，这个时候就需要使用 ClientProtocol.addBlock()，分配新的数据块。通过 create() 创建一个新文件后，如果有数据写入，客户端也会通过这个方法，获得文件的第一个数据块。

getAdditionalBlock() 方法实现了 addBlock() 的主要功能，它们的参数表是一样的，包括：申请分配新的数据块的文件（即 src）和客户端 clientName，excludedNodes 包含了一些数据节点，分配数据块时，要排除这些节点。这个方法的参数不多，业务逻辑需要的参数，如数据块大小、打开文件的客户端等，都可以在 INodeFileUnderConstruction 对象中找到。

和打开文件相比，getAdditionalBlock() 需要执行的检查少了很多，这些检查包括：是否达到系统容量、增加数据块的文件是否已经打开、文件的写进度和系统是否处于安全模式等。

文件写进度检查 checkFileProgress() 是为了保证数据安全引入的一个机制，分配新数据块时，要保证文件倒数第二个数据块的副本数达到系统最小副本数 minReplication（默认值为 1）。

---

**注意**　这里检查的是倒数第二个数据块，而不是最后一个数据块，由于数据节点会将多个提交合并成一个 ClientProtocol.blockReceived() 请求，带来了一定的延迟，所以，客户端申请新数据块时，最后一个数据块往往还没有提交，也就不对其进行检查。

checkFileProgress() 方法在系统负载很高或者有大量数据节点失效（如机架故障）的情况下会失败。checkFileProgress() 方法还提供了另一种检查，当参数 checkall 为 true 时，它会检查文件所有的数据块副本数，客户端关闭文件时，会进行这样的检查。

getAdditionalBlock() 和 checkFileProgress() 代码如下：

```
public LocatedBlock getAdditionalBlock(String src, String clientName,
      List<Node> excludedNodes) throws IOException {
   long fileLength, blockSize;
   int replication;
   DatanodeDescriptor clientNode = null;
   Block newBlock = null;
   ……
   synchronized (this) {
      checkFsObjectLimit();// 检查是否达到系统容量
      INodeFileUnderConstruction pendingFile
            = checkLease(src,clientName); // 检查租约
      if (!checkFileProgress(pendingFile, false)) {
         throw new NotReplicatedYetException(……);
      }
      …… // 从 pendingFile 中获得文件当前长度 fileLength、数据块大小 blockSize、
          // 客户端 clientNode 和副本系数 replication 等
   }

   DatanodeDescriptor targets[] = replicator.chooseTarget(replication,
         clientNode, excludedNodes, blockSize);
   if (targets.length <this.minReplication) { // 没有分配到足够的数据块副本
      throw new IOException(……);
   }

   synchronized (this) {
      if (isInSafeMode()) { // 安全模式
         throw new SafeModeException(……);
      }
      // 重新进入同步块，再次检查
      INode[] pathINodes = dir.getExistingPathINodes(src);
      int inodesLen = pathINodes.length;
      checkLease(src, clientName, pathINodes[inodesLen - 1]);
      INodeFileUnderConstruction pendingFile
            = (INodeFileUnderConstruction) pathINodes[inodesLen - 1];
      if (!checkFileProgress(pendingFile, false)) {
         throw new NotReplicatedYetException(……);
      }
      newBlock = allocateBlock(src, pathINodes); // 为文件分配一个新的数据块
      pendingFile.setTargets(targets); // 该数据块的数据流管道成员
      ……
   }
   LocatedBlock b = new LocatedBlock(newBlock, targets, fileLength);
   ……
   return b;
```

```
}

synchronized boolean checkFileProgress(INodeFile v, boolean checkall) {
   if (checkall) { //检查文件拥有的所有数据块
      for (Block block : v.getBlocks()) {
         if (blocksMap.numNodes(block) <this.minReplication) {
            return false;
         }
      }
   } else { //只检查倒数第二个数据块
      Block b = v.getPenultimateBlock();
      if (b != null) {
         if (blocksMap.numNodes(b) <this.minReplication) {
            return false;
         }
      }
   }
   return true;
}
```

除去必要的参数获取和执行一些必要的检查，getAdditionalBlock() 的分配数据块过程并不复杂，首先通过 ReplicationTargetChooser 的 chooseTarget() 方法选择数据块副本的保存位置，接下来分配一个新的数据块，和保存位置一起添加到 INodeFileUnderConstruction 对象中。

在 HDFS 中，数据块的标识是怎么确定的呢？分配新数据块的 allocateBlock() 方法，以随机的方式产生数据块 ID 号，当然，如果随机产生的标识和系统中已有的标识冲突，那就重新生成，直到标识有效为止。代码如下：

```
private Block allocateBlock(String src, INode[] inodes)
      throws IOException {
   Block b = new Block(FSNamesystem.randBlockId.nextLong(), 0, 0);
   while (isValidBlock(b)) { //标识冲突?
      b.setBlockId(FSNamesystem.randBlockId.nextLong());
   }
   b.setGenerationStamp(getGenerationStamp());
   b = dir.addBlock(src, inodes, b);
   ......
   return b;
}
```

分配的新数据块，通过 FSDirectory.addBlock() 添加到 INodeFileUnderConstruction 对象中，也添加到名字节点的 blocksMap 对象中。也就是说，数据块一经分配，就会立即进入 blocksMap 中。代码如下：

```
Block addBlock(String path, INode[] inodes, Block block)
      throws IOException {
   ......
   synchronized (rootDir) {
      INodeFile fileNode = (INodeFile) inodes[inodes.length - 1];
      ......
```

```
            namesystem.blocksMap.addINode(block, fileNode);// 添加到 blocksMap
            BlockInfo blockInfo = namesystem.blocksMap.getStoredBlock(block);
            fileNode.addBlock(blockInfo);// 添加到 INodeFileUnderConstruction
            ……
        }
        return block;
    }
```

最后介绍的是 abandonBlock() 方法，客户端根据 addBlock() 返回的 LocatedBlock 对象，创建数据流管道失败时，通过这个方法放弃已经申请的数据块，并重新申请。重新申请数据块时，出错的数据节点信息会放入请求的 excludedNodes 参数中，防止再次在该节点上分配数据块。

由于这个时候数据节点上并不存在数据块的副本，FSNamesystem.abandonBlock() 方法的实现相当简单，使用 FSDirectory 的成员函数 removeBlock() 在文件和 BlocksMap 中移除数据块信息。代码如下：

```
public synchronized boolean abandonBlock(Block b, String src, String holder)
        throws IOException {
    ……
    if (isInSafeMode()) {
        throw new SafeModeException(……);
    }
    INodeFileUnderConstruction file = checkLease(src, holder);
    dir.removeBlock(src, file, b);
    ……
    return true;
}
```

removeBlock() 方法的作用正好和 FSNamesystem.addBlock() 相反，为了讨论的完整性，下面给出 removeBlock() 的代码，其读者自行分析。

```
public synchronized boolean abandonBlock(Block b, String src, String holder)
        throws IOException {
    ……
    if (isInSafeMode()) {
        throw new SafeModeException(……);
    }
    INodeFileUnderConstruction file = checkLease(src, holder);
    dir.removeBlock(src, file, b);
    ……
    return true;
}
```

### 3. 写文件：关闭文件

一般情况下，客户端通过 create() 创建文件并调用 addBlock()，分配第一个数据块，或通过 append() 打开一个已经存在的文件（也可能需要分配数据块）后，就可以使用数据节点的流式接口，打开数据流管道并写文件数据，在写完一个数据块时，数据节点通过 blockReceived()，提交数据块，而客户端则根据需要，或使用 addBlock() 分配新数据块继续

写文件，或通过 ClientProtocol 的远程方法 complete()，关闭文件。下面分析 ClientProtocol.complete() 的实现。

和大多数 ClientProtocol 中的方法一样，关闭文件主要实现在 FSNamesystem 的 completeFile() 和 completeFileInternal() 中。completeFile() 方法的返回值是一个枚举变量，它们的含义如下：

❑OPERATION_FAILED：操作失败，如文件已经被其他客户端删除。

❑STILL_WAITING：操作成功，但还需要等待数据节点的数据块提交。

❑COMPLETE_SUCCESS：操作成功，文件被成功关闭。

方法返回 STILL_WAITING 时，表明文件还有数据块没有达到系统要求的最小副本数，目前还不能判断文件能否成功关闭，客户端还需继续等待。这里，使用了前面分析过的 checkFileProgress()。如果这个方法返回 true，表明文件可以关闭，completeFileInternal() 使用 finalizeINodeFileUnderConstruction()，执行相关工作。包括：从租约管理器中释放文件租约；将目录树中文件对应的 INodeFileUnderConstruction 对象替换为 INodeFile 对象，并更新数据块映射对象 blocksMap 中的信息；调用 FSDirectory.closeFile() 关闭文件；使用前面分析过的 checkReplicationFactor() 方法，判断文件是否有需要进行复制的数据块，以保证副本数达到文件的副本系数。这些工作完成后，方法返回 COMPLETE_SUCCESS。

这些方法代码如下：

```
enum CompleteFileStatus {
   OPERATION_FAILED, STILL_WAITING, COMPLETE_SUCCESS
}

public CompleteFileStatus completeFile(String src, String holder)
      throws IOException {
   CompleteFileStatus status = completeFileInternal(src, holder);
   getEditLog().logSync(); // 记录日志
   return status;
}

private synchronized CompleteFileStatus completeFileInternal(String src,
      String holder) throws IOException {
   ……
   if (isInSafeMode())// 安全模式检查
      throw new SafeModeException(……);

   // 检查租约
   INodeFileUnderConstruction pendingFile = checkLease(src, holder);
   Block[] fileBlocks = dir.getFileBlocks(src);
   if (fileBlocks == null) {
      ……
      return CompleteFileStatus.OPERATION_FAILED;
   }

   if (!checkFileProgress(pendingFile, true)) {
```

```
            return CompleteFileStatus.STILL_WAITING;
        }
        finalizeINodeFileUnderConstruction(src, pendingFile);
        ......
        return CompleteFileStatus.COMPLETE_SUCCESS;
    }

    private void finalizeINodeFileUnderConstruction(String src,
            INodeFileUnderConstruction pendingFile) throws IOException {
        ......
        leaseManager.removeLease(pendingFile.clientName, src);

        INodeFile newFile = pendingFile.convertToInodeFile();
        dir.replaceNode(src, pendingFile, newFile);
        dir.closeFile(src, newFile);

        checkReplicationFactor(newFile);
    }
```

FSDirectory.closeFile() 会在日志文件中记录文件关闭，和 startFileInternal() 中通过 FSDirectory.addFile() 记录的文件打开日志，正好对应。代码如下：

```
    void closeFile(String path, INodeFile file) throws IOException {
        ......
        synchronized (rootDir) {
            fsImage.getEditLog().logCloseFile(path, file);
            ......
        }
    }
```

#### 4. 租约管理器

租约是 HDFS 实现中的一个重要概念。对于 HDFS 来说，租约是名字节点给予租约持有者，一般是客户端，在规定时间期限内一定权限（写文件）的合同。租约在分布式系统中有着大量的应用，如动态主机设置协议（Dyanamic Host Configuration Protocol，DHCP），该协议用于为用户提供网络参数配置，其最重要的功能就是动态分配 IP 地址。在 DHCP 中，租约就是服务器分配给客户端的网络参数配置使用期限，一定的时间后，服务器可以回收该配置，提供给其他主机使用。当租用时间达到租约期限的 50% 时，DHCP 客户端会尝试进行租约更新，当然，服务器可以确认租约，客户端可以继续使用已有配置，服务器也可以否认续租，甚至取消租约。在 NFS 的版本 4 中，提供了基于租约的锁机制，当 NFS 的客户端希望对文件或文件的一部分进行锁定时，它向网络锁管理器（Network Lock Manager）发起一个请求，管理器会返回给客户端一个租约，如果在租约过期前客户端没有更新，管理器就会释放锁。客户端可以通过租约更新，或者其他的隐含租约更新操作，维护对文件的锁定。

HDFS 中，为写打开的文件（下面简称为打开文件）采取了类似机制，这个机制被实现为租约管理器，图 8-28 是它的类图。

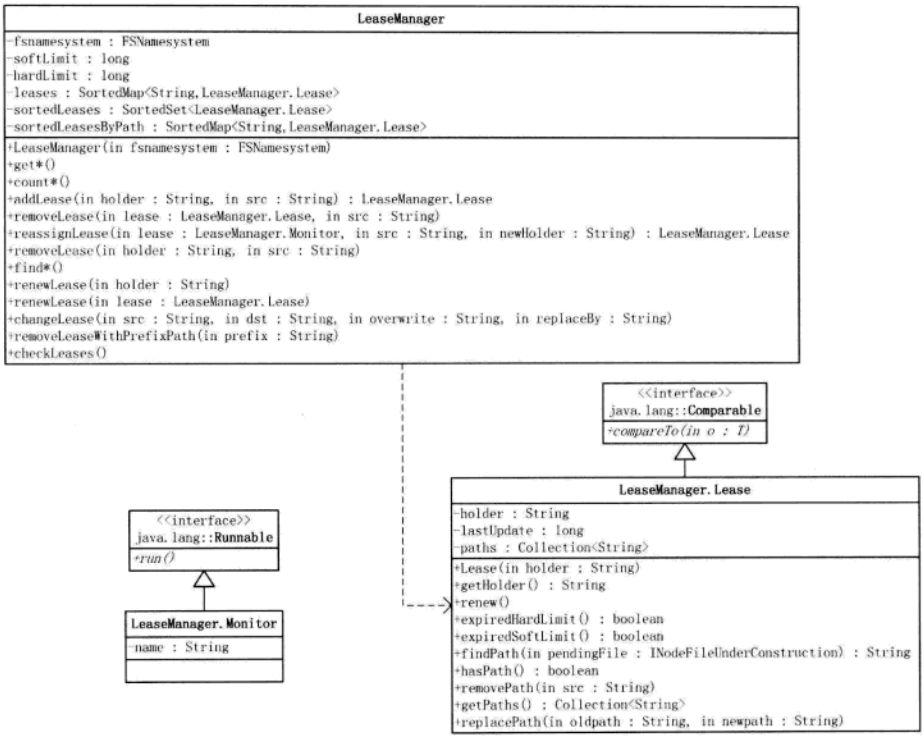



图 8-28　租约管理器的相关类

如同 DHCP 中的网络参数配置（具体为 IP 地址分配）或 NFS 中的锁机制，每一个打开文件在租约管理器中都有一条记录，所以，已经打开的文件不能被其他客户端再次打开；同样，客户端需要通过租约更新维护租约，关闭文件时，必须在租约管理器中释放租约。

由于一个客户端往往会同时写多个文件，我们把一个客户端打开的所有文件组织在一起，构成管理器中的一条记录，并实现为 LeaseManager.Lease 类。客户端的信息（租约持有者）保存在成员变量 holder 中，租约的最后更新时间记录在 lastUpdate 中，成员变量 paths 则保存了该客户端当前打开的所有文件。声明如下：

```
class Lease implements Comparable<Lease> {
   private final String holder;
   private long lastUpdate;
   private final Collection<String> paths = new TreeSet<String>();
   ……
}
```

为了提高部分操作的效率，租约管理器 LeaseManager 保存了租约持有者打开文件到租约对象的映射，代码如下：

```
public class LeaseManager {
```

```
    ……
    private final FSNamesystem fsnamesystem;  // 用于访问 FSNamesystem
    ……
    // 客户端到 Lease 对象的映射
    private SortedMap<String, Lease> leases = new TreeMap<String, Lease>();
    //Lease 对象集合
    private SortedSet<Lease> sortedLeases = new TreeSet<Lease>();
    // 打开文件（路径）到 Lease 对象的映射
    private SortedMap<String, Lease> sortedLeasesByPath
          = new TreeMap<String, Lease>();     ……
}
```

**注意** 租约 Lease 是有序的，比较先按照租约的最后更新时间，该时间如果相等，则比较租约持有者，LeaseManager.sortedLeases 中保存了有序的租约集合。

下面，按照打开写文件中几个重要调用的顺序，考察租约管理器对文件写操作提供的支持。

不管是创建文件还是为追加数据而打开文件，都会调用 FSNamesystem.startFileInternal()，这个方法的最后，通过 LeaseManager.addLease() 在租约管理器中添加打开文件的信息。其中，holder 就是远程接口 ClientProtocol 中方法 create() 或 append() 中的参数 clientName，由客户端提供。代码如下：

```
synchronized Lease addLease(String holder, String src) {
   Lease lease = getLease(holder); // 根据 holder 查找 Lease 对象
   if (lease == null) { // 没找到 Lease 对象，创建一个新的
      lease = new Lease(holder);
      leases.put(holder, lease);
      sortedLeases.add(lease);
   } else {
      renewLease(lease); // 租约更新
   }
   sortedLeasesByPath.put(src, lease);
   lease.paths.add(src); // 将文件加入 Lease 对象中
   return lease;
}
```

为追加数据打开文件，添加、放弃数据块和关闭文件时，都需要对被操作的文件进行租约检查，使用的是 FSNamesystem.checkLease() 方法。

租约检查都会检查哪些内容呢？答案在下面的代码中：

```
private void checkLease(String src, String holder, INode file)
      throws IOException {
   if (file == null || file.isDirectory()) {
      Lease lease = leaseManager.getLease(holder);
      throw new LeaseExpiredException(……);
   }
   if (!file.isUnderConstruction()) {
      Lease lease = leaseManager.getLease(holder);
      throw new LeaseExpiredException(……);
```

```
    }
    INodeFileUnderConstruction pendingFile
          = (INodeFileUnderConstruction) file;
    if (holder != null && !pendingFile.getClientName().equals(holder)) {
       throw new LeaseExpiredException(……}
}
```

checkLease() 在被检查文件为空、或者 INode 是目录，文件没有被打开和虽然文件被打开，但输入的租约持有者并不是实际持有者的情况下，都会抛出 LeaseExpiredException 异常。如果 checkLease() 正常返回，表明当前客户端的确拥有文件的租约，可以进行上述操作。但 checkLease() 中对租约的访问，只是为 LeaseExpiredException 提供一些附加的信息，上述检查，不需要使用到租约管理器中的信息。

文件打开期间，客户端会定时通过 ClientProtocol.renewLease() 维护租约。这个远程方法是 ClientProtocol 中实现最简单的方法：FSNamesystem.renewLease() 在确认名字节点不处于安全模式后，调用 LeaseManager.renewLease()，更新变量 Lease.lastUpdate 的值为当前时间。代码如下：

```
synchronized void renewLease(String holder) throws IOException {
   if (isInSafeMode())
      throw new SafeModeException(……);
   leaseManager.renewLease(holder);
}

synchronized void renewLease(String holder) {
   renewLease(getLease(holder));
}

synchronized void renewLease(Lease lease) {
   if (lease != null) {
      sortedLeases.remove(lease);
      lease.renew(); // 更新 Lease.lastUpdate 为当前时间
      sortedLeases.add(lease); //SortedSet<Lease>，先删除再添加，保持顺序
   }
}
```

关闭文件时，finalizeINodeFileUnderConstruction() 方法释放租约使用的是 removeLease() 方法，它的功能和 addLease() 正好相反，其实现也很简单，就不展开介绍了。

为了让读者对租约管理有更完整的理解，下面介绍一些名字节点第一关系处理中涉及 LeaseManager 的内容。名字节点中涉及租约管理器的方法如图 8-29 所示。

介绍命名空间镜像时，已经给出了方法对 saveFilesUnderConstruction() 和 loadFilesUnderConstruction() 的要点，下面给出的是 FSEditLog.loadFSEdits() 中对 OP_ADD，也就是对创建文件的日志记录进行处理的一个片段：

```
……
switch (opcode) {
case OP_ADD:
```

```
case OP_CLOSE: {
    ......
    //添加文件到目录树
    INodeFile node = (INodeFile) fsDir.unprotectedAddFile(path,
          permissions, blocks, replication, mtime, atime, blockSize);
    if (opcode == OP_ADD) {
       numOpAdd++;
       //将节点替换为 INodeUnderConstruction，并重新创建租约记录
       INodeFileUnderConstruction cons
                = new INodeFileUnderConstruction(……);
          fsDir.replaceNode(path, node, cons);
          fsNamesys.leaseManager.addLease(cons.clientName, path);
       }
       break;
    }
```

图 8-29 使用租约管理器的方法

FSDirectory.addFile() 会记录被创建文件的绝对路径、副本数、修改时间、访问时间、数据块大小、数据块列表、权限（用户名、用户组名和访问权限）、客户端名称、客户端所在的主机等信息。loadFSEdits() 方法会从日志文件中读取上述信息，并根据这些信息，在目录树中创建对应的 INodeFileUnderConstruction 对象；同时，还需要恢复在租约管理器中的记录，使用的是 LeaseManager 的成员函数 addLease()。

文件 / 目录改名或删除时，都可能会影响到租约管理器中的内容。如文件 / 目录改名时，如果所在目录子树上有已经打开的文件，则文件的绝对路径会被修改，这时，需要调用 changeLease() 修改租约管理器中保存的记录；而删除操作也可能影响已打开文件，也就需要删除租约管理器中的记录（使用 removePathAndBlocks() 方法）。这两个方法的实现如下，它们调用的 LeaseManager 的 changeLease() 和 removeLeaseWithPrefixPath() 都很简单，这里就不继续分析了。

```
//FSNamesystem 的 changeLease() 方法，被 FSNamesystem.renameToInternal() 调用
void changeLease(String src, String dst, HdfsFileStatus dinfo)
      throws IOException {
   ......
   leaseManager.changeLease(src, dst, overwrite, replaceBy);
}

void removePathAndBlocks(String src, List<Block> blocks) throws IOException
{
   // 删除以 src 作为路径前缀的
   leaseManager.removeLeaseWithPrefixPath(src);
   ......
}
```

### 5. 写文件：租约恢复

到目前为止，和租约管理相关的内容都平淡无奇，只对租约管理器中保存的内容进行增加、查询、修改和删除。但当客户端出现故障，并且永远地消失了，这个时候，该客户端打开的文件将永远不会被关闭，怎么办？

租约管理器中的信息将用于关闭文件。租约管理器会定期地检查租约，对于长时间没有进行租约更新的文件，名字节点通过租约恢复机制，关闭文件。

租约的定期检查实现在 LeaseManager 的内部类 Monitor 中，Monitor 实现了 Runnable 接口，拥有自己的线程，在该线程中，每隔两秒会调用 LeaseManager.checkLeases() 检查租约。

租约管理器中有两个租约过期时间，分别保存在成员变量 softLimit 和 hardLimit 中，它们被称为软超时时间和硬超时时间。软超时时间为 1 分钟，硬超时时间为 1 小时，不可配置。

checkLeases() 方法中使用的是硬超时时间，也就是说，客户端在一个小时内没有更新租约，才会被 LeaseManager.Monitor 发现并触发租约恢复。代码如下：

```
synchronized void checkLeases() {
   for(; sortedLeases.size() > 0; ) {
      final Lease oldest = sortedLeases.first();
      if (!oldest.expiredHardLimit()) {
         return; //没有发现过期的租约，返回
      }
      final List<String> removing = new ArrayList<String>();
      String[] leasePaths = new String[oldest.getPaths().size()];
      oldest.getPaths().toArray(leasePaths);
      for(String p : leasePaths) {
         try {
            fsnamesystem.internalReleaseLeaseOne(oldest, p); //租约恢复
         } catch (IOException e) {
            ......
            removing.add(p); //失败
         }
      }

      for(String p : removing) {
```

```
            // 调用 internalReleaseLeaseOne() 失败，直接删除
            removeLease(oldest, p);
        }
    }
}
```

租约恢复使用的是 FSNamesystem 的成员函数 internalReleaseLeaseOne()，如果调用这个函数时出现异常，则使用 removeLease() 直接删除租约记录。租约恢复是针对已打开文件，所以，根据路径找不到文件或者文件不处于打开状态时，方法抛异常，让 checkLeases() 直接删除租约。相关代码如下：

```
void internalReleaseLeaseOne(Lease lease, String src) throws IOException {
    ……
    INodeFile iFile = dir.getFileINode(src);
    if (iFile == null) { // 找不到文件，抛异常
        ……
        throw new IOException(message);
    }
    …… // 文件不处于构建状态，抛异常

    INodeFileUnderConstruction pendingFile
            = (INodeFileUnderConstruction) iFile;
    if (pendingFile.getTargets() == null
            || pendingFile.getTargets().length == 0) {
        if (pendingFile.getBlocks().length == 0) {
            // 情况 1：文件没有拥有数据块，直接关闭文件
            finalizeINodeFileUnderConstruction(src, pendingFile);
            ……
            return;
        }
        ……// 情况 2：数据流管道成员为空，选择倒数第二个数据块为最后一个数据块
    }
    // 开始租约恢复
    pendingFile.assignPrimaryDatanode();
    Lease reassignedLease = reassignLease(lease, src,
            HdfsConstants.NN_RECOVERY_LEASEHOLDER, pendingFile);
    leaseManager.renewLease(reassignedLease);
}
```

当文件处于构建状态时，还有两种特殊情况需要处理，一种是文件不拥有任何数据块，也就是客户端刚创建文件，还没来得及写数据就出现了故障，这种情形下可直接关闭文件；另一种情况是客户端通过 abandonBlock()，放弃数据块后故障，这个时候打开文件当前的数据流管道为空，那么，使用倒数第二个数据块的数据节点，作为参与到租约恢复过程的数据节点。

租约恢复由数据节点执行，需要通过名字节点指令 DatanodeProtocol.DNA_RECOVERBLOCK 下发，这个过程实现在 INodeFileUnderConstruction.assignPrimaryDatanode() 中，这个方法会在正常工作的数据流管道成员中选择一个数据节点，作为恢复的主节点，其他节点作为参与节点，并将这些信息添加到主数据节点的描述符中，等待该数据节点的心跳上报。名字节

点触发的租约恢复会修改恢复文件的租约信息，它们的租约持有者统一更改为特殊值“NN_Recovery”。

租约恢复在恢复主数据节点上执行，细节请参考第 7 章的内容，图 8-30 给出了租约恢复过程中的一些关键点。

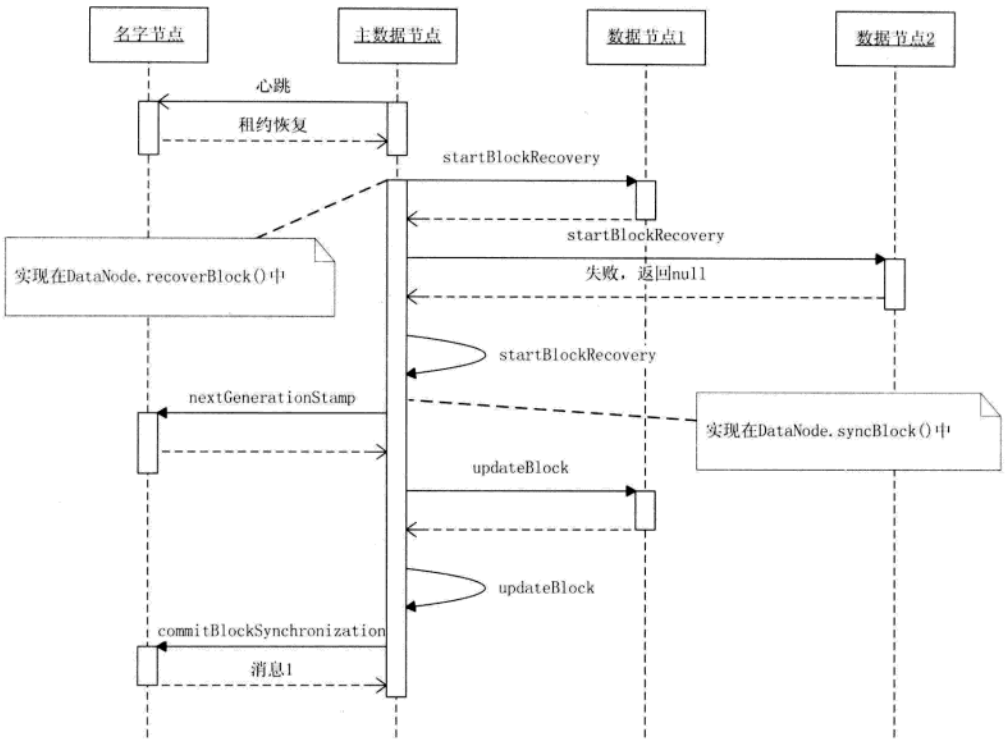



图 8-30　租约恢复过程的关键点

租约恢复的名字节点指令通过心跳返回，主数据节点接收到这个指令后开始租约恢复，它会通过 startBlockRecovery() 方法联系各个参与节点（通过远程方法调用），包括它自己（通过普通方法调用），开始数据块恢复并收集数据块信息，根据收集到的信息，取最小值作为数据块恢复后的长度。接着，主数据节点向名字节点申请新的数据块版本号，并利用 updateBlock() 方法同步各个节点上的数据块，同步结束后，这些数据块拥有相同的长度和新数据块版本号。最后，使用 DatanodeProtocol.commitBlockSynchronization() 向名字节点报告这次租约恢复的结果。

图中，数据节点 2 在执行 startBlockRecovery() 方法后返回空，就不会参加后续的流程，数据节点 2 上的数据块版本号还是旧的版本号。

主数据节点申请新数据块版本号时，使用的是数据节点和名字节点间的 IPC 接口，方法名为 nextGenerationStampForBlock()。分配新版本号其实很简单，名字节点维护了当前已经

使用的最高版本号信息，实现在 GenerationStamp 类中，通过它的成员方法 nextStamp() 就可以获得一个新的数据块版本号。当然，分配新版本号前需要做一系列的检查，如系统是否处于安全模式、申请新版本号的数据块是否存在等，这里就不进一步讲解了。

数据块同步提交方法 commitBlockSynchronization() 的实现比较复杂，毕竟数据块恢复修改了不少数据块相关的信息，如块的大小、版本号等。同时，这个方法的参数也不少，有被恢复数据块信息 lastblock、数据块的新版本号 newgenerationstamp 和新长度 newlength，标志位 closeFile 指示是否关闭对应的文件，另一个标志位 deleteblock 用于指示是否删除数据块，最后一个参数是成功进行了数据块恢复的数据节点列表，也就意味着，图 8-30 的数据节点 2 不会出现在这个列表中。

FSNamesystem.commitBlockSynchronization() 会先在 blocksMap 中移除老数据块信息，为插入新数据块信息做准备。当数据块恢复过程中，主数据节点没有找到一个可恢复的数据块时，则 deleteblock 被设置为 true，commitBlockSynchronization() 在 INodeFileUnderConstruction 对象中删除数据块。如果成功地恢复了部分数据块，则先更新数据块信息，然后将新信息添加到 blocksMap 中。

注意，如果不关闭文件，按照打开文件时的讨论，在数据块提交前，名字节点第二关系中的 DatanodeDescriptor 不会保存数据块信息。所以，标志位 closeFile 为 true 时，即方法调用结束后，文件关闭，需要把数据块信息加到成功进行了数据块恢复的数据节点的 DatanodeDescriptor 中。无论 closeFile 取什么值，上述数据节点的信息都会被收集起来，设置到打开文件的 INodeUnderConstruction 对象中，以更新持有最后一个数据块的数据节点信息。

commitBlockSynchronization() 的代码如下：

```
public void commitBlockSynchronization(Block lastblock,
      long newgenerationstamp, long newlength, boolean closeFile,
      boolean deleteblock, DatanodeID[] newtargets) throws IOException {
   ......
   String src = null;
   synchronized (this) {
      ...... // 系统是否处于安全模式，是，抛异常
      final BlockInfo oldblockinfo = blocksMap.getStoredBlock(lastblock);
      ...... // 数据块是否存在，不存在就抛异常
      INodeFile iFile = oldblockinfo.getINode();
      ...... // 系统是否处于安全模式，是，抛异常
      INodeFileUnderConstruction pendingFile
            = (INodeFileUnderConstruction) iFile;

      // 移除旧数据块信息
      blocksMap.removeBlock(oldblockinfo);

      if (deleteblock) {
         pendingFile.removeBlock(lastblock);
      } else {
         // 更新数据块信息
         lastblock.set(lastblock.getBlockId(),
```

```
                newlength, newgenerationstamp);
            final BlockInfo newblockinfo
                = blocksMap.addINode(lastBlock, pendingFile);

            DatanodeDescriptor[] descriptors = null;
            List<DatanodeDescriptor> descriptorsList
                = new ArrayList<DatanodeDescriptor>(newtargets.length);
            for (int i = 0; i < newtargets.length; i++) {
               DatanodeDescriptor node
                   = datanodeMap.get(newtargets[i].getStorageID());
               if (node != null) {
                  if (closeFile) {
                     // 关闭文件，将数据块信息加入各数据节点的描述符中
                     node.addBlock(newblockinfo);
                  }
                  descriptorsList.add(node);
               }
            }
            if (!descriptorsList.isEmpty()) {
               descriptors
                   = descriptorsList.toArray(new DatanodeDescriptor[0]);
            }
            // 添加数据节点信息到 INodeUnderConstruction 对象中
            pendingFile.setLastBlock(newblockinfo, descriptors);
         }

         src = leaseManager.findPath(pendingFile);
         if (!closeFile) {// 不关闭文件
            if (supportAppends) {
               dir.persistBlocks(src, pendingFile);
               getEditLog().logSync();
            }
            ......
            return;
         }
         // 移除租约并关闭文件
         finalizeINodeFileUnderConstruction(src, pendingFile);
      }

      getEditLog().logSync();
      ......
   }
```

commitBlockSynchronization() 方法的最后一部分和租约相关，如果 closeFile 为 false，即不关闭文件，同时，系统支持追加操作，方法需要持久化被修改文件的数据块信息，同步日志并返回（可以肯定的是，这不是一个名字节点发起的数据块恢复过程）；否则，移除租约并关闭文件。

至此，租约恢复结束。

名字节点中，除了 LeaseManager 的内部类 Monitor 可以启动租约恢复以外，远程接口

ClientProtocol 的 recoverLease() 方法也可用于发起租约恢复，该远程方法的实现最终调用了 FSNamesystem.recoverLeaseInternal()。

打开文件的 startFileInternal() 也使用这个方法，它们调用 recoverLeaseInternal() 的差别在于参数 force 的值，如果是客户端发起的租约恢复，该标志位为 true；startFileInternal() 则使用 false 调用这个方法。

客户端发起的租约恢复，在确认文件处于打开状态后，以原文件拥有者的名义调用 internalReleaseLeaseOne()，进行租约恢复。代码如下：

```
private void recoverLeaseInternal(INode fileInode, String src,
      String holder, String clientMachine, boolean force)
      throws IOException {
   if (fileInode != null && fileInode.isUnderConstruction()) {
      // 获取 INodeFileUnderConstruction 对象和租约
      INodeFileUnderConstruction pendingFile
         = (INodeFileUnderConstruction) fileInode;
      Lease lease = leaseManager.getLease(holder);

      if (!force && lease != null) { //startFileInternal() 调用
         Lease leaseFile = leaseManager.getLeaseByPath(src);
         if (leaseFile != null && leaseFile.equals(lease)) {
            throw new AlreadyBeingCreatedException(……);
         }
      }
      // 获取打开文件的拥有者的租约
      lease = leaseManager.getLease(pendingFile.clientName);
      if (lease == null) {
         throw new AlreadyBeingCreatedException(……);
      }
      if (force) { // 客户端调用
         ……
         internalReleaseLeaseOne(lease, src);
      } else {
         if (lease.expiredSoftLimit()) {
            ……
            internalReleaseLease(lease, src);
         }
         throw new AlreadyBeingCreatedException(……);
      }
   }
}

void internalReleaseLease(Lease lease, String src) throws IOException {
   if (lease.hasPath()) { // 该租约拥有多个文件
      String[] leasePaths = new String[lease.getPaths().size()];
      lease.getPaths().toArray(leasePaths);
      for (String p : leasePaths) {
         internalReleaseLeaseOne(lease, p);
      }
```

```
    } else { //租约没有记录文件的信息，恢复这个文件
        internalReleaseLeaseOne(lease, src);
    }
}
```

如果是 startFileInternal() 调用了 recoverLeaseInternal() 方法，在大多数情况下，目录树上对应的 INode 节点不处于构造状态，方法返回。如果被打开文件处于构造状态，即文件已经打开，这时，执行路径会抛异常 AlreadyBeingCreatedException，方法只不过在确认原文件拥有者的租约已经软超时，即 lease.expiredSoftLimit() 返回 true 的情况下，“顺便”通过 internalReleaseLease() 对该租约进行租约恢复。也就是说，startFileInternal() 调用 recoverLeaseInternal() 的主要目的，在于判断文件是否已经被其他客户端打开，防止同时有多个客户端写同一个文件。

**6. 写文件：其他**

这一小节分析 ClientProtocol.fsync() 和 DatanodeProtocol.blocksBeingWrittenReport() 的实现。

在 Linux 文件系统中，大多数磁盘 I/O 都会使用内核缓冲区，当将数据写到文件上时，通常这些数据先由内核复制到缓冲区中，如果缓冲区尚未写满，则不会进行实际的 I/O 操作，而是等待其写满或者当内核需要重用缓冲区存放其他数据时，才会将数据写入磁盘。为了保证磁盘上实际文件系统与缓冲区中内容的一致性，Linux 系统提供了 sync() 和 fsync() 两个系统调用。函数 sync() 将缓冲区刷出，然后就返回，fsync() 则会等待，直到实际的 I/O 操作结束，这时，文件的内容和元数据都已经持久化到磁盘上。

HDFS 的 ClientProtocol 接口也提供了 fsync() 方法，它的实现如下：

```
void fsync(String src, String clientName) throws IOException {
    ……
    synchronized (this) {
        if (isInSafeMode()) {// 安全模式
            throw new SafeModeException(……);
        }
        INodeFileUnderConstruction pendingFile
                = checkLease(src, clientName);
        dir.persistBlocks(src, pendingFile); // 持久化文件的数据块信息
    }
    getEditLog().logSync();
}
```

作为一个分布式文件系统，HDFS 和 Linux 文件系统有着较大的差别，它们的缓存刷新功能和实现也不一样。但有一点是肯定的，就是刷新缓存时，既需要刷新缓存到数据节点，也需要将元数据及时更新到名字节点上的磁盘中，fsync() 的作用就是持久化名字节点上的文件元数据。它的实现很简单，分析就先略过。

DatanodeProtocol.blocksBeingWrittenReport() 也是一个和写文件相关的远程方法，当 HDFS 打开支持 Append 操作开关时，数据节点注册后，会报告节点上处于客户端写状态的

数据块信息，也就是上次数据节点退出时，写操作还没结束，没提交的数据块。这些数据块对应的数据文件，保存在“${dfs.data.dir}/blocksBeingWritten”目录下。

**注意** 数据节点注册后，数据块复制的未提交数据块不会上报（它们保存在${dfs.data.dir}/tmp 目录下），而是简单删除。

名字节点上该远程方法的实现，主体在processBlocksBeingWrittenReport()中。这个方法的大部分逻辑都是在处理各种异常情况，如数据节点不存在、文件已经被关闭等。通过一系列检查后，该数据节点会作为对应文件的数据流管道成员，添加到INodeFileUnderConstruction对象中，后续该数据块可能参与到租约恢复过程，或者被删除。代码如下：

```
public synchronized void processBlocksBeingWrittenReport(
      DatanodeID nodeID,BlockListAsLongs blocksBeingWritten)
      throws IOException {
   DatanodeDescriptor dataNode = getDatanode(nodeID);
   if (dataNode == null) {
      throw new IOException(……); // 数据节点不存在
   }
   if (shouldNodeShutdown(dataNode)) {
      // 典型的DatanodeProtocol远程方法检查
      setDatanodeDead(dataNode);
      throw new DisallowedDatanodeException(dataNode);
   }

   Block block = new Block();
   // 循环处理
   for (int i = 0; i < blocksBeingWritten.getNumberOfBlocks(); i++) {
      block.set(blocksBeingWritten.getBlockId(i),
            blocksBeingWritten.getBlockLen(i),
            blocksBeingWritten.getBlockGenStamp(i));
      BlockInfo storedBlock
            = blocksMap.getStoredBlockWithoutMatchingGS(block);

      if (storedBlock == null) { // 数据块不存在
         // 检查不通过时，调用rejectAddStoredBlock()并处理下一个数据块，下同
         rejectAddStoredBlock(new Block(block), dataNode, "……");
         continue;
      }
      INodeFile inode = storedBlock.getINode();
      if (inode == null) …… // 文件不存在
      boolean underConstruction = inode.isUnderConstruction();
      boolean isLastBlock = inode.getLastBlock() != null
            && inode.getLastBlock().getBlockId() == block.getBlockId();
      if (!underConstruction) …… // 文件已经被关闭
      if (!isLastBlock) …… // 不是最后一个数据块
```

```
            INodeFileUnderConstruction pendingFile
                  = (INodeFileUnderConstruction) inode;
            pendingFile.addTarget(dataNode);
            incrementSafeBlockCount(pendingFile.getTargets().length);
        }
    }
```

### 8.3.4　工具 dfsadmin 依赖的方法

ClientProtocol 接口也为实现多功能工具 dfsadmin 提供了一系列的远程方法，这些方法的实现，和 8.3.1 节中介绍的文件和目录相关事务远程方法类似，为了讨论的完整性，表 8-3 列出了 ClientProtocol 中的这些方法。有些方法已经在 8.3.1 节中出现过，它属于文件客户端和多功能工具共同使用的远程方法，也一并给出。

表 8-3　ClientProtocol 中的方法

| 使用的远程方法 | dfsadmin 命令 | 方法实现要点 |
|---|---|---|
| getStats、setSafeMode、distributedUpgradeProgress、getDatanodeReport | report | 这里只介绍 getDatanodeReport()，FSNamesystem.datanodeReport() 根据条件，在 datanodeMap 中选择符合条件的 DatanodeInfo 数组，返回给调用者使用 |
| setSafeMode | safemode | 方法 setSafeMode() 通过枚举参数 action，可以查询 HDFS 是否处于安全模式，让 HDFS 进入、离开安全模式。后续有更详细的讨论 |
| saveNamespace | saveNamespace | 保存当前名字节点元数据的命名空间镜像。需要超级用户权限，同时，文件系统还需要处于安全模式，镜像通过 FSImage.saveNamespace() 输出 |
| refreshNodes | refreshNodes | 实现在 FSNamesystem.refreshNodes() 中，已经在 8.2.1 节中介绍 |
| finalizeUpgrade | finalizeUpgrade | 升级相关（不讨论） |
| distributedUpgradeProgress | upgradeProgress | 升级相关（不讨论） |
| metaSave | metasave | 保存名字节点当前活动情况。实现在 FSNamesystem.metaSave() 中，方法将 datanodeMap、neededReplications、pendingReplications 等变量保存的信息输出到指定文件 |
| setQuota | setQuota、clrQuota、setSpaceQuota 和 clrSpaceQuota | 8.3.1 节中已经介绍 |
| setBalancerBandwidth | setBalancerBandwidth | 由 FSNamesystem 的成员函数 setBalancerBandwidth() 执行，遍历 datanodeMap 中保存的数据节点并调用 DatanodeDescriptor 的同名方法，设置新的均衡器工作带宽 |

工具 dfsadmin 的 refreshServiceAcl、refreshUserToGroupsMappings 和 refreshSuperUserGroupsConfiguration 命令不使用 ClientProtocol 远程接口和名字节点通信，而是依赖于 RefreshAuthorizationPolicyProtocol、RefreshUserMappingsProtocol 等接口，它们用于实现 HDFS 的安全特性，就不再讨论了。

## 8.4 名字节点的启动和停止

现在，可以启动名字节点了。

### 8.4.1 安全模式

名字节点启动时，内存中一片空白，第一要紧的事情就是把命名空间镜像加载到内存，并应用镜像编辑日志，这个过程结束时，会创建一个新的检查点，包括新命名空间镜像和一个空的编辑日志。完成上述工作后，名字节点开始监听 IPC 和 HTTP 请求，虽然这个时候名字节点已经对外提供服务，但它只为客户端提供一个只读视图，这种名字节点的只读模式称为安全模式。

继续上面的过程，在名字节点第一关系建立起来以后，需要处理数据节点启动时的若干远程调用，以建立名字节点第二关系。在安全模式下，需要给名字节点一些时间，使名字节点获得足够多的数据节点，数据块也能获得足够多的保存数据块的数据节点的信息，以高效地运行文件系统。如果名字节点没有等到足够多的数据节点，大量的数据块会出现副本数不足的情况，进入复制队列并最终产生复制请求。这在很多情况下显然是没有必要的，而且还会浪费系统资源。从前面的代码分析，我们知道，在安全模式下，名字节点不会向数据节点下发数据块复制或数据块删除指令。

当收集到足够多的第二关系信息后，名字节点会离开安全模式，进入正常工作状态，允许客户端和系统自身对文件系统进行修改。足够多的信息指的是文件系统中一定比例的数据块副本信息到达最低副本水平，默认情况下，需要系统 95% 的数据块在第二关系中有一个副本的位置信息。名字节点离开安全模式前，还可以通过配置安全模式等待时间，在满足上述离开安全模式的副本条件后，让名字节点继续等待一段时间，等待时间默认值为 0，也就是立即离开安全模式。表 8-4 给出了安全模式中的一些配置项。

**表 8-4 安全模式中的配置项**

| 配置名 | 类型 | 默认值 | SafeModeInfo 中对应变量 | 描 述 |
|---|---|---|---|---|
| dfs.replication.min | int | 1 | safeReplication | 最低副本水平数/系统最小副本数，也用于写操作中，如 completeFileInternal() 方法中 |
| dfs.safemode.threshold.pct | float | 0.95 | threshold | 离开安全模式时系统满足最低副本水平的副本比例。后面讨论中，把这个条件称为最小副本水平条件 |
| dfs.safemode.extension | int | 0 | extension | 安全模式等待时间，在满足最小副本水平条件 ${fs.safemode.threshold.pct} 后，离开安全模式的时间，单位是毫秒 |

工具 dfsadmin 中的 safemode 选项，可以查看名字节点是否处于安全模式。

```
%Hadoop dfsadmin -safemode get
```

```
Saft mode is ON
```

运维人员也可以从管理页面了解 HDFS 当前是否处于安全模式。

下面的命令可用于等待名称节点离开安全模式：

```
%Hadoop dfsadmin -safemode wait
```

有些 HDFS 运维命令，如保存当前命名空间镜像的 saveNamespace 命令需要名字节点处于安全模式，所以有必要提供进入、退出安全模式的管理命令。要让 HDFS 系统进入安全模式，使用以下命令即可：

```
%Hadoop dfsadmin -safemode enter
Saft mode is ON
```

使用上面命令时，名字节点可处于任何状态，当名字节点处于启动时的安全模式时，这个命令也可以生效，即使收集到足够多的副本位置信息，名字节点也不会离开安全模式。而下面的命令可以让名字节点离开安全模式。

```
%Hadoop dfsadmin -safemode leave
Saft mode is OFF
```

HDFS 安全模式的需求总结如下：

❑处于安全模式下，HDFS 只提供系统的只读视图，不能进行修改。

❑名字节点启动时，根据配置，检查第二关系中数据块的副本信息，满足条件时离开安全模式。

❑通过命令行支持安全模式状态查询、等待和设置。

上述需求的第一项，已经实现在名字节点的代码中，大量对文件系统进行修改的操作，都必须检查系统当前是否处于安全模式。后面两项是本小节讨论的重点，相关的代码大部分实现在 FSNamesystem 中，如图 8-31 所示。其中，SafeModeInfo 保存了安全模式的配置信息和工作时需要的状态信息，安全模式检查器 SafeModeMonitor 继承自 Runnable 接口，用于实现安全模式等待时间。

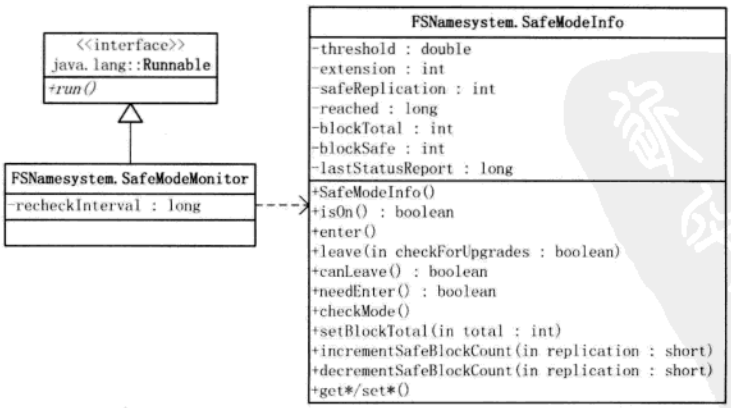



图 8-31　安全模式相关类的类图

SafeModeInfo 的成员变量不多，除了上述配置参数外，blockSafe 和 blockTotal 用于计算离开安全模式的副本比例，名字节点第一关系加载后，系统也就知道 blockTotal 的值，安全模式副本数 blockSafe 的值则需要动态构建，往名字节点第二关系中添加数据块到数据节点对应关系的 addStoredBlock() 和 processBlocksBeingWrittenReport() 方法，会增加该计数器；移除数据块时，即在 removeStoredBlock() 方法中，会减去相应的值，对这两个计数器进行修改的方法如下：

```
synchronized void setBlockTotal(int total) {
   this.blockTotal = total;
   checkMode(); //后面介绍
}

synchronized void incrementSafeBlockCount(short replication) {
   if ((int) replication == safeReplication) //达到最低副本水平数
      this.blockSafe++;
   checkMode();
}

synchronized void decrementSafeBlockCount(short replication) {
   if (replication == safeReplication - 1)
      this.blockSafe--;
   checkMode();
}
```

**注意** 在安全模式下，名字节点也会对数据节点进行心跳检查，长时间没有心跳的数据节点会被移除。SafeModeInfo.reached 保存了安全模式的状态，lastStatusReport 则用于安全模式检查器中。

上面这些更新安全模式计数器的方法，都会调用 checkMode() 检查安全模式的状态。这个方法会先利用 SafeModeInfo.needEnter() 判断当前系统是否满足最小副本水平条件，如果不满足，设置 reached 的值为 0 并返回。满足最小副本水平条件后，继续判断是否需要进行等待，如果不需要等待，调用 SafeModeInfo.leave() 离开安全模式；否则，在第一次满足最小副本水平条件时，checkMode() 就会修改 reached 的值为当前时间，并启动 SafeModeMonitor 对象和它的运行线程，由该线程继续检查安全模式状态。代码如下：

```
private void checkMode() {
   if (needEnter()) {
      enter();  //this.reached = 0;
      ......
      return;
   }

   // 已经满足最小副本水平条件
   if (!isOn() || // 安全模式是关闭的
        extension <= 0 || threshold <= 0) { // 不需要等待，立即离开安全模式
```

```
            this.leave(true); // 离开安全模式
            return;
        }
        if (reached > 0) { // 不是第一次满足最小副本水平条件
            ……
            return;
        }
        // 创建 SafeModeMonitor 对象和它的运行线程
        reached = now();
        smmthread = new Daemon(new SafeModeMonitor());
        smmthread.start();
        ……
    }

    boolean needEnter() {
        return getSafeBlockRatio() < threshold;
    }

    private float getSafeBlockRatio() {
        return (blockTotal == 0 ? 1 : (float) blockSafe / blockTotal);
    }
```

SafeModeMonitor.run() 循环通过 SafeModeInfo.canLeave() 方法，判断是否能离开安全模式。代码如下：

```
    class SafeModeMonitor implements Runnable {
        private static final long recheckInterval = 1000; // 检查间隔时间

        public void run() {
            while (fsRunning && (safeMode != null && !safeMode.canLeave())) {
                try {
                    Thread.sleep(recheckInterval); // 等待 1 秒
                } catch (InterruptedException ie) {
                }
            }
            try {
                leaveSafeMode(true); // 离开安全模式
            } catch (SafeModeException es) {
                ……
                throw new RuntimeException(msg, es);
            }
            smmthread = null; // 释放 SafeModeMonitor 所在的线程
        }
    } //SafeModeMonitor 所在的线程，即 smmthread 原来指向的线程会结束执行
```

canLeave() 方法的实现不算复杂，但必须明确成员变量 reached 的各类取值：当 reached 取默认值 –1 时，SafeModeInfo 对象处于非安全模式状态；通过 SafeModeInfo.enter() 修改该值为 0，进入安全模式状态；当系统第一次满足最小副本水平条件时，它会被修改为系统时间，并用于实现安全模式等待时间。了解了变量 reached 的各种取值，canLeave() 的实现也就显而易见。代码如下：

```
synchronized boolean canLeave() {
   if (reached == 0) // 在安全模式中
      return false;
   if (now() - reached < extension) { // 执行安全模式等待时间
      reportStatus("STATE* Safe mode ON.", false);
      return false;
   }
   return !needEnter(); // 再次判断
}
```

当 canLeave() 方法返回 true 时，名字节点就可以离开安全模式了，SafeModeMonitor.run() 会做一些清理工作，包括调用 SafeModeInfo.leave()、设置变量 FSNamesystem.smmthread 为 null，然后这个由 checkMode() 创建的，SafeModeMonitor.run() 所在的线程结束执行。

无论是在 SafeModeMonitor.run() 或者是 SafeModeInfo.checkMode() 中离开安全模式，都会调用 SafeModeInfo.leave()，安全模式下，名字节点不进行数据块复制或数据块删除的工作，离开安全模式时，必须恢复上述功能，另外，leave() 还会执行一些清理工作。代码如下：

```
synchronized void leave(boolean checkForUpgrades) {
   if (checkForUpgrades) {
      ......
   }
   // verify blocks replications
   ......
   processMisReplicatedBlocks(); // 检查数据块副本，并启动可能的复制和删除
   ......
   reached = -1; // 清理工作之一，设置 reached 的值
   safeMode = null; // 清理工作之二，释放安全模式工作需要的 SafeModeInfo 对象
   ......
}
```

安全模式下，用于复制数据块和删除数据块的 ReplicationMonitor 是工作的，但它会抛弃所有的复制或删除请求。FSNamesystem.processMisReplicatedBlocks() 是一个属于名字节点第二关系的方法，它只被 SafeModeInfo.leave() 调用，用于在名字节点离开安全模式时，对 blocksMap 中的副本信息进行分析，并重新生成数据块复制或删除请求。我们已经研究过很多名字节点第二关系方法，在此就不给出 processMisReplicatedBlocks() 的代码了。

安全模式的另一个需求是通过命令行，支持安全模式状态查询、等待和设置。为此，名字节点提供了远程方法 ClientProtocol.setSafeMode()，它需要一个类型为 SafeModeAction 的参数，表明请求的类型，枚举值包括 SAFEMODE_GET（获取）、SAFEMODE_LEAVE（离开）和 SAFEMODE_ENTER（进入）。除了查询，其他两个操作要求客户端有管理员权限。下面以进入安全模式为例，分析 setSafeMode() 的实现。

不管名字节点处于什么状态，通过 SAFEMODE_ENTER 调用 setSafeMode() 后，系统会进入安全模式，直到使用 SAFEMODE_LEAVE 再次调用 setSafeMode() 才离开。如何“骗过”SafeModeMonitor.run() 的检查呢？只需要简单地将安全模式等待时间设置为 Integer.MAX_VALUE，就可让 SafeModeInfo.canLeave() 方法即使在满足最小副本水平条件的情况

下，也一直返回 false。代码如下：

```
boolean setSafeMode(SafeModeAction action) throws IOException {
   if (action != SafeModeAction.SAFEMODE_GET) {
      checkSuperuserPrivilege();
      switch (action) {
      case SAFEMODE_LEAVE:// 离开安全模式
         leaveSafeMode(false);
         break;
      case SAFEMODE_ENTER: // 进入安全模式
         enterSafeMode();
         break;
      }
   }
   return isInSafeMode();
}

synchronized void enterSafeMode() throws IOException {
   if (!isInSafeMode()) {
      safeMode = new SafeModeInfo();
      return;
   }
   safeMode.setManual();
   getEditLog().logSync();
   ……
}

void setManual() { //SafeModeInfo 方法
   extension = Integer.MAX_VALUE;
}
```

安全模式提供了名字节点的只读视图，是名字节点的一种特殊工作状态，安全模式的需求简单但多变，通过 SafeModeInfo.reached，将涉及的状态集中到一个变量进行管理，简化了安全模式的实现。

## 8.4.2　名字节点的启动

NameNode.main() 是名字节点启动时的入口，和数据节点一样，这个 main() 方法通过类的静态方法 createNameNode() 创建一个 NameNode 对象，顺利创建对象后通过 NameNode.join()，等待对象执行结束。代码如下：

```
public static void main(String argv[]) throws Exception {
   try {
      StringUtils.startupShutdownMessage(NameNode.class, argv, LOG);
      NameNode namenode = createNameNode(argv, null);
      if (namenode != null)
         namenode.join();
   } catch (Throwable e) {
      …… // 处理异常情况
```

```
    }

    public void join() {
        try {
            this.server.join(); //等待IPC服务器结束
        } catch (InterruptedException ie) {
        }
    }

    public static NameNode createNameNode(String argv[], Configuration conf)
            throws IOException {
        if (conf == null)
            conf = new Configuration();
        StartupOption startOpt = parseArguments(argv);
        …… //处理startOpt为空的情况，由printUsage()输出使用说明并返回null
        setStartupOption(conf, startOpt);

        switch (startOpt) {
        case FORMAT:
            boolean aborted = format(conf, true);  //格式化
            System.exit(aborted ? 1 : 0);  //退出
        case FINALIZE: //升级提交
            ……
        default:
        }
        ……
        NameNode namenode = new NameNode(conf);
        return namenode;
    }
```

createNameNode() 方法会处理名字节点的启动选项，这些选项包括：

❑FORMAT：格式化，和数据节点不同，启动 HDFS 集群前需要对名字节点进行格式化，以建立或重新建立名字节点文件结构。

❑REGULAR：正常启动。

❑UPGRADE/ ROLLBACK/ FINALIZE：升级、升级回滚或升级提交（不再介绍）。

❑IMPORT：恢复名字节点上的元数据到某一个检查点（不再介绍）。

格式化名字节点使用的是 NameNode 的静态方法 format()。格式化是名字节点独有的过程，下面给出它的一些细节。格式化时，如果目录存在，会有提示，它就实现在方法 format() 中。代码如下：

```
    private static boolean format(Configuration conf,
            boolean isConfirmationNeeded) throws IOException {
        Collection<File> dirsToFormat = …… //命名空间镜像fsimage保存目录
        Collection<File> editDirsToFormat = …… //编辑日志edits保存目录
        for (Iterator<File> it = dirsToFormat.iterator(); it.hasNext();) {
            File curDir = it.next();
            if (!curDir.exists())continue; //目录为空
            if (isConfirmationNeeded) { //目录非空
```

```
            System.err.print("Re-format filesystem in "
                  + curDir + " ? (Y or N) "); //输出提示
            if (!(System.in.read() == 'Y')) { //读入应答
               System.err.println("Format aborted in " + curDir);
               return true;
            }
            while (System.in.read() != '\n');
         }
      }

      FSNamesystem nsys = new FSNamesystem(
            new FSImage(dirsToFormat,editDirsToFormat), conf);
      nsys.dir.fsImage.format();
      return false;
   }
```

实际的格式化操作，如分配新的存储系统标识、删除原目录（如果目录存在）、创建新目录并建立子目录结构、保存“VERSION”文件等，实现在 FSImage.format() 和 FSImage.format(StorageDirectory) 中，这两个方法的逻辑都很直接，就不给出代码了。

除了格式化名字节点和升级提交外，其他的选项都必须构造名字节点对象。构造函数 NameNode() 和 NameNode.main() 一样，将主要的工作委托给另一个方法，initialize() 用于初始化一个 NameNode 对象。

NameNode.initialize() 不算复杂，在成功创建 FSNamesystem 对象后，它会创建名字节点的 IPC 服务器和 HTTP 服务器，并启动 IPC 服务器和 HDFS 的垃圾桶功能。

当 getServiceRpcServerAddress() 的返回值不为空，即配置了 ${dfs.namenode.servicerpc-address} 参数时，NameNode.initialize() 会创建 IPC 服务器 serviceRpcServer。而 IPC 服务器 server 则使用 ${fs.default.name} 中的地址 / 端口创建。这两个服务器的区别在于，当 serviceRpcServer 存在时，接口 DatanodeProtocol 和 NamenodeProtocol 上的远程调用必须使用 serviceRpcServer 服务器，而提供给 HDFS 客户端的 IPC，则使用服务器 server。客户端的 IPC 调用执行时间短，但往往需要等待日志的持久化；而数据节点和第二名字节点调用的远程过程，执行的逻辑比较复杂，执行的时间也比较长。在一个繁忙的系统中，对客户端和 HDFS 其他实体（包括数据节点和第二名字节点）的 IPC 请求，使用不同的 IPC 服务器，可以保证客户端的请求得到及时的响应。当然，如果不启动 serviceRpcServer 服务器，所有的 IPC 请求都会由服务器 server 处理。代码如下：

```
private void initialize(Configuration conf) throws IOException {
   InetSocketAddress socAddr = NameNode.getAddress(conf);
   …… //安全相关的初始化
   int handlerCount = conf.getInt("dfs.namenode.handler.count", 10);
   …… //安全和度量信息相关的初始化
   this.namesystem = new FSNamesystem(this, conf);
   …… //安全相关的初始化

   …… //创建 IPC 服务器
   InetSocketAddress dnSocketAddr = getServiceRpcServerAddress(conf);
```

```
    if (dnSocketAddr != null) {
        int serviceHandlerCount = conf.getInt(
              DFSConfigKeys.DFS_NAMENODE_SERVICE_HANDLER_COUNT_KEY,
              DFSConfigKeys.DFS_NAMENODE_SERVICE_HANDLER_COUNT_DEFAULT);
        this.serviceRpcServer = RPC.getServer(this,
              dnSocketAddr.getHostName(), dnSocketAddr.getPort(),
              serviceHandlerCount, false, conf,……);
        this.serviceRPCAddress
              = this.serviceRpcServer.getListenerAddress();
        setRpcServiceServerAddress(conf);
    }
    this.server = RPC.getServer(this, socAddr.getHostName(),……);

    startHttpServer(conf);
    this.server.start(); //启动 IPC 服务器
    if (serviceRpcServer != null) {
        serviceRpcServer.start();
    }
    startTrashEmptier(conf); //启动垃圾桶功能
}
```

NameNode 类的实现中，将大量的具体工作交由 FSNamesystem 完成，初始化也是一样的。创建名字节点的 FSNamesystem 对象时，会完成如下工作：

1）通过配置项为 FSNamesystem 的成员变量赋值。

2）创建 FSDirectory 对象，即名字节点第一关系的门面。

3）通过 FSDirectory.loadFSImage()，加载命名空间镜像并应用编辑日志。

4）创建安全模式管理对象 SafeModeInfo()，并调用 setBlockTotal() 更新 SafeModeInfo 的成员变量 blockTotal，调用结束后，名字节点一般应处于安全模式。

5）创建并启动心跳检查线程、租约检查线程、数据块复制 / 删除线程。

6）读入数据节点的 include 文件和 exclude 文件，创建并启动数据节点撤销检查线程。

7）创建网络布局感知需要的 dnsToSwitchMapping 对象，如果可能，分析 include 文件中包含的主机位置。

8）读入配置项 ${fs.default.name}。

FSNamesystem.initialize() 让我们回顾了名字节点的主要组成部分，如名字节点第一关系的 FSDirectory、命名空间镜像和编辑日志，名字节点第二关系的数据节点管理和数据块副本复制、删除，安全模式等。构造完 FSNamesystem 对象后，NameNode.initialize() 打开 IPC 服务器，名字节点就开始对外提供服务了。代码如下：

```
FSNamesystem(NameNode nn, Configuration conf) throws IOException {
    try {
        initialize(nn, conf);
    } catch (IOException e) {
        ……
        close();
        throw e;
    }
```

```
}

private void initialize(NameNode nn, Configuration conf) throws IOException
{
    this.systemStart = now();
    setConfigurationParameters    (conf); //通过配置项为成员变量赋值
    ......
    this.nameNodeAddress = nn.getNameNodeAddress();
    ......
    this.dir = new FSDirectory(this, conf);
    StartupOption startOpt = NameNode.getStartupOption(conf);
    this.dir.loadFSImage(getNamespaceDirs(conf),
          getNamespaceEditsDirs(conf), startOpt);
    ......
    this.safeMode = new SafeModeInfo(conf);
    setBlockTotal();
    pendingReplications = new PendingReplicationBlocks(
          conf.getInt("dfs.replication.pending.timeout.sec", -1) * 1000L);
    ......
    this.hbthread = new Daemon(new HeartbeatMonitor());
    this.lmthread = new Daemon(leaseManager.new Monitor());
    this.replmon = new ReplicationMonitor();
    this.replthread = new Daemon(replmon);
    hbthread.start();
    lmthread.start();
    replthread.start();

    this.hostsReader = new HostsFileReader(conf.get("dfs.hosts", ""),
          conf.get("dfs.hosts.exclude", ""));
    this.dnthread = new Daemon(new DecommissionManager(this).new Monitor(
          conf.getInt("dfs.namenode.decommission.interval", 30),
          conf.getInt("dfs.namenode.decommission.nodes.per.interval",
                5)));
    dnthread.start();

    this.dnsToSwitchMapping = ReflectionUtils.newInstance(
          conf.getClass("topology.node.switch.mapping.impl",
                ScriptBasedMapping.class, DNSToSwitchMapping.class),
          conf);

    if (dnsToSwitchMapping instanceof CachedDNSToSwitchMapping) {
          dnsToSwitchMapping.resolve(
                new ArrayList<String>(hostsReader.getHosts()));
    }

    InetSocketAddress socAddr = NameNode.getAddress(conf);
    this.nameNodeHostName = socAddr.getHostName();

    registerWith(DefaultMetricsSystem.INSTANCE);
}
```

FSNamesystem 和 NameNode 的构造函数调用各自的 initialize() 方法失败时，都有相应的清理函数，释放已经初始化的资源，具体请参考 FSNamesystem.close() 和 NameNode.

stop() 的实现。

### 8.4.3 名字节点的停止

相对于启动，停止名字节点不需要做太多的工作：名字节点第一关系在执行的过程中，会不断通过日志将更新持久化，而第二关系则可通过数据节点上保存的数据块信息恢复。所以，名字节点停止时，只在日志中输出下面的信息就可以退出：

```
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at example.com/192.168.7.78
************************************************************/
```

执行打印的逻辑在启动时，由 StringUtils.startupShutdownMessage() 方法，通过 Runtime.getRuntime().addShutdownHook() 添加到关闭挂钩中，由 Java 虚拟机在退出时执行。

## 8.5 小结

名字节点维护着 Hadoop 分布式文件系统中最重要的两个关系：

❑HDFS 文件系统的文件目录树，以及文件的数据块索引，即每个文件对应的数据块列表。

❑数据块和数据节点的对应关系，即某一数据块保存在那些数据节点的信息。

在传统文件系统中，上述信息属于文件的元信息，和文件数据的保存一样，是文件系统物理实现中的关键问题。

我们自下而上地介绍了名字节点第一关系和第二关系。第一关系的实现包括 INode 及其子类、命名空间镜像 FSImage、编辑日志 FSEditLog、第二名字节点和 FSDirectory 等组件。其中，INode 及其子类在内存中维护着文件系统的文件目录树，并通过命名空间镜像和编辑日志，将目录树和目录树的改动持久化在磁盘上；保存在名字节点磁盘上的命名空间镜像和编辑日志，会被第二名字节点定时地合并，以产生检查点；FSDirectory 是名字节点第一关系的门面，对外提供一个简单的接口。

第二关系的实现以数据结构和过程为中心，缺乏第一关系那样清晰的组件和组件间的协作：围绕着数据块映射和数据节点描述符，研究了数据节点管理和数据块管理的关键流程，如数据节点的添加和撤销，数据节点启动时名字节点的配合，数据节点心跳检查和网络布局感知；而数据块管理则围绕着数据块副本状态和状态转移，通过关键的转移路径，剖析了副本添加、副本删除和副本复制等过程。建立在名字节点第一关系和第二关系基础上的，是远程接口 ClientProtocol 的实现，这个接口提供了对文件元数据的增、删、查、改操作，对文件数据读写的支持和对 HDFS 系统进行管理的能力，是构建客户端的基础。上述模块和功能，由类 NameNode 和 FSNamesystem 组织起来，形成一个统一的整体，承担着 HDFS 主 / 从式体系结构中的主节点角色。

与传统文件系统不同，HDFS 中的元信息复杂而且多变，名字节点将元数据进行分类，并采用不同的实现策略，成功地为 Hadoop 文件系统提供了一个高效的元信息存储和访问机制。

# 第9章　HDFS客户端

我们已经完成了对Hadoop分布式文件系统的名字节点和数据节点的原理解析，可以开始研究客户端的工作机制了。客户端DFSClient和建立在DFSClient基础上的DistributedFileSystem、DFSAdmin和FsShell，屏蔽了HDFS系统的复杂性，为应用提供了标准的Hadoop文件系统应用程序接口、文件系统Shell和管理工具。本章会介绍这些组件的重要操作的实现，并回顾数据节点和名字节点的相关机制。

## 9.1　认识DFSClient

DFSClient，顾名思义，即分布式文件系统客户端，在Hadoop文件系统中，它封装了和HDFS其他实体的复杂交互关系，为应用提供了一个简单的接口。使用这个接口提供的各种功能之前，必须先构造DFSClient对象。

### 9.1.1　DFSClient的构造和关闭

DFSClient提供了4种形式的构造函数，最简单的形式只需要一个保存了配置信息的Configuration对象参数，最复杂的，包含了4个参数，其他构造函数会使用这个方法初始化对象。构造方法的主要任务有两个：

❑读入配置项并初始化一些成员变量。

❑建立和名字节点的IPC连接。

DFSClient的构造函数代码如下，在构造函数中初始化的成员变量的具体含义见代码片段的注释：

```
DFSClient(InetSocketAddress nameNodeAddr, ClientProtocol rpcNamenode,
      Configuration conf, FileSystem.Statistics stats) throws IOException
{
   this.conf = conf; //保存配置信息
   ……

   //和网络相关的一些参数
   this.nnAddress = nameNodeAddr;
   this.socketTimeout = conf.getInt("dfs.socket.timeout",
         HdfsConstants.READ_TIMEOUT);
   this.datanodeWriteTimeout = conf.getInt(
         "dfs.datanode.socket.write.timeout",
         HdfsConstants.WRITE_TIMEOUT);
   this.timeoutValue = this.socketTimeout;
   this.socketFactory = NetUtils.getSocketFactory(conf,
         ClientProtocol.class);
```

```
    // 往数据节点写数据的数据包的最大长度
    this.writePacketSize = conf.getInt("dfs.write.packet.size", 64 * 1024);
    // 读数据块失败时的重试次数
    this.maxBlockAcquireFailures = getMaxBlockAcquireFailures(conf);

    ugi = UserGroupInformation.getCurrentUser();  // 用户信息

    String taskId = conf.get("mapred.task.id");
    if (taskId != null) { // 是一个 MapReduce 任务，使用任务 ID 作为客户端名称
       this.clientName = "DFSClient_" + taskId;
    } else {
       this.clientName = "DFSClient_" + r.nextInt();
    }

    // 默认的数据块大小和副本系数
    defaultBlockSize = conf.getLong("dfs.block.size", DEFAULT_BLOCK_SIZE);
    defaultReplication = (short) conf.getInt("dfs.replication", 3);

    // 建立和名字节点的 IPC 连接
    if (nameNodeAddr != null && rpcNamenode == null) {
       this.rpcNamenode = createRPCNamenode(nameNodeAddr, conf, ugi);
       this.namenode = createNamenode(this.rpcNamenode);
    } else if (nameNodeAddr == null && rpcNamenode != null) {
       // 仅用于测试
       this.namenode = this.rpcNamenode = rpcNamenode;
    } else {
       throw new IllegalArgumentException(……);  // 建立连接失败，抛异常
    }

    // 如果客户端和数据块在一台主机上，是否使用本地读优化，默认值为 false
    this.shortCircuitLocalReads = conf.getBoolean(
          DFSConfigKeys.DFS_CLIENT_READ_SHORTCIRCUIT_KEY,
          DFSConfigKeys.DFS_CLIENT_READ_SHORTCIRCUIT_DEFAULT);
    ……
}
```

DFSClient 的静态成员函数 createRPCNamenode() 和 createNamenode() 用于建立到名字节点的 IPC 连接，它们的差别在于：createNamenode() 在连接失败时，会进行重试。成功构造 DFSClient 对象后，就可以使用对象访问 HDFS 提供的各种功能。

DFSClient.close() 用于关闭客户端。标志位 clientRunning 为 true 时，客户端处于运行状态，设置 clientRunning 为 false，会中断通过该 DFSClient 对象打开的输入流和输出流的工作，然后，方法停止租约管理器，并等待它的退出，最后，关闭到名字节点的 IPC 连接。DFSClient.close() 是一个非常简单的方法，代码如下：

```
public synchronized void close() throws IOException {
   if (clientRunning) {
      leasechecker.close();
      clientRunning = false;
      try {
         leasechecker.interruptAndJoin(); // 停止客户端的租约管理器
      } catch (InterruptedException ie) {
```

```
        }
        RPC.stopProxy(rpcNamenode);
    }
}
```

## 9.1.2　文件和目录、系统管理相关事务

对于文件系统，在本书的讨论中一直区分两种情况：文件和目录相关事务，以及文件数据读写，DFSClient 的原理解析也遵循这个惯例，同时，多功能工具 dfsadmin 依赖的 DFSClient 方法的实现与文件 / 目录 API 类似，都不需要和数据节点打交道，也一并在本小节中研究。

第 8 章选择了增、删、查、改四个典型的目录树操作，即 mkdirs()、delete()、getFileInfo() 和 setOwner()，分析了它们在名字节点上的实现。DFSClient 中，上述方法实现上的差别远比名字节点中的小。除了 mkdirs()，其他三个方法都有一样的模式：在通过了 checkOpen() 检查后，使用远程接口客户端 namenode，调用名字节点上的同名远程方法。而 mkdirs() 在获得必要的参数 masked 后，也使用相同的方式访问名字节点。

这几个方法的代码如下：

```
private void checkOpen() throws IOException { //判断 DFSClient 对象是否关闭
    if (!clientRunning) {
        IOException result = new IOException("Filesystem closed");
        throw result;
    }
}

public boolean mkdirs(String src, FsPermission permission)
                throws IOException {
    checkOpen();
    if (permission == null) {
        permission = FsPermission.getDefault();
    }
    FsPermission masked
          = permission.applyUMask(FsPermission.getUMask(conf));
    ......
    try {
        return namenode.mkdirs(src, masked);
    } catch (RemoteException re) {
        throw re.unwrapRemoteException(……);
    }
}

public boolean delete(String src, boolean recursive) throws IOException {
    checkOpen();
    try {
        return namenode.delete(src, recursive);
    } catch (RemoteException re) {
        throw re.unwrapRemoteException(AccessControlException.class);
    }
}
```

```
public HdfsFileStatus getFileInfo(String src) throws IOException {
    ......
}

public void setOwner(String src, String username, String groupname)
    ......
}
```

多功能工具 dfsadmin 依赖的方法的实现更简单，如安全模式查询、设置使用的 setSafeMode() 方法，就直接调用 ClientProtocol 的同名方法，如下所示：

```
public boolean setSafeMode(SafeModeAction action) throws IOException {
    return namenode.setSafeMode(action);
}
```

为什么这里不需要通过 checkOpen() 检查 DFSClient 的状态呢？dfsadmin 工具只提供了命令行的交互方式，在建立和名字节点的连接后便立即执行命令，也就没有必要检查 DFSClient 是否关闭了。

表 9-1 总结了 DFSClient 中这部分方法的功能和使用的 ClientProtocol 远程方法。

**表 9-1 DFSClient 方法的功能和使用的 ClientProtocol 远程方法**

| DFSClient 方法 | 功　能 | 使用的 ClientProtocol 方法 |
|---|---|---|
| setReplication | 设置文件的副本系数 | setReplication |
| setPermission | 修改文件的权限 | setPermission |
| setOwner | 修改文件的文件主和所属用户组 | setOwner |
| rename | 更改文件名 | rename |
| delete | 从文件系统中删除文件 / 目录 | delete |
| mkdirs | 在给定目录下创建一个子目录 | mkdirs |
| listPaths | 获取某目录下的（部分）目录项 | getListing |
| getCorruptBlocksCount | 获取 HDFS 的当前坏块数 | getStats<br>这几个方法都通过 getStats() 实现，getStats() 返回的数组中包含这些计数器 |
| getDiskStatus | 获取 HDFS 的总空间、已使用空间和剩余空间 | |
| getMissingBlocksCount | 获取 HDFS 中已经丢失数据的数据块数，即没有任何副本的数据块数 | |
| getUnderReplicatedBlocksCount | 获取 HDFS 中需要复制的数据块数，包括正在复制的块 | |
| totalRawCapacity | 获取 HDFS 的总空间 | |
| totalRawUsed | 获取 HDFS 的已使用空间 | |
| datanodeReport | 获取数据节点信息 | getDatanodeReport |
| getBlockSize | 系统配置的数据块大小 | getPreferredBlockSize |
| setSafeMode | 查询 HDFS 是否处于安全模式，让 HDFS 进入、离开安全模式 | setSafeMode |
| saveNamespace | 保存当前名字节点元数据的文件系统映像 | saveNamespace |
| refreshNodes | 刷新用于数据节点管理的 include 文件和 exclude 文件 | refreshNodes |
| finalizeUpgrade | 升级相关（不讨论） | finalizeUpgrade |

（续）

| DFSClient 方法 | 功　能 | 使用的 ClientProtocol 方法 |
|---|---|---|
| distributedUpgradeProgress | 升级相关（不讨论） | distributedUpgradeProgress |
| metaSave | 保存 NameNode 当前活动情况 | metaSave |
| setBalancerBandwidth | 设置数据节点均衡器工作带宽 | setBalancerBandwidth |
| getFileInfo | 获取文件 / 目录元信息 | getFileInfo |
| getContentSummary | 获取文件 / 目录的资源占用情况 | getContentSummary |
| setQuota | 用于目录配额，会对输入参数进行检查 | setQuota |
| setTimes | 修改文件 / 目录的修改时间 / 访问时间 | setTimes |

## 9.1.3　删除 HDFS 文件 / 目录的流程

下面以客户端调用 DFSClient.delete() 为例，介绍文件 / 目录 API 实现时，HDFS 各个主要模块间的配合，如图 9-1 所示。

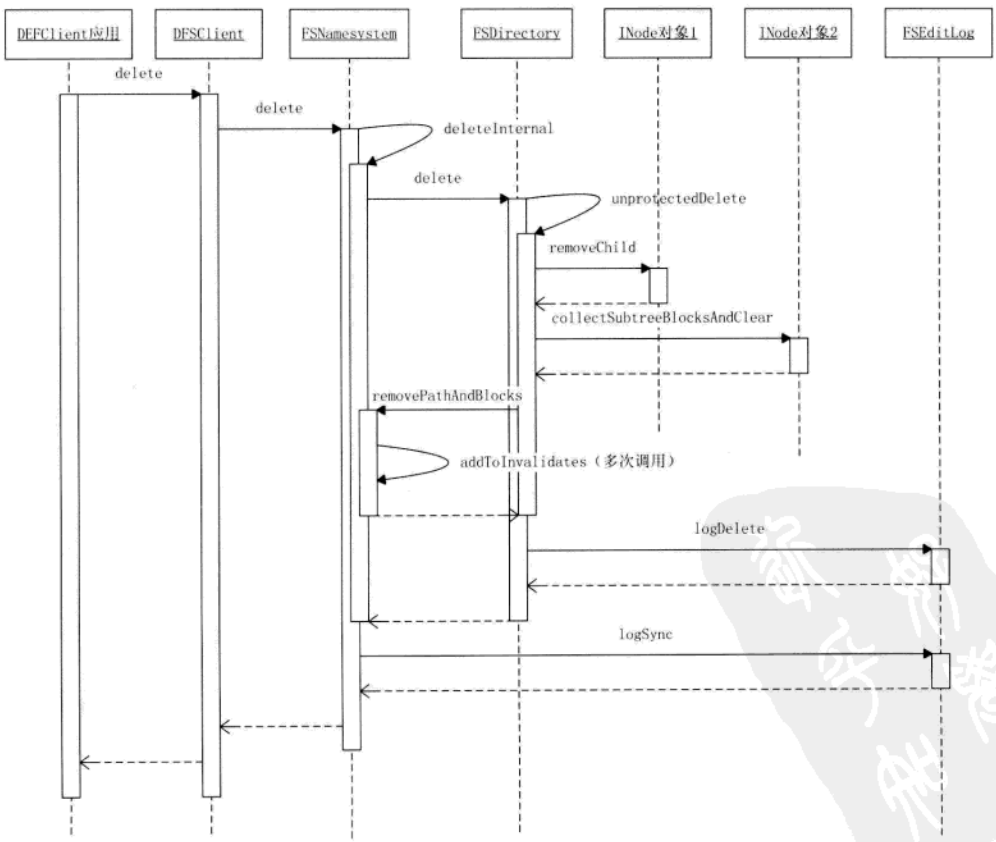



图 9-1　删除操作的执行

图 9-1 省略了一些非关键环节，如 NameNode.delete()，客户端通过远程方法的实现 NameNode.delete()，最终调用了 FSNamesystem 的同名方法。删除操作涉及名字节点第一关系和第二关系，首先通过 FSDirectory 对象在目录树中删除目标文件 / 目录，即图中的 INode.removeChild() 调用，并在返回的 INode 对象上，通过 collectSubtreeBlocksAndClear() 方法获取等待删除的数据块，并使用名字节点第二关系方法 removePathAndBlocks() 和 addToInvalidates()，将上述数据块放入 FSNamesystem 的成员变量 recentInvalidateSets 中。由于删除操作修改了名字节点第一关系，必须将操作记录到日志中，所以也就有了 FSEditLog 对象上的 logDelete() 和 logSync() 调用。

DFSClient.delete() 只把待删除数据块信息放入 FSNamesystem.recentInvalidateSets 中，数据节点删除数据块流程如图 9-2 所示。

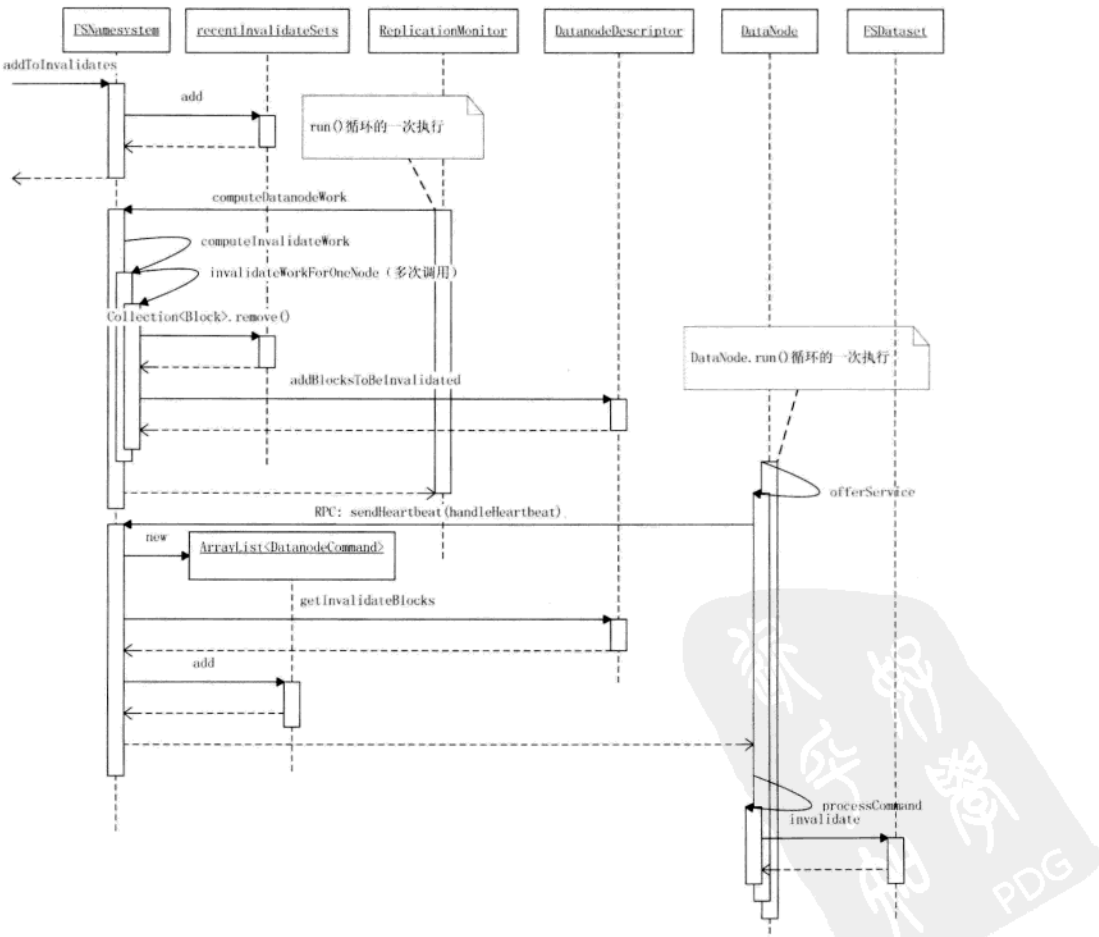



图 9-2 在数据节点上删除数据块的执行

如图 9-2 所示，在删除操作触发的 addToInvalidates() 方法执行完毕后，后续步骤明显划分为两个阶段：

❑删除任务选择阶段

❑数据节点执行阶段

删除任务选择阶段由 ReplicationMonitor 对象所在的线程执行，它选择一部分保存在 recentInvalidateSets 中的待删除数据块，放入对应的 DatanodeDescriptor 对象中，并移除 recentInvalidateSets 中的信息。上述阶段只涉及名字节点第二关系。数据节点执行阶段由数据节点的心跳触发，在心跳处理过程中，数据节点描述符中的删除数据块信息会转化成名字节点指令，作为远程方法 sendHeartbeat() 的返回值发送到数据节点上，数据节点通过文件系统数据集 FSDataset 提供的 invalidate() 方法，在磁盘上删除数据块文件和它的校验信息文件。

看似简单的 DFSClient.delete() 方法，触发了名字节点和数据节点大量的方法，而从调用开始到数据节点上的数据块删除，也会持续一段较长的时间，是一个比较复杂的文件和目录相关事务方法。

## 9.2　输入流

输入流和输出流是 DFSClient 实现中最复杂的部分，它们不但需要和名字节点通信，还需要访问数据节点。相比之下，输入流比输出流简单，读数据的过程中，名字节点只提供了两个远程方法：

❑getBlockLocations()：确定数据的位置。

❑reportBadBlocks()：向名字节点报告客户端发现的坏块。

图 9-3 给出了 HDFS 客户端输入流的继承关系。

这个继承树的根是 Java 的基于字节输入流的所有类的超类 java.io.InputStream，而客户端中实现的 DFSInputStream、DFSDataInputStream 和 BlockReader 是这棵树的叶子节点。中间出现的类，主要来自 java.io 包和 org.apache.hadoop.fs 包，在介绍 Hadoop 文件系统时已经研究过。

位于树的左侧有过滤器流 FilterInputStream，它用于在其他输入流的基础上提供额外的功能，是典型的修饰器模式运用。DataInputStream 继承自 FilterInputStream，它允许应用程序以与机器无关的方式从基础输入流中读取基本的 Java 数据类型，这两个类都属于 java.io 包。FSDataInputStream 来自 org.apache.hadoop.fs 包，是 Hadoop 抽象文件系统中定义的标准输入流。位于图 9-3 树的另一侧有 FSInputStream 和 FSInputChecker。抽象类 FSInputStream 在基本的 InputStream 基础上，引入了接口 Seekable 和接口 PositionedReadable 的功能（但没有全部实现），FSInputChecker 继承自 FSInputStream，提供了对输入数据进行校验的基本能力。

图 9-3 中除了类还有几个接口，重要的有 Closeable 接口，表明输入流是一个可以关闭的数据源；Seekable 和 PositionedReadable 来自 org.apache.hadoop.fs 包，Seekable 提供了在（文件）流中进行随机存取的方法，PositionedReadable 用于从流的某一个位置开始读数据。

客户端中的输入流都实现了上面的这些接口。

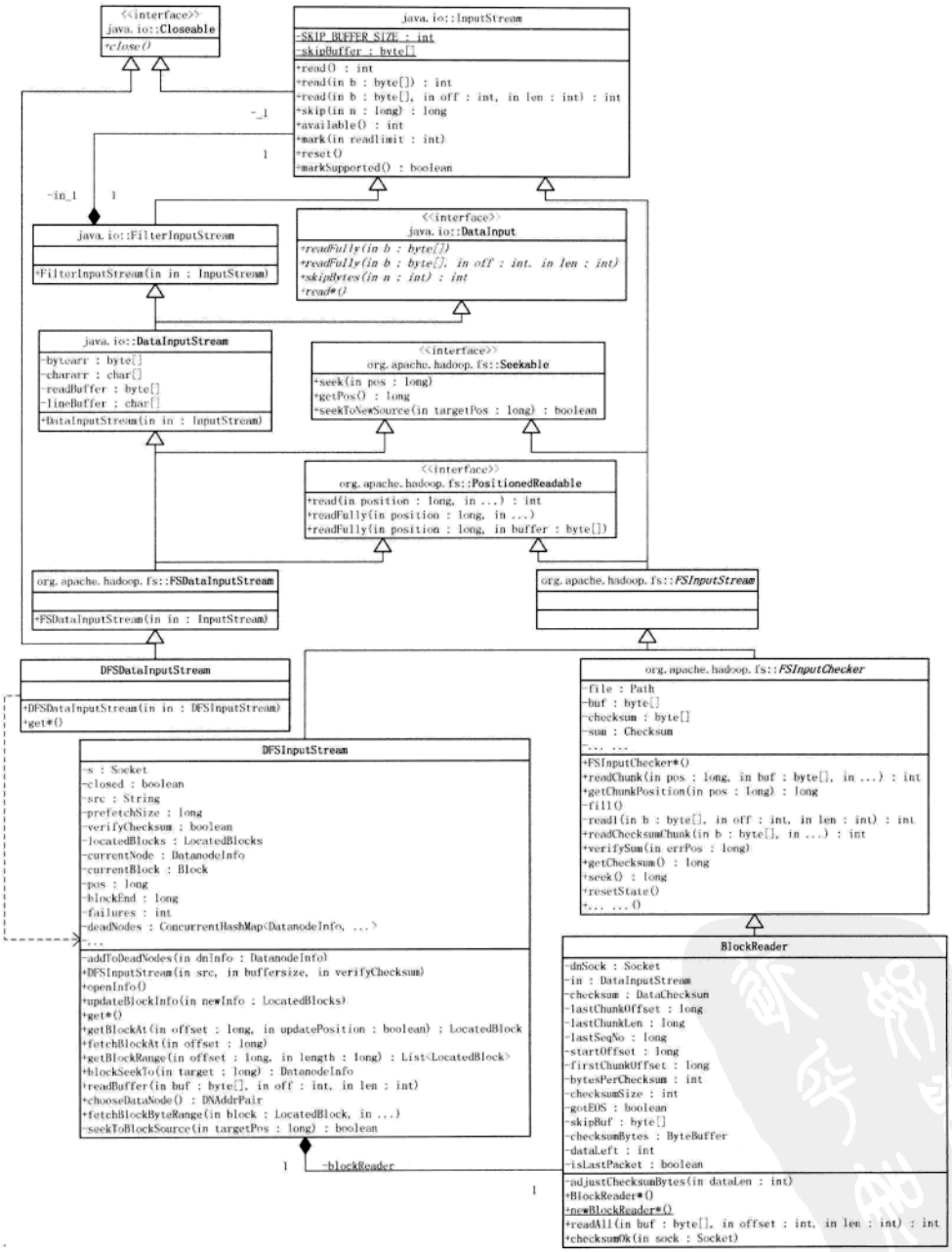



图 9-3 HDFS 客户端输入流的继承关系

由于 org.apache.hadoop.fs.FileSystem 的 open() 方法要求返回 FSDataInputStream 对象，DFSClient 实现了 DFSDataInputStream，但 DFSDataInputStream 对象上的读写操作都委托给了 DFSInputStream；DFSInputStream 对象会拥有一个 BlockReader 实例，利用 BlockReader 继承自 FSInputChecker 而拥有的数据校验能力，简化实现。

### 9.2.1 读数据前的准备：打开文件

DFSClient.open() 用于打开文件，它非常简单，构造并返回一个 DFSInputStream 对象，接下来用户就可以通过这个对象读取 HDFS 文件数据。代码如下：

```
public DFSInputStream open(String src, int buffersize,
      boolean verifyChecksum, FileSystem.Statistics stats)
      throws IOException {
   checkOpen();
   return new DFSInputStream(src, buffersize, verifyChecksum);
}
```

DFSInputStream 的构造过程就复杂了，但构造函数本身很简单，它首先保存输入的各个参数：被读取文件 src，输入缓冲区大小 buffersize，是否对数据进行校验的标志位 verifyChecksum，同时也获取配置项 ${dfs.read.prefetch.size} 的值并保存在 prefetchSize 中，它的默认值是 10 个数据块的大小，如果系统数据块大小是 64MB，那么 prefetchSize 的值为 640MB。

调用远程方法 getBlockLocations() 时，prefetchSize 作为请求的数据长度 length 参数，可一次获取多个数据块的位置信息，以默认值为例，每次调用 getBlockLocations()，会最多获取自起始位置偏移量开始的 10 个数据块的位置信息，后面当输入流读位置越过数据块边界时，不需要每次都要使用 ClientProtocol.getBlockLocations() 获取数据块位置信息。

另外，输入缓冲区大小可通过配置项 ${io.file.buffer.size} 修改，默认值是 4KB，如果使用 DFSClient 的应用瓶颈不在内存，建议适当提高这个配置项的值。相关代码如下：

```
DFSInputStream(String src, int buffersize, boolean verifyChecksum)
      throws IOException {
   …… // 为 this.verifyChecksum、this.buffersize 和 this.src 赋值
   prefetchSize = conf.getLong("dfs.read.prefetch.size", prefetchSize);
   openInfo();
}

synchronized void openInfo() throws IOException {
   LocatedBlocks newInfo = callGetBlockLocations(namenode, src, 0,
         prefetchSize);
   if (newInfo == null) { // 文件不存在
      throw new FileNotFoundException("File does not exist: " + src);
   }

   //I think this check is not correct. A file could have been
   //appended to between two calls to openInfo().
```

```
    // 上述注释的中文是“我认为这个检查是不正确的，在两次调用 openInfo() 间
    // 文件可能由于追加操作而发生变化”
    // 这反映了开源项目发展过程中的一些遗留问题
    if (locatedBlocks != null && !locatedBlocks.isUnderConstruction()
        && !newInfo.isUnderConstruction()) {
      Iterator<LocatedBlock> oldIter
          = locatedBlocks.getLocatedBlocks().iterator();
      Iterator<LocatedBlock> newIter
          = newInfo.getLocatedBlocks().iterator();
      while (oldIter.hasNext() && newIter.hasNext()) {
        if (!oldIter.next().getBlock()
            .equals(newIter.next().getBlock())) {
          throw new IOException(……);
        }
      } //while
    }
    updateBlockInfo(newInfo);
    this.locatedBlocks = newInfo; // 已经解析了数据位置的数据块信息
    this.currentNode = null;
  }
```

DFSInputStream() 初始化一些成员变量后，接着调用 openInfo()。DFSInputStream.callGetBlockLocations() 用于获取被打开文件的初始数据块位置信息。在调用 updateBlockInfo() 后，会初始化 locatedBlocks 和 currentNode。locatedBlocks 是已经解析了位置的数据块信息，currentNode 的类型是 DatanodeInfo，被设置为空，如变量名所示，它是当前 DFSInputStream 对象读取文件数据联系的数据节点。

DFSInputStream.updateBlockInfo() 只在标志位 serverSupportsHdfs200 为 true，并且读数据的目标文件处于构建状态下，才会执行更新数据块信息的逻辑，serverSupportsHdfs200 用于指示系统是否支持 sync()[⊖]。由于数据块长度变化只可能发生在构建文件的最后一个数据块，在保证上述条件后，updateBlockInfo() 利用远程方法 ClientDatanodeProtocol.getBlockInfo()，获得数据节点上的数据块长度，代替名字节点返回的块长度。显然，这是一个分布式系统才有的典型问题，数据节点上的数据块长度发生变化时，名字节点不可能立即知道，通过数据节点获取的长度更准确。但要注意的是，updateBlockInfo() 只和一个数据节点联系，这时，其他数据节点上的数据块长度和这个节点上的数据块长度不一定一致。updateBlockInfo() 代码如下：

```
  private void updateBlockInfo(LocatedBlocks newInfo) {
    …… // 系统不支持长度更新或者 newInfo 不是构建文件的最后一个数据块等特殊情况，返回
    LocatedBlock last = newInfo.get(newInfo.locatedBlockCount() - 1);
    ……
    ClientDatanodeProtocol primary = null;
    DatanodeInfo primaryNode = last.getLocations()[0];
    try {
```

⊖ JIRA HDFS-200 的详细内容可参考 https://issues.apache.org/jira/browse/HDFS-200，在讨论 DFSOutputStream 时，还有更详细的分析。

```
            primary = createClientDatanodeProtocolProxy(primaryNode, conf,
                  last.getBlock(), last.getBlockToken(), socketTimeout);
            // 获取数据节点上数据块的长度
            Block newBlock = primary.getBlockInfo(last.getBlock());
            long newBlockSize = newBlock.getNumBytes();
            long delta = newBlockSize - last.getBlockSize();

            // 如果数据节点上获得的数据块长度和名字节点的不一样，使用数据节点的长度
            last.getBlock().setNumBytes(newBlockSize);
            long newlength = newInfo.getFileLength() + delta;
            newInfo.setFileLength(newlength);
            ……
         } catch (IOException e) {
            if (e.getMessage().startsWith("…….getBlockInfo")) {
               // 数据节点不支持 getBlockInfo 方法，意味着系统不支持 HDFS-200
               serverSupportsHdfs200 = false;
            } else {
               ……
            }
         }
      }
```

至此，DFSInputStream 构造完毕。

## 9.2.2　读数据

DFSInputStream 对象构造完毕后，就可以通过 read() 方法读取数据了。

输入流超类 java.io.InputStream 提供了三种 read() 方法，而 PositionedReadable 接口又添加了对应的三种基于位置的 read()/readFully() 方法，这些读数据的方法在实现上也分成两类：InputStream 中的读方法依赖于流的“下一个”数据字节，所以必须提供机制保证能记录流的当前位置；而 PositionedReadable 中的方法可以在流的任意位置上读数据，也就无所谓“下一个”。

### 1. DFSInputStream 的 read() 方法

能够记住“下一个”数据字节的 DFSInputStream.read() 代码如下，它将输入流中最多 len 个数据字节读入字节数组中：

```
@Override
public synchronized int read(byte buf[], int off, int len)
      throws IOException {
   …… // 检查客户端和 DFSInputStream 对象是打开的，否则抛异常

   if (pos < getFileLength()) { // 读取的位置未到文件尾
      int retries = 2;
      while (retries > 0) {
         try {
            if (pos > blockEnd) { // 已经读到某个数据块的尾部
               currentNode = blockSeekTo(pos);
```

```
                }
                // 计算读取的长度
                int realLen = Math.min(len, (int) (blockEnd - pos + 1));
                int result = readBuffer(buf, off, realLen); // 读入数据

                if (result >= 0) {
                    pos += result; // 更新 pos
                } else {
                    throw new IOException("Unexpected EOS from the reader");
                }
                ......
                return result;
            } catch (ChecksumException ce) {
                throw ce; // 校验错误，抛异常
            } catch (IOException e) {
                ......
                blockEnd = -1;
                if (currentNode != null) {
                    addToDeadNodes(currentNode); // 将节点加入黑名单
                }
                if (--retries == 0) { // 到达重试次数上限，抛异常
                    throw e;
                }
            }
        }
    }
    return -1;
}
```

这是一个比较“平淡”的方法，它将难点都交给了 blockSeekTo() 和 readBuffer()。这个方法的要点包括：DFSInputStream 的成员变量 pos 用于保存“下一个”数据字节的位置，它的初始值为 0，当 readBuffer() 成功读入数据后，pos 会被更新以跟踪读位置；读数据出现异常时，如果是校验错误，由于在 readBuffer() 中已经尝试其他可用副本，所以直接抛异常，如果是其他 I/O 异常，则将当前访问的数据节点加入黑名单后重试。

### 2. blockSeekTo() 和 readBuffer() 方法

DFSInputStream.readBuffer() 执行读数据的具体逻辑，而 blockSeekTo() 则在读操作越过某个数据块边界时，用于建立到下一个数据块所在数据节点的联系，并初始化读数据需要的 BlockReader 对象。由于 blockEnd 的初值是 –1，所以，通过 DFSClient.open() 获得 DFSInputStream 对象后，立即调用 read() 方法读数据，就会触发 blockSeekTo() 的执行。这个方法先选择合适的目标数据节点，然后构造 BlockReader 对象。代码如下：

```
private synchronized DatanodeInfo blockSeekTo(long target)
        throws IOException {
    ......
    if (blockReader != null) {……}// 如果 blockReader 非空，关闭该对象
    if (s != null) {……}// 到上一个数据节点的 Socket 连接还打开着，关闭连接
```

```
DatanodeInfo chosenNode = null;
int refetchToken = 1;
while (true) { // 多次尝试，直到没有可用的数据节点
   LocatedBlock targetBlock = getBlockAt(target, true);
   long offsetIntoBlock = target - targetBlock.getStartOffset();

   DNAddrPair retval = chooseDataNode(targetBlock);
   chosenNode = retval.info;
   InetSocketAddress targetAddr = retval.addr;

   // 能否进行本地读，否则连接数据节点
   Block blk = targetBlock.getBlock();
   ……
   if (shouldTryShortCircuitRead(targetAddr)) {
      try {
         blockReader = getLocalBlockReader(conf, src, blk,
               accessToken, chosenNode,
               DFSClient.this.socketTimeout, offsetIntoBlock);
         return chosenNode;
      } catch (AccessControlException ex) {
         ……
         shortCircuitLocalReads = false;
      } catch (IOException ex) {
         if (refetchToken > 0 && tokenRefetchNeeded(ex, targetAddr))
         {
            ……// 安全相关
            continue;
         } else {
            ……
         }
      }
   }

   try {
      s = socketFactory.createSocket();
      // 建立到数据节点的 Socket 连接
      NetUtils.connect(s, targetAddr, socketTimeout);
      s.setSoTimeout(socketTimeout);
      blockReader = BlockReader.newBlockReader(s, src,
      blk.getBlockId(), accessToken,blk.getGenerationStamp(),
      offsetIntoBlock,blk.getNumBytes() - offsetIntoBlock,
      buffersize,verifyChecksum, clientName);
   return chosenNode;
} catch (IOException ex) {
   // 处理异常
   if (refetchToken > 0 && tokenRefetchNeeded(ex, targetAddr))
   {
      ……// 安全相关
   } else {
      ……
```

```
                addToDeadNodes(chosenNode);
            }
                ..... // 在 s 不为空的情况下关闭 s
            }
            s = null;
            }
        }
    }
```

DFSInputStream.getBlockAt() 用于获取文件指定位置的 LocatedBlock 对象，该对象包含了数据块的位置信息。这个方法会先利用 DFSInputStream 中已经缓存的 LocatedBlock 实例，如 openInfo() 中获取并缓存的信息，否则，使用 callGetBlockLocations() 方法获取并缓存数据块的位置信息。当然，这个调用也会从名字节点获取并缓存多个数据块的 LocatedBlock 对象，当然也包括指定位置的对象。

接下来，chooseDataNode() 会选择一个合适的数据节点。NameNode.getBlockLocations() 会对数据块所在的数据节点列表进行整理，将离客户端网络距离近的节点放在数组的最前面，chooseDataNode() 在这个基础上，排除掉客户端联系不上的“死”节点（通过 addToDeadNodes() 方法添加的数据节点），返回可用的数据节点位置信息。由于篇幅的原因，就不给出 getBlockAt() 和 chooseDataNode() 的代码了。

顺利找到数据节点后，blockSeekTo() 开始构造 BlockReader 对象，当系统支持本地读优化，而本地节点恰好被 chooseDataNode() 选中时，方法会构造一个 BlockReaderLocal（它是 BlockReader 的子类）对象，直接读取本地文件系统上的数据块数据。对 MapReduce 应用来说，本地读是一个非常有用的优化，执行 map 操作的节点和被处理的数据往往位于同一台主机，读操作不通过数据节点，可大幅降低节点的负载。

当需要通过 TCP 连接使用数据节点的流式接口时，blockSeekTo() 简单地构造 Socket 对象并连接到目标数据节点，剩下的事情由 BlockReader.newBlockReader() 完成。

由于已经和数据节点建立了 TCP 连接，newBlockReader() 在 Socket 上获取输出流后，开始发送操作码为 81 的流式接口请求包，并刷出数据，构造请求包需要的信息，都通过方法参数传递进来。应答包的处理也很简单：当返回码不是 DataTransferProtocol.OP_STATUS_SUCCESS 时，根据情况抛异常；如果操作成功，读入应答。

应答的第一部分内容用于构造数据校验检查器 checksum，接下来的字段提供了应答数据实际开始的位置。由于校验的需要，数据节点在某些条件下，会给客户端多发送一些数据，实际应答数据的开始位置保存在 firstChunkOffset 中。相关代码如下：

```
public static BlockReader newBlockReader(Socket sock, String file,
      long blockId, Token<BlockTokenIdentifier> accessToken,
      long genStamp, long startOffset, long len, int bufferSize,
      boolean verifyChecksum, String clientName) throws IOException {
   DataOutputStream out = new DataOutputStream(.....);// 获取 Socket 输出流

   // 读请求的请求数据帧
   out.writeShort(DataTransferProtocol.DATA_TRANSFER_VERSION); // 版本号
```

```
        out.write(DataTransferProtocol.OP_READ_BLOCK); // 操作码
        out.writeLong(blockId); // 数据块 ID
        out.writeLong(genStamp); // 时间戳
        out.writeLong(startOffset); // 偏移量
        out.writeLong(len); // 数据长度
        Text.writeString(out, clientName); // 客户端名字
        accessToken.write(out); // 访问令牌
        out.flush();

        DataInputStream in = new DataInputStream(……);// 获取 Socket 输入流

        short status = in.readShort();// 返回码
        if (status != DataTransferProtocol.OP_STATUS_SUCCESS) { // 出错处理
           if (status == DataTransferProtocol.OP_STATUS_ERROR_ACCESS_TOKEN) {
              throw new InvalidBlockTokenException(……);
           } else {
              throw new IOException(……);
           }
        }

        // 构造校验和检查器
        DataChecksum checksum = DataChecksum.newDataChecksum(in);

        long firstChunkOffset = in.readLong();// 读入偏移量并检查
        if (firstChunkOffset < 0|| firstChunkOffset > startOffset
              || firstChunkOffset >=
                    (startOffset + checksum.getBytesPerChecksum())) {
           throw new IOException(……);
        }

        return new BlockReader(file, blockId, in, checksum, verifyChecksum,
              startOffset, firstChunkOffset, sock);
    }
```

数据节点接下来就会发送包含文件内容的数据包，但客户端这边还有工作未完成，newBlockReader() 方法会创建 BlockReader 对象，这个对象有很多字段，所以构造函数也需要很多参数，后面的讨论中会陆续给出这些成员变量的作用。

在上面的 read() 方法中，readBuffer() 用于读入数据，但这个方法将具体的读操作委托给了 BlockReader 对象，它的主要作用是在读取数据块数据发生校验错误时进行重试。一切正常时，读入数据后 readBuffer() 就返回了。当出现错误时，如果是数据校验的问题，表明这个数据节点上的数据块出现问题，通过 reportChecksumFailure() 向名字节点汇报错误，然后将 retryCurrentNode 设置为 false。

方法的变量 retryCurrentNode 用于标识是否对当前数据节点进行重试，由于数据节点上的数据块已经损坏，也就没有必要再次在当前数据节点上读取数据了。如果是一般的 IOException 异常，如客户端长时间没有发生读操作导致数据节点关闭连接，则可以在当前节点上继续进行尝试。相关代码如下：

```
private synchronized int readBuffer(byte buf[], int off, int len)
      throws IOException {
   IOException ioe;
   boolean retryCurrentNode = true;

   while (true) {
      try {
         return blockReader.read(buf, off, len);
      } catch (ChecksumException ce) {
         ......
         reportChecksumFailure(src, currentBlock, currentNode);
         ioe = ce;
         retryCurrentNode = false;
      } catch (IOException e) {
         ......
         ioe = e;
      }
      boolean sourceFound = false;
      if (retryCurrentNode) {
         sourceFound = seekToBlockSource(pos); // 在当前数据节点上进行重试
      } else {
         addToDeadNodes(currentNode); // 将当前节点列入黑名单
         sourceFound = seekToNewSource(pos); // 寻找新的可用数据节点
      }
      if (!sourceFound) {
         throw ioe; // 没有可用数据节点，抛异常
      }
      retryCurrentNode = false;
   }
}
```

readBuffer() 使用的 seekToBlockSource() 和 seekToNewSource() 方法用于在当前数据节点或另一个新数据节点上，使用前面分析过的 blockSeekTo() 方法建立连接。它们的实现都很简单，就不再给出具体代码了。

DFSClient.reportChecksumFailure() 用于向名字节点报告客户端发现的坏块信息，它最终使用了远程方法 ClientProtocol.reportBadBlocks() 通知名字节点，代码如下：

```
void reportChecksumFailure(String file, LocatedBlock lblocks[]) {
   try {
      reportBadBlocks(lblocks);
   } catch (IOException ie) {
      ......
   }
}

public void reportBadBlocks(LocatedBlock[] blocks) throws IOException {
   namenode.reportBadBlocks(blocks);
}
```

### 3. BlockReader 的实现原理

newBlockReader() 创建的 BlockReader 对象继承自 FSInputChecker，在介绍 Hadoop 文件系统时已经研究过 FSInputChecker，这里回顾它的实现要点。由于需要对数据进行校验，FSInputChecker 使用了两个输入缓冲区，缓冲区 buf 保存了输入数据，缓冲区 checksum 则包含对应的校验数据。readChunk() 是 FSInputChecker 的抽象方法，用于读入一块原始数据及其对应的校验数据。方法声明如下：

```
abstract protected int readChunk(long pos, byte[] buf, int offset, int len,
      byte[] checksum) throws IOException;
```

使用 readChecksumChunk() 读取数据时，它会使用校验数据对原始数据进行校验，以保证数据的完整性，所以，它会通过 readChunk() 至少读取一块可校验的数据。FSInputChecker 的私有方法 fill() 和 read1() 利用了 readChecksumChunk()，fill() 方法会将数据读入内部缓冲区 buf 中，read1() 则提供了在条件允许情况下使用外部缓冲区读取数据的优化，以减少数据的拷贝。对外，FSInputChecker 实现了两个 java.io.InputStream 中定义的 read() 方法。

FSInputChecker 中与读数据相关方法的调用关系如图 9-4 所示。

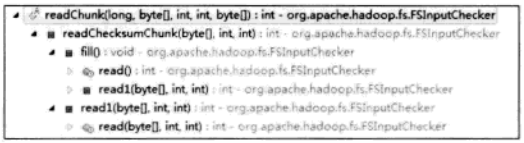



图 9-4　FSInputChecker 中读数据方法的调用关系

显然，BlockReader 要实现 readChunk() 方法，从数据节点的返回结果中获取一块原始数据及其对应的校验数据，其中原始数据放入缓冲区 buf 中，而缓冲区 checksum 用于保存对应的校验数据。这个方法使用了 BlockReader 的大量成员变量，理解这些成员变量的具体作用，需要回顾数据节点中 BlockSender.sendChunks() 的一些细节。在基于块的校验机制上实现基于流的读接口，是这部分代码的一个难点，为了保证客户端能够对文件数据进行校验，sendChunks() 会返回包含用户请求数据的校验块，以图 9-5 为例，由于请求数据的起始位置和结束位置都不在校验块的边界上，所以，应答数据最前面和最后面可能包含了请求范围外的数据。

图 9-5 的上半部分给出了 BlockReader 初始化后，对象中一些关键成员变量的取值。其中，firstChunkOffset 保存了返回数据在数据块中的起始位置，startOffset 则是请求数据在数据块中的起始位置，bytesPerChecksum 记录了校验块的大小，这些成员变量的值在初始化后就不会发生变化。变量 lastChunkOffset 保存了下一次读取文件数据在数据块中的起始位置，也就是上次数据读取的结束位置，dataLeft，顾名思义，就是这个数据包中还有多少未读取的数据。

readChunk() 方法的分析从中间部分的代码开始。当 dataLeft 小于或等于 0 时，即当前数据包已经没有数据可读的情况下，意味着要开始处理一个新的数据包，dataLeft 的初始是 0，所以，第一次调用 readChunk() 时，方法会执行这段代码，读入数据包的包头，包括包长度、偏移量、顺序号等。接下来，调用 adjustChecksumBytes()，保证用于读入校验数据的 BlockReader 成员变量 checksumBytes 有足够的空间，并通过 IOUtils.readFully() 读入这个

数据应答包的所有校验信息。也就是说，如果这个应答包的数据分成了多个校验块，也会一次读入所有的校验数据。所以，成员变量 dataLeft 的未读数据都是文件数据，不包括校验数据。代码如下：

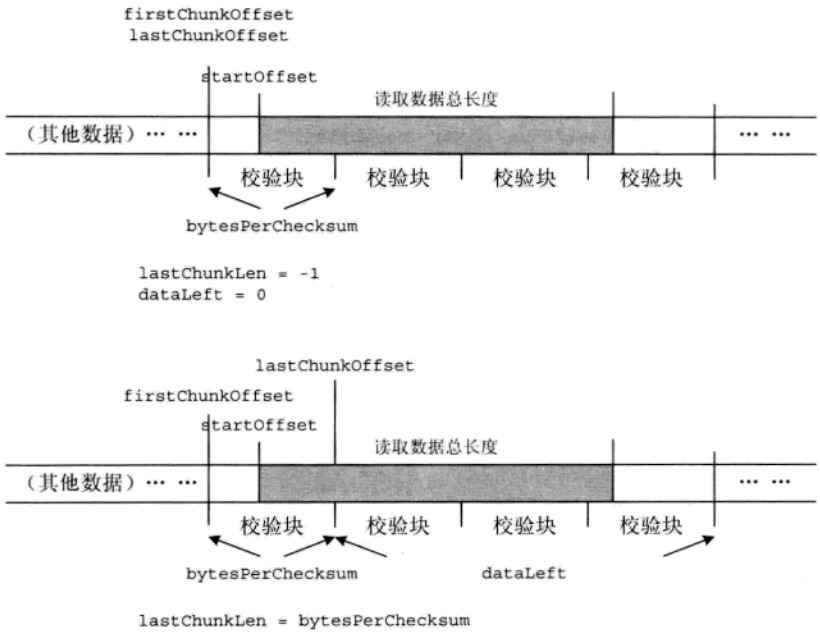



图 9-5 BlockReader 中与数据读取有关的成员变量

```
protected synchronized int readChunk(long pos, byte[] buf, int offset,
      int len, byte[] checksumBuf) throws IOException {
   if (gotEOS) { //是否已经读入这个数据块上需要的全部数据
      startOffset = -1;
      return -1;
   }

   long chunkOffset = lastChunkOffset;
   if (lastChunkLen > 0) {
      chunkOffset += lastChunkLen;
   }
   if ((pos + firstChunkOffset) != chunkOffset) {
      throw new IOException(……);
   }

   //开始处理一个新的数据包
   if (dataLeft <= 0) {
      //读入数据包的包头
      int packetLen = in.readInt(); //包长度
      long offsetInBlock = in.readLong(); //偏移量
```

```
            long seqno = in.readLong(); //顺序号
            boolean lastPacketInBlock = in.readBoolean(); //最后应答包标志
            int dataLen = in.readInt(); //数据长度
            ……//检查包头数据的合法性，如序列号是否正确等，出错时抛出异常

            lastSeqNo = seqno; //更新成员变量
            isLastPacket = lastPacketInBlock;
            dataLeft = dataLen;

            adjustChecksumBytes(dataLen);
            if (dataLen > 0) { //读入包头后面所有的校验数据
               IOUtils.readFully(in, checksumBytes.array(),
                     0, checksumBytes.limit());
            }
         }

         int chunkLen = Math.min(dataLeft, bytesPerChecksum);

         if (chunkLen > 0) {
            // 获得一个校验块大小的数据
            IOUtils.readFully(in, buf, offset, chunkLen);
            // 获得数据的校验数据
            checksumBytes.get(checksumBuf, 0, checksumSize);
         }

         dataLeft -= chunkLen;
         lastChunkOffset = chunkOffset;
         lastChunkLen = chunkLen;

         if ((dataLeft == 0 && isLastPacket) || chunkLen == 0) {
            gotEOS = true; // 已经处理完读请求的所有应答，设置 gotEOS 标记位
         }
         if (chunkLen == 0) {
            return -1; // 读到数据块尾
         }

         return chunkLen;
      }
```

BlockReader.readChunk() 的目标是：获得一个校验块大小的数据和数据的校验和，在方法实现中，只用了两行代码，就实现了这个目标。文件数据还是通过到数据节点的 Socket 连接的输入流读入，数据的校验和由于保存在 checksumBytes 中，所以利用 ByteBuffer.get() 方法填充到缓冲区 checksumBuf 中。读入一块数据后，BlockReader 的成员变量 lastChunkOffset、dataLeft 和 lastChunkLen 都会发生变化，图 9-5 的下半部分给出了读入第一个校验块后，上述成员变量的值的含义。

readChunk() 方法的其他代码主要用于参数检查和 BlockReader 成员变量的更新。注意，布尔变量 gotEOS 可判断是否已经读入了一次读请求的所有数据。另一个需要注意的地方是，成员变量 chunkLen 通常都等于 bytesPerChecksum 的值，只有在文件的最后一个校验块时，才可能出现 chunkLen 的值小于 bytesPerChecksum 的情况，这时，请求的数据已经到达数据

块的尾部，而且数据没有填满最后一个校验块。

终于，可以开始读入文件数据。读数据的 read() 方法的代码如下：

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

   // 第一次读入数据，跳过 firstChunkOffset 和 startOffset 间的内容
   if (lastChunkLen < 0 && startOffset > firstChunkOffset && len > 0) {
      int toSkip = (int) (startOffset - firstChunkOffset);
      if (skipBuf == null) {  // 准备缓冲区
         skipBuf = new byte[bytesPerChecksum];
      }
      if (super.read(skipBuf, 0, toSkip) != toSkip) { // 读入并抛弃数据
         throw new IOException("Could not skip required number of bytes");
      }
   }

   boolean eosBefore = gotEOS;
   int nRead = super.read(buf, off, len);
   if (dnSock != null && gotEOS && !eosBefore && nRead >= 0 && needChecksum())
   {
      checksumOk(dnSock); // 上报数据节点，数据校验成功
   }
   return nRead;
}
```

BlockReader.read() 覆盖了 FSInputChecker 中的实现，但具体的读数据操作，还是通过 FSInputChecker.read() 进行的。它需要处理两个情况：第一次读入数据时，BlockReader.read() 通过读取并抛弃 firstChunkOffset 和 startOffset 间的文件内容，排除应答数据最前面可能出现的，请求范围外的数据；另外，顺利读入并校验所有数据后，通过 checksumOk() 方法通知数据节点。代码如下：

```
private void checksumOk(Socket sock) {
   try {
      OutputStream out = NetUtils.getOutputStream(sock,
            HdfsConstants.WRITE_TIMEOUT);
      byte buf[] = {
            (DataTransferProtocol.OP_STATUS_CHECKSUM_OK>>> 8) & 0xff,
            (DataTransferProtocol.OP_STATUS_CHECKSUM_OK) & 0xff };
      out.write(buf);
      out.flush();
   } catch (IOException e) {
      ......
   }
}
```

如果客户端读取的是一个完整的数据块，数据节点会接收这个通知，并在数据块扫描器中进行标记，以减少扫描器的工作。由于客户端不清楚读入的是否是完整的数据块，所以，它必须无条件发送这个通知。

本节的开始分析了 DFSInputStream 的两类读数据的方法，PositionedReadable 接口也提

供了三个 read() 方法，DFSInputStream 实现了其中的一个，它和上面分析的读数据方法类似，也充分利用了 BlockReader 提供的数据校验功能，实现也不算太复杂，由于篇幅的原因，就不再详细讲解了。

### 9.2.3 关闭输入流

应用读入所需的数据后，需要关闭输入流。DFSInputStream.close() 用于关闭流，这个方法很简单，检查了对象和所属 DFSClient 对象的状态后，关闭可能打开的 BlockReader 对象和到数据节点的 Socket 对象，在调用了父类的 close() 方法后，设置标志位 closed。代码如下：

```
public synchronized void close() throws IOException {
   if (closed) {
      return;
   }
   checkOpen(); //检查

   if (blockReader != null) {
      blockReader.close(); // 关闭 BlockReader 对象
      blockReader = null;
   }

   if (s != null) {
      s.close(); // 关闭 Socket
      s = null;
   }
   super.close();
   closed = true;
}
```

BlockReader 对象的 close() 方法更加简单，由于它使用的是 DFSInputStream 的 Socket 对象，也就没有什么资源需要释放。

```
public synchronized void close() throws IOException {
   startOffset = -1;
   checksum = null;
}
```

DFSInputStream 中还有一些方法，如 skip() 和 seek()，它们方便了用户对流的使用。相对于 read() 等方法，它们的实现不算复杂，就不再赘述了。

在图 9-3 中，DFSClient 不但实现了 DFSInputStream，也实现了继承自 FSDataInputStream 的内部类 DFSDataInputStream。和 DFSInputStream 相比，DFSDataInputStream 十分简单，它只是为了兼容 Hadoop 文件系统界面而引入的包装器，其大部分操作都委托给了 DFSInputStream。

### 9.2.4 读取 HDFS 文件数据的流程

我们研究了读取 HDFS 文件数据最重要的三个方法：DFSClient.open()、DFSInputStream 的 read() 和 close() 方法，它们的实现使用了 DFSClient、DFSInputStream 和 BlockReader（包

括 BlockReader 的父类 FSInputChecker）的十几个成员函数，复杂但是很有条理。

图 9-6 整理了 DFSClient.open() 执行过程中的重要调用，以及名字节点、数据节点上的关键过程。

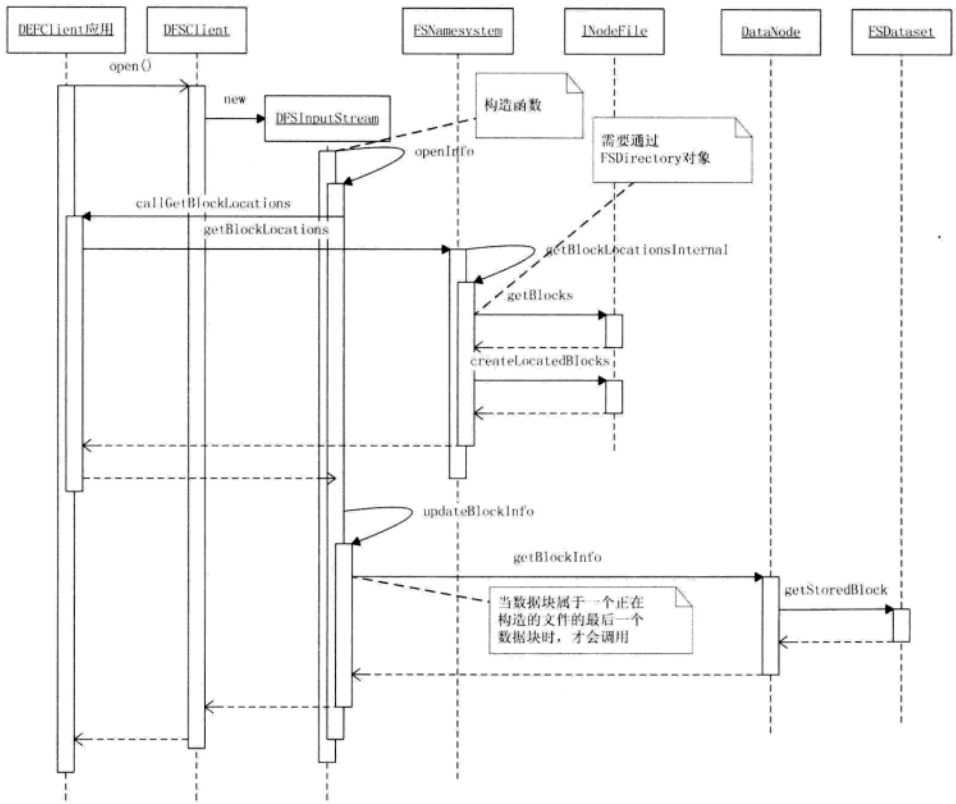



图 9-6 DFSClient.open() 序列图

HDFS 应用调用 DFSClient.open() 时，会创建 DFSInputStream() 对象，在构造对象时，DFSInputStream.openInfo() 方法会通过远程接口访问名字节点，调用 getBlockLocations() 方法，以获取数据块的位置信息。这个信息保存在 INodeFile（或它的子类 INodeFileUnderConstruction）中。如果数据块是属于某个处于构建状态文件的最后一个数据块时，openInfo() 调用对象的 updateBlockInfo() 方法，通过数据节点提供的远程方法 getBlockInfo() 更新该数据块的信息。

**注意** *updateBlockInfo() 方法在实际的系统运行中比较少有机会使用，打开输出流时，客户端主要还是和名字节点联系。*

图 9-7 给出了 DFSInputStream 对象构造后，第一次调用 read() 时的序列图，这是该方法

的一个比较复杂的执行流，后续对 read() 的调用就不会执行这么多的操作了。

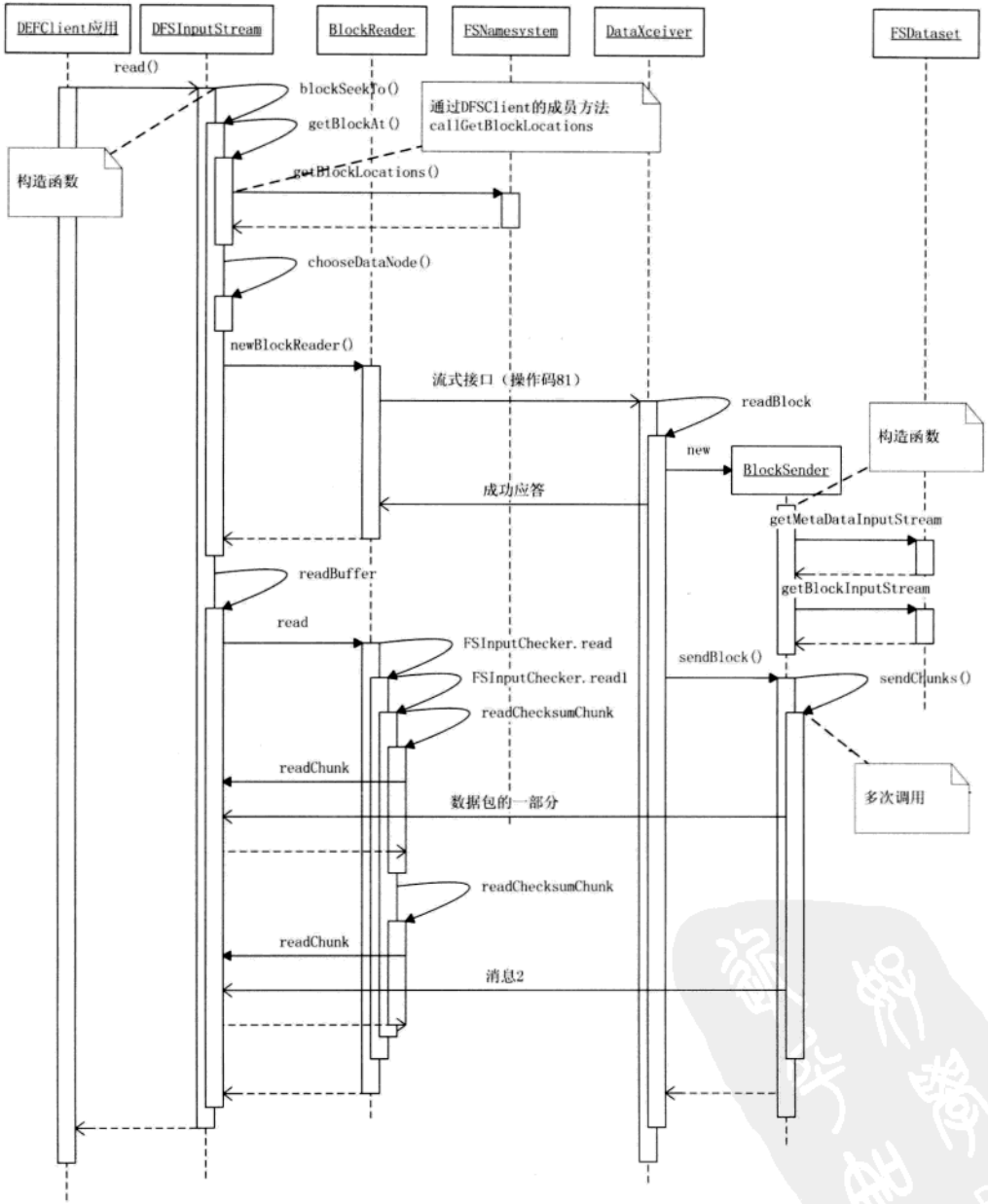



图 9-7　DFSInputStream.read() 序列图

由于未和数据节点建立联系，DFSInputStream 对象使用 blockSeekTo()，建立和某一个拥有数据块的数据节点的联系。在获取读取数据的数据块所在的位置信息并选择合适的数据节点后，blockSeekTo() 通过 BlockReader.newBlockReader() 方法建立和数据节点流式接口的连接并发送读请求。数据节点的服务器 DataXceiver 接收并处理该请求，它创建 BlockSender 对象，数据节点上的后续工作主要由该对象完成。BlockSender 的构造函数通过文件数据集 FSDataset，打开到数据块文件和校验信息文件的输入流后，由 TCP 连接给客户端一个成功的应答，收到应答后，newBlockReader() 方法完成 BlockReader 对象的构造并返回该对象。

数据节点在发送应答头后，就会立即通过 sendChunks()，发送（多个）应答数据包，直到客户端所需数据全部发送完毕。

在客户端这边，序列图接下来出现了连续几个读数据的方法，DFSInputStream.readBuffer() 调用 BlockReader 的 read() 方法，这个方法又调用了 BlockReader 父类 FSInputChecker 的 read() 方法，该 read() 进一步调用 read1()，接着 read1() 使用 readChecksumChunk()，最终调用了 BlockReader 实现的 readChunk()，由 readChunk() 从和数据节点的 TCP 连接上读入数据。这些方法的作用如下：

- ❑DFSInputStream 的成员函数 readBuffer() 处理读入数据时的错误，并尝试在下一个可用的数据节点上重新读入数据。
- ❑BlockReader.read() 处理两个特殊情况：读取并抛弃校验带来的，数据节点返回的多余数据，顺利读入并校验所有数据后将校验结果上报到数据节点。
- ❑FSInputChecker 的 read() 会连续读入多个校验块的数据，直到读满缓冲区或者读到数据块尾。
- ❑FSInputChecker.read1() 主要是为了提高读数据的效率，尽量避免内存拷贝，即尽量使用用户缓冲区调用 readChecksumChunk()，避免使用 FSInputChecker 的内部工作缓冲区，也就避免了从内部工作缓冲区到用户缓冲区的数据拷贝。
- ❑readChecksumChunk() 方法最多读入一个校验块数据，并对数据进行校验。
- ❑BlockReader.readChunk() 实现了 FSInputChecker 的抽象方法，它从数据节点流式接口上读入一个校验块数据。

这些方法分工配合，完成了从数据节点读入 HDFS 文件数据需要的各种各样的处理。

## 9.3 输出流

输出流 DFSOutputStream 用于向 HDFS 文件中写入数据，它也拥有一个复杂的实现。图 9-8 是 DFSOutputStream 的类图。

继承树的根是 Java 所有基于字节输出流的超类 java.io.OutputStream。和输入流不同，DFSOutputStream 的继承层次不多，在 DFSOutputStream 和 OutputStream 之间是 FSOutputSummer，它属于 org.apache.hadoop.fs 包，提供了在流上计算数据校验和并输出的能力。但 DFSOutputStream 依赖不少内部类，如数据包 Packet、数据发送器 DataStreamer 和应答处理器

ResponseProcessor，内部类 DataStreamer 和 ResponseProcessor 甚至运行在独立的线程中。

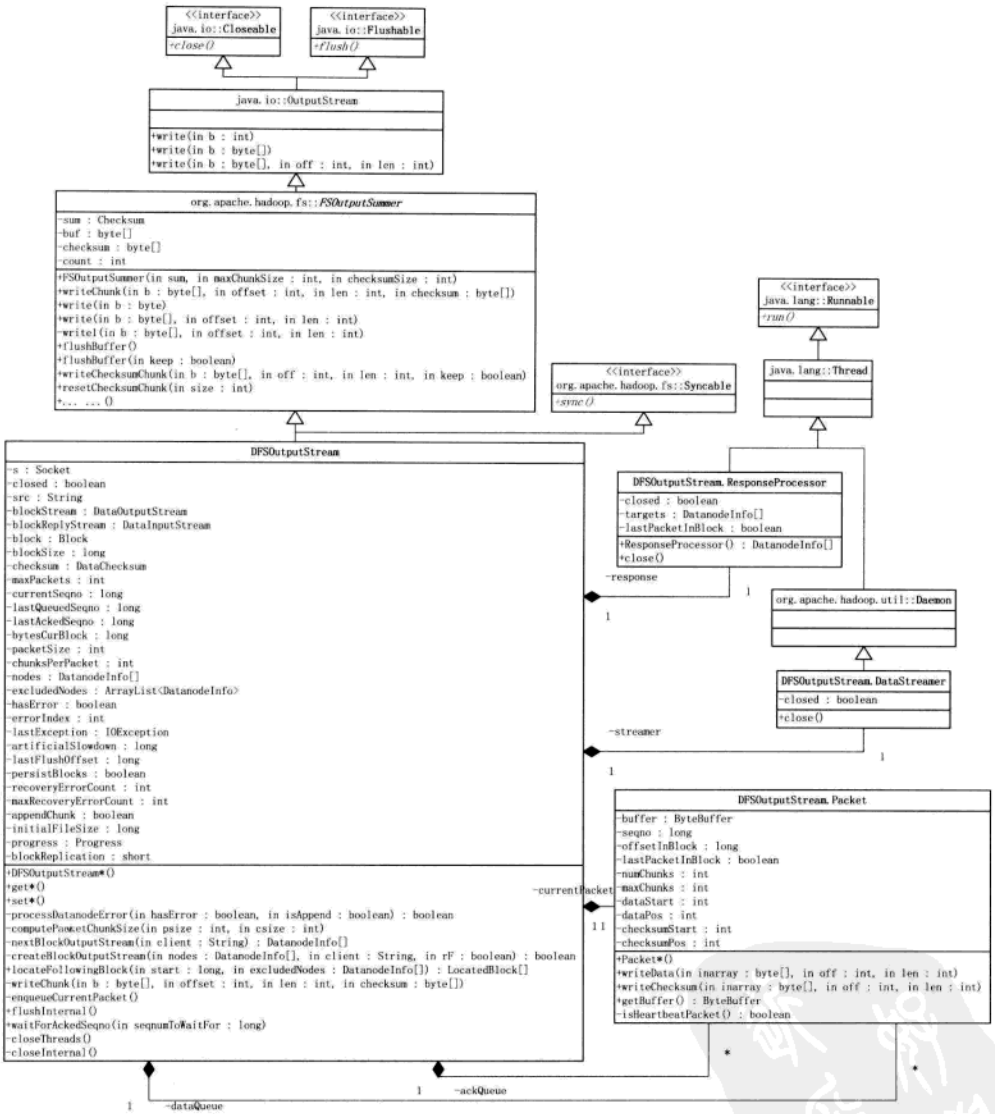



图 9-8　HDFS 客户端输出流的类图

正常情况下，创建一个新文件并往文件写数据的关键步骤如下：

1）通过 DFSClient 的 create() 方法创建一个新文件，获得一个输出流，构造输出流时就会通过远程调用 ClientProtocol.create() 在名字节点上创建文件。

2）通过输出流的 write() 方法写数据，数据会先缓存在内存中，到达一定条件后，才会

放入输出队列，并由 DataStreamer 对象处理，该对象拥有自己的线程。

3）DataStreamer 处理数据包时，如果数据流管道还没有建立起来，会联系名字节点，使用远程方法 ClientProtocol.addBlock() 分配新的数据块，然后使用数据节点操作码为 82 的流式接口创建数据流管道。

4）DFSOutputStream 的 DataStreamer 对象通过写请求数据包往数据节点发送数据，同时，另外一个线程 ResponseProcessor 对象，会接收数据节点对数据包的应答。

5）当写满一个数据块时，DataStreamer 会写出一个长度为 0 的数据包，数据节点接收到这个数据包时，会往名字节点提交当前数据块，并开始释放请求处理的资源。

6）如果客户端还有数据写入，则重复上述分配数据块、写数据等过程。

7）最后，应用通过流的 close() 方法关闭输出流，远程方法 ClientProtocol.complete() 会被调用，名字节点上的打开文件会被关闭，租约会被释放。

8）在打开文件到关闭文件期间，客户端会不断地通过 ClientProtocol.renewLease() 远程方法更新租约，保证对打开文件的写权限。

通过 DFSClient.append() 打开已经存在的文件，追加数据，也会触发写数据流程。和上面描述的创建文件并写数据不同，append() 往往在创建输出流对象的时候，就可以联系数据节点并建立数据流管道，其他过程和上述步骤并没有太大差异。

在故障处理方面，写数据也远比读操作复杂。读操作有这几种常见的问题：联系不上名字节点，联系不上数据节点，或读文件时数据校验失败。对于名字节点故障，由于系统中没有第二个可用节点，一般会导致操作失败；而数据块往往有多个副本，连接数据节点失败或读数据时，数据校验失败，可以在其他副本上重新尝试。写操作不同，它要保证将数据成功地写入多个副本，也就面临着更多的挑战：

- 和读操作一样，如果在创建或打开文件时，联系不上名字节点，往往导致操作失败。
- 成功打开文件后，名字节点发生故障，这种情况比较复杂，如果名字节点能很快恢复工作，对于客户端来说，就像名字节点没有发生故障一样，否则，会导致后续的写数据操作失败。
- 在建立数据流管道的过程中，数据节点出现问题将导致管道建立失败，这时，客户端会放弃刚分配的数据块（使用名字节点提供的 abandonBlock() 远程方法），并重新申请数据块。
- 写数据的过程中，某些数据节点如果发生故障，这时，需要使用数据节点提供给客户端的 ClientDatanodeProtocol.recoverBlock() 远程方法，通知其他数据节点进行数据块恢复。
- 执行读操作的客户端如果出现问题，对系统并不会造成太大的影响，但如果在写数据的过程中，客户端崩溃，已经打开的文件需要名字节点，或重启以后的客户端触发名字节点执行租约恢复（ClientProtocol 中的 recoverLease() 方法），并关闭文件。

本节将通过 DFSClient.create() 创建一个文件，写入多个数据块的数据后关闭文件为主线，研究这个客户端上最复杂的流程的实现。

### 9.3.1　写数据前的准备：创建文件

DFSClient.create() 用于创建一个空文件，它返回一个输出流对象，create() 有 7 种不同的重载形式，但最终都调用了下面的这个方法：

```
public OutputStream create(String src, FsPermission permission,
      boolean overwrite, boolean createParent, short replication,
      long blockSize, Progressable progress, int buffersize)
      throws IOException {
   checkOpen(); //检查客户端是否已经打开
   if (permission == null) {
      permission = FsPermission.getDefault();
   }
   FsPermission masked =
         permission.applyUMask(FsPermission.getUMask(conf));
   OutputStream result = new DFSOutputStream(src, masked, overwrite,
         createParent, replication, blockSize, progress, buffersize,
         conf.getInt("io.bytes.per.checksum", 512));
   leasechecker.put(src, result);
   return result;
}
```

如同 DFSClient.open() 一样，create() 也将主要的准备工作交给了 DFSOutputStream 的构造方法，但比 open() 方法复杂的地方在于，create() 方法需要新文件的一些属性，如权限、是否覆盖原文件等；另外，成功创建输出流后，还需要将创建文件和输出流的信息，放入客户端的租约检查器对象 leasechecker 中。

DFSOutputStream 的构造过程略微复杂，同时需要注意的是，通过 create() 和 append() 创建输出流，会使用不同的 DFSOutputStream 构造函数，如图 9-9 所示。

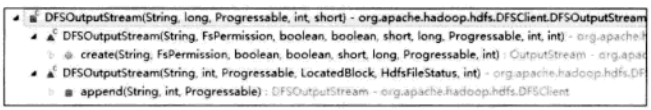



图 9-9　DFSOutputStream 构造函数的调用关系

create() 方法使用的输出流构造函数比较简单，先通过 this() 调用私有的构造函数，初始化一些成员变量。接下来的 computePacketChunkSize() 方法会计算发往数据节点的数据包能包含多少个校验块，以及包的长度。一般来说，传入该方法的 writePacketSize 值为 64K，即数据包最大能达到 64K 字节。输出流和名字节点的第一次交互，即通过远程方法 ClientProtocol.create() 在名字节点上创建一个新的处于构建状态的文件。接下来，启动输出流对象的数据发送器。代码如下：

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

```
        computePacketChunkSize(writePacketSize, bytesPerChecksum);
        try {
            namenode.create(src, masked, clientName, overwrite,
                  createParent, replication, blockSize);
        } catch (RemoteException re) {
            throw re.unwrapRemoteException(……);
        }
        streamer.start();
    }

    private DFSOutputStream(String src, long blockSize,
          Progressable progress, int bytesPerChecksum, short replication)
          throws IOException {
        // 调用父类 FSOutputSummer 的构造函数，提供校验方法、校验块大小等参数
        super(new CRC32(), bytesPerChecksum, 4);
        this.src = src; // 目标文件名
        this.blockSize = blockSize; // 数据块大小
        this.blockReplication = replication; // 数据块副本系数
        ……
        if (bytesPerChecksum < 1 || blockSize % bytesPerChecksum != 0) {
            throw new IOException(……);
        }
        checksum = DataChecksum.newDataChecksum(
              DataChecksum.CHECKSUM_CRC32, bytesPerChecksum);
    }
```

通过 append() 方法打开输出流时，使用的 DFSOutputStream 构造函数就复杂了，难点在于：首先，由于是在已有数据块上追加数据，所以往往需要处理不完整的数据校验块，即输出数据的最开始部分不能构成一个完整的校验块，需要单独计算校验信息；其次，创建输出流对象时，就必须和数据节点联系，建立数据流管道，也就必须处理在建立数据流管道过程中可能出现的问题。限于篇幅，就不再详细介绍 append() 打开文件的细节了。

### 9.3.2 写数据：数据流管道的建立

往数据节点写数据前，需要分配数据块并建立数据流管道。这个过程由 DataStreamer.run() 触发，并由 DFSOutputStream 的成员函数 nextBlockOutputStream() 执行，顾名思义，该方法会建立到下一个数据块的输出流。由于一些比较复杂的过程实现在其他的成员函数里，这个方法比较简单，它通过 locateFollowingBlock() 分配一个新的数据块，然后调用 createBlockOutputStream() 创建数据流管道。如果创建管道失败，调用 ClientProtocol 的 abandonBlock() 方法放弃已申请的数据块。同时，将出现错误的数据节点放入成员变量 excludedNodes 中。然后，就可以重试，即向名字节点申请一个新的数据块。

在 nextBlockOutputStream() 中，第一次分析到写流程中出现的故障，即创建数据流管道失败，这里的故障恢复采取的措施是：放弃旧的数据块，重新申请新的数据块，并在发送新申请时将故障节点排除在外，以避免再次访问到故障节点。相关代码如下：

```
private DatanodeInfo[] nextBlockOutputStream(String client)
      throws IOException {
   LocatedBlock lb = null;
   boolean retry = false;
   DatanodeInfo[] nodes;
   int count = conf.getInt("dfs.client.block.write.retries", 3);
   boolean success;
   do {
      ..... //定义了一些方法工作时需要的变量
      DatanodeInfo[] excluded
            = excludedNodes.toArray(new DatanodeInfo[0]);
      lb = locateFollowingBlock(startTime,
            excluded.length > 0 ? excluded : null);
      block = lb.getBlock(); //数据块信息
      accessToken = lb.getBlockToken();
      nodes = lb.getLocations();//数据节点信息

      success = createBlockOutputStream(nodes, clientName, false);

      if (!success) {
         .....
         //放弃已申请的数据块
         namenode.abandonBlock(block, src, clientName);

         if (errorIndex < nodes.length) {
            .....
            //添加错误节点到 excludedNodes 中
            excludedNodes.add(nodes[errorIndex]);
         }

         retry = true;
      }
   } while (retry && --count >= 0);

   if (!success) { //不成功，抛异常
      throw new IOException("Unable to create new block.");
   }
   return nodes;
}
```

申请数据块使用的是 locateFollowingBlock() 方法，这个方法的核心是调用 NameNode 上的 addBlock() 远程方法，并对异常做一些处理。首先是不支持 HDFS-630[⊖] 的情况，即 addBlock() 方法不支持 excludedNodes 参数，这时使用原有的两参数 ClientProtocol .addBlock() 方法即可。

另一个需要特殊处理的异常是 NotReplicatedYetException，申请新数据块时，如果名字节点上该文件的倒数第二个数据块副本数还没有达到系统最小副本数，就会抛出该异常。出现这种情况的主要原因是：HDFS 系统繁忙，导致数据节点的提交数据块请求没有得到及时

⊖ 关于 HDFS-630 的更多信息，请参考 https://issues.apache.org/jira/browse/HDFS-630。

处理；另一个可能的原因是客户端写数据太快，在倒数第二个数据块的提交还没处理完时，就已经写满了另一个数据块。在这两种情况下，客户端都会等待一段时间，再进行重试。代码如下：

```
private LocatedBlock locateFollowingBlock(long start,
      DatanodeInfo[] excludedNodes) throws IOException {
   int retries =
conf.getInt("dfs.client.block.write.locateFollowingBlock.retries", 5);
   ......
   while (true) {
      while (true) {
         try {
            if (serverSupportsHdfs630) {
               return namenode.addBlock(src, clientName, excludedNodes);
            } else {
               return namenode.addBlock(src, clientName);
            }
         } catch (RemoteException e) {
            if (e.getMessage().startsWith(……)) {
               ……
               // 处理不支持 HDFS-630 的情况
               serverSupportsHdfs630 = false;
               continue;
            }

            if (NotReplicatedYetException.……) {
               …… // 处理 NotReplicatedYetException 的情况
               Thread.sleep(sleeptime); // 等待一段时间
               sleeptime *= 2;
            }
            ……
         }
      }
   }
}

private boolean createBlockOutputStream(DatanodeInfo[] nodes,
      String client, boolean recoveryFlag) {
   short pipelineStatus = (short) DataTransferProtocol.OP_STATUS_SUCCESS;
   String firstBadLink = "";
   ……
   boolean result = false;
   try {
      ……
      InetSocketAddress target
            = NetUtils.createSocketAddr(nodes[0].getName());
      s = socketFactory.createSocket();
      timeoutValue = 3000 * nodes.length + socketTimeout;
      NetUtils.connect(s, target, timeoutValue); // 连接到第一个数据节点
      ……
```

```
        // 到第一个数据节点的输出流和输入流
        DataOutputStream out = new DataOutputStream(
              new BufferedOutputStream(NetUtils.getOutputStream(s,
                    writeTimeout), DataNode.SMALL_BUFFER_SIZE));
        blockReplyStream = new DataInputStream(NetUtils.getInputStream(s));

        // 往数据节点写操作码为 80 的请求
        out.writeShort(DataTransferProtocol.DATA_TRANSFER_VERSION);
        out.write(DataTransferProtocol.OP_WRITE_BLOCK);
        …… // 请求的其他字段
        for (int i = 1; i < nodes.length; i++) {
           nodes[i].write(out); // 管道成员
        }
        accessToken.write(out);
        checksum.writeHeader(out);
        out.flush();

        // 等待连接上的应答
        pipelineStatus = blockReplyStream.readShort();
        firstBadLink = Text.readString(blockReplyStream);
        if (pipelineStatus != DataTransferProtocol.OP_STATUS_SUCCESS) {
           …… // 抛异常
        }

        blockStream = out;
        result = true; // 成功
     } catch (IOException ie) {
        …… // 找到错误的数据节点，并将错误的节点下标保存在成员变量 errorIndex 中
        result = false;
     } finally {
        ……// 失败时，关闭 Socket 连接
     }
     return result;
  }
```

在建立数据流管道方面，客户端的实现比数据节点的实现简单。它只需要往第一个数据节点写入请求即可。所以，createBlockOutputStream() 建立到数据节点的 TCP 连接后，写入数据节点流式接口的 80 请求包，并等待应答。成功建立数据流管道的应答是 DataTransferProtocol.OP_STATUS_SUCCESS，接收到这个应答后，方法就可以返回了。

出现错误时，需要把错误节点的下标找出来。由于出错数据节点的上游节点会将该节点的名字通过流式接口返回，createBlockOutputStream () 读入该信息并保存在 firstBadLink 中，然后查找数据流管道的成员，即可获得该节点的下标。在接下来的处理中，该节点会被添加到故障节点列表 excludedNodes 中。

---

**注意**　建立数据流管道是向数据节点写数据的第一步，在 DFSClient 中，DFSOutputStream 的 nextBlockOutputStream()、locateFollowingBlock() 和 createBlockOutputStream() 等成员函数，用于分配数据块并建立数据流管道。

---

### 9.3.3 写数据：数据包的发送

DFSOutputStream 没有重写 write() 方法，而是复用了父类 FSOutputSummer 中的实现。FSOutputSummer.write() 写入数据后，会调用 write1() 方法，尽可能地通过 writeChecksumChunk() 写出数据。

这里的 write1() 方法和 FSInputChecker.read1() 的作用类似，主要是为了提高处理效率，避免数据拷贝。FSOutputSummer.writeChecksumChunk() 用于写出一个校验块的数据，包括原始数据和校验信息，它会调用 DFSOutputStream 实现的 writeChunk() 方法。

#### 1. DFSOutputStream.writeChunk() 方法

writeChunk() 方法是 FSOutputSummer 中定义的虚方法，它的作用非常明确，提供带校验数据输出流和具体存储系统的适配，对于 DFSOutputStream，这里的具体存储系统就是数据节点。这个方法需要处理两类数据，原始数据存放在缓冲区 b 中，而 checksum 则保存了原始数据的校验信息。

DFSOutputStream.writeChunk() 的入口是一些检查，和输出相关的处理从对 dataQueue 的同步块开始。成员变量 dataQueue 保存了输出流中等待发送的数据包，而 ackQueue 保存的是已经发送，但还没有收到确认的数据包。数据节点在处理完一个数据包，并接收到数据流管道下游对这个包的处理确认后，会往上游发送确认信息。DFSOutputStream 对象收到确认后，才会从 ackQueue 中移除数据包。由于 dataQueue 和 ackQueue 中的数据包都是先进先出的，所以它们的类型是链表，链表元素是 Packet 对象。

Packet 是一个很简单的辅助类，它用于构造数据包，也用于构造数据包的特殊形式——心跳包。Packet 有一些实现细节：如怎么把通过 writeChunk() 方法输出的数据，保存成数据包要求的形式，即校验数据在前，文件数据在后的协议帧；在发送数据时，数据没有填满整个数据包，如何整理数据等，具体实现请参考 Packet 的成员函数 writeData()、writeChecksum() 和 getBuffer()，这里就不再细究了。

writeChunk() 方法代码如下：

```
@Override
protected synchronized void writeChunk(byte[] b, int offset, int len,
      byte[] checksum) throws IOException {
   checkOpen(); // 检查 DFSClient 对象的状态
   isClosed(); // 检查 DFSOutputStream 对象的状态

   int cklen = checksum.length;
   int bytesPerChecksum = this.checksum.getBytesPerChecksum();
   if (len > bytesPerChecksum) { // 输入数据比一个校验块还大
      throw new IOException(……);
   }
   if (checksum.length != this.checksum.getChecksumSize()) {
      throw new IOException(……); // 输入校验数据大小不正确
   }
```

```
    synchronized (dataQueue) {
        while (!closed
                && dataQueue.size() + ackQueue.size() > maxPackets) {
            try {
                dataQueue.wait();
            } catch (InterruptedException e) {
            }
        }
        isClosed();

        // 当前数据包为空，创建数据包
        if (currentPacket == null) {
            currentPacket = new Packet(packetSize,
                    chunksPerPacket, bytesCurBlock);
            ......
        }

        currentPacket.writeChecksum(checksum, 0, cklen); // 写入校验数据
        currentPacket.writeData(b, offset, len); // 写入文件数据
        currentPacket.numChunks++;
        bytesCurBlock += len;

        // 当前数据包已满
        if (currentPacket.numChunks == currentPacket.maxChunks
                || bytesCurBlock == blockSize) {
            ......
            if (bytesCurBlock == blockSize) {
                currentPacket.lastPacketInBlock = true;
                bytesCurBlock = 0;
                lastFlushOffset = 0;
            }
            enqueueCurrentPacket(); // 将数据包放入 dataQueue 队列中
            ......  // 处理追加数据的特殊情况
            int psize = Math.min((int) (blockSize - bytesCurBlock),
                    writePacketSize);
            computePacketChunkSize(psize, bytesPerChecksum);
        }
    }
}

private synchronized void enqueueCurrentPacket() {
    synchronized (dataQueue) {
        ..... // 参数检查，保证 currentPacket 非空
        dataQueue.addLast(currentPacket); // 数据包加入队列
        dataQueue.notifyAll();
        lastQueuedSeqno = currentPacket.seqno;
        currentPacket = null; // 设置当前数据包为空
    }
}
```

继续分析同步块中的代码，显然，队列 dataQueue 和 ackQueue 中若有大量未处理的包，

那么，等待直到数据节点处理完一些数据包，再进行后续的操作。接下来，如果当前工作数据包 currentPacket 为空，则分配一个新的数据包。

往数据包中写入数据使用的是 Packet 提供的方法，校验数据和文件数据分别通过 writeChecksum() 和 writeData() 写入。如果数据包已写满，writeChunk() 还需要使用 enqueueCurrentPacket() 将数据包放入队列 dataQueue 中，并重新计算下一个数据包能容纳的数据大小，以免越过数据块的边界。

在 DFSOutputStream 对象上调用 write() 方法，返回时，数据已经写入数据包，但往往还没有发往数据节点。

在 enqueueCurrentPacket() 方法中，调用了对象 dataQueue 的 notifyAll() 方法，显然有线程在通过 wait() 方法等待这个通知。这个线程就是 DFSOutputStream 对象拥有的 DataStreamer 线程，在构造输出流时启动的数据发送器 DataStreamer，它的 run() 方法在单独的线程中执行。

**2. 数据发送器的实现原理**

DataStreamer.run() 是一个“大”方法，它的主要作用是将 dataQueue 中的数据包发往数据节点，主循环中包含如下主要处理步骤：

1）发送数据过程中如果出现错误，关闭应答器，处理数据节点故障。

2）等待一个数据包。

3）执行可能的文件数据块分配并建立数据流管道。

4）发送数据包。

5）处理可能的数据块写满的情况。

下面按照上述顺序分析 DataStreamer.run()。这个方法有点特殊，它的错误处理在循环的最前面，当标志位 hasError 为 true，即写数据过程出现错误时，需要关闭应答器。同时，循环中的逻辑会调用 processDatanodeError() 处理可能的数据节点错误。

用于等待一个数据包的 while 语句条件比较复杂，它必须检查 DFSClient、DFSOutputStream 对象的状态（对 closed、hasError 和 clientRunning 等状态变量的判断），也要等待队列 dataQueue 中的数据，同时要在需要发送心跳的时候发送心跳包（blockStream 不为空的情况下），还要考虑 processDatanodeError() 的处理结果（doSleep 如果大于 0，循环逻辑相当于等待一段时间后再重试 processDatanodeError() 方法）。

离开 while 循环时，如果还有错误或者 DataStreamer/DFSClient 已经关闭，通过 continue 重新执行循环或退出循环（当 DataStreamer/DFSClient 关闭时）。否则，当 dataQueue 队列为空时，DataStreamer 会往数据节点发送一个心跳包，队列非空时，dataQueue 队列中的第一个包就是接下来要发送的数据包。

DataStreamer.run() 的代码如下：

```
public void run() {
   long lastPacket = 0;
```

```
while (!closed && clientRunning) {
   // 如果应答处理器出现错误，关闭应答处理器
   if (hasError && response != null) {
      try {
         response.close(); // 关闭应答处理器
         response.join(); // 等待线程退出
         response = null;
      } catch (InterruptedException e) {
      }
   }

   Packet one = null;
   synchronized (dataQueue) {
      // 处理可能的数据节点错误
      boolean doSleep = processDatanodeError(hasError, false);

      // 等待一个数据包
      long now = System.currentTimeMillis();
      while ((!closed && !hasError && clientRunning
            && dataQueue.size() == 0 && (blockStream == null ||
            (blockStream != null
                  && now - lastPacket < timeoutValue / 2)))
            || doSleep) {
         long timeout = timeoutValue / 2  - (now - lastPacket);
         timeout = timeout <= 0 ? 1000 : timeout;

         try {
            dataQueue.wait(timeout);
            now = System.currentTimeMillis();
         } catch (InterruptedException e) {
         }
         doSleep = false;
      }
      if (closed || hasError || !clientRunning) {
         continue; // 出错或需要退出，continue，重新执行循环或离开循环
      }

      try {
         // 获得一个用于发送的数据包
         if (dataQueue.isEmpty()) {
            one = new Packet(); // 心跳包
         } else {
            one = dataQueue.getFirst(); // 普通数据包
         }

         long offsetInBlock = one.offsetInBlock;

         // 需要一个新的数据块
         if (blockStream == null) {
            nodes = nextBlockOutputStream(src);
```

```
                response = new ResponseProcessor(nodes);
                response.start();
            }

            if (offsetInBlock >= blockSize) {
                throw new IOException(……);
            }

            ByteBuffer buf = one.getBuffer();

            // 将数据包从队列 dataQueue 转移到队列 ackQueue 中
            if (!one.isHeartbeatPacket()) {
                dataQueue.removeFirst();
                dataQueue.notifyAll();
                synchronized (ackQueue) {
                    ackQueue.addLast(one);
                    ackQueue.notifyAll();
                }
            }

            // 写出数据包
            blockStream.write(buf.array(), buf.position(),
                    buf.remaining());

            if (one.lastPacketInBlock) {
                blockStream.writeInt(0); // 通知数据节点，数据已经发送完毕
            }
            blockStream.flush();
            ……
        } catch (Throwable e) {
            ……
            hasError = true;
        }
    }

    if (closed || hasError || !clientRunning) {
        continue; // 出错，continue，转出错代码
    }

    // 数据块已经写满
    if (one.lastPacketInBlock) {
        synchronized (ackQueue) {
            while (!hasError && ackQueue.size() != 0
                    && clientRunning) {
                try {
                    ackQueue.wait(); // 等待所有的应答
                } catch (InterruptedException e) {
                }
            }
        }
```

```
                response.close(); // 关闭应答处理器
                try {
                    response.join();
                    response = null;
                } catch (InterruptedException e) {
                }

                if (closed || hasError || !clientRunning) {
                    continue;  // 有错误，继续循环，由前面的代码处理错误
                }

                synchronized (dataQueue) {
                    // 关闭流并释放资源
                    IOUtils.cleanup(LOG, blockStream, blockReplyStream);
                    nodes = null;
                    response = null;
                    blockStream = null;
                    blockReplyStream = null;
                }
            }
            ......
        }
    }
```

如果 DataStreamer 的成员变量 blockStream 为空，表明还没有和数据节点建立联系，使用上一小节分析过的 nextBlockOutputStream() 方法，建立数据流管道，并启动流的应答处理器 ResponseProcessor。接下来的发送数据包包括如下主要步骤：

1）从 Packet 对象中获取待发送数据，Packet.getBuffer() 会对数据进行整理，保证发送的检验数据和文件数据都保存在缓冲区 buf 中。

2）接下来，数据包会从 dataQueue 队列移动到 ackQueue 队列，即等待应答队列中。

3）最后，通过和数据节点的 TCP 连接 blockStream，发送数据并刷新流，如果待发送的数据包是最后一个数据包，还需要写出一个特殊标记，通知数据节点，数据块发送完毕。

数据节点在接收到一个数据块后，需要执行一些清理动作，如在文件数据集 FSDataset 对象上提交数据块，通知名字节点等。相应的，客户端上也有一些收尾工作，如等待所有的数据包应答，关闭应答处理器和释放相关资源。当然，数据虽然发送完毕，但管道上的数据节点处理仍可能出错，出错的故障恢复由 run() 方法开始的 processDatanodeError() 处理。

DataStreamer.run() 虽然冗长，但实现逻辑还算清晰。

### 3. 应答处理器的实现原理

DFSOutputStream 写数据的最后一个主要过程是数据包应答的接收和处理。客户端接收到数据包应答，表明管道流上所有的数据节点已经接收到对应的数据包，可以删除 ackQueue 队列中的对应项。这个任务实现在应答处理器 ResponseProcessor 中，和数据发送器 DataStreamer 一样，它也拥有自己的线程。所以，DFSOutputStream 写数据时需要两个独立的线程，分别用于发送数据和接收应答。这一点，和数据节点写数据处理中使用的

BlockReceiver 线程和 PacketResponder 线程，采用了同样的策略。

ResponseProcessor.run() 代码如下：

```
public void run() {
   ……
   PipelineAck ack = new PipelineAck();

   while (!closed && clientRunning && !lastPacketInBlock) {
      try {
         ack.readFields(blockReplyStream); // 读入 ack 对象
         ……
         // 处理应答
         for (int i = ack.getNumOfReplies() - 1;
               i >= 0 && clientRunning; i--) {
            short reply = ack.getReply(i);
            if (reply != DataTransferProtocol.OP_STATUS_SUCCESS) {
               errorIndex = i; // 应答中有错误
               throw new IOException(……);
            }
         }

         long seqno = ack.getSeqno();
         ……// 忽略心跳包
         Packet one = null;
         synchronized (ackQueue) {
            one = ackQueue.getFirst();
         }
         …… // 检查序号
         lastPacketInBlock = one.lastPacketInBlock;

         synchronized (ackQueue) {
            assert ack.getSeqno() == lastAckedSeqno + 1;
            lastAckedSeqno = ack.getSeqno();
            ackQueue.removeFirst(); // 删除数据包
            ackQueue.notifyAll();
         }
      } catch (Exception e) {
         if (!closed) {
            hasError = true; // 设置错误标志
            ……
         }
      }

      synchronized (dataQueue) {
         dataQueue.notifyAll();
      }
         synchronized (ackQueue) {
         ackQueue.notifyAll();
      }
   }
}
```

和 DataStreamer.run() 比，ResponseProcessor.run() 方法很简单，它先读入 PipelineAck 对象，该对象包含了多个数据节点对数据包的处理应答，如果某数据节点的应答码不是 DataTransferProtocol.OP_STATUS_SUCCESS，记录位置信息到 errorIndex 中，表明数据流管道的这个节点出错了。出错会抛出异常，在异常处理中会设置错误标志 hasError，并触发 DataStreamer.run() 循环中对 processDatanodeError() 方法的调用。

如果各数据节点都给出成功的应答，这时数据包就可以从 ackQueue 中移除。DFSOutputStream 的输出数据通过 writeChunk() 进入数据包，并进一步到达 dataQueue 队列；在 DataStreamer.run() 中，发往数据节点的数据包会移动到 ackQueue；最后，当 ResponseProcessor.run() 接收到管道流发送回来的成功应答后，数据和数据包一起从 ackQueue 中删除，这就是输出数据在客户端经历的几个主要阶段。

### 9.3.4　写数据：数据流管道出错处理

这一节分析数据流管道上出现错误的故障处理过程，它实现在 processDatanodeError() 中。该方法有一定的长度，但逻辑不算复杂，在研究数据节点的时候我们已经了解了数据节点上的数据块恢复过程，这个过程涉及多个数据节点间的通信，同时还需要名字节点的配合，是 HDFS 系统中比较复杂的流程之一。对于输出流 DFSOutputStream 来说，它只需要和数据节点打交道，所以，它的实现也就相对简单。

processDatanodeError() 方法可以划分为三个部分：

1）错误状态确认；

2）数据块恢复准备并调用数据节点的远程方法；

3）根据恢复结果做进一步的处理。

由于数据流管道上已经发现错误，所以和管道相关的资源，如应答处理器（在 DataStreamer.run() 中被关闭），到原有数据流管道的连接（blockStream 和 blockReplyStream）等都应该关闭，队列 ackQueue 中的数据包需要移动到 dataQueue 中，供数据块恢复使用。

接下来，需要将出现故障的数据节点从数据流管道成员列表中移除，然后选择一个恢复的主数据节点，通过该节点的 ClientDatanodeProtocol 远程接口的 recoverBlock() 方法，启动数据块恢复过程。processDatanodeError() 方法代码如下：

```
private boolean processDatanodeError(boolean hasError, boolean isAppend) {
   if (!hasError) {
      return false;
   }
   if (response != null) {
      ......
      return true;
   }
   ......
   if (blockStream != null) {
```

```
        IOUtils.cleanup(LOG, blockStream, blockReplyStream);
    }
    blockStream = null;
    blockReplyStream = null;

    // 将 ackQueue 中的数据包移动到 dataQueue 中，准备重新发送
    synchronized (ackQueue) {
        dataQueue.addAll(0, ackQueue);
        ackQueue.clear();
    }

    boolean success = false;
    while (!success && clientRunning) {
        DatanodeInfo[] newnodes = null;
        if (nodes == null) {// 数据节点列表为空，恢复失败
            ......
            closed = true;
            if (streamer != null)
                streamer.close();
            return false;
        }
        ......

        // 通过 append() 打开文件时，DFSOutputStream 构造函数会调用这个方法
        // 这个时候不需要删除任何数据节点
        if (errorIndex < 0) {
            newnodes = nodes;
        } else {
            if (nodes.length <= 1) {
                // 没有可以使用的数据节点，出错，处理逻辑和 if (nodes == null) 类似
                ......
                return false;
            }
            ......
            // 移除故障数据节点
            newnodes = new DatanodeInfo[nodes.length - 1];
            System.arraycopy(nodes, 0, newnodes, 0, errorIndex);
            System.arraycopy(nodes, errorIndex + 1, newnodes,
                    errorIndex, newnodes.length - errorIndex);
        }

        LocatedBlock newBlock = null;
        ClientDatanodeProtocol primary = null;
        DatanodeInfo primaryNode = null;
        try {
            // 选择恢复的主数据节点，调用 ClientDatanodeProtocol.recoverBlock()
            primaryNode = Collections.min(Arrays.asList(newnodes));
            primary = createClientDatanodeProtocolProxy(primaryNode,
                    conf, block, accessToken, socketTimeout);
            newBlock = primary.recoverBlock(block, isAppend, newnodes);
```

```
        } catch (IOException e) {
            ......
            recoveryErrorCount++;
            if (recoveryErrorCount > maxRecoveryErrorCount) {
                if (nodes.length > 1) {
                    // 处理主数据节点失败的情况，为重试做准备
                    ......
                }
                ...... // 没有可以使用的数据节点，数据块恢复失败，抛异常
            }
            ......
            return true; // 让 DataStreamer.run() 等待一段时间后进行重试
        } finally {
            RPC.stopProxy(primary);
        }
        recoveryErrorCount = 0; // 成功地进行了数据块恢复

        // 使用新的数据块版本号建立新的数据流管道
        block = newBlock.getBlock();
        accessToken = newBlock.getBlockToken();
        nodes = newBlock.getLocations();

        this.hasError = false;
        lastException = null;
        errorIndex = 0;
        success = createBlockOutputStream(nodes, clientName, true);
    }

    response = new ResponseProcessor(nodes); // 重新建立应答处理器
    response.start();
    return false;// 成功进行了数据块恢复，不需要 sleep
}
```

数据块恢复有两种结果：如果出错，但还有可供恢复的节点，processDatanodeError() 通过返回 true，让 DataStreamer.run() 等待一段时间后，再次调用 processDatanodeError()，并选择新的主数据节点，再次启动一次数据块恢复过程；如果已经没有任何可用的数据节点，设置 closed 标志位为 false，设置输出流的成员变量 lastException，然后关闭数据发送器，这也意味着写数据失败，输出流不再可用了。

如果成功进行了数据块恢复，可以继续写数据，这时通过 createBlockOutputStream() 重新创建数据流管道，并重新建立应答处理器 ResponseProcessor。在方法返回后，DataStreamer.run() 又可以继续往剩余的数据节点发送数据了。发送的数据里包括 processDatanodeError() 从 ackQueue 中转移到 dataQueue 的那些数据。

客户端通过数据节点提供的数据块恢复，保证了在数据流管道的某一个节点出现故障后，还能够剔除问题节点，继续写文件数据，降低失效数据节点的影响。

### 9.3.5 写数据：租约更新

第 8 章给出了租约的定义，对于 HDFS 来说，租约是名字节点给予租约持有者（一般是客户端）在规定时间期限内一定权限（写文件）的合同。名字节点通过租约管理器管理租约，客户端需要定期更新租约，否则，在租约过期后，名字节点通过租约恢复机制关闭文件。所以，客户端打开文件后，需要有一定的机制维护和名字节点的契约关系。LeaseChecker 实现了客户端需要的租约管理功能。

在创建 DFSOutputStream 对象时，构造函数调用了 LeaseChecker 的 put() 方法，将打开文件名和流对象作为一个租约记录，放入 DFSClient 的租约管理器对象 leasechecker 中，在随后的写数据过程中，就再也没有和租约相关的操作了。代码如下：

```
public OutputStream create(……)throws IOException {
   ……
   OutputStream result = new DFSOutputStream(……);
   leasechecker.put(src, result);
   return result;
}
```

租约检查器对象负责后面的租约更新，LeaseChecker.put() 会将该租约记录保存在它的成员变量 pendingCreates 中，如果租约检查器线程还没有启动，会启动检查器线程，如下：

```
class LeaseChecker implements Runnable {
   // 保存了所有的租约记录，即打开文件的文件名和 DFSOutputStream 对象
   private final SortedMap<String, OutputStream> pendingCreates
         = new TreeMap<String, OutputStream>();
   ……
   synchronized void put(String src, OutputStream out) {
      if (clientRunning) {
         if (daemon == null) {
            daemon = new Daemon(this);
            daemon.start(); // 启动检查器线程
         }
         pendingCreates.put(src, out);
      }
   }
   ……
}
```

LeaseChecker 对象拥有的线程工作很简单，当当前时间和上次租约更新时间的间隔大于租约软超时时间的一半时（软超时时间为 1 分钟，不可配置），通过 renew() 方法调用远程方法 ClientProtocol .renewLease()，更新租约。

LeaseChecker.run() 的实现非常简单，对异常的处理也非常严格，保证了租约检查器能够可靠地工作，避免 LeaseChecker 出现问题，长时间未更新租约，导致名字节点错误地启动租约恢复过程。代码如下：

```
public void run() {
   long lastRenewed = 0;
```

```
    while (clientRunning && !Thread.interrupted()) {
      if (System.currentTimeMillis() - lastRenewed
          > (LEASE_SOFTLIMIT_PERIOD / 2)) {
        try {
          renew();
          lastRenewed = System.currentTimeMillis();
        } catch (IOException ie) {
          ......
        }
      }

      try {
        Thread.sleep(1000);
      } catch (InterruptedException ie) {
        ......
        return;
      }
    }
  }

  private void renew() throws IOException {
    ...... // 如果 pendingCreates 中不包含任何记录，返回
    namenode.renewLease(clientName);
  }
```

ClientProtocol 中还提供了一个客户端发起名字节点租约恢复的方法 recoverLease()，在输出流的实现中，不需要用到这个方法。它提供了一个客户端干预名字节点租约的手段，一个典型的应用场景是：HDFS 应用因为错误重启后，通过该方法对上一次运行打开的文件进行租约恢复，以关闭文件。

### 9.3.6　写数据：DFSOutputStream.sync() 的作用

名字节点的编辑日志保存，使用了日志输出流的 EditLogFileOutputStream 的 flushAndSync() 方法，将日志记录从内存缓冲区中输出并持久化到磁盘中。其中，flush 强制写出所有已缓冲的输出，而 sync 则将输出同步到磁盘，保证了数据的持久化。

DFSOutputStream.sync() 的功能和上述 flush、sync 的含义都不一样，它保证了写入 DFSOutputStream 中的数据都已经输出到数据节点，但不保证数据节点持久化了这些数据，也就是说，DFSOutputStream 的 sync() 方法实现的能力介于上述 flush 和 sync 功能之间。它的代码如下：

```
  public void sync() throws IOException {
    checkOpen();
    if (closed) {
      throw new IOException("DFSOutputStream is closed");
    }
    try {
      long toWaitFor;
```

```
        synchronized (this) {
            // 记录 blockOffset，由于 sync() 调用 flushBuffer()，可能会产生一个带有
            // 不完整校验块的数据包。使用 true 调用 flushBuffer()，该不完整校验块的数
            // 据虽然被强制写出，但还保留在缓冲区中，下次组包的时候还会使用
            long saveOffset = bytesCurBlock;
            Packet oldCurrentPacket = currentPacket;
            flushBuffer(true);

            if (lastFlushOffset != bytesCurBlock) {
                assert bytesCurBlock > lastFlushOffset;
                lastFlushOffset = bytesCurBlock;
                enqueueCurrentPacket();
            } else {
                if (oldCurrentPacket == null && currentPacket != null) {
                    // 数据已经写出，丢弃 flushBuffer() 产生的数据包
                    currentSeqno--;
                }
                currentPacket = null;
            }
            bytesCurBlock = saveOffset; // 恢复状态
            toWaitFor = lastQueuedSeqno;
        }
        waitForAckedSeqno(toWaitFor);

        // 自上次 sync() 操作后输出流新分配了数据块，需要持久化名字节点上的信息
        boolean willPersist;
        synchronized (this) {
            willPersist = persistBlocks && !closed;
            persistBlocks = false;
        }
        if (willPersist) {
            try {
                namenode.fsync(src, clientName);
            } catch (IOException ioe) {
                ......
                throw ioe;
            }
        }
    } catch (IOException e) {
        ......
    }
}
```

按照上面的讨论，sync() 返回时，调用该方法时位于流缓冲区中的数据都会输出到数据节点，这里强调了“调用该方法时”，如果后续有其他线程往流中写入数据，sync() 不需要理会它们写入的数据是否到达数据节点，返回时，只能保证调用方法时缓冲区中的数据成功到达数据节点。在检查对象的状态后，sync() 首先要将 FSOutputSummer 缓冲区中的数据通过 flushBuffer() 强制写出。

这是 sync() 实现中的一个难点，调用 flushBuffer() 一般会产生一个带着部分校验块的数据包，即包中的文件数据不是结束于某一个校验块边界。sync() 方法使用了 true 调用 flushBuffer()，即强制写出数据后不清除部分校验块里的这些数据，下次组包时还会继续使

用它们。所以，当 lastFlushOffset，即上次调用 sync() 时写出数据的位置和数据的当前位置不一致时，才需要将数据包通过 enqueueCurrentPacket()，放入发送队列，否则，这些数据已经被上一次 sync() 调用写出。这时，方法需要丢弃这次 flushBuffer() 产生的数据包。显然，这又是一个基于流的数据输出和基于检验块的数据节点流式接口带来的适配问题。

接下来的代码就简单了，通过 toWaitFor() 等待收到“调用该方法时”发送的最后一个数据包的应答，即可判断数据已经都成功发送到数据节点上，紧接着调用名字节点的 fsync()，完成名字节点上的元数据输出，DFSOutputStream.sync() 的工作就完成了。

### 9.3.7　关闭输出流

我们已经研究过 DFSOutputStream 的大部分主要方法，现在可以关闭输出流了。和 DFSInputStream.close() 比较，DFSOutputStream.close() 需要执行更多的清理工作。在调用 closeInternal() 后，输出流的 close() 方法会从租约管理器中移除记录，并关闭到数据节点的（可能存在的）Socket 连接。代码如下：

```
public void close() throws IOException {
   if (closed) {
      …… // 已经处于关闭的状态
   }
   closeInternal();
   leasechecker.remove(src);

   if (s != null) {
      s.close();
      s = null;
   }
}

private synchronized void closeInternal() throws IOException {
   …… // 状态检查
   try {
      flushBuffer(); // 写出所有缓冲的数据
      synchronized (dataQueue) {
         if (currentPacket == null && bytesCurBlock != 0) {
            // 没有可用的数据包，创建一个空数据包
            currentPacket = new Packet(packetSize,
                  chunksPerPacket, bytesCurBlock);
         }
         if (currentPacket != null) {
            currentPacket.lastPacketInBlock = true;
         }
      }

      flushInternal(); // 等待所有数据到达数据节点
      isClosed();
      closed = true; // 设置标志位，让相关线程退出
      closeThreads();

      synchronized (dataQueue) {
```

```
            if (blockStream != null) {
                blockStream.writeInt(0); // 通知数据节点数据包发送结束
                IOUtils.cleanup(LOG, blockStream, blockReplyStream);
            }
            …… // 关闭到数据节点的（可能存在的）Socket 连接
            }
            ……
            boolean fileComplete = false;
            while (!fileComplete) {
                fileComplete = namenode.complete(src, clientName);
                if (!fileComplete) {
                    …… // 在重试前等待一段时间
                }
            }
        } finally {
            closed = true;
    }
}
```

DFSOutputStream.close() 将大量的收尾工作交给了 closeInternal()。这个方法的入口调用了 FSOutputSummer.flushBuffer()，将输出流可能缓存在流对象中的数据全部写入数据包中；然后，设置 Packet 对象最后一个数据包标志位，调用 flushInternal() 等待所有数据到达数据节点。flushInternal() 方法的代码如下：

```
private void flushInternal() throws IOException {
    …… // 状态检查
    long toWaitFor;
    synchronized (this) {
        enqueueCurrentPacket();
        toWaitFor = lastQueuedSeqno;
    }
    waitForAckedSeqno(toWaitFor); // 等待，直到收到所有的数据包应答
}
```

使用 waitForAckedSeqno() 方法等待收到所有的数据包应答，这时就可以关闭 DFSOutputStream 对象拥有的数据发送器和应答处理器。在通知数据节点数据包发送结束后，下一个清理动作是关闭到数据节点的输入 / 输出流。

上述工作完成后，通过 ClientProtocol.complete() 关闭名字节点上打开的文件。当文件还有数据块没有达到系统要求的最小副本数时，远程方法 complete() 会返回 false，closeInternal() 会等待一段时间，然后重试，直到操作成功，文件被成功关闭为止。所以，在 closeInternal() 返回时，数据节点和名字节点上和这次写操作相关的资源都被释放（严格地说，是最少系统最小副本数 minReplication 那么多个数据节点资源被释放）。

### 9.3.8 向 HDFS 文件写入数据的流程

回顾输出流实现的主要流程，首先，应用通过 DFSClient 的 create() 方法创建一个新文件，获得一个输出流，输出流构造时，就会通过远程调用 ClientProtocol.create() 在名字节点上创建文件，具体的序列图如图 9-10 所示。

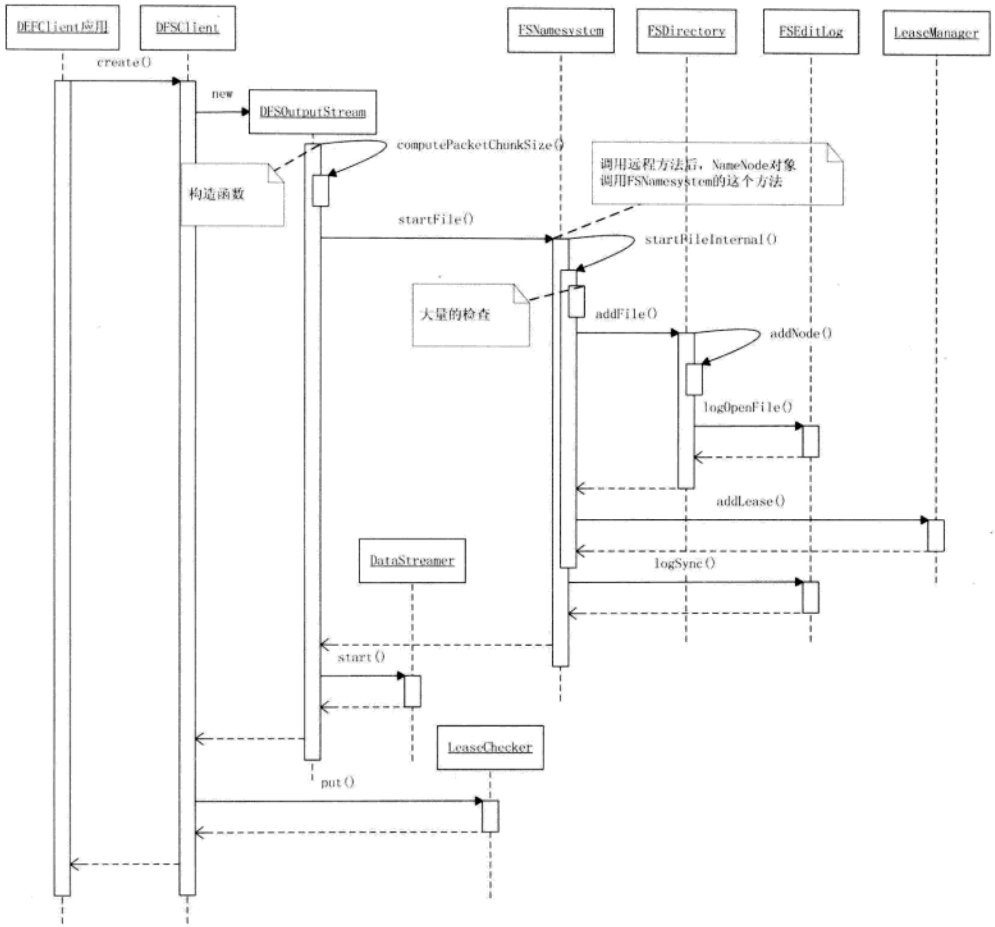



图 9-10　DFSClient.create() 序列图

DFSClient.create() 被调用后，DFSOutputStream 的构造函数就会和名字节点联系，通过 NameNode 对象再到 FSNamesystem.startFile()，最后调用 startFileInternal() 创建文件。startFileInternal() 方法根据输入参数，在执行大量的检查后，使用 FSDirectory.addFile()，最终由 addNode() 在目录树上添加一个 INodeFileUnderConstruction 对象，也就创建了文件。接下来，addFile() 会往日志中写入添加文件的记录，startFileInternal() 则往名字节点的租约管理器中添加记录，startFile() 最后将日志刷出并持久化到磁盘，完成了名字节点上 create() 远程方法的执行。DFSOutputStream 接下来会启动数据发送器 DataStreamer，并在客户端的租约管理器中也添加一条记录。

应用在获取 DFSOutputStream 对象后，就可以通过 write() 输出数据，在调用了 FSOutputSummer 的一系列方法后，执行流到达 DFSOutputStream.writeChunk()，它会将一个校

验块的数据写入到流的 Packet 对象中。DFSOutputStream.write() 的部分时序图如图 9-11 所示。

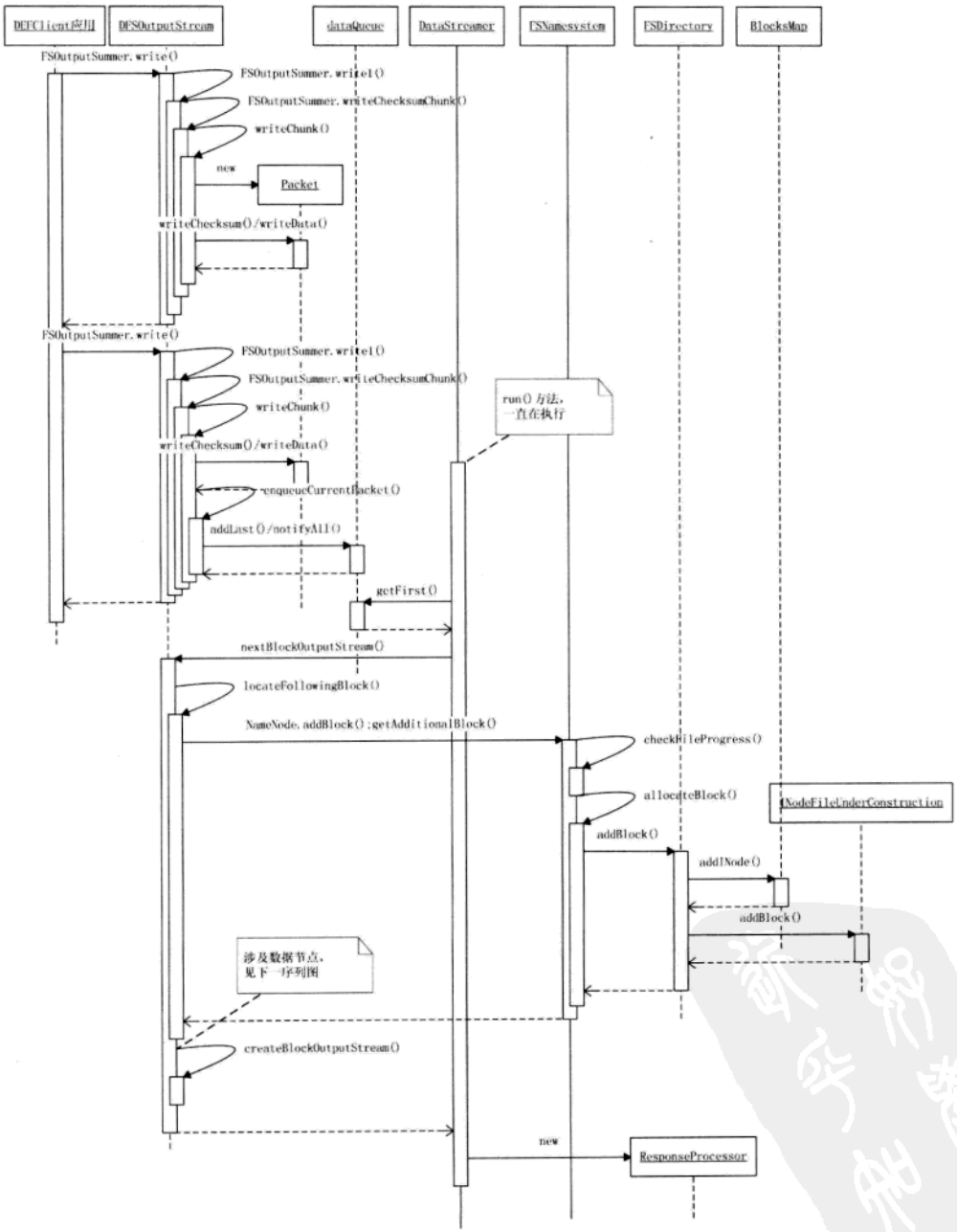



图 9-11 DFSOutputStream.write() 序列图（第一部分）

多次调用 DFSOutputStream.writeChunk() 会写满一个数据包，并将该数据包放入流对象的发送队列 dataQueue 中，等待发往数据节点。接下来的工作由数据发送器负责，在等待到第一个数据包后，DataStreamer.run() 发现到数据节点的数据流管道还没有建立。于是，调用 DFSOutputStream.nextBlockOutputStream()，为文件申请一个新的数据块并建立数据流管道。名字节点的远程方法 addBlock() 在一个打开的文件中追加数据块，具体的逻辑实现在 FSNamesystem.getAdditionalBlock() 方法中，在使用 checkFileProgress() 方法检查文件的状态后，利用 allocateBlock()，最终调用 FSDirectory.addBlock() 为文件分配一个新的数据块。addBlock() 方法有两个关键步骤，分别往数据块映射 BlocksMap 对象和 INodeFileUnderConstruction 对象中添加新的数据块信息。

完成新数据块申请后，nextBlockOutputStream() 接着调用 createBlockOutputStream() 建立数据流管道，如图 9-12（这个方法涉及数据节点，所以放在序列图的第二部分）。

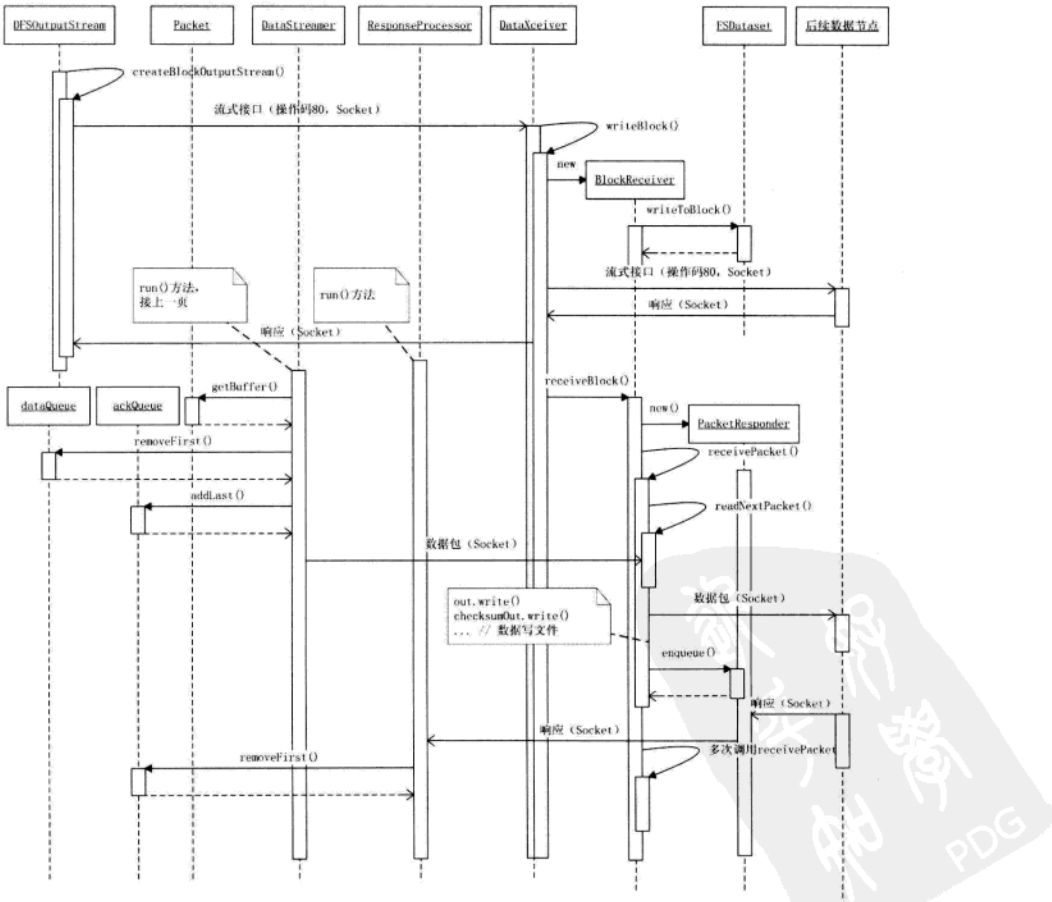



图 9-12　DFSOutputStream.write() 序列图（第二部分）

在接收到流式接口上的 80 请求后，数据节点 DataXceiver 服务器会调用 writeBlock() 方法构造一个 BlockReceiver 对象，并由该对象继续处理请求。BlockReceiver 的构造函数调用 FSDataset.writeToBlock()，获取文件数据集上的数据块文件输出流和校验信息文件输出流，为后续的写操作做准备。如果当前数据节点位于管道流中间，writeBlock() 方法接下来会向下游节点发送 80 请求，在接收到下游的应答后，方法才往上游发送应答，当应答到达 DFSOutputStream 对象时，数据流被成功建立。

接下来，DataStreamer 对象会创建应答处理器 ResponseProcessor（图 9-11），而数据节点的 BlockReceiver.receiveBlock() 方法会创建 PacketResponder，它们的设计目标一样，即对同一 Socket 对象上的输入 / 输出流分开操作，避免相互影响。至此，客户端和数据节点完成写 HDFS 文件数据的准备工作。

DataStreamer.run() 方法继续执行，先整理要发送数据包的数据，接着将数据包从队列 dataQueue 中移动到 ackQueue，然后就可以往数据流管道中写出数据。BlockReceiver.receivePacket() 调用 readNextPacket() 方法，完整地接收一个数据包后，先通过到数据流管道下游的 TCP 连接发送数据，然后往数据块文件和对应的校验信息文件写入文件数据和校验数据。数据包处理结束后，会将响应通过 enqueue() 放入 PacketResponder 对象中，在等待到下游的响应后，该数据包的响应可以往上游发送，一直到达客户端，并由 ResponseProcessor 处理。接收到应答后，客户端的应答处理器就可以从 ackQueue 中移除数据包，通过 write() 写入的数据，顺利发送到（多个）数据节点上。

DataStreamer.run() 调用的 DFSOutputStream.processDatanodeError() 的实现不算复杂，大部分数据节点间的配合和主数据节点与名字节点的配合序列，在第 8 章分析租约恢复时就已经给出，差异只在于发起流程的方式不同而已。

最后给出的流程是 DFSOutputStream.close()，主要关注的是数据节点和名字节点间的配合。如图 9-13 所示。

DFSOutputStream.close() 用于关闭往 HDFS 写数据的输出流。一般情况下，输出缓冲区里还有未发往数据节点的数据，这时，flushInternal() 方法会通过 enqueueCurrentPacket() 将这些数据组包并放入队列 dataQueue 中，并由 DataStreamer 发往数据节点。这个数据包的 tail 字段会被设置为 true，表明它是最后一个数据包。

数据节点的写流程逻辑这个时候往往处于 readNextPacket() 方法中，等待下一个数据包。和一般的数据包处理流程一样，数据节点接收到的这个数据包会发往下游节点，数据和数据的校验则写入磁盘，然后通过 PacketResponder.enqueue()，将工作交给 PacketResponder 线程，由它等待下游的处理结果。调用 enqueue() 时，第二个参数值为 true，表明这是最后一个数据包。获得下游数据节点的响应时，由于这是最后一个数据包，PacketResponder 会做一些特殊的处理，首先调用的是 BlockReceiver.close() 方法。注意，该 close() 方法只关闭到数据节点磁盘的（两个）输出流，并没有关闭对象。接下来，通过 FSDataset.finalizeBlock() 提交数据块；通过 notifyNamenodeReceivedBlock() 确认写数据块操作成功；通过往上游的 TCP 连接发送响应。完成上述操作后，PacketResponder 线程就完成了它的使命，正常结束了。

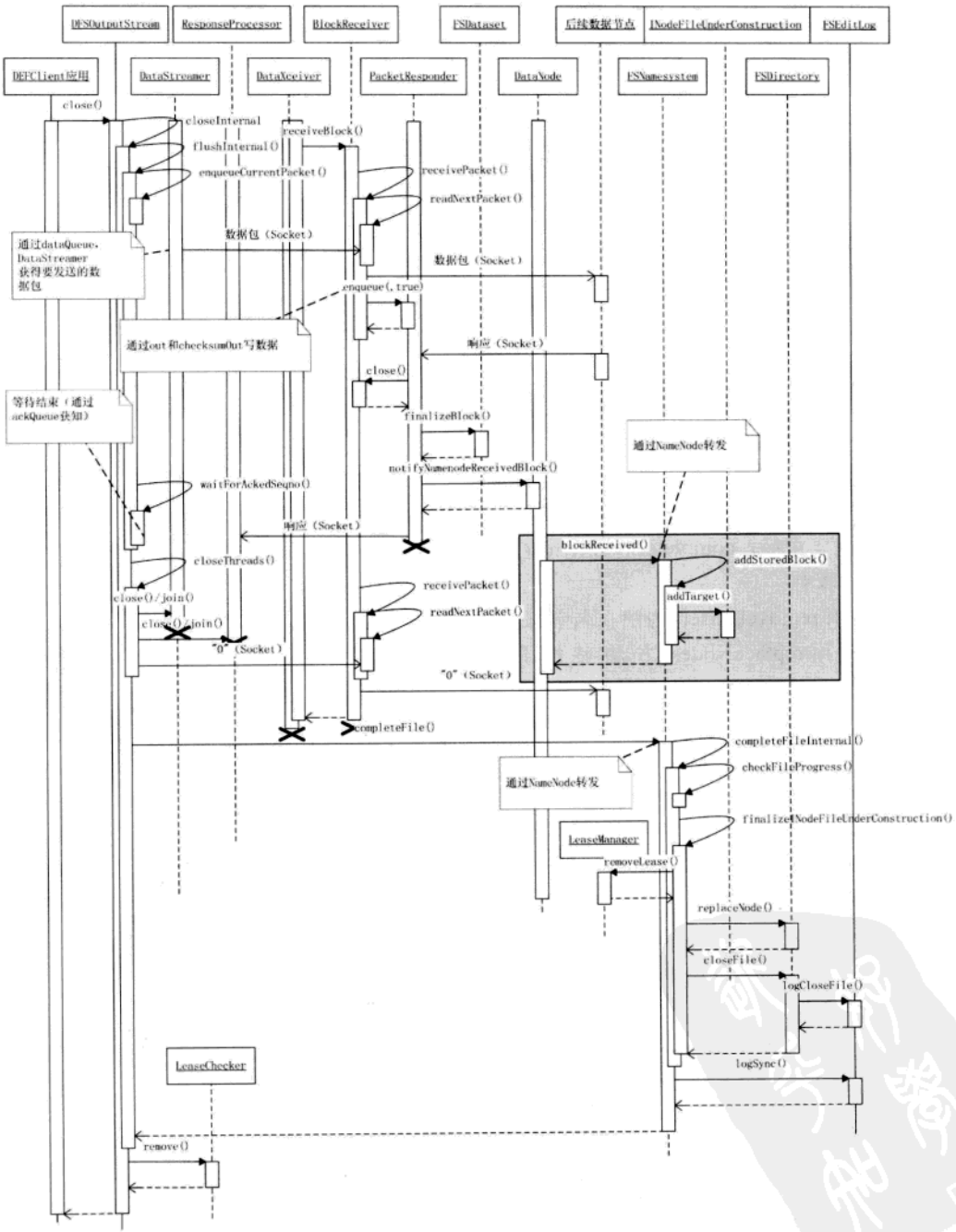



图 9-13　DFSOutputStream.close() 序列图

当客户端接收到数据流管道上发送过来的响应时，waitForAckedSeqno() 调用返回，flushInternal() 执行结束，通过输出流写出的数据，已经都保持在数据节点上了。接下来执行的 closeThreads() 方法，会结束输出流拥有的数据发送器 DataStreamer 线程和应答处理器 ResponseProcessor 线程，并往数据流管道中写入一个长度为 0 的数据包（即发送整数 0 到 TCP 连接上）。数据节点上的 BlockReceiver 接收到这个特殊的数据包，也就知道客户端已经发送了所有的数据，在将该包转发给数据流管道的下游后，BlockReceiver.receiveBlock() 的执行也就结束了，相应的线程也可以退出了（DataXceiverServer 使用了一客户一线程的模式，BlockReceiver 是服务线程执行的主要方法）。

NameNode.notifyNamenodeReceivedBlock() 只是将数据块提交的信息放入数据节点的 receivedBlockList 队列中，数据节点服务线程会定期通过远程方法 DatanodeProtocol.blockReceived() 向名字节点提交数据块。名字节点的 FSNamesystem .addStoredBlock() 方法，会将该数据节点的信息放入打开文件的 INodeFileUnderConstruction 对象中，即记录了文件在该数据节点上拥有一个数据块副本。

客户端在写出长度为 0 的数据包后，就会开始通过客户端与名字节点间的远程方法 completeFile() 试图关闭文件。在使用 checkFileProgress() 方法检查了文件当前的数据块副本数符合系统要求后，completeFileInternal() 调用 finalizeINodeFileUnderConstruction() 关闭文件。主要步骤有：

1）使用 removeLease() 方法在名字节点的租约管理器中移除记录。

2）使用 replaceNode() 方法替换目录树中的 INodeFileUnderConstruction 节点为 INodeFile。

3）由 closeFile() 方法并通过 logCloseFile() 方法记录日志。

4）FSNamesystem.completeFile() 会通过 logSync() 持久化日志记录。

图 9-13 中，DFSOutputStream.close() 调用 ClientProtocol.completeFile() 时，已经有数据节点提交了数据块，即图中灰色背景部分已经在调用该远程方法前执行了。如果参与写操作的数据节点，在 completeFile() 被调用时，都没有执行数据块上报，那么，在执行了 checkFileProgress() 方法后，远程方法就会返回，返回值为 false，close() 方法需等待一段时间后，再调用远程方法 completeFile()，关闭文件。

最后，DFSOutputStream.close() 在客户端的租约管理器对象 leasechecker 中移除租约记录，结束整个方法的执行。

## 9.4 DistributedFileSystem 的实现

Hadoop 定义了抽象文件系统 org.apache.hadoop.fs.FileSystem，以满足不同应用的各种数据访问需求，Hadoop 分布式文件系统作为该抽象文件系统的一个具体文件系统，其实现为 DistributedFileSystem。也就是说，当以 hdfs 模式通过 FileSystem.get() 获取具体文件系统时，返回的是一个 DistributedFileSystem 对象。代码如下：

```
Path inPath  = new Path("hdfs://<ip>:<port>/user/alice/in/hello.txt");
FileSystem hdfs = FileSystem.get(inPath.toUri(), conf);     }
```

DistributedFileSystem 是适配器模式的一个典型应用，它将 DFSClient 提供的访问 HDFS 能力，包装成 FileSystem 要求的界面。下面代码给出了 DistributedFileSystem 的成员变量和几个典型方法的实现：

```
public class DistributedFileSystem extends FileSystem {
   private Path workingDir;
   private URI uri;

   DFSClient dfs;
   private boolean verifyChecksum = true;

   static{ //加载默认配置
      Configuration.addDefaultResource("hdfs-default.xml");
      Configuration.addDefaultResource("hdfs-site.xml");
   };
   ……
   public URI getUri() { return uri; }

   public void initialize(URI uri, Configuration conf)throws…… {
      super.initialize(uri, conf);
      setConf(conf);

      String host = uri.getHost();
      ……
      InetSocketAddress namenode
            = NameNode.getAddress(uri.getAuthority());
      this.dfs = new DFSClient(namenode, conf, statistics);
      this.uri = URI.create(uri.getScheme()+"://"+uri.getAuthority());
      this.workingDir = getHomeDirectory();
   }
   ……
   public FSDataInputStream open(Path f, int bufferSize) throws……{
      statistics.incrementReadOps(1); //更新度量信息
      return new DFSClient.DFSDataInputStream(
            dfs.open(getPathName(f), bufferSize,
            verifyChecksum, statistics));
   }
   ……
   public boolean delete(Path f, boolean recursive) throws IOException {
      statistics.incrementWriteOps(1); //更新度量信息
      return dfs.delete(getPathName(f), recursive);
   }
   ……
}
```

其中，成员变量 dfs 的类型是 DFSClient，而其他三个成员变量，用于实现 FileSystem 要求的，而 DFSClient 中没有提供的功能，如 workingDir 保存了文件系统的当前工作路径，uri

保存文件系统的 URI 模式和授权机构部分。通过 getUri() 可以获得成员变量 uri 保存的信息。

DistributedFileSystem.initialize() 用于初始化对象，成员变量 workingDir、uri 和 dfs 都在这个方法中被初始化。该方法需要一个 Configuration 对象，注意，类的 static 块会加载 hdfs-default.xml 和 hdfs-site.xml 两个配置文件，它们提供了访问 Hadoop 文件系统的一些默认配置。删除文件的 delete() 成员函数是典型的文件和目录相关事务方法，它的实现，就是简单地调用 DFSClient 的同名方法。DistributedFileSystem .open() 用于打开 HDFS 文件，它返回一个 FSDataInputStream 对象，用于读取文件数据。方法 open() 实际返回一个 DFSDataInputStream 对象，它是 FSDataInputStream 的子类，也是为了兼容 Hadoop 文件系统界面而引入的包装器，它的大部分操作都委托给了通过 DFSClient.open() 创建的 DFSInputStream 对象。

DistributedFileSystem 的实现就介绍到这里，它确实是适配器模式的一个中规中矩的应用。

## 9.5 HDFS 常用工具

使用 HDFS 时的两个常用工具是多任务工具 dfsadmin 和文件系统 Shell 命令。

多任务工具 dfsadmin 的实现是 org.apache.hadoop.hdfs.tools.DFSAdmin。严格来说，文件系统 Shell 是 Hadoop 文件系统（注意，不是 Hadoop 分布式文件系统）的一部分，它不但支持对 HDFS 进行操作，也可用于其他实现了 Hadoop 文件系统的具体文件系统中，它的实现在 org.apache.hadoop.fs 包中。DFSAdmin 继承自文件系统 Shell 命令的实现 FsShell，它们在用法上有很多的相同点。

### 9.5.1 FsShell

Hadoop 文件系统 Shell 命令可以执行其他文件系统中常见的操作，例如读取文件、移动文件、创建目录、删除数据等。在终端上可以通过下面的命令，获得 Shell 命令的详细帮助信息：

```
hadoop fs -help
```

“hadoop fs”命令还可以在本地文件系统和其他文件系统，如 HDFS 之间进行文件复制，copyFromLocal 命令可以将本地文件复制到 HDFS 文件中，下面的命令就将“/home/hadoopinternal/demo.txt”复制到 HDFS 中的“/user/admin/data/demo.txt”文件。这个命令省略了 HDFS 文件 URI 的模式和授权机构部分，直接使用了配置文件中的配置信息：

```
hadoop fs -copyFromLocal /home/hadoopinternal/demo.txt /user/admin/data/demo.txt
```

下面以 copyFromLocal 命令的执行为例，来研究文件系统 Shell 的实现。

FsShell 是一个 Java 程序，实现了应用入口的 main() 方法，是一个典型的基于 ToolRunner 实现的应用。为了简化 Hadoop 命令行应用的开发（大量 MapReduce 程序使用命令行方式运行作业），Hadoop 提供了一些辅助类，包括 ToolRunner、GenericOptionsParser

和 Tool。

GenericOptionsParser 是一个类，用来解释常用的 Hadoop 命令行选项，并根据需要为 Hadoop 配置 Configuration 对象设置相应的配置项。一般情况下不直接使用 GenericOptionsParser，更方便的方式是：实现 Tool 接口，通过 ToolRunner 来运行应用程序，ToolRunner 内部调用 GenericOptionsParser，相关代码如下：

```
public class FsShell extends Configured implements Tool {
   ......
   public static void main(String argv[]) throws Exception {
      FsShell shell = new FsShell();
      int res;
      try {
         res = ToolRunner.run(shell, argv);
      } finally {
         shell.close();
      }
      System.exit(res);
   }
   ......
}
```

FsShell 实现了 Tool 接口，在其 main() 方法中，通过 ToolRunner 来运行 FsShell 对象。ToolRunner.run() 的代码如下：

```
public interface Tool extends Configurable {
   // 用于实现具体业务逻辑
   int run(String [] args) throws Exception;
}

public class ToolRunner {
   public static int run(Configuration conf, Tool tool, String[] args)
         throws Exception{
      if(conf == null) {
         conf = new Configuration();
      }
      GenericOptionsParser parser = new GenericOptionsParser(conf, args);
      tool.setConf(conf);

      String[] toolArgs = parser.getRemainingArgs();
      return tool.run(toolArgs);
   }

   public static int run(Tool tool, String[] args) throws Exception{
      return run(tool.getConf(), tool, args);
   }
   ......
}
```

ToolRunner.run() 方法会创建必要的 Configuration 对象，然后使用 GenericOptionsParser

对一些常用的命令行参数进行分析，并设置到 Configuration 对象中。GenericOptionsParser 不感兴趣的参数，可通过 getRemainingArgs() 方法获得，并传递给具体的应用，如上述 copyFromLocal 命令中，GenericOptionsParser 不会处理参数“-copyFromLocal”、“/home/hadoopinternal/demo.txt”和“/user/admin/data/demo.txt”，它们由 getRemainingArgs() 返回，并传递给 FsShell。

FsShell.main() 使用的两个参数的 ToolRunner.run() 方法，是一个简单的辅助方法。

GenericOptionsParser 可以处理 -conf、-D、-fs、-jt 等选项，如 -fs 就用于设置系统的默认文件系统，下面的命令行会将文件“demo.txt”复制到 Hadoop 文件系统 hdfs://example.com:54300/ 中：

```
hadoop fs -fs hdfs://example.com:54300/ -copyFromLocal /home/hadoopinternal/demo.txt /user/admin/data/demo.txt
```

选项 -D 用于设置 Configuration 对象中的配置属性，下述命令的效果和上面通过 -fs 修改默认文件系统是一样的：

```
hadoop fs -D fs.default.name=hdfs://example.com:54300/ -copyFromLocal /home/hadoopinternal/demo.txt /user/admin/data/demo.txt
```

GenericOptionsParser 在 ApacheCommons CLI（Command Line Interface）的基础上开发，CLI 是一个解析命令行输入的工具包，支持多种输入参数格式。感兴趣的读者可以分析 GenericOptionsParser 的实现，本书就不再深入探讨了。

FsShell.main() 中的 ToolRunner.run() 最终会执行 FsShell.run() 方法，这个方法根据不同的 Shell 命令调用不同的处理函数。如 -copyFromLocal 命令就实现在 FsShell 的成员函数 copyFromLocal() 中，代码如下：

```
void copyFromLocal(Path[] srcs, String dstf) throws IOException {
   Path dstPath = new Path(dstf);
   FileSystem dstFs = dstPath.getFileSystem(getConf());
   if (srcs.length == 1 && srcs[0].toString().equals("-"))
      copyFromStdin(dstPath, dstFs);
   else
      dstFs.copyFromLocalFile(false, false, srcs, dstPath);
}
```

该方法会将数组 srcs 中包含的本地文件复制到目标 dstf 中，复制使用的是 FileSystem.copyFromLocalFile()，它的实现不算复杂，这里就不给出代码了。

## 9.5.2 DFSAdmin

DFSAdmin 继承自 FsShell，它的实现和 FsShell 类似，也是通过 ToolRunner.run() 执行 DFSAdmin.run() 方法，该方法根据不同的命令调用相应的处理函数。

以改变或查询安全模式状态的命令为例，DFSAdmin.setSafeMode() 在分析了 -safemode 的命令参数后，使用 DistributedFileSystem.setSafeMode()，最终调用了远程方法 setSafeMode.setSafeMode()，代码如下：

```
public void setSafeMode(String[] argv, int idx) throws IOException {
    ……// 参数检查
    FSConstants.SafeModeAction action;
    Boolean waitExitSafe = false;

    if ("leave".equalsIgnoreCase(argv[idx])) { // 离开安全模式
        action = FSConstants.SafeModeAction.SAFEMODE_LEAVE;
    } else if ("enter".equalsIgnoreCase(argv[idx])) { // 进入安全模式
        action = FSConstants.SafeModeAction.SAFEMODE_ENTER;
    } else if ("get".equalsIgnoreCase(argv[idx])) { // 查询安全模式状态
        action = FSConstants.SafeModeAction.SAFEMODE_GET;
    } else if ("wait".equalsIgnoreCase(argv[idx])) {
        // 等待 HDFS 离开安全模式
        action = FSConstants.SafeModeAction.SAFEMODE_GET;
        waitExitSafe = true;
    } else {
        ……// 输出帮助信息
    }
    DistributedFileSystem dfs = (DistributedFileSystem) fs;
    boolean inSafeMode = dfs.setSafeMode(action); // 调用远程方法

    // 等待 HDFS 离开安全模式其实是通过不断轮询安全模式状态实现的
    if (waitExitSafe) {
        while (inSafeMode) {
        try {
            Thread.sleep(5000); // 等待
        } catch (java.lang.InterruptedException e) {
            throw new IOException("Wait Interrupted");
        }
            inSafeMode = dfs.setSafeMode(action); // 再次查询状态
        }
    }

    System.out.println("Safe mode is " + (inSafeMode ? "ON" : "OFF"));
}
```

## 9.6　小结

客户端 DFSClient 是应用访问 Hadoop 分布式文件系统的起点，也是我们研究 HDFS 的终点。作为文件系统界面，DFSClient 提供了处理文件和目录相关事务和基于流的文件数据读写等典型文件系统应用开发接口，同时也可支持对 HDFS 进行系统管理。

从实现角度看，文件和目录、系统管理相关事务等都不需要和数据节点直接打交道，它们的实现比较简单，通过远程接口访问名字节点提供的服务即可。支持文件数据读写的输入 / 输出流比较复杂，为保证数据的完整性，它们都必须处理数据检验带来的基于块的校验和基于字节流的读写差异。

对于读数据，在确定数据块的位置后，就可以通过数据节点的流式接口获取数据，但客

户端需要处理校验错误信息上报名字节点，失败重试、数据节点返回数据的边界等细节；对于写数据，正常的流程涉及打开或创建文件、新数据块分配、数据流管道的建立、数据组包的发送、数据包应答处理、租约更新、关闭文件等要点，同时还要应对数据流管道建立失败、写数据时数据节点出错等故障，是 HDFS 上最复杂的流程之一，也是掌握 HDFS 原理的难点。本章最后还研究了基于 DFSClient 实现的 DistributedFileSystem、DFSAdmin 和 FsShell。

客户端提供了一个很好的观测点，让我们回顾 HDFS 的名字节点和数据节点提供的功能和它们的配合，展示了 HDFS 的实现全景。