



华章科技

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。



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

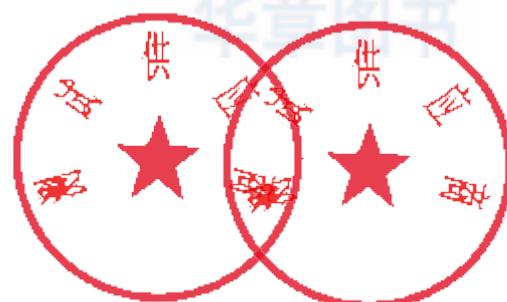


# Hadoop 技术内幕： 深入解析 Hadoop Common 和 HDFS 架构设计与实现原理

蔡斌 陈湘萍 著

HZ BOOKS

华章图书



## 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

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

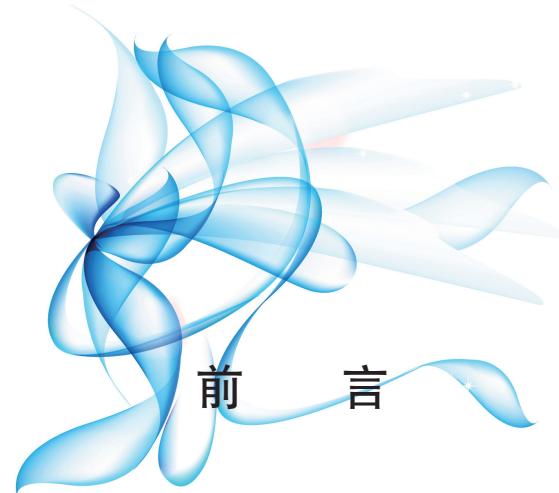
凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com



## 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<sup>⊖</sup>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。

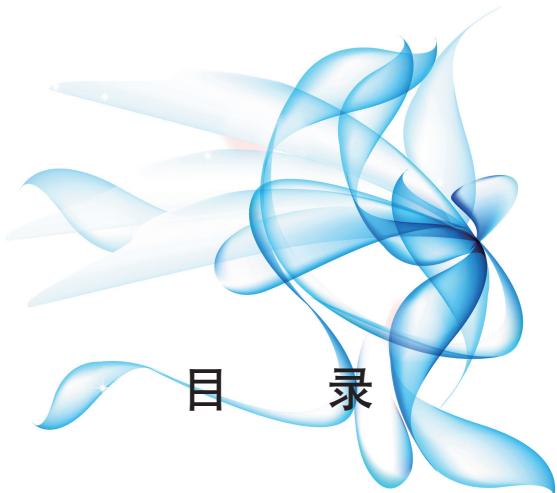
感谢和我们一起工作、研究和应用 Hadoop 的腾讯数据平台部，以及 IBM 中国研究中心和中山大学的领导和同事们，本书的很多内容是对实际项目的总结。

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。



---

<sup>⊖</sup> 参见华章网站 [www.hzbook.com](http://www.hzbook.com)——编辑注



## 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 Cafarella (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。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  $\mu$  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。

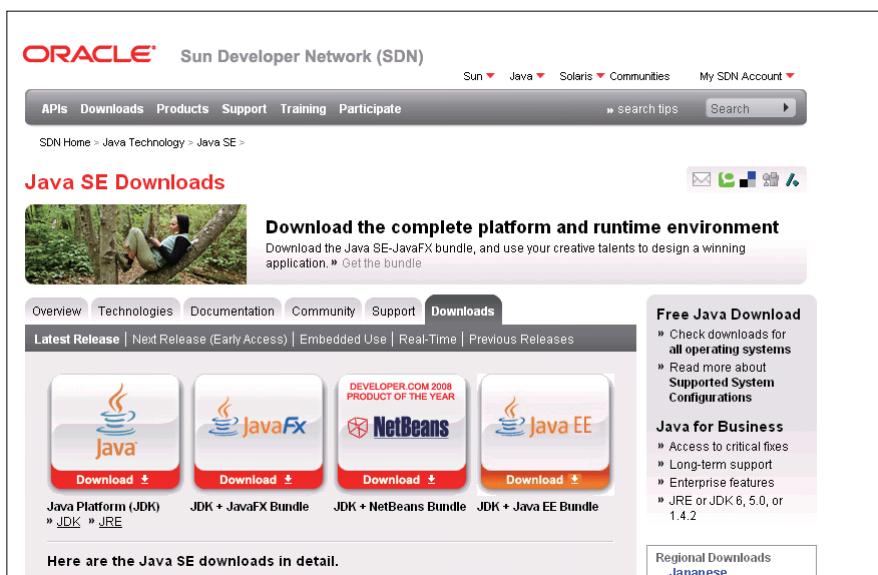


图 1-2 JDK 下载首页

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。

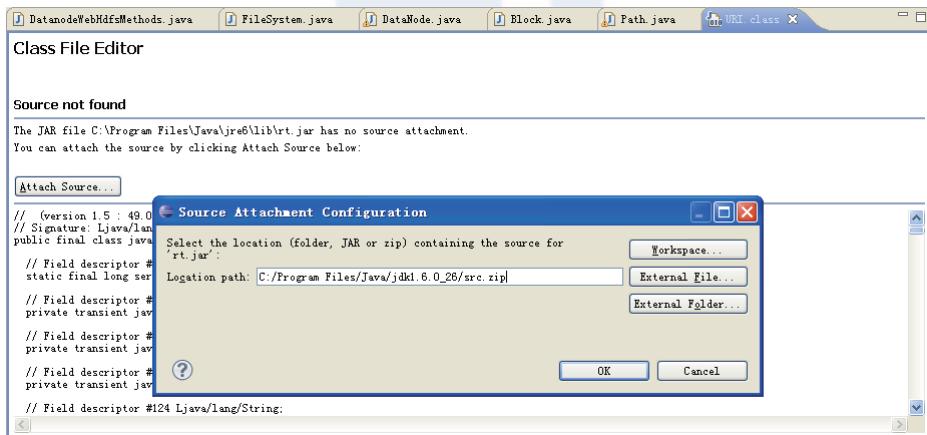


图 1-6 字节码和源代码关联

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。

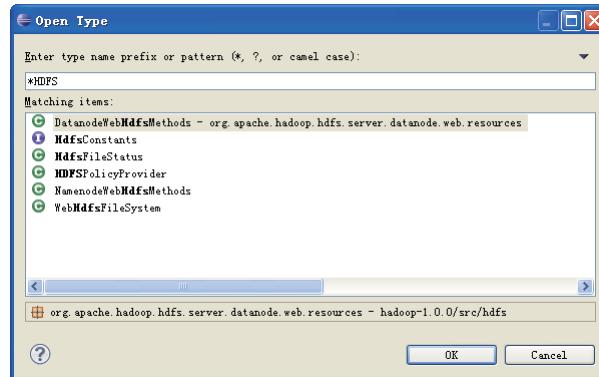


图 1-7 在 Eclipse 中查看名字包含 HDFS 的所有类

---

**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。

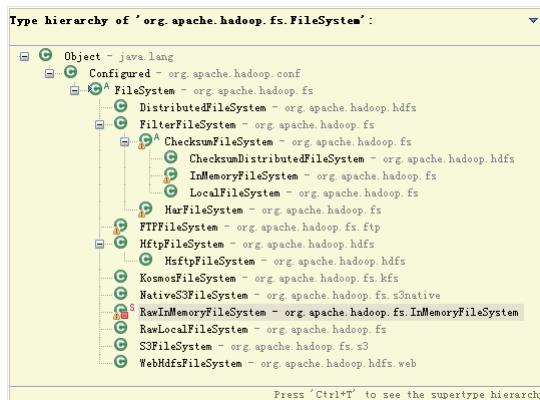


图 1-8 在 Eclipse 中显示类型层次结构

#### 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。

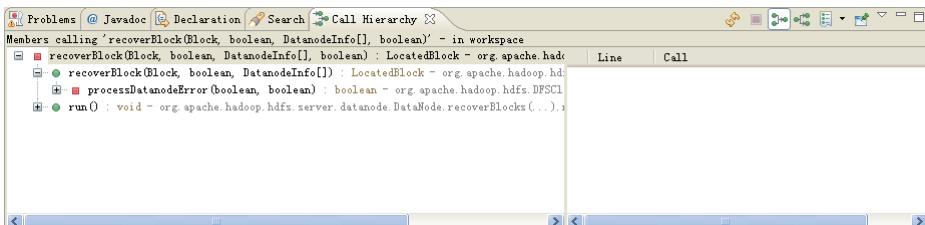


图 1-9 在 Eclipse 中查看方法的调用关系

---

**注意** 快捷键是日常开发调试中最为便捷的技巧。Eclipse 中的快捷键可谓博大精深，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。

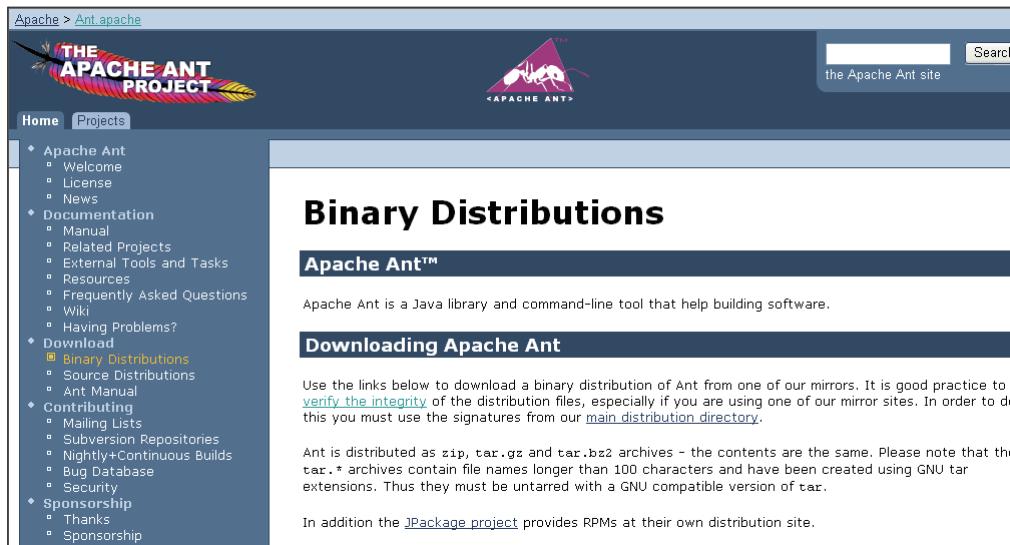


图 1-10 Apache Ant 下载页面

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。

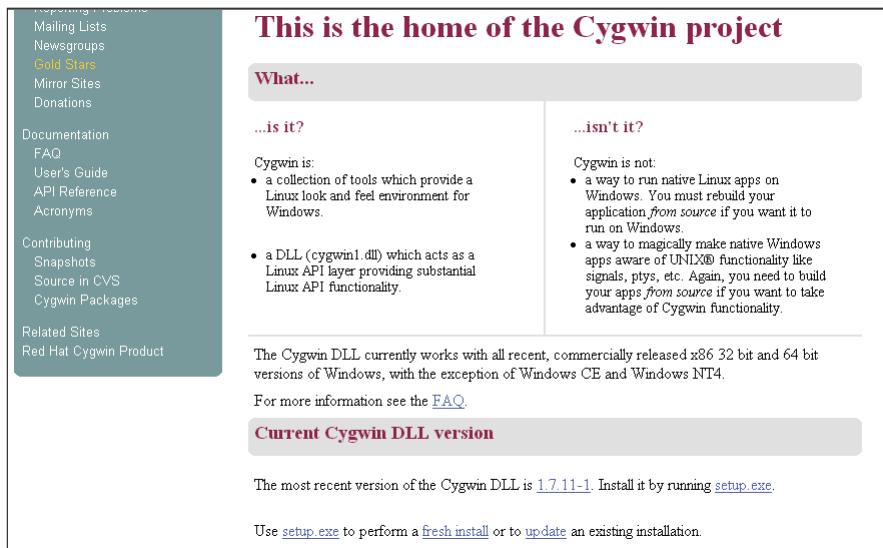


图 1-11 Cygwin 下载页面

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)。

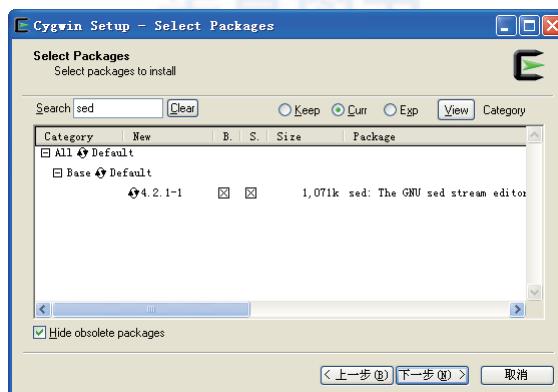


图 1-12 Cygwin 中选择在线编辑器 sed

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。

```
Administrator@winxp-pro-vm ~
$ ant -version | sed "s/version/Version/g"
Apache Ant(TM) Version 1.8.3 compiled on February 26 2012
Administrator@winxp-pro-vm ~
$
```

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。

Welcome to Apache™ Hadoop™!

- What Is Apache Hadoop?
- Download Hadoop
- Who Uses Hadoop?
- News
  - 27 December, 2011: release 1.0.0 available
  - March 2011 - Apache Hadoop takes top prize at Media Guardian Innovation Awards
  - January 2011 - Zookeeper Graduates
  - September 2010 - Hive and Pig Graduate
  - May 2010 - Avro and HBase Graduate
  - July 2009 - New Hadoop Subprojects
  - March 2009 - ApacheCon EU
  - November 2008 - ApacheCon US

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。

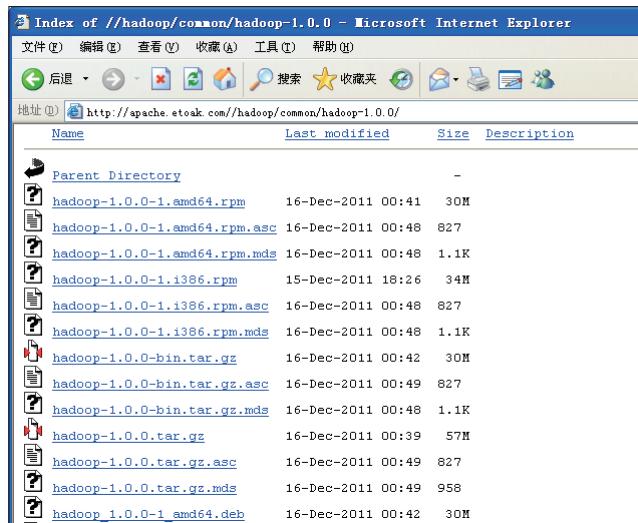


图 1-15 Apache Hadoop 1.0 的一个下载页

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。

```
E:/cygdrive/e/hadoop-1.0.0
Administrator@winxp-pro-vm ~
$ cd e:
Administrator@winxp-pro-vm /cygdrive/e
$ cd hadoop-1.0.0
Administrator@winxp-pro-vm /cygdrive/e/hadoop-1.0.0
$ ant eclipse
```

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。

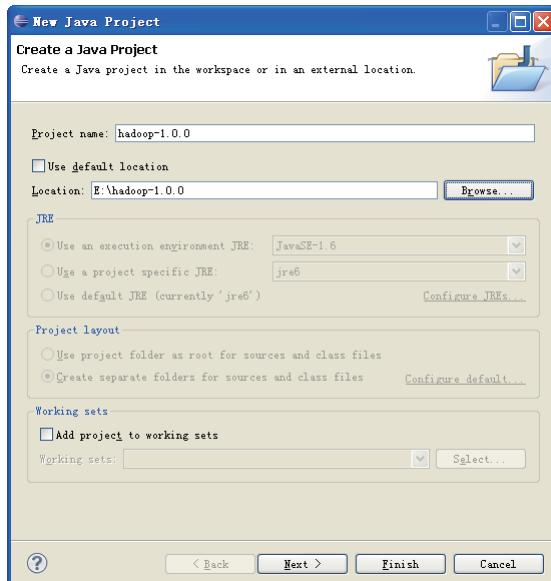


图 1-17 创建 Eclipse 项目

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“Variable”，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。

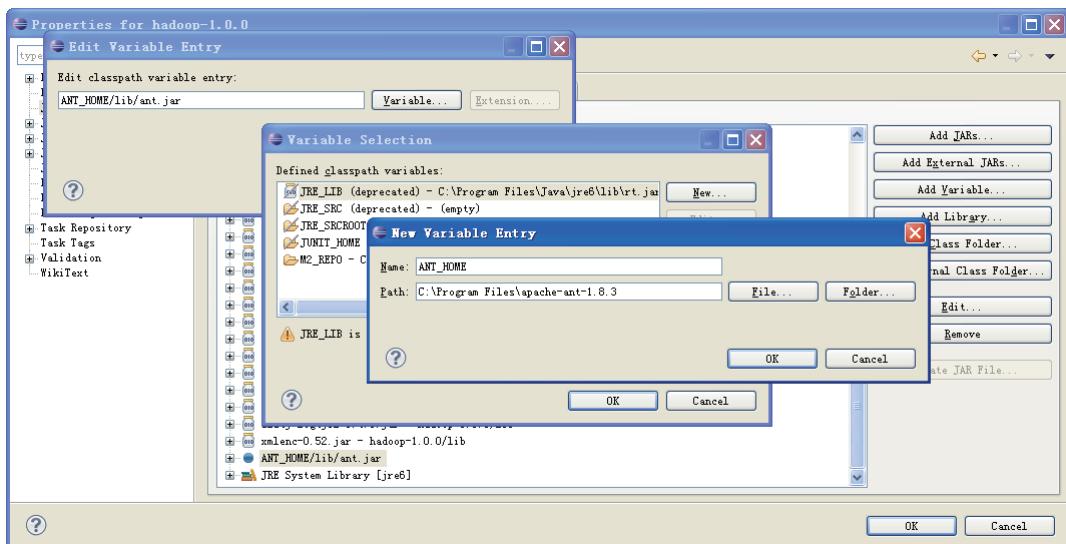


图 1-18 创建 ANT\_HOME 变量

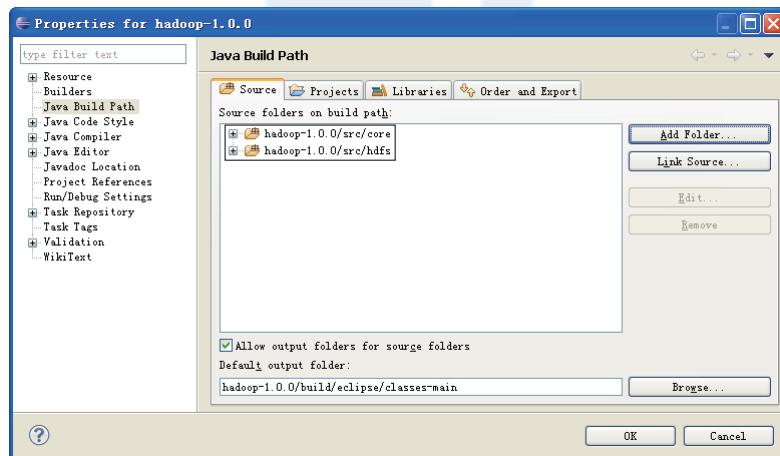


图 1-19 保留 core 和 hdfs 两个源码目录

### 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。

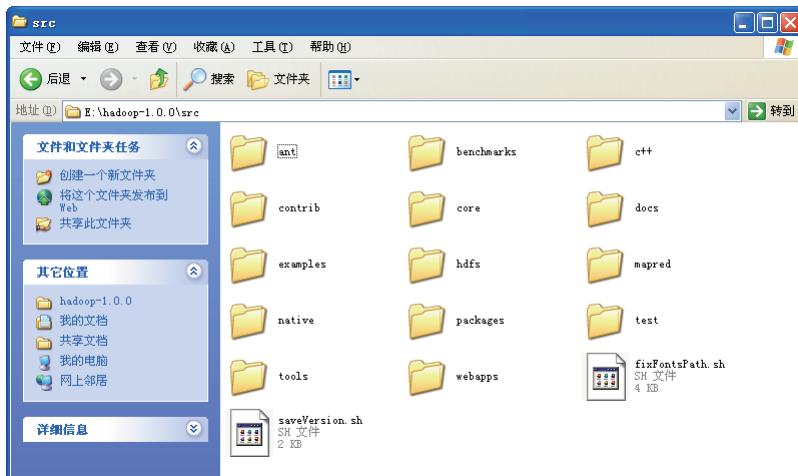


图 1-20 Hadoop 源码组织

## 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。

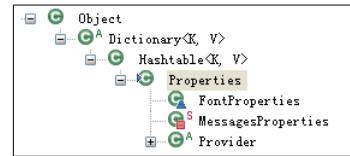


图 2-1 Properties 的继承关系

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
nodes should 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：

```

Configuration conf = 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。

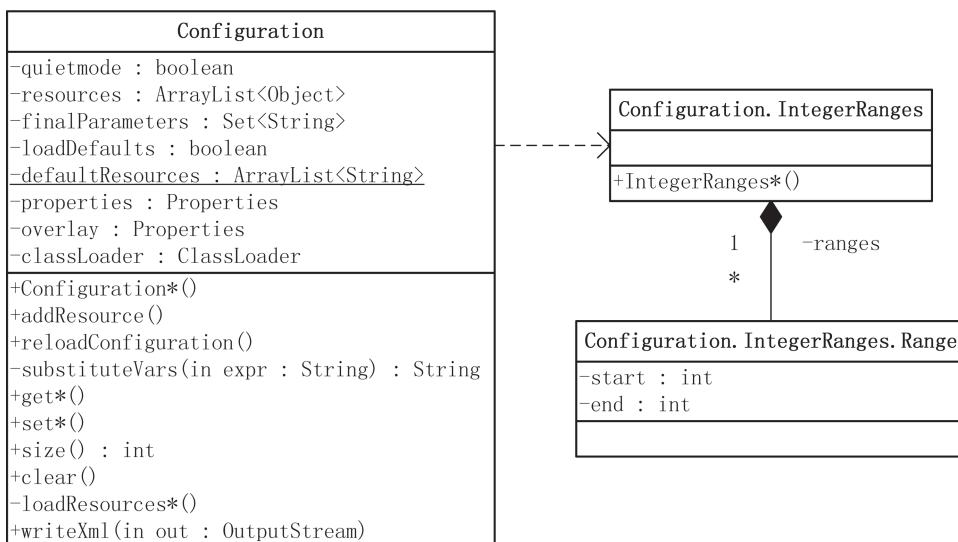


图 2-2 Configuration 类图

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
        DocumentBuilderFactory docBuilderFactory
            = 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。

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

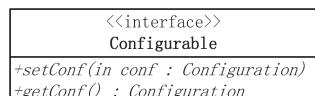


图 2-3 Configurable 类图

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 使用的基于键 - 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）。

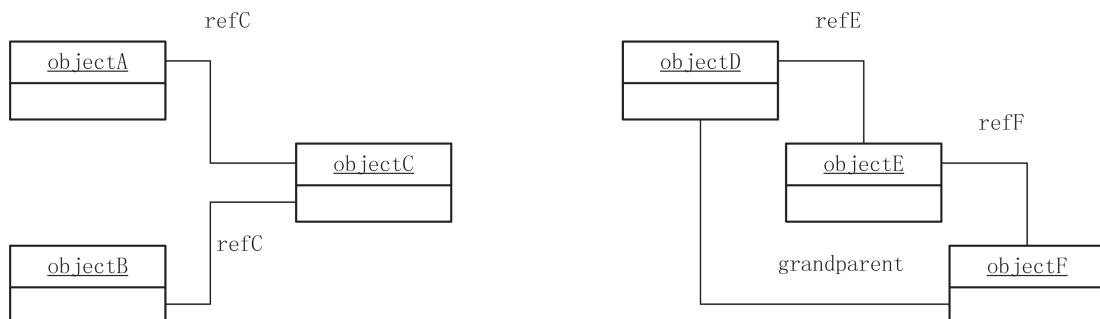


图 3-1 复杂的序列化例子

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 1Ug.h...
00 00 00 00 03 61 BB 8B 00 00 00 00 02 59 EC CB .....a.. .....Y..
.....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。

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 DataOutput 流, 序列化的结果保存在流中
     * @throws IOException
    
```

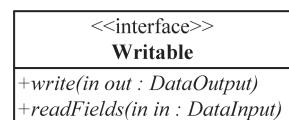


图 3-2 Writable 类图

```

    */
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 Hadoop I/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：

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

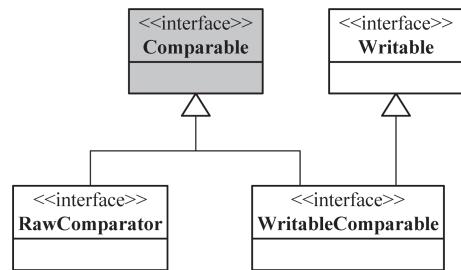


图 3-3 RawComparator 和  
WritableComparable 类图

以 IntWritable 为例，它的 RawComparator 实现中（WritableComparable 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 WritableComparable {
    .....
    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));
    }
    .....
}
```

WritableComparable 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=
    WritableComparable.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。

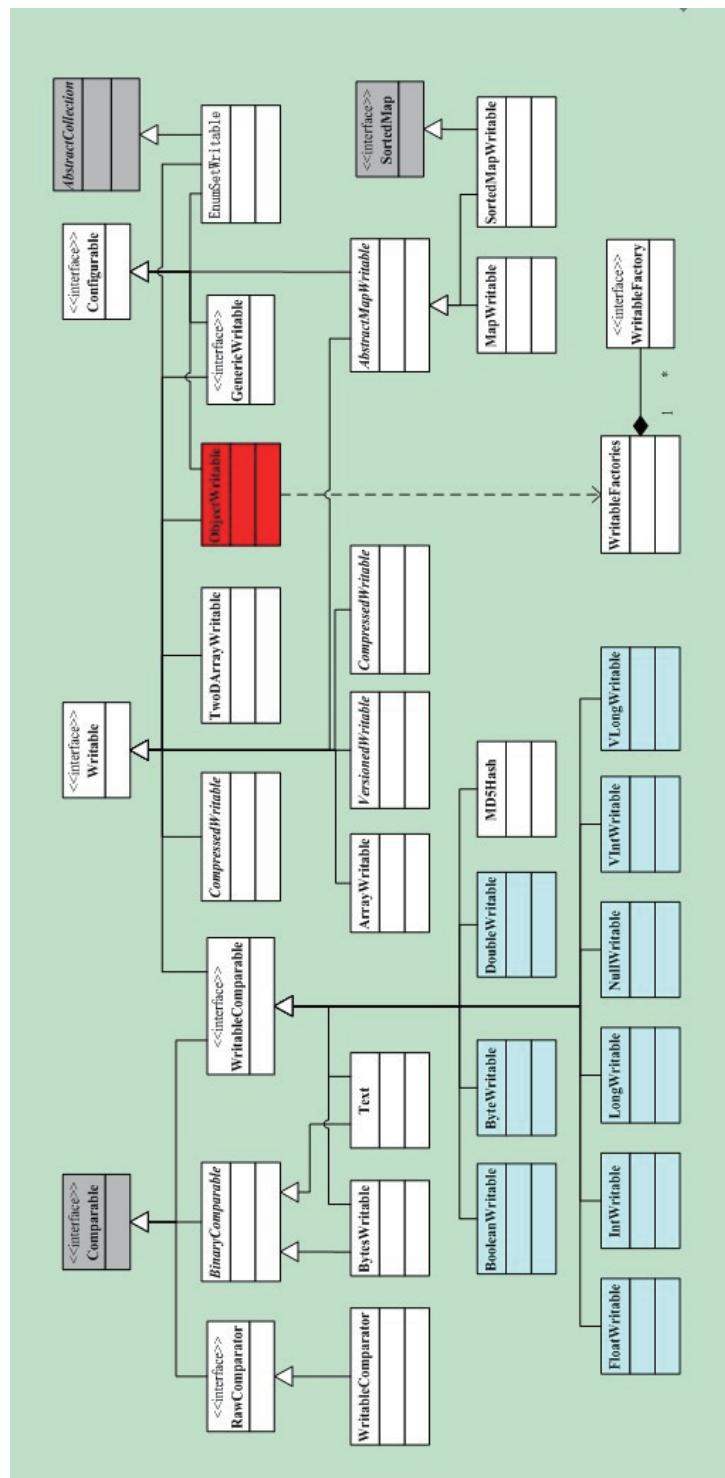


图 3-4 Writable 的子类

### 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<sup>⊖</sup>

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 VIntWritable	4 1~5
浮点型 (float)	FloatWritable	4
长整型 (long)	LongWritable VLongWritable	8 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

<sup>⊖</sup> 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 static void writeVInt(DataOutput stream, int i) throws IOException
    {
        writeVLong(stream, i);
    }

    /**
     * @param stream 保存序列化结果输出流
     * @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.writeString(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。<sup>⊖</sup>

- ❑ 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）。

---

<sup>⊖</sup> 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 <sup>⊖</sup>

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 =

```

<sup>⊖</sup> 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。

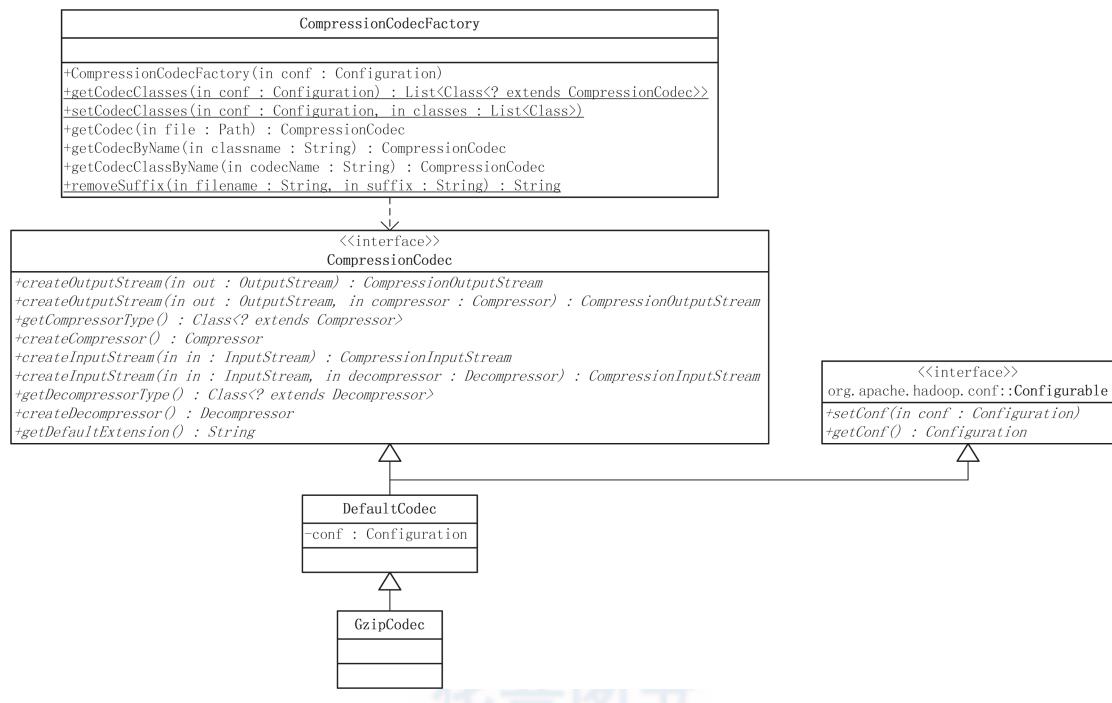


图 3-5 `CompressionCodec` 及其相关类

`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。

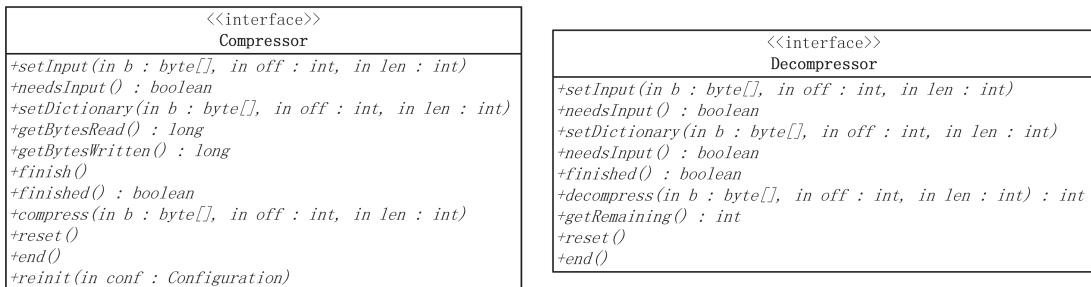


图 3-6 Compressor 和 Decompressor 的类图

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 ObjectOutputStream 和 OutputStream。

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。

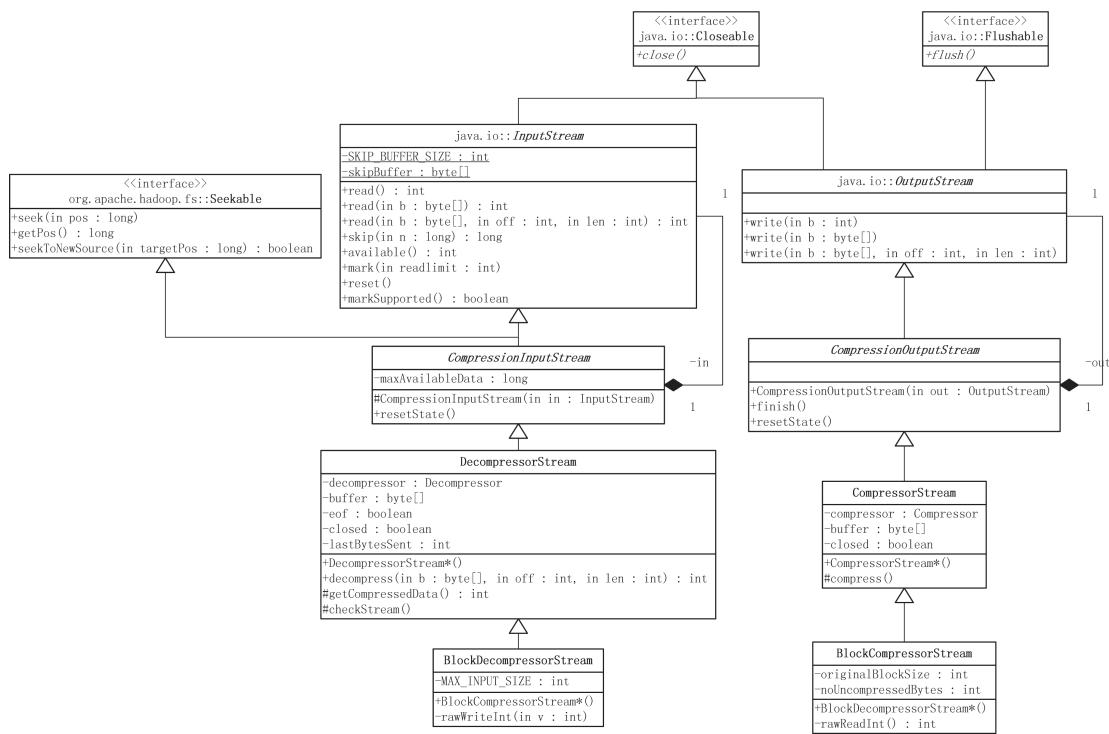


图 3-7 压缩流 / 解压缩流类图

`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。

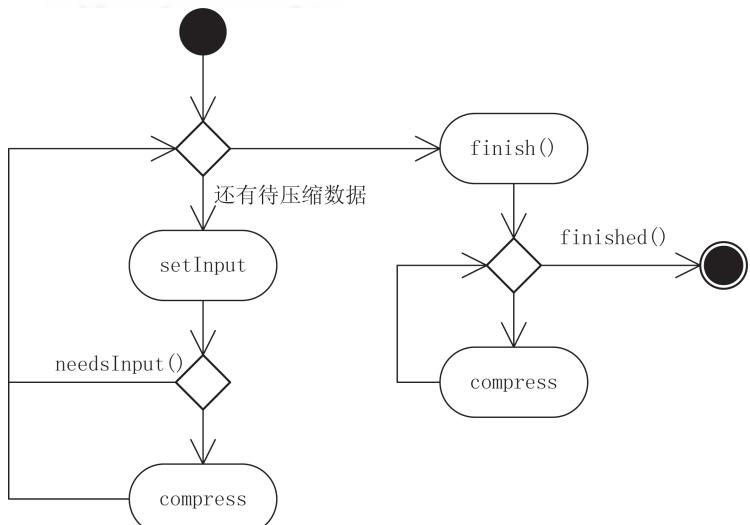


图 3-8 压缩器方法的典型调用顺序

### 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<sup>⊖</sup>。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）

---

<sup>⊖</sup> 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
#define 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);

#endif // _Included_org_apache_hadoop_io_compress_snappy_SnappyCompressor
```

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 libsnaappy.so 库。

### 3.2.5 支持 Snappy 压缩

Snappy<sup>⊖</sup>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;
```

---

<sup>⊖</sup> 项目地址: [http://code.google.com/p/snappy/。](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。

