



Java领域影响力超群的著作之一，与《Java编程思想》齐名，10余年
全球畅销不衰，广受好评

针对Java 11全面更新，系统讲解Java语言的核心概念、语法、重要特性
和开发方法，包含大量案例，实践性强



P Pearson

Java 核心技术 卷II

高级特性（原书第11版）

Core Java Volume II—Advanced Features

(Eleventh Edition)

[美] 凯·S·霍斯特曼 (Cay S. Horstmann) 著

陈昊鹏 译



机械工业出版社
China Machine Press



Java

核心技术 卷II

高级特性 (原书第11版)

Core Java Volume II—Advanced Features
(Eleventh Edition)

[美] 凯·S·霍斯特曼 (Cay S. Horstmann) 著
陈昊鹏 译



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Java 核心技术 卷 II 高级特性 (原书第 11 版) / (美) 凯·S. 霍斯特曼 (Cay S. Horstmann) 著; 陈昊鹏译. —北京: 机械工业出版社, 2019.12 (2020.5 重印)
(Java 核心技术系列)

书名原文: Core Java, Volume II—Advanced Features (Eleventh Edition)

ISBN 978-7-111-64343-2

I. J… II. ①凯… ②陈… III. JAVA 语言 – 程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字 (2019) 第 268688 号

本书版权登记号: 图字 01-2019-2815

Authorized translation from the English language edition, entitled *Core Java, Volume II—Advanced Features (Eleventh Edition)*, ISBN: 978-0-13-516631-4, by Cay S. Horstmann, published by Pearson Education, Inc., Copyright © 2019 Pearson Education Inc., Portions Copyright © 1996–2013 Oracle and/or its affiliates.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by China Machine Press, Copyright © 2020.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括香港、澳门特别行政区及台湾地区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

Java 核心技术 卷 II 高级特性 (原书第 11 版)

出版发行: 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: 关 敏

责任校对: 殷 虹

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2020 年 5 月第 1 版第 3 次印刷

开 本: 186mm×240mm 1/16

印 张: 43.5

书 号: ISBN 978-7-111-64343-2

定 价: 149.00 元

客服电话: (010) 88361066 88379833 68326294

投稿热线: (010) 88379604

华章网站: www.hzbook.com

读者信箱: hzit@hzbook.com

版权所有 • 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

内容简介

本书由拥有20多年教学与研究经验的资深Java技术专家撰写（获Jolt大奖），本版针对Java 11全面更新。

全书共分12章。第1章介绍了Java中的流库；第2章涵盖输入/输出处理，探讨了Java 11中引入的广受欢迎的改进和优化；第3章介绍了XML，展示如何解析XML文件、生成XML和使用XSL转换；第4章讲解了网络API，以及怎样连接到服务器、实现自己的服务器、创建HTTP连接，并讨论了新的HTTP客户端；第5章介绍了数据库编程，重点讲解JDBC，即Java数据库连接API；第6章涵盖如何使用新的日期和时间库来处理日历及时区；第7章讨论国际化；第8章介绍3种处理代码的技术；第9章讲解从Java 9开始引入的Java平台模块系统，以促进Java平台和核心类库的有序演化；第10章继续介绍Java安全模型，展示怎样编写类加载器和安全管理器，以及允许使用消息、代码签名、授权和认证及加密等重要特性的安全API；第11章讨论没有纳入卷Ⅰ的所有Swing知识，包括树形构件、表格构件，以及Java 2D API；第12章介绍本地方法，这个功能支持你调用为微软Windows API这样的特殊机制而编写的各种方法。

作者简介

凯·S·霍斯特曼
(Cay S. Horstmann)

圣何塞州立大学计算机科学系教授、Java的倡导者。他是《Java核心技术》两卷本的作者，并著有*Core Java SE 9 for the Impatient, Second Edition*和*Scala for the Impatient, Second Edition*（均由Addison-Wesley出版）。他还为专业程序员和计算机科学专业的学生撰写过数十本其他图书。



华章图书

一本打开的书，
一扇开启的门，
通向科学殿堂的阶梯，
托起一流人才的基石。

译 者 序

《Java核心技术 卷II 高级特性》(原书第11版)就要出版了！本书在上一版的基础上针对Java 11进行了全面修订，除了之前版本中包含的高级UI特性、企业编程、网络、安全等主题之外，还新纳入了模块系统等内容，以全面反映Java 11的最新特性。从某种程度上说，这本书的版本演进反映了Java语言本身的演进过程。虽然在人工智能和大数据的发展风起云涌的大环境下，以Python为代表的新兴语言不断涌现并迅速被广为接受，但Java在企业级Web应用方面的优势还是非常明显，其完备的语言特性和丰富的生态圈使得它在编程语言排行榜上一直位居前列，深受广大程序员的青睐和追捧。可以预见，Java在未来仍然会有广阔的市场空间，Java程序员的数量会持续增长，Java的生态圈也会不断地扩展壮大。

本书在写作上保留了以往的风格，并针对新内容进行了扩展，读者既可以将其作为学习Java语言的进阶教材，学习利用Java语言编写企业级Web应用的高级特性，又可以将其当作Java语言的API手册，在编程时作为查询API细节的工具书。本书提供的所有样例代码几乎都进行了更新，以反映Java 11中新的变化。

本书中文版是在之前版本的基础上完成的。在翻译本书的过程中，译者不但对新增加的内容进行了翻译，还对之前版本中存在的错误和不符合中文表达习惯的地方进行了修正，力求以准确和流畅的语言重现英文版的内容和韵味，希望读者在阅读本书时能够感受到Java的魅力和作者的风格。

Java技术在物联网时代还会因其卓越的跨平台能力继续大放异彩，让我们一起拥抱Java，拥抱新一代互联网技术，让世界变得更智慧！

陈昊鹏

前　　言

致读者

本书是按照 Java SE 11 进行更新的。卷 I 主要介绍了 Java 语言的一些关键特性，而本卷主要介绍编程人员进行专业软件开发时需要了解的高级主题。因此，与卷 I 和之前的版本一样，我们仍将本书定位于用 Java 技术进行实际项目开发的编程人员。

编写任何一本书籍都难免会有一些错误或不准确的地方。我们非常乐意听到读者的意见。当然，我们更希望对相同问题的报告只出现一次。为此，我们创建了一个 FAQ、bug 修正以及应急方案的网站 <http://horstmann.com/corejava>。你可以在 bug 报告网页的末尾处（鼓励读者阅读以前的报告）添加 bug 报告，以此来发布 bug 和问题并给出建议，以便我们提高本书将来版本的质量。

内容提要

本书中的章节大部分是相互独立的。你可以研究自己最感兴趣的主題，并可以按照任意顺序阅读这些章节。

在第 1 章中，你将学习 Java 的流库，它带来了现代风格的数据处理机制，即只需指定想要的结果，而无须详细描述应该如何获得该结果。这使得流库可以专注于优化的计算策略，对于优化并发计算来说，这显得特别有利。

第 2 章的主题是输入 / 输出处理。在 Java 中，所有 I/O 都是通过输入 / 输出流来处理的。这些流（不要与第 1 章的那些流混淆了）使你可以按照统一的方式来处理与各种数据源之间的通信，例如文件、网络连接或内存块。我们对各种读入器和写出器类进行了详细的讨论，它们使得对 Unicode 的处理变得很容易。我们还展示了如何使用对象序列化机制使保存和加载对象变得容易而方便，以及对象序列化机制背后的原理。然后，我们讨论了正则表达式以及如何操作文件与路径。该章通篇都包含了最新的 Java 版本中引入的广受欢迎的改进和优化。

第 3 章介绍 XML，展示怎样解析 XML 文件、怎样生成 XML 以及怎样使用 XSL 转换。在一个实用示例中，我们将展示怎样在 XML 中指定 Swing 窗体的布局。我们还讨论了 XPath API，它使得“在 XML 的干草堆中寻找绣花针”变得更加容易。

第 4 章介绍网络 API。Java 使复杂的网络编程工作变得很容易实现。我们将介绍怎样连接到服务器，怎样实现你自己的服务器，以及怎样创建 HTTP 连接。该章还讨论了新的 HTTP 客户端。

第 5 章介绍数据库编程，重点讲解 JDBC，即 Java 数据库连接 API，这是用于将 Java 程

序与关系数据库进行连接的 API。我们将介绍怎样通过使用 JDBC API 的核心子集，编写能够处理实际的数据库日常操作事务的实用程序。(如果要完整介绍 JDBC API 的功能，可能需要编写一本像本书一样厚的书才行。) 最后我们简要介绍了层次数据库，探讨了 JNDI (Java 命名及目录接口) 以及 LDAP (轻量级目录访问协议)。

Java 对于处理日期和时间的类库做出过两次设计，而在 Java 8 中做出的第三次设计则极富魅力。在第 6 章，你将学习如何使用新的日期和时间库来处理日历和时区的复杂性。

第 7 章讨论一个我们认为其重要性将会不断提升的特性——国际化。Java 编程语言是少数几种一开始就被设计为可以处理 Unicode 的语言之一，不过 Java 平台的国际化支持则走得更远。因此，你可以对 Java 应用程序进行国际化，使其不仅可以跨平台，而且还可以跨国界。例如，我们会展示怎样编写一个使用英语、德语和汉语的退休金计算器。

第 8 章讨论三种处理代码的技术。脚本机制和编译器 API 允许程序去调用使用诸如 JavaScript 或 Groovy 之类的脚本语言编写的代码，并且允许程序去编译 Java 代码。可以使用注解向 Java 程序中添加任意信息(有时称为元数据)。我们将展示注解处理器怎样在源码级别或者在类文件级别上收集注解，以及怎样运用注解来影响运行时的类行为。注解只有在工具的支持下才有用，因此，我们希望这些讨论能够帮助你根据需要选择有用的注解处理工具。

第 9 章讲解从 Java 9 开始引入的 Java 平台模块系统，以促进 Java 平台和核心类库的有序演化。这个模块系统提供了对包的封装和用于描述模块需求的机制。你将学习模块的属性，以便决定是否要在自己的应用程序中使用它们。即使你决定不使用，也需要了解这些新规则，这样你才能和 Java 平台以及其他模块化的类库交互。

第 10 章继续介绍 Java 安全模型。Java 平台一开始就是基于安全而设计的，该章会带你深入内部，查看这种设计是怎样实现的。我们将展示怎样编写用于特殊应用的类加载器和安全管理器。然后介绍允许使用消息、代码签名、授权和认证以及加密等重要特性的安全 API。最后，我们用一个使用 AES 和 RSA 加密算法的示例进行总结。

第 11 章讨论没有纳入卷 I 的所有 Swing 知识，尤其是重要但很复杂的树形构件和表格构件。我们还会介绍 Java 2D API，你可以用它来创建实际的图形和特殊的效果。当然，如今已经没有多少程序员需要编写 Swing 用户界面了，因此我们会将注意力放到在服务器端生成图像的实用特性上。

第 12 章介绍本地方法，这个功能支持你调用为微软 Windows API 这样的特殊机制而编写的各种方法。很显然，这种特性具有争议：使用本地方法，那么 Java 平台的跨平台特性将会随之消失。毫无疑问，每个为特定平台编写 Java 应用程序的专业开发人员都需要了解这些技术。有时，当你与不支持 Java 平台的设备或服务进行交互时，为了你的目标平台，你可能需要求助于操作系统 API。我们将通过展示如何从某个 Java 程序访问 Windows 注册表 API 来阐明这一点。

所有章节都按照最新版本的 Java 进行了修订，过时的材料都删除了，Java 9、10 和 11 的新 API 都详细地进行了讨论。

约定

我们使用等宽字体表示计算机代码，这种格式在众多的计算机书籍中极为常见。各种图标的含义如下：

 **注释：**需要引起注意的地方。

 **提示：**有用的提示。

 **警告：**关于缺陷或危险情况的警告信息。

 **C++ 注释：**本书中有许多这类提示，用于解释 Java 程序设计语言和 C++ 语言之间的不同。如果你对这部分不感兴趣，可以跳过。

Java 平台配备有大量的编程类库或者应用程序编程接口（API）。当第一次使用某个 API 时，我们在每一节的末尾都添加了简短的描述。这些描述可能有点不太规范，但是比官方在线 API 文档更具指导性。接口的名字以斜体显示，就像许多官方文档一样。类、接口或方法名后面的数字是 JDK 的版本，表示在该版本中才引入了相应的特性。

 **应用程序编程接口**

本书示例代码以程序清单的形式列举出来，例如：

程序清单 1-1 ScriptTest.java

可以从网站 <http://horstmann.com/corejava> 下载示例代码。

致 谢

写一本书需要投入大量的精力，升级一本书也并不像想象的那样轻松，尤其是 Java 技术一直在持续不断地更新。出版一本书会让很多人耗费很多心血，在此衷心地感谢《Java 核心技术》小组的每一位成员。

Pearson 公司的许多人提供了非常有价值的帮助，却甘愿做幕后英雄。在此，我希望大家都能够知道我对他们努力的感恩。与以往一样，我要真诚地感谢我的编辑 Greg Doench，从本书的写作到出版他一直在给予我们指导，同时感谢那些不知其姓名的为本书做出贡献的幕后人士。非常感谢 Julie Nahil 在图书制作方面给予的支持，感谢 Dmitry Kirsanov 和 Alina Kirsanova 完成手稿的编辑与排版工作。我还要感谢早期版本中我的合作者 Gary Cornell，他已经转向了其他行业。

感谢早期版本的许多读者，他们指出了许多令人尴尬的错误并给出了许多具有建设性的修改意见。我还要特别感谢本书优秀的审校小组，他们仔细地审阅我的手稿，使本书减少了许多错误。

这一版及以前版本是由以下人员评审的：Chuck Allison（特约编辑，*C/C++ Users Journal*）、Lance Anderson（Oracle）、Alec Beaton（PointBase, Inc.）、Cliff Berg（iSavvix Corporation）、Joshua Bloch、David Brown、Corky Cartwright、Frank Cohen（PushToTest）、Chris Crane（devXsolution）、Nicholas J. De Lillo 博士（曼哈顿学院）、Rakesh Dhoopar（Oracle）、Robert Evans（资深教师，约翰·霍普金斯大学应用物理实验室）、David Geary（Sabreware）、Jim Gish（Oracle）、Brian Goetz（Oracle）、Angela Gordon、Dan Gordon、Rob Gordon、John Gray（Hartford 大学）、Cameron Gregory（olabs.com）、Steve Haines、Marty Hall（约翰·霍普金斯大学应用物理实验室）、Vincent Hardy、Dan Harkey（圣何塞州立大学）、William Higgins（IBM）、Vladimir Ivanovic（PointBase）、Jerry Jackson（ChannelPoint Software）、Tim Kimmet（Preview Systems）、Chris Laffra、Charlie Lai、Angelika Langer、Doug Langston、Hang Lau（McGill 大学）、Mark Lawrence、Doug Lea（SUNY Oswego）、Gregory Longshore、Bob Lynch（Lynch Associates）、Philip Milne（顾问）、Mark Morrissey（俄勒冈研究生院）、Mahesh Neelakanta（佛罗里达大西洋大学）、Hao Pham、Paul Philion、Blake Ragsdell、Ylber Ramadani（Ryerson 大学）、Stuart Reges（亚利桑那大学）、Simon Ritter、Rich Rosen（Interactive Data Corporation）、Peter Sanders（法国尼斯 ESSI 大学）、Paul Sanghera 博士（圣何塞州立大学和布鲁克斯学院）、Paul Sevinc（Teamup AG）、Yoshiki Shabata、Devang Shah、Richard Slywczak（NASA/Glenn 研究中心）、Bradley A. Smith、Steven Stelting、Christopher Taylor、Luke Taylor（Valtech）、George Thiruvathukal、Kim Topley（*Core JFC, Second Edition* 的作者）、Janet Traub、Paul Tyma（顾问）、Christian Ullenboom、Peter van der Linden、Burt Walsh、Joe Wang（Oracle）和 Dan Xu（Oracle）。

Cay Horstmann
2018 年 12 月于加州旧金山

目 录

译者序

前言

致谢

第1章 Java 8 的流库	<i>1</i>
1.1 从迭代到流的操作	<i>1</i>
1.2 流的创建	<i>3</i>
1.3 filter、map 和 flatMap 方法	<i>8</i>
1.4 抽取子流和组合流	<i>9</i>
1.5 其他的流转换	<i>10</i>
1.6 简单约简	<i>11</i>
1.7 Optional 类型	<i>13</i>
1.7.1 获取 Optional 值	<i>13</i>
1.7.2 消费 Optional 值	<i>13</i>
1.7.3 管道化 Optional 值	<i>14</i>
1.7.4 不适合使用 Optional 值的方式	<i>15</i>
1.7.5 创建 Optional 值	<i>16</i>
1.7.6 用 flatMap 构建 Optional 值的函数	<i>16</i>
1.7.7 将 Optional 转换为流	<i>17</i>
1.8 收集结果	<i>19</i>
1.9 收集到映射表中	<i>24</i>
1.10 群组和分区	<i>27</i>
1.11 下游收集器	<i>28</i>
1.12 约简操作	<i>32</i>
1.13 基本类型流	<i>34</i>
1.14 并行流	<i>39</i>
第2章 输入与输出	<i>43</i>
2.1 输入 / 输出流	<i>43</i>
2.1.1 读写字节	<i>43</i>

2.1.2 完整的流家族	<i>46</i>
2.1.3 组合输入 / 输出流过滤器	<i>50</i>
2.1.4 文本输入与输出	<i>53</i>
2.1.5 如何写出文本输出	<i>53</i>
2.1.6 如何读入文本输入	<i>55</i>
2.1.7 以文本格式存储对象	<i>56</i>
2.1.8 字符编码方式	<i>59</i>
2.2 读写二进制数据	<i>61</i>
2.2.1 DataInput 和 DataOutput 接口	<i>61</i>
2.2.2 随机访问文件	<i>63</i>
2.2.3 ZIP 文档	<i>67</i>
2.3 对象输入 / 输出流与序列化	<i>70</i>
2.3.1 保存和加载序列化对象	<i>70</i>
2.3.2 理解对象序列化的文件格式	<i>74</i>
2.3.3 修改默认的序列化机制	<i>79</i>
2.3.4 序列化单例和类型安全的枚举	<i>81</i>
2.3.5 版本管理	<i>82</i>
2.3.6 为克隆使用序列化	<i>84</i>
2.4 操作文件	<i>86</i>
2.4.1 Path	<i>86</i>
2.4.2 读写文件	<i>89</i>
2.4.3 创建文件和目录	<i>90</i>
2.4.4 复制、移动和删除文件	<i>91</i>
2.4.5 获取文件信息	<i>92</i>
2.4.6 访问目录中的项	<i>94</i>
2.4.7 使用目录流	<i>95</i>
2.4.8 ZIP 文件系统	<i>98</i>
2.5 内存映射文件	<i>99</i>

2.5.1 内存映射文件的性能	99
2.5.2 缓冲区数据结构	105
2.6 文件加锁机制	107
2.7 正则表达式	109
2.7.1 正则表达式语法	109
2.7.2 匹配字符串	112
2.7.3 找出多个匹配	115
2.7.4 用分隔符来分割	117
2.7.5 替换匹配	117
第3章 XML	120
3.1 XML 概述	120
3.2 XML 文档的结构	122
3.3 解析 XML 文档	124
3.4 验证 XML 文档	133
3.4.1 文档类型定义	134
3.4.2 XML Schema	140
3.4.3 一个实践示例	142
3.5 使用 XPath 来定位信息	148
3.6 使用命名空间	152
3.7 流机制解析器	154
3.7.1 使用 SAX 解析器	154
3.7.2 使用 StAX 解析器	159
3.8 生成 XML 文档	162
3.8.1 不带命名空间的文档	162
3.8.2 带命名空间的文档	163
3.8.3 写出文档	163
3.8.4 使用 StAX 写出 XML 文档	165
3.8.5 示例：生成 SVG 文件	170
3.9 XSL 转换	171
第4章 网络	180
4.1 连接到服务器	180
4.1.1 使用 telnet	180
4.1.2 用 Java 连接到服务器	182
4.1.3 套接字超时	184
4.1.4 因特网地址	185
4.2 实现服务器	186
4.2.1 服务器套接字	186
4.2.2 为多个客户端服务	189
4.2.3 半关闭	192
4.2.4 可中断套接字	193
4.3 获取 Web 数据	199
4.3.1 URL 和 URI	199
4.3.2 使用 URLConnection 获取 信息	201
4.3.3 提交表单数据	207
4.4 HTTP 客户端	215
4.5 发送 E-mail	221
第5章 数据库编程	225
5.1 JDBC 的设计	225
5.1.1 JDBC 驱动程序类型	226
5.1.2 JDBC 的典型用法	227
5.2 结构化查询语言	227
5.3 JDBC 配置	232
5.3.1 数据库 URL	232
5.3.2 驱动程序 JAR 文件	233
5.3.3 启动数据库	233
5.3.4 注册驱动器类	234
5.3.5 连接到数据库	234
5.4 使用 JDBC 语句	237
5.4.1 执行 SQL 语句	237
5.4.2 管理连接、语句和 结果集	240
5.4.3 分析 SQL 异常	240
5.4.4 组装数据库	242
5.5 执行查询操作	246
5.5.1 预备语句	246
5.5.2 读写 LOB	252
5.5.3 SQL 转义	253
5.5.4 多结果集	254
5.5.5 获取自动生成的键	255
5.6 可滚动和可更新的结果集	256

5.6.1 可滚动的结果集	256	7.6.1 文本文件	327
5.6.2 可更新的结果集	258	7.6.2 行结束符	327
5.7 行集	261	7.6.3 控制台	328
5.7.1 构建行集	262	7.6.4 日志文件	328
5.7.2 被缓存的行集	262	7.6.5 UTF-8 字节顺序标志	329
5.8 元数据	265	7.6.6 源文件的字符编码	329
5.9 事务	274	7.7 资源包	330
5.9.1 用 JDBC 对事务编程	274	7.7.1 定位资源包	330
5.9.2 保存点	275	7.7.2 属性文件	331
5.9.3 批量更新	275	7.7.3 包类	332
5.9.4 高级 SQL 类型	277	7.8 一个完整的例子	333
5.10 Web 与企业应用中的连接 管理	278		
第 6 章 日期和时间 API	280	第 8 章 脚本、编译与注解处理	348
6.1 时间线	280	8.1 Java 平台的脚本机制	348
6.2 本地日期	284	8.1.1 获取脚本引擎	348
6.3 日期调整器	288	8.1.2 脚本计算与绑定	349
6.4 本地时间	289	8.1.3 重定向输入和输出	351
6.5 时区时间	290	8.1.4 调用脚本的函数和方法	352
6.6 格式化和解析	294	8.1.5 编译脚本	353
6.7 与遗留代码的互操作	298	8.1.6 示例：用脚本处理 GUI 事件	354
第 7 章 国际化	300	8.2 编译器 API	358
7.1 locale	300	8.2.1 调用编译器	358
7.1.1 为什么需要 locale	300	8.2.2 发起编译任务	359
7.1.2 指定 locale	301	8.2.3 捕获诊断消息	359
7.1.3 默认 locale	303	8.2.4 从内存中读取源文件	360
7.1.4 显示名字	304	8.2.5 将字节码写出到内存中	360
7.2 数字格式	305	8.2.6 示例：动态 Java 代码 生成	362
7.2.1 格式化数字值	306	8.3 使用注解	367
7.2.2 货币	310	8.3.1 注解简介	368
7.3 日期和时间	311	8.3.2 示例：注解事件处理器	369
7.4 排序和规范化	318	8.4 注解语法	373
7.5 消息格式化	323	8.4.1 注解接口	373
7.5.1 格式化数字和日期	324	8.4.2 注解	375
7.5.2 选择格式	325	8.4.3 注解各类声明	376
7.6 文本输入和输出	327	8.4.4 注解类型用法	377

8.4.5 注解 this	378
8.5 标准注解	379
8.5.1 用于编译的注解	380
8.5.2 用于管理资源的注解	381
8.5.3 元注解	381
8.6 源码级注解处理	383
8.6.1 注解处理器	384
8.6.2 语言模型 API	384
8.6.3 使用注解来生成源码	385
8.7 字节码工程	388
8.7.1 修改类文件	388
8.7.2 在加载时修改字节码	393
第 9 章 Java 平台模块系统	395
9.1 模块的概念	395
9.2 对模块命名	396
9.3 模块化的“Hello, World!” 程序	397
9.4 对模块的需求	398
9.5 导出包	400
9.6 模块化的 JAR	403
9.7 模块和反射式访问	404
9.8 自动模块	406
9.9 不具名模块	408
9.10 用于迁移的命令行标识	409
9.11 传递的需求和静态的需求	410
9.12 限定导出和开放	411
9.13 服务加载	412
9.14 操作模块的工具	414
第 10 章 安全	417
10.1 类加载器	417
10.1.1 类加载过程	418
10.1.2 类加载器的层次结构	419
10.1.3 将类加载器用作命名 空间	420
10.1.4 编写你自己的类加载器	421
10.1.5 字节码校验	426
10.2 安全管理器与访问权限	429
10.2.1 权限检查	429
10.2.2 Java 平台安全性	431
10.2.3 安全策略文件	434
10.2.4 定制权限	439
10.2.5 实现权限类	440
10.3 用户认证	446
10.3.1 JAAS 框架	446
10.3.2 JAAS 登录模块	451
10.4 数字签名	459
10.4.1 消息摘要	460
10.4.2 消息签名	463
10.4.3 校验签名	465
10.4.4 认证问题	467
10.4.5 证书签名	469
10.4.6 证书请求	469
10.4.7 代码签名	470
10.5 加密	472
10.5.1 对称密码	473
10.5.2 密钥生成	474
10.5.3 密码流	478
10.5.4 公共密钥密码	479
第 11 章 高级 Swing 和图形化 编程	483
11.1 表格	483
11.1.1 一个简单表格	483
11.1.2 表格模型	486
11.1.3 对行和列的操作	489
11.1.4 单元格的绘制和编辑	503
11.2 树	513
11.2.1 简单的树	514
11.2.2 节点枚举	526
11.2.3 绘制节点	528
11.2.4 监听树事件	530
11.2.5 定制树模型	536
11.3 高级 AWT	544

11.3.1 绘图操作流程	544	12.3 字符串参数	639
11.3.2 形状	546	12.4 访问域	644
11.3.3 区域	560	12.4.1 访问实例域	644
11.3.4 笔画	561	12.4.2 访问静态域	648
11.3.5 着色	567	12.5 编码签名	648
11.3.6 坐标变换	569	12.6 调用 Java 方法	650
11.3.7 剪切	574	12.6.1 实例方法	650
11.3.8 透明与组合	575	12.6.2 静态方法	653
11.4 像素图	583	12.6.3 构造器	654
11.4.1 图像的读取器和写入器	583	12.6.4 另一种方法调用	654
11.4.2 图像处理	591	12.7 访问数组元素	656
11.5 打印	604	12.8 错误处理	659
11.5.1 图形打印	604	12.9 使用调用 API	663
11.5.2 打印多页文件	612	12.10 完整的示例：访问 Windows 注册表	668
11.5.3 打印服务程序	620	12.10.1 Windows 注册表概述	668
11.5.4 流打印服务程序	622	12.10.2 访问注册表的 Java 平台 接口	669
11.5.5 打印属性	625	12.10.3 以本地方法实现注册表访问 函数	670
第 12 章 本地方法	632		
12.1 从 Java 程序中调用 C 函数	633		
12.2 数值参数与返回值	637		

第1章 Java 8 的流库

- ▲ 从迭代到流的操作
- ▲ 流的创建
- ▲ filter、map 和 flatMap 方法
- ▲ 抽取子流和组合流
- ▲ 其他的流转换
- ▲ 简单约简
- ▲ Optional 类型
- ▲ 收集结果
- ▲ 收集到映射表中
- ▲ 群组和分区
- ▲ 下游收集器
- ▲ 约简操作
- ▲ 基本类型流
- ▲ 并行流

与集合相比，流提供了一种可以让我们在更高的概念级别上指定计算任务的数据视图。通过使用流，我们可以说明想要完成什么任务，而不是说明如何去实现它。我们将操作的调度留给具体实现去解决。例如，假设我们想要计算某个属性的平均值，那么我们就可以指定数据源和该属性，然后，流库就可以对计算进行优化，例如，使用多线程来计算总和与个数，并将结果合并。

在本章中，你将会学习如何使用 Java 的流库，它是在 Java 8 中引入的，用来以“做什么而非怎么做”的方式处理集合。

1.1 从迭代到流的操作

在处理集合时，我们通常会迭代遍历它的元素，并在每个元素上执行某项操作。例如，假设我们想要对某本书中的所有长单词进行计数。首先，将所有单词放到一个列表中：

```
var contents = new String(Files.readAllBytes(  
    Paths.get("alice.txt")), StandardCharsets.UTF_8); // Read file into string  
List<String> words = List.of(contents.split("\\PL+"));  
// Split into words; nonletters are delimiters
```

现在，我们可以迭代它了：

```
int count = 0;  
for (String w : words) {  
    if (w.length() > 12) count++;  
}
```

在使用流时，相同的操作看起来像下面这样：

```
long count = words.stream()  
    .filter(w -> w.length() > 12)  
    .count();
```

现在我们不必扫描整个代码去查找过滤和计数操作，方法名就可以直接告诉我们其代码意欲何为。而且，循环需要非常详细地指定操作的顺序，而流却能够以其想要的任何方式来调度这些操作，只要结果是正确的即可。

仅将 `stream` 修改为 `parallelStream` 就可以让流库以并行方式来执行过滤和计数。

```
long count = words.parallelStream()
    .filter(w -> w.length() > 12)
    .count();
```

流遵循了“做什么而非怎么做”的原则。在流的示例中，我们描述了需要做什么：获取长单词，并对它们计数。我们没有指定该操作应该以什么顺序或者在哪个线程中执行。相比之下，本节开头处的循环要确切地指定计算应该如何工作，因此也就丧失了进行优化的机会。

流表面上看起来和集合很类似，都可以让我们转换和获取数据。但是，它们之间存在着显著的差异：

1. 流并不存储其元素。这些元素可能存储在底层的集合中，或者是按需生成的。
2. 流的操作不会修改其数据源。例如，`filter` 方法不会从流中移除元素，而是会生成一个新的流，其中不包含被过滤掉的元素。
3. 流的操作是尽可能惰性执行的。这意味着直至需要其结果时，操作才会执行。例如，如果我们只想查找前 5 个长单词而不是所有长单词，那么 `filter` 方法就会在匹配到第 5 个单词后停止过滤。因此，我们甚至可以操作无限流。

我们再来看看这个示例。`stream` 和 `parallelStream` 方法会产生一个用于 `words` 列表的流。`filter` 方法会返回另一个流，其中只包含长度大于 12 的单词。`count` 方法会将这个流化简为一个结果。

这个工作流是操作流时的典型流程。我们建立了一个包含三个阶段的操作管道：

1. 创建一个流。
2. 指定将初始流转换为其他流的中间操作，可能包含多个步骤。
3. 应用终止操作，从而产生结果。这个操作会强制执行之前的惰性操作。从此之后，这个流就再也不能用了。

在程序清单 1-1 的示例中，流是用 `stream` 或 `parallelStream` 方法创建的。`filter` 方法对其进行转换，而 `count` 方法是终止操作。

程序清单 1-1 streams/CountLongWords.java

```
1 package streams;
2
3 /**
4  * @version 1.01 2018-05-01
5  * @author Cay Horstmann
6 */
7
8 import java.io.*;
9 import java.nio.charset.*;
10 import java.nio.file.*;
```

```
11 import java.util.*;  
12  
13 public class CountLongWords  
14 {  
15     public static void main(String[] args) throws IOException  
16     {  
17         var contents = new String(Files.readAllBytes(  
18             Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);  
19         List<String> words = List.of(contents.split("\\PL+"));  
20  
21         long count = 0;  
22         for (String w : words)  
23         {  
24             if (w.length() > 12) count++;  
25         }  
26         System.out.println(count);  
27  
28         count = words.stream().filter(w -> w.length() > 12).count();  
29         System.out.println(count);  
30  
31         count = words.parallelStream().filter(w -> w.length() > 12).count();  
32         System.out.println(count);  
33     }  
34 }
```

在下一节中，你将会看到如何创建流。后续的三节将讨论流的转换。再后面的五节将讨论终止操作。

API `java.util.stream.Stream<T>` 8

- `Stream<T> filter(Predicate<? super T> p)`
产生一个流，其中包含当前流中满足 p 的所有元素。
- `long count()`
产生当前流中元素的数量。这是一个终止操作。

API `java.util.Collection<E>` 1.2

- `default Stream<E> stream()`
- `default Stream<E> parallelStream()`
产生当前集合中所有元素的顺序流或并行流。

1.2 流的创建

你已经看到了可以用 `Collection` 接口的 `stream` 方法将任何集合转换为一个流。如果你有一个数组，那么可以使用静态的 `Stream.of` 方法。

```
Stream<String> words = Stream.of(contents.split("\\PL+"));  
// split returns a String[] array
```

`of` 方法具有可变长参数，因此我们可以构建具有任意数量引元的流：

```
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

使用 `Array.stream(array, from, to)` 可以用数组中的一部分元素来创建一个流。

为了创建不包含任何元素的流，可以使用静态的 `Stream.empty` 方法：

```
Stream<String> silence = Stream.empty();
// Generic type <String> is inferred; same as Stream.<String>empty()
```

`Stream` 接口有两个用于创建无限流的静态方法。`generate` 方法会接受一个不包含任何引元的函数（或者从技术上讲，是一个 `Supplier<T>` 接口的对象）。无论何时，只要需要一个流类型的值，该函数就会被调用以产生一个这样的值。我们可以像下面这样获得一个常量值的流：

```
Stream<String> echos = Stream.generate(() -> "Echo");
```

或者像下面这样获取一个随机数的流：

```
Stream<Double> randoms = Stream.generate(Math::random);
```

如果要产生像 0 1 2 3 … 这样的序列，可以使用 `iterate` 方法。它会接受一个“种子”值，以及一个函数（从技术上讲，是一个 `UnaryOperation<T>`），并且会反复地将该函数应用到之前的结果上。例如，

```
Stream<BigInteger> integers
= Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

该序列中的第一个元素是种子 `BigInteger.ZERO`，第二个元素是 `f(seed)`，即 1（作为大整数），下一个元素是 `f(f(seed))`，即 2，后续以此类推。

如果要产生一个有限序列，则需要添加一个谓词来描述迭代应该如何结束：

```
var limit = new BigInteger("10000000");
Stream<BigInteger> integers
= Stream.iterate(BigInteger.ZERO,
n -> n.compareTo(limit) < 0,
n -> n.add(BigInteger.ONE));
```

只要该谓词拒绝了某个迭代生成的值，这个流即结束。

最后，`Stream.ofNullable` 方法会用一个对象来创建一个非常短的流。如果该对象为 `null`，那么这个流的长度就为 0；否则，这个流的长度为 1，即只包含该对象。这个方法与 `flatMap` 相结合时最有用，可以查看 1.7.7 节中的示例。

 **注释：**Java API 中有大量方法都可以产生流。例如，`Pattern` 类有一个 `splitAsStream` 方法，它会按照某个正则表达式来分割一个 `CharSequence` 对象。可以使用下面的语句来将一个字符串分割为一个个的单词：

```
Stream<String> words = Pattern.compile("\\PL+").splitAsStream(contents);
```

`Scanner.tokens` 方法会产生一个扫描器的符号流。另一种从字符串中获取单词流的方式是：

```
Stream<String> words = new Scanner(contents).tokens();
```

静态的 `Files.lines` 方法会返回一个包含了文件中所有行的 Stream:

```
try (Stream<String> lines = Files.lines(path)) {
    Process lines
}
```

注释: 如果我们持有的 Iterable 对象不是集合, 那么可以通过下面的调用将其转换为一个流:

```
StreamSupport.stream(iterable.iterator(), false);
```

如果我们持有的是 Iterator 对象, 并且希望得到一个由它的结果构成的流, 那么可以使用下面的语句:

```
StreamSupport.stream(Spliterators.spliteratorUnknownSize(
    iterator, Spliterator.ORDERED), false);
```

警告: 至关重要的是, 在执行流的操作时, 我们并没有修改流背后的集合。记住, 流并没有收集其数据, 数据一直存储在单独的集合中。如果修改了该集合, 那么流操作的结果就会变成未定义的。JDK 文档称这种要求为不干涉性。

准确地讲, 因为中间的流操作是惰性的, 所以在终止操作得以执行时, 集合有可能已经发生了变化。例如, 尽管我们不推荐下面这段代码, 但是它仍旧可以工作:

```
List<String> wordList = . . .;
Stream<String> words = wordList.stream();
wordList.add("END");
long n = words.distinct().count();
```

但是下面的代码是错误的:

```
Stream<String> words = wordList.stream();
words.forEach(s -> if (s.length() < 12) wordList.remove(s));
// ERROR--interference
```

程序清单 1-2 中的示例程序展示了创建流的各种方式。

程序清单 1-2 streams/CreatingStreams.java

```
1 package streams;
2
3 /**
4  * @version 1.01 2018-05-01
5  * @author Cay Horstmann
6  */
7
8 import java.io.IOException;
9 import java.math.BigInteger;
10 import java.nio.charset.StandardCharsets;
11 import java.nio.file.*;
12 import java.util.*;
```

```
13 import java.util.regex.Pattern;
14 import java.util.stream.*;
15
16 public class CreatingStreams
17 {
18     public static <T> void show(String title, Stream<T> stream)
19     {
20         final int SIZE = 10;
21         List<T> firstElements = stream
22             .limit(SIZE + 1)
23             .collect(Collectors.toList());
24         System.out.print(title + ": ");
25         for (int i = 0; i < firstElements.size(); i++)
26         {
27             if (i > 0) System.out.print(", ");
28             if (i < SIZE) System.out.print(firstElements.get(i));
29             else System.out.print("...");  

30         }
31         System.out.println();
32     }
33
34     public static void main(String[] args) throws IOException
35     {
36         Path path = Paths.get("../gutenberg/alice30.txt");
37         var contents = new String(Files.readAllBytes(path), StandardCharsets.UTF_8);
38
39         Stream<String> words = Stream.of(contents.split("\\PL+"));
40         show("words", words);
41         Stream<String> song = Stream.of("gently", "down", "the", "stream");
42         show("song", song);
43         Stream<String> silence = Stream.empty();
44         show("silence", silence);
45
46         Stream<String> echos = Stream.generate(() -> "Echo");
47         show("echos", echos);
48
49         Stream<Double> randoms = Stream.generate(Math::random);
50         show("randoms", randoms);
51
52         Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
53             n -> n.add(BigInteger.ONE));
54         show("integers", integers);
55
56         Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStream(contents);
57         show("wordsAnotherWay", wordsAnotherWay);
58
59         try (Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8))
60         {
61             show("lines", lines);
62         }
63
64         Iterable<Path> iterable = FileSystems.getDefault().getRootDirectories();
65         Stream<Path> rootDirectories = StreamSupport.stream(iterable.spliterator(), false);
66         show("rootDirectories", rootDirectories);
```

```

67     Iterator<Path> iterator = Paths.get("/usr/share/dict/words").iterator();
68     Stream<Path> pathComponents = StreamSupport.stream(Spliterators.spliteratorUnknownSize(
69         iterator, Spliterator.ORDERED), false);
70     show("pathComponents", pathComponents);
71 }
72 }
```

API java.util.stream.Stream 8

- static <T> Stream<T> of(T... values)
产生一个元素为给定值的流。
- static <T> Stream<T> empty()
产生一个不包含任何元素的流。
- static <T> Stream<T> generate(Supplier<T> s)
产生一个无限流，它的值是通过反复调用函数 s 而构建的。
- static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)
● static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> f)
产生一个无限流，它的元素包含 seed、在 seed 上调用 f 产生的值、在前一个元素上调用 f 产生的值，等等。第一个方法会产生一个无限流，而第二个方法的流会在碰到第一个不满足 hasNext 谓词的元素时终止。
- static <T> Stream<T> ofNullable(T t) 9
如果 t 为 null，返回一个空流，否则返回包含 t 的流。

API java.util.Spliterators 8

- static <T> Spliterator<T> spliteratorUnknownSize(Iterator<? extends T> iterator, int characteristics)
用给定的特性（一种包含诸如 Spliterator.ORDERED 之类的常量的位模式）将一个迭代器转换为一个具有未知尺寸的可分割的迭代器。

API java.util.Arrays 1.2

- static <T> Stream<T> stream(T[] array, int startInclusive, int endExclusive) 8
产生一个流，它的元素是由数组中指定范围内的元素构成的。

API java.util.regex.Pattern 1.4

- Stream<String> splitAsStream(CharSequence input) 8
产生一个流，它的元素是输入中由该模式界定的部分。

API java.nio.file.Files 7

- static Stream<String> lines(Path path) 8
- static Stream<String> lines(Path path, Charset cs) 8

产生一个流，它的元素是指定文件中的行，该文件的字符集为 UTF-8，或者为指定的字符集。

API `java.util.stream.StreamSupport` 8

- `static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel)`

产生一个流，它包含了由给定的可分割迭代器产生的值。

API `java.lang.Iterable` 5

- `Spliterator<T> spliterator()` 8

为这个 Iterable 产生一个可分割的迭代器。默认实现不分割也不报告尺寸。

API `java.util.Scanner` 5

- `public Stream<String> tokens()` 9

产生一个字符串流，该字符串是调用这个扫描器的 next 方法时返回的。

API `java.util.function.Supplier<T>` 8

- `T get()`

提供一个值。

1.3 filter、map 和 flatMap 方法

流的转换会产生一个新的流，它的元素派生自另一个流中的元素。我们已经看到了 filter 转换会产生一个新流，它的元素与某种条件相匹配。下面，我们将一个字符串流转换为只包含长单词的另一个流：

```
List<String> words = . . .;
Stream<String> longWords = words.stream().filter(w -> w.length() > 12);
```

filter 的引元是 `Predicate<T>`，即从 T 到 boolean 的函数。

通常，我们想要按照某种方式来转换流中的值，此时，可以使用 map 方法并传递执行该转换的函数。例如，我们可以像下面这样将所有单词都转换为小写：

```
Stream<String> lowercaseWords = words.stream().map(String::toLowerCase);
```

这里，我们使用的是带有方法引用的 map，但是，通常我们可以使用 lambda 表达式来代替：

```
Stream<String> firstLetters = words.stream().map(s -> s.substring(0, 1));
```

上面的语句所产生的流中包含了所有单词的首字母。

在使用 map 时，会有一个函数应用到每个元素上，并且其结果是包含了应用该函数后所产生的所有结果的流。现在，假设我们有一个函数，它返回的不是一个值，而是一个包含众多值的流。下面的示例展示的方法会将字符串转换为字符串流，即一个个的编码点：

```
public static Stream<String> codePoints(String s)
{
    var result = new ArrayList<String>();
```

```

int i = 0;
while (i < s.length())
{
    int j = s.offsetByCodePoints(i, 1);
    result.add(s.substring(i, j));
    i = j;
}
return result.stream();
}

```

这个方法可以正确地处理需要用两个 `char` 值来表示的 Unicode 字符，因为本来就应该这样处理。但是，我们不用再次纠结其细节。

例如，`codePoints("boat")` 的返回值是流 `["b", "o", "a", "t"]`。

假设我们将 `codePoints` 方法映射到一个字符串流上：

```
Stream<Stream<String>> result = words.stream().map(w -> codePoints(w));
```

那么会得到一个包含流的流，就像 `[...["y", "o", "u", "r"], ["b", "o", "a", "t"], ...]`。为了将其摊平为单个流 `[... "y", "o", "u", "r", "b", "o", "a", "t", ...]`，可以使用 `flatMap` 方法而不是 `map` 方法：

```
Stream<String> flatResult = words.stream().flatMap(w -> codePoints(w));
// Calls codePoints on each word and flattens the results
```

 **注释：**在流之外的类中你也会发现 `flatMap` 方法，因为它是计算机科学中的一种通用概念。假设我们有一个泛型 G (例如 `Stream`)，以及将某种类型 T 转换为 G<U> 的函数 f 和将类型 U 转换为 G<V> 的函数 g。我们可以通过使用 `flatMap` 来组合它们，即首先应用 f，然后应用 g。这是单子论的关键概念。但是不必担心，我们无须了解任何有关单子的知识就可以使用 `flatMap`。

API `java.util.stream.Stream` 8

- `Stream<T> filter(Predicate<? super T> predicate)`
产生一个流，它包含当前流中所有满足谓词条件的元素。
- `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
产生一个流，它包含将 `mapper` 应用于当前流中所有元素所产生的结果。
- `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`
产生一个流，它是通过将 `mapper` 应用于当前流中所有元素所产生的结果连接到一起而获得的。(注意，这里的每个结果都是一个流。)

1.4 抽取子流和组合流

调用 `stream.limit(n)` 会返回一个新的流，它在 n 个元素之后结束 (如果原来的流比 n 短，那么就会在该流结束时结束)。这个方法对于裁剪无限流的尺寸特别有用。例如，

```
Stream<Double> randoms = Stream.generate(Math::random).limit(100);
```

会产生一个包含 100 个随机数的流。

调用 `stream.skip(n)` 正好相反：它会丢弃前 `n` 个元素。这个方法对于本书的读操作示例很方便，因为按照 `split` 方法的工作方式，第一个元素是没什么用的空字符串。我们可以通过调用 `skip` 来跳过它：

```
Stream<String> words = Stream.of(contents.split("\\PL+")).skip(1);
```

`stream.takeWhile(predicate)` 调用会在谓词为真时获取流中的所有元素，然后停止。

例如，假设我们使用上一节的 `codePoints` 方法将字符串分割为字符，然后收集所有的数字元素。`takeWhile` 方法可以实现此目标：

```
Stream<String> initialDigits = codePoints(str).takeWhile(
    s -> "0123456789".contains(s));
```

`dropWhile` 方法的做法正好相反，它会在条件为真时丢弃元素，并产生一个由第一个使该条件为假的字符开始的所有元素构成的流：

```
Stream<String> withoutInitialWhiteSpace = codePoints(str).dropWhile(
    s -> s.trim().length() == 0);
```

我们可以用 `Stream` 类的静态 `concat` 方法将两个流连接起来：

```
Stream<String> combined = Stream.concat(
    codePoints("Hello"), codePoints("World"));
// Yields the stream ["H", "e", "l", "l", "o", "W", "o", "r", "l", "d"]
```

当然，第一个流不应该是无限的，否则第二个流永远都不会有机会处理。

API `java.util.stream.Stream` 8

- `Stream<T> limit(long maxSize)`
产生一个流，其中包含了当前流中最初的 `maxSize` 个元素。
- `Stream<T> skip(long n)`
产生一个流，它的元素是当前流中除了前 `n` 个元素之外的所有元素。
- `Stream<T> takeWhile(Predicate<? super T> predicate) 9`
产生一个流，它的元素是当前流中所有满足谓词条件的元素。
- `Stream<T> dropWhile(Predicate<? super T> predicate) 9`
产生一个流，它的元素是当前流中排除不满足谓词条件的元素之外的所有元素。
- `static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)`
产生一个流，它的元素是 `a` 的元素后面跟着 `b` 的元素。

1.5 其他的流转换

`distinct` 方法会返回一个流，它的元素是从原有流中产生的，即原来的元素按照同样的顺序剔除重复元素后产生的。这些重复元素并不一定是毗邻的。

```
Stream<String> uniqueWords
= Stream.of("merrily", "merrily", "merrily", "gently").distinct();
// Only one "merrily" is retained
```

对于流的排序，有多种 `sorted` 方法的变体可用。其中一种用于操作 `Comparable` 元素的流，而另一种可以接受一个 `Comparator`。下面，我们对字符串排序，使得最长的字符串排在最前面：

```
Stream<String> longestFirst
    = words.stream().sorted(Comparator.comparing(String::length).reversed());
```

与所有的流转换一样，`sorted` 方法会产生一个新的流，它的元素是原有流中按照顺序排列的元素。

当然，我们在对集合排序时可以不使用流。但是，当排序处理是流管道的一部分时，`sorted` 方法就会显得很有用。

最后，`peek` 方法会产生另一个流，它的元素与原来流中的元素相同，但是在每次获取一个元素时，都会调用一个函数。这对于调试来说很方便：

```
Object[] powers = Stream.iterate(1.0, p -> p * 2)
    .peek(e -> System.out.println("Fetching " + e))
    .limit(20).toArray();
```

当实际访问一个元素时，就会打印出来一条消息。通过这种方式，你可以验证 `iterate` 返回的无限流是被惰性处理的。

 **提示：**当我们使用调试器来调试流的计算程序时，可以针对各个流转换操作中的某一个，在它所调用的方法中设置断点。对于大多数 IDE，我们都可以在 lambda 表达式中设置断点。如果只想了解在流管道的某个特定点上会发生什么，那么可以添加下面的代码，并在第二行上设置断点：

```
.peek(x -> {
    return; })
```

API `java.util.stream.Stream` 8

- `Stream<T> distinct()`

产生一个流，包含当前流中所有不同的元素。

- `Stream<T> sorted()`

- `Stream<T> sorted(Comparator<? super T> comparator)`

产生一个流，它的元素是当前流中的所有元素按照顺序排列的。第一个方法要求元素是实现了 `Comparable` 的类的实例。

- `Stream<T> peek(Consumer<? super T> action)`

产生一个流，它与当前流中的元素相同，在获取其中每个元素时，会将其传递给 `action`。

1.6 简单约简

现在你已经看到了如何创建和转换流，我们终于可以讨论最重要的内容了，即从流数据中获得答案。我们在本节所讨论的方法被称为约简。约简是一种终结操作（terminal

operation)，它们会将流约简为可以在程序中使用的非流值。

你已经看到过一种简单约简：count 方法会返回流中元素的数量。

其他的简单约简还有 max 和 min，它们分别返回最大值和最小值。这里要稍作解释，这些方法返回的是一个类型 Optional<T> 的值，它要么在其中包装了答案，要么表示没有任何值（因为流碰巧为空）。在过去，碰到这种情况返回 null 是很常见的，但是这样做会导致在未做完备测试的程序中产生空指针异常。Optional 类型是一种表示缺少返回值的更好的方式。我们将在下一节中详细讨论 Optional 类型。下面展示了如何获得流中的最大值：

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
System.out.println("largest: " + largest.orElse "");
```

findFirst 返回的是非空集合中的第一个值。它通常在与 filter 组合使用时很有用。例如，下面展示了如何找到第一个以字母 Q 开头的单词，前提是存在这样的单词：

```
Optional<String> startsWithQ
= words.filter(s -> s.startsWith("Q")).findFirst();
```

如果不强调使用第一个匹配，而是使用任意的匹配都可以，那么就可以使用 findAny 方法。这个方法在并行处理流时很有效，因为流可以报告任何它找到的匹配而不是被限制为必须报告第一个匹配。

```
Optional<String> startsWithQ
= words.parallel().filter(s -> s.startsWith("Q")).findAny();
```

如果只想知道是否存在匹配，那么可以使用 anyMatch。这个方法会接受一个断言引元，因此不需要使用 filter。

```
boolean aWordStartsWithQ
= words.parallel().anyMatch(s -> s.startsWith("Q"));
```

还有 allMatch 和 noneMatch 方法，它们分别在所有元素和没有任何元素匹配谓词的情况下返回 true。这些方法也可以通过并行运行而获益。

API `java.util.stream.Stream` 8

- `Optional<T> max(Comparator<? super T> comparator)`
- `Optional<T> min(Comparator<? super T> comparator)`

分别产生这个流的最大元素和最小元素，使用由给定比较器定义的排序规则，如果这个流为空，会产生一个空的 Optional 对象。这些操作都是终结操作。

- `Optional<T> findFirst()`
- `Optional<T> findAny()`

分别产生这个流的第一个和任意一个元素，如果这个流为空，会产生一个空的 Optional 对象。这些操作都是终结操作。

- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`

分别在这个流中任意元素、所有元素和没有任何元素匹配给定谓词时返回 true。这些操作都是终结操作。

1.7 Optional 类型

`Optional<T>` 对象是一种包装器对象，要么包装了类型 `T` 的对象，要么没有包装任何对象。对于第一种情况，我们称这种值是存在的。`Optional<T>` 类型被当作一种更安全的方式，用来替代类型 `T` 的引用，这种引用要么引用某个对象，要么为 `null`。但是，它只有在正确使用的情况下才会更安全，接下来的三个小节我们将讨论如何正确使用。

1.7.1 获取 Optional 值

有效地使用 `Optional` 的关键是要使用这样的方法：它在值不存在的情况下会产生一个可替代物，而只有在值存在的情况下才会使用这个值。

本小节我们先来看看第一条策略。通常，在没有任何匹配时，我们会希望使用某种默认值，可能是空字符串：

```
String result = optionalString.orElse("");
// The wrapped string, or "" if none
```

你还可以调用代码来计算默认值：

```
String result = optionalString.orElseGet(() -> System.getProperty("myapp.default"));
// The function is only called when needed
```

或者可以在没有任何值时抛出异常：

```
String result = optionalString.orElseThrow(IllegalStateException::new);
// Supply a method that yields an exception object
```

API `java.util.Optional` 8

- `T orElse(T other)`
产生这个 `Optional` 的值，或者在该 `Optional` 为空时，产生 `other`。
- `T orElseGet(Supplier<? extends T> other)`
产生这个 `Optional` 的值，或者在该 `Optional` 为空时，产生调用 `other` 的结果。
- `<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)`
产生这个 `Optional` 的值，或者在该 `Optional` 为空时，抛出调用 `exceptionSupplier` 的结果。

1.7.2 消费 Optional 值

在上一小节，我们看到了如何在不存在任何值的情况下产生相应的替代物。另一条使用可选值的策略是只有在其存在的情况下才消费该值。

`ifPresent` 方法会接受一个函数。如果可选值存在，那么它会被传递给该函数。否则，不会发生任何事情。

```
optionalValue.ifPresent(v -> Process v);
```

例如，如果在该值存在的情况下想要将其添加到某个集中，那么就可以调用

```
optionalValue.ifPresent(v -> results.add(v));
```

或者直接调用

```
optionalValue.ifPresent(results::add);
```

如果想要在可选值存在时执行一种动作，在可选值不存在时执行另一种动作，可以使用 ifPresentOrElse：

```
optionalValue.ifPresentOrElse(
    v -> System.out.println("Found " + v),
    () -> logger.warning("No match"));
```

API `java.util.Optional` 8

- `void ifPresent(Consumer<? super T> action)`

如果该 Optional 不为空，就将它的值传递给 action。

- `void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction) 9`

如果该 Optional 不为空，就将它的值传递给 action，否则调用 emptyAction。

1.7.3 管道化 Optional 值

在上一节中，你看到了如何从 Optional 对象获取值。另一种有用的策略是保持 Optional 完整，使用 map 方法来转换 Optional 内部的值：

```
Optional<String> transformed = optionalString.map(String::toUpperCase);
```

如果 optionalString 为空，那么 transformed 也为空。

下面是另一个例子，我们将一个结果添加到列表中，如果它存在的话：

```
optionalValue.map(results::add);
```

如果 optionalValue 为空，则什么也不会发生。

注释：这个 map 方法与 1.3 节中描述的 Stream 接口的 map 方法类似。你可以直接将可选值想象成尺寸为 0 或 1 的流。结果的尺寸也是 0 或 1，并且在后一种情况中，函数会应用于其上。

类似地，可以使用 filter 方法来只处理那些在转换它之前或之后满足某种特定属性的 Optional 值。如果不满足该属性，那么管道会产生空的结果：

```
Optional<String> transformed = optionalString
    .filter(s -> s.length() >= 8)
    .map(String::toUpperCase);
```

你也可以用 or 方法将空 Optional 替换为一个可替代的 Optional。这个可替代值将以惰性方式计算。

```
Optional<String> result = optionalString.or(() -> // Supply an Optional
    alternatives.stream().findFirst());
```

如果 `optionalString` 的值存在，那么 `result` 为 `optionalString`。如果值不存在，那么就会计算 lambda 表达式，并使用计算出来的结果。

API `java.util.Optional` 8

- <U> `Optional<U> map(Function<? super T, ? extends U> mapper)`

产生一个 `Optional`，如果当前的 `Optional` 的值存在，那么所产生的 `Optional` 的值是通过将给定的函数应用于当前的 `Optional` 的值而得到的；否则，产生一个空的 `Optional`。

- `Optional<T> filter(Predicate<? super T> predicate)`

产生一个 `Optional`，如果当前的 `Optional` 的值满足给定的谓词条件，那么所产生的 `Optional` 的值就是当前 `Optional` 的值；否则，产生一个空 `Optional`。

- `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier) 9`

如果当前 `Optional` 不为空，则产生当前的 `Optional`；否则由 `supplier` 产生一个 `Optional`。

1.7.4 不适合使用 `Optional` 值的方式

如果没有正确地使用 `Optional` 值，那么相比以往得到“某物或 `null`”的方式，你并没有得到任何好处。

`get` 方法会在 `Optional` 值存在的情况下获得其中包装的元素，或者在不存在的情况下抛出一个 `NoSuchElementException` 异常。因此，

```
Optional<T> optionalValue = . . .;
optionalValue.get().someMethod()
```

并不比下面的方式更安全：

```
T value = . . .;
value.someMethod();
```

`isPresent` 方法会报告某个 `Optional<T>` 对象是否具有值。但是

```
if (optionalValue.isPresent()) optionalValue.get().someMethod();
```

并不比下面的方式更容易处理：

```
if (value != null) value.someMethod();
```

注释：Java 10 为 `get` 方法引入了一个耸人听闻的同义词，称为 `optionalValue.orElseThrow()`，这个名字明确表示该方法会在 `optionalValue` 为空时抛出一个 `NoSuchElementException`。这样命名是希望程序员只有在非常明确地知道 `Optional` 永远都不会为空时才去调用该方法。

下面是一些有关 `Optional` 类型正确用法的提示：

- `Optional` 类型的变量永远都不应该为 `null`。
- 不要使用 `Optional` 类型的域。因为其代价是额外多出来一个对象。在类的内部，使用 `null` 表示缺失的域更易于操作。

- 不要在集合中放置 Optional 对象，并且不要将它们用作 map 的键。应该直接收集其中的值。

API java.util.Optional 8

- `T get()`
- `T orElseThrow() 10`
产生这个 Optional 的值，或者在该 Optional 为空时，抛出一个 NoSuchElementException 异常。
- `boolean isPresent()`
如果该 Optional 不为空，则返回 true。

1.7.5 创建 Optional 值

到目前为止，我们已经讨论了如何使用其他人创建的 Optional 对象。如果想要编写方法来创建 Optional 对象，那么有多个方法可以用于此目的，包括 `Optional.of(result)` 和 `Optional.empty()`。例如，

```
public static Optional<Double> inverse(Double x)
{
    return x == 0 ? Optional.empty() : Optional.of(1 / x);
}
```

`ofNullable` 方法被用来作为可能出现的 null 值和可选值之间的桥梁。`Optional.ofNullable(obj)` 会在 obj 不为 null 的情况下返回 `Optional.of(obj)`，否则会返回 `Optional.empty()`。

API java.util.Optional 8

- `static <T> Optional<T> of(T value)`
- `static <T> Optional<T> ofNullable(T value)`
产生一个具有给定值的 Optional。如果 value 为 null，那么第一个方法会抛出一个 `NullPointerException` 异常，而第二个方法会产生一个空 Optional。
- `static <T> Optional<T> empty()`
产生一个空 Optional。

1.7.6 用 flatMap 构建 Optional 值的函数

假设你有一个可以产生 `Optional<T>` 对象的方法 f，并且目标类型 T 具有一个可以产生 `Optional<U>` 对象的方法 g。如果它们都是普通的方法，那么你可以通过调用 `s.f().g()` 来将它们组合起来。但是这种组合无法工作，因为 `s.f()` 的类型为 `Optional<T>`，而不是 T。因此，需要调用

```
Optional<U> result = s.f().flatMap(T::g);
```

如果 `s.f()` 的值存在，那么 g 就可以应用到它上面。否则，就会返回一个空 `Optional<U>`。

很明显，如果有更多可以产生 Optional 值的方法或 lambda 表达式，那么就可以重复此过程。你可以直接将对 `flatMap` 的调用链接起来，从而构建由这些步骤构成的管道，只有所有步

骤都成功，该管道才会成功。

例如，考虑前一节中安全的 `inverse` 方法。假设我们还有一个安全的平方根：

```
public static Optional<Double> squareRoot(Double x)
{
    return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
}
```

那么你可以像下面这样计算倒数的平方根：

```
Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);
```

或者，你可以选择下面的方式：

```
Optional<Double> result
= Optional.of(-4.0).flatMap(Demo::inverse).flatMap(Demo::squareRoot);
```

无论是 `inverse` 方法还是 `squareRoot` 方法返回 `Optional.empty()`，整个结果都会为空。

注释：你已经在 `Stream` 接口中看到过 `flatMap` 方法（参见 1.3 节），当时这个方法被用来将产生流的两个方法组合起来，其实现方式是摊平由流构成的流。如果将可选值解释为具有 0 个或 1 个元素，那么 `Optional.flatMap` 方法与其操作方式一样。

API `java.util.Optional` 8

- `<U> Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper)`

如果 `Optional` 存在，产生将 `mapper` 应用于当前 `Optional` 值所产生的结果，或者在当前 `Optional` 为空时，返回一个空 `Optional`。

1.7.7 将 `Optional` 转换为流

`stream` 方法会将一个 `Optional<T>` 对象转换为一个具有 0 个或 1 个元素的 `Stream<T>` 对象。这种做法看起来很自然，但是我们为什么希望这么做呢？

这会使返回 `Optional` 结果的方法变得很有用。假设我们有一个用户 ID 流和下面的方法：

```
Optional<User> lookup(String id)
```

怎样才能在获取用户流时，跳过那些无效的 ID 呢？

当然，我们可以过滤掉无效 ID，然后将 `get` 方法应用于剩余的 ID：

```
Stream<String> ids = . . .;
Stream<User> users = ids.map(Users::lookup)
    .filter(Optional::isPresent)
    .map(Optional::get);
```

但是这样就需要使用我们之前警告过要慎用的 `isPresent` 和 `get` 方法。下面的调用显得更优雅：

```
Stream<User> users = ids.map(Users::lookup)
    .flatMap(Optional::stream);
```

每一个对 `stream` 的调用都会返回一个具有 0 个或 1 个元素的流。`flatMap` 方法将这些方法

组合在一起，这意味着不存在的用户会直接被丢弃。

注释：本节我们研究了一些令人愉快的场景，在其中我们拥有可以返回 Optional 值的方法。当前，许多方法都会在没有任何有效结果的情况下返回 null。假设 Users.classicLookup(id) 会返回一个 User 对象或者 null，而不是 Optional<User>，我们当然可以过滤掉 null 值：

```
Stream<User> users = ids.map(Users::classicLookup)
    .filter(Objects::nonNull);
```

但是如果更喜欢 flatMap 的方式，那么我们可以使用下面的代码：

```
Stream<User> users = ids.flatMap(
    id -> Stream.ofNullable(Users.classicLookup(id)));
```

或者是下面的代码：

```
Stream<User> users = ids.map(Users::classicLookup)
    .flatMap(Stream::ofNullable);
```

Stream.ofNullable(obj) 这个调用在 obj 为 null 时，会产生一个空的流，否则会产生一个只包含 obj 的流。

程序清单 1-3 中的示例程序演示了 Optional API 的使用方式。

程序清单 1-3 optional/OptimalTest.java

```
1 package optional;
2
3 /**
4 * @version 1.01 2018-05-01
5 * @author Cay Horstmann
6 */
7
8 import java.io.*;
9 import java.nio.charset.*;
10 import java.nio.file.*;
11 import java.util.*;
12
13 public class OptimalTest
14 {
15     public static void main(String[] args) throws IOException
16     {
17         var contents = new String(Files.readAllBytes(
18             Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
19         List<String> wordList = List.of(contents.split("\\PL+"));
20
21         Optional<String> optionalValue = wordList.stream()
22             .filter(s -> s.contains("fred"))
23             .findFirst();
24         System.out.println(optionalValue.orElse("No word") + " contains fred");
25
26         Optional<String> optionalString = Optional.empty();
27         String result = optionalString.orElse("N/A");
```

```
28     System.out.println("result: " + result);
29     result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
30     System.out.println("result: " + result);
31     try
32     {
33         result = optionalString.orElseThrow(IllegalStateException::new);
34         System.out.println("result: " + result);
35     }
36     catch (Throwable t)
37     {
38         t.printStackTrace();
39     }
40
41     optionalValue = wordList.stream()
42         .filter(s -> s.contains("red"))
43         .findFirst();
44     optionalValue.ifPresent(s -> System.out.println(s + " contains red"));
45
46     var results = new HashSet<String>();
47     optionalValue.ifPresent(results::add);
48     Optional<Boolean> added = optionalValue.map(results::add);
49     System.out.println(added);
50
51     System.out.println(inverse(4.0).flatMap(OptionalTest::squareRoot));
52     System.out.println(inverse(-1.0).flatMap(OptionalTest::squareRoot));
53     System.out.println(inverse(0.0).flatMap(OptionalTest::squareRoot));
54     Optional<Double> result2 = Optional.of(-4.0)
55         .flatMap(OptionalTest::inverse).flatMap(OptionalTest::squareRoot);
56     System.out.println(result2);
57 }
58
59 public static Optional<Double> inverse(Double x)
60 {
61     return x == 0 ? Optional.empty() : Optional.of(1 / x);
62 }
63
64 public static Optional<Double> squareRoot(Double x)
65 {
66     return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
67 }
68 }
```

API java.util.Optional 8

- <U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper) 9

产生将 `mapper` 应用于当前 `Optional` 值所产生的结果，或者在当前 `Optional` 为空时，返回一个空 `Optional`。

1.8 收集结果

当处理完流之后，通常会想要查看其结果。此时可以调用 `iterator` 方法，它会产生用来

访问元素的旧式风格的迭代器。

或者，可以调用 `forEach` 方法，将某个函数应用于每个元素：

```
stream.forEach(System.out::println);
```

在并行流上，`forEach` 方法会以任意顺序遍历各个元素。如果想要按照流中的顺序来处理它们，可以调用 `forEachOrdered` 方法。当然，这个方法会丧失并行处理的部分甚至全部优势。

但是，更常见的情况是，我们想要将结果收集到数据结构中。此时，可以调用 `toArray`，获得由流的元素构成的数组。

因为无法在运行时创建泛型数组，所以表达式 `stream.toArray()` 会返回一个 `Object[]` 数组。如果想要让数组具有正确的类型，可以将其传递到数组构造器中：

```
String[] result = stream.toArray(String[]::new);
// stream.toArray() has type Object[]
```

针对将流中的元素收集到另一个目标中，有一个便捷方法 `collect` 可用，它会接受一个 `Collector` 接口的实例。收集器是一种收集众多元素并产生单一结果的对象，`Collectors` 类提供了大量用于生成常见收集器的工厂方法。要想将流的元素收集到一个列表中，应该使用 `Collectors.toList()` 方法产生的收集器：

```
List<String> result = stream.collect(Collectors.toList());
```

类似地，下面的代码展示了如何将流的元素收集到一个集中：

```
Set<String> result = stream.collect(Collectors.toSet());
```

如果想要控制获得的集的种类，那么可以使用下面的调用：

```
TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new));
```

假设想要通过连接操作来收集流中的所有字符串。我们可以调用

```
String result = stream.collect(Collectors.joining());
```

如果想要在元素之间增加分隔符，可以将分隔符传递给 `joining` 方法：

```
String result = stream.collect(Collectors.joining(", "));
```

如果流中包含除字符串以外的其他对象，那么我们需要先将其转换为字符串，就像下面这样：

```
String result = stream.map(Object::toString).collect(Collectors.joining(", "));
```

如果想要将流的结果约简为总和、数量、平均值、最大值或最小值，可以使用 `summarizing` (`Int`|`Long`|`Double`) 方法中的某一个。这些方法会接受一个将流对象映射为数值的函数，产生类型为 (`Int`|`Long`|`Double`)`SummaryStatistics` 的结果，同时计算总和、数量、平均值、最大值和最小值。

```
IntSummaryStatistics summary = stream.collect(
    Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

程序清单 1-4 中的示例程序展示了如何从流中收集元素。

程序清单 1-4 collecting/CollectingResults.java

```
1 package collecting;
2
3 /**
4  * @version 1.01 2018-05-01
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.nio.charset.*;
10 import java.nio.file.*;
11 import java.util.*;
12 import java.util.stream.*;
13
14 public class CollectingResults
15 {
16     public static Stream<String> noVowels() throws IOException
17     {
18         var contents = new String(Files.readAllBytes(
19             Paths.get("../gutenberg/alice30.txt")),
20             StandardCharsets.UTF_8);
21         List<String> wordList = List.of(contents.split("\\PL+"));
22         Stream<String> words = wordList.stream();
23         return words.map(s -> s.replaceAll("[aeiouAEIOU]", ""));
24     }
25
26     public static <T> void show(String label, Set<T> set)
27     {
28         System.out.print(label + ": " + set.getClass().getName());
29         System.out.println("["
30             + set.stream().limit(10).map(Object::toString).collect(Collectors.joining(", "))
31             + "]");
32     }
33
34     public static void main(String[] args) throws IOException
35     {
36         Iterator<Integer> iter = Stream.iterate(0, n -> n + 1).limit(10).iterator();
37         while (iter.hasNext())
38             System.out.println(iter.next());
39
40         Object[] numbers = Stream.iterate(0, n -> n + 1).limit(10).toArray();
41         System.out.println("Object array:" + numbers);
42         // Note it's an Object[] array
43
44         try
45         {
46             var number = (Integer) numbers[0]; // OK
47             System.out.println("number: " + number);
48             System.out.println("The following statement throws an exception:");
49             var numbers2 = (Integer[]) numbers; // Throws exception
50         }
```

```

51     catch (ClassCastException ex)
52     {
53         System.out.println(ex);
54     }
55
56     Integer[] numbers3 = Stream.iterate(0, n -> n + 1)
57         .limit(10)
58         .toArray(Integer[]::new);
59     System.out.println("Integer array: " + numbers3);
60     // Note it's an Integer[] array
61
62     Set<String> noVowelSet = noVowels().collect(Collectors.toSet());
63     show("noVowelSet", noVowelSet);
64
65     TreeSet<String> noVowelTreeSet = noVowels().collect(
66         Collectors.toCollection(TreeSet::new));
67     show("noVowelTreeSet", noVowelTreeSet);
68
69     String result = noVowels().limit(10).collect(Collectors.joining());
70     System.out.println("Joining: " + result);
71     result = noVowels().limit(10)
72         .collect(Collectors.joining(", "));
73     System.out.println("Joining with commas: " + result);
74
75     IntSummaryStatistics summary = noVowels().collect(
76         Collectors.summarizingInt(String::length));
77     double averageWordLength = summary.getAverage();
78     double maxWordLength = summary.getMax();
79     System.out.println("Average word length: " + averageWordLength);
80     System.out.println("Max word length: " + maxWordLength);
81     System.out.println("forEach:");
82     noVowels().limit(10).forEach(System.out::println);
83 }
84 }
```

API **java.util.stream.BaseStream** 8

- `Iterator<T> iterator()`

产生一个用于获取当前流中各个元素的迭代器。这是一种终结操作。

API **java.util.stream.Stream** 8

- `void forEach(Consumer<? super T> action)`

在流的每个元素上调用 `action`。这是一种终结操作。

- `Object[] toArray()`

- `<A> A[] toArray(IntFunction<A[]> generator)`

产生一个对象数组，或者在将引用 `A[]::new` 传递给构造器时，返回一个 `A` 类型的数组。这些操作都是终结操作。

- `<R,A> R collect(Collector<? super T,A,R> collector)`

使用给定的收集器来收集当前流中的元素。Collectors 类有用于多种收集器的工厂方法。

API `java.util.stream.Collectors` 8

- `static <T> Collector<T,?,List<T>> toList()`
- `static <T> Collector<T,?,List<T>> toUnmodifiableList() 10`
- `static <T> Collector<T,?,Set<T>> toSet()`
- `static <T> Collector<T,?,Set<T>> toUnmodifiableSet() 10`
产生一个将元素收集到列表或集合中的收集器。
- `static <T,C extends Collection<T>> Collector<T,?,C> toCollection(Supplier<C> collectionFactory)`
产生一个将元素收集到任意集合中的收集器。可以传递一个诸如 `TreeSet::new` 的构造器引用。
- `static Collector<CharSequence,?,String> joining()`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter)`
- `static Collector<CharSequence,?,String> joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`
产生一个连接字符串的收集器。分隔符会置于字符串之间，而第一个字符串之前可以有前缀，最后一个字符串之后可以有后缀。如果没有指定，那么它们都为空。
- `static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)`
- `static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)`
- `static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)`
产生能够生成 (`Int|Long|Double`)`SummaryStatistics` 对象的收集器，通过它们可以获得将 `mapper` 应用于每个元素后所产生的结果的数量、总和、平均值、最大值和最小值。

API `IntSummaryStatistics` 8

LongSummaryStatistics 8

DoubleSummaryStatistics 8

- `long getCount()`
产生汇总后的元素的个数。
- `(int|long|double) getSum()`
- `double getAverage()`
产生汇总后的元素的总和或平均值，或者在没有任何元素时返回 0。
- `(int|long|double) getMax()`
- `(int|long|double) getMin()`
产生汇总后的元素的最大值和最小值，或者在没有任何元素时，产生 (`Integer| Long| Double`).`(MAX|MIN)_VALUE`。

1.9 收集到映射表中

假设我们有一个 Stream<Person>, 并且想要将其元素收集到一个映射表中, 这样后续就可以通过它们的 ID 来查找人员了。Collectors.toMap 方法有两个函数引元, 它们用来产生映射表的键和值。例如,

```
Map<Integer, String> idToName = people.collect(
    Collectors.toMap(Person::getId, Person::getName));
```

通常情况下, 值应该是实际的元素, 因此第二个函数可以使用 Function.identity()。

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(Person::getId, Function.identity()));
```

如果有多个元素具有相同的键, 就会存在冲突, 收集器将会抛出一个 IllegalStateException 异常。可以通过提供第 3 个函数引元来覆盖这种行为, 该函数会针对给定的已有值和新值来解决冲突并确定键对应的值。这个函数应该返回已有值、新值或它们的组合。

在下面的代码中, 我们构建了一个映射表, 存储了所有可用 locale 中的语言, 其中每种语言在默认 locale 中的名字 (例如 “German”) 为键, 而其本地化的名字 (例如 “Deutsch”) 为值:

```
Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
Map<String, String> languageNames = locales.collect(
    Collectors.toMap(
        Locale::getDisplayLanguage,
        loc -> loc.getDisplayLanguage(loc),
        (existingValue, newValue) -> newValue));
```

我们不关心同一种语言是否可能会出现两次 (例如, 德国和瑞士都使用德语), 因此我们只记录第一项。

 **注释:** 在本章中, 我们使用 Locale 类作为感兴趣的数据集的数据源。请参阅第 7 章以了解有关 locale 的更多信息。

现在, 假设我们想要了解给定国家的所有语言, 这样我们就要一个 Map<String, Set<String>>。例如, "Switzerland" 的值是集 [French, German, Italian]。首先, 我们为每种语言都存储一个单例集。无论何时, 只要找到了给定国家的新语言, 我们就会对已有集和新集进行并操作。

```
Map<String, Set<String>> countryLanguageSets = locales.collect(
    Collectors.toMap(
        Locale::getDisplayCountry,
        l -> Collections.singleton(l.getDisplayLanguage()),
        (a, b) -> { // Union of a and b
            var union = new HashSet<String>(a);
            union.addAll(b);
            return union; }));
```

在下一节中, 你将会看到一种更简单的获取这种映射表的方式。

如果想要得到 TreeMap，那么可以将构造器作为第 4 个引元来提供。你必须提供一种合并函数。下面是本节一开始所列举的示例之一，现在它会产生一个 TreeMap：

```
Map<Integer, Person> idToPerson = people.collect(
    Collectors.toMap(
        Person::getId,
        Function.identity(),
        (existingValue, newValue) -> { throw new IllegalStateException(); },
        TreeMap::new));
```

注释：对于每一个 toMap 方法，都有一个等价的可以产生并发映射表的 toConcurrentMap 方法。单个并发映射表可以用于并行集合处理。当使用并行流时，共享的映射表比合并映射表更高效。注意，元素不再是按照流中的顺序收集的，但是通常这不会有什么问题。

程序清单 1-5 给出了将流的结果收集到映射表中的示例。

程序清单 1-5 collecting/CollectingIntoMaps.java

```
1 package collecting;
2
3 /**
4  * @version 1.00 2016-05-10
5  * @author Cay Horstmann
6 */
7
8 import java.io.*;
9 import java.util.*;
10 import java.util.function.*;
11 import java.util.stream.*;
12
13 public class CollectingIntoMaps
14 {
15
16     public static class Person
17     {
18         private int id;
19         private String name;
20
21         public Person(int id, String name)
22         {
23             this.id = id;
24             this.name = name;
25         }
26
27         public int getId()
28         {
29             return id;
30         }
31
32         public String getName()
33         {
34             return name;
35         }
36     }
37 }
```

```

35     }
36
37     public String toString()
38     {
39         return getClass().getName() + "[id=" + id + ",name=" + name + "]";
40     }
41 }
42
43     public static Stream<Person> people()
44     {
45         return Stream.of(new Person(1001, "Peter"), new Person(1002, "Paul"),
46             new Person(1003, "Mary"));
47     }
48
49     public static void main(String[] args) throws IOException
50     {
51         Map<Integer, String> idToName = people().collect(
52             Collectors.toMap(Person::getId, Person::getName));
53         System.out.println("idToName: " + idToName);
54
55         Map<Integer, Person> idToPerson = people().collect(
56             Collectors.toMap(Person::getId, Function.identity()));
57         System.out.println("idToPerson: " + idToPerson.getClass().getName()
58             + idToPerson);
59
60         idToPerson = people().collect(
61             Collectors.toMap(Person::getId, Function.identity(),
62                 (existingValue, newValue) -> { throw new IllegalStateException(); },
63                 TreeMap::new));
64         System.out.println("idToPerson: " + idToPerson.getClass().getName()
65             + idToPerson);
66
67         Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
68         Map<String, String> languageNames = locales.collect(
69             Collectors.toMap(
70                 Locale::getDisplayLanguage,
71                 l -> l.getDisplayLanguage(l),
72                 (existingValue, newValue) -> existingValue));
73         System.out.println("languageNames: " + languageNames);
74
75         locales = Stream.of(Locale.getAvailableLocales());
76         Map<String, Set<String>> countryLanguageSets = locales.collect(
77             Collectors.toMap(
78                 Locale::getDisplayCountry,
79                 l -> Set.of(l.getDisplayLanguage()),
80                 (a, b) ->
81                     { // union of a and b
82                         Set<String> union = new HashSet<>(a);
83                         union.addAll(b);
84                         return union;
85                     }));
86         System.out.println("countryLanguageSets: " + countryLanguageSets);
87     }
88 }

```

API `java.util.stream.Collectors` 8

- static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)
- static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)
- static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)
- static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper) 10
- static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction) 10
- static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)

产生一个收集器，它会产生一个映射表、不可修改的映射表或并发映射表。`keyMapper` 和 `valueMapper` 函数会应用于每个收集到的元素上，从而在所产生的映射表中生成一个键 / 值项。默认情况下，当两个元素产生相同的键时，会抛出一个 `IllegalStateException` 异常。你可以提供一个 `mergeFunction` 来合并具有相同键的值。默认情况下，其结果是一个 `HashMap` 或 `ConcurrentHashMap`。你可以提供一个 `mapSupplier`，它会产生所期望的映射表实例。

1.10 群组和分区

在上一节中，你看到了如何收集给定国家的所有语言，但是其处理显得有些冗长。你必须为每个映射表的值都生成单例集，然后指定如何将现有值与新值合并。将具有相同特性的值群聚成组是非常常见的，并且 `groupingBy` 方法直接就支持它。

我们来看看通过国家聚成组 `Locale` 的问题。首先，构建该映射表：

```
Map<String, List<Locale>> countryToLocales = locales.collect(
    Collectors.groupingBy(Locale::getCountry));
```

函数 `Locale::getCountry` 是群组的分类函数，你现在可以查找给定国家代码对应的所有地点了，例如：

```
List<Locale> swissLocales = countryToLocales.get("CH");
// Yields locales de_CH, fr_CH, it_CH and maybe more
```

注释：快速复习一下 `locale`：每个 `locale` 都有一个语言代码（例如英语的 `en`）和一个国家代码（例如美国的 `US`）。`locale en_US` 描述的是美国英语，而 `en_IE` 是爱尔兰英语。某些国家有多个 `locale`。例如，`ga_IE` 是爱尔兰的盖尔语，而前面的示例也展示了我的 JDK 知道瑞士至少有三个 `locale`。

当分类函数是断言函数（即返回 boolean 值的函数）时，流的元素可以分为两个列表：该函数返回 true 的元素和其他的元素。在这种情况下，使用 `partitioningBy` 比使用 `groupingBy` 更高效。例如，在下面的代码中，我们将所有 locale 分成了使用英语和使用所有其他语言的两类：

```
Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(
    Collectors.partitioningBy(l -> l.getLanguage().equals("en")));
List<Locale> englishLocales = englishAndOtherLocales.get(true);
```

注释：如果调用 `groupingByConcurrent` 方法，就会在使用并行流时获得一个被并行组装的并行映射表。这与 `toConcurrentMap` 方法完全类似。

API `java.util.stream.Collectors` 8

- static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)
- static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)
 产生一个收集器，它会产生一个映射表或并发映射表，其键是将 `classifier` 应用于所有收集到的元素上所产生的结果，而值是由具有相同键的元素构成的一个个列表。
- static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)
 产生一个收集器，它会产生一个映射表，其键是 `true/false`，而值是由满足 / 不满足断言的元素构成的列表。

1.11 下游收集器

`groupingBy` 方法会产生一个映射表，它的每个值都是一个列表。如果想要以某种方式来处理这些列表，就需要提供一个“下游收集器”。例如，如果想要获得集而不是列表，那么可以使用上一节中看到的 `Collectors.toSet` 收集器：

```
Map<String, Set<Locale>> countryToLocaleSet = locales.collect(
    groupingBy(Locale::getCountry, toSet()));
```

注释：在本节的这个示例以及后续示例中，我们认为静态导入 `java.util.stream.Collectors.*` 会使表达式更容易阅读。

Java 提供了多种可以将收集到的元素约简为数字的收集器：

- `counting` 会产生收集到的元素的个数。例如：

```
Map<String, Long> countryToLocaleCounts = locales.collect(
    groupingBy(Locale::getCountry, counting()));
```

可以对每个国家有多少个 locale 进行计数。

- `summing(Int|Long|Double)` 会接受一个函数作为引元，将该函数应用到下游元素中，并产生它们的和。例如：

```
Map<String, Integer> stateToCityPopulation = cities.collect(
    groupingBy(City::getState, summarizingInt(City::getPopulation)));
```

可以计算城市流中每个州的人口总和。

- `maxBy` 和 `minBy` 会接受一个比较器，并分别产生下游元素中的最大值和最小值。例如：

```
Map<String, Optional<City>> stateToLargestCity = cities.collect(
    groupingBy(City::getState,
        maxBy(Comparator.comparing(City::getPopulation))));
```

可以产生每个州中最大的城市。

`collectingAndThen` 收集器在收集器后面添加了一个最终处理步骤。例如，如果我们想要知道有多少不同的结果，那么就可以将它们收集到一个集中，然后计算其尺寸：

```
Map<Character, Integer> stringCountsByStartingLetter = strings.collect(
    groupingBy(s -> s.charAt(0),
        collectingAndThen(toSet(), Set::size)));
```

`mapping` 收集器的做法正好相反，它会将一个函数应用于收集到的每个元素，并将结果传递给下游收集器。

```
Map<Character, Set<Integer>> stringLengthsByStartingLetter = strings.collect(
    groupingBy(s -> s.charAt(0),
        mapping(String::length, toSet())));
```

这里，我们按照首字符对字符串进行了分组。在每个组内部，我们会计算字符串的长度，然后将这些长度收集到一个集中。

`mapping` 方法还针对上一节中的问题，即把某国所有的语言收集到一个集中，产生了一种更佳的解决方案。

```
Map<String, Set<String>> countryToLanguages = locales.collect(
    groupingBy(Locale::getDisplayCountry,
        mapping(Locale::getDisplayLanguage,
            toSet())));
```

还有一个 `flatMap` 方法，可以与返回流的函数一起使用。

如果群组和映射函数的返回值为 `int`、`long` 或 `double`，那么可以将元素收集到汇总统计对象中，就像 1.8 节中所讨论的一样。例如，

```
Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities.collect(
    groupingBy(City::getState,
        summarizingInt(City::getPopulation)));
```

然后，可以从每个组的汇总统计对象中获取这些函数值的总和、数量、平均值、最小值和最大值。

`filtering` 收集器会将一个过滤器应用到每个组上，例如：

```
Map<String, Set<City>> largeCitiesByState
= cities.collect(
    groupingBy(City::getState,
        filtering(c -> c.getPopulation() > 500000,
            toSet()))); // States without large cities have empty sets
```

注释：还有3个版本的reducing方法，它们都应用了通用的约简操作，正如1.12节中所描述的一样。

将收集器组合起来是一种很强大的方式，但是它也可能会导致产生非常复杂的表达式。最佳用法是与groupingBy和partitioningBy一起处理“下游的”映射表中的值。否则，应该直接在流上应用诸如map、reduce、count、max或min这样的方法。

程序清单1-6中的示例程序演示了下游收集器。

程序清单1-6 collecting/DownstreamCollectors.java

```

1 package collecting;
2
3 /**
4  * @version 1.00 2016-05-10
5  * @author Cay Horstmann
6  */
7
8 import static java.util.stream.Collectors.*;
9
10 import java.io.*;
11 import java.nio.file.*;
12 import java.util.*;
13 import java.util.stream.*;
14
15 public class DownstreamCollectors
16 {
17
18     public static class City
19     {
20         private String name;
21         private String state;
22         private int population;
23
24         public City(String name, String state, int population)
25         {
26             this.name = name;
27             this.state = state;
28             this.population = population;
29         }
30
31         public String getName()
32         {
33             return name;
34         }
35
36         public String getState()
37         {
38             return state;
39         }
40
41         public int getPopulation()

```

```
42     {
43         return population;
44     }
45 }
46
47 public static Stream<City> readCities(String filename) throws IOException
48 {
49     return Files.lines(Paths.get(filename))
50         .map(l -> l.split(", "))
51         .map(a -> new City(a[0], a[1], Integer.parseInt(a[2])));
52 }
53
54 public static void main(String[] args) throws IOException
55 {
56     Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());
57     locales = Stream.of(Locale.getAvailableLocales());
58     Map<String, Set<Locale>> countryToLocaleSet = locales.collect(groupingBy(
59         Locale::getCountry, toSet()));
60     System.out.println("countryToLocaleSet: " + countryToLocaleSet);
61
62     locales = Stream.of(Locale.getAvailableLocales());
63     Map<String, Long> countryToLocaleCounts = locales.collect(groupingBy(
64         Locale::getCountry, counting()));
65     System.out.println("countryToLocaleCounts: " + countryToLocaleCounts);
66
67     Stream<City> cities = readCities("cities.txt");
68     Map<String, Integer> stateToCityPopulation = cities.collect(groupingBy(
69         City::getState, summingInt(City::getPopulation)));
70     System.out.println("stateToCityPopulation: " + stateToCityPopulation);
71
72     cities = readCities("cities.txt");
73     Map<String, Optional<String>> stateToLongestCityName = cities
74         .collect(groupingBy(City::getState,
75             mapping(City::getName, maxBy(Comparator.comparing(String::length)))));
76     System.out.println("stateToLongestCityName: " + stateToLongestCityName);
77
78     locales = Stream.of(Locale.getAvailableLocales());
79     Map<String, Set<String>> countryToLanguages = locales.collect(groupingBy(
80         Locale::getDisplayCountry, mapping(Locale::getDisplayLanguage, toSet())));
81     System.out.println("countryToLanguages: " + countryToLanguages);
82
83     cities = readCities("cities.txt");
84     Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities
85         .collect(groupingBy(City::getState, summarizingInt(City::getPopulation)));
86     System.out.println(stateToCityPopulationSummary.get("NY"));
87
88     cities = readCities("cities.txt");
89     Map<String, String> stateToCityNames = cities.collect(groupingBy(
90         City::getState,
91         reducing("", City::getName, (s, t) -> s.length() == 0 ? t : s + ", " + t)));
92
93     cities = readCities("cities.txt");
94     stateToCityNames = cities.collect(groupingBy(City::getState,
95         mapping(City::getName, joining(", "))));
96 }
```

```

96     System.out.println("stateToCityNames: " + stateToCityNames);
97 }
98 }
```

API `java.util.stream.Collectors` 8

- `public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`

产生一个收集器，该收集器会产生一个映射表，其中的键是将 `classifier` 应用到所有收集到的元素上之后产生的结果，而值是使用下游收集器收集具有相同的键的元素所产生的结果。

- `static <T> Collector<T,?,Long> counting()`

产生一个可以对收集到的元素进行计数的收集器。

- `static <T> Collector<T,?,Integer> summingInt(ToIntFunction<? super T> mapper)`

- `static <T> Collector<T,?,Long> summingLong(ToLongFunction<? super T> mapper)`

- `static <T> Collector<T,?,Double> summingDouble(ToDoubleFunction<? super T> mapper)`

产生一个收集器，对将 `mapper` 应用到收集到的元素上之后产生的结果计算总和。

- `static <T> Collector<T,?,Optional<T>> maxBy(Comparator<? super T> comparator)`

- `static <T> Collector<T,?,Optional<T>> minBy(Comparator<? super T> comparator)`

产生一个收集器，使用 `comparator` 指定的排序方法，计算收集到的元素中的最大值和最小值。

- `static <T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)`

产生一个收集器，它会将元素发送到下游收集器中，然后将 `finisher` 函数应用到其结果上。

- `static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)`

产生一个收集器，它会在每个元素上调用 `mapper`，并将结果发送到下游收集器中。

- `static <T,U,A,R> Collector<T,?,R> flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)`

产生一个收集器，它会在每个元素上调用 `mapper`，并将结果中的元素发送到下游收集器中。

- `static <T,A,R> Collector<T,?,R> filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)`

产生一个收集器，它会将满足谓词逻辑的元素发送到下游收集器中。

1.12 约简操作

`reduce` 方法是一种用于从流中计算某个值的通用机制，其最简单的形式将接受一个二元

函数，并从前两个元素开始持续应用它。如果该函数是求和函数，那么就很容易解释这种机制：

```
List<Integer> values = . . .;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

在上面的情况下，`reduce` 方法会计算 $v_0+v_1+v_2+\cdots$ ，其中 v_i 是流中的元素。如果流为空，那么该方法会返回一个 `Optional`，因为没有任何有效的结果。

注释：在上面的情况下，可以写成 `reduce(Integer::sum)` 而不是 `reduce((x, y) -> x+y)`。

更一般地，我们可以使用任何约简操作将部分结果 x 与下一个值 y 组合起来以产生新的部分结果。

下面是另一种看待约简的方式。给定约简操作 op ，该约简会产生 $v_0 op v_1 op v_2 op \dots$ ，其中我们将函数调用 $op(v_i, v_{i+1})$ 写作 $v_i op v_{i+1}$ 。有很多种在实践中很有用的可结合操作，例如求和、乘积、字符串连接、求最大值和最小值、求集的并与交等。

如果要用并行流来约简，那么这项约简操作必须是可结合的，即组合元素时使用的顺序不会产生任何影响。在数学标记法中， $(x op y) op z$ 必须等于 $x op (y op z)$ 。减法是一个不可结合操作的例子，例如， $(6 - 3) - 2 \neq 6 - (3 - 2)$ 。

通常，会有一个幺元值 e 使得 $e op x = x$ ，可以使用这个元素作为计算的起点。例如，0 是加法的幺元值。由此，我们可以使用第 2 种形式的 `reduce`：

```
List<Integer> values = . . .;
Integer sum = values.stream().reduce(0, (x, y) -> x + y);
// Computes 0 + v_0 + v_1 + v_2 + . . .
```

如果流为空，则会返回幺元值，你就再也不需要处理 `Optional` 类了。

现在，假设你有一个对象流，并且想要对某些属性求和，例如字符串流中所有字符串的长度，那么你就不能使用简单形式的 `reduce`，而是需要 $(T, T) \rightarrow T$ 这样的函数，即引元和结果的类型相同的函数。但是在这种情况下，你有两种类型：流的元素具有 `String` 类型，而累积结果是整数。有一种形式的 `reduce` 可以处理这种情况。

首先，你需要提供一个“累积器”函数 $(total, word) \rightarrow total + word.length()$ 。这个函数会被反复调用，产生累积的总和。但是，当计算被并行化时，会有多个这种类型的计算，你需要将它们的结果合并。因此，你需要提供第二个函数来执行此处理。完整的调用如下：

```
int result = words.reduce(0,
    (total, word) -> total + word.length(),
    (total1, total2) -> total1 + total2);
```

注释：在实践中，你可能并不会频繁地用到 `reduce` 方法。通常，映射为数字流并使用其方法来计算总和、最大值和最小值会更容易。（我们将在 1.13 节中讨论数字流。）在这个特定示例中，你可以调用 `words.mapToInt(String::length).sum()`，因为它不涉及装箱操作，所以更简单也更高效。

注释：有时 `reduce` 会显得不够通用。例如，假设我们想要收集 `BitSet` 中的结果。如果收集操作是并行的，那么就不能直接将元素放到单个 `BitSet` 中，因为 `BitSet` 对象不是线程安全的。因此，我们不能使用 `reduce`，因为每个部分都需要以其自己的空集开始，并且 `reduce` 只能让我们提供一个幺元值。此时，应该使用 `collect`，它会接受单个引元：

1. 一个提供者，它会创建目标对象的新实例，例如散列集的构造器。
2. 一个累积器，它会将一个元素添加到该目标上，例如 `add` 方法。
3. 一个组合器，它会将两个对象合并成一个，例如 `addAll`。

下面的代码展示了 `collect` 方法是如何操作位集的：

```
BitSet result = stream.collect(BitSet::new, BitSet::set, BitSet::or);
```

java.util.Stream 8

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `T reduce(T identity, BinaryOperator<T> accumulator)`
- `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`
用给定的 `accumulator` 函数产生流中元素的累积总和。如果提供了幺元，那么第一个被累积的元素就是该幺元。如果提供了组合器，那么它可以用来将分别累积的各个部分整合成总和。
- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`

将元素收集到类型 `R` 的结果中。在每个部分上，都会调用 `supplier` 来提供初始结果，调用 `accumulator` 来交替地将元素添加到结果中，并调用 `combiner` 来整合两个结果。

1.13 基本类型流

到目前为止，我们都是将整数收集到 `Stream<Integer>` 中，尽管很明显，但是将每个整数都包装到包装器对象中却是很低效的。对其他基本类型来说，情况也是一样，这些基本类型是 `double`、`float`、`long`、`short`、`char`、`byte` 和 `boolean`。流库中具有专门的类型 `IntStream`、`LongStream` 和 `DoubleStream`，用来直接存储基本类型值，而无须使用包装器。如果想要存储 `short`、`char`、`byte` 和 `boolean`，可以使用 `IntStream`；而对于 `float`，可以使用 `DoubleStream`。

为了创建 `IntStream`，需要调用 `IntStream.of` 和 `Arrays.stream` 方法：

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5);
stream = Arrays.stream(values, from, to); // values is an int[] array
```

与对象流一样，我们还可以使用静态的 `generate` 和 `iterate` 方法。此外，`IntStream` 和 `LongStream` 有静态方法 `range` 和 `rangeClosed`，可以生成步长为 1 的整数范围：

```
IntStream zeroToNinetyNine = IntStream.range(0, 100); // Upper bound is excluded
IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper bound is included
```

CharSequence 接口拥有 codePoints 和 chars 方法，可以生成由字符的 Unicode 码或由 UTF-16 编码机制的码元构成的 IntStream。

```
String sentence = "\uD835\uDD46 is the set of octonions.";
// \uD835\uDD46 is the UTF-16 encoding of the letter Ⓛ, unicode U+1D546

IntStream codes = sentence.codePoints();
// The stream with hex values 1D546 20 69 73 20 . . .
```

当你有一个对象流时，可以用 mapToInt、mapToLong 或 mapToDouble 将其转换为基本类型流。例如，如果你有一个字符串流，并想将其长度处理为整数，那么就可以在 IntStream 中实现此目的：

```
Stream<String> words = . . .;
IntStream lengths = words.mapToInt(String::length);
```

为了将基本类型流转换为对象流，需要使用 boxed 方法：

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

通常，基本类型流上的方法与对象流上的方法类似。下面是主要的差异：

- toArray 方法会返回基本类型数组。
- 产生可选结果的方法会返回一个 OptionalInt、OptionalLong 或 OptionalDouble。这些类与 Optional 类类似，但是具有 getAsInt、getAsLong 和 getAsDouble 方法，而不是 get 方法。
- 具有分别返回总和、平均值、最大值和最小值的 sum、average、max 和 min 方法。对象流没有定义这些方法。
- summaryStatistics 方法会产生一个类型为 IntSummaryStatistics、LongSummaryStatistics 或 DoubleSummaryStatistics 的对象，它们可以同时报告流的总和、数量、平均值、最大值和最小值。

注释：Random 类具有 ints、longs 和 doubles 方法，它们会返回由随机数构成的基本类型流。如果需要的是并行流中的随机数，那么需要使用 SplittableRandom 类。

程序清单 1-7 给出了基本类型流的 API 的示例。

程序清单 1-7 streams/PrimitiveTypeStreams.java

```
1 package streams;
2
3 /**
4 * @version 1.01 2018-05-01
5 * @author Cay Horstmann
6 */
7
8 import java.io.IOException;
9 import java.nio.charset.StandardCharsets;
10 import java.nio.file.Files;
11 import java.nio.file.Path;
12 import java.nio.file.Paths;
13 import java.util.stream.Collectors;
```

```

14 import java.util.stream.IntStream;
15 import java.util.stream.Stream;
16
17 public class PrimitiveTypeStreams
18 {
19     public static void show(String title, IntStream stream)
20     {
21         final int SIZE = 10;
22         int[] firstElements = stream.limit(SIZE + 1).toArray();
23         System.out.print(title + ": ");
24         for (int i = 0; i < firstElements.length; i++)
25         {
26             if (i > 0) System.out.print(", ");
27             if (i < SIZE) System.out.print(firstElements[i]);
28             else System.out.print("...");  

29         }
30         System.out.println();
31     }
32
33     public static void main(String[] args) throws IOException
34     {
35         IntStream is1 = IntStream.generate(() -> (int) (Math.random() * 100));
36         show("is1", is1);
37         IntStream is2 = IntStream.range(5, 10);
38         show("is2", is2);
39         IntStream is3 = IntStream.rangeClosed(5, 10);
40         show("is3", is3);
41
42         Path path = Paths.get("../gutenberg/alice30.txt");
43         var contents = new String(Files.readAllBytes(path), StandardCharsets.UTF_8);
44
45         Stream<String> words = Stream.of(contents.split("\\PL+"));
46         IntStream is4 = words.mapToInt(String::length);
47         show("is4", is4);
48         var sentence = "\uD835\uDD46 is the set of octonions.";
49         System.out.println(sentence);
50         IntStream codes = sentence.codePoints();
51         System.out.println(codes.mapToObj(c -> String.format("%X ", c)).collect(
52             Collectors.joining()));
53
54         Stream<Integer> integers = IntStream.range(0, 100).boxed();
55         IntStream is5 = integers.mapToInt(Integer::intValue);
56         show("is5", is5);
57     }
58 }

```

API `java.util.stream.IntStream` 8

- `static IntStream range(int startInclusive, int endExclusive)`
- `static IntStream rangeClosed(int startInclusive, int endInclusive)`
产生一个由给定范围内的整数构成的 `IntStream`。
- `static IntStream of(int... values)`

产生一个由给定元素构成的 IntStream。

- int[] toArray()

产生一个由当前流中的元素构成的数组。

- int sum()

- OptionalDouble average()

- OptionalInt max()

- OptionalInt min()

- IntSummaryStatistics summaryStatistics()

产生当前流中元素的总和、平均值、最大值和最小值，或者产生一个从中获取所有这四个值的对象。

- Stream<Integer> boxed()

产生用于当前流中的元素的包装器对象流。

API `java.util.stream.LongStream 8`

- static LongStream range(long startInclusive, long endExclusive)

- static LongStream rangeClosed(long startInclusive, long endInclusive)

用给定范围内的整数产生一个 LongStream。

- static LongStream of(long... values)

用给定元素产生一个 LongStream。

- long[] toArray()

用当前流中的元素产生一个数组。

- long sum()

- OptionalDouble average()

- OptionalLong max()

- OptionalLong min()

- LongSummaryStatistics summaryStatistics()

产生当前流中元素的总和、平均值、最大值和最小值，或者产生一个从中获取所有这四个值的对象。

- Stream<Long> boxed()

产生用于当前流中的元素的包装器对象流。

API `java.util.stream.DoubleStream 8`

- static DoubleStream of(double... values)

用给定元素产生一个 DoubleStream。

- double[] toArray()

用当前流中的元素产生一个数组。

- double sum()

- `OptionalDouble average()`
- `OptionalDouble max()`
- `OptionalDouble min()`
- `DoubleSummaryStatistics summaryStatistics()`

产生当前流中元素的总和、平均值、最大值和最小值，或者产生一个从中获取所有这四个值的对象。

- `Stream<Double> boxed()`
- 产生用于当前流中的元素的包装器对象流。

API `java.lang.CharSequence 1.0`

- `IntStream codePoints() 8`
- 产生由当前字符串的所有 Unicode 码点构成的流。

API `java.util.Random 1.0`

- `IntStream ints()`
 - `IntStream ints(int randomNumberOrigin, int randomNumberBound) 8`
 - `IntStream ints(long streamSize) 8`
 - `IntStream ints(long streamSize, int randomNumberOrigin, int randomNumberBound) 8`
 - `LongStream longs() 8`
 - `LongStream longs(long randomNumberOrigin, long randomNumberBound) 8`
 - `LongStream longs(long streamSize) 8`
 - `LongStream longs(long streamSize, long randomNumberOrigin, long randomNumberBound) 8`
 - `DoubleStream doubles() 8`
 - `DoubleStream doubles(double randomNumberOrigin, double randomNumberBound) 8`
 - `DoubleStream doubles(long streamSize) 8`
 - `DoubleStream doubles(long streamSize, double randomNumberOrigin, double randomNumberBound) 8`
- 产生随机数流。如果提供了 `streamSize`，这个流就是具有给定数量元素的有限流。当提供了边界时，其元素将位于 `randomNumberOrigin`（包含）和 `randomNumberBound`（不包含）的区间内。

API `java.util.Optional(Int|Long|Double) 8`

- `static Optional(Int|Long|Double) of((int|long|double) value)`
用所提供的基本类型值产生一个可选对象。
- `(int|long|double) getAs(Int|Long|Double)()`
产生当前可选对象的值，或者在其为空时抛出一个 `NoSuchElementException` 异常。
- `(int|long|double) orElse((int|long|double) other)`
- `(int|long|double) orElseGet((Int|Long|Double)Supplier other)`

产生当前可选对象的值，或者在这个对象为空时产生可替代的值。

- void ifPresent((Int|Long|Double)Consumer consumer)

如果当前可选对象不为空，则将其值传递给 consumer。

API `java.util.(Int|Long|Double)SummaryStatistics` 8

- long getCount()
- (int|long|double) getSum()
- double getAverage()
- (int|long|double) getMax()
- (int|long|double) getMin()

产生收集到的元素的数量、总和、平均值、最大值和最小值。

1.14 并行流

流使并行处理块操作变得很容易。这个过程几乎是自动的，但是需要遵守一些规则。首先，必须有一个并行流。可以用 `Collection.parallelStream()` 方法从任何集合中获取一个并行流：

```
Stream<String> parallelWords = words.parallelStream();
```

而且，`parallel` 方法可以将任意的顺序流转换为并行流。

```
Stream<String> parallelWords = Stream.of(wordArray).parallel();
```

只要在终结方法执行时流处于并行模式，所有的中间流操作就都将被并行化。

当流操作并行运行时，其目标是让其返回结果与顺序执行时返回的结果相同。重要的是，这些操作是无状态的，并且可以以任意顺序执行。

下面的示例是一项你无法完成的任务。假设你想要对字符串流中的所有短单词计数：

```
var shortWords = new int[12];
words.parallelStream().forEach(
    s -> { if (s.length() < 12) shortWords[s.length()]++; });
// ERROR--race condition!
System.out.println(Arrays.toString(shortWords));
```

这是一种非常糟糕的代码。传递给 `forEach` 的函数会在多个并发线程中运行，每个都会更新共享的数组。正如我们在卷 I 第 12 章中所解释的，这是一种经典的竞争情况。如果多次运行这个程序，你很可能就会发现每次运行都产生不同的计数值，而且每个都是错的。

你的职责是确保传递给并行流操作的任何函数都可以安全地并行执行，达到这个目的的最佳方式是远离易变状态。在本例中，如果用长度将字符串分组，然后分别对它们进行计数，那么就可以安全地并行化这项计算。

```
Map<Integer, Long> shortWordCounts
= words.parallelStream()
    .filter(s -> s.length() < 12)
    .collect(groupingBy(
        String::length,
        counting()));
```

默认情况下，从有序集合（数组和列表）、范围、生成器和迭代器产生的流，或者通过调用 Stream.sorted 产生的流，都是有序的。它们的结果是按照原来元素的顺序累积的，因此是完全可预知的。如果运行相同的操作两次，将会得到完全相同的结果。

排序并不排斥高效的并行处理。例如，当计算 stream.map(fun) 时，流可以被划分为 n 部分，它们会被并行地处理。然后，结果将会按照顺序重新组装起来。

当放弃排序需求时，有些操作可以被更有效地并行化。通过在流上调用 Stream.unordered 方法，就可以明确表示我们对排序不感兴趣。Stream.distinct 就是从这种方式中获益的一种操作。在有序的流中，distinct 会保留所有相同元素中的第一个，这对并行化是一种阻碍，因为处理每个部分的线程在其之前的所有部分都被处理完之前，并不知道应该丢弃哪些元素。如果可以接受保留唯一元素中任意一个的做法，那么所有部分就可以并行地处理（使用共享的集合来跟踪重复元素）。

还可以通过放弃排序要求来提高 limit 方法的速度。如果只想从流中取出任意 n 个元素，而并不在意到底要获取哪些，那么可以调用：

```
Stream<String> sample = words.parallelStream().unordered().limit(n);
```

正如 1.9 节所讨论的，合并映射表的代价很高昂。正是这个原因，Collectors.groupingByConcurrent 方法使用了共享的并发映射表。为了从并行化中获益，映射表中值的顺序不会与流中的顺序相同。

```
Map<Integer, List<String>> result = words.parallelStream().collect(
    Collectors.groupingByConcurrent(String::length));
// Values aren't collected in stream order
```

当然，如果使用独立于排序的下游收集器，那么就不必在意了，例如：

```
Map<Integer, Long> wordCounts
= words.parallelStream()
.collect(
    groupingByConcurrent(
        String::length,
        counting()));
```

不要指望通过将所有的流都转换为并行流就能够加速操作，要牢记下面几条：

- 并行化会导致大量的开销，只有面对非常大的数据集才划算。
- 只有在底层的数据源可以被有效地分割为多个部分时，将流并行化才有意义。
- 并行流使用的线程池可能会因诸如文件 I/O 或网络访问这样的操作被阻塞而饿死。

只有面对海量的内存数据和运算密集处理，并行流才会工作最佳。

 **提示：**在 Java 9 之前，对 Files.lines 方法返回的流进行并行化是没有意义的。因为数据是不可分割的，所以我们只能在读取文件的后半部分之前读取前半部分。现在，该方法使用的是内存映射文件，因此可以有效地进行分割。如果想要处理一个大型文件的各个行，并行化这个流可能会提高性能。

注释：默认情况下，并行流使用的是 ForkJoinPool.commonPool 返回的全局 fork-join 池。只有在操作不会阻塞并且我们不会将这个池与其他任务共享的情况下，这种方式才不会有有什么问题。有一种解决方法是使用另一个不同的池，即把操作放置到定制的池的 submit 方法中：

```
ForkJoinPool customPool = . . .;
result = customPool.submit(() ->
    stream.parallel().map(. . .).collect(. . .)).get();
```

或者，使用异步方式：

```
CompletableFuture.supplyAsync(() ->
    stream.parallel().map(. . .).collect(. . .),
    customPool).thenAccept(result -> . . .);
```

注释：如果想要并行化基于随机数的流计算，那么请不要以从 Random.ints、Random.longs 或 Random.doubles 方法中获得的流为起点，因为这些流不可分割。应该使用 SplittableRandom 类的 ints、longs 或 doubles。

程序清单 1-8 中的示例程序展示了如何操作并行流。

程序清单 1-8 parallel/ParallelStreams.java

```
1 package parallel;
2
3 /**
4  * @version 1.01 2018-05-01
5  * @author Cay Horstmann
6 */
7
8 import static java.util.stream.Collectors.*;
9
10 import java.io.*;
11 import java.nio.charset.*;
12 import java.nio.file.*;
13 import java.util.*;
14 import java.util.stream.*;
15
16 public class ParallelStreams
17 {
18     public static void main(String[] args) throws IOException
19     {
20         var contents = new String(Files.readAllBytes(
21             Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
22         List<String> wordList = List.of(contents.split("\\PL+"));
23
24         // Very bad code ahead
25         var shortWords = new int[10];
26         wordList.parallelStream().forEach(s ->
27         {
28             if (s.length() < 10) shortWords[s.length()]++;
```

```

29     });
30     System.out.println(Arrays.toString(shortWords));
31
32     // Try again--the result will likely be different (and also wrong)
33     Arrays.fill(shortWords, 0);
34     wordList.parallelStream().forEach(s ->
35     {
36         if (s.length() < 10) shortWords[s.length()]++;
37     });
38     System.out.println(Arrays.toString(shortWords));
39
40     // Remedy: Group and count
41     Map<Integer, Long> shortWordCounts = wordList.parallelStream()
42         .filter(s -> s.length() < 10)
43         .collect(groupingBy(String::length, counting()));
44
45     System.out.println(shortWordCounts);
46
47     // Downstream order not deterministic
48     Map<Integer, List<String>> result = wordList.parallelStream().collect(
49         Collectors.groupingByConcurrent(String::length));
50
51     System.out.println(result.get(14));
52
53     result = wordList.parallelStream().collect(
54         Collectors.groupingByConcurrent(String::length));
55
56     System.out.println(result.get(14));
57
58     Map<Integer, Long> wordCounts = wordList.parallelStream().collect(
59         groupingByConcurrent(String::length, counting()));
60
61     System.out.println(wordCounts);
62 }
63 }
```

API `java.util.stream.BaseStream<T,S extends BaseStream<T,S>>` 8

- `S parallel()`
产生一个与当前流中元素相同的并行流。
- `S unordered()`
产生一个与当前流中元素相同的无序流。

API `java.util.Collection<E>` 1.2

- `Stream<E> parallelStream()` 8
用当前集合中的元素产生一个并行流。

在本章中，你学习到了如何运用 Java 8 的流库。下一章将讨论另一个重要的主题：输入与输出。

第 2 章 输入与输出

- ▲ 输入 / 输出流
- ▲ 读写二进制数据
- ▲ 对象输入 / 输出流与序列化
- ▲ 操作文件

- ▲ 内存映射文件
- ▲ 文件锁机制
- ▲ 正则表达式

本章将介绍 Java 中用于输入和输出的各种应用编程接口 (Application Programming Interface, API)。你将要学习如何访问文件与目录，以及如何以二进制格式和文本格式来读写数据。本章还要向你展示对象序列化机制，它可以使存储对象像存储文本和数值数据一样容易。然后，我们将介绍如何使用文件和目录。最后，本章将讨论正则表达式，尽管这部分内容实际上与输入和输出并不相关，但是我们确实也找不到更合适的地方来处理这个话题。很明显，Java 设计团队在这个问题的处理上和我们一样，因为正则表达式 API 的规格说明隶属于“新 I/O”特性的规格说明。

2.1 输入 / 输出流

在 Java API 中，可以从其中读入一个字节序列的对象称作输入流，而可以向其中写入一个字节序列的对象称作输出流。这些字节序列的来源地和目的地可以是文件，而且通常都是文件，但是也可以是网络连接，甚至是内存块。抽象类 `InputStream` 和 `OutputStream` 构成了输入 / 输出 (I/O) 类层次结构的基础。

 **注释：**这些输入 / 输出流与在前一章中看到的流没有任何关系。为了清楚起见，只要是讨论用于输入和输出的流，我们都将使用术语输入流、输出流或输入 / 输出流。

因为面向字节的流不便于处理以 Unicode 形式存储的信息（回忆一下，Unicode 中每个字符都使用了多个字节来表示），所以从抽象类 `Reader` 和 `Writer` 中继承出来了一个专门用于处理 Unicode 字符的单独的类层次结构。这些类拥有的读入和写出操作都是基于两字节的 `Char` 值的（即 Unicode 码元），而不是基于 `byte` 值的。

2.1.1 读写字节

`InputStream` 类有一个抽象方法：

```
abstract int read()
```

这个方法将读入一个字节，并返回读入的字节，或者在遇到输入源结尾时返回 -1。在设计具体的输入流类时，必须覆盖这个方法以提供适用的功能，例如，在 `FileInputStream` 类中，

这个方法将从某个文件中读入一个字节，而 `System.in`（它是 `InputStream` 的一个子类的预定义对象）却是从“标准输入”中读入信息，即从控制台或重定向的文件中读入信息。

`InputStream` 类还有若干个非抽象的方法，它们可以读入一个字节数组，或者跳过大量的字节。从 Java 9 开始，有了一个非常有用可以读取流中所有字节的方法：

```
byte[] bytes = in.readAllBytes();
```

还有多个用来读取给定数量字节的方法，可以参见 API 说明。

这些方法都要调用抽象的 `read` 方法，因此，各个子类都只需覆盖这一个方法。

与此类似，`OutputStream` 类定义了下面的抽象方法：

```
abstract void write(int b)
```

它可以向某个输出位置写出一个字节。

如果我们有一个字节数组，那么就可以一次性地写出它们：

```
byte[] values = . . .;
out.write(values);
```

`transferTo` 方法可以将所有字节从一个输入流传递到一个输出流：

```
in.transferTo(out);
```

`read` 和 `write` 方法在执行时都将阻塞，直至字节确实被读入或写出。这就意味着如果流不能被立即访问（通常是因为网络连接忙），那么当前的线程将被阻塞。这使得在这两个方法等待指定的流变为可用的这段时间里，其他的线程就有机会去执行有用的工作。

`available` 方法使我们可以去检查当前可读入的字节数量，这意味着像下面这样的代码片段不可能被阻塞：

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
    var data = new byte[bytesAvailable];
    in.read(data);
}
```

当你完成对输入 / 输出流的读写时，应该通过调用 `close` 方法来关闭它，这个调用会释放掉十分有限的操作系统资源。如果一个应用程序打开了过多的输入 / 输出流而没有关闭，那么系统资源将被耗尽。关闭一个输出流的同时还会冲刷用于该输出流的缓冲区：所有被临时置于缓冲区中，以便用更大的包的形式传递的字节在关闭输出流时都将被送出。特别是，如果不关闭文件，那么写出字节的最后一个包可能永远也得不到传递。当然，我们还可以用 `flush` 方法来人为地冲刷这些输出。

即使某个输入 / 输出流类提供了使用原生的 `read` 和 `write` 功能的某些具体方法，应用系统的程序员还是很少使用它们，因为大家感兴趣的数据可能包含数字、字符串和对象，而不是原生字节。

我们可以使用众多的构建于基本的 `InputStream` 和 `OutputStream` 类之上的某个输入 / 输出类，而不只是直接使用字节。

API **java.io.InputStream 1.0**

- **abstract int read()**
从数据中读入一个字节，并返回该字节。这个 `read` 方法在碰到输入流的结尾时返回 -1。
- **int read(byte[] b)**
读入一个字节数组，并返回实际读入的字节数，或者在碰到输入流的结尾时返回 -1。
这个 `read` 方法最多读入 `b.length` 个字节。
- **int read(byte[] b, int off, int len)**
- **int readNBytes(byte[] b, int off, int len) 9**
如果未阻塞 (`read`)，则读入由 `len` 指定数量的字节，或者阻塞至所有的值都被读入 (`readNBytes`)。读入的值将置于 `b` 中从 `off` 开始的位置。返回实际读入的字节数，或者在碰到输入流的结尾时返回 -1。
- **byte[] readAllBytes() 9**
产生一个数组，包含可以从当前流中读入的所有字节。
- **long transferTo(OutputStream out) 9**
将当前输入流中的所有字节传送到给定的输出流，返回传递的字节数。这两个流都不应该处于关闭状态。
- **long skip(long n)**
在输入流中跳过 `n` 个字节，返回实际跳过的字节数（如果碰到输入流的结尾，则可能小于 `n`）。
- **int available()**
返回在不阻塞的情况下可获取的字节数（回忆一下，阻塞意味着当前线程将失去它对资源的占用）。
- **void close()**
关闭这个输入流。
- **void mark(int readlimit)**
在输入流的当前位置打一个标记（并非所有的流都支持这个特性）。如果从输入流中已经读入的字节多于 `readlimit` 个，则这个流允许忽略这个标记。
- **void reset()**
返回到最后一个标记，随后对 `read` 的调用将重新读入这些字节。如果当前没有任何标记，则这个流不被重置。
- **boolean markSupported()**
如果这个流支持打标记，则返回 `true`。

API **java.io.OutputStream 1.0**

- **abstract void write(int n)**
写出一个字节的数据。
- **void write(byte[] b)**

- void write(byte[] b, int off, int len)
写出所有字节或者某个范围的字节到数组 b 中。
 - void close()
冲刷并关闭输出流。
 - void flush()
冲刷输出流，也就是将所有缓冲的数据发送到目的地。

2.1.2 完整的流家族

与 C 语言只有单一类型 FILE* 包打天下不同, Java 拥有一个流家族, 包含各种输入 / 输出流类型, 其数量超过 60 个! 请参见图 2-1 和图 2-2。

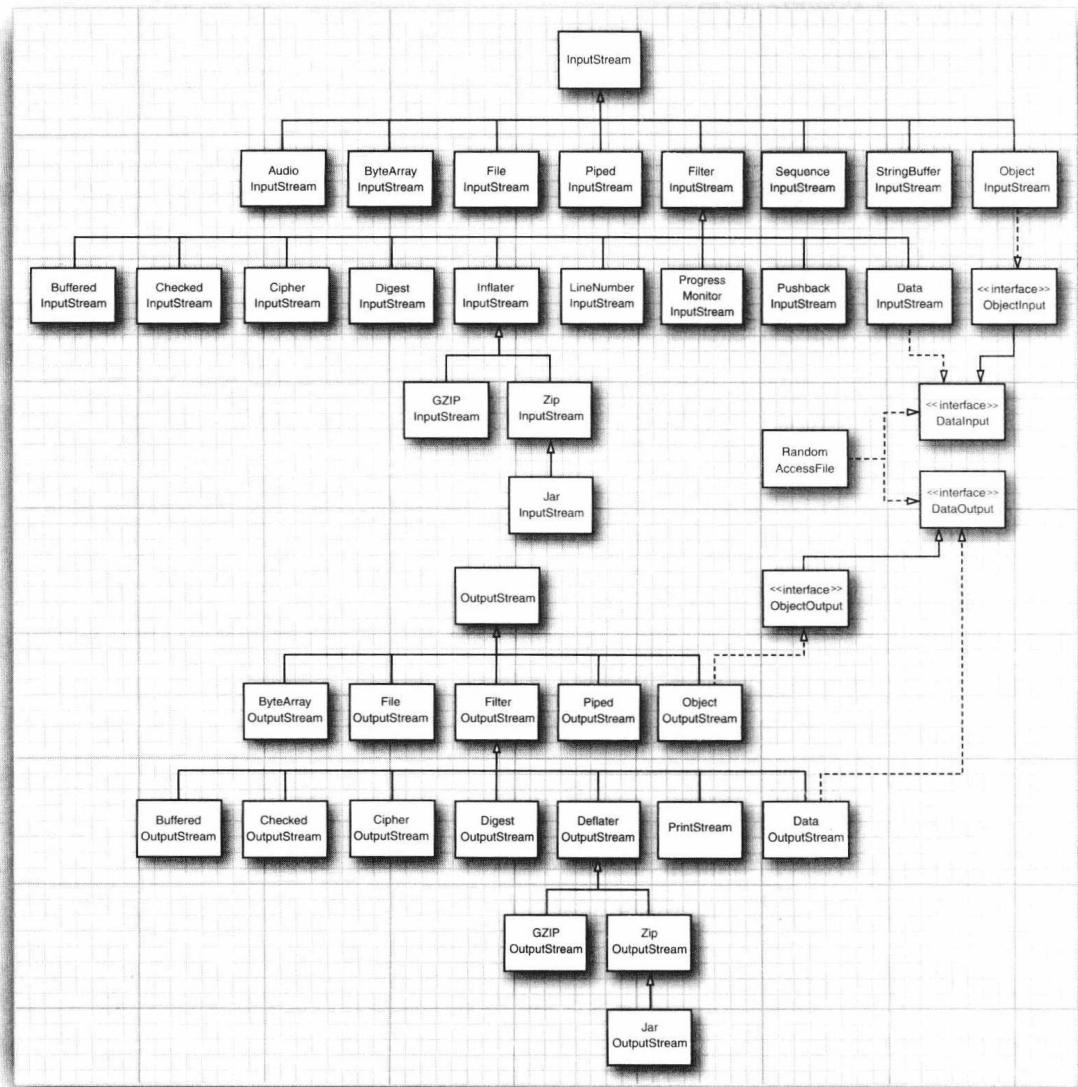


图 2-1 输入流与输出流的层次结构

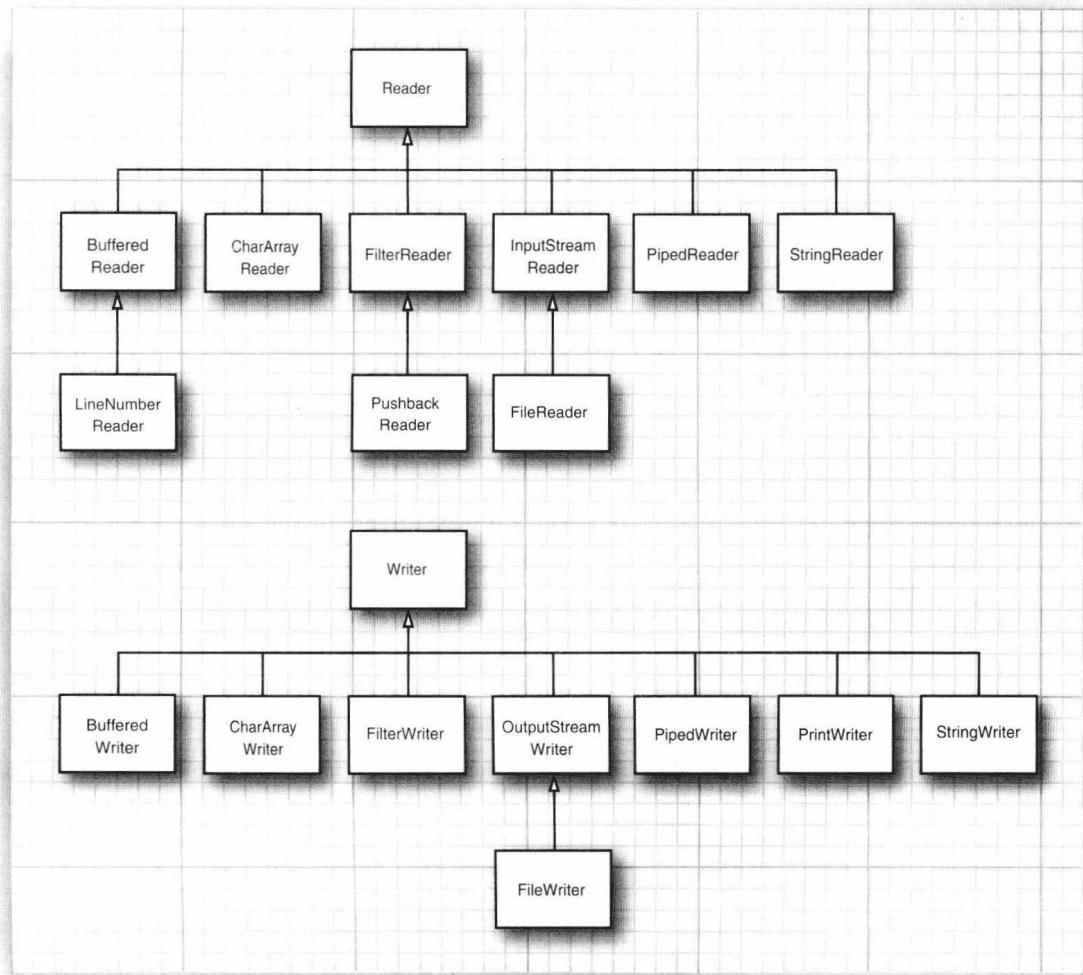


图 2-2 Reader 和 Writer 的层次结构

让我们把输入 / 输出流家族中的成员按照它们的使用方法来进行划分，这样就形成了处理字节和字符的两个单独的层次结构。正如所见，`InputStream` 和 `OutputStream` 类可以读写单个字节或字节数组，这些类构成了图 2-1 所示的层次结构的基础。要想读写字符串和数字，就需要功能更强大的子类，例如，`DataInputStream` 和 `DataOutputStream` 可以以二进制格式读写所有的基本 Java 类型。最后，还包含了多个很有用的输入 / 输出流，例如，`ZipInputStream` 和 `ZipOutputStream` 可以以我们常见的 ZIP 压缩格式读写文件。

另一方面，对于 Unicode 文本，可以使用抽象类 `Reader` 和 `Writer` 的子类（请参见图 2-2）。`Reader` 和 `Writer` 类的基本方法与 `InputStream` 和 `OutputStream` 中的方法类似。

```
abstract int read()
abstract void write(int c)
```

`read` 方法将返回一个 Unicode 码元（一个在 0 ~ 65535 之间的整数），或者在碰到文件结

尾时返回 -1。write方法在被调用时，需要传递一个 Unicode 码元（请查看卷 I 第 3 章有关 Unicode 码元的讨论）。

还有 4 个附加的接口：Closeable、Flushable、Readable 和 Appendable（请查看图 2-3）。前两个接口非常简单，它们分别拥有下面的方法：

```
void close() throws IOException
```

和

```
void flush()
```

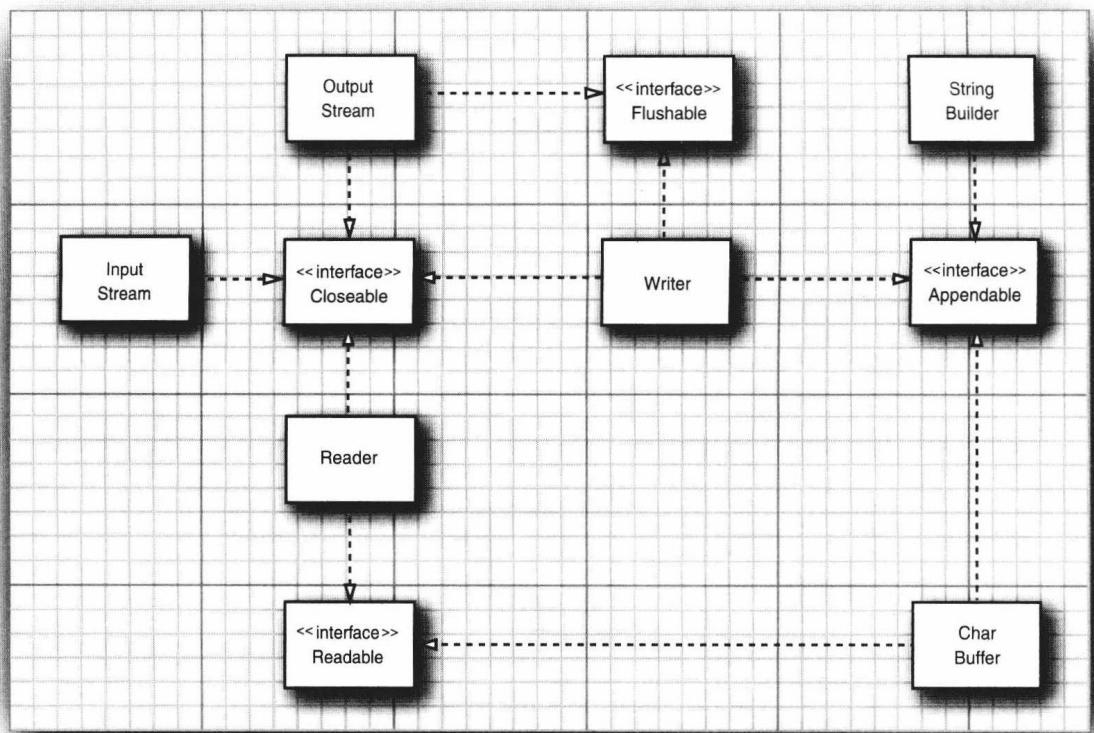


图 2-3 Closeable、Flushable、Readable 和 Appendable 接口

InputStream、OutputStream、Reader 和 Writer 都实现了 Closeable 接口。

注释：java.io.Closeable 接口扩展了 java.lang.AutoCloseable 接口。因此，对任何 Closeable 进行操作时，都可以使用 try-with-resource 语句[⊖]。为什么要有两个接口呢？因为 Closeable 接口的 close 方法只抛出 IOException，而 AutoCloseable.close 方法可以抛出任何异常。

而 OutputStream 和 Writer 还实现了 Flushable 接口。

[⊖] try-with-resource 语句是指声明了一个或多个资源的 try 语句。——译者注

Readable 接口只有一个方法：

```
int read(CharBuffer cb)
```

CharBuffer 类拥有按顺序和随机地进行读写访问的方法，它表示一个内存中的缓冲区或者一个内存映像的文件（请参见 2.5.2 节以了解细节）。

Appendable 接口有两个用于添加单个字符和字符序列的方法：

```
Appendable append(char c)
Appendable append(CharSequence s)
```

CharSequence 接口描述了一个 char 值序列的基本属性，String、CharBuffer、StringBuilder 和 StringBuffer 都实现了它。

在流类的家族中，只有 Writer 实现了 Appendable。

API **java.io.Closeable 5.0**

- void close()

关闭这个 Closeable，这个方法可能会抛出 IOException。

API **java.io.Flushable 5.0**

- void flush()

冲刷这个 Flushable。

API **java.lang.Readable 5.0**

- int read(CharBuffer cb)

尝试着向 cb 读入其可持有数量的 char 值。返回读入的 char 值的数量，或者当从这个 Readable 中无法再获得更多的值时返回 -1。

API **java.lang.Appendable 5.0**

- Appendable append(char c)
- Appendable append(CharSequence cs)

向这个 Appendable 中追加给定的码元或者给定的序列中的所有码元，返回 this。

API **java.lang.CharSequence 1.4**

- char charAt(int index)

返回给定索引处的码元。

- int length()

返回在这个序列中的码元的数量。

- CharSequence subSequence(int startIndex, int endIndex)

返回由存储在 startIndex 到 endIndex-1 处的所有码元构成的 CharSequence。

- String toString()

返回这个序列中所有码元构成的字符串。

2.1.3 组合输入 / 输出流过滤器

`FileInputStream` 和 `FileOutputStream` 可以提供附着在一个磁盘文件上的输入流和输出流，而你只需向其构造器提供文件名或文件的完整路径名。例如：

```
var fin = new FileInputStream("employee.dat");
```

这行代码可以查看用户目录下名为“`employee.dat`”的文件。

 提示：所有在 `java.io` 中的类都将相对路径名解释为以用户工作目录开始，你可以通过调用 `System.getProperty("user.dir")` 来获得这个信息。

 警告：由于反斜杠字符在 Java 字符串中是转义字符，因此要确保在 Windows 风格的路径名中使用 `\`（例如，`C:\\Windows\\win.ini`）。在 Windows 中，还可以使用单斜杠字符（`C:/Windows/win.ini`），因为大部分 Windows 文件处理的系统调用都会将斜杠解释成文件分隔符。但是，并不推荐这样做，因为 Windows 系统函数的行为会因与时俱进而发生变化。因此，对于可移植的程序来说，应该使用程序所运行平台的文件分隔符，我们可以通过常量字符串 `java.io.File.separator` 获得它。

与抽象类 `InputStream` 和 `OutputStream` 一样，这些类只支持在字节级别上的读写。也就是说，我们只能从 `fin` 对象中读入字节和字节数组。

```
byte b = (byte) fin.read();
```

正如下节中看到的，如果我们只有 `DataInputStream`，那么我们就只能读入数值类型：

```
DataInputStream din = . . .;
double x = din.readDouble();
```

但是正如 `FileInputStream` 没有任何读入数值类型的方法一样，`DataInputStream` 也没有任何从文件中获取数据的方法。

Java 使用了一种灵巧的机制来分离这两种职责。某些输入流（例如 `FileInputStream` 和由 `URL` 类的 `openStream` 方法返回的输入流）可以从文件和其他更外部的位置上获取字节，而其他的输入流（例如 `DataInputStream`）可以将字节组装到更有用的数据类型中。Java 程序员必须对二者进行组合。例如，为了从文件中读入数字，首先需要创建一个 `FileInputStream`，然后将其传递给 `DataInputStream` 的构造器：

```
var fin = new FileInputStream("employee.dat");
var din = new DataInputStream(fin);
double x = din.readDouble();
```

如果再次查看图 2-1，你就会看到 `FilterInputStream` 和 `FilterOutputStream` 类。这些文件的子类用于向处理字节的输入 / 输出流添加额外的功能。

你可以通过嵌套过滤器来添加多重功能。例如，输入流在默认情况下是不被缓冲区缓存的，也就是说，每个对 `read` 的调用都会请求操作系统再分发一个字节。相比之下，请求一个数据块并将其置于缓冲区中会显得更加高效。如果我们想使用缓冲机制和用于文件的数据输

入方法，那么就需要使用下面这种相当复杂的构造器序列：

```
var din = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

注意，我们把 `DataInputStream` 置于构造器链的最后，这是因为我们希望使用 `DataInputStream` 的方法，并且希望它们能够使用带缓冲机制的 `read` 方法。

有时当多个输入流链接在一起时，你需要跟踪各个中介输入流 (intermediate input stream)。例如，当读入输入时，你经常需要预览下一个字节，以了解它是否是你想要的值。Java 提供了用于此目的的 `PushbackInputStream`：

```
var pbin = new PushbackInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

现在你可以预读下一个字节：

```
int b = pbin.read();
```

并且在它并非你所期望的值时将其推回流中。

```
if (b != '<') pbin.unread(b);
```

但是读入和推回是可应用于可回推 (pushback) 输入流的仅有方法。如果你希望能够预先浏览并且还可以读入数字，那么就需要一个既是可回推输入流，又是一个数据输入流的引用。

```
var din = new DataInputStream(
    pbin = new PushbackInputStream(
        new BufferedInputStream(
            new FileInputStream("employee.dat"))));
```

当然，在其他编程语言的输入 / 输出流类库中，诸如缓冲机制和预览等细节都是自动处理的。因此，相比较而言，Java 就有一点麻烦，它必须将多个流过滤器组合起来。但是，这种混合并匹配过滤器类以构建真正有用的输入 / 输出流序列的能力，将带来极大的灵活性，例如，你可以从一个 ZIP 压缩文件中通过使用下面的输入流序列来读入数字（请参见图 2-4）：

```
var zin = new ZipInputStream(new FileInputStream("employee.zip"));
var din = new DataInputStream(zin);
```

（请查看 2.3.3 节以了解更多有关 Java 处理 ZIP 文件功能的知识。）

API `java.io.FileInputStream 1.0`

- `FileInputStream(String name)`
- `FileInputStream(File file)`

使用由 `name` 字符串或 `file` 对象指定路径名的文件创建一个新的文件输入流 (`File` 类在本章结尾处描述)。非绝对的路径名将按照相对于 VM 启动时所设置的工作目录来解析。

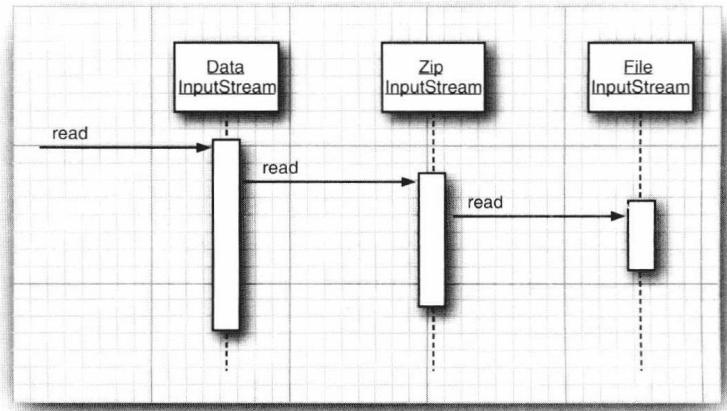


图 2-4 过滤器流序列

API java.io.FileOutputStream 1.0

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`
- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

使用由 `name` 字符串或 `file` 对象指定路径名的文件创建一个新的文件输出流（`File` 类在本章结尾处描述）。如果 `append` 参数为 `true`，那么数据将被添加到文件尾，而具有相同名字的已有文件不会被删除；否则，这个方法会删除所有具有相同名字的已有文件。

API java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

创建一个带缓冲区的输入流。带缓冲区的输入流在从流中读入字符时，不会每次都访问设备。当缓冲区为空时，会向缓冲区中读入一个新的数据块。

API java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

创建一个带缓冲区的输出流。带缓冲区的输出流在收集要写出的字符时，不会每次都访问设备。当缓冲区填满或当流被冲刷时，数据就被写出。

API java.io.PushbackInputStream 1.0

- `PushbackInputStream(InputStream in)`

- `PushbackInputStream(InputStream in, int size)`

构建一个可以预览一个字节或者具有指定尺寸的回推缓冲区的输入流。

- `void unread(int b)`

回推一个字节，它可以在下次调用 `read` 时被再次获取。

2.1.4 文本输入与输出

在保存数据时，可以选择二进制格式或文本格式。例如，整数 1234 存储成二进制数时，会被写为由字节 00 00 04 D2 构成的序列（十六进制表示法），而存储成文本格式时，则被存成了字符串“1234”。尽管二进制格式的 I/O 高速且高效，但是不适合人类阅读。我们首先讨论文本格式的 I/O，然后在 2.2 节中讨论二进制格式的 I/O。

在存储文本字符串时，需要考虑字符编码（character encoding）方式。在 Java 内部使用的 UTF-16 编码方式中，字符串“José”编码为 00 4A 00 6F 00 73 00 E9（十六进制）。但是，许多程序都希望文本文件按照其他的编码方式编码。在 UTF-8 这种在互联网上最常用的编码方式中，这个字符串将写出为 4A 6F 73 C3 A9，其中并没有用于前 3 个字母的任何 0 字节，而字符 é 占用了两个字节。

`OutputStreamWriter` 类将使用选定的字符编码方式，把 Unicode 码元的输出流转换为字节流。而 `InputStreamReader` 类将包含字节（用某种字符编码方式表示的字符）的输入流转换为可以产生 Unicode 码元的读入器。

例如，下面的代码就展示了如何让输入读入器从控制台读入键盘敲击信息，并将其转换为 Unicode：

```
var in = new InputStreamReader(System.in);
```

这个输入流读入器会假定使用主机系统所使用的默认字符编码方式。在桌面操作系统中，它可能是像 Windows 1252 或 MacRoman 这样古老的字符编码方式。你应该总是在 `InputStreamReader` 的构造器中选择一种具体的编码方式。例如，

```
var in = new InputStreamReader(new FileInputStream("data.txt"), StandardCharsets.UTF_8);
```

请查看 2.1.8 节以了解字符编码方式的更多信息。

`Reader` 和 `Writer` 类都只有读入和写出单个字符的基础方法。在使用流时，可以使用处理字符串和数字的子类。

2.1.5 如何写出文本输出

对于文本输出，可以使用 `PrintWriter`。这个类拥有以文本格式打印字符串和数字的方法。为了打印文件，需要用文件名和字符编码方式构建一个 `PrintStream` 对象：

```
var out = new PrintWriter("employee.txt", StandardCharsets.UTF_8);
```

为了输出到打印写出器，需要使用与使用 `System.out` 时相同的 `print`、`println` 和 `printf` 方法。你可以用这些方法来打印数字（`int`、`short`、`long`、`float`、`double`）、字符、`boolean` 值、字符串和对象。

例如，考虑下面的代码：

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

它将把字符

Harry Hacker 75000.0

输出到写出器 out，之后这些字符将会被转换成字节并最终写入 employee.txt 中。

`println` 方法在行中添加了对目标系统来说恰当的行结束符（Windows 系统是 "\r\n"，UNIX 系统是 "\n"），也就是通过调用 `System.getProperty("line.separator")` 而获得的字符串。

如果写出器设置为自动冲刷模式，那么只要 `println` 被调用，缓冲区中的所有字符都会被发送到它们的目的地（打印写出器总是带缓冲区的）。默认情况下，自动冲刷机制是禁用的，你可以通过使用 `PrintWriter(Writer writer, boolean autoFlush)` 来启用或禁用自动冲刷机制：

```
var out = new PrintWriter(
    new OutputStreamWriter(
        new FileOutputStream("employee.txt"), StandardCharsets.UTF_8),
    true); // autoflush
```

`print` 方法不抛出异常，你可以调用 `checkError` 方法来查看输出流是否出现了某些错误。

注释：Java 的老手们可能会很想知道 `PrintStream` 类和 `System.out` 到底怎么了。在 Java 1.0 中，`PrintStream` 类只是通过将高字节丢弃的方式把所有 Unicode 字符截断成 ASCII 字符。（那时，Unicode 仍旧是 16 位编码方式。）很明显，这并非一种干净利落和可移植的方式，这个问题在 Java 1.1 中通过引入读入器和写出器得到了修正。为了与已有的代码兼容，`System.in`、`System.out` 和 `System.err` 仍旧是输入 / 输出流而不是读入器和写出器。但是现在 `PrintStream` 类在内部采用与 `PrintWriter` 相同的方式将 Unicode 字符转换成了默认的主机编码方式。当你在使用 `print` 和 `println` 方法时，`PrintStream` 类型的对象的行为看起来确实很像打印写出器，但是与打印写出器不同的是，它们允许用 `write(int)` 和 `write(byte[])` 方法输出原生字节。

API `java.io.PrintWriter` 1.1

- `PrintWriter(Writer out)`
- `PrintWriter(Writer writer)`

创建一个向给定的写出器写出的新的 `PrintWriter`。

- `PrintWriter(String filename, String encoding)`
- `PrintWriter(File file, String encoding)`

创建一个使用给定的编码方式向给定的文件写出的新的 `PrintWriter`。

- `void print(Object obj)`

通过打印从 `toString` 产生的字符串来打印一个对象。

- `void print(String s)`

打印一个包含 Unicode 码元的字符串。

- `void println(String s)`

打印一个字符串，后面紧跟一个行终止符。如果这个流处于自动冲刷模式，那么就会

冲刷这个流。

- `void print(char[] s)`

打印在给定的字符串中的所有 Unicode 码元。

- `void print(char c)`

打印一个 Unicode 码元。

- `void print(int i)`

- `void print(long l)`

- `void print(float f)`

- `void print(double d)`

- `void print(boolean b)`

以文本格式打印给定的值。

- `void printf(String format, Object... args)`

按照格式字符串指定的方式打印给定的值。请查看卷 I 第 3 章以了解格式化字符串的相关规范。

- `boolean checkError()`

如果产生格式化或输出错误，则返回 `true`。一旦这个流碰到了错误，它就受到了污染，并且所有对 `checkError` 的调用都将返回 `true`。

2.1.6 如何读入文本输入

最简单的处理任意文本的方式就是使用在卷 I 中我们广泛使用的 `Scanner` 类。我们可以从任何输入流中构建 `Scanner` 对象。

或者，我们也可以将短小的文本文件像下面这样读入到一个字符串中：

```
var content = (Files.read String path, charset);
```

但是，如果想要将这个文件一行行地读入，那么可以调用：

```
List<String> lines = Files.readAllLines(path, charset);
```

如果文件太大，那么可以将惰性处理为一个 `Stream<String>` 对象：

```
try (Stream<String> lines = Files.lines(path, charset))
{
    ...
}
```

还可以使用扫描器来读入符号 (token)，即由分隔符分隔的字符串，默认的分隔符是空白字符。可以将分隔符修改为任意的正则表达式。例如，下面的代码

```
Scanner in = ...;
in.useDelimiter("\\PL+");
```

将接受任何非 Unicode 字母作为分隔符。之后，这个扫描器将只接受 Unicode 字母。

调用 `next` 方法可以产生下一个符号：

```

while (in.hasNext())
{
    String word = in.next();
    ...
}

```

或者，可以像下面这样获取一个包含所有符号的流：

```
Stream<String> words = in.tokens();
```

在早期的 Java 版本中，处理文本输入的唯一方式就是通过 `BufferedReader` 类。它的 `readLine` 方法会产生一行文本，或者在无法获得更多的输入时返回 `null`。典型的输入循环看起来像下面这样：

```

InputStream inputStream = . . .;
try (var in = new BufferedReader(new InputStreamReader(inputStream, charset)))
{
    String line;
    while ((line = in.readLine()) != null)
    {
        do something with line
    }
}

```

如今，`BufferedReader` 类又有了一个 `lines` 方法，可以产生一个 `Stream<String>` 对象。但是，与 `Scanner` 不同，`BufferedReader` 没有用于任何读入数字的方法。

2.1.7 以文本格式存储对象

在本节，我们将带你领略一个示例程序，它将一个 `Employee` 记录数组存储成了一个文本文件，其中每条记录都保存成单独的一行，而实例字段彼此之间使用分隔符分离开，这里我们使用竖线 (|) 作为分隔符 (冒号 (:) 是另一种流行的选择，有趣的是，每个人都会使用不同的分隔符)。因此，我们这里是在假设不会发生在要存储的字符串中存在 | 的情况。

下面是一个记录集的样本：

```

Harry Hacker|35500|1989-10-01
Carl Cracker|75000|1987-12-15
Tony Tester|38000|1990-03-15

```

写出记录相当简单，因为是要写出到一个文本文件中，所以我们使用 `PrintWriter` 类。我们直接写出所有的字段，每个字段后面跟着一个 |，而最后一个字段的后面跟着一个换行符。这项工作是在下面这个我们添加到 `Employee` 类中的 `writeEmployee` 方法里完成的：

```

public static void writeEmployee(PrintWriter out, Employee e)
{
    out.println(e.getName() + "|" + e.getSalary() + "|" + e.getHireDay());
}

```

为了读入记录，我们每次读入一行，然后分离所有的字段。我们使用一个扫描器来读入每一行，然后用 `String.split` 方法将这一行断开成一组标记。

```

public static Employee readEmployee(Scanner in)
{
    String line = in.nextLine();
    String[] tokens = line.split("\\|");
    String name = tokens[0];
    double salary = Double.parseDouble(tokens[1]);
    LocalDate hireDate = LocalDate.parse(tokens[2]);
    int year = hireDate.getYear();
    int month = hireDate.getMonthValue();
    int day = hireDate.getDayOfMonth();
    return new Employee(name, salary, year, month, day);
}

```

split 方法的参数是一个描述分隔符的正则表达式，我们在本章的末尾将详细讨论正则表达式。碰巧的是，竖线在正则表达式中具有特殊的含义，因此需要用\字符来表示转义，而这个\又需要用另一个\来转义，这样就产生了“\\|”表达式。

完整的程序如程序清单 2-1 所示。静态方法

```
void writeData(Employee[] e, PrintWriter out)
```

首先写出该数组的长度，然后写出每条记录。静态方法

```
Employee[] readData(BufferedReader in)
```

首先读入该数组的长度，然后读入每条记录。这显得稍微有点棘手：

```

int n = in.nextInt();
in.nextLine(); // consume newline
var employees = new Employee[n];
for (int i = 0; i < n; i++)
{
    employees[i] = new Employee();
    employees[i].readData(in);
}

```

对 nextInt 的调用读入的是数组长度，但不包括行尾的换行字符，我们必须处理掉这个换行符，这样，在调用 nextLine 方法后，readData 方法就可以获得下一行输入了。

程序清单 2-1 textFile/TextFileTest.java

```

1 package textFile;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.time.*;
6 import java.util.*;
7
8 /**
9  * @version 1.15 2018-03-17
10 * @author Cay Horstmann
11 */
12 public class TextFileTest
13 {
14     public static void main(String[] args) throws IOException
15     {

```

```
16     var staff = new Employee[3];
17
18     staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
19     staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
20     staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
21
22     // save all employee records to the file employee.dat
23     try (var out = new PrintWriter("employee.dat", StandardCharsets.UTF_8))
24     {
25         writeData(staff, out);
26     }
27
28     // retrieve all records into a new array
29     try (var in = new Scanner(
30         new FileInputStream("employee.dat"), "UTF-8"))
31     {
32         Employee[] newStaff = readData(in);
33
34         // print the newly read employee records
35         for (Employee e : newStaff)
36             System.out.println(e);
37     }
38 }
39
40 /**
41 * Writes all employees in an array to a print writer
42 * @param employees an array of employees
43 * @param out a print writer
44 */
45 private static void writeData(Employee[] employees, PrintWriter out)
46     throws IOException
47 {
48     // write number of employees
49     out.println(employees.length);
50
51     for (Employee e : employees)
52         writeEmployee(out, e);
53 }
54
55 /**
56 * Reads an array of employees from a scanner
57 * @param in the scanner
58 * @return the array of employees
59 */
60 private static Employee[] readData(Scanner in)
61 {
62     // retrieve the array size
63     int n = in.nextInt();
64     in.nextLine(); // consume newline
65
66     var employees = new Employee[n];
67     for (int i = 0; i < n; i++)
68     {
69         employees[i] = readEmployee(in);
```

```

70     }
71     return employees;
72 }
73
74 /**
75 * Writes employee data to a print writer
76 * @param out the print writer
77 */
78 public static void writeEmployee(PrintWriter out, Employee e)
79 {
80     out.println(e.getName() + " | " + e.getSalary() + " | " + e.getHireDay());
81 }
82
83 /**
84 * Reads employee data from a buffered reader
85 * @param in the scanner
86 */
87 public static Employee readEmployee(Scanner in)
88 {
89     String line = in.nextLine();
90     String[] tokens = line.split("\\|");
91     String name = tokens[0];
92     double salary = Double.parseDouble(tokens[1]);
93     LocalDate hireDate = LocalDate.parse(tokens[2]);
94     int year = hireDate.getYear();
95     int month = hireDate.getMonthValue();
96     int day = hireDate.getDayOfMonth();
97     return new Employee(name, salary, year, month, day);
98 }
99 }
```

2.1.8 字符编码方式

输入和输出流都是用于字节序列的，但是在许多情况下，我们希望操作的是文本，即字符序列。于是，字符如何编码成字节就成了问题。

Java 针对字符使用的是 Unicode 标准。每个字符或“编码点”都具有一个 21 位的整数。有多种不同的字符编码方式，也就是说，将这些 21 位数字包装成字节的方法有多种。

最常见的编码方式是 UTF-8，它会将每个 Unicode 编码点编码为 1 到 4 个字节的序列（请参阅表 2-1）。UTF-8 的好处是传统的包含了英语中用到的所有字符的 ASCII 字符集中的每个字符都只会占用一个字节。

表 2-1 UTF-8 编码方式

字符范围	编码方式
0...7F	0a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
80...7FF	110a ₈ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
800...FFFF	1110a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	11110a ₂₈ a ₁₉ a ₁₈ 10a ₁₇ a ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ 10a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ 10a ₅ a ₄ a ₃ a ₂ a ₁ a ₀

另一种常见的编码方式是 UTF-16，它会将每个 Unicode 编码点编码为 1 个或 2 个 16 位值（请参阅表 2-2）。这是一种在 Java 字符串中使用的编码方式。实际上，有两种形式的 UTF-16，被称为“高位优先”和“低位优先”。考虑一下 16 位值 0x2122。在高位优先格式中，高位字节会先出现：0x21 后面跟着 0x22。但是在低位优先格式中，是另外一种排列方式：0x22 0x21。为了表示使用的是哪一种格式，文件可以以“字节顺序标记”开头，这个标记为 16 位数值 0xFEFF。读入器可以使用这个值来确定字节顺序，然后丢弃它。

表 2-2 UTF-16 编码方式

字符范围	编码方式
0...FFFF	$a_{15}a_{14}a_{13}a_{12}a_{11}a_{10}a_9a_8 a_7a_6a_5a_4a_3a_2a_1a_0$
10000...10FFFF	$110110b_{19}b_{18}b_{17}b_{16}a_{15}a_{14}a_{13}a_{12}a_{11}a_{10} 110111a_9a_8a_7a_6a_5a_4a_3a_2a_1a_0$ 其中 $b_{19}b_{18}b_{17}b_{16} = a_{28}a_{19}a_{18}a_{17}a_{16} - 1$

! **警告：**有些程序，包括 Microsoft Notepad（微软记事本）在内，都在 UTF-8 编码的文件开头处添加了一个字节顺序标记。很明显，这并不需要，因为在 UTF-8 中，并不存在字节顺序的问题。但是 Unicode 标准允许这样做，甚至认为这是一种好的做法，因为这样做可以使编码机制不留疑惑。遗憾的是，Java 并没有这么做，有关这个问题的缺陷报告最终是以“will not fix”（不做修正）关闭的。对你来说，最好的做法是将输入中发现的所有先导的 \uFEFF 都剥离掉。

除了 UTF 编码方式，还有一些编码方式，它们各自都覆盖了适用于特定用户人群的字符范围。例如，ISO 8859-1 是一种单字节编码，它包含了西欧各种语言中用到的带有重音符号的字符，而 Shift-JIS 是一种用于日文字符的可变长编码。类似这些的大量编码方式至今仍被广泛使用。

不存在任何可靠的方式可以自动地探测出字节流中所使用的字符编码方式。某些 API 方法让我们使用“默认字符集”，即计算机的操作系统首选的字符编码方式。这种字符编码方式与我们的字节源中所使用的编码方式相同吗？字节源中的字节可能来自世界上的其他国家或地区，因此，你应该总是明确指定编码方式。例如，在编写网页时，应该检查 Content-Type 头信息。

! **注释：**平台使用的编码方式可以由静态方法 Charset.defaultCharset 返回。静态方法 Charset.availableCharsets 会返回所有可用的 Charset 实例，返回结果是一个从字符集的规范名称到 Charset 对象的映射表。

! **警告：**Oracle 的 Java 实现有一个用于覆盖平台默认值的系统属性 file.encoding。但是它并非官方支持的属性，并且 Java 库的 Oracle 实现的所有部分并非都以一致的方式处理该属性，因此，你不应该设置它。

StandardCharsets 类具有类型为 Charset 的静态变量，用于表示每种 Java 虚拟机都必须支持

的字符编码方式：

```
StandardCharsets.UTF_8
StandardCharsets.UTF_16
StandardCharsets.UTF_16BE
StandardCharsets.UTF_16LE
StandardCharsets.ISO_8859_1
StandardCharsets.US_ASCII
```

为了获得另一种编码方式的 Charset，可以使用静态的 `forName` 方法：

```
Charset shiftJIS = Charset.forName("Shift-JIS");
```

在读入或写出文本时，应该使用 `Charset` 对象。例如，我们可以像下面这样将一个字节数组转换为字符串：

```
var str = new String(bytes, StandardCharsets.UTF_8);
```

 **提示：**在 Java 10 中，`java.io` 包中的所有方法都允许我们用一个 `Charset` 对象或字符串来指定字符编码方式。应该选择的是 `StandardCharsets` 常量，这样就不会在编译时捕获到任何拼写错误了。

 **警告：**在不指定任何编码方式时，有些方法（例如 `String(byte[])` 构造器）会使用默认的平台编码方式，而其他方法（例如 `Files.readAllLines`）会使用 UTF-8。

2.2 读写二进制数据

文本格式对于测试和调试而言会显得很方便，因为它是人类可阅读的，但是它并不像以二进制格式传递数据那样高效。在下面的各小节中，你将会学习如何用二进制数据来完成输入和输出。

2.2.1 DataInput 和 DataOutput 接口

`DataOutput` 接口定义了下面用于以二进制格式写数组、字符、`boolean` 值和字符串的方法：

<code>writeChars</code>	<code>writeFloat</code>
<code>writeByte</code>	<code>writeDouble</code>
<code>writeInt</code>	<code>writeChar</code>
<code>writeShort</code>	<code>writeBoolean</code>
<code>writeLong</code>	<code>writeUTF</code>

例如，`.writeInt` 总是将一个整数写出为 4 字节的二进制数量值，而不管它有多少位，`writeDouble` 总是将一个 `double` 值写出为 8 字节的二进制数量值。这样产生的结果并非人可阅读的，但是对于给定类型的每个值，使用的空间都是相同的，而且将其读回也比解析文本要更快。

 **注释：**根据你所使用的处理器类型，在内存存储整数和浮点数有两种不同的方法。例如，假设你使用的是 4 字节的 `int`，如果有一个十进制数 1234，也就是十六进制的

4D2 ($1234 = 4 \times 256 + 13 \times 16 + 2$)，那么它可以按照内存中 4 字节的第一个字节存储最高位字节的方式来存储为 00 00 04 D2，这就是所谓的高位在前顺序 (MSB)；我们也可以从最低位字节开始，即 D2 04 00 00，这种方式自然就是所谓的低位在前顺序 (LSB)。例如，SPARC 使用的是高位在前顺序，而 Pentium 使用的则是低位在前顺序。这就可能会带来问题，当存储 C 或者 C++ 文件时，数据会精确地按照处理器存储它们的方式来存储，这就使得即使是最简单的数据在从一个平台迁移到另一个平台上时也是一种挑战。在 Java 中，所有的值都按照高位在前的模式写出，不管使用何种处理器，这使得 Java 数据文件可以独立于平台。

`writeUTF` 方法使用修订版的 8 位 Unicode 转换格式写出字符串。这种方式与直接使用标准的 UTF-8 编码方式不同，其中，Unicode 码元序列首先用 UTF-16 表示，其结果之后使用 UTF-8 规则进行编码。修订后的编码方式对于编码大于 0xFFFF 的字符的处理有所不同，这是为了向后兼容在 Unicode 还没有超过 16 位时构建的虚拟机。

因为没有其他方法会使用 UTF-8 的这种修订，所以你应该只在写出用于 Java 虚拟机的字符串时才使用 `writeUTF` 方法，例如，当你需要编写一个生成字节码的程序时。对于其他场合，都应该使用 `writeChars` 方法。

为了读回数据，可以使用在 `DataInput` 接口中定义的下列方法：

<code>readInt</code>	<code>readDouble</code>
<code>readShort</code>	<code>readChar</code>
<code>readLong</code>	<code>readBoolean</code>
<code>readFloat</code>	<code>readUTF</code>

`DataInputStream` 类实现了 `DataInput` 接口，为了从文件中读入二进制数据，可以将 `DataInputStream` 与某个字节源相组合，例如 `FileInputStream`：

```
var in = new DataInputStream(new FileInputStream("employee.dat"));
```

与此类似，要想写出二进制数据，你可以使用实现了 `DataOutput` 接口的 `DataOutputStream` 类：

```
var out = new DataOutputStream(new FileOutputStream("employee.dat"));
```

API `java.io.DataInput` 1.0

- `boolean readBoolean()`
- `byte readByte()`
- `char readChar()`
- `double readDouble()`
- `float readFloat()`
- `int readInt()`
- `long readLong()`
- `short readShort()`

读入一个给定类型的值。

- void `readFully(byte[] b)`
将字节读入到数组 `b` 中，其间阻塞直至所有字节都读入。
- void `readFully(byte[] b, int off, int len)`
将由 `len` 指定数量的字节放置到数组 `b` 从 `off` 开始的位置，其间阻塞直至所有字节都读入。
- String `readUTF()`
读入由“修订过的 UTF-8”格式的字符构成的字符串。
- int `skipBytes(int n)`
跳过 `n` 个字节，其间阻塞直至所有字节都被跳过。

API java.io.DataOutput 1.0

- void `writeBoolean(boolean b)`
- void `writeByte(int b)`
- void `writeChar(int c)`
- void `writeDouble(double d)`
- void `writeFloat(float f)`
- void `writeInt(int i)`
- void `writeLong(long l)`
- void `writeShort(int s)`
写出一个给定类型的值。
- void `writeChars(String s)`
写出字符串中的所有字符。
- void `writeUTF(String s)`
写出由“修订过的 UTF-8”格式的字符构成的字符串。

2.2.2 随机访问文件

`RandomAccessFile` 类可以在文件中的任何位置查找或写入数据。磁盘文件都是随机访问的，但是与网络套接字通信的输入 / 输出流却不是。你可以打开一个随机访问文件，只用于读入或者同时用于读写，你可以通过使用字符串“`r`”（用于读入访问）或“`rw`”（用于读入 / 写出访问）作为构造器的第二个参数来指定这个选项。

```
var in = new RandomAccessFile("employee.dat", "r");
var inOut = new RandomAccessFile("employee.dat", "rw");
```

当你将已有文件作为 `RandomAccessFile` 打开时，这个文件并不会被删除。

随机访问文件有一个表示下一个将被读入或写出的字节所处位置的文件指针，`seek` 方法可以用来将这个文件指针设置到文件中的任意字节位置，`seek` 的参数是一个 `long` 类型的整数，它的值位于 0 到文件按照字节来度量的长度之间。

`getFilePointer` 方法将返回文件指针的当前位置。

`RandomAccessFile` 类同时实现了 `DataInput` 和 `DataOutput` 接口。为了读写随机访问文件，可以使用在前面小节中讨论过的诸如 `readInt/writeInt` 和 `readChar/writeChar` 之类的方法。

我们现在要剖析一个将雇员记录存储到随机访问文件中的示例程序，其中每条记录都拥有相同的大小，这样我们可以很容易地读入任何记录。假设你希望将文件指针置于第三条记录处，那么你只需将文件指针置于恰当的字节位置，然后就可以开始读入了。

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
var e = new Employee();
e.readData(in);
```

如果你希望修改记录，然后将其存回到相同的位置，那么请切记要将文件指针置回到这条记录的开始处：

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

要确定文件中的字节总数，可以使用 `length` 方法，而记录的总数则等于用字节总数除以每条记录的大小。

```
long nbytes = in.length(); // length in bytes
int nrecords = (int) (nbytes / RECORD_SIZE);
```

整数和浮点值在二进制格式中都具有固定的尺寸，但是在处理字符串时就有些麻烦了，因此我们提供了两个助手方法来读写具有固定尺寸的字符串。

`writeFixedString` 写出从字符串开头开始的指定数量的码元（如果码元过少，该方法将用 0 值来补齐字符串）。

```
public static void writeFixedString(String s, int size, DataOutput out)
    throws IOException
{
    for (int i = 0; i < size; i++)
    {
        char ch = 0;
        if (i < s.length()) ch = s.charAt(i);
        out.writeChar(ch);
    }
}
```

`readFixedString` 方法从输入流中读入字符，直至读入 `size` 个码元，或者直至遇到具有 0 值的字符值，然后跳过输入字段中剩余的 0 值。为了提高效率，这个方法使用了 `StringBuilder` 类来读入字符串。

```
public static String readFixedString(int size, DataInput in)
    throws IOException
{
    var b = new StringBuilder(size);
    int i = 0;
    var done = false;
    while (!done && i < size)
    {
        char ch = in.readChar();
```

```
i++;
if (ch == 0) done = true;
else b.append(ch);
}
in.skipBytes(2 * (size - i));
return b.toString();
}
```

我们将 `writeFixedString` 和 `readFixedString` 方法放到了 `DataIO` 助手类的内部。

为了写出一条固定尺寸的记录，我们直接以二进制方式写出所有的字段：

```
DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
out.writeDouble(e.getSalary());
LocalDate hireDay = e.getHireDay();
out.writeInt(hireDay.getYear());
out.writeInt(hireDay.getMonthValue());
out.writeInt(hireDay.getDayOfMonth());
```

读回数据也很简单：

```
String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
double salary = in.readDouble();
int y = in.readInt();
int m = in.readInt();
int d = in.readInt();
```

让我们来计算每条记录的大小：我们将使用 40 个字符来表示姓名字符串，因此，每条记录包含 100 个字节：

- 40 字符 = 80 字节，用于姓名。
- 1 double = 8 字节，用于薪水。
- 3 int = 12 字节，用于日期。

程序清单 2-2 中所示的程序将三条记录写到了一个数据文件中，然后以逆序将它们从文件中读回。为了高效地执行，这里需要使用随机访问，因为我们需要首先读入第三条记录。

程序清单 2-2 randomAccess/RandomAccessTest.java

```
1 package randomAccess;
2
3 import java.io.*;
4 import java.time.*;
5
6 /**
7  * @version 1.14 2018-05-01
8  * @author Cay Horstmann
9  */
10 public class RandomAccessTest
11 {
12     public static void main(String[] args) throws IOException
13     {
14         var staff = new Employee[3];
15
16         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
17         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

```
18     staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
19
20     try (var out = new DataOutputStream(new FileOutputStream("employee.dat")))
21     {
22         // save all employee records to the file employee.dat
23         for (Employee e : staff)
24             writeData(out, e);
25     }
26
27     try (var in = new RandomAccessFile("employee.dat", "r"))
28     {
29         // retrieve all records into a new array
30
31         // compute the array size
32         int n = (int)(in.length() / Employee.RECORD_SIZE);
33         var newStaff = new Employee[n];
34
35         // read employees in reverse order
36         for (int i = n - 1; i >= 0; i--)
37         {
38             newStaff[i] = new Employee();
39             in.seek(i * Employee.RECORD_SIZE);
40             newStaff[i] = readData(in);
41         }
42
43         // print the newly read employee records
44         for (Employee e : newStaff)
45             System.out.println(e);
46     }
47 }
48
49 /**
50  * Writes employee data to a data output
51  * @param out the data output
52  * @param e the employee
53  */
54 public static void writeData(DataOutput out, Employee e) throws IOException
55 {
56     DataIO.writeFixedString(e.getName(), Employee.NAME_SIZE, out);
57     out.writeDouble(e.getSalary());
58
59     LocalDate hireDay = e.getHireDay();
60     out.writeInt(hireDay.getYear());
61     out.writeInt(hireDay.getMonthValue());
62     out.writeInt(hireDay.getDayOfMonth());
63 }
64
65 /**
66  * Reads employee data from a data input
67  * @param in the data input
68  * @return the employee
69  */
70 public static Employee readData(DataInput in) throws IOException
71 {
```

```

72     String name = DataIO.readFixedString(Employee.NAME_SIZE, in);
73     double salary = in.readDouble();
74     int y = in.readInt();
75     int m = in.readInt();
76     int d = in.readInt();
77     return new Employee(name, salary, y, m - 1, d);
78   }
79 }
```

API **java.io.RandomAccessFile 1.0**

- `RandomAccessFile(String file, String mode)`

- `RandomAccessFile(File file, String mode)`

打开给定的用于随机访问的文件。`mode` 字符串“`r`”表示只读模式；“`rw`”表示读 / 写模式；“`rws`”表示每次更新时，都对数据和元数据的写磁盘操作进行同步的读 / 写模式；“`rwd`”表示每次更新时，只对数据的写磁盘操作进行同步的读 / 写模式。

- `long getFilePointer()`

返回文件指针的当前位置。

- `void seek(long pos)`

将文件指针设置到距文件开头 `pos` 个字节处。

- `long length()`

返回文件按照字节来度量的长度。

2.2.3 ZIP 文档

ZIP 文档（通常）以压缩格式存储了一个或多个文件，每个 ZIP 文档都有一个头，包含诸如每个文件名字和所使用的压缩方法等信息。在 Java 中，可以使用 `ZipInputStream` 来读入 ZIP 文档。你可能需要浏览文档中每个单独的项，`getNextEntry` 方法就可以返回一个描述这些项的 `ZipEntry` 类型的对象。该方法会从流中读入数据直至末尾，实际上这里的末尾是指正在读入的项的末尾，然后调用 `closeEntry` 来读入下一项。在读入最后一项之前，不要关闭 `zin`。下面是典型的通读 ZIP 文件的代码序列：

```

var zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    read the contents of zin
    zin.closeEntry();
}
zin.close();
```

要写出到 ZIP 文件，可以使用 `ZipOutputStream`，而对于你希望放入到 ZIP 文件中的每一项，都应该创建一个 `ZipEntry` 对象，并将文件名传递给 `ZipEntry` 的构造器，它将设置其他诸如文件日期和解压缩方法等参数。如果需要，你可以覆盖这些设置。然后，你需要调用 `ZipOutputStream` 的 `putNextEntry` 方法来写出新文件，并将文件数据发送到 ZIP 输出流中。当完

成时，需要调用 `closeEntry`。然后，你需要对所有希望存储的文件都重复这个过程。下面是代码框架：

```
var fout = new FileOutputStream("test.zip");
var zout = new ZipOutputStream(fout);
for all files
{
    var ze = new ZipEntry(filename);
    zout.putNextEntry(ze);
    send data to zout
    zout.closeEntry();
}
zout.close();
```

注释：JAR 文件（在卷 I 第 4 章中讨论过）只是带有一个特殊项的 ZIP 文件，这个项称作清单。你可以使用 `JarInputStream` 和 `JarOutputStream` 类来读写清单项。

ZIP 输入流是一个能够展示流的抽象化的强大之处的实例。当你读入以压缩格式存储的数据时，不必担心边请求边解压数据的问题，而且 ZIP 格式的字节源并非必须是文件，也可以是来自网络连接的 ZIP 数据。

注释：2.4.8 节将展示如何使用 Java 7 的 `FileSystem` 类而无须特殊 API 来访问 ZIP 文档。

API `java.util.zip.ZipInputStream` 1.1

- `ZipInputStream(InputStream in)`
创建一个 `ZipInputStream`，使得我们可以从给定的 `InputStream` 向其中填充数据。
- `ZipEntry getNextEntry()`
为下一项返回 `ZipEntry` 对象，或者在没有更多的项时返回 `null`。
- `void closeEntry()`
关闭这个 ZIP 文件中当前打开的项。之后可以通过使用 `getNextEntry()` 读入下一项。

API `java.util.zip.ZipOutputStream` 1.1

- `ZipOutputStream(OutputStream out)`
创建一个将压缩数据写出到指定的 `OutputStream` 的 `ZipOutputStream`。
- `void putNextEntry(ZipEntry ze)`
将给定的 `ZipEntry` 中的信息写出到输出流中，并定位用于写出数据的流，然后这些数据可以通过 `write()` 写出到这个输出流中。
- `void closeEntry()`
关闭这个 ZIP 文件中当前打开的项。使用 `putNextEntry` 方法可以开始下一项。
- `void setLevel(int level)`
将后续的各个 `DEFLATED` 项的默认压缩级别设置为从 `Deflater.NO_COMPRESSION` 到 `Deflater.BEST_COMPRESSION` 中的某个值，默认值是 `Deflater.DEFAULT_COMPRESSION`。如果级别无效，则

抛出 `IllegalArgumentException`。

- `void setMethod(int method)`

设置用于这个 `ZipOutputStream` 的默认压缩方法，这个压缩方法会作用于所有没有指定压缩方法的项。`method` 可以是 `DEFLATED` 或 `STORED`。

API `java.util.zip.ZipEntry 1.1`

- `ZipEntry(String name)`

用给定的名字构建一个 ZIP 项。

- `long getCrc()`

返回用于这个 `ZipEntry` 的 CRC32 校验和的值。

- `String getName()`

返回这一项的名字。

- `long getSize()`

返回这一项未压缩的尺寸，或者在未压缩的尺寸不可知的情况下返回 -1。

- `boolean isDirectory()`

当这一项是目录时返回 `true`。

- `void setMethod(int method)`

设置用于这一项的压缩方法，必须是 `DEFLATED` 或 `STORED`。

- `void setSize(long size)`

设置这一项的尺寸，只有在压缩方法是 `STORED` 时才是必需的。

- `void setCrc(long crc)`

给这一项设置 CRC32 校验和，这个校验和是使用 CRC32 类计算的。只有在压缩方法是 `STORED` 时才是必需的。

API `java.util.zip.ZipFile 1.1`

- `ZipFile(String name)`

- `ZipFile(File file)`

创建一个 `ZipFile`，用于从给定的字符串或 `File` 对象中读入数据。

- `Enumeration entries()`

返回一个 `Enumeration` 对象，它枚举了描述这个 `ZipFile` 中各个项的 `ZipEntry` 对象。

- `ZipEntry getEntry(String name)`

返回给定名字所对应的项，或者在没有对应项的时候返回 `null`。

- `InputStream getInputStream(ZipEntry ze)`

返回用于给定项的 `InputStream`。

- `String getName()`

返回这个 ZIP 文件的路径。

2.3 对象输入 / 输出流与序列化

当你需要存储相同类型的数据时，使用固定长度的记录格式是一个不错的选择。但是，在面向对象程序中创建的对象很少全部都具有相同的类型。例如，你可能有一个称为 `staff` 的数组，它名义上是一个 `Employee` 记录数组，但是实际上却包含诸如 `Manager` 这样的子类实例。

我们当然可以自己设计出一种数据格式来存储这种多态集合，但是幸运的是，我们并不需要这么做。Java 语言支持一种称为对象序列化（object serialization）的非常通用的机制，它可以将任何对象写出到输出流中，并在之后将其读回。（你将在本章稍后看到“序列化”这个术语的出处。）

2.3.1 保存和加载序列化对象

为了保存对象数据，首先需要打开一个 `ObjectOutputStream` 对象：

```
var out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

现在，为了保存对象，可以直接使用 `ObjectOutputStream` 的 `writeObject` 方法，如下所示：

```
var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
var boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);

out.writeObject(harry);
out.writeObject(boss);
```

为了将这些对象读回，首先需要获得一个 `ObjectInputStream` 对象：

```
var in = new ObjectInputStream(new FileInputStream("employee.dat"));
```

然后，用 `readObject` 方法以这些对象被写出时的顺序获得它们：

```
var e1 = (Employee) in.readObject();
var e2 = (Employee) in.readObject();
```

但是，对希望在对象输出流中存储或从对象输入流中恢复的所有类都应进行一下修改，这些类必须实现 `Serializable` 接口：

```
class Employee implements Serializable { . . . }
```

`Serializable` 接口没有任何方法，因此你不需要对这些类做任何改动。在这一点上，它与在卷 I 第 6 章中讨论过的 `Cloneable` 接口很相似。但是，为了使类可克隆，你仍旧需要覆盖 `Object` 类中的 `clone` 方法，而为了使类可序列化，你不需要做任何事。

 **注释：**你只有在写出对象时才能用 `writeObject/readObject` 方法，对于基本类型值，你需要使用诸如 `writeInt/readInt` 或 `writeDouble/readDouble` 这样的方法。（对象流类都实现了 `DataInput/DataOutput` 接口。）

在幕后，是 `ObjectOutputStream` 在浏览对象的所有域，并存储它们的内容。例如，当写出一个 `Employee` 对象时，其名字、日期和薪水域都会被写出到输出流中。

但是，有一种重要的情况需要考虑：当一个对象被多个对象共享，作为它们各自状态的

一部分时，会发生什么呢？

为了说明这个问题，我们对 Manager 类稍微做些修改，假设每个经理都有一个秘书：

```
class Manager extends Employee
{
    private Employee secretary;
    ...
}
```

现在每个 Manager 对象都包含一个表示秘书的 Employee 对象的引用，当然，两个经理可以共用一个秘书，正如图 2-5 和下面的代码所示的那样：

```
var harry = new Employee("Harry Hacker", ...);
var carl = new Manager("Carl Cracker", ...);
carl.setSecretary(harry);
var tony = new Manager("Tony Tester", ...);
tony.setSecretary(harry);
```

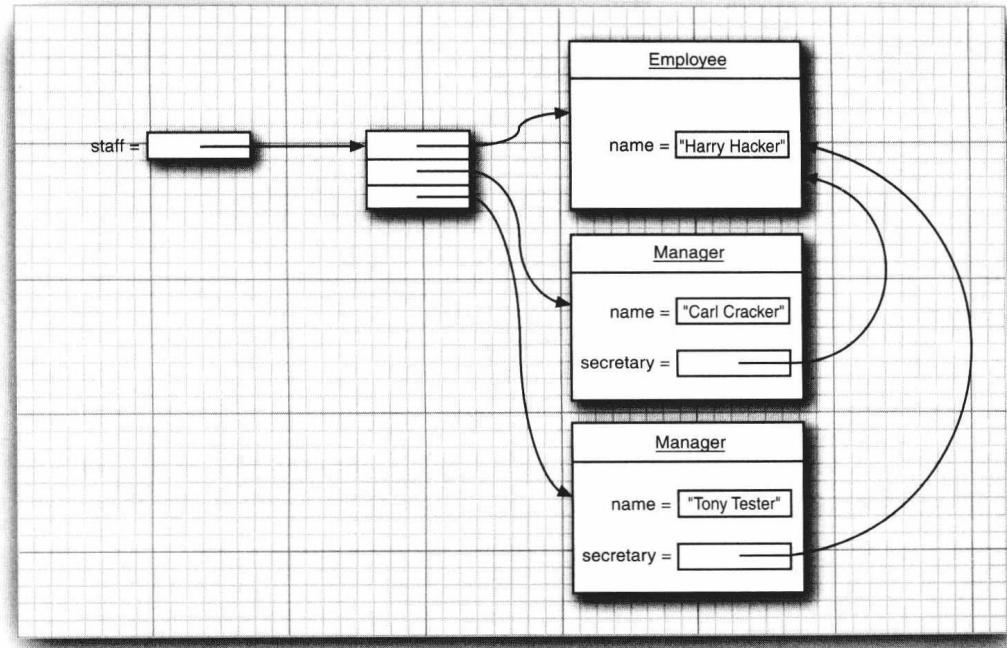


图 2-5 两个经理可以共用一个共有的雇员

保存这样的对象网络是一种挑战，在这里我们当然不能去保存和恢复秘书对象的内存地址，因为当对象被重新加载时，它可能占据的是与原来完全不同的内存地址。

与此不同的是，每个对象都是用一个序列号（serial number）保存的，这就是这种机制之所以称为对象序列化的原因。下面是其算法：

- 对你遇到的每一个对象引用都关联一个序列号（如图 2-6 所示）。
- 对于每个对象，当第一次遇到时，保存其对象数据到输出流中。
- 如果某个对象之前已经被保存过，那么只写出“与之前保存过的序列号为 x 的对象相同”。

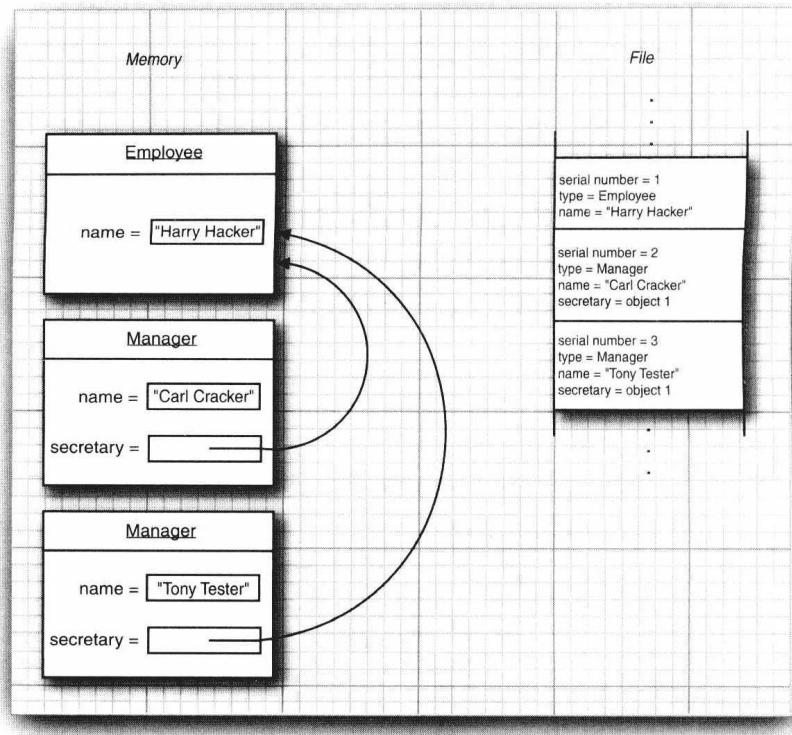


图 2-6 一个对象序列化的实例

在读回对象时，整个过程是反过来的。

- 对于对象输入流中的对象，在第一次遇到其序列号时，构建它，并使用流中数据来初始化它，然后记录这个顺序号和新对象之间的关联。
- 当遇到“与之前保存过的序列号为 x 的对象相同”这一标记时，获取与这个序列号相关联的对象引用。

注释：在本章中，我们使用序列化将对象集合保存到磁盘文件中，并按照它们被存储的样子获取它们。序列化的另一种非常重要的应用是通过网络将对象集合传送到另一台计算机上。正如在文件中保存原生的内存地址毫无意义一样，这些地址对于在不同的处理器之间的通信也是毫无意义的。因为序列化用序列号代替了内存地址，所以它允许将对象集合从一台机器传送到另一台机器。

程序清单 2-3 是保存和重新加载 Employee 和 Manager 对象网络的代码（有些对象共享相同的表示秘书的雇员）。注意，秘书对象在重新加载之后是唯一的，当 newStaff[1] 被恢复时，它会反映到经理们的 secretary 域中。

程序清单 2-3 objectStream/ObjectStreamTest.java

```
1 package objectStream;
2
```

```

3 import java.io.*;
4
5 /**
6  * @version 1.11 2018-05-01
7  * @author Cay Horstmann
8 */
9 class ObjectStreamTest
10 {
11     public static void main(String[] args) throws IOException, ClassNotFoundException
12     {
13         var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
14         var carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
15         carl.setSecretary(harry);
16         var tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
17         tony.setSecretary(harry);
18
19         var staff = new Employee[3];
20
21         staff[0] = carl;
22         staff[1] = harry;
23         staff[2] = tony;
24
25         // save all employee records to the file employee.dat
26         try (var out = new ObjectOutputStream(new FileOutputStream("employee.dat")))
27         {
28             out.writeObject(staff);
29         }
30
31         try (var in = new ObjectInputStream(new FileInputStream("employee.dat")))
32         {
33             // retrieve all records into a new array
34
35             var newStaff = (Employee[]) in.readObject();
36
37             // raise secretary's salary
38             newStaff[1].raiseSalary(10);
39
40             // print the newly read employee records
41             for (Employee e : newStaff)
42                 System.out.println(e);
43         }
44     }
45 }

```

API `java.io.ObjectOutputStream` 1.1

- `ObjectOutputStream(OutputStream out)`

创建一个 `ObjectOutputStream` 使得你可以将对象写出到指定的 `OutputStream`。

- `void writeObject(Object obj)`

写出指定的对象到 `ObjectOutputStream`，这个方法将存储指定对象的类、类的签名以及这个类及其超类中所有非静态和非瞬时的域的值。

API `java.io.ObjectInputStream 1.1`

- `ObjectInputStream(InputStream in)`

创建一个 `ObjectInputStream` 用于从指定的 `InputStream` 中读回对象信息。

- `Object readObject()`

从 `ObjectInputStream` 中读入一个对象。特别是，这个方法会读回对象的类、类的签名以及这个类及其超类中所有非静态和非瞬时的域的值。它执行的反序列化允许恢复多个对象引用。

2.3.2 理解对象序列化的文件格式

对象序列化是以特殊的文件格式存储对象数据的，当然，我们不必了解文件中表示对象的确切字节序列，就可以使用 `writeObject/readObject` 方法。但是，我们发现研究这种数据格式对于洞察对象流化的处理过程非常有益。因为其细节显得有些专业，所以如果你对其实现不感兴趣，则可以跳过这一节。

每个文件都是以下面这两个字节的“魔幻数字”开始的

AC ED

后面紧跟着对象序列化格式的版本号，目前是

00 05

(我们在本节中统一使用十六进制数字来表示字节。) 然后是它包含的对象序列，其顺序即它们存储的顺序。

字符串对象被存为

74 两字节表示的字符串长度 所有字符

例如，字符串“Harry”被存为

74 00 05 Harry

字符串中的 Unicode 字符被存储为修订过的 UTF-8 格式。

当存储一个对象时，这个对象所属的类也必须存储。这个类的描述包含

- 类名。
- 序列化的版本唯一的 ID，它是数据域类型和方法签名的指纹。
- 描述序列化方法的标志集。
- 对数据域的描述。

指纹是通过对类、超类、接口、域类型和方法签名按照规范方式排序，然后将安全散列算法 (SHA) 应用于这些数据而获得的。

SHA 是一种可以为较大的信息块提供指纹的快速算法，不论最初的数据块尺寸有多大，这种指纹总是 20 个字节的数据包。它是通过在数据上执行一个灵巧的位操作序列而创建的，这个序列在本质上可以百分之百地保证无论这些数据以何种方式发生变化，其指纹也都会跟着变化。(关于 SHA 的更多细节，可以查看一些参考资料，例如 William Stallings 所著的

Cryptography and Network Security: Principles and Practice (第 7 版, Prentice Hall, 2016)。但是, 序列化机制只使用了 SHA 码的前 8 个字节作为类的指纹。即便这样, 当类的数据域或方法发生变化时, 其指纹跟着变化的可能性还是非常大。

在读入一个对象时, 会拿其指纹与它所属的类的当前指纹进行比对, 如果它们不匹配, 那么就说明这个类的定义在该对象被写出之后发生过变化, 因此会产生一个异常。在实际情况下, 类当然是会演化的, 因此对于程序来说, 读入较旧版本的对象可能是必需的。我们将在 2.4.5 节中讨论这个问题。

下面表示了类标识符是如何存储的:

- 72
- 2 字节的类名长度
- 类名
- 8 字节长的指纹
- 1 字节长的标志
- 2 字节长的数据域描述符的数量
- 数据域描述符
- 78 (结束标记)
- 超类型 (如果没有就是 70)

标志字节是由在 `java.io.ObjectStreamConstants` 中定义的 3 位掩码构成的:

```
static final byte SC_WRITE_METHOD = 1;
    // class has a writeObject method that writes additional data
static final byte SC_SERIALIZABLE = 2;
    // class implements the Serializable interface
static final byte SC_EXTERNALIZABLE = 4;
    // class implements the Externalizable interface
```

我们会在本章稍后讨论 `Externalizable` 接口。可外部化的类提供了定制的接管其实例域输出的读写方法。我们要写出的这些类实现了 `Serializable` 接口, 并且其标志值为 02, 而可序列化的 `java.util.Date` 类定义了它自己的 `readObject/writeObject` 方法, 并且其标志值为 03。

每个数据域描述符的格式如下:

- 1 字节长的类型编码
- 2 字节长的域名长度
- 域名
- 类名 (如果域是对象)

其中类型编码是下列取值之一:

B	byte
C	char
D	double
F	float
I	int

J	long
L	对象
S	short
Z	boolean
[数组

当类型编码为 L 时，域名后面紧跟域的类型。类名和域名字符串不是以字符串编码 74 开头的，但域类型是。域类型使用的是与域名稍有不同的编码机制，即本地方法使用的格式。

例如，Employee 类的薪水域被编码为：

D 00 06 salary

下面是 Employee 类完整的类描述符：

72 00 08 Employee

E6 D2 86 7D AE AC 18 1B 02	指纹和标志
00 03	实例域的数量
D 00 06 salary	实例域的类型和名字
L 00 07 hireDay	实例域的类型和名字
74 00 10 Ljava/util/Date;	实例域的类名: Date
L 00 04 name	实例域的类型和名字
74 00 12 Ljava/lang/String;	实例域的类名: String
78	结束标记
70	无超类

这些描述符相当长，如果在文件中再次需要相同的类描述符，可以使用一种缩写版：

71 4 字节长的序列号

这个序列号将引用前面已经描述过的类描述符，我们稍后将讨论编号模式。

对象将被存储为：

73 类描述符 对象数据

例如，下面展示的就是 Employee 对象如何存储：

40 E8 6A 00 00 00 00 00	salary 域的值: double
73	hireDate 域的值: 新对象
71 00 7E 00 08	已有的类 java.util.Date
77 08 00 00 00 91 1B 4E B1 80 78	外部存储，稍后讨论细节

74 00 0C Harry Hacker name 域的值: String

正如你所看见的，数据文件包含了足够的信息来恢复这个 Employee 对象。

数组总是被存储成下面的格式：

75 类描述符 4 字节长的数组项的数量 数组项

在类描述符中的数组类名的格式与本地方法中使用的格式相同（它与在其他的类描述符中的类名稍微有些差异）。在这种格式中，类名以 L 开头，以分号结束。

例如，3个 Employee 对象构成的数组写出时就像下面一样：

75	数组
72 00 0B [LEmployee;	新类、字符串长度、类名Employee[]]
FC BF 36 11 C5 91 11 C7 02	指纹和标志
00 00	实例域的数量
78	结束标记
70	无超类
00 00 00 03	数组项的数量

注意，Employee 对象数组的指纹与 Employee 类自身的指纹并不相同。

所有对象（包含数组和字符串）和所有的类描述符在存储到输出文件时都被赋予了一个序列号，这个数字以 00 7E 00 00 开头。

我们已经看到过，任何给定类的完整类描述符只保存一次，后续的描述符将引用它。例如，在前面的示例中，对 Date 类的重复引用就被编码为：

71 00 7E 00 08

相同的机制还被用于对象。如果要写出一个对之前存储过的对象的引用，那么这个引用也会以完全相同的方式存储，即 71 后面跟随序列号，从上下文中可以很清楚地了解这个特殊的序列引用表示的是类描述符还是对象。

最后，空引用被存储为：

70

下面是前面小节中 ObjectRefTest 程序的带注释的输出。如果你喜欢，可以运行这个程序，然后查看其数据文件 employee.dat 的十六进制码，并将其与注释列表比较。在输出中接近结束部分的几行重要编码展示了对之前存储过的对象的引用。

AC ED 00 05	文件头
75	数组 staff (序列#1)
72 00 0B [LEmployee;	新类、字符串长度、类名Employee[] (序列#0)
FC BF 36 11 C5 91 11 C7 02	指纹和标志
00 00	实例域的数量
78	结束标记
70	无超类
00 00 00 03	数组项的数量
73	staff[0] : 新对象(序列#7)
72 00 07 Manager	新类、字符串长度、类名 (序列#2)
36 06 AE 13 63 8F 59 B7 02	指纹和标志
00 01	数据的数量

L 00 09	secretary	实例域的类型和名字
74 00 0A	LEmployee;	实例域的类名: String (序列#3)
78		结束标记
72 00 08	Employee	超类-: 新类、字符串长度、类名 (序列#4)
E6 D2 86 7D AE AC 18 1B 02		指纹和标志
00 03		实例域的数量
D 00 06	salary	实例域的类型和名字
L 00 07	hireDay	实例域的类型和名字
74 00 10	Ljava/util/Date;	实例域的类名: String (序列#5)
L 00 04	name	实例域的类型和名字
74 00 12	Ljava/lang/String;	实例域的类名: String (序列#6)
78		结束标记
70		无超类
40 F3 88 00 00 00 00 00 00		salary 域的值: double
73		hireDate 域的值: 新对象 (序列#9)
72 00 0E	java.util.Date	新类、字符串长度、类名 (序列#8)
68 6A 81 01 4B 59 74 19 03		指纹和标志
00 00		无实例变量
78		结束标记
70		无超类
77 08		外部存储、字节的数量
00 00 00 83 E9 39 E0 00		日期
78		结束标记
74 00 0C	Carl Cracker	name 域的值: String (序列#10)
73		secretary 域的值: 新对象 (序列#11)
71 00 7E 00 04		已有的类 (使用序列#4)
40 E8 6A 00 00 00 00 00 00		salary 域的值: double
73		hireDate 域的值: 新对象 (序列#12)
71 00 7E 00 08		已有的类 (使用序列#8)
77 08		外部存储、字节的数量
00 00 00 91 1B 4E B1 80		日期
78		结束标记
74 00 0C	Harry Hacker	name 域的值: String (序列#13)
71 00 7E 00 0B		staff[1]: 已有的对象 (使用序列#11)
73		staff[2]: 新对象 (序列#14)
71 00 7E 00 02		已有的类 (使用序列#2)
40 E3 88 00 00 00 00 00 00		salary 域的值: double

73	hireDay 域的值: 新对象 (序列#15)
71 00 7E 00 08	已有的类 (使用序列#8)
77 08	外部存储、字节的数量
00 00 00 94 6D 3E EC 00 00	日期
78	结束标记
74 00 0B Tony Tester	name 域的值: String (序列#16)
71 00 7E 00 0B	secretary 域的值: 已有的对象 (使用序列#11)

当然，研究这些编码大概与阅读常用的电话号码簿一样枯燥。了解确切的文件格式确实不那么重要（除非你试图通过修改数据来达到不可告人的目的），但是对象流对其所包含的所有对象都有详细描述，并且这些充足的细节可以用来重构对象和对象数组，因此了解它还是大有益处的。

你应该记住：

- 对象流输出中包含所有对象的类型和数据域。
- 每个对象都被赋予一个序列号。
- 相同对象的重复出现将被存储为对这个对象的序列号的引用。

2.3.3 修改默认的序列化机制

某些数据域是不可以序列化的，例如，只对本地方法有意义的存储文件句柄或窗口句柄的整数值，这种信息在稍后重新加载对象或将其传送到其他机器上时都是没有用处的。事实上，这种域的值如果不恰当，还会引起本地方法崩溃。Java 拥有一种很简单的机制来防止这种域被序列化，那就是将它们标记成 `transient` 的。如果这些域属于不可序列化的类，你也需要将它们标记成 `transient` 的。瞬时的域在对象被序列化时总是被跳过的。

序列化机制为单个的类提供了一种方式，去向默认的读写行为添加验证或任何其他想要的行为。可序列化的类可以定义具有下列签名的方法：

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

之后，数据域就再也不会被自动序列化，取而代之的是调用这些方法。

下面是一个典型的示例。在 `java.awt.geom` 包中有大量的类都是不可序列化的，例如 `Point2D.Double`。现在假设你想要序列化一个 `LabeledPoint` 类，它存储了一个 `String` 和一个 `Point2D.Double`。首先，你需要将 `Point2D.Double` 标记成 `transient`，以避免抛出 `NotSerializableException`。

```
public class LabeledPoint implements Serializable
{
    private String label;
    private transient Point2D.Double point;
    ...
}
```

在 `writeObject` 方法中，我们首先通过调用 `defaultWriteObject` 方法写出对象描述符和 `String`

域 `label`, 这是 `ObjectOutputStream` 类中的一个特殊的方法, 它只能在可序列化类的 `writeObject` 方法中被调用。然后, 我们使用标准的 `DataOutput` 调用写出点的坐标。

```
private void writeObject(ObjectOutputStream out)
    throws IOException
{
    out.defaultWriteObject();
    out.writeDouble(point.getX());
    out.writeDouble(point.getY());
}
```

在 `readObject` 方法中, 我们反过来执行上述过程:

```
private void readObject(ObjectInputStream in)
    throws IOException
{
    in.defaultReadObject();
    double x = in.readDouble();
    double y = in.readDouble();
    point = new Point2D.Double(x, y);
}
```

另一个例子是 `java.util.Date` 类, 它提供了自己的 `readObject` 和 `writeObject` 方法, 这些方法将日期写出为从纪元 (UTC 时间 1970 年 1 月 1 日 0 点) 开始的毫秒数。`Date` 类有一个复杂的内部表示, 为了优化查询, 它存储了一个 `Calendar` 对象和一个毫秒计数值。`Calendar` 的状态是冗余的, 因此并不需要保存。

`readObject` 和 `writeObject` 方法只需要保存和加载它们的数据域, 而不需要关心超类数据和任何其他类的信息。

除了让序列化机制来保存和恢复对象数据, 类还可以定义它自己的机制。为了做到这一点, 这个类必须实现 `Externalizable` 接口, 这需要它定义两个方法:

```
public void readExternal(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
    throws IOException;
```

与前面一节描述的 `readObject` 和 `writeObject` 不同, 这些方法对包括超类数据在内的整个对象的存储和恢复负全责。在写出对象时, 序列化机制在输出流中仅仅只是记录该对象所属的类。在读入可外部化的类时, 对象输入流将用无参构造器创建一个对象, 然后调用 `readExternal` 方法。下面展示了如何为 `Employee` 类实现这些方法:

```
public void readExternal(ObjectInput s)
    throws IOException
{
    name = s.readUTF();
    salary = s.readDouble();
    hireDay = LocalDate.ofEpochDay(s.readLong());
}

public void writeExternal(ObjectOutput s)
    throws IOException
```

```

{
    s.writeUTF(name);
    s.writeDouble(salary);
    s.writeLong(hireDay.toEpochDay());
}

```

◆ 警告：readObject 和 writeObject 方法是私有的，并且只能被序列化机制调用。与此不同的是，readExternal 和 writeExternal 方法是公共的。特别是，readExternal 还潜在地允许修改现有对象的状态。

2.3.4 序列化单例和类型安全的枚举

在序列化和反序列化时，如果目标对象是唯一的，那么你必须加倍当心，这通常会在实现单例和类型安全的枚举时发生。

如果你使用 Java 语言的 enum 结构，那么你就不必担心序列化，它能够正常工作。但是，假设你在维护遗留代码，其中包含下面这样的枚举类型：

```

public class Orientation
{
    public static final Orientation HORIZONTAL = new Orientation(1);
    public static final Orientation VERTICAL   = new Orientation(2);

    private int value;

    private Orientation(int v) { value = v; }
}

```

这种风格在枚举被添加到 Java 语言中之前是很普遍的。注意，其构造器是私有的。因此，不可能创建出超出 Orientation.HORIZONTAL 和 Orientation.VERTICAL 之外的对象。特别是，你可以使用 == 操作符来测试对象的等同性：

```
if (orientation == Orientation.HORIZONTAL) . . .
```

当类型安全的枚举实现 Serializable 接口时，你必须牢记存在着一种重要的变化，此时，默认的序列化机制是不适用的。假设我们写出一个 Orientation 类型的值，并再次将其读回：

```

Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.writeObject(original);
out.close();
ObjectInputStream in = . . .;
var saved = (Orientation) in.read();

```

现在，下面的测试

```
if (saved == Orientation.HORIZONTAL) . . .
```

将失败。事实上，saved 的值是 Orientation 类型的一个全新的对象，它与任何预定义的常量都不等同。即使构造器是私有的，序列化机制也可以创建新的对象！

为了解决这个问题，你需要定义另外一种称为 readResolve 的特殊序列化方法。如果定义

了 `readResolve` 方法，在对象被序列化之后就会调用它。它必须返回一个对象，而该对象之后会成为 `readObject` 的返回值。在上面的情况中，`readResolve` 方法将检查 `value` 域并返回恰当的枚举常量：

```
protected Object readResolve() throws ObjectStreamException
{
    if (value == 1) return Orientation.HORIZONTAL;
    if (value == 2) return Orientation.VERTICAL;
    throw new ObjectStreamException(); // this shouldn't happen
}
```

请记住向遗留代码中所有类型安全的枚举以及向所有支持单例设计模式的类中添加 `readResolve` 方法。

2.3.5 版本管理

如果使用序列化来保存对象，就需要考虑在程序演化时会有什么问题。例如，1.1 版本可以读入旧文件吗？仍旧使用 1.0 版本的用户可以读入新版本产生的文件吗？显然，如果对象文件可以处理类的演化问题，那它正是我们想要的。

乍一看，这好像是不可能的。无论类的定义产生了什么样的变化，它的 SHA 指纹也会跟着变化，而我们都应该知道对象输入流将拒绝读入具有不同指纹的对象。但是，类可以表明它对其早期版本保持兼容，要想这样做，就必须首先获得这个类的早期版本的指纹。我们可以使用 JDK 中的单机程序 `serialver` 来获得这个数字，例如，运行下面的命令

```
serialver Employee
```

将会打印出

```
Employee: static final long serialVersionUID = -1814239825517340645L;
```

这个类的所有较新的版本都必须把 `serialVersionUID` 常量定义为与最初版本的指纹相同。

```
class Employee implements Serializable // version 1.1
{
    ...
    public static final long serialVersionUID = -1814239825517340645L;
}
```

如果一个类具有名为 `serialVersionUID` 的静态数据成员，它就不再需要人工计算指纹，而只需直接使用这个值。

一旦这个静态数据成员被置于某个类的内部，那么序列化系统就可以读入这个类的对象的不同版本。

如果这个类只有方法产生了变化，那么在读入新对象数据时是不会有任何问题的。但是，如果数据域产生了变化，那么就可能会有问题。例如，旧文件对象可能比程序中的对象具有更多或更少的数据域，或者数据域的类型可能有所不同。在这些情况下，对象输入流将尽力将流对象转换成这个类当前的版本。

对象输入流会将这个类当前版本的数据域与被序列化的版本中的数据域进行比较，当

然，对象流只会考虑非瞬时和非静态的数据域。如果这两部分数据域之间名字匹配而类型不匹配，那么对象输入流不会尝试将一种类型转换成另一种类型，因为这两个对象不兼容；如果被序列化的对象具有在当前版本中所没有的数据域，那么对象输入流会忽略这些额外的数据；如果当前版本具有在被序列化的对象中所没有的数据域，那么这些新添加的域将被设置成它们的默认值（如果是对象则是 null，如果是数字则为 0，如果是 boolean 值则是 false）。

下面是一个示例：假设我们已经用雇员类的最初版本（1.0）在磁盘上保存了大量的雇员记录，现在我们在 Employee 类中添加了称为 department 的数据域，从而将其演化到了 2.0 版本。图 2-7 展示了将 1.0 版的对象读入到使用 2.0 版对象的程序中的情形，可以看到 department 域被设置成了 null。图 2-8 展示了相反的情况：一个使用 1.0 版对象的程序读入了 2.0 版的对象，可以看到额外的 department 域被忽略。

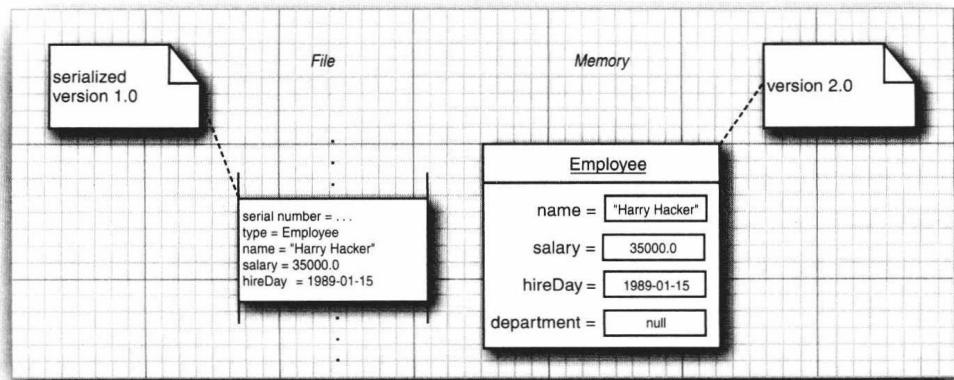


图 2-7 读入具有较少数据域的对象

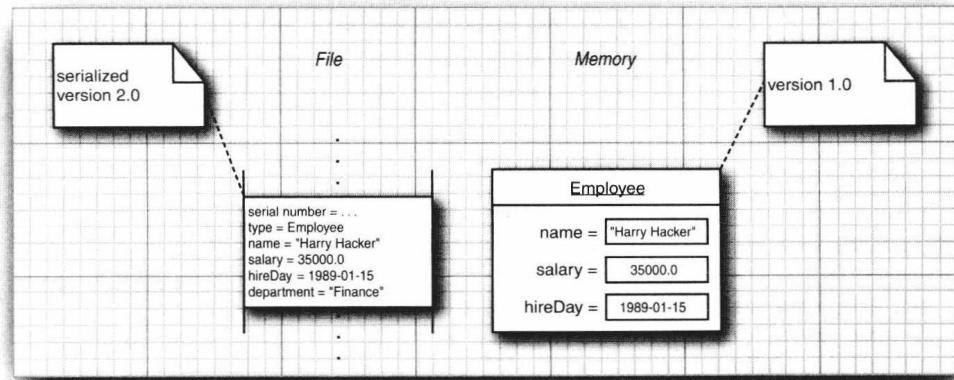


图 2-8 读入具有较多数据域的对象

这种处理是安全的吗？视情况而定。丢掉数据域看起来是无害的，因为接收者仍旧拥有它知道如何处理的所有数据，但是将数据域设置为 null 却有可能并不那么安全。许多类都费尽

心思地在其所有的构造器中将所有的数据域都初始化为非 null 的值，以使得其各个方法都不必去处理 null 数据。因此，这个问题取决于类的设计者是否能够在 `readObject` 方法中实现额外的代码去订正版本不兼容问题，或者是否能够确保所有的方法在处理 null 数据时都足够健壮。

 提示：在将 `serialVersionUID` 域添加到类中之前，需要问问自己为什么要让这个类是可序列化的。如果序列化只是用于短期持久化，例如在应用服务器中的分布式方法调用，那么就不需要关心版本机制和 `serialVersionUID`。如果碰巧要扩展一个可序列化的类，但是又从来没想过要持久化该扩展类的任何实例，那么同样不需要关心它们。如果 IDE 总是报有关此问题的烦人的警告消息，那么可以修改 IDE 偏好，将它们关闭，或者添加 `@SuppressWarnings("serial")` 注解。这样做比添加 `serialVersionUID` 要更安全，因为也许后续我们会忘记修改 `serialVersionUID`。

2.3.6 为克隆使用序列化

序列化机制有一种很有趣的用法：即提供了一种克隆对象的简便途径，只要对应的类是可序列化的即可。其做法很简单：直接将对象序列化到输出流中，然后将其读回。这样产生的新对象是对现有对象的一个深拷贝（deep copy）。在此过程中，我们不必将对象写出到文件中，因为可以用 `ByteArrayOutputStream` 将数据保存到字节数组中。

正如程序清单 2-4 所示，要想得到 `clone` 方法，只需扩展 `Serializable` 类，这样就完事了。

程序清单 2-4 serialClone/SerialCloneTest.java

```

1 package serialClone;
2
3 /**
4  * @version 1.22 2018-05-01
5  * @author Cay Horstmann
6  */
7
8 import java.io.*;
9 import java.time.*;
10
11 public class SerialCloneTest
12 {
13     public static void main(String[] args) throws CloneNotSupportedException
14     {
15         var harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
16         // clone harry
17         var harry2 = (Employee) harry.clone();
18
19         // mutate harry
20         harry.raiseSalary(10);
21
22         // now harry and the clone are different
23         System.out.println(harry);
24         System.out.println(harry2);

```

```
25     }
26 }
27
28 /**
29  * A class whose clone method uses serialization.
30 */
31 class SerialCloneable implements Cloneable, Serializable
32 {
33     public Object clone() throws CloneNotSupportedException
34     {
35         try {
36             // save the object to a byte array
37             var bout = new ByteArrayOutputStream();
38             try (var out = new ObjectOutputStream(bout))
39             {
40                 out.writeObject(this);
41             }
42
43             // read a clone of the object from the byte array
44             try (var bin = new ByteArrayInputStream(bout.toByteArray()))
45             {
46                 var in = new ObjectInputStream(bin);
47                 return in.readObject();
48             }
49         }
50         catch (IOException | ClassNotFoundException e)
51         {
52             var e2 = new CloneNotSupportedException();
53             e2.initCause(e);
54             throw e2;
55         }
56     }
57 }
58
59 /**
60  * The familiar Employee class, redefined to extend the
61  * SerialCloneable class.
62 */
63 class Employee extends SerialCloneable
64 {
65     private String name;
66     private double salary;
67     private LocalDate hireDay;
68
69     public Employee(String n, double s, int year, int month, int day)
70     {
71         name = n;
72         salary = s;
73         hireDay = LocalDate.of(year, month, day);
74     }
75
76     public String getName()
77     {
78         return name;
```

```

79 }
80
81 public double getSalary()
82 {
83     return salary;
84 }
85
86 public LocalDate getHireDay()
87 {
88     return hireDay;
89 }
90
91 /**
92      Raises the salary of this employee.
93      @byPercent the percentage of the raise
94 */
95 public void raiseSalary(double byPercent)
96 {
97     double raise = salary * byPercent / 100;
98     salary += raise;
99 }
100
101 public String toString()
102 {
103     return getClass().getName()
104         + "[name=" + name
105         + ",salary=" + salary
106         + ",hireDay=" + hireDay
107         + "]";
108 }
109 }

```

我们应该当心这个方法，尽管它很灵巧，但是通常会比显式地构建新对象并复制或克隆数据域的克隆方法慢得多。

2.4 操作文件

你已经学习了如何从文件中读写数据，然而文件管理的内涵远远比读写要广。Path 和 Files 类封装了在用户机器上处理文件系统所需的所有功能。例如，Files 类可以用来移除或重命名文件，或者查询文件最后被修改的时间。换句话说，输入 / 输出流类关心的是文件的内容，而我们在此处要讨论的类关心的是文件在磁盘上的存储。

Path 接口和 Files 类是在 Java 7 中新添加进来的，它们用起来比自 JDK 1.0 以来就一直使用的 File 类要方便得多。我们认为这两个类会在 Java 程序员中流行起来，因此在这里做深度讨论。

2.4.1 Path

Path (路径) 表示的是一个目录名序列，其后还可以跟着一个文件名。路径中的第一个部

件可以是根部件，例如 / 或 C:\，而允许访问的根部件取决于文件系统。以根部件开始的路径是绝对路径；否则，就是相对路径。例如，我们要分别创建一个绝对路径和一个相对路径；其中，对于绝对路径，我们假设计算机运行的是类 UNIX 的文件系统：

```
Path absolute = Paths.get("/home", "harry");
Path relative = Paths.get("myprog", "conf", "user.properties");
```

静态的 Paths.get 方法接受一个或多个字符串，并将它们用默认文件系统的路径分隔符（类 UNIX 文件系统是 /，Windows 是 \）连接起来。然后它解析连接起来的结果，如果其表示的不是给定文件系统中的合法路径，那么就抛出 InvalidPathException 异常。这个连接起来的结果就是一个 Path 对象。

get 方法可以获取包含多个部件的单个字符串，例如，可以像下面这样从配置文件中读取路径：

```
String baseDir = props.getProperty("base.dir");
// May be a string such as /opt/myprog or c:\Program Files\myprog
Path basePath = Paths.get(baseDir); // OK that baseDir has separators
```

注释：路径不必对应着某个实际存在的文件，它仅仅是一个抽象的名字序列。在接下来的小节中将会看到，当你想要创建文件时，首先要创建一个路径，然后才调用方法去创建对应的文件。

组合或解析路径是司空见惯的操作，调用 p.resolve(q) 将按照下列规则返回一个路径：

- 如果 q 是绝对路径，则结果就是 q。
- 否则，根据文件系统的规则，将“p 后面跟着 q”作为结果。

例如，假设你的应用系统需要查找相对于给定基目录的工作目录，其中基目录是从配置文件中读取的，就像前一个例子一样。

```
Path workRelative = Paths.get("work");
Path workPath = basePath.resolve(workRelative);
```

resolve 方法有一种快捷方式，它接受一个字符串而不是路径：

```
Path workPath = basePath.resolve("work");
```

还有一个很方便的方法 resolveSibling，它通过解析指定路径的父路径产生其兄弟路径。例如，如果 workPath 是 /opt/myapp/work，那么下面的调用

```
Path tempPath = workPath.resolveSibling("temp");
```

将创建 /opt/myapp/temp。

resolve 的对立面是 relativize，即调用 p.relativize(r) 将产生路径 q，而对 q 进行解析的结果正是 r。例如，以 “/home/harry” 为目标对 “/home/fred/input.txt” 进行相对化操作，会产生 “..../input.txt”，其中，我们假设 .. 表示文件系统中的父目录。

normalize 方法将移除所有冗余的 . 和 .. 部件（或者文件系统认为冗余的所有部件）。例如，规范化 /home/harry/..../fred/./input.txt 将产生 /home/fred/input.txt。

`toAbsolutePath` 方法将产生给定路径的绝对路径，该绝对路径从根部件开始，例如 `/home/fred/input.txt` 或 `c:\Users\fred\input.txt`。

`Path` 类有许多有用的方法用来将路径断开。下面的代码示例展示了其中部分最有用的方法：

```
Path p = Paths.get("/home", "fred", "myprog.properties");
Path parent = p.getParent(); // the path /home/fred
Path file = p.getFileName(); // the path myprog.properties
Path root = p.getRoot(); // the path /
```

正如你已经在卷 I 中看到的，还可以从 `Path` 对象中构建 `Scanner` 对象：

```
var in = new Scanner(Paths.get("/home/fred/input.txt"));
```

注释：偶尔，你可能需要与遗留系统的 API 交互，它们使用的是 `File` 类而不是 `Path` 接口。`Path` 接口有一个 `toFile` 方法，而 `File` 类有一个 `toPath` 方法。

API `java.nio.file.Paths`

- `static Path get(String first, String... more)`

通过连接给定的字符串创建一个路径。

API `java.nio.file.Path`

- `Path resolve(Path other)`
- `Path resolve(String other)`

如果 `other` 是绝对路径，那么就返回 `other`；否则，返回通过连接 `this` 和 `other` 获得的路径。

- `Path resolveSibling(Path other)`
- `Path resolveSibling(String other)`

如果 `other` 是绝对路径，那么就返回 `other`；否则，返回通过连接 `this` 的父路径和 `other` 获得的路径。

- `Path relativize(Path other)`

返回用 `this` 进行解析，相对于 `other` 的相对路径。

- `Path normalize()`

移除诸如 `.` 和 `..` 等冗余的路径元素。

- `Path toAbsolutePath()`

返回与该路径等价的绝对路径。

- `Path getParent()`

返回父路径，或者在该路径没有父路径时，返回 `null`。

- `Path getFileName()`

返回该路径的最后一个部件，或者在该路径没有任何部件时，返回 `null`。

- `Path getRoot()`

返回该路径的根部件，或者在该路径没有任何根部件时，返回 `null`。

- `toFile()`

从该路径中创建一个 `File` 对象。

API `java.io.File 1.0`

- `Path toPath() 7`

从该文件中创建一个 `Path` 对象。

2.4.2 读写文件

`Files` 类可以使得普通文件操作变得快捷。例如，可以用下面的方式很容易地读取文件的所有内容：

```
byte[] bytes = Files.readAllBytes(path);
```

正如在 2.1.6 节中介绍过的，我们可以如下从文本文件中读取内容：

```
var content = Files.readString(path, charset);
```

但是如果希望将文件当作行序列读入，那么可以调用：

```
List<String> lines = Files.readAllLines(path, charset);
```

相反，如果希望写出一个字符串到文件中，可以调用：

```
Files.writeString(path, content, charset);
```

向指定文件追加内容，可以调用：

```
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
```

还可以用下面的语句将一个行的集合写出到文件中：

```
Files.write(path, lines, charset);
```

这些简便方法适用于处理中等长度的文本文件，如果要处理的文件长度比较大，或者是二进制文件，那么还是应该使用所熟知的输入 / 输出流或者读入器 / 写出器：

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

这些便捷方法可以将你从处理 `FileInputStream`、`FileOutputStream`、`BufferedReader` 和 `BufferedWriter` 的繁复操作中解脱出来。

API `java.nio.file.Files 7`

- `static byte[] readAllBytes(Path path)`
- `static String readString(Path path, Charset charset)`
- `static List<String> readAllLines(Path path, Charset charset)`
读入文件的内容。
- `static Path write(Path path, byte[] contents, OpenOption... options)`
- `static Path write(Path path, String contents, Charset charset, OpenOption... options)`

- static Path write(Path path, Iterable<? extends CharSequence> contents, OpenOption options)
将给定内容写出到文件中，并返回 path。
- static InputStream newInputStream(Path path, OpenOption... options)
- static OutputStream newOutputStream(Path path, OpenOption... options)
- static BufferedReader newBufferedReader(Path path, Charset charset)
- static BufferedWriter newBufferedWriter(Path path, Charset charset, OpenOption... options)
打开一个文件，用于读入或写出。

2.4.3 创建文件和目录

创建新目录可以调用

```
Files.createDirectory(path);
```

其中，路径中除最后一个部件外，其他部分都必须是已存在的。要创建路径中的中间目录，应该使用

```
Files.createDirectories(path);
```

可以使用下面的语句创建一个空文件：

```
Files.createFile(path);
```

如果文件已经存在了，那么这个调用就会抛出异常。检查文件是否存在和创建文件是原子性的，如果文件不存在，该文件就会被创建，并且其他程序在此过程中是无法执行文件创建操作的。

有些便捷方法可以用来在给定位置或者系统指定位置创建临时文件或临时目录：

```
Path newPath = Files.createTempFile(dir, prefix, suffix);
Path newPath = Files.createTempFile(prefix, suffix);
Path newPath = Files.createTempDirectory(dir, prefix);
Path newPath = Files.createTempDirectory(prefix);
```

其中，dir 是一个 Path 对象，prefix 和 suffix 是可以为 null 的字符串。例如，调用 Files.createTempFile(null, ".txt") 可能会返回一个像 /tmp/1234405522364837194.txt 这样的路径。

在创建文件或目录时，可以指定属性，例如文件的拥有者和权限。但是，指定属性的细节取决于文件系统，本书在此不做讨论。

API `java.nio.file.Files`

- static Path createFile(Path path, FileAttribute<?>... attrs)
- static Path createDirectory(Path path, FileAttribute<?>... attrs)
- static Path createDirectories(Path path, FileAttribute<?>... attrs)
创建一个文件或目录，createDirectories 方法还会创建路径中所有的中间目录。
- static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)
- static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)

- static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)
- static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)

在适合临时文件的位置，或者在给定的父目录中，创建一个临时文件或目录。返回所创建的文件或目录的路径。

2.4.4 复制、移动和删除文件

将文件从一个位置复制到另一个位置可以直接调用

```
Files.copy(fromPath, toPath);
```

移动文件（即复制并删除原文件）可以调用

```
Files.move(fromPath, toPath);
```

如果目标路径已经存在，那么复制或移动将失败。如果想要覆盖已有的目标路径，可以使用 REPLACE_EXISTING 选项。如果想要复制所有的文件属性，可以使用 COPY_ATTRIBUTES 选项。也可以像下面这样同时选择这两个选项：

```
Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING,  
          StandardCopyOption.COPY_ATTRIBUTES);
```

你可以将移动操作定义为原子性的，这样就可以保证要么移动操作成功完成，要么源文件继续保持在原来位置。具体可以使用 ATOMIC_MOVE 选项来实现：

```
Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);
```

你还可以将一个输入流复制到 Path 中，这表示想要将该输入流存储到硬盘上。类似地，你可以将一个 Path 复制到输出流中。可以使用下面的调用：

```
Files.copy(inputStream, toPath);  
Files.copy(fromPath, outputStream);
```

至于其他对 copy 的调用，可以根据需要提供相应的复制选项。

最后，删除文件可以调用：

```
Files.delete(path);
```

如果要删除的文件不存在，这个方法就会抛出异常。因此，可转而使用下面的方法：

```
boolean deleted = Files.deleteIfExists(path);
```

该删除方法还可以用来移除空目录。

请查阅表 2-3 以了解对文件操作而言可用的选项。

表 2-3 用于文件操作的标准选项

选项	描述
StandardOpenOption	与 newBufferedWriter、newInputStream、newOutputStream、write 一起使用
READ	用于读取而打开
WRITE	用于写入而打开
APPEND	如果用于写入而打开，那么在文件末尾追加

(续)

选项	描述
TRUNCATE_EXISTING	如果用于写入而打开，那么移除已有内容
CREATE_NEW	创建新文件并且在文件已存在的情况下会创建失败
CREATE	自动在文件不存在的情况下创建新文件
DELETE_ON_CLOSE	当文件被关闭时，尽“可能”地删除该文件
SPARSE	给文件系统一个提示，表示该文件是稀疏的
DSYNC 或 SYNC	要求对文件数据 数据和元数据的每次更新都必须同步地写入到存储设备中
StandardCopyOption	与 copy 和 move 一起使用
ATOMIC_MOVE	原子性地移动文件
COPY_ATTRIBUTES	复制文件的属性
REPLACE_EXISTING	如果目标已存在，则替换它
LinkOption	与上面所有方法以及 exists、isDirectory、isRegularFile 等一起使用
NOFOLLOW_LINKS	不要跟踪符号链接
FileVisitOption	与 find、walk、walkFileTree 一起使用
FOLLOW_LINKS	跟踪符号链接

API `java.nio.file.Files` 7

- static Path copy(Path from, Path to, CopyOption... options)
- static Path move(Path from, Path to, CopyOption... options)
将 from 复制或移动到给定位置，并返回 to。
- static long copy(InputStream from, Path to, CopyOption... options)
- static long copy(Path from, OutputStream to, CopyOption... options)
从输入流复制到文件中，或者从文件复制到输出流中，返回复制的字节数。
- static void delete(Path path)
- static boolean deleteIfExists(Path path)
删除给定文件或空目录。第一个方法在文件或目录不存在情况下抛出异常，而第二个方法在这种情况下会返回 false。

2.4.5 获取文件信息

下面的静态方法都将返回一个 boolean 值，表示检查路径的某个属性的结果：

- exists
- isHidden
- isReadable, isWritable, isExecutable
- isRegularFile, isDirectory, isSymbolicLink

size 方法将返回文件的字节数：

```
long fileSize = Files.size(path);
```

`getOwner` 方法将文件的拥有者作为 `java.nio.file.attribute.UserPrincipal` 的一个实例返回。

所有的文件系统都会报告一个基本属性集，它们被封装在 `BasicFileAttributes` 接口中，这些属性与上述信息有部分重叠。基本文件属性包括：

- 创建文件、最后一次访问以及最后一次修改文件的时间，这些时间都表示成 `java.nio.file.attribute.FileTime`。
- 文件是常规文件、目录还是符号链接，抑或这三者都不是。
- 文件尺寸。
- 文件主键，这是某种类的对象，具体所属类与文件系统相关，有可能是文件的唯一标识符，也可能不是。

要获取这些属性，可以调用

```
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);
```

如果你了解到用户的文件系统兼容 POSIX，那么你可以获取一个 `PosixFileAttributes` 实例：

```
PosixFileAttributes attributes = Files.readAttributes(path, PosixFileAttributes.class);
```

然后从中找到组拥有者，以及文件的拥有者、组和访问权限。我们不会详细讨论其细节，因为这种信息中很多内容在操作系统之间并不具备可移植性。

API `java.nio.file.Files` 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`

检查由路径指定的文件的给定属性。

- `static long size(Path path)`
获取文件按字节数度量的尺寸。
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`
读取类型为 A 的文件属性。

API `java.nio.file.attribute.BasicFileAttributes` 7

- `FileTime creationTime()`
- `FileTime lastAccessTime()`
- `FileTime lastModifiedTime()`
- `boolean isRegularFile()`

- boolean isDirectory()
- boolean isSymbolicLink()
- long size()
- Object fileKey()

获取所请求的属性。

2.4.6 访问目录中的项

静态的 `Files.list` 方法会返回一个可以读取目录中各个项的 `Stream<Path>` 对象。目录是被惰性读取的，这使得处理具有大量项的目录可以变得更高效。

因为读取目录涉及需要关闭的系统资源，所以应该使用 `try` 块：

```
try (Stream<Path> entries = Files.list(pathToDirectory))
{
    ...
}
```

`list` 方法不会进入子目录。为了处理目录中的所有子目录，需要使用 `File.walk` 方法。

```
try (Stream<Path> entries = Files.walk(pathToRoot))
{
    // Contains all descendants, visited in depth-first order
}
```

下面是解压后的 `src.zip` 树的遍历样例：

```
java
java/nio
java/nio/DirectCharBufferU.java
java/nio/ByteBufferAsShortBufferRL.java
java/nio/MappedByteBuffer.java
...
java/nio/ByteBufferAsDoubleBufferB.java
java/nio/charset
java/nio/charset/CoderMalfunctionError.java
java/nio/charset/CharsetDecoder.java
java/nio/charset/UnsupportedCharsetException.java
java/nio/charset/spi
java/nio/charset/spi/CharsetProvider.java
java/nio/charset/StandardCharsets.java
java/nio/charset/Charset.java
...
java/nio/charset/CoderResult.java
java/nio/HeapFloatBufferR.java
...
```

正如你所见，无论何时，只要遍历的项是目录，那么在继续访问它的兄弟项之前，会先进入它。

可以通过调用 `File.walk(pathToRoot, depth)` 来限制想要访问的树的深度。两种 `walk` 方法都具有 `FileVisitOption...` 的可变长参数，但是你只能提供一种选项——`FOLLOW_LINKS`，即跟踪符号链接。

注释：如果要过滤 walk 返回的路径，并且过滤标准涉及与目录存储相关的文件属性，例如尺寸、创建时间和类型（文件、目录、符号链接），那么应该使用 find 方法来替代 walk 方法。可以用某个谓词函数来调用这个方法，该函数接受一个路径和一个 BasicFileAttributes 对象。这样做唯一的优势就是效率高。因为路径总是会被读入，所以这些属性很容易获取。

这段代码使用了 Files.walk 方法来将一个目录复制到另一个目录：

```
Files.walk(source).forEach(p ->
{
    try
    {
        Path q = target.resolve(source.relativize(p));
        if (Files.isDirectory(p))
            Files.createDirectory(q);
        else
            Files.copy(p, q);
    }
    catch (IOException ex)
    {
        throw new UncheckedIOException(ex);
    }
});
});
```

遗憾的是，你无法很容易地使用 Files.walk 方法来删除目录树，因为你必须在删除父目录之前先删除子目录。下一节将展示如何克服此问题。

2.4.7 使用目录流

正如在前一节中所看到的，Files.walk 方法会产生一个可以遍历目录中所有子孙的 Stream<Path> 对象。有时，需要对遍历过程进行更加细粒度的控制。在这种情况下，应该使用 Files.newDirectoryStream 对象，它会产生一个 DirectoryStream。注意，它不是 java.util.stream.Stream 的子接口，而是专门用于目录遍历的接口。它是 Iterable 的子接口，因此可以在增强的 for 循环中使用目录流。下面是其使用模式：

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir))
{
    for (Path entry : entries)
        Process entries
}
```

带资源的 try 语句块用来确保目录流可以被正确关闭。访问目录中的项并没有具体的顺序。

可以用 glob 模式来过滤文件：

```
try (DirectoryStream<Path> entries = Files.newDirectoryStream(dir, "*.java"))
```

表 2-4 展示了所有的 glob 模式。

表 2-4 glob 模式

模式	描述	示例
*	匹配路径组成部分中 0 个或多个字符	*.java 匹配当前目录中的所有 Java 文件
**	匹配跨目录边界的 0 个或多个字符	**.java 匹配在所有子目录中的 Java 文件
?	匹配一个字符	????.java 匹配所有四个字符的 Java 文件 (不包括扩展名)
[...]	匹配一个字符集合, 可以使用连线符 [0-9] 和取反符 [!0-9]	Test[0-9A-F].java 匹配 Testx.java, 其中 x 是一个十六进制数字
{...}	匹配由逗号隔开的多个可选项之一	*.{java,class} 匹配所有的 Java 文件和类文件
\	转义上述任意模式中的字符以及 \ 字符	*** 匹配所有文件名中包含 * 的文件

！ 警告：如果使用 Windows 的 glob 语法，则必须对反斜杠转义两次：一次为 glob 语法转义，一次为 Java 字符串转义。例如 `Files.newDirectoryStream(dir, "C:\\\\\\")`。

如果想要访问某个目录的所有子孙成员，可以转而调用 `walkFileTree` 方法，并向其传递一个 `FileVisitor` 类型的对象，这个对象会得到下列通知：

- 在遇到一个文件或目录时：`FileVisitResult visitFile(T path, BasicFileAttributes attrs)`
- 在一个目录被处理前：`FileVisitResult preVisitDirectory(T dir, IOException ex)`
- 在一个目录被处理后：`FileVisitResult postVisitDirectory(T dir, IOException ex)`
- 在试图访问文件或目录时发生错误，例如没有权限打开目录：`FileVisitResult visitFileFailed(path, IOException)`

对于上述每种情况，都可以指定是否希望执行下面的操作：

- 继续访问下一个文件：`FileVisitResult.CONTINUE`
- 继续访问，但是不再访问这个目录下的任何项了：`FileVisitResult.SKIP_SUBTREE`
- 继续访问，但是不再访问这个文件的兄弟文件（和该文件在同一个目录下的文件）了：`FileVisitResult.SKIP_SIBLINGS`
- 终止访问：`FileVisitResult.TERMINATE`

当有任何方法抛出异常时，就会终止访问，而这个异常会从 `walkFileTree` 方法中抛出。

！ 注释：`FileVisitor` 接口是泛化类型，但是你不太可能会使用除 `FileVisitor<Path>` 之外的东西。`walkFileTree` 方法可以接受 `FileVisitor<? Super Path>` 类型的参数，但是 `Path` 并没有多少超类型。

便捷类 `SimpleFileVisitor` 实现了 `FileVisitor` 接口，但是其除 `visitFileFailed` 方法之外的所有方法并不做任何处理而是直接继续访问，而 `visitFileFailed` 方法会抛出由失败导致的异常，并进而终止访问。

例如，下面的代码展示了如何打印出给定目录下的所有子目录：

```
Files.walkFileTree(Paths.get("/"), new SimpleFileVisitor<Path>()
{
    public FileVisitResult preVisitDirectory(Path path, BasicFileAttributes attrs)
        throws IOException
```

```

{
    System.out.println(path);
    return FileVisitResult.CONTINUE;
}

public FileVisitResult postVisitDirectory(Path dir, IOException exc)
{
    return FileVisitResult.CONTINUE;
}

public FileVisitResult visitFileFailed(Path path, IOException exc)
throws IOException
{
    return FileVisitResult.SKIP_SUBTREE;
}
});

```

值得注意的是，我们需要覆盖 `postVisitDirectory` 方法和 `visitFileFailed` 方法，否则，访问会在遇到不允许打开的目录或不允许访问的文件时立即失败。

还应该注意的是，路径的众多属性是作为 `preVisitDirectory` 和 `visitFile` 方法的参数传递的。访问者不得不通过操作系统调用来获得这些属性，因为它需要区分文件和目录。因此，你就不再需要再次执行系统调用了。

如果你需要在进入或离开一个目录时执行某些操作，那么 `FileVisitor` 接口的其他方法就显得非常有用了。例如，在删除目录树时，需要在移除当前目录的所有文件之后，才能移除该目录。下面是删除目录树的完整代码：

```

// Delete the directory tree starting at root
Files.walkFileTree(root, new SimpleFileVisitor<Path>()
{
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException
    {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }

    public FileVisitResult postVisitDirectory(Path dir, IOException e) throws IOException
    {
        if (e != null) throw e;
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
    }
});

```

API java.nio.file.Files 7

- `static DirectoryStream<Path> newDirectoryStream(Path path)`
- `static DirectoryStream<Path> newDirectoryStream(Path path, String glob)`
获取给定目录中可以遍历所有文件和目录的迭代器。第二个方法只接受那些与给定的 `glob` 模式匹配的项。
- `static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`

遍历给定路径的所有子孙，并将访问器应用于这些子孙之上。

API `java.nio.file.SimpleFileVisitor<T>`

- `static FileVisitResult visitFile(T path, BasicFileAttributes attrs)`

在访问文件或目录时被调用，返回 `CONTINUE`、`SKIP_SUBTREE`、`SKIP_SIBLINGS` 和 `TERMINATE` 之一，默认实现是不做任何操作而继续访问。

- `static FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)`

- `static FileVisitResult postVisitDirectory(T dir, BasicFileAttributes attrs)`

在访问目录之前和之后被调用，默认实现是不做任何操作而继续访问。

- `static FileVisitResult visitFileFailed(T path, IOException exc)`

如果在试图获取给定文件的信息时抛出异常，则该方法被调用。默认实现是重新抛出异常，这会导致访问操作以这个异常而终止。如果你想自己访问，可以覆盖这个方法。

2.4.8 ZIP 文件系统

`Paths` 类会在默认文件系统中查找路径，即在用户本地磁盘中的文件。你也可以有别的文件系统，其中最有用的之一是 ZIP 文件系统。如果 `zipname` 是某个 ZIP 文件的名字，那么下面的调用

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
```

将建立一个文件系统，它包含 ZIP 文档中的所有文件。如果知道文件名，那么从 ZIP 文档中复制出这个文件就会变得很容易：

```
Files.copy(fs.getPath(sourceName), targetPath);
```

其中的 `fs.getPath` 对于任意文件系统来说都与 `Paths.get` 类似。

要列出 ZIP 文档中的所有文件，可以遍历文件树：

```
FileSystem fs = FileSystems.newFileSystem(Paths.get(zipname), null);
Files.walkFileTree(fs.getPath("/"), new SimpleFileVisitor<Path>()
{
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException
    {
        System.out.println(file);
        return FileVisitResult.CONTINUE;
    }
});
```

这比 2.2.3 节中描述的 API 要好用，后者使用的是多个专门处理 ZIP 文档的新类。

API `java.nio.file.FileSystems`

- `static FileSystem newFileSystem(Path path, ClassLoader loader)`

对所安装的文件系统提供者进行迭代，并且如果 `loader` 不为 `null`，那么就还会迭代给定的类加载器能够加载的文件系统，返回由第一个可以接受给定路径的文件系统提供

者创建的文件系统。默认情况下，对于 ZIP 文件系统是有一个提供者的，它接受名字以 .zip 或 .jar 结尾的文件。

API java.nio.file.FileSystem 7

- static Path getPath(String first, String... more)

将给定的字符串连接起来创建一个路径。

2.5 内存映射文件

大多数操作系统都可以利用虚拟内存实现来将一个文件或者文件的一部分“映射”到内存中。然后，这个文件就可以被当作内存数组一样地访问，这比传统的文件操作要快得多。

2.5.1 内存映射文件的性能

在本节的末尾，你可以看到一个计算传统的文件输入和内存映射文件的 CRC32 校验和的程序。在同一台机器上，我们对 JDK 的 jre/lib 目录中 37MB 的 rt.jar 文件用不同的方式来计算校验和，记录下来的时间数据如表 2-5 所示。

表 2-5 文件操作的处理时间数据

方法	时间
普通输入流	110 秒
带缓冲的输入流	9.9 秒
随机访问文件	162 秒
内存映射文件	7.2 秒

正如你所见，在这台特定的机器上，内存映射比使用带缓冲的顺序输入要稍微快一点，但是比使用 RandomAccessFile 快很多。

当然，精确的值因机器不同会产生很大的差异，但是很明显，与随机访问相比，性能提高总是很显著的。另一方面，对于中等尺寸文件的顺序读入则没有必要使用内存映射。

java.nio 包使内存映射变得十分简单，下面就是我们需要做的。

首先，从文件中获得一个通道（channel），通道是用于磁盘文件的一种抽象，它使我们可以访问诸如内存映射、文件加锁机制以及文件间快速数据传递等操作系统特性。

```
FileChannel channel = FileChannel.open(path, options);
```

然后，通过调用 FileChannel 类的 map 方法从这个通道中获得一个 ByteBuffer。你可以指定想要映射的文件区域与映射模式，支持的模式有三种：

- FileChannel.MapMode.READ_ONLY：所产生的缓冲区是只读的，任何对该缓冲区写入的尝试都会导致 ReadOnlyBufferException 异常。

- `FileChannel.MapMode.READ_WRITE`：所产生的缓冲区是可写的，任何修改都会在某个时刻写回到文件中。注意，其他映射同一个文件的程序可能不能立即看到这些修改，多个程序同时进行文件映射的确切行为是依赖于操作系统的。
- `FileChannel.MapMode.PRIVATE`：所产生的缓冲区是可写的，但是任何修改对这个缓冲区来说都是私有的，不会传播到文件中。

一旦有了缓冲区，就可以使用 `ByteBuffer` 类和 `Buffer` 超类的方法读写数据了。

缓冲区支持顺序和随机数据访问，它有一个可以通过 `get` 和 `put` 操作来移动的位置。例如，可以像下面这样顺序遍历缓冲区中的所有字节：

```
while (buffer.hasRemaining())
{
    byte b = buffer.get();
    ...
}
```

或者，像下面这样进行随机访问：

```
for (int i = 0; i < buffer.limit(); i++)
{
    byte b = buffer.get(i);
    ...
}
```

你可以用下面的方法来读写字节数组：

```
get(byte[] bytes)
get(byte[], int offset, int length)
```

最后，还有下面的方法：

<code>getInt</code>	<code>getChar</code>
<code>getLong</code>	<code>getFloat</code>
<code>getShort</code>	<code>getDouble</code>

用来读入在文件中存储为二进制值的基本类型值。正如我们提到的，Java 对二进制数据使用高位在前的排序机制，但是，如果需要以低位在前的排序方式处理包含二进制数字的文件，那么只需调用

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

要查询缓冲区内当前的字节顺序，可以调用：

```
ByteOrder b = buffer.order();
```

◆ **警告：**这一对方法没有使用 `set/get` 命名惯例。

要向缓冲区写数字，可以使用下列的方法：

<code>putInt</code>	<code>putChar</code>
<code>putLong</code>	<code>putFloat</code>
<code>putShort</code>	<code>putDouble</code>

在恰当的时机，以及当通道关闭时，会将这些修改写回到文件中。

程序清单 2-5 用于计算文件的 32 位的循环冗余校验和 (CRC32)，这个数值就是经常用来判断一个文件是否已损坏的校验和，因为文件损坏极有可能导致校验和改变。`java.util.zip` 包中包含一个 CRC32 类，可以使用下面的循环来计算一个字节序列的校验和：

```
var crc = new CRC32();
while (more bytes)
    crc.update(next byte);
long checksum = crc.getValue();
```

CRC 计算的细节并不重要，我们只是将它作为一个有用的文件操作的实例来使用。(在实践中，每次会以更大的块而不是一个字节为单位来读取和更新数据，而它们的速度差异并不明显。)

应该像下面这样运行程序：

```
java memoryMap.MemoryMapTest filename
```

程序清单 2-5 memoryMap/MemoryMapTest.java

```
1 package memoryMap;
2
3 import java.io.*;
4 import java.nio.*;
5 import java.nio.channels.*;
6 import java.nio.file.*;
7 import java.util.zip.*;
8
9 /**
10  * This program computes the CRC checksum of a file in four ways. <br>
11  * Usage: java memoryMap.MemoryMapTest filename
12  * @version 1.02 2018-05-01
13  * @author Cay Horstmann
14 */
15 public class MemoryMapTest
16 {
17     public static long checksumInputStream(Path filename) throws IOException
18     {
19         try (InputStream in = Files.newInputStream(filename))
20         {
21             var crc = new CRC32();
22
23             int c;
24             while ((c = in.read()) != -1)
25                 crc.update(c);
26             return crc.getValue();
27         }
28     }
29
30     public static long checksumBufferedInputStream(Path filename) throws IOException
31     {
32         try (var in = new BufferedInputStream(Files.newInputStream(filename)))
33         {
34             var crc = new CRC32();
```

```
36     int c;
37     while ((c = in.read()) != -1)
38         crc.update(c);
39     return crc.getValue();
40 }
41 }
42
43 public static long checksumRandomAccessFile(Path filename) throws IOException
44 {
45     try (var file = new RandomAccessFile(filename.toFile(), "r"))
46     {
47         long length = file.length();
48         var crc = new CRC32();
49
50         for (long p = 0; p < length; p++)
51         {
52             file.seek(p);
53             int c = file.readByte();
54             crc.update(c);
55         }
56         return crc.getValue();
57     }
58 }
59
60 public static long checksumMappedFile(Path filename) throws IOException
61 {
62     try (FileChannel channel = FileChannel.open(filename))
63     {
64         var crc = new CRC32();
65         int length = (int) channel.size();
66         MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
67
68         for (int p = 0; p < length; p++)
69         {
70             int c = buffer.get(p);
71             crc.update(c);
72         }
73         return crc.getValue();
74     }
75 }
76
77 public static void main(String[] args) throws IOException
78 {
79     System.out.println("Input Stream:");
80     long start = System.currentTimeMillis();
81     Path filename = Paths.get(args[0]);
82     long crcValue = checksumInputStream(filename);
83     long end = System.currentTimeMillis();
84     System.out.println(Long.toHexString(crcValue));
85     System.out.println((end - start) + " milliseconds");
86
87     System.out.println("Buffered Input Stream:");
88     start = System.currentTimeMillis();
89     crcValue = checksumBufferedInputStream(filename);
```

```

90     end = System.currentTimeMillis();
91     System.out.println(Long.toHexString(crcValue));
92     System.out.println((end - start) + " milliseconds");
93
94     System.out.println("Random Access File:");
95     start = System.currentTimeMillis();
96     crcValue = checksumRandomAccessFile(filename);
97     end = System.currentTimeMillis();
98     System.out.println(Long.toHexString(crcValue));
99     System.out.println((end - start) + " milliseconds");
100
101    System.out.println("Mapped File:");
102    start = System.currentTimeMillis();
103    crcValue = checksumMappedFile(filename);
104    end = System.currentTimeMillis();
105    System.out.println(Long.toHexString(crcValue));
106    System.out.println((end - start) + " milliseconds");
107  }
108 }
```

API **java.io.FileInputStream 1.0**

- `FileChannel getChannel() 1.4`

返回用于访问这个输入流的通道。

API **java.io.FileOutputStream 1.0**

- `FileChannel getChannel() 1.4`

返回用于访问这个输出流的通道。

API **java.io.RandomAccessFile 1.0**

- `FileChannel getChannel() 1.4`

返回用于访问这个文件的通道。

API **java.nio.channels.FileChannel 1.4**

- `static FileChannel open(Path path, OpenOption... options) 7`

打开指定路径的文件通道，默认情况下，通道打开时用于读入。参数 `options` 是 `StandardOpenOption` 枚举中的 `WRITE`、`APPEND`、`TRUNCATE_EXISTING`、`CREATE` 值。

- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`

将文件的一个区域映射到内存中。参数 `mode` 是 `FileChannel.MapMode` 类中的常量 `READ_ONLY`、`READ_WRITE` 或 `PRIVATE` 之一。

API **java.nio.Buffer 1.4**

- `boolean hasRemaining()`

如果当前的缓冲区位置没有到达这个缓冲区的界限位置，则返回 `true`。

- `int limit()`

返回这个缓冲区的界限位置，即没有任何值可用的第一个位置。

API `java.nio.ByteBuffer 1.4`

- `byte get()`
从当前位置获得一个字节，并将当前位置移动到下一个字节。
- `byte get(int index)`
从指定索引处获得一个字节。
- `ByteBuffer put(byte b)`
向当前位置推入一个字节，并将当前位置移动到下一个字节。返回对这个缓冲区的引用。
- `ByteBuffer put(int index, byte b)`
向指定索引处推入一个字节。返回对这个缓冲区的引用。
- `ByteBuffer get(byte[] destination)`
- `ByteBuffer get(byte[] destination, int offset, int length)`
用缓冲区中的字节来填充字节数组，或者字节数组的某个区域，并将当前位置向前移动读入的字节数个位置。如果缓冲区不够大，那么就不会读入任何字节，并抛出 `BufferUnderflowException`。返回对这个缓冲区的引用。
- `ByteBuffer put(byte[] source)`
- `ByteBuffer put(byte[] source, int offset, int length)`
将字节数组中的所有字节或者给定区域的字节都推入缓冲区中，并将当前位置向前移动写出的字节数个位置。如果缓冲区不够大，那么就不会读入任何字节，并抛出 `BufferUnderflowException`。返回对这个缓冲区的引用。
- `Xxx getXxx()`
- `Xxx getXxx(int index)`
- `ByteBuffer putXxx(Xxx value)`
- `ByteBuffer putXxx(int index, Xxx value)`
获得或放置一个二进制数。`Xxx` 是 `Int`、`Long`、`Short`、`Char`、`Float` 或 `Double` 中的一个。
- `ByteBuffer order(ByteOrder order)`
- `ByteOrder order()`
设置或获得字节顺序，`order` 的值是 `ByteOrder` 类的常量 `BIG_ENDIAN` 或 `LITTLE_ENDIAN` 中的一个。
- `static ByteBuffer allocate(int capacity)`
构建具有给定容量的缓冲区。
- `static ByteBuffer wrap(byte[] values)`
构建具有指定容量的缓冲区，该缓冲区是对给定数组的包装。
- `CharBuffer asCharBuffer()`

构建字符缓冲区，它是对这个缓冲区的包装。对该字符缓冲区的变更将在这个缓冲区中反映出来，但是该字符缓冲区有自己的位置、界限和标记。

API `java.nio.CharBuffer 1.4`

- `char get()`
- `CharBuffer get(char[] destination)`
- `CharBuffer get(char[] destination, int offset, int length)`

从这个缓冲区的当前位置开始，获取一个 `char` 值，或者一个范围内的所有 `char` 值，然后将位置向前移动以越过所有读入的字符。最后两个方法将返回 `this`。

- `CharBuffer put(char c)`
- `CharBuffer put(char[] source)`
- `CharBuffer put(char[] source, int offset, int length)`
- `CharBuffer put(String source)`
- `CharBuffer put(CharBuffer source)`

从这个缓冲区的当前位置开始，放置一个 `char` 值，或者一个范围内的所有 `char` 值，然后将位置向前移动越过所有被写出的字符。当放置的值是从 `CharBuffer` 读入时，将读入所有剩余字符。所有方法将返回 `this`。

2.5.2 缓冲区数据结构

在使用内存映射时，我们创建了单一的缓冲区横跨整个文件或我们感兴趣的文件区域。我们还可以使用更多的缓冲区来读写大小适度的信息块。

本节将简要地介绍 `Buffer` 对象上的基本操作。缓冲区是由具有相同类型的数值构成的数据组，`Buffer` 类是一个抽象类，它有众多的具体子类，包括 `ByteBuffer`、`CharBuffer`、`DoubleBuffer`、`IntBuffer`、`LongBuffer` 和 `ShortBuffer`。

注释：`StringBuffer` 类与这些缓冲区没有关系。

在实践中，最常用的将是 `ByteBuffer` 和 `CharBuffer`。如图 2-9 所示，每个缓冲区都具有：

- 一个容量，它永远不能改变。
- 一个读写位置，下一个值将在此进行读写。
- 一个界限，超过它进行读写是没有意义的。
- 一个可选的标记，用于重复一个读入或写出操作。

这些值满足下面的条件：

$$0 \leqslant \text{标记} \leqslant \text{读写位置} \leqslant \text{界限} \leqslant \text{容量}$$

使用缓冲区的主要目的是执行“写，然后读入”循环。假设我们有一个缓冲区，在一开始，它的位置为 0，界限等于容量。我们不断地调用 `put` 将值添加到这个缓冲区中，当我们耗尽所有的数据或者写出的数据量达到容量大小时，就该切换到读入操作了。

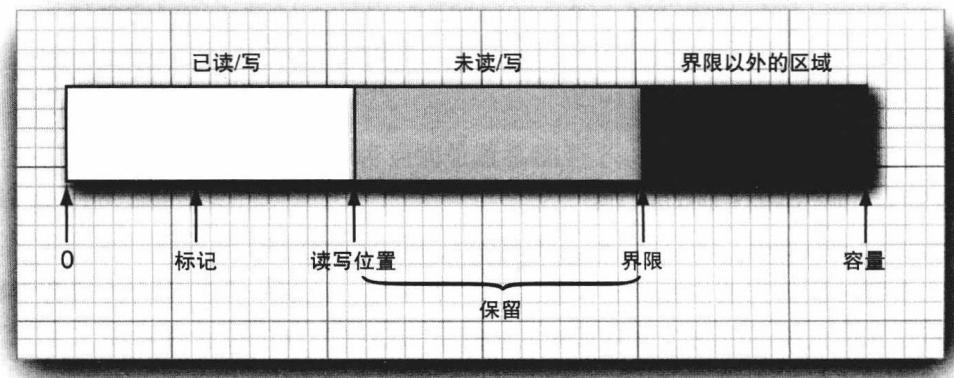


图 2-9 一个缓冲区

这时调用 `flip` 方法将界限设置到当前位置，并把位置复位到 0。现在在 `remaining` 方法返回正数时（它返回的值是界限 - 位置），不断地调用 `get`。在我们将缓冲区中所有的值都读入之后，调用 `clear` 使缓冲区为下一次写循环做好准备。`clear` 方法将位置复位到 0，并将界限复位到容量。

如果你想重读缓冲区，可以使用 `rewind` 或 `mark/reset` 方法，详细内容请查看 API 注释。

要获取缓冲区，可以调用诸如 `ByteBuffer.allocate` 或 `ByteBuffer.wrap` 这样的静态方法。

然后，可以用来自某个通道的数据填充缓冲区，或者将缓冲区的内容写出到通道中。例如：

```
ByteBuffer buffer = ByteBuffer.allocate(RECORD_SIZE);
channel.read(buffer);
channel.position(newpos);
buffer.flip();
channel.write(buffer);
```

这是一种非常有用的方法，可以替代随机访问文件。

API `java.nio.Buffer 1.4`

- `Buffer clear()`

通过将位置复位到 0，并将界限设置到容量，使这个缓冲区为写出做好准备。返回 `this`。

- `Buffer flip()`

通过将界限设置到位置，并将位置复位到 0，使这个缓冲区为读入做好准备。返回 `this`。

- `Buffer rewind()`

通过将读写位置复位到 0，并保持界限不变，使这个缓冲区为重新读入相同的值做好准备。返回 `this`。

- `Buffer mark()`

将这个缓冲区的标记设置到读写位置，返回 this。

- `Buffer reset()`

将这个缓冲区的位置设置到标记，从而允许被标记的部分再次被读入或写出，返回 this。

- `int remaining()`

返回剩余可读入或可写出的值的数量，即界限与位置之间的差异。

- `int position()`

- `void position(int newValue)`

返回这个缓冲区的位置。

- `int capacity()`

返回这个缓冲区的容量。

2.6 文件加锁机制

考虑一下多个同时执行的程序需要修改同一个文件的情形，很明显，这些程序需要以某种方式进行通信，不然这个文件很容易被损坏。文件锁可以解决这个问题，它可以控制对文件或文件中某个范围的字节的访问。

假设你的应用程序将用户的偏好存储在一个配置文件中，当用户调用这个应用的两个实例时，这两个实例就有可能会同时希望写配置文件。在这种情况下，第一个实例应该锁定文件，当第二个实例发现文件被锁定时，它必须决策是等待直至文件解锁，还是直接跳过这个写操作过程。

要锁定一个文件，可以调用 `FileChannel` 类的 `lock` 或 `tryLock` 方法：

```
FileChannel = FileChannel.open(path);
FileLock lock = channel.lock();
```

或

```
FileLock lock = channel.tryLock();
```

第一个调用会阻塞直至可获得锁，而第二个调用将立即返回，要么返回锁，要么在锁不可获得的情况下返回 `null`。这个文件将保持锁定状态，直至通道关闭，或者在锁上调用了 `release` 方法。

你还可以通过下面的调用锁定文件的一部分：

```
FileLock lock(long start, long size, boolean shared)
```

或

```
FileLock tryLock(long start, long size, boolean shared)
```

如果 `shared` 标志为 `false`，则锁定文件的目的是读写；而如果为 `true`，则这是一个共享锁，允许多个进程从文件中读入，并阻止任何进程获得独占的锁。并非所有的操作系统都支

持共享锁，因此你可能会在请求共享锁的时候得到独占的锁。调用 `FileLock` 类的 `isShared` 方法可以查询所持有的锁的类型。

注释：如果你锁定了文件的尾部，而这个文件的长度随后增长并超过了锁定的部分，那么增长出来的额外区域是未锁定的，要想锁定所有的字节，可以使用 `Long.MAX_VALUE` 来表示尺寸。

要确保在操作完成时释放锁，与往常一样，最好在一个带资源的 `try` 语句中执行释放锁的操作：

```
try (FileLock lock = channel.lock())
{
    access the locked file or segment
}
```

请记住，文件加锁机制是依赖于操作系统的，下面是需要注意的几点：

- 在某些系统中，文件加锁仅仅是建议性的，如果一个应用未能得到锁，它仍旧可以向被另一个应用并发锁定的文件执行写操作。
- 在某些系统中，不能在锁定一个文件的同时将其映射到内存中。
- 文件锁是由整个 Java 虚拟机持有的。如果有两个程序是由同一个虚拟机启动的（例如 Applet 和应用程序启动器），那么它们不可能每一个都获得一个在同一个文件上的锁。当调用 `lock` 和 `tryLock` 方法时，如果虚拟机已经在同一个文件上持有了另一个重叠的锁，那么这两个方法将抛出 `OverlappingFileLockException`。
- 在一些系统中，关闭一个通道会释放由 Java 虚拟机持有的底层文件上的所有锁。因此，在同一个锁定文件上应避免使用多个通道。
- 在网络文件系统上锁定文件是高度依赖于系统的，因此应该尽量避免。

API `java.nio.channels.FileChannel 1.4`

- `FileLock lock()`

在整个文件上获得一个独占的锁，这个方法将阻塞直至获得锁。

- `FileLock tryLock()`

在整个文件上获得一个独占的锁，或者在无法获得锁的情况下返回 `null`。

- `FileLock lock(long position, long size, boolean shared)`

- `FileLock tryLock(long position, long size, boolean shared)`

在文件的一个区域上获得锁。第一个方法将阻塞直至获得锁，而第二个方法将在无法获得锁时返回 `null`。参数 `shared` 的值为 `true` 表示共享锁，为 `false` 表示独占锁。

API `java.nio.channels.FileLock 1.4`

- `void close() 1.7`

释放这个锁。

2.7 正则表达式

正则表达式 (regular expression) 用于指定字符串的模式，可以在任何需要定位匹配某种特定模式的字符串的情况下使用正则表达式。例如，我们有一个示例程序就是用来定位 HTML 文件中的所有超链接的，它是通过查找 `` 模式的字符串来实现此目的。

当然，在指定模式时，`...` 标记法并不够精确。需要精确地指定什么样的字符序列才是合法的匹配，这就要求无论何时，当你要描述一个模式时，都需要使用某种特定的语法。

在下面各节中，我们将介绍 Java API 用到的正则表达式的语法，并讨论如何使用正则表达式。

2.7.1 正则表达式语法

下面是一个简单的示例，正则表达式

`[Jj]ava.+`

匹配下列形式的所有字符串：

- 第一个字母是 J 或 j。
- 接下来的三个字母是 ava。
- 字符串的其余部分由一个或多个任意的字符构成。

例如，字符串 "javanese" 就匹配这个特定的正则表达式，但是字符串 "Core Java" 就不匹配。

正如你所见，你需要了解一点这种语法，以理解正则表达式的含义。幸运的是，对于大多数情况，一小部分很直观的语法结构就足够用了。

- 字符类 (character class) 是一个括在括号中的可选择的字符集，例如，`[Jj]`、`[0-9]`、`[A-Za-z]` 或 `[^0-9]`。这里 “-” 表示是一个范围（所有 Unicode 值落在两个边界范围之内的字符），而 ^ 表示补集（除了指定字符之外的所有字符）。
- 如果字符类中包含 “-”，那么它必须是第一项或最后一项；如果要包含 “[”，那么它必须是第一项；如果要包含 “^”，那么它可以是除开始位置之外的任何位置。其中，你只需要转义 “[” 和 “\”。
- 有许多预定的字符类，例如 `\d` (数字) 和 `\p{Sc}` (Unicode 货币符号)。请查看表 2-6 和表 2-7。

表 2-6 正则表达式语法

表达式	描述	示例
字符		
<code>c, 除 .*+?{ () \^\$之外</code>	字符 <code>c</code>	<code>]</code>
<code>.</code>	任何除行终止符之外的字符，或者在 DOTALL 标志被设置时表示任何字符	
<code>\x{p}</code>	十六进制码为 <code>p</code> 的 Unicode 码点	<code>\x{1D546}</code>

(续)

表达式	描述	示例
\uhhhh,\xhh,\@o,\@oo,\@ooo	具有给定十六进制或八进制值的码元	\uFEFF
\a,\e,\f,\n,\r,\t	响铃符 (\x{7})、转义符 (\x{1B})、换页符 (\x{8})、换行符 (\x{A})、回车符 (\x{D})、指标符 (\x{9})	\n
\cc, 其中 c 在 [A-Z] 的范围内, 或者是 @[\]^_? 之一	对应于字符 c 的控制字符	\cH 是退格符 (\x{8})
\c, 其中 c 不在 [A-Za-z0-9] 的范围内	字符 c	\\"
\Q...\E	在左引号和右引号之间的所有字符	\Q(...)\\E 匹配字符串 (...)
字符类		
[C ₁ C ₂ ...], 其中 C _i 是多个字符, 范围从 c-d, 或者是字符类	任何由 C ₁ , C ₂ , ... 表示的字符	[0-9+-]
[^...]	某个字符类的补集	[^\d\s]
[...&...]	字符集的交集	[\\p{L}&&[^A-Za-z]]
\p{...}, \P{...}	某个预定义字符类 (参阅表 2-7); 它的补集	\p{L} 匹配一个 Unicode 字母, 而 \pl 也匹配这个字母, 可以忽略单个字母情况下的括号
\d, \D	数字 ([0-9], 或者在 UNICODE_CHARACTER_CLASS 标志被设置时表示 \p{Digit}); 它的补集	\d+ 是一个数字序列
\w, \W	单词字符 ([a-zA-Z0-9], 或者在 UNICODE_CHARACTER_CLASS 标志被设置时表示 Unicode 单词字符); 它的补集	
\s, \S	空格 ([\n\r\t\f\x{B}], 或者在 UNICODE_CHARACTER_CLASS 标志被设置时表示 \p{IsWhite_Space}); 它的补集	\s*, \s* 是由可选的空格字符包围的逗号
\h, \v, \H, \V	水平空白字符、垂直空白字符, 它们的补集	
序列和选择		
XY	任何 X 中的字符串, 后面跟随任何 Y 中的字符串	[1-9][0-9]* 表示没有前导零的正整数
X Y	任何 X 或 Y 中的字符串	http ftp
群组		
(X)	捕获 X 的匹配	'([^\"]*)' 捕获的是被引用的文本
\n	第 n 组	(["\"]).*\1 可以匹配 'Fred' 和 "Fred", 但是不能匹配 "Fred"
(?<name>X)	捕获与给定名字匹配的 X	'(?<id>[A-Za-z0-9]+)' 可以捕获名字为 id 的匹配
\k<name>	具有给定名字的组	\k<id> 可以匹配名字为 id 的组
(?:X)	使用括号但是不捕获 X	在(?:http ftp)://(.*) 中, 在 :// 之后的匹配是 \1

(续)

表达式	描述	示例
(? <i>f</i> ₁ ...: <i>X</i>) (? <i>f</i> ₁ ...: <i>X</i>), 其中 <i>f</i> 在 [dimsubx] 的范围内	匹配但是不捕获给定标志开或关 (在 - 之后) 的 <i>X</i>	(?i:jpe?g) 是大小写不敏感的匹配
其他 (?...)	请参阅 Pattern API 文档	
量词		
<i>X</i> ?	可选 <i>X</i>	\? 是可选的 + 号
<i>X</i> [*] , <i>X</i> ⁺	0 或多个 <i>X</i> , 1 或多个 <i>X</i>	[1-9][0-9]+ 是大于等于 10 的整数
<i>X</i> { <i>n</i> }, <i>X</i> { <i>n</i> }, <i>X</i> { <i>m</i> , <i>n</i> }	<i>n</i> 个 <i>X</i> , 至少 <i>n</i> 个 <i>X</i> , <i>m</i> 到 <i>n</i> 个 <i>X</i>	[0-7]{1,3} 是一位到三位的八进制数
<i>Q</i> ?, 其中 <i>Q</i> 是一个量词表达式	勉强量词, 在尝试最长匹配之前先尝试最短匹配	.*(<.+?>).* 捕获尖括号括起来的最短序列
<i>Q</i> +, 其中 <i>Q</i> 是一个量词表达式	占有量词, 在不回溯的情况下获取最长匹配	'[^']*+' 匹配单引号引起的字符串, 并且在字符串中没有右单引号的情况下立即匹配失败
边界匹配		
^, \$	输入的开头和结尾 (或者多行模式中的开头和结尾行)	^Java\$ 匹配输入中的 Java 或 Java 构成的行
\A, \Z, \z	输入的开头、输入的结尾、输入的绝对结尾 (在多行模式中不会发生变化)	
\b, \B	单词边界, 非单词边界	\bJava\b 匹配单词 Java
\R	Unicode 行分隔符	
\G	前一个匹配的结尾	

表 2-7 与 \p 一起使用的预定义字符类名字

字符类名字	解释
posixClass	<i>posixClass</i> 是 Lower、Upper、Alpha、Digit、Alnum、Punct、Graph、Print、Cntrl、XDigit、Space、Blank、ASCII 之一, 它会依 UNICODE_CHARACTER_CLASS 标志的值而被解释为 POSIX 或 Unicode 类
IsScript, sc=Script, script=Script	Character.UnicodeScript.forName 可以接受的脚本
InBlock, blk=Block, block=Block	Character.UnicodeScript.forName 可以接受的块
Category, InCategory, gc=Category, general_category=Category	Unicode 通用分类的单字母或双字母名字
IsProperty	<i>Property</i> 是 Alphabetic、Ideographic、Letter、Lowercase、Uppercase、Titlecase、Punctuation、Control、White_Space、Digit、Hex_Digit、Join_Control、Noncharacter_Code_Point、Assigned 之一
javaMethod	调用 Character.isMethod 方法 (必须不是过时的方法)

- 大部分字符都可以与它们自身匹配, 例如在前面示例中的 ava 字符。
- . 符号可以匹配任何字符 (有可能不包括行终止符, 这取决于标志的设置)。

- 使用 \ 作为转义字符，例如，\. 匹配句号而 \\ 匹配反斜线。
- ^ 和 \$ 分别匹配一行的开头和结尾。
- 如果 X 和 Y 是正则表达式，那么 XY 表示“任何 X 的匹配后面跟随 Y 的匹配”，X | Y 表示“任何 X 或 Y 的匹配”。
- 可以将量词运用到表达式 X：X+（1个或多个）、X*（0个或多个）与 X?（0个或1个）。
- 默认情况下，量词要匹配能够使整个匹配成功的最大可能的重复次数。可以修改这种行为，方法是使用后缀 ?（使用勉强或吝啬匹配，也就是匹配最小的重复次数）或使用后缀 +（使用占有或贪婪匹配，也就是即使让整个匹配失败，也要匹配最大的重复次数）。

例如，字符串 cab 匹配 [a-z]*ab，但是不匹配 [a-z]*+ab。在第一种情况下，表达式 [a-z]* 只匹配字符 c，使得字符 ab 匹配该模式的剩余部分；但是贪婪版本 [a-z]*+ 将匹配字符 cab，模式的剩余部分将无法匹配。

- 我们使用群组来定义子表达式，其中群组用括号 () 括起来。例如，([+-]?)(([0-9]+))。然后可以询问模式匹配器，让其返回每个组的匹配，或者用 \n 来引用某个群组，其中 n 是群组号（从 \1 开始）。

例如，下面是一个有些复杂但是却可能很有用的正则表达式，它描述了十进制和十六进制整数：

```
[+-]?(0-9)+|0[Xx][0-9A-Fa-f]+
```

遗憾的是，在使用正则表达式的各种程序和类库之间，表达式语法并未完全标准化。尽管在基本结构上达成了一致，但是它们在细节上仍旧存在着许多令人抓狂的差异。Java 正则表达式类使用的语法与 Perl 语言使用的语法十分相似，但是并不完全一样。表 2-6 展示的是 Java 语法中的所有结构。关于正则表达式语法的更多信息，可以求教于 Pattern 类的 API 文档和 Jeffrey E. F. Friedl 的 *Mastering Regular Expressions* (O'Reilly and Associates, 2006)。

2.7.2 匹配字符串

正则表达式的最简单用法就是测试某个特定的字符串是否与它匹配。下面展示了如何用 Java 来编写这种测试，首先用表示正则表达式的字符串构建一个 Pattern 对象。然后从这个模式中获得一个 Matcher，并调用它的 matches 方法：

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

这个匹配器的输入可以是任何实现了 CharSequence 接口的类的对象，例如 String、StringBuilder 和 CharBuffer。

在编译这个模式时，可以设置一个或多个标志，例如：

```
Pattern pattern = Pattern.compile(expression,
    Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

或者可以在模式中指定它们：

```
String regex = "(?iU:expression)";
```

下面是各个标志。

- Pattern.CASE_INSENSITIVE 或 i：匹配字符时忽略字母的大小写，默认情况下，这个标志只考虑 US ASCII 字符。
- Pattern.UNICODE_CASE 或 u：当与 CASE_INSENSITIVE 组合使用时，用 Unicode 字母的大小写来匹配。
- Pattern.UNICODE_CHARACTER_CLASS 或 U：选择 Unicode 字符类代替 POSIX，其中蕴含了 UNICODE_CASE。
- Pattern.MULTILINE 或 m：^ 和 \$ 匹配行的开头和结尾，而不是整个输入的开头和结尾。
- Pattern.UNIX_LINES 或 d：在多行模式中匹配 ^ 和 \$ 时，只有 '\n' 被识别成行终止符。
- Pattern.DOTALL 或 s：当使用这个标志时，. 符号匹配所有字符，包括行终止符。
- Pattern.COMMENTS 或 x：空白字符和注释（从 # 到行末尾）将被忽略。
- Pattern.LITERAL：该模式将被逐字地采纳，必须精确匹配，因字母大小写而造成的差异除外。
- Pattern.CANON_EQ：考虑 Unicode 字符规范的等价性，例如，u 后面跟随 “（分音符号）匹配 ü。

最后两个标志不能在正则表达式内部指定。

如果想要在集合或流中匹配元素，那么可以将模式转换为谓词：

```
Stream<String> strings = . . .;
Stream<String> result = strings.filter(pattern.asPredicate());
```

其结果中包含了匹配正则表达式的所有字符串。

如果正则表达式包含群组，那么 Matcher 对象可以揭示群组的边界。下面的方法

```
int start(int groupIndex)
int end(int groupIndex)
```

将产生指定群组的开始索引和结尾之后的索引。

可以直接通过调用下面的方法抽取匹配的字符串：

```
String group(int groupIndex)
```

群组 0 是整个输入，而用于第一个实际群组的群组索引是 1。调用 groupCount 方法可以获得全部群组的数量。对于具名的组，使用下面的方法

```
int start(String groupName)
int end(String groupName)
String group(String groupName)
```

嵌套群组是按照前括号排序的，例如，假设我们有下面的模式

```
(([1-9]|1[0-2]):([0-5][0-9]))[ap]m
```

和下面的输出

11:59am

那么，匹配器会报告下面的群组：

群组索引	开始	结束	字符串
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

程序清单 2-6 的程序提示输入一个模式，然后提示输入用于匹配的字符串，随后将打印出输入是否与模式相匹配。如果输入匹配模式，并且模式包含群组，那么这个程序将用括号打印出群组边界，例如

((11):(59))am

程序清单 2-6 regex/RegexTest.java

```

1 package regex;
2
3 import java.util.*;
4 import java.util.regex.*;
5
6 /**
7 * This program tests regular expression matching. Enter a pattern and strings to match,
8 * or hit Cancel to exit. If the pattern contains groups, the group boundaries are displayed
9 * in the match.
10 * @version 1.03 2018-05-01
11 * @author Cay Horstmann
12 */
13 public class RegexTest
14 {
15     public static void main(String[] args) throws PatternSyntaxException
16     {
17         var in = new Scanner(System.in);
18         System.out.println("Enter pattern: ");
19         String patternString = in.nextLine();
20
21         Pattern pattern = Pattern.compile(patternString);
22
23         while (true)
24         {
25             System.out.println("Enter string to match: ");
26             String input = in.nextLine();
27             if (input == null || input.equals("")) return;
28             Matcher matcher = pattern.matcher(input);
29             if (matcher.matches())
30             {
31                 System.out.println("Match");
32                 int g = matcher.groupCount();
33                 if (g > 0)
34                 {

```

```

35         for (int i = 0; i < input.length(); i++)
36     {
37         // Print any empty groups
38         for (int j = 1; j <= g; j++)
39             if (i == matcher.start(j) && i == matcher.end(j))
40                 System.out.print("(");
41         // Print ( for non-empty groups starting here
42         for (int j = 1; j <= g; j++)
43             if (i == matcher.start(j) && i != matcher.end(j))
44                 System.out.print('(');
45             System.out.print(input.charAt(i));
46         // Print ) for non-empty groups ending here
47         for (int j = 1; j <= g; j++)
48             if (i + 1 != matcher.start(j) && i + 1 == matcher.end(j))
49                 System.out.print(')');
50     }
51     System.out.println();
52 }
53 }
54 else
55     System.out.println("No match");
56 }
57 }
58 }
```

2.7.3 找出多个匹配

通常，你不希望用正则表达式来匹配全部输入，而只是想找出输入中一个或多个匹配的子字符串。这时可以使用 `Matcher` 类的 `find` 方法来查找匹配内容，如果返回 `true`，再使用 `start` 和 `end` 方法来查找匹配的内容，或使用不带引元的 `group` 方法来获取匹配的字符串。

```

while (matcher.find())
{
    int start = matcher.start();
    int end = matcher.end();
    String match = input.group();

    ...
}
```

在这种方式中，可以依次处理每个匹配。正如上面的代码片段所示，可以获取匹配的字符串，以及它在输入字符串中的位置。

更优雅的是，可以调用 `results` 方法来获取一个 `Stream<MatchResult>`。`MatchResult` 接口有 `group`、`start` 和 `end` 方法，就像 `Matcher` 一样。（事实上，`Matcher` 类实现了这个接口。）下面展示了如何获取所有匹配的列表：

```

List<String> matches = pattern.matcher(input)
    .results()
    .map(Matcher::group)
    .collect(Collectors.toList());
```

如果要处理的是文件中的数据，那么可以使用 `Scanner.findAll` 方法来获取一个 `Stream<Match-`

`Result`>, 这样就无须先将内容读取到一个字符串中。可以给 `Scanner` 传递一个 `Pattern` 或一个模式字符串:

```
var in = new Scanner(path, StandardCharsets.UTF_8);
Stream<String> words = in.findAll("\\pL+")
    .map(ResultSet::group);
```

程序清单 2-7 对这种机制进行了应用, 它定位一个 Web 页面上的所有超文本引用, 并打印它们。为了运行这个程序, 需要在命令行中提供一个 URL, 例如

```
java match.HrefMatch http://horstmann.com
```

程序清单 2-7 match/HrefMatch.java

```
1 package match;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.regex.*;
7
8 /**
9 * This program displays all URLs in a web page by matching a regular expression that
10 * describes the <a href=...> HTML tag. Start the program as <br>
11 * java match.HrefMatch URL
12 * @version 1.03 2018-03-19
13 * @author Cay Horstmann
14 */
15 public class HrefMatch
16 {
17     public static void main(String[] args)
18     {
19         try
20         {
21             // get URL string from command line or use default
22             String urlString;
23             if (args.length > 0) urlString = args[0];
24             else urlString = "http://openjdk.java.net/";
25
26             // read contents of URL
27             InputStream in = new URL(urlString).openStream();
28             var input = new String(in.readAllBytes(), StandardCharsets.UTF_8);
29
30             // search for all occurrences of pattern
31             var patternString = "<a\\s+href\\s*=\\s*(\"[^"]*\"|[^\s>]*\")\\s*>";
32             Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
33             pattern.matcher(input)
34                 .results()
35                 .map(ResultSet::group)
36                 .forEach(System.out::println);
37         }
38         catch (IOException | PatternSyntaxException e)
39         {
40             e.printStackTrace();
41         }
42     }
43 }
```

```

41     }
42 }
43 }
```

2.7.4 用分隔符来分割

有时，需要将输入按照匹配的分隔符断开，而其他部分保持不变。`Pattern.split` 方法可以自动完成这项任务。调用此方法后可以获得一个剔除分隔符之后的字符串数组：

```

String input = . . .;
Pattern commas = Pattern.compile("\s*,\s*");
String[] tokens = commas.split(input);
// "1, 2, 3" turns into ["1", "2", "3"]
```

如果有多个标记，那么可以惰性地获取它们：

```
Stream<String> tokens = commas.splitAsStream(input);
```

如果不关心预编译模式和惰性获取，那么可以使用 `String.split` 方法：

```
String[] tokens = input.split("\s*,\s*");
```

如果输入数据在文件中，那么需要使用扫描器：

```

var in = new Scanner(path, StandardCharsets.UTF_8);
in.useDelimiter("\s*,\s*");
Stream<String> tokens = in.tokens();
```

2.7.5 替换匹配

`Matcher` 类的 `replaceAll` 方法将正则表达式出现的所有地方都用替换字符串来替换。例如，下面的指令将所有的数字序列都替换成 # 字符。

```

Pattern pattern = Pattern.compile("[0-9]+");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");
```

替换字符串可以包含对模式中群组的引用：`$n` 表示替换成第 `n` 个群组， `${name}` 被替换成具有给定名字的组，因此我们需要用 `\$` 来表示在替换文本中包含一个 \$ 字符。

如果字符串中包含 \$ 和 \，但是又不希望它们被解释成群组的替换符，那么就可以调用 `matcher.replaceAll(Matcher.quoteReplacement(str))`。

如果想要执行比按照群组匹配拼接更复杂的操作，可以提供一个替换函数而不是替换字符串。该函数接受一个 `MatchResult` 对象，并会产生一个字符串。例如，在下面的代码中，我们将所有单词都替换成至少 4 个字母转换为大写形式的版本：

```

String result = Pattern.compile("\pL{4,}")
.matcher("Mary had a little lamb")
.replaceAll(m -> m.group().toUpperCase());
// Yields "MARY had a LITTLE LAMB"
```

`replaceFirst` 方法将只替换模式的第一次出现。

API **java.util.regex.Pattern 1.4**

- static Pattern compile(String expression)
- static Pattern compile(String expression, int flags)

把正则表达式字符串编译到一个用于快速处理匹配的模式对象中。flags 参数是 CASE_INSENSITIVE、UNICODE_CASE、MULTILINE、UNIX_LINES、DOTALL 和 CANON_EQ 标志中的一个。

- Matcher matcher(CharSequence input)
返回一个 matcher 对象，你可以用它在输入中定位模式的匹配。
- String[] split(CharSequence input)
- String[] split(CharSequence input, int limit)
- Stream<String> splitAsStream(CharSequence input) 8

将输入分割成标记，其中模式指定了分隔符的形式。返回标记数组，分隔符并非标记的一部分。第二种形式有一个名为 limit 的参数，表示所产生的字符串的最大数量。如果已经发现了 limit-1 个匹配的分隔符，那么返回的数组中的最后一项就包含所有剩余未分割的输入。如果 $limit \leq 0$ ，那么整个输入都被分割；如果 limit 为 0，那么结尾的空字符串将不会置于返回的数组中。

API **java.util.regex.Matcher 1.4**

- boolean matches()
如果输入匹配模式，则返回 true。
- boolean lookingAt()
如果输入的开头匹配模式，则返回 true。
- boolean find()
- boolean find(int start)
尝试查找下一个匹配，如果找到了另一个匹配，则返回 true。
- int start()
● int end()
返回当前匹配的开始索引和结尾之后的索引位置。
- String group()
返回当前的匹配。
- int groupCount()
返回输入模式中的群组数量。
- int start(int groupIndex)
- int start(String name) 8
- int end(int groupIndex)
- int end(String name) 8

返回当前匹配中给定群组的开始和结尾之后的位置。群组是由从 1 开始的索引指定

的，或者用 0 表示整个匹配，或者用表示具名群组的字符串来指定。

- `String group(int groupIndex)`
- `String group(String name)` 7

返回匹配给定群组的字符串。该群组是由从 1 开始的索引指定的，或者用 0 表示整个匹配，或者用表示具名群组的字符串来指定。

- `String replaceAll(String replacement)`
- `String replaceFirst(String replacement)`

返回从匹配器输入获得的通过将所有匹配或第一个匹配用替换字符串替换之后的字符串。

替换字符串可以包含用 `$n` 表示的对群组的引用，这时需要用 `\$` 来表示字符串中包含一个 `$` 符号。

- `static String quoteReplacement(String str) 5.0`
引用 `str` 中的所有 `\` 和 `$`。
- `String replaceAll(Function<MatchResult, String> replacer) 9`
将每个匹配都替换为 `replacer` 函数应用于 `MatchResult` 上所产生的结果。
- `Stream<MatchResult> results() 9`
产生一个包含所有匹配结果的流。

API `java.util.regex.MatchResult 5`

- `String group()`
- `String group(int group)`
产生匹配的字符串，或者匹配给定群组的字符串。
- `int start()`
- `int end()`
- `int start(int group)`
- `int end(int group)`
产生匹配字符串或匹配给定群组的字符串的开始与结尾的偏移量。

API `java.util.Scanner 5.0`

- `Stream<MatchResult> findAll(Pattern pattern) 9`
产生一个流，其中包含了这个扫描器所产生的输入中针对给定模式的所有匹配。

你现在已经看到了在 Java 中输入输出操作是如何实现的，也对作为“新 I/O”规范一部分的正则表达式有了概略的了解。在下一章中，我们将转而研究对 XML 数据的处理。

第3章 XML

- ▲ XML 概述
- ▲ XML 文档的结构
- ▲ 解析 XML 文档
- ▲ 验证 XML 文档
- ▲ 使用 XPath 来定位信息

- ▲ 使用命名空间
- ▲ 流机制解析器
- ▲ 生成 XML 文档
- ▲ XSL 转换

Don Box 等人在其合著的 *Essential XML* (Addison-Wesley 出版社 2000 年出版) 的前言中半开玩笑地说道：“可扩展标记语言 (Extensible Markup Language, XML) 已经取代了 Java、设计模式、对象技术，成为软件行业解决世界饥荒的方案。”这种炒作早就不新鲜了，但是正如你将在本章中看到的，XML 是一种非常有用的数据结构化信息的技术。XML 工具使处理和转化信息变得十分容易。但是，XML 并不是万能药，我们需要领域相关的标准和代码库才能有效地使用 XML。此外，XML 非但没有使 Java 技术过时，还与 Java 配合得很好。从 20 世纪 90 年代末以来，IBM、Apache 和其他许多公司一直在帮助开发用于 XML 处理的高质量 Java 库，其中大部分重要的代码库都整合到了 Java 平台中。

本章将介绍 XML，并涵盖了 Java 库的 XML 特性。一如既往，我们将指出何时大量地使用 XML 是正确的；而何时必须有保留地使用 XML，通过利用良好的设计和代码，来采用老办法解决问题。

3.1 XML 概述

在卷 I 第 13 章中，你已经看见过用属性文件 (property file) 来描述程序配置。属性文件包含了一组名 / 值对，例如：

```
fontname=Times Roman  
fontsize=12  
windowsize=400 200  
color=0 50 100
```

可以用 `Properties` 类在单个方法调用中读入这样的属性文件。这是一个很好的特性，但这还不够。在许多情况下，想要描述的信息的结构比较复杂，属性文件不能很方便地处理它。例如，对于下面例子中的 `fontname/fontsize` 项，使用以下的单一项将更符合面向对象的要求：

```
font=Times Roman 12
```

但是，这时对字体描述的解析就变得很讨厌了，必须确定字体名在何处结束，字体大小

在何处开始。

属性文件采用的是一种单一的平面层次结构。你常常会看到程序员用如下的键名来努力解决这种局限性：

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

属性文件格式的另一个缺点是要求键是唯一的。如果要存放一个值序列，则需要另一个变通方法，例如：

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

XML 格式解决了这些问题，因为它能够表示层次结构，这比属性文件的平面表结构更灵活。

描述程序配置的 XML 文件可能会像这样：

```
<config>
  <entry id="title">
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </entry>
  <entry id="body">
    <font>
      <name>Times Roman</name>
      <size>12</size>
    </font>
  </entry>
  <entry id="background">
    <color>
      <red>0</red>
      <green>50</green>
      <blue>100</blue>
    </color>
  </entry>
</config>
```

XML 格式能够表达层次结构，并且重复的元素不会被曲解。

正如上面看到的，XML 文件的格式非常直观，它与 HTML 文件非常相似。这是有原因的，因为 XML 和 HTML 格式是古老的标准通用标记语言（Standard Generalized Markup Language，SGML）的衍生语言。

SGML 从 20 世纪 70 年代开始就用于描述复杂文件的结构。它的使用在一些要求对海量文献进行持续维护的产业中取得了成功，特别是在飞机制造业中。但是，SGML 相当复杂，所以它从未风行。造成 SGML 如此复杂的主要原因是 SGML 有两个相互矛盾的目标。它既想要确保文档能够根据其文档类型的规则来形成，又想要通过可以减少数据键入的快捷方式

使数据项变得容易表示。XML 设计成了一个用于因特网的 SGML 的简化版本。和通常情况一样，越简单的东西越好，XML 立即得到了长期以来一直在躲避 SGML 的用户的热情追捧。

注释：在 <http://www.xml.com/axml/axml.html> 处可以找到一个由 Tim Bray 注释的 XML 标准的极佳版本。

尽管 HTML 和 XML 同宗同源，但是两者之间存在着重要的区别：

- 与 HTML 不同，XML 是大小写敏感的。例如，`<H1>` 和 `<h1>` 是不同的 XML 标签。
- 在 HTML 中，如果从上下文中可以分清哪里是段落或列表项的结尾，那么结束标签（如 `</p>` 或 ``）就可以省略，而在 XML 中结束标签绝对不能省略。
- 在 XML 中，只有单个标签而没有相对应的结束标签的元素必须以 / 结尾，比如 ``。这样，解析器就知道不需要查找 `` 标签了。
- 在 XML 中，属性值必须用引号括起来。在 HTML 中，引号是可有可无的。例如，`<applet code="MyApplet.class" width=300 height=300>` 对 HTML 来说是合法的，但是对 XML 来说则是不合法的。在 XML 中，必须使用引号，比如，`width= "300"`。
- 在 HTML 中，属性名可以没有值。例如，`<input type="radio" name="language" value="Java" checked>`。在 XML 中，所有属性必须都有属性值。比如，`checked= "true"` 或 `checked="checked"`。

3.2 XML 文档的结构

XML 文档应当以一个文档头开始，例如：

```
<?xml version="1.0"?>
```

或者

```
<?xml version="1.0" encoding="UTF-8"?>
```

严格来说，文档头是可选的，但是强烈推荐使用文档头。

注释：因为建立 SGML 是为了处理真正的文档，因此 XML 文件被称为文档，尽管许多 XML 文件是用来描述通常不被称作文档的数据集的。

文档头之后通常是文档类型定义 (Document Type Definition, DTD)，例如：

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

文档类型定义是确保文档正确的一个重要机制，但是它不是必需的。我们将在本章的后面讨论这个问题。

最后，XML 文档的正文包含根元素，根元素包含其他元素。例如：

```
<?xml version="1.0"?>
<!DOCTYPE config . . .>
<config>
```

```

<entry id="title">
  <font>
    <name>Helvetica</name>
    <size>36</size>
  </font>
</entry>
. . .
</config>

```

元素可以有子元素 (child element)、文本或两者皆有。在上述例子中，`font` 元素有两个子元素，它们是 `name` 和 `size`。`name` 元素包含文本 “Helvetica”。

 提示：在设计 XML 文档结构时，最好让元素要么包含子元素，要么包含文本。换句话说，你应该避免下面的情况：

```

<font>
  Helvetica
  <size>36</size>
</font>

```

在 XML 规范中，这叫作混合式内容 (mixed content)。在本章中，稍后你将会看到，如果避免了混合式内容，就可以简化解析过程。

XML 元素可以包含属性，例如：

```
<size unit="pt">36</size>
```

何时用元素，何时用属性，在 XML 设计人员中存在一些分歧。例如，将 `font` 做如下描述：

```
<font name="Helvetica" size="36"/>
```

似乎比下面的描述更简单一些：

```

<font>
  <name>Helvetica</name>
  <size>36</size>
</font>

```

但是，属性的灵活性要差很多。假设你想把单位添加到 `size` 的值中去，如果使用属性，那么就必须把单位添加到属性值中去：

```
<font name="Helvetica" size="36 pt"/>
```

嗨！现在必须对字符串 “36 pt” 进行解析，而这正是 XML 被设计用来避免的那种麻烦。而向 `size` 元素中添加一个属性看起来会清晰得多：

```

<font>
  <name>Helvetica</name>
  <size unit="pt">36</size>
</font>

```

一条常用的经验法则是，属性只应该用来修改值的解释，而不是用来指定值。如果你发现自己陷入了争论，在纠结于某个设置是否是对某个值的解释所做的修改，那么你就应该对

属性说“不”，转而使用元素，许多有用的文档根本就不使用属性。

注释：在 HTML 中，属性的使用规则很简单：凡是不显示在网页上的都是属性。例如在下面的超链接中：

```
<a href="http://java.sun.com">Java Technology</a>
```

字符串 Java Technology 要在网页上显示，但是这个链接的 URL 并不是显示页面的一部分。然而，这个规则对于大多数 XML 并不那么管用，因为 XML 文件中的数据并非像通常意义那样是让人浏览的。

元素和文本是 XML 文档“主要的支撑要素”，你可能还会遇到的其他一些标记，说明如下：

- 字符引用 (character reference) 的形式是 &#十进制值；或 &x十六进制值；。例如，字符 é 可以用下面两种形式表示：

```
&#233; &#xE9;
```

- 实体引用 (entity reference) 的形式是 &name;。下面这些实体引用：

```
&lt; &gt; &amp; &quot; &apos;
```

都有预定义的含义：小于、大于、&、引号、省略号等字符。还可以在 DTD 中定义其他的实体引用。

- CDATA 部分 (CDATA Section) 用 <![CDATA[和]]> 来限定其界限。它们是字符数据的一种特殊形式。可以使用它们来囊括那些含有 <、>、& 之类字符的字符串，而不必将它们解释为标记，例如：

```
<![CDATA[< &gt; are my favorite delimiters]]>
```

CDATA 部分不能包含字符串]]>。使用这一特性时要特别小心，因为它常用来当作将遗留数据偷偷纳入 XML 文档的一个后门。

- 处理指令 (processing instruction) 是那些专门在处理 XML 文档的应用程序中使用的指令，它们由 <? 和 ?> 来限定其界限，例如：

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

每个 XML 都以一个处理指令开头：

```
<?xml version="1.0"?>
```

- 注释 (comment) 用 <!-- 和 --> 限定其界限，例如：

```
<!-- This is a comment. -->
```

注释不应该含有字符串 --。注释只能是给文档的读者提供的信息，其中绝不应该含有隐藏的命令，命令应该是用处理指令来实现。

3.3 解析 XML 文档

要处理 XML 文档，就要先解析 (parse) 它。解析器是这样一个程序：它读入一个文件，

确认这个文件具有正确的格式，然后将其分解成各种元素，使得程序员能够访问这些元素。Java 库提供了两种 XML 解析器：

- 像文档对象模型（Document Object Model, DOM）解析器这样的树型解析器（tree parser），它们将读入的 XML 文档转换成树结构。
- 像 XML 简单 API（Simple API for XML, SAX）解析器这样的流机制解析器（streaming parser），它们在读入 XML 文档时生成相应的事件。

DOM 解析器对于实现我们的大多数目的来说都更容易一些，所以我们首先介绍它。如果要处理很长的文档，用它生成树结构将会消耗大量内存，或者如果只是对于某些元素感兴趣，而不关心它们的上下文，那么在这些情况下应该考虑使用流机制解析器。更多的信息可以查看 3.7 节。

DOM 解析器的接口已经被 W3C 标准化了。`org.w3c.dom` 包中包含了这些接口类型的定义，比如：`Document` 和 `Element` 等。不同的提供者，比如 Apache 组织和 IBM，都编写了实现这些接口的 DOM 解析器。Java XML 处理 API（Java API for XML Processing, JAXP）库使得我们实际上可以以插件形式使用这些解析器中的任意一个。但是 JDK 中也包含了从 Apache 解析器导出的 DOM 解析器。

要读入一个 XML 文档，首先需要一个 `DocumentBuilder` 对象，可以从 `DocumentBuilderFactory` 中得到这个对象，例如：

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

现在，可以从文件中读入某个文档：

```
File f = . . .;
Document doc = builder.parse(f);
```

或者，可以用一个 URL：

```
URL u = . . .;
Document doc = builder.parse(u);
```

甚至可以指定一个任意的输入流：

```
InputStream in = . . .;
Document doc = builder.parse(in);
```

注释：如果使用输入流作为输入源，那么对于那些以该文档的位置为相对路径而被引用的文档，解析器将无法定位，比如在同一个目录中的 DTD。但是，可以通过安装一个“实体解析器”（entity resolver）来解决这个问题。请查看 www.xml.com/pub/a/2004/03/03/catalogs.html 或 www.ibm.com/developerworks/xml/library/x-mxd3.html，以了解更多信息。

`Document` 对象是 XML 文档的树型结构在内存中的表示方式，它由实现了 `Node` 接口及其各子接口的类的对象构成。图 3-1 显示了各个子接口的层次结构。

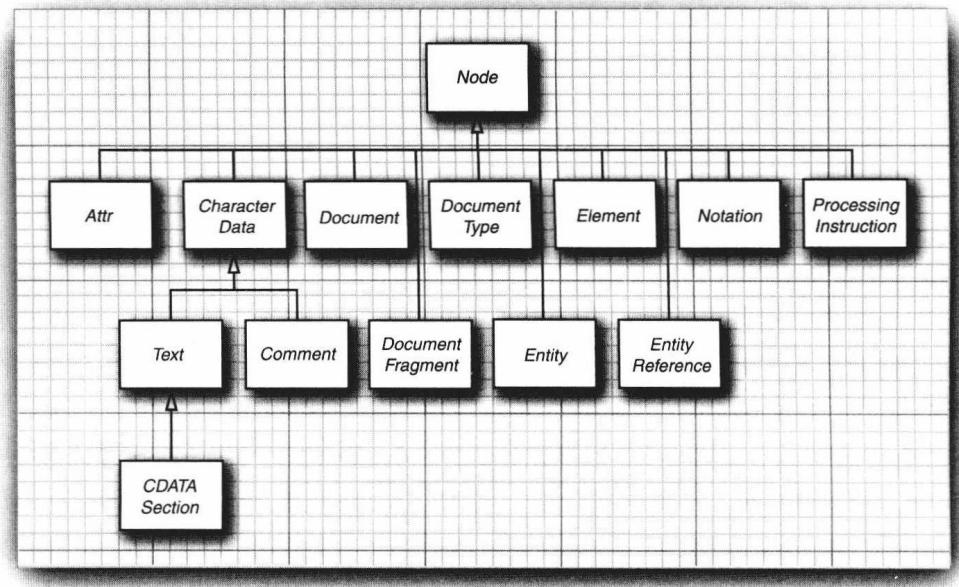


图 3-1 Node 接口及其子接口

可以通过调用 `getDocumentElement` 方法来启动对文档内容的分析，它将返回根元素。

```
Element root = doc.getDocumentElement();
```

例如，如果要处理下面的文档：

```
<?xml version="1.0"?>
<font>
  .
  .
</font>
```

那么，调用 `getDocumentElement` 方法可以返回 `font` 元素。`getTagName` 方法可以返回元素的标签名。在前面这个例子中，`root.getTagName()` 返回字符串 "font"。

如果要得到该元素的子元素（可能是子元素、文本、注释或其他节点），请使用 `getChildNodes` 方法，这个方法会返回一个类型为 `NodeList` 的集合。这个类型在标准的 Java 集合类创建之前就已经被标准化了，因此它具有一种不同的访问协议；`item` 方法将得到指定索引值的项；`getLength` 方法则提供了项的总数。因此，我们可以像下面这样枚举所有子元素：

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    .
}
```

分析子元素时要很仔细。例如，假设你正在处理以下文档：

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

你预期 font 有两个子元素，但是解析器却报告说有 5 个：

- 和 <name> 之间的空白字符
- name 元素
- </name> 和 <size> 之间的空白字符
- size 元素
- </size> 和 之间的空白字符

图 3-2 显示了其 DOM 树。

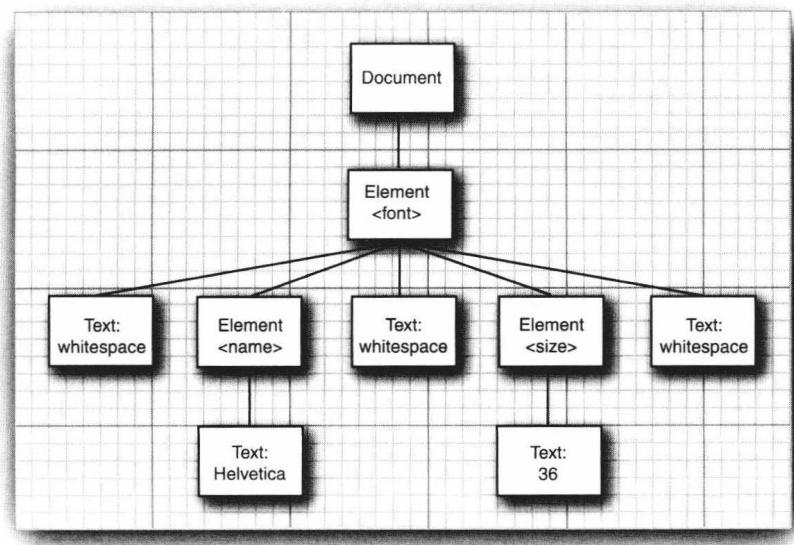


图 3-2 一棵简单的 DOM 树

如果只希望得到子元素，那么可以忽略空白字符：

```

for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        var childElement = (Element) child;
        ...
    }
}
  
```

现在，只会看到两个元素，它们的标签名是 name 和 size。

正如将在下一节中所看到的那样，如果你的文档有 DTD，那么你就可以做得更好。这时，解析器知道哪些元素没有文本节点的子元素，而且它会帮你剔除空白字符。

在分析 name 和 size 元素时，你肯定想获取它们包含的文本字符串。这些文本字符串本身都包含在 Text 类型的子节点中。既然知道了这些 Text 节点是唯一的子元素，就可以用 `getFirstChild` 方法而不用再遍历另一个 `NodeList`。然后可以用 `getData` 方法获取存储在 Text 节

点中的字符串。

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        var childElement = (Element) child;
        var textNode = (Text) childElement.getFirstChild();
        String text = textNode.getData().trim();
        if (childElement.getTagName().equals("name"))
            name = text;
        else if (childElement.getTagName().equals("size"))
            size = Integer.parseInt(text);
    }
}
```

 提示：对 `getData` 的返回值调用 `trim` 方法是个好主意。如果 XML 文件的作者将起始和结束的标签放在不同的行上，例如：

```
<size>
  36
</size>
```

那么，解析器将会把所有的换行符和空格都包含到文本节点中去。调用 `trim` 方法可以把位于实际数据前后的空白字符删掉。

也可以用 `getLastChild` 方法得到最后一项子元素，用 `getNextSibling` 得到下一个兄弟节点。这样，另一种遍历子节点集的方法就是：

```
for (Node childNode = element.getFirstChild();
     childNode != null;
     childNode = childNode.getNextSibling())
{
    ...
}
```

如果要枚举节点的属性，可以调用 `getAttributes` 方法。它返回一个 `NamedNodeMap` 对象，其中包含了描述属性的 `Node` 对象。可以用和遍历 `NodeList` 一样的方式在 `NamedNodeMap` 中遍历各子节点。然后，调用 `getNodeName` 和 `getNodeValue` 方法可以得到属性名和属性值。

```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
    Node attribute = attributes.item(i);
    String name = attribute.getNodeName();
    String value = attribute.getNodeValue();
    ...
}
```

或者，如果知道属性名，则可以直接获取相应的属性值：

```
String unit = element.getAttribute("unit");
```

现在你已经知道怎么分析 DOM 树了。程序清单 3-1 中的程序将这些技术都运用了一遍，

将一个 XML 文档转换成了 JSON 格式。

程序清单 3-1 dom/JSONConverter.java

```
1 package dom;
2
3 import java.io.*;
4 import java.util.*;
5
6 import javax.xml.parsers.*;
7
8 import org.w3c.dom.*;
9 import org.w3c.dom.CharacterData;
10 import org.xml.sax.*;
11
12 /**
13 * This program displays an XML document as a tree in JSON format.
14 * @version 1.2 2018-04-02
15 * @author Cay Horstmann
16 */
17 public class JSONConverter
18 {
19     public static void main(String[] args)
20         throws SAXException, IOException, ParserConfigurationException
21     {
22         String filename;
23         if (args.length == 0)
24         {
25             try (var in = new Scanner(System.in))
26             {
27                 System.out.print("Input file: ");
28                 filename = in.nextLine();
29             }
30         }
31         else
32             filename = args[0];
33         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
34         DocumentBuilder builder = factory.newDocumentBuilder();
35
36         Document doc = builder.parse(filename);
37         Element root = doc.getDocumentElement();
38         System.out.println(convert(root, 0));
39     }
40
41     public static StringBuilder convert(Node node, int level)
42     {
43         if (node instanceof Element)
44         {
45             return elementObject((Element) node, level);
46         }
47         else if (node instanceof CharacterData)
48         {
49             return characterString((CharacterData) node, level);
50         }
51         else
```

```

53         return pad(new StringBuilder(), level).append(
54             jsonEscape(node.getClass().getName()));
55     }
56 }
57
58 private static Map<Character, String> replacements = Map.of('\b', "\\b",
59             '\f', "\\f",
60             '\n', "\\n",
61             '\r', "\\r",
62             '\t', "\\t",
63             '\"', "\\\"",
64             '\\', "\\\\");

65 private static StringBuilder jsonEscape(String str)
66 {
67     var result = new StringBuilder("\\\"");
68     for (int i = 0; i < str.length(); i++)
69     {
70         char ch = str.charAt(i);
71         String replacement = replacements.get(ch);
72         if (replacement == null) result.append(ch);
73         else result.append(replacement);
74     }
75     result.append("\\\"");
76     return result;
77 }

78 private static StringBuilder characterString(CharacterData node, int level)
79 {
80     var result = new StringBuilder();
81     StringBuilder data = jsonEscape(node.getData());
82     if (node instanceof Comment) data.insert(1, "Comment: ");
83     pad(result, level).append(data);
84     return result;
85 }

86 private static StringBuilder elementObject(Element elem, int level)
87 {
88     var result = new StringBuilder();
89     pad(result, level).append("{\n");
90     pad(result, level + 1).append("\"name\": ");
91     result.append(jsonEscape(elem.getTagName()));
92     NamedNodeMap attrs = elem.getAttributes();
93     if (attrs.getLength() > 0)
94     {
95         pad(result.append(",\n"), level + 1).append("\"attributes\": ");
96         result.append(attributeObject(attrs));
97     }
98     NodeList children = elem.getChildNodes();
99     if (children.getLength() > 0)
100    {
101        pad(result.append(",\n"), level + 1).append("\"children\": [\n");
102        for (int i = 0; i < children.getLength(); i++)
103        {
104            if (i > 0) result.append(",\n");
105            result.append(convert(children.item(i), level + 2));
106        }
107        result.append("\n");
108        pad(result, level + 1).append("]\n");
109    }
110 }

```

```

107     }
108     pad(result, level).append("}");
109     return result;
110 }
111
112 private static StringBuilder pad(StringBuilder builder, int level)
113 {
114     for (int i = 0; i < level; i++) builder.append("  ");
115     return builder;
116 }
117
118 private static StringBuilder attributeObject(NamedNodeMap attrs)
119 {
120     var result = new StringBuilder("{}");
121     for (int i = 0; i < attrs.getLength(); i++)
122     {
123         if (i > 0) result.append(", ");
124         result.append(jsonEscape(attrs.item(i).getNodeName()));
125         result.append(": ");
126         result.append(jsonEscape(attrs.item(i).getNodeValue()));
127     }
128     result.append("}");
129     return result;
130 }
131 }

```

该树形结构清楚地显示了子元素是怎样被包含空白字符和注释的文本包围起来的。你可以清楚地看到换行符和回车符显示成\n。

无须熟悉 JSON 就可以理解这个程序是如何操作 DOM 树的，你只需观察以下几点：

- 我们使用了一个 DocumentBuilderFactory 来从文件中读取一个 Document。
- 对于每一个元素，我们打印了标签名、属性和元素。
- 对于字符数据，我们用这些数据产生了一个字符串。如果数据来自于注释，那么我们就会添加 "Comment:" 前缀。

API javax.xml.parsers.DocumentBuilderFactory 1.4

- static DocumentBuilderFactory newInstance()
返回 DocumentBuilderFactory 类的一个实例。
- DocumentBuilder newDocumentBuilder()
返回 DocumentBuilder 类的一个实例。

API javax.xml.parsers.DocumentBuilder 1.4

- Document parse(File f)
- Document parse(String url)
- Document parse(InputStream in)

解析来自给定文件、URL 或输入流的 XML 文档，返回解析后的文档。

API ***org.w3c.dom.Document*** 1.4

- `Element getDocumentElement()`

返回文档的根元素。

API ***org.w3c.dom.Element*** 1.4

- `String getTagName()`
返回元素的名字。
- `String getAttribute(String name)`
返回给定名字的属性值，没有该属性时返回空字符串。

API ***org.w3c.dom.Node*** 1.4

- `NodeList getChildNodes()`
返回包含该节点所有子元素的节点列表。
- `Node getFirstChild()`
- `Node getLastChild()`
获取该节点的第一个或最后一个子节点，在该节点没有子节点时返回 `null`。
- `Node getNextSibling()`
- `Node getPreviousSibling()`
获取该节点的下一个或上一个兄弟节点，在该节点没有兄弟节点时返回 `null`。
- `Node getParentNode()`
获取该节点的父节点，在该节点是文档节点时返回 `null`。
- `NamedNodeMap getAttributes()`
返回含有描述该节点所有属性的 `Attr` 节点的映射表。
- `String getNodeName()`
返回该节点的名字。当该节点是 `Attr` 节点时，该名字就是属性名。
- `String getNodeValue()`
返回该节点的值。当该节点是 `Attr` 节点时，该值就是属性值。

API ***org.w3c.dom.CharacterData*** 1.4

- `String getData()`
返回存储在节点中的文本。

API ***org.w3c.dom.NodeList*** 1.4

- `int getLength()`
返回列表中的节点数。
- `Node item(int index)`
返回给定索引值处的节点。索引值范围在 0 到 `getLength()-1` 之间。

API org.w3c.dom.NamedNodeMap 1.4

- int getLength()
返回该节点映射表中的节点数。
- Node item(int index)
返回给定索引值处的节点。索引值范围在 0 到 getLength()-1 之间。

3.4 验证 XML 文档

在前一节中，我们了解了如何遍历 DOM 文档的树形结构。然而，如果仅仅按照这种方法来操作，会发现需要大量冗长的编程和错误检查工作。不但需要处理元素间的空白字符，还要检查该文档包含的节点是否和期望的一样。例如，在读入下面这个元素时：

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

将首先得到第一个子节点，这是一个含有空白字符 "\n" 的文本节点。跳过文本节点找到第一个元素节点。然后，要检查它的标签名是不是 "name"，还要检查它是否有一个 Text 类型的子节点。接下来，转到下一个非空白字符的子节点，并进行同样的检查。那么，当文档作者改变了子元素的顺序或是加入另一个子元素时又会怎样呢？要是对所有的错误检查都进行编码，就会显得太琐碎麻烦了，而跳过这些检查又显得不慎重。

幸好，XML 解析器的一个很大的好处就是它能自动校验某个文档是否具有正确的结构。这样，解析就变得简单多了。例如，如果知道 font 片段已经通过了验证，那么不用进一步检查就能得到其两个孙节点，并把它们转换成 Text 节点，得到它们的文本数据。

如果要指定文档结构，可以提供一个文档类型定义（DTD）或一个 XML Schema 定义。DTD 或 schema 包含了用于解释文档应如何构成的规则，这些规则指定了每个元素的合法子元素和属性。例如，某个 DTD 可能含有一项规则：

```
<!ELEMENT font (name,size)>
```

这项规则表示，一个 font 元素必须总是有两个子元素，分别是 name 和 size。将同样的约束用 XML Schema 表示如下：

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

与 DTD 相比，XML Schema 可以表达更加复杂的验证条件（比如 size 元素必须包含一个整数）。与 DTD 语法不同，XML Schema 自身使用的就是 XML，这为处理 Schema 文件带来了方便。

在下一节中，我们将详细讨论 DTD。接着简要介绍 XML Schema 的一些基础知识。最后，我们会展示一个完整的应用程序来演示验证是如何简化 XML 编程的。

3.4.1 文档类型定义

提供 DTD 的方式有多种。可以像下面这样将其纳入到 XML 文档中：

```
<?xml version="1.0"?>
<!DOCTYPE config [
    <!ELEMENT config . . .>
    more rules
    .
    .
]>
<config>
    .
    .
</config>
```

正如你看到的，这些规则被纳入到 `DOCTYPE` 声明中，位于由 [...] 限定界限的块中。文档类型必须匹配根元素的名字，比如我们例子中的 `configuration`。

在 XML 文档内部提供 DTD 不是很普遍，因为 DTD 会使文件长度变得很长。把 DTD 存储在外部会更具意义，`SYSTEM` 声明可以用来实现这个目标。可以指定一个包含 DTD 的 URL，例如：

```
<!DOCTYPE config SYSTEM "config.dtd">
```

或者

```
<!DOCTYPE config SYSTEM "http://myserver.com/config.dtd">
```

！ 警告：如果使用的是 DTD 的相对 URL（比如 "config.dtd"），那么要给解析器一个 `File` 或 `URL` 对象，而不是 `InputStream`。如果必须从一个输入流来解析，那么请提供一个实体解析器（请看下面的说明）。

最后，有一个来源于 SGML 的用于识别“众所周知的” DTD 的机制，下面是一个例子：

```
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

如果 XML 处理器知道如何定位带有公共标识符的 DTD，那么就不需要 URL 了。

■ 注释：DTD 的系统标识符 URL 可能实际无法工作，或者会显著地降低性能。后者有一个例子，即 XHTML1.0 Strict DTD 的系统标识符，可以在 <http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd> 处找到它的信息。如果要解析一个 XHTML 文件，可能会花费一两分钟来处理 DTD。

有一种解决方案是使用 **实体解析器**，它会将公共标识符映射为本地文件。在 Java 9 之前，我们不得不提供一个实现了 `EntityResolver` 接口并实现了 `resolveEntity` 方法的某个类的对象。

但是，现在我们可以使用 XML 目录来管理这种映射。我们需要提供一个或多个具有下面这种形式的目录文件：

```
<?xml version="1.0"?>
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.0//EN"
    "http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:catalog" prefer="public">
    <public publicId="..." uri="..." />
    ...
</catalog>
```

然后像下面这样构造和安装一个解析器：

```
builder.setEntityResolver(CatalogManager.catalogResolver(
    CatalogFeatures.defaults(),
    Paths.get("catalog.xml").toAbsolutePath().toUri()));
```

请参阅程序清单 3.6 中完整的示例。

除了在程序中设置目录文件的位置，还可以在命令行中用 `javax.xml.catalog.files` 系统属性来设置它，我们需要提供由分号分隔的 `file` 的绝对 URL。

既然你已经知道解析器怎样定位 DTD 了，那么下面就让我们来看看不同种类的规则。

`ELEMENT` 规则用于指定某个元素可以拥有什么样的子元素。可以指定一个正则表达式，它由表 3-1 中所示的组成部分构成。

表 3-1 用于元素内容的规则

规则	含义
E^*	0 或多个 E
E^+	1 或多个 E
$E^?$	0 或 1 个 E
$E_1 E_2 \dots E_n$	E_1, E_2, \dots, E_n 中的一个
E_1, E_2, \dots, E_n	E_1 后面跟着 E_2, \dots, E_n
<code>#PCDATA</code>	文本
$(\#PCDATA E_1 E_2 \dots E_n)^*$	0 或多个文本且 E_1, E_2, \dots, E_n 以任意顺序排列（混合式内容）
<code>ANY</code>	允许有任意子元素
<code>EMPTY</code>	不允许有子元素

下面是一些简单而典型的例子。下面的规则声明了 `menu` 元素包含 0 或多个 `item` 元素：

```
<!ELEMENT menu (item)*>
```

下面这组规则声明 `font` 是用一个 `name` 后跟一个 `size` 来描述的，它们都包含文本：

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

缩写 `PCDATA` 表示被解析的字符数据。这些数据之所以被称为“被解析的”是因为解析器通过寻找表示一个新标签起始的 `<` 字符或表示一个实体起始的 `&` 字符，来解释这些文本字符串。

元素的规格说明可以包含嵌套的和复杂的正则表达式，例如，下面是一个描述了本书中每一章的结构的规则：

```
<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)
```

每章都以简介开头，其后是 1 或多个小节，每个小节由一个标题和 1 个或多个段落、图片、表格或说明构成。

然而，有一种常见的情况是无法把规则定义得像你希望的那样灵活的。当一个元素可以包含文本时，那么就只有两种合法的情况。要么该元素只包含文本，比如：

```
<!ELEMENT name (#PCDATA)>
```

要么该元素包含任意顺序的文本和标签的组合，比如：

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

指定其他任何类型的包含 #PCDATA 的规则都是不合法的。例如，以下规则是非法的：

```
<!ELEMENT captionedImage (image,#PCDATA)>
```

必须重写这项规则，以引入另一个 `caption` 元素或者允许使用 `image` 元素和文本的任意组合。

这种限制简化了 XML 解析器在解析混合式内容（标签和文本的混合）时的工作。因为在允许使用混合式内容时难免会失控，所以最好在设计 DTD 时，让其中所有的元素要么包含其他元素，要么只有文本。

注释：实际上，在 DTD 规则中并不能为元素指定任意的正则表达式，XML 解析器会拒绝某些导致非确定性的复杂规则。例如，正则表达式 $((x,y)|(x,z))$ 就是非确定性的。当解析器看到 `x` 时，它不知道在两个选择中应该选取哪一个。这个表达式可以改写成确定性的形式，如 $(x,(y|z))$ 。然而，有一些表达式不能被改写，如 $((x,y)^*|x?)$ 。Java XML 库中的解析器在遇到有歧义的 DTD 时，不会给出警告。在解析时，它仅仅在两者中选取第一个匹配项，这将导致它会拒绝一些正确的输入。当然，解析器有权这么做，因为 XML 标准允许解析器假设 DTD 都是非二义性的。

还可以指定描述合法的元素属性的规则，其通用语法为：

```
<!ATTLIST element attribute type default>
```

表 3-2 显示了合法的属性类型 (type)，表 3-3 显示了属性默认值 (default) 的语法。

表 3-2 属性类型

类型	含义
CDATA	任意字符串
$(A_1 A_2 \dots A_n)$	字符串属性 $A_1 A_2 \dots A_n$ 之一
NMTOKEN, NMTOKENS	1 或多个名字标记
ID	1 个唯一的 ID
IDREF, IDREFS	1 或多个对唯一 ID 的引用
ENTITY, ENTITIES	1 或多个未解析的实体

表 3-3 属性的默认值

默认值	含义
#REQUIRED	属性是必需的
#IMPLIED	属性是可选的
A	属性是可选的；若未指定，解析器报告的属性是 A
#FIXED A	属性必须是未指定的或者是 A；在这两种情况下，解析器报告的属性都是 A

以下是两个典型的属性规格说明：

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

第一个规格说明描述了 font 元素的 style 属性。它有 4 个合法的属性值，默认值是 plain。第二个规格说明表示 size 元素的 unit 属性可以包含任意的字符数据序列。

注释：一般情况下，我们推荐用元素而非属性来描述数据。按照这个推荐，font style 应该是一个独立的元素，例如 <style>plain</style>...。然而，对于枚举类型，属性有一个不可否认的优点，那就是解析器能够校验其取值是否合法。例如，如果 font style 是一个属性，那么解析器就会检查它是不是 4 个允许的值之一，并且如果没有为其提供属性值，那么解析器还会为其提供一个默认值。

CDATA 属性值的处理与前面看到的对 #PCDATA 的处理有着微妙的差别，并且与 <![CDATA [...]]> 部分没有多大关系。属性值首先被规范化，也就是说，解析器要先处理对字符和实体的引用（比如 é 或 <），并且要用空格来替换空白字符。

NMTOKEN（即名字标记）与 CDATA 相似，但是大多数非字母数字字符和内部的空白字符是不允许使用的，而且解析器会删除起始和结尾的空白字符。NMTOKENS 是一个以空白字符分隔的名字标记列表。

ID 结构是很有用的，ID 是在文档中必须唯一的名字标记，解析器会检查其唯一性。在下一个示例程序中，你会看到它的应用。IDREF 是对同一文档中已存在的 ID 的引用，解析器也会对它进行检查。IDREFS 是以空白字符分隔的 ID 引用的列表。

ENTITY 属性值将引用一个“未解析的外部实体”。这是从 SGML 那里沿用下来的，在实际应用中很少见到。在 <http://www.xml.com/axml/axml.html> 处的被注解的 XML 规范中有该属性的一个例子。

DTD 也可以定义实体，或者定义解析过程中被替换的缩写。你可以在 Firefox 浏览器的用户界面描述中找到一个很好的使用实体的例子。这些描述被格式化为 XML 格式，包含了如下的实体定义：

```
<!ENTITY back.label "Back">
```

其他地方的文本可以包含对这个实体的引用，例如：

```
<menuitem label=&back.label; />
```

解析器会用替代字符串来替换该实体引用。如果要对应用程序进行国际化处理，只需修

改实体定义中的字符串即可。其他的实体使用方法更加复杂，且不太常用，详细说明参见 XML 规范。

这样我们就结束了对 DTD 的介绍。既然你已经知道如何使用 DTD 了，那么你就可以配置你的解析器以充分利用它们了。首先，通知文档生成工厂打开验证特性。

```
factory.setValidating(true);
```

这样，该工厂生成的所有文档生成器都将根据 DTD 来验证它们的输入。验证的最大好处是可以忽略元素内容中的空白字符。例如，考虑下面的 XML 代码片段：

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

一个不进行验证的解析器会报告 font、name 和 size 元素之间的空白字符，因为它无法知道 font 的子元素是：

```
(name,size)
(#PCDATA,name,size)*
```

还是：

```
ANY
```

一旦 DTD 指定了子元素是 (name,size)，解析器就知道它们之间的空白字符不是文本。调用下面的代码：

```
factory.setIgnoringElementContentWhitespace(true);
```

这样，生成器将不会报告文本节点中的空白字符。这意味着，你可以依赖 font 节点拥有 2 个子元素这一事实。你再也不用编写下面这样的单调冗长的循环代码了：

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        var childElement = (Element) child;
        if (childElement.getTagName().equals("name")) . . . ;
        else if (childElement.getTagName().equals("size")) . . . ;
    }
}
```

而只需仅仅通过如下代码访问第一个和第二个子元素：

```
var nameElement = (Element) children.item(0);
var sizeElement = (Element) children.item(1);
```

这就是 DTD 如此有用的原因。你不会为了检查规则而使程序负担过重。在得到文档之前，解析器已经做完了这些工作。

当解析器报告错误时，应用程序希望对该错误执行某些操作。例如，记录到日志中，把它显示给用户，或是抛出一个异常以放弃解析。因此，只要使用验证，就应该安装一个错误

处理器，这需要提供一个实现了 ErrorHandler 接口的对象。这个接口有三个方法：

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

可以通过 DocumentBuilder 类的 setErrorHandler 方法来安装错误处理器：

```
builder.setErrorHandler(handler);
```

API **javax.xml.parsers.DocumentBuilder 1.4**

- void setEntityResolver(EntityResolver resolver)

设置解析器，来定位要解析的 XML 文档中引用的实体。

- void setErrorHandler(ErrorHandler handler)

设置用来报告在解析过程中出现的错误和警告的处理器。

API **org.xml.sax.EntityResolver 1.4**

- public InputSource resolveEntity(String publicID, String systemID)

返回一个输入源，它包含了被给定 ID 所引用的数据，或者，当解析器不知道如何解析这个特定名字时，返回 null。如果没有提供公共 ID，那么参数 publicID 可以为 null。

API **org.xml.sax.InputSource 1.4**

- InputSource(InputStream in)
- InputSource(Reader in)
- InputSource(String systemID)

从流、读入器或系统 ID (通常是相对或绝对 URL) 中构建输入源。

API **org.xml.sax.ErrorHandler 1.4**

- void fatalError(SAXParseException exception)

- void error(SAXParseException exception)

- void warning(SAXParseException exception)

覆盖这些方法以提供对致命错误、非致命错误和警告进行处理的处理器。

API **org.xml.sax.SAXParseException 1.4**

- int getLineNumber()

- int getColumnNumber()

返回引起异常的已处理的输入信息末尾的行号和列号。

API **javax.xml.catalog.CatalogManager 9**

- static CatalogResolver catalogResolver(CatalogFeatures features, URI... uris)

产生一个解析器，它将使用由所提供的 URI 指定的位置上的目录文件。这个类实现了 EntityResolver 接口，StAX、Schema 校验和 XSL 转换用到的类也实现了该接口。

API javax.xml.catalog.CatalogFeatures 9

- static CatalogFeatures defaults()

用默认设置产生一个实例。

API javax.xml.parsers.DocumentBuilderFactory 1.4

- boolean isValidation()

- void setValidation(boolean value)

获取和设置工厂的 validation 属性。当它设为 true 时，该工厂生成的解析器会验证它们的输入信息。

- boolean isIgnoringElementContentWhitespace()

- void setIgnoringElementContentWhitespace(boolean value)

获取和设置工厂的 ignoringElementContentWhitespace 属性。当它设为 true 时，该工厂生成的解析器会忽略不含混合内容（即，元素与 #PCDATA 混合）的元素节点之间的空白字符。

3.4.2 XML Schema

因为 XML Schema 比起 DTD 语法要复杂许多，所以我们只涉及其基本知识。更多信息请参考 <http://www.w3.org/TR/xmlschema-0> 上的指南。

如果要在文档中引用 Schema 文件，需要在根元素中添加属性，例如：

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
    ...
</config>
```

这个声明说明 Schema 文件 config.xsd 会被用来验证该文档。如果使用命名空间，语法就更加复杂了。详情请参见 XML Schema 指南（前缀 xsi 是一个命名空间别名（namespace alias），请查看 3.6 节以了解更多信息）。

Schema 为每个元素和属性都定义了类型。类型中的简单类型是对内容有限制的字符串，其他都是复杂类型。具有简单类型的元素可以没有任何属性和子元素。否则，它就必然是复杂类型。与此相反，属性总是简单类型。

一些简单类型已经被内建到了 XML Schema 内，包括：

```
xsd:string
xsd:int
xsd:boolean
```

注释：我们用前缀 xsd: 来表示 XSL Schema 定义的命名空间。一些作者代之以 xs:。

可以定义自己的简单类型。例如，下面是一个枚举类型：

```
<xsd:simpleType name="StyleType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="PLAIN" />
```

```

<xsd:enumeration value="BOLD" />
<xsd:enumeration value="ITALIC" />
<xsd:enumeration value="BOLD_ITALIC" />
</xsd:restriction>
</xsd:simpleType>

```

当定义元素时，要指定它的类型：

```

<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>

```

类型约束了元素的内容。例如，下面的元素将被验证为具有正确格式：

```

<size>10</size>
<style>PLAIN</style>

```

但是，下面的元素会被解析器拒绝：

```

<size>default</size>
<style>SLANTED</style>

```

可以把类型组合成复杂类型，例如：

```

<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element ref="name"/>
    <xsd:element ref="size"/>
    <xsd:element ref="style"/>
  </xsd:sequence>
</xsd:complexType>

```

FontType 是 name、size 和 style 元素的序列。在这个类型定义中，我们使用了 ref 属性来引用在 Schema 中位于别处的定义。也可以嵌套定义，像这样：

```

<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

```

请注意 style 元素的匿名类型定义。

xsd:sequence 结构和 DTD 中的连接符号等价，而 xsd:choice 结构和 | 操作符等价，例如：

```

<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>

```

```

<xsd:element ref="phone"/>
</xsd:choice>
</xsd:complexType>
```

这和 DTD 中的类型 `email|phone` 类型是等价的。

如果要允许重复元素，可以使用 `minoccurs` 和 `maxoccurs` 属性，例如，与 DTD 类型 `item*` 等价的形式如下：

```
<xsd:element name="item" type="..." minoccurs="0" maxoccurs="unbounded">
```

如果要指定属性，可以把 `xsd:attribute` 元素添加到 `complexType` 定义中去：

```

<xsd:element name="size">
  <xsd:complexType>
    .
    .
    <xsd:attribute name="unit" type="xsd:string" use="optional" default="cm"/>
  </xsd:complexType>
</xsd:element>
```

这与下面的 DTD 语句等价：

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

可以把 Schema 的元素和类型定义封装在 `xsd:schema` 元素中：

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  .
  .
</xsd:schema>
```

解析带有 Schema 的 XML 文件和解析带有 DTD 的文件相似，但有 2 点差别：

- 必须打开对命名空间的支持，即使在 XML 文件里可能不会用到它。

```
factory.setNamespaceAware(true);
```

- 必须通过如下的“魔咒”来准备好处理 Schema 的工厂。

```

final String JAXP_SCHEMA_LANGUAGE =
  "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

3.4.3 一个实践示例

在本节中，我们将要介绍一个实用的示例程序，用来说明在实际环境中 XML 的用法。

假设有一个应用程序需要配置数据，这些数据可以指定任意对象，而不仅是文本字符串。我们提供了两种机制来实例化对象：使用构造器和使用工厂方法。下面展示了如何使用构造器来创建 `Color` 对象：

```

<construct class="java.awt.Color">
  <int>55</int>
  <int>200</int>
  <int>100</int>
</construct>
```

下面是使用工厂方法的例子：

```
<factory class="java.util.logging.Logger" method="getLogger">
    <string>com.horstmann.corejava</string>
</factory>
```

如果忽略工厂方法名，那么其默认值就是 `getInstance`。

正如你所见，有多个元素用来描述字符串和整数。我们还支持 `boolean` 类型，其他基本类型也都可以按照相同的方式添加进来。

只是为了显摆一下，我们给出了第二种针对基本类型的机制：

```
<value type="int">30</value>
```

配置是由多个项构成的序列。每一项都有一个 ID 和一个对象：

```
<config>
    <entry id="background">
        <construct class="java.awt.Color">
            <value type="int">55</value>
            <value type="int">200</value>
            <value type="int">100</value>
        </construct>
    </entry>
    .
    .
</config>
```

解析器会检查这些 ID 是否唯一。

DTD 显示在程序清单 3-4 中，很简单。

程序清单 3-5 包含了一个等价的 Schema。在这个 Schema 中，我们可以提供额外的检查：一个 `int` 或 `boolean` 元素只能包含整数或布尔值。注意，这里使用了 `xsd:group` 结构来定义会反复使用的复杂类型的各个部件。

程序清单 3-2 中的程序展示了如何解析配置文件。程序清单 3-3 中定义配置样例。

如果选择了包含字符串 -Schema 的文件，那么该程序使用 Schema 而不是 DTD。

如果选择了包含字符串 -Schema 的文件，那么该程序除了 DTD，还可以处理 Schema。

这个例子是 XML 的典型用法。XML 格式十分健壮，足以表达复杂的关系。在此基础上，通过接管有效性检查和提供默认值等例行工作，XML 解析器添加了新的价值。

程序清单 3-2 read/XML ReadTest.java

```

1 package read;
2
3 import java.io.*;
4 import java.lang.reflect.*;
5 import java.util.*;
6
7 import javax.xml.parsers.*;
8
9 import org.w3c.dom.*;
10 import org.xml.sax.*;
11
12 /**
13 * This program shows how to use an XML file to describe Java objects

```

```
14 * @version 1.0 2018-04-03
15 * @author Cay Horstmann
16 */
17 public class XMLReadTest
18 {
19     public static void main(String[] args) throws ParserConfigurationException,
20             SAXException, IOException, ReflectiveOperationException
21     {
22         String filename;
23         if (args.length == 0)
24         {
25             try (var in = new Scanner(System.in))
26             {
27                 System.out.print("Input file: ");
28                 filename = in.nextLine();
29             }
30         }
31         else
32             filename = args[0];
33
34         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
35         factory.setValidating(true);
36
37         if (filename.contains("-schema"))
38         {
39             factory.setNamespaceAware(true);
40             final String JAXP_SCHEMA_LANGUAGE =
41                 "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
42             final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
43             factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
44         }
45
46         factory.setIgnoringElementContentWhitespace(true);
47
48         DocumentBuilder builder = factory.newDocumentBuilder();
49
50         builder.setErrorHandler(new ErrorHandler()
51         {
52             public void warning(SAXParseException e) throws SAXException
53             {
54                 System.err.println("Warning: " + e.getMessage());
55             }
56
57             public void error(SAXParseException e) throws SAXException
58             {
59                 System.err.println("Error: " + e.getMessage());
60                 System.exit(0);
61             }
62
63             public void fatalError(SAXParseException e) throws SAXException
64             {
65                 System.err.println("Fatal error: " + e.getMessage());
66                 System.exit(0);
67             }
68         });
69     }
70 }
```

```

68     });
69
70     Document doc = builder.parse(filename);
71     Map<String, Object> config = parseConfig(doc.getDocumentElement());
72     System.out.println(config);
73 }
74
75 private static Map<String, Object> parseConfig(Element e)
76     throws ReflectiveOperationException
77 {
78     var result = new HashMap<String, Object>();
79     NodeList children = e.getChildNodes();
80     for (int i = 0; i < children.getLength(); i++)
81     {
82         var child = (Element) children.item(i);
83         String name = child.getAttribute("id");
84         Object value = parseObject((Element) child.getFirstChild());
85         result.put(name, value);
86     }
87     return result;
88 }
89
90 private static Object parseObject(Element e)
91     throws ReflectiveOperationException
92 {
93     String tagName = e.getTagName();
94     if (tagName.equals("factory")) return parseFactory(e);
95     else if (tagName.equals("construct")) return parseConstruct(e);
96     else
97     {
98         String childData = ((CharacterData) e.getFirstChild()).getData();
99         if (tagName.equals("int"))
100             return Integer.valueOf(childData);
101         else if (tagName.equals("boolean"))
102             return Boolean.valueOf(childData);
103         else
104             return childData;
105     }
106 }
107
108 private static Object parseFactory(Element e)
109     throws ReflectiveOperationException
110 {
111     String className = e.getAttribute("class");
112     String methodName = e.getAttribute("method");
113     Object[] args = parseArgs(e.getChildNodes());
114     Class<?>[] parameterTypes = getParameterTypes(args);
115     Method method = Class.forName(className).getMethod(methodName, parameterTypes);
116     return method.invoke(null, args);
117 }
118
119 private static Object parseConstruct(Element e)
120     throws ReflectiveOperationException
121 {

```

```

122     String className = e.getAttribute("class");
123     Object[] args = parseArgs(e.getChildNodes());
124     Class<?>[] parameterTypes = getParameterTypes(args);
125     Constructor<?> constructor = Class.forName(className).getConstructor(parameterTypes);
126     return constructor.newInstance(args);
127 }
128
129 private static Object[] parseArgs(NodeList elements)
130     throws ReflectiveOperationException
131 {
132     var result = new Object[elements.getLength()];
133     for (int i = 0; i < result.length; i++)
134         result[i] = parseObject((Element) elements.item(i));
135     return result;
136 }
137
138 private static Map<Class<?>, Class<?>> toPrimitive = Map.of(
139     Integer.class, int.class,
140     Boolean.class, boolean.class);
141
142 private static Class<?>[] getParameterTypes(Object[] args)
143 {
144     var result = new Class<?>[args.length];
145     for (int i = 0; i < result.length; i++)
146     {
147         Class<?> cl = args[i].getClass();
148         result[i] = toPrimitive.get(cl);
149         if (result[i] == null) result[i] = cl;
150     }
151     return result;
152 }
153 }

```

程序清单 3-3 read/config.xml

```

1 <?xml version="1.0"?>
2 <!DOCTYPE config SYSTEM "config.dtd">
3 <config>
4   <entry id="background">
5     <construct class="java.awt.Color">
6       <int>55</int>
7       <int>200</int>
8       <int>100</int>
9     </construct>
10   </entry>
11   <entry id="currency">
12     <factory class="java.util.Currency">
13       <string>USD</string>
14     </factory>
15   </entry>
16 </config>

```

程序清单 3-4 read/config.dtd

```

1 <!ELEMENT config (entry)*>
2
3 <!ELEMENT entry (string|int|boolean|construct|factory)>
4 <!ATTLIST entry id ID #IMPLIED>
5
6 <!ELEMENT construct (string|int|boolean|construct|factory)*>
7 <!ATTLIST construct class CDATA #IMPLIED>
8
9 <!ELEMENT factory (string|int|boolean|construct|factory)*>
10 <!ATTLIST factory class CDATA #IMPLIED>
11 <!ATTLIST factory method CDATA "getInstance">
12
13 <!ELEMENT string (#PCDATA)>
14 <!ELEMENT int (#PCDATA)>
15 <!ELEMENT boolean (#PCDATA)>
```

程序清单 3-5 read/config.xsd

```

1 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2   <xsd:element name="config">
3     <xsd:complexType>
4       <xsd:sequence>
5         <xsd:element name="entry" minOccurs="0" maxOccurs="unbounded">
6           <xsd:complexType>
7             <xsd:group ref="Object"/>
8             <xsd:attribute name="id" type="xsd:ID"/>
9           </xsd:complexType>
10          </xsd:element>
11        </xsd:sequence>
12      </xsd:complexType>
13    </xsd:element>
14
15   <xsd:element name="construct">
16     <xsd:complexType>
17       <xsd:group ref="Arguments"/>
18       <xsd:attribute name="class" type="xsd:string"/>
19     </xsd:complexType>
20   </xsd:element>
21
22   <xsd:element name="factory">
23     <xsd:complexType>
24       <xsd:group ref="Arguments"/>
25       <xsd:attribute name="class" type="xsd:string"/>
26       <xsd:attribute name="method" type="xsd:string" default="getInstance"/>
27     </xsd:complexType>
28   </xsd:element>
29
30   <xsd:group name="Object">
31     <xsd:choice>
32       <xsd:element ref="construct"/>
33       <xsd:element ref="factory"/>
```

```

34      <xsd:element name="string" type="xsd:string"/>
35      <xsd:element name="int" type="xsd:int"/>
36      <xsd:element name="boolean" type="xsd:boolean"/>
37  </xsd:choice>
38 </xsd:group>
39
40 <xsd:group name="Arguments">
41   <xsd:sequence>
42     <xsd:group ref="Object" minOccurs="0" maxOccurs="unbounded"/>
43   </xsd:sequence>
44 </xsd:group>
45 </xsd:schema>

```

3.5 使用 XPath 来定位信息

如果要定位某个 XML 文档中的一段特定信息，那么，通过遍历 DOM 树的众多节点来进行查找会显得有些麻烦。XPath 语言使得访问树节点变得很容易。例如，假设有如下 HTML 文档：

```

<html>
  <head>
    ...
    <title>...</title>
    ...
  </head>
  ...
</html>

```

可以通过对 XPath 表达式 `/html/head/title/text()` 求值来得到标题的文本。

使用 Xpath 执行下列操作比普通的 DOM 方式要简单得多：

1. 获得文档根节点。
2. 获取第一个子节点，并将其转型为一个 Element 对象。
3. 在其所有子节点中定位 title 元素。
4. 获取其第一个子元素，并将其转型为一个 CharacterData 节点。
5. 获取其数据。

XPath 可以描述 XML 文档中的一个节点集，例如，下面的 XPath：

`/html/body/form`

描述了 XHTML 文件中 body 元素的子元素中所有的 form 元素。可以用 [] 操作符来选择特定元素：

`/html/body/form[1]`

这表示的是第一个 form (索引号从 1 开始)。

使用 @ 操作符可以得到属性值。XPath 表达式

`/html/body/form[1]/@action`

描述了第一个表中的 action 属性。XPath 表达式

```
/html/body/form/@action
```

描述了 body 元素的子元素中所有 form 元素的所有 action 属性节点。

XPath 有很多有用的函数，例如：

```
count(/html/body/form)
```

返回 body 根元素的 form 子元素的数量。精细的 XPath 表达式还有很多，请参见 <http://www.w3c.org/TR/xpath> 的规范，或者在 <http://www.zvon.org/xxl/XPathTutorial/General/examples.html> 上的一个非常好的在线指南。

要计算 XPath 表达式，首先需要从 XPathFactory 创建一个 XPath 对象：

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

然后，调用 evaluate 方法来计算 XPath 表达式：

```
String username = path.evaluate("/html/head/title/text()", doc);
```

可以用同一个 XPath 对象来计算多个表达式。

这种形式的 evaluate 方法将返回一个字符串。这很适合用来获取文本，比如前面的例子中的 title 元素的文本子节点。如果 XPath 表达式产生了一组节点，请做如下调用：

```
XPathNodes result = path.evaluateExpression("/html/body/form", doc, XPathNodes.class);
```

XPathNodes 类与 NodeList 类相似，但是它扩展了 Iterable 接口，使得我们可以使用增强型 for 循环。

这个方法是在 Java 9 中添加进来的，在老版本中，需要使用下面这条语句：

```
var nodes = (NodeList) path.evaluate("/html/body/form", doc, XPathConstants.NODESET);
```

如果结果只有一个节点，则使用下面的调用：

```
Node node = path.evaluateExpression("/html/body/form[1]", doc, Node.class);
node = (Node) path.evaluate("/html/body/form[1]", doc, XPathConstants.NODE);
```

如果结果是一个数字，则使用：

```
int count = path.evaluateExpression("count(/html/body/form)", doc, Integer.class);
count = ((Number) path.evaluate("count(/html/body/form)",
    doc, XPathConstants.NUMBER)).intValue();
```

不必从文档的根节点开始搜索，可以从任意一个节点或节点列表开始。例如，如果有前一次计算得到的节点，那么就可以调用：

```
String result = path.evaluate(expression, node);
```

如果不知道 XPath 表达式的计算结果是什么（可能该表达式来自于用户），那么就调用

```
XPathEvaluationResult<?> result = path.evaluateExpression(expression, doc);
```

表达式 result.type() 是下列 XPathEvaluationResult.XPathResultType 枚举常量之一：

STRING

NODESET
NODE
NUMBER
BOOLEAN

调用 `result.value()` 可以获取结果值。

程序清单 3-6 展示了对任意的 XPath 表达式的计算过程。加载一个 XML 文件，输入一个表达式，该表达式的结果就会显示出来。

程序清单 3-6 xpath/XPathTest.java

```

1 package xpath;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6
7 import javax.xml.catalog.*;
8 import javax.xml.parsers.*;
9 import javax.xml.xpath.*;
10
11 import org.w3c.dom.*;
12 import org.xml.sax.*;
13
14 /**
15  * This program evaluates XPath expressions.
16  * @version 1.1 2018-04-06
17  * @author Cay Horstmann
18  */
19 public class XPathTest
20 {
21     public static void main(String[] args) throws Exception
22     {
23         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
24         DocumentBuilder builder = factory.newDocumentBuilder();
25
26         // Avoid a delay in parsing an XHTML file--see the first note in
27         // Section 3.3.1
28         builder.setEntityResolver(CatalogManager.catalogResolver(
29             CatalogFeatures.defaults(),
30             Paths.get("xpath/catalog.xml").toAbsolutePath().toUri()));
31
32         XPathFactory xpfactory = XPathFactory.newInstance();
33         XPath path = xpfactory.newXPath();
34         try (var in = new Scanner(System.in))
35         {
36             String filename;
37             if (args.length == 0)
38             {
39                 System.out.print("Input file: ");
40                 filename = in.nextLine();
41             }
42             else
43                 filename = args[0];

```

```

44
45     Document doc = builder.parse(filename);
46     var done = false;
47     while (!done)
48     {
49         System.out.print("XPath expression (empty line to exit): " );
50         String expression = in.nextLine();
51         if (expression.trim().isEmpty()) done = true;
52         else
53         {
54             try
55             {
56                 XPathEvaluationResult<?> result
57                     = path.evaluateExpression(expression, doc);
58                 if (result.type() == XPathEvaluationResult.XPathResultType.NODESET)
59                 {
60                     for (Node n : (XPathNodes) result.value())
61                         System.out.println(description(n));
62                 }
63                 else if (result.type() == XPathEvaluationResult.XPathResultType.NODESET)
64                     System.out.println((Node) result.value());
65                 else
66                     System.out.println(result.value());
67             }
68             catch (XPathExpressionException e)
69             {
70                 System.out.println(e.getMessage());
71             }
72         }
73     }
74 }
75
76     public static String description(Node n)
77     {
78         if (n instanceof Element) return "Element " + n.getNodeName();
79         else if (n instanceof Attr) return "Attribute " + n;
80         else return n.toString();
81     }
82 }
83 }
```

API **javax.xml.xpath.XPathFactory 5.0**

- **static XPathFactory newInstance()**
返回用于创建 XPath 对象的 XPathFactory 实例。
- **XPath newPath()**
构建用于计算 XPath 表达式的 XPath 对象。

API **javax.xml.xpath.XPath 5.0**

- **String evaluate(String expression, Object startingPoint)**
从给定的起点计算表达式。起点可以是一个节点或节点列表。如果结果是一个节点或

节点集，则返回的字符串由所有文本节点子元素的数据构成。

- `Object evaluate(String expression, Object startingPoint, QName resultType)`

从给定的起点计算表达式。起点可以是一个节点或节点列表。`resultType` 是 `XPathConstants` 类的常量 `STRING`、`NODE`、`NODESET`、`NUMBER` 或 `BOOLEAN` 之一。返回值是 `String`、`Node`、`NodeList`、`Number` 或 `Boolean`。

- `<T> T evaluateExpression(String expression, Object item, Class<T> type) 9`

计算给定表达式，并产生给定类型值的结果。

- `XPathEvaluationResult<?> evaluateExpression(String expression, InputSource source) 9`

计算给定表达式。

API `javax.xml.xpath.XPathEvaluationResult<T> 9`

- `XPathEvaluationResult.XPathResultType type()`

返回枚举常量 `STRING`、`NODESET`、`NODE`、`NUMBER` 和 `BOOLEAN` 之一。

- `T value()`

返回结果值。

3.6 使用命名空间

Java 语言使用包来避免名字冲突。程序员可以为不同的类使用相同的名字，只要它们不在同一个包中即可。XML 也有类似的命名空间（namespace）机制，可以用于元素名和属性名。

名字空间是由统一资源标识符（Uniform Resource Identifier，URI）来标识的，比如：

```
http://www.w3.org/2001/XMLSchema
uuid:1c759aed-b748-475c-ab68-10679700c4f2
urn:com:books-r-us
```

HTTP 的 URL 格式是最常见的标识符。注意，URL 只用作标识符字符串，而不是一个文件的定位符。例如，名字空间标识符：

```
http://www.horstmann.com/corejava
http://www.horstmann.com/corejava/index.html
```

表示了不同的命名空间，尽管 Web 服务器将为这两个 URL 提供同一个文档。

在命名空间的 URL 所表示的位置上不需要有任何文档，XML 解析器不会尝试去该处查找任何东西。然而，为了给可能会遇到不熟悉的命名空间的程序员提供一些帮助，人们习惯于将解释该命名空间的文档放在 URL 位置上。例如，如果把浏览器指向 XML Schema 的命名空间 URL (`http://www.w3.org/2001/XMLSchema`)，就会发现一个描述 XML Schema 标准的文档。

为什么要用 HTTP URL 作为命名空间的标识符？这是因为这样容易确保它们是独一无二的。如果使用实际的 URL，那么主机部分的唯一性就将由域名系统来保证。然后，你的组织可以安排 URL 余下部分的唯一性，这和 Java 包名中的反向域名是一个原理。

尽管长名字空间的唯一性很好，但是你肯定不想处理超出必需范围的长标识符。在 Java

编程语言中，可以用 `import` 机制来指定很长的包名，然后就可以只使用较短的类名了。在 XML 中有类似的机制，比如：

```
<element xmlns="namespaceURI">
    children
</element>
```

现在，该元素和它的子元素都是给定命名空间的一部分了。

子元素可以提供自己的命名空间，例如：

```
<element xmlns="namespaceURI1">
    <child xmlns="namespaceURI2">
        grandchildren
    </child>
    more children
</element>
```

这时，第一个子元素和孙元素都是第二个命名空间的一部分。

无论是只需要一个命名空间，还是命名空间本质上是嵌套的，这个简单机制都工作得很好。如若不然，就需要使用第二种机制，而 Java 中并没有类似的机制。你可以用一个前缀来表示命名空间，即为特定文档选取的一个短的标识符。下面是一个典型的例子：

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="config"/>
    ...
</xsd:schema>
```

下面的属性：

```
xmlns:prefix="namespaceURI"
```

用于定义命名空间和前缀。在我们的例子中，前缀是字符串 `xsd`。这样，`xsd:schema` 实际上指的是命名空间 `http://www.w3.org/2001/XMLSchema` 中的 `schema`。

 **注释：**只有子元素继承了它们父元素的命名空间，而不带显式前缀的属性并不是命名空间的一部分。请看下面这个特意构造出来的例子：

```
<configuration xmlns="http://www.horstmann.com/corejava"
    xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
    <size value="210" si:unit="mm"/>
    ...
</configuration>
```

在这个示例中，元素 `configuration` 和 `size` 是 URI 为 `http://www.horstmann.com/corejava` 的命名空间的一部分。属性 `si:unit` 是 URI 为 `http://www.bipm.fr/enus/3_SI/si.html` 的命名空间的一部分。然而，属性 `value` 不是任何命名空间的一部分。

可以控制解析器对命名空间的处理。默认情况下，Java XML 库的 DOM 解析器并非“命名空间感知的”。

要打开命名空间处理特性，请调用 `DocumentBuilderFactory` 类的 `setNamespaceAware` 方法：

```
factory.setNamespaceAware(true);
```

这样，该工厂产生的所有生成器便都支持命名空间了。每个节点有三个属性：

- 带有前缀的限定名 (qualified)，由 `getNodeName` 和 `getTagName` 等方法返回。
- 命名空间 URI，由 `getNamespaceURI` 方法返回。
- 不带前缀和命名空间的本地名 (local name)，由 `getLocalName` 方法返回。

下面是一个例子。假设解析器看到了以下元素：

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

它会报告如下信息：

- 限定名 = `xsd:schema`
- 命名空间 URI = `http://www.w3.org/2001/XMLSchema`
- 本地名 = `schema`

注释：如果对命名空间的感知特性被关闭，`getLocalName` 和 `getNamespaceURI` 方法将返回 `null`。

API `org.w3c.dom.Node 1.4`

- `String getLocalName()`
返回本地名 (不带前缀)，或者在解析器不感知命名空间时，返回 `null`。
- `String getNamespaceURI()`
返回命名空间 URI，或者在解析器不感知命名空间时，返回 `null`。

API `javax.xml.parsers.DocumentBuilderFactory 1.4`

- `boolean isNamespaceAware()`
- `void setNamespaceAware(boolean value)`

获取或设置工厂的 `namespaceAware` 属性。当设为 `true` 时，工厂产生的解析器是命名空间感知的。

3.7 流机制解析器

DOM 解析器会完整地读入 XML 文档，然后将其转换成一个树形的数据结构。对于大多数应用，DOM 都运行得很好。但是，如果文档很大，并且处理算法又非常简单，可以在运行时解析节点，而不必看到完整的树形结构，那么 DOM 可能就会显得效率低下了。在这种情况下，我们应该使用流机制解析器 (streaming parser)。

在下面的小节中，我们将讨论 Java 类库提供的流机制解析器：老而弥坚的 SAX 解析器和添加到 Java 6 中的更现代化的 StAX 解析器。SAX 解析器使用的是事件回调 (event callback)，而 StAX 解析器提供了遍历解析事件的迭代器，后者用起来通常更方便一些。

3.7.1 使用 SAX 解析器

SAX 解析器在解析 XML 输入数据的各个组成部分时会报告事件，但不会以任何方式存

储文档，而是由事件处理器建立相应的数据结构。实际上，DOM 解析器是在 SAX 解析器的基础上构建的，它在接收到解析器事件时构建 DOM 树。

在使用 SAX 解析器时，需要一个处理器来为各种解析器事件定义事件动作。ContentHandler 接口定义了若干个在解析文档时解析器会调用的回调方法。下面是最主要的几个：

- startElement 和 endElement 在每当遇到起始或终止标签时调用。
- characters 在每当遇到字符数据时调用。
- startDocument 和 endDocument 分别在文档开始和结束时各调用一次。

例如，在解析以下片段时：

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

解析器会产生以下回调：

1. startElement, 元素名: font
2. startElement, 元素名: name
3. characters, 内容: Helvetica
4. endElement, 元素名: name
5. startElement, 元素名: size, 属性: units="pt"
6. characters, 内容: 36
7. endElement, 元素名: size
8. endElement, 元素名: font

处理器必须覆盖这些方法，让它们执行在解析文件时我们想要让它们执行的动作。本节最后的程序会打印出一个 HTML 文件中的所有链接 ``。它直接覆盖了处理器的 `startElement` 方法，以检查名字为 `a`，且属性名为 `href` 的链接，其潜在用途包括用于实现“网络爬虫”，即一个沿着链接到达越来越多网页的程序。

注释：遗憾的是，HTML 不必是合法的 XML，大多数 HTML 页面都与良构的 XML 差别很大，以至于示例程序无法解析它们。但是，W3C 编写的大部分页面都是用 XHTML 编写的，XHTML 是一种 HTML 方言，且是良构的 XML，你可以用这些页面来测试示例程序。例如，运行：

```
java SAXTest http://www.w3c.org/MarkUp
```

将看到那个页面上所有链接的 URL 列表。

示例程序是一个很好的使用 SAX 的例子。我们根本不在乎 `a` 元素出现的上下文环境，而且不必存储树形结构。

下面是如何得到 SAX 解析器的代码：

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

现在可以处理文档了：

```
parser.parse(source, handler);
```

这里的 source 可以是一个文件、一个 URL 字符串或者是一个输入流。handler 属于 DefaultHandler 的一个子类，DefaultHandler 类为以下四个接口定义了空的方法：

```
ContentHandler  
DTDHandler  
EntityResolver  
ErrorHandler
```

示例程序定义了一个处理器，它覆盖了 ContentHandler 接口的 startElement 方法，以观察带有 href 属性的 a 元素。

```
var handler = new DefaultHandler()  
{  
    public void startElement(String namespaceURI, String lname, String qname,  
        Attributes attrs) throws SAXException  
    {  
        if (lname.equalsIgnoreCase("a") && attrs != null)  
        {  
            for (int i = 0; i < attrs.getLength(); i++)  
            {  
                String aname = attrs.getLocalName(i);  
                if (aname.equalsIgnoreCase("href"))  
                    System.out.println(attrs.getValue(i));  
            }  
        }  
    }  
};
```

startElement 方法有 3 个描述元素名的参数，其中 qname 参数以 prefix:localname 的形式报告限定名。如果命名空间处理特性已经打开，那么 namespaceURI 和 lname 参数提供的就是命名空间和本地（非限定）名。

与 DOM 解析器一样，命名空间处理特性默认是关闭的，可以调用工厂类的 setNamespaceAware 方法来激活命名空间处理特性：

```
SAXParserFactory factory = SAXParserFactory.newInstance();  
factory.setNamespaceAware(true);  
SAXParser saxParser = factory.newSAXParser();
```

在这个程序中，我们还处理了另一个常见的问题。XHTML 文件总是以一个包含对 DTD 引用的标签开头，解析器会加载这个 DTD。可以理解的是，W3C 肯定不乐意对诸如 www.w3.org/TR/xhtml/DTD/xhtml Strict.dtd 这样的文件提供千万亿次的下载。总有一天他们会完全拒绝提供这些文件，但到写本章时为止，他们还在并不情愿地提供 DTD 下载。如果你不需要验证文件，只需调用：

```
factory.setFeature("http://apache.org/xml/features/nonvalidating/load-external-dtd", false);
```

程序清单 3-7 包含了网络爬虫程序的代码。在本章的后续部分，将会看到 SAX 的另一个有趣用法，即将非 XML 数据源转换成 XML 的一种简单方式是报告 XML 解析器将要报告的

SAX事件。详情请参见3.9节。

程序清单3-7 sax/SAXTest.java

```

1 package sax;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.xml.parsers.*;
6 import org.xml.sax.*;
7 import org.xml.sax.helpers.*;
8
9 /**
10  * This program demonstrates how to use a SAX parser. The program prints all
11  * hyperlinks of an XHTML web page. <br>
12  * Usage: java sax.SAXTest URL
13  * @version 1.01 2018-05-01
14  * @author Cay Horstmann
15  */
16 public class SAXTest
17 {
18     public static void main(String[] args) throws Exception
19     {
20         String url;
21         if (args.length == 0)
22         {
23             url = "http://www.w3c.org";
24             System.out.println("Using " + url);
25         }
26         else url = args[0];
27
28         var handler = new DefaultHandler()
29         {
30             public void startElement(String namespaceURI, String lname,
31                         String qname, Attributes attrs)
32             {
33                 if (lname.equals("a") && attrs != null)
34                 {
35                     for (int i = 0; i < attrs.getLength(); i++)
36                     {
37                         String aname = attrs.getLocalName(i);
38                         if (aname.equals("href"))
39                             System.out.println(attrs.getValue(i));
40                     }
41                 }
42             }
43         };
44
45         SAXParserFactory factory = SAXParserFactory.newInstance();
46         factory.setNamespaceAware(true);
47         factory.setFeature(
48             "http://apache.org/xml/features/nonvalidating/load-external-dtd",
49             false);
50         SAXParser saxParser = factory.newSAXParser();
51         InputStream in = new URL(url).openStream();

```

```

52     saxParser.parse(in, handler);
53 }
54 }
```

API **javax.xml.parsers.SAXParserFactory 1.4**

- **static SAXParserFactory newInstance()**
返回 SAXParserFactory 类的一个实例。
- **SAXParser newSAXParser()**
返回 SAXParser 类的一个实例。
- **boolean isNamespaceAware()**
- **void setNamespaceAware(boolean value)**
获取和设置工厂的 namespaceAware 属性。当设为 true 时，该工厂生成的解析器是命名空间感知的。
- **boolean isvalidating()**
- **void setValidating(boolean value)**
获取和设置工厂的 validating 属性。当设为 true 时，该工厂生成的解析器将要验证其输入。

API **javax.xml.parsers.SAXParser 1.4**

- **void parse(File f, DefaultHandler handler)**
- **void parse(String url, DefaultHandler handler)**
- **void parse(InputStream in, DefaultHandler handler)**
解析来自给定文件、URL 或输入流的 XML 文档，并把解析事件报告给指定的处理器。

API **org.xml.sax.ContentHandler 1.4**

- **void startDocument()**
- **void endDocument()**
在文档的开头和结尾处被调用。
- **void startElement(String uri, String lname, String qname, Attributes attr)**
- **void endElement(String uri, String lname, String qname)**
在元素的开头和结尾处被调用。如果解析器是名字空间感知的，那么它会报告名字空间的 URI、无前缀的本地名字，以及带前缀的限定名。
- **void characters(char[] data, int start, int length)**
解析器报告字符数据时被调用。

API **org.xml.sax.Attributes 1.4**

- **int getLength()**

返回存储在该属性集合中的属性数量。

- `String getLocalName(int index)`

返回给定索引的属性的本地名（无前缀），或在解析器不是命名空间感知的情况下返回空字符串。

- `String getURI(int index)`

返回给定索引的属性的命名空间 URI，或者，当该节点不是命名空间的一部分，或解析器并非命名空间感知时返回空字符串。

- `String getQName(int index)`

返回给定索引的属性的限定名（带前缀），或当解析器不报告限定名时返回空字符串。

- `String getValue(int index)`

- `String getValue(String qname)`

- `String getValue(String uri, String lname)`

根据给定索引、限定名或命名空间 URI+ 本地名来返回属性值；当该值不存在时，返回 `null`。

3.7.2 使用 StAX 解析器

StAX 解析器是一种“拉解析器”（pull parser），与安装事件处理器不同，你只需使用下面这样的基本循环来迭代所有的事情：

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
    int event = parser.next();
    Call parser methods to obtain event details
}
```

例如，在解析下面的片段时

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

解析器将产生下面的事件：

1. `START_ELEMENT`, 元素名: `font`
2. `CHARACTERS`, 内容: 空白字符
3. `START_ELEMENT`, 元素名: `name`
4. `CHARACTERS`, 内容: `Helvetica`
5. `END_ELEMENT`, 元素名: `name`
6. `CHARACTERS`, 内容: 空白字符
7. `START_ELEMENT`, 元素名: `size`

8. CHARACTERS, 内容: 36
9. END_ELEMENT, 元素名: size
10. CHARACTERS, 内容: 空白字符
11. END_ELEMENT, 元素名: font

要分析这些属性值, 需要调用 `XMLStreamReader` 类中恰当的方法, 例如:

```
String units = parser.getAttributeValue(null, "units");
```

它 can 获取当前元素的 `units` 属性。

默认情况下, 命名空间处理是启用的, 可以通过像下面这样修改工厂来使其无效:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

程序清单 3-8 包含了用 StAX 解析器实现的网络爬虫程序。正如你所见, 这段代码比等效的 SAX 代码要简短了许多, 因为此时我们不必操心事件处理问题。

程序清单 3-8 stax/StAXTest.java

```
1 package stax;
2
3 import java.io.*;
4 import java.net.*;
5 import javax.xml.stream.*;
6
7 /**
8  * This program demonstrates how to use a StAX parser. The program prints all
9  * hyperlinks links of an XHTML web page. <br>
10 * Usage: java stax.StAXTest URL
11 * @author Cay Horstmann
12 * @version 1.1 2018-05-01
13 */
14 public class StAXTest
15 {
16     public static void main(String[] args) throws Exception
17     {
18         String urlString;
19         if (args.length == 0)
20         {
21             urlString = "http://www.w3c.org";
22             System.out.println("Using " + urlString);
23         }
24         else urlString = args[0];
25         var url = new URL(urlString);
26         InputStream in = url.openStream();
27         XMLInputFactory factory = XMLInputFactory.newInstance();
28         XMLStreamReader parser = factory.createXMLStreamReader(in);
29         while (parser.hasNext())
30         {
31             int event = parser.next();
32             if (event == XMLStreamConstants.START_ELEMENT)
33             {
34                 if (parser.getLocalName().equals("a"))
```

```

35         {
36             String href = parser.getAttributeValue(null, "href");
37             if (href != null)
38                 System.out.println(href);
39         }
40     }
41 }
42 }
43 }

```

API **javax.xml.stream.XMLInputFactory** 6

- static XMLInputFactory newInstance()
- 返回 XMLInputFactory 类的一个实例。
- void setProperty(String name, Object value)

设置这个工厂的属性，或者在要设置的属性不支持设置成给定值时，抛出 IllegalArgumentException。JDK 的实现支持下列 Boolean 类型的属性：

"javax.xml.stream.isValidating"	为 false (默认值) 时，不验证文档 (规范不要求必须支持)。
"javax.xml.stream.isNamespaceAware"	为 true (默认值) 时，将处理命名空间 (规范不要求必须支持)。
"javax.xml.stream.isCoalescing"	为 false (默认值) 时，邻近的字符数据不进行连接。
"javax.xml.stream.isReplacingEntityReferences"	为 true (默认值) 时，实体引用将作为字符数据被替换和报告。
"javax.xml.stream.isSupportingExternalEntities"	为 true (默认值) 时，外部实体将被解析。规范对于这个属性没有给出默认值。
"javax.xml.stream.supportDTD"	为 true (默认值) 时，DTD 将作为事件被报告。

- XMLStreamReader createXMLStreamReader(InputStream in)
- XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)
- XMLStreamReader createXMLStreamReader(Reader in)
- XMLStreamReader createXMLStreamReader(Source in)

创建一个从给定的流、阅读器或 JAXP 源读入的解析器。

API **javax.xml.stream.XMLStreamReader** 6

- boolean hasNext()
- 如果有另一个解析事件，则返回 true。
- int next()

将解析器的状态设置为下一个解析事件，并返回下列常量之一：START_ELEMENT、END_ELEMENT、CHARACTERS、START_DOCUMENT、END_DOCUMENT、CDATA、COMMENT、SPACE (可忽略的空白字符)、

PROCESSING_INSTRUCTION、ENTITY_REFERENCE、DTD。

- boolean isStartElement()
- boolean isEndElement()
- boolean isCharacters()
- boolean isWhiteSpace()

如果当前事件是一个开始元素、结束元素、字符数据或空白字符，则返回 true。

- QName getName()
- String getLocalName()

获取在 START_ELEMENT 或 END_ELEMENT 事件中的元素的名字。

- String getText()

返回一个 CHARACTERS、COMMENT 或 CDATA 事件中的字符，或一个 ENTITY_REFERENCE 的替换值，或者一个 DTD 的内部子集。

- int getAttributeCount()
- QName getAttributeName(int index)
- String getAttributeLocalName(int index)
- String getAttributeValue(int index)

如果当前事件是 START_ELEMENT，则获取属性数量和属性的名字与值。

- String getAttributeValue(String namespaceURI, String name)

如果当前事件是 START_ELEMENT，则获取具有给定名称的属性的值。如果 namespaceURI 为 null，则不检查名字空间。

3.8 生成 XML 文档

现在你已经知道怎样编写读取 XML 的 Java 程序了。下面让我们开始介绍它的反向过程，即产生 XML 输出。当然，你可以直接通过一系列 print 调用，打印出各元素、属性和文本内容，以此来编写 XML 文件，但这并不是一个好主意。这样的代码会非常冗长复杂，对于属性值和文本内容中的那些特殊符号（如：“ 和 <），一不注意就会出错。

一种更好的方式是用文档的内容构建一棵 DOM 树，然后再写出该树的所有内容。下面的小节将讨论其细节。

3.8.1 不带命名空间的文档

要建立一棵 DOM 树，可以从一个空的文档开始。通过调用 DocumentBuilder 类的 newDocument 方法可以得到一个空文档。

```
Document doc = builder.newDocument();
```

使用 Document 类的 createElement 方法可以构建文档里的元素：

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

使用 `createTextNode` 方法可以构建文本节点：

```
Text textNode = doc.createTextNode(textContents);
```

使用以下方法可以给文档添加根元素，给父结点添加子节点：

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

在建立 DOM 树时，可能还需要设置元素属性，这只需调用 `Element` 类的 `setAttribute` 方法即可：

```
rootElement.setAttribute(name, value);
```

3.8.2 带命名空间的文档

如果要使用命名空间，那么创建文档的过程就会稍微有些差异。

首先需要将生成器工厂设置为是命名空间感知的，然后创建生成器：

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
builder = factory.newDocumentBuilder();
```

再使用 `createElementNS` 而不是 `createElement` 来创建所有节点：

```
String namespace = "http://www.w3.org/2000/svg";
Element rootElement = doc.createElementNS(namespace, "svg");
```

如果节点具有带命名空间前缀的限定名，那么所有必需的带有 `xmlns` 前缀的属性都会被自动创建。例如，如果需要在 HTML 中包含 SVG，那么就可以像下面这样构建元素：

```
Element svgElement = doc.createElement(namespace, "svg:svg")
```

当该元素被写入 XML 文件时，它会转变为：

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
```

如果需要设置的元素属性的名字位于命名空间中，那么可以使用 `Element` 类的 `setAttributeNS` 方法：

```
rootElement.setAttributeNS(namespace, qualifiedName, value);
```

3.8.3 写出文档

有些奇怪的是，把 DOM 树写出到输出流中并非一件易事。最容易的方式是使用可扩展的样式表语言转换（Extensible Stylesheet Language Transformations, XSLT）API。关于 XSLT 的更多信息请参见 3.9 节。当下，我们先考虑根据生成 XML 输出的“魔咒”而编写的代码。

我们把“不做任何操作”的转换应用于文档，并且捕获它的输出。为了将 `DOCTYPE` 节点纳入输出，我们还需要将 `SYSTEM` 和 `PUBLIC` 标识符设置为输出属性。

```
// construct the do-nothing transformation
```

```

Transformer t = TransformerFactory.newInstance().newTransformer();
// set output properties to get a DOCTYPE node
t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, systemIdentifier);
t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, publicIdentifier);
// set indentation
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
// apply the do-nothing transformation and send the output to a file
t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(file)));

```

另一种方式是使用 `LSSerializer` 接口。为了获取实例，可以使用下面的魔咒：

```

DOMImplementation impl = doc.getImplementation();
var implLS = (DOMImplementationLS) impl.getFeature("LS", "3.0");
LSSerializer ser = implLS.createLSSerializer();

```

如果需要空格和换行，可以设置下面的标志：

```
ser.getDomConfig().setParameter("format-pretty-print", true);
```

然后可以易如反掌地将文档转换为字符串：

```
String str = ser.writeToString(doc);
```

如果想要将输出直接写入到文件中，则需要一个 `LSOutput`：

```

LSOutput out = implLS.createLSOutput();
out.setEncoding("UTF-8");
out.setByteStream(Files.newOutputStream(path));
ser.write(doc, out);

```

API `javax.xml.parsers.DocumentBuilder` 1.4

- `Document newDocument()`

返回一个空文档。

API `org.w3c.dom.Document` 1.4

- `Element createElement(String name)`
- `Element createElementNS(String uri, String qname)`
返回具有给定名字的元素。
- `Text createTextNode(String data)`
返回具有给定数据的文本节点。

API `org.w3c.dom.Node` 1.4

- `Node appendChild(Node child)`

在该节点的子节点列表中追加一个节点。返回被追加的节点。

API `org.w3c.dom.Element` 1.4

- `void setAttribute(String name, String value)`
- `void setAttributeNS(String uri, String qname, String value)`

将有给定名字的属性设置为指定的值。

如果限定名有别名前缀，则 uri 不能为 null。

API **javax.xml.transform.TransformerFactory 1.4**

- static TransformerFactory newInstance()
 - 返回 TransformerFactory 类的一个实例。
- Transformer newTransformer()
 - 返回 Transformer 类的一个实例，它实现了标识符转换（不做任何事情的转换）。

API **javax.xml.transform.Transformer 1.4**

- void setOutputProperty(String name, String value)

设置输出属性。标准输出属性参见 <http://www.w3.org/TR/xslt#output>，其中最有用的几个如下所示：

doctype-public DOCTYPE 声明中使用的公共 ID
 doctype-system DOCTYPE 声明中使用的系统 ID
 Indent “yes” 或者 “no”
 method “xml”“html”、“text”或定制的字符串

- void transform(Source from, Result to)

转换一个 XML 文档。

API **javax.xml.transform.dom.DOMSource 1.4**

- DOMSource(Node n)

从给定的节点中构建一个源。通常，n 是文档节点。

API **javax.xml.transform.stream.StreamResult 1.4**

- StreamResult(File f)
- StreamResult(OutputStream out)
- StreamResult(Writer out)
- StreamResult(String systemID)

从文件、流、写出器或系统 ID（通常是相对或绝对 URL）中构建流结果。

3.8.4 使用 StAX 写出 XML 文档

在前一节中，你看到了如何通过写出 DOM 树的方法来产生 XML 文件。如果这个 DOM 树没有其他任何用途，那么这种方式就不是很高效。

StAX API 使我们可以直接将 XML 树写出，这需要从某个 OutputStream 中构建一个 XML-StreamWriter，就像下面这样：

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

要产生 XML 文件头，需要调用

```
writer.writeStartDocument()
```

然后调用

```
writer.writeStartElement(name);
```

添加属性需要调用

```
writer.writeAttribute(name, value);
```

现在，可以通过再次调用 `writeStartElement` 添加新的子节点，或者用下面的调用写出字符：

```
writer.writeCharacters(text);
```

在写完所有子节点之后，调用

```
writer.writeEndElement();
```

这会导致当前元素被关闭。

要写出没有子节点的元素（例如 ``），可以使用下面的调用

```
writer.writeEmptyElement(name);
```

最后，在文档的结尾，调用

```
writer.writeEndDocument();
```

将关闭所有打开的元素。

你仍旧需要关闭 `XMLStreamWriter`，并且需要人为关闭它，因为 `XMLStreamWriter` 接口没有扩展 `AutoCloseable` 接口。

与使用 DOM/XSLT 的方式一样，我们不必担心属性值和字符数据中的转义字符。但是，我们仍旧有可能会产生非良构的 XML，例如具有多个根节点的文档。并且，StAX 当前的版本还没有任何对产生缩进输出的支持。

程序清单 3-9 中的程序展示了写出 XML 的两种方式。

程序清单 3-9 write/XMLWriteTest.java

```

1 package write;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6
7 import javax.xml.parsers.*;
8 import javax.xml.stream.*;
9 import javax.xml.transform.*;
10 import javax.xml.transform.dom.*;
11 import javax.xml.transform.stream.*;
12
13 import org.w3c.dom.*;
14
15 /**
16  * This program shows how to write an XML file. It produces modern art in SVG

```

```
17 * format.
18 * @version 1.12 2016-04-27
19 * @author Cay Horstmann
20 */
21 public class XMLWriteTest
22 {
23     public static void main(String[] args) throws Exception
24     {
25         Document doc = newDrawing(600, 400);
26         writeDocument(doc, "drawing1.svg");
27         writeNewDrawing(600, 400, "drawing2.svg");
28     }
29
30     private static Random generator = new Random();
31
32     /**
33      * Creates a new random drawing.
34      * @return the DOM tree of the SVG document
35      */
36     public static Document newDrawing(int drawingWidth, int drawingHeight)
37             throws ParserConfigurationException
38     {
39         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
40         factory.setNamespaceAware(true);
41         DocumentBuilder builder = factory.newDocumentBuilder();
42         var namespace = "http://www.w3.org/2000/svg";
43         Document doc = builder.newDocument();
44         Element svgElement = doc.createElementNS(namespace, "svg");
45         doc.appendChild(svgElement);
46         svgElement.setAttribute("width", "" + drawingWidth);
47         svgElement.setAttribute("height", "" + drawingHeight);
48         int n = 10 + generator.nextInt(20);
49         for (int i = 1; i <= n; i++)
50         {
51             int x = generator.nextInt(drawingWidth);
52             int y = generator.nextInt(drawingHeight);
53             int width = generator.nextInt(drawingWidth - x);
54             int height = generator.nextInt(drawingHeight - y);
55             int r = generator.nextInt(256);
56             int g = generator.nextInt(256);
57             int b = generator.nextInt(256);
58
59             Element rectElement = doc.createElementNS(namespace, "rect");
60             rectElement.setAttribute("x", "" + x);
61             rectElement.setAttribute("y", "" + y);
62             rectElement.setAttribute("width", "" + width);
63             rectElement.setAttribute("height", "" + height);
64             rectElement.setAttribute("fill",
65                     String.format("#%02x%02x%02x", r, g, b));
66             svgElement.appendChild(rectElement);
67         }
68     return doc;
69 }
70 }
```

```

71  /**
72   * Saves a document using DOM/XSLT
73   */
74  public static void writeDocument(Document doc, String filename)
75      throws TransformerException, IOException
76  {
77      Transformer t = TransformerFactory.newInstance().newTransformer();
78      t.setOutputProperty(OutputKeys.DOCUMENT_TYPE_SYSTEM,
79          "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
80      t.setOutputProperty(OutputKeys.DOCUMENT_PUBLIC "-//W3C//DTD SVG 20000802//EN";
81      t.setOutputProperty(OutputKeys.INDENT, "yes");
82      t.setOutputProperty(OutputKeys.METHOD, "xml");
83      t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
84      t.transform(new DOMSource(doc), new StreamResult(
85          Files.newOutputStream(Paths.get(filename))));}
86  }
87
88 /**
89  * Writes an SVG document of the current drawing.
90  * @param writer the document destination
91  * @throws IOException
92  */
93 public static void writeNewDrawing(int drawingWidth, int drawingHeight,
94      String filename) throws XMLStreamException, IOException
95  {
96      XMLOutputFactory factory = XMLOutputFactory.newInstance();
97      XMLStreamWriter writer = factory.createXMLStreamWriter(
98          Files.newOutputStream(Paths.get(filename)));
99      writer.writeStartDocument();
100     writer.writeDTD("<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN" "
101         + "\nhttp://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd\n>");}
102     writer.writeStartElement("svg");
103     writer.writeDefaultNamespace("http://www.w3.org/2000/svg");
104     writer.writeAttribute("width", "" + drawingWidth);
105     writer.writeAttribute("height", "" + drawingHeight);
106     int n = 10 + generator.nextInt(20);
107     for (int i = 1; i <= n; i++)
108     {
109         int x = generator.nextInt(drawingWidth);
110         int y = generator.nextInt(drawingHeight);
111         int width = generator.nextInt(drawingWidth - x);
112         int height = generator.nextInt(drawingHeight - y);
113         int r = generator.nextInt(256);
114         int g = generator.nextInt(256);
115         int b = generator.nextInt(256);
116         writer.writeEmptyElement("rect");
117         writer.writeAttribute("x", "" + x);
118         writer.writeAttribute("y", "" + y);
119         writer.writeAttribute("width", "" + width);
120         writer.writeAttribute("height", "" + height);
121         writer.writeAttribute("fill", String.format("#%02x%02x%02x", r, g, b));
122     }
123     writer.writeEndDocument(); // closes sv element

```

```
125 }
126 }
```

API **javax.xml.stream.XMLOutputFactory** 6

- static XMLOutputFactory newInstance()

返回 XMLOutputFactory 类的一个实例。
- XMLStreamWriter createXMLStreamWriter(OutputStream in)
 ● XMLStreamWriter createXMLStreamWriter(OutputStream in, String characterEncoding)
 ● XMLStreamWriter createXMLStreamWriter(Writer in)
 ● XMLStreamWriter createXMLStreamWriter(Result in)

创建写出到给定流、写出器或 JAXP 结果的写出器。

API **javax.xml.stream.XMLStreamWriter** 6

- void writeStartDocument()
 ● void writeStartDocument(String xmlVersion)
 ● void writeStartDocument(String encoding, String xmlVersion)

在文档的顶部写入 XML 处理指令。注意，encoding 参数只是用于写入这个属性，它不会设置输出的字符编码机制。
- void setDefaultNamespace(String namespaceURI)
 ● void setPrefix(String prefix, String namespaceURI)

设置默认的命名空间，或者具有前缀的命名空间。这种声明的作用域只是当前元素，如果没有写明具体元素，其作用域为文档的根。
- void writeStartElement(String localName)
 ● void writeStartElement(String namespaceURI, String localName)

写出一个开始标签，其中 namespaceURI 将用相关联的前缀来代替。
- void writeEndElement()

关闭当前元素。
- void writeEndDocument()

关闭所有打开的元素。
- void writeEmptyElement(String localName)
 ● void writeEmptyElement(String namespaceURI, String localName)

写出一个自闭合的标签，其中 namespaceURI 将用相关联的前缀来代替。
- void writeAttribute(String localName, String value)
 ● void writeAttribute(String namespaceURI, String localName, String value)

写出一个用于当前元素的属性，其中 namespaceURI 将用相关联的前缀来代替。
- void writeCharacters(String text)

写出字符数据。

- void writeCData(String text)
写出 CDATA 块。
- void writeDTD(String dtd)
写出 dtd 字符串，该字符串需要包含一个 DOCTYPE 声明。
- void writeComment(String comment)
写出一个注释。
- void close()
关闭这个写出器。

3.8.5 示例：生成 SVG 文件

程序清单 3-9 是一个生成 XML 输出的典型程序。该程序绘制了一幅现代派绘画，即一组随机的彩色矩形（参见图 3-3）。我们使用可伸缩向量图形（Scalable Vector Graphics, SVG）来保存作品。SVG 是 XML 格式的，它使用设备无关的方式描述复杂图形。你可以在 <http://www.w3c.org/Graphics/SVG> 找到更多关于 SVG 的信息。要查看 SVG 文件，只需使用任意的现在主流的浏览器。

该程序演示了两种产生 XML 的方式：通过构建并保存 DOM 树，以及通过直接用 StAX API 写出 XML。

我们并没有涉及 SVG 的细节。就我们的目的而言，我们只需要知道怎样表示一组彩色的矩形。下面是一个例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
 "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="150">
<rect x="231" y="61" width="9" height="12" fill="#6e4a13"/>
<rect x="107" y="106" width="56" height="5" fill="#c406be"/>
...
</svg>
```

正如你看到的，每个矩形都被描述成了一个 rect 节点。它有位置、宽度、高度和填充色等属性，其中填充色以十六进制 RGB 值表示。

 **注释：**SVG 大量使用了属性。实际上，某些属性相当复杂。例如，下面的 path 元素：

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

M 是指“moveto”命令、L 是指“lineto”、z 是指“closepath”(!)。显然，该数据格式的设计者不太信任 XML 表示结构化数据的能力。在你自己的 XML 格式中，你可能想使用元素来替代复杂的属性。

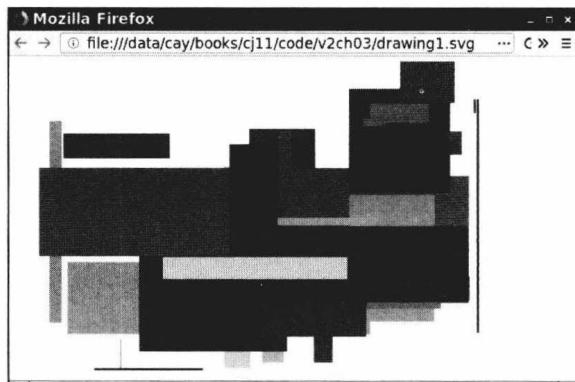


图 3-3 生成的现代艺术品

3.9 XSL 转换

XSL 转换 (XSLT) 机制可以指定将 XML 文档转换为其他格式的规则，例如，转换为纯文本、XHTML 或任何其他的 XML 格式。XSLT 通常用来将某种机器可读的 XML 格式转译为另一种机器可读的 XML 格式，或者将 XML 转译为适于人类阅读的表示格式。

你需要提供 XSLT 样式表，它描述了 XML 文档向某种其他格式转换的规则。XSLT 处理器将读入 XML 文档和这个样式表，并产生所要的输出（参见图 3-4）。

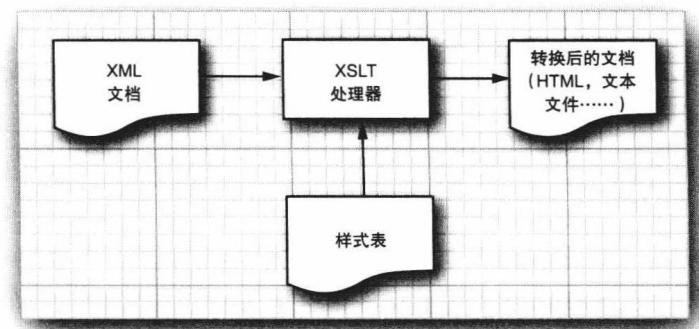


图 3-4 应用 XSL 转换

XSLT 规范很复杂，已经有很多书描述了该主题。我们不可能讨论 XSLT 的全部特性，所以我们只能介绍一个有代表性的例子。你可以在 Don Box 等人合著的 *Essential XML* 一书中找到更多的信息。XSLT 规范可以在 <http://www.w3.org/TR/xslt> 获得。

假设我们想要把有雇员记录的 XML 文件转换成 HTML 文件。请看这个输入文件：

```

<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
    <hiredate year="1989" month="10" day="1"/>
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000</salary>
    <hiredate year="1990" month="3" day="15"/>
  </employee>
</staff>
  
```

我们希望的输出是一张 HTML 表格：

```

<table border="1">
<tr>
  
```

```

<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>

```

具有转换模板的样式表形式如下：

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="html"/>
    template1

    template2
    .
    .
    .
</xsl:stylesheet>

```

在我们的例子中，`xsl:output` 元素将方法设定为 HTML，而其他有效的方法设置是 `xml` 和 `text`。

下面是一个典型的模板：

```

<xsl:template match="/staff/employee">
    <tr><xsl:apply-templates/></tr>
</xsl:template>

```

`match` 属性的值是一个 XPath 表达式。该模板声明，每当看到 XPath 集 `/staff/ employee` 中的一个节点时，将做以下操作：

1. 产生字符串 `<tr>`。
2. 在处理其子节点时，持续应用该模板。
3. 当处理完所有子节点后，产生字符串 `</tr>`。

换句话说，该模板会生成围绕每条雇员记录的 HTML 表格的行标记。

XSLT 处理器以检查根元素开始其处理过程。每当一个节点匹配某个模板时，就会应用该模板（如果匹配多个模板，就会使用最佳匹配的那个，详情请参见 <http://www.w3.org/TR/xslt>）。如果没有匹配的模板，处理器会执行默认操作。对于文本节点，默认操作是把它的内容囊括到输出中去。对于元素，默认操作是不产生任何输出，但会继续处理其子节点。

下面是一个用来转换雇员记录文件中的 `name` 节点的模板：

```

<xsl:template match="/staff/employee/name">
    <td><xsl:apply-templates/></td>
</xsl:template>

```

正如你所见，模板产生定界符 `<td>...</td>`，并且让处理器递归访问 `name` 元素的子节点。它只有一个子节点，即文本节点。当处理器访问该节点时，它会提取出其中的文本内容（当然，前提是没有任何其他匹配的模板）。

如果想要把属性值复制到输出中去，就不得不再做一些稍微复杂的操作了。下面是一个例子：

```
<xsl:template match="/staff/employee/hiredate">
    <td><xsl:value-of select="@year"/>-<xsl:value-of
        select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

当处理 hiredate 节点时，该模板会产生：

1. 字符串 `<td>`
2. year 属性的值
3. 一个连字符
4. month 属性的值
5. 一个连字符
6. day 属性的值
7. 字符串 `</td>`

`xsl:value-of` 语句用于计算节点集的字符串值，其中，节点集由 `select` 属性的 XPath 值指定。在这个例子中，路径是相对于当前正在处理的节点的相对路径。节点集通过将各个节点的字符串值连接起来而被转换成一个字符串。属性节点的字符串值就是它的值，文本节点的字符串值是它的内容，元素节点的字符串值是它的所有子节点（而不是其属性）的字符串值的连接。

程序清单 3-10 包含了将带有雇员记录的 XML 文件转换成 HTML 表格的样式表。

程序清单 3-10 transform/makehtml.xsl

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2
3 <xsl:stylesheet
4     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5     version="1.0">
6
7     <xsl:output method="html"/>
8
9     <xsl:template match="/staff">
10         <table border="1"><xsl:apply-templates/></table>
11     </xsl:template>
12
13     <xsl:template match="/staff/employee">
14         <tr><xsl:apply-templates/></tr>
15     </xsl:template>
16
17     <xsl:template match="/staff/employee/name">
18         <td><xsl:apply-templates/></td>
19     </xsl:template>
20
21     <xsl:template match="/staff/employee/salary">
22         <td>$<xsl:apply-templates/></td>
23     </xsl:template>
```

```

24
25   <xsl:template match="/staff/employee/hiredate">
26     <td><xsl:value-of select="@year"/>-<xsl:value-of
27       select="@month"/>-<xsl:value-of select="@day"/></td>
28   </xsl:template>
29
30 </xsl:stylesheet>

```

程序清单 3-11 显示了一组不同的转换，其输入是相同的 XML 文件，输出是我们熟悉的属性文件格式的纯文本。

```

employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15

```

程序清单 3-11 transform/makeprop.xsl

```

1  <?xml version="1.0"?>
2
3  <xsl:stylesheet
4    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5    version="1.0">
6
7    <xsl:output method="text" omit-xml-declaration="yes"/>
8
9      <xsl:template match="/staff/employee">
10     employee.<xsl:value-of select="position()">
11     />.name=<xsl:value-of select="name/text()" />
12     employee.<xsl:value-of select="position()">
13     />.salary=<xsl:value-of select="salary/text()" />
14     employee.<xsl:value-of select="position()">
15     />.hiredate=<xsl:value-of select="hiredate/@year"
16     />-<xsl:value-of select="hiredate/@month"
17     />-<xsl:value-of select="hiredate/@day"/>
18   </xsl:template>
19
20 </xsl:stylesheet>

```

该示例使用 `position()` 函数来产生以其父节点的角度来看的当前节点的位置。我们只要切换样式表就可以得到一个完全不同的输出。这样，就可以安全地使用 XML 来描述数据了，即便一些应用程序需要的是其他格式的数据，我们只要用 XSLT 来产生对应的可替代格式即可。

在 Java 平台上产生 XML 的转换极其简单，只需为每个样式表设置一个转换器工厂，然后得到一个转换器对象，并告诉它把一个源转换成结果。

```

var styleSheet = new File(filename);
var styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
t.transform(source, result);

```

`transform` 方法的参数是 `Source` 和 `Result` 接口的实现类的对象。`Source` 接口有 4 个实现类：

```

DOMSource
SAXSource
StAXSource
StreamSource

```

你可以从一个文件、流、阅读器或 URL 中构建 `StreamSource` 对象，或者从 DOM 树节点中构建 `DOMSource` 对象。例如，在上一节中，我们调用了如下的标识转换：

```
t.transform(new DOMSource(doc), result);
```

在示例程序中，我们做了一些更有趣的事情。我们并不是从一个现有的 XML 文件开始工作，而是产生一个 SAX XML 阅读器，通过产生适合的 SAX 事件，给人以解析 XML 文件的感觉。实际上，XML 阅读器读入的是一个如第 2 章所描述的扁平文件，输入文件看上去是这样的：

```

Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15

```

处理输入时，XML 阅读器将产生 SAX 事件。下面是实现了 `XMLReader` 接口的 `EmployeeReader` 类的 `parse` 方法的一部分代码：

```

var attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
    handler.startElement("", "employee", "employee", attributes);
    var tokenizer = new StringTokenizer(line, "|");
    handler.startElement("", "name", "name", attributes);
    String s = tokenizer.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
    handler.endElement("", "name", "name");

    .
    .
    .
    handler.endElement("", "employee", "employee");
}
handler.endElement("", rootElement, rootElement);
handler.endDocument();

```

用于转换器的 `SAXSource` 是从 XML 阅读器中构建的：

```

t.transform(new SAXSource(new EmployeeReader(),
    new InputSource(new FileInputStream(filename))), result);

```

这是将非 XML 的遗留数据转换成 XML 的一个小技巧。当然，大多数 XSLT 应用程序都已经有了 XML 格式的输入数据，只需要在一个 `StreamSource` 对象上调用 `transform` 方法即可，例如：

```
t.transform(new StreamSource(file), result);
```

其转换结果是 `Result` 接口的实现类的一个对象。Java 库提供了 3 个类：

`DOMResult`
`SAXResult`
`StreamResult`

要把结果存储到 DOM 树中，请使用 `DocumentBuilder` 产生一个新的文档节点，并将其包装到 `DOMResult` 中：

```
Document doc = builder.newDocument();
t.transform(source, new DOMResult(doc));
```

要将输出保存到文件中，请使用 `StreamResult`：

```
t.transform(source, new StreamResult(file));
```

程序清单 3-12 包含了完整的源代码。

程序清单 3-12 transform/TransformTest.java

```
1 package transform;
2
3 import java.io.*;
4 import java.nio.file.*;
5 import java.util.*;
6 import javax.xml.transform.*;
7 import javax.xml.transform.sax.*;
8 import javax.xml.transform.stream.*;
9 import org.xml.sax.*;
10 import org.xml.sax.helpers.*;
11
12 /**
13 * This program demonstrates XSL transformations. It applies a transformation to a set of
14 * employee records. The records are stored in the file employee.dat and turned into XML
15 * format. Specify the stylesheet on the command line, e.g.<br>
16 *      java transform.TransformTest transform/makeprop.xsl
17 * @version 1.04 2018-04-10
18 * @author Cay Horstmann
19 */
20 public class TransformTest
21 {
22     public static void main(String[] args) throws Exception
23     {
24         Path path;
25         if (args.length > 0) path = Paths.get(args[0]);
26         else path = Paths.get("transform", "makehtml.xsl");
27         try (InputStream styleIn = Files.newInputStream(path))
28         {
29             var styleSource = new StreamSource(styleIn);
30
31             Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
32             t.setOutputProperty(OutputKeys.INDENT, "yes");
33             t.setOutputProperty(OutputKeys.METHOD, "xml");
34             t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
35
36             try (InputStream docIn = Files.newInputStream(Paths.get("transform", "employee.dat")));
37             {
38                 String result = t.transform(new StreamSource(docIn), null).getOutput();
39                 System.out.println(result);
40             }
41         }
42     }
43 }
```

```
37     {
38         t.transform(new SAXSource(new EmployeeReader(), new InputSource(docIn)),
39             new StreamResult(System.out));
40     }
41 }
42 }
43 }
44
45 /**
46  * This class reads the flat file employee.dat and reports SAX parser events to act as if it
47  * was parsing an XML file.
48 */
49 class EmployeeReader implements XMLReader
50 {
51     private ContentHandler handler;
52
53     public void parse(InputSource source) throws IOException, SAXException
54     {
55         InputStream stream = source.getByteStream();
56         var in = new BufferedReader(new InputStreamReader(stream));
57         String rootElement = "staff";
58         var atts = new AttributesImpl();
59
60         if (handler == null) throw new SAXException("No content handler");
61
62         handler.startDocument();
63         handler.startElement("", rootElement, rootElement, atts);
64         String line;
65         while ((line = in.readLine()) != null)
66         {
67             handler.startElement("", "employee", "employee", atts);
68             var t = new StringTokenizer(line, "|");
69
70             handler.startElement("", "name", "name", atts);
71             String s = t.nextToken();
72             handler.characters(s.toCharArray(), 0, s.length());
73             handler.endElement("", "name", "name");
74
75             handler.startElement("", "salary", "salary", atts);
76             s = t.nextToken();
77             handler.characters(s.toCharArray(), 0, s.length());
78             handler.endElement("", "salary", "salary");
79
80             atts.addAttribute("", "year", "year", "CDATA", t.nextToken());
81             atts.addAttribute("", "month", "month", "CDATA", t.nextToken());
82             atts.addAttribute("", "day", "day", "CDATA", t.nextToken());
83             handler.startElement("", "hiredate", "hiredate", atts);
84             handler.endElement("", "hiredate", "hiredate");
85             atts.clear();
86
87             handler.endElement("", "employee", "employee");
88     }
89
90     handler.endElement("", rootElement, rootElement);
```

```

91     handler.endDocument();
92 }
93
94 public void setContentHandler(ContentHandler newValue)
95 {
96     handler = newValue;
97 }
98
99 public ContentHandler getContentHandler()
100 {
101     return handler;
102 }
103
104 // the following methods are just do-nothing implementations
105 public void parse(String systemId) throws IOException, SAXException {}
106 public void setErrorHandler(ErrorHandler handler) {}
107 public ErrorHandler getErrorHandler() { return null; }
108 public void setDTDHandler(DTDHandler handler) {}
109 public DTDHandler getDTDHandler() { return null; }
110 public void setEntityResolver(EntityResolver resolver) {}
111 public EntityResolver getEntityResolver() { return null; }
112 public void setProperty(String name, Object value) {}
113 public Object getProperty(String name) { return null; }
114 public void setFeature(String name, boolean value) {}
115 public boolean getFeature(String name) { return false; }
116 }

```

API **javax.xml.transform.TransformerFactory 1.4**

- Transformer newTransformer(Source styleSheet)

返回一个 transformer 类的实例，用来从指定的源中读取样式表。

API **javax.xml.transform.stream.StreamSource 1.4**

- StreamSource(File f)
- StreamSource(InputStream in)
- StreamSource(Reader in)
- StreamSource(String systemID)

自一个文件、流、阅读器或系统 ID（通常是相对或绝对 URL）构建一个数据流源。

API **javax.xml.transform.sax.SAXSource 1.4**

- SAXSource(XMLReader reader, InputSource source)

构建一个 SAX 数据源，以便从给定输入源中获取数据，并使用给定的阅读器来解析输入数据。

API **org.xml.sax.XMLReader 1.4**

- void setContentHandler(ContentHandler handler)

设置在输入被解析时会被告知解析事件的处理器。

- void parse(InputSource source)

解析来自给定输入源的输入数据，并将解析事件发送到内容处理器。

API javax.xml.transform.dom.DOMResult 1.4

- DOMResult(Node n)

自给定节点构建一个数据源。通常，n 是一个新文档节点。

API org.xml.sax.helpers.AttributesImpl 1.4

- void addAttribute(String uri, String lname, String qname, String type, String value)

将一个属性添加到该属性集合。

lname 参数是无前缀的本地名，而 qname 参数是带前缀的限定名，type 参数是 "CDATA" "ID" "IDREF" "IDREFS" "NMTOKEN" "NMTOKENS" "ENTITY" "ENTITIES" 或 "NOTATION" 之一

- void clear()

删除当前属性集合中的所有属性。

我们以该示例结束对 Java 库中的 XML 支持特性的讨论。现在，你应该对 XML 的强大功能有了很好的了解，尤其是它的自动解析、验证和强大的转换机制。当然，所有这些技术只有在你很好地设计了 XML 格式之后才能发挥作用。你必须确保那些格式足够丰富，能够表达全部业务需求，随着时间的推移也依旧稳定，你的业务伙伴也愿意接受你的 XML 文档。这些问题要远比处理解析器、DTD 或转换更具挑战。

在下一章，我们将讨论在 Java 平台上的网络编程，从最基础的网络套接字开始，逐渐过渡到用于 E-mail 和万维网的更高层协议。

第4章 网络

- ▲ 连接到服务器
- ▲ 实现服务器
- ▲ 获取 Web 数据

- ▲ HTTP 客户端
- ▲ 发送 E-mail

本章的开头部分将首先回顾一下网络方面的基本概念，然后进一步介绍如何编写连接网络服务的 Java 程序，并演示网络客户端和服务器是如何实现的，最后将介绍如何通过 Java 程序发送 E-mail，以及如何从 Web 服务器获得信息。

4.1 连接到服务器

在下面各节中，你将会学习如何连接到服务器，先是手工用 telnet 连接，然后是用 Java 程序连接。

4.1.1 使用 telnet

telnet 是一种用于网络编程的非常强大的调试工具，可以在命令 shell 中输入 telnet 来启动它。

注释：在 Windows 中，需要激活 telnet。要激活它，需要到“控制面板”，选择“程序”，点击“打开 / 关闭 Windows 特性”，然后选择“Telnet 客户端”复选框。Windows 防火墙将会阻止我们在本章中使用的很多网络端口，你可能需要管理员账户才能解除对它们的禁用。

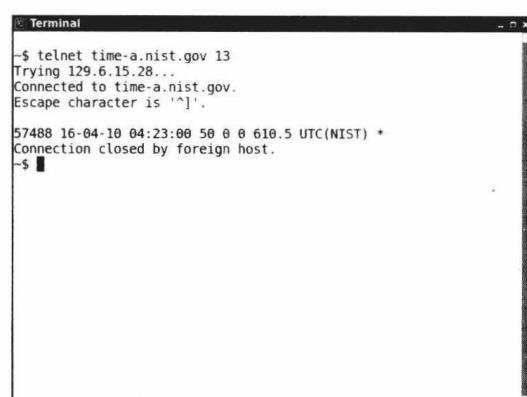
你可能曾经使用过 telnet 来连接远程计算机，但其实你也可以用它与因特网主机所提供的其他服务进行通信。下面是一个可以操作的例子。请输入：

```
telnet time-a.nist.gov 13
```

如图 4-1 所示，你可以得到与下面这一行相似的信息：

```
57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *
```

上面例子说明了什么？它说明你已经连接到了大多数 UNIX 计算机都支持的“当日时



The screenshot shows a terminal window titled "Terminal". The command entered was "telnet time-a.nist.gov 13". The output shows the connection process: "Trying 129.6.15.28...", "Connected to time-a.nist.gov.", and "Escape character is '^]'. The timestamp at the end of the session is "57488 16-04-10 04:23:00 50 0 0 610.5 UTC(NIST) *". The connection is closed by the foreign host.

图 4-1 “当日时间”服务的输出

间”服务。而你刚才所连接的那台服务器就是由国家标准与技术研究所运维的，这家研究所负责提供铯原子钟的计量时间。（当然，由于网络延迟的缘故，原子钟反馈过来的时间并不完全准确。）

按照惯例，“当日时间”服务总是连接到端口 13。

注释：在网络术语中，端口并不是指物理设备，而是为了便于实现服务器与客户端之间的通信所使用的抽象概念（见图 4-2）。



图 4-2 连接到服务器端口的客户端

运行在远程计算机上的服务器软件不停地等待那些希望与端口 13 连接的网络请求。当远程计算机上的操作系统接收到一个请求与端口 13 连接的网络数据包时，它便唤醒正在监听网络连接请求的服务器进程，并为两者建立连接。这种连接将一直保持下去，直到被其中任何一方中止。

当你开始用 `time-a.nist.gov` 在端口 13 上建立 telnet 会话时，网络软件中有一段代码非常清楚地知道应该将字符串 “`time-a.nist.gov`” 转换为正确的 IP 地址 129.6.15.28。随后，telnet 软件发送一个连接请求给该地址，请求一个到端口 13 的连接。一旦建立连接，远程程序便发送回一行数据，然后关闭该连接。当然，一般而言，客户端和服务器在其中一方关闭连接之前，会进行更多的对话。

下面是另一个同类型的试验，但它更加有趣。请执行以下操作：

```
telnet horstmann.com 80
```

然后非常仔细地键入以下内容：

```
GET / HTTP/1.1
Host: horstmann.com
blank line
```

也就是在末尾按两次 Enter 键。

图 4-3 显示了以上操作的响应结果。它看上去应该是你非常熟悉的——你得到的是一个

HTML 格式的文本页，即 Cay Horstmann 的主页。

```

$ telnet horstmann.com 80
Trying 67.210.118.65...
Connected to horstmann.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: horstmann.com

HTTP/1.1 200 OK
Date: Sun, 10 Apr 2016 04:36:27 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/0.9.8e-fips-rhel5 mod_auth_p
assthrough/2.1 mod_bwlimited/1.4 mod_fcgid/2.3.6 Sun-ONE-ASP/4.0.3
Last-Modified: Thu, 17 Mar 2016 18:32:18 GMT
ETag: "25900elc-1c47-52e42d9a8f080"
Accept-Ranges: bytes
Content-Length: 7239
Content-Type: text/html

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/x
html/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<title>Cay Horstmann's Home Page</title>
<link href="styles.css" rel="stylesheet" type="text/css"/>
```

图 4-3 使用 telnet 访问 HTTP 端口

上述操作与 Web 浏览器访问某个网页所经历的过程是完全一致的，它使用 HTTP 向服务器请求 Web 页面。当然，浏览器能够更精致地显示 HTML 代码。

注释：如果一台 Web 服务器用相同的 IP 地址为多个域提供宿主环境，那么在连接这台 Web Server 时，就必须提供 Host 键 / 值对。如果服务器只为单个域提供宿主环境，则可以忽略该键 / 值对。

4.1.2 用 Java 连接到服务器

程序清单 4-1 是我们的第一个网络程序。它的作用与我们使用 telnet 工具是相同的，即连接到某个端口并打印出它所找到的信息。

程序清单 4-1 socket/SocketTest.java

```

1 package socket;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.*;
7
8 /**
9 * This program makes a socket connection to the atomic clock in Boulder, Colorado, and prints
10 * the time that the server sends.
11 * @version 1.22 2018-03-17
12 * @author Cay Horstmann
13 */
14 public class SocketTest
15 {
16     public static void main(String[] args) throws IOException
17     {
18         try (var s = new Socket("time-a.nist.gov", 13);
```

```

19         var in = new Scanner(s.getInputStream(), StandardCharsets.UTF_8)
20     {
21         while (in.hasNextLine())
22         {
23             String line = in.nextLine();
24             System.out.println(line);
25         }
26     }
27 }
28 }
```

下面是这个简单程序的几行关键代码：

```
var s = new Socket("time-a.nist.gov", 13);
InputStream inStream = s.getInputStream();
```

第一行代码用于打开一个套接字，它也是网络软件中的一个抽象概念，负责启动该程序内部和外部之间的通信。我们将远程地址和端口号传递给套接字的构造器，如果连接失败，它将抛出一个 `UnknownHostException` 异常；如果存在其他问题，它将抛出一个 `IOException` 异常。因为 `UnknownHostException` 是 `IOException` 的一个子类，况且这只是一个示例程序，所以我们在那里仅仅捕获超类的异常。

一旦套接字被打开，`java.net.Socket` 类中的 `getInputStream` 方法就会返回一个 `InputStream` 对象，该对象可以像其他任何流对象一样使用。而一旦获取了这个流，该程序将直接把每一行打印到标准输出。这个过程将一直持续到流发送完毕且服务器断开连接为止。

该程序只适用于非常简单的服务器，比如“当日时间”之类的服务。在比较复杂的网络程序中，客户端发送请求数据给服务器，而服务器可能在响应结束时并不立刻断开连接。在本章的若干个示例程序中，都会看到我们是如何实现这种行为的。

`Socket` 类非常简单易用，因为 Java 库隐藏了建立网络连接和通过连接发送数据的复杂过程。实际上，`java.net` 包提供的编程接口与操作文件时所使用的接口基本相同。

注释：本书所介绍的内容仅覆盖了 TCP（传输控制协议）网络协议。Java 平台另外还支持 UDP（用户数据报协议）协议，该协议可以用于发送数据包（也称为数据报），它所需付出的开销要比 TCP 少得多。UDP 有一个重要的缺点：数据包无须按照顺序传递到接收应用程序，它们甚至可能在传输过程中全部丢失。UDP 让数据包的接收者自己负责对它们进行排序，并请求发送者重新发送那些丢失的数据包。UDP 比较适合于那些可以忍受数据包丢失的应用，例如用于音频流和视频流的传输，或者用于连续测量的应用领域。

API `java.net.Socket` 1.0

- `Socket(String host, int port)`
构建一个套接字，用来连接给定的主机和端口。
- `InputStream getInputStream()`

- `OutputStream getOutputStream()`

获取可以从套接字中读取数据的流，以及可以向套接字写出数据的流。

4.1.3 套接字超时

从套接字读取信息时，在有数据可供访问之前，读操作将会被阻塞。如果此时主机不可达，那么应用将要等待很长的时间，并且因为受底层操作系统的限制而最终会导致超时。

对于不同的应用，应该确定合理的超时值。然后调用 `setSoTimeout` 方法设置这个超时值（单位：毫秒）。

```
var s = new Socket(. . .);
s.setSoTimeout(10000); // time out after 10 seconds
```

如果已经为套接字设置了超时值，并且之后的读操作和写操作在没有完成之前就超过了时间限制，那么这些操作就会抛出 `SocketTimeoutException` 异常。你可以捕获这个异常，并对超时做出反应。

```
try
{
    InputStream in = s.getInputStream(); // read from in
    ...
}
catch (SocketTimeoutException e)
{
    react to timeout
}
```

另外还有一个超时问题是必须解决的。下面这个构造器：

```
Socket(String host, int port)
```

会一直无限期地阻塞下去，直到建立了到达主机的初始连接为止。

可以通过先构建一个无连接的套接字，然后再使用一个超时来进行连接的方式解决这个问题。

```
var s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

如果你希望允许用户在任何时刻都可以中断套接字连接，请查看 4.2.4 节。

API `java.net.Socket` 1.0

- `Socket() 1.1`

创建一个还未被连接的套接字。

- `void connect(SocketAddress address) 1.4`

将该套接字连接到给定的地址。

- `void connect(SocketAddress address, int timeoutInMilliseconds) 1.4`

将套接字连接到给定的地址。如果在给定的时间内没有响应，则返回。

- `void setSoTimeout(int timeoutInMilliseconds) 1.1`

设置该套接字上读请求的阻塞时间。如果超出给定时间，则抛出一个 `SocketTimeoutException` 异常。

- `boolean isConnected()` 1.4

如果该套接字已被连接，则返回 `true`。

- `boolean isClosed()` 1.4

如果套接字已经被关闭，则返回 `true`。

4.1.4 因特网地址

通常，不用过多考虑因特网地址的问题，它们是用一串数字表示的主机地址，一个因特网地址由 4 个字节组成（在 IPv6 中是 16 个字节），比如 129.6.15.28。但是，如果需要在主机名和因特网地址之间进行转换，那么就可以使用 `InetAddress` 类。

只要主机操作系统支持 IPv6 格式的因特网地址，`java.net` 包也将支持它。

静态的 `getByName` 方法可以返回代表某个主机的 `InetAddress` 对象。例如，

```
InetAddress address = InetAddress.getByName("time-a.nist.gov");
```

将返回一个 `InetAddress` 对象，该对象封装了一个 4 字节的序列：129.6.15.28。然后，可以使用 `getAddress` 方法来访问这些字节：

```
byte[] addressBytes = address.getAddress();
```

一些访问量较大的主机名通常会对应于多个因特网地址，以实现负载均衡。例如，在撰写本书时，主机名 `google.com` 就对应着 12 个不同的因特网地址。当访问主机时，会随机选取其中的一个。可以通过调用 `getAllByName` 方法来获得所有主机：

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

最后需要说明的是，有时我们可能需要本地主机的地址。如果只是要求得到 `localhost` 的地址，那总会得到本地回环地址 127.0.0.1，但是其他程序无法用这个地址来连接到这台机器上。此时，可以使用静态的 `getLocalHost` 方法来得到本地主机的地址：

```
InetAddress address = InetAddress.getLocalHost();
```

程序清单 4-2 是一段比较简单的程序代码。如果不命令行中设置任何参数，那么它将打印出本地主机的因特网地址。反之，如果在命令行中指定了主机名，那么它将打印出该主机的所有因特网地址，例如：

```
java InetAddress/InetAddressTest www.horstmann.com
```

程序清单 4-2 InetAddress/InetAddressTest.java

```
1 package InetAddress;
2
3 import java.io.*;
4 import java.net.*;
5
6 /**
7 * This program demonstrates the InetAddress class. Supply a host name as command-line
```

```

8 * argument, or run without command-line arguments to see the address of the local host.
9 * @version 1.02 2012-06-05
10 * @author Cay Horstmann
11 */
12 public class InetAddressTest
13 {
14     public static void main(String[] args) throws IOException
15     {
16         if (args.length > 0)
17         {
18             String host = args[0];
19             InetAddress[] addresses = InetAddress.getAllByName(host);
20             for (InetAddress a : addresses)
21                 System.out.println(a);
22         }
23         else
24         {
25             InetAddress localHostAddress = InetAddress.getLocalHost();
26             System.out.println(localHostAddress);
27         }
28     }
29 }

```

API `java.net.InetAddress 1.0`

- `static InetAddress getByName(String host)`
为给定的主机名创建一个 `InetAddress` 对象，或者一个包含了该主机名所对应的所有因特网地址的数组。
- `static InetAddress[] getAllByName(String host)`
为本地主机创建一个 `InetAddress` 对象。
- `static InetAddress getLocalHost()`
返回一个包含数字型地址的字节数组。
- `byte[] getAddress()`
返回一个由十进制数组成的字符串，各数字间用圆点符号隔开，例如，“129.6.15.28”。
- `String getHostName()`
返回主机名。

4.2 实现服务器

在上一节中，我们已经实现了一个基本的网络客户端，并且用它从因特网上获取了数据。在这一节中，我们将实现一个简单的服务器，它可以向客户端发送信息。

4.2.1 服务器套接字

一旦启动了服务器程序，它便会等待某个客户端连接到它的端口。在我们的示例程序

中，我们选择端口号 8189，因为所有标准服务都不使用这个端口。`ServerSocket` 类用于建立套接字。在我们的示例中，下面这行命令：

```
var s = new ServerSocket(8189);
```

用于建立一个负责监控端口 8189 的服务器。以下命令：

```
- Socket incoming = s.accept();
```

用于告诉程序不停地等待，直到有客户端连接到这个端口。一旦有人通过网络发送了正确的连接请求，并以此连接到了端口上，该方法就会返回一个表示连接已经建立的 `Socket` 对象。你可以使用这个对象来得到输入流和输出流，代码如下：

```
InputStream inStream = incoming.getInputStream();
OutputStream outStream = incoming.getOutputStream();
```

服务器发送给服务器输出流的所有信息都会成为客户端程序的输入，同时来自客户端程序的所有输出都会被包含在服务器输入流中。

因为在本章的所有示例程序中，我们都要通过套接字来发送文本，所以我们将流转换成扫描器和写入器。

```
var in = new Scanner(inStream, StandardCharsets.UTF_8);
var out = new PrintWriter(new OutputStreamWriter(outStream, StandardCharsets.UTF_8),
    true /* autoFlush */);
```

以下代码将给客户端发送一条问候信息：

```
out.println("Hello! Enter BYE to exit.");
```

当使用 telnet 通过端口 8189 连接到这个服务器程序时，将会在终端屏幕上看到上述问候信息。

在这个简单的服务器程序中，它只是读取客户端输入，每次读取一行，并回送这一行。这表明程序接收到了客户端的输入。当然，实际应用中的服务器都会对输入进行计算并返回处理结果。

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

在代码的最后，我们关闭了连接进来的套接字。

```
incoming.close();
```

这就是整个示例代码的大致情况。每一个服务器程序，比如一个 HTTP Web 服务器，都会不间断地执行下面这个循环：

1. 通过输入数据流从客户端接收一个命令（“get me this information”）。
2. 解码这个客户端命令。
3. 收集客户端所请求的信息。
4. 通过输出数据流发送信息给客户端。

程序清单 4-3 给出了这个程序的完整代码。

程序清单 4-3 server/EchoServer.java

```

1 package server;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.*;
7
8 /**
9 * This program implements a simple server that listens to port 8189 and echoes back all
10 * client input.
11 * @version 1.22 2018-03-17
12 * @author Cay Horstmann
13 */
14 public class EchoServer
15 {
16     public static void main(String[] args) throws IOException
17     {
18         // establish server socket
19         try (var s = new ServerSocket(8189))
20         {
21             // wait for client connection
22             try (Socket incoming = s.accept())
23             {
24                 InputStream inStream = incoming.getInputStream();
25                 OutputStream outStream = incoming.getOutputStream();
26
27                 try (var in = new Scanner(inStream, StandardCharsets.UTF_8))
28                 {
29                     var out = new PrintWriter(
30                         new OutputStreamWriter(outStream, StandardCharsets.UTF_8),
31                         true /* autoFlush */);
32
33                     out.println("Hello! Enter BYE to exit.");
34
35                     // echo client input
36                     var done = false;
37                     while (!done && in.hasNextLine())
38                     {
39                         String line = in.nextLine();
40                         out.println("Echo: " + line);
41                         if (line.trim().equals("BYE")) done = true;
42                     }
43                 }
44             }
45         }
46     }
47 }
```

想要试一下这个例子，就请编译并运行这个程序。然后使用 telnet 连接到服务器 localhost（或 IP 地址 127.0.0.1）和端口 8189。

如果你直接连接到因特网上，那么世界上任何人都可以访问到你的回送服务器，只要他

们知道你的 IP 地址和端口号。

当你连接到该端口时，将看到如图 4-4 所示的信息：

Hello! Enter BYE to exit.

```

Terminal
File Edit View Terminal Help
-$ telnet localhost 8189
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello! Enter BYE to exit.
Hello Sailor!
Echo: Hello Sailor!
BYE
Echo: BYE
Connection closed by foreign host.
-$

```

图 4-4 访问一个回送服务器

可以随意键入一条信息，然后观察屏幕上的回送信息。输入 BYE（全为大写字母）可以断开连接，同时，服务器程序也会终止运行。

API **java.net.ServerSocket 1.0**

- `ServerSocket(int port)`

创建一个监听端口的服务器套接字。

- `Socket accept()`

等待连接。该方法阻塞（即，使之空闲）当前线程直到建立连接为止。该方法返回一个 `Socket` 对象，程序可以通过这个对象与连接中的客户端进行通信。

- `void close()`

关闭服务器套接字。

4.2.2 为多个客户端服务

前面例子中的简单服务器存在一个问题。假设我们希望有多个客户端同时连接到我们的服务器上。通常，服务器总是不间断地运行在服务器计算机上，来自整个因特网的用户希望同时使用服务器。前面的简单服务器会提供对客户端连接的支持，使得任何一个客户端都可以因长时间地连接服务而独占服务，其实我们可以运用线程的魔力把这个问题解决得更好。

每当程序建立一个新的套接字连接，也就是说当调用 `accept()` 时，将会启动一个新的线程来处理服务器和该客户端之间的连接，而主程序将立即返回并等待下一个连接。为了实现这种机制，服务器应该具有类似以下代码的循环操作：

```

while (true)
{
    Socket incoming = s.accept();
}

```

```

var r = new ThreadedEchoHandler(incoming);

var t = new Thread(r);
t.start();
}

```

ThreadedEchoHandler 类实现了 Runnable 接口，而且在它的 run 方法中包含了与客户端循环通信的代码。

```

class ThreadedEchoHandler implements Runnable
{
    ...
    public void run()
    {
        try (InputStream inStream = incoming.getInputStream();
             OutputStream outStream = incoming.getOutputStream())
        {
            Process input and send response
        }
        catch(IOException e)
        {
            Handle exception
        }
    }
}

```

由于每一个连接都会启动一个新的线程，因而多个客户端就可以同时连接到服务器了。对此可以做个简单的测试：

1. 编译和运行服务器程序（程序清单 4-4）。
2. 如图 4-5 所示打开数个 telnet 窗口。
3. 在这些窗口之间切换，并键入命令。注意你可以同时通过这些窗口进行通信。
4. 当完成之后，切换到启动服务器程序的窗口，并使用 CTRL+C 强行关闭它。

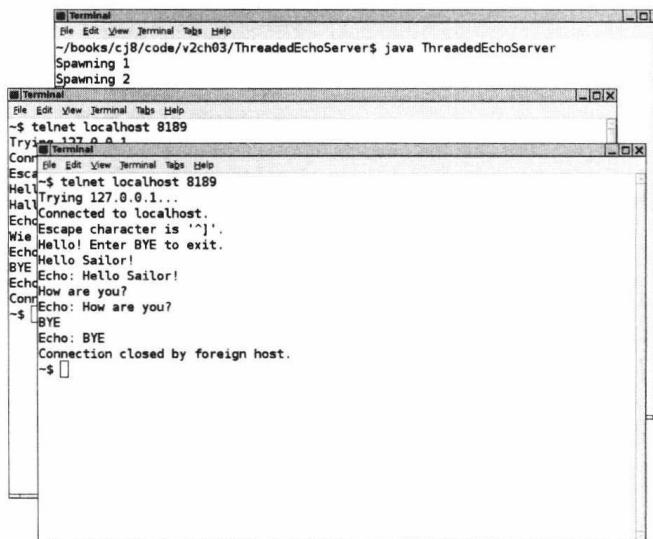


图 4-5 多个同时通信的 telnet 窗口

注释：在这个程序中，我们为每个连接生成一个单独的线程。这种方法并不能满足高性能服务器的要求。为使服务器实现更高的吞吐量，可以使用java.nio包中一些特性。详情请参见以下链接：<http://www.ibm.com/developerworks/java/library/j-javaio>。

程序清单 4-4 threaded/ThreadedEchoServer.java

```

1 package threaded;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.*;
7
8 /**
9  * This program implements a multithreaded server that listens to port 8189 and echoes back
10 * all client input.
11 * @author Cay Horstmann
12 * @version 1.23 2018-03-17
13 */
14 public class ThreadedEchoServer
15 {
16     public static void main(String[] args )
17     {
18         try (var s = new ServerSocket(8189))
19         {
20             int i = 1;
21
22             while (true)
23             {
24                 Socket incoming = s.accept();
25                 System.out.println("Spawning " + i);
26                 Runnable r = new ThreadedEchoHandler(incoming);
27                 var t = new Thread(r);
28                 t.start();
29                 i++;
30             }
31         }
32         catch (IOException e)
33         {
34             e.printStackTrace();
35         }
36     }
37 }
38
39 /**
40  * This class handles the client input for one server socket connection.
41 */
42 class ThreadedEchoHandler implements Runnable
43 {
44     private Socket incoming;
45
46     /**
47      Constructs a handler.

```

```

48     @param incomingSocket the incoming socket
49     */
50     public ThreadedEchoHandler(Socket incomingSocket)
51     {
52         incoming = incomingSocket;
53     }
54
55     public void run()
56     {
57         try (InputStream inStream = incoming.getInputStream();
58              OutputStream outStream = incoming.getOutputStream();
59              var in = new Scanner(inStream, StandardCharsets.UTF_8);
60              var out = new PrintWriter(
61                  new OutputStreamWriter(outStream, StandardCharsets.UTF_8),
62                  true /* autoFlush */))
63         {
64             out.println( "Hello! Enter BYE to exit." );
65
66             // echo client input
67             var done = false;
68             while (!done && in.hasNextLine())
69             {
70                 String line = in.nextLine();
71                 out.println("Echo: " + line);
72                 if (line.trim().equals("BYE"))
73                     done = true;
74             }
75         }
76         catch (IOException e)
77         {
78             e.printStackTrace();
79         }
80     }
81 }

```

4.2.3 半关闭

半关闭 (half-close) 提供了这样一种能力：套接字连接的一端可以终止其输出，同时仍旧可以接收来自另一端的数据。

这是一种很典型的情况，例如我们在向服务器传输数据，但是一开始并不知道要传输多少数据。在向文件写数据时，我们只需在数据写入后关闭文件即可。但是，如果关闭一个套接字，那么与服务器的连接将立刻断开，因而也就无法读取服务器的响应了。

使用半关闭的方法就可以解决上述问题。可以通过关闭一个套接字的输出流来表示发送给服务器的请求数据已经结束，但是必须保持输入流处于打开状态。

如下代码演示了如何在客户端使用半关闭方法：

```

try (var socket = new Socket(host, port))
{
    var in = new Scanner(socket.getInputStream(), StandardCharsets.UTF_8);
    var writer = new PrintWriter(socket.getOutputStream());

```

```

// send request data
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// now socket is half-closed
// read response data
while (in.hasNextLine() != null)
{
    String line = in.nextLine();
    ...
}
}

```

服务器端将读取输入信息，直至到达输入流的结尾，然后它再发送响应。

当然，该协议只适用于一站式（one-shot）的服务，例如 HTTP 服务，在这种服务中，客户端连接服务器，发送一个请求，捕获响应信息，然后断开连接。

API `java.net.Socket 1.0`

- `void shutdownOutput() 1.3`

将输出流设为“流结束”。

- `void shutdownInput() 1.3`

将输入流设为“流结束”。

- `boolean isOutputShutdown() 1.4`

如果输出已被关闭，则返回 `true`。

- `boolean isInputShutdown() 1.4`

如果输入已被关闭，则返回 `true`。

4.2.4 可中断套接字

当连接到一个套接字时，当前线程将会被阻塞直到建立连接或产生超时为止。同样地，当通过套接字读数据时，当前线程也会被阻塞直到操作成功或产生超时为止。

在交互式的应用中，也许会考虑为用户提供一个选项，用以取消那些看似不会产生结果的连接。但是，当线程因套接字无法响应而发生阻塞时，则无法通过调用 `interrupt` 来解除阻塞。

为了中断套接字操作，可以使用 `java.nio` 包提供的一个特性——`SocketChannel` 类。可以使用如下方法打开 `SocketChannel`：

```
SocketChannel channel = SocketChannel.open(new InetSocketAddress(host, port));
```

通道（channel）并没有与之相关联的流。实际上，它所拥有的 `read` 和 `write` 方法都是通过使用 `Buffer` 对象来实现的（关于 NIO 缓冲区的相关信息请参见第 2 章）。`ReadableByteChannel` 接口和 `WritableByteChannel` 接口都声明了这两个方法。

如果不处理缓冲区，可以使用 `Scanner` 类从 `SocketChannel` 中读取信息，因为 `Scanner` 有一个带 `ReadableByteChannel` 参数的构造器：

```
var in = new Scanner(channel, StandardCharsets.UTF_8);
```

通过调用静态方法 `Channels.newOutputStream`，可以将通道转换成输出流。

```
OutputStream outStream = Channels.newOutputStream(channel);
```

上述操作就是所有要做的事情。当线程正在执行打开、读取或写入操作时，如果线程发生中断，那么这些操作将不会陷入阻塞，而是以抛出异常的方式结束。

程序清单 4-5 的程序对比了可中断套接字和阻塞套接字：服务器将连续发送数字，并在每发送十个数字之后停滞一下。点击两个按钮中的任何一个，都会启动一个线程来连接服务器并打印输出。第一个线程使用可中断套接字，而第二个线程使用阻塞套接字。如果在第一批的十个数字的读取过程中点击“Cancel”按钮，这两个线程都会中断。

程序清单 4-5 interruptible/InterruptibleSocketTest.java

```

1 package interruptible;
2
3 import java.awt.*;
4 import java.awt.event.*;
5 import java.util.*;
6 import java.net.*;
7 import java.io.*;
8 import java.nio.charset.*;
9 import java.nio.channels.*;
10 import javax.swing.*;
11
12 /**
13 * This program shows how to interrupt a socket channel.
14 * @author Cay Horstmann
15 * @version 1.05 2018-03-17
16 */
17 public class InterruptibleSocketTest
18 {
19     public static void main(String[] args)
20     {
21         EventQueue.invokeLater(() ->
22         {
23             var frame = new InterruptibleSocketFrame();
24             frame.setTitle("InterruptibleSocketTest");
25             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26             frame.setVisible(true);
27         });
28     }
29 }
30
31 class InterruptibleSocketFrame extends JFrame
32 {
33     private Scanner in;
34     private JButton interruptibleButton;
35     private JButton blockingButton;
36     private JButton cancelButton;
37     private JTextArea messages;
38     private TestServer server;

```

```
39     private Thread connectThread;
40
41     public InterruptibleSocketFrame()
42     {
43         var northPanel = new JPanel();
44         add(northPanel, BorderLayout.NORTH);
45
46         final int TEXT_ROWS = 20;
47         final int TEXT_COLUMNS = 60;
48         messages = new JTextArea(TEXT_ROWS, TEXT_COLUMNS);
49         add(new JScrollPane(messages));
50
51         interruptibleButton = new JButton("Interruptible");
52         blockingButton = new JButton("Blocking");
53
54         northPanel.add(interruptibleButton);
55         northPanel.add(blockingButton);
56
57         interruptibleButton.addActionListener(event ->
58             {
59                 interruptibleButton.setEnabled(false);
60                 blockingButton.setEnabled(false);
61                 cancelButton.setEnabled(true);
62                 connectThread = new Thread(() ->
63                     {
64                         try
65                         {
66                             connectInterruptibly();
67                         }
68                         catch (IOException e)
69                         {
70                             messages.append("\nInterruptibleSocketTest.connectInterruptibly: " + e);
71                         }
72                     });
73                 connectThread.start();
74             });
75
76         blockingButton.addActionListener(event ->
77             {
78                 interruptibleButton.setEnabled(false);
79                 blockingButton.setEnabled(false);
80                 cancelButton.setEnabled(true);
81                 connectThread = new Thread(() ->
82                     {
83                         try
84                         {
85                             connectBlocking();
86                         }
87                         catch (IOException e)
88                         {
89                             messages.append("\nInterruptibleSocketTest.connectBlocking: " + e);
90                         }
91                     });
92                 connectThread.start();
93             });
94     }
95 }
```

```
93         });
94
95     cancelButton = new JButton("Cancel");
96     cancelButton.setEnabled(false);
97     northPanel.add(cancelButton);
98     cancelButton.addActionListener(event ->
99     {
100         connectThread.interrupt();
101         cancelButton.setEnabled(false);
102     });
103     server = new TestServer();
104     new Thread(server).start();
105     pack();
106 }
107
108 /**
109 * Connects to the test server, using interruptible I/O
110 */
111 public void connectInterruptibly() throws IOException
112 {
113     messages.append("Interruptible:\n");
114     try (SocketChannel channel
115          = SocketChannel.open(new InetSocketAddress("localhost", 8189)))
116     {
117         in = new Scanner(channel, StandardCharsets.UTF_8);
118         while (!Thread.currentThread().isInterrupted())
119         {
120             messages.append("Reading ");
121             if (in.hasNextLine())
122             {
123                 String line = in.nextLine();
124                 messages.append(line);
125                 messages.append("\n");
126             }
127         }
128     }
129     finally
130     {
131         EventQueue.invokeLater(() ->
132         {
133             messages.append("Channel closed\n");
134             interruptibleButton.setEnabled(true);
135             blockingButton.setEnabled(true);
136         });
137     }
138 }
139
140 /**
141 * Connects to the test server, using blocking I/O
142 */
143 public void connectBlocking() throws IOException
144 {
145     messages.append("Blocking:\n");
146     try (var sock = new Socket("localhost", 8189))
```

```
147     {
148         in = new Scanner(sock.getInputStream(), StandardCharsets.UTF_8);
149         while (!Thread.currentThread().isInterrupted())
150         {
151             messages.append("Reading ");
152             if (in.hasNextLine())
153             {
154                 String line = in.nextLine();
155                 messages.append(line);
156                 messages.append("\n");
157             }
158         }
159     }
160     finally
161     {
162         EventQueue.invokeLater(() ->
163         {
164             messages.append("Socket closed\n");
165             interruptibleButton.setEnabled(true);
166             blockingButton.setEnabled(true);
167         });
168     }
169 }
170 /**
171 * A multithreaded server that listens to port 8189 and sends numbers to the client,
172 * simulating a hanging server after 10 numbers.
173 */
174 class TestServer implements Runnable
175 {
176     public void run()
177     {
178         try (var s = new ServerSocket(8189))
179         {
180             while (true)
181             {
182                 Socket incoming = s.accept();
183                 Runnable r = new TestServerHandler(incoming);
184                 new Thread(r).start();
185             }
186         }
187     }
188     catch (IOException e)
189     {
190         messages.append("\nTestServer.run: " + e);
191     }
192 }
193 /**
194 * This class handles the client input for one server socket connection.
195 */
196 class TestServerHandler implements Runnable
197 {
198     private Socket incoming;
```

```

201     private int counter;
202
203     /**
204      * Constructs a handler.
205      * @param i the incoming socket
206      */
207     public TestServerHandler(Socket i)
208     {
209         incoming = i;
210     }
211
212     public void run()
213     {
214         try
215         {
216             try
217             {
218                 OutputStream outStream = incoming.getOutputStream();
219                 var out = new PrintWriter(
220                     new OutputStreamWriter(outStream, StandardCharsets.UTF_8),
221                     true /* autoFlush */);
222                 while (counter < 100)
223                 {
224                     counter++;
225                     if (counter <= 10) out.println(counter);
226                     Thread.sleep(100);
227                 }
228             }
229             finally
230             {
231                 incoming.close();
232                 messages.append("Closing server\n");
233             }
234         }
235         catch (Exception e)
236         {
237             messages.append("\nTestServerHandler.run: " + e);
238         }
239     }
240 }
241 }
```

但是，在第一批十个数字之后，就只能中断第一个线程了，第二个线程将保持阻塞直到服务器最终关闭连接（参见图 4-6）。

java.net.InetSocketAddress 1.4

- `InetSocketAddress(String hostname, int port)`

用给定的主机和端口参数创建一个地址对象，并在创建过程中解析主机名。如果主机名不能被解析，那么该地址对象的 `unresolved` 属性将被设为 `true`。

- `boolean isUnresolved()`

如果不能解析该地址对象，则返回 `true`。

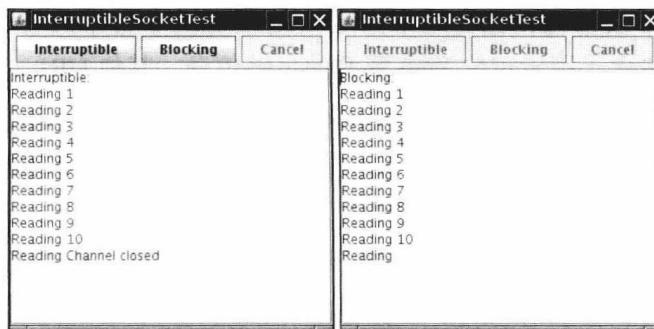


图 4-6 中断一个套接字

API `java.nio.channels.SocketChannel 1.4`

- `static SocketChannel open(SocketAddress address)`

打开一个套接字通道，并将其连接到远程地址。

API `java.nio.channels.Channels 1.4`

- `static InputStream newInputStream(ReadableByteChannel channel)`

创建一个输入流，用以从指定的通道读取数据。

- `static OutputStream newOutputStream(WritableByteChannel channel)`

创建一个输出流，用以向指定的通道写入数据。

4.3 获取 Web 数据

为了在 Java 程序中访问 Web 服务器，你可能希望在更高的级别上进行处理，而不只是创建套接字连接和发送 HTTP 请求。在下面的各个小节中，我们将讨论专用于此目的的 Java 类库中的各个类。

4.3.1 URL 和 URI

URL 和URLConnection 类封装了大量复杂的实现细节，这些细节涉及如何从远程站点获取信息。例如，可以自一个字符串构建一个 URL 对象：

```
var url = new URL(urlString);
```

如果只是想获得该资源的内容，可以使用 URL 类中的 `openStream` 方法。该方法将产生一个 `InputStream` 对象，然后就可以按照一般的用法来使用这个对象了，比如用它构建一个 `Scanner` 对象：

```
InputStream inStream = url.openStream();
var in = new Scanner(inStream, StandardCharsets.UTF_8);
```

`java.net` 包对统一资源定位符 (Uniform Resource Locator, URL) 和统一资源标识符 (Uniform Resource Identifier, URI) 进行了非常有用的区别。

URI 是个纯粹的语法结构，包含用来指定 Web 资源的字符串的各种组成部分。URL 是 URI 的一个特例，它包含了用于定位 Web 资源的足够信息。其他 URI，比如

```
mailto:cay@horstmann.com
```

则不属于定位符，因为根据该标识符我们无法定位任何数据。像这样的 URI 我们称之为 URN (uniform resource name，统一资源名称)。

在 Java 类库中，URI 类并不包含任何用于访问资源的方法，它的唯一作用就是解析。但是，URL 类可以打开一个连接到资源的流。因此，URL 类只能作用于那些 Java 类库知道该如何处理的模式，例如 http:、https:、ftp:、本地文件系统 (file:) 和 JAR 文件 (jar:)。

要想了解为什么对 URI 进行解析并非小事一桩，那么考虑一下 URL 会变得多么复杂。例如，

```
http://google.com?q=Beach+Chalet  
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

URI 规范给出了标记这些标识符的规则。一个 URI 具有以下句法：

```
[scheme:]schemeSpecificPart[#fragment]
```

上式中，[...] 表示可选部分，并且 : 和 # 可以被包含在标识符内。

包含 *scheme:* 部分的 URI 称为绝对 URI。否则，称为相对 URI。

如果绝对 URI 的 *schemeSpecificPart* 不是以 / 开头的，我们就称它是不透明的。例如：

```
mailto:cay@horstmann.com
```

所有绝对的透明 URI 和所有相对 URI 都是分层的 (hierarchical)。例如：

```
http://horstmann.com/index.html  
../../../java/net/Socket.html#Socket()
```

一个分层 URI 的 *schemeSpecificPart* 具有以下结构：

```
[//authority][path][?query]
```

在这里，[...] 同样表示可选的部分。

对于那些基于服务器的 URI，*authority* 部分具有以下形式：

```
[user-info@]host[:port]
```

port 必须是一个整数。

RFC 2396 (标准化 URI 的文献) 还支持一种基于注册表的机制，此时 *authority* 采用了一种不同的格式。不过，这种情况并不常见。

URI 类的作用之一是解析标识符并将它分解成各种不同的组成部分。你可以用以下方法读取它们：

```
getScheme  
getSchemeSpecificPart  
getAuthority  
getUserInfo  
getHost  
getPort  
getPath
```

```
getQuery
getFragment
```

URI 类的另一个作用是处理绝对标识符和相对标识符。如果存在一个如下的绝对 URI:

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

和一个如下的相对 URI:

```
../../java/net/Socket.html#Socket()
```

那么可以用它们组合出一个绝对 URI:

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

这个过程称为解析相对 URL。

与此相反的过程称为相对化 (relativization)。例如，假设有一个基本 URI:

```
http://docs.mycompany.com/api
```

和另一个 URI:

```
http://docs.mycompany.com/api/java/lang/String.html
```

那么相对化之后的 URI 就是:

```
java/lang/String.html
```

URI 类同时支持以下两个操作:

```
relative = base.relativize(combined);
combined = base.resolve(relative);
```

4.3.2 使用URLConnection 获取信息

如果想从某个 Web 资源获取更多信息，那么应该使用 `URLConnection` 类，通过它能够得到比基本的 URL 类更多的控制功能。

当操作一个 `URLConnection` 对象时，必须像下面这样非常小心地安排操作步骤：

1. 调用 URL 类中的 `openConnection` 方法获得 `URLConnection` 对象：

```
URLConnection connection = url.openConnection();
```

2. 使用以下方法来设置任意的请求属性：

```
setDoInput
setDoOutput
setIfModifiedSince
setUseCaches
setAllowUserInteraction
setRequestProperty
setConnectTimeout
setReadTimeout
```

我们将在本节的稍后部分以及 API 说明中讨论这些方法。

3. 调用 `connect` 方法连接远程资源：

```
connection.connect();
```

除了与服务器建立套接字连接外，该方法还可用于向服务器查询头信息（header information）。

4. 与服务器建立连接后，你可以查询头信息。`getHeaderFieldKey` 和 `getHeaderField` 这两个方法枚举了消息头的所有字段。`getHeaderFields` 方法返回一个包含了消息头中所有字段的标准 `Map` 对象。为了方便使用，以下方法可以查询各标准字段：

```
getContentType  
getContentLength  
getContentEncoding  
getDate  
getExpiration  
getLastModified
```

5. 最后，访问资源数据。使用 `getInputStream` 方法获取一个输入流用以读取信息（这个输入流与 `URL` 类中的 `openStream` 方法所返回的流相同）。另一个方法 `getContent` 在实际操作中并不是很有用。由标准内容类型（比如 `text/plain` 和 `image/gif`）所返回的对象需要使用 `com.sun` 层次结构中的类来进行处理。也可以注册自己的内容处理器，但是在本书中我们不讨论这项技术。

！ 警告：一些程序员在使用 `URLConnection` 类的过程中形成了错误的观念，他们认为 `URLConnection` 类中的 `getInputStream` 和 `getOutputStream` 方法与 `Socket` 类中的这些方法相似，但是这种想法并不十分正确。`URLConnection` 类具有很多表象之下的神奇功能，尤其在处理请求和响应消息头时。正因为如此，严格遵循建立连接的每个步骤显得非常重要。

下面将详细介绍一下 `URLConnection` 类中的一些方法。有几个方法可以在与服务器建立连接之前设置连接属性，其中最重要的是 `setDoInput` 和 `setDoOutput`。在默认情况下，建立的连接只产生从服务器读取信息的输入流，并不产生任何执行写操作的输出流。如果想获得输出流（例如，用于向一个 Web 服务器提交数据），那么需要调用：

```
connection.setDoOutput(true);
```

接下来，也许想设置某些请求头（request header）。请求头是与请求命令一起被发送到服务器的。例如：

```
GET www.server.com/index.html HTTP/1.0  
Referer: http://www.somewhere.com/links.html  
Proxy-Connection: Keep-Alive  
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)  
Host: www.server.com  
Accept: text/html, image/gif, image/jpeg, image/png, */*  
Accept-Language: en  
Accept-Charset: iso-8859-1,*;utf-8  
Cookie: orangemilano=192218887821987
```

`setIfModifiedSince` 方法用于告诉连接你只对自某个特定日期以来被修改过的数据感兴趣。

最后我们再介绍一个总览全局的方法：`setRequestProperty`，它可以用来设置对特定协议起作用的任何“名 – 值（name/value）对”。关于 HTTP 请求头的格式，请参见 RFC 2616，其

中的某些参数没有很好地建档，它们通常在程序员之间口头传授。例如，如果你想访问一个有密码保护的 Web 页，那么就必须按如下步骤操作：

1. 将用户名、冒号和密码以字符串形式连接在一起。

```
String input = username + ":" + password;
```

2. 计算上一步骤所得字符串的 Base64 编码。(Base64 编码用于将字节序列编码成可打印的 ASCII 字符序列。)

```
Base64.Encoder encoder = Base64.getEncoder();
String encoding = encoder.encodeToString(input.getBytes(StandardCharsets.UTF_8));
```

3. 用 "Authorization" 这个名字和 "Basic"+encoding 的值调用 setRequestProperty 方法。

```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```

 **提示：**我们上面介绍的是如何访问一个有密码保护的 Web 页。如果想要通过 FTP 访问一个有密码保护的文件时，则需要采用一种完全不同的方法，即构建如下格式的 URL：

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

一旦调用了 connect 方法，就可以查询响应头信息了。首先，我们将介绍如何枚举所有响应头的字段。似乎是为了展示自己的个性，该类的实现者引入了另一种迭代协议。调用如下方法：

```
String key = connection.getHeaderFieldKey(n);
```

可以获得响应头的第 n 个键，其中 n 从 1 开始！如果 n 为 0 或大于消息头的字段总数，该方法将返回 null 值。没有哪种方法可以返回字段的数量，必须反复调用 getHeaderFieldKey 方法直到返回 null 为止。同样地，调用以下方法：

```
String value = connection.getHeaderField(n);
```

可以得到第 n 个值。

getHeaderFields 方法可以返回一个封装了响应头字段的 Map 对象。

```
Map<String, List<String>> headerFields = connection.getHeaderFields();
```

下面是一组来自典型的 HTTP 请求的响应头字段。

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

 **注释：**可以用 connection.getHeaderField(0) 或 headerFields.get(null) 获取响应状态行（例如 "HTTP/1.1 200 OK"）。

为了简便起见，Java 提供了 6 个方法用以访问最常用的消息头类型的值，并在需要的时候将它们转换成数字类型，这些方法的详细信息请参见表 4-1。返回类型为 long 的方法返回的是从格林尼治时间 1970 年 1 月 1 日开始计算的秒数。

表 4-1 用于访问响应头值的简便方法

键名	方法名	返回类型
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

通过程序清单 4-6 的程序，可以对 URL 连接做一些试验。程序运行起来后，请在命令行中输入一个 URL 以及用户名和密码（可选），例如：

```
java urlConnection.URLConnectionTest http://www.yourserver.com user password
```

该程序将输出以下内容：

- 消息头中的所有键和值。
- 表 4-1 中 6 个简便方法的返回值。
- 被请求资源的前 10 行信息。

程序清单 4-6 urlConnection/URLConnectionTest.java

```

1 package urlConnection;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.util.*;
7
8 /**
9  * This program connects to an URL and displays the response header data and the first
10 * 10 lines of the requested data.
11 *
12 * Supply the URL and an optional username and password (for HTTP basic authentication) on the
13 * command line.
14 * @version 1.12 2018-03-17
15 * @author Cay Horstmann
16 */
17 public class URLConnectionTest
18 {
19     public static void main(String[] args)
20     {
21         try
22         {
23             String urlName;
24             if (args.length > 0) urlName = args[0];

```

```
25     else urlName = "http://horstmann.com";
26
27     var url = new URL(urlName);
28     URLConnection connection = url.openConnection();
29
30     // set username, password if specified on command line
31
32     if (args.length > 2)
33     {
34         String username = args[1];
35         String password = args[2];
36         String input = username + ":" + password;
37         Base64.Encoder encoder = Base64.getEncoder();
38         String encoding = encoder.encodeToString(input.getBytes(StandardCharsets.UTF_8));
39         connection.setRequestProperty("Authorization", "Basic " + encoding);
40     }
41
42     connection.connect();
43
44     // print header fields
45
46     Map<String, List<String>> headers = connection.getHeaderFields();
47     for (Map.Entry<String, List<String>> entry : headers.entrySet())
48     {
49         String key = entry.getKey();
50         for (String value : entry.getValue())
51             System.out.println(key + ": " + value);
52     }
53
54     // print convenience functions
55
56     System.out.println("-----");
57     System.out.println("getContentType: " + connection.getContentType());
58     System.out.println("getContentLength: " + connection.getContentLength());
59     System.out.println("getContentEncoding: " + connection.getContentEncoding());
60     System.out.println("getDate: " + connection.getDate());
61     System.out.println("getExpiration: " + connection.getExpiration());
62     System.out.println("getLastModified: " + connection.getLastModified());
63     System.out.println("-----");
64
65     String encoding = connection.getContentEncoding();
66     if (encoding == null) encoding = "UTF-8";
67     try (var in = new Scanner(connection.getInputStream(), encoding))
68     {
69         // print first ten lines of contents
70
71         for (int n = 1; in.hasNextLine() && n <= 10; n++)
72             System.out.println(in.nextLine());
73         if (in.hasNextLine()) System.out.println("... ");
74     }
75 }
76 catch (IOException e)
77 {
78     e.printStackTrace();
```

```

79      }
80  }
81 }
```

API **java.net.URL 1.0**

- `InputStream openStream()`
打开一个用于读取资源数据的输入流。
- `URLConnection openConnection()`
返回一个 `URLConnection` 对象，该对象负责管理与资源之间的连接。

API **java.netURLConnection 1.0**

- `void setDoInput(boolean doInput)`
- `boolean getDoInput()`
如果 `doInput` 为 `true`，那么用户可以接收来自该 `URLConnection` 的输入。
- `void setDoOutput(boolean doOutput)`
- `boolean getDoOutput()`
如果 `doOutput` 为 `true`，那么用户可以将输出发送到该 `URLConnection`。
- `void setIfModifiedSince(long time)`
- `long getIfModifiedSince()`
属性 `ifModifiedSince` 用于配置该 `URLConnection` 对象，使它只获取那些自从某个给定时间以来被修改过的数据。调用方法时需要传入的 `time` 参数指的是从格林尼治时间 1970 年 1 月 1 日午夜开始计算的秒数。
- `void setConnectTimeout(int timeout) 5.0`
- `int getConnectTimeout() 5.0`
设置或得到连接超时时限（单位：毫秒）。如果在连接建立之前就已经达到了超时的时限，那么相关联的输入流的 `connect` 方法就会抛出一个 `SocketTimeoutException` 异常。
- `void setReadTimeout(int timeout) 5.0`
- `int getReadTimeout() 5.0`
设置读取数据的超时时限（单位：毫秒）。如果在一个读操作成功之前就已经达到了超时的时限，那么 `read` 方法就会抛出一个 `SocketTimeoutException` 异常。
- `void setRequestProperty(String key, String value)`
设置请求头的一个字段。
- `Map<String, List<String>> getRequestProperties() 1.4`
返回请求头属性的一个映射表。相同的键对应的所有值被放置在同一个列表中。
- `void connect()`
连接远程资源并获取响应头信息。
- `Map<String, List<String>> getHeaderFields() 1.4`

返回响应头的一个映射表。相同的键对应的所有值被放置在同一个列表中。

- `String getHeaderFieldKey(int n)`

得到响应头第 `n` 个字段的键。如果 `n` 小于等于 0 或大于响应头字段的总数，则该方法返回 `null` 值。

- `String getHeaderField(int n)`

得到响应头第 `n` 个字段的值。如果 `n` 小于等于 0 或大于响应头字段的总数，则该方法返回 `null` 值。

- `int getContentLength()`

如果内容长度可获得，则返回该长度值，否则返回 -1。

- `String getContentType()`

获取内容的类型，比如 `text/plain` 或 `image/gif`。

- `String getContentEncoding()`

获取内容的编码机制，比如 `gzip`。这个值不太常用，因为默认的 `identity` 编码机制并不是用 `Content-Encoding` 头来设定的。

- `long getDate()`

- `long getExpiration()`

- `long getLastModified()`

获取创建日期、过期日以及最后一次被修改的日期。这些日期指的是从格林尼治时间 1970 年 1 月 1 日午夜开始计算的秒数。

- `InputStream getInputStream()`

- `OutputStream getOutputStream()`

返回从资源读取信息或向资源写入信息的流。

- `Object getContent()`

选择适当的内容处理器，以便读取资源数据并将它转换成对象。该方法对于读取诸如 `text/plain` 或 `image/gif` 之类的标准内容类型并没有什么用处，除非你安装了自己的内容处理器。

4.3.3 提交表单数据

在上一节中，我们介绍了如何从 Web 服务器读取数据。现在，我们将介绍如何让程序再将数据反馈回 Web 服务器和那些被 Web 服务器调用的程序。

为了将信息从 Web 浏览器发送到 Web 服务器，用户需要填写一个类似图 4-7 中所示的表单。

当用户点击提交按钮时，文本框中的文本以及复选框、单选按钮和其他输入元素的设定值都被发送到了 Web 服务器。此时，Web 服务器调用程序对用户的输入进行处理。

有许多技术可以让 Web 服务器实现对程序的调用。其中最广人所知的是 Java Servlet、JavaServer Face、微软的 ASP（Active Server Pages，动态服务器主页）以及 CGI（Common Gateway Interface，通用网关接口）脚本。

图 4-7 HTML 表单

服务器端程序用于处理表单数据并生成另一个 HTML 页，该页会被 Web 服务器发回给浏览器，这个操作过程我们在图 4-8 中做了说明。返回给浏览器的响应页可以包含新的信息（例如，信息检索程序中的响应页）或者只是一个确认。之后，Web 浏览器将显示响应页。

我们不会在本书中介绍应该如何实现服务器端程序，而是将侧重点放在如何编写客户端程序使之与已有的服务器端程序进行交互。

当表单数据被发送到 Web 服务器时，数据到底由谁来解释并不重要，可能是 Servlet 或 CGI 脚本，也可能是其他服务器端技术。客户端以标准格式将数据发送给 Web 服务器，而 Web 服务器则负责将数据传递给具体的程序以产生响应。

在向 Web 服务器发送信息时，通常有两个命令会被用到：GET 和 POST。

在使用 GET 命令时，只需将参数附在 URL 的结尾处即可。这种 URL 的格式如下：

`http://host/path?query`

其中，每个参数都具有“名字 = 值”的形式，而这些参数之间用 & 字符分隔开。参数的值将遵循下面的规则，使用 URL 编码模式进行编码：

- 保留字符 A 到 Z、a 到 z、0 到 9，以及 . - ~ _。

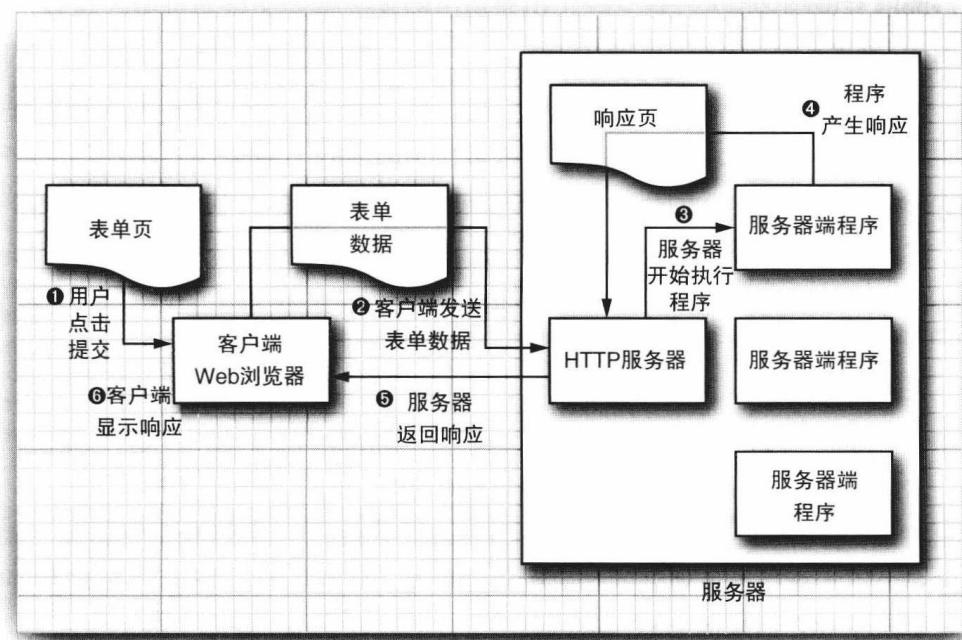


图 4-8 执行服务器端脚本过程中的数据流

- 用 + 字符替换所有的空格。
- 将其他所有字符编码为 UTF-8，并将每个字节都编码为 % 后面紧跟一个两位的十六进制数字。

例如，若要发送街道名 San Francisco, CA，可以使用 San+Francisco%2c+CA，因为十六进制数 2c（即十进制数 44）是“,”的 UTF-8 码值。

这种编码方式使得在任何中间程序中都不会混入空格和其他特殊字符。

例如，就在写作本书的时候，Google Map 网站 (www.google.com/maps) 可以接受带有两个名为 q 和 hl 参数的查询请求，这两个参数分别表示查询的位置和响应中所使用的人类语言。为了得到 1 Market Street, San Francisco, CA 的地图，并且让响应使用德语，只需访问下面的 URL 即可：

```
http://www.google.com/maps?q=1+Market+Street+San+Francisco&hl=de
```

在浏览器中出现很长的查询字符串很让人郁闷，而且老式的浏览器和代理对在 GET 请求中能够包含的字符数量做出了限制。正因为此，POST 请求经常用来处理具有大量数据的表单。在 POST 请求中，我们不会在 URL 上附着参数，而是从 `URLConnection` 中获得输出流，并将名 / 值对写入到该输出流中。我们仍旧需要对这些值进行 URL 编码，并用 & 字符将它们隔开。

下面，我们将详细介绍这个过程。在提交数据给服务器端程序之前，首先需要创建一个 `URLConnection` 对象。

```
var url = new URL("http://host/path");
URLConnection connection = url.openConnection();
```

然后，调用 `setDoOutput` 方法建立一个用于输出的连接。

```
connection.setDoOutput(true);
```

接着，调用 `getOutputStream` 方法获得一个流，可以通过这个流向服务器发送数据。如果要向服务器发送文本信息，那么可以非常方便地将流包装在 `PrintWriter` 对象中。

```
var out = new PrintWriter(connection.getOutputStream(), StandardCharsets.UTF_8);
```

现在，可以向服务器发送数据了。

```
out.print(name1 + "=" + URLEncoder.encode(value1, StandardCharsets.UTF_8) + "&");  
out.print(name2 + "=" + URLEncoder.encode(value2, StandardCharsets.UTF_8));
```

之后，关闭输出流：

```
out.close();
```

最后，调用 `getInputStream` 方法读取服务器的响应。

下面我们来实际操作一个例子。地址为 <https://tools.usps.com/zip-code-lookup.htm?byaddress> 的网站包含一个用于查找街道地址的邮政编码的表单（见图 4-7）。要想在 Java 程序中使用这个表单，需要知道 POST 请求的 URL 和参数。

你可以通过查看这个表单的 HTML 源码来获取这些信息，但是通常用网络监视器来“窥视”发出的请求会更容易一些。作为其开发工具包的组成部分，大多数浏览器都具有网络监视器。例如，图 4-9 展示了 Firefox 网络监视器向我们的示例网站提交数据时的截屏。你可以发现其中的提交 URL 以及参数名和参数值。

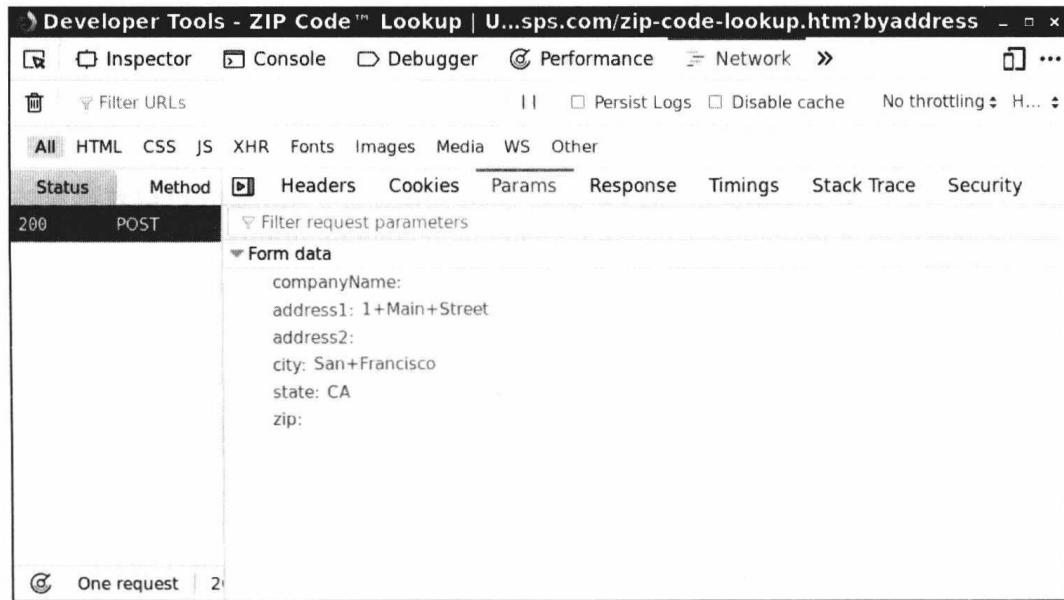


图 4-9 监视表单的提交

在提交表单数据时，HTTP 头包含了内容类型：

```
Content-Type: application/x-www-form-urlencoded
```

你还可以以其他格式提交表单。例如，发送用 JavaScript 对象表示法（JSON）表示的数据，将内容类型设置为 application/json。

POST 的头还必须包括内容长度，例如：

Content-Length: 124

程序清单 4-7 用于将 POST 数据发送给任何脚本，它将数据放在如下的 .properties 文件：

```
url=https://tools.usps.com/tools/app/ziplookup/zipByAddress
User-Agent=HTTPPie/0.9.2
address1=1 Market Street
address2=
city=San Francisco
state=CA
companyName=
.
.
```

程序清单 4-7 post/PostTest.java

```
1 package post;
2
3 import java.io.*;
4 import java.net.*;
5 import java.nio.charset.*;
6 import java.nio.file.*;
7 import java.util.*;
8
9 /**
10  * This program demonstrates how to use the URLConnection class for a POST request.
11  * @version 1.42 2018-03-17
12  * @author Cay Horstmann
13  */
14 public class PostTest
15 {
16     public static void main(String[] args) throws IOException
17     {
18         String propsFilename = args.length > 0 ? args[0] : "post/post.properties";
19         var props = new Properties();
20         try (InputStream in = Files.newInputStream(Paths.get(propsFilename)))
21         {
22             props.load(in);
23         }
24         String urlString = props.remove("url").toString();
25         Object userAgent = props.remove("User-Agent");
26         Object redirects = props.remove("redirects");
27         CookieHandler.setDefault(new CookieManager(null, CookiePolicy.ACCEPT_ALL));
28         String result = doPost(new URL(urlString), props,
29             userAgent == null ? null : userAgent.toString(),
30             redirects == null ? -1 : Integer.parseInt(redirections.toString()));
31         System.out.println(result);
32     }
33
34 /**
35  * Do an HTTP POST.
```

```

36     * @param url the URL to post to
37     * @param nameValuePairs the query parameters
38     * @param userAgent the user agent to use, or null for the default user agent
39     * @param redirects the number of redirects to follow manually, or -1 for automatic
40     * redirects
41     * @return the data returned from the server
42     */
43    public static String doPost(URL url, Map<Object, Object> nameValuePairs, String userAgent,
44                               int redirects) throws IOException
45    {
46        var connection = (HttpURLConnection) url.openConnection();
47        if (userAgent != null)
48            connection.setRequestProperty("User-Agent", userAgent);
49
50        if (redirects >= 0)
51            connection.setInstanceFollowRedirects(false);
52
53        connection.setDoOutput(true);
54
55        try (var out = new PrintWriter(connection.getOutputStream()))
56        {
57            var first = true;
58            for (Map.Entry<Object, Object> pair : nameValuePairs.entrySet())
59            {
60                if (first) first = false;
61                else out.print('&');
62                String name = pair.getKey().toString();
63                String value = pair.getValue().toString();
64                out.print(name);
65                out.print('=');
66                out.print(URLEncoder.encode(value, StandardCharsets.UTF_8));
67            }
68        }
69        String encoding = connection.getContentEncoding();
70        if (encoding == null) encoding = "UTF-8";
71
72        if (redirects > 0)
73        {
74            int responseCode = connection.getResponseCode();
75            if (responseCode == HttpURLConnection.HTTP_MOVED_PERM
76                || responseCode == HttpURLConnection.HTTP_MOVED_TEMP
77                || responseCode == HttpURLConnection.HTTP_SEE_OTHER)
78            {
79                String location = connection.getHeaderField("Location");
80                if (location != null)
81                {
82                    URL base = connection.getURL();
83                    connection.disconnect();
84                    return doPost(new URL(base, location), nameValuePairs, userAgent,
85                                 redirects - 1);
86                }
87            }
88        }
89    }
90    else if (redirects == 0)

```

```

90     {
91         throw new IOException("Too many redirects");
92     }
93
94     var response = new StringBuilder();
95     try (var in = new Scanner(connection.getInputStream(), encoding))
96     {
97         while (in.hasNextLine())
98         {
99             response.append(in.nextLine());
100            response.append("\n");
101        }
102    }
103    catch (IOException e)
104    {
105        InputStream err = connection.getErrorStream();
106        if (err == null) throw e;
107        try (var in = new Scanner(err))
108        {
109            response.append(in.nextLine());
110            response.append("\n");
111        }
112    }
113
114    return response.toString();
115 }
116 }
```

这个程序移除了 url 和 User-Agent 项，并将其他内容都发送到了 doPost 方法。

在 doPost 方法中，我们首先打开连接并设置用户代理。(邮政编码服务在默认的 User-Agent 请求参数包含字符串 Java 时无法工作，这可能是因为邮政局不想为程序自动产生的请求服务。)

然后调用 setDoOutput(true) 并打开输出流。然后，枚举 Map 对象中的所有键和值。对每一个键 – 值对，我们发送 key、= 字符、value 和 & 分隔符：

```

out.print(key);
out.print('=');
out.print(URLEncoder.encode(value, StandardCharsets.UTF_8));
if (more pairs) out.print('&');
```

在从写出请求切换到读取响应的任何部分时，就会发生与服务器的实际交互。Content-Length 头被设置为输出的尺寸，而 Content-Type 头被设置为 application/x-www-form-urlencoded，除非指定了不同的内容类型。这些头信息和数据都被发送给服务器，然后，响应头和服务器响应会被读取，并可以被查询。在我们的示例程序中，这种切换发生在对 connection.getContentEncoding() 的调用中。

在读取响应过程中会碰到一个问题。如果服务器端出现错误，那么调用 connection.getInputStream() 时就会抛出一个 FileNotFoundException 异常。但是，此时服务器仍然会向浏览器返回一个错误页面（例如，常见的“错误 404– 找不到该页”）。为了捕捉这个错误页，可

以调用 `getErrorStream` 方法：

```
InputStream err = connection.getErrorStream();
```

注释：`getErrorStream` 方法与这个程序中的许多其他方法一样，属于 `URLConnection` 类的子类 `HttpURLConnection`。如果要创建以 `http://` 或 `https://` 开头的 URL，那么可以将所产生的连接对象强制转型为 `HttpURLConnection`。

在将 POST 数据发送给服务器时，服务器端程序产生的响应可能是 `redirect:`，后面跟着一个完全不同的 URL，该 URL 应该被调用以获取实际的信息。服务器可以这么做，因为这些信息位于他处，或者提供了一个可以作为书签标记的 URL。`HttpURLConnection` 类在大多数情况下可以处理这种重定向。

注释：如果 `cookie` 需要在重定向中从一个站点发送给另一个站点，那么可以像下面这样配置一个全局的 `cookie` 处理器：

```
CookieHandler.setDefault(new CookieManager(null, CookiePolicy.ACCEPT_ALL));
```

然后，`cookie` 就可以被正确地包含在重定向请求中了。

尽管重定向通常是自动处理的，但是有些情况下，你需要自己完成重定向。例如，在 HTTP 和 HTTPS 之间的自动重定向因为安全原因而不被支持。重定向还会因更细微的原因而失败。例如，早期版本的邮政编码服务就使用了重定向。回忆一下，我们设置了 `User-Agent` 请求参数，以便让邮局认为我们不是在通过 Java API 发送请求。尽管可以在最初的请求中将用户代理设置为其他的字符串，但是这项设置在自动重定向中并没有用到。自动重定向总是会发送包含单词 `Java` 的通用用户代理字符串。

在这些情况下，可以人工实现重定向。在连接到服务器之前，将关闭自动重定向：

```
connection.setInstanceFollowRedirects(false);
```

在发送请求之后，获取响应码：

```
int responseCode = connection.getResponseCode();
```

检查它是否是下列值之一：

```
HttpURLConnection.HTTP_MOVED_PERM  
HttpURLConnection.HTTP_MOVED_TEMP  
HttpURLConnection.HTTP_SEE_OTHER
```

如果是这些值之一，那么获取 `Location` 响应头，以获得重定向的 URL。然后，断开连接，并创建到新的 URL 的连接：

```
String location = connection.getHeaderField("Location");
if (location != null)
{
    URL base = connection.getURL();
    connection.disconnect();
    connection = (HttpURLConnection) new URL(base, location).openConnection();
    ...
}
```

每当需要从某个现有的 Web 站点查询信息时，该程序所展示的处理技术就会显得很有用。只需找出需要发送的参数，然后从回复信息中剔除 HTML 和其他不必要的信息。

API `java.net.HttpURLConnection 1.0`

- `InputStream getErrorStream()`
返回一个流，通过这个流可以读取 Web 服务器的错误信息。

API `java.net.URLEncoder 1.0`

- `static String encode(String s, String encoding) 1.4`

采用指定的字符编码模式（推荐使用“UTF-8”）对字符串 s 进行编码，并返回它的 URL 编码形式。在 URL 编码中，'A'-'Z'，'a'-'z'，'0'-'9'，'-'，'_'，'.' 和 '*' 等字符保持不变，空格被编码成'+'，所有其他字符被编码成 "%XY" 形式的字节序列，其中 0xXY 为该字节十六进制数。

API `java.net.URLDecoder 1.2`

- `static string decode(String s, String encoding) 1.4`

采用指定编码模式对已编码字符串 s 进行解码，并返回结果。

4.4 HTTP 客户端

`URLConnection` 类是在 HTTP 成为 Web 普适协议之前设计的，它提供了对大量协议的支持，但是它对 HTTP 的支持有些笨重。当做出决定要支持 HTTP/2 时，情况就很清楚了，它最好是提供一个新的客户端接口，而不是对现有 API 做重构。`HttpClient` 提供了更便捷的 API 和对 HTTP/2 的支持。在 Java 9 和 10 中，其 API 类位于 `jdk.incubator.http` 包中，使该 API 有机会成为根据用户反馈不断演化的产物。到了 Java 11，`HttpClient` 位于 `java.net.http` 包中。

 **注释：**在使用 Java 9 和 10 时，需要用下面的命令行选项来运行程序：

```
--add-modules jdk.incubator.httpclient
```

与 `URLConnection` 类相比，HTTP 客户端 API 从设计初始就提供了一种更简单的连接到 Web 服务器的机制。

`HttpClient` 对象可以发出请求并接收响应。可以通过下面的调用获取客户端：

```
HttpClient client = HttpClient.newHttpClient()
```

或者，如果需要配置客户端，可以使用像下面这样的构建器 API：

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.ALWAYS)
    .build();
```

即，获取一个构建器，调用其方法定制需要待构建的项，然后调用 `build` 方法来终结构建过程。这是一种构建不可修改对象的常见模式。

还可以遵循构建器模式来定制请求，下面是一个 Get 请求：

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("http://horstmann.com"))
    .GET()
    .build();
```

URI 是指“统一资源标识符”，在使用 HTTP 时，它与 URL 相同。但是，在 Java 中，URL 类确实有一些用来打开到某个 URL 的连接的方法，而 URI 类只关心语法（模式、主机、端口、路径、查询、片段等）。

对于 POST 请求，需要一个“体发布器”（body publisher），它会将请求数据转换为要推送的数据。有针对字符串、字节数组和文件的体发布器。例如，如果请求是 JSON 格式的，那么只需将 JSON 字符串提供给某个字符串体发布器：

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI(url))
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString(jsonString))
    .build();
```

遗憾的是，该 API 不支持对常见内容类型做上面所要求的格式化处理。程序清单 4-8 中的样例程序提供了用于表单数据和文件上传的体发布器。

在发送请求时，必须告诉客户端如何处理响应。如果只是想将体当作字符串处理，那么就可以像下面这样用 `HttpResponse.BodyHandlers.ofString()` 来发送请求：

```
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
```

`HttpResponse` 类是一个泛化类，它的类型参数表示体的类型。可以直接获取响应体字符串：

```
String bodyString = response.body();
```

还有其他的响应体处理器，可以将响应作为字节数组或输入流来获取。`BodyHandlers.ofFile(filePath)` 会产生一个处理器，将响应存储到给定的文件中，`BodyHandlers.ofFileDownload(directoryPath)` 会用 `Content-Disposition` 头中的信息将响应存入给定的目录中。最后，从 `BodyHandlers.discard()` 中获得的处理器会直接丢弃响应。

处理响应的内容并不在该 API 的考虑范围内。例如，如果收到了 JSON 数据，那么就需要某个 JSON 库来解析其中的内容。

`HttpResponse` 对象还会产生状态码与响应头。

```
int status = response.statusCode();
HttpHeaders responseHeaders = response.headers();
```

可以将 `HttpHeader` 对象转换为一个映射表：

```
Map<String, List<String>> headerMap = responseHeaders.map();
```

这个映射表的值是列表，因为在 HTTP 中，每个键都可以有多个值。

如果只想要某个特定键的值，并且知道它没有多个值，那么可以调用 `firstValue` 方法：

```
Optional<String> lastModified = headerMap.firstValue("Last-Modified");
```

这样可以得到该响应的值，或者在没有提供该值时，返回空的 Optional 对象。

可以异步地处理响应。在构建客户端时，可以提供一个执行器：

```
ExecutorService executor = Executors.newCachedThreadPool();
HttpClient client = HttpClient.newBuilder().executor(executor).build();
```

构建一个请求，然后在该客户端上调用 sendAsync 方法，就会收到一个 CompletableFuture<HttpServletResponse<T>> 对象，其中 T 是体处理器的类型。需要使用卷 I 第 12 章描述的 CompletableFuture API：

```
HttpRequest request = HttpRequest.newBuilder().uri(uri).GET().build();
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenAccept(response -> . . .);
```

 **提示：**为了启用针对 HttpClient 记录日志的功能，需要在 JDK 的 net.properties 文件中添加下面的行：

```
jdk.httpclient.HttpClient.log=all
```

除了 all，还可以指定为一个由逗号分隔的列表，其中包含 headers、requests、content、errors、ssl、trace 和 frames，后面还可以选择跟着 :control、:data、:window 或 :all。中间不要使用任何空格。

然后，将名为 jdk.httpclient.HttpClient 的日志记录器的日志级别设置为 INFO，例如，在 JDK 的 logging.properties 文件中添加下面的行：

```
jdk.httpclient.HttpClient.level=INFO
```

程序清单 4-8 client/HttpClientTest.java

```

1 package client;
2
3 import java.io.*;
4 import java.math.*;
5 import java.net.*;
6 import java.nio.charset.*;
7 import java.nio.file.*;
8 import java.util.*;
9
10 import java.net.http.*;
11 import java.net.http.HttpRequest.*;
12
13 class MoreBodyPublishers
14 {
15     public static BodyPublisher ofFormData(Map<Object, Object> data)
16     {
17         var first = true;
18         var builder = new StringBuilder();
19         for (Map.Entry<Object, Object> entry : data.entrySet())
20         {
21             if (first) first = false;
```

```

22         else builder.append("&");
23         builder.append(URLEncoder.encode(entry.getKey().toString(),
24             StandardCharsets.UTF_8));
25         builder.append("=");
26         builder.append(URLEncoder.encode(entry.getValue().toString(),
27             StandardCharsets.UTF_8));
28     }
29     return BodyPublishers.ofString(builder.toString());
30 }
31
32 private static byte[] bytes(String s) { return s.getBytes(StandardCharsets.UTF_8); }
33
34 public static BodyPublisher ofMimeMultipartData(Map<Object, Object> data, String boundary)
35     throws IOException
36 {
37     var byteArrays = new ArrayList<byte[]>();
38     byte[] separator = bytes("--" + boundary + "\nContent-Disposition: form-data; name=");
39     for (Map.Entry<Object, Object> entry : data.entrySet())
40     {
41         byteArrays.add(separator);
42
43         if (entry.getValue() instanceof Path)
44         {
45             var path = (Path) entry.getValue();
46             String mimeType = Files.probeContentType(path);
47             byteArrays.add(bytes("\r\n" + entry.getKey() + "\r\n; filename=" + path.getFileName()
48                 + "\r\nContent-Type: " + mimeType + "\r\n\r\n"));
49             byteArrays.add(Files.readAllBytes(path));
50         }
51         else
52             byteArrays.add(bytes("\r\n" + entry.getKey() + "\r\n\r\n" + entry.getValue() + "\r\n"));
53     }
54     byteArrays.add(bytes("--" + boundary + "--"));
55     return BodyPublishers.ofByteArrays(byteArrays);
56 }
57
58 public static BodyPublisher ofSimpleJSON(Map<Object, Object> data)
59 {
60     var builder = new StringBuilder();
61     builder.append("{");
62     var first = true;
63     for (Map.Entry<Object, Object> entry : data.entrySet())
64     {
65         if (first) first = false;
66         else
67             builder.append(",");
68         builder.append(jsonEscape(entry.getKey().toString())).append(": ")
69             .append(jsonEscape(entry.getValue().toString()));
70     }
71     builder.append("}");
72     return BodyPublishers.ofString(builder.toString());
73 }
74
75 private static Map<Character, String> replacements = Map.of('\b', "\\b", '\f', "\\f",

```

```

76     '\n', '\\n', '\\r', '\\r', '\\t', '\\t', '''', '\\\\'', '\\\\', '\\\\\\');
77
78     private static StringBuilder jsonEscape(String str)
79     {
80         var result = new StringBuilder("\\");
81         for (int i = 0; i < str.length(); i++)
82         {
83             char ch = str.charAt(i);
84             String replacement = replacements.get(ch);
85             if (replacement == null) result.append(ch);
86             else result.append(replacement);
87         }
88         result.append("\\");
89         return result;
90     }
91 }
92
93 public class HttpClientTest
94 {
95     public static void main(String[] args)
96         throws IOException, URISyntaxException, InterruptedException
97     {
98         System.setProperty("jdk.httpclient.HttpClient.log", "headers,errors");
99         String propsFilename = args.length > 0 ? args[0] : "client/post.properties";
100        Path propsPath = Paths.get(propsFilename);
101        var props = new Properties();
102        try (InputStream in = Files.newInputStream(propsPath))
103        {
104            props.load(in);
105        }
106        String urlString = "" + props.remove("url");
107        String contentType = "" + props.remove("Content-Type");
108        if (contentType.equals("multipart/form-data"))
109        {
110            var generator = new Random();
111            String boundary = new BigInteger(256, generator).toString();
112            contentType += ";boundary=" + boundary;
113            props.replaceAll((k, v) ->
114                v.toString().startsWith("file://")
115                    ? propsPath.getParent().resolve(Paths.get(v.toString().substring(7)))
116                    : v);
117        }
118        String result = doPost(urlString, contentType, props);
119        System.out.println(result);
120    }
121
122    public static String doPost(String url, String contentType, Map<Object, Object> data)
123        throws IOException, URISyntaxException, InterruptedException
124    {
125        HttpClient client = HttpClient.newBuilder()
126            .followRedirects(HttpClient.Redirect.ALWAYS).build();
127
128        BodyPublisher publisher = null;
129        if (contentType.startsWith("multipart/form-data"))

```

```

130     {
131         String boundary = contentType.substring(contentType.lastIndexOf("=") + 1);
132         publisher = MoreBodyPublishers.ofMimeMultipartData(data, boundary);
133     }
134     else if (contentType.equals("application/x-www-form-urlencoded"))
135         publisher = MoreBodyPublishers.ofFormData(data);
136     else
137     {
138         contentType = "application/json";
139         publisher = MoreBodyPublishers.ofSimpleJSON(data);
140     }
141
142     HttpRequest request = HttpRequest.newBuilder()
143         .uri(new URI(url))
144         .header("Content-Type", contentType)
145         .POST(publisher)
146         .build();
147     HttpResponse<String> response
148         = client.send(request, HttpResponse.BodyHandlers.ofString());
149     return response.body();
150 }
151 }
```

API `java.net.http.HttpClient` 11

- `static HttpClient newHttpClient()`
用默认配置产生一个 `HttpClient` 对象。
- `static HttpClient.Builder newBuilder()`
产生一个用于构建 `HttpClient` 对象的构建器。
- `<T> HttpResponse<T> send(HttpRequest request, HttpResponse.BodyHandler<T> responseBodyHandler)`
- `<T> CompletableFuture<HttpResponse<T>> sendAsync(HttpRequest request, HttpResponse.BodyHandler<T> responseBodyHandler)`
产生一个同步或异步的请求，并使用给定的处理器来处理响应体。

API `java.net.http.HttpClient.Builder` 11

- `HttpClient build()`
用由当前构建器配置的属性产生一个 `HttpClient` 对象。
- `HttpClient.Builder followRedirects(HttpClient.Redirect policy)`
将重定向策略设置为 `HttpClient.Redirect` 枚举中的 `ALWAYS`、`NEVER` 或 `NORMAL` 之一（仅拒绝从 HTTPS 重定向到 HTTP）
- `HttpClient.Builder executor(Executor executor)`
设置用于异步请求的执行器。

API `java.net.http.HttpRequest` 11

- `HttpRequest.Builder newBuilder()`

产生一个用于构建 `HttpRequest` 对象的构建器。

API `java.net.http.HttpRequest.Builder` 11

- `HttpRequest build()`
用由当前构建器配置的属性产生一个 `HttpRequest` 对象。
- `HttpRequest.Builder uri(URI uri)`
为当前请求设置 URI。
- `HttpRequest.Builder header(String name, String value)`
为当前请求设置请求头。
- `HttpRequest.Builder GET()`
- `HttpRequest.Builder DELETE()`
- `HttpRequest.Builder POST(HttpRequest.BodyPublisher bodyPublisher)`
- `HttpRequest.Builder PUT(HttpRequest.BodyPublisher bodyPublisher)`
为当前请求设置请求方法和请求体。

API `java.net.http.HttpResponse<T>` 11

- `T body()`
产生当前响应的体。
- `int statusCode()`
产生当前响应的状态码。
- `HttpHeaders headers()`
产生响应头。

API `java.net.http.HttpHeaders` 11

- `Map<String, List<String>> map()`
产生这些头的映射。
- `Optional<String> firstValue(String name)`
在头中具有给定名的第一个值，如果存在的话。

4.5 发送 E-mail

过去，编写程序通过创建到邮件服务器上 SMTP 专用的端口 25 来发送邮件是一件很简单的事。简单邮件传输协议用于描述 E-mail 消息的格式。一旦连接到服务器，就可以发送一个邮件报头（采用 SMTP 格式，该格式很容易生成）。紧随其后的是邮件消息。

以下是操作的详细过程。

1. 打开一个到达主机的套接字：

```
var s = new Socket("mail.yourserver.com", 25); // 25 is SMTP
var out = new PrintWriter(s.getOutputStream(), StandardCharsets.UTF_8);
```

2. 发送以下信息到打印流：

```

HELO sending host
MAIL FROM: sender e-mail address
RCPT TO: recipient e-mail address
DATA
Subject: subject
(blank line)
mail message (any number of lines)

QUIT

```

SMTP 规范 (RFC 821) 规定，每一行都要以 \r 再紧跟一个 \n 来结尾。

SMTP 曾经总是例行公事般地路由任何人的 E-mail，但是，在蠕虫泛滥的今天，许多服务器都内置了检查功能，并且只接受来自授信用户或授信 IP 地址范围的请求。其中，认证通常是通过安全套接字连接来实现的。

实现人工认证模式的代码非常冗长乏味，因此，我们将展示如何利用 JavaMail API 在 Java 程序中发送 E-mail。

可以从 www.oracle.com/technetwork/java/javamail 处下载 JavaMail，然后将它解压到硬盘上的某处。

如果要使用 JavaMail，则需要设置一些和邮件服务器相关的属性。例如，在使用 GMail 时，需要设置：

```

mail.transport.protocol=smtpls
mail.smtps.auth=true
mail.smtps.host=smtp.gmail.com
mail.smtps.user=accountname@gmail.com

```

我们的示例程序是从一个属性文件中读取这些属性值的。

出于安全的原因，我们没有将密码放在属性文件中，而是要求提示用户需要输入。

首先要读入属性文件，然后像下面这样获取一个邮件会话：

```
Session mailSession = Session.getDefaultInstance(props);
```

接着，用恰当的发送者、接受者、主题和消息文本来创建消息：

```

var message = new MimeMessage(mailSession);
message.setFrom(new InternetAddress(from));
message.addRecipient(RecipientType.T0, new InternetAddress(to));
message.setSubject(subject);
message.setText(builder.toString());

```

然后将消息发送走：

```

Transport tr = mailSession.getTransport();
tr.connect(null, password);
tr.sendMessage(message, message.getAllRecipients());
tr.close();

```

程序清单 4-9 中的程序是从具有下面这种格式的文本文件中读取消息的：

Sender

```

Recipient
Subject
Message text (any number of lines)

```

要运行该程序，需要从 <https://javaee.github.io/javamail> 下载 JavaMail 的实现，还需要 Java 激活框架（Java Activation Framework）的 JAR 文件，可以从 <http://www.oracle.com/technetwork/java/javase/jaf-135115.html> 处获得，或者可以在 Maven Central 中搜索。然后运行：

```
java -classpath .:javax.mail.jar:activation-1.1.1.jar path/to/message.txt
```

到撰写本章时为止，GMail 还不会检查信息的真实性，即你可以输入任何你喜欢的发送者。（当你下一次收到来自 president@whitehouse.gov 的 E-mail 消息邀请你盛装出席白宫南草坪的活动时，请牢记这一点，谨防上当。）

 提示：如果你搞不清楚为什么你的邮件连接无法正常工作，那么可以调用：

```
mailSession.setDebug(true);
```

并检查消息。而且，JavaMail API FAQ 也有些挺有用的调试提示。

程序清单 4-9 mail/MailTest.java

```

1 package mail;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7 import javax.mail.*;
8 import javax.mail.internet.*;
9 import javax.mail.internet.MimeMessage.RecipientType;
10
11 /**
12 * This program shows how to use JavaMail to send mail messages.
13 * @author Cay Horstmann
14 * @version 1.01 2018-03-17
15 */
16 public class MailTest
17 {
18     public static void main(String[] args) throws MessagingException, IOException
19     {
20         var props = new Properties();
21         try (InputStream in = Files.newInputStream(Paths.get("mail", "mail.properties")))
22         {
23             props.load(in);
24         }
25         List<String> lines = Files.readAllLines(Paths.get(args[0]), StandardCharsets.UTF_8);
26
27         String from = lines.get(0);
28         String to = lines.get(1);
29         String subject = lines.get(2);
30
31         var builder = new StringBuilder();

```

```
32     for (int i = 3; i < lines.size(); i++)
33     {
34         builder.append(lines.get(i));
35         builder.append("\n");
36     }
37
38     Console console = System.console();
39     var password = new String(console.readPassword("Password: "));
40
41     Session mailSession = Session.getDefaultInstance(props);
42     // mailSession.setDebug(true);
43     var message = new MimeMessage(mailSession);
44     message.setFrom(new InternetAddress(from));
45     message.addRecipient(RecipientType.TO, new InternetAddress(to));
46     message.setSubject(subject);
47     message.setText(builder.toString());
48     Transport tr = mailSession.getTransport();
49     try
50     {
51         tr.connect(null, password);
52         tr.sendMessage(message, message.getAllRecipients());
53     }
54     finally
55     {
56         tr.close();
57     }
58 }
59 }
```

在本章中，你已经看到了如何用 Java 编写网络客户端和服务器，以及如何从 Web 服务器上获取数据。下一章将讨论数据库连接，你将会学习如何通过使用 JDBC API 来实现用 Java 操作关系型数据库。

第 5 章 数据库编程

- ▲ JDBC 的设计
- ▲ 结构化查询语言
- ▲ JDBC 配置
- ▲ 使用 JDBC 语句
- ▲ 执行查询操作
- ▲ 可滚动和可更新的结果集
- ▲ 行集
- ▲ 元数据
- ▲ 事务
- ▲ Web 和企业应用中的连接管理

1996 年，Sun 公司发布了第 1 版的 Java 数据库连接（JDBC）API，使编程人员可以通过这个 API 接口连接到数据库，并使用结构化查询语言（即 SQL）完成对数据库的查找与更新。（SQL 通常发音为“sequel”，它是数据库访问的业界标准。）JDBC 自此成为 Java 类库中最常使用的 API 之一。

JDBC 的版本已更新过数次。在本书出版之际，最新版的 JDBC 4.3 也被囊括到了 Java 9 中。

在本章中，我们将阐述 JDBC 幕后的关键思想，并将介绍（或者是复习）一下 SQL（Structured Query Language，结构化查询语言），它是关系数据库的业界标准。我们还将提供足够的细节，使你可以将 JDBC 融入常见的编程场景中。

 **注释：**根据 Oracle 的声明，JDBC 是一个注册了商标的术语，而并非 Java Database Connectivity 的首字母缩写。对它的命名体现了对 ODBC 的致敬，后者是微软开创的标准数据库 API，并因此而并入了 SQL 标准中。

5.1 JDBC 的设计

从一开始，Java 技术开发人员就意识到了 Java 在数据库应用方面的巨大潜力。从 1995 年开始，他们就致力于扩展 Java 标准类库，使之可以运用 SQL 访问数据库。他们最初希望通过扩展 Java，就可以让人们“纯”用 Java 语言与任何数据库进行通信。但是，他们很快发现这是一项无法完成的任务：因为业界存在许多不同的数据库，且它们所使用的协议也各不相同。尽管很多数据库供应商都表示支持 Java 提供一套数据库访问的标准网络协议，但是每一家企业都希望 Java 能采用自己的网络协议。

所有的数据库供应商和工具开发商都认为，如果 Java 能够为 SQL 访问提供一套“纯”Java API，同时提供一个驱动管理器，以允许第三方驱动程序可以连接到特定的数据库，那它就会显得非常有用。这样，数据库供应商就可以提供自己的驱动程序，将其插入到驱动管理器中。这将成为一种向驱动管理器注册第三方驱动程序的简单机制。

这种接口组织方式遵循了微软公司非常成功的 ODBC 模式，ODBC 为 C 语言访问数据库提供了一套编程接口。JDBC 和 ODBC 都基于同一个思想：根据 API 编写的程序都可以与驱动管理器进行通信，而驱动管理器则通过驱动程序与实际的数据库进行通信。

所有这些都意味着 JDBC API 是大部分程序员不得不使用的接口。

5.1.1 JDBC 驱动程序类型

JDBC 规范将驱动程序归结为以下几类：

- 第 1 类驱动程序将 JDBC 翻译成 ODBC，然后使用 ODBC 驱动程序与数据库进行通信。较早版本的 Java 包含了一个这样的驱动程序：JDBC/ODBC 桥，不过在使用这个桥接器之前需要对 ODBC 进行相应的部署和正确的设置。在 JDBC 面世之初，桥接器可以方便地用于测试，却不太适用于产品的开发。现在，有很多更好的驱动程序可用，所以 JDK 已经不再提供 JDBC/ODBC 桥了。
- 第 2 类驱动程序是由部分 Java 程序和部分本地代码组成的，用于与数据库的客户端 API 进行通信。在使用这种驱动程序之前，客户端不仅需要安装 Java 类库，还需要安装一些与平台相关的代码。
- 第 3 类驱动程序是纯 Java 客户端类库，它使用一种与具体数据库无关的协议将数据库请求发送给服务器构件，然后该构件再将数据库请求翻译成数据库相关的协议。这简化了部署，因为平台相关的代码只位于服务器端。
- 第 4 类驱动程序是纯 Java 类库，它将 JDBC 请求直接翻译成数据库相关的协议。

 **注释：**JDBC 规范可以在 <https://jcp.org/aboutJava/communityprocess/mrel/jsr221/index3.html> 处获得。

大部分数据库供应商都为他们的产品提供第 3 类或第 4 类驱动程序。与数据库供应商提供的驱动程序相比，许多第三方公司专门开发了很多更符合标准的产品，它们支持更多的平台、运行性能也更佳，某些情况下甚至具有更高的可靠性。

总之，JDBC 最终是为了实现以下目标：

- 通过使用标准的 SQL 语句，甚至是专门的 SQL 扩展，程序员就可以利用 Java 语言开发访问数据库的应用，同时还依旧遵守 Java 语言的相关约定。
- 数据库供应商和数据库工具开发商可以提供底层的驱动程序。因此，他们可以优化各自数据库产品的驱动程序。

 **注释：**也许你会问为什么 Java 没有采用 ODBC 模型，下面就是在 1996 年举行的 JavaOne 研讨会上给出的说法：

- ODBC 很难学会。
- ODBC 中有几个命令需要配置很多复杂的选项，而在 Java 编程语言中所采用的风格是要让方法简单而直观，但数量巨大。

- ODBC 依赖于 void* 指针和其他 C 语言特性，而这些特性 Java 编程语言并不具备。
- 与纯 Java 的解决方案相比，基于 ODBC 的解决方案天生就缺乏安全性，且难于部署。

5.1.2 JDBC 的典型用法

在传统的客户端 / 服务器模型中，通常是在服务器端部署数据库，而在客户端安装富 GUI 程序（参见图 5-1）。在此模型中，JDBC 驱动程序应该部署在客户端。

但是，如今三层模型更加常见。在三层应用模型中，客户端不直接调用数据库，而是调用服务器上的中间件层，由中间件层完成数据库查询操作。这种三层模型有以下优点：它将可视化表示（位于客户端）从业务逻辑（位于中间层）和原始数据（位于数据库）中分离出来。因此，我们可以从不同的客户端，如 Java 桌面应用、浏览器或者移动 App，来访问相同的数据和相同的业务规则。

客户端和中间层之间的通信在典型情况下是通过 HTTP 来实现的。JDBC 管理着中间层和后台数据库之间的通信，图 5-2 展示了这种通信模型的基本架构。

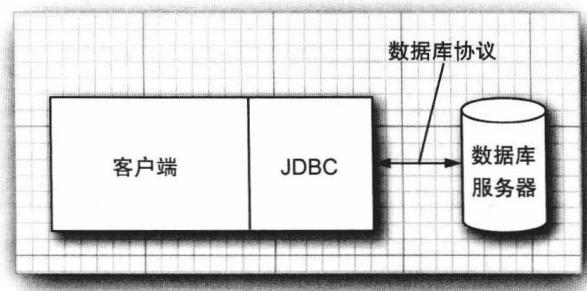


图 5-1 传统的客户端 / 服务器应用

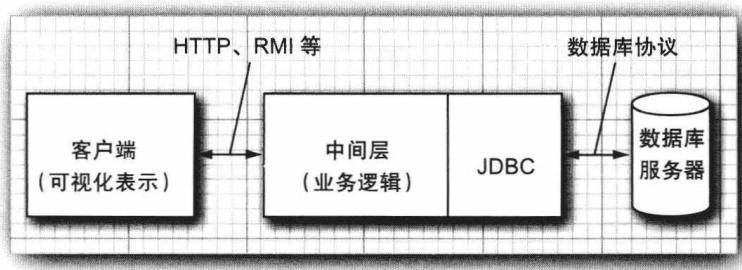


图 5-2 三层结构的应用

5.2 结构化查询语言

SQL 是对所有现代关系型数据库都至关重要的命令行语言，JDBC 则使得我们可以通过 SQL 与数据库进行通信。桌面数据库通常都有一个图形用户界面；通过这种界面，用户可以直接操作数据。但是，基于服务器的数据库只能使用 SQL 进行访问。

我们可以将 JDBC 包看作是一个用于将 SQL 语句传递给数据库的应用编程接口（API）。在本节中，我们将简单介绍一下 SQL。如果之前没有接触过 SQL，你会发现这些介绍是远

远不够的，你可以参阅关于 SQL 的其他著作。我们推荐 Alan Beaulieu 所著的 *Learning SQL* (2009 年由 O'Reilly 出版社出版)，或者还可以参考在线图书 *Learn SQL The Hard Way*，该书可在 <http://sql.learncodethehardway.org> 处获得。

可以将数据库想象成一组由行和列构成的具名表，其中每一列都有列名 (column name)，而每一行则包含了一个相关的信息集。

作为本书的数据库实例，我们将使用一组数据库表来描述一组经典的计算机著作（请参见表 5-1～表 5-4）。

表 5-1 Authors 表

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BR00	Brooks	Frederick P.
...

表 5-2 Books 表

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
...

表 5-3 BooksAuthors 表

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
...

表 5-4 Publishers 表

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...

图 5-3 显示的是一个 Books 表的视图，而图 5-4 显示了对 Books 表和 Publishers 表执行连接操作后的结果。Books 表和 Publishers 表都包含了一个表示出版社的 ID 字段。当我们利用出版社编号对这两个表进行连接操作时，我们就得到了由连接后的表格的值所组成的查询结果。结果中的每一行都包含了图书的信息、出版社名称及其 Web 页的 URL 地址。注意，有的出版社名称和 URL 地址会重复出现在数行中，因为这些行都对应于同一个出版社。

	Title	ISBN	Publisher_ID	Price
>	JNIX System Administration Handbook	0-13-020601-6	013	68.00
	The C Programming Language	0-13-110362-8	013	42.00
	A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
	Introduction to Automata Theory, Languages, and Computation	0-201-44124-1	0201	105.00
	Design Patterns	0-201-63361-2	0201	54.99
	The C++ Programming Language	0-201-70073-5	0201	64.99
	The Mythical Man-Month	0-201-83595-9	0201	29.95
	Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
	The Art of Computer Programming vol. 1	0-201-89683-4	0201	59.99
	The Art of Computer Programming vol. 2	0-201-89584-2	0201	59.99
	The Art of Computer Programming vol. 3	0-201-89685-0	0201	59.99
	A Guide to the SQL Standard	0-201-96426-0	0201	47.95
	Introduction to Algorithms	0-262-03293-7	0262	80.00
	Applied Cryptography	0-471-11709-9	0471	60.00
	JavaScript: The Definitive Guide	0-596-00048-0	0596	44.95
	The Cathedral and the Bazaar	0-596-00108-8	0596	16.95
	The Soul of a New Machine	0-679-60261-5	0679	18.95
	The Codebreakers	0-684-83130-9	07434	70.00
	Cuckoo's Egg	0-7434-1146-3	07434	13.95
	The UNIX Hater's Handbook	1-56884-203-1	0471	16.95

图 5-3 包含图书信息的示例表格

Field	Title	Publisher_ID	Price	Name	URL
Alias					
Table	Books	Books	Books	Publishers	Publishers
Sort					
Visible	<input checked="" type="checkbox"/>				
Function					
Criterion					
Or					
And					

图 5-4 对两个表进行连接操作

对表格进行连接操作的好处是能够避免在数据库表中出现不必要的重复数据。例如，有一种比较简陋的数据库设计是在 Books 表中设置出版社名称和 URL 地址字段。但是这样一来，数据库本身，而非查询结果，将出现许多重复数据。如果出版社的 Web 地址发生了改变，就需要更新所有的重复数据。显然，这在一定程度上很容易导致错误。在关系模型中，我们将数据分布到多个表中，使得所有信息都不会出现不必要的重复。例如，每个出版社的 URL 地址只在出版社表中出现一次。如果需要将此信息与其他信息组合，我们只需对表进行连接操作。

在上述两幅图中，可以看到一个用于查看和链接表的图形工具。许多数据库提供商都具

有相应的工具，通过连接列名和在表单中填入信息，让用户能够以某种简单的方式来表示其各种查询。这种工具通常称为实例查询（Query by Example, QBE）工具。而使用 SQL 的查询则是利用 SQL 语法以文本方式编写的。例如，

```
SELECT Books, Books.Publisher_Id, Books.Price, Publishers.Name, Publishers.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

在本节的余下部分中，我们将介绍如何编写这样的查询语句。如果你已经熟悉 SQL 了，就可以跳过这部分内容。

按照惯例，SQL 关键字全部使用大写字母。当然，也可以不这样做。

SELECT 语句相当灵活。仅使用下面这个查询语句，就可以查出 Books 表中的所有记录：

```
SELECT * FROM Books
```

在每一个 SQL 的 SELECT 语句中，FROM 子句都是必不可少的。FROM 子句用于告知数据库应该在哪个表上查询数据。

我们还可以选择所需要的列：

```
SELECT ISBN, Price, Title
FROM Books
```

并且还可以在查询语句中使用 WHERE 子句来限定所要选择的行：

```
SELECT ISBN, Price, Title
FROM Books
WHERE Price <= 29.95
```

请小心使用“相等”这个比较操作。与 Java 编程语言不同，SQL 使用 = 和 <> 而非 == 和 != 来进行相等性比较。

注释：有些数据库供应商的产品支持在进行不等于比较时使用 !=。这不符合标准 SQL 的语法，所以我们建议不要使用这种方法。

WHERE 子句也可以使用 LIKE 操作符来实现模式匹配。不过，这里的通配符并不是通常使用的 * 和 ?，而是用 % 表示 0 或多个字符，用下划线表示单个字符。例如，

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE '%n_x%'
```

这条语句排除了所有书名中包含 UNIX 或者 Linux 的图书。

请注意，字符串都是用单引号括起来的，而非双引号。字符串中的单引号则需要用一对单引号代替。例如，

```
SELECT Title
FROM Books
WHERE Title LIKE '%''%'
```

上述语句会返回所有包含单引号的书名。

你也可以从多个表中选取数据：

```
SELECT * FROM Books, Publishers
```