

版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



仅供非商业用途或交流学习使用

Java领域最有影响力和价值的著作之一，与《Java编程思想》齐名，10余年
全球畅销不衰，广受好评

根据Java SE 8全面更新，系统全面讲解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

Java
核心技术
系列

P Pearson

Java 核心技术 卷II

高级特性（原书第10版）

Core Java Volume II—Advanced Features

(10th 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



华章IT

内容简介

Java领域最有影响力和价值的著作之一，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），与《Java编程思想》齐名，10余年全球畅销不衰，广受好评。第10版根据Java SE 8全面更新，同时修正了第9版中的不足，系统全面讲解了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。

本书共12章。第1章概述Java 8的流库；第2章的主题是输入输出处理；第3章介绍XML，怎样解析XML文件，怎样生成XML以及怎样使用XSL转换；第4章讲解网络API；第5章介绍数据库编程，重点讲解JDBC；第6章讲解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章讨论国际化；第8章介绍3种处理代码的技术；第9章讲解安全模型；第10章涵盖没有纳入卷I的所有Swing知识；第11章介绍Java 2D API；第12章讲解本地方法。





Java 核心技术 卷 II

高级特性 (原书第10版)

Core Java Volume II—Advanced Features

(10th 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



华章 IT

图书在版编目 (CIP) 数据

Java 核心技术 卷 II 高级特性 (原书第 10 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, 2017.6 (2018.4 重印)
(Java 核心技术系列)

书名原文: Core Java Volume II—Advanced Features (10th Edition)

ISBN 978-7-111-57331-9

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 (2017) 第 142782 号

本书版权登记号: 图字: 01-2017-1400

Authorized translation from the English language edition, entitled *Core Java Volume II—Advanced Features (10th Edition)*, 9780134177298, by Cay S. Horstmann, published by Pearson Education, Inc., Copyright © 2017 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 Pearson Education Asia Ltd., and China Machine Press Copyright © 2017.

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 10 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: 北京文昌阁彩色印刷有限责任公司

版 次: 2018 年 4 月第 1 版第 3 次印刷

开 本: 186mm×240mm 1/16

印 张: 51

书 号: ISBN 978-7-111-57331-9

定 价: 139.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

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



译者序

《Java核心技术 卷Ⅱ 高级特性(原书第10版)》中文版又要呈现在广大读者的面前了!这是我翻译的本书的第4个版本,细心一看,才发现距离最早的第7版已经过去了将近12年,岁月神偷悄然改变着我的音容相貌,也让Java语言不断地完善演化,发生了脱胎换骨般的变化。

随着Java语言的更新,本书的内容也进行了大幅度的调整,新增了Java SE 8中的流库,以及日期和时间API的内容,调整掉了JavaBean和RMI等内容,使得本书的内容既反映了Java语言的新变化,又显得更加紧凑,达到了与时俱进的目的。本书实际上并不适合Java初学者,它更适合有一定Java编程基础的程序员,因为具备一定的基础知识才能更好地理解本书的内容,这也是卷Ⅱ被称为“高级特性”的原因。通过阅读本书,你会了解到高级特性的细节,它们涉及复杂系统的各个方面,是开发更好、更快、更安全和更易维护的系统所必不可少的语言特性。

在这一版的翻译工作中,我对原文没有变化的部分也进行了仔细修订,尽量修改了其中的错误和翻译不通顺的语句。令人汗颜的是,虽然已经修订过3版了,在这一版中还是发现了不少错误,在此我向所有之前版本的读者道歉,也恳请读者对这一版中的谬误提出批评。

最后,祝大家通过阅读本书不但能够提升Java编程能力,更能够加深对Java编程语言的理解和认识。让我们共同学习,永远在路上!

陈昊鹏
2015年1月于北京

陈昊鹏,男,1982年生,现居北京,IT行业从业者,业余喜欢研究各种编程语言,尤其对Java情有独钟,并乐于将自己对Java的理解与感悟分享给身边的朋友。陈昊鹏在IT行业工作多年,曾就职于IBM、中软国际等公司,担任过项目经理、架构师等职位,并参与过多个大型项目的开发与管理。陈昊鹏热爱编程,喜欢通过编写代码来解决问题,同时也喜欢通过阅读书籍来提升自己的技术水平。陈昊鹏在业余时间喜欢阅读各种类型的书籍,尤其对计算机科学领域的书籍有着浓厚的兴趣。陈昊鹏还喜欢摄影,经常利用业余时间拍摄各种美景,并将拍摄的照片发布到自己的博客上,与大家分享。

陈昊鹏的博客地址: <http://www.mhp.org.cn>。陈昊鹏的微博地址: <http://weibo.com/mhp082>。陈昊鹏的微信公众号: mhp082。

陈昊鹏的个人简介: 陈昊鹏,男,1982年生,现居北京,IT行业从业者,业余喜欢研究各种编程语言,尤其对Java情有独钟,并乐于将自己对Java的理解与感悟分享给身边的朋友。陈昊鹏在IT行业工作多年,曾就职于IBM、中软国际等公司,担任过项目经理、架构师等职位,并参与过多个大型项目的开发与管理。陈昊鹏热爱编程,喜欢通过编写代码来解决问题,同时也喜欢通过阅读书籍来提升自己的技术水平。陈昊鹏在业余时间喜欢阅读各种类型的书籍,尤其对计算机科学领域的书籍有着浓厚的兴趣。陈昊鹏还喜欢摄影,经常利用业余时间拍摄各种美景,并将拍摄的照片发布到自己的博客上,与大家分享。



前　　言

致读者

本书是按照 Java SE 8 完全更新后的《Java核心技术 卷Ⅱ：高级特性（原书第10版）》。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，与本书卷Ⅰ和之前的版本一样，我们仍将本书定位为用 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，我们更希望对本书问题的报告只听到一次。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（该网页的目的是鼓励读者阅读以前的报告）的末尾处添加 bug 报告，以此来发布 bug 和问题并给出建议，以便我们改进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题，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 8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从而使保存和加载对象变得容易而方便，及其背后的原理。然后，我们讨论了正则表达式和操作文件与路径。

第3章介绍XML，介绍怎样解析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。我们将介绍怎样创建连接到服务器上，怎样实现你自己的服务器，以及怎样创建HTTP连接。

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，探讨了一下 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 章讨论了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加深远。因此，你可以对 Java 应用程序进行国际化，使得它们不仅可以跨平台，而且还可以跨越国界。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 章讨论了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这些注解，以及怎样运用这些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，我们希望我们的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以及安全管理器。然后介绍允许使用消息、代码签名、授权以及认证和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了总结。

第 10 章涵盖了没有纳入卷 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。随后我们介绍了编辑面板的基本用法、“多文档”界面的 Java 实现、在多线程程序中用到的进度指示器，以及诸如闪屏和支持系统托盘这样的“桌面集成特性”。我们仍着重介绍在实际编程中可能遇到的最为有用的构件，因为对 Swing 类库进行百科全书般的介绍可能会占据好几卷书的篇幅，并且只有专门的分类学家才感兴趣。

第 11 章介绍 Java 2D API，你可以用它来创建实际的图形和特殊的效果。该章还介绍了抽象窗口操作工具包（AWT）的一些高级特性，这部分内容看起来过于专业，不适合在卷 I 中介绍。虽然如此，这些技术还是应该成为每一个编程人员工具包的一部分。这些特性包括打印和用于剪切粘贴及拖放的 API。

第 12 章介绍本地方法，这个功能可以让你调用为微软 Windows API 这样的特殊机制而编写的各种方法。很显然，这种特性具有争议性：使用本地方法，那么 Java 平台的跨平台特性将会随之消失。虽然如此，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 SE 8 的新 API

也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。各种图标的意义如下：

 **注意：**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++ 注意：**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一个简短的描述。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。接口的名字都是斜体的，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这些特性。

API Application Programming Interface 1.2

本书示例代码以程序清单的形式列举了出来，例如：

程序清单 1-1 ScriptTest.java

可以从网站 <http://horstmann.com/corejava>^① 下载示例代码。

致谢

写书总是需要付出极大的努力，而重写也并不像看上去那么容易，特别是在 Java 技术方面，要跟上其飞快的发展速度，更是如此。一本书的面世需要众多有奉献精神的人共同努力，我非常荣幸地在此向整个《Java 核心技术》团队致谢。

Prentice Hall 出版社的许多人都提供了颇有价值的帮助，但是他们甘愿居于幕后。我希望他们都能够知道我是多么感谢他们付出的努力。与以往一样，我要热切地感谢我的编辑，Prentice Hall 出版社的 Greg Doench，他对本书从编写到出版进行全程掌舵，并使我可以十分幸福地根本意识不到幕后那些人的存在。我还非常感谢 Julie Nahil 在撰写上的支持，以及感谢 Dmitry Kirsanov 和 Alina Kirsanova 对手稿的编辑和排版。

我非常感谢找到了很多令人尴尬的错误并提出了许多颇具创见性的建议的早先版本的读

^① 也可登录华章网站 (<http://www.hzbook.com>) 下载相关代码。——编辑注

者。我特别要感谢十分出色的评审团队，他们用令人惊异的眼睛仔细浏览了所有原稿，并将我从许多令人尴尬的错误中拯救了出来。

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）、Dr. Nicholas J. De Lillo（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）、Jin 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 Phlion、Blake Ragsdell、Ylber Ramadani（Ryerson 大学）、Stuart Reges（亚利桑那大学）、Simon Ritter、Rich Rosen（Interactive Data Corporation）、Peter Sanders（ESSI 大学，Nice, France）、Dr. Paul Sanghera（圣何塞州立大学和布鲁克学院）、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

2016 年 9 月于加州旧金山



Java 8 的新特性，特别是对流 API 的改进，让 Java 成为了处理大数据的利器。本书将通过深入浅出的讲解，帮助读者掌握 Java 8 新增的流 API，从而更好地利用 Java 8 处理大数据。

目 录

译者序	如何读入文本输入	51
前言	以文本格式存储对象	52
在新的时代里，数据是最重要的资产	字符编码方式	55
第1章 Java SE 8 的流库	读写二进制数据	57
1.1 从迭代到流的操作	2.3.1 DataInput 和 DataOutput	57
1.2 流的创建	2.3.2 随机访问文件	59
1.3 filter、map 和 flatMap 方法	2.3.3 ZIP 文档	63
1.4 抽取子流和连接流	2.4 对象输入 / 输出流与序列化	66
1.5 其他的流转换	2.4.1 保存和加载序列化对象	66
1.6 简单约简	2.4.2 理解对象序列化的文件	67
1.7 Optional 类型	1.7.1 如何使用 Optional 值	70
	1.7.2 不适合使用 Optional 值的方式	75
	1.7.3 创建 Optional 值	77
	1.7.4 用 flatMap 来构建 Optional 值的函数	78
1.8 收集结果	2.4.4 版本管理	78
1.9 收集到映射表中	2.4.6 为克隆使用序列化	80
1.10 群组和分区	2.5 操作文件	83
1.11 下游收集器	2.5.1 Path	83
1.12 约简操作	2.5.2 读写文件	85
1.13 基本类型流	2.5.3 创建文件和目录	87
1.14 并行流	2.5.4 复制、移动和删除文件	88
第2章 输入与输出	2.5.5 获取文件信息	89
2.1 输入 / 输出流	2.5.6 访问目录中的项	91
2.1.1 读写字节	2.5.7 使用目录流	92
2.1.2 完整的流家族	2.5.8 ZIP 文件系统	95
2.1.3 组合输入 / 输出流过滤器	2.6 内存映射文件	96
2.2 文本输入与输出	2.6.1 内存映射文件的性能	96
2.2.1 如何写出文本输出	2.6.2 缓冲区数据结构	103
	2.6.3 文件加锁机制	105
	2.7 正则表达式	106

第3章 XML	117	4.4.3 提交表单数据	220
3.1 XML概述	117	4.4.5 发送E-mail	228
3.1.1 XML文档的结构	119	第5章 数据库编程	232
3.2 解析XML文档	122	5.1 JDBC的设计	232
3.3 验证XML文档	132	5.1.1 JDBC驱动程序类型	233
3.3.1 文档类型定义	133	5.1.2 JDBC的典型用法	234
3.3.2 XML Schema	139	5.2 结构化查询语言	234
3.3.3 实用示例	142	5.3 JDBC配置	239
3.4 使用XPath来定位信息	154	5.3.1 数据库URL	240
3.5 使用命名空间	159	5.3.2 驱动程序JAR文件	240
3.6 流机制解析器	162	5.3.3 启动数据库	240
3.6.1 使用SAX解析器	162	5.3.4 注册驱动器类	241
3.6.2 使用StAX解析器	166	5.3.5 连接到数据库	242
3.7 生成XML文档	170	5.4 使用JDBC语句	244
3.7.1 不带命名空间的文档	170	5.4.1 执行SQL语句	244
3.7.2 带命名空间的文档	170	5.4.2 管理连接、语句和结果集	247
3.7.3 写出文档	171	5.4.3 分析SQL异常	248
3.7.4 示例：生成SVG文件	172	5.4.4 组装数据库	250
3.7.5 使用StAX写出XML	174	5.5 执行查询操作	254
3.8 XSL转换	181	5.5.1 预备语句	254
第4章 网络	191	5.5.2 读写LOB	259
4.1 连接到服务器	191	5.5.3 SQL转义	261
4.1.1 使用telnet	191	5.5.4 多结果集	262
4.1.2 用Java连接到服务器	193	5.5.5 获取自动生成的键	263
4.1.3 套接字超时	195	5.6 可滚动和可更新的结果集	263
4.1.4 因特网地址	196	5.6.1 可滚动的结果集	264
4.2 实现服务器	198	5.6.2 可更新的结果集	266
4.2.1 服务器套接字	198	5.7 行集	269
4.2.2 为多个客户端服务	201	5.7.1 构建行集	270
4.2.3 半关闭	204	5.7.2 被缓存的行集	270
4.3 可中断套接字	205	5.8 元数据	273
4.4 获取Web数据	211	5.9 事务	282
4.4.1 URL和URI	211	5.9.1 用JDBC对事务编程	282
4.4.2 使用URLConnection获取		5.9.2 保存点	283
信息	213	5.9.3 批量更新	283
		5.10 高级SQL类型	285

第 5 章 Web 与企业应用中的连接	286	8.1.4 调用脚本的函数和方法	356
管理	286	8.1.5 编译脚本	357
第 6 章 日期和时间 API	288	8.1.6 一个示例：用脚本处理	
6.1 时间线	288	GUI 事件	358
6.2 本地时间	291	8.2 编译器 API	363
6.3 日期调整器	294	8.2.1 编译便捷之法	363
6.4 本地时间	295	8.2.2 使用编译工具	363
6.5 时区时间	296	8.2.3 一个示例：动态 Java 代码	
6.6 格式化和解析	299	生成	368
6.7 与遗留代码的互操作	302	8.3 使用注解	373
第 7 章 国际化	304	8.3.1 注解简介	373
7.1 Locale 对象	304	8.3.2 一个示例：注解事件处理器	374
7.2 数字格式	309	8.4 注解语法	379
7.3 货币	314	8.4.1 注解接口	379
7.4 日期和时间	315	8.4.2 注解	380
7.5 排序和范化	321	8.4.3 注解各类声明	382
7.6 消息格式化	327	8.4.4 注解类型用法	383
7.6.1 格式化数字和日期	327	8.4.5 注解 this	384
7.6.2 选择格式	329	8.5 标准注解	385
7.7 文本文件和字符集	331	8.5.1 用于编译的注解	386
7.7.1 文本文件	331	8.5.2 用于管理资源的注解	386
7.7.2 行结束符	331	8.5.3 元注解	387
7.7.3 控制台	331	8.6 源码级注解处理	389
7.7.4 日志文件	332	8.6.1 注解处理	389
7.7.5 UTF-8 字节顺序标志	332	8.6.2 语言模型 API	390
7.7.6 源文件的字符编码	333	8.6.3 使用注解来生成源码	390
7.8 资源包	333	8.7 字节码工程	393
7.8.1 定位资源包	334	8.7.1 修改类文件	393
7.8.2 属性文件	335	8.7.2 在加载时修改字节码	398
7.8.3 包类	335	第 9 章 安全	401
7.9 一个完整的例子	337	9.1 类加载器	401
第 8 章 脚本、编译与注解处理	352	9.1.1 类加载过程	402
8.1 Java 平台的脚本	352	9.1.2 类加载器的层次结构	403
8.1.1 获取脚本引擎	352	9.1.3 将类加载器作为命名空间	404
8.1.2 脚本赋值与绑定	353	9.1.4 编写你自己的类加载器	405
8.1.3 重定向输入和输出	355	9.1.5 字节码校验	410

9.2 安全管理器与访问权限	414	10.3.3 节点枚举	530
9.2.1 权限检查	414	10.3.4 绘制节点	532
9.2.2 Java 平台安全性	415	10.3.5 监听树事件	534
9.2.3 安全策略文件	418	10.3.6 定制树模型	541
9.2.4 定制权限	424	10.4 文本构件	548
9.2.5 实现权限类	426	10.4.1 文本构件中的修改跟踪	549
9.3 用户认证	431	10.4.2 格式化的输入框	552
9.3.1 JAAS 框架	431	10.4.3 JSpinner 构件	567
9.3.2 JAAS 登录模块	437	10.4.4 用 JEditorPane 显示	
9.4 数字签名	445	HTML	574
9.4.1 消息摘要	445	10.5 进度指示器	579
9.4.2 消息签名	448	10.5.1 进度条	580
9.4.3 校验签名	449	10.5.2 进度监视器	582
9.4.4 认证问题	452	10.5.3 监视输入流的进度	585
9.4.5 证书签名	454	10.6 构件组织器和装饰器	590
9.4.6 证书请求	454	10.6.1 分割面板	590
9.4.7 代码签名	455	10.6.2 选项卡面板	592
9.5 加密	460	10.6.3 桌面面板和内部框架	597
9.5.1 对称密码	461	10.6.4 层	613
9.5.2 密钥生成	462	第 11 章 高级 AWT	618
9.5.3 密码流	466	11.1 绘图操作流程	618
9.5.4 公共密钥密码	467	11.2 形状	620
第 10 章 高级 Swing	472	11.2.1 形状类层次结构	621
10.1 列表	472	11.2.2 使用形状类	623
10.1.1 JList 构件	472	11.3 区域	634
10.1.2 列表模式	477	11.4 笔划	635
10.1.3 插入和移除值	481	11.5 着色	642
10.1.4 值的绘制	482	11.6 坐标变换	644
10.2 表格	486	11.7 剪切	648
10.2.1 简单表格	486	11.8 透明与组合	650
10.2.2 表格模型	489	11.9 绘图提示	657
10.2.3 对行和列的操作	493	11.10 图像的读取器和写入器	663
10.2.4 单元格的绘制和编辑	506	11.10.1 获得适合图像文件类型的	
10.3 树	517	读取器和写入器	663
10.3.1 简单的树	518	11.10.2 读取和写入带有多个图像	
10.3.2 编辑树和树的路径	524	的文件	664

11.11	图像处理	671	11.15.1	闪屏	739
11.11.1	构建光栅图像	672	11.15.2	启动桌面应用程序	743
11.11.2	图像过滤	678	11.15.3	系统托盘	748
11.12	打印	685	第12章	本地方法	752
11.12.1	图形打印	685	12.1	从 Java 程序中调用 C 函数	752
11.12.2	打印多页文件	693	12.2	数值参数与返回值	757
11.12.3	打印预览	694	12.3	字符串参数	759
11.12.4	打印服务程序	702	12.4	访问域	764
11.12.5	流打印服务程序	706	12.4.1	访问实例域	765
11.12.6	打印属性	707	12.4.2	访问静态域	768
11.13	剪贴板	712	12.5	编码签名	769
11.13.1	用于数据传递的类和接口	713	12.6	调用 Java 方法	770
11.13.2	传递文本	714	12.6.1	实例方法	771
11.13.3	Transferable 接口和数据风格	717	12.6.2	静态方法	774
11.13.4	构建一个可传递的图像	718	12.6.3	构造器	775
11.13.5	通过系统剪贴板传递 Java 对象	722	12.6.4	另一种方法调用	775
11.13.6	使用本地剪贴板来传递对象引用	725	12.7	访问数组元素	777
11.14	拖放操作	725	12.8	错误处理	780
11.14.1	Swing 对数据传递的支持	726	12.9	使用调用 API	785
11.14.2	拖曳源	730	12.10	完整的示例：访问 Windows 注册表	789
11.14.3	放置目标	732	12.10.1	Windows 注册表概述	789
11.15	平台集成	739	12.10.2	访问注册表的 Java 平台接口	791

第1章 Java SE 8 的流库

- ▲ 从迭代到流的操作
- ▲ 流的创建
- ▲ `filter`、`map` 和 `flatMap` 方法
- ▲ 抽取子流和连接流
- ▲ 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

流提供了一种让我们可以在比集合更高的概念级别上指定计算的数据视图。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 SE 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：

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

现在，我们可以迭代它了：

```
long 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();
```



流的版本比循环版本要更易于阅读，因为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。

```
1 long count = words.parallelStream()
2     .filter(w -> w.length() > 12)
3     .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` 方法不会从新的流中移除元素，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。

让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 `stream`。`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 中的示例中，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 import java.io.IOException;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.nio.file.Paths;
7 import java.util.Arrays;
8 import java.util.List;
9
```



```

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

```

在下一节中，你将会看到如何创建流。后续的三个小节将处理流的转换。再后面的五个小节将讨论终止操作。

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)` 可以从数组中位于 `from` (包括) 和 `to` (不包括) 的元素中创建一个流。

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);
```

为了产生无限序列, 例如 0 1 2 3 …, 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。

注意: 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);
```

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
}
```

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 import java.io.IOException;
4 import java.math.BigInteger;
5 import java.nio.charset.StandardCharsets;
6 import java.nio.file.Files;
```



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

28         }
29         System.out.println();
30     }
31
32     public static void main(String[] args) throws IOException
33     {
34         Path path = Paths.get("../gutenberg/alice30.txt");
35         String contents = new String(Files.readAllBytes(path),
36             StandardCharsets.UTF_8);
37
38         Stream<String> words = Stream.of(contents.split("\\PL+"));
39         show("words", words);
40         Stream<String> song = Stream.of("gently", "down", "the", "stream");
41         show("song", song);
42         Stream<String> silence = Stream.empty();
43         show("silence", silence);
44
45         Stream<String> echos = Stream.generate(() -> "Echo");
46         show("echos", echos);
47
48         Stream<Double> randoms = Stream.generate(Math::random);
49         show("randoms", randoms);
50
51         Stream<BigInteger> integers = Stream.iterate(BigInteger.ONE,
52             n -> n.add(BigInteger.ONE));
53         show("integers", integers);
54
55         Stream<String> wordsAnotherWay = Pattern.compile("\\PL+").splitAsStream(
56             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 }
```

[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)**

产生一个无限流，它的元素包含种子、在种子上调用 f 产生的值、在前一个元素上调用 f 产生的值，等等。

[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.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了只包含长单词的另一个流：

```
List<String> wordList = ...;
Stream<String> longWords = wordList.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：

```
public static Stream<String> letters(String s)
{
    List<String> result = new ArrayList<>();
    for (int i = 0; i < s.length(); i++)
        result.add(s.substring(i, i + 1));
    return result.stream();
}
```

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

注意：通过使用 1.13 节中的 `IntStream.range` 方法，我们实现这个方法可以优雅得多。

假设我们在一个字符串流上映射 `letters` 方法：

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

那么就会得到一个包含流的流，就像 `[...["y", "o", "u", "r"], ["b", "o", "a", "t"], ...]`。为了将其摊平为字母流 `[..."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 -> letters(w))
    // Calls letters on each word and flattens the results
```

注意：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`。然后，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断言条件的元素。
- `<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 抽取子流和连接流

调用 `stream.limit(n)` 会返回一个新的流，它在 `n` 个元素之后结束 (如果原来的流更短，那么就会在流结束时结束)。这个方法对于裁剪无限流的尺寸会显得特别有用。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。这个方法在将文本分隔为单词时会显得很方便，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);
```

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

```
Stream<String> combined = Stream.concat(
    letters("Hello"), letters("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得到处理的机会。

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。
- `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。这个流显然能够记住它已经看到过的元素。

```
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。

对于调试，你可以让 `peek` 调用一个你设置了断点的方法。

API	java.util.stream.Stream 8
●	<code>Stream<T> distinct()</code>
	产生一个流，包含当前流中所有不同的元素。
●	<code>Stream<T> sorted()</code>
●	<code>Stream<T> sorted(Comparator<? super T> comparator)</code>
	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 <code>Comparable</code> 的类的实例。
●	<code>Stream<T> peek(Consumer<? super T> action)</code>
	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 <code>action</code> 。

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`，它们会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` 类型是一种更好的表示缺少返回值的方式。我们将在下一节中详细讨论 `Optional` 类型。下面展示了可以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。它通常会在与 `filter` 组合使用时显得很有用。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会很有效，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，它们分别会在所有元素和没有任何元素匹配断言的情况下返回 `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断言时返回 `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，我们称这种值为存在的。`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，下一节我们将讨论如何正确使用。

1.7.1 如何使用 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。

让我们来看看第一条策略。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(() -> Locale.getDefault().getDisplayName());
// 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
```

你刚刚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。如果该可选值存在，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);
```

当调用 `ifPresent` 时，从该函数不会返回任何值。如果想要处理函数的结果，应该使用 `map`：

```
Optional<Boolean> added = optionalValue.map(results::add);
```

现在 `added` 具有三种值之一：在 `optionalValue` 存在的情况下包装在 `Optional` 中的 `true` 或 `false`，以及在 `optionalValue` 不存在的情况下空 `Optional`。

注意：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，会应用到函数。

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。
- `void ifPresent(Consumer<? super T> consumer)` 如果该 Optional 不为空，那么就将它的值传递给 consumer。
- `<U> Optional<U> map(Function<? super T, ? extends U> mapper)` 产生将该 Optional 的值传递给 mapper 后的结果，只要这个 Optional 不为空且结果不为 null，否则产生一个空 Optional。

1.7.2 不适合使用 Optional 值的方式

如果没有正确地使用 Optional 值，那么相比较以往的得到“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` 对象。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>` 对象是否具有一个值。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();
```

API java.util.Optional 8

- `T get()`

产生这个 Optional 的值，或者在该 Optional 为空时，抛出一个 `NoSuchElementException` 对象。

- `boolean isPresent()`

如果该 `Optional` 不为空，则返回 `true`。

1.7.3 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` 对象，而第二个方法会产生一个空 `Optional`。
- `static <T> Optional<T> empty()`
- 产生一个空 `Optional`。

1.7.4 用 flatMap 来构建 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。但是这种组合没法工作，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，如果有更多的可以产生 `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时，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了：

```
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(MyMath::inverse).flatMap(MyMath::squareRoot);
```

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

注意：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可以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。如果将可选值当作尺寸为 0 和 1 的流来解释，那么 `Optional.flatMap` 方法与其操作方式一样。

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/ObservableTest.java

```
1 package optional;
2
3 import java.io.*;
4 import java.nio.charset.*;
5 import java.nio.file.*;
6 import java.util.*;
7
8 public class OptionalTest
9 {
10     public static void main(String[] args) throws IOException
11     {
12         String contents = new String(Files.readAllBytes(
13             Paths.get("../gutenberg/alice30.txt")), StandardCharsets.UTF_8);
14         List<String> wordList = Arrays.asList(contents.split("\\PL+"));
15
16         Optional<String> optionalValue = wordList.stream()
17             .filter(s -> s.contains("fred"))
18             .findFirst();
19         System.out.println(optionalValue.orElse("No word") + " contains fred");
20
21         Optional<String> optionalString = Optional.empty();
22         String result = optionalString.orElse("N/A");
23         System.out.println("result: " + result);
24         result = optionalString.orElseGet(() -> Locale.getDefault().getDisplayName());
25         System.out.println("result: " + result);
26         try
27         {
28             result = optionalString.orElseThrow(IllegalStateException::new);
29             System.out.println("result: " + result);
30         }
31         catch (Throwable t)
32         {
33             t.printStackTrace();
34         }
35
36         optionalValue = wordList.stream()
```



```

37     .filter(s -> s.contains("red"))
38     .findFirst();
39     optionalValue.ifPresent(s -> System.out.println(s + " contains red"));
40
41     Set<String> results = new HashSet<>();
42     optionalValue.ifPresent(results::add);
43     Optional<Boolean> added = optionalValue.map(results::add);
44     System.out.println(added);
45
46     System.out.println(inverse(4.0).flatMap(OptionalTest::squareRoot));
47     System.out.println(inverse(-1.0).flatMap(OptionalTest::squareRoot));
48     System.out.println(inverse(0.0).flatMap(OptionalTest::squareRoot));
49     Optional<Double> result2 = Optional.of(-4.0)
50         .flatMap(OptionalTest::inverse).flatMap(OptionalTest::squareRoot);
51     System.out.println(result2);
52 }
53
54 public static Optional<Double> inverse(Double x)
55 {
56     return x == 0 ? Optional.empty() : Optional.of(1 / x);
57 }
58
59 public static Optional<Double> squareRoot(Double x)
60 {
61     return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));
62 }
63 }
```

API java.util.Optional 8

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

产生将 `mapper` 应用于当前的 `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，通常会想要查看其元素。此时可以调用 `iterator` 方法，它会产生可以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。

