

Java 完全参考手册

(第 8 版)

[美] Herbert Schildt 著

王德才 吴明飞 唐业军 译



北 京

Herbert Schildt

Java:The Complete Reference,Eighth Edition

EISBN：978-0-07-160630-1

Copyright © 2011 by The McGraw-Hill Companies, Inc.

All Rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation is jointly published by McGraw-Hill Education (Asia) and Tsinghua University Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2012 by McGraw-Hill Education (Asia), a division of the Singapore Branch of The McGraw-Hill Companies, Inc. and Tsinghua University Press.

版权所有。未经出版人事先书面许可，对本出版物的任何部分不得以任何方式或途径复制或传播，包括但不限于复印、录制、录音，或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔(亚洲)教育出版公司和清华大学出版社合作出版。此版本经授权仅限在中华人民共和国境内(不包括香港特别行政区、澳门特别行政区和台湾)销售。

版权©2012 由麦格劳-希尔(亚洲)教育出版公司与清华大学出版社所有。北京市版权局著作权合同登记号 图字：01-2011-5682

本书封面贴有 McGraw-Hill 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

Java 完全参考手册：第 8 版/(美)施密特(Schildt, H.) 著；王德才，吴明飞，唐业军 译.

—北京：清华大学出版社，2012.9

书名原文：Java:The Complete Reference,Eighth Edition

ISBN 978-7-302-30083-0

Ⅰ. ①J… Ⅱ. ①施… ②王… ③吴… ④唐… Ⅲ. ①Java 语言—程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2012)第 211677 号

责任编辑：王 军 李维杰装帧设计：牛艳敏责任校对：蔡 娟责任印制：

出版发行：清华大学出版社

网 址：http://www.tup.com.cn，http://www.wqbook.com

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

装 订 者：

经 销：全国新华书店

开 本：185mm×260mm 印 张：70.5 字 数：1804 千字

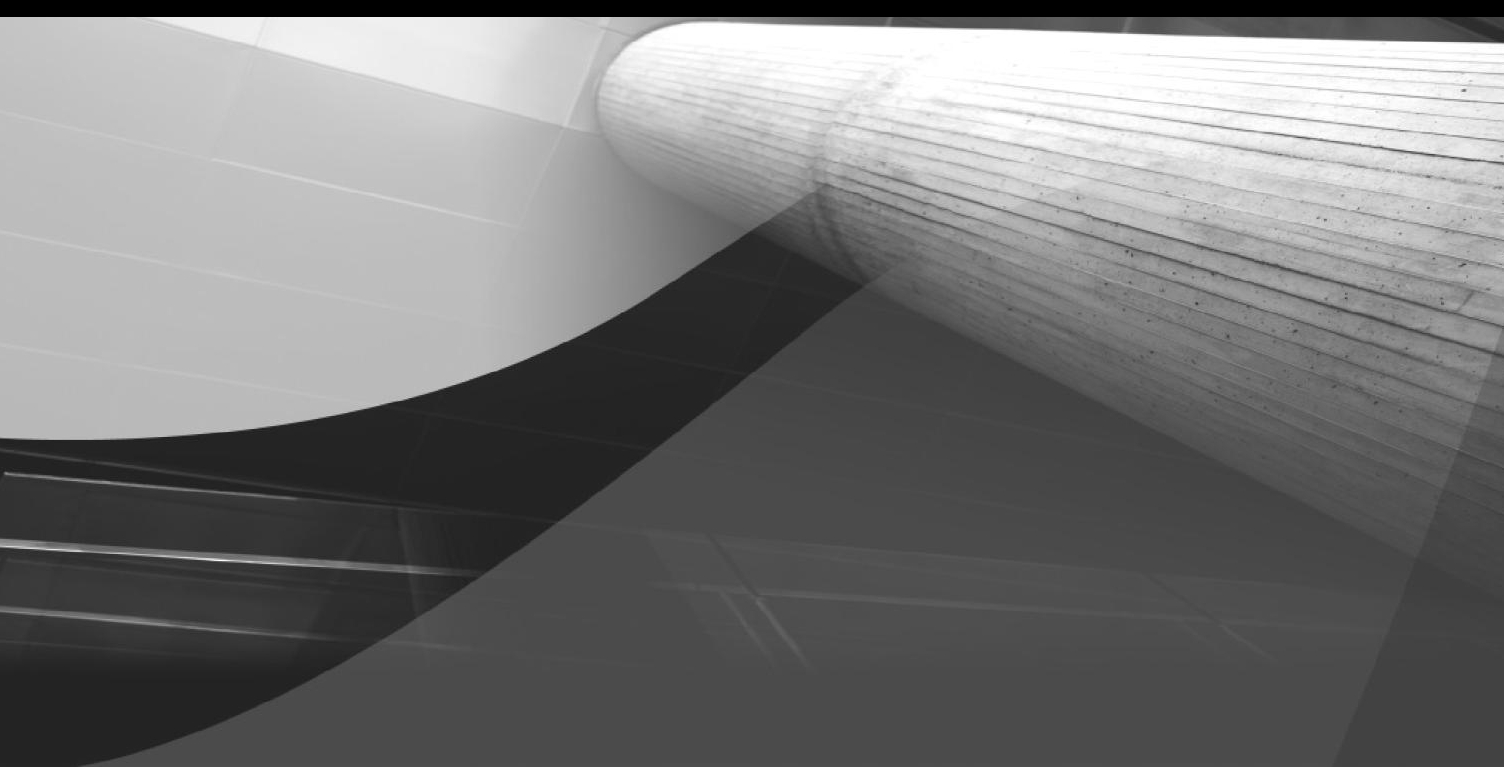
版 次：2012 年 9 月第 1 版 印 次：2012 年 9 月第 1 次印刷

印 数：1～4000

定 价：128.00 元

————————————————————————————————————————

产品编号：



作 者 简 介

Herbert Schildt 是 Java、C++、C 以及 C#语言方面的权威。他撰写的编程书籍在世界

范围内销售了数百万册，并且已经被翻译成所有主要的外国语言。他撰写了大量 Java 方

面的书籍，包括 *Java：A* *Beginner’s* *Guide*、*Herb Schildt’s* *Java Programming Cookbook*、

*Swing：A Beginner’s Guide* 以及 *The Art of Java*。在他撰写的其他书籍中，销售最好的有

*C++：The Complete Reference*、*C#：The Complete Reference* 以及 *C：The Complete Reference*。

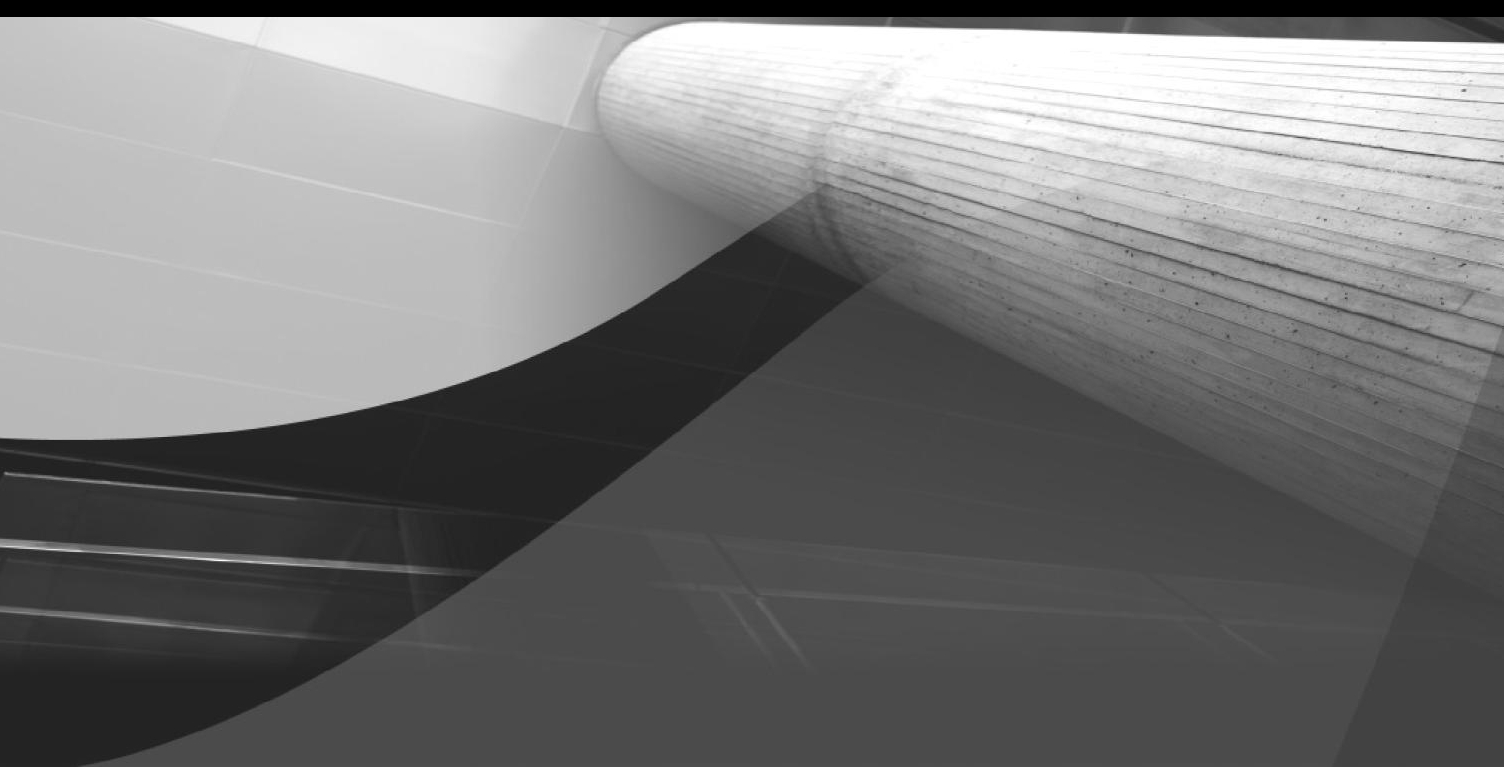
尽管对计算机的所有方面都感兴趣，但是他主要关注计算机语言，包括编译器、解释器以及机器人控制语言。他还对语言的标准化非常感兴趣。Schildt 获得了 Illinois 大学的本

科和研究生学位。他的个人网站是 www.HerbSchildt.com。

**II** Java 完全参考手册(第 8 版)

技术编辑简介

Danny Coward 博士自 1997 年就已经是 Java 平台的贡献者。还在 Sun 公司时，他就是 Java EE 小组的资深成员。他是 Java 社区进程执行委员会(Java Community Process Executive Committee)的成员，并且是 Java 平台所有版本——Java SE、Java ME 以及 Java EE 的权威，他还组建了最初的 JavaFX 团队。



前 言

Java 是当今世界最重要、也是使用最广泛的计算机语言之一。而且，在多年之前它就已经拥有这一荣誉。与其他一些计算机语言随着时间的流逝、影响也逐渐减弱不同，Java 随着时间的推移反而变得更加强大。从首次发布开始，Java 就跃到了 Internet 编程的前沿。后续的每一个版本都进一步巩固了这一地位。如今，Java 依然是开发基于 Web 的应用程序的最佳选择。此外，Java 还是智能手机变革的推手，Android 编程采用的就是 Java 语言。在现实世界中，很多应用都是使用 Java 开发的，Java 真的是非常重要！

Java 成功的一个关键原因是它的敏捷性。自从最初的 1.0 版发布以来，Java 不断地进行完善以适应编程环境和开发人员编程方式的变化。最重要的是，Java 不仅仅是在跟随潮流，更是在帮助创造潮流。Java 能够适应计算机世界快速变化的能力，是它一直并且仍将如此成功的关键因素。

本书自从 1996 年首次出版以来，已经经历了 7 次改版，每次改版都反映了 Java 的不断演化进程。这是本书的第 8 版，并且已经针对 Java SE 7 进行了升级。因此，本版包含了大量

**IV** Java 完全参考手册(第 8 版)

新的内容。例如，包含了 Project Coin 语言改进、扩展的 NIO(NIO.2)以及 Fork/Join 框架。总体而言，对新特性的讨论被集成到已有的章节中。然而，本书的整体结构保持不变。这意味着如果您熟悉本书的以前版本，那么阅读本版本会感觉很自然。

一本适合所有编程人员的书

本书面向所有开发人员，不管是初学者还是有经验的编程人员。初学者将从本书中发现每个主题的详细讨论，以及许多特别有帮助的例子。而对 Java 更高级特性和库的深入讨论，将会吸引有经验的编程人员。无论是对于初学者还是有经验的编程人员，本书都提供了持久的资源和方便实用的参考。

本书内容

本书是对 Java 语言的全面指导，描述了它的语法、关键字以及基本的编程原则，另外还介绍了 Java API 库的重要部分。本书分为 4 部分，每部分关注 Java 开发环境的不同方面。

第Ⅰ部分是对 Java 语言的深入辅导。该部分从基础知识开始讲解，包括数据类型、运算符、控制语句以及类等。然后本部分介绍继承、包、接口、异常处理以及多线程。本部分的最后几章介绍注解、枚举、自动装箱以及泛型等内容。本部分还介绍了 I/O 以及 applet。

第Ⅱ部分介绍 Java 标准 API 库的关键内容。本部分的主题包括字符串、I/O、网络、标准实用工具、集合框架、applet、基于 GUI 的控件、图像以及并发编程(包括新的 Fork/Join

框架)等。

第Ⅲ部分分析三个重要的 Java 技术：Java Bean、servlet 和 Swing。

第Ⅳ部分包含两章，这两章展示了几个实际的 Java 程序。本部分的第 33 章开发了几个执行各种流行金融计算的 applet，如定期还贷、获得预期养老金的最小投资额等。本章还介绍如何将这些 applet 转换成 servlet。本部分的第 34 章开发了一个用于监控文件下载过程的下载管理器，其功能包括启动、停止和恢复传输。这两章都改编自我的著作 *The Art Of Java*，

本书由我和 James Holmes 合著。

不要忘记：Web 上的代码

请记住，本书中全部示例程序的源代码都可以从 www.oraclepressbooks.com 上免费下载。

致 谢

在此我要特别感谢 Patrick Naughton、Joe O’Neil、James Holmes 和 Danny Coward。 Patrick Naughton 是 Java 语言的创立者之一，他还帮助我编写了本书的第 1 版。例如，

本书第 19、20 和 25 章的大部分材料最初都是由 Patrick 提供的，他还为本书做出了其他许多贡献。他的洞察力、专业知识和活力都对本书的成功付梓贡献极大。

在准备本书的第 2 版和第 3 版的过程中，Joe O’Neil 为目前本书第 28、29、31 和 32 章

的内容提供了原始素材。Joe 对我的好几本书都有帮助，并且他提供的帮助一直都是最高质量的。

James Holmes 提供了第 34 章的内容。James 是一位卓越的程序员和作者，他和我合著了

*The Art of Java* 一书，他还是 *Struts：The Complete Reference* 的作者以及 *JSF：The Complete*

前 言 **V**

*Reference* 的合著者之一。

Danny Coward 是本书第 8 版的技术编辑。他的忠告、洞察力和建议都有巨大的价值，对

此表示非常感谢。

如何进一步学习

《Java 完全参考手册(第 8 版)》开启了 Herb Schildt 编程图书系列的大门。下面是其他一些您可能感兴趣的图书。

要进一步学些 Java 编程，我们推荐下列书籍：

*Herb Schildt’s Java Programming Cookbook*

*Java：A Beginner’s Guide*

*Swing：A Beginner’s Guide*

*The Art of Java*

要学习 C++，您将会发现下列书籍特别有用：

*C++：The Complete Reference*

*Herb Schildt’s C++ Programming Cookbook*

*C++：A Beginner’s Guide*

*The Art of C++*

*C++ From the Ground Up*

*STL Programming From the Ground Up*

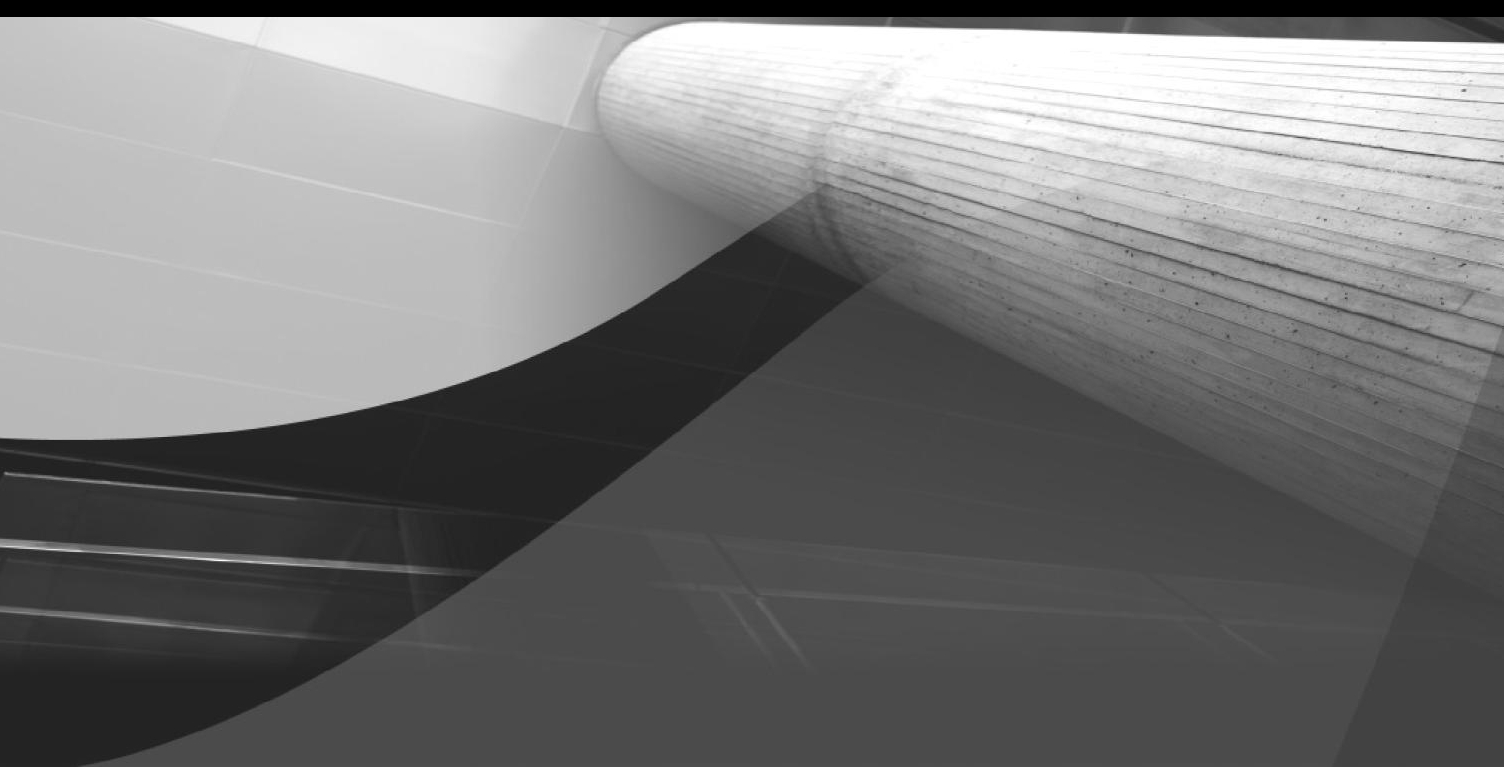
要学习 C#，我们建议阅读下列图书：

*C#：The Complete Reference*

*C#：A Beginner’s Guide*

要学习 C 语言，下面这本书您可能会感兴趣：

*C：The Complete Reference*



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  | 目 |  | 录 | |  |
|  | 第Ⅰ部分 | | Java 语言 |  |  |  |  | Java 的魔力：字节码 ................. 9 | | | | | |  |
|  |  |  |  | 1.4 |  |
| 第 1 章 | Java 的历史和演变 .................... | | | | | 3 | 1.5 | servlet：服务器端的 Java | | | | ......... | 10 |  |
|  |  |
| 1.6 | Java 的关键特性 | | | ....................... | | 10 |  |
| 1.1 | Java 的家世 | | ................................. | | | 4 |  |
|  |  |  |
|  |  | 简单性 ................................... | | | | 10 |  |
|  |  |  |  | 1.6.1 |  |
|  | 1.1.1 | 现代编程语言的诞生： | | |  |  |  |  |
|  |  |  |  | 1.6.2 | 面向对象 ............................... | | | | 11 |  |
|  |  | C 语言 ...................................... | | | | 4 |  |  |
|  |  |  | 1.6.3 | 健壮性 ................................... | | | | 11 |  |
|  | 1.1.2 | C++：下一个阶段 ................... | | | | 5 |  |  |
|  |  | 1.6.4 | 多线程 ................................... | | | | 11 |  |
|  | 1.1.3 | Java 出现的时机已经成熟 | | | ..... | 6 |  |  |
|  |  |  | 体系结构中立 ....................... | | | | 11 |  |
|  |  |  | 1.6.5 |  |
| 1.2 | Java 的诞生 | | ................................. | | | 6 |  |  |
|  |  |  |  | ................ | | 12 |  |
|  |  |  |  | 1.6.6 |  |  |  |
| 1.3 Java 改变 Internet 的方式 | | | | ........... | | 8 |  | 解释执行和高性能 | | |  |  |
|  |  | 分布式 ................................... | | | | 12 |  |
|  |  |  | 1.6.7 |  |
|  | 1.3.1 | Java applet ................................ | | | | 8 |  |  |
|  |  | 1.6.8 | 动态性 ................................... | | | | 12 |  |
|  | 1.3.2 | 安全性 ...................................... | | | | 8 |  |  |
|  | 1.7 | Java 的演变历程 | | | ....................... | | 12 |  |
|  |  | 可移植性 .................................. | | | | 9 |  |
|  | 1.3.3 |  |  |  |
|  | 1.8 | 文化革新 | | ................................... | | | 15 |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **VIII**Java 完全参考手册(第 8 版) | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 第 2 章 | Java 综述 | |  | 17 | |  | 3.7.2 | 浮点型字面值 ....................... | | | | | | | | 43 |  |
| ................................ |  |  |
| 2.1 | 面向对象编程 ........................... | | |  | 17 |  | 3.7.3 | 布尔型字面值 ....................... | | | | | | | | 44 |  |
|  | 2.1.1 | 两种范式 ................................ | |  | 18 |  | 3.7.4 | 字符型字面值 ....................... | | | | | | | | 44 |  |
|  | 2.1.2 | 抽象........................................ | |  | 18 |  | 3.7.5 | 字符串字面值 | | | | | | ....................... | | 44 |  |
|  |  |  |  |  |  |
|  | 2.1.3 | OOP 三原则 ........................... | |  | 19 | 3.8 | 变量 ........................................... | | | | | | | | | 45 |  |
| 2.2 | 第一个简单程序 ....................... | | |  | 23 |  | 3.8.1 | 变量的声明 ........................... | | | | | | | | 45 |  |
|  | 2.2.1 | 输入程序 ................................ | |  | 23 |  | 3.8.2 | 动态初始化 ........................... | | | | | | | | 45 |  |
|  | 2.2.2 | 编译程序 ................................ | |  | 23 |  | 3.8.3 | 变量的作用域和生命周期 | | | | | | | .... | 46 |  |
|  |  |  |  |  |
|  | 2.2.3 | 深入分析第一个示例程序 .... 24 | | | | 3.9 | 类型转换和强制类型转换 ........ | | | | | | | | | 48 |  |
| 2.3 | 第二个简短程序 ....................... | | |  | 26 |  | 3.9.1 | Java 的自动类型转换 ........... | | | | | | | | 49 |  |
| 2.4 | 两种控制语句 ........................... | | |  | 28 |  | 3.9.2 | 强制转换不兼容的类型 ........ | | | | | | | | 49 |  |
|  | 2.4.1 | if 语句 .................................... | |  | 28 | 3.10 | 表达式中的自动类型提升 ...... | | | | | | | | | 50 |  |
|  | 2.4.2 | for 循环 .................................. | |  | 29 | 3.11 | 数组 .......................................... | | | | | | | | | 52 |  |
| 2.5 | 使用代码块 ............................... | | |  | 30 |  | 3.11.1 | 一维数组 ............................ | | | | | | | | 52 |  |
| 2.6 | 词汇问题 | | ................................... |  | 32 |  | 3.11.2 | 多维数组 ............................ | | | | | | | | 54 |  |
|  | 2.6.1 | 空白符 .................................... | |  | 32 |  | 3.11.3 | 另一种数组声明语法 ........ | | | | | | | | 58 |  |
|  | 2.6.2 | 标识符 .................................... | |  | 32 | 3.12 | 关于字符串的一些说明 .......... | | | | | | | | | 59 |  |
|  | 2.6.3 | 字面值 .................................... 32 | | | | 3.13 | 向 C/C++程序员提供指针 | | | | | | | |  |  |  |
|  | 2.6.4 | 注释........................................ | |  | 32 |  | 方面的注解 | | | | | ............................. | | | | 60 |  |
|  |  |  |  |  |  |  |  |
|  | 2.6.5 | 分隔符 .................................... 33 | | | | 第 4 章 | 运算符 ...................................... 61 | | | | | | | | | |  |
|  | 2.6.6 | Java 关键字 ............................ 33 | | | |  |
|  | 4.1 |  |  |  | ................................ | | | | | | 61 |  |
| 2.7 | Java | 类库 | ................................... | | 34 | 算术运算符 | | | |  |  |  |  |  |  |
|  |  | 基本算术运算符 ................... | | | | | | | | 62 |  |
|  |  |  | 4.1.1 |  |
|  |  |  |  |  |  |  |  |
| 第 3 章 | 数据类型、变量和数组 ............ | | | 35 | |  | 4.1.2 | 求模运算符 ........................... | | | | | | | | 63 |  |
| 3.1 | Java 是强类型化的语言 ........... | | |  | 35 |  | 4.1.3 | 算术与赋值复合运算符 ........ | | | | | | | | 64 |  |
| 3.2 | 基本类型 | | ................................... |  | 36 |  | 4.1.4 |  |  |  |  |  |  | ................ | | 65 |  |
|  |  |  | 自增与自减运算符 | | | | | | |  |  |
| 3.3 | 整型 | ........................................... |  |  | 36 | 4.2 | 位运算符 | | ................................... | | | | | | | 66 |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | 3.3.1 | byte ......................................... |  |  | 37 |  | 4.2.1 | 位逻辑运算符 ....................... | | | | | | | | 68 |  |
|  | 3.3.2 | short........................................ | |  | 37 |  | 4.2.2 | 左移 | | ....................................... | | | | | | 70 |  |
|  |  |  |  |  |  |  |  |  |  |
|  | 3.3.3 | int ........................................... |  |  | 37 |  | 4.2.3 | 右移 ....................................... | | | | | | | | 71 |  |
|  | 3.3.4 | long | ........................................ |  | 37 |  | 4.2.4 | 无符号右移 ........................... | | | | | | | | 73 |  |
| 3.4 | 浮点型 ....................................... | |  |  | 38 |  | 4.2.5 | 位运算符与赋值的组合 ........ | | | | | | | | 74 |  |
|  | 3.4.1 | float ........................................ | |  | 38 | 4.3 |  |  |  | ................................ | | | | | | 75 |  |
|  |  | 关系运算符 | | | |  |  |  |  |  |  |
|  | 3.4.2 | double ..................................... | |  | 39 | 4.4 | 布尔逻辑运算符 ........................ | | | | | | | | | 76 |  |
| 3.5 | 字符型 ....................................... | |  |  | 39 | 4.5 |  |  |  | ................................ | | | | | | 78 |  |
|  |  | 赋值运算符 | | | |  |  |  |  |  |  |
| 3.6 | 布尔型 ....................................... | |  |  | 41 | 4.6 | “?”运算符 | | | | .............................. | | | | | 78 |  |
|  |  |  |  |  |  |  |  |
| 3.7 | 深入分析字面值 ....................... | | |  | 42 | 4.7 | 运算符的优先级 | | | | | | ........................ | | | 79 |  |
|  |  |  |  |  |
|  | 3.7.1 | 整型字面值 ............................ | |  | 42 | 4.8 |  |  |  | ................................ | | | | | | 80 |  |
|  |  | 使用圆括号 | | | |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 第 5 章 | 控制语句 .................................. 83 | | | | | | | | | | | |  |
| 5.1 | Java | 的选择语句 | | | | | | | ....................... | | | 83 |  |
|  |  |  |  |
|  | 5.1.1 | if 语句 .................................... | | | | | | | | | | 84 |  |
|  | 5.1.2 | switch 语句 ............................ | | | | | | | | | | 87 |  |
| 5.2 | 迭代语句 | | | ................................... | | | | | | | | 91 |  |
|  |  |  |  |  |  |  |  |  |
|  | 5.2.1 | while 语句 .............................. | | | | | | | | | | 92 |  |
|  | 5.2.2 | do-while | | | | |  |  | ......................... | | | 93 |  |
|  | 语句 | | |  |  |  |
|  | 5.2.3 | for 语句 .................................. | | | | | | | | | | 96 |  |
|  | 5.2.4 for 循环的 for-each 版本 .... | | | | | | | | | | 100 | |  |
|  | 5.2.5 | 嵌套的循环 ......................... | | | | | | | | | 105 | |  |
| 5.3 | 跳转语句 | | | ................................. | | | | | | | 106 | |  |
|  |  |  |  |  |  |  |  |
|  | 5.3.1 | 使用 break 语句 .................. | | | | | | | | | 106 | |  |
|  | 5.3.2 | 使用 continue 语句 .............. 110 | | | | | | | | | | |  |
|  | 5.3.3 | return 语句 ........................... 111 | | | | | | | | | | |  |
| 第 6 章 | 类........................................... 113 | | | | | | | | | | | |  |
| 6.1 | 类的基础知识 | | | | | | | ......................... | | | 114 | |  |
|  |  |  |  |
|  | 6.1.1 | 类的一般形式 ...................... 114 | | | | | | | | | | |  |
|  | 6.1.2 | 一个简单的类 ...................... 115 | | | | | | | | | | |  |
| 6.2 | 声明对象 | | | ................................. | | | | | | | 117 | |  |
|  |  |  |  |  |  |  |  |
| 6.3 | 为对象引用变量赋值 | | | | | | | | | ............. | 119 | |  |
|  |  |
| 6.4 | 方法 | ......................................... | | | | | | | | | 120 | |  |
|  |  |  |  |  |  |  |  |  |  |
|  | 6.4.1 | 为 Box 类添加方法 ............ | | | | | | | | | 120 | |  |
|  | 6.4.2 | 返回值 ................................. | | | | | | | | | 122 | |  |
|  | 6.4.3 | 添加带参数的方法 ............. | | | | | | | | | 124 | |  |
| 6.5 | 构造函数 | | | ................................. | | | | | | | 126 | |  |
|  |  |  |  |  |  |  |  |
| 6.6 | this 关键字 | | | | | .............................. | | | | | 129 | |  |
|  |  |  |  |  |  |
| 6.7 | 垃圾回收 | | | ................................. | | | | | | | 130 | |  |
|  |  |  |  |  |  |  |  |
| 6.8 | finalize()方法 | | | | | | ........................... | | | | 130 | |  |
|  |  |  |  |  |
| 6.9 | 堆栈类 | | ..................................... | | | | | | | | 131 | |  |
|  |  |  |  |  |  |  |  |  |
| 第 7 章 | 方法和类的深入分析.............. 135 | | | | | | | | | | | |  |
| 7.1 | 重载方法 | | | ................................. | | | | | | | 135 | |  |
|  |  |  |  |  |  |  |  |
| 7.2 | 将对象用作参数 | | | | | | | | ..................... | | 140 | |  |
|  |  |  |
| 7.3 | 参数传递的深入分析 | | | | | | | | | ............. | 143 | |  |
|  |  |
| 7.4 | 返回对象 | | | ................................. | | | | | | | 145 | |  |
|  |  |  |  |  |  |  |  |
| 7.5 | 递归 | ......................................... | | | | | | | | | 146 | |  |
|  |  |  |  |  |  |  |  |  |  |
| 7.6 | 访问控制 | | | ................................. | | | | | | | 148 | |  |
|  |  |  |  |  |  |  |  |
| 7.7 | 理解 static | | | | ................................ | | | | | | 151 | |  |
|  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  | 目 | 录**IX** |  |
| 7.8 | final 介绍 | | | ................................. | | | | | 153 |  |
|  |  |  |  |  |  |
| 7.9 |  |  |  | .......................... | | | | | 154 |  |
| 重新审视数组 | | | |  |  |  |  |  |
| 7.10 | 嵌套类和内部类 | | | | | | .................... | | 156 |  |
|  |  |  |
| 7.11 | String 类介绍 | | | | ......................... | | | | 158 |  |
|  |  |  |  |  |
| 7.12 | 使用命令行参数 | | | | | | .................... | | 161 |  |
|  |  |  |
| 7.13 | varargs：可变长度参数 | | | | | | | ........ | 162 |  |
|  |  |
|  | 7.13.1 | | 重载 varargs 方法 ............ | | | | | | 165 |  |
|  | 7.13.2 | | varargs 方法与模糊性 ..... | | | | | | 166 |  |
| 第 8 章 | 继承 ....................................... 169 | | | | | | | | |  |
| 8.1 | 继承的基础知识 | | | | | ...................... | | | 169 |  |
|  |  |  |  |
|  | 8.1.1 | 成员访问与继承 ................. | | | | | | | 171 |  |
|  | 8.1.2 |  |  |  |  |  | .............. | | 172 |  |
|  | 一个更实际的例子 | | | | | |  |  |

1. 超类变量可以引用子类

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 对象 ..................................... | | | | | | 174 |  |
| 8.2 | 使用 super 关键字 | | | ................... | | | | 175 |  |
|  |  |  |  |  |
|  | 8.2.1 | 使用 super 调用超类的 | | | | |  |  |  |
|  |  | 构造函数 ............................. | | | | | | 176 |  |
|  | 8.2.2 | super 的另一种用法 ............ | | | | | | 179 |  |
| 8.3 | 创建多级继承层次 | | | | .................. | | | 180 |  |
|  |  |  |  |
| 8.4 | 构造函数的调用时机 | | | | | .............. | | 183 |  |
|  |  |  |
| 8.5 | 方法重写 | | ................................. | | | | | 184 |  |
|  |  |  |  |  |  |
| 8.6 |  |  | .......................... | | | | | 187 |  |
| 动态方法调度 | | |  |  |  |  |  |
|  | 8.6.1 | 重写方法的目的 ................. | | | | | | 189 |  |
|  | 8.6.2 | 应用方法重写 ..................... | | | | | | 189 |  |
| 8.7 |  |  | .............................. | | | | | 191 |  |
| 使用抽象类 | | |  |  |  |  |  |
| 8.8 | 在继承中使用 final 关键字 | | | | | | .... | 193 |  |
|  |  |

1. 使用 final 关键字阻止

重写 ..................................... 194

1. 使用 final 关键字阻止

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 继承 ..................................... | | | 194 |  |
| 8.9 | Object 类 | | | .................................. | | 195 |  |
|  |  |  |
| 第 9 章 | 包和接口 ............................... 197 | | | | | |  |
| 9.1 | 包 | ............................................. | | | | 198 |  |
|  |  |  |  |  |
|  | 9.1.1 | | 定义包 ................................. | | | 198 |  |
|  | 9.1.2 | | 包查找与 CLASSPATH....... | | | 199 |  |
|  | 9.1.3 | |  |  | .............. | 199 |  |
|  | 一个简短的包示例 | | |  |
| 9.2 | 访问保护 | | | | ................................. | 200 |  |
|  |  |

**X** Java 完全参考手册(第 8 版)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9.3 | 导入包 | | | | ..................................... | | | | | | | | | | | 204 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 9.4 | 接口 | ......................................... | | | | | | | | | | | | | | 206 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 9.4.1 |  | 定义接口 ............................. | | | | | | | | | | | | | 207 |  |
|  | 9.4.2 |  | 实现接口 ............................. | | | | | | | | | | | | | 207 |  |
|  | 9.4.3 |  | 嵌套接口 ............................. | | | | | | | | | | | | | 210 |  |
|  | 9.4.4 |  | 应用接口 .............................. 211 | | | | | | | | | | | | | |  |
|  | 9.4.5 |  | 接口中的变量 ..................... | | | | | | | | | | | | | 214 |  |
|  | 9.4.6 |  | 接口可以扩展 ..................... | | | | | | | | | | | | | 216 |  |
| 第 10 章 | 异常处理 .............................. 219 | | | | | | | | | | | | | | | |  |
| 10.1 | 异常处理的基础知识 | | | | | | | | | | | | | ........... | | 219 |  |
|  |  |  |
| 10.2 | 异常类型 | | | | | | | ............................... | | | | | | | | 220 |  |
|  |  |  |  |  |  |  |  |  |
| 10.3 | 未捕获的异常 | | | | | | | | | ....................... | | | | | | 221 |  |
|  |  |  |  |  |  |  |
| 10.4 | 使用 try 和 catch | | | | | | | | | | | ................... | | | | 222 |  |
|  |  |  |  |  |
| 10.5 | 多条 catch | | | | | | | | 子句 | | ..................... | | | | | 224 |  |
|  |  |  |  |  |  |
| 10.6 | 嵌套的 try | | | | | | | | 语句 | | ..................... | | | | | 226 |  |
|  |  |  |  |  |  |
| 10.7 | throw | | | ...................................... | | | | | | | | | | | | 228 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 10.8 | throws | | | | | .................................... | | | | | | | | | | 229 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 10.9 | finally | | | | ..................................... | | | | | | | | | | | 230 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 10.10 | Java 的内置异常 | | | | | | | | | | | | ................. | | | 232 |  |
|  |  |  |  |
| 10.11 | 创建自己的异常子类 | | | | | | | | | | | | | | ......... | 233 |  |
|  |  |
| 10.12 | 链式异常 | | | | | | | | ............................. | | | | | | | 235 |  |
|  |  |  |  |  |  |  |  |
| 10.13 JDK 7 中 3 个新的 | | | | | | | | | | | | | |  |  |  |  |
|  | 异常特性 | | | | | | | | ............................. | | | | | | | 237 |  |
|  |  |  |  |  |  |  |  |  |
| 10.14 | 使用异常 | | | | | | | | ............................. | | | | | | | 238 |  |
|  |  |  |  |  |  |  |  |
| 第 11 章 | 多线程编程 .......................... 239 | | | | | | | | | | | | | | | |  |
| 11.1 | Java 线程模型 | | | | | | | | | ....................... | | | | | | 240 |  |
|  |  |  |  |  |  |  |
|  | 11.1.1 | |  |  |  |  | 线程优先级 ..................... | | | | | | | | | 241 |  |
|  | 11.1.2 | |  |  |  |  | 同步 ................................. | | | | | | | | | 242 |  |
|  | 11.1.3 | |  |  |  |  | 消息传递 ......................... | | | | | | | | | 242 |  |
|  | 11.1.4 | |  |  |  |  | Thread 类和 Runnable | | | | | | | | |  |  |
|  |  |  |  |  |  | 接口 ................................. | | | | | | | | | | 242 |  |
| 11.2 | 主线程 | | | | | | ................................... | | | | | | | | | 243 |  |
|  |  |  |  |  |  |  |  |  |  |
| 11.3 | 创建线程 | | | | | | | ............................... | | | | | | | | 244 |  |
|  |  |  |  |  |  |  |  |  |
|  | 11.3.1 | |  |  |  |  | 实现 Runnable 接口........ | | | | | | | | | 245 |  |
|  | 11.3.2 | |  |  |  |  | 扩展 Thread 类 ............... | | | | | | | | | 247 |  |
|  | 11.3.3 | |  |  |  |  | 选择一种创建方式 ......... | | | | | | | | | 248 |  |
| 11.4 | 创建多个线程 | | | | | | | | | ....................... | | | | | | 248 |  |
|  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11.5 | 使用 isAlive()和 join()方法 | | | | | .... | 250 |  |
|  |  |
| 11.6 | 线程优先级 | | | ............................ | | | 252 |  |
|  |  |  |  |
| 11.7 | 同步 | ........................................ | | | | | 253 |  |
|  |  |  |  |  |  |
|  | 11.7.1 | | 使用同步方法 ................. | | | | 253 |  |
|  | 11.7.2 | | synchronized 语句 ........... | | | | 255 |  |
| 11.8 | 线程间通信 | | | ............................ | | | 257 |  |
|  |  |  |  |
| 11.9 | 挂起、恢复与停止线程 | | | | ........ | | 263 |  |
|  |  |  |

1. Java 1.1 以前使用的挂起、

|  |  |  |  |
| --- | --- | --- | --- |
| 恢复和停止线程的方式 | .... | 264 |  |
|  |  |

1. 挂起、恢复与停止线程的

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 现代方式.......................... | | | | | 266 |  |
| 11.10 | 获取线程的状态 | | | | | .................. | 268 |  |
|  |  |
| 11.11 | 使用多线程 | | | .......................... | | | 270 |  |
|  |  |  |  |
| 第 12 章 | 枚举、自动装箱与注解 | | | | | |  |  |
|  | (元数据) ............................... 271 | | | | | | |  |
| 12.1 | 枚举 | ....................................... | | | | | 271 |  |
|  |  |  |  |  |  |
|  | 12.1.1 | 枚举的基础知识 ............. | | | | | 272 |  |
|  | 12.1.2 | values()和 valueOf() | | | | |  |  |
|  |  | 方法 ................................. | | | | | 274 |  |
|  | 12.1.3 | Java 枚举是类类型 ......... | | | | | 275 |  |
|  | 12.1.4 | 枚举继承自 Enum 类 ...... | | | | | 278 |  |
|  | 12.1.5 | 另一个枚举示例 ............. | | | | | 279 |  |
| 12.2 |  |  | ............................ | | | | 281 |  |
| 类型封装器 | | |  |  |  |  |
|  | 12.2.1 | Character 封装器 ............. | | | | | 281 |  |
|  | 12.2.2 | Boolean 封装器 ............... | | | | | 282 |  |
|  | 12.2.3 | 数值类型封装器 ............. | | | | | 282 |  |
| 12.3 | 自动装箱 | | ............................... | | | | 283 |  |
|  |  |  |  |  |
|  | 12.3.1 | 自动装箱与方法 ............. | | | | | 284 |  |
|  | 12.3.2 | 表达式中发生的 | | | | |  |  |
|  |  | 自动装箱/拆箱................. | | | | | 285 |  |
|  | 12.3.3 | 布尔型和字符型数值的 | | | | |  |  |
|  |  | 自动装箱/拆箱................. | | | | | 287 |  |
|  | 12.3.4 | 自动装箱/拆箱有助于 | | | | |  |  |
|  |  | 防止错误.......................... | | | | | 287 |  |
|  | 12.3.5 | 一些警告 ......................... | | | | | 288 |  |
| 12.4 注解(元数据) | | | | | ......................... | | 289 |  |
|  |  |  |
|  | 12.4.1 | 注解的基础知识 ............. | | | | | 289 |  |

12.4.2 指定保留策略 ................. 289

1. 在运行时使用反射获取

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 注解 ................................. | | | | | | | | | | | |  | 290 |  |
|  | 12.4.4 |  |  |  |  |  |  |  |  |  |  |  | .... | | 296 |  |
|  | AnnotatedElement 接口 | | | | | | | | | | | |  |  |
|  | 12.4.5 | 使用默认值 ..................... | | | | | | | | | | | |  | 296 |  |
|  | 12.4.6 |  |  |  |  |  |  | ......................... | | | | | |  | 297 |  |
|  | 标记注解 | | | | | | |  |  |  |  |  |  |  |
|  | 12.4.7 | 单成员注解 ..................... | | | | | | | | | | | |  | 298 |  |
|  | 12.4.8 |  |  |  |  |  |  | ......................... | | | | | |  | 300 |  |
|  | 内置注解 | | | | | | |  |  |  |  |  |  |  |
|  | 12.4.9 |  |  |  |  |  |  | ......................... | | | | | |  | 301 |  |
|  | 一些限制 | | | | | | |  |  |  |  |  |  |  |
| 第 13 章 | I/O、applet | | | |  |  |  |  |  |  |  | ...... | | 303 | |  |
| 以及其他主题 | | | | | | | |  |  |
| 13.1 | I/O 的基础知识 | | | | | | | | ..................... | | | | | | 304 |  |
|  |  |  |  |  |  |  |
|  | 13.1.1 | 流 .................................... | | | | | | | | | | | |  | 304 |  |
|  | 13.1.2 | 字节流和字符流 ............. | | | | | | | | | | | |  | 304 |  |
|  | 13.1.3 |  |  |  |  |  |  | ......................... | | | | | |  | 306 |  |
|  | 预定义流 | | | | | | |  |  |  |  |  |  |  |
| 13.2 | 读取控制台输入 | | | | | | | | | ................... | | | | | 307 |  |
|  |  |  |  |  |  |
|  | 13.2.1 |  |  |  |  |  |  | ......................... | | | | | |  | 307 |  |
|  | 读取字符 | | | | | | |  |  |  |  |  |  |  |
|  | 13.2.2 | 读取字符串 ..................... | | | | | | | | | | | |  | 308 |  |
| 13.3 | 向控制台写输出 | | | | | | | | | ................... | | | | | 310 |  |
|  |  |  |  |  |  |
| 13.4 | PrintWriter | | | | 类 | | ........................ | | | | | | | | 311 |  |
|  |  |  |  |  |  |  |  |  |
| 13.5 | 读/写文件 | | | .............................. | | | | | | | | | | | 312 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 13.6 | 自动关闭文件 | | | | | | | ....................... | | | | | | | 318 |  |
|  |  |  |  |  |  |  |  |
| 13.7 |  |  |  |  |  |  |  |  |  | ................. | | | | | 321 |  |
| applet 的基础知识 | | | | | | | | | |  |  |  |  |  |
| 13.8 transient 和 volatile 修饰符 | | | | | | | | | | | | | .... | | 324 |  |
|  |  |  |
| 13.9 | 使用 instanceof | | | | | | | | 运算符 | | | ......... | | | 324 |  |
|  |  |  |  |
| 13.10 | strictfp | | .................................. | | | | | | | | | | | | 327 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 13.11 | 本地方法 | | | | ............................. | | | | | | | | | | 327 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 13.12 | 使用断言 | | | | ............................. | | | | | | | | | | 331 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 13.13 | 静态导入 | | | | ............................. | | | | | | | | | | 334 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 13.14 | 通过 this()调用重载的 | | | | | | | | | | | |  |  |  |  |
|  | 构造函数 | | | | ............................. | | | | | | | | | | 336 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 第 14 章 | 泛型 ..................................... | | | | | | | | | | | | | 339 | |  |
| 14.1 | 什么是泛型 | | | | | ........................... | | | | | | | | | 340 |  |
|  |  |  |  |  |  |  |  |  |  |
| 14.2 | 一个简单的泛型示例 | | | | | | | | | | ........... | | | | 340 |  |
|  |  |  |  |  |
|  | 14.2.1 | 泛型只使用对象 ............. | | | | | | | | | | | |  | 344 |  |

1. 基于不同类型参数的泛型类型是不同的 ......... 344

目 录 **XI**

1. 泛型提升类型安全性的

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 原理 ................................. | | | | | | 344 |  |
| 14.3 | 带两个类型参数的泛型类 | | | | | .... | 347 |  |
|  |  |
| 14.4 | 泛型类的一般形式 | | | ................ | | | 348 |  |
|  |  |  |  |
| 14.5 | 有界类型 | ............................... | | | | | 349 |  |
|  |  |  |  |  |  |
| 14.6 | 使用通配符参数 | | .................... | | | | 351 |  |
|  |  |  |  |  |
| 14.7 |  | ........................ | | | | | 359 |  |
| 创建泛型方法 | |  |  |  |  |  |
| 14.8 | 泛型接口 | ............................... | | | | | 362 |  |
|  |  |  |  |  |  |
| 14.9 | 原始类型与遗留代码 | | | | ............ | | 364 |  |
|  |  |  |
| 14.10 |  | .......................... | | | | | 367 |  |
| 泛型类层次 | |  |  |  |  |  |

1. 使用泛型超类 ............. 367

14.10.2 泛型子类 ..................... 369

1. 泛型层次中的运行时

类型比较 ..................... 370 14.10.4 强制转换 ..................... 373

1. 重写泛型类的方法 ..... 373

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 14.11 | 泛型的类型推断 | | .................. | 374 |  |
|  |  |
| 14.12 | 擦拭 | ..................................... | | 376 |  |
|  |  |  |
| 14.13 |  | .......................... | | 379 |  |
| 模糊性错误 | |  |  |

1. 使用泛型的一些限制 .......... 381
   1. 不能实例化类型参数 ... 381
   2. 对静态成员的一些

限制 ............................. 381

1. 对泛型数组的一些

限制 ............................. 382

1. 对泛型异常的限制 ..... 383

第Ⅱ部分 Java 库

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 第 15 章 | 字符串处理 .......................... 387 | | | | |  |
| 15.1 | String 类的构造函数 | | | ............. | 388 |  |
|  |  |
| 15.2 |  | ........................ | | | 390 |  |
| 字符串的长度 | |  |  |  |
| 15.3 | 特殊的字符串操作 | | ................ | | 391 |  |
|  |  |  |
|  | 15.3.1 | 字符串字面值 ................. | | | 391 |  |
|  | 15.3.2 | 字符串连接 ..................... | | | 391 |  |
|  | 15.3.3 | 字符串和其他数据 | | |  |  |
|  |  | 类型的连接 ..................... | | | 392 |  |
|  | 15.3.4 | 字符串转换和 toString() | | |  |  |
|  |  | 方法 ................................. | | | 393 |  |

**XII** Java 完全参考手册(第 8 版)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15.4 | 提取字符 | | | ............................... | | | | | 394 |  |
|  |  |  |  |  |  |
|  | 15.4.1 |  | charAt() ........................... | | | | | | 394 |  |
|  | 15.4.2 |  | getChars() ........................ | | | | | | 394 |  |
|  | 15.4.3 |  | getBytes() ........................ | | | | | | 395 |  |
|  | 15.4.4 | toCharArray() .................. | | | | | | | 395 |  |
| 15.5 | 比较字符串 | | | | ........................... | | | | 395 |  |
|  |  |  |  |  |
|  | 15.5.1 |  | equals()和 | | |  |  |  |  |  |
|  |  | equalsIgnoreCase() .......... | | | | | | | 395 |  |
|  | 15.5.2 | regionMatches() .............. | | | | | | | 396 |  |
|  | 15.5.3 |  | startsWith()和 | | | | |  |  |  |
|  |  | endsWith() ....................... | | | | | | | 397 |  |
|  | 15.5.4 equals()与== ................... | | | | | | | | 397 |  |
|  | 15.5.5 |  | compareTo() .................... | | | | | | 398 |  |
| 15.6 | 查找字符串 | | | | ........................... | | | | 399 |  |
|  |  |  |  |  |
| 15.7 | 修改字符串 | | | | ........................... | | | | 401 |  |
|  |  |  |  |  |
|  | 15.7.1 |  | substring() ....................... | | | | | | 401 |  |
|  | 15.7.2 |  | concat() ........................... | | | | | | 402 |  |
|  | 15.7.3 | replace() .......................... | | | | | | | 402 |  |
|  | 15.7.4 | trim() ............................... | | | | | | | 403 |  |
| 15.8 | 使用 valueOf()转换数据 | | | | | | | ....... | 404 |  |
|  |  |
| 15.9 | 改变字符串中字符的 | | | | | | |  |  |  |
|  | 大小写 | | ................................... | | | | | | 404 |  |
|  |  |  |  |  |  |  |  |
| 15.10 | 其他 String 方法 | | | | | | .................. | | 405 |  |
|  |  |  |
| 15.11 | StringBuffer 类 | | | | | .................... | | | 406 |  |
|  |  |  |  |

1. StringBuffer 类的

构造函数 .................... 406

15.11.2 length()与 capacity() ... 407

1. ensureCapacity() ......... 407

15.11.4 setLength() .................. 408

1. charAt()与

|  |  |
| --- | --- |
|  | setCharAt() ................. 408 |
| 15.11.6 | getChars() ................... 409 |
| 15.11.7 | append() ...................... 409 |
| 15.11.8 | insert() ......................... 409 |
| 15.11.9 | reverse() ...................... 410 |
| 15.11.10 | delete()与 |
|  | deleteCharAt()............ 411 |
| 15.11.11 | replace() ..................... 411 |

15.11.12 substring() .................. 412

1. 其他 StringBuffer

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 方法 ........................... | | | 412 |  |
| 15.12 | StringBuilder 类 | | | ................... | 413 |  |
|  |  |
| 第 16 章 | 探究 java.lang ..................... 415 | | | | |  |
| 16.1 | 基本类型封装器 | | .................... | | 416 |  |
|  |  |  |
|  | 16.1.1 | Number ............................ | | | 416 |  |
|  | 16.1.2 | Double 与 Float ............... | | | 417 |  |
|  | 16.1.3 | 理解 isInfinite()与 | | |  |  |
|  |  | isNaN() ............................ | | | 420 |  |
|  | 16.1.4 | Byte、Short、Integer 和 | | |  |  |
|  |  | Long ................................. | | | 420 |  |
|  | 16.1.5 | Character .......................... | | | 427 |  |

1. 对 Unicode 代码点的

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 附加支持 ......................... | | | | | | | | | 430 |  |
|  | 16.1.7 | Boolean ............................ | | | | | | | | | 431 |  |
| 16.2 | Void 类 | | ................................... | | | | | | | | 432 |  |
|  |  |  |  |  |  |  |  |  |
| 16.3 | Process | | 类 | | | | .............................. | | | | 432 |  |
|  |  |  |  |  |
| 16.4 | Runtime | | | 类 | | | | ............................. | | | 433 |  |
|  |  |  |  |
|  | 16.4.1 | 内存管理 ......................... | | | | | | | | | 434 |  |
|  | 16.4.2 | 执行其他程序 ................. | | | | | | | | | 435 |  |
| 16.5 | ProcessBuilder 类 | | | | | | | | | .................. | 436 |  |
|  |  |
| 16.6 | System | | 类 | | | ............................... | | | | | 439 |  |
|  |  |  |  |  |  |
|  | 16.6.1 | 使用 currentTimeMillis() | | | | | | | | |  |  |
|  |  | 计时程序的执行 ............. | | | | | | | | | 440 |  |
|  | 16.6.2 | 使用 arraycopy() .............. | | | | | | | | | 441 |  |
|  | 16.6.3 | 环境属性 ......................... | | | | | | | | | 442 |  |
| 16.7 | Object 类 | | | | ................................ | | | | | | 442 |  |
|  |  |  |  |  |  |  |
| 16.8 | 使用 clone()方法和 Cloneable | | | | | | | | | | |  |
|  | 接口 | ....................................... | | | | | | | | | 443 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 16.9 | Class 类 | | | .................................. | | | | | | | 445 |  |
|  |  |  |  |  |  |  |  |
| 16.10 | ClassLoader 类 | | | | | | | | ..................... | | 448 |  |
|  |  |  |
| 16.11 | Math 类 | | | | ................................ | | | | | | 448 |  |
|  |  |  |  |  |  |  |
|  | 16.11.1 | |  |  | 三角函数 ..................... | | | | | | 448 |  |
|  | 16.11.2 | |  |  | 指数函数 ..................... | | | | | | 449 |  |
|  | 16.11.3 | |  |  | 舍入函数 ..................... | | | | | | 449 |  |
|  | 16.11.4 | |  |  | 其他数学方法 ............. | | | | | | 450 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16.12 | StrictMath 类 | | | | | | | ....................... | | | | | | | | | | 452 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 16.13 | Compiler 类 | | | | | ......................... | | | | | | | | | | | | 452 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 16.14 | Thread 类、ThreadGroup 类 | | | | | | | | | | | | | | | | | |  |
|  | 和 Runnable 接口 | | | | | | | | | | | ................ | | | | | | 452 |  |
|  |  |  |  |  |  |  |  |
|  | 16.14.1 | Runnable 接口 ............ | | | | | | | | | | | | | | |  | 452 |  |
|  | 16.14.2 | Thread 类 .................... | | | | | | | | | | | | | | |  | 452 |  |
|  | 16.14.3 | ThreadGroup 类 .......... | | | | | | | | | | | | | | |  | 454 |  |
| 16.15 | ThreadLocal 和 | | | | | | | | | |  |  |  |  |  |  |  |  |  |
|  | InheritableThreadLocal 类 | | | | | | | | | | | | | | | .... | | 459 |  |
|  |  |  |  |
| 16.16 | Package | | 类 | | ........................... | | | | | | | | | | | | | 459 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 16.17 | RuntimePermission | | | | | | | | | | | | 类 | | ......... | | | 460 |  |
|  |  |  |  |
| 16.18 | Throwable 类 | | | | | | | ....................... | | | | | | | | | | 460 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 16.19 | SecurityManager 类 | | | | | | | | | | | |  | ............. | | | | 460 |  |
|  |  |  |  |  |  |
| 16.20 | StackTraceElement 类 | | | | | | | | | | | | | | ......... | | | 460 |  |
|  |  |  |  |
| 16.21 | Enum | 类 | | ............................... | | | | | | | | | | | | | | 461 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 16.22 | ClassValue 类 | | | | | | | | ...................... | | | | | | | | | 462 |  |
|  |  |  |  |  |  |  |  |  |  |
| 16.23 | CharSequence | | | | | | | |  | 接口 | | |  | ............. | | | | 462 |  |
|  |  |  |  |  |  |  |
| 16.24 | Comparable | | | | | 接口 | | | | | ................. | | | | | | | 462 |  |
|  |  |  |  |  |  |  |  |
| 16.25 | Appendable | | | | | 接口 | | | | | ................. | | | | | | | 463 |  |
|  |  |  |  |  |  |  |  |
| 16.26 | Iterable 接口 | | | | | | ........................ | | | | | | | | | | | 463 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 16.27 | Readable 接口 | | | | | | | | | ..................... | | | | | | | | 464 |  |
|  |  |  |  |  |  |  |  |  |
| 16.28 | AutoCloseable | | | | | | | | | 接口 | | | | ............. | | | | 464 |  |
|  |  |  |  |  |
| 16.29 | Thread.UncaughtException- | | | | | | | | | | | | | | | |  |  |  |
|  | Handler 接口 | | | | | | | ....................... | | | | | | | | | | 464 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 16.30 | java.lang 子包 | | | | | | | | ...................... | | | | | | | | | 464 |  |
|  |  |  |  |  |  |  |  |  |  |
|  | 16.30.1 | java.lang.annotation .... | | | | | | | | | | | | | | |  | 465 |  |
|  | 16.30.2 | java.lang.instrument .... | | | | | | | | | | | | | | |  | 465 |  |
|  | 16.30.3 | java.lang.invoke .......... | | | | | | | | | | | | | | |  | 465 |  |
|  | 16.30.4 | java.lang.management ... 465 | | | | | | | | | | | | | | | | |  |
|  | 16.30.5 | java.lang.ref ................ | | | | | | | | | | | | | | |  | 465 |  |
|  | 16.30.6 | java.lang.reflect........... | | | | | | | | | | | | | | |  | 465 |  |
| 第 17 章 | java.util 第 1 部分： | | | | | | | | | | | |  |  |  |  |  |  |  |
|  | 集合框架 .............................. | | | | | | | | | | | | | | | | 467 | |  |
| 17.1 | 集合概述 | |  | ............................... | | | | | | | | | | | | | | 468 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 17.2 | JDK 5 对集合框架的修改 | | | | | | | | | | | | | | | .... | | 469 |  |
|  |  |  |

1. 泛型从根本上改变了

集合框架 ......................... 470

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | 目 | 录**XIII** |  |
|  | 17.2.2 |  | 自动装箱使得使用基本 | | |  |  |
|  |  | 类型更加容易 ................. | | | | 470 |  |
|  | 17.2.3 |  | for-each 风格的循环 ....... | | | 470 |  |
| 17.3 | 集合接口 | | | ............................... | | 470 |  |
|  |  |  |
|  | 17.3.1 |  | Collection 接口 ................ | | | 471 |  |
|  | 17.3.2 |  | List 接口 .......................... | | | 473 |  |
|  | 17.3.3 | Set 接口 ........................... | | | | 474 |  |
|  | 17.3.4 | SortedSet 接口 ................. | | | | 474 |  |
|  | 17.3.5 |  | NavigableSet 接口 ........... | | | 475 |  |
|  | 17.3.6 |  | Queue 接口 ...................... | | | 476 |  |
|  | 17.3.7 |  | Deque 接口 ...................... | | | 477 |  |
| 17.4 | 集合类 | | ................................... | | | 479 |  |
|  |  |  |  |
|  | 17.4.1 |  | ArrayList 类 ..................... | | | 480 |  |
|  | 17.4.2 |  | LinkedList 类 ................... | | | 483 |  |
|  | 17.4.3 |  | HashSet 类 ....................... | | | 484 |  |
|  | 17.4.4 |  | LinkedHashSet 类 ............ | | | 486 |  |
|  | 17.4.5 |  | TreeSet 类 ........................ | | | 486 |  |
|  | 17.4.6 | PriorityQueue 类 .............. | | | | 487 |  |
|  | 17.4.7 |  | ArrayDeque 类 ................. | | | 488 |  |
|  | 17.4.8 | EnumSet 类 ...................... | | | | 489 |  |
| 17.5 | 通过迭代器访问集合 | | | | ............ | 490 |  |
|  |  |
|  | 17.5.1 |  | 使用迭代器 ..................... | | | 491 |  |

1. 使用 for-each 循环替代

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 迭代器 ............................. | | | | | | | 493 |  |
| 17.6 | 在集合中存储用户 | | | | | |  |  |  |  |
|  |  |  | ................................ | | | | | | 494 |  |
|  | 定义的类 | | |  |  |  |  |  |  |
| 17.7 | RandomAccess 接口 | | | | | | .............. | | 495 |  |
|  |  |  |
| 17.8 | 使用映射 | | | ............................... | | | | | 496 |  |
|  |  |  |  |  |  |
|  | 17.8.1 | 映射接口 ......................... | | | | | | | 496 |  |
|  | 17.8.2 | 映射类 ............................. | | | | | | | 500 |  |
| 17.9 | 比较器 | | ................................... | | | | | | 505 |  |
|  |  |  |  |  |  |  |
| 17.10 |  |  |  |  | .............................. | | | | 508 |  |
| 集合算法 | | | | |  |  |  |  |
| 17.11 | Arrays 类 | | | | .............................. | | | | 513 |  |
|  |  |  |  |  |
| 17.12 | 需要泛型集合的原因 | | | | | | | .......... | 518 |  |
|  |  |
| 17.13 | 遗留的类和接口 | | | | | .................. | | | 521 |  |
|  |  |  |  |
|  | 17.13.1 | | Enumeration 接口 ....... | | | | | | 521 |  |
|  | 17.13.2 | | Vector 类 ..................... | | | | | | 521 |  |
|  | 17.13.3 | | Stack 类 ....................... | | | | | | 525 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **XIV**Java 完全参考手册(第 8 版) | | | | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 17.13.4 |  | Dictionary 类 | | | | | | | | |  |  | 527 |  | 18.13.14 “#：”标志 ............... | | | | | | | | 574 |  |
|  |  |  |  |  |  |
|  | 17.13.5 |  | Hashtable 类 ............... | | | | | | | | |  |  | 528 |  | 18.13.15 | | 大写选项 ................... | | | | | | 574 |  |
|  | 17.13.6 |  | Properties 类 ............... | | | | | | | | |  |  | 532 |  | 18.13.16 | | 使用参数索引 ........... | | | | | | 575 |  |
|  | 17.13.7 |  | 使用 store()和 load() .... | | | | | | | | | | | 535 |  | 18.13.17 | | 关闭 Formatter 对象 ... 577 | | | | | | |  |
| 17.14 | 集合小结 | | | | ............................. |  |  |  |  |  |  |  |  | 537 |  | 18.13.18 | | printf()方法 ............... | | | | | | 577 |  |
| 第 18 章 | java.util 第 2 部分：更多实用 | | | | | | | | | | | | |  | 18.14 | Scanner 类 | | | ............................ | | | | | 577 |  |
|  |  |  |  |  |  |  |
|  |  | 18.14.1 | | Scanner 类的 | | |  |  |  |  |  |
|  | 工具类 ................................. 539 | | | | | | | | | | | | | |  |  |  |  |  |  |
|  |  |  |  | 构造函数 ..................... | | | | | | 577 |  |
| 18.1 | StringTokenizer 类 | | | | | | | | | ................. | | | | 539 |  |  |  |  |
|  |  |  | 扫描的基础知识 ......... | | | | | | 578 |  |
|  |  |  |  |  | 18.14.2 | |  |
| 18.2 | BitSet 类 | | ................................ | | | | | | | | | | | 541 |  |  |
|  |  |  | 一些 Scanner 示例 ...... | | | | | | 581 |  |
|  |  |  |  |  |  |  |  |  |  |  |  | 18.14.3 | |  |
| 18.3 | Date 类 | ................................... | | | | | | | | | | | | 544 |  |  |
|  |  |  | 设置定界符 ................. | | | | | | 585 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  | 18.14.4 | |  |
| 18.4 | Calendar | | 类 | | | ............................ | | | | | | | | 546 |  |  |
|  |  |  | 其他 Scanner 特性 ...... | | | | | | 587 |  |
|  |  |  |  |  |  |  |  |  | 18.14.5 | |  |
| 18.5 | GregorianCalendar | | | | | | | | | 类 | | ............ | | 549 |  |  |
| 18.15 | ResourceBundle、 | | | | |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 18.6 | TimeZone 类 | | | | | | .......................... | | | | | | | 550 |  |  |  |  |  |
|  | ListResourceBundle 和 | | | | |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 18.7 | SimpleTimeZone 类 | | | | | | | | | | .............. | | | 551 |  |  |  |  |  |  |
|  |  |  |  |  |  |  | .... | | 588 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 18.8 | Locale 类 | | | ............................... | | | | | | | | | | 552 |  | PropertyResourceBundle 类 | | | | | | |  |  |
| 18.16 | 其他实用工具类和接口 | | | | | | ...... | | 592 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 18.9 | Random 类 | | | | ............................. | | | | | | | | | 554 |  |
|  |  |  |
| 18.17 | java.util 子包 | | | | ........................ | | | | 593 |  |
|  |  |  |  |  |  |  |  |  |  |
| 18.10 | Observable 类 | | | | | | | | ...................... | | | | | 555 |  |
|  |  |  |  |  |
|  | 18.17.1 | | java.util.concurrent、 | | | | |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  | 18.10.1 |  | Observer 接口 ............. | | | | | | | | | | | 556 |  |  |  |  |
|  |  |  |  |  | java.util.concurrent.atomic | | | | | | |  |
|  | 18.10.2 |  | Observer 示例 ............. | | | | | | | | | | | 556 |  |  |  |  |
|  |  |  |  |  | 和 java.util.concurrent. | | | | | |  |  |
| 18.11 Timer 和 TimerTask 类 | | | | | | | | | | | | | ........ | 559 |  |  |  |  |  |
|  |  |  | locks ............................ | | | | | | 593 |  |
|  |  |  |  |  |
| 18.12 | Currency 类 | | | | | | | ......................... | | | | | | 561 |  |  |  |  |
|  |  |  | java.util.jar ................... | | | | | | 593 |  |
|  |  |  |  |  |  |  | 18.17.2 | |  |
| 18.13 | Formatter | | | | 类 | | | ........................ | | | | | | 562 |  |  |
|  |  |  | java.util.logging ........... | | | | | | 593 |  |
|  |  |  |  |  |  |  | 18.17.3 | |  |
|  | 18.13.1 |  | Formatter 类的 | | | | | | | | | |  |  |  |  |
|  |  |  |  |  | 18.17.4 | | java.util.prefs ............... | | | | | | 593 |  |
|  |  |  | 构造函数 .................... | | | | | | | | | | | 563 |  |  |
|  |  |  |  | 18.17.5 | | java.util.regex .............. | | | | | | 594 |  |
|  | 18.13.2 |  | Formatter 类的方法 .... | | | | | | | | | | | 563 |  |  |
|  |  |  | 18.17.6 | | java.util.spi .................. | | | | | | 594 |  |
|  | 18.13.3 |  | 格式化的基础知识 .... | | | | | | | | | | | 564 |  |  |
|  |  |  | 18.17.7 | | java.util.zip .................. | | | | | | 594 |  |
|  | 18.13.4 |  | 格式化字符串和字符 ... | | | | | | | | | | | 566 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 18.13.5 |  | 格式化数字 | | | | | | | | ................ |  |  | 566 | 第 19 章 | 输入/输出：探究 Java.io | | | | | ...... | | 595 | |  |
|  |  |  |  |  |  |  |  |
|  | 18.13.6 |  | 格式化时间和日期 .... | | | | | | | | | | | 567 | 19.1 | I/O 类和接口 ......................... | | | | | | | | 596 |  |
|  | 18.13.7 |  | %n 和%%说明符 ....... | | | | | | | | | | | 569 | 19.2 | File 类 .................................... | | | | | | | | 597 |  |
|  | 18.13.8 |  | 指定最小字段宽度 .... | | | | | | | | | | | 569 |  | 19.2.1 | 目录 ................................. | | | | | | | 600 |  |
|  | 18.13.9 |  | 指定精度 .................... | | | | | | | | | | | 571 |  | 19.2.2 | 使用 FilenameFilter | | | |  |  |  |  |  |
|  | 18.13.10 | |  | 使用格式标志 .......... | | | | | | | | |  | 572 |  |  | 接口 ................................. | | | | | | | 601 |  |
|  | 18.13.11 | |  | 对齐输出 .................. | | | | | | |  |  |  | 572 |  | 19.2.3 | listFiles()方法 .................. | | | | | | | 602 |  |
|  | 18.13.12 空格、“+”、“0”以及 | | | | | | | | | | | | | |  | 19.2.4 | 创建目录 ......................... | | | | | | | 602 |  |
|  |  |  | “(”标志 ................... | | | | | | | | | | | 573 | 19.3 | AutoCloseable、Closeable 和 | | | | | | | | |  |
|  | 18.13.13 | |  | 逗号标志 .................. | | | | | | |  |  |  | 574 |  | Flushable 接口 ....................... | | | | | | | | 602 |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 19.4 | I/O 异常 | | | ................................. | | 603 |  |
|  |  |  |
| 19.5 | 关闭流的两种方式 | | | | ............... | 604 |  |
|  |  |
| 19.6 | 流类 | ....................................... | | | | 605 |  |
|  |  |  |  |  |
| 19.7 | 字节流 | | ................................... | | | 605 |  |
|  |  |  |  |
|  | 19.7.1 | InputStream 类 ................ | | | | 605 |  |
|  | 19.7.2 |  | OutputStream 类 ............. | | | 606 |  |
|  | 19.7.3 | FileInputStream 类 .......... | | | | 606 |  |
|  | 19.7.4 | FileOutputStream 类 ....... | | | | 609 |  |
|  | 19.7.5 |  | ByteArrayInputStream | | |  |  |
|  |  | 类 ..................................... 611 | | | | |  |

1. ByteArrayOutputStream

|  |  |  |  |
| --- | --- | --- | --- |
|  | 类 .................................... | 613 |  |
| 19.7.7 | 过滤的字节流 ................. | 614 |  |
| 19.7.8 | 缓存的字节流 ................. | 614 |  |
| 19.7.9 | .... | 618 |  |
| SequenceInputStream 类 |  |
| 19.7.10 | PrintStream 类 ............... | 620 |  |
| 19.7.11 | DataOutputStream 和 |  |  |
|  | DataInputStream 类 ...... | 622 |  |

1. RandomAccessFile 类 ... 624

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 19.8 | 字符流 | | ................................... | | | 625 |  |
|  |  |  |  |
|  | 19.8.1 |  | Reader 类 ........................ | | | 625 |  |
|  | 19.8.2 |  | Writer 类 ......................... | | | 626 |  |
|  | 19.8.3 | FileReader 类 .................. | | | | 627 |  |
|  | 19.8.4 | FileWriter 类 ................... | | | | 627 |  |
|  | 19.8.5 |  | CharArrayReader 类 ....... | | | 628 |  |
|  | 19.8.6 |  | CharArrayWriter 类 ........ | | | 630 |  |
|  | 19.8.7 |  | BufferedReader 类 .......... | | | 631 |  |
|  | 19.8.8 |  | BufferedWriter 类 ........... | | | 633 |  |
|  | 19.8.9 | PushbackReader 类 ......... | | | | 633 |  |
|  | 19.8.10 |  | PrintWriter 类 ................ | | | 634 |  |
| 19.9 | Console | | | 类 | ............................. | 635 |  |
|  |  |
| 19.10 | 串行化 | | | ................................. | | 637 |  |
|  |  |  |
|  | 19.10.1 | | | Serializable 接口 ........ | | 637 |  |
|  | 19.10.2 | | | Externalizable 接口 .... | | 637 |  |
|  | 19.10.3 | | | ObjectOutput 接口...... | | 638 |  |
|  | 19.10.4 | | | ObjectOutputStream | |  |  |
|  |  |  |  | 类 ................................ | | 638 |  |
|  | 19.10.5 | | | ObjectInput 接口 ........ | | 639 |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | 目 | | 录**XV** |  |
|  | 19.10.6 | | ObjectInputStream 类 | | | | | .... | 640 |  |
|  |  |  |
|  | 19.10.7 | | 串行化示例 ................. | | | | | | 641 |  |
| 19.11 | 流的优点 | | | .............................. | | | | | 642 |  |
|  |  |  |  |  |  |
| 第 20 章 | 探究 NIO ............................. 643 | | | | | | | | |  |
| 20.1 | NIO 类 | | ................................... | | | | | | 643 |  |
|  |  |  |  |  |  |  |
| 20.2 | NIO 的基础知识 | | | | | ................... | | | 644 |  |
|  |  |  |  |
|  | 20.2.1 | 缓存 ................................. | | | | | | | 644 |  |
|  | 20.2.2 | 通道 ................................. | | | | | | | 646 |  |
|  | 20.2.3 | 字符集和选择器 ............. | | | | | | | 647 |  |
| 20.3 | JDK 7 | 对 NIO 的增强 | | | | | ........... | | 648 |  |
|  |  |  |
|  | 20.3.1 | Path 接口 ......................... | | | | | | | 648 |  |
|  | 20.3.2 | Files 类 ............................ | | | | | | | 649 |  |
|  | 20.3.3 | Path 接口 ......................... | | | | | | | 651 |  |
|  | 20.3.4 | 文件属性接口 ................. | | | | | | | 652 |  |
|  | 20.3.5 FileSystem、FileSystems | | | | | | | |  |  |
|  |  | 和 FileStore 类 ................ | | | | | | | 654 |  |
| 20.4 | 使用 NIO 系统 | | | | ...................... | | | | 654 |  |
|  |  |  |  |  |

1. 为基于通道的 I/O 使用

NIO .................................. 654

1. 为基于流的 I/O

使用 NIO ......................... 663

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 20.4.3 | 为路径和文件系统操作 | | | | |  |  |
|  |  | 使用 NIO ......................... | | | | | 666 |  |
| 20.5 | JDK 7 之前基于通道的 | | | | | |  |  |
|  | 例子 | ....................................... | | | | | 674 |  |
|  |  |  |  |  |  |  |
|  | 20.5.1 | 读文件(JDK 7 之前) ....... | | | | | 674 |  |
|  | 20.5.2 | 写文件(JDK 7 之前) ....... | | | | | 677 |  |
| 第 21 章 | 联网 ..................................... 681 | | | | | | |  |
| 21.1 | 联网的基础知识 | | | | .................... | | 682 |  |
|  |  |  |
| 21.2 |  |  | ........................ | | | | 683 |  |
| 联网类和接口 | | |  |  |  |  |
| 21.3 | InetAddress 类 | | | ....................... | | | 683 |  |
|  |  |  |  |
|  | 21.3.1 | 工厂方法 ......................... | | | | | 684 |  |
|  | 21.3.2 | 实例方法 ......................... | | | | | 685 |  |
| 21.4 | Inet4Address 类和 | | | | |  |  |  |
|  |  |  |  | ...................... | | | 685 |  |
|  | Inet6Address 类 | | | |  |  |  |
| 21.5 | TCP/IP | | 客户端套接字 | | | ........... | 686 |  |
|  |  |
| 21.6 | URL | 类 | .................................. | | | | 689 |  |
|  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **XVI**Java 完全参考手册(第 8 版) | | | | | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 21.7 | URLConnection 类 | | | | | | | | |  |  |  | 691 |  | 23.3.4 |  | ContainerEvent | | | | | | | 类 | ........... | 727 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| 21.8 | HttpURLConnection 类 ......... | | | | | | | | | | |  | 694 |  | 23.3.5 | FocusEvent 类 | | | | | | | ................. | | | 727 |  |
|  |  |  |  |  |  |
| 21.9 | URI 类 | ................................... |  |  |  |  |  |  |  |  |  |  | 696 |  | 23.3.6 | InputEvent | | | |  | ................... | | | | | 728 |  |
|  |  |  |  |  |  |  |  |  |  |  |  | 类 | |  |  |  |  |  |
| 21.10 | cookie................................... | |  |  |  |  |  |  |  |  |  |  | 696 |  | 23.3.7 | ItemEvent 类 | | | | | .................... | | | | | 729 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 21.11 | TCP/IP 服务器套接字 | | | | | | | | | | | ........ | 696 |  | 23.3.8 |  | KeyEvent 类 | | | | .................... | | | | | 730 |  |
|  |  |  |  |  |  |  |  |  |
| 21.12 | 数据报 ................................. | |  |  |  |  |  |  |  |  |  |  | 697 |  | 23.3.9 | MouseEvent 类 | | | | | | | | ................ | | 730 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 21.12.1 | | DatagramSocket 类 ..... | | | | | | | | | | 697 |  | 23.3.10 |  | MouseWheelEvent 类 .... | | | | | | | | | 732 |  |
|  | 21.12.2 | | DatagramPacket 类 ..... | | | | | | | | | | 698 |  | 23.3.11 |  | TextEvent 类 .................. | | | | | | | | | 733 |  |
|  | 21.12.3 | | 数据报示例 ................ | | | | | | |  |  |  | 699 |  | 23.3.12 |  | WindowEvent 类 ........... | | | | | | | | | 733 |  |
| 第 22 章 | Applet 类 ............................. 701 | | | | | | | | | | | | | 23.4 | 事件源 | | ................................... | | | | | | | | | 734 |  |
|  |  |  |  |  |  |  |  |  |  |
| 23.5 | 事件监听器接口 | | | | | .................... | | | | | | 735 |  |
| 22.1 |  |  |  |  |  |  |  |  | ................. | | | | 701 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| applet 的两种类型 | | | | | | | | |  |  |  |  | 23.5.1 |  | ActionListener 接口 ........ | | | | | | | | | 736 |  |
| 22.2 |  |  |  |  |  |  |  |  | ................. | | | | 702 |  |  |  |
| applet 的基础知识 | | | | | | | | |  |  |  |  | 23.5.2 |  | AdjustmentListener | | | | | | | | |  |  |
| 22.3 | applet 的架构 | | | | ........................ | | | | | | | | 704 |  |  |  |  |
|  |  | 接口 ................................. | | | | | | | | | | 736 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 22.4 | applet 的骨架 | | | | ........................ | | | | | | | | 705 |  |  |  |
|  | 23.5.3 |  | ComponentListener | | | | | | | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 22.4.1 | applet 的初始化和终止 ... | | | | | | | | | | | 706 |  |  |  |  |
|  |  |  | 接口 ................................. | | | | | | | | | | 736 |  |
|  | 22.4.2 | 重写 update()方法 .......... | | | | | | | | | | | 707 |  |  |  |
|  |  | 23.5.4 |  | ContainerListener 接口 .... | | | | | | | | | 736 |  |
| 22.5 | 简单的 applet | | | | 显示方法 | | | | | | | ........ | 707 |  |  |  |
|  | 23.5.5 | FocusListener 接口 .......... | | | | | | | | | | 736 |  |
|  |  |  |
| 22.6 | 请求重画 | | | ............................... | | | | | | | | | 709 |  |  |
|  | 23.5.6 | ItemListener 接口 ............ | | | | | | | | | | 736 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 22.7 | 使用状态栏窗口 | | | | | | | ................... | | | | | 712 |  |  |
|  | 23.5.7 |  | KeyListener 接口 ............. | | | | | | | | | 737 |  |
|  |  |  |  |  |  |  |  |
| 22.8 | HTML APPLET 标签 | | | | | | | | | ............ | | | 713 |  |  |  |
|  | 23.5.8 | MouseListener 接口 ........ | | | | | | | | | | 737 |  |
|  |  |  |  |  |
| 22.9 | 向 applet | |  |  |  |  |  |  | ................ | | | | 714 |  |  |
| 传递参数 | | | | | | |  |  |  |  | 23.5.9 | MouseMotionListener | | | | | | | | | |  |  |
| 22.10 | getDocumentBase()和 | | | | | | | | | | |  |  |  |  |  |
|  |  |  |  | 接口 ................................. | | | | | | | | | | 737 |  |
|  | getCodeBase() | | | | | ...................... | | | | | | | 718 |  |  |  |
|  |  | 23.5.10 |  | MouseWheelListener | | | | | | | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| 22.11 | AppletContext 接口和 | | | | | | | | | | |  |  |  |  |  |  |
|  |  |  |  |  | 接口 ............................... | | | | | | | | | 737 |  |
|  | showDocument()方法 | | | | | | | | | | .......... | | 718 |  |  |  |  |
|  |  | 23.5.11 |  | TextListener 接口 .......... | | | | | | | | | 738 |  |
|  |  |  |  |  |  |
| 22.12 | AudioClip 接口 | | | | | | | ................... | | | | | 720 |  |  |  |
|  | 23.5.12 |  | WindowFocusListener | | | | | | | | |  |  |
|  |  |  |  |  |  |  |  |  |
| 22.13 | AppletStub 接口 | | | | | | | | .................. | | | | 720 |  |  |  |  |
|  |  |  | 接口 ............................... | | | | | | | | | 738 |  |
|  |  |  |  |  |  |  |  |
| 22.14 | 向控制台输出 | | | | | | ..................... | | | | | | 720 |  |  |  |  |
|  | 23.5.13 |  | WindowListener 接口 .... | | | | | | | | | 738 |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 第 23 章 | 事件处理 .............................. | |  |  |  |  |  |  |  |  |  | 721 | | 23.6 | 使用委托事件模型 | | | | | | | ................ | | | | 738 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 23.1 | 两种事件处理机制 ............... | | | | | | | | | | | | 722 |  | 23.6.1 |  | 处理鼠标事件 | | | | | | ................. | | | 739 |  |
|  |  |  |  |  |  |
| 23.2 | 委托事件模型 ....................... | | | | | | | | | | | | 722 |  | 23.6.2 |  | 处理键盘事件 | | | | | | ................. | | | 741 |  |
|  |  |  |  |  |  |
|  | 23.2.1 | 事件 ................................ | | | | | | | | | | | 722 | 23.7 | 适配器类 | | | | ............................... | | | | | | | 744 |  |
|  |  |  |  |  |  |  |  |  |
|  | 23.2.2 | 事件源 ............................ | | | | | | | | | | | 723 | 23.8 | 内部类 | | ................................... | | | | | | | | | 746 |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | 23.2.3 | 事件监听器 ..................... | | | | | | | | | | | 723 | 第 24 章 | AWT 介绍：使用窗口、 | | | | | | | | | | |  |  |
| 23.3 | 事件类 | ................................... | | | | | | | | | | | 724 |  |  |
|  | 图形和文本 .......................... 749 | | | | | | | | | | | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 23.3.1 | ActionEvent 类 ............... | | | | | | | | | | | 725 |  |  |
|  | 24.1 | AWT 类 | | | .................................. | | | | | | | | 750 |  |
|  |  | AdjustmentEvent 类 ........ | | | | | | | | | | | 726 |  |
|  | 23.3.2 |  |  |  |  |  |  |  |  |  |
|  | 24.2 |  |  |  |  | ........................ | | | | | | | 752 |  |
|  | 23.3.3 | ComponentEvent 类 ........ | | | | | | | | | | | 726 | 窗口基本元素 | | | | |  |  |  |  |  |  |  |
|  |  | 24.2.1 |  | Component 类 .................. | | | | | | | | | 752 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 目 | 录**XVII** |  |
|  | 24.2.2 | Container 类 | | | |  |  | 753 |  | 25.1.1 | 添加和移除控件 ............. | | | | | | | | 788 |  |
|  |  |  |  |  |
|  | 24.2.3 | Panel 类 ........................... | | |  |  |  | 753 |  | 25.1.2 | 响应控件 ......................... | | | | | | | | 788 |  |
|  | 24.2.4 | Window 类 ...................... | | |  |  |  | 753 |  | 25.1.3 | HeadlessException 异常 ... 788 | | | | | | | | |  |
|  | 24.2.5 | Frame 类 ......................... | | |  |  |  | 753 | 25.2 | 使用标签 ............................... | | | | | | | | | 789 |  |
|  | 24.2.6 | Canvas 类 ........................ | | |  |  |  | 753 | 25.3 |  |  | ........................ | | | | | | | 790 |  |
|  |  |  |  | 使用命令按钮 | | |  |  |  |  |  |  |  |
| 24.3 | 使用框架窗口 ....................... | | | |  |  |  | 753 | 25.4 |  |  | ............................ | | | | | | | 793 |  |
|  |  |  | 使用复选框 | | |  |  |  |  |  |  |  |
|  | 24.3.1 | 设置窗口的尺寸 ............. | | | |  |  | 754 | 25.5 |  |  | ........................ | | | | | | | 795 |  |
|  |  |  | 使用复选框组 | | |  |  |  |  |  |  |  |
|  | 24.3.2 | 隐藏和显示窗口 ............. | | | |  |  | 754 | 25.6 |  |  | ........................ | | | | | | | 796 |  |
|  |  |  | 使用下拉列表 | | |  |  |  |  |  |  |  |
|  | 24.3.3 | 设置窗口的标题 ............. | | | |  |  | 754 | 25.7 |  |  | ............................ | | | | | | | 798 |  |
|  |  |  | 使用列表框 | | |  |  |  |  |  |  |  |
|  | 24.3.4 | 关闭框架窗口 ................. | | | |  |  | 754 | 25.8 |  |  | ............................ | | | | | | | 801 |  |
|  |  |  | 管理滚动条 | | |  |  |  |  |  |  |  |
| 24.4 | 在 applet 中创建框架窗口 | | | | | .... | | 755 | 25.9 | 使用 TextField | | | ....................... | | | | | | 804 |  |
|  |  |  |  |  |  |  |  |  |
| 24.5 | 创建基于窗口的程序 ........... | | | | |  |  | 761 | 25.10 | 使用 TextArea | | | | ...................... | | | | | 807 |  |
|  |  |  |  |  |  |  |  |
| 24.6 | 在窗口中显示信息 ............... | | | | |  |  | 763 | 25.11 | 理解布局管理器 | | | | | | .................. | | | 808 |  |
|  |  |  |  |  |  |
| 24.7 | 使用图形 | | | ............................... | | | | 763 |  | 25.11.1 | | FlowLayout 布局 | | | | | | |  |  |
|  |  |  |  |  |  |  |
|  | 24.7.1 | 绘制直线......................... | | |  |  |  | 764 |  |  |  | 管理器 | | | ......................... | | | | 809 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  | 24.7.2 |  |  | ......................... | | |  | 765 |  | 25.11.2 | | BorderLayout 布局 | | | | | | |  |  |
|  | 绘制矩形 | | |  |  |  |  |  |  |
|  | 24.7.3 | 绘制椭圆和圆 ................. | | | |  |  | 765 |  |  |  | 管理器 ......................... | | | | | | | 811 |  |
|  | 24.7.4 | 绘制弧形......................... | | |  |  |  | 766 |  | 25.11.3 | | 使用 Insets | | | | | ................... | | 812 |  |
|  |  |  |  |  |  |  |  |
|  | 24.7.5 | 绘制多边形 ..................... | | | | |  | 767 |  | 25.11.4 | | GridLayout 布局 | | | | | | |  |  |
|  | 24.7.6 | 改变图形的大小 ............. | | | |  |  | 767 |  |  |  | 管理器 ......................... | | | | | | | 814 |  |
| 24.8 | 使用颜色 | | | ............................... | | | | 768 |  | 25.11.5 | | CardLayout 布局 | | | | | | |  |  |
|  |  |  |  |  |  |  |
|  | 24.8.1 | Color 类的方法............... | | | |  |  | 769 |  |  |  | 管理器 ......................... | | | | | | | 815 |  |
|  | 24.8.2 | 设置当前图形的颜色 ..... | | | | |  | 770 |  | 25.11.6 | | GridBagLayout 布局 | | | | | | |  |  |
|  | 24.8.3 | 一个演示颜色的 applet ... 770 | | | | | | |  |  |  | 管理器 ......................... | | | | | | | 818 |  |
| 24.9 | 设置绘图模式 ....................... | | | |  |  |  | 771 | 25.12 | 菜单栏和菜单 | | | | ...................... | | | | | 823 |  |
|  |  |  |  |  |  |  |  |  |
| 24.10 | 使用字体 ............................. | | | |  |  |  | 773 | 25.13 | 对话框 | | ................................. | | | | | | | 829 |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  | 24.10.1 | | 确定可用字体 ............ | | |  |  | 774 | 25.14 | FileDialog 类 ....................... | | | | | | | | | 834 |  |
|  | 24.10.2 | | 创建和选择字体 ........ | | | |  | 775 | 25.15 | 扩展 AWT 控件以处理 | | | | | | | | |  |  |
|  | 24.10.3 | | 获取字体信息 ............ | | |  |  | 777 |  | 事件 ..................................... | | | | | | | | | 835 |  |
| 24.11 | 使用 FontMetrics 管理文本 | | | | | |  |  |  | 25.15.1 | | 扩展按钮 ..................... | | | | | | | 836 |  |
|  | 输出 | ...................................... |  |  |  |  |  | 778 |  | 25.15.2 | | 扩展复选框 ................. | | | | | | | 837 |  |
|  | 24.11.1 | | 显示多行文本 ............ | | |  |  | 779 |  | 25.15.3 | | 扩展复选框组 ............. | | | | | | | 838 |  |
|  | 24.11.2 | | 居中显示文本 ............ | | |  |  | 781 |  | 25.15.4 | | 扩展下拉列表 ............. | | | | | | | 839 |  |
|  | 24.11.3 | | 对齐多行文本 ............ | | |  |  | 782 |  | 25.15.5 | | 扩展列表框 ................. | | | | | | | 840 |  |
| 第 25 章 | 使用 AWT 控件、布局 | | | | |  |  |  |  | 25.15.6 | | 扩展滚动条 ................. | | | | | | | 841 |  |
|  |  |  | 25.16 | 关于重载 paint()方法 | | | | | | | | .......... | 842 |  |
|  | 管理器和菜单 ...................... | | | | | | 787 | |  |
|  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 25.1 | 控件的基础知识 | | | | ................... | | | 788 |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **XVIII**Java 完全参考手册(第 8 版) | | | | | | | |  |  |  |  |  |  |  |
| 第 26 章 | 图像 ..................................... | | | | | | | | | | | 843 | |  |
| 26.1 | 文件格式 | | | ............................... | | | | | | | | | 844 |  |
|  |  |  |  |  |  |  |  |  |  |
| 26.2 | 图像基础：创建、加载与 | | | | | | | | | | |  |  |  |
|  | 显示 | ....................................... | | | | | | | | | | | 844 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 26.2.1 |  | 创建 Image 对象 ............. | | | | | | | | |  | 844 |  |
|  | 26.2.2 |  |  |  |  |  |  | ......................... | | | |  | 845 |  |
|  |  | 加载图像 | | | | | |  |  |  |  |  |
|  | 26.2.3 |  |  |  |  |  |  | ......................... | | | |  | 845 |  |
|  |  | 显示图像 | | | | | |  |  |  |  |  |
| 26.3 | ImageObserver 接口 | | | | | | | | | .............. | | | 846 |  |
|  |  |  |  |
| 26.4 | 双缓存 | | ................................... | | | | | | | | | | 848 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 26.5 | MediaTracker 类 | | | | | | | | .................... | | | | 850 |  |
|  |  |  |  |  |
| 26.6 | ImageProducer 接口 | | | | | | | | | .............. | | | 853 |  |
|  |  |  |  |
| 26.7 | ImageConsumer 接口 | | | | | | | | | | ............ | | 855 |  |
|  |  |  |
| 26.8 | ImageFilter 类 | | | | | | ........................ | | | | | | 857 |  |
|  |  |  |  |  |  |  |
|  | 26.8.1 |  | CropImageFilter 类 ......... | | | | | | | | |  | 858 |  |
|  | 26.8.2 |  | RGBImageFilter 类 ......... | | | | | | | | |  | 860 |  |
| 26.9 | 单元格动画 | | | | ........................... | | | | | | | | 870 |  |
|  |  |  |  |  |  |  |  |  |
| 26.10 | 其他图像类 | | | | | ......................... | | | | | | | 873 |  |
|  |  |  |  |  |  |  |  |
| 第 27 章 | 并发实用工具 ...................... | | | | | | | | | | | 875 | |  |
| 27.1 | 并发 API | | | 包 | ........................... | | | | | | | | 876 |  |
|  |  |  |  |  |  |  |  |  |
|  | 27.1.1 | java.util.concurrent 包 ..... | | | | | | | | | |  | 876 |  |
|  | 27.1.2 | java.util.concurrent.atomic | | | | | | | | | | | |  |
|  |  | 包 .................................... | | | | | | | | | |  | 877 |  |
|  | 27.1.3 | java.util.concurrent.locks | | | | | | | | | |  |  |  |
|  |  | 包 .................................... | | | | | | | | | |  | 877 |  |
| 27.2 | 使用同步对象 | | | | | | | ....................... | | | | | 877 |  |
|  |  |  |  |  |  |
|  | 27.2.1 | Semaphore 类 .................. | | | | | | | | | |  | 878 |  |
|  | 27.2.2 |  | CountDownLatch 类 ....... | | | | | | | | |  | 884 |  |
|  | 27.2.3 |  | CyclicBarrier 类 .............. | | | | | | | | |  | 885 |  |
|  | 27.2.4 | Exchanger 类................... | | | | | | | | | |  | 888 |  |
|  | 27.2.5 | Phaser 类 ......................... | | | | | | | | | |  | 890 |  |
| 27.3 | 使用执行器 | | | | ........................... | | | | | | | | 898 |  |
|  |  |  |  |  |  |  |  |  |

1. 一个简单的执行器

|  |  |  |  |
| --- | --- | --- | --- |
| 示例 | ............................ | 898 |  |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 27.3.2 使用 Callable 和 Future | |  |  |
| 接口 ................................ | | 900 |  |
| 27.4 TimeUnit 枚举 | ....................... | 903 |  |
|  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 27.5 | 并发集合 | | | ............................... | | 904 |  |
|  |  |  |
| 27.6 | 锁 | ........................................... | | | | 905 |  |
|  |  |  |  |  |
| 27.7 | 原子操作 | | | ............................... | | 908 |  |
|  |  |  |
| 27.8 | 通过 Fork/Join Framework | | | | |  |  |
|  | 进行并行编程 | | | | ........................ | 909 |  |
|  |  |  |
|  | 27.8.1 | | 主要的 Fork/Join 类 ........ | | | 909 |  |
|  | 27.8.2 | | 分而治之的策略 ............. | | | 912 |  |
|  | 27.8.3 | | 一个简单的 Fork/Join | | |  |  |
|  |  |  | 示例 ................................. | | | 913 |  |
|  | 27.8.4 | | 理解并行级别带来的 | | |  |  |
|  |  |  | 影响 ................................. | | | 915 |  |

1. 一个使用 RecursiveTask<V>

|  |  |
| --- | --- |
|  | 的例子 ............................. 918 |
| 27.8.6 | 异步执行任务 ................. 921 |
| 27.8.7 | 取消任务 ......................... 921 |

1. 确定任务的完成状态 ..... 921

|  |  |
| --- | --- |
| 27.8.9 | 重新启动任务 ................. 922 |
| 27.8.10 | 深入研究 ....................... 922 |

1. 关于 Fork/Join Framework

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 的一些提示 ................... | | | 923 |  |
| 27.9 | 并发实用工具与 Java 传统 | | | |  |  |
|  |  | ............................ | | | 924 |  |
|  | 方式的比较 | |  |  |  |
| 第 28 章 | 正则表达式和其他包 ........... 925 | | | | |  |
| 28.1 | 核心 Java API 包 | | | ................... | 926 |  |
|  |  |
| 28.2 | 正则表达式处理 | | .................... | | 927 |  |
|  |  |  |
|  | 28.2.1 | Pattern 类 ......................... | | | 928 |  |
|  | 28.2.2 | Matcher 类 ....................... | | | 928 |  |

1. 正则表达式的语法 ......... 929

28.2.4 演示模式匹配 ................. 929

1. 模式匹配的两个选项 ..... 935

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 28.2.6 | 探究正则表达式 ............. | 935 |  |
| 28.3 | 反射 | ....................................... | 936 |  |
|  |  |
| 28.4 |  | ........................ | 939 |  |
| 远程方法调用 | |  |
| 28.5 |  | ............................ | 943 |  |
| 文本格式化 | |  |
|  | 28.5.1 | DateFormat 类 ................. | 943 |  |
|  | 28.5.2 | SimpleDateFormat 类 ...... | 945 |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 目 |  | 录**XIX** |  |
| 第Ⅲ部分 | | 使用 Java 开发软件 | | | | | |  | 30.10.3 一个绘图示例 ............. | | | | | | | | | | | | 979 |  |
|  |  |
| 第 29 章 Java Bean ............................. | | | |  |  | 951 | | 第 31 章 | 探究 Swing .......................... 983 | | | | | | | | | | | | |  |
| 29.1 | Java Bean 是什么 .................. | | | | |  | 952 | 31.1 | JLabel 与 ImageIcon | | | | | | | | | | .............. | | 984 |  |
|  |  |  |  |
| 29.2 | Java Bean 的优势 .................. | | | | |  | 952 | 31.2 | JTextField | | | ............................... | | | | | | | | | 985 |  |
|  |  |  |  |  |  |  |  |  |  |  |
| 29.3 | 反省 | ....................................... |  |  |  |  | 952 | 31.3 | Swing 按钮 | | | | | ............................ | | | | | | | 987 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 29.3.1 | 属性的设计模式 ............. | | | |  | 952 |  | 31.3.1 | | JButton ............................. | | | | | | | | | | 988 |  |
|  | 29.3.2 | 事件的设计模式 ............. | | | |  | 954 |  | 31.3.2 | | JToggleButton .................. | | | | | | | | | | 990 |  |
|  | 29.3.3 | 方法与设计模式 ............. | | | |  | 954 |  | 31.3.3 | | 复选框 ............................. | | | | | | | | | | 992 |  |
|  | 29.3.4 | 使用 BeanInfo 接口 ........ | | | |  | 955 |  | 31.3.4 | | 单选按钮 ......................... | | | | | | | | | | 994 |  |
| 29.4 | 绑定属性与约束属性 ........... | | | | |  | 955 | 31.4 | JTabbedPane | | | | | | | .......................... | | | | | 996 |  |
|  |  |  |  |  |  |  |
| 29.5 | 持久性 ................................... | |  |  |  |  | 955 | 31.5 | JScrollPane | | | | ............................. | | | | | | | | 998 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 29.6 | 定制器 ................................... | |  |  |  |  | 956 | 31.6 | JList | ...................................... | | | | | | | | | | 1000 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 29.7 | Java Bean API ........................ | | | |  |  | 956 | 31.7 |  |  |  |  |  | .......................... | | | | | | 1003 | |  |
|  |  | JComboBox | | | | | |  |  |  |  |  |  |
|  | 29.7.1 | Introspector 类 ................ | | | |  | 958 | 31.8 | 树 ......................................... | | | | | | | | | | | 1006 | |  |
|  | 29.7.2 | PropertyDescriptor 类 ..... | | | | | 958 | 31.9 | JTable ................................... | | | | | | | | | | | 1009 | |  |
|  | 29.7.3 | EventSetDescriptor 类 .... | | | | | 958 | 31.10 | 继续研究 Swing ................ | | | | | | | | | | | 1011 | |  |
|  | 29.7.4 | MethodDescriptor 类 ...... | | | | | 958 | 第 32 章 | servlet ............................... 1013 | | | | | | | | | | | | |  |
| 29.8 | 一个 Bean 示例 | | | | ..................... | | 958 |  |
| 32.1 | 背景 | | ..................................... | | | | | | | | | 1013 | |  |
|  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 第 30 章 | Swing 简介 .......................... | | |  |  | 963 | | 32.2 | sevelet 的生命周期 ............. | | | | | | | | | | | 1014 | |  |
| 30.1 | Swing 的起源 ........................ | | | |  |  | 964 | 32.3 | Servlet 开发选项 | | | | | | | | ................. | | | 1015 | |  |
|  |  |  |  |  |  |
| 30.2 | Swing 以 AWT 为基础 ......... | | | | |  | 964 | 32.4 | 使用 Tomcat | | | | | | ......................... | | | | | 1015 | |  |
|  |  |  |  |  |  |  |
| 30.3 | 两个关键的 Swing 特性 ....... | | | | |  | 964 | 32.5 | 一个简单的 servlet | | | | | | | | | .............. | | 1017 | |  |
|  |  |  |  |
|  | 30.3.1 | Swing 组件是轻量级的 ... 964 | | | | | |  | 32.5.1 | | 创建和编译 servlet | | | | | | | | |  |  |  |
|  | 30.3.2 | Swing 支持可插入外观 ... 965 | | | | | |  |  |  | 源代码 .......................... | | | | | | | | | 1017 | |  |
| 30.4 | MVC 连接 ............................. | | |  |  |  | 965 |  | 32.5.2 | | 启动 Tomcat ................. | | | | | | | | | 1018 | |  |
| 30.5 | 组件与容器 | | | ........................... | | | 966 |  | 32.5.3 | | 启动 Web 浏览器并 | | | | | | | | |  |  |  |
|  |  |  |  |  |  |  |
|  | 30.5.1 | 组件 ................................ | |  |  |  | 966 |  |  |  | 请求 servlet ................... | | | | | | | | | 1018 | |  |
|  | 30.5.2 | 容器 ................................ | |  |  |  | 967 | 32.6 | servlet API ............................ | | | | | | | | | | | 1018 | |  |
|  | 30.5.3 | 顶级容器窗格 ................. | | | |  | 967 | 32.7 | javax.servlet 包 .................... | | | | | | | | | | | 1018 | |  |
| 30.6 | Swing 包 ................................ | |  |  |  |  | 967 |  | 32.7.1 | | Servlet 接口 .................. | | | | | | | | | 1019 | |  |
| 30.7 | 一个简单的 Swing | | | | |  |  |  | 32.7.2 | | ServletConfig 接口 ....... | | | | | | | | | 1020 | |  |
|  | 应用程序 ............................... | | |  |  |  | 968 |  | 32.7.3 | | ServletContext 接口 ..... | | | | | | | | | 1020 | |  |
| 30.8 | 事件处理 ............................... | | |  |  |  | 972 |  | 32.7.4 | | ServletRequest 接口 ..... | | | | | | | | | 1020 | |  |
| 30.9 | 创建 Swing applet ................. | | | | |  | 975 |  | 32.7.5 | | ServletResponse 接口 ... 1021 | | | | | | | | | | |  |
| 30.10 | 在 Swing 中绘图 ................. | | | | |  | 977 |  | 32.7.6 | | GenericServlet 类 ......... | | | | | | | | | 1022 | |  |
|  | 30.10.1 | | 绘图的基础知识 ........ | | |  | 977 |  | 32.7.7 | | ServletInputStream 类 ... 1022 | | | | | | | | | | |  |
|  | 30.10.2 | | 计算可绘制区域 ........ | | |  | 978 |  | 32.7.8 | | ServletOutputStream 类... 1022 | | | | | | | | | | |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **XX**Java 完全参考手册(第 8 版) | | |  |  |  |  |
|  | 32.7.9 | servlet 异常类 ............... | | | 1022 |  |
| 32.8 | 读取 servlet 参数 | | ................. | | 1022 |  |
|  |  |  |
| 32.9 | javax.servlet.http 包 | | | ............. | 1024 |  |
|  |  |
|  | 32.9.1 | HttpServletRequest | | |  |  |
|  |  | 接口 .............................. | | | 1025 |  |
|  | 32.9.2 | HttpServletResponse | | |  |  |
|  |  | 接口 .............................. | | | 1025 |  |
|  | 32.9.3 | HttpSession 接口 .......... | | | 1026 |  |

1. HttpSessionBindingListener

|  |  |
| --- | --- |
|  | 接口 .............................. 1027 |
| 32.9.5 | Cookie 类 ...................... 1027 |
| 32.9.6 | HttpServlet 类 ............... 1028 |
| 32.9.7 | HttpSessionEvent 类 ..... 1029 |
| 32.9.8 | HttpSessionBindingEvent |
|  | 类 .................................. 1030 |

1. 处理 HTTP 请求和响应 ... 1030

32.10.1 处理 HTTP GET

请求 .......................... 1030

1. 处理 HTTP POST

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 请求 .......................... | | | | 1032 |  |
| 32.11 | 使用 cookie | | | ........................ | | 1033 |  |
|  |  |  |
| 32.12 | 会话跟踪 | | ........................... | | | 1035 |  |
|  |  |  |  |
| 第Ⅳ部分 | | | Java 应用 | | |  |  |
| 第 33 章 | applet 和 servlet 在金融领域的 | | | | | |  |
|  | 应用 ................................... 1039 | | | | | |  |
| 33.1 | 计算贷款的还款额 | | | | ............. | 1040 |  |
|  |  |
|  | 33.1.1 | RegPay 中声明的变量 ... 1044 | | | | |  |
|  | 33.1.2 | init()方法....................... | | | | 1044 |  |
|  | 33.1.3 | makeGUI()方法 ............ | | | | 1044 |  |
|  | 33.1.4 |  |  |  | .... | 1047 |  |
|  | actionPerformed()方法 | | | |  |

33.1.5 compute()方法 .............. 1048

1. 计算投资的未来收益 ......... 1048
2. 计算为达到未来收益所需

|  |  |  |  |
| --- | --- | --- | --- |
| 要的初始投资额 | ................. | 1052 |  |
|  |  |

1. 计算为拿到期望的养老金所需要的初始投资额 ......... 1056
2. 计算给定投资的最大收益 ... 1060

|  |  |  |  |
| --- | --- | --- | --- |
| 33.6 计算贷款余额 | ..................... | 1064 |  |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 33.7 创建应用于金融领域的 | |  |  |
| servlet | ................................... | 1068 |  |
|  |  |

1. 将 RegPay applet 转换为

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | servlet............................ | | | | 1069 |  |
|  | 33.7.2 RegPayS servlet ............ | | | | 1069 |  |
| 33.8 | 一些尝试 | ............................. | | | 1072 |  |
|  |  |  |  |
| 第 34 章 | 使用 Java 创建下载管理器 ... 1073 | | | | |  |
| 34.1 | 理解 Internet 下载 | | | ............... | 1074 |  |
|  |  |
| 34.2 | 下载管理器概述 | | .................. | | 1074 |  |
|  |  |  |
| 34.3 |  | ....................... | | | 1075 |  |
| DownLoad 类 | |  |  |  |

1. DownLoad 类的变量 ... 1079
2. DownLoad 类的

|  |  |
| --- | --- |
|  | 构造函数 ...................... 1079 |
| 34.3.3 | download()方法 ............ 1079 |
| 34.3.4 | run()方法 ...................... 1079 |

1. stateChanged()方法 ...... 1083
2. 动作和访问器方法 ...... 1083

|  |  |  |  |
| --- | --- | --- | --- |
| 34.4 ProgressRenderer 类 | ............ | 1083 |  |
|  |  |

1. DownloadsTableModel 类 ... 1084
   1. addDownload()方法 ..... 1086
   2. clearDownload()方法 ... 1087
   3. getColumnClass()方法 ... 1087
   4. getValueAt()方法 .......... 1087

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 34.5.5 | update()方法 ................. | | 1088 |  |
| 34.6 DownloadManager 类 | | .......... | 1088 |  |
|  |  |
| 34.6.1 | DownloadManager 类的 | |  |  |
|  | 变量 .............................. | | 1094 |  |

1. DownloadManager 类的

构造函数 1095

34.6.3 verifyUrl()方法 1095

34.6.4 tableSelectionChanged()

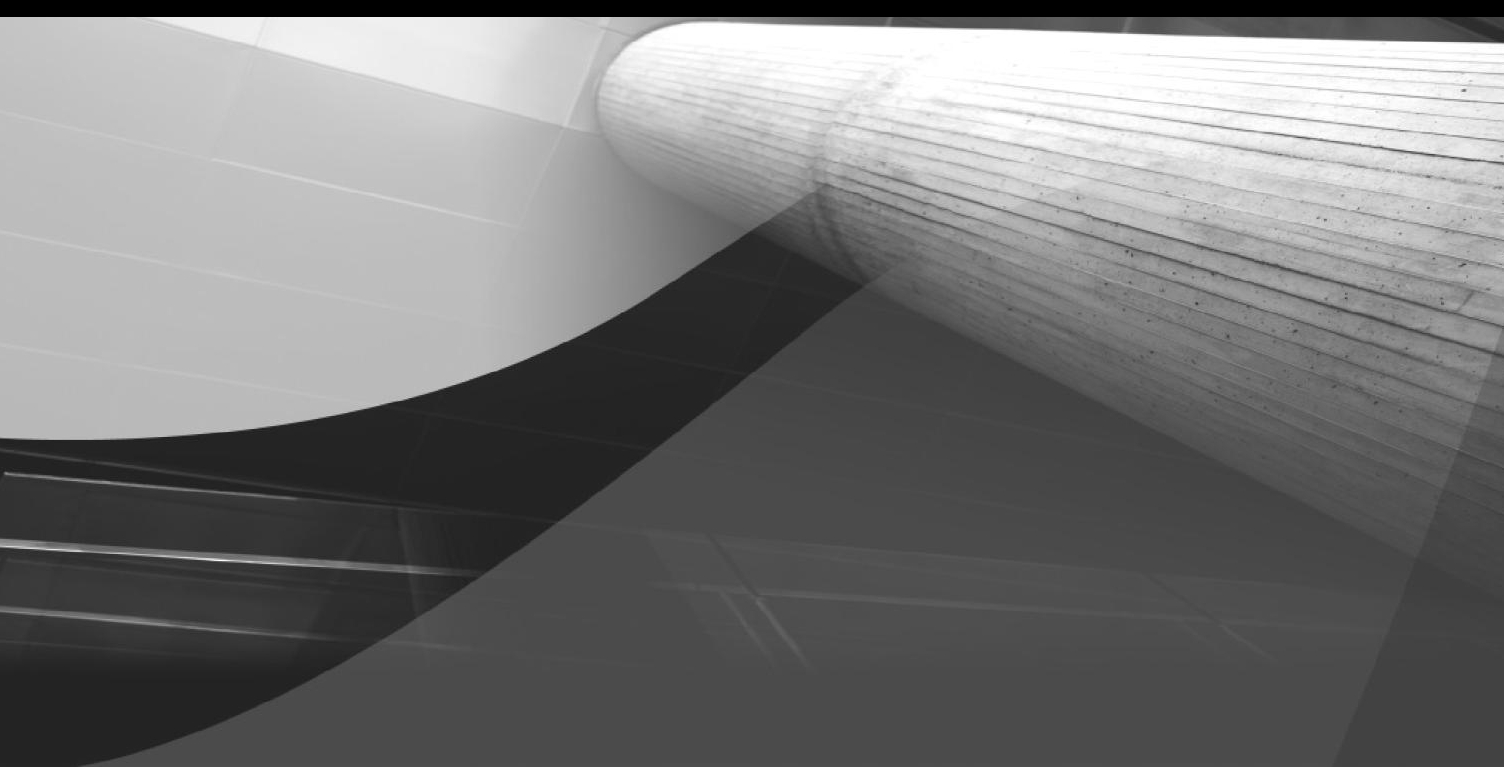
方法 1096

34.6.5 updateButtons()方法 1096

34.6.6 处理动作事件 1097

1. 编译和运行下载管理器 ...... 1098
2. 进一步完善下载管理器 ...... 1098

附录 使用 Java 的文档注释 ............. 1099



**第 1 章**

Java 的历史和演变

为了完全地理解 Java，您必须理解创建它的背后原因、促使其成型的动力以及它所继承的思想。与以前所有成功的计算机语言一样，Java 是一个混合物，它是由大量继承自其他编程语言的特性中的最优元素，以及 Java 为完成自身特殊使命所必需的创新性概念联合组成的。尽管本书其他各章将描述实际使用 Java 的相关内容，包括它的语法、关键库以及应用程序，但是本章将介绍 Java 出现的背景、创建 Java 的原因、是什么原因使 Java 如此重要，以及多年来 Java 的演变过程。

尽管 Java 已经变得与 Internet 的在线环境密不可分，但是 Java 首先并且首要的仍然是一种语言，记住这一点很重要。计算机语言的创新与发展取决于以下两个基本原因：

● 适应环境和用途的变化

**4** 第Ⅰ部分 Java 语言

● 实现编程艺术的完善与提高

在后面将会看到，Java 的发展就是由这两个因素驱动的，而且这两个因素的驱动程度几乎相同。

**1.1** **Java 的家世**

Java 与 C++相关，C++是 C 的直接后代。Java 的大量特性就是从这两种语言继承的。Java

从 C 继承了它的语法。Java 的许多面向对象特性则受 C++的影响。实际上，Java 的一些特性来自它的前辈，或受其影响。而且，Java 的创建基于过去几十年来计算机编程语言的改良和发展过程。由于这些原因，本节将回顾促使 Java 产生的一系列事件和动力。您将会看到，语言设计的每次革新，都是为了解决之前语言不能解决的基本问题，Java 也不例外。

1.1.1 现代编程语言的诞生：C 语言

C 语言的诞生震惊了计算机界。不应当低估它的影响，因为它从根本上改变了编程的方式和思路。C 语言的产生是人们对结构化、高效率、(在创建系统程序时能够取代汇编代码的)

高级语言需求的直接结果。正如您可能知道的，当设计一种计算机语言时，经常需要进行取舍，例如权衡下面这些因素：

* 易用性与功能
* 安全性与效率
* 稳定性与可扩展性

在 C 语言以前，程序员通常需要在品质不同的各种计算机语言之间进行选择。例如，尽管可以使用 FORTRAN 为科学计算应用程序编写出相当高效的程序，但是对于编写系统代码它不是很好。再比如，尽管 BASIC 易于学习，但它的功能不是很强大，并且由于缺少结构化设计，很让人怀疑是否可以将其应用于大型程序。汇编语言可以生成非常高效的代码，但是它不易于学习，使用效率低。而且，调试汇编代码可能相当困难。

另外一个复杂的问题是，早期的计算机语言，例如 BASIC、COBOL 以及 FORTRAIN，

没有遵循结构化设计原则。反而，它们依赖于 GOTO 作为程序控制的主要手段。因此，使用这些语言更容易编写出“意大利面条式的代码”——大量混乱的跳转语句和条件分支语句，使程序实际上很难理解。而类似 Pascal 的语言虽然是结构化的，但是它们不是针对高效率而设计的，并且没有提供使它们能够应用于大范围编程领域所需要的特性(特别是，在确定标准

Pascal 语言时，实际上并没有考虑将它用于系统级代码)。

因此，在 C 语言出现以前，没有哪种语言能够解决这些矛盾。但是对这样一种语言的需要是迫切的。到了 20 世纪 70 年代早期，计算机革命开始出现，并且对软件的需求快速增长，超出了程序员的能力。为了尝试创建出一种更好的计算机语言，学术界为此付出了大量的努力。但是，促使 C 语言诞生的第二个因素，也许是最重要的因素正在出现。计算机硬件最终变得非常普遍，达到了发生变化的临界状态。计算机不再被锁起来，程序员第一次可以真正地随意使用他们的计算机，从而可以随意地进行尝试。并且程序员还可以开始创建他们自己的工具。在 C 语言诞生前夕，计算机语言向前飞跃发展的舞台已经具备了。

C 语言是由 Dennis Ritchie 在运行 UNIX 操作系统的 DEC PDP-11 机器上发明并首次实现

第1章 Java 的历史和演变 **5**

的，它是老式 BCPL 语言不断发展的结果，BCPL 语言是由 Martin Richards 开发的。BCPL

语言对 Ken Thompson 发明的 B 语言产生了影响，B 语言导致了在 20 世纪 70 年代对 C 语言的开发。多年来，由 UNIX 操作系统提供的标准成为 C 语言事实上的标准，并且在 Brian

Kernighan 和 Dennis Ritchie 编写的 *The C Programming Language*(Prentice-Hall，1978)一书中得到了描述。1989 年 12 月，当美国国家标准学会(American National Standards Institute，ANSI)

制定的 C 语言标准被采纳后，C 语言被正式标准化。

C 语言的诞生被许多人认为是现代计算机语言时代开始的标志，它成功地综合了早期计算机语言曾经非常麻烦的矛盾特性，从而使 C 语言成为功能强大、高效率、结构化的语言，并且相对容易学习。C 语言还具有另外一个几乎是在无形中产生的特性：它是程序员的语言。在 C 语言诞生之前，计算机语言通常要么是作为学术实验而设计的，要么是由官方委员会设计的。而 C 语言不同，它是由真正从事编程工作的程序员设计、实现和开发的，反映了程序员进行实际编程工作的方法。C 语言的特性经过实际使用该语言的人们不断提炼、测试、思考、再思考，成为广大程序员最喜欢使用的语言。确实，C 语言迅速吸引了许多狂热的追随者。于是，C 语言被程序员广泛采用并被迅速接受。总之，C 语言是由程序员设计并由他们使用的一种语言。正如即将看到的，Java 继承了这一传统。

1. C++：下一个阶段
   * 20 世纪 70 年代晚期到 80 年代早期，C 语言成为主要的计算机编程语言，并且在今天仍然被广泛使用。既然 C 语言是一种成功并且有用的语言，您可能会好奇为什么还需要其他语言呢。答案是复杂性。纵观编程的历史，正是程序复杂性的不断增加驱动了管理复杂性的更好方式的需要。C++是对这一需求的响应。为了更好地理解为何管理程序的复杂性成为创建 C++的基础，我们来分析下面的内容。

自从发明计算机以来，编程方式发生了很大的变化。例如，计算机刚出现时，编程是通过面板用手工打孔的方法输入二进制机器指令实现的。对于那些只有几百行指令的程序，这种方法可以工作。随着程序的增长，引入了汇编语言，通过使用机器指令的符号化表示，程序员可以编写更大、更复杂的程序。随着程序的不断增长，出现了高级语言，为程序员提供了更多用于处理复杂性的工具。

当然，第一种广泛使用的高级语言是 FORTRAN。虽然 FORTRAN 迈出了令人印象深刻的第一步，但是它很难开发出条理清晰且易于理解的程序。20 世纪 60 年代诞生了结构化编程(structured programming)。这种编程方法被 C 语言这类语言采用。通过使用结构化编程语

言，程序员第一次能够比较容易地编写出相对复杂的程序。但是，即使是使用结构化编程方法，一旦项目达到一定的规模，它的复杂性就会超出程序能够管理的范围。到了 20 世纪 80

年代早期，许多项目超出了结构化方法的极限。为了解决这一问题，发明了一种新的编程方法，称为面向对象编程(Object-Oriented Programming，OOP)。面向对象编程将在本书后面详细讨论，但是在此先给出它的简短定义：OOP 是一种编程方法论，通过使用继承、封装和多态来帮助组织复杂的程序。

通过分析可以看出，尽管 C 语言是世界上最伟大的编程语言之一，但是它处理复杂性的能力也是有一定限度的。一旦程序规模超过特定的临界点，就会变得非常复杂以至于难以从整体上进行把握。虽然根据程序的自身特征以及程序员的不同，发生这种情况的准确界限会

1. 第Ⅰ部分 Java 语言

有所不同，但总是存在这样一个门槛，一旦超过这个门槛，程序就变得难以管理。C++添加了能够突破这一界限的特征，允许程序员理解并管理更大的程序。

C++语言是由 Bjarne Stroustrup 于 1979 年发明的，当时他在位于美国新泽西州 Murray Hill

的 Bel1 实验室工作。Stroustrup 最初将这种新语言称为“带类的 C”。但是，在 1983 年他将名称修改为 C++。C++通过添加面向对象特征对 C 语言进行了扩展。因为 C++构建于 C 语言的基础之上，所以它包含了 C 语言的全部特征、特性以及优点。这是 C++作为一种语言能够成功的关键原因。发明 C++语言不是试图创建一种全新的编程语言，相反，它是对已经取得极大成功的 C 语言的改进。

1.1.3 Java 出现的时机已经成熟

到了 20 世纪 80 年代末 90 年代初，使用面向对象编程的 C++语言占据了主导地位。确实，程序员好像一度找到了完美的语言。因为 C++既支持面向对象编程模式，又具有 C 语言的高效率以及风格优点，它确实是一种可以用于创建各种程序的语言。然而，就像过去一样，推动计算机语言向前演变的力量又一次在酝酿。在短短的几年中，万维网(World Wide Web)

和 Internet 达到了临界规模。这一事件又将会促成编程的另一场革命。

**1.2** **Java 的诞生**

Java 是由 James Gosling、Patrick Naughton、Chris Warth、Ed Frank 和 Mike Sheridan 于 1991 年在 Sun 公司构想出来的。开发第一个版本花费了 18 个月。这种语言最初称为 Oak，在 1995 年被命名为 Java。从 1992 年秋 Oak 最初实现到 1995 年春 Java 语言的公开发布，许多人对 Java 的设计和改进作出了贡献。Bill Joy、Arthur van Hoff、Jonathan Payne、Frank Yellin

和 Tim Lindholm 是主要贡献者，他们的奉献使 Java 的最初原型逐渐成熟。

有些让人惊奇的是，Java 的最初推动力不是 Internet！相反，主要动机是对平台独立(即

体系结构中立)语言的需要，这种语言可用于开发能够嵌入到各种消费类电子设备(例如微波炉、遥控器)的软件。您可能已经猜到了，许多不同类型的 CPU 被用作控制器。使用 C 和 C++

语言(以及大部分其他语言)的麻烦是，它们被设计为针对特定的目标进行编译。尽管能够为各种类型的 CPU 编译 C++程序，但是这需要一个完整的以该 CPU 为目标的 C++编译器。问题是创建编译器很耗费时间，所以需要一种更容易并且更经济的解决方案。在寻找这样一种方案的尝试过程中，Gosling 和其他人一起开始开发一种可移植的、平台独立的语言，可以使用这种语言生成在不同环境下运行于各种 CPU 之上的代码。他们的努力最终导致了 Java 的

出现。

在 Java 的细节被开发出来的同时，第二个并且最终也更加重要的因素出现了，它在 Java

的未来中扮演了关键的角色。第二个动力当然是万维网。假如 Web 的形成和 Java 的出现不在同一时间，那么 Java 虽然仍会有用，但可能只是一种用于为消费类电子产品编写代码的没有名气的语言。然而，随着万维网的出现，Java 被推到计算机语言设计的最前沿，因为 Web

也需要可移植的程序。

大部分程序员在职业生涯的早期就知道，可移植程序既让人向往又让人逃避。尽管人们对创建高效、可移植(平台独立的)程序的探索，几乎和编程自身的历史一样久远，但它总是

第1章 Java 的历史和演变 **7**

让位于其他更为紧迫的问题。此外，因为在那时计算机界已经被 Intel、Macintosh 和 UNIX

这三个竞争阵营垄断，大多数程序员都在其中的某个领域内工作，所以对可移植性编码的迫切需求降低了。但是，随着 Internet 和 Web 的出现，古老的可移植性问题又出现了。毕竟，

Internet 是由各种各样的、分布式的系统构成的，这些系统使用各种类型的计算机、操作系统和 CPU。尽管许多类型的平台都依附于 Internet，但是用户仍然希望他们能够运行相同的程序。曾经是一个令人烦恼、但是优先级较低的问题，已经变成了必须解决的问题。

在为嵌入式控制器编写代码时经常遇到的可移植性问题，在尝试为 Internet 编写代码的过程中也出现了。到了 1993 年，这个问题对于 Java 设计小组的成员而言已经变得很明显了。实际上，最初针对解决小范围问题而设计的 Java，也可以应用于更大范围的 Internet。这一认识导致 Java 的关注点由消费类电子产品转移到了 Internet 编程。因此，虽然对体系结构中立的编程语言的需求提供了最初的思想火花，但最终是 Internet 成就了 Java 的成功。

正如在前面提到的，Java 从 C 和 C++继承了许多特性，这是有意而为之。Java 设计人员清楚，使用与 C 语言类似的语法以及模仿 C++的面向对象特性，可以使 Java 语言对于众多经验丰富的 C/C++程序员更具吸引力。除了表面类似外，Java 还借鉴了帮助 C 和 C++成功的其他一些特性。首先，Java 的设计、测试和不断改进是由真正从事编程工作的人员完成的。它是扎根于设计人员的需要和经验的一种语言。因此 Java 是程序员的语言。其次，Java 结构紧凑并且逻辑上协调一致。最后，除了 Internet 强加的那些约束外，Java 为程序员提供了完全的控制权。如果程序编写的好，程序本身就能反映出来。如果程序编写的不好，程序本身也能反映出来。因为存在这样的区别，所以 Java 不是一种用于培训的语言，而是针对专业程序员的语言。

因为 Java 与 C++之间的相似性，可能会简单地将 Java 看作“Internet 版的 C++”。但是，如果这么认为，将会是很大的错误。Java 无论是在实践上还是在理论上都与 C++有着很大的区别。虽然 Java 深受 C++的影响，但它不是 C++的增强版。例如，Java 与 C++既不向上兼容，也不向下兼容。当然，与 C++之间的相似性还是很明显的。并且，如果您是一位 C++程序员的话，会感觉 Java 很熟悉。另外一点：设计 Java 的目的不是取代 C++。Java 是针对解决特定的一系列问题而设计的。Java 和 C++将会长期共存。

正如在本章开头提到的，计算机语言的发展取决于两个因素：适应环境的变化以及实现编程艺术的提高。促使 Java 发展的环境变化是对平台独立程序的需求，Internet 上的分布式系统天生就需要平台独立的程序。同时，Java 也体现了编程方式的变化。例如，Java 增强并改进了 C++使用的面向对象编程，增加了对多线程的支持，提供了简化 Internet 访问的库。总之，并不是 Java 的某个单一特征，而是整体上作为一种语言，使它如此非凡。Java 是对新出现的高度分布计算领域需求的完美响应。Java 对于 Internet 编程的意义，就如同 C 语言对系统编程一样：它们都是改变世界的革命性力量。

C#连接

在计算机语言开发领域，人们会继续感受到 Java 的影响和力量。许多创新性的特征、结构以及概念，已经成为所有新语言的基准组成部分。Java 是如此成功以至于不可忽视。

Java 影响力的最重要例子可能是 C#。C#是由 Microsoft 创建的用于支持.NET Framework

的语言，C#与 Java 密切相关。例如，两者共享相同的语法，都支持分布式编程，并且都利用

1. 第Ⅰ部分 Java 语言

相同的对象模型。当然，Java 和 C#之间也有一些区别，但是总的看来这两种语言很相似。从 Java 到 C#的这种“异花授粉”，正是对 Java 影响力最强有力的证明。

**1.3** **Java 改变 Internet 的方式**

Internet 将 Java 推到了编程的最前沿，反过来，Java 也对 Internet 产生了深远的影响。一般来说，除了简化 Web 编程以外，Java 还创立了一种新的网络程序类型，称为 applet，这种程序类型改变了在线世界考虑内容的方式。Java 还解决了一些与 Internet 相关的棘手问题：可移植性和安全性。下面进一步分析这些内容。

1.3.1 Java applet

Java applet 是一种特殊类型的 Java 程序，是为了能够在 Internet 上传送而设计的，可以在兼容 Java 的 Web 浏览器中自动运行。此外，applet 可以根据需要自动下载，不需要用户的进一步交互。如果用户单击包含 applet 的链接，就会自动下载 applet，并在浏览器中运行。 applet 一般是小的程序，它们通常用于显示服务器提供的数据、处理用户输入或者提供在本地执行而不是在服务器上执行的简单功能，例如贷款计算器。本质上，applet 使得可以将某些功能从服务器移到客户端。

applet 的创建改变了 Internet 编程，因为它扩展了可以在网络空间(cyberspace)中自由流动

的对象的范畴。一般而言，在服务器和客户端之间传输的对象有两大类：被动的信息和动态、主动的程序。例如，当阅读电子邮件时，是在查看被动的数据。甚至当下载一个程序时，在执行该程序之前，它的代码也只是被动的数据。与之相对应，applet 是动态的、自我执行的程序。虽然这类程序是客户端计算机上的活动代码，然而它们却是由服务器初始化的。

网络化程序虽然在动态性方面符合人们的愿望，但是它们在安全性和可移植性方面带来了严重问题。显然，必须防止在客户端计算机上自动下载和执行的程序执行破坏性的工作，并且它们还必须能够运行于各种不同的环境以及不同的操作系统中。正如在后面即将看到的，

Java 以一种高效且优美的方式解决了这些问题。下面让我们进一步分析 Java 解决这些问题的

原理。

1.3.2 安全性

您可能知道，每次下载一个“正常的”程序时，都具有一定的风险，因为下载的代码可能包含病毒、特洛伊木马或其他有害代码。问题的核心是恶意代码能够导致损害，因为它已经获得对系统资源的未授权访问。例如，病毒程序可以通过搜索计算机本地文件系统的内容来收集私人信息，如信用卡号码、银行账户余额和口令等。为了使 Java 能够在客户端计算机上安全地下载并执行 applet，需要阻止 applet 进行这样的攻击。

Java 通过将 applet 限制在 Java 执行环境中，并且不允许访问计算机的其他部分来实现这种保护(稍后会看到实现这种保护的原理)。下载 applet 并能确保不会造成危害，并且不会破坏安全性的能力，被认为是 Java 最重要的一个创新。

第1章 Java 的历史和演变 **9**

1.3.3 可移植性

可移植性是 Internet 的一个主要方面，因为有许多不同类型的计算机和操作系统被连接到 Internet。要使 Java 程序真正能够运行于连接到 Internet 的任何计算机上，需要具有某种方法使 Java 程序能够在不同的系统中执行。例如，就 applet 而言，同一个 applet 必须能够在连接到 Internet 上的大量不同类型的 CPU、操作系统和浏览器环境中下载并执行。为不同的计算机开发不同版本的 applet 是不切实际的。相同的代码必须能够在所有计算机上工作。所以，需要一些能够生成可移植的可执行代码的方法。很快将会看到，有助于提高安全性的相同机制也可以用于帮助实现可移植性。

**1.4** **Java 的魔力：字节码**

允许 Java 解决刚才描述的安全性和可移植性问题的关键是，Java 编译器的输出不是可执行代码，而是字节码(bytecode)。字节码是高度优化的指令集合，这些指令由 Java 运行时系

统执行，Java 运行时系统也称为 Java 虚拟机(Java Virtual Machine，JVM)。本质上，原始的

JVM 被设计为字节码解释器(interpreter for bytecode)。这可能有点让人吃惊，因为出于性能方面的考虑，许多现代语言被设计为将源代码编译成可执行代码。然而，Java 程序是由 JVM 执

行的这一事实，有助于解决与基于 Web 的程序相关的主要问题。下面分析其中的原因。

将 Java 程序翻译成字节码，可以使其更容易地在广大范围的可变环境中运行，因为只需要针对每种平台实现 Java 虚拟机就可以了。对于给定的系统只要存在运行时包，所有 Java

程序就可以在该系统上运行。请记住，尽管对于不同的平台，Java 虚拟机的细节可能有所不同，但是它们都能理解相同的 Java 字节码。如果 Java 程序被编译成本机代码，就必须为相同的程序针对连接到 Internet 的不同类型的 CPU 提供不同版本。因此，通过 JVM 执行字节码是创建真正可移植程序最容易的方法。

Java 程序由 JVM 执行的这一事实，还有助于提供安全性。因为 JVM 是可控的，所以能够包含程序并防止它在系统之外产生负效应。在后面将会看到，安全性也是通过 Java 语言具有的特定限制得到增强的。

一般而言，如果将程序编译成中间形式，然后由虚拟机解释执行，相对于直接编译成可执行代码，执行速度要慢一些。然而对于 Java 而言，这两种方式之间的区别不是特别大。因为字节码已经被高度优化，使用字节码能够使 JVM 执行程序的速度，比您可能认为的速度快得多。

尽管 Java 被设计成一种解释语言，但是为了提高性能，完全可以将字节码编译为本机代码。为此，在 Java 最初发布不久就引入了 HotSpot 技术。HotSpot 为字节码提供了即时

(Just-In-Time，JIT)编译器。如果 JVM 包含 JIT 编译器，就可以根据要求一部分一部分地将选择的字节码实时编译为可执行代码。不是将整个 Java 程序一次性地全部编译为可执行代码，理解这一点很重要，因为 Java 执行各种运行时检查，而这些检查只有在运行时才执行。事实是，JIT 编译器在执行期间根据需要编译代码。此外，不是编译所有字节码序列——而是只编译那些能从编译中受益的字节码，剩余代码仍然只是进行解释。尽管如此，这种即时编译方法仍然可以显著提高性能。当对字节码应用动态编译时，仍然可以获得可移植性和安全性，

1. 第Ⅰ部分 Java 语言

因为 JVM 仍然控制着执行环境。

**1.5** **servlet：服务器端的 Java**

尽管 applet 很有用，但它只是客户端/服务器这个整体的一半。Java 对于服务器端也是很有用的，在 Java 最初发布不久这个问题就变得很明显了，于是出现了 servlet。servlet 是在服务器上执行的小程序。就像 applet 动态扩展了 Web 浏览器的功能一样，servlet 动态扩展了 Web 服务器的功能。因此，随着 servlet 的出现，Java 横跨了客户端/服务器连接的两端。

servlet 用于动态地创建发送到客户端的内容。例如，在线商店可以使用 servlet 在数据库中查找某件商品的价格，然后使用价格信息动态生成发送到浏览器的 Web 页面。尽管通过

CGI(Common Gateway Interface，公共网关接口)这类机制也可以获取动态生成的内容，但是

servlet 提供了一些优点，包括性能的提高。

因为 servlet(以及所有 Java 程序)被编译成字节码，并且由 JVM 执行，所以它们具有高度的可移植性。因此，相同的 servlet 可以用于各种不同的服务器环境中。唯一的要求是：服务

器支持 JVM 和 servlet 容器。

**1.6** **Java 的关键特性**

如果不介绍 Java 的关键特性，对 Java 历史的讨论就不是完整的。尽管促使 Java 必然诞生的基本动力是可移植性和安全性，但是在 Java 语言最终成型的过程中，其他因素也扮演了重要角色。Java 团队对设计 Java 时的关键考虑因素进行了总结，下面的关键特性列表包含了这些考虑因素：

* 简单性
* 安全性
* 可移植性
* 面向对象
* 健壮性
* 多线程
* 体系结构中立
* 解释执行
* 高性能
* 分布式
* 动态性在这些关键特性中，安全性和可移植性已经在前面介绍过了，下面分析其他各个特性的

含义。

1.6.1 简单性

Java 的设计目标之一是让专业程序员能够高效地学习和使用。如果具有一定的编程经验，就会发现掌握 Java 并不难。如果您已经理解了面向对象编程的基本概念，学习 Java 会

第1章 Java 的历史和演变 **11**

更加容易。最好的情况是，您是一位有经验的 C++程序员，只需要非常少的努力就可以迁移到 Java。因为 Java 继承了 C/C++的语法以及许多面向对象特性，大部分程序员学习 Java 都

不困难。

1.6.2 面向对象

尽管受到前辈的影响，但是 Java 没有被设计成兼容其他语言的源代码，这使得 Java 团

队可以自由地从头开始设计。这样设计的结果是：对象采用清晰、可用、实用的方法。通过大量借鉴过去几十年中的诸多对象软件环境，Java 设法在纯进化论者的“任何事物都是对象”模式和实用主义者的“够用就好”模式之间找到了平衡。Java 中的对象模型既简单又易于扩展，而基本类型(例如整型)仍然是高性能的非对象类型。

1.6.3 健壮性

Web 的多平台环境对程序有特别的要求，因为程序必须在各种系统中可靠地执行。因此，在设计 Java 时，使其具备创建健壮程序的能力被提到了高优先级的地位。为了获得可靠性，

Java 在一些关键领域进行了限制，从而迫使程序员在程序开发中及早地发现错误。同时，使程序员不必再担心会引起编程错误的许多最常见的问题。因为 Java 是强类型化的语言，它在编译时检查代码。当然不管怎样，在运行时也检查代码。许多难以跟踪的 bug，在运行时通常难以再现，这种情况在 Java 中几乎不可能产生。因为使编写好的程序在不同的运行条件下，以可预见的方式运行是 Java 的关键特性之一。

为了更好地理解 Java 是多么健壮，分析程序失败的两个主要原因：内存管理错误和未处理的异常(即运行时错误)。在传统的编程环境中，内存管理是一件困难、乏味的工作。例如，在 C/C++中，程序员必须手动分配和释放所有动态内存。有时这会导致问题，因为程序员可能会忘记释放以前分配的内存，或者更糟糕的是，试图释放程序其他部分仍然在使用的内存。

Java 通过为您管理内存的分配和释放，可以从根本上消除这些问题(事实上，释放内存完全是自动的，因为 Java 为不再使用的对象提供了垃圾回收功能)。传统环境中的异常情况通常是由“除 0”或“没有找到文件”这类错误引起的，并且必须使用既笨拙又难以理解的结构对它们进行管理。Java 通过提供面向对象的异常处理功能在这个领域提供帮助。在编写良好的

Java 程序中，所有运行时错误都能够并且应当由程序进行管理。

1.6.4 多线程

Java 的设计目标之一是满足对创建交互式、网络化程序的现实需求。为了满足这一目标，

Java 支持多线程编程，允许编写同时执行许多工作的程序。Java 运行时系统为多线程同步提供了优美且完善的解决方案，能够创造出运行平稳的交互式系统。Java 提供了易用的多线程方法，使得您只需要考虑程序的特定行为，而不需要考虑多任务子系统。

1.6.5 体系结构中立

对于 Java 设计人员来说，核心问题是程序代码的持久性和可移植性。在创建 Java 时，程序员面临的一个主要问题是，即使是在同一台机器上也不能保证今天编写的程序到了明天仍然能够运行。操作系统升级、处理器升级以及核心系统资源的变化，都可能导致程序出现

1. 第Ⅰ部分 Java 语言

故障。Java 设计人员对 Java 语言做出了一些艰难的决策，Java 虚拟机就是试图用于解决这个问题的。它们的目标是“编写一次，无论何时、何地都能永远运行”。在很大程度上，这个目标已经实现了。

1.6.6 解释执行和高性能

正如在前面描述的，Java 通过将源代码编译成称为 Java 字节码的中间表示形式，可以创建跨平台的程序。这种代码可以在所有实现了 Java 虚拟机的系统上运行。以前大部分对跨平台解决方案的尝试对性能的影响太大。在前面解释过，Java 字节码经过了仔细设计，通过使用即时编译器，可以很容易地将字节码直接转换为高性能的本机代码。Java 运行时系统提供了这个特性，并且没有丢失平台独立代码的优点。

1.6.7 分布式

Java 是针对 Internet 的分布式环境而设计的，因为它能处理 TCP/IP 协议。实际上，使用 URL 访问资源与访问文件没有多大区别。 Java 还支持远程方法调用 (Remote Method Invocation，RMI)。这个特性允许程序通过网络调用方法。

1.6.8 动态性

Java 程序本身带有大量的运行时类型信息，这些信息可以用于在运行时验证和解决对象

访问问题。这使得以一种安全、方便的方式动态地连接代码成为可能。对于那些可以在运行的系统中动态更新小段字节码的 Java 环境的健壮性来说，这一特性也是很关键的。

**1.7** **Java 的演变历程**

Java 的最初发布不亚于一场革命，但是它并不标志着 Java 快速革新时代的结束。与其他大多数软件系统经常进行小的、增量式的改进不同，Java 持续以爆炸性的步伐向前发展。在 Java 1.0 发布不久，Java 的设计人员就已经创建了 Java 1.1。Java 1.1 新增的特性比您可能认为的在次要版本中修订增加的内容更重要并且更丰富。Java 1.1 添加了许多新的库元素，改进了事件处理方式，并且重新配置了 Java 1.0 版本中库的许多特性。它还建议不再使用(放弃)

最初由 Java 1.0 定义的一些特性。因此，Java 1.1 既添加了新特性，也去掉了最初版本中的一些特性。

Java 的下一个主要发布版本是 Java 2，此处的“2”表示“第二代”。Java 2 的创建是一个分水岭，它标志着 Java“新时代”的开始。首次发布 Java 2 时使用的版本号是 1.2，这看起来可能有些奇怪。因为 1.2 最初指的是 Java 库的内部版本号，但是之后被推广至表示整个

发布版本。通过 Java 2，Sun 公司将 Java 产品重新包装成 J2SE(Java 2 Platform Standard Edition，

Java 2 平台标准版)，并且版本号也开始用于这个产品。

Java 2 添加了大量新特性，例如 Swing 和集合框架，并且改进了 Java 虚拟机和各种编程工具。Java 2 也建议不再使用某些特性。最重要的影响是 Thread 类，建议不再使用该类中的

suspend()、resume()和 stop()等方法。

J2SE 1.3 是对 Java 2 原始版本的第一次重要升级。这次升级主要是更新 Java 的现有功能

第1章 Java 的历史和演变 **13**

以及“限制”开发环境。一般来说，为版本 1.2 和 1.3 编写的程序的源代码是兼容的。尽管版本 1.3 包含的变化比以前三次重要升级更小，但这次升级仍然是十分重要的。

J2SE 1.4 的发布进一步增强了 Java。这个发布版本包含了一些重要的升级、改进和新增功能。例如，添加了新的关键字 assert、链式异常(chained exception)以及基于通道的 I/O 子系统。该版本还对集合框架和联网类(networking class)进行了修改。此外，从头到尾还有大量小的改动。尽管引入了很多新特性，但版本 1.4 与以前的版本保持了几乎百分之百的源代码级兼容。

Java 1.4 之后的下一个发布版本是 J2SE 5，该版本也是革命性的。与先前的大多数 Java

升级不同，该版本提供了重要、但是有规律的改进，J2SE 5 从根本上扩展了 Java 语言的应用领域、功能和范围。为了领会 J2SE 5 对 Java 修改的重要性，考虑下面列出的 J2SE 5 的主要新特性：

* 泛型
* 注解(annotation)
* 自动装箱和自动拆箱
* 枚举
* 增强的 for-each 风格的 for 循环
* 可变长度参数(varargs)
* 静态导入
* 格式化的 I/O
* 并发实用工具上述新特性不是细枝末节的改动或增量式的升级，列表中的每一项都表示对 Java 语言的

重大弥补。某些新特性，比如泛型、增强的 for-each 风格的 for 循环以及可变长度参数，引入了新的语法元素。其他新特性，比如自动装箱和自动拆箱，改变了 Java 语言的语义。注解为编程增加了一个全新的维度。所有这些新增特性的影响都超出了它们的直接效果。它们极大地改变了 Java 本身的每一特性。

这些新增特性的重要性反映在版本号“5”的使用上。正常情况下，Java 的下一个版本号应为 1.5。然而，这些新增特性实在是太重要了，以至于从 1.4 转向 1.5 不足以看出变化的重要性。因此，Sun 公司选择将版本号增至 5，以强调一个重要事件正在发生。因此，该版本被命名为 J2SE 5，同时开发人员的工具包也称为 JDK 5。不过，为了保持一致性，Sun 公

司决定使用 1.5 作为内部版本号，它也指开发版本号。J2SE 5 中的“5”被称为产品版本号。

Java 的下一个发布版本称为 Java SE 6。Sun 公司再次决定改变 Java 平台的名称。首先，注意“2”已经被取消了。因此，平台现在被命名为 Java SE，官方产品名称是“Java Platform，

Standard Edition 6”。与 J2SE 5 中的“5”一样，Java SE 6 中的“6”是产品版本号，而内部的

开发版本号是 1.6。

Java SE 6 建立在 J2SE 5 的基础之上，进行了一些增量式的改进。Java SE 6 没有为 Java

语言添加真正重要的新特性，但它确实增强了 API 库，添加了几个新的包，并且对运行时进行了改进。随着几次升级，在漫长的生命周期中 Java SE 6 还进行了几次更新。总之，Java SE 6

进一步巩固了 J2SE 5 的发展成果。

**14** 第Ⅰ部分 Java 语言

Java SE 7

Java 最新的发布版本是 Java SE 7，Java 开发人员工具包也随之被称为 JDK 7，并且内部

版本号为 1.7。Java SE 7 是自从 Sun Microsystems 被 Oracle 公司收购(这一过程始于 2009 年 4

月，并在 2010 年 1 月最终完成)之后第一个重要的发布版本。Java SE 7 包含许多新特性，包括为 Java 语言增加的重要特性和 API 库，并且对 Java 运行时系统进行了升级，升级的内容包括对非 Java 语言的支持。不过对 Java 开发人员来说，他们最感兴趣的还是为语言和 API

增加的特性。

新增的语言特性是作为 Project Coin 的一部分开发的。Project Coin 的目的是识别大量将被合并到 JDK 7 中的对 Java 语言的小修改。尽管这些新特性被集中描述为“小的”修改，但就它们对代码的影响而言，这些修改产生的影响却相当大。实际上，对于许多开发人员，这些修改可能是 Java SE 7 中最重要的新特性。下面是新增语言特性的列表：

* String 现在能够控制 switch 语句。
* 二进制整型字面值。
* 数值字面值中的下划线。
* 扩展的 try 语句，称为带资源的 try(try-with-resources)语句，这种 try 语句支持自动资源管理(例如，当流(stream)不再需要时，现在能够自动关闭它们)。
* 构造泛型实例时的类型推断(借助菱形“<>”运算符)。
* 对异常处理进行了增强，单个 catch(multi-catch)子句能够捕获两个或更多个异常，并且对重新抛出的异常提供了更好的类型检查。
* 对与某些方法(参数的长度可变)类型关联的编译器警告进行了改进，尽管语法没有发生变化，并且对警告具有更大的控制权。

正如您可能看到的，尽管 Projection Coin 特性被认为是对语言小的修改，但是它们带来的好处却是很大的。特别地，带资源的 try 语句将会对编写基于流的代码的方式产生深远影响。此外，使用 String 控制 switch 语句的能力是大家盼望已久的改进，在许多情况下这一改进将会简化代码。

Java SE 7 为 Java API 库新增了一些内容。其中最重要的两个方面是对 NIO 框架进行了

增强并且增加了 Frok/Join 框架。NIO(最初表示新 I/O(New I/O))是在 1.4 版本中被添加到 Java

中的。然而，Java SE 7 对 NIO 的增强从根本上扩展了它的功能。这一修改非常重要，以至于经常使用术语 NIO.2。

Frok/Join 框架对并行编程(parallel programming)提供了重要支持。并行编程通常是指有效

使用具有多个处理器(包括多核系统)的计算机的技术。多核环境提供的优点是可以在相当大的程度上提高程序的性能。Fork/Join 框架通过以下两个方面对多核编程提供支持：

* 简化同时执行的任务的创建和使用。
* 自动使用多个处理器。

所以，使用 Fork/Join 框架可以很容易地创建可伸缩的应用程序，它们能够自动利用执行环境中的可用处理器。当然，并不是所有的算法都可以并行执行，但是如果算法确实可以并行执行的话，执行速度就可以得到相当大的提升。

本书内容已经进行了更新以反映 Java SE 7 中的变化，涵盖许多新特性、更新以及新增内容。

第1章 Java 的历史和演变 **15**

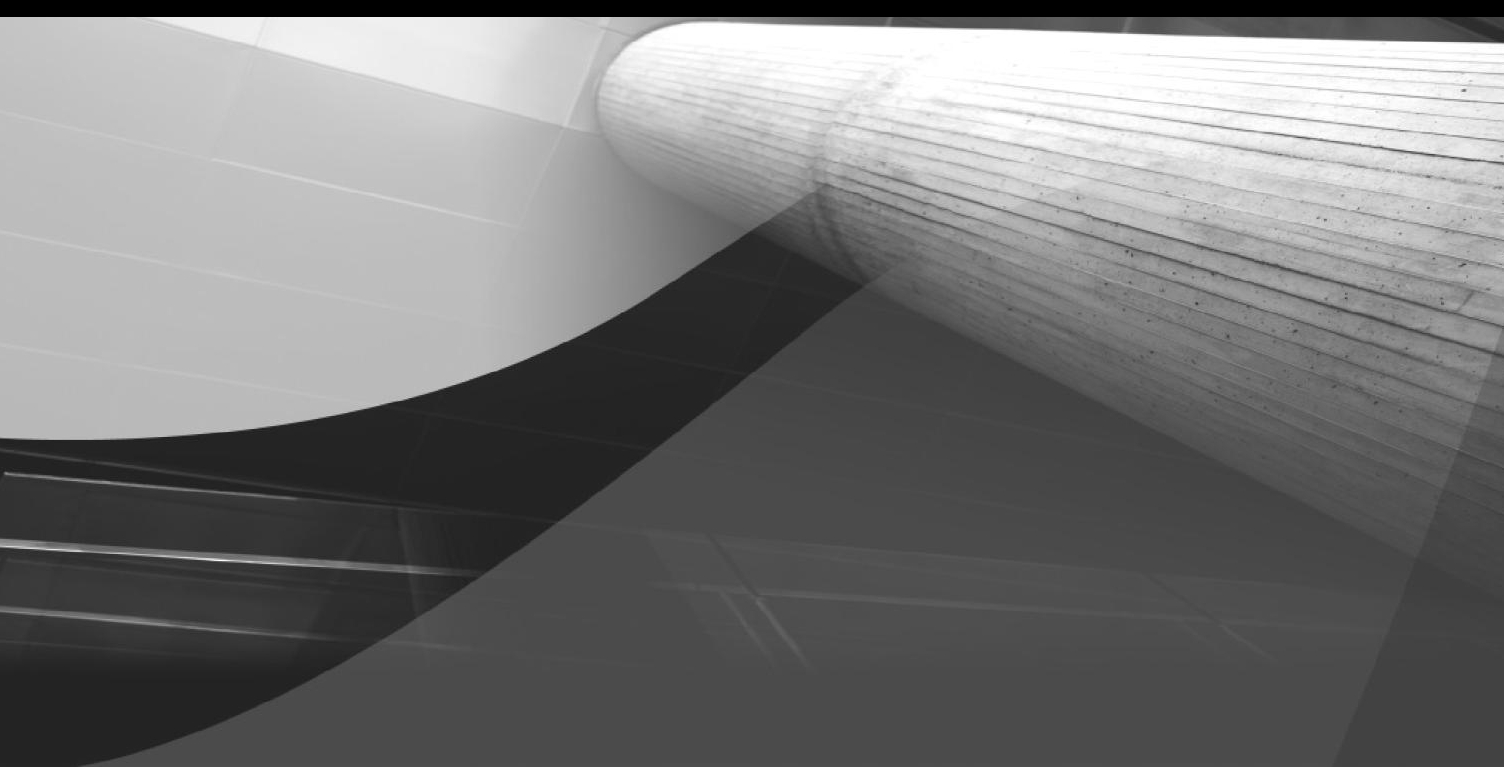
**1.8** **文化革新**

从一开始，Java 就位于文化革新的中心位置。它的首次发布就重新定义了 Internet 编程。 Java 虚拟机(JVM)和字节码改变了我们对安全性和可移植性的思考方式。applet(以及 servlet)

使 Web 充满活力。Java 标准制定组织(Java Community Process，JCP)重新定义了编程语言吸收新思想的方式。因为 Java 还被用于 Android 编程，所以 Java 也是智能手机革命的一部分。

Java 世界从来都不是长时间静止不前的，Java SE 7 是 Java 不断前进的动态历史中最新发布的

版本。



**第 3 章**

数据类型、变量和数组

本章分析 Java 中最基本的 3 个元素：数据类型、变量与数组。与所有现代编程语言一样，

Java 支持好几种类型的数据。可以使用这些类型声明变量并创建数组。正如您将会看到的，

Java 支持这些元素的方式是清晰、高效并且内聚的。

**3.1** **Java 是强类型化的语言**

Java 是一种强类型化的语言，在开始时指出这一点是很重要的。实际上，Java 的部分安

全性和健壮性正是来自这一事实。强类型化意味着什么呢？首先，每个变量具有一种类型，

**36** 第Ⅰ部分 Java 语言

每个表达式具有一种类型，并且每种类型都是严格定义的。其次，所有赋值，不管是显式的还是在方法调用中通过参数传递的，都要进行类型兼容性检查。在有些语言中，则不对存在

冲突的类型进行自动强制转换。Java 编译器检查所有表达式和参数，以确保类型是兼容的。任何类型不匹配都是错误，在编译器完成类的编译之前必须改正这些错误。

**3.2** **基本类型**

Java 定义了 8 种基本数据类型：byte、short、int、long、char、float、double 和 boolean。

基本类型通常也称为简单类型，并且在本书中这两个术语都会使用。这些类型可以被分成 4 组：

* **整型** 这一组包括byte、short、int和long，它们用于全数值有符号数字。
* **浮点型** 这一组包括float和double，它们表示带小数位的数字。
* **字符型** 这一组包括char，表示字符集中的符号，比如字母和数字。
* **布尔型** 这一组包括boolean，是一种用于表示true/false值的特殊类型。

可以直接使用这些类型，也可以使用它们构造数组以及自定义类型。因此，它们形成了所有可以创建的其他类型的基础。

基本类型表示单个值——而不是复杂对象。尽管 Java 是完全面向对象的，但是基本类型不是面向对象的。它们与大多数其他非面向对象语言中的简单类型类似。这样设计的原因是效率。将基本类型设计成对象会极大地降低性能。

基本类型被定义为具有明确的范围和数学行为。C 和 C++这类语言允许整数的大小随着

执行环境的要求而变化。然而，Java 与之不同。因为 Java 需要具备可移植性，所有数据类型都具有严格定义的范围。例如，无论在哪种特定平台上，int 总是 32 位的，因而可以编写出不经修改就能确保在任何体系结构的计算机上都能运行的程序。虽然严格指定整数的范围在某些环境中可能会造成一些性能损失，但为了实现可移植性这么做是必要的。

下面依次分析每种数据类型。

**3.3** **整型**

Java 定义了 4 种整数类型：byte、short、int 和 long。所有这些类型都是有符号的、正的或负的整数。Java 不支持无符号的、只是正值的整数。许多其他计算机语言同时支持有符号和无符号整数。然而，Java 的设计者觉得无符号整数不是必需的。特别是，他们觉得“无符号”的概念通常用于指定“高阶位”(high-order bit)的行为，高阶位用于定义整型值的符号。在第 4 章将会看到，Java 通过添加特殊的“无符号右移”运算符，以稍微不同的方式管理高阶位的含义。因此，Java 消除了对无符号整数类型的需要。

不应将整数类型的宽度看成整数所消耗的存储量，而应当理解成定义这种类型的变量和

表达式的行为。Java 运行时环境可以自由使用它们希望的、任何大小的空间，只要类型的行为符合声明它们时的约定即可。这些整数类型的宽度和范围相差很大，如表 3-1 所示。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | 第 3 章 | 数据类型、变量和数组**37** | |
|  |  |  |  |  | 表 3-1 整数类型的宽度和范围 |  |  |  |
|  | 名 | 称 | 宽 | 度 | 范 | 围 |  |  |
|  |  |  |  |  |  | | |  |
|  |  | long |  | 64 | −9 223 372 036 854 775 808～9 223 372 036 854 775 807 | | | |
|  |  | int |  | 32 | −2 147 483 648～2 147 483 647 |  |  |  |
|  |  | short |  | 16 | −32 768～32 767 |  |  |  |
|  |  | byte |  | 8 | −128～127 |  |  |  |

接下来看一看每种整数类型。

3.3.1 byte

最小的整数类型是 byte。它是有符号的 8 位类型，范围为−128～127。当操作来自网络或文件的数据流时，byte 类型的变量特别有用。当操作与 Java 的其他内置类型不直接兼容的原始二进制数据时，byte 类型的变量也很有用。

字节变量是通过关键字 byte 声明的。例如，下面声明了两个 byte 变量 b 和 c：

byte b, c;

3.3.2 short

short 是有符号的 16 位类型。它的范围为−32 768～32 767。它可能是最不常用的 Java 类

型。下面是声明 short 变量的一些例子：

short s; short t;

3.3.3 int

最常用的整数类型是 int。它是有符号的 32 位类型，范围为−2 147 483 648～2 147 483 647。

除了其他用途外，int 类型变量通常用于控制循环和索引数组。对于那些不需要更大范围的 int

类型数值的情况，您可能会认为使用范围更小的 byte 和 short 类型效率更高，然而事实并非如此。原因是如果在表达式中使用 byte 和 short 值，当对表达式求值时它们会被提升(promote)

为 int 类型(类型提升将在本章后面描述)。所以，当需要使用整数时，int 通常是最好的选择。

3.3.4 long

long 是有符号的 64 位类型，对于那些 int 类型不足以容纳期望数值的情况，long 类型是有用的。long 类型的范围相当大，这使得当需要很大的整数时它非常有用。例如，下面的程序计算光在指定的天数中传播的距离(以英里为单位)：

// Compute distance light travels using long variables. class Light {

**38** 第Ⅰ部分 Java 语言

public static void main(String args[]) { int lightspeed;

long days; long seconds; long distance;

// approximate speed of light in miles per second lightspeed = 186000;

days = 1000; // specify number of days here

seconds = days \* 24 \* 60 \* 60; // convert to seconds

distance = lightspeed \* seconds; // compute distance

System.out.print("In " + days);

System.out.print(" days light will travel about "); System.out.println(distance + " miles.");

}

}

这个程序产生的输出如下所示：

In 1000 days light will travel about 16070400000000 miles.

显然，int 变量无法保存这么大的结果。

**3.4** **浮点型**

浮点数也称为实数(real number)，当计算需要小数精度的表达式时使用。例如，求平方根这类计算以及和正弦和余弦这类超越数，保存结果就需要使用浮点类型。Java 执行 IEEE-754

标准集的浮点类型和运算符。有两种浮点类型——float 和 double，它们分别表示单精度和双精度浮点数。它们的宽度和范围如表 3-2 所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | 表 3-2 | 浮点型的宽度和范围 | |
|  | 名 | 称 | 宽度(位) |  | 大 致 范 围 |
|  | double | | 64 |  | 4.9e－324～1.8e+308 |
|  |  | float | 32 |  | 1.4e－045～3.4e+038 |

下面详细解释每种浮点类型。

3.4.1 float

float 类型表示使用 32 位存储的单精度(single-precision)数值。在某些处理器上，单精度

运算速度更快，并且占用的空间是双精度的一半，但是当数值非常大或非常小时会变得不精

第3章 数据类型、变量和数组 **39**

确。如果需要小数部分，并且精度要求不是很高时，float 类型的变量是很有用的。例如，表示美元和美分时可以使用 float 类型。

下面是声明 float 变量的一些例子：

float hightemp, lowtemp;

3.4.2 double

双精度使用 double 关键字表示，并使用 64 位存储数值。在针对高速数学运算进行了优化的某些现代处理器上，实际上双精度数值的运算速度更快。所有超越数学函数，如 sin()、cos()和 sqrt()，都返回双精度值。如果需要在很多次迭代运算中保持精度，或是操作非常大的数值，double 类型是最佳选择。

下面的简短程序使用 double 变量计算圆的面积：

// Compute the area of a circle. class Area {

public static void main(String args[]) { double pi, r, a;

r = 10.8; // radius of circle

pi = 3.1416; // pi, approximately

a = pi \* r \* r; // compute area

System.out.println("Area of circle is " + a);

}

}

1. **字符型**
   * Java 中，用于存储字符的数据类型是 char。然而，C/C++程序员要当心：Java 中的 char 与 C 或 C++中的 char 是不同的。在 C/C++中，char 的宽度是 8 位。而在 Java 中不是这样的。相反，Java 使用 Unicode 表示字符。Unicode 定义了一个完全国际化的字符集，能够表示全

部人类语言中的所有字符。Unicode 是数十种字符集的统一体，比如拉丁字符集、希腊字符集、阿拉伯字符集、斯拉夫语字符集、希伯来语字符集、日文字符集、韩文字符集等。为此，

Unicode 需要 16 位宽。因此，在 Java 中 char 是 16 位类型。char 的范围为 0～65 536。没有负的 char 值。著名的 ASCII 标准字符集，其范围仍然是 0～127；而扩展的 8 位字符集

ISO-Latin-1，其范围是 0～255。既然 Java 的设计初衷是允许程序员编写在世界范围内均可使用的程序，那么使用 Unicode 表示字符是合理的。当然，对于英语、德语或法语这类语言，使用 Unicode 在一定程度上会降低效率，因为可以很容易地使用 8 位表示这类语言的字符。但这是为了在全球获得可移植性而必须付出的代价。

**40** 第Ⅰ部分 Java 语言

注意：



在 http//www.unicode.org 上可以找到有关 Unicode 的更多信息。

下面是演示 char 变量用法的一个程序：

// Demonstrate char data type. class CharDemo {

public static void main(String args[]) { char ch1, ch2;

ch1 = 88; // code for X ch2 = 'Y';

System.out.print("ch1 and ch2: "); System.out.println(ch1 + " " + ch2);

}

}

这个程序显示如下所示的输出：

ch1 and ch2: X Y

注意，ch1 被赋值为 88，该值是与字母 X 对应的 ASCII(以及 Unicode)值。前面提到过， ASCII 字符集占用 Unicode 字符集中的前 127 个值。因此，在其他语言中对字符使用的所有“旧式技巧"，在 Java 中仍然管用。

尽管 char 被设计成容纳 Unicode 字符，但它也可以用作整数类型，可以对 char 类型的变量执行算术运算。例如，可以将两个字符相加到一起，或者增加字符变量的值。分析下面的程序：

// char variables behave like integers. class CharDemo2 {

public static void main(String args[]) { char ch1;

ch1 = 'X';

System.out.println("ch1 contains " + ch1);

ch1++; // increment ch1 System.out.println("ch1 is now " + ch1);

}

}

这个程序产生的输出如下所示：

ch1 contains X

第3章 数据类型、变量和数组 **41**

ch1 is now Y

在该程序中，首先将 X 赋给 ch1，然后递增 ch1 的值。现在 ch1 中包含的结果 Y，是 ASCII(以及 Unicode)序列中的下一个字符。

注意：



在 Java 的正式规范中，char 被当作整数类型，这意味着它和 int、short、long以及 byte 位于同一分类中。然而，因为 char 类型的主要用途是表示 Unicode

字符，所以通常考虑将 char 放到单独的分类中。

**3.6** **布尔型**

Java 有一个称为 boolean 的基本类型，用于表示逻辑值。它只能是两个可能的值之一：true

或 false。所有关系运算(例如 a<b)都返回这种类型的值。对于 if 和 for 这类控制语句的条件表达式，也需要 boolean 类型。

下面的程序演示了 boolean 类型：

// Demonstrate boolean values. class BoolTest {

public static void main(String args[]) { boolean b;

b = false; System.out.println("b is " + b); b = true;

System.out.println("b is " + b);

// a boolean value can control the if statement if(b) System.out.println("This is executed.");

b = false;

if(b) System.out.println("This is not executed.");

// outcome of a relational operator is a boolean value System.out.println("10 > 9 is " + (10 > 9));

}

}

这个程序产生的输出如下所示：

b is false b is true

This is executed. 10 > 9 is true

关于这个程序有三个有趣的地方需要注意。首先可以看出，当通过 println()方法输出

**42** 第Ⅰ部分 Java 语言

boolean 值时，显示的是 true 或 false。其次，对于控制 if 语句来说，boolean 变量的值本身是足够的。不需要像下面这样编写 if 语句：

if(b == true) …

最后，关系运算符(例如<)的输出是 boolean 值。这就是为什么表达式 10>9 显示 true 的

原因。此外，10>9 周围的圆括号是必需的，因为运算符+比>具有更高的优先级。

**3.7** **深入分析字面值**

在第 2 章已经简要提及了字面值。前面已经正式描述了内置类型，接下来深入分析字面值。

3.7.1 整型字面值

在典型的程序中，整型可能是最常用的类型。所有数字都是整型字面值，例如 1、2、3

和 42。这些都是十进制数字，表示它们是以 10 为基数描述的。在整型字面值中，还可以使用另外两种进制——八进制(以 8 为基数)和十六进制(以 16 为基数)。在 Java 中，八进制数值以 0 开头。常规的十进制数字不以 0 开头。因此，对于看似有效的值 09，编译器会产生一个错误，因为 9 超出了八进制数字 0～7 的范围。程序员针对数字更常使用的是十六进制，以便整齐地匹配以 8 为模的字的尺寸，如 8 位、16 位、32 位和 64 位。以 0x 或 0X 开头来标识十六进制常量。十六进制数字的范围是 0～15，因此分别用 A～F(或 a～f)替代数字 10～15。

整型字面值用于创建 int 类型数值，在 Java 中是 32 位的整数。既然 Java 是强类型化的，您可能会好奇 Java 如何将整型字面值赋给其他整数类型，如 byte 或 long，而不会导致类型不匹配错误？幸运的是这种情况很容易处理。当将字面值赋给 byte 或 short 变量时，如果字面值位于目标类型的范围之内，就不会产生错误。整型字面值总是可以赋给 long 变量。然而，为了标识 long 字面值，需要明确告诉编译器字面值是 long 类型的。可以通过为字面值附加一个大写或小写的 L 来明确地标识其类型为 long，例如 0x7ffffffffffffffL 或 9223372036854775807L

是最大的 long 类型的字面值。也可以将整数赋给 char，只要在 char 类型的范围之内即可。

从 JDK 7 开始，可以使用二进制指定整型字面值。为此，使用 0b 或 0B 作为数值的前缀。例如，下面这行代码使用二进制字面值指定十进制值 10：

int x = 0b1010;

除了其他用途之外，二进制字面值简化了用作位屏蔽的数值的输入。对于这种情况，十进制(或十六进制)表示的数值不能很直观地表达出与其用途相关的含义，而二进制字面值却可以。

从 JDK 7 开始，在整型字面值中还可以嵌入一个或多个下划线。嵌入下划线可以使阅读很大的整数变得更加容易。当编译字面值时，会丢弃下划线。例如，下面这行代码：

int x = 123\_456\_789;

第3章 数据类型、变量和数组 **43**

为 x 提供的值为 123 456 789，下划线将被忽略。下划线只能用于分隔数字，不能位于字面值的开头和结尾。然而，在两个数字之间使用多个下划线是允许的。例如，下面这行代码是合法的：

int x = 123\_\_\_456\_\_\_789;

当编码电话号码、消费者 ID 号、护照号码等事物时，在整型字面值中使用下划线特别有用。当指定二进制字面值时，下划线对于提供视觉分组也是有用的。例如，二进制数值经常以 4 位进行视觉分组，如下所示：

int x = 0b1101\_0101\_0001\_1010;

3.7.2 浮点型字面值

浮点数表示具有小数部分的十进制数值。可以使用标准计数法或科学计数法表示浮点数。标准计数法由前面的整数部分、其后的小数点以及小数点后面的小数部分构成。例如，2.0、3.14159 以及 0.6667 都表示有效的标准计数法浮点数。科学计数法使用一个由标准计数法表示的浮点数加上一个后缀表示，其中的后缀指定为 10 的幂，它与前面的浮点数是相乘的关系。指数部分用 E(或 e)后面跟上一个十进制数表示，该十进制数可以是正数也可以是负数，

例如 6.022E23、314159E-05 以及 2e+100。

* + Java 中，浮点型字面值默认是双精度的。为了指定浮点型字面值，必须为常量附加一
* F 或 f。也可以通过附加 D 或 d 来显式地指定 double 字面值。当然，这么做是多余的。默认的 double 类型使用 64 位存储，而更小的 float 类型只需要 32 位。

Java 也支持十六进制浮点型字面量，但是很少使用。它们必须使用与科学计数法类似的形式来表示，不过使用的是 P 或 p，而不是 E 或 e。例如，0x12.2P2 是一个有效的浮点型字面值。P 后面的数值称为二进制指数，表示 2 的幂，并且和前面的数字相乘。所以，0x12.2P2

代表 72.5。

从 JDK 7 开始，在浮点型字面值中可以嵌入一个或多个下划线。该特性和用于整型字面值时的工作方式相同，刚才已经介绍过。这一特性的目的是使阅读很大的浮点型字面值更加容易。当编译字面值时，会丢弃下划线。例如，下面这行代码：

double num = 9\_423\_497\_862.0;

将变量 num 赋值为 9 423 497 862.0，下划线会被忽略。与整型字面值一样，下划线只能用于分隔数字。它们不能位于字面值的开头或结尾。然而，在两个数字之间使用多个下划线是允许的。在小数部分中也可以使用下划线，例如：

double num = 9\_423\_497.1\_0\_9;

这是合法的。这时，小数部分是.109。

**44** 第Ⅰ部分 Java 语言

3.7.3 布尔型字面值

布尔型字面值很简单。布尔型只有两个逻辑值——true 和 false。true 和 false 不能转换成任何数字表示形式。在 Java 中，字面值 true 不等于 1，字面值 false 也不等于 0。在 Java 中，只能将布尔型字面值赋给以布尔型声明的变量，或用于使用布尔运算符的表达式中。

3.7.4 字符型字面值

Java 中的字符被索引到 Unicode 字符集，它们是可以转换成整数的 16 位值，并且可以使

用整数运算符进行操作，例如加和减运算符。字符型字面值使用位于一对单引号中的字符来表示。所有可见的 ASCII 字符都可以直接输入到单引号中，如'a'、'z'以及'@'。对于那些不能直接输入的字符，可以使用转义字符序列输入需要的字符，例如'\"表示单引号、'\n'表示换行符。还有一种以八进制或十六进制直接输入字符值的机制。对于八进制表示法，使用反斜杠后跟三位数字表示，例如'\141'是字母'a'。对于十六进制，先输入“\u”，然后是 4 位的十六进制数。例如'\u0061'表示 ISO-Latin-1 字符'a'，因为第一个字节为 0；'\ua432'是一个日本片假名字符。表 3-3 显示了字符转义序列。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 表 3-3 | 字符转义序列 |  |
|  | 转 义 序 列 |  | 描 | 述 |
|  |  |  |  |  |
|  | \ddd | 八进制字符(ddd) |  |  |
|  | \uxxxx | 十六进制 Unicode 字符(xxxx) | |  |
|  | \' | 单引号 |  |  |
|  | \" | 双引号 |  |  |
|  | \\ | 反斜杠 |  |  |
|  | \r | 回车符 |  |  |
|  | \n | 新行符(也称为换行符) |  |  |
|  | \f | 换页符 |  |  |
|  | \t | 制表符 |  |  |
|  | \b | 回格符 |  |  |

1. 字符串字面值
   * Java 中，指定字符串字面值的方法与其他大多数语言相同——使用位于一对双引号中的字符序列。下面是字符串字面值的几个例子：

"Hello World" "two\nlines"

" \"This is in quotes\""

为字符型字面值定义的转义序列和八进制/十六进制表示法，在字符串字面值中同样适

第3章 数据类型、变量和数组 **45**

用。关于 Java 字符串需要重点指出的是，它们的开头和结尾必须位于同一行中。与其他某些语言不同，在 Java 中没有续行的转义序列。

注意：



您可能知道，在其他某些语言中，包括 C/C++，字符串是作为字符数组实现的。然而，在 Java 中不是如此。在 Java 中，字符串实际上是对象类型。在本书的后面将会看到，因为 Java 将字符串作为对象实现，所以提供了广泛的、功能强大且易于使用的字符串处理功能。

1. **变量**
   * Java 程序中，变量是基本存储单元。变量是通过联合标识符、类型以及可选的初始化器来定义的。此外，所有的变量都有作用域，作用域定义了变量的可见性和生命周期。下面分析这些元素。
2. 变量的声明
   * Java 中，所有变量在使用之前必须声明。声明变量的基本形式如下所示：

type identifier [ = value ][, identifier [= value ] …];

其中，type 是 Java 的原子类型或是类或接口的名称(类和接口类型将在本书第Ⅰ部分的后面讨论)。identifier 是变量的名称。可以通过指定一个等号和一个值来初始化变量。请牢记，初始化表达式的结果类型必须与为变量指定的类型相同(或兼容)。为了声明指定类型的多个变量，需要使用以逗号分隔的列表。

下面是声明各种类型变量的一些例子，注意有些变量声明包含初始化部分：

int a, b, c; // declares three ints, a, b, and c.

int d = 3, e, f = 5; // declares three more ints, initializing

// d and f.

byte z = 22; // initializes z.

double pi = 3.14159; // declares an approximation of pi.

char x = 'x'; // the variable x has the value 'x'.

在此选择的标识符与用来指定变量类型的名称没有任何内在联系。Java 允许将任何形式正确的标识符声明为任何类型。

3.8.2 动态初始化

尽管前面的例子只使用常量作为初始化器，但是在声明变量时，Java 也允许使用任何有效的表达式动态地初始化变量。

例如，下面的简短程序根据直角三角形的两条直角边来计算斜边的长度：

1. 第Ⅰ部分 Java 语言
   * Demonstrate dynamic initialization. class DynInit {

public static void main(String args[]) { double a = 3.0, b = 4.0;

// c is dynamically initialized double c = Math.sqrt(a \* a + b \* b);

System.out.println("Hypotenuse is " + c);

}

}

在此，声明了三个局部变量：a、b 和 c。其中的前两个变量 a 和 b，使用常量进行初始化，而 c 被动态初始化为斜边的长度(使用勾股定理)。该程序使用了另外一个内置的 Java 方法 sqrt()，该方法是 Math 类的成员，用于计算参数的平方根。在此的关键点是，初始化表达式可以使用任何在初始化时有效的元素，包括方法调用、其他变量或字面值。

3.8.3 变量的作用域和生命周期

到目前为止，使用的所有变量都是在 main()方法开始时声明的。然而，Java 允许在任何代码块中声明变量。正如在第 2 章中所解释的，代码块以开花括号开始并以闭花括号结束。代码块定义了作用域。因此，每当开始一个新的代码块时就创建了一个新的作用域。作用域决定了对象对程序其他部分的可见性，并且也决定了这些对象的生命周期。

许多其他计算机语言定义了两种通用的作用域类别：全局作用域和局部作用域。然而，这些传统的作用域不能很好地适应 Java 中严格的、面向对象的模型。虽然可以创建属于全局作用域的变量，但这只是例外，而不是规则。在 Java 中，两种主要的作用域分别是由类和方法定义的。尽管这种分类有些人为因素，但是，由于类作用域具有的一些独特属性和特性，不能应用于由方法定义的作用域，因此这种分类方法有一定的道理。由于存在这种差别，对类作用域(以及在其中声明的变量)的讨论将推迟到第 6 章，那时会介绍类的相关内容。现在，只分析由方法定义以及在方法中定义的作用域。

由方法定义的作用域从方法的开花括号开始。然而，如果方法具有参数，那么它们也会被包含到方法的作用域中。尽管本书在第 6 章会进一步分析参数，但是出于作用域讨论的目的，在此将它们与任何其他方法变量一样一并考虑。

作为通用规则，在作用域中声明的变量，对于在作用域之外定义的代码是不可见的(即不可访问)。因此，当在某个作用域中声明变量时，就局部化了该变量，并保护它免受未授权的访问和/或修改。实际上，作用域规则为封装提供了基础。

作用域是可以嵌套的。例如，每当创建一个代码块时，就创建了一个新的、嵌套的作用域。当遇到这种情况时，外层的作用域包围了内层作用域。这意味着在外层作用域中声明的对象对于内层作用域中的代码是可见的。然而，反过来就不是这样了，在内层作用域中声明的对象，在内层作用域之外是不可见的。

为了理解嵌套作用域的影响，分析下面的程序：

第3章 数据类型、变量和数组 **47**

// Demonstrate block scope. class Scope {

public static void main(String args[]) { int x; // known to all code within main

x = 10;

if(x == 10) { // start new scope

int y = 20; // known only to this block

// x and y both known here. System.out.println("x and y: " + x + " " + y); x = y \* 2;

}

* y = 100; // Error! y not known here
* x is still known here. System.out.println("x is " + x);

}

}

正如注释指出的，变量 x 是在 main()作用域的开始处声明的，因此 main()方法中的所有后续代码都可以访问变量 x。变量 y 是在 if 代码块中声明的，既然代码块定义了作用域，所以只有对于 if 代码块中的代码 y 才是可见的。这就是为什么在 if 代码块之外，将“y=100;”

注释掉的原因。如果删除前面的注释符号，就会发生编译时错误，因为在 if 代码块之外 y 不

是可见的。在 if 代码块的内部可以使用 x，因为代码块(即嵌套的作用域)中的代码可以访问在外部作用域中声明的变量。

在代码块中，可以在任意位置声明变量，但是只有在声明之后变量才是有效的。因此，如果在方法的开头定义变量，那么变量对于该方法的所有代码都是可见的。相反，如果在代码块的末尾声明变量，那么变量是无用的，因为没有代码能够访问该变量。例如，下面的代码片段是无效的，因为 count 在声明之前不能使用：

// This fragment is wrong!

count = 100; // oops! cannot use count before it is declared! int count;

下面是另外一个重点：当进入变量的作用域时创建变量，当离开它们的作用域时销毁变量。这意味着一旦离开作用域，变量就不会再保持原来的值。所以，对于在方法中声明的变量来说，在两次调用该方法之间，变量不会保持它们的值。此外，对于在代码块中声明的变量来说，当离开代码块时会丢失它们的值。因此，变量的生命周期被限制在作用域之内。

如果变量声明包含初始化器，那么每当进入声明变量的代码块时都会重新初始化变量。例如，分析下面的程序：

// Demonstrate lifetime of a variable. class LifeTime {

public static void main(String args[]) {

**48** 第Ⅰ部分 Java 语言

int x;

for(x = 0; x < 3; x++) {

int y = -1; // y is initialized each time block is entered System.out.println("y is: " + y); // this always prints -1 y = 100;

System.out.println("y is now: " + y);

}

}

}

这个程序产生的输出如下所示：

y is: -1

y is now: 100

y is: -1

y is now: 100

y is: -1

y is now: 100

可以看出，每次进入内部的 for 循环时 y 都被重新初始化为−1。尽管随后 y 被赋值为 100，

但是这个值丢失了。

最后一点：尽管可以嵌套代码块，但是在内层代码块中不能声明与外层代码块具有相同名称的变量。例如，下面的程序是非法的：

// This program will not compile class ScopeErr {

public static void main(String args[]) { int bar = 1;

{// creates a new scope

int bar = 2; // Compile-time error – bar already defined!

}

}

}

**3.9** **类型转换和强制类型转换**

如果您已经具备了前面的编程经验，就会知道将某种类型的值赋给另外一种类型的变量是很常见的。如果这两种类型是兼容的，那么 Java 会自动进行类型转换。例如，总是可以将 int 类型的值赋给 long 类型的变量。然而，并不是所有类型都是兼容的，从而也不是所有类型转换默认都是允许的。例如，没有定义从 double 类型到 float 类型的自动转换。幸运的是，在两种不兼容的类型之间，仍然可以进行转换。为此，必须使用强制类型转换(cast)，在不兼容的类型之间执行显式转换。下面分析自动类型转换和强制类型转换这两种情况。

第3章 数据类型、变量和数组 **49**

3.9.1 Java 的自动类型转换

当将某种类型的数据赋给另外一种类型的变量时，如果满足如下两个条件，就会发生自动类型转换：

* 两种类型是兼容的。
* 目标类型大于源类型。

当满足这两个条件时，会发生扩宽转换(widening conversion)。例如，要保存所有有效的 byte 值，int 类型总是足够的，所以不需要显式的强制转换语句。

对于扩宽转换，数值类型(包括整型和浮点型)是相互兼容的。然而，不存在从数值类型到 char 或 boolean 类型的转换。此外，char 和 boolean 相互之间也不是兼容的。

在前面提到过，当将字面整数常量保存到 byte、short、long 或 char 类型的变量中时，Java

也会执行自动类型转换。

3.9.2 强制转换不兼容的类型

尽管自动类型转换很有帮助，但是它们不能完全满足全部需要。例如，如果希望将 int

类型的值赋给 byte 变量，会发生什么情况呢？不会自动执行转换，因为 byte 比 int 更小。这种转换有时被称为缩小转换(narrowing conversion)，因为是显式地使数值变得更小以适应目标类型。

为了实现两种不兼容类型之间的转换，必须使用强制类型转换。强制类型转换只不过是一种显式类型转换，它的一般形式如下所示：

(target-type) value

其中，target-type 指定了期望将特定值转换成哪种类型。例如，下面的代码片段将 int 类

型的值强制转换为 byte 类型。如果整数的值超出了 byte 类型的范围，结果将以 byte 类型的范围为模(用整数除以 byte 范围后的余数)减少。

int a; byte b;

// …

b = (byte) a;

当将浮点值赋给整数类型时会发生另一种不同类型的转换：截尾(truncation)。您知道，整数没有小数部分。因此，当将浮点值赋给整数类型时，小数部分会丢失。例如，如果将数值 1.23 赋给一个整数，结果值为 1，0.23 将被截去。当然，如果整数部分的数值太大，以至于无法保存到目标整数类型中，那么数值将以目标类型的范围为模减少。

下面的程序演示了一些需要进行强制类型转换的转换：

// Demonstrate casts. class Conversion {

public static void main(String args[]) { byte b;

**50** 第Ⅰ部分 Java 语言

int i = 257; double d = 323.142;

System.out.println("\nConversion of int to byte."); b = (byte) i;

System.out.println("i and b " + i + " " + b);

System.out.println("\nConversion of double to int."); i = (int) d;

System.out.println("d and i " + d + " " + i);

System.out.println("\nConversion of double to byte."); b = (byte) d;

System.out.println("d and b " + d + " " + b);

}

}

这个程序产生的输出如下所示：

Conversion of int to byte. i and b 257 1

Conversion of double to int. d and i 323.142 323

Conversion of double to byte. d and b 323.142 67

下面对每个转换进行分析。当数值 257 被强制转换为 byte 变量时，结果是 257 除以 256(byte 类型的范围)的余数，也就是 1。当将 d 转换成 int 类型时，小数部分丢失了。当将 d

转换成 byte 类型时，小数部分也丢失了，并且值以 256 为模减少，结果为 67。

**3.10** **表达式中的自动类型提升**

除了赋值外，还有另外一个地方也可能会发生某些类型转换：在表达式中。为了分析其中的原因，考虑下面的情况。在表达式中，中间值要求的精度有时会超出操作数的范围。例如，检查下面的表达式：

byte a = 40; byte b = 50; byte c = 100;

int d = a \* b / c;

中间部分 a\*b 的结果很容易超出 byte 操作数的范围。为了解决这类问题，当对表达式求值时，Java 自动将每个 byte、short 或 char 操作数提升为 int 类型。这意味着使用 int 类型而不是 byte 类型执行子表达式 a\*b。因此，尽管 a 和 b 都被指定为 byte 类型，中间表达式(50\*40)的结果 2000 是合法的。

第3章 数据类型、变量和数组 **51**

尽管自动类型提升很有用，但是它们会导致编译时混淆错误。例如，下面的代码看起来是正确的，但是会导致问题：

byte b = 50;

b = b \* 2; // Error! Cannot assign an int to a byte!

上面的代码试图将 50\*2——一个完全有效的 byte 值——保存在一个 byte 变量中。但是，当计算表达式的值时，操作数被自动提升为 int 类型，所以结果也被提升为 int 类型。因此，现在表达式的结果是 int 类型，如果不使用强制类型转换，就不能将结果赋给那个 byte 变量。尽管对于这个特定情况来说，所赋予的值仍然满足目标类型，但是仍然需要进行强制类型转换。

如果能理解溢出产生的后果，就应当使用显式的强制类型转换，如下所示：

byte b = 50;

b = (byte)(b \* 2);

这样就可以得到正确的值 100。

类型提升规则

Java 定义了几个应用于表达式的类型提升规则，如下所示：首先，正如刚才所描述的，所有 byte、short 和 char 类型的值都被提升为 int 类型。然后，如果有一个操作数是 long 类型，就将整个表达式提升为 long 类型；如果有一个操作数是 float 类型，就将整个表达式提升为 float 类型；如果任何一个操作数为 double 类型，结果将为 double 类型。

下面的程序演示了为了使第二个参数与每个二元运算符相匹配，如何提升表达式中的每个值：

class Promote {

public static void main(String args[]) { byte b = 42;

char c = 'a'; short s = 1024; int i = 50000; float f = 5.67f; double d = .1234;

double result = (f \* b) + (i / c) - (d \* s); System.out.println((f \* b) + " + " + (i / c) + " - " + (d \* s)); System.out.println("result = " + result);

}

}

下面进一步分析程序中如下这行代码中的类型提升：

double result = (f \* b) + (i / c) - (d \* s);

**52** 第Ⅰ部分 Java 语言

在第一个子表达式 f\*b 中，b 被提升为 float 类型，并且该子表达式的结果也是 float 类型。接下来在子表达式 i/c 中，c 被提升为 int 类型，并且结果也是 int 类型。然后在 d\*s 中，s 的值被提升为 double 类型，并且该子表达式的类型为 double。最后考虑三个中间值的类型——

float、int 和 double。float 加上 int 的结果是 float。之后，作为结果的 float 减去最后的 double，

会被提升为 double，这就是表达式最终结果的类型。

**3.11** **数组**

数组是以通用名称引用的一组类型相同的变量。可以创建任意类型的数组，并且数组可以是一维或多维的。数组中的特定元素通过索引进行访问。数组为分组相关信息提供了一种便利方法。

注意：



如果熟悉 C/C++的话，就要小心，在 Java 中，数组的工作方式与这些语言不同。

3.11.1 一维数组

一维数组本质上是一连串类型相同的变量。为了创建数组，首先必须创建期望类型的数组变量。声明一维数组的一般形式如下所示：

type var-name[];

其中，type 声明了数组的元素类型(也称为基本类型)。元素类型决定了构成数组的每个元素的类型。因此，数组的元素类型决定了数组可以包含什么类型的数据。例如，下面的语句声明了一个名为 month\_days 的数组，该数组的类型是“int 数组”：

int month\_days[];

尽管这个声明确立了 month\_days 是数组变量的事实，但是这个数组实际上并不存在。事实上，month\_days 的值被设置为 null，这表示数组没有值。为了将 month\_days 链接到一个实际的、物理的整数数组，必须使用 new 分配一个数组，并将之赋给 month\_days。new 是一个用于分配内存的特殊运算符。

在后面的章节中会更加详细地分析 new 运算符，但是现在需要使用它为数组分配内存。将 new 运算符用于一维数组的一般形式如下所示：

array-var = new type[size];

其中，type 指定了将要分配的数据的类型，size 指定了数组中元素的数量。array-var 是

链接到数组的数组变量，即为了使用 new 分配一个数组，必须指定要分配元素的类型和数量。通过 new 分配的数组，其元素会被自动初始化为 0(对于数值类型)、false(对于布尔类型)或null(对于引用类型，引用类型将在后面的章节中描述)。下面这个例子分配了一个具有 12 个

第3章 数据类型、变量和数组 **53**

元素的整数数组，并将该数组链接到 month\_day：

month\_days = new int[12];

执行完这条语句之后，month\_days 将指向具有 12 个整数的数组。此外，数组中的所有元素都被初始化为 0。

下面回顾一下：获得一个数组需要两个步骤。首先，必须声明一个期望数组类型的变量。其次，必须使用 new 分配容纳该数组的内存，并将其赋给数组变量。因此，在 Java 中所有数组都是动态分配的。如果不熟悉动态分配的概念，不要着急。在本书的后面会对其进行详细描述。

一旦分配数组，就可以通过在方括号中指定索引的方法来访问数组中的特定元素。所有数组索引都是从 0 开始的。例如，下面这条语句将数值 28 赋给 month\_days 的第 2 个元素：

month\_days[1]=28;

下面这条语句显示在索引 3 处保存的值：

System.out.println(month\_days[3]);

下面的程序将所有这些内容组合到一起，创建了一个包含一年中每个月份天数的数组：

// Demonstrate a one-dimensional array. class Array {

public static void main(String args[]) { int month\_days[];

month\_days = new int[12]; month\_days[0] = 31; month\_days[1] = 28; month\_days[2] = 31; month\_days[3] = 30; month\_days[4] = 31; month\_days[5] = 30; month\_days[6] = 31; month\_days[7] = 31; month\_days[8] = 30; month\_days[9] = 31; month\_days[10] = 30; month\_days[11] = 31;

System.out.println("April has " + month\_days[3] + " days.");

}

}

当运行这个程序时，会打印出 4 月份中的天数。前面提到过，Java 数组的索引从 0 开始，

所以 4 月份中的天数是 month\_days[3]或 30。

可以将数组变量的声明和数组本身的分配组合起来，如下所示：

**54** 第Ⅰ部分 Java 语言

int month\_days[] = new int[12];

在专业编写的 Java 程序中，通常采用的就是这种方式。

当声明数组时，可以对其进行初始化，这一过程与初始化简单类型的过程相同。数组初始化器(array initializer)是一个位于花括号中由逗号分隔的表达式列表。用逗号分隔开数组元素的值。Java 会自动创建足够大的数组，以容纳在数组初始化器中指定的元素的数量。这时不需要使用 new 运算符。例如，为了保存每个月份中的天数，下面的代码创建了一个已初始化的整数数组：

// An improved version of the previous program. class AutoArray {

public static void main(String args[]) {

int month\_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

System.out.println("April has " + month\_days[3] + " days.");

}

}

当运行这个程序时，看到的输出与程序前面版本生成的输出相同。

Java 会进行严格的检查，以确保不会意外地试图保存或引用数组范围之外的值。Java 运

行时系统会进行检查，以保证所有数组索引都在正确的范围之内。例如，运行时系统会检查 month\_days 的每个索引值，以确保它们在 0 到 11 之间。如果试图访问数组范围之外(索引为负数或大于数组长度)的元素，就会导致运行时错误。

下面是使用一维数组的另外一个例子，该例计算一组数字的平均值。

// Average an array of values. class Average {

public static void main(String args[]) {

double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5}; double result = 0;

int i;

for(i=0; i<5; i++)

result = result + nums[i]; System.out.println("Average is " + result / 5);

}

}

1. 多维数组
   * Java 中，多维数组实际上是数组的数组。可能正如您所愿，Java 中的多维数组无论是形式还是行为都与常规的多维数组类似。然而，您将会看到它们之间有一些微妙的区别。为了声明多维数组变量，需要使用另外一组方括号指定每个额外的索引。例如，下面声明了一

第3章 数据类型、变量和数组 **55**

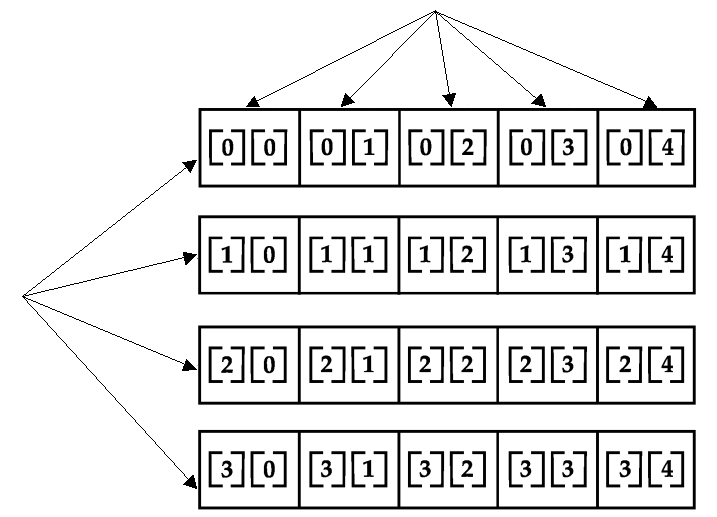
个名为 twoD 的二维数组：

int twoD[][] = new int[4][5];

这条语句分配了一个 4×5 的数组，并将之赋给 twoD。在内部，这个矩阵是作为 int 数

组的数组实现的。从概念上讲，这个数组看起来如图 3-1 所示。

右索引决定列



左索引决定行

给定：int twoD[ ] [ ] = new int [4] [5]

图 3-1 4×5 二维数组的概念视图

下面的程序按照从左向右、从上向下的顺序列出数组中的每个元素，然后显示这些元素的值：

// Demonstrate a two-dimensional array. class TwoDArray {

public static void main(String args[]) { int twoD[][]= new int[4][5];

int i, j, k = 0;

for(i=0; i<4; i++) for(j=0; j<5; j++) {

twoD[i][j] = k; k++;

}

for(i=0; i<4; i++) { for(j=0; j<5; j++)

System.out.print(twoD[i][j] + " "); System.out.println();

}

}

}

**56** 第Ⅰ部分 Java 语言

这个程序产生的输出如下所示：

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19

当为多维数组分配内存时，只需要为第一(最左边的)维分配内存。可以单独为余下的维分配内存。例如，下面的代码在声明时为 twoD 的第一维分配内存，然后手动分配第二维：

int twoD[][] = new int[4][]; twoD[0] = new int[5]; twoD[1] = new int[5]; twoD[2] = new int[5]; twoD[3] = new int[5];

虽然对于这种情况单独分配第二维没有优点，但是这对于其他情况可能有优点。例如，当手动分配维数时，不必为每一维分配相同数量的元素。正如在前面说明的，既然多维数组实际上是数组的数组，那么您可以控制每个数组的长度。例如，下面的程序创建了一个二维数组，其中第二维的长度不是相同的：

// Manually allocate differing size second dimensions. class TwoDAgain {

public static void main(String args[]) { int twoD[][] = new int[4][];

twoD[0] = new int[1]; twoD[1] = new int[2]; twoD[2] = new int[3]; twoD[3] = new int[4];

int i, j, k = 0;

for(i=0; i<4; i++) for(j=0; j<i+1; j++) { twoD[i][j] = k;

k++;

}

for(i=0; i<4; i++) { for(j=0; j<i+1; j++)

System.out.print(twoD[i][j] + " "); System.out.println();

}

}

}

这个程序产生的输出如下所示：

第3章 数据类型、变量和数组 **57**

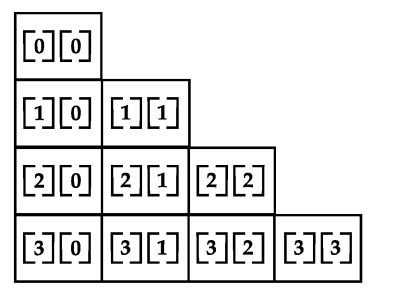
0

1 2

3 4 5

6 7 8 9

这个程序创建的数组看起来如下所示：



使用不一致(或不规则)的多维数组对于许多程序可能不合适，因为这和人们遇到多维数组时所期望的情况不同。然而，在某些情况下却可以高效地使用不规则数组。例如，如果需要一个非常大的二维稀疏数组(即只使用其中的部分元素)，那么不规则数组可能是完美的解决方案。

可以初始化多维数组。为此，只需要在一连串的花括号中包含每一维的初始化器。下面的程序创建了一个矩阵，其中的每个元素包含各自列索引和行索引的乘积。还应当注意，在数组初始化器中也可以使用表达式以及字面值。

// Initialize a two-dimensional array. class Matrix {

public static void main(String args[]) { double m[][] = {

{ 0\*0, 1\*0, 2\*0, 3\*0 }, { 0\*1, 1\*1, 2\*1, 3\*1 }, { 0\*2, 1\*2, 2\*2, 3\*2 }, { 0\*3, 1\*3, 2\*3, 3\*3 }

};

int i, j;

for(i=0; i<4; i++) { for(j=0; j<4; j++)

System.out.print(m[i][j] + " "); System.out.println();

}

}

}

当运行这个程序时，会得到如下所示的输出：

0.0 0.0 0.0 0.0

0.0 1.0 2.0 3.0

0.0 2.0 4.0 6.0

0.0 3.0 6.0 9.0

**58** 第Ⅰ部分 Java 语言

可以看出，数组中的每一行都被初始化为初始化列表中指定的值。

下面再看一个使用多维数组的例子。下面的程序创建了一个 3×4×5 的三维数组，然后将每个元素设置为各自索引的乘积，最后显示这些乘积：

// Demonstrate a three-dimensional array. class ThreeDMatrix {

public static void main(String args[]) { int threeD[][][] = new int[3][4][5]; int i, j, k;

for(i=0; i<3; i++) for(j=0; j<4; j++)

for(k=0; k<5; k++) threeD[i][j][k] = i \* j \* k;

for(i=0; i<3; i++) { for(j=0; j<4; j++) {

for(k=0; k<5; k++) System.out.print(threeD[i][j][k] + " ");

System.out.println();

}

System.out.println();

}

}

}

这个程序产生的输出如下所示：

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 1 2 3 4

0 2 4 6 8

0 3 6 9 12

0 0 0 0 0

0 2 4 6 8

0 4 8 12 16

0 6 12 18 24

3.11.3 另一种数组声明语法

还有另一种用于声明数组的形式：

type[] var-name;

第3章 数据类型、变量和数组 **59**

其中，方括号位于类型限定符之后，而不是位于数组变量名的后面。例如，下面两个声明是等价的：

int al[] = new int[3]; int[] a2 = new int[3];

下面的声明也是等价的：

char twod1[][] = new char[3][4]; char[][] twod2 = new char[3][4];

当同时声明多个数组时，这种形式可以提供便利，例如：

int[] nums, nums2, nums3; // create three arrays

这可以创建 3 个 int 类型的数组变量，上述语句和下面的声明语句是等价的：

int nums[], nums2[], nums3[]; // create three arrays

当将数组指定为方法的返回类型时，这种数组声明方法也是有用的。在本书中，这两种形式都将使用。

**3.12** **关于字符串的一些说明**

您可能已经注意到了，在前面对数据类型和数组的讨论中，没有提到字符串或字符串数据类型。这不是因为 Java 不支持这种类型，事实上 Java 支持字符串类型。Java 支持的字符串类型称为 String，它不是基本类型，也不是简单的字符数组，相反，String 定义了一个对象。全面描述 String 需要理解一些面向对象特性。为此，有关 String 的讨论将放在本书的后面，在描述完对象之后，再进行探讨。但是，既然在示例程序中要使用简单的字符串，下面的简要介绍就是必要的。

String 类型用于声明字符串变量，也可以声明字符串数组。可以将带引号的字符串常量赋给 String 变量，可以将 String 类型的变量赋给其他 String 类型的变量，甚至可以使用 String 类型的对象作为 println()方法的参数。例如，考虑下面的代码片段：

String str = "this is a test";

System.out.println(str);

在此，str 是 String 类型的对象，它被设置为字符串“this is a test”。这个字符串由 println()

语句显示。

在后面将会看到，String 对象具有许多特殊的特征和特性，这使得它们非常强大而且易于使用。然而，对于接下来的几章，将只会以最简单的形式使用它们。

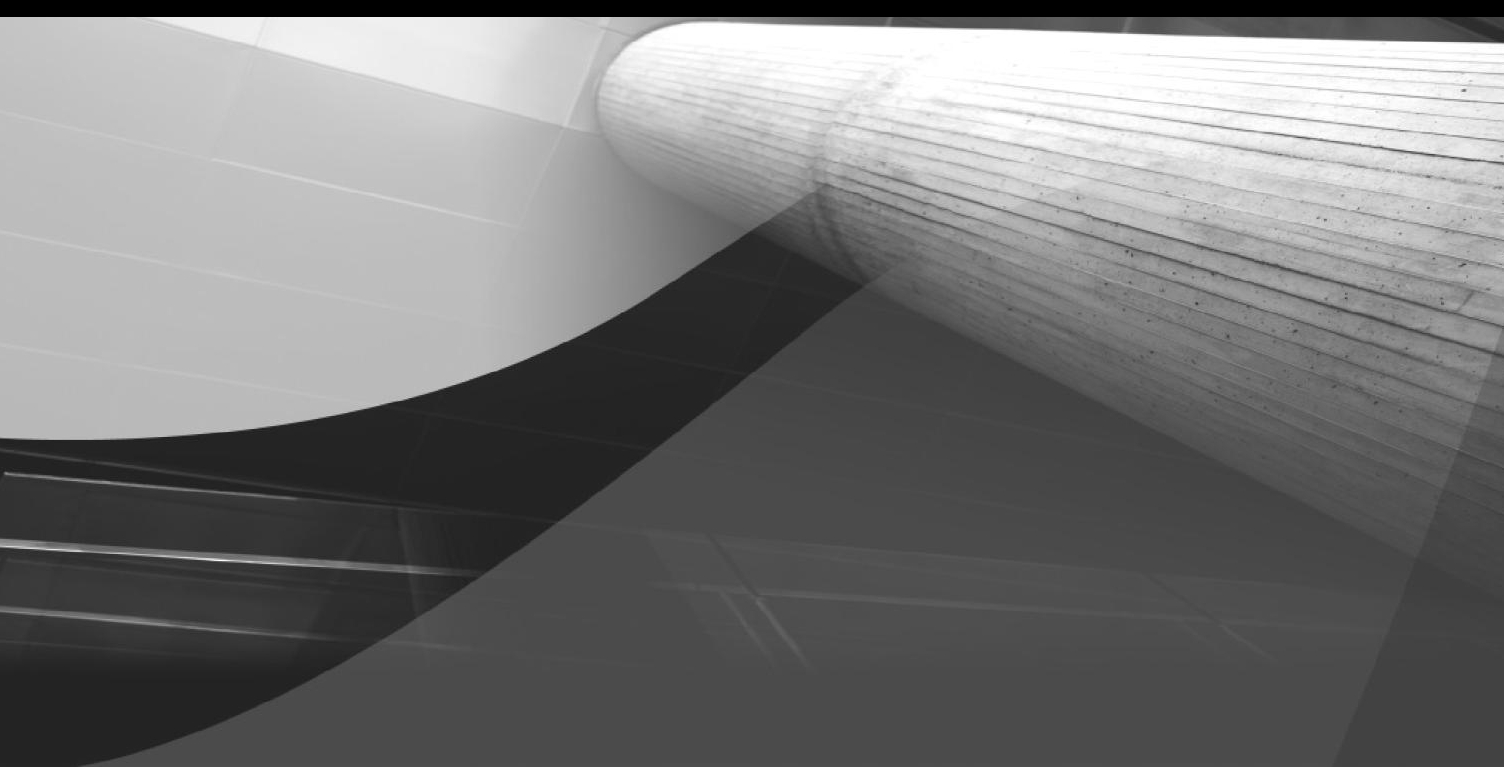
**60** 第Ⅰ部分 Java 语言

**3.13** **向 C/C++程序员提供指针方面的注解**

如果您是一位有经验的 C/C++程序员，可能知道这些语言支持指针。然而，在本章中没有提及指针。原因很简单：Java 不支持或者说不允许使用指针(更确切地说，Java 不支持程序员能够访问和修改的指针)。Java 不允许使用指针，因为如果支持指针的话，就会使 Java 程

序能够突破 Java 执行环境和宿主计算机之间的防火墙(请记住，可以将内存中的任何地址赋给指针——甚至是超出了 Java 运行时系统的地址)。既然 C/C++广泛使用指针，您可能会认为 Java 不支持指针是一个重要的缺陷。然而，情况并非如此。Java 的设计理念是：只要位于

Java 执行环境的边界以内，就永远不需要使用指针，或者即使使用指针也不会带来任何好处。



**第 12 章**

枚举、自动装箱与注解(元数据)

本章介绍 Java 语言中 3 个相对比较新的特性：枚举、自动装箱以及注解(也称为元数据)。

这 3 个特性为处理通用编程任务提供了流线型的方式，从而扩展了 Java 语言的功能。本章还将讨论 Java 的类型封装器，并介绍反射的有关知识。

**12.1** **枚举**

JDK 5 之前的 Java 版本缺失的一项特性是枚举，许多 Java 程序员都感觉需要这一特性。形式最简单的枚举(enumeration)是一系列具有名称的常量。尽管 Java 提供了其他一些能提供类似功能的特性，例如 final 变量，但是许多程序员仍然怀念枚举概念的单纯性——特别是因

1. 第Ⅰ部分 Java 语言

为大多数其他常用语言都支持枚举。从 JDK 5 开始，枚举被添加到了 Java 语言中，现在 Java 程序员可以使用它们了。

形式最简单的枚举，看起来和其他语言中的枚举类似。但是，这种相似性只是表面上的。在 C++这类语言中，枚举只不过是一系列具有名称的整型常量。在 Java 中，枚举定义了一种类类型。通过将枚举定义为类，极大地扩展了枚举的概念。例如在 Java 中，枚举可以具有构造函数、方法以及实例变量。所以，尽管枚举的打造历时多年，但是 Java 对枚举的丰富实现使得漫长的等待是值得的。

12.1.1 枚举的基础知识

创建枚举需要使用关键字 enum。例如，下面是一个简单的枚举，其中列出了各种苹果的品种：

// An enumeration of apple varieties. enum Apple {

Jonathan, GoldenDel, RedDel, Winesap, Cortland

}

标识符 Jonathan、GoldenDel 等被称为枚举常量。每个枚举常量被隐式声明为 Apple 的公有、静态 final 成员。此外，枚举常量的类型是声明它们的枚举的类型，对于这个例子为 Apple。因此在 Java 语言中，这些常量被称为是“自类型化的”(self-typed)，其中的“自”是指封装常量的枚举。

定义了枚举之后，可以创建枚举类型的变量。但是，尽管枚举定义了类类型，却不能使用 new 实例化枚举。反而，枚举变量的使用方式在许多方面与基本类型相同。例如，下面这行代码将 ap 声明为 Apple 枚举类型的变量：

Apple ap;

因为 ap 是 Apple 类型，所以只能被赋值为(或包含)在 Apple 枚举中定义的那些值。例如，下面这行代码将 ap 赋值为 RedDel：

ap = Apple.RedDel;

注意在符号 RedDel 之前的 Apple。

可以使用关系运算符“==”比较两个枚举常量的相等性。例如，下面这条语句比较 ap

的值和 GoldenDel 常量：

if(ap == Apple.GoldenDel) // ...

枚举值也可以用于控制 switch 语句。当然，所有 case 语句使用的常量的枚举类型，都必须与 switch 表达式使用的枚举类型相同。例如，下面这条 switch 语句是完全合法的：

第12章 枚举、自动装箱与注解(元数据) **273**

// Use an enum to control a switch statement. switch(ap) {

case Jonathan: // ...

case Winesap: // ...

注意在 case 语句中，枚举常量的名称没有使用枚举类型的名称进行限定。也就是说，使用的是 Winesap 而不是 Apple.Winesap。这是因为 switch 表达式中的枚举类型已经隐式指定了 case 常量的枚举类型。所以在 case 语句中不需要使用枚举类型的名称对常量进行限定。实际

上，如果试图这么做的话，会造成编译时错误。

当显示枚举常量时，例如在 println()语句中，会输出枚举常量的名称。例如下面这条语句：

System.out.println(Apple.Winesap);

会显示名称“Winesap”。

下面的程序用到了刚才介绍的所有内容，并演示了 Apple 枚举：

// An enumeration of apple varieties. enum Apple {

Jonathan, GoldenDel, RedDel, Winesap, Cortland

}

class EnumDemo {

public static void main(String args[])

{

Apple ap;

ap = Apple.RedDel;

// Output an enum value. System.out.println("Value of ap: " + ap); System.out.println();

ap = Apple.GoldenDel;

* Compare two enum values. if(ap == Apple.GoldenDel)

System.out.println("ap contains GoldenDel.\n");

* Use an enum to control a switch statement. switch(ap) {

case Jonathan: System.out.println("Jonathan is red."); break;

case GoldenDel:

System.out.println("Golden Delicious is yellow.");

**274** 第Ⅰ部分 Java 语言

break; case RedDel:

System.out.println("Red Delicious is red."); break;

case Winesap: System.out.println("Winesap is red."); break;

case Cortland: System.out.println("Cortland is red."); break;

}

}

}

该程序的输出如下所示：

Value of ap: RedDel

ap contains GoldenDel.

Golden Delicious is yellow.

12.1.2 values()和 valueOf()方法

所有枚举都自动包含两个预定义方法：values()和 valueOf()。它们的一般形式如下所示：

public static enum-type [ ] values( ) public static enum-type valueOf(String str )

values()方法返回一个包含枚举常量列表的数组，valueOf()方法返回与传递到参数 str 的

字符串相对应的枚举常量。对于这两个方法，enum-type 是枚举类型。例如，对于前面显示

的 Apple 枚举，Apple.valueOf("Winesap")的返回类型是 Apple。下面的程序演示了 values()和 valueOf()方法：

* Use the built-in enumeration methods.
* An enumeration of apple varieties. enum Apple {

Jonathan, GoldenDel, RedDel, Winesap, Cortland

}

class EnumDemo2 {

public static void main(String args[])

{

Apple ap;

System.out.println("Here are all Apple constants:");

第12章 枚举、自动装箱与注解(元数据) **275**

// use values()

Apple allapples[] = Apple.values(); for(Apple a : allapples)

System.out.println(a);

System.out.println();

// use valueOf()

ap = Apple.valueOf("Winesap"); System.out.println("ap contains " + ap);

}

}

该程序的输出如下所示：

Here are all Apple constants:

Jonathan

GoldenDel

RedDel

Winesap

Cortland

ap contains Winesap

注意这个程序使用 for-each 风格的 for 循环来遍历 values()方法返回的常量数组。为了进行说明，创建变量 allapples，并将其赋值为对枚举数组的引用。但是，这个步骤不是必需的，因为可以像下面这样编写 for 循环，从而消除对 allapples 变量的需要：

for(Apple a : Apple.values()) System.out.println(a);

现在，请注意通过调用 valueOf()方法获取与名称 Winesap 对应的枚举值的方式：

ap = Apple.valueOf("Winesap");

正如前面所解释的，valueOf()方法返回与以字符串形式表示的常量名称相关联的枚举值。

注意：



C/C++程序员会注意到，相对于其他语言，Java 极大地简化了枚举在人类能够阅读的常量形式和其二进制数值形式之间的转换。这是 Java 枚举实现方式的一个重要优点。

12.1.3 Java 枚举是类类型

正如前面所解释的，Java 枚举是类类型。虽然不能使用 new 实例化枚举，但是枚举却有许多和其他类相同的功能。实际上，枚举定义了为 Java 枚举提供功能的类，这些功能是其他

1. 第Ⅰ部分 Java 语言

语言中的枚举所不具备的。例如，可以为枚举提供构造函数、添加实例变量和方法，甚至可以实现接口。

需要理解的重要一点是：每个枚举常量都是所属枚举类型的对象。因此，如果为枚举定义了构造函数，那么当创建每个枚举常量时都会调用该构造函数。此外，对于枚举定义的实例变量，每个枚举常量都有它自己的副本。例如，分析下面版本的 Apple 枚举：

// Use an enum constructor, instance variable, and method. enum Apple {

Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

private int price; // price of each apple

// Constructor

Apple(int p) { price = p; }

int getPrice() { return price; }

}

class EnumDemo3 {

public static void main(String args[])

{

Apple ap;

* Display price of Winesap. System.out.println("Winesap costs " +

Apple.Winesap.getPrice() +

* + cents.\n");
* Display all apples and prices. System.out.println("All apple prices:"); for(Apple a : Apple.values())

System.out.println(a + " costs " + a.getPrice() +

* + - cents.");

}

}

输出如下所示：

Winesap costs 15 cents.

All apple prices:

Jonathan costs 10 cents.

GoldenDel costs 9 cents.

RedDel costs 12 cents.

Winesap costs 15 cents.

Cortland costs 8 cents.

这个版本的 Apple 枚举添加了 3 个内容：第 1 个内容是实例变量 price，用于保存每种苹

第12章 枚举、自动装箱与注解(元数据) **277**

果的价格；第 2 个内容是 Apple 构造函数，用于传递苹果的价格，第 3 个内容是 getPrice()方

法，用于返回 price 变量的值。

当在 main()方法中声明变量 ap 时，对于每个特定的常量调用 Apple 的构造函数一次。注意指定构造函数参数的方式，通过将它们放置到每个常量后面的圆括号中来加以指定，如下所示：

Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

这些数值被传递给 Apple()方法的参数 p，然后将这些值赋给 price 变量。再强调一次，为每个常量调用构造函数一次。

因为每个枚举常量都有自己的 price 变量副本，所以可以调用 getPrice()方法来获取指定类型苹果的价格。例如在 main()方法中，通过下面的调用获取 Winesap 的价格：

Apple.Winesap.getPrice( )

通过 for 循环遍历枚举可以获取所有品种的苹果的价格。因为每个枚举常量都有 price 变

量的副本，所以与枚举常量关联的值是独立的，并且和与其他常量关联的值不同。这是一个强大的功能，只有将枚举作为类实现，像 Java 这样，才会具有这种功能。

尽管前面的例子只包含一个构造函数，但是枚举可以提供两种甚至更多种重载形式，就像其他类那样。例如，下面版本的 Apple 提供了一个默认构造函数，可以将 price 变量初始化为−1，表明不能获得价格数据：

// Use an enum constructor. enum Apple {

Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);

private int price; // price of each apple

// Constructor

Apple(int p) { price = p; }

// Overloaded constructor Apple() { price = -1; }

int getPrice() { return price; }

}

注意在这个版本中，没有为 RedDel 提供参数。这意味着会调用默认构造函数，并将 RedDel 的 price 变量设置为−1。

枚举有两条限制：第一，枚举不能继承其他类；第二，枚举不能是超类。这意味着枚举不能扩展。在其他方面，枚举和其他类很相似。需要记住的关键是：每个枚举常量都是定义它的类的对象。

**278** 第Ⅰ部分 Java 语言

12.1.4 枚举继承自 Enum 类

尽管声明枚举时不能继承超类，但是所有枚举都自动继承超类 java.lang.Enum，这个类定义了所有枚举都可以使用的一些方法。Enum 类将在本书第Ⅱ部分详细介绍，但是在此先讨论它的 3 个方法。

可以获取用于指示枚举常量在常量列表中位置的值，这称为枚举常量的原始值。通过ordinal()方法可以检索原始值，该方法的声明如下所示：

final int ordinal( )

该方法返回所调用常量的原始值，原始值从 0 开始。因此在 Apple 枚举中，Jonathan 的原始值为 0，GoldenDel 的原始值为 1，RedDel 的原始值为 2，等等。

可以使用 compareTo()方法比较相同类型的两个枚举常量的原始值，该方法的一般形式如下：

final int compareTo(enum-type e)

其中，enum-type 是枚举的类型，e 是和调用常量进行比较的常量。请记住，调用常量和 e 必须是相同的枚举。如果调用常量的原始值小于 e 的原始值，那么 compareTo()方法返回负值；如果两个原始值相同，就返回 0；如果调用常量的原始值大于 e 的原始值，就返回正值。

可以使用 equals()方法来比较枚举常量和其他对象的相等性，该方法重写了 Object 类定义的 equals()方法。尽管 equals()方法可以将枚举常量和任意其他对象进行比较，但是只有当两个对象都引用同一个枚举中相同的常量时，它们才相等。如果两个常量来自不同的枚举，那么即使它们的原始值相同，equals()方法也不会返回 true。

请记住，可以使用“==”比较两个枚举引用。

下面的程序演示了 ordinal()、compareTo()以及 equals()方法：

* Demonstrate ordinal(), compareTo(), and equals().
* An enumeration of apple varieties.

enum Apple {

Jonathan, GoldenDel, RedDel, Winesap, Cortland

}

class EnumDemo4 {

public static void main(String args[])

{

Apple ap, ap2, ap3;

// Obtain all ordinal values using ordinal(). System.out.println("Here are all apple constants" +

" and their ordinal values: "); for(Apple a : Apple.values())

System.out.println(a + " " + a.ordinal());

第12章 枚举、自动装箱与注解(元数据) **279**

ap = Apple.RedDel; ap2 = Apple.GoldenDel; ap3 = Apple.RedDel;

System.out.println();

// Demonstrate compareTo() and equals() if(ap.compareTo(ap2) < 0)

System.out.println(ap + " comes before " + ap2);

if(ap.compareTo(ap2) > 0)

System.out.println(ap2 + " comes before " + ap);

if(ap.compareTo(ap3) == 0) System.out.println(ap + " equals " + ap3);

System.out.println();

if(ap.equals(ap2))

System.out.println("Error!");

if(ap.equals(ap3))

System.out.println(ap + " equals " + ap3);

if(ap == ap3)

System.out.println(ap + " == " + ap3);

}

}

该程序的输出如下所示：

Here are all apple constants and their ordinal values:

Jonathan 0

GoldenDel 1

RedDel 2

Winesap 3

Cortland 4

GoldenDel comes before RedDel

RedDel equals RedDel

RedDel equals RedDel

RedDel == RedDel

12.1.5 另一个枚举示例

在继续学习之前，下面再分析一个使用枚举的例子。在第 9 章，创建了一个自动的“决策生成器”程序。那个版本在一个接口中声明了变量 No、Yes、Later、Soon 以及 Never，并

1. 第Ⅰ部分 Java 语言

使用这些变量代表可能的答案。虽然从技术上讲这种方式没有问题，但是枚举是更好的选择。下面是那个程序的改进版，该版本使用 Answers 枚举定义答案。应当将该版本和第 9 章中的原始版本进行比较。

* + An improved version of the "Decision Maker"
  + program from Chapter 9. This version uses an
  + enum, rather than interface variables, to
  + represent the answers.

import java.util.Random;

// An enumeration of the possible answers. enum Answers {

NO, YES, MAYBE, LATER, SOON, NEVER

}

class Question {

Random rand = new Random(); Answers ask() {

int prob = (int) (100 \* rand.nextDouble());

if (prob < 15)

return Answers.MAYBE; // 15% else if (prob < 30)

return Answers.NO; // 15% else if (prob < 60)

return Answers.YES; // 30% else if (prob < 75)

return Answers.LATER; // 15% else if (prob < 98)

return Answers.SOON; // 13% else

return Answers.NEVER; // 2%

}

}

class AskMe {

static void answer(Answers result) { switch(result) {

case NO: System.out.println("No"); break;

case YES: System.out.println("Yes"); break;

case MAYBE: System.out.println("Maybe"); break;

case LATER:

第12章 枚举、自动装箱与注解(元数据) **281**

System.out.println("Later");

break; case SOON:

System.out.println("Soon");

break; case NEVER:

System.out.println("Never");

break;

}

}

public static void main(String args[]) { Question q = new Question(); answer(q.ask());

answer(q.ask());

answer(q.ask());

answer(q.ask());

}

}

**12.2** **类型封装器**

您已经知道，Java 使用基本类型(也称为简单类型)，比如 int 或 double，来保存语言支持的基本数据类型。出于性能考虑，为这些数据使用基本类型而不是对象。为这些数据使用对象，即使是最简单的计算也会增加不可接受的负担。因此，基本类型不是对象层次的组成部分，它们不继承 Object 类。

虽然基本类型提供了性能方面的好处，但有时会需要对象这种表示形式。例如，不能通过引用为方法传递基本类型。此外，Java 使用的许多标准数据结构是针对对象进行操作的，这意味着不能使用这些结构存储基本类型。为了处理这些(以及其他)情况，Java 提供了类型封装器，用来将基本类型封装到对象中。类型封装器是类，将在本书第Ⅱ部分详细分析，但是在此先对其进行简要介绍，因为它们与 Java 的自动装箱特性直接相关。

类型封装器包括 Double、Float、Long、Integer、Short、Byte、Character 以及 Boolean。

这些类提供了大量的方法，通过这些方法可以完全将基本类型集成到 Java 的对象层次中。下面对这些封装器逐一进行简要介绍。

12.2.1 Character 封装器

Character 是 char 类型的封装器。Character 的构造函数为：

Character(char ch)

其中，ch 指定了由即将创建的 Character 对象封装的字符。

为了获取 Character 对象中的 char 值，可以调用 charValue()方法，如下所示：

**282** 第Ⅰ部分 Java 语言

char charValue( )

该方法返回封装的字符。

12.2.2 Boolean 封装器

Boolean 是用来封装布尔值的封装器，其中定义了以下构造函数：

Boolean(boolean boolValue)

Boolean(String boolString)

在第一个版本中，boolValue 必须是 true 或 false。在第二个版本中，如果 boolString 包含

字符串“true”(大写或小写形式都可以)，那么新的 Boolean 对象将为 true，否则将为 false。为了从 Boolean 对象获取布尔值，可以使用 booleanValue()方法，如下所示：

boolean booleanValue( )

该方法返回与调用对象等价的布尔值。

12.2.3 数值类型封装器

到目前为止，最常用的封装器是那些表示数值的封装器，包括 Byte、Short、Integer、Long、 Float 以及 Double。所有这些数值类型封装器都继承自抽象类 Number。Number 声明了以不

同数字格式从对象返回数值的方法，如下所示：

byte byteValue( ) double doubleValue( ) float floatValue( ) int intValue( )

long longValue( ) short shortValue( )

例如，doubleValue()方法返回 double 类型的值， floatValue()方法返回 float 类型的值，等等。每个数值类型封装器都实现了这些方法。

所有数值类型封装器都定义了用于从给定数值或数值的字符串表示形式构造对象的构造函数。例如，下面是为 Integer 定义的构造函数：

Integer(int num)

Integer(String str)

如果 str 没有包含有效的数值，就会抛出 NumberFormatException 异常。

所有类型封装器都重写了 toString()方法，用来返回封装器所包含数值的人类可以阅读的形式，从而可以通过将封装器对象传递给 println()方法来输出数值。例如，不必将之转换为基本类型。

下面的程序演示了如何使用数值类型的封装器封装数值以及如何提取数值：

第12章 枚举、自动装箱与注解(元数据) **283**

// Demonstrate a type wrapper. class Wrap {

public static void main(String args[]) {

Integer iOb = new Integer(100);

int i = iOb.intValue();

System.out.println(i + " " + iOb); // displays 100 100

}

}

这个程序将整型值 100 封装到 Integer 对象 iOb 中，然后程序调用 intValue()方法以获取这个数值，并将结果存储到 i 中。

将数值封装到对象中的过程称为装箱。因此在这个程序中，下面这行代码将数值 100 装箱到一个 Integer 对象中：

Integer iOb = new Integer(100);

从类型封装器中抽取数值的过程称为拆箱。例如，这个程序使用下面这条语句拆箱 iOb

中的数值：

int i = iOb.intValue();

上述程序使用的装箱和拆箱数值的一般过程，在 Java 原始版本中就已经提供了。但是，随着 JDK 5 的发布，Java 通过自动装箱从根本上改进了这一过程，下面将对此进行介绍。

1. **自动装箱**
   * JDK 5 开始，Java 增加了两个重要特性：自动装箱和自动拆箱。自动装箱是这样一个过程：无论何时，只要需要基本类型的对象，就自动将基本类型自动封装(装箱)到与之等价的类型封装器中，而不需要显式地构造对象。自动拆箱是当需要时自动抽取(拆箱)已装箱对象的数值的过程。不需要调用 intValue()或 doubleValue()这类方法。

自动装箱和自动拆箱特性极大地简化了一些算法的编码，移除了单调乏味的手动装箱和拆箱数值操作。它们还有助于防止错误发生。此外，它们对于泛型非常重要，因为泛型只能操作对象。最后，集合框架(将在本书第Ⅱ部分介绍)需要利用自动装箱特性进行工作。

有了自动装箱特性，封装基本类型将不必再手动创建对象。只需要将数值赋给类型封装器引用即可，Java 会自动创建对象。例如，下面是构造具有数值 100 的 Integer 对象的现代方式：

Integer iOb = 100; // autobox an int

注意没有使用 new 显式地创建对象。Java 自动处理了这个过程。

**284** 第Ⅰ部分 Java 语言

为了拆箱对象，可以简单地将对象引用赋值给基本类型的变量。例如，为了拆箱 iOb，

可以使用下面这行代码：

int i = iOb; // auto-unbox

Java 处理了这一个过程中的细节。

下面的程序对前面的程序进行了改写，以使用自动装箱和自动拆箱特性：

// Demonstrate autoboxing/unboxing. class AutoBox {

public static void main(String args[]) {

Integer iOb = 100; // autobox an int

int i = iOb; // auto-unbox

System.out.println(i + " " + iOb); // displays 100 100

}

}

12.3.1 自动装箱与方法

除了赋值这种简单情况之外，无论何时，如果必须将基本类型转换为对象，就会发生自动装箱；无论何时，如果对象必须转换为基本类型，就会发生自动拆箱。因此，当向方法传递参数或者从方法返回数值时，都可能会发生自动装箱/拆箱。例如，分析下面的程序：

* Autoboxing/unboxing takes place with
* method parameters and return values.

class AutoBox2 {

* Take an Integer parameter and return
* an int value;

static int m(Integer v) {

return v ; // auto-unbox to int

}

public static void main(String args[]) {

* Pass an int to m() and assign the return value
* to an Integer. Here, the argument 100 is autoboxed
* into an Integer. The return value is also autoboxed
* into an Integer.

Integer iOb = m(100);

System.out.println(iOb);

}

}

这个程序显示的结果如下所示：

第12章 枚举、自动装箱与注解(元数据) **285**

100

在这个程序中，请注意 m()方法指定了一个 Integer 类型的参数并返回 int 型结果。在 main() 方法中，为 m()方法传递的数值是 100。因为 m()方法期望传递过来的是 Integer 对象，所以对这个数值进行自动装箱。之后，m()方法返回与其参数等价的 int 型数值，这会导致对 v 进行自动拆箱。接下来，在 main()方法中将 int 型数值赋给 iOb，这会导致对返回的 int 型数值进行自动装箱。

12.3.2 表达式中发生的自动装箱/拆箱

通常无论何时，当需要将基本类型转换为对象或者将对象转换为基本类型时，就会发生自动装箱和自动拆箱。对于表达式也是如此。在表达式中，数值对象会被自动拆箱。如果需要的话，还可以对表达式的输出进行重新装箱。例如，分析下面的程序：

// Autoboxing/unboxing occurs inside expressions. class AutoBox3 {

public static void main(String args[]) {

Integer iOb, iOb2; int i;

iOb = 100;

System.out.println("Original value of iOb: " + iOb);

* The following automatically unboxes iOb,
* performs the increment, and then reboxes
* the result back into iOb.

++iOb;

System.out.println("After ++iOb: " + iOb);

* Here, iOb is unboxed, the expression is
* evaluated, and the result is reboxed and
* assigned to iOb2.

iOb2 = iOb + (iOb / 3);

System.out.println("iOb2 after expression: " + iOb2);

* The same expression is evaluated, but the
* result is not reboxed.

i = iOb + (iOb / 3);

System.out.println("i after expression: " + i);

}

}

输出如下所示：

Original value of iOb: 100

After ++iOb: 101

**286** 第Ⅰ部分 Java 语言

iOb2 after expression: 134

i after expression: 134

在这个程序中，应特别注意下面这行代码：

++iOb;

这会导致 iOb 中的值递增。具体工作过程如下：将 iOb 自动拆箱，将值递增，然后将结果自动装箱。

自动拆箱还允许在表达式中混合不同数值类型的对象。一旦数值被拆箱，就会应用标准的类型提升和转换。例如，下面的程序是完全合法的：

class AutoBox4 {

public static void main(String args[]) {

Integer iOb = 100;

Double dOb = 98.6;

dOb = dOb + iOb;

System.out.println("dOb after expression: " + dOb);

}

}

输出如下所示：

dOb after expression: 198.6

可以看出，Double 对象 dOb 和 Integer 对象 iOb 都参与了加法运算，对结果再次装箱并存储在 dOb 中。

因为提供了自动拆箱特性，所以可以使用 Integer 数值对象控制 switch 语句。例如，分析下面的代码段：

Integer iOb = 2; switch(iOb) {

case 1: System.out.println("one"); break;

case 2: System.out.println("two"); break;

default: System.out.println("error");

}

当对 switch 表达式进行求值时，iOb 被拆箱，从而得到其中存储的 int 型数值。

正如程序中的例子所显示的，因为提供了自动装箱/拆箱特性，在表达式中使用数值类型对象不仅很直观而且很容易。在过去，这种代码需要涉及强制类型转换，并且需要调用

intValue()这类方法。

第12章 枚举、自动装箱与注解(元数据) **287**

12.3.3 布尔型和字符型数值的自动装箱/拆箱

如前所述，Java 也为 boolean 和 char 类型提供了封装器，它们是 Boolean 和 Character。

这些封装器也应用自动装箱/拆箱特性。例如，分析下面的程序：

// Autoboxing/unboxing a Boolean and Character.

class AutoBox5 {

public static void main(String args[]) {

* Autobox/unbox a boolean. Boolean b = true;
* Below, b is auto-unboxed when used in
* a conditional expression, such as an if. if(b) System.out.println("b is true");
* Autobox/unbox a char.

Character ch = 'x'; // box a char char ch2 = ch; // unbox a char

System.out.println("ch2 is " + ch2);

}

}

输出如下所示：

b is true ch2 is x

对于这个程序，需要注意的最重要地方是：在 if 条件表达式中对 b 进行自动拆箱。您应该记得，控制 if 的条件表达式的求值结果必须是 boolean 类型。因为有了自动拆箱特性，当对条件表达式进行求值时，b 中的布尔值被自动拆箱。因此，因为提供了自动装箱/拆箱特性，所以在 if 语句中可以使用 Boolean 对象。

正是因为有了自动拆箱特性，现在也可以使用 Boolean 对象控制所有循环语句。当将 Boolean 用作 while、for 或 do/while 的条件表达式时，会自动拆箱为它的布尔等价形式。例如，

现在下面的代码是完全合法的：

Boolean b;

// ...

while(b) { // ...

12.3.4 自动装箱/拆箱有助于防止错误

除了可以提供便利之外，自动装箱/拆箱还有助于防止错误。例如，分析下面的代码：

**288** 第Ⅰ部分 Java 语言

// An error produced by manual unboxing. class UnboxingError {

public static void main(String args[]) {

Integer iOb = 1000; // autobox the value 1000

int i = iOb.byteValue(); // manually unbox as byte !!!

System.out.println(i); // does not display 1000 !

}

}

该程序不会显示期望的数值 1000，而会显示−24！原因是：通过 byteValue()方法对 iOb

中的值进行手动拆箱，会导致存储在 iOb 中的值(在本例中是 1000)被截断。结果是将垃圾值

−24 赋给 i。自动拆箱可以防止这种类型的错误，因为 iOb 中的值会总是会被拆箱为与 int 类

型兼容的值。

通常，因为自动装箱总是会创建正确的对象，并且自动拆箱总是会产生正确的数值，所

以不会产生错误类型的对象或数值。在极端情况下，如果您所期望的类型和自动装箱/拆箱生

成的类型不同的话，仍然可以对数值进行手动装箱/拆箱。当然，这会丢失自动装箱/拆箱带

来的好处。通常，新代码应当使用自动装箱/拆箱。这是编写现代 Java 代码的方式。

12.3.5 一些警告

既然 Java 提供了自动装箱/拆箱特性，有些程序员可能会专门使用 Integer 或 Double，而

完全放弃基本类型。例如，可能使用自动装箱/拆箱编写类似下面的代码：

// A bad use of autoboxing/unboxing! Double a, b, c;

a = 10.0; b = 4.0;

c = Math.sqrt(a\*a + b\*b); System.out.println("Hypotenuse is " + c);

在这个例子中，使用 Double 类型的对象保存用于计算直角三角形斜边的值。尽管这段代

码从技术上讲是正确的，并且可以工作，实际工作得很好，但这是对自动装箱/拆箱的滥用。

与使用基本类型 double 编写的等价代码相比，上面代码的效率低很多。原因是每次进行自动

装箱/拆箱都会增加开销，而使用基本类型不需要这些开销。

通常，应当限制类型封装器的使用，只有当需要基本类型的对象表示形式时才应当使用。

提供的自动装箱/拆箱特性不是用来作为消除基本类型的“后门”。

第12章 枚举、自动装箱与注解(元数据) **289**

1. **注解(元数据)**
   * JDK 5 开始，Java 支持在源文件中嵌入补充信息，这类信息称为注解(annotation)。注解不会改变程序的动作，因此也就不会改变程序的语义。但是在开发和部署期间，各种工具可以使用这类信息。例如，源代码生成器可以处理注解。术语“元数据”(metadata)也用于表

示这个特性，但是术语“注解”更具描述性并且更常用。

12.4.1 注解的基础知识

注解是通过基于接口的机制创建的。首先看一个例子。下面的代码声明了注解 MyAnno：

// A simple annotation type. @interface MyAnno {

String str(); int val();

}

首先，注意关键字 interface 前面的@，这告诉编译器正在声明一种注解类型。接下来，注意两个成员 str()和 val()。所有注解都只包含方法声明。但是，不能为方法提供方法体，而是由 Java 实现这些方法。此外，正如后面即将看到的，这些方法的行为更像是域变量。

注解不能包含 extends 子句。但是，所有注解类型都自动扩展了 Annotation 接口。因此，

Annotation 是所有注解的超接口。该接口是在 java.lang.annotation 包中声明的，其中重写了hashCode()、equals()以及 toString()方法，这些方法是由 Object 类定义的。另外还指定了annotationType()方法，该方法返回表示所调用注解的 Class 对象。

在声明注解之后，就可以用来注解声明了。所有类型的声明都可以有一个与之关联的注解。例如，类、方法、域变量、参数以及枚举常量都可以带有注解。甚至注解本身也可以被注解。对于所有情况，注解都要放在声明的最前面。

应用注解时，需要为注解的成员提供值。例如，下面的例子将 MyAnno 应用到某个方法声明中：

// Annotate a method.

@MyAnno(str = "Annotation Example", val = 100) public static void myMeth() { // ...

这个注解被链接到方法 myMeth()。下面进一步分析注解的语法。注解的名称以@作为前缀，后面跟一个位于圆括号中的成员初始化列表。为了给成员提供值，需要为成员的名称赋

予值。所以在这个例子中，将字符串“Annotation Example”赋给 MyAnno 的 str 成员。注意

在这条赋值语句中，在 str 之后没有圆括号。当为注解成员提供数值时，只使用成员的名称。因此，在这个上下文中，注解成员看起来像域变量。

12.4.2 指定保留策略

在进一步介绍注解之前，有必要讨论一下注解保留策略。保留策略决定了在什么位置丢

1. 第Ⅰ部分 Java 语言

弃注解。Java 定义了 3 种策略，它们被封装到 java.lang.annotation.RetentionPolicy 枚举中。这 3 种策略分别是 SOURCE、CLASS 和 RUNTIME。

* + 使用 SOURCE 保留策略的注解，只在源文件中保留，在编译期间会被抛弃。
  + 使用 CLASS 保留策略的注解，在编译时被存储到.class 文件中。但是，在运行时通过 JVM 不能得到这些注解。
  + 使用 RUNTIME 保留策略的注解，在编译时被存储到.class 文件中，并且在运行时可以通过 JVM 获取这些注解。因此，RUNTIME 保留策略提供了最永久的注解。

注意：



局部变量的注解不能存储在.class 文件中。

保留策略是通过 Java 的内置注解——@Retention 指定的，它的一般形式如下所示：

@Retention(retention-policy)

其中，retention-policy 必须是上面讨论的枚举常量之一。如果没有为注解指定保留策略，将使用默认策略 CLASS。

下面版本的 MyAnno 使用@Retention 指定了 RUNTIME 保留策略。因此，在程序执行期

间通过 JVM 可以获取 MyAnno。

@Retention(RetentionPolicy.RUNTIME) @interface MyAnno {

String str(); int val();

}

12.4.3 在运行时使用反射获取注解

尽管设计注解的目的主要是用于其他的开发和部署工具，但是如果为注解指定

RUNTIME 保留策略，那么任何程序在运行时都可以使用反射来查询注解。反射是能够在运行时获取类相关信息的特性。反射 API 位于 java.lang.reflect 包中。使用反射的方式有很多，在此不可能解释所有这些方式。但是，我们将分析应用了注解的几个例子。

使用反射的第一步是获取 Class 对象，表示希望获取其中注解的类。Class 是 Java 的内置类，是在 java.lang 包中定义的。在本书第Ⅱ部分将对这个包进行详细介绍。可以使用多种方式来获取 Class 对象。其中最简单的方式是调用 getClass()方法，该方法是由 Object 类定义的，它的一般形式如下所示：

final Class<?> getClass( )

该方法返回用来表示调用对象的 Class 对象。

第12章 枚举、自动装箱与注解(元数据) **291**

注意：



注意上面显示的 getClass()方法声明中跟在 Class 后面的<?>，这与 Java 中的泛型特性有关。在本章讨论的 getClass()方法以及其他几个与反射有关的方法，需要使用泛型。泛型将在第 14 章介绍。但是，理解反射的基本原则不需要先理解泛型。

获得 Class 对象后，可以使用其他方法获取与类声明中各个条目相关的信息，包括注解。如果希望获取与类声明中特定条目关联的注解，那么首先必须获取表示该特定条目的对象。

例如，Class 提供了 getMethod()、getField()以及 getConstructor()方法(还有其他方法)，这些方法分别获取与方法、域变量以及构造函数相关的信息，这些方法返回 Method、Field 以及

Constructor 类型的对象。

为了理解这个过程，分析一个获取与方法关联的注解的例子。为此，首先获取表示类的

Class 对象，然后调用 Class 对象的 getMethod()方法并指定方法的名称。getMethod()方法的一

般形式如下：

Method getMethod(String methName, Class<?> ... paramTypes)

方法的名称被传递到 methName 中。如果方法有参数，那么必须通过 paramTypes 指定表示这些参数类型的 Class 对象。注意 paramTypes 是可变长度参数，这意味着可以指定需要的任意多个参数，包括指定 0 个参数。getMethod()方法返回表示方法的 Method 对象。如果没有找到方法，就抛出 NoSuchMethodException 异常。

对 Class、Method、Field 以及 Constructor 对象调用 getAnnotation()方法，可以获得与对象关联的特定信息。该方法的一般形式如下：

<A extends Annotation> getAnnotation(Class<A> annoType)

其中，annoType 是表示您感兴趣注解的 Class 对象。该方法返回对注解的一个引用，使用这个引用可以获取与注解成员关联的值。如果没有找到注解，该方法会返回 null。如果注解的保留策略不是 RUNTIME，就会出现这种情况。

下面的程序总结了在前面介绍的所有内容，并使用反射显示与某个方法关联的注解：

import java.lang.annotation.\*; import java.lang.reflect.\*;

// An annotation type declaration. @Retention(RetentionPolicy.RUNTIME) @interface MyAnno {

String str(); int val();

}

class Meta {

1. 第Ⅰ部分 Java 语言
   * Annotate a method.

@MyAnno(str = "Annotation Example", val = 100) public static void myMeth() {

Meta ob = new Meta();

* Obtain the annotation for this method
* and display the values of the members. try {
  + First, get a Class object that represents
  + this class.

Class<?> c = ob.getClass();

* Now, get a Method object that represents
* this method.

Method m = c.getMethod("myMeth");

* + Next, get the annotation for this class. MyAnno anno = m.getAnnotation(MyAnno.class);
  + Finally, display the values. System.out.println(anno.str() + " " + anno.val());

} catch (NoSuchMethodException exc) { System.out.println("Method Not Found.");

}

}

public static void main(String args[]) { myMeth();

}

}

该程序的输出如下所示：

Annotation Example 100

这个程序使用前面介绍的反射，获取并显示与 Meta 类中 myMeth()方法关联的 MyAnno

注解中 str 和 val 的值。有两点需要特别注意。第一点，注意下面这行代码中的表达式

MyAnno.class：

MyAnno anno = m.getAnnotation(MyAnno.class);

对这个表达式求值的结果是表示 MyAnno 类型的 Class 对象，即注解。这种结构被称为

“类字面值”。无论何时，当需要已知类的 Class 对象时，就可以使用这类表达式。例如，可以使用下面这条语句获取 Meta 的 Class 对象：

Class<?> c = Meta.class;

第12章 枚举、自动装箱与注解(元数据) **293**

当然，只有当事先知道对象的类名时才能使用这种方式，但我们并不总是知道对象的类名。通常，可以获取类、接口、基本类型以及数组的类字面值(记住，<?>语法与 Java 的泛型特性有关，泛型将在第 14 章介绍)。

需要注意的第二点是，当通过下面这行代码进行输出时，如何获取与 str 和 val 关联的数值：

System.out.println(anno.str() + " " + anno.val());

注意这里使用方法调用语法来调用它们。当需要注解成员的值时，可以使用相同的方式。

1. 第二个反射示例

在前面的例子中，myMeth()方法没有参数。因此，当调用 getMethod()方法时，只传递名称 myMeth。但是，为了获取带有参数的方法，必须指定表示参数类型的类对象作为 getMethod()

方法的参数。例如，下面的程序与前面的程序稍微有些区别：

import java.lang.annotation.\*; import java.lang.reflect.\*;

@Retention(RetentionPolicy.RUNTIME) @interface MyAnno {

String str(); int val();

}

class Meta {

// myMeth now has two arguments. @MyAnno(str = "Two Parameters", val = 19) public static void myMeth(String str, int i)

{

Meta ob = new Meta();

try {

Class<?> c = ob.getClass();

// Here, the parameter types are specified.

Method m = c.getMethod("myMeth", String.class, int.class);

MyAnno anno = m.getAnnotation(MyAnno.class);

System.out.println(anno.str() + " " + anno.val());

} catch (NoSuchMethodException exc) { System.out.println("Method Not Found.");

}

}

public static void main(String args[]) { myMeth("test", 10);

**294** 第Ⅰ部分 Java 语言

}

}

该版本的输出如下所示：

Two Parameters 19

在这个版本中，myMeth()方法带有一个 String 参数和一个 int 参数。为了获取关于这个方法的信息，必须以如下方式调用 getMethod()方法：

Method m = c.getMethod("myMeth", String.class, int.class);

在此，作为附加参数传递表示 String 和 int 类型的 Class 对象。

2. 获取所有注解

可以获取与某个条目关联的具有 RUNTIME 保留策略的所有注解，具体方法是为该条目调用 getAnnotations()方法。该方法的一般形式如下：

Annotation[ ] getAnnotations( )

上述方法返回一个注解数组。可以针对 Class、Method、Constructor 以及 Field 类型的对象调用 getAnnotations()方法。

下面是另外一个使用反射的例子，该例显示了如何获取与类和方法关联的所有注解。该例声明了两个注解。然后使用这两个注解来注解类和方法。

// Show all annotations for a class and a method. import java.lang.annotation.\*;

import java.lang.reflect.\*;

@Retention(RetentionPolicy.RUNTIME) @interface MyAnno {

String str(); int val();

}

@Retention(RetentionPolicy.RUNTIME) @interface What {

String description();

}

@What(description = "An annotation test class") @MyAnno(str = "Meta2", val = 99)

class Meta2 {

@What(description = "An annotation test method") @MyAnno(str = "Testing", val = 100)

第12章 枚举、自动装箱与注解(元数据) **295**

public static void myMeth() { Meta2 ob = new Meta2();

try {

Annotation annos[] = ob.getClass().getAnnotations();

// Display all annotations for Meta2. System.out.println("All annotations for Meta2:"); for(Annotation a : annos)

System.out.println(a);

System.out.println();

// Display all annotations for myMeth.

Method m = ob.getClass( ).getMethod("myMeth"); annos = m.getAnnotations();

System.out.println("All annotations for myMeth:"); for(Annotation a : annos)

System.out.println(a);

} catch (NoSuchMethodException exc) { System.out.println("Method Not Found.");

}

}

public static void main(String args[]) { myMeth();

}

}

输出如下所示：

All annotations for Meta2: @What(description=An annotation test class) @MyAnno(str=Meta2, val=99)

All annotations for myMeth: @What(description=An annotation test method) @MyAnno(str=Testing, val=100)

该程序使用 getAnnotations()方法来获取与类 Meta2 和方法 myMeth()相关联的所有注解，并将它们保存到数组中。正如前面所解释的，getAnnotations()方法返回 Annotation 对象的数

组。回忆一下，Annotation 是所有注解接口的超接口，并且它重载了 Object 类中的 toString() 方法。因此，当输出对 Annotation 的引用时，会调用 toString()方法来生成描述注解的字符串，如前面的输出所示。

**296** 第Ⅰ部分 Java 语言

12.4.4 AnnotatedElement 接口

前面例子中使用的 getAnnotation()和 getAnnotations()方法是由 AnnotatedElement 接口定义的，该接口在 java.lang.reflect 包中定义。这个接口支持注解反射，并且类 Method、Field、Constructor、Class 以及 Package 也都实现了该接口。

除了 getAnnotation()和 getAnnotations()方法外，AnnotatedElement 接口还定义了另外两个方法。第一个方法是 getDeclaredAnnotations()，该方法的一般形式如下所示：

Annotation[ ] getDeclaredAnnotations( )

上述方法返回调用对象中存在的所有非继承注解。第二个方法是 isAnnotationPresent()，

该方法的一般形式如下所示：

boolean isAnnotationPresent(Class<? extends Annotation> annoType)

如果 annoType 指定的注解与调用对象相关联，那么该方法返回 true，否则返回 false。

12.4.5 使用默认值

可以为注解成员提供默认值，应用注解时如果没有为注解成员指定值，就会使用默认值。默认值是通过为成员声明添加 default 子句指定的，一般形式如下所示：

*type member( ) default value;*

其中，value 的类型必须与 type 兼容。

下面是@MyAnno 的改写版，该版本提供了默认值：

// An annotation type declaration that includes defaults. @Retention(RetentionPolicy.RUNTIME)

@interface MyAnno {

String str() default "Testing"; int val() default 9000;

}

这个声明为 str 提供了默认值“Testing”，并为 val 提供了默认值 9000。这意味着使用

@MyAnno 时不需要指定这两个值。但是如果愿意的话，可以指定其中的一个或两个。所以，可以采用以下 4 种方式使用@MyAnno：

@MyAnno() // both str and val default @MyAnno(str = "some string") // val defaults @MyAnno(val = 100) // str defaults

@MyAnno(str = "Testing", val = 100) // no defaults

下面的程序演示了注解中默认值的使用：

第12章 枚举、自动装箱与注解(元数据) **297**

import java.lang.annotation.\*; import java.lang.reflect.\*;

// An annotation type declaration that includes defaults. @Retention(RetentionPolicy.RUNTIME)

@interface MyAnno {

String str() default "Testing"; int val() default 9000;

}

class Meta3 {

// Annotate a method using the default values. @MyAnno()

public static void myMeth() { Meta3 ob = new Meta3();

* Obtain the annotation for this method
* and display the values of the members. try {

Class<?> c = ob.getClass();

Method m = c.getMethod("myMeth");

MyAnno anno = m.getAnnotation(MyAnno.class);

System.out.println(anno.str() + " " + anno.val());

} catch (NoSuchMethodException exc) { System.out.println("Method Not Found.");

}

}

public static void main(String args[]) { myMeth();

}

}

输出如下所示：

Testing 9000

12.4.6 标记注解

标记注解是特殊类型的注解，其中不包含成员。标记注解的唯一目的就是标记声明，因此，这种注解作为注解而存在的理由是充分的。确定标记注解是否存在的最好方式是使用

isAnnotationPresent()方法，该方法是由 AnnotatedElement 接口定义的。

下面是一个使用标记注解的例子。因为标记注解不包含成员，所以只需要简单地判断其是否存在即可。

**298** 第Ⅰ部分 Java 语言

import java.lang.annotation.\*; import java.lang.reflect.\*;

// A marker annotation. @Retention(RetentionPolicy.RUNTIME) @interface MyMarker { }

class Marker {

* Annotate a method using a marker.
* Notice that no ( ) is needed. @MyMarker

public static void myMeth() { Marker ob = new Marker();

try {

Method m = ob.getClass().getMethod("myMeth");

// Determine if the annotation is present. if(m.isAnnotationPresent(MyMarker.class))

System.out.println("MyMarker is present.");

} catch (NoSuchMethodException exc) { System.out.println("Method Not Found.");

}

}

public static void main(String args[]) { myMeth();

}

}

输出如下所示，@MyMarker 确实是存在的：

MyMarker is present.

在这个程序中，应用@MyMarker 时后面不需要有圆括号。因此，只通过名称即可应用

@MyMarker，如下所示：

@MyMarker

提供空的圆括号虽然不是错误，但不是必需的。

12.4.7 单成员注解

单成员注解只包含一个成员。除了允许使用缩写形式指定成员的值之外，单成员注解的工作方式和常规注解类似。如果只有一个成员，应用注解时就可以简单地为该成员指定值，而不需要指定成员的名称。但是，为了使用这种缩写形式，成员名称必须是 value。

第12章 枚举、自动装箱与注解(元数据) **299**

下面是一个创建和使用单成员注解的例子：

import java.lang.annotation.\*; import java.lang.reflect.\*;

// A single-member annotation. @Retention(RetentionPolicy.RUNTIME) @interface MySingle {

int value(); // this variable name must be value

}

class Single {

// Annotate a method using a single-member annotation. @MySingle(100)

public static void myMeth() { Single ob = new Single();

try {

Method m = ob.getClass().getMethod("myMeth");

MySingle anno = m.getAnnotation(MySingle.class);

System.out.println(anno.value()); // displays 100

} catch (NoSuchMethodException exc) { System.out.println("Method Not Found.");

}

}

public static void main(String args[]) { myMeth();

}

}

正如所期望的，这个程序显示值 100。在这个程序中，@MySingle 用于注解 myMeth()

方法，如下所示：

@MySingle(100)

注意没有具体指定“value=”。

当使用含有其他成员的注解时，也可以使用单值语法，但是其他成员必须都有默认值。例如，下面添加了成员 xyz，它带有默认值 0：

@interface SomeAnno { int value();

int xyz() default 0;

}

**300** 第Ⅰ部分 Java 语言

对于希望为 xyz 使用默认值的情况，可以使用如下所示的方式应用@SomeAnno，使用单成员语法简单地指定 value 的值：

@SomeAnno(88)

在这个例子中，xyz 默认为 0，value 的值为 88。当然，如果为 xyz 指定不同的值，就需要显式地提供两个成员的名称，如下所示：

@SomeAnno(value = 88, xyz = 99)

请记住，只要使用单成员注解，成员的名称就必须是 value。

12.4.8 内置注解

Java 提供了许多内置注解。大部分是专用注解，但是有 8 个用于一般目的。在这个 8 个

注解中，有 4 个来自 java.lang.annotation 包，分别是@Retention、@Documented、@Target 和@Inherited；另外 4 个——@Override、@Deprecated、@SafeVarargs 和@SuppressWarnings 来

* java.lang 包。下面逐一介绍这些注解。
  1. @Retention

@Retention 被设计为只能用于注解其他注解。如前所述，@Retention 用于指定保留策略。

2. @Documented

@Documented 注解是标记接口，用于通知某个工具——注解将被文档化。@Documented

被设计为只能注解其他注解。

3. @Target

@Target 用于指定可以应用注解的声明的类型，被设计为只能注解其他注解。@Target

只有一个参数，这个参数必须是来自 ElementType 枚举的常量，这个参数指定了将为其应用注解的声明的类型。表 12-1 中显示了这些常量以及与之对应的声明的类型。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 表 12-1 | 目标常量以及与之对应的声明的类型 | |
|  | 目 标 常 量 |  | 可应用注解的声明的类型 |
|  |  |  |  |
|  | ANNOTATION\_TYPE |  | 另外一个注解 |
|  | CONSTRUCTOR |  | 构造函数 |
|  | FIELD |  | 域变量 |
|  | LOCAL\_VARIABLE |  | 局部变量 |
|  | METHOD |  | 方法 |
|  | PACKAGE |  | 包 |
|  | PARAMETER |  | 参数 |
|  | TYPE |  | 类、接口或枚举 |

第12章 枚举、自动装箱与注解(元数据) **301**

在@Target 注解中可以指定这些值中的一个或多个。为了指定多个值，必须在由花括号包围起来的列表中指定它们。例如，为了指定注解只能应用于域变量和局部变量，可以使用下面这个@Target 注解：

@Target( { ElementType.FIELD, ElementType.LOCAL\_VARIABLE } )

4. @Inherited

@Inherited 是标记注解，只能用于另外一个注解声明。此外，@Inherited 只影响用于类声明的注解。@Inherited 会导致超类的注解被子类继承。所以，当查询子类的特定注解时，如果那种注解在子类中不存在，就会检查超类。如果那种注解存在于超类中，并且如果使用 @Inherited 进行了注解，就将返回那种注解。

5. @Override

@Override 是标记注解，只能用于方法。使用带有@Override 注解的方法必须重写超类中的方法。如果不这样做的话，就会产生编译时错误。@Override 注解用于确保超类方法被真正地重写，而不是简单地重载。

6. @Deprecated

@Deprecated 是标记注解，用于指示声明是过时的，并且已经被更新的形式取代。

7. @SafeVarargs

@SafeVarargs 是标记注解，只能用于方法和构造函数，指示没有发生与可变长度参数相关的不安全动作。如果不安全代码与不能具体化的 varargs 类型相关，或者与参数化的数组实例相关，那么@SafeVarargs 注解用于抑制“未检查不安全代码”警告(本质上，不能具体化的类是泛型类，泛型将在第 14 章讨论)。@SafeVarargs 注解只能用于 varargs 方法或者声明为 static

* final 的构造函数。@SafeVarargs 注解是由 JDK 7 新增的。
  1. @SuppressWarnings

@SuppressWarnings 注解用于指定能抑制一个或多个编译器可能会报告的警告。使用名

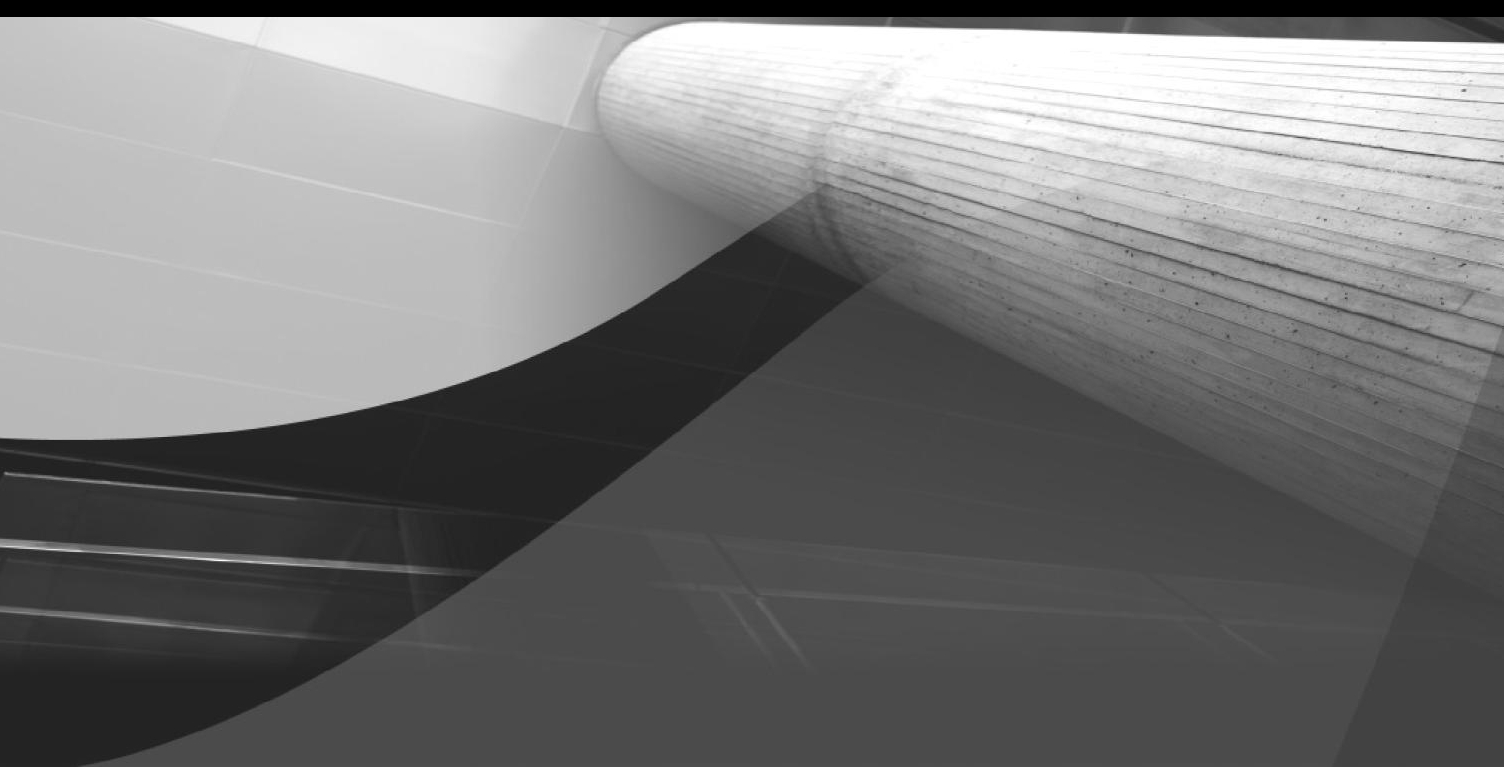
称以字符串形式指定要被抑制的警告。这个注解可以用于所有类型的声明。

12.4.9 一些限制

使用注解声明有许多限制。首先，一个注解不能继承另一个注解。其次，注解声明的所有方法都必须不带参数。此外，它们不能返回以下类型的值：

* 基本类型，例如 int 或 double。
* String 或 Class 类型的对象。
* 枚举类型。
* 其他注解类型。

注解不能被泛型化。换句话说，它们不能带有类型参数(泛型将在第 14 章介绍)。最后，注解方法不能指定 throws 子句。



**第 20 章**

探究 NIO

从 1.4 版本开始，Java 提供了另一套 I/O 系统，称为 NIO(New I/O 的缩写)。NIO 支持面向缓存的、基于通道的 I/O 操作。随着 JDK 7 的发布，Java 对 NIO 系统进行了极大扩展，增强了对文件处理和文件系统特性的支持。事实上，这些修改是如此重要，以至于经常使用术语 NIO.2。缘于新的 NIO 文件类提供的功能，NIO 预期会成为文件处理中越来越重要的部分。本章将研究 NIO 系统的一些关键特性，包括新的文件处理功能。

**20.1** **NIO 类**

包含 NIO 类的包如表 20-1 所示。

**644** 第Ⅱ部分 Java 库

表 20-1 包含 INO 类的包

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 包 | 目 | 的 |  |
|  |  |  | |  |
|  | java.nio | NIO 系统的顶级包，用于封装各种类型的缓存，这些缓存包含 NIO 系统所操 | |  |
|  | 作的数据 |  |  |
|  |  |  |  |
|  |  |  | |  |
|  | java.nio.channels | 支持通道，通道本质上是打开的 I/O 连接 | |  |
|  | java.nio.channels.spi | 支持通道的服务提供者 |  |  |
|  |  | 封装字符集，另外还支持分别将字符转换成字节以及将字节转换成字符的编码 | |  |
|  | java.nio.charset | 器和解码器 |  |  |
|  |  |  |  |
|  |  |  |  |  |
|  | java.nio.charset.spi | 支持字符集的服务提供者 |  |  |
|  | java.nio.file | 提供对文件的支持(该包是由 JDK 7 新增的) | |  |
|  | java.nio.file.attribute | 提供对文件属性的支持(该包是由 JDK 7 新增的) | |  |
|  | java.nio.file.spi | 支持文件系统的服务提供者(该包是由 JDK 7 新增的) | |  |

在开始介绍之前需要强调的是，NIO 系统并非用于替换 java.io 中基于流的 I/O 类，在第

19 章讨论了这些类。如果能够很好地掌握 java.io 中基于流的 I/O，那么对于理解 NIO 是有帮

助的。

注意：



本章假定您已经阅读了第 13 章中的 I/O 概述以及第 19 章提供的对基于流的 I/O 的讨论。

**20.2** **NIO 的基础知识**

NIO 系统构建于两个基础术语之上：缓存和通道。缓存用于容纳数据，通道表示打开的

到 I/O 设备(例如文件或套接字)的连接。通常，为了使用 NIO 系统，需要获取用于连接 I/O

设备的通道以及用于容纳数据的缓存。然后操作缓存，根据需要输入或输出数据。接下来将

更加详细地分析缓存和通道。

20.2.1 缓存

缓存是在 java.nio 包中定义的。所有缓存都是 Buffer 类的子类，Buffer 类定义了对所有

缓存都通用的核心功能：当前位置、界限和容量。当前位置是缓存中下一次发生读取和写入

操作的索引，当前位置通过大多数读或写操作向前推进。界限是缓存中最后一个有效位置之

后下一个位置的索引值。容量是缓存能够容纳的元素的数量。通常界限等于缓存的容量。

Buffer 类还支持标记和重置。Buffer 类定义了一些方法，表 20-2 显示了这些方法。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | 第20章 探究NIO**645** | |  |
|  |  |  | 表 20-2 Buffer 类定义的方法 | |  |  |  |
|  | 方 | 法 |  | 描 | 述 | |  |
|  |  |  |  |  | |  |  |
|  |  |  |  | 如果调用缓存是基于数组的，就返回对数组的引用；否则抛出 | | |  |
|  | abstract Object array() |  |  | UnsupportedOperationException | 异常；如果数组是只读的，就抛出 | |  |
|  |  |  |  | ReadOnlyBufferException 异常 |  |  |  |
|  |  |  |  | 如果调用缓存是基于数组的，就返回第一个元素的索引，否则抛出 | | |  |
|  | abstract int arrayOffset() | |  | UnsupportedOperationException | 异常；如果数组是只读的，就抛出 | |  |
|  |  |  |  | ReadOnlyBufferException 异常 |  |  |  |
|  | final int capacity() |  |  | 返回调用缓存能够容纳的元素数量 | |  |  |
|  | final Buffer clear() |  |  | 清空调用缓存并返回对缓存的引用 | |  |  |
|  | final Buffer flip() |  |  | 将调用缓存的界限设置为当前位置，并将当前位置重置为 0。返回对 | | |  |
|  |  |  | 缓存的引用 |  |  |  |
|  |  |  |  |  |  |  |
|  |  | |  |  | |  |  |
|  | abstract boolean hasArray() | |  | 如果调用缓存是基于读/写数组的，就返回 true；否则返回 false | |  |  |
|  | final boolean hasRemaining() | |  | 如果在调用缓存中还有剩余元素，就返回 true；否则返回 false | |  |  |
|  | abstract boolean isDirect() | |  | 如果调用缓存是定向的，就返回 true，这意味着可以直接对缓存进行 | | |  |
|  |  | I/O 操作；否则返回 false |  |  |  |
|  |  |  |  |  |  |  |
|  | abstract boolean isReadOnly() | |  | 如果调用缓存是只读的，就返回 true；否则返回 false | | |  |
|  | final int limit() |  |  | 返回调用缓存的界限 |  |  |  |
|  | final Buffer limit (int n) | |  | 将调用缓存的界限设置为 n。返回对缓存的引用 | |  |  |
|  | final Buffer mark() |  |  | 对调用缓存设置标记并返回对缓存的引用 | | |  |
|  | final int position () |  |  | 返回当前位置 |  |  |  |
|  | final Buffer position (int n) | |  | 将调用缓存的当前位置设置为 n。返回对缓存的引用 | |  |  |
|  |  |  |  | 返回在到达界限之前可用元素的数量。换句话说，返回界限减去当 | | |  |
|  | int remaining() |  |  | 前位置后的值 |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  | |  |  |
|  | final Buffer reset () |  |  | 将调用缓存的当前位置重置为以前设置的标记。返回对缓存的引用 | |  |  |
|  | final Buffer rewind( ) |  |  | 将调用缓存的位置设置为 0。返回对缓存的引用 | |  |  |

下面这些特定的缓存类派生自 Buffer，这些类的名称暗含了它们所能容纳的数据的类型：

ByteBuffer CharBuffer DoubleBuffer FloatBuffer

IntBuffer LongBuffer MappedByteBuff ShortBuffer

er

MappedByteBuffer 是 ByteBuffer 的子类，用于将文件映射到缓存。

所有缓存类都提供了不同的 get()和 put()方法，使用这些方法可以从缓存获取数据或将数据放入到缓存中(当然，如果缓存是只读的，就不能使用 put()操作)。表 20-3 显示了 ByteBuffer

类定义的 get()和 put()方法。其他缓存类具有类似的方法。所有缓存类都支持用于执行各种缓

1. 第Ⅱ部分 Java 库

存操作的方法。例如，可以使用 allocate()方法手动分配缓存，使用 wrap()方法在缓存中封装数组，使用 slice()方法创建缓存的子序列。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 表 20-3 | ByteBuffer 类定义的 get()和 put()方法 | |  |  |
|  | 方 | 法 |  | 描 | 述 |  |
|  |  |  |  |  |  |  |
|  | abstract byte get() |  |  | 返回当前位置的字节 |  |  |
|  |  |  |  | 将调用缓存复制到 val 引用的数组中，返回对缓存的引用。如果 | |  |
|  | ByteBuffer get(byte vals[]) | |  | 缓存中剩余元素的数量小于 vals.length，就会抛出 BufferUnderflow | |  |
|  |  |  |  | Exception 异常 |  |  |
|  | ByteBuffer get(byte val[], | |  | 从调用缓存复制 num 个元素到 val 引用的数组中，从 start 指定 | |  |
|  |  | 的索引位置开始保存。返回对缓存的引用。如果缓存中剩余元素 | |  |
|  |  |  |  |  |
|  | int start, int num) | |  | 的数量不足 num，就会抛出 BufferUnderflowException 异常 | |  |
|  |  |  |  |  |
|  | abstract byte get(int idx) |  |  | 返回调用缓存中由 idx 指定的索引位置的字节 | |  |
|  | abstract ByteBuffer put(byte b) | |  | 将 b 复制到调用缓存的当前位置，返回对缓存的引用，如果缓存 | |  |
|  |  | 已满，就会抛出 BufferOverflowException 异常 | |  |
|  |  |  |  |  |
|  |  |  |  | 将 vals 中的所有元素复制到调用缓存中，从当前位置开始。返回 | |  |
|  | final ByteBuffer put(byte vals[]) | |  | 对缓存的引用。如果缓存不能容纳所有元素，就会抛出 | |  |
|  |  |  |  | BufferOverflowException 异常 |  |  |
|  | ByteBuffer put(byte vals [], | |  | 将 vals 中从 start 开始的 num 个元素复制到调用缓存中。返回对 | |  |
|  |  | 缓存的引用。如果缓存不能容纳全部元素，就会抛出 | |  |
|  |  |  |  |  |
|  | int start, int num) | |  | BufferOverfowException 异常。 |  |  |
|  |  |  |  |  |  |
|  |  |  |  | 将 bb 中的元素复制到调用缓存中，从当前位置开始。如果缓存 | |  |
|  | ByteBuffer put(ByteBuffer bb) | |  | 不能容纳全部元素，就会抛出 BufferOverflowException 异常。返 | |  |
|  |  |  |  | 回对缓存的引用 |  |  |
|  |  | |  |  | |  |
|  | abstract ByteBuffer put(int idx, byte b) | |  | 将 b 复制到调用缓存中 idx 指定的位置，返回对缓存的引用 | |  |

20.2.2 通道

通道是由 java.nio.channels 包定义的。通道表示到 I/O 源或目标的打开的连接。通道实现了 Channel 接口并扩展了 Closeable 接口，并且从 JDK 7 开始还扩展了 AutoCloseable 接口。通过实现 AutoCloseable 接口，可以使用 JDK 7 新增的带资源的 try 语句管理通道。如果使用带资源的 try 语句，那么当通道不再需要时会自动关闭(关于带资源的 try 语句的讨论，请查看第 13 章)。

获取通道的一种方式是对支持通道的对象调用 getChannel()方法。例如，以下 I/O 类支持

getChannel()方法：

DatagramSocket FileInputStream FileOutputStream

RandomAccessFile ServerSocket Socket

第20章 探究 NIO **647**

根据调用 getChannel() 方法的对象的类型返回特定类型的通道。例如，当对

FileInputStream、FileOutputStream 或 RandomAccessFile 对象调用 getChannel()方法时，会返回 FileChannel 类型的通道。当对 Socket 对象调用 getChannel()方法时，会返回 SocketChannel

类型的通道。

获取通道的另外一种方式是使用 Files 类定义的静态方法，该类是由 JDK 7 新增的。例如，使用 Files 类，可以通过 newByteChannel() 方法获取字节通道。该方法返回一个

SeekableByteChannel 对象，SeekableByteChannel 是 FileChannel 实现的一个接口(在本章的后

面将详细分析 Files 类)。

FileChannel 和 SocketChannel 这类通道支持各种 read()和 write()方法，使用这些方法可以通过通道执行 I/O 操作。例如，表 20-4 中是为 FileChannel 定义的一些 read()和 write()方法。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 表 20-4 | 为 FileChannel 定义的 read()和 write()方法 | | |  |
|  | 方 | 法 |  | 描 | 述 |  |
|  |  | |  |  | |  |
|  | abstract int read(ByteBuffer bb) | |  | 从调用通道读取字节到 bb 中，直到缓存已满或者不再有输入内容 | |  |
|  | throws IOException | |  | 为止。返回实际读取的字节数 |  |  |
|  | abstract int read(ByteBuffer bb, long start) | | | 从 start 指定的文件位置开始，从调用通道读取字节到 bb 中，直 | |  |
|  | 到缓存已满或者不再有输入内容为止。不改变当前位置。返回实 | |  |
|  |  |  |  |  |
|  | throws IOException | |  | 际读取的字节数，如果 start 超出文件的末尾，就返回−l | |  |
|  |  |  |  |  |
|  | abstract int write(ByteBuffer bb) | |  | 将 bb 的内容写入到调用通道中，从当前位置开始。返回写入的字 | |  |
|  | throws IOException | |  | 节数 |  |  |
|  | abstract int write(ByteBuffer bb, long start) | | | 从 start 指定的文件位置开始，将 bb 中的内容写入调用通道中。 | |  |
|  | throws IOException | |  | 不改变当前位置。返回写入的字节数 | |  |

所有通道都支持一些额外的方法，通过这些方法可以访问和控制通道。例如，FileChannel

支持获取或设置当前位置的方法、在文件通道之间传递信息的方法、获取当前通道大小的方法以及锁定通道的方法，等等。从 JDK 7 开始，FileChannel 还提供了静态的 open()方法，该方法打开文件并返回指向文件的通道。这提供了获取通道的另外一种方式。FileChannel 还提供了 map()方法，通过该方法可以将文件映射到缓存。

20.2.3 字符集和选择器

NIO 使用的两个实体是字符集和选择器。字符集定义了将字节映射为字符的方法。可以

使用编码器将一系列字符编码成字节，使用解码器将一系列字节解码成字符。字符集、编码器和解码器由 java.nio.charset 包中定义的类支持。因为提供了默认的编码器和解码器，所以通常不需要显式地使用字符集进行工作。

选择器支持基于键的、非锁定的多通道 I/O。换句话说，使用选择器可以通过多个通道执行 I/O。选择器由 java.nio.channels 包中定义的类支持。选择器最适合用于基于套接字的通道。

**648** 第Ⅱ部分 Java 库

在本章不会使用字符集或选择器，但是您在自己的程序中可能会发现它们很有用。

1. **JDK 7 对 NIO 的增强**
   * JDK 7 开始，Java 对 NIO 系统进行了充分扩展和增强。除了支持带资源的 try 语句(这种 try 语句提供了自动资源管理功能)外，JDK 7 对 NIO 的改进包括：3 个新包(java.nio.file、 java.nio.file.attribute 和 java.nio.file.spi)；一些新类、接口和方法；以及对基于流的 I/O 的定向

支持。这些增加的内容极大扩展了 NIO 的使用方式，特别是文件。接下来将介绍一些关键的新增内容。

20.3.1 Path 接口

对 NIO 系统最重要的新增内容可能是 Path 接口，因为该接口封装了文件的路径。在后面会看到，Path 接口是 NIO.2 中将基于文件的新特性捆绑在一起的黏合剂，描述了目录结构中文件的位置。Path 接口被打包到 java.nio.file 中，并且继承自下列接口：Watchable、Iterable<Path>和 Comparable<Path>。Watchable 接口描述了可以被监视是否发生变化的对象，该接口也是由 JDK 7 新增的。Iterable 和 Comparable 接口在本书前面介绍过。

Path 接口声明了操作路径的大量方法。表 20-5 显示了其中的一些方法。请特别注意getName()方法，该方法用于获取路径中的元素并使用索引进行工作。在 0 索引位置，也就是路径中最靠近根路径的部分，是路径中最左边的元素。后续索引标识根路径右侧的元素。通过调用 getNameCount()方法可以获取路径中元素的数量。如果希望获取整个路径的字符串表示，可简单地调用 toString()方法。注意可以使用 resolve()方法将相对路径解析为绝对路径。

表 20-5 Path 接口定义的方法举例

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 方 | 法 | 描 | 述 |  |
|  |  | |  | |  |
|  | boolean endsWith(String path) | | 如果调用 Path 对象以 path 指定的路径结束，就返回 true；否则返回 false | |  |
|  |  | |  | |  |
|  | boolean endsWith(Path path) | | 如果调用 Path 对象以 path 指定的路径结束，就返回 true；否则返回 false | |  |
|  |  |  |  |  |  |
|  | Path getFileName() |  | 返回与调用 Path 对象关联的文件名 |  |  |
|  |  |  |  | |  |
|  |  |  | 返回的 Path 对象包含调用对象中由 idx 指定的路径元素的名称。最左边 | |  |
|  | Path getName(int idx) | | 的元素位于 0 索引位置，这是离根路径最近的元素。最右边的元素位于 | |  |
|  |  |  | getNameCount()－1 索引位置 |  |  |
|  |  |  |  | |  |
|  | int getNameCount() |  | 返回调用对象中超出根目录的元素的数量 | |  |
|  |  |  |  | |  |
|  | Path getParent() |  | 返回的 Path 对象包含整个路径，但是不包含由调用 Path 对象指定的文 | |  |
|  |  | 件的名称 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |  |
|  | Path getRoot() |  | 返回调用 Path 对象的根路径 |  |  |
|  |  |  |  | |  |
|  | boolean isAbsolute() |  | 如果调用 Path 对象是绝对路径，就返回 true；否则返回 false | |  |
|  |  |  |  | |  |
|  | Path resolve(Path path) | | 如果 path 是绝对路径，就返回 path；否则，如果 path 包含根路径，就 | |  |
|  | 在 path 前面加上由调用 Path 对象指定的根路径，并返回结果。如果 path | |  |
|  |  |  |  |
|  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  | 第20章 探究NIO**649** | |
|  |  |  |  | |  |
|  |  |  | 为空，就返回调用 Path 对象；否则不对行为进行指定的 | | |
|  |  |  |  |  |  |
|  |  |  |  | (续表) | |
|  |  |  |  |  |  |
|  | 方 | 法 | 描 | 述 | |
|  |  |  |  | |  |
|  |  |  | 如果 path 是绝对路径，就返回 path；否则，如果 path 包含根路径，就 | | |
|  | Path resolve(String path) | | 在 path 前面加上由调用 Path 对象指定的根路径，并返回结果。如果 path | | |
|  |  |  | 为空，就返回调用 Path 对象；否则不对行为进行指定的 | | |
|  |  | |  | |  |
|  | boolean startsWith(String path) | | 如果调用 Path 对象以 path 指定的路径开始，就返回 true；否则返回 false | |  |
|  | boolean startsWith(Path path) | | 如果调用 Path 对象以 path 指定的路径开始，就返回 true；否则返回 false | | |
|  | Path toAbsolutePath() |  | 作为绝对路径返回调用 Path 对象 |  |  |
|  | String toString() |  | 返回调用 Path 对象的字符串表示形式 |  |  |

另外一点：当更新那些使用 File 类(在 java.io 包中定义)的遗留代码时，可以通过对 File 对象调用 toPath()方法，将 File 实例转换成 Path 实例。该方法是由 JDK 7 添加到 File 类中的。此外，可以通过调用 Path 接口定义的 toFile()方法来获取 File 实例。

20.3.2 Files 类

对文件执行的许多操作都是由 Files 类(在 java.io.file 包中定义)中的静态方法提供的。要进行操作的文件是由文件的 Path 对象指定的；因此，Files 类的方法使用 Path 对象指定将要进行操作的文件。Files 类提供了广泛的功能。例如，提供了打开或创建具有特定路径的文件的方法。可以获取关于 Path 对象的信息，例如是否可执行、是隐藏的还是只读的。Files 类还支持复制和移动文件的方法。表 20-6 中显示了 Files 类提供的一些方法。除了可能抛出 IOException 异常外，也可能抛出其他异常。

表 20-6 Files 类定义的方法举例

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 方 | 法 | 描 | 述 |  |
|  |  | |  |  |  |
|  | static Path copy(Path src, Path dest, | | 将 src 指定的文件复制到 dest 指定的位置。参数 how 指 | |  |
|  | CopyOption… how) | |  |
|  | 定了复制是如何发生的 |  |  |
|  | throws IOException |  |  |  |
|  |  |  |  |  |
|  |  | |  |  |  |
|  | static Path createDirectory(Path path, | | 创建一个目录，path 指定了该目录的路径。目录属性是 | |  |
|  |  | FileAttribute<?>… attribs) |  |
|  |  | 由 attribs 指定的 |  |  |
|  | throws IOException |  |  |  |
|  |  |  |  |  |
|  |  | |  |  |  |
|  | static Path createFile(Path path, | | 创建一个文件，path 指定了该文件的路径。文件属性是 | |  |
|  | FileAttribute<?>… attribs) | |  |
|  | 由 attribs 指定的 |  |  |
|  | throws IOException |  |  |  |
|  |  |  |  |  |
|  |  | |  | |  |
|  | static void delete(Path path) throws IOException | | 删除一个文件，path 指定了该文件的路径 | |  |
|  |  | |  | |  |
|  | static boolean exists(Path path, | | 如果 path 指定的文件存在，就返回 true；否则返回 false。 | |  |
|  |  |  |  |  |  |

**650** 第Ⅱ部分 Java 库

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | LinkOptions… opts) | | 如果没有指定 opts，就使用符号链接。为了阻止符号链 | |  |
|  |  |  | 接，可以为 opts 传递 NOFOLLOW\_LINKS | |  |
|  |  |  |  |  |  |
|  |  |  |  | (续表) |  |
|  |  |  |  |  |  |
|  | 方 | 法 | 描 | 述 |  |
|  |  |  |  | |  |
|  | static boolean isDirectory(Path path, | | 如果 path 指定的是目录，就返回 true；否则返回 false。 | |  |
|  | 如果没有指定 opts，就使用符号链接。为了阻止符号链 | |  |
|  |  | LinkOptions… opts) |  |
|  |  | 接，可以为 opts 传递 NOFOLLOW\_LINKS | |  |
|  |  |  |  |
|  |  |  |  | |  |
|  | static boolean isExecutable(Path path) | | 如果 path 指定的是可执行文件，就返回 true；否则返回 | |  |
|  | false |  |  |
|  |  |  |  |  |
|  |  | |  | |  |
|  | static boolean isHidden(Path path) | | 如果 path 指定的文件是隐藏的，就返回 true；否则返回 | |  |
|  | throws IOException |  | false |  |  |
|  |  |  |  | |  |
|  | static boolean isReadable(Path path) | | 如果 path 指定的文件是可读的，就返回 true；否则返回 | |  |
|  | false |  |  |
|  |  |  |  |  |
|  |  |  |  | |  |
|  | static boolean isRegularFile(Path path, | | 如果 path 指定的是文件，就返回 true；否则返回 false。 | |  |
|  | 如果没有指定 opts，就使用符号链接。为了阻止符号链 | |  |
|  |  | LinkOptions… opts) |  |
|  |  | 接，可以为 opts 传递 NOFOLLOW\_LINKS | |  |
|  |  |  |  |
|  |  |  |  | |  |
|  | static boolean isWritable(Path path) | | 如果 path 指定的文件是可写的，就返回 true；否则返回 | |  |
|  | false |  |  |
|  |  |  |  |  |
|  |  | |  |  |  |
|  | static Path move(Path src, Path dest | | 将 src 指定的文件移动到 dest 指定的位置。参数 how 指 | |  |
|  | CopyOption… how) | |  |
|  | 定了文件移动是如何发生的 |  |  |
|  | throws IOException |  |  |  |
|  |  |  |  |  |
|  |  |  |  | |  |
|  | static SeekableByteChannel | | 打开 path 指定的文件，how 指定了打开方式。返回一个 | |  |
|  | 链接到该文件的 SeekableByteChannel 对象。这是一个可 | |  |
|  | new ByteChannel(Path path, OpenOption… how) | |  |
|  | 以改变当前位置的字节通道。FileChannel 类实现了 | |  |
|  | throws IOException |  |  |
|  |  | SeekableByteChannel 方法 |  |  |
|  |  |  |  |  |
|  |  | |  |  |  |
|  | static DirectoryStream<Path> | | 打开 path 指定的目录。返回一个链接到该目录的 | |  |
|  | newDirectoryStream(Path path) | |  |
|  | DirectoryStream 对象 |  |  |
|  | throws IOException |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |  |
|  | static InputStream |  | 打开 path 指定的文件，how 指定了打开方式。返回一个 | |  |
|  | newInputStream(Path path, OpenOption… how) | |  |
|  | 链接到该文件的 InputStream 对象 | |  |
|  | throws IOException |  |  |
|  |  |  |  |  |
|  |  |  |  |  |  |
|  | static OutputStream |  |  |  |  |
|  | newOutputStream(Path path, | | 打开 path 指定的文件，how 指定了打开方式。返回一个 | |  |
|  | OpenOption… how) | | 链接到该文件的 OutputStream 对象 | |  |
|  | throws IOException |  |  |  |  |
|  |  |  |  | |  |
|  | static boolean |  | 如果 path 指定的文件不存在，就返回 true；否则返回 | |  |
|  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  | 第20章 探究NIO**651** | |
|  |  | |  | |  |
|  | notExists(Path path, LinkOption… opts) | | false。如果没有指定 opts，就使用符号链接。为了阻止 | | |
|  |  |  | 符号链接，可以为 opts 传递 NOFOLLOW\_LINKS | | |
|  |  |  |  |  |  |
|  |  |  |  | (续表) | |
|  |  |  |  |  |  |
|  | 方 | 法 | 描 | 述 | |

static <A extends BasicFileAttributes> A

获取与 path 指定的文件相关联的属性。获取的属性类型

readAttributes(Path path,

是由 attibType 传递的。如果没有指定 opts，就使用符号

Class<A> attribType,

链接。为了阻止符号链接，可以为 opts 传递 NOFOLLOW\_

LinkOption … opts)

LINKS

throws IOException

static long size(Path path)

返回由 path 指定的文件的大小

throws IOException

注意表 20-6 中的一些方法采用类型为 OpenOption 的参数。OpenOption 是描述打开文件方式的接口，StandardOpenOption 类实现了该接口。StandardOpenOption 类定义了一个枚举，

* 20-7 显示了这个枚举包含的值。
  + 20-7 标准的文件打开选项

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 值 | 含 | 义 |  |
|  |  |  |  |  |
|  | APPEND | 导致输出被写入到文件的末尾 |  |  |
|  | CREATE | 如果文件不存在，就创建文件 |  |  |
|  | CREATE\_NEW | 只有当文件不存在时才创建文件 |  |  |
|  | DELETE\_ON\_CLOSE | 当文件被关闭时删除文件 |  |  |
|  |  | 导致对文件的修改被立即写入到物理文件。正常情况下，出于效率考虑， | |  |
|  | DYSNC | 对文件的修改由文件系统进行缓存，只有当需要时才写入到文件中 | |  |
|  |  |  |
|  |  |  |  |  |
|  | READ | 为输入操作打开文件 |  |  |
|  |  | 指示文件系统——文件是稀疏的，这意味着文件中可能没有完全填满数 | |  |
|  | SPARSE | 据。如果文件系统不支持稀疏文件，那么会忽略该选项 | |  |
|  |  |  |
|  |  |  | |  |
|  |  | 导致对文件或文件中元数据的修改被立即写入到物理文件。正常情况下， | |  |
|  | SYNC | 出于效率考虑，对文件的修改由文件系统进行缓存，只有当需要时才写 | |  |
|  |  | 入到文件中 |  |  |
|  |  |  | |  |
|  | TRUNCATE\_EXISTING | 将为输出操作而打开的、之前就存在的文件的长度减少到 0 | |  |
|  | WRITE | 为输出操作打开文件 |  |  |

20.3.3 Path 接口

因为 Path 是接口，不是类，所以不能通过构造函数直接创建 Path 的实例。但是，可以

**652** 第Ⅱ部分 Java 库

通过调用方法来返回 Path 的实例。通常，使用 Path 接口定义的 get()方法来完成该工作。get()

方法有两种形式。在本章中使用的形式如下所示：

static Path get(String pathname, String ... parts)

该方法返回一个封装指定路径的 Path 对象。可以通过两种形式指定路径。第一种，如果没有使用 parts，就必须通过 pathname 以整体来指定路径。如果使用了 parts，那么可以分块传递路径，使用 pathName 传递第 1 部分，通过 parts 可变参数指定后续部分。对于这两种情况，如果指定的路径在语法上无效，get()方法会抛出 InvalidPathException 异常。

get()方法的第二种形式根据 URI 来创建 Path 对象，如下所示：

static Path get(URI uri)

这种形式返回与 uri 对应的 Path 对象。

创建链接到文件的 Path 对象不会导致打开或创建文件，理解这一点很重要。这仅仅创建了封装文件目录路径的对象而已。

20.3.4 文件属性接口

与文件关联的是一套属性。这些属性包括文件的创建时间、最后一次修改时间、文件是否是目录以及文件的大小等内容。NIO 将文件属性组织到几个接口中。属性是通过在 java.nio.file.attribute 包中定义的接口层次表示的。顶部是 BasicFileAttributes，该接口封装了在各种文件系统中都通用的一组属性。表 20-8 显示了 BasicFileAttributes 接口定义的方法。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 表 20-8 | | BasicFileAttributes 接口定义的方法 | |  |
|  | 方 | 法 |  | 描 | 述 |  |
|  |  |  |  |  | |  |
|  |  |  |  | 返回文件的创建时间。如果文件系统没有提供创建时间，就返回一个依 | |  |
|  | FileTime creationTime() | |  | 赖于实现的时间值 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  | |  |
|  | Object fileKey() |  |  | 返回文件键。如果不支持，就返回 null | |  |
|  | boolean isDirectory() | |  | 如果文件表示目录，就返回 true |  |  |
|  | boolean isOther() |  |  | 如果文件不是文件、符号链接或目录，就返回 true | |  |
|  | boolean isRegularFile() | |  | 如果文件是常规文件，而不是目录或符号链接，就返回 true | |  |
|  | boolean isSymbolicLink() | |  | 如果文件是符号链接，就返回 true |  |  |
|  |  |  |  | 返回文件的最后一次访问时间。如果文件系统没有提供最后访问时间， | |  |
|  | FileTime lastAccessTime() | |  | 就返回一个依赖于实现的时间值 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  | |  |
|  |  |  |  | 返回文件的最后一次修改时间。如果文件系统没有提供最后一次修改时 | |  |
|  | FileTime lastModifiedTime() | |  | 间，就返回一个依赖于实现的时间值 | |  |
|  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | long size() |  |  | 返回文件的大小 |  |  |

有两个接口派生自 BasicFileAttributes：DosFileAttributes 和 PosixFileAttributes。DosFile-

第20章 探究 NIO **653**

Attributes 描述了与 FAT 文件系统相关的那些属性，FAT 文件系统最初是由 DOS 定义的。

DosFileAttributes 接口定义的方法如表 20-9 所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | 表 20-9 | | DosFileAttributes 接口定义的方法 |  |
|  | 方 | 法 |  | 描 | 述 |
|  |  |  |  |  | |
|  | boolean isArchive() |  |  | 如果文件被标记为存档文件，就返回 true；否则返回 false | |
|  | boolean isHidden() |  |  | 如果文件是隐藏的，就返回 true；否则返回 false | |
|  | boolean isReadOnly() |  |  | 如果文件是只读的，就返回 true；否则返回 false | |
|  | boolean isSystem() |  |  | 如果文件被标记为系统文件，就返回 true；否则返回 false | |

PosixFileAttributes 封装了 POSIX 标准定义的属性(POSIX 代表 Portable Operating System Interface，即轻便型操作系统接口)，该接口定义的方法如表 20-10 所示。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 表 20-10 | PosixFileAttributes 接口定义的方法 |  |
|  | 方 | 法 | 描 | 述 |
|  |  | |  |  |
|  | GroupPrincipal group() | | 返回文件的组拥有者 |  |
|  | UserPrincipal owner() |  | 返回文件的拥有者 |  |
|  | Set<PosixFilePermission> permissions() | | 返回文件的权限 |  |

访问文件属性的方式有多种。第一种方式，可以通过 readAttributes()方法获取用于封装文件属性的对象，该方法是由 Files 类定义的静态方法，它的其中一种形式如下所示：

static <A extends BasicFileAttributes>

A readAttributes(Path path, Class<A> attrType, LinkOption... opts) throws IOException

这个方法返回一个指向对象的引用，该对象标识了与 path 传递的文件相关联的属性。使用 attrType 参数作为 Class 对象指定特性类型的属性。例如，为了获取基本文件属性，向 attrType

传递 BasicFileAttributes.class；对于 DOS 属性，使用 DOSFileAttributes.class；对于 POSIX 属

性，使用 PosixFileAttributes.class。可选的链接选项是通过 opts 传递的。如果没有指定该选项，就使用伴随一个符号链接。该方法返回指向所请求属性的引用。如果请求的属性类型不可用，

就会抛出 UnsupportedOperationException 异常。可以使用返回的对象访问文件属性。

访问文件属性的第二种方式是调用 Files 类定义的 getFileAttributeView()方法。NIO 定义了一些属性视图接口，包括 AttributeView、BasicFileAttributeView、DosFileAttributeView、

PosixFileAttributeView 等等。尽管在本章不会使用这些属性视图，但是在有些情况下会发现

它们很有帮助。

在有些情况下，不需要直接使用文件属性接口，因为 Files 类提供了一些访问文件属性的静态的便利方法。例如 File 类提供了 isHidden()和 isWritable()这类方法。

并不是所有的文件系统都支持所有可能的属性，理解这一点很重要。例如，DOS 文件属

**654** 第Ⅱ部分 Java 库

性应用于最初由 DOS 定义的 FAT 文件系统。广泛应用于各种文件系统的属性是由 BasicFileAttributes 接口描述的。因此，在本章的例子中使用这些属性。

20.3.5 FileSystem、FileSystems 和 FileStore 类

JDK 7 通过打包到 java.nio.file 中的 FileSystem 和 FileSystems 类，增强了访问文件系统的能力，使用这些类可以很容易地访问文件系统。实际上，使用 FileSystems 类定义的newFileSystem()方法，甚至可以获取新的文件系统。FileStore 类封装了文件存储系统。尽管

在本章中不直接使用这些类，但是您在自己的应用程序可能会发现它们很有帮助。

**20.4** **使用 NIO 系统**

下面将演示如何应用 NIO 系统完成各种任务。在开始之前有必要强调一下，随着 JDK 7

的发布，Java 对 NIO 系统进行了极大扩展。因此，NIO 系统的应用也被极大扩展了。前面提到过，这个增强版本有时被称为 NIO.2。因为 NIO.2 添加的特性是如此丰富，以至于它们改变了许多基于 NIO 的代码的编写方式，并且增加了使用 NIO 可以完成的任务类型。缘于重要性，本章剩余的大部分内容和例子都将使用 NIO.2 的新特性，因此需要 JDK 7 或更新版本。但是，在本章末尾简要介绍了 JDK 7 之前的代码。这对于使用 JDK 7 之前的环境进行工作以及维护旧代码是有帮助的。

请记住：



本章中的大部分例子都需要 JDK 7 或更新版本。

在过去，NIO 的主要目的是进行基于通道的 I/O，这到目前仍然是一个非常重要的应用。然而，可以为基于流的 I/O 以及执行文件系统操作使用 NIO。因此，对 NIO 使用的讨论分为以下 3 个部分：

* 为基于通道的 I/O 使用 NIO。
* 为基于流的 I/O 使用 NIO。
* 为路径和文件系统使用 NIO。

因为最常用的 I/O 设备是磁盘文件，所以本章剩余部分在示例中使用磁盘文件。因为所有文件通道操作都是基于字节的，所以我们将使用的缓存类型是 ByteBuffer。

在为了通过 NIO 系统进行访问而打开文件之前，必须获取描述文件的 Path 对象。完成该工作的一种方式是调用 Paths.get()工厂方法，在前面介绍过该方法。在示例中使用的 get()

方法如下所示：

static Path get(String pathname, String ... parts)

回顾一下指定路径的两种方式。可以分块传递，第 1 部分使用 pathname 传递，后续部分通过 parts 可变参数指定。另外一种方式，可以使用 pathname 指定整个路径，而不使用 parts，

示例程序将使用这种方式。

第20章 探究 NIO **655**

20.4.1 为基于通道的 I/O 使用 NIO

NIO 的重要应用是通过通道和缓存访问文件。下面演示了使用通道读取文件以及写入文

件的一些技术。

1. 通过通道读取文件

使用通道从文件读取数据有多种方式。最常用的方式可能是手动分配缓存，然后执行显式的读取操作，读取操作使用来自文件的数据加载缓存。下面首先介绍这种方式。

在能够从文件读取数据之前必须打开文件。为此，首先创建描述文件的 Path 对象，然后使用 Path 对象打开文件。根据使用文件的方式，有各种打开文件的方式。在这个例子中，将为基于字节的输入打开文件，通过显式的输入操作进行字节输入。所以，这个例子将通过调

用 Files.newByteChannel()来打开文件并建立链接到文件的通道。newByteChannel()方法的一般形式如下：

static SeekableByteChannel newByteChannel(Path path, OpenOption ... how) throws IOException

该方法返回的 SeekableByteChannel 对象封装了文件操作的通道。描述文件的 Path 对象是通过 path 传递的。参数 how 指定了打开文件的方式，因为是可变长度参数，所以可以指定

0 个或多个由逗号隔开的参数(在前面讨论过有效值，并在表 20-7 中显示了这些值)。如果没有指定参数，将为输入操作打开文件。SeekableByteChannel 是接口，用于描述能够用于文件操作的通道。FileChannel 类实现了该接口。如果使用的是默认文件系统，那么可以将返回对象强制转换成 FileChannel 类型。通道使用完之后必须关闭。既然所有通道 —— 包括

FileChannel，都实现了 AutoCloseable 接口，那么可以使用带资源的 try 语句自动关闭文件，而不必显式地调用 close()方法，在示例中将使用这种方式。

接下来，必须通过封装已经存在的数组或通过动态分配缓存来获取缓存，缓存将由通道使用。示例程序将动态分配缓存，但是可以自行选择任何一种方式。因为文件通道操作字节数组，所以将使用 ByteBuffer 定义的 allocate()方法获取缓存。该方法的一般形式如下所示：

static ByteBuffer allocate(int cap)

其中，cap 指定了缓存的容量。该方法返回对缓存的引用。

创建缓存后，在通道上调用 read()方法，传递指向缓存的引用。在此使用的 read()版本如下所示：

int read(ByteBuffer buf) throws IOException

每次调用 read()方法时，都使用来自文件的数据填充 buf 指定的缓存。读取是连续的，这意味着每次调用 read()方法都会从文件读取后续字节以填充缓存。read()方法返回实际读取的字节数量。当试图在文件末尾读取时，该方法会返回−1。

下面的程序将应用前面讨论的技术，使用显式的输入操作通过通道读取文件 test.txt：

**656** 第Ⅱ部分 Java 库

// Use Channel I/O to read a file. Requires JDK 7 or later.

import java.io.\*; import java.nio.\*;

import java.nio.channels.\*; import java.nio.file.\*;

public class ExplicitChannelRead {

public static void main(String args[]) { int count;

Path filepath = null;

* First, obtain a path to the file. try {

filepath = Paths.get("test.txt"); } catch(InvalidPathException e) {

System.out.println("Path Error " + e); return;

}

* Next, obtain a channel to that file within a try-with-resources block. try ( SeekableByteChannel fChan = Files.newByteChannel(filepath) )

{

* + Allocate a buffer.

ByteBuffer mBuf = ByteBuffer.allocate(128);

do {

// Read a buffer.

count = fChan.read(mBuf);

// Stop when end of file is reached. if(count != -1) {

* Rewind the buffer so that it can be read. mBuf.rewind();
* Read bytes from the buffer and show
* them on the screen as characters. for(int i=0; i < count; i++)

System.out.print((char)mBuf.get());

}

} while(count != -1); System.out.println();

} catch (IOException e) { System.out.println("I/O Error " + e);

}

}

}

第20章 探究 NIO **657**

下面是该程序的工作原理。首先，获取一个 Path 对象，其中包含 test.txt 文件的相对路径。将指向该对象的引用赋给 filepath。接下来调用 newByteChannel()方法，并传递 filepath 作为参数，获取链接到文件的通道。因为没有指定打开方式，所以为读取操作打开文件。注意这个通道是由带资源的 try 语句管理的对象。因此，当代码块结束时会自动关闭通道。然后该程序调用 ByteBuffer 的 allocate()方法分配缓存，当读取文件时，缓存将容纳文件的内容。指向缓存的引用保存在 mBuf 中。然后调用 read()方法将文件内容读取到 mBuf 中，读取的字节数量保存在 count 中。接下来调用 rewind()方法回绕缓存。这个调用是必需的，因为在调用 read()

方法之后，当前位置位于缓存的末尾。为了通过 get()方法读取 mBuf 中的字节，必须将当前位置重置到缓存的开头(记住，get()方法是由 ByteBuffer 定义的)。因为 mBuf 是字节缓存，所以 get()方法返回的值是字节。将它们强制转换为 char 类型，从而可以作为文本显示文件(也

可以创建将字节编码成字符的缓存，然后读取该缓存)。当到达文件末尾时，read()方法返回的值将是−1。当到达文件末尾时，自动关闭通道，结束程序。

有趣的一点是：注意程序在一个 try 代码块中获取 Path，然后使用另外一个 try 代码块获取并管理与这个路径链接的通道。尽管使用这种方式没有什么错误，但是在许多情况下，可

以对其进行简化，从而只使用必需的 try 代码块。在这种方式中，Paths.get()和 newByteChannel()

方法调用被连接到一起。例如，下面是对该程序进行改写后的版本，该版本使用这种方式：

// A more compact way to open a channel. Requires JDK 7 or later.

import java.io.\*; import java.nio.\*;

import java.nio.channels.\*; import java.nio.file.\*;

public class ExplicitChannelRead {

public static void main(String args[]) { int count;

* Here, the channel is opened on the Path returned by Paths.get().
* There is no need for the filepath variable.

try ( SeekableByteChannel fChan = Files.newByteChannel(Paths.get("test.txt")) )

{

// Allocate a buffer.

ByteBuffer mBuf = ByteBuffer.allocate(128);

do {

// Read a buffer.

count = fChan.read(mBuf);

// Stop when end of file is reached. if(count != -1) {

// Rewind the buffer so that it can be read.

**658** 第Ⅱ部分 Java 库

mBuf.rewind();

* Read bytes from the buffer and show
* them on the screen as characters. for(int i=0; i < count; i++)

System.out.print((char)mBuf.get());

}

} while(count != -1); System.out.println();

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch (IOException e) {

System.out.println("I/O Error " + e);

}

}

}

在这个版本中，不再需要 filepath 变量，并且两个异常都通过相同的 try 语句进行处理。因为这种方式更紧凑，所以在本章的其他例子中也将使用这种方式。当然，在您自己的代码中可能会遇到以下这种情况：不能将 Path 对象的创建和通道的获取放到一起。对于这种情况，可以使用前一种方式。

读取文件的另外一种方式是将文件映射到缓存。这种方式的优点是缓存自动包含文件的内容，不需要显式的读操作。为了映射和读取文件内容，需要遵循以下一般过程。首先，像前面介绍的那样，获取用于封装文件的 Path 对象。接下来调用 Files.newByteChannel()方法，并传递获取的 Path 对象作为参数，获取链接到文件的通道，然后将返回的对象转换成

FileChannel 类型。如前所述，newByteChannel()方法返回 SeekableByteChannel 类型的对象。

当使用默认文件系统时，可以将这个对象转换成 FileChannel 类型。然后，在通道上调用 map() 方法，将通道映射到缓存。map()方法是由 FileChannel 定义的，所以需要将返回的对象转换

成 FileChannel 类型。map()方法如下所示：

MappedByteBuffer map(FileChannel.MapMode how,

long pos, long size) throws IOException

map()方法导致将文件中的数据映射到内存中的缓存。参数 how 的值决定了允许的操作

类型，必须是以下这些值中的一个：

MapMode.READ\_ONLY MapMode.READ\_WRITE MapMode.PRIVATE

对 于 读 取 文 件 ， 使 用 MapMode.READ\_ONLY 。 要 读 取 并 写 入 文 件 ， 使 用

MapMode.READ\_WRITE。MapMode.PRIVATE 导致创建文件的私有副本，并且对缓存的修改

不会影响底层的文件。文件中开始进行映射的位置是由 pos 指定的，并且映射的字节数量是

由 size 指定的。作为 MappedByteBuffer 返回指向缓存的引用，MappedByteBuffer 是 ByteBuffer

的子类。一旦将文件映射到缓存，就可以从缓存读取文件了。下面是演示这种方式的一个例子：

// Use a mapped file to read a file. Requires JDK 7 or later.

第20章 探究 NIO **659**

import java.io.\*; import java.nio.\*;

import java.nio.channels.\*; import java.nio.file.\*;

public class MappedChannelRead {

public static void main(String args[]) {

// Obtain a channel to a file within a try-with-resources block. try ( FileChannel fChan =

(FileChannel) Files.newByteChannel(Paths.get("test.txt")) )

{

* Get the size of the file. long fSize = fChan.size();
* Now, map the file into a buffer.

MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ\_ONLY, 0, fSize); // Read and display bytes from buffer.

for(int i=0; i < fSize; i++) System.out.print((char)mBuf.get());

System.out.println();

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch (IOException e) {

System.out.println("I/O Error " + e);

}

}

}

在这个程序中，首先创建链接到文件的 Path 对象，然后通过 newByteChannel()方法打开文件。通道被转换成 FileChannel 类型并保存在 fChan 中。接下来，调用 size()方法以获取文件的大小。然后，调用 map()方法将整个文件映射到 fChan，并将指向缓存的引用保存到 mBuf

中。注意 mBuf 被声明为指向 MappedByteBuffer 的引用。最后通过 get()方法读取 mBuf 中的字节。

2. 通过通道写入文件

与读取文件一样，使用通道将数据写入文件的方式也有多种。首先介绍最常用的一种。在这种方式中，手动分配缓存，将数据写入到缓存中，然后执行显式的写操作，将数据写入到文件中。

在向文件中写入数据之前，必须打开文件。为此，首先获取描述文件的 Path 对象，然后使用 Path 对象打开文件。在这个例子中，将为进行基于字节的输出打开文件，通过显式的输出操作写入数据。所以，这个例子调用 Files.newByteChannel()方法来打开文件，并建立链接到文件的通道。正如前面所显示的，newByteChannel()方法的一般形式如下：

static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)

**660** 第Ⅱ部分 Java 库

throws IOException

该方法返回的 SeekableByteChannel 对象中封装了用于文件操作的通道。为了针对输入操

作打开文件，how 参数必须为 StandardOpenOption.WRITE。当文件不存在时，如果希望创建文件，就必须指定 StandardOpenOption.CREATE(也可以使用表 20-7 中显示的其他选项)。在前面解释过，SeekableByteChannel 是接口，用于描述能够用于文件操作的通道。FileChannel

类实现了该接口。如果使用的是默认文件系统，可以将返回对象转换成 FileChannel 类型。当通道使用完之后必须关闭。

下面是通过通道写入文件的一种方式，这种方式显式调用 write()方法。首先，调用newByteChannel()方法以获取与文件链接的 Path 对象，然后打开文件，将返回的结果转换成

FileChannel 类型。接下来分配字节缓存，并将数据写入到缓存中。在将数据写入到文件之前，在缓存上调用 rewind()方法，将当前位置设置为 0(在缓存上的每次输出操作都会增加当前位置。因此在写入文件之前，必须重置当前位置)。然后，对通道调用 write()方法，传递缓存。下面的程序演示了这个过程。该程序将字母表写入到 test.txt 文件中。

// Write to a file using NIO. Requires JDK 7 or later.

import java.io.\*; import java.nio.\*;

import java.nio.channels.\*; import java.nio.file.\*;

public class ExplicitChannelWrite { public static void main(String args[]) {

// Obtain a channel to a file within a try-with-resources block. try ( FileChannel fChan = (FileChannel)

Files.newByteChannel(Paths.get("test.txt"),

StandardOpenOption.WRITE, StandardOpenOption.CREATE) )

{

// Create a buffer.

ByteBuffer mBuf = ByteBuffer.allocate(26);

* + Write some bytes to the buffer. for(int i=0; i<26; i++)

mBuf.put((byte)('A' + i));

* + Reset the buffer so that it can be written. mBuf.rewind();
  + Write the buffer to the output file. fChan.write(mBuf);

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

第20章 探究 NIO **661**

} catch (IOException e) { System.out.println("I/O Error: " + e); System.exit(1);

}

}

}

该程序有一个重要方面值得强调。如前所述，在将数据写入 mBuf 中之后，但是在将数据写入文件中之前，对 mBuf 调用 rewind()方法。在将数据写入 mBuf 中之后，为了将当前位置重置为 0，这是必须做的。请记住，每次对 mBuf 调用 put()方法都会向前推进当前位置。所以在调用 write()方法之前，需要将当前位置重置到缓存的开头。如果没有这么做，write()

方法会认为缓存中没有数据。

在输入和输出操作之间，处理缓存重置的另外一种方式是调用 flip()方法而不是调用rewind()方法。flip()方法将当前位置设置为 0，并将界限设置为当前位置之前。在前面的示例中，因为缓存的容量等于界限，所以可以使用 flip()方法而不是 rewind()方法。然而，并不是在所有情况下都可以互换这两个方法。

通常，在读和写操作之间必须重置缓存。例如，对于前面的例子下面的循环会将字母表写入文件 3 次。请特别注意对 rewind()方法的调用。

for(int h=0; h<3; h++) {

* Write some bytes to the buffer. for(int i=0; i<26; i++)

mBuf.put((byte)('A' + i));

* Rewind the buffer so that it can be written. mBuf.rewind();
* Write the buffer to the output file. fChan.write(mBuf);
* Rewind the buffer so that it can be written to again. mBuf.rewind();

}

注意在每次读和写操作之间都要调用 rewind()方法。

关于该程序需要注意的另外一点是：当将缓存写入文件时，文件中的前 26 个字节将包含输出。如果文件 test.txt 先前就已经存在，那么在执行程序后，test.txt 中的前 26 个字节将包含字母表，但是文件的剩余部分会保持不变。

写入文件的另外一种方式是将文件映射到缓存。这种方式的优点是：写入到缓存中的数据会被自动写入到文件中，不需要显式的写操作。为了映射和写入文件内容，需要使用以下一般过程。首先，获取封装文件的 Path 对象，然后调用 Files.newByteChannel()方法，传递获取的 Path 对象作为参数，创建链接到文件的通道。将 newByteChannel()方法返回的引用转换成 FileChannel 类型。接下来对通道调用 map()方法将通道映射到缓存。在前面已经详细描述了 map()方法。为了方便起见，在此对其进行总结。下面是 map()方法的一般形式：

**662** 第Ⅱ部分 Java 库

MappedByteBuffer map(FileChannel.MapMode how,

long pos, long size) throws IOException

map()方法导致文件中的数据被映射到内存中的缓存。how 的值决定了允许的操作类型。为了写入文件，how 必须是 MapMode.READ\_WRITE。文件中开始映射的位置是由 pos 指定的，映射的字节数量是由 size 决定的。map()方法返回指向缓存的引用。一旦将文件映射到缓存，就可以向缓存写入数据，并且这些数据会被自动写入到文件中。所以，不需要对通道执行显式的写入操作。

下面的程序对前面的程序进行了改写，从而使用映射文件。注意在 newByteChannel()方法调用中，添加了 StandardOpenOption.READ 打开选项。这是因为映射文件要么是只读的，要么是读/写的。因此，为了向映射文件中写入数据，必须以读/写模式打开通道。

// Write to a mapped file. Requires JDK 7 or later.

import java.io.\*; import java.nio.\*;

import java.nio.channels.\*; import java.nio.file.\*;

public class MappedChannelWrite {

public static void main(String args[]) {

// Obtain a channel to a file within a try-with-resources block. try ( FileChannel fChan = (FileChannel)

Files.newByteChannel(Paths.get("test.txt"),

StandardOpenOption.WRITE,

StandardOpenOption.READ, StandardOpenOption.CREATE) )

{

// Then, map the file into a buffer.

MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ\_WRITE, 0, 26);

// Write some bytes to the buffer. for(int i=0; i<26; i++)

mBuf.put((byte)('A' + i));

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch (IOException e) { System.out.println("I/O Error " + e);

}

}

}

可以看出，对于通道自身没有显式的写操作。因为 mBuf 被映射到文件，所以对 mBuf

的修改会自动反映到底层的文件中。

第20章 探究 NIO **663**

3. 使用 NIO 复制文件

NIO 简化了好几种类型的文件操作。尽管不可能对所有这些操作进行分析，不过可以通过一个例子提供这一思想。下面的程序调用 NIO 的 copy()方法来复制文件，该方法是由 File

类定义的静态方法。Copy()方法有好几种形式，下面是我们将使用的其中一种形式：

static Path copy(Path src, Path dest, CopyOption ... how) throws IOException

这种形式将 src 指定的文件复制到 dest 指定的文件。执行复制的方式是由 how 指定的。因为是可变长度参数，所以可以省略。如果指定 how 参数，那么参数值可以是下面这些值中的一个或多个，对于所有文件系统这些值都是合法的。

* StandardCopyOption.COPY\_ATTRIBUTES：要求复制文件的属性。
* StandardCopyOption.NOFOLLOW\_LINKS：不使用符号链接。
* StandardCopyOption.REPLACE\_EXISTING：覆盖先前存在的文件。

根据具体的实现，也可能支持其他选项。

下面的程序演示了 copy()方法。源文件和目标文件都是在命令行上指定的，首先指定源文件。注意这个程序是多么简短！您可能希望将这个版本的文件复制程序与第 13 章中的版本进行比较。可以发现在使用 NIO 的版本中，实际复制文件的程序部分明显缩短了，如下所示：

// Copy a file using NIO. Requires JDK 7 or later. import java.io.\*;

import java.nio.\*;

import java.nio.channels.\*; import java.nio.file.\*;

public class NIOCopy {

public static void main(String args[]) {

if(args.length != 2) { System.out.println("Usage: Copy from to"); return;

}

try {

Path source = Paths.get(args[0]); Path target = Paths.get(args[1]);

// Copy the file.

Files.copy(source, target, StandardCopyOption.REPLACE\_EXISTING);

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch (IOException e) {

System.out.println("I/O Error " + e);

}

}

**664** 第Ⅱ部分 Java 库

}

20.4.2 为基于流的 I/O 使用 NIO

从 JDK 7 开始，可以使用 NIO 打开 I/O 流。如果拥有 Path 对象，那么可以通过调用newInputStream()或 newOutputStream()方法来打开文件，它们是由 File 类定义的静态方法。这些方法返回链接到指定文件的流。对于这两种情况，可以使用在第 19 章中描述的方法来操作流，并且应用相同的技术。使用 Path 对象打开文件的优点是：NIO 定义的所有属性都可以使用。

为了针对基于流的输入操作打开文件，可以使用 Files.newInputStream()方法，该方法的一般形式如下所示：

static InputStream newInputStream(Path path, OpenOption ... how) throws IOException

其中，path 指定了要打开的文件，how 指定了打开文件的方式，how 参数的值必须是一个或多个由 StandardOpenOption 定义的值，在前面描述了这些值(当然，只能应用与输入流相关的选项)。如果没有指定选项，那么文件的打开方式为 StandardOpenOption.READ。

一旦打开文件，就可以使用 InputStream 定义的任何方法。例如，可以使用 read()方法从文件读取字节。

下面的程序演示了基于 NIO 的流 I/O 的使用。该程序对第 13 章中的 ShowFile 程序进行了重写，从而使用 NIO 特性打开文件并获取流。可以看出，除了使用 Path 对象和newInputStream()方法外，该版本与原始版本很类似。

/\* Display a text file using stream-based, NIO code. Requires JDK 7 or later.

To use this program, specify the name of the file that you want to see.

For example, to see a file called TEST.TXT, use the following command line.

java ShowFile TEST.TXT

\*/

import java.io.\*; import java.nio.file.\*;

class ShowFile {

public static void main(String args[])

{

int i;

// First, confirm that a filename has been specified. if(args.length != 1) {

第20章 探究 NIO **665**

System.out.println("Usage: ShowFile filename"); return;

}

// Open the file and obtain a stream linked to it.

try ( InputStream fin = Files.newInputStream(Paths.get(args[0])) )

{

do {

i = fin.read();

if(i != -1) System.out.print((char) i);

} while(i != -1);

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch(IOException e) { System.out.println("I/O Error " + e);

}

}

}

因为 newInputStream()方法返回的流是常规流，所以可以像所有其他流那样使用。例如，为了提供缓存，可以在缓存流(如 BufferedInputStream)中封装流，如下所示：

new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))

现在，读取的所有数据都会被自动缓存。

为了针对输出操作打开文件，可以使用 Files.newOutputStream()方法。该方法如下所示：

static OutputStream newOutputStream(Path path, OpenOption ... how) throws IOException

其中，path 指定了要打开的文件，how 指定了打开文件的方式，how 参数的值必须是 StandardOpenOption 定义的一个或多个值，在前面描述了这些值(当然，只能应用那些与输出

流 相 关 的 选 项 ) 。 如 果 没 有 指 定 选 项 ， 那 么 使 用 StandardOpenOption.WRITE 、 StandardOpenOption.CREATE 和 StandardOpenOption.TRUNCATE\_EXISTING 打开文件。

使用 newOutputStream()方法的方式与前面显示的使用 newInputStream()方法的方式类似。一旦打开文件，就可以使用 OutputStream 定义的任何方法了。例如，可以使用 write()方法将字节写入到文件中。为了缓存流，还可以在 BufferedOutputSteam 对象中封装流。

下面的程序显示了 newOutputStream()方法的使用。该程序将字母表写入到 test.txt 文件中。注意缓存 I/O 的使用。

// Demonstrate NIO-based, stream output. Requires JDK 7 or later.

import java.io.\*; import java.nio.file.\*;

**666** 第Ⅱ部分 Java 库

class NIOStreamWrite {

public static void main(String args[])

{

// Open the file and obtain a stream linked to it. try ( OutputStream fout =

new BufferedOutputStream( Files.newOutputStream(Paths.get("test.txt"))) )

{

// Write some bytes to the stream. for(int i=0; i < 26; i++)

fout.write((byte)('A' + i));

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch(IOException e) { System.out.println("I/O Error: " + e);

}

}

}

20.4.3 为路径和文件系统操作使用 NIO

在第 19 章的开头分析了 java.io 包中的 File 类。在那儿解释过，File 类处理文件系统以及与文件关联的各种属性，例如文件是否是只读的、隐藏的等等。还可以使用 File 类获取与文件路径相关的信息。随着 JDK 7 的出现，NIO.2 定义的接口和类为执行这些功能提供了更好的方式。优点包括支持符号链接、能更好地支持目录树遍历以及改进的元数据处理等等。下面将显示这两种通用文件系统操作的一些例子：获取与路径和文件相关的信息以及获取目录的内容。

请记住：



如果希望将使用 java.io.file 的旧代码更新为使用 Path 接口，可以使用 toPath() 方法从 File 实例获取 Path 实例。toPath()方法是由 JDK 7 为 File 类新增的。

1. 获取与路径和文件相关的信息

可以使用 Path 定义的方法获取与路径相关的信息。有些与 Path 对象描述的文件相关的信息，可以使用 Files 类定义的方法来获取。在此使用的由 Path 接口定义的方法有 getName()、 getParent() 和 toAbsolutePath() 。 File 类提供的那些方法是 isExecutable() 、 isHidden() 、isReadable()、isWritable()和 exists()。如前所示，在表 20-5 和表 20-6 中对这些方法进行了总结。

警告：



使用 isExecutable()、isHidden()、isReadable()、isWritable()和 exists()这类方法时必须小心，因为在调用这些方法之后，文件系统的状态可能会改变。在这种情况下，程序可能会发生故障。这类情况可能暗示存在安全性问题。

第20章 探究 NIO **667**

其他文件属性可以通过调用 Files.readAttributes()方法，请求文件属性列表来进行获取。在下面的程序中，调用这个方法获取与文件关联的 BasicFileAttributes，但是也可以将这种通用方式应用到其他类型的文件属性。

下面的程序演示了 Path 接口和 Files 类定义的一些方法，以及 BasicFileAttributes 提供的一些方法。这个程序假定文件 test.txt 位于 examples 目录中，examples 目录必须是当前目录的子目录。

* Obtain information about a path and a file.
* Requires JDK 7 or later.

import java.io.\*; import java.nio.file.\*;

import java.nio.file.attribute.\*;

class PathDemo {

public static void main(String args[]) {

Path filepath = Paths.get("examples\\test.txt"); System.out.println("File Name: " + filepath.getName(1)); System.out.println("Path: " + filepath); System.out.println("Absolute Path: " + filepath.toAbsolutePath()); System.out.println("Parent: " + filepath.getParent());

if(Files.exists(filepath)) System.out.println("File exists");

else

System.out.println("File does not exist");

try { if(Files.isHidden(filepath))

System.out.println("File is hidden"); else

System.out.println("File is not hidden"); } catch(IOException e) {

System.out.println("I/O Error: " + e);

}

Files.isWritable(filepath);

System.out.println("File is writable");

Files.isReadable(filepath);

System.out.println("File is readable");

try {

BasicFileAttributes attribs = Files.readAttributes(filepath, BasicFileAttributes.class);

if(attribs.isDirectory()) System.out.println("The file is a directory");

**668** 第Ⅱ部分 Java 库

else

System.out.println("The file is not a directory");

if(attribs.isRegularFile()) System.out.println("The file is a normal file");

else

System.out.println("The file is not a normal file");

if(attribs.isSymbolicLink()) System.out.println("The file is a symbolic link");

else

System.out.println("The file is not a symbolic link");

System.out.println("File last modified: " + attribs.lastModifiedTime());

System.out.println("File size: " + attribs.size() + " Bytes"); } catch(IOException e) {

System.out.println("Error reading attributes: " + e);

}

}

}

如果从 MyDir 目录执行这个程序，将会看到与下面类似的输出。MyDir 含有子目录

examples，并且 examples 目录包含 test.txt 文件(当然，文件的大小和时间可能不同)。

File Name: test.txt

Path: examples\test.txt

Absolute Path: C:\MyDir\examples\test.txt

Parent: examples

File exists

File is not hidden

File is writable

File is readable

The file is not a directory

The file is a normal file

The file is not a symbolic link

File last modified: 2010-09-01T18:20:46.380445Z

File size: 18 Bytes

如果使用的计算机支持 FAT 文件系统(比如 DOS 文件系统)，那么您可能希望尝试使用 DosFileAttributes 定义的方法。如果使用的是与 POSIX 兼容的系统，那么可以尝试使用

PosixFileAttributes。

2. 列出目录的内容

如果路径描述的是目录，那么可以使用 Files 类定义的静态方法来读取目录的内容。为此，

首先调用 newDirectoryStream()方法以获取目录流，传递描述目录的 Path 对象作为参数。newDirectoryStream()方法的其中一种形式如下所示：

第20章 探究 NIO **669**

static DirectoryStream<Path> newDirectoryStream(Path dirPath) throws IOException

其中，dirPath 封装了链接到目录的路径。该方法返回一个 DirectoryStream<Path>对象，可以使用该对象获取目录的内容。如果发生 I/O 错误，该方法会抛出 IOException 异常，并且

如果指定的路径不是目录，那么还会抛出 NotDirectoryException 异常(NotDirectoryException是 IOException 的子类)。如果不允许访问目录，还可能抛出 SecurityException 异常。

DirectoryStream<Path>实现了 AutoCloseable 接口，所以可以使用带资源的 try 语句进行

管理。另外还实现了 Iterable<Path>，这意味着可以通过迭代 DirectoryStream 对象来获取目录的内容。进行迭代时，每个目录实体都是由一个 Path 实例表示的。迭代 DirectoryStream 对象的一种简单方式是使用 for-each 风格的 for 循环。但是，DirectoryStream<Path>实现的迭代器，针对每个实例只能获取一次，理解这一点很重要。因此，iterator()方法只能调用一次，并且 for-each 循环只能执行一次。

下面的程序显示了 MyDir 目录的内容：

// Display a directory. Requires JDK 7 or later.

import java.io.\*; import java.nio.file.\*;

import java.nio.file.attribute.\*;

class DirList {

public static void main(String args[]) { String dirname = "\\MyDir";

// Obtain and manage a directory stream within a try block. try ( DirectoryStream<Path> dirstrm =

Files.newDirectoryStream(Paths.get(dirname)) )

{

System.out.println("Directory of " + dirname);

* Because DirectoryStream implements Iterable, we
* can use a "foreach" loop to display the directory. for(Path entry : dirstrm) {

BasicFileAttributes attribs =

Files.readAttributes(entry, BasicFileAttributes.class);

if(attribs.isDirectory()) System.out.print("<DIR> ");

else

System.out.print(" ");

System.out.println(entry.getName(1));

}

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch(NotDirectoryException e) { System.out.println(dirname + " is not a directory.");

} catch (IOException e) {

**670** 第Ⅱ部分 Java 库

System.out.println("I/O Error: " + e);

}

}

}

下面是该程序的样本输出：

Directory of \MyDir DirList.class DirList.java

<DIR> examples Test.txt

可以使用两种方式过滤目录的内容。最简单的方式是使用下面这个版本的newDirectoryStream()方法：

static DirectoryStream<Path> newDirectoryStream(Path dirPath, String wildcard) throws IOException

在这个版本中，只能获取与通配符文件名相匹配的方法。其中，通配符文件名是由 wildcard 指定的。对于 wildcard，可以指定完整的文件名，也可以指定 glob。glob 是定义通用模式的字符串，通用模式使用熟悉的“\*”和“？”通配符来匹配一个或多个文件。通配符“\*”

匹配 0 个或多个任意字符，通配符“？”匹配任意一个字符。在 glob 中还能识别表 20-11 中

的通配符。

表 20-11 在 glob 中还能识别的通配符

|  |  |  |
| --- | --- | --- |
| 通 配 符 | 描 | 述 |

* 匹配 0 个或多个跨越目录的任意字符

匹配 chars 中的任意一个字符。chars 中的\*或？被看作常规字符而不是通配符。通过使用连字

[chars]

符，可以指定某个范围，例如[x-z]

[globlist] 匹配一组 glob 中的任意一个 glob，这些 glob 是由 globlist 指定的由逗号隔开的 glob 列表

使用“\\*”和“\？”可以指定一个“\*”或“？”字符。为了指定“\”字符，需要使用“\\”。可以将下面这个 newDirectoryStream()方法调用替换到前面的程序中来进行试验：

Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}\*.{java,class}")

这个调用获取的目录流只包含名称由“Path”或“Dir”开头、并且扩展名为“java”或

“class”的文件。因此，能够匹配 DirList.java 和 PathDemo.java 这类名称，但是不能匹配MyPathDemo.java。

过滤目录的另外一种方式是使用下面这个版本的 newDirectoryStream()方法：

static DirectoryStream<Path> newDirectoryStream(Path dirPath, DirectoryStream.Filter<? super Path> filefilter)

第20章 探究 NIO **671**

throws IOException

其中，DirectoryStream.Filter 是定义了以下方法的接口：

boolean accept(T entry) throws IOException

在这个方法中，T 将会是 Path 类型。如果希望在列表中包含 entry，就返回 true；否则返回 false。这种形式的 newDirectoryStream()方法的优点是：可以基于文件名之外的其他内容过滤目录。例如，可以根据文件大小、创建日期、修改日期或属性进行过滤。

下面的程序演示了这个过程，该程序只列出可写的文件。

// Display a directory of only those files that are writable.

import java.io.\*; import java.nio.file.\*;

import java.nio.file.attribute.\*;

class DirList {

public static void main(String args[]) { String dirname = "\\MyDir";

* Create a filter that returns true only for writable files. DirectoryStream.Filter<Path> how = new DirectoryStream.Filter<Path>() {

public boolean accept(Path filename) throws IOException { if(Files.isWritable(filename)) return true;

return false;

}

};

* Obtain and manage a directory stream of writable files.

try (DirectoryStream<Path> dirstrm = Files.newDirectoryStream(Paths.get(dirname), how) )

{

System.out.println("Directory of " + dirname);

for(Path entry : dirstrm) { BasicFileAttributes attribs =

Files.readAttributes(entry, BasicFileAttributes.class);

if(attribs.isDirectory()) System.out.print("<DIR> ");

else

System.out.print(" ");

System.out.println(entry.getName(1));

}

} catch(InvalidPathException e) { System.out.println("Path Error " + e);

} catch(NotDirectoryException e) { System.out.println(dirname + " is not a directory.");

} catch (IOException e) {

**672** 第Ⅱ部分 Java 库

System.out.println("I/O Error: " + e);

}

}

}

3. 使用 walkFileTree()列出目录树

前面的例子只获取单个目录的内容。然而，有时会希望获取目录树中的文件列表。在过去，这项工作很繁杂，但 NIO.2 使完成这项工作变得容易，因为现在可以使用 File 类定义的walkFileTree()方法来处理目录树。该方法有两种形式，在本章中使用的形式如下所示：

static Path walkFileTree(Path root, FileVisitor<? extends Path> fv) throws IOException

对于要遍历的目录，起始位置的路径是由 root 传递的。fv 传递 FileVisitor 实例。FileVisitor

的实现决定了如何遍历目录树，并且支持访问目录信息。如果发生 I/O 错误，会抛出

IOException 异常，也可能抛出 SecurityException 异常。

FileVisitor 是定义遍历目录时访问文件方式的接口。FileVisitor 是泛型接口，其声明如下

所示：

interface FileVisitor<T>

为了能在 walkFileTree()中使用，T 需要是 Path 类型(或派生自 Path 的任意类型)。FileVisitor

接口定义了表 20-12 中所示的方法。

表 20-12 FileVisitor 接口定义的方法

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 方 | 法 | 描 | 述 |  |
|  |  |  |  | |  |
|  | FileVisitResultpostVisitDirectory(T dir, IOException exc) | | 在访问目录之后调用。目录被传递给 dir ，任何 | |  |
|  | IOException 异常都会被传递给 exc。如果 exc 为 null， | |  |
|  | throws IOException |  |  |
|  |  | 就表示没有发生异常。结果被返回 | |  |
|  |  |  |  |
|  |  | |  | |  |
|  | FileVisitResultpreVisitDirectory(T dir, | | 在访问目录之前调用。目录被传递给 dir，与目录关 | |  |
|  |  | BasicFileAttributes attribs) | 联的属性被传递给 attribs。结果被返回。为了继续检 | |  |
|  | throws IOException |  | 查目录，返回 FileVisitResult.CONTINUE | |  |
|  | FileVisitResult visitFile(T file, | | 当访问文件时调用。文件被传递给 file，与文件关联 | |  |
|  | BasicFileAttributes attribs) | |  |
|  | 的属性被传递给 attribs。结果被返回 | |  |
|  | throws IOException |  |  |
|  |  |  |  |  |
|  | FileVisitResultvisitFileFailed(T file, IOException exc) | | 当尝试访问文件失败时调用。访问失败的文件由 file | |  |
|  | throws IOException |  | 传递，IOException 异常由 exc 传递。结果被返回 | |  |

注意每个方法都返回 FileVisitResult 枚举对象。这个枚举定义了以下值：

CONTINUE SKIP\_SIBLINGS SKIP\_SUBTREE TERMINATE

第20章 探究 NIO **673**

通常，为了继续遍历目录和子目录，方法应当返回 CONTINUE。对于 preVisitDirectory()，为了绕过目录及其兄弟目录并阻止调用 postVisitDirectory()方法，会返回 SKIP\_SIBLINGS；

为了只绕过目录及其子目录，返回 SKIP\_SUBTREE；为了停止目录遍历，返回 TERMINATE。

尽管完全可以通过实现 FileVisitor 定义的这些方法来创建自己的访问器，但是通常不会这么做，因为 SimpleFileVisitor 已经提供了一个简单实现。可以只重写感兴趣方法的默认实现。下面的简单示例演示了这个过程。该例显示目录中以“\MyDir”作为根目录的所有文件。注意这个程序是多么简短！

* A simple example that uses walkFileTree( ) to display a directory tree.
* Requires JDK 7 or later.

import java.io.\*; import java.nio.file.\*;

import java.nio.file.attribute.\*;

* Create a custom version of SimpleFileVisitor that overrides
* the visitFile( ) method.

class MyFileVisitor extends SimpleFileVisitor<Path> {

public FileVisitResult visitFile(Path path, BasicFileAttributes attribs) throws IOException

{

System.out.println(path); return FileVisitResult.CONTINUE;

}

}

class DirTreeList {

public static void main(String args[]) { String dirname = "\\MyDir";

System.out.println("Directory tree starting with " + dirname + ":\n");

try {

Files.walkFileTree(Paths.get(dirname), new MyFileVisitor()); } catch (IOException exc) {

System.out.println("I/O Error");

}

}

}

下面是当在前面显示的相同的 MyDir 目录下执行程序时生成的样本输出。在这个例子中，

子目录 examples 包含文件 MyProgram.java。

Directory tree starting with \MyDir:

\MyDir\DirList.class

\MyDir\DirList.java

**674** 第Ⅱ部分 Java 库

\MyDir\examples\MyProgram.java

\MyDir\Test.txt

在这个程序中，MyFileVisitor 扩展了 SimpleFileVisitor，并且只重写了 visitFile()方法。在这个例子中，visitFile()方法简单地显示文件，但是更复杂的功能也很容易实现。例如，可以过滤文件或对文件执行操作，比如将它们复制到备份设备。为了清晰起见，使用命名的类重写 visitFile()方法，但是也可以使用匿名的内部类。

最后一点：可以使用 java.nio.file.WatchService 来观察目录的变化。

**20.5** **JDK 7 之前基于通道的例子**

在结束本章之前，需要介绍 NIO 的另外一个方面。前面使用了 JDK 7 添加到 NIO 的一些新特性。然而，仍然存在许多 JDK 7 之前的代码需要维护，或者可能需要对它们进行转换以使用新特性。为此，下面显示使用 JDK 7 之前的 NIO 系统读取和写入文件的方式。它们重写前面显示的一些例子，从而使用原始的 NIO 特性，而不是新的 JDK 7(NIO.2)特性。这意味着本节中的例子可以使用 JDK 7 之前的 Java 版本进行工作。

JDK 7 之前的代码和新的 NIO 代码之间的关键区别是 Path 接口，该接口是由 JDK 7 新增的。因此，JDK 7 之前的代码不能使用 Path 对象描述文件，也不能为之打开通道。此外，JDK 7 之前的代码不能使用带资源的 try 语句，因为自动资源管理特性也是由 JDK 7 新增的。

请记住：



本节中的例子描述了遗留 NIO 代码的工作方式。本节完全是为了使用 JDK 7

之前的代码或使用 JDK 7 以前版本编译器的程序员。新代码应当使用 JDK 7 新增的 NIO 特性。

20.5.1 读文件(JDK 7 之前)

下面重写前面显示的两个基于通道的文件输入示例，从而只使用 JDK 7 之前的特性。第一个例子通过手动分配缓存，然后执行显式的读操作来读取文件。第二个例子使用映射文件，自动进行读文件过程。

如果使用 JDK 7 之前的 Java 版本，当使用通道和手动分配的缓存读文件时，首先需要使用在第 19 章解释的相同机制，使用 FileInputStream 为读操作打开文件。接下来，通过对 FileInputStream 对象调用 getChannel()方法，获取链接到这个文件的通道。该方法的一般形式

如下所示：

FileChannel getChannel( )

这个方法返回一个 FileChannel 对象，该对象为文件操作封装了通道。然后，调用 allocate() 方法以分配缓存。因为文件通道操作字节缓存，所以需要使用 ByteBuffer 定义的 allocate()方

法，在前面介绍过该方法的工作方式。

下面的程序显示了 JDK 7 之前的 Java 版本，如何使用显式输入操作的通道读取并显示

第20章 探究 NIO **675**

test.txt 文件的内容：

// Use Channels to read a file. Pre-JDK 7 version. import java.io.\*;

import java.nio.\*;

import java.nio.channels.\*;

public class ExplicitChannelRead {

public static void main(String args[]) { FileInputStream fIn = null; FileChannel fChan = null;

ByteBuffer mBuf; int count;

try {

// First, open a file for input.

fIn = new FileInputStream("test.txt");

* Next, obtain a channel to that file. fChan = fIn.getChannel();
* Allocate a buffer.

mBuf = ByteBuffer.allocate(128);

do {

// Read a buffer.

count = fChan.read(mBuf);

// Stop when end of file is reached. if(count != -1) {

* Rewind the buffer so that it can be read. mBuf.rewind();
* Read bytes from the buffer and show
* them on the screen.

for(int i=0; i < count; i++) System.out.print((char)mBuf.get());

}

} while(count != -1);

System.out.println();

} catch (IOException e) { System.out.println("I/O Error " + e);

} finally {

try {

if(fChan != null) fChan.close(); // close channel } catch(IOException e) {

**676** 第Ⅱ部分 Java 库

System.out.println("Error Closing Channel.");

}

try {

if(fIn != null) fIn.close(); // close file } catch(IOException e) {

System.out.println("Error Closing File.");

}

}

}

}

在这个程序中，注意文件是使用 FileInputStream 类的构造函数打开的，并且将指向这种对象的引用赋给 fIn。接下来对 fIn 调用 getChannel()方法，获取连接到该文件的通道。之后，该程序就与前面显示的 JDK 7 版本的程度类似了。总结一下：程序调用 ByteBuffer 的 allocate()

方法以分配缓存，当读取文件时，该缓存用于容纳文件的内容。之所以使用字节缓存，是因为 FileChannel 操作字节。指向这个缓存的引用保存在 mBuf 中。然后调用 read()方法，将文件内容读取到 mBuf 中。读取的字节数量保存在 count 中。接下来调用 rewind()方法以回绕缓存。这个调用是必需的，因为在执行完 read()方法后，当前位置位于缓存的末尾，为了通过get()方法读取 mBuf 缓存中的字节，必须将当前位置重置到缓存的开头。当到达文件末尾时，read()方法的返回值将是−1。当发生这种情况时，显式地关闭通道和文件，程序结束。

读取文件的另外一种方式是将文件映射到缓存。如前所述，这种方式的主要优点是：缓存会自动包含文件的内容，不需要进行显式的读操作。为了使用 JDK 7 之前的 NIO 映射和读取文件的内容，首先使用 FileInputStream 打开文件。接下来对文件对象调用 getChannel()方法，获取链接到文件的通道。然后，对 FileChannel 对象调用 map()方法，将通道映射到缓存。map()

方法的工作方式在前面已经介绍过。

下面的程序重写了前面的示例，从而只使用 JDK 7 之前的特性创建映射文件：

// Use a mapped file to read a file. Pre-JDK 7 version.

import java.io.\*; import java.nio.\*;

import java.nio.channels.\*;

public class MappedChannelRead {

public static void main(String args[]) { FileInputStream fIn = null; FileChannel fChan = null;

long fSize; MappedByteBuffer mBuf;

try {

// First, open a file for input.

fIn = new FileInputStream("test.txt");

// Next, obtain a channel to that file.

第20章 探究 NIO **677**

fChan = fIn.getChannel();

* Get the size of the file. fSize = fChan.size();
* Now, map the file into a buffer.

mBuf = fChan.map(FileChannel.MapMode.READ\_ONLY, 0, fSize);

// Read and display bytes from buffer. for(int i=0; i < fSize; i++)

System.out.print((char)mBuf.get());

} catch (IOException e) { System.out.println("I/O Error " + e);

} finally {

try {

if(fChan != null) fChan.close(); // close channel } catch(IOException e) {

System.out.println("Error Closing Channel.");

}

try {

if(fIn != null) fIn.close(); // close file } catch(IOException e) {

System.out.println("Error Closing File.");

}

}

}

}

在这个程序中，文件是使用 FileInputStream 类的构造函数打开的，并且将指向这种对象的引用赋给 fIn。通过对 fIn 调用 getChannel()方法，获取链接到文件的通道。接下来获取文件的大小。然后调用 map()方法，将整个文件映射到内存缓存，并将缓存的引用保存到 mBuf 中。通过调用 get()方法读取 mBuf 中的字节。

20.5.2 写文件(JDK 7 之前)

下面重写前面显示的两个基于通道的文件输出示例，从而只使用 JDK 7 之前的特性。第一个例子通过手动分配缓存，然后执行显式的输出操作来写入文件。第二个例子使用映射文件自动执行这个过程。对于这两个例子，都既不使用 Path 对象也不使用带资源的 try 语句。这是因为它们都是由 JDK 7 新增的部分。

如果使用 JDK 7 之前的 Java 版本，当使用通道和手动分配的缓存写文件时，首先需要针对输出打开文件。这是通过创建 FileOutputStream 对象完成的，正如在第 19 章所描述的。接下来调用 getChannel()方法，获取链接到文件的通道，然后调用 allocate()方法以分配字节缓存，就像刚才描述的那样。接下来将希望写入的输入放到缓存中，然后对通道调用 write()方法。下面的程序演示了这个过程。该程序将字母表写入到 test.txt 文件中。

// Write to a file using NIO. Pre-JDK 7 Version.

**678** 第Ⅱ部分 Java 库

import java.io.\*; import java.nio.\*;

import java.nio.channels.\*;

public class ExplicitChannelWrite { public static void main(String args[]) {

FileOutputStream fOut = null; FileChannel fChan = null; ByteBuffer mBuf;

try {

// First, open the output file.

fOut = new FileOutputStream("test.txt");

* Next, get a channel to the output file. fChan = fOut.getChannel();
* Create a buffer.

mBuf = ByteBuffer.allocate(26);

* + Write some bytes to the buffer. for(int i=0; i<26; i++)

mBuf.put((byte)('A' + i));

* + Rewind the buffer so that it can be written. mBuf.rewind();
  + Write the buffer to the output file. fChan.write(mBuf);

} catch (IOException e) { System.out.println("I/O Error " + e);

} finally {

try {

if(fChan != null) fChan.close(); // close channel } catch(IOException e) {

System.out.println("Error Closing Channel.");

}

try {

if(fOut != null) fOut.close(); // close file } catch(IOException e) {

System.out.println("Error Closing File.");

}

}

}

}

在将数据写入到 mBuf 中之后，为了将当前位置重置为 0，需要对 mBuf 调用 rewind()方

法。请记住，每次 put()调用都会向前推进当前位置。所以在调用 write()方法之前，需要将当

第20章 探究 NIO **679**

前位置重置到缓存的开头。如果没有这么做，write()方法就会认为缓存中没有数据。

如果使用 JDK 7 之前的 Java 版本，通过映射文件来写入文件，就需要遵循以下步骤。首先，通过创建 RandomAccessFile 对象为读/写操作打开文件。为了能够从文件读取数据并且能够向文件中写入数据，这是必须要做的。接下来对 RandomAccessFile 对象调用 map()方法，将文件映射到缓存。然后写入缓存。因为缓存被映射到文件，所以对缓存的任何修改都会反映到文件中，因此不需要对通道执行显式的写操作。

下面的程序对前面的程序进行了重写，从而使用映射文件：

// Write to a mapped file. Pre JDK 7 version. import java.io.\*;

import java.nio.\*;

import java.nio.channels.\*;

public class MappedChannelWrite {

public static void main(String args[]) { RandomAccessFile fOut = null; FileChannel fChan = null;

ByteBuffer mBuf;

try {

fOut = new RandomAccessFile("test.txt", "rw");

* Next, obtain a channel to that file. fChan = fOut.getChannel();
* Then, map the file into a buffer.

mBuf = fChan.map(FileChannel.MapMode.READ\_WRITE, 0, 26);

// Write some bytes to the buffer. for(int i=0; i<26; i++)

mBuf.put((byte)('A' + i));

} catch (IOException e) { System.out.println("I/O Error " + e);

} finally {

try {

if(fChan != null) fChan.close(); // close channel } catch(IOException e) {

System.out.println("Error Closing Channel.");

}

try {

if(fOut != null) fOut.close(); // close file } catch(IOException e) {

System.out.println("Error Closing File.");

}

}

}

}

**680** 第Ⅱ部分 Java 库

可以看出，没有对通道本身进行显式的写操作。因为 mBuf 被映射到文件，所以对 mBuf

的修改会自动反映到底层的文件中。