BRUCE ECKEL ON JAVA 8

版权 ©2017

作者 Bruce Eckel, President, MindView LLC.

版本号：7

ISBN 978-0-9818725-2-0

原书可在该网站购买 www.OnJava8.com

本书出版自美国，版权所有，翻版必究。未经授权不得非法存储在检索系统中，或以电子，机械，影印，录制任何形式传输等。制造商和销售商使用商标用来区分其产品标识。如果这些名称出现在这本书中，并且出版商知道商标要求，则这些名称已经用大写字母或所有大写字母打印。

Java 是甲骨文公司（Oracle. Inc.）的商标。Windows 95，Windows NT，Windows 2000，Windows XP，Windows 7，Windows 8 和 Windows 10 是微软公司（Microsoft Corporation）的商标。此处提及的所有其他产品名称和公司名称均为其各自所有者的财产。作者和出版商在编写本书时已经仔细校对过，但不作任何明示或暗示的保证，对错误或遗漏不承担任何责任。对于因使用此处包含的信息或程序而产生的偶然或间接损失，我们不承担任何责任。

这本书是以平板电脑和计算机为载体的电子书，非传统纸质版书籍。故所有布局和格式设计旨在优化您在各种电子书阅读平台和系统上的观看体验。封面由 Daniel Will-Harris 设计，www.Will-Harris.com。

译者

一起交流

交流群：721698221 OnJava8 翻译交流加群时请简单备注下来源或说明

群名称：OnJava8翻译交流
群　号：721698221

贡献者

主译：LingCoder，sjsdfg，xiangflight
参译：Langdon-Chen，1326670425，LortSir
校对：LingCoder，jason31520，xiangflight，nickChenyx

翻译说明

1. 本书排版布局和翻译风格上参考阮一峰老师的中文技术文档的写作规范采用第一人称叙述。
2. 由于中英行文差异，完全的逐字逐句翻译会很冗余啰嗦。所以本人在翻译过程中，去除了部分主题无关内容、重复描写。
3. 译者在翻译中同时参考了谷歌、百度、有道翻译的译文以及《Java 编程思想》第四版中文版的部分内容（对其翻译死板，生造名词，语言精炼度差问题进行规避和改正）。最后结合译者自己的理解进行本地化，尽量做到专业和言简意赅，方便大家更好的理解学习。
4. 由于译者个人能力、时间有限，如有翻译错误和笔误的地方，还请大家批评指正！

如何参与

如果你想对本书做出一些贡献的话。可以在阅读本书过程中帮忙校对，找 bug 错别字等等。可以提出专业方面的修改建议。可以把一些不尽人意的语句翻译的更好更有趣。对于以上各类建议，请以 issue 或 pr 的形式发送，我看到之后会尽快处理。使用 MarkDown 编辑器，md 语法格式进行文档翻译及排版工作。完成之后 PullRequest。如没问题的话，我会合并到主分支。如不熟悉 md 排版，可不必纠结，我会在合并 pr 时代为排版。如还有其它问题，欢迎发送 issue，谢谢 。

友情链接

Effective.Java.3rd.Edition 中文版

开源协议

本项目基于 MIT 协议开源。

联系方式

E-mail: lingcoder@gmail.com

前言

> 本书基于 Java 8 版本来教授当前 Java 编程的最优实践。

此前，我的另一本 Java 书籍 *Thinking in Java, 4th Edition*（《Java 编程思想》第 4 版 Prentice Hall 2006）依然适用于 Java 5 编程。Android 编程就是始于此语言版本。

随着 Java 8 的出现，这门语言在许多地方发生了翻天覆地的变化。在新的版本中，代码的运用和实现上与以往不尽相同。这也促使了我时隔多年后再次创作了这本新书。《On Java 8》旨在面向已具有编程基础的开发者们。对于初学者，可以先在 Code.org 或者 Khan Academy 等网站上补充必要的前置知识。同时，OnJava8.com 上也有免费的 Thinking in C（《C 编程思想》）专题知识。

与几年前我们依赖印刷媒体相比，YouTube，博客和 StackOverflow 等网站的出现让寻找答案变得简单。请结合这些学习途径和努力坚持下去。本书可作为编程入门书籍，同时也适用于想要扩展知识的在职程序员。每次在世界各地的演讲中，我都非常感谢《*Thinking in Java*》这本书给我带来的所有荣誉。它对于我重塑 Reinventing Business 项目和促进交流是非常宝贵的。最后，写这本书的原因之一希望这本书可以为我的这个项目众筹。似乎下一步要创建一个所谓的蓝绿色组织（Teal Organization）才合乎逻辑的。

教学目标

每章教授一个或一组相关的概念，并且这些知识不依赖于尚未学习到的章节。如此，学习者可以在当前知识的背景框架下循序渐进地掌握 Java。

本书的教学目标：

1. 循序渐进地呈现学习内容，以便于你在不依赖后置知识框架的情况下轻松完成现有的学习任务，同时尽量保证前面章节的内容在后面的学习中得到运用。如果确有必要引入我们还没学习到的知识概念，我会做个简短地介绍。
2. 尽可能地使用简单和简短的示例，方便读者理解。而不强求引入解决实际问题的例子。因为我发现，相比解决某个实际问题，读者更乐于看到自己真正理解了示例的每个细节。或许我会因为这些“玩具示例”而被一些人所诟病，但我更愿意看到我的读者们因此能保持饶有兴趣地学习。
3. 把我知道以及我认为对于你学习语言很重要的东西都告诉你。我认为信息的重要性是分层次结构的。绝大多数情况下，我们没必要弄清问题的所有本质。好比编程语言中的某些特性和实现细节，

95% 的程序员都不需要去知道。这些细节除了会加重你的学习成本，还让你更觉得这门语言好复杂。如果你非要考虑这些细节，那么它还会迷惑该代码的阅读者/维护者，所以我主张选择简单的方法解决问题。

4. 希望本书能为你打下坚实的基础，方便你将来学习更难的课程和书籍。

语言设计错误

每种语言都有设计错误。当新手程序员涉足语言特性并猜测应用场景和使用方式时，他们体验到极大的不确定性和挫折感。承认错误令人尴尬，但这种糟糕的初学者经历比认识到你错了什么还要糟糕。哎，每一种语言/库的设计错误都会永久地嵌入在 Java 的发行版中。

诺贝尔经济学奖得主约瑟夫·斯蒂格利茨（*Joseph Stiglitz*）有一套适用于这里的人生哲学，叫做“承诺升级理论”：继续犯错误的成本由别人承担，而承认错误的成本由自己承担。

看过我此前作品的读者们应该清楚，我一般倾向于指出这些错误。Java 拥有一批狂热的粉丝。他们把语言当成是阵营而不是纯粹的编程工具。我写过 Java 书籍，所以他们兀自认为我自然也是这个“阵营”的一份子。当我指出 Java 的这些错误时，会造成两种影响：

1. 早先许多错误“阵营”的人成为了牺牲品。最终，时隔多年后，大家都意识到这是个设计上的错误。然而错误已然成为 Java 历史的一部分了。
2. 更重要的是，新手程序员并没有经历过“语言为何采用某种方式实现”的争议过程。特别是那些隐约察觉不对却依然说服自己“我必须要这么做”或“我只是没学明白”从而继续错下去的人。更糟糕的是，教授这些编程知识的老师们没能深入地去研究这里是否有设计上的错误，而是继续错误的解读。总之，通过了解语言设计上的错误，能让开发者们更好地理解和意识到错误的本质，从而更快地进步。

对编程语言的设计错误理解至关重要，甚至影响程序员的开发效率。部分公司在开发过程中避免使用语言的某些功能特性。这些功能特性表面上看起来高大上，但是弄不好却可能出现意料之外的错误，影响整个开发进程。

已知的语言设计错误会给新的一门编程语言的作者提供参考。探索一门语言能做什么是很有趣的一件事，而语言设计错误能提醒你哪些“坑”是不能再趟的。多年以来，我一直感觉 Java 的设计者们有点脱离群众。Java 的有些设计错误错的太明显，我甚至怀疑设计者们到底是为出于服务用户还是其他动机设计了这些功能。Java 语言有许多臭名昭著的设计错误，很可能这也是诱惑所在。Java 似乎并不尊重开发者。为此我很长时间内不想与 Java 有任何瓜葛。很大程度上，这也是我不想碰 Java 的原因吧。

如今再审视 Java 8，我发现了许多变化。设计者们对于语言和用户的态度似乎发生了根本性上的改变。忽视用户投诉多年之后，Java 的许多功能和类库都已被搞砸了。

新功能的设计与以往有很大不同。掌舵者开始重视程序员的编程经验。新功能的开发都是在努力使语言变得更好，而非仅仅停留在快速堆砌功能而不去深入研究它们的含义。甚至有些新特性的实现方式非常优雅（至少在 Java 约束下尽可能优雅）。

我猜测可能是部分设计者的离开让他们意识到了这点。说真的，我没想到会有这些变化！因为这些原因，写这本书的体验要比以往好很多。Java 8 包含了一系列基础和重要的改进。遗憾的是，为了严格地“向后兼容”，我们不大可能看到戏剧性的变化，当然我希望我是错的。尽管如此，我很赞赏那些敢于自我颠覆，并为 Java 设定更好路线的人。第一次，对于自己所写的部分 Java 8 代码我终于可以说“赞！”

最后，本书所著时间似乎也还不错，因为 Java 8 引入的新功能已经强烈地影响了今后 Java 的编码方式。截止我在写这本书时，Java 9 似乎更专注于对语言底层的基础结构功能的重要更新，而非本书所关注的新编码方式。话说回来，得益于电子书出版形式的便捷，假如我发现本书有需要更新或添加的内容，我可以第一时间将新版本推送给现有读者。

测试用例

书中代码示例基于 Java 8 和 Gradle 编译构建，并且代码示例都保存在这个自由访问的 GitHub 的仓库 中。我们需要内置的测试框架，以便于在每次构建系统时自动运行。否则，你将无法保证自己代码的可靠性。为了实现这一点，我创建了一个测试系统来显示和验证大多数示例的输出结果。这些输出结果我会附加在示例结尾的代码块中。有时仅显示必要的那几行或者首尾行。利用这种方式来改善读者的阅读和学习体验，同时也提供了一种验证示例正确性的方法。

普及性

Java 的普及性对于其受欢迎程度有重要意义。学习 Java 会让你更容易找到工作。相关的培训材料，课程和其他可用的学习资源也很多。对于企业来说，招聘 Java 程序员相对容易。如果你不喜欢 Java 语言，那么最好不要拿他当作你谋生的工具，因为这种生活体验并不好。作为一家公司，在技术选型前一定不要单单只考虑 Java 程序员好招。每种语言都有其适用的范围，有可能你们的业务更适用于另一种编程语言来达到事半功倍的效果。如果你真的喜欢 Java，那么欢迎你。希望这本书能丰富你的编程经验！

关于安卓

本书基于 Java 8 版本。如果你是 Andriod 程序员，请务必学习 Java 5。在《On Java 8》出版的时候，我的另一本基于 Java 5 的著作 *Thinking in Java 4th Edition*（《Java 编程思想》第四版）已经可以在www.OnJava8.com上免费下载了。此外，还有许多其他专用于 Andriod 编程的资源。

电子版权声明

《On Java 8》仅提供电子版，并且仅通过 www.OnJava8.com 提供。任何未经 mindviewinc@gmail.com 授权的其他来源或流传送机构都是非法的。本作品受版权保护！未经许可，请勿通过以任何方式分享或发布。你可以使用这些示例进行教学，只要不对本书非法重新出版。有关完整详细信息，请参阅示例分发中的 Copyright.txt 文件。对于视觉障碍者，电子版本有可搜索性，字体大小调整或文本到语音等诸多好处。

任何购买这本书的读者，还需要一台计算机来运行和写作代码。另外电子版在计算机上和移动设备上的显示效果俱佳，推荐使用平板设备阅读。相比购买传统纸质版的价格，平板电脑价格都足够便宜。在床上阅读电子版比看这样一本厚厚的实体书要方便得多。起初你可能会有些不习惯，但我相信很快你就会发现它带来的优点远胜过不适。我已经走过这个阶段，Google Play 图书的浏览器阅读体验非常好，包括在 Linux 和 iOS 设备上。作为一次尝试，我决定尝试通过 Google 图书进行出版。

注意：在撰写本文时，通过 Google Play 图书网络浏览器应用阅读图书虽然可以忍受，但体验还是有点差强人意，我强烈推荐读者们使用平板电脑来阅读。

版本说明

本书采用 Pandoc 风格的 Markdown 编写，使用 Pandoc 生成 ePub v3 格式。

正文字体为 Georgia，标题字体为 Verdana。代码字体使用的 Ubuntu Mono，因为它特别紧凑，单行能容纳更多的代码。我选择将代码内联（而不是将列表放入图像，参照其他书籍），因为我觉得这个功能很重要：让代码块能适应字体大小得改变而改变（否则，买电子版，还图什么呢?）。

书中的提取，编译和测试代码示例的构建过程都是自动化的。所有自动化操作都是通过我在 Python 3 中编写的程序来实现的。

封面设计

《On Java 8》的封面是根据 W.P.A.（Works Progress Administration 1935 年至 1943 年美国大萧条期间的一个巨大项目，它使数百万失业人员重新就业）的马赛克创作的。它还让我想起了《绿野仙踪》（*The Wizard of Oz*）系列丛书中的插图。我的好朋友、设计师丹 *Daniel Will-Harris*（www.will-harris.com）和我都喜欢这个形象。

感谢的人

感谢 *Domain-Driven Design*（《领域驱动设计》）的作者 *Eric Evans* 建议书名，以及其他新闻组校对的帮助。

感谢 *James Ward* 为我开始使用 Gradle 工具构建这本书，以及他多年来的帮助和友谊。

感谢 *Ben Muschko* 在整理构建文件方面的工作，还有感谢 *Hans Dockter* 给 *Ben* 提供了时间。

感谢 *Jeremy Cerise* 和 *Bill Frasure* 来到开发商务虚会预订，并随后提供了宝贵的帮助。

感谢所有花时间和精力来科罗拉多州克雷斯特德比特（Crested Butte, Colorado）镇参加我的研讨会，开发商务聚会和其他活动的人！你们的贡献可能不容易看到，但却非常重要！

献礼

> 谨以此书献给我敬爱的父亲 E. Wayne Eckel。1924 年 4 月 1 日至 2016 年 11 月 23 日

目录

1 简介 **1**
1.1 前提条件 2
1.2 JDK 文档 2
1.3 C 编程思想 2
1.4 源码下载 3
1.5 编码样式 4
1.6 BUG 提交 4
1.7 邮箱订阅 4
1.8 Java 图形界面 4

2 对象的概念 **7**
2.1 抽象 7
2.2 接口 8
2.3 服务提供 10
2.4 封装 10
2.5 复用 11
2.6 继承 12
2.6.1 “是一个”与“像是一个”的关系 15
2.7 多态 16
2.8 单继承结构 19
2.9 集合 19
2.10 对象创建与生命周期 21
2.11 异常处理 22
2.12 本章小结 23

3 安装 Java 和本书用例 **25**
3.1 编辑器 25
3.2 Shell 25

3.3 Java 安装 27
3.4 校验安装 28
3.5 安装和运行代码示例 28

4 万物皆对象 31
4.1 对象操纵 31
4.2 对象创建 32
4.2.1 数据存储 32
4.2.2 基本类型的存储 33
4.2.3 高精度数值 34
4.2.4 数组的存储 34
4.3 代码注释 35
4.4 对象清理 35
4.4.1 作用域 35
4.4.2 对象作用域 36
4.5 类的创建 37
4.5.1 类型 37
4.5.2 字段 37
4.5.3 基本类型默认值 38
4.5.4 方法使用 38
4.6 程序编写 40
4.6.1 命名可见性 40
4.6.2 使用其他组件 41
4.6.3 static 关键字 42
4.7 小试牛刀 43
4.7.1 编译和运行 45
4.8 编码风格 46
4.9 本章小结 46

5 运算符 47
5.1 开始使用 47
5.2 优先级 47
5.3 赋值 48
5.4 算术运算符 50
5.4.1 一元加减运算符 52
5.5 递增和递减 52

5.6 关系运算符 53
5.6.1 测试对象等价 54
5.7 逻辑运算符 55
5.7.1 短路 56
5.8 字面值常量 57
5.8.1 下划线 59
5.8.2 指数计数法 60
5.9 位运算符 61
5.10 移位运算符 61
5.11 三元运算符 65
5.12 字符串运算符 66
5.13 常见陷阱 67
5.14 类型转换 67
5.14.1 截断和舍入 68
5.14.2 类型提升 69
5.15 Java 没有 sizeof 69
5.16 运算符总结 70
5.17 本章小结 79

6 控制流 81
6.1 true 和 false 81
6.2 if-else 82
6.3 迭代语句 83
6.3.1 while 83
6.3.2 do-while 84
6.3.3 for 84
6.4 for-in 语法 86
6.5 return 88
6.6 break 和 continue 89
6.7 臭名昭著的 goto 90
6.8 switch 94
6.9 switch 字符串 96
6.10 本章小结 98

7 初始化和清理 99
7.1 利用构造器保证初始化 99

7.2 方法重载 101
7.2.1 区分重载方法 103
7.2.2 重载与基本类型 103
7.2.3 返回值的重载 107
7.3 无参构造器 107
7.4 this 关键字 108
7.4.1 在构造器中调用构造器 110
7.4.2 static 的含义 111
7.5 垃圾回收器 112
7.5.1 `finalize()` 的用途 112
7.5.2 你必须实施清理 113
7.5.3 终结条件 113
7.5.4 垃圾回收器如何工作 115
7.6 成员初始化 117
7.6.1 指定初始化 119
7.7 构造器初始化 120
7.7.1 初始化的顺序 121
7.7.2 静态数据的初始化 122
7.7.3 显式的静态初始化 124
7.7.4 非静态实例初始化 126
7.8 数组初始化 127
7.8.1 动态数组创建 128
7.8.2 可变参数列表 131
7.9 枚举类型 136
7.10 本章小结 139

8 封装 141

8.1 包的概念 142
8.1.1 代码组织 143
8.1.2 创建独一无二的包名 144
8.1.3 冲突 146
8.1.4 定制工具库 146
8.1.5 使用 import 改变行为 147
8.1.6 使用包的忠告 148

8.2 访问权限修饰符 148
8.2.1 包访问权限 148
8.2.2 public: 接口访问权限 149
8.2.3 默认包 150
8.2.4 private: 你无法访问 150
8.2.5 protected: 继承访问权限 151
8.2.6 包访问权限 Vs Public 构造器 153
8.3 接口和实现 154
8.4 类访问权限 155
8.5 本章小结 157

9 复用 159
9.1 组合语法 159
9.2 继承语法 162
9.2.1 初始化基类 164
9.2.2 带参数的构造函数 165
9.3 委托 166
9.4 结合组合与继承 168
9.4.1 保证适当的清理 170
9.4.2 名称隐藏 173
9.5 组合与继承的选择 174
9.6 protected 176
9.7 向上转型 177
9.7.1 再论组合和继承 178
9.8 final 关键字 179
9.8.1 final 数据 179
9.8.2 空白 final 181
9.8.3 final 参数 182
9.8.4 final 方法 183
9.8.5 final 和 private 183
9.8.6 final 类 185
9.8.7 final 忠告 185
9.9 类初始化和加载 186
9.9.1 继承和初始化 186
9.10 本章小结 188

10 多态 **189**
10.1 向上转型回顾 189
10.1.1 忘掉对象类型 191
10.2 转机 192
10.2.1 方法调用绑定 192
10.2.2 产生正确的行为 193
10.2.3 可扩展性 196
10.2.4 陷阱:“重写”私有方法 200
10.2.5 陷阱:属性与静态方法 201
10.3 构造器和多态 203
10.3.1 构造器调用顺序 203
10.3.2 继承和清理 205
10.3.3 构造器内部多态方法的行为 210
10.4 协变返回类型 212
10.5 使用继承设计 213
10.5.1 替代 vs 扩展 215
10.5.2 向下转型与运行时类型信息 216
10.6 本章小结 218

11 接口 **219**
11.1 抽象类和方法 219
11.2 接口创建 225
11.2.1 默认方法 226
11.2.2 多继承 228
11.2.3 接口中的静态方法 231
11.2.4 Instrument 作为接口 232
11.3 抽象类和接口 235
11.4 完全解耦 236
11.5 多接口结合 242
11.6 使用继承扩展接口 244
11.6.1 结合接口时的命名冲突 245
11.7 接口适配 247
11.8 接口字段 250
11.8.1 初始化接口中的字段 250
11.9 接口嵌套 251

11.10 接口和工厂方法模式 254
11.11 本章小结 257

12 内部类 259
12.1 创建内部类 259
12.2 链接外部类 261
12.3 使用.this 和.new 263
12.4 内部类与向上转型 264
12.5 内部类方法和作用域 266
12.6 匿名内部类 268
12.7 嵌套类 271
12.7.1 接口内部的类 273
12.7.2 从多层嵌套类中访问外部类的成员 274
12.8 为什么需要内部类 274
12.8.1 闭包与回调 276
12.8.2 内部类与控制框架 279
12.9 继承内部类 285
12.10 内部类可以被覆盖么? 286
12.11 局部内部类 288
12.12 内部类标识符 289
12.13 本章小结 290

13 集合 291
13.1 泛型和类型安全的集合 292
13.2 基本概念 295
13.3 添加元素组 296
13.4 集合的打印 298
13.5 列表 List 299
13.6 迭代器 Iterators 303
13.6.1 ListIterator 306
13.7 链表 LinkedList 307
13.8 堆栈 Stack 309
13.9 集合 Set 311
13.10 映射 Map 315
13.11 队列 Queue 318
13.11.1 优先级队列 PriorityQueue 319

13.12集合与迭代器 321
13.13for-in 和迭代器 324
13.13.1 适配器方法惯用法 326
13.14本章小结 330
13.14.1 简单集合分类 331

14 函数式编程 335
14.1 新旧对比 336
14.2 Lambda 表达式 338
14.2.1 递归 340
14.3 方法引用 341
14.3.1 Runnable 接口 343
14.3.2 未绑定的方法引用 344
14.3.3 构造函数引用 345
14.4 函数式接口 346
14.4.1 多参数函数式接口 354
14.4.2 缺少基本类型的函数 354
14.5 高阶函数 356
14.6 闭包 358
14.6.1 作为闭包的内部类 362
14.7 函数组合 363
14.8 柯里化和部分求值 365
14.9 纯函数式编程 367
14.10本章小结 367

15 流式编程 369
15.1 流支持 371
15.2 流创建 371
15.2.1 随机数流 373
15.2.2 int 类型的范围 376
15.2.3 generate() 377
15.2.4 iterate() 379
15.2.5 流的建造者模式 380
15.2.6 Arrays 381
15.2.7 正则表达式 383

15.3 中间操作 384
15.3.1 跟踪和调试 384
15.3.2 流元素排序 384
15.3.3 移除元素 385
15.3.4 应用函数到元素 386
15.3.5 在 map() 中组合流 388
15.4 Optional 类 391
15.4.1 便利函数 393
15.4.2 创建 Optional 395
15.4.3 Optional 对象操作 396
15.4.4 Optional 流 401
15.5 终端操作 402
15.5.1 数组 402
15.5.2 循环 403
15.5.3 集合 404
15.5.4 组合 407
15.5.5 匹配 408
15.5.6 查找 409
15.5.7 信息 410
15.5.8 数字流信息 411
15.6 本章小结 412

16 异常 413
16.1 异常概念 413
16.2 基本异常 414
16.2.1 异常参数 415
16.3 异常捕获 416
16.3.1 try 语句块 416
16.3.2 异常处理程序 416
16.3.3 终止与恢复 417
16.4 自定义异常 417
16.4.1 异常与记录日志 419
16.5 异常声明 423
16.6 捕获所有异常 423
16.6.1 多重捕获 425

16.6.2 栈轨迹 426
16.6.3 重新抛出异常 427
16.6.4 精准的重新抛出异常 430
16.6.5 异常链 430
16.7 Java 标准异常 434
16.7.1 特例：RuntimeException 434
16.8 使用 finally 进行清理 436
16.8.1 finally 用来做什么？ 437
16.8.2 在 return 中使用 finally 439
16.8.3 缺憾：异常丢失 440
16.9 异常限制 442
16.10构造器 444
16.11Try-With-Resources 用法 449
16.11.1 揭示细节 451
16.12异常匹配 455
16.13其他可选方式 456
16.13.1 历史 457
16.13.2 观点 458
16.13.3 把异常传递给控制台 459
16.13.4 把“被检查的异常”转换为“不检查的异常” 460
16.14异常指南 462
16.15本章小结 462
16.16后记：Exception Bizarro World 463

17 代码校验 465

17.0.1 你永远不能保证你的代码是正确的，你只能证明它是错的。 465
17.1 测试 465
17.1.1 如果没有测试过，它就是不能工作的。 465
17.2 前置条件 470
17.3 测试驱动开发 485
17.3.1 测试驱动 vs. 测试优先 491
17.4 日志 492
17.4.1 日志会给出正在运行的程序的各种信息。 492
17.4.2 日志等级 493

17.5 调试 494
17.5.1 使用 JDB 调试 494
17.5.2 图形化调试器 497
17.6 基准测试 497
17.6.1 微基准测试 498
17.6.2 JMH 的引入 500
17.7 剖析和优化 506
17.7.1 优化准则 507
17.8 风格检测 507
17.9 静态错误分析 508
17.10 代码重审 508
17.11 结对编程 508
17.12 重构 509
17.12.1 重构基石 509
17.13 持续集成 510
17.14 本章小结 511

18 文件 513

18.1 文件和目录路径 513
18.1.1 选取路径部分片段 516
18.1.2 路径分析 517
18.1.3 **Paths** 的增减修改 518
18.2 目录 520
18.3 文件系统 523
18.4 路径监听 524
18.5 文件查找 527
18.6 文件读写 529
18.7 本章小结 532

19 字符串 533

19.1 字符串的不可变 533
19.2 + 的重载与 `StringBuilder` 534
19.3 意外递归 538
19.4 字符串操作 540
19.5 格式化输出 543
19.5.1 `Formatter` 类 544

19.6 正则表达式 551
19.7 扫描输入 568
19.8 StringTokenizer 类 571
19.9 本章小结 572

20 类型信息 573

20.1 为什么需要 RTTI 573
20.2 `Class` 对象 575
20.2.1 类字面常量 580
20.2.2 泛化的 `Class` 引用 582
20.2.3 `cast()` 方法 585
20.3 类型转换检测 586
20.3.1 使用类字面量 592
20.3.2 一个动态 `instanceof` 函数 594
20.3.3 递归计数 596
20.4 注册工厂 597
20.5 类的等价比较 600
20.6 反射：运行时类信息 602
20.6.1 类方法提取器 603
20.7 动态代理 605
20.8 Optional 类 610
20.8.1 标记接口 615
20.8.2 Mock 对象和桩 619
20.9 接口和类型 619
20.10 本章小结 626

21 泛型 629

21.0.1 与 C++ 的比较 629
21.1 简单泛型 630
21.1.1 一个元组类库 632
21.1.2 一个堆栈类 635
21.1.3 RandomList 636
21.2 泛型接口 637
21.3 泛型方法 641
21.3.1 变长参数和泛型方法 642
21.3.2 一个泛型的 Supplier 643

21.3.3 简化元组的使用 645
21.3.4 一个 Set 工具 646
21.4 构建复杂模型 652
21.5 泛型擦除 654
21.5.1 C++ 的方式 656
21.5.2 迁移兼容性 659
21.5.3 擦除的问题 660
21.5.4 边界处的动作 661
21.6 补偿擦除 665
21.6.1 创建类型的实例 667
21.6.2 泛型数组 671
21.7 边界 676
21.8 通配符 683
21.8.1 编译器有多聪明 685
21.8.2 逆变 687
21.8.3 无界通配符 689
21.8.4 捕获转换 698
21.9 问题 700
21.9.1 任何基本类型都不能作为类型参数 700
21.9.2 实现参数化接口 702
21.9.3 转型和警告 702
21.9.4 重载 705
21.9.5 基类劫持接口 705
21.10 自限定的类型 706
21.10.1 古怪的循环泛型 707
21.10.2 自限定 708
21.10.3 参数协变 710
21.11 动态类型安全 714
21.12 泛型异常 715
21.13 混型 717
21.13.1 C++ 中的混型 717
21.13.2 与接口混合 718
21.13.3 使用装饰器模式 720
21.13.4 与动态代理混合 721

21.14潜在类型机制 723
21.14.1 pyhton 中的潜在类型 724
21.14.2 C++ 中的潜在类型 725
21.14.3 Go 中的潜在类型 726
21.14.4 java 中的直接潜在类型 727
21.15对缺乏潜在类型机制的补偿 728
21.15.1 反射 729
21.15.2 将一个方法应用于序列 730
21.16Java8 中的辅助潜在类型 734
21.16.1 使用 **Suppliers** 类的通用方法 736
21.17总结：类型转换真的如此之糟吗？ 739
21.18进阶阅读 740

22 数组 743

22.1 数组特性 743
22.1.1 用于显示数组的实用程序 745
22.2 一等对象 747
22.3 返回数组 749
22.4 多维数组 751
22.5 泛型数组 755
22.6 Arrays 的 fill 方法 757
22.7 Arrays 的 setAll 方法 758
22.8 增量生成 760
22.9 随机生成 768
22.10泛型和基本数组 777
22.11数组元素修改 781
22.12数组并行 782
22.13Arrays 工具类 784
22.14数组拷贝 784
22.15数组比较 786
22.16流和数组 788
22.17数组排序 789
22.18Arrays.sort() 的使用 792
22.19并行排序 794
22.20binarySearch 二分查找 794

22.21 parallelPrefix 并行前缀 796
22.22 本章小结 798

23 枚举 801

23.1 基本 enum 特性 801
23.1.1 将静态类型导入用于 enum 802
23.2 方法添加 803
23.2.1 覆盖 enum 的方法 804
23.3 switch 语句中的 enum 805
23.4 values 方法的神秘之处 806
23.5 实现而非继承 809
23.6 随机选择 810
23.7 使用接口组织枚举 811
23.8 使用 EnumSet 替代 Flags 816
23.9 使用 EnumMap 818
23.10 常量特定方法 819
23.10.1 使用 enum 的职责链 824
23.10.2 使用 enum 的状态机 828
23.11 多路分发 834
23.11.1 使用 enum 分发 836
23.11.2 使用常量相关的方法 839
23.11.3 使用 EnumMap 进行分发 841
23.11.4 使用二维数组 843
23.12 本章小结 844

24 注解 847

24.1 基本语法 848
24.1.1 定义注解 848
24.1.2 元注解 850
24.2 编写注解处理器 850
24.2.1 注解元素 851
24.2.2 默认值限制 852
24.2.3 生成外部文件 852
24.2.4 替代方案 855
24.2.5 注解不支持继承 855
24.2.6 实现处理器 856

24.3 使用 javac 处理注解 858
24.3.1 最简单的处理器 859
24.3.2 更复杂的处理器 862
24.4 基于注解的单元测试 865
24.4.1 在 @Unit 中使用泛型 874
24.4.2 实现 @Unit 876
24.5 本章小结 886

25 并发编程 887
25.1 术语问题 888
25.1.1 并发的新定义 889
25.2 并发为速度而生 891
25.3 四句格言 893
25.3.1 1. 不要这样做 893
25.3.2 2. 没有什么是真的，一切可能都有问题 893
25.3.3 3. 它起作用, 并不意味着它没有问题 893
25.3.4 4. 你必须仍然理解 894
25.4 残酷的真相 895
25.5 本章其余部分 896
25.6 创建和运行任务 907
25.7 终止耗时任务 917
25.8 CompletableFuture 类 919
25.8.1 基本用法 920
25.8.2 结合 CompletableFutures 926
25.8.3 模拟 929
25.8.4 例外情况 931
25.8.5 流异常 935
25.8.6 检查异常 936
25.9 构造函数非线程安全 941
25.10 本章小结 953
25.10.1 缺点 953
25.10.2 共享内存陷阱 954
25.10.3 This Albatross is Big 954
25.10.4 其他类库 955
25.10.5 考虑为并发设计的语言 955

25.10.6 拓展阅读 956

26 设计模式 957

26.1 概念 957
26.1.1 单例模式 958
26.1.2 模式分类 959
26.2 构建应用程序框架 959
26.3 面向实现 961
26.4 工厂模式 967
26.4.1 动态工厂 969
26.4.2 多态工厂 971
26.4.3 抽象工厂 972
26.5 函数对象 975
26.6 接口改变 975
26.7 解释器 975
26.8 回调 975
26.9 多次调度 975
26.10 模式重构 975
26.11 抽象用法 975
26.12 多次派遣 975
26.13 访问者模式 975
26.14 RTTI 的优劣 975
26.15 本章小结 975

1 补充 977

1.1 可下载的补充 977
1.2 通过 Thinking-in-C 来巩固 Java 基础 977
1.3 Hand-On Java 电子演示文稿 977

2 编程指南 979

2.1 设计 979
2.2 实现 982

3 文档注释 987

3.1 句法规则 987
3.2 内嵌 HTML 988

3.3 示例标签 989
3.3.1 @see 989
3.3.2 {@link package.class#member label} 989
3.3.3 {@docRoot} 989
3.3.4 {@inheritDoc} 989
3.3.5 @version 989
3.3.6 @author 990
3.3.7 @since 990
3.3.8 @param 990
3.3.9 @return 990
3.3.10 @throws 990
3.3.11 @deprecated 991
3.4 文档示例 991

4 对象传递和返回 993
4.1 传递引用 993
4.2 本地拷贝 994
4.3 控制克隆 994
4.4 不可变类 994
4.5 本章小结 994

5 流式 IO 995
5.1 输入流类型 996
5.2 输出流类型 997
5.3 添加属性和有用的接口 998
5.3.1 通过 `FilterInputStream` 从 `InputStream` 读取 999
5.3.2 通过 `FilterOutputStream` 向 `OutputStream` 写入 1000
5.4 Reader 和 Writer 1001
5.4.1 数据的来源和去处 1002
5.4.2 更改流的行为 1002
5.4.3 未发生改变的类 1003
5.5 RandomAccessFile 类 1004
5.6 IO 流典型用途 1005
5.6.1 缓冲输入文件 1005
5.6.2 从内存输入 1005
5.6.3 格式化内存输入 1006

5.6.4 基本文件的输出 1007
5.6.5 文本文件输出快捷方式 1008
5.6.6 存储和恢复数据 1009
5.6.7 读写随机访问文件 1010
5.7 本章小结 1012

6 标准 IO 1015
6.1 从标准输入中读取 1015
6.2 将 `System.out` 转换成 `PrintWriter` 1016
6.3 重定向标准 I/O 1017
6.4 执行控制 1018

7 新 IO 1021
7.1 ByteBuffer 1021
7.2 数据转换 1025
7.3 基本类型获取 1028
7.4 视图缓冲区 1030
7.4.1 字节存储次序 1033
7.5 缓冲区数据操作 1034
7.5.1 缓冲区细节 1036
7.6 内存映射文件 1040
7.6.1 性能 1041
7.7 文件锁定 1044
7.7.1 映射文件的部分锁定 1045

8 理解 equals 和 hashCode 方法 1049
8.1 equals 规范 1049
8.1.1 不同子类的相等性 1054
8.2 哈希和哈希码 1057
8.2.1 理解 hashCode 1061
8.2.2 为了速度而散列 1063
8.2.3 重写 hashCode() 1066
8.3 调优 HashMap 1071

9 集合主题 1073
9.1 示例数据 1073

9.2 List 行为 1080
9.3 Set 行为 1084
9.4 在 Map 中使用函数式操作 1086
9.5 选择 Map 片段 1087
9.6 填充集合 1090
9.6.1 使用 Suppliers 填充集合 1091
9.6.2 Map Suppliers 1092
9.7 使用享元（Flyweight）自定义 Collection 和 Map 1095
9.8 集合功能 1106
9.9 可选操作 1110
9.9.1 不支持的操作 1111
9.10 Set 和存储顺序 1113
9.10.1 SortedSet 1116
9.11 队列 1117
9.11.1 优先级队列 1119
9.11.2 双端队列 1120
9.12 理解 Map 1121
9.12.1 性能 1123
9.12.2 SortedMap 1126
9.12.3 LinkedHashMap 1127
9.13 集合工具类 1128
9.13.1 排序和搜索列表 1132
9.13.2 创建不可修改的 Collection 或 Map 1134
9.13.3 同步 Collection 或 Map 1135
9.14 持有引用 1137
9.14.1 WeakHashMap 1139
9.15 Java 1.0 / 1.1 的集合类 1141
9.15.1 Vector 和 Enumeration 1141
9.15.2 Hashtable 1142
9.15.3 Stack 1142
9.15.4 BitSet 1144
9.16 本章小结 1146

10 并发底层原理 1147
10.1 什么是线程? 1147
10.1.1 最佳线程数 1148
10.1.2 我可以创建多少个线程? 1149
10.1.3 The WorkStealingPool (工作窃取线程池) 1150
10.2 异常捕获 1151
10.3 资源共享 1155
10.3.1 资源竞争 1155
10.3.2 解决资源竞争 1159
10.3.3 同步控制 EventProducer 1161
10.4 volatile 关键字 1162
10.4.1 字分裂 1162
10.4.2 可见性 1162
10.4.3 重排与 *Happen-Before* 原则 1163
10.4.4 什么时候使用 volatile 1164
10.5 原子性 1164
10.5.1 Josh 的序列号 1168
10.5.2 原子类 1171
10.6 临界区 1172
10.6.1 在其他对象上同步 1175
10.6.2 使用显式锁对象 1177
10.7 库组件 1179
10.7.1 DelayQueue 1179
10.7.2 PriorityBlockingQueue 1182
10.7.3 无锁集合 1185
10.8 本章小结 1186

11 数据压缩 1189
11.1 使用 Gzip 简单压缩 1190
11.2 使用 zip 多文件存储 1191
11.3 Java 的 jar 1193

12 对象序列化 1197
12.1 查找类 1200
12.2 控制序列化 1202
12.2.1 transient 关键字 1205

12.2.2 Externalizable 的替代方法 1207
12.2.3 版本控制 1210
12.3 使用持久化 1210
12.4 XML 1216

13 静态语言类型检查 1221
13.1 前言 1221
13.2 静态类型检查和测试 1221
13.3 如何提升打字 1221
13.4 生产力的成本 1221
13.5 静态和动态 1221

14 C++ 和 Java 的优良传统 1223

15 成为一名程序员 1225
15.1 如何开始 1225
15.2 码农生涯 1226
15.2.1 延伸阅读 1229
15.3 百分之五的神话 1229
15.4 重在动手 1229
15.5 像打字般编程 1229
15.6 做你喜欢的事 1229
15.7 词汇表 1229

1 简介

> “我的语言极限，即是我的世界的极限。” — 路德维希·维特根斯坦（*Wittgenstein*）

这句话无论对于自然语言还是编程语言来说都是一样的。你所使用的编程语言会将你的思维模式固化并逐渐远离其他语言，而且往往发生在潜移默化中。Java 作为一门傲娇的语言尤其如此。

Java 是一门派生语言，早期语言设计者为了不想在项目中使用 C++ 而创造了这种看起来很像 C++，却比 C++ 有了改进的新语言（原始的项目并未成功）。Java 最核心的变化就是加入了“虚拟机”和“垃圾回收机制”，这两个概念在之后的章节会有详细描述。此外，Java 还在其他方面推动了行业发展。例如，现在绝大多数编程语言都支持文档注释语法和 HTML 文档生成工具。

Java 最主要的概念之一“对象”来自 SmallTalk 语言。SmallTalk 语言恪守“对象”（在下一章中描述）是编程的最基本单元。于是，万物皆对象。历经时间的检验，人们发现这种信念太过狂热。有些人甚至认为“对象”的概念是完全错误的，应该舍弃。就我个人而言，把一切事物都抽象成对象不仅是一项不必要的负担，同时还会招致许多设计朝着不好的方向发展。尽管如此，“对象”的概念依然有其闪光点。固执地要求所有东西都是一个对象（特别是一直到最底层级别）是一种设计错误；相反，完全逃避“对象”的概念似乎同样太过苛刻。

Java 语言曾规划设计的许多功能并未按照承诺兑现。本书中，我将尝试解释这些原因，力争让读者知晓这些功能，并明白为什么这些功能最终并不适用。这无关 Java 是一种好语言或者坏语言，一旦你了解了该语言的缺陷和局限性，你就能够：

1. 明白有些功能特性为什么会被“废弃”。
2. 熟悉语言边界，更好地设计和编码。

编程的过程就是复杂性管理的过程：业务问题的复杂性，以及依赖的计算机的复杂性。由于这种复杂性，我们的大多数软件项目都失败了。

许多语言设计决策时都考虑到了复杂性，并试图降低语言的复杂性，但在设计过程中遇到了一些更棘手的问题，最终导致语言设计不可避免地“碰壁”，复杂性增加。例如，C++ 必须向后兼容 C（允许 C 程序员轻松迁移），并且效率很高。这些目标非常实用，并且也是 C++ 在编程界取得了成功的原因之一，但同时也引入了额外的复杂性，导致某些用 C++ 编写的项目开发失败。当然，你可以责怪程序员和管理人员手艺不精，但如果有一种编程语言可以帮助你在开发过程中发现错误，那岂不是更好？

虽然 VB（Visual BASIC）绑定在 BASIC 上，但 BASIC 实际上并不是一种可扩展的语言。大量扩展的堆积造成 VB 的语法难以维护。Perl 向后兼容 awk、sed、grep 以及其它要替换的 Unix 工具。因此它常常被诟病产生了一堆“只写代码”（*write-only code*，写代码的人自己都看不懂的代码）。

挣钱容易攒钱难。

另一方面，C ++，VB，Perl 和其他语言（如 SmallTalk）在设计时重点放在了对某些复杂问题的处理上，因而在解决这些特定类型的问题方面非常成功。

通信革命使我们相互沟通更加便利。无论是一对一沟通，还是团队里的互相沟通，甚至是地球上不同地区的沟通。据说下一次革命需要的是一种全球性的思维，这种思维源于足量的人以及足量相互连接。我不知道 Java 是否能成为这场革命的工具之一，但至少这种可能性让我觉得：我现在正在做的传道授业的事情是有意义的！

1.1 前提条件

阅读本书需要读者对编程有基本的了解:

- 程序是一系列“陈述（语句、代码）”构成
- 子程序、方法、宏的概念
- 控制语句（例如 **if**），循环结构（例如 **while**）

可能你已在学校、书籍或网络上了学过这些。只要你觉得对上述的编程基本概念熟悉，你就可以完成本书的学习。

你可以通过在 On Java 8 的网站上免费下载《Think in C》来补充学习 Java 所需要的前置知识。本书介绍了 Java 语言的基本控制机制以及面对对象编程（OOP）的概念。在本书中我引述了一些 C/C++ 语言中的一些特性来帮助读者更好的理解 Java。毕竟 Java 是在它们的基础之上发明的，理解他们之间的区别，有助于读者更好地学习 Java。我会试图简化这些引述，尽量让没有 C/C++ 基础的读者也能很好地理解。

1.2 JDK 文档

甲骨文公司已经提供了免费的标准 JDK 文档。除非有必要，否则本书中将不再赘述 API 相关的使用细节。使用浏览器来即时搜索最新最全的 JDK 文档好过翻阅本书来查找。只有在需要补充特定的示例时，我才会提供有关的额外描述。

1.3 C 编程思想

Thinking in C 已经可以在 www.OnJava8.com 免费下载。Java 的基础语法是基于 C 语言的。*Thinking in C* 中有更适合初学者的编程基础介绍。我已经委托 Chuck Allison 将这本 C 基础的书籍作为独立产品附赠于本书的 CD 中。希望大家在阅读本书时，都已具备了学习 Java 的良好基础。

1.4 源码下载

本书中所有源代码的示例都在版权保护的前提下通过 GitHub 免费提供。你可以将这些代码用于教育。任何人不得在未经正确引用代码来源的情况下随意重新发布此代码示例。在每个代码文件中，你都可以找到以下版权声明文件作为参考：

Copyright.txt

©2017 MindView LLC。版权所有。如果上述版权声明，本段和以下内容，特此授予免费使用，复制，修改和分发此计算机源代码（源代码）及其文档的许可，且无需出于下述目的的书面协议所有副本中都有五个编号的段落。

1. 允许编译源代码并将编译代码仅以可执行格式包含在个人和商业软件程序中。
2. 允许在课堂情况下使用源代码而不修改源代码，包括在演示材料中，前提是 “On Java 8” 一书被引用为原点。
3. 可以通过以下方式获得将源代码合并到印刷媒体中的许可：MindView LLC，PO Box 969，Crested Butte，CO 81224 MindViewInc@gmail.com
4. 源代码和文档的版权归 MindView LLC 所有。提供的源代码没有任何明示或暗示的担保，包括任何适销性，适用于特定用途或不侵权的默示担保。MindView LLC 不保证任何包含源代码的程序的运行不会中断或没有错误。MindView LLC 不对任何目的的源代码或包含源代码的任何软件的适用性做出任何陈述。包含源代码的任何程序的质量和性能的全部风险来自源代码的用户。用户理解源代码是为研究和教学目的而开发的，建议不要仅仅因任何原因依赖源代码或任何包含源代码的程序。如果源代码或任何产生的软件证明有缺陷，则用户承担所有必要的维修，修理或更正的费用。
5. 在任何情况下，MINDVIEW LLC 或其出版商均不对任何一方根据任何法律理论对直接，间接，特殊，偶发或后果性损害承担任何责任，包括利润损失，业务中断，商业信息丢失或任何其他保险公司。由于 MINDVIEW LLC 或其出版商已被告知此类损害的可能性，因此使用本源代码及其文档或因无法使用任何结果程序而导致的个人受伤或者个人受伤。MINDVIEW LLC 特别声明不提供任何担保，包括但不限于对适销性和特定用途适用性的暗示担保。此处提供的源代码和文档基于“原样”基础，没有 MINDVIEW LLC 的任何随附服务，MINDVIEW LLC 没有义务提供维护，支持，更新，增强或修改。

请注意，MindView LLC 仅提供以下唯一网址发布更新书中的代码示例，https://github.com/BruceEckel/OnJava8-examples 。你可在上述条款范围内将示例免费使用于项目和课堂中。

如果你在源代码中发现错误，请在下面的网址提交更正：https://github.com/BruceEckel/OnJava8-examples/issues

1.5 编码样式

本书中代码标识符（关键字，方法，变量和类名）以粗体，固定宽度代码字体显示。像 “*class” 这种在代码中高频率出现的关键字可能让你觉得粗体有点乏味。（译者注：由于中英排版差异，中文翻译过程并未完全参照原作者的说明。具体排版格式请参考此处）其他显示为正常字体。本书文本格式尽可能遵循 Oracle 常见样式，并保证在大多数 Java 开发环境中被支持。书中我使用了自己喜欢的字体风格。Java 是一种自由的编程语言，你也可以使用 IDE（集成开发环境）工具（如 IntelliJ IDEA，Eclipse 或 NetBeans）将格式更改为适合你的格式。

本书代码文件使用自动化工具进行测试，并在最新版本的 Java 编译通过（除了那些特别标记的错误之外）。本书重点介绍并使用 Java 8 进行测试。如果你必须了解更早的语言版本，可以在 www.OnJava8.com 免费下载《Thinking in Java》。

1.6 BUG 提交

本书经过多重校订，但还是难免有所遗漏被新读者发现。如果你在正文或示例中发现任何错误的内容，请在此处提交错误以及建议更正，作者感激不尽。

1.7 邮箱订阅

你可以在 www.OnJava8.com 上 订阅邮件。邮件不含广告并尽量提供干货。

1.8 Java 图形界面

Java 在图形用户界面和桌面程序方面的发展可以说是一段悲伤的历史。Java 1.0 中图形用户界面（GUI）库的原始设计目标是让用户能在所有平台提供一个漂亮的界面。但遗憾的是，这个理想没有实现。相反，Java 1.0 AWT（抽象窗口工具包）在所有平台都表现平平，并且有诸多限制。你只能使用四种字体。另外，Java 1.0 AWT 编程模型也很笨拙且非面向对象。我的一个曾在 Java 设计期间工作过的学生道出了缘由：早期的 AWT 设计是在仅仅在一个月内构思、设计和实施的。不得不说这是一个“奇迹”，但同时更是“设计失败”的绝佳教材。

在 Java 1.1 版本的 AWT 中情况有所改善，事件模型带来更加清晰的面向对象方法，并添加了 JavaBeans，致力于面向易于创建可视化编程环境的组件编程模型（已废弃）。

Java 2（Java 1.2）通过使用 Java 基类（JFC）内容替换来完成从旧版 Java 1.0 AWT 的转换。其中 GUI 部分称为 Swing。这是一组丰富的 JavaBeans，它们创建了一个合理的 GUI。修订版 3（3 之前都不好）比以往更适用于开发图形界面程序。

Sun 在图形界面的最后一次尝试，称为 JavaFX。当 Oracle 收购 Sun 时，他们将原来雄心勃勃的项目（包括脚本语言）改为库，现在它似乎是 Java 官方唯一还在开发中的 UI 工具包（参见维基百科关于 JavaFX 的文章）- 但即使如此，JavaFX 最终似乎也失败了。

现今 Swing 依然是 Java 发行版的一部分（只接受维护，不再有新功能开发）。而 Java 现在是一个开源项目，它应该始终可用。此外，Swing 和 JavaFX 有一些有限的交互性。这些可能是为了帮助开发者过渡到 JavaFX。

桌面程序领域似乎从未尝勾起 Java 设计师的野心。Java 没有在图形界面取得该有的一席之地。另外，曾被大肆吹嘘的 JavaBeans 也没有获得任何影响力。（许多不幸的作者花了很多精力在 Swing 上编写书籍，甚至只用 JavaBeans 编写书籍）。Java 图形界面程序大多数情况下仅用于 IDE（集成开发环境）和一些企业内部应用程序。你可以采用 Java 开发图形界面，但这并非 Java 最擅长的领域。如果你必须学习 Swing，可以参考 *Thinking in Java* 第 4 版（可从 www.OnJava8.com 获得）或者通过其他专门的书籍中学习。。

Choose an author as you choose a friend.

2 对象的概念

> “我们没有意识到惯用语言的结构有多大的力量。可以毫不夸张地说，它通过语义反应机制奴役我们。语言表现出来并在无意识中给我们留下深刻印象的结构会自动投射到我们周围的世界。” – Alfred Korzybski (1930)

计算机革命的起源来自机器。编程语言就像是那台机器。它不仅是我们思维放大的工具与另一种表达媒介，更像是我们思想的一部分。语言的灵感来自其他形式的表达，如写作，绘画，雕塑，动画和电影制作。编程语言就是创建应用程序的思想结构。

面向对象编程（Object-Oriented Programming OOP）是一种编程思维方式和编码架构。本章讲述 OOP 的基本概述。如果读者对此不太理解，可先行跳过本章。等你具备一定编程基础后，请务必再回头看。只有这样你才能深刻理解面向对象编程的重要性及设计方式。

2.1 抽象

所有编程语言都提供抽象机制。从某种程度上来说，问题的复杂度直接取决于抽象的类型和质量。这里的“类型”意思是：抽象的内容是什么？汇编语言是对底层机器的轻微抽象。接着出现的“命令式”语言（如 FORTRAN，BASIC 和 C）是对汇编语言的抽象。与汇编相比，这类语言已有了长足的改进，但它们的抽象原理依然要求我们着重考虑计算机的结构，而非问题本身的结构。

程序员必须要在机器模型（“解决方案空间”）和实际解决的问题模型（“问题空间”）之间建立起一种关联。这个过程既费精力，又脱离编程语言本身的范畴。这使得程序代码很难编写，维护代价高昂。同时还造就了一门副产业的“编程方法”学科。

为机器建模的另一个方法是为要解决的问题制作模型。对一些早期语言来说，如 LISP 和 APL，它们的做法是“从不同的角度观察世界”——“所有问题都归纳为列表”或“所有问题都归纳为算法”。PROLOG 则将所有问题都归纳为决策链。对于这些语言，我们认为它们一部分是“基于约束”的编程，另一部分则是专为处理图形符号设计的（后者被证明限制性太强）。每种方法都有自己特殊的用途，适合解决某一类的问题。只要超出了它们力所能及的范围，就会显得非常笨拙。

面向对象的程序设计在此基础上跨出了一大步，程序员可利用一些工具表达“问题空间”内的元素。由于这种表达非常具有普遍性，所以不必受限于特定类型的问题。我们将问题空间中的元素以及它们在解决方案空间的表示称作“对象”（**Object**）。当然，还有一些在问题空间没有对应的对象体。通过添加

新的对象类型，程序可进行灵活的调整，以便与特定的问题配合。所以当你在阅读描述解决方案的代码时，也是在阅读问题的表述。与我们以前见过的相比，这无疑是一种更加灵活、更加强大的语言抽象方法。总之，OOP 允许我们根据问题来描述问题，而不是根据运行解决方案的计算机。然而，它仍然与计算机有联系，每个对象都类似一台小计算机：它们有自己的状态并且可以进行特定的操作。这与现实世界的“对象”或者“物体”相似：它们都有自己的特征和行为。

Smalltalk 作为第一个成功的面向对象并影响了 Java 的程序设计语言，*Alan Kay* 总结了其五大基本特征。通过这些特征，我们可理解“纯粹”的面向对象程序设计方法是什么样的：

> 1. **万物皆对象**。你可以将对象想象成一种特殊的变量。它存储数据，但可以在你对其“发出请求”时执行本身的操作。理论上讲，你总是可以从要解决的问题身上抽象出概念性的组件，然后在程序中将其表示为一个对象。
> 2. **程序是一组对象，通过消息传递来告知彼此该做什么**。要请求调用一个对象的方法，你需要向该对象发送消息。
> 3. **每个对象都有自己的存储空间，可容纳其他对象**。或者说，通过封装现有对象，可制作出新型对象。所以，尽管对象的概念非常简单，但在程序中却可达到任意高的复杂程度。
> 4. **每个对象都有一种类型**。根据语法，每个对象都是某个“类”的一个“实例”。其中，“类”（Class）是“类型”（Type）的同义词。一个类最重要的特征就是“能将什么消息发给它？”。
> 5. **同一类所有对象都能接收相同的消息**。这实际是别有含义的一种说法，大家不久便能理解。由于类型为“圆”（Circle）的一个对象也属于类型为“形状”（Shape）的一个对象，所以一个圆完全能接收发送给“形状”的消息。这意味着可让程序代码统一指挥“形状”，令其自动控制所有符合“形状”描述的对象，其中自然包括“圆”。这一特性称为对象的“可替换性”，是 OOP 最重要的概念之一。

Grady Booch 提供了对对象更简洁的描述：一个对象具有自己的状态，行为和标识。这意味着对象有自己的内部数据 (提供状态)、方法 (产生行为)，并彼此区分（每个对象在内存中都有唯一的地址）。

2.2 接口

亚里士多德（*Aristotle*）大概是第一个认真研究“类型”的哲学家，他曾提出过“鱼类和鸟类”这样的概念。所有对象都是唯一的，但同时也是具有相同的特性和行为的对象所归属的类的一部分。这种思想被首次应用于第一个面向对象编程语言 Simula-67，它在程序中使用基本关键字 **class** 来引入新的类型（class 和 type 通常可互换使用，有些人对它们进行了进一步区分，他们强调 type 决定了接口，而 class 是那个接口的一种特殊实现方式）。

Simula 是一个很好的例子。正如这个名字所暗示的，它的作用是“模拟”（Simulate）类似“银行出纳员”这样的经典问题。在这个例子里，我们有一系列出纳员、客户、帐号、交易和货币单位等许多“对象”。每类成员（元素）都具有一些通用的特征：每个帐号都有一定的余额；每名出纳都能接收客户的

存款；等等。与此同时，每个成员都有自己的状态；每个帐号都有不同的余额；每名出纳都有一个名字。所以在计算机程序中，能用独一无二的实体分别表示出纳员、客户、帐号以及交易。这个实体便是“对象”，而且每个对象都隶属一个特定的“类”，那个类具有自己的通用特征与行为。

因此，在面向对象的程序设计中，尽管我们真正要做的是新建各种各样的数据“类型”（Type），但几乎所有面向对象的程序设计语言都采用了 `class` 关键字。当你看到 “type” 这个词的时候，请同时想到 `class`；反之亦然。

创建好一个类后，可根据情况生成许多对象。随后，可将那些对象作为要解决问题中存在的元素进行处理。事实上，当我们进行面向对象的程序设计时，面临的最大一项挑战性就是：如何在“问题空间”（问题实际存在的地方）的元素与“方案空间”（对实际问题进行建模的地方，如计算机）的元素之间建立理想的“一对一”的映射关系。

那么如何利用对象完成真正有用的工作呢？必须有一种办法能向对象发出请求，令其解决一些实际的问题，比如完成一次交易、在屏幕上画一些东西或者打开一个开关等等。每个对象仅能接受特定的请求。我们向对象发出的请求是通过它的“接口”（Interface）定义的，对象的“类型”或“类”则规定了它的接口形式。“类型”与“接口”的对应关系是面向对象程序设计的基础。

下面让我们以电灯泡为例：

Figure 2.1: reader

```
Light lt = new Light();
lt.on();
```

在这个例子中，类型／类的名称是 **Light**，可向 **Light** 对象发出的请求包括包括打开 `on`、关闭 `off`、变得更明亮 `brighten` 或者变得更暗淡 `dim`。通过声明一个引用，如 `lt` 和 `new` 关键字，我们创建了一个 **Light** 类型的对象，再用等号将其赋给引用。

为了向对象发送消息，我们使用句点符号 `.` 将 `lt` 和消息名称 `on` 连接起来。可以看出，使用一些预先定义好的类时，我们在程序里采用的代码是非常简单直观的。

上图遵循 **UML**（Unified Modeling Language，统一建模语言）的格式。每个类由一个框表示，框的顶部有类型名称，框中间部分是要描述的任何数据成员，方法（属于此对象的方法，它们接收任何

发送到该对象的消息）在框的底部。通常，只有类的名称和公共方法在 **UML** 设计图中显示，因此中间部分未显示，如本例所示。如果你只对类名感兴趣，则也不需要显示方法信息。

2.3 服务提供

在开发或理解程序设计时，我们可以将对象看成是“服务提供者”。你的程序本身将为用户提供服务，并且它能通过调用其他对象提供的服务来实现这一点。我们的最终目标是开发或调用工具库中已有的一些对象，提供理想的服务来解决问题。

那么问题来了：我们该选择哪个对象来解决问题呢？例如，你正在开发一个记事本程序。*你可能会想到在屏幕输入默认的记事本对象*，一个用于检测不同类型打印机并执行打印的对象。这些对象中的某些已经有了。那对于还没有的对象，我们该设计成啥样呢？这些对象需要提供哪些服务，以及还需要调用其他哪些对象？

我们可以将这些问题一一分解，抽象成一组服务。软件设计的基本原则是高内聚：每个组件的内部作用明确，功能紧密相关。然而经常有人将太多功能塞进一个对象中。例如：在支票打印模块中，你需要设计一个可以同时读取文本格式又能正确识别不同打印机型号的对象。正确的做法是提供三个或更多对象：一个对象检查所有排版布局的目录；一个或一组可以识别不同打印机型号的对象展示通用的打印界面；第三个对象组合上述两个服务来完成任务。这样，每个对象都提供了一组紧密的服务。在良好的面向对象设计中，每个对象功能单一且高效。这样的程序设计可以提高我们代码的复用性，同时也方便别人阅读和理解我们的代码。只有让人知道你提供什么服务，别人才能更好地将其应用到其他模块或程序中。

2.4 封装

我们可以把编程的侧重领域划分为研发和应用。应用程序员调用研发程序员构建的基础工具类来做快速开发。研发程序员开发一个工具类，该工具类仅向应用程序员公开必要的内容，并隐藏内部实现的细节。这样可以有效地避免该工具类被错误的使用和更改，从而减少程序出错的可能。彼此职责划分清晰，相互协作。当应用程序员调用研发程序员开发的工具类时，双方建立了关系。应用程序员通过使用现成的工具类组装应用程序或者构建更大的工具库。如果工具类的创建者将类的内部所有信息都公开给调用者，那么有些使用规则就不容易被遵守。因为前者无法保证后者是否会按照正确的规则来使用，甚至是改变该工具类。只有设定访问控制，才能从根本上阻止这种情况的发生。

因此，使用访问控制的原因有以下两点：

1. 让应用程序员不要触摸他们不应该触摸的部分。（请注意，这也是一个哲学决策。部分编程语言认为如果程序员有需要，则应该让他们访问细节部分。）；
2. 使类库的创建者（研发程序员）在不影响后者使用的情况下完善更新工具库。例如，我们开发了一个功能简单的工具类，后来发现可以通过优化代码来提高执行速度。假如工具类的接口和实现

部分明确分开并受到保护，那我们就可以轻松地完成改造。

Java 有三个显式关键字来设置类中的访问权限：`public`（公开），`private`(私有) 和`protected`（受保护）。这些访问修饰符决定了谁能使用它们修饰的方法、变量或类。

1. `public`（公开）表示任何人都可以访问和使用该元素；
2. `private`（私有）除了类本身和类内部的方法，外界无法直接访问该元素。`private` 是类和调用者之间的屏障。任何试图访问私有成员的行为都会报编译时错误；
3. `protected`（受保护）类似于 `private`，区别是子类（下一节就会引入继承的概念）可以访问 `protected` 的成员，但不能访问 `private` 成员；
4. `default`（默认）如果你不使用前面的三者，默认就是 `default` 访问权限。`default` 被称为包访问，因为该权限下的资源可以被同一包（库组件）中其他类的成员访问。

2.5 复用

一个类经创建和测试后，理应是可复用的。然而很多时候，由于程序员没有足够的编程经验和远见，我们的代码复用性并不强。

代码和设计方案的复用性是面向对象程序设计的优点之一。我们可以通过重复使用某个类的对象来达到这种复用性。同时，我们也可以将一个类的对象作为另一个类的成员变量使用。新的类可以是由任意数量和任意类型的其他对象构成。这里涉及到“组合”和“聚合”的概念：

- **组合**（Composition）经常用来表示“拥有”关系（has-a relationship）。例如，“汽车拥有引擎”。
- **聚合**（Aggregation）动态的**组合**。

Figure 2.2: UML-example

上图中实心三角形指向 “ **Car** ” 表示 **组合**的关系；如果是 **聚合**关系，可以使用空心三角形。

（**译者注**：组合和聚合都属于关联关系的一种，只是额外具有整体-部分的意义。至于是聚合还是组合，需要根据实际的业务需求来判断。可能相同超类和子类，在不同的业务场景，关联关系会发生变化。只看代码是无法区分聚合和组合的，具体是哪一种关系，只能从语义级别来区分。聚合关系中，整件不会拥有部件的生命周期，所以整件删除时，部件不会被删除。再者，多个整件可以共享同一个部件。组合关系中，整件拥有部件的生命周期，所以整件删除时，部件一定会跟着删除。而且，多个整件不可以同时共享同一个部件。这个区别可以用来区分某个关联关系到底是组合还是聚合。两个类生命周期不同步，则是聚合关系，生命周期同步就是组合关系。）

再好的朋友也有分手的时候。

使用“组合”关系给我们的程序带来极大的灵活性。通常新建的类中，成员对象会使用 `private` 访问权限，这样应用程序员则无法对其直接访问。我们就可以在不影响客户代码的前提下，从容地修改那些成员。我们也可以在“运行时”改变成员对象从而动态地改变程序的行为，这进一步增大了灵活性。下面一节要讲到的“继承”并不具备这种灵活性，因为编译器对通过继承创建的类进行了限制。

在面向对象编程中经常重点强调“继承”。在新手程序员的印象里，或许先入为主地认为“继承应当随处可见”。沿着这种思路产生的程序设计通常拙劣又复杂。相反，在创建新类时首先要考虑“组合”，因为它更简单灵活，而且设计更加清晰。等我们有一些编程经验后，一旦需要用到继承，就会明显意识到这一点。

2.6 继承

“继承”给面向对象编程带来极大的便利。它在概念上允许我们将各式各样的数据和功能封装到一起，这样便可恰当表达“问题空间”的概念，而不用受制于必须使用底层机器语言。

通过使用 `class` 关键字，这些概念形成了编程语言中的基本单元。遗憾的是，这么做还是有很多麻烦：在创建了一个类之后，即使另一个新类与其具有相似的功能，你还是得重新创建一个新类。但我们若能利用现成的数据类型，对其进行“克隆”，再根据情况进行添加和修改，情况就显得理想多了。“继承”正是针对这个目标而设计的。但继承并不完全等价于克隆。在继承过程中，若原始类（正式名称叫作基类、超类或父类）发生了变化，修改过的“克隆”类（正式名称叫作继承类或者子类）也会反映出这种变化。

Figure 2.3: Inheritance-example

这个图中的箭头从派生类指向基类。正如你将看到的，通常有多个派生类。类型不仅仅描述一组对象的约束，它还涉及其他类型。两种类型可以具有共同的特征和行为，但是一种类型可能包含比另一种类型更多的特征，并且还可以处理更多的消息（或者以不同的方式处理它们）。继承通过基类和派生类的概念来表达这种相似性。基类包含派生自它的类型之间共享的所有特征和行为。创建基类以表示思想的

核心。从基类中派生出其他类型来表示实现该核心的不同方式。

Figure 2.4: 1545764724202

例如，垃圾回收机对垃圾进行分类。基类是“垃圾”。每块垃圾都有重量、价值等特性，它们可以被切碎、熔化或分解。在此基础上，可以通过添加额外的特性 (瓶子有颜色，钢罐有磁性) 或行为 (铝罐可以被压碎) 派生出更具体的垃圾类型。此外，一些行为可以不同（纸张的价值取决于它的类型和状态）。使用继承，你将构建一个类型层次结构，来表示你试图解决的某种类型的问题。第二个例子是常见的“形状”例子，可能用于计算机辅助设计系统或游戏模拟。基类是“形状”，每个形状都有大小、颜色、位置等等。每个形状可以绘制、擦除、移动、着色等。由此，可以派生出（继承出）具体类型的形状——圆形、正方形、三角形等等——每个形状可以具有附加的特征和行为。

Figure 2.5: 1545764780795

例如，某些形状可以翻转。有些行为可能不同，比如计算形状的面积。类型层次结构体现了形状之间的相似性和差异性。以相同的术语将解决方案转换成问题是有用的，因为你不需要在问题描述和解决方案描述之间建立许多中间模型。通过使用对象，类型层次结构成为了主要模型，因此你可以直接从真实世界中对系统的描述过渡到用代码对系统进行描述。事实上，有时候，那些善于寻找复杂解决方案的人会被面向对象设计的简单性难倒。从现有类型继承创建新类型。这种新类型不仅包含现有类型的所有成员（尽管私有成员被隐藏起来并且不可访问），而且更重要的是它复制了基类的接口。也就是说，基类对象接收的所有消息也能被派生类对象接收。根据类接收的消息，我们知道类的类型，因此派生类与基类是相同的类型。

在前面的例子中，“圆是形状”。这种通过继承的类型等价性是理解面向对象编程含义的基本门槛之一。因为基类和派生类都具有相同的基本接口，所以伴随此接口的必定有某些具体实现。也就是说，当对象接收到特定消息时，必须有可执行代码。如果继承一个类而不做其他任何事，则来自基类接口的方法直接进入派生类。这意味着派生类和基类不仅具有相同的类型，而且具有相同的行为，这么做没什么特别意义。

有两种方法可以区分新的派生类与原始的基类。第一种方法很简单：在派生类中添加新方法。这些新方法不是基类接口的一部分。这意味着基类不能满足你的所有需求，所以你添加了更多的方法。继承的这种简单而原始的用途有时是解决问题的完美解决方案。然而，还是要仔细考虑是否在基类中也要有这些额外的方法。这种设计的发现与迭代过程在面向对象程序设计中会经常发生。

尽管继承有时意味着你要在接口中添加新方法（尤其是在以 **extends** 关键字表示继承的 Java 中），

但并非总需如此。第二种也是更重要地区分派生类和基类的方法是改变现有基类方法的行为，这被称为覆盖 (overriding)。要想覆盖一个方法，只需要在派生类中重新定义这个方法即可。

2.6.1 “是一个”与“像是一个”的关系

对于继承可能会引发争论：继承应该只覆盖基类的方法 (不应该添加基类中没有的方法) 吗？如果这样的话，基类和派生类就是相同的类型了，因为它们具有相同的接口。这会造成，你可以用一个派生类对象完全替代基类对象，这叫作“纯粹替代”，也经常被称作“替代原则”。在某种意义上，这是一种处理继承的理想方式。我们经常把这种基类和派生类的关系称为是一个（is-a）关系，因为可以说“圆是一个形状”。判断是否继承，就看在你的类之间有无这种 is-a 关系。

有时你在派生类添加了新的接口元素，从而扩展接口。虽然新类型仍然可以替代基类，但是这种替代不完美，原因在于基类无法访问新添加的方法。这种关系称为像是一个 (is-like-a) 关系。新类型不但拥有旧类型的接口，而且包含其他方法，所以不能说新旧类型完全相同。

Figure 2.6: 1545764820176

以空调为例，假设房间里已经安装好了制冷设备的控制器，即你有了控制制冷设备的接口。想象一下，现在空调坏了，你重新安装了一个既制冷又制热的热力泵。热力泵就像是一个（is-like-a）空调，但它可以做更多。因为当初房间的控制系统被设计成只能控制制冷设备，所以它只能与新对象 (热力泵) 的制冷部分通信。新对象的接口已经扩展了，现有控制系统却只知道原来的接口，一旦看到这个设计，你就会发现，作为基类的制冷系统不够一般化，应该被重新命名为“温度控制系统”，也应该包含制热功能，这样的话，我们就可以使用替代原则了。上图反映了在现实世界中进行设计时可能会发生的事情。

当你看到替代原则时，很容易会认为纯粹替代是唯一可行的方式，并且使用纯粹替代的设计是很好的。但有些时候，你会发现必须得在派生 (扩展) 类中添加新方法 (提供新的接口)。只要仔细审视，你可以很明显地区分两种设计方式的使用场合。

2.7 多态

我们在处理类的层次结构时，通常把一个对象看成是它所属的基类，而不是把它当成具体类。通过这种方式，我们可以编写出不局限于特定类型的代码。在上个“形状”的例子中，“方法”（method）操纵的是通用“形状”，而不关心它们是“圆”、“正方形”、“三角形”还是某种尚未定义的形状。所有的形状都可以被绘制、擦除和移动，因此“方法”向其中的任何代表“形状”的对象发送消息都不必担心对象如何处理信息。

这样的代码不会受添加的新类型影响，并且添加新类型是扩展面向对象程序以处理新情况的常用方法。例如，你可以通过通用的“形状”基类派生出新的“五角形”形状的子类，而不需要修改通用“形状”基类的方法。通过派生新的子类来扩展设计的这种能力是封装变化的基本方法之一。

这种能力改善了我们的设计，且减少了软件的维护代价。如果我们把派生的对象类型统一看成是它本身的基类（“圆”当作“形状”，“自行车”当作“车”，“鸬鹚”当作“鸟”等等），编译器（compiler）在编译时期就无法准确地知道什么“形状”被擦除，哪一种“车”在行驶，或者是哪种“鸟”在飞行。这就是关键所在：当程序接收这种消息时，程序员并不想知道哪段代码会被执行。“绘图”的方法可以平等地应用到每种可能的“形状”上，形状会依据自身的具体类型执行恰当的代码。

如果不需要知道执行了哪部分代码，那我们就能添加一个新的不同执行方式的子类而不需要更改调用它的方法。那么编译器在不确定该执行哪部分代码时是怎么做的呢？举个例子，下图的 **BirdController** 对象和通用 **Bird** 对象中，**BirdController** 不知道 **Bird** 的确切类型却还能一起工作。从 **BirdController** 的角度来看，这是很方便的，因为它不需要编写特别的代码来确定 **Bird** 对象的确切类型或行为。那么，在调用 **move()** 方法时是如何保证发生正确的行为（鹅走路、飞或游泳、企鹅走路或游泳）的呢？

An ounce of prevention is worth a pound of cure.

Figure 2.7: Bird-example

这个问题的答案，是面向对象程序设计的妙诀：在传统意义上，编译器不能进行函数调用。由非 OOP 编译器产生的函数调用会引起所谓的**早期绑定**，这个术语你可能从未听说过，不会想过其他的函数调用方式。这意味着编译器生成对特定函数名的调用，该调用会被解析为将执行的代码的绝对地址。

通过继承，程序直到运行时才能确定代码的地址，因此发送消息给对象时，还需要其他一些方案。为了解决这个问题，面向对象语言使用**后期绑定**的概念。当向对象发送信息时，被调用的代码直到运行时才确定。编译器确保方法存在，并对参数和返回值执行类型检查，但是它不知道要执行的确切代码。

为了执行后期绑定，Java 使用一个特殊的代码位来代替绝对调用。这段代码使用对象中存储的信息来计算方法主体的地址（此过程在多态性章节中有详细介绍）。因此，每个对象的行为根据特定代码位的内容而不同。当你向对象发送消息时，对象知道该如何处理这条消息。在某些语言中，必须显式地授予方法后期绑定属性的灵活性。例如，C++ 使用 **virtual** 关键字。在这些语言中，默认情况下方法不是动态绑定的。在 Java 中，动态绑定是默认行为，不需要额外的关键字来实现多态性。

为了演示多态性，我们编写了一段代码，它忽略了类型的具体细节，只与基类对话。该代码与具体类型信息分离，因此更易于编写和理解。而且，如果通过继承添加了一个新类型（例如，一个六边形），那么代码对于新类型的 Shape 就像对现有类型一样有效。因此，该程序是可扩展的。

代码示例：

```
void doSomething(Shape shape) {
    shape.erase();
    // ...
    shape.draw();
}
```

此方法与任何 **Shape** 对话，因此它与所绘制和擦除的对象的具体类型无关。如果程序的其他部分使用 `doSomething()` 方法：

```
    Circle circle = new Circle();
    Triangle triangle = new Triangle();
    Line line = new Line();
    doSomething(circle);
    doSomething(triangle);
    doSomething(line);
```

可以看到无论传入的“形状”是什么，程序都正确的执行了。

Figure 2.8: shape-example

这是一个非常令人惊奇的编程技巧。分析下面这行代码：

```
    doSomething(circle);
```

当预期接收 **Shape** 的方法被传入了 **Circle**，会发生什么。由于 **Circle** 也是一种 **Shape**，所以 `doSomething(circle)` 能正确地执行。也就是说，`doSomething()` 能接收任意发送给 **Shape** 的消息。这是完全安全和合乎逻辑的事情。

这种把子类当成其基类来处理的过程叫做“向上转型”（**upcasting**）。在面向对象的编程里，经常利用这种方法来给程序解耦。再看下面的 `doSomething()` 代码示例：

```
    shape.erase();
    // ...
    shape.draw();
```

我们可以看到程序并未这样表达：“如果你是一个 Circle ，就这样做；如果你是一个 Square，就那样做…”。若那样编写代码，就需检查 Shape 所有可能的类型，如圆、矩形等等。这显然是非常麻烦的，而且每次添加了一种新的 Shape 类型后，都要相应地进行修改。在这里，我们只需说：“你是一种几何形状，我知道你能删掉 `erase()` 和绘制 `draw()`，你自己去做吧，注意细节。”

尽管我们没作出任何特殊指示，程序的操作也是完全正确和恰当的。我们知道，为 Circle 调

用`draw()` 时执行的代码与为一个 Square 或 Line 调用 `draw()` 时执行的代码是不同的。但在将 `draw()` 信息发给一个匿名 Shape 时，根据 Shape 句柄当时连接的实际类型，会相应地采取正确的操作。这非常神奇，因为当 Java 编译器为 `doSomething()` 编译代码时，它并不知道自己要操作的准确类型是什么。

尽管我们确实可以保证最终会为 Shape 调用 `erase()` 和 `draw()`，但并不能确定特定的 Circle, Square 或者 Line 调用什么。最后，程序执行的操作却依然是正确的，这是怎么做到的呢?

发送消息给对象时，如果程序不知道接收的具体类型是什么，但最终执行是正确的，这就是对象的“多态性”（Polymorphism）。面向对象的程序设计语言是通过“动态绑定”的方式来实现对象的多态性的。编译器和运行时系统会负责对所有细节的控制；我们只需知道要做什么，以及如何利用多态性来更好地设计程序。

2.8 单继承结构

自从 C++ 引入以来，一个 OOP 问题变得尤为突出：是否所有的类都应该默认从一个基类继承呢? 这个答案在 Java 中是肯定的（实际上，除 C++ 以外的几乎所有 OOP 语言中也是这样）。在 Java 中，这个最终基类的名字就是 `Object`。

Java 的单继承结构有很多好处。由于所有对象都具有一个公共接口，因此它们最终都属于同一个基类。相反的，对于 C++ 所使用的多继承的方案则是不保证所有的对象都属于同一个基类。从向后兼容的角度看，多继承的方案更符合 C 的模型，而且受限较少。

对于完全面向对象编程，我们必须要构建自己的层次结构，以提供与其他 OOP 语言同样的便利。我们经常会使用到新的类库和不兼容的接口。为了整合它们而花费大气力（有可能还要用上多继承）以获得 C++ 样的“灵活性”值得吗? 如果从零开始，Java 这样的替代方案会是更好的选择。

另外，单继承的结构使得垃圾收集器的实现更为容易。这也是 Java 在 C++ 基础上的根本改进之一。

由于运行期的类型信息会存在于所有对象中，所以我们永远不会遇到判断不了对象类型的情况。这对于系统级操作尤其重要，例如异常处理。同时，这也让我们的编程具有更大的灵活性。

2.9 集合

通常，我们并不知道解决某个具体问题需要的对象数量和持续时间，以及对象的存储方式。那么我们如何知悉程序在运行时需要分配的内存空间呢?

在面向对象的设计中，问题的解决方案有些过于轻率：创建一个新类型的对象来引用、容纳其他的对象。当然，我们也可以使用多数编程语言都支持的“数组”（array）。在 Java 中“集合”（Collection）的使用率更高。（也可称之为“容器”，但“集合”这个称呼更通用。）

与其挂在嘴上，不如落实在行动上。

“集合”这种类型的对象可以存储任意类型、数量的其他对象。它能根据需要自动扩容，我们不用关心过程是如何实现的。

还好，一般优秀的 OOP 语言都会将“集合”作为其基础包。在 C++ 中，“集合”是其标准库的一部分，通常被称为 STL（Standard Template Library，标准模板库）。SmallTalk 有一套非常完整的集合库。同样，Java 的标准库中也提供许多现成的集合类。

在一些库中，一两个泛型集合就能满足我们所有的需求了，而在其他一些类库（Java）中，不同类型的集合对应不同的需求：常见的有 List，常用于保存序列；Map，也称为关联数组，常用于将对象与其他对象关联）；Set，只能保存非重复的值；其他还包括如队列（Queue）、树（Tree）、栈（Stack）、堆（Heap）等等。从设计的角度来看，我们真正想要的是一个能够解决某个问题的集合。如果一种集合就满足所有需求，那么我们就不需要剩下的了。之所以选择集合有以下两个原因：

1. 集合可以提供不同类型的接口和外部行为。堆栈、队列的应用场景和集合、列表不同，它们中的一种提供的解决方案可能比其他灵活得多。
2. 不同的集合对某些操作有不同的效率。例如，List 的两种基本类型：ArrayList 和 LinkedList。虽然两者具有相同接口和外部行为，但是在某些操作中它们的效率差别很大。在 ArrayList 中随机查找元素是很高效的，而 LinkedList 随机查找效率低下。反之，在 LinkedList 中插入元素的效率要比在 ArrayList 中高。由于底层数据结构的不同，每种集合类型在执行相同的操作时会表现出效率上的差异。

我们可以一开始使用 LinkedList 构建程序，在优化系统性能时改用 ArrayList。通过对 List 接口的抽象，我们可以很容易地将 LinkedList 改为 ArrayList。

在 Java 5 泛型出来之前，集合中保存的是通用类型 `Object`。Java 单继承的结构意味着所有元素都基于 `Object` 类，所以在集合中可以保存任何类型的数据，易于重用。要使用这样的集合，我们先要往集合添加元素。由于 Java 5 版本前的集合只保存 `Object`，当我们往集合中添加元素时，元素便向上转型成了 `Object`，从而丢失自己原有的类型特性。这时我们再从集合中取出该元素时，元素的类型变成了 `Object`。那么我们该怎么将其转回原先具体的类型呢？这里，我们使用了强制类型转换将其转为更具体的类型，这个过程称为对象的“向下转型”。通过“向上转型”，我们知道“圆形”也是一种“形状”，这个过程是安全的。可是我们不能从“Object”看出其就是“圆形”或“形状”，所以除非我们能确定元素的具体类型信息，否则“向下转型”就是不安全的。也不能说这样的错误就是完全危险的，因为一旦我们转化了错误的类型，程序就会运行出错，抛出“运行时异常”（RuntimeException）。（后面的章节会提到）无论如何，我们要寻找一种在取出集合元素时确定其具体类型的方法。另外，每次取出元素都要做额外的“向下转型”对程序和程序员都是一种开销。以某种方式创建集合，以确认保存元素的具体类型，减少集合元素“向下转型”的开销和可能出现的错误难道不好吗？这种解决方案就是：参数化类型机制（Parameterized Type Mechanism）。

参数化类型机制可以使得编译器能够自动识别某个 **`class`** 的具体类型并正确地执行。举个例子，对集合的参数化类型机制可以让集合仅接受“形状”这种类型的元素，并以“形状”类型取出元素。Java 5 版本支持了参数化类型机制，称之为“泛型”（Generic）。泛型是 Java 5 的主要特性之一。你可以按以

下方式向 ArrayList 中添加 Shape（形状）：

```
ArrayList<Shape> shapes = new ArrayList<>();
```

泛型的应用，让 Java 的许多标准库和组件都发生了改变。在本书的代码示例中，你也会经常看到泛型的身影。

2.10 对象创建与生命周期

我们在使用对象时要注意的一个关键问题就是对象的创建和销毁方式。每个对象的生存都需要资源，尤其是内存。为了资源的重复利用，当对象不再被使用时，我们应该及时释放资源，清理内存。

在简单的编程场景下，对象的清理并不是问题。我们创建对象，按需使用，最后销毁它。然而，情况往往要比这更复杂：

假设，我们正在为机场设计一个空中交通管制的系统（该例也适用于仓库货柜管理、影带出租或者宠物寄养仓库系统）。第一步比较简单：创建一个用来保存飞机的集合，每当有飞机进入交通管制区域时，我们就创建一个“飞机”对象并将其加入到集合中，等到飞机离开时将其从这个集合中清除。与此同时，我们还需要一个记录飞机信息的系统，也许这些数据不像主要控制功能那样引人注意。比如，我们要记录所有飞机中的小型飞机的的信息（比如飞行计划)。此时，我们又创建了第二个集合来记录所有小型飞机。每当创建一个“飞机”对象的时候，将其放入第一个集合；若它属于小型飞机，也必须同时将其放入第二个集合里。

现在问题开始棘手了：我们怎么知道何时该清理这些对象呢？当某一个系统处理完成，而其他系统可能还没有处理完成。这样的问题在其他的场景下也可能发生。在 C++ 程序设计中，当使用完一个对象后，必须明确将其删除，这就让问题变复杂了。

对象的数据在哪？它的生命周期是怎么被控制的？在 C++ 设计中采用的观点是效率第一，因此它将选择权交给了程序员。为了获得最大的运行时速度，程序员可以在编写程序时，通过将对象放在栈（Stack，有时称为自动变量或作用域变量）或静态存储区域（static storage area）中来确定内存占用和生存时间。这些区域的对象会被优先分配内存和释放。这种控制在某些情况下非常有用。

然而相对的，我们也牺牲了程序的灵活性。因为在编写代码时，我们必须要弄清楚对象的数量、生存时间还有类型。如果我们要用它来解决一个相当普遍的问题时（如计算机辅助设计、仓库管理或空中交通管制等），限制就太大了。

第二种方法是在堆内存（Heap）中动态地创建对象。在这种方式下，直到程序运行我们才能确定需要创建的对象数量、生存时间和类型。什么时候需要，什么时候在堆内存中创建。因为内存的占用是动态管理的，所以在运行时，在堆内存上开辟空间所需的时间可能比在栈内存上要长（但也不一定)。在栈内存开辟和释放空间通常是一条将栈指针向下移动和一条将栈指针向上移动的汇编指令。开辟堆内存空间的时间取决于内存机制的设计。

动态方法有这样一个一般性的逻辑假设：对象趋向于变得复杂，因此额外的内存查找和释放

对对象的创建影响不大。（原文：*The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object.*）此外，更好的灵活性对于问题的解决至关重要。

Java 使用动态内存分配。每次创建对象时，使用 `new` 关键字构建该对象的动态实例。这又带来另一个问题：对象的生命周期。较之堆内存，在栈内存中创建对象，编译器能够确定该对象的生命周期并自动销毁它；然而如果你在堆内存创建对象的话，编译器是不知道它的生命周期的。在 C++ 中你必须以编程方式确定何时销毁对象，否则可能导致内存泄漏。Java 的内存管理是建立在垃圾收集器上的，它能自动发现对象不再被使用并释放内存。垃圾收集器的存在带来了极大的便利，它减少了我们之前必须要跟踪的问题和编写相关代码的数量。因此，垃圾收集器提供了更高级别的保险，以防止潜在的内存泄漏问题，这个问题使得许多 C++ 项目没落。

Java 的垃圾收集器被设计用来解决内存释放的问题（虽然这不包括对象清理的其他方面）。垃圾收集器知道对象什么时候不再被使用并且自动释放内存。结合单继承和仅可在堆中创建对象的机制，Java 的编码过程比用 C++ 要简单得多。我们所要做的决定和要克服的障碍也会少很多！

2.11 异常处理

自编程语言被发明以来，程序的错误处理一直都是个难题。因为很难设计出一个好的错误处理方案，所以许多编程语言都忽略了这个问题，把这个问题丢给了程序类库的设计者。他们提出了在许多情况下都可以工作但很容易被规避的半途而废的措施，通常只需忽略错误。多数错误处理方案的主要问题是：它们依赖程序员之间的约定俗成而不是语言层面的限制。换句话说，如果程序员赶时间或没想起来，这些方案就很容易被忘记。

异常处理机制将程序错误直接交给编程语言甚至是操作系统。“异常”（Exception）是一个从出错点“抛出”（thrown）后能被特定类型的异常处理程序捕获 (catch) 的一个对象。它不会干扰程序的正常运行，仅当程序出错的时候才被执行。这让我们的编码更简单：不用再反复检查错误了。另外，异常不像方法返回的错误值和方法设置用来表示发生错误的标志位那样可以被忽略。异常的发生是不会被忽略的，它终究会在某一时刻被处理。

最后，“异常机制”提供了一种可靠地从错误状况中恢复的方法，使得我们可以编写出更健壮的程序。有时你只要处理好抛出的异常情况并恢复程序的运行即可，无需退出。

Java 的异常处理机制在编程语言中脱颖而出。Java 从一开始就内置了异常处理，因此你不得不使用它。这是 Java 语言唯一接受的错误报告方法。如果没有编写适当的异常处理代码，你将会收到一条编译时错误消息。这种有保障的一致性有时会让程序的错误处理变得更容易。值得注意的是，异常处理并不是面向对象的特性。尽管在面向对象的语言中异常通常由对象表示，但是在面向对象语言之前也存在异常处理。

Fools learn nothing from wise men, but wise men learn much from fools.

2.12 本章小结

面向过程程序包含数据定义和函数调用。要找到程序的意图，你必须要在脑中建立一个模型，弄清函数调用和更底层的概念。这些程序令人困扰，因为它们的表示更多地面向计算机而不是我们要解决的问题，这就是我们在设计程序时需要中间表示的原因。OOP 在面向过程编程的基础上增加了许多新的概念，所以有人会认为使用 Java 来编程会比同等的面向过程编程要更复杂。在这里，我想给大家一个惊喜：通常按照 Java 规范编写的程序会比面向过程程序更容易被理解。

你看到的是对象的概念，这些概念是站在“问题空间”的（而不是站在计算机角度的“解决方案空间”），以及发送消息给对象以指示该空间中的活动。面向对象编程的一个优点是：设计良好的 Java 程序代码更容易被人阅读理解。由于 Java 类库的复用性，通常程序要写的代码也会少得多。

OOP 和 Java 不一定适合每个人。评估自己的需求以及与现有方案作比较是很重要的。请充分考虑后再决定是不是选择 Java。如果在可预见的未来，Java 并不能很好的满足你的特定需求，那么你应该去寻找其他替代方案（特别是，我推荐看 Python）。如果你依然选择 Java 作为你的开发语言，我希望你至少应该清楚你选择的是什么，以及为什么选择这个方向。

Custom makes all things easy.

3 安装 Java 和本书用例

现在，我们来为这次阅读之旅做些准备吧！

在开始学习 Java 之前，你必须要先安装好 Java 和本书的源代码示例。因为考虑到可能有“专门的初学者”从本书开始学习编程，所以我会详细地教你如何使用命令行。如果你已经有此方面的经验了，可以跳过这段安装说明。如果你对此处描述的任何术语或过程仍不清楚，还可以通过 Google 搜索找到答案。具体的问题或困难请试着在 StackOverflow 上提问。或者去 YouTube 看有没有相关的安装说明。

3.1 编辑器

首先你需要安装一个编辑器来创建和修改本书用例里的 Java 代码。有可能你还需要使用编辑器来更改系统配置文件。

相比一些重量级的 IDE（Integrated Development Environments，集成开发环境），如 Eclipse、NetBeans 和 IntelliJ IDEA (译者注：做项目强烈推荐 IDEA)，编辑器是一种更纯粹的文本编辑器。如果你已经有了一个用着顺手的 IDE，那就可以直接用了。为了方便后面的学习和统一下教学环境，我推荐大家使用 Atom 这个编辑器。大家可以在 atom.io 上下载。

Atom 是一个免费开源、易于安装且跨平台（支持 Window、Mac 和 Linux）的文本编辑器。内置支持 Java 文件。相比 IDE 的厚重，它比较轻量级，是学习本书的理想工具。Atom 包含了许多方便的编辑功能，相信你一定会爱上它！更多关于 Atom 使用的细节问题可以到它的网站上寻找。

还有很多其他的编辑器。有一种亚文化的群体，他们热衷于争论哪个更好用！如果你找到一个你更喜欢的编辑器，换一种使用也没什么难度。重要的是，你要找一个用着舒服的。

3.2 Shell

如果你之前没有接触过编程，那么有可能对 Shell（命令行窗口）不太熟悉。shell 的历史可以追溯到早期的计算时代，当时在计算机上的操作是都通过输入命令进行的，计算机通过回显响应。所有的操作都是基于文本的。

尽管和现在的图形用户界面相比，Shell 操作方式很原始。但是同时 shell 也为我们提供了许多有用的功能特性。在学习本书的过程中，我们会经常使用到 Shell，包括现在这部分的安装，还有运行

Java 程序。

Mac：单击聚光灯（屏幕右上角的放大镜图标），然后键入 `terminal`。单击看起来像小电视屏幕的应用程序（你也可以单击“return”）。这就启动了你的用户下的 shell 窗口。

windows：首先，通过目录打开 windows 资源管理器：

- Windows 7: 单击屏幕左下角的“开始”图标，输入“explorer”后按回车键。
- Windows 8: 按 Windows+Q，输入 “explorer” 后按回车键。
- Windows 10: 按 Windows+E 打开资源管理器，导航到所需目录，单击窗口左上角的“文件 “选项卡，选择 “打开 Window PowerShell”启动 Shell。

Linux: 在 home 目录打开 Shell。

- Debian: 按 Alt+F2，在弹出的对话框中输入“gnome-terminal”
- Ubuntu: 在屏幕中鼠标右击，选择 “打开终端”，或者按住 Ctrl+Alt+T
- Redhat: 在屏幕中鼠标右击，选择 “打开终端”
- Fedora: 按 Alt+F2，在弹出的对话框中输入“gnome-terminal”

目录

目录是 Shell 的基础元素之一。目录用来保存文件和其他目录。目录就好比树的分支。如果书籍是你系统上的一个目录，并且它有两个其他目录作为分支，例如数学和艺术，那么我们就可以说你有一个书籍目录，它包含数学和艺术两个子目录。注意：Windows 使用 \ 而不是 / 来分隔路径。

Shell 基本操作

我在这展示的 Shell 操作和系统中大体相同。出于本书的原因，下面列举一些在 Shell 中的基本操作：

```
更改目录：  cd <路径>
          cd .. 移动到上级目录
          pushd <路径> 记住来源的同时移动到其他目录，popd 返回来源

目录列举：  ls 列举出当前目录下所有的文件和子目录名（不包含隐藏文件），
              可以选择使用通配符 * 来缩小搜索范围。
              示例(1)：  列举所有以“.java”结尾的文件，输入 ls *.java (Windows: dir *.java)
              示例(2)：  列举所有以“F”开头，“.java”结尾的文件，输入ls F*.java (Windows: dir F*.java)

创建目录：
    Mac/Linux 系统：mkdir
              示例：mkdir books
    Windows   系统：md
              示例：md books

移除文件：
    Mac/Linux 系统：rm
              示例：rm somefile.java
    Windows   系统：del
```

```
                示例: del somefile.java

移除目录:
    Mac/Linux 系统: rm -r
                示例: rm -r books
    Windows   系统: deltree
                示例: deltree books

重复命令:  !!  重复上条命令
                示例: !n 重复倒数第n条命令

命令历史:
    Mac/Linux 系统: history
    Windows   系统: 按 F7 键

文件解压:
    Linux/Mac 都有命令行解压程序 unzip, 你可以通过互联网为 Windows 安装命令行解压程序 unzip。
    图形界面下 (Windows 资源管理器, Mac Finder, Linux Nautilus 或其他等效软件) 右键单击该文件,
    在 Mac 上选择 "open", 在 Linux 上选择 "extract here", 或在 Windows 上选择 "extract all…"。
    要了解关于 shell 的更多信息, 请在维基百科中搜索 Windows shell, Mac/Linux用户可搜索 bash shell。
```

3.3 Java 安装

为了编译和运行代码示例，首先你必须安装 JDK（Java Development Kit，JAVA 软件开发工具包）。本书中采用的是 JDK 8。

Windows

1. 以下为 Chocolatey 的安装说明。
2. 在命令行提示符下输入下面的命令，等待片刻，结束后 Java 安装完成并自动完成环境变量设置。

```
choco install jdk8
```

Macintosh

Mac 系统自带的 Java 版本太老，为了确保本书的代码示例能被正确执行，你必须将它先更新到 Java 8。我们需要管理员权限来运行下面的步骤：

1. 以下为 HomeBrew 的安装说明。安装完成后执行命令 `brew update` 更新到最新版本
2. 在命令行下执行下面的命令来安装 Java。

```
brew cask install java
```

当以上安装都完成后，如果你有需要，可以使用游客账户来运行本书中的代码示例。

英雄所见略同。

Linux

- **Ubuntu/Debian**:

```
sudo apt-get update
sudo apt-get install default-jdk
```

- **Fedora/Redhat**:

```
su-c "yum install java-1.8.0-openjdk"(注: 执行引号内的内容就可以安装)
```

3.4 校验安装

打开新的命令行输入:

```
java -version
```

正常情况下你应该看到以下类似信息 (版本号信息可能不一样):

```
java version "1.8.0_112"
Java(TM) SE Runtime Environment (build 1.8.0_112-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.112-b15, mixed mode)
```

如果提示命令找不到或者无法被识别，请根据安装说明重试；如果还不行，尝试到 StackOverflow 寻找答案。

3.5 安装和运行代码示例

当 Java 安装完毕，下一步就是安装本书的代码示例了。安装步骤所有平台一致:

1. 从 GitHub 仓库中下载本书代码示例
2. 解压到你所选目录里。
3. 使用 Windows 资源管理器，Mac Finder，Linux 的 Nautilus 或其他等效工具浏览，在该目录下打开 Shell。
4. 如果你在正确的目录中，你应该看到该目录中名为 gradlew 和 gradlew.bat 的文件，以及许多其他文件和目录。目录与书中的章节相对应。
5. 在 shell 中输入下面的命令运行:

```
Windows 系统:
     gradlew run

Mac/Linux 系统:
   ./gradlew run
```

第一次安装时 Gradle 需要安装自身和其他的相关的包，请稍等片刻。安装完成后，后续的安装将会快很多。

注意：第一次运行 gradlew 命令时必须连接互联网。

Gradle 基础任务

本书构建的大量 Gradle 任务都可以自动运行。Gradle 使用约定大于配置的方式，简单设置即可具备高可用性。本书中“一起去骑行”的某些任务不适用于此或无法执行成功。以下是你通常会使用上的 Gradle 任务列表：

```
编译本书中的所有 java 文件，除了部分错误示范的
gradlew compileJava

编译并执行 java 文件（某些文件是库组件）
gradlew run

执行所有的单元测试（在本书第16章会有详细介绍）
gradlew test

编译并运行一个具体的示例程序
gradlew <本书章节>:<示例名称>
示例：gradlew objects:HelloDate
```

He sets the fox to keep the geese.

4 万物皆对象

> 如果我们说另外一种不同的语言，我们会发觉一个不同的世界！— Ludwig Wittgenstein (1889-1951)

相比 C++，Java 是一种更纯粹的面向对象编程语言。虽然它们都是混合语言，但在 Java 中，设计者们认为混合的作用并非像在 C++ 中那般重要。混合语言允许多种编程风格，这也是 C++ 支持向后兼容 C 的原因。正因为 C++ 是 C 语言的超集，所以它也同时包含了许多 C 语言不具备的特性，这使得 C++ 在某些方面过于复杂。

Java 语言假设你只进行面向对象编程。开始学习之前，我们需要将思维置于面向对象的世界。本章你将了解到 Java 程序的基本组成，学习在 Java 中万物（几乎）皆对象的思想。

4.1 对象操纵

“名字代表什么？玫瑰即使不叫玫瑰，也依旧芬芳”。（引用自莎士比亚，《罗密欧与朱丽叶》）。

所有的编程语言都会操纵内存中的元素。有时程序员必须要有意识地直接或间接地操纵它们。在 C/C++ 中，对象的操纵是通过指针来完成的。

Java 利用万物皆对象的思想和单一一致的语法方式来简化问题。虽万物皆可为对象，但我们所操纵的标识符实际上只是对对象的“引用” [1]。举例：我们可以用遥控器（引用）去操纵电视（对象）。只要拥有对象的“引用”，就可以操纵该“对象”。换句话说，我们无需直接接触电视，就可通过遥控器（引用）自由地控制电视（对象）的频道和音量。此外，没有电视，遥控器也可以单独存在。就是说，你仅仅有一个“引用”并不意味着你必然有一个与之关联的“对象”。

下面来创建一个 **String** 引用，用于保存单词或语句。代码示例：

```
String s;
```

这里我们只是创建了一个 **String** 对象的引用，而非对象。直接拿来使用会出现错误：因为此时你并没有给变量 `s` 赋值–指向任何对象。通常更安全的做法是：创建一个引用的同时进行初始化。代码示例：

```
String s = "asdf";
```

[1] 功能粘贴在一起的方法的确有点与众不同，但它仍不失为一个库。

Java 语法允许我们使用带双引号的文本内容来初始化字符串。同样，其他类型的对象也有相应的初始化方式。

4.2 对象创建

“引用”用来关联“对象”。在 Java 中，通常我们使用**new**操作符来创建一个新对象。**new** 关键字代表：创建一个新的对象实例。所以，我们也可以这样来表示前面的代码示例：

```
1     String s = new String("asdf");
```

以上展示了字符串对象的创建过程，以及如何初始化生成字符串。除了 **String** 类型以外，Java 本身自带了许多现成的数据类型。除此之外，我们还可以创建自己的数据类型。事实上，这是 Java 程序设计中的一项基本行为。在本书后面的学习中将会接触到。

4.2.1 数据存储

那么，程序在运行时是如何存储的呢？尤其是内存是怎么分配的。有 5 个不同的地方可以存储数据：

1. **寄存器**（Registers）最快的存储区域，位于 CPU 内部 [2]。然而，寄存器的数量十分有限，所以寄存器根据需求进行分配。我们对其没有直接的控制权，也无法在自己的程序里找到寄存器存在的踪迹（另一方面，C/C++ 允许开发者向编译器建议寄存器的分配）。
2. **栈内存**（Stack）存在于常规内存 RAM（随机访问存储器，Random Access Memory）区域中，可通过栈指针获得处理器的直接支持。栈指针下移分配内存，上移释放内存，这是一种快速有效的内存分配方法，速度仅次于寄存器。创建程序时，Java 系统必须准确地知道栈内保存的所有项的生命周期。这种约束限制了程序的灵活性。因此，虽然在栈内存上存在一些 Java 数据，特别是对象引用，但 Java 对象却是保存在堆内存的。
3. **堆内存**（Heap）这是一种通用的内存池（也在 RAM 区域），所有 Java 对象都存在于其中。与栈内存不同，编译器不需要知道对象必须在堆内存上停留多长时间。因此，用堆内存保存数据更具灵活性。创建一个对象时，只需用 **new** 命令实例化对象即可，当执行代码时，会自动在堆中进行内存分配。这种灵活性是有代价的：分配和清理堆内存要比栈内存需要更多的时间（如果可以用 Java 在栈内存上创建对象，就像在 C++ 中那样的话）。随着时间的推移，Java 的堆内存分配机制现在已经非常快，因此这不是一个值得关心的问题了。
4. **常量存储**（Constant storage）常量值通常直接放在程序代码中，因为它们永远不会改变。如需严格保护，可考虑将它们置于只读存储器 ROM （只读存储器，Read Only Memory）中 [3]。

[2] 例如，这个电子书是利用 Pandoc 制作出来的，它是用纯函数式语言 Haskell 编写的一个程序。
[3] 有时函数式语言将其描述为“代码即数据”。

5. **非 RAM 存储**（Non-RAM storage）数据完全存在于程序之外，在程序未运行以及脱离程序控制后依然存在。两个主要的例子：（1）序列化对象：对象被转换为字节流，通常被发送到另一台机器；（2）持久化对象：对象被放置在磁盘上，即使程序终止，数据依然存在。这些存储的方式都是将对象转存于另一个介质中，并在需要时恢复成常规的、基于 RAM 的对象。Java 为轻量级持久化提供了支持。而诸如 JDBC 和 Hibernate 这些类库为使用数据库存储和检索对象信息提供了更复杂的支持。

4.2.2 基本类型的存储

有一组类型在 Java 中使用频率很高，它们需要特殊对待，这就是 Java 的基本类型。之所以这么说，是因为它们的创建并不是通过 **new** 关键字来产生。通常 **new** 出来的对象都是保存在堆内存中的，以此方式创建小而简单的变量往往是不划算的。所以对于这些基本类型的创建方法，Java 使用了和 C/C++ 一样的策略。也就是说，不是使用 **new** 创建变量，而是使用一个“自动”变量。这个变量直接存储“值”，并置于栈内存中，因此更加高效。

Java 确定了每种基本类型的内存占用大小。这些大小不会像其他一些语言那样随着机器环境的变化而变化。这种不变性也是 Java 更具可移植性的一个原因。

基本类型	大小	最小值	最大值	包装类型
boolean	—	—	—	Boolean
char	16 bits	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bits	-128	$+127$	Byte
short	16 bits	-2^{15}	$+2^{15}-1$	Short
int	32 bits	-2^{31}	$+2^{31}-1$	Integer
long	64 bits	-2^{63}	$+2^{63}-1$	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void	—	—	—	Void

所有的数值类型都是有正/负符号的。布尔（boolean）类型的大小没有明确的规定，通常定义为取字面值 “true” 或 “false” 。基本类型有自己对应的包装类型，如果你希望在堆内存里表示基本类型的数据，就需要用到它们的包装类。代码示例：

```
char c = 'x';
```

```
Character ch = new Character(c);
```

或者你也可以使用下面的形式:

```
Character ch = new Character('x');
```

基本类型自动转换成包装类型（自动装箱）

```
Character ch = 'x';
```

相对的，包装类型转化为基本类型（自动拆箱）:

```
char c = ch;
```

个中原因将在以后的章节里解释。

4.2.3 高精度数值

在 Java 中有两种类型的数据可用于高精度的计算。它们是 `BigInteger` 和 `BigDecimal`。尽管它们大致可以划归为“包装类型”，但是它们并没有对应的基本类型。

这两个类包含的方法提供的操作，与对基本类型执行的操作相似。也就是说，能对 int 或 float 做的运算，在 BigInteger 和 BigDecimal 这里也同样可以，只不过必须要通过调用它们的方法来实现而非运算符。此外，由于涉及到的计算量更多，所以运算速度会慢一些。诚然，我们牺牲了速度，但换来了精度。

BigInteger 支持任意精度的整数。可用于精确表示任意大小的整数值，同时在运算过程中不会丢失精度。BigDecimal 支持任意精度的定点数字。例如，可用它进行精确的货币计算。

关于这两个类的详细信息，请参考 JDK 官方文档。

4.2.4 数组的存储

许多编程语言都支持数组类型。在 C 和 C++ 中使用数组是危险的，因为那些数组只是内存块。如果程序访问了内存块之外的数组或在初始化之前使用该段内存（常见编程错误），则结果是不可预测的。

Java 的设计主要目标之一是安全性，因此许多困扰 C 和 C++ 程序员的问题不会在 Java 中再现。在 Java 中，数组使用前需要被初始化，并且不能访问数组长度以外的数据。这种范围检查，是以每个数组上少量的内存开销及运行时检查下标的额外时间为代价的，但由此换来的安全性和效率的提高是值得的。（并且 Java 经常可以优化这些操作）。

当我们创建对象数组时，实际上是创建了一个引用数组，并且每个引用的初始值都为 **null** 。在使用该数组之前，我们必须为每个引用指定一个对象。如果我们尝试使用为 **null** 的引用，则会在运行时报错。因此，在 Java 中就防止了数组操作的常规错误。

我们还可创建基本类型的数组。编译器通过将该数组的内存全部置零来保证初始化。本书稍后将详细介绍数组，特别是在数组章节中。

4.3 代码注释

Java 中有两种类型的注释。第一种是传统的 C 风格的注释，以 /* 开头，可以跨越多行，到 */ 结束。注意，许多程序员在多行注释的每一行开头添加 *，所以你经常会看到：

```
/* 这是
* 跨越多行的
* 注释
*/
```

但请记住，/* 和 */ 之间的内容都是被忽略的。所以你将其改为下面这样也是没有区别的。

```
/* 这是跨越多
行的注释 */
```

第二种注释形式来自 C++ 。它是单行注释，以 // 开头并一直持续到行结束。这种注释方便且常用，因为直观简单。所以你经常看到：

```
// 这是单行注释
```

4.4 对象清理

在一些编程语言中，管理变量的生命周期需要大量的工作。一个变量需要存活多久？如果我们想销毁它，应该什么时候去做呢？变量生命周期的混乱会导致许多 bug，本小结向你介绍 Java 是如何通过释放存储来简化这个问题的。

4.4.1 作用域

大多数程序语言都有作用域的概念。作用域决定了在该范围内定义的变量名的可见性和生存周期。在 C、C++ 和 Java 中，作用域是由大括号 {} 的位置决定的。例如：

```
{
    int x = 12;
    // 仅 x 变量可用
    {
        int q = 96;
        // x 和 q 变量皆可用
    }
    // 仅 x 变量可用
```

```
    // 变量 q 不在作用域内
}
```

Java 的变量只有在其作用域内才可用。缩进使得 Java 代码更易于阅读。由于 Java 是一种自由格式的语言，额外的空格、制表符和回车并不会影响程序的执行结果。在 Java 中，你不能执行以下操作，即使这在 C 和 C++ 中是合法的：

```
{
    int x = 12;
    {
        int x = 96; // Illegal
    }
}
```

在上例中，Java 编译器会在提示变量 x 已经被定义过了。因此，在 C/C++ 中将一个较大作用域的变量“隐藏”起来的做法，在 Java 中是不被允许的。因为 Java 的设计者认为这样做会导致程序混乱。

4.4.2 对象作用域

Java 对象与基本类型具有不同的生命周期。当我们使用 **new** 关键字来创建 Java 对象时，它的生命周期将会超出作用域。因此，下面这段代码示例：

```
{
    String s = new String("a string");
}
// 作用域终点
```

上例中，引用 s 在作用域终点就结束了。但是，引用 s 指向的字符串对象依然还在占用内存。在这段代码中，我们无法在这个作用域之后访问这个对象，因为唯一对它的引用 s 已超出了作用域的范围。在后面的章节中，我们还会学习怎么在编程中传递和复制对象的引用。

只要你需要，**new** 出来的对象就会一直存活下去。相比在 C++ 编码中操作内存可能会出现的诸多问题，这些困扰在 Java 中都不复存在了。在 C++ 中你不仅要确保对象的内存在你操作的范围内存在，还必须在使用完它们之后，将其销毁。

那么问题来了：我们在 Java 中并没有主动清理这些对象，那么它是如何避免 C++ 中出现的内存被填满从而阻塞程序的问题呢？答案是：Java 的垃圾收集器会检查所有 **new** 出来的对象并判断哪些不再可达，继而释放那些被占用的内存，供其他新的对象使用。也就是说，我们不必担心内存回收的问题了。你只需简单创建对象即可。当其不再被需要时，能自行被垃圾收集器释放。垃圾回收机制有效防止了因程序员忘记释放内存而造成的“内存泄漏”问题。

A bad thing never dies.

4.5 类的创建

4.5.1 类型

如果一切都是对象，那么是什么决定了某一类对象的外观和行为呢？换句话说，是什么确定了对象的类型？你可能很自然地想到 `type` 关键字。但是，事实上大多数面向对象的语言都使用 **`class`** 关键字类来描述一种新的对象。通常在 **`class`** 关键字的后面的紧跟类的的名称。如下代码示例：

```
class ATypeName {
 // 这里是类的内部
}
```

在上例中，我们引入了一个新的类型，尽管这个类里只有一行注释。但是我们一样可以通过 **`new`** 关键字来创建一个这种类型的对象。如下：

```
ATypeName a = new ATypeName();
```

到现在为止，我们还不能用这个对象来做什么事（即不能向它发送任何有意义的消息），除非我们在这个类里定义一些方法。

4.5.2 字段

当我们创建好一个类之后，我们可以往类里存放两种类型的元素：方法（method）和字段（field）。类的字段可以是基本类型，也可以是引用类型。如果类的字段是对某个对象的引用，那么必须要初始化该引用将其关联到一个实际的对象上（通过之前介绍的创建对象的方法）。每个对象都有用来存储其字段的空间。通常，字段不在对象间共享。下面是一个具有某些字段的类的代码示例：

```
class DataOnly {
    int i;
    double d;
    boolean b;
}
```

这个类除了存储数据之外什么也不能做。但是，我们仍然可以通过下面的代码来创建它的一个对象：

```
    DataOnly data = new DataOnly();
```

我们必须通过这个对象的引用来指定字段值。格式：对象名称. 方法名称或字段名称。代码示例：

```
    data.i = 47;
    data.d = 1.1;
    data.b = false;
```

如果你想修改对象内部包含的另一个对象的数据，可以通过这样的格式修改。代码示例：

```
myPlane.leftTank.capacity = 100;
```

你可以用这种方式嵌套许多对象（尽管这样的设计会带来混乱）。

4.5.3 基本类型默认值

如果类的成员变量（字段）是基本类型，那么在类初始化时，这些类型将会被赋予一个初始值。

基本类型	初始值
boolean	false
char	\u0000 (null)
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0d

这些默认值仅在 Java 初始化类的时候才会被赋予。这种方式确保了基本类型的字段始终能被初始化（在 C++ 中不会），从而减少了 bug 的来源。但是，这些初始值对于程序来说并不一定是合法或者正确的。所以，为了安全，我们最好始终显式地初始化变量。

这种默认值的赋予并不适用于局部变量 —— 那些不属于类的字段的变量。因此，若在方法中定义的基本类型数据，如下：

```
int x;
```

这里的变量 x 不会自动初始化为 0，因而在使用变量 x 之前，程序员有责任主动地为其赋值（和 C、C++ 一致）。如果我们忘记了这一步，Java 将会提示我们“编译时错误，该变量可能尚未被初始化”。这一点做的比 C++ 更好，在后者中，编译器只是提示警告，而在 Java 中则直接报错。

4.5.4 方法使用

在许多语言（如 C 和 C++）中，使用术语 *函数* (function) 用来命名子程序。在 Java 中，我们使用术语 *方法*（method）来表示“做某事的方式”。

A little knowledge is a dangerous thing.

在 Java 中，方法决定对象能接收哪些消息。方法的基本组成部分包括名称、参数、返回类型、方法体。格式如：

```
[返回类型] [方法名](/*参数列表*/){
    // 方法体
}
```

返回类型 方法的返回类型表明了当你调用它时会返回的结果类型。参数列表则显示了可被传递到方法内部的参数类型及名称。方法名和参数列表统称为**方法签名**（signature of the method）。签名作为方法的唯一标识。

Java 中的方法只能作为类的一部分创建。它只能被对象所调用 [4]，并且该对象必须有权限来执行调用。若对象调用错误的方法，则程序将在编译时报错。

我们可以像下面这样调用一个对象的方法：

```
[对象引用].[方法名](参数1, 参数2, 参数3);
```

若方法不带参数，例如一个对象引用 `a` 的方法 `f` 不带参数并返回 **int** 型结果，我们可以如下表示：

```
int x = a.f();
```

上例中方法 `f` 的返回值类型必须和变量 `x` 的类型兼容。调用方法的行为有时被称为向对象发送消息。面向对象编程可以总结为：向对象发送消息。

参数列表 方法参数列表指定了传递给方法的信息。正如你可能猜到的，这些信息就像 Java 中的其他所有信息，以对象的形式传递。参数列表必须指定每个对象的类型和名称。同样，我们并没有直接处理对象，而是在传递对象引用 [5]。但是引用的类型必须是正确的。如果方法需要 String 参数，则必须传入 String，否则编译器将报错。

```
int storage(String s) {
    return s.length() * 2;
}
```

此方法计算并返回某个字符串所占的字节数。参数 `s` 的类型为 **String** 。将 s 传递给 `storage()` 后，我们可以把它看作和任何其他对象一样，可以向它发送消息。在这里，我们调用 `length()` 方法，它是一个 String 方法，返回字符串中的字符数。字符串中每个字符的大小为 16 位或 2 个字节。你还看到了 **return** 关键字，它执行两项操作。首先，它意味着“方法执行结束”。其次，如果方法有返回值，那么该值就紧跟 **return** 语句之后。这里，返回值是通过计算

[4] 这个语法来自 C++。
[5] 我还没有验证过这种说法。

```
s.length() * 2
```

产生的。在方法中，我们可以返回任何类型的数据。如果我们不想方法返回数据，则可以通过给方法标识 **void** 来表明这是一个无需返回值的方法。代码示例：

```
boolean flag() {
    return true;
}

double naturalLogBase() {
    return 2.718;
}

void nothing() {
     return;
}

void nothing2() {

}
```

当返回类型为 **void** 时，**return** 关键字仅用于退出方法，因此在方法结束处的 **return** 可被省略。我们可以随时从方法中返回，但若方法返回类型为非 **void**，则编译器会强制我们返回相应类型的值。

上面的描述可能会让你感觉程序只不过是一堆包含各种方法的对象，在这些方法中，将对象作为参数并发送消息给其他对象。大部分情况下确实如此。但在下一章的运算符中我们将会学习如何在方法中做出决策来完成更底层、详细的工作。对于本章，知道如何发送消息就够了。

4.6 程序编写

在看到第一个 Java 程序之前，我们还必须理解其他几个问题。

4.6.1 命名可见性

命名控制在任何一门编程语言中都是一个问题。如果你在两个模块中使用相同的命名，那么如何区分这两个名称，并防止两个名称发生“冲突”呢？在 C 语言编程中这是很具有挑战性的，因为程序通常是一个无法管理的名称海洋。C++ 将函数嵌套在类中，所以它们不会和嵌套在其他类中的函数名冲突。然而，C++ 还是允许全局数据和全局函数，因此仍有可能发生冲突。为了解决这个问题，C++ 使用附加的关键字引入了*命名空间*。

Java 采取了一种新的方法避免了以上这些问题：为一个类库生成一个明确的名称，Java 创建者希望我们反向使用自己的网络域名，因为域名通常是唯一的。因此我的域名是 MindviewInc.com，所以我将我的 foibles 类库命名为 com.mindviewinc.utility.foibles。反转域名后，**.** 用来代表子目录的划分。

Cannot see the wood for the trees.

在 Java 1.0 和 Java 1.1 中，域扩展名 com、edu、org 和 net 等按惯例大写，因此类库中会出现这样类似的名称：Com.mindviewinc.utility.foibles。然而，在 Java 2 的开发过程中，他们发现这会导致问题，所以现在整个包名都是小写的。此机制意味着所有文件都自动存在于自己的命名空间中，文件中的每个类都具有唯一标识符。这样，Java 语言可以防止名称冲突。

使用反向 URL 是一种新的命名空间方法，在此之前尚未有其他语言这么做过。Java 中有许多这些“创造性”地解决问题的方法。正如你想象，如果我们未经测试就添加一个功能并用于生产，那么在将来发现该功能的问题再想纠正，通常为时已晚（有些错误太严重了就得从语言中删除新功能。）

使用反向 URL 将命名空间与文件路径相关联不会导致 BUG，但它却给源代码管理带来麻烦。例如在 `com.mindviewinc.utility.foibles` 这样的目录结构中，我们创建了 `com` 和 `mindviewinc` 空目录。它们存在的唯一目的就是用来表示这个反向的 URL。

这种方式似乎为我们在编写 Java 程序中的某个问题打开了大门。空目录填充了深层次结构，它们不仅用于表示反向 URL，还用于捕获其他信息。这些长路径基本上用于存储有关目录中的内容的数据。如果你希望以最初设计的方式使用目录，这种方法可以从“令人沮丧”到“令人抓狂”，对于生产级的 Java 代码，你必须使用专门为此设计的 IDE 来管理代码。例如 NetBeans，Eclipse 或 IntelliJ IDEA。实际上，这些 IDE 都为我们管理和创建深层次空目录结构。

对于这本书中的例子，我不想让深层次结构给你的学习带来额外的麻烦，这实际上需要你在开始之前学习熟悉一种重量级的 IDE。所以，我们的每个章节的示例都位于一个浅的子目录中，以章节标题为名。这导致我偶尔会与遵循深层次方法的工具发生冲突。

4.6.2 使用其他组件

无论何时在程序中使用预先定义好的类，编译器都必须找到该类。最简单的情况下，该类存在于被调用的源代码文件中。此时我们使用该类 —— 即使该类在文件的后面才会被定义（Java 消除了所谓的“前向引用”问题）。而如果一个类位于其他文件中，又会怎样呢？你可能认为编译器应该足够智能去找到它，但这样是有问题的。想象一下，假如你要使用某个类，但目录中存在多个同名的类（可能用途不同）。或者更糟糕的是，假设你正在编写程序，在构建过程中，你想将某个新类添加到类库中，但却与已有的类名称冲突。

要解决此问题，你必须通过使用 **import** 关键字来告诉 Java 编译器具体要使用的类。**import** 指示编译器导入一个包，也就是一个类库（在其他语言中，一个库不仅包含类，还可能包括函数和数据，但请记住 Java 中的所有代码都必须写在类里）。大多数时候，我们都在使用 Java 标准库中的组件。有了这些构件，你就不必写一长串的反转域名。例如：

```
import java.util.ArrayList;
```

上例可以告诉编译器使用位于标准库 **util** 下的 ArrayList 类。但是，**util** 中包含许多类，我们可以使用通配符 * 来导入其中部分类，而无需显式得逐一声明这些类。代码示例：

一叶障目，不见泰山。

```
import java.util.*;
```

本书中的示例很小，为简单起见，我们通常会使用 `.*` 形式略过导入。然而，许多教程书籍都会要求程序员逐一导入每个类。

4.6.3 static 关键字

类是对象的外观及行为方式的描述。通常只有在使用 **new** 创建那个类的对象后，数据存储空间才被分配，对象的方法才能供外界调用。这种方式在两种情况下是不足的。

1. 有时你只想为特定字段（注：也称为属性、域）分配一个共享存储空间，而不去考虑究竟要创建多少对象，甚至根本就不创建对象。
2. 创建一个与此类的任何对象无关的方法。也就是说，即使没有创建对象，也能调用该方法。

static 关键字（从 C++ 采用）就符合上述两点要求。当我们说某个事物是静态时，就意味着该字段或方法不依赖于任何特定的对象实例。即使我们从未创建过该类的对象，也可以调用其静态方法或访问其静态字段。相反，对于普通的非静态字段和方法，我们必须要先创建一个对象并使用该对象来访问字段或方法，因为非静态字段和方法必须与特定对象关联 [6] 。

一些面向对象的语言使用类数据（class data）和类方法（class method），表示静态数据和方法只是作为类，而不是类的某个特定对象而存在的。有时 Java 文献也使用这些术语。

我们可以在类的字段或方法前添加 **static** 关键字来表示这是一个静态字段或静态方法。代码示例：

```
class StaticTest {
    static int i = 47;
}
```

现在，即使你创建了两个 `StaticTest` 对象，但是静态变量 `i` 仍只占一份存储空间。两个对象都会共享相同的变量 `i`。代码示例：

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

`st1.i` 和 `st2.i` 指向同一块存储空间，因此它们的值都是 47。引用静态变量有两种方法。在前面的示例中，我们通过一个对象来定位它，例如 `st2.i`。我们也可以通过类名直接引用它，这种方式对于非静态成员则不可行：

```
StaticTest.i++;
```

++ 运算符将会使变量结果 + 1。此时 `st1.i` 和 `st2.i` 的值都变成了 48。

使用类名直接引用静态变量是首选方法，因为它强调了变量的静态属性。类似的逻辑也适用于静态方法。我们可以通过对象引用静态方法，就像使用任何方法一样，也可以通过特殊的语法方式

[6] 当你理解了并发编程章节的内容，你就能明白为什么更改共享变量“不是线程安全的”的了。

`Classname.method()` 来直接调用静态字段或方法 [7]。代码示例:

```
class Incrementable {
    static void increment() {
      StaticTest.i++;
    }
}
```

上例中，`Incrementable` 的 `increment()` 方法通过 ++ 运算符将静态数据 `i` 加 1。我们依然可以先实例化对象再调用该方法。代码示例:

```
Incrementable sf = new Incrementable();
sf.increment();
```

当然了，首选的方法是直接通过类来调用它。代码示例:

```
Incrementable.increment();
```

相比非静态的对象，**static** 属性改变了数据创建的方式。同样，当 **static** 关键字修饰方法时，它允许我们无需创建对象就可以直接通过类的引用来调用该方法。正如我们所知，**static** 关键字的这些特性对于应用程序入口点的 `main()` 方法尤为重要。

4.7 小试牛刀

最后，我们开始编写第一个完整的程序。我们使用 Java 标准库中的 **Date** 类来展示一个字符串和日期。

```

// objects/HelloDate.java
import java.util.*;

public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
```

在本书中，所有代码示例的第一行都是注释行，其中包含文件的路径信息（比如本章的目录名是 **objects**），后跟文件名。我的工具可以根据这些信息自动提取和测试书籍的代码，你也可以通过参考第一行注释轻松地在 Github 库中找到对应的代码示例。

如果你想在代码中使用一些额外的类库，那么就必须在程序文件的开始处使用 **import** 关键字来导入它们。之所以说是额外的，因为有一些类库已经默认自动导入到每个文件里了。例如: `java.lang`

[7] 接口能够支持方法的原因是它们是 Java 8 默认方法，你将在下一章中了解到。

包。

现在打开你的浏览器在 Oracle 上查看文档。如果你还没有从 Oracle 网站上下载 JDK 文档，那现在就去 [8] 。查看包列表，你会看到 Java 附带的所有不同的类库。

选择 `java.lang`，你会看到该库中所有类的列表。由于 `java.lang` 隐式包含在每个 Java 代码文件中，因此这些类是自动可用的。`java.lang` 类库中没有 **Date** 类，所以我们必须导入其他的类库(即 Date 所在的类库)。如果你不清楚某个类所在的类库或者想查看类库中所有的类，那么可以在 Java 文档中选择 “Tree” 查看。

现在，我们可以找到 Java 附带的每个类。使用浏览器的“查找”功能查找 **Date**，搜索结果中将会列出 **java.util.Date**，我们就知道了 **Date** 在 **util** 库中，所以必须导入 **java.util.*** 才能使用 **Date**。

如果你在文档中选择 **java.lang**，然后选择 **System**，你会看到 **System** 类中有几个字段，如果你选择了 **out**，你会发现它是一个静态的 **PrintStream** 对象。所以，即使我们不使用 **new** 创建，**out** 对象就已经存在并可以使用。**out** 对象可以执行的操作取决于它的类型：**PrintStream** ，其在文档中是一个超链接，如果单击该链接，我们将可以看到 **PrintStream** 对应的方法列表（更多详情，将在本书后面介绍）。现在我们重点说的是 **println()** 这个方法。它的作用是 “将信息输出到控制台，并以换行符结束”。既然如此，我们可以这样编码来输出信息到控制台。代码示例：

```
System.out.println("A String of things");
```

每个 java 源文件中允许有多个类。同时，源文件的名称必须要和其中一个类名相同，否则编译器将会报错。每个独立的程序应该包含一个 `main()` 方法作为程序运行的入口。其方法签名和返回类型如下。代码示例：

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

关键字 **public** 表示方法可以被外界访问到。（更多详情将在 **隐藏实现**章节讲到）**main()** 方法的参数是一个字符串（**String**）数组。参数 **args** 并没有在当前的程序中使用到，但是 Java 编译器强制要求必须要有，这是因为它们被用于接收从命令行输入的参数。

下面我们来看一段有趣的代码：

```
System.out.println(new Date());
```

上面的示例中，我们创建了一个日期（**Date**）类型的对象并将其转化为字符串类型，输出到控制台中。一旦这一行语句执行完毕，我们就不再需要该日期对象了。这时，Java 垃圾回收器就可以将其占用的内存回收，我们无需去主动清除它们。

查看 JDK 文档时，我们可以看到在 **System** 类下还有很多其他有用的方法（Java 的牛逼之处还在于，它拥有一个庞大的标准库资源）。代码示例：

```
// objects/ShowProperties.java
```

[8] 一些语言，如 Python，允许像调用其他函数一样调用组合函数。但这是 Java，所以我们做做可为之事。

```
public class ShowProperties {
    public static void main(String[] args) {
        System.getProperties().list(System.out);
        System.out.println(System.getProperty("user.name"));
        System.out.println(System.getProperty("java.library.path"));
    }
}
```

输出结果 (前 20 行):

```
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Program
Files\Java\jdk1.8.0_112\jr...
java.vm.version=25.112-b15
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator=;
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
file.encoding.pkg=sun.io
user.script=
user.country=US
sun.java.launcher=SUN_STANDARD
sun.os.patch.level=
java.vm.specification.name=Java Virtual Machine
Specification
user.dir=C:\Users\Bruce\Documents\GitHub\on-ja...
java.runtime.version=1.8.0_112-b15
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs=C:\Program
Files\Java\jdk1.8.0_112\jr...
os.arch=amd64
java.io.tmpdir=C:\Users\Bruce\AppData\Local\Temp\
```

`main()` 方法中的第一行会输出所有的系统字段，也就是环境信息。**list()** 方法将结果发送给它的参数 **System.out**。在本书的后面，我们还会接触到将结果输出到其他地方，例如文件中。另外，我们还可以请求特定的字段。该例中我们使用到了 **user.name** 和 **java.library.path**。

4.7.1 编译和运行

要编译和运行本书中的代码示例，首先必须具有 Java 编程环境。第二章的示例中描述了安装过程。如果你遵循这些说明，那么你将会在不受 Oracle 的限制的条件下用到 Java 开发工具包（JDK）。如果你使用其他开发系统，请查看该系统的文档以确定如何编译和运行程序。第二章还介绍了如何安装本书的示例。

移动到子目录 **objects** 下并键入:

```
javac HelloDate.java
```

一心不能二用。

此命令不应产生任何响应。如果我们收到任何类型的错误消息，则表示未正确安装 JDK，那就得检查这些问题。

若执行不报错的话，此时可以键入：

```
java HelloDate
```

我们将会得到正确的日期输出。这是我们编译和运行本书中每个程序（包含 `main()` 方法）的过程[9]。此外，本书的源代码在根目录中也有一个名为 **build.gradle** 的文件，其中包含用于自动构建，测试和运行本书文件的 **Gradle** 配置。当你第一次运行 `gradlew` 命令时，**Gradle** 将自动安装（前提是已安装 Java）。

4.8 编码风格

Java 编程语言编码规范（Code Conventions for the Java Programming Language）[10] 要求类名的首字母大写。如果类名是由多个单词构成的，则每个单词的首字母都应大写（不采用下划线来分隔）例如：

```
class AllTheColorsOfTheRainbow {
    // ...
}
```

有时称这种命名风格叫“驼峰命名法”。对于几乎所有其他方法，字段（成员变量）和对象引用名都采用驼峰命名的方式，但是它们的首字母不需要大写。代码示例：

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

在 Oracle 的官方类库中，花括号的位置同样遵循和本书中上述示例相同的规范。

4.9 本章小结

本章向你展示了简单的 Java 程序编写以及该语言相关的基本概念。到目前为止，我们的示例都只是些简单的顺序执行。在接下来的两章里，我们将会接触到 Java 的一些基本操作符，以及如何去控制程序执行的流程。

[9] 例如，Immutables 和 Mutability Detector。
[10] 为了保持本书的代码排版紧凑，我并没完全遵守规范，但我尽量会做到符合 Java 标准。

5 运算符

> 运算符操纵数据。

Java 是从 C++ 的基础上做了一些改进和简化发展而成的。对于 C/C++ 程序员来说，Java 的运算符并不陌生。如果你已了解 C 或 C++，大可以跳过本章和下一章，直接阅读 Java 与 C/C++ 不同的地方。

如果理解这两章的内容对你来说还有点困难，那么我推荐你先了解下《Thinking in C》再继续后面的学习。这本书现在可以在 www.OnJava8.com 上免费下载。它的内容包含音频讲座、幻灯片、练习和解答，专门用于帮助你快速掌握学习 Java 所需的基础知识。

5.1 开始使用

运算符接受一个或多个参数并生成新值。这个参数与普通方法调用的形式不同，但效果是相同的。加法 `+`、减法 `-`、乘法 `*`、除法 `/` 以及赋值 `=` 在任何编程语言中的工作方式都是类似的。所有运算符都能根据自己的运算对象生成一个值。除此以外，一些运算符可改变运算对象的值，这叫作“副作用”（**Side Effect**）。运算符最常见的用途就是修改自己的运算对象，从而产生副作用。但要注意生成的值亦可由没有副作用的运算符生成。

几乎所有运算符都只能操作基本类型（Primitives）。唯一的例外是 `=`、`==` 和 `!=`，它们能操作所有对象（这也是令人混淆的一个地方）。除此以外，**String** 类支持 `+` 和 `+=`。

5.2 优先级

运算符的优先级决定了存在多个运算符时一个表达式各部分的运算顺序。Java 对运算顺序作出了特别的规定。其中，最简单的规则就是乘法和除法在加法和减法之前完成。程序员经常都会忘记其他优先级规则，所以应该用括号明确规定运算顺序。代码示例:

```
// operators/Precedence.java
public class Precedence {

    public static void main(String[] args) {
        int x = 1, y = 2, z = 3;
```

一天一苹果，医生远离我。

```
        int a = x + y - 2/2 + z; // [1]
        int b = x + (y - 2)/(2 + z); // [2]
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
}
```

输出结果:

```
    a = 5
    b = 1
```

这些语句看起来大致相同，但从输出中我们可以看出它们具有非常不同的含义，具体取决于括号的使用。

我们注意到，在 `System.out.println()` 语句中使用了 + 运算符。但是在这里 + 代表的意思是字符串连接符。编译器会将 + 连接的非字符串尝试转换为字符串。上例中的输出结果说明了 a 和 b 都已经被转化成了字符串。

5.3 赋值

运算符的赋值是由符号 = 完成的。它代表着获取 = 右边的值并赋给左边的变量。右边可以是任何常量、变量或者可产生一个返回值的表达式。但左边必须是一个明确的、已命名的变量。也就是说，必须要有一个物理的空间来存放右边的值。举个例子来说，可将一个常数赋给一个变量（A = 4），但不可将任何东西赋给一个常数（比如不能 4 = A）。

基本类型的赋值都是直接的，而不像对象，赋予的只是其内存的引用。举个例子，a = b ，如果 b 是基本类型，那么赋值操作会将 b 的值复制一份给变量 a，此后若 a 的值发生改变是不会影响到 b 的。作为一名程序员，这应该成为我们的常识。

如果是为对象赋值，那么结果就不一样了。对一个对象进行操作时，我们实际上操作的是它的引用。所以我们将右边的对象赋予给左边时，赋予的只是该对象的引用。此时，两者指向的堆中的对象还是同一个。代码示例:

```
// operators/Assignment.java
// Assignment with objects is a bit tricky
class Tank {
    int level;
}

public class Assignment {

    public static void main(String[] args) {
        Tank t1 = new Tank();
        Tank t2 = new Tank();
        t1.level = 9;
```

```
        t2.level = 47;
        System.out.println("1: t1.level: " + t1.level +
            ", t2.level: " + t2.level);
        t1 = t2;
        System.out.println("2: t1.level: " + t1.level +
            ", t2.level: " + t2.level);
        t1.level = 27;
        System.out.println("3: t1.level: " + t1.level +
            ", t2.level: " + t2.level);
    }
}
```

输出结果:

```
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
```

这是一个简单的 `Tank` 类，在 `main()` 方法创建了两个实例对象。两个对象的 `level` 属性分别被赋予不同的值。然后，t2 的值被赋予给 t1。在许多编程语言里，预期的结果是 t1 和 t2 的值会一直相对独立。但是，在 Java 中，由于赋予的只是对象的引用，改变 t1 也就改变了 t2。这是因为 t1 和 t2 此时指向的是堆中同一个对象。(t1 原始对象的引用在 t2 赋值给其时丢失，它引用的对象会在垃圾回收时被清理)。

这种现象通常称为别名（aliasing），这是 Java 处理对象的一种基本方式。但是假若你不想出现这里的别名引起混淆的话，你可以这么做。代码示例:

```
t1.level = t2.level;
```

较之前的做法，这样做保留了两个单独的对象，而不是丢弃一个并将 t1 和 t2 绑定到同一个对象。但是这样的操作有点违背 Java 的设计原则。对象的赋值是个需要重视的环节，否则你可能收获意外的“惊喜”。

方法调用中的别名现象

当我们把对象传递给方法时，会发生别名现象。

```
// operators/PassObject.java
// 正在传递的对象可能不是你之前使用的
class Letter {
    char c;
}

public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }

    public static void main(String[] args) {
        Letter x = new Letter();
```

```
        x.c = 'a';
        System.out.println("1: x.c: " + x.c);
        f(x);
        System.out.println("2: x.c: " + x.c);
    }
}
```

输出结果:

```
1: x.c: a
2: x.c: z
```

在许多编程语言中，方法 `f()` 似乎会在内部复制其参数 **Letter y**。但是一旦传递了一个引用，那么实际上 `y.c ='z';` 是在方法 `f()` 之外改变对象。别名现象以及其解决方案是个复杂的问题，在附录中有包含：对象传递和返回。意识到这一点，我们可以警惕类似的陷阱。

5.4 算术运算符

Java 的基本算术运算符与其他大多编程语言是相同的。其中包括加号 +、减号 -、除号 /、乘号 * 以及取模 %（从整数除法中获得余数）。整数除法会直接砍掉小数，而不是进位。

Java 也用一种与 C++ 相同的简写形式同时进行运算和赋值操作，由运算符后跟等号表示，并且与语言中的所有运算符一致（只要有意义）。可用 x += 4 来表示：将 x 的值加上 4 的结果再赋值给 x。更多代码示例:

```
// operators/MathOps.java
// The mathematical operators
import java.util.*;

public class MathOps {
    public static void main(String[] args) {
        // Create a seeded random number generator:
        Random rand = new Random(47);
        int i, j, k;
        // Choose value from 1 to 100:
        j = rand.nextInt(100) + 1;
        System.out.println("j : " + j);
        k = rand.nextInt(100) + 1;
        System.out.println("k : " + k);
        i = j + k;
        System.out.println("j + k : " + i);
        i = j - k;
        System.out.println("j - k : " + i);
        i = k / j;
        System.out.println("k / j : " + i);
        i = k * j;
        System.out.println("k * j : " + i);
```

```
        i = k % j;
        System.out.println("k % j : " + i);
        j %= k;
        System.out.println("j %= k : " + j);
        // 浮点运算测试
        float u, v, w; // Applies to doubles, too
        v = rand.nextFloat();
        System.out.println("v : " + v);
        w = rand.nextFloat();
        System.out.println("w : " + w);
        u = v + w;
        System.out.println("v + w : " + u);
        u = v - w;
        System.out.println("v - w : " + u);
        u = v * w;
        System.out.println("v * w : " + u);
        u = v / w;
        System.out.println("v / w : " + u);
        // 下面的操作同样适用于 char,
        // byte, short, int, long, and double:
        u += v;
        System.out.println("u += v : " + u);
        u -= v;
        System.out.println("u -= v : " + u);
        u *= v;
        System.out.println("u *= v : " + u);
        u /= v;
        System.out.println("u /= v : " + u);
    }
}
```

输出结果:

```
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
```

为了生成随机数字，程序首先创建一个 **Random** 对象。不带参数的 **Random** 对象会利用当前的时间用作随机数生成器的“种子”（seed），从而为程序的每次执行生成不同的输出。在本书的示例中，重要的是每个示例末尾的输出尽可能一致，以便可以使用外部工具进行验证。所以我们通过在创建 **Random** 对象时提供种子（随机数生成器的初始化值，其始终为特定种子值产生相同的序列），让程序每次执行都生成相同的随机数，如此以来输出结果就是可验证的 [1]。若需要生成随机值，可删除代码示例中的种子参数。该对象通过调用方法 `nextInt()` 和 `nextFloat()`（还可以调用 `nextLong()` 或 `nextDouble()`），使用 **Random** 对象生成许多不同类型的随机数。`nextInt()` 的参数设置生成的数字的上限，下限为零，为了避免零除的可能性，结果偏移 1。

5.4.1 一元加减运算符

一元加 + 减 - 运算符的操作和二元是相同的。编译器可自动识别使用何种方式解析运算：

```
x = -a;
```

上例的代码表意清晰，编译器可正确识别。下面再看一个示例：

```
x = a * -b;
```

虽然编译器可以正确的识别，但是程序员可能会迷惑。为了避免混淆，推荐下面的写法：

```
x = a * (-b);
```

一元减号可以得到数据的负值。一元加号的作用相反，不过它唯一能影响的就是把较小的数值类型自动转换为 **int** 类型。

5.5 递增和递减

和 C 语言类似，Java 提供了许多快捷运算方式。快捷运算可使代码可读性，可写性都更强。其中包括递增 `++` 和递减 `--`，意为“增加或减少一个单位”。举个例子来说，假设 a 是一个 **int** 类型的值，则表达式 `++a` 就等价于 `a = a + 1`。递增和递减运算符不仅可以修改变量，还可以生成变量的值。

每种类型的运算符，都有两个版本可供选用；通常将其称为“前缀”和“后缀”。“前递增”表示 `++` 运算符位于变量或表达式的前面；而“后递增”表示 `++` 运算符位于变量的后面。类似地，“前递减”意味着 `--` 运算符位于变量的前面；而“后递减”意味着 `--` 运算符位于变量的后面。对于前递增和前递减（如 `++a` 或 `--a`），会先执行递增/减运算，再返回值。而对于后递增和后递减（如 `a++` 或 `a--`），会先返回值，再执行递增/减运算。代码示例：

[1] 功能粘贴在一起的方法的确有点与众不同，但它仍不失为一个库。

```java
// operators/AutoInc.java
// 演示 ++ 和 -- 运算符
public class AutoInc {
    public static void main(String[] args) {
        int i = 1;
        System.out.println("i: " + i);
        System.out.println("++i: " + ++i); // 前递增
        System.out.println("i++: " + i++); // 后递增
        System.out.println("i: " + i);
        System.out.println("--i: " + --i); // 前递减
        System.out.println("i--: " + i--); // 后递减
        System.out.println("i: " + i);
    }
}
```

输出结果:

```
i: 1
++i: 2
i++: 2
i: 3
--i: 2
i--: 2
i: 1
```

对于前缀形式，我们将在执行递增/减操作后获取值；使用后缀形式，我们将在执行递增/减操作之前获取值。它们是唯一具有“副作用”的运算符（除那些涉及赋值的以外）—— 它们修改了操作数的值。

C++ 名称来自于递增运算符，暗示着“比 C 更进一步”。在早期的 Java 演讲中，*Bill Joy*（Java 作者之一）说“**Java = C++ –**”（C++ 减减），意味着 Java 在 C++ 的基础上减少了许多不必要的东西，因此语言更简单。随着进一步地学习，我们会发现 Java 的确有许多地方相对 C++ 来说更简便，但是在其他方面，难度并不会比 C++ 小多少。

5.6 关系运算符

关系运算符会通过产生一个布尔（**boolean**）结果来表示操作数之间的关系。如果关系为真，则结果为 **true**，如果关系为假，则结果为 **false**。关系运算符包括小于 <，大于 >，小于或等于 <=，大于或等于 >=，等于 == 和不等于 ! =。== 和 != 可用于所有基本类型，但其他运算符不能用于基本类型 **boolean**，因为布尔值只能表示 **true** 或 **false**，所以比较它们之间的“大于”或“小于”没有意义。

一年之计在于春。

5.6.1 测试对象等价

关系运算符 == 和 != 同样适用于所有对象之间的比较运算，但它们比较的内容却经常困扰 Java 的初学者。下面是代码示例：

```
// operators/Equivalence.java
public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = 47;
        Integer n2 = 47;
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
    }
}
```

输出结果：

```
true
false
```

表达式 `System.out.println(n1 == n2)` 将会输出比较的结果。因为两个 **Integer** 对象相同，所以先输出 **true**，再输出 **false**。但是，尽管对象的内容一样，对象的引用却不一样。== 和 != 比较的是对象引用，所以输出实际上应该是先输出 **false**，再输出 **true**（译者注：如果你把 47 改成 128，那么打印的结果就是这样，因为 Integer 内部维护着一个 IntegerCache 的缓存，默认缓存范围是 [-128, 127]，所以 [-128, 127] 之间的值用 == 和 != 比较也能能到正确的结果，但是不推荐用关系运算符比较，具体见 JDK 中的 Integer 类源码）。

那么怎么比较两个对象的内容是否相同呢？你必须使用所有对象（不包括基本类型）中都存在的 `equals()` 方法，下面是如何使用 `equals()` 方法的示例：

```
// operators/EqualsMethod.java
public class EqualsMethod {
    public static void main(String[] args) {
        Integer n1 = 47;
        Integer n2 = 47;
        System.out.println(n1.equals(n2));
    }
}
```

输出结果：

```
true
```

上例的结果看起来是我们所期望的。但其实事情并非那么简单。下面我们来创建自己的类：

```
// operators/EqualsMethod2.java
// 默认的 equals() 方法没有比较内容
class Value {
    int i;
```

```
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
}
```

输出结果:

```
false
```

上例的结果再次令人困惑：结果是 **false**。原因：`equals()` 的默认行为是比较对象的引用而非具体内容。因此，除非你在新类中覆写 `equals()` 方法，否则我们将获取不到想要的结果。不幸的是，在学习 复用（**Reuse**）章节后我们才能接触到“覆写”（**Override**），并且直到 集合主题，才能知道定义 `equals()` 方法的正确方式，但是现在明白 `equals()` 行为方式也可能为你节省一些时间。

大多数 Java 库类通过覆写 `equals()` 方法比较对象的内容而不是其引用。

5.7 逻辑运算符

每个逻辑运算符 `&&` （**AND**）、`||`（**OR**）和 `!`（**非**）根据参数的逻辑关系生成布尔值 **`true`** 或 **`false`**。下面的代码示例使用了关系运算符和逻辑运算符:

```
// operators/Bool.java
// 关系运算符和逻辑运算符
import java.util.*;
public class Bool {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt(100);
        int j = rand.nextInt(100);
        System.out.println("i = " + i);
        System.out.println("j = " + j);
        System.out.println("i > j is " + (i > j));
        System.out.println("i < j is " + (i < j));
        System.out.println("i >= j is " + (i >= j));
        System.out.println("i <= j is " + (i <= j));
        System.out.println("i == j is " + (i == j));
        System.out.println("i != j is " + (i != j));
        // 将 int 作为布尔处理不是合法的 Java 写法
        //- System.out.println("i && j is " + (i && j));
        //- System.out.println("i || j is " + (i || j));
        //- System.out.println("!i is " + !i);
```

一见钟情难维久。

```
        System.out.println("(i < 10) && (j < 10) is "
        + ((i < 10) && (j < 10)) );
        System.out.println("(i < 10) || (j < 10) is "
        + ((i < 10) || (j < 10)) );
    }
}
```

输出结果:

```
i = 58
j = 55
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is false
```

在 Java 逻辑运算中，我们不能像 C/C++ 那样使用非布尔值，而仅能使用 **AND**、**OR**、**NOT**。上面的例子中，我们将使用非布尔值的表达式注释掉了（你可以看到表达式前面是 //-）。但是，后续的表达式使用关系比较生成布尔值，然后对结果使用了逻辑运算。请注意，如果在预期为 **String** 类型的位置使用 **boolean** 类型的值，则结果会自动转为适当的文本格式（即 “true” 或 “false” 字符串）。

我们可以将前一个程序中 **int** 的定义替换为除 **boolean** 之外的任何其他基本数据类型。但请注意，**float** 类型的数值比较非常严格，只要两个数字的最小位不同则两个数仍然不相等；只要数字最小位是大于 0 的，那么它就不等于 0。

5.7.1 短路

逻辑运算符支持一种称为“短路”（short-circuiting）的现象。整个表达式会在运算到可以明确结果时就停止并返回结果，这意味着该逻辑表达式的后半部分不会被执行到。代码示例:

```
// operators / ShortCircuit.java
// 逻辑运算符的短路行为
public class ShortCircuit {

    static boolean test1(int val) {
        System.out.println("test1(" + val + ")");
        System.out.println("result: " + (val < 1));
        return val < 1;
    }

    static boolean test2(int val) {
        System.out.println("test2(" + val + ")");
        System.out.println("result: " + (val < 2));
```

```
        return val < 2;
    }

    static boolean test3(int val) {
        System.out.println("test3(" + val + ")");
        System.out.println("result: " + (val < 3));
        return val < 3;
    }

    public static void main(String[] args) {
        boolean b = test1(0) && test2(2) && test3(2);
        System.out.println("expression is " + b);
    }
}
```

输出结果:

```
test1(0)
result: true
test2(2)
result: false
expression is false
```

每个测试都对参数执行比较并返回 **true** 或 **false**。同时控制台也会在方法执行时打印他们的执行状态。下面的表达式:

```
test1 (0) && test2 (2) && test3 (2)
```

可能你的预期是程序会执行 3 个 **test** 方法并返回。我们来分析一下: 第一个方法的结果返回 **true**, 因此表达式会继续走下去。紧接着, 第二个方法的返回结果是 **false**。这就代表这整个表达式的结果肯定为 **false**, 所以就没有必要再判断剩下的表达式部分了。

所以, 运用"短路"可以节省部分不必要的运算, 从而提高程序潜在的性能。

5.8 字面值常量

通常, 当我们向程序中插入一个字面值常量 (**Literal**) 时, 编译器会确切地识别它的类型。当类型不明确时, 必须辅以字面值常量关联来帮助编译器识别。代码示例:

```
// operators/Literals.java
public class Literals {
    public static void main(String[] args) {
        int i1 = 0x2f; // 16进制 (小写)
        System.out.println(
        "i1: " + Integer.toBinaryString(i1));
        int i2 = 0X2F; // 16进制 (大写)
        System.out.println(
        "i2: " + Integer.toBinaryString(i2));
```

```
        int i3 = 0177; // 8进制 (前导0)
        System.out.println(
        "i3: " + Integer.toBinaryString(i3));
        char c = 0xffff; // 最大 char 型16进制值
        System.out.println(
        "c: " + Integer.toBinaryString(c));
        byte b = 0x7f; // 最大 byte 型16进制值  10101111;
        System.out.println(
        "b: " + Integer.toBinaryString(b));
        short s = 0x7fff; // 最大 short 型16进制值
        System.out.println(
        "s: " + Integer.toBinaryString(s));
        long n1 = 200L; // long 型后缀
        long n2 = 200l; // long 型后缀 (容易与数值1混淆)
        long n3 = 200;

        // Java 7 二进制字面值常量:
        byte blb = (byte)0b00110101;
        System.out.println(
        "blb: " + Integer.toBinaryString(blb));
        short bls = (short)0B0010111110101111;
        System.out.println(
        "bls: " + Integer.toBinaryString(bls));
        int bli = 0b00101111101011111010111110101111;
        System.out.println(
        "bli: " + Integer.toBinaryString(bli));
        long bll = 0b0010111110101111101011110101111;
        System.out.println(
        "bll: " + Long.toBinaryString(bll));
        float f1 = 1;
        float f2 = 1F; // float 型后缀
        float f3 = 1f; // float 型后缀
        double d1 = 1d; // double 型后缀
        double d2 = 1D; // double 型后缀
        // (long 型的字面值同样适用于十六进制和8进制 )
    }
}
```

输出结果:

```
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 111111111111111
blb: 110101
bls: 10111110101111
bli: 10111110101111101011110101111
bll: 10111110101111101011110101111
```

在文本值的后面添加字符可以让编译器识别该文本值的类型。对于 **Long** 型数值，结尾使用大写 `L` 或小写 `l` 皆可（不推荐使用 `l`，因为容易与阿拉伯数值 1 混淆）。大写 `F` 或小写 `f` 表示 **float** 浮点数。大写 `D` 或小写 `d` 表示 **double** 双精度。

十六进制（以 16 为基数），适用于所有整型数据类型，由前导 `0x` 或 `0X` 表示，后跟 0-9 或 a-f（大写或小写）。如果我们在初始化某个类型的数值时，赋值超出其范围，那么编译器会报错（不管值的数字形式如何）。在上例的代码中，**char**、**byte** 和 **short** 的值已经是最大了。如果超过这些值，编译器将自动转型为 **int**，并且提示我们需要声明强制转换（强制转换将在本章后面定义），意味着我们已越过该类型的范围界限。

八进制（以 8 为基数）由 0~7 之间的数字和前导零 `0` 表示。

Java 7 引入了二进制的字面值常量，由前导 `0b` 或 `0B` 表示，它可以初始化所有的整数类型。

使用整型数值类型时,显示其二进制形式会很有用。在 Long 型和 Integer 型中这很容易实现,调用其静态的 `toBinaryString()` 方法即可。但是请注意,若将较小的类型传递给 **Integer.**`tobinarystring()` 时，类型将自动转换为 **int**。

5.8.1 下划线

Java 7 中有一个深思熟虑的补充：我们可以在数字字面量中包含下划线 _，以使结果更清晰。这对于大数值的分组特别有用。代码示例：

```
// operators/Underscores.java
public class Underscores {
    public static void main(String[] args) {
        double d = 341_435_936.445_667;
        System.out.println(d);
        int bin = 0b0010_1111_1010_1111_1010_1111_1010_1111;
        System.out.println(Integer.toBinaryString(bin));
        System.out.printf("%x%n", bin); // [1]
        long hex = 0x7f_e9_b7_aa;
        System.out.printf("%x%n", hex);
    }
}
```

输出结果：

```
3.41435936445667E8
10111110101111101011111010111
2fafafaf
7fe9b7aa
```

下面是合理使用的规则：

1. 仅限单 _，不能多条相连。
2. 数值开头和结尾不允许出现 _。

3. F、D 和 L的前后禁止出现 _。
4. 二进制前导 b 和十六进制 x 前后禁止出现 _。

[1] 注意 %n的使用。熟悉 C 风格的程序员可能习惯于看到 \n 来表示换行符。问题在于它给你的是一个“Unix 风格”的换行符。此外，如果我们使用的是 Windows，则必须指定 \r\n。这种差异的包袱应该由编程语言来解决。这就是 Java 用 %n 实现的可以忽略平台间差异而生成适当的换行符，但只有当你使用 System.out.printf() 或 System.out.format() 时。对于 System.out.println()，我们仍然必须使用 \n；如果你使用 %n，println() 只会输出 %n 而不是换行符。

5.8.2 指数计数法

指数总是采用一种我认为很不直观的记号方法:

```
// operators/Exponents.java
// "e" 表示 10 的几次幂
public class Exponents {
    public static void main(String[] args) {
        // 大写 E 和小写 e 的效果相同:
        float expFloat = 1.39e-43f;
        expFloat = 1.39E-43f;
        System.out.println(expFloat);
        double expDouble = 47e47d; // 'd' 是可选的
        double expDouble2 = 47e47; // 自动转换为 double
        System.out.println(expDouble);
    }
}
```

输出结果:

```
1.39E-43
4.7E48
```

在科学与工程学领域，**e** 代表自然对数的基数，约等于 2.718 （Java 里用一种更精确的 **double** 值 **Math.E** 来表示自然对数）。指数表达式 “1.39 x e-43”，意味着 “1.39 × 2.718 的 -43 次方”。然而，自 FORTRAN 语言发明后，人们自然而然地觉得 e 代表 “10 的几次幂”。这种做法显得颇为古怪，因为 FORTRAN 最初是为科学与工程领域设计的。

理所当然，它的设计者应对这样的混淆概念持谨慎态度 [2]。但不管怎样，这种特别的表达方法在 C，C++ 以及现在的 Java 中顽固地保留下来了。所以倘若习惯 e 作为自然对数的基数使用，那么在 Java 中看到类似“1.39e-43f”这样的表达式时，请转换你的思维，从程序设计的角度思考它；它真正的含义是 “1.39 × 10 的 -43 次方”。

注意如果编译器能够正确地识别类型，就不必使用后缀字符。对于下述语句:

```
long n3 = 200;
```

[2] 例如，这个电子书是利用 Pandoc 制作出来的，它是用纯函数式语言 Haskell 编写的一个程序。

它并不存在含糊不清的地方，所以 200 后面的 L 大可省去。然而，对于下述语句：

```
float f4 = 1e-43f; //10 的幂数
```

编译器通常会将指数作为 **double** 类型来处理，所以假若没有这个后缀字符 `f`，编译器就会报错，提示我们应该将 **double** 型转换成 **float** 型。

5.9 位运算符

位运算符允许我们操作一个整型数字中的单个二进制位。位运算符会对两个整数对应的位执行布尔代数，从而产生结果。

位运算源自 C 语言的底层操作。我们经常要直接操纵硬件，频繁设置硬件寄存器内的二进制位。Java 的设计初衷是电视机顶盒嵌入式开发，所以这种底层的操作仍被保留了下来。但是，你可能不会使用太多位运算。

若两个输入位都是 1，则按位“与运算符” `&` 运算后结果是 1，否则结果是 0。若两个输入位里至少有一个是 1，则按位“或运算符” `|` 运算后结果是 1；只有在两个输入位都是 0 的情况下，运算结果才是 0。若两个输入位的某一个是 1，另一个不是 1，那么按位“异或运算符” `^` 运算后结果才是 1。按位“非运算符” `~` 属于一元运算符；它只对一个自变量进行操作（其他所有运算符都是二元运算符）。按位非运算后结果与输入位相反。例如输入 0，则输出 1；输入 1，则输出 0。

位运算符和逻辑运算符都使用了同样的字符，只不过数量不同。位短，所以位运算符只有一个字符。位运算符可与等号 `=` 联合使用以接收结果及赋值：`&=`，`|=` 和 `^=` 都是合法的（由于 `~` 是一元运算符，所以不可与 `=` 联合使用）。

我们将 **Boolean** 类型被视为“单位值”（one-bit value），所以它多少有些独特的地方。我们可以对 boolean 型变量执行与、或、异或运算，但不能执行非运算（大概是为了避免与逻辑“非”混淆）。对于布尔值，位运算符具有与逻辑运算符相同的效果，只是它们不会中途“短路”。此外，针对布尔值进行的位运算为我们新增了一个“异或”逻辑运算符，它并未包括在逻辑运算符的列表中。在移位表达式中，禁止使用布尔值，原因将在下面解释。

5.10 移位运算符

移位运算符面向的运算对象也是二进制的“位”。它们只能用于处理整数类型（基本类型的一种）。左移位运算符 `<<` 能将其左边的运算对象向左移动右侧指定的位数（在低位补 0）。右移位运算符 `>>` 则相反。右移位运算符有“正”、“负”值：若值为正，则在高位插入 0；若值为负，则在高位插入 1。Java 也添加了一种“不分正负”的右移位运算符（`>>>`），它使用了“零扩展”（zero extension）：无论正负，都在高位插入 0。这一运算符是 C/C++ 没有的。

如果移动 **char**、**byte** 或 **short**，则会在移动发生之前将其提升为 **int**，结果为 **int**。仅使用右值（rvalue）的 5 个低阶位。这可以防止我们移动超过 **int** 范围的位数。若对一个 **long** 值进行处理，最后得到的结果也是 **long**。

移位可以与等号 <<= 或 >>= 或 >>>= 组合使用。左值被替换为其移位运算后的值。但是，问题来了，当无符号右移与赋值相结合时，若将其与 **byte** 或 **short** 一起使用的话，则结果错误。取而代之的是，它们被提升为 **int** 型并右移，但在重新赋值时被截断。在这种情况下，结果为 -1。下面是代码示例：

```java
// operators/URShift.java
// 测试无符号右移
public class URShift {
    public static void main(String[] args) {
        int i = -1;
        System.out.println(Integer.toBinaryString(i));
        i >>>= 10;
        System.out.println(Integer.toBinaryString(i));
        long l = -1;
        System.out.println(Long.toBinaryString(l));
        l >>>= 10;
        System.out.println(Long.toBinaryString(l));
        short s = -1;
        System.out.println(Integer.toBinaryString(s));
        s >>>= 10;
        System.out.println(Integer.toBinaryString(s));
        byte b = -1;
        System.out.println(Integer.toBinaryString(b));
        b >>>= 10;
        System.out.println(Integer.toBinaryString(b));
        b = -1;
        System.out.println(Integer.toBinaryString(b));
        System.out.println(Integer.toBinaryString(b>>>10));
    }
}
```

输出结果：

```
11111111111111111111111111111111
11111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
```

在上例中，结果并未重新赋值给变量 **b**，而是直接打印出来，因此一切正常。下面是一个涉及所有位运算符的代码示例：

```java
// operators/BitManipulation.java
// 使用位运算符
import java.util.*;
public class BitManipulation {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt();
        int j = rand.nextInt();
        printBinaryInt("-1", -1);
        printBinaryInt("+1", +1);
        int maxpos = 2147483647;
        printBinaryInt("maxpos", maxpos);
        int maxneg = -2147483648;
        printBinaryInt("maxneg", maxneg);
        printBinaryInt("i", i);
        printBinaryInt("~i", ~i);
        printBinaryInt("-i", -i);
        printBinaryInt("j", j);
        printBinaryInt("i & j", i & j);
        printBinaryInt("i | j", i | j);
        printBinaryInt("i ^ j", i ^ j);
        printBinaryInt("i << 5", i << 5);
        printBinaryInt("i >> 5", i >> 5);
        printBinaryInt("(~i) >> 5", (~i) >> 5);
        printBinaryInt("i >>> 5", i >>> 5);
        printBinaryInt("(~i) >>> 5", (~i) >>> 5);
        long l = rand.nextLong();
        long m = rand.nextLong();
        printBinaryLong("-1L", -1L);
        printBinaryLong("+1L", +1L);
        long ll = 9223372036854775807L;
        printBinaryLong("maxpos", ll);
        long lln = -9223372036854775808L;
        printBinaryLong("maxneg", lln);
        printBinaryLong("l", l);
        printBinaryLong("~l", ~l);
        printBinaryLong("-l", -l);
        printBinaryLong("m", m);
        printBinaryLong("l & m", l & m);
        printBinaryLong("l | m", l | m);
        printBinaryLong("l ^ m", l ^ m);
        printBinaryLong("l << 5", l << 5);
        printBinaryLong("l >> 5", l >> 5);
        printBinaryLong("(~l) >> 5", (~l) >> 5);
        printBinaryLong("l >>> 5", l >>> 5);
        printBinaryLong("(~l) >>> 5", (~l) >>> 5);
    }

    static void printBinaryInt(String s, int i) {
        System.out.println(
```

```
        s + ", int: " + i + ", binary:\n " +
        Integer.toBinaryString(i));
    }

    static void printBinaryLong(String s, long l) {
        System.out.println(
        s + ", long: " + l + ", binary:\n " +
        Long.toBinaryString(l));
    }
}
```

输出结果（前 32 行）:

```
-1, int: -1, binary:
11111111111111111111111111111111
+1, int: 1, binary:
1
maxpos, int: 2147483647, binary:
1111111111111111111111111111111
maxneg, int: -2147483648, binary:
10000000000000000000000000000000
i, int: -1172028779, binary:
10111010001001000100001010010101
~i, int: 1172028778, binary:
 1000101110110111011110101101010
-i, int: 1172028779, binary:
1000101110110111011110101101011
j, int: 1717241110, binary:
1100110010110110000010100010110
i & j, int: 570425364, binary:
100010000000000000000000010100
i | j, int: -25213033, binary:
11111110011111110100011110010111
i ^ j, int: -595638397, binary:
11011100011111110100011110000011
i << 5, int: 1149784736, binary:
1000100100010000101001010100000
i >> 5, int: -36625900, binary:
11111101110100010010001000010100
(~i) >> 5, int: 36625899, binary:
1000101110110111011101011
i >>> 5, int: 97591828, binary:
1011101000100100010000010100
(~i) >>> 5, int: 36625899, binary:
1000101110110111011101011
    ...
```

结尾的两个方法 `printBinaryInt()` 和 `printBinaryLong()` 分别操作一个 **int** 和 **long** 值，并转换为二进制格式输出，同时附有简要的文字说明。除了演示 **int** 和 **long** 的所有位运算符的效果之外，本示例还显示 **int** 和 **long** 的最小值、最大值、+1 和 -1 值，以便我们了解它们的形式。注意高位代

表符号：0 表示正，1 表示负。上面显示了 **int** 部分的输出。以上数字的二进制表示形式是带符号的补码（2's complement）。

5.11 三元运算符

三元运算符，也称为条件运算符。这种运算符比较罕见，因为它有三个运算对象。但它确实属于运算符的一种，因为它最终也会生成一个值。这与本章后一节要讲述的普通 **if-else** 语句是不同的。下面是它的表达式格式：

布尔表达式? 值 1 : 值 2

若表达式计算为 **true**，则返回结果 **值 1**；如果表达式的计算为 **false**，则返回结果 **值 2**。

当然，也可以换用普通的 **if-else** 语句（在后面介绍），但三元运算符更加简洁。作为三元运算符的创造者，C 自诩为一门简练的语言。三元运算符的引入多半就是为了高效编程，但假若我们打算频繁使用它的话，还是先多作一些思量：它易于产生可读性差的代码。与 **if-else** 不同的是，三元运算符是有返回结果的。请看下面的代码示例：

```java
// operators/TernaryIfElse.java
public class TernaryIfElse {

static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
}

static int standardIfElse(int i) {
    if(i < 10)
        return i * 100;
    else
        return i * 10;
}

    public static void main(String[] args) {
        System.out.println(ternary(9));
        System.out.println(ternary(10));
        System.out.println(standardIfElse(9));
        System.out.println(standardIfElse(10));
    }
}
```

输出结果：

```
900
100
900
100
```

可以看出，`ternary()` 中的代码更简短。然而，**standardIfElse()** 中的代码更易理解且不要求

更多的录入。所以我们在挑选三元运算符时，请务必权衡一下利弊。

5.12 字符串运算符

这个运算符在 Java 里有一项特殊用途：连接字符串。这点已在前面展示过了。尽管与 + 的传统意义不符，但如此使用也还是比较自然的。这一功能看起来还不错，于是在 C++ 里引入了“运算符重载”机制，以便 C++ 程序员为几乎所有运算符增加特殊的含义。但遗憾得是，与 C++ 的一些限制结合以后，它变得复杂。这要求程序员在设计自己的类时必须对此有周全的考虑。虽然在 Java 中实现运算符重载机制并非难事（如 C# 所展示的，它具有简单的运算符重载），但因该特性过于复杂，因此 Java 并未实现它。

我们注意到运用 `String` + 时有一些有趣的现象。若表达式以一个 **String** 类型开头（编译器会自动将双引号 `""` 标注的的字符序列转换为字符串），那么后续所有运算对象都必须是字符串。代码示例：

```
// operators/StringOperators.java
public class StringOperators {
    public static void main(String[] args) {
        int x = 0, y = 1, z = 2;
        String s = "x, y, z ";
        System.out.println(s + x + y + z);
        // 将 x 转换为字符串
        System.out.println(x + " " + s);
        s += "(summed) = ";
        // 级联操作
        System.out.println(s + (x + y + z));
        // Integer.toString()方法的简写:
        System.out.println("" + x);
    }
}
```

输出结果：

```
x, y, z 012
0 x, y, z
x, y, z (summed) = 3
0
```

注意：上例中第 1 输出语句的执行结果是 `012` 而并非 `3`，这是因为编译器将其分别转换为其字符串形式然后与字符串变量 **s** 连接。在第 2 条输出语句中，编译器将开头的变量转换为了字符串，由此可以看出，这种转换与数据的位置无关，只要当中有一条数据是字符串类型，其他非字符串数据都将被转换为字符串形式并连接。最后一条输出语句，我们可以看出 `+=` 运算符可以拼接其右侧的字符串连接结果并重赋值给自身变量 `s`。括号 `()` 可以控制表达式的计算顺序，以便在显示 **int** 之前对其进行实际求和。

请注意主方法中的最后一个例子：我们经常会看到一个空字符串 `""` 跟着一个基本类型的数据。这

样可以隐式地将其转换为字符串，以代替繁琐的显式调用方法（如这里可以使用 **Integer.toString()**）。

5.13 常见陷阱

使用运算符时很容易犯的一个错误是，在还没搞清楚表达式的计算方式时就试图忽略括号 `()`。在 Java 中也一样。在 C++ 中你甚至可能犯这样极端的错误. 代码示例：

```
while(x = y) {
// ...
}
```

显然，程序员原意是测试等价性 `==`，而非赋值 `=`。若变量 **y** 非 0 的话，在 C/C++ 中，这样的赋值操作总会返回 **`true`**。于是，上面的代码示例将会无限循环。而在 Java 中，这样的表达式结果并不会转化为一个布尔值。而编译器会试图把这个 **int** 型数据转换为预期应接收的布尔类型。最后，我们将会在试图运行前收到编译期错误。因此，Java 天生避免了这种陷阱发生的可能。

唯一有种情况例外：当变量 `x` 和 `y` 都是布尔值，例如 `x=y` 是一个逻辑表达式。除此之外，之前的那个例子，很大可能是错误。

在 C/C++ 里，类似的一个问题还有使用按位“与” `&` 和“或” `|` 运算，而非逻辑“与” `&&` 和“或” `||`。就象 `=` 和 `==` 一样，键入一个字符当然要比键入两个简单。在 Java 中，编译器同样可防止这一点，因为它不允许我们强行使用另一种并不符的类型。

5.14 类型转换

“类型转换”（Casting）的作用是“与一个模型匹配”。在适当的时候，Java 会将一种数据类型自动转换成另一种。例如，假设我们为 **float** 变量赋值一个整数值，计算机会将 **int** 自动转换成 **float**。我们可以在程序未自动转换时显式、强制地使此类型发生转换。

要执行强制转换，需要将所需的数据类型放在任何值左侧的括号内，如下所示：

```
// operators/Casting.java
public class Casting {
    public static void main(String[] args) {
        int i = 200;
        long lng = (long)i;
        lng = i; // 没有必要的类型提升
        long lng2 = (long)200;
        lng2 = 200;
        // 类型收缩
        i = (int)lng2; // Cast required
    }
}
```

诚然，你可以这样地去转换一个数值类型的变量。但是上例这种做法是多余的：因为编译器会在必要时自动提升 **int** 型数据为 **long** 型。

当然，为了程序逻辑清晰或提醒自己留意，我们也可以显式地类型转换。在其他情况下，类型转换型只有在代码编译时才显出其重要性。在 C/C++ 中，类型转换有时会让人头痛。在 Java 里，类型转换则是一种比较安全的操作。但是，若将数据类型进行“向下转换”（**Narrowing Conversion**）的操作（将容量较大的数据类型转换成容量较小的类型），可能会发生信息丢失的危险。此时，编译器会强迫我们进行转型，好比在提醒我们：该操作可能危险，若你坚持让我这么做，那么对不起，请明确需要转换的类型。对于“向上转换”（**Widening conversion**），则不必进行显式的类型转换，因为较大类型的数据肯定能容纳较小类型的数据，不会造成任何信息的丢失。

除了布尔类型的数据，Java 允许任何基本类型的数据转换为另一种基本类型的数据。此外，类是不能进行类型转换的。为了将一个类转换为另一个类型，需要使用特殊的方法（后面将会学习到如何在父子类之间进行向上/向下转型，例如，“橡树”可以转换为“树”，反之亦然。而对于“岩石”是无法转换为“树”的）。

5.14.1 截断和舍入

在执行“向下转换”时，必须注意数据的截断和舍入问题。若从浮点值转换为整型值，Java 会做什么呢？例如：浮点数 29.7 被转换为整型值，结果会是 29 还是 30 呢？下面是代码示例：

```
// operators/CastingNumbers.java
// 尝试转换 float 和 double 型数据为整型数据
public class CastingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        System.out.println("(int)above: " + (int)above);
        System.out.println("(int)below: " + (int)below);
        System.out.println("(int)fabove: " + (int)fabove);
        System.out.println("(int)fbelow: " + (int)fbelow);
    }
}
```

输出结果：

```
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
```

因此，答案是，从 **float** 和 **double** 转换为整数值时，小数位将被截断。若你想对结果进行四舍五入，可以使用 `java.lang.Math` 的 `round()` 方法：

```
// operators/RoundingNumbers.java
```

```java
// float 和 double 类型数据的四舍五入
public class RoundingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        System.out.println(
        "Math.round(above): " + Math.round(above));
        System.out.println(
        "Math.round(below): " + Math.round(below));
        System.out.println(
        "Math.round(fabove): " + Math.round(fabove));
        System.out.println(
        "Math.round(fbelow): " + Math.round(fbelow));
    }
}
```

输出结果:

```
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
```

因为 `round()` 方法是 `java.lang` 的一部分，所以我们无需通过 **`import`** 就可以使用。

5.14.2 类型提升

你会发现，如果我们对小于 **int** 的基本数据类型（即 **char**、**byte** 或 **short**）执行任何算术或按位操作，这些值会在执行操作之前类型提升为 **int**，并且结果值的类型为 **int**。若想重新使用较小的类型，必须使用强制转换（由于重新分配回一个较小的类型，结果可能会丢失精度）。通常，表达式中最大的数据类型是决定表达式结果的数据类型。**float** 型和 **double** 型相乘，结果是 **double** 型的；**int** 和 **long** 相加，结果是 **long** 型。

5.15 Java 没有 sizeof

在 C/C++ 中，经常需要用到 `sizeof()` 方法来获取数据项被分配的字节大小。C/C++ 中使用 `sizeof()` 最有说服力的原因是为了移植性，不同数据在不同机器上可能有不同的大小，所以在进行大小敏感的运算时，程序员必须对这些类型有多大做到心中有数。例如，一台计算机可用 32 位来保存整数，而另一台只用 16 位保存。显然，在第一台机器中，程序可保存更大的值。所以，移植是令 C/C++ 程序员颇为头痛的一个问题。

Java 不需要 `sizeof()` 方法来满足这种需求，因为所有类型的大小在不同平台上是相同的。我们不必考虑这个层次的移植问题 —— Java 本身就是一种“与平台无关”的语言。

5.16 运算符总结

上述示例分别向我们展示了哪些基本类型能被用于特定的运算符。基本上，下面的代码示例是对上述所有示例的重复，只不过概括了所有的基本类型。这个文件能被正确地编译，因为我已经把编译不通过的那部分用注释 // 过滤了。代码示例：

```
// operators/AllOps.java
// 测试所有基本类型的运算符操作
// 看看哪些是能被 Java 编译器接受的
public class AllOps {
    // 布尔值的接收测试:
    void f(boolean b) {}
    void boolTest(boolean x, boolean y) {
        // 算数运算符:
        //- x = x * y;
        //- x = x / y;
        //- x = x % y;
        //- x = x + y;
        //- x = x - y;
        //- x++;
        //- x--;
        //- x = +y;
        //- x = -y;
        // 关系运算符和逻辑运算符:
        //- f(x > y);
        //- f(x >= y);
        //- f(x < y);
        //- f(x <= y);
        f(x == y);
        f(x != y);
        f(!y);
        x = x && y;
        x = x || y;
        // 按位运算符:
        //- x = ~y;
        x = x & y;
        x = x | y;
        x = x ^ y;
        //- x = x << 1;
        //- x = x >> 1;
        //- x = x >>> 1;
        // 联合赋值:
        //- x += y;
        //- x -= y;
        //- x *= y;
        //- x /= y;
        //- x %= y;
        //- x <<= 1;
        //- x >>= 1;
```

```
        //- x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // 类型转换:
        //- char c = (char)x;
        //- byte b = (byte)x;
        //- short s = (short)x;
        //- int i = (int)x;
        //- long l = (long)x;
        //- float f = (float)x;
        //- double d = (double)x;
    }

    void charTest(char x, char y) {
        // 算数运算符:
        x = (char)(x * y);
        x = (char)(x / y);
        x = (char)(x % y);
        x = (char)(x + y);
        x = (char)(x - y);
        x++;
        x--;
        x = (char) + y;
        x = (char) - y;
        // 关系和逻辑运算符:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //- f(!x);
        //- f(x && y);
        //- f(x || y);
        // 按位运算符:
        x= (char)~y;
        x = (char)(x & y);
        x = (char)(x | y);
        x = (char)(x ^ y);
        x = (char)(x << 1);
        x = (char)(x >> 1);
        x = (char)(x >>> 1);
        // 联合赋值:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
        x >>= 1;
```

```
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // 类型转换
        //- boolean bl = (boolean)x;
        byte b = (byte)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        float f = (float)x;
        double d = (double)x;
    }

    void byteTest(byte x, byte y) {
        // 算数运算符:
        x = (byte)(x* y);
        x = (byte)(x / y);
        x = (byte)(x % y);
        x = (byte)(x + y);
        x = (byte)(x - y);
        x++;
        x--;
        x = (byte) + y;
        x = (byte) - y;
        // 关系和逻辑运算符:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //- f(!x);
        //- f(x && y);
        //- f(x || y);
        //按位运算符:
        x = (byte)~y;
        x = (byte)(x & y);
        x = (byte)(x | y);
        x = (byte)(x ^ y);
        x = (byte)(x << 1);
        x = (byte)(x >> 1);
        x = (byte)(x >>> 1);
        // 联合赋值:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
        x >>= 1;
```

```
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // 类型转换:
        //- boolean bl = (boolean)x;
        char c = (char)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        float f = (float)x;
        double d = (double)x;
    }

    void shortTest(short x, short y) {
        // 算术运算符:
        x = (short)(x * y);
        x = (short)(x / y);
        x = (short)(x % y);
        x = (short)(x + y);
        x = (short)(x - y);
        x++;
        x--;
        x = (short) + y;
        x = (short) - y;
        // 关系和逻辑运算符:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //- f(!x);
        //- f(x && y);
        //- f(x || y);
        // 按位运算符:
        x = (short) ~ y;
        x = (short)(x & y);
        x = (short)(x | y);
        x = (short)(x ^ y);
        x = (short)(x << 1);
        x = (short)(x >> 1);
        x = (short)(x >>> 1);
        // Compound assignment:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
        x >>= 1;
```

```
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // 类型转换:
        //- boolean bl = (boolean)x;
        char c = (char)x;
        byte b = (byte)x;
        int i = (int)x;
        long l = (long)x;
        float f = (float)x;
        double d = (double)x;
    }

    void intTest(int x, int y) {
        // 算术运算符:
        x = x * y;
        x = x / y;
        x = x % y;
        x = x + y;
        x = x - y;
        x++;
        x--;
        x = +y;
        x = -y;
        // 关系和逻辑运算符:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //- f(!x);
        //- f(x && y);
        //- f(x || y);
        // 按位运算符:
        x = ~y;
        x = x & y;
        x = x | y;
        x = x ^ y;
        x = x << 1;
        x = x >> 1;
        x = x >>> 1;
        // 联合赋值:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
        x >>= 1;
```

```
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // 类型转换:
        //- boolean bl = (boolean)x;
        char c = (char)x;
        byte b = (byte)x;
        short s = (short)x;
        long l = (long)x;
        float f = (float)x;
        double d = (double)x;
    }

    void longTest(long x, long y) {
        // 算数运算符:
        x = x * y;
        x = x / y;
        x = x % y;
        x = x + y;
        x = x - y;
        x++;
        x--;
        x = +y;
        x = -y;
        // 关系和逻辑运算符:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //- f(!x);
        //- f(x && y);
        //- f(x || y);
        // 按位运算符:
        x = ~y;
        x = x & y;
        x = x | y;
        x = x ^ y;
        x = x << 1;
        x = x >> 1;
        x = x >>> 1;
        // 联合赋值:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
        x >>= 1;
```

```
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // 类型转换:
        //- boolean bl = (boolean)x;
        char c = (char)x;
        byte b = (byte)x;
        short s = (short)x;
        int i = (int)x;
        float f = (float)x;
        double d = (double)x;
    }

    void floatTest(float x, float y) {
        // 算数运算符:
        x = x * y;
        x = x / y;
        x = x % y;
        x = x + y;
        x = x - y;
        x++;
        x--;
        x = +y;
        x = -y;
        // 关系和逻辑运算符:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //- f(!x);
        //- f(x && y);
        //- f(x || y);
        // 按位运算符:
        //- x = ~y;
        //- x = x & y;
        //- x = x | y;
        //- x = x ^ y;
        //- x = x << 1;
        //- x = x >> 1;
        //- x = x >>> 1;
        // 联合赋值:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        //- x <<= 1;
        //- x >>= 1;
```

```
        //- x >>>= 1;
        //- x &= y;
        //- x ^= y;
        //- x |= y;
        // 类型转换:
        //- boolean bl = (boolean)x;
        char c = (char)x;
        byte b = (byte)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        double d = (double)x;
    }

    void doubleTest(double x, double y) {
        // 算术运算符:
        x = x * y;
        x = x / y;
        x = x % y;
        x = x + y;
        x = x - y;
        x++;
        x--;
        x = +y;
        x = -y;
        // 关系和逻辑运算符:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //- f(!x);
        //- f(x && y);
        //- f(x || y);
        // 按位运算符:
        //- x = ~y;
        //- x = x & y;
        //- x = x | y;
        //- x = x ^ y;
        //- x = x << 1;
        //- x = x >> 1;
        //- x = x >>> 1;
        // 联合赋值:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        //- x <<= 1;
        //- x >>= 1;
```

```
        //- x >>>= 1;
        //- x &= y;
        //- x ^= y;
        //- x |= y;
        // 类型转换:
        //- boolean bl = (boolean)x;
        char c = (char)x;
        byte b = (byte)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        float f = (float)x;
    }
}
```

注意：**boolean** 类型的运算是受限的。你能为其赋值 **true** 或 **false**，也可测试它的值是否是 **true** 或 **false**。但你不能对其作加减等其他运算。

在 **char**，**byte** 和 **short** 类型中，我们可以看到算术运算符的“类型转换”效果。我们必须要显式强制类型转换才能将结果重新赋值为原始类型。对于 **int** 类型的运算则不用转换，因为默认就是 **int** 型。虽然我们不用再停下来思考这一切是否安全，但是两个大的 int 型整数相乘时，结果有可能超出 **int** 型的范围，这种情况下结果会发生溢出。下面的代码示例：

```
// operators/Overflow.java
// 厉害了！内存溢出
public class Overflow {
    public static void main(String[] args) {
        int big = Integer.MAX_VALUE;
        System.out.println("big = " + big);
        int bigger = big * 4;
        System.out.println("bigger = " + bigger);
    }
}
```

输出结果：

```
big = 2147483647
bigger = -4
```

编译器没有报错或警告，运行时一切看起来都无异常。诚然，Java 是优秀的，但是还不足够优秀。

对于 **char**，**byte** 或者 **short**，混合赋值并不需要类型转换。即使为它们执行转型操作，也会获得与直接算术运算相同的结果。另外，省略类型转换可以使代码显得更加简练。总之，除 **boolean** 以外，其他任何两种基本类型间都可进行类型转换。当我们进行向下转换类型时，需要注意结果的范围是否溢出，否则我们就很可能在不知不觉中丢失精度。

5.17 本章小结

如果你已接触过一门 C 语法风格编程语言，那么你在学习 Java 的运算符时实际上没有任何曲线。如果你觉得有难度，那么我推荐你要先去 www.OnJava8.com 观看《Thinking in C》的视频教程来补充一些前置知识储备。

A new broom sweeps clean.

6 控制流

> 程序必须在执行过程中控制它的世界并做出选择。在 Java 中，你需要执行控制语句来做出选择。

Java 使用了 C 的所有执行控制语句，因此对于熟悉 C/C++ 编程的人来说，这部分内容轻车熟路。大多数面向过程编程语言都有共通的某种控制语句。在 Java 中，涉及的关键字包括 **if-else**，**while**，**do-while**，**for**，**return**，**break** 和选择语句 **switch**。Java 并不支持备受诟病的 **goto**（尽管它在某些特殊场景中依然是最行之有效的方法）。尽管如此，在 Java 中我们仍旧可以进行类似的逻辑跳转，但较之典型的 **goto** 用法限制更多。

6.1 true 和 false

所有的条件语句都利用条件表达式的“真”或“假”来决定执行路径。举例：`a == b`。它利用了条件表达式 `==` 来比较 `a` 与 `b` 的值是否相等。该表达式返回 **`true`** 或 **`false`**。代码示例：

```java
// control/TrueFalse.java
public class TrueFalse {
    public static void main(String[] args) {
        System.out.println(1 == 1);
        System.out.println(1 == 2);
    }
}
```

输出结果：

```
true false
```

通过上一章的学习，我们知道任何关系运算符都可以产生条件语句。**注意**：在 Java 中使用数值作为布尔值是非法的，即便这种操作在 C/C++ 中是被允许的（在这些语言中，“真”为非零，而“假”是零）。如果想在布尔测试中使用一个非布尔值，那么首先需要使用条件表达式来产生 **boolean** 类型的结果，例如 `if(a != 0)`。

新官上任三把火。

6.2 if-else

if-else 语句是控制程序执行流程最基本的形式。其中 **else** 是可选的，因此可以有两种形式的 **if**。代码示例：

```
if(Boolean-expression)
     "statement"
```

或

```
if(Boolean-expression)
     "statement"
else
   "statement"
```

布尔表达式（Boolean-expression）必须生成 **boolean** 类型的结果，执行语句 statement 既可以是以分号 ; 结尾的一条简单语句，也可以是包含在大括号 {} 内的的复合语句 —— 封闭在大括号内的一组简单语句。凡本书中提及"statement"一词，皆表示类似的执行语句。

下面是一个有关 **if-else** 语句的例子。test() 方法可以告知你两个数值之间的大小关系。代码示例：

```
// control/IfElse.java
public class IfElse {
  static int result = 0;
  static void test(int testval, int target) {
    if(testval > target)
      result = +1;
    else if(testval < target) // [1]
      result = -1;
    else
      result = 0; // Match
  }

  public static void main(String[] args) {
    test(10, 5);
    System.out.println(result);
    test(5, 10);
    System.out.println(result);
    test(5, 5);
    System.out.println(result);
  }
}
```

输出结果：

```
1
-1
0
```

注解：`else if` 并非新关键字，它仅是 `else` 后紧跟的一条新 `if` 语句。

Java 和 C/C++ 同属“自由格式”的编程语言，但通常我们会在 Java 控制流程语句中采用首部缩进的规范，以便代码更具可读性。

6.3 迭代语句

while，**do-while** 和 **for** 用来控制循环语句（有时也称迭代语句）。只有控制循环的布尔表达式计算结果为 `false`，循环语句才会停止。

6.3.1 while

while 循环的形式是：

```
while(Boolean-expression)
  statement
```

执行语句会在每一次循环前，判断布尔表达式返回值是否为 `true`。下例可产生随机数，直到满足特定条件。代码示例：

```
// control/WhileTest.java
// 演示 while 循环
public class WhileTest {
  static boolean condition() {
    boolean result = Math.random() < 0.99;
    System.out.print(result + ", ");
    return result;
  }
  public static void main(String[] args) {
    while(condition())
      System.out.println("Inside 'while'");
    System.out.println("Exited 'while'");
  }
}
```

输出结果：

```
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
...________...________...________...________...
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
```

```
false, Exited 'while'
```

condition() 方法使用到了 **Math** 库的**静态**方法 random()。该方法的作用是产生 0 和 1 之间 (包括 0，但不包括 1) 的一个 **double** 值。

result 的值是通过比较运算符 < 产生的 **boolean** 类型的结果。当控制台输出 **boolean** 型值时，会自动将其转换为对应的文字形式 **true** 或 **false**。此处 **while** 条件表达式代表："仅在 condition() 返回 **false** 时停止循环"。

6.3.2 do-while

do-while 的格式如下：

```
do
    statement
while(Boolean-expression);
```

while 和 **do-while** 之间的唯一区别是：即使条件表达式返回结果为 **false**，**do-while** 语句也至少会执行一次。在 **while** 循环体中，如布尔表达式首次返回的结果就为 **false**，那么循环体内的语句不会被执行。实际应用中，**while** 形式比 **do-while** 更为常用。

6.3.3 for

for 循环可能是最常用的迭代形式。该循环在第一次迭代之前执行初始化。随后，它会执行布尔表达式，并在每次迭代结束时，进行某种形式的步进。**for** 循环的形式是：

```
for(initialization; Boolean-expression; step)
  statement
```

初始化 (initialization) 表达式、布尔表达式 (Boolean-expression)，或者步进 (step) 运算，都可以为空。每次迭代之前都会判断布尔表达式的结果是否成立。一旦计算结果为 **false**，则跳出 **for** 循环体并继续执行后面代码。每次循环结束时，都会执行一次步进。

for 循环通常用于"计数"任务。代码示例：

```
// control/ListCharacters.java

public class ListCharacters {
  public static void main(String[] args) {
    for(char c = 0; c < 128; c++)
      if(Character.isLowerCase(c))
        System.out.println("value: " + (int)c +
          " character: " + c);
  }
}
```

输出结果（前 10 行）：

```
value: 97 character: a
value: 98 character: b
value: 99 character: c
value: 100 character: d
value: 101 character: e
value: 102 character: f
value: 103 character: g
value: 104 character: h
value: 105 character: i
value: 106 character: j
  ...
```

注意：变量 **c** 是在 **for** 循环执行时才被定义的，并不是在主方法的开头。**c** 的作用域范围仅在 **for** 循环体内。

传统的面向过程语言如 C 需要先在代码块（block）前定义好所有变量才能够使用。这样编译器才能在创建块时，为这些变量分配内存空间。在 Java 和 C++ 中，我们可以在整个块使用变量声明，并且可以在需要时才定义变量。这种自然的编码风格使我们的代码更容易被人理解 [1]。

上例使用了 **java.lang.Character** 包装类，该类不仅包含了基本类型 `char` 的值，还封装了一些有用的方法。例如这里就用到了静态方法 `isLowerCase()` 来判断字符是否为小写。

逗号操作符 在 Java 中逗号运算符（这里并非指我们平常用于分隔定义和方法参数的逗号分隔符）仅有一种用法：在 **for** 循环的初始化和步进控制中定义多个变量。我们可以使用逗号分隔多个语句，并按顺序计算这些语句。**注意**：要求定义的变量类型相同。代码示例：

```java
// control/CommaOperator.java

public class CommaOperator {
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
      System.out.println("i = " + i + " j = " + j);
    }
  }
}
```

输出结果：

```
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
```

上例中 **int** 类型声明包含了 `i` 和 `j`。实际上，在初始化部分我们可以定义任意数量的同类型变量。

[1] 功能粘贴在一起的方法的确有点与众不同，但它仍不失为一个库。

注意：在 Java 中，仅允许 **for** 循环在控制表达式中定义变量。我们不能将此方法与其他的循环语句和选择语句中一起使用。同时，我们可以看到：无论在初始化还是在步进部分，语句都是顺序执行的。

6.4 for-in 语法

Java 5 引入了更为简洁的"增强版 **for** 循环"语法来操纵数组和集合。（更多细节，可参考 数组 和 集合 章节内容）。大部分文档也称其为 **for-each** 语法，但因为了不与 Java 8 新添的 `forEach()` 产生混淆，因此我称之为 **for-in** 循环。（Python 已有类似的先例，如：**for x in sequence**）。**注意**：你可能会在其他地方看到不同叫法。

for-in 无需你去创建 **int** 变量和步进来控制循环计数。下面我们来遍历获取 **float** 数组中的元素。代码示例：

```
// control/ForInFloat.java

import java.util.*;

public class ForInFloat {
  public static void main(String[] args) {
    Random rand = new Random(47);
    float[] f = new float[10];
    for(int i = 0; i < 10; i++)
      f[i] = rand.nextFloat();
    for(float x : f)
      System.out.println(x);
  }
}
```

输出结果：

```
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
```

上例中我们展示了传统 **for** 循环的用法。接下来再来看下 **for-in** 的用法。代码示例：

```
for(float x : f) {
```

这条语句定义了一个 **float** 类型的变量 `x`，继而将每一个 `f` 的元素赋值给它。

任何一个返回数组的方法都可以使用 **for-in** 循环语法来遍历元素。例如 **String** 类有一个方法

`toCharArray()`，返回值类型为 **char** 数组，我们可以很容易地在 **for-in** 循环中遍历它。代码示例：

```
// control/ForInString.java

public class ForInString {
  public static void main(String[] args) {
    for(char c: "An African Swallow".toCharArray())
      System.out.print(c + " ");
  }
}
```

输出结果：

```
A n   A f r i c a n   S w a l l o w
```

很快我们能在 集合 章节里学习到，**for-in** 循环适用于任何可迭代（*iterable*）的对象。

通常，**for** 循环语句都会在一个整型数值序列中步进。代码示例：

```
for(int i = 0; i < 100; i++)
```

正因如此，除非先创建一个 **int** 数组，否则我们无法使用 **for-in** 循环来操作。为简化测试过程，我已在 `onjava` 包中封装了 **Range** 类，利用其 `range()` 方法可自动生成恰当的数组。

在 封装（Implementation Hiding）这一章里我们介绍了静态导入（static import），无需了解细节就可以直接使用。有关静态导入的语法，可以在 **import** 语句中看到：

```
// control/ForInInt.java

import static onjava.Range.*;

public class ForInInt {
  public static void main(String[] args) {
    for(int i : range(10)) // 0..9
      System.out.print(i + " ");
    System.out.println();
    for(int i : range(5, 10)) // 5..9
      System.out.print(i + " ");
    System.out.println();
    for(int i : range(5, 20, 3)) // 5..20 step 3
      System.out.print(i + " ");
    System.out.println();
    for(int i : range(20, 5, -3)) // Count down
      System.out.print(i + " ");
    System.out.println();
  }
}
```

输出结果：

```
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
```

```
5 8 11 14 17
20 17 14 11 8
```

range() 方法已被 重载（重载：同名方法，参数列表或类型不同）。上例中 range() 方法有多种重载形式：第一种产生从 0 至范围上限（不包含）的值；第二种产生参数一至参数二（不包含）范围内的整数值；第三种形式有一个步进值，因此它每次的增量为该值；第四种 range() 表明还可以递减。range() 无参方法是该生成器最简单的版本。有关内容会在本书稍后介绍。

range() 的使用提高了代码可读性，让 **for-in** 循环在本书中适应更多的代码示例场景。

请注意，System.out.print() 不会输出换行符，所以我们可以分段输出同一行。

for-in 语法可以节省我们编写代码的时间。更重要的是，它提高了代码可读性以及更好地描述代码意图（获取数组的每个元素）而不是详细说明这操作细节（创建索引，并用它来选择数组元素）本书推荐使用 *for-in* 语法。

6.5 return

在 Java 中有几个关键字代表无条件分支，这意味无需任何测试即可发生。这些关键字包括 **return**，**break**，**continue** 和跳转到带标签语句的方法，类似于其他语言中的 **goto**。

return 关键字有两方面的作用：1. 指定一个方法返回值 (在方法返回类型非 **void** 的情况下)；2. 退出当前方法，并返回作用 1 中值。我们可以利用 **return** 的这些特点来改写上例 IfElse.java 文件中的 test() 方法。代码示例：

```java
// control/TestWithReturn.java

public class TestWithReturn {
  static int test(int testval, int target) {
    if(testval > target)
      return +1;
    if(testval < target)
      return -1;
    return 0; // Match
  }

  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
}
```

输出结果：

```
1
-1
```

```
0
```

这里不需要 **else**，因为该方法执行到 **return** 就结束了。

如果在方法签名中定义了返回值类型为 **void**，那么在代码执行结束时会有一个隐式的 **return**。也就是说我们不用在总是在方法中显式地包含 **return** 语句。**注意**：如果你的方法声明的返回值类型为非 **void** 类型，那么则必须确保每个代码路径都返回一个值。

6.6 break 和 continue

在任何迭代语句的主体内，都可以使用 **break** 和 **continue** 来控制循环的流程。其中，**break** 表示跳出当前循环体。而 **continue** 表示停止本次循环，开始下一次循环。

下例向大家展示 **break** 和 **continue** 在 **for**、**while** 循环中的使用。代码示例：

```java
// control/BreakAndContinue.java
// Break 和 continue 关键字

import static onjava.Range.*;

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) { // [1]
      if(i == 74) break; // 跳出循环
      if(i % 9 != 0) continue; // 下一次循环
      System.out.print(i + " ");
    }
    System.out.println();
    // 使用 for-in 循环:
    for(int i : range(100)) { // [2]
      if(i == 74) break; // 跳出循环
      if(i % 9 != 0) continue; // 下一次循环
      System.out.print(i + " ");
    }
    System.out.println();
    int i = 0;
    // "无限循环":
    while(true) { // [3]
      i++;
      int j = i * 27;
      if(j == 1269) break; // 跳出循环
      if(i % 10 != 0) continue; // 循环顶部
      System.out.print(i + " ");
    }
  }
}
```

输出结果：

```
0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
```

[1] 在这个 **for** 循环中，`i` 的值永远不会达到 100，因为一旦 `i` 等于 74，**break** 语句就会中断循环。通常，只有在不知道中断条件何时满足时，才需要 **break**。因为 `i` 不能被 9 整除，**continue** 语句就会使循环从头开始。这使 **i** 递增)。如果能够整除，则将值显示出来。**[2]** 使用 **for-in** 语法，结果相同。**[3]** 无限 **while** 循环。循环内的 **break** 语句可中止循环。**注意**，**continue** 语句可将控制权移回循环的顶部，而不会执行 **continue** 之后的任何操作。因此，只有当 `i` 的值可被 10 整除时才会输出。在输出中，显示值 0，因为 `0%9` 产生 0。还有一种无限循环的形式：`for(;;)`。在编译器看来，它与 `while(true)` 无异，使用哪种完全取决于你的编程品味。

6.7 臭名昭著的 goto

goto 关键字 很早就在程序设计语言中出现。事实上，**goto** 起源于汇编（assembly language）语言中的程序控制："若条件 A 成立，则跳到这里；否则跳到那里"。如果你读过由编译器编译后的代码，你会发现在其程序控制中充斥了大量的跳转。较之汇编产生的代码直接运行在硬件 CPU 中，Java 也会产生自己的"汇编代码"（字节码），只不过它是运行在 Java 虚拟机里的（Java Virtual Machine）。

一个源码级别跳转的 **goto**，为何招致名誉扫地呢？若程序总是从一处跳转到另一处，还有什么办法能识别代码的控制流程呢？随着 *Edsger Dijkstra* 发表著名的 "Goto 有害" 论（*Goto considered harmful*）以后，**goto** 便从此失宠。甚至有人建议将它从关键字中剔除。

正如上述提及的经典情况，我们不应走向两个极端。问题不在 **goto**，而在于过度使用 **goto**。在极少数情况下，**goto** 实际上是控制流程的最佳方式。

尽管 **goto** 仍是 Java 的一个保留字，但其并未被正式启用。可以说，Java 中并不支持 **goto**。然而，在 **break** 和 **continue** 这两个关键字的身上，我们仍能看出一些 **goto** 的影子。它们并不属于一次跳转，而是中断循环语句的一种方法。之所以把它们纳入 **goto** 问题中一起讨论，是由于它们使用了相同的机制：标签。

"标签"是后面跟一个冒号的标识符。代码示例：

```
label1:
```

对 Java 来说，唯一用到标签的地方是在循环语句之前。进一步说，它实际需要紧靠在循环语句的前方 —— 在标签和循环之间置入任何语句都是不明智的。而在循环之前设置标签的唯一理由是：我们希望在其中嵌套另一个循环或者一个开关。这是由于 **break** 和 **continue** 关键字通常只中断当前循环，但若搭配标签一起使用，它们就会中断并跳转到标签所在的地方开始执行。代码示例：

```
label1:
outer-iteration {
```

A stitch in time saves nine.

```
  inner-iteration {
  // ...
  break; // [1]
  // ...
  continue; // [2]
  // ...
  continue label1; // [3]
  // ...
  break label1; // [4]
  }
}
```

[1] break 中断内部循环，并在外部循环结束。**[2] continue** 移回内部循环的起始处。但在条件 3 中，**continue label1** 却同时中断内部循环以及外部循环，并移至 **label1** 处。**[3]** 随后，它实际是继续循环，但却从外部循环开始。**[4] break label1** 也会中断所有循环，并回到 **label1** 处，但并不重新进入循环。也就是说，它实际是完全中止了两个循环。

下面是 **for** 循环的一个例子：

```java
// control/LabeledFor.java
// 搭配“标签 break”的 for 循环中使用 break 和 continue

public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // 此处不允许存在执行语句
    for(; true ;) { // 无限循环
      inner: // 此处不允许存在执行语句
      for(; i < 10; i++) {
        System.out.println("i = " + i);
        if(i == 2) {
          System.out.println("continue");
          continue;
        }
        if(i == 3) {
          System.out.println("break");
          i++; // 否则 i 永远无法获得自增
               // 获得自增
          break;
        }
        if(i == 7) {
          System.out.println("continue outer");
          i++;  // 否则 i 永远无法获得自增
                // 获得自增
          continue outer;
        }
        if(i == 8) {
          System.out.println("break outer");
          break outer;
        }
```

```
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            System.out.println("continue inner");
            continue inner;
          }
        }
      }
    }
    // 在此处无法 break 或 continue 标签
  }
}
```

输出结果:

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
```

注意 **break** 会中断 **for** 循环，而且在抵达 **for** 循环的末尾之前，递增表达式不会执行。由于 **break** 跳过了递增表达式，所以递增会在 `i==3` 的情况下直接执行。在 `i==7` 的情况下，`continue outer` 语句也会到达循环顶部，而且也会跳过递增，所以它也是直接递增的。

如果没有 **break outer** 语句，就没有办法在一个内部循环里找到出外部循环的路径。这是由于 **break** 本身只能中断最内层的循环（对于 **continue** 同样如此）。当然，若想在中断循环的同时退出方法，简单地用一个 **return** 即可。

下面这个例子向大家展示了带标签的 **break** 以及 **continue** 语句在 **while** 循环中的用法:

```
// control/LabeledWhile.java
// 带标签的 break 和 conitue 在 while 循环中的使用

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
```

```
    while(true) {
      System.out.println("Outer while loop");
      while(true) {
        i++;
        System.out.println("i = " + i);
        if(i == 1) {
          System.out.println("continue");
          continue;
        }
        if(i == 3) {
          System.out.println("continue outer");
          continue outer;
        }
        if(i == 5) {
          System.out.println("break");
          break;
        }
        if(i == 7) {
          System.out.println("break outer");
          break outer;
        }
      }
    }
  }
}
```

输出结果:

```
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
```

同样的规则亦适用于 **while**:

1. 简单的一个 **continue** 会退回最内层循环的开头（顶部），并继续执行。
2. 带有标签的 **continue** 会到达标签的位置，并重新进入紧接在那个标签后面的循环。
3. **break** 会中断当前循环，并移离当前标签的末尾。
4. 带标签的 **break** 会中断当前循环，并移离由那个标签指示的循环的末尾。

大家要记住的重点是：在 Java 里需要使用标签的唯一理由就是因为有循环嵌套存在，而且想从多

层嵌套中 **break** 或 **continue**。

break 和 **continue** 标签在编码中的使用频率相对较低 (此前的语言中很少使用或没有先例)，所以我们很少在代码里看到它们。

在 *Dijkstra* 的 **"Goto 有害"** 论文中，他最反对的就是标签，而非 **goto**。他观察到 BUG 的数量似乎随着程序中标签的数量而增加[2]。标签和 **goto** 使得程序难以分析。但是，Java 标签不会造成这方面的问题，因为它们的应用场景受到限制，无法用于以临时方式传输控制。由此也引出了一个有趣的情形：对语言能力的限制，反而使它这项特性更加有价值。

6.8 switch

switch 有时也被划归为一种选择语句。根据整数表达式的值，**switch** 语句可以从一系列代码中选出一段去执行。它的格式如下：

```
switch(integral-selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement;   break;
    case integral-value3 : statement;   break;
    case integral-value4 : statement;   break;
    case integral-value5 : statement;   break;
    // ...
    default: statement;
}
```

其中，**integral-selector** （整数选择因子）是一个能够产生整数值的表达式，**switch** 能够将这个表达式的结果与每个 **integral-value** （整数值）相比较。若发现相符的，就执行对应的语句（简单或复合语句，其中并不需要括号）。若没有发现相符的，就执行 **default** 语句。

在上面的定义中，大家会注意到每个 **case** 均以一个 **break** 结尾。这样可使执行流程跳转至 **switch** 主体的末尾。这是构建 **switch** 语句的一种传统方式，但 **break** 是可选的。若省略 **break**，会继续执行后面的 **case** 语句的代码，直到遇到一个 **break** 为止。通常我们不想出现这种情况，但对有经验的程序员来说，也许能够善加利用。注意最后的 **default** 语句没有 **break**，因为执行流程已到了 **break** 的跳转目的地。当然，如果考虑到编程风格方面的原因，完全可以在 **default** 语句的末尾放置一个 **break**，尽管它并没有任何实际的作用。

switch 语句是一种实现多路选择的干净利落的一种方式（比如从一系列执行路径中挑选一个）。但它要求使用一个选择因子，并且必须是 **int** 或 **char** 那样的整数值。例如，假若将一个字串或者浮点数作为选择因子使用，那么它们在 switch 语句里是不会工作的。对于非整数类型（Java 7 以上版本中的 String 型除外），则必须使用一系列 **if** 语句。在下一章的结尾 中，我们将会了解到**枚举类型**被用来搭配 **switch** 工作，并优雅地解决了这种限制。

[2]例如，这个电子书是利用 Pandoc 制作出来的，它是用纯函数式语言 Haskell 编写的一个程序。

下面这个例子可随机生成字母，并判断它们是元音还是辅音字母：

```
// control/VowelsAndConsonants.java

// switch 执行语句的演示
import java.util.*;

public class VowelsAndConsonants {
  public static void main(String[] args) {
    Random rand = new Random(47);
    for(int i = 0; i < 100; i++) {
      int c = rand.nextInt(26) + 'a';
      System.out.print((char)c + ", " + c + ": ");
      switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': System.out.println("vowel");
                  break;
        case 'y':
        case 'w': System.out.println("Sometimes vowel");
                  break;
        default:  System.out.println("consonant");
      }
    }
  }
}
```

输出结果：

```
y, 121: Sometimes vowel
n, 110: consonant
z, 122: consonant
b, 98: consonant
r, 114: consonant
n, 110: consonant
y, 121: Sometimes vowel
g, 103: consonant
c, 99: consonant
f, 102: consonant
o, 111: vowel
w, 119: Sometimes vowel
z, 122: consonant
  ...
```

由于 `Random.nextInt(26)` 会产生 0 到 25 之间的一个值，所以在其上加上一个偏移量 `a`，即可产生小写字母。在 **case** 语句中，使用单引号引起的字符也会产生用于比较的整数值。

请注意 **case** 语句能够堆叠在一起，为一段代码形成多重匹配，即只要符合多种条件中的一种，就执行那段特别的代码。这时也应该注意将 **break** 语句置于特定 **case** 的末尾，否则控制流程会继续往下

执行，处理后面的 **case**。在下面的语句中：

```
int c = rand.nextInt(26) + 'a';
```

此处 `Random.nextInt()` 将产生 0~25 之间的一个随机 **int** 值，它将被加到 `a` 上。这表示 `a` 将自动被转换为 **int** 以执行加法。为了把 `c` 当作字符打印，必须将其转型为 **char**；否则，将会输出整数。

6.9 switch 字符串

Java 7 增加了在字符串上 **switch** 的用法。下例展示了从一组 **String** 中选择可能值的传统方法，以及新式方法：

```
// control/StringSwitch.java

public class StringSwitch {
  public static void main(String[] args) {
    String color = "red";
    // 老的方式：使用 if-then 判断
    if("red".equals(color)) {
      System.out.println("RED");
    } else if("green".equals(color)) {
      System.out.println("GREEN");
    } else if("blue".equals(color)) {
      System.out.println("BLUE");
    } else if("yellow".equals(color)) {
      System.out.println("YELLOW");
    } else {
      System.out.println("Unknown");
    }
    // 新的方法：字符串搭配 switch
    switch(color) {
      case "red":
        System.out.println("RED");
        break;
      case "green":
        System.out.println("GREEN");
        break;
      case "blue":
        System.out.println("BLUE");
        break;
      case "yellow":
        System.out.println("YELLOW");
        break;
      default:
        System.out.println("Unknown");
        break;
    }
  }
```

```
}
```

输出结果：

```
RED
RED
```

一旦理解了 **switch**，你会明白这其实就是一个逻辑扩展的语法糖。新的编码方式能使得结果更清晰，更易于理解和维护。

作为 **switch** 字符串的第二个例子，我们重新访问 `Math.random()`。它是否产生从 0 到 1 的值，包括还是不包括值 1 呢？在数学术语中，它属于 (0,1)、[0,1)、(0,1] 、[0,1] 中的哪种呢？（方括号表示“包括”，而括号表示“不包括”）

下面是一个可能提供答案的测试程序。所有命令行参数都作为 **String** 对象传递，因此我们可以 **switch** 参数来决定要做什么。那么问题来了：如果用户不提供参数，索引到 `args` 的数组就会导致程序失败。解决这个问题，我们需要预先检查数组的长度，若长度为 0，则使用**空字符串** `""` 替代；否则，选择 `args` 数组中的第一个元素：

```java
// control/RandomBounds.java

// Math.random() 会产生 0.0 和 1.0 吗？
// {java RandomBounds lower}
import onjava.*;

public class RandomBounds {
  public static void main(String[] args) {
    new TimedAbort(3);
    switch(args.length == 0 ? "" : args[0]) {
      case "lower":
        while(Math.random() != 0.0)
          ; // 保持重试
        System.out.println("Produced 0.0!");
        break;
      case "upper":
        while(Math.random() != 1.0)
          ; // 保持重试
        System.out.println("Produced 1.0!");
        break;
      default:
        System.out.println("Usage:");
        System.out.println("\tRandomBounds lower");
        System.out.println("\tRandomBounds upper");
        System.exit(1);
    }
  }
}
```

要运行该程序，请键入以下任一命令：

```
java RandomBounds lower
// 或者
java RandomBounds upper
```

使用 `onjava` 包中的 **TimedAbort** 类可使程序在三秒后中止。从结果来看，似乎 `Math.random()` 产生的随机值里不包含 0.0 或 1.0。这就是该测试容易混淆的地方：若要考虑 0 至 1 之间所有不同 **double** 数值的可能性，那么这个测试的耗费的时间可能超出一个人的寿命了。这里我们直接给出正确的结果：`Math.random()` 的结果集范围包含 0.0 ，不包含 1.0。在数学术语中，可用 [0,1）来表示。由此可知，我们必须小心分析实验并了解它们的局限性。

6.10 本章小结

本章总结了我们对大多数编程语言中出现的基本特性的探索：计算，运算符优先级，类型转换，选择和迭代。现在让我们准备好，开始步入面向对象和函数式编程的世界吧。下一章的内容涵盖了 Java 编程中的重要问题：对象的初始化和清理。紧接着，还会介绍封装（implementation hiding）的核心概念。

A joke never gains an enemy but loses a friend.

7 初始化和清理

“不安全”的编程是造成编程代价昂贵的罪魁祸首之一。有两个安全性问题：初始化和清理。C 语言中很多的 bug 都是因为程序员忘记初始化导致的。尤其是很多类库的使用者不知道如何初始化类库组件，甚至他们必须得去初始化。清理则是另一个特殊的问题，因为当你使用一个元素做完事后就不会去关心这个元素，所以你很容易忘记清理它。这样就造成了元素使用的资源滞留不会被回收，直到程序消耗完所有的资源（特别是内存）。

C++ 引入了构造器的概念，这是一个特殊的方法，每创建一个对象，这个方法就会被自动调用。Java 采用了构造器的概念，另外还使用了垃圾收集器（Garbage Collector, GC）去自动回收不再被使用的对象所占的资源。这一章将讨论初始化和清理的问题，以及在 Java 中对它们的支持。

7.1 利用构造器保证初始化

你可能想为每个类创建一个 `initialize()` 方法，该方法名暗示着在使用类之前需要先调用它。不幸的是，用户必须得记得去调用它。在 Java 中，类的设计者通过构造器保证每个对象的初始化。如果一个类有构造器，那么 Java 会在用户使用对象之前（即对象刚创建完成）自动调用对象的构造器方法，从而保证初始化。下个挑战是如何命名构造器方法。存在两个问题：第一个是任何命名都可能与类中其他已有元素的命名冲突；第二个是编译器必须始终知道构造器方法名称，从而调用它。C++ 的解决方法看起来是最简单且最符合逻辑的，所以 Java 中使用了同样的方式：构造器名称与类名相同。在初始化过程中自动调用构造器方法是有意义的。

以下示例是包含了一个构造器的类：

```java
// housekeeping/SimpleConstructor.java
// Demonstration of a simple constructor

class Rock {
    Rock() { // 这是一个构造器
        System.out.print("Rock ");
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
```

```
            new Rock();
        }
    }
}
```

输出:

```
Rock Rock Rock Rock Rock Rock Rock Rock Rock Rock
```

现在，当创建一个对象时：`new Rock()`，内存被分配，构造器被调用。构造器保证了对象在你使用它之前进行了正确的初始化。

有一点需要注意，构造器方法名与类名相同，不需要符合首字母小写的编程风格。在 C++ 中，无参构造器被称为默认构造器，这个术语在 Java 出现之前使用了很多年。但是，出于一些原因，Java 设计者们决定使用无参构造器这个名称，我（作者）认为这种叫法笨拙而且没有必要，所以我打算继续使用默认构造器。Java 8 引入了 **default** 关键字修饰方法，所以算了，我还是用无参构造器的叫法吧。

跟其他方法一样，构造器方法也可以传入参数来定义如何创建一个对象。之前的例子稍作修改，使得构造器接收一个参数:

```
// housekeeping/SimpleConstructor2.java
// Constructors can have arguments

class Rock2 {
    Rock2(int i) {
        System.out.print("Rock " + i + " ");
    }
}

public class SimpleConstructor2 {
    public static void main(String[] args) {
        for (int i = 0; i < 8; i++) {
            new Rock2(i);
        }
    }
}
```

输出:

```
Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
```

如果类 **Tree** 有一个构造方法，只接收一个参数用来表示树的高度，那么你可以像下面这样创建一棵树:

```
Tree t = new Tree(12); // 12-foot 树
```

如果 **Tree(int)** 是唯一的构造器，那么编译器就不允许你以其他任何方式创建 **Tree** 类型的对象。

构造器消除了一类重要的问题，使得代码更易读。例如，在上面的代码块中，你看不到对

initialize() 方法的显式调用，而从概念上来看，initialize() 方法应该与对象的创建分离。在 Java 中，对象的创建与初始化是统一的概念，二者不可分割。

构造器没有返回值，它是一种特殊的方法。但它和返回类型为 **void** 的普通方法不同，普通方法可以返回空值，你还能选择让它返回别的类型；而构造器没有返回值，却同时也没有给你选择的余地（**new** 表达式虽然返回了刚创建的对象的引用，但构造器本身却没有返回任何值）。如果它有返回值，并且你也可以自己选择让它返回什么，那么编译器就还得知道接下来该怎么处理那个返回值（这个返回值没有接收者）。

7.2 方法重载

任何编程语言中都具备的一项重要特性就是命名。当你创建一个对象时，就会给此对象分配的内存空间命名。方法是行为的命名。你通过名字指代所有的对象，属性和方法。良好命名的系统易于理解和修改。就好比写散文——目的是与读者沟通。

将人类语言细微的差别映射到编程语言中会产生一个问题。通常，相同的词可以表达多种不同的含义——它们被“重载”了。特别是当含义的差别很小时，这会更加有用。你会说“清洗衬衫”、“清洗车”和“清洗狗”。而如果硬要这么说就会显得很愚蠢：“以洗衬衫的方式洗衬衫”、“以洗车的方式洗车”和“以洗狗的方式洗狗”，因为听众根本不需要区分行为的动作。大多数人类语言都具有“冗余”性，所以即使漏掉几个词，你也能明白含义。你不需要对每个概念都使用不同的词汇——可以从上下文推断出含义。

大多数编程语言（尤其是 C 语言）要求为每个方法（在这些语言中经常称为函数）提供一个独一无二的标识符。所以，你不能有一个 print() 函数既能打印整型，也能打印浮点型——每个函数名都必须不同。

在 Java (C++) 中，还有一个因素也促使了必须使用方法重载：构造器。因为构造器方法名肯定是与类名相同，所以一个类中只会有一个构造器名。那么你怎么通过不同的方式创建一个对象呢？例如，你想创建一个类，这个类的初始化方式有两种：一种是标准化方式，另一种是从文件中读取信息的方式。你需要两个构造器：无参构造器和有一个 **String** 类型参数的构造器，该参数传入文件名。两个构造器具有相同的名字——与类名相同。因此，方法重载是必要的，它允许方法具有相同的方法名但接收的参数不同。尽管方法重载对于构造器是重要的，但是也可以对任何方法很方便地进行重载。

下例展示了如何重载构造器和方法：

```
// housekeeping/Overloading.java
// Both constructor and ordinary method overloading

class Tree {
    int height;
    Tree() {
        System.out.println("Planting a seedling");
        height = 0;
```

习惯是后天养成的。

```
    }
    Tree(int initialHeight) {
        height = initialHeight;
        System.out.println("Creating new Tree that is " + height + "
            feet tall");
    }
    void info() {
        System.out.println("Tree is " + height + " feet tall");
    }
    void info(String s) {
        System.out.println(s + ": Tree is " + height + " feet tall");
    }
}
public class Overloading {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        new Tree();
    }
}
```

输出:

```
Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall
Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Creating new Tree that is 4 feet tall
Tree is 4 feet tall
overloaded method: Tree is 4 feet tall
Planting a seedling
```

一个 **Tree** 对象既可以是一颗树苗，使用无参构造器创建，也可以是一颗在温室中已长大的树，已经有一定高度，这时候，就需要使用有参构造器创建。

你也许想以多种方式调用 `info()` 方法。比如，如果你想打印额外的消息，就可以使用 `info(String)` 方法。如果你无话可说，就可以使用 `info()` 方法。用两个命名定义完全相同的概念看起来很奇怪，而使用方法重载，你就可以使用一个命名来定义一个概念。

7.2.1 区分重载方法

如果两个方法命名相同，Java 是怎么知道你调用的是哪个呢？有一条简单的规则：每个被重载的方法必须有独一无二的参数列表。你稍微思考下，就会很明了了，除了通过参数列表的不同来区分两个相同命名的方法，其他也没什么方式了。你甚至可以根据参数列表中的参数顺序来区分不同的方法，尽管这会造成代码难以维护。例如：

```java
// housekeeping/OverloadingOrder.java
// Overloading based on the order of the arguments

public class OverloadingOrder {
    static void f(String s, int i) {
        System.out.println("String: " + s + ", int: " + i);
    }

    static void f(int i, String s) {
        System.out.println("int: " + i + ", String: " + s);
    }

    public static void main(String[] args) {
        f("String first", 1);
        f(99, "Int first");
    }
}
```

输出：

```
String: String first, int: 1
int: 99, String: Int first
```

两个 `f()` 方法具有相同的参数，但是参数顺序不同，根据这个就可以区分它们。

7.2.2 重载与基本类型

基本类型可以自动从较小的类型转型为较大的类型。当这与重载结合时，这会令人有点困惑，下面是一个这样的例子：

```java
// housekeeping/PrimitiveOverloading.java
// Promotion of primitives and overloading

public class PrimitiveOverloading {
    void f1(char x) {
        System.out.print("f1(char)");
    }
    void f1(byte x) {
        System.out.print("f1(byte)");
    }
    void f1(short x) {
```

```
        System.out.print("f1(short)");
    }
    void f1(int x) {
        System.out.print("f1(int)");
    }
    void f1(long x) {
        System.out.print("f1(long)");
    }
    void f1(float x) {
        System.out.print("f1(float)");
    }
    void f1(double x) {
        System.out.print("f1(double)");
    }
    void f2(byte x) {
        System.out.print("f2(byte)");
    }
    void f2(short x) {
        System.out.print("f2(short)");
    }
    void f2(int x) {
        System.out.print("f2(int)");
    }
    void f2(long x) {
        System.out.print("f2(long)");
    }
    void f2(float x) {
        System.out.print("f2(float)");
    }
    void f2(double x) {
        System.out.print("f2(double)");
    }
    void f3(short x) {
        System.out.print("f3(short)");
    }
    void f3(int x) {
        System.out.print("f3(int)");
    }
    void f3(long x) {
        System.out.print("f3(long)");
    }
    void f3(float x) {
        System.out.print("f3(float)");
    }
    void f3(double x) {
        System.out.print("f3(double)");
    }
    void f4(int x) {
        System.out.print("f4(int)");
    }
    void f4(long x) {
```

```
        System.out.print("f4(long)");
    }
    void f4(float x) {
        System.out.print("f4(float)");
    }
    void f4(double x) {
        System.out.print("f4(double)");
    }
    void f5(long x) {
        System.out.print("f5(long)");
    }
    void f5(float x) {
        System.out.print("f5(float)");
    }
    void f5(double x) {
        System.out.print("f5(double)");
    }
    void f6(float x) {
        System.out.print("f6(float)");
    }
    void f6(double x) {
        System.out.print("f6(double)");
    }
    void f7(double x) {
        System.out.print("f7(double)");
    }
    void testConstVal() {
        System.out.print("5: ");
        f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
        System.out.println();
    }
    void testChar() {
        char x = 'x';
        System.out.print("char: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        System.out.println();
    }
    void testByte() {
        byte x = 0;
        System.out.print("byte: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        System.out.println();
    }
    void testShort() {
        short x = 0;
        System.out.print("short: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        System.out.println();
    }
    void testInt() {
        int x = 0;
```

```
        System.out.print("int: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        System.out.println();
    }
    void testLong() {
        long x = 0;
        System.out.print("long: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        System.out.println();
    }
    void testFloat() {
        float x = 0;
        System.out.print("float: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        System.out.println();
    }
    void testDouble() {
        double x = 0;
        System.out.print("double: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        System.out.println();
    }

    public static void main(String[] args) {
        PrimitiveOverloading p = new PrimitiveOverloading();
        p.testConstVal();
        p.testChar();
        p.testByte();
        p.testShort();
        p.testInt();
        p.testLong();
        p.testFloat();
        p.testDouble();
    }
}
```

输出:

```
5: f1(int)f2(int)f3(int)f4(int)f5(long)f6(float)f7(double)
char: f1(char)f2(int)f3(int)f4(int)f5(long)f6(float)f7(double)
byte: f1(byte)f2(byte)f3(short)f4(int)f5(long)f6(float)f7(double)
short: f1(short)f2(short)f3(short)f4(int)f5(long)f6(float)f7(double)
int: f1(int)f2(int)f3(int)f4(int)f5(long)f6(float)f7(double)
long: f1(long)f2(long)f3(long)f4(long)f5(long)f6(float)f7(double)
float: f1(float)f2(float)f3(float)f4(float)f5(float)f6(float)f7(double)
double: f1(double)f2(double)f3(double)f4(double)f5(double)f6(double)f7(
    double)
```

如果传入的参数类型大于方法期望接收的参数类型，你必须首先做下转换，如果你不做的话，编译器就会报错。

7.2.3 返回值的重载

经常会有人困惑，“为什么只能通过类名和参数列表，不能通过方法的返回值区分方法呢?”。例如以下两个方法，它们有相同的命名和参数，但是很容易区分：

```
void f(){}
int f() {return 1;}
```

有些情况下，编译器很容易就可以从上下文准确推断出该调用哪个方法，如 `int x = f()`。

但是，你可以调用一个方法且忽略返回值。这叫做调用一个函数的副作用，因为你不在乎返回值，只是想利用方法做些事。所以如果你直接调用 `f()`，Java 编译器就不知道你想调用哪个方法，阅读者也不明所以。因为这个原因，所以你不能根据返回值类型区分重载的方法。为了支持新特性，Java 8 在一些具体情形下提高了猜测的准确度，但是通常来说并不起作用。

7.3 无参构造器

如前文所说，一个无参构造器就是不接收参数的构造器，用来创建一个“默认的对象”。如果你创建一个类，类中没有构造器，那么编译器就会自动为你创建一个无参构造器。例如：

```
// housekeeping/DefaultConstructor.java
class Bird {}
public class DefaultConstructor {
    public static void main(String[] args) {
        Bird bird = new Bird(); // 默认的
    }
}
```

表达式 `new Bird()` 创建了一个新对象，调用了无参构造器，尽管在 **Bird** 类中并没有显式的定义无参构造器。试想如果没有构造器，我们如何创建一个对象呢。但是，一旦你显式地定义了构造器（无论有参还是无参），编译器就不会自动为你创建无参构造器。如下：

```
// housekeeping/NoSynthesis.java
class Bird2 {
    Bird2(int i) {}
    Bird2(double d) {}
}
public class NoSynthesis {
    public static void main(String[] args) {
        //- Bird2 b = new Bird2(); // No default
        Bird2 b2 = new Bird2(1);
        Bird2 b3 = new Bird2(1.0);
    }
}
```

如果你调用了 `new Bird2()`，编译器会提示找不到匹配的构造器。当类中没有构造器时，编译器

会说“你一定需要构造器，那么让我为你创建一个吧”。但是如果类中有构造器，编译器会说“你已经写了构造器了，所以肯定知道你在做什么，如果你没有创建默认构造器，说明你本来就不需要”。

7.4 this 关键字

对于两个相同类型的对象 **a** 和 **b**，你可能在想如何调用这两个对象的 `peel()` 方法：

```
// housekeeping/BananaPeel.java

class Banana {
    void peel(int i) {
        /*...*/
    }
}
public class BananaPeel {
    public static void main(String[] args) {
        Banana a = new Banana(), b = new Banana();
        a.peel(1);
        b.peel(2);
    }
}
```

如果只有一个方法 `peel()` ，那么怎么知道调用的是对象 **a** 的 `peel()`方法还是对象 **b** 的 `peel()` 方法呢？编译器做了一些底层工作，所以你可以像这样编写代码。`peel()` 方法中第一个参数隐密地传入了一个指向操作对象的

引用。因此，上述例子中的方法调用像下面这样：

```
Banana.peel(a, 1)
Banana.peel(b, 1)
```

这是在内部实现的，你不可以直接这么编写代码，编译器不会接受，但能说明到底发生了什么。假设现在在方法内部，你想获得对当前对象的引用。但是，对象引用是被秘密地传达给编译器——并不在参数列表中。方便的是，有一个关键字：**this** 。**this** 关键字只能在非静态方法内部使用。当你调用一个对象的方法时，**this** 生成了一个对象引用。你可以像对待其他引用一样对待这个引用。如果你在一个类的方法里调用其他该类中的方法，不要使用 **this**，直接调用即可，**this** 自动地应用于其他方法上了。因此你可以像这样：

```
// housekeeping/Apricot.java

public class Apricot {
    void pick() {
        /* ... */
    }

    void pit() {
```

```
        pick();
        /* ... */
    }
}
```

在 pit() 方法中，你可以使用 **this**.pick()，但是没有必要。编译器自动为你做了这些。**this** 关键字只用在一些必须显式使用当前对象引用的特殊场合。例如，用在 **return** 语句中返回对当前对象的引用。

```
// housekeeping/Leaf.java
// Simple use of the "this" keyword

public class Leaf {

    int i = 0;

    Leaf increment() {
        i++;
        return this;
    }

    void print() {
        System.out.println("i = " + i);
    }

    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
    }
}
```

输出:

```
i = 3
```

因为 increment() 通过 **this** 关键字返回当前对象的引用，因此在相同的对象上可以轻易地执行多次操作。

this 关键字在向其他方法传递当前对象时也很有用:

```
// housekeeping/PassingThis.java

class Person {
    public void eat(Apple apple) {
        Apple peeled = apple.getPeeled();
        System.out.println("Yummy");
    }
}

public class Peeler {
```

```
    static Apple peel(Apple apple) {
        // ... remove peel
        return apple; // Peeled
    }
}

public class Apple {
    Apple getPeeled() {
        return Peeler.peel(this);
    }
}

public class PassingThis {
    public static void main(String[] args) {
        new Person().eat(new Apple());
    }
}
```

输出:

```
Yummy
```

Apple 因为某些原因（比如说工具类中的方法在多个类中重复出现，你不想代码重复），必须调用一个外部工具方法 `Peeler.peel()` 做一些行为。必须使用 **this** 才能将自身传递给外部方法。

7.4.1 在构造器中调用构造器

当你在一个类中写了多个构造器，有时你想在一个构造器中调用另一个构造器来避免代码重复。你通过 **this** 关键字实现这样的调用。

通常当你说 **this**，意味着“这个对象”或“当前对象”，它本身生成对当前对象的引用。在一个构造器中，当你给 **this** 一个参数列表时，它是另一层意思。它通过最直接的方式显式地调用匹配参数列表的构造器:

```
// housekeeping/Flower.java
// Calling constructors with "this"

public class Flower {
    int petalCount = 0;
    String s = "initial value";

    Flower(int petals) {
        petalCount = petals;
        System.out.println("Constructor w/ int arg only, petalCount = "
            + petalCount);
    }

    Flower(String ss) {
```

```
        System.out.println("Constructor w/ string arg only, s = " + ss);
        s = ss;
    }

    Flower(String s, int petals) {
        this(petals);
        //- this(s); // Can't call two!
        this.s = s; // Another use of "this"
        System.out.println("String & int args");
    }

    Flower() {
        this("hi", 47);
        System.out.println("no-arg constructor");
    }

    void printPetalCount() {
        //- this(11); // Not inside constructor!
        System.out.println("petalCount = " + petalCount + " s = " + s);
    }

    public static void main(String[] args) {
        Flower x = new Flower();
        x.printPetalCount();
    }
}
```

输出:

```
Constructor w/ int arg only, petalCount = 47
String & int args
no-arg constructor
petalCount = 47 s = hi
```

从构造器 `Flower(String s, int petals)` 可以看出, 其中只能通过 **this** 调用一次构造器。另外, 必须首先调用构造器, 否则编译器会报错。这个例子同样展示了 **this** 的另一个用法。参数列表中的变量名 **s** 和成员变量名 **s** 相同, 会引起混淆。你可以通过 `this.s` 表明你指的是成员变量 **s**, 从而避免重复。你经常会在 Java 代码中看到这种用法, 同时本书中也会多次出现这种写法。在 `printPetalCount()` 方法中, 编译器不允许你在一个构造器之外的方法里调用构造器。

7.4.2 static 的含义

记住了 **this** 关键字的内容, 你会对 **static** 修饰的方法有更加深入的理解: **static** 方法中不会存在 **this**。你不能在静态方法中调用非静态方法（反之可以）。静态方法是为类而创建的, 不需要任何对象。事实上, 这就是静态方法的主要目的, 静态方法看起来就像全局方法一样, 但是 Java 中不允许全局方

法，一个类中的静态方法可以被其他的静态方法和静态属性访问。一些人认为静态方法不是面向对象的，因为它们的确具有全局方法的语义。使用静态方法，因为不存在 **this**，所以你没有向一个对象发送消息。的确，如果你发现代码中出现了大量的 **static** 方法，就该重新考虑自己的设计了。然而，**static** 的概念很实用，许多时候都要用到它。至于它是否真的“面向对象”，就留给理论家去讨论吧。

7.5 垃圾回收器

程序员都了解初始化的重要性，但通常会忽略清理的重要性。毕竟，谁会去清理一个 **int** 呢？但是使用完一个对象就不管它并非总是安全的。Java 中有垃圾回收器回收无用对象占用的内存。但现在考虑一种特殊情况：你创建的对象不是通过 **new** 来分配内存的，而垃圾回收器只知道如何释放用 **new** 创建的对象的内存，所以它不知道如何回收不是 **new** 分配的内存。为了处理这种情况，Java 允许在类中定义一个名为 `finalize()` 的方法。

它的工作原理“假定”是这样的：当垃圾回收器准备回收对象的内存时，首先会调用其 `finalize()` 方法，并在下一轮的垃圾回收动作发生时，才会真正回收对象占用的内存。所以如果你打算使用 `finalize()`，就能在垃圾回收时做一些重要的清理工作。`finalize()` 是一个潜在的编程陷阱，因为一些程序员（尤其是 C++ 程序员）会一开始把它误认为是 C++ 中的析构函数（C++ 在销毁对象时会调用这个函数）。所以有必要明确区分一下：在 C++ 中，对象总是被销毁的（在一个 bug-free 的程序中），而在 Java 中，对象并非总是被垃圾回收，或者换句话说：

1. 对象可能不被垃圾回收。
2. 垃圾回收不等同于析构。

这意味着在你不再需要某个对象之前，如果必须执行某些动作，你得自己去做。Java 没有析构器或类似的概念，所以你必须得自己创建一个普通的方法完成这项清理工作。例如，对象在创建的过程中会将自己绘制到屏幕上。如果不是明确地从屏幕上将其擦除，它可能永远得不到清理。如果在 `finalize()` 方法中加入某种擦除功能，那么当垃圾回收发生时，`finalize()` 方法被调用（不保证一定会发生），图像就会被擦除，要是“垃圾回收”没有发生，图像则仍会保留下来。

也许你会发现，只要程序没有濒临内存用完的那一刻，对象占用的空间就总也得不到释放。如果程序执行结束，而垃圾回收器一直没有释放你创建的任何对象的内存，则当程序退出时，那些资源会全部交还给操作系统。这个策略是恰当的，因为垃圾回收本身也有开销，要是不使用它，那就不用支付这部分开销了。

7.5.1 `finalize()` 的用途

如果你不能将 `finalize()` 作为通用的清理方法，那么这个方法有什么用呢？

这引入了要记住的第 3 点：

3. 垃圾回收只与内存有关。

也就是说，使用垃圾回收的唯一原因就是为了回收程序不再使用的内存。所以对于与垃圾回收有关的任何行为来说（尤其是 `finalize()` 方法），它们也必须同内存及其回收有关。

但这是否意味着如果对象中包括其他对象，`finalize()` 方法就应该明确释放那些对象呢？不是，无论对象是如何创建的，垃圾回收器都会负责释放对象所占用的所有内存。这就将对 `finalize()` 的需求限制到一种特殊情况，即通过某种创建对象方式之外的方式为对象分配了存储空间。不过，你可能会想，Java 中万物皆对象，这种情况怎么可能发生？

看起来之所以有 `finalize()` 方法，是因为在分配内存时可能采用了类似 C 语言中的做法，而非 Java 中的通常做法。这种情况主要发生在使用“本地方法”的情况下，本地方法是一种用 Java 语言调用非 Java 语言代码的形式（关于本地方法的讨论，见本书电子版第 2 版的附录 B）。本地方法目前只支持 C 和 C++，但是它们可以调用其他语言写的代码，所以实际上可以调用任何代码。在非 Java 代码中，也许会调用 C 的 `malloc()` 函数系列来分配存储空间，而且除非调用 `free()` 函数，不然存储空间永远得不到释放，造成内存泄露。但是，`free()` 是 C 和 C++ 中的函数，所以你需要在 `finalize()` 方法里用本地方法调用它。

读到这里，你可能明白了不会过多使用 `finalize()` 方法。对，它确实不是进行普通的清理工作的合适场所。那么，普通的清理工作在哪里执行呢？

7.5.2 你必须实施清理

要清理一个对象，用户必须在需要清理的时候调用执行清理动作的方法。这听上去相当直接，但却与 C++ 中的“析构函数”的概念稍有抵触。在 C++ 中，所有对象都会被销毁，或者说应该被销毁。如果在 C++ 中创建了一个局部对象（在栈上创建，在 Java 中不行），此时的销毁动作发生在以“右花括号”为边界的、此对象作用域的末尾处。如果对象是用 **new** 创建的（类似于 Java 中），那么当程序员调用 C++ 的 **delete** 操作符时（Java 中不存在），就会调用相应的析构函数。如果程序员忘记调用 **delete**，那么永远不会调用析构函数，这样就会导致内存泄露，对象的其他部分也不会得到清理。这种 bug 很难跟踪，也是让 C++ 程序员转向 Java 的一个主要因素。相反，在 Java 中，没有用于释放对象的 **delete**，因为垃圾回收器会帮助你释放存储空间。甚至可以肤浅地认为，正是由于垃圾回收的存在，使得 Java 没有析构函数。然而，随着学习的深入，你会明白垃圾回收器的存在并不能完全替代析构函数（而且绝对不能直接调用 `finalize()`，所以这也不是一种解决方案）。如果希望进行除释放存储空间之外的清理工作，还是得明确调用某个恰当的 Java 方法：这就等同于使用析构函数了，只是没有它方便。

记住，无论是“垃圾回收”还是“终结”，都不保证一定会发生。如果 Java 虚拟机（JVM）并未面临内存耗尽的情形，它可能不会浪费时间执行垃圾回收以恢复内存。

7.5.3 终结条件

通常，不能指望 `finalize()`，你必须创建其他的“清理”方法，并明确地调用它们。所以看起来，`finalize()` 只对大部分程序员很难用到的一些晦涩内存清理里有用了。但是，`finalize()` 还有一个

有趣的用法，它不依赖于每次都要对 `finalize()` 进行调用，这就是对象终结条件的验证。

当对某个对象不感兴趣时——也就是它将被清理了，这个对象应该处于某种状态，这种状态下它占用的内存可以被安全地释放掉。例如，如果对象代表了一个打开的文件，在对象被垃圾回收之前程序员应该关闭这个文件。只要对象中存在没有被适当清理的部分，程序就存在很隐晦的 bug。`finalize()` 可以用来最终发现这个情况，尽管它并不总是被调用。如果某次 `finalize()` 的动作使得 bug 被发现，那么就可以据此找出问题所在——这才是人们真正关心的。以下是个简单的例子，示范了 `finalize()` 的可能使用方式：

```java
// housekeeping/TerminationCondition.java
// Using finalize() to detect a object that
// hasn't been properly cleaned up

import onjava.*;

class Book {
    boolean checkedOut = false;

    Book(boolean checkOut) {
        checkedOut = checkOut;
    }

    void checkIn() {
        checkedOut = false;
    }

    @Override
    protected void finalize() throws Throwable {
        if (checkedOut) {
            System.out.println("Error: checked out");
        }
        // Normally, you'll also do this:
        // super.finalize(); // Call the base-class version
    }
}

public class TerminationCondition {

    public static void main(String[] args) {
        Book novel = new Book(true);
        // Proper cleanup:
        novel.checkIn();
        // Drop the reference, forget to clean up:
        new Book(true);
        // Force garbage collection & finalization:
        System.gc();
        new Nap(1); // One second delay
    }

```

```
}
```

输出:

```
Error: checked out
```

本例的终结条件是：所有的 **Book** 对象在被垃圾回收之前必须被登记。但在 `main()` 方法中，有一本书没有登记。要是没有 `finalize()` 方法来验证终结条件，将会很难发现这个 bug。

你可能注意到使用了 `@Override`。`@` 意味着这是一个注解，注解是关于代码的额外信息。在这里，该注解告诉编译器这不是偶然地重定义在每个对象中都存在的 `finalize()` 方法——程序员知道自己在做什么。编译器确保你没有拼错方法名，而且确保那个方法存在于基类中。注解也是对读者的提醒，`@Override` 在 Java 5 引入，在 Java 7 中改善，本书通篇会出现。

注意，`System.gc()` 用于强制进行终结动作。但是即使不这么做，只要重复地执行程序（假设程序将分配大量的存储空间而导致垃圾回收动作的执行），最终也能找出错误的 **Book** 对象。

你应该总是假设基类版本的 `finalize()` 也要做一些重要的事情，使用 **super** 调用它，就像在 `Book.finalize()` 中看到的那样。本例中，它被注释掉了，因为它需要进行异常处理，而我们到现在还没有涉及到。

7.5.4 垃圾回收器如何工作

如果你以前用过的语言，在堆上分配对象的代价十分高昂，你可能自然会觉得 Java 中所有对象（基本类型除外）在堆上分配的方式也十分高昂。然而，垃圾回收器能很明显地提高对象的创建速度。这听起来很奇怪——存储空间的释放影响了存储空间的分配，但这确实是某些 Java 虚拟机的工作方式。这也意味着，Java 从堆空间分配的速度可以和其他语言在栈上分配空间的速度相媲美。

例如，你可以把 C++ 里的堆想象成一个院子，里面每个对象都负责管理自己的地盘。一段时间后，对象可能被销毁，但地盘必须复用。在某些 Java 虚拟机中，堆的实现截然不同：它更像一个传送带，每分配一个新对象，它就向前移动一格。这意味着对象存储空间的分配速度特别快。Java 的“堆指针”只是简单地移动到尚未分配的区域，所以它的效率与 C++ 在栈上分配空间的效率相当。当然实际过程中，在簿记工作方面还有少量额外开销，但是这部分开销比不上查找可用空间开销大。

你可能意识到了，Java 中的堆并非完全像传送带那样工作。要是那样的话，势必会导致频繁的内存页面调度——将其移进移出硬盘，因此会显得需要拥有比实际需要更多的内存。页面调度会显著影响性能。最终，在创建了足够多的对象后，内存资源被耗尽。其中的秘密在于垃圾回收器的介入。当它工作时，一边回收内存，一边使堆中的对象紧凑排列，这样“堆指针”就可以很容易地移动到更靠近传送带的开始处，也就尽量避免了页面错误。垃圾回收器通过重新排列对象，实现了一种高速的、有无限空间可分配的堆模型。

要想理解 Java 中的垃圾回收，先了解其他系统中的垃圾回收机制将会很有帮助。一种简单但速度很慢的垃圾回收机制叫做*引用计数*。每个对象中含有一个引用计数器，每当有引用指向该对象时，引用

计数加 1。当引用离开作用域或被置为 **null** 时，引用计数减 1。因此，管理引用计数是一个开销不大但是在程序的整个生命周期频繁发生的负担。垃圾回收器会遍历含有全部对象的列表，当发现某个对象的引用计数为 0 时，就释放其占用的空间（但是，引用计数模式经常会在计数为 0 时立即释放对象）。这个机制存在一个缺点：如果对象之间存在循环引用，那么它们的引用计数都不为 0，就会出现应该被回收但无法被回收的情况。对垃圾回收器而言，定位这样的循环引用所需的工作量极大。引用计数常用来说明垃圾回收的工作方式，但似乎从未被应用于任何一种 Java 虚拟机实现中。

在更快的策略中，垃圾回收器并非基于引用计数。它们依据的是：对于任意“活”的对象，一定能最终追溯到其存活在栈或静态存储区中的引用。这个引用链条可能会穿过数个对象层次，由此，如果从栈或静态存储区出发，遍历所有的引用，你将会发现所有“活”的对象。对于发现的每个引用，必须追踪它所引用的对象，然后是该对象包含的所有引用，如此反复进行，直到访问完“根源于栈或静态存储区的引用”所形成的整个网络。你所访问过的对象一定是“活”的。注意，这解决了对象间循环引用的问题，这些对象不会被发现，因此也就被自动回收了。

在这种方式下，Java 虚拟机采用了一种*自适应*的垃圾回收技术。至于如何处理找到的存活对象，取决于不同的 Java 虚拟机实现。其中有一种做法叫做停止-复制（stop-and-copy）。顾名思义，这需要先暂停程序的运行（不属于后台回收模式），然后将所有存活的对象从当前堆复制到另一个堆，没有复制的就是需要被垃圾回收的。另外，当对象被复制到新堆时，它们是一个挨着一个紧凑排列，然后就可以按照前面描述的那样简单、直接地分配新空间了。

当对象从一处复制到另一处，所有指向它的引用都必须修正。位于栈或静态存储区的引用可以直接被修正，但可能还有其他指向这些对象的引用，它们在遍历的过程中才能被找到（可以想象成一个表格，将旧地址映射到新地址）。

这种所谓的“复制回收器”效率低下主要因为两个原因。其一：得有两个堆，然后在这两个分离的堆之间来回折腾，得维护比实际需要多一倍的空间。某些 Java 虚拟机对此问题的处理方式是，按需从堆中分配几块较大的内存，复制动作发生在这些大块内存之间。

其二在于复制本身。一旦程序进入稳定状态之后，可能只会产生少量垃圾，甚至没有垃圾。尽管如此，复制回收器仍然会将所有内存从一处复制到另一处，这很浪费。为了避免这种状况，一些 Java 虚拟机会进行检查：要是没有新垃圾产生，就会转换到另一种模式（即“自适应”）。这种模式称为标记-清扫（mark-and-sweep），Sun 公司早期版本的 Java 虚拟机一直使用这种技术。对一般用途而言，“标记-清扫”方式速度相当慢，但是当你知道程序只会产生少量垃圾甚至不产生垃圾时，它的速度就很快了。

“标记-清扫”所依据的思路仍然是从栈和静态存储区出发，遍历所有的引用，找出所有存活的对象。但是，每当找到一个存活对象，就给对象设一个标记，并不回收它。只有当标记过程完成后，清理动作才开始。在清理过程中，没有标记的对象将被释放，不会发生任何复制动作。“标记-清扫”后剩下的堆空间是不连续的，垃圾回收器要是希望得到连续空间的话，就需要重新整理剩下的对象。

“停止-复制”指的是这种垃圾回收动作不是在后台进行的；相反，垃圾回收动作发生的同时，程序将会暂停。在 Oracle 公司的文档中会发现，许多参考文献将垃圾回收视为低优先级的后台进程，但是早期版本的 Java 虚拟机并不是这么实现垃圾回收器的。当可用内存较低时，垃圾回收器会暂停程序。同

样，“标记-清扫”工作也必须在程序暂停的情况下才能进行。

如前文所述，这里讨论的 Java 虚拟机中，内存分配以较大的“块”为单位。如果对象较大，它会占用单独的块。严格来说，“停止-复制”要求在释放旧对象之前，必须先将所有存活对象从旧堆复制到新堆，这导致了大量的内存复制行为。有了块，垃圾回收器就可以把对象复制到废弃的块。每个块都有年代数来记录自己是否存活。通常，如果块在某处被引用，其年代数加 1，垃圾回收器会对上次回收动作之后新分配的块进行整理。这对处理大量短命的临时对象很有帮助。垃圾回收器会定期进行完整的清理动作——大型对象仍然不会复制（只是年代数会增加），含有小型对象的那些块则被复制并整理。Java 虚拟机会监视，如果所有对象都很稳定，垃圾回收的效率降低的话，就切换到“标记-清扫”方式。同样，Java 虚拟机会跟踪“标记-清扫”的效果，如果堆空间出现很多碎片，就会切换回“停止-复制”方式。这就是“自适应”的由来，你可以给它个啰嗦的称呼：“自适应的、分代的、停止-复制、标记-清扫”式的垃圾回收器。

Java 虚拟机中有许多附加技术用来提升速度。尤其是与加载器操作有关的，被称为“即时”（Just-In-Time, JIT）编译器的技术。这种技术可以把程序全部或部分翻译成本地机器码，所以不需要 JVM 来进行翻译，因此运行得更快。当需要装载某个类（通常是创建该类的第一个对象）时，编译器会先找到其 **.class** 文件，然后将该类的字节码装入内存。你可以让即时编译器编译所有代码，但这种做法有两个缺点：一是这种加载动作贯穿整个程序生命周期内，累加起来需要花更多时间；二是会增加可执行代码的长度（字节码要比即时编译器展开后的本地机器码小很多），这会导致页面调度，从而一定降低程序速度。另一种做法称为*惰性评估*，意味着即时编译器只有在必要的时候才编译代码。这样，从未被执行的代码也许就压根不会被 JIT 编译。新版 JDK 中的 Java HotSpot 技术就采用了类似的做法，代码每被执行一次就优化一些，所以执行的次数越多，它的速度就越快。

7.6 成员初始化

Java 尽量保证所有变量在使用前都能得到恰当的初始化。对于方法的局部变量，这种保证会以编译时错误的方式呈现，所以如果写成：

```
void f() {
    int i;
    i++;
}
```

你会得到一条错误信息，告诉你 **i** 可能尚未初始化。编译器可以为 **i** 赋一个默认值，但是未初始化的局部变量更有可能是程序员的疏忽，所以采用默认值反而会掩盖这种失误。强制程序员提供一个初始值，往往能帮助找出程序里的 bug。

要是类的成员变量是基本类型，情况就会变得有些不同。正如在“万物皆对象”一章中所看到的，类的每个基本类型数据成员保证都会有一个初始值。下面的程序可以验证这类情况，并显示它们的值：

```
// housekeeping/InitialValues.java
```

```java
// Shows default initial values

public class InitialValues {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    InitialValues reference;

    void printInitialValues() {
        System.out.println("Data type Initial value");
        System.out.println("boolean " + t);
        System.out.println("char[" + c + "]");
        System.out.println("byte " + b);
        System.out.println("short " + s);
        System.out.println("int " + i);
        System.out.println("long " + l);
        System.out.println("float " + f);
        System.out.println("double " + d);
        System.out.println("reference " + reference);
    }

    public static void main(String[] args) {
        new InitialValues().printInitialValues();
    }
}
```

输出:

```
Data type Initial value
boolean false
char[NUL]
byte 0
short 0
int 0
long 0
float 0.0
double 0.0
reference null
```

可见尽管数据成员的初值没有给出，但它们确实有初值（char 值为 0，所以显示为空白）。所以这样至少不会出现“未初始化变量”的风险了。

在类里定义一个对象引用时，如果不将其初始化，那么引用就会被赋值为 **null**。

7.6.1 指定初始化

怎么给一个变量赋初值呢？一种很直接的方法是在定义类成员变量的地方为其赋值。以下代码修改了 InitialValues 类成员变量的定义，直接提供了初值：

```
// housekeeping/InitialValues2.java
// Providing explicit initial values

public class InitialValues2 {
    boolean bool = true;
    char ch = 'x';
    byte b = 47;
    short s = 0xff;
    int i = 999;
    long lng = 1;
    float f = 3.14f;
    double d = 3.14159;
}
```

你也可以用同样的方式初始化非基本类型的对象。如果 **Depth** 是一个类，那么可以像下面这样创建一个对象并初始化它：

```
// housekeeping/Measurement.java

class Depth {}

public class Measurement {
    Depth d = new Depth();
    // ...
}
```

如果没有为 **d** 赋予初值就尝试使用它，就会出现运行时错误，告诉你产生了一个异常（详细见“异常”章节）。

你也可以通过调用某个方法来提供初值：

```
// housekeeping/MethodInit.java

public class MethodInit {
    int i = f();

    int f() {
        return 11;
    }

}
```

这个方法可以带有参数，但这些参数不能是未初始化的类成员变量。因此，可以这么写：

```
// housekeeping/MethodInit2.java
```

```java

public class MethodInit2 {
    int i = f();
    int j = g(i);

    int f() {
        return 11;
    }

    int g(int n) {
        return n * 10;
    }
}
```

但是你不能这么写：

```java
// housekeeping/MethodInit3.java

public class MethodInit3 {
    //- int j = g(i); // Illegal forward reference
    int i = f();

    int f() {
        return 11;
    }

    int g(int n) {
        return n * 10;
    }
}
```

显然，上述程序的正确性取决于初始化的顺序，而与其编译方式无关。所以，编译器恰当地对“向前引用”发出了警告。

这种初始化方式简单直观，但有个限制：类 **InitialValues** 的每个对象都有相同的初值，有时这的确是我们需要的，但有时却需要更大的灵活性。

7.7 构造器初始化

可以用构造器进行初始化，这种方式给了你更大的灵活性，因为你可以在运行时调用方法进行初始化。但是，这无法阻止自动初始化的进行，他会在构造器被调用之前发生。因此，如果使用如下代码：

```java
// housekeeping/Counter.java

public class Counter {
    int i;

    Counter() {
```

```
        i = 7;
    }
    // ...
}
```

i 首先会被初始化为 **0**，然后变为 **7**。对于所有的基本类型和引用，包括在定义时已明确指定初值的变量，这种情况都是成立的。因此，编译器不会强制你一定要在构造器的某个地方或在使用它们之前初始化元素——初始化早已得到了保证。，

7.7.1 初始化的顺序

在类中变量定义的顺序决定了它们初始化的顺序。即使变量定义散布在方法定义之间，它们仍会在任何方法（包括构造器）被调用之前得到初始化。例如：

```java
// housekeeping/OrderOfInitialization.java
// Demonstrates initialization order
// When the constructor is called to create a
// Window object, you'll see a message:

class Window {
    Window(int marker) {
        System.out.println("Window(" + marker + ")");
    }
}

class House {
    Window w1 = new Window(1); // Before constructor

    House() {
        // Show that we're in the constructor:
        System.out.println("House()");
        w3 = new Window(33); // Reinitialize w3
    }

    Window w2 = new Window(2); // After constructor

    void f() {
        System.out.println("f()");
    }

    Window w3 = new Window(3); // At end
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        House h = new House();
        h.f(); // Shows that construction is done
    }
```

```
}
```

输出:

```
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
```

在 **House** 类中，故意把几个 **Window** 对象的定义散布在各处，以证明它们全都会在调用构造器或其他方法之前得到初始化。此外，**w3** 在构造器中被再次赋值。

由输出可见，引用 **w3** 被初始化了两次：一次在调用构造器前，一次在构造器调用期间（第一次引用的对象将被丢弃，并作为垃圾回收）。这乍一看可能觉得效率不高，但保证了正确的初始化。试想，如果定义了一个重载构造器，在其中没有初始化 **w3**，同时在定义 **w3** 时没有赋予初值，那会产生怎样的后果呢?

7.7.2 静态数据的初始化

无论创建多少个对象，静态数据都只占用一份存储区域。**static** 关键字不能应用于局部变量，所以只能作用于属性（字段、域）。如果一个字段是静态的基本类型，你没有初始化它，那么它就会获得基本类型的标准初值。如果它是对象引用，那么它的默认初值就是 **null**。

如果在定义时进行初始化，那么静态变量看起来就跟非静态变量一样。

下面例子显示了静态存储区是何时初始化的:

```java
// housekeeping/StaticInitialization.java
// Specifying initial values in a class definition

class Bowl {
    Bowl(int marker) {
        System.out.println("Bowl(" + marker + ")");
    }

    void f1(int marker) {
        System.out.println("f1(" + marker + ")");
    }
}

class Table {
    static Bowl bowl1 = new Bowl(1);

    Table() {
        System.out.println("Table()");
        bowl2.f1(1);
    }
```

```java

    void f2(int marker) {
        System.out.println("f2(" + marker + ")");
    }

    static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
    Bowl bowl3 = new Bowl(3);
    static Bowl bowl4 = new Bowl(4);

    Cupboard() {
        System.out.println("Cupboard()");
        bowl4.f1(2);
    }

    void f3(int marker) {
        System.out.println("f3(" + marker + ")");
    }

    static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
    public static void main(String[] args) {
        System.out.println("main creating new Cupboard()");
        new Cupboard();
        System.out.println("main creating new Cupboard()");
        new Cupboard();
        table.f2(1);
        cupboard.f3(1);
    }

    static Table table = new Table();
    static Cupboard cupboard = new Cupboard();
}
```

输出:

```
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f1(2)
main creating new Cupboard()
Bowl(3)
```

```
Cupboard()
f1(2)
main creating new Cupboard()
Bowl(3)
Cupboard()
f1(2)
f2(1)
f3(1)
```

Bowl 类展示类的创建，而 **Table** 和 **Cupboard** 在它们的类定义中包含 **Bowl** 类型的静态数据成员。注意，在静态数据成员定义之前，**Cupboard** 类中先定义了一个 **Bowl** 类型的非静态成员 **b3**。

由输出可见,静态初始化只有在必要时刻才会进行。如果不创建 **Table** 对象,也不引用 **Table.bowl1** 或 **Table.bowl2**，那么静态的 **Bowl** 类对象 **bowl1** 和 **bowl2** 永远不会被创建。只有在第一个 Table 对象被创建（或被访问）时，它们才会被初始化。此后，静态对象不会再次被初始化。

初始化的顺序先是静态对象（如果它们之前没有被初始化的话），然后是非静态对象，从输出中可以看出。要执行 `main()` 方法，必须加载 **StaticInitialization** 类，它的静态属性 **table** 和 **cupboard** 随后被初始化，这会导致它们对应的类也被加载，而由于它们都包含静态的 **Bowl** 对象，所以 **Bowl** 类也会被加载。因此，在这个特殊的程序中，所有的类都会在 `main()` 方法之前被加载。实际情况通常并非如此，因为在典型的程序中，不会像本例中所示的那样，将所有事物通过 **static** 联系起来。

概括一下创建对象的过程，假设有个名为 **Dog** 的类：

1. 即使没有显式地使用 **static** 关键字，构造器实际上也是静态方法。所以，当首次创建 **Dog** 类型的对象或是首次访问 **Dog** 类的静态方法或属性时，Java 解释器必须在类路径中查找，以定位 **Dog.class**。
2. 当加载完 **Dog.class** 后（后面会学到，这将创建一个 **Class** 对象），有关静态初始化的所有动作都会执行。因此，静态初始化只会在首次加载 **Class** 对象时初始化一次。
3. 当用 `new Dog()` 创建对象时，首先会在堆上为 **Dog** 对象分配足够的存储空间。
4. 分配的存储空间首先会被清零，即会将 **Dog** 对象中的所有基本类型数据设置为默认值（数字会被置为 0，布尔型和字符型也相同），引用被置为 **null**。
5. 执行所有出现在字段定义处的初始化动作。
6. 执行构造器。你将会在“复用”这一章看到，这可能会牵涉到很多动作，尤其当涉及继承的时候。

7.7.3 显式的静态初始化

你可以将一组静态初始化动作放在类里面一个特殊的“静态子句”（有时叫做静态块）中。像下面这样：

```
// housekeeping/Spoon.java

public class Spoon {
    static int i;
```

```

    static {
        i = 47;
    }
}
```

这看起来像个方法，但实际上它只是一段跟在 **static** 关键字后面的代码块。与其他静态初始化动作一样，这段代码仅执行一次：当首次创建这个类的对象或首次访问这个类的静态成员（甚至不需要创建该类的对象）时。例如：

```
// housekeeping/ExplicitStatic.java
// Explicit static initialization with "static" clause

class Cup {
    Cup(int marker) {
        System.out.println("Cup(" + marker + ")");
    }

    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Cups {
    static Cup cup1;
    static Cup cup2;

    static {
        cup1 = new Cup(1);
        cup2 = new Cup(2);
    }

    Cups() {
        System.out.println("Cups()");
    }
}

public class ExplicitStatic {
    public static void main(String[] args) {
        System.out.println("Inside main()");
        Cups.cup1.f(99); // [1]
    }

    // static Cups cups1 = new Cups(); // [2]
    // static Cups cups2 = new Cups(); // [2]
}
```

输出：

```
Inside main
```

```
Cup(1)
Cup(2)
f(99)
```

无论是通过标为 [1] 的行访问静态的 **cup1** 对象，还是把标为 [1] 的行去掉，让它去运行标为 [2] 的那行代码（去掉 [2] 的注释），**Cups** 的静态初始化动作都会执行。如果同时注释 [1] 和 [2] 处，那么 **Cups** 的静态初始化就不会进行。此外，把标为 [2] 处的注释都去掉还是只去掉一个，静态初始化只会执行一次。

7.7.4 非静态实例初始化

Java 提供了被称为*实例初始化*的类似语法，用来初始化每个对象的非静态变量，例如：

```java
// housekeeping/Mugs.java
// Instance initialization

class Mug {
    Mug(int marker) {
        System.out.println("Mug(" + marker + ")");
    }
}

public class Mugs {
    Mug mug1;
    Mug mug2;
    { // [1]
        mug1 = new Mug(1);
        mug2 = new Mug(2);
        System.out.println("mug1 & mug2 initialized");
    }

    Mugs() {
        System.out.println("Mugs()");
    }

    Mugs(int i) {
        System.out.println("Mugs(int)");
    }

    public static void main(String[] args) {
        System.out.println("Inside main()");
        new Mugs();
        System.out.println("new Mugs() completed");
        new Mugs(1);
        System.out.println("new Mugs(1) completed");
    }
}
```

输出:

```
Inside main
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs()
new Mugs() completed
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs(int)
new Mugs(1) completed
```

看起来它很像静态代码块，只不过少了 **static** 关键字。这种语法对于支持“匿名内部类”（参见“内部类”一章）的初始化是必须的，但是你也可以使用它保证某些操作一定会发生，而不管哪个构造器被调用。从输出看出，实例初始化子句是在两个构造器之前执行的。

7.8 数组初始化

数组是相同类型的、用一个标识符名称封装到一起的一个对象序列或基本类型数据序列。数组是通过方括号下标操作符 [] 来定义和使用的。要定义一个数组引用，只需要在类型名加上方括号:

```
int[] a1;
```

方括号也可放在标识符的后面，两者的含义是一样的:

```
int a1[];
```

这种格式符合 C 和 C++ 程序员的习惯。不过前一种格式或许更合理，毕竟它表明类型是“一个 **int** 型数组”。本书中采用这种格式。

编译器不允许指定数组的大小。这又把我们带回有关“引用”的问题上。你所拥有的只是对数组的一个引用（你已经为该引用分配了足够的存储空间），但是还没有给数组对象本身分配任何空间。为了给数组创建相应的存储空间，必须写初始化表达式。对于数组，初始化动作可以出现在代码的任何地方，但是也可以使用一种特殊的初始化表达式，它必须在创建数组的地方出现。这种特殊的初始化是由一对花括号括起来的值组成。这种情况下，存储空间的分配（相当于使用 **new**）将由编译器负责。例如:

```
int[] a1 = {1, 2, 3, 4, 5};
```

那么为什么在还没有数组的时候定义一个数组引用呢?

```
int[] a2;
```

在 Java 中可以将一个数组赋值给另一个数组，所以可以这样:

```
a2 = a1;
```

其实真正做的只是复制了一个引用，就像下面演示的这样：

```
// housekeeping/ArraysOfPrimitives.java

public class ArraysOfPrimitives {
    public static void main(String[] args) {
        int[] a1 = {1, 2, 3, 4, 5};
        int[] a2;
        a2 = a1;
        for (int i = 0; i < a2.length; i++) {
            a2[i] += 1;
        }
        for (int i = 0; i < a1.length; i++) {
            System.out.println("a1[" + i + "] = " + a1[i]);
        }
    }
}
```

输出：

```
a1[0] = 2;
a1[1] = 3;
a1[2] = 4;
a1[3] = 5;
a1[4] = 6;
```

a1 初始化了，但是 **a2** 没有；这里，**a2** 在后面被赋给另一个数组。由于 **a1** 和 **a2** 是相同数组的别名，因此通过 **a2** 所做的修改在 **a1** 中也能看到。

所有的数组（无论是对象数组还是基本类型数组）都有一个固定成员 **length**，告诉你这个数组有多少个元素，你不能对其修改。与 C 和 C++ 类似，Java 数组计数也是从 0 开始的，所能使用的最大下标数是 **length - 1**。超过这个边界，C 和 C++ 会默认接受，允许你访问所有内存，许多声名狼藉的 bug 都是由此而生。但是 Java 在你访问超出这个边界时，会报运行时错误（异常），从而避免此类问题。

7.8.1 动态数组创建

如果在编写程序时，不确定数组中需要多少个元素，那么该怎么办呢？你可以直接使用 **new** 在数组中创建元素。下面例子中，尽管创建的是基本类型数组，**new** 仍然可以工作（不能用 **new** 创建单个的基本类型数组）：

```
// housekeeping/ArrayNew.java
// Creating arrays with new
import java.util.*;

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

Easier said than done.

```
        int[] a;
        Random rand = new Random(47);
        a = new int[rand.nextInt(20)];
        System.out.println("length of a = " + a.length);
        System.out.println(Arrays.toString(a));
    }
}
```

输出:

```
length of a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

数组的大小是通过 `Random.nextInt()` 随机确定的，这个方法会返回 0 到输入参数之间的一个值。由于随机性，很明显数组的创建确实是在运行时进行的。此外，程序输出表明，数组元素中的基本数据类型值会自动初始化为空值（对于数字和字符是 0；对于布尔型是 **false**）。`Arrays.toString()` 是 **java.util** 标准类库中的方法，会产生一维数组的可打印版本。

本例中，数组也可以在定义的同时进行初始化:

```
int[] a = new int[rand.nextInt(20)];
```

如果可能的话，应该尽量这么做。

如果你创建了一个非基本类型的数组，那么你创建的是一个引用数组。以整型的包装类型 **Integer** 为例，它是一个类而非基本类型:

```
// housekeeping/ArrayClassObj.java
// Creating an array of nonprimitive objects

import java.util.*;

public class ArrayClassObj {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] a = new Integer[rand.nextInt(20)];
        System.out.println("length of a = " + a.length);
        for (int i = 0; i < a.length; i++) {
            a[i] = rand.nextInt(500); // Autoboxing
        }
        System.out.println(Arrays.toString(a));
    }
}
```

输出:

```
length of a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51, 89, 309, 278,
    498, 361, 20]
```

这里，即使使用 new 创建数组之后:

```
Integer[] a = new Integer[rand.nextInt(20)];
```

它只是一个引用数组，直到通过创建新的 **Integer** 对象（通过自动装箱），并把对象赋值给引用，初始化才算结束：

```
a[i] = rand.nextInt(500);
```

如果忘记了创建对象，但试图使用数组中的空引用，就会在运行时产生异常。

也可以用花括号括起来的列表来初始化数组，有两种形式：

```
// housekeeping/ArrayInit.java
// Array initialization
import java.util.*;

public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
                1, 2,
                3, // Autoboxing
        };
        Integer[] b = new Integer[] {
                1, 2,
                3, // Autoboxing
        };
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.toString(b));

    }
}
```

输出：

```
[1, 2, 3]
[1, 2, 3]
```

在这两种形式中，初始化列表的最后一个逗号是可选的（这一特性使维护长列表变得更容易）。

尽管第一种形式很有用，但是它更加受限，因为它只能用于数组定义处。第二种和第三种形式可以用在任何地方，甚至用在方法的内部。例如，你创建了一个 **String** 数组，将其传递给另一个类的 `main()` 方法，如下：

```
// housekeeping/DynamicArray.java
// Array initialization

public class DynamicArray {
    public static void main(String[] args) {
        Other.main(new String[] {"fiddle", "de", "dum"});
    }
}
```

```

class Other {
    public static void main(String[] args) {
        for (String s: args) {
            System.out.print(s + " ");
        }
    }
}
```

输出:

```
fiddle de dum
```

`Other.main()` 的参数是在调用处创建的，因此你甚至可以在方法调用处提供可替换的参数。

7.8.2 可变参数列表

你可以以一种类似 C 语言中的可变参数列表（C 通常把它称为“varargs”）来创建和调用方法。这可以应用在参数个数或类型未知的场合。由于所有的类都最后继承于 **Object** 类（随着本书的进展，你会对此有更深的认识），所以你可以创建一个以 Object 数组为参数的方法，并像下面这样调用:

```
// housekeeping/VarArgs.java
// Using array syntax to create variable argument lists

class A {}

public class VarArgs {
    static void printArray(Object[] args) {
        for (Object obj: args) {
            System.out.print(obj + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        printArray(new Object[] {47, (float) 3.14, 11.11});
        printArray(new Object[] {"one", "two", "three"});
        printArray(new Object[] {new A(), new A(), new A()});
    }
}
```

输出:

```
47 3.14 11.11
one two three
A@15db9742 A@6d06d69c A@7852e922
```

`printArray()` 的参数是 **Object** 数组，使用 for-in 语法遍历和打印数组的每一项。标准 Java 库能输出有意义的内容，但这里创建的是类的对象，打印出的内容是类名，后面跟着一个 **@** 符号

以及多个十六进制数字。因而，默认行为（如果没有定义 toString() 方法的话，后面会讲这个方法）就是打印类名和对象的地址。

你可能看到像上面这样编写的 Java 5 之前的代码，它们可以产生可变的参数列表。在 Java 5 中，这种期盼已久的特性终于添加了进来，就像在 printArray() 中看到的那样：

```
// housekeeping/NewVarArgs.java
// Using array syntax to create variable argument lists

public class NewVarArgs {
    static void printArray(Object... args) {
        for (Object obj: args) {
            System.out.print(obj + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Can take individual elements:
        printArray(47, (float) 3.14, 11.11);
        printArray(47, 3.14F, 11.11);
        printArray("one", "two", "three");
        printArray(new A(), new A(), new A());
        // Or an array:
        printArray((Object[]) new Integer[] {1, 2, 3, 4});
        printArray(); // Empty list is OK
    }
}
```

输出：

```
47 3.14 11.11
47 3.14 11.11
one two three
A@15db9742 A@6d06d69c A@7852e922
1 2 3 4
```

有了可变参数，你就再也不用显式地编写数组语法了，当你指定参数时，编译器实际上会为你填充数组。你获取的仍然是一个数组，这就是为什么 printArray() 可以使用 for-in 迭代数组的原因。但是，这不仅仅只是从元素列表到数组的自动转换。注意程序的倒数第二行，一个 **Integer** 数组（通过自动装箱创建）被转型为一个 **Object** 数组（为了移除编译器的警告），并且传递给了 printArray()。显然，编译器会发现这是一个数组，不会执行转换。因此，如果你有一组事物，可以把它们当作列表传递，而如果你已经有了一个数组，该方法会把它们当作可变参数列表来接受。

程序的最后一行表明，可变参数的个数可以为 0。当具有可选的尾随参数时，这一特性会有帮助：

```
// housekeeping/OptionalTrailingArguments.java

public class OptionalTrailingArguments {
```

```
    static void f(int required, String... trailing) {
        System.out.print("required: " + required + " ");
        for (String s: trailing) {
            System.out.print(s + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        f(1, "one");
        f(2, "two", "three");
        f(0);
    }
}
```

输出:

```
required: 1 one
required: 2 two three
required: 0
```

这段程序展示了如何使用除了 **Object** 类之外类型的可变参数列表。这里，所有的可变参数都是 **String** 对象。可变参数列表中可以使用任何类型的参数，包括基本类型。下面例子展示了可变参数列表变为数组的情形，并且如果列表中没有任何元素，那么转变为大小为 0 的数组:

```
// housekeeping/VarargType.java

public class VarargType {
    static void f(Character... args) {
        System.out.print(args.getClass());
        System.out.println(" length " + args.length);
    }

    static void g(int... args) {
        System.out.print(args.getClass());
        System.out.println(" length " + args.length)
    }

    public static void main(String[] args) {
        f('a');
        f();
        g(1);
        g();
        System.out.println("int[]: "+ new int[0].getClass());
    }
}
```

输出:

```
class [Ljava.lang.Character; length 1
```

```
class [Ljava.lang.Character; length 0
class [I length 1
class [I length 0
int[]: class [I
```

getClass() 方法属于 Object 类，将在“类型信息”一章中全面介绍。它会产生对象的类，并在打印该类时，看到表示该类类型的编码字符串。前导的 **[** 代表这是一个后面紧随的类型的数组，**I** 表示基本类型 **int**；为了进行双重检查，我在最后一行创建了一个 **int** 数组，打印了其类型。这样也验证了使用可变参数列表不依赖于自动装箱，而使用的是基本类型。

然而，可变参数列表与自动装箱可以和谐共处，如下：

```java
// housekeeping/AutoboxingVarargs.java

public class AutoboxingVarargs {
    public static void f(Integer... args) {
        for (Integer i: args) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        f(1, 2);
        f(4, 5, 6, 7, 8, 9);
        f(10, 11, 12);

    }
}
```

输出：

```
1 2
4 5 6 7 8 9
10 11 12
```

注意吗，你可以在单个参数列表中将类型混合在一起，自动装箱机制会有选择地把 **int** 类型的参数提升为 **Integer**。

可变参数列表使得方法重载更加复杂了，尽管乍看之下似乎足够安全：

```java
// housekeeping/OverloadingVarargs.java

public class OverloadingVarargs {
    static void f(Character... args) {
        System.out.print("first");
        for (Character c: args) {
            System.out.print(" " + c);
        }
        System.out.println();
```

```
    }

    static void f(Integer... args) {
        System.out.print("second");
        for (Integer i: args) {
            System.out.print(" " + i);
        }
        System.out.println();
    }

    static void f(Long... args) {
        System.out.println("third");
    }

    public static void main(String[] args) {
        f('a', 'b', 'c');
        f(1);
        f(2, 1);
        f(0);
        f(0L);
        //- f(); // Won's compile -- ambiguous
    }
}
```

输出:

```
first a b c
second 1
second 2 1
second 0
third
```

在每种情况下，编译器都会使用自动装箱来匹配重载的方法，然后调用最明确匹配的方法。

但是如果调用不含参数的 `f()`，编译器就无法知道应该调用哪个方法了。尽管这个错误可以弄清楚，但是它可能会使客户端程序员感到意外。

你可能会通过在某个方法中增加一个非可变参数解决这个问题:

```
// housekeeping/OverloadingVarargs2.java
// {WillNotCompile}

public class OverloadingVarargs2 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }

    static void f(Character... args) {
        System.out.println("second");
    }

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

```
        f(1, 'a');
        f('a', 'b');
    }
}
```

{WillNotCompile} 注释把该文件排除在了本书的 Gradle 构建之外。如果你手动编译它，会得到下面的错误信息：

```
OverloadingVarargs2.java:14:error:reference to f is ambiguous f('a', 'b
    ');
\^
both method f(float, Character...) in OverloadingVarargs2 and method f(
    Character...) in OverloadingVarargs2 match 1 error
```

如果你给这两个方法都添加一个非可变参数，就可以解决问题了：

```
// housekeeping/OverloadingVarargs3

public class OverloadingVarargs3 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }

    static void f(char c, Character... args) {
        System.out.println("second");
    }

    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
}
```

输出：

```
first
second
```

你应该总是在重载方法的一个版本上使用可变参数列表，或者压根不用它。

7.9 枚举类型

Java 5 中添加了一个看似很小的特性 **enum** 关键字，它使得我们在需要群组并使用枚举类型集时，可以很方便地处理。以前，你需要创建一个整数常量集，但是这些值并不会将自身限制在这个常量集的范围内，因此使用它们更有风险，而且更难使用。枚举类型属于非常普遍的需求，C、C++ 和其他许多语言都已经拥有它了。在 Java 5 之前，Java 程序员必须了解许多细节并格外仔细地去达成 **enum** 的效果。现在 Java 也有了 **enum**，并且它的功能比 C/C++ 中的完备得多。下面是个简单的例子：

```java
// housekeeping/Spiciness.java

public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
}
```

这里创建了一个名为 **Spiciness** 的枚举类型，它有 5 个值。由于枚举类型的实例是常量，因此按照命名惯例，它们都用大写字母表示（如果名称中含有多个单词，使用下划线分隔）。

要使用 **enum**，需要创建一个该类型的引用，然后将其赋值给某个实例：

```java
// housekeeping/SimpleEnumUse.java

public class SimpleEnumUse {
    public static void main(String[] args) {
        Spiciness howHot = Spiciness.MEDIUM;
        System.out.println(howHot);
    }
}
```

输出：

```
MEDIUM
```

在你创建 **enum** 时，编译器会自动添加一些有用的特性。例如，它会创建 `toString()` 方法，以便你方便地显示某个 **enum** 实例的名称，这从上面例子中的输出可以看出。编译器还会创建 `ordinal()` 方法表示某个特定 **enum** 常量的声明顺序，`static values()` 方法按照 enum 常量的声明顺序，生成这些常量值构成的数组：

```java
// housekeeping/EnumOrder.java

public class EnumOrder {
    public static void main(String[] args) {
        for (Spiciness s: Spiciness.values()) {
            System.out.println(s + ", ordinal " + s.ordinal());
        }
    }
}
```

输出：

```
NOT, ordinal 0
MILD, ordinal 1
MEDIUM, ordinal 2
HOT, ordinal 3
FLAMING, ordinal 4
```

尽管 **enum** 看起来像是一种新的数据类型，但是这个关键字只是在生成 **enum** 的类时，产生了某些编译器行为，因此在很大程度上你可以将 **enum** 当作其他任何类。事实上，**enum** 确实是类，并且具

有自己的方法。

enum 有一个很实用的特性，就是在 **switch** 语句中使用：

```
// housekeeping/Burrito.java

public class Burrito {
    Spiciness degree;

    public Burrito(Spiciness degree) {
        this.degree = degree;
    }

    public void describe() {
        System.out.print("This burrito is ");
        switch(degree) {
            case NOT:
                System.out.println("not spicy at all.");
                break;
            case MILD:
            case MEDIUM:
                System.out.println("a little hot.");
                break;
            case HOT:
            case FLAMING:
            default:
                System.out.println("maybe too hot");
        }
    }

    public static void main(String[] args) {
        Burrito plain = new Burrito(Spiciness.NOT),
        greenChile = new Burrito(Spiciness.MEDIUM),
        jalapeno = new Burrito(Spiciness.HOT);
        plain.describe();
        greenChile.describe();
        jalapeno.describe();
    }
}
```

输出：

```
This burrito is not spicy at all.
This burrito is a little hot.
This burrito is maybe too hot.
```

由于 **switch** 是在有限的可能值集合中选择，因此它与 **enum** 是绝佳的组合。注意，enum 的名称是如何能够倍加清楚地表明程序的目的的。

通常，你可以将 **enum** 用作另一种创建数据类型的方式，然后使用所得到的类型。这正是关键所在，所以你不用过多地考虑它们。在 **enum** 被引入之前，你必须花费大量的精力去创建一个等同的枚举

类型，并是安全可用的。

这些介绍对于你理解和使用基本的 **enum** 已经足够了，我们会在“枚举”一章中进行更深入的探讨。

7.10 本章小结

构造器，这种看起来精巧的初始化机制，应该给了你很强的暗示：初始化在编程语言中的重要地位。C++ 的发明者 Bjarne Stroustrup 在设计 C++ 期间，在针对 C 语言的生产效率进行的最初调查中发现，错误的初始化会导致大量编程错误。这些错误很难被发现，同样，不合理的清理也会如此。因为构造器能保证进行正确的初始化和清理（没有正确的构造器调用，编译器就不允许创建对象），所以你就有了完全的控制和安全。

在 C++ 中，析构器很重要，因为用 **new** 创建的对象必须被明确地销毁。在 Java 中，垃圾回收器会自动地释放所有对象的内存，所以很多时候类似的清理方法就不太需要了（但是当要用到的时候，你得自己动手）。在不需要类似析构器行为的时候，Java 的垃圾回收器极大地简化了编程，并加强了内存管理上的安全性。一些垃圾回收器甚至能清理其他资源，如图形和文件句柄。然而，垃圾回收器确实增加了运行时开销，由于 Java 解释器从一开始就很慢，所以这种开销到底造成多大的影响很难看出来。随着时间的推移，Java 在性能方面提升了很多，但是速度问题仍然是它涉足某些特定编程领域的障碍。

由于要保证所有对象被创建，实际上构造器比这里讨论得更加复杂。特别是当通过*组合*或*继承*创建新类的时候，这种保证仍然成立，并且需要一些额外的语法来支持。在后面的章节中，你会学习组合，继承以及它们如何影响构造器。

Avoid arc-sine and arc-cosine functions - you can usually do better by applying a trig identity or computing a vector dot-product. — Jim Conyngham

8 封装

访问控制（或者隐藏实现）与“最初的实现不恰当”有关。

所有优秀的作者——包括这些编写软件的人——都知道一件好的作品都是经过反复打磨才变得优秀的。如果你把一段代码置于某个位置一段时间，过一会重新来看，你可能发现更好的实现方式。这是重构的原动力之一，重构就是重写可工作的代码，使之更加可读，易懂，因而更易维护。

但是，在修改和完善代码的愿望下，也存在巨大的压力。通常，客户端程序员希望你的代码在某些方面保持不变。所以你想修改代码，但他们希望代码保持不变。由此引出了面向对象设计中的一个基本问题：“如何区分变动的事物和不变的事物”。

这个问题对于类库而言尤其重要。类库的使用者必须依赖他们所使用的那部分类库，并且知道如果使用了类库的新版本，不需要改写代码。另一方面，类库的开发者必须有修改和改进类库的自由，并保证客户代码不会受这些改动影响。

这可以通过约定解决。例如，类库开发者必须同意在修改类库中的一个类时，不会移除已有的方法，因为那样将会破坏客户端程序员的代码。与之相反的情况更加复杂。在有成员属性的情况下，类库开发者如何知道哪些属性被客户端程序员使用？这同样会发生在那些只为实现类库类而创建的方法上，它们也不是设计成可供客户端程序员调用的。如果类库开发者想删除旧的实现，添加新的实现，结果会怎样呢？任何这些成员的改动都可能破坏客户端程序员的代码。因此类库开发者会被束缚，不能修改任何事物。

为了解决这一问题，Java 提供了访问修饰符供类库开发者指明哪些对于客户端程序员是可用的，哪些是不可用的。访问控制权限的等级，从“最大权限”到“最小权限”依次是：**public**，**protected**，包访问权限（没有关键字）和 **private**。根据上一段的内容，你可能会想，作为一名类库设计者，你会尽可能将一切都设为 **private**，仅向客户端程序员暴露你愿意他们使用的方法。这就是你通常所做的，尽管这与使用其他语言（尤其是 C）编程和访问不受任何限制的人们的直觉相违背。

然而，构建类库的概念和对类库组件的访问控制仍然不完善。其中仍然存在问题就是如何将类库组件捆绑到一个内聚到类库单元中。Java 中通过 package 关键字加以控制，类是在相同包下还是不同包下会影响访问修饰符。所以在这章开始，你将会学习如何将类库组件置于同一个包下，之后你就能明白访问修饰符的全部含义。

Allocate four digits for the year part of a date: a new millennium is commg. — David Martin

8.1 包的概念

包内包含一组类，它们被组织在一个单独的命名空间下。

例如，标准 Java 发布中有一个工具库，它被组织在 **java.util** 命名空间下。**java.util** 中含有一个类，叫做 **ArrayList**。使用 **ArrayList** 的一种方式是用其全名 **java.util.ArrayList**。

```
// hiding/FullQualification.java

public class FullQualification {
    public static void main(String[] args) {
        java.util.ArrayList list = new java.util.ArrayList();
    }
}
```

这种方式使得程序冗长乏味，因此你可以换一种方式，使用 **import** 关键字。如果需要导入某个类，就需要在 **import** 语句中声明：

```
// hiding/SingleImport.java
import java.util.ArrayList;

public class SingleImport {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
    }
}
```

现在你就可以不加限定词，直接使用 **ArrayList** 了。但是对于 **java.util** 包下的其他类，你还是不能用。要导入其中所有的类，只需使用 ***** ，就像本书中其他示例那样：

```
import java.util.*
```

之所以使用导入，是为了提供一种管理命名空间的机制。所有类名之间都是相互隔离的。类 **A** 中的方法 `f()` 不会与类 **B** 中具有相同签名的方法 `f()` 冲突。但是如果类名冲突呢？假设你创建了一个 **Stack** 类，打算安装在一台已经有别人所写的 **Stack** 类的机器上，该怎么办呢？这种类名的潜在冲突，正是我们需要在 Java 中对命名空间进行完全控制的原因。为了解决冲突，我们为每个类创建一个唯一标识符组合。

到目前为止的大部分示例都只存在单个文件，并为本地使用的，所以尚未受到包名的干扰。但是，这些示例其实已经位于包中了，叫做“未命名”包或默认包。这当然是一种选择，为了简单起见，本书其余部分会尽可能采用这种方式。但是，如果你打算为相同机器上的其他 Java 程序创建友好的类库或程序时，就必须仔细考虑以防类名冲突。

一个 Java 源代码文件称为一个*编译单元*（有时也称*翻译单元*）。每个编译单元的文件名后缀必须是 **.java**。在编译单元中可以有一个 **public** 类，它的类名必须与文件名相同（包括大小写，但不包括后缀名 **.java**）。每个编译单元中只能有一个 **public** 类，否则编译器不接受。如果这个编译单元中还有其

他类，那么在包之外是无法访问到这些类的，因为它们不是 **public** 类，此时它们支持主 **public** 类。

8.1.1 代码组织

当编译一个 **.java** 文件时，**.java** 文件的每个类都会有一个输出文件。每个输出的文件名和 **.java** 文件中每个类的类名相同，只是后缀名是 **.class**。因此，在编译少量的 **.java** 文件后，会得到大量的 **.class** 文件。如果你使用过编译型语言，那么你可能习惯编译后产生一个中间文件（通常称为“obj”文件），然后与使用链接器（创建可执行文件）或类库生成器（创建类库）产生的其他同类文件打包到一起的情况。这不是 Java 工作的方式。在 Java 中，可运行程序是一组 **.class** 文件，它们可以打包压缩成一个 Java 文档文件（JAR，使用 **jar** 文档生成器）。Java 解释器负责查找、加载和解释这些文件。

类库是一组类文件。每个源文件通常都含有一个 **public** 类和任意数量的非 **public** 类，因此每个文件都有一个构件。如果把这些组件集中在一起，就需要使用关键字 **package**。

如果你使用了 **package** 语句，它必须是文件中除了注释之外的第一行代码。当你如下这样写：

```
package hiding;
```

意味着这个编译单元是一个名为 **hiding** 类库的一部分。换句话说，你正在声明的编译单元中的 **public** 类名称位于名为 **hiding** 的保护伞下。任何人想要使用该名称，必须指明完整的类名或者使用 **import** 关键字导入 **hiding**。（注意，Java 包名按惯例一律小写，即使中间的单词也需要小写，与驼峰命名不同）

例如，假设文件名是 **MyClass.java**，这意味着文件中只能有一个 **public** 类，且类名必须是 MyClass（大小写也与文件名相同）：

```
// hiding/mypackage/MyClass.java
package hiding.mypackage

public class MyClass {
    // ...
}
```

现在，如果有人想使用 **MyClass** 或 **hiding.mypackage** 中的其他 **public** 类，就必须使用关键字 **import** 来使 **hiding.mypackage** 中的名称可用。还有一种选择是使用完整的名称：

```
// hiding/QualifiedMyClass.java

public class QualifiedMyClass {
    public static void main(String[] args) {
        hiding.mypackage.MyClass m = new hiding.mypackage.MyClass();
    }
}
```

关键字 **import** 使之更简洁：

```
// hiding/ImportedMyClass.java
```

```
import hiding.mypackage.*;

public class ImportedMyClass {
    public static void main(String[] args) {
        MyClass m = new MyClass();
    }
}
```

package 和 **import** 这两个关键字将单一的全局命名空间分隔开，从而避免名称冲突。

8.1.2 创建独一无二的包名

你可能注意到，一个包从未真正被打包成单一的文件，它可以由很多 **.class** 文件构成，因而事情就变得有点复杂了。为了避免这种情况，一种合乎逻辑的做法是将特定包下的所有 **.class** 文件都放在一个目录下。也就是说，利用操作系统的文件结构的层次性。这是 Java 解决混乱问题的一种方式；稍后你还会在我们介绍 **jar** 工具时看到另一种方式。

将所有的文件放在一个子目录还解决了其他的两个问题：创建独一无二的包名和查找可能隐藏于目录结构某处的类。这是通过将 **.class** 文件所在的路径位置编码成 **package** 名称来实现的。按照惯例，**package** 名称是类的创建者的反顺序的 Internet 域名。如果你遵循惯例，因为 Internet 域名是独一无二的，所以你的 **package** 名称也应该是独一无二的，不会发生名称冲突。如果你没有自己的域名，你就得构造一组不大可能与他人重复的组合（比如你的姓名），来创建独一无二的 package 名称。如果你打算发布 Java 程序代码，那么花些力气去获取一个域名是值得的。

此技巧的第二部分是把 **package** 名称分解成你机器上的一个目录，所以当 Java 解释器必须要加载一个.class 文件时，它能定位到 **.class** 文件所在的位置。首先，它找出环境变量 **CLASSPATH**（通过操作系统设置，有时也能通过 Java 的安装程序或基于 Java 的工具设置）。**CLASSPATH** 包含一个或多个目录，用作查找.**class** 文件的根目录。从根目录开始，Java 解释器获取包名并将每个句点替换成反斜杠，生成一个基于根目录的路径名（包名 `foo.bar.baz` 变成 `foobar baz` 或 `foo/bar/baz` 或其它，取决于你的操作系统）。然后这个路径与 **CLASSPATH** 的不同项连接，解释器就在这些目录中查找与你所创建的类名称相关的 **.class** 文件（解释器还会查找某些涉及 Java 解释器所在位置的标准目录）。

为了理解这点，比如说我的域名 **MindviewInc.com**，将之反转并全部改为小写后就是 **com.mindviewinc**，这将作为我创建的类的独一无二的全局名称。（com、edu、org 等扩展名之前在 Java 包中都是大写，但是 Java 2 之后都统一用小写。）我决定再创建一个名为 **simple** 的类库，从而细分名称：

```
package com.mindviewinc.simple;
```

这个包名可以用作下面两个文件的命名空间保护伞：

```
// com/mindviewinc/simple/Vector.java
```

```java
// Creating a package
package com.mindviewinc.simple;

public class Vector {
    public Vector() {
        System.out.println("com.mindviewinc.simple.Vector");
    }
}
```

如前所述，**package** 语句必须是文件的第一行非注释代码。第二个文件看上去差不多：

```java
// com/mindviewinc/simple/List.java
// Creating a package
package com.mindviewinc.simple;

public class List {
    System.out.println("com.mindview.simple.List");
}
```

这两个文件都位于我机器上的子目录中，如下：

```
C:\DOC\Java\com\mindviewinc\simple
```

（注意，本书的每个文件的第一行注释都指明了文件在源代码目录树中的位置——供本书的自动代码提取工具使用。）

如果你回头看这个路径，会看到包名 **com.mindviewinc.simple**，但是路径的第一部分呢？CLASSPATH 环境变量会处理它。我机器上的环境变量部分如下：

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\Java
```

CLASSPATH 可以包含多个不同的搜索路径。

但是在使用 JAR 文件时，有点不一样。你必须在类路径写清楚 JAR 文件的实际名称，不能仅仅是 JAR 文件所在的目录。因此，对于一个名为 **grape.jar** 的 JAR 文件，类路径应包括：

```
CLASSPATH=.;D\JAVA\LIB;C:\flavors\grape.jar
```

一旦设置好类路径，下面的文件就可以放在任意目录：

```java
// hiding/LibTest.java
// Uses the library
import com.mindviewinc.simple.*;

public class LibTest {
    public static void main(String[] args) {
        Vector v = new Vector();
        List l = new List();
    }
}
```

输出:

```
com.mindviewinc.simple.Vector
com.mindviewinc.simple.List
```

当编译器遇到导入 **simple** 库的 **import** 语句时，它首先会在 CLASSPATH 指定的目录中查找子目录 **com/mindviewinc/simple**，然后从已编译的文件中找出名称相符者（对 **Vector** 而言是 **Vector.class**，对 **List** 而言是 **List.class**）。注意，这两个类和其中要访问的方法都必须是 **public** 修饰的。

对于 Java 新手而言，设置 CLASSPATH 是一件麻烦的事（我最初使用时这么觉得），后面版本的 JDK 更加智能。你会发现当你安装好 JDK 时，即使不设置 CLASSPATH，也能够编译和运行基本的 Java 程序。但是，为了编译和运行本书的代码示例（从https://github.com/BruceEckel/OnJava8-examples 取得），你必须将本书程序代码树的基本目录加入到 CLASSPATH 中（gradlew 命令管理自身的 CLASSPATH，所以如果你想直接使用 javac 和 java，不用 Gradle 的话，就需要设置 CLASSPATH）。

8.1.3 冲突

如果通过 * 导入了两个包含相同名字类名的类库，会发生什么？例如，假设程序如下:

```
import com.mindviewinc.simple.*;
import java.util.*;
```

因为 **java.util.*** 也包含了 **Vector** 类，这就存在潜在的冲突。但是只要你不写导致冲突的代码，就不会有问题——这样很好，否则就得做很多类型检查工作来防止那些根本不会出现的冲突。

现在如果要创建一个 Vector 类，就会出现冲突:

```
Vector v = new Vector();
```

这里的 **Vector** 类指的是谁呢？编译器不知道，读者也不知道。所以编译器报错，强制你明确指明。对于标准的 Java 类 **Vector**，你可以这么写:

```
java.util.Vector v = new java.util.Vector();
```

这种写法完全指明了 **Vector** 类的位置（配合 CLASSPATH），那么就没有必要写 **import java.util.*** 语句，除非使用其他来自 **java.util** 中的类。

或者，可以导入单个类以防冲突——只要不在同一个程序中使用有冲突的名字（若使用了有冲突的名字，必须明确指明全名）。

8.1.4 定制工具库

具备了以上知识，现在就可以创建自己的工具库来减少重复的程序代码了。

一般来说，我会使用反转后的域名来命名要创建的工具包，比如 **com.mindviewinc.util**，但为了

简化，这里我把工具包命名为 **onjava**。

比如，下面是“控制流”一章中使用到的 `range()` 方法，采用了 for-in 语法进行简单的遍历：

```java
// onjava/Range.java
// Array creation methods that can be used without
// qualifiers, using static imports:
package onjava;

public class Range {
    // Produce a sequence [0,n)
    public static int[] range(int n) {
        int[] result = new int[n];
        for (int i = 0; i < n; i++) {
            result[i] = i;
        }
        return result;
    }
    // Produce a sequence [start..end)
    public static int[] range(int start, int end) {
        int sz = end - start;
        int[] result = new int[sz];
        for (int i = 0; i < sz; i++) {
            result[i] = start + i;
        }
        return result;
    }
    // Produce sequence [start..end) incrementing by step
    public static int[] range(int start, int end, int step) {
        int sz = (end - start) / step;
        int[] result = new int[sz];
        for (int i = 0; i < sz; i++) {
            result[i] = start + (i * step);
        }
        return result;
    }
}
```

这个文件的位置一定是在某个以一个 CLASSPATH 位置开始，然后接着是 **onjava** 的目录下。编译完之后，就可以在系统的任何地方使用 **import static** 语句来使用这些方法了。

从现在开始，无论何时你创建了有用的新工具，都可以把它加入到自己的类库中。在本书中，你将会看到更多的组件加入到 **onjava** 库。

8.1.5 使用 import 改变行为

Java 没有 C 的*条件编译*功能，该功能使你不必更改任何程序代码而能够切换开关产生不同的行为。Java 之所以去掉此功能，可能是因为 C 在绝大多数情况下使用该功能解决跨平台问题：程序代码的不同部分要根据不同的平台来编译。而 Java 自身就是跨平台设计的，这个功能就没有必要了。

但是，条件编译还有其他的用途。调试是一个很常见的用途，调试功能在开发过程中是开启的，在发布的产品中是禁用的。可以通过改变导入的 **package** 来实现这一目的，修改的方法是将程序中的代码从调试版改为发布版。这个技术可用于任何种类的条件代码。

8.1.6 使用包的忠告

当创建一个包时，包名就隐含了目录结构。这个包必须位于包名指定的目录中，该目录必须在以 CLASSPATH 开始的目录中可以查询到。最初使用关键字 **package** 可能会有点不顺，因为除非遵守“包名对应目录路径”的规则，否则会收到很多意外的运行时错误信息如找不到特定的类，即使这个类就位于同一目录中。如果你收到类似信息，尝试把 **package** 语句注释掉，如果程序能运行的话，你就知道问题出现在哪里了。

注意，编译过的代码通常位于与源代码的不同目录中。这是很多工程的标准，而且集成开发环境（IDE）通常会自动为我们做这些。必须保证 JVM 通过 CLASSPATH 能找到编译后的代码。

8.2 访问权限修饰符

Java 访问权限修饰符 **public**，**protected** 和 **private** 位于定义的类名，属性名和方法名之前。每个访问权限修饰符只能控制它所修饰的对象。

如果不提供访问修饰符，就意味着“包访问权限”。所以无论如何，万物都有某种形式的访问控制权。接下来的几节中，你将学习各种类型的访问权限。

8.2.1 包访问权限

本章之前的所有示例要么使用 **public** 访问修饰符，要么就没使用修饰符（默认访问）。默认访问权限没有关键字，通常被称为包访问权限（有时也称为 friendly）。这意味着当前包中的所有其他类都可以访问那个成员。对于这个包之外的类，这个成员看上去是 **private** 的。由于一个编译单元（即一个文件）只能隶属于一个包，所以通过包访问权限，位于同一编译单元中的所有类彼此之间都是可访问的。

包访问权限可以把相关类聚到一个包下，以便它们能轻易地相互访问。包里的类给它们的包访问权限的成员赋予了相互访问的权限，所以你“拥有”了包内的程序代码。只能通过你所拥有的代码去访问你所拥有的其他代码，这样规定很有意义。构建包访问权限机制是将类聚集在包中的重要原因之一。在许多语言中，在文件中组织定义的方式是任意的，但是在 Java 中你被强制以一种合理的方式组织它们。另外，你可能会将不应该对当前包中的类具有访问权限的类排除在包外。

类控制着哪些代码有权访问自己的成员。其他包中的代码不能一上来就说“嗨，我是 **Bob** 的朋友！”，然后想看到 **Bob** 的 **protected**，包访问权限和 **private** 成员。取得对成员的访问权的唯一方式是：

1. 使成员成为 public。那么无论是谁，无论在哪，都可以访问它。

2. 赋予成员默认包访问权限，不用加任何访问修饰符，然后将其他类放在相同的包内。这样，其他类就可以访问该成员。
3. 在“复用”这一章你将看到，继承的类既可以访问 **public** 成员，也可以访问 **protected** 成员（但不能访问 **private** 成员）。只有当两个类处于同一个包内，它才可以访问包访问权限的成员。但现在不用担心继承和 **protected**。
4. 提供访问器（accessor）和修改器（mutator）方法（有时也称为“get/set” 方法），从而读取和改变值。

8.2.2 public: 接口访问权限

当你使用关键字 **public**，就意味着紧随 public 后声明的成员对于每个人都是可用的，尤其是使用类库的客户端程序员更是如此。假设定义了一个包含下面编译单元的 **dessert** 包：

```java
// hiding/dessert/Cookie.java
// Creates a library
package hiding.dessert;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }

    void bite() {
        System.out.println("bite");
    }
}
```

记住，**Cookie.java** 文件产生的类文件必须位于名为 **dessert** 的子目录中，该子目录在 **hiding**（表明本书的“封装”章节）下，它必须在 CLASSPATH 的几个目录之下。不要错误地认为 Java 总是会将当前目录视作查找行为的起点之一。如果你的 CLASSPATH 中没有 .，Java 就不会查找单独当前目录。

现在，使用 **Cookie** 创建一个程序：

```java
// hiding/Dinner.java
// Uses the library
import hiding.dessert.*;

public class Dinner {
    public static void main(String[] args) {
        Cookie x = new Cookie();
        // -x.bite(); // Can't access
    }
}
```

输出：

```
Cookie constructor
```

你可以创建一个 **Cookie** 对象，因为它构造器和类都是 **public** 的。（后面会看到更多 **public** 的概念）但是，在 **Dinner.java** 中无法访问到 **Cookie** 对象中的 `bite()` 方法，因为 `bite()` 只提供了包访问权限，因而在 **dessert** 包之外无法访问，编译器禁止你使用它。

8.2.3 默认包

你可能惊讶地发现，以下代码尽管看上去破坏了规则，但是仍然可以编译：

```java
// hiding/Cake.java
// Accesses a class in a separate compilation unit
class Cake {
    public static void main(String[] args) {
        Pie x = new Pie();
        x.f();
    }
}
```

输出：

```
Pie.f()
```

同一目录下的第二个文件：

```java
// hiding/Pie.java
// The other class
class Pie {
    void f() {
        System.out.println("Pie.f()");
    }
}
```

最初看上去这两个文件毫不相关，但在 **Cake** 中可以创建一个 **Pie** 对象并调用它的 `f()` 方法。（注意，你的 CLASSPATH 中一定得有 `.`，这样文件才能编译）通常会认为 **Pie** 和 `f()` 具有包访问权限，因此不能被 **Cake** 访问。它们的确具有包访问权限，这是部分正确。**Cake.java** 可以访问它们是因为它们在相同的目录中且没有给自己设定明确的包名。Java 把这样的文件看作是隶属于该目录的默认包中，因此它们为该目录中所有的其他文件都提供了包访问权限。

8.2.4 private: 你无法访问

关键字 **private** 意味着除了包含该成员的类，其他任何类都无法访问这个成员。同一包中的其他类无法访问 **private** 成员，因此这等于说是自己隔离自己。另一方面，让许多人合作创建一个包也是有可能的。使用 **private**，你可以自由地修改那个被修饰的成员，无需担心会影响同一包下的其他类。

默认的包访问权限通常提供了足够的隐藏措施；记住，使用类的客户端程序员无法访问包访问权限成员。这样做很好，因为默认访问权限是一种我们常用的权限（同时也是一种在忘记添加任何访问权限时自动得到的权限）。因此，通常考虑的是把哪些成员声明成 **public** 供客户端程序员使用。所以，最初不常使用关键字 **private**，因为程序没有它也可以照常工作。然而，使用 **private** 是非常重要的，尤其是在多线程环境中。（在“并发编程”一章中将看到）。

以下是一个使用 **private** 的例子：

```
// hiding/IceCream.java
// Demonstrates "private" keyword

class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        //- Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
}
```

以上展示了 **private** 的用武之地：控制如何创建对象，防止别人直接访问某个特定的构造器（或全部构造器）。例子中，你无法通过构造器创建一个 **Sundae** 对象，而必须调用 `makeASundae()` 方法创建对象。

任何可以肯定只是该类的“助手”方法，都可以声明为 **private**，以确保不会在包中的其他地方误用它，也防止了你会去改变或删除它。将方法声明为 **private** 确保了你拥有这种选择权。

对于类中的 **private** 属性也是一样。除非必须公开底层实现（这种情况很少见），否则就将属性声明为 **private**。然而，不能因为类中某个对象的引用是 **private**，就认为其他对象也无法拥有该对象的 **public** 引用（参见附录：对象传递和返回）。

8.2.5 protected: 继承访问权限

要理解 **protected** 的访问权限，我们在内容上需要作一点跳跃。首先，在介绍本书“复用”章节前，你不必真正理解本节的内容。但为了内容的完整性，这里作了简要介绍，举了个使用 **protected** 的例子。

关键字 **protected** 处理的是继承的概念，通过继承可以利用一个现有的类——我们称之为基类，然后添加新成员到现有类中而不必碰现有类。我们还可以改变类的现有成员的行为。为了从一个类中继承，需要声明新类 extends 一个现有类，像这样：

```
class Foo extends Bar {}
```

类定义的其他部分看起来是一样的。

如果你创建了一个新包，并从另一个包继承类，那么唯一能访问的就是被继承类的 **public** 成员。（如果在同一个包中继承，就可以操作所有的包访问权限的成员。）有时，基类的创建者会希望某个特定成员能被继承类访问，但不能被其他类访问。这时就需要使用 **protected**。**protected** 也提供包访问权限，也就是说，相同包内的其他类可以访问 **protected** 元素。

回顾下先前的文件 **Cookie.java**，下面的类不能调用包访问权限的方法 `bite()`：

```
// hiding/ChocolateChip.java
// Can't use package-access member from another package
import hiding.dessert.*;

public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println("ChocolateChip constructor");
    }

    public void chomp() {
        //- bite(); // Can't access bite
    }

    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        x.chomp();
    }
}
```

输出：

```
Cookie constructor
ChocolateChip constructor
```

如果类 **Cookie** 中存在一个方法 `bite()`，那么它的任何子类中都存在 `bite()` 方法。但是因为 `bite()` 具有包访问权限并且位于另一个包中，所以我们在这个包中无法使用它。你可以把它声明为 **public**，但这样一来每个人都能访问它，这可能也不是你想要的。如果你将 **Cookie** 改成如下这样：

```
// hiding/cookie2/Cookie.java
package hiding.cookie2;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }

    protected void bite() {
        System.out.println("bite");
```

```
    }
}
```

这样，`bite()` 对于所有继承 **Cookie** 的类，都是可访问的：

```
// hiding/ChocolateChip2.java
import hiding.cookie2.*;

public class ChocolateChip2 extends Cookie {
    public ChocoalteChip2() {
        System.out.println("ChocolateChip2 constructor");
    }

    public void chomp() {
        bite(); // Protected method
    }

    public static void main(String[] args) {
        ChocolateChip2 x = new ChocolateChip2();
        x.chomp();
    }
}
```

输出：

```
Cookie constructor
ChocolateChip2 constructor
bite
```

尽管 `bite()` 也具有包访问权限，但它不是 **public** 的。

8.2.6 包访问权限 Vs Public 构造器

当你定义一个具有包访问权限的类时，你可以在类中定义一个 public 构造器，编译器不会报错：

```
// hiding/packageaccess/PublicConstructor.java
package hiding.packageaccess;

class PublicConstructor {
    public PublicConstructor() {}
}
```

有一个 Checkstyle 工具，你可以运行命令 **gradlew hiding:checkstyleMain** 使用它，它会指出这种写法是虚假的，而且从技术上来说是错误的。实际上你不能从包外访问到这个 **public** 构造器：

```
// hiding/CreatePackageAccessObject.java
// {WillNotCompile}
import hiding.packageaccess.*;

public class CreatePackageAcessObject {
```

```
    public static void main(String[] args) {
        new PublicConstructor();
    }
}
```

如果你编译下这个类，会得到编译错误信息：

```
CreatePackageAccessObject.java:6:error:
PublicConstructor is not public in hiding.packageaccess;
cannot be accessed from outside package
new PublicConstructor();
^
1 error
```

因此，在一个具有包访问权限的类中定义一个 **public** 的构造器并不能真的使这个构造器成为 **public**，在声明的时候就应该标记为编译时错误。

8.3 接口和实现

访问控制通常被称为实现的隐藏。将数据和方法包装进类中并把具体实现隐藏被称作是封装。其结果就是一个同时带有特征和行为的数据类型。

出于两个重要的原因，访问控制在数据类型内部划定了边界。第一个原因是确立客户端程序员可以使用和不能使用的边界。可以在结构中建立自己的内部机制而不必担心客户端程序员偶尔将内部实现作为他们可以使用的接口的一部分。

这直接引出了第二个原因：将接口与实现分离。如果在一组程序中使用结构，而客户端程序员只能向 **public** 接口发送消息的话，那么就可以自由地修改任何不是 **public** 的事物（例如包访问权限，protected，或 private 修饰的事物），却不会破坏客户端代码。

为了清晰起见，你可以采用一种创建类的风格：**public** 成员放在类的开头，接着是 **protected** 成员，包访问权限成员，最后是 **private** 成员。这么做的好处是类的使用者可以从头读起，首先会看到对他们而言最重要的部分（public 成员，因为可以从文件外访问它们），直到遇到非 **public** 成员时停止阅读，下面就是内部实现了：

```
// hiding/OrganizedByAccess.java

public class OrganizedByAccess {
    public void pub1() {/* ... */}
    public void pub2() {/* ... */}
    public void pub3() {/* ... */}
    private void priv1() {/* ... */}
    private void priv2() {/* ... */}
    private void priv3() {/* ... */}
    private int i;
    // ...
```

```
}
```

这么做只能是程序阅读起来稍微容易一些，因为实现和接口还是混合在一起。也就是说，你仍然能看到源代码——实现部分，因为它就在类中。另外，javadoc 提供的注释文档功能降低了程序代码的可读性对客户端程序员的重要性。将接口展现给类的使用者实际上是类浏览器的任务，类浏览器会展示所有可用的类，并告诉你如何使用它们（比如说哪些成员可用）。在 Java 中，JDK 文档起到了类浏览器的作用。

8.4 类访问权限

访问权限修饰符也可以用于确定类库中的哪些类对于类库的使用者是可用的。如果希望某个类可以被客户端程序员使用，就把关键字 **public** 作用于整个类的定义。这甚至控制着客户端程序员能否创建类的对象。

为了控制一个类的访问权限，修饰符必须出现在关键字 **class** 之前：

```
public class Widget {
```

如果你的类库名是 **hiding**，那么任何客户端程序员都可以通过如下声明访问 **Widget**：

```
import hiding.Widget;
```

或者

```
import hiding.*;
```

这里有一些额外的限制：

1. 每个编译单元（即每个文件）中只能有一个 **public** 类。这表示，每个编译单元有一个公共的接口用 **public** 类表示。该接口可以包含许多支持包访问权限的类。一旦一个编译单元中出现一个以上的 **public** 类，编译就会报错。
2. **public** 类的名称必须与含有该编译单元的文件名相同，包括大小写。所以对于 **Widget** 来说，文件名必须是 **Widget.java**，不能是 **widget.java** 或 **WIDGET.java**。再次强调，如果名字不匹配，编译器会报错。
3. 虽然不是很常见，但是编译单元内没有 **public** 类也是可能的。这时可以随意命名文件（尽管随意命名会让代码的阅读者和维护者感到困惑）。

如果获取了一个在 **hiding** 包中的类，只用来完成 **Widget** 或 **hiding** 包下一些其他 **public** 类所要执行的任务，怎么办呢？你不想自找麻烦为客户端程序员创建说明文档，并且你认为不久后会完全改变原有方案并将旧版本删除，替换成新版本。为了保留此灵活性，需要确保客户端程序员不依赖隐藏在 **hiding** 中的任何特定细节，那么把 **public** 关键字从类中去掉，给予它包访问权限，就可以了。

当你创建了一个包访问权限的类，把类中的属性声明为 **private** 仍然是有意义的——应该尽可能将所有属性都声明为 **private**，但是通常把方法声明成与类（包访问权限）相同的访问权限也是合理的。

A program designed .for inputs from people is usually stressed beyond the breaking point by computer-generated inputs. — Dennis Ritchie

由于一个包访问权限的类只能被用于包内，除非你被强制将某些方法声明为 **public**，这种情况下，编译器会告诉你。

注意，类既不能是 **private** 的（这样除了该类自身，任何类都不能访问它），也不能是 **protected** 的。所以对于类的访问权限只有两种选择：包访问权限或者 **public**。为了防止类被外界访问，可以将所有的构造器声明为 **private**，这样只有你自己能创建对象（在类的 static 成员中）：

```java
// hiding/Lunch.java
// Demonstrates class access specifiers. Make a class
// effectively private with private constructors:

class Soup1 {
    private Soup1() {}

    public static Soup1 makeSoup() { // [1]
        return new Soup1();
    }
}

class Soup2 {
    private Soup2() {}

    private static Soup2 ps1 = new Soup2(); // [2]

    public static Soup2 access() {
        return ps1;
    }

    public void f() {}
}
// Only one public class allowed per file:
public class Lunch {
    void testPrivate() {
        // Can't do this! Private constructor:
        //- Soup1 soup = new Soup1();
    }

    void testStatic() {
        Soup1 soup = Soup1.makeSoup();
    }

    void testSingleton() {
        Soup2.access().f();
    }
}
```

可以像 [1] 那样通过 **static** 方法创建对象，也可以像 [2] 那样先创建一个静态对象，当用户需要访问它时返回对象的引用即可。

到目前为止，大部分的方法要么返回 void，要么返回基本类型，所以 [1] 处的定义乍看之下会有点

困惑。方法名（**makeSoup**）前面的 **Soup1** 表明了方法返回的类型。到目前为止，这里经常是 **void**，即不返回任何东西。然而也可以返回对象的引用，就像这里一样。这个方法返回了对 **Soup1** 类对象的引用。

Soup1 和 **Soup2** 展示了如何通过将你所有的构造器声明为 **private** 的方式防止直接创建某个类的对象。记住，如果你不显式地创建构造器，编译器会自动为你创建一个无参构造器（没有参数的构造器）。如果我们编写了无参构造器，那么编译器就不会自动创建构造器了。将构造器声明为 **private**，那么谁也无法创建该类的对象了。但是现在别人该怎么使用这个类呢？上述例子给出了两个选择。在 **Soup1** 中，有一个 **static** 方法，它的作用是创建一个新的 **Soup1** 对象并返回对象的引用。如果想要在返回引用之前在 **Soup1** 上做一些额外操作，或是记录创建了多少个 **Soup1** 对象（可以用来限制数量），这种做法是有用的。

Soup2 用到了所谓的*设计模式*。这种模式叫做*单例模式*，因为它只允许创建类的一个对象。**Soup2** 类的对象是作为 **Soup2** 的 **static private** 成员而创建的，所以有且只有一个，你只能通过 **public** 修饰的 `access()` 方法访问到这个对象。

8.5 本章小结

无论在什么样的关系中，划定一些供各成员共同遵守的界限是很重要的。当你创建了一个类库，也就与该类库的使用者产生了联系，他们是类库的客户端程序员，需要使用你的类库创建应用或更大的类库。

没有规则，客户端程序员就可以对类的所有成员为所欲为，即使你希望他们不要操作部分成员。这种情况下，所有事物都是公开的。

本章讨论了类库是如何通过类构建的：首先，介绍了将一组类打包到类库的方式，其次介绍了类如何控制对其成员的访问。

据估计，用 C 语言开发项目，当代码量达到 5 万行和 10 万行时就会出现问题，因为 C 语言只有单一的命名空间，名称开始冲突造成额外的管理开销。在 Java 中，关键字 **package**，包命名模式和关键字 **import** 给了你对于名称的完全控制权，因此可以轻易地避免名称冲突的问题。

控制成员访问权限有两个原因。第一个原因是使用户不要接触他们不该接触的部分，这部分对于类内部来说是必要的，但是不属于客户端程序员所需接口的一部分。因此将方法和属性声明为 **private** 对于客户端程序员来说是一种服务，可以让他们清楚地看到什么是重要的，什么可以忽略。这可以简化他们对类的理解。

第二个也是最重要的原因是为了让类库设计者更改类内部的工作方式，而不用担心会影响到客户端程序员。比如最初以某种方式创建一个类，随后发现如果更改代码结构可以极大地提高运行速度。如果接口与实现被明确地隔离和保护，你可以实现这一目的，而不必强制客户端程序员重新编写代码。访问权限控制确保客户端程序员不会依赖某个类的底层实现的任何部分。

当你具备更改底层实现的能力时，不但可以自由地改善设计，还可能会随意地犯错。无论如何细心

地计划和设计，都有可能犯错。当了解到犯错是相对安全的时候，你可以更加放心地实验，更快地学会，更快地完成项目。

类的 **public** 接口是用户真正看到的，所以在分析和设计阶段决定这部分接口是最重要的部分。尽管如此，你仍然有改变的空间。如果最初没有创建出正确的接口，可以添加更多的方法，只要你不删除那些客户端程序员已经在他们的代码中使用的东西。

注意到访问权限控制关注的是类库创建者和外部使用者之间的关系，一种交流方式。很多情况下，事实并非如此。例如，你自己编写了所有的代码，或者在一个小组中工作，所有的东西都放在同一个包下。这些情况下，交流方式则是另外一种，此时严格地遵循访问权限规则也许不是最佳选择，默认（包）访问权限也许就足够好了。

9 复用

> 代码复用是面向对象编程（OOP）最具魅力的原因之一。

对于像 C 语言等面向过程语言来说，“复用”通常指的就是“复制代码”。任何语言都可通过简单复制来达到代码复用的目的，但是这样做的效果并不好。Java 围绕“类”（Class）来解决问题。我们可以直接使用别人构建或调试过的代码，而非创建新类、重新开始。

如何在不污染源代码的前提下使用现存代码是需要技巧的。在本章里，你将学习到两种方式来达到这个目的：

1. 第一种方式直接了当。在新类中创建现有类的对象。这种方式叫做“组合”（Composition），通过这种方式复用代码的功能，而非其形式。
2. 第二种方式更为微妙。创建现有类类型的新类。照字面理解：采用现有类形式，又无需在编码时改动其代码，这种方式就叫做“继承”（Inheritance），编译器会做大部分的工作。**继承**是面向对象编程（OOP）的重要基础之一。更多功能相关将在多态（Polymorphism）章节中介绍。

组合与继承的语法、行为上有许多相似的地方（这其实是有道理的，毕竟都是基于现有类型构建新的类型）。在本章中，你会学到这两种代码复用的方法。

9.1 组合语法

在前面的学习中，“组合”（Composition）已经被多次使用。你仅需要把对象的引用（object references）放置在一个新的类里，这就使用了组合。例如，假设你需要一个对象，其中内置了几个 **String** 对象，两个基本类型（primitives）的属性字段，一个其他类的对象。对于非基本类型对象，将引用直接放置在新类中，对于基本类型属性字段则仅进行声明。

```
// reuse/SprinklerSystem.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Composition for code reuse

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
```

```
    s = "Constructed";
  }
  @Override
  public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  private WaterSource source = new WaterSource();
  private int i;
  private float f;
  @Override
  public String toString() {
    return
      "valve1 = " + valve1 + " " +
      "valve2 = " + valve2 + " " +
      "valve3 = " + valve3 + " " +
      "valve4 = " + valve4 + "\n" +
      "i = " + i + " " + "f = " + f + " " +
      "source = " + source; // [1]
  }
  public static void main(String[] args) {
    SprinklerSystem sprinklers = new SprinklerSystem();
    System.out.println(sprinklers);
  }
}
/* Output:
WaterSource()
valve1 = null valve2 = null valve3 = null valve4 = null
i = 0 f = 0.0 source = Constructed
*/
```

这两个类中定义的一个方法是特殊的: `toString()`。每个非基本类型对象都有一个 `toString()` 方法，在编译器需要字符串但它有对象的特殊情况下调用该方法。因此，在 [1] 中，编译器看到你试图“添加”一个 **WaterSource** 类型的字符串对象。因为字符串只能拼接另一个字符串，所以它就先会调用 `toString()` 将 **source** 转换成一个字符串。然后，它可以拼接这两个字符串并将结果字符串传递给 `System.out.println()`。要对创建的任何类允许这种行为，只需要编写一个 **toString()** 方法。在 `toString()` 上使用 **@Override** 注释来告诉编译器，以确保正确地覆盖。**@Override** 是可选的，但它有助于验证你没有拼写错误 (或者更微妙地说，大小写字母输入错误)。类中的基本类型字段自动初始化为零，正如 **object Everywhere** 一章中所述。但是对象引用被初始化为 **null**，如果你尝试调用其任何一个方法，你将得到一个异常（一个运行时错误）。方便的是，打印 **null** 引用却不会得到异常。

编译器不会为每个引用创建一个默认对象，这是有意义的，因为在许多情况下，这会导致不必要的开销。初始化引用有四种方法:

1. 当对象被定义时。这意味着它们总是在调用构造函数之前初始化。
2. 在该类的构造函数中。

在我所有的程序错误中，80% 是语法错误，剩下 20% 里，80% 是简单的逻辑错误，在剩下 4% 里，80% 是指针错误，只有余下的 0.8% 才是困难的问题。 — Marc Donner

3. 在实际使用对象之前。这通常称为*延迟初始化*。在对象创建开销大且不需要每次都创建对象的情况下，它可以减少开销。
4. 使用实例初始化。

以上四种实例创建的方法例子在这：

```java
// reuse/Bath.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Constructor initialization with composition

class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = "Constructed";
  }
  @Override
  public String toString() { return s; }
}

public class Bath {
  private String // Initializing at point of definition:
    s1 = "Happy",
    s2 = "Happy",
    s3, s4;
  private Soap castille;
  private int i;
  private float toy;
  public Bath() {
    System.out.println("Inside Bath()");
    s3 = "Joy";
    toy = 3.14f;
    castille = new Soap();
  }
  // Instance initialization:
  { i = 47; }
  @Override
  public String toString() {
    if(s4 == null) // Delayed initialization:
      s4 = "Joy";
    return
      "s1 = " + s1 + "\n" +
      "s2 = " + s2 + "\n" +
      "s3 = " + s3 + "\n" +
      "s4 = " + s4 + "\n" +
      "i = " + i + "\n" +
      "toy = " + toy + "\n" +
      "castille = " + castille;
```

在系统测试阶段找出并修正错误，要比开发者自己完成这一工作多付出 2 倍的努力。而当系统已经交付使用之后找出并修正错误，要比系统测试阶段多付出 9 倍的努力。因此，请坚持让开发者进行单元测试吧。 — Larry Bernstain

```
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    System.out.println(b);
  }
}
/* Output:
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
*/
```

在 **Bath** 构造函数中，有一个代码块在所有初始化发生前就已经执行了。当你不在定义处初始化时，仍然不能保证在向对象引用发送消息之前执行任何初始化——如果你试图对未初始化的引用调用方法，则未初始化的引用将产生运行时异常。

当调用 `toString()` 时，它将赋值 s4，以便在使用字段的时候所有的属性都已被初始化。

9.2 继承语法

继承是所有面向对象语言的一个组成部分。事实证明，在创建类时总是要继承，因为除非显式地继承其他类，否则就隐式地继承 Java 的标准根类对象（Object）。

组合的语法很明显，但是继承使用了一种特殊的语法。当你继承时，你说，“这个新类与那个旧类类似。你可以在类主体的左大括号前的代码中声明这一点，使用关键字 **extends** 后跟基类的名称。当你这样做时，你将自动获得基类中的所有字段和方法。这里有一个例子:

```
// reuse/Detergent.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Inheritance syntax & properties

class Cleanser {
  private String s = "Cleanser";
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  @Override
  public String toString() { return s; }
```

```
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    System.out.println(x);
  }
}

public class Detergent extends Cleanser {
  // Change a method:
  @Override
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    System.out.println(x);
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
}
/* Output:
Cleanser dilute() apply() Detergent.scrub() scrub()
foam()
Testing base class:
Cleanser dilute() apply() scrub()
*/
```

这演示了一些特性。首先，在 **Cleanser** 的 `append()` 方法中，使用 `+=` 操作符将字符串连接到 **s**，这是 Java 设计人员“重载”来处理字符串的操作符之一 (还有 `+`)。

第二，**Cleanser** 和 **Detergent** 都包含一个 `main()` 方法。你可以为每个类创建一个 `main()`；这允许对每个类进行简单的测试。当你完成测试时，不需要删除 `main()`；你可以将其留在以后的测试中。即使程序中有很多类都有 `main()` 方法，惟一运行的只有在命令行上调用的 `main()`。这里，当你使用 **java Detergent** 时候，就调用了 `Detergent.main()`。但是你也可以使用 **java Cleanser** 来调用 `Cleanser.main()`，即使 **Cleanser** 不是一个公共类。即使类只具有包访问权，也可以访问 `public main()`。

在这里，`Detergent.main()` 显式地调用 `Cleanser.main()`，从命令行传递相同的参数 (当然，你可以传递任何字符串数组)。

Cleanser 中的所有方法都是公开的。请记住，如果不使用任何访问修饰符，则成员默认为包访问

权限，这只允许包内成员访问。因此，如果没有访问修饰符，那么包内的任何人都可以使用这些方法。例如，**Detergent** 就没有问题。但是，如果其他包中的类继承 **Cleanser**，则该类只能访问 **Cleanser** 的公共成员。因此，为了允许继承，一般规则是所有字段为私有，所有方法为公共。(受保护成员也允许派生类访问; 你以后会知道的。) 在特定的情况下，你必须进行调整，但这是一个有用的指南。

Cleanser 的接口中有一组方法: `append()`、`dilute()`、`apply()`、`scrub()` 和 `toString()`。因为 **Detergent** 是从 **Cleanser** 派生的 (通过 **extends** 关键字)，所以它会在其接口中自动获取所有这些方法，即使你没有在 **Detergent** 中看到所有这些方法的显式定义。那么，可以把继承看作是复用类。如在 `scrub()` 中所见，可以使用基类中定义的方法并修改它。在这里，你可以在新类中调用基类的该方法。但是在 `scrub()` 内部，不能简单地调用 `scrub()`，因为这会产生递归调用。为了解决这个问题，Java 的 **super** 关键字引用了当前类继承的“超类”(基类)。因此表达式 `super.scrub()` 调用方法 `scrub()` 的基类版本。

继承时，你不受限于使用基类的方法。你还可以像向类添加任何方法一样向派生类添加新方法: 只需定义它。方法 `foam()` 就是一个例子。`Detergent.main()` 中可以看到，对于 **Detergent** 对象，你可以调用 **Cleanser** 和 **Detergent** 中可用的所有方法 (如 `foam()`)。

9.2.1 初始化基类

现在涉及到两个类: 基类和派生类。想象派生类生成的结果对象可能会让人感到困惑。从外部看，新类与基类具有相同的接口，可能还有一些额外的方法和字段。但是继承并不只是复制基类的接口。当你创建派生类的对象时，它包含基类的子对象。这个子对象与你自己创建基类的对象是一样的。只是从外部看，基类的子对象被包装在派生类的对象中。

必须正确初始化基类子对象，而且只有一种方法可以保证这一点: 通过调用基类构造函数在构造函数中执行初始化，该构造函数具有执行基类初始化所需的所有适当信息和特权。Java 自动在派生类构造函数中插入对基类构造函数的调用。下面的例子展示了三个层次的继承:

```
// reuse/Cartoon.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Constructor calls during inheritance

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
```

```
}

public class Cartoon extends Drawing {
  public Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
}
/* Output:
Art constructor
Drawing constructor
Cartoon constructor
*/
```

构造从基类“向外”进行，因此基类在派生类构造函数能够访问它之前进行初始化。即使不为 **Cartoon** 创建构造函数，编译器也会为你合成一个无参数构造函数，调用基类构造函数。尝试删除 **Cartoon** 构造函数来查看这个。

9.2.2 带参数的构造函数

上面的所有例子中构造函数都是无参数的; 编译器很容易调用这些构造函数，因为不需要参数。如果没有无参数的基类构造函数，或者必须调用具有参数的基类构造函数，则必须使用 **super** 关键字和适当的参数列表显式地编写对基类构造函数的调用:

```
// reuse/Chess.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Inheritance, constructors and arguments

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
```

```
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
}
/* Output:
Game constructor
BoardGame constructor
Chess constructor
*/
```

如果没有在 **BoardGame** 构造函数中调用基类构造函数，编译器就会报错找不到 `Game()` 的构造函数。此外，对基类构造函数的调用必须是派生类构造函数中的第一个操作。(如果你写错了，编译器会提醒你。)

9.3 委托

Java 不直接支持的第三种重用关系称为委托。这介于继承和组合之间，因为你将一个成员对象放在正在构建的类中 (比如组合)，但同时又在新类中公开来自成员对象的所有方法 (比如继承)。例如，宇宙飞船需要一个控制模块:

```
// reuse/SpaceShipControls.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.

public class SpaceShipControls {
  void up(int velocity) {}
  void down(int velocity) {}
  void left(int velocity) {}
  void right(int velocity) {}
  void forward(int velocity) {}
  void back(int velocity) {}
  void turboBoost() {}
}
```

建造宇宙飞船的一种方法是使用继承:

```
// reuse/DerivedSpaceShip.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.

public class
DerivedSpaceShip extends SpaceShipControls {
  private String name;
```

```
  public DerivedSpaceShip(String name) {
    this.name = name;
  }
  @Override
  public String toString() { return name; }
  public static void main(String[] args) {
    DerivedSpaceShip protector =
        new DerivedSpaceShip("NSEA Protector");
    protector.forward(100);
  }
}
```

然而，**DerivedSpaceShip** 并不是真正的“一种” **SpaceShipControls**，即使你“告诉” **DerivedSpaceShip** 调用 `forward()`。更准确地说，一艘宇宙飞船包含了 **SpaceShipControls**，同时 **SpaceShipControls** 中的所有方法都暴露在宇宙飞船中。委托解决了这个难题:

```
// reuse/SpaceShipDelegation.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.

public class SpaceShipDelegation {
  private String name;
  private SpaceShipControls controls =
    new SpaceShipControls();
  public SpaceShipDelegation(String name) {
    this.name = name;
  }
  // Delegated methods:
  public void back(int velocity) {
    controls.back(velocity);
  }
  public void down(int velocity) {
    controls.down(velocity);
  }
  public void forward(int velocity) {
    controls.forward(velocity);
  }
  public void left(int velocity) {
    controls.left(velocity);
  }
  public void right(int velocity) {
    controls.right(velocity);
  }
  public void turboBoost() {
    controls.turboBoost();
  }
  public void up(int velocity) {
    controls.up(velocity);
  }
```

```
  public static void main(String[] args) {
    SpaceShipDelegation protector =
      new SpaceShipDelegation("NSEA Protector");
    protector.forward(100);
  }
}
```

方法被转发到底层 **control** 对象，因此接口与继承的接口是相同的。但是，你对委托有更多的控制，因为你可以选择只在成员对象中提供方法的子集。

虽然 Java 语言不支持委托，但是开发工具常常支持。例如，上面的例子是使用 JetBrains Idea IDE 自动生成的。

9.4 结合组合与继承

你将经常同时使用组合和继承。下面的例子展示了使用继承和组合创建类，以及必要的构造函数初始化:

```
// reuse/PlaceSetting.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Combining composition & inheritance

class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}

class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println("DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
}

class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
```

```
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    System.out.println("Knife constructor");
  }
}

// A cultural way of doing something:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}

public class PlaceSetting extends Custom {
  private Spoon sp;
  private Fork frk;
  private Knife kn;
  private DinnerPlate pl;
  public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println("PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
  }
}
/* Output:
Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor
```

```
*/
```

尽管编译器强制你初始化基类，并要求你在构造函数的开头就初始化基类，但它并不监视你以确保你初始化了成员对象。注意类是如何干净地分离的。你甚至不需要方法重用代码的源代码。你最多只导入一个包。(这对于继承和组合都是正确的。)

9.4.1 保证适当的清理

Java 没有 C++ 中析构函数的概念，析构函数是在对象被销毁时自动调用的方法。原因可能是，在 Java 中，通常是忘掉而不是销毁对象，从而允许垃圾收集器根据需要回收内存。通常这是可以的，但是有时你的类可能在其生命周期中执行一些需要清理的活动。初始化和清理章节提到，你无法知道垃圾收集器何时会被调用，甚至它是否会被调用。因此，如果你想为类清理一些东西，必须显式地编写一个特殊的方法来完成它，并确保客户端程序员知道他们必须调用这个方法。最重要的是——正如在“异常”章节中描述的——你必须通过在 **finally** 子句中放置此类清理来防止异常。

请考虑一个在屏幕上绘制图片的计算机辅助设计系统的例子:

```java
// reuse/CADSystem.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Ensuring proper cleanup
// {java reuse.CADSystem}
package reuse;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void dispose() {
    System.out.println("Shape dispose");
  }
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
    System.out.println("Drawing Circle");
  }
  @Override
  void dispose() {
    System.out.println("Erasing Circle");
    super.dispose();
  }
}

class Triangle extends Shape {
```

It takes three times the effort to find and fix bugs in system test than when done by the developer. It takes ten times the effort to find and fix bugs in the field than when done in system test. Therefore, insist on unit tests by the developer.
— Larry Bernstein

```
  Triangle(int i) {
    super(i);
    System.out.println("Drawing Triangle");
  }
  @Override
  void dispose() {
    System.out.println("Erasing Triangle");
    super.dispose();
  }
}

class Line extends Shape {
  private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;
    System.out.println(
      "Drawing Line: " + start + ", " + end);
  }
  @Override
  void dispose() {
    System.out.println(
      "Erasing Line: " + start + ", " + end);
    super.dispose();
  }
}

public class CADSystem extends Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[3];
  public CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < lines.length; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    System.out.println("Combined constructor");
  }
  @Override
  public void dispose() {
    System.out.println("CADSystem.dispose()");
    // The order of cleanup is the reverse
    // of the order of initialization:
    t.dispose();
    c.dispose();
    for(int i = lines.length - 1; i >= 0; i--)
      lines[i].dispose();
    super.dispose();
  }
```

```
  public static void main(String[] args) {
    CADSystem x = new CADSystem(47);
    try {
      // Code and exception handling...
    } finally {
      x.dispose();
    }
  }
}
/* Output:
Shape constructor
Shape constructor
Drawing Line: 0, 0
Shape constructor
Drawing Line: 1, 1
Shape constructor
Drawing Line: 2, 4
Shape constructor
Drawing Circle
Shape constructor
Drawing Triangle
Combined constructor
CADSystem.dispose()
Erasing Triangle
Shape dispose
Erasing Circle
Shape dispose
Erasing Line: 2, 4
Shape dispose
Erasing Line: 1, 1
Shape dispose
Erasing Line: 0, 0
Shape dispose
Shape dispose
*/
```

这个系统中的所有东西都是某种 **Shape** (它本身是一种 **Object**，因为它是从根类隐式继承的) 。除了使用 **super** 调用该方法的基类版本外，每个类还覆盖 `dispose()` 方法。特定的 **Shape** 类——**Circle**、**Triangle** 和 **Line**，都有 “draw” 构造函数，尽管在对象的生命周期中调用的任何方法都可以负责做一些需要清理的事情。每个类都有自己的 `dispose()` 方法来将非内存的内容恢复到对象存在之前的状态。

在 `main()` 中，有两个关键字是你以前没有见过的，在“异常”一章之前不会详细解释：**try** 和 **finally**。**try** 关键字表示后面的块 (用花括号分隔) 是一个受保护的区域，这意味着它得到了特殊处理。其中一个特殊处理是，无论 **try** 块如何退出，在这个保护区域之后的 **finally** 子句中的代码总是被执行。(通过异常处理，可以用许多不同寻常的方式留下 **try** 块。) 这里，**finally** 子句的意思是，“无论发生什么，始终调用 `x.dispose()`。”

在清理方法 (在本例中是 dispose()) 中，还必须注意基类和成员对象清理方法的调用顺序，以防一个子对象依赖于另一个子对象。首先，按与创建的相反顺序执行特定于类的所有清理工作。(一般来说，这要求基类元素仍然是可访问的。) 然后调用基类清理方法，如这所示。

在很多情况下，清理问题不是问题；你只需要让垃圾收集器来完成这项工作。但是，当你必须执行显式清理时，就需要多做努力，更加细心，因为在垃圾收集方面没有什么可以依赖的。可能永远不会调用垃圾收集器。如果调用，它可以按照它想要的任何顺序回收对象。除了内存回收外，你不能依赖垃圾收集来做任何事情。如果希望进行清理，可以使用自己的清理方法，不要使用 finalize()。

9.4.2 名称隐藏

如果 Java 基类的方法名多次重载，则在派生类中重新定义该方法名不会隐藏任何基类版本。不管方法是在这个级别定义的，还是在基类中定义的，重载都会起作用:

```java
// reuse/Hide.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Overloading a base-class method name in a derived
// class does not hide the base-class versions

class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {
    System.out.println("doh(Milhouse)");
  }
}

public class Hide {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1);
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
```

```
  }
}
/* Output:
doh(float)
doh(char)
doh(float)
doh(Milhouse)
*/
```

Homer 的所有重载方法在 **Bart** 中都是可用的，尽管 **Bart** 引入了一种新的重载方法。在下一章中你将看到，使用与基类中完全相同的签名和返回类型覆盖相同名称的方法要常见得多。否则就会令人困惑。

你已经看到了 Java 5 **@Override** 注释，它不是关键字，但是可以像使用关键字一样使用它。当你打算重写一个方法时，你可以选择添加这个注释，如果你不小心用了重载而不是重写，编译器会产生一个错误消息:

```
// reuse/Lisa.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// {WillNotCompile}

class Lisa extends Homer {
  @Override void doh(Milhouse m) {
    System.out.println("doh(Milhouse)");
  }
}
```

{WillNotCompile} 标记将该文件排除在本书的 **Gradle** 构建之外，但是如果你手工编译它，你将看到: 方法不会覆盖超类中的方法，**@Override** 注释防止你意外地重载。

9.5 组合与继承的选择

组合和继承都允许在新类中放置子对象（组合是显式的，而继承是隐式的）。你或许想知道这二者之间的区别，以及怎样在二者间做选择。

当你想在新类中包含一个已有类的功能时，使用组合，而非继承。也就是说，在新类中嵌入一个对象（通常是私有的），以实现其功能。新类的使用者看到的是你所定义的新类的接口，而非嵌入对象的接口。

有时让类的用户直接访问到新类中的组合成分是有意义的。只需将成员对象声明为 **public** 即可（可以把这当作“半委托”的一种）。成员对象隐藏了具体实现，所以这是安全的。当用户知道你正在组装一组部件时，会使得接口更加容易理解。下面的 car 对象是个很好的例子:

```
// reuse/Car.java
```

```java
// Composition with public objects
class Engine {
    public void start() {}
    public void rev() {}
    public void stop() {}
}

class Wheel {
    public void inflate(int psi) {}
}

class Window {
    public void rollup() {}
    public void rolldown() {}
}

class Door {
    public Window window = new Window();

    public void open() {}
    public void close() {}
}

public class Car {
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door left = new Door(), right = new Door(); // 2-door

    public Car() {
        for (int i = 0; i < 4; i++) {
            wheel[i] = new Wheel();
        }
    }

    public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel[0].inflate(72);
    }
}
```

因为在这个例子中 car 的组合也是问题分析的一部分（不是底层设计的部分），所以声明成员为 **public** 有助于客户端程序员理解如何使用类，且降低了类创建者面临的代码复杂度。但是，记住这是一个特例。通常来说，属性还是应该声明为 **private**。

当使用继承时，使用一个现有类并开发出它的新版本。通常这意味着使用一个通用类，并为了某个特殊需求将其特殊化。稍微思考下，你就会发现，用一个交通工具对象来组成一部车是毫无意义的——车不包含交通工具，它就是交通工具。这种“是一个”的关系是用继承来表达的，而 “有一个 “的关系则用组合来表达。

9.6 protected

即然你已经接触到继承，关键字 **protected** 就变得有意义了。在理想世界中，仅靠关键字 **private** 就足够了。在实际项目中，却经常想把一个事物尽量对外界隐藏，而允许派生类的成员访问。

关键字 **protected** 就起这个作用。它表示“就类的用户而言，这是 **private** 的。但对于任何继承它的子类或在同一包中的类，它是可访问的。”（**protected** 也提供了包访问权限）

尽管可以创建 **protected** 属性，但是最好的方式是将属性声明为 **private** 以一直保留更改底层实现的权利。然后通过 **protected** 控制类的继承者的访问权限。

```java
// reuse/Orc.java
// The protected keyword
class Villain {
    private String name;

    protected void set(String nm) {
        name = nm;
    }

    Villain(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "I'm a Villain and my name is " + name;
    }
}

public class Orc extends Villain {
    private int orcNumber;

    public Orc(String name, int orcNumber) {
        super(name);
        this.orcNumber = orcNumber;
    }

    public void change(String name, int orcNumber) {
        set(name); // Available because it's protected
        this.orcNumber = orcNumber;
    }

    @Override
    public String toString() {
        return "Orc " + orcNumber + ": " + super.toString();
    }

    public static void main(String[] args) {
        Orc orc = new Orc("Limburger", 12);
```

```
        System.out.println(orc);
        orc.change("Bob", 19);
        System.out.println(orc);
    }
}
```

输出:

```
Orc 12: I'm a Villain and my name is Limburger
Orc 19: I'm a Villain and my name is Bob
```

`change()` 方法可以访问 `set()` 方法，因为 `set()` 方法是 **protected**。注意到，类 **Orc** 的 `toString()` 方法也使用了基类的版本。

9.7 向上转型

继承最重要的方面不是为新类提供方法。它是新类与基类的一种关系。简而言之，这种关系可以表述为“新类是已有类的一种类型”。

这种描述并非是解释继承的一种花哨方式，这是直接由语言支持的。例如，假设有一个基类 **Instrument** 代表音乐乐器和一个派生类 **Wind**。因为继承保证了基类的所有方法在派生类中也是可用的，所以任意发送给该基类的消息也能发送给派生类。如果 **Instrument** 有一个 `play()` 方法，那么 **Wind** 也有该方法。这意味着你可以准确地说 **Wind** 对象也是一种类型的 **Instrument**。下面例子展示了编译器是如何支持这一概念的:

```
// reuse/Wind.java
// Inheritance & upcasting
class Instrument {
    public void play() {}

    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
}
```

`tune()` 方法接受了一个 **Instrument** 类型的引用。但是，在 **Wind** 的 `main()` 方法里，`tune()`

方法却传入了一个 **Wind** 引用。鉴于 Java 对类型检查十分严格，一个接收一种类型的方法接受了另一种类型看起来很奇怪，除非你意识到 **Wind** 对象同时也是一个 **Instrument** 对象，而且 **Instrument** 的 `tune` 方法一定会存在于 **Wind** 中。在 `tune()` 中，代码对 **Instrument** 和所有 **Instrument** 的派生类起作用，这种把 **Wind** 引用转换为 **Instrument** 引用的行为称作*向上转型*。

该术语是基于传统的类继承图：图最上面是根，然后向下铺展。（当然你可以以任意方式画你认为有帮助的类图。）于是，**Wind.java** 的类图是：

Figure 9.1: Wind 类图

继承图中派生类转型为基类是向上的，所以通常称作*向上转型*。因为是从一个更具体的类转化为一个更一般的类，所以向上转型永远是安全的。也就是说，派生类是基类的一个超集。它可能比基类包含更多的方法，但它必须至少具有与基类一样的方法。在向上转型期间，类接口只可能失去方法，不会增加方法。这就是为什么编译器在没有任何明确转型或其他特殊标记的情况下，仍然允许向上转型的原因。

也可以执行与向上转型相反的向下转型，但是会有问题，对于该问题会放在下一章和“类型信息”一章进行更深入的探讨。

9.7.1 再论组合和继承

在面向对象编程中，创建和使用代码最有可能的方法是将数据和方法一起打包到类中，然后使用该类的对象。也可以使用已有的类通过组合来创建新类。继承其实不太常用。因此尽管在教授 OOP 的过

程中我们多次强调继承，但这并不意味着要尽可能使用它。恰恰相反，尽量少使用它，除非确实使用继承是有帮助的。一种判断使用组合还是继承的最清晰的方法是问一问自己是否需要把新类向上转型为基类。如果必须向上转型，那么继承就是必要的，但如果不需要，则要进一步考虑是否该采用继承。“多态”一章提出了一个使用向上转型的最有力的理由，但是只要记住问一问“我需要向上转型吗？”，就能在这两者中作出较好的选择。

9.8 final 关键字

根据上下文环境，Java 的关键字 **final** 的含义有些微的不同，但通常它指的是“这是不能被改变的”。防止改变有两个原因：设计或效率。因为这两个原因相差很远，所以有可能误用关键字 **final**。

以下几节讨论了可能使用 **final** 的三个地方：数据、方法和类。

9.8.1 final 数据

许多编程语言都有某种方法告诉编译器有一块数据是恒定不变的。恒定是有用的，如：

1. 一个永不改变的编译时常量。
2. 一个在运行时初始化就不会改变的值。

对于编译时常量这种情况，编译器可以把常量带入计算中；也就是说，可以在编译时计算，减少了一些运行时的负担。在 Java 中，这类常量必须是基本类型，而且用关键字 **final** 修饰。你必须在定义常量的时候进行赋值。

一个被 **static** 和 **final** 同时修饰的属性只会占用一段不能改变的存储空间。

当用 **final** 修饰对象引用而非基本类型时，其含义会有一点令人困惑。对于基本类型，**final** 使数值恒定不变，而对于对象引用，**final** 使引用恒定不变。一旦引用被初始化指向了某个对象，它就不能改为指向其他对象。但是，对象本身是可以修改的，Java 没有提供将任意对象设为常量的方法。（你可以自己编写类达到使对象恒定不变的效果）这一限制同样适用数组，数组也是对象。

下面例子展示了 **final** 属性的使用：

```
// reuse/FinalData.java
// The effect of final on fields
import java.util.*;

class Value {
    int i; // package access

    Value(int i) {
        this.i = i;
    }
}

```

```java
public class FinalData {
    private static Random rand = new Random(47);
    private String id;

    public FinalData(String id) {
        this.id = id;
    }
    // Can be compile-time constants:
    private final int valueOne = 9;
    private static final int VALUE_TWO = 99;
    // Typical public constant:
    public static final int VALUE_THREE = 39;
    // Cannot be compile-time constants:
    private final int i4 = rand.nextInt(20);
    static final int INT_5 = rand.nextInt(20);
    private Value v1 = new Value(11);
    private final Value v2 = new Value(22);
    private static final Value VAL_3 = new Value(33);
    // Arrays:
    private final int[] a = {1, 2, 3, 4, 5, 6};

    @Override
    public String toString() {
        return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
    }

    public static void main(String[] args) {
        FinalData fd1 = new FinalData("fd1");
        //- fd1.valueOne++; // Error: can't change value
        fd1.v2.i++; // Object isn't constant
        fd1.v1 = new Value(9); // OK -- not final
        for (int i = 0; i < fd1.a.length; i++) {
            fd1.a[i]++; // Object isn't constant
        }
        //- fd1.v2 = new Value(0); // Error: Can't
        //- fd1.VAL_3 = new Value(1); // change reference
        //- fd1.a = new int[3];
        System.out.println(fd1);
        System.out.println("Creating new FinalData");
        FinalData fd2 = new FinalData("fd2");
        System.out.println(fd1);
        System.out.println(fd2);
    }
}
```

输出:

```
fd1: i4 = 15, INT_5 = 18
Creating new FinalData
fd1: i4 = 15, INT_5 = 18
fd2: i4 = 13, INT_5 = 18
```

因为 **valueOne** 和 **VALUE_TWO** 都是带有编译时值的 **final** 基本类型，它们都可用作编译时常量，没有多大区别。**VALUE_THREE** 是一种更加典型的常量定义的方式：**public** 意味着可以在包外访问，**static** 强调只有一个，**final** 说明是一个常量。

按照惯例，带有恒定初始值的 **final static** 基本变量（即编译时常量）命名全部使用大写，单词之间用下划线分隔。（源于 C 语言中定义常量的方式。）

我们不能因为某数据被 **final** 修饰就认为在编译时可以知道它的值。由上例中的 **i4** 和 **INT_5** 可以看出，它们在运行时才会赋值随机数。示例部分也展示了将 **final** 值定义为 **static** 和非 **static** 的区别。此区别只有当值在运行时被初始化时才会显现，因为编译器对编译时数值一视同仁。（而且编译时数值可能因优化而消失。）当运行程序时就能看到这个区别。注意到 **fd1** 和 **fd2** 的 **i4** 值不同，但 **INT_5** 的值并没有因为创建了第二个 **FinalData** 对象而改变，这是因为它是 **static** 的，在加载时已经被初始化，并不是每次创建新对象时都初始化。

v1 到 **VAL_3** 变量说明了 **final** 引用的意义。正如你在 `main()` 中所见，**v2** 是 **final** 的并不意味着你不能修改它的值。因为它是引用，所以只是说明它不能指向一个新的对象。这对于数组具有同样的意义，数组只不过是另一种引用。（我不知道有什么方法能使数组引用本身成为 **final**。）看起来，声明引用为 **final** 没有声明基本类型 **final** 有用。

9.8.2 空白 final

空白 final 指的是没有初始化值的 **final** 属性。编译器确保空白 final 在使用前必须被初始化。这样既能使一个类的每个对象的 **final** 属性值不同，也能保持它的不变性。

```
// reuse/BlankFinal.java
// "Blank" final fields
class Poppet {
    private int i;

    Poppet(int ii) {
        i = ii;
    }
}

public class BlankFinal {
    private final int i = 0; // Initialized final
    private final int j; // Blank final
    private final Poppet p; // Blank final reference
    // Blank finals MUST be initialized in constructor
    public BlankFinal() {
        j = 1; // Initialize blank final
        p = new Poppet(1); // Init blank final reference
    }

```

```
    public BlankFinal(int x) {
        j = x; // Initialize blank final
        p = new Poppet(x); // Init blank final reference
    }

    public static void main(String[] args) {
        new BlankFinal();
        new BlankFinal(47);
    }
}
```

你必须在定义时或在每个构造器中执行 final 变量的赋值操作。这保证了 final 属性在使用前已经被初始化过。

9.8.3 final 参数

在参数列表中，将参数声明为 final 意味着在方法中不能改变参数指向的对象或基本变量：

```
// reuse/FinalArguments.java
// Using "final" with method arguments
class Gizmo {
    public void spin() {

    }
}

public class FinalArguments {
    void with(final Gizmo g) {
        //-g = new Gizmo(); // Illegal -- g is final
    }

    void without(Gizmo g) {
        g = new Gizmo(); // OK -- g is not final
        g.spin();
    }

    //void f(final int i) { i++; } // Can't change
    // You can only read from a final primitive
    int g(final int i) {
        return i + 1;
    }

    public static void main(String[] args) {
        FinalArguments bf = new FinalArguments();
        bf.without(null);
        bf.with(null);
    }
}
```

方法 `f()` 和 `g()` 展示了 **final** 基本类型参数的使用情况。你只能读取而不能修改参数。这个特性主要用于传递数据给匿名内部类。这将在" 内部类 "章节中详解。

9.8.4 final 方法

使用 **final** 方法的原因有两个。第一个原因是给方法上锁，防止子类通过覆写改变方法的行为。这是出于继承的考虑，确保方法的行为不会因继承而改变。

过去建议使用 **final** 方法的第二个原因是效率。在早期的 Java 实现中，如果将一个方法指明为 **final**，就是同意编译器把对该方法的调用转化为内嵌调用。当编译器遇到 **final** 方法的调用时，就会很小心地跳过普通的插入代码以执行方法的调用机制（将参数压栈，跳至方法代码处执行，然后跳回并清理栈中的参数，最终处理返回值），而用方法体内实际代码的副本替代方法调用。这消除了方法调用的开销。但是如果一个方法很大代码膨胀，你也许就看不到内嵌带来的性能提升，因为内嵌调用带来的性能提高被花费在方法里的时间抵消了。

在最近的 Java 版本中，虚拟机可以探测到这些情况（尤其是 *hotspot* 技术），并优化去掉这些效率反而降低的内嵌调用方法。有很长一段时间，使用 **final** 来提高效率都被阻止。你应该让编译器和 JVM 处理性能问题，只有在为了明确禁止覆写方法时才使用 **final**。

9.8.5 final 和 private

类中所有的 **private** 方法都隐式地指定为 **final**。因为不能访问 **private** 方法，所以不能覆写它。可以给 **private** 方法添加 **final** 修饰，但是并不能给方法带来额外的含义。

以下情况会令人困惑，当你试图覆写一个 **private** 方法（隐式是 **final** 的）时，看上去奏效，而且编译器不会给出错误信息：

```java
// reuse/FinalOverridingIllusion.java
// It only looks like you can override
// a private or private final method
class WithFinals {
    // Identical to "private" alone:
    private final void f() {
        System.out.println("WithFinals.f()");
    }
    // Also automatically "final":
    private void g() {
        System.out.println("WithFinals.g()");
    }
}

class OverridingPrivate extends WithFinals {
    private final void f() {
        System.out.println("OverridingPrivate.f()");
    }
```

```java

    private void g() {
        System.out.println("OverridingPrivate.g()");
    }
}

class OverridingPrivate2 extends OverridingPrivate {
    public final void f() {
        System.out.println("OverridingPrivate2.f()");
    }

    public void g() {
        System.out.println("OverridingPrivate2.g()");
    }
}

public class FinalOverridingIllusion {
    public static void main(String[] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // You can upcast:
        OverridingPrivate op = op2;
        // But you can't call the methods:
        //- op.f();
        //- op.g();
        // Same here:
        WithFinals wf = op2;
        //- wf.f();
        //- wf.g();
    }
}
```

输出:

```
OverridingPrivate2.f()
OverridingPrivate2.g()
```

“覆写”只发生在方法是基类的接口时。也就是说，必须能将一个对象向上转型为基类并调用相同的方法（这一点在下一章阐明）。如果一个方法是 **private** 的，它就不是基类接口的一部分。它只是隐藏在类内部的代码，且恰好有相同的命名而已。但是如果你在派生类中以相同的命名创建了 **public**, **protected** 或包访问权限的方法，这些方法与基类中的方法没有联系，你没有覆写方法，只是在创建新的方法而已。由于 **private** 方法无法触及且能有效隐藏，除了把它看作类中的一部分，其他任何事物都不需要考虑到它。

9.8.6 final 类

当说一个类是 **final** （**final** 关键字在类定义之前），就意味着它不能被继承。之所以这么做，是因为类的设计就是永远不需要改动，或者是出于安全考虑不希望它有子类。

```
// reuse/Jurassic.java
// Making an entire class final
class SmallBrain {}

final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();

    void f() {}
}

//- class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'
public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
}
```

final 类的属性可以根据个人选择是或不是 **final**。这同样适用于不管类是否是 **final** 的内部 **final** 属性。然而，由于 **final** 类禁止继承，类中所有的方法都被隐式地指定为 **final**，所以没有办法覆写它们。你可以在 final 类中的方法加上 **final** 修饰符，但不会增加任何意义。

9.8.7 final 忠告

在设计类时将一个方法指明为 final 看上去是明智的。你可能会觉得没人会覆写那个方法。有时这是对的。

但请留意你的假设。通常来说，预见一个类如何被复用是很困难的，特别是通用类。如果将一个方法指定为 **final**，可能会防止其他程序员的项目中通过继承来复用你的类，而这仅仅是因为你没有想到它被以那种方式使用。

Java 标准类库就是一个很好的例子。尤其是 Java 1.0/1.1 的 **Vector** 类被广泛地使用，而且从效率考虑（这近乎是个幻想），如果它的所有方法没有被指定为 **final**，可能会更加有用。很容易想到，你可能会继承并覆写这么一个基础类，但是设计者们认为这么做不合适。有两个讽刺的原因。第一，**Stack** 继承自 **Vector**，就是说 **Stack** 是个 **Vector**，但从逻辑上来说不对。尽管如此，Java 设计者们仍然这么做，在用这种方式创建 **Stack** 时，他们应该意识到了 **final** 方法过于约束。

要加速一个 IO 密集型程序，请首先考虑所有的 IO，消除那些不必要的或冗余的 IO，并使余下的部分尽可能地块。 —David Martin

第二，**Vector** 中的很多重要方法，比如 `addElement()` 和 `elementAt()` 方法都是同步的。在“并发编程”一章中会看同步会导致很大的执行开销，可能会抹煞 **final** 带来的好处。这加强了程序员永远无法正确猜到优化应该发生在何处的观点。如此笨拙的设计却出现在每个人都要使用的标准库中，太糟糕了。庆幸的是，现代 Java 容器用 **ArrayList** 代替了 **Vector**，它的行为要合理得多。不幸的是，仍然有很多新代码使用旧的集合类库，其中就包括 **Vector**。

Java 1.0/1.1 标准类库中另一个重要的类是 **Hashtable**（后来被 **HashMap** 取代），它不含任何 **final** 方法。本书中其他地方也提到，很明显不同的类是由不同的人设计的。**Hashtable** 就比 **Vector** 中的方法名简洁得多，这又是一条证据。对于类库的使用者来说，这是一个本不应该如此草率的事情。这种不规则的情况造成用户需要做更多的工作——这是对粗糙的设计和代码的又一讽刺。

9.9 类初始化和加载

在许多传统语言中，程序在启动时一次性全部加载。接着初始化，然后程序开始运行。必须仔细控制这些语言的初始化过程，以确保 **statics** 初始化的顺序不会造成麻烦。在 C++ 中，如果一个 **static** 期望使用另一个 **static**，而另一个 **static** 还没有初始化，就会出现问题。

Java 中不存在这样的问题，因为它采用了一种不同的方式加载。因为 Java 中万物皆对象，所以加载活动就容易得多。记住每个类的编译代码都存在于它自己独立的文件中。该文件只有在使用程序代码时才会被加载。一般可以说“类的代码在首次使用时加载“。这通常是指创建类的第一个对象，或者是访问了类的 **static** 属性或方法。构造器也是一个 **static** 方法尽管它的 **static** 关键字是隐式的。因此，准确地说，一个类当它任意一个 **static** 成员被访问时，就会被加载。

首次使用时就是 **static** 初始化发生时。所有的 **static** 对象和 **static** 代码块在加载时按照文本的顺序（在类中定义的顺序）依次初始化。**static** 变量只被初始化一次。

9.9.1 继承和初始化

了解包括继承在内的整个初始化过程是有帮助的，这样可以对所发生的一切有全局性的把握。考虑下面的例子：

```
// reuse/Beetle.java
// The full process of initialization
class Insect {
    private int i = 9;
    protected int j;

    Insect() {
        System.out.println("i = " + i + ", j = " + j);
        j = 39;
    }

```

最快的 IO 就是不 IO。 — Nils-Peter Nelson 那些最便宜、最快而且可靠性最高的计算机组件压根就不存在。 — Gordon Bell

```
    private static int x1 = printInit("static Insect.x1 initialized");

    static int printInit(String s) {
        System.out.println(s);
        return 47;
    }
}

public class Beetle extends Insect {
    private int k = printInit("Beetle.k.initialized");

    public Beetle() {
        System.out.println("k = " + k);
        System.out.println("j = " + j);
    }

    private static int x2 = printInit("static Beetle.x2 initialized");

    public static void main(String[] args) {
        System.out.println("Beetle constructor");
        Beetle b = new Beetle();
    }
}
```

输出:

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
```

当执行 **java Beetle**，首先会试图访问 **Beetle** 类的 `main()` 方法（一个静态方法），加载器启动并找出 **Beetle** 类的编译代码（在名为 **Beetle.class** 的文件中）。在加载过程中，编译器注意到有一个基类，于是继续加载基类。不论是否创建了基类的对象，基类都会被加载。（可以尝试把创建基类对象的代码注释掉证明这点。）

如果基类还存在自身的基类，那么第二个基类也将被加载，以此类推。接下来，根基类（例子中根基类是 **Insect**）的 **static** 的初始化开始执行，接着是派生类，以此类推。这点很重要，因为派生类中 **static** 的初始化可能依赖基类成员是否被正确地初始化。

至此，必要的类都加载完毕，可以创建对象了。首先，对象中的所有基本类型变量都被置为默认值，对象引用被设为 **null** —— 这是通过将对象内存设为二进制零值一举生成的。接着会调用基类的构造器。本例中是自动调用的，但是你也可以使用 **super** 调用指定的基类构造器（在 **Beetle** 构造器中的第一步操作）。基类构造器和派生类构造器一样以相同的顺序经历相同的过程。当基类构造器完成后，实例变量按文本顺序初始化。最终，构造器的剩余部分被执行。

9.10 本章小结

继承和组合都是从已有类型创建新类型。组合将已有类型作为新类型底层实现的一部分，继承复用的是接口。

使用继承时，派生类具有基类接口，因此可以向上转型为基类，这对于多态至关重要，在下一章你将看到。

尽管在面向对象编程时极力强调继承，但在开始设计时，优先使用组合（或委托），只有当确实需要时再使用继承。组合更具灵活性。另外，通过对成员类型使用继承的技巧，可以在运行时改变成员的类型和行为。因此，可以在运行时改变组合对象的行为。

在设计一个系统时，目标是发现或创建一系列类，每个类有特定的用途，而且既不应太大（包括太多功能难以复用），也不应太小（不添加其他功能就无法使用）。如果设计变得过于复杂，通过将现有类拆分为更小的部分而添加更多的对象，通常是有帮助的。

当开始设计一个系统时，记住程序开发是一个增量过程，正如人类学习。它依赖实验，你可以尽可能多做分析，然而在项目开始时仍然无法知道所有的答案。如果把项目视作一个有机的，进化着的生命去培养，而不是视为像摩天大楼一样快速见效，就能获得更多的成功和更迅速的反馈。继承和组合正是可以让你执行如此实验的面向对象编程中最基本的两个工具。

10 多态

> 曾经有人请教我 “ Babbage 先生，如果输入错误的数字到机器中，会得出正确结果吗？ ” 我无法理解产生如此问题的概念上的困惑。—— Charles Babbage (1791 - 1871)

多态是面向对象编程语言中，继数据抽象和继承之外的第三个重要特性。

多态提供了另一个维度的接口与实现分离，以解耦做什么和怎么做。多态不仅能改善代码的组织，提高代码的可读性，而且能创建有扩展性的程序——无论在最初创建项目时还是在添加新特性时都可以“生长”的程序。

封装通过合并特征和行为来创建新的数据类型。隐藏实现通过将细节**私有化**把接口与实现分离。这种类型的组织机制对于有面向过程编程背景的人来说，更容易理解。而多态是消除类型之间的耦合。在上一章中，继承允许把一个对象视为它本身的类型或它的基类类型。这样就能把很多派生自一个基类的类型当作同一类型处理，因而一段代码就可以无差别地运行在所有不同的类型上了。多态方法调用允许一种类型表现出与相似类型的区别，只要这些类型派生自一个基类。这种区别是当你通过基类调用时，由方法的不同行为表现出来的。

在本章中，通过一些基本、简单的例子（这些例子中只保留程序中与多态有关的行为），你将逐步学习多态（也称为*动态绑定*或*后期绑定*或*运行时绑定*）。

10.1 向上转型回顾

在上一章中，你看到了如何把一个对象视作它的自身类型或它的基类类型。这种把一个对象引用当作它的基类引用的做法称为向上转型，因为继承图中基类一般都位于最上方。

同样你也在下面的音乐乐器例子中发现了问题。即然几个例子都要演奏乐符（**Note**），首先我们先在包中单独创建一个 Note 枚举类：

```
// polymorphism/music/Note.java
// Notes to play on musical instruments
package polymorphism.music;

public enum Note {
    MIDDLE_C, C_SHARP, B_FLAT; // Etc.
}
```

The First Rule of Program Optimization. Don’t do it. The Second Rule of Program Optimization - For experts only. Don’t do it yet. — Michael Jackson

枚举已经在" 第 6 章初始化和清理 "一章中介绍过了。

这里，**Wind** 是一种 **Instrument**；因此，**Wind** 继承 **Instrument**：

```
// polymorphism/music/Instrument.java
package polymorphism.music;

class Instrument {
    public void play(Note n) {
        System.out.println("Instrument.play()");
    }
}

// polymorphism/music/Wind.java
package polymorphism.music;
// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
    // Redefine interface method:
    @Override
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
}
```

Music 的方法 `tune()` 接受一个 **Instrument** 引用，同时也接受任何派生自 **Instrument** 的类引用：

```
// polymorphism/music/Music.java
// Inheritance & upcasting
// {java polymorphism.music.Music}
package polymorphism.music;

public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }

    public static void main(String[] args) {
        Wind flute = new Wind();
        tune(flute); // Upcasting
    }
}
```

输出：

```
Wind.play() MIDDLE_C
```

在 `main()` 中你看到了 `tune()` 方法传入了一个 **Wind** 引用，而没有做类型转换。这样做是允许的—— **Instrument** 的接口一定存在于 **Wind** 中，因此 **Wind** 继承了 **Instrument**。从 **Wind** 向上

The fastest algorithm can frequently be replaced by one that is almost as fast and much easier to understand. — Douglas W. Jones

转型为 **Instrument** 可能“缩小”接口，但不会比 **Instrument** 的全部接口更少。

10.1.1 忘掉对象类型

Music.java 看起来似乎有点奇怪。为什么所有人都故意忘记掉对象类型呢？当向上转型时，就会发生这种情况，而且看起来如果 `tune()` 接受的参数是一个 **Wind** 引用会更为直观。这会带来一个重要问题：如果你那么做，就要为系统内 **Instrument** 的每种类型都编写一个新的 `tune()` 方法。假设按照这种推理，再增加 **Stringed** 和 **Brass** 这两种 **Instrument**：

```java
// polymorphism/music/Music2.java
// Overloading instead of upcasting
// {java polymorphism.music.Music2}
package polymorphism.music;

class Stringed extends Instrument {
    @Override
    public void play(Note n) {
        System.out.println("Stringed.play() " + n);
    }
}

class Brass extends Instrument {
    @Override
    public void play(Note n) {
        System.out.println("Brass.play() " + n);
    }
}

public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }

    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }

    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }

    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute); // No upcasting
        tune(violin);
        tune(frenchHorn);
```

On some machines indirection is slower with displacement, so the most-used member of a structure or a record should be first. — Mike Morton

```
    }
}
```

输出:

```
Wind.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
```

这样行得通，但是有一个主要缺点：必须为添加的每个新 **Instrument** 类编写特定的方法。这意味着开始时就需要更多的编程，而且以后如果添加类似 `tune()` 的新方法或 **Instrument** 的新类型时，还有大量的工作要做。考虑到如果你忘记重载某个方法，编译器也不会提示你，这会造成类型的整个处理过程变得难以管理。

如果只写一个方法以基类作为参数，而不用管是哪个具体派生类，这样会变得更好吗？也就是说，如果忘掉派生类，编写的代码只与基类打交道，会不会更好呢？

这正是多态所允许的。但是大部分拥有面向过程编程背景的程序员会对多态的运作方式感到一些困惑。

10.2 转机

运行程序后会看到 **Music.java** 的难点。**Wind.play()** 的输出结果正是我们期望的，然而它看起来似乎不应该得出这样的结果。观察 `tune()` 方法:

```
public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
}
```

它接受一个 **Instrument** 引用。那么编译器是如何知道这里的 **Instrument** 引用指向的是 **Wind**，而不是 **Brass** 或 **Stringed** 呢？编译器无法得知。为了深入理解这个问题，有必要研究一下*绑定*这个主题。

10.2.1 方法调用绑定

将一个方法调用和一个方法主体关联起来称作*绑定*。若绑定发生在程序运行前（如果有的话，由编译器和链接器实现），叫做*前期绑定*。你可能从来没有听说这个术语，因为它是面向过程语言不需选择默认的绑定方式，例如在 C 语言中就只有*前期绑定*这一种方法调用。

上述程序让人困惑的地方就在于前期绑定，因为编译器只知道一个 **Instrument** 引用，它无法得知究竟会调用哪个方法。

解决方法就是*后期绑定*，意味着在运行时根据对象的类型进行绑定。后期绑定也称为*动态绑定*或*运*

In non-I/O-bound programs, less than four per cent of a program generally accounts for more than half of its running time. — Don Knuth

行时绑定。当一种语言实现了后期绑定，就必须具有某种机制在运行时能判断对象的类型，从而调用恰当的方法。也就是说，编译器仍然不知道对象的类型，但是方法调用机制能找到正确的方法体并调用。每种语言的后期绑定机制都不同，但是可以想到，对象中一定存在某种类型信息。

Java 中除了 **static** 和 **final** 方法（**private** 方法也是隐式的 **final**）外，其他所有方法都是后期绑定。这意味着通常情况下，我们不需要判断后期绑定是否会发生——它自动发生。

为什么将一个对象指明为 **final**？正如前一章所述，它可以防止方法被重写。但更重要的一点可能是，它有效地”关闭了“动态绑定，或者说告诉编译器不需要对其进行动态绑定。这可以让编译器为 **final** 方法生成更高效的代码。然而，大部分情况下这样做不会对程序的整体性能带来什么改变，因此最好是为了设计使用 **final**，而不是为了提升性能而使用。

10.2.2 产生正确的行为

一旦当你知道 Java 中所有方法都是通过后期绑定来实现多态时，就可以编写只与基类打交道的代码，而且代码对于派生类来说都能正常地工作。或者换种说法，你向对象发送一条消息，让对象自己做正确的事。

面向对象编程中的经典例子是形状 **Shape**。这个例子很直观，但不幸的是，它可能让初学者困惑，认为面向对象编程只适合图形化程序设计，实际上不是这样。

形状的例子中，有一个基类称为 **Shape**，多个不同的派生类型分别是：**Circle**, **Square**, **Triangle** 等等。这个例子之所以好用，是因为我们可以直接说“圆 (Circle) 是一种形状 (Shape)”，这很容易理解。继承图展示了它们之间的关系：

Figure 10.1: 形状继承图

向上转型就像下面这么简单：

```
Shape s = new Circle();
```

这会创建一个 **Circle** 对象，引用被赋值给 **Shape** 类型的变量 s，这看似错误（将一种类型赋值给另一种类型），然而是没问题的，因此从继承上可认为圆 (Circle) 就是一个形状 (Shape)。因此编译器认可了赋值语句，没有报错。

假设你调用了一个基类方法（在各个派生类中都被重写）：

```
s.draw()
```

你可能再次认为 **Shape** 的 `draw()` 方法被调用，因为 s 是一个 **Shape** 引用——编译器怎么可能知道要做其他的事呢？然而，由于后期绑定（多态）被调用的是 **Circle** 的 `draw()` 方法，这是正确的。

下面的例子稍微有些不同。首先让我们创建一个可复用的 **Shape** 类库，基类 **Shape** 为它的所有子类建立了公共接口——所有的形状都可以被绘画和擦除：

```
// polymorphism/shape/Shape.java
package polymorphism.shape;

public class Shape {
    public void draw() {}
    public void erase() {}
}
```

派生类通过重写这些方法为每个具体的形状提供独一无二的方法行为：

```
// polymorphism/shape/Circle.java
package polymorphism.shape;

public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Circle.draw()");
    }
    @Override
    public void erase() {
        System.out.println("Circle.erase()");
    }
}

// polymorphism/shape/Square.java
package polymorphism.shape;

public class Square extends Shape {
    @Override
    public void draw() {
        System.out.println("Square.draw()");
    }
    @Override
    public void erase() {
        System.out.println("Square.erase()");
    }
```

Conservation of Code Size When you turn an ordinary page of code into just a handful of instructions for speed, expand the comments to keep the number of source lines constant. — Mike Morton

```
 }

// polymorphism/shape/Triangle.java
package polymorphism.shape;

public class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Triangle.draw()");
    }
    @Override
    public void erase() {
        System.out.println("Triangle.erase()");
    }
}
```

RandomShapes 是一种工厂，每当我们调用 `get()` 方法时，就会产生一个指向随机创建的 **Shape** 对象的引用。注意，向上转型发生在 **return** 语句中，每条 **return** 语句取得一个指向某个 **Circle**，**Square** 或 **Triangle** 的引用，并将其以 **Shape** 类型从 `get()` 方法发送出去。因此无论何时调用 `get()` 方法，你都无法知道具体的类型是什么，因为你总是得到一个简单的 **Shape** 引用：

```
// polymorphism/shape/RandomShapes.java
// A "factory" that randomly creates shapes
package polymorphism.shape;
import java.util.*;

public class RandomShapes {
    private Random rand = new Random(47);

    public Shape get() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
            case 2: return new Triangle();
        }
    }

    public Shape[] array(int sz) {
        Shape[] shapes = new Shape[sz];
        // Fill up the array with shapes:
        for (int i = 0; i < shapes.length; i++) {
            shapes[i] = get();
        }
        return shapes;
    }
}
```

`array()` 方法分配并填充了 **Shape** 数组，这里使用了 for-in 表达式：

If the programmer can simulate a construct faster than the compiler can implement the construct itself! then the compiler writer has blown it badly. — Guy L. Steele, Jr.

```java
// polymorphism/Shapes.java
// Polymorphism in Java
import polymorphism.shape.*;

public class Shapes {
    public static void main(String[] args) {
        RandomShapes gen = new RandomShapes();
        // Make polymorphic method calls:
        for (Shape shape: gen.array(9)) {
            shape.draw();
        }
    }
}
```

输出:

```
Triangle.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Circle.draw()
```

`main()` 方法中包含了一个 **Shape** 引用组成的数组，其中每个元素通过调用 **RandomShapes** 类的 `get()` 方法生成。现在你只知道拥有一些形状，但除此之外一无所知（编译器也是如此）。然而当遍历这个数组为每个元素调用 `draw()` 方法时，从运行程序的结果中可以看到，与类型有关的特定行为奇迹般地发生了。

随机生成形状是为了让大家理解：在编译时，编译器不需要知道任何具体信息以进行正确的调用。所有对方法 `draw()` 的调用都是通过动态绑定进行的。

10.2.3 可扩展性

现在让我们回头看音乐乐器的例子。由于多态机制，你可以向系统中添加任意多的新类型，而不需要修改 `tune()` 方法。在一个设计良好的面向对象程序中，许多方法将会遵循 `tune()` 的模型，只与基类接口通信。这样的程序是可扩展的，因为可以从通用的基类派生出新的数据类型，从而添加新的功能。那些操纵基类接口的方法不需要改动就可以应用于新类。

考虑一下乐器的例子，如果在基类中添加更多的方法，并加入一些新类，将会发生什么呢:

To speed up an I/O-bound program, begin by accounting for all I/O. Eliminate that which is unnecessary or redundant, and make the remaining as fast as possible. — David Martin

Figure 10.2: 乐器继承图

所有的新类都可以和原有类正常运行，不需要改动 `tune()` 方法。即使 `tune()` 方法单独存放在某个文件中，而且向 **Instrument** 接口中添加了新的方法，`tune()` 方法也无需再编译就能正确运行。下面是类图的实现：

```
// polymorphism/music3/Music3.java
// An extensible program
// {java polymorphism.music3.Music3}
package polymorphism.music3;
import polymorphism.music.Note;

class Instrument {
    void play(Note n) {
        System.out.println("Instrument.play() " + n);
    }

    String what() {
        return "Instrument";
    }

    void adjust() {
        System.out.println("Adjusting Instrument");
    }
}

```

```
class Wind extends Instrument {
    @Override
    void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
    @Override
    String what() {
        return "Wind";
    }
    @Override
    void adjust() {
        System.out.println("Adjusting Wind");
    }
}

class Percussion extends Instrument {
    @Override
    void play(Note n) {
        System.out.println("Percussion.play() " + n);
    }
    @Override
    String what() {
        return "Percussion";
    }
    @Override
    void adjust() {
        System.out.println("Adjusting Percussion");
    }
}

class Stringed extends Instrument {
    @Override
    void play(Note n) {
        System.out.println("Stringed.play() " + n);
    }
    @Override
    String what() {
        return "Stringed";
    }
    @Override
    void adjust() {
        System.out.println("Adjusting Stringed");
    }
}

class Brass extends Wind {
    @Override
    void play(Note n) {
        System.out.println("Brass.play() " + n);
    }
    @Override
```

```
    void adjust() {
        System.out.println("Adjusting Brass");
    }
}

class Woodwind extends Wind {
    @Override
    void play(Note n) {
        System.out.println("Woodwind.play() " + n);
    }
    @Override
    String what() {
        return "Woodwind";
    }
}

public class Music3 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }

    public static void tuneAll(Instrument[] e) {
        for (Instrument i: e) {
            tune(i);
        }
    }

    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
}
```

输出:

```
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
```

新方法 `what()` 返回一个带有类描述的 **String** 引用，`adjust()` 提供一些乐器调音的方法。

在 `main()` 方法中，当向 **orchestra** 数组添加元素时，元素会自动向上转型为 **Instrument**。

`tune()` 方法可以忽略周围所有代码发生的变化，仍然可以正常运行。这正是我们期待多态能提供的特性。代码中的修改不会破坏程序中其他不应受到影响的部分。换句话说，多态是一项“将改变的事物与不变的事物分离”的重要技术。

10.2.4 陷阱：“重写”私有方法

你可能天真地试图像下面这样做：

```
// polymorphism/PrivateOverride.java
// Trying to override a private method
// {java polymorphism.PrivateOverride}
package polymorphism;

public class PrivateOverride {
    private void f() {
        System.out.println("private f()");
    }

    public static void main(String[] args) {
        PrivateOverride po = new Derived();
        po.f();
    }
}

public Derived extends PrivateOverride {
    public void f() {
        System.out.println("public f()");
    }
}
```

输出：

```
private f()
```

你可能期望输出是 **public f()**，然而 **private** 方法可以当作是 **final** 的，对于派生类来说是隐蔽的。因此，这里 **Derived** 的 `f()` 是一个全新的方法；因为基类版本的 `f()` 屏蔽了 **Derived**，因此它都不算是重写方法。

结论是只有非 **private** 方法才能被重写，但是得小心重写 **private** 方法的现象，编译器不报错，但不会按我们所预期的执行。为了清晰起见，派生类中的方法名采用与基类中 **private** 方法名不同的命名。

如果使用了 `@Override` 注解，就能检测出问题：

```
// polymorphism/PrivateOverride2.java
```

```
// Detecting a mistaken override using @Override
// {WillNotCompile}
package polymorphism;

public class PrivateOverride2 {
    private void f() {
        System.out.println("private f()");
    }

    public static void main(String[] args) {
        PrivateOverride2 po = new Derived2();
        po.f();
    }
}

class Derived2 extends PrivateOverride2 {
    @Override
    public void f() {
        System.out.println("public f()");
    }
}
```

编译器报错信息是:

```
error: method does not override or
implement a method from a supertype
```

10.2.5 陷阱：属性与静态方法

一旦学会了多态，就可以以多态的思维方式考虑每件事。然而，只有普通的方法调用可以是多态的。例如，如果你直接访问一个属性，该访问会在编译时解析:

```
// polymorphism/FieldAccess.java
// Direct field access is determined at compile time
class Super {
    public int field = 0;

    public int getField() {
        return field;
    }
}

class Sub extends Super {
    public int field = 1;

    @Override
    public int getField() {
        return field;
    }
```

```java

    public int getSuperField() {
        return super.field;
    }
}

public class FieldAccess {
    public static void main(String[] args) {
        Super sup = new Sub(); // Upcast
        System.out.println("sup.field = " + sup.field +
                           ", sup.getField() = " + sup.getField());
        Sub sub = new Sub();
        System.out.println("sub.field = " + sub.field +
                           ", sub.getField() = " + sub.getField()
                           + ", sub.getSuperField() = " + sub.
                               getSuperField())
    }
}
```

输出:

```
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
```

当 **Sub** 对象向上转型为 **Super** 引用时，任何属性访问都被编译器解析，因此不是多态的。在这个例子中，**Super.field** 和 **Sub.field** 被分配了不同的存储空间，因此，**Sub** 实际上包含了两个称为 **field** 的属性：它自己的和来自 **Super** 的。然而，在引用 **Sub** 的 **field** 时，默认的 **field** 属性并不是 **Super** 版本的 **field** 属性。为了获取 **Super** 的 **field** 属性，需要显式地指明 **super.field**。

尽管这看起来是个令人困惑的问题，实际上基本不会发生。首先，通常会将所有的属性都指明为 **private**，因此不能直接访问它们，只能通过方法来访问。此外，你可能也不会给基类属性和派生类属性起相同的名字，这样做会令人困惑。

如果一个方法是静态 (**static**) 的，它的行为就不具有多态性:

```java
// polymorphism/StaticPolymorphism.java
// static methods are not polymorphic
class StaticSuper {
    public static String staticGet() {
        return "Base staticGet()";
    }

    public String dynamicGet() {
        return "Base dynamicGet()";
    }
}

class StaticSub extends StaticSuper {
    public static String staticGet() {
        return "Derived staticGet()";
```

```
    }
    @Override
    public String dynamicGet() {
        return "Derived dynamicGet()";
    }
}

public class StaticPolymorphism {
    public static void main(String[] args) {
        StaticSuper sup = new StaticSub(); // Upcast
        System.out.println(StaticSuper.staticGet());
        System.out.println(sup.dynamicGet());
    }
}
```

输出:

```
Base staticGet()
Derived dynamicGet()
```

静态的方法只与类关联，与单个的对象无关。

10.3 构造器和多态

通常，构造器不同于其他类型的方法。在涉及多态时也是如此。尽管构造器不具有多态性（事实上人们会把它看作是隐式声明的静态方法），但是理解构造器在复杂层次结构中运作多态还是非常重要的。这个理解可以帮助你避免一些不愉快的困扰。

10.3.1 构造器调用顺序

在“初始化和清理”和“复用”两章中已经简单地介绍过构造器的调用顺序，但那时还没有介绍多态。

在派生类的构造过程中总会调用基类的构造器。初始化会自动按继承层次结构上移，因此每个基类的构造器都会被调用到。这么做是有意义的，因为构造器有着特殊的任务：检查对象是否被正确地构造。由于属性通常声明为 **private**，你必须假定派生类只能访问自己的成员而不能访问基类的成员。只有基类的构造器拥有恰当的知识和权限来初始化自身的元素。因此，必须得调用所有构造器；否则就不能构造完整的对象。这就是编译器强制每个派生类部分必须调用构造器的原因。如果在派生类的构造器主体中没有显式地调用基类构造器，编译器就会默默地调用无参构造器。如果没有无参构造器，编译器就会报错（当类中不含构造器时，编译器会自动合成一个无参构造器）。

下面的例子展示了组合、继承和多态在构建顺序上的作用:

```
// polymorphism/Sandwich.java
```

```java
// Order of constructor calls
// {java polymorphism.Sandwich}
package polymorphism;

class Meal {
    Meal() {
        System.out.println("Meal()");
    }
}

class Bread {
    Bread() {
        System.out.println("Bread()");
    }
}

class Cheese {
    Cheese() {
        System.out.println("Cheese()");
    }
}

class Lettuce {
    Lettuce() {
        System.out.println("Lettuce()");
    }
}

class Lunch extends Meal {
    Lunch() {
        System.out.println("Lunch()");
    }
}

class PortableLunch extends Lunch {
    PortableLunch() {
        System.out.println("PortableLunch()");
    }
}

public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();

    public Sandwich() {
        System.out.println("Sandwich()");
    }

    public static void main(String[] args) {
        new Sandwich();
```

```
    }
}
```

输出:

```
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

这个例子用其他类创建了一个复杂的类。每个类都在构造器中声明自己。重要的类是 **Sandwich**，它反映了三层继承（如果算上 **Object** 的话，就是四层），包含了三个成员对象。

从创建 **Sandwich** 对象的输出中可以看出对象的构造器调用顺序如下:

1. 基类构造器被调用。这个步骤被递归地重复，这样一来类层次的顶级父类会被最先构造，然后是它的派生类，以此类推，直到最底层的派生类。
2. 按声明顺序初始化成员。
3. 调用派生类构造器的方法体。

构造器的调用顺序很重要。当使用继承时，就已经知道了基类的一切，并可以访问基类中任意 **public** 和 **protected** 的成员。这意味着在派生类中可以假定所有的基类成员都是有效的。在一个标准方法中，构造动作已经发生过，对象其他部分的所有成员都已经创建好。

在构造器中必须确保所有的成员都已经构建完。唯一能保证这点的方法就是首先调用基类的构造器。接着，在派生类的构造器中，所有你可以访问的基类成员都已经初始化。另一个在构造器中能知道所有成员都是有效的理由是：无论何时有可能的话，你应该在所有成员对象（通过组合将对象置于类中）定义处初始化它们（例如，例子中的 **b**、**c** 和 **l**）。如果遵循这条实践，就可以帮助确保所有的基类成员和当前对象的成员对象都已经初始化。

不幸的是，这不能处理所有情况，在下一节会看到。

10.3.2 继承和清理

在使用组合和继承创建新类时，大部分时候你无需关心清理。子对象通常会留给垃圾收集器处理。如果你存在清理问题，那么必须用心地为新类创建一个 `dispose()` 方法（这里用的是我选择的名称，你可以使用更好的名称）。由于继承，如果有其他特殊的清理工作的话，就必须在派生类中重写 `dispose()` 方法。当重写 `dispose()` 方法时，记得调用基类的 `dispose()` 方法，否则基类的清理工作不会发生:

```
// polymorphism/Frog.java
// Cleanup and inheritance
// {java polymorphism.Frog}
```

When explaining a command, or language feature, or hardware widget, first describe the problem it is designed to solve. — David Martin

```java
package polymorphism;

class Characteristic {
    private String s;

    Characteristic(String s) {
        this.s = s;
        System.out.println("Creating Characteristic " + s);
    }

    protected void dispose() {
        System.out.println("disposing Characteristic " + s);
    }
}

class Description {
    private String s;

    Description(String s) {
        this.s = s;
        System.out.println("Creating Description " + s);
    }

    protected void dispose() {
        System.out.println("disposing Description " + s);
    }
}

class LivingCreature {
    private Characteristic p = new Characteristic("is alive");
    private Description t = new Description("Basic Living Creature");

    LivingCreature() {
        System.out.println("LivingCreature()");
    }

    protected void dispose() {
        System.out.println("LivingCreature dispose");
        t.dispose();
        p.dispose();
    }
}

class Animal extends LivingCreature {
    private Characteristic p = new Characteristic("has heart");
    private Description t = new Description("Animal not Vegetable");

    Animal() {
        System.out.println("Animal()");
    }

```

```java
    @Override
    protected void dispose() {
        System.out.println("Animal dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

class Amphibian extends Animal {
    private Characteristic p = new Characteristic("can live in water");
    private Description t = new Description("Both water and land");

    Amphibian() {
        System.out.println("Amphibian()");
    }

    @Override
    protected void dispose() {
        System.out.println("Amphibian dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

public class Frog extends Amphibian {
    private Characteristic p = new Characteristic("Croaks");
    private Description t = new Description("Eats Bugs");

    public Frog() {
        System.out.println("Frog()");
    }

    @Override
    protected void dispose() {
        System.out.println("Frog dispose");
        t.dispose();
        p.dispose();
        super.dispose();
    }

    public static void main(String[] args) {
        Frog frog = new Frog();
        System.out.println("Bye!");
        frog.dispose();
    }
}
```

输出:

```
Creating Characteristic is alive
Creating Description Basic Living Creature
LivingCreature()
Creating Characteristiv has heart
Creating Description Animal not Vegetable
Animal()
Creating Characteristic can live in water
Creating Description Both water and land
Amphibian()
Creating Characteristic Croaks
Creating Description Eats Bugs
Frog()
Bye!
Frog dispose
disposing Description Eats Bugs
disposing Characteristic Croaks
Amphibian dispose
disposing Description Both wanter and land
disposing Characteristic can live in water
Animal dispose
disposing Description Animal not Vegetable
disposing Characteristic has heart
LivingCreature dispose
disposing Description Basic Living Creature
disposing Characteristic is alive
```

层级结构中的每个类都有 **Characteristic** 和 **Description** 两个类型的成员对象，它们必须得被销毁。销毁的顺序应该与初始化的顺序相反，以防一个对象依赖另一个对象。对于属性来说，就意味着与声明的顺序相反（因为属性是按照声明顺序初始化的）。对于基类（遵循 C++ 析构函数的形式），首先进行派生类的清理工作，然后才是基类的清理。这是因为派生类的清理可能调用基类的一些方法，所以基类组件这时得存活，不能过早地被销毁。输出显示了，**Frog** 对象的所有部分都是按照创建的逆序销毁的。

尽管通常不必进行清理工作，但万一需要时，就得谨慎小心地执行。

Frog 对象拥有自己的成员对象，它创建了这些成员对象，并且知道它们能存活多久，所以它知道何时调用 `dispose()` 方法。然而，一旦某个成员对象被其它一个或多个对象共享时，问题就变得复杂了，不能只是简单地调用 `dispose()`。这里，也许就必须使用*引用计数*来跟踪仍然访问着共享对象的对象数量，如下：

```java
// polymorphism/ReferenceCounting.java
// Cleaning up shared member objects
class Shared {
    private int refcount = 0;
    private static long counter = 0;
    private final long id = counter++;

    Shared() {
```

```
        System.out.println("Creating " + this);
    }

    public void addRef() {
        refcount++;
    }

    protected void dispose() {
        if (--refcount == 0) {
            System.out.println("Disposing " + this);
        }
    }

    @Override
    public String toString() {
        return "Shared " + id;
    }
}

class Composing {
    private Shared shared;
    private static long counter = 0;
    private final long id = counter++;

    Composing(Shared shared) {
        System.out.println("Creating " + this);
        this.shared = shared;
        this.shared.addRef();
    }

    protected void dispose() {
        System.out.println("disposing " + this);
        shared.dispose();
    }

    @Override
    public String toString() {
        return "Composing " + id;
    }
}

public class ReferenceCounting {
    public static void main(String[] args) {
        Shared shared = new Shared();
        Composing[] composing = {
            new Composing(shared),
            new Composing(shared),
            new Composing(shared),
            new Composing(shared),
            new Composing(shared),
        };
```

```
        for (Composing c: composing) {
            c.dispose();
        }
    }
}
```

输出:

```
Creating Shared 0
Creating Composing 0
Creating Composing 1
Creating Composing 2
Creating Composing 3
Creating Composing 4
disposing Composing 0
disposing Composing 1
disposing Composing 2
disposing Composing 3
disposing Composing 4
Disposing Shared 0
```

static long counter 跟踪所创建的 **Shared** 实例数量，还提供了 **id** 的值。**counter** 的类型是 **long** 而不是 **int**，以防溢出（这只是个良好实践，对于本书的所有示例，**counter** 不会溢出）。**id** 是 **final** 的，因为它的值在初始化时确定后不应该变化。

在将一个 **shared** 对象附着在类上时，必须记住调用 `addRef()`，而 `dispose()` 方法会跟踪引用数，以确定在何时真正地执行清理工作。使用这种技巧需要加倍细心，但是如果正在共享需要被清理的对象，就没有太多选择了。

10.3.3 构造器内部多态方法的行为

构造器调用的层次结构带来了一个困境。如果在构造器中调用了正在构造的对象的动态绑定方法，会发生什么呢?

在普通的方法中，动态绑定的调用是在运行时解析的，因为对象不知道它属于方法所在的类还是类的派生类。

如果在构造器中调用了动态绑定方法，就会用到那个方法的重写定义。然而，调用的结果难以预料因为被重写的方法在对象被完全构造出来之前已经被调用，这使得一些 bug 很隐蔽，难以发现。

从概念上讲，构造器的工作就是创建对象（这并非是平常的工作）。在构造器内部，整个对象可能只是部分形成——只知道基类对象已经初始化。如果构造器只是构造对象过程中的一个步骤，且构造的对象所属的类是从构造器所属的类派生出的，那么派生部分在当前构造器被调用时还没有初始化。然而，一个动态绑定的方法调用向外深入到继承层次结构中，它可以调用派生类的方法。如果你在构造器中这么做，就可能调用一个方法，该方法操纵的成员可能还没有初始化——这肯定会带来灾难。

下面例子展示了这个问题:

只有不到 10% 的代码用于完成这个程序表面上的目的，余下的都在处理输入输出、数据验证、数据结构维护等家务活。 —Mary Shaw

```java
// polymorphism/PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect
class Glyph {
    void draw() {
        System.out.println("Glyph.draw()");
    }

    Glyph() {
        System.out.println("Glyph() before draw()");
        draw();
        System.out.println("Glyph() after draw()");
    }
}

class RoundGlyph extends Glyph {
    private int radius = 1;

    RoundGlyph(int r) {
        radius = r;
        System.out.println("RoundGlyph.RoundGlyph(), radius = " +
            radius);
    }

    @Override
    void draw() {
        System.out.println("RoundGlyph.draw(), radius = " + radius);
    }
}

public class PolyConstructors {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
}
```

输出:

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
```

Glyph 的 `draw()` 被设计为可重写，在 **RoundGlyph** 这个方法被重写。但是 **Glyph** 的构造器里调用了这个方法，结果调用了 **RoundGlyph** 的 `draw()` 方法，这看起来正是我们的目的。输出结果表明，当 **Glyph** 构造器调用了 `draw()` 时，**radius** 的值不是默认初始值 1 而是 0。这可能会导致在屏幕上只画了一个点或干脆什么都不画，于是我们只能干瞪眼，试图找到程序不工作的原因。

前一小节描述的初始化顺序并不十分完整，而这正是解决谜团的关键所在。初始化的实际过程

是：

1. 在所有事发生前，分配给对象的存储空间会被初始化为二进制 0。
2. 如前所述调用基类构造器。此时调用重写后的 `draw()` 方法（是的，在调用 **RoundGraph** 构造器之前调用），由步骤 1 可知，**radius** 的值为 0。
3. 按声明顺序初始化成员。
4. 最终调用派生类的构造器。

这么做有个优点：所有事物至少初始化为 0（或某些特殊数据类型与 0 等价的值），而不是仅仅留作垃圾。这包括了通过组合嵌入类中的对象引用，被赋予 **null**。如果忘记初始化该引用，就会在运行时出现异常。观察输出结果，就会发现所有事物都是 0。

另一方面，应该震惊于输出结果。逻辑方面我们已经做得非常完美，然而行为仍不可思议的错了，编译器也没有报错（C++ 在这种情况下会产生更加合理的行为）。像这样的 bug 很容易被忽略，需要花很长时间才能发现。

因此，编写构造器有一条良好规范：做尽量少的事让对象进入良好状态。如果有可能的话，尽量不要调用类中的任何方法。在基类的构造器中能安全调用的只有基类的 **final** 方法（这也适用于可被看作是 **final** 的 **private** 方法）。这些方法不能被重写，因此不会产生意想不到的结果。你可能无法永远遵循这条规范，但应该朝着它努力。

10.4 协变返回类型

Java 5 中引入了协变返回类型，这表示派生类的被重写方法可以返回基类方法返回类型的派生类型：

```java
// polymorphism/CovariantReturn.java
class Grain {
    @Override
    public String toString() {
        return "Grain";
    }
}

class Wheat extends Grain {
    @Override
    public String toString() {
        return "Wheat";
    }
}

class Mill {
    Grain process() {
        return new Grain();
    }
```

如果有人基本上做出了你想要的东西，你就没必要自己写一个新程序，就算你非写不可，也请尽可能地利用现有的代码。—Richard Hill

```
}

class WheatMill extends Mill {
    @Override
    Wheat process() {
        return new Wheat();
    }
}

public class CovariantReturn {
    public static void main(String[] args) {
        Mill m = new Mill();
        Grain g = m.process();
        System.out.println(g);
        m = new WheatMill();
        g = m.process();
        System.out.println(g);
    }
}
```

输出:

```
Grain
Wheat
```

关键区别在于 Java 5 之前的版本强制要求被重写的 `process()` 方法必须返回 **Grain** 而不是 **Wheat**，即使 **Wheat** 派生自 **Grain**，因而也应该是一种合法的返回类型。协变返回类型允许返回更具体的 **Wheat** 类型。

10.5 使用继承设计

学习过多态之后，一切看似都可以被继承，因为多态是如此巧妙的工具。这会给设计带来负担。事实上，如果利用已有类创建新类首先选择继承的话，事情会变得莫名的复杂。

更好的方法是首先选择组合，特别是不知道该使用哪种方法时。组合不会强制设计是继承层次结构，而且组合更加灵活，因为可以动态地选择类型（因而选择相应的行为），而继承要求必须在编译时知道确切类型。下面例子说明了这点:

```
// polymorphism/Transmogrify.java
// Dynamically changing the behavior of an object
// via composition (the "State" design pattern)
class Actor {
    public void act() {}
}

class HappyActor extends Actor {
    @Override
```

```
    public void act() {
        System.out.println("HappyActor");
    }
}

class SadActor extends Actor {
    @Override
    public void act() {
        System.out.println("SadActor");
    }
}

class Stage {
    private Actor actor = new HappyActor();

    public void change() {
        actor = new SadActor();
    }

    public void performPlay() {
        actor.act();
    }
}

public class Transmogrify {
    public static void main(String[] args) {
        Stage stage = new Stage();
        stage.performPlay();
        stage.change();
        stage.performPlay();
    }
}
```

输出:

```
HappyActor
SadActor
```

Stage 对象中包含了 **Actor** 引用，该引用被初始化为指向一个 **HappyActor** 对象，这意味着 `performPlay()` 会产生一个特殊行为。但是既然引用可以在运行时与其他不同的对象绑定，那么它就可以被替换成对 **SadActor** 的引用，`performPlay()` 的行为随之改变。这样你就获得了运行时的动态灵活性（这被称为状态模式）。与之相反，我们不能在运行时决定继承不同的对象，那在编译时就完全确定下来了。

有一条通用准则：使用继承表达行为的差异，使用属性表达状态的变化。在上个例子中，两者都用到了。通过继承的到的两个不同类在 `act()` 方法中表达了不同的行为，**Stage** 通过组合使自己的状态发生变化。这里状态的改变产生了行为的改变。

10.5.1 替代 vs 扩展

采用“纯粹”的方式创建继承层次结构看上去是最清晰的方法。即只有基类的方法才能在派生类中被重写，就像下图这样：

Figure 10.3: 类图

这被称作纯粹的“is - a”关系，因为类的接口已经确定了它是什么。继承可以确保任何派生类都拥有基类的接口，绝对不会少。如果按图上这么做，派生类将只拥有基类的接口。

纯粹的替代意味着派生类可以完美地替代基类，当使用它们时，完全不需要知道这些子类的信息。也就是说，基类可以接收任意发送给派生类的消息，因为它们具有完全相同的接口。只需将派生类向上转型，不要关注对象的具体类型。所有一切都可以通过多态处理。

按这种方式思考，似乎只有纯粹的“is - a”关系才是唯一明智的做法，其他任何设计只会导致混乱且注定失败。这其实也是个陷阱。一旦按这种方式开始思考，就会转而发现继承扩展接口（遗憾的是，extends 关键字似乎怂恿我们这么做）才是解决特定问题的完美方案。这可以称为“is - like - a” 关系，因为派生类就像是基类——它有着相同的基本接口，但还具有需要额外方法实现的其他特性：

虽然这是一种有用且明智的方法（依赖具体情况），但是也存在缺点。派生类中接口的扩展部分在基类中不存在（不能通过基类访问到这些扩展接口），因此一旦向上转型，就不能通过基类调用这些新方法：

如果不向上转型，就不会遇到这个问题。但是通常情况下，我们需要重新查明对象的确切类型，从而能够访问该类型中的扩展方法。下一节说明如何做到这点。

10.5.2 向下转型与运行时类型信息

由于向上转型（在继承层次中向上移动）会丢失具体的类型信息，那么为了重新获取类型信息，就需要在继承层次中向下移动，使用*向下转型*。

向上转型永远是安全的，因为基类不会具有比派生类更多的接口。因此，每条发送给基类接口的消息都能被接收。但是对于向下转型，你无法知道一个形状是圆，它有可能是三角形、正方形或其他一些

类型。

为了解决这个问题，必须得有某种方法确保向下转型是正确的，防止意外转型到一个错误类型，进而发送对象无法接收的消息。这么做是不安全的。

在某些语言中（如 C++），必须执行一个特殊的操作来获得安全的向下转型，但是在 Java 中，每次转型都会被检查！所以即使只是进行一次普通的加括号形式的类型转换，在运行时这个转换仍会被检查，以确保它的确是希望的那种类型。如果不是，就会得到 ClassCastException （类转型异常）。这种在运行时检查类型的行为称作运行时类型信息。下面例子展示了 RTTI 的行为：

```
// polymorphism/RTTI.java
// Downcasting & Runtime type information (RTTI)
// {ThrowsException}
class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    @Override
    public void f() {}
    @Override
    public void g() {}
    public void u() {}
    public void v() {}
    public void w() {}
}

public class RTTI {
    public static void main(String[] args) {
        Useful[] x = {
            new Useful(),
            new MoreUseful()
        };
        x[0].f();
        x[1].g();
        // Compile time: method not found in Useful:
        //- x[1].u();
        ((MoreUseful) x[1]).u(); // Downcast/RTTI
        ((MoreUseful) x[0]).u(); // Exception thrown
    }
}
```

输出：

```
Exception in thread "main"
java.lang.ClassCastException: Useful cannot be cast to
MoreUseful
at RTTI.main
```

正如前面类图所示，**MoreUseful** 扩展了 **Useful** 的接口。而 **MoreUseful** 也继承了 **Useful**，所以它可以向上转型为 **Useful**。在 `main()` 方法中可以看到这种情况的发生。因为两个对象都是 **Useful** 类型，所以对它们都可以调用 `f()` 和 `g()` 方法。如果试图调用 `u()` 方法（只存在于 **MoreUseful** 中），就会得到编译时错误信息。

为了访问 **MoreUseful** 对象的扩展接口，就得尝试向下转型。如果转型为正确的类型，就转型成功。否则，就会得到 ClassCastException 异常。你不必为这个异常编写任何特殊代码，因为它指出了程序员在程序的任何地方都可能犯的错误。**{ThrowsException}** 注释标签告知本书的构建系统：在运行程序时，预期抛出一个异常。

RTTI 不仅仅包括简单的转型。例如，它还提供了一种方法，使你可以在试图向下转型前检查所要处理的类型。“类型信息”一章中会详细阐述运行时类型信息的方方面面。

10.6 本章小结

多态意味着“不同的形式”。在面向对象编程中，我们持有从基类继承而来的相同接口和使用该接口的不同形式：不同版本的动态绑定方法。

在本章中，你可以看到，如果不使用数据抽象和继承，就不可能理解甚至创建多态的例子。多态是一种不能单独看待的特性（比如像 **switch** 语句那样），它只能作为类关系全景中的一部分，与其他特性协同工作。

为了在程序中有效地使用多态乃至面向对象的技术，就必须扩展自己的编程视野，不能只看到单一类中的成员和消息，而要看到类之间的共同特性和它们之间的关系。尽管这需要很大的努力，但是这么做是值得的。它能带来更快的程序开发、更好的代码组织、扩展性更好的程序和更易维护的代码。

但是记住，多态可能被滥用。仔细分析代码以确保多态确实能带来好处。

11 接口

接口和抽象类提供了一种将接口与实现分离的更加结构化的方法。

这种机制在编程语言中不常见，例如 C++ 只对这种概念有间接的支持。而在 Java 中存在这些关键字，说明这些思想很重要，Java 为它们提供了直接支持。

首先，我们将学习抽象类，一种介于普通类和接口之间的折中手段。尽管你的第一想法是创建接口，但是对于构建具有属性和未实现方法的类来说，抽象类也是重要且必要的工具。你不可能总是使用纯粹的接口。

11.1 抽象类和方法

在上一章的乐器例子中，基类 **Instrument** 中的方法往往是“哑”方法。如果调用了这些方法，就会出现一些错误。这是因为接口的目的是为它的派生类创建一个通用接口。

在那些例子中，创建这个通用接口的唯一理由是，不同的子类可以用不同的方式表示此接口。通用接口建立了一个基本形式，以此表达所有派生类的共同部分。另一种说法把 **Instrument** 称为抽象基类，或简称抽象类。

对于像 **Instrument** 那样的抽象类来说，它的对象几乎总是没有意义的。创建一个抽象类是为了通过通用接口操纵一系列类。因此，**Instrument** 只是表示接口，不是具体实现，所以创建一个 **Instrument** 的对象毫无意义，我们可能希望阻止用户这么做。通过让 **Instrument** 所有的方法产生错误，就可以达到这个目的，但是这么做会延迟到运行时才能得知错误信息，并且需要用户进行可靠、详尽的测试。最好能在编译时捕捉问题。

Java 提供了一个叫做*抽象方法*的机制，这个方法是不完整的：它只有声明没有方法体。下面是抽象方法的声明语法：

```
abstract void f();
```

包含抽象方法的类叫做*抽象类*。如果一个类包含一个或多个抽象方法，那么类本身也必须限定为抽象的，否则，编译器会报错。

```
// interface/Basic.java
abstract class Basic {
    abstract void unimplemented();
}
```

如果一个抽象类是不完整的，当试图创建这个类的对象时，Java 会怎么做呢？它不会创建抽象类的对象，所以我们只会得到编译器的错误信息。这样保证了抽象类的纯粹性，我们不用担心误用它。

```
// interfaces/AttemptToUseBasic.java
// {WillNotCompile}
public class AttemptToUseBasic {
    Basic b = new Basic();
    // error: Basic is abstract; cannot be instantiated
}
```

如果创建一个继承抽象类的新类并为之创建对象，那么就必须为基类的所有抽象方法提供方法定义。如果不这么做（可以选择不做），新类仍然是一个抽象类，编译器会强制我们为新类加上 **abstract** 关键字。

```
// interfaces/Basic2.java
abstract class Basic2 extends Basic {
    int f() {
        return 111;
    }

    abstract void g() {
        // unimplemented() still not implemented
    }
}
```

可以将一个不包含任何抽象方法的类指明为 **abstract**，在类中的抽象方法没啥意义但想阻止创建类的对象时，这么做就很有用。

```
// interfaces/AbstractWithoutAbstracts.java
abstract class Basic3 {
    int f() {
        return 111;
    }

    // No abstract methods
}

public class AbstractWithoutAbstracts {
    // Basic b3 = new Basic3();
    // error: Basic 3 is abstract; cannot be instantiated
}
```

为了创建可初始化的类，就要继承抽象类，并提供所有抽象方法的定义：

```
// interfaces/Instantiable.java
abstract class Uninstantiable {
    abstract void f();
    abstract int g();
}
```

```java

public class Instantiable extends Uninstantiable {
    @Override
    void f() {
        System.out.println("f()");
    }

    @Override
    int g() {
        return 22;
    }

    public static void main(String[] args) {
        Uninstantiable ui = new Instantiable();
    }
}
```

留意 `@Override` 的使用。没有这个注解的话，如果你没有定义相同的方法名或签名，抽象机制会认为你没有实现抽象方法从而产生编译时错误。因此，你可能认为这里的 `@Override` 是多余的。但是，`@Override` 还提示了这个方法被覆写——我认为这是有用的，所以我会使用 `@Override`，即使在没有这个注解，编译器告诉我错误的时候。

记住，事实上的访问权限是“friendly”。你很快会看到接口自动将其方法指明为 **public**。事实上，接口只允许 **public** 方法，如果不加访问修饰符的话，接口的方法不是 **friendly** 而是 **public**。然而，抽象类允许每件事：

```java
// interfaces/AbstractAccess.java
abstract class AbstractAccess {
    private void m1() {}

    // private abstract void m1a(); // illegal

    protected void m2() {}

    protected abstract void m2a();

    void m3() {}

    abstract void m3a();

    public void m4() {}

    public abstract void m4a();
}
```

private abstract 被禁止了是有意义的，因为你不可能在 **AbstractAccess** 的任何子类中合法地定义它。

上一章的 **Instrument** 类可以很轻易地转换为一个抽象类。只需要部分方法是 **abstract** 即可。将

一个类指明为 **abstract** 并不强制类中的所有方法必须都是抽象方法。如下图所示:

Figure 11.1: 类图

下面是修改成使用抽象类和抽象方法的管弦乐器的例子:

```
// interfaces/music4/Music4.java
// Abstract classes and methods
// {java interfaces.music4.Music4}
package interfaces.music4;
import polymorphism.music.Note;

abstract class Instrument {
    private int i; // Storage allocated for each

    public abstract void play(Note n);

    public String what() {
        return "Instrument";
    }

    public abstract void adjust();
}

class Wind extends Instrument {
```

```
    @Override
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }

    @Override
    public String what() {
        return "Wind";
    }

    @Override
    public void adjust() {
        System.out.println("Adjusting Wind");
    }
}

class Percussion extends Instrument {
    @Override
    public void play(Note n) {
        System.out.println("Percussion.play() " + n);
    }

    @Override
    public String what() {
        return "Percussion";
    }

    @Override
    public void adjust() {
        System.out.println("Adjusting Percussion");
    }
}

class Stringed extends Instrument {
    @Override
    public void play(Note n) {
        System.out.println("Stringed.play() " + n);
    }

    @Override
    public String what() {
        return "Stringed";
    }

    @Override
    public void adjust() {
        System.out.println("Adjusting Stringed");
    }
}

class Brass extends Wind {
```

```
    @Override
    public void play(Note n) {
        System.out.println("Brass.play() " + n);
    }

    @Override
    public void adjust() {
        System.out.println("Adjusting Brass");
    }
}

class Woodwind extends Wind {
    @Override
    public void play(Note n) {
        System.out.println("Woodwind.play() " + n);
    }

    @Override
    public String what() {
        return "Woodwind";
    }
}

public class Music4 {
    // Doesn't care about type, so new types
    // added to system still work right:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }

    static void tuneAll(Instrument[] e) {
        for (Instrument i: e) {
            tune(i);
        }
    }

    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
}
```

输出:

```
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
```

除了 **Instrument**，基本没区别。

创建抽象类和抽象方法是有帮助的，因为它们使得类的抽象性很明确，并能告知用户和编译器使用意图。抽象类同时也是一种有用的重构工具，使用它们使得我们很容易地将沿着继承层级结构上移公共方法。

11.2 接口创建

使用 **interface** 关键字创建接口。在本书中，interface 和 class 一样随处常见，除非特指关键字 **interface**，其他情况下都采用正常字体书写 interface。

描述 Java 8 之前的接口更加容易，因为它们只允许抽象方法。像下面这样：

```java
// interfaces/PureInterface.java
// Interface only looked like this before Java 8
public interface PureInterface {
    int m1();
    void m2();
    double m3();
}
```

我们甚至不用为方法加上 **abstract** 关键字，因为方法在接口中。Java 知道这些方法不能有方法体（仍然可以为方法加上 **abstract** 关键字，但是看起来像是不明白接口，徒增难堪罢了）。

因此，在 Java 8 之前我们可以这么说：**interface** 关键字产生一个完全抽象的类，没有提供任何实现。我们只能描述类应该像什么，做什么，但不能描述怎么做，即只能决定方法名、参数列表和返回类型，但是无法确定方法体。接口只提供形式，通常来说没有实现，尽管在某些受限制的情况下可以有实现。

一个接口表示：所有实现了该接口的类看起来都像这样。因此，任何使用某特定接口的代码都知道可以调用该接口的哪些方法，而且仅需知道这些。所以，接口被用来建立类之间的协议。（一些面向对象编程语言中，使用 protocol 关键字完成相同的功能。）

Java 8 中接口稍微有些变化，因为 Java 8 允许接口包含默认方法和静态方法——基于某些重要原因，看到后面你会理解。接口的基本概念仍然没变，介于类型之上、实现之下。接口与抽象类最明显的区别可能就是使用上的惯用方式。接口的典型使用是代表一个类的类型或一个形容词，如 Runnable 或 Serializable，而抽象类通常是类层次结构的一部分或一件事物的类型，如 String 或 ActionHero。

使用关键字 **interface** 而不是 **class** 来创建接口。和类一样，需要在关键字 **interface** 前加上 **public** 关键字（但只是在接口名与文件名相同的情况下），否则接口只有包访问权限，只能在接口相同

Rule of Credibility The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time. — Tom Cargill

的包下才能使用它。

接口同样可以包含属性，这些属性被隐式指明为 **static** 和 **final**。

使用 **implements** 关键字使一个类遵循某个特定接口（或一组接口），它表示：接口只是外形，现在我要说明它是如何工作的。除此之外，它看起来像继承。

```
// interfaces/ImplementingAnInterface.java
interface Concept { // Package access
    void idea1();
    void idea2();
}

class Implementation implements Concept {
    @Override
    public void idea1() {
        System.out.println("idea1");
    }

    @Override
    public void idea2() {
        System.out.println("idea2");
    }
}
```

你可以选择显式地声明接口中的方法为 **public**，但是即使你不这么做，它们也是 **public** 的。所以当实现一个接口时，来自接口中的方法必须被定义为 **public**。否则，它们只有包访问权限，这样在继承时，它们的可访问权限就被降低了，这是 Java 编译器所不允许的。

11.2.1 默认方法

Java 8 为关键字 **default** 增加了一个新的用途（之前只用于 **switch** 语句和注解中）。当在接口中使用它时，任何实现接口却没有定义方法的时候可以使用 **default** 创建的方法体。默认方法比抽象类中的方法受到更多的限制，但是非常有用，我们将在“流式编程”一章中看到。现在让我们看下如何使用：

```
// interfaces/AnInterface.java
interface AnInterface {
    void firstMethod();
    void secondMethod();
}
```

我们可以像这样实现接口：

```
// interfaces/AnImplementation.java
public class AnImplementation implements AnInterface {
    public void firstMethod() {
        System.out.println("firstMethod");
    }
```

Less than 10% of the code has to do with the ostensible purpose of the system; the rest deals with input-output, data validation, data structure maintenance, and other housekeeping. — Mary Shaw

```

    public void secondMethod() {
        System.out.println("secondMethod");
    }

    public static void main(String[] args) {
        AnInterface i = new AnImplementation();
        i.firstMethod();
        i.secondMethod();
    }
}
```

输出:

```
firstMethod
secondMethod
```

如果我们在 **AnInterface** 中增加一个新方法 `newMethod()`，而在 **AnImplementation** 中没有实现它，编译器就会报错:

```
AnImplementation.java:3:error: AnImplementation is not abstract and
   does not override abstract method newMethod() in AnInterface
public class AnImplementation implements AnInterface {
^
1 error
```

如果我们使用关键字 **default** 为 `newMethod()` 方法提供默认的实现，那么所有与接口有关的代码能正常工作，不受影响，而且这些代码还可以调用新的方法 `newMethod()`:

```
// interfaces/InterfaceWithDefault.java
interface InterfaceWithDefault {
    void firstMethod();
    void secondMethod();

    default void newMethod() {
        System.out.println("newMethod");
    }
}
```

关键字 **default** 允许在接口中提供方法实现——在 Java 8 之前被禁止。

```
// interfaces/Implementation2.java
public class Implementation2 implements InterfaceWithDefault {
    @Override
    public void firstMethod() {
        System.out.println("firstMethod");
    }

    @Override
    public void secondMethod() {
```

```
        System.out.println("secondMethod")
    }

    public static void main(String[] args) {
        InterfaceWithDefault i = new Implementation2();
        i.firstMethod();
        i.secondMethod();
        i.newMethod();
    }
}
```

输出:

```
firstMethod
secondMethod
newMethod
```

尽管 **Implementation2** 中未定义 `newMethod()`，但是可以使用 `newMethod()` 了。

增加默认方法的极具说服力的理由是它允许在不破坏已使用接口的代码的情况下，在接口中增加新的方法。默认方法有时也被称为*守卫方法*或*虚拟扩展方法*。

11.2.2 多继承

多继承意味着一个类可能从多个父类型中继承特征和特性。

Java 在设计之初，C++ 的多继承机制饱受诟病。Java 过去是一种严格要求单继承的语言：只能继承自一个类（或抽象类），但可以实现任意多个接口。在 Java 8 之前，接口没有包袱——它只是方法外貌的描述。

多年后的现在，Java 通过默认方法具有了某种多继承的特性。结合带有默认方法的接口意味着结合了多个基类中的行为。因为接口中仍然不允许存在属性（只有静态属性，不适用），所以属性仍然只会来自单个基类或抽象类，也就是说，不会存在状态的多继承。正如下面这样:

```
// interfaces/MultipleInheritance.java
import java.util.*;

interface One {
    default void first() {
        System.out.println("first");
    }
}

interface Two {
    default void second() {
        System.out.println("second");
    }
}

```

```java
interface Three {
    default void third() {
        System.out.println("third");
    }
}

class MI implements One, Two, Three {}

public class MultipleInheritance {
    public static void main(String[] args) {
        MI mi = new MI();
        mi.first();
        mi.second();
        mi.third();
    }
}
```

输出:

```
first
second
third
```

现在我们做些在 Java 8 之前不可能完成的事: 结合多个源的实现。只要基类方法中的方法名和参数列表不同, 就能工作得很好, 否则会得到编译器错误:

```java
// interface/MICollision.java
import java.util.*;

interface Bob1 {
    default void bob() {
        System.out.println("Bob1::bob");
    }
}

interface Bob2 {
    default void bob() {
        System.out.println("Bob2::bob");
    }
}

// class Bob implements Bob1, Bob2 {}
/* Produces:
error: class Bob inherits unrelated defaults
for bob() from types Bob1 and Bob2
class Bob implements Bob1, Bob2 {}
^
1 error
*/

```

```
interface Sam1 {
    default void sam() {
        System.out.println("Sam1::sam");
    }
}

interface Sam2 {
    default void sam(int i) {
        System.out.println(i * 2);
    }
}

// This works because the argument lists are distinct:
class Sam implements Sam1, Sam2 {}

interface Max1 {
    default void max() {
        System.out.println("Max1::max");
    }
}

interface Max2 {
    default int max() {
        return 47;
    }
}

// class Max implements Max1, Max2 {}
/* Produces:
error: types Max2 and Max1 are imcompatible;
both define max(), but with unrelated return types
class Max implements Max1, Max2 {}
^
1 error
*/
```

Sam 类中的两个 `sam()` 方法有相同的方法名但是签名不同——方法签名包括方法名和参数类型，编译器也是用它来区分方法。但是从 **Max** 类可看出，返回类型不是方法签名的一部分，因此不能用来区分方法。为了解决这个问题，需要覆写冲突的方法：

```
// interfaces/Jim.java
import java.util.*;

interface Jim1 {
    default void jim() {
        System.out.println("Jim1::jim");
    }
}

interface Jim2 {
```

```
    default void jim() {
        System.out.println("Jim2::jim");
    }
}

public class Jim implements Jim1, Jim2 {
    @Override
    public void jim() {
        Jim2.super.jim();
    }

    public static void main(String[] args) {
        new Jim().jim();
    }
}
```

输出:

```
Jim2::jim
```

当然，你可以重定义 `jim()` 方法，但是也能像上例中那样使用 **super** 关键字选择基类实现中的一种。

11.2.3 接口中的静态方法

Java 8 允许在接口中添加静态方法。这么做能恰当地把工具功能置于接口中，从而操作接口，或者成为通用的工具:

```
// onjava/Operations.java
package onjava;
import java.util.*;

public interface Operations {
    void execute();

    static void runOps(Operations... ops) {
        for (Operations op: ops) {
            op.execute();
        }
    }

    static void show(String msg) {
        System.out.println(msg);
    }
}
```

这是模版方法设计模式的一个版本（在“设计模式”一章中详细描述），`runOps()` 是一个模版方法。`runOps()` 使用可变参数列表，因而我们可以传入任意多的 **Operation** 参数并按顺序运行它们:

Translating a working program to a new language or system takes ten percent of the original developmefit time or manpower or cost. — Douglas W. Jones

```java
// interface/Machine.java
import java.util.*;
import onjava.Operations;

class Bing implements Operations {
    @Override
    public void execute() {
        Operations.show("Bing");
    }
}

class Crack implements Operations {
    @Override
    public void execute() {
        Operations.show("Crack");
    }
}

class Twist implements Operations {
    @Override
    public void execute() {
        Operations.show("Twist");
    }
}

public class Machine {
    public static void main(String[] args) {
        Operations.runOps(
            new Bing(), new Crack(), new Twist());
    }
}
```

输出:

```
Bing
Crack
Twist
```

这里展示了创建 **Operations** 的不同方式：一个外部类 (Bing)，一个匿名类，一个方法引用和 lambda 表达式——毫无疑问用在这里是最好的解决方法。

这个特性是一项改善，因为它允许把静态方法放在更合适的地方。

11.2.4 Instrument 作为接口

回顾下乐器的例子，使用接口的话:

Figure 11.2: 类图

类 **Woodwind** 和 **Brass** 说明一旦实现了某个接口，那么其实现就变成一个普通类，可以按常规方式扩展它。

接口的工作方式使得我们不需要显式声明其中的方法为 **public**，它们自动就是 **public** 的。`play()` 和 `adjust()` 使用 **default** 关键字定义实现。在 Java 8 之前，这些定义要在每个实现中重复实现，显得多余且令人烦恼：

```
// interfaces/music5/Music5.java
// {java interfaces.music5.Music5}
package interfaces.music5;
import polymorphism.music.Note;

interface Instrument {
    // Compile-time constant:
    int VALUE = 5; // static & final

    default void play(Note n)  // Automatically public
        System.out.println(this + ".play() " + n);
    }

    default void adjust() {
        System.out.println("Adjusting " + this);
    }
}
```

```java

class Wind implements Instrument {
    @Override
    public String toString() {
        return "Wind";
    }
}

class Percussion implements Instrument {
    @Override
    public String toString() {
        return "Percussion";
    }
}

class Stringed implements Instrument {
    @Override
    public String toString() {
        return "Stringed";
    }
}

class Brass extends Wind {
    @Override
    public String toString() {
        return "Brass";
    }
}

class Woodwind extends Wind {
    @Override
    public String toString() {
        return "Woodwind";
    }
}

public class Music5 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }

    static void tuneAll(Instrument[] e) {
        for (Instrument i: e) {
            tune(i);
        }
    }

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

```
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        }
        tuneAll(orchestra);
    }
}
```

输出:

```
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
```

这个版本的例子的另一个变化是: `what()` 被修改为 `toString()` 方法，因为 `toString()` 实现的正是 `what()` 方法要实现的逻辑。因为 `toString()` 是根基类 **Object** 的方法，所以它不需要出现在接口中。

注意到，无论是将其向上转型为称作 **Instrument** 的普通类，或称作 **Instrument** 的抽象类，还是叫作 **Instrument** 的接口，其行为都是相同的。事实上，从 `tune()` 方法上看不出来 **Instrument** 到底是一个普通类、抽象类，还是一个接口。

11.3 抽象类和接口

尤其是在 Java 8 引入 **default** 方法之后，选择用抽象类还是用接口变得更加令人困惑。下表做了明确的区分:

特性	接口	抽象类
组合	新类可以组合多个接口	只能继承单一抽象类
状态	不能包含属性（除了静态属性，不支持对象状态）	可以包含属性，非抽象方法可能引用这些属性
默认方法和抽象方法	不需要在子类中实现默认方法。默认方法可以引用其他接口的方法	必须在子类中实现抽象方法
构造器	没有构造器	可以有构造器

特性	接口	抽象类
可见性	隐式 **public**	可以是 **protected** 或友元

抽象类仍然是一个类，在创建新类时只能继承它一个。而创建类的过程中可以实现多个接口。

有一条实际经验：尽可能地抽象。因此，更倾向使用接口而不是抽象类。只有当必要时才使用抽象类。除非必须使用，否则不要用接口和抽象类。大多数时候，普通类已经做得很好，如果不行的话，再移动到接口或抽象类中。

11.4 完全解耦

当方法操纵的是一个类而非接口时，它就只能作用于那个类或其子类。如果想把方法应用于那个继承层级结构之外的类，就会触霉头。接口在很大程度上放宽了这个限制，因而使用接口可以编写复用性更好的代码。

例如有一个类 **Process** 有两个方法 `name()` 和 `process()`。`process()` 方法接受输入，修改并输出。把这个类作为基类用来创建各种不同类型的 **Processor**。下例中，**Processor** 的各个子类修改 String 对象（注意，返回类型可能是协变类型而非参数类型）：

```
// interfaces/Applicator.java
import java.util.*;

class Processor {
    public String name() {
        return getClass().getSimpleName();
    }

    public Object process(Object input) {
        return input;
    }
}

class Upcase extends Processor {
    // 返回协变类型
    @Override
    public String process(Object input) {
        return ((String) input).toUpperCase();
    }
}

class Downcase extends Processor {
    @Override
    public String process(Object input) {
        return ((String) input).toLowerCase();
```

```
    }
}

class Splitter extends Processor {
    @Override
    public String process(Object input) {
        // split() divides a String into pieces:
        return Arrays.toString(((String) input).split(" "));
    }
}

public class Applicator {
    public static void apply(Processor p, Object s) {
        System.out.println("Using Processor " + p.name());
        System.out.println(p.process(s));
    }

    public static void main(String[] args) {
        String s = "We are such stuff as dreams are made on";
        apply(new Upcase(), s);
        apply(new Downcase(), s);
        apply(new Splitter(), s);
    }
}
```

输出:

```
Using Processor Upcase
WE ARE SUCH STUFF AS DREAMS ARE MADE ON
Using Processor Downcase
we are such stuff as dreams are made on
Using Processor Splitter
[We, are, such, stuff, as, dreams, are, made, on]
```

Applicator 的 `apply()` 方法可以接受任何类型的 **Processor**，并将其应用到一个 **Object** 对象上输出结果。像本例中这样，创建一个能根据传入的参数类型从而具备不同行为的方法称为*策略*设计模式。方法包含算法中不变的部分，策略包含变化的部分。策略就是传入的对象，它包含要执行的代码。在这里，**Processor** 对象是策略，`main()` 方法展示了三种不同的应用于 **String s** 上的策略。

`split()` 是 **String** 类中的方法，它接受 **String** 类型的对象并以传入的参数作为分割界限，返回一个数组 **String[]**。在这里用它是为了更快地创建 **String** 数组。

假设现在发现了一组电子滤波器，它们看起来好像能使用 **Applicator** 的 `apply()` 方法:

```
// interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
    private static long counter;
    private final long id = counter++;
```

```java

    @Override
    public String toString() {
        return "Waveform " + id;
    }
}

// interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
    public String name() {
        return getClass().getSimpleName();
    }

    public Waveform process(Waveform input) {
        return input;
    }
}

// interfaces/filters/LowPass.java
package interfaces.filters;

public class LowPass extends Filter {
    double cutoff;

    public LowPass(double cutoff) {
        this.cutoff = cutoff;
    }

    @Override
    public Waveform process(Waveform input) {
        return input; // Dummy processing 哑处理
    }
}

// interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
    double cutoff;

    public HighPass(double cutoff) {
        this.cutoff = cutoff;
    }

    @Override
    public Waveform process(Waveform input) {
        return input;
    }
}
```

```java

// interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
    double lowCutoff, highCutoff;

    public BandPass(double lowCut, double highCut) {
        lowCutoff = lowCut;
        highCutoff = highCut;
    }

    @Override
    public Waveform process(Waveform input) {
        return input;
    }
}
```

Filter 类与 **Processor** 类具有相同的接口元素，但是因为它不是继承自 **Processor** —— 因为 **Filter** 类的创建者根本不知道你想将它当作 **Processor** 使用 —— 因此你不能将 **Applicator** 的 `apply()` 方法应用在 **Filter** 类上，即使这样做也能正常运行。主要是因为 **Applicator** 的 `apply()` 方法和 **Processor** 过于耦合，这阻止了 **Applicator** 的 `apply()` 方法被复用。另外要注意的一点是 Filter 类中 `process()` 方法的输入输出都是 **Waveform**。

但如果 **Processor** 是一个接口，那么限制就会变得松动到足以复用 **Applicator** 的 `apply()` 方法，用来接受那个接口参数。下面是修改后的 **Processor** 和 **Applicator** 版本：

```java
// interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;

public interface Processor {
    default String name() {
        return getClass().getSimpleName();
    }

    Object process(Object input);
}

// interfaces/interfaceprocessor/Applicator.java
package interfaces.interfaceprocessor;

public class Applicator {
    public static void apply(Processor p, Object s) {
        System.out.println("Using Processor " + p.name());
        System.out.println(p.process(s));
    }
}
```

复用代码的第一种方式是客户端程序员遵循接口编写类，像这样：

```java
// interfaces/interfaceprocessor/StringProcessor.java
// {java interfaces.interfaceprocessor.StringProcessor}
package interfaces.interfaceprocessor;
import java.util.*;

interface StringProcessor extends Processor {
    @Override
    String process(Object input); // [1]
    String S = "If she weighs the same as a duck, she's made of wood"; // [2]

    static void main(String[] args) { // [3]
        Applicator.apply(new Upcase(), S);
        Applicator.apply(new Downcase(), S);
        Applicator.apply(new Splitter(), S);
    }
}

class Upcase implements StringProcessor {
    // 返回协变类型
    @Override
    public String process(Object input) {
        return ((String) input).toUpperCase();
    }
}

class Downcase implements StringProcessor {
    @Override
    public String process(Object input) {
        return ((String) input).toLowerCase();
    }
}

class Splitter implements StringProcessor {
    @Override
    public String process(Object input) {
        return Arrays.toString(((String) input).split(" "));
    }
}
```

输出:

```
Using Processor Upcase
IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
Using Processor Downcase
if she weighs the same as a duck, she's made of wood
Using Processor Splitter
[If, she, weighs, the, same, as, a, duck,, she's, made, of, wood]
```

> [1] 该声明不是必要的，即使移除它，编译器也不会报错。但是注意这里的协变返回类型从 Object 变成了 String。
>
> [2] S 自动就是 final 和 static 的，因为它是在接口中定义的。
>
> [3] 可以在接口中定义 `main()` 方法。

这种方式运作得很好，然而你经常遇到的情况是无法修改类。例如在电子滤波器的例子中，类库是被发现而不是创建的。在这些情况下，可以使用*适配器*设计模式。适配器允许代码接受已有的接口产生需要的接口，如下：

```java
// interfaces/interfaceprocessor/FilterProcessor.java
// {java interfaces.interfaceprocessor.FilterProcessor}
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
    Filter filter;

    FilterAdapter(Filter filter) {
        this.filter = filter;
    }

    @Override
    public String name() {
        return filter.name();
    }

    @Override
    public Waveform process(Object input) {
        return filter.process((Waveform) input);
    }
}

public class FilterProcessor {
    public static void main(String[] args) {
        Waveform w = new Waveform();
        Applicator.apply(new FilterAdapter(new LowPass(1.0)), w);
        Applicator.apply(new FilterAdapter(new HighPass(2.0)), w);
        Applicator.apply(new FilterAdapter(new BandPass(3.0, 4.0)), w);
    }
}
```

输出：

```
Using Processor LowPass
Waveform 0
Using Processor HighPass
Waveform 0
```

```
Using Processor BandPass
Waveform 0
```

在这种使用适配器的方式中，**FilterAdapter** 的构造器接受已有的接口 **Filter**，继而产生需要的 **Processor** 接口的对象。你可能还注意到 **FilterAdapter** 中使用了委托。

协变允许我们从 `process()` 方法中产生一个 **Waveform** 而非 **Object** 对象。

将接口与实现解耦使得接口可以应用于多种不同的实现，因而代码更具可复用性。

11.5 多接口结合

接口没有任何实现——也就是说，没有任何与接口相关的存储——因此无法阻止结合的多接口。这是有价值的，因为你有时需要表示“一个 **x** 是一个 **a** 和一个 **b** 以及一个 **c**”。

Figure 11.3: 类图

派生类并不要求必须继承自抽象的或“具体的”（没有任何抽象方法）的基类。如果继承一个非接口的类，那么只能继承一个类，其余的基元素必须都是接口。需要将所有的接口名称置于 **implements** 关键字之后且用逗号分隔。可以有任意多个接口，并可以向上转型为每个接口，因为每个接口都是独立的类型。下例展示了一个由多个接口组合而成的具体类产生的新类：

```java
// interfaces/Adventure.java
// Multiple interfaces
interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
```

```
}

class ActionCharacter {
    public void fight(){}
}

class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly
    {
    public void swim() {}

    public void fly() {}
}

public class Adventure {
    public static void t(CanFight x) {
        x.fight();
    }

    public static void u(CanSwim x) {
        x.swim();
    }

    public static void v(CanFly x) {
        x.fly();
    }

    public static void w(ActionCharacter x) {
        x.fight();
    }

    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
}
```

类 **Hero** 结合了具体类 **ActionCharacter** 和接口 **CanFight**、**CanSwim** 和 **CanFly**。当通过这种方式结合具体类和接口时，需要将具体类放在前面，后面跟着接口（否则编译器会报错）。

接口 **CanFight** 和类 **ActionCharacter** 中的 `fight()` 方法签名相同，而在类 Hero 中也没有提供 `fight()` 的定义。可以扩展一个接口，但是得到的是另一个接口。当想创建一个对象时，所有的定义必须首先都存在。类 **Hero** 中没有显式地提供 `fight()` 的定义，是由于该方法在类 **ActionCharacter** 中已经定义过，这样才使得创建 **Hero** 对象成为可能。

在类 **Adventure** 中可以看到四个方法，它们把不同的接口和具体类作为参数。当创建一个 **Hero** 对象时，它可以被传入这些方法中的任意一个，意味着它可以依次向上转型为每个接口。Java 中这种接

口的设计方式，使得程序员不需要付出特别的努力。

记住，前面例子展示了使用接口的核心原因之一：为了能够向上转型为多个基类型（以及由此带来的灵活性）。然而，使用接口的第二个原因与使用抽象基类相同：防止客户端程序员创建这个类的对象，确保这仅仅只是一个接口。这带来了一个问题：应该使用接口还是抽象类呢？如果创建不带任何方法定义或成员变量的基类，就选择接口而不是抽象类。事实上，如果知道某事物是一个基类，可以考虑用接口实现它（这个主题在本章总结会再次讨论）。

11.6 使用继承扩展接口

通过继承，可以很容易在接口中增加方法声明，还可以在新接口中结合多个接口。这两种情况都可以得到新接口，如下例所示：

```java
// interfaces/HorrorShow.java
// Extending an interface with inheritance
interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    @Override
    public void menace() {}

    @Override
    public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
    void drinkBlood();
}

class VeryBadVampire implements Vampire {
    @Override
    public void menace() {}

    @Override
    public void destroy() {}

    @Override
```

提供这种选择, 不要找蹩脚的借口

```
    public void kill() {}

    @Override
    public void drinkBlood() {}
}

public class HorrorShow {
    static void u(Monster b) {
        b.menace();
    }

    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }

    static void w(Lethal l) {
        l.kill();
    }

    public static void main(String[] args) {
        DangerousMonster barney = new DragonZilla();
        u(barney);
        v(barney);
        Vampire vlad = new VeryBadVampire();
        u(vlad);
        v(vlad);
        w(vlad);
    }
}
```

接口 **DangerousMonster** 是 **Monster** 简单扩展的一个新接口，类 **DragonZilla** 实现了这个接口。

Vampire 中使用的语法仅适用于接口继承。通常来说，**extends** 只能用于单一类，但是在构建接口时可以引用多个基类接口。注意到，接口名之间用逗号分隔。

11.6.1 结合接口时的命名冲突

当实现多个接口时可能会存在一个小陷阱。在前面的例子中，**CanFight** 和 **ActionCharacter** 具有完全相同的 `fight()` 方法。完全相同的方法没有问题，但是如果它们的签名或返回类型不同会怎么样呢？这里有一个例子：

```
// interfaces/InterfaceCollision.java
interface I1 {
    void f();
}

```

```
interface I2 {
    int f(int i);
}

interface I3 {
    int f();
}

class C {
    public int f() {
        return 1;
    }
}

class C2 implements I1, I2 {
    @Override
    public void f() {}

    @Override
    public int f(int i) {
        return 1;  // 重载
    }
}

class C3 extends C implements I2 {
    @Override
    public int f(int i) {
        return 1; // 重载
    }
}

class C4 extends C implements I3 {
    // 完全相同，没问题
    @Override
    public int f() {
        return 1;
    }
}

// 方法的返回类型不同
//- class C5 extends C implements I1 {}
//- interface I4 extends I1, I3 {}
```

覆写、实现和重载令人不快地搅和在一起带来了困难。同时，重载方法仅根据返回类型是区分不了的。当不注释最后两行时，报错信息如下：

```
error: C5 is not abstract and does not override abstract
method f() in I1
class C5 extends C implements I1 {}
error: types I3 and I1 are incompatible; both define f(),
```

```
but with unrelated return types
interfacce I4 extends I1, I3 {}
```

当打算组合接口时，在不同的接口中使用相同的方法名通常会造成代码可读性的混乱，尽量避免这种情况。

11.7 接口适配

接口最吸引人的原因之一是相同的接口可以有多个实现。在简单情况下体现在一个方法接受接口作为参数，该接口的实现和传递对象给方法则交由你来做。

因此，接口的一种常见用法是前面提到的*策略*设计模式。编写一个方法执行某些操作并接受一个指定的接口作为参数。可以说："只要对象遵循接口，就可以调用方法"，这使得方法更加灵活，通用，并更具可复用性。

例如，类 **Scanner** 的构造器接受的是一个 **Readable** 接口（在"字符串"一章中学习更多相关内容）。你会发现 **Readable** 没有用作 Java 标准库中其他任何方法的参数——它是单独为 **Scanner** 创建的，因此 **Scanner** 没有将其参数限制为某个特定类。通过这种方式，**Scanner** 可以与更多的类型协作。如果你创建了一个新类并想让 **Scanner** 作用于它，就让它实现 **Readable** 接口，像这样：

```java
// interfaces/RandomStrings.java
// Implementing an interface to conform to a method
import java.nio.*;
import java.util.*;

public class RandomStrings implements Readable {
    private static Random rand = new Random(47);
    private static final char[] CAPITALS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        .toCharArray();
    private static final char[] LOWERS = "abcdefghijklmnopqrstuvwxyz".
        toCharArray();
    private static final char[] VOWELS = "aeiou".toCharArray();
    private int count;

    public RandomStrings(int count) {
        this.count = count;
    }

    @Override
    public int read(CharBuffer cb) {
        if (count-- == 0) {
            return -1; // indicates end of input
        }
        cb.append(CAPITALS[rand.nextInt(CAPITALS.length)]);
        for (int i = 0; i < 4; i++) {
            cb.append(VOWELS[rand.nextInt(VOWELS.length)]);
            cb.append(LOWERS[rand.nextInt(LOWERS.length)]);
```

```
        }
        cb.append(" ");
        return 10; // Number of characters appended
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(new RandomStrings(10));
        while (s.hasNext()) {
            System.out.println(s.next());
        }
    }
}
```

输出:

```
Yazeruyac
Fowenucor
Goeazimom
Raeuuacio
Nuoadesiw
Hageaikux
Ruqicibui
Numasetih
Kuuuuozog
Waqizeyoy
```

Readable 接口只需要实现 `read()` 方法（注意 `@Override` 注解的突出方法）。在 `read()` 方法里，将输入内容添加到 **CharBuffer** 参数中（有多种方法可以实现，查看 **CharBuffer** 文档），或在没有输入时返回 **-1**。

假设你有一个类没有实现 **Readable** 接口，怎样才能让 **Scanner** 作用于它呢？下面是一个产生随机浮点数的例子:

```
// interfaces/RandomDoubles.java
import java.util.*;

public interface RandomDoubles {
    Random RAND = new Random(47);

    default double next() {
        return RAND.nextDouble();
    }

    static void main(String[] args) {
        RandomDoubles rd = new RandomDoubles(){};
        for (int i = 0; i < 7; i++) {
            System.out.println(rd.next() + " ");
        }
    }
}
```

输出:

```
0.7271157860730044
0.5309454508634242
0.16020656493302599
0.18847866977771732
0.5166020801268457
0.2678662084200585
0.2613610344283964
```

我们可以再次使用适配器模式，但这里适配器类可以实现两个接口。因此，通过关键字 **interface** 提供的多继承，我们可以创建一个既是 **RandomDoubles**，又是 **Readable** 的类:

```java
// interfaces/AdaptedRandomDoubles.java
// creating an adapter with inheritance
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles implements RandomDoubles, Readable {
    private int count;

    public AdaptedRandomDoubles(int count) {
        this.count = count;
    }

    @Override
    public int read(CharBuffer cb) {
        if (count-- == 0) {
            return -1;
        }
        String result = Double.toString(next()) + " ";
        cb.append(result);
        return result.length();
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(new AdaptedRandomDoubles(7));
        while (s.hasNextDouble()) {
            System.out.print(s.nextDouble() + " ");
        }
    }
}
```

输出:

```
0.7271157860730044 0.5309454508634242
0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585
0.2613610344283964
```

因为你可以以这种方式在已有类中增加新接口，所以这就意味着一个接受接口类型的方法提供了

一种让任何类都可以与该方法进行适配的方式。这就是使用接口而不是类的强大之处。

11.8 接口字段

因为接口中的字段都自动是 **static** 和 **final** 的，所以接口就成为了创建一组常量的方便的工具。在 Java 5 之前，这是产生与 C 或 C++ 中的 enum (枚举类型) 具有相同效果的唯一方式。所以你可能在 Java 5 之前的代码中看到：

```
// interfaces/Months.java
// Using interfaces to create groups of constants
public interface Months {
    int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
}
```

注意 Java 中使用大写字母的风格定义具有初始化值的 **static final** 变量。接口中的字段自动是 **public** 的，所以没有显式指明这点。

自 Java 5 开始，我们有了更加强大和灵活的关键字 **enum**，那么在接口中定义常量组就显得没什么意义了。然而当你阅读遗留的代码时，在很多场合你还会碰到这种旧的习惯用法。在“枚举”一章中你会学习到更多关于枚举的内容。

11.8.1 初始化接口中的字段

接口中定义的字段不能是“空 **final**”，但是可以用非常量表达式初始化。例如：

```
// interfaces/RandVals.java
// Initializing interface fields with
// non-constant initializers
import java.util.*;

public interface RandVals {
    Random RAND = new Random(47);
    int RANDOM_INT = RAND.nextInt(10);
    long RANDOM_LONG = RAND.nextLong() * 10;
    float RANDOM_FLOAT = RAND.nextLong() * 10;
    double RANDOM_DOUBLE = RAND.nextDouble() * 10;
}
```

因为字段是 **static** 的，所以它们在类第一次被加载时初始化，这发生在任何字段首次被访问时。下面是个简单的测试：

```java
// interfaces/TestRandVals.java
public class TestRandVals {
    public static void main(String[] args) {
        System.out.println(RandVals.RANDOM_INT);
        System.out.println(RandVals.RANDOM_LONG);
        System.out.println(RandVals.RANDOM_FLOAT);
        System.out.println(RandVals.RANDOM_DOUBLE);
    }
}
```

输出:

```
8
-32032247016559954
-8.5939291E18
5.779976127815049
```

这些字段不是接口的一部分，它们的值被存储在接口的静态存储区域中。

11.9 接口嵌套

接口可以嵌套在类或其他接口中。下面揭示一些有趣的特性:

```java
// interfaces/nesting/NestingInterfaces.java
// {java interfaces.nesting.NestingInterfaces}
package interfaces.nesting;

class A {
    interface B {
        void f();
    }

    public class BImp implements B {
        @Override
        public void f() {}
    }

    public class BImp2 implements B {
        @Override
        public void f() {}
    }

    public interface C {
        void f();
    }

    class CImp implements C {
        @Override
        public void f() {}
```

```
    }

    private class CImp2 implements C {
        @Override
        public void f() {}
    }

    private interface D {
        void f();
    }

    private class DImp implements D {
        @Override
        public void f() {}
    }

    public class DImp2 implements D {
        @Override
        public void f() {}
    }

    public D getD() {
        return new DImp2();
    }

    private D dRef;

    public void receiveD(D d) {
        dRef = d;
        dRef.f();
    }
}

interface E {
    interface G {
        void f();
    }
    // Redundant "public"
    public interface H {
        void f();
    }

    void g();
    // Cannot be private within an interface
    //- private interface I {}
}

public class NestingInterfaces {
    public class BImp implements A.B {
        @Override
        public void f() {}
```

```
    }

    class CImp implements A.C {
        @Override
        public void f() {}
    }
    // Cannot implements a private interface except
    // within that interface's defining class:
    //- class DImp implements A.D {
    //- public void f() {}
    //- }
    class EImp implements E {
        @Override
        public void g() {}
    }

    class EGImp implements E.G {
        @Override
        public void f() {}
    }

    class EImp2 implements E {
        @Override
        public void g() {}

        class EG implements E.G {
            @Override
            public void f() {}
        }
    }

    public static void main(String[] args) {
        A a = new A();
        // Can't access to A.D:
        //- A.D ad = a.getD();
        // Doesn't return anything but A.D:
        //- A.DImp2 di2 = a.getD();
        // cannot access a member of the interface:
        //- a.getD().f();
        // Only another A can do anything with getD():
        A a2 = new A();
        a2.receiveD(a.getD());
    }
}
```

在类中嵌套接口的语法是相当显而易见的。就像非嵌套接口一样，它们具有 **public** 或包访问权限的可见性。

作为一种新添加的方式，接口也可以是 **private** 的，例如 **A.D**（同样的语法同时适用于嵌套接口和嵌套类）。那么 **private** 嵌套接口有什么好处呢？你可能猜测它只是被用来实现一个 **private** 内部类，

就像 **DImp**。然而 **A.DImp2** 展示了它可以被实现为 **public** 类，但是 **A.DImp2** 只能被自己使用，你无法说它实现了 **private** 接口 **D**，所以实现 **private** 接口是一种可以强制该接口中的方法定义不会添加任何类型信息（即不可以向上转型）的方式。

`getD()` 方法产生了一个与 **private** 接口有关的窘境。它是一个 **public** 方法却返回了对 **private** 接口的引用。能对这个返回值做些什么呢？`main()` 方法里进行了一些使用返回值的尝试但都失败了。返回值必须交给有权使用它的对象，本例中另一个 **A** 通过 `receiveD()` 方法接受了它。

接口 **E** 说明了接口之间也能嵌套。然而，作用于接口的规则——尤其是，接口中的元素必须是 **public** 的——在此都会被严格执行，所以嵌套在另一个接口中的接口自动就是 **public** 的，不能指明为 **private**。

类 **NestingInterfaces** 展示了嵌套接口的不同实现方式。尤其是当实现某个接口时，并不需要实现嵌套在其内部的接口。同时，**private** 接口不能在定义它的类之外被实现。

添加这些特性的最初原因看起来像是出于对严格的语法一致性的考虑，但是我通常认为，一旦你了解了某种特性，就总能找到其用武之地。

11.10 接口和工厂方法模式

接口是多实现的途径，而生成符合某个接口的对象的典型方式是*工厂方法*设计模式。不同于直接调用构造器，只需调用工厂对象中的创建方法就能生成对象的实现——理论上，通过这种方式可以将接口与实现的代码完全分离，使得可以透明地将某个实现替换为另一个实现。这里是一个展示工厂方法结构的例子：

```java
// interfaces/Factories.java
interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Service1 implements Service {
    Service1() {} // Package access

    @Override
    public void method1() {
        System.out.println("Service1 method1");
    }

    @Override
    public void method2() {
        System.out.println("Service1 method2");
```

```
    }
}

class Service1Factory implements ServiceFactory {
    @Override
    public Service getService() {
        return new Service1();
    }
}

class Service2 implements Service {
    Service2() {} // Package access

    @Override
    public void method1() {
        System.out.println("Service2 method1");
    }

    @Override
    public void method2() {
        System.out.println("Service2 method2");
    }
}

class Service2Factory implements ServiceFactory {
    @Override
    public Service getService() {
        return new Service2();
    }
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }

    public static void main(String[] args) {
        serviceConsumer(new Service1Factory());
        // Services are completely interchangeable:
        serviceConsumer(new Service2Factory());
    }
}
```

输出:

```
Service1 method1
Service1 method2
Service2 method1
Service2 method2
```

如果没有工厂方法，代码就必须在某处指定将要创建的 **Service** 的确切类型，从而调用恰当的构造器。

为什么要添加额外的间接层呢？一个常见的原因是创建框架。假设你正在创建一个游戏系统；例如，在相同的棋盘下国际象棋和西洋跳棋：

```
// interfaces/Games.java
// A Game framework using Factory Methods
interface Game {
    boolean move();
}

interface GameFactory {
    Game getGame();
}

class Checkers implements Game {
    private int moves = 0;
    private static final int MOVES = 3;

    @Override
    public boolean move() {
        System.out.println("Checkers move " + moves);
        return ++moves != MOVES;
    }
}

class CheckersFactory implements GameFactory {
    @Override
    public Game getGame() {
        return new Checkers();
    }
}

class Chess implements Game {
    private int moves = 0;
    private static final int MOVES = 4;

    @Override
    public boolean move() {
        System.out.println("Chess move " + moves);
        return ++moves != MOVES;
    }
}

class ChessFactory implements GameFactory {
    @Override
    public Game getGame() {
        return new Chess();
```

```
    }
}

public class Games {
    public static void playGame(GameFactory factory) {
        Game s = factory.getGame();
        while (s.move()) {
            ;
        }
    }

    public static void main(String[] args) {
        playGame(new CheckersFactory());
        playGame(new ChessFactory());
    }
}
```

输出:

```
Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
```

如果类 **Games** 表示一段很复杂的代码，那么这种方式意味着你可以在不同类型的游戏里复用这段代码。你可以再想象一些能够从这个模式中受益的更加精巧的游戏。

在下一章，你将会看到一种更加优雅的使用匿名内部类的工厂实现方式。

11.11 本章小结

认为接口是好的选择，从而使用接口不用具体类，这具有诱惑性。几乎任何时候，创建类都可以替代为创建一个接口和工厂。

很多人都掉进了这个陷阱，只要有可能就创建接口和工厂。这种逻辑看起来像是可能会使用不同的实现，所以总是添加这种抽象性。这变成了一种过早的设计优化。

任何抽象性都应该是由真正的需求驱动的。当有必要时才应该使用接口进行重构，而不是到处添加额外的间接层，从而带来额外的复杂性。这种复杂性非常显著，如果你让某人去处理这种复杂性，只是因为你意识到“以防万一”而添加新接口，而没有其他具有说服力的原因——好吧，如果我碰上了这种设计，就会质疑此人所作的所有其他设计了。

恰当的原则是优先使用类而不是接口。从类开始，如果使用接口的必要性变得很明确，那么就重构。接口是一个伟大的工具，但它们容易被滥用。

12 内部类

> 一个定义在另一个类中的类，叫作内部类。

内部类是一种非常有用的特性，因为它允许你把一些逻辑相关的类组织在一起，并控制位于内部的类的可见性。然而必须要了解，内部类与组合是完全不同的概念，这一点很重要。在最初，内部类看起来就像是一种代码隐藏机制：将类置于其他类的内部。但是，你将会了解到，内部类远不止如此，它了解外围类，并能与之通信，而且你用内部类写出的代码更加优雅而清晰，尽管并不总是这样（而且 Java 8 的 Lambda 表达式和方法引用减少了编写内部类的需求）。

最初，内部类可能看起来有些奇怪，而且要花些时间才能在设计中轻松地使用它们。对内部类的需求并非总是很明显的，但是在描述完内部类的基本语法与语义之后，“Why inner classes?”就应该使得内部类的益处明确显现了。

本章剩余部分包含了对内部类语法更加详尽的探索，这些特性是为了语言的完备性而设计的，但是你也许不需要使用它们，至少一开始不需要。因此，本章最初的部分也许就是你现在所需的全部，你可以将更详尽的探索当作参考资料。

12.1 创建内部类

创建内部类的方式就如同你想的一样——把类的定义置于外围类的里面：

```
// innerclasses/Parcel1.java
// Creating inner classes
public class Parcel1 {
    class Contents {
        private int i = 11;

        public int value() { return i; }
    }

    class Destination {
        private String label;

        Destination(String whereTo) {
            label = whereTo;
        }
```

```java

        String readLabel() { return label; }
    }
    // Using inner classes looks just like
    // using any other class, within Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tasmania");
    }
}
```

输出为:

```
Tasmania
```

当我们在 `ship()` 方法里面使用内部类的时候，与使用普通类没什么不同。在这里，明显的区别只是内部类的名字是嵌套在 **Parcel1** 里面的。

更典型的情况是，外部类将有一个方法，该方法返回一个指向内部类的引用，就像在 `to()` 和 `contents()` 方法中看到的那样:

```java
// innerclasses/Parcel2.java
// Returning a reference to an inner class
public class Parcel2 {
    class Contents {
        private int i = 11;

        public int value() { return i; }
    }

    class Destination {
        private String label;

        Destination(String whereTo) {
            label = whereTo;
        }

        String readLabel() { return label; }
    }

    public Destination to(String s) {
        return new Destination(s);
    }

    public Contents contents() {
```

```
        return new Contents();
    }

    public void ship(String dest) {
        Contents c = contents();
        Destination d = to(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String[] args) {
        Parcel2 p = new Parcel2();
        p.ship("Tasmania");
        Parcel2 q = new Parcel2();
        // Defining references to inner classes:
        Parcel2.Contents c = q.contents();
        Parcel2.Destination d = q.to("Borneo");
    }
}
```

输出为:

```
Tasmania
```

如果想从外部类的非静态方法之外的任意位置创建某个内部类的对象，那么必须像在 `main()` 方法中那样，具体地指明这个对象的类型：*OuterClassName.InnerClassName*。(译者注：在外部类的静态方法中也可以直接指明类型 *InnerClassName*，在其他类中需要指明 *OuterClassName.InnerClassName*。)

12.2 链接外部类

到目前为止，内部类似乎还只是一种名字隐藏和组织代码的模式。这些是很有用，但还不是最引人注目的，它还有其他的用途。当生成一个内部类的对象时，此对象与制造它的外围对象（enclosing object）之间就有了一种联系，所以它能访问其外围对象的所有成员，而不需要任何特殊条件。此外，内部类还拥有其外围类的所有元素的访问权。

```
// innerclasses/Sequence.java
// Holds a sequence of Objects
interface Selector {
    boolean end();
    Object current();
    void next();
}
public class Sequence {
    private Object[] items;
    private int next = 0;
    public Sequence(int size) {
```

```
        items = new Object[size];
    }
    public void add(Object x) {
        if(next < items.length)
            items[next++] = x;
    }
    private class SequenceSelector implements Selector {
        private int i = 0;
        @Override
        public boolean end() { return i == items.length; }
        @Override
        public Object current() { return items[i]; }
        @Override
        public void next() { if(i < items.length) i++; }
    }
    public Selector selector() {
        return new SequenceSelector();
    }
    public static void main(String[] args) {
        Sequence sequence = new Sequence(10);
        for(int i = 0; i < 10; i++)
            sequence.add(Integer.toString(i));
        Selector selector = sequence.selector();
        while(!selector.end()) {
            System.out.print(selector.current() + " ");
            selector.next();
        }
    }
}
```

输出为:

```
0 1 2 3 4 5 6 7 8 9
```

Sequence 类只是一个固定大小的 **Object** 的数组，以类的形式包装了起来。可以调用 `add()` 在序列末尾增加新的 **Object**（只要还有空间），要获取 **Sequence** 中的每一个对象，可以使用 **Selector** 接口。这是“迭代器”设计模式的一个例子，在本书稍后的部分将更多地学习它。**Selector** 允许你检查序列是否到末尾了（`end()`），访问当前对象（`current()`），以及移到序列中的下一个对象（`next()`）。因为 **Selector** 是一个接口，所以别的类可以按它们自己的方式来实现这个接口，并且其他方法能以此接口为参数，来生成更加通用的代码。

这里，**SequenceSelector** 是提供 **Selector** 功能的 **private** 类。可以看到，在 `main()` 中创建了一个 **Sequence**，并向其中添加了一些 **String** 对象。然后通过调用 `selector()` 获取一个 **Selector**，并用它在 **Sequence** 中移动和选择每一个元素。最初看到 **SequenceSelector**，可能会觉得它只不过是另一个内部类罢了。但请仔细观察它，注意方法 `end()`，`current()` 和 `next()` 都用到了 **items**，这是一个引用，它并不是 **SequenceSelector** 的一部分，而是外围类中的一个 **private** 字段。然而内部类可以访问其外围类的方法和字段，就像自己拥有它们似的，这带来了很大的方便，就如前面的例子所示。

所以内部类自动拥有对其外围类所有成员的访问权。这是如何做到的呢？当某个外围类的对象创建了一个内部类对象时，此内部类对象必定会秘密地捕获一个指向那个外围类对象的引用。然后，在你访问此外围类的成员时，就是用那个引用来选择外围类的成员。幸运的是，编译器会帮你处理所有的细节，但你现在可以看到：内部类的对象只能在与其外围类的对象相关联的情况下才能被创建（就像你应该看到的，内部类是非 **static** 类时）。构建内部类对象时，需要一个指向其外围类对象的引用，如果编译器访问不到这个引用就会报错。不过绝大多数时候这都无需程序员操心。

12.3 使用.this 和.new

如果你需要生成对外部类对象的引用，可以使用外部类的名字后面紧跟圆点和 **this**。这样产生的引用自动地具有正确的类型，这一点在编译期就被知晓并受到检查，因此没有任何运行时开销。下面的示例展示了如何使用 **.this**：

```java
// innerclasses/DotThis.java
// Accessing the outer-class object
public class DotThis {
    void f() { System.out.println("DotThis.f()"); }

    public class Inner {
        public DotThis outer() {
            return DotThis.this;
            // A plain "this" would be Inner's "this"
        }
    }

    public Inner inner() { return new Inner(); }

    public static void main(String[] args) {
        DotThis dt = new DotThis();
        DotThis.Inner dti = dt.inner();
        dti.outer().f();
    }
}
```

输出为：

```
DotThis.f()
```

有时你可能想要告知某些其他对象，去创建其某个内部类的对象。要实现此目的，你必须在 **new** 表达式中提供对其他外部类对象的引用，这是需要使用 **.new** 语法，就像下面这样：

```java
// innerclasses/DotNew.java
// Creating an inner class directly using .new syntax
public class DotNew {
    public class Inner {}
```

```
    public static void main(String[] args) {
        DotNew dn = new DotNew();
        DotNew.Inner dni = dn.new Inner();
    }
}
```

要想直接创建内部类的对象，你不能按照你想象的方式，去引用外部类的名字 **DotNew**，而是必须使用外部类的对象来创建该内部类对象，就像在上面的程序中所看到的那样。这也解决了内部类名字作用域的问题，因此你不必声明（实际上你不能声明）dn.new DotNew.Inner。

下面你可以看到将 **.new** 应用于 Parcel 的示例：

```
// innerclasses/Parcel3.java
// Using .new to create instances of inner classes
public class Parcel3 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) { label = whereTo; }
        String readLabel() { return label; }
    }
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        // Must use instance of outer class
        // to create an instance of the inner class:
        Parcel3.Contents c = p.new Contents();
        Parcel3.Destination d =
                p.new Destination("Tasmania");
    }
}
```

在拥有外部类对象之前是不可能创建内部类对象的。这是因为内部类对象会暗暗地连接到建它的外部类对象上。但是，如果你创建的是嵌套类（静态内部类），那么它就不需要对外部类对象的引用。

12.4 内部类与向上转型

当将内部类向上转型为其基类，尤其是转型为一个接口的时候，内部类就有了用武之地。（从实现了某个接口的对象，得到对此接口的引用，与向上转型为这个对象的基类，实质上效果是一样的。）这是因为此内部类-某个接口的实现-能够完全不可见，并且不可用。所得到的只是指向基类或接口的引用，所以能够很方便地隐藏实现细节。

我们可以创建前一个示例的接口：

```
// innerclasses/Destination.java
```

```
public interface Destination {
    String readLabel();
}
```

```
// innerclasses/Contents.java
public interface Contents {
    int value();
}
```

现在 **Contents** 和 **Destination** 表示客户端程序员可用的接口。记住，接口的所有成员自动被设置为 **public**。

当取得了一个指向基类或接口的引用时，甚至可能无法找出它确切的类型，看下面的例子：

```
// innerclasses/TestParcel.java
class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        @Override
        public int value() { return i; }
    }
    protected final class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        @Override
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}
public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("Tasmania");
        // Illegal -- can't access private class:
        //- Parcel4.PContents pc = p.new PContents();
    }
}
```

在 **Parcel4** 中，内部类 **PContents** 是 **private**，所以除了 **Parcel4**，没有人能访问它。普通（非内部）类的访问权限不能被设为 **private** 或者 **protected**；他们只能设置为 **public** 或 **package** 访问权限。

PDestination 是 **protected**，所以只有 **Parcel4** 及其子类、还有与 **Parcel4** 同一个包中的类（因为 **protected** 也给予了包访问权）能访问 **PDestination**，其他类都不能访问 **PDestination**，这意味着，如果客户端程序员想了解或访问这些成员，那是要受到限制的。实际上，甚至不能向下转型成 **private** 内部类（或 **protected** 内部类，除非是继承自它的子类），因为不能访问其名字，就像在 **TestParcel** 类中看到的那样。

private 内部类给类的设计者提供了一种途径，通过这种方式可以完全阻止任何依赖于类型的编码，并且完全隐藏了实现的细节。此外，从客户端程序员的角度来看，由于不能访问任何新增加的、原本不属于公共接口的方法，所以扩展接口是没有价值的。这也给 Java 编译器提供了生成高效代码的机会。

12.5 内部类方法和作用域

到目前为止，读者所看到的只是内部类的典型用途。通常，如果所读、写的代码包含了内部类，那么它们都是“平凡的”内部类，简单并且容易理解。然而，内部类的语法覆盖了大量其他的更加难以理解的技术。例如，可以在一个方法里面或者在任意的作用域内定义内部类。

这么做有两个理由：

1. 如前所示，你实现了某类型的接口，于是可以创建并返回对其的引用。
2. 你要解决一个复杂的问题，想创建一个类来辅助你的解决方案，但是又不希望这个类是公共可用的。

在后面的例子中，先前的代码将被修改，以用来实现：

1. 一个定义在方法中的类。
2. 一个定义在作用域内的类，此作用域在方法的内部。
3. 一个实现了接口的匿名类。
4. 一个匿名类，它扩展了没有默认构造器的类。
5. 一个匿名类，它执行字段初始化。
6. 一个匿名类，它通过实例初始化实现构造（匿名内部类不可能有构造器）。

第一个例子展示了在方法的作用域内（而不是在其他类的作用域内）创建一个完整的类。这被称作局部内部类：

```
// innerclasses/Parcel5.java
// Nesting a class within a method
public class Parcel5 {
    public Destination destination(String s) {
        final class PDestination implements Destination {
            private String label;

            private PDestination(String whereTo) {
                label = whereTo;
```

```
            }

            @Override
            public String readLabel() { return label; }
        }
        return new PDestination(s);
    }

    public static void main(String[] args) {
        Parcel5 p = new Parcel5();
        Destination d = p.destination("Tasmania");
    }
}
```

PDestination 类是 `destination()` 方法的一部分，而不是 **Parcel5** 的一部分。所以，在 `destination()` 之外不能访问 **PDestination**，注意出现在 **return** 语句中的向上转型-返回的是 **Destination** 的引用，它是 **PDestination** 的基类。当然，在 `destination()` 中定义了内部类 **PDestination**，并不意味着一旦 `dest()` 方法执行完毕，**PDestination** 就不可用了。

你可以在同一个子目录下的任意类中对某个内部类使用类标识符 **PDestination**，这并不会有命名冲突。

下面的例子展示了如何在任意的作用域内嵌入一个内部类：

```
// innerclasses/Parcel6.java
// Nesting a class within a scope
public class Parcel6 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
                String getSlip() { return id; }
            }
            TrackingSlip ts = new TrackingSlip("slip");
            String s = ts.getSlip();
        }
        // Can't use it here! Out of scope:
        //- TrackingSlip ts = new TrackingSlip("x");
    }
    public void track() { internalTracking(true); }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        p.track();
    }
}
```

TrackingSlip 类被嵌入在 **if** 语句的作用域内，这并不是说该类的创建是有条件的，它其实与别

的类一起编译过了。然而，在定义 **Trackingslip** 的作用域之外，它是不可用的，除此之外，它与普通的类一样。

12.6 匿名内部类

下面的例子看起来有点奇怪：

```
// innerclasses/Parcel7.java
// Returning an instance of an anonymous inner class
public class Parcel7 {
    public Contents contents() {
        return new Contents() { // Insert class definition
            private int i = 11;

            @Override
            public int value() { return i; }
        }; // Semicolon required
    }

    public static void main(String[] args) {
        Parcel7 p = new Parcel7();
        Contents c = p.contents();
    }
}
```

`contents()` 方法将返回值的生成与表示这个返回值的类的定义结合在一起！另外，这个类是匿名的，它没有名字。更糟的是，看起来似乎是你正要创建一个 **Contents** 对象。但是然后（在到达语句结束的分号之前）你却说："等一等，我想在这里插入一个类的定义。

这种奇怪的语法指的是："创建一个继承自 **Contents** 的匿名类的对象。"通过 **new** 表达式返回的引用被自动向上转型为对 **Contents** 的引用。上述匿名内部类的语法是下述形式的简化形式：

```
// innerclasses/Parcel7b.java
// Expanded version of Parcel7.java
public class Parcel7b {
    class MyContents implements Contents {
        private int i = 11;
        @Override
        public int value() { return i; }
    }

    public Contents contents() {
        return new MyContents();
    }

    public static void main(String[] args) {
        Parcel7b p = new Parcel7b();
```

```
        Contents c = p.contents();
    }
}
```

在这个匿名内部类中，使用了默认的构造器来生成 **Contents**。下面的代码展示的是，如果你的基类需要一个有参数的构造器，应该怎么办：

```
// innerclasses/Parcel8.java
// Calling the base-class constructor
public class Parcel8 {
    public Wrapping wrapping(int x) {
        // Base constructor call:
        return new Wrapping(x) { // [1]
            @Override
            public int value() {
                return super.value() * 47;
            }
        }; // [2]
    }
    public static void main(String[] args) {
        Parcel8 p = new Parcel8();
        Wrapping w = p.wrapping(10);
    }
}
```

- [1] 将合适的参数传递给基类的构造器。
- [2] 在匿名内部类末尾的分号，并不是用来标记此内部类结束的。实际上，它标记的是表达式的结束，只不过这个表达式正巧包含了匿名内部类罢了。因此，这与别的地方使用的分号是一致的。

尽管 **Wrapping** 只是一个具有具体实现的普通类，但它还是被导出类当作公共“接口”来使用。

```
// innerclasses/Wrapping.java
public class Wrapping {
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
}
```

为了多样性，**Wrapping** 拥有一个要求传递一个参数的构造器。

在匿名类中定义字段时，还能够对其执行初始化操作：

```
// innerclasses/Parcel9.java
public class Parcel9 {
    // Argument must be final or "effectively final"
    // to use within the anonymous inner class:
    public Destination destination(final String dest) {
        return new Destination() {
            private String label = dest;
            @Override
            public String readLabel() { return label; }
```

```
        };
    }
    public static void main(String[] args) {
        Parcel9 p = new Parcel9();
        Destination d = p.destination("Tasmania");
    }
}
```

如果定义一个匿名内部类，并且希望它使用一个在其外部定义的对象，那么编译器会要求其参数引用是 **final** 的（也就是说，它在初始化后不会改变，所以可以被当作 **final**），就像你在 `destination()` 的参数中看到的那样。这里省略掉 **final** 也没问题，但是通常最好加上 **final** 作为一种暗示。

如果只是简单地给一个字段赋值，那么此例中的方法是很好的。但是，如果想做一些类似构造器的行为，该怎么办呢？在匿名类中不可能有命名构造器（因为它根本没名字!），但通过实例初始化，就能够达到为匿名内部类创建一个构造器的效果，就像这样：

```
// innerclasses/AnonymousConstructor.java
// Creating a constructor for an anonymous inner class
abstract class Base {
    Base(int i) {
        System.out.println("Base constructor, i = " + i);
    }
    public abstract void f();
}
public class AnonymousConstructor {
    public static Base getBase(int i) {
        return new Base(i) {
            { System.out.println(
                    "Inside instance initializer"); }
            @Override
            public void f() {
                System.out.println("In anonymous f()");
            }
        };
    }
    public static void main(String[] args) {
        Base base = getBase(47);
        base.f();
    }
}
```

输出为：

```
Base constructor, i = 47
Inside instance initializer
In anonymous f()
```

在此例中，不要求变量一定是 **final** 的。因为被传递给匿名类的基类的构造器，它并不会在匿名类内部被直接使用。

下例是带实例初始化的“parcel”形式。注意 `destination()` 的参数必须是 **final** 的，因为它们是在匿名类内部使用的（译者注：即使不加 **final**, Java 8 的编译器也会为我们自动加上 **final**，以保证数据的一致性）。

```
// innerclasses/Parcel10.java
// Using "instance initialization" to perform
// construction on an anonymous inner class
public class Parcel10 {
    public Destination
    destination(final String dest, final float price) {
        return new Destination() {
            private int cost;
            // Instance initialization for each object:
            {
                cost = Math.round(price);
                if(cost > 100)
                    System.out.println("Over budget!");
            }
            private String label = dest;
            @Override
            public String readLabel() { return label; }
        };
    }
    public static void main(String[] args) {
        Parcel10 p = new Parcel10();
        Destination d = p.destination("Tasmania", 101.395F);
    }
}
```

输出为：

```
Over budget!
```

在实例初始化操作的内部，可以看到有一段代码，它们不能作为字段初始化动作的一部分来执行（就是 **if** 语句）。所以对于匿名类而言，实例初始化的实际效果就是构造器。当然它受到了限制-你不能重载实例初始化方法，所以你仅有一个这样的构造器。

匿名内部类与正规的继承相比有些受限，因为匿名内部类既可以扩展类，也可以实现接口，但是不能两者兼备。而且如果是实现接口，也只能实现一个接口。

12.7 嵌套类

如果不需要内部类对象与其外围类对象之间有联系，那么可以将内部类声明为 **static**，这通常称为嵌套类。想要理解 **static** 应用于内部类时的含义，就必须记住，普通的内部类对象隐式地保存了一个引用，指向创建它的外围类对象。然而，当内部类是 **static** 的时，就不是这样了。嵌套类意味着：

1. 要创建嵌套类的对象，并不需要其外围类的对象。

2. 不能从嵌套类的对象中访问非静态的外围类对象。

嵌套类与普通的内部类还有一个区别。普通内部类的字段与方法，只能放在类的外部层次上，所以普通的内部类不能有 **static** 数据和 **static** 字段，也不能包含嵌套类。但是嵌套类可以包含所有这些东西：

```java
// innerclasses/Parcel11.java
// Nested classes (static inner classes)
public class Parcel11 {
    private static class ParcelContents implements Contents {
        private int i = 11;
        @Override
        public int value() { return i; }
    }
    protected static final class ParcelDestination
            implements Destination {
        private String label;
        private ParcelDestination(String whereTo) {
            label = whereTo;
        }
        @Override
        public String readLabel() { return label; }
        // Nested classes can contain other static elements:
        public static void f() {}
        static int x = 10;
        static class AnotherLevel {
            public static void f() {}
            static int x = 10;
        }
    }
    public static Destination destination(String s) {
        return new ParcelDestination(s);
    }
    public static Contents contents() {
        return new ParcelContents();
    }
    public static void main(String[] args) {
        Contents c = contents();
        Destination d = destination("Tasmania");
    }
}
```

在 `main()` 中，没有任何 **Parcel11** 的对象是必需的；而是使用选取 **static** 成员的普通语法来调用方法-这些方法返回对 **Contents** 和 **Destination** 的引用。

就像你在本章前面看到的那样，在一个普通的（非 **static**）内部类中，通过一个特殊的 **this** 引用可以链接到其外围类对象。嵌套类就没有这个特殊的 **this** 引用，这使得它类似于一个 **static** 方法。

12.7.1 接口内部的类

嵌套类可以作为接口的一部分。你放到接口中的任何类都自动地是 **public** 和 **static** 的。因为类是 **static** 的，只是将嵌套类置于接口的命名空间内，这并不违反接口的规则。你甚至可以在内部类中实现其外围接口，就像下面这样：

```
// innerclasses/ClassInInterface.java
// {java ClassInInterface$Test}
public interface ClassInInterface {
    void howdy();
    class Test implements ClassInInterface {
        @Override
        public void howdy() {
            System.out.println("Howdy!");
        }
        public static void main(String[] args) {
            new Test().howdy();
        }
    }
}
```

输出为：

```
Howdy!
```

如果你想要创建某些公共代码，使得它们可以被某个接口的所有不同实现所共用，那么使用接口内部的嵌套类会显得很方便。

我曾在本书中建议过，在每个类中都写一个 `main()` 方法，用来测试这个类。这样做有一个缺点，那就是必须带着那些已编译过的额外代码。如果这对你是个麻烦，那就可以使用嵌套类来放置测试代码。

```
// innerclasses/TestBed.java
// Putting test code in a nested class
// {java TestBed$Tester}
public class TestBed {
    public void f() { System.out.println("f()"); }
    public static class Tester {
        public static void main(String[] args) {
            TestBed t = new TestBed();
            t.f();
        }
    }
}
```

输出为：

```
f()
```

这生成了一个独立的类 **TestBed**$Tester**$ $**javaTestBed$**Tester**，在 Unix/Linux 系统中需要转义 $**$ $**TestBed$**Tester.class**。

12.7.2 从多层嵌套类中访问外部类的成员

一个内部类被嵌套多少层并不重要——它能透明地访问所有它所嵌入的外围类的所有成员，如下所示：

```
// innerclasses/MultiNestingAccess.java
// Nested classes can access all members of all
// levels of the classes they are nested within
class MNA {
    private void f() {}
    class A {
        private void g() {}
        public class B {
            void h() {
                g();
                f();
            }
        }
    }
}
public class MultiNestingAccess {
    public static void main(String[] args) {
        MNA mna = new MNA();
        MNA.A mnaa = mna.new A();
        MNA.A.B mnaab = mnaa.new B();
        mnaab.h();
    }
}
```

可以看到在 **MNA.A.B** 中，调用方法 `g()` 和 `f()` 不需要任何条件（即使它们被定义为 **private**）。这个例子同时展示了如何从不同的类里创建多层嵌套的内部类对象的基本语法。“**.new**”语法能产生正确的作用域，所以不必在调用构造器时限定类名。

12.8 为什么需要内部类

至此，我们已经看到了许多描述内部类的语法和语义，但是这并不能回答“为什么需要内部类”这个问题。那么，Java 设计者们为什么会如此费心地增加这项基本的语言特性呢？

一般说来，内部类继承自某个类或实现某个接口，内部类的代码操作创建它的外围类的对象。所以可以认为内部类提供了某种进入其外围类的窗口。

内部类必须要回答的一个问题是：如果只是需要一个对接口的引用，为什么不通过外围类实现那个

接口呢？答案是："如果这能满足需求，那么就应该这样做。"那么内部类实现一个接口与外围类实现这个接口有什么区别呢？答案是：后者不是总能享用到接口带来的方便，有时需要用到接口的实现。所以，使用内部类最吸引人的原因是：

> 每个内部类都能独立地继承自一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。

如果没有内部类提供的、可以继承多个具体的或抽象的类的能力，一些设计与编程问题就很难解决。从这个角度看，内部类使得多重继承的解决方案变得完整。接口解决了部分问题，而内部类有效地实现了"多重继承"。也就是说，内部类允许继承多个非接口类型（译注：类或抽象类）。

为了看到更多的细节，让我们考虑这样一种情形：即必须在一个类中以某种方式实现两个接口。由于接口的灵活性，你有两种选择；使用单一类，或者使用内部类：

```
// innerclasses/mui/MultiInterfaces.java
// Two ways a class can implement multiple interfaces
// {java innerclasses.mui.MultiInterfaces}
package innerclasses.mui;
interface A {}
interface B {}
class X implements A, B {}
class Y implements A {
    B makeB() {
        // Anonymous inner class:
        return new B() {};
    }
}
public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        takesA(x);
        takesA(y);
        takesB(x);
        takesB(y.makeB());
    }
}
```

当然，这里假设在两种方式下的代码结构都确实有逻辑意义。然而遇到问题的时候，通常问题本身就能给出某些指引，告诉你是应该使用单一类，还是使用内部类。但如果没有任何其他限制，从实现的观点来看，前面的例子并没有什么区别，它们都能正常运作。

如果拥有的是抽象的类或具体的类，而不是接口，那就只能使用内部类才能实现多重继承：

```
// innerclasses/MultiImplementation.java
// For concrete or abstract classes, inner classes
// produce "multiple implementation inheritance"
```

```
// {java innerclasses.MultiImplementation}
package innerclasses;

class D {}

abstract class E {}

class Z extends D {
    E makeE() {
      return new E() {};
    }
}

public class MultiImplementation {
    static void takesD(D d) {}
    static void takesE(E e) {}

    public static void main(String[] args) {
        Z z = new Z();
        takesD(z);
        takesE(z.makeE());
    }
}
```

如果不需要解决“多重继承”的问题，那么自然可以用别的方式编码，而不需要使用内部类。但如果使用内部类，还可以获得其他一些特性：

1. 内部类可以有多个实例，每个实例都有自己的状态信息，并且与其外围类对象的信息相互独立。
2. 在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或继承同一个类。稍后就会展示一个这样的例子。
3. 创建内部类对象的时刻并不依赖于外围类对象的创建
4. 内部类并没有令人迷惑的“is-a”关系，它就是一个独立的实体。

举个例子，如果 **Sequence.java** 不使用内部类，就必须声明“**Sequence** 是一个 **Selector**”，对于某个特定的 **Sequence** 只能有一个 **Selector**，然而使用内部类很容易就能拥有另一个方法 `reverseSelector()`，用它来生成一个反方向遍历序列的 **Selector**，只有内部类才有这种灵活性。

12.8.1 闭包与回调

闭包（**closure**）是一个可调用的对象，它记录了一些信息，这些信息来自于创建它的作用域。通过这个定义，可以看出内部类是面向对象的闭包，因为它不仅包含外围类对象（创建内部类的作用域）的信息，还自动拥有一个指向此外围类对象的引用，在此作用域内，内部类有权操作所有的成员，包括 **private** 成员。

在 Java 8 之前，生成闭包行为的唯一方式就是内部类。在 Java 8 之后，我们可以使用 lambda 表达式来生成闭包行为，并且语法更加精细和简洁；你将会在 函数式编程 这一章节中学习相关细节。即使

估算, 以避免发生意外

应该优先使用 lambda 表达式用于内部类闭包，你依旧会看到那些 Java 8 以前的代码，即使用内部类来表示闭包的方式，所以非常有必要来理解这种形式。

Java 最引人争议的问题之一就是，人们认为 Java 应该包含某种类似指针的机制，以允许回调（callback）。通过回调，对象能够携带一些信息，这些信息允许它在稍后的某个时刻调用初始的对象。稍后将会看到这是一个非常有用的概念。如果回调是通过指针实现的，那么就只能寄希望于程序员不会误用该指针。然而，读者应该已经了解到，Java 更小心仔细，所以没有在语言中包括指针。

通过内部类提供闭包的功能是优良的解决方案，它比指针更灵活、更安全。见下例：

```java
// innerclasses/Callbacks.java
// Using inner classes for callbacks
// {java innerclasses.Callbacks}
package innerclasses;
interface Incrementable {
    void increment();
}
// Very simple to just implement the interface:
class Callee1 implements Incrementable {
    private int i = 0;
    @Override
    public void increment() {
        i++;
        System.out.println(i);
    }
}
class MyIncrement {
    public void increment() {
        System.out.println("Other operation");
    }
    static void f(MyIncrement mi) { mi.increment(); }
}
// If your class must implement increment() in
// some other way, you must use an inner class:
class Callee2 extends MyIncrement {
    private int i = 0;
    @Override
    public void increment() {
        super.increment();
        i++;
        System.out.println(i);
    }
    private class Closure implements Incrementable {
        @Override
        public void increment() {
            // Specify outer-class method, otherwise
            // you'll get an infinite recursion:
            Callee2.this.increment();
        }
    }
```

```
    Incrementable getCallbackReference() {
        return new Closure();
    }
}
class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) {
        callbackReference = cbh;
    }
    void go() { callbackReference.increment(); }
}
public class Callbacks {
    public static void main(String[] args) {
        Callee1 c1 = new Callee1();
        Callee2 c2 = new Callee2();
        MyIncrement.f(c2);
        Caller caller1 = new Caller(c1);
        Caller caller2 =
                new Caller(c2.getCallbackReference());
        caller1.go();
        caller1.go();
        caller2.go();
        caller2.go();
    }
}
```

输出为:

```
Other operation
1
1
2
Other operation
2
Other operation
3
```

这个例子进一步展示了外围类实现一个接口与内部类实现此接口之间的区别。就代码而言，**Callee1** 是更简单的解决方式。**Callee2** 继承自 **MyIncrement**，后者已经有了一个不同的 `increment()` 方法，并且与 **Incrementable** 接口期望的 `increment()` 方法完全不相关。所以如果 **Callee2** 继承了 **MyIncrement**，就不能为了 **Incrementable** 的用途而覆盖 `increment()` 方法，于是只能使用内部类独立地实现 **Incrementable**，还要注意，当创建了一个内部类时，并没有在外围类的接口中添加东西，也没有修改外围类的接口。

注意，在 **Callee2** 中除了 `getCallbackReference()` 以外，其他成员都是 **private** 的。要想建立与外部世界的任何连接，接口 **Incrementable** 都是必需的。在这里可以看到，**interface** 是如何允许接口与接口的实现完全独立的。内部类 **Closure** 实现了 **Incrementable**，以提供一个返回 **Callee2** 的“钩子”（hook）-而且是一个安全的钩子。无论谁获得此 **Incrementable** 的引用，都只能

调用 `increment()`，除此之外没有其他功能（不像指针那样，允许你做很多事情）。

Caller 的构造器需要一个 **Incrementable** 的引用作为参数（虽然可以在任意时刻捕获回调引用），然后在以后的某个时刻，**Caller** 对象可以使用此引用回调 **Callee** 类。

回调的价值在于它的灵活性-可以在运行时动态地决定需要调用什么方法。例如，在图形界面实现 GUI 功能的时候，到处都用到回调。

12.8.2 内部类与控制框架

在将要介绍的控制框架（control framework）中，可以看到更多使用内部类的具体例子。

应用程序框架（application framework）就是被设计用以解决某类特定问题的一个类或一组类。要运用某个应用程序框架，通常是继承一个或多个类，并覆盖某些方法。在覆盖后的方法中，编写代码定制应用程序框架提供的通用解决方案，以解决你的特定问题。这是设计模式中模板方法的一个例子，模板方法包含算法的基本结构，并且会调用一个或多个可覆盖的方法，以完成算法的动作。设计模式总是将变化的事物与保持不变的事物分离开，在这个模式中，模板方法是保持不变的事物，而可覆盖的方法就是变化的事物。

控制框架是一类特殊的应用程序框架，它用来解决响应事件的需求。主要用来响应事件的系统被称作*事件驱动系统*。应用程序设计中常见的问题之一是图形用户接口（GUI），它几乎完全是事件驱动的系统。

要理解内部类是如何允许简单的创建过程以及如何使用控制框架的，请考虑这样一个控制框架，它的工作就是在事件“就绪”的时候执行事件。虽然“就绪”可以指任何事，但在本例中是指基于时间触发的事件。接下来的问题就是，对于要控制什么，控制框架并不包含任何具体的信息。那些信息是在实现算法的 `action()` 部分时，通过继承来提供的。

首先，接口描述了要控制的事件。因为其默认的行为是基于时间去执行控制，所以使用抽象类代替实际的接口。下面的例子包含了某些实现：

```java
// innerclasses/controller/Event.java
// The common methods for any control event
package innerclasses.controller;
import java.time.*; // Java 8 time classes
public abstract class Event {
    private Instant eventTime;
    protected final Duration delayTime;
    public Event(long millisecondDelay) {
        delayTime = Duration.ofMillis(millisecondDelay);
        start();
    }
    public void start() { // Allows restarting
        eventTime = Instant.now().plus(delayTime);
    }
    public boolean ready() {
        return Instant.now().isAfter(eventTime);
```

```
    }
    public abstract void action();
}
```

当希望运行 **Event** 并随后调用 `start()` 时，那么构造器就会捕获（从对象创建的时刻开始的）时间，此时间是这样得来的：`start()` 获取当前时间，然后加上一个延迟时间，这样生成触发事件的时间。`start()` 是一个独立的方法，而没有包含在构造器内，因为这样就可以在事件运行以后重新启动计时器，也就是能够重复使用 **Event** 对象。例如，如果想要重复一个事件，只需简单地在 `action()` 中调用 `start()` 方法。

`ready()` 告诉你何时可以运行 `action()` 方法了。当然，可以在派生类中覆盖 `ready()` 方法，使得 **Event** 能够基于时间以外的其他因素而触发。

下面的文件包含了一个用来管理并触发事件的实际控制框架。**Event** 对象被保存在 **List<Event>** 类型（读作“Event 的列表”）的容器对象中，容器会在 集合 中详细介绍。目前读者只需要知道 `add()` 方法用来将一个 **Event** 添加到 **List** 的尾端，`size()` 方法用来得到 **List** 中元素的个数，foreach 语法用来连续获联 **List** 中的 **Event**，`remove()` 方法用来从 **List** 中移除指定的 **Event**。

```
// innerclasses/controller/Controller.java
// The reusable framework for control systems
package innerclasses.controller;
import java.util.*;
public class Controller {
    // A class from java.util to hold Event objects:
    private List<Event> eventList = new ArrayList<>();
    public void addEvent(Event c) { eventList.add(c); }
    public void run() {
        while(eventList.size() > 0)
            // Make a copy so you're not modifying the list
            // while you're selecting the elements in it:
            for(Event e : new ArrayList<>(eventList))
                if(e.ready()) {
                    System.out.println(e);
                    e.action();
                    eventList.remove(e);
                }
    }
}
```

`run()` 方法循环遍历 **eventList**，寻找就绪的（`ready()`）、要运行的 **Event** 对象。对找到的每一个就绪的（`ready()`）事件，使用对象的 `toString()` 打印其信息，调用其 `action()` 方法，然后从列表中移除此 **Event**。

注意，在目前的设计中你并不知道 **Event** 到底做了什么。这正是此设计的关键所在—“使变化的事物与不变的事物相互分离”。用我的话说，“变化向量”就是各种不同的 **Event** 对象所具有的不同行为，而你通过创建不同的 **Event** 子类来表现不同的行为。

这正是内部类要做的事情，内部类允许：

1. 控制框架的完整实现是由单个的类创建的，从而使得实现的细节被封装了起来。内部类用来表示解决问题所必需的各种不同的 `action()`。
2. 内部类能够很容易地访问外围类的任意成员，所以可以避免这种实现变得笨拙。如果没有这种能力，代码将变得令人讨厌，以至于你肯定会选择别的方法。

考虑此控制框架的一个特定实现，如控制温室的运作：控制灯光、水、温度调节器的开关，以及响铃和重新启动系统，每个行为都是完全不同的。控制框架的设计使得分离这些不同的代码变得非常容易。使用内部类，可以在单一的类里面产生对同一个基类 **Event** 的多种派生版本。对于温室系统的每一种行为，都继承创建一个新的 **Event** 内部类，并在要实现的 `action()` 中编写控制代码。

作为典型的应用程序框架，**GreenhouseControls** 类继承自 **Controller**：

```java
// innerclasses/GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import innerclasses.controller.*;
public class GreenhouseControls extends Controller {
    private boolean light = false;
    public class LightOn extends Event {
        public LightOn(long delayTime) {
            super(delayTime);
        }
        @Override
        public void action() {
            // Put hardware control code here to
            // physically turn on the light.
            light = true;
        }
        @Override
        public String toString() {
            return "Light is on";
        }
    }
    public class LightOff extends Event {
        public LightOff(long delayTime) {
            super(delayTime);
        }
        @Override
        public void action() {
            // Put hardware control code here to
            // physically turn off the light.
            light = false;
        }
        @Override
        public String toString() {
            return "Light is off";
        }
```

```
    }
    private boolean water = false;
    public class WaterOn extends Event {
        public WaterOn(long delayTime) {
            super(delayTime);
        }
        @Override
        public void action() {
            // Put hardware control code here.
            water = true;
        }
        @Override
        public String toString() {
            return "Greenhouse water is on";
        }
    }
    public class WaterOff extends Event {
        public WaterOff(long delayTime) {
            super(delayTime);
        }
        @Override
        public void action() {
            // Put hardware control code here.
            water = false;
        }
        @Override
        public String toString() {
            return "Greenhouse water is off";
        }
    }
    private String thermostat = "Day";
    public class ThermostatNight extends Event {
        public ThermostatNight(long delayTime) {
            super(delayTime);
        }
        @Override
        public void action() {
            // Put hardware control code here.
            thermostat = "Night";
        }
        @Override
        public String toString() {
            return "Thermostat on night setting";
        }
    }
    public class ThermostatDay extends Event {
        public ThermostatDay(long delayTime) {
            super(delayTime);
        }
        @Override
        public void action() {
```

```
            // Put hardware control code here.
            thermostat = "Day";
        }
        @Override
        public String toString() {
            return "Thermostat on day setting";
        }
    }
    // An example of an action() that inserts a
    // new one of itself into the event list:
    public class Bell extends Event {
        public Bell(long delayTime) {
            super(delayTime);
        }
        @Override
        public void action() {
            addEvent(new Bell(delayTime.toMillis()));
        }
        @Override
        public String toString() {
            return "Bing!";
        }
    }
    public class Restart extends Event {
        private Event[] eventList;
        public
        Restart(long delayTime, Event[] eventList) {
            super(delayTime);
            this.eventList = eventList;
            for(Event e : eventList)
                addEvent(e);
        }
        @Override
        public void action() {
            for(Event e : eventList) {
                e.start(); // Rerun each event
                addEvent(e);
            }
            start(); // Rerun this Event
            addEvent(this);
        }
        @Override
        public String toString() {
            return "Restarting system";
        }
    }
    public static class Terminate extends Event {
        public Terminate(long delayTime) {
            super(delayTime);
        }
        @Override
```

```
        public void action() { System.exit(0); }
        @Override
        public String toString() {
            return "Terminating";
        }
    }
}
```

注意，**light**，**water** 和 **thermostat** 都属于外围类 **GreenhouseControls**，而这些内部类能够自由地访问那些字段，无需限定条件或特殊许可。而且，`action()` 方法通常都涉及对某种硬件的控制。

大多数 **Event** 类看起来都很相似，但是 **Bell** 和 **Restart** 则比较特别。**Bell** 控制响铃，然后在事件列表中增加一个 **Bell** 对象，于是过一会儿它可以再次响铃。读者可能注意到了内部类是多么像多重继承：**Bell** 和 **Restart** 有 **Event** 的所有方法，并且似乎也拥有外围类 **GreenhouseContrlos** 的所有方法。

一个由 **Event** 对象组成的数组被递交给 **Restart**，该数组要加到控制器上。由于 `Restart()` 也是一个 **Event** 对象，所以同样可以将 **Restart** 对象添加到 `Restart.action()` 中，以使系统能够有规律地重新启动自己。

下面的类通过创建一个 **GreenhouseControls** 对象，并添加各种不同的 **Event** 对象来配置该系统，这是命令设计模式的一个例子—**eventList** 中的每个对象都被封装成对象的请求：

```
// innerclasses/GreenhouseController.java
// Configure and execute the greenhouse system
import innerclasses.controller.*;
public class GreenhouseController {
    public static void main(String[] args) {
        GreenhouseControls gc = new GreenhouseControls();
        // Instead of using code, you could parse
        // configuration information from a text file:
        gc.addEvent(gc.new Bell(900));
        Event[] eventList = {
                gc.new ThermostatNight(0),
                gc.new LightOn(200),
                gc.new LightOff(400),
                gc.new WaterOn(600),
                gc.new WaterOff(800),
                gc.new ThermostatDay(1400)
        };
        gc.addEvent(gc.new Restart(2000, eventList));
        gc.addEvent(
                new GreenhouseControls.Terminate(5000));
        gc.run();
    }
}
```

输出为：

```
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Greenhouse water is off
Bing!
Thermostat on day setting
Bing!
Restarting system
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Bing!
Greenhouse water is off
Thermostat on day setting
Bing!
Restarting system
Thermostat on night setting
Light is on
Light is off
Bing!
Greenhouse water is on
Greenhouse water is off
Terminating
```

这个类的作用是初始化系统，所以它添加了所有相应的事件。**Restart** 事件反复运行，而且它每次都会将 **eventList** 加载到 **GreenhouseControls** 对象中。如果提供了命令行参数，系统会以它作为毫秒数，决定什么时候终止程序（这是测试程序时使用的）。

当然，更灵活的方法是避免对事件进行硬编码。

这个例子应该使读者更了解内部类的价值了，特别是在控制框架中使用内部类的时候。

12.9 继承内部类

因为内部类的构造器必须连接到指向其外围类对象的引用，所以在继承内部类的时候，事情会变得有点复杂。问题在于，那个指向外围类对象的“秘密的”引用必须被初始化，而在派生类中不再存在可连接的默认对象。要解决这个问题，必须使用特殊的语法来明确说清它们之间的关联：

```java
// innerclasses/InheritInner.java
// Inheriting an inner class
class WithInner {
    class Inner {}
}
public class InheritInner extends WithInner.Inner {
    //- InheritInner() {} // Won't compile
    InheritInner(WithInner wi) {
```

```
        wi.super();
    }
    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
}
```

可以看到，**InheritInner** 只继承自内部类，而不是外围类。但是当要生成一个构造器时，默认的构造器并不算好，而且不能只是传递一个指向外围类对象的引用。此外，必须在构造器内使用如下语法：

```
enclosingClassReference.super();
```

这样才提供了必要的引用，然后程序才能编译通过。

12.10 内部类可以被覆盖么？

如果创建了一个内部类，然后继承其外围类并重新定义此内部类时，会发生什么呢？也就是说，内部类可以被覆盖吗？这看起来似乎是个很有用的思想，但是“覆盖”内部类就好像它是外围类的一个方法，其实并不起什么作用：

```
// innerclasses/BigEgg.java
// An inner class cannot be overridden like a method
class Egg {
    private Yolk y;
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg.Yolk()");
        }
    }
    Egg() {
        System.out.println("New Egg()");
        y = new Yolk();
    }
}
public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() {
            System.out.println("BigEgg.Yolk()");
        }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
}
```

输出为：

```
New Egg()
Egg.Yolk()
```

默认的无参构造器是编译器自动生成的，这里是调用基类的默认构造器。你可能认为既然创建了 **BigEgg** 的对象，那么所使用的应该是“覆盖后”的 **Yolk** 版本，但从输出中可以看到实际情况并不是这样的。

这个例子说明，当继承了某个外围类的时候，内部类并没有发生什么特别神奇的变化。这两个内部类是完全独立的两个实体，各自在自己的命名空间内。当然，明确地继承某个内部类也是可以的：

```java
// innerclasses/BigEgg2.java
// Proper inheritance of an inner class
class Egg2 {
    protected class Yolk {
        public Yolk() {
            System.out.println("Egg2.Yolk()");
        }
        public void f() {
            System.out.println("Egg2.Yolk.f()");
        }
    }
    private Yolk y = new Yolk();
    Egg2() { System.out.println("New Egg2()"); }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}
public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() {
            System.out.println("BigEgg2.Yolk()");
        }
        @Override
        public void f() {
            System.out.println("BigEgg2.Yolk.f()");
        }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
}
```

输出为：

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
```

```
BigEgg2.Yolk.f()
```

现在 **BigEgg2.Yolk** 通过 **extends Egg2.Yolk** 明确地继承了此内部类，并且覆盖了其中的方法。`insertYolk()` 方法允许 **BigEgg2** 将它自己的 **Yolk** 对象向上转型为 **Egg2** 中的引用 **y**。所以当 `g()` 调用 `y.f()` 时，覆盖后的新版的 `f()` 被执行。第二次调用 `Egg2.Yolk()`，结果是 **BigEgg2.Yolk** 的构造器调用了其基类的构造器。可以看到在调用 `g()` 的时候，新版的 `f()` 被调用了。

12.11 局部内部类

前面提到过，可以在代码块里创建内部类，典型的方式是在一个方法体的里面创建。局部内部类不能有访问说明符，因为它不是外围类的一部分；但是它可以访问当前代码块内的常量，以及此外围类的所有成员。下面的例子对局部内部类与匿名内部类的创建进行了比较。

```
// innerclasses/LocalInnerClass.java
// Holds a sequence of Objects
interface Counter {
    int next();
}
public class LocalInnerClass {
    private int count = 0;
    Counter getCounter(final String name) {
        // A local inner class:
        class LocalCounter implements Counter {
            LocalCounter() {
                // Local inner class can have a constructor
                System.out.println("LocalCounter()");
            }
            @Override
            public int next() {
                System.out.print(name); // Access local final
                return count++;
            }
        }
        return new LocalCounter();
    }
    // Repeat, but with an anonymous inner class:
    Counter getCounter2(final String name) {
        return new Counter() {
            // Anonymous inner class cannot have a named
            // constructor, only an instance initializer:
            {
                System.out.println("Counter()");
            }
            @Override
            public int next() {
                System.out.print(name); // Access local final
```

```
                return count++;
            }
        };
    }
    public static void main(String[] args) {
        LocalInnerClass lic = new LocalInnerClass();
        Counter
                c1 = lic.getCounter("Local inner "),
                c2 = lic.getCounter2("Anonymous inner ");
        for(int i = 0; i < 5; i++)
            System.out.println(c1.next());
        for(int i = 0; i < 5; i++)
            System.out.println(c2.next());
    }
}
```

输出为:

```
LocalCounter()
Counter()
Local inner 0
Local inner 1
Local inner 2
Local inner 3
Local inner 4
Anonymous inner 5
Anonymous inner 6
Anonymous inner 7
Anonymous inner 8
Anonymous inner 9
```

Counter 返回的是序列中的下一个值。我们分别使用局部内部类和匿名内部类实现了这个功能，它们具有相同的行为和能力，既然局部内部类的名字在方法外是不可见的，那为什么我们仍然使用局部内部类而不是匿名内部类呢？唯一的理由是，我们需要一个已命名的构造器，或者需要重载构造器，而匿名内部类只能用于实例初始化。

所以使用局部内部类而不使用匿名内部类的另一个理由就是，需要不止一个该内部类的对象。

12.12 内部类标识符

由于编译后每个类都会产生一个**.class** 文件，其中包含了如何创建该类型的对象的全部信息（此信息产生一个“meta-class”，叫做 **Class** 对象）。

你可能猜到了，内部类也必须生成一个**.class** 文件以包含它们的 **Class** 对象信息。这些类文件的命名有严格的规则：外围类的名字，加上“**$**”，再加上内部类的名字。例如，**LocalInnerClass.java** 生成的 **.class** 文件包括:

```
Counter.class
LocalInnerClass$1.class
LocalInnerClass$LocalCounter.class
LocalInnerClass.class
```

如果内部类是匿名的，编译器会简单地产生一个数字作为其标识符。如果内部类是嵌套在别的内部类之中，只需直接将它们的名字加在其外围类标识符与“**$**”的后面。

虽然这种命名格式简单而直接，但它还是很健壮的，足以应对绝大多数情况。因为这是 java 的标准命名方式，所以产生的文件自动都是平台无关的。（注意，为了保证你的内部类能起作用，Java 编译器会尽可能地转换它们。）

12.13 本章小结

比起面向对象编程中其他的概念来，接口和内部类更深奥复杂，比如 C++ 就没有这些。将两者结合起来，同样能够解决 C++ 中的用多重继承所能解决的问题。然而，多重继承在 C++ 中被证明是相当难以使用的，相比较而言，Java 的接口和内部类就容易理解多了。

虽然这些特性本身是相当直观的，但是就像多态机制一样，这些特性的使用应该是设计阶段考虑的问题。随着时间的推移，读者将能够更好地识别什么情况下应该使用接口，什么情况使用内部类，或者两者同时使用。但此时，读者至少应该已经完全理解了它们的语法和语义。

当读者见到这些语言特性的实际应用时，就能最终理解它们了。

13 集合

> 如果一个程序只包含固定数量的对象且对象的生命周期都是已知的，那么这是一个非常简单的程序。

通常，程序总是根据运行时才知道的某些条件去创建新的对象。在此之前，无法知道所需对象的数量甚至确切类型。为了解决这个普遍的编程问题，需要在任意时刻和任意位置创建任意数量的对象。因此，不能依靠创建命名的引用来持有每一个对象：

```
MyType aReference;
```

因为从来不会知道实际需要多少个这样的引用。

大多数编程语言都提供了某种方法来解决这个基本问题。Java 有多种方式保存对象（确切地说，是对象的引用）。例如前边曾经学习过的数组，它是编译器支持的类型。数组是保存一组对象的最有效的方式，如果想要保存一组基本类型数据，也推荐使用数组。但是数组具有固定的大小尺寸，而且在更一般的情况下，在写程序的时候并不知道将需要多少个对象，或者是否需要更复杂的方式来存储对象，因此数组尺寸固定这一限制就显得太过受限了。

java.util 库提供了一套相当完整的*集合类*（collection classes）来解决这个问题，其中基本的类型有 **List** 、**Set** 、**Queue** 和 **Map**。这些类型也被称作*容器类*（container classes），但我将使用 Java 类库使用的术语。集合提供了完善的方法来保存对象，可以使用这些工具来解决大量的问题。

集合还有一些其它特性。例如，**Set** 对于每个值都只保存一个对象，**Map** 是一个关联数组，允许将某些对象与其他对象关联起来。Java 集合类都可以自动地调整自己的大小。因此，与数组不同，在编程时，可以将任意数量的对象放置在集合中，而不用关心集合应该有多大。

尽管在 Java 中没有直接的关键字支持，[1]但集合类仍然是可以显著增强编程能力的基本工具。在本章中，将介绍 Java 集合类库的基本知识，并重点介绍一些典型用法。这里将专注于在日常编程中使用的集合。稍后，在附录：集合主题中，还将学习到其余的那些集合和相关功能，以及如何使用它们的更多详细信息。

[1]功能粘贴在一起的方法的确有点与众不同，但它仍不失为一个库。

13.1 泛型和类型安全的集合

使用 Java 5 之前的集合的一个主要问题是编译器允许你向集合中插入不正确的类型。例如，考虑一个 **Apple** 对象的集合，这里使用最基本最可靠的 **ArrayList** 。现在，可以把 **ArrayList** 看作“可以自动扩充自身尺寸的数组”来看待。使用 **ArrayList** 相当简单：创建一个实例，用 `add()` 插入对象；然后用 `get()` 来访问这些对象，此时需要使用索引，就像数组那样，但是不需要方括号。[2] **ArrayList** 还有一个 `size()` 方法，来说明集合中包含了多少个元素，所以不会不小心因数组越界而引发错误（通过抛出*运行时异常*，异常章节介绍了异常）。

在本例中，**Apple** 和 **Orange** 都被放到了集合中，然后将它们取出。正常情况下，Java 编译器会给出警告，因为这个示例没有使用泛型。在这里，使用特定的注解来抑制警告信息。注解以“@”符号开头，可以带参数。这里的 `@SuppressWarning` 注解及其参数表示只抑制“unchecked”类型的警告（注解章节将介绍更多有关注解的信息）：

```
// collections/ApplesAndOrangesWithoutGenerics.java
// Simple collection use (suppressing compiler warnings)
// {ThrowsException}
import java.util.*;

class Apple {
  private static long counter;
  private final long id = counter++;
  public long id() { return id; }
}

class Orange {}

public class ApplesAndOrangesWithoutGenerics {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    ArrayList apples = new ArrayList();
    for(int i = 0; i < 3; i++)
      apples.add(new Apple());
    // No problem adding an Orange to apples:
    apples.add(new Orange());
    for(Object apple : apples) {
      ((Apple) apple).id();
      // Orange is detected only at run time
    }
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "main"
java.lang.ClassCastException: Orange cannot be cast to
```

[2]例如，这个电子书是利用 Pandoc 制作出来的，它是用纯函数式语言 Haskell 编写的一个程序。

```
Apple
        at ApplesAndOrangesWithoutGenerics.main(ApplesA
ndOrangesWithoutGenerics.java:23)
*/
```

Apple 和 **Orange** 是截然不同的，它们除了都是 **Object** 之外没有任何共同点（如果一个类没有显式地声明继承自哪个类，那么它就自动继承自 **Object**）。因为 **ArrayList** 保存的是 **Object** ，所以不仅可以通过 **ArrayList** 的 `add()` 方法将 **Apple** 对象放入这个集合，而且可以放入 **Orange** 对象，这无论在编译期还是运行时都不会有问题。当使用 **ArrayList** 的 `get()` 方法来取出你认为是 **Apple** 的对象时，得到的只是 **Object** 引用，必须将其转型为 **Apple**。然后需要将整个表达式用括号括起来，以便在调用 **Apple** 的 `id()` 方法之前，强制执行转型。否则，将会产生语法错误。

在运行时，当尝试将 **Orange** 对象转为 **Apple** 时，会出现输出中显示的错误。

在泛型章节中，你将了解到使用 Java 泛型来创建类可能很复杂。但是，使用预先定义的泛型类却相当简单。例如，要定义一个用于保存 **Apple** 对象的 **ArrayList** ，只需要使用 **ArrayList<Apple>** 来代替 **ArrayList** 。尖括号括起来的是*类型参数*（可能会有多个），它指定了这个集合实例可以保存的类型。

通过使用泛型，就可以在编译期防止将错误类型的对象放置到集合中。[3]下面还是这个示例，但是使用了泛型：

```java
// collections/ApplesAndOrangesWithGenerics.java
import java.util.*;

public class ApplesAndOrangesWithGenerics {
  public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<>();
    for(int i = 0; i < 3; i++)
      apples.add(new Apple());
    // Compile-time error:
    // apples.add(new Orange());
    for(Apple apple : apples) {
      System.out.println(apple.id());
    }
  }
}
/* Output:
0
1
2
*/
```

在 **apples** 定义的右侧，可以看到 `new ArrayList<>()` 。这有时被称为“菱形语法”（diamond syntax）。在 Java 7 之前，必须要在两端都进行类型声明，如下所示：

[3]有时函数式语言将其描述为“代码即数据”。

```
ArrayList<Apple> apples = new ArrayList<Apple>();
```

随着类型变得越来越复杂，这种重复产生的代码非常混乱且难以阅读。程序员发现所有类型信息都可以从左侧获得，因此，编译器没有理由强迫右侧再重复这些。虽然*类型推断*（type inference）只是个很小的请求，Java 语言团队仍然欣然接受并进行了改进。

有了 **ArrayList** 声明中的类型指定，编译器会阻止将 **Orange** 放入 **apples**，因此，这会成为一个编译期错误而不是运行时错误。

使用泛型，从 **List** 中获取元素不需要强制类型转换。因为 **List** 知道它持有什么类型，因此当调用 `get()` 时，它会替你执行转型。因此，使用泛型，你不仅知道编译器将检查放入集合的对象类型，而且在使用集合中的对象时也可以获得更清晰的语法。

当指定了某个类型为泛型参数时，并不仅限于只能将确切类型的对象放入集合中。向上转型也可以像作用于其他类型一样作用于泛型：

```
// collections/GenericsAndUpcasting.java
import java.util.*;

class GrannySmith extends Apple {}
class Gala extends Apple {}
class Fuji extends Apple {}
class Braeburn extends Apple {}

public class GenericsAndUpcasting {
  public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<>();
    apples.add(new GrannySmith());
    apples.add(new Gala());
    apples.add(new Fuji());
    apples.add(new Braeburn());
    for(Apple apple : apples)
      System.out.println(apple);
  }
}
/* Output:
GrannySmith@15db9742
Gala@6d06d69c
Fuji@7852e922
Braeburn@4e25154f
*/
```

因此，可以将 **Apple** 的子类型添加到被指定为保存 **Apple** 对象的集合中。

程序的输出是从 **Object** 默认的 `toString()` 方法产生的，该方法打印类名，后边跟着对象的散列码的无符号十六进制表示（这个散列码是通过 `hashCode()` 方法产生的）。将在附录：理解 equals 和 hashCode 方法中了解有关散列码的内容。

13.2 基本概念

Java 集合类库采用“持有对象”（holding objects）的思想，并将其分为两个不同的概念，表示为类库的基本接口：

1. **集合（Collection）**：一个独立元素的序列，这些元素都服从一条或多条规则。**List** 必须以插入的顺序保存元素，**Set** 不能包含重复元素，**Queue** 按照*排队规则*来确定对象产生的顺序（通常与它们被插入的顺序相同）。
2. **映射（Map）**：一组成对的“键值对”对象，允许使用键来查找值。**ArrayList** 使用数字来查找对象，因此在某种意义上讲，它是将数字和对象关联在一起。**map** 允许我们使用一个对象来查找另一个对象，它也被称作*关联数组*（associative array），因为它将对象和其它对象关联在一起；或者称作*字典*（dictionary），因为可以使用一个键对象来查找值对象，就像在字典中使用单词查找定义一样。**Map** 是强大的编程工具。

尽管并非总是可行，但在理想情况下，你编写的大部分代码都在与这些接口打交道，并且唯一需要指定所使用的精确类型的地方就是在创建的时候。因此，可以像下面这样创建一个 **List**：

```
List<Apple> apples = new ArrayList<>();
```

请注意，**ArrayList** 已经被向上转型为了 **List**，这与之前示例中的处理方式正好相反。使用接口的目的是，如果想要改变具体实现，只需在创建时修改它就行了，就像下面这样：

```
List<Apple> apples = new LinkedList<>();
```

因此，应该创建一个具体类的对象，将其向上转型为对应的接口，然后在其余代码中都是用这个接口。

这种方式并非总是有效的，因为某些具体类有额外的功能。例如，**LinkedList** 具有 **List** 接口中未包含的额外方法，而 **TreeMap** 也具有在 **Map** 接口中未包含的方法。如果需要使用这些方法，就不能将它们向上转型为更通用的接口。

Collection 接口概括了*序列*的概念——一种存放一组对象的方式。下面是个简单的示例，用 **Integer** 对象填充了一个 **Collection**（这里用 **ArrayList** 表示），然后打印集合中的每个元素：

```
// collections/SimpleCollection.java
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
    Collection<Integer> c = new ArrayList<>();
    for(int i = 0; i < 10; i++)
      c.add(i); // Autoboxing
    for(Integer i : c)
      System.out.print(i + ", ");
  }
}
```

```
/* Output:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
*/
```

这个例子仅使用了 **Collection** 中的方法（即 `add()` ），所以使用任何继承自 **Collection** 的类的对象都可以正常工作。但是 **ArrayList** 是最基本的序列类型。

`add()` 方法的名称就表明它是在 **Collection** 中添加一个新元素。但是，文档中非常详细地叙述到 `add()` "要确保这个 **Collection** 包含指定的元素。"这是因为考虑到了 **Set** 的含义，因为在 **Set** 中，只有当元素不存在时才会添加元素。在使用 **ArrayList** ，或任何其他类型的 **List** 时，`add()` 总是表示"把它放进去"，因为 **List** 不关心是否存在重复元素。

可以使用 *for-in* 语法来遍历所有的 **Collection** ，就像这里所展示的那样。在本章的后续部分，还将学习到一个更灵活的概念，*迭代器*。

13.3 添加元素组

在 **java.util** 包中的 **Arrays** 和 **Collections** 类中都有很多实用的方法，可以在一个 **Collection** 中添加一组元素。

`Arrays.asList()` 方法接受一个数组或是逗号分隔的元素列表（使用可变参数），并将其转换为 **List** 对象。`Collections.addAll()` 方法接受一个 **Collection** 对象，以及一个数组或是一个逗号分隔的列表，将其中元素添加到 **Collection** 中。下边的示例展示了这两个方法，以及更通用的 `addAll()` 方法，所有 **Collection** 类型都包含该方法：

```java
// collections/AddingGroups.java
// Adding groups of elements to Collection objects
import java.util.*;

public class AddingGroups {
  public static void main(String[] args) {
    Collection<Integer> collection =
      new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
    Integer[] moreInts = { 6, 7, 8, 9, 10 };
    collection.addAll(Arrays.asList(moreInts));
    // Runs significantly faster, but you can't
    // construct a Collection this way:
    Collections.addAll(collection, 11, 12, 13, 14, 15);
    Collections.addAll(collection, moreInts);
    // Produces a list "backed by" an array:
    List<Integer> list = Arrays.asList(16,17,18,19,20);
    list.set(1, 99); // OK -- modify an element
    // list.add(21); // Runtime error; the underlying
                     // array cannot be resized.
  }
}
```

Collection 的构造器可以接受另一个 **Collection**，用它来将自身初始化。因此，可以使用 `Arrays.asList()` 来为这个构造器产生输入。但是，`Collections.addAll()` 运行得更快，而且很容易构建一个不包含元素的 **Collection**，然后调用 `Collections.addAll()`，因此这是首选方式。

`Collection.addAll()` 方法只能接受另一个 **Collection** 作为参数，因此它没有 `Arrays.asList()` 或 `Collections.addAll()` 灵活。这两个方法都使用可变参数列表。

也可以直接使用 `Arrays.asList()` 的输出作为一个 **List**，但是这里的底层实现是数组，没法调整大小。如果尝试在这个 **List** 上调用 `add()` 或 `remove()`，由于这两个方法会尝试修改数组大小，所以会在运行时得到"Unsupported Operation（不支持的操作）"错误：

```java
// collections/AsListInference.java
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
class Crusty extends Snow {}
class Slush extends Snow {}

public class AsListInference {
  public static void main(String[] args) {
    List<Snow> snow1 = Arrays.asList(
      new Crusty(), new Slush(), new Powder());
    //- snow1.add(new Heavy()); // Exception

    List<Snow> snow2 = Arrays.asList(
      new Light(), new Heavy());
    //- snow2.add(new Slush()); // Exception

    List<Snow> snow3 = new ArrayList<>();
    Collections.addAll(snow3,
      new Light(), new Heavy(), new Powder());
    snow3.add(new Crusty());

    // Hint with explicit type argument specification:
    List<Snow> snow4 = Arrays.<Snow>asList(
       new Light(), new Heavy(), new Slush());
    //- snow4.add(new Powder()); // Exception
  }
}
```

在 **snow4** 中，注意 `Arrays.asList()` 中间的"暗示"（即 `<Snow>`），告诉编译器 `Arrays.asList()` 生成的结果 **List** 类型的实际目标类型是什么。这称为*显式类型参数说明*（explicit type argument specification）。

13.4 集合的打印

必须使用 `Arrays.toString()` 来生成数组的可打印形式。但是打印集合无需任何帮助。下面是一个例子，这个例子中也介绍了基本的 Java 集合：

```java
// collections/PrintingCollections.java
// Collections print themselves automatically
import java.util.*;

public class PrintingCollections {
  static Collection
  fill(Collection<String> collection) {
    collection.add("rat");
    collection.add("cat");
    collection.add("dog");
    collection.add("dog");
    return collection;
  }
  static Map fill(Map<String, String> map) {
    map.put("rat", "Fuzzy");
    map.put("cat", "Rags");
    map.put("dog", "Bosco");
    map.put("dog", "Spot");
    return map;
  }
  public static void main(String[] args) {
    System.out.println(fill(new ArrayList<>()));
    System.out.println(fill(new LinkedList<>()));
    System.out.println(fill(new HashSet<>()));
    System.out.println(fill(new TreeSet<>()));
    System.out.println(fill(new LinkedHashSet<>()));
    System.out.println(fill(new HashMap<>()));
    System.out.println(fill(new TreeMap<>()));
    System.out.println(fill(new LinkedHashMap<>()));
  }
}
/* Output:
[rat, cat, dog, dog]
[rat, cat, dog, dog]
[rat, cat, dog]
[cat, dog, rat]
[rat, cat, dog]
{rat=Fuzzy, cat=Rags, dog=Spot}
{cat=Rags, dog=Spot, rat=Fuzzy}
{rat=Fuzzy, cat=Rags, dog=Spot}
*/
```

这显示了 Java 集合库中的两个主要类型。它们的区别在于集合中的每个“槽”（slot）保存的元素个数。**Collection** 类型在每个槽中只能保存一个元素。此类集合包括：**List**，它以特定的顺序保存一

组元素；**Set**，其中元素不允许重复；**Queue**，只能在集合一端插入对象，并从另一端移除对象（就本例而言，这只是查看序列的另一种方式，因此并没有显示它）。**Map** 在每个槽中存放了两个元素，即*键*和与之关联的*值*。

默认的打印行为，使用集合提供的 `toString()` 方法即可生成可读性很好的结果。**Collection** 打印出的内容用方括号括住，每个元素由逗号分隔。**Map** 则由大括号括住，每个键和值用等号连接（键在左侧，值在右侧）。

第一个 `fill()` 方法适用于所有类型的 **Collection**，这些类型都实现了 `add()` 方法以添加新元素。

ArrayList 和 **LinkedList** 都是 **List** 的类型，从输出中可以看出，它们都按插入顺序保存元素。两者之间的区别不仅在于执行某些类型的操作时的性能，而且 **LinkedList** 包含的操作多于 **ArrayList**。本章后面将对这些内容进行更全面的探讨。

HashSet，**TreeSet** 和 **LinkedHashSet** 是 **Set** 的类型。从输出中可以看到，**Set** 仅保存每个相同项中的一个，并且不同的 **Set** 实现存储元素的方式也不同。**HashSet** 使用相当复杂的方法存储元素，这在附录：集合主题中进行了探讨。现在只需要知道，这种技术是检索元素的最快方法，因此，存储顺序看上去没有什么意义（通常只关心某事物是否是 **Set** 的成员，而存储顺序并不重要）。如果存储顺序很重要，则可以使用 **TreeSet**，它将按比较结果的升序保存对象）或 **LinkedHashSet**，它按照被添加的先后顺序保存对象。

Map（也称为*关联数组*）使用*键*来查找对象，就像一个简单的数据库。所关联的对象称为*值*。假设有一个 **Map** 将美国州名与它们的首府联系在一起，如果想要俄亥俄州（Ohio）的首府，可以用“Ohio”作为键来查找，几乎就像使用数组下标一样。正是由于这种行为，对于每个键，**Map** 只存储一次。

`Map.put(key, value)` 添加一个所想要添加的值并将它与一个键（用来查找值）相关联。`Map.get(key)` 生成与该键相关联的值。上面的示例仅添加键值对，并没有执行查找。这将在稍后展示。

请注意，这里没有指定（或考虑）**Map** 的大小，因为它会自动调整大小。此外，**Map** 还知道如何打印自己，它会显示相关联的键和值。

本例使用了 **Map** 的三种基本风格：**HashMap**，**TreeMap** 和 **LinkedHashMap**。

键和值保存在 **HashMap** 中的顺序不是插入顺序，因为 **HashMap** 实现使用了非常快速的算法来控制顺序。**TreeMap** 通过比较结果的升序来保存键，**LinkedHashMap** 在保持 **HashMap** 查找速度的同时按键的插入顺序保存键。

13.5 列表 List

Lists 承诺将元素保存在特定的序列中。**List** 接口在 **Collection** 的基础上添加了许多方法，允许在 **List** 的中间插入和删除元素。

有两种类型的 **List**：

- 基本的 **ArrayList**，擅长随机访问元素，但在 **List** 中间插入和删除元素时速度较慢。
- **LinkedList**，它通过代价较低的在 **List** 中间进行的插入和删除操作，提供了优化的顺序访问。**LinkedList** 对于随机访问来说相对较慢，但它具有比 **ArrayList** 更大的特征集。

下面的示例导入 **typeinfo.pets**，超前使用了类型信息一章中的类库。这个类库包含了 **Pet** 类层次结构，以及用于随机生成 **Pet** 对象的一些工具类。此时不需要了解完整的详细信息，只需要知道两点：

1. 有一个 **Pet** 类，以及 **Pet** 的各种子类型。
2. 静态的 `Pets.arrayList()` 方法返回一个填充了随机选取的 **Pet** 对象的 **ArrayList**：

```
// collections/ListFeatures.java
import typeinfo.pets.*;
import java.util.*;

public class ListFeatures {
  public static void main(String[] args) {
    Random rand = new Random(47);
    List<Pet> pets = Pets.list(7);
    System.out.println("1: " + pets);
    Hamster h = new Hamster();
    pets.add(h); // Automatically resizes
    System.out.println("2: " + pets);
    System.out.println("3: " + pets.contains(h));
    pets.remove(h); // Remove by object
    Pet p = pets.get(2);
    System.out.println(
      "4: " +  p + " " + pets.indexOf(p));
    Pet cymric = new Cymric();
    System.out.println("5: " + pets.indexOf(cymric));
    System.out.println("6: " + pets.remove(cymric));
    // Must be the exact object:
    System.out.println("7: " + pets.remove(p));
    System.out.println("8: " + pets);
    pets.add(3, new Mouse()); // Insert at an index
    System.out.println("9: " + pets);
    List<Pet> sub = pets.subList(1, 4);
    System.out.println("subList: " + sub);
    System.out.println("10: " + pets.containsAll(sub));
    Collections.sort(sub); // In-place sort
    System.out.println("sorted subList: " + sub);
    // Order is not important in containsAll():
    System.out.println("11: " + pets.containsAll(sub));
    Collections.shuffle(sub, rand); // Mix it up
    System.out.println("shuffled subList: " + sub);
    System.out.println("12: " + pets.containsAll(sub));
    List<Pet> copy = new ArrayList<>(pets);
    sub = Arrays.asList(pets.get(1), pets.get(4));
    System.out.println("sub: " + sub);
```

```
    copy.retainAll(sub);
    System.out.println("13: " + copy);
    copy = new ArrayList<>(pets); // Get a fresh copy
    copy.remove(2); // Remove by index
    System.out.println("14: " + copy);
    copy.removeAll(sub); // Only removes exact objects
    System.out.println("15: " + copy);
    copy.set(1, new Mouse()); // Replace an element
    System.out.println("16: " + copy);
    copy.addAll(2, sub); // Insert a list in the middle
    System.out.println("17: " + copy);
    System.out.println("18: " + pets.isEmpty());
    pets.clear(); // Remove all elements
    System.out.println("19: " + pets);
    System.out.println("20: " + pets.isEmpty());
    pets.addAll(Pets.list(4));
    System.out.println("21: " + pets);
    Object[] o = pets.toArray();
    System.out.println("22: " + o[3]);
    Pet[] pa = pets.toArray(new Pet[0]);
    System.out.println("23: " + pa[3].id());
  }
}
/* Output:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
subList: [Manx, Mutt, Mouse]
10: true
sorted subList: [Manx, Mouse, Mutt]
11: true
shuffled subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Manx, Cymric, Rat, EgyptianMau]
22: EgyptianMau
23: 14
```

```
90  */
```

打印行都编了号，因此可从输出追溯到源代码。第 1 行输出展示了原始的由 **Pet** 组成的 **List** 。与数组不同，**List** 可以在创建后添加或删除元素，并自行调整大小。这正是它的重要价值：一种可修改的序列。在第 2 行输出中可以看到添加一个 **Hamster** 的结果，该对象将被追加到列表的末尾。

可以使用 `contains()` 方法确定对象是否在列表中。如果要删除一个对象，可以将该对象的引用传递给 `remove()` 方法。同样，如果有一个对象的引用，可以使用 `indexOf()` 在 **List** 中找到该对象所在位置的下标号，如第 4 行输出所示中所示。

当确定元素是否是属于某个 **List** ，寻找某个元素的索引，以及通过引用从 **List** 中删除元素时，都会用到 `equals()` 方法（根类 **Object** 的一个方法）。每个 **Pet** 被定义为一个唯一的对象，所以即使列表中已经有两个 **Cymrics** ，如果再创建一个新的 **Cymric** 对象并将其传递给 `indexOf()` 方法，结果仍为 **-1** （表示未找到），并且尝试调用 `remove()` 方法来删除这个对象将返回 **false** 。对于其他类，`equals()` 的定义可能有所不同。例如，如果两个 **String** 的内容相同，则这两个 **String** 相等。因此，为了防止出现意外，请务必注意 **List** 行为会根据 `equals()` 行为而发生变化。

第 7、8 行输出展示了删除与 **List** 中的对象完全匹配的对象是成功的。

可以在 **List** 的中间插入一个元素，就像在第 9 行输出和它之前的代码那样。但这会带来一个问题：对于 **LinkedList** ，在列表中间插入和删除都是廉价操作（在本例中，除了对列表中间进行的真正的随机访问），但对于 **ArrayList** ，这可是代价高昂的操作。这是否意味着永远不应该在 **ArrayList** 的中间插入元素，并最好是转换为 **LinkedList** ？不，它只是意味着你应该意识到这个问题，如果你开始在某个 **ArrayList** 中间执行很多插入操作，并且程序开始变慢，那么你应该看看你的 **List** 实现有可能就是罪魁祸首（发现此类瓶颈的最佳方式是使用分析器 profiler）。优化是一个很棘手的问题，最好的策略就是置之不顾，直到发现必须要去担心它了（尽管去理解这些问题总是一个很好的主意）。

`subList()` 方法可以轻松地从更大的列表中创建切片，当将切片结果传递给原来这个较大的列表的 `containsAll()` 方法时，很自然地会得到 **true**。请注意，顺序并不重要，在第 11、12 行输出中可以看到，在 **sub** 上调用直观命名的 `Collections.sort()` 和 `Collections.shuffle()` 方法，不会影响 `containsAll()` 的结果。`subList()` 所产生的列表的幕后支持就是原始列表。因此，对所返回列表的更改都将会反映在原始列表中，反之亦然。

`retainAll()` 方法实际上是一个“集合交集”操作，在本例中，它保留了同时在 **copy** 和 **sub** 中的所有元素。请再次注意，所产生的结果行为依赖于 `equals()` 方法。

第 14 行输出展示了使用索引号来删除元素的结果，与通过对象引用来删除元素相比，它显得更加直观，因为在使用索引时，不必担心 `equals()` 的行为。

`removeAll()` 方法也是基于 `equals()` 方法运行的。顾名思义，它会从 **List** 中删除在参数 **List** 中的所有元素。

`set()` 方法的命名显得很不合时宜，因为它与 **Set** 类存在潜在的冲突。在这里使用“replace”可能更适合，因为它的功能是用第二个参数替换索引处的元素（第一个参数）。

第 17 行输出表明，对于 **List** ，有一个重载的 `addAll()` 方法可以将新列表插入到原始列表的中

间位置，而不是仅能用 **Collection** 的 `addAll()` 方法将其追加到列表的末尾。

第 18 - 20 行输出展示了 `isEmpty()` 和 `clear()` 方法的效果。

第 22、23 行输出展示了如何使用 `toArray()` 方法将任意的 **Collection** 转换为数组。这是一个重载方法，其无参版本返回一个 **Object** 数组，但是如果将目标类型的数组传递给这个重载版本，那么它会生成一个指定类型的数组（假设它通过了类型检查）。如果参数数组太小而无法容纳 **List** 中的所有元素（就像本例一样），则 `toArray()` 会创建一个具有合适尺寸的新数组。**Pet** 对象有一个 `id()` 方法，可以在所产生的数组中的对象上调用这个方法。

13.6 迭代器 Iterators

在任何集合中，都必须有某种方式可以插入元素并再次获取它们。毕竟，保存事物是集合最基本的工作。对于 **List** ，`add()` 是插入元素的一种方式，`get()` 是获取元素的一种方式。

如果从更高层次的角度考虑，会发现这里有个缺点：要使用集合，必须对集合的确切类型编程。这一开始可能看起来不是很糟糕，但是考虑下面的情况：如果原本是对 **List** 编码的，但是后来发现如果能够将相同的代码应用于 **Set** 会更方便，此时应该怎么做？或者假设想从一开始就编写一段通用代码，它不知道或不关心它正在使用什么类型的集合，因此它可以用于不同类型的集合，那么如何才能不重写代码就可以应用于不同类型的集合？

迭代器（也是一种设计模式）的概念实现了这种抽象。迭代器是一个对象，它在一个序列中移动并选择该序列中的每个对象，而客户端程序员不知道或不关心该序列的底层结构。另外，迭代器通常被称为*轻量级对象*（lightweight object）：创建它的代价小。因此，经常可以看到一些对迭代器有些奇怪的约束。例如，Java 的 **Iterator** 只能单向移动。这个 **Iterator** 只能用来：

1. 使用 `iterator()` 方法要求集合返回一个 **Iterator**。**Iterator** 将准备好返回序列中的第一个元素。
2. 使用 `next()` 方法获得序列中的下一个元素。
3. 使用 `hasNext()` 方法检查序列中是否还有元素。
4. 使用 `remove()` 方法将迭代器最近返回的那个元素删除。

为了观察它的工作方式，这里再次使用类型信息章节中的 **Pet** 工具：

```
// collections/SimpleIteration.java
import typeinfo.pets.*;
import java.util.*;

public class SimpleIteration {
  public static void main(String[] args) {
    List<Pet> pets = Pets.list(12);
    Iterator<Pet> it = pets.iterator();
    while(it.hasNext()) {
      Pet p = it.next();
```

```
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
    // A simpler approach, when possible:
    for(Pet p : pets)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
    // An Iterator can also remove elements:
    it = pets.iterator();
    for(int i = 0; i < 6; i++) {
      it.next();
      it.remove();
    }
    System.out.println(pets);
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx 8:Cymric 9:Rat 10:EgyptianMau 11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx 8:Cymric 9:Rat 10:EgyptianMau 11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
*/
```

有了 **Iterator**，就不必再为集合中元素的数量操心了。这是由 `hasNext()` 和 `next()` 关心的事情。

如果只是想向前遍历 **List**，并不打算修改 **List** 对象本身，那么使用 *for-in* 语法更加简洁。

Iterator 还可以删除由 `next()` 生成的最后一个元素，这意味着在调用 `remove()` 之前必须先调用 `next()`。[4]

在集合中的每个对象上执行操作，这种思想十分强大，并且贯穿于本书。

现在考虑创建一个 `display()` 方法，它不必知晓集合的确切类型：

```
// collections/CrossCollectionIteration.java
import typeinfo.pets.*;
import java.util.*;

public class CrossCollectionIteration {
  public static void display(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
    List<Pet> pets = Pets.list(8);
```

[4]这个语法来自 C++。

```
    LinkedList<Pet> petsLL = new LinkedList<>(pets);
    HashSet<Pet> petsHS = new HashSet<>(pets);
    TreeSet<Pet> petsTS = new TreeSet<>(pets);
    display(pets.iterator());
    display(petsLL.iterator());
    display(petsHS.iterator());
    display(petsTS.iterator());
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug
0:Rat
*/
```

`display()` 方法不包含任何有关它所遍历的序列的类型信息。这也展示了 **Iterator** 的真正威力：能够将遍历序列的操作与该序列的底层结构分离。出于这个原因，我们有时会说：迭代器统一了对集合的访问方式。

我们可以使用 **Iterable** 接口生成上一个示例的更简洁版本，该接口描述了“可以产生 **Iterator** 的任何东西”：

```
// collections/CrossCollectionIteration2.java
import typeinfo.pets.*;
import java.util.*;

public class CrossCollectionIteration2 {
  public static void display(Iterable<Pet> ip) {
    Iterator<Pet> it = ip.iterator();
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
    List<Pet> pets = Pets.list(8);
    LinkedList<Pet> petsLL = new LinkedList<>(pets);
    HashSet<Pet> petsHS = new HashSet<>(pets);
    TreeSet<Pet> petsTS = new TreeSet<>(pets);
    display(pets);
    display(petsLL);
    display(petsHS);
    display(petsTS);
  }
```

```
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug
0:Rat
*/
```

这里所有的类都是 **Iterable**，所以现在对 `display()` 的调用显然更简单。

13.6.1 ListIterator

ListIterator 是一个更强大的 **Iterator** 子类型，它只能由各种 **List** 类生成。**Iterator** 只能向前移动，而 **ListIterator** 可以双向移动。它还可以生成相对于迭代器在列表中指向的当前位置的后一个和前一个元素的索引，并且可以使用 `set()` 方法替换它访问过的最近一个元素。可以通过调用 `listIterator()` 方法来生成指向 **List** 开头处的 **ListIterator**，还可以通过调用 `listIterator(n)` 创建一个一开始就指向列表索引号为 **n** 的元素处的 **ListIterator**。下面的示例演示了所有这些能力：

```java
// collections/ListIteration.java
import typeinfo.pets.*;
import java.util.*;

public class ListIteration {
  public static void main(String[] args) {
    List<Pet> pets = Pets.list(8);
    ListIterator<Pet> it = pets.listIterator();
    while(it.hasNext())
      System.out.print(it.next() +
        ", " + it.nextIndex() +
        ", " + it.previousIndex() + "; ");
    System.out.println();
    // Backwards:
    while(it.hasPrevious())
      System.out.print(it.previous().id() + " ");
    System.out.println();
    System.out.println(pets);
    it = pets.listIterator(3);
    while(it.hasNext()) {
      it.next();
      it.set(Pets.get());
    }
    System.out.println(pets);
  }
}
```

```
/* Output:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug,
5, 4; Cymric, 6, 5; Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
[Rat, Manx, Cymric, Cymric, Rat, EgyptianMau, Hamster,
EgyptianMau]
*/
```

`Pets.get()` 方法用来从位置 3 开始替换 **List** 中的所有 Pet 对象。

13.7 链表 LinkedList

LinkedList 也像 **ArrayList** 一样实现了基本的 **List** 接口，但它在 **List** 中间执行插入和删除操作时比 **ArrayList** 更高效。然而, 它在随机访问操作效率方面却要逊色一些。

LinkedList 还添加了一些方法，使其可以被用作栈、队列或双端队列（deque）。在这些方法中，有些彼此之间可能只是名称有些差异，或者只存在些许差异，以使得这些名字在特定用法的上下文环境中更加适用（特别是在 **Queue** 中）。例如：

- `getFirst()` 和 `element()` 是相同的，它们都返回列表的头部（第一个元素）而并不删除它，如果 **List** 为空，则抛出 **NoSuchElementException** 异常。`peek()` 方法与这两个方法只是稍有差异，它在列表为空时返回 **null** 。
- `removeFirst()` 和 `remove()` 也是相同的，它们删除并返回列表的头部元素，并在列表为空时抛出 **NoSuchElementException** 异常。`poll()` 稍有差异，它在列表为空时返回 **null** 。
- `addFirst()` 在列表的开头插入一个元素。
- `offer()` 与 `add()` 和 `addLast()` 相同。它们都在列表的尾部（末尾）添加一个元素。
- `removeLast()` 删除并返回列表的最后一个元素。

下面的示例展示了这些功能之间基本的相似性和差异性。它并不是重复执行 **ListFeatures.java** 中所示的行为：

```java
// collections/LinkedListFeatures.java
import typeinfo.pets.*;
import java.util.*;

public class LinkedListFeatures {
  public static void main(String[] args) {
    LinkedList<Pet> pets =
      new LinkedList<>(Pets.list(5));
    System.out.println(pets);
    // Identical:
    System.out.println(
      "pets.getFirst(): " + pets.getFirst());
    System.out.println(
```

```
      "pets.element(): " + pets.element());
    // Only differs in empty-list behavior:
    System.out.println("pets.peek(): " + pets.peek());
    // Identical; remove and return the first element:
    System.out.println(
      "pets.remove(): " + pets.remove());
    System.out.println(
      "pets.removeFirst(): " + pets.removeFirst());
    // Only differs in empty-list behavior:
    System.out.println("pets.poll(): " + pets.poll());
    System.out.println(pets);
    pets.addFirst(new Rat());
    System.out.println("After addFirst(): " + pets);
    pets.offer(Pets.get());
    System.out.println("After offer(): " + pets);
    pets.add(Pets.get());
    System.out.println("After add(): " + pets);
    pets.addLast(new Hamster());
    System.out.println("After addLast(): " + pets);
    System.out.println(
      "pets.removeLast(): " + pets.removeLast());
  }
}
/* Output:
[Rat, Manx, Cymric, Mutt, Pug]
pets.getFirst(): Rat
pets.element(): Rat
pets.peek(): Rat
pets.remove(): Rat
pets.removeFirst(): Manx
pets.poll(): Cymric
[Mutt, Pug]
After addFirst(): [Rat, Mutt, Pug]
After offer(): [Rat, Mutt, Pug, Cymric]
After add(): [Rat, Mutt, Pug, Cymric, Pug]
After addLast(): [Rat, Mutt, Pug, Cymric, Pug, Hamster]
pets.removeLast(): Hamster
*/
```

`Pets.list()` 的结果被传递给 **LinkedList** 的构造器，以便使用它来填充 **LinkedList** 。如果查看 **Queue** 接口就会发现，它在 **LinkedList** 的基础上添加了 `element()` ，`offer()` ，`peek()` ，`poll()` 和 `remove()` 方法，以使其可以成为一个 **Queue** 的实现。**Queue** 的完整示例将在本章稍后给出。

13.8 堆栈 Stack

堆栈是“后进先出”（LIFO）集合。它有时被称为*叠加栈*（pushdown stack），因为最后“压入”（push）栈的元素，第一个被“弹出”（pop）栈。经常用来类比栈的事物是带有弹簧支架的自助餐厅托盘。最后装入的托盘总是最先拿出来使用的。

Java 1.0 中附带了一个 **Stack** 类，结果设计得很糟糕（为了向后兼容，我们永远坚持 Java 中的旧设计错误）。Java 6 添加了 **ArrayDeque** ，其中包含直接实现堆栈功能的方法：

```java
// collections/StackTest.java
import java.util.*;

public class StackTest {
  public static void main(String[] args) {
    Deque<String> stack = new ArrayDeque<>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.isEmpty())
      System.out.print(stack.pop() + " ");
  }
}
/* Output:
fleas has dog My
*/
```

即使它是作为一个堆栈在使用，我们仍然必须将其声明为 **Deque** 。有时一个名为 **Stack** 的类更能把事情讲清楚：

```java
// onjava/Stack.java
// A Stack class built with an ArrayDeque
package onjava;
import java.util.Deque;
import java.util.ArrayDeque;

public class Stack<T> {
  private Deque<T> storage = new ArrayDeque<>();
  public void push(T v) { storage.push(v); }
  public T peek() { return storage.peek(); }
  public T pop() { return storage.pop(); }
  public boolean isEmpty() { return storage.isEmpty(); }
  @Override
  public String toString() {
    return storage.toString();
  }
}
```

这里引入了使用泛型的类定义的最简单的可能示例。类名称后面的 告诉编译器这是一个参数化类型，而其中的类型参数 **T** 会在使用类时被实际类型替换。基本上，这个类是在声明“我们在定义一个可

以持有 **T** 类型对象的 **Stack** 。” **Stack** 是使用 **ArrayDeque** 实现的，而 **ArrayDeque** 也被告知它将持有 **T** 类型对象。注意，`push()` 接受类型为 **T** 的对象，而 `peek()` 和 `pop()` 返回类型为 **T** 的对象。`peek()` 方法将返回栈顶元素，但并不将其从栈顶删除，而 `pop()` 删除并返回顶部元素。

如果只需要栈的行为，那么使用继承是不合适的，因为这将产生一个具有 **ArrayDeque** 的其它所有方法的类（在附录：集合主题中将会看到，**Java 1.0** 设计者在创建 **java.util.Stack** 时，就犯了这个错误）。使用组合，可以选择要公开的方法以及如何命名它们。

下面将使用 **StackTest.java** 中的相同代码来演示这个新的 **Stack** 类：

```
// collections/StackTest2.java
import onjava.*;

public class StackTest2 {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.isEmpty())
      System.out.print(stack.pop() + " ");
  }
}
/* Output:
fleas has dog My
*/
```

如果想在自己的代码中使用这个 **Stack** 类，当在创建其实例时，就需要完整指定包名，或者更改这个类的名称；否则，就有可能会与 **java.util** 包中的 **Stack** 发生冲突。例如，如果我们在上面的例子中导入 **java.util.***，那么就必须使用包名来防止冲突：

```
// collections/StackCollision.java

public class StackCollision {
  public static void main(String[] args) {
    onjava.Stack<String> stack = new onjava.Stack<>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.isEmpty())
      System.out.print(stack.pop() + " ");
    System.out.println();
    java.util.Stack<String> stack2 =
      new java.util.Stack<>();
    for(String s : "My dog has fleas".split(" "))
      stack2.push(s);
    while(!stack2.empty())
      System.out.print(stack2.pop() + " ");
  }
}
/* Output:
```

```
fleas has dog My
fleas has dog My
*/
```

尽管已经有了 **java.util.Stack**，但是 **ArrayDeque** 可以产生更好的 **Stack**，因此更可取。

还可以使用显式导入来控制对“首选” **Stack** 实现的选择：

```
import onjava.Stack;
```

现在，任何对 **Stack** 的引用都将选择 **onjava** 版本，而在选择 **java.util.Stack** 时，必须使用全限定名称（full qualification）。

13.9 集合 Set

Set 不保存重复的元素。如果试图将相同对象的多个实例添加到 **Set** 中，那么它会阻止这种重复行为。**Set** 最常见的用途是测试归属性，可以很轻松地询问某个对象是否在一个 **Set** 中。因此，查找通常是 **Set** 最重要的操作，因此通常会选择 **HashSet** 实现，该实现针对快速查找进行了优化。

Set 具有与 **Collection** 相同的接口，因此没有任何额外的功能，不像前面两种不同类型的 **List** 那样。实际上，**Set** 就是一个 **Collection**，只是行为不同。（这是继承和多态思想的典型应用：表现不同的行为。）**Set** 根据对象的“值”确定归属性，更复杂的内容将在附录：集合主题中介绍。

下面是使用存放 **Integer** 对象的 **HashSet** 的示例：

```
// collections/SetOfInteger.java
import java.util.*;

public class SetOfInteger {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Set<Integer> intset = new HashSet<>();
    for(int i = 0; i < 10000; i++)
      intset.add(rand.nextInt(30));
    System.out.println(intset);
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*/
```

在 0 到 29 之间的 10000 个随机整数被添加到 **Set** 中，因此可以想象每个值都重复了很多次。但是从结果中可以看到，每一个数只有一个实例出现在结果中。

早期 Java 版本中的 **HashSet** 产生的输出没有可辨别的顺序。这是因为出于对速度的追求，**HashSet** 使用了散列，请参阅附录：集合主题一章。由 **HashSet** 维护的顺序与 **TreeSet** 或 **LinkedHashSet** 不同，因为它们的实现具有不同的元素存储方式。**TreeSet** 将元素存储在红-黑树数

据结构中，而 **HashSet** 使用散列函数。**LinkedHashSet** 因为查询速度的原因也使用了散列，但是看起来使用了链表来维护元素的插入顺序。看起来散列算法好像已经改变了，现在 **Integer** 按顺序排序。但是，您不应该依赖此行为：

```
// collections/SetOfString.java
import java.util.*;

public class SetOfString {
  public static void main(String[] args) {
    Set<String> colors = new HashSet<>();
    for(int i = 0; i < 100; i++) {
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Red");
      colors.add("Red");
      colors.add("Orange");
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Purple");
    }
    System.out.println(colors);
  }
}
/* Output:
[Red, Yellow, Blue, Purple, Orange]
*/
```

String 对象似乎没有排序。要对结果进行排序，一种方法是使用 **TreeSet** 而不是 **HashSet**：

```
// collections/SortedSetOfString.java
import java.util.*;

public class SortedSetOfString {
  public static void main(String[] args) {
    Set<String> colors = new TreeSet<>();
    for(int i = 0; i < 100; i++) {
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Red");
      colors.add("Red");
      colors.add("Orange");
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Purple");
    }
    System.out.println(colors);
  }
}
/* Output:
[Blue, Orange, Purple, Red, Yellow]
```

```
*/
```

最常见的操作之一是使用 `contains()` 测试成员归属性，但也有一些其它操作，这可能会让你想起在小学学过的维恩图（译者注：利用图形的交合表示多个集合之间的逻辑关系）：

```
// collections/SetOperations.java
import java.util.*;

public class SetOperations {
  public static void main(String[] args) {
    Set<String> set1 = new HashSet<>();
    Collections.addAll(set1,
      "A B C D E F G H I J K L".split(" "));
    set1.add("M");
    System.out.println("H: " + set1.contains("H"));
    System.out.println("N: " + set1.contains("N"));
    Set<String> set2 = new HashSet<>();
    Collections.addAll(set2, "H I J K L".split(" "));
    System.out.println(
      "set2 in set1: " + set1.containsAll(set2));
    set1.remove("H");
    System.out.println("set1: " + set1);
    System.out.println(
      "set2 in set1: " + set1.containsAll(set2));
    set1.removeAll(set2);
    System.out.println(
      "set2 removed from set1: " + set1);
    Collections.addAll(set1, "X Y Z".split(" "));
    System.out.println(
      "'X Y Z' added to set1: " + set1);
  }
}
/* Output:
H: true
N: false
set2 in set1: true
set1: [A, B, C, D, E, F, G, I, J, K, L, M]
set2 in set1: false
set2 removed from set1: [A, B, C, D, E, F, G, M]
'X Y Z' added to set1: [A, B, C, D, E, F, G, M, X, Y,
Z]
*/
```

这些方法名都是自解释的，JDK 文档中还有一些其它的方法。

能够产生每个元素都唯一的列表是相当有用的功能。例如，假设想要列出上面的 **SetOperations.java** 文件中的所有单词，通过使用本书后面介绍的 `java.nio.file.Files.readAllLines()` 方法，可以打开一个文件，并将其作为一个 **List<String>** 读取，每个 **String** 都是输入文件中的一行：

```java
// collections/UniqueWords.java
import java.util.*;
import java.nio.file.*;

public class UniqueWords {
  public static void
  main(String[] args) throws Exception {
    List<String> lines = Files.readAllLines(
      Paths.get("SetOperations.java"));
    Set<String> words = new TreeSet<>();
    for(String line : lines)
      for(String word : line.split("\\W+"))
        if(word.trim().length() > 0)
          words.add(word);
    System.out.println(words);
  }
}
/* Output:
[A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K,
L, M, N, Output, Set, SetOperations, String, System, X,
Y, Z, add, addAll, added, args, class, collections,
contains, containsAll, false, from, import, in, java,
main, new, out, println, public, remove, removeAll,
removed, set1, set2, split, static, to, true, util,
void]
*/
```

我们逐步浏览文件中的每一行，并使用 `String.split()` 将其分解为单词，这里使用正则表达式 **\W+**，这意味着它会依据一个或多个（即 **+**）非单词字母来拆分字符串（正则表达式将在字符串章节介绍）。每个结果单词都会添加到 **Set words** 中。因为它是 **TreeSet**，所以对结果进行排序。这里，排序是按*字典顺序*（lexicographically）完成的，因此大写和小写字母位于不同的组中。如果想按*字母顺序*（alphabetically）对其进行排序，可以向 **TreeSet** 构造器传入 **String.CASE_INSENSITIVE_ORDER** 比较器（比较器是一个建立排序顺序的对象）：

```java
// collections/UniqueWordsAlphabetic.java
// Producing an alphabetic listing
import java.util.*;
import java.nio.file.*;

public class UniqueWordsAlphabetic {
  public static void
  main(String[] args) throws Exception {
    List<String> lines = Files.readAllLines(
      Paths.get("SetOperations.java"));
    Set<String> words =
      new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
    for(String line : lines)
      for(String word : line.split("\\W+"))
```

```
        if(word.trim().length() > 0)
          words.add(word);
    System.out.println(words);
  }
}
/* Output:
[A, add, addAll, added, args, B, C, class, collections,
contains, containsAll, D, E, F, false, from, G, H,
HashSet, I, import, in, J, java, K, L, M, main, N, new,
out, Output, println, public, remove, removeAll,
removed, Set, set1, set2, SetOperations, split, static,
String, System, to, true, util, void, X, Y, Z]
*/
```

Comparator 比较器将在数组章节详细介绍。

13.10 映射 Map

将对象映射到其他对象的能力是解决编程问题的有效方法。例如，考虑一个程序，它被用来检查 Java 的 **Random** 类的随机性。理想情况下，**Random** 会产生完美的数字分布，但为了测试这一点，则需要生成大量的随机数，并计算落在各种范围内的数字个数。**Map** 可以很容易地解决这个问题。在本例中，键是 **Random** 生成的数字，而值是该数字出现的次数：

```
// collections/Statistics.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Simple demonstration of HashMap
import java.util.*;

public class Statistics {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Map<Integer, Integer> m = new HashMap<>();
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      int r = rand.nextInt(20);
      Integer freq = m.get(r); // [1]
      m.put(r, freq == null ? 1 : freq + 1);
    }
    System.out.println(m);
  }
}
/* Output:
{0=481, 1=502, 2=489, 3=508, 4=481, 5=503, 6=519,
7=471, 8=468, 9=549, 10=513, 11=531, 12=521, 13=506,
14=477, 15=497, 16=533, 17=509, 18=478, 19=464}
```

```
*/
```

- **[1]** 自动包装机制将随机生成的 **int** 转换为可以与 **HashMap** 一起使用的 **Integer** 引用（不能使用基本类型的集合）。如果键不在集合中，则 `get()` 返回 **null**（这意味着该数字第一次出现）。否则，`get()` 会为键生成与之关联的 **Integer** 值，然后该值被递增（自动包装机制再次简化了表达式，但实际上确实发生了对 **Integer** 的装箱和拆箱）。

接下来的示例将使用一个 **String** 描述来查找 **Pet** 对象。它还展示了通过使用 `containsKey()` 和 `containsValue()` 方法去测试一个 **Map**，以查看它是否包含某个键或某个值：

```java
// collections/PetMap.java
import typeinfo.pets.*;
import java.util.*;

public class PetMap {
  public static void main(String[] args) {
    Map<String, Pet> petMap = new HashMap<>();
    petMap.put("My Cat", new Cat("Molly"));
    petMap.put("My Dog", new Dog("Ginger"));
    petMap.put("My Hamster", new Hamster("Bosco"));
    System.out.println(petMap);
    Pet dog = petMap.get("My Dog");
    System.out.println(dog);
    System.out.println(petMap.containsKey("My Dog"));
    System.out.println(petMap.containsValue(dog));
  }
}
/* Output:
{My Dog=Dog Ginger, My Cat=Cat Molly, My
Hamster=Hamster Bosco}
Dog Ginger
true
true
*/
```

Map 与数组和其他的 **Collection** 一样，可以轻松地扩展到多个维度，只需要创建一个值为 **Map** 的 **Map**（这些 **Map** 的值可以是其他集合，甚至是其他 **Map**）。因此，能够很容易地将集合组合起来以快速生成强大的数据结构。例如，假设你正在追踪有多个宠物的人，只需要一个 **Map<Person, List<Pet>>** 即可：

```

// collections/MapOfList.java
// {java collections.MapOfList}
package collections;
import typeinfo.pets.*;
import java.util.*;

public class MapOfList {
```

```
  public static final Map<Person, List< ? extends Pet>>
    petPeople = new HashMap<>();
  static {
    petPeople.put(new Person("Dawn"),
      Arrays.asList(
        new Cymric("Molly"),
        new Mutt("Spot")));
    petPeople.put(new Person("Kate"),
      Arrays.asList(new Cat("Shackleton"),
        new Cat("Elsie May"), new Dog("Margrett")));
    petPeople.put(new Person("Marilyn"),
      Arrays.asList(
        new Pug("Louie aka Louis Snorkelstein Dupree"),
        new Cat("Stanford"),
        new Cat("Pinkola")));
    petPeople.put(new Person("Luke"),
      Arrays.asList(
        new Rat("Fuzzy"), new Rat("Fizzy")));
    petPeople.put(new Person("Isaac"),
      Arrays.asList(new Rat("Freckly")));
  }
  public static void main(String[] args) {
    System.out.println("People: " + petPeople.keySet());
    System.out.println("Pets: " + petPeople.values());
    for(Person person : petPeople.keySet()) {
      System.out.println(person + " has:");
      for(Pet pet : petPeople.get(person))
        System.out.println("    " + pet);
    }
  }
}
/* Output:
People: [Person Dawn, Person Kate, Person Isaac, Person
Marilyn, Person Luke]
Pets: [[Cymric Molly, Mutt Spot], [Cat Shackleton, Cat
Elsie May, Dog Margrett], [Rat Freckly], [Pug Louie aka
Louis Snorkelstein Dupree, Cat Stanford, Cat Pinkola],
[Rat Fuzzy, Rat Fizzy]]
Person Dawn has:
    Cymric Molly
    Mutt Spot
Person Kate has:
    Cat Shackleton
    Cat Elsie May
    Dog Margrett
Person Isaac has:
    Rat Freckly
Person Marilyn has:
    Pug Louie aka Louis Snorkelstein Dupree
    Cat Stanford
    Cat Pinkola
```

```
Person Luke has:
    Rat Fuzzy
    Rat Fizzy
*/
```

Map 可以返回由其键组成的 **Set**，由其值组成的 **Collection**，或者其键值对的 **Set**。`keySet()` 方法生成由在 **petPeople** 中的所有键组成的 **Set**，它在 *for-in* 语句中被用来遍历该 **Map**。

13.11 队列 Queue

队列是一个典型的"先进先出"（FIFO）集合。即从集合的一端放入事物，再从另一端去获取它们，事物放入集合的顺序和被取出的顺序是相同的。队列通常被当做一种可靠的将对象从程序的某个区域传输到另一个区域的途径。队列在并发编程中尤为重要，因为它们可以安全地将对象从一个任务传输到另一个任务。

LinkedList 实现了 **Queue** 接口，并且提供了一些方法以支持队列行为，因此 **LinkedList** 可以用作 **Queue** 的一种实现。通过将 **LinkedList** 向上转换为 **Queue**，下面的示例使用了在 **Queue** 接口中与 **Queue** 相关 (Queue-specific) 的方法:

```java
// collections/QueueDemo.java
// Upcasting to a Queue from a LinkedList
import java.util.*;

public class QueueDemo {
  public static void printQ(Queue queue) {
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    Queue<Integer> queue = new LinkedList<>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      queue.offer(rand.nextInt(i + 10));
    printQ(queue);
    Queue<Character> qc = new LinkedList<>();
    for(char c : "Brontosaurus".toCharArray())
      qc.offer(c);
    printQ(qc);
  }
}
/* Output:
8 1 1 1 5 14 3 1 0 1
B r o n t o s a u r u s
*/
```

`offer()` 是与 **Queue** 相关的方法之一，它在允许的情况下，在队列的尾部插入一个元素，或者

返回 **false** 。`peek()` 和 `element()` 都返回队头元素而不删除它，但是如果队列为空，则 `element()` 抛出 **NoSuchElementException** ，而 `peek()` 返回 **null** 。`poll()` 和 `remove()`* 都删除并返回队头元素，但如果队列为空，`poll()` 返回 **null** ，而 `remove()` 抛出 **NoSuchElementException** 。

自动包装机制会自动将 `nextInt()` 的 **int** 结果转换为 **queue** 所需的 **Integer** 对象，并将 **char c** 转换为 **qc** 所需的 **Character** 对象。**Queue** 接口窄化了对 **LinkedList** 方法的访问权限，因此只有适当的方法才能使用，因此能够访问到的 **LinkedList** 的方法会变少（这里实际上可以将 **Queue** 强制转换回 **LinkedList** ，但至少我们不鼓励这样做）。

与 **Queue** 相关的方法提供了完整而独立的功能。也就是说，对于 **Queue** 所继承的 **Collection** ，在不需要使用它的任何方法的情况下，就可以拥有一个可用的 **Queue** 。

13.11.1 优先级队列 PriorityQueue

先进先出（FIFO）描述了最典型的*队列规则*（queuing discipline）。队列规则是指在给定队列中的一组元素的情况下，确定下一个弹出队列的元素的规则。先进先出声明的是下一个弹出的元素应该是等待时间最长的元素。

优先级队列声明下一个弹出的元素是最需要的元素（具有最高的优先级）。例如，在机场，当飞机临近起飞时，这架飞机的乘客可以在办理登机手续时排到队头。如果构建了一个消息传递系统，某些消息比其他消息更重要，应该尽快处理，而不管它们何时到达。在 Java 5 中添加了 **PriorityQueue** ，以便自动实现这种行为。

当在 **PriorityQueue** 上调用 `offer()` 方法来插入一个对象时，该对象会在队列中被排序。[5]默认的排序使用队列中对象的*自然顺序*（natural order），但是可以通过提供自己的 **Comparator** 来修改这个顺序。**PriorityQueue** 确保在调用 `peek()` ，`poll()` 或 `remove()` 方法时，获得的元素将是队列中优先级最高的元素。

让 **PriorityQueue** 与 **Integer** ，**String** 和 **Character** 这样的内置类型一起工作易如反掌。在下面的示例中，第一组值与前一个示例中的随机值相同，可以看到它们从 **PriorityQueue** 中弹出的顺序与前一个示例不同：

```java
// collections/PriorityQueueDemo.java
import java.util.*;

public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<Integer> priorityQueue =
      new PriorityQueue<>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      priorityQueue.offer(rand.nextInt(i + 10));
    QueueDemo.printQ(priorityQueue);
```

[5]我还没有验证过这种说法。

```

    List<Integer> ints = Arrays.asList(25, 22, 20,
      18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
    priorityQueue = new PriorityQueue<>(ints);
    QueueDemo.printQ(priorityQueue);
    priorityQueue = new PriorityQueue<>(
        ints.size(), Collections.reverseOrder());
    priorityQueue.addAll(ints);
    QueueDemo.printQ(priorityQueue);

    String fact = "EDUCATION SHOULD ESCHEW OBFUSCATION";
    List<String> strings =
      Arrays.asList(fact.split(""));
    PriorityQueue<String> stringPQ =
      new PriorityQueue<>(strings);
    QueueDemo.printQ(stringPQ);
    stringPQ = new PriorityQueue<>(
      strings.size(), Collections.reverseOrder());
    stringPQ.addAll(strings);
    QueueDemo.printQ(stringPQ);

    Set<Character> charSet = new HashSet<>();
    for(char c : fact.toCharArray())
      charSet.add(c); // Autoboxing
    PriorityQueue<Character> characterPQ =
      new PriorityQueue<>(charSet);
    QueueDemo.printQ(characterPQ);
  }
}
/* Output:
0 1 1 1 1 1 3 5 8 14
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
      A A B C C C D D E E E F H H I I L N N O O O O S S
S T T U U U W
W U U U T T S S S O O O O N N L I I H H F E E E D D C C
C B A A
  A B C D E F H I L N O S T U W
*/
```

PriorityQueue 是允许重复的，最小的值具有最高的优先级（如果是 **String** ，空格也可以算作值，并且比字母的优先级高）。为了展示如何通过提供自己的 **Comparator** 对象来改变顺序，第三个对 **PriorityQueue<Integer>** 构造器的调用，和第二个对 **PriorityQueue<String>** 的调用使用了由 `Collections.reverseOrder()` （Java 5 中新添加的）产生的反序的 **Comparator** 。

最后一部分添加了一个 **HashSet** 来消除重复的 **Character**。

Integer ，**String** 和 **Character** 可以与 **PriorityQueue** 一起使用，因为这些类已经内置了自然排序。如果想在 **PriorityQueue** 中使用自己的类，则必须包含额外的功能以产生自然排序，或者必须提供自己的 **Comparator** 。在附录：集合主题中有一个更复杂的示例来演示这种情况。

13.12 集合与迭代器

Collection 是所有序列集合共有的根接口。它可能会被认为是一种“附属接口”（incidental interface），即因为要表示其他若干个接口的共性而出现的接口。此外，**java.util.AbstractCollection** 类提供了 **Collection** 的默认实现，使得你可以创建 **AbstractCollection** 的子类型，而其中没有不必要的代码重复。

使用接口描述的一个理由是它可以使我们创建更通用的代码。通过针对接口而非具体实现来编写代码，我们的代码可以应用于更多类型的对象。[6]因此，如果所编写的方法接受一个 **Collection**，那么该方法可以应用于任何实现了 **Collection** 的类——这也就使得一个新类可以选择去实现 **Collection** 接口，以便该方法可以使用它。标准 C++ 类库中的的集合并没有共同的基类——集合之间的所有共性都是通过迭代器实现的。在 Java 中，遵循 C++ 的方式看起来似乎很明智，即用迭代器而不是 **Collection** 来表示集合之间的共性。但是，这两种方法绑定在了一起，因为实现 **Collection** 就意味着需要提供 `iterator()` 方法：

```java
// collections/InterfaceVsIterator.java
import typeinfo.pets.*;
import java.util.*;

public class InterfaceVsIterator {
  public static void display(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void display(Collection<Pet> pets) {
    for(Pet p : pets)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    List<Pet> petList = Pets.list(8);
    Set<Pet> petSet = new HashSet<>(petList);
    Map<String, Pet> petMap = new LinkedHashMap<>();
    String[] names = ("Ralph, Eric, Robin, Lacey, " +
      "Britney, Sam, Spot, Fluffy").split(", ");
    for(int i = 0; i < names.length; i++)
      petMap.put(names[i], petList.get(i));
    display(petList);
    display(petSet);
    display(petList.iterator());
    display(petSet.iterator());
    System.out.println(petMap);
```

[6]当你理解了并发编程章节的内容，你就能明白为什么更改共享变量“不是线程安全的”的了。

```
    System.out.println(petMap.keySet());
    display(petMap.values());
    display(petMap.values().iterator());
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
{Ralph=Rat, Eric=Manx, Robin=Cymric, Lacey=Mutt,
Britney=Pug, Sam=Cymric, Spot=Pug, Fluffy=Manx}
[Ralph, Eric, Robin, Lacey, Britney, Sam, Spot, Fluffy]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
*/
```

两个版本的 `display()` 方法都可以使用 **Map** 或 **Collection** 的子类型来工作。而且 **Collection** 接口和 **Iterator** 都将 `display()` 方法与低层集合的特定实现解耦。

在本例中，这两种方式都可以奏效。事实上，**Collection** 要更方便一点，因为它是 **Iterable** 类型，因此在 `display(Collection)` 的实现中可以使用 *for-in* 构造，这使得代码更加清晰。

当需要实现一个不是 **Collection** 的外部类时，由于让它去实现 **Collection** 接口可能非常困难或麻烦，因此使用 **Iterator** 就会变得非常吸引人。例如，如果我们通过继承一个持有 **Pet** 对象的类来创建一个 **Collection** 的实现，那么我们必须实现 **Collection** 所有的方法，即使我们不在 `display()` 方法中使用它们，也必须这样做。虽然这可以通过继承 **AbstractCollection** 而很容易地实现，但是无论如何还是要被强制去实现 `iterator()` 和 `size()` 方法，这些方法 **AbstractCollection** 没有实现，但是 **AbstractCollection** 中的其它方法会用到：

```
// collections/CollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

public class CollectionSequence
extends AbstractCollection<Pet> {
  private Pet[] pets = Pets.array(8);
  @Override
  public int size() { return pets.length; }
  @Override
  public Iterator<Pet> iterator() {
    return new Iterator<Pet>() { // [1]
      private int index = 0;
```

```
      @Override
      public boolean hasNext() {
        return index < pets.length;
      }
      @Override
      public Pet next() { return pets[index++]; }
      @Override
      public void remove() { // Not implemented
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    CollectionSequence c = new CollectionSequence();
    InterfaceVsIterator.display(c);
    InterfaceVsIterator.display(c.iterator());
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
*/
```

`remove()` 方法是一个“可选操作”，在附录：集合主题中详细介绍。这里可以不必实现它，如果你调用它，它将抛出异常。

- **[1]** 你可能会认为，因为 `iterator()` 返回 **Iterator<Pet>**，匿名内部类定义可以使用菱形语法，Java 可以推断出类型。但这不起作用，类型推断仍然非常有限。

这个例子表明，如果实现了 **Collection**，就必须实现 `iterator()`，并且只拿实现 `iterator()` 与继承 **AbstractCollection** 相比，花费的代价只有略微减少。但是，如果类已经继承了其他的类，那么就不能继承再 **AbstractCollection** 了。在这种情况下，要实现 **Collection**，就必须实现该接口中的所有方法。此时，继承并提供创建迭代器的能力要容易得多：

```
// collections/NonCollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

class PetSequence {
  protected Pet[] pets = Pets.array(8);
}

public class NonCollectionSequence extends PetSequence {
  public Iterator<Pet> iterator() {
    return new Iterator<Pet>() {
      private int index = 0;
      @Override
      public boolean hasNext() {
```

```
        return index < pets.length;
      }
      @Override
      public Pet next() { return pets[index++]; }
      @Override
      public void remove() { // Not implemented
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    NonCollectionSequence nc =
      new NonCollectionSequence();
    InterfaceVsIterator.display(nc.iterator());
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
*/
```

生成 **Iterator** 是将序列与消费该序列的方法连接在一起耦合度最小的方式，并且与实现 **Collection** 相比，它在序列类上所施加的约束也少得多。

13.13 for-in 和迭代器

到目前为止，*for-in* 语法主要用于数组，但它也适用于任何 **Collection** 对象。实际上在使用 **ArrayList** 时，已经看到了一些使用它的示例，下面是一个更通用的证明：

```
// collections/ForInCollections.java
// All collections work with for-in
import java.util.*;

public class ForInCollections {
  public static void main(String[] args) {
    Collection<String> cs = new LinkedList<>();
    Collections.addAll(cs,
      "Take the long way home".split(" "));
    for(String s : cs)
      System.out.print("'" + s + "' ");
  }
}
/* Output:
'Take' 'the' 'long' 'way' 'home'
*/
```

由于 **cs** 是一个 **Collection** ，因此该代码展示了使用 *for-in* 是所有 **Collection** 对象的特征。

这样做的原因是 Java 5 引入了一个名为 **Iterable** 的接口，该接口包含一个能够生成 **Iterator** 的 `iterator()` 方法。*for-in* 使用此 **Iterable** 接口来遍历序列。因此，如果创建了任何实现了 **Iterable** 的类，都可以将它用于 *for-in* 语句中：

```
// collections/IterableClass.java
// Anything Iterable works with for-in
import java.util.*;

public class IterableClass implements Iterable<String> {
  protected String[] words = ("And that is how " +
    "we know the Earth to be banana-shaped."
    ).split(" ");
  @Override
  public Iterator<String> iterator() {
    return new Iterator<String>() {
      private int index = 0;
      @Override
      public boolean hasNext() {
        return index < words.length;
      }
      @Override
      public String next() { return words[index++]; }
      @Override
      public void remove() { // Not implemented
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    for(String s : new IterableClass())
      System.out.print(s + " ");
  }
}
/* Output:
And that is how we know the Earth to be banana-shaped.
*/
```

`iterator()` 返回的是实现了 **Iterator<String>** 的匿名内部类的实例，该匿名内部类可以遍历数组中的每个单词。在主方法中，可以看到 **IterableClass** 确实可以用于 *for-in* 语句。

在 Java 5 中，许多类都是 **Iterable** ，主要包括所有的 **Collection** 类（但不包括各种 **Maps** ）。例如，下面的代码可以显示所有的操作系统环境变量：

```
// collections/EnvironmentVariables.java
// {VisuallyInspectOutput}
import java.util.*;

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

```
    for(Map.Entry entry: System.getenv().entrySet()) {
      System.out.println(entry.getKey() + ": " +
        entry.getValue());
    }
  }
}
```

`System.getenv()` [7]返回一个 **Map** ，`entrySet()` 产生一个由 **Map.Entry** 的元素构成的 **Set** ，并且这个 **Set** 是一个 **Iterable** ，因此它可以用于 *for-in* 循环。

for-in 语句适用于数组或其它任何 **Iterable** ，但这并不意味着数组肯定也是个 **Iterable** ，也不会发生任何自动装箱:

```
// collections/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {
  static <T> void test(Iterable<T> ib) {
    for(T t : ib)
      System.out.print(t + " ");
  }
  public static void main(String[] args) {
    test(Arrays.asList(1, 2, 3));
    String[] strings = { "A", "B", "C" };
    // An array works in for-in, but it's not Iterable:
    //- test(strings);
    // You must explicitly convert it to an Iterable:
    test(Arrays.asList(strings));
  }
}
/* Output:
1 2 3 A B C
*/
```

尝试将数组作为一个 **Iterable** 参数传递会导致失败。这说明不存在任何从数组到 **Iterable** 的自动转换; 必须手工执行这种转换。

13.13.1 适配器方法惯用法

如果现在有一个 **Iterable** 类，你想要添加一种或多种在 *for-in* 语句中使用这个类的方法，应该怎么做呢？例如，你希望可以选择正向还是反向遍历一个单词列表。如果直接继承这个类，并覆盖 `iterator()` 方法，则只能替换现有的方法，而不能实现遍历顺序的选择。

一种解决方案是所谓*适配器方法*（Adapter Method）的惯用法。“适配器”部分来自于设计模式，因为必须要提供特定的接口来满足 *for-in* 语句。如果已经有一个接口并且需要另一个接口时，则编写适配器就可以解决这个问题。在这里，若希望在默认的正向迭代器的基础上，添加产生反向迭代器的能力，

[7]接口能够支持方法的原因是它们是 Java 8 默认方法，你将在下一章中了解到。

因此不能使用覆盖，相反，而是添加了一个能够生成 **Iterable** 对象的方法，该对象可以用于 *for-in* 语句。这使得我们可以提供多种使用 *for-in* 语句的方式：

```
// collections/AdapterMethodIdiom.java
// The "Adapter Method" idiom uses for-in
// with additional kinds of Iterables
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
  ReversibleArrayList(Collection<T> c) {
    super(c);
  }
  public Iterable<T> reversed() {
    return new Iterable<T>() {
      public Iterator<T> iterator() {
        return new Iterator<T>() {
          int current = size() - 1;
          public boolean hasNext() {
            return current > -1;
          }
          public T next() { return get(current--); }
          public void remove() { // Not implemented
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}

public class AdapterMethodIdiom {
  public static void main(String[] args) {
    ReversibleArrayList<String> ral =
      new ReversibleArrayList<String>(
        Arrays.asList("To be or not to be".split(" ")));
    // Grabs the ordinary iterator via iterator():
    for(String s : ral)
      System.out.print(s + " ");
    System.out.println();
    // Hand it the Iterable of your choice
    for(String s : ral.reversed())
      System.out.print(s + " ");
  }
}
/* Output:
To be or not to be
be to not or be To
*/
```

在主方法中，如果直接将 **ral** 对象放在 *for-in* 语句中，则会得到（默认的）正向迭代器。但是如果

在该对象上调用 `reversed()` 方法，它会产生不同的行为。

通过使用这种方式，可以在 **IterableClass.java** 示例中添加两种适配器方法：

```
// collections/MultiIterableClass.java
// Adding several Adapter Methods
import java.util.*;

public class MultiIterableClass extends IterableClass {
  public Iterable<String> reversed() {
    return new Iterable<String>() {
      public Iterator<String> iterator() {
        return new Iterator<String>() {
          int current = words.length - 1;
          public boolean hasNext() {
            return current > -1;
          }
          public String next() {
            return words[current--];
          }
          public void remove() { // Not implemented
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
  public Iterable<String> randomized() {
    return new Iterable<String>() {
      public Iterator<String> iterator() {
        List<String> shuffled =
          new ArrayList<String>(Arrays.asList(words));
        Collections.shuffle(shuffled, new Random(47));
        return shuffled.iterator();
      }
    };
  }
  public static void main(String[] args) {
    MultiIterableClass mic = new MultiIterableClass();
    for(String s : mic.reversed())
      System.out.print(s + " ");
    System.out.println();
    for(String s : mic.randomized())
      System.out.print(s + " ");
    System.out.println();
    for(String s : mic)
      System.out.print(s + " ");
  }
}
/* Output:
banana-shaped. be to Earth the know we how is that And
```

```
is banana-shaped. Earth that how the be And we know to
And that is how we know the Earth to be banana-shaped.
*/
```

注意，第二个方法 `random()` 没有创建它自己的 **Iterator** ，而是直接返回被打乱的 **List** 中的 **Iterator** 。

从输出中可以看到,`Collections.shuffle()` 方法不会影响到原始数组,而只是打乱了 **shuffled** 中的引用。之所以这样，是因为 `randomized()` 方法用一个 **ArrayList** 将 `Arrays.asList()` 的结果包装了起来。如果这个由 `Arrays.asList()` 生成的 **List** 被直接打乱，那么它将修改底层数组，如下所示:

```java
// collections/ModifyingArraysAsList.java
import java.util.*;

public class ModifyingArraysAsList {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    List<Integer> list1 =
      new ArrayList<>(Arrays.asList(ia));
    System.out.println("Before shuffling: " + list1);
    Collections.shuffle(list1, rand);
    System.out.println("After shuffling: " + list1);
    System.out.println("array: " + Arrays.toString(ia));

    List<Integer> list2 = Arrays.asList(ia);
    System.out.println("Before shuffling: " + list2);
    Collections.shuffle(list2, rand);
    System.out.println("After shuffling: " + list2);
    System.out.println("array: " + Arrays.toString(ia));
  }
}
/* Output:
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
array: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*/
```

在第一种情况下，`Arrays.asList()` 的输出被传递给了 **ArrayList** 的构造器，这将创建一个引用 **ia** 的元素的 **ArrayList** ，因此打乱这些引用不会修改该数组。但是，如果直接使用 `Arrays.asList(ia)` 的结果，这种打乱就会修改 **ia** 的顺序。重要的是要注意 `Arrays.asList()` 生成一个 **List** 对象，该对象使用底层数组作为其物理实现。如果执行的操作会修改这个 **List** ，并且不希望修改原始数组，那么就应该在另一个集合中创建一个副本。

13.14 本章小结

Java 提供了许多保存对象的方法:

1. 数组将数字索引与对象相关联。它保存类型明确的对象，因此在查找对象时不必对结果做类型转换。它可以是多维的，可以保存基本类型的数据。虽然可以在运行时创建数组，但是一旦创建数组，就无法更改数组的大小。
2. **Collection** 保存单一的元素，而 **Map** 包含相关联的键值对。使用 Java 泛型，可以指定集合中保存的对象的类型，因此不能将错误类型的对象放入集合中，并且在从集合中获取元素时，不必进行类型转换。各种 **Collection** 和各种 **Map** 都可以在你向其中添加更多的元素时，自动调整其尺寸大小。集合不能保存基本类型，但自动装箱机制会负责执行基本类型和集合中保存的包装类型之间的双向转换。
3. 像数组一样，**List** 也将数字索引与对象相关联，因此，数组和 **List** 都是有序集合。
4. 如果要执行大量的随机访问，则使用 **ArrayList** ，如果要经常从表中间插入或删除元素，则应该使用 **LinkedList** 。
5. 队列和堆栈的行为是通过 **LinkedList** 提供的。
6. **Map** 是一种将对象（而非数字）与对象相关联的设计。**HashMap** 专为快速访问而设计，而 **TreeMap** 保持键始终处于排序状态，所以没有 **HashMap** 快。**LinkedHashMap** 按插入顺序保存其元素，但使用散列提供快速访问的能力。
7. **Set** 不接受重复元素。**HashSet** 提供最快的查询速度，而 **TreeSet** 保持元素处于排序状态。**LinkedHashSet** 按插入顺序保存其元素，但使用散列提供快速访问的能力。
8. 不要在新代码中使用遗留类 **Vector** ，**Hashtable** 和 **Stack** 。

浏览一下 Java 集合的简图（不包含抽象类或遗留组件）会很有帮助。这里仅包括在一般情况下会碰到的接口和类。（译者注：下图为原著 PDF 中的截图，可能由于未知原因存在问题。这里可参考译者绘制版[8]）

[8]一些语言，如 Python，允许像调用其他函数一样调用组合函数。但这是 Java，所以我们做做可为之事。

Figure 13.1: simple collection taxonomy

13.14.1 简单集合分类

可以看到，实际上只有四个基本的集合组件：**Map**，**List**，**Set** 和 **Queue**，它们各有两到三个实现版本（**Queue** 的 **java.util.concurrent** 实现未包含在此图中）。最常使用的集合用黑色粗线线框表示。

虚线框表示接口，实线框表示普通的（具体的）类。带有空心箭头的虚线表示特定的类实现了一个接口。实心箭头表示某个类可以生成箭头指向的类的对象。例如，任何 **Collection** 都可以生成 **Iterator**，**List** 可以生成 **ListIterator**（也能生成普通的 **Iterator**，因为 **List** 继承自 **Collection**）。

下面的示例展示了各种不同的类在方法上的差异。实际代码来自泛型章节，在这里只是调用它来产生输出。程序的输出还展示了在每个类或接口中所实现的接口：

```
// collections/CollectionDifferences.java
import onjava.*;

public class CollectionDifferences {
  public static void main(String[] args) {
    CollectionMethodDifferences.main(args);
  }
}
/* Output:
Collection: [add, addAll, clear, contains, containsAll,
equals, forEach, hashCode, isEmpty, iterator,
parallelStream, remove, removeAll, removeIf, retainAll,
size, spliterator, stream, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
```

```
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable,
Serializable]
TreeSet extends Set, adds: [headSet,
descendingIterator, descendingSet, pollLast, subSet,
floor, tailSet, ceiling, last, lower, comparator,
pollFirst, first, higher]
Interfaces in TreeSet: [NavigableSet, Cloneable,
Serializable]
List extends Collection, adds: [replaceAll, get,
indexOf, subList, set, sort, lastIndexOf, listIterator]
Interfaces in List: [Collection]
ArrayList extends List, adds: [trimToSize,
ensureCapacity]
Interfaces in ArrayList: [List, RandomAccess,
Cloneable, Serializable]
LinkedList extends List, adds: [offerFirst, poll,
getLast, offer, getFirst, removeFirst, element,
removeLastOccurrence, peekFirst, peekLast, push,
pollFirst, removeFirstOccurrence, descendingIterator,
pollLast, removeLast, pop, addLast, peek, offerLast,
addFirst]
Interfaces in LinkedList: [List, Deque, Cloneable,
Serializable]
Queue extends Collection, adds: [poll, peek, offer,
element]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, compute, computeIfAbsent,
computeIfPresent, containsKey, containsValue, entrySet,
equals, forEach, get, getOrDefault, hashCode, isEmpty,
keySet, merge, put, putAll, putIfAbsent, remove,
replace, replaceAll, size, values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [lastKey, subMap,
comparator, firstKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingKeySet,
navigableKeySet, higherEntry, higherKey, floorKey,
subMap, ceilingKey, pollLastEntry, firstKey, lowerKey,
headMap, tailMap, lowerEntry, ceilingEntry,
descendingMap, pollFirstEntry, lastKey, firstEntry,
floorEntry, comparator, lastEntry]
Interfaces in TreeMap: [NavigableMap, Cloneable,
```

```
Serializable]
*/
```

除 **TreeSet** 之外的所有 **Set** 都具有与 **Collection** 完全相同的接口。**List** 和 **Collection** 存在着明显的不同，尽管 **List** 所要求的方法都在 **Collection** 中。另一方面，在 **Queue** 接口中的方法是独立的，在创建具有 **Queue** 功能的实现时，不需要使用 **Collection** 方法。最后，**Map** 和 **Collection** 之间唯一的交集是 **Map** 可以使用 `entrySet()` 和 `values()` 方法来产生 **Collection** 。

请注意，标记接口 **java.util.RandomAccess** 附加到了 **ArrayList** 上，但不附加到 **LinkedList** 上。这为根据特定 **List** 动态改变其行为的算法提供了信息。

从面向对象的继承层次结构来看，这种组织结构确实有些奇怪。但是，当了解了 **java.util** 中更多的有关集合的内容后（特别是在附录：集合主题中的内容），就会发现出了继承结构有点奇怪外，还有更多的问题。集合类库一直以来都是设计难题——解决这些问题涉及到要去满足经常彼此之间互为牵制的各方面需求。所以要做好准备，在各处做出妥协。

尽管存在这些问题，但 Java 集合仍是在日常工作中使用的基本工具，它可以使程序更简洁、更强大、更有效。你可能需要一段时间才能熟悉集合类库的某些方面，但我想你很快就会找到自己的路子，来获得和使用这个类库中的类。

早重构, 常重构

14 函数式编程

> 函数式编程语言操纵代码片段就像操作数据一样容易。虽然 Java 不是函数式语言，但 Java 8 Lambda 表达式和方法引用 (Method References) 允许你以函数式编程。

在计算机时代早期，内存是稀缺和昂贵的。几乎每个人都用汇编语言编程。人们对编译器有所了解，但仅仅想到编译生成的代码肯定会比手工编码多很多字节。

通常，只是为了使程序适合有限的内存，程序员通过修改内存中的代码来节省代码空间，以便在程序执行时执行不同的操作。这种技术被称为**自修改代码**（self-modifying code）。只要程序足够小，少数人可以维护所有棘手和神秘的汇编代码，你就可以让它运行起来。

随着内存和处理器变得更便宜、更快。C 语言出现并被大多数汇编程序员认为更“高级”。人们发现使用 C 可以显著提高生产力。同时，使用 C 创建自修改代码仍然不难。

随着硬件越来越便宜，程序的规模和复杂性都在增长。这一切只是让程序工作变得困难。我们想方设法使代码更加一致和易懂。使用纯粹的自修改代码造成的结果就是：我们很难确定程序在做什么。它也难以测试：除非你想一点点测试输出，代码转换和修改等等过程?

然而，使用代码以某种方式操纵其他代码的想法也很有趣，只要能保证它更安全。从代码创建，维护和可靠性的角度来看，这个想法非常吸引人。我们不用从头开始编写大量代码，而是从易于理解、充分测试及可靠的现有小块开始，最后将它们组合在一起以创建新代码。难道这不会让我们更有效率，同时创造更健壮的代码吗?

这就是**函数式编程**（FP）的意义所在。通过合并现有代码来生成新功能而不是从头开始编写所有内容，我们可以更快地获得更可靠的代码。至少在某些情况下，这套理论似乎很有用。在这一过程中，一些非函数式语言已经习惯了使用函数式编程产生的优雅的语法。

你也可以这样想：

OO（object oriented，面向对象）是抽象数据，FP（functional programming，函数式编程）是抽象行为。

纯粹的函数式语言在安全性方面更进一步。它强加了额外的约束，即所有数据必须是不可变的：设置一次，永不改变。将值传递给函数，该函数然后生成新值但从不修改自身外部的任何东西（包括其参数或该函数范围之外的元素）。当强制执行此操作时，你知道任何错误都不是由所谓的副作用引起的，因为该函数仅创建并返回结果，而不是其他任何错误。

更好的是，“不可变对象和无副作用”范式解决了并发编程中最基本和最棘手的问题之一（当程序的某些部分同时在多个处理器上运行时）。这是可变共享状态的问题，这意味着代码的不同部分（在不同的

处理器上运行）可以尝试同时修改同一块内存（谁赢了？没人知道）。如果函数永远不会修改现有值但只生成新值，则不会对内存产生争用，这是纯函数式语言的定义。因此，经常提出纯函数式语言作为并行编程的解决方案（还有其他可行的解决方案）。

需要提醒大家的是，函数式语言背后有很多动机，这意味着描述它们可能会有些混淆。它通常取决于各种观点：为“并行编程”，“代码可靠性”和“代码创建和库复用”。[1] 关于函数式编程能高效创建更健壮的代码这一观点仍存在部分争议。虽然已有一些好的范例[2]，但还不足以证明纯函数式语言就是解决编程问题的最佳方法。

FP 思想值得融入非 FP 语言，如 Python。Java 8 也从中吸收并支持了 FP。我们将在此章探讨。

14.1 新旧对比

通常，传递给方法的数据不同，结果不同。如果我们希望方法在调用时行为不同，该怎么做呢？结论是：只要能将代码传递给方法，我们就可以控制它的行为。此前，我们通过在方法中创建包含所需行为的对象，然后将该对象传递给我们想要控制的方法来完成此操作。下面我们用传统形式和 Java 8 的方法引用、Lambda 表达式分别演示。代码示例：

```
// functional/Strategize.java

interface Strategy {
  String approach(String msg);
}

class Soft implements Strategy {
  public String approach(String msg) {
    return msg.toLowerCase() + "?";
  }
}

class Unrelated {
  static String twice(String msg) {
    return msg + " " + msg;
  }
}

public class Strategize {
  Strategy strategy;
  String msg;
  Strategize(String msg) {
    strategy = new Soft(); // [1]
    this.msg = msg;
  }
```

[1]功能粘贴在一起的方法的确有点与众不同，但它仍不失为一个库。
[2]例如，这个电子书是利用 Pandoc 制作出来的，它是用纯函数式语言 Haskell 编写的一个程序。

```java

  void communicate() {
    System.out.println(strategy.approach(msg));
  }

  void changeStrategy(Strategy strategy) {
    this.strategy = strategy;
  }

  public static void main(String[] args) {
    Strategy[] strategies = {
      new Strategy() { // [2]
        public String approach(String msg) {
          return msg.toUpperCase() + "!";
        }
      },
      msg -> msg.substring(0, 5), // [3]
      Unrelated::twice // [4]
    };
    Strategize s = new Strategize("Hello there");
    s.communicate();
    for(Strategy newStrategy : strategies) {
      s.changeStrategy(newStrategy); // [5]
      s.communicate(); // [6]
    }
  }
}
```

输出结果:

```
hello there?
HELLO THERE!
Hello
Hello there Hello there
```

Strategy 接口提供了单一的 `approach()` 方法来承载函数式功能。通过创建不同的 **Strategy** 对象，我们可以创建不同的行为。

传统上，我们通过创建一个实现 **Strategy** 接口的类来实现此行为，比如在 **Soft**。

- **[1]** 在 **Strategize** 中，**Soft** 作为默认策略，在构造函数中赋值。
- **[2]** 一种略显简短且更自发的方法是创建一个**匿名内部类**。即使这样，仍有相当数量的冗余代码。你总是要仔细观察："哦，原来这样，这里使用了匿名内部类。"
- **[3]** Java 8 的 Lambda 表达式。由箭头 `->` 分隔开参数和函数体，箭头左边是参数，箭头右侧是从 Lambda 返回的表达式，即函数体。这实现了与定义类、匿名内部类相同的效果，但代码少得多。
- **[4]** Java 8 的**方法引用**，由 `::` 区分。在 `::` 的左边是类或对象的名称，在 `::` 的右边是方法的名称，但没有参数列表。

- **[5]** 在使用默认的 **Soft strategy** 之后，我们逐步遍历数组中的所有 **Strategy**，并使用 `changeStrategy()` 方法将每个 **Strategy** 放入变量 `s` 中。
- **[6]** 现在，每次调用 `communicate()` 都会产生不同的行为，具体取决于此刻正在使用的策略**代码对象**。我们传递的是行为，而非仅数据。[3]

在 Java 8 之前，我们能够通过 **[1]** 和 **[2]** 的方式传递功能。然而，这种语法的读写非常笨拙，并且我们别无选择。方法引用和 Lambda 表达式的出现让我们可以在需要时**传递功能**，而不是仅在必要才这么做。

14.2 Lambda 表达式

Lambda 表达式是使用**最小可能**语法编写的函数定义：

1. Lambda 表达式产生函数，而不是类。在 JVM（Java Virtual Machine，Java 虚拟机）上，一切都是一个类，因此在幕后执行各种操作使 Lambda 看起来像函数 —— 但作为程序员，你可以高兴地假装它们“只是函数”。
2. Lambda 语法尽可能少，这正是为了使 Lambda 易于编写和使用。

我们在 **Strategize.java** 中看到了一个 Lambda 表达式，但还有其他语法变体：

```
// functional/LambdaExpressions.java

interface Description {
  String brief();
}

interface Body {
  String detailed(String head);
}

interface Multi {
  String twoArg(String head, Double d);
}

public class LambdaExpressions {

  static Body bod = h -> h + " No Parens!"; // [1]

  static Body bod2 = (h) -> h + " More details"; // [2]

  static Description desc = () -> "Short info"; // [3]

  static Multi mult = (h, n) -> h + n; // [4]

  static Description moreLines = () -> { // [5]
```

[3] 有时函数式语言将其描述为“代码即数据”。

```
    System.out.println("moreLines()");
    return "from moreLines()";
  };

  public static void main(String[] args) {
    System.out.println(bod.detailed("Oh!"));
    System.out.println(bod2.detailed("Hi!"));
    System.out.println(desc.brief());
    System.out.println(mult.twoArg("Pi! ", 3.14159));
    System.out.println(moreLines.brief());
  }
}
```

输出结果:

```
Oh! No Parens!
Hi! More details
Short info
Pi! 3.14159
moreLines()
from moreLines()
```

我们从三个接口开始，每个接口都有一个单独的方法（很快就会理解它的重要性）。但是，每个方法都有不同数量的参数，以便演示 Lambda 表达式语法。

任何 Lambda 表达式的基本语法是:

1. 参数。
2. 接着 `->`，可视为“产出”。
3. `->` 之后的内容都是方法体。

- **[1]** 当只用一个参数，可以不需要括号 `()`。然而，这是一个特例。
- **[2]** 正常情况使用括号 `()` 包裹参数。为了保持一致性，也可以使用括号 `()` 包裹单个参数，虽然这种情况并不常见。
- **[3]** 如果没有参数，则必须使用括号 `()` 表示空参数列表。
- **[4]** 对于多个参数，将参数列表放在括号 `()` 中。

到目前为止，所有 Lambda 表达式方法体都是单行。该表达式的结果自动成为 Lambda 表达式的返回值，在此处使用 **return** 关键字是非法的。这是 Lambda 表达式缩写用于描述功能的语法的另一种方式。

[5] 如果在 Lambda 表达式中确实需要多行，则必须将这些行放在花括号中。在这种情况下，就需要使用 **return**。

Lambda 表达式通常比**匿名内部类**产生更易读的代码，因此我们将在本书中尽可能使用它们。

14.2.1 递归

递归函数是一个自我调用的函数。可以编写递归的 Lambda 表达式，但需要注意：递归方法必须是实例变量或静态变量，否则会出现编译时错误。我们将为每个案例创建一个示例。

这两个示例都需要一个接受 **int** 型参数并生成 **int** 的接口：

```
// functional/IntCall.java

interface IntCall {
  int call(int arg);
}
```

整数 n 的阶乘将所有小于或等于 n 的正整数相乘。阶乘函数是一个常见的递归示例：

```
// functional/RecursiveFactorial.java

public class RecursiveFactorial {
  static IntCall fact;
  public static void main(String[] args) {
    fact = n -> n == 0 ? 1 : n * fact.call(n - 1);
    for(int i = 0; i <= 10; i++)
      System.out.println(fact.call(i));
  }
}
```

输出结果：

```
1
1
2
6
24
120
720
5040
40320
362880
3628800
```

这里，`fact` 是一个静态变量。注意使用三元 **if-else**。递归函数将一直调用自己，直到 `i == 0`。所有递归函数都有“停止条件”，否则将无限递归并产生异常。

我们可以将 `Fibonacci` 序列改为使用递归 Lambda 表达式来实现，这次使用实例变量：

```
// functional/RecursiveFibonacci.java

public class RecursiveFibonacci {
  IntCall fib;

  RecursiveFibonacci() {
```

```
    fib = n -> n == 0 ? 0 :
               n == 1 ? 1 :
               fib.call(n - 1) + fib.call(n - 2);
  }

  int fibonacci(int n) { return fib.call(n); }

  public static void main(String[] args) {
    RecursiveFibonacci rf = new RecursiveFibonacci();
    for(int i = 0; i <= 10; i++)
      System.out.println(rf.fibonacci(i));
  }
}
```

输出结果:

```
0
1
1
2
3
5
8
13
21
34
55
```

将 `Fibonacci` 序列中的最后两个元素求和来产生下一个元素。

14.3 方法引用

Java 8 方法引用没有历史包袱。方法引用组成：类名或对象名，后面跟 `::`[4]，然后跟方法名称。

```
// functional/MethodReferences.java

import java.util.*;

interface Callable { // [1]
  void call(String s);
}

class Describe {
  void show(String msg) { // [2]
    System.out.println(msg);
  }
}
```

[4]这个语法来自 C++。

```java

public class MethodReferences {
  static void hello(String name) { // [3]
    System.out.println("Hello, " + name);
  }
  static class Description {
    String about;
    Description(String desc) { about = desc; }
    void help(String msg) { // [4]
      System.out.println(about + " " + msg);
    }
  }
  static class Helper {
    static void assist(String msg) { // [5]
      System.out.println(msg);
    }
  }
  public static void main(String[] args) {
    Describe d = new Describe();
    Callable c = d::show; // [6]
    c.call("call()"); // [7]

    c = MethodReferences::hello; // [8]
    c.call("Bob");

    c = new Description("valuable")::help; // [9]
    c.call("information");

    c = Helper::assist; // [10]
    c.call("Help!");
  }
}
```

输出结果:

```
call()
Hello, Bob
valuable information
Help!
```

[1] 我们从单一方法接口开始（同样，你很快就会了解到这一点的重要性）。

[2] `show()` 的签名（参数类型和返回类型）符合 **Callable** 的 `call()` 的签名。

[3] `hello()` 也符合 `call()` 的签名。

[4] `help()` 也符合，它是静态内部类中的非静态方法。

[5] `assist()` 是静态内部类中的静态方法。

[6] 我们将 **Describe** 对象的方法引用赋值给 **Callable**，它没有 `show()` 方法，而是 `call()` 方法。但是，Java 似乎接受用这个看似奇怪的赋值，因为方法引用符合 **Callable** 的 `call()` 方法的签名。

[7] 我们现在可以通过调用 `call()` 来调用 `show()`，因为 Java 将 `call()` 映射到 `show()`。

[8] 这是一个**静态**方法引用。

[9] 这是 **[6]** 的另一个版本：对已实例化对象的方法的引用，有时称为*绑定方法引用*。

[10] 最后，获取静态内部类的方法引用的操作与 **[8]** 中外部类方式一样。

上例只是简短的介绍，我们很快就能看到方法引用的全部变化。

14.3.1 Runnable 接口

Runnable 接口自 1.0 版以来一直在 Java 中，因此不需要导入。它也符合特殊的单方法接口格式：它的方法 `run()` 不带参数，也没有返回值。因此，我们可以使用 Lambda 表达式和方法引用作为 **Runnable**：

```java
// functional/RunnableMethodReference.java

// 方法引用与 Runnable 接口的结合使用

class Go {
  static void go() {
    System.out.println("Go::go()");
  }
}

public class RunnableMethodReference {
  public static void main(String[] args) {

    new Thread(new Runnable() {
      public void run() {
        System.out.println("Anonymous");
      }
    }).start();

    new Thread(
      () -> System.out.println("lambda")
    ).start();

    new Thread(Go::go).start();
  }
}
```

输出结果：

```
Anonymous
lambda
Go::go()
```

Thread 对象将 **Runnable** 作为其构造函数参数，并具有会调用 `run()` 的方法 `start()`。**注意**，

只有**匿名内部类**才需要具有名为 `run()` 的方法。

14.3.2 未绑定的方法引用

未绑定的方法引用是指没有关联对象的普通（非静态）方法。使用未绑定的引用之前，我们必须先提供对象：

```
// functional/UnboundMethodReference.java

// 没有方法引用的对象

class X {
  String f() { return "X::f()"; }
}

interface MakeString {
  String make();
}

interface TransformX {
  String transform(X x);
}

public class UnboundMethodReference {
  public static void main(String[] args) {
    // MakeString ms = X::f; // [1]
    TransformX sp = X::f;
    X x = new X();
    System.out.println(sp.transform(x)); // [2]
    System.out.println(x.f()); // 同等效果
  }
}
```

输出结果：

```
X::f()
X::f()
```

截止目前，我们已经知道了与接口方法同名的方法引用。在 **[1]**，我们尝试把 `X` 的 `f()` 方法引用赋值给 **MakeString**。结果：即使 `make()` 与 `f()` 具有相同的签名，编译也会报“invalid method reference”（无效方法引用）错误。这是因为实际上还有另一个隐藏的参数：我们的老朋友 **`this`**。你不能在没有 `X` 对象的前提下调用 `f()`。因此，`X :: f` 表示未绑定的方法引用，因为它尚未“绑定”到对象。

要解决这个问题，我们需要一个 `X` 对象，所以我们的接口实际上需要一个额外的参数的接口，如上例中的 **TransformX**。如果将 `X :: f` 赋值给 **TransformX**，这在 Java 中是允许的。这次我们需要调整下心里预期——使用未绑定的引用时，函数方法的签名（接口中的单个方法）不再与方法引用的签

名完全匹配。理由是：你需要一个对象来调用方法。

[2] 的结果有点像脑筋急转弯。我接受未绑定的引用并对其调用 `transform()`，将其传递给 `X`，并以某种方式导致对 `x.f()` 的调用。Java 知道它必须采用第一个参数，这实际上就是 **this**，并在其上调用方法。

```
// functional/MultiUnbound.java

// 未绑定的方法与多参数的结合运用

class This {
  void two(int i, double d) {}
  void three(int i, double d, String s) {}
  void four(int i, double d, String s, char c) {}
}

interface TwoArgs {
  void call2(This athis, int i, double d);
}

interface ThreeArgs {
  void call3(This athis, int i, double d, String s);
}

interface FourArgs {
  void call4(
    This athis, int i, double d, String s, char c);
}

public class MultiUnbound {
  public static void main(String[] args) {
    TwoArgs twoargs = This::two;
    ThreeArgs threeargs = This::three;
    FourArgs fourargs = This::four;
    This athis = new This();
    twoargs.call2(athis, 11, 3.14);
    threeargs.call3(athis, 11, 3.14, "Three");
    fourargs.call4(athis, 11, 3.14, "Four", 'Z');
  }
}
```

为了说明这一点，我将类命名为 **This**，函数方法的第一个参数则是 **athis**，但是你应该选择其他名称以防止生产代码混淆。

14.3.3 构造函数引用

你还可以捕获构造函数的引用，然后通过引用调用该构造函数。

```
// functional/CtorReference.java
```

```

class Dog {
  String name;
  int age = -1; // For "unknown"
  Dog() { name = "stray"; }
  Dog(String nm) { name = nm; }
  Dog(String nm, int yrs) { name = nm; age = yrs; }
}

interface MakeNoArgs {
  Dog make();
}

interface Make1Arg {
  Dog make(String nm);
}

interface Make2Args {
  Dog make(String nm, int age);
}

public class CtorReference {
  public static void main(String[] args) {
    MakeNoArgs mna = Dog::new; // [1]
    Make1Arg m1a = Dog::new;   // [2]
    Make2Args m2a = Dog::new;  // [3]

    Dog dn = mna.make();
    Dog d1 = m1a.make("Comet");
    Dog d2 = m2a.make("Ralph", 4);
  }
}
```

Dog 有三个构造函数，函数接口内的 `make()` 方法反映了构造函数参数列表（`make()` 方法名称可以不同）。

注意我们如何对 **[1]**，**[2]** 和 **[3]** 中的每一个使用 `Dog :: new`。这 3 个构造函数只有一个相同名称：`:: new`，但在每种情况下都赋值给不同的接口。编译器可以检测并知道从哪个构造函数引用。

编译器能识别并调用你的构造函数（在本例中为 `make()`）。

14.4 函数式接口

方法引用和 Lambda 表达式必须被赋值，同时编译器需要识别类型信息以确保类型正确。Lambda 表达式特别引入了新的要求。代码示例：

```
x -> x.toString()
```

我们清楚这里返回类型必须是 **String**，但 x 是什么类型呢？

Lambda 表达式包含类型推导（编译器会自动推导出类型信息，避免了程序员显式地声明）。编译器必须能够以某种方式推导出 x 的类型。

下面是第 2 个代码示例：

```
(x, y) -> x + y
```

现在 x 和 y 可以是任何支持 + 运算符连接的数据类型，可以是两个不同的数值类型或者是 1 个 **String** 加任意一种可自动转换为 **String** 的数据类型（这包括了大多数类型）。但是，当 Lambda 表达式被赋值时，编译器必须确定 x 和 y 的确切类型以生成正确的代码。

该问题也适用于方法引用。假设你要传递 System.out :: println 到你正在编写的方法，你怎么知道传递给方法的参数的类型？

为了解决这个问题，Java 8 引入了 java.util.function 包。它包含一组接口，这些接口是 Lambda 表达式和方法引用的目标类型。每个接口只包含一个抽象方法，称为函数式方法。

在编写接口时，可以使用 @FunctionalInterface 注解强制执行此“函数式方法”模式：

```
// functional/FunctionalAnnotation.java

@FunctionalInterface
interface Functional {
  String goodbye(String arg);
}

interface FunctionalNoAnn {
  String goodbye(String arg);
}

/*
@FunctionalInterface
interface NotFunctional {
  String goodbye(String arg);
  String hello(String arg);
}
产生错误信息：
NotFunctional is not a functional interface
multiple non-overriding abstract methods
found in interface NotFunctional
*/

public class FunctionalAnnotation {
  public String goodbye(String arg) {
    return "Goodbye, " + arg;
  }
  public static void main(String[] args) {
    FunctionalAnnotation fa =
      new FunctionalAnnotation();
```

```
    Functional f = fa::goodbye;
    FunctionalNoAnn fna = fa::goodbye;
    // Functional fac = fa; // Incompatible
    Functional fl = a -> "Goodbye, " + a;
    FunctionalNoAnn fnal = a -> "Goodbye, " + a;
  }
}
```

`@FunctionalInterface` 注解是可选的; Java 在 `main()` 中把 **Functional** 和 **FunctionalNoAnn** 都当作函数式接口。`@FunctionalInterface` 的值在 `NotFunctional` 的定义中可见：接口中如果有多个方法则会产生编译时错误消息。

仔细观察在定义 `f` 和 `fna` 时发生了什么。`Functional` 和 `FunctionalNoAnn` 定义接口，然而被赋值的只是方法 `goodbye()`。首先，这只是一个方法而不是类；其次，它甚至都不是实现了该接口的类中的方法。Java 8 在这里添加了一点小魔法：如果将方法引用或 Lambda 表达式赋值给函数式接口（类型需要匹配），Java 会适配你的赋值到目标接口。编译器会自动包装方法引用或 Lambda 表达式到实现目标接口的类的实例中。

尽管 `FunctionalAnnotation` 确实适合 `Functional` 模型，但 Java 不允许我们将 `FunctionalAnnotation` 像 `fac` 定义一样直接赋值给 `Functional`，因为它没有明确地实现 `Functional` 接口。令人惊奇的是，Java 8 允许我们以简便的语法为接口赋值函数。

`java.util.function` 包旨在创建一组完整的目标接口，使得我们一般情况下不需再定义自己的接口。这主要是因为基本类型会产生一小部分接口。如果你了解命名模式，顾名思义就能知道特定接口的作用。

以下是基本命名准则：

1. 如果只处理对象而非基本类型，名称则为 `Function`，`Consumer`，`Predicate` 等。参数类型通过泛型添加。
2. 如果接收的参数是基本类型，则由名称的第一部分表示，如 `LongConsumer`，`DoubleFunction`，`IntPredicate` 等，但基本 `Supplier` 类型例外。
3. 如果返回值为基本类型，则用 `To` 表示，如 `ToLongFunction <T>` 和 `IntToLongFunction`。
4. 如果返回值类型与参数类型一致，则是一个运算符：单个参数使用 `UnaryOperator`，两个参数使用 `BinaryOperator`。
5. 如果接收两个参数且返回值为布尔值，则是一个谓词（Predicate）。
6. 如果接收的两个参数类型不同，则名称中有一个 `Bi`。

下表描述了 `java.util.function` 中的目标类型（包括例外情况）：

特征	函数式方法名	示例
无参数；无返回值	**Runnable** (java.lang) `run()`	**Runnable**

特征	函数式方法名	示例
无参数；返回类型任意	**Supplier** `get()` `getAs类型()`	**Supplier<T> BooleanSupplier IntSupplier LongSupplier DoubleSupplier**
无参数；返回类型任意	**Callable** (java.util.concurrent) `call()`	**Callable<V>**
1 参数；无返回值	**Consumer** `accept()`	**Consumer<T> IntConsumer LongConsumer DoubleConsumer**
2 参数 **Consumer**	**BiConsumer** `accept()`	**BiConsumer<T,U>**
2 参数 **Consumer**；1 引用；1 基本类型	**Obj 类型 Consumer** `accept()`	**ObjIntConsumer<T> ObjLongConsumer<T> ObjDoubleConsumer<T>**
1 参数；返回类型不同	**Function** `apply()` **To 类型**和**类型 To 类型** `applyAs类型()`	**Function<T,R> IntFunction<R> LongFunction<R> DoubleFunction<R> ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T> IntToLongFunction IntToDoubleFunction LongToIntFunction LongToDoubleFunction DoubleToIntFunction DoubleToLongFunction**
1 参数；返回类型相同	**UnaryOperator** `apply()`	**UnaryOperator<T> IntUnaryOperator LongUnaryOperator DoubleUnaryOperator**
2 参数类型相同；返回类型相同	**BinaryOperator** `apply()`	**BinaryOperator<T> IntBinaryOperator LongBinaryOperator DoubleBinaryOperator**

特征	函数式方法名	示例
2 参数类型相同; 返回整型	Comparator (java.util) `compare()`	`Comparator<T>`
2 参数; 返回布尔型	**Predicate** `test()`	`Predicate<T>` **BiPredicate** `<T,U>` **IntPredicate LongPredicate DoublePredicate**
参数基本类型; 返回基本类型	**类型 To 类型 Function** `applyAs类型()`	**IntToLongFunction IntToDoubleFunction LongToIntFunction LongToDoubleFunction DoubleToIntFunction DoubleToLongFunction**
2 参数类型不同	**Bi 操作** (不同方法名)	`BiFunction<T,U,R>` `BiConsumer<T,U>` `BiPredicate<T,U>` `ToIntBiFunction<T,U>` `ToLongBiFunction<T,U>` `ToDoubleBiFunction<T>`

此表仅提供些常规方案。通过上表，你应该或多或少能自行推导出更多行的函数式接口。

可以看出，在创建 `java.util.function` 时，设计者们做出了一些选择。

例如，为什么没有 `IntComparator`，`LongComparator` 和 `DoubleComparator` 呢？有 `BooleanSupplier` 却没有其他表示 **Boolean** 的接口；有通用的 `BiConsumer` 却没有用于 **int**，**long** 和 **double** 的 `BiConsumers` 变体（我对他们放弃的原因表示同情）。这些选择是疏忽还是有人认为其他组合的使用情况出现得很少（他们是如何得出这个结论的）？

你还可以看到基本类型给 Java 添加了多少复杂性。为了缓和效率问题，该语言的第一版中就包含了基本类型。现在，在语言的生命周期中，我们仍然受到语言设计选择不佳的影响。

下面枚举了基于 Lambda 表达式的所有不同 **Function** 变体的示例：

```
// functional/FunctionVariants.java

import java.util.function.*;

class Foo {}

class Bar {
```

```
  Foo f;
  Bar(Foo f) { this.f = f; }
}

class IBaz {
  int i;
  IBaz(int i) {
    this.i = i;
  }
}

class LBaz {
  long l;
  LBaz(long l) {
    this.l = l;
  }
}

class DBaz {
  double d;
  DBaz(double d) {
    this.d = d;
  }
}

public class FunctionVariants {
  static Function<Foo,Bar> f1 = f -> new Bar(f);
  static IntFunction<IBaz> f2 = i -> new IBaz(i);
  static LongFunction<LBaz> f3 = l -> new LBaz(l);
  static DoubleFunction<DBaz> f4 = d -> new DBaz(d);
  static ToIntFunction<IBaz> f5 = ib -> ib.i;
  static ToLongFunction<LBaz> f6 = lb -> lb.l;
  static ToDoubleFunction<DBaz> f7 = db -> db.d;
  static IntToLongFunction f8 = i -> i;
  static IntToDoubleFunction f9 = i -> i;
  static LongToIntFunction f10 = l -> (int)l;
  static LongToDoubleFunction f11 = l -> l;
  static DoubleToIntFunction f12 = d -> (int)d;
  static DoubleToLongFunction f13 = d -> (long)d;

  public static void main(String[] args) {
    Bar b = f1.apply(new Foo());
    IBaz ib = f2.apply(11);
    LBaz lb = f3.apply(11);
    DBaz db = f4.apply(11);
    int i = f5.applyAsInt(ib);
    long l = f6.applyAsLong(lb);
    double d = f7.applyAsDouble(db);
    l = f8.applyAsLong(12);
    d = f9.applyAsDouble(12);
    i = f10.applyAsInt(12);
```

```
    d = f11.applyAsDouble(12);
    i = f12.applyAsInt(13.0);
    l = f13.applyAsLong(13.0);
  }
}
```

这些 Lambda 表达式尝试生成适合函数签名的最简代码。在某些情况下，有必要进行强制类型转换，否则编译器会报截断错误。

主方法中的每个测试都显示了 `Function` 接口中不同类型的 `apply()` 方法。每个都产生一个与其关联的 Lambda 表达式的调用。

方法引用有自己的小魔法:

```
/ functional/MethodConversion.java

import java.util.function.*;

class In1 {}
class In2 {}

public class MethodConversion {
  static void accept(In1 i1, In2 i2) {
    System.out.println("accept()");
  }
  static void someOtherName(In1 i1, In2 i2) {
    System.out.println("someOtherName()");
  }
  public static void main(String[] args) {
    BiConsumer<In1,In2> bic;

    bic = MethodConversion::accept;
    bic.accept(new In1(), new In2());

    bic = MethodConversion::someOtherName;
    // bic.someOtherName(new In1(), new In2()); // Nope
    bic.accept(new In1(), new In2());
  }
}
```

输出结果:

```
accept()
someOtherName()
```

查看 `BiConsumer` 的文档, 你会看到 `accept()` 方法。实际上, 如果我们将方法命名为 `accept()`, 它就可以作为方法引用。但是我们也可用不同的名称，比如 `someOtherName()`。只要参数类型、返回类型与 `BiConsumer` 的 `accept()` 相同即可。

因此，在使用函数接口时，名称无关紧要——只要参数类型和返回类型相同。Java 会将你的方法

映射到接口方法。要调用方法，可以调用接口的函数式方法名（在本例中为 `accept()`），而不是你的方法名。

现在我们来看看所有基于类的函数式，应用于方法引用（即那些不涉及基本类型的函数）。下例我们创建了一个最简单的函数式签名。代码示例：

```
// functional/ClassFunctionals.java

import java.util.*;
import java.util.function.*;

class AA {}
class BB {}
class CC {}

public class ClassFunctionals {
  static AA f1() { return new AA(); }
  static int f2(AA aa1, AA aa2) { return 1; }
  static void f3(AA aa) {}
  static void f4(AA aa, BB bb) {}
  static CC f5(AA aa) { return new CC(); }
  static CC f6(AA aa, BB bb) { return new CC(); }
  static boolean f7(AA aa) { return true; }
  static boolean f8(AA aa, BB bb) { return true; }
  static AA f9(AA aa) { return new AA(); }
  static AA f10(AA aa1, AA aa2) { return new AA(); }
  public static void main(String[] args) {
    Supplier<AA> s = ClassFunctionals::f1;
    s.get();
    Comparator<AA> c = ClassFunctionals::f2;
    c.compare(new AA(), new AA());
    Consumer<AA> cons = ClassFunctionals::f3;
    cons.accept(new AA());
    BiConsumer<AA,BB> bicons = ClassFunctionals::f4;
    bicons.accept(new AA(), new BB());
    Function<AA,CC> f = ClassFunctionals::f5;
    CC cc = f.apply(new AA());
    BiFunction<AA,BB,CC> bif = ClassFunctionals::f6;
    cc = bif.apply(new AA(), new BB());
    Predicate<AA> p = ClassFunctionals::f7;
    boolean result = p.test(new AA());
    BiPredicate<AA,BB> bip = ClassFunctionals::f8;
    result = bip.test(new AA(), new BB());
    UnaryOperator<AA> uo = ClassFunctionals::f9;
    AA aa = uo.apply(new AA());
    BinaryOperator<AA> bo = ClassFunctionals::f10;
    aa = bo.apply(new AA(), new AA());
  }
}
```

请**注意**，每个方法名称都是随意的（如 `f1()`，`f2()`等）。正如你刚才看到的，一旦将方法引用赋值给函数接口，我们就可以调用与该接口关联的函数方法。在此示例中为 `get()`、`compare()`、`accept()`、`apply()` 和 `test()`。

14.4.1 多参数函数式接口

`java.util.functional` 中的接口是有限的。比如有了 `BiFunction`，但它不能变化。如果需要三参数函数的接口怎么办？其实这些接口非常简单，很容易查看 Java 库源代码并自行创建。代码示例：

```
// functional/TriFunction.java

@FunctionalInterface
public interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}
```

简单测试，验证它是否有效：

```
// functional/TriFunctionTest.java

public class TriFunctionTest {
  static int f(int i, long l, double d) { return 99; }
  public static void main(String[] args) {
    TriFunction<Integer, Long, Double, Integer> tf =
      TriFunctionTest::f;
    tf = (i, l, d) -> 12;
  }
}
```

这里我们测试了方法引用和 Lambda 表达式。

14.4.2 缺少基本类型的函数

让我们重温一下 `BiConsumer`，看看我们如何创建缺少 **int**，**long** 和 **double** 的各种排列：

```
// functional/BiConsumerPermutations.java

import java.util.function.*;

public class BiConsumerPermutations {
  static BiConsumer<Integer, Double> bicid = (i, d) ->
    System.out.format("%d, %f%n", i, d);
  static BiConsumer<Double, Integer> bicdi = (d, i) ->
    System.out.format("%d, %f%n", i, d);
  static BiConsumer<Integer, Long> bicil = (i, l) ->
    System.out.format("%d, %d%n", i, l);
```

```
  public static void main(String[] args) {
    bicid.accept(47, 11.34);
    bicdi.accept(22.45, 92);
    bicil.accept(1, 11L);
  }
}
```

输出结果:

```
47, 11.340000
92, 22.450000
1, 11
```

这里使用 `System.out.format()` 来显示。它类似于 `System.out.println()` 但提供了更多的显示选项。这里，`%f` 表示我将 `n` 作为浮点值给出，`%d` 表示 `n` 是一个整数值。这其中可以包含空格，输入 `%n` 会换行 — 当然使用传统的 `\n` 也能换行，但 `%n` 是自动跨平台的，这是使用 `format()` 的另一个原因。

上例简单使用了包装类型，装箱和拆箱用于在基本类型之间来回转换。我们也可以使用包装类型，如 `Function`，而不是预定义的基本类型。代码示例:

```
// functional/FunctionWithWrapped.java

import java.util.function.*;

public class FunctionWithWrapped {
  public static void main(String[] args) {
    Function<Integer, Double> fid = i -> (double)i;
    IntToDoubleFunction fid2 = i -> i;
  }
}
```

如果没有强制转换，则会收到错误消息:“Integer cannot be converted to Double”（**Integer** 无法转换为 **Double**），而使用 **IntToDoubleFunction** 就没有此类问题。**IntToDoubleFunction** 接口的源代码是这样的:

```
@FunctionalInterface
public interface IntToDoubleFunction {
  double applyAsDouble(int value);
}
```

之所以我们可以简单地编写 `Function <Integer, Double>` 并返回合适的结果，很明显是为了性能。使用基本类型可以防止传递参数和返回结果过程中的自动装箱和自动拆箱。

似乎是考虑到使用频率，某些函数类型并没有预定义。

当然，如果因缺少基本类型而造成的性能问题，你也可以轻松编写自己的接口（参考 Java 源代码）——尽管这里出现性能瓶颈的可能性不大。

14.5 高阶函数

这个名字可能听起来令人生畏，但是：高阶函数（Higher-order Function）只是一个消费或产生函数的函数。

我们先来看看如何产生一个函数：

```
// functional/ProduceFunction.java

import java.util.function.*;

interface
FuncSS extends Function<String, String> {} // [1]

public class ProduceFunction {
  static FuncSS produce() {
    return s -> s.toLowerCase(); // [2]
  }
  public static void main(String[] args) {
    FuncSS f = produce();
    System.out.println(f.apply("YELLING"));
  }
}
```

输出结果：

```
yelling
```

这里，`produce()` 是高阶函数。

[1] 使用继承，可以轻松地为专用接口创建别名。

[2] 使用 Lambda 表达式，可以轻松地在方法中创建和返回一个函数。

要消费一个函数，消费函数需要在参数列表正确地描述函数类型。代码示例：

```
// functional/ConsumeFunction.java

import java.util.function.*;

class One {}
class Two {}

public class ConsumeFunction {
  static Two consume(Function<One,Two> onetwo) {
    return onetwo.apply(new One());
  }
  public static void main(String[] args) {
    Two two = consume(one -> new Two());
  }
}
```

当基于消费函数生成新函数时，事情就变得相当有趣了。代码示例如下：

```
// functional/TransformFunction.java

import java.util.function.*;

class I {
  @Override
  public String toString() { return "I"; }
}

class O {
  @Override
  public String toString() { return "O"; }
}

public class TransformFunction {
  static Function<I,O> transform(Function<I,O> in) {
    return in.andThen(o -> {
      System.out.println(o);
      return o;
    });
  }
  public static void main(String[] args) {
    Function<I,O> f2 = transform(i -> {
      System.out.println(i);
      return new O();
    });
    O o = f2.apply(new I());
  }
}
```

输出结果：

```
I
O
```

在这里，`transform()` 生成一个与传入的函数具有相同签名的函数，但是你可以生成任何你想要的类型。

这里使用到了 `Function` 接口中名为 `andThen()` 的默认方法，该方法专门用于操作函数。顾名思义，在调用 `in` 函数之后调用 `toThen()`（还有个 `compose()` 方法，它在 `in` 函数之前应用新函数）。要附加一个 `andThen()` 函数，我们只需将该函数作为参数传递。`transform()` 产生的是一个新函数，它将 `in` 的动作与 `andThen()` 参数的动作结合起来。

14.6 闭包

在上一节的 `ProduceFunction.java` 中，我们从方法中返回 Lambda 函数。虽然过程简单，但是有些问题必须再回过头来探讨一下。

闭包（Closure）一词总结了这些问题。它非常重要，利用闭包可以轻松生成函数。

考虑一个更复杂的 Lambda，它使用函数作用域之外的变量。返回该函数会发生什么？也就是说，当你调用函数时，它对那些 “外部”变量引用了什么？如果语言不能自动解决这个问题，那将变得非常具有挑战性。能够解决这个问题的语言被称为**支持闭包**，或者叫作在词法上限定范围 (也使用术语*变量捕获*)。Java 8 提供了有限但合理的闭包支持，我们将用一些简单的例子来研究它。

首先，下例函数中，方法返回访问对象字段和方法参数。代码示例：

```
// functional/Closure1.java

import java.util.function.*;

public class Closure1 {
  int i;
  IntSupplier makeFun(int x) {
    return () -> x + i++;
  }
}
```

但是，仔细考虑一下，`i` 的这种用法并非是个大难题，因为对象很可能在你调用 `makeFun()` 之后就存在了——实际上，垃圾收集器几乎肯定会保留一个对象，并将现有的函数以这种方式绑定到该对象上[5]。当然，如果你对同一个对象多次调用 `makeFun()` ，你最终会得到多个函数，它们共享 `i` 的存储空间：

```
// functional/SharedStorage.java

import java.util.function.*;

public class SharedStorage {
  public static void main(String[] args) {
    Closure1 c1 = new Closure1();
    IntSupplier f1 = c1.makeFun(0);
    IntSupplier f2 = c1.makeFun(0);
    IntSupplier f3 = c1.makeFun(0);
    System.out.println(f1.getAsInt());
    System.out.println(f2.getAsInt());
    System.out.println(f3.getAsInt());
  }
}
```

输出结果：

[5]我还没有验证过这种说法。

```
0
1
2
```

每次调用 `getAsInt()` 都会增加 `i`，表明存储是共享的。

如果 `i` 是 `makeFun()` 的局部变量怎么办？在正常情况下，当 `makeFun()` 完成时 `i` 就消失。但它仍可以编译：

```java
// functional/Closure2.java

import java.util.function.*;

public class Closure2 {
  IntSupplier makeFun(int x) {
    int i = 0;
    return () -> x + i;
  }
}
```

由 `makeFun()` 返回的 `IntSupplier` "关闭" `i` 和 `x`，因此当你调用返回的函数时两者仍然有效。但请**注意**，我没有像 `Closure1.java` 那样递增 `i`，因为会产生编译时错误。代码示例：

```java
// functional/Closure3.java

// {WillNotCompile}
import java.util.function.*;

public class Closure3 {
  IntSupplier makeFun(int x) {
    int i = 0;
    // x++ 和 i++ 都会报错:
    return () -> x++ + i++;
  }
}
```

`x` 和 `i` 的操作都犯了同样的错误：从 Lambda 表达式引用的局部变量必须是 **`final`** 或者是等同 **`final`** 效果的。

如果使用 **`final`** 修饰 `x`和 `i`，就不能再递增它们的值了。代码示例：

```java
// functional/Closure4.java

import java.util.function.*;

public class Closure4 {
  IntSupplier makeFun(final int x) {
    final int i = 0;
    return () -> x + i;
  }
```

```
}
```

那么为什么在 `Closure2.java` 中，`x` 和 `i` 非 **`final`** 却可以运行呢?

这就叫做**等同 final 效果**（Effectively Final）。这个术语是在 Java 8 才开始出现的，表示虽然没有明确地声明变量是 **`final`** 的，但是因变量值没被改变过而实际有了 **`final`** 同等的效果。如果局部变量的初始值永远不会改变，那么它实际上就是 **`final`** 的。

如果 `x` 和 `i` 的值在方法中的其他位置发生改变（但不在返回的函数内部），则编译器仍将视其为错误。每个递增操作则会分别产生错误消息。代码示例:

```
/ functional/Closure5.java

// {无法编译成功}
import java.util.function.*;

public class Closure5 {
  IntSupplier makeFun(int x) {
    int i = 0;
    i++;
    x++;
    return () -> x + i;
  }
}
```

等同 final 效果意味着可以在变量声明前加上 **final** 关键字而不用更改任何其余代码。实际上它就是具备 **`final`** 效果的，只是没有明确说明。

通过在闭包中使用 **`final`** 关键字提前修饰变量 `x` 和 `i`，我们解决了 `Closure5.java` 中的问题。代码示例:

```

// functional/Closure6.java

import java.util.function.*;

public class Closure6 {
  IntSupplier makeFun(int x) {
    int i = 0;
    i++;
    x++;
    final int iFinal = i;
    final int xFinal = x;
    return () -> xFinal + iFinal;
  }
}
```

上例中 `iFinal` 和 `xFinal` 的值在赋值后并没有改变过，因此在这里使用 **`final`** 是多余的。

如果这里是引用的话，需要把 **int** 型更改为 **Integer** 型。代码示例:

```java
// functional/Closure7.java

// {无法编译成功}
import java.util.function.*;

public class Closure7 {
  IntSupplier makeFun(int x) {
    Integer i = 0;
    i = i + 1;
    return () -> x + i;
  }
}
```

编译器非常智能，它能识别变量 `i` 的值被更改过了。对于包装类型的处理可能比较特殊，因此我们尝试下 **List**:

```java
// functional/Closure8.java

import java.util.*;
import java.util.function.*;

public class Closure8 {
  Supplier<List<Integer>> makeFun() {
    final List<Integer> ai = new ArrayList<>();
    ai.add(1);
    return () -> ai;
  }
  public static void main(String[] args) {
    Closure8 c7 = new Closure8();
    List<Integer>
      l1 = c7.makeFun().get(),
      l2 = c7.makeFun().get();
    System.out.println(l1);
    System.out.println(l2);
    l1.add(42);
    l2.add(96);
    System.out.println(l1);
    System.out.println(l2);
  }
}
```

输出结果:

```
[1]
[1]
[1, 42]
[1, 96]
```

可以看到，这次一切正常。我们改变了 **List** 的值却没产生编译时错误。通过观察本例的输出结果，我们发现这看起来非常安全。这是因为每次调用 `makeFun()` 时，其实都会创建并返回一个全新的

ArrayList。也就是说，每个闭包都有自己独立的 ArrayList，它们之间互不干扰。

请**注意**我已经声明 ai 是 **final** 的了。尽管在这个例子中你可以去掉 **final** 并得到相同的结果（试试吧!）。应用于对象引用的 **final** 关键字仅表示不会重新赋值引用。它并不代表你不能修改对象本身。

下面我们来看看 Closure7.java 和 Closure8.java 之间的区别。我们看到：在 Closure7.java 中变量 i 有过重新赋值。也许这就是**等同 final 效果**错误消息的触发点。

```
// functional/Closure9.java

// {无法编译成功}
import java.util.*;
import java.util.function.*;

public class Closure9 {
  Supplier<List<Integer>> makeFun() {
    List<Integer> ai = new ArrayList<>();
    ai = new ArrayList<>(); // Reassignment
    return () -> ai;
  }
}
```

上例，重新赋值引用会触发错误消息。如果只修改指向的对象则没问题，只要没有其他人获得对该对象的引用（这意味着你有多个实体可以修改对象，此时事情会变得非常混乱），基本上就是安全的[6]。

让我们回顾一下 Closure1.java。那么现在问题来了：为什么变量 i 被修改编译器却没有报错呢。它既不是 **final** 的，也不是**等同 final 效果**的。因为 i 是外围类的成员，所以这样做肯定是安全的（除非你正在创建共享可变内存的多个函数）。是的，你可以辩称在这种情况下不会发生变量捕获（Variable Capture）。但可以肯定的是，Closure3.java 的错误消息是专门针对局部变量的。因此，规则并非只是“在 Lambda 之外定义的任何变量必须是 **final** 的或**等同 final 效果**那么简单。相反，你必须考虑捕获的变量是否是**等同 final 效果**的。如果它是对象中的字段，那么它拥有独立的生存周期，并且不需要任何特殊的捕获，以便稍后在调用 Lambda 时存在。

14.6.1 作为闭包的内部类

我们可以使用匿名内部类重写之前的例子：

```
// functional/AnonymousClosure.java

import java.util.function.*;

public class AnonymousClosure {
  IntSupplier makeFun(int x) {
    int i = 0;
```

[6]当你理解了并发编程章节的内容，你就能明白为什么更改共享变量“不是线程安全的”的了。

```
    // 同样规则的应用:
    // i++; // 非等同 final 效果
    // x++; // 同上
    return new IntSupplier() {
      public int getAsInt() { return x + i; }
    };
  }
}
```

实际上只要有内部类，就会有闭包（Java 8 只是简化了闭包操作）。在 Java 8 之前，变量 `x` 和 `i` 必须被明确声明为 **`final`**。在 Java 8 中，内部类的规则放宽，包括**等同 final 效果**。

14.7 函数组合

函数组合（Function Composition）意为“多个函数组合成新函数”。它通常是函数式编程的基本组成部分。在前面的 `TransformFunction.java` 类中，有一个使用 `andThen()` 的函数组合示例。一些 `java.util.function` 接口中包含支持函数组合的方法 [7]。

组合方法	支持接口
`andThen(argument)` 根据参数执行原始操作	**Function BiFunction Consumer BiConsumer IntConsumer LongConsumer DoubleConsumer UnaryOperator IntUnaryOperator LongUnaryOperator DoubleUnaryOperator BinaryOperator**
`compose(argument)` 根据参数执行原始操作	**Function UnaryOperator IntUnaryOperator LongUnaryOperator DoubleUnaryOperator**
`and(argument)` 短路**逻辑与**原始谓词和参数谓词	**Predicate BiPredicate IntPredicate LongPredicate DoublePredicate**
`or(argument)` 短路**逻辑或**原始谓词和参数谓词	**Predicate BiPredicate IntPredicate LongPredicate DoublePredicate**
`negate()` 该谓词的**逻辑否**谓词	**Predicate BiPredicate IntPredicate LongPredicate DoublePredicate**

[7] 接口能够支持方法的原因是它们是 Java 8 默认方法，你将在下一章中了解到。

下例使用了 Function 里的 compose()和 andThen()。代码示例:

```
// functional/FunctionComposition.java

import java.util.function.*;

public class FunctionComposition {
  static Function<String, String>
    f1 = s -> {
      System.out.println(s);
      return s.replace('A', '_');
    },
    f2 = s -> s.substring(3),
    f3 = s -> s.toLowerCase(),
    f4 = f1.compose(f2).andThen(f3);
  public static void main(String[] args) {
    System.out.println(
      f4.apply("GO AFTER ALL AMBULANCES"));
  }
}
```

输出结果:

```
AFTER ALL AMBULANCES
_fter _ll _mbul_nces
```

这里我们重点看正在创建的新函数 f4。它调用 apply() 的方式与常规几乎无异[8]。

当 f1 获得字符串时，它已经被f2 剥离了前三个字符。这是因为 compose（f2） 表示 f2 的调用发生在 f1 之前。

下例是 Predicate 的逻辑运算演示. 代码示例:

```
// functional/PredicateComposition.java

import java.util.function.*;
import java.util.stream.*;

public class PredicateComposition {
  static Predicate<String>
    p1 = s -> s.contains("bar"),
    p2 = s -> s.length() < 5,
    p3 = s -> s.contains("foo"),
    p4 = p1.negate().and(p2).or(p3);
  public static void main(String[] args) {
    Stream.of("bar", "foobar", "foobaz", "fongopuckey")
      .filter(p4)
      .forEach(System.out::println);
  }
}
```

[8]一些语言，如 Python，允许像调用其他函数一样调用组合函数。但这是 Java，所以我们做做可为之事。

输出结果：

```
foobar
foobaz
```

p4 获取到了所有谓词并组合成一个更复杂的谓词。解读：如果字符串中不包含 bar 且长度小于 5，或者它包含 foo ，则结果为 **true**。

正因它产生如此清晰的语法，我在主方法中采用了一些小技巧，并借用了下一章的内容。首先，我创建了一个字符串对象的流，然后将每个对象传递给 filter() 操作。filter() 使用 p4 的谓词来确定对象的去留。最后我们使用 forEach() 将 println 方法引用应用在每个留存的对象上。

从输出结果我们可以看到 p4 的工作流程：任何带有 foo 的东西都会留下，即使它的长度大于 5。fongopuckey 因长度超出和不包含 bar 而被丢弃。

14.8 柯里化和部分求值

柯里化（Currying）的名称来自于其发明者之一 *Haskell Curry*。他可能是计算机领域唯一名字被命名重要概念的人（另外就是 Haskell 编程语言）。柯里化意为：将一个多参数的函数，转换为一系列单参数函数。

```java
// functional/CurryingAndPartials.java

import java.util.function.*;

public class CurryingAndPartials {
   // 未柯里化:
   static String uncurried(String a, String b) {
      return a + b;
   }
   public static void main(String[] args) {
      // 柯里化的函数:
      Function<String, Function<String, String>> sum =
         a -> b -> a + b; // [1]

      System.out.println(uncurried("Hi ", "Ho"));

      Function<String, String>
        hi = sum.apply("Hi "); // [2]
      System.out.println(hi.apply("Ho"));

      // 部分应用:
      Function<String, String> sumHi =
        sum.apply("Hup ");
      System.out.println(sumHi.apply("Ho"));
      System.out.println(sumHi.apply("Hey"));
   }
```

```
}
```

输出结果:

```
Hi Ho
Hi Ho
Hup Ho
Hup Hey
```

[1] 这一连串的箭头很巧妙。注意，在函数接口声明中，第二个参数是另一个函数。

[2] 柯里化的目的是能够通过提供一个参数来创建一个新函数，所以现在有了一个“带参函数”和剩下的“无参函数”。实际上，你从一个双参数函数开始，最后得到一个单参数函数。

我们可以通过添加级别来柯里化一个三参数函数:

```java
// functional/Curry3Args.java

import java.util.function.*;

public class Curry3Args {
   public static void main(String[] args) {
      Function<String,
        Function<String,
          Function<String, String>>> sum =
            a -> b -> c -> a + b + c;
      Function<String,
        Function<String, String>> hi =
          sum.apply("Hi ");
      Function<String, String> ho =
        hi.apply("Ho ");
      System.out.println(ho.apply("Hup"));
   }
}
```

输出结果:

```
Hi Ho Hup
```

对于每个级别的箭头级联（Arrow-cascading），你在类型声明中包裹了另一个 **Function**。

处理基本类型和装箱时，请使用适当的 **Function** 接口:

```java
// functional/CurriedIntAdd.java

import java.util.function.*;

public class CurriedIntAdd {
  public static void main(String[] args) {
    IntFunction<IntUnaryOperator>
      curriedIntAdd = a -> b -> a + b;
    IntUnaryOperator add4 = curriedIntAdd.apply(4);
```

```
    System.out.println(add4.applyAsInt(5));
      }
}
```

输出结果:

```
9
```

可以在互联网上找到更多的柯里化示例。通常它们是用 Java 之外的语言实现的，但如果理解了柯里化的基本概念，你可以很轻松地用 Java 实现它们。

14.9 纯函数式编程

即使没有函数式支持，像 C 这样的基础语言，也可以按照一定的原则编写纯函数式程序。Java 8 让函数式编程更简单，不过我们要确保一切是 `final` 的，同时你的所有方法和函数没有副作用。因为 Java 在本质上并非是不可变语言，我们无法通过编译器查错。

这种情况下，我们可以借助第三方工具[9]，但使用 Scala 或 Clojure 这样的语言可能更简单。因为它们从一开始就是为保持不变性而设计的。你可以采用这些语言来编写你的 Java 项目的一部分。如果必须要用纯函数式编写，则可以用 Scala（需要一些规则）或 Clojure （需要的规则更少）。虽然 Java 支持并发编程，但如果这是你项目的核心部分，你应该考虑在项目部分功能中使用 `Scala` 或 `Clojure` 之类的语言。

14.10 本章小结

Lambda 表达式和方法引用并没有将 Java 转换成函数式语言，而是提供了对函数式编程的支持。这对 Java 来说是一个巨大的改进。因为这允许你编写更简洁明了，易于理解的代码。在下一章中，你会看到它们在流式编程中的应用。相信你会像我一样，喜欢上流式编程。

这些特性满足大部分 Java 程序员的需求。他们开始羡慕嫉妒 Clojure、Scala 这类新语言的功能，并试图阻止 Java 程序员流失到其他阵营（就算不能阻止，起码提供了更好的选择）。

但是，Lambdas 和方法引用远非完美，我们永远要为 Java 设计者早期的草率决定付出代价。特别是没有泛型 Lambda，所以 Lambda 在 Java 中并非一等公民。虽然我不否认 Java 8 的巨大改进，但这意味着和许多 Java 特性一样，它的使用还是会让人感觉沮丧和鸡肋。

当你遇到学习困难时，请记住通过 IDE（NetBeans、IntelliJ Idea 和 Eclipse）获得帮助，因为 IDE 可以智能提示你何时使用 Lambda 表达式或方法引用，甚至有时还能为你优化代码。

[9]例如，Immutables 和 Mutability Detector。

通过” 蓄意破坏” 测试你的测试

15 流式编程

> 集合优化了对象的存储，而流和对象的处理有关。

流是一系列与特定存储机制无关的元素——实际上，流并没有“存储”之说。

利用流，我们无需迭代集合中的元素，就可以提取和操作它们。这些管道通常被组合在一起，在流上形成一条操作管道。

在大多数情况下，将对象存储在集合中是为了处理他们，因此你将会发现你将把编程的主要焦点从集合转移到了流上。流的一个核心好处是，它使得程序更加短小并且更易理解。当 Lambda 表达式和方法引用（method references）和流一起使用的时候会让人感觉自成一体。流使得 Java 8 更具吸引力。

举个例子，假如你要随机展示 5 至 20 之间不重复的整数并进行排序。实际上，你的关注点首先是创建一个有序集合。围绕这个集合进行后续的操作。但是使用流式编程，你就可以简单陈述你想做什么：

```java
// streams/Randoms.java
import java.util.*;
public class Randoms {
    public static void main(String[] args) {
        new Random(47)
            .ints(5, 20)
            .distinct()
            .limit(7)
            .sorted()
            .forEach(System.out::println);
    }
}
```

输出结果：

```
6
10
13
16
17
18
19
```

首先，我们给 **Random** 对象一个种子（以便程序再次运行时产生相同的输出）。`ints()` 方法产生一个流并且 `ints()` 方法有多种方式的重载 — 两个参数限定了数值产生的边界。这将生成一个整

数流。我们可以使用中间流操作（intermediate stream operation）`distinct()` 来获取它们的非重复值，然后使用 `limit()` 方法获取前 7 个元素。接下来，我们使用 `sorted()` 方法排序。最终使用 `forEach()` 方法遍历输出，它根据传递给它的函数对每个流对象执行操作。在这里，我们传递了一个可以在控制台显示每个元素的方法引用。`System.out::println` 。

注意 `Randoms.java` 中没有声明任何变量。流可以在不使用赋值或可变数据的情况下对有状态的系统建模，这非常有用。

声明式编程（Declarative programming）是一种：声明要做什么，而非怎么做的编程风格。正如我们在函数式编程中所看到的。**注意**，命令式编程的形式更难以理解。代码示例：

```java
// streams/ImperativeRandoms.java
import java.util.*;
public class ImperativeRandoms {
    public static void main(String[] args) {
        Random rand = new Random(47);
        SortedSet<Integer> rints = new TreeSet<>();
        while(rints.size() < 7) {
            int r = rand.nextInt(20);
            if(r < 5) continue;
            rints.add(r);
        }
        System.out.println(rints);
    }
}
```

输出结果：

```
[7, 8, 9, 11, 13, 15, 18]
```

在 `Randoms.java` 中，我们无需定义任何变量，但在这里我们定义了 3 个变量：`rand`，`rints` 和 `r`。由于 `nextInt()` 方法没有下限的原因（其内置的下限永远为 0），这段代码实现起来更复杂。所以我们要生成额外的值来过滤小于 5 的结果。

注意，你必须要研究程序的真正意图，而在 `Randoms.java` 中，代码只是告诉了你它正在做什么。这种语义清晰性也是 Java 8 的流式编程更受推崇的重要原因。

在 `ImperativeRandoms.java` 中显式地编写迭代机制称为外部迭代。而在 `Randoms.java` 中，流式编程采用内部迭代，这是流式编程的核心特性之一。这种机制使得编写的代码可读性更强，也更能利用多核处理器的优势。通过放弃对迭代过程的控制，我们把控制权交给并行化机制。我们将在并发编程一章中学习这部分内容。

另一个重要方面，流是懒加载的。这代表着它只在绝对必要时才计算。你可以将流看作“延迟列表”。由于计算延迟，流使我们能够表示非常大（甚至无限）的序列，而不需要考虑内存问题。

15.1 流支持

Java 设计者面临着这样一个难题：现存的大量类库不仅为 Java 所用，同时也被应用在整个 Java 生态圈数百万行的代码中。如何将一个全新的流的概念融入到现有类库中呢？

比如在 **Random** 中添加更多的方法。只要不改变原有的方法，现有代码就不会受到干扰。

问题是，接口部分怎么改造呢？特别是涉及集合类接口的部分。如果你想把一个集合转换为流，直接向接口添加新方法会破坏所有老的接口实现类。

Java 8 采用的解决方案是：在接口中添加被 `default`（默认）修饰的方法。通过这种方案，设计者们可以将流式（*stream*）方法平滑地嵌入到现有类中。流方法预置的操作几乎已满足了我们平常所有的需求。流操作的类型有三种：创建流，修改流元素（中间操作，Intermediate Operations），消费流元素（终端操作，Terminal Operations）。最后一种类型通常意味着收集流元素（通常是到集合中）。

下面我们来看下每种类型的流操作。

15.2 流创建

你可以通过 `Stream.of()` 很容易地将一组元素转化成为流（`Bubble` 类在本章的后面定义）：

```
// streams/StreamOf.java
import java.util.stream.*;
public class StreamOf {
    public static void main(String[] args) {
        Stream.of(new Bubble(1), new Bubble(2), new Bubble(3))
            .forEach(System.out::println);
        Stream.of("It's ", "a ", "wonderful ", "day ", "for ", "pie!")
            .forEach(System.out::print);
        System.out.println();
        Stream.of(3.14159, 2.718, 1.618)
            .forEach(System.out::println);
    }
}
```

输出结果：

```
Bubble(1)
Bubble(2)
Bubble(3)
It's a wonderful day for pie!
3.14159
2.718
1.618
```

除此之外，每个集合都可以通过调用 `stream()` 方法来产生一个流。代码示例：

```
// streams/CollectionToStream.java
```

```java
import java.util.*;
import java.util.stream.*;
public class CollectionToStream {
    public static void main(String[] args) {
        List<Bubble> bubbles = Arrays.asList(new Bubble(1), new Bubble(2), new Bubble(3));
        System.out.println(bubbles.stream()
            .mapToInt(b -> b.i)
            .sum());

        Set<String> w = new HashSet<>(Arrays.asList("It's a wonderful day for pie!".split(" ")));
        w.stream()
         .map(x -> x + " ")
         .forEach(System.out::print);
        System.out.println();

        Map<String, Double> m = new HashMap<>();
        m.put("pi", 3.14159);
        m.put("e", 2.718);
        m.put("phi", 1.618);
        m.entrySet().stream()
                    .map(e -> e.getKey() + ": " + e.getValue())
                    .forEach(System.out::println);
    }
}
```

输出结果:

```
6
a pie! It's for wonderful day
phi: 1.618
e: 2.718
pi: 3.14159
```

在创建 `List<Bubble>` 对象之后，我们只需要简单地调用所有集合中都有的 `stream()`。中间操作 `map()` 会获取流中的所有元素，并且对流中元素应用操作从而产生新的元素，并将其传递到后续的流中。通常 `map()` 会获取对象并产生新的对象，但在这里产生了特殊的用于数值类型的流。例如，`mapToInt()` 方法将一个对象流（object stream）转换成为包含整型数字的 `IntStream`。同样，针对 `Float` 和 `Double` 也有类似名字的操作。

我们通过调用字符串的 `split()`（该方法会根据参数来拆分字符串）来获取元素用于定义变量 `w`。稍后你会知道 `split()` 参数可以是十分复杂，但在这里我们只是根据空格来分割字符串。

为了从 **Map** 集合中产生流数据，我们首先调用 `entrySet()` 产生一个对象流，每个对象都包含一个 `key` 键以及与其相关联的 `value` 值。然后分别调用 `getKey()` 和 `getValue()` 获取值。

15.2.1 随机数流

Random 类被一组生成流的方法增强了。代码示例:

```
// streams/RandomGenerators.java
import java.util.*;
import java.util.stream.*;
public class RandomGenerators {
    public static <T> void show(Stream<T> stream) {
        stream
        .limit(4)
        .forEach(System.out::println);
        System.out.println("++++++++");
    }

    public static void main(String[] args) {
        Random rand = new Random(47);
        show(rand.ints().boxed());
        show(rand.longs().boxed());
        show(rand.doubles().boxed());
        // 控制上限和下限:
        show(rand.ints(10, 20).boxed());
        show(rand.longs(50, 100).boxed());
        show(rand.doubles(20, 30).boxed());
        // 控制流大小:
        show(rand.ints(2).boxed());
        show(rand.longs(2).boxed());
        show(rand.doubles(2).boxed());
        // 控制流的大小和界限
        show(rand.ints(3, 3, 9).boxed());
        show(rand.longs(3, 12, 22).boxed());
        show(rand.doubles(3, 11.5, 12.3).boxed());
    }
}
```

输出结果:

```
-1172028779
1717241110
-2014573909
229403722
++++++++
2955289354441303771
3476817843704654257
-8917117694134521474
4941259272818818752
++++++++
0.2613610344283964
0.0508673570556899
0.8037155449603999
0.7620665811558285
```

```
++++++++
16
10
11
12
++++++++
65
99
54
58
++++++++
29.86777681078574
24.83968447804611
20.09247112332014
24.046793846338723
++++++++
1169976606
1947946283
++++++++
2970202997824602425
-2325326920272830366
++++++++
0.7024254510631527
0.6648552384607359
++++++++
6
7
7
++++++++
17
12
20
++++++++
12.27872414236691
11.732085449736195
12.196509449817267
++++++++
```

为了消除冗余代码，我创建了一个泛型方法 `show(Stream<T> stream)`（在讲解泛型之前就使用这个特性，确实有点作弊，但是回报是值得的）。类型参数 `T` 可以是任何类型，所以这个方法对 **Integer**、**Long** 和 **Double** 类型都生效。但是 **Random** 类只能生成基本类型 **int**，**long**，**double** 的流。幸运的是，`boxed()` 流操作将会自动地把基本类型包装成为对应的装箱类型，从而使得 `show()` 能够接受流。

我们可以使用 **Random** 为任意对象集合创建 **Supplier**。如下是一个文本文件提供字符串对象的例子。

Cheese.dat 文件内容：

```
// streams/Cheese.dat
```

```
Not much of a cheese shop really, is it?
Finest in the district, sir.
And what leads you to that conclusion?
Well, it's so clean.
It's certainly uncontaminated by cheese.
```

我们通过 **File** 类将 Cheese.dat 文件的所有行读取到 `List<String>` 中。代码示例:

```java
// streams/RandomWords.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import java.io.*;
import java.nio.file.*;
public class RandomWords implements Supplier<String> {
    List<String> words = new ArrayList<>();
    Random rand = new Random(47);
    RandomWords(String fname) throws IOException {
        List<String> lines = Files.readAllLines(Paths.get(fname));
        // 略过第一行
        for (String line : lines.subList(1, lines.size())) {
            for (String word : line.split("[ .?,]+"))
                words.add(word.toLowerCase());
        }
    }
    public String get() {
        return words.get(rand.nextInt(words.size()));
    }
    @Override
    public String toString() {
        return words.stream()
            .collect(Collectors.joining(" "));
    }
    public static void main(String[] args) throws Exception {
        System.out.println(
            Stream.generate(new RandomWords("Cheese.dat"))
                .limit(10)
                .collect(Collectors.joining(" ")));
    }
}
```

输出结果:

```
it shop sir the much cheese by conclusion district is
```

在这里你可以看到更为复杂的 `split()` 运用。在构造器中，每一行都被 `split()` 通过空格或者被方括号包裹的任意标点符号进行分割。在结束方括号后面的 + 代表 + 前面的东西可以出现一次或者多次。

我们注意到在构造函数中循环体使用命令式编程（外部迭代）。在以后的例子中，你甚至会看到我

们如何消除这一点。这种旧的形式虽不是特别糟糕，但使用流会让人感觉更好。

在 toString() 和主方法中你看到了 collect() 收集操作，它根据参数来组合所有流中的元素。

当你使用 **Collectors.**joining()，你将会得到一个 String 类型的结果，每个元素都根据 joining() 的参数来进行分割。还有许多不同的 Collectors 用于产生不同的结果。

在主方法中，我们提前看到了 **Stream.**generate() 的用法，它可以把任意 Supplier<T> 用于生成 T 类型的流。

15.2.2 int 类型的范围

IntStream 类提供了 range() 方法用于生成整型序列的流。编写循环时，这个方法会更加便利：

```
// streams/Ranges.java
import static java.util.stream.IntStream.*;
public class Ranges {
    public static void main(String[] args) {
        // 传统方法:
        int result = 0;
        for (int i = 10; i < 20; i++)
            result += i;
        System.out.println(result);
        // for-in 循环:
        result = 0;
        for (int i : range(10, 20).toArray())
            result += i;
        System.out.println(result);
        // 使用流:
        System.out.println(range(10, 20).sum());
    }
}
```

输出结果：

```
145
145
145
```

在主方法中的第一种方式是我们传统编写 **for** 循环的方式；第二种方式，我们使用 range() 创建了流并将其转化为数组，然后在 **for**-in 代码块中使用。但是，如果你能像第三种方法那样全程使用流是更好的。我们对范围中的数字进行求和。在流中可以很方便的使用 sum() 操作求和。

注意 **IntStream.**range() 相比 onjava.Range.range() 拥有更多的限制。这是由于其可选的第三个参数，后者允许步长大于 1，并且可以从大到小来生成。

实用小功能 repeat() 可以用来替换简单的 **for** 循环。代码示例：

```java
// onjava/Repeat.java
package onjava;
import static java.util.stream.IntStream.*;
public class Repeat {
    public static void repeat(int n, Runnable action) {
        range(0, n).forEach(i -> action.run());
    }
}
```

其产生的循环更加清晰:

```java
// streams/Looping.java
import static onjava.Repeat.*;
public class Looping {
    static void hi() {
        System.out.println("Hi!");
    }
    public static void main(String[] args) {
        repeat(3, () -> System.out.println("Looping!"));
        repeat(2, Looping::hi);
    }
}
```

输出结果:

```
Looping!
Looping!
Looping!
Hi!
Hi!
```

原则上，在代码中包含并解释 `repeat()` 并不值得。诚然它是一个相当透明的工具，但结果取决于你的团队和公司的运作方式。

15.2.3 generate()

参照 `RandomWords.java` 中 **Stream.**`generate()` 搭配 `Supplier<T>` 使用的例子。代码示例:

```java
// streams/Generator.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Generator implements Supplier<String> {
    Random rand = new Random(47);
    char[] letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();

```

```
    public String get() {
        return "" + letters[rand.nextInt(letters.length)];
    }

    public static void main(String[] args) {
        String word = Stream.generate(new Generator())
                            .limit(30)
                            .collect(Collectors.joining());
        System.out.println(word);
    }
}
```

输出结果:

```
YNZBRNYGCFOWZNTCQRGSEGZMMJMROE
```

使用 `Random.nextInt()` 方法来挑选字母表中的大写字母。`Random.nextInt()` 的参数代表可以接受的最大的随机数范围，所以使用数组边界是经过深思熟虑的。

如果要创建包含相同对象的流，只需要传递一个生成那些对象的 `lambda` 到 `generate()` 中:

```
// streams/Duplicator.java
import java.util.stream.*;
public class Duplicator {
    public static void main(String[] args) {
        Stream.generate(() -> "duplicate")
              .limit(3)
              .forEach(System.out::println);
    }
}
```

输出结果:

```
duplicate
duplicate
duplicate
```

如下是在本章之前例子中使用过的 `Bubble` 类。**注意**它包含了自己的静态生成器(Static generator)方法。

```
// streams/Bubble.java
import java.util.function.*;
public class Bubble {
    public final int i;

    public Bubble(int n) {
        i = n;
    }

    @Override
    public String toString() {
```

```
        return "Bubble(" + i + ")";
    }

    private static int count = 0;
    public static Bubble bubbler() {
        return new Bubble(count++);
    }
}
```

由于 `bubbler()` 与 `Supplier<Bubble>` 是接口兼容的，我们可以将其方法引用直接传递给 **Stream.**`generate()`：

```
// streams/Bubbles.java
import java.util.stream.*;
public class Bubbles {
    public static void main(String[] args) {
        Stream.generate(Bubble::bubbler)
              .limit(5)
              .forEach(System.out::println);
    }
}
```

输出结果：

```
Bubble(0)
Bubble(1)
Bubble(2)
Bubble(3)
Bubble(4)
```

这是创建单独工厂类（Separate Factory class）的另一种方式。在很多方面它更加整洁，但是这是一个对于代码组织和品味的问题——你总是可以创建一个完全不同的工厂类。

15.2.4 iterate()

Stream.`iterate()` 以种子（第一个参数）开头，并将其传给方法（第二个参数）。方法的结果将添加到流，并存储作为第一个参数用于下次调用 `iterate()`，依次类推。我们可以利用 `iterate()` 生成一个斐波那契数列。代码示例：

```
// streams/Fibonacci.java
import java.util.stream.*;
public class Fibonacci {
    int x = 1;

    Stream<Integer> numbers() {
        return Stream.iterate(0, i -> {
            int result = x + i;
            x = i;
```

```
            return result;
        });
    }

    public static void main(String[] args) {
        new Fibonacci().numbers()
                       .skip(20) // 过滤前 20 个
                       .limit(10) // 然后取 10 个
                       .forEach(System.out::println);
    }
}
```

输出结果:

```
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
```

斐波那契数列将数列中最后两个元素进行求和以产生下一个元素。`iterate()` 只能记忆结果，因此我们需要利用一个变量 `x` 追踪另外一个元素。

在主方法中，我们使用了一个之前没有见过的 `skip()` 操作。它根据参数丢弃指定数量的流元素。在这里，我们丢弃了前 20 个元素。

15.2.5 流的建造者模式

在建造者设计模式（也称构造器模式）中，首先创建一个 `builder` 对象，传递给它多个构造器信息，最后执行“构造”。**Stream** 库提供了这样的 `Builder`。在这里，我们重新审视文件读取并将其转换成为单词流的过程。代码示例:

```
// streams/FileToWordsBuilder.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class FileToWordsBuilder {
    Stream.Builder<String> builder = Stream.builder();

    public FileToWordsBuilder(String filePath) throws Exception {
        Files.lines(Paths.get(filePath))
             .skip(1) // 略过开头的注释行
```

```
                .forEach(line -> {
                    for (String w : line.split("[ .?,]+"))
                        builder.add(w);
                });
    }

    Stream<String> stream() {
        return builder.build();
    }

    public static void main(String[] args) throws Exception {
        new FileToWordsBuilder("Cheese.dat")
            .stream()
            .limit(7)
            .map(w -> w + " ")
            .forEach(System.out::print);
    }
}
```

输出结果:

```
Not much of a cheese shop really
```

注意，构造器会添加文件中的所有单词（除了第一行，它是包含文件路径信息的注释），但是其并没有调用 `build()`。只要你不调用 `stream()` 方法，就可以继续向 `builder` 对象中添加单词。

在该类的更完整形式中，你可以添加一个标志位用于查看 `build()` 是否被调用，并且可能的话增加一个可以添加更多单词的方法。在 `Stream.Builder` 调用 `build()` 方法后继续尝试添加单词会产生一个异常。

15.2.6 Arrays

`Arrays` 类中含有一个名为 `stream()` 的静态方法用于把数组转换成为流。我们可以重写 `interfaces/Machine.java` 中的主方法用于创建一个流，并将 `execute()` 应用于每一个元素。代码示例:

```
// streams/Machine2.java
import java.util.*;
import onjava.Operations;
public class Machine2 {
    public static void main(String[] args) {
        Arrays.stream(new Operations[] {
            () -> Operations.show("Bing"),
            () -> Operations.show("Crack"),
            () -> Operations.show("Twist"),
            () -> Operations.show("Pop")
        }).forEach(Operations::execute);
    }
```

```
}
```

输出结果:

```
Bing
Crack
Twist
Pop
```

`new Operations[]` 表达式动态创建了 `Operations` 对象的数组。

`stream()` 同样可以产生 **IntStream**，**LongStream** 和 **DoubleStream**。

```
// streams/ArrayStreams.java
import java.util.*;
import java.util.stream.*;

public class ArrayStreams {
    public static void main(String[] args) {
        Arrays.stream(new double[] { 3.14159, 2.718, 1.618 })
            .forEach(n -> System.out.format("%f ", n));
        System.out.println();

        Arrays.stream(new int[] { 1, 3, 5 })
            .forEach(n -> System.out.format("%d ", n));
        System.out.println();

        Arrays.stream(new long[] { 11, 22, 44, 66 })
            .forEach(n -> System.out.format("%d ", n));
        System.out.println();

        // 选择一个子域:
        Arrays.stream(new int[] { 1, 3, 5, 7, 15, 28, 37 }, 3, 6)
            .forEach(n -> System.out.format("%d ", n));
    }
}
```

输出结果:

```
3.141590 2.718000 1.618000
1 3 5
11 22 44 66
7 15 28
```

最后一次 `stream()` 的调用有两个额外的参数。第一个参数告诉 `stream()` 从数组的哪个位置开始选择元素，第二个参数用于告知在哪里停止。每种不同类型的 `stream()` 都有类似的操作。

”Even though it doesn’t work, how does it feel? How is that possible?”

15.2.7 正则表达式

Java 的正则表达式将在字符串这一章节详细介绍。Java 8 在 `java.util.regex.Pattern` 中增加了一个新的方法 `splitAsStream()`。这个方法可以根据传入的公式将字符序列转化为流。但是有一个限制，输入只能是 **CharSequence**，因此不能将流作为 `splitAsStream()` 的参数。

我们再一次查看将文件处理为单词流的过程。这一次，我们使用流将文件分割为单独的字符串，接着使用正则表达式将字符串转化为单词流。

```
// streams/FileToWordsRegexp.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;
import java.util.regex.Pattern;
public class FileToWordsRegexp {
    private String all;
    public FileToWordsRegexp(String filePath) throws Exception {
        all = Files.lines(Paths.get(filePath))
        .skip(1) // First (comment) line
        .collect(Collectors.joining(" "));
    }
    public Stream<String> stream() {
        return Pattern
        .compile("[ .,?]+").splitAsStream(all);
    }
    public static void
    main(String[] args) throws Exception {
        FileToWordsRegexp fw = new FileToWordsRegexp("Cheese.dat");
        fw.stream()
          .limit(7)
          .map(w -> w + " ")
          .forEach(System.out::print);
        fw.stream()
          .skip(7)
          .limit(2)
          .map(w -> w + " ")
          .forEach(System.out::print);
    }
}
```

输出结果：

```
Not much of a cheese shop really is it
```

在构造器中我们读取了文件中的所有内容（跳过第一行注释，并将其转化成为单行字符串）。现在，当你调用 `stream()` 的时候，可以像往常一样获取一个流，但这次你可以多次调用 `stream()` 在已存储的字符串中创建一个新的流。这里有个限制，整个文件必须存储在内存中；在大多数情况下这并不是什么问题，但是这损失了流操作非常重要的优势：

1. 流“不需要存储”。当然它们需要一些内部存储，但是这只是序列的一小部分，和持有整个序列并不相同。
2. 它们是懒加载计算的。

幸运的是，我们稍后就会知道如何解决这个问题。

15.3 中间操作

中间操作用于从一个流中获取对象，并将对象作为另一个流从后端输出，以连接到其他操作。

15.3.1 跟踪和调试

`peek()` 操作的目的是帮助调试。它允许你无修改地查看流中的元素。代码示例：

```
// streams/Peeking.java
class Peeking {
    public static void main(String[] args) throws Exception {
        FileToWords.stream("Cheese.dat")
        .skip(21)
        .limit(4)
        .map(w -> w + " ")
        .peek(System.out::print)
        .map(String::toUpperCase)
        .peek(System.out::print)
        .map(String::toLowerCase)
        .forEach(System.out::print);
    }
}
```

输出结果：

```
Well WELL well it IT it s S s so SO so
```

`FileToWords` 稍后定义，但它的功能实现貌似和之前我们看到的差不多：产生字符串对象的流。之后在其通过管道时调用 `peek()` 进行处理。

因为 `peek()` 符合无返回值的 **Consumer** 函数式接口，所以我们只能观察，无法使用不同的元素来替换流中的对象。

15.3.2 流元素排序

在 `Randoms.java` 中，我们熟识了 `sorted()` 的默认比较器实现。其实它还有另一种形式的实现：传入一个 **Comparator** 参数。代码示例：

```
// streams/SortedComparator.java
```

I haven’t touched that module in weeks!

```
import java.util.*;
public class SortedComparator {
    public static void main(String[] args) throws Exception {
        FileToWords.stream("Cheese.dat")
        .skip(10)
        .limit(10)
        .sorted(Comparator.reverseOrder())
        .map(w -> w + " ")
        .forEach(System.out::print);
    }
}
```

输出结果:

```
you what to the that sir leads in district And
```

`sorted()` 预设了一些默认的比较器。这里我们使用的是反转“自然排序”。当然你也可以把 Lambda 函数作为参数传递给 `sorted()`。

15.3.3 移除元素

- `distinct()`: 在 `Randoms.java` 类中的 `distinct()` 可用于消除流中的重复元素。相比创建一个 **Set** 集合，该方法的工作量要少得多。
- `filter(Predicate)`: 过滤操作会保留与传递进去的过滤器函数计算结果为 **`true`** 元素。

在下例中，`isPrime()` 作为过滤器函数，用于检测质数。

```
// streams/Prime.java
import java.util.stream.*;
import static java.util.stream.LongStream.*;
public class Prime {
    public static Boolean isPrime(long n) {
        return rangeClosed(2, (long)Math.sqrt(n))
        .noneMatch(i -> n % i == 0);
    }
    public LongStream numbers() {
        return iterate(2, i -> i + 1)
        .filter(Prime::isPrime);
    }
    public static void main(String[] args) {
        new Prime().numbers()
        .limit(10)
        .forEach(n -> System.out.format("%d ", n));
        System.out.println();
        new Prime().numbers()
        .skip(90)
        .limit(10)
        .forEach(n -> System.out.format("%d ", n));
    }
```

```
}
```

输出结果:

```
2 3 5 7 11 13 17 19 23 29
467 479 487 491 499 503 509 521 523 541
```

`rangeClosed()` 包含了上限值。如果不能整除，即余数不等于 0，则 `noneMatch()` 操作返回 **`true`**，如果出现任何等于 0 的结果则返回 **`false`**。`noneMatch()` 操作一旦有失败就会退出。

15.3.4 应用函数到元素

- `map(Function)`：将函数操作应用在输入流的元素中，并将返回值传递到输出流中。
- `mapToInt(ToIntFunction)`：操作同上，但结果是 **IntStream**。
- `mapToLong(ToLongFunction)`：操作同上，但结果是 **LongStream**。
- `mapToDouble(ToDoubleFunction)`：操作同上，但结果是 **DoubleStream**。

在这里，我们使用 `map()` 映射多种函数到一个字符串流中。代码示例:

```java
// streams/FunctionMap.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
class FunctionMap {
    static String[] elements = { "12", "", "23", "45" };
    static Stream<String>
    testStream() {
        return Arrays.stream(elements);
    }
    static void test(String descr, Function<String, String> func) {
        System.out.println(" ---( " + descr + " )---");
        testStream()
        .map(func)
        .forEach(System.out::println);
    }
    public static void main(String[] args) {
        test("add brackets", s -> "[" + s + "]");
        test("Increment", s -> {
            try {
                return Integer.parseInt(s) + 1 + "";
            }
            catch(NumberFormatException e) {
                return s;
            }
        }
        );
        test("Replace", s -> s.replace("2", "9"));
        test("Take last digit", s -> s.length() > 0 ?
```

```
        s.charAt(s.length() - 1) + "" : s);
    }
}
```

输出结果:

```
---( add brackets )---
[12]
[]
[23]
[45]
---( Increment )---
13
24
46
---( Replace )---
19
93
45
---( Take last digit )---
2
3
5
```

在上面的自增示例中，我们使用 `Integer.parseInt()` 尝试将一个字符串转化为整数。如果字符串不能转化成为整数就会抛出 **NumberFormatException** 异常，我们只须回过头来将原始字符串放回到输出流中。

在以上例子中，`map()` 将一个字符串映射为另一个字符串，但是我们完全可以产生和接收类型完全不同的类型，从而改变流的数据类型。下面代码示例:

```
// streams/FunctionMap2.java
// Different input and output types （不同的输入输出类型）
import java.util.*;
import java.util.stream.*;
class Numbered {
    final int n;
    Numbered(int n) {
        this.n = n;
    }
    @Override
    public String toString() {
        return "Numbered(" + n + ")";
    }
}
class FunctionMap2 {
    public static void main(String[] args) {
        Stream.of(1, 5, 7, 9, 11, 13)
        .map(Numbered::new)
        .forEach(System.out::println);
```

```
    }
}
```

输出结果:

```
Numbered(1)
Numbered(5)
Numbered(7)
Numbered(9)
Numbered(11)
Numbered(13)
```

我们将获取到的整数通过构造器 `Numbered::new` 转化成为 `Numbered` 类型。

如果使用 **Function** 返回的结果是数值类型的一种，我们必须使用合适的 `mapTo数值类型` 进行替代。代码示例:

```
// streams/FunctionMap3.java
// Producing numeric output streams ( 产生数值输出流)
import java.util.*;
import java.util.stream.*;
class FunctionMap3 {
    public static void main(String[] args) {
        Stream.of("5", "7", "9")
        .mapToInt(Integer::parseInt)
        .forEach(n -> System.out.format("%d ", n));
        System.out.println();
        Stream.of("17", "19", "23")
        .mapToLong(Long::parseLong)
        .forEach(n -> System.out.format("%d ", n));
        System.out.println();
        Stream.of("17", "1.9", ".23")
        .mapToDouble(Double::parseDouble)
        .forEach(n -> System.out.format("%f ", n));
    }
}
```

输出结果:

```
5 7 9
17 19 23
17.000000 1.900000 0.230000
```

遗憾的是，Java 设计者并没有尽最大努力去消除基本类型。

15.3.5 在 map() 中组合流

假设我们现在有了一个传入的元素流，并且打算对流元素使用 `map()` 函数。现在你已经找到了一些可爱并独一无二的函数功能，但是问题来了：这个函数功能是产生一个流。我们想要产生一个元素流，

It works, but it hasn’t been tested.

而实际却产生了一个元素流的流。

flatMap() 做了两件事：将产生流的函数应用在每个元素上（与 map() 所做的相同），然后将每个流都扁平化为元素，因而最终产生的仅仅是元素。

flatMap(Function)：当 Function 产生流时使用。

flatMapToInt(Function)：当 Function 产生 IntStream 时使用。

flatMapToLong(Function)：当 Function 产生 LongStream 时使用。

flatMapToDouble(Function)：当 Function 产生 DoubleStream 时使用。

为了弄清它的工作原理，我们从传入一个刻意设计的函数给 map() 开始。该函数接受一个整数并产生一个字符串流：

```
// streams/StreamOfStreams.java
import java.util.stream.*;
public class StreamOfStreams {
    public static void main(String[] args) {
        Stream.of(1, 2, 3)
        .map(i -> Stream.of("Gonzo", "Kermit", "Beaker"))
        .map(e-> e.getClass().getName())
        .forEach(System.out::println);
    }
}
```

输出结果：

```
java.util.stream.ReferencePipeline$Head
java.util.stream.ReferencePipeline$Head
java.util.stream.ReferencePipeline$Head
```

我们天真地希望能够得到字符串流，但实际得到的却是“Head”流的流。我们可以使用 flatMap() 解决这个问题：

```
// streams/FlatMap.java
import java.util.stream.*;
public class FlatMap {
    public static void main(String[] args) {
        Stream.of(1, 2, 3)
        .flatMap(i -> Stream.of("Gonzo", "Fozzie", "Beaker"))
        .forEach(System.out::println);
    }
}
```

输出结果：

```
Gonzo
Fozzie
Beaker
Gonzo
Fozzie
```

It’s just some unlucky coincidence.

```
Beaker
Gonzo
Fozzie
Beaker
```

从映射返回的每个流都会自动扁平为组成它的字符串。

下面是另一个演示，我们从一个整数流开始，然后使用每一个整数去创建更多的随机数。

```java
// streams/StreamOfRandoms.java
import java.util.*;
import java.util.stream.*;
public class StreamOfRandoms {
    static Random rand = new Random(47);
    public static void main(String[] args) {
        Stream.of(1, 2, 3, 4, 5)
            .flatMapToInt(i -> IntStream.concat(
        rand.ints(0, 100).limit(i), IntStream.of(-1)))
            .forEach(n -> System.out.format("%d ", n));
    }
}
```

输出结果:

```
58 -1 55 93 -1 61 61 29 -1 68 0 22 7 -1 88 28 51 89 9 -1
```

在这里我们引入了 `concat()`，它以参数顺序组合两个流。如此，我们在每个随机 `Integer` 流的末尾添加一个 -1 作为标记。你可以看到最终流确实是从一组扁平流中创建的。

因为 `rand.ints()` 产生的是一个 `IntStream`，所以我必须使用 `flatMap()`、`concat()` 和 `of()` 的特定整数形式。

让我们再看一下将文件划分为单词流的任务。我们最后使用到的是 **FileToWordsRegexp.java**，它的问题是需要将整个文件读入行列表中 —— 显然需要存储该列表。而我们真正想要的是创建一个不需要中间存储层的单词流。

下面，我们再使用 `flatMap()` 来解决这个问题:

```java
// streams/FileToWords.java
import java.nio.file.*;
import java.util.stream.*;
import java.util.regex.Pattern;
public class FileToWords {
    public static Stream<String> stream(String filePath) throws
        Exception {
        return Files.lines(Paths.get(filePath))
        .skip(1) // First (comment) line
        .flatMap(line ->
        Pattern.compile("\\W+").splitAsStream(line));
    }
}
```

It’s never done that before.

`stream()` 现在是一个静态方法，因为它可以自己完成整个流创建过程。

注意：`\\W+` 是一个正则表达式。他表示“非单词字符”，`+` 表示“可以出现一次或者多次”。小写形式的 `\\w` 表示“单词字符”。

我们之前遇到的问题是 `Pattern.compile().splitAsStream()` 产生的结果为流，这意味着当我们只是想要一个简单的单词流时，在传入的行流（stream of lines）上调用 `map()` 会产生一个单词流的流。幸运的是，`flatMap()` 可以将元素流的流扁平化为一个简单的元素流。或者，我们可以使用 `String.split()` 生成一个数组，其可以被 `Arrays.stream()` 转化成为流：

```
.flatMap(line -> Arrays.stream(line.split("\\W+"))))
```

有了真正的、而非 `FileToWordsRegexp.java` 中基于集合存储的流，我们每次使用都必须从头创建，因为流并不能被复用：

```
// streams/FileToWordsTest.java
import java.util.stream.*;
public class FileToWordsTest {
    public static void main(String[] args) throws Exception {
        FileToWords.stream("Cheese.dat")
        .limit(7)
        .forEach(s -> System.out.format("%s ", s));
        System.out.println();
        FileToWords.stream("Cheese.dat")
        .skip(7)
        .limit(2)
        .forEach(s -> System.out.format("%s ", s));
    }
}
```

输出结果：

```
Not much of a cheese shop really
```

在 `System.out.format()` 中的 `%s` 表明参数为 **String** 类型。

15.4 Optional 类

在我们学习终端操作之前，我们必须考虑如果你在一个空流中获取元素会发生什么。我们喜欢为了“happy path”而将流连接起来，并假设流不会被中断。在流中放置 **`null`** 是很好的中断方法。那么是否有某种对象，可作为流元素的持有者，即使查看的元素不存在也能友好地提示我们（也就是说，不会发生异常）？

Optional 可以实现这样的功能。一些标准流操作返回 **Optional** 对象，因为它们并不能保证预期结果一定存在。包括：

- `findFirst()` 返回一个包含第一个元素的 **Optional** 对象,如果流为空则返回 **Optional.empty**

Somebody must have changed my code.

- `findAny()` 返回包含任意元素的 **Optional** 对象，如果流为空则返回 **Optional.empty**
- `max()` 和 `min()` 返回一个包含最大值或者最小值的 **Optional** 对象，如果流为空则返回 **Optional.empty**

`reduce()` 不再以 `identity` 形式开头，而是将其返回值包装在 **Optional** 中。(`identity` 对象成为其他形式的 `reduce()` 的默认结果，因此不存在空结果的风险)

对于数字流 **IntStream**、**LongStream** 和 **DoubleStream**，`average()` 会将结果包装在 **Optional** 以防止流为空。

以下是对空流进行所有这些操作的简单测试:

```
// streams/OptionalsFromEmptyStreams.java
import java.util.*;
import java.util.stream.*;
class OptionalsFromEmptyStreams {
    public static void main(String[] args) {
        System.out.println(Stream.<String>empty()
             .findFirst());
        System.out.println(Stream.<String>empty()
             .findAny());
        System.out.println(Stream.<String>empty()
             .max(String.CASE_INSENSITIVE_ORDER));
        System.out.println(Stream.<String>empty()
             .min(String.CASE_INSENSITIVE_ORDER));
        System.out.println(Stream.<String>empty()
             .reduce((s1, s2) -> s1 + s2));
        System.out.println(IntStream.empty()
             .average());
    }
}
```

输出结果:

```
Optional.empty
Optional.empty
Optional.empty
Optional.empty
Optional.empty
OptionalDouble.empty
```

当流为空的时候你会获得一个 **Optional.empty** 对象,而不是抛出异常。**Optional** 拥有 `toString()` 方法可以用于展示有用信息。

注意，空流是通过 `Stream.<String>empty()` 创建的。如果你在没有任何上下文环境的情况下调用 `Stream.empty()`，Java 并不知道它的数据类型；这个语法解决了这个问题。如果编译器拥有了足够的上下文信息，比如:

```
Stream<String> s = Stream.empty();
```

就可以在调用 `empty()` 时推断类型。

这个示例展示了 **Optional** 的两个基本用法:

```
// streams/OptionalBasics.java
import java.util.*;
import java.util.stream.*;
class OptionalBasics {
    static void test(Optional<String> optString) {
        if(optString.isPresent())
            System.out.println(optString.get());
        else
            System.out.println("Nothing inside!");
    }
    public static void main(String[] args) {
        test(Stream.of("Epithets").findFirst());
        test(Stream.<String>empty().findFirst());
    }
}
```

输出结果:

```
Epithets
Nothing inside!
```

当你接收到 **Optional** 对象时，应首先调用 `isPresent()` 检查其中是否包含元素。如果存在，可使用 `get()` 获取。

15.4.1 便利函数

有许多便利函数可以解包 **Optional** ，这简化了上述“对所包含的对象的检查和执行操作”的过程:

- `ifPresent(Consumer)`: 当值存在时调用 **Consumer**，否则什么也不做。
- `orElse(otherObject)`: 如果值存在则直接返回，否则生成 **otherObject**。
- `orElseGet(Supplier)`: 如果值存在则直接返回，否则使用 **Supplier** 函数生成一个可替代对象。
- `orElseThrow(Supplier)`: 如果值存在直接返回，否则使用 **Supplier** 函数生成一个异常。

如下是针对不同便利函数的简单演示:

```
// streams/Optionals.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
public class Optionals {
    static void basics(Optional<String> optString) {
        if(optString.isPresent())
            System.out.println(optString.get());
```

```
        else
            System.out.println("Nothing inside!");
    }
    static void ifPresent(Optional<String> optString) {
        optString.ifPresent(System.out::println);
    }
    static void orElse(Optional<String> optString) {
        System.out.println(optString.orElse("Nada"));
    }
    static void orElseGet(Optional<String> optString) {
        System.out.println(
        optString.orElseGet(() -> "Generated"));
    }
    static void orElseThrow(Optional<String> optString) {
        try {
            System.out.println(optString.orElseThrow(
            () -> new Exception("Supplied")));
        } catch(Exception e) {
            System.out.println("Caught " + e);
        }
    }
    static void test(String testName, Consumer<Optional<String>> cos) {
        System.out.println(" === " + testName + " === ");
        cos.accept(Stream.of("Epithets").findFirst());
        cos.accept(Stream.<String>empty().findFirst());
    }
    public static void main(String[] args) {
        test("basics", Optionals::basics);
        test("ifPresent", Optionals::ifPresent);
        test("orElse", Optionals::orElse);
        test("orElseGet", Optionals::orElseGet);
        test("orElseThrow", Optionals::orElseThrow);
    }
}
```

输出结果:

```
=== basics ===
Epithets
Nothing inside!
=== ifPresent ===
Epithets
=== orElse ===
Epithets
Nada
=== orElseGet ===
Epithets
Generated
=== orElseThrow ===
Epithets
Caught java.lang.Exception: Supplied
```

`test()` 通过传入所有方法都适用的 **Consumer** 来避免重复代码。

`orElseThrow()` 通过 **catch** 关键字来捕获抛出的异常。更多细节，将在 异常 这一章节中学习。

15.4.2 创建 Optional

当我们在自己的代码中加入 **Optional** 时，可以使用下面 3 个静态方法：

- `empty()`：生成一个空 **Optional**。
- `of(value)`：将一个非空值包装到 **Optional** 里。
- `ofNullable(value)`：针对一个可能为空的值，为空时自动生成 **Optional.empty**，否则将值包装在 **Optional** 中。

下面来看看它是如何工作的。代码示例：

```
// streams/CreatingOptionals.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
class CreatingOptionals {
    static void test(String testName, Optional<String> opt) {
        System.out.println(" === " + testName + " === ");
        System.out.println(opt.orElse("Null"));
    }
    public static void main(String[] args) {
        test("empty", Optional.empty());
        test("of", Optional.of("Howdy"));
        try {
            test("of", Optional.of(null));
        } catch(Exception e) {
            System.out.println(e);
        }
        test("ofNullable", Optional.ofNullable("Hi"));
        test("ofNullable", Optional.ofNullable(null));
    }
}
```

输出结果：

```
=== empty ===
Null
=== of ===
Howdy
java.lang.NullPointerException
=== ofNullable ===
Hi
=== ofNullable ===
Null
```

我们不能通过传递 **null** 到 of() 来创建 Optional 对象。最安全的方法是，使用 ofNullable() 来优雅地处理 **null**。

15.4.3 Optional 对象操作

当我们的流管道生成了 **Optional** 对象，下面 3 个方法可使得 **Optional** 的后续能做更多的操作：

- filter(Predicate)：将 **Predicate** 应用于 **Optional** 中的内容并返回结果。当 **Optional** 不满足 **Predicate** 时返回空。如果 **Optional** 为空，则直接返回。
- map(Function)：如果 **Optional** 不为空，应用 **Function** 于 **Optional** 中的内容，并返回结果。否则直接返回 **Optional.empty**。
- flatMap(Function)：同 map()，但是提供的映射函数将结果包装在 **Optional** 对象中，因此 flatMap() 不会在最后进行任何包装。

以上方法都不适用于数值型 **Optional**。一般来说，流的 filter() 会在 **Predicate** 返回 **false** 时移除流元素。而 Optional.filter() 在失败时不会删除 **Optional**，而是将其保留下来，并转化为空。下面请看代码示例：

```
// streams/OptionalFilter.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
class OptionalFilter {
    static String[] elements = {
            "Foo", "", "Bar", "Baz", "Bingo"
    };
    static Stream<String> testStream() {
        return Arrays.stream(elements);
    }
    static void test(String descr, Predicate<String> pred) {
        System.out.println(" ---( " + descr + " )---");
        for(int i = 0; i <= elements.length; i++) {
            System.out.println(
                    testStream()
                            .skip(i)
                            .findFirst()
                            .filter(pred));
        }
    }
    public static void main(String[] args) {
        test("true", str -> true);
        test("false", str -> false);
        test("str != \"\"", str -> str != "");
        test("str.length() == 3", str -> str.length() == 3);
        test("startsWith(\"B\")",
```

```
                    str -> str.startsWith("B"));
    }
}
```

输出结果:

```
---( true )---
Optional[Foo]
Optional[]
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
---( false )---
Optional.empty
Optional.empty
Optional.empty
Optional.empty
Optional.empty
Optional.empty
---( str != "" )---
Optional[Foo]
Optional.empty
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
---( str.length() == 3 )---
Optional[Foo]
Optional.empty
Optional[Bar]
Optional[Baz]
Optional.empty
Optional.empty
---( startsWith("B") )---
Optional.empty
Optional.empty
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
```

即使输出看起来像流，特别是 `test()` 中的 for 循环。每一次的 for 循环时重新启动流，然后根据 for 循环的索引跳过指定个数的元素，这就是它最终在流中的每个连续元素上的结果。接下来调用 `findFirst()` 获取剩余元素中的第一个元素，结果会包装在 **Optional** 中。

注意，不同于普通 for 循环，这里的索引值范围并不是 `i < elements.length`，而是 `i <= elements.length`。所以最后一个元素实际上超出了流。方便的是，这将自动成为 **Optional.empty**，你可以在每一个测试的结尾中看到。

同 `map()` 一样，`Optional.map()` 应用于函数。它仅在 **Optional** 不为空时才应用映射函数，并将 **Optional** 的内容提取到映射函数。代码示例：

```
// streams/OptionalMap.java
import java.util.Arrays;
import java.util.function.Function;
import java.util.stream.Stream;

class OptionalMap {
    static String[] elements = {"12", "", "23", "45"};

    static Stream<String> testStream() {
        return Arrays.stream(elements);
    }

    static void test(String descr, Function<String, String> func) {
        System.out.println(" ---( " + descr + " )---");
        for (int i = 0; i <= elements.length; i++) {
            System.out.println(
                    testStream()
                            .skip(i)
                            .findFirst() // Produces an Optional
                            .map(func));
        }
    }

    public static void main(String[] args) {
        // If Optional is not empty, map() first extracts
        // the contents which it then passes
        // to the function:
        test("Add brackets", s -> "[" + s + "]");
        test("Increment", s -> {
            try {
                return Integer.parseInt(s) + 1 + "";
            } catch (NumberFormatException e) {
                return s;
            }
        });
        test("Replace", s -> s.replace("2", "9"));
        test("Take last digit", s -> s.length() > 0 ?
                s.charAt(s.length() - 1) + "" : s);
    }
    // After the function is finished, map() wraps the
    // result in an Optional before returning it:
}
```

输出结果：

```
---( Add brackets )---
Optional[[12]]
```

```
Optional[[]]
Optional[[23]]
Optional[[45]]
Optional.empty
---( Increment )---
Optional[13]
Optional[]
Optional[24]
Optional[46]
Optional.empty
---( Replace )---
Optional[19]
Optional[]
Optional[93]
Optional[45]
Optional.empty
---( Take last digit )---
Optional[2]
Optional[]
Optional[3]
Optional[5]
Optional.empty
```

映射函数的返回结果会自动包装成为 **Optional**。**Optional.empty** 会被直接跳过。

Optional 的 `flatMap()` 应用于已生成 **Optional** 的映射函数，所以 `flatMap()` 不会像 `map()` 那样将结果封装在 **Optional** 中。代码示例：

```java
// streams/OptionalFlatMap.java
import java.util.Arrays;
import java.util.Optional;
import java.util.function.Function;
import java.util.stream.Stream;

class OptionalFlatMap {
    static String[] elements = {"12", "", "23", "45"};

    static Stream<String> testStream() {
        return Arrays.stream(elements);
    }

    static void test(String descr,
                     Function<String, Optional<String>> func) {
        System.out.println(" ---( " + descr + " )---");
        for (int i = 0; i <= elements.length; i++) {
            System.out.println(
                    testStream()
                            .skip(i)
                            .findFirst()
                            .flatMap(func));
        }
```

```
    }

    public static void main(String[] args) {
        test("Add brackets",
                s -> Optional.of("[" + s + "]"));
        test("Increment", s -> {
            try {
                return Optional.of(
                        Integer.parseInt(s) + 1 + "");
            } catch (NumberFormatException e) {
                return Optional.of(s);
            }
        });
        test("Replace",
                s -> Optional.of(s.replace("2", "9")));
        test("Take last digit",
                s -> Optional.of(s.length() > 0 ?
                        s.charAt(s.length() - 1) + ""
                        : s));
    }
}
```

输出结果:

```
---( Add brackets )---
Optional[[12]]
Optional[[]]
Optional[[23]]
Optional[[45]]
Optional.empty
 ---( Increment )---
Optional[13]
Optional[]
Optional[24]
Optional[46]
Optional.empty
 ---( Replace )---
Optional[19]
Optional[]
Optional[93]
Optional[45]
Optional.empty
 ---( Take last digit )---
Optional[2]
Optional[]
Optional[3]
Optional[5]
Optional.empty
```

同 `map()`, `flatMap()` 将提取非空 **Optional** 的内容并将其应用在映射函数。唯一的区别就是

`flatMap()` 不会把结果包装在 **Optional** 中，因为映射函数已经被包装过了。在如上示例中，我们已经在每一个映射函数中显式地完成了包装，但是很显然 `Optional.flatMap()` 是为那些自己已经生成 **Optional** 的函数而设计的。

15.4.4 Optional 流

假设你的生成器可能产生 `null` 值，那么当用它来创建流时，你会自然地想到用 **Optional** 来包装元素。如下是它的样子，代码示例：

```
// streams/Signal.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
public class Signal {
    private final String msg;
    public Signal(String msg) { this.msg = msg; }
    public String getMsg() { return msg; }
    @Override
    public String toString() {
        return "Signal(" + msg + ")";
    }
    static Random rand = new Random(47);
    public static Signal morse() {
        switch(rand.nextInt(4)) {
            case 1: return new Signal("dot");
            case 2: return new Signal("dash");
            default: return null;
        }
    }
    public static Stream<Optional<Signal>> stream() {
        return Stream.generate(Signal::morse)
                .map(signal -> Optional.ofNullable(signal));
    }
}
```

当我们使用这个流的时候，必须要弄清楚如何解包 **Optional**。代码示例：

```
// streams/StreamOfOptionals.java
import java.util.*;
import java.util.stream.*;
public class StreamOfOptionals {
    public static void main(String[] args) {
        Signal.stream()
                .limit(10)
                .forEach(System.out::println);
        System.out.println(" ---");
        Signal.stream()
                .limit(10)
```

```
                    .filter(Optional::isPresent)
                    .map(Optional::get)
                    .forEach(System.out::println);
    }
}
```

输出结果:

```
Optional[Signal(dash)]
Optional[Signal(dot)]
Optional[Signal(dash)]
Optional.empty
Optional.empty
Optional[Signal(dash)]
Optional.empty
Optional[Signal(dot)]
Optional[Signal(dash)]
Optional[Signal(dash)]
---
Signal(dot)
Signal(dot)
Signal(dash)
Signal(dash)
```

在这里，我们使用 `filter()` 来保留那些非空 **Optional**，然后在 `map()` 中使用 `get()` 获取元素。由于每种情况都需要定义“空值”的含义，所以通常我们要为每个应用程序采用不同的方法。

15.5 终端操作

以下操作将会获取流的最终结果。至此我们无法再继续往后传递流。可以说，终端操作总是我们在流管道中所做的最后一件事。

15.5.1 数组

- `toArray()`：将流转换成适当类型的数组。
- `toArray(generator)`：在特殊情况下，生成自定义类型的数组。

当我们需要得到数组类型的数据以便于后续操作时，上面的方法就很有用。假设我们需要复用流产生的随机数时，就可以这么使用。代码示例:

```
// streams/RandInts.java
package streams;
import java.util.*;
import java.util.stream.*;
public class RandInts {
```

```
    private static int[] rints = new Random(47).ints(0, 1000).limit
        (100).toArray();
    public static IntStream rands() {
        return Arrays.stream(rints);
    }
}
```

上例将 100 个数值范围在 0 到 1000 之间的随机数流转换成为数组并将其存储在 `rints` 中。这样一来，每次调用 `rands()` 的时候可以重复获取相同的整数流。

15.5.2 循环

- `forEach(Consumer)`常见如 `System.out::println` 作为 **Consumer** 函数。
- `forEachOrdered(Consumer)`：保证 `forEach` 按照原始流顺序操作。

第一种形式：无序操作，仅在引入并行流时才有意义。在 并发编程 章节之前我们不会深入研究这个问题。这里简单介绍下 `parallel()`：可实现多处理器并行操作。实现原理为将流分割为多个（通常数目为 CPU 核心数）并在不同处理器上分别执行操作。因为我们采用的是内部迭代，而不是外部迭代，所以这是可能实现的。

`parallel()` 看似简单，实则棘手。更多内容将在稍后的 并发编程 章节中学习。

下例引入 `parallel()` 来帮助理解 `forEachOrdered(Consumer)` 的作用和使用场景。代码示例：

```
// streams/ForEach.java
import java.util.*;
import java.util.stream.*;
import static streams.RandInts.*;
public class ForEach {
    static final int SZ = 14;
    public static void main(String[] args) {
        rands().limit(SZ)
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();
        rands().limit(SZ)
                .parallel()
                .forEach(n -> System.out.format("%d ", n));
        System.out.println();
        rands().limit(SZ)
                .parallel()
                .forEachOrdered(n -> System.out.format("%d ", n));
    }
}
```

输出结果：

```
258 555 693 861 961 429 868 200 522 207 288 128 551 589
551 861 429 589 200 522 555 693 258 128 868 288 961 207
```

```
258 555 693 861 961 429 868 200 522 207 288 128 551 589
```

为了方便测试不同大小的数组，我们抽离出了 SZ 变量。结果很有趣：在第一个流中，未使用 parallel()，所以 rands() 按照元素迭代出现的顺序显示结果；在第二个流中，引入parallel()，即便流很小，输出的结果顺序也和前面不一样。这是由于多处理器并行操作的原因。多次运行测试，结果均不同。多处理器并行操作带来的非确定性因素造成了这样的结果。

在最后一个流中，同时使用了 parallel() 和 forEachOrdered() 来强制保持原始流顺序。因此，对非并行流使用 forEachOrdered() 是没有任何影响的。

15.5.3 集合

- collect(Collector)：使用 **Collector** 收集流元素到结果集合中。
- collect(Supplier, BiConsumer, BiConsumer)：同上，第一个参数 **Supplier** 创建了一个新结果集合，第二个参数 **BiConsumer** 将下一个元素包含到结果中，第三个参数 **BiConsumer** 用于将两个值组合起来。

在这里我们只是简单介绍了几个 **Collectors** 的运用示例。实际上，它还有一些非常复杂的操作实现，可通过查看 java.util.stream.Collectors 的 API 文档了解。例如，我们可以将元素收集到任意一种特定的集合中。

假设我们现在为了保证元素有序，将元素存储在 **TreeSet** 中。**Collectors** 里面没有特定的 toTreeSet()，但是我们可以通过将集合的构造函数引用传递给 Collectors.toCollection()，从而构建任何类型的集合。下面我们来将一个文件中的单词收集到 **TreeSet** 集合中。代码示例：

```java
// streams/TreeSetOfWords.java
import java.util.*;
import java.nio.file.*;
import java.util.stream.*;
public class TreeSetOfWords {
    public static void
    main(String[] args) throws Exception {
        Set<String> words2 =
                Files.lines(Paths.get("TreeSetOfWords.java"))
                        .flatMap(s -> Arrays.stream(s.split("\\W+")))
                        .filter(s -> !s.matches("\\d+")) // No numbers
                        .map(String::trim)
                        .filter(s -> s.length() > 2)
                        .limit(100)
                        .collect(Collectors.toCollection(TreeSet::new))
                            ;
        System.out.println(words2);
    }
}
```

输出结果：

```
[Arrays, Collectors, Exception, Files, Output, Paths,
Set, String, System, TreeSet, TreeSetOfWords, args,
class, collect, file, filter, flatMap, get, import,
java, length, limit, lines, main, map, matches, new,
nio, numbers, out, println, public, split, static,
stream, streams, throws, toCollection, trim, util,
void, words2]
```

Files.`lines()` 打开 **Path** 并将其转换成为行流。下一行代码将匹配一个或多个非单词字符（`\\w+`）行进行分割，然后使用 **Arrays.**`stream()` 将其转化成为流，并将结果展平映射成为单词流。使用 `matches(\\d+)` 查找并移除全数字字符串（**注意**,`words2` 是通过的）。接下来我们使用 **String.**`trim()` 去除单词两边的空白，`filter()` 过滤所有长度小于 3 的单词，紧接着只获取 100 个单词，最后将其保存到 **TreeSet** 中。

我们也可以在流中生成 **Map**。代码示例：

```
// streams/MapCollector.java
import java.util.*;
import java.util.stream.*;
class Pair {
    public final Character c;
    public final Integer i;
    Pair(Character c, Integer i) {
        this.c = c;
        this.i = i;
    }
    public Character getC() { return c; }
    public Integer getI() { return i; }
    @Override
    public String toString() {
        return "Pair(" + c + ", " + i + ")";
    }
}
class RandomPair {
    Random rand = new Random(47);
    // An infinite iterator of random capital letters:
    Iterator<Character> capChars = rand.ints(65,91)
            .mapToObj(i -> (char)i)
            .iterator();
    public Stream<Pair> stream() {
        return rand.ints(100, 1000).distinct()
                .mapToObj(i -> new Pair(capChars.next(), i));
    }
}
public class MapCollector {
    public static void main(String[] args) {
        Map<Integer, Character> map =
                new RandomPair().stream()
                        .limit(8)
```

```
                        .collect(
                                Collectors.toMap(Pair::getI, Pair::getC
                                    ));
        System.out.println(map);
    }
}
```

输出结果:

```
{688=W, 309=C, 293=B, 761=N, 858=N, 668=G, 622=F, 751=N}
```

Pair 只是一个基础的数据对象。**RandomPair** 创建了随机生成的 **Pair** 对象流。在 Java 中，我们不能直接以某种方式组合两个流。所以这里创建了一个整数流，并且使用 `mapToObj()` 将其转化成为 **Pair** 流。**capChars** 随机生成的大写字母迭代器从流开始，然后 `iterator()` 允许我们在 `stream()` 中使用它。就我所知，这是组合多个流以生成新的对象流的唯一方法。

在这里，我们只使用最简单形式的 `Collectors.toMap()`，这个方法值需要一个可以从流中获取键值对的函数。还有其他重载形式，其中一种形式是在遇到键值冲突时，需要一个函数来处理这种情况。

大多数情况下，`java.util.stream.Collectors` 中预设的 **Collector** 就能满足我们的要求。除此之外，你还可以使用第二种形式的 `collect()`。我把它留作更高级的练习，下例给出基本用法:

```java
// streams/SpecialCollector.java
import java.util.*;
import java.util.stream.*;
public class SpecialCollector {
    public static void main(String[] args) throws Exception {
        ArrayList<String> words =
                FileToWords.stream("Cheese.dat")
                        .collect(ArrayList::new,
                                ArrayList::add,
                                ArrayList::addAll);
        words.stream()
                .filter(s -> s.equals("cheese"))
                .forEach(System.out::println);
    }
}
```

输出结果:

```
cheese
cheese
```

在这里，**ArrayList** 的方法已经执行了你所需要的操作，但是似乎更有可能的是，如果你必须使用这种形式的 `collect()`，则必须自己创建特殊的定义。

15.5.4 组合

- `reduce(BinaryOperator)`：使用 **BinaryOperator** 来组合所有流中的元素。因为流可能为空，其返回值为 **Optional**。
- `reduce(identity, BinaryOperator)`：功能同上，但是使用 **identity** 作为其组合的初始值。因此如果流为空，**identity** 就是结果。
- `reduce(identity, BiFunction, BinaryOperator)`：更复杂的使用形式（暂不介绍），这里把它包含在内，因为它可以提高效率。通常，我们可以显式地组合 `map()` 和 `reduce()` 来更简单的表达它。

下面来看下 `reduce` 的代码示例：

```
// streams/Reduce.java
import java.util.*;
import java.util.stream.*;
class Frobnitz {
    int size;
    Frobnitz(int sz) { size = sz; }
    @Override
    public String toString() {
        return "Frobnitz(" + size + ")";
    }
    // Generator:
    static Random rand = new Random(47);
    static final int BOUND = 100;
    static Frobnitz supply() {
        return new Frobnitz(rand.nextInt(BOUND));
    }
}
public class Reduce {
    public static void main(String[] args) {
        Stream.generate(Frobnitz::supply)
                .limit(10)
                .peek(System.out::println)
                .reduce((fr0, fr1) -> fr0.size < 50 ? fr0 : fr1)
                .ifPresent(System.out::println);
    }
}
```

输出结果：

```
Frobnitz(58)
Frobnitz(55)
Frobnitz(93)
Frobnitz(61)
Frobnitz(61)
Frobnitz(29)
Frobnitz(68)
```

```
Frobnitz(0)
Frobnitz(22)
Frobnitz(7)
Frobnitz(29)
```

Frobnitz 包含了一个名为 `supply()` 的生成器；因为这个方法对于 `Supplier<Frobnitz>` 是签名兼容的，我们可以将其方法引用传递给 `Stream.generate()`（这种签名兼容性被称作结构一致性）。无“初始值”的 `reduce()`方法返回值是 **Optional** 类型。`Optional.ifPresent()` 只有在结果非空的时候才会调用 `Consumer<Frobnitz>` （`println` 方法可以被调用是因为 **Frobnitz** 可以通过 `toString()` 方法转换成 **String**）。

Lambda 表达式中的第一个参数 `fr0` 是上一次调用 `reduce()` 的结果。而第二个参数 `fr1` 是从流传递过来的值。

`reduce()` 中的 Lambda 表达式使用了三元表达式来获取结果，当其长度小于 50 的时候获取 `fr0` 否则获取序列中的下一个值 `fr1`。当取得第一个长度小于 50 的 `Frobnitz`，只要得到结果就会忽略其他。这是个非常奇怪的约束，也确实让我们对 `reduce()` 有了更多的了解。

15.5.5 匹配

- `allMatch(Predicate)`：如果流的每个元素根据提供的 **Predicate** 都返回 true 时，结果返回为 true。在第一个 false 时，则停止执行计算。
- `anyMatch(Predicate)`：如果流中的任意一个元素根据提供的 **Predicate** 返回 true 时，结果返回为 true。在第一个 false 是停止执行计算。
- `noneMatch(Predicate)`：如果流的每个元素根据提供的 **Predicate** 都返回 false 时，结果返回为 true。在第一个 true 时停止执行计算。

我们已经在 `Prime.java` 中看到了 `noneMatch()` 的示例；`allMatch()` 和 `anyMatch()` 的用法基本上是等同的。下面我们来探究一下短路行为。为了消除冗余代码，我们创建了 `show()`。首先我们必须知道如何统一地描述这三个匹配器的操作，然后再将其转换为 **Matcher** 接口。代码示例：

```java
// streams/Matching.java
// Demonstrates short-circuiting of *Match() operations
import java.util.stream.*;
import java.util.function.*;
import static streams.RandInts.*;

interface Matcher extends BiPredicate<Stream<Integer>, Predicate<
    Integer>> {}

public class Matching {
    static void show(Matcher match, int val) {
        System.out.println(
                match.test(
                        IntStream.rangeClosed(1, 9)
```

“这”不可能是问题的原因。

```
                                .boxed()
                                .peek(n -> System.out.format("%d ", n))
                                    ,
                        n -> n < val));
    }
    public static void main(String[] args) {
        show(Stream::allMatch, 10);
        show(Stream::allMatch, 4);
        show(Stream::anyMatch, 2);
        show(Stream::anyMatch, 0);
        show(Stream::noneMatch, 5);
        show(Stream::noneMatch, 0);
    }
}
```

输出结果:

```
1 2 3 4 5 6 7 8 9 true
1 2 3 4 false
1 true
1 2 3 4 5 6 7 8 9 false
1 false
1 2 3 4 5 6 7 8 9 true
```

BiPredicate 是一个二元谓词，它只能接受两个参数且只返回 true 或者 false。它的第一个参数是我们要测试的流，第二个参数是一个谓词 **Predicate**。**Matcher** 适用于所有的 **Stream::*Match** 方法，所以我们可以传递每一个到 `show()` 中。`match.test()` 的调用会被转换成 **Stream::*Match** 函数的调用。

`show()` 获取两个参数，**Matcher** 匹配器和用于表示谓词测试 **n < val** 中最大值的 **val**。这个方法生成一个 1-9 之间的整数流。`peek()` 是用于向我们展示测试在短路之前的情况。从输出中可以看到每次都发生了短路。

15.5.6 查找

- `findFirst()`：返回第一个流元素的 **Optional**，如果流为空返回 **Optional.empty**。
- `findAny(`：返回含有任意流元素的 **Optional**，如果流为空返回 **Optional.empty**。

代码示例:

```
// streams/SelectElement.java
import java.util.*;
import java.util.stream.*;
import static streams.RandInts.*;
public class SelectElement {
    public static void main(String[] args) {
        System.out.println(rands().findFirst().getAsInt());
        System.out.println(
```

```
                rands().parallel().findFirst().getAsInt());
        System.out.println(rands().findAny().getAsInt());
        System.out.println(
                rands().parallel().findAny().getAsInt());
    }
}
```

输出结果:

```
258
258
258
242
```

`findFirst()` 无论流是否为并行化的, 总是会选择流中的第一个元素。对于非并行流, `findAny()` 会选择流中的第一个元素（即使从定义上来看是选择任意元素)。在这个例子中，我们使用 `parallel()` 来并行流从而引入 `findAny()` 选择非第一个流元素的可能性。

如果必须选择流中最后一个元素，那就使用 `reduce()`。代码示例:

```java
// streams/LastElement.java
import java.util.*;
import java.util.stream.*;
public class LastElement {
    public static void main(String[] args) {
        OptionalInt last = IntStream.range(10, 20)
                .reduce((n1, n2) -> n2);
        System.out.println(last.orElse(-1));
        // Non-numeric object:
        Optional<String> lastobj =
                Stream.of("one", "two", "three")
                        .reduce((n1, n2) -> n2);
        System.out.println(
                lastobj.orElse("Nothing there!"));
    }
}
```

输出结果:

```
19
three
```

`reduce()` 的参数只是用最后一个元素替换了最后两个元素，最终只生成最后一个元素。如果为数字流，你必须使用相近的数字 **Optional** 类型（numeric optional type)，否则使用 **Optional** 类型，就像上例中的 `Optional<String>`。

15.5.7 信息

- `count()`: 流中的元素个数。

- max(Comparator)：根据所传入的 **Comparator** 所决定的“最大”元素。
- min(Comparator)：根据所传入的 **Comparator** 所决定的“最小”元素。

String 类型有预设的 **Comparator** 实现。代码示例：

```
// streams/Informational.java
import java.util.stream.*;
import java.util.function.*;
public class Informational {
    public static void
    main(String[] args) throws Exception {
        System.out.println(
                FileToWords.stream("Cheese.dat").count());
        System.out.println(
                FileToWords.stream("Cheese.dat")
                        .min(String.CASE_INSENSITIVE_ORDER)
                        .orElse("NONE"));
        System.out.println(
                FileToWords.stream("Cheese.dat")
                        .max(String.CASE_INSENSITIVE_ORDER)
                        .orElse("NONE"));
    }
}
```

输出结果：

```
32
a
you
```

min() 和 max() 的返回类型为 **Optional**，这需要我们使用 orElse()来解包。

15.5.8 数字流信息

- average()：求取流元素平均值。
- max() 和 min()：数值流操作无需 **Comparator**。
- sum()：对所有流元素进行求和。
- summaryStatistics()：生成可能有用的数据。目前并不太清楚这个方法存在的必要性，因为我们其实可以用更直接的方法获得需要的数据。

```
// streams/NumericStreamInfo.java
import java.util.stream.*;
import static streams.RandInts.*;
public class NumericStreamInfo {
    public static void main(String[] args) {
        System.out.println(rands().average().getAsDouble());
        System.out.println(rands().max().getAsInt());
        System.out.println(rands().min().getAsInt());
```

```
        System.out.println(rands().sum());
        System.out.println(rands().summaryStatistics());
    }
}
```

输出结果:

```
507.94
998
8
50794
IntSummaryStatistics{count=100, sum=50794, min=8, average=507.940000,
    max=998}
```

上例操作对于 **LongStream** 和 **DoubleStream** 同样适用。

15.6 本章小结

流式操作改变并极大地提升了 Java 语言的可编程性，并可能极大地阻止了 Java 编程人员向诸如 Scala 这种函数式语言的流转。在本书的剩余部分，我们将尽可能地使用流。

C 程序员永远不会灭亡。他们只是 cast 成了 void。

16 异常

> Java 的基本理念是“结构不佳的代码不能运行”。

改进的错误恢复机制是提高代码健壮性的最强有力的方式。错误恢复在我们所编写的每一个程序中都是基本的要素，但是在 Java 中它显得格外重要，因为 Java 的主要目标之一就是创建供他人使用的程序构件。

发现错误的理想时机是在编译阶段，也就是在你试图运行程序之前。然而，编译期间并不能找出所有的错误，余下的问题必须在运行期间解决。这就需要错误源能通过某种方式，把适当的信息传递给某个接收者——该接收者将知道如何正确处理这个问题。

> 要想创建健壮的系统，它的每一个构件都必须是健壮的。

Java 使用异常来提供一致的错误报告模型，使得构件能够与客户端代码可靠地沟通问题。

Java 中的异常处理的目的在于通过使用少于目前数量的代码来简化大型、可靠的程序的生成，并且通过这种方式可以使你更加确信：你的应用中没有未处理的错误。异常的相关知识学起来并非艰涩难懂，并且它属于那种可以使你的项目受益明显、立竿见影的特性之一。

因为异常处理是 Java 中唯一官方的错误报告机制，并且通过编译器强制执行，所以不学习异常处理的话，书中也就只能写出那么些例子了。本章将向读者介绍如何编写正确的异常处理程序，并将展示当方法出问题的时候，如何产生自定义的异常。

16.1 异常概念

C 以及其他早期语言常常具有多种错误处理模式，这些模式往往建立在约定俗成的基础之上，而并不属于语言的一部分。通常会返回某个特殊值或者设置某个标志，并且假定接收者将对这个返回值或标志进行检查，以判定是否发生了错误。然而，随着时间的推移，人们发现，高傲的程序员们在使用程序库的时候更倾向于认为：“对，错误也许会发生，但那是别人造成的，不关我的事”。所以，程序员不去检查错误情形也就不足为奇了（何况对某些错误情形的检查确实很无聊）。如果的确在每次调用方法的时候都彻底地进行错误检查，代码很可能会变得难以阅读。正是由于程序员还仍然用这些方式拼凑系统，所以他们拒绝承认这样一个事实：对于构造大型、健壮、可维护的程序而言，这种错误处理模式已经成为了主要障碍。

解决的办法是，用强制规定的形式来消除错误处理过程中随心所欲的因素。这种做法由来已久，对异常处理的实现可以追溯到 20 世纪 60 年代的操作系统，甚至于 BASIC 语言中的“on error goto”语句。而 C++ 的异常处理机制基于 Ada，Java 中的异常处理机制则建立在 C++ 的基础之上（尽管看上去更像 Object Pascal）。

“异常”这个词有“我对此感到意外”的意思。问题出现了，你也许不清楚该如何处理，但你的确知道不应该置之不理，你要停下来，看看是不是有别人或在别的地方，能够处理这个问题。只是在当前的环境中还没有足够的信息来解决这个问题，所以就把这个问题提交到一个更高级别的环境中，在那里将作出正确的决定。

异常往往能降低错误处理代码的复杂度。如果不使用异常，那么就必须检查特定的错误，并在程序中的许多地方去处理它。而如果使用异常，那就不必在方法调用处进行检查，因为异常机制将保证能够捕获这个错误。理想情况下，只需在一个地方处理错误，即所谓的异常处理程序中。这种方式不仅节省代码，而且把“描述在正常执行过程中做什么事”的代码和“出了问题怎么办”的代码相分离。总之，与以前的错误处理方法相比，异常机制使代码的阅读、编写和调试工作更加井井有条。

16.2 基本异常

异常情形（exceptional condition）是指阻止当前方法或作用域继续执行的问题。把异常情形与普通问题相区分很重要，所谓的普通问题是指，在当前环境下能得到足够的信息，总能处理这个错误。而对于异常情形，就不能继续下去了，因为在当前环境下无法获得必要的信息来解决问题。你所能做的就是从当前环境跳出，并且把问题提交给上一级环境。这就是抛出异常时所发生的事情。

除法就是一个简单的例子。除数有可能为 0，所以先进行检查很有必要。但除数为 0 代表的究竟是什么意思呢？通过当前正在解决的问题环境，或许能知道该如何处理除数为 0 的情况。但如果这是一个意料之外的值，你也不清楚该如何处理，那就要抛出异常，而不是顺着原来的路径继续执行下去。

当抛出异常后，有几件事会随之发生。首先，同 Java 中其他对象的创建一样，将使用 new 在堆上创建异常对象。然后，当前的执行路径（它不能继续下去了）被终止，并且从当前环境中弹出对异常对象的引用。此时，异常处理机制接管程序，并开始寻找一个恰当的地方来继续执行程序。这个恰当的地方就是异常处理程序，它的任务是将程序从错误状态中恢复，以使程序能要么换一种方式运行，要么继续运行下去。

举一个抛出异常的简单例子。对于对象引用 t，传给你的时候可能尚未被初始化。所以在使用这个对象引用调用其方法之前，会先对引用进行检查。可以创建一个代表错误信息的对象，并且将它从当前环境中“抛出”，这样就把错误信息传播到了“更大”的环境中。这被称为*抛出一个异常*，看起来像这样：

```
if(t == null)
    throw new NullPointerException();
```

这就抛出了异常，于是在当前环境下就不必再为这个问题操心了，它将在别的地方得到处理。具体

是哪个“地方”后面很快就会介绍。

异常使得我们可以将每件事都当作一个事务来考虑，而异常可以看护着这些事务的底线“…事务的基本保障是我们所需的在分布式计算中的异常处理。事务是计算机中的合同法，如果出了什么问题，我们只需要放弃整个计算。”我们还可以将异常看作是一种内建的恢复（undo）系统，因为（在细心使用的情况下）我们在程序中可以拥有各种不同的恢复点。如果程序的某部分失败了，异常将“恢复”到程序中某个已知的稳定点上。

异常最重要的方面之一就是如果发生问题，它们将不允许程序沿着其正常的路径继续走下去。在 C 和 C++ 这样的语言中，这可真是个问题，尤其是 C，它没有任何办法可以强制程序在出现问题时停止在某条路径上运行下去，因此我们有可能会较长时间地忽略问题，从而会陷入完全不恰当的状态中。异常允许我们（如果没有其他手段）强制程序停止运行，并告诉我们出现了什么问题，或者（理想状态下）强制程序处理问题，并返回到稳定状态。

16.2.1 异常参数

与使用 Java 中的其他对象一样，我们总是用 new 在堆上创建异常对象，这也伴随着存储空间的分配和构造器的调用。所有标准异常类都有两个构造器：一个是无参构造器；另一个是接受字符串作为参数，以便能把相关信息放入异常对象的构造器：

```
throw new NullPointerException("t = null");
```

不久读者将看到，要把这个字符串的内容提取出来可以有多种不同的方法。

关键字 **throw** 将产生许多有趣的结果。在使用 **new** 创建了异常对象之后，此对象的引用将传给 **throw**。尽管异常对象的类型通常与方法设计的返回类型不同，但从效果上看，它就像是从方法“返回”的。可以简单地把异常处理看成一种不同的返回机制，当然若过分强调这种类比的话，就会有麻烦了。另外还能用抛出异常的方式从当前的作用域退出。在这两种情况下，将会返回一个异常对象，然后退出方法或作用域。

抛出异常与方法正常返回的相似之处到此为止。因为异常返回的“地点”与普通方法调用返回的“地点”完全不同。（异常将在一个恰当的异常处理程序中得到解决，它的位置可能离异常被抛出的地方很远，也可能会跨越方法调用栈的许多层级。）

此外，能够抛出任意类型的 **Throwable** 对象，它是异常类型的根类。通常，对于不同类型的错误，要抛出相应的异常。错误信息可以保存在异常对象内部或者用异常类的名称来暗示。上一层环境通过这些信息来决定如何处理异常。（通常，唯一的信息只有异常的类型名，而在异常对象内部没有任何有意义的信息。）

16.3 异常捕获

要明白异常是如何被捕获的，必须首先理解监控区域（guarded region）的概念。它是一段可能产生异常的代码，并且后面跟着处理这些异常的代码。

16.3.1 try 语句块

如果在方法内部抛出了异常（或者在方法内部调用的其他方法抛出了异常），这个方法将在抛出异常的过程中结束。要是不希望方法就此结束，可以在方法内设置一个特殊的块来捕获异常。因为在这个块里“尝试”各种（可能产生异常的）方法调用，所以称为 try 块。它是跟在 try 关键字之后的普通程序块：

```
try {
    // Code that might generate exceptions
}
```

对于不支持异常处理的程序语言，要想仔细检查错误，就得在每个方法调用的前后加上设置和错误检查的代码，甚至在每次调用同一方法时也得这么做。有了异常处理机制，可以把所有动作都放在 try 块里，然后只需在一个地方就可以捕获所有异常。这意味着你的代码将更容易编写和阅读，因为代码的意图和错误检查不是混淆在一起的。

16.3.2 异常处理程序

当然，抛出的异常必须在某处得到处理。这个“地点”就是异常处理程序，而且针对每个要捕获的异常，得准备相应的处理程序。异常处理程序紧跟在 try 块之后，以关键字 catch 表示：

```
try {
    // Code that might generate exceptions
} catch(Type1 id1) {
    // Handle exceptions of Type1
} catch(Type2 id2) {
    // Handle exceptions of Type2
} catch(Type3 id3) {
    // Handle exceptions of Type3
}
// etc.
```

每个 catch 子句（异常处理程序）看起来就像是接收且仅接收一个特殊类型的参数的方法。可以在处理程序的内部使用标识符（id1，id2 等等），这与方法参数的使用很相似。有时可能用不到标识符，因为异常的类型已经给了你足够的信息来对异常进行处理，但标识符并不可以省略。

异常处理程序必须紧跟在 try 块之后。当异常被抛出时，异常处理机制将负责搜寻参数与异常类型相匹配的第一个处理程序。然后进入 catch 子句执行，此时认为异常得到了处理。一旦 catch 子句结

束，则处理程序的查找过程结束。注意，只有匹配的 catch 子句才能得到执行；这与 switch 语句不同，switch 语句需要在每一个 case 后面跟一个 break，以避免执行后续的 case 子句。

注意在 try 块的内部，许多不同的方法调用可能会产生类型相同的异常，而你只需要提供一个针对此类型的异常处理程序。

16.3.3 终止与恢复

异常处理理论上有两种基本模型。Java 支持终止模型（它是 Java 和 C++ 所支持的模型）。在这种模型中，将假设错误非常严重，以至于程序无法返回到异常发生的地方继续执行。一旦异常被抛出，就表明错误已无法挽回，也不能回来继续执行。

另一种称为恢复模型。意思是异常处理程序的工作是修正错误，然后重新尝试调用出问题的方法，并认为第二次能成功。对于恢复模型，通常希望异常被处理之后能继续执行程序。如果想要用 Java 实现类似恢复的行为，那么在遇见错误时就不能抛出异常，而是调用方法来修正该错误。或者，把 try 块放在 while 循环里，这样就不断地进入 try 块，直到得到满意的结果。

在过去，使用支持恢复模型异常处理的操作系统的程序员们最终还是转向使用类似“终止模型”的代码，并且忽略恢复行为。所以虽然恢复模型开始显得很吸引人，但不是很实用。其中的主要原因可能是它所导致的耦合：恢复性的处理程序需要了解异常抛出的地点，这势必要包含依赖于抛出位置的非通用性代码。这增加了代码编写和维护的困难，对于异常可能会从许多地方抛出的大型程序来说，更是如此。

16.4 自定义异常

不必拘泥于 Java 中已有的异常类型。Java 提供的异常体系不可能预见所有的希望加以报告的错误，所以可以自己定义异常类来表示程序中可能会遇到的特定问题。

要自己定义异常类，必须从已有的异常类继承，最好是选择意思相近的异常类继承（不过这样的异常并不容易找）。建立新的异常类型最简单的方法就是让编译器为你产生无参构造器，所以这几乎不用写多少代码：

```
// exceptions/InheritingExceptions.java
// Creating your own exceptions
class SimpleException extends Exception {}

public class InheritingExceptions {
    public void f() throws SimpleException {
        System.out.println(
                "Throw SimpleException from f()");
        throw new SimpleException();
    }
    public static void main(String[] args) {
        InheritingExceptions sed =
```

```
                    new InheritingExceptions();
            try {
                sed.f();
            } catch(SimpleException e) {
                System.out.println("Caught it!");
            }
        }
}
```

输出为:

```
Throw SimpleException from f()
Caught it!
```

编译器创建了无参构造器，它将自动调用基类的无参构造器。本例中不会得到像 SimpleException(String) 这样的构造器，这种构造器也不实用。你将看到，对异常来说，最重要的部分就是类名，所以本例中建立的异常类在大多数情况下已经够用了。

本例的结果被打印到了控制台上，本书的输出显示系统正是在控制台上自动地捕获和测试这些结果的。但是，你也许想通过写入 System.err 而将错误发送给标准错误流。通常这比把错误信息输出到 System.out 要好，因为 System.out 也许会被重定向。如果把结果送到 System.err，它就不会随 System.out 一起被重定向，这样更容易被用户注意。

你也可以为异常类创建一个接受字符串参数的构造器:

```
// exceptions/FullConstructors.java
class MyException extends Exception {
    MyException() {}
    MyException(String msg) { super(msg); }
}
public class FullConstructors {
    public static void f() throws MyException {
        System.out.println("Throwing MyException from f()");
        throw new MyException();
    }
    public static void g() throws MyException {
        System.out.println("Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException e) {
            e.printStackTrace(System.out);
        }
        try {
            g();
        } catch(MyException e) {
            e.printStackTrace(System.out);
        }
```

```
    }
}
```

输出为:

```
Throwing MyException from f()
MyException
    at FullConstructors.f(FullConstructors.java:11)
    at
FullConstructors.main(FullConstructors.java:19)
Throwing MyException from g()
MyException: Originated in g()
    at FullConstructors.g(FullConstructors.java:15)
    at
FullConstructors.main(FullConstructors.java:24)
```

新增的代码非常简短：两个构造器定义了 MyException 类型对象的创建方式。对于第二个构造器，使用 super 关键字明确调用了其基类构造器，它接受一个字符串作为参数。

在异常处理程序中，调用了在 Throwable 类声明（Exception 即从此类继承）的 printStackTrace() 方法。就像从输出中看到的，它将打印“从方法调用处直到异常抛出处”的方法调用序列。这里，信息被发送到了 System.out，并自动地被捕获和显示在输出中。但是，如果调用默认版本：

```
e.printStackTrace();
```

信息就会被输出到标准错误流。

16.4.1 异常与记录日志

你可能还想使用 java.util.logging 工具将输出记录到日志中。基本的日志记录功能还是相当简单易懂的：

```
// exceptions/LoggingExceptions.java
// An exception that reports through a Logger
// {ErrorOutputExpected}
import java.util.logging.*;
import java.io.*;
class LoggingException extends Exception {
    private static Logger logger =
            Logger.getLogger("LoggingException");
    LoggingException() {
        StringWriter trace = new StringWriter();
        printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
}
public class LoggingExceptions {
    public static void main(String[] args) {
        try {
```

```
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Caught " + e);
        }
        try {
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Caught " + e);
        }
    }
}
```

输出为:

```
___[ Error Output ]___
May 09, 2017 6:07:17 AM LoggingException <init>
SEVERE: LoggingException
at
LoggingExceptions.main(LoggingExceptions.java:20)
Caught LoggingException
May 09, 2017 6:07:17 AM LoggingException <init>
SEVERE: LoggingException
at
LoggingExceptions.main(LoggingExceptions.java:25)
Caught LoggingException
```

静态的 Logger.getLogger() 方法创建了一个 String 参数相关联的 Logger 对象（通常与错误相关的包名和类名），这个 Logger 对象会将其输出发送到 System.err。向 Logger 写入的最简单方式就是直接调用与日志记录消息的级别相关联的方法，这里使用的是 severe()。为了产生日志记录消息，我们欲获取异常抛出处的栈轨迹，但是 printStackTrace() 不会默认地产生字符串。为了获取字符串，我们需要使用重载的 printStackTrace() 方法，它接受一个 java.io.PrintWriter 对象作为参数（PrintWriter 会在 附录：I/O 流 一章详细介绍）。如果我们将一个 java.io.StringWriter 对象传递给这个 PrintWriter 的构造器，那么通过调用 toString() 方法，就可以将输出抽取为一个 String。

尽管由于 LoggingException 将所有记录日志的基础设施都构建在异常自身中，使得它所使用的方式非常方便，并因此不需要客户端程序员的干预就可以自动运行，但是更常见的情形是我们需要捕获和记录其他人编写的异常，因此我们必须在异常处理程序中生成日志消息；

```
// exceptions/LoggingExceptions2.java
// Logging caught exceptions
// {ErrorOutputExpected}
import java.util.logging.*;
import java.io.*;
public class LoggingExceptions2 {
    private static Logger logger =
            Logger.getLogger("LoggingExceptions2");
    static void logException(Exception e) {
        StringWriter trace = new StringWriter();
```

```
        e.printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
    public static void main(String[] args) {
        try {
            throw new NullPointerException();
        } catch(NullPointerException e) {
            logException(e);
        }
    }
}
```

输出结果为:

```
___[ Error Output ]___
May 09, 2017 6:07:17 AM LoggingExceptions2 logException
SEVERE: java.lang.NullPointerException
at
LoggingExceptions2.main(LoggingExceptions2.java:17)
```

还可以更进一步自定义异常，比如加入额外的构造器和成员:

```
// exceptions/ExtraFeatures.java
// Further embellishment of exception classes
class MyException2 extends Exception {
    private int x;
    MyException2() {}
    MyException2(String msg) { super(msg); }
    MyException2(String msg, int x) {
        super(msg);
        this.x = x;
    }
    public int val() { return x; }
    @Override
    public String getMessage() {
        return "Detail Message: "+ x
                + " "+ super.getMessage();
    }
}
public class ExtraFeatures {
    public static void f() throws MyException2 {
        System.out.println(
                "Throwing MyException2 from f()");
        throw new MyException2();
    }
    public static void g() throws MyException2 {
        System.out.println(
                "Throwing MyException2 from g()");
        throw new MyException2("Originated in g()");
    }
    public static void h() throws MyException2 {
```

```
        System.out.println(
                "Throwing MyException2 from h()");
        throw new MyException2("Originated in h()", 47);
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            g();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
            System.out.println("e.val() = " + e.val());
        }
    }
}
```

输出为:

```
Throwing MyException2 from f()
MyException2: Detail Message: 0 null
at ExtraFeatures.f(ExtraFeatures.java:24)
at ExtraFeatures.main(ExtraFeatures.java:38)
Throwing MyException2 from g()
MyException2: Detail Message: 0 Originated in g()
at ExtraFeatures.g(ExtraFeatures.java:29)
at ExtraFeatures.main(ExtraFeatures.java:43)
Throwing MyException2 from h()
MyException2: Detail Message: 47 Originated in h()
at ExtraFeatures.h(ExtraFeatures.java:34)
at ExtraFeatures.main(ExtraFeatures.java:48)
e.val() = 47
```

新的异常添加了字段 x 以及设定 x 值的构造器和读取数据的方法。此外，还覆盖了 Throwable.getMessage() 方法，以产生更详细的信息。对于异常类来说，getMessage() 方法有点类似于 toString() 方法。

既然异常也是对象的一种，所以可以继续修改这个异常类，以得到更强的功能。但要记住，使用程序包的客户端程序员可能仅仅只是查看一下抛出的异常类型，其他的就不管了（大多数 Java 库里的异常都是这么用的），所以对异常所添加的其他功能也许根本用不上。

16.5 异常声明

Java 鼓励人们把方法可能会抛出的异常告知使用此方法的客户端程序员。这是种优雅的做法，它使得调用者能确切知道写什么样的代码可以捕获所有潜在的异常。当然，如果提供了源代码，客户端程序员可以在源代码中查找 throw 语句来获知相关信息，然而程序库通常并不与源代码一起发布。为了预防这样的问题，Java 提供了相应的语法（并强制使用这个语法），使你能以礼貌的方式告知客户端程序员某个方法可能会抛出的异常类型，然后客户端程序员就可以进行相应的处理。这就是异常说明，它属于方法声明的一部分，紧跟在形式参数列表之后。

异常说明使用了附加的关键字 throws，后面接一个所有潜在异常类型的列表，所以方法定义可能看起来像这样：

```
void f() throws TooBig, TooSmall, DivZero { // ...
```

但是，要是这样写：

```
void f() { // ...
```

就表示此方法不会抛出任何异常（除了从 RuntimeException 继承的异常，它们可以在没有异常说明的情况下被抛出，这些将在后面进行讨论）。

代码必须与异常说明保持一致。如果方法里的代码产生了异常却没有进行处理，编译器会发现这个问题并提醒你：要么处理这个异常，要么就在异常说明中表明此方法将产生异常。通过这种自顶向下强制执行的异常说明机制，Java 在编译时就可以保证一定水平的异常正确性。

不过还是有个能“作弊”的地方：可以声明方法将抛出异常，实际上却不抛出。编译器相信了这个声明，并强制此方法的用户像真的抛出异常那样使用这个方法。这样做的好处是，为异常先占个位子，以后就可以抛出这种异常而不用修改已有的代码。在定义抽象基类和接口时这种能力很重要，这样派生类或接口实现就能够抛出这些预先声明的异常。

这种在编译时被强制检查的异常称为被检查的异常。

16.6 捕获所有异常

可以只写一个异常处理程序来捕获所有类型的异常。通过捕获异常类型的基类 Exception，就可以做到这一点（事实上还有其他的基类，但 Exception 是所有编程行为相关的基类）：

```
catch(Exception e) {
    System.out.println("Caught an exception");
}
```

这将捕获所有异常，所以最好把它放在处理程序列表的末尾，以防它抢在其他处理程序之前先把异常捕获了。

因为 Exception 是与编程有关的所有异常类的基类，所以它不会含有太多具体的信息，不过可以

调用它从其基类 Throwable 继承的方法:

```
String getMessage()
String getLocalizedMessage()
```

用来获取详细信息，或用本地语言表示的详细信息。

```
String toString()
```

返回对 Throwable 的简单描述，要是有详细信息的话，也会把它包含在内。

```
void printStackTrace()
void printStackTrace(PrintStream)
void printStackTrace(java.io.PrintWriter)
```

打印 Throwable 和 Throwable 的调用栈轨迹。调用栈显示了“把你带到异常抛出地点”的方法调用序列。其中第一个版本输出到标准错误，后两个版本允许选择要输出的流（在附录 I/O 流 中，你将会理解为什么有两种不同的流）。

```
Throwable fillInStackTrace()
```

用于在 Throwable 对象的内部记录栈帧的当前状态。这在程序重新抛出错误或异常（很快就会讲到）时很有用。

此外，也可以使用 Throwable 从其基类 Object（也是所有类的基类）继承的方法。对于异常来说，getClass）也许是个很好用的方法，它将返回一个表示此对象类型的对象。然后可以使用 getName）方法查询这个 Class 对象包含包信息的名称，或者使用只产生类名称的 getSimpleName() 方法。

下面的例子演示了如何使用 Exception 类型的方法:

```
// exceptions/ExceptionMethods.java
// Demonstrating the Exception Methods
public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("My Exception");
        } catch(Exception e) {
            System.out.println("Caught Exception");
            System.out.println(
                    "getMessage():" + e.getMessage());
            System.out.println("getLocalizedMessage():" +
                    e.getLocalizedMessage());
            System.out.println("toString():" + e);
            System.out.println("printStackTrace():");
            e.printStackTrace(System.out);
        }
    }
}
```

输出为:

```
Caught Exception
getMessage():My Exception
getLocalizedMessage():My Exception
toString():java.lang.Exception: My Exception
printStackTrace():
java.lang.Exception: My Exception
at
ExceptionMethods.main(ExceptionMethods.java:7)
```

可以发现每个方法都比前一个提供了更多的信息——实际上它们每一个都是前一个的超集。

16.6.1 多重捕获

如果有一组具有相同基类的异常，你想使用同一方式进行捕获，那你直接 catch 它们的基类型。但是，如果这些异常没有共同的基类型，在 Java 7 之前，你必须为每一个类型编写一个 catch：

```java
// exceptions/SameHandler.java
class EBase1 extends Exception {}
class Except1 extends EBase1 {}
class EBase2 extends Exception {}
class Except2 extends EBase2 {}
class EBase3 extends Exception {}
class Except3 extends EBase3 {}
class EBase4 extends Exception {}
class Except4 extends EBase4 {}

public class SameHandler {
    void x() throws Except1, Except2, Except3, Except4 {}
    void process() {}
    void f() {
        try {
            x();
        } catch(Except1 e) {
            process();
        } catch(Except2 e) {
            process();
        } catch(Except3 e) {
            process();
        } catch(Except4 e) {
            process();
        }
    }
}
```

通过 Java 7 的多重捕获机制，你可以使用“或”将不同类型的异常组合起来，只需要一行 catch 语句：

```java
// exceptions/MultiCatch.java
```

```
public class MultiCatch {
    void x() throws Except1, Except2, Except3, Except4 {}
    void process() {}
    void f() {
        try {
            x();

        } catch(Except1 | Except2 | Except3 | Except4 e) {
            process();
        }
    }
}
```

或者以其他的组合方式:

```
// exceptions/MultiCatch2.java
public class MultiCatch2 {
    void x() throws Except1, Except2, Except3, Except4 {}
    void process1() {}
    void process2() {}
    void f() {
        try {
            x();
        } catch(Except1 | Except2 e) {
            process1();
        } catch(Except3 | Except4 e) {
            process2();
        }
    }
}
```

这对书写更整洁的代码很有帮助。

16.6.2 栈轨迹

printStackTrace() 方法所提供的信息可以通过 getStackTrace() 方法来直接访问，这个方法将返回一个由栈轨迹中的元素所构成的数组，其中每一个元素都表示栈中的一桢。元素 0 是栈顶元素，并且是调用序列中的最后一个方法调用（这个 Throwable 被创建和抛出之处）。数组中的最后一个元素和栈底是调用序列中的第一个方法调用。下面的程序是一个简单的演示示例:

```
// exceptions/WhoCalled.java
// Programmatic access to stack trace information
public class WhoCalled {
    static void f() {
// Generate an exception to fill in the stack trace
        try {
            throw new Exception();
        } catch(Exception e) {
```

```
            for(StackTraceElement ste : e.getStackTrace())
                System.out.println(ste.getMethodName());
        }
    }
    static void g() { f(); }
    static void h() { g(); }
    public static void main(String[] args) {
        f();
        System.out.println("*******");
        g();
        System.out.println("*******");
        h();
    }
}
```

输出为:

```
f
main
*******
f
g
main
*******
f
g
h
main
```

这里，我们只打印了方法名，但实际上还可以打印整个 StackTraceElement，它包含其他附加的信息。

16.6.3 重新抛出异常

有时希望把刚捕获的异常重新抛出，尤其是在使用 Exception 捕获所有异常的时候。既然已经得到了对当前异常对象的引用，可以直接把它重新抛出:

```
catch(Exception e) {
    System.out.println("An exception was thrown");
    throw e;
}
```

重抛异常会把异常抛给上一级环境中的异常处理程序，同一个 try 块的后续 catch 子句将被忽略。此外，异常对象的所有信息都得以保持，所以高一级环境中捕获此异常的处理程序可以从这个异常对象中得到所有信息。

如果只是把当前异常对象重新抛出，那么 printStackTrace() 方法显示的将是原来异常抛出点的调用栈信息，而并非重新抛出点的信息。要想更新这个信息，可以调用 fillInStackTrace() 方法，这将返

回一个 Throwable 对象，它是通过把当前调用栈信息填入原来那个异常对象而建立的，就像这样：

```
// exceptions/Rethrowing.java
// Demonstrating fillInStackTrace()
public class Rethrowing {
    public static void f() throws Exception {
        System.out.println(
                "originating the exception in f()");
        throw new Exception("thrown from f()");
    }
    public static void g() throws Exception {
        try {
            f();
        } catch(Exception e) {
            System.out.println(
                    "Inside g(), e.printStackTrace()");
            e.printStackTrace(System.out);
            throw e;
        }
    }
    public static void h() throws Exception {
        try {
            f();
        } catch(Exception e) {
            System.out.println(
                    "Inside h(), e.printStackTrace()");
            e.printStackTrace(System.out);
            throw (Exception)e.fillInStackTrace();
        }
    }
    public static void main(String[] args) {
        try {
            g();
        } catch(Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch(Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
    }
}
```

输出为：

```
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
```

```
at Rethrowing.f(Rethrowing.java:8)
at Rethrowing.g(Rethrowing.java:12)
at Rethrowing.main(Rethrowing.java:32)
main: printStackTrace()
java.lang.Exception: thrown from f()
at Rethrowing.f(Rethrowing.java:8)
at Rethrowing.g(Rethrowing.java:12)
at Rethrowing.main(Rethrowing.java:32)
originating the exception in f()
Inside h(), e.printStackTrace()
java.lang.Exception: thrown from f()
at Rethrowing.f(Rethrowing.java:8)
at Rethrowing.h(Rethrowing.java:22)
at Rethrowing.main(Rethrowing.java:38)
main: printStackTrace()
java.lang.Exception: thrown from f()
at Rethrowing.h(Rethrowing.java:27)
at Rethrowing.main(Rethrowing.java:38)
```

调用 fillInStackTrace() 的那一行就成了异常的新发生地了。

有可能在捕获异常之后抛出另一种异常。这么做的话，得到的效果类似于使用 filInStackTrace()，有关原来异常发生点的信息会丢失，剩下的是与新的抛出点有关的信息：

```java
// exceptions/RethrowNew.java
// Rethrow a different object from the one you caught
class OneException extends Exception {
    OneException(String s) { super(s); }
}
class TwoException extends Exception {
    TwoException(String s) { super(s); }
}
public class RethrowNew {
    public static void f() throws OneException {
        System.out.println(
                "originating the exception in f()");
        throw new OneException("thrown from f()");
    }
    public static void main(String[] args) {
        try {
            try {
                f();
            } catch(OneException e) {
                System.out.println(
                        "Caught in inner try, e.printStackTrace()");
                e.printStackTrace(System.out);
                throw new TwoException("from inner try");
            }
        } catch(TwoException e) {
            System.out.println(
                    "Caught in outer try, e.printStackTrace()");
```

```
            e.printStackTrace(System.out);
        }
    }
}
```

输出为:

```
originating the exception in f()
Caught in inner try, e.printStackTrace()
OneException: thrown from f()
at RethrowNew.f(RethrowNew.java:16)
at RethrowNew.main(RethrowNew.java:21)
Caught in outer try, e.printStackTrace()
TwoException: from inner try
at RethrowNew.main(RethrowNew.java:26)
```

最后那个异常仅知道自己来自 main()，而对 f() 一无所知。

永远不必为清理前一个异常对象而担心，或者说为异常对象的清理而担心。它们都是用 new 在堆上创建的对象，所以垃圾回收器会自动把它们清理掉。

16.6.4 精准的重新抛出异常

在 Java 7 之前，如果遇到异常，则只能重新抛出该类型的异常。这导致在 Java 7 中修复的代码不精确。所以在 Java 7 之前，这无法编译:

```
class BaseException extends Exception {}
class DerivedException extends BaseException {}

public class PreciseRethrow {
    void catcher() throws DerivedException {
        try {
            throw new DerivedException();
        } catch(BaseException e) {
            throw e;
        }
    }
}
```

因为 catch 捕获了一个 BaseException，编译器强迫你声明 catcher() 抛出 BaseException，即使它实际上抛出了更具体的 DerivedException。从 Java 7 开始，这段代码就可以编译，这是一个很小但很有用的修复。

16.6.5 异常链

常常会想要在捕获一个异常后抛出另一个异常，并且希望把原始异常的信息保存下来，这被称为异常链。在 JDK1.4 以前，程序员必须自己编写代码来保存原始异常的信息。现在所有 Throwable 的子

类在构造器中都可以接受一个 cause（因由）对象作为参数。这个 cause 就用来表示原始异常，这样通过把原始异常传递给新的异常，使得即使在当前位置创建并抛出了新的异常，也能通过这个异常链追踪到异常最初发生的位置。

有趣的是，在 Throwable 的子类中，只有三种基本的异常类提供了带 cause 参数的构造器。它们是 Error（用于 Java 虚拟机报告系统错误）、Exception 以及 RuntimeException。如果要把其他类型的异常链接起来，应该使用 initCause0 方法而不是构造器。

下面的例子能让你在运行时动态地向 DymamicFields 对象添加字段：

```
// exceptions/DynamicFields.java
// A Class that dynamically adds fields to itself to
// demonstrate exception chaining
class DynamicFieldsException extends Exception {}
public class DynamicFields {
    private Object[][] fields;
    public DynamicFields(int initialSize) {
        fields = new Object[initialSize][2];
        for(int i = 0; i < initialSize; i++)
            fields[i] = new Object[] { null, null };
    }
    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(Object[] obj : fields) {
            result.append(obj[0]);
            result.append(": ");
            result.append(obj[1]);
            result.append("\n");
        }
        return result.toString();
    }
    private int hasField(String id) {
        for(int i = 0; i < fields.length; i++)
            if(id.equals(fields[i][0]))
                return i;
        return -1;
    }
    private int getFieldNumber(String id)
            throws NoSuchFieldException {
        int fieldNum = hasField(id);
        if(fieldNum == -1)
            throw new NoSuchFieldException();
        return fieldNum;
    }
    private int makeField(String id) {
        for(int i = 0; i < fields.length; i++)
            if(fields[i][0] == null) {
                fields[i][0] = id;
                return i;
```

```
            }
// No empty fields. Add one:
        Object[][] tmp = new Object[fields.length + 1][2];
        for(int i = 0; i < fields.length; i++)
            tmp[i] = fields[i];
        for(int i = fields.length; i < tmp.length; i++)
            tmp[i] = new Object[] { null, null };
        fields = tmp;
// Recursive call with expanded fields:
        return makeField(id);
    }
    public Object
    getField(String id) throws NoSuchFieldException {
        return fields[getFieldNumber(id)][1];
    }
    public Object setField(String id, Object value)
            throws DynamicFieldsException {
        if(value == null) {
// Most exceptions don't have a "cause"
// constructor. In these cases you must use
// initCause(), available in all
// Throwable subclasses.
            DynamicFieldsException dfe =
                    new DynamicFieldsException();
            dfe.initCause(new NullPointerException());
            throw dfe;
        }
        int fieldNumber = hasField(id);
        if(fieldNumber == -1)
            fieldNumber = makeField(id);
        Object result = null;
        try {
            result = getField(id); // Get old value
        } catch(NoSuchFieldException e) {
// Use constructor that takes "cause":
            throw new RuntimeException(e);
        }
        fields[fieldNumber][1] = value;
        return result;
    }
    public static void main(String[] args) {
        DynamicFields df = new DynamicFields(3);
        System.out.println(df);
        try {
            df.setField("d", "A value for d");
            df.setField("number", 47);
            df.setField("number2", 48);
            System.out.println(df);
            df.setField("d", "A new value for d");
            df.setField("number3", 11);
            System.out.println("df: " + df);
```

```
            System.out.println("df.getField(\"d\") : "
                    + df.getField("d"));
            Object field =
                    df.setField("d", null); // Exception
        } catch(NoSuchFieldException |
                DynamicFieldsException e) {
            e.printStackTrace(System.out);
        }
    }
}
```

输出为:

```
null: null
null: null
null: null
d: A value for d
number: 47
number2: 48
df: d: A new value for d
number: 47
number2: 48
number3: 11
df.getField("d") : A new value for d
DynamicFieldsException
at
DynamicFields.setField(DynamicFields.java:65)
at DynamicFields.main(DynamicFields.java:97)
Caused by: java.lang.NullPointerException
at
DynamicFields.setField(DynamicFields.java:67)
... 1 more
```

每个 DynamicFields 对象都含有一个数组，其元素是“成对的对象”。第一个对象表示字段标识符（一个字符串），第二个表示字段值，值的类型可以是除基本类型外的任意类型。当创建对象的时候，要合理估计一下需要多少字段。当调用 setField() 方法的时候，它将试图通过标识修改已有字段值，否则就建一个新的字段，并把值放入。如果空间不够了，将建立一个更长的数组，并把原来数组的元素复制进去。如果你试图为字段设置一个空值，将抛出一个 DynamicFieldsException 异常，它是通过使用 initCause() 方法把 NullPointerException 对象插入而建立的。

至于返回值，setField() 将用 getField() 方法把此位置的旧值取出，这个操作可能会抛出 NoSuchFieldException 异常。如果客户端程序员调用了 getField() 方法，那么他就有责任处理这个可能抛出的 NoSuchFieldException 异常，但如果异常是从 setField0 方法里抛出的，这种情况将被视为编程错误，所以就使用接受 cause 参数的构造器把 NoSuchFieldException 异常转换为 RuntimeException 异常。

你会注意到，toString0 方法使用了一个 StringBuilder 来创建其结果。在 字符串 这章中你将会

不可能出现这种情况的。

了解到更多的关于 StringBuilder 的知识，但是只要你编写设计循环的 toString() 方法，通常都会想使用它，就像本例一样。

主方法中的 catch 子句看起来不同 - 它使用相同的子句处理两种不同类型的异常，并结合“或（|）”符号。此 Java 7 功能有助于减少代码重复，并使你更容易指定要捕获的确切类型，而不是简单地捕获基本类型。你可以通过这种方式组合多种异常类型。

16.7 Java 标准异常

Throwable 这个 Java 类被用来表示任何可以作为异常被抛出的类。Throwable 对象可分为两种类型（指从 Throwable 继承而得到的类型）：Error 用来表示编译时和系统错误（除特殊情况外，一般不用你关心）；Exception 是可以被抛出的基本类型，在 Java 类库、用户方法以及运行时故障中都可能抛出 Exception 型异常。所以 Java 程序员关心的基类型通常是 Exception。要想对异常有全面的了解，最好去浏览一下 HTML 格式的 Java 文档（可以从 java.sun.com 下载）。为了对不同的异常有个感性的认识，这么做是值得的。但很快你就会发现，这些异常除了名称外其实都差不多。同时，Java 中异常的数目在持续增加，所以在书中简单罗列它们毫无意义。所使用的第三方类库也可能会有自己的异常。对异常来说，关键是理解概念以及如何使用。

异常的基本的概念是用名称代表发生的问题，并且异常的名称应该可以望文知意。异常并非全是在 java.lang 包里定义的；有些异常是用来支持其他像 util、net 和 io 这样的程序包，这些异常可以通过它们的完整名称或者从它们的父类中看出端倪。比如，所有的输入/输出异常都是从 java.io.IOException 继承而来的。

16.7.1 特例：RuntimeException

在本章的第一个例子中：

```
if(t == null)
    throw new NullPointerException();
```

如果必须对传递给方法的每个引用都检查其是否为 null（因为无法确定调用者是否传入了非法引用），这听起来着实吓人。幸运的是，这不必由你亲自来做，它属于 Java 的标准运行时检测的一部分。如果对 null 引用进行调用，Java 会自动抛出 NullPointerException 异常，所以上述代码是多余的，尽管你也许想要执行其他的检查以确保 NullPointerException 不会出现。

属于运行时异常的类型有很多，它们会自动被 java 虚拟机抛出，所以不必在异常说明中把它们列出来。这些异常都是从 RuntimeException 类继承而来，所以既体现了继承的优点，使用起来也很方便。这构成了一组具有相同特征和行为的异常类型。并且，也不再需要在异常说明中声明方法将抛出 RuntimeException 类型的异常（或者任何从 RuntimeException 继承的异常），它们也被称为“不受检查异常”。这种异常属于错误，将被自动捕获，就不用你亲自动手了。要是自己去检查 RuntimeException

的话，代码就显得太混乱了。不过尽管通常不用捕获 RuntimeException 异常，但还是可以在代码中抛出 RuntimeException 类型的异常。

RuntimeException 代表的是编程错误：

1. 无法预料的错误。比如从你控制范围之外传递进来的 null 引用。
2. 作为程序员，应该在代码中进行检查的错误。（比如对于 ArrayIndexOutOfBoundsException，就得注意一下数组的大小了。）在一个地方发生的异常，常常会在另一个地方导致错误。

在这些情况下使用异常很有好处，它们能给调试带来便利。

如果不捕获这种类型的异常会发生什么事呢？因为编译器没有在这个问题上对异常说明进行强制检查，RuntimeException 类型的异常也许会穿越所有的执行路径直达 main() 方法，而不会被捕获。要明白到底发生了什么，可以试试下面的例子：

```
// exceptions/NeverCaught.java
// Ignoring RuntimeExceptions
// {ThrowsException}
public class NeverCaught {
    static void f() {
        throw new RuntimeException("From f()");
    }
    static void g() {
        f();
    }
    public static void main(String[] args) {
        g();
    }
}
```

输出结果为：

```
___[ Error Output ]___
Exception in thread "main" java.lang.RuntimeException:
From f()
at NeverCaught.f(NeverCaught.java:7)
at NeverCaught.g(NeverCaught.java:10)
at NeverCaught.main(NeverCaught.java:13)
```

如果 RuntimeException 没有被捕获而直达 main(), 那么在程序退出前将调用异常的 printStackTrace() 方法。

你会发现，RuntimeException（或任何从它继承的异常）是一个特例。对于这种异常类型，编译器不需要异常说明，其输出被报告给了 System.err。

请务必记住：只能在代码中忽略 RuntimeException（及其子类）类型的异常，因为所有受检查类型异常的处理都是由编译器强制实施的。

值得注意的是：不应把 Java 的异常处理机制当成是单一用途的工具。是的，它被设计用来处理一些烦人的运行时错误，这些错误往往是由代码控制能力之外的因素导致的；然而，它对于发现某些编译

不要放过任何一个看上去很简单的小问题 hfill —他们往往并不那么简单，或者可以引伸出很多知识点；不会举一反三你就永远学不会。

器无法检测到的编程错误，也是非常重要的。

16.8 使用 finally 进行清理

有一些代码片段，可能会希望无论 try 块中的异常是否抛出，它们都能得到执行。这通常适用于内存回收之外的情况（因为回收由垃圾回收器完成），为了达到这个效果，可以在异常处理程序后面加上 finally 子句。完整的异常处理程序看起来像这样：

```
try {
// The guarded region: Dangerous activities
// that might throw A, B, or C
} catch(A a1) {
// Handler for situation A
} catch(B b1) {
// Handler for situation B
} catch(C c1) {
// Handler for situation C
} finally {
// Activities that happen every time
}
```

为了证明 finally 子句总能运行，可以试试下面这个程序：

```
// exceptions/FinallyWorks.java
// The finally clause is always executed
class ThreeException extends Exception {}
public class FinallyWorks {
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                // Post-increment is zero first time:
                if(count++ == 0)
                    throw new ThreeException();
                System.out.println("No exception");
            } catch(ThreeException e) {
                System.out.println("ThreeException");
            } finally {
                System.out.println("In finally clause");
                if(count == 2) break; // out of "while"
            }
        }
    }
}
```

输出为：

```
ThreeException
```

不要看到别人的回复第一句话就说：给个代码吧！你应该想想为什么。当你自己想出来再参考别人的提示，你就知道自己和别人思路的差异。

```
In finally clause
No exception
In finally clause
```

可以从输出中发现，无论异常是否被抛出，finally 子句总能被执行。这个程序也给了我们一些思路，当 Java 中的异常不允许我们回到异常抛出的地点时，那么该如何应对呢？如果把 try 块放在循环里，就建立了一个“程序继续执行之前必须要达到”的条件。还可以加入一个 static 类型的计数器或者别的装置，使循环在放弃以前能尝试一定的次数。这将使程序的健壮性更上一个台阶。

16.8.1 finally 用来做什么？

对于没有垃圾回收和析构函数自动调用机制的语言来说，finally 非常重要。它能使程序员保证：无论 try 块里发生了什么，内存总能得到释放。但 Java 有垃圾回收机制，所以内存释放不再是问题。而且，Java 也没有析构函数可供调用。那么，Java 在什么情况下才能用到 finally 呢？

当要把除内存之外的资源恢复到它们的初始状态时，就要用到 finally 子句。这种需要清理的资源包括：已经打开的文件或网络连接，在屏幕上画的图形，甚至可以是外部世界的某个开关，如下面例子所示：

```java
// exceptions/Switch.java
public class Switch {
    private boolean state = false;
    public boolean read() { return state; }
    public void on() {
        state = true;
        System.out.println(this);
    }
    public void off() {
        state = false;
        System.out.println(this);
    }
    @Override
    public String toString() {
        return state ? "on" : "off";
    }
}
// exceptions/OnOffException1.java
public class OnOffException1 extends Exception {}
// exceptions/OnOffException2.java
public class OnOffException2 extends Exception {}
// exceptions/OnOffSwitch.java
// Why use finally?
public class OnOffSwitch {
    private static Switch sw = new Switch();
    public static void f()
            throws OnOffException1, OnOffException2 {}
    public static void main(String[] args) {
```

```
        try {
            sw.on();
            // Code that can throw exceptions...
            f();
            sw.off();
        } catch(OnOffException1 e) {
            System.out.println("OnOffException1");
            sw.off();
        } catch(OnOffException2 e) {
            System.out.println("OnOffException2");
            sw.off();
        }
    }
}
```

输出为:

```
on
off
```

程序的目的是要确保 main() 结束的时候开关必须是关闭的，所以在每个 try 块和异常处理程序的末尾都加入了对 sw.offo 方法的调用。但也可能有这种情况：异常被抛出，但没被处理程序捕获，这时 sw.off() 就得不到调用。但是有了 finally，只要把 try 块中的清理代码移放在一处即可:

```
// exceptions/WithFinally.java
// Finally Guarantees cleanup
public class WithFinally {
    static Switch sw = new Switch();
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            OnOffSwitch.f();
        } catch(OnOffException1 e) {
            System.out.println("OnOffException1");
        } catch(OnOffException2 e) {
            System.out.println("OnOffException2");
        } finally {
            sw.off();
        }
    }
}
```

输出为:

```
on
off
```

这里 sw.off() 被移到一处，并且保证在任何情况下都能得到执行。

甚至在异常没有被当前的异常处理程序捕获的情况下，异常处理机制也会在跳到更高一层的异常

不用担心，这次肯定不会有问题了。

处理程序之前，执行 finally 子句：

```
// exceptions/AlwaysFinally.java
// Finally is always executed
class FourException extends Exception {}
public class AlwaysFinally {
    public static void main(String[] args) {
        System.out.println("Entering first try block");
        try {
            System.out.println("Entering second try block");
            try {
                throw new FourException();
            } finally {
                System.out.println("finally in 2nd try block");
            }
        } catch(FourException e) {
            System.out.println(
                    "Caught FourException in 1st try block");
        } finally {
            System.out.println("finally in 1st try block");
        }
    }
}
```

输出为：

```
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
```

当涉及 break 和 continue 语句的时候，finally 子句也会得到执行。请注意，如果把 finally 子句和带标签的 break 及 continue 配合使用，在 Java 里就没必要使用 goto 语句了。

16.8.2 在 return 中使用 finally

因为 finally 子句总是会执行，所以可以从一个方法内的多个点返回，仍然能保证重要的清理工作会执行：

```
// exceptions/MultipleReturns.java
public class MultipleReturns {
    public static void f(int i) {
        System.out.println(
                "Initialization that requires cleanup");
        try {
            System.out.println("Point 1");
            if(i == 1) return;
            System.out.println("Point 2");
```

```
            if(i == 2) return;
            System.out.println("Point 3");
            if(i == 3) return;
            System.out.println("End");
            return;
        } finally {
            System.out.println("Performing cleanup");
        }
    }
    public static void main(String[] args) {
        for(int i = 1; i <= 4; i++)
            f(i);
    }
}
```

输出为:

```
Initialization that requires cleanup
Point 1
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
End
Performing cleanup
```

从输出中可以看出，从何处返回无关紧要，finally 子句永远会执行。

16.8.3 缺憾：异常丢失

遗憾的是，Java 的异常实现也有瑕疵。异常作为程序出错的标志，决不应该被忽略，但它还是有可能被轻易地忽略。用某些特殊的方式使用 finally 子句，就会发生这种情况:

```
// exceptions/LostMessage.java
// How an exception can be lost
class VeryImportantException extends Exception {
    @Override
    public String toString() {
        return "A very important exception!";
```

```
    }
}
class HoHumException extends Exception {
    @Override
    public String toString() {
        return "A trivial exception";
    }
}
public class LostMessage {
    void f() throws VeryImportantException {
        throw new VeryImportantException();
    }
    void dispose() throws HoHumException {
        throw new HoHumException();
    }
    public static void main(String[] args) {
        try {
            LostMessage lm = new LostMessage();
            try {
                lm.f();
            } finally {
                lm.dispose();
            }
        } catch(VeryImportantException | HoHumException e) {
            System.out.println(e);
        }
    }
}
```

输出为:

```
A trivial exception
```

从输出中可以看到, VeryImportantException 不见了, 它被 finally 子句里的 HoHumException 所取代。这是相当严重的缺陷，因为异常可能会以一种比前面例子所示更微妙和难以察觉的方式完全丢失。相比之下，C++ 把“前一个异常还没处理就抛出下一个异常”的情形看成是糟糕的编程错误。也许在 Java 的未来版本中会修正这个问题（另一方面，要把所有抛出异常的方法，如上例中的 dispose() 方法，全部打包放到 try-catch 子句里面）。

一种更加简单的丢失异常的方式是从 finally 子句中返回:

```
// exceptions/ExceptionSilencer.java
public class ExceptionSilencer {
    public static void main(String[] args) {
        try {
            throw new RuntimeException();
        } finally {
            // Using 'return' inside the finally block
            // will silence any thrown exception.
            return;
```

```
        }
    }
}
```

如果运行这个程序，就会看到即使方法里抛出了异常，它也不会产生任何输出。

16.9 异常限制

当覆盖方法的时候，只能抛出在基类方法的异常说明里列出的那些异常。这个限制很有用，因为这意味着，若当基类使用的代码应用到其派生类对象的时候，一样能够工作（当然，这是面向对象的基本概念），异常也不例外。

下面例子演示了这种（在编译时）施加在异常上面的限制：

```
// exceptions/StormyInning.java
// Overridden methods can throw only the exceptions
// specified in their base-class versions, or exceptions
// derived from the base-class exceptions
class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}
abstract class Inning {
    Inning() throws BaseballException {}
    public void event() throws BaseballException {
// Doesn't actually have to throw anything
    }
    public abstract void atBat() throws Strike, Foul;
    public void walk() {} // Throws no checked exceptions
}
class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}
interface Storm {
    void event() throws RainedOut;
    void rainHard() throws RainedOut;
}
public class StormyInning extends Inning implements Storm {
    // OK to add new exceptions for constructors, but you
// must deal with the base constructor exceptions:
    public StormyInning()
            throws RainedOut, BaseballException {}
    public StormyInning(String s)
            throws BaseballException {}
    // Regular methods must conform to base class:
//- void walk() throws PopFoul {} //Compile error
// Interface CANNOT add exceptions to existing
// methods from the base class:
//- public void event() throws RainedOut {}
```

```
// If the method doesn't already exist in the
// base class, the exception is OK:
    @Override
    public void rainHard() throws RainedOut {}
    // You can choose to not throw any exceptions,
// even if the base version does:
    @Override
    public void event() {}
    // Overridden methods can throw inherited exceptions:
    @Override
    public void atBat() throws PopFoul {}
    public static void main(String[] args) {
        try {
            StormyInning si = new StormyInning();
            si.atBat();
        } catch(PopFoul e) {
            System.out.println("Pop foul");
        } catch(RainedOut e) {
            System.out.println("Rained out");
        } catch(BaseballException e) {
            System.out.println("Generic baseball exception");
        }
// Strike not thrown in derived version.
        try {
// What happens if you upcast?
            Inning i = new StormyInning();
            i.atBat();
// You must catch the exceptions from the
// base-class version of the method:
        } catch(Strike e) {
            System.out.println("Strike");
        } catch(Foul e) {
            System.out.println("Foul");
        } catch(RainedOut e) {
            System.out.println("Rained out");
        } catch(BaseballException e) {
            System.out.println("Generic baseball exception");
        }
    }
}
```

在 Inning 类中，可以看到构造器和 event() 方法都声明将抛出异常，但实际上没有抛出。这种方式使你能强制用户去捕获可能在覆盖后的 event() 版本中增加的异常，所以它很合理。这对于抽象方法同样成立，比如 atBat()。

接口 Storm 包含了一个在 Inning 中定义的方法 event() 和一个不在 Inning 中定义的方法 rainHard()。这两个方法都抛出新的异常 RainedOut，如果 StormyInning 类在扩展 Inning 类的同时又实现了 Storm 接口，那么 Storm 里的 event() 方法就不能改变在 Inning 中的 event（方法的异常接口。否则的话，在使用基类的时候就不能判断是否捕获了正确的异常，所以这也很合理。当然，如果接

口里定义的方法不是来自于基类，比如 rainHard()，那么此方法抛出什么样的异常都没有问题。

异常限制对构造器不起作用。你会发现 StormyInning 的构造器可以抛出任何异常，而不必理会基类构造器所抛出的异常。然而，因为基类构造器必须以这样或那样的方式被调用（这里默认构造器将自动被调用），派生类构造器的异常说明必须包含基类构造器的异常说明。

派生类构造器不能捕获基类构造器抛出的异常。

StormyInning.walk() 不能通过编译是因为它抛出了异常，而 Inning.walk() 并没有声明此异常。如果编译器允许这么做的话，就可以在调用 Inning.walk() 的时候不用做异常处理了，而且当把它替换成 Inning 的派生类的对象时，这个方法就有可能会抛出异常，于是程序就失灵了。通过强制派生类遵守基类方法的异常说明，对象的可替换性得到了保证。

覆盖后的 event() 方法表明，派生类方法可以不抛出任何异常，即使它是基类所定义的异常。同样这是因为，假使基类的方法会抛出异常，这样做也不会破坏已有的程序，所以也没有问题。类似的情况出现在 atBat() 身上，它抛出的是 PopFoul，这个异常是继承自“会被基类的 atBat() 抛出”的 Foul，这样，如果你写的代码是同 Inning 打交道，并且调用了它的 atBat() 的话，那么肯定能捕获 Foul，而 PopFoul 是由 Foul 派生出来的，因此异常处理程序也能捕获 PopFoul。

最后一个值得注意的地方是 main()。这里可以看到，如果处理的刚好是 Stormylnning 对象的话，编译器只会强制要求你捕获这个类所抛出的异常。但是如果将它向上转型成基类型，那么编译器就会（正确地）要求你捕获基类的异常。所有这些限制都是为了能产生更为强壮的异常处理代码。

尽管在继承过程中，编译器会对异常说明做强制要求，但异常说明本身并不属于方法类型的一部分，方法类型是由方法的名字与参数的类型组成的。因此，不能基于异常说明来重载方法。此外，一个出现在基类方法的异常说明中的异常，不一定会出现在派生类方法的异常说明里。这点同继承的规则明显不同，在继承中，基类的方法必须出现在派生类里，换句话说，在继承和覆盖的过程中，某个特定方法的“异常说明的接口”不是变大了而是变小了——这恰好和类接口在继承时的情形相反。

16.10 构造器

有一点很重要，即你要时刻询问自己“如果异常发生了，所有东西能被正确的清理吗？”尽管大多数情况下是非常安全的，但涉及构造器时，问题就出现了。构造器会把对象设置成安全的初始状态，但还会有别的动作，比如打开一个文件，这样的动作只有在对象使用完毕并且用户调用了特殊的清理方法之后才能得以清理。如果在构造器内抛出了异常，这些清理行为也许就不能正常工作了。这意味着在编写构造器时要格外细心。

你也许会认为使用 finally 就可以解决问题。但问题并非如此简单，因为 finally 会每次都执行清理代码。如果构造器在其执行过程中半途而废，也许该对象的某些部分还没有被成功创建，而这些部分在 finaly 子句中却是要被清理的。

在下面的例子中，建立了一个 InputFile 类，它能打开一个文件并且每次读取其中的一行。这里使用了 Java 标准输入/输出库中的 FileReader 和 BufferedReader 类（将在 附录：I/O 流 中讨论），这

程序能用，不过还没有测试。

些类的基本用法很简单，你应该很容易明白：

```java
// exceptions/InputFile.java
// Paying attention to exceptions in constructors
import java.io.*;
public class InputFile {
    private BufferedReader in;
    public InputFile(String fname) throws Exception {
        try {
            in = new BufferedReader(new FileReader(fname));
            // Other code that might throw exceptions
        } catch(FileNotFoundException e) {
            System.out.println("Could not open " + fname);
            // Wasn't open, so don't close it
            throw e;
        } catch(Exception e) {
            // All other exceptions must close it
            try {
                in.close();
            } catch(IOException e2) {
                System.out.println("in.close() unsuccessful");
            }
            throw e; // Rethrow
        } finally {
        // Don't close it here!!!
        }
    }
    public String getLine() {
        String s;
        try {
            s = in.readLine();
        } catch(IOException e) {
            throw new RuntimeException("readLine() failed");
        }
        return s;
    }
    public void dispose() {
        try {
            in.close();
            System.out.println("dispose() successful");
        } catch(IOException e2) {
            throw new RuntimeException("in.close() failed");
        }
    }
}
```

InputFile 的构造器接受字符串作为参数，该字符串表示所要打开的文件名。在 try 块中，会使用此文件名建立 FileReader 对象。FileReader 对象本身用处并不大，但可以用它来建立 BufferedReader 对象。注意，使用 InputFile 的好处之一是把两步操作合而为一。

如果 FileReader 的构造器失败了，将抛出 FileNotFoundException 异常。对于这个异常，并不需要关闭文件，因为这个文件还没有被打开。而任何其他捕获异常的 catch 子句必须关闭文件，因为在它们捕获到异常之时，文件已经打开了（当然，如果还有其他方法能抛出 FileNotFoundException，这个方法就显得有些投机取巧了。这时，通常必须把这些方法分别放到各自的 try 块里），close() 方法也可能会抛出异常，所以尽管它已经在另一个 catch 子句块里了，还是要再用一层 try-catch，这对 Java 编译器而言只不过是多了一对花括号。在本地做完处理之后，异常被重新抛出，对于构造器而言这么做是很合适的，因为你总不希望去误导调用方，让他认为“这个对象已经创建完毕，可以使用了”。

在本例中，由于 finally 会在每次完成构造器之后都执行一遍，因此它实在不该是调用 close() 关闭文件的地方。我们希望文件在 InputFlle 对象的整个生命周期内都处于打开状态。

getLine() 方法会返回表示文件下一行内容的字符串。它调用了能抛出异常的 readLine()，但是这个异常已经在方法内得到处理，因此 getLine() 不会抛出任何异常。在设计异常时有一个问题：应该把异常全部放在这一层处理；还是先处理一部分，然后再向上层抛出相同的（或新的）异常；又或者是不做任何处理直接向上层抛出。如果用法恰当的话，直接向上层抛出的确能简化编程。在这里，getLine() 方法将异常转换为 RuntimeException，表示一个编程错误。

用户在不再需要 InputFile 对象时，就必须调用 dispose() 方法，这将释放 BufferedReader 和/或 FileReader 对象所占用的系统资源（比如文件句柄），在使用完 InputFile 对象之前是不会调用它的。可能你会考虑把上述功能放到 finalize() 里面，但我在 封装 讲过，你不知道 finalize() 会不会被调用（即使能确定它将被调用，也不知道在什么时候调用），这也是 Java 的缺陷：除了内存的清理之外，所有的清理都不会自动发生。所以必须告诉客户端程序员，这是他们的责任。

对于在构造阶段可能会抛出异常，并且要求清理的类，最安全的使用方式是使用嵌套的 try 子句：

```java
// exceptions/Cleanup.java
// Guaranteeing proper cleanup of a resource
public class Cleanup {
    public static void main(String[] args) {
        try {
            InputFile in = new InputFile("Cleanup.java");
            try {
                String s;
                int i = 1;
                while((s = in.getLine()) != null)
                    ; // Perform line-by-line processing here...
            } catch(Exception e) {
                System.out.println("Caught Exception in main");
                e.printStackTrace(System.out);
            } finally {
                in.dispose();
            }
        } catch(Exception e) {
            System.out.println(
                    "InputFile construction failed");
        }
```

```
    }
}
```

输出为:

```
dispose() successful
```

请仔细观察这里的逻辑：对 InputFile 对象的构造在其自己的 try 语句块中有效，如果构造失败，将进入外部的 catch 子句，而 dispose() 方法不会被调用。但是，如果构造成功，我们肯定想确保对象能够被清理，因此在构造之后立即创建了一个新的 try 语句块。执行清理的 finally 与内部的 try 语句块相关联。在这种方式中，finally 子句在构造失败时是不会执行的，而在构造成功时将总是执行。

这种通用的清理惯用法在构造器不抛出任何异常时也应该运用，其基本规则是：在创建需要清理的对象之后，立即进入一个 try-finally 语句块:

```java
// exceptions/CleanupIdiom.java
// Disposable objects must be followed by a try-finally
class NeedsCleanup { // Construction can't fail
    private static long counter = 1;
    private final long id = counter++;
    public void dispose() {
        System.out.println(
                "NeedsCleanup " + id + " disposed");
    }
}
class ConstructionException extends Exception {}
class NeedsCleanup2 extends NeedsCleanup {
    // Construction can fail:
    NeedsCleanup2() throws ConstructionException {}
}
public class CleanupIdiom {
    public static void main(String[] args) {
        // [1]:
        NeedsCleanup nc1 = new NeedsCleanup();
        try {
        // ...
        } finally {
            nc1.dispose();
        }
        // [2]:
        // If construction cannot fail,
        // you can group objects:
        NeedsCleanup nc2 = new NeedsCleanup();
        NeedsCleanup nc3 = new NeedsCleanup();
        try {
        // ...
        } finally {
            nc3.dispose(); // Reverse order of construction
            nc2.dispose();
        }
```

```
        // [3]:
        // If construction can fail you must guard each one:
        try {
            NeedsCleanup2 nc4 = new NeedsCleanup2();
            try {
                NeedsCleanup2 nc5 = new NeedsCleanup2();
                try {
                // ...
                } finally {
                    nc5.dispose();
                }
            } catch(ConstructionException e) { // nc5 const.
                System.out.println(e);
            } finally {
                nc4.dispose();
            }
        } catch(ConstructionException e) { // nc4 const.
            System.out.println(e);
        }
    }
}
```

输出为:

```
NeedsCleanup 1 disposed
NeedsCleanup 3 disposed
NeedsCleanup 2 disposed
NeedsCleanup 5 disposed
NeedsCleanup 4 disposed
```

- [1] 相当简单，遵循了在可去除对象之后紧跟 try-finally 的原则。如果对象构造不会失败，就不需要任何 catch。
- [2] 为了构造和清理，可以看到将具有不能失败的构造器的对象分组在一起。
- [3] 展示了如何处理那些具有可以失败的构造器，且需要清理的对象。为了正确处理这种情况，事情变得很棘手，因为对于每一个构造，都必须包含在其自己的 try-finally 语句块中，并且每一个对象构造必须都跟随一个 try-finally 语句块以确保清理。

本例中的异常处理的棘手程度，对于应该创建不能失败的构造器是一个有力的论据，尽管这么做并非总是可行。

注意，如果 dispose() 可以抛出异常，那么你可能需要额外的 try 语句块。基本上，你应该仔细考虑所有的可能性，并确保正确处理每一种情况。

16.11 Try-With-Resources 用法

上一节的内容可能让你有些头疼。在考虑所有可能失败的方法时，找出放置所有 try-catch-finally 块的位置变得令人生畏。确保没有任何故障路径，使系统远离不稳定状态，这非常具有挑战性。

InputFile.java 是一个特别棘手的情况，因为文件被打开（包含所有可能的异常），然后它在对象的生命周期中保持打开状态。每次调用 getLine() 都会导致异常，因此可以调用 dispose() 方法。这是一个很好的例子，因为它显示了事物的混乱程度。它还表明你应该尝试最好不要那样设计代码（当然，你经常会遇到这种你无法选择的代码设计的情况，因此你必须仍然理解它）。

InputFile.java 一个更好的实现方式是如果构造函数读取文件并在内部缓冲它 —— 这样，文件的打开，读取和关闭都发生在构造函数中。或者，如果读取和存储文件不切实际，你可以改为生成 Stream。理想情况下，你可以设计成如下的样子：

```
// exceptions/InputFile2.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;
public class InputFile2 {
    private String fname;

    public InputFile2(String fname) {
        this.fname = fname;
    }

    public Stream<String> getLines() throws IOException {
        return Files.lines(Paths.get(fname));
    }

    public static void
    main(String[] args) throws IOException {
        new InputFile2("InputFile2.java").getLines()
                .skip(15)
                .limit(1)
                .forEach(System.out::println);
    }
}
```

输出为：

```
main(String[] args) throws IOException {
```

现在，getLines() 全权负责打开文件并创建 Stream。

你不能总是轻易地回避这个问题。有时会有以下问题：

1. 需要资源清理
2. 需要在特定的时刻进行资源清理，比如你离开作用域的时候（在通常情况下意味着通过异常进行清理）。

程序员中的前辈在谈到一个人学写程序有没有前途时，总是会说：“学程序不光要能吃苦能用功，还得看有没有 sense，没 sense 的人，就是再怎么学，再怎么给他讲效果都不好。”

一个常见的例子是 jav.io.FileInputstream（将会在 附录：I/O 流 中提到）。要正确使用它，你必须编写一些棘手的样板代码：

```
// exceptions/MessyExceptions.java
import java.io.*;
public class MessyExceptions {
    public static void main(String[] args) {
        InputStream in = null;
        try {
            in = new FileInputStream(
                    new File("MessyExceptions.java"));
            int contents = in.read();
            // Process contents
        } catch(IOException e) {
            // Handle the error
        } finally {
            if(in != null) {
                try {
                    in.close();
                } catch(IOException e) {
                    // Handle the close() error
                }
            }
        }
    }
}
```

当 finally 子句有自己的 try 块时，感觉事情变得过于复杂。

幸运的是，Java 7 引入了 try-with-resources 语法，它可以非常清楚地简化上面的代码：

```
// exceptions/TryWithResources.java
import java.io.*;
public class TryWithResources {
    public static void main(String[] args) {
        try(
                InputStream in = new FileInputStream(
                        new File("TryWithResources.java"))
        ) {
            int contents = in.read();
            // Process contents
        } catch(IOException e) {
            // Handle the error
        }
    }
}
```

在 Java 7 之前，try 总是后面跟着一个 {，但是现在可以跟一个带括号的定义 - 这里是我们创建的 FileInputStream 对象。括号内的部分称为资源规范头（resource specification header）。现在可用于整个 try 块的其余部分。更重要的是，无论你如何退出 try 块（正常或异常），都会执行前一个 finally

子句的等价物，但不会编写那些杂乱而棘手的代码。这是一项重要的改进。

它是如何工作的？在 try-with-resources 定义子句中创建的对象（在括号内）必须实现 java.lang.AutoCloseable 接口，这个接口有一个方法：close()。当在 Java 7 中引入 AutoCloseable 时，许多接口和类被修改以实现它；查看 Javadocs 中的 AutoCloseable，可以找到所有实现该接口的类列表，其中包括 Stream 对象：

```
// exceptions/StreamsAreAutoCloseable.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;
public class StreamsAreAutoCloseable {
    public static void
    main(String[] args) throws IOException{
        try(
                Stream<String> in = Files.lines(
                        Paths.get("StreamsAreAutoCloseable.java"));
                PrintWriter outfile = new PrintWriter(
                        "Results.txt"); // [1]
        ) {
            in.skip(5)
                    .limit(1)
                    .map(String::toLowerCase)
                    .forEachOrdered(outfile::println);
        } // [2]
    }
}
```

- [1] 你在这里可以看到其他的特性：资源规范头中可以包含多个定义，并且通过分号进行分割（最后一个分号是可选的）。规范头中定义的每个对象都会在 try 语句块运行结束之后调用 close() 方法。
- [2] try-with-resources 里面的 try 语句块可以不包含 catch 或者 finally 语句而独立存在。在这里，IOException 被 main() 方法抛出，所以这里并不需要在 try 后面跟着一个 catch 语句块。

Java 5 中的 Closeable 已经被修改，修改之后的接口继承了 AutoCloseable 接口。所以所有实现了 Closeable 接口的对象，都支持了 try-with-resources 特性。

16.11.1 揭示细节

为了研究 try-with-resources 的基本机制，我们将创建自己的 AutoCloseable 类：

```
// exceptions/AutoCloseableDetails.java
class Reporter implements AutoCloseable {
    String name = getClass().getSimpleName();
    Reporter() {
        System.out.println("Creating " + name);
    }
```

```
    public void close() {
        System.out.println("Closing " + name);
    }
}
class First extends Reporter {}
class Second extends Reporter {}
public class AutoCloseableDetails {
    public static void main(String[] args) {
        try(
                First f = new First();
                Second s = new Second()
        ) {
        }
    }
}
```

输出为:

```
Creating First
Creating Second
Closing Second
Closing First
```

退出 try 块会调用两个对象的 close() 方法，并以与创建顺序相反的顺序关闭它们。顺序很重要，因为在此配置中，Second 对象可能依赖于 First 对象，因此如果 First 在第 Second 关闭时已经关闭。Second 的 close() 方法可能会尝试访问 First 中不再可用的某些功能。

假设我们在资源规范头中定义了一个不是 AutoCloseable 的对象

```
// exceptions/TryAnything.java
// {WillNotCompile}
class Anything {}
public class TryAnything {
    public static void main(String[] args) {
        try(
                Anything a = new Anything()
        ) {
        }
    }
}
```

正如我们所希望和期望的那样，Java 不会让我们这样做，并且出现编译时错误。

如果其中一个构造函数抛出异常怎么办?

```
// exceptions/ConstructorException.java
class CE extends Exception {}
class SecondExcept extends Reporter {
    SecondExcept() throws CE {
        super();
        throw new CE();
```

从来没有出现过这种情况

```
    }
}
public class ConstructorException {
    public static void main(String[] args) {
        try(
                First f = new First();
                SecondExcept s = new SecondExcept();
                Second s2 = new Second()
        ) {
            System.out.println("In body");
        } catch(CE e) {
            System.out.println("Caught: " + e);
        }
    }
}
```

输出为:

```
Creating First
Creating SecondExcept
Closing First
Caught: CE
```

现在资源规范头中定义了 3 个对象，中间的对象抛出异常。因此，编译器强制我们使用 catch 子句来捕获构造函数异常。这意味着资源规范头实际上被 try 块包围。

正如预期的那样，First 创建时没有发生意外，SecondExcept 在创建期间抛出异常。请注意，不会为 SecondExcept 调用 close()，因为如果构造函数失败，则无法假设你可以安全地对该对象执行任何操作，包括关闭它。由于 SecondExcept 的异常，Second 对象实例 s2 不会被创建，因此也不会有清除事件发生。

如果没有构造函数抛出异常，但你可能会在 try 的主体中获取它们，则再次强制你实现 catch 子句:

```
// exceptions/BodyException.java
class Third extends Reporter {}
public class BodyException {
    public static void main(String[] args) {
        try(
                First f = new First();
                Second s2 = new Second()
        ) {
            System.out.println("In body");
            Third t = new Third();
            new SecondExcept();
            System.out.println("End of body");
        } catch(CE e) {
            System.out.println("Caught: " + e);
        }
    }
```

```
}
```

输出为:

```
Creating First
Creating Second
In body
Creating Third
Creating SecondExcept
Closing Second
Closing First
Caught: CE
```

请注意，第 3 个对象永远不会被清除。那是因为它不是在资源规范头中创建的，所以它没有被保护。这很重要，因为 Java 在这里没有以警告或错误的形式提供指导，因此像这样的错误很容易漏掉。实际上，如果依赖某些集成开发环境来自动重写代码，以使用 try-with-resources 特性，那么它们（在撰写本文时）通常只会保护它们遇到的第一个对象，而忽略其余的对象。

最后，让我们看一下抛出异常的 close() 方法:

```java
// exceptions/CloseExceptions.java
class CloseException extends Exception {}
class Reporter2 implements AutoCloseable {
    String name = getClass().getSimpleName();
    Reporter2() {
        System.out.println("Creating " + name);
    }
    public void close() throws CloseException {
        System.out.println("Closing " + name);
    }
}
class Closer extends Reporter2 {
    @Override
    public void close() throws CloseException {
        super.close();
        throw new CloseException();
    }
}
public class CloseExceptions {
    public static void main(String[] args) {
        try(
                First f = new First();
                Closer c = new Closer();
                Second s = new Second()
        ) {
            System.out.println("In body");
        } catch(CloseException e) {
            System.out.println("Caught: " + e);
        }
    }
```

```
}
```

输出为:

```
Creating First
Creating Closer
Creating Second
In body
Closing Second
Closing Closer
Closing First
Caught: CloseException
```

从技术上讲，我们并没有被迫在这里提供一个 catch 子句；你可以通过 **main() throws CloseException** 的方式来报告异常。但 catch 子句是放置错误处理代码的典型位置。

请注意，因为所有三个对象都已创建，所以它们都以相反的顺序关闭 - 即使 Closer 也是如此。close() 抛出异常。当你想到它时，这就是你想要发生的事情，但是如果你必须自己编写所有这些逻辑，那么你可能会错过一些错误。想象一下所有代码都在那里，程序员没有考虑清理的所有含义，并且做错了。因此，应始终尽可能使用 try-with-resources。它有助于实现该功能，使得生成的代码更清晰，更易于理解。

16.12 异常匹配

抛出异常的时候，异常处理系统会按照代码的书写顺序找出“最近”的处理程序。找到匹配的处理程序之后，它就认为异常将得到处理，然后就不再继续查找。

查找的时候并不要求抛出的异常同处理程序所声明的异常完全匹配。派生类的对象也可以匹配其基类的处理程序，就像这样:

```java
// exceptions/Human.java
// Catching exception hierarchies
class Annoyance extends Exception {}
class Sneeze extends Annoyance {}
public class Human {
    public static void main(String[] args) {
        // Catch the exact type:
        try {
            throw new Sneeze();
        } catch(Sneeze s) {
            System.out.println("Caught Sneeze");
        } catch(Annoyance a) {
            System.out.println("Caught Annoyance");
        }
        // Catch the base type:
        try {
            throw new Sneeze();
```

```
        } catch(Annoyance a) {
            System.out.println("Caught Annoyance");
        }
    }
}
```

输出为:

```
Caught Sneeze
Caught Annoyance
```

Sneeze 异常会被第一个匹配的 catch 子句捕获，也就是程序里的第一个。然而如果将这个 catch 子句删掉，只留下 Annoyance 的 catch 子句，该程序仍然能运行，因为这次捕获的是 Sneeze 的基类。换句话说，catch（Annoyance a）会捕获 Annoyance 以及所有从它派生的异常。这一点非常有用，因为如果决定在方法里加上更多派生异常的话，只要客户程序员捕获的是基类异常，那么它们的代码就无需更改。

如果把捕获基类的 catch 子句放在最前面，以此想把派生类的异常全给“屏蔽”掉，就像这样:

```
try {
    throw new Sneeze();
} catch(Annoyance a) {
    // ...
} catch(Sneeze s) {
    // ...
}
```

此时，编译器会发现 Sneeze 的 catch 子句永远得不到执行，因此它会向你报告错误。

16.13 其他可选方式

异常处理系统就像一个活门（trap door），使你能放弃程序的正常执行序列。当“异常情形” 发生的时候，正常的执行已变得不可能或者不需要了，这时就要用到这个“活门”。异常代表了当前方法不能继续执行的情形。开发异常处理系统的原因是，如果为每个方法所有可能发生的错误都进行处理的话，任务就显得过于繁重了，程序员也不愿意这么做。结果常常是将错误忽格。应该注意到，开发异常处理的初衷是为了方便程序员处理错误。

异常处理的一个重要原则是“只有在你知道如何处理的情况下才捕获异常”。实际上，异常处理的一个重要目标就是把错误处理的代码同错误发生的地点相分离。这使你能在一段代码中专注于要完成的事情，至于如何处理错误，则放在另一段代码中完成。这样一来，主要代码就不会与错误处理逻辑混在一起，也更容易理解和维护。通过允许一个处理程序去处理多个出错点，异常处理还使得错误处理代码的数量趋于减少。

“被检查的异常”使这个问题变得有些复杂，因为它们强制你在可能还没准备好处理错误的时候被迫加上 catch 子句，这就导致了吞食则有害（harmful if swallowed）的问题:

对职业程序员来说，漂亮的程序是单纯而自然的逻辑、简单而基本的指令、丰富的批注，也就是新手程序员也能马上动手改的程序。而要写这样的程序，需要单纯、简单、美丽的需求。但可惜客人总是喜欢搞很复杂。

```
try {
    // ... to do something useful
} catch(ObligatoryException e) {} // Gulp!
```

程序员们只做最简单的事情（包括我自己，在本书第 1 版中也有这个问题），常常是无意中“吞食”了异常，然而一旦这么做，虽然能通过编译，但除非你记得复查并改正代码，否则异常将会丢失。异常确实发生了，但“吞食”后它却完全消失了。因为编译器强迫你立刻写代码来处理异常，所以这种看起来最简单的方法，却可能是最糟糕的做法。

当我意识到犯了这么大一个错误时，简直吓了一大跳，在本书第 2 版中，我在处理程序里通过打印栈轨迹的方法“修补”了这个问题（本章中的很多例子还是使用了这种方法，看起来还是比较合适的），虽然这样可以跟踪异常的行为，但是仍旧不知道该如何处理异常。这一节，我们来研究一下“被检查的异常”及其并发症，以及采用什么方法来解决这些问题。

这个话题看起来简单，但实际上它不仅复杂，更重要的是还非常多变。总有人会顽固地坚持自己的立场，声称正确答案（也是他们的答案）是显而易见的。我觉得之所以会有这种观点，是因为我们使用的工具已经不是 ANS1 标准出台前的像 C 那样的弱类型语言，而是像 C++ 和 Java 这样的“强静态类型语言”（也就是编译时就做类型检查的语言），这是前者所无法比拟的。当刚开始这种转变的时候（就像我一样），会觉得它带来的好处是那样明显，好像类型检查总能解决所有的问题。在此，我想结合我自己的认识过程，告诉读者我是怎样从对类型检查的绝对迷信变成持怀疑态度的，当然，很多时候它还是非常有用的，但是当它挡住我们的去路并成为障碍的时候，我们就得跨过去。只是这条界限往往并不是很清晰（我最喜欢的一句格言是：所有模型都是错误的，但有些是能用的）。

16.13.1 历史

异常处理起源于 PL/1 和 Mesa 之类的系统中，后来又出现在 CLU、Smalltalk、Modula-3、Ada、Eiffel、C++、Python、Java 以及后 Java 语言 Ruby 和 C# 中。Java 的设计和 C++ 很相似，只是 Java 的设计者去掉了一些他们认为 C++ 设计得不好的东西。

为了能向程序员提供一个他们更愿意使用的错误处理和恢复的框架，异常处理机制很晚才被加入 C++ 标准化过程中，这是由 C++ 的设计者 Bjarne Stroustrup 所倡议。C++ 的异常模型主要借鉴了 CLU 的做法。然而，当时其他语言已经支持异常处理了：包括 Ada、Smalltalk（两者都有异常处理，但是都没有异常说明），以及 Modula-3（它既有异常处理也有异常说明）。

Liskov 和 Snyder 在他们的一篇讨论该主题的独创性论文中指出，用瞬时风格（transient fashion）报告错误的语言（如 C 中）有一个主要缺陷，那就是：

> “…. 每次调用的时候都必须执行条件测试，以确定会产生何种结果。这使程序难以阅读并且有可能降低运行效率，因此程序员们既不愿意指出，也不愿意处理异常。”

因此，异常处理的初衷是要消除这种限制，但是我们又从 Java 的“被检查的异常”中看到了这种代码。他们继续写道：

多想个 10 秒钟，你可以不说「嗯，这个做得到」。

> “…. 要求程序员把异常处理程序的代码文本附接到会引发异常的调用上，这会降低程序的可读性，使得程序的正常思路被异常处理给破坏了。”

C++ 中异常的设计参考了 CLU 方式。Stroustrup 声称其目标是减少恢复错误所需的代码。我想他这话是说给那些通常情况下都不写 C 的错误处理的程序员们听的，因为要把那么多代码放到那么多地方实在不是什么好差事。所以他们写 C 程序的习惯是，忽略所有的错误，然后使用调试器来跟踪错误。这些程序员知道，使用异常就意味着他们要写一些通常不用写的、“多出来的”代码。因此，要把他们拉到“使用错误处理”的正轨上，“多出来的”代码决不能太多。我认为，评价 Java 的“被检查的异常”的时候，这一点是很重要的。

C++ 从 CLU 那里还带来另一种思想：异常说明。这样，就可以用编程的方式在方法签名中声明这个方法将会抛出异常。异常说明有两个目的：一个是“我的代码会产生这种异常，这由你来处理”。另一个是“我的代码忽略了这些异常，这由你来处理”。学习异常处理的机制和语法的时候，我们一直在关注“你来处理”部分，但这里特别值得注意的事实是，我们通常都忽略了异常说明所表达的完整含义。

C++ 的异常说明不属于函数的类型信息。编译时唯一要检查的是异常说明是不是前后一致；比如，如果函数或方法会抛出某些异常，那么它的重载版本或者派生版本也必须抛出同样的异常。与 Java 不同，C++ 不会在编译时进行检查以确定函数或方法是不是真的抛出异常，或者异常说明是不是完整（也就是说，异常说明有没有精确描述所有可能被抛出的异常）。这样的检查只发生在运行期间。如果抛出的异常与异常说明不符，C++ 会调用标准类库的 unexpected() 函数。

值得注意的是，由于使用了模板，C++ 的标准类库实现里根本没有使用异常说明。在 Java 中，对于泛型用于异常说明的方式存在着一些限制。

16.13.2 观点

首先，Java 无谓地发明了“被检查的异常”（很明显是受 C++ 异常说明的启发，以及受 C++ 程序员们一般对此无动于衷的事实的影响），但是，这还只是一次尝试，目前为止还没有别的语言采用这种做法。

其次，仅从示意性的例子和小程序来看，“被检查的异常”的好处很明显。但是当程序开始变大的时候，就会带来一些微妙的问题。当然，程序不是一下就变大的，这有个过程。如果把不适用于大项目的语言用于小项目，当这些项目不断膨胀时，突然有一天你会发现，原来可以管理的东西，现在已经变得无法管理了。这就是我所说的过多的类型检查，特别是“被检查的异常”所造成的问题。

看来程序的规模是个重要因素。由于很多讨论都用小程序来做演示，因此这并不足以说明问题。一名 C# 的设计人员发现：

> “仅从小程序来看，会认为异常说明能增加开发人员的效率，并提高代码的质量；但考察大项目的时候，结论就不同了-开发效率下降了，而代码质量只有微不足道的提高，甚至毫无提高”。

谈到未被捕获的异常的时候，CLU 的设计师们认为：

> “我们觉得强迫程序员在不知道该采取什么措施的时候提供处理程序，是不现实的。”

在解释为什么“函数没有异常说明就表示可以抛出任何异常”的时候，Stroustrup 这样认为：

> “但是，这样一来几乎所有的函数都得提供异常说明了，也就都得重新编译，而且还会妨碍它同其他语言的交互。这样会迫使程序员违反异常处理机制的约束，他们会写欺骗程序来掩盖异常。这将给没有注意到这些异常的人造成一种虚假的安全感。”

我们已经看到这种破坏异常机制的行为了-就在 Java 的“被检查的异常”里。

Martin Fowler（UML Distilled，Refactoring 和 Analysis Patterns 的作者）给我写了下面这段话：

> “…总体来说，我觉得异常很不错，但是 Java 的”被检查的异常“带来的麻烦比好处要多。”

过去，我曾坚定地认为“被检查的异常”和强静态类型检查对开发健壮的程序是非常必要的。但是，我看到的以及我使用一些动态（类型检查）语言的亲身经历告诉我，这些好处实际上是来自于：

1. 不在于编译器是否会强制程序员去处理错误，而是要有一致的、使用异常来报告错误的模型。
2. 不在于什么时候进行检查，而是一定要有类型检查。也就是说，必须强制程序使用正确的类型，至于这种强制施加于编译时还是运行时，那倒没关系。

此外，减少编译时施加的约束能显著提高程序员的编程效率。事实上，反射和泛型就是用来补偿静态类型检查所带来的过多限制，在本书很多例子中都会见到这种情形。

我已经听到有人在指责了，他们认为这种言论会令我名誉扫地，会让文明堕落，会导致更高比例的项目失败。他们的信念是应该在编译时指出所有错误，这样才能挽救项目，这种信念可以说是无比坚定的；其实更重要的是要理解编译器的能力限制。在 http://MindView.net/Books/BetterJava 上的补充材料中，我强调了自动构建过程和单元测试的重要性，比起把所有的东西都说成是语法错误，它们的效果可以说是事半功倍。下面这段话是至理名言：

> 好的程序设计语言能帮助程序员写出好程序，但无论哪种语言都避免不了程序员用它写出坏程序。

不管怎么说，要让 Java 把“被检查的异常”从语言中去除，这种可能性看来非常渺茫。对语言来说，这个变化可能太激进了点，况且 Sun 的支持者们也非常强大。Sun 有完全向后兼容的历史和策略，实际上所有 Sun 的软件都能在 Sun 的硬件上运行，无论它们有多么古老。然而，如果发现有些“被检查的异常”挡住了路，尤其是发现你不得不去对付那些不知道该如何处理的异常，还是有些办法的。

16.13.3 把异常传递给控制台

对于简单的程序，比如本书中的许多例子，最简单而又不用写多少代码就能保护异常信息的方法，就是把它们从 main() 传递到控制台。例如，为了读取信息而打开一个文件（在第 12 章将详细介绍），必须对 FileInputStream 进行打开和关闭操作，这就可能会产生异常。对于简单的程序，可以像这样做（本书中很多地方采用了这种方法）：

疯狂的程序员决不是靠狂妄和拼命的程序员，而是能够脚踏实地，持续努力的程序员，一个程序员真正做到这两点，技术上去后，唯一能限制他的只有想象力，到那个时候才算“疯狂的程序员”，这种程序员，才能令对手无比恐惧。

```java
// exceptions/MainException.java
import java.util.*;
import java.nio.file.*;
public class MainException {
    // Pass exceptions to the console:
    public static void main(String[] args) throws Exception {
        // Open the file:
        List<String> lines = Files.readAllLines(
                Paths.get("MainException.java"));
        // Use the file ...
    }
}
```

注意，main() 作为一个方法也可以有异常说明，这里异常的类型是 Exception，它也是所有“被检查的异常”的基类。通过把它传递到控制台，就不必在 main() 里写 try-catch 子句了。（不过，实际的文件输入输出操作比这个例子要复杂得多。你将会在 文件 和 附录：I/O 流 章节中学到更多）

16.13.4 把“被检查的异常”转换为“不检查的异常”

在编写你自己使用的简单程序时，从主方法中抛出异常是很方便的，但这不是通用的方法。

问题的实质是，当在一个普通方法里调用别的方法时，要考虑到“我不知道该这样处理这个异常，但是也不想把它‘吞’了，或若打印一些无用的消息”。异常链提供了一种新的思路来解决这个问题。可以直接把“被检查的异常”包装进 RuntimeException 里面，就像这样：

```java
try {
    // ... to do something useful
} catch(IDontKnowWhatToDoWithThisCheckedException e) {
    throw new RuntimeException(e);
}
```

如果想把“被检查的异常”这种功能“屏蔽”掉的话，这看上去像是一个好办法。不用“吞下”异常，也不必把它放到方法的异常说明里面，而异常链还能保证你不会丢失任何原始异常的信息。

这种技巧给了你一种选择，你可以不写 try-catch 子句和/或异常说明，直接忽略异常，让它自己沿着调用栈往上“冒泡”，同时，还可以用 getCause() 捕获并处理特定的异常，就像这样：

```java
// exceptions/TurnOffChecking.java
// "Turning off" Checked exceptions
import java.io.*;
class WrapCheckedException {
    void throwRuntimeException(int type) {
        try {
            switch(type) {
                case 0: throw new FileNotFoundException();
                case 1: throw new IOException();
                case 2: throw new
```

```
                        RuntimeException("Where am I?");
                default: return;
            }
        } catch(IOException | RuntimeException e) {
            // Adapt to unchecked:
            throw new RuntimeException(e);
        }
    }
}
class SomeOtherException extends Exception {}
public class TurnOffChecking {
    public static void main(String[] args) {
        WrapCheckedException wce =
                new WrapCheckedException();
        // You can call throwRuntimeException() without
        // a try block, and let RuntimeExceptions
        // leave the method:
        wce.throwRuntimeException(3);
        // Or you can choose to catch exceptions:
        for(int i = 0; i < 4; i++)
            try {
                if(i < 3)
                    wce.throwRuntimeException(i);
                else
                    throw new SomeOtherException();
            } catch(SomeOtherException e) {
                System.out.println(
                        "SomeOtherException: " + e);
            } catch(RuntimeException re) {
                try {
                    throw re.getCause();
                } catch(FileNotFoundException e) {
                    System.out.println(
                            "FileNotFoundException: " + e);
                } catch(IOException e) {
                    System.out.println("IOException: " + e);
                } catch(Throwable e) {
                    System.out.println("Throwable: " + e);
                }
            }
    }
}
```

输出为:

```
FileNotFoundException: java.io.FileNotFoundException
IOException: java.io.IOException
Throwable: java.lang.RuntimeException: Where am I?
SomeOtherException: SomeOtherException
```

WrapCheckedException.throwRuntimeException() 的代码可以生成不同类型的异常。这些异

常被捕获并包装进了 RuntimeException 对象，所以它们成了这些运行时异常的“cause”了。

在 TurnOfChecking 里，可以不用 try 块就调用 throwRuntimeException()，因为它没有抛出“被检查的异常”。但是，当你准备好去捕获异常的时候，还是可以用 try 块来捕获任何你想捕获的异常的。应该捕获 try 块肯定会抛出的异常，这里就是 SomeOtherException，RuntimeException 要放到最后去捕获。然后把 getCause() 的结果（也就是被包装的那个原始异常）抛出来。这样就把原先的那个异常给提取出来了，然后就可以用它们自己的 catch 子句进行处理。

本书余下部分将会在合适的时候使用这种“用 RuntimeException 来包装，被检查的异常”的技术。另一种解决方案是创建自己的 RuntimeException 的子类。在这种方式中，不必捕获它，但是希望得到它的其他代码都可以捕获它。

16.14 异常指南

应该在下列情况下使用异常：

1. 尽可能使用 try-with-resource。
2. 在恰当的级别处理问题。（在知道该如何处理的情况下才捕获异常。）
3. 解决问题并且重新调用产生异常的方法。
4. 进行少许修补，然后绕过异常发生的地方继续执行。
5. 用别的数据进行计算，以代替方法预计会返回的值。
6. 把当前运行环境下能做的事情尽量做完，然后把相同的异常重抛到更高层。
7. 把当前运行环境下能做的事情尽量做完，然后把不同的异常抛到更高层。
8. 终止程序。
9. 进行简化。（如果你的异常模式使问题变得太复杂，那用起来会非常痛苦也很烦人。）
10. 让类库和程序更安全。（这既是在为调试做短期投资，也是在为程序的健壮性做长期投资。）

16.15 本章小结

异常是 Java 程序设计不可分割的一部分，如果不了解如何使用它们，那你只能完成很有限的工作。正因为如此，本书专门在此介绍了异常——对于许多类库（例如提到过的 I/O 库），如果不处理异常，你就无法使用它们。

异常处理的优点之一就是它使得你可以在某处集中精力处理你要解决的问题，而在另一处处理你编写的这段代码中产生的错误。尽管异常通常被认为是一种工具，使得你可以在运行时报告错误并从错误中恢复，但是我一直怀疑到底有多少时候“恢复”真正得以实现了，或者能够得以实现。我认为这种情况少于 10%，并且即便是这 10%，也只是将栈展开到某个已知的稳定状态，而并没有实际执行任何种类的恢复性行为。无论这是否正确，我一直相信“报告”功能是异常的精髓所在. Java 坚定地强调将所有的错误都以异常形式报告的这一事实，正是它远远超过语如 C++ 这类语言的长处之一，因为在 C++ 这类

语言中，需要以大量不同的方式来报告错误，或者根本就没有提供错误报告功能。一致的错误报告系统意味着，你再也不必对所写的每一段代码，都质问自己“错误是否正在成为漏网之鱼？”（只要你没有“吞咽”异常，这是关键所在！）。

就像你将要在后续章节中看到的，通过将这个问题甩给其他代码-即使你是通过抛出 RuntimeException 来实现这一点的–你在设计和实现时，便可以专注于更加有趣和富有挑战性的问题了。

16.16 后记：Exception Bizarro World

（来自于 2011 年的一篇博文）

我的朋友 James Ward 正在尝试使用 JDBC 创建一些非常简单的教学示例，并且不断被检查的异常所挫败。他向我指出 Howard Lewis Ship 的帖子“被检查的例外的悲剧”。特别是。James 对他必须跳过去做一些应该简单的事情的所有环感到沮丧。即使在 finally 块中，他也不得不放入更多的 try-catch 子句，因为关闭连接也会导致异常。它在哪里结束？为了简单起见，你必须在环之后跳过环（请注意，try-with-resources 语句可以显著改善这种情况）。

我们开始讨论 Go 编程语言，我很着迷，因为 Rob Pike 等人。我们已经清楚地提出了许多关于语言设计的非常尖锐和基本的问题。基本上，他们已经采取了我们开始接受的有关语言的所有内容，并询问“为什么？”关于每一种语言。学习这门语言真的让你思考和怀疑。

我的印象是，Go 团队决定不做任何假设，只有在明确需要特征的情况下才能改进语言。他们似乎并不担心进行破坏旧代码的更改 - 他们创建了一个重写工具，因此如果他们进行了这些更改，它将为您重写代码。这使他们能够使语言成为一个持续的实验，以发现真正需要的东西，而不是做 Big Upfront Design。

他们做出的最有趣的决定之一是完全排除异常。你没有看错 —— 他们不只是遗漏了经过检查的异常情况。他们遗漏了所有异常情况。

替代方案非常简单，起初它几乎看起来像 C 一样。因为 Go 从一开始就包含了元组，所以你可以轻松地从函数调用中返回两个对象：

```
result, err := functionCall()
```

（:= 告诉 Go 语言这里定义 result 和 err，并且推断他们的数据类型）

就是这样：对于每次调用，您都会获得结果对象和错误对象。您可以立即检查错误（这是典型的，因为如果某些操作失败，则不太可能继续下一步），或者稍后检查是否有效。

起初这似乎很原始，是古代的回归。但到目前为止，我发现 Go 中的决定都得到了很好的考虑，值得深思。我只是做出反应，因为我的大脑是异常的吗？这会如何影响 James 的问题？

它发生在我身上，我已经将异常处理视为一种并行执行路径。如果你遇到异常，你会跳出正常的路径进入这个并行执行路径，这是一种“奇异世界”，你不再做你写的东西，而是跳进 catch 和 finally 子句。正是这种替代执行路径的世界导致了 James 抱怨的问题。

即便程序不行了，（你觉得）程序写得如何？

James 创造了一个对象。理想的情况下。对象创建不会导致潜在的异常，因此你必须抓住它们。你必须通过 try-finally 跟踪创建以确保清理发生（Python 团队意识到清理不是一个特殊的条件，而是一个单独的问题，所以他们创建了一个不同的语言构造 - 以便停止混淆二者）。任何导致异常的调用都会停止正常的执行路径并跳转（通过并行 bizarro-world）到 catch 子句。

关于异常的一个基本假设是，我们通过在块结束时收集所有错误处理代码而不是在它们发生时处理错误来获益。在这两种情况下，我们都会停止正常执行，但是异常处理有一个自动机制，它会将你从正常的执行路径中抛出，跳转到你的并行异常世界，然后在正确的处理程序中再次弹出你。

跳入奇异的世界会给 James 带来问题，它为所有程序员增加了更多的工作：因为你无法知道什么时候会发生什么事（你可以随时进入奇怪的世界），你必须添加一些 try 块来确保没有任何东西从裂缝中滑落。您最终必须进行额外的编程以补偿异常机制（它似乎类似于补偿共享内存并发所需的额外工作）。

Go 团队采取了大胆的举动，质疑所有这些，并说，“让我们毫无例外地尝试它，看看会发生什么。”是的，这意味着你通常会在发生错误的地方处理错误，而不是最后将它们聚集在一起 try 块。但这也意味着关于一件事的代码是本地化的，也许这并不是那么糟糕。这也可能意味着您无法轻松组合常见的错误处理代码（除非您确定了常用代码并将其放入函数中，也不是那么糟糕）。但这绝对意味着您不必担心有多个可能的执行路径而且所有这些都需要。

17 代码校验

17.0.1 你永远不能保证你的代码是正确的，你只能证明它是错的。

让我们先暂停编程语言特性的学习，看看一些代码基础知识。特别是能让你的代码更加健壮的知识。

17.1 测试

17.1.1 如果没有测试过，它就是不能工作的。

Java 是一个静态类型的语言，程序员经常对一种编程语言明显的安全性感到过于舒适，“能通过编译器，那就是没问题的”。但静态类型检查是一种非常局限性的测试，只是说明编译器接受你代码中的语法和基本类型规则，并不意味着你的代码达到程序的目标。随着你代码经验的丰富，你逐渐了解到你的代码从来没有满足过这些目标。迈向代码校验的第一步就是创建测试，针对你的目标检查代码的行为。

单元测试 这个过程是将集成测试构建到你创建的所有代码中，并在每次构建系统时运行这些测试。这样，构建过程不仅能检查语法的错误，同时也能检查语义的错误。

“单元”是指测试一小部分代码。通常，每个类都有测试来检查它所有方法的行为。“系统”测试则是不同的，它检查的是整个程序是否满足要求。

C 风格的语言，尤其是 C++，通常会认为性能比安全更重要。用 Java 编程比 C++（一般认为大概快两倍）快的原因是 Java 的安全性保障：比如垃圾回收以及改良的类型检测等特性。通过将单元测试集成到构建过程中，你扩大了这个安全保障，因而有了更快的开发效率。当发现设计或实现的缺陷时，可以更容易、更大胆地重构你的代码。

我自己的测试经历开始于我意识到要确保书中代码的正确性，书中的所有程序必须能够通过合适的构建系统自动提取、编译。这本书所使用的构建系统是 Gradle。你只要在安装 JDK 后输入 **gradlew compileJava**，就能编译本书的所有代码。自动提取和自动编译的效果对本书代码的质量是如此的直接和引人注目，（在我看来）这会很快成为任何编程书籍的必备条件——你怎么能相信没有编译的代码呢？我还发现我可以使用搜索和替换在整本书进行大范围的修改，如果引入了一个错误，代码提取器和构建系统就会清除它。随着程序越来越复杂，我在系统中发现了一个严重的漏洞。编译程序毫无疑问是重要

简单不先于复杂，而是在复杂之后。

的第一步，对于一本要出版的书而言，这看来是相当具有革命意义的发现（由于出版压力，你经常打开一本程序设计的书会发现书中代码的错误）。但是，我收到了来自读者反馈代码中存在语义问题。当然，这些问题可以通过运行代码发现。我在早期实现一个自动化执行测试系统时尝试了一些不太有效的方式，但迫于出版压力，我明白我的程序绝对有问题，并会以 bug 报告的方式让我自食恶果。我也经常收到读者的抱怨说，我没有显示足够的代码输出。我需要验证程序的输出，并且在书中显示验证的输出。我以前的意见是读者应该一边看书一边运行代码，许多读者就是这么做的并且从中受益。然而，这种态度背后的原因是，我无法保证书中的输出是正确的。从经验来看，我知道随着时间的推移，会发生一些事情，使得输出不再正确（或者一开始就不正确）。为了解决这个问题，我利用 Python 创建了一个工具（你将在下载的示例中找到此工具）。本书中的大多数程序都产生控制台输出，该工具将该输出与源代码清单末尾的注释中显示的预期输出进行比较，所以读者可以看到预期的输出，并且知道这个输出已经被构建程序验证过。

JUnit 最初的 JUnit 发布于 2000 年，大概是基于 Java 1.0，因此不能使用 Java 的反射工具。因此，用旧的 JUnit 编写单元测试是一项相当繁忙和冗长的工作。我发现这个设计令人不爽，并编写了自己的单元测试框架作为 注解 一章的示例。这个框架走向了另一个极端，“尝试最简单可行的方法”（极限编程中的一个关键短语）。从那之后，JUnit 通过反射和注解得到了极大的改进，大大简化了编写单元测试代码的过程。在 Java8 中，他们甚至增加了对 lambdas 表达式的支持。本书使用当时最新的 Junit5 版本

在 JUnit 最简单的使用中，使用 **@Test** 注解标记表示测试的每个方法。JUnit 将这些方法标识为单独的测试，并一次设置和运行一个测试，采取措施避免测试之间的副作用。

让我们尝试一个简单的例子。**CountedList** 继承 **ArrayList**，添加信息来追踪有多少个 **CountedLists** 被创建:

```
// validating/CountedList.java
// Keeps track of how many of itself are created.
package validating;
import java.util.*;
public class CountedList extends ArrayList<String> {
    private static int counter = 0;
    private int id = counter++;
    public CountedList() {
        System.out.println("CountedList #" + id);
    }
    public int getId() { return id; }
}
```

标准做法是将测试放在它们自己的子目录中。测试还必须放在包中，以便 JUnit 能发现它们:

```
// validating/tests/CountedListTest.java
// Simple use of JUnit to test CountedList.
package validating;
```

今天的最后，是我最喜欢的一段话，每次看到都特别有感觉，喜欢这段话里那种感情，那种美好的景象，所以把它作为今天的结束。

```java
import java.util.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
public class CountedListTest {
private CountedList list;
    @BeforeAll
    static void beforeAllMsg() {
        System.out.println(">>> Starting CountedListTest");
    }

    @AfterAll
    static void afterAllMsg() {
        System.out.println(">>> Finished CountedListTest");
    }

    @BeforeEach
    public void initialize() {
        list = new CountedList();
        System.out.println("Set up for " + list.getId());
        for(int i = 0; i < 3; i++)
            list.add(Integer.toString(i));
    }

    @AfterEach
    public void cleanup() {
        System.out.println("Cleaning up " + list.getId());
    }

    @Test
    public void insert() {
        System.out.println("Running testInsert()");
        assertEquals(list.size(), 3);
        list.add(1, "Insert");
        assertEquals(list.size(), 4);
        assertEquals(list.get(1), "Insert");
    }

    @Test
    public void replace() {
        System.out.println("Running testReplace()");
        assertEquals(list.size(), 3);
        list.set(1, "Replace");
        assertEquals(list.size(), 3);
        assertEquals(list.get(1), "Replace");
    }

    // A helper method to simplify the code. As
    // long as it's not annotated with @Test, it will
    // not be automatically executed by JUnit.
    private void compare(List<String> lst, String[] strs) {
        assertArrayEquals(lst.toArray(new String[0]), strs);
```

```
    }

    @Test
    public void order() {
        System.out.println("Running testOrder()");
        compare(list, new String[] { "0", "1", "2" });
    }

    @Test
    public void remove() {
        System.out.println("Running testRemove()");
        assertEquals(list.size(), 3);
        list.remove(1);
        assertEquals(list.size(), 2);
        compare(list, new String[] { "0", "2" });
    }

    @Test
    public void addAll() {
        System.out.println("Running testAddAll()");
        list.addAll(Arrays.asList(new String[] {
        "An", "African", "Swallow"}));
        assertEquals(list.size(), 6);
        compare(list, new String[] { "0", "1", "2",
        "An", "African", "Swallow" });
    }
}

/* Output:
>>> Starting CountedListTest
CountedList #0
Set up for 0
Running testRemove()
Cleaning up 0
CountedList #1
Set up for 1
Running testReplace()
Cleaning up 1
CountedList #2
Set up for 2
Running testAddAll()
Cleaning up 2
CountedList #3
Set up for 3
Running testInsert()
Cleaning up 3
CountedList #4
Set up for 4
Running testOrder()
Cleaning up 4
>>> Finished CountedListTest
```

```
*/
```

@BeforeAll 注解是在任何其他测试操作之前运行一次的方法。**@AfterAll** 是所有其他测试操作之后只运行一次的方法。两个方法都必须是静态的。

@BeforeEach 注解是通常用于创建和初始化公共对象的方法，并在每次测试前运行。可以将所有这样的初始化放在测试类的构造函数中，尽管我认为 **@BeforeEach** 更加清晰。JUnit 为每个测试创建一个对象，确保测试运行之间没有副作用。然而，所有测试的所有对象都是同时创建的 (而不是在测试之前创建对象)，所以使用 **@BeforeEach** 和构造函数之间的唯一区别是 **@BeforeEach** 在测试前直接调用。在大多数情况下，这不是问题，如果你愿意，可以使用构造函数方法。

如果你必须在每次测试后执行清理（如果修改了需要恢复的静态文件，打开文件需要关闭，打开数据库或者网络连接，etc），那就用注解 **@AfterEach**。

每个测试创建一个新的 **CountedListTest** 对象，任何非静态成员变量也会在同一时间创建。然后为每个测试调用 **initialize()** ，于是 list 被赋值为一个新的用字符串“0”、“1” 和 “2” 初始化的 **CountedList** 对象。观察 **@BeforeEach** 和 **@AfterEach** 的行为，这些方法在初始化和清理测试时显示有关测试的信息。

insert() 和 **replace()** 演示了典型的测试方法。JUnit 使用 **@Test** 注解发现这些方法，并将每个方法作为测试运行。在方法内部，你可以执行任何所需的操作并使用 JUnit 断言方法（以“assert”开头）验证测试的正确性（更全面的“assert”说明可以在 Junit 文档里找到）。如果断言失败，将显示导致失败的表达式和值。这通常就足够了，但是你也可以使用每个 JUnit 断言语句的重载版本，它包含一个字符串，以便在断言失败时显示。

断言语句不是必须的；你可以在没有断言的情况下运行测试，如果没有异常，则认为测试是成功的。

compare() 是“helper 方法”的一个例子，它不是由 JUnit 执行的，而是被类中的其他测试使用。只要没有 **@Test** 注解，JUnit 就不会运行它，也不需要特定的签名。在这里，**compare()** 是私有方法，表示仅在测试类中使用，但他同样可以是 **public** 。其余的测试方法通过将其重构为 **compare()** 方法来消除重复的代码。

本书使用 **build.gradle** 控制测试，运行本章节的测试，使用命令：`gradlew validating:test`，Gradle 不会运行已经运行过的测试，所以如果你没有得到测试结果，得先运行：`gradlew validating:clean`。

可以用下面这个命令运行本书的所有测试：

gradlew test

尽管可以用最简单的方法，如 **CountedListTest.java** 所示没那样，JUnit 还包括大量的测试结构，你可以到官网上学习它们。

JUnit 是 Java 最流行的单元测试框架，但也有其它可以替代的。你可以通过互联网发现更适合的那一个。

测试覆盖率的幻觉 测试覆盖率，同样也称为代码覆盖率，度量代码的测试百分比。百分比越高，测试的覆盖率越大。这里有很多方法

计算覆盖率，还有有帮助的文章Java 代码覆盖工具。

对于没有知识但处于控制地位的人来说，很容易在没有任何了解的情况下也有概念认为 100% 的测试覆盖是唯一可接受的值。这有一个问题，因为 100% 并不意味着是对测试有效性的良好测量。你可以测试所有需要它的东西，但是只需要 65% 的覆盖率。如果需要 100% 的覆盖，你将浪费大量时间来生成剩余的代码，并且在向项目添加代码时浪费的时间更多。

当分析一个未知的代码库时，测试覆盖率作为一个粗略的度量是有用的。如果覆盖率工具报告的值特别低（比如，少于百分之 40），则说明覆盖不够充分。然而，一个非常高的值也同样值得怀疑，这表明对编程领域了解不足的人迫使团队做出了武断的决定。覆盖工具的最佳用途是发现代码库中未测试的部分。但是，不要依赖覆盖率来得到测试质量的任何信息。

17.2 前置条件

前置条件的概念来自于契约式设计 (**Design By Contract, DbC**), 利用断言机制实现。我们从 Java 的断言机制开始来介绍 DBC，最后使用谷歌的 Guava 库作为前置条件。

断言（Assertions） 断言通过验证在程序执行期间满足某些条件，从而增加了程序的健壮性。举例，假设在一个对象中有一个数值字段表示日历上的月份。这个数字总是介于 1-12 之间。断言可以检查这个数字，如果超出了该范围，则报告错误。如果在方法的内部，则可以使用断言检查参数的有效性。这些是确保程序正确的重要测试，但是它们不能在编译时被检查，并且它们不属于单元测试的范围。

Java 断言语法 你可以通过其它程序设计架构来模拟断言的效果，因此，在 Java 中包含断言的意义在于它们易于编写。断言语句有两种形式:

assert boolean-expression;

assert boolean-expression: information-expression;

两者似乎告诉我们 **“我断言这个布尔表达式会产生 true”**，否则，将抛出 **AssertionError** 异常。

AssertionError 是 **Throwable** 的派生类，因此不需要异常说明。

不幸的是，第一种断言形式的异常不会生成包含布尔表达式的任何信息（与大多数其他语言的断言机制相反）。

下面是第一种形式的例子:

```
// validating/Assert1.java

// Non-informative style of assert
```

```
// Must run using -ea flag:
// {java -ea Assert1}
// {ThrowsException}
public class Assert1 {
    public static void main(String[] args) {
        assert false;
    }
}

/* Output:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError
at Assert1.main(Assert1.java:9)
*/
```

如果你正常运行程序，没有任何特殊的断言标志，则不会发生任何事情。你需要在运行程序时显式启用断言。一种简单的方法是使用 **-ea** 标志，它也可以表示为: **-enableassertion**，这将运行程序并执行任何断言语句。

输出中并没有包含多少有用的信息。另一方面，如果你使用 **information-expression** ，将生成一条有用的消息作为异常堆栈跟踪的一部分。最有用的 **information-expression** 通常是一串针对程序员的文本:

```
// validating/Assert2.java
// Assert with an information-expression
// {java Assert2 -ea}
// {ThrowsException}

public class Assert2 {
    public static void main(String[] args) {
    assert false:
    "Here's a message saying what happened";
    }
}
/* Output:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError:
Here's a message saying what happened
at Assert2.main(Assert2.java:8)
*/
```

information-expression 可以产生任何类型的对象，因此，通常将构造一个包含对象值的更复杂的字符串，它包含失败的断言。

你还可以基于类名或包名打开或关闭断言；也就是说，你可以对整个包启用或禁用断言。实现这一点的详细信息在 JDK 的断言文档中。此特性对于使用断言的大型项目来说很有用当你想打开或关闭某些断言时。但是，日志记录（*Logging*）或者调试（*Debugging*），可能是捕获这类信息的更好工具。

你还可以通过编程的方式通过链接到类加载器对象（**ClassLoader**）来控制断言。类加载器中有

几种方法允许动态启用和禁用断言，其中 **setDefaultAssertionStatus ()** , 它为之后加载的所有类设置断言状态。因此，你可以像下面这样悄悄地开启断言：

```
// validating/LoaderAssertions.java
// Using the class loader to enable assertions
// {ThrowsException}
public class LoaderAssertions {
public static void main(String[] args) {

    ClassLoader.getSystemClassLoader().
        setDefaultAssertionStatus(true);
        new Loaded().go();
    }
}

class Loaded {
    public void go() {
    assert false: "Loaded.go()";
    }
}
/* Output:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError:
Loaded.go()
at Loaded.go(LoaderAssertions.java:15)
at
LoaderAssertions.main(LoaderAssertions.java:9)
*/
```

这消除了在运行程序时在命令行上使用 **-ea** 标志的需要，使用 **-ea** 标志启用断言可能同样简单。当交付独立产品时，可能必须设置一个执行脚本让用户能够启动程序，配置其他启动参数，这么做是有意义的。然而，决定在程序运行时启用断言可以使用下面的 **static** 块来实现这一点，该语句位于系统的主类中：

```
static {
    boolean assertionsEnabled = false;
    // Note intentional side effect of assignment:
    assert assertionsEnabled = true;
    if(!assertionsEnabled)
        throw new RuntimeException("Assertions disabled");
}
```

如果启用断言，然后执行 **assert** 语句，**assertionsEnabled** 变为 **true** 。断言不会失败，因为分配的返回值是赋值的值。如果不启用断言，**assert** 语句不执行，**assertionsEnabled** 保持 false，将导致异常。

Guava 断言 因为启用 Java 本地断言很麻烦，Guava 团队添加一个始终启用的用来替换断言的 **Verify** 类。他们建议静态导入 **Verify** 方法：

```
// validating/GuavaAssertions.java
// Assertions that are always enabled.

import com.google.common.base.*;
import static com.google.common.base.Verify.*;
public class GuavaAssertions {
    public static void main(String[] args) {
        verify(2 + 2 == 4);
        try {
            verify(1 + 2 == 4);
        } catch(VerifyException e) {
            System.out.println(e);
        }

        try {
            verify(1 + 2 == 4, "Bad math");
        } catch(VerifyException e) {
            System.out.println(e.getMessage());
        }

        try {
            verify(1 + 2 == 4, "Bad math: %s", "not 4");
        } catch(VerifyException e) {
            System.out.println(e.getMessage());
        }

        String s = "";
        s = verifyNotNull(s);
        s = null;
        try {
            verifyNotNull(s);
        } catch(VerifyException e) {
            System.out.println(e.getMessage());
        }

        try {
            verifyNotNull(
                s, "Shouldn't be null: %s", "arg s");
        } catch(VerifyException e) {
            System.out.println(e.getMessage());
        }
    }
}
/* Output:
com.google.common.base.VerifyException
Bad math
Bad math: not 4
```

```
expected a non-null reference
Shouldn't be null: arg s
*/
```

这里有两个方法，使用变量 **verify()** 和 **verifyNotNull()** 来支持有用的错误消息。注意，**verifyNotNull()** 内置的错误消息通常就足够了，而 **verify()** 太一般，没有有用的默认错误消息。

使用断言进行契约式设计 *契约式设计 (DbC)* 是 Eiffel 语言的发明者 Bertrand Meyer 提出的一个概念，通过确保对象遵循某些规则来帮助创建健壮的程序。这些规则是由正在解决的问题的性质决定的，这超出了编译器可以验证的范围。虽然断言没有直接实现 **DBC**（Eiffel 语言也是如此），但是它们创建了一种非正式的 DbC 编程风格。DbC 假定服务供应者与该服务的消费者或客户之间存在明确指定的契约。在面向对象编程中，服务通常由对象提供，对象的边界 — 供应者和消费者之间的划分 — 是对象类的接口。当客户端调用特定的公共方法时，它们希望该调用具有特定的行为：对象状态改变，以及一个可预测的返回值。

Meyer 认为：

1. 应该明确指定行为，就好像它是一个契约一样。

2. 通过实现某些运行时检查来保证这种行为，他将这些检查称为前置条件、后置条件和不变项。

不管你是否同意，第一条总是对的，在大多数情况下，DbC 确实是一种有用的方法。（我认为，与任何解决方案一样，它的有用性也有界限。但如果你知道这些界限，你就知道什么时候去尝试。）尤其是，设计过程中一个有价值的部分是特定类 DbC 约束的表达式；如果无法指定约束，则你可能对要构建的内容了解得不够。

检查指令 详细研究 DbC 之前，思考最简单使用断言的办法，**Meyer** 称它为检查指令。检查指令说明你确信代码中的某个特定属性此时已经得到满足。检查指令的思想是在代码中表达非明显性的结论，而不仅仅是为了验证测试，也同样为了将来能够满足阅读者而有一个文档。

在化学领域，你也许会用一种纯液体去滴定测量另一种液体，当达到一个特定的点时，液体变蓝了。从两个液体的颜色上并不能明显看出；这是复杂反应的一部分。滴定完成后一个有用的检查指令是能够断定液体变蓝了。

检查指令是对你的代码进行补充，当你可以测试并阐明对象或程序的状态时，应该使用它。

前置条件 前置条件确保客户端 (调用此方法的代码) 履行其部分契约。这意味着在方法调用开始时几乎总是会检查参数（在你用那个方法做任何操作之前）以此保证它们的调用在方法中是合适的。因为你永远无法知道客户端会传递给你什么，前置条件是确保检查的一个好做法。

后置条件 后置条件测试你在方法中所做的操作的结果。这段代码放在方法调用的末尾，在 **return** 语句之前 (如果有的话)。对于长时间、复杂的方法，在返回计算结果之前需要对计算结果进行验证（也

就是说，在某些情况下，由于某种原因，你不能总是相信结果)，后置条件很重要，但是任何时候你可以描述方法结果上的约束时，最好将这些约束在代码中表示为后置条件。

不变性 不变性保证了必须在方法调用之间维护的对象的状态。但是，它并不会阻止方法在执行过程中暂时偏离这些保证，它只是在说对象的状态信息应该总是遵守状态规则：

1. 在进入该方法时。

2. 在离开方法之前。

此外，不变性是构造后对于对象状态的保证。

根据这个描述，一个有效的不变性被定义为一个方法，可能被命名为 **invariant()** ，它在构造之后以及每个方法的开始和结束时调用。方法以如下方式调用：

assert invariant();

这样，如果出于性能原因禁用断言，就不会产生开销。

放松 DbC 检查或非严格的 DbC 尽管 Meyer 强调了前置条件、后置条件和不变性的价值以及在开发过程中使用它们的重要性，他承认在一个产品中包含所有 DbC 代码并不总是实际的。你可以基于对特定位置的代码的信任程度放松 DbC 检查。以下是放松检查的顺序，最安全到最不安全：

1. 不变性检查在每个方法一开始的时候是不能进行的，因为在每个方法结束的时候进行不变性检查能保证一开始的时候对象处于有效状态。也就是说，通常情况下，你可以相信对象的状态不会在方法调用之间发生变化。这是一个非常安全的假设，你可以只在代码末尾使用不变性检查来编写代码。

2. 接下来禁用后置条件检查，当你进行合理的单元测试以验证方法是否返回了适当的值时。因为不变性检查是观察对象的状态，后置条件检查仅在方法期间验证计算结果，因此可能会被丢弃，以便进行单元测试。单元测试不会像运行时后置条件检查那样安全，但是它可能已经足够了，特别是当对自己的代码有信心时。

3. 如果你确信方法主体没有把对象改成无效状态，则可以禁用方法调用末尾的不变性检查。可以通过白盒单元测试 (通过访问私有字段的单元测试来验证对象状态) 来验证这一点。尽管它可能没有调用 **invariant()** 那么稳妥，可以将不变性检查从运行时测试 “迁移” 到构建时测试 (通过单元测试)，就像使用后置条件一样。

4. 禁用前置条件检查，但除非这是万不得已的情况下。因为这是最不安全、最不明智的选择，因为尽管你知道并且可以控制自己的代码，但是你无法控制客户端可能会传递给方法的参数。然而，某些情况下对性能要求很高，通过分析得到前置条件造成了这个瓶颈，而且你有某种合理的保证客户端不会违反前置条件 (比如自己编写客户端的情况下)，那么禁用前置条件检查是可接受的。

你不应该直接删除检查的代码，而只需要禁用检查 (添加注释)。这样如果发现错误，就可以轻松地恢复检查以快速发现问题。

没办法，这是一个公认的 bug。

DbC + 单元测试 下面的例子演示了将契约式设计中的概念与单元测试相结合的有效性。它显示了一个简单的先进先出 (FIFO) 队列，该队列实现为一个“循环”数组，即以循环方式使用的数组。当到达数组的末尾时，将绕回到开头。

我们可以对这个队列做一些契约定义：

1. 前置条件 (用于 put())：不允许将空元素添加到队列中。
2. 前置条件 (用于 put())：将元素放入完整队列是非法的。
3. 前置条件 (用于 get())：试图从空队列中获取元素是非法的。
4. 后置条件用于 get())：不能从数组中生成空元素。
5. 不变性：包含对象的区域不能包含任何空元素。
6. 不变性：不包含对象的区域必须只有空值。

下面是实现这些规则的一种方式，为每个 DbC 元素类型使用显式的方法调用。

首先，我们创建一个专用的 **Exception**：

```
// validating/CircularQueueException.java
package validating;
public class CircularQueueException extends RuntimeException {
        public CircularQueueException(String why) {
        super(why);
    }
}
```

它用来报告 **CircularQueue** 中出现的错误：

```
// validating/CircularQueue.java
// Demonstration of Design by Contract (DbC)
package validating;
import java.util.*;
public class CircularQueue {
    private Object[] data;
    private int in = 0, // Next available storage space
    out = 0; // Next gettable object
             // Has it wrapped around the circular queue?
    private boolean wrapped = false;
    public CircularQueue(int size) {
        data = new Object[size];
        // Must be true after construction:
        assert invariant();
    }

    public boolean empty() {
        return !wrapped && in == out;
    }

    public boolean full() {
        return wrapped && in == out;
    }
```

```java

    public boolean isWrapped() { return wrapped; }

    public void put(Object item) {
        precondition(item != null, "put() null item");
        precondition(!full(),
        "put() into full CircularQueue");
        assert invariant();
        data[in++] = item;
        if(in >= data.length) {
            in = 0;
            wrapped = true;
        }
        assert invariant();
    }

    public Object get() {
        precondition(!empty(),
        "get() from empty CircularQueue");
        assert invariant();
        Object returnVal = data[out];
        data[out] = null;
        out++;
        if(out >= data.length) {
            out = 0;
            wrapped = false;
        }
        assert postcondition(
        returnVal != null,
        "Null item in CircularQueue");
        assert invariant();
        return returnVal;
    }

    // Design-by-contract support methods:
    private static void precondition(boolean cond, String msg) {
        if(!cond) throw new CircularQueueException(msg);
    }

    private static boolean postcondition(boolean cond, String msg) {
        if(!cond) throw new CircularQueueException(msg);
        return true;
    }

    private boolean invariant() {
        // Guarantee that no null values are in the
        // region of 'data' that holds objects:
        for(int i = out; i != in; i = (i + 1) % data.length)
            if(data[i] == null)
                throw new CircularQueueException("null in
                    CircularQueue");
```

```
                    // Guarantee that only null values are outside the
                    // region of 'data' that holds objects:
            if(full()) return true;
            for(int i = in; i != out; i = (i + 1) % data.length)
                if(data[i] != null)
                    throw new CircularQueueException(
                        "non-null outside of CircularQueue range: " +
                            dump());
            return true;
        }

        public String dump() {
            return "in = " + in +
                ", out = " + out +
                ", full() = " + full() +
                ", empty() = " + empty() +
                ", CircularQueue = " + Arrays.asList(data);
        }
    }
```

in 计数器指示数组中下一个对象所在的位置。**out** 计数器指示下一个对象来自何处。**wrapped** 的 flag 表示 **in** 已经“绕着圆圈”走了，现在从后面出来了。当 **in** 和 **out** 重合时，队列为空 (如果包装为 **false**) 或满 (如果 **wrapped** 为 **true**)。

put() 和 **get()** 方法调用 **precondition()** ， **postcondition()**, 和 **invariant()**，这些都是在类中定义的私有方法。前置 **precondition()** 和 **postcondition()** 是用来阐明代码的辅助方法。

注意，**precondition()** 返回 **void** , 因为它不与断言一起使用。按照之前所说的，通常你会在代码中保留前置条件。通过将它们封装在 **precondition()** 方法调用中，如果你不得不做出关掉它们的可怕举动，你会有更好的选择。

postcondition() 和 **constant()** 都返回一个布尔值，因此可以在 **assert** 语句中使用它们。此外，如果出于性能考虑禁用断言，则根本不存在方法调用。**invariant()** 对对象执行内部有效性检查，如果你在每个方法调用的开始和结束都这样做，这是一个花销巨大的操作，就像 **Meyer** 建议的那样。所以，用代码清晰地表明是有帮助的，它帮助我调试了实现。此外，如果你对代码实现做任何更改，那么 **invariant()** 将确保你没有破坏代码，将不变性测试从方法调用移到单元测试代码中是相当简单的。如果你的单元测试是足够的，那么你应当对不变性保持一定的信心。

dump() 帮助方法返回一个包含所有数据的字符串，而不是直接打印数据。这允许我们用这部分信息做更多事。

现在我们可以为类创建 JUnit 测试:

```
// validating/tests/CircularQueueTest.java
package validating;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
public class CircularQueueTest {
    private CircularQueue queue = new CircularQueue(10);
```

```
    private int i = 0;

    @BeforeEach
    public void initialize() {
        while(i < 5) // Pre-load with some data
            queue.put(Integer.toString(i++));
    }

    // Support methods:
    private void showFullness() {
        assertTrue(queue.full());
        assertFalse(queue.empty());
        System.out.println(queue.dump());
    }

    private void showEmptiness() {
        assertFalse(queue.full());
        assertTrue(queue.empty());
        System.out.println(queue.dump());
    }

    @Test
    public void full() {
        System.out.println("testFull");
        System.out.println(queue.dump());
        System.out.println(queue.get());
        System.out.println(queue.get());
        while(!queue.full())
            queue.put(Integer.toString(i++));
            String msg = "";
        try {
            queue.put("");
        } catch(CircularQueueException e) {
            msg = e.getMessage();
            ∂System.out.println(msg);
        }
        assertEquals(msg, "put() into full CircularQueue");
        showFullness();
    }

    @Test
    public void empty() {
        System.out.println("testEmpty");
        while(!queue.empty())
            System.out.println(queue.get());
            String msg = "";
        try {
            queue.get();
        } catch(CircularQueueException e) {
            msg = e.getMessage();
            System.out.println(msg);
```

```
        }
        assertEquals(msg, "get() from empty CircularQueue");
        showEmptiness();
    }
    @Test
    public void nullPut() {
        System.out.println("testNullPut");
        String msg = "";
        try {
            queue.put(null);
        } catch(CircularQueueException e) {
            msg = e.getMessage();
            System.out.println(msg);
        }
        assertEquals(msg, "put() null item");
    }

    @Test
    public void circularity() {
        System.out.println("testCircularity");
        while(!queue.full())
            queue.put(Integer.toString(i++));
            showFullness();
            assertTrue(queue.isWrapped());

        while(!queue.empty())
            System.out.println(queue.get());
            showEmptiness();

        while(!queue.full())
            queue.put(Integer.toString(i++));
            showFullness();

        while(!queue.empty())
            System.out.println(queue.get());
            showEmptiness();
     }
 }
/* Output:
testNullPut
put() null item
testCircularity
in = 0, out = 0, full() = true, empty() = false,
CircularQueue =
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0
1
2
3
4
5
```

```
6
7
8
9
in = 0, out = 0, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
in = 0, out = 0, full() = true, empty() = false,
CircularQueue =
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
10
11
12
13
14
15
16
17
18
19
in = 0, out = 0, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
testFull
in = 5, out = 0, full() = false, empty() = false,
CircularQueue =
[0, 1, 2, 3, 4, null, null, null, null, null]
0
1
put() into full CircularQueue
in = 2, out = 2, full() = true, empty() = false,
CircularQueue =
[10, 11, 2, 3, 4, 5, 6, 7, 8, 9]
testEmpty
0
1
2
3
4
get() from empty CircularQueue
in = 5, out = 5, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
*/
```

initialize() 添加了一些数据，因此每个测试的 **CircularQueue** 都是部分满的。**showFullness()** 和 **showempty()** 表明 **CircularQueue** 是满的还是空的，这四种测试方法中的每一种都确保了

CircularQueue 功能在不同的地方正确运行。

通过将 Dbc 和单元测试结合起来，你不仅可以同时使用这两种方法，还可以有一个迁移路径—你可以将一些 Dbc 测试迁移到单元测试中，而不是简单地禁用它们，这样你仍然有一定程度的测试。

使用 Guava 前置条件 在非严格的 DbC 中，前置条件是 DbC 中你不想删除的那一部分，因为它可以检查方法参数的有效性。那是你没有办法控制的事情，所以你需要对其检查。因为 Java 在默认情况下禁用断言，所以通常最好使用另外一个始终验证方法参数的库。

谷歌的 Guava 库包含了一组很好的前置条件测试，这些测试不仅易于使用，而且命名也足够好。在这里你可以看到它们的简单用法。库的设计人员建议静态导入前置条件:

```java
// validating/GuavaPreconditions.java
// Demonstrating Guava Preconditions
import java.util.function.*;
import static com.google.common.base.Preconditions.*;
public class GuavaPreconditions {
    static void test(Consumer<String> c, String s) {
        try {
            System.out.println(s);
            c.accept(s);
            System.out.println("Success");
        } catch(Exception e) {
            String type = e.getClass().getSimpleName();
            String msg = e.getMessage();
            System.out.println(type +
            (msg == null ? "" : ": " + msg));
        }
    }

    public static void main(String[] args) {
        test(s -> s = checkNotNull(s), "X");
        test(s -> s = checkNotNull(s), null);
        test(s -> s = checkNotNull(s, "s was null"), null);
        test(s -> s = checkNotNull(
        s, "s was null, %s %s", "arg2", "arg3"), null);
        test(s -> checkArgument(s == "Fozzie"), "Fozzie");
        test(s -> checkArgument(s == "Fozzie"), "X");
        test(s -> checkArgument(s == "Fozzie"), null);
        test(s -> checkArgument(
        s == "Fozzie", "Bear Left!"), null);
        test(s -> checkArgument(
        s == "Fozzie", "Bear Left! %s Right!", "Frog"),
        null);
        test(s -> checkState(s.length() > 6), "Mortimer");
        test(s -> checkState(s.length() > 6), "Mort");
        test(s -> checkState(s.length() > 6), null);
        test(s ->
        checkElementIndex(6, s.length()), "Robert");
```

```
        test(s ->
        checkElementIndex(6, s.length()), "Bob");
        test(s ->
        checkElementIndex(6, s.length()), null);
        test(s ->
        checkPositionIndex(6, s.length()), "Robert");
        test(s ->
        checkPositionIndex(6, s.length()), "Bob");
        test(s ->
        checkPositionIndex(6, s.length()), null);
        test(s -> checkPositionIndexes(
        0, 6, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
        0, 10, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
        0, 11, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
        -1, 6, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
        7, 6, s.length()), "Hieronymus");
        test(s -> checkPositionIndexes(
        0, 6, s.length()), null);
    }
}
/* Output:
X
Success
null
NullPointerException
null
NullPointerException: s was null
null
NullPointerException: s was null, arg2 arg3
Fozzie
Success
X
IllegalArgumentException
null
IllegalArgumentException
null
IllegalArgumentException: Bear Left!
null
IllegalArgumentException: Bear Left! Frog Right!
Mortimer
Success
Mort
IllegalStateException
null
NullPointerException
Robert
IndexOutOfBoundsException: index (6) must be less than
```

```
size (6)
Bob
IndexOutOfBoundsException: index (6) must be less than
size (3)
null
NullPointerException
Robert
Success
Bob
IndexOutOfBoundsException: index (6) must not be
greater than size (3)
null
NullPointerException
Hieronymus
Success
Hieronymus
Success
Hieronymus
IndexOutOfBoundsException: end index (11) must not be
greater than size (10)
Hieronymus
IndexOutOfBoundsException: start index (-1) must not be
negative
Hieronymus
IndexOutOfBoundsException: end index (6) must not be
less than start index (7)
null
NullPointerException
*/
```

虽然 Guava 的前置条件适用于所有类型，但我这里只演示 **字符串（String）**类型。**test()** 方法需要一个 Consumer，因此我们可以传递一个 lambda 表达式作为第一个参数，传递给 lambda 表达式的字符串作为第二个参数。它显示字符串，以便在查看输出时确定方向，然后将字符串传递给 lambda 表达式。try 块中的第二个 **println**() 仅在 lambda 表达式成功时才显示; 否则 catch 块将捕获并显示错误信息。注意 **test()** 方法消除了多少重复的代码。

每个前置条件都有三种不同的重载形式：一个什么都没有，一个带有简单字符串消息，以及带有一个字符串和替换值。为了提高效率，只允许 **%s** (字符串类型) 替换标记。在上面的例子中，演示了 **checkNotNull()** 和 **checkArgument()** 这两种形式。但是它们对于所有前置条件方法都是相同的。注意 **checkNotNull()** 的返回参数，所以你可以在表达式中内联使用它。下面是如何在构造函数中使用它来防止包含 **Null** 值的对象构造：

```java
/ validating/NonNullConstruction.java
import static com.google.common.base.Preconditions.*;
public class NonNullConstruction {
    private Integer n;
    private String s;
    NonNullConstruction(Integer n, String s) {
```

```
        this.n = checkNotNull(n);
        this.s = checkNotNull(s);
    }
    public static void main(String[] args) {
        NonNullConstruction nnc =
        new NonNullConstruction(3, "Trousers");
    }
}
```

checkArgument() 接受布尔表达式来对参数进行更具体的测试，失败时抛出 **IllegalArgumentException**，**checkState()** 用于测试对象的状态（例如，不变性检查），而不是检查参数，并在失败时抛出 **IllegalStateException** 。

最后三个方法在失败时抛出 **IndexOutOfBoundsException**。**checkElementIndex()** 确保其第一个参数是列表、字符串或数组的有效元素索引，其大小由第二个参数指定。**checkPositionIndex()** 确保它的第一个参数在 0 到第二个参数 (包括第二个参数) 的范围内。**checkPositionIndexes()** 检查 **[first_arg, second_arg]** 是一个列表的有效子列表，由第三个参数指定大小的字符串或数组。

所有的 Guava 前置条件对于基本类型和对象都有必要的重载。

17.3 测试驱动开发

之所以可以有测试驱动开发（TDD）这种开发方式，是因为如果你在设计和编写代码时考虑到了测试，那么你不仅可以写出可测试性更好的代码，而且还可以得到更好的代码设计。一般情况下这个说法都是正确的。一旦我想到“我将如何测试我的代码？”，这个想法将使我的代码产生变化，并且往往是从“可测试”转变为“可用”。

纯粹的 TDD 主义者会在实现新功能之前就为其编写测试，这称为测试优先的开发。我们采用一个简易的示例程序来进行说明，它的功能是反转 **String** 中字符的大小写。让我们随意添加一些约束：**String** 必须小于或等于 30 个字符，并且必须只包含字母，空格，逗号和句号 (英文)。

此示例与标准 TDD 不同，因为它的作用在于接收 **StringInverter** 的不同实现，以便在我们逐步满足测试的过程中来体现类的演变。为了满足这个要求，将 **StringInverter** 作为接口：

```
// validating/StringInverter.java
package validating;

interface StringInverter {
    String invert(String str);
}
```

现在我们通过可以编写测试来表述我们的要求。以下所述通常不是你编写测试的方式，但由于我们在此处有一个特殊的约束：我们要对 **StringInverter** 多个版本的实现进行测试，为此，我们利用了 JUnit5 中最复杂的新功能之一：动态测试生成。顾名思义，通过它你可以使你所编写的代码在运行时生成测试，而不需要你对每个测试显式编码。这带来了许多新的可能性，特别是在明确地需要编写一整套

测试而令人望而却步的情况下。

JUnit5 提供了几种动态生成测试的方法，但这里使用的方法可能是最复杂的。**DynamicTest.stream()** 方法采用了：

- 对象集合上的迭代器 (versions)，这个迭代器在不同组的测试中是不同的。迭代器生成的对象可以是任何类型，但是只能有一种对象生成，因此对于存在多个不同的对象类型时，必须人为地将它们打包成单个类型。
- **Function**，它从迭代器获取对象并生成描述测试的 **String** 。
- **Consumer**，它从迭代器获取对象并包含基于该对象的测试代码。

在此示例中，所有代码将在 **testVersions()** 中进行组合以防止代码重复。迭代器生成的对象是对 **DynamicTest** 的不同实现，这些对象体现了对接口不同版本的实现：

```java
// validating/tests/DynamicStringInverterTests.java
package validating;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.DynamicTest.*;

class DynamicStringInverterTests {
    // Combine operations to prevent code duplication:
    Stream<DynamicTest> testVersions(String id,
        Function<StringInverter, String> test) {
        List<StringInverter> versions = Arrays.asList(
            new Inverter1(), new Inverter2(),
            new Inverter3(), new Inverter4());
        return DynamicTest.stream(
            versions.iterator(),
            inverter -> inverter.getClass().getSimpleName(),
            inverter -> {
                System.out.println(
                    inverter.getClass().getSimpleName() +
                        ": " + id);
                try {
                    if(test.apply(inverter) != "fail")
                        System.out.println("Success");
                } catch(Exception | Error e) {
                    System.out.println(
                        "Exception: " + e.getMessage());
                }
            }
        );
    }
    String isEqual(String lval, String rval) {
        if(lval.equals(rval))
```

```
            return "success";
        System.out.println("FAIL: " + lval + " != " + rval);
        return "fail";
    }
    @BeforeAll
    static void startMsg() {
        System.out.println(
            ">>> Starting DynamicStringInverterTests <<<");
    }
    @AfterAll
    static void endMsg() {
        System.out.println(
            ">>> Finished DynamicStringInverterTests <<<");
    }
    @TestFactory
    Stream<DynamicTest> basicInversion1() {
        String in = "Exit, Pursued by a Bear.";
        String out = "eXIT, pURSUED BY A bEAR.";
        return testVersions(
            "Basic inversion (should succeed)",
            inverter -> isEqual(inverter.invert(in), out)
        );
    }
    @TestFactory
    Stream<DynamicTest> basicInversion2() {
        return testVersions(
            "Basic inversion (should fail)",
            inverter -> isEqual(inverter.invert("X"), "X"));
    }
    @TestFactory
    Stream<DynamicTest> disallowedCharacters() {
        String disallowed = ";-_()*&^%$#@!~`0123456789";
        return testVersions(
            "Disallowed characters",
            inverter -> {
                String result = disallowed.chars()
                    .mapToObj(c -> {
                        String cc = Character.toString((char)c);
                        try {
                            inverter.invert(cc);
                            return "";
                        } catch(RuntimeException e) {
                            return cc;
                        }
                    }).collect(Collectors.joining(""));
                if(result.length() == 0)
                    return "success";
                System.out.println("Bad characters: " + result);
                return "fail";
            }
        );
```

```
    }
    @TestFactory
    Stream<DynamicTest> allowedCharacters() {
        String lowcase = "abcdefghijklmnopqrstuvwxyz ,.";
        String upcase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.";
        return testVersions(
            "Allowed characters (should succeed)",
            inverter -> {
                assertEquals(inverter.invert(lowcase), upcase);
                assertEquals(inverter.invert(upcase), lowcase);
                return "success";
            }
        );
    }
    @TestFactory
    Stream<DynamicTest> lengthNoGreaterThan30() {
        String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        assertTrue(str.length() > 30);
        return testVersions(
            "Length must be less than 31 (throws exception)",
            inverter -> inverter.invert(str)
        );
    }
    @TestFactory
    Stream<DynamicTest> lengthLessThan31() {
        String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        assertTrue(str.length() < 31);
        return testVersions(
            "Length must be less than 31 (should succeed)",
            inverter -> inverter.invert(str)
        );
    }
}
```

在一般的测试中，你可能认为在进行一个结果为失败的测试时应该停止代码构建。但是在这里，我们只希望系统报告问题，但仍然继续运行，以便你可以看到不同版本的 **StringInverter** 的效果。

每个使用 **@TestFactory** 注释的方法都会生成一个 **DynamicTest** 对象的 **Stream**（通过 **testVersions()** ），每个 JUnit 都像常规的 **@Test** 方法一样执行。

现在测试都已经准备好了，我们就可以开始实现 **StringInverter** 了。我们从一个仅返回其参数的假的实现类开始：

```
// validating/Inverter1.java
package validating;
public class Inverter1 implements StringInverter {
    public String invert(String str) { return str; }
}
```

接下来我们实现反转操作：

```java
// validating/Inverter2.java
package validating;
import static java.lang.Character.*;
public class Inverter2 implements StringInverter {
    public String invert(String str) {
        String result = "";
        for(int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            result += isUpperCase(c) ?
                      toLowerCase(c) :
                      toUpperCase(c);
        }
        return result;
    }
}
```

现在添加代码以确保输入不超过 30 个字符:

```java
// validating/Inverter3.java
package validating;
import static java.lang.Character.*;
public class Inverter3 implements StringInverter {
    public String invert(String str) {
        if(str.length() > 30)
            throw new RuntimeException("argument too long!");
        String result = "";
        for(int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            result += isUpperCase(c) ?
                      toLowerCase(c) :
                      toUpperCase(c);
        }
        return result;
    }
}
```

最后，我们排除了不允许的字符:

```java
// validating/Inverter4.java
package validating;
import static java.lang.Character.*;
public class Inverter4 implements StringInverter {
    static final String ALLOWED =
        "abcdefghijklmnopqrstuvwxyz ,." +
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    public String invert(String str) {
        if(str.length() > 30)
            throw new RuntimeException("argument too long!");
        String result = "";
        for(int i = 0; i < str.length(); i++) {
```

```
            char c = str.charAt(i);
            if(ALLOWED.indexOf(c) == -1)
                throw new RuntimeException(c + " Not allowed");
            result += isUpperCase(c) ?
                      toLowerCase(c) :
                      toUpperCase(c);
        }
        return result;
    }
}
```

你将从测试输出中看到，每个版本的 **Inverter** 都几乎能通过所有测试。当你在进行测试优先的开发时会有相同的体验。

DynamicStringInverterTests.java 仅是为了显示 TDD 过程中不同 **StringInverter** 实现的开发。通常，你只需编写一组如下所示的测试，并修改单个 **StringInverter** 类直到它满足所有测试：

```
// validating/tests/StringInverterTests.java
package validating;
import java.util.*;
import java.util.stream.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class StringInverterTests {
    StringInverter inverter = new Inverter4();
    @BeforeAll
    static void startMsg() {
        System.out.println(">>> StringInverterTests <<<");
    }
    @Test
    void basicInversion1() {
        String in = "Exit, Pursued by a Bear.";
        String out = "eXIT, pURSUED BY A bEAR.";
        assertEquals(inverter.invert(in), out);
    }
    @Test
    void basicInversion2() {
        expectThrows(Error.class, () -> {
            assertEquals(inverter.invert("X"), "X");
        });
    }
    @Test
    void disallowedCharacters() {
        String disallowed = ";-_()*&^%$#@!~`0123456789";
        String result = disallowed.chars()
            .mapToObj(c -> {
                String cc = Character.toString((char)c);
                try {
                    inverter.invert(cc);
```

```
                    return "";
                } catch(RuntimeException e) {
                    return cc;
                }
            }).collect(Collectors.joining(""));
        assertEquals(result, disallowed);
    }
    @Test
    void allowedCharacters() {
        String lowcase = "abcdefghijklmnopqrstuvwxyz ,.";
        String upcase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.";
        assertEquals(inverter.invert(lowcase), upcase);
        assertEquals(inverter.invert(upcase), lowcase);
    }
    @Test
    void lengthNoGreaterThan30() {
        String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        assertTrue(str.length() > 30);
        expectThrows(RuntimeException.class, () -> {
            inverter.invert(str);
        });
    }
    @Test
    void lengthLessThan31() {
        String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
        assertTrue(str.length() < 31);
        inverter.invert(str);
    }
}
```

你可以通过这种方式进行开发：一开始在测试中建立你期望程序应有的所有特性，然后你就能在实现中一步步添加功能，直到所有测试通过。完成后，你还可以在将来通过这些测试来得知（或让其他任何人得知）当修复错误或添加功能时，代码是否被破坏了。TDD 的目标是产生更好，更周全的测试，因为在完全实现之后尝试实现完整的测试覆盖通常会产生匆忙或无意义的测试。

17.3.1 测试驱动 vs. 测试优先

虽然我自己还没有达到测试优先的意识水平，但我最感兴趣的是来自测试优先中的“测试失败的书签”这一概念。当你离开你的工作一段时间后，重新回到工作进展中，甚至找到你离开时工作到的地方有时会很有挑战性。然而，以失败的测试为书签能让你找到之前停止的地方。这似乎让你能更轻松地暂时离开你的工作，因为不用担心找不到工作进展的位置。

纯粹的测试优先编程的主要问题是它假设你事先了解了你正在解决的问题。根据我自己的经验，我通常是从实验开始，而只有当我处理问题一段时间后，我对它的理解才会达到能给它编写测试的程度。当然，偶尔会有一些问题在你开始之前就已经完全定义，但我个人并不常遇到这些问题。实际上，可能用“*面向测试的开发* (*Test-Oriented Development*)”这个短语来描述编写测试良好的代码或许更好。

17.4 日志

17.4.1 日志会给出正在运行的程序的各种信息。

在调试程序中，日志可以是普通状态数据，用于显示程序运行过程（例如，安装程序可能会记录安装过程中采取的步骤，存储文件的目录，程序的启动值等）。

在调试期间，日志也能带来好处。如果没有日志，你可能会尝试通过插入 **println()** 语句来打印出程序的行为。本书中的一些例子使用了这种技术，并且在没有调试器的情况下（下文中很快就会介绍这样一个主题），它就是你唯一的工具。但是，一旦你确定程序正常运行，你可能会将 **println()** 语句注释或者删除。然而，如果你遇到更多错误，你可能又需要运行它们。因此，如果能够只在需要时轻松启用输出程序状态就好多了。

程序员在日志包可供使用之前，都只能依赖 Java 编译器移除未调用的代码。如果 **debug** 是一个 **static final boolean** ，你就可以这么写：

```
if(debug) {
    System.out.println("Debug info");
}
```

然后，当 **debug** 为 **false** 时，编译器将移除大括号内的代码。因此，未调用的代码不会对运行时产生影响。使用这种方法，你可以在整个程序中放置跟踪代码，并轻松启用和关闭它。但是，该技术的一个缺点是你必须重新编译代码才能启用和关闭跟踪语句。因此，通过更改配置文件来修改日志属性，从而起到启用跟踪语句但不用重新编译程序会更方便。

业内普遍认为标准 Java 发行版本中的日志包 **(java.util.logging)** 的设计相当糟糕。大多数人会选择其他的替代日志包。如 *Simple Logging Facade for Java(SLF4J)* ，它为多个日志框架提供了一个封装好的调用方式，这些日志框架包括 **java.util.logging** ，**logback** 和 **log4j** 。SLF4J 允许用户在部署时插入所需的日志框架。

SLF4J 提供了一个复杂的工具来报告程序的信息，它的效率与前面示例中的技术几乎相同。对于非常简单的信息日志记录，你可以执行以下操作：

```
// validating/SLF4JLogging.java
import org.slf4j.*;
public class SLF4JLogging {
    private static Logger log =
        LoggerFactory.getLogger(SLF4JLogging.class);
    public static void main(String[] args) {
        log.info("hello logging");
    }
}
/* Output:
2017-05-09T06:07:53.418
[main] INFO SLF4JLogging - hello logging
*/
```

日志输出中的格式和信息，甚至输出是否正常或“错误”都取决于 SLF4J 所连接的后端程序包是怎样实现的。在上面的示例中，它连接到的是 **logback** 库（通过本书的 **build.gradle** 文件），并显示为标准输出。

如果我们修改 **build.gradle** 从而使用内置在 JDK 中的日志包作为后端，则输出显示为错误输出，如下所示：

Aug 16, 2016 5:40:31 PM InfoLogging main INFO: hello logging

日志系统会检测日志消息处所在的的类名和方法名。但它不能保证这些名称是正确的，所以不要纠结于其准确性。

17.4.2 日志等级

SLF4J 提供了多个等级的日志消息。下面这个例子以“严重性”的递增顺序对它们作出演示：

```
// validating/SLF4JLevels.java
import org.slf4j.*;
public class SLF4JLevels {
    private static Logger log =
        LoggerFactory.getLogger(SLF4JLevels.class);
    public static void main(String[] args) {
        log.trace("Hello");
        log.debug("Logging");
        log.info("Using");
        log.warn("the SLF4J");
        log.error("Facade");
    }
}
/* Output:
2017-05-09T06:07:52.846
[main] TRACE SLF4JLevels - Hello
2017-05-09T06:07:52.849
[main] DEBUG SLF4JLevels - Logging
2017-05-09T06:07:52.849
[main] INFO SLF4JLevels - Using
2017-05-09T06:07:52.850
[main] WARN SLF4JLevels - the SLF4J
2017-05-09T06:07:52.851
[main] ERROR SLF4JLevels - Facade
*/
```

你可以按等级来查找消息。级别通常设置在单独的配置文件中，因此你可以重新配置而无需重新编译。配置文件格式取决于你使用的后端日志包实现。如 **logback** 使用 XML：

```
<!-- validating/logback.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT"
```

```
        class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>
%d{yyyy-MM-dd'T'HH:mm:ss.SSS}
[%thread] %-5level %logger - %msg%n
            </pattern>
        </encoder>
    </appender>
    <root level="TRACE">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

你可以尝试将 行更改为其他级别，然后重新运行该程序查看日志输出的更改情况。如果你没有写 **logback.xml** 文件，日志系统将采取默认配置。

这只是 SLF4J 最简单的介绍和一般的日志消息，但也足以作为使用日志的基础 - 你可以沿着这个进行更长久的学习和实践。你可以查阅 SLF4J 文档来获得更深入的信息。

17.5 调试

尽管聪明地使用 **System.out** 或日志信息能给我们带来对程序行为的有效见解，但对于困难问题来说，这种方式就显得笨拙且耗时了。

你也可能需要更加深入地理解程序，仅依靠打印日志做不到。此时你需要调试器。除了比打印语句更快更轻易地展示信息以外，调试器还可以设置断点，并在程序运行到这些断点处暂停程序。

使用调试器，可以展示任何时刻的程序状态，查看变量的值，一步一步运行程序，连接远程运行的程序等等。特别是当你构建较大规模的系统（bug 容易被掩埋）时，熟练使用调试器是值得的。

17.5.1 使用 JDB 调试

Java 调试器（JDB）是 JDK 内置的命令行工具。从调试的指令和命令行接口两方面看的话，JDB 至少从概念上是 GNU 调试器（GDB，受 Unix DB 的影响）的继承者。JDB 对于学习调试和执行简单的调试任务来说是有用的，而且知道只要安装了 JDK 就可以使用 JDB 是有帮助的。然而，对于大型项目来说，你可能想要一个图形化的调试器，这在后面会描述。

假设你写了如下程序：

```
// validating/SimpleDebugging.java
// {ThrowsException}
public class SimpleDebugging {
    private static void foo1() {
        System.out.println("In foo1");
        foo2();
    }
```

```java

    private static void foo2() {
        System.out.println("In foo2");
        foo3();
    }

    private static void foo3() {
        System.out.println("In foo3");
        int j = 1;
        j--;
        int i = 5 / j;
    }

    public static void main(String[] args) {
        foo1();
    }
}
/* Output
In foo1
In foo2
In foo3
__[Error Output]__
Exception in thread "main"
java.lang.ArithmeticException: /by zero
at
SimpleDebugging.foo3(SimpleDebugging.java:17)
at
SimpleDebugging.foo2(SimpleDebugging.java:11)
at
SimpleDebugging.foo1(SimpleDebugging.java:7)
at
SimpleDebugging.main(SimpleDebugging.java:20)
```

首先看方法 `foo3()`，问题很明显：除数是 0。但是假如这段代码被埋没在大型程序中（像这里的调用序列暗示的那样）而且你不知道从哪儿开始查找问题。结果呢，异常会给出足够的信息让你定位问题。然而，假设事情更加复杂，你必须更加深入程序中来获得比异常提供的更多的信息。

为了运行 JDB，你需要在编译 **SimpleDebugging.java** 时加上 **-g** 标记，从而告诉编译器生成编译信息。然后使用如下命令开始调试程序：

jdb SimpleDebugging

接着 JDB 就会运行，出现命令行提示。你可以输入 **?** 查看可用的 JDB 命令。

这里展示了如何使用交互式追踪一个问题的调试历程：

Initializing jdb…

> catch Exception

> 表明 JDB 在等待输入命令。命令 **catch Exception** 在任何抛出异常的地方设置断点（然而，即使你不显式地设置断点，调试器也会停止— JDB 中好像是默认在异常抛出处设置了异常）。接着命令行

你一定在用错误的版本。

会给出如下响应:

Deferring exception catch Exception.

It will be set after the class is loaded.

继续输入:

> run

现在程序将运行到下个断点处，在这个例子中就是异常发生的地方。下面是运行 **run** 命令的结果:

run SimpleDebugging

Set uncaught java.lang.Throwable

Set deferred uncaught java.lang.Throwable

>

VM Started: In foo1

In foo2

In foo3

Exception occurred: java.lang.ArithmeticException

(uncaught)“thread=main”,

SimpleDebugging.foo3(),line=16 bci=15

16 int i = 5 / j

程序运行到第 16 行时发生异常，但是 JDB 在异常发生时就不复存在。调试器还展示了是哪一行导致了异常。你可以使用 **list** 将导致程序终止的执行点列出来:

main1 list

12 private static void foo3() {

13 System.out.println(“In foo3”);

14 int j = 1;

15 j–;

16 => int i = 5 / j;

17 }

18 public static void main(String[] args) {

19 foo1();

20 }

21 }

/* Output:

上述 => 展示了程序将继续运行的执行点。你可以使用命令 **cont**(continue) 继续运行，但是会导致 JDB 在异常发生时退出并打印出栈轨迹信息。

命令 **locals** 能转储所有的局部变量值:

main1 locals
Method arguments:
Local variables:
j = 0

命令 **wherei** 打印进入当前线程的方法栈中的栈帧信息:

main1 wherei
1 SimpleDebugging.foo3(SimpleDebugging.java:16), pc =15
2 SimpleDebugging.foo2(SimpleDebugging.java:10), pc = 8
[3] SimpleDebugging.foo1(SimpleDebugging.java:6), pc = 8
[4] SimpleDebugging.main(SimpleDebugging.java:19), pc = 10

wherei 后的每一行代表一个方法调用和调用返回点（由程序计数器显示数值）。这里的调用序列是 **main()**, **foo1()**, **foo2()** 和 **foo3()**。

因为命令 **list** 展示了执行停止的地方，所以你通常有足够的信息得知发生了什么并修复它。命令 **help** 将会告诉你更多关于 **jdb** 的用法，但是在花更多的时间学习它之前必须明白命令行调试器往往需要花费更多的精力得到结果。使用 **jdb** 学习调试的基础部分，然后转而学习图形界面调试器。

17.5.2 图形化调试器

使用类似 JDB 的命令行调试器是不方便的。它需要显式的命令去查看变量的状态 (**locals**, **dump**)，列出源代码中的执行点 (**list**)，查找系统中的线程 (**threads**)，设置断点 (**stop in**, **stop at**) 等等。使用图形化调试器只需要点击几下，不需要使用显式的命令就能使用这些特性，而且能查看被调试程序的最新细节。

因此，尽管你可能一开始用 JDB 尝试调试，但是你将发现使用图形化调试器能更加高效、更快速地追踪 bug。IBM 的 Eclipse，Oracle 的 NetBeans 和 JetBrains 的 IntelliJ 这些集成开发环境都含有面向 Java 语言的好用的图形化调试器。

17.6 基准测试

> 我们应该忘掉微小的效率提升，说的就是这些 97% 的时间做的事：过早的优化是万恶之源。
>
> —— Donald Knuth

如果你发现自己正在过早优化的滑坡上，你可能浪费了几个月的时间 (如果你雄心勃勃的话)。通常，一个简单直接的编码方法就足够好了。如果你进行了不必要的优化，就会使你的代码变得无谓的复杂和难以理解。

基准测试意味着对代码或算法片段进行计时看哪个跑得更快，与下一节的分析和优化截然相反，分析优化是观察整个程序，找到程序中最耗时的部分。

可以简单地对一个代码片段的执行计时吗？在像 C 这样直接的编程语言中，这个方法的确可行。在像 Java 这样拥有复杂的运行时系统的编程语言中，基准测试变得更有挑战性。为了生成可靠的数据，环境设置必须控制诸如 CPU 频率，节能特性，其他运行在相同机器上的进程，优化器选项等等。

17.6.1 微基准测试

写一个计时工具类从而比较不同代码块的执行速度是具有吸引力的。看上去这会产生一些有用的数据。比如，这里有一个简单的 **Timer** 类，可以用以下两种方式使用它：

1. 创建一个 **Timer** 对象，执行一些操作然后调用 **Timer** 的 **duration()** 方法产生以毫秒为单位的运行时间。
2. 向静态的 **duration()** 方法中传入 **Runnable**。任何符合 **Runnable** 接口的类都有一个函数式方法 **run()**，该方法没有入参，且没有返回。

```
// onjava/Timer.java
package onjava;
import static java.util.concurrent.TimeUnit.*;

public class Timer {
    private long start = System.nanoTime();

    public long duration() {
        return NANOSECONDS.toMillis(System.nanoTime() - start);
    }

    public static long duration(Runnable test) {
        Timer timer = new Timer();
        test.run();
        return timer.duration();
    }
}
```

这是一个很直接的计时方式。难道我们不能只运行一些代码然后看它的运行时长吗？

有许多因素会影响你的结果，即使是生成提示符也会造成计时的混乱。这里举一个看上去天真的例子，它使用了标准的 Java **Arrays** 库（后面会详细介绍）：

```
// validating/BadMicroBenchmark.java
// {ExcludeFromTravisCI}
import java.util.*;
import onjava.Timer;

public class BadMicroBenchmark {
    static final int SIZE = 250_000_000;

    public static void main(String[] args) {
        try { // For machines with insufficient memory
```

```
            long[] la = new long[SIZE];
            System.out.println("setAll: " + Timer.duration(() -> Arrays
                .setAll(la, n -> n)));
            System.out.println("parallelSetAll: " + Timer.duration(()
                -> Arrays.parallelSetAll(la, n -> n)));
        } catch (OutOfMemoryError e) {
            System.out.println("Insufficient memory");
            System.exit(0);
        }
    }

}
/* Output
setAll: 272
parallelSetAll: 301
```

main() 方法的主体包含在 **try** 语句块中，因为一台机器用光内存后会导致构建停止。

对于一个长度为 250,000,000 的 **long** 型（仅仅差一点就会让大部分机器内存溢出）数组，我们比较了 **Arrays.setAll()** 和 **Arrays.parallelSetAll()** 的性能。这个并行的版本会尝试使用多个处理器加快完成任务（尽管我在这一节谈到了一些并行的概念，但是在 并发编程 章节我们才会详细讨论这些）。然而非并行的版本似乎运行得更快，尽管在不同的机器上结果可能不同。

BadMicroBenchmark.java 中的每一步操作都是独立的，但是如果你的操作依赖于同一资源，那么并行版本运行的速度会骤降，因为不同的进程会竞争相同的那个资源。

```
// validating/BadMicroBenchmark2.java
// Relying on a common resource

import java.util.*;
import onjava.Timer;

public class BadMicroBenchmark2 {
    static final int SIZE = 5_000_000;

    public static void main(String[] args) {
        long[] la = new long[SIZE];
        Random r = new Random();
        System.out.println("parallelSetAll: " + Timer.duration(() ->
            Arrays.parallelSetAll(la, n -> r.nextLong())));
        System.out.println("setAll: " + Timer.duration(() -> Arrays.
            setAll(la, n -> r.nextLong())));
        SplittableRandom sr = new SplittableRandom();
        System.out.println("parallelSetAll: " + Timer.duration(() ->
            Arrays.parallelSetAll(la, n -> sr.nextLong())));
        System.out.println("setAll: " + Timer.duration(() -> Arrays.
            setAll(la, n -> sr.nextLong())));
    }
}
/* Output
```

```
parallelSetAll: 1147
setAll: 174
parallelSetAll: 86
setAll: 39
```

SplittableRandom 是为并行算法设计的，它当然看起来比普通的 **Random** 在 **parallelSetAll()** 中运行得更快。但是看上去还是比非并发的 **setAll()** 运行时间更长，有点难以置信（也许是真的，但我们不能通过一个坏的微基准测试得到这个结论）。

这只考虑了微基准测试的问题。Java 虚拟机 Hotspot 也非常影响性能。如果你在测试前没有通过运行代码给 JVM 预热，那么你就会得到“冷”的结果，不能反映出代码在 JVM 预热之后的运行速度（假如你运行的应用没有在预热的 JVM 上运行，你就可能得不到所预期的性能，甚至可能减缓速度）。

优化器有时可以检测出你创建了没有使用的东西，或者是部分代码的运行结果对程序没有影响。如果它优化掉你的测试，那么你可能得到不好的结果。

一个良好的微基准测试系统能自动地弥补像这样的问题（和很多其他的问题）从而产生合理的结果，但是创建这么一套系统是非常棘手，需要深入的知识。

17.6.2 JMH 的引入

截止目前为止，唯一能产生像样结果的 Java 微基准测试系统就是 Java Microbenchmarking Harness，简称 JMH。本书的 **build.gradle** 自动引入了 JMH 的设置，所以你可以轻松地使用它。

你可以在命令行编写 JMH 代码并运行它，但是推荐的方式是让 JMH 系统为你运行测试；**build.gradle** 文件已经配置成只需要一条命令就能运行 JMH 测试。

JMH 尝试使基准测试变得尽可能简单。例如，我们将使用 JMH 重新编写 **BadMicroBenchmark.java**。这里只有 **@State** 和 **@Benchmark** 这两个注解是必要的。其余的注解要么是为了产生更多易懂的输出，要么是加快基准测试的运行速度（JMH 基准测试通常需要运行很长时间）：

```java
// validating/jmh/JMH1.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
// Increase these three for more accuracy:
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH1 {
    private long[] la;

    @Setup
```

```
    public void setup() {
        la = new long[250_000_000];
    }

    @Benchmark
    public void setAll() {
        Arrays.setAll(la, n -> n);
    }

    public void parallelSetAll() {
        Arrays.parallelSetAll(la, n -> n);
    }
}
```

“forks” 的默认值是 10，意味着每个测试都运行 10 次。为了减少运行时间，这里使用了 **@Fork** 注解来减少这个次数到 1。我还使用了 **@Warmup** 和 **@Measurement** 注解将它们默认的运行次数从 20 减少到 5 次。尽管这降低了整体的准确率，但是结果几乎与使用默认值相同。可以尝试将 **@Warmup**、**@Measurement** 和 **@Fork** 都注释掉然后看使用它们的默认值，结果会有多大显著的差异；一般来说，你应该只能看到长期运行的测试使错误因素减少，而结果没有多大变化。

需要使用显式的 gradle 命令才能运行基准测试（在示例代码的根目录处运行）。这能防止耗时的基准测试运行其他的 **gradlew** 命令：

gradlew validating:jmh

这会花费几分钟的时间，取决于你的机器 (如果没有注解上的调整，可能需要几个小时)。控制台会显示 **results.txt** 文件的路径，这个文件统计了运行结果。注意，**results.txt** 包含这一章所有 **jmh** 测试的结果：**JMH1.java**，**JMH2.java** 和 **JMH3.java**。

因为输出是绝对时间，所以在不同的机器和操作系统上结果各不相同。重要的因素不是绝对时间，我们真正观察的是一个算法和另一个算法的比较，尤其是哪一个运行得更快，快多少。如果你在自己的机器上运行测试，你将看到不同的结果却有着相同的模式。

我在大量的机器上运行了这些测试，尽管不同的机器上得到的绝对值结果不同，但是相对值保持着合理的稳定性。我只列出了 **results.txt** 中适当的片段并加以编辑使输出更加易懂，而且内容大小适合页面。所有测试中的 **Mode** 都以 **avgt** 展示，代表 “平均时长”。**Cnt**（测试的数目）的值是 200，尽管这里的一个例子中配置的 **Cnt** 值是 5。**Units** 是 **us/op**，是 “Microseconds per operation” 的缩写，因此，这个值越小代表性能越高。

我同样也展示了使用 warmups、measurements 和 forks 默认值的输出。我删除了示例中相应的注解，就是为了获取更加准确的测试结果（这将花费数小时）。结果中数字的模式应该仍然看起来相同，不论你如何运行测试。

下面是 **JMH1.java** 的运行结果：

Benchmark Score

JMH1.setAll 196280.2

JMH1.parallelSetAll 195412.9

即使像 JMH 这么高级的基准测试工具，基准测试的过程也不容易，练习时需要倍加小心。这里测试产生了反直觉的结果：并行的版本 **parallelSetAll()** 花费了与非并行版本的 **setAll()** 相同的时间，两者似乎都运行了相当长的时间。

当创建这个示例时，我假设如果我们要测试数组初始化的话，那么使用非常大的数组是有意义的。所以我选择了尽可能大的数组；如果你实验的话会发现一旦数组的大小超过 2 亿 5000 万，你就开始会得到内存溢出的异常。然而，在这么大的数组上执行大量的操作从而震荡内存系统，产生无法预料的结果是有可能的。不管这个假设是否正确，看上去我们正在测试的并非是我们想测试的内容。

考虑其他的因素：

C：客户端执行操作的线程数量

P：并行算法使用的并行数量

N：数组的大小：**10^(2*k)，**通常来说，**k=1..7** 足够来练习不同的缓存占用。

Q：setter 的操作成本

这个 C/P/N/Q 模型在早期 JDK 8 的 Lambda 开发期间付出水面，大多数并行的 Stream 操作 (**parallelSetAll()** 也基本相似) 都满足这些结论：**N*Q**(主要工作量) 对于并发性能尤为重要。并行算法在工作量较少时可能实际运行得更慢。

在一些情况下操作竞争如此激烈使得并行毫无帮助，而不管 **N*Q 有多大。当 C 很大时，P 就变得不太相关（内部并行在大量的外部并行面前显得多余）。此外，在一些情况下，并行分解会让相同的 C** 个客户端运行得比它们顺序运行代码更慢。

基于这些信息，我们重新运行测试，并在这些测试中使用不同大小的数组（改变 **N**）：

```
// validating/jmh/JMH2.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH2 {

    private long[] la;

    @Param({
            "1",
            "10",
            "100",
            "1000",
```

```
            "10000",
            "100000",
            "1000000",
            "10000000",
            "100000000",
            "250000000"
    })
    int size;

    @Setup
    public void setup() {
        la = new long[size];
    }

    @Benchmark
    public void setAll() {
        Arrays.setAll(la, n -> n);
    }

    @Benchmark
    public void parallelSetAll() {
        Arrays.parallelSetAll(la, n -> n);
    }
}
```

@Param 会自动地将其自身的值注入到变量中。其自身的值必须是字符串类型，并可以转化为适当的类型，在这个例子中是 **int** 类型。

下面是已经编辑过的结果，包含精确计算出的加速数值：

JMH2 Benchmark	Size	Score %	Speedup
setAll	1	0.001	
parallelSetAll	1	0.036	0.028
setAll	10	0.005	
parallelSetAll	10	3.965	0.001
setAll	100	0.031	
parallelSetAll	100	3.145	0.010
setAll	1000	0.302	
parallelSetAll	1000	3.285	0.092
setAll	10000	3.152	
parallelSetAll	10000	9.669	0.326

JMH2 Benchmark	Size	Score %	Speedup
setAll	100000	34.971	
parallelSetAll	100000	20.153	1.735
setAll	1000000	420.581	
parallelSetAll	1000000	165.388	2.543
setAll	10000000	8160.054	
parallelSetAll	10000000	7610.190	1.072
setAll	100000000	79128.752	
parallelSetAll	100000000	76734.671	1.031
setAll	250000000	199552.121	
parallelSetAll	250000000	191791.927	1.040

可以看到当数组大小达到 10 万左右时，**parallelSetAll()** 开始反超，而后趋于与非并行的运行速度相同。即使它运行速度上胜了，看起来也不足以证明由于并行的存在而使速度变快。

setAll()/parallelSetAll() 中工作的计算量起很大影响吗？在前面的例子中，我们所做的只有对数组的赋值操作，这可能是最简单的任务。所以即使 **N** 值变大，**N*Q **也仍然没有达到巨大，所以看起来像是我们没有为并行提供足够的机会（JMH 提供了一种模拟变量 Q 的途径；如果想了解更多的话，可搜索** Blackhole.consumeCPU**）。

我们通过使方法 **f()** 中的任务变得更加复杂，从而产生更多的并行机会：

```
// validating/jmh/JMH3.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH3 {
    private long[] la;

    @Param({
            "1",
```

```
            "10",
            "100",
            "1000",
            "10000",
            "100000",
            "1000000",
            "10000000",
            "100000000",
            "250000000"
    })
    int size;

    @Setup
    public void setup() {
        la = new long[size];
    }

    public static long f(long x) {
        long quadratic = 42 * x * x + 19 * x + 47;
        return Long.divideUnsigned(quadratic, x + 1);
    }

    @Benchmark
    public void setAll() {
        Arrays.setAll(la, n -> f(n));
    }

    @Benchmark
    public void parallelSetAll() {
        Arrays.parallelSetAll(la, n -> f(n));
    }
}
```

f() 方法提供了更加复杂且耗时的操作。现在除了简单的给数组赋值外，**setAll()** 和 **parallelSetAll()** 都有更多的工作去做，这肯定会影响结果。

JMH2 Benchmark	Size	Score %	Speedup
setAll	1	0.012	
parallelSetAll	1	0.047	0.255
setAll	10	0.107	
parallelSetAll	10	3.894	0.027
setAll	100	0.990	
parallelSetAll	100	3.708	0.267

JMH2 Benchmark	Size	Score %	Speedup
setAll	1000	133.814	
parallelSetAll	1000	11.747	11.391
setAll	10000	97.954	
parallelSetAll	10000	37.259	2.629
setAll	100000	988.475	
parallelSetAll	100000	276.264	3.578
setAll	1000000	9203.103	
parallelSetAll	1000000	2826.974	3.255
setAll	10000000	92144.951	
parallelSetAll	10000000	28126.202	3.276
setAll	100000000	921701.863	
parallelSetAll	100000000	266750.543	3.455
setAll	250000000	2299127.273	
parallelSetAll	250000000	538173.425	4.272

可以看到当数组的大小达到 1000 左右时，**parallelSetAll()** 的运行速度反超了 **setAll()**。看来 **parallelSetAll()** 严重依赖数组中计算的复杂度。这正是基准测试的价值所在，因为我们已经得到了关于 **setAll()** 和 **parallelSetAll()** 间微妙的信息，知道在何时使用它们。

这显然不是从阅读 Javadocs 就能得到的。

大多数时候，JMH 的简单应用会产生好的结果（正如你将在本书后面例子中所见），但是我们从这里知道，你不能一直假定 JMH 会产生好的结果。JMH 网站上的范例可以帮助你开始。

17.7 剖析和优化

有时你必须检测程序运行时间花在哪儿，从而看是否可以优化那一块的性能。剖析器可以找到这些导致程序慢的地方，因而你可以找到最轻松，最明显的方式加快程序运行速度。

剖析器收集的信息能显示程序哪一部分消耗内存，哪个方法最耗时。一些剖析器甚至能关闭垃圾回收，从而帮助限定内存分配的模式。

人呐，眼光放得长远一些，看到的东西也会多一些，生活也就会过得更有意义一点。

剖析器还可以帮助检测程序中的线程死锁。注意剖析和基准测试的区别。剖析关注的是已经运行在真实数据上的整个程序，而基准测试关注的是程序中隔离的片段，通常是去优化算法。

安装 Java 开发工具包（JDK）时会顺带安装一个虚拟的剖析器，叫做 **VisualVM**。它会被自动安装在与 **javac** 相同的目录下，你的执行路径应该已经包含该目录。启动 VisualVM 的控制台命令是：

> jvisualvm

运行该命令后会弹出一个窗口，其中包括一些指向帮助信息的链接。

17.7.1 优化准则

- 避免为了性能牺牲代码的可读性。
- 不要独立地看待性能。衡量与带来的收益相比所需投入的工作量。
- 程序的大小很重要。性能优化通常只对运行了长时间的大型项目有价值。性能通常不是小项目的关注点。
- 运行起来程序比一心钻研它的性能具有更高的优先级。一旦你已经有了可工作的程序，如有必要的话，你可以使用剖析器提高它的效率。只有当性能是关键因素时，才需要在设计/开发阶段考虑性能。
- 不要猜测瓶颈发生在哪。运行剖析器，让剖析器告诉你。
- 无论何时有可能的话，显式地设置实例为 null 表明你不再用它。这对垃圾收集器来说是个有用的暗示。
- **static final** 修饰的变量会被 JVM 优化从而提高程序的运行速度。因而程序中的常量应该声明 **static final**。

17.8 风格检测

当你在一个团队中工作时 (包括尤其是开源项目)，让每个人遵循相同的代码风格是非常有帮助的。这样阅读项目的代码时，不会因为风格的不同产生思维上的中断。然而，如果你习惯了某种不同的代码风格，那么记住项目中所有的风格准则对你来说可能是困难的。幸运的是，存在可以指出你代码中不符合风格准则的工具。

一个流行的风格检测器是 **Checkstyle**。查看本书 示例代码 中的 **gradle.build** 和 **checkstyle.xml** 文件中配置代码风格的方式。checkstyle.xml 是一个常用检测的集合，其中一些检测被注释掉了以允许使用本书中的代码风格。

运行所有风格检测的命令是：

gradlew checkstyleMain

一些文件仍然产生了风格检测警告，通常是因为这些例子展示了你在生产代码中不会使用的样例。

你还可以针对一个具体的章节运行代码检测。例如，下面命令会运行 Annotations 章节的风格检测：

gradlew annotations:checkstyleMain

17.9 静态错误分析

尽管 Java 的静态类型检测可以发现基本的语法错误，其他的分析工具可以发现躲避 **javac** 检测的更加复杂的 bug。一个这样的工具叫做 **Findbugs**。本书 示例代码 中的 **build.gradle** 文件包含了 Findbugs 的配置，所以你可以输入如下命令：

gradlew findbugsMain

这会为每一章生成一个名为 **main.html** 的报告，报告中会说明代码中潜在的问题。Gradle 命令的输出会告诉你每个报告在何处。

当你查看报告时，你将会看到很多 false positive 的情况，即代码没问题却报告了问题。我在一些文件中展示了不要做一些事的代码确实是正确的。

当我最初看到本书的 Findbugs 报告时，我发现了一些不是技术错误的地方，但能使我改善代码。如果你正在寻找 bug，那么在调试之前运行 Findbugs 是值得的，因为这将可能节省你数小时的时间找到问题。

17.10 代码重审

单元测试能找到明显重要的 bug 类型，风格检测和 Findbugs 能自动执行代码重审，从而发现额外的问题。最终你走到了必须人为参与进来的地步。代码重审是一个或一群人的一段代码被另一个或一群人阅读和评估的众多方式之一。这最初看起来会使人不安，而且需要情感信任，但它的目的肯定不是羞辱任何人。它的目标是找到程序中的错误，代码重审是最成功的能做到这点的途径之一。可惜的是，它们也经常被认为是“过于昂贵的”（有时这会成为程序员避免代码被重审时感到尴尬的借口）。

代码重审可以作为结对编程的一部分，作为代码签入过程的一部分（另一个程序员自动安排上审查新代码的任务）或使用群组预排的方式，即每个人阅读代码并讨论之。后一种方式对于分享知识和营造代码文化是极其有益的。

17.11 结对编程

结对编程是指两个程序员一起编程的实践活动。通常来说，一个人“驱动”（敲击键盘，输入代码），另一人（观察者或指引者）重审和分析代码，同时也要思考策略。这产生了一种实时的代码重审。通常程序员会定期地互换角色。

任何傻瓜都能写出计算机可以理解的代码。好的程序员能写出人能读懂的代码。

结对编程有很多好处，但最显著的是分享知识和防止阻塞。最佳传递信息的方式之一就是一起解决问题，我已经在很多次研讨会使用了结对编程，都取得了很好的效果（同时，研讨会上的众人可以通过这种方式互相了解对方）。而且两个人一起工作时，可以更容易地推进开发的进展，而只有一个程序员的话，可能被轻易地卡住。结对编程的程序员通常可以从工作中感到更高的满足感。有时很难向管理人员们推行结对编程，因为他们可能觉得两个程序员解决同一个问题的效率比他们分开解决不同问题的效率低。尽管短期内是这样，但是结对编程能带来更高的代码质量；除了结对编程的其他益处，如果你眼光长远的话，这会产生更高的生产力。

维基百科上这篇 结对编程的文章 可以作为你深入了解结对编程的开始。

17.12 重构

技术负债是指迭代发展的软件中为了应急而生的丑陋解决方案从而导致设计难以理解，代码难以阅读的部分。特别是当你必须修改和增加新特性的时候，这会造成麻烦。

重构可以矫正技术负债。重构的关键是它能改善代码设计，结构和可读性（因而减少代码负债），但是它不能改变代码的行为。

很难向管理人员推行重构："我们将投入很多工作不是增加新的特性，当我们完成时，外界无感知变化。但是相信我们，事情会变得更加美好"。不幸的是，管理人员意识到重构的价值时都为时已晚了：当他们提出增加新的特性时，你不得不告诉他们做不到，因为代码基底已经埋藏了太多的问题，试图增加新特性可能会使软件崩溃，即使你能想出怎么做。

17.12.1 重构基石

在开始重构代码之前，你需要有以下三个系统的支撑：

1. 测试（通常，JUnit 测试作为最小的根基），因此你能确保重构不会改变代码的行为。
2. 自动构建，因而你能轻松地构建代码，运行所有的测试。通过这种方式做些小修改并确保修改不会破坏任何事物是毫不费力的。本书使用的是 Gradle 构建系统，你可以在 代码示例 的 **build.gradle** 文件中查看示例。
3. 版本控制，以便你能回退到可工作的代码版本，能够一直记录重构的每一步。

本书的代码托管在 Github 上，使用的是 **git** 版本控制系统。

没有这三个系统的支持，重构几乎是不可能的。确实，没有这些系统，起初维护和增加代码是一个巨大的挑战。令人意外的是，有很多成功的公司竟然在没有这三个系统的情况下在相当长的时间里勉强过得去。然而，对于这样的公司来说，在他们遇到严重的问题之前，这只是个时间问题。

维基百科上的 重构文章 提供了更多的细节。

17.13 持续集成

在软件开发的早期，人们只能一次处理一步，所以他们坚信他们总是在经历快乐之旅，每个开发阶段无缝进入下一个。这种错觉经常被称为软件开发中的“瀑布流模型”。很多人告诉我瀑布流是他们的选择方法，好像这是一个选择工具，而不仅是一厢情愿。

在这片童话的土地上，每一步都按照指定的预计时间准时完美结束，然后下一步开始。当最后一步结束时，所有的部件都可以无缝地滑在一起，瞧，一个装载产品诞生了！

当然，现实中没有事能按计划或预计时间运作。相信它应该，然后当它不能时更加相信，只会使整件事变得更糟。否认证据不会产生好的结果。

除此之外，产品本身经常也不是对客户有价值的事物。有时一大堆的特性完全是浪费时间，因为创造出这些特性需求的人不是客户而是其他人。

因为受流水工作线的思路影响，所以每个开发阶段都有自己的团队。上游团队的延期传递到下游团队，当到了需要进行测试和集成的时候，这些团队被指望赶上预期时间，当他们必然做不到时，就认为他们是“差劲的团队成员”。不可能的时间安排和负相关的结合产生了自实现的预期：只有最绝望的开发者才会乐意做这些工作。

另外，商学院培养出的管理人员仍然被训练成只在已有的流程上做一些改动——这些流程都是基于工业时代制造业的想法上。注重培养创造力而不是墨守成规的商学院仍然很稀有。终于一些编程领域的人们再也忍受不了这种情况并开始进行实验。最初一些实验叫做“极限编程”，因为它们与工业时代的思想完全不同。随着实验展示的结果，这些思想开始看起来像是常识。这些实验逐渐形成了如今显而易见的观点——尽管非常小——即把生产可运作的产品交到客户手中，询问他们 (A) 是否想要它 (B) 是否喜欢它工作的方式 (C) 还希望有什么其他有用的功能特性。然后这些信息反馈给开发，从而继续产出一个新版本。版本不断迭代，项目最终演变成为客户带来真正价值的事物。

这完全颠倒了瀑布流开发的方式。你停止假设你要处理产品测试和把部署“作为最后一步”这类的事情。相反，每件事从开始到结束必须都在进行——即使一开始产品几乎没有任何特性。这么做对于在开发周期的早期发现更多问题有巨大的益处。此外，不是做大量宏大超前的计划和花费时间金钱在许多无用的特性上，而是一直都能从顾客那得到反馈。当客户不再需要其他特性时，你就完成了。这节省了大量的时间和金钱，并提高了顾客的满意度。

有许多不同的想法导向这种方式，但是目前首要的术语叫持续集成（CI）。CI 与导向 CI 的想法之间的不同之处在于 CI 是一种独特的机械式的过程，过程中涵盖了这些想法；它是一种定义好的做事方式。事实上，它定义得如此明确以至于整个过程是自动化的。

当前 CI 技术的高峰是持续集成服务器。这是一台独立的机器或虚拟机，通常是由第三方公司托管的完全独立的服务。这些公司通常免费提供基本服务，如果你需要额外的特性如更多的处理器或内存或者专门的工具或系统，你需要付费。CI 服务器起初是完全空白状态，即只是可用的操作系统的最小配置。这很重要因为你可能之前在你的开发机器上安装过一些程序，却没有在你的构建和部署系统中包含它。正如重构一样，持续集成需要分布式版本管理，自动构建和自动测试系统作为基础。通常来说，CI

服务器会绑定到你的版本控制仓库上。当 CI 服务器发现仓库中有改变时，就会拉取最新版本的代码，并按照 CI 脚本中的过程处理。这包括安装所有必要的工具和类库（记住，CI 服务器起初只有一个干净、基本的操作系统），所以如果过程中出现任何问题，你都可以发现它们。接着它会执行脚本中定义的构建和测试操作；通常脚本中使用的命令与人们在安装和测试中使用的命令完全相同。如果执行成功或失败，CI 服务器会有许多种方式汇报给你，包括在你的代码仓库上显示一个简单的标记。

使用持续集成，每次你合进仓库时，这些改变都会被从头到尾验证。通过这种方式，一旦出现问题你能立即发现。甚至当你准备交付一个产品的新版本时，都不会有延迟或其他必要的额外步骤（在任何时刻都可以交付叫做持续交付）。

本书的示例代码都是在 Travis-CI(基于 Linux 的系统) 和 AppVeyor(Windows) 上自动测试的。你可以在 Gihub 仓库 上的 Readme 看到通过/失败的标记。

17.14 本章小结

“它在我的机器上正常工作了。”“我们不会运载你的机器！”

代码校验不是单一的过程或技术。每种方法只能发现特定类型的 bug，作为程序员的你在开发过程中会明白每个额外的技术都能增加代码的可靠性和鲁棒性。校验不仅能在开发过程中，还能在为应用添加新功能的整个项目期间帮你发现更多的错误。现代化开发意味着比仅仅编写代码更多的内容，每种你在开发过程中融入的测试技术—— 包括而且尤其是你创建的能适应特定应用的自定义工具——都会带来更好、更快和更加愉悦的开发过程，同时也能为客户提供更高的价值和满意度体验。

18 文件

> 在丑陋的 Java I/O 编程方式诞生多年以后，Java 终于简化了文件读写的基本操作。

这种“困难方式”的全部细节都在 Appendix: I/O Streams。如果你读过这个部分，就会认同 Java 设计者毫不在意他们的使用者的体验这一观念。打开并读取文件对于大多数编程语言来是非常常用的，由于 I/O 糟糕的设计以至于很少有人能够在不依赖其他参考代码的情况下完成打开文件的操作。

好像 Java 设计者终于意识到了 Java 使用者多年来的痛苦，在 Java7 中对此引入了巨大的改进。这些新元素被放在 **java.nio.file** 包下面，过去人们通常把 **nio** 中的 **n** 理解为 **new** 即新的 **io**，现在更应该当成是 **non-blocking** 非阻塞 **io**(**io** 就是 *input/output 输入/输出*)。**java.nio.file** 库终于将 Java 文件操作带到与其他编程语言相同的水平。最重要的是 Java8 新增的 streams 与文件结合使得文件操作编程变得更加优雅。我们将看一下文件操作的两个基本组件：

1. 文件或者目录的路径；
2. 文件本身。

18.1 文件和目录路径

一个 **Path** 对象表示一个文件或者目录的路径，是一个跨操作系统（OS）和文件系统的抽象，目的是在构造路径时不必关注底层操作系统，代码可以在不进行修改的情况下运行在不同的操作系统上。**java.nio.file.Paths** 类包含一个重载方法 **static get()**，该方法接受一系列 **String** 字符串或一个*统一资源标识符* (URI) 作为参数，并且进行转换返回一个 **Path** 对象：

```
// files/PathInfo.java
import java.nio.file.*;
import java.net.URI;
import java.io.File;
import java.io.IOException;

public class PathInfo {
    static void show(String id, Object p) {
        System.out.println(id + ": " + p);
    }

    static void info(Path p) {
        show("toString", p);
```

```
        show("Exists", Files.exists(p));
        show("RegularFile", Files.isRegularFile(p));
        show("Directory", Files.isDirectory(p));
        show("Absolute", p.isAbsolute());
        show("FileName", p.getFileName());
        show("Parent", p.getParent());
        show("Root", p.getRoot());
        System.out.println("******************");
    }
    public static void main(String[] args) {
        System.out.println(System.getProperty("os.name"));
        info(Paths.get("C:", "path", "to", "nowhere", "NoFile.txt"));
        Path p = Paths.get("PathInfo.java");
        info(p);
        Path ap = p.toAbsolutePath();
        info(ap);
        info(ap.getParent());
        try {
            info(p.toRealPath());
        } catch(IOException e) {
           System.out.println(e);
        }
        URI u = p.toUri();
        System.out.println("URI: " + u);
        Path puri = Paths.get(u);
        System.out.println(Files.exists(puri));
        File f = ap.toFile(); // Don't be fooled
    }
}

/* 输出:
Windows 10
toString: C:\path\to\nowhere\NoFile.txt
Exists: false
RegularFile: false
Directory: false
Absolute: true
FileName: NoFile.txt
Parent: C:\path\to\nowhere
Root: C:\
******************
toString: PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: false
FileName: PathInfo.java
Parent: null
Root: null
******************
toString: C:\Users\Bruce\Documents\GitHub\onjava\
```

```
ExtractedExamples\files\PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: true
FileName: PathInfo.java
Parent: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files
Root: C:\
******************
toString: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files
Exists: true
RegularFile: false
Directory: true
Absolute: true
FileName: files
Parent: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples
Root: C:\
******************
toString: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files\PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: true
FileName: PathInfo.java
Parent: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files
Root: C:\
******************
URI: file:///C:/Users/Bruce/Documents/GitHub/onjava/
ExtractedExamples/files/PathInfo.java
true
*/
```

我已经在这一章第一个程序的 **main()** 方法添加了第一行用于展示操作系统的名称，因此你可以看到不同操作系统之间存在哪些差异。理想情况下，差别会相对较小，并且使用 / 或者 \ 路径分隔符进行分隔。你可以看到我运行在 Windows 10 上的程序输出。

当 **toString()** 方法生成完整形式的路径，你可以看到 **getFileName()** 方法总是返回当前文件名。通过使用 **Files** 工具类 (我们接下类将会更多地使用它)，可以测试一个文件是否存在，测试是否是一个“普通”文件还是一个目录等等。“Nofile.txt”这个示例展示我们描述的文件可能并不在指定的位置；这样可以允许你创建一个新的路径。“PathInfo.java”存在于当前目录中，最初它只是没有路径的文件名，但它仍然被检测为“存在”。一旦我们将其转换为绝对路径，我们将会得到一个从“C:”盘 (因为我们是在 Windows 机器下进行测试) 开始的完整路径，现在它也拥有一个父路径。“真实”路径的定义在文档中有点模糊，因为它取决于具体的文件系统。例如，如果文件名不区分大小写，即使路径由于大小写

设计是一个发现问题、而不是发现解决方案的过程。

的缘故而不是完全相同，也可能得到肯定的匹配结果。在这样的平台上，**toRealPath()** 将返回实际情况下的 **Path**，并且还会删除任何冗余元素。

这里你会看到 **URI** 看起来只能用于描述文件，实际上 **URI** 可以用于描述更多的东西；通过 维基百科 可以了解更多细节。现在我们成功地将 **URI** 转为一个 **Path** 对象。

最后，你会在 **Path** 中看到一些有点欺骗的东西，这就是调用 **toFile()** 方法会生成一个 **File** 对象。听起来似乎可以得到一个类似文件的东西 (毕竟被称为 **File**)，但是这个方法的存在仅仅是为了向后兼容。虽然看上去应该被称为“路径”，实际上却应该表示目录或者文件本身。这是个非常草率并且令人困惑的命名，但是由于 **java.nio.file** 的存在我们可以安全地忽略它的存在。

18.1.1 选取路径部分片段

Path 对象可以非常容易地生成路径的某一部分：

```java
// files/PartsOfPaths.java
import java.nio.file.*;

public class PartsOfPaths {
    public static void main(String[] args) {
        System.out.println(System.getProperty("os.name"));
        Path p = Paths.get("PartsOfPaths.java").toAbsolutePath();
        for(int i = 0; i < p.getNameCount(); i++)
            System.out.println(p.getName(i));
        System.out.println("ends with '.java': " +
        p.endsWith(".java"));
        for(Path pp : p) {
            System.out.print(pp + ": ");
            System.out.print(p.startsWith(pp) + " : ");
            System.out.println(p.endsWith(pp));
        }
        System.out.println("Starts with " + p.getRoot() + " " + p.
           startsWith(p.getRoot()));
    }
}

/* 输出:
Windows 10
Users
Bruce
Documents
GitHub
on-java
ExtractedExamples
files
PartsOfPaths.java
ends with '.java': false
Users: false : false
```

```
Bruce: false : false
Documents: false : false
GitHub: false : false
on-java: false : false
ExtractedExamples: false : false
files: false : false
PartsOfPaths.java: false : true
Starts with C:\ true
*/
```

可以通过 **getName()** 来索引 **Path** 的各个部分，直到达到上限 **getNameCount()**。**Path** 也实现了 **Iterable** 接口，因此我们也可以通过增强的 for-each 进行遍历。请注意，即使路径以 **.java** 结尾，使用 **endsWith()** 方法也会返回 **false**。这是因为使用 **endsWith()** 比较的是整个路径部分，而不会包含文件路径的后缀。通过使用 **startsWith()** 和 **endsWith()** 也可以完成路径的遍历。但是我们可以看到，遍历 **Path** 对象并不包含根路径，只有使用 **startsWith()** 检测根路径时才会返回 **true**。

18.1.2 路径分析

Files 工具类包含一系列完整的方法用于获得 **Path** 相关的信息。

```java
// files/PathAnalysis.java
import java.nio.file.*;
import java.io.IOException;

public class PathAnalysis {
    static void say(String id, Object result) {
        System.out.print(id + ": ");
        System.out.println(result);
    }

    public static void main(String[] args) throws IOException {
        System.out.println(System.getProperty("os.name"));
        Path p = Paths.get("PathAnalysis.java").toAbsolutePath();
        say("Exists", Files.exists(p));
        say("Directory", Files.isDirectory(p));
        say("Executable", Files.isExecutable(p));
        say("Readable", Files.isReadable(p));
        say("RegularFile", Files.isRegularFile(p));
        say("Writable", Files.isWritable(p));
        say("notExists", Files.notExists(p));
        say("Hidden", Files.isHidden(p));
        say("size", Files.size(p));
        say("FileStore", Files.getFileStore(p));
        say("LastModified: ", Files.getLastModifiedTime(p));
        say("Owner", Files.getOwner(p));
        say("ContentType", Files.probeContentType(p));
        say("SymbolicLink", Files.isSymbolicLink(p));
        if(Files.isSymbolicLink(p))
```

```
            say("SymbolicLink", Files.readSymbolicLink(p));
        if(FileSystems.getDefault().supportedFileAttributeViews().
            contains("posix"))
            say("PosixFilePermissions",
        Files.getPosixFilePermissions(p));
    }
}

/* 输出:
Windows 10
Exists: true
Directory: false
Executable: true
Readable: true
RegularFile: true
Writable: true
notExists: false
Hidden: false
size: 1631
FileStore: SSD (C:)
LastModified: : 2017-05-09T12:07:00.428366Z
Owner: MINDVIEWTOSHIBA\Bruce (User)
ContentType: null
SymbolicLink: false
*/
```

在调用最后一个测试方法 **getPosixFilePermissions()** 之前我们需要确认一下当前文件系统是否支持 **Posix** 接口，否则会抛出运行时异常。

18.1.3 Paths 的增减修改

我们必须能通过对 **Path** 对象增加或者删除一部分来构造一个新的 **Path** 对象。我们使用 **relativize()** 移除 **Path** 的根路径，使用 **resolve()** 添加 **Path** 的尾路径 (不一定是“可发现”的名称)。

对于下面代码中的示例，我使用 **relativize()** 方法从所有的输出中移除根路径，部分原因是为了示范，部分原因是为了简化输出结果，这说明你可以使用该方法将绝对路径转为相对路径。这个版本的代码中包含 **id**，以便于跟踪输出结果：

```
// files/AddAndSubtractPaths.java
import java.nio.file.*;
import java.io.IOException;

public class AddAndSubtractPaths {
    static Path base = Paths.get("..", "..", "..").toAbsolutePath().
        normalize();

    static void show(int id, Path result) {
        if(result.isAbsolute())
```

```
            System.out.println("(" + id + ")r " + base.relativize(result));
        else
            System.out.println("(" + id + ") " + result);
        try {
            System.out.println("RealPath: " + result.toRealPath());
        } catch(IOException e) {
            System.out.println(e);
        }
    }

    public static void main(String[] args) {
        System.out.println(System.getProperty("os.name"));
        System.out.println(base);
        Path p = Paths.get("AddAndSubtractPaths.java").toAbsolutePath();
        show(1, p);
        Path convoluted = p.getParent().getParent()
        .resolve("strings").resolve("..")
        .resolve(p.getParent().getFileName());
        show(2, convoluted);
        show(3, convoluted.normalize());
        Path p2 = Paths.get("..", "..");
        show(4, p2);
        show(5, p2.normalize());
        show(6, p2.toAbsolutePath().normalize());
        Path p3 = Paths.get(".").toAbsolutePath();
        Path p4 = p3.resolve(p2);
        show(7, p4);
        show(8, p4.normalize());
        Path p5 = Paths.get("").toAbsolutePath();
        show(9, p5);
        show(10, p5.resolveSibling("strings"));
        show(11, Paths.get("nonexistent"));
    }
}

/* 输出:
Windows 10
C:\Users\Bruce\Documents\GitHub
(1)r onjava\
ExtractedExamples\files\AddAndSubtractPaths.java
RealPath: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files\AddAndSubtractPaths.java
(2)r on-java\ExtractedExamples\strings\..\files
RealPath: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files
(3)r on-java\ExtractedExamples\files
RealPath: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files
(4) ..\..
```

```
RealPath: C:\Users\Bruce\Documents\GitHub\on-java
(5) ..\..
RealPath: C:\Users\Bruce\Documents\GitHub\on-java
(6)r on-java
RealPath: C:\Users\Bruce\Documents\GitHub\on-java
(7)r on-java\ExtractedExamples\files\.\..\..
RealPath: C:\Users\Bruce\Documents\GitHub\on-java
(8)r on-java
RealPath: C:\Users\Bruce\Documents\GitHub\on-java
(9)r on-java\ExtractedExamples\files
RealPath: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files
(10)r on-java\ExtractedExamples\strings
RealPath: C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\strings
(11) nonexistent
java.nio.file.NoSuchFileException:
C:\Users\Bruce\Documents\GitHub\onjava\
ExtractedExamples\files\nonexistent
*/
```

我还为 **toRealPath()** 添加了更多的测试，这是为了扩展和规则化，防止路径不存在时抛出运行时异常。

18.2 目录

Files 工具类包含大部分我们需要的目录操作和文件操作方法。出于某种原因，它们没有包含删除目录树相关的方法，因此我们将实现并将其添加到 **onjava** 库中。

```java
// onjava/RmDir.java
package onjava;

import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;
import java.io.IOException;

public class RmDir {
    public static void rmdir(Path dir) throws IOException {
        Files.walkFileTree(dir, new SimpleFileVisitor<Path>() {
            @Override
            public FileVisitResult visitFile(Path file,
                BasicFileAttributes attrs) throws IOException {
                Files.delete(file);
                return FileVisitResult.CONTINUE;
            }

            @Override
            public FileVisitResult postVisitDirectory(Path dir,
```

```
                    IOException exc) throws IOException {
                     Files.delete(dir);
                     return FileVisitResult.CONTINUE;
                }
            });
        }
    }
```

删除目录树的方法实现依赖于 **Files.walkFileTree()**，“walking” 目录树意味着遍历每个子目录和文件。*Visitor* 设计模式提供了一种标准机制来访问集合中的每个对象，然后你需要提供在每个对象上执行的操作。此操作的定义取决于实现的 **FileVisitor** 的四个抽象方法，包括：

```
1.  **preVisitDirectory()**：在访问目录中条目之前在目录上运行。
2.  **visitFile()**：运行目录中的每一个文件。
3.  **visitFileFailed()**：调用无法访问的文件。
4.  **postVisitDirectory()**：在访问目录中条目之后在目录上运行，包括所有的子目录。
```

为了简化，**java.nio.file.SimpleFileVisitor** 提供了所有方法的默认实现。这样，在我们的匿名内部类中，我们只需要重写非标准行为的方法：**visitFile()** 和 **postVisitDirectory()** 实现删除文件和删除目录。两者都应该返回标志位决定是否继续访问 (这样就可以继续访问，直到找到所需要的)。作为探索目录操作的一部分，现在我们可以有条件地删除已存在的目录。在以下例子中，**makeVariant()** 接受基本目录测试，并通过旋转部件列表生成不同的子目录路径。这些旋转与路径分隔符 **sep** 使用 **String.join()** 贴在一起，然后返回一个 **Path** 对象。

```java
// files/Directories.java
import java.util.*;
import java.nio.file.*;
import onjava.RmDir;

public class Directories {
    static Path test = Paths.get("test");
    static String sep = FileSystems.getDefault().getSeparator();
    static List<String> parts = Arrays.asList("foo", "bar", "baz", "bag
        ");

    static Path makeVariant() {
        Collections.rotate(parts, 1);
        return Paths.get("test", String.join(sep, parts));
    }

    static void refreshTestDir() throws Exception {
        if(Files.exists(test))
        RmDir.rmdir(test);
        if(!Files.exists(test))
        Files.createDirectory(test);
    }

```

```
    public static void main(String[] args) throws Exception {
        refreshTestDir();
        Files.createFile(test.resolve("Hello.txt"));
        Path variant = makeVariant();
        // Throws exception (too many levels):
        try {
            Files.createDirectory(variant);
        } catch(Exception e) {
            System.out.println("Nope, that doesn't work.");
        }
        populateTestDir();
        Path tempdir = Files.createTempDirectory(test, "DIR_");
        Files.createTempFile(tempdir, "pre", ".non");
        Files.newDirectoryStream(test).forEach(System.out::println);
        System.out.println("*********");
        Files.walk(test).forEach(System.out::println);
    }

    static void populateTestDir() throws Exception  {
        for(int i = 0; i < parts.size(); i++) {
            Path variant = makeVariant();
            if(!Files.exists(variant)) {
                Files.createDirectories(variant);
                Files.copy(Paths.get("Directories.java"),
                    variant.resolve("File.txt"));
                Files.createTempFile(variant, null, null);
            }
        }
    }
}

/* 输出:
Nope, that doesn't work.
test\bag
test\bar
test\baz
test\DIR_5142667942049986036
test\foo
test\Hello.txt
*********
test
test\bag
test\bag\foo
test\bag\foo\bar
test\bag\foo\bar\baz
test\bag\foo\bar\baz\8279660869874696036.tmp
test\bag\foo\bar\baz\File.txt
test\bar
test\bar\baz
test\bar\baz\bag
test\bar\baz\bag\foo
```

```
test\bar\baz\bag\foo\1274043134240426261.tmp
test\bar\baz\bag\foo\File.txt
test\baz
test\baz\bag
test\baz\bag\foo
test\baz\bag\foo\bar
test\baz\bag\foo\bar\6130572530014544105.tmp
test\baz\bag\foo\bar\File.txt
test\DIR_5142667942049986036
test\DIR_5142667942049986036\pre7704286843227113253.non
test\foo
test\foo\bar
test\foo\bar\baz
test\foo\bar\baz\bag
test\foo\bar\baz\bag\5412864507741775436.tmp
test\foo\bar\baz\bag\File.txt
test\Hello.txt
*/
```

首先, **refreshTestDir()** 用于检测 **test** 目录是否已经存在。若存在, 则使用我们新工具类 **rmdir()** 删除其整个目录。检查是否 **exists** 是多余的，但我想说明一点，因为如果你对于已经存在的目录调用 **createDirectory()** 将会抛出异常。**createFile()** 使用参数 **Path** 创建一个空文件; **resolve()** 将文件名添加到 **test Path** 的末尾。

我们尝试使用 **createDirectory()** 来创建多级路径，但是这样会抛出异常，因为这个方法只能创建单级路径。我已经将 **populateTestDir()** 作为一个单独的方法，因为它将在后面的例子中被重用。对于每一个变量 **variant**，我们都能使用 **createDirectories()** 创建完整的目录路径，然后使用此文件的副本以不同的目标名称填充该终端目录。然后我们使用 **createTempFile()** 生成一个临时文件。

在调用 **populateTestDir()** 之后，我们在 **test** 目录下面下面创建一个临时目录。请注意，**createTempDirectory()** 只有名称的前缀选项。与 **createTempFile()** 不同，我们再次使用它将临时文件放入新的临时目录中。你可以从输出中看到，如果未指定后缀，它将默认使用“.tmp”作为后缀。

为了展示结果，我们首次使用看起来很有希望的 **newDirectoryStream()**，但事实证明这个方法只是返回 **test** 目录内容的 Stream 流，并没有更多的内容。要获取目录树的全部内容的流，请使用 **Files.walk()**。

18.3 文件系统

为了完整起见，我们需要一种方法查找文件系统相关的其他信息。在这里，我们使用静态的 **FileSystems** 工具类获取“默认”的文件系统，但你同样也可以在 **Path** 对象上调用 **getFileSystem()** 以获取创建该 **Path** 的文件系统。你可以获得给定 *URI* 的文件系统，还可以构建新的文件系统 (对于支持它的操作系统)。

```java
// files/FileSystemDemo.java
import java.nio.file.*;

public class FileSystemDemo {
    static void show(String id, Object o) {
        System.out.println(id + ": " + o);
    }

    public static void main(String[] args) {
        System.out.println(System.getProperty("os.name"));
        FileSystem fsys = FileSystems.getDefault();
        for(FileStore fs : fsys.getFileStores())
            show("File Store", fs);
        for(Path rd : fsys.getRootDirectories())
            show("Root Directory", rd);
        show("Separator", fsys.getSeparator());
        show("UserPrincipalLookupService",
            fsys.getUserPrincipalLookupService());
        show("isOpen", fsys.isOpen());
        show("isReadOnly", fsys.isReadOnly());
        show("FileSystemProvider", fsys.provider());
        show("File Attribute Views",
        fsys.supportedFileAttributeViews());
    }
}
/* 输出:
Windows 10
File Store: SSD (C:)
Root Directory: C:\
Root Directory: D:\
Separator: \
UserPrincipalLookupService:
sun.nio.fs.WindowsFileSystem$LookupService$1@15db9742
isOpen: true
isReadOnly: false
FileSystemProvider:
sun.nio.fs.WindowsFileSystemProvider@6d06d69c
File Attribute Views: [owner, dos, acl, basic, user]
*/
```

一个 **FileSystem** 对象也能生成 **WatchService** 和 **PathMatcher** 对象，将会在接下来两章中详细讲解。

18.4 路径监听

通过 **WatchService** 可以设置一个进程对目录中的更改做出响应。在这个例子中，**delTxtFiles()** 作为一个单独的任务执行，该任务将遍历整个目录并删除以 **.txt** 结尾的所有文件，**WatchService** 会

对文件删除操作做出反应:

```
// files/PathWatcher.java
// {ExcludeFromGradle}
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
import java.util.concurrent.*;

public class PathWatcher {
    static Path test = Paths.get("test");

    static void delTxtFiles() {
        try {
            Files.walk(test)
            .filter(f ->
                f.toString()
                .endsWith(".txt"))
                .forEach(f -> {
                try {
                    System.out.println("deleting " + f);
                    Files.delete(f);
                } catch(IOException e) {
                    throw new RuntimeException(e);
                }
            });
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws Exception {
        Directories.refreshTestDir();
        Directories.populateTestDir();
        Files.createFile(test.resolve("Hello.txt"));
        WatchService watcher = FileSystems.getDefault().newWatchService
            ();
        test.register(watcher, ENTRY_DELETE);
        Executors.newSingleThreadScheduledExecutor()
        .schedule(PathWatcher::delTxtFiles,
        250, TimeUnit.MILLISECONDS);
        WatchKey key = watcher.take();
        for(WatchEvent evt : key.pollEvents()) {
            System.out.println("evt.context(): " + evt.context() +
            "\nevt.count(): " + evt.count() +
            "\nevt.kind(): " + evt.kind());
            System.exit(0);
        }
    }
}
/* Output:
```

```
deleting test\bag\foo\bar\baz\File.txt
deleting test\bar\baz\bag\foo\File.txt
deleting test\baz\bag\foo\bar\File.txt
deleting test\foo\bar\baz\bag\File.txt
deleting test\Hello.txt
evt.context(): Hello.txt
evt.count(): 1
evt.kind(): ENTRY_DELETE
*/
```

delTxtFiles() 中的 **try** 代码块看起来有些多余，因为它们捕获的是同一种类型的异常，外部的 **try** 语句似乎已经足够了。然而出于某种原因，Java 要求两者都必须存在 (这也可能是一个 bug)。还要注意的是在 **filter()** 中，我们必须显式地使用 **f.toString()** 转为字符串，否则我们调用 **endsWith()** 将会与整个 **Path** 对象进行比较，而不是路径名称字符串的一部分进行比较。

一旦我们从 **FileSystem** 中得到了 **WatchService** 对象，我们将其注册到 **test** 路径以及我们感兴趣的项目的变量参数列表中，可以选择 **ENTRY_CREATE**，**ENTRY_DELETE** 或 **ENTRY_MODIFY**(其中创建和删除不属于修改)。

因为接下来对 **watcher.take()** 的调用会在发生某些事情之前停止所有操作，所以我们希望 **deltxtfiles()** 能够并行运行以便生成我们感兴趣的事件。为了实现这个目的，我通过调用 **Executors.newSingleThreadScheduledExecutor()** 产生一个 **ScheduledExecutorService** 对象，然后调用 **schedule()** 方法传递所需函数的方法引用，并且设置在运行之前应该等待的时间。

此时，**watcher.take()** 将等待并阻塞在这里。当目标事件发生时，会返回一个包含 **WatchEvent** 的 **Watchkey** 对象。展示的这三种方法是能对 **WatchEvent** 执行的全部操作。

查看输出的具体内容。即使我们正在删除以 **.txt** 结尾的文件，在 **Hello.txt** 被删除之前，**WatchService** 也不会被触发。你可能认为，如果说“监视这个目录”，自然会包含整个目录和下面子目录，但实际上的：只会监视给定的目录，而不是下面的所有内容。如果需要监视整个树目录，必须在整个树的每个子目录上放置一个 **Watchservice**。

```java
// files/TreeWatcher.java
// {ExcludeFromGradle}
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
import java.util.concurrent.*;

public class TreeWatcher {

    static void watchDir(Path dir) {
        try {
            WatchService watcher =
            FileSystems.getDefault().newWatchService();
            dir.register(watcher, ENTRY_DELETE);
            Executors.newSingleThreadExecutor().submit(() -> {
                try {
```

```
                    WatchKey key = watcher.take();
                    for(WatchEvent evt : key.pollEvents()) {
                        System.out.println(
                        "evt.context(): " + evt.context() +
                        "\nevt.count(): " + evt.count() +
                        "\nevt.kind(): " + evt.kind());
                        System.exit(0);
                    }
                } catch(InterruptedException e) {
                    return;
                }
            });
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws Exception {
        Directories.refreshTestDir();
        Directories.populateTestDir();
        Files.walk(Paths.get("test"))
            .filter(Files::isDirectory)
            .forEach(TreeWatcher::watchDir);
        PathWatcher.delTxtFiles();
    }
}

/* Output:
deleting test\bag\foo\bar\baz\File.txt
deleting test\bar\baz\bag\foo\File.txt
evt.context(): File.txt
evt.count(): 1
evt.kind(): ENTRY_DELETE
*/
```

在 **watchDir()** 方法中给 **WatchSevice** 提供参数 **ENTRY_DELETE**，并启动一个独立的线程来监视该 **Watchservice**。这里我们没有使用 **schedule()** 进行启动，而是使用 **submit()** 启动线程。我们遍历整个目录树，并将 **watchDir()** 应用于每个子目录。现在，当我们运行 **deltxtfiles()** 时，其中一个 **Watchservice** 会检测到每一次文件删除。

18.5 文件查找

到目前为止，为了找到文件，我们一直使用相当粗糙的方法，在 `path` 上调用 `toString()`，然后使用 `string` 操作查看结果。事实证明，`java.nio.file` 有更好的解决方案：通过在 `FileSystem` 对象上调用 `getPathMatcher()` 获得一个 `PathMatcher`，然后传入您感兴趣的模式。模式有两个选项：`glob` 和 `regex`。`glob` 比较简单，实际上功能非常强大，因此您可以使用 `glob` 解决许多问题。如

我会在代码更替的时候添加单元测试。

果您的问题更复杂，可以使用 regex，这将在接下来的 Strings 一章中解释。

在这里，我们使用 glob 查找以 .tmp 或 .txt 结尾的所有 Path:

```
// files/Find.java
// {ExcludeFromGradle}
import java.nio.file.*;

public class Find {
    public static void main(String[] args) throws Exception {
        Path test = Paths.get("test");
        Directories.refreshTestDir();
        Directories.populateTestDir();
        // Creating a *directory*, not a file:
        Files.createDirectory(test.resolve("dir.tmp"));

        PathMatcher matcher = FileSystems.getDefault()
          .getPathMatcher("glob:**/*.{tmp,txt}");
        Files.walk(test)
          .filter(matcher::matches)
          .forEach(System.out::println);
        System.out.println("***************");

        PathMatcher matcher2 = FileSystems.getDefault()
          .getPathMatcher("glob:*.tmp");
        Files.walk(test)
          .map(Path::getFileName)
          .filter(matcher2::matches)
          .forEach(System.out::println);
        System.out.println("***************");

        Files.walk(test) // Only look for files
          .filter(Files::isRegularFile)
          .map(Path::getFileName)
          .filter(matcher2::matches)
          .forEach(System.out::println);
    }
}
/* Output:
test\bag\foo\bar\baz\5208762845883213974.tmp
test\bag\foo\bar\baz\File.txt
test\bar\baz\bag\foo\7918367201207778677.tmp
test\bar\baz\bag\foo\File.txt
test\baz\bag\foo\bar\8016595521026696632.tmp
test\baz\bag\foo\bar\File.txt
test\dir.tmp
test\foo\bar\baz\bag\5832319279813617280.tmp
test\foo\bar\baz\bag\File.txt
***************
5208762845883213974.tmp
7918367201207778677.tmp
```

```
8016595521026696632.tmp
dir.tmp
5832319279813617280.tmp
***************
5208762845883213974.tmp
7918367201207778677.tmp
8016595521026696632.tmp
5832319279813617280.tmp
*/
```

在 matcher 中，glob 表达式开头的 **/ 表示“当前目录及所有子目录”，这在当你不仅仅要匹配当前目录下特定结尾的 Path 时非常有用。单 * 表示“任何东西”，然后是一个点，然后大括号表示一系列的可能性—我们正在寻找以 .tmp 或 .txt 结尾的东西。您可以在 getPathMatcher() 文档中找到更多详细信息。

matcher2 只使用 *.tmp，通常不匹配任何内容，但是添加 map() 操作会将完整路径减少到末尾的名称。

注意，在这两种情况下，输出中都会出现 dir.tmp，即使它是一个目录而不是一个文件。要只查找文件，必须像在最后 files.walk() 中那样对其进行筛选。

18.6 文件读写

此时，我们可以对路径和目录做任何事情。现在让我们看一下操纵文件本身的内容。

如果一个文件很“小”，也就是说“它运行得足够快且占用内存小”，那么 java.nio.file.Files 类中的实用程序将帮助你轻松读写文本和二进制文件。

Files.readAllLines() 一次读取整个文件（因此，“小”文件很有必要），产生一个List<String>。对于示例文件，我们将重用streams/Cheese.dat：

```java
// files/ListOfLines.java
import java.util.*;
import java.nio.file.*;

public class ListOfLines {
    public static void main(String[] args) throws Exception {
        Files.readAllLines(
        Paths.get("../streams/Cheese.dat"))
        .stream()
        .filter(line -> !line.startsWith("//"))
        .map(line ->
            line.substring(0, line.length()/2))
        .forEach(System.out::println);
    }
}
/* Output:
Not much of a cheese
```

```
Finest in the
And what leads you
Well, it's
It's certainly uncon
*/
```

跳过注释行，其余的内容每行只打印一半。这实现起来很简单：你只需将 Path 传递给 readAllLines()（以前的 java 实现这个功能很复杂）。readAllLines() 有一个重载版本，包含一个 Charset 参数来存储文件的 Unicode 编码。

Files.write() 被重载以写入 byte 数组或任何 Iterable 对象（它也有 Charset 选项）：

```java
// files/Writing.java
import java.util.*;
import java.nio.file.*;

public class Writing {
    static Random rand = new Random(47);
    static final int SIZE = 1000;

    public static void main(String[] args) throws Exception {
        // Write bytes to a file:
        byte[] bytes = new byte[SIZE];
        rand.nextBytes(bytes);
        Files.write(Paths.get("bytes.dat"), bytes);
        System.out.println("bytes.dat: " + Files.size(Paths.get("bytes.
            dat")));

        // Write an iterable to a file:
        List<String> lines = Files.readAllLines(
          Paths.get("../streams/Cheese.dat"));
        Files.write(Paths.get("Cheese.txt"), lines);
        System.out.println("Cheese.txt: " + Files.size(Paths.get("
            Cheese.txt")));
    }
}
/* Output:
bytes.dat: 1000
Cheese.txt: 199
*/
```

我们使用 Random 来创建一个随机的 byte 数组; 你可以看到生成的文件大小是 1000。

一个 List 被写入文件，任何 Iterable 对象也可以这么做。

如果文件大小有问题怎么办？比如说：

1. 文件太大，如果你一次性读完整个文件，你可能会耗尽内存。
2. 您只需要在文件的中途工作以获得所需的结果，因此读取整个文件会浪费时间。

Files.lines() 方便地将文件转换为行的 Stream：

```java
// files/ReadLineStream.java
import java.nio.file.*;

public class ReadLineStream {
    public static void main(String[] args) throws Exception {
        Files.lines(Paths.get("PathInfo.java"))
          .skip(13)
          .findFirst()
          .ifPresent(System.out::println);
    }
}
/* Output:
    show("RegularFile", Files.isRegularFile(p));
*/
```

这对本章中第一个示例代码做了流式处理，跳过 13 行，然后选择下一行并将其打印出来。

`Files.lines()` 对于把文件处理行的传入流时非常有用，但是如果你想在 `Stream` 中读取，处理或写入怎么办？这就需要稍微复杂的代码：

```java
// files/StreamInAndOut.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class StreamInAndOut {
    public static void main(String[] args) {
        try(
          Stream<String> input =
            Files.lines(Paths.get("StreamInAndOut.java"));
          PrintWriter output =
            new PrintWriter("StreamInAndOut.txt")
        ) {
            input.map(String::toUpperCase)
              .forEachOrdered(output::println);
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

因为我们在同一个块中执行所有操作，所以这两个文件都可以在相同的 try-with-resources 语句中打开。`PrintWriter` 是一个旧式的 `java.io` 类，允许你“打印”到一个文件，所以它是这个应用的理想选择。如果你看一下 `StreamInAndOut.txt`，你会发现它里面的内容确实是大写的。

18.7 本章小结

虽然本章对文件和目录操作做了相当全面的介绍，但是仍然有没被介绍的类库中的功能——一定要研究 `java.nio.file` 的 Javadocs，尤其是 `java.nio.file.Files` 这个类。

Java 7 和 8 对于处理文件和目录的类库做了大量改进。如果您刚刚开始使用 Java，那么您很幸运。在过去，它令人非常不愉快，我确信 Java 设计者以前对于文件操作不够重视才没做简化。对于初学者来说这是一件很棒的事，对于教学者来说也一样。我不明白为什么花了这么长时间来解决这个明显的问题，但不管怎么说它被解决了，我很高兴。使用文件现在很简单，甚至很有趣，这是你以前永远想不到的。

19 字符串

> 字符串操作毫无疑问是计算机程序设计中最常见的行为之一。

在 Java 大展拳脚的 Web 系统中更是如此。在本章中，我们将深入学习在 Java 语言中应用最广泛的 `String` 类，并研究与之相关的类及工具。

19.1 字符串的不可变

`String` 对象是不可变的。查看 JDK 文档你就会发现，`String` 类中每一个看起来会修改 `String` 值的方法，实际上都是创建了一个全新的 `String` 对象，以包含修改后的字符串内容。而最初的 `String` 对象则丝毫未动。

看看下面的代码：

```
// strings/Immutable.java
public class Immutable {
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = "howdy";
        System.out.println(q); // howdy
        String qq = upcase(q);
        System.out.println(qq); // HOWDY
        System.out.println(q); // howdy
    }
}
/* Output:
howdy
HOWDY
howdy
*/
```

当把 `q` 传递给 `upcase()` 方法时，实际传递的是引用的一个拷贝。其实，每当把 String 对象作为方法的参数时，都会复制一份引用，而该引用所指向的对象其实一直待在单一的物理位置上，从未动过。

回到 `upcase()` 的定义，传入其中的引用有了名字 `s`，只有 `upcase()` 运行的时候，局部引用 `s`

才存在。一旦 upcase() 运行结束，s 就消失了。当然了，upcase() 的返回值，其实是最终结果的引用。这足以说明，upcase() 返回的引用已经指向了一个新的对象，而 q 仍然在原来的位置。

String 的这种行为正是我们想要的。例如：

```
String s = "asdf";
String x = Immutable.upcase(s);
```

难道你真的希望 upcase() 方法改变其参数吗？对于一个方法而言，参数是为该方法提供信息的，而不是想让该方法改变自己的。在阅读这段代码时，读者自然会有这样的感觉。这一点很重要，正是有了这种保障，才使得代码易于编写和阅读。

19.2 + 的重载与 StringBuilder

String 对象是不可变的，你可以给一个 String 对象添加任意多的别名。因为 String 是只读的，所以指向它的任何引用都不可能修改它的值，因此，也就不会影响到其他引用。

不可变性会带来一定的效率问题。为 String 对象重载的 + 操作符就是一个例子。重载的意思是，一个操作符在用于特定的类时，被赋予了特殊的意义（用于 String 的 + 与 += 是 Java 中仅有的两个重载过的操作符，Java 不允许程序员重载任何其他的操作符 [1]）。

操作符 + 可以用来连接 String：

```
// strings/Concatenation.java

public class Concatenation {
    public static void main(String[] args) {
        String mango = "mango";
        String s = "abc" + mango + "def" + 47;
        System.out.println(s);
    }
}
/* Output:
abcmangodef47
*/
```

可以想象一下，这段代码是这样工作的：String 可能有一个 append() 方法，它会生成一个新的 String 对象，以包含“abc”与 mango 连接后的字符串。该对象会再创建另一个新的 String 对象，然后与“def”相连，生成另一个新的对象，依此类推。

这种方式当然是可行的，但是为了生成最终的 String 对象，会产生一大堆需要垃圾回收的中间对象。我猜想，Java 设计者一开始就是这么做的（这也是软件设计中的一个教训：除非你用代码将系统实现，并让它运行起来，否则你无法真正了解它会有什么问题），然后他们发现其性能相当糟糕。

想看看以上代码到底是如何工作的吗？可以用 JDK 自带的 javap 工具来反编译以上代码。命令

[1]

我想到了一个绝妙的 idea!

如下:

```
javap -c Concatenation
```

这里的 -c 标志表示将生成 JVM 字节码。我们剔除不感兴趣的部分，然后做细微的修改，于是有了以下的字节码:

```
public static void main(java.lang.String[]);
 Code:
  Stack=2, Locals=3, Args_size=1
  0: ldc #2; //String mango
  2: astore_1
  3: new #3; //class StringBuilder
  6: dup
  7: invokespecial #4; //StringBuilder."<init>":()
  10: ldc #5; //String abc
  12: invokevirtual #6; //StringBuilder.append:(String)
  15: aload_1
  16: invokevirtual #6; //StringBuilder.append:(String)
  19: ldc #7; //String def
  21: invokevirtual #6; //StringBuilder.append:(String)
  24: bipush 47
  26: invokevirtual #8; //StringBuilder.append:(I)
  29: invokevirtual #9; //StringBuilder.toString:()
  32: astore_2
  33: getstatic #10; //Field System.out:PrintStream;
  36: aload_2
  37: invokevirtual #11; //PrintStream.println:(String)
  40: return
```

如果你有汇编语言的经验，以上代码应该很眼熟 (其中的 dup 和 invokevirtual 语句相当于 Java 虚拟机上的汇编语句。即使你完全不了解汇编语言也无需担心)。需要重点注意的是：编译器自动引入了 java.lang.StringBuilder 类。虽然源代码中并没有使用 StringBuilder 类，但是编译器却自作主张地使用了它，就因为它更高效。

在这里，编译器创建了一个 StringBuilder 对象，用于构建最终的 String，并对每个字符串调用了一次 append() 方法，共计 4 次。最后调用 toString() 生成结果，并存为 s (使用的命令为 astore_2)。

现在，也许你会觉得可以随意使用 String 对象，反正编译器会自动为你做性能优化。可是在这之前，让我们更深入地看看编译器能为我们优化到什么程度。下面的例子采用两种方式生成一个 String：方法一使用了多个 String 对象；方法二在代码中使用了 StringBuilder。

```
// strings/WhitherStringBuilder.java

public class WhitherStringBuilder {
    public String implicit(String[] fields) {
        String result = "";
```

我需要重构代码，上一个人写得太烂了。

```
        for(String field : fields) {
            result += field;
        }
        return result;
    }
    public String explicit(String[] fields) {
        StringBuilder result = new StringBuilder();
        for(String field : fields) {
            result.append(field);
        }
        return result.toString();
    }
}
```

现在运行 `javap -c WitherStringBuilder`，可以看到两种不同方法（我已经去掉不相关的细节）对应的字节码。首先是 `implicit()` 方法：

```
public java.lang.String implicit(java.lang.String[]);
0: ldc #2 // String
2: astore_2
3: aload_1
4: astore_3
5: aload_3
6: arraylength
7: istore 4
9: iconst_0
10: istore 5
12: iload 5
14: iload 4
16: if_icmpge 51
19: aload_3
20: iload 5
22: aaload
23: astore 6
25: new #3 // StringBuilder
28: dup
29: invokespecial #4 // StringBuilder."<init>"
32: aload_2
33: invokevirtual #5 // StringBuilder.append:(String)
36: aload 6
38: invokevirtual #5 // StringBuilder.append:(String;)
41: invokevirtual #6 // StringBuilder.toString:()
44: astore_2
45: iinc 5, 1
48: goto 12
51: aload_2
52: areturn
```

注意从第 16 行到第 48 行构成了一个循环体。第 16 行：对堆栈中的操作数进行“大于或等于的整数比较运算”，循环结束时跳转到第 51 行。第 48 行：重新回到循环体的起始位置（第 12 行）。注意：

我又不能测试所有的功能。

StringBuilder 是在循环内构造的，这意味着每进行一次循环，会创建一个新的 StringBuilder 对象。

下面是 explicit() 方法对应的字节码：

```
public java.lang.String explicit(java.lang.String[]);
0: new #3 // StringBuilder
3: dup
4: invokespecial #4 // StringBuilder."<init>"
7: astore_2
8: aload_1
9: astore_3
10: aload_3
11: arraylength
12: istore 4
14: iconst_0
15: istore 5
17: iload 5
19: iload 4
21: if_icmpge 43
24: aload_3
25: iload 5
27: aaload
28: astore 6
30: aload_2
31: aload 6
33: invokevirtual #5 // StringBuilder.append:(String)
36: pop
37: iinc 5, 1
40: goto 17
43: aload_2
44: invokevirtual #6 // StringBuilder.toString:()
47: areturn
```

可以看到，不仅循环部分的代码更简短、更简单，而且它只生成了一个 StringBuilder 对象。显式地创建 StringBuilder 还允许你预先为其指定大小。如果你已经知道最终字符串的大概长度，那预先指定 StringBuilder 的大小可以避免频繁地重新分配缓冲。

因此，当你为一个类编写 toString() 方法时，如果字符串操作比较简单，那就可以信赖编译器，它会为你合理地构造最终的字符串结果。但是，如果你要在 toString() 方法中使用循环，且可能有性能问题，那么最好自己创建一个 StringBuilder 对象，用它来构建最终结果。请参考以下示例：

```java
// strings/UsingStringBuilder.java

import java.util.*;
import java.util.stream.*;
public class UsingStringBuilder {
    public static String string1() {
        Random rand = new Random(47);
        StringBuilder result = new StringBuilder("[");
```

```
        for(int i = 0; i < 25; i++) {
            result.append(rand.nextInt(100));
            result.append(", ");
        }
        result.delete(result.length()-2, result.length());
        result.append("]");
        return result.toString();
    }
    public static String string2() {
        String result = new Random(47)
            .ints(25, 0, 100)
            .mapToObj(Integer::toString)
            .collect(Collectors.joining(", "));
        return "[" + result + "]";
    }
    public static void main(String[] args) {
        System.out.println(string1());
        System.out.println(string2());
    }
}
/* Output:
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89,
9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89,
9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
*/
```

在方法 `string1()` 中，最终结果是用 `append()` 语句拼接起来的。如果你想走捷径，例如：`append(a + ": "+ c)`，编译器就会掉入陷阱，从而为你另外创建一个 `StringBuilder` 对象处理括号内的字符串操作。如果拿不准该用哪种方式，随时可以用 `javap` 来分析你的程序。

`StringBuilder` 提供了丰富而全面的方法，包括 `insert()`、`replace()`、`substring()`，甚至还有`reverse()`，但是最常用的还是 `append()` 和 `toString()`。还有 `delete()`，上面的例子中我们用它删除最后一个逗号和空格，以便添加右括号。

`string2()` 使用了 `Stream`，这样代码更加简洁美观。可以证明，`Collectors.joining()` 内部也是使用的 `StringBuilder`，这种写法不会影响性能！

`StringBuilder`是 Java SE5 引入的，在这之前用的是 `StringBuffer`。后者是线程安全的（参见并发编程），因此开销也会大些。使用 `StringBuilder` 进行字符串操作更快一点。

19.3 意外递归

Java 中的每个类从根本上都是继承自 `Object`，标准集合类也是如此，它们都有 `toString()` 方法，并且覆盖了该方法，使得它生成的 `String` 结果能够表达集合自身，以及集合包含的对象。例如 `ArrayList.toString()`，它会遍历 `ArrayList` 中包含的所有对象，调用每个元素上的 `toString()`

无论在排练中演示是如何的顺利（高效），当面对真正的现场观众时，出现错误的可能性跟在场观看的人数成正比。

方法:

```
// strings/ArrayListDisplay.java
import java.util.*;
import java.util.stream.*;
import generics.coffee.*;
public class ArrayListDisplay {
    public static void main(String[] args) {
        List<Coffee> coffees =
            Stream.generate(new CoffeeSupplier())
                .limit(10)
                .collect(Collectors.toList());
        System.out.println(coffees);
    }
}
/* Output:
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4,
Breve 5, Americano 6, Latte 7, Cappuccino 8, Cappuccino 9]
*/
```

如果你希望 `toString()` 打印出类的内存地址，也许你会考虑使用 **this** 关键字:

```
// strings/InfiniteRecursion.java
// Accidental recursion
// {ThrowsException}
// {VisuallyInspectOutput} Throws very long exception
import java.util.*;
import java.util.stream.*;

public class InfiniteRecursion {
    @Override
    public String toString() {
        return " InfiniteRecursion address: " + this + "\n"
    }
    public static void main(String[] args) {
        Stream.generate(InfiniteRecursion::new)
            .limit(10)
            .forEach(System.out::println);
    }
}
```

当你创建了 `InfiniteRecursion` 对象，并将其打印出来的时候，你会得到一串很长的异常信息。如果你将该 `InfiniteRecursion` 对象存入一个 `ArrayList` 中，然后打印该 `ArrayList`，同样也会抛出异常。其实，当运行到如下代码时:

```
"InfiniteRecursion address: " + this
```

这里发生了自动类型转换，由 `InfiniteRecursion` 类型转换为 `String` 类型。因为编译器发现一个 `String` 对象后面跟着一个 “+”，而 “+” 后面的对象不是 `String`，于是编译器试着将 **this** 转换

成一个 String。它怎么转换呢？正是通过调用 **this** 上的 toString() 方法，于是就发生了递归调用。

如果你真的想要打印对象的内存地址，应该调用 Object.toString() 方法，这才是负责此任务的方法。所以，不要使用 **this**，而是应该调用 **super**.toString() 方法。

19.4 字符串操作

以下是 String 对象具备的一些基本方法。重载的方法归纳在同一行中：

方法	参数，重载版本	作用
构造方法	默认版本，String，StringBuilder，StringBuffer，**char**数组，**byte**数组	创建String对象
length()		String中字符的个数
charAt()	**int**索引	获取String中索引位置上的**char**
getChars()，getBytes()	待复制部分的开始和结束索引，复制的目标数组，目标数组的开始索引	复制**char**或**byte**到一个目标数组中
toCharArray()		生成一个**char**[]，包含String中的所有字符
equals()，equalsIgnoreCase()	与之进行比较的String	比较两个String的内容是否相同。如果相同，结果为**true**
compareTo()，compareToIgnoreCase()	与之进行比较的String	按词典顺序比较String的内容，比较结果为负数、零或正数。注意，大小写不等价
contains()	要搜索的CharSequence	如果该String对象包含参数的内容，则返回**true**
contentEquals()	与之进行比较的CharSequence或StringBuffer	如果该String对象与参数的内容完全一致，则返回**true**

方法	参数，重载版本	作用
isEmpty()		返回**boolean**结果，以表明String对象的长度是否为 0
regionMatches()	该String的索引偏移量，另一个String及其索引偏移量，要比较的长度。重载版本增加了“忽略大小写”功能	返回**boolean**结果，以表明所比较区域是否相等
startsWith()	可能的起始String。重载版本在参数中增加了偏移量	返回**boolean**结果，以表明该String是否以传入参数开始
endsWith()	该String可能的后缀String	返回**boolean**结果，以表明此参数是否是该字符串的后缀
indexOf(), lastIndexOf()	重载版本包括：**char**，**char**与起始索引，String，String与起始索引	如果该String并不包含此参数，就返回-1；否则返回此参数在String中的起始索引。lastIndexOf() 是从后往前搜索
matches()	一个正则表达式	返回**boolean**结果，以表明该String和给出的正则表达式是否匹配
split()	一个正则表达式。可选参数为需要拆分的最大数量	按照正则表达式拆分String，返回一个结果数组
join()（Java8 引入的）	分隔符，待拼字符序列。用分隔符将字符序列拼接成一个新的String	用分隔符拼接字符片段，产生一个新的String
substring()（即subSequence()）	重载版本：起始索引；起始索引 + 终止索引	返回一个新的String对象，以包含参数指定的子串
concat()	要连接的String	返回一个新的String对象，内容为原始String连接上参数String

方法	参数，重载版本	作用
replace()	要替换的字符，用来进行替换的新字符。也可以用一个CharSequence替换另一个CharSequence	返回替换字符后的新String对象。如果没有替换发生，则返回原始的String对象
replaceFirst()	要替换的正则表达式，用来进行替换的String	返回替换首个目标字符串后的String对象
replaceAll()	要替换的正则表达式，用来进行替换的String	返回替换所有目标字符串后的String对象
toLowerCase(), toUpperCase()		将字符的大小写改变后，返回一个新的String对象。如果没有任何改变，则返回原始的String对象
trim()		将String两端的空白符删除后，返回一个新的String对象。如果没有任何改变，则返回原始的String对象
valueOf() (**static**)	重载版本：Object；**char**[]；**char**[]，偏移量，与字符个数；**boolean**；**char**；**int**；**long**；**float**；**double**	返回一个表示参数内容的String
intern()		为每个唯一的字符序列生成一个且仅生成一个String引用
format()	要格式化的字符串，要替换到格式化字符串的参数	返回格式化结果String

从这个表可以看出，当需要改变字符串的内容时，String 类的方法都会返回一个新的 String 对象。同时，如果内容不改变，String 方法只是返回原始对象的一个引用而已。这可以节约存储空间以及避免额外的开销。

本章稍后还将介绍正则表达式在 String 方法中的应用。

19.5 格式化输出

在长久的等待之后，Java SE5 终于推出了 C 语言中 `printf()` 风格的格式化输出这一功能。这不仅使得控制输出的代码更加简单，同时也给与 Java 开发者对于输出格式与排列更强大的控制能力。### `printf()` C 语言的 `printf()` 并不像 Java 那样连接字符串，它使用一个简单的格式化字符串，加上要插入其中的值，然后将其格式化输出。`printf()` 并不使用重载的 + 操作符（C 语言没有重载）来连接引号内的字符串或字符串变量，而是使用特殊的占位符来表示数据将来的位置。而且它还将插入格式化字符串的参数，以逗号分隔，排成一行。例如：

```
System.out.printf("Row 1: [%d %f]%n", x, y);
```

这一行代码在运行的时候，首先将 `x` 的值插入到 `%d_` 的位置，然后将 `y` 的值插入到 `%f` 的位置。这些占位符叫做*格式修饰符*，它们不仅指明了插入数据的位置，同时还指明了将会插入什么类型的变量，以及如何格式化。在这个例子中 `%d` 表示 `x` 是一个整数，`%f` 表示 `y` 是一个浮点数（**`float`** 或者 **`double`**）。### `System.out.format()` Java SE5 引入了 `format()` 方法，可用于 `PrintStream` 或者 `PrintWriter` 对象（你可以在 流式 I/O 了解更多内容），其中也包括 `System.out` 对象。`format()` 方法模仿了 C 语言的 `printf()`。如果你比较怀旧的话，也可以使用 `printf()`。以下是一个简单的示例：

```
// strings/SimpleFormat.java

public class SimpleFormat {
    public static void main(String[] args) {
        int x = 5;
        double y = 5.332542;
        // The old way:
        System.out.println("Row 1: [" + x + " " + y + "]");
        // The new way:
        System.out.format("Row 1: [%d %f]%n", x, y);
        // or
        System.out.printf("Row 1: [%d %f]%n", x, y);
    }
}
/* Output:
Row 1: [5 5.332542]
Row 1: [5 5.332542]
Row 1: [5 5.332542]
*/
```

可以看到，`format()` 和 `printf()` 是等价的，它们只需要一个简单的格式化字符串，加上一串参数即可，每个参数对应一个格式修饰符。

`String` 类也有一个 **`static`** `format()` 方法，可以格式化字符串。

19.5.1 Formatter 类

在 Java 中，所有的格式化功能都是由 `java.util.Formatter` 类处理的。可以将 `Formatter` 看做一个翻译器，它将你的格式化字符串与数据翻译成需要的结果。当你创建一个 `Formatter` 对象时，需要向其构造器传递一些信息，告诉它最终的结果将向哪里输出：

```java
// strings/Turtle.java
import java.io.*;
import java.util.*;

public class Turtle {
    private String name;
    private Formatter f;
    public Turtle(String name, Formatter f) {
        this.name = name;
        this.f = f;
    }
    public void move(int x, int y) {
        f.format("%s The Turtle is at (%d,%d)%n",
            name, x, y);
    }
    public static void main(String[] args) {
        PrintStream outAlias = System.out;
        Turtle tommy = new Turtle("Tommy",
            new Formatter(System.out));
        Turtle terry = new Turtle("Terry",
            new Formatter(outAlias));
        tommy.move(0,0);
        terry.move(4,8);
        tommy.move(3,4);
        terry.move(2,5);
        tommy.move(3,3);
        terry.move(3,3);
    }
}
/* Output:
Tommy The Turtle is at (0,0)
Terry The Turtle is at (4,8)
Tommy The Turtle is at (3,4)
Terry The Turtle is at (2,5)
Tommy The Turtle is at (3,3)
Terry The Turtle is at (3,3)
*/
```

格式化修饰符 `%s` 表明这里需要 `String` 参数。

所有的 `tommy` 都将输出到 `System.out`，而所有的 `terry` 则都输出到 `System.out` 的一个别名中。`Formatter` 的重载构造器支持输出到多个路径，不过最常用的还是 `PrintStream()`（如上例）、`OutputStream` 和 `File`。你可以在 流式 I/O 中了解更多信息。### 格式化修饰符在插入数据时，如

果想要优化空格与对齐，你需要更精细复杂的格式修饰符。以下是其通用语法：

```
%[argument_index$][flags][width][.precision]conversion
```

最常见的应用是控制一个字段的最小长度，这可以通过指定 *width* 来实现。`Formatter`对象通过在必要时添加空格，来确保一个字段至少达到设定长度。默认情况下，数据是右对齐的，不过可以通过使用 - 标志来改变对齐方向。

与 *width* 相对的是 *precision*，用于指定最大长度。*width* 可以应用于各种类型的数据转换，并且其行为方式都一样。*precision* 则不然，当应用于不同类型的数据转换时，*precision* 的意义也不同。在将 *precision* 应用于 `String` 时，它表示打印 `string` 时输出字符的最大数量。而在将 *precision* 应用于浮点数时，它表示小数部分要显示出来的位数（默认是 6 位小数），如果小数位数过多则舍入，太少则在尾部补零。由于整数没有小数部分，所以 *precision* 无法应用于整数，如果你对整数应用 *precision*，则会触发异常。

下面的程序应用格式修饰符来打印一个购物收据。这是 *Builder* 设计模式的一个简单实现，即先创建一个初始对象，然后逐渐添加新东西，最后调用 `build()` 方法完成构建：

```java
// strings/ReceiptBuilder.java
import java.util.*;

public class ReceiptBuilder {
    private double total = 0;
    private Formatter f =
        new Formatter(new StringBuilder());
    public ReceiptBuilder() {
        f.format(
          "%-15s %5s %10s%n", "Item", "Qty", "Price");
        f.format(
          "%-15s %5s %10s%n", "----", "---", "-----");
        }
    public void add(String name, int qty, double price) {
        f.format("%-15.15s %5d %10.2f%n", name, qty, price);
        total += price * qty;
    }
    public String build() {
        f.format("%-15s %5s %10.2f%n", "Tax", "",
          total * 0.06);
        f.format("%-15s %5s %10s%n", "", "", "-----");
        f.format("%-15s %5s %10.2f%n", "Total", "",
          total * 1.06);
        return f.toString();
    }
    public static void main(String[] args) {
        ReceiptBuilder receiptBuilder =
          new ReceiptBuilder();
        receiptBuilder.add("Jack's Magic Beans", 4, 4.25);
        receiptBuilder.add("Princess Peas", 3, 5.1);
```

```
        receiptBuilder.add(
          "Three Bears Porridge", 1, 14.29);
        System.out.println(receiptBuilder.build());
    }
}
/* Output:
Item                Qty      Price
----                ---      -----
Jack's Magic Be       4       4.25
Princess Peas         3       5.10
Three Bears Por       1      14.29
Tax                           2.80
                             -----
Total                        49.39
*/
```

通过传入一个 `StringBuilder` 对象到 `Formatter` 的构造器，我们指定了一个容器来构建目标 `String`。你也可以通过不同的构造器参数，把结果输出到标准输出，甚至是一个文件里。

正如你所见，通过相当简洁的语法，`Formatter` 提供了对空格与对齐的强大控制能力。在该程序中，为了恰当地控制间隔，格式化字符串被重复利用了多遍。### `Formatter` 转换下面的表格展示了最常用的类型转换：

类型	含义
d	整型（十进制）
c	Unicode 字符
b	Boolean 值
s	String
f	浮点数（十进制）
e	浮点数（科学计数）
x	整型（十六进制）
h	散列码（十六进制）
%	字面值“%”

下面的程序演示了这些转换是如何工作的：

```
// strings/Conversion.java
import java.math.*;
import java.util.*;

```

```java
public class Conversion {
    public static void main(String[] args) {
        Formatter f = new Formatter(System.out);

        char u = 'a';
        System.out.println("u = 'a'");
        f.format("s: %s%n", u);
        // f.format("d: %d%n", u);
        f.format("c: %c%n", u);
        f.format("b: %b%n", u);
        // f.format("f: %f%n", u);
        // f.format("e: %e%n", u);
        // f.format("x: %x%n", u);
        f.format("h: %h%n", u);

        int v = 121;
        System.out.println("v = 121");
        f.format("d: %d%n", v);
        f.format("c: %c%n", v);
        f.format("b: %b%n", v);
        f.format("s: %s%n", v);
        // f.format("f: %f%n", v);
        // f.format("e: %e%n", v);
        f.format("x: %x%n", v);
        f.format("h: %h%n", v);

        BigInteger w = new BigInteger("50000000000000");
        System.out.println(
          "w = new BigInteger(\"50000000000000\")");
        f.format("d: %d%n", w);
        // f.format("c: %c%n", w);
        f.format("b: %b%n", w);
        f.format("s: %s%n", w);
        // f.format("f: %f%n", w);
        // f.format("e: %e%n", w);
        f.format("x: %x%n", w);
        f.format("h: %h%n", w);

        double x = 179.543;
        System.out.println("x = 179.543");
        // f.format("d: %d%n", x);
        // f.format("c: %c%n", x);
        f.format("b: %b%n", x);
        f.format("s: %s%n", x);
        f.format("f: %f%n", x);
        f.format("e: %e%n", x);
        // f.format("x: %x%n", x);
        f.format("h: %h%n", x);

        Conversion y = new Conversion();
        System.out.println("y = new Conversion()");
```

```

        // f.format("d: %d%n", y);
        // f.format("c: %c%n", y);
        f.format("b: %b%n", y);
        f.format("s: %s%n", y);
        // f.format("f: %f%n", y);
        // f.format("e: %e%n", y);
        // f.format("x: %x%n", y);
        f.format("h: %h%n", y);

        boolean z = false;
        System.out.println("z = false");
        // f.format("d: %d%n", z);
        // f.format("c: %c%n", z);
        f.format("b: %b%n", z);
        f.format("s: %s%n", z);
        // f.format("f: %f%n", z);
        // f.format("e: %e%n", z);
        // f.format("x: %x%n", z);
        f.format("h: %h%n", z);
    }
}
/* Output:
u = 'a'
s: a
c: a
b: true
h: 61
v = 121
d: 121
c: y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("50000000000000")
d: 50000000000000
b: true
s: 50000000000000
x: 2d79883d2000
h: 8842a1a7
x = 179.543
b: true
s: 179.543
f: 179.543000
e: 1.795430e+02
h: 1ef462c
y = new Conversion()
b: true
s: Conversion@15db9742
h: 15db9742
```

```
z = false
b: false
s: false
h: 4d5
*/
```

被注释的代码表示，针对相应类型的变量，这些转换是无效的。如果执行这些转换，则会触发异常。

注意，程序中的每个变量都用到了 b 转换。虽然它对各种类型都是合法的，但其行为却不一定与你想象的一致。对于 **boolean** 基本类型或 Boolean 对象，其转换结果是对应的 **true** 或 **false**。但是，对其他类型的参数，只要该参数不为 **null**，其转换结果永远都是 **true**。即使是数字 0，转换结果依然为 **true**，而这在其他语言中（包括 C），往往转换为 **false**。所以，将 b 应用于非布尔类型的对象时请格外小心。

还有许多不常用的类型转换与格式修饰符选项，你可以在 JDK 文档中的 Formatter 类部分找到它们。### String.format() Java SE5 也参考了 C 中的 sprintf() 方法，以生成格式化的 String 对象。String.format() 是一个 **static** 方法，它接受与 Formatter.format() 方法一样的参数，但返回一个 String 对象。当你只需使用一次 format() 方法的时候，String.format() 用起来很方便。例如：

```java
// strings/DatabaseException.java

public class DatabaseException extends Exception {
    public DatabaseException(int transactionID,
      int queryID, String message) {
      super(String.format("(t%d, q%d) %s", transactionID,
        queryID, message));
    }
    public static void main(String[] args) {
      try {
        throw new DatabaseException(3, 7, "Write failed");
      } catch(Exception e) {
        System.out.println(e);
      }
    }
}
/*
Output:
DatabaseException: (t3, q7) Write failed
*/
```

其实在 String.format() 内部，它也是创建了一个 Formatter 对象，然后将你传入的参数转给 Formatter。不过，与其自己做这些事情，不如使用便捷的 String.format() 方法，何况这样的代码更清晰易读。#### 一个十六进制转储（dump）工具在第二个例子中，我们把二进制文件转换为十六进制格式。下面的小工具使用了 String.format() 方法，以可读的十六进制格式将字节数组打印

出来:

```
// strings/Hex.java
// {java onjava.Hex}
package onjava;
import java.io.*;
import java.nio.file.*;

public class Hex {
    public static String format(byte[] data) {
        StringBuilder result = new StringBuilder();
        int n = 0;
        for(byte b : data) {
            if(n % 16 == 0)
                result.append(String.format("%05X: ", n));
            result.append(String.format("%02X ", b));
            n++;
            if(n % 16 == 0) result.append("\n");
        }
        result.append("\n");
        return result.toString();
    }
    public static void main(String[] args) throws Exception {
        if(args.length == 0)
            // Test by displaying this class file:
            System.out.println(format(
                Files.readAllBytes(Paths.get(
                  "build/classes/main/onjava/Hex.class"))));
        else
            System.out.println(format(
                Files.readAllBytes(Paths.get(args[0]))));
    }
}
/* Output: (First 6 Lines)
00000: CA FE BA BE 00 00 00 34 00 61 0A 00 05 00 31 07
00010: 00 32 0A 00 02 00 31 08 00 33 07 00 34 0A 00 35
00020: 00 36 0A 00 0F 00 37 0A 00 02 00 38 08 00 39 0A
00030: 00 3A 00 3B 08 00 3C 0A 00 02 00 3D 09 00 3E 00
00040: 3F 08 00 40 07 00 41 0A 00 42 00 43 0A 00 44 00
00050: 45 0A 00 14 00 46 0A 00 47 00 48 07 00 49 01 00
                  ...
*/
```

为了打开及读入二进制文件，我们用到了另一个工具 `Files.readAllBytes()`，这已经在 Files 章节 介绍过了。这里的 `readAllBytes()` 方法将整个文件以 **byte** 数组的形式返回。

一定有人改了我的代码。

19.6 正则表达式

很久之前，*正则表达式*就已经整合到标准 Unix 工具集之中，例如 sed、awk 和程序语言之中了，如 Python 和 Perl（有些人认为正是正则表达式促成了 Perl 的成功）。而在 Java 中，字符串操作还主要集中于`String`、`StringBuffer` 和 `StringTokenizer` 类。与正则表达式相比较，它们只能提供相当简单的功能。

正则表达式是一种强大而灵活的文本处理工具。使用正则表达式，我们能够以编程的方式，构造复杂的文本模式，并对输入 `String` 进行搜索。一旦找到了匹配这些模式的部分，你就能随心所欲地对它们进行处理。初学正则表达式时，其语法是一个难点，但它确实是一种简洁、动态的语言。正则表达式提供了一种完全通用的方式，能够解决各种 `String` 处理相关的问题：匹配、选择、编辑以及验证。### 基础一般来说，正则表达式就是以某种方式来描述字符串，因此你可以说："如果一个字符串含有这些东西，那么它就是我正在找的东西。"例如，要找一个数字，它可能有一个负号在最前面，那你就写一个负号加上一个问号，就像这样：

```
-?
```

要描述一个整数，你可以说它有一位或多位阿拉伯数字。在正则表达式中，用 `\d` 表示一位数字。如果在其他语言中使用过正则表达式，那你可能就能发现 Java 对反斜线 的不同处理方式。在其他语言中，`\\` 表示"我想要在正则表达式中插入一个普通的（字面上的）反斜线，请不要给它任何特殊的意义。"而在 Java 中，`\\` 的意思是"我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。"例如，如果你想表示一位数字，那么正则表达式应该是 `\\d`。如果你想插入一个普通的反斜线，应该这样写 `\\\\`。不过换行符和制表符之类的东西只需要使用单反斜线：`\n\t`。[2]

要表示"一个或多个之前的表达式"，应该使用 `+`。所以，如果要表示"可能有一个负号，后面跟着一位或多位数字"，可以这样：

```
-?\\d+
```

应用正则表达式最简单的途径，就是利用 `String` 类内建的功能。例如，你可以检查一个 `String` 是否匹配如上所述的正则表达式：

```java
// strings/IntegerMatch.java

public class IntegerMatch {
    public static void main(String[] args) {
        System.out.println("-1234".matches("-?\\d+"));
        System.out.println("5678".matches("-?\\d+"));
        System.out.println("+911".matches("-?\\d+"));
        System.out.println("+911".matches("(-|\\+)?\\d+"));
    }
```

[2]事实证明，质数实际上并不是散列桶的理想容量。近来，（经过广泛的测试）Java 的散列函数都使用 2 的整数次方。对现代的处理器来说，除法与求余数是最慢的操作。使用 2 的整数次方长度的散列表，可用掩码代替除法。

```
}
/* Output:
true
true
false
true
*/
```

前两个字符串都满足对应的正则表达式，匹配成功。第三个字符串以 + 开头，这也是一个合法的符号，但与对应的正则表达式却不匹配。因此，我们的正则表达式应该描述为："可能以一个加号或减号开头"。在正则表达式中，用括号将表达式进行分组，用竖线 | 表示或操作。也就是：

```
(-|\\+)?
```

这个正则表达式表示字符串的起始字符可能是一个 - 或 +，或者二者都没有（因为后面跟着 ? 修饰符）。因为字符 + 在正则表达式中有特殊的意义，所以必须使用 \\ 将其转义，使之成为表达式中的一个普通字符。

`String`类还自带了一个非常有用的正则表达式工具——`split()` 方法，其功能是"将字符串从正则表达式匹配的地方切开。"

```java
// strings/Splitting.java import java.util.*;

public class Splitting {
    public static String knights =
      "Then, when you have found the shrubbery, " +
      "you must cut down the mightiest tree in the " +
      "forest...with... a herring!";
    public static void split(String regex) {
        System.out.println(
          Arrays.toString(knights.split(regex)));
        }
    public static void main(String[] args) {
        split(" "); // Doesn't have to contain regex chars
        split("\\W+"); // Non-word characters
        split("n\\W+"); // 'n' followed by non-words
    }
}
/* Output:
[Then,, when, you, have, found, the, shrubbery,, you,
must, cut, down, the, mightiest, tree, in, the,
forest...with..., a, herring!]
[Then, when, you, have, found, the, shrubbery, you,
must, cut, down, the, mightiest, tree, in, the, forest,
with, a, herring]
[The, whe, you have found the shrubbery, you must cut
dow, the mightiest tree i, the forest...with... a
herring!]
*/
```

首先看第一个语句，注意这里用的是普通的字符作为正则表达式，其中并不包含任何特殊字符。因此第一个 split() 只是按空格来划分字符串。

第二个和第三个 split() 都用到了 \\W，它的意思是一个非单词字符（如果 W 小写，\\w，则表示一个单词字符）。通过第二个例子可以看到，它将标点字符删除了。第三个 split() 表示“字母 n 后面跟着一个或多个非单词字符。”可以看到，在原始字符串中，与正则表达式匹配的部分，在最终结果中都不存在了。

String.split() 还有一个重载的版本，它允许你限制字符串分割的次数。

用正则表达式进行替换操作时，你可以只替换第一处匹配，也可以替换所有的匹配：

```
// strings/Replacing.java

public class Replacing {
    static String s = Splitting.knights;
    public static void main(String[] args) {
        System.out.println(
          s.replaceFirst("f\\w+", "located"));
        System.out.println(
          s.replaceAll("shrubbery|tree|herring","banana"));
    }
}
/* Output:
Then, when you have located the shrubbery, you must cut
down the mightiest tree in the forest...with... a
herring!
Then, when you have found the banana, you must cut down
the mightiest banana in the forest...with... a banana!
*/
```

第一个表达式要匹配的是，以字母 f 开头，后面跟一个或多个字母（注意这里的 w 是小写的）。并且只替换掉第一个匹配的部分，所以 “found” 被替换成 “located”。

第二个表达式要匹配的是三个单词中的任意一个，因为它们以竖线分割表示“或”，并且替换所有匹配的部分。

稍后你会看到，String 之外的正则表达式还有更强大的替换工具，例如，可以通过方法调用执行替换。而且，如果正则表达式不是只使用一次的话，非 String 对象的正则表达式明显具备更佳的性能。### 创建正则表达式我们首先从正则表达式可能存在的构造集中选取一个很有用的子集，以此开始学习正则表达式。正则表达式的完整构造子列表，请参考 JDK 文档 java.util.regex 包中的 Pattern类。

表达式	含义
B	指定字符B
\xhh	十六进制值为0xhh的字符

表达式	含义
`\uhhhh`	十六进制表现为`0xhhhh`的 Unicode 字符
`\t`	制表符 Tab
`\n`	换行符
`\r`	回车
`\f`	换页
`\e`	转义（Escape）

当你学会了使用字符类（character classes）之后，正则表达式的威力才能真正显现出来。以下是一些创建字符类的典型方式，以及一些预定义的类：

表达式	含义
`.`	任意字符
`[abc]`	包含`a`、`b`或`c`的任何字符（和`a|b|c`作用相同）
`[^abc]`	除`a`、`b`和`c`之外的任何字符（否定）
`[a-zA-Z]`	从`a`到`z`或从`A`到`Z`的任何字符（范围）
`[abc[hij]]`	`a`、`b`、`c`、`h`、`i`、`j`中的任意字符（与`a|b|c|h|i|j`作用相同）（合并）
`[a-z&&[hij]]`	任意`h`、`i`或`j`（交）
`\s`	空白符（空格、tab、换行、换页、回车）
`\S`	非空白符（`[^\s]`）
`\d`	数字（`[0-9]`）
`\D`	非数字（`[^0-9]`）
`\w`	词字符（`[a-zA-Z_0-9]`）
`\W`	非词字符（`[^\w]`）

这里只列出了部分常用的表达式，你应该将 JDK 文档中 `java.util.regex.Pattern` 那一页加入浏览器书签中，以便在需要的时候方便查询。

一匹真正的好马，即使在鞭子的影子下，也能飞奔。

逻辑操作符	含义
XY	Y跟在X后面
X|Y	X或Y
(X)	捕获组（capturing group）。可以在表达式中用\i引用第 i 个捕获组

下面是不同的边界匹配符：

边界匹配符	含义
^	一行的开始
$	一行的结束
\b	词的边界
\B	非词的边界
\G	前一个匹配的结束

作为演示，下面的每一个正则表达式都能成功匹配字符序列“Rudolph”：

```
// strings/Rudolph.java

public class Rudolph {
    public static void main(String[] args) {
        for(String pattern : new String[]{
          "Rudolph",
          "[rR]udolph",
          "[rR][aeiou][a-z]ol.*",
          "R.*" })
        System.out.println("Rudolph".matches(pattern));
    }
}
/* Output:
true
true
true
true
*/
```

我们的目的并不是编写最难理解的正则表达式，而是尽量编写能够完成任务的、最简单以及最必要的正则表达式。一旦真正开始使用正则表达式了，你就会发现，在编写新的表达式之前，你通常会参考代码中已经用到的正则表达式。### 量词量词描述了一个模式捕获输入文本的方式：

已经做好了，但还有一些细节要调一下。

- **贪婪型**：量词总是贪婪的，除非有其他的选项被设置。贪婪表达式会为所有可能的模式发现尽可能多的匹配。导致此问题的一个典型理由就是假定我们的模式仅能匹配第一个可能的字符组，如果它是贪婪的，那么它就会继续往下匹配。
- **勉强型**：用问号来指定，这个量词匹配满足模式所需的最少字符数。因此也被称作懒惰的、最少匹配的、非贪婪的或不贪婪的。
- **占有型**：目前，这种类型的量词只有在 Java 语言中才可用（在其他语言中不可用），并且也更高级，因此我们大概不会立刻用到它。当正则表达式被应用于 `String` 时，它会产生相当多的状态，以便在匹配失败时可以回溯。而“占有的”量词并不保存这些中间状态，因此它们可以防止回溯。它们常常用于防止正则表达式失控，因此可以使正则表达式执行起来更高效。

贪婪型	勉强型	占有型	如何匹配
`X?`	`X??`	`X?+`	一个或零个`X`
`X*`	`X*?`	`X*+`	零个或多个`X`
`X+`	`X+?`	`X++`	一个或多个`X`
`X{n}`	`X{n}?`	`X{n}+`	恰好`n`次`X`
`X{n,}`	`X{n,}?`	`X{n,}+`	至少`n`次`X`
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	`X`至少`n`次，但不超过`m`次

应该非常清楚地意识到，表达式 `X` 通常必须要用圆括号括起来，以便它能够按照我们期望的效果去执行。例如：

```
abc+
```

看起来它似乎应该匹配 1 个或多个`abc`序列，如果我们把它应用于输入字符串`abcabcabc`，则实际上会获得 3 个匹配。然而，这个表达式实际上表示的是：匹配`ab`，后面跟随 1 个或多个`c`。要表明匹配 1 个或多个完整的字符串`abc`，我们必须这样表示：

```
(abc)+
```

你会发现，在使用正则表达式时很容易混淆，因为它是一种在 Java 之上的新语言。### `CharSequence` 接口 `CharSequence` 从 `CharBuffer`、`String`、`StringBuffer`、`StringBuilder` 类中抽象出了字符序列的一般化定义：

```
interface CharSequence {
    char charAt(int i);
    int length();
    CharSequence subSequence(int start, int end);
```

以前没这样过的。

```
    String toString();
}
```

因此，这些类都实现了该接口。多数正则表达式操作都接受 CharSequence 类型参数。### Pattern 和 Matcher 通常，比起功能有限的 String 类，我们更愿意构造功能强大的正则表达式对象。只需导入 java.util.regex包，然后用 static Pattern.compile() 方法来编译你的正则表达式即可。它会根据你的 String 类型的正则表达式生成一个 Pattern 对象。接下来，把你想要检索的字符串传入 Pattern 对象的 matcher() 方法。matcher() 方法会生成一个 Matcher 对象，它有很多功能可用（可以参考 java.util.regext.Matcher 的 JDK 文档）。例如，它的 replaceAll() 方法能将所有匹配的部分都替换成你传入的参数。

作为第一个示例，下面的类可以用来测试正则表达式，看看它们能否匹配一个输入字符串。第一个控制台参数是将要用来搜索匹配的输入字符串，后面的一个或多个参数都是正则表达式，它们将被用来在输入的第一个字符串中查找匹配。在 Unix/Linux 上，命令行中的正则表达式必须用引号括起来。这个程序在测试正则表达式时很有用，特别是当你想验证它们是否具备你所期待的匹配功能的时候。[3]

```
// strings/TestRegularExpression.java
// Simple regular expression demonstration
// {java TestRegularExpression
// abcabcabcdefabc "abc+" "(abc)+" }
import java.util.regex.*;

public class TestRegularExpression {
    public static void main(String[] args) {
        if(args.length < 2) {
            System.out.println(
              "Usage:\njava TestRegularExpression " +
              "characterSequence regularExpression+");
            System.exit(0);
        }
        System.out.println("Input: \"" + args[0] + "\"");
        for(String arg : args) {
            System.out.println(
              "Regular expression: \"" + arg + "\"");
            Pattern p = Pattern.compile(arg);
            Matcher m = p.matcher(args[0]);
            while(m.find()) {
                System.out.println(
                  "Match \"" + m.group() + "\" at positions " +
                m.start() + "-" + (m.end() - 1));
            }
        }
    }
}
/* Output:
```

[3]请注意，在 64 位处理器上可能不会发生这种情况，从而消除了这个问题。

```
Input: "abcabcabcdefabc"
Regular expression: "abcabcabcdefabc"
Match "abcabcabcdefabc" at positions 0-14
Regular expression: "abc+"
Match "abc" at positions 0-2
Match "abc" at positions 3-5
Match "abc" at positions 6-8
Match "abc" at positions 12-14
Regular expression: "(abc)+"
Match "abcabcabc" at positions 0-8
Match "abc" at positions 12-14
*/
```

还可以在控制台参数中加入“`(abc){2,}`”，看看执行结果。

`Pattern` 对象表示编译后的正则表达式。从这个例子可以看到，我们使用已编译的 `Pattern` 对象上的 `matcher()` 方法，加上一个输入字符串，从而共同构造了一个 `Matcher` 对象。同时，`Pattern` 类还提供了一个**static**方法：

```
static boolean matches(String regex, CharSequence input)
```

该方法用以检查 `regex` 是否匹配整个 `CharSequence` 类型的 `input` 参数。编译后的 `Pattern` 对象还提供了 `split()` 方法，它从匹配了 `regex` 的地方分割输入字符串，返回分割后的子字符串 `String` 数组。

通过调用 `Pattern.matcher()` 方法，并传入一个字符串参数，我们得到了一个 `Matcher` 对象。使用 `Matcher` 上的方法，我们将能够判断各种不同类型的匹配是否成功：

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

其中的 `matches()` 方法用来判断整个输入字符串是否匹配正则表达式模式，而 `lookingAt()` 则用来判断该字符串（不必是整个字符串）的起始部分是否能够匹配模式。### `find()` `Matcher.find()` 方法可用来在 `CharSequence` 中查找多个匹配。例如：

```
// strings/Finding.java
import java.util.regex.*;

public class Finding {
    public static void main(String[] args) {
        Matcher m = Pattern.compile("\\w+")
          .matcher(
            "Evening is full of the linnet's wings");
        while(m.find())
            System.out.print(m.group() + " ");
        System.out.println();
        int i = 0;
```

```
        while(m.find(i)) {
            System.out.print(m.group() + " ");
            i++;
        }
    }
}
/* Output:
Evening is full of the linnet s wings
Evening vening ening ning ing ng g is is s full full
ull ll l of of f the the he e linnet linnet innet nnet
net et t s s wings wings ings ngs gs s
*/
```

模式 `\\w+` 将字符串划分为词。`find()` 方法像迭代器那样向前遍历输入字符串。而第二个重载的 `find()` 接收一个整型参数，该整数表示字符串中字符的位置，并以其作为搜索的起点。从结果可以看出，后一个版本的 `find()` 方法能够根据其参数的值，不断重新设定搜索的起始位置。### 组（Groups）组是用括号划分的正则表达式，可以根据组的编号来引用某个组。组号为 0 表示整个表达式，组号 1 表示被第一对括号括起来的组，以此类推。因此，下面这个表达式，

```
A(B(C))D
```

中有三个组：组 0 是 `ABCD`，组 1 是 `BC`，组 2 是 `C`。

`Matcher` 对象提供了一系列方法，用以获取与组相关的信息：

- `public int groupCount()` 返回该匹配器的模式中的分组数目，组 0 不包括在内。
- `public String group()` 返回前一次匹配操作（例如 `find()`）的第 0 组（整个匹配）。
- `public String group(int i)` 返回前一次匹配操作期间指定的组号，如果匹配成功，但是指定的组没有匹配输入字符串的任何部分，则将返回 `null`。
- `public int start(int group)` 返回在前一次匹配操作中寻找到的组的起始索引。
- `public int end(int group)` 返回在前一次匹配操作中寻找到的组的最后一个字符索引加一的值。

下面是正则表达式组的例子：

```
// strings/Groups.java
import java.util.regex.*;

public class Groups {
    public static final String POEM =
      "Twas brillig, and the slithy toves\n" +
      "Did gyre and gimble in the wabe.\n" +
      "All mimsy were the borogoves,\n" +
      "And the mome raths outgrabe.\n\n" +
      "Beware the Jabberwock, my son,\n" +
      "The jaws that bite, the claws that catch.\n" +
      "Beware the Jubjub bird, and shun\n" +
      "The frumious Bandersnatch.";
```

```
    public static void main(String[] args) {
        Matcher m = Pattern.compile(
          "(?m)(\\S+)\\s+((\\S+)\\s+(\\S+))$")
          .matcher(POEM);
        while(m.find()) {
            for(int j = 0; j <= m.groupCount(); j++)
                System.out.print("[" + m.group(j) + "]");
            System.out.println();
        }
    }
}
/* Output:
[the slithy toves][the][slithy toves][slithy][toves]
[in the wabe.][in][the wabe.][the][wabe.]
[were the borogoves,][were][the
borogoves,][the][borogoves,]
[mome raths outgrabe.][mome][raths
outgrabe.][raths][outgrabe.]
[Jabberwock, my son,][Jabberwock,][my son,][my][son,]
[claws that catch.][claws][that catch.][that][catch.]
[bird, and shun][bird,][and shun][and][shun]
[The frumious Bandersnatch.][The][frumious
Bandersnatch.][frumious][Bandersnatch.]
*/
```

这首诗来自于 Lewis Carroll 所写的 *Through the Looking Glass* 中的 “Jabberwocky”。可以看到这个正则表达式模式有许多圆括号分组，由任意数目的非空白符（`\\S+`）及随后的任意数目的空白符（`\\s+`）所组成。目的是捕获每行的最后 3 个词，每行最后以 `\$` 结束。不过，在正常情况下是将 `\$` 与整个输入序列的末端相匹配。所以我们一定要显式地告知正则表达式注意输入序列中的换行符。这可以由序列开头的模式标记（`?m`）来完成（模式标记马上就会介绍）。### `start()` 和 `end()` 在匹配操作成功之后，`start()` 返回先前匹配的起始位置的索引，而 `end()` 返回所匹配的最后字符的索引加一的值。匹配操作失败之后（或先于一个正在进行的匹配操作去尝试）调用 `start()` 或 `end()` 将会产生 `IllegalStateException`。下面的示例还同时展示了 `matches()` 和 `lookingAt()` 的用法 [4]：

```
// strings/StartEnd.java
import java.util.regex.*;

public class StartEnd {
    public static String input =
      "As long as there is injustice, whenever a\n" +
      "Targathian baby cries out, wherever a distress\n" +
      "signal sounds among the stars " +
      "... We'll be there.\n"+
      "This fine ship, and this fine crew ...\n" +
      "Never give up! Never surrender!";
```

[4]这个测试的推论是，“如果某人表示线程是容易并且简单的，请确保这个人没有对你的项目做出重要的决策。如果那个人已经做出，那么你就已经陷入麻烦之中了。”

```
    private static class Display {
        private boolean regexPrinted = false;
        private String regex;
        Display(String regex) { this.regex = regex; }

        void display(String message) {
            if(!regexPrinted) {
                System.out.println(regex);
                regexPrinted = true;
            }
            System.out.println(message);
        }
    }

    static void examine(String s, String regex) {
        Display d = new Display(regex);
        Pattern p = Pattern.compile(regex);
        Matcher m = p.matcher(s);
        while(m.find())
            d.display("find() '" + m.group() +
              "' start = "+ m.start() + " end = " + m.end());
            if(m.lookingAt()) // No reset() necessary
                d.display("lookingAt() start = "
                  + m.start() + " end = " + m.end());
        if(m.matches()) // No reset() necessary
            d.display("matches() start = "
              + m.start() + " end = " + m.end());
    }

    public static void main(String[] args) {
        for(String in : input.split("\n")) {
            System.out.println("input : " + in);
            for(String regex : new String[]{"\\w*ere\\w*",
              "\\w*ever", "T\\w+", "Never.*?!"})
                examine(in, regex);
        }
    }
}
/* Output:
input : As long as there is injustice, whenever a
\w*ere\w*
find() 'there' start = 11 end = 16
\w*ever
find() 'whenever' start = 31 end = 39
input : Targathian baby cries out, wherever a distress
\w*ere\w*
find() 'wherever' start = 27 end = 35
\w*ever
find() 'wherever' start = 27 end = 35
T\w+ find() 'Targathian' start = 0 end = 10
lookingAt() start = 0 end = 10
```

```
input : signal sounds among the stars ... We'll be
there.
\w*ere\w*
find() 'there' start = 43 end = 48
input : This fine ship, and this fine crew ...
T\w+ find() 'This' start = 0 end = 4
lookingAt() start = 0 end = 4
input : Never give up! Never surrender!
\w*ever
find() 'Never' start = 0 end = 5
find() 'Never' start = 15 end = 20
lookingAt() start = 0 end = 5
Never.*?!
find() 'Never give up!' start = 0 end = 14
find() 'Never surrender!' start = 15 end = 31
lookingAt() start = 0 end = 14
matches() start = 0 end = 31
*/
```

注意，`find()` 可以在输入的任意位置定位正则表达式，而 `lookingAt()` 和 `matches()` 只有在正则表达式与输入的最开始处就开始匹配时才会成功。`matches()` 只有在整个输入都匹配正则表达式时才会成功，而 `lookingAt()` [5] 只要输入的第一部分匹配就会成功。### `Pattern` 标记 `Pattern` 类的 `compile()` 方法还有另一个版本，它接受一个标记参数，以调整匹配行为：

```
Pattern Pattern.compile(String regex, int flag)
```

其中的 `flag` 来自以下 `Pattern` 类中的常量

编译标记	效果
`Pattern.CANON_EQ`	当且仅当两个字符的完全规范分解相匹配时，才认为它们是匹配的。例如，如果我们指定这个标记，表达式`\u003F`就会匹配字符串`?`。默认情况下，匹配不考虑规范的等价性
`Pattern.CASE_INSENSITIVE(?i)`	默认情况下，大小写不敏感的匹配假定只有 US-ASCII 字符集中的字符才能进行。这个标记允许模式匹配不考虑大小写（大写或小写）。通过指定`UNICODE_CASE`标记及结合此标记。基于 Unicode 的大小写不敏感的匹配就可以开启了

[5]这在即将产生的 C++ 的标准中得到了补救。

有两种方法能写出没有错误的程序；但只有第三种好用。

编译标记	效果
`Pattern.COMMENTS(?x)`	在这种模式下，空格符将被忽略掉，并且以##开始直到行末的注释也会被忽略掉。通过嵌入的标记表达式也可以开启 Unix 的行模式
`Pattern.DOTALL(?s)`	在 dotall 模式下，表达式.匹配所有字符，包括行终止符。默认情况下，.不会匹配行终止符
`Pattern.MULTILINE(?m)`	在多行模式下，表达式^和$分别匹配一行的开始和结束。^还匹配输入字符串的开始，而$还匹配输入字符串的结尾。默认情况下，这些表达式仅匹配输入的完整字符串的开始和结束
`Pattern.UNICODE_CASE(?u)`	当指定这个标记，并且开启CASE_INSENSITIVE时，大小写不敏感的匹配将按照与 Unicode 标准相一致的方式进行。默认情况下，大小写不敏感的匹配假定只能在 US-ASCII 字符集中的字符才能进行
`Pattern.UNIX_LINES(?d)`	在这种模式下，在.、^和$的行为中，只识别行终止符\n

在这些标记中，`Pattern.CASE_INSENSITIVE`、`Pattern.MULTILINE` 以及 `Pattern.COMMENTS`（对声明或文档有用）特别有用。请注意，你可以直接在正则表达式中使用其中的大多数标记，只需要将上表中括号括起来的字符插入到正则表达式中，你希望它起作用的位置即可。

你还可以通过“或”(|) 操作符组合多个标记的功能：

```java
// strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
    public static void main(String[] args) {
        Pattern p =  Pattern.compile("^java",
          Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
        Matcher m = p.matcher(
          "java has regex\nJava has regex\n" +
          "JAVA has pretty good regular expressions\n" +
          "Regular expressions are in Java");
        while(m.find())
            System.out.println(m.group());
    }
}
```

```
/* Output:
java
Java
JAVA
*/
```

在这个例子中，我们创建了一个模式，它将匹配所有以“java”、“Java”和“JAVA”等开头的行，并且是在设置了多行标记的状态下，对每一行（从字符序列的第一个字符开始，至每一个行终止符）都进行匹配。注意，`group()` 方法只返回已匹配的部分。### `split()` `split()`方法将输入 `String` 断开成 `String` 对象数组，断开边界由正则表达式确定：

```
String[] split(CharSequence input)
String[] split(CharSequence input, int limit)
```

这是一个快速而方便的方法，可以按照通用边界断开输入文本：

```
// strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;

public class SplitDemo {
    public static void main(String[] args) {
        String input =
          "This!!unusual use!!of exclamation!!points";
        System.out.println(Arrays.toString(
        Pattern.compile("!!").split(input)));
        // Only do the first three:
        System.out.println(Arrays.toString(
        Pattern.compile("!!").split(input, 3)));
    }
}
/* Output:
[This, unusual use, of exclamation, points]
[This, unusual use, of exclamation!!points]
*/
```

第二种形式的 `split()` 方法可以限制将输入分割成字符串的数量。### 替换操作正则表达式在进行文本替换时特别方便，它提供了许多方法：+ `replaceFirst(String replacement)` 以参数字符串 `replacement` 替换掉第一个匹配成功的部分。+ `replaceAll(String replacement)` 以参数字符串 `replacement` 替换所有匹配成功的部分。+ `appendReplacement(StringBuffer sbuf, String replacement)` 执行渐进式的替换，而不是像 `replaceFirst()` 和 `replaceAll()` 那样只替换第一个匹配或全部匹配。这是一个非常重要的方法。它允许你调用其他方法来生成或处理 `replacement`（`replaceFirst()` 和 `replaceAll()` 则只能使用一个固定的字符串），使你能够以编程的方式将目标分割成组，从而具备更强大的替换功能。+ `appendTail(StringBuffer sbuf)` 在执行了一次或多次 `appendReplacement()` 之后，调用此方法可以将输入字符串余下的部分复制到 `sbuf` 中。

下面的程序演示了如何使用这些替换方法。开头部分注释掉的文本，就是正则表达式要处理的输入字符串：

```
// strings/TheReplacements.java
import java.util.regex.*;
import java.nio.file.*;
import java.util.stream.*;

/*! Here's a block of text to use as input to
    the regular expression matcher. Note that we
    first extract the block of text by looking for
    the special delimiters, then process the
    extracted block. !*/

public class TheReplacements {
    public static void main(String[] args) throws Exception {
        String s = Files.lines(
          Paths.get("TheReplacements.java"))
          .collect(Collectors.joining("\n"));
        // Match specially commented block of text above:
        Matcher mInput = Pattern.compile(
          "/\\*!(.*)!\\*/", Pattern.DOTALL).matcher(s);
        if(mInput.find())
            s = mInput.group(1); // Captured by parentheses
        // Replace two or more spaces with a single space:
        s = s.replaceAll(" {2,}", " ");
        // Replace 1+ spaces at the beginning of each
        // line with no spaces. Must enable MULTILINE mode:
        s = s.replaceAll("(?m)^ +", "");
        System.out.println(s);
        s = s.replaceFirst("[aeiou]", "(VOWEL1)");
        StringBuffer sbuf = new StringBuffer();
        Pattern p = Pattern.compile("[aeiou]");
        Matcher m = p.matcher(s);
        // Process the find information as you
        // perform the replacements:
        while(m.find())
            m.appendReplacement(sbuf, m.group().toUpperCase());
        // Put in the remainder of the text:
        m.appendTail(sbuf);
        System.out.println(sbuf);
    }
}
/* Output:
Here's a block of text to use as input to
the regular expression matcher. Note that we
first extract the block of text by looking for
the special delimiters, then process the
extracted block.
H(VOWEL1)rE's A blOck Of tExt tO UsE As InpUt tO
```

```
thE rEgUlAr ExprEssIOn mAtchEr. NOtE thAt wE
fIrst ExtrAct thE blOck Of tExt by lOOkIng fOr
thE spEcIAl dElImItErs, thEn prOcEss thE
ExtrActEd blOck.
*/
```

此处使用上一章介绍过的 `Files` 类打开并读入文件。`Files.lines()` 返回一个 `Stream` 对象，包含读入的所有行，`Collectors.joining()` 在每一行的结尾追加参数字符序列，最终拼接成一个 `String` 对象。

`mInput` 匹配 `/*!` 和 `! */` 之间的所有文字（注意分组的括号）。接下来，将存在两个或两个以上空格的地方，缩减为一个空格，并且删除每行开头部分的所有空格（为了使每一行都达到这个效果，而不仅仅是删除文本开头部分的空格，这里特意开启了多行模式）。这两个替换操作所使用的的 `replaceAll()` 是 `String` 对象自带的方法，在这里，使用此方法更方便。注意，因为这两个替换操作都只使用了一次 `replaceAll()`，所以，与其编译为 `Pattern`，不如直接使用 `String` 的 `replaceAll()` 方法，而且开销也更小些。

`replaceFirst()` 只对找到的第一个匹配进行替换。此外，`replaceFirst()` 和 `replaceAll()` 方法用来替换的只是普通字符串，所以，如果想对这些替换字符串进行某些特殊处理，这两个方法时无法胜任的。如果你想要那么做，就应该使用 `appendReplacement()` 方法。该方法允许你在执行替换的过程中，操作用来替换的字符串。在这个例子中，先构造了 `sbuf` 用来保存最终结果，然后用 `group()` 选择一个组，并对其进行处理，将正则表达式找到的元音字母替换成大些字母。一般情况下，你应该遍历执行所有的替换操作，然后再调用 `appendTail()` 方法，但是，如果你想模拟 `replaceFirst()`（或替换 n 次）的行为，那就只需要执行一次替换，然后调用 `appendTail()` 方法，将剩余未处理的部分存入 `sbuf` 即可。

同时，`appendReplacement()` 方法还允许你通过 `\$g` 直接找到匹配的某个组，这里的 `g` 就是组号。然而，它只能应付一些简单的处理，无法实现类似前面这个例子中的功能。### `reset()` 通过 `reset()` 方法，可以将现有的 `Matcher` 对象应用于一个新的字符序列：

```java
// strings/Resetting.java
import java.util.regex.*;

public class Resetting {
    public static void main(String[] args) throws Exception {
        Matcher m = Pattern.compile("[frb][aiu][gx]")
          .matcher("fix the rug with bags");
        while(m.find())
            System.out.print(m.group() + " ");
        System.out.println();
        m.reset("fix the rig with rags");
        while(m.find())
            System.out.print(m.group() + " ");
    }
}
```

运行那么久，第一次出现这样的问题啊，我之前都没见过。

```
/* Output:
fix rug bag
fix rig rag
*/
```

使用不带参数的 `reset()` 方法，可以将 `Matcher` 对象重新设置到当前字符序列的起始位置。### 正则表达式与 Java I/O 到目前为止，我们看到的例子都是将正则表达式用于静态的字符串。下面的例子将向你演示，如何应用正则表达式在一个文件中进行搜索匹配操作。`JGrep.java` 的灵感源自于 Unix 上的 *grep*。它有两个参数：文件名以及要匹配的正则表达式。输出的是每行有匹配的部分以及匹配部分在行中的位置。

```
// strings/JGrep.java
// A very simple version of the "grep" program
// {java JGrep
// WhitherStringBuilder.java 'return|for|String'}
import java.util.regex.*;
import java.nio.file.*;
import java.util.stream.*;

public class JGrep {
    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println(
              "Usage: java JGrep file regex");
            System.exit(0);
        }
        Pattern p = Pattern.compile(args[1]);
        // Iterate through the lines of the input file:
        int index = 0;
        Matcher m = p.matcher("");
        for(String line: Files.readAllLines(Paths.get(args[0]))) {

            m.reset(line);
            while(m.find())
                System.out.println(index++ + ": " +
                  m.group() + ": " + m.start());
        }
    }
}
/* Output:
0: for: 4
1: for: 4
*/
```

`Files.readAllLines()` 返回一个 `List<String>` 对象，这意味着可以用 *for-in* 进行遍历。虽然可以在 **for** 循环内部创建一个新的 `Matcher` 对象，但是，在循环体外创建一个空的 `Matcher` 对象，然后用 `reset()` 方法每次为 `Matcher` 加载一行输入，这种处理会有一定的性能优化。最后用 `find()` 搜索结果。

这里读入的测试参数是 `JGrep.java` 文件，然后搜索以 `[Ssct]` 开头的单词。

如果想要更深入地学习正则表达式，你可以阅读 Jeffrey E. F. Friedl 的《精通正则表达式（第 2 版）》。网络上也有很多正则表达式的介绍，你还可以从 Perl 和 Python 等其他语言的文档中找到有用的信息。

19.7 扫描输入

到目前为止，从文件或标准输入读取数据还是一件相当痛苦的事情。一般的解决办法就是读入一行文本，对其进行分词，然后使用 `Integer`、`Double` 等类的各种解析方法来解析数据：

```java
// strings/SimpleRead.java
import java.io.*;

public class SimpleRead {
    public static BufferedReader input =
      new BufferedReader(new StringReader(
        "Sir Robin of Camelot\n22 1.61803"));
    public static void main(String[] args) {
        try {
            System.out.println("What is your name?");
            String name = input.readLine();
            System.out.println(name);
            System.out.println("How old are you? " +
              "What is your favorite double?");
            System.out.println("(input: <age> <double>)");
            String numbers = input.readLine();
            System.out.println(numbers);
            String[] numArray = numbers.split(" ");
            int age = Integer.parseInt(numArray[0]);
            double favorite = Double.parseDouble(numArray[1]);
            System.out.format("Hi %s.%n", name);
            System.out.format("In 5 years you will be %d.%n", age + 5);
            System.out.format("My favorite double is %f.", favorite / 2);
        } catch(IOException e) {
            System.err.println("I/O exception");
        }
    }
}
/* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22 1.61803
Hi Sir Robin of Camelot.
```

```
In 5 years you will be 27.
My favorite double is 0.809015.
*/
```

input 字段使用的类来自 java.io,流式 I/O 详细介绍了相关内容。StringReader 将 String 转化为可读的流对象,然后用这个对象来构造 BufferedReader 对象,因为我们要使用 BufferedReader 的 readLine() 方法。最终，我们可以使用 input 对象一次读取一行文本，就像从控制台读入标准输入一样。

readLine() 方法将一行输入转为 String 对象。如果每一行数据正好对应一个输入值，那这个方法也还可行。但是，如果两个输入值在同一行中，事情就不好办了，我们必须分解这个行，才能分别解析所需的输入值。在这个例子中，分解的操作发生在创建 numArray时。

终于，Java SE5 新增了 Scanner 类，它可以大大减轻扫描输入的工作负担:

```java
// strings/BetterRead.java
import java.util.*;

public class BetterRead {
  public static void main(String[] args) {
    Scanner stdin = new Scanner(SimpleRead.input);
    System.out.println("What is your name?");
    String name = stdin.nextLine();
    System.out.println(name);
    System.out.println(
      "How old are you? What is your favorite double?");
    System.out.println("(input: <age> <double>)");
    int age = stdin.nextInt();
    double favorite = stdin.nextDouble();
    System.out.println(age);
    System.out.println(favorite);
    System.out.format("Hi %s.%n", name);
    System.out.format("In 5 years you will be %d.%n",
      age + 5);
    System.out.format("My favorite double is %f.",
      favorite / 2);
  }
}
/* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22
1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*/
```

Scanner 的构造器可以接收任意类型的输入对象，包括 File、InputStream、String 或者像此例中的Readable 实现类。Readable 是 Java SE5 中新加入的一个接口，表示“具有 read() 方法的某种东西”。上一个例子中的 BufferedReader 也归于这一类。

有了 Scanner，所有的输入、分词、以及解析的操作都隐藏在不同类型的 next 方法中。普通的 next() 方法返回下一个 String。所有的基本类型（除 **char** 之外）都有对应的 next 方法，包括 BigDecimal 和 BigInteger。所有的 next 方法，只有在找到一个完整的分词之后才会返回。Scanner 还有相应的 hasNext 方法，用以判断下一个输入分词是否是所需的类型，如果是则返回 **true**。

在 BetterRead.java 中没有用 **try** 区块捕获IOException。因为，Scanner 有一个假设，在输入结束时会抛出 IOException，所以 Scanner 会把 IOException 吞掉。不过，通过 ioException() 方法，你可以找到最近发生的异常，因此，你可以在必要时检查它。### Scanner 分隔符默认情况下，Scanner 根据空白字符对输入进行分词，但是你可以用正则表达式指定自己所需的分隔符：

```
// strings/ScannerDelimiter.java
import java.util.*;
public class ScannerDelimiter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner("12, 42, 78, 99, 42");
        scanner.useDelimiter("\\s*,\\s*");
        while(scanner.hasNextInt())
            System.out.println(scanner.nextInt());
    }
}
/* Output:
12
42
78
99
42
*/
```

这个例子使用逗号（包括逗号前后任意的空白字符）作为分隔符，同样的技术也可以用来读取逗号分隔的文件。我们可以用 useDelimiter() 来设置分隔符，同时，还有一个 delimiter() 方法，用来返回当前正在作为分隔符使用的 Pattern 对象。### 用正则表达式扫描除了能够扫描基本类型之外，你还可以使用自定义的正则表达式进行扫描，这在扫描复杂数据时非常有用。下面的例子将扫描一个防火墙日志文件中的威胁数据：

```
// strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;
public class ThreatAnalyzer {
    static String threatData =
      "58.27.82.161@08/10/2015\n" +
      "204.45.234.40@08/11/2015\n" +
      "58.27.82.161@08/11/2015\n" +
```

在我的电脑上是正常的啊。

```
        "58.27.82.161@08/12/2015\n" +
        "58.27.82.161@08/12/2015\n" +
        "[Next log section with different data format]";
    public static void main(String[] args) {
        Scanner scanner = new Scanner(threatData);
        String pattern = "(\\d+[.]\\d+[.]\\d+[.]\\d+)@" +
            "(\\d{2}/\\d{2}/\\d{4})";
        while(scanner.hasNext(pattern)) {
            scanner.next(pattern);
            MatchResult match = scanner.match();
            String ip = match.group(1);
            String date = match.group(2);
            System.out.format(
                "Threat on %s from %s%n", date,ip);
        }
    }
}
/* Output:
Threat on 08/10/2015 from 58.27.82.161
Threat on 08/11/2015 from 204.45.234.40
Threat on 08/11/2015 from 58.27.82.161
Threat on 08/12/2015 from 58.27.82.161
Threat on 08/12/2015 from 58.27.82.161
*/
```

当 `next()` 方法配合指定的正则表达式使用时，将找到下一个匹配该模式的输入部分，调用 `match()` 方法就可以获得匹配的结果。如上所示，它的工作方式与之前看到的正则表达式匹配相似。

在配合正则表达式使用扫描时，有一点需要注意：它仅仅针对下一个输入分词进行匹配，如果你的正则表达式中含有分隔符，那永远不可能匹配成功。

19.8 StringTokenizer 类

在 Java 引入正则表达式（J2SE1.4）和 `Scanner` 类（Java SE5）之前，分割字符串的唯一方法是使用 `StringTokenizer` 来分词。不过，现在有了正则表达式和 `Scanner`，我们可以使用更加简单、更加简洁的方式来完成同样的工作了。下面的例子中，我们将 `StringTokenizer` 与另外两种技术简单地做了一个比较：

```
// strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
    public static void main(String[] args) {
        String input =
          "But I'm not dead yet! I feel happy!";
        StringTokenizer stoke = new StringTokenizer(input);
        while(stoke.hasMoreElements())
```

```
            System.out.print(stoke.nextToken() + " ");
        System.out.println();
        System.out.println(Arrays.toString(input.split(" ")));
        Scanner scanner = new Scanner(input);
        while(scanner.hasNext())
            System.out.print(scanner.next() + " ");
    }
}
/* Output:
But I'm not dead yet! I feel happy!
[But, I'm, not, dead, yet!, I, feel, happy!]
But I'm not dead yet! I feel happy!
*/
```

使用正则表达式或 Scanner 对象，我们能够以更加复杂的模式来分割一个字符串，而这对于 StringTokenizer 来说就很困难了。基本上，我们可以放心地说，StringTokenizer 已经可以废弃不用了。

19.9 本章小结

过去，Java 对于字符串操作的技术相当不完善。不过随着近几个版本的升级，我们可以看到，Java 已经从其他语言中吸取了许多成熟的经验。到目前为止，它对字符串操作的支持已经很完善了。不过，有时你还要在细节上注意效率问题，例如恰当地使用 StringBuilder 等。

20 类型信息

> RTTI（RunTime Type Information，运行时类型信息）能够在程序运行时发现和使用类型信息

RTTI 把我们从只能在编译期进行面向类型操作的禁锢中解脱了出来，并且让我们可以使用某些非常强大的程序。对 RTTI 的需要，揭示了面向对象设计中许多有趣（并且复杂）的特性，同时也带来了关于如何组织程序的基本问题。

本章将讨论 Java 是如何在运行时识别对象和类信息的。主要有两种方式：

1. “传统的” RTTI：假定我们在编译时已经知道了所有的类型；
2. “反射”机制：允许我们在运行时发现和使用类的信息。

20.1 为什么需要 RTTI

下面看一下我们已经很熟悉的一个例子，它使用了多态的类层次结构。基类 `Shape` 是泛化的类型，从它派生出了三个具体类：`Circle` 、`Square` 和 `Triangle`（见下图所示）。

Figure 20.1: 多态例子 Shape 的类层次结构图

这是一个典型的类层次结构图，基类位于顶部，派生类向下扩展。面向对象编程的一个基本目的是：让代码只操纵对基类 (这里即 `Shape`) 的引用。这样，如果你想添加一个新类 (比如从 `Shape` 派生出

这不可能，肯定是用户问头，或者编译器出错了。

Rhomboid) 来扩展程序，就不会影响原来的代码。在这个例子中，Shape 接口中动态绑定了 draw() 方法，这样做的目的就是让客户端程序员可以使用泛化的 Shape 引用来调用 draw()。draw() 方法在所有派生类里都会被覆盖，而且由于它是动态绑定的，所以即使通过 Shape 引用来调用它，也能产生恰当的行为，这就是多态。

因此，我们通常会创建一个具体的对象 (Circle、Square 或者 Triangle)，把它向上转型成 Shape (忽略对象的具体类型)，并且在后面的程序中使用 Shape 引用来调用在具体对象中被重载的方法（如 draw()）。

代码如下：

```java
// typeinfo/Shapes.java
import java.util.stream.*;

abstract class Shape {
    void draw() { System.out.println(this + ".draw()"); }
    @Override
    public abstract String toString();
}

class Circle extends Shape {
    @Override
    public String toString() { return "Circle"; }
}

class Square extends Shape {
    @Override
    public String toString() { return "Square"; }
}

class Triangle extends Shape {
    @Override
    public String toString() { return "Triangle"; }
}

public class Shapes {
    public static void main(String[] args) {
        Stream.of(
            new Circle(), new Square(), new Triangle())
            .forEach(Shape::draw);
    }
}
```

输出结果：

```
Circle.draw()
Square.draw()
Triangle.draw()
```

基类中包含 draw() 方法，它通过传递 **this** 参数传递给 System.out.println()，间接地使

这不是 bug，这只不过是配置问题或者网络问题。

用 toString() 打印类标识符 (注意：这里将 toString() 声明为 **abstract**，以此强制继承者覆盖该方法，并防止对 Shape 的实例化)。如果某个对象出现在字符串表达式中 (涉及“+”和字符串对象的表达式)，toString() 方法就会被自动调用，以生成表示该对象的 String。每个派生类都要覆盖（从 Object 继承来的）toString() 方法，这样 draw() 在不同情况下就打印出不同的消息 (多态)。

这个例子中，在把 Shape 对象放入 Stream<Shape> 中时就会进行向上转型 (隐式)，但在向上转型的时候也丢失了这些对象的具体类型。对 stream 而言，它们只是 Shape 对象。

严格来说，Stream<Shape> 实际上是把放入其中的所有对象都当做 Object 对象来持有，只是取元素时会自动将其类型转为 Shape。这也是 RTTI 最基本的使用形式，因为在 Java 中，所有类型转换的正确性检查都是在运行时进行的。这也正是 RTTI 的含义所在：在运行时，识别一个对象的类型。

另外在这个例子中，类型转换并不彻底：Object 被转型为 Shape ，而不是 Circle、Square 或者 Triangle。这是因为目前我们只能确保这个 Stream<Shape> 保存的都是 Shape：

- 编译期，stream 和 Java 泛型系统确保放入 stream 的都是 Shape 对象（Shape 子类的对象也可视为 Shape 的对象），否则编译器会报错；
- 运行时，自动类型转换确保了从 stream 中取出的对象都是 Shape 类型。

接下来就是多态机制的事了，Shape 对象实际执行什么样的代码，是由引用所指向的具体对象（Circle、Square 或者 Triangle）决定的。这也符合我们编写代码的一般需求，通常，我们希望大部分代码尽可能少了解对象的具体类型，而是只与对象家族中的一个通用表示打交道（本例中即为 Shape）。这样，代码会更容易写，更易读和维护；设计也更容易实现，更易于理解和修改。所以多态是面向对象的基本目标。

但是，有时你会碰到一些编程问题，在这些问题中如果你能知道某个泛化引用的具体类型，就可以把问题轻松解决。例如，假设我们允许用户将某些几何形状高亮显示，现在希望找到屏幕上所有高亮显示的三角形；或者，我们现在需要旋转所有图形，但是想跳过圆形 (因为圆形旋转没有意义)。这时我们就希望知道 Stream<Shape> 里边的形状具体是什么类型，而 Java 实际上也满足了我们的这种需求。使用 RTTI，我们可以查询某个 Shape 引用所指向对象的确切类型，然后选择或者剔除特例。

20.2 Class 对象

要理解 RTTI 在 Java 中的工作原理，首先必须知道类型信息在运行时是如何表示的。这项工作是由称为 **Class对象**的特殊对象完成的，它包含了与类有关的信息。实际上，Class 对象就是用来创建该类所有“常规”对象的。Java 使用 Class 对象来实现 RTTI，即便是类型转换这样的操作都是用 Class 对象实现的。不仅如此，Class 类还提供了很多使用 RTTI 的其它方式。

类是程序的一部分，每个类都有一个 Class 对象。换言之，每当我们编写并且编译了一个新类，就会产生一个 Class 对象（更恰当的说，是被保存在一个同名的 **.class** 文件中）。为了生成这个类的对象，Java 虚拟机 (JVM) 先会调用“类加载器”子系统把这个类加载到内存中。

类加载器子系统可能包含一条类加载器链，但有且只有一个**原生类加载器**，它是 JVM 实现的一部

这个功能下个版本做。

分。原生类加载器加载的是”可信类”（包括 Java API 类）。它们通常是从本地盘加载的。在这条链中，通常不需要添加额外的类加载器，但是如果你有特殊需求（例如以某种特殊的方式加载类，以支持 Web 服务器应用，或者通过网络下载类），也可以挂载额外的类加载器。

所有的类都是第一次使用时动态加载到 JVM 中的，当程序创建第一个对类的静态成员的引用时，就会加载这个类。

> 其实构造器也是类的静态方法，虽然构造器前面并没有 **static** 关键字。所以，使用 **new** 操作符创建类的新对象，这个操作也算作对类的静态成员引用。

因此，Java 程序在它开始运行之前并没有被完全加载，很多部分是在需要时才会加载。这一点与许多传统编程语言不同，动态加载使得 Java 具有一些静态加载语言（如 C++）很难或者根本不可能实现的特性。

类加载器首先会检查这个类的 `Class` 对象是否已经加载，如果尚未加载，默认的类加载器就会根据类名查找 **.class** 文件（如果有附加的类加载器，这时候可能就会在数据库中或者通过其它方式获得字节码）。这个类的字节码被加载后，JVM 会对其进行验证，确保它没有损坏，并且不包含不良的 Java 代码 (这是 Java 安全防范的一种措施)。

一旦某个类的 `Class` 对象被载入内存，它就可以用来创建这个类的所有对象。下面的示范程序可以证明这点：

```
// typeinfo/SweetShop.java
// 检查类加载器工作方式
class Cookie {
    static { System.out.println("Loading Cookie"); }
}

class Gum {
    static { System.out.println("Loading Gum"); }
}

class Candy {
    static { System.out.println("Loading Candy"); }
}

public class SweetShop {
    public static void main(String[] args) {
        System.out.println("inside main");
        new Candy();
        System.out.println("After creating Candy");
        try {
            Class.forName("Gum");
        } catch(ClassNotFoundException e) {
            System.out.println("Couldn't find Gum");
        }
        System.out.println("After Class.forName(\"Gum\")");
        new Cookie();
```

这个技术上实现不了。

```
        System.out.println("After creating Cookie");
    }
}
```

输出结果:

```
inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
```

上面的代码中，Candy、Gum 和 Cookie 这几个类都有一个 **static**{...} 静态初始化块，这些静态初始化块在类第一次被加载的时候就会执行。也就是说，静态初始化块会打印出相应的信息，告诉我们这些类分别是什么时候被加载了。而在主方法里边，创建对象的代码都放在了 print() 语句之间，以帮助我们判断类加载的时间点。

从输出中可以看到，Class 对象仅在需要的时候才会被加载，**static** 初始化是在类加载时进行的。

代码里面还有特别有趣的一行:

```
Class.forName("Gum");
```

所有 Class 对象都属于 Class 类，而且它跟其他普通对象一样，我们可以获取和操控它的引用(这也是类加载器的工作)。forName() 是 Class 类的一个静态方法，我们可以使用 forName() 根据目标类的类名（String）得到该类的 Class 对象。上面的代码忽略了 forName() 的返回值，因为那个调用是为了得到它产生的“副作用”。从结果可以看出，forName() 执行的副作用是如果 Gum 类没有被加载就加载它，而在加载的过程中，Gum 的 **static** 初始化块被执行了。

还需要注意的是，如果 Class.forName() 找不到要加载的类，它就会抛出异常 ClassNotFoundException。上面的例子中我们只是简单地报告了问题，但在更严密的程序里，就要考虑在异常处理程序中把问题解决掉（具体例子详见设计模式章节）。

无论何时，只要你想在运行时使用类型信息，就必须先得到那个 Class 对象的引用。Class.forName() 就是实现这个功能的一个便捷途径，因为使用该方法你不需要先持有这个类型的对象。但是，如果你已经拥有了目标类的对象，那就可以通过调用 getClass() 方法来获取 Class 引用了，这个方法来自根类 Object，它将返回表示该对象实际类型的 Class 对象的引用。Class 包含很多有用的方法，下面代码展示了其中的一部分:

```
// typeinfo/toys/ToyTest.java
// 测试 Class 类
// {java typeinfo.toys.ToyTest}
package typeinfo.toys;

```

```java
interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
    // 注释下面的无参数构造器会引起 NoSuchMethodError 错误
    Toy() {}
    Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
    FancyToy() { super(1); }
}

public class ToyTest {
    static void printInfo(Class cc) {
        System.out.println("Class name: " + cc.getName() +
            " is interface? [" + cc.isInterface() + "]");
        System.out.println(
            "Simple name: " + cc.getSimpleName());
        System.out.println(
            "Canonical name : " + cc.getCanonicalName());
    }

    public static void main(String[] args) {
        Class c = null;
        try {
            c = Class.forName("typeinfo.toys.FancyToy");
        } catch(ClassNotFoundException e) {
            System.out.println("Can't find FancyToy");
            System.exit(1);
        }

        printInfo(c);
        for(Class face : c.getInterfaces())
            printInfo(face);

        Class up = c.getSuperclass();
        Object obj = null;

        try {
            // Requires no-arg constructor:
            obj = up.newInstance();
        } catch(InstantiationException e) {
            System.out.println("Cannot instantiate");
            System.exit(1);
        } catch(IllegalAccessException e) {
            System.out.println("Cannot access");
            System.exit(1);
        }
```

```

        printInfo(obj.getClass());
    }
}
```

输出结果:

```
Class name: typeinfo.toys.FancyToy is interface?
[false]
Simple name: FancyToy
Canonical name : typeinfo.toys.FancyToy
Class name: typeinfo.toys.HasBatteries is interface?
[true]
Simple name: HasBatteries
Canonical name : typeinfo.toys.HasBatteries
Class name: typeinfo.toys.Waterproof is interface?
[true]
Simple name: Waterproof
Canonical name : typeinfo.toys.Waterproof
Class name: typeinfo.toys.Shoots is interface? [true]
Simple name: Shoots
Canonical name : typeinfo.toys.Shoots
Class name: typeinfo.toys.Toy is interface? [false]
Simple name: Toy
Canonical name : typeinfo.toys.Toy
```

`FancyToy` 继承自 `Toy` 并实现了 `HasBatteries`、`Waterproof` 和 `Shoots` 接口。在 `main` 方法中，我们创建了一个 `Class` 引用，然后在 **try** 语句里边用 `forName()` 方法创建了一个 `FancyToy` 的类对象并赋值给该引用。需要注意的是，传递给 `forName()` 的字符串必须使用类的全限定名（包含包名）。

`printInfo()` 函数使用 `getName()` 来产生完整类名，使用 `getSimpleName()` 产生不带包名的类名，`getCanonicalName()` 也是产生完整类名（除内部类和数组外，对大部分类产生的结果与 `getName()` 相同）。`isInterface()` 用于判断某个 `Class` 对象代表的是否为一个接口。因此，通过 `Class` 对象，你可以得到关于该类型的所有信息。

在主方法中调用的 `Class.getInterface()` 方法返回的是存放 `Class` 对象的数组，里面的 `Class` 对象表示的是那个类实现的接口。

另外，你还可以调用 `getSuperclass()` 方法来得到父类的 `Class` 对象，再用父类的 `Class` 对象调用该方法，重复多次，你就可以得到一个对象完整的类继承结构。

`Class` 对象的 `newInstance()` 方法是实现“虚拟构造器”的一种途径，虚拟构造器可以让你在不知道一个类的确切类型的时候，创建这个类的对象。在前面的例子中，`up` 只是一个 `Class` 对象的引用，在编译期并不知道这个引用会指向哪个类的 `Class` 对象。当你创建新实例时，会得到一个 `Object` 引用，但是这个引用指向的是 `Toy` 对象。当然，由于得到的是 `Object` 引用，目前你只能给它发送 `Object` 对象能够接受的调用。而如果你想请求具体对象才有的调用，你就得先获取该对象更多的类型

信息，并执行某种转型。另外，使用 `newInstance()` 来创建的类，必须带有无参数的构造器。在本章稍后部分，你将会看到如何通过 Java 的反射 API，用任意的构造器来动态地创建类的对象。

20.2.1 类字面常量

Java 还提供了另一种方法来生成类对象的引用：**类字面常量**。对上述程序来说，就像这样：`FancyToy.class;`。这样做不仅更简单，而且更安全，因为它在编译时就会受到检查（因此不必放在 `try` 语句块中）。并且它根除了对 `forName()` 方法的调用，所以效率更高。

类字面常量不仅可以应用于普通类，也可以应用于接口、数组以及基本数据类型。另外，对于基本数据类型的包装类，还有一个标准字段 `TYPE`。`TYPE` 字段是一个引用，指向对应的基本数据类型的 `Class` 对象，如下所示：

…等价于…
boolean.class
Boolean.TYPE
char.class
Character.TYPE
byte.class
Byte.TYPE
short.class
Short.TYPE
int.class
Integer.TYPE
long.class
Long.TYPE
float.class
Float.TYPE
double.class
Double.TYPE
void.class
Void.TYPE

我的建议是使用 `.class` 的形式，以保持与普通类的一致性。

注意，有一点很有趣：当使用 `.class` 来创建对 `Class` 对象的引用时，不会自动地初始化该 `Class` 对象。为了使用类而做的准备工作实际包含三个步骤：

1. **加载**，这是由类加载器执行的。该步骤将查找字节码（通常在 classpath 所指定的路径中查找，但这并非是必须的），并从这些字节码中创建一个 `Class` 对象。

2. **链接**。在链接阶段将验证类中的字节码，为 **static** 字段分配存储空间，并且如果需要的话，将解析这个类创建的对其他类的所有引用。
3. **初始化**。如果该类具有超类，则先初始化超类，执行 **static** 初始化器和 **static** 初始化块。

直到第一次引用一个 **static** 方法（构造器隐式地是 **static**）或者非常量的 **static** 字段，才会进行类初始化。

```java
// typeinfo/ClassInitialization.java
import java.util.*;

class Initable {
    static final int STATIC_FINAL = 47;
    static final int STATIC_FINAL2 =
        ClassInitialization.rand.nextInt(1000);
    static {
        System.out.println("Initializing Initable");
    }
}

class Initable2 {
    static int staticNonFinal = 147;
    static {
        System.out.println("Initializing Initable2");
    }
}

class Initable3 {
    static int staticNonFinal = 74;
    static {
        System.out.println("Initializing Initable3");
    }
}

public class ClassInitialization {
    public static Random rand = new Random(47);
    public static void
    main(String[] args) throws Exception {
        Class initable = Initable.class;
        System.out.println("After creating Initable ref");
        // Does not trigger initialization:
        System.out.println(Initable.STATIC_FINAL);
        // Does trigger initialization:
        System.out.println(Initable.STATIC_FINAL2);
        // Does trigger initialization:
        System.out.println(Initable2.staticNonFinal);
        Class initable3 = Class.forName("Initable3");
        System.out.println("After creating Initable3 ref");
        System.out.println(Initable3.staticNonFinal);
    }
}
```

输出结果:

```
After creating Initable ref
47
Initializing Initable
258
Initializing Initable2
147
Initializing Initable3
After creating Initable3 ref
74
```

初始化有效地实现了尽可能的“惰性”，从对 `initable` 引用的创建中可以看到，仅使用 **`.class`** 语法来获得对类对象的引用不会引发初始化。但与此相反，使用 `Class.forName()` 来产生 `Class` 引用会立即就进行初始化，如 `initable3`。

如果一个 **`static final`** 值是“编译期常量”（如 `Initable.staticFinal`），那么这个值不需要对 `Initable` 类进行初始化就可以被读取。但是，如果只是将一个字段设置成为 **`static`** 和 **`final`**，还不足以确保这种行为。例如，对 `Initable.staticFinal2` 的访问将强制进行类的初始化，因为它不是一个编译期常量。

如果一个 **`static`** 字段不是 **`final`** 的，那么在对它访问时，总是要求在它被读取之前，要先进行链接（为这个字段分配存储空间）和初始化（初始化该存储空间），就像在对 `Initable2.staticNonFinal` 的访问中所看到的那样。

20.2.2 泛化的 Class 引用

`Class` 引用总是指向某个 `Class` 对象，而 `Class` 对象可以用于产生类的实例，并且包含可作用于这些实例的所有方法代码。它还包含该类的 **`static`** 成员，因此 `Class` 引用表明了它所指向对象的确切类型，而该对象便是 `Class` 类的一个对象。

但是，Java 设计者看准机会，将它的类型变得更具体了一些。Java 引入泛型语法之后，我们可以使用泛型对 `Class` 引用所指向的 `Class` 对象的类型进行限定。在下面的实例中，两种语法都是正确的:

```java
// typeinfo/GenericClassReferences.java

public class GenericClassReferences {
    public static void main(String[] args) {
        Class intClass = int.class;
        Class<Integer> genericIntClass = int.class;
        genericIntClass = Integer.class; // 同一个东西
        intClass = double.class;
        // genericIntClass = double.class; // 非法
    }
}
```

这问题你可以重现吗?

普通的类引用不会产生警告信息。你可以看到，普通的类引用可以重新赋值指向任何其他的 Class 对象，但是使用泛型限定的类引用只能指向其声明的类型。通过使用泛型语法，我们可以让编译器强制执行额外的类型检查。

那如果我们希望稍微放松一些限制，应该怎么办呢？乍一看，下面的操作好像是可以的：

```
Class<Number> geenericNumberClass = int.class;
```

这看起来似乎是起作用的，因为 Integer 继承自 Number。但事实却是不行，因为 Integer 的 Class 对象并不是 Number的 Class 对象的子类（这看起来可能有点诡异，我们将在泛型这一章详细讨论）。

为了在使用 Class 引用时放松限制，我们使用了通配符，它是 Java 泛型中的一部分。通配符就是 ?，表示“任何事物”。因此，我们可以在上例的普通 Class 引用中添加通配符，并产生相同的结果：

```
// typeinfo/WildcardClassReferences.java

public class WildcardClassReferences {
    public static void main(String[] args) {
        Class<?> intClass = int.class;
        intClass = double.class;
    }
}
```

使用 Class<?> 比单纯使用 Class 要好，虽然它们是等价的，并且单纯使用 Class 不会产生编译器警告信息。使用 Class<?> 的好处是它表示你并非是碰巧或者由于疏忽才使用了一个非具体的类引用，而是特意为之。

为了创建一个限定指向某种类型或其子类的 Class 引用，我们需要将通配符与 **extends** 关键字配合使用，创建一个范围限定。这与仅仅声明 Class<Number> 不同，现在做如下声明：

```
// typeinfo/BoundedClassReferences.java

public class BoundedClassReferences {
    public static void main(String[] args) {
        Class<? extends Number> bounded = int.class;
        bounded = double.class;
        bounded = Number.class;
        // Or anything else derived from Number.
    }
}
```

向 Class 引用添加泛型语法的原因只是为了提供编译期类型检查，因此如果你操作有误，稍后就会发现这点。使用普通的 Class 引用你要确保自己不会犯错，因为一旦你犯了错误，就要等到运行时才能发现它，很不方便。

下面的示例使用了泛型语法，它保存了一个类引用，稍后又用 newInstance() 方法产生类的对象：

```java
// typeinfo/DynamicSupplier.java
import java.util.function.*;
import java.util.stream.*;

class CountedInteger {
    private static long counter;
    private final long id = counter++;
    @Override
    public String toString() { return Long.toString(id); }
}

public class DynamicSupplier<T> implements Supplier<T> {
    private Class<T> type;
    public DynamicSupplier(Class<T> type) {
        this.type = type;
    }
    public T get() {
        try {
            return type.newInstance();
        } catch(InstantiationException |
                        IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
    public static void main(String[] args) {
        Stream.generate(
            new DynamicSupplier<>(CountedInteger.class))
            .skip(10)
            .limit(5)
            .forEach(System.out::println);
    }
}
```

输出结果:

```
10
11
12
13
14
```

注意，这个类必须假设与它一起工作的任何类型都有一个无参构造器，否者运行时会抛出异常。编译期对该程序不会产生任何警告信息。

当你将泛型语法用于 `Class` 对象时，`newInstance()` 将返回该对象的确切类型，而不仅仅只是在 `ToyTest.java` 中看到的基类 `Object`。然而，这在某种程度上有些受限:

```java
// typeinfo/toys/GenericToyTest.java
// 测试 Class 类
// {java typeinfo.toys.GenericToyTest}
```

这些代码是上一个开发者写的，不是我写的。

```
package typeinfo.toys;

public class GenericToyTest {
    public static void
    main(String[] args) throws Exception {
        Class<FancyToy> ftClass = FancyToy.class;
        // Produces exact type:
        FancyToy fancyToy = ftClass.newInstance();
        Class<? super FancyToy> up =
            ftClass.getSuperclass();
        // This won't compile:
        // Class<Toy> up2 = ftClass.getSuperclass();
        // Only produces Object:
        Object obj = up.newInstance();
    }
}
```

如果你手头的是超类，那编译期将只允许你声明超类引用为“某个类，它是 FancyToy 的超类”，就像在表达式 Class<? super FancyToy> 中所看到的那样。而不会接收 Class<Toy> 这样的声明。这看上去显得有些怪，因为 getSuperClass() 方法返回的是基类（不是接口），并且编译器在编译期就知道它是什么类型了（在本例中就是 Toy.class），而不仅仅只是“某个类”。不管怎样，正是由于这种含糊性，up.newInstance 的返回值不是精确类型，而只是 Object。

20.2.3 cast() 方法

Java 中还有用于 Class 引用的转型语法，即 cast() 方法：

```
// typeinfo/ClassCasts.java

class Building {}
class House extends Building {}

public class ClassCasts {
    public static void main(String[] args) {
        Building b = new House();
        Class<House> houseType = House.class;
        House h = houseType.cast(b);
        h = (House)b; // ... 或者这样做.
    }
}
```

cast() 方法接受参数对象，并将其类型转换为 Class 引用的类型。但是，如果观察上面的代码，你就会发现，与实现了相同功能的 main 方法中最后一行相比，这种转型好像做了很多额外的工作。

cast() 在无法使用普通类型转换的情况下会显得非常有用，在你编写泛型代码（你将在泛型这一章学习到）时，如果你保存了 Class 引用，并希望以后通过这个引用来执行转型，你就需要用到 cast()。但事实却是这种情况非常少见，我发现整个 Java 类库中，只有一处使用了 cast()（在

这一定是硬件问题。

com.sun.mirror.util.DeclarationFilter 中)。

Java 类库中另一个没有任何用处的特性就是 Class.asSubclass()，该方法允许你将一个 Class 对象转型为更加具体的类型。

20.3 类型转换检测

直到现在，我们已知的 RTTI 类型包括：

1. 传统的类型转换，如“(Shape)”，由 RTTI 确保转换的正确性，如果执行了一个错误的类型转换，就会抛出一个 ClassCastException 异常。
2. 代表对象类型的 Class 对象. 通过查询 Class 对象可以获取运行时所需的信息.

在 C++ 中，经典的类型转换“(Shape)”并不使用 RTTI。它只是简单地告诉编译器将这个对象作为新的类型对待. 而 Java 会进行类型检查，这种类型转换一般被称作“类型安全的向下转型”。之所以称作“向下转型”，是因为传统上类继承图是这么画的。将 Circle 转换为 Shape 是一次向上转型, 将 Shape 转换为 Circle 是一次向下转型。但是, 因为我们知道 Circle 肯定是一个 Shape，所以编译器允许我们自由地做向上转型的赋值操作，且不需要任何显式的转型操作。当你给编译器一个 Shape 的时候，编译器并不知道它到底是什么类型的 Shape——它可能是 Shape，也可能是 Shape 的子类型，例如 Circle、Square、Triangle 或某种其他的类型。在编译期，编译器只能知道它是 Shape。因此，你需要使用显式地进行类型转换，以告知编译器你想转换的特定类型，否则编译器就不允许你执行向下转型赋值。(编译器将会检查向下转型是否合理，因此它不允许向下转型到实际不是待转型类型的子类类型上)。

RTTI 在 Java 中还有第三种形式，那就是关键字 **instanceof**。它返回一个布尔值，告诉我们对象是不是某个特定类型的实例，可以用提问的方式使用它，就像这个样子：

```
if(x instanceof Dog)
    ((Dog)x).bark();
```

在将 x 的类型转换为 Dog 之前，**if** 语句会先检查 x 是否是 Dog 类型的对象。进行向下转型前，如果没有其他信息可以告诉你这个对象是什么类型，那么使用 **instanceof** 是非常重要的，否则会得到一个 ClassCastException 异常。

一般, 可能想要查找某种类型(比如要找三角形, 并填充为紫色), 这时可以轻松地使用 **instanceof** 来度量所有对象。举个例子，假如你有一个类的继承体系，描述了 Pet（以及它们的主人，在后面一个例子中会用到这个特性)。在这个继承体系中的每个 Individual 都有一个 id 和一个可选的名字。尽管下面的类都继承自 Individual，但是 Individual 类复杂性较高，因此其代码将放在附录：容器中进行解释说明。正如你所看到的，此处并不需要去了解 Individual 的代码——你只需了解你可以创建其具名或不具名的对象，并且每个 Individual 都有一个 id() 方法，如果你没有为 Individual 提供名字，toString() 方法只产生类型名。

这只是凑巧。

下面是继承自 Individual 的类的继承体系:

```
// typeinfo/pets/Person.java
package typeinfo.pets;

public class Person extends Individual {
    public Person(String name) { super(name); }
}
```

```
// typeinfo/pets/Pet.java
package typeinfo.pets;

public class Pet extends Individual {
    public Pet(String name) { super(name); }
    public Pet() { super(); }
}
```

```
// typeinfo/pets/Dog.java
package typeinfo.pets;

public class Dog extends Pet {
    public Dog(String name) { super(name); }
    public Dog() { super(); }
}
```

```
// typeinfo/pets/Mutt.java
package typeinfo.pets;

public class Mutt extends Dog {
    public Mutt(String name) { super(name); }
    public Mutt() { super(); }
}
```

```
// typeinfo/pets/Pug.java
package typeinfo.pets;

public class Pug extends Dog {
    public Pug(String name) { super(name); }
    public Pug() { super(); }
}
```

```
// typeinfo/pets/Cat.java
package typeinfo.pets;

public class Cat extends Pet {
    public Cat(String name) { super(name); }
    public Cat() { super(); }
}
```

这只是暂时的解决方案，在正式版我会修改的。

```java
// typeinfo/pets/EgyptianMau.java
package typeinfo.pets;

public class EgyptianMau extends Cat {
    public EgyptianMau(String name) { super(name); }
    public EgyptianMau() { super(); }
}
```

```java
// typeinfo/pets/Manx.java
package typeinfo.pets;

public class Manx extends Cat {
    public Manx(String name) { super(name); }
    public Manx() { super(); }
}
```

```java
// typeinfo/pets/Cymric.java
package typeinfo.pets;

public class Cymric extends Manx {
    public Cymric(String name) { super(name); }
    public Cymric() { super(); }
}
```

```java
// typeinfo/pets/Rodent.java
package typeinfo.pets;

public class Rodent extends Pet {
    public Rodent(String name) { super(name); }
    public Rodent() { super(); }
}
```

```java
// typeinfo/pets/Rat.java
package typeinfo.pets;

public class Rat extends Rodent {
    public Rat(String name) { super(name); }
    public Rat() { super(); }
}
```

```java
// typeinfo/pets/Mouse.java
package typeinfo.pets;

public class Mouse extends Rodent {
    public Mouse(String name) { super(name); }
    public Mouse() { super(); }
}
```

```java
// typeinfo/pets/Hamster.java
package typeinfo.pets;

public class Hamster extends Rodent {
    public Hamster(String name) { super(name); }
    public Hamster() { super(); }
}
```

我们必须显式地为每一个子类编写无参构造器。因为我们有一个带一个参数的构造器，所以编译器不会自动地为我们加上无参构造器。

接下来，我们需要一个类，它可以随机地创建不同类型的宠物，同时，它还可以创建宠物数组和持有宠物的 `List`。为了使这个类更加普遍适用，我们将其定义为抽象类：

```java
// typeinfo/pets/PetCreator.java
// Creates random sequences of Pets
package typeinfo.pets;
import java.util.*;
import java.util.function.*;

public abstract class PetCreator implements Supplier<Pet> {
    private Random rand = new Random(47);

    // The List of the different types of Pet to create:
    public abstract List<Class<? extends Pet>> types();

    public Pet get() { // Create one random Pet
        int n = rand.nextInt(types().size());
        try {
            return types().get(n).newInstance();
        } catch (InstantiationException |
                IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
}
```

抽象的 `types()` 方法需要子类来实现，以此来获取 `Class` 对象构成的 `List`（这是模板方法设计模式的一种变体）。注意，其中类的类型被定义为“任何从 `Pet` 导出的类型”，因此 `newInstance()` 不需要转型就可以产生 `Pet`。`get()` 随机的选取出一个 `Class` 对象，然后可以通过 `Class.newInstance()` 来生成该类的新实例。

在调用 `newInstance()` 时，可能会出现两种异常。在紧跟 **`try`** 语句块后面的 **`catch`** 子句中可以看到对它们的处理。异常的名字再次成为了一种对错误类型相对比较有用的解释（`IllegalAccessException` 违反了 Java 安全机制，在本例中，表示默认构造器为 **`private`** 的情况）。

当你创建 `PetCreator` 的子类时，你需要为 `get()` 方法提供 `Pet` 类型的 `List`。`types()` 方法

会简单地返回一个静态 List 的引用。下面是使用 forName() 的一个具体实现:

```
// typeinfo/pets/ForNameCreator.java
package typeinfo.pets;
import java.util.*;

public class ForNameCreator extends PetCreator {
    private static List<Class<? extends Pet>> types =
            new ArrayList<>();
    // 需要随机生成的类型名:
    private static String[] typeNames = {
            "typeinfo.pets.Mutt",
            "typeinfo.pets.Pug",
            "typeinfo.pets.EgyptianMau",
            "typeinfo.pets.Manx",
            "typeinfo.pets.Cymric",
            "typeinfo.pets.Rat",
            "typeinfo.pets.Mouse",
            "typeinfo.pets.Hamster"
    };

    @SuppressWarnings("unchecked")
    private static void loader() {
        try {
            for (String name : typeNames)
                types.add(
                        (Class<? extends Pet>) Class.forName(name));
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }

    static {
        loader();
    }

    @Override
    public List<Class<? extends Pet>> types() {
        return types;
    }
}
```

loader() 方法使用 Class.forName() 创建了 Class 对象的 List。这可能会导致 ClassNotFoundException 异常，因为你传入的是一个 String 类型的参数，它不能再编译期间被确认是否合理。由于 Pet 相关的文件在 typeinfo 包里面，所以使用它们的时候需要填写完整的包名。

为了使得 List 装入的是具体的 Class 对象，类型转换是必须的，它会产生一个编译时警告。loader() 方法是分开编写的，然后它被放入到一个静态代码块里，因为 @SuppressWarning 注解不能够直接放置在静态代码块之上。

为了对 Pet 进行计数，我们需要一个能跟踪不同类型的 Pet 的工具。Map 的是这个需求的首选，我们将 Pet 类型名作为键，将保存 Pet 数量的 Integer 作为值。通过这种方式，你就看可以询问："有多少个 Hamster 对象？"我们可以使用 **instanceof** 来对 Pet 进行计数：

```
// typeinfo/PetCount.java
// 使用 instanceof
import typeinfo.pets.*;
import java.util.*;

public class PetCount {
    static class Counter extends HashMap<String, Integer> {
        public void count(String type) {
            Integer quantity = get(type);
            if (quantity == null)
                put(type, 1);
            else
                put(type, quantity + 1);
        }
    }

    public static void
    countPets(PetCreator creator) {
        Counter counter = new Counter();
        for (Pet pet : Pets.array(20)) {
            // List each individual pet:
            System.out.print(
                    pet.getClass().getSimpleName() + " ");
            if (pet instanceof Pet)
                counter.count("Pet");
            if (pet instanceof Dog)
                counter.count("Dog");
            if (pet instanceof Mutt)
                counter.count("Mutt");
            if (pet instanceof Pug)
                counter.count("Pug");
            if (pet instanceof Cat)
                counter.count("Cat");
            if (pet instanceof EgyptianMau)
                counter.count("EgyptianMau");
            if (pet instanceof Manx)
                counter.count("Manx");
            if (pet instanceof Cymric)
                counter.count("Cymric");
            if (pet instanceof Rodent)
                counter.count("Rodent");
            if (pet instanceof Rat)
                counter.count("Rat");
            if (pet instanceof Mouse)
                counter.count("Mouse");
            if (pet instanceof Hamster)
```

```
                counter.count("Hamster");
        }
        // Show the counts:
        System.out.println();
        System.out.println(counter);
    }

    public static void main(String[] args) {
        countPets(new ForNameCreator());
    }
}
```

输出结果:

```
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{EgyptianMau=2, Pug=3, Rat=2, Cymric=5, Mouse=2, Cat=9,
Manx=7, Rodent=5, Mutt=3, Dog=6, Pet=20, Hamster=1}
```

在 `countPets()` 中，一个简短的静态方法 `Pets.array()` 生产出了一个随机动物的集合。每个 `Pet` 都被 **`instanceof`** 检测到并计算了一遍。

`instanceof` 有一个严格的限制：只可以将它与命名类型进行比较，而不能与 `Class` 对象作比较。在前面的例子中，你可能会觉得写出一大堆 **`instanceof`** 表达式很乏味，事实也是如此。但是，也没有办法让 **`instanceof`** 聪明起来，让它能够自动地创建一个 `Class` 对象的数组，然后将目标与这个数组中的对象逐一进行比较（稍后会看到一种替代方案）。其实这并不是那么大的限制，如果你在程序中写了大量的 **`instanceof`**，那就说明你的设计可能存在瑕疵。

20.3.1 使用类字面量

如果我们使用类字面量重新实现 `PetCreator` 类的话，其结果在很多方面都会更清晰:

```
// typeinfo/pets/LiteralPetCreator.java
// 使用类字面量
// {java typeinfo.pets.LiteralPetCreator}
package typeinfo.pets;
import java.util.*;

public class LiteralPetCreator extends PetCreator {
    // try 代码块不再需要
    @SuppressWarnings("unchecked")
    public static final List<Class<? extends Pet>> ALL_TYPES =
            Collections.unmodifiableList(Arrays.asList(
                    Pet.class, Dog.class, Cat.class, Rodent.class,
                    Mutt.class, Pug.class, EgyptianMau.class,
                    Manx.class, Cymric.class, Rat.class,
                    Mouse.class, Hamster.class));
```

```
    // 用于随机创建的类型:
    private static final List<Class<? extends Pet>> TYPES =
            ALL_TYPES.subList(ALL_TYPES.indexOf(Mutt.class),
                    ALL_TYPES.size());

    @Override
    public List<Class<? extends Pet>> types() {
        return TYPES;
    }

    public static void main(String[] args) {
        System.out.println(TYPES);
    }
}
```

输出结果:

```
[class typeinfo.pets.Mutt, class typeinfo.pets.Pug,
class typeinfo.pets.EgyptianMau, class
typeinfo.pets.Manx, class typeinfo.pets.Cymric, class
typeinfo.pets.Rat, class typeinfo.pets.Mouse, class
typeinfo.pets.Hamster]
```

在即将到来的 `PetCount3.java` 示例中，我们用所有 `Pet` 类型预先加载一个 `Map`（不仅仅是随机生成的），因此 `ALL_TYPES` 类型的列表是必要的。`types` 列表是 `ALL_TYPES` 类型（使用 `List.subList()` 创建）的一部分，它包含精确的宠物类型，因此用于随机生成 `Pet`。

这次，`types` 的创建没有被 **try** 块包围，因为它是在编译时计算的，因此不会引发任何异常，不像 `Class.forName()`。

我们现在在 `typeinfo.pets` 库中有两个 `PetCreator` 的实现。为了提供第二个作为默认实现，我们可以创建一个使用 `LiteralPetCreator` 的 *外观模式*:

```
// typeinfo/pets/Pets.java
// Facade to produce a default PetCreator
package typeinfo.pets;

import java.util.*;
import java.util.stream.*;

public class Pets {
    public static final PetCreator CREATOR = new LiteralPetCreator();

    public static Pet get() {
        return CREATOR.get();
    }

    public static Pet[] array(int size) {
        Pet[] result = new Pet[size];
        for (int i = 0; i < size; i++)
```

```
            result[i] = CREATOR.get();
        return result;
    }

    public static List<Pet> list(int size) {
        List<Pet> result = new ArrayList<>();
        Collections.addAll(result, array(size));
        return result;
    }

    public static Stream<Pet> stream() {
        return Stream.generate(CREATOR);
    }
}
```

这还提供了对 get()、array() 和 list() 的间接调用，以及生成 Stream<Pet> 的新方法。

因为 PetCount.countPets() 采用了 PetCreator 参数，所以我们可以很容易地测试 LiteralPetCreator（通过上面的外观模式）：

```
// typeinfo/PetCount2.java
import typeinfo.pets.*;

public class PetCount2 {
    public static void main(String[] args) {
        PetCount.countPets(Pets.CREATOR);
    }
}
```

输出结果：

```
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{EgyptianMau=2, Pug=3, Rat=2, Cymric=5, Mouse=2, Cat=9,
Manx=7, Rodent=5, Mutt=3, Dog=6, Pet=20, Hamster=1}
```

输出与 PetCount.java 的输出相同。

20.3.2 一个动态 instanceof 函数

Class.isInstance() 方法提供了一种动态测试对象类型的方法。因此，所有这些繁琐的 instanceof 语句都可以从 PetCount.java 中删除：

```
// typeinfo/PetCount3.java
// 使用 isInstance() 方法

import java.util.*;
import java.util.stream.*;
```

```java

import onjava.*;
import typeinfo.pets.*;

public class PetCount3 {
    static class Counter extends
            LinkedHashMap<Class<? extends Pet>, Integer> {
        Counter() {
            super(LiteralPetCreator.ALL_TYPES.stream()
                    .map(lpc -> Pair.make(lpc, 0))
                    .collect(
                            Collectors.toMap(Pair::key, Pair::value)));
        }

        public void count(Pet pet) {
            // Class.isInstance() 替换 instanceof:
            entrySet().stream()
                    .filter(pair -> pair.getKey().isInstance(pet))
                    .forEach(pair ->
                            put(pair.getKey(), pair.getValue() + 1));
        }

        @Override
        public String toString() {
            String result = entrySet().stream()
                    .map(pair -> String.format("%s=%s",
                            pair.getKey().getSimpleName(),
                            pair.getValue()))
                    .collect(Collectors.joining(", "));
            return "{" + result + "}";
        }
    }

    public static void main(String[] args) {
        Counter petCount = new Counter();
        Pets.stream()
                .limit(20)
                .peek(petCount::count)
                .forEach(p -> System.out.print(
                        p.getClass().getSimpleName() + " "));
        System.out.println("n" + petCount);
    }
}
```

输出结果:

```
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{Rat=2, Pug=3, Mutt=3, Mouse=2, Cat=9, Dog=6, Cymric=5,
EgyptianMau=2, Rodent=5, Hamster=1, Manx=7, Pet=20}
```

为了计算所有不同类型的 Pet，Counter Map 预先加载了来自 LiteralPetCreator.ALL_TYPES 的类型。如果不预先加载 Map，将只计数随机生成的类型，而不是像 Pet 和 Cat 这样的基本类型。

isInstance() 方法消除了对 **instanceof** 表达式的需要。此外，这意味着你可以通过更改 LiteralPetCreator.types 数组来添加新类型的 Pet；程序的其余部分不需要修改（就像使用 **instanceof** 表达式时那样）。

toString() 方法被重载，以便更容易读取输出，该输出仍与打印 Map 时看到的典型输出匹配。

20.3.3 递归计数

PetCount3.Counter 中的 Map 预先加载了所有不同的 Pet 类。我们可以使用 Class.isAssignableFrom() 而不是预加载地图，并创建一个不限于计数 Pet 的通用工具：

```
// onjava/TypeCounter.java
// 计算类型家族的实例数
package onjava;
import java.util.*;
import java.util.stream.*;

public class TypeCounter extends HashMap<Class<?>, Integer> {
    private Class<?> baseType;

    public TypeCounter(Class<?> baseType) {
        this.baseType = baseType;
    }

    public void count(Object obj) {
        Class<?> type = obj.getClass();
        if(!baseType.isAssignableFrom(type))
              throw new RuntimeException(
                obj + " incorrect type: " + type +
                ", should be type or subtype of " + baseType);
        countClass(type);
    }

    private void countClass(Class<?> type) {
        Integer quantity = get(type);
        put(type, quantity == null ? 1 : quantity + 1);
        Class<?> superClass = type.getSuperclass();
        if(superClass != null &&
               baseType.isAssignableFrom(superClass))
              countClass(superClass);
    }

    @Override
```

之前一直都没有出现过这种情况啊。

```
    public String toString() {
        String result = entrySet().stream()
              .map(pair -> String.format("%s=%s",
                pair.getKey().getSimpleName(),
                pair.getValue()))
              .collect(Collectors.joining(", "));
        return "{" + result + "}";
    }
}
```

count() 方法获取其参数的 Class，并使用 isAssignableFrom() 进行运行时检查，以验证传递的对象实际上属于感兴趣的层次结构。countClass() 首先计算类的确切类型。然后，如果 baseType 可以从超类赋值，则在超类上递归调用 countClass()。

```
// typeinfo/PetCount4.java
import typeinfo.pets.*;
import onjava.*;

public class PetCount4 {
    public static void main(String[] args) {
        TypeCounter counter = new TypeCounter(Pet.class);
        Pets.stream()
              .limit(20)
              .peek(counter::count)
              .forEach(p -> System.out.print(
                p.getClass().getSimpleName() + " "));
        System.out.println("n" + counter);
  }
}
```

输出结果：

```
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{Dog=6, Manx=7, Cat=9, Rodent=5, Hamster=1, Rat=2,
Pug=3, Mutt=3, Cymric=5, EgyptianMau=2, Pet=20,
Mouse=2}
```

输出表明两个基类型以及精确类型都被计数了。

20.4 注册工厂

从 Pet 层次结构生成对象的问题是，每当向层次结构中添加一种新类型的 Pet 时，必须记住将其添加到 LiteralPetCreator.java 的条目中。在一个定期添加更多类的系统中，这可能会成为问题。

你可能会考虑向每个子类添加静态初始值设定项，因此初始值设定项会将其类添加到某个列表中。

只改一行代码，不会对整个程序造成影响的，放心。

不幸的是，静态初始值设定项仅在首次加载类时调用，因此存在鸡和蛋的问题：生成器的列表中没有类，因此它无法创建该类的对象，因此类不会被加载并放入列表中。

基本上，你必须自己手工创建列表（除非你编写了一个工具来搜索和分析源代码，然后创建和编译列表）。所以你能做的最好的事情就是把列表集中放在一个明显的地方。层次结构的基类可能是最好的地方。

我们在这里所做的另一个更改是使用*工厂方法*设计模式将对象的创建推迟到类本身。工厂方法可以以多态方式调用，并为你创建适当类型的对象。事实证明，`java.util.function.Supplier` 用 `T get()` 描述了原型工厂方法。协变返回类型允许 `get()` 为 `Supplier` 的每个子类实现返回不同的类型。

在本例中，基类 `Part` 包含一个工厂对象的静态列表，列表成员类型为 `Supplier<Part>`。对于应该由 `get()` 方法生成的类型的工厂，通过将它们添加到 `prototypes` 列表向基类“注册”。奇怪的是，这些工厂本身就是对象的实例。此列表中的每个对象都是用于创建其他对象的*原型*：

```
// typeinfo/RegisteredFactories.java
// 注册工厂到基础类
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

class Part implements Supplier<Part> {
    @Override
    public String toString() {
        return getClass().getSimpleName();
    }

    static List<Supplier<? extends Part>> prototypes =
        Arrays.asList(
          new FuelFilter(),
          new AirFilter(),
          new CabinAirFilter(),
          new OilFilter(),
          new FanBelt(),
          new PowerSteeringBelt(),
          new GeneratorBelt()
        );

    private static Random rand = new Random(47);
    public Part get() {
        int n = rand.nextint(prototypes.size());
        return prototypes.get(n).get();
    }
}

class Filter extends Part {}

class FuelFilter extends Filter {
```

```
    @Override
    public FuelFilter get() {
        return new FuelFilter();
    }
}

class AirFilter extends Filter {
    @Override
    public AirFilter get() {
        return new AirFilter();
    }
}

class CabinAirFilter extends Filter {
    @Override
    public CabinAirFilter get() {
        return new CabinAirFilter();
    }
}

class OilFilter extends Filter {
    @Override
    public OilFilter get() {
        return new OilFilter();
    }
}

class Belt extends Part {}

class FanBelt extends Belt {
    @Override
    public FanBelt get() {
        return new FanBelt();
    }
}

class GeneratorBelt extends Belt {
    @Override
    public GeneratorBelt get() {
        return new GeneratorBelt();
    }
}

class PowerSteeringBelt extends Belt {
    @Override
    public PowerSteeringBelt get() {
        return new PowerSteeringBelt();
    }
}

public class RegisteredFactories {
```

```
    public static void main(String[] args) {
        Stream.generate(new Part())
              .limit(10)
              .forEach(System.out::println);
    }
}
```

输出结果:

```
GeneratorBelt
CabinAirFilter
GeneratorBelt
AirFilter
PowerSteeringBelt
CabinAirFilter
FuelFilter
PowerSteeringBelt
PowerSteeringBelt
FuelFilter
```

并非层次结构中的所有类都应实例化；这里的 `Filter` 和 `Belt` 只是分类器，这样你就不会创建任何一个类的实例，而是只创建它们的子类（请注意，如果尝试这样做，你将获得 `Part` 基类的行为）。

因为 `Part implements Supplier<Part>`，`Part` 通过其 `get()` 方法供应其他 `Part`。如果为基类 `Part` 调用 `get()`（或者如果 `generate()` 调用 `get()`），它将创建随机特定的 `Part` 子类型，每个子类型最终都从 `Part` 继承，并重写相应的 `get()` 以生成它们中的一个。

20.5 类的等价比较

当你查询类型信息时，需要注意：instanceof 的形式 (即 **`instanceof`** 或 `isInstance()`，这两者产生的结果相同) 和与 Class 对象直接比较这两者间存在重要区别。下面的例子展示了这种区别:

```
// typeinfo/FamilyVsExactType.java
// instanceof 与 class 的差别
// {java typeinfo.FamilyVsExactType}
package typeinfo;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
    static void test(Object x) {
        System.out.println(
              "Testing x of type " + x.getClass());
        System.out.println(
              "x instanceof Base " + (x instanceof Base));
        System.out.println(
```

最初的 90% 的代码用去了最初 90% 的开发时间。余下的 10% 的代码用掉另外 90% 的开发时间。

```
                "x instanceof Derived " + (x instanceof Derived));
        System.out.println(
                "Base.isInstance(x) " + Base.class.isInstance(x));
        System.out.println(
                "Derived.isInstance(x) " +
                Derived.class.isInstance(x));
        System.out.println(
                "x.getClass() == Base.class " +
                (x.getClass() == Base.class));
        System.out.println(
                "x.getClass() == Derived.class " +
                (x.getClass() == Derived.class));
        System.out.println(
                "x.getClass().equals(Base.class)) "+
                (x.getClass().equals(Base.class)));
        System.out.println(
                "x.getClass().equals(Derived.class)) " +
                (x.getClass().equals(Derived.class)));
    }

    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
    }
}
```

输出结果:

```
Testing x of type class typeinfo.Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class)) true
x.getClass().equals(Derived.class)) false
Testing x of type class typeinfo.Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derived.class)) true
```

`test()` 方法使用两种形式的 **`instanceof`** 对其参数执行类型检查。然后，它获取 `Class` 引用，并使用 `==` 和 `equals()` 测试 `Class` 对象的相等性。令人放心的是，**`instanceof`** 和 `isInstance()` 产生的结果相同，`equals()` 和 `==` 产生的结果也相同。但测试本身得出了不同的结论。与类型的概念一

致，**instanceof** 说的是“你是这个类，还是从这个类派生的类？”。而如果使用 == 比较实际的`Class`对象，则与继承无关 —— 它要么是确切的类型，要么不是。

20.6 反射：运行时类信息

如果你不知道对象的确切类型，RTTI 会告诉你。但是，有一个限制：必须在编译时知道类型，才能使用 RTTI 检测它，并对信息做一些有用的事情。换句话说，编译器必须知道你使用的所有类。

起初，这看起来并没有那么大的限制，但是假设你引用了一个对不在程序空间中的对象。实际上，该对象的类在编译时甚至对程序都不可用。也许你从磁盘文件或网络连接中获得了大量的字节，并被告知这些字节代表一个类。由于这个类在编译器为你的程序生成代码后很长时间才会出现，你如何使用这样的类?

在传统编程环境中，这是一个牵强的场景。但是，当我们进入一个更大的编程世界时，会有一些重要的情况发生。第一个是基于组件的编程，你可以在应用程序构建器*集成开发环境*中使用*快速应用程序开发*（RAD）构建项目。这是一种通过将表示组件的图标移动到窗体上来创建程序的可视化方法。然后，通过在编程时设置这些组件的一些值来配置这些组件。这种设计时配置要求任何组件都是可实例化的，它公开自己的部分，并且允许读取和修改其属性。此外，处理*图形用户界面*（GUI）事件的组件必须公开有关适当方法的信息，以便 IDE 可以帮助程序员覆写这些事件处理方法。反射提供了检测可用方法并生成方法名称的机制。

在运行时发现类信息的另一个令人信服的动机是提供跨网络在远程平台上创建和执行对象的能力。这称为*远程方法调用*（RMI），它使 Java 程序的对象分布在许多机器上。这种分布有多种原因。如果你想加速一个计算密集型的任务，你可以把它分解成小块放到空闲的机器上。或者你可以将处理特定类型任务的代码（例如，多层次客户机/服务器体系结构中的“业务规则”）放在特定的机器上，这样机器就成为描述这些操作的公共存储库，并且可以很容易地更改它以影响系统中的每个人。分布式计算还支持专门的硬件，这些硬件可能擅长于某个特定的任务——例如矩阵转换——但对于通用编程来说不合适或过于昂贵。

类 `Class` 支持*反射*的概念，`java.lang.reflect` 库中包含类 `Field`、`Method` 和 `Constructor`（每一个都实现了 `Member` 接口）。这些类型的对象由 JVM 在运行时创建，以表示未知类中的对应成员。然后，可以使用 `Constructor` 创建新对象，`get()` 和 `set()` 方法读取和修改与 `Field` 对象关联的字段，`invoke()` 方法调用与 `Method` 对象关联的方法。此外，还可以调用便利方法 `getFields()`、`getMethods()`、`getConstructors()` 等，以返回表示字段、方法和构造函数的对象数组。（你可以通过在 JDK 文档中查找类 `Class` 来了解更多信息。）因此，匿名对象的类信息可以在运行时完全确定，编译时不需要知道任何信息。

重要的是要意识到反射没有什么魔力。当你使用反射与未知类型的对象交互时，JVM 将查看该对象，并看到它属于特定的类（就像普通的 RTTI）。在对其执行任何操作之前，必须加载 `Class` 对象。因此，该特定类型的 **`.class`** 文件必须在本地计算机上或通过网络对 JVM 仍然可用。因此，RTTI 和

昨天还好好的。

反射的真正区别在于，使用 RTTI 时，编译器在编译时会打开并检查 `.class` 文件。换句话说，你可以用“正常”的方式调用一个对象的所有方法。通过反射，`.class` 文件在编译时不可用；它由运行时环境打开并检查。

20.6.1 类方法提取器

通常，你不会直接使用反射工具，但它们可以帮助你创建更多的动态代码。反射是用来支持其他 Java 特性的，例如对象序列化（参见附录：对象序列化）。但是，有时动态提取有关类的信息很有用。

考虑一个类方法提取器。查看类定义的源代码或 JDK 文档，只显示*在该类定义中*定义或重写的方法。但是，可能还有几十个来自基类的可用方法。找到它们既单调又费时[1]。幸运的是，反射提供了一种方法，可以简单地编写一个工具类自动地向你展示所有的接口：

```
// typeinfo/ShowMethods.java
// 使用反射展示一个类的所有方法，甚至包括定义在基类中方法
// {java ShowMethods ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;

public class ShowMethods {
    private static String usage =
            "usage:\n" +
            "ShowMethods qualified.class.name\n" +
            "To show all methods in class or:\n" +
            "ShowMethods qualified.class.name word\n" +
            "To search for methods involving 'word'";
    private static Pattern p = Pattern.compile("\\w+\\.");

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println(usage);
            System.exit(0);
        }
        int lines = 0;
        try {
            Class<?> c = Class.forName(args[0]);
            Method[] methods = c.getMethods();
            Constructor[] ctors = c.getConstructors();
            if (args.length == 1) {
                for (Method method : methods)
                    System.out.println(
                            p.matcher(
                                    method.toString()).replaceAll(""));
                for (Constructor ctor : ctors)
                    System.out.println(
                            p.matcher(ctor.toString()).replaceAll(""));
```

1

```
                lines = methods.length + ctors.length;
            } else {
                for (Method method : methods)
                    if (method.toString().contains(args[1])) {
                        System.out.println(p.matcher(
                                method.toString()).replaceAll(""));
                        lines++;
                    }
                for (Constructor ctor : ctors)
                    if (ctor.toString().contains(args[1])) {
                        System.out.println(p.matcher(
                                ctor.toString()).replaceAll(""));
                        lines++;
                    }
            }
        } catch (ClassNotFoundException e) {
            System.out.println("No such class: " + e);
        }
    }
}
```

输出结果:

```
public static void main(String[])
public final void wait() throws InterruptedException
public final void wait(long,int) throws
InterruptedException
public final native void wait(long) throws
InterruptedException
public boolean equals(Object)
public String toString()
public native int hashCode()
public final native Class getClass()
public final native void notify()
public final native void notifyAll()
public ShowMethods()
```

`Class` 方法 `getmethods()` 和 `getconstructors()` 分别返回 `Method` 数组和 `Constructor` 数组。这些类中的每一个都有进一步的方法来解析它们所表示的方法的名称、参数和返回值。但你也可以像这里所做的那样，使用 `toString()`，生成带有整个方法签名的 `String`。代码的其余部分提取命令行信息，确定特定签名是否与目标 `String`（使用 `indexOf()`）匹配，并使用正则表达式（在Strings一章中介绍）删除名称限定符。

编译时无法知道 `Class.forName()` 生成的结果，因此所有方法签名信息都是在运行时提取的。如果你研究 JDK 反射文档，你将看到有足够的支持来实际设置和对编译时完全未知的对象进行方法调用（本书后面有这样的例子）。虽然最初你可能认为你永远都不需要这样做，但是反射的全部价值可能会令人惊讶。

上面的输出来自命令行:

```
java ShowMethods ShowMethods
```

输出包含一个 **public** 无参数构造函数，即使未定义构造函数。你看到的构造函数是由编译器自动合成的。如果将 ShowMethods 设置为非 **public** 类（即只有包级访问权），则合成的无参数构造函数将不再显示在输出中。自动为合成的无参数构造函数授予与类相同的访问权。

尝试运行 java ShowMethods java.lang.String，并附加一个 **char**、**int**、String 等参数。

编程时，当你不记得某个类是否有特定的方法，并且不想在 JDK 文档中搜索索引或类层次结构时，或者如果你不知道该类是否可以对 Color 对象执行任何操作时，该工具能节省不少时间。

20.7 动态代理

*代理*是基本的设计模式之一。一个对象封装真实对象，代替其提供其他或不同的操作—这些操作通常涉及到与“真实”对象的通信，因此代理通常充当中间对象。这是一个简单的示例，显示代理的结构：

```java
// typeinfo/SimpleProxyDemo.java

interface Interface {
    void doSomething();

    void somethingElse(String arg);
}

class RealObject implements Interface {
    @Override
    public void doSomething() {
        System.out.println("doSomething");
    }

    @Override
    public void somethingElse(String arg) {
        System.out.println("somethingElse " + arg);
    }
}

class SimpleProxy implements Interface {
    private Interface proxied;

    SimpleProxy(Interface proxied) {
        this.proxied = proxied;
    }

    @Override
    public void doSomething() {
        System.out.println("SimpleProxy doSomething");
```

```
        proxied.doSomething();
    }

    @Override
    public void somethingElse(String arg) {
        System.out.println(
                "SimpleProxy somethingElse " + arg);
        proxied.somethingElse(arg);
    }
}

class SimpleProxyDemo {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }

    public static void main(String[] args) {
        consumer(new RealObject());
        consumer(new SimpleProxy(new RealObject()));
    }
}
```

输出结果:

```
doSomething
somethingElse bonobo
SimpleProxy doSomething
doSomething
SimpleProxy somethingElse bonobo
somethingElse bonobo
```

因为 `consumer()` 接受 `Interface`，所以它不知道获得的是 `RealObject` 还是 `SimpleProxy`，因为两者都实现了 `Interface`。但是，在客户端和 `RealObject` 之间插入的 `SimpleProxy` 执行操作，然后在 `RealObject` 上调用相同的方法。

当你希望将额外的操作与“真实对象”做分离时，代理可能会有所帮助，尤其是当你想要轻松地启用额外的操作时，反之亦然（设计模式就是封装变更—所以你必须改变一些东西以证明模式的合理性）。例如，如果你想跟踪对 `RealObject` 中方法的调用，或衡量此类调用的开销，该怎么办？你不想这部分代码耦合到你的程序中，而代理能使你可以很轻松地添加或删除它。

Java 的*动态代理*更进一步，不仅动态创建代理对象而且动态处理对代理方法的调用。在动态代理上进行的所有调用都被重定向到单个*调用处理程序*，该处理程序负责发现调用的内容并决定如何处理。这是 `SimpleProxyDemo.java` 使用动态代理重写的例子:

```
// typeinfo/SimpleDynamicProxy.java

import java.lang.reflect.*;

```

```
class DynamicProxyHandler implements InvocationHandler {
    private Object proxied;

    DynamicProxyHandler(Object proxied) {
        this.proxied = proxied;
    }

    @Override
    public Object
    invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        System.out.println(
                "**** proxy: " + proxy.getClass() +
                        ", method: " + method + ", args: " + args);
        if (args != null)
            for (Object arg : args)
                System.out.println("  " + arg);
        return method.invoke(proxied, args);
    }
}

class SimpleDynamicProxy {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }

    public static void main(String[] args) {
        RealObject real = new RealObject();
        consumer(real);
        // Insert a proxy and call again:
        Interface proxy = (Interface) Proxy.newProxyInstance(
                Interface.class.getClassLoader(),
                new Class[]{Interface.class},
                new DynamicProxyHandler(real));
        consumer(proxy);
    }
}
```

输出结果:

```
doSomething
somethingElse bonobo
**** proxy: class $Proxy0, method: public abstract void
Interface.doSomething(), args: null
doSomething
**** proxy: class $Proxy0, method: public abstract void
Interface.somethingElse(java.lang.String), args:
[Ljava.lang.Object;@6bc7c054
  bonobo
somethingElse bonobo
```

可以通过调用静态方法 Proxy.newProxyInstance() 来创建动态代理，该方法需要一个类加载器（通常可以从已加载的对象中获取），希望代理实现的接口列表（不是类或抽象类），以及接口 InvocationHandler 的一个实现。动态代理会将所有调用重定向到调用处理程序，因此通常为调用处理程序的构造函数提供对“真实”对象的引用，以便一旦执行中介任务便可以转发请求。

invoke() 方法被传递给代理对象，以防万一你必须区分请求的来源—但是在很多情况下都无需关心。但是，在 invoke() 内的代理上调用方法时要小心，因为接口的调用是通过代理重定向的。

通常执行代理操作，然后使用 Method.invoke() 将请求转发给被代理对象，并携带必要的参数。这在一开始看起来是有限制的，好像你只能执行一般的操作。但是，可以过滤某些方法调用，同时传递其他方法调用：

```
// typeinfo/SelectingMethods.java
// Looking for particular methods in a dynamic proxy

import java.lang.reflect.*;

class MethodSelector implements InvocationHandler {
    private Object proxied;

    MethodSelector(Object proxied) {
        this.proxied = proxied;
    }

    @Override
    public Object
    invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        if (method.getName().equals("interesting"))
            System.out.println(
                    "Proxy detected the interesting method");
        return method.invoke(proxied, args);
    }
}

interface SomeMethods {
    void boring1();

    void boring2();

    void interesting(String arg);

    void boring3();
}

class Implementation implements SomeMethods {
    @Override
```

```
    public void boring1() {
        System.out.println("boring1");
    }

    @Override
    public void boring2() {
        System.out.println("boring2");
    }

    @Override
    public void interesting(String arg) {
        System.out.println("interesting " + arg);
    }

    @Override
    public void boring3() {
        System.out.println("boring3");
    }
}

class SelectingMethods {
    public static void main(String[] args) {
        SomeMethods proxy =
                (SomeMethods) Proxy.newProxyInstance(
                        SomeMethods.class.getClassLoader(),
                        new Class[]{ SomeMethods.class },
                        new MethodSelector(new Implementation()));
        proxy.boring1();
        proxy.boring2();
        proxy.interesting("bonobo");
        proxy.boring3();
    }
}
```

输出结果:

```
boring1
boring2
Proxy detected the interesting method
interesting bonobo
boring3
```

在这个示例里，我们只是在寻找方法名，但是你也可以寻找方法签名的其他方面，甚至可以搜索特定的参数值。

动态代理不是你每天都会使用的工具，但是它可以很好地解决某些类型的问题。你可以在 Erich Gamma 等人的*设计模式*中了解有关*代理*和其他设计模式的更多信息。(Addison-Wesley，1995 年)，以及设计模式一章。

其实, 我尝试着使 Ruby 更自然, 而不是简单。Ruby 看起来很简单, 但内部是非常复杂的, 就像我们的身体一样。 —松本行弘,Ruby 之父

20.8 Optional 类

如果你使用内置的 **null** 来表示没有对象，每次使用引用的时候就必须测试一下引用是否为 **null**，这显得有点枯燥，而且势必会产生相当乏味的代码。问题在于 **null** 没什么自己的行为，只会在你想用它执行任何操作的时候产生 NullPointException。java.util.Optional（首次出现是在函数式编程这章）为 **null** 值提供了一个轻量级代理，Optional 对象可以防止你的代码直接抛出 NullPointException。

虽然 Optional 是 Java 8 为了支持流式编程才引入的，但其实它是一个通用的工具。为了证明这点，在本节中，我们会把它用在普通的类中。因为涉及一些运行时检测，所以把这一小节放在了本章。

实际上，在所有地方都使用 Optional 是没有意义的，有时候检查一下是不是 **null** 也挺好的，或者有时我们可以合理地假设不会出现 **null**，甚至有时候检查 NullPointException 异常也是可以接受的。Optional 最有用武之地的是在那些“更接近数据”的地方，在问题空间中代表实体的对象上。举个简单的例子，很多系统中都有 Person 类型，代码中有些情况下你可能没有一个实际的 Person 对象（或者可能有，但是你还没用关于那个人的所有信息）。这时，在传统方法下，你会用到一个 **null** 引用，并且在使用的时候测试它是不是 **null**。而现在，我们可以使用 Optional：

```
// typeinfo/Person.java
// Using Optional with regular classes

import onjava.*;

import java.util.*;

class Person {
    public final Optional<String> first;
    public final Optional<String> last;
    public final Optional<String> address;
    // etc.
    public final Boolean empty;

    Person(String first, String last, String address) {
        this.first = Optional.ofNullable(first);
        this.last = Optional.ofNullable(last);
        this.address = Optional.ofNullable(address);
        empty = !this.first.isPresent()
                && !this.last.isPresent()
                && !this.address.isPresent();
    }

    Person(String first, String last) {
        this(first, last, null);
    }

    Person(String last) {
```

```
        this(null, last, null);
    }

    Person() {
        this(null, null, null);
    }

    @Override
    public String toString() {
        if (empty)
            return "<Empty>";
        return (first.orElse("") +
                " " + last.orElse("") +
                " " + address.orElse("")).trim();
    }

    public static void main(String[] args) {
        System.out.println(new Person());
        System.out.println(new Person("Smith"));
        System.out.println(new Person("Bob", "Smith"));
        System.out.println(new Person("Bob", "Smith",
                "11 Degree Lane, Frostbite Falls, MN"));
    }
}
```

输出结果：

```
<Empty>
Smith
Bob Smith
Bob Smith 11 Degree Lane, Frostbite Falls, MN
```

`Person` 的设计有时候又叫“数据传输对象（DTO，data-transfer object）”。注意，所有字段都是 **`public`** 和 **`final`** 的，所以没有 `getter` 和 `setter` 方法。也就是说，`Person` 是不可变的，你只能通过构造器给它赋值，之后就只能读而不能修改它的值（字符串本身就是不可变的，因此你无法修改字符串的内容，也无法给它的字段重新赋值）。如果你想修改一个 `Person`，你只能用一个新的 `Person` 对象来替换它。`empty` 字段在对象创建的时候被赋值，用于快速判断这个 `Person` 对象是不是空对象。

如果想使用 `Person`，就必须使用 `Optional` 接口才能访问它的 `String` 字段，这样就不会意外触发 `NullPointerException` 了。

现在假设你已经因你惊人的理念而获得了一大笔风险投资，现在你要招兵买马了，但是在虚位以待时，你可以将 `Person` `Optional` 对象放在每个 `Position` 上：

```
// typeinfo/Position.java

import java.util.*;

class EmptyTitleException extends RuntimeException {
```

软件开发往往是这样：最开始的 90% 代码占用了开始的 90% 的开发时间；剩下 10% 代码同样需要 90% 的开发时间。
—Tom Cargill

```
}

class Position {
    private String title;
    private Person person;

    Position(String jobTitle, Person employee) {
        setTitle(jobTitle);
        setPerson(employee);
    }

    Position(String jobTitle) {
        this(jobTitle, null);
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String newTitle) {
        // Throws EmptyTitleException if newTitle is null:
        title = Optional.ofNullable(newTitle)
                .orElseThrow(EmptyTitleException::new);
    }

    public Person getPerson() {
        return person;
    }

    public void setPerson(Person newPerson) {
        // Uses empty Person if newPerson is null:
        person = Optional.ofNullable(newPerson)
                .orElse(new Person());
    }

    @Override
    public String toString() {
        return "Position: " + title +
                ", Employee: " + person;
    }

    public static void main(String[] args) {
        System.out.println(new Position("CEO"));
        System.out.println(new Position("Programmer",
                new Person("Arthur", "Fonzarelli")));
        try {
            new Position(null);
        } catch (Exception e) {
            System.out.println("caught " + e);
        }
    }
```

```
}
```

输出结果:

```
Position: CEO, Employee: <Empty>
Position: Programmer, Employee: Arthur Fonzarelli
caught EmptyTitleException
```

这里使用 Optional 的方式不太一样。请注意，title 和 person 都是普通字段，不受 Optional 的保护。但是，修改这些字段的唯一途径是调用 setTitle() 和 setPerson() 方法，这两个都借助 Optional 对字段进行了严格的限制。

同时，我们想保证 title 字段永远不会变成 **null** 值。为此，我们可以自己在 setTitle() 方法里边检查参数 newTitle 的值。但其实还有更好的做法，函数式编程一大优势就是可以让我们重用经过验证的功能（即便是个很小的功能），以减少自己手动编写代码可能产生的一些小错误。所以在这里，我们用 ofNullable() 把 newTitle 转换一个 Optional（如果传入的值为 **null**，ofNullable() 返回的将是 Optional.empty()）。紧接着我们调用了 orElseThrow() 方法，所以如果 newTitle 的值是 **null**，你将会得到一个异常。这里我们并没有把 title 保存成 Optional，但通过应用 Optional 的功能，我们仍然如愿以偿地对这个字段施加了约束。

EmptyTitleException 是一个 RuntimeException，因为它意味着程序存在错误。在这个方案里边，你仍然可能会得到一个异常。但不同的是，在错误产生的那一刻（向 setTitle() 传 **null** 值时）就会抛出异常，而不是发生在其它时刻，需要你通过调试才能发现问题所在。另外，使用 EmptyTitleException 还有助于定位 BUG。

Person 字段的限制又不太一样：如果你把它的值设为 **null**，程序会自动把将它赋值成一个空的 Person 对象。先前我们也用过类似的方法把字段转换成 Option，但这里我们是在返回结果的时候使用 orElse(**new** Person()) 插入一个空的 Person 对象替代了 **null**。

在 Position 里边，我们没有创建一个表示“空”的标志位或者方法，因为 person 字段的 Person 对象为空，就表示这个 Position 是个空缺位置。之后，你可能会发现你必须添加一个显式的表示“空位”的方法，但是正如 YAGNI[2] (You Aren't Going to Need It，你永远不需要它) 所言，在初稿时“实现尽最大可能的简单”，直到程序在某些方面要求你为其添加一些额外的特性，而不是假设这是必要的。

请注意，虽然你清楚你使用了 Optional，可以免受 NullPointerExceptions 的困扰，但是 Staff 类却对此毫不知情。

```
// typeinfo/Staff.java

import java.util.*;

public class Staff extends ArrayList<Position> {
```

[2]事实证明，质数实际上并不是散列桶的理想容量。近来，（经过广泛的测试）Java 的散列函数都使用 2 的整数次方。对现代的处理器来说，除法与求余数是最慢的操作。使用 2 的整数次方长度的散列表，可用掩码代替除法。

```
    public void add(String title, Person person) {
        add(new Position(title, person));
    }

    public void add(String... titles) {
        for (String title : titles)
            add(new Position(title));
    }

    public Staff(String... titles) {
        add(titles);
    }

    public Boolean positionAvailable(String title) {
        for (Position position : this)
            if (position.getTitle().equals(title) &&
                    position.getPerson().empty)
                return true;
        return false;
    }

    public void fillPosition(String title, Person hire) {
        for (Position position : this)
            if (position.getTitle().equals(title) &&
                    position.getPerson().empty) {
                position.setPerson(hire);
                return;
            }
        throw new RuntimeException(
                "Position " + title + " not available");
    }

    public static void main(String[] args) {
        Staff staff = new Staff("President", "CTO",
                "Marketing Manager", "Product Manager",
                "Project Lead", "Software Engineer",
                "Software Engineer", "Software Engineer",
                "Software Engineer", "Test Engineer",
                "Technical Writer");
        staff.fillPosition("President",
                new Person("Me", "Last", "The Top, Lonely At"));
        staff.fillPosition("Project Lead",
                new Person("Janet", "Planner", "The Burbs"));
        if (staff.positionAvailable("Software Engineer"))
            staff.fillPosition("Software Engineer",
                    new Person(
                            "Bob", "Coder", "Bright Light City"));
        System.out.println(staff);
    }
}
```

输出结果:

```
[Position: President, Employee: Me Last The Top, Lonely
At, Position: CTO, Employee: <Empty>, Position:
Marketing Manager, Employee: <Empty>, Position: Product
Manager, Employee: <Empty>, Position: Project Lead,
Employee: Janet Planner The Burbs, Position: Software
Engineer, Employee: Bob Coder Bright Light City,
Position: Software Engineer, Employee: <Empty>,
Position: Software Engineer, Employee: <Empty>,
Position: Software Engineer, Employee: <Empty>,
Position: Test Engineer, Employee: <Empty>, Position:
Technical Writer, Employee: <Empty>]
```

注意，在有些地方你可能还是要测试引用是不是 `Optional`，这跟检查是否为 **`null`** 没什么不同。但是在其它地方（例如本例中的 `toString()` 转换），你就不必执行额外的测试了，而可以直接假设所有对象都是有效的。

20.8.1 标记接口

有时候使用一个**标记接口**来表示空值会更方便。标记接口里边什么都没有，你只要把它的名字当做标签来用就可以。

```
// onjava/Null.java
package onjava;
public interface Null {}
```

如果你用接口取代具体类，那么就可以使用 `DynamicProxy` 来自动地创建 `Null` 对象。假设我们有一个 `Robot` 接口，它定义了一个名字、一个模型和一个描述 `Robot` 行为能力的 `List<Operation>`:

```
// typeinfo/Robot.java

import onjava.*;

import java.util.*;

public interface Robot {
    String name();

    String model();

    List<Operation> operations();

    static void test(Robot r) {
        if (r instanceof Null)
            System.out.println("[Null Robot]");
        System.out.println("Robot name: " + r.name());
        System.out.println("Robot model: " + r.model());
```

```
        for (Operation operation : r.operations()) {
            System.out.println(operation.description.get());
            operation.command.run();
        }
    }
}
```

你可以通过调用 `operations()` 来访问 `Robot` 的服务。`Robot` 里边还有一个 **`static`** 方法来执行测试。

`Operation` 包含一个描述和一个命令（这用到了**命令模式**）。它们被定义成函数式接口的引用，所以可以把 lambda 表达式或者方法的引用传给 `Operation` 的构造器：

```
// typeinfo/Operation.java

import java.util.function.*;

public class Operation {
    public final Supplier<String> description;
    public final Runnable command;

    public Operation(Supplier<String> descr, Runnable cmd) {
        description = descr;
        command = cmd;
    }
}
```

现在我们可以创建一个扫雪 `Robot`：

```
// typeinfo/SnowRemovalRobot.java

import java.util.*;

public class SnowRemovalRobot implements Robot {
    private String name;

    public SnowRemovalRobot(String name) {
        this.name = name;
    }

    @Override
    public String name() {
        return name;
    }

    @Override
    public String model() {
        return "SnowBot Series 11";
    }

    private List<Operation> ops = Arrays.asList(
```

```
            new Operation(
                    () -> name + " can shovel snow",
                    () -> System.out.println(
                            name + " shoveling snow")),
            new Operation(
                    () -> name + " can chip ice",
                    () -> System.out.println(name + " chipping ice")),
            new Operation(
                    () -> name + " can clear the roof",
                    () -> System.out.println(
                            name + " clearing roof")));

    public List<Operation> operations() {
        return ops;
    }

    public static void main(String[] args) {
        Robot.test(new SnowRemovalRobot("Slusher"));
    }
}
```

输出结果:

```
Robot name: Slusher
Robot model: SnowBot Series 11
Slusher can shovel snow
Slusher shoveling snow
Slusher can chip ice
Slusher chipping ice
Slusher can clear the roof
Slusher clearing roof
```

假设存在许多不同类型的 `Robot`，我们想让每种 `Robot` 都创建一个 `Null` 对象来执行一些特殊的操作——在本例中，即提供 `Null` 对象所代表 `Robot` 的确切类型信息。这些信息是通过动态代理捕获的:

```
// typeinfo/NullRobot.java
// Using a dynamic proxy to create an Optional

import java.lang.reflect.*;
import java.util.*;
import java.util.stream.*;

import onjava.*;

class NullRobotProxyHandler
        implements InvocationHandler {
    private String nullName;
    private Robot proxied = new NRobot();

```

```
    NullRobotProxyHandler(Class<? extends Robot> type) {
        nullName = type.getSimpleName() + " NullRobot";
    }

    private class NRobot implements Null, Robot {
        @Override
        public String name() {
            return nullName;
        }

        @Override
        public String model() {
            return nullName;
        }

        @Override
        public List<Operation> operations() {
            return Collections.emptyList();
        }
    }

    @Override
    public Object
    invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        return method.invoke(proxied, args);
    }
}

public class NullRobot {
    public static Robot
    newNullRobot(Class<? extends Robot> type) {
        return (Robot) Proxy.newProxyInstance(
                NullRobot.class.getClassLoader(),
                new Class[] { Null.class, Robot.class },
                new NullRobotProxyHandler(type));
    }

    public static void main(String[] args) {
        Stream.of(
                new SnowRemovalRobot("SnowBee"),
                newNullRobot(SnowRemovalRobot.class)
        ).forEach(Robot::test);
    }
}
```

输出结果:

```
Robot name: SnowBee
Robot model: SnowBot Series 11
SnowBee can shovel snow
```

```
SnowBee shoveling snow
SnowBee can chip ice
SnowBee chipping ice
SnowBee can clear the roof
SnowBee clearing roof
[Null Robot]
Robot name: SnowRemovalRobot NullRobot
Robot model: SnowRemovalRobot NullRobot
```

无论何时，如果你需要一个空 `Robot` 对象，只需要调用 `newNullRobot()`，并传递需要代理的 `Robot` 的类型。这个代理满足了 `Robot` 和 `Null` 接口的需要，并提供了它所代理的类型的确切名字。

20.8.2 Mock 对象和桩

Mock 对象和 **桩（Stub）**在逻辑上都是 `Optional` 的变体。他们都是最终程序中所使用的“实际”对象的代理。不过，Mock 对象和桩都是假扮成那些可以传递实际信息的实际对象，而不是像 `Optional` 那样把包含潜在 **`null`** 值的对象隐藏。

Mock 对象和桩之间的的差别在于程度不同。Mock 对象往往是轻量级的，且用于自测试。通常，为了处理各种不同的测试场景，我们会创建出很多 Mock 对象。而桩只是返回桩数据，它通常是重量级的，并且经常在多个测试中被复用。桩可以根据它们被调用的方式，通过配置进行修改。因此，桩是一种复杂对象，它可以做很多事情。至于 Mock 对象，如果你要做很多事，通常会创建大量又小又简单的 Mock 对象。

20.9 接口和类型

`interface` 关键字的一个重要目标就是允许程序员隔离组件，进而降低耦合度。使用接口可以实现这一目标，但是通过类型信息，这种耦合性还是会传播出去——接口并不是对解耦的一种无懈可击的保障。比如我们先写一个接口：

```java
// typeinfo/interfacea/A.java
package typeinfo.interfacea;

public interface A {
    void f();
}
```

然后实现这个接口，你可以看到其代码是怎么从实际类型开始顺藤摸瓜的：

```java
// typeinfo/InterfaceViolation.java
// Sneaking around an interface

import typeinfo.interfacea.*;

```

```
class B implements A {
    public void f() {
    }

    public void g() {
    }
}

public class InterfaceViolation {
    public static void main(String[] args) {
        A a = new B();
        a.f();
        // a.g(); // Compile error
        System.out.println(a.getClass().getName());
        if (a instanceof B) {
            B b = (B) a;
            b.g();
        }
    }
}
```

输出结果:

```
B
```

通过使用 RTTI，我们发现 a 是用 B 实现的。通过将其转型为 B，我们可以调用不在 A 中的方法。

这样的操作完全是合情合理的，但是你也许并不想让客户端开发者这么做，因为这给了他们一个机会，使得他们的代码与你的代码的耦合度超过了你的预期。也就是说，你可能认为 **interface** 关键字正在保护你，但其实并没有。另外，在本例中使用 B 来实现 A 这种情况是有公开案例可查的[3]。

一种解决方案是直接声明，如果开发者决定使用实际的类而不是接口，他们需要自己对自己负责。这在很多情况下都是可行的，但“可能”还不够，你或许希望能有一些更严格的控制方式。

最简单的方式是让实现类只具有包访问权限，这样在包外部的客户端就看不到它了:

```
// typeinfo/packageaccess/HiddenC.java
package typeinfo.packageaccess;

import typeinfo.interfacea.*;

class C implements A {
    @Override
    public void f() {
        System.out.println("public C.f()");
    }

    public void g() {
        System.out.println("public C.g()");
```

[3]请注意，在 64 位处理器上可能不会发生这种情况，从而消除了这个问题。

```
    }

    void u() {
        System.out.println("package C.u()");
    }

    protected void v() {
        System.out.println("protected C.v()");
    }

    private void w() {
        System.out.println("private C.w()");
    }
}

public class HiddenC {
    public static A makeA() {
        return new C();
    }
}
```

在这个包中唯一 **public** 的部分就是 HiddenC，在被调用时将产生 A接口类型的对象。这里有趣之处在于：即使你从 makeA() 返回的是 C 类型，你在包的外部仍旧不能使用 A 之外的任何方法，因为你不能在包的外部命名 C。

现在如果你试着将其向下转型为 C，则将被禁止，因为在包的外部没有任何 C 类型可用：

```
// typeinfo/HiddenImplementation.java
// Sneaking around package hiding

import typeinfo.interfacea.*;
import typeinfo.packageaccess.*;

import java.lang.reflect.*;

public class HiddenImplementation {
    public static void main(String[] args) throws Exception {
        A a = HiddenC.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Compile error: cannot find symbol 'C':
        /* if(a instanceof C) {
            C c = (C)a;
            c.g();
        } */
        // Oops! Reflection still allows us to call g():
        callHiddenMethod(a, "g");
        // And even less accessible methods!
        callHiddenMethod(a, "u");
        callHiddenMethod(a, "v");
```

```
        callHiddenMethod(a, "w");
    }

    static void callHiddenMethod(Object a, String methodName) throws
        Exception {
        Method g = a.getClass().getDeclaredMethod(methodName);
        g.setAccessible(true);
        g.invoke(a);
    }
}
```

输出结果:

```
public C.f()
typeinfo.packageaccess.C
public C.g()
package C.u()
protected C.v()
private C.w()
```

正如你所看到的，通过使用反射，仍然可以调用所有方法，甚至是 **private** 方法！如果知道方法名，你就可以在其 `Method` 对象上调用 `setAccessible(true)`，就像在 `callHiddenMethod()` 中看到的那样。

你可能觉得，可以通过只发布编译后的代码来阻止这种情况，但其实这并不能解决问题。因为只需要运行 `javap`（一个随 JDK 发布的反编译器）即可突破这一限制。下面是一个使用 `javap` 的命令行:

```
javap -private C
```

-private 标志表示所有的成员都应该显示，甚至包括私有成员。下面是输出:

```
class typeinfo.packageaccess.C extends
java.lang.Object implements typeinfo.interfacea.A {
  typeinfo.packageaccess.C();
  public void f();
  public void g();
  void u();
  protected void v();
  private void w();
}
```

因此，任何人都可以获取你最私有的方法的名字和签名，然后调用它们。

那如果把接口实现为一个私有内部类，又会怎么样呢？下面展示了这种情况:

```
// typeinfo/InnerImplementation.java
// Private inner classes can't hide from reflection

import typeinfo.interfacea.*;

```

```java
class InnerA {
    private static class C implements A {
        public void f() {
            System.out.println("public C.f()");
        }

        public void g() {
            System.out.println("public C.g()");
        }

        void u() {
            System.out.println("package C.u()");
        }

        protected void v() {
            System.out.println("protected C.v()");
        }

        private void w() {
            System.out.println("private C.w()");
        }
    }

    public static A makeA() {
        return new C();
    }
}

public class InnerImplementation {
    public static void
    main(String[] args) throws Exception {
        A a = InnerA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Reflection still gets into the private class:
        HiddenImplementation.callHiddenMethod(a, "g");
        HiddenImplementation.callHiddenMethod(a, "u");
        HiddenImplementation.callHiddenMethod(a, "v");
        HiddenImplementation.callHiddenMethod(a, "w");
    }
}
```

输出结果:

```
public C.f()
InnerA$C
public C.g()
package C.u()
protected C.v()
private C.w()
```

这里对反射仍然没有任何东西可以隐藏。那么如果是匿名类呢?

```
// typeinfo/AnonymousImplementation.java
// Anonymous inner classes can't hide from reflection

import typeinfo.interfacea.*;

class AnonymousA {
    public static A makeA() {
        return new A() {
            public void f() {
                System.out.println("public C.f()");
            }

            public void g() {
                System.out.println("public C.g()");
            }

            void u() {
                System.out.println("package C.u()");
            }

            protected void v() {
                System.out.println("protected C.v()");
            }

            private void w() {
                System.out.println("private C.w()");
            }
        };
    }
}

public class AnonymousImplementation {
    public static void
    main(String[] args) throws Exception {
        A a = AnonymousA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Reflection still gets into the anonymous class:
        HiddenImplementation.callHiddenMethod(a, "g");
        HiddenImplementation.callHiddenMethod(a, "u");
        HiddenImplementation.callHiddenMethod(a, "v");
        HiddenImplementation.callHiddenMethod(a, "w");
    }
}
```

输出结果:

```
public C.f()
AnonymousA$1
```

```
public C.g()
package C.u()
protected C.v()
private C.w()
```

看起来任何方式都没法阻止反射调用那些非公共访问权限的方法。对于字段来说也是这样，即便是 **private** 字段：

```
// typeinfo/ModifyingPrivateFields.java

import java.lang.reflect.*;

class WithPrivateFinalField {
    private int i = 1;
    private final String s = "I'm totally safe";
    private String s2 = "Am I safe?";

    @Override
    public String toString() {
        return "i = " + i + ", " + s + ", " + s2;
    }
}

public class ModifyingPrivateFields {
    public static void main(String[] args) throws Exception {
        WithPrivateFinalField pf =
                new WithPrivateFinalField();
        System.out.println(pf);
        Field f = pf.getClass().getDeclaredField("i");
        f.setAccessible(true);
        System.out.println(
                "f.getInt(pf): " + f.getInt(pf));
        f.setInt(pf, 47);
        System.out.println(pf);
        f = pf.getClass().getDeclaredField("s");
        f.setAccessible(true);
        System.out.println("f.get(pf): " + f.get(pf));
        f.set(pf, "No, you're not!");
        System.out.println(pf);
        f = pf.getClass().getDeclaredField("s2");
        f.setAccessible(true);
        System.out.println("f.get(pf): " + f.get(pf));
        f.set(pf, "No, you're not!");
        System.out.println(pf);
    }
}
```

输出结果：

```
i = 1, I'm totally safe, Am I safe?
```

```
f.getInt(pf): 1
i = 47, I'm totally safe, Am I safe?
f.get(pf): I'm totally safe
i = 47, I'm totally safe, Am I safe?
f.get(pf): Am I safe?
i = 47, I'm totally safe, No, you're not!
```

但实际上 **final** 字段在被修改时是安全的。运行时系统会在不抛出异常的情况下接受任何修改的尝试，但是实际上不会发生任何修改。

通常，所有这些违反访问权限的操作并不是什么十恶不赦的。如果有人使用这样的技术去调用标志为 **private** 或包访问权限的方法（很明显这些访问权限表示这些人不应该调用它们），那么对他们来说，如果你修改了这些方法的某些地方，他们不应该抱怨。另一方面，总是在类中留下后门，也许会帮助你解决某些特定类型的问题（这些问题往往除此之外，别无它法）。总之，不可否认，反射给我们带来了很多好处。

程序员往往对编程语言提供的访问控制过于自信，甚至认为 Java 在安全性上比其它提供了（明显）更宽松的访问控制的语言要优越[4]。然而，正如你所看到的，事实并不是这样。

20.10 本章小结

RTTI 允许通过匿名类的引用来获取类型信息。初学者极易误用它，因为在学会使用多态调用方法之前，这么做也很有效。有过程化编程背景的人很容易把程序组织成一系列 **switch** 语句，你可以用 RTTI 和 **switch** 实现功能，但这样就损失了多态机制在代码开发和维护过程中的重要价值。面向对象编程语言是想让我们尽可能地使用多态机制，只在非用不可的时候才使用 RTTI。

然而使用多态机制的方法调用，要求我们拥有基类定义的控制权。因为在你扩展程序的时候，可能会发现基类并未包含我们想要的方法。如果基类来自别人的库，这时 RTTI 便是一种解决之道：可继承一个新类，然后添加你需要的方法。在代码的其它地方，可以检查你自己特定的类型，并调用你自己的方法。这样做不会破坏多态性以及程序的扩展能力，因为这样添加一个新的类并不需要修改程序中的 **switch** 语句。但如果想在程序中增加具有新特性的代码，你就必须使用 RTTI 来检查这个特定的类型。

如果只是为了方便某个特定的类，就将某个特性放进基类里边，这将使得从那个基类派生出的所有其它子类都带有这些可能毫无意义的东西。这会导致接口更加不清晰，因为我们必须覆盖从基类继承而来的所有抽象方法，事情就变得很麻烦。举个例子，现在有一个表示乐器 `Instrument` 的类层次结构。假设我们想清理管弦乐队中某些乐器残留的口水，一种办法是在基类 `Instrument` 中放入 `clearSpitValve()` 方法。但这样做会导致类结构混乱，因为这意味着打击乐器 `Percussion`、弦乐器 `Stringed` 和电子乐器 `Electronic` 也需要清理口水。在这个例子中，RTTI 可以提供一种更合理

[4]这个测试的推论是，“如果某人表示线程是容易并且简单的，请确保这个人没有对你的项目做出重要的决策。如果那个人已经做出，那么你就已经陷入麻烦之中了。”

我认为对象就像是生物学里的细胞, 或者网络中的一台计算机, 只能够通过消息来通信 —Alan Kay,Smalltalk 的发明人, 面向对象之父

的解决方案。可以将 `clearSpitValve()` 放在某个合适的类中，在这个例子中是管乐器 `Wind`。不过，在这里你可能会发现还有更好的解决方法，就是将 `prepareInstrument()` 放在基类中，但是初次面对这个问题的读者可能想不到还有这样的解决方案，而误认为必须使用 RTTI。

最后一点，RTTI 有时候也能解决效率问题。假设你的代码运用了多态，但是为了实现多态，导致其中某个对象的效率非常低。这时候，你就可以挑出那个类，使用 RTTI 为它编写一段特别的代码以提高效率。然而必须注意的是，不要太早地关注程序的效率问题，这是个诱人的陷阱。最好先让程序能跑起来，然后再去看看程序能不能跑得更快，下一步才是去解决效率问题（比如使用 Profiler）[5]。

我们已经看到，反射，因其更加动态的编程风格，为我们开创了编程的新世界。但对有些人来说，反射的动态特性却是一种困扰。对那些已经习惯于静态类型检查的安全性的人来说，Java 中允许这种动态类型检查（只在运行时才能检查到，并以异常的形式上报检查结果）的操作似乎是一种错误的方向。有些人想得更远，他们认为引入运行时异常本身就是一种指示，指示我们应该避免这种代码。我发现这种意义的安全是一种错觉，因为总是有些事情是在运行时才发生并抛出异常的，即使是在那些不包含任何 **`try`** 语句块或异常声明的程序中也是如此。因此，我认为一致性错误报告模型的存在使我们能够通过使用反射编写动态代码。当然，尽力编写能够进行静态检查的代码是有价值的，只要你有这样的能力。但是我相信动态代码是将 Java 与其它诸如 C++ 这样的语言区分开的重要工具之一。

[5]这在即将产生的 C++ 的标准中得到了补救。

质量、速度、廉价, 选择其中两个。 —匿名

21 泛型

> 普通的类和方法只能使用特定的类型：基本数据类型或类类型。如果编写的代码需要应用于多种类型，这种严苛的限制对代码的束缚就会很大。

多态是一种面向对象思想的泛化机制。你可以将方法的参数类型设为基类，这样的方法就可以接受任何派生类作为参数，包括暂时还不存在的类。这样的方法更通用，应用范围更广。在类内部也是如此，在任何使用特定类型的地方，基类意味着更大的灵活性。除了 `final` 类（或只提供私有构造函数的类）任何类型都可被扩展，所以大部分时候这种灵活性是自带的。

拘泥于单一的继承体系太过局限，因为只有继承体系中的对象才能适用基类作为参数的方法中。如果方法以接口而不是类作为参数，限制就宽松多了，只要实现了接口就可以。这给予调用方一种选项，通过调整现有的类来实现接口，满足方法参数要求。接口可以突破继承体系的限制。

即便是接口也还是有诸多限制。一旦指定了接口，它就要求你的代码必须使用特定的接口。而我们希望编写更通用的代码，能够适用“非特定的类型”，而不是一个具体的接口或类。

这就是泛型的概念，是 Java 5 的重大变化之一。泛型实现了*参数化类型*，这样你编写的组件（通常是集合）可以适用于多种类型。“泛型”这个术语的含义是“适用于很多类型”。编程语言中泛型出现的初衷是通过解耦类或方法与所使用的类型之间的约束，使得类或方法具备最宽泛的表达力。随后你会发现 Java 中泛型的实现并没有那么“泛”，你可能会质疑“泛型”这个词是否合适用来描述这一功能。

如果你从未接触过参数化类型机制，你会发现泛型对 Java 语言确实是个很有益的补充。在你实例化一个类型参数时，编译器会负责转型并确保类型的正确性。这是一大进步。

然而，如果你了解其他语言（例如 C++ ）的参数化机制，你会发现，Java 泛型并不能满足所有的预期。使用别人创建好的泛型相对容易，但是创建自己的泛型时，就会遇到很多意料之外的麻烦。

这并不是说 Java 泛型毫无用处。在很多情况下，它可以使代码更直接更优雅。不过，如果你见识过那种实现了更纯粹的泛型的编程语言，那么，Java 可能会令你失望。本章会介绍 Java 泛型的优点与局限。我会解释 Java 的泛型是如何发展成现在这样的，希望能够帮助你更有效地使用这个特性。[1]

21.0.1 与 C++ 的比较

Java 的设计者曾说过，这门语言的灵感主要来自 C++ 。尽管如此，学习 Java 时基本不用参考 C++ 。

[1]

过早的优化是罪恶之源。 —Donald Knuth

但是，Java 中的泛型需要与 C++ 进行对比，理由有两个：首先，理解 C++ *模板*（泛型的主要灵感来源，包括基本语法）的某些特性，有助于理解泛型的基础理念。同时，非常重要的一点是，你可以了解 Java 泛型的局限是什么，以及为什么会有这些局限。最终的目标是明确 Java 泛型的边界，让你成为一个程序高手。只有知道了某个技术不能做什么，你才能更好地做到所能做的（部分原因是，不必浪费时间在死胡同里）。

第二个原因是，在 Java 社区中，大家普遍对 C++ 模板有一种误解，而这种误解可能会令你在理解泛型的意图时产生偏差。

因此，本章中会介绍少量 C++ 模板的例子，仅当它们确实可以加深理解时才会引入。

21.1 简单泛型

促成泛型出现的最主要的动机之一是为了创建*集合类*，参见 集合 章节。集合用于存放要使用到的对象。数组也是如此，不过集合比数组更加灵活，功能更丰富。几乎所有程序在运行过程中都会涉及到一组对象，因此集合是可复用性最高的类库之一。

我们先看一个只能持有单个对象的类。这个类可以明确指定其持有的对象的类型：

```
// generics/Holder1.java

class Automobile {}

public class Holder1 {
    private Automobile a;
    public Holder1(Automobile a) { this.a = a; }
    Automobile get() { return a; }
}
```

这个类的可复用性不高，它无法持有其他类型的对象。我们可不希望为碰到的每个类型都编写一个新的类。

在 Java 5 之前，我们可以让这个类直接持有 `Object` 类型的对象：

```
// generics/ObjectHolder.java

public class ObjectHolder {
    private Object a;
    public ObjectHolder(Object a) { this.a = a; }
    public void set(Object a) { this.a = a; }
    public Object get() { return a; }

    public static void main(String[] args) {
        ObjectHolder h2 = new ObjectHolder(new Automobile());
        Automobile a = (Automobile)h2.get();
        h2.set("Not an Automobile");
        String s = (String)h2.get();
```

```
        h2.set(1); // 自动装箱为 Integer
        Integer x = (Integer)h2.get();
    }
}
```

现在，ObjectHolder 可以持有任何类型的对象，在上面的示例中，一个 ObjectHolder 先后持有了三种不同类型的对象。

一个集合中存储多种不同类型的对象的情况很少见，通常而言，我们只会用集合存储同一种类型的对象。泛型的主要目的之一就是用来约定集合要存储什么类型的对象，并且通过编译器确保规约得以满足。

因此，与其使用 Object，我们更希望先指定一个类型占位符，稍后再决定具体使用什么类型。要达到这个目的，需要使用*类型参数*，用尖括号括住，放在类名后面。然后在使用这个类时，再用实际的类型替换此类型参数。在下面的例子中，T 就是类型参数：

```
// generics/GenericHolder.java

public class GenericHolder<T> {
    private T a;
    public GenericHolder() {}
    public void set(T a) { this.a = a; }
    public T get() { return a; }

    public static void main(String[] args) {
        GenericHolder<Automobile> h3 = new GenericHolder<Automobile>();
        h3.set(new Automobile()); // 此处有类型校验
        Automobile a = h3.get();  // 无需类型转换
        //- h3.set("Not an Automobile"); // 报错
        //- h3.set(1);  // 报错
    }
}
```

创建 GenericHolder 对象时，必须指明要持有的对象的类型，将其置于尖括号内，就像 main() 中那样使用。然后，你就只能在 GenericHolder 中存储该类型（或其子类，因为多态与泛型不冲突）的对象了。当你调用 get() 取值时，直接就是正确的类型。

这就是 Java 泛型的核心概念：你只需告诉编译器要使用什么类型，剩下的细节交给它来处理。

你可能注意到 h3 的定义非常繁复。在 = 左边有 GenericHolder<Automobile>，右边又重复了一次。在 Java 5 中，这种写法被解释成“必要的”，但在 Java 7 中设计者修正了这个问题（新的简写语法随后成为备受欢迎的特性）。以下是简写的例子：

```
// generics/Diamond.java

class Bob {}

public class Diamond<T> {
    public static void main(String[] args) {
```

```
        GenericHolder<Bob> h3 = new GenericHolder<>();
        h3.set(new Bob());
    }
}
```

注意，在 h3 的定义处，= 右边的尖括号是空的（称为“钻石语法”），而不是重复左边的类型信息。在本书剩余部分都会使用这种语法。

一般来说，你可以认为泛型和其他类型差不多，只不过它们碰巧有类型参数罢了。在使用泛型时，你只需要指定它们的名称和类型参数列表即可。

21.1.1 一个元组类库

有时一个方法需要能返回多个对象。而 **return** 语句只能返回单个对象，解决方法就是创建一个对象，用它打包想要返回的多个对象。当然，可以在每次需要的时候，专门创建一个类来完成这样的工作。但是有了泛型，我们就可以一劳永逸。同时，还获得了编译时的类型安全。

这个概念称为*元组*，它是将一组对象直接打包存储于单一对象中。可以从该对象读取其中的元素，但不允许向其中存储新对象（这个概念也称为 *数据传输对象*或 *信使*）。

通常，元组可以具有任意长度，元组中的对象可以是不同类型的。不过，我们希望能够为每个对象指明类型，并且从元组中读取出来时，能够得到正确的类型。要处理不同长度的问题，我们需要创建多个不同的元组。下面是一个可以存储两个对象的元组：

```
// onjava/Tuple2.java
package onjava;

public class Tuple2<A, B> {
    public final A a1;
    public final B a2;
    public Tuple2(A a, B b) { a1 = a; a2 = b; }
    public String rep() { return a1 + ", " + a2; }

    @Override
    public String toString() {
        return "(" + rep() + ")";
    }
}
```

构造函数传入要存储的对象。这个元组隐式地保持了其中元素的次序。

初次阅读上面的代码时，你可能认为这违反了 Java 编程的封装原则。a1 和 a2 应该声明为 **private**，然后提供 getFirst() 和 getSecond() 取值方法才对呀？考虑下这样做能提供的“安全性”是什么：元组的使用程序可以读取 a1 和 a2 然后对它们执行任何操作，但无法对 a1 和 a2 重新赋值。例子中的 **final** 可以实现同样的效果，并且更为简洁明了。

另一种设计思路是允许元组的用户给 a1 和 a2 重新赋值。然而，采用上例中的形式无疑更加安全，

如果用户想存储不同的元素，就会强制他们创建新的 Tuple2 对象。

我们可以利用继承机制实现长度更长的元组。添加更多的类型参数就行了：

```
// onjava/Tuple3.java
package onjava;

public class Tuple3<A, B, C> extends Tuple2<A, B> {
    public final C a3;
    public Tuple3(A a, B b, C c) {
        super(a, b);
        a3 = c;
    }

    @Override
    public String rep() {
        return super.rep() + ", " + a3;
    }
}

// onjava/Tuple4.java
package onjava;

public class Tuple4<A, B, C, D>
  extends Tuple3<A, B, C> {
    public final D a4;
    public Tuple4(A a, B b, C c, D d) {
        super(a, b, c);
        a4 = d;
    }

    @Override
    public String rep() {
        return super.rep() + ", " + a4;
    }
}

// onjava/Tuple5.java
package onjava;

public class Tuple5<A, B, C, D, E>
  extends Tuple4<A, B, C, D> {
    public final E a5;
    public Tuple5(A a, B b, C c, D d, E e) {
        super(a, b, c, d);
        a5 = e;
    }

    @Override
    public String rep() {
        return super.rep() + ", " + a5;
```

```
    }
}
```

演示需要，再定义两个类：

```
// generics/Amphibian.java
public class Amphibian {}

// generics/Vehicle.java
public class Vehicle {}
```

使用元组时，你只需要定义一个长度适合的元组，将其作为返回值即可。注意下面例子中方法的返回类型：

```
// generics/TupleTest.java
import onjava.*;

public class TupleTest {
    static Tuple2<String, Integer> f() {
        // 47 自动装箱为 Integer
        return new Tuple2<>("hi", 47);
    }

    static Tuple3<Amphibian, String, Integer> g() {
        return new Tuple3<>(new Amphibian(), "hi", 47);
    }

    static Tuple4<Vehicle, Amphibian, String, Integer> h() {
        return new Tuple4<>(new Vehicle(), new Amphibian(), "hi", 47);
    }

    static Tuple5<Vehicle, Amphibian, String, Integer, Double> k() {
        return new Tuple5<>(new Vehicle(), new Amphibian(), "hi", 47,
            11.1);
    }

    public static void main(String[] args) {
        Tuple2<String, Integer> ttsi = f();
        System.out.println(ttsi);
        // ttsi.a1 = "there"; // 编译错误，因为 final 不能重新赋值
        System.out.println(g());
        System.out.println(h());
        System.out.println(k());
    }
}

/* 输出:
 (hi, 47)
 (Amphibian@1540e19d, hi, 47)
 (Vehicle@7f31245a, Amphibian@6d6f6e28, hi, 47)
```

```
 (Vehicle@330bedb4, Amphibian@2503dbd3, hi, 47, 11.1)
 */
```

有了泛型，你可以很容易地创建元组，令其返回一组任意类型的对象。

通过 `ttsi.a1 = "there"` 语句的报错，我们可以看出，**final** 声明确实可以确保 **public** 字段在对象被构造出来之后就不能重新赋值了。

在上面的程序中，**new** 表达式有些啰嗦。本章稍后会介绍，如何利用 *泛型方法*简化它们。

21.1.2 一个堆栈类

接下来我们看一个稍微复杂一点的例子：堆栈。在 集合 一章中，我们用 `LinkedList` 实现了 `onjava.Stack` 类。在那个例子中，`LinkedList` 本身已经具备了创建堆栈所需的方法。`Stack` 是通过两个泛型类 `Stack<T>` 和 `LinkedList<T>` 的组合来创建。我们可以看出，泛型只不过是一种类型罢了（稍后我们会看到一些例外的情况）。

这次我们不用 `LinkedList` 来实现自己的内部链式存储机制。

```
// generics/LinkedStack.java
// 用链式结构实现的堆栈

public class LinkedStack<T> {
    private static class Node<U> {
        U item;
        Node<U> next;

        Node() { item = null; next = null; }

        Node(U item, Node<U> next) {
            this.item = item;
            this.next = next;
        }

        boolean end() {
            return item == null && next == null;
        }
    }

    private Node<T> top = new Node<>();  // 栈顶

    public void push(T item) {
        top = new Node<>(item, top);
    }

    public T pop() {
        T result = top.item;
        if (!top.end()) {
            top = top.next;
```

```
        }
        return result;
    }

    public static void main(String[] args) {
        LinkedStack<String> lss = new LinkedStack<>();
        for (String s : "Phasers on stun!".split(" ")) {
            lss.push(s);
        }
        String s;
        while ((s = lss.pop()) != null) {
            System.out.println(s);
        }
    }
}
```

输出结果:

```
stun!
on
Phasers
```

内部类 `Node` 也是一个泛型，它拥有自己的类型参数。

这个例子使用了一个 *末端标识* (end sentinel) 来判断栈何时为空。这个末端标识是在构造 `LinkedStack` 时创建的。然后，每次调用 `push()` 就会创建一个 `Node<T>` 对象，并将其链接到前一个 `Node<T>` 对象。当你调用 `pop()` 方法时，总是返回 `top.item`，然后丢弃当前 `top` 所指向的 `Node<T>`，并将 `top` 指向下一个 `Node<T>`，除非到达末端标识，这时就不能再移动 `top` 了。如果已经到达末端，程序还继续调用 `pop()` 方法，它只能得到 **`null`**，说明栈已经空了。

21.1.3 RandomList

作为容器的另一个例子，假设我们需要一个持有特定类型对象的列表，每次调用它的 `select()` 方法时都随机返回一个元素。如果希望这种列表可以适用于各种类型，就需要使用泛型:

```
// generics/RandomList.java
import java.util.*;
import java.util.stream.*;

public class RandomList<T> extends ArrayList<T> {
    private Random rand = new Random(47);

    public T select() {
        return get(rand.nextInt(size()));
    }

    public static void main(String[] args) {
        RandomList<String> rs = new RandomList<>();
```

```
        Array.stream("The quick brown fox jumped over the lazy brown dog".split(" ")).forEach(rs::add);
        IntStream.range(0, 11).forEach(i ->
            System.out.print(rs.select() + " "));
    }
}
```

输出结果:

```
brown over fox quick quick dog brown The brown lazy brown
```

`RandomList` 继承了 `ArrayList` 的所有方法。本例中只添加了 `select()` 这个方法。

21.2 泛型接口

泛型也可以应用于接口。例如 *生成器*，这是一种专门负责创建对象的类。实际上，这是 *工厂方法* 设计模式的一种应用。不过，当使用生成器创建新的对象时，它不需要任何参数，而工厂方法一般需要参数。生成器无需额外的信息就知道如何创建新对象。

一般而言，一个生成器只定义一个方法，用于创建对象。例如 `java.util.function` 类库中的 `Supplier` 就是一个生成器，调用其 `get()` 获取对象。`get()` 是泛型方法，返回值为类型参数 `T`。

为了演示 `Supplier`，我们需要定义几个类。下面是个咖啡相关的继承体系:

```
// generics/coffee/Coffee.java
package generics.coffee;

public class Coffee {
    private static long counter = 0;
    private final long id = counter++;

    @Override
    public String toString() {
        return getClass().getSimpleName() + " " + id;
    }
}


// generics/coffee/Latte.java
package generics.coffee;
public class Latte extends Coffee {}


// generics/coffee/Mocha.java
package generics.coffee;
public class Mocha extends Coffee {}


```

```
// generics/coffee/Cappuccino.java
package generics.coffee;
public class Cappuccino extends Coffee {}


// generics/coffee/Americano.java
package generics.coffee;
public class Americano extends Coffee {}


// generics/coffee/Breve.java
package generics.coffee;
public class Breve extends Coffee {}
```

现在, 我们可以编写一个类, 实现 Supplier<Coffee> 接口, 它能够随机生成不同类型的 Coffee 对象:

```
// generics/coffee/CoffeeSupplier.java
// {java generics.coffee.CoffeeSupplier}
package generics.coffee;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class CoffeeSupplier
implements Supplier<Coffee>, Iterable<Coffee> {
    private Class<?>[] types = { Latte.class, Mocha.class,
        Cappuccino.class, Americano.class, Breve.class };
    private static Random rand = new Random(47);

    public CoffeeSupplier() {}
    // For iteration:
    private int size = 0;
    public CoffeeSupplier(int sz) { size = sz; }

    @Override
    public Coffee get() {
        try {
            return (Coffee) types[rand.nextInt(types.length)].
                newInstance();
        } catch (InstantiationException | IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }

    class CoffeeIterator implements Iterator<Coffee> {
        int count = size;
        @Override
        public boolean hasNext() { return count > 0; }
        @Override
```

```
        public Coffee next() {
            count--;
            return CoffeeSupplier.this.get();
        }
        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    @Override
    public Iterator<Coffee> iterator() {
        return new CoffeeIterator();
    }

    public static void main(String[] args) {
        Stream.generate(new CoffeeSupplier())
              .limit(5)
              .forEach(System.out::println);
        for (Coffee c : new CoffeeSupplier(5)) {
            System.out.println(c);
        }
    }
}
```

输出结果:

```
Americano 0
Latte 1
Americano 2
Mocha 3
Mocha 4
Breve 5
Americano 6
Latte 7
Cappuccino 8
Cappuccino 9
```

参数化的 `Supplier` 接口确保 `get()` 返回值是参数的类型。`CoffeeSupplier` 同时还实现了 `Iterable` 接口，所以能用于 *for-in* 语句。不过，它还需要知道何时终止循环，这正是第二个构造函数的作用。

下面是另一个实现 `Supplier<T>` 接口的例子，它负责生成 Fibonacci 数列:

```
// generics/Fibonacci.java
// Generate a Fibonacci sequence
import java.util.function.*;
import java.util.stream.*;

public class Fibonacci implements Supplier<Integer> {
    private int count = 0;
```

```
    @Override
    public Integer get() { return fib(count++); }

    private int fib(int n) {
        if(n < 2) return 1;
        return fib(n-2) + fib(n-1);
    }

    public static void main(String[] args) {
        Stream.generate(new Fibonacci())
              .limit(18)
              .map(n -> n + " ")
              .forEach(System.out::print);
    }
}
```

输出结果:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
```

虽然我们在 `Fibonacci` 类的里里外外使用的都是 **`int`** 类型，但是其参数类型却是 `Integer`。这个例子引出了 Java 泛型的一个局限性：基本类型无法作为类型参数。不过 Java 5 具备自动装箱和拆箱的功能，可以很方便地在基本类型和相应的包装类之间进行转换。通过这个例子中 `Fibonacci` 类对 **`int`** 的使用，我们已经看到了这种效果。

如果还想更进一步，编写一个实现了 `Iterable` 的 `Fibnoacci` 生成器。我们的一个选择是重写这个类，令其实现 `Iterable` 接口。不过，你并不是总能拥有源代码的控制权，并且，除非必须这么做，否则，我们也不愿意重写一个类。而且我们还有另一种选择，就是创建一个 *适配器* (Adapter) 来实现所需的接口，我们在前面介绍过这个设计模式。

有多种方法可以实现适配器。例如，可以通过继承来创建适配器类:

```
// generics/IterableFibonacci.java
// Adapt the Fibonacci class to make it Iterable
import java.util.*;

public class IterableFibonacci
extends Fibonacci implements Iterable<Integer> {
    private int n;
    public IterableFibonacci(int count) { n = count; }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            @Override
            public boolean hasNext() { return n > 0; }
            @Override
            public Integer next() {
                n--;
```

‘理论’是你知道是这样, 但它却不好用。‘实践’是它很好用, 但你不知道是为什么。程序员将理论和实践结合到一起：既不好用, 也不知道是为什么。 —佚名

```
                return IterableFibonacci.this.get();
            }
            @Override
            public void remove() { // Not implemented
                throw new UnsupportedOperationException();
            }
        };
    }

    public static void main(String[] args) {
        for(int i : new IterableFibonacci(18))
            System.out.print(i + " ");
    }
}
```

输出结果:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
```

在 *for-in* 语句中使用 `IterableFibonacci`,必须在构造函数中提供一个边界值,这样 `hasNext()` 才知道何时返回 **false**，结束循环。

21.3 泛型方法

到目前为止，我们已经研究了参数化整个类。其实还可以参数化类中的方法。类本身可能是泛型的，也可能不是，不过这与它的方法是否是泛型的并没有什么关系。

泛型方法独立于类而改变方法。作为准则，请“尽可能”使用泛型方法。通常将单个方法泛型化要比将整个类泛型化更清晰易懂。

如果方法是 **static** 的，则无法访问该类的泛型类型参数，因此，如果使用了泛型类型参数，则它必须是泛型方法。

要定义泛型方法，请将泛型参数列表放置在返回值之前，如下所示:

```
// generics/GenericMethods.java

public class GenericMethods {
    public <T> void f(T x) {
        System.out.println(x.getClass().getName());
    }

    public static void main(String[] args) {
        GenericMethods gm = new GenericMethods();
        gm.f("");
        gm.f(1);
        gm.f(1.0);
        gm.f(1.0F);
        gm.f('c');
```

```
        gm.f(gm);
    }
}
/* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*/
```

尽管可以同时对类及其方法进行参数化，但这里未将 **GenericMethods** 类参数化。只有方法 `f()` 具有类型参数，该参数由方法返回类型之前的参数列表指示。

对于泛型类，必须在实例化该类时指定类型参数。使用泛型方法时，通常不需要指定参数类型，因为编译器会找出这些类型。这称为 *类型参数推断*。因此，对 `f()` 的调用看起来像普通的方法调用，并且 `f()` 看起来像被重载了无数次一样。它甚至会接受 **GenericMethods** 类型的参数。

如果使用基本类型调用 `f()`，自动装箱就开始起作用，自动将基本类型包装在它们对应的包装类型中。

21.3.1 变长参数和泛型方法

泛型方法和变长参数列表可以很好地共存：

```java
// generics/GenericVarargs.java

import java.util.ArrayList;
import java.util.List;

public class GenericVarargs {
    @SafeVarargs
    public static <T> List<T> makeList(T... args) {
        List<T> result = new ArrayList<>();
        for (T item : args)
            result.add(item);
        return result;
    }

    public static void main(String[] args) {
        List<String> ls = makeList("A");
        System.out.println(ls);
        ls = makeList("A", "B", "C");
        System.out.println(ls);
        ls = makeList(
                "ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
        System.out.println(ls);
    }
```

```
}
/* Output:
[A]
[A, B, C]
[A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R,
S, T, U, V, W, X, Y, Z]
*/
```

此处显示的 `makeList()` 方法产生的功能与标准库的 `java.util.Arrays.asList()` 方法相同。

`@SafeVarargs` 注解保证我们不会对变长参数列表进行任何修改，这是正确的，因为我们只从中读取。如果没有此注解，编译器将无法知道这些并会发出警告。

21.3.2 一个泛型的 Supplier

这是一个为任意具有无参构造方法的类生成 **Supplier** 的类。为了减少键入，它还包括一个用于生成 **BasicSupplier** 的泛型方法：

```java
// onjava/BasicSupplier.java
// Supplier from a class with a no-arg constructor
package onjava;

import java.util.function.Supplier;

public class BasicSupplier<T> implements Supplier<T> {
    private Class<T> type;

    public BasicSupplier(Class<T> type) {
        this.type = type;
    }

    @Override
    public T get() {
        try {
            // Assumes type is a public class:
            return type.newInstance();
        } catch (InstantiationException |
                IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }

    // Produce a default Supplier from a type token:
    public static <T> Supplier<T> create(Class<T> type) {
        return new BasicSupplier<>(type);
    }
}
```

此类提供了产生以下对象的基本实现:

1. 是 **public** 的。因为 **BasicSupplier** 在单独的包中，所以相关的类必须具有 **public** 权限，而不仅仅是包级访问权限。
2. 具有无参构造方法。要创建一个这样的 **BasicSupplier** 对象，请调用 `create()` 方法，并将要生成类型的类型令牌传递给它。通用的 `create()` 方法提供了 `BasicSupplier.create(MyType.class)` 这种较简洁的语法来代替较笨拙的 `new BasicSupplier <MyType>(MyType.class)`。

例如，这是一个具有无参构造方法的简单类:

```
// generics/CountedObject.java

public class CountedObject {
    private static long counter = 0;
    private final long id = counter++;

    public long id() {
        return id;
    }

    @Override
    public String toString() {
        return "CountedObject " + id;
    }
}
```

CountedObject 类可以跟踪自身创建了多少个实例，并通过 `toString()` 报告这些实例的数量。**BasicSupplier** 可以轻松地为 **CountedObject** 创建 **Supplier**:

```
  // generics/BasicSupplierDemo.java

import onjava.BasicSupplier;

import java.util.stream.Stream;

public class BasicSupplierDemo {
    public static void main(String[] args) {
        Stream.generate(
                BasicSupplier.create(CountedObject.class))
                .limit(5)
                .forEach(System.out::println);
    }
}
/* Output:
CountedObject 0
CountedObject 1
CountedObject 2
CountedObject 3
```

```
CountedObject 4
*/
```

泛型方法减少了产生 **Supplier** 对象所需的代码量。Java 泛型强制传递 **Class** 对象，以便在 `create()` 方法中将其用于类型推断。

21.3.3 简化元组的使用

使用类型参数推断和静态导入，我们将把早期的元组重写为更通用的库。在这里，我们使用重载的静态方法创建元组：

```java
// onjava/Tuple.java
// Tuple library using type argument inference
package onjava;

public class Tuple {
    public static <A, B> Tuple2<A, B> tuple(A a, B b) {
        return new Tuple2<>(a, b);
    }

    public static <A, B, C> Tuple3<A, B, C>
    tuple(A a, B b, C c) {
        return new Tuple3<>(a, b, c);
    }

    public static <A, B, C, D> Tuple4<A, B, C, D>
    tuple(A a, B b, C c, D d) {
        return new Tuple4<>(a, b, c, d);
    }

    public static <A, B, C, D, E>
    Tuple5<A, B, C, D, E> tuple(A a, B b, C c, D d, E e) {
        return new Tuple5<>(a, b, c, d, e);
    }
}
```

我们修改 **TupleTest.java** 来测试 **Tuple.java**：

```java
// generics/TupleTest2.java

import onjava.Tuple2;
import onjava.Tuple3;
import onjava.Tuple4;
import onjava.Tuple5;

import static onjava.Tuple.tuple;

public class TupleTest2 {
    static Tuple2<String, Integer> f() {
```

```
        return tuple("hi", 47);
    }

    static Tuple2 f2() {
        return tuple("hi", 47);
    }

    static Tuple3<Amphibian, String, Integer> g() {
        return tuple(new Amphibian(), "hi", 47);
    }

    static Tuple4<Vehicle, Amphibian, String, Integer> h() {
        return tuple(
                new Vehicle(), new Amphibian(), "hi", 47);
    }

    static Tuple5<Vehicle, Amphibian,
            String, Integer, Double> k() {
        return tuple(new Vehicle(), new Amphibian(),
                "hi", 47, 11.1);
    }

    public static void main(String[] args) {
        Tuple2<String, Integer> ttsi = f();
        System.out.println(ttsi);
        System.out.println(f2());
        System.out.println(g());
        System.out.println(h());
        System.out.println(k());
    }
}
/* Output:
(hi, 47)
(hi, 47)
(Amphibian@14ae5a5, hi, 47)
(Vehicle@135fbaa4, Amphibian@45ee12a7, hi, 47)
(Vehicle@4b67cf4d, Amphibian@7ea987ac, hi, 47, 11.1)
*/
```

请注意，`f()` 返回一个参数化的 **Tuple2** 对象，而 `f2()` 返回一个未参数化的 **Tuple2** 对象。编译器不会在这里警告 `f2()` ，因为返回值未以参数化方式使用。从某种意义上说，它被“向上转型”为一个未参数化的 **Tuple2** 。但是，如果尝试将 `f2()` 的结果放入到参数化的 **Tuple2** 中，则编译器将发出警告。

21.3.4 一个 Set 工具

对于泛型方法的另一个示例，请考虑由 **Set** 表示的数学关系。这些被方便地定义为可用于所有不同类型的泛型方法：

```java
// onjava/Sets.java

package onjava;

import java.util.HashSet;
import java.util.Set;

public class Sets {
    public static <T> Set<T> union(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<>(a);
        result.addAll(b);
        return result;
    }

    public static <T>
    Set<T> intersection(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<>(a);
        result.retainAll(b);
        return result;
    }

    // Subtract subset from superset:
    public static <T> Set<T>
    difference(Set<T> superset, Set<T> subset) {
        Set<T> result = new HashSet<>(superset);
        result.removeAll(subset);
        return result;
    }

    // Reflexive--everything not in the intersection:
    public static <T> Set<T> complement(Set<T> a, Set<T> b) {
        return difference(union(a, b), intersection(a, b));
    }
}
```

前三个方法通过将第一个参数的引用复制到新的 **HashSet** 对象中来复制第一个参数，因此不会直接修改参数集合。因此，返回值是一个新的 **Set** 对象。

这四种方法代表数学集合操作：`union()` 返回一个包含两个参数并集的 **Set**，`intersection()` 返回一个包含两个参数集合交集的 **Set**，`difference()` 从 **superset** 中减去 **subset** 的元素，而 `complement()` 返回所有不在交集中的元素的 **Set**。作为显示这些方法效果的简单示例的一部分，下面是一个包含不同水彩名称的 **enum**：

```java
// generics/watercolors/Watercolors.java

package watercolors;

public enum Watercolors {
    ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW,
```

```
    ORANGE, BRILLIANT_RED, CRIMSON, MAGENTA,
    ROSE_MADDER, VIOLET, CERULEAN_BLUE_HUE,
    PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE,
    PERMANENT_GREEN, VIRIDIAN_HUE, SAP_GREEN,
    YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
    BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
}
```

为了方便起见（不必全限定所有名称），将其静态导入到以下示例中。本示例使用 **EnumSet** 轻松从 **enum** 中创建 **Set** 。（可以在第二十二章枚举一章中了解有关 **EnumSet** 的更多信息。）在这里，静态方法 `EnumSet.range()` 要求提供所要在结果 **Set** 中创建的元素范围的第一个和最后一个元素：

```
// generics/WatercolorSets.java

import watercolors.*;

import java.util.EnumSet;
import java.util.Set;

import static watercolors.Watercolors.*;
import static onjava.Sets.*;

public class WatercolorSets {
    public static void main(String[] args) {
        Set<Watercolors> set1 =
                EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
        Set<Watercolors> set2 =
                EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
        System.out.println("set1: " + set1);
        System.out.println("set2: " + set2);
        System.out.println(
                "union(set1, set2): " + union(set1, set2));
        Set<Watercolors> subset = intersection(set1, set2);
        System.out.println(
                "intersection(set1, set2): " + subset);
        System.out.println("difference(set1, subset): " +
                difference(set1, subset));
        System.out.println("difference(set2, subset): " +
                difference(set2, subset));
        System.out.println("complement(set1, set2): " +
                complement(set1, set2));
    }
}
/* Output:
set1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER,
VIOLET, CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE]
set2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
```

```
BURNT_UMBER]
union(set1, set2): [BURNT_SIENNA, BRILLIANT_RED,
YELLOW_OCHRE, MAGENTA, SAP_GREEN, CERULEAN_BLUE_HUE,
ULTRAMARINE, VIRIDIAN_HUE, VIOLET, RAW_UMBER,
ROSE_MADDER, PERMANENT_GREEN, BURNT_UMBER,
PHTHALO_BLUE, CRIMSON, COBALT_BLUE_HUE]
intersection(set1, set2): [PERMANENT_GREEN,
CERULEAN_BLUE_HUE, ULTRAMARINE, VIRIDIAN_HUE,
PHTHALO_BLUE, COBALT_BLUE_HUE]
difference(set1, subset): [BRILLIANT_RED, MAGENTA,
VIOLET, CRIMSON, ROSE_MADDER]
difference(set2, subset): [BURNT_SIENNA, YELLOW_OCHRE,
BURNT_UMBER, SAP_GREEN, RAW_UMBER]
complement(set1, set2): [BURNT_SIENNA, BRILLIANT_RED,
YELLOW_OCHRE, MAGENTA, SAP_GREEN, VIOLET, RAW_UMBER,
ROSE_MADDER, BURNT_UMBER, CRIMSON]
*/
```

接下来的例子使用 `Sets.difference()` 方法来展示 **java.util** 包中各种 **Collection** 和 **Map** 类之间的方法差异:

```java
// onjava/CollectionMethodDifferences.java
// {java onjava.CollectionMethodDifferences}

package onjava;

import java.lang.reflect.Method;
import java.util.*;
import java.util.stream.Collectors;

public class CollectionMethodDifferences {
    static Set<String> methodSet(Class<?> type) {
        return Arrays.stream(type.getMethods())
                .map(Method::getName)
                .collect(Collectors.toCollection(TreeSet::new));
    }

    static void interfaces(Class<?> type) {
        System.out.print("Interfaces in " +
                type.getSimpleName() + ": ");
        System.out.println(
                Arrays.stream(type.getInterfaces())
                        .map(Class::getSimpleName)
                        .collect(Collectors.toList()));
    }

    static Set<String> object = methodSet(Object.class);

    static {
        object.add("clone");
```

```
    }

    static void
    difference(Class<?> superset, Class<?> subset) {
        System.out.print(superset.getSimpleName() +
                " extends " + subset.getSimpleName() +
                ", adds: ");
        Set<String> comp = Sets.difference(
                methodSet(superset), methodSet(subset));
        comp.removeAll(object); // Ignore 'Object' methods
        System.out.println(comp);
        interfaces(superset);
    }

    public static void main(String[] args) {
        System.out.println("Collection: " +
                methodSet(Collection.class));
        interfaces(Collection.class);
        difference(Set.class, Collection.class);
        difference(HashSet.class, Set.class);
        difference(LinkedHashSet.class, HashSet.class);
        difference(TreeSet.class, Set.class);
        difference(List.class, Collection.class);
        difference(ArrayList.class, List.class);
        difference(LinkedList.class, List.class);
        difference(Queue.class, Collection.class);
        difference(PriorityQueue.class, Queue.class);
        System.out.println("Map: " + methodSet(Map.class));
        difference(HashMap.class, Map.class);
        difference(LinkedHashMap.class, HashMap.class);
        difference(SortedMap.class, Map.class);
        difference(TreeMap.class, Map.class);
    }
}
/* Output:
Collection: [add, addAll, clear, contains, containsAll,
equals, forEach, hashCode, isEmpty, iterator,
parallelStream, remove, removeAll, removeIf, retainAll,
size, spliterator, stream, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable,
Serializable]
TreeSet extends Set, adds: [headSet,
descendingIterator, descendingSet, pollLast, subSet,
floor, tailSet, ceiling, last, lower, comparator,
pollFirst, first, higher]
```

```
Interfaces in TreeSet: [NavigableSet, Cloneable,
Serializable]
List extends Collection, adds: [replaceAll, get,
indexOf, subList, set, sort, lastIndexOf, listIterator]
Interfaces in List: [Collection]
ArrayList extends List, adds: [trimToSize,
ensureCapacity]
Interfaces in ArrayList: [List, RandomAccess,
Cloneable, Serializable]
LinkedList extends List, adds: [offerFirst, poll,
getLast, offer, getFirst, removeFirst, element,
removeLastOccurrence, peekFirst, peekLast, push,
pollFirst, removeFirstOccurrence, descendingIterator,
pollLast, removeLast, pop, addLast, peek, offerLast,
addFirst]
Interfaces in LinkedList: [List, Deque, Cloneable,
Serializable]
Queue extends Collection, adds: [poll, peek, offer,
element]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, compute, computeIfAbsent,
computeIfPresent, containsKey, containsValue, entrySet,
equals, forEach, get, getOrDefault, hashCode, isEmpty,
keySet, merge, put, putAll, putIfAbsent, remove,
replace, replaceAll, size, values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [lastKey, subMap,
comparator, firstKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingKeySet,
navigableKeySet, higherEntry, higherKey, floorKey,
subMap, ceilingKey, pollLastEntry, firstKey, lowerKey,
headMap, tailMap, lowerEntry, ceilingEntry,
descendingMap, pollFirstEntry, lastKey, firstEntry,
floorEntry, comparator, lastEntry]
Interfaces in TreeMap: [NavigableMap, Cloneable,
Serializable]
*/
```

在第十二章 集合的本章小节 部分将会用到这里的输出结果。

21.4 构建复杂模型

泛型的一个重要好处是能够简单安全地创建复杂模型。例如，我们可以轻松地创建一个元组列表：

```
// generics/TupleList.java
// Combining generic types to make complex generic types

import onjava.Tuple4;

import java.util.ArrayList;

public class TupleList<A, B, C, D>
        extends ArrayList<Tuple4<A, B, C, D>> {
    public static void main(String[] args) {
        TupleList<Vehicle, Amphibian, String, Integer> tl =
                new TupleList<>();
        tl.add(TupleTest2.h());
        tl.add(TupleTest2.h());
        tl.forEach(System.out::println);
    }
}
/* Output:
(Vehicle@7cca494b, Amphibian@7ba4f24f, hi, 47)
(Vehicle@3b9a45b3, Amphibian@7699a589, hi, 47)
*/
```

这将产生一个功能强大的数据结构，而无需太多代码。

下面是第二个例子。每个类都是组成块，总体包含很多个块。在这里，该模型是一个具有过道，货架和产品的零售商店：

```
// generics/Store.java
// Building a complex model using generic collections

import onjava.Suppliers;

import java.util.ArrayList;
import java.util.Random;
import java.util.function.Supplier;

class Product {
    private final int id;
    private String description;
    private double price;

    Product(int idNumber, String descr, double price) {
        id = idNumber;
        description = descr;
        this.price = price;
```

```
        System.out.println(toString());
    }

    @Override
    public String toString() {
        return id + ": " + description +
                ", price: $" + price;
    }

    public void priceChange(double change) {
        price += change;
    }

    public static Supplier<Product> generator =
            new Supplier<Product>() {
                private Random rand = new Random(47);

                @Override
                public Product get() {
                    return new Product(rand.nextInt(1000), "Test",
                            Math.round(
                                    rand.nextDouble() * 1000.0) + 0.99)
                                        ;
                }
            };
}

class Shelf extends ArrayList<Product> {
    Shelf(int nProducts) {
        Suppliers.fill(this, Product.generator, nProducts);
    }
}

class Aisle extends ArrayList<Shelf> {
    Aisle(int nShelves, int nProducts) {
        for (int i = 0; i < nShelves; i++)
            add(new Shelf(nProducts));
    }
}

class CheckoutStand {
}

class Office {
}

public class Store extends ArrayList<Aisle> {
    private ArrayList<CheckoutStand> checkouts =
            new ArrayList<>();
    private Office office = new Office();

```

```
    public Store(
            int nAisles, int nShelves, int nProducts) {
        for (int i = 0; i < nAisles; i++)
            add(new Aisle(nShelves, nProducts));
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        for (Aisle a : this)
            for (Shelf s : a)
                for (Product p : s) {
                    result.append(p);
                    result.append("\n");
                }
        return result.toString();
    }

    public static void main(String[] args) {
        System.out.println(new Store(5, 4, 3));
    }
}
/* Output: (First 8 Lines)
258: Test, price: $400.99
861: Test, price: $160.99
868: Test, price: $417.99
207: Test, price: $268.99
551: Test, price: $114.99
278: Test, price: $804.99
520: Test, price: $554.99
140: Test, price: $530.99
                  ...
*/
```

`Store.toString()` 显示了结果：尽管有复杂的层次结构，但多层的集合仍然是类型安全的和可管理的。令人印象深刻的是，组装这样的模型并不需要耗费过多精力。

Shelf 使用 `Suppliers.fill()` 这个实用程序，该实用程序接受 **Collection**（第一个参数），并使用 **Supplier**（第二个参数），以元素的数量为 **n**（第三个参数）来填充它。**Suppliers** 类将会在本章末尾定义，其中的方法都是在执行某种填充操作，并在本章的其他示例中使用。

21.5 泛型擦除

当你开始更深入地钻研泛型时，会发现有大量的东西初看起来是没有意义的。例如，尽管可以说 `ArrayList.class`，但不能说成 `ArrayList<Integer>.class`。考虑下面的情况：

```
// generics/ErasedTypeEquivalence.java
```

有两种软件设计的方式：一种是使它足够简单以致于明显没有缺陷，另一种则是使它足够复杂以致于没有明显的缺陷。
—C.A.R. Hoare

```java

import java.util.*;

public class ErasedTypeEquivalence {

    public static void main(String[] args) {
        Class c1 = new ArrayList<String>().getClass();
        Class c2 = new ArrayList<Integer>().getClass();
        System.out.println(c1 == c2);
    }

}
/* Output:
true
*/
```

ArrayList<String> 和 ArrayList<Integer> 应该是不同的类型。不同的类型会有不同的行为。例如，如果尝试向 ArrayList<String> 中放入一个 Integer，所得到的行为（失败）和向 ArrayList<Integer> 中放入一个 Integer 所得到的行为（成功）完全不同。然而上面的程序认为它们是相同的类型。

下面的例子是对该谜题的补充：

```java
// generics/LostInformation.java

import java.util.*;

class Frob {}
class Fnorkle {}
class Quark<Q> {}

class Particle<POSITION, MOMENTUM> {}

public class LostInformation {

    public static void main(String[] args) {
        List<Frob> list = new ArrayList<>();
        Map<Frob, Fnorkle> map = new HashMap<>();
        Quark<Fnorkle> quark = new Quark<>();
        Particle<Long, Double> p = new Particle<>();
        System.out.println(Arrays.toString(list.getClass().
            getTypeParameters()));
        System.out.println(Arrays.toString(map.getClass().
            getTypeParameters()));
        System.out.println(Arrays.toString(quark.getClass().
            getTypeParameters()));
        System.out.println(Arrays.toString(p.getClass().
            getTypeParameters()));
    }

```

```
}
/* Output:
[E]
[K,V]
[Q]
[POSITION,MOMENTUM]
```

根据 JDK 文档，**Class.getTypeParameters()** “返回一个 **TypeVariable** 对象数组，表示泛型声明中声明的类型参数…” 这暗示你可以发现这些参数类型。但是正如上例中输出所示，你只能看到用作参数占位符的标识符，这并非有用的信息。

残酷的现实是：

在泛型代码内部，无法获取任何有关泛型参数类型的信息。

因此，你可以知道如类型参数标识符和泛型边界这些信息，但无法得知实际的类型参数从而用来创建特定的实例。如果你曾是 C++ 程序员，那么这个事实会让你很沮丧，在使用 Java 泛型工作时，它是必须处理的最基本的问题。

Java 泛型是使用擦除实现的。这意味着当你在使用泛型时，任何具体的类型信息都被擦除了，你唯一知道的就是你在使用一个对象。因此，`List<String>` 和 `List<Integer>` 在运行时实际上是相同的类型。它们都被擦除成原生类型 `List`。

理解擦除并知道如何处理它，是你在学习 Java 泛型时面临的最大障碍之一。这也是本节将要探讨的内容。

21.5.1 C++ 的方式

下面是使用模版的 C++ 示例。你会看到类型参数的语法十分相似，因为 Java 是受 C++ 启发的：

```
// generics/Templates.cpp

#include <iostream>
using namespace std;

template<class T> class Manipulator {
    T obj;
public:
    Manipulator(T x) { obj = x; }
    void manipulate() { obj.f(); }
};

class HasF {
public:
    void f() { cout << "HasF::f()" << endl; }
};

int main() {
    HasF hf;
```

```
    Manipulator<HasF> manipulator(hf);
    manipulator.manipulate();
}
/* Output:
HasF::f()
*/
```

Manipulator 类存储了一个 **T** 类型的对象。`manipulate()` 方法会调用 **obj** 上的 `f()` 方法。它是如何知道类型参数 **T** 中存在 `f()` 方法的呢？C++ 编译器会在你实例化模版时进行检查，所以在 `Manipulator<HasF>` 实例化的那一刻，它看到 **HasF** 中含有一个方法 `f()`。如果情况并非如此，你就会得到一个编译期错误，保持类型安全。

用 C++ 编写这种代码很简单，因为当模版被实例化时，模版代码就知道模版参数的类型。Java 泛型就不同了。下面是 **HasF** 的 Java 版本：

```
// generics/HasF.java

public class HasF {
    public void f() {
        System.out.println("HasF.f()");
    }
}
```

如果我们将示例的其余代码用 Java 实现，就不会通过编译：

```
// generics/Manipulation.java
// {WillNotCompile}

class Manipulator<T> {
    private T obj;

    Manipulator(T x) {
        obj = x;
    }

    // Error: cannot find symbol: method f():
    public void manipulate() {
        obj.f();
    }
}

public class Manipulation {
    public static void main(String[] args) {
        HasF hf = new HasF();
        Manipulator<HasF> manipulator = new Manipulator<>(hf);
        manipulator.manipulate();
    }
}
```

因为擦除，Java 编译器无法将 `manipulate()` 方法必须能调用 **obj** 的 `f()` 方法这一需求映射到

HasF 具有 f() 方法这个事实上。为了调用 f()，我们必须协助泛型类，给定泛型类一个边界，以此告诉编译器只能接受遵循这个边界的类型。这里重用了 **extends** 关键字。由于有了边界，下面的代码就能通过编译：

```
public class Manipulator2<T extends HasF> {
    private T obj;

    Manipulator2(T x) {
        obj = x;
    }

    public void manipulate() {
        obj.f();
    }
}
```

边界 <T extends HasF> 声明 T 必须是 HasF 类型或其子类。如果情况确实如此，就可以安全地在 **obj** 上调用 f() 方法。

我们说泛型类型参数会擦除到它的第一个边界（可能有多个边界，稍后你将看到）。我们还提到了类型参数的擦除。编译器实际上会把类型参数替换为它的擦除，就像上面的示例，**T** 擦除到了 **HasF**，就像在类的声明中用 **HasF** 替换了 **T** 一样。

你可能正确地观察到了泛型在 **Manipulator2.java** 中没有贡献任何事。你可以很轻松地自己去执行擦除，生成没有泛型的类：

```
// generics/Manipulator3.java

class Manipulator3 {
    private HasF obj;

    Manipulator3(HasF x) {
        ojb = x;
    }

    public void manipulate() {
        obj.f();
    }
}
```

这提出了很重要的一点：泛型只有在类型参数比某个具体类型（以及其子类）更加“泛化”——代码能跨多个类工作时才有用。因此，类型参数和它们在有用的泛型代码中的应用，通常比简单的类替换更加复杂。但是，不能因此认为使用 <T extends HasF> 形式就是有缺陷的。例如，如果某个类有一个返回 **T** 的方法，那么泛型就有所帮助，因为它们之后将返回确切的类型：

```
// generics/ReturnGenericType.java

public class ReturnGenericType<T extends HasF> {
```

程序必须是为了给人看而写，给机器去执行只是附带任务。

```
    private T obj;

    ReturnGenericType(T x) {
        obj = x;
    }

    public T get() {
        return obj;
    }
}
```

你必须查看所有的代码，从而确定代码是否复杂到必须使用泛型的程度。

我们将在本章稍后看到有关边界的更多细节。

21.5.2 迁移兼容性

为了减少潜在的关于擦除的困惑，你必须清楚地认识到这不是一个语言特性。它是 Java 实现泛型的一种妥协，因为泛型不是 Java 语言出现时就有的，所以就有了这种妥协。它会使你痛苦，因此你需要尽早习惯它并了解为什么它会这样。

如果 Java 1.0 就含有泛型的话，那么这个特性就不会使用擦除来实现——它会使用具体化，保持参数类型为第一类实体，因此你就能在类型参数上执行基于类型的语言操作和反射操作。本章稍后你会看到，擦除减少了泛型的泛化性。泛型在 Java 中仍然是有用的，只是不如它们本来设想的那么有用，而原因就是擦除。

在基于擦除的实现中，泛型类型被当作第二类类型处理，即不能在某些重要的上下文使用泛型类型。泛型类型只有在静态类型检测期间才出现，在此之后，程序中的所有泛型类型都将被擦除，替换为它们的非泛型上界。例如，`List<T>` 这样的类型注解会被擦除为 **List**，普通的类型变量在未指定边界的情况下会被擦除为 **Object**。

擦除的核心动机是你可以在泛化的客户端上使用非泛型的类库，反之亦然。这经常被称为“迁移兼容性”。在理想情况下，所有事物将在指定的某天被泛化。在现实中，即使程序员只编写泛型代码，他们也必须处理 Java 5 之前编写的非泛型类库。这些类库的作者可能从没想过要泛化他们的代码，或许他们可能刚刚开始接触泛型。

因此 Java 泛型不仅必须支持向后兼容性——现有的代码和类文件仍然合法，继续保持之前的含义——而且还必须支持迁移兼容性，使得类库能按照它们自己的步调变为泛型，当某个类库变为泛型时，不会破坏依赖于它的代码和应用。在确定了这个目标后，Java 设计者们和从事此问题相关工作的各个团队决策认为擦除是唯一可行的解决方案。擦除使得这种向泛型的迁移成为可能，允许非泛型的代码和泛型代码共存。

例如，假设一个应用使用了两个类库 **X** 和 **Y**，**Y** 使用了类库 **Z**。随着 Java 5 的出现，这个应用和这些类库的创建者最终可能希望迁移到泛型上。但是当进行迁移时，它们有着不同的动机和限制。为了实现迁移兼容性，每个类库与应用必须与其他所有的部分是否使用泛型无关。因此，它们不能探测其

他类库是否使用了泛型。因此，某个特定的类库使用了泛型这样的证据必须被” 擦除 “。

如果没有某种类型的迁移途径，所有已经构建了很长时间的类库就需要与希望迁移到 Java 泛型上的开发者们说再见了。类库毫无争议是编程语言的一部分，对生产效率有着极大的影响，所以这种代码无法接受。擦除是否是最佳的或唯一的迁移途径，还待时间来证明。

21.5.3 擦除的问题

因此，擦除主要的正当理由是从非泛化代码到泛化代码的转变过程，以及在不破坏现有类库的情况下将泛型融入到语言中。擦除允许你继续使用现有的非泛型客户端代码，直至客户端准备好用泛型重写这些代码。这是一个崇高的动机，因为它不会骤然破坏所有现有的代码。

擦除的代价是显著的。泛型不能用于显式地引用运行时类型的操作中，例如转型、**instanceof** 操作和 **new** 表达式。因为所有关于参数的类型信息都丢失了，当你在编写泛型代码时，必须时刻提醒自己，你只是看起来拥有有关参数的类型信息而已。

考虑如下的代码段：

```
class Foo<T> {
    T var;
}
```

看上去当你创建一个 **Foo** 实例时：

```
Foo<Cat> f = new Foo<>();
```

class Foo 中的代码应该知道现在工作于 **Cat** 之上。泛型语法也在强烈暗示整个类中所有 T 出现的地方都被替换，就像在 C++ 中一样。但是事实并非如此，当你在编写这个类的代码时，必须提醒自己：“不，这只是一个 **Object**“。

另外，擦除和迁移兼容性意味着，使用泛型并不是强制的，尽管你可能希望这样：

```
// generics/ErasureAndInheritance.java

class GenericBase<T> {
    private T element;

    public void set(T arg) {
        element = arg;
    }

    public T get() {
        return element;
    }
}

class Derived1<T> extends GenericBase<T> {}

```

```
class Derived2 extends GenericBase {} // No warning

// class Derived3 extends GenericBase<?> {}
// Strange error:
// unexpected type
// required: class or interface without bounds
public class ErasureAndInteritance {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Derived2 d2 = new Derived2();
        Object obj = d2.get();
        d2.set(obj); // Warning here!
    }
}
```

Derived2 继承自 **GenericBase**，但是没有任何类型参数，编译器没有发出任何警告。直到调用 `set()` 方法时才出现警告。

为了关闭警告，Java 提供了一个注解，我们可以在列表中看到它：

```
@SuppressWarnings("unchecked")
```

这个注解放置在产生警告的方法上，而不是整个类上。当你要关闭警告时，最好尽可能地“聚焦”，这样就不会因为过于宽泛地关闭警告，而导致意外地遮蔽掉真正的问题。

可以推断，**Derived3** 产生的错误意味着编译器期望得到一个原生基类。

当你希望将类型参数不仅仅当作 Object 处理时，就需要付出额外努力来管理边界，并且与在 C++、Ada 和 Eiffel 这样的语言中获得参数化类型相比，你需要付出多得多的努力来获得少得多的回报。这并不是说，对于大多数的编程问题而言，这些语言通常都会比 Java 更得心应手，只是说它们的参数化类型机制相比 Java 更灵活、更强大。

21.5.4 边界处的动作

因为擦除，我发现了泛型最令人困惑的方面是可以表示没有任何意义的事物。例如：

```
// generics/ArrayMaker.java

import java.lang.reflect.*;
import java.util.*;

public class ArrayMaker<T> {
    private Class<T> kind;

    public ArrayMaker(Class<T> kind) {
        this.kind = kind;
    }

    @SuppressWarnings("unchecked")
```

```
    T[] create(int size) {
        return (T[]) Array.newInstance(kind, size);
    }

    public static void main(String[] args) {
        ArrayMaker<String> stringMaker = new ArrayMaker<>(String.class);
        String[] stringArray = stringMaker.create(9);
        System.out.println(Arrays.toString(stringArray));
    }
}
/* Output
[null,null,null,null,null,null,null,null,null]
*/
```

即使 **kind** 被存储为 `Class<T>`，擦除也意味着它实际被存储为没有任何参数的 **Class**。因此，当你在使用它时，例如创建数组，`Array.newInstance()` 实际上并未拥有 **kind** 所蕴含的类型信息。所以它不会产生具体的结果，因而必须转型，这会产生一条令你无法满意的警告。

注意，对于在泛型中创建数组，使用 `Array.newInstance()` 是推荐的方式。

如果我们创建一个集合而不是数组，情况就不同了：

```
// generics/ListMaker.java

import java.util.*;

public class ListMaker<T> {
    List<T> create() {
        return new ArrayList<>();
    }

    public static void main(String[] args) {
        ListMaker<String> stringMaker = new ListMaker<>();
        List<String> stringList = stringMaker.create();
    }
}
```

编译器不会给出任何警告，尽管我们知道（从擦除中）在 `create()` 内部的 `new ArrayList<>()` 中的 `<T>` 被移除了——在运行时，类内部没有任何 `<T>`，因此这看起来毫无意义。但是如果你遵从这种思路，并将这个表达式改为 `new ArrayList()`，编译器就会发出警告。

本例中这么做真的毫无意义吗？如果在创建 **List** 的同时向其中放入一些对象呢，像这样：

```
// generics/FilledList.java

import java.util.*;
import java.util.function.*;
import onjava.*;

public class FilledList<T> extends ArrayList<T> {
```

```
    FilledList<Supplier<T> gen, int size) {
        Suppliers.fill(this, gen, size);
    }

    public FilledList(T t, int size) {
        for (int i = 0; i < size; i++) {
            this.add(t);
        }
    }

    public static void main(String[] args) {
        List<String> list = new FilledList<>("Hello", 4);
        System.out.println(list);
        // Supplier version:
        List<Integer> ilist = new FilledList<>(() -> 47, 4);
        System.out.println(ilist);
    }
}
/* Output:
[Hello,Hello,Hello,Hello]
[47,47,47,47]
```

即使编译器无法得知 `add()` 中的 **T** 的任何信息，但它仍可以在编译期确保你放入 **FilledList** 中的对象是 **T** 类型。因此，即使擦除移除了方法或类中的实际类型的信息，编译器仍可以确保方法或类中使用的类型的内部一致性。

因为擦除移除了方法体中的类型信息，所以在运行时的问题就是*边界*：即对象进入和离开方法的地点。这些正是编译器在编译期执行类型检查并插入转型代码的地点。

考虑如下这段非泛型示例：

```
// generics/SimpleHolder.java

public class SimpleHolder {
    private Object obj;

    public void set(Object obj) {
        this.obj = obj;
    }

    public Object get() {
        return obj;
    }

    public static void main(String[] args) {
        SimpleHolder holder = new SimpleHolder();
        holder.set("Item");
        String s = (String) holder.get();
    }
}
```

如果用 **javap -c SimpleHolder** 反编译这个类，会得到如下内容（经过编辑）：

```
public void set(java.lang.Object);
   0: aload_0
   1: aload_1
   2: putfield #2; // Field obj:Object;
   5: return

public java.lang.Object get();
   0: aload_0
   1: getfield #2; // Field obj:Object;
   4: areturn

public static void main(java.lang.String[]);
   0: new #3; // class SimpleHolder
   3: dup
   4: invokespecial #4; // Method "<init>":()V
   7: astore_1
   8: aload_1
   9: ldc #5; // String Item
   11: invokevirtual #6; // Method set:(Object;)V
   14: aload_1
   15: invokevirtual #7; // Method get:()Object;
   18: checkcast #8; // class java/lang/String
   21: astore_2
   22: return
```

`set()` 和 `get()` 方法存储和产生值，转型在调用 `get()` 时接受检查。

现在将泛型融入上例代码中：

```java
// generics/GenericHolder2.java

public class GenericHolder2<T> {
    private T obj;

    public void set(T obj) {
        this.obj = obj;
    }

    public T get() {
        return obj;
    }

    public static void main(String[] args) {
        GenericHolder2<String> holder =  new GenericHolder2<>();
        holder.set("Item");
        String s = holder.get();
    }
}
```

从 get() 返回后的转型消失了，但是我们还知道传递给 set() 的值在编译期会被检查。下面是相关的字节码：

```
public void set(java.lang.Object);
   0: aload_0
   1: aload_1
   2: putfield #2; // Field obj:Object;
   5: return

public java.lang.Object get();
   0: aload_0
   1: getfield #2; // Field obj:Object;
   4: areturn

public static void main(java.lang.String[]);
   0: new #3; // class GenericHolder2
   3: dup
   4: invokespecial #4; // Method "<init>":()V
   7: astore_1
   8: aload_1
   9: ldc #5; // String Item
   11: invokevirtual #6; // Method set:(Object;)V
   14: aload_1
   15: invokevirtual #7; // Method get:()Object;
   18: checkcast #8; // class java/lang/String
   21: astore_2
   22: return
```

所产生的字节码是相同的。对进入 set() 的类型进行检查是不需要的，因为这将由编译器执行。而对 get() 返回的值进行转型仍然是需要的，只不过不需要你来操作，它由编译器自动插入，这样你就不用编写（阅读）杂乱的代码。

get() 和 set() 产生了相同的字节码，这就告诉我们泛型的所有动作都发生在边界处——对入参的编译器检查和对返回值的转型。这有助于澄清对擦除的困惑，记住："边界就是动作发生的地方"。

21.6 补偿擦除

因为擦除，我们将失去执行泛型代码中某些操作的能力。无法在运行时知道确切类型：

```
// generics/Erased.java
// {WillNotCompile}

public class Erased<T> {
    private final int SIZE = 100;

    public void f(Object arg) {
        // error: illegal generic type for instanceof
```

```
        if (arg instanceof T) {
        }
        // error: unexpected type
        T var = new T();
        // error: generic array creation
        T[] array = new T[SIZE];
        // warning: [unchecked] unchecked cast
        T[] array = (T[]) new Object[SIZE];

    }
}
```

有时，我们可以对这些问题进行编程，但是有时必须通过引入类型标签来补偿擦除。这意味着为所需的类型显式传递一个 **Class** 对象，以在类型表达式中使用它。

例如，由于擦除了类型信息，因此在上一个程序中尝试使用 **instanceof** 将会失败。类型标签可以使用动态 `isInstance()`：

```
// generics/ClassTypeCapture.java

class Building {
}

class House extends Building {
}

public class ClassTypeCapture<T> {
    Class<T> kind;

    public ClassTypeCapture(Class<T> kind) {
        this.kind = kind;
    }

    public boolean f(Object arg) {
        return kind.isInstance(arg);
    }

    public static void main(String[] args) {
        ClassTypeCapture<Building> ctt1 =
                new ClassTypeCapture<>(Building.class);
        System.out.println(ctt1.f(new Building()));
        System.out.println(ctt1.f(new House()));
        ClassTypeCapture<House> ctt2 =
                new ClassTypeCapture<>(House.class);
        System.out.println(ctt2.f(new Building()));
        System.out.println(ctt2.f(new House()));
    }
}
/* Output:
true
```

```
true
false
true
*/
```

编译器来保证类型标签与泛型参数相匹配。

21.6.1 创建类型的实例

试图在 **Erased.java** 中 `new T()` 是行不通的，部分原因是由于擦除，部分原因是编译器无法验证 **T** 是否具有默认（无参）构造函数。但是在 C++ 中，此操作自然，直接且安全（在编译时检查）：

```cpp
// generics/InstantiateGenericType.cpp
// C++, not Java!

template<class T> class Foo {
  T x; // Create a field of type T
  T* y; // Pointer to T
public:
  // Initialize the pointer:
  Foo() { y = new T(); }
};

class Bar {};

int main() {
  Foo<Bar> fb;
  Foo<int> fi; // ... and it works with primitives
}
```

Java 中的解决方案是传入一个工厂对象，并使用该对象创建新实例。方便的工厂对象只是 **Class** 对象，因此，如果使用类型标记，则可以使用 `newInstance()` 创建该类型的新对象：

```java
// generics/InstantiateGenericType.java

import java.util.function.Supplier;

class ClassAsFactory<T> implements Supplier<T> {
    Class<T> kind;

    ClassAsFactory(Class<T> kind) {
        this.kind = kind;
    }

    @Override
    public T get() {
        try {
            return kind.newInstance();
        } catch (InstantiationException |
```

```
                IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
}

class Employee {
    @Override
    public String toString() {
        return "Employee";
    }
}

public class InstantiateGenericType {
    public static void main(String[] args) {
        ClassAsFactory<Employee> fe =
                new ClassAsFactory<>(Employee.class);
        System.out.println(fe.get());
        ClassAsFactory<Integer> fi =
                new ClassAsFactory<>(Integer.class);
        try {
            System.out.println(fi.get());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
/* Output:
Employee
java.lang.InstantiationException: java.lang.Integer
*/
```

这样可以编译，但对于 `ClassAsFactory<Integer>` 会失败，这是因为 **Integer** 没有无参构造函数。由于错误不是在编译时捕获的，因此语言创建者不赞成这种方法。他们建议使用显式工厂（**Supplier**）并约束类型，以便只有实现该工厂的类可以这样创建对象。这是创建工厂的两种不同方法：

```
// generics/FactoryConstraint.java

import onjava.Suppliers;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Supplier;

class IntegerFactory implements Supplier<Integer> {
    private int i = 0;

    @Override
```

```
    public Integer get() {
        return ++i;
    }
}

class Widget {
    private int id;

    Widget(int n) {
        id = n;
    }

    @Override
    public String toString() {
        return "Widget " + id;
    }

    public static
    class Factory implements Supplier<Widget> {
        private int i = 0;

        @Override
        public Widget get() {
            return new Widget(++i);
        }
    }
}

class Fudge {
    private static int count = 1;
    private int n = count++;

    @Override
    public String toString() {
        return "Fudge " + n;
    }
}

class Foo2<T> {
    private List<T> x = new ArrayList<>();

    Foo2(Supplier<T> factory) {
        Suppliers.fill(x, factory, 5);
    }

    @Override
    public String toString() {
        return x.toString();
    }
}

```

```
public class FactoryConstraint {
    public static void main(String[] args) {
        System.out.println(
                new Foo2<>(new IntegerFactory()));
        System.out.println(
                new Foo2<>(new Widget.Factory()));
        System.out.println(
                new Foo2<>(Fudge::new));
    }
}
/* Output:
[1, 2, 3, 4, 5]
[Widget 1, Widget 2, Widget 3, Widget 4, Widget 5]
[Fudge 1, Fudge 2, Fudge 3, Fudge 4, Fudge 5]
*/
```

IntegerFactory 本身就是通过实现 `Supplier<Integer>` 的工厂。**Widget** 包含一个内部类，它是一个工厂。还要注意，**Fudge** 并没有做任何类似于工厂的操作，并且传递 `Fudge::new` 仍然会产生工厂行为，因为编译器将对函数方法 `::new` 的调用转换为对 `get()` 的调用。

另一种方法是模板方法设计模式。在以下示例中，`create()` 是模板方法，在子类中被重写以生成该类型的对象：

```
// generics/CreatorGeneric.java

abstract class GenericWithCreate<T> {
    final T element;

    GenericWithCreate() {
        element = create();
    }

    abstract T create();
}

class X {
}

class XCreator extends GenericWithCreate<X> {
    @Override
    X create() {
        return new X();
    }

    void f() {
        System.out.println(
                element.getClass().getSimpleName());
    }
}

```

```
public class CreatorGeneric {
    public static void main(String[] args) {
        XCreator xc = new XCreator();
        xc.f();
    }
}
/* Output:
X
*/
```

GenericWithCreate 包含 `element` 字段，并通过无参构造函数强制其初始化，该构造函数又调用抽象的 `create()` 方法。这种创建方式可以在子类中定义，同时建立 **T** 的类型。

21.6.2 泛型数组

正如在 **Erased.java** 中所看到的，我们无法创建泛型数组。通用解决方案是在试图创建泛型数组的时候使用 **ArrayList**：

```
// generics/ListOfGenerics.java

import java.util.ArrayList;
import java.util.List;

public class ListOfGenerics<T> {
    private List<T> array = new ArrayList<>();

    public void add(T item) {
        array.add(item);
    }

    public T get(int index) {
        return array.get(index);
    }
}
```

这样做可以获得数组的行为，并且还具有泛型提供的编译时类型安全性。

有时，仍然会创建泛型类型的数组（例如，**ArrayList** 在内部使用数组）。可以通过使编译器满意的方式定义对数组的通用引用：

```
// generics/ArrayOfGenericReference.java

class Generic<T> {
}

public class ArrayOfGenericReference {
    static Generic<Integer>[] gia;
}
```

编译器接受此操作而不产生警告。但是我们永远无法创建具有该确切类型（包括类型参数）的数组，因此有点令人困惑。由于所有数组，无论它们持有什么类型，都具有相同的结构（每个数组插槽的大小和数组布局），因此似乎可以创建一个 **Object** 数组并将其转换为所需的数组类型。实际上，这确实可以编译，但是会产生 **ClassCastException**：

```
// generics/ArrayOfGeneric.java

public class ArrayOfGeneric {
    static final int SIZE = 100;
    static Generic<Integer>[] gia;

    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        try {
            gia = (Generic<Integer>[]) new Object[SIZE];
        } catch (ClassCastException e) {
            System.out.println(e.getMessage());
        }
        // Runtime type is the raw (erased) type:
        gia = (Generic<Integer>[]) new Generic[SIZE];
        System.out.println(gia.getClass().getSimpleName());
        gia[0] = new Generic<>();
        //- gia[1] = new Object(); // Compile-time error
        // Discovers type mismatch at compile time:
        //- gia[2] = new Generic<Double>();
    }
}
/* Output:
[Ljava.lang.Object; cannot be cast to [LGeneric;
Generic[]
*/
```

问题在于数组会跟踪其实际类型，而该类型是在创建数组时建立的。因此，即使 `gia` 被强制转换为 `Generic<Integer>[]`，该信息也仅在编译时存在（并且没有 **@SuppressWarnings** 注解，将会收到有关该强制转换的警告）。在运行时，它仍然是一个 **Object** 数组，这会引起问题。成功创建泛型类型的数组的唯一方法是创建一个已擦除类型的新数组，并将其强制转换。

让我们看一个更复杂的示例。考虑一个包装数组的简单泛型包装器：

```
// generics/GenericArray.java

public class GenericArray<T> {
    private T[] array;

    @SuppressWarnings("unchecked")
    public GenericArray(int sz) {
        array = (T[]) new Object[sz];
    }

```

当有这样的一种编程语言出现：它能让程序员用简单的英语编程，你将会发现，程序员都开始不会说英语。— Anonymous Linguist

```
    public void put(int index, T item) {
        array[index] = item;
    }

    public T get(int index) {
        return array[index];
    }

    // Method that exposes the underlying representation:
    public T[] rep() {
        return array;
    }

    public static void main(String[] args) {
        GenericArray<Integer> gai = new GenericArray<>(10);
        try {
            Integer[] ia = gai.rep();
        } catch (ClassCastException e) {
            System.out.println(e.getMessage());
        }
        // This is OK:
        Object[] oa = gai.rep();
    }
}
/* Output:
[Ljava.lang.Object; cannot be cast to
[Ljava.lang.Integer;
*/
```

和以前一样，我们不能说 `T[] array = new T[sz]`，所以我们创建了一个 **Object** 数组并将其强制转换。

`rep()` 方法返回一个 `T[]`，在主方法中它应该是 `gai` 的 `Integer[]`，但是如果调用它并尝试将结果转换为 `Integer[]` 引用，则会得到 **ClassCastException**，这再次是因为实际的运行时类型为 `Object[]`。

如果再注释掉 **@SuppressWarnings** 注解后编译 **GenericArray.java**，则编译器会产生警告：

```
GenericArray.java uses unchecked or unsafe operations.
Recompile with -Xlint:unchecked for details.
```

在这里，我们收到了一个警告，我们认为这是有关强制转换的。

但是要真正确定，请使用 `-Xlint: unchecked` 进行编译：

```
GenericArray.java:7: warning: [unchecked] unchecked cast    array = (T
    [])new Object[sz];                    ^  required: T[]  found:
    Object[]  where T is a type-variable:    T extends Object declared
    in class GenericArray 1 warning
```

确实是在抱怨那个强制转换。由于警告会变成噪音，因此，一旦我们确认预期会出现特定警告，我们可以做的最好的办法就是使用 **@SuppressWarnings** 将其关闭。这样，当警告确实出现时，我们将进行实际调查。

由于擦除，数组的运行时类型只能是 `Object[]` 。如果我们立即将其转换为 `T[]` ，则在编译时会丢失数组的实际类型，并且编译器可能会错过一些潜在的错误检查。因此，最好在集合中使用 `Object[]` ，并在使用数组元素时向 **T** 添加强制类型转换。让我们来看看在 **GenericArray.java** 示例中会是怎么样的：

```
// generics/GenericArray2.java

public class GenericArray2<T> {
    private Object[] array;

    public GenericArray2(int sz) {
        array = new Object[sz];
    }

    public void put(int index, T item) {
        array[index] = item;
    }

    @SuppressWarnings("unchecked")
    public T get(int index) {
        return (T) array[index];
    }

    @SuppressWarnings("unchecked")
    public T[] rep() {
        return (T[]) array; // Unchecked cast
    }

    public static void main(String[] args) {
        GenericArray2<Integer> gai =
                new GenericArray2<>(10);
        for (int i = 0; i < 10; i++)
            gai.put(i, i);
        for (int i = 0; i < 10; i++)
            System.out.print(gai.get(i) + " ");
        System.out.println();
        try {
            Integer[] ia = gai.rep();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
/* Output:
0 1 2 3 4 5 6 7 8 9
```

```
java.lang.ClassCastException: [Ljava.lang.Object;
cannot be cast to [Ljava.lang.Integer;
*/
```

最初，看起来并没有太大不同，只是转换的位置移动了。没有 **@SuppressWarnings** 注解，仍然会收到“unchecked”警告。但是，内部表示现在是 `Object[]` 而不是 `T[]` 。调用 `get()` 时，它将对象强制转换为 **T** ，实际上这是正确的类型，因此很安全。但是，如果调用 `rep()` ，它将再次尝试将 `Object[]` 强制转换为 `T[]` ，但仍然不正确，并在编译时生成警告，并在运行时生成异常。因此，无法破坏基础数组的类型，该基础数组只能是 `Object[]` 。在内部将数组视为 `Object[]` 而不是 `T[]` 的优点是，我们不太可能会忘记数组的运行时类型并意外地引入了 bug，尽管大多数（也许是全部）此类错误会在运行时被迅速检测到。

对于新代码，请传入类型标记。在这种情况下，**GenericArray** 如下所示：

```java
// generics/GenericArrayWithTypeToken.java

import java.lang.reflect.Array;

public class GenericArrayWithTypeToken<T> {
    private T[] array;

    @SuppressWarnings("unchecked")
    public GenericArrayWithTypeToken(Class<T> type, int sz) {
        array = (T[]) Array.newInstance(type, sz);
    }

    public void put(int index, T item) {
        array[index] = item;
    }

    public T get(int index) {
        return array[index];
    }

    // Expose the underlying representation:
    public T[] rep() {
        return array;
    }

    public static void main(String[] args) {
        GenericArrayWithTypeToken<Integer> gai =
                new GenericArrayWithTypeToken<>(
                        Integer.class, 10);
        // This now works:
        Integer[] ia = gai.rep();
    }
}
```

类型标记 **Class<T>** 被传递到构造函数中以从擦除中恢复，因此尽管必须使用 **@Suppress-**

Warnings 关闭来自强制类型转换的警告，但我们仍可以创建所需的实际数组类型。一旦获得了实际的类型，就可以返回它并产生所需的结果，如在主方法中看到的那样。数组的运行时类型是确切的类型 `T[]` 。

不幸的是，如果查看 Java 标准库中的源代码，你会发现到处都有从 **Object** 数组到参数化类型的转换。例如，这是 **ArrayList** 中，复制一个 **Collection** 的构造函数，这里为了简化，去除了源码中对此不重要的代码：

```
public ArrayList(Collection c) {
  size = c.size();
  elementData = (E[])new Object[size];
  c.toArray(elementData);
}
```

如果你浏览 **ArrayList.java** 的代码，将会发现很多此类强制转换。当我们编译它时会发生什么？

```
Note: ArrayList.java uses unchecked or unsafe operations
Note: Recompile with -Xlint:unchecked for details.
```

果然，标准库会产生很多警告。如果你使用过 C 语言，尤其是使用 ANSI C 之前的语言，你会记住警告的特殊效果：发现警告后，可以忽略它们。因此，除非程序员必须对其进行处理，否则最好不要从编译器发出任何类型的消息。

Neal Gafter（Java 5 的主要开发人员之一）在他的博客中[2]指出，他在重写 Java 库时是很随意、马虎的，我们不应该像他那样做。Neal 还指出，他在不破坏现有接口的情况下无法修复某些 Java 库代码。因此，即使在 Java 库源代码中出现了一些习惯用法，它们也不一定是正确的做法。当查看库代码时，我们不能认为这就是要在自己代码中必须遵循的示例。

请注意，在 Java 文献中推荐使用类型标记技术，例如 Gilad Bracha 的论文《Generics in the Java Programming Language》[3]，他指出：“例如，这种用法已广泛用于新的 API 中以处理注解。”我发现此技术在人们对于舒适度的看法方面存在一些不一致之处；有些人强烈喜欢本章前面介绍的工厂方法。

21.7 边界

边界（bounds）在本章的前面进行了简要介绍。边界允许我们对泛型使用的参数类型施加约束。尽管这可以强制执行有关应用了泛型类型的规则，但潜在的更重要的效果是我们可以在绑定的类型中调用方法。

[2]事实证明，质数实际上并不是散列桶的理想容量。近来，（经过广泛的测试）Java 的散列函数都使用 2 的整数次方。对现代的处理器来说，除法与求余数是最慢的操作。使用 2 的整数次方长度的散列表，可用掩码代替除法。

[3]请注意，在 64 位处理器上可能不会发生这种情况，从而消除了这个问题。

由于擦除会删除类型信息，因此唯一可用于无限制泛型参数的方法是那些 **Object** 可用的方法。但是，如果将该参数限制为某类型的子集，则可以调用该子集中的方法。为了应用约束，Java 泛型使用了 **extends** 关键字。

重要的是要理解，当用于限定泛型类型时，**extends** 的含义与通常的意义截然不同。此示例展示边界的基础应用：

```
// generics/BasicBounds.java

interface HasColor {
    java.awt.Color getColor();
}

class WithColor<T extends HasColor> {
    T item;

    WithColor(T item) {
        this.item = item;
    }

    T getItem() {
        return item;
    }

    // The bound allows you to call a method:
    java.awt.Color color() {
        return item.getColor();
    }
}

class Coord {
    public int x, y, z;
}

// This fails. Class must be first, then interfaces:
// class WithColorCoord<T extends HasColor & Coord> {

// Multiple bounds:
class WithColorCoord<T extends Coord & HasColor> {
    T item;

    WithColorCoord(T item) {
        this.item = item;
    }

    T getItem() {
        return item;
    }

    java.awt.Color color() {
```

```
        return item.getColor();
    }

    int getX() {
        return item.x;
    }

    int getY() {
        return item.y;
    }

    int getZ() {
        return item.z;
    }
}

interface Weight {
    int weight();
}

// As with inheritance, you can have only one
// concrete class but multiple interfaces:
class Solid<T extends Coord & HasColor & Weight> {
    T item;

    Solid(T item) {
        this.item = item;
    }

    T getItem() {
        return item;
    }

    java.awt.Color color() {
        return item.getColor();
    }

    int getX() {
        return item.x;
    }

    int getY() {
        return item.y;
    }

    int getZ() {
        return item.z;
    }

    int weight() {
        return item.weight();
```

```
    }
}

class Bounded
        extends Coord implements HasColor, Weight {
    @Override
    public java.awt.Color getColor() {
        return null;
    }

    @Override
    public int weight() {
        return 0;
    }
}

public class BasicBounds {
    public static void main(String[] args) {
        Solid<Bounded> solid =
                new Solid<>(new Bounded());
        solid.color();
        solid.getY();
        solid.weight();
    }
}
```

你可能会观察到 **BasicBounds.java** 中似乎包含一些冗余，它们可以通过继承来消除。在这里，每个继承级别还添加了边界约束：

```
// generics/InheritBounds.java

class HoldItem<T> {
    T item;

    HoldItem(T item) {
        this.item = item;
    }

    T getItem() {
        return item;
    }
}

class WithColor2<T extends HasColor>
        extends HoldItem<T> {
    WithColor2(T item) {
        super(item);
    }

    java.awt.Color color() {
```

```
        return item.getColor();
    }
}

class WithColorCoord2<T extends Coord & HasColor>
        extends WithColor2<T> {
    WithColorCoord2(T item) {
        super(item);
    }

    int getX() {
        return item.x;
    }

    int getY() {
        return item.y;
    }

    int getZ() {
        return item.z;
    }
}

class Solid2<T extends Coord & HasColor & Weight>
        extends WithColorCoord2<T> {
    Solid2(T item) {
        super(item);
    }

    int weight() {
        return item.weight();
    }
}

public class InheritBounds {
    public static void main(String[] args) {
        Solid2<Bounded> solid2 =
                new Solid2<>(new Bounded());
        solid2.color();
        solid2.getY();
        solid2.weight();
    }
}
```

HoldItem 拥有一个对象,因此此行为将继承到 **WithColor2** 中,这也需要其参数符合 **HasColor**。**WithColorCoord2** 和 **Solid2** 进一步扩展了层次结构，并在每个级别添加了边界。现在，这些方法已被继承，并且在每个类中不再重复。

这是一个具有更多层次的示例:

```java
// generics/EpicBattle.java
// Bounds in Java generics

import java.util.List;

interface SuperPower {
}

interface XRayVision extends SuperPower {
    void seeThroughWalls();
}

interface SuperHearing extends SuperPower {
    void hearSubtleNoises();
}

interface SuperSmell extends SuperPower {
    void trackBySmell();
}

class SuperHero<POWER extends SuperPower> {
    POWER power;

    SuperHero(POWER power) {
        this.power = power;
    }

    POWER getPower() {
        return power;
    }
}

class SuperSleuth<POWER extends XRayVision>
        extends SuperHero<POWER> {
    SuperSleuth(POWER power) {
        super(power);
    }

    void see() {
        power.seeThroughWalls();
    }
}

class
CanineHero<POWER extends SuperHearing & SuperSmell>
        extends SuperHero<POWER> {
    CanineHero(POWER power) {
        super(power);
    }

    void hear() {
```

```
        power.hearSubtleNoises();
    }

    void smell() {
        power.trackBySmell();
    }
}

class SuperHearSmell
        implements SuperHearing, SuperSmell {
    @Override
    public void hearSubtleNoises() {
    }

    @Override
    public void trackBySmell() {
    }
}

class DogPerson extends CanineHero<SuperHearSmell> {
    DogPerson() {
        super(new SuperHearSmell());
    }
}

public class EpicBattle {
    // Bounds in generic methods:
    static <POWER extends SuperHearing>
    void useSuperHearing(SuperHero<POWER> hero) {
        hero.getPower().hearSubtleNoises();
    }

    static <POWER extends SuperHearing & SuperSmell>
    void superFind(SuperHero<POWER> hero) {
        hero.getPower().hearSubtleNoises();
        hero.getPower().trackBySmell();
    }

    public static void main(String[] args) {
        DogPerson dogPerson = new DogPerson();
        useSuperHearing(dogPerson);
        superFind(dogPerson);
        // You can do this:
        List<? extends SuperHearing> audioPeople;
        // But you can't do this:
        // List<? extends SuperHearing & SuperSmell> dogPs;
    }
}
```

接下来将要研究的通配符将会把范围限制在单个类型。

21.8 通配符

你已经在 集合 章节中看到了一些简单示例使用了通配符——在泛型参数表达式中的问号，在 类型信息 一章中这种示例更多。本节将更深入地探讨这个特性。

我们的起始示例要展示数组的一种特殊行为：你可以将派生类的数组赋值给基类的引用：

```
// generics/CovariantArrays.java

class Fruit {}

class Apple extends Fruit {}

class Jonathan extends Apple {}

class Orange extends Fruit {}

public class CovariantArrays {

    public static void main(String[] args) {
        Fruit[] fruit = new Apple[10];
        fruit[0] = new Apple(); // OK
        fruit[1] = new Jonathan(); // OK
        // Runtime type is Apple[], not Fruit[] or Orange[]:
        try {
            // Compiler allows you to add Fruit:
            fruit[0] = new Fruit(); // ArrayStoreException
        } catch (Exception e) {
            System.out.println(e);
        }
        try {
            // Compiler allows you to add Oranges:
            fruit[0] = new Orange(); // ArrayStoreException
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
/* Output:
java.lang.ArrayStoreException: Fruit
java.lang.ArrayStoreException: Orange
```

`main()` 中的第一行创建了 **Apple** 数组，并赋值给一个 **Fruit** 数组引用。这是有意义的，因为 **Apple** 也是一种 **Fruit**，因此 **Apple** 数组应该也是一个 **Fruit** 数组。

但是，如果实际的数组类型是 **Apple[]**，你可以在其中放置 **Apple** 或 **Apple** 的子类型，这在编译期和运行时都可以工作。但是你也可以在数组中放置 **Fruit** 对象。这对编译器来说是有意义的，因为它有一个 **Fruit[]** 引用——它有什么理由不允许将 **Fruit** 对象或任何从 **Fruit** 继承出来的对象（比如 **Orange**），放置到这个数组中呢？因此在编译期，这是允许的。然而，运行时的数组机制知道它处理的

是 **Apple[]**，因此会在向数组中放置异构类型时抛出异常。

向上转型用在这里不合适。你真正在做的是将一个数组赋值给另一个数组。数组的行为是持有其他对象，这里只是因为我们能够向上转型而已，所以很明显，数组对象可以保留有关它们包含的对象类型的规则。看起来就像数组对它们持有的对象是有意识的，因此在编译期检查和运行时检查之间，你不能滥用它们。

数组的这种赋值并不是那么可怕，因为在运行时你可以发现插入了错误的类型。但是泛型的主要目标之一是将这种错误检测移到编译期。所以当我们试图使用泛型集合代替数组时，会发生什么呢？

```
// generics/NonCovariantGenerics.java
// {WillNotCompile}

import java.util.*;

public class NonCovariantGenerics {
    // Compile Error: incompatible types:
    List<Fruit> flist = new ArrayList<Apple>();
}
```

尽管你在首次阅读这段代码时会认为“不能将一个 **Apple** 集合赋值给一个 **Fruit** 集合”。记住，泛型不仅仅是关于集合，它真正要表达的是“不能把一个涉及 **Apple** 的泛型赋值给一个涉及 **Fruit** 的泛型”。如果像在数组中的情况一样，编译器对代码的了解足够多，可以确定所涉及到的集合，那么它可能会留下一些余地。但是它不知道任何有关这方面的信息，因此它拒绝向上转型。然而实际上这也不是向上转型—— **Apple** 的 **List** 不是 **Fruit** 的 **List**。**Apple** 的 **List** 将持有 **Apple** 和 **Apple** 的子类型，**Fruit** 的 **List** 将持有任何类型的 **Fruit**。是的，这包括 **Apple**，但是它不是一个 **Apple** 的 **List**，它仍然是 **Fruit** 的 **List**。**Apple** 的 **List** 在类型上不等价于 **Fruit** 的 **List**，即使 **Apple** 是一种 **Fruit** 类型。

真正的问题是我们在讨论的集合类型，而不是集合持有对象的类型。与数组不同，泛型没有内建的协变类型。这是因为数组是完全在语言中定义的，因此可以具有编译期和运行时的内建检查，但是在使用泛型时，编译器和运行时系统不知道你想用类型做什么，以及应该采用什么规则。

但是，有时你想在两个类型间建立某种向上转型关系。通配符可以产生这种关系。

```
// generics/GenericsAndCovariance.java

import java.util.*;

public class GenericsAndCovariance {

    public static void main(String[] args) {
        // Wildcards allow covariance:
        List<? extends Fruit> flist = new ArrayList<>();
        // Compile Error: can't add any type of object:
        // flist.add(new Apple());
        // flist.add(new Fruit());
```

```
        // flist.add(new Object());
        flist.add(null); // Legal but uninteresting
        // We know it returns at least Fruit:
        Fruit f = flist.get(0);
    }

}
```

flist 的类型现在是 `List<? extends Fruit>`，你可以读作"一个具有任何从 **Fruit** 继承的类型的列表"。然而，这实际上并不意味着这个 **List** 将持有任何类型的 **Fruit**。通配符引用的是明确的类型，因此它意味着"某种 **flist** 引用没有指定的具体类型"。因此这个被赋值的 **List** 必须持有诸如 **Fruit** 或 **Apple** 这样的指定类型，但是为了向上转型为 **Fruit**，这个类型是什么没人在意。

List 必须持有一种具体的 **Fruit** 或 **Fruit** 的子类型，但是如果你不关心具体的类型是什么，那么你能对这样的 **List** 做什么呢？如果不知道 **List** 中持有的对象是什么类型，你怎能保证安全地向其中添加对象呢？就像在 **CovariantArrays.java** 中向上转型一样，你不能，除非编译器而不是运行时系统可以阻止这种操作的发生。你很快就会发现这个问题。

你可能认为事情开始变得有点走极端了，因为现在你甚至不能向刚刚声明过将持有 **Apple** 对象的 **List** 中放入一个 **Apple** 对象。是的，但编译器并不知道这一点。`List<? extends Fruit>` 可能合法地指向一个 `List<Orange>`。一旦执行这种类型的向上转型，你就丢失了向其中传递任何对象的能力，甚至传递 **Object** 也不行。

另一方面，如果你调用了一个返回 **Fruit** 的方法，则是安全的，因为你知道这个 **List** 中的任何对象至少具有 **Fruit** 类型，因此编译器允许这么做。

21.8.1 编译器有多聪明

现在你可能会猜想自己不能去调用任何接受参数的方法，但是考虑下面的代码：

```
// generics/CompilerIntelligence.java

import java.util.*;

public class CompilerIntelligence {

    public static void main(String[] args) {
        List<? extends Fruit> flist = Arrays.asList(new Apple());
        Apple a = (Apple) flist.get(0); // No warning
        flist.contains(new Apple()); // Argument is 'Object'
        flist.indexOf(new Apple()); // Argument is 'Object'
    }

}
```

这里对 `contains()` 和 `indexOf()` 的调用接受 **Apple** 对象作为参数，执行没问题。这是否意味

着编译器实际上会检查代码，以查看是否有某个特定的方法修改了它的对象？

通过查看 **ArrayList** 的文档，我们发现编译器没有那么聪明。尽管 `add()` 接受一个泛型参数类型的参数，但 `contains()` 和 `indexOf()` 接受的参数类型是 **Object**。因此当你指定一个 `ArrayList<? extends Fruit>` 时，`add()` 的参数就变成了“**? extends Fruit**”。从这个描述中，编译器无法得知这里需要 **Fruit** 的哪个具体子类型，因此它不会接受任何类型的 **Fruit**。如果你先把 **Apple** 向上转型为 **Fruit**，也没有关系——编译器仅仅会拒绝调用像 `add()` 这样参数列表中涉及通配符的方法。

`contains()` 和 `indexOf()` 的参数类型是 **Object**，不涉及通配符，所以编译器允许调用它们。这意味着将由泛型类的设计者来决定哪些调用是“安全的”，并使用 **Object** 类作为它们的参数类型。为了禁止对类型中使用了通配符的方法调用，需要在参数列表中使用类型参数。

下面展示一个简单的 **Holder** 类：

```
public class Holder<T> {

    private T value;

    public Holder() {}

    public Holder(T val) {
        value = val;
    }

    public void set(T val) {
        value = val;
    }

    public T get() {
        return value;
    }

    @Override
    public boolean equals(Object o) {
        return o instanceof Holder && Objects.equals(value, ((Holder) o
            ).value);
    }

    @Override
    public int hashCode() {
        return Objects.hashCode(value);
    }

    public static void main(String[] args) {
        Holder<Apple> apple = new Holder<>(new Apple());
        Apple d = apple.get();
        apple.set(d);
//        Holder<Fruit> fruit = apple; // Cannot upcast
        Holder<? extends Fruit> fruit = apple; // OK
```

```
        Fruit p = fruit.get();
        d = (Apple) fruit.get();
        try {
            Orange c = (Orange) fruit.get(); // No warning
        } catch (Exception e) {
            System.out.println(e);
        }
//        fruit.set(new Apple()); // Cannot call set()
//        fruit.set(new Fruit()); // Cannot call set()
        System.out.println(fruit.equals(d)); // OK
    }
}
/* Output
java.lang.ClassCastException: Apple cannot be cast to Orange
false
*/
```

Holder 有一个接受 **T** 类型对象的 `set()` 方法，一个返回 T 对象的 `get()` 方法和一个接受 Object 对象的 `equals()` 方法。正如你所见，如果创建了一个 `Holder<Apple>`，就不能将其向上转型为 `Holder<Fruit>`，但是可以向上转型为 `Holder<? extends Fruit>`。如果调用 `get()`，只能返回一个 **Fruit**——这就是在给定"任何；额扩展自 **Fruit** 的对象"这一边界后，它所能知道的一切了。如果你知道更多的信息，就可以将其转型到某种具体的 **Fruit** 而不会导致任何警告，但是存在得到 **ClassCastException** 的风险。`set()` 方法不能工作在 **Apple** 和 **Fruit** 上，因为 `set()` 的参数也是"**? extends Fruit**"，意味着它可以是任何事物，编译器无法验证"任何事物"的类型安全性。

但是，`equals()` 方法可以正常工作，因为它接受的参数是 **Object** 而不是 **T** 类型。因此，编译器只关注传递进来和要返回的对象类型。它不会分析代码，以查看是否执行了任何实际的写入和读取操作。

Java 7 引入了 **java.util.Objects** 库，使创建 `equals()` 和 `hashCode()` 方法变得更加容易，当然还有很多其他功能。`equals()` 方法的标准形式参考 附录：理解 equals 和 hashCode 方法 一章。

21.8.2 逆变

还可以走另外一条路，即使用超类型通配符。这里，可以声明通配符是由某个特定类的任何基类来界定的，方法是指定 `<? super MyClass>`，或者甚至使用类型参数：`<? super T>`（尽管你不能对泛型参数给出一个超类型边界；即不能声明 `<T super MyClass>`）。这使得你可以安全地传递一个类型对象到泛型类型中。因此，有了超类型通配符，就可以向 **Collection** 写入了：

```
// generics/SuperTypeWildcards.java
import java.util.*;
public class SuperTypeWildcards {
    static void writeTo(List<? super Apple> apples) {
        apples.add(new Apple());
        apples.add(new Jonathan());
```

```
        // apples.add(new Fruit()); // Error
    }
}
```

参数 **apples** 是 **Apple** 的某种基类型的 **List**，这样你就知道向其中添加 **Apple** 或 **Apple** 的子类型是安全的。但是因为 **Apple** 是下界，所以你知道向这样的 **List** 中添加 **Fruit** 是不安全的，因为这将使这个 **List** 敞开口子，从而可以向其中添加非 **Apple** 类型的对象，而这是违反静态类型安全的。下面的示例复习了一下逆变和通配符的的使用：

```java
// generics/GenericReading.java
import java.util.*;

public class GenericReading {
    static List<Apple> apples = Arrays.asList(new Apple());
    static List<Fruit> fruit = Arrays.asList(new Fruit());

    static <T> T readExact(List<T> list) {
        return list.get(0);
    }

    // A static method adapts to each call:
    static void f1() {
        Apple a = readExact(apples);
        Fruit f = readExact(fruit);
        f = readExact(apples);
    }

    // A class type is established
    // when the class is instantiated:
    static class Reader<T> {
        T readExact(List<T> list) {
            return list.get(0);
        }
    }

    static void f2() {
        Reader<Fruit> fruitReader = new Reader<>();
        Fruit f = fruitReader.readExact(fruit);
        //- Fruit a = fruitReader.readExact(apples);
        // error: incompatible types: List<Apple>
        // cannot be converted to List<Fruit>
    }

    static class CovariantReader<T> {
        T readCovariant(List<? extends T> list) {
            return list.get(0);
        }
    }

```

```
    static void f3() {
        CovariantReader<Fruit> fruitReader = new CovariantReader<>();
        Fruit f = fruitReader.readCovariant(fruit);
        Fruit a = fruitReader.readCovariant(apples);
    }

    public static void main(String[] args) {
        f1();
        f2();
        f3();
    }
}
```

readExact() 方法使用了精确的类型。如果使用这个没有任何通配符的精确类型，就可以向 **List** 中写入和读取这个精确类型。另外，对于返回值，静态的泛型方法 readExact() 可以有效地“适应”每个方法调用，并能够从 List<Apple> 中返回一个 **Apple** ，从 List<Fruit> 中返回一个 **Fruit** ，就像在 f1() 中看到的那样。因此，如果可以摆脱静态泛型方法，那么在读取时就不需要协变类型了。然而对于泛型类来说，当你创建这个类的实例时，就要为这个类确定参数。就像在 f2() 中看到的，**fruitReader** 实例可以从 List<Fruit> 中读取一个 **Fruit** ，因为这就是它的确切类型。但是 List<Apple> 也应该产生 **Fruit** 对象，而 **fruitReader** 不允许这么做。为了修正这个问题，CovariantReader.readCovariant() 方法将接受 List<? extends T> ，因此，从这个列表中读取一个 **T** 是安全的（你知道在这个列表中的所有对象至少是一个 **T** ，并且可能是从 T 导出的某种对象）。在 f3() 中，你可以看到现在可以从 List<Apple> 中读取 **Fruit** 了。

21.8.3 无界通配符

无界通配符 <?> 看起来意味着“任何事物”，因此使用无界通配符好像等价于使用原生类型。事实上，编译器初看起来是支持这种判断的：

```
// generics/UnboundedWildcards1.java
import java.util.*;

public class UnboundedWildcards1 {
    static List list1;
    static List<?> list2;
    static List<? extends Object> list3;

    static void assign1(List list) {
        list1 = list;
        list2 = list;
        //- list3 = list;
        // warning: [unchecked] unchecked conversion
        // list3 = list;
        //         ^
        // required: List<? extends Object>
```

```
        // found:    List
    }

    static void assign2(List<?> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }

    static void assign3(List<? extends Object> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }

    public static void main(String[] args) {
        assign1(new ArrayList());
        assign2(new ArrayList());
        //- assign3(new ArrayList());
        // warning: [unchecked] unchecked method invocation:
        // method assign3 in class UnboundedWildcards1
        // is applied to given types
        // assign3(new ArrayList());
        //        ^
        // required: List<? extends Object>
        // found: ArrayList
        // warning: [unchecked] unchecked conversion
        // assign3(new ArrayList());
        //         ^
        // required: List<? extends Object>
        // found:    ArrayList
        // 2 warnings
        assign1(new ArrayList<>());
        assign2(new ArrayList<>());
        assign3(new ArrayList<>());
        // Both forms are acceptable as List<?>:
        List<?> wildList = new ArrayList();
        wildList = new ArrayList<>();
        assign1(wildList);
        assign2(wildList);
        assign3(wildList);
    }
}
```

有很多情况都和你在这里看到的情况类似，即编译器很少关心使用的是原生类型还是 `<?>` 。在这些情况中，`<?>` 可以被认为是一种装饰，但是它仍旧是很有价值的，因为，实际上它是在声明：“我是想用 Java 的泛型来编写这段代码，我在这里并不是要用原生类型，但是在当前这种情况下，泛型参数可以持有任何类型。” 第二个示例展示了无界通配符的一个重要应用。当你在处理多个泛型参数时，有时允许一个参数可以是任何类型，同时为其他参数确定某种特定类型的这种能力会显得很重要：

```java
// generics/UnboundedWildcards2.java
import java.util.*;

public class UnboundedWildcards2 {
    static Map map1;
    static Map<?,?> map2;
    static Map<String,?> map3;

    static void assign1(Map map) {
        map1 = map;
    }

    static void assign2(Map<?,?> map) {
        map2 = map;
    }

    static void assign3(Map<String,?> map) {
        map3 = map;
    }

    public static void main(String[] args) {
        assign1(new HashMap());
        assign2(new HashMap());
        //- assign3(new HashMap());
        // warning: [unchecked] unchecked method invocation:
        // method assign3 in class UnboundedWildcards2
        // is applied to given types
        //     assign3(new HashMap());
        //             ^
        //   required: Map<String,?>
        //   found: HashMap
        // warning: [unchecked] unchecked conversion
        //     assign3(new HashMap());
        //             ^
        //   required: Map<String,?>
        //   found:    HashMap
        // 2 warnings
        assign1(new HashMap<>());
        assign2(new HashMap<>());
        assign3(new HashMap<>());
    }
}
```

但是，当你拥有的全都是无界通配符时，就像在 `Map<?,?>` 中看到的那样，编译器看起来就无法将其与原生 **Map** 区分开了。另外，**UnboundedWildcards1.java** 展示了编译器处理 `List<?>` 和 `List<? extends Object>` 是不同的。令人困惑的是，编译器并非总是关注像 `List` 和 `List<?>` 之间的这种差异，因此它们看起来就像是相同的事物。事实上，因为泛型参数擦除到它的第一个边界，因此 `List<?>` 看起来等价于 `List<Object>` ，而 **List** 实际上也是 `List<Object>` ——除非这些语句

都不为真。**List** 实际上表示“持有任何 **Object** 类型的原生 **List** ”，而 `List<?>` 表示“具有某种特定类型的非原生 **List** ，只是我们不知道类型是什么。” 编译器何时才会关注原生类型和涉及无界通配符的类型之间的差异呢？下面的示例使用了前面定义的 `Holder<T>` 类，它包含接受 **Holder** 作为参数的各种方法，但是它们具有不同的形式：作为原生类型，具有具体的类型参数以及具有无界通配符参数：

```
// generics/Wildcards.java
// Exploring the meaning of wildcards

public class Wildcards {
    // Raw argument:
    static void rawArgs(Holder holder, Object arg) {
        //- holder.set(arg);
        // warning: [unchecked] unchecked call to set(T)
        // as a member of the raw type Holder
        //     holder.set(arg);
        //                ^
        //   where T is a type-variable:
        //     T extends Object declared in class Holder
        // 1 warning

        // Can't do this; don't have any 'T':
        // T t = holder.get();

        // OK, but type information is lost:
        Object obj = holder.get();
    }

    // Like rawArgs(), but errors instead of warnings:
    static void unboundedArg(Holder<?> holder, Object arg) {
        //- holder.set(arg);
        // error: method set in class Holder<T>
        // cannot be applied to given types;
        //     holder.set(arg);
        //           ^
        //   required: CAP#1
        //   found: Object
        //   reason: argument mismatch;
        //     Object cannot be converted to CAP#1
        //   where T is a type-variable:
        //     T extends Object declared in class Holder
        //   where CAP#1 is a fresh type-variable:
        //     CAP#1 extends Object from capture of ?
        // 1 error

        // Can't do this; don't have any 'T':
        // T t = holder.get();

        // OK, but type information is lost:
        Object obj = holder.get();
```

```
    }

    static <T> T exact1(Holder<T> holder) {
        return holder.get();
    }

    static <T> T exact2(Holder<T> holder, T arg) {
        holder.set(arg);
        return holder.get();
    }

    static <T> T wildSubtype(Holder<? extends T> holder, T arg) {
        //- holder.set(arg);
        // error: method set in class Holder<T#2>
        // cannot be applied to given types;
        //     holder.set(arg);
        //           ^
        //   required: CAP#1
        //   found: T#1
        //   reason: argument mismatch;
        //     T#1 cannot be converted to CAP#1
        //   where T#1,T#2 are type-variables:
        //     T#1 extends Object declared in method
        //     <T#1>wildSubtype(Holder<? extends T#1>,T#1)
        //     T#2 extends Object declared in class Holder
        //   where CAP#1 is a fresh type-variable:
        //     CAP#1 extends T#1 from
        //       capture of ? extends T#1
        // 1 error
        return holder.get();
    }

    static <T> void wildSupertype(Holder<? super T> holder, T arg) {
        holder.set(arg);
        //- T t = holder.get();
        // error: incompatible types:
        // CAP#1 cannot be converted to T
        //     T t = holder.get();
        //                     ^
        //   where T is a type-variable:
        //     T extends Object declared in method
        //       <T>wildSupertype(Holder<? super T>,T)
        //   where CAP#1 is a fresh type-variable:
        //     CAP#1 extends Object super:
        //       T from capture of ? super T
        // 1 error

        // OK, but type information is lost:
        Object obj = holder.get();
    }

```

```
    public static void main(String[] args) {
        Holder raw = new Holder<>();
        // Or:
        raw = new Holder();
        Holder<Long> qualified = new Holder<>();
        Holder<?> unbounded = new Holder<>();
        Holder<? extends Long> bounded = new Holder<>();
        Long lng = 1L;

        rawArgs(raw, lng);
        rawArgs(qualified, lng);
        rawArgs(unbounded, lng);
        rawArgs(bounded, lng);

        unboundedArg(raw, lng);
        unboundedArg(qualified, lng);
        unboundedArg(unbounded, lng);
        unboundedArg(bounded, lng);

        //- Object r1 = exact1(raw);
        // warning: [unchecked] unchecked method invocation:
        // method exact1 in class Wildcards is applied
        // to given types
        //      Object r1 = exact1(raw);
        //                        ^
        //   required: Holder<T>
        //   found: Holder
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>exact1(Holder<T>)
        // warning: [unchecked] unchecked conversion
        //      Object r1 = exact1(raw);
        //                         ^
        //   required: Holder<T>
        //   found:    Holder
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>exact1(Holder<T>)
        // 2 warnings

        Long r2 = exact1(qualified);
        Object r3 = exact1(unbounded); // Must return Object
        Long r4 = exact1(bounded);

        //- Long r5 = exact2(raw, lng);
        // warning: [unchecked] unchecked method invocation:
        // method exact2 in class Wildcards is
        // applied to given types
        //     Long r5 = exact2(raw, lng);
        //                     ^
        //   required: Holder<T>,T
```

```
        //   found: Holder,Long
        //   where T is a type-variable:
        //     T extends Object declared in
        //       method <T>exact2(Holder<T>,T)
        // warning: [unchecked] unchecked conversion
        //     Long r5 = exact2(raw, lng);
        //                      ^
        //   required: Holder<T>
        //   found:    Holder
        //   where T is a type-variable:
        //     T extends Object declared in
        //       method <T>exact2(Holder<T>,T)
        // 2 warnings

        Long r6 = exact2(qualified, lng);

        //- Long r7 = exact2(unbounded, lng);
        // error: method exact2 in class Wildcards
        // cannot be applied to given types;
        //     Long r7 = exact2(unbounded, lng);
        //               ^
        //   required: Holder<T>,T
        //   found: Holder<CAP#1>,Long
        //   reason: inference variable T has
        //     incompatible bounds
        //     equality constraints: CAP#1
        //     lower bounds: Long
        //   where T is a type-variable:
        //     T extends Object declared in
        //       method <T>exact2(Holder<T>,T)
        //   where CAP#1 is a fresh type-variable:
        //     CAP#1 extends Object from capture of ?
        // 1 error

        //- Long r8 = exact2(bounded, lng);
        // error: method exact2 in class Wildcards
        // cannot be applied to given types;
        //      Long r8 = exact2(bounded, lng);
        //                ^
        //   required: Holder<T>,T
        //   found: Holder<CAP#1>,Long
        //   reason: inference variable T
        //     has incompatible bounds
        //     equality constraints: CAP#1
        //     lower bounds: Long
        //   where T is a type-variable:
        //     T extends Object declared in
        //       method <T>exact2(Holder<T>,T)
        //   where CAP#1 is a fresh type-variable:
        //     CAP#1 extends Long from
        //       capture of ? extends Long
```

```
        // 1 error

        //- Long r9 = wildSubtype(raw, lng);
        // warning: [unchecked] unchecked method invocation:
        // method wildSubtype in class Wildcards
        // is applied to given types
        //     Long r9 = wildSubtype(raw, lng);
        //                           ^
        //   required: Holder<? extends T>,T
        //   found: Holder,Long
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>wildSubtype(Holder<? extends T>,T)
        // warning: [unchecked] unchecked conversion
        //     Long r9 = wildSubtype(raw, lng);
        //                            ^
        //   required: Holder<? extends T>
        //   found:    Holder
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>wildSubtype(Holder<? extends T>,T)
        // 2 warnings

        Long r10 = wildSubtype(qualified, lng);
        // OK, but can only return Object:
        Object r11 = wildSubtype(unbounded, lng);
        Long r12 = wildSubtype(bounded, lng);

        //- wildSupertype(raw, lng);
        // warning: [unchecked] unchecked method invocation:
        //   method wildSupertype in class Wildcards
        //   is applied to given types
        //     wildSupertype(raw, lng);
        //                   ^
        //   required: Holder<? super T>,T
        //   found: Holder,Long
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>wildSupertype(Holder<? super T>,T)
        // warning: [unchecked] unchecked conversion
        //     wildSupertype(raw, lng);
        //                    ^
        //   required: Holder<? super T>
        //   found:    Holder
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>wildSupertype(Holder<? super T>,T)
        // 2 warnings

        wildSupertype(qualified, lng);

```

```
        //- wildSupertype(unbounded, lng);
        // error: method wildSupertype in class Wildcards
        // cannot be applied to given types;
        //     wildSupertype(unbounded, lng);
        //     ^
        //   required: Holder<? super T>,T
        //   found: Holder<CAP#1>,Long
        //   reason: cannot infer type-variable(s) T
        //     (argument mismatch; Holder<CAP#1>
        //     cannot be converted to Holder<? super T>)
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>wildSupertype(Holder<? super T>,T)
        //   where CAP#1 is a fresh type-variable:
        //     CAP#1 extends Object from capture of ?
        // 1 error

        //- wildSupertype(bounded, lng);
        // error: method wildSupertype in class Wildcards
        // cannot be applied to given types;
        //     wildSupertype(bounded, lng);
        //     ^
        //   required: Holder<? super T>,T
        //   found: Holder<CAP#1>,Long
        //   reason: cannot infer type-variable(s) T
        //     (argument mismatch; Holder<CAP#1>
        //     cannot be converted to Holder<? super T>)
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>wildSupertype(Holder<? super T>,T)
        //   where CAP#1 is a fresh type-variable:
        //     CAP#1 extends Long from capture of
        //     ? extends Long
        // 1 error
    }
}
```

在 `rawArgs()` 中，编译器知道 `Holder` 是一个泛型类型，因此即使它在这里被表示成一个原生类型，编译器仍旧知道向 `set()` 传递一个 **Object** 是不安全的。由于它是原生类型，你可以将任何类型的对象传递给 `set()` ，而这个对象将被向上转型为 **Object** 。因此无论何时，只要使用了原生类型，都会放弃编译期检查。对 `get()` 的调用说明了相同的问题：没有任何 **T** 类型的对象，因此结果只能是一个 **Object**。人们很自然地会开始考虑原生 `Holder` 与 `Holder<?>` 是大致相同的事物。但是 `unboundedArg()` 强调它们是不同的——它揭示了相同的问题，但是它将这些问题作为错误而不是警告报告，因为原生 **Holder** 将持有任何类型的组合，而 `Holder<?>` 将持有具有某种具体类型的同构集合，因此不能只是向其中传递 **Object** 。在 `exact1()` 和 `exact2()` 中，你可以看到使用了确切的泛型参数——没有任何通配符。你将看到，`exact2()`与 `exact1()` 具有不同的限制，因为它有额外的参数。在 `wildSubtype()` 中，在 **Holder** 类型上的限制被放松为包括持有任何扩展自 **T** 的对象的 **Holder**

。这还是意味着如果 T 是 **Fruit** ，那么 `holder` 可以是 `Holder<Apple>` ，这是合法的。为了防止将 **Orange** 放置到 `Holder<Apple>` 中，对 `set()` 的调用（或者对任何接受这个类型参数为参数的方法的调用）都是不允许的。但是，你仍旧知道任何来自 `Holder<? extends Fruit>` 的对象至少是 **Fruit** ，因此 `get()` （或者任何将产生具有这个类型参数的返回值的方法）都是允许的。`wildSupertype()` 展示了超类型通配符，这个方法展示了与 `wildSubtype()` 相反的行为：`holder` 可以是持有任何 T 的基类型的容器。因此，`set()` 可以接受 **T** ，因为任何可以工作于基类的对象都可以多态地作用于导出类（这里就是 **T** ）。但是，尝试着调用 `get()` 是没有用的，因为由 `holder` 持有的类型可以是任何超类型，因此唯一安全的类型就是 **Object** 。这个示例还展示了对于在 `unbounded()` 中使用无界通配符能够做什么不能做什么所做出的限制：因为你没有 **T**，所以你不能将 `set()` 或 `get()` 作用于 **T** 上。

在 `main()` 方法中你看到了某些方法在接受某些类型的参数时没有报错和警告。为了迁移兼容性，`rawArgs()` 将接受所有 **Holder** 的不同变体，而不会产生警告。`unboundedArg()` 方法也可以接受相同的所有类型，尽管如前所述，它在方法体内部处理这些类型的方式并不相同。

如果向接受“确切”泛型类型（没有通配符）的方法传递一个原生 **Holder** 引用，就会得到一个警告，因为确切的参数期望得到在原生类型中并不存在的信息。如果向 `exact1()` 传递一个无界引用，就不会有任何可以确定返回类型的类型信息。可以看到，`exact2()` 具有最多的限制，因为它希望精确地得到一个 `Holder<T>` ，以及一个具有类型 **T** 的参数，正由于此，它将产生错误或警告，除非提供确切的参数。有时，这样做很好，但是如果它过于受限，那么就可以使用通配符，这取决于是否想要从泛型参数中返回类型确定的返回值（就像在 `wildSubtype()` 中看到的那样），或者是否想要向泛型参数传递类型确定的参数（就像在 `wildSupertype()` 中看到的那样）。因此，使用确切类型来替代通配符类型的好处是，可以用泛型参数来做更多的事，但是使用通配符使得你必须接受范围更宽的参数化类型作为参数。因此，必须逐个情况地权衡利弊，找到更适合你的需求的方法。

21.8.4 捕获转换

有一种特殊情况需要使用 `<?>` 而不是原生类型。如果向一个使用 `<?>` 的方法传递原生类型，那么对编译器来说，可能会推断出实际的类型参数，使得这个方法可以回转并调用另一个使用这个确切类型的方法。下面的示例演示了这种技术，它被称为捕获转换，因为未指定的通配符类型被捕获，并被转换为确切类型。这里，有关警告的注释只有在 `@SuppressWarnings` 注解被移除之后才能起作用：

```
// generics/CaptureConversion.java

public class CaptureConversion {
    static <T> void f1(Holder<T> holder) {
        T t = holder.get();
        System.out.println(t.getClass().getSimpleName());
    }

    static void f2(Holder<?> holder) {
        f1(holder); // Call with captured type
```

```
    }

    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        Holder raw = new Holder<>(1);
        f1(raw);
        // warning: [unchecked] unchecked method invocation:
        // method f1 in class CaptureConversion
        // is applied to given types
        //     f1(raw);
        //       ^
        //   required: Holder<T>
        //   found: Holder
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>f1(Holder<T>)
        // warning: [unchecked] unchecked conversion
        //     f1(raw);
        //        ^
        //   required: Holder<T>
        //   found:    Holder
        //   where T is a type-variable:
        //     T extends Object declared in
        //     method <T>f1(Holder<T>)
        // 2 warnings
        f2(raw); // No warnings

        Holder rawBasic = new Holder();
        rawBasic.set(new Object());
        // warning: [unchecked] unchecked call to set(T)
        // as a member of the raw type Holder
        //     rawBasic.set(new Object());
        //                  ^
        //   where T is a type-variable:
        //     T extends Object declared in class Holder
        // 1 warning
        f2(rawBasic); // No warnings

        // Upcast to Holder<?>, still figures it out:
        Holder<?> wildcarded = new Holder<>(1.0);
        f2(wildcarded);
    }
}
/* Output:
Integer
Integer
Object
Double
*/
```

`f1()` 中的类型参数都是确切的，没有通配符或边界。在 `f2()` 中，**Holder** 参数是一个无界通配

符，因此它看起来是未知的。但是，在 f2() 中调用了 f1()，而 f1() 需要一个已知参数。这里所发生的是：在调用 f2() 的过程中捕获了参数类型，并在调用 f1() 时使用了这种类型。你可能想知道这项技术是否可以用于写入，但是这要求在传递 Holder<?> 时同时传递一个具体类型。捕获转换只有在这样的情况下可以工作：即在方法内部，你需要使用确切的类型。注意，不能从 f2() 中返回 **T**，因为 **T** 对于 f2() 来说是未知的。捕获转换十分有趣，但是非常受限。

21.9 问题

本节将阐述在使用 Java 泛型时会出现的各类问题。

21.9.1 任何基本类型都不能作为类型参数

正如本章早先提到的，Java 泛型的限制之一是不能将基本类型用作类型参数。因此，不能创建 ArrayList<int> 之类的东西。解决方法是使用基本类型的包装器类以及自动装箱机制。如果创建一个 ArrayList<Integer>，并将基本类型 **int** 应用于这个集合，那么你将发现自动装箱机制将自动地实现 **int** 到 **Integer** 的双向转换——因此，这几乎就像是有一个 ArrayList<int> 一样：

```
// generics/ListOfInt.java
// Autoboxing compensates for the inability
// to use primitives in generics
import java.util.*;
import java.util.stream.*;

public class ListOfInt {
    public static void main(String[] args) {
        List<Integer> li = IntStream.range(38, 48)
            .boxed() // Converts ints to Integers
            .collect(Collectors.toList());
        System.out.println(li);
    }
}
/* Output:
[38, 39, 40, 41, 42, 43, 44, 45, 46, 47]
*/
```

通常，这种解决方案工作得很好——能够成功地存储和读取 **int**，自动装箱隐藏了转换的过程。但是如果性能成为问题的话，就需要使用专门为基本类型适配的特殊版本的集合；一个开源版本的实现是 **org.apache.commons.collections.primitives**。下面是另外一种方式，它可以创建持有 **Byte** 的 **Set**：

```
// generics/ByteSet.java
import java.util.*;

```

```
public class ByteSet {
    Byte[] possibles = { 1,2,3,4,5,6,7,8,9 };
    Set<Byte> mySet = new HashSet<>(Arrays.asList(possibles));
    // But you can't do this:
    // Set<Byte> mySet2 = new HashSet<>(
    // Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
}
```

自动装箱机制解决了一些问题，但并没有解决所有问题。

在下面的示例中，**FillArray** 接口包含一些通用方法，这些方法使用 **Supplier** 来用对象填充数组（这使得类泛型在本例中无法工作，因为这个方法是静态的）。**Supplier** 实现来自 数组 一章，并且在 `main()` 中，可以看到 `FillArray.fill()` 使用对象填充了数组：

```
// generics/PrimitiveGenericTest.java
import onjava.*;
import java.util.*;
import java.util.function.*;

// Fill an array using a generator:
interface FillArray {
    static <T> T[] fill(T[] a, Supplier<T> gen) {
        Arrays.setAll(a, n -> gen.get());
        return a;
    }

    static int[] fill(int[] a, IntSupplier gen) {
        Arrays.setAll(a, n -> gen.getAsInt());
        return a;
    }

    static long[] fill(long[] a, LongSupplier gen) {
        Arrays.setAll(a, n -> gen.getAsLong());
        return a;
    }

    static double[] fill(double[] a, DoubleSupplier gen) {
        Arrays.setAll(a, n -> gen.getAsDouble());
        return a;
    }
}

public class PrimitiveGenericTest {
    public static void main(String[] args) {
        String[] strings = FillArray.fill(
            new String[5], new Rand.String(9));
        System.out.println(Arrays.toString(strings));
        int[] integers = FillArray.fill(
            new int[9], new Rand.Pint());
        System.out.println(Arrays.toString(integers));
```

```
    }
}
/* Output:
[btpenpccu, xszgvgmei, nneeloztd, vewcippcy, gpoalkljl]
[635, 8737, 3941, 4720, 6177, 8479, 6656, 3768, 4948]
*/
```

自动装箱不适用于数组，因此我们必须创建 `FillArray.fill()` 的重载版本，或创建产生 **Wrapped** 输出的生成器。**FillArray** 仅比 `java.util.Arrays.setAll()` 有用一点，因为它返回填充的数组。

21.9.2 实现参数化接口

一个类不能实现同一个泛型接口的两种变体，由于擦除的原因，这两个变体会成为相同的接口。下面是产生这种冲突的情况：

```
// generics/MultipleInterfaceVariants.java
// {WillNotCompile}
package generics;

interface Payable<T> {}

class Employee implements Payable<Employee> {}

class Hourly extends Employee implements Payable<Hourly> {}
```

Hourly 不能编译，因为擦除会将 `Payable<Employe>` 和 `Payable<Hourly>` 简化为相同的类 **Payable**，这样，上面的代码就意味着在重复两次地实现相同的接口。十分有趣的是，如果从 **Payable** 的两种用法中都移除掉泛型参数（就像编译器在擦除阶段所做的那样）这段代码就可以编译。

在使用某些更基本的 Java 接口，例如 `Comparable<T>` 时，这个问题可能会变得十分令人恼火，就像你在本节稍后看到的那样。

21.9.3 转型和警告

使用带有泛型类型参数的转型或 **instanceof** 不会有任何效果。下面的集合在内部将各个值存储为 **Object**，并在获取这些值时，再将它们转型回 **T**：

```
// generics/GenericCast.java
import java.util.*;
import java.util.stream.*;

class FixedSizeStack<T> {
    private final int size;
    private Object[] storage;
    private int index = 0;
```

```java

    FixedSizeStack(int size) {
        this.size = size;
        storage = new Object[size];
    }

    public void push(T item) {
        if(index < size)
            storage[index++] = item;
    }

    @SuppressWarnings("unchecked")
    public T pop() {
        return index == 0 ? null : (T)storage[--index];
    }

    @SuppressWarnings("unchecked")
    Stream<T> stream() {
        return (Stream<T>)Arrays.stream(storage);
    }
}

public class GenericCast {
    static String[] letters = "ABCDEFGHIJKLMNOPQRS".split("");

    public static void main(String[] args) {
        FixedSizeStack<String> strings =
            new FixedSizeStack<>(letters.length);
        Arrays.stream("ABCDEFGHIJKLMNOPQRS".split(""))
            .forEach(strings::push);
        System.out.println(strings.pop());
        strings.stream()
            .map(s -> s + " ")
            .forEach(System.out::print);
    }
}
/* Output:
S
A B C D E F G H I J K L M N O P Q R S
*/
```

如果没有 **@SuppressWarnings** 注解，编译器将对 `pop()` 产生 “unchecked cast” 警告。由于擦除的原因，编译器无法知道这个转型是否是安全的，并且 `pop()` 方法实际上并没有执行任何转型。这是因为，**T** 被擦除到它的第一个边界，默认情况下是 **Object** ，因此 `pop()` 实际上只是将 **Object** 转型为 **Object**。有时，泛型没有消除对转型的需要，这就会由编译器产生警告，而这个警告是不恰当的。例如：

```java
// generics/NeedCasting.java
import java.io.*;
```

代码纠错要比新编写代码困难一倍。因为，如果你写出了最聪明的代码，按此推算，你将没有更大的智慧来 debug 它。

```
import java.util.*;

public class NeedCasting {
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(args[0]));
        List<Widget> shapes = (List<Widget>)in.readObject();
    }
}
```

正如你将在 附录：对象序列化 中学到的那样，`readObject()` 无法知道它正在读取的是什么，因此它返回的是必须转型的对象。但是当注释掉 **@SuppressWarnings** 注解并编译这个程序时，就会得到下面的警告。

```
NeedCasting.java uses unchecked or unsafe operations.
Recompile with -Xlint:unchecked for details.

And if you follow the instructions and recompile with  -
Xlint:unchecked :(如果遵循这条指示，使用-Xlint:unchecked来重新编译：)

NeedCasting.java:10: warning: [unchecked] unchecked cast
    List<Widget> shapes = (List<Widget>)in.readObject();
    required: List<Widget>
    found: Object
1 warning
```

你会被强制要求转型，但是又被告知不应该转型。为了解决这个问题，必须使用 Java 5 引入的新的转型形式，既通过泛型类来转型：

```
// generics/ClassCasting.java
import java.io.*;
import java.util.*;

public class ClassCasting {
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(args[0]));
        // Won't Compile:
        //    List<Widget> lw1 =
        //    List<>.class.cast(in.readObject());
        List<Widget> lw2 = List.class.cast(in.readObject());
    }
}
```

但是，不能转型到实际类型（`List<Widget>`）。也就是说，不能声明：

```
List<Widget>.class.cast(in.readobject())
```

甚至当你添加一个像下面这样的另一个转型时：

```
(List<Widget>)List.class.cast(in.readobject())
```

仍旧会得到一个警告。

21.9.4 重载

下面的程序是不能编译的，即使它看起来是合理的：

```
// generics/UseList.java
// {WillNotCompile}
import java.util.*;

public class UseList<W, T> {
    void f(List<T> v) {}
    void f(List<W> v) {}
}
```

因为擦除，所以重载方法产生了的类型签名。

因而，当擦除后的参数不能产生唯一的参数列表时，你必须提供不同的方法名：

```
// generics/UseList2.java

import java.util.*;

public class UseList2<W, T> {
    void f1(List<T> v) {}
    void f2(List<W> v) {}
}
```

幸运的是，编译器可以检测到这类问题。

21.9.5 基类劫持接口

假设你有一个实现了 **Comparable** 接口的 **Pet** 类：

```
// generics/ComparablePet.java

public class ComparablePet implements Comparable<ComparablePet> {
    @Override
    public int compareTo(ComparablePet o) {
        return 0;
    }
}
```

尝试缩小 **ComparablePet** 子类的比较类型是有意义的。例如，**Cat** 类可以与其他的 **Cat** 比较：

```java
// generics/HijackedInterface.java
// {WillNotCompile}

class Cat extends ComparablePet implements Comparable<Cat> {
    // error: Comparable cannot be inherited with
    // different arguments: <Cat> and <ComparablePet>
    // class Cat
    // ^
    // 1 error
    public int compareTo(Cat arg) {
        return 0;
    }
}
```

不幸的是，这不能工作。一旦 **Comparable** 的类型参数设置为 **ComparablePet**，其他的实现类只能比较 **ComparablePet**:

```java
// generics/RestrictedComparablePets.java

public class Hamster extends ComparablePet implements Comparable<
    ComparablePet> {

    @Override
    public int compareTo(ComparablePet arg) {
        return 0;
    }
}
// Or just:
class Gecko extends ComparablePet {
    public int compareTo(ComparablePet arg) {
        return 0;
    }
}
```

Hamster 显示了重新实现 **ComparableSet** 中相同的接口是可能的，只要接口完全相同，包括参数类型。然而正如 **Gecko** 中所示，这与直接覆写基类的方法完全相同。

21.10 自限定的类型

在 Java 泛型中，有一个似乎经常性出现的惯用法，它相当令人费解:

```java
class SelfBounded<T extends SelfBounded<T>> { // ...
```

这就像两面镜子彼此照向对方所引起的目眩效果一样，是一种无限反射。**SelfBounded** 类接受泛型参数 **T**，而 **T** 由一个边界类限定，这个边界就是拥有 **T** 作为其参数的 **SelfBounded**。

当你首次看到它时，很难去解析它，它强调的是当 **extends** 关键字用于边界与用来创建子类明显是不同的。

21.10.1 古怪的循环泛型

为了理解自限定类型的含义，我们从这个惯用法的一个简单版本入手，它没有自限定的边界。

不能直接继承一个泛型参数，但是，可以继承在其自己的定义中使用这个泛型参数的类。也就是说，可以声明：

```
// generics/CuriouslyRecurringGeneric.java

class GenericType<T> {}

public class CuriouslyRecurringGeneric
  extends GenericType<CuriouslyRecurringGeneric> {}
```

这可以按照 Jim Coplien 在 C++ 中的*古怪的循环模版模式*的命名方式，称为古怪的循环泛型（CRG）。“古怪的循环”是指类相当古怪地出现在它自己的基类中这一事实。为了理解其含义，努力大声说：“我在创建一个新类，它继承自一个泛型类型，这个泛型类型接受我的类的名字作为其参数。”当给出导出类的名字时，这个泛型基类能够实现什么呢？好吧，Java 中的泛型关乎参数和返回类型，因此它能够产生使用导出类作为其参数和返回类型的基类。它还能将导出类型用作其域类型，尽管这些将被擦除为 **Object** 的类型。下面是表示了这种情况的一个泛型类：

```
// generics/BasicHolder.java

public class BasicHolder<T> {
    T element;
    void set(T arg) { element = arg; }
    T get() { return element; }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
}
```

这是一个普通的泛型类型，它的一些方法将接受和产生具有其参数类型的对象，还有一个方法在其存储的域上执行操作（尽管只是在这个域上执行 **Object** 操作）。我们可以在一个古怪的循环泛型中使用 **BasicHolder**：

```
// generics/CRGWithBasicHolder.java

class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
    public static void main(String[] args) {
        Subtype st1 = new Subtype(), st2 = new Subtype();
        st1.set(st2);
        Subtype st3 = st1.get();
        st1.f();
    }
}
```

```
/* Output:
Subtype
*/
```

注意，这里有些东西很重要：新类 **Subtype** 接受的参数和返回的值具有 **Subtype** 类型而不仅仅是基类 **BasicHolder** 类型。这就是 CRG 的本质：基类用导出类替代其参数。这意味着泛型基类变成了一种其所有导出类的公共功能的模版，但是这些功能对于其所有参数和返回值，将使用导出类型。也就是说，在所产生的类中将使用确切类型而不是基类型。因此，在 **Subtype** 中，传递给 `set()` 的参数和从 `get()` 返回的类型都是确切的 **Subtype**。

21.10.2 自限定

BasicHolder 可以使用任何类型作为其泛型参数，就像下面看到的那样：

```
// generics/Unconstrained.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.

class Other {}
class BasicOther extends BasicHolder<Other> {}

public class Unconstrained {
    public static void main(String[] args) {
        BasicOther b = new BasicOther();
        BasicOther b2 = new BasicOther();
        b.set(new Other());
        Other other = b.get();
        b.f();
    }
}
/* Output:
Other
*/
```

限定将采取额外的步骤，强制泛型当作其自身的边界参数来使用。观察所产生的类可以如何使用以及不可以如何使用：

```
// generics/SelfBounding.java

class SelfBounded<T extends SelfBounded<T>> {
    T element;
    SelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
```

```
}

class A extends SelfBounded<A> {}
class B extends SelfBounded<A> {} // Also OK

class C extends SelfBounded<C> {
    C setAndGet(C arg) {
        set(arg);
        return get();
    }
}

class D {}
// Can't do this:
// class E extends SelfBounded<D> {}
// Compile error:
//   Type parameter D is not within its bound

// Alas, you can do this, so you cannot force the idiom:
class F extends SelfBounded {}

public class SelfBounding {
    public static void main(String[] args) {
        A a = new A();
        a.set(new A());
        a = a.set(new A()).get();
        a = a.get();
        C c = new C();
        c = c.setAndGet(new C());
    }
}
```

自限定所做的，就是要求在继承关系中，像下面这样使用这个类：

```
class A extends SelfBounded<A>{}
```

这会强制要求将正在定义的类当作参数传递给基类。

自限定的参数有何意义呢？它可以保证类型参数必须与正在被定义的类相同。正如你在 B 类的定义中所看到的，还可以从使用了另一个 **SelfBounded** 参数的 **SelfBounded** 中导出，尽管在 **A** 类看到的用法看起来是主要的用法。对定义 **E** 的尝试说明不能使用不是 **SelfBounded** 的类型参数。遗憾的是，**F** 可以编译，不会有任何警告，因此自限定惯用法不是可强制执行的。如果它确实很重要，可以要求一个外部工具来确保不会使用原生类型来替代参数化类型。注意，可以移除自限定这个限制，这样所有的类仍旧是可以编译的，但是 **E** 也会因此而变得可编译：

```
// generics/NotSelfBounded.java

public class NotSelfBounded<T> {
    T element;
```

```
    NotSelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A2 extends NotSelfBounded<A2> {}
class B2 extends NotSelfBounded<A2> {}

class C2 extends NotSelfBounded<C2> {
    C2 setAndGet(C2 arg) {
        set(arg);
        return get();
    }
}

class D2 {}
// Now this is OK:
class E2 extends NotSelfBounded<D2> {}
```

因此很明显，自限定限制只能强制作用于继承关系。如果使用自限定，就应该了解这个类所用的类型参数将与使用这个参数的类具有相同的基类型。这会强制要求使用这个类的每个人都要遵循这种形式。还可以将自限定用于泛型方法：

```
// generics/SelfBoundingMethods.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.

public class SelfBoundingMethods {
    static <T extends SelfBounded<T>> T f(T arg) {
        return arg.set(arg).get();
    }

    public static void main(String[] args) {
        A a = f(new A());
    }
}
```

这可以防止这个方法被应用于除上述形式的自限定参数之外的任何事物上。

21.10.3 参数协变

自限定类型的价值在于它们可以产生*协变参数类型*——方法参数类型会随子类而变化。

尽管自限定类型还可以产生与子类类型相同的返回类型，但是这并不十分重要，因为*协变返回类型*是在 Java 5 引入：

```java
// generics/CovariantReturnTypes.java

class Base {}
class Derived extends Base {}

interface OrdinaryGetter {
    Base get();
}

interface DerivedGetter extends OrdinaryGetter {
    // Overridden method return type can vary:
    @Override
    Derived get();
}

public class CovariantReturnTypes {
    void test(DerivedGetter d) {
        Derived d2 = d.get();
    }
}
```

DerivedGetter 中的 `get()` 方法覆盖了 **OrdinaryGetter** 中的 `get()`，并返回了一个从 `OrdinaryGetter.get()` 的返回类型中导出的类型。尽管这是完全合乎逻辑的事情（导出类方法应该能够返回比它覆盖的基类方法更具体的类型）但是这在早先的 Java 版本中是不合法的。

自限定泛型事实上将产生确切的导出类型作为其返回值，就像在 `get()` 中所看到的一样：

```java
// generics/GenericsAndReturnTypes.java

interface GenericGetter<T extends GenericGetter<T>> {
    T get();
}

interface Getter extends GenericGetter<Getter> {}

public class GenericsAndReturnTypes {
    void test(Getter g) {
        Getter result = g.get();
        GenericGetter gg = g.get(); // Also the base type
    }
}
```

注意，这段代码不能编译，除非是使用囊括了协变返回类型的 Java 5。

然而，在非泛型代码中，参数类型不能随子类型发生变化：

```java
// generics/OrdinaryArguments.java

class OrdinarySetter {
    void set(Base base) {
```

```
        System.out.println("OrdinarySetter.set(Base)");
    }
}

class DerivedSetter extends OrdinarySetter {
    void set(Derived derived) {
        System.out.println("DerivedSetter.set(Derived)");
    }
}

public class OrdinaryArguments {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
        DerivedSetter ds = new DerivedSetter();
        ds.set(derived);
        // Compiles--overloaded, not overridden!:
        ds.set(base);
    }
}
/* Output:
DerivedSetter.set(Derived)
OrdinarySetter.set(Base)
*/
```

set(derived) 和 set(base) 都是合法的，因此 DerivedSetter.set() 没有覆盖 OrdinarySetter.set()，而是重载了这个方法。从输出中可以看到，在 **DerivedSetter** 中有两个方法，因此基类版本仍旧是可用的，因此可以证明它被重载过。但是，在使用自限定类型时，在导出类中只有一个方法，并且这个方法接受导出类型而不是基类型为参数：

```
// generics/SelfBoundingAndCovariantArguments.java

interface SelfBoundSetter<T extends SelfBoundSetter<T>> {
    void set(T arg);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundingAndCovariantArguments {
    void
    testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
        s1.set(s2);
        //- s1.set(sbs);
        // error: method set in interface SelfBoundSetter<T>
        // cannot be applied to given types;
        //     s1.set(sbs);
        //       ^
        //   required: Setter
        //   found: SelfBoundSetter
```

```
        //   reason: argument mismatch;
        // SelfBoundSetter cannot be converted to Setter
        //   where T is a type-variable:
        //     T extends SelfBoundSetter<T> declared in
        //     interface SelfBoundSetter
        // 1 error
    }
}
```

编译器不能识别将基类型当作参数传递给 `set()` 的尝试，因为没有任何方法具有这样的签名。实际上，这个参数已经被覆盖。如果不使用自限定类型，普通的继承机制就会介入，而你将能够重载，就像在非泛型的情况下一样：

```
// generics/PlainGenericInheritance.java

class GenericSetter<T> { // Not self-bounded
    void set(T arg) {
        System.out.println("GenericSetter.set(Base)");
    }
}

class DerivedGS extends GenericSetter<Base> {
    void set(Derived derived) {
        System.out.println("DerivedGS.set(Derived)");
    }
}

public class PlainGenericInheritance {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
        DerivedGS dgs = new DerivedGS();
        dgs.set(derived);
        dgs.set(base); // Overloaded, not overridden!
    }
}
/* Output:
DerivedGS.set(Derived)
GenericSetter.set(Base)
*/
```

这段代码在模仿 **OrdinaryArguments.java**；在那个示例中，**DerivedSetter** 继承自包含一个 `set(Base)` 的 **OrdinarySetter** 。而这里，**DerivedGS** 继承自泛型创建的也包含有一个 `set(Base)`的 `GenericSetter<Base>`。就像 **OrdinaryArguments.java** 一样，你可以从输出中看到，**DerivedGS** 包含两个 `set()` 的重载版本。如果不使用自限定，将重载参数类型。如果使用了自限定，只能获得方法的一个版本，它将接受确切的参数类型。

21.11 动态类型安全

因为可以向 Java 5 之前的代码传递泛型集合，所以旧式代码仍旧有可能会破坏你的集合。Java 5 的 **java.util.Collections** 中有一组便利工具，可以解决在这种情况下的类型检查问题，它们是：静态方法 `checkedCollection()` 、`checkedList()`、`checkedMap()` 、`checkedSet()` 、`checkedSortedMap()`和 `checkedSortedSet()`。这些方法每一个都会将你希望动态检查的集合当作第一个参数接受，并将你希望强制要求的类型作为第二个参数接受。

受检查的集合在你试图插入类型不正确的对象时抛出 **ClassCastException** , 这与泛型之前的（原生）集合形成了对比，对于后者来说，当你将对象从集合中取出时，才会通知你出现了问题。在后一种情况中，你知道存在问题，但是不知道罪魁祸首在哪里，如果使用受检查的集合，就可以发现谁在试图插入不良对象。让我们用受检查的集合来看看“将猫插入到狗列表中”这个问题。这里，`oldStyleMethod()` 表示遗留代码，因为它接受的是原生的 **List** ，而 **@SuppressWarnings（“unchecked”）** 注解对于压制所产生的警告是必需的：

```
// generics/CheckedList.java
// Using Collection.checkedList()
import typeinfo.pets.*;
import java.util.*;

public class CheckedList {
    @SuppressWarnings("unchecked")
    static void oldStyleMethod(List probablyDogs) {
        probablyDogs.add(new Cat());
    }

    public static void main(String[] args) {
        List<Dog> dogs1 = new ArrayList<>();
        oldStyleMethod(dogs1); // Quietly accepts a Cat
        List<Dog> dogs2 = Collections.checkedList(
            new ArrayList<>(), Dog.class);
        try {
            oldStyleMethod(dogs2); // Throws an exception
        } catch(Exception e) {
            System.out.println("Expected: " + e);
        }
        // Derived types work fine:
        List<Pet> pets = Collections.checkedList(
            new ArrayList<>(), Pet.class);
        pets.add(new Dog());
        pets.add(new Cat());
    }
}
/* Output:
Expected: java.lang.ClassCastException: Attempt to
insert class typeinfo.pets.Cat element into collection
```

```
with element type class typeinfo.pets.Dog
*/
```

运行这个程序时，你会发现插入一个 **Cat** 对于 **dogs1** 来说没有任何问题，而 **dogs2** 立即会在这个错误类型的插入操作上抛出一个异常。还可以看到，将导出类型的对象放置到将要检查基类型的受检查容器中是没有问题的。

21.12 泛型异常

由于擦除的原因，**catch** 语句不能捕获泛型类型的异常，因为在编译期和运行时都必须知道异常的确切类型。泛型类也不能直接或间接继承自 **Throwable**（这将进一步阻止你去定义不能捕获的泛型异常）。但是，类型参数可能会在一个方法的 **throws** 子句中用到。这使得你可以编写随检查型异常类型变化的泛型代码：

```java
// generics/ThrowGenericException.java

import java.util.*;

interface Processor<T, E extends Exception> {
    void process(List<T> resultCollector) throws E;
}

class ProcessRunner<T, E extends Exception>
extends ArrayList<Processor<T, E>> {
    List<T> processAll() throws E {
        List<T> resultCollector = new ArrayList<>();
        for(Processor<T, E> processor : this)
            processor.process(resultCollector);
        return resultCollector;
    }
}

class Failure1 extends Exception {}

class Processor1
implements Processor<String, Failure1> {
    static int count = 3;
    @Override
    public void process(List<String> resultCollector)
    throws Failure1 {
        if(count-- > 1)
            resultCollector.add("Hep!");
        else
            resultCollector.add("Ho!");
        if(count < 0)
            throw new Failure1();
```

```
    }
}

class Failure2 extends Exception {}

class Processor2
implements Processor<Integer, Failure2> {
    static int count = 2;
    @Override
    public void process(List<Integer> resultCollector)
    throws Failure2 {
        if(count-- == 0)
            resultCollector.add(47);
        else {
            resultCollector.add(11);
        }
        if(count < 0)
            throw new Failure2();
    }
}

public class ThrowGenericException {
    public static void main(String[] args) {
        ProcessRunner<String, Failure1> runner =
            new ProcessRunner<>();
        for(int i = 0; i < 3; i++)
            runner.add(new Processor1());
        try {
            System.out.println(runner.processAll());
        } catch(Failure1 e) {
            System.out.println(e);
        }

        ProcessRunner<Integer, Failure2> runner2 =
            new ProcessRunner<>();
        for(int i = 0; i < 3; i++)
            runner2.add(new Processor2());
        try {
            System.out.println(runner2.processAll());
        } catch(Failure2 e) {
            System.out.println(e);
        }
    }
}
/* Output:
[Hep!, Hep!, Ho!]
Failure2
*/
```

Processor 执行 `process()` 方法，并且可能会抛出具有类型 **E** 的异常。`process()` 的结果存储

在 `List<T>resultCollector` 中（这被称为*收集参数*）。**ProcessRunner** 有一个 `processAll()` 方法，它会在所持有的每个 **Process** 对象执行，并返回 **resultCollector** 。如果不能参数化所抛出的异常，那么由于检查型异常的缘故，将不能编写出这种泛化的代码。

21.13 混型

术语*混型*随时间的推移好像拥有了无数的含义，但是其最基本的概念是混合多个类的能力，以产生一个可以表示混型中所有类型的类。这往往是你最后的手段，它将使组装多个类变得简单易行。混型的价值之一是它们可以将特性和行为一致地应用于多个类之上。如果想在混型类中修改某些东西，作为一种意外的好处，这些修改将会应用于混型所应用的所有类型之上。正由于此，混型有一点*面向切面编程*（AOP）的味道，而切面经常被建议用来解决混型问题。

21.13.1 C++ 中的混型

在 C++ 中，使用多重继承的最大理由，就是为了使用混型。但是，对于混型来说，更有趣、更优雅的方式是使用参数化类型，因为混型就是继承自其类型参数的类。在 C++ 中，可以很容易地创建混型，因为 C++ 能够记住其模版参数的类型。下面是一个 C++ 示例，它有两个混型类型：一个使得你可以在每个对象中混入拥有一个时间戳这样的属性，而另一个可以混入一个序列号。

```
// generics/Mixins.cpp

#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class TimeStamped : public T {
    long timeStamp;
public:
    TimeStamped() { timeStamp = time(0); }
    long getStamp() { return timeStamp; }
};

template<class T> class SerialNumbered : public T {
    long serialNumber;
    static long counter;
public:
    SerialNumbered() { serialNumber = counter++; }
    long getSerialNumber() { return serialNumber; }
};

// Define and initialize the static storage:
template<class T> long SerialNumbered<T>::counter = 1;
```

如今的大部分软件都非常像埃及金字塔，由成千上万的石块一个摞一个构成，没有结构上的集成，是由暴力强制和成千上万的奴隶完成。
— Alan Kay

```

class Basic {
    string value;
public:
    void set(string val) { value = val; }
    string get() { return value; }
};

int main() {
    TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
    mixin1.set("test string 1");
    mixin2.set("test string 2");
    cout << mixin1.get() << " " << mixin1.getStamp() <<
      " " << mixin1.getSerialNumber() << endl;
    cout << mixin2.get() << " " << mixin2.getStamp() <<
      " " << mixin2.getSerialNumber() << endl;
}
/* Output:
test string 1 1452987605 1
test string 2 1452987605 2
*/
```

在 `main()` 中，**mixin1** 和 **mixin2** 所产生的类型拥有所混入类型的所有方法。可以将混型看作是一种功能，它可以将现有类映射到新的子类上。注意，使用这种技术来创建一个混型是多么的轻而易举。基本上，只需要声明“这就是我想要的”，紧跟着它就发生了：

```
TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
```

遗憾的是，Java 泛型不允许这样。擦除会忘记基类类型，因此

> 泛型类不能直接继承自一个泛型参数

这突显了许多我在 Java 语言设计决策（以及与这些功能一起发布）中遇到的一大问题：处理一件事很有希望，但是当您实际尝试做一些有趣的事情时，您会发现自己做不到。

21.13.2 与接口混合

一种更常见的推荐解决方案是使用接口来产生混型效果，就像下面这样：

```
// generics/Mixins.java

import java.util.*;

interface TimeStamped { long getStamp(); }

class TimeStampedImp implements TimeStamped {
    private final long timeStamp;
    TimeStampedImp() {
        timeStamp = new Date().getTime();
```

```
    }
    @Override
    public long getStamp() { return timeStamp; }
}

interface SerialNumbered { long getSerialNumber(); }

class SerialNumberedImp implements SerialNumbered {
    private static long counter = 1;
    private final long serialNumber = counter++;
    @Override
    public long getSerialNumber() { return serialNumber; }
}

interface Basic {
    void set(String val);
    String get();
}

class BasicImp implements Basic {
    private String value;
    @Override
    public void set(String val) { value = val; }
    @Override
    public String get() { return value; }
}

class Mixin extends BasicImp
implements TimeStamped, SerialNumbered {
    private TimeStamped timeStamp = new TimeStampedImp();
    private SerialNumbered serialNumber =
        new SerialNumberedImp();
    @Override
    public long getStamp() {
        return timeStamp.getStamp();
    }
    @Override
    public long getSerialNumber() {
        return serialNumber.getSerialNumber();
    }
}

public class Mixins {
    public static void main(String[] args) {
        Mixin mixin1 = new Mixin(), mixin2 = new Mixin();
        mixin1.set("test string 1");
        mixin2.set("test string 2");
        System.out.println(mixin1.get() + " " +
            mixin1.getStamp() +  " " + mixin1.getSerialNumber());
        System.out.println(mixin2.get() + " " +
            mixin2.getStamp() +  " " + mixin2.getSerialNumber());
```

```
    }
}
/* Output:
test string 1 1494331663026 1
test string 2 1494331663027 2
*/
```

Mixin 类基本上是在使用*委托*，因此每个混入类型都要求在 **Mixin** 中有一个相应的域，而你必须在 **Mixin** 中编写所有必需的方法，将方法调用转发给恰当的对象。这个示例使用了非常简单的类，但是当使用更复杂的混型时，代码数量会急速增加。

21.13.3 使用装饰器模式

当你观察混型的使用方式时，就会发现混型概念好像与*装饰器*设计模式关系很近。装饰器经常用于满足各种可能的组合，而直接子类化会产生过多的类，因此是不实际的。装饰器模式使用分层对象来动态透明地向单个对象中添加责任。装饰器指定包装在最初的对象周围的所有对象都具有相同的基本接口。某些事物是可装饰的，可以通过将其他类包装在这个可装饰对象的四周，来将功能分层。这使得对装饰器的使用是透明的——无论对象是否被装饰，你都拥有一个可以向对象发送的公共消息集。装饰类也可以添加新方法，但是正如你所见，这将是受限的。装饰器是通过使用组合和形式化结构（可装饰物/装饰器层次结构）来实现的，而混型是基于继承的。因此可以将基于参数化类型的混型当作是一种泛型装饰器机制，这种机制不需要装饰器设计模式的继承结构。前面的示例可以被改写为使用装饰器：

```java
// generics/decorator/Decoration.java

// {java generics.decorator.Decoration}
package generics.decorator;
import java.util.*;

class Basic {
    private String value;
    public void set(String val) { value = val; }
    public String get() { return value; }
}

class Decorator extends Basic {
    protected Basic basic;
    Decorator(Basic basic) { this.basic = basic; }
    @Override
    public void set(String val) { basic.set(val); }
    @Override
    public String get() { return basic.get(); }
}

class TimeStamped extends Decorator {
    private final long timeStamp;
```

```
    TimeStamped(Basic basic) {
        super(basic);
        timeStamp = new Date().getTime();
    }
    public long getStamp() { return timeStamp; }
}

class SerialNumbered extends Decorator {
    private static long counter = 1;
    private final long serialNumber = counter++;
    SerialNumbered(Basic basic) { super(basic); }
    public long getSerialNumber() { return serialNumber; }
}

public class Decoration {
    public static void main(String[] args) {
        TimeStamped t = new TimeStamped(new Basic());
        TimeStamped t2 = new TimeStamped(
            new SerialNumbered(new Basic()));
        //- t2.getSerialNumber(); // Not available
        SerialNumbered s = new SerialNumbered(new Basic());
        SerialNumbered s2 = new SerialNumbered(
            new TimeStamped(new Basic()));
        //- s2.getStamp(); // Not available
  }
}
```

产生自泛型的类包含所有感兴趣的方法，但是由使用装饰器所产生的对象类型是最后被装饰的类型。也就是说，尽管可以添加多个层，但是最后一层才是实际的类型，因此只有最后一层的方法是可视的，而混型的类型是所有被混合到一起的类型。因此对于装饰器来说，其明显的缺陷是它只能有效地工作于装饰中的一层（最后一层），而混型方法显然会更自然一些。因此，装饰器只是对由混型提出的问题的一种局限的解决方案。

21.13.4 与动态代理混合

可以使用动态代理来创建一种比装饰器更贴近混型模型的机制（查看 类型信息 一章中关于 Java 的动态代理如何工作的解释）。通过使用动态代理，所产生的类的动态类型将会是已经混入的组合类型。由于动态代理的限制，每个被混入的类都必须是某个接口的实现：

```
// generics/DynamicProxyMixin.java

import java.lang.reflect.*;
import java.util.*;
import onjava.*;
import static onjava.Tuple.*;

class MixinProxy implements InvocationHandler {
```

```
    Map<String, Object> delegatesByMethod;
    @SuppressWarnings("unchecked")
    MixinProxy(Tuple2<Object, Class<?>>... pairs) {
        delegatesByMethod = new HashMap<>();
        for(Tuple2<Object, Class<?>> pair : pairs) {
            for(Method method : pair.a2.getMethods()) {
                String methodName = method.getName();
                // The first interface in the map
                // implements the method.
                if(!delegatesByMethod.containsKey(methodName))
                    delegatesByMethod.put(methodName, pair.a1);
            }
        }
    }
    @Override
    public Object invoke(Object proxy, Method method,
      Object[] args) throws Throwable {
        String methodName = method.getName();
        Object delegate = delegatesByMethod.get(methodName);
        return method.invoke(delegate, args);
    }

    @SuppressWarnings("unchecked")
    public static Object newInstance(Tuple2... pairs) {
        Class[] interfaces = new Class[pairs.length];
        for(int i = 0; i < pairs.length; i++) {
            interfaces[i] = (Class)pairs[i].a2;
        }
        ClassLoader cl = pairs[0].a1.getClass().getClassLoader();
        return Proxy.newProxyInstance(cl, interfaces, new MixinProxy(
           pairs));
    }
}

public class DynamicProxyMixin {
    public static void main(String[] args) {
        Object mixin = MixinProxy.newInstance(
          tuple(new BasicImp(), Basic.class),
          tuple(new TimeStampedImp(), TimeStamped.class),
          tuple(new SerialNumberedImp(), SerialNumbered.class));
        Basic b = (Basic)mixin;
        TimeStamped t = (TimeStamped)mixin;
        SerialNumbered s = (SerialNumbered)mixin;
        b.set("Hello");
        System.out.println(b.get());
        System.out.println(t.getStamp());
        System.out.println(s.getSerialNumber());
    }
}
/* Output:
Hello
```

```
1494331653339
1
*/
```

因为只有动态类型而不是静态类型才包含所有的混入类型，因此这仍旧不如 C++ 的方式好，因为可以在具有这些类型的对象上调用方法之前，你被强制要求必须先将这些对象向下转型到恰当的类型。但是，它明显地更接近于真正的混型。为了让 Java 支持混型，人们已经做了大量的工作朝着这个目标努力，包括创建了至少一种附加语言（Jam 语言），它是专门用来支持混型的。

21.14 潜在类型机制

在本章的开头介绍过这样的思想，即要编写能够尽可能广泛地应用的代码。为了实现这一点，我们需要各种途径来放松对我们的代码将要作用的类型所作的限制，同时不丢失静态类型检查的好处。然后，我们就可以编写出无需修改就可以应用于更多情况的代码，即更加“泛化”的代码。

Java 泛型看起来是向这一方向迈进了一步。当你在编写或使用只是持有对象的泛型时，这些代码将可以工作于任何类型（除了基本类型，尽管正如你所见到的，自动装箱机制可以克服这一点）。或者，换个角度讲，“持有器”泛型能够声明：“我不关心你是什么类型”。如果代码不关心它将要作用的类型，那么这种代码就可以真正地应用于任何地方，并因此而相当泛化。

还是正如你所见到的，当要在泛型类型上执行操作（即调用 **Object** 方法之外的方法）时，就会产生问题。擦除强制要求指定可能会用到的泛型类型的边界，以安全地调用代码中的泛型对象上的具体方法。这是对“泛化”概念的一种明显的限制，因为必须限制你的泛型类型，使它们继承自特定的类，或者实现特定的接口。在某些情况下，你最终可能会使用普通类或普通接口，因为限定边界的泛型可能会和指定类或接口没有任何区别。

某些编程语言提供的一种解决方案称为*潜在类型机制*或*结构化类型机制*，而更古怪的术语称为*鸭子类型机制*，即“如果它走起来像鸭子，并且叫起来也像鸭子，那么你就可以将它当作鸭子对待。”鸭子类型机制变成了一种相当流行的术语，可能是因为它不像其他的术语那样承载着历史的包袱。

泛型代码典型地只能在泛型类型上调用少量方法，而具有潜在类型机制的语言只要求实现某个方法子集，而不是某个特定类或接口，从而放松了这种限制（并且可以产生更加泛化的代码）。正由于此，潜在类型机制使得你可以横跨类继承结构，调用不属于某个公共接口的方法。因此，实际上一段代码可以声明：“我不关心你是什么类型，只要你可以 `speak()` 和 `sit()` 即可。”由于不要求具体类型，因此代码就可以更加泛化。

潜在类型机制是一种代码组织和复用机制。有了它，编写出的代码相对于没有它编写出的代码，能够更容易地复用。代码组织和复用是所有计算机编程的基本手段：编写一次，多次使用，并在一个位置保存代码。因为我并未被要求去命名我的代码要操作于其上的确切接口，所以，有了潜在类型机制，我就可以编写更少的代码，并更容易地将其应用于多个地方。

支持潜在类型机制的语言包括 Python（可以从 www.Python.org 免费下载）、C++、Ruby、

SmallTalk 和 Go。Python 是动态类型语言（几乎所有的类型检查都发生在运行时），而 C++ 和 Go 是静态类型语言（类型检查发生在编译期），因此潜在类型机制不要求静态或动态类型检查。

21.14.1 pyhton 中的潜在类型

如果我们将上面的描述用 Python 来表示，如下所示：

```python
# generics/DogsAndRobots.py

class Dog:
    def speak(self):
        print("Arf!")
    def sit(self):
        print("Sitting")
    def reproduce(self):
        pass

class Robot:
    def speak(self):
        print("Click!")
    def sit(self):
        print("Clank!")
    def oilChange(self):
        pass

def perform(anything):
    anything.speak()
    anything.sit()

a = Dog()
b = Robot()
perform(a)
perform(b)

output = """
Arf!
Sitting
Click!
Clank!
"""
```

Python 使用缩进来确定作用域（因此不需要任何花括号），而冒号将表示新的作用域的开始。“#”表示注释到行尾，就像 Java 中的 “ // ”。类的方法需要显式地指定 **this** 引用的等价物作为第一个参数，按惯例成为 **self** 。构造器调用不要求任何类型的 “ **new** ” 关键字，并且 Python 允许普通（非成员）函数，就像 `perform()` 所表明的那样。注意，在 `perform(anything)` 中，没有任何针对 **anything** 的类型，**anything** 只是一个标识符，它必须能够执行 `perform()` 期望它执行的操作，因此这里隐含着一个接口。但是你从来都不必显式地写出这个接口——它是潜在的。`perform()` 不关心其参数的类型，

想法把自己变得不重要，以实现其他重要的目标；

因此我可以向它传递任何对象，只要该对象支持 `speak()` 和 `sit()` 方法。如果传递给 `perform()` 的对象不支持这些操作，那么将会得到运行时异常。

输出规定使用三重引号创建带有内嵌换行符的字符串。

21.14.2 C++ 中的潜在类型

我们可以用 C++ 产生相同的效果:

```
// generics/DogsAndRobots.cpp

#include <iostream>
using namespace std;

class Dog {
public:
    void speak() { cout << "Arf!" << endl; }
    void sit() { cout << "Sitting" << endl; }
    void reproduce() {}
};

class Robot {
public:
    void speak() { cout << "Click!" << endl; }
    void sit() { cout << "Clank!" << endl; }
    void oilChange() {}
};

template<class T> void perform(T anything) {
    anything.speak();
    anything.sit();
}

int main() {
    Dog d;
    Robot r;
    perform(d);
    perform(r);
}
/* Output:
Arf!
Sitting
Click!
Clank!
*/
```

在 Python 和 C++ 中，**Dog** 和 **Robot** 没有任何共同的东西，只是碰巧有两个方法具有相同的签名。从类型的观点看，它们是完全不同的类型。但是，`perform()` 不关心其参数的具体类型，并且潜在类型机制允许它接受这两种类型的对象。C++ 确保了它实际上可以发送的那些消息，如果试图传递错误

类型，编译器就会给你一个错误消息（这些错误消息从历史上看是相当可怕和冗长的，是 C++ 的模版名声欠佳的主要原因）。尽管它们是在不同时期实现这一点的，C++ 在编译期，而 Python 在运行时，但是这两种语言都可以确保类型不会被误用，因此被认为是强类型的。潜在类型机制没有损害强类型机制。

21.14.3 Go 中的潜在类型

这里用 Go 语言编写相同的程序：

```go
// generics/dogsandrobots.go

package main
import "fmt"

type Dog struct {}
func (this Dog) speak() { fmt.Printf("Arf!\n")}
func (this Dog) sit() { fmt.Printf("Sitting\n")}
func (this Dog) reproduce() {}

type Robot struct {}
func (this Robot) speak() { fmt.Printf("Click!\n") }
func (this Robot) sit() { fmt.Printf("Clank!\n") }
func (this Robot) oilChange() {}

func perform(speaker interface { speak(); sit() }) {
  speaker.speak();
  speaker.sit();
}

func main() {
  perform(Dog{})
  perform(Robot{})
}
/* Output:
Arf!
Sitting
Click!
Clank!
*/
```

Go 没有 **class** 关键字，但是可以使用上述形式创建等效的基本类：它通常不定义为类，而是定义为 **struct** ，在其中定义数据字段（此处不存在）。对于每种方法，都以 **func** 关键字开头，然后（为了将该方法附加到您的类上）放在括号中，该括号包含对象引用，该对象引用可以是任何标识符，但是我在这里使用 **this** 来提醒您，就像在 C ++ 或 Java 中的 **this** 一样。然后，在 Go 中像这样定义其余的函数。

Go 也没有继承关系，因此这种“面向对象的目标”形式是相对原始的，并且可能是我无法花更多的时间来学习该语言的主要原因。但是，Go 的组成很简单。

perform() 函数使用潜在类型：参数的确切类型并不重要，只要它包含了 speak() 和 sit() 方法即可。该接口在此处匿名定义，内联，如 perform() 的参数列表所示。

main() 证明 perform() 确实对其参数的确切类型不在乎，只要可以在该参数上调用 talk() 和 sit() 即可。但是，就像 C ++ 模板函数一样，在编译时检查类型。

语法 **Dog {}** 和 **Robot {}** 创建匿名的 **Dog** 和 **Robot** 结构。

21.14.4 java 中的直接潜在类型

因为泛型是在这场竞赛的后期才添加到 Java 中，因此没有任何机会可以去实现任何类型的潜在类型机制，因此 Java 没有对这种特性的支持。所以，初看起来，Java 的泛型机制比支持潜在类型机制的语言更“缺乏泛化性”。（使用擦除来实现 Java 泛型的实现有时称为第二类泛型类型）例如，在 Java 8 之前如果我们试图用 Java 实现上面 dogs-and-robots 的示例，那么就会被强制要求使用一个类或接口，并在边界表达式中指定它：

```
// generics/Performs.java

public interface Performs {
    void speak();
    void sit();
}
```

```
// generics/DogsAndRobots.java
// No (direct) latent typing in Java
import typeinfo.pets.*;

class PerformingDog extends Dog implements Performs {
    @Override
    public void speak() { System.out.println("Woof!"); }
    @Override
    public void sit() { System.out.println("Sitting"); }
    public void reproduce() {}
}

class Robot implements Performs {
    public void speak() { System.out.println("Click!"); }
    public void sit() { System.out.println("Clank!"); }
    public void oilChange() {}
}

class Communicate {
    public static <T extends Performs>
      void perform(T performer) {
        performer.speak();
        performer.sit();
    }
```

```
}

public class DogsAndRobots {
    public static void main(String[] args) {
        Communicate.perform(new PerformingDog());
        Communicate.perform(new Robot());
    }
}
/* Output:
Woof!
Sitting
Click!
Clank!
*/
```

但是要注意，`perform()` 不需要使用泛型来工作，它可以被简单地指定为接受一个 **Performs** 对象：

```
// generics/SimpleDogsAndRobots.java
// Removing the generic; code still works

class CommunicateSimply {
    static void perform(Performs performer) {
        performer.speak();
        performer.sit();
    }
}

public class SimpleDogsAndRobots {
    public static void main(String[] args) {
        CommunicateSimply.perform(new PerformingDog());
        CommunicateSimply.perform(new Robot());
    }
}
/* Output:
Woof!
Sitting
Click!
Clank!
*/
```

在本例中，泛型不是必需的，因为这些类已经被强制要求实现 **Performs** 接口。

21.15 对缺乏潜在类型机制的补偿

尽管 Java 不直接支持潜在类型机制，但是这并不意味着泛型代码不能在不同的类型层次结构之间应用。也就是说，我们仍旧可以创建真正的泛型代码，但是这需要付出一些额外的努力。

最近一帮做技术的在微博上吵谁比谁 NB。其实做技术不像练武，非得打遍天下。能很好解决自己遇到的问题搞定公司的需求就很好；进一步，精益求精下，提升下性能重构下接口，做到优雅可读；再进一步，整理成可重用的方式发布出来，让更多人受益。真正 NB 是多少人因为你而活得更好，而非其他；

21.15.1 反射

可以使用的一种方式是反射，下面的 `perform()` 方法就是用了潜在类型机制：

```
// generics/LatentReflection.java
// Using reflection for latent typing
import java.lang.reflect.*;

// Does not implement Performs:
class Mime {
    public void walkAgainstTheWind() {}
    public void sit() {
        System.out.println("Pretending to sit");
    }
    public void pushInvisibleWalls() {}
    @Override
    public String toString() { return "Mime"; }
}

// Does not implement Performs:
class SmartDog {
    public void speak() { System.out.println("Woof!"); }
    public void sit() { System.out.println("Sitting"); }
    public void reproduce() {}
}

class CommunicateReflectively {
    public static void perform(Object speaker) {
        Class<?> spkr = speaker.getClass();
        try {
            try {
                Method speak = spkr.getMethod("speak");
                speak.invoke(speaker);
            } catch(NoSuchMethodException e) {
                System.out.println(speaker + " cannot speak");
            }
            try {
                Method sit = spkr.getMethod("sit");
                sit.invoke(speaker);
            } catch(NoSuchMethodException e) {
                System.out.println(speaker + " cannot sit");
            }
        } catch(SecurityException |
            IllegalAccessException |
            IllegalArgumentException |
            InvocationTargetException e) {
            throw new RuntimeException(speaker.toString(), e);
        }
    }
}
```

```

public class LatentReflection {
    public static void main(String[] args) {
        CommunicateReflectively.perform(new SmartDog());
        CommunicateReflectively.perform(new Robot());
        CommunicateReflectively.perform(new Mime());
    }
}
/* Output:
Woof!
Sitting
Click!
Clank!
Mime cannot speak
Pretending to sit
*/
```

上例中，这些类完全是彼此分离的，没有任何公共基类（除了 **Object** ）或接口。通过反射，`CommunicateReflectively.perform()` 能够动态地确定所需要的方法是否可用并调用它们。它甚至能够处理 **Mime** 只具有一个必需的方法这一事实，并能够部分实现其目标。

21.15.2 将一个方法应用于序列

反射提供了一些有用的可能性，但是它将所有的类型检查都转移到了运行时，因此在许多情况下并不是我们所希望的。如果能够实现编译期类型检查，这通常会更符合要求。但是有可能实现编译期类型检查和潜在类型机制吗?

让我们看一个说明这个问题的示例。假设想要创建一个 `apply()` 方法，它能够将任何方法应用于某个序列中的所有对象。这种情况下使用接口不适合，因为你想要将任何方法应用于一个对象集合，而接口不可能描述任何方法。如何用 Java 来实现这个需求呢?

最初，我们可以用反射来解决这个问题，由于有了 Java 的可变参数，这种方式被证明是相当优雅的:

```
// generics/Apply.java

import java.lang.reflect.*;
import java.util.*;

public class Apply {
    public static <T, S extends Iterable<T>>
      void apply(S seq, Method f, Object... args) {
        try {
            for(T t: seq)
                f.invoke(t, args);
        } catch(IllegalAccessException |
            IllegalArgumentException |
```

编写 Java 开发包都是一群神人，拿 Java 源码来分析问题，我们无疑站在了巨人的肩膀上。所谓站得高，尿的远也。当然，所谓偷拍都是避免不了的;

```
            InvocationTargetException e) {
            // Failures are programmer errors
            throw new RuntimeException(e);
        }
    }
}
```

在 **Apply.java** 中，异常被转换为 **RuntimeException** ，因为没有多少办法可以从这种异常中恢复——在这种情况下，它们实际上代表着程序员的错误。

为什么我们不只使用 Java 8 方法参考（稍后显示）而不是反射方法 **f** ？注意，`invoke()` 和 `apply()` 的优点是它们可以接受任意数量的参数。在某些情况下，灵活性可能至关重要。

为了测试 **Apply** ，我们首先创建一个 **Shape** 类：

```
// generics/Shape.java

public class Shape {
    private static long counter = 0;
    private final long id = counter++;
    @Override
    public String toString() {
        return getClass().getSimpleName() + " " + id;
    }
    public void rotate() {
        System.out.println(this + " rotate");
    }
    public void resize(int newSize) {
        System.out.println(this + " resize " + newSize);
    }
}
```

被一个子类 **Square** 继承：

```
// generics/Square.java

public class Square extends Shape {}
```

通过这些，我们可以测试 **Apply**：

```
// generics/ApplyTest.java

import java.util.*;
import java.util.function.*;
import onjava.*;

public class ApplyTest {
    public static
    void main(String[] args) throws Exception {
        List<Shape> shapes =
          Suppliers.create(ArrayList::new, Shape::new, 3);
```

```
        Apply.apply(shapes, Shape.class.getMethod("rotate"));
        Apply.apply(shapes, Shape.class.getMethod("resize", int.class),
            7);

        List<Square> squares =
          Suppliers.create(ArrayList::new, Square::new, 3);
        Apply.apply(squares, Shape.class.getMethod("rotate"));
        Apply.apply(squares, Shape.class.getMethod("resize", int.class)
            , 7);

        Apply.apply(new FilledList<>(Shape::new, 3),
          Shape.class.getMethod("rotate"));
        Apply.apply(new FilledList<>(Square::new, 3),
          Shape.class.getMethod("rotate"));

        SimpleQueue<Shape> shapeQ = Suppliers.fill(
          new SimpleQueue<>(), SimpleQueue::add,
          Shape::new, 3);
        Suppliers.fill(shapeQ, SimpleQueue::add,
          Square::new, 3);
        Apply.apply(shapeQ, Shape.class.getMethod("rotate"));
    }
}
/* Output:
Shape 0 rotate
Shape 1 rotate
Shape 2 rotate
Shape 0 resize 7
Shape 1 resize 7
Shape 2 resize 7
Square 3 rotate
Square 4 rotate
Square 5 rotate
Square 3 resize 7
Square 4 resize 7
Square 5 resize 7
Shape 6 rotate
Shape 7 rotate
Shape 8 rotate
Square 9 rotate
Square 10 rotate
Square 11 rotate
Shape 12 rotate
Shape 13 rotate
Shape 14 rotate
Square 15 rotate
Square 16 rotate
Square 17 rotate
*/
```

在 **Apply** 中，我们运气很好，因为碰巧在 Java 中内建了一个由 Java 集合类库使用的 **Iterable**

接口。正由于此，`apply()` 方法可以接受任何实现了 **Iterable** 接口的事物，包括诸如 **List** 这样的所有 **Collection** 类。但是它还可以接受其他任何事物，只要能够使这些事物是 **Iterable** 的——例如，在 `main()` 中使用下面定义的 **SimpleQueue** 类：

```
// generics/SimpleQueue.java

// A different kind of Iterable collection
import java.util.*;

public class SimpleQueue<T> implements Iterable<T> {
    private LinkedList<T> storage = new LinkedList<>();
    public void add(T t) { storage.offer(t); }
    public T get() { return storage.poll(); }
    @Override
    public Iterator<T> iterator() {
        return storage.iterator();
    }
}
```

正如反射解决方案看起来那样优雅，我们必须观察到反射（尽管在 Java 的最新版本中得到了显着改进）通常比非反射实现要慢，因为在运行时发生了很多事情。但它不应阻止您尝试这种解决方案，这依然是值得考虑的一点。

几乎可以肯定，你会首先使用 Java 8 的函数式方法，并且只有在解决了特殊需求时才诉诸反射。这里对 **ApplyTest.java** 进行了重写，以利用 Java 8 的流和函数工具：

```
// generics/ApplyFunctional.java

import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import onjava.*;

public class ApplyFunctional {
    public static void main(String[] args) {
        Stream.of(
          Stream.generate(Shape::new).limit(2),
          Stream.generate(Square::new).limit(2))
        .flatMap(c -> c) // flatten into one stream
        .peek(Shape::rotate)
        .forEach(s -> s.resize(7));

        new FilledList<>(Shape::new, 2)
          .forEach(Shape::rotate);
        new FilledList<>(Square::new, 2)
          .forEach(Shape::rotate);

        SimpleQueue<Shape> shapeQ = Suppliers.fill(
          new SimpleQueue<>(), SimpleQueue::add,
```

```
            Shape::new, 2);
          Suppliers.fill(shapeQ, SimpleQueue::add,
            Square::new, 2);
          shapeQ.forEach(Shape::rotate);
      }
}
/* Output:
Shape 0 rotate
Shape 0 resize 7
Shape 1 rotate
Shape 1 resize 7
Square 2 rotate
Square 2 resize 7
Square 3 rotate
Square 3 resize 7
Shape 4 rotate
Shape 5 rotate
Square 6 rotate
Square 7 rotate
Shape 8 rotate
Shape 9 rotate
Square 10 rotate
Square 11 rotate
*/
```

由于使用 Java 8，因此不需要 `Apply.apply()` 。

我们首先生成两个 **Stream** ： 一个是 **Shape** ， 一个是 **Square** ， 并将它们展平为单个流。尽管 Java 缺少功能语言中经常出现的 `flatten()` ，但是我们可以使用 `flatMap(c-> c)` 产生相同的结果，后者使用身份映射将操作简化为 “ **flatten** ”。

我们使用 `peek()` 当做对 `rotate()` 的调用，因为 `peek()` 执行一个操作（此处是出于副作用），并在未更改的情况下传递对象。

注意，使用 **FilledList** 和 **shapeQ** 调用 `forEach()` 比 `Apply.apply()` 代码整洁得多。在代码简单性和可读性方面，结果比以前的方法好得多。并且，现在也不可能从 `main()` 引发异常。

21.16 Java8 中的辅助潜在类型

先前声明关于 Java 缺乏对潜在类型的支持在 Java 8 之前是完全正确的。但是，Java 8 中的非绑定方法引用使我们能够产生一种潜在类型的形式，以满足创建一段可工作在不相干类型上的代码。因为 Java 最初并不是如此设计，所以结果可想而知，比其他语言中要尴尬一些。但是，至少现在成为了可能，只是缺乏令人惊艳之处。

我在其他地方从没遇过这种技术，因此我将其称为辅助潜在类型。

我们将重写 **DogsAndRobots.java** 来演示该技术。为使外观看起来与原始示例尽可能相似，我仅向每个原始类名添加了 **A**:

```java
// generics/DogsAndRobotMethodReferences.java

// "Assisted Latent Typing"
import typeinfo.pets.*;
import java.util.function.*;

class PerformingDogA extends Dog {
    public void speak() { System.out.println("Woof!"); }
    public void sit() { System.out.println("Sitting"); }
    public void reproduce() {}
}

class RobotA {
    public void speak() { System.out.println("Click!"); }
    public void sit() { System.out.println("Clank!"); }
    public void oilChange() {}
}

class CommunicateA {
    public static <P> void perform(P performer,
      Consumer<P> action1, Consumer<P> action2) {
        action1.accept(performer);
        action2.accept(performer);
    }
}

public class DogsAndRobotMethodReferences {
    public static void main(String[] args) {
        CommunicateA.perform(new PerformingDogA(),
          PerformingDogA::speak, PerformingDogA::sit);
        CommunicateA.perform(new RobotA(),
          RobotA::speak, RobotA::sit);
        CommunicateA.perform(new Mime(),
          Mime::walkAgainstTheWind,
          Mime::pushInvisibleWalls);
    }
}
/* Output:
Woof!
Sitting
Click!
Clank!
*/
```

PerformingDogA 和 **RobotA** 与 **DogsAndRobots.java** 中的相同，不同之处在于它们不继承通用接口 **Performs**，因此它们没有通用性。

`CommunicateA.perform()` 在没有约束的 **P** 上生成。只要可以使用 `Consumer <P>`，它在这里就可以是任何东西，这些 `Consumer<P>` 代表不带参数的 **P** 方法的未绑定方法引用。当您调用

Consumer 的 `accept()` 方法时，它将方法引用绑定到执行者对象并调用该方法。由于 函数式编程 一章中描述的“魔术”，我们可以将任何符合签名的未绑定方法引用传递给 `CommunicateA.perform()`。

之所以称其为“辅助”，是因为您必须显式地为 `perform()` 提供要使用的方法引用。它不能只按名称调用方法。

尽管传递未绑定的方法引用似乎要花很多力气，但潜在类型的最终目标还是可以实现的。我们创建了一个代码片段 `CommunicateA.perform()`，该代码可用于任何具有符合签名的方法引用的类型。请注意，这与我们看到的其他语言中的潜在类型有所不同，因为这些语言不仅需要签名以符合规范，还需要方法名称。因此，该技术可以说产生了更多的通用代码。

为了证明这一点，我还从 **LatentReflection.java** 中引入了 **Mime**。

21.16.1 使用 Suppliers 类的通用方法

通过辅助潜在类型，我们可以定义本章其他部分中使用的 **Suppliers** 类。此类包含使用生成器填充 **Collection** 的工具方法。泛化这些操作很有意义：

```java
// onjava/Suppliers.java

// A utility to use with Suppliers
package onjava;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Suppliers {
    // Create a collection and fill it:
    public static <T, C extends Collection<T>> C
      create(Supplier<C> factory, Supplier<T> gen, int n) {
        return Stream.generate(gen)
            .limit(n)
            .collect(factory, C::add, C::addAll);
    }

    // Fill an existing collection:
    public static <T, C extends Collection<T>>
      C fill(C coll, Supplier<T> gen, int n) {
        Stream.generate(gen)
            .limit(n)
            .forEach(coll::add);
        return coll;
    }

    // Use an unbound method reference to
    // produce a more general method:
    public static <H, A> H fill(H holder,
```

羡慕别人有个好爸爸，没什么不可以。问题是，你的下一代，会有一个好爸爸吗？至于问到为什么不能有同样的赢面概率？我只能问：为什么物种竞争中，人和猴子不能有同样的赢面概率？物竞天择。猴子的灵魂不一定比你卑微，但你身后有几十万年的类人猿进化积淀；

```
        BiConsumer<H, A> adder, Supplier<A> gen, int n) {
          Stream.generate(gen)
              .limit(n)
              .forEach(a -> adder.accept(holder, a));
          return holder;
      }
  }
```

create() 为你创建一个新的 **Collection** 子类型，而 fill() 的第一个版本将元素放入 **Collection** 的现有子类型中。请注意，还会返回传入的容器的确切类型，因此不会丢失类型信息。

前两种方法一般都受约束，只能与 **Collection** 子类型一起使用。fill() 的第二个版本适用于任何类型的 **holder** 。它需要一个附加参数：未绑定方法引用 adder. fill() ，使用辅助潜在类型来使其与任何具有添加元素方法的 **holder** 类型一起使用。因为此未绑定方法 **adder** 必须带有一个参数（要添加到 **holder** 的元素），所以 **adder** 必须是 BiConsumer <H, A> ，其中 **H** 是要绑定到的 **holder** 对象的类型，而 **A** 是要被添加的绑定元素类型。对 accept() 的调用将使用参数 a 调用对象 **holder** 上的未绑定方法 **holder**。

在一个稍作模拟的测试中对 **Suppliers** 工具程序进行了测试，该仿真还使用了本章前面定义的 **RandomList** ：

```
// generics/BankTeller.java

// A very simple bank teller simulation
import java.util.*;
import onjava.*;

class Customer {
    private static long counter = 1;
    private final long id = counter++;
    @Override
    public String toString() {
        return "Customer " + id;
    }
}

class Teller {
    private static long counter = 1;
    private final long id = counter++;
    @Override
    public String toString() {
        return "Teller " + id;
    }
}

class Bank {
    private List<BankTeller> tellers =
        new ArrayList<>();
    public void put(BankTeller bt) {
```

```
        tellers.add(bt);
    }
}

public class BankTeller {
    public static void serve(Teller t, Customer c) {
        System.out.println(t + " serves " + c);
    }
    public static void main(String[] args) {
        // Demonstrate create():
        RandomList<Teller> tellers =
            Suppliers.create(
            RandomList::new, Teller::new, 4);
        // Demonstrate fill():
        List<Customer> customers = Suppliers.fill(
            new ArrayList<>(), Customer::new, 12);
        customers.forEach(c ->
            serve(tellers.select(), c));
        // Demonstrate assisted latent typing:
        Bank bank = Suppliers.fill(
            new Bank(), Bank::put, BankTeller::new, 3);
        // Can also use second version of fill():
        List<Customer> customers2 = Suppliers.fill(
            new ArrayList<>(),
            List::add, Customer::new, 12);
    }
}
/* Output:
Teller 3 serves Customer 1
Teller 2 serves Customer 2
Teller 3 serves Customer 3
Teller 1 serves Customer 4
Teller 1 serves Customer 5
Teller 3 serves Customer 6
Teller 1 serves Customer 7
Teller 2 serves Customer 8
Teller 3 serves Customer 9
Teller 3 serves Customer 10
Teller 2 serves Customer 11
Teller 4 serves Customer 12
*/
```

可以看到 `create()` 生成一个新的 **Collection** 对象，而 `fill()` 添加到现有 **Collection** 中。第二个版本`fill()` 显示，它不仅与无关的新类型 **Bank** 一起使用，还能与 **List** 一起使用。因此，从技术上讲，`fill()` 的第一个版本在技术上不是必需的，但在使用 **Collection** 时提供了较短的语法。

21.17 总结：类型转换真的如此之糟吗？

自从 C++ 模版出现以来，我就一直在致力于解释它，我可能比大多数人都更早地提出了下面的论点。直到最近，我才停下来，去思考这个论点到底在多少时间内是有效的——我将要描述的问题到底有多少次可以穿越障碍得以解决。

这个论点就是：使用泛型类型机制的最吸引人的地方，就是在使用集合类的地方，这些类包括诸如各种 **List** 、各种 **Set** 、各种 **Map** 等你在 集合 和 附录：集合主题 这两章所见。在 Java 5 之前，当你将一个对象放置到集合中时，这个对象就会被向上转型为 **Object** ，因此你会丢失类型信息。当你想要将这个对象从集合中取回，用它去执行某些操作时，必须将其向下转型回正确的类型。我用的示例是持有 **Cat** 的 **List** （这个示例的一种使用苹果和桔子的变体在 集合 章节的开头展示过）。如果没有 Java 5 泛型版本的集合，你放到容集里和从集合中取回的都是 **Object** 。因此，我们很可能会将一个 **Dog** 放置到 **Cat** 的 **List** 中。

但是，泛型出现之前的 Java 并不会让你误用放入到集合中的对象。如果将一个 **Dog** 扔到 **Cat** 的集合中，并且试图将这个集合中的所有东西都当作 **Cat** 处理，那么当你从这个 **Cat** 集合中取回那个 **Dog** 引用，并试图将其转型为 **Cat** 时，就会得到一个 **RuntimeException** 。你仍旧可以发现问题，但是是在运行时而非编译期发现它的。

在本书以前的版本中，我曾经说过：

> 这不止令人恼火，它还可能会产生难以发现的缺陷。如果这个程序的某个部分（或数个部分）向集合中插入了对象，并且通过异常，你在程序的另一个独立的部分中发现有不良对象被放置到了集合中，那么必须发现这个不良插入到底是在何处发生的。

但是，随着对这个论点的进一步检查，我开始怀疑它了。首先，这会多么频繁地发生呢？我记得这类事情从未发生在我身上，并且当我在会议上询问其他人时，我也从来没有听说过有人碰上过。另一本书使用了一个称为 **files** 的 list 示例，它包含 **String** 对象。在这个示例中，向 **files** 中添加一个 **File** 对象看起来相当自然，因此这个对象的名字可能叫 **fileNames** 更好。无论 Java 提供了多少类型检查，仍旧可能会写出晦涩的程序，而编写差劲儿的程序即便可以编译，它仍旧是编写差劲儿的程序。可能大多数人都会使用命名良好的集合，例如 **cats** ，因为它们可以向试图添加非 **Cat** 对象的程序员提供可视的警告。并且即便这类事情发生了，它真正又能潜伏多久呢？只要你开始用真实数据来运行测试，就会非常快地看到异常。

有一位作者甚至断言，这样的缺陷将“*潜伏数年*”。但是我不记得有任何大量的相关报告，来说明人们在查找“狗在猫列表中”这类缺陷时困难重重，或者是说明人们会非常频繁地产生这种错误。然而，你将在 多线程编程 章节中看到，在使用线程时，出现那些可能看起来极罕见的缺陷，是很寻常并容易发生的事，而且，对于到底出了什么错，这些缺陷只能给你一个很模糊的概念。因此，对于泛型是添加到 Java 中的非常显著和相当复杂的特性这一点，“狗在猫列表中”这个论据真的能够成为它的理由吗？我相信被称为*泛型*的通用语言特性（并非必须是其在 Java 中的特定实现）的目的在于可表达性，而不仅仅是为了创建类型安全的集合。类型安全的集合是能够创建更通用代码这一能力所带来的副作用。因此，即

记得，对于那些以夺取你最重要的东西——比如健康、生活、好心情、从容、家庭为代价的工资，千万别碰；

便“狗在猫列表中”这个论据经常被用来证明泛型是必要的，但是它仍旧是有问题的。就像我在本章开头声称的，我不相信这就是泛型这个概念真正的含义。相反，泛型正如其名称所暗示的：它是一种方法，通过它可以编写出更“泛化”的代码，这些代码对于它们能够作用的类型具有更少的限制，因此单个的代码段可以应用到更多的类型上。正如你在本章中看到的，编写真正泛化的“持有器”类（Java 的容器就是这种类）相当简单，但是编写出能够操作其泛型类型的泛化代码就需要额外的努力了，这些努力需要类创建者和类消费者共同付出，他们必须理解这些代码的概念和实现。这些额外的努力会增加使用这种特性的难度，并可能会因此而使其在某些场合缺乏可应用性，而在这些场合中，它可能会带来附加的价值。

还要注意到，因为泛型是后来添加到 Java 中，而不是从一开始就设计到这种语言中的，所以某些容器无法达到它们应该具备的健壮性。例如，观察一下 **Map** ，在特定的方法 `containsKey(Object key)`和 `get(Object key)` 中就包含这类情况。如果这些类是使用在它们之前就存在的泛型设计的，那么这些方法将会使用参数化类型而不是 **Object** ，因此也就可以提供这些泛型假设会提供的编译期检查。例如，在 C++ 的 **map** 中，键的类型总是在编译期检查的。

有一件事很明显：在一种语言已经被广泛应用之后，在其较新的版本中引入任何种类的泛型机制，都会是一项非常非常棘手的任务，并且是一项不付出艰辛就无法完成的任务。在 C++ 中，模版是在其最初的 ISO 版本中就引入的（即便如此，也引发了阵痛，因为在第一个标准 C++ 出现之前，有很多非模版版本在使用），因此实际上模版一直都是这种语言的一部分。在 Java 中，泛型是在这种语言首次发布大约 10 年之后才引入的，因此向泛型迁移的问题特别多，并且对泛型的设计产生了明显的影响。其结果就是，程序员将承受这些痛苦，而这一切都是由于 Java 设计者在设计 1.0 版本时所表现出来的短视造成的。当 Java 最初被创建时，它的设计者们当然了解 C++ 的模版，他们甚至考虑将其囊括到 Java 语言中，但是出于这样或那样的原因，他们决定将模版排除在外（其迹象就是他们过于匆忙）。因此，Java 语言和使用它的程序员都将承受这些痛苦。只有时间将会说明 Java 的泛型方式对这种语言所造成的最终影响。某些语言，已经融入了更简洁、影响更小的方式，来实现参数化类型。我们不可能不去想象这样的语言将会成为 Java 的继任者，因为它们采用的方式，与 C++ 通过 C 来实现的方式相同：按原样使用它，然后对其进行改进。

21.18 进阶阅读

泛型的入门文档是《Generics in the Java Programming Language》，作者是 Gilad Bracha，可以从 http://java.oracle.com 获取。

Angelika Langer 的《Java Generics FAQs》是一份非常有帮助的资料，可以从 http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html 获取。

你可以从《Adding Wildcards to the Java Programming Language》中学到更多关于通配符的知识，作者是 Torgerson、Ernst、Hansen、von der Ahe、Bracha 和 Gafter，地址是 http://www.jot.fm/issues/issue_2004_12/article5。

Neal After 对于 Java 问题(尤其是擦除)的看法可以从这里找到:http://www.infoq.com/articles/neal-

gafter-on-java。

能力固然重要，但并不能代表全部；

22 数组

> 在 初始化和清理 一章的最后，你已经学过如何定义和初始化一个数组。

简单来看，数组需要你去创建和初始化，你可以通过下标对数组元素进行访问，数组的大小不会改变。大多数时候你只需要知道这些，但有时候你必须在数组上进行更复杂的操作，你也可能需要在数组和更加灵活的 **集合** (Collection) 之间做出评估。因此本章我们将对数组进行更加深入的分析。

注意：随着 Java Collection 和 Stream 类中高级功能的不断增加，日常编程中使用数组的需求也在变少，所以你暂且可以放心地略读甚至跳过这一章。但是，即使你自己避免使用数组，也总会有需要阅读别人数组代码的那一天。那时候，本章依然在这里等着你来翻阅。

22.1 数组特性

明明还有很多其他的办法来保存对象，那么是什么令数组如此特别?

将数组和其他类型的集合区分开来的原因有三：效率，类型，保存基本数据类型的能力。在 Java 中，使用数组存储和随机访问对象引用序列是非常高效的。数组是简单的线性序列，这使得对元素的访问变得非常快。然而这种高速也是有代价的，代价就是数组对象的大小是固定的，且在该数组的生存期内不能更改。

速度通常并不是问题，如果有问题，你保存和检索对象的方式也很少是罪魁祸首。你应该总是从 **ArrayList** (来自 集合) 开始，它将数组封装起来。必要时，它会自动分配更多的数组空间，创建新数组，并将旧数组中的引用移动到新数组。这种灵活性需要开销，所以一个 **ArrayList** 的效率不如数组。在极少的情况下效率会成为问题，所以这种时候你可以直接使用数组。

数组和集合 (Collections) 都不能滥用。不管你使用数组还是集合，如果你越界，你都会得到一个 **RuntimeException** 的异常提醒，这表明你的程序中存在错误。

在泛型前，其他的集合类以一种宽泛的方式处理对象（就好像它们没有特定类型一样）。事实上，这些集合类把保存对象的类型默认为 **Object**，也就是 Java 中所有类的基类。而数组是优于 **预泛型** (pre-generic) 集合类的，因为你创建一个数组就可以保存特定类型的数据。这意味着你获得了一个编译时的类型检查，而这可以防止你插入错误的数据类型，或者搞错你正在提取的数据类型。

当然，不管在编译时还是运行时，Java 都会阻止你犯向对象发送不正确消息的错误。然而不管怎样，使用数组都不会有更大的风险。比较好的地方在于，如果编译器报错，最终的用户更容易理解抛出

栈和队列的区别是啥? 吃多了拉就是队列，吃多了吐就是栈;

异常的含义。

一个数组可以保存基本数据类型，而一个预泛型的集合不可以。然而对于泛型而言，集合可以指定和检查他们保存对象的类型，而通过 **自动装箱** (autoboxing) 机制，集合表现地就像它们可以保存基本数据类型一样，因为这种转换是自动的。

下面给出一例用于比较数组和泛型集合：

```
// arrays/CollectionComparison.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

class BerylliumSphere {
  private static long counter;
  private final long id = counter++;
  @Override
  public String toString() {
    return "Sphere " + id;
  }
}

public class CollectionComparison {
  public static void main(String[] args) {
    BerylliumSphere[] spheres =
      new BerylliumSphere[10];
    for(int i = 0; i < 5; i++)
      spheres[i] = new BerylliumSphere();
    show(spheres);
    System.out.println(spheres[4]);

    List<BerylliumSphere> sphereList = Suppliers.create(
      ArrayList::new, BerylliumSphere::new, 5);
    System.out.println(sphereList);
    System.out.println(sphereList.get(4));

    int[] integers = { 0, 1, 2, 3, 4, 5 };
    show(integers);
    System.out.println(integers[4]);

    List<Integer> intList = new ArrayList<>(
      Arrays.asList(0, 1, 2, 3, 4, 5));
    intList.add(97);
    System.out.println(intList);
    System.out.println(intList.get(4));
  }
}
/* Output:
```

```
[Sphere 0, Sphere 1, Sphere 2, Sphere 3, Sphere 4,
null, null, null, null, null]
Sphere 4
[Sphere 5, Sphere 6, Sphere 7, Sphere 8, Sphere 9]
Sphere 9
[0, 1, 2, 3, 4, 5]
4
[0, 1, 2, 3, 4, 5, 97]
4
*/
```

Suppliers.create() 方法在泛型一章中被定义。上面两种保存对象的方式都是有类型检查的，唯一比较明显的区别就是数组使用 **[ ]** 来随机存取元素，而一个 **List** 使用诸如 `add()` 和 `get()` 等方法。数组和 **ArrayList** 之间的相似是设计者有意为之，所以在概念上，两者很容易切换。但是就像你在集合中看到的，集合的功能明显多于数组。随着 Java 自动装箱技术的出现，通过集合使用基本数据类型几乎和通过数组一样简单。数组唯一剩下的优势就是效率。然而，当你解决一个更加普遍的问题时，数组可能限制太多，这种情形下，您可以使用集合类。

22.1.1 用于显示数组的实用程序

在本章中，我们处处都要显示数组。Java 提供了 **Arrays.toString()** 来将数组转换为可读字符串，然后可以在控制台上显示。然而这种方式视觉上噪音太大，所以我们创建一个小的库来完成这项工作。

```java
// onjava/ArrayShow.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
package onjava;
import java.util.*;

public interface ArrayShow {
  static void show(Object[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(boolean[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(byte[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(char[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(short[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(int[] a) {
```

```
    System.out.println(Arrays.toString(a));
  }
  static void show(long[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(float[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(double[] a) {
    System.out.println(Arrays.toString(a));
  }
  // Start with a description:
  static void show(String info, Object[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, boolean[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, byte[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, char[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, short[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, int[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, long[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, float[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, double[] a) {
    System.out.print(info + ": ");
    show(a);
  }
}
```

第一个方法适用于对象数组，包括那些包装基本数据类型的数组。所有的方法重载对于不同的数据

类型是必要的。

第二组重载方法可以让你显示带有信息 **字符串**前缀的数组。

为了简单起见，你通常可以静态地导入它们。

22.2 一等对象

不管你使用的什么类型的数组，数组中的数据集实际上都是对堆中真正对象的引用。数组是保存指向其他对象的引用的对象，数组可以隐式地创建，作为数组初始化语法的一部分，也可以显式地创建，比如使用一个 **new** 表达式。数组对象的一部分（事实上，你唯一可以使用的方法）就是只读的 **length** 成员函数，它能告诉你数组对象中可以存储多少元素。**[ ]** 语法是你访问数组对象的唯一方式。

下面的例子总结了初始化数组的多种方式，并且展示了如何给不同的数组对象分配数组引用。同时也可以看出对象数组和基元数组在使用上是完全相同的。唯一的不同之处就是对象数组存储的是对象的引用，而基元数组则直接存储基本数据类型的值。

```
// arrays/ArrayOptions.java
// Initialization & re-assignment of arrays
import java.util.*;
import static onjava.ArrayShow.*;

public class ArrayOptions {
  public static void main(String[] args) {
    // Arrays of objects:
    BerylliumSphere[] a; // Uninitialized local
    BerylliumSphere[] b = new BerylliumSphere[5];

    // The references inside the array are
    // automatically initialized to null:
    show("b", b);
    BerylliumSphere[] c = new BerylliumSphere[4];
    for(int i = 0; i < c.length; i++)
      if(c[i] == null) // Can test for null reference
        c[i] = new BerylliumSphere();

    // Aggregate initialization:
    BerylliumSphere[] d = {
      new BerylliumSphere(),
      new BerylliumSphere(),
      new BerylliumSphere()
    };

    // Dynamic aggregate initialization:
    a = new BerylliumSphere[]{
      new BerylliumSphere(), new BerylliumSphere(),
    };
    // (Trailing comma is optional)
```

```java

    System.out.println("a.length = " + a.length);
    System.out.println("b.length = " + b.length);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);

    // Arrays of primitives:
    int[] e; // Null reference
    int[] f = new int[5];

    // The primitives inside the array are
    // automatically initialized to zero:
    show("f", f);
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };

    //  Compile error: variable e not initialized:
    //- System.out.println("e.length = " + e.length);
    System.out.println("f.length = " + f.length);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[]{ 1, 2 };
    System.out.println("e.length = " + e.length);
  }
}
/* Output:
b: [null, null, null, null, null]
a.length = 2
b.length = 5
c.length = 4
d.length = 3
a.length = 3
f: [0, 0, 0, 0, 0]
f.length = 5
g.length = 4
h.length = 3
e.length = 3
e.length = 2
*/
```

数组 **a** 是一个未初始化的本地变量,编译器不会允许你使用这个引用直到你正确地对其进行初始化。数组 **b** 被初始化成一系列指向 **BerylliumSphere** 对象的引用，但是并没有真正的 **BerylliumSphere** 对象被存储在数组中。尽管你仍然可以获得这个数组的大小，因为 **b** 指向合法对象。这带来了一个小问

题：你无法找出到底有多少元素存储在数组中，因为 **length** 只能告诉你数组可以存储多少元素；这就是说，数组对象的大小并不是真正存储在数组中对象的个数。然而，当你创建一个数组对象，其引用将自动初始化为 **null**，因此你可以通过检查特定数组元素中的引用是否为 **null** 来判断其中是否有对象。基元数组也有类似的机制，比如自动将数值类型初始化为 **0**，char 型初始化为 **(char)0**，布尔类型初始化为 **false**。

数组 **c** 展示了创建数组对象后给数组中各元素分配 **BerylliumSphere** 对象。数组 **d** 展示了创建数组对象的聚合初始化语法（隐式地使用 **new** 在堆中创建对象，就像 **c** 一样）并且初始化成 **BeryliumSphere** 对象，这一切都在一条语句中完成。

下一个数组初始化可以被看做是一个"动态聚合初始化"。**d** 使用的聚合初始化必须在 **d** 定义处使用，但是使用第二种语法，你可以在任何地方创建和初始化数组对象。例如，假设 **hide()** 是一个需要使用一系列的 **BeryliumSphere** 对象。你可以这样调用它：

```
hide(d);
```

你也可以动态地创建你用作参数传递的数组：

```
hide(new BerylliumSphere[]{
    new BerlliumSphere(),
    new BerlliumSphere()
});
```

很多情况下这种语法写代码更加方便。

表达式：

```
a = d;
```

显示了你如何获取指向一个数组对象的引用并将其分配给另一个数组对象。就像你可以处理其他类型的对象引用。现在 **a** 和 **d** 都指向了堆中的同一个数组对象。

ArrayOptions.java 的第二部分展示了基元数组的语法就像对象数组一样，除了基元数组直接保存基本数据类型的值。

22.3 返回数组

假设你写了一个方法，这个方法不是返回一个元素，而是返回多个元素。对 C++/C 这样的语言来说这是很困难的，因为你无法返回一个数组，只能是返回一个指向数组的指针。这会带来一些问题，因为对数组生存期的控制变得很混乱，这会导致内存泄露。

而在 Java 中，你只需返回数组，你永远不用为数组担心，只要你需要它，它就可用，垃圾收集器会在你用完后把它清理干净。

下面，我们返回一个 **字符串**数组：

是否使用框架，使用什么样的框架，这都不是重要的，重要的是我们要有框架的意识；

```java
// arrays/IceCreamFlavors.java
// Returning arrays from methods
import java.util.*;
import static onjava.ArrayShow.*;

public class IceCreamFlavors {
  private static SplittableRandom rand =
    new SplittableRandom(47);
  static final String[] FLAVORS = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
    "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  public static String[] flavorSet(int n) {
    if(n > FLAVORS.length)
      throw new IllegalArgumentException("Set too big");
    String[] results = new String[n];
    boolean[] picked = new boolean[FLAVORS.length];
    for(int i = 0; i < n; i++) {
      int t;
      do
        t = rand.nextInt(FLAVORS.length);
      while(picked[t]);
      results[i] = FLAVORS[t];
      picked[t] = true;
    }
    return results;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 7; i++)
      show(flavorSet(3));
  }
}
/* Output:
[Praline Cream, Mint Chip, Vanilla Fudge Swirl]
[Strawberry, Vanilla Fudge Swirl, Mud Pie]
[Chocolate, Strawberry, Vanilla Fudge Swirl]
[Rum Raisin, Praline Cream, Chocolate]
[Mint Chip, Rum Raisin, Mocha Almond Fudge]
[Mocha Almond Fudge, Mud Pie, Vanilla Fudge Swirl]
[Mocha Almond Fudge, Mud Pie, Mint Chip]
*/
```

flavorset() 创建名为 **results** 的 **String** 类型的数组。这个数组的大小 **n** 取决于你传进方法的参数。然后从数组 **FLAVORS** 中随机选择 flavors 并且把它们放进 **results** 里并返回。返回一个数组就像返回其他任何对象一样，实际上返回的是引用。数组是在 **flavorSet()** 中或者是在其他什么地方创建的并不重要。垃圾收集器会清理你用完的数组，你需要的数组则会保留。

如果你必须要返回一系列不同类型的元素，你可以使用 泛型 中介绍的 **元组**。

注意，当 **flavorSet()** 随机选择 flavors，它应该确保某个特定的选项没被选中。这在一个 **do** 循环中执行，它将一直做出随机选择直到它发现一个元素不在 **picked** 数组中。(一个字符串

比较将显示出随机选中的元素是不是已经存在于 **results** 数组中)。如果成功了，它将添加条目并且寻找下一个（**i** 递增)。输出结果显示 **flavorSet()** 每一次都是按照随机顺序选择 flavors。

一直到现在，随机数都是通过 **java.util.Random** 类生成的，这个类从 Java 1.0 就有，甚至更新过以提供 Java 8 流。现在我们可以介绍 Java 8 中的 **SplittableRandom** , 它不仅能在并行操作使用(你最终会学到)，而且提供了一个高质量的随机数。这本书的剩余部分都使用 **SplittableRandom** 。

22.4 多维数组

要创建多维的基元数组，你要用大括号来界定数组中的向量:

```
// arrays/MultidimensionalPrimitiveArray.java
import java.util.*;

public class MultidimensionalPrimitiveArray {
  public static void main(String[] args) {
    int[][] a = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[1, 2, 3], [4, 5, 6]]
*/。
```

每个嵌套的大括号都代表了数组的一个维度。

这个例子使用 **Arrays.deepToString()** 方法，将多维数组转换成 **String** 类型，就像输出中显示的那样。

你也可以使用 **new** 分配数组。这是一个使用 **new** 表达式分配的三维数组:

```
// arrays/ThreeDWithNew.java
import java.util.*;

public class ThreeDWithNew {
  public static void main(String[] args) {
    // 3-D array with fixed length:
    int[][][] a = new int[2][2][4];
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0,
```

```
0]]]
*/
```

倘若你不对基元数组进行显式的初始化，它的值会自动初始化。而对象数组将被初始化为 **null** 。

组成矩阵的数组中每一个向量都可以是任意长度的（这叫做不规则数组）：

```
// arrays/RaggedArray.java
import java.util.*;

public class RaggedArray {
  static int val = 1;
  public static void main(String[] args) {
    SplittableRandom rand = new SplittableRandom(47);
    // 3-D array with varied-length vectors:
    int[][][] a = new int[rand.nextInt(7)][][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new int[rand.nextInt(5)][];
      for(int j = 0; j < a[i].length; j++) {
        a[i][j] = new int[rand.nextInt(5)];
        Arrays.setAll(a[i][j], n -> val++); // [1]
      }
    }
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[[1], []], [[2, 3, 4, 5], [6]], [[7, 8, 9], [10, 11,
12], []]]
*/
```

第一个 **new** 创建了一个数组，这个数组首元素长度随机，其余的则不确定。第二个 **new** 在 for 循环中给数组填充了第二个元素，第三个 **new** 为数组的最后一个索引填充元素。

- **1** Java 8 增加了 **Arrays.setAll()** 方法，其使用生成器来生成插入数组中的值。此生成器符合函数式接口 **IntUnaryOperator** ，只使用一个非 **默认**的方法 **ApplyAsint(int 操作数)** 。**Arrays.setAll()** 传递当前数组索引作为操作数，因此一个选项是提供 **n -> n** 的 lambda 表达式来显示数组的索引（在上面的代码中很容易尝试）。这里，我们忽略索引，只是插入递增计数器的值。

非基元的对象数组也可以定义为不规则数组。这里，我们收集了许多使用大括号的 **new** 表达式：

```
// arrays/MultidimensionalObjectArrays.java
import java.util.*;

public class MultidimensionalObjectArrays {
  public static void main(String[] args) {
    BerylliumSphere[][] spheres = {
      { new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
```

```
        new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
    };
    System.out.println(Arrays.deepToString(spheres));
  }
}
/* Output:
[[Sphere 0, Sphere 1], [Sphere 2, Sphere 3, Sphere 4,
Sphere 5], [Sphere 6, Sphere 7, Sphere 8, Sphere 9,
Sphere 10, Sphere 11, Sphere 12, Sphere 13]]
*/
```

数组初始化时使用自动装箱技术:

```
// arrays/AutoboxingArrays.java
import java.util.*;

public class AutoboxingArrays {
  public static void main(String[] args) {
    Integer[][] a = { // Autoboxing:
      { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
      { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
      { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
      { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
    };
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25,
26, 27, 28, 29, 30], [51, 52, 53, 54, 55, 56, 57, 58,
59, 60], [71, 72, 73, 74, 75, 76, 77, 78, 79, 80]]
*/
```

以下是如何逐个构建非基元的对象数组:

```
// arrays/AssemblingMultidimensionalArrays.java
// Creating multidimensional arrays
import java.util.*;

public class AssemblingMultidimensionalArrays {
  public static void main(String[] args) {
    Integer[][] a;
    a = new Integer[3][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer[3];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = i * j; // Autoboxing
```

```
    }
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
*/
```

i * j 在这里只是为了向 **Integer** 中添加有趣的值。

Arrays.deepToString() 方法同时适用于基元数组和对象数组：

```
// arrays/MultiDimWrapperArray.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Multidimensional arrays of "wrapper" objects
import java.util.*;

public class MultiDimWrapperArray {
  public static void main(String[] args) {
    Integer[][] a1 = { // Autoboxing
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    Double[][][] a2 = { // Autoboxing
      { { 1.1, 2.2 }, { 3.3, 4.4 } },
      { { 5.5, 6.6 }, { 7.7, 8.8 } },
      { { 9.9, 1.2 }, { 2.3, 3.4 } },
    };
    String[][] a3 = {
      { "The", "Quick", "Sly", "Fox" },
      { "Jumped", "Over" },
      { "The", "Lazy", "Brown", "Dog", "&", "friend" },
    };
    System.out.println(
      "a1: " + Arrays.deepToString(a1));
    System.out.println(
      "a2: " + Arrays.deepToString(a2));
    System.out.println(
      "a3: " + Arrays.deepToString(a3));
  }
}
/* Output:
a1: [[1, 2, 3], [4, 5, 6]]
a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7,
8.8]], [[9.9, 1.2], [2.3, 3.4]]]
a3: [[The, Quick, Sly, Fox], [Jumped, Over], [The,
Lazy, Brown, Dog, &, friend]]
*/
```

同样的，在 **Integer** 和 **Double** 数组中，自动装箱可为你创建包装器对象。

22.5 泛型数组

一般来说，数组和泛型并不能很好的结合。你不能实例化参数化类型的数组：

```
Peel<Banana>[] peels = new Peel<Banana>[10]; // Illegal
```

类型擦除需要删除参数类型信息，而且数组必须知道它们所保存的确切类型，以强制保证类型安全。

但是，可以参数化数组本身的类型：

```
// arrays/ParameterizedArrayType.java

class ClassParameter<T> {
  public T[] f(T[] arg) { return arg; }
}

class MethodParameter {
  public static <T> T[] f(T[] arg) { return arg; }
}

public class ParameterizedArrayType {
  public static void main(String[] args) {
    Integer[] ints = { 1, 2, 3, 4, 5 };
    Double[] doubles = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Integer[] ints2 =
      new ClassParameter<Integer>().f(ints);
    Double[] doubles2 =
      new ClassParameter<Double>().f(doubles);
    ints2 = MethodParameter.f(ints);
    doubles2 = MethodParameter.f(doubles);
  }
}
```

比起使用参数化类，使用参数化方法很方便。您不必为应用它的每个不同类型都实例化一个带有参数的类，但是可以使它成为 **静态**的。你不能总是选择使用参数化方法而不用参数化的类，但通常参数化方法是更好的选择。

你不能创建泛型类型的数组，这种说法并不完全正确。是的，编译器不会让你 *实例化*一个泛型的数组。但是，它将允许您创建对此类数组的引用。例如：

```
List<String>[] ls;
```

无可争议的，这可以通过编译。尽管不能创建包含泛型的实际数组对象，但是你可以创建一个非泛型的数组并对其进行强制类型转换：

不要担心它能否正常工作。如果一切正常，那么你就会失去工作。

```java
// arrays/ArrayOfGenerics.java
import java.util.*;

public class ArrayOfGenerics {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    List<String>[] ls;
    List[] la = new List[10];
    ls = (List<String>[])la; // Unchecked cast
    ls[0] = new ArrayList<>();

    //- ls[1] = new ArrayList<Integer>();
    // error: incompatible types: ArrayList<Integer>
    // cannot be converted to List<String>
    //     ls[1] = new ArrayList<Integer>();
    //             ^

    // The problem: List<String> is a subtype of Object
    Object[] objects = ls; // So assignment is OK
    // Compiles and runs without complaint:
    objects[1] = new ArrayList<>();

    // However, if your needs are straightforward it is
    // possible to create an array of generics, albeit
    // with an "unchecked cast" warning:
    List<BerylliumSphere>[] spheres =
      (List<BerylliumSphere>[])new List[10];
    Arrays.setAll(spheres, n -> new ArrayList<>());
  }
}
```

一旦你有了对 **List[]** 的引用，你会发现多了一些编译时检查。问题是数组是协变的，所以 **List[]** 也是一个 **Object[]**，你可以用这来将 **ArrayList** 分配进你的数组，在编译或者运行时都不会出错。

如果你知道你不会进行向上类型转换，你的需求相对简单，那么可以创建一个泛型数组，它将提供基本的编译时类型检查。然而，一个泛型 **Collection** 实际上是一个比泛型数组更好的选择。

一般来说，您会发现泛型在类或方法的边界上是有效的。在内部，擦除常常会使泛型不可使用。所以，就像下面的例子，不能创建泛型类型的数组：

```java
// arrays/ArrayOfGenericType.java

public class ArrayOfGenericType<T> {
  T[] array; // OK
  @SuppressWarnings("unchecked")
  public ArrayOfGenericType(int size) {
    // error: generic array creation:
    //- array = new T[size];
    array = (T[])new Object[size]; // unchecked cast
  }
```

The trouble with programmers is that you can never tell what a programmer is doing until it’s too late. — Seymour Cray

```
  // error: generic array creation:
  //- public <U> U[] makeArray() { return new U[10]; }
}
```

擦除再次从中作梗，这个例子试图创建已经擦除的类型数组，因此它们是未知的类型。你可以创建一个 **对象**数组，然后对其进行强制类型转换，但如果没有 **@SuppressWarnings** 注释，你将会得到一个 “unchecked” 警告，因为数组实际上不真正支持而且将对类型 **T** 动态检查。这就是说，如果我创建了一个 **String[]** , Java 将在编译时和运行时强制执行，我只能在数组中放置字符串对象。然而，如果我创建一个 **Object[]** , 我可以把除了基元类型外的任何东西放入数组。

22.6 Arrays 的 fill 方法

通常情况下，当对数组和程序进行实验时，能够很轻易地生成充满测试数据的数组是很有帮助的。Java 标准库 **Arrays** 类包括一个普通的 **fill()** 方法，该方法将单个值复制到整个数组，或者在对象数组的情况下，将相同的引用复制到整个数组：

```java
// arrays/FillingArrays.java
// Using Arrays.fill()
import java.util.*;
import static onjava.ArrayShow.*;

public class FillingArrays {
  public static void main(String[] args) {
    int size = 6;
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays.fill(a1, true);
    show("a1", a1);
    Arrays.fill(a2, (byte)11);
    show("a2", a2);
    Arrays.fill(a3, 'x');
    show("a3", a3);
    Arrays.fill(a4, (short)17);
    show("a4", a4);
    Arrays.fill(a5, 19);
    show("a5", a5);
    Arrays.fill(a6, 23);
    show("a6", a6);
    Arrays.fill(a7, 29);
```

```
    show("a7", a7);
    Arrays.fill(a8, 47);
    show("a8", a8);
    Arrays.fill(a9, "Hello");
    show("a9", a9);
    // Manipulating ranges:
    Arrays.fill(a9, 3, 5, "World");
    show("a9", a9);
  }
}gedan
/* Output:
a1: [true, true, true, true, true, true]
a2: [11, 11, 11, 11, 11, 11]
a3: [x, x, x, x, x, x]
a4: [17, 17, 17, 17, 17, 17]
a5: [19, 19, 19, 19, 19, 19]
a6: [23, 23, 23, 23, 23, 23]
a7: [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
a8: [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
a9: [Hello, Hello, Hello, Hello, Hello, Hello]
a9: [Hello, Hello, Hello, World, World, Hello]
*/
```

你既可以填充整个数组，也可以像最后两个语句所示，填充一系列的元素。但是由于你只能使用单个值调用 **Arrays.fill()** ，因此结果并非特别有用。

22.7 Arrays 的 setAll 方法

在 Java 8 中，在 **RaggedArray.java** 中引入并在 **ArrayOfGenerics.java.Array.setAll()** 中重用。它使用一个生成器并生成不同的值，可以选择基于数组的索引元素（通过访问当前索引，生成器可以读取数组值并对其进行修改）。**static Arrays.setAll()** 的重载签名为：

- **void setAll(int[] a, IntUnaryOperator gen)**
- **void setAll(long[] a, IntToLongFunction gen)**
- **void setAll(double[] a, IntToDoubleFunctiongen)**
- **void setAll(T[] a, IntFunction<? extendsT> gen)**

除了 **int** , **long** , **double** 有特殊的版本，其他的一切都由泛型版本处理。生成器不是 **Supplier** 因为它们不带参数，并且必须将 **int** 数组索引作为参数。

```
// arrays/SimpleSetAll.java

import java.util.*;
import static onjava.ArrayShow.*;

class Bob {
  final int id;
```

Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris. — Larry Wall

```
  Bob(int n) { id = n; }
  @Override
  public String toString() { return "Bob" + id; }
}

public class SimpleSetAll {
  public static final int SZ = 8;
  static int val = 1;
  static char[] chars = "abcdefghijklmnopqrstuvwxyz"
    .toCharArray();
  static char getChar(int n) { return chars[n]; }
  public static void main(String[] args) {
    int[] ia = new int[SZ];
    long[] la = new long[SZ];
    double[] da = new double[SZ];
    Arrays.setAll(ia, n -> n); // [1]
    Arrays.setAll(la, n -> n);
    Arrays.setAll(da, n -> n);
    show(ia);
    show(la);
    show(da);
    Arrays.setAll(ia, n -> val++); // [2]
    Arrays.setAll(la, n -> val++);
    Arrays.setAll(da, n -> val++);
    show(ia);
    show(la);
    show(da);

    Bob[] ba = new Bob[SZ];
    Arrays.setAll(ba, Bob::new); // [3]
    show(ba);

    Character[] ca = new Character[SZ];
    Arrays.setAll(ca, SimpleSetAll::getChar); // [4]
    show(ca);
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
[1, 2, 3, 4, 5, 6, 7, 8]
[9, 10, 11, 12, 13, 14, 15, 16]
[17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0]
[Bob0, Bob1, Bob2, Bob3, Bob4, Bob5, Bob6, Bob7]
[a, b, c, d, e, f, g, h]
*/
```

- **1** 这里，我们只是将数组索引作为值插入数组。这将自动转化为 **long** 和 **double** 版本。
- **2** 这个函数只需要接受索引就能产生正确结果。这个，我们忽略索引值并且使用 **val** 生成结果。

- **[3]** 方法引用有效，因为 **Bob** 的构造器接收一个 **int** 参数。只要我们传递的函数接收一个 **int** 参数且能产生正确的结果，就认为它完成了工作。
- **[4]** 为了处理除了 **int** , **long** , **double** 之外的基元类型，请为基元创建包装类的数组。然后使用 **setAll()** 的泛型版本。请注意，**getChar** () 生成基元类型，因此这是自动装箱到 **Character** 。

22.8 增量生成

这是一个方法库，用于为不同类型生成增量值。

这些被作为内部类来生成容易记住的名字；比如，为了使用 **Integer** 工具你可以用 **new Conut.Interger()** , 如果你想要使用基本数据类型 **int** 工具，你可以用 **new Count.Pint()** (基本类型的名字不能被直接使用，所以它们都在前面添加一个 **P** 来表示基本数据类型'primitive', 我们的第一选择是使用基本类型名字后面跟着下划线，比如 **int_** 和 **double_** , 但是这种方式违背 Java 的命名习惯)。每个包装类的生成器都使用 **get()** 方法实现了它的 **Supplier** 。要使用 **Array.setAll()** ，一个重载的 **get(int n)** 方法要接受（并忽略）其参数，以便接受 **setAll()** 传递的索引值。

注意，通过使用包装类的名称作为内部类名，我们必须调用 **java.lang** 包来保证我们可以使用实际包装类的名字:

```
// onjava/Count.java
// Generate incremental values of different types
package onjava;
import java.util.*;
import java.util.function.*;
import static onjava.ConvertTo.*;

public interface Count {
  class Boolean
  implements Supplier<java.lang.Boolean> {
    private boolean b = true;
    @Override
    public java.lang.Boolean get() {
      b = !b;
      return java.lang.Boolean.valueOf(b);
    }
    public java.lang.Boolean get(int n) {
      return get();
    }
    public java.lang.Boolean[] array(int sz) {
      java.lang.Boolean[] result =
        new java.lang.Boolean[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
```

```
class Pboolean {
  private boolean b = true;
  public boolean get() {
    b = !b;
    return b;
  }
  public boolean get(int n) { return get(); }
  public boolean[] array(int sz) {
    return primitive(new Boolean().array(sz));
  }
}
class Byte
implements Supplier<java.lang.Byte> {
  private byte b;
  @Override
  public java.lang.Byte get() { return b++; }
  public java.lang.Byte get(int n) {
    return get();
  }
  public java.lang.Byte[] array(int sz) {
    java.lang.Byte[] result =
      new java.lang.Byte[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pbyte {
  private byte b;
  public byte get() { return b++; }
  public byte get(int n) { return get(); }
  public byte[] array(int sz) {
    return primitive(new Byte().array(sz));
  }
}
char[] CHARS =
  "abcdefghijklmnopqrstuvwxyz".toCharArray();
class Character
implements Supplier<java.lang.Character> {
  private int i;
  @Override
  public java.lang.Character get() {
    i = (i + 1) % CHARS.length;
    return CHARS[i];
  }
  public java.lang.Character get(int n) {
    return get();
  }
  public java.lang.Character[] array(int sz) {
    java.lang.Character[] result =
      new java.lang.Character[sz];
    Arrays.setAll(result, n -> get());
```

```
      return result;
    }
  }
  class Pchar {
    private int i;
    public char get() {
      i = (i + 1) % CHARS.length;
      return CHARS[i];
    }
    public char get(int n) { return get(); }
    public char[] array(int sz) {
      return primitive(new Character().array(sz));
    }
  }
  class Short
  implements Supplier<java.lang.Short> {
    short s;
    @Override
    public java.lang.Short get() { return s++; }
    public java.lang.Short get(int n) {
      return get();
    }
    public java.lang.Short[] array(int sz) {
      java.lang.Short[] result =
        new java.lang.Short[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pshort {
    short s;
    public short get() { return s++; }
    public short get(int n) { return get(); }
    public short[] array(int sz) {
      return primitive(new Short().array(sz));
    }
  }
  class Integer
  implements Supplier<java.lang.Integer> {
    int i;
    @Override
    public java.lang.Integer get() { return i++; }
    public java.lang.Integer get(int n) {
      return get();
    }
    public java.lang.Integer[] array(int sz) {
      java.lang.Integer[] result =
        new java.lang.Integer[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
```

```
  }
  class Pint implements IntSupplier {
    int i;
    public int get() { return i++; }
    public int get(int n) { return get(); }
    @Override
    public int getAsInt() { return get(); }
    public int[] array(int sz) {
      return primitive(new Integer().array(sz));
    }
  }
  class Long
  implements Supplier<java.lang.Long> {
    private long l;
    @Override
    public java.lang.Long get() { return l++; }
    public java.lang.Long get(int n) {
      return get();
    }
    public java.lang.Long[] array(int sz) {
      java.lang.Long[] result =
        new java.lang.Long[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Plong implements LongSupplier {
    private long l;
    public long get() { return l++; }
    public long get(int n) { return get(); }
    @Override
    public long getAsLong() { return get(); }
    public long[] array(int sz) {
      return primitive(new Long().array(sz));
    }
  }
  class Float
  implements Supplier<java.lang.Float> {
    private int i;
    @Override
    public java.lang.Float get() {
      return java.lang.Float.valueOf(i++);
    }
    public java.lang.Float get(int n) {
      return get();
    }
    public java.lang.Float[] array(int sz) {
      java.lang.Float[] result =
        new java.lang.Float[sz];
      Arrays.setAll(result, n -> get());
      return result;
```

```
    }
  }
  class Pfloat {
    private int i;
    public float get() { return i++; }
    public float get(int n) { return get(); }
    public float[] array(int sz) {
      return primitive(new Float().array(sz));
    }
  }
  class Double
  implements Supplier<java.lang.Double> {
    private int i;
    @Override
    public java.lang.Double get() {
      return java.lang.Double.valueOf(i++);
    }
    public java.lang.Double get(int n) {
      return get();
    }
    public java.lang.Double[] array(int sz) {
      java.lang.Double[] result =
        new java.lang.Double[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pdouble implements DoubleSupplier {
    private int i;
    public double get() { return i++; }
    public double get(int n) { return get(); }
    @Override
    public double getAsDouble() { return get(0); }
    public double[] array(int sz) {
      return primitive(new Double().array(sz));
    }
  }
}
```

对于 **int** ， **long** ， **double** 这三个有特殊 **Supplier** 接口的原始数据类型来说， **Pint** ， **Plong** 和 **Pdouble** 实现了这些接口。

这里是对 **Count** 的测试，这同样给我们提供了如何使用它的例子：

```
// arrays/TestCount.java
// Test counting generators
import java.util.*;
import java.util.stream.*;
import onjava.*;
import static onjava.ArrayShow.*;

```

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along wound destroy civilization. — Gerald Weinberg

```java
public class TestCount {
  static final int SZ = 5;
  public static void main(String[] args) {
    System.out.println("Boolean");
    Boolean[] a1 = new Boolean[SZ];
    Arrays.setAll(a1, new Count.Boolean()::get);
    show(a1);
    a1 = Stream.generate(new Count.Boolean())
      .limit(SZ + 1).toArray(Boolean[]::new);
    show(a1);
    a1 = new Count.Boolean().array(SZ + 2);
    show(a1);
    boolean[] a1b =
      new Count.Pboolean().array(SZ + 3);
    show(a1b);

    System.out.println("Byte");
    Byte[] a2 = new Byte[SZ];
    Arrays.setAll(a2, new Count.Byte()::get);
    show(a2);
    a2 = Stream.generate(new Count.Byte())
      .limit(SZ + 1).toArray(Byte[]::new);
    show(a2);
    a2 = new Count.Byte().array(SZ + 2);
    show(a2);
    byte[] a2b = new Count.Pbyte().array(SZ + 3);
    show(a2b);

    System.out.println("Character");
    Character[] a3 = new Character[SZ];
    Arrays.setAll(a3, new Count.Character()::get);
    show(a3);
    a3 = Stream.generate(new Count.Character())
      .limit(SZ + 1).toArray(Character[]::new);
    show(a3);
    a3 = new Count.Character().array(SZ + 2);
    show(a3);
    char[] a3b = new Count.Pchar().array(SZ + 3);
    show(a3b);

    System.out.println("Short");
    Short[] a4 = new Short[SZ];
    Arrays.setAll(a4, new Count.Short()::get);
    show(a4);
    a4 = Stream.generate(new Count.Short())
      .limit(SZ + 1).toArray(Short[]::new);
    show(a4);
    a4 = new Count.Short().array(SZ + 2);
    show(a4);
    short[] a4b = new Count.Pshort().array(SZ + 3);
    show(a4b);
```

```java

    System.out.println("Integer");
    int[] a5 = new int[SZ];
    Arrays.setAll(a5, new Count.Integer()::get);
    show(a5);
    Integer[] a5b =
      Stream.generate(new Count.Integer())
        .limit(SZ + 1).toArray(Integer[]::new);
    show(a5b);
    a5b = new Count.Integer().array(SZ + 2);
    show(a5b);
    a5 = IntStream.generate(new Count.Pint())
      .limit(SZ + 1).toArray();
    show(a5);
    a5 = new Count.Pint().array(SZ + 3);
    show(a5);

    System.out.println("Long");
    long[] a6 = new long[SZ];
    Arrays.setAll(a6, new Count.Long()::get);
    show(a6);
    Long[] a6b = Stream.generate(new Count.Long())
      .limit(SZ + 1).toArray(Long[]::new);
    show(a6b);
    a6b = new Count.Long().array(SZ + 2);
    show(a6b);
    a6 = LongStream.generate(new Count.Plong())
      .limit(SZ + 1).toArray();
    show(a6);
    a6 = new Count.Plong().array(SZ + 3);
    show(a6);

    System.out.println("Float");
    Float[] a7 = new Float[SZ];
    Arrays.setAll(a7, new Count.Float()::get);
    show(a7);
    a7 = Stream.generate(new Count.Float())
      .limit(SZ + 1).toArray(Float[]::new);
    show(a7);
    a7 = new Count.Float().array(SZ + 2);
    show(a7);
    float[] a7b = new Count.Pfloat().array(SZ + 3);
    show(a7b);

    System.out.println("Double");
    double[] a8 = new double[SZ];
    Arrays.setAll(a8, new Count.Double()::get);
    show(a8);
    Double[] a8b =
      Stream.generate(new Count.Double())
        .limit(SZ + 1).toArray(Double[]::new);
```

```
    show(a8b);
    a8b = new Count.Double().array(SZ + 2);
    show(a8b);
    a8 = DoubleStream.generate(new Count.Pdouble())
      .limit(SZ + 1).toArray();
    show(a8);
    a8 = new Count.Pdouble().array(SZ + 3);
    show(a8);
  }
}
/* Output:
Boolean
[false, true, false, true, false]
[false, true, false, true, false, true]
[false, true, false, true, false, true, false]
[false, true, false, true, false, true, false, true]
Byte
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
Character
[b, c, d, e, f]
[b, c, d, e, f, g]
[b, c, d, e, f, g, h]
[b, c, d, e, f, g, h, i]
Short
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
Integer
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6, 7]
Long
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6, 7]
Float
[0.0, 1.0, 2.0, 3.0, 4.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
Double
[0.0, 1.0, 2.0, 3.0, 4.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
```

```
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
*/
```

注意到原始数组类型 **int[]**，**long[]**，**double[]** 可以直接被 **Arrays.setAll()** 填充，但是其他的原始类型都要求用包装器类型的数组。

通过 **Stream.generate()** 创建的包装数组显示了 **toArray**（）的重载用法，在这里你应该提供给它要创建的数组类型的构造器。

22.9 随机生成

我们可以按照 **Count.java** 的结构创建一个生成随机值的工具：

```java
// onjava/Rand.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Generate random values of different types
package onjava;
import java.util.*;
import java.util.function.*;
import static onjava.ConvertTo.*;

public interface Rand {
  int MOD = 10_000;
  class Boolean
  implements Supplier<java.lang.Boolean> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Boolean get() {
      return r.nextBoolean();
    }
    public java.lang.Boolean get(int n) {
      return get();
    }
    public java.lang.Boolean[] array(int sz) {
      java.lang.Boolean[] result =
        new java.lang.Boolean[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pboolean {
    public boolean[] array(int sz) {
      return primitive(new Boolean().array(sz));
    }
```

It’s a curious thing about our industry: not only do we not learn from our mistakes, we also don’t learn from our successes.
— Keith Braithwaite

```
  }
  class Byte
  implements Supplier<java.lang.Byte> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Byte get() {
      return (byte)r.nextInt(MOD);
    }
    public java.lang.Byte get(int n) {
      return get();
    }
    public java.lang.Byte[] array(int sz) {
      java.lang.Byte[] result =
        new java.lang.Byte[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pbyte {
    public byte[] array(int sz) {
      return primitive(new Byte().array(sz));
    }
  }
  class Character
  implements Supplier<java.lang.Character> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Character get() {
      return (char)r.nextInt('a', 'z' + 1);
    }
    public java.lang.Character get(int n) {
      return get();
    }
    public java.lang.Character[] array(int sz) {
      java.lang.Character[] result =
        new java.lang.Character[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pchar {
    public char[] array(int sz) {
      return primitive(new Character().array(sz));
    }
  }
  class Short
  implements Supplier<java.lang.Short> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Short get() {
      return (short)r.nextInt(MOD);
```

```
    }
    public java.lang.Short get(int n) {
      return get();
    }
    public java.lang.Short[] array(int sz) {
      java.lang.Short[] result =
        new java.lang.Short[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pshort {
    public short[] array(int sz) {
      return primitive(new Short().array(sz));
    }
  }
  class Integer
  implements Supplier<java.lang.Integer> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Integer get() {
      return r.nextInt(MOD);
    }
    public java.lang.Integer get(int n) {
      return get();
    }
    public java.lang.Integer[] array(int sz) {
      int[] primitive = new Pint().array(sz);
      java.lang.Integer[] result =
        new java.lang.Integer[sz];
      for(int i = 0; i < sz; i++)
        result[i] = primitive[i];
      return result;
    }
  }
  class Pint implements IntSupplier {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public int getAsInt() {
      return r.nextInt(MOD);
    }
    public int get(int n) { return getAsInt(); }
    public int[] array(int sz) {
      return r.ints(sz, 0, MOD).toArray();
    }
  }
  class Long
  implements Supplier<java.lang.Long> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Long get() {
```

```
      return r.nextLong(MOD);
    }
    public java.lang.Long get(int n) {
      return get();
    }
    public java.lang.Long[] array(int sz) {
      long[] primitive = new Plong().array(sz);
      java.lang.Long[] result =
        new java.lang.Long[sz];
      for(int i = 0; i < sz; i++)
        result[i] = primitive[i];
      return result;
    }
  }
  class Plong implements LongSupplier {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public long getAsLong() {
      return r.nextLong(MOD);
    }
    public long get(int n) { return getAsLong(); }
    public long[] array(int sz) {
      return r.longs(sz, 0, MOD).toArray();
    }
  }
  class Float
  implements Supplier<java.lang.Float> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Float get() {
      return (float)trim(r.nextDouble());
    }
    public java.lang.Float get(int n) {
      return get();
    }
    public java.lang.Float[] array(int sz) {
      java.lang.Float[] result =
        new java.lang.Float[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pfloat {
    public float[] array(int sz) {
      return primitive(new Float().array(sz));
    }
  }
  static double trim(double d) {
    return
      ((double)Math.round(d * 1000.0)) / 100.0;
  }
```

```
  class Double
  implements Supplier<java.lang.Double> {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public java.lang.Double get() {
      return trim(r.nextDouble());
    }
    public java.lang.Double get(int n) {
      return get();
    }
    public java.lang.Double[] array(int sz) {
      double[] primitive =
        new Rand.Pdouble().array(sz);
      java.lang.Double[] result =
        new java.lang.Double[sz];
      for(int i = 0; i < sz; i++)
        result[i] = primitive[i];
      return result;
    }
  }
  class Pdouble implements DoubleSupplier {
    SplittableRandom r = new SplittableRandom(47);
    @Override
    public double getAsDouble() {
      return trim(r.nextDouble());
    }
    public double get(int n) {
      return getAsDouble();
    }
    public double[] array(int sz) {
      double[] result = r.doubles(sz).toArray();
      Arrays.setAll(result,
        n -> result[n] = trim(result[n]));
      return result;
    }
  }
  class String
  implements Supplier<java.lang.String> {
    SplittableRandom r = new SplittableRandom(47);
    private int strlen = 7; // Default length
    public String() {}
    public String(int strLength) {
      strlen = strLength;
    }
    @Override
    public java.lang.String get() {
      return r.ints(strlen, 'a', 'z' + 1)
        .collect(StringBuilder::new,
                 StringBuilder::appendCodePoint,
                 StringBuilder::append).toString();
    }
```

```
    public java.lang.String get(int n) {
      return get();
    }
    public java.lang.String[] array(int sz) {
      java.lang.String[] result =
        new java.lang.String[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
}
```

对于除了 **int** 、**long** 和 **double** 之外的所有基本类型元素生成器，只生成数组，而不是 Count 中看到的完整操作集。这只是一个设计选择，因为本书不需要额外的功能。

下面是对所有 **Rand** 工具的测试：

```
// arrays/TestRand.java
// Test random generators
import java.util.*;
import java.util.stream.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class TestRand {
  static final int SZ = 5;
  public static void main(String[] args) {
    System.out.println("Boolean");
    Boolean[] a1 = new Boolean[SZ];
    Arrays.setAll(a1, new Rand.Boolean()::get);
    show(a1);
    a1 = Stream.generate(new Rand.Boolean())
      .limit(SZ + 1).toArray(Boolean[]::new);
    show(a1);
    a1 = new Rand.Boolean().array(SZ + 2);
    show(a1);
    boolean[] a1b =
      new Rand.Pboolean().array(SZ + 3);
    show(a1b);

    System.out.println("Byte");
    Byte[] a2 = new Byte[SZ];
    Arrays.setAll(a2, new Rand.Byte()::get);
    show(a2);
    a2 = Stream.generate(new Rand.Byte())
      .limit(SZ + 1).toArray(Byte[]::new);
    show(a2);
    a2 = new Rand.Byte().array(SZ + 2);
    show(a2);
    byte[] a2b = new Rand.Pbyte().array(SZ + 3);
    show(a2b);
```

```java

    System.out.println("Character");
    Character[] a3 = new Character[SZ];
    Arrays.setAll(a3, new Rand.Character()::get);
    show(a3);
    a3 = Stream.generate(new Rand.Character())
      .limit(SZ + 1).toArray(Character[]::new);
    show(a3);
    a3 = new Rand.Character().array(SZ + 2);
    show(a3);
    char[] a3b = new Rand.Pchar().array(SZ + 3);
    show(a3b);

    System.out.println("Short");
    Short[] a4 = new Short[SZ];
    Arrays.setAll(a4, new Rand.Short()::get);
    show(a4);
    a4 = Stream.generate(new Rand.Short())
      .limit(SZ + 1).toArray(Short[]::new);
    show(a4);
    a4 = new Rand.Short().array(SZ + 2);
    show(a4);
    short[] a4b = new Rand.Pshort().array(SZ + 3);
    show(a4b);

    System.out.println("Integer");
    int[] a5 = new int[SZ];
    Arrays.setAll(a5, new Rand.Integer()::get);
    show(a5);
    Integer[] a5b =
      Stream.generate(new Rand.Integer())
        .limit(SZ + 1).toArray(Integer[]::new);
    show(a5b);
    a5b = new Rand.Integer().array(SZ + 2);
    show(a5b);
    a5 = IntStream.generate(new Rand.Pint())
      .limit(SZ + 1).toArray();
    show(a5);
    a5 = new Rand.Pint().array(SZ + 3);
    show(a5);

    System.out.println("Long");
    long[] a6 = new long[SZ];
    Arrays.setAll(a6, new Rand.Long()::get);
    show(a6);
    Long[] a6b = Stream.generate(new Rand.Long())
      .limit(SZ + 1).toArray(Long[]::new);
    show(a6b);
    a6b = new Rand.Long().array(SZ + 2);
    show(a6b);
    a6 = LongStream.generate(new Rand.Plong())
```

```
      .limit(SZ + 1).toArray();
    show(a6);
    a6 = new Rand.Plong().array(SZ + 3);
    show(a6);

    System.out.println("Float");
    Float[] a7 = new Float[SZ];
    Arrays.setAll(a7, new Rand.Float()::get);
    show(a7);
    a7 = Stream.generate(new Rand.Float())
      .limit(SZ + 1).toArray(Float[]::new);
    show(a7);
    a7 = new Rand.Float().array(SZ + 2);
    show(a7);
    float[] a7b = new Rand.Pfloat().array(SZ + 3);
    show(a7b);

    System.out.println("Double");
    double[] a8 = new double[SZ];
    Arrays.setAll(a8, new Rand.Double()::get);
    show(a8);
    Double[] a8b =
      Stream.generate(new Rand.Double())
        .limit(SZ + 1).toArray(Double[]::new);
    show(a8b);
    a8b = new Rand.Double().array(SZ + 2);
    show(a8b);
    a8 = DoubleStream.generate(new Rand.Pdouble())
      .limit(SZ + 1).toArray();
    show(a8);
    a8 = new Rand.Pdouble().array(SZ + 3);
    show(a8);

    System.out.println("String");
    String[] s = new String[SZ - 1];
    Arrays.setAll(s, new Rand.String()::get);
    show(s);
    s = Stream.generate(new Rand.String())
      .limit(SZ).toArray(String[]::new);
    show(s);
    s = new Rand.String().array(SZ + 1);
    show(s);

    Arrays.setAll(s, new Rand.String(4)::get);
    show(s);
    s = Stream.generate(new Rand.String(4))
      .limit(SZ).toArray(String[]::new);
    show(s);
    s = new Rand.String(4).array(SZ + 1);
    show(s);
  }
```

```
}
/* Output:
Boolean
[true, false, true, true, true]
[true, false, true, true, true, false]
[true, false, true, true, true, false, false]
[true, false, true, true, true, false, false, true]
Byte
[123, 33, 101, 112, 33]
[123, 33, 101, 112, 33, 31]
[123, 33, 101, 112, 33, 31, 0]
[123, 33, 101, 112, 33, 31, 0, -72]
Character
[b, t, p, e, n]
[b, t, p, e, n, p]
[b, t, p, e, n, p, c]
[b, t, p, e, n, p, c, c]
Short
[635, 8737, 3941, 4720, 6177]
[635, 8737, 3941, 4720, 6177, 8479]
[635, 8737, 3941, 4720, 6177, 8479, 6656]
[635, 8737, 3941, 4720, 6177, 8479, 6656, 3768]
Integer
[635, 8737, 3941, 4720, 6177]
[635, 8737, 3941, 4720, 6177, 8479]
[635, 8737, 3941, 4720, 6177, 8479, 6656]
[635, 8737, 3941, 4720, 6177, 8479]
[635, 8737, 3941, 4720, 6177, 8479, 6656, 3768]
Long
[6882, 3765, 692, 9575, 4439]
[6882, 3765, 692, 9575, 4439, 2638]
[6882, 3765, 692, 9575, 4439, 2638, 4011]
[6882, 3765, 692, 9575, 4439, 2638]
[6882, 3765, 692, 9575, 4439, 2638, 4011, 9610]
Float
[4.83, 2.89, 2.9, 1.97, 3.01]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99, 8.28]
Double
[4.83, 2.89, 2.9, 1.97, 3.01]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99, 8.28]
String
[btpenpc, cuxszgv, gmeinne, eloztdv]
[btpenpc, cuxszgv, gmeinne, eloztdv, ewcippc]
[btpenpc, cuxszgv, gmeinne, eloztdv, ewcippc, ygpoalk]
[btpe, npcc, uxsz, gvgm, einn, eelo]
[btpe, npcc, uxsz, gvgm, einn]
```

```
[btpe, npcc, uxsz, gvgm, einn, eelo]
*/
```

注意（除了 **String** 部分之外），这段代码与 **TestCount.java** 中的代码相同，**Count** 被 **Rand** 替换。

22.10 泛型和基本数组

在本章的前面，我们被提醒，泛型不能和基元一起工作。在这种情况下，我们必须从基元数组转换为包装类型的数组，并且还必须从另一个方向转换。下面是一个转换器可以同时对所有类型的数据执行操作：

```java
// onjava/ConvertTo.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
package onjava;

public interface ConvertTo {
  static boolean[] primitive(Boolean[] in) {
    boolean[] result = new boolean[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i]; // Autounboxing
    return result;
  }
  static char[] primitive(Character[] in) {
    char[] result = new char[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static byte[] primitive(Byte[] in) {
    byte[] result = new byte[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static short[] primitive(Short[] in) {
    short[] result = new short[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static int[] primitive(Integer[] in) {
    int[] result = new int[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
```

```
    return result;
  }
  static long[] primitive(Long[] in) {
    long[] result = new long[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static float[] primitive(Float[] in) {
    float[] result = new float[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static double[] primitive(Double[] in) {
    double[] result = new double[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  // Convert from primitive array to wrapped array:
  static Boolean[] boxed(boolean[] in) {
    Boolean[] result = new Boolean[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i]; // Autoboxing
    return result;
  }
  static Character[] boxed(char[] in) {
    Character[] result = new Character[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Byte[] boxed(byte[] in) {
    Byte[] result = new Byte[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Short[] boxed(short[] in) {
    Short[] result = new Short[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Integer[] boxed(int[] in) {
    Integer[] result = new Integer[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
```

```
  static Long[] boxed(long[] in) {
    Long[] result = new Long[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Float[] boxed(float[] in) {
    Float[] result = new Float[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Double[] boxed(double[] in) {
    Double[] result = new Double[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
}
```

primitive() 的每个版本都创建一个准确长度的适当基元数组，然后从包装类的 **in** 数组中复制元素。如果任何包装的数组元素是 **null**，你将得到一个异常（这是合理的—否则无法选择有意义的值进行替换)。注意在这个任务中自动装箱如何发生。

下面是对 **ConvertTo** 中所有方法的测试:

```
// arrays/TestConvertTo.java
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;
import static onjava.ConvertTo.*;

public class TestConvertTo {
  static final int SIZE = 6;
  public static void main(String[] args) {
    Boolean[] a1 = new Boolean[SIZE];
    Arrays.setAll(a1, new Rand.Boolean()::get);
    boolean[] a1p = primitive(a1);
    show("a1p", a1p);
    Boolean[] a1b = boxed(a1p);
    show("a1b", a1b);

    Byte[] a2 = new Byte[SIZE];
    Arrays.setAll(a2, new Rand.Byte()::get);
    byte[] a2p = primitive(a2);
    show("a2p", a2p);
    Byte[] a2b = boxed(a2p);
    show("a2b", a2b);

    Character[] a3 = new Character[SIZE];
```

```
    Arrays.setAll(a3, new Rand.Character()::get);
    char[] a3p = primitive(a3);
    show("a3p", a3p);
    Character[] a3b = boxed(a3p);
    show("a3b", a3b);

    Short[] a4 = new Short[SIZE];
    Arrays.setAll(a4, new Rand.Short()::get);
    short[] a4p = primitive(a4);
    show("a4p", a4p);
    Short[] a4b = boxed(a4p);
    show("a4b", a4b);

    Integer[] a5 = new Integer[SIZE];
    Arrays.setAll(a5, new Rand.Integer()::get);
    int[] a5p = primitive(a5);
    show("a5p", a5p);
    Integer[] a5b = boxed(a5p);
    show("a5b", a5b);

    Long[] a6 = new Long[SIZE];
    Arrays.setAll(a6, new Rand.Long()::get);
    long[] a6p = primitive(a6);
    show("a6p", a6p);
    Long[] a6b = boxed(a6p);
    show("a6b", a6b);

    Float[] a7 = new Float[SIZE];
    Arrays.setAll(a7, new Rand.Float()::get);
    float[] a7p = primitive(a7);
    show("a7p", a7p);
    Float[] a7b = boxed(a7p);
    show("a7b", a7b);

    Double[] a8 = new Double[SIZE];
    Arrays.setAll(a8, new Rand.Double()::get);
    double[] a8p = primitive(a8);
    show("a8p", a8p);
    Double[] a8b = boxed(a8p);
    show("a8b", a8b);
  }
}
/* Output:
a1p: [true, false, true, true, true, false]
a1b: [true, false, true, true, true, false]
a2p: [123, 33, 101, 112, 33, 31]
a2b: [123, 33, 101, 112, 33, 31]
a3p: [b, t, p, e, n, p]
a3b: [b, t, p, e, n, p]
a4p: [635, 8737, 3941, 4720, 6177, 8479]
a4b: [635, 8737, 3941, 4720, 6177, 8479]
```

```
a5p: [635, 8737, 3941, 4720, 6177, 8479]
a5b: [635, 8737, 3941, 4720, 6177, 8479]
a6p: [6882, 3765, 692, 9575, 4439, 2638]
a6b: [6882, 3765, 692, 9575, 4439, 2638]
a7p: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
a7b: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
a8p: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
a8b: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
*/
```

在每种情况下，原始数组都是为包装类型创建的，并使用 **Arrays.setAll()** 填充，正如我们在 **TestCouner.java** 中所做的那样（这也验证了 **Arrays.setAll()** 是否能同 **Integer** , **Long** , 和 **Double** ）。然后 **ConvertTo.primitive()** 将包装器数组转换为对应的基元数组，**ConverTo.boxed()** 将其转换回来。

22.11 数组元素修改

传递给 **Arrays.setAll()** 的生成器函数可以使用它接收到的数组索引修改现有的数组元素:

```java
// arrays/ModifyExisting.java

import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class ModifyExisting {
    public static void main(String[] args) {
        double[] da = new double[7];
        Arrays.setAll(da, new Rand.Double()::get);
        show(da);
        Arrays.setAll(da, n -> da[n] / 100); // [1]
        show(da);

    }
}

/* Output:
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99]
[0.0483, 0.028900000000000002, 0.028999999999999998,
0.0197, 0.0301, 0.0018, 0.009899999999999999]
*/
```

1 Lambdas 在这里特别有用，因为数组总是在 lambda 表达式的范围内。

22.12 数组并行

我们很快就不得不面对并行的主题。例如，“并行”一词在许多 Java 库方法中使用。您可能听说过类似“并行程序运行得更快”这样的说法，这是有道理的—当您可以有多个处理器时，为什么只有一个处理器在您的程序上工作呢？如果您认为您应该利用其中的“并行”，这是很容易被原谅的。要是这么简单就好了。不幸的是，通过采用这种方法，您可以很容易地编写比非并行版本运行速度更慢的代码。在你深刻理解所有的问题之前，并行编程看起来更像是一门艺术而非科学。以下是简短的版本: 用简单的方法编写代码。不要开始处理并行性，除非它成为一个问题。您仍然会遇到并行性。在本章中，我们将介绍一些为并行执行而编写的 Java 库方法。因此，您必须对它有足够的了解，以便进行基本的讨论，并避免出现错误。

在阅读并发编程这一章之后，您将更深入地理解它 (但是，唉，这还远远不够。只是这些的话，充分理解这个主题是不可能的)。在某些情况下，即使您只有一个处理器，无论您是否显式地尝试并行，并行实现是惟一的、最佳的或最符合逻辑的选择。它是一个可以一直使用的工具，所以您必须了解它的相关问题。

最好从数据的角度来考虑并行性。对于大量数据 (以及可用的额外处理器)，并行可能会有所帮助。但您也可能使事情变得更糟。

在本书的其余部分，我们将遇到不同的情况:

- 1、所提供的惟一选项是并行的。这很简单，因为我们别无选择，只能使用它。这种情况是比较罕见的。
- 2、有多个选项，但是并行版本 (通常是最新的版本) 被设计成在任何地方都可以使用 (甚至在那些不关心并行性的代码中)，如案例 #1。我们将按预期使用并行版本。
- 3、案例 1 和案例 2 并不经常发生。相反，您将遇到某些算法的两个版本，一个用于并行使用，另一个用于正常使用。我将描述并行的一个，但不会在普通代码中使用它，因为它也许会产生所有可能的问题。

我建议您在自己的代码中采用这种方法。

http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

parallelSetAll()

流式编程产生优雅的代码。例如，假设我们想要创建一个数值由从零开始填充的长数组:

```
// arrays/CountUpward.java

import java.util.stream.LongStream;

public class CountUpward {
    static long[] fillCounted(int size) {
        return LongStream.iterate(0, i -> i + 1).limit(size).toArray();
    }

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

```
        long[] l1 = fillCounted(20); // No problem
        show(l1);
        // On my machine, this runs out of heap space:
        // - long[] l2 = fillCounted(10_000_000);
    }
}

/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19]
*/
```

流实际上可以存储到将近 1000 万，但是之后就会耗尽堆空间。常规的 **setAll()** 是有效的，但是如果我们能更快地处理如此大量的数字，那就更好了。我们可以使用 **setAll()** 初始化更大的数组。如果速度成为一个问题，**Arrays.parallelSetAll()** 将 (可能) 更快地执行初始化 (请记住并行性中描述的问题)。

```

// arrays/ParallelSetAll.java

import onjava.*;
import java.util.Arrays;

public class ParallelSetAll {
    static final int SIZE = 10_000_000;

    static void intArray() {
        int[] ia = new int[SIZE];
        Arrays.setAll(ia, new Rand.Pint()::get);
        Arrays.parallelSetAll(ia, new Rand.Pint()::get);
    }

    static void longArray() {
        long[] la = new long[SIZE];
        Arrays.setAll(la, new Rand.Plong()::get);
        Arrays.parallelSetAll(la, new Rand.Plong()::get);
    }

    public static void main(String[] args) {
        intArray();
        longArray();
    }
}
```

数组分配和初始化是在单独的方法中执行的，因为如果两个数组都在 **main()** 中分配，它会耗尽内存 (至少在我的机器上是这样。还有一些方法可以告诉 Java 在启动时分配更多的内存)。

22.13 Arrays 工具类

您已经看到了 **java.util.Arrays** 中的 **fill()** 和 **setAll()/parallelSetAll()** 。该类包含许多其他有用的 **静态**程序方法，我们将对此进行研究。

概述:

- **asList()**: 获取任何序列或数组，并将其转换为一个 **列表集合**（集合章节介绍了此方法)。
- **copyOf()**：以新的长度创建现有数组的新副本。
- **copyOfRange()**：创建现有数组的一部分的新副本。
- **equals()**：比较两个数组是否相等。
- **deepEquals()**：多维数组的相等性比较。
- **stream()**：生成数组元素的流。
- **hashCode()**：生成数组的哈希值 (您将在附录中了解这意味着什么: 理解 equals() 和 hashCode())。
- **deepHashCode()**: 多维数组的哈希值。
- **sort()**：排序数组
- **parallelSort()**：对数组进行并行排序，以提高速度。
- **binarySearch()**：在已排序的数组中查找元素。
- **parallelPrefix()**：使用提供的函数并行累积 (以获得速度)。基本上，就是数组的 reduce()。
- **spliterator()**：从数组中产生一个 Spliterator; 这是本书没有涉及到的流的高级部分。
- **toString()**：为数组生成一个字符串表示。你在整个章节中经常看到这种用法。
- **deepToString()**：为多维数组生成一个字符串。你在整个章节中经常看到这种用法。对于所有基本类型和对象，所有这些方法都是重载的。

22.14 数组拷贝

与使用 for 循环手工执行复制相比，**copyOf()** 和 **copyOfRange()** 复制数组要快得多。这些方法被重载以处理所有类型。

我们开始复制数组的整数和整数:

```
// arrays/ArrayCopying.java
// Demonstrate Arrays.copy() and Arrays.copyOf()

import onjava.*;

import java.util.Arrays;

import static onjava.ArrayShow.*;

```

Any code of your own that you haven’t looked at for six or more months might as well have been written by someone else. — Eagleson’s law

```java
class Sup {
    // Superclass
    private int id;

    Sup(int n) {
        id = n;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + id;
    }
}

class Sub extends Sup { // Subclass

    Sub(int n) {
        super(n);
    }
}

public class ArrayCopying {
    public static final int SZ = 15;

    public static void main(String[] args) {
        int[] a1 = new int[SZ];
        Arrays.setAll(a1, new Count.Integer()::get);
        show("a1", a1);
        int[] a2 = Arrays.copyOf(a1, a1.length); // [1]
        // Prove they are distinct arrays:
        Arrays.fill(a1, 1);
        show("a1", a1);
        show("a2", a2);
        // Create a shorter result:
        a2 = Arrays.copyOf(a2, a2.length / 2); // [2]
        show("a2", a2);
        // Allocate more space:
        a2 = Arrays.copyOf(a2, a2.length + 5);
        show("a2", a2);
        // Also copies wrapped arrays:
        Integer[] a3 = new Integer[SZ]; // [3]
        Arrays.setAll(a3, new Count.Integer()::get);
        Integer[] a4 = Arrays.copyOfRange(a3, 4, 12);
        show("a4", a4);
        Sub[] d = new Sub[SZ / 2];
        Arrays.setAll(d, Sub::new); // Produce Sup[] from Sub[]:
        Sup[] b = Arrays.copyOf(d, d.length, Sup[].class); // [4]
        show(b); // This "downcast" works fine:
        Sub[] d2 = Arrays.copyOf(b, b.length, Sub[].class); // [5]
        show(d2); // Bad "downcast" compiles but throws exception:
        Sup[] b2 = new Sup[SZ / 2];
```

```
        Arrays.setAll(b2, Sup::new);
        try {
            Sub[] d3 = Arrays.copyOf(b2, b2.length, Sub[].class); // [6]
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
/* Output: a1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] a1: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
           a2:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]a2:[0, 1, 2, 3, 4, 5, 6]a2:[
           0, 1, 2, 3, 4, 5, 6, 0, 0, 0, 0, 0]a4:[4, 5, 6, 7, 8, 9, 10, 11][Sub0, Sub1, Sub2, Sub3, Sub4, Sub5, Sub6][
           Sub0, Sub1, Sub2, Sub3, Sub4, Sub5, Sub6]java.lang.ArrayStoreException */
```

1 这是复制的基本方法; 只需给出返回的复制数组的大小。这对于编写需要调整存储大小的算法很有帮助。复制之后，我们把 a1 的所有元素都设为 1，以证明 a1 的变化不会影响 a2 中的任何东西。

2 通过更改最后一个参数，我们可以缩短或延长返回的复制数组。

[3] **copyOf()** 和 **copyOfRange()** 也可以使用包装类型。**copyOfRange()** 需要一个开始和结束索引。

[4] **copyOf()** 和 **copyOfRange()** 都有一个版本，该版本通过在方法调用的末尾添加目标类型来创建不同类型的数组。我首先想到的是，这可能是一种从原生数组生成包装数组的方法，反之亦然。但这没用。它的实际用途是“向上转换”和“向下转换”数组。也就是说，如果您有一个子类型 (派生类型) 的数组，而您想要一个基类型的数组，那么这些方法将生成所需的数组。

[5] 您甚至可以成功地“向下强制转换”，并从超类型的数组生成子类型的数组。这个版本运行良好，因为我们只是“upcast”。

[6] 这个“数组转换”将编译，但是如果类型不兼容，您将得到一个运行时异常。在这里，强制将基类型转换为派生类型是非法的，因为派生对象中可能有基对象中没有的属性和方法。

实例表明，原生数组和对象数组都可以被复制。但是，如果复制对象的数组，那么只复制引用—不复制对象本身。这称为浅拷贝 (有关更多细节，请参阅附录: 传递和返回对象)。

还有一个方法 **System.arraycopy()** ，它将一个数组复制到另一个已经分配的数组中。这将不会执行自动装箱或自动卸载—两个数组必须是完全相同的类型。

22.15 数组比较

数组提供了 **equals()** 来比较一维数组，以及 **deepEquals()** 来比较多维数组。对于所有原生类型和对象，这些方法都是重载的。

数组相等的含义：数组必须有相同数量的元素，并且每个元素必须与另一个数组中的对应元素相等，对每个元素使用 **equals()**(对于原生类型，使用原生类型的包装类的 **equals()** 方法; 例如，int 的 Integer.equals()。

```
// arrays/ComparingArrays.java
// Using Arrays.equals()

import java.util.*;
import onjava.*;

public class ComparingArrays {
    public static final int SZ = 15;

    static String[][] twoDArray() {
        String[][] md = new String[5][];
        Arrays.setAll(md, n -> new String[n]);
        for (int i = 0; i < md.length; i++) Arrays.setAll(md[i], new
            Rand.String()::get);
        return md;
    }

    public static void main(String[] args) {
        int[] a1 = new int[SZ], a2 = new int[SZ];
        Arrays.setAll(a1, new Count.Integer()::get);
        Arrays.setAll(a2, new Count.Integer()::get);
        System.out.println("a1 == a2: " + Arrays.equals(a1, a2));
        a2[3] = 11;
        System.out.println("a1 == a2: " + Arrays.equals(a1, a2));
        Integer[] a1w = new Integer[SZ], a2w = new Integer[SZ];
        Arrays.setAll(a1w, new Count.Integer()::get);
        Arrays.setAll(a2w, new Count.Integer()::get);
        System.out.println("a1w == a2w: " + Arrays.equals(a1w, a2w));
        a2w[3] = 11;
        System.out.println("a1w == a2w: " + Arrays.equals(a1w, a2w));
        String[][] md1 = twoDArray(), md2 = twoDArray();
        System.out.println(Arrays.deepToString(md1));
        System.out.println("deepEquals(md1, md2): " + Arrays.deepEquals
            (md1, md2));
        System.out.println("md1 == md2: " + Arrays.equals(md1, md2));
        md1[4][1] = "#$#$#$#";
        System.out.println(Arrays.deepToString(md1));
        System.out.println("deepEquals(md1, md2): " + Arrays.deepEquals
            (md1, md2));
    }
}

/* Output:
a1 == a2: true
a1 == a2: false
a1w == a2w: true
```

```
a1w == a2w: false
[[], [btpenpc], [btpenpc, cuxszgv], [btpenpc, cuxszgv,
 gmeinne], [btpenpc, cuxszgv, gmeinne, eloztdv]]
 deepEquals(md1, md2): true
 md1 == md2: false
 [[], [btpenpc], [btpenpc, cuxszgv], [btpenpc, cuxszgv,
 gmeinne], [btpenpc, #$#$#$#, gmeinne, eloztdv]]
 deepEquals(md1, md2): false
 */
```

最初，a1 和 a2 是完全相等的，所以输出是 true，但是之后其中一个元素改变了，这使得结果为 false。a1w 和 a2w 是对一个封装类型数组重复该练习。

md1 和 **md2** 是通过 **twoDArray()** 以相同方式初始化的多维字符串数组。注意，**deepEquals()** 返回 **true**，因为它执行了适当的比较，而普通的 **equals()** 错误地返回 **false**。如果我们更改数组中的一个元素，**deepEquals()** 将检测它。

22.16 流和数组

stream() 方法很容易从某些类型的数组中生成元素流。

```java
// arrays/StreamFromArray.java

import java.util.*;
import onjava.*;

public class StreamFromArray {
    public static void main(String[] args) {
        String[] s = new Rand.String().array(10);
        Arrays.stream(s).skip(3).limit(5).map(ss -> ss + "!").forEach(
            System.out::println);
        int[] ia = new Rand.Pint().array(10);
        Arrays.stream(ia).skip(3).limit(5)
                .map(i -> i * 10).forEach(System.out::println);
        Arrays.stream(new long[10]);
        Arrays.stream(new double[10]);
        // Only int, long and double work:
        // - Arrays.stream(new boolean[10]);
        // - Arrays.stream(new byte[10]);
        // - Arrays.stream(new char[10]);
        // - Arrays.stream(new short[10]);
        // - Arrays.stream(new float[10]);
        // For the other types you must use wrapped arrays:
        float[] fa = new Rand.Pfloat().array(10);
        Arrays.stream(ConvertTo.boxed(fa));
        Arrays.stream(new Rand.Float().array(10));
    }
}
```

```
/* Output:
    eloztdv!
    ewcippc!
    ygpoalk!
    ljlbynx!
    taprwxz!
    47200
    61770
    84790
    66560
    37680
*/
```

只有“原生类型” **int**、**long** 和 **double** 可以与 **Arrays.stream()** 一起使用; 对于其他的，您必须以某种方式获得一个包装类型的数组。

通常，将数组转换为流来生成所需的结果要比直接操作数组容易得多。请注意，即使流已经“用完”(您不能重复使用它)，您仍然拥有该数组，因此您可以以其他方式使用它—-包括生成另一个流。

22.17 数组排序

根据对象的实际类型执行比较排序。一种方法是为不同的类型编写对应的排序方法，但是这样的代码不能复用。

编程设计的一个主要目标是“将易变的元素与稳定的元素分开”，在这里，保持不变的代码是一般的排序算法，但是变化的是对象的比较方式。因此，使用策略设计模式而不是将比较代码放入许多不同的排序源码中。使用策略模式时，变化的代码部分被封装在一个单独的类 (策略对象) 中。

您将一个策略对象交给相同的代码，该代码使用策略模式来实现其算法。通过这种方式，您将使用相同的排序代码，使不同的对象表达不同的比较方式。

Java 有两种方式提供比较功能。第一种方法是通过实现 **java.lang.Comparable** 接口的原生方法。这是一个简单的接口，只含有一个方法 **compareTo()**。该方法接受另一个与参数类型相同的对象作为参数，如果当前对象小于参数，则产生一个负值; 如果参数相等，则产生零值; 如果当前对象大于参数，则产生一个正值。

这里有一个类，它实现了 **Comparable** 接口并演示了可比性，而且使用 Java 标准库方法 **Arrays.sort()**:

```
// arrays/CompType.java
// Implementing Comparable in a class

import onjava.*;

import java.util.Arrays;
import java.util.SplittableRandom;

```

```java
import static onjava.ArrayShow.*;

public class CompType implements Comparable<CompType> {
    private static int count = 1;
    private static SplittableRandom r = new SplittableRandom(47);
    int i;
    int j;

    public CompType(int n1, int n2) {
        i = n1;
        j = n2;
    }

    public static CompType get() {
        return new CompType(r.nextInt(100), r.nextInt(100));
    }

    public static void main(String[] args) {
        CompType[] a = new CompType[12];
        Arrays.setAll(a, n -> get());
        show("Before sorting", a);
        Arrays.sort(a);
        show("After sorting", a);
    }

    @Override
    public String toString() {
        String result = "[i = " + i + ", j = " + j + "]";
        if (count++ % 3 == 0) result += "\n";
        return result;
    }

    @Override
    public int compareTo(CompType rv) {
        return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
    }
}
/* Output:
Before sorting: [[i = 35, j = 37], [i = 41, j = 20], [i = 77, j = 79] ,
                [i = 56, j = 68], [i = 48, j = 93],
                [i = 70, j = 7] , [i = 0, j = 25],
                [i = 62, j = 34], [i = 50, j = 82] ,
                [i = 31, j = 67], [i = 66, j = 54],
                [i = 21, j = 6] ]
After sorting: [[i = 0, j = 25], [i = 21, j = 6], [i = 31, j = 67] ,
               [i = 35, j = 37], [i = 41, j = 20], [i = 48, j = 93] ,
               [i = 50, j = 82], [i = 56, j = 68], [i = 62, j = 34] ,
               [i = 66, j = 54], [i = 70, j = 7], [i = 77, j = 79] ]
*/
```

当您定义比较方法时，您有责任决定将一个对象与另一个对象进行比较意味着什么。这里，在比较

中只使用 i 值和 j 值将被忽略。

get() 方法通过使用随机值初始化 CompType 对象来构建它们。在 **main()** 中，**get()** 与 **Arrays.setAll()** 一起使用，以填充一个 **CompType 类型**数组，然后对其排序。如果没有实现 **Comparable 接口**，那么当您试图调用 **sort()** 时，您将在运行时获得一个 **ClassCastException** 。这是因为 **sort()** 将其参数转换为 **Comparable 类型**。

现在假设有人给了你一个没有实现 **Comparable 接口**的类，或者给了你一个实现 **Comparable 接口**的类，但是你不喜欢它的工作方式而愿意有一个不同的对于此类型的比较方法。为了解决这个问题，创建一个实现 **Comparator** 接口的单独的类 (在集合一章中简要介绍)。它有两个方法，**compare()** 和 **equals()**。但是，除了特殊的性能需求外，您不需要实现 **equals()**，因为无论何时创建一个类，它都是隐式地继承自 **Object**，**Object** 有一个 equals()。您可以只使用默认的 **Object equals()** 来满足接口的规范。

集合类 (注意复数; 我们将在下一章节讨论它) 包含一个方法 **reverseOrder()**，它生成一个来 **Comparator**（比较器）反转自然排序顺序。这可以应用到比较对象：

```
// arrays/Reverse.java
// The Collections.reverseOrder() Comparator

import onjava.*;

import java.util.Arrays;
import java.util.Collections;

import static onjava.ArrayShow.*;

public class Reverse {
    public static void main(String[] args) {
        CompType[] a = new CompType[12];
        Arrays.setAll(a, n -> CompType.get());
        show("Before sorting", a);
        Arrays.sort(a, Collections.reverseOrder());
        show("After sorting", a);
    }
}
/* Output:
Before sorting: [[i = 35, j = 37], [i = 41, j = 20],
                [i = 77, j = 79] , [i = 56, j = 68],
                [i = 48, j = 93], [i = 70, j = 7],
                [i = 0, j = 25], [i = 62, j = 34],
                [i = 50, j = 82] , [i = 31, j = 67],
                [i = 66, j = 54], [i = 21, j = 6] ]
After sorting: [[i = 77, j = 79], [i = 70, j = 7],
                [i = 66, j = 54] , [i = 62, j = 34],
                [i = 56, j = 68], [i = 50, j = 82] ,
                [i = 48, j = 93], [i = 41, j = 20],
                [i = 35, j = 37] , [i = 31, j = 67],
```

```
                [i = 21, j = 6], [i = 0, j = 25] ]
*/
```

您还可以编写自己的比较器。这个比较 CompType 对象基于它们的 j 值而不是它们的 i 值:

```
// arrays/ComparatorTest.java
// Implementing a Comparator for a class

import onjava.*;

import java.util.Arrays;
import java.util.Comparator;

import static onjava.ArrayShow.*;

class CompTypeComparator implements Comparator<CompType> {
    public int compare(CompType o1, CompType o2) {
        return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
    }
}

public class ComparatorTest {
    public static void main(String[] args) {
        CompType[] a = new CompType[12];
        Arrays.setAll(a, n -> CompType.get());
        show("Before sorting", a);
        Arrays.sort(a, new CompTypeComparator());
        show("After sorting", a);
    }
}
/* Output:
Before sorting:[[i = 35, j = 37], [i = 41, j = 20], [i = 77, j = 79] ,
                [i = 56, j = 68], [i = 48, j = 93], [i = 70, j = 7] ,
                [i = 0, j = 25], [i = 62, j = 34], [i = 50, j = 82],
                [i = 31, j = 67], [i = 66, j = 54], [i = 21, j = 6] ]
After sorting: [[i = 21, j = 6], [i = 70, j = 7], [i = 41, j = 20] ,
                [i = 0, j = 25], [i = 62, j = 34], [i = 35, j = 37] ,
                [i = 66, j = 54], [i = 31, j = 67], [i = 56, j = 68] ,
                [i = 77, j = 79], [i = 50, j = 82], [i = 48, j = 93] ]
*/
```

22.18 Arrays.sort() 的使用

使用内置的排序方法，您可以对实现了 **Comparable** 接口或具有 **Comparator** 的任何对象数组或任何原生数组进行排序。这里我们生成一个随机字符串对象数组并对其排序:

```
// arrays/StringSorting.java
// Sorting an array of Strings
```

The first 90 percent of the code accounts for the first 90 percent of the development time…The remaining 10 percent of the code accounts for the other 90 percent of the development time. — Tom Cargill

```java

import onjava.*;

import java.util.Arrays;
import java.util.Collections;

import static onjava.ArrayShow.*;

public class StringSorting {
    public static void main(String[] args) {
        String[] sa = new Rand.String().array(20);
        show("Before sort", sa);
        Arrays.sort(sa);
        show("After sort", sa);
        Arrays.sort(sa, Collections.reverseOrder());
        show("Reverse sort", sa);
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        show("Case-insensitive sort", sa);
    }
}
/* Output:
Before sort: [btpenpc, cuxszgv, gmeinne, eloztdv, ewcippc,
            ygpoalk, ljlbynx, taprwxz, bhmupju, cjwzmmr,
            anmkkyh, fcjpthl, skddcat, jbvlgwc, mvducuj,
            ydpulcq, zehpfmm, zrxmclh, qgekgly, hyoubzl]

After sort: [anmkkyh, bhmupju, btpenpc, cjwzmmr, cuxszgv,
            eloztdv, ewcippc, fcjpthl, gmeinne, hyoubzl,
            jbvlgwc, ljlbynx, mvducuj, qgekgly, skddcat,
            taprwxz, ydpulcq, ygpoalk, zehpfmm, zrxmclh]

Reverse sort: [zrxmclh, zehpfmm, ygpoalk, ydpulcq,taprwxz,
            skddcat, qgekgly, mvducuj, ljlbynx, jbvlgwc,
            hyoubzl, gmeinne, fcjpthl, ewcippc, eloztdv,
            cuxszgv, cjwzmmr, btpenpc, bhmupju, anmkkyh]

Case-insensitive sort: [anmkkyh, bhmupju, btpenpc, cjwzmmr,
                cuxszgv, eloztdv, ewcippc, fcjpthl, gmeinne,
                hyoubzl, jbvlgwc, ljlbynx, mvducuj, qgekgly,
                skddcat, taprwxz, ydpulcq, ygpoalk, zehpfmm, zrxmclh]
*/
```

注意字符串排序算法中的输出。它是字典式的，所以它把所有以大写字母开头的单词放在前面，然后是所有以小写字母开头的单词。(电话簿通常是这样分类的。) 无论大小写，要将单词组合在一起，请使用 **String.CASE_INSENSITIVE_ORDER** ，如对 sort() 的最后一次调用所示。

Java 标准库中使用的排序算法被设计为最适合您正在排序的类型—-原生类型的快速排序和对象的归并排序。

开始 90%的代码占开始 90%的开发时间……剩下 10%的代码还是占 90%的开发时间。

22.19 并行排序

如果排序性能是一个问题，那么可以使用 **Java 8 parallelSort()**，它为所有不可预见的情况 (包括数组的排序区域或使用了比较器) 提供了重载版本。为了查看相比于普通的 sort(), **parallelSort()** 的优点，我们使用了用来验证代码时的 **JMH**:

```
// arrays/jmh/ParallelSort.java
package arrays.jmh;

import onjava.*;
import org.openjdk.jmh.annotations.*;

import java.util.Arrays;

@State(Scope.Thread)
public class ParallelSort {
    private long[] la;

    @Setup
    public void setup() {
        la = new Rand.Plong().array(100_000);
    }

    @Benchmark
    public void sort() {
        Arrays.sort(la);
    }

    @Benchmark
    public void parallelSort() {
        Arrays.parallelSort(la);
    }
}
```

parallelSort() 算法将大数组拆分成更小的数组，直到数组大小达到极限，然后使用普通的 **Arrays .sort()** 方法。然后合并结果。该算法需要不大于原始数组的额外工作空间。

您可能会看到不同的结果，但是在我的机器上，并行排序将速度提高了大约 3 倍。由于并行版本使用起来很简单，所以很容易考虑在任何地方使用它，而不是 **Arrays.sort ()**。当然，它可能不是那么简单—看看微基准测试。

22.20 binarySearch 二分查找

一旦数组被排序，您就可以通过使用 **Arrays.binarySearch()** 来执行对特定项的快速搜索。但是，如果尝试在未排序的数组上使用 **binarySearch()**，结果是不可预测的。下面的示例使用 **Rand.Pint**

Commenting your code is like cleaning your bathroom —you never want to do it, but it really does create a more pleasant experience for you and your guests. — Ryan Campbell

类来创建一个填充随机整形值的数组，然后调用 **getAsInt()** (因为 **Rand.Pint** 是一个 **IntSupplier**) 来产生搜索值:

```
// arrays/ArraySearching.java
// Using Arrays.binarySearch()

import onjava.*;

import java.util.Arrays;

import static onjava.ArrayShow.*;

public class ArraySearching {
    public static void main(String[] args) {
        Rand.Pint rand = new Rand.Pint();
        int[] a = new Rand.Pint().array(25);
        Arrays.sort(a);
        show("Sorted array", a);
        while (true) {
            int r = rand.getAsInt();
            int location = Arrays.binarySearch(a, r);
            if (location >= 0) {
                System.out.println("Location of " + r + " is " +
                    location + ", a[" + location + "] is " + a[location
                    ]);
                break; // Out of while loop
            }
        }
    }
}
/* Output:
Sorted array: [125, 267, 635, 650, 1131, 1506, 1634, 2400, 2766,
               3063, 3768, 3941, 4720, 4762, 4948, 5070, 5682,
               5807, 6177, 6193, 6656, 7021, 8479, 8737, 9954]
Location of 635 is 2, a[2] is 635
*/
```

在 while 循环中，随机值作为搜索项生成，直到在数组中找到其中一个为止。

如果找到了搜索项，**Arrays.binarySearch()** 将生成一个大于或等于零的值。否则，它将产生一个负值，表示如果手动维护已排序的数组，则应该插入元素的位置。产生的值是 -(插入点) - 1 。插入点是大于键的第一个元素的索引，如果数组中的所有元素都小于指定的键，则是 **a.size()** 。

如果数组包含重复的元素，则无法保证找到其中的那些重复项。搜索算法不是为了支持重复的元素，而是为了容忍它们。如果需要没有重复元素的排序列表，可以使用 **TreeSet** (用于维持排序顺序) 或 **LinkedHashSet** (用于维持插入顺序)。这些类自动为您处理所有的细节。只有在出现性能瓶颈的情况下，才应该使用手工维护的数组替换这些类中的一个。

如果使用比较器 (原语数组不允许使用比较器进行排序) 对对象数组进行排序，那么在执行

binarySearch() (使用重载版本的 binarySearch()) 时必须包含相同的比较器。例如，可以修改 **StringSorting.java** 来执行搜索:

```
// arrays/AlphabeticSearch.java
// Searching with a Comparator

import onjava.*;

import java.util.Arrays;

import static onjava.ArrayShow.*;

public class AlphabeticSearch {
    public static void main(String[] args) {
        String[] sa = new Rand.String().array(30);
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        show(sa);
        int index = Arrays.binarySearch(sa, sa[10], String.
            CASE_INSENSITIVE_ORDER);
        System.out.println("Index: " + index + "\n" + sa[index]);
    }
}
/* Output:
[anmkkyh, bhmupju, btpenpc, cjwzmmr, cuxszgv, eloztdv, ewcippc,
ezdeklu, fcjpthl, fqmlgsh, gmeinne, hyoubzl, jbvlgwc, jlxpqds,
ljlbynx, mvducuj, qgekgly, skddcat, taprwxz, uybypgp, vjsszkn,
vniyapk, vqqakbm, vwodhcf, ydpulcq, ygpoalk, yskvett, zehpfmm,
zofmmvm, zrxmclh]
Index: 10 gmeinne
*/
```

比较器必须作为第三个参数传递给重载的 **binarySearch()** 。在本例中，成功是有保证的，因为搜索项是从数组本身中选择的。

22.21 parallelPrefix 并行前缀

没有“prefix()”方法，只有 **parallelPrefix()**。这类似于 **Stream** 类中的 **reduce()** 方法: 它对前一个元素和当前元素执行一个操作，并将结果放入当前元素位置:

```
// arrays/ParallelPrefix1.java

import onjava.*;

import java.util.Arrays;

import static onjava.ArrayShow.*;

```

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the universe trying to produce bigger and better idiots. So far, the universe is winning. — Rick Cook

```
public class ParallelPrefix1 {
    public static void main(String[] args) {
        int[] nums = new Count.Pint().array(10);
        show(nums);
        System.out.println(Arrays.stream(nums).reduce(Integer::sum).
            getAsInt());
        Arrays.parallelPrefix(nums, Integer::sum);
        show(nums);
        System.out.println(Arrays.stream(new Count.Pint().array(6)).
            reduce(Integer::sum).getAsInt());
    }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
45
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
15
*/
```

这里我们对数组应用 Integer::sum。在位置 0 中，它将先前计算的值 (因为没有先前的值) 与原始数组位置 0 中的值组合在一起。在位置 1 中，它获取之前计算的值 (它只是存储在位置 0 中)，并将其与位置 1 中先前计算的值相结合。依次往复。

使用 **Stream.reduce()**, 您只能得到最终结果, 而使用 **Arrays.parallelPrefix()**, 您还可以得到所有中间计算, 以确保它们是有用的。注意, 第二个 **Stream.reduce()** 计算的结果已经在 **parallelPrefix()** 计算的数组中。

使用字符串可能更清楚:

```
// arrays/ParallelPrefix2.java

import onjava.*;

import java.util.Arrays;

import static onjava.ArrayShow.*;

public class ParallelPrefix2 {
    public static void main(String[] args) {
        String[] strings = new Rand.String(1).array(8);
        show(strings);
        Arrays.parallelPrefix(strings, (a, b) -> a + b);
        show(strings);
    }
}
/* Output:
[b, t, p, e, n, p, c, c]
[b, bt, btp, btpe, btpen, btpenp, btpenpc, btpenpcc]
*/
```

如前所述，使用流进行初始化非常优雅，但是对于大型数组，这种方法可能会耗尽堆空间。使用 **setAll()** 执行初始化更节省内存:

```
// arrays/ParallelPrefix3.java
// {ExcludeFromTravisCI}

import java.util.Arrays;

public class ParallelPrefix3 {
    static final int SIZE = 10_000_000;

    public static void main(String[] args) {
        long[] nums = new long[SIZE];
        Arrays.setAll(nums, n -> n);
        Arrays.parallelPrefix(nums, Long::sum);
        System.out.println("First 20: " + nums[19]);
        System.out.println("First 200: " + nums[199]);
        System.out.println("All: " + nums[nums.length - 1]);
    }
}
/* Output:
First 20: 190
First 200: 19900
All: 49999995000000
*/
```

因为正确使用 **parallelPrefix()** 可能相当复杂，所以通常应该只在存在内存或速度问题 (或两者都有) 时使用。否则，**Stream.reduce()** 应该是您的首选。

22.22 本章小结

Java 为固定大小的低级数组提供了合理的支持。这种数组强调的是性能而不是灵活性，就像 C 和 c++ 数组模型一样。在 Java 的最初版本中，固定大小的低级数组是绝对必要的，这不仅是因为 Java 设计人员选择包含原生类型 (也考虑到性能)，还因为那个版本对集合的支持非常少。因此，在早期的 Java 版本中，选择数组总是合理的。

在 Java 的后续版本中，集合支持得到了显著的改进，现在集合在除性能外的所有方面都优于数组，即使这样，集合的性能也得到了显著的改进。正如本书其他部分所述，无论如何，性能问题通常不会出现在您设想的地方。

使用自动装箱和泛型，在集合中保存原生类型是毫不费力的，这进一步鼓励您用集合替换低级数组。由于泛型产生类型安全的集合，数组在这方面也不再有优势。

如本章所述，当您尝试使用泛型时，您将看到泛型对数组是相当不友好的。通常，即使可以让泛型和数组以某种形式一起工作 (在下一章中您将看到)，在编译期间仍然会出现“unchecked”警告。

有几次，当我们讨论特定的例子时，我直接从 Java 语言设计人员那里听到我应该使用集合而不是

数组 (我使用数组来演示特定的技术，所以我没有这个选项)。

所有这些问题都表明，在使用 Java 的最新版本进行编程时，应该“优先选择集合而不是数组”。只有当您证明性能是一个问题 (并且切换到一个数组实际上会有很大的不同) 时，才应该重构到数组。这是一个相当大胆的声明，但是有些语言根本没有固定大小的低级数组。它们只有可调整大小的集合，而且比 C/C++/java 风格的数组功能多得多。例如，Python 有一个使用基本数组语法的列表类型，但是具有更大的功能—您甚至可以从它继承:

```
# arrays/PythonLists.py

aList=[1,2,3,4,5]print(type(aList)) #<type 'list'>
print(aList) # [1,2,3,4,5]
        print(aList[4]) # 5Basic list indexing
        aList.append(6) # lists can be resized
        aList+=[7,8] # Add a list to a list
        print(aList) # [1,2,3,4,5,6,7,8]
        aSlice=aList[2:4]
        print(aSlice) # [3,4]

class MyList(list): # Inherit from list
        # Define a method;'this'pointer is explicit:
        def getReversed(self):
            reversed=self[:] # Copy list using slices
            reversed.reverse() # Built-in list method
            return reversed
        # No'new'necessary for object creation:
        list2=MyList(aList)
        print(type(list2)) #<class '__main__.MyList'>
        print(list2.getReversed()) # [8,7,6,5,4,3,2,1]
        output="""
        <class 'list'>
        [1, 2, 3, 4, 5]
        5
        [1, 2, 3, 4, 5, 6, 7, 8]
        [3, 4]
        <class '__main__.MyList'>
        [8, 7, 6, 5, 4, 3, 2, 1]
        """
```

前一章介绍了基本的 Python 语法。在这里，通过用方括号包围以逗号分隔的对象序列来创建列表。结果是一个运行时类型为 list 的对象 (print 语句的输出显示为同一行中的注释)。打印列表的结果与在 Java 中使用 Arrays.toString() 的结果相同。通过将: 操作符放在索引操作中，通过切片来创建列表的子序列。list 类型有更多的内置操作，通常只需要序列类型。MyList 是一个类定义; 基类放在括号内。在类内部，def 语句生成方法，该方法的第一个参数在 Java 中自动与之等价，除了在 Python 中它是显式的，而且标识符 self 是按约定使用的 (它不是关键字)。注意构造函数是如何自动继承的。

虽然一切在 Python 中真的是一个对象 (包括整数和浮点类型), 你仍然有一个安全门, 因为你可以

设计中最难的部分……是阻止功能。

优化性能关键型的部分代码编写扩展的 C, c++ 或使用特殊的工具设计容易加速您的 Python 代码 (有很多)。通过这种方式，可以在不影响性能改进的情况下保持对象的纯度。

PHP 甚至更进一步，它只有一个数组类型，既充当 int 索引数组，又充当关联数组 (Map)。

在经历了这么多年的 Java 发展之后，我们可以很有趣地推测，如果重新开始，设计人员是否会将原生类型和低级数组放在该语言中 (同样在 JVM 上运行的 Scala 语言不包括这些)。如果不考虑这些，就有可能开发出一种真正纯粹的面向对象语言 (尽管有这样的说法，Java 并不是一种纯粹的面向对象语言，这正是因为它的底层缺陷)。关于效率的最初争论总是令人信服的，但是随着时间的推移，我们已经看到了从这个想法向更高层次的组件 (如集合) 的演进。此外，如果集合可以像在某些语言中一样构建到核心语言中，那么编译器就有更好的机会进行优化。

撇开""Green-fields"的推测不谈，我们肯定会被数组所困扰，当你阅读代码时就会看到它们。然而，集合几乎总是更好的选择。

23 枚举

> 关键字 enum 可以将一组具名的值的有限集合创建为一种新的类型，而这些具名的值可以作为常规的程序组件使用。这是一种非常有用的功能

在初始化和清理 这章结束的时候，我们已经简单地介绍了枚举的概念。现在，你对 Java 已经有了更深刻的理解，因此可以更深入地学习 Java 中的枚举了。你将在本章中看到，使用 enum 可以做很多有趣的事情，同时，我们也会深入其他的 Java 特性，例如泛型和反射。在这个过程中，我们还将学习一些设计模式。

23.1 基本 enum 特性

我们已经在初始化和清理 这章章看到, 调用 enum 的 values() 方法, 可以遍历 enum 实例.values() 方法返回 enum 实例的数组，而且该数组中的元素严格保持其在 enum 中声明时的顺序，因此你可以在循环中使用 values() 返回的数组。

创建 enum 时，编译器会为你生成一个相关的类，这个类继承自 Java.lang.Enum。下面的例子演示了 Enum 提供的一些功能：

```
// enums/EnumClass.java
// Capabilities of the Enum class
enum Shrubbery { GROUND, CRAWLING, HANGING }
public class EnumClass {
    public static void main(String[] args) {
        for(Shrubbery s : Shrubbery.values()) {
            System.out.println(
                    s + " ordinal: " + s.ordinal());
            System.out.print(
                    s.compareTo(Shrubbery.CRAWLING) + " ");
            System.out.print(
                    s.equals(Shrubbery.CRAWLING) + " ");
            System.out.println(s == Shrubbery.CRAWLING);
            System.out.println(s.getDeclaringClass());
            System.out.println(s.name());
            System.out.println("********************");
        }
// Produce an enum value from a String name:
```

要想了解递归，首先必须理解递归。

```
        for(String s :
                "HANGING CRAWLING GROUND".split(" ")) {
            Shrubbery shrub =
                    Enum.valueOf(Shrubbery.class, s);
            System.out.println(shrub);
        }
    }
}
```

输出:

```
GROUND ordinal: 0
-1 false false
class Shrubbery
GROUND
********************
CRAWLING ordinal: 1
0 true true
class Shrubbery
CRAWLING
********************
HANGING ordinal: 2
1 false false
class Shrubbery
HANGING
********************
HANGING
CRAWLING
GROUND
```

ordinal() 方法返回一个 int 值，这是每个 enum 实例在声明时的次序，从 0 开始。可以使用 == 来比较 enum 实例，编译器会自动为你提供 equals() 和 hashCode() 方法。Enum 类实现了 Comparable 接口，所以它具有 compareTo() 方法。同时，它还实现了 Serializable 接口。

如果在 enum 实例上调用 getDeclaringClass() 方法，我们就能知道其所属的 enum 类。

name() 方法返回 enum 实例声明时的名字，这与使用 toString() 方法效果相同。valueOf() 是在 Enum 中定义的 static 方法，它根据给定的名字返回相应的 enum 实例，如果不存在给定名字的实例，将会抛出异常。

23.1.1 将静态类型导入用于 enum

先看一看 初始化和清理 这章中 Burrito.java 的另一个版本:

```
// enums/SpicinessEnum.java
package enums;
public enum SpicinessEnum {
    NOT, MILD, MEDIUM, HOT, FLAMING
}
```

There are only two kinds of programming languages: those people always bitch about and those nobody uses. — Bjarne Stroustrup

```
// enums/Burrito2.java
// {java enums.Burrito2}
package enums;
import static enums.SpicinessEnum.*;
public class Burrito2 {
    SpicinessEnum degree;
    public Burrito2(SpicinessEnum degree) {
        this.degree = degree;
    }
    @Override
    public String toString() {
        return "Burrito is "+ degree;
    }
    public static void main(String[] args) {
        System.out.println(new Burrito2(NOT));
        System.out.println(new Burrito2(MEDIUM));
        System.out.println(new Burrito2(HOT));
    }
}
```

输出为:

```
Burrito is NOT
Burrito is MEDIUM
Burrito is HOT
```

使用 static import 能够将 enum 实例的标识符带入当前的命名空间，所以无需再用 enum 类型来修饰 enum 实例。这是一个好的想法吗？或者还是显式地修饰 enum 实例更好？这要看代码的复杂程度了。编译器可以确保你使用的是正确的类型，所以唯一需要担心的是，使用静态导入会不会导致你的代码令人难以理解。多数情况下，使用 static import 还是有好处的，不过，程序员还是应该对具体情况进行具体分析。

注意，在定义 enum 的同一个文件中，这种技巧无法使用，如果是在默认包中定义 enum，这种技巧也无法使用（在 Sun 内部对这一点显然也有不同意见）。

23.2 方法添加

除了不能继承自一个 enum 之外，我们基本上可以将 enum 看作一个常规的类。也就是说我们可以向 enum 中添加方法。enum 甚至可以有 main() 方法。

一般来说，我们希望每个枚举实例能够返回对自身的描述，而不仅仅只是默认的 toString() 实现，这只能返回枚举实例的名字。为此，你可以提供一个构造器，专门负责处理这个额外的信息，然后添加一个方法，返回这个描述信息。看一看下面的示例:

```
// enums/OzWitch.java
// The witches in the land of Oz
```

```
public enum OzWitch {
    // Instances must be defined first, before methods:
    WEST("Miss Gulch, aka the Wicked Witch of the West"),
    NORTH("Glinda, the Good Witch of the North"),
    EAST("Wicked Witch of the East, wearer of the Ruby " +
            "Slippers, crushed by Dorothy's house"),
    SOUTH("Good by inference, but missing");
    private String description;
    // Constructor must be package or private access:
    private OzWitch(String description) {
        this.description = description;
    }
    public String getDescription() { return description; }
    public static void main(String[] args) {
        for(OzWitch witch : OzWitch.values())
            System.out.println(
                    witch + ": " + witch.getDescription());
    }
}
```

输出为:

```
WEST: Miss Gulch, aka the Wicked Witch of the West
NORTH: Glinda, the Good Witch of the North
EAST: Wicked Witch of the East, wearer of the Ruby
Slippers, crushed by Dorothy's house
SOUTH: Good by inference, but missing
```

注意，如果你打算定义自己的方法，那么必须在 enum 实例序列的最后添加一个分号。同时，Java 要求你必须先定义 enum 实例。如果在定义 enum 实例之前定义了任何方法或属性，那么在编译时就会得到错误信息。

enum 中的构造器与方法和普通的类没有区别，因为除了有少许限制之外，enum 就是一个普通的类。所以，我们可以使用 enum 做许多事情（虽然，我们一般只使用普通的枚举类型）

在这个例子中，虽然我们有意识地将 enum 的构造器声明为 private，但对于它的可访问性而言，其实并没有什么变化，因为（即使不声明为 private）我们只能在 enum 定义的内部使用其构造器创建 enum 实例。一旦 enum 的定义结束，编译器就不允许我们再使用其构造器来创建任何实例了。

23.2.1 覆盖 enum 的方法

覆盖 toSring() 方法，给我们提供了另一种方式来为枚举实例生成不同的字符串描述信息。在下面的示例中，我们使用的就是实例的名字，不过我们希望改变其格式。覆盖 enum 的 toSring() 方法与覆盖一般类的方法没有区别:

```
// enums/SpaceShip.java
import java.util.stream.*;
public enum SpaceShip {
```

```
    SCOUT, CARGO, TRANSPORT,
    CRUISER, BATTLESHIP, MOTHERSHIP;
    @Override
    public String toString() {
        String id = name();
        String lower = id.substring(1).toLowerCase();
        return id.charAt(0) + lower;
    }
    public static void main(String[] args) {
        Stream.of(values())
                .forEach(System.out::println);
    }
}
```

输出为:

```
Scout
Cargo
Transport
Cruiser
Battleship
Mothership
```

toString() 方法通过调用 name() 方法取得 SpaceShip 的名字，然后将其修改为只有首字母大写的格式。

23.3 switch 语句中的 enum

在 switch 中使用 enum，是 enum 提供的一项非常便利的功能。一般来说，在 switch 中只能使用整数值，而枚举实例天生就具备整数值的次序，并且可以通过 ordinal() 方法取得其次序（显然编译器帮我们做了类似的工作），因此我们可以在 switch 语句中使用 enum。

虽然一般情况下我们必须使用 enum 类型来修饰一个 enum 实例，但是在 case 语句中却不必如此。下面的例子使用 enum 构造了一个小型状态机:

```
// enums/TrafficLight.java
// Enums in switch statements
// Define an enum type:
enum Signal { GREEN, YELLOW, RED, }

public class TrafficLight {
    Signal color = Signal.RED;
    public void change() {
        switch(color) {
           // Note you don't have to say Signal.RED
            // in the case statement:
            case RED: color = Signal.GREEN;
```

```
                break;
            case GREEN: color = Signal.YELLOW;
                break;
            case YELLOW: color = Signal.RED;
                break;
        }
    }
    @Override
    public String toString() {
        return "The traffic light is " + color;
    }
    public static void main(String[] args) {
        TrafficLight t = new TrafficLight();
        for(int i = 0; i < 7; i++) {
            System.out.println(t);
            t.change();
        }
    }
}
```

输出为:

```
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
```

编译器并没有抱怨 switch 中没有 default 语句，但这并不是因为每一个 Signal 都有对应的 case 语句。如果你注释掉其中的某个 case 语句，编译器同样不会抱怨什么。这意味着，你必须确保自己覆盖了所有的分支。但是，如果在 case 语句中调用 return，那么编译器就会抱怨缺少 default 语句了。这与是否覆盖了 enum 的所有实例无关。

23.4 values 方法的神秘之处

前面已经提到，编译器为你创建的 enum 类都继承自 Enum 类。然而，如果你研究一下 Enum 类就会发现，它并没有 values() 方法。可我们明明已经用过该方法了，难道存在某种“隐藏的”方法吗？我们可以利用反射机制编写一个简单的程序，来查看其中的究竟:

```
// enums/Reflection.java
// Analyzing enums using reflection
import java.lang.reflect.*;
import java.util.*;
import onjava.*;
```

```
enum Explore { HERE, THERE }
public class Reflection {
    public static
    Set<String> analyze(Class<?> enumClass) {
        System.out.println(
                "_____ Analyzing " + enumClass + " _____");
        System.out.println("Interfaces:");
        for(Type t : enumClass.getGenericInterfaces())
            System.out.println(t);
        System.out.println(
                "Base: " + enumClass.getSuperclass());
        System.out.println("Methods: ");
        Set<String> methods = new TreeSet<>();
        for(Method m : enumClass.getMethods())
            methods.add(m.getName());
        System.out.println(methods);
        return methods;
    }
    public static void main(String[] args) {
        Set<String> exploreMethods =
                analyze(Explore.class);
        Set<String> enumMethods = analyze(Enum.class);
        System.out.println(
                "Explore.containsAll(Enum)? " +
                        exploreMethods.containsAll(enumMethods));
        System.out.print("Explore.removeAll(Enum): ");
        exploreMethods.removeAll(enumMethods);
        System.out.println(exploreMethods);
// Decompile the code for the enum:
        OSExecute.command(
                "javap -cp build/classes/main Explore");
    }
}
```

输出为:

```
_____ Analyzing class Explore _____
Interfaces:
Base: class java.lang.Enum
Methods:
[compareTo, equals, getClass, getDeclaringClass,
hashCode, name, notify, notifyAll, ordinal, toString,
valueOf, values, wait]
_____ Analyzing class java.lang.Enum _____
Interfaces:
java.lang.Comparable<E>
interface java.io.Serializable
Base: class java.lang.Object
Methods:
[compareTo, equals, getClass, getDeclaringClass,
hashCode, name, notify, notifyAll, ordinal, toString,
```

```
valueOf, wait]
Explore.containsAll(Enum)? true
Explore.removeAll(Enum): [values]
Compiled from "Reflection.java"
final class Explore extends java.lang.Enum<Explore> {
  public static final Explore HERE;
  public static final Explore THERE;
  public static Explore[] values();
  public static Explore valueOf(java.lang.String);
  static {};
}
```

答案是，values() 是由编译器添加的 static 方法。可以看出，在创建 Explore 的过程中，编译器还为其添加了 valueOf() 方法。这可能有点令人迷惑，Enum 类不是已经有 valueOf() 方法了吗。

不过 Enum 中的 valueOf() 方法需要两个参数，而这个新增的方法只需一个参数。由于这里使用的 Set 只存储方法的名字，而不考虑方法的签名，所以在调用 Explore.removeAll(Enum) 之后，就只剩下 [values] 了。

从最后的输出中可以看到，编译器将 Explore 标记为 final 类，所以无法继承自 enum，其中还有一个 static 的初始化子句，稍后我们将学习如何重定义该句。

由于擦除效应（在泛型 章节中介绍过），反编译无法得到 Enum 的完整信息，所以它展示的 Explore 的父类只是一个原始的 Enum，而非事实上的 Enum<Explore>。

由于 values() 方法是由编译器插入到 enum 定义中的 static 方法，所以，如果你将 enum 实例向上转型为 Enum，那么 values() 方法就不可访问了。不过，在 Class 中有一个 getEnumConstants0 方法，所以即便 Enum 接口中没有 values0 方法，我们仍然可以通过 Class 对象取得所有 enum 实例。

```java
// enums/UpcastEnum.java
// No values() method if you upcast an enum
enum Search { HITHER, YON }
public class UpcastEnum {
    public static void main(String[] args) {
        Search[] vals = Search.values();
        Enum e = Search.HITHER; // Upcast
// e.values(); // No values() in Enum
        for(Enum en : e.getClass().getEnumConstants())
            System.out.println(en);
    }
}
```

输出为:

```
HITHER
YON
```

因为 getEnumConstants() 是 Class 上的方法，所以你甚至可以对不是枚举的类调用此方法:

```java
// enums/NonEnum.java
```

```
public class NonEnum {
    public static void main(String[] args) {
        Class<Integer> intClass = Integer.class;
        try {
            for(Object en : intClass.getEnumConstants())
                System.out.println(en);
        } catch(Exception e) {
            System.out.println("Expected: " + e);
        }
    }
}
```

输出为:

```
Expected: java.lang.NullPointerException
```

只不过，此时该方法返回 null，所以当你试图使用其返回的结果时会发生异常。

23.5 实现而非继承

我们已经知道，所有的 enum 都继承自 Java.lang.Enum 类。由于 Java 不支持多重继承，所以你的 enum 不能再继承其他类:

```
enum NotPossible extends Pet { ... // Won't work
```

然而，在我们创建一个新的 enum 时，可以同时实现一个或多个接口:

```
// enums/cartoons/EnumImplementation.java
// An enum can implement an interface
// {java enums.cartoons.EnumImplementation}
package enums.cartoons;
import java.util.*;
import java.util.function.*;
enum CartoonCharacter
        implements Supplier<CartoonCharacter> {
    SLAPPY, SPANKY, PUNCHY,
    SILLY, BOUNCY, NUTTY, BOB;
    private Random rand =
            new Random(47);
    @Override
    public CartoonCharacter get() {
        return values()[rand.nextInt(values().length)];
    }
}
public class EnumImplementation {
    public static <T> void printNext(Supplier<T> rg) {
        System.out.print(rg.get() + ", ");
    }
```

```
    public static void main(String[] args) {
// Choose any instance:
        CartoonCharacter cc = CartoonCharacter.BOB;
        for(int i = 0; i < 10; i++)
            printNext(cc);
    }
}
```

输出为:

```
BOB, PUNCHY, BOB, SPANKY, NUTTY, PUNCHY, SLAPPY, NUTTY,
NUTTY, SLAPPY,
```

这个结果有点奇怪，不过你必须要有一个 enum 实例才能调用其上的方法。现在，在任何接受 Supplier 参数的方法中，例如 printNext()，都可以使用 CartoonCharacter。

23.6 随机选择

就像你在 CartoonCharacter.get() 中看到的那样，本章中的很多示例都需要从 enum 实例中进行随机选择。我们可以利用泛型，从而使得这个工作更一般化，并将其加入到我们的工具库中。

```
// onjava/Enums.java
package onjava;
import java.util.*;
public class Enums {
    private static Random rand = new Random(47);

    public static <T extends Enum<T>> T random(Class<T> ec) {
        return random(ec.getEnumConstants());
    }

    public static <T> T random(T[] values) {
        return values[rand.nextInt(values.length)];
    }
}
```

古怪的语法 <T extends Enum<T>> 表示 T 是一个 enum 实例。而将 Class<T> 作为参数的话，我们就可以利用 Class 对象得到 enum 实例的数组了。重载后的 random() 方法只需使用 T[] 作为参数，因为它并不会调用 Enum 上的任何操作，它只需从数组中随机选择一个元素即可。这样，最终的返回类型正是 enum 的类型。

下面是 random() 方法的一个简单示例:

```
// enums/RandomTest.java
import onjava.*;
enum Activity { SITTING, LYING, STANDING, HOPPING,
    RUNNING, DODGING, JUMPING, FALLING, FLYING }
```

Ready, fire, aim: the fast approach to software development. Ready, aim, aim, aim, aim: the slow approach to software development. — Anonymous

```java

public class RandomTest {
    public static void main(String[] args) {
        for(int i = 0; i < 20; i++)
            System.out.print(
                    Enums.random(Activity.class) + " ");
    }
}
```

输出为:

```
STANDING FLYING RUNNING STANDING RUNNING STANDING LYING
DODGING SITTING RUNNING HOPPING HOPPING HOPPING RUNNING
STANDING LYING FALLING RUNNING FLYING LYING
```

23.7 使用接口组织枚举

无法从 enum 继承子类有时很令人沮丧。这种需求有时源自我们希望扩展原 enum 中的元素，有时是因为我们希望使用子类将一个 enum 中的元素进行分组。

在一个接口的内部，创建实现该接口的枚举，以此将元素进行分组，可以达到将枚举元素分类组织的目的。举例来说，假设你想用 enum 来表示不同类别的食物，同时还希望每个 enum 元素仍然保持 Food 类型。那可以这样实现:

```java
// enums/menu/Food.java
// Subcategorization of enums within interfaces
package enums.menu;
public interface Food {
    enum Appetizer implements Food {
        SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
        LASAGNE, BURRITO, PAD_THAI,
        LENTILS, HUMMOUS, VINDALOO;
    }
    enum Dessert implements Food {
        TIRAMISU, GELATO, BLACK_FOREST_CAKE,
        FRUIT, CREME_CARAMEL;
    }
    enum Coffee implements Food {
        BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
        LATTE, CAPPUCCINO, TEA, HERB_TEA;
    }
}
```

对于 enum 而言，实现接口是使其子类化的唯一办法，所以嵌入在 Food 中的每个 enum 都实现了 Food 接口。现在，在下面的程序中，我们可以说“所有东西都是某种类型的 Food”。

准备，开火，瞄准：快速的软件开发方法。准备，瞄准，瞄准，瞄准，瞄准：缓慢的软件开发方法。

```java
// enums/menu/TypeOfFood.java
// {java enums.menu.TypeOfFood}
package enums.menu;
import static enums.menu.Food.*;
public class TypeOfFood {
    public static void main(String[] args) {
        Food food = Appetizer.SALAD;
        food = MainCourse.LASAGNE;
        food = Dessert.GELATO;
        food = Coffee.CAPPUCCINO;
    }
}
```

如果 enum 类型实现了 Food 接口，那么我们就可以将其实例向上转型为 Food，所以上例中的所有东西都是 Food。

然而，当你需要与一大堆类型打交道时，接口就不如 enum 好用了。例如，如果你想创建一个“校举的枚举”，那么可以创建一个新的 enum，然后用其实例包装 Food 中的每一个 enum 类：

```java
// enums/menu/Course.java
package enums.menu;
import onjava.*;
public enum Course {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
    private Course(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
}
```

每一个 Course 的实例都将其对应的 Class 对象作为构造器的参数。通过 getEnumConstants0 方法，可以从该 Class 对象中取得某个 Food 子类的所有 enum 实例。这些实例在 randomSelection() 中被用到。因此，通过从每一个 Course 实例中随机地选择一个 Food，我们便能够生成一份菜单：

```java
// enums/menu/Meal.java
// {java enums.menu.Meal}
package enums.menu;
public class Meal {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            for(Course course : Course.values()) {
                Food food = course.randomSelection();
                System.out.println(food);
```

```
            }
            System.out.println("***");
        }
    }
}
```

输出为:

```
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
***
SOUP
VINDALOO
FRUIT
TEA
***
SALAD
BURRITO
FRUIT
TEA
***
SALAD
BURRITO
CREME_CARAMEL
LATTE
***
SOUP
BURRITO
TIRAMISU
ESPRESSO
***
```

在这个例子中，我们通过遍历每一个 Course 实例来获得“枚举的枚举”的值。稍后，在 VendingMachine.java 中，我们会看到另一种组织枚举实例的方式，但其也有一些其他的限制。

此外，还有一种更简洁的管理枚举的办法，就是将一个 enum 嵌套在另一个 enum 内。就像这样:

```
// enums/SecurityCategory.java
// More succinct subcategorization of enums
import onjava.*;
enum SecurityCategory {
    STOCK(Security.Stock.class),
    BOND(Security.Bond.class);
    Security[] values;
    SecurityCategory(Class<? extends Security> kind) {
        values = kind.getEnumConstants();
    }
    interface Security {
```

```
        enum Stock implements Security {
            SHORT, LONG, MARGIN
        }
        enum Bond implements Security {
            MUNICIPAL, JUNK
        }
    }
    public Security randomSelection() {
        return Enums.random(values);
    }
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            SecurityCategory category =
                    Enums.random(SecurityCategory.class);
            System.out.println(category + ": " +
                    category.randomSelection());
        }
    }
}
```

输出为:

```
BOND: MUNICIPAL
BOND: MUNICIPAL
STOCK: MARGIN
STOCK: MARGIN
BOND: JUNK
STOCK: SHORT
STOCK: LONG
STOCK: LONG
BOND: MUNICIPAL
BOND: JUNK
```

Security 接口的作用是将其所包含的 enum 组合成一个公共类型，这一点是有必要的。然后，SecurityCategory 才能将 Security 中的 enum 作为其构造器的参数使用，以起到组织的效果。

如果我们将这种方式应用于 Food 的例子，结果应该这样:

```
// enums/menu/Meal2.java
// {java enums.menu.Meal2}
package enums.menu;
import onjava.*;
public enum Meal2 {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
    private Meal2(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
```

```
    public interface Food {
        enum Appetizer implements Food {
            SALAD, SOUP, SPRING_ROLLS;
        }
        enum MainCourse implements Food {
            LASAGNE, BURRITO, PAD_THAI,
            LENTILS, HUMMOUS, VINDALOO;
        }
        enum Dessert implements Food {
            TIRAMISU, GELATO, BLACK_FOREST_CAKE,
            FRUIT, CREME_CARAMEL;
        }
        enum Coffee implements Food {
            BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
            LATTE, CAPPUCCINO, TEA, HERB_TEA;
        }
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            for(Meal2 meal : Meal2.values()) {
                Food food = meal.randomSelection();
                System.out.println(food);
            }
            System.out.println("***");
        }
    }
}
```

输出为:

```
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
***
SOUP
VINDALOO
FRUIT
TEA
***
SALAD
BURRITO
FRUIT
TEA
***
SALAD
BURRITO
CREME_CARAMEL
```

```
LATTE
***
SOUP
BURRITO
TIRAMISU
ESPRESSO
***
```

其实，这仅仅是重新组织了一下代码，不过多数情况下，这种方式使你的代码具有更清晰的结构。

23.8 使用 EnumSet 替代 Flags

Set 是一种集合，只能向其中添加不重复的对象。当然，enum 也要求其成员都是唯一的，所以 enumi 看起来也具有集合的行为。不过，由于不能从 enum 中删除或添加元素，所以它只能算是不太有用的集合。Java SE5 引入 EnumSet，是为了通过 enum 创建一种替代品，以替代传统的基于 int 的“位标志”。这种标志可以用来表示某种“开/关”信息，不过，使用这种标志，我们最终操作的只是一些 bit，而不是这些 bit 想要表达的概念，因此很容易写出令人难以理解的代码。

EnumSet 的设计充分考虑到了速度因素，因为它必须与非常高效的 bit 标志相竞争（其操作与 HashSet 相比，非常地快），就其内部而言，它（可能）就是将一个 long 值作为比特向量，所以 EnumSet 非常快速高效。使用 EnumSet 的优点是，它在说明一个二进制位是否存在时，具有更好的表达能力，并且无需担心性能。

EnumSet 中的元素必须来自一个 enum。下面的 enum 表示在一座大楼中，警报传感器的安放位置：

```java
// enums/AlarmPoints.java
package enums;
public enum AlarmPoints {
    STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3,
    OFFICE4, BATHROOM, UTILITY, KITCHEN
}
```

然后，我们用 EnumSet 来跟踪报警器的状态：

```java
// enums/EnumSets.java
// Operations on EnumSets
// {java enums.EnumSets}
package enums;
import java.util.*;
import static enums.AlarmPoints.*;
public class EnumSets {
    public static void main(String[] args) {
        EnumSet<AlarmPoints> points =
                EnumSet.noneOf(AlarmPoints.class); // Empty
        points.add(BATHROOM);
```

```
        System.out.println(points);
        points.addAll(
                EnumSet.of(STAIR1, STAIR2, KITCHEN));
        System.out.println(points);
        points = EnumSet.allOf(AlarmPoints.class);
        points.removeAll(
                EnumSet.of(STAIR1, STAIR2, KITCHEN));
        System.out.println(points);
        points.removeAll(
                EnumSet.range(OFFICE1, OFFICE4));
        System.out.println(points);
        points = EnumSet.complementOf(points);
        System.out.println(points);
    }
}
```

输出为:

```
[BATHROOM]
[STAIR1, STAIR2, BATHROOM, KITCHEN]
[LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM,
UTILITY]
[LOBBY, BATHROOM, UTILITY]
[STAIR1, STAIR2, OFFICE1, OFFICE2, OFFICE3, OFFICE4,
KITCHEN]
```

使用 static import 可以简化 enum 常量的使用。EnumSet 的方法的名字都相当直观，你可以查阅 JDK 文档找到其完整详细的描述。如果仔细研究了 EnunSet 的文档，你还会发现 of() 方法被重载了很多次，不但为可变数量参数进行了重载，而且为接收 2 至 5 个显式的参数的情况都进行了重载。这也从侧面表现了 EnumSet 对性能的关注。因为，其实只使用单独的 of() 方法解决可变参数已经可以解决整个问题了，但是对比显式的参数，会有一点性能损失。采用现在这种设计，当你只使用 2 到 5 个参数调用 of() 方法时，你可以调用对应的重载过的方法（速度稍快一点），而当你使用一个参数或多过 5 个参数时，你调用的将是使用可变参数的 of() 方法。注意，如果你只使用一个参数，编译器并不会构造可变参数的数组，所以与调用只有一个参数的方法相比，也就不会有额外的性能损耗。

EnumSet 的基础是 long，一个 long 值有 64 位，而一个 enum 实例只需一位 bit 表示其是否存在。也就是说，在不超过一个 long 的表达能力的情况下，你的 EnumSet 可以应用于最多不超过 64 个元素的 enum。如果 enum 超过了 64 个元素会发生什么呢?

```
// enums/BigEnumSet.java
import java.util.*;
public class BigEnumSet {
    enum Big { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9,
        A10, A11, A12, A13, A14, A15, A16, A17, A18, A19,
        A20, A21, A22, A23, A24, A25, A26, A27, A28, A29,
        A30, A31, A32, A33, A34, A35, A36, A37, A38, A39,
        A40, A41, A42, A43, A44, A45, A46, A47, A48, A49,
```

```
        A50, A51, A52, A53, A54, A55, A56, A57, A58, A59,
        A60, A61, A62, A63, A64, A65, A66, A67, A68, A69,
        A70, A71, A72, A73, A74, A75 }
    public static void main(String[] args) {
        EnumSet<Big> bigEnumSet = EnumSet.allOf(Big.class);
        System.out.println(bigEnumSet);
    }
}
```

输出为:

```
[A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12,
A13, A14, A15, A16, A17, A18, A19, A20, A21, A22, A23,
A24, A25, A26, A27, A28, A29, A30, A31, A32, A33, A34,
A35, A36, A37, A38, A39, A40, A41, A42, A43, A44, A45,
A46, A47, A48, A49, A50, A51, A52, A53, A54, A55, A56,
A57, A58, A59, A60, A61, A62, A63, A64, A65, A66, A67,
A68, A69, A70, A71, A72, A73, A74, A75]
```

显然，EnumSet 可以应用于多过 64 个元素的 enum，所以我猜测，Enum 会在必要的时候增加一个 long。

23.9 使用 EnumMap

EnumMap 是一种特殊的 Map，它要求其中的键（key）必须来自一个 enum，由于 enum 本身的限制，所以 EnumMap 在内部可由数组实现。因此 EnumMap 的速度很快，我们可以放心地使用 enum 实例在 EnumMap 中进行查找操作。不过，我们只能将 enum 的实例作为键来调用 put() 可方法，其他操作与使用一般的 Map 差不多。

下面的例子演示了*命令设计模式*的用法。一般来说，命令模式首先需要一个只有单一方法的接口，然后从该接口实现具有各自不同的行为的多个子类。接下来，程序员就可以构造命令对象，并在需要的时候使用它们了:

```
// enums/EnumMaps.java
// Basics of EnumMaps
// {java enums.EnumMaps}
package enums;
import java.util.*;
import static enums.AlarmPoints.*;
interface Command { void action(); }
public class EnumMaps {
    public static void main(String[] args) {
        EnumMap<AlarmPoints,Command> em =
                new EnumMap<>(AlarmPoints.class);
        em.put(KITCHEN,
                () -> System.out.println("Kitchen fire!"));
        em.put(BATHROOM,
```

```
                    () -> System.out.println("Bathroom alert!"));
            for(Map.Entry<AlarmPoints,Command> e:
                    em.entrySet()) {
                System.out.print(e.getKey() + ": ");
                e.getValue().action();
            }
            try { // If there's no value for a particular key:
                em.get(UTILITY).action();
            } catch(Exception e) {
                System.out.println("Expected: " + e);
            }
        }
}
```

输出为:

```
BATHROOM: Bathroom alert!
KITCHEN: Kitchen fire!
Expected: java.lang.NullPointerException
```

与 EnumSet 一样，enum 实例定义时的次序决定了其在 EnumMap 中的顺序。

main0 方法的最后部分说明，enum 的每个实例作为一个键，总是存在的。但是，如果你没有为这个键调用 put() 方法来存入相应的值的话，其对应的值就是 null。

与常量相关的方法（constant-specific methods 将在下一节中介绍）相比，EnumMap 有一个优点，那 EnumMap 允许程序员改变值对象，而常量相关的方法在编译期就被固定了。稍后你会看到，在你有多种类型的 enum，而且它们之间存在互操作的情况下，我们可以用 EnumMap 实现多路分发（multiple dispatching）。

23.10 常量特定方法

Java 的 enum 有一个非常有趣的特性，即它允许程序员为 enum 实例编写方法，从而为每个 enum 实例赋予各自不同的行为。要实现常量相关的方法，你需要为 enum 定义一个或多个 abstract 方法，然后为每个 enum 实例实现该抽象方法。参考下面的例子:

```
// enums/ConstantSpecificMethod.java
import java.util.*;
import java.text.*;
public enum ConstantSpecificMethod {
    DATE_TIME {
        @Override
        String getInfo() {
            return
                    DateFormat.getDateInstance()
                            .format(new Date());
        }
```

```
    },
    CLASSPATH {
        @Override
        String getInfo() {
            return System.getenv("CLASSPATH");
        }
    },
    VERSION {
        @Override
        String getInfo() {
            return System.getProperty("java.version");
        }
    };
    abstract String getInfo();
    public static void main(String[] args) {
        for(ConstantSpecificMethod csm : values())
            System.out.println(csm.getInfo());
    }
}
```

输出为:

```
May 9, 2017
C:\Users\Bruce\Documents\GitHub\on-
java\ExtractedExamples\\gradle\wrapper\gradle-
wrapper.jar
1.8.0_112
```

通过相应的 enum 实例，我们可以调用其上的方法。这通常也称为表驱动的代码（table-driven code，请注意它与前面提到的命令模式的相似之处）。

在面向对象的程序设计中，不同的行为与不同的类关联。而通过常量相关的方法，每个 enum 实例可以具备自己独特的行为，这似乎说明每个 enum 实例就像一个独特的类。在上面的例子中，enum 实例似乎被当作其“超类”ConstantSpecificMethod 来使用，在调用 getInfo() 方法时，体现出多态的行为。

然而，enum 实例与类的相似之处也仅限于此了。我们并不能真的将 enum 实例作为一个类型来使用:

```
// enums/NotClasses.java
// {javap -c LikeClasses}
enum LikeClasses {
    WINKEN {
        @Override
        void behavior() {
            System.out.println("Behavior1");
        }
    },
    BLINKEN {
        @Override
```

```
12          void behavior() {
13              System.out.println("Behavior2");
14          }
15      },
16      NOD {
17          @Override
18          void behavior() {
19              System.out.println("Behavior3");
20          }
21      };
22      abstract void behavior();
23  }
24  public class NotClasses {
25      // void f1(LikeClasses.WINKEN instance) {} // Nope
26  }
```

输出为（前 12 行）：

```
1   Compiled from "NotClasses.java"
2   abstract class LikeClasses extends
3   java.lang.Enum<LikeClasses> {
4   public static final LikeClasses WINKEN;
5   public static final LikeClasses BLINKEN;
6   public static final LikeClasses NOD;
7   public static LikeClasses[] values();
8   Code:
9   0: getstatic #2 // Field
10  $VALUES:[LLikeClasses;
11  3: invokevirtual #3 // Method
12  "[LLikeClasses;".clone:()Ljava/lang/Object;
13  ...
```

在方法 f1() 中，编译器不允许我们将一个 enum 实例当作 class 类型。如果我们分析一下编译器生成的代码，就知道这种行为也是很正常的。因为每个 enum 元素都是一个 LikeClasses 类型的 static final 实例。

同时，由于它们是 static 实例，无法访问外部类的非 static 元素或方法，所以对于内部的 enum 的实例而言，其行为与一般的内部类并不相同。

再看一个更有趣的关于洗车的例子。每个顾客在洗车时，都有一个选择菜单，每个选择对应一个不同的动作。可以将一个常量相关的方法关联到一个选择上，再使用一个 EnumSet 来保存客户的选择：

```
1   // enums/CarWash.java
2   import java.util.*;
3   public class CarWash {
4       public enum Cycle {
5           UNDERBODY {
6               @Override
7               void action() {
8                   System.out.println("Spraying the underbody");
```

```
            }
        },
        WHEELWASH {
            @Override
            void action() {
                System.out.println("Washing the wheels");
            }
        },
        PREWASH {
            @Override
            void action() {
                System.out.println("Loosening the dirt");
            }
        },
        BASIC {
            @Override
            void action() {
                System.out.println("The basic wash");
            }
        },
        HOTWAX {
            @Override
            void action() {
                System.out.println("Applying hot wax");
            }
        },
        RINSE {
            @Override
            void action() {
                System.out.println("Rinsing");
            }
        },
        BLOWDRY {
            @Override
            void action() {
                System.out.println("Blowing dry");
            }
        };
        abstract void action();
    }
    EnumSet<Cycle> cycles =
            EnumSet.of(Cycle.BASIC, Cycle.RINSE);
    public void add(Cycle cycle) {
        cycles.add(cycle);
    }
    public void washCar() {
        for(Cycle c : cycles)
            c.action();
    }
    @Override
    public String toString() {
```

```
        return cycles.toString();
    }
    public static void main(String[] args) {
        CarWash wash = new CarWash();
        System.out.println(wash);
        wash.washCar();
// Order of addition is unimportant:
        wash.add(Cycle.BLOWDRY);
        wash.add(Cycle.BLOWDRY); // Duplicates ignored
        wash.add(Cycle.RINSE);
        wash.add(Cycle.HOTWAX);
        System.out.println(wash);
        wash.washCar();
    }
}
```

输出为:

```
[BASIC, RINSE]
The basic wash
Rinsing
[BASIC, HOTWAX, RINSE, BLOWDRY]
The basic wash
Applying hot wax
Rinsing
Blowing dry
```

与使用匿名内部类相比较，定义常量相关方法的语法更高效、简洁。

这个例子也展示了 EnumSet 了一些特性。因为它是一个集合，所以对于同一个元素而言，只能出现一次，因此对同一个参数重复地调用 add0 方法会被忽略掉（这是正确的行为，因为一个 bit 位开关只能“打开”一次），同样地，向 EnumSet 添加 enum 实例的顺序并不重要，因为其输出的次序决定于 enum 实例定义时的次序。

除了实现 abstract 方法以外，程序员是否可以覆盖常量相关的方法呢？答案是肯定的，参考下面的程序:

```
// enums/OverrideConstantSpecific.java
public enum OverrideConstantSpecific {
    NUT, BOLT,
    WASHER {
        @Override
        void f() {
            System.out.println("Overridden method");
        }
    };
    void f() {
        System.out.println("default behavior");
    }
    public static void main(String[] args) {
```

Java 之于 JavaScript 就像 car 之于 carpet（地毯）。后者看上去像是前者的衍生物，实则是完全不同的两个东西。

```
        for(OverrideConstantSpecific ocs : values()) {
            System.out.print(ocs + ": ");
            ocs.f();
        }
    }
}
```

输出为:

```
NUT: default behavior
BOLT: default behavior
WASHER: Overridden method
```

虽然 enum 有某些限制，但是一般而言，我们还是可以将其看作是类。

23.10.1 使用 enum 的职责链

在职责链（Chain of Responsibility）设计模式中，程序员以多种不同的方式来解决一个问题，然后将它们链接在一起。当一个请求到来时，它遍历这个链，直到链中的某个解决方案能够处理该请求。

通过常量相关的方法，我们可以很容易地实现一个简单的职责链。我们以一个邮局的模型为例。邮局需要以尽可能通用的方式来处理每一封邮件，并且要不断尝试处理邮件，直到该邮件最终被确定为一封死信。其中的每一次尝试可以看作为一个策略（也是一个设计模式），而完整的处理方式列表就是一个职责链。

我们先来描述一下邮件。邮件的每个关键特征都可以用 enum 来表示。程序将随机地生成 Mail 对象，如果要减小一封邮件的 GeneralDelivery 为 YES 的概率，那最简单的方法就是多创建几个不是 YES 的 enum 实例，所以 enum 的定义看起来有点古怪。

我们看到 Mail 中有一个 randomMail() 方法，它负责随机地创建用于测试的邮件。而 generator() 方法生成一个 Iterable 对象，该对象在你调用 next() 方法时，在其内部使用 randomMail() 来创建 Mail 对象。这样的结构使程序员可以通过调用 Mail.generator() 方法，很容易地构造出一个 foreach 循环:

```
// enums/PostOffice.java
// Modeling a post office
import java.util.*;
import onjava.*;
class Mail {
    // The NO's reduce probability of random selection:
    enum GeneralDelivery {YES,NO1,NO2,NO3,NO4,NO5}
    enum Scannability {UNSCANNABLE,YES1,YES2,YES3,YES4}
    enum Readability {ILLEGIBLE,YES1,YES2,YES3,YES4}
    enum Address {INCORRECT,OK1,OK2,OK3,OK4,OK5,OK6}
    enum ReturnAddress {MISSING,OK1,OK2,OK3,OK4,OK5}
    GeneralDelivery generalDelivery;
    Scannability scannability;
```

```
    Readability readability;
    Address address;
    ReturnAddress returnAddress;
    static long counter = 0;
    long id = counter++;
    @Override
    public String toString() { return "Mail " + id; }
    public String details() {
        return toString() +
                ", General Delivery: " + generalDelivery +
                ", Address Scanability: " + scannability +
                ", Address Readability: " + readability +
                ", Address Address: " + address +
                ", Return address: " + returnAddress;
    }
    // Generate test Mail:
    public static Mail randomMail() {
        Mail m = new Mail();
        m.generalDelivery =
                Enums.random(GeneralDelivery.class);
        m.scannability =
                Enums.random(Scannability.class);
        m.readability =
                Enums.random(Readability.class);
        m.address = Enums.random(Address.class);
        m.returnAddress =
                Enums.random(ReturnAddress.class);
        return m;
    }
    public static
    Iterable<Mail> generator(final int count) {
        return new Iterable<Mail>() {
            int n = count;
            @Override
            public Iterator<Mail> iterator() {
                return new Iterator<Mail>() {
                    @Override
                    public boolean hasNext() {
                        return n-- > 0;
                    }
                    @Override
                    public Mail next() {
                        return randomMail();
                    }
                    @Override
                    public void remove() { // Not implemented
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
```

```
    }
}
public class PostOffice {
    enum MailHandler {
        GENERAL_DELIVERY {
            @Override
            boolean handle(Mail m) {
                switch(m.generalDelivery) {
                    case YES:
                        System.out.println(
                                "Using general delivery for " + m);
                        return true;
                    default: return false;
                }
            }
        },
        MACHINE_SCAN {
            @Override
            boolean handle(Mail m) {
                switch(m.scannability) {
                    case UNSCANNABLE: return false;
                    default:
                        switch(m.address) {
                            case INCORRECT: return false;
                            default:
                                System.out.println(
                                        "Delivering "+ m + "
                                           automatically");
                                return true;
                        }
                }
            }
        },
        VISUAL_INSPECTION {
            @Override
            boolean handle(Mail m) {
                switch(m.readability) {
                    case ILLEGIBLE: return false;
                    default:
                        switch(m.address) {
                            case INCORRECT: return false;
                            default:
                                System.out.println(
                                        "Delivering " + m + " normally"
                                           );
                                return true;
                        }
                }
            }
        },
        RETURN_TO_SENDER {
```

Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration.
— Stan Kelly-Bootle

```
            @Override
            boolean handle(Mail m) {
                switch(m.returnAddress) {
                    case MISSING: return false;
                    default:
                        System.out.println(
                                "Returning " + m + " to sender");
                        return true;
                }
            }
        };
        abstract boolean handle(Mail m);
    }
    static void handle(Mail m) {
        for(MailHandler handler : MailHandler.values())
            if(handler.handle(m))
                return;
        System.out.println(m + " is a dead letter");
    }
    public static void main(String[] args) {
        for(Mail mail : Mail.generator(10)) {
            System.out.println(mail.details());
            handle(mail);
            System.out.println("*****");
        }
    }
}
```

输出为:

```
Mail 0, General Delivery: NO2, Address Scanability:
UNSCANNABLE, Address Readability: YES3, Address
Address: OK1, Return address: OK1
Delivering Mail 0 normally
*****
Mail 1, General Delivery: NO5, Address Scanability:
YES3, Address Readability: ILLEGIBLE, Address Address:
OK5, Return address: OK1
Delivering Mail 1 automatically
*****
Mail 2, General Delivery: YES, Address Scanability:
YES3, Address Readability: YES1, Address Address: OK1,
Return address: OK5
Using general delivery for Mail 2
*****
Mail 3, General Delivery: NO4, Address Scanability:
YES3, Address Readability: YES1, Address Address:
INCORRECT, Return address: OK4
Returning Mail 3 to sender
*****
Mail 4, General Delivery: NO4, Address Scanability:
```

数组的下标一定要从 0 或 1 开始吗？我认为，我采取的中间值 0.5 没有经过适当考虑就被驳回了。

```
UNSCANNABLE, Address Readability: YES1, Address
Address: INCORRECT, Return address: OK2
Returning Mail 4 to sender
*****
Mail 5, General Delivery: NO3, Address Scanability:
YES1, Address Readability: ILLEGIBLE, Address Address:
OK4, Return address: OK2
Delivering Mail 5 automatically
*****
Mail 6, General Delivery: YES, Address Scanability:
YES4, Address Readability: ILLEGIBLE, Address Address:
OK4, Return address: OK4
Using general delivery for Mail 6
*****
Mail 7, General Delivery: YES, Address Scanability:
YES3, Address Readability: YES4, Address Address: OK2,
Return address: MISSING
Using general delivery for Mail 7
*****
Mail 8, General Delivery: NO3, Address Scanability:
YES1, Address Readability: YES3, Address Address:
INCORRECT, Return address: MISSING
Mail 8 is a dead letter
*****
Mail 9, General Delivery: NO1, Address Scanability:
UNSCANNABLE, Address Readability: YES2, Address
Address: OK1, Return address: OK4
Delivering Mail 9 normally
*****
```

职责链由 enum MailHandler 实现，而 enum 定义的次序决定了各个解决策略在应用时的次序。对每一封邮件，都要按此顺序尝试每个解决策略，直到其中一个能够成功地处理该邮件，如果所有的策略都失败了，那么该邮件将被判定为一封死信。

23.10.2 使用 enum 的状态机

枚举类型非常适合用来创建状态机。一个状态机可以具有有限个特定的状态，它通常根据输入，从一个状态转移到下一个状态，不过也可能存在瞬时状态（transient states），而一旦任务执行结束，状态机就会立刻离开瞬时状态。

每个状态都具有某些可接受的输入，不同的输入会使状态机从当前状态转移到不同的新状态。由于 enum 对其实例有严格限制，非常适合用来表现不同的状态和输入。一般而言，每个状态都具有一些相关的输出。

自动售贷机是一个很好的状态机的例子。首先，我们用一个 enum 定义各种输入：

```
// enums/Input.java
import java.util.*;
```

```
public enum Input {
    NICKEL(5), DIME(10), QUARTER(25), DOLLAR(100),
    TOOTHPASTE(200), CHIPS(75), SODA(100), SOAP(50),
    ABORT_TRANSACTION {
        @Override
        public int amount() { // Disallow
            throw new RuntimeException("ABORT.amount()");
        }
    },
    STOP { // This must be the last instance.
        @Override
        public int amount() { // Disallow
            throw new RuntimeException("SHUT_DOWN.amount()");
        }
    };
    int value; // In cents
    Input(int value) { this.value = value; }
    Input() {}
    int amount() { return value; }; // In cents
    static Random rand = new Random(47);
    public static Input randomSelection() {
        // Don't include STOP:
        return values()[rand.nextInt(values().length - 1)];
    }
}
```

注意，除了两个特殊的 Input 实例之外，其他的 Input 都有相应的价格，因此在接口中定义了 amount（方法。然而，对那两个特殊 Input 实例而言，调用 amount（方法并不合适，所以如果程序员调用它们的 amount）方法就会有异常抛出（在接口内定义了一个方法，然后在你调用该方法的某个实现时就会抛出异常），这似乎有点奇怪，但由于 enum 的限制，我们不得不采用这种方式。

VendingMachine 对输入的第一个反应是将其归类为 Category enum 中的某一个 enum 实例，这可以通过 switch 实现。下面的例子演示了 enum 是如何使代码变得更加清晰且易于管理的：

```
// enums/VendingMachine.java
// {java VendingMachine VendingMachineInput.txt}
import java.util.*;
import java.io.IOException;
import java.util.function.*;
import java.nio.file.*;
import java.util.stream.*;
enum Category {
    MONEY(Input.NICKEL, Input.DIME,
            Input.QUARTER, Input.DOLLAR),
    ITEM_SELECTION(Input.TOOTHPASTE, Input.CHIPS,
            Input.SODA, Input.SOAP),
    QUIT_TRANSACTION(Input.ABORT_TRANSACTION),
    SHUT_DOWN(Input.STOP);
    private Input[] values;
```

```
    Category(Input... types) { values = types; }
    private static EnumMap<Input,Category> categories =
            new EnumMap<>(Input.class);
    static {
        for(Category c : Category.class.getEnumConstants())
            for(Input type : c.values)
                categories.put(type, c);
    }
    public static Category categorize(Input input) {
        return categories.get(input);
    }
}

public class VendingMachine {
    private static State state = State.RESTING;
    private static int amount = 0;
    private static Input selection = null;
    enum StateDuration { TRANSIENT } // Tagging enum
    enum State {
        RESTING {
            @Override
            void next(Input input) {
                switch(Category.categorize(input)) {
                    case MONEY:
                        amount += input.amount();
                        state = ADDING_MONEY;
                        break;
                    case SHUT_DOWN:
                        state = TERMINAL;
                    default:
                }
            }
        },
        ADDING_MONEY {
            @Override
            void next(Input input) {
                switch(Category.categorize(input)) {
                    case MONEY:
                        amount += input.amount();
                        break;
                    case ITEM_SELECTION:
                        selection = input;
                        if(amount < selection.amount())
                            System.out.println(
                                    "Insufficient money for " +
                                        selection);
                        else state = DISPENSING;
                        break;
                    case QUIT_TRANSACTION:
                        state = GIVING_CHANGE;
                        break;
```

```
                    case SHUT_DOWN:
                        state = TERMINAL;
                    default:
                }
            }
        },
        DISPENSING(StateDuration.TRANSIENT) {
            @Override
            void next() {
                System.out.println("here is your " + selection);
                amount -= selection.amount();
                state = GIVING_CHANGE;
            }
        },
        GIVING_CHANGE(StateDuration.TRANSIENT) {
            @Override
            void next() {
                if(amount > 0) {
                    System.out.println("Your change: " + amount);
                    amount = 0;
                }
                state = RESTING;
            }
        },
        TERMINAL {@Override
        void output() { System.out.println("Halted"); } };
        private boolean isTransient = false;
        State() {}
        State(StateDuration trans) { isTransient = true; }
        void next(Input input) {
            throw new RuntimeException("Only call " +
                    "next(Input input) for non-transient states");
        }
        void next() {
            throw new RuntimeException(
                    "Only call next() for " +
                            "StateDuration.TRANSIENT states");
        }
        void output() { System.out.println(amount); }
    }
    static void run(Supplier<Input> gen) {
        while(state != State.TERMINAL) {
            state.next(gen.get());
            while(state.isTransient)
                state.next();
            state.output();
        }
    }
    public static void main(String[] args) {
        Supplier<Input> gen = new RandomInputSupplier();
        if(args.length == 1)
```

```
            gen = new FileInputSupplier(args[0]);
        run(gen);
    }
}

// For a basic sanity check:
class RandomInputSupplier implements Supplier<Input> {
    @Override
    public Input get() {
        return Input.randomSelection();
    }
}

// Create Inputs from a file of ';'-separated strings:
class FileInputSupplier implements Supplier<Input> {
    private Iterator<String> input;
    FileInputSupplier(String fileName) {
        try {
            input = Files.lines(Paths.get(fileName))
                    .skip(1) // Skip the comment line
                    .flatMap(s -> Arrays.stream(s.split(";")))
                    .map(String::trim)
                    .collect(Collectors.toList())
                    .iterator();
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }
    @Override
    public Input get() {
        if(!input.hasNext())
            return null;
        return Enum.valueOf(Input.class, input.next().trim());
    }
}
```

输出为:

```
25
50
75
here is your CHIPS
0
100
200
here is your TOOTHPASTE
0
25
35
Your change: 35
0
```

```
25
35
Insufficient money for SODA
35
60
70
75
Insufficient money for SODA
75
Your change: 75
0
Halted
```

由于用 switch 语句从 enum 实例中进行选择是最常见的一种方式（请注意，为了使 enum 在 switch 语句中的使用变得简单，我们是需要付出其他代价的），所以，我们经常遇到这样的问题：将多个 enum 进行分类时，“我们希望在什么 enum 中使用 switch 语句？”我们通过 VendingMachine 的例子来研究一下这个问题。对于每一个 State，我们都需要在输入动作的基本分类中进行查找：用户塞入钞票，选择了某个货物，操作被取消，以及机器停止。然而，在这些基本分类之下，我们又可以塞入不同类型的钞票，可以选择不同的货物。Category enum 将不同类型的 Input 进行分组，因而，可以使用 categorize0 方法为 switch 语句生成恰当的 Cateroy 实例。并且，该方法使用的 EnumMap 确保了在其中进行查询时的效率与安全。

如果读者仔细研究 VendingMachine 类，就会发现每种状态的不同之处，以及对于输入的不同响应，其中还有两个瞬时状态。在 run() 方法中，状态机等待着下一个 Input，并一直在各个状态中移动，直到它不再处于瞬时状态。

通过两种不同的 Generator 对象，我们可以使用不同的 Supplier 对象来测试 VendingMachine，首先是 RandomInputSupplier，它会不停地生成除了 SHUT-DOWN 之外的各种输入。通过长时间地运行 RandomInputSupplier，可以起到健全测试（sanity test）的作用，能够确保该状态机不会进入一个错误状态。另一个是 FileInputSupplier，使用文件以文本的方式来描述输入，然后将它们转换成 enum 实例，并创建对应的 Input 对象。上面的程序使用的正是如下的文本文件：

```
// enums/VendingMachineInput.txt
QUARTER; QUARTER; QUARTER; CHIPS;
DOLLAR; DOLLAR; TOOTHPASTE;
QUARTER; DIME; ABORT_TRANSACTION;
QUARTER; DIME; SODA;
QUARTER; DIME; NICKEL; SODA;
ABORT_TRANSACTION;
STOP;
```

FileInputSupplier 构造函数将此文件转换为流，并跳过注释行。然后它使用 String.split() 以分号进行分割。这会生成一个 String 数组，并可以通过将其转换为 Stream，然后应用 flatMap() 来将其输入到流中。其输出结果将去除所有空格空格，并转换为 List<String>，且从中获取 Iterator<String>。

这种设计有一个缺陷，它要求 enum State 实例访问的 VendingMachine 属性必须声明为 static，

复制粘贴是一个设计错误。

这意味着，你只能有一个 VendingMachine 实例。不过如果我们思考一下实际的（嵌入式 Java）应用，这也许并不是一个大问题，因为在一台机器上，我们可能只有一个应用程序。

23.11 多路分发

当你要处理多种交互类型时，程序可能会变得相当杂乱。举例来说，如果一个系统要分析和执行数学表达式。我们可能会声明 Number.plus(Number)，Number.multiple(Number) 等等，其中 Number 是各种数字对象的超类。然而，当你声明 a.plus(b) 时，你并不知道 a 或 b 的确切类型，那你如何能让它们正确地交互呢?

你可能从未思考过这个问题的答案.Java 只支持单路分发。也就是说，如果要执行的操作包含了不止一个类型未知的对象时，那么 Java 的动态绑定机制只能处理其中一个的类型。这就无法解决我们上面提到的问题。所以，你必须自己来判定其他的类型，从而实现自己的动态线定行为。

解决上面问题的办法就是多路分发（在那个例子中，只有两个分发，一般称之为两路分发）. 多态只能发生在方法调用时，所以，如果你想使用两路分发，那么就必须有两个方法调用：第一个方法调用决定第一个未知类型，第二个方法调用决定第二个未知的类型。要利用多路分发，程序员必须为每一个类型提供一个实际的方法调用，如果你要处理两个不同的类型体系，就需要为每个类型体系执行一个方法调用。一般而言，程序员需要有设定好的某种配置，以便一个方法调用能够引出更多的方法调用，从而能够在这个过程中处理多种类型。为了达到这种效果，我们需要与多个方法一同工作：因为每个分发都需要一个方法调用。在下面的例子中（实现了“石头、剪刀、布”游戏，也称为 RoShamBo）对应的方法是 compete() 和 eval()，二者都是同一个类型的成员，它们可以产生三种 Outcome 实例中的一个作为结果：

```
// enums/Outcome.java
package enums;
public enum Outcome { WIN, LOSE, DRAW }
// enums/RoShamBo1.java
// Demonstration of multiple dispatching
// {java enums.RoShamBo1}
package enums;
        import java.util.*;
        import static enums.Outcome.*;
interface Item {
    Outcome compete(Item it);
    Outcome eval(Paper p);
    Outcome eval(Scissors s);
    Outcome eval(Rock r);
}
class Paper implements Item {
    @Override
    public Outcome compete(Item it) {
        return it.eval(this);
```

```
    }
    @Override
    public Outcome eval(Paper p) { return DRAW; }
    @Override
    public Outcome eval(Scissors s) { return WIN; }
    @Override
    public Outcome eval(Rock r) { return LOSE; }
    @Override
    public String toString() { return "Paper"; }
}
class Scissors implements Item {
    @Override
    public Outcome compete(Item it) {
        return it.eval(this);
    }
    @Override
    public Outcome eval(Paper p) { return LOSE; }
    @Override
    public Outcome eval(Scissors s) { return DRAW; }
    @Override
    public Outcome eval(Rock r) { return WIN; }
    @Override
    public String toString() { return "Scissors"; }
}
class Rock implements Item {
    @Override
    public Outcome compete(Item it) {
        return it.eval(this);
    }
    @Override
    public Outcome eval(Paper p) { return WIN; }
    @Override
    public Outcome eval(Scissors s) { return LOSE; }
    @Override
    public Outcome eval(Rock r) { return DRAW; }
    @Override
    public String toString() { return "Rock"; }
}
public class RoShamBo1 {
    static final int SIZE = 20;
    private static Random rand = new Random(47);
    public static Item newItem() {
        switch(rand.nextInt(3)) {
            default:
            case 0: return new Scissors();
            case 1: return new Paper();
            case 2: return new Rock();
        }
    }
    public static void match(Item a, Item b) {
        System.out.println(
```

```
                    a + " vs. " + b + ": " + a.compete(b));
    }
    public static void main(String[] args) {
        for(int i = 0; i < SIZE; i++)
            match(newItem(), newItem());
    }
}
```

输出为:

```
Rock vs. Rock: DRAW
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Scissors vs. Scissors: DRAW
Scissors vs. Paper: WIN
Rock vs. Paper: LOSE
Paper vs. Paper: DRAW
Rock vs. Paper: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Rock vs. Scissors: WIN
Rock vs. Paper: LOSE
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
```

Item 是这几种类型的接口，将会被用作多路分发。RoShamBo1.match() 有两个 Item 参数，通过调用 Item.compete90) 方法开始两路分发。要判定 a 的类型，分发机制会在 a 的实际类型的 compete（内部起到分发的作用。compete() 方法通过调用 eval() 来为另一个类型实现第二次分法。

将自身（this）作为参数调用 evalo，能够调用重载过的 eval() 方法，这能够保留第一次分发的类型信息。当第二次分发完成时，你就能够知道两个 Item 对象的具体类型了。

要配置好多路分发需要很多的工序，不过要记住，它的好处在于方法调用时的优雅的话法，这避免了在一个方法中判定多个对象的类型的丑陋代码，你只需说，“嘿，你们两个，我不在乎你们是什么类型，请你们自己交流！”不过，在使用多路分发前，请先明确，这种优雅的代码对你确实有重要的意义。

23.11.1 使用 enum 分发

直接将 RoShamBol.java 翻译为基于 enum 的版本是有问题的，因为 enum 实例不是类型，不能将 enum 实例作为参数的类型，所以无法重载 eval() 方法。不过，还有很多方式可以实现多路分发，并从 enum 中获益。

一种方式是使用构造器来初始化每个 enum 实例，并以“一组”结果作为参数。这二者放在一块，形成了类似查询表的结构：

```
// enums/RoShamBo2.java
// Switching one enum on another
// {java enums.RoShamBo2}
package enums;
import static enums.Outcome.*;
public enum RoShamBo2 implements Competitor<RoShamBo2> {
    PAPER(DRAW, LOSE, WIN),
    SCISSORS(WIN, DRAW, LOSE),
    ROCK(LOSE, WIN, DRAW);
    private Outcome vPAPER, vSCISSORS, vROCK;
    RoShamBo2(Outcome paper,
              Outcome scissors, Outcome rock) {
        this.vPAPER = paper;
        this.vSCISSORS = scissors;
        this.vROCK = rock;
    }
    @Override
    public Outcome compete(RoShamBo2 it) {
        switch(it) {
            default:
            case PAPER: return vPAPER;
            case SCISSORS: return vSCISSORS;
            case ROCK: return vROCK;
        }
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo2.class, 20);
    }
}
```

输出为：

```
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
```

删除的代码是经过调试的代码。

```
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
```

在 compete() 方法中，一旦两种类型都被确定了，那么唯一的操作就是返回结果 Outcome 然而，你可能还需要调用其他的方法，（例如）甚至是调用在构造器中指定的某个命令对象上的方法。

RoShamBo2.javal 之前的例子短小得多，而且更直接，更易于理解。注意，我们仍然是使用两路分发来判定两个对象的类型。在 RoShamBol.java 中，两次分发都是通过实际的方法调用实现，而在这个例子中，只有第一次分发是实际的方法调用。第二个分发使用的是 switch，不过这样做是安全的，因为 enum 限制了 switch 语句的选择分支。

在代码中，enum 被单独抽取出来，因此它可以应用在其他例子中。首先，Competitor 接口定义了一种类型，该类型的对象可以与另一个 Competitor 相竞争：

```java
// enums/Competitor.java
// Switching one enum on another
package enums;
public interface Competitor<T extends Competitor<T>> {
    Outcome compete(T competitor);
}
```

然后，我们定义两个 static 方法（static 可以避免显式地指明参数类型），第一个是 match() 方法，它会为一个 Competitor 对象调用 compete() 方法，并与另一个 Competitor 对象作比较。在这个例子中，我们看到，match()）方法的参数需要是 Competitor<T> 类型。但是在 play() 方法中，类型参数必须同时是 Enum<T> 类型（因为它将在 Enums.random() 中使用）和 Competitor<T> 类型因为它将被传递给 match() 方法）：

```java
// enums/RoShamBo.java
// Common tools for RoShamBo examples
package enums;
import onjava.*;
public class RoShamBo {
    public static <T extends Competitor<T>>
    void match(T a, T b) {
        System.out.println(
                a + " vs. " + b + ": " + a.compete(b));
    }
    public static <T extends Enum<T> & Competitor<T>>
    void play(Class<T> rsbClass, int size) {
        for(int i = 0; i < size; i++)
            match(Enums.random(rsbClass),Enums.random(rsbClass));
    }
}
```

play() 方法没有将类型参数 T 作为返回值类型，因此，似乎我们应该在 Class<T> 中使用通配符

If debugging is the process of removing software bugs, then programming must be the process of putting them in.
— Edsger Dijkstra

来代替上面的参数声明。然而，通配符不能扩展多个基类，所以我们必须采用以上的表达式。

23.11.2 使用常量相关的方法

常量相关的方法允许我们为每个 enum 实例提供方法的不同实现，这使得常量相关的方法似乎是实现多路分发的完美解决方案。不过，通过这种方式，enum 实例虽然可以具有不同的行为，但它们仍然不是类型，不能将其作为方法签名中的参数类型来使用。最好的办法是将 enum 用在 switch 语句中，见下例：

```
// enums/RoShamBo3.java
// Using constant-specific methods
// {java enums.RoShamBo3}
package enums;
import static enums.Outcome.*;
public enum RoShamBo3 implements Competitor<RoShamBo3> {
    PAPER {
        @Override
        public Outcome compete(RoShamBo3 it) {
            switch(it) {
                default: // To placate the compiler
                case PAPER: return DRAW;
                case SCISSORS: return LOSE;
                case ROCK: return WIN;
            }
        }
    },
    SCISSORS {
        @Override
        public Outcome compete(RoShamBo3 it) {
            switch(it) {
                default:
                case PAPER: return WIN;
                case SCISSORS: return DRAW;
                case ROCK: return LOSE;
            }
        }
    },
    ROCK {
        @Override
        public Outcome compete(RoShamBo3 it) {
            switch(it) {
                default:
                case PAPER: return LOSE;
                case SCISSORS: return WIN;
                case ROCK: return DRAW;
            }
        }
    };
```

```
    @Override
    public abstract Outcome compete(RoShamBo3 it);
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo3.class, 20);
    }
}
```

输出为:

```
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
```

虽然这种方式可以工作，但是却不甚合理，如果采用 RoShamB02.java 的解决方案，那么在添加一个新的类型时，只需更少的代码，而且也更直接。

: 然而，RoShamBo3.java 还可以压缩简化一下:

```
// enums/RoShamBo4.java
// {java enums.RoShamBo4}
package enums;
public enum RoShamBo4 implements Competitor<RoShamBo4> {
    ROCK {
        @Override
        public Outcome compete(RoShamBo4 opponent) {
            return compete(SCISSORS, opponent);
        }
    },
    SCISSORS {
        @Override
        public Outcome compete(RoShamBo4 opponent) {
            return compete(PAPER, opponent);
        }
    },
```

```
    PAPER {
        @Override
        public Outcome compete(RoShamBo4 opponent) {
            return compete(ROCK, opponent);
        }
    };
    Outcome compete(RoShamBo4 loser, RoShamBo4 opponent) {
        return ((opponent == this) ? Outcome.DRAW
                : ((opponent == loser) ? Outcome.WIN
                : Outcome.LOSE));
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo4.class, 20);
    }
}
```

输出为:

```
PAPER vs. PAPER: DRAW
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. SCISSORS: WIN
ROCK vs. ROCK: DRAW
ROCK vs. SCISSORS: WIN
PAPER vs. SCISSORS: LOSE
SCISSORS vs. SCISSORS: DRAW
PAPER vs. SCISSORS: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. ROCK: WIN
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
```

其中，具有两个参数的 compete() 方法执行第二个分发，该方法执行一系列的比较，其行为类似 switch 语句。这个版本的程序更简短，不过却比较难理解，对于一个大型系统而言，难以理解的代码将导致整个系统不够健壮。

23.11.3 使用 EnumMap 进行分发

使用 EnumMap 能够实现“真正的”两路分发。EnumMap 是为 enum 专门设计的一种性能非常好的特殊 Map。由于我们的目的是摸索出两种未知的类型，所以可以用一个 EnumMap 的 EnumMap 来实现两路分发:

调试一段程序的难度是写出这段程序的难度的两倍，因此，如果你的代码尽可能清楚的话，那么你就不用费力地调试它。

```java
// enums/RoShamBo5.java
// Multiple dispatching using an EnumMap of EnumMaps
// {java enums.RoShamBo5}
package enums;
import java.util.*;
import static enums.Outcome.*;
enum RoShamBo5 implements Competitor<RoShamBo5> {
    PAPER, SCISSORS, ROCK;
    static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Outcome>>
            table = new EnumMap<>(RoShamBo5.class);
    static {
        for(RoShamBo5 it : RoShamBo5.values())
            table.put(it, new EnumMap<>(RoShamBo5.class));
        initRow(PAPER, DRAW, LOSE, WIN);
        initRow(SCISSORS, WIN, DRAW, LOSE);
        initRow(ROCK, LOSE, WIN, DRAW);
    }
    static void initRow(RoShamBo5 it,
                        Outcome vPAPER, Outcome vSCISSORS, Outcome
                            vROCK) {
        EnumMap<RoShamBo5,Outcome> row =
                RoShamBo5.table.get(it);
        row.put(RoShamBo5.PAPER, vPAPER);
        row.put(RoShamBo5.SCISSORS, vSCISSORS);
        row.put(RoShamBo5.ROCK, vROCK);
    }
    @Override
    public Outcome compete(RoShamBo5 it) {
        return table.get(this).get(it);
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo5.class, 20);
    }
}
```

输出为:

```
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
```

It’s hard enough to find an error in your code when you’re looking for it; it’s even harder when you’ve assumed your code is error-free. — Steve McConnel

```
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
```

该程序在一个 static 子句中初始化 EnumMap 对象，具体见表格似的 initRow() 方法调用。请注意 compete() 方法，您可以看到，在一行语句中发生了两次分发。

23.11.4 使用二维数组

我们还可以进一步简化实现两路分发的解决方案。我们注意到，每个 enum 实例都有一个固定的值（基于其声明的次序），并且可以通过 ordinal() 方法取得该值。因此我们可以使用二维数组，将竞争者映射到竞争结果。采用这种方式能够获得最简洁、最直接的解决方案（很可能也是最快速的，虽然我们知道 EnumMap 内部其实也是使用数组实现的）。

```
We can simplify the solution even more by noting that each enum instance has a fixed
        value (based on its declaration order) and that ordinal() produces this value. A two-
        dimensional array mapping the competitors onto the outcomes produces the smallest
        and most straightforward solution (and possibly the fastest, although remember that
        EnumMap uses an internal array):
// enums/RoShamBo6.java
// Enums using "tables" instead of multiple dispatch
// {java enums.RoShamBo6}
        package enums;
        import static enums.Outcome.*;
enum RoShamBo6 implements Competitor<RoShamBo6> {
    PAPER, SCISSORS, ROCK;
    private static Outcome[][] table = {
            { DRAW, LOSE, WIN }, // PAPER
            { WIN, DRAW, LOSE }, // SCISSORS
            { LOSE, WIN, DRAW }, // ROCK
    };
    @Override
    public Outcome compete(RoShamBo6 other) {
        return table[this.ordinal()][other.ordinal()];
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo6.class, 20);
    }
}
```

输出为:

```
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
```

table 与前一个例子中 initRow() 方法的调用次序完全相同。

与前面一个例子相比，这个程序代码虽然简短，但表达能力却更强，部分原因是其代码更易于理解与修改，而且也更直接。不过，由于它使用的是数组，所以这种方式不太“安全”。如果使用一个大型数组，可能会不小心使用了错误的尺寸，而且，如果你的测试不能覆盖所有的可能性，有些错误可能会从你眼前溜过。

事实上，以上所有的解决方案只是各种不同类型的表罢了。不过，分析各种表的表现形式，找出最适合的那一种，还是很有价值的。注意，虽然上例是最简洁的一种解决方案，但它也是相当僵硬的方案，因为它只能针对给定的常量输入产生常量输出。然而，也没有什么特别的理由阻止你用 table 来生成功能对象。对于某类问题而言，“表驱动式编码”的概念具有非常强大的功能。

23.12 本章小结

虽然枚举类型本身并不是特别复杂，但我还是将本章安排在全书比较靠后的位置，这是因为，程序员可以将 enum 与 Java 语言的其他功能结合使用，例如多态、泛型和反射。

虽然 Java 中的枚举比 C 或 C++ 中的 enum 更成熟，但它仍然是一个“小”功能，Java 没有它也已经（虽然有点笨拙）存在很多年了。而本章正好说明了一个“小”功能所能带来的价值。有时恰恰因为它，你才能够优雅而干净地解决问题。正如我在本书中一再强调的那样，优雅与清晰很重要，正是它们区别了成功的解决方案与失败的解决方案。而失败的解决方案就是因为其他人无法理解它。

关于清晰的话题，Java 1.0 对术语 enumeration 的选择正是一个不幸的反例。对于一个专门用于

从序列中选择每一个元素的对象而言，Java 竟然没有使用更通用、更普遍接受的术语 ierator 来表示它（参见集合 章节），有些语言甚至将枚举的数据类型称为 “enumerators”！Java 修正了这个错误，但是 Enumeration 接口已经无法轻易地抹去了，因此它将一直存在于旧的（甚至有些新的）代码、类库以及文档中。

这不是 bug——这是没有被证明的功能。

24 注解

注解（也被称为元数据）为我们在代码中添加信息提供了一种形式化的方式，使我们可以在稍后的某个时刻更容易的使用这些数据。

注解在一定程度上是把元数据和源代码文件结合在一起的趋势所激发的，而不是保存在外部文档。这同样是对像 C# 语言对于 Java 语言特性压力的一种回应。

注解是 Java 5 所引入的众多语言变化之一。它们提供了 Java 无法表达的但是你需要完整表述程序所需的信息。因此，注解使得我们可以以编译器验证的格式存储程序的额外信息。注解可以生成描述符文件，甚至是新的类定义，并且有助于减轻编写“样板”代码的负担。通过使用注解，你可以将元数据保存在 Java 源代码中。并拥有如下优势：简单易读的代码，编译器类型检查，使用 annotation API 为自己的注解构造处理工具。即使 Java 定义了一些类型的元数据，但是一般来说注解类型的添加和如何使用完全取决于你。

注解的语法十分简单，主要是在现有语法中添加 @ 符号。Java 5 引入了前三种定义在 **java.lang** 包中的注解：

- **@Override**：表示当前的方法定义将覆盖基类的方法。如果你不小心拼写错误，或者方法签名被错误拼写的时候，编译器就会发出错误提示。
- **@Deprecated**：如果使用该注解的元素被调用，编译器就会发出警告信息。
- **@SuppressWarnings**：关闭不当的编译器警告信息。
- **@SafeVarargs**：在 Java 7 中加入用于禁止对具有泛型 varargs 参数的方法或构造函数的调用方发出警告。
- **@FunctionalInterface**：Java 8 中加入用于表示类型声明为函数式接口

还有 5 种额外的注解类型用于创造新的注解。你将会在这一章学习它们。

每当创建涉及重复工作的类或接口时，你通常可以使用注解来自动化和简化流程。例如在 Enterprise JavaBean（EJB）中的许多额外工作就是通过注解来消除的。

注解的出现可以替代一些现有的系统，例如 XDoclet，它是一种独立的文档化工具，专门设计用来生成注解风格的文档。与之相比，注解是真正语言层级的概念，以前构造出来就享有编译器的类型检查保护。注解在源代码级别保存所有信息而不是通过注释文字，这使得代码更加整洁和便于维护。通过使用拓展的 annotation API 或稍后在本章节可以看到的外部的字节码工具类库，你会拥有对源代码及字节码强大的检查与操作能力。

24.1 基本语法

在下面的例子中，使用 @Test 对 testExecute() 进行注解。该注解本身不做任何事情，但是编译器要保证其类路径上有 @Test 注解的定义。你将在本章看到，我们通过注解创建了一个工具用于运行这个方法：

```
// annotations/Testable.java
package annotations;
import onjava.atunit.*;
public class Testable {
    public void execute() {
        System.out.println("Executing..");
    }
    @Test
    void testExecute() { execute(); }
}
```

被注解标注的方法和其他的方法没有任何区别。在这个例子中，注解 @Test 可以和任何修饰符共同用于方法，诸如 **public**、**static** 或 **void**。从语法的角度上看，注解的使用方式和修饰符的使用方式一致。

24.1.1 定义注解

如下是一个注解的定义。注解的定义看起来很像接口的定义。事实上，它们和其他 Java 接口一样，也会被编译成 class 文件。

```
// onjava/atunit/Test.java
// The @Test tag
package onjava.atunit;
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {}
```

除了 @ 符号之外，@Test 的定义看起来更像一个空接口。注解的定义也需要一些元注解（meta-annoation），比如 @Target 和 @Retention。@Target 定义你的注解可以应用在哪里（例如是方法还是字段）。@Retention 定义了注解在哪里可用，在源代码中（SOURCE），class 文件（CLASS）中或者是在运行时（RUNTIME）。

注解通常会包含一些表示特定值的元素。当分析处理注解的时候，程序或工具可以利用这些值。注解的元素看起来就像接口的方法，但是可以为其指定默认值。

不包含任何元素的注解称为标记注解（marker annotation），例如上例中的 @Test 就是标记注解。

下面是一个简单的注解，我们可以用它来追踪项目中的用例。程序员可以使用该注解来标注满足特

定用例的一个方法或者一组方法。于是，项目经理可以通过统计已经实现的用例来掌控项目的进展，而开发者在维护项目时可以轻松的找到用例用于更新，或者他们可以调试系统中业务逻辑。

```
// annotations/UseCase.java
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    int id();
    String description() default "no description";
}
```

注意 **id** 和 **description** 与方法定义类似。由于编译器会对 **id** 进行类型检查，因此将跟踪数据库与用例文档和源代码相关联是可靠的方式。**description** 元素拥有一个 **default** 值，如果在注解某个方法时没有给出 **description** 的值。则该注解的处理器会使用此元素的默认值。

在下面的类中，有三个方法被注解为用例：

```
// annotations/PasswordUtils.java
import java.util.*;
public class PasswordUtils {
    @UseCase(id = 47, description =
            "Passwords must contain at least one numeric")
    public boolean validatePassword(String passwd) {
        return (passwd.matches("\\w*\\d\\w*"));
    }
    @UseCase(id = 48)
    public String encryptPassword(String passwd) {
        return new StringBuilder(passwd)
                .reverse().toString();
    }
    @UseCase(id = 49, description =
            "New passwords can't equal previously used ones")
    public boolean checkForNewPassword(
            List<String> prevPasswords, String passwd) {
        return !prevPasswords.contains(passwd);
    }
}
```

注解的元素在使用时表现为名-值对的形式，并且需要放置在 `@UseCase` 声明之后的括号内。在 `encryptPassword()` 方法的注解中，并没有给出 **description** 的值，所以在 **@interface UseCase** 的注解处理器分析处理这个类的时候会使用该元素的默认值。

你应该能够想象到如何使用这套工具来“勾勒”出将要建造的系统，然后在建造的过程中逐渐实现系统的各项功能。

24.1.2 元注解

Java 语言中目前有 5 种标准注解（前面介绍过），以及 5 种元注解。元注解用于注解其他的注解

注解	解释
@Target	表示注解可以用于哪些地方。可能的 **ElementType** 参数包括：**CONSTRUCTOR**：构造器的声明 **FIELD**：字段声明（包括 enum 实例）**LOCAL_VARIABLE**：局部变量声明 **METHOD**：方法声明 **PACKAGE**：包声明 **PARAMETER**：参数声明 **TYPE**：类、接口（包括注解类型）或者 enum 声明
@Retention	表示注解信息保存的时长。可选的 **RetentionPolicy** 参数包括：**SOURCE**：注解将被编译器丢弃 **CLASS**：注解在 class 文件中可用，但是会被 VM 丢弃。**RUNTIME**：VM 将在运行期也保留注解，因此可以通过反射机制读取注解的信息。
@Documented	将此注解保存在 Javadoc 中
@Inherited	允许子类继承父类的注解
@Repeatable	允许一个注解可以被使用一次或者多次（Java 8）。

大多数时候，程序员定义自己的注解，并编写自己的处理器来处理他们。

24.2 编写注解处理器

如果没有用于读取注解的工具，那么注解不会比注释更有用。使用注解中一个很重要的部分就是，创建与使用注解处理器。Java 拓展了反射机制的 API 用于帮助你创造这类工具。同时他还提供了 javac 编译器钩子在编译时使用注解。

下面是一个非常简单的注解处理器，我们用它来读取被注解的 **PasswordUtils** 类，并且使用反射机制来寻找 **@UseCase** 标记。给定一组 **id** 值，然后列出在 **PasswordUtils** 中找到的用例，以及缺失的用例。

```
// annotations/UseCaseTracker.java
import java.util.*;
import java.util.stream.*;
import java.lang.reflect.*;
public class UseCaseTracker {
    public static void
    trackUseCases(List<Integer> useCases, Class<?> cl) {
```

```
        for(Method m : cl.getDeclaredMethods()) {
            UseCase uc = m.getAnnotation(UseCase.class);
            if(uc != null) {
                System.out.println("Found Use Case " +
                        uc.id() + "\n " + uc.description());
                useCases.remove(Integer.valueOf(uc.id()));
            }
        }
        useCases.forEach(i ->
                System.out.println("Missing use case " + i));
    }
    public static void main(String[] args) {
        List<Integer> useCases = IntStream.range(47, 51)
                .boxed().collect(Collectors.toList());
        trackUseCases(useCases, PasswordUtils.class);
    }
}
```

输出为:

```
Found Use Case 48
no description
Found Use Case 47
Passwords must contain at least one numeric
Found Use Case 49
New passwords can't equal previously used ones
Missing use case 50
```

这个程序用了两个反射的方法: `getDeclaredMethods()` 和 `getAnnotation()`，它们都属于 **AnnotatedElement** 接口（**Class**, **Method** 与 **Field** 类都实现了该接口）。`getAnnotation()` 方法返回指定类型的注解对象，在本例中就是 "**UseCase**"。如果被注解的方法上没有该类型的注解，返回值就为 **null**。我们通过调用 `id()` 和 `description()` 方法来提取元素值。注意 `encryptPassword()` 方法在注解的时候没有指定 **description** 的值，因此处理器在处理它对应的注解时，通过 `description()` 取得的是默认值 "no description"。

24.2.1 注解元素

在 **UseCase.java** 中定义的 **@UseCase** 的标签包含 int 元素 **id** 和 String 元素 **description**。注解元素可用的类型如下所示:

- 所有基本类型（int、float、boolean 等）
- String
- Class
- enum
- Annotation

- 以上类型的数组

如果你使用了其他类型，编译器就会报错。注意，也不允许使用任何包装类型，但是由于自动装箱的存在，这不算是什么限制。注解也可以作为元素的类型。稍后你会看到，注解嵌套是一个非常有用的技巧。

24.2.2 默认值限制

编译器对于元素的默认值有些过于挑剔。首先，元素不能有不确定的值。也就是说，元素要么有默认值，要么就在使用注解时提供元素的值。

这里有另外一个限制：任何非基本类型的元素，无论是在源代码声明时还是在注解接口中定义默认值时，都不能使用 null 作为其值。这个限制使得处理器很难表现一个元素的存在或者缺失的状态，因为在每个注解的声明中，所有的元素都存在，并且具有相应的值。为了绕开这个约束，可以自定义一些特殊的值，比如空字符串或者负数用于表达某个元素不存在。

```
// annotations/SimulatingNull.java
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulatingNull {
    int id() default -1;
    String description() default "";
}
```

这是一个在定义注解的习惯用法。

24.2.3 生成外部文件

当有些框架需要一些额外的信息才能与你的源代码协同工作，这种情况下注解就会变得十分有用。像 Enterprise JavaBeans (EJB3 之前) 这样的技术，每一个 Bean 都需要需要大量的接口和部署描述文件，而这些就是“样板”文件。Web Service，自定义标签库以及对象/关系映射工具（例如 Toplink 和 Hibernate）通常都需要 XML 描述文件，而这些文件脱离于代码之外。除了定义 Java 类，程序员还必须忍受沉闷，重复的提供某些信息，例如类名和包名等已经在原始类中已经提供的信息。每当你使用外部描述文件时，他就拥有了一个类的两个独立信息源，这经常导致代码的同步问题。同时这也要求了为项目工作的程序员在知道如何编写 Java 程序的同时，也必须知道如何编辑描述文件。

假设你想提供一些基本的对象/关系映射功能，能够自动生成数据库表。你可以使用 XML 描述文件来指明类的名字、每个成员以及数据库映射的相关信息。但是，通过使用注解，你可以把所有信息都保存在 **JavaBean** 源文件中。为此你需要一些用于定义数据库表名称、数据库列以及将 SQL 类型映射到属性的注解。

以下是一个注解的定义，它告诉注解处理器应该创建一个数据库表：

写没有错误的程序有两种方式；但第三种才有效。

```java
// annotations/database/DBTable.java
package annotations.database;
import java.lang.annotation.*;
@Target(ElementType.TYPE) // Applies to classes only
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
    String name() default "";
}
```

在 `@Target` 注解中指定的每一个 **ElementType** 就是一个约束，它告诉编译器，这个自定义的注解只能用于指定的类型。你可以指定 **enum ElementType** 中的一个值，或者以逗号分割的形式指定多个值。如果想要将注解应用于所有的 **ElementType**，那么可以省去 `@Target` 注解，但是这并不常见。

注意 **@DBTable** 中有一个 `name()` 元素，该注解通过这个元素为处理器创建数据库时提供表的名字。

如下是修饰字段的注解：

```java
// annotations/database/Constraints.java
package annotations.database;
import java.lang.annotation.*;
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
    boolean primaryKey() default false;
    boolean allowNull() default true;
    boolean unique() default false;
}
```

```java
// annotations/database/SQLString.java
package annotations.database;
import java.lang.annotation.*;
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
    int value() default 0;
    String name() default "";
    Constraints constraints() default @Constraints;
}
```

```java
// annotations/database/SQLInteger.java
package annotations.database;
import java.lang.annotation.*;
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
    String name() default "";
    Constraints constraints() default @Constraints;
```

```
}
```

@Constraints 注解允许处理器提供数据库表的元数据。**@Constraints** 代表了数据库通常提供的约束的一小部分，但是它所要表达的思想已经很清楚了。`primaryKey()`，`allowNull()` 和 `unique()` 元素明显的提供了默认值，从而使得在大多数情况下，该注解的使用者不需要输入太多东西。

另外两个 **@interface** 定义的是 SQL 类型。如果希望这个框架更有价值的话，我们应该为每个 SQL 类型都定义相应的注解。不过为为示例，两个元素足够了。

这些 SQL 类型具有 `name()` 元素和 `constraints()` 元素。后者利用了嵌套注解的功能，将数据库列的类型约束信息嵌入其中。注意 `constraints()` 元素的默认值是 **@Constraints**。由于在 **@Constraints** 注解类型之后，没有在括号中指明 **@Constraints** 元素的值，因此，**constraints()** 的默认值为所有元素都为默认值的 **@Constraints** 注解。如果要使得嵌入的 **@Constraints** 注解中的 `unique()` 元素为 true，并作为 `constraints()` 元素的默认值，你可以像如下定义：

```java
// annotations/database/Uniqueness.java
// Sample of nested annotations
package annotations.database;
public @interface Uniqueness {
    Constraints constraints()
            default @Constraints(unique = true);
}
```

下面是一个简单的，使用了如上注解的类：

```java
// annotations/database/Member.java
package annotations.database;
@DBTable(name = "MEMBER")
public class Member {
    @SQLString(30) String firstName;
    @SQLString(50) String lastName;
    @SQLInteger Integer age;
    @SQLString(value = 30,
            constraints = @Constraints(primaryKey = true))
    String reference;
    static int memberCount;
    public String getReference() { return reference; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    @Override
    public String toString() { return reference; }
    public Integer getAge() { return age; }
}
```

类注解 **@DBTable** 注解给定了元素值 MEMBER，它将会作为标的名字。类的属性 **firstName** 和 **lastName** 都被注解为 **@SQLString** 类型并且给了默认元素值分别为 30 和 50。这些注解都有两个有趣的地方：首先，他们都使用了嵌入的 **@Constraints** 注解的默认值；其次，它们都是用了快捷方

小心上面代码中的 bug；我只证明了它是正确的，还没有尝试过。

式特性。如果你在注解中定义了名为 **value** 的元素，并且在使用该注解时，**value** 为唯一一个需要赋值的元素，你就不需要使用名—值对的语法，你只需要在括号中给出 **value** 元素的值即可。这可以应用于任何合法类型的元素。这也限制了你必须将元素命名为 **value**，不过在上面的例子中，这样的注解语句也更易于理解：

```
@SQLString(30)
```

处理器将在创建表的时候使用该值设置 SQL 列的大小。

默认值的语法虽然很灵巧，但是它很快就变的复杂起来。以 **reference** 字段的注解为例，上面拥有 **@SQLString** 注解，但是这个字段也将成为表的主键，因此在嵌入的 **@Constraint** 注解中设定 **primaryKey** 元素的值。这时事情就变的复杂了。你不得不为这个嵌入的注解使用很长的键—值对的形式，来指定元素名称和 **@interface** 的名称。同时，由于有特殊命名的 **value** 也不是唯一需要赋值的元素，因此不能再使用快捷方式特性。如你所见，最终结果不算清晰易懂。

24.2.4 替代方案

可以使用多种不同的方式来定义自己的注解用于上述任务。例如，你可以使用一个单一的注解类 **@TableColumn**，它拥有一个 **enum** 元素，元素值定义了 **STRING**，**INTEGER**，**FLOAT** 等类型。这消除了每个 SQL 类型都需要定义一个 **@interface** 的负担，不过也使得用额外信息修饰 SQL 类型变的不可能，这些额外的信息例如长度或精度等，都可能是非常有用的。

你也可以使用一个 **String** 类型的元素来描述实际的 SQL 类型，比如 “VARCHAR(30)” 或者 “INTEGER”。这使得你可以修饰 SQL 类型，但是这也将 Java 类型到 SQL 类型的映射绑在了一起，这不是一个好的设计。你并不想在数据库更改之后重新编译你的代码；如果我们只需要告诉注解处理器，我们正在使用的是什么“口味（favor）”的 SQL，然后注解助力器来为我们处理 SQL 类型的细节，那将是一个优雅的设计。

第三种可行的方案是一起使用两个注解，**@Constraints** 和相应的 SQL 类型（例如，**@SQLInteger**）去注解同一个字段。这可能会让代码有些混乱，但是编译器允许你对同一个目标使用多个注解。在 Java 8，在使用多个注解的时候，你可以重复使用同一个注解。

24.2.5 注解不支持继承

你不能使用 **extends** 关键字来继承 **@interfaces**。这真是一个遗憾，如果可以定义 **@TableColumn** 注解（参考前面的建议），同时嵌套一个 **@SQLType** 类型的注解，将成为一个优雅的设计。按照这种方式，你可以通过继承 **@SQLType** 来创造各种 SQL 类型。例如 **@SQLInteger** 和 **@SQLString**。如果支持继承，就会大大减少打字的工作量并且使得语法更整洁。在 Java 的未来版本中，似乎没有任何关于让注解支持继承的提案，所以在当前情况下，上例中的解决方案可能已经是最佳方案了。

24.2.6 实现处理器

下面是一个注解处理器的例子，他将读取一个类文件，检查上面的数据库注解，并生成用于创建数据库的 SQL 命令:

```
// annotations/database/TableCreator.java
// Reflection-based annotation processor
// {java annotations.database.TableCreator
// annotations.database.Member}
package annotations.database;

import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.List;

public class TableCreator {
    public static void
    main(String[] args) throws Exception {
        if (args.length < 1) {
            System.out.println(
                    "arguments: annotated classes");
            System.exit(0);
        }
        for (String className : args) {
            Class<?> cl = Class.forName(className);
            DBTable dbTable = cl.getAnnotation(DBTable.class);
            if (dbTable == null) {
                System.out.println(
                        "No DBTable annotations in class " +
                                className);
                continue;
            }
            String tableName = dbTable.name();
            // If the name is empty, use the Class name:
            if (tableName.length() < 1)
                tableName = cl.getName().toUpperCase();
            List<String> columnDefs = new ArrayList<>();
            for (Field field : cl.getDeclaredFields()) {
                String columnName = null;
                Annotation[] anns =
                        field.getDeclaredAnnotations();
                if (anns.length < 1)
                    continue; // Not a db table column
                if (anns[0] instanceof SQLInteger) {
                    SQLInteger sInt = (SQLInteger) anns[0];
                    // Use field name if name not specified
                    if (sInt.name().length() < 1)
                        columnName = field.getName().toUpperCase();
                    else
```

```
                        columnName = sInt.name();
                    columnDefs.add(columnName + " INT" +
                            getConstraints(sInt.constraints()));
                }
                if (anns[0] instanceof SQLString) {
                    SQLString sString = (SQLString) anns[0];
                    // Use field name if name not specified.
                    if (sString.name().length() < 1)
                        columnName = field.getName().toUpperCase();
                    else
                        columnName = sString.name();
                    columnDefs.add(columnName + " VARCHAR(" +
                            sString.value() + ")" +
                            getConstraints(sString.constraints()));
                }
                StringBuilder createCommand = new StringBuilder(
                        "CREATE TABLE " + tableName + "(");
                for (String columnDef : columnDefs)
                    createCommand.append(
                            "\n " + columnDef + ",");
                // Remove trailing comma
                String tableCreate = createCommand.substring(
                        0, createCommand.length() - 1) + ");";
                System.out.println("Table Creation SQL for " +
                        className + " is:\n" + tableCreate);
            }
        }
    }

    private static String getConstraints(Constraints con) {
        String constraints = "";
        if (!con.allowNull())
            constraints += " NOT NULL";
        if (con.primaryKey())
            constraints += " PRIMARY KEY";
        if (con.unique())
            constraints += " UNIQUE";
        return constraints;
    }
}
```

输出为:

```
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30));
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50));
Table Creation SQL for annotations.database.Member is:
```

```
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT);
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT,
    REFERENCE VARCHAR(30) PRIMARY KEY);
```

主方法会循环处理命令行传入的每一个类名。每一个类都是用 `forName()` 方法进行加载，并使用 `getAnnotation(DBTable.class)` 来检查该类是否带有 **@DBTable** 注解。如果存在，将表名存储起来。然后读取这个类的所有字段，并使用 `getDeclaredAnnotations()` 进行检查。这个方法返回一个包含特定字段上所有注解的数组。然后使用 **instanceof** 操作符判断这些注解是否是 **@SQLInteger** 或者 **@SQLString** 类型。如果是的话，在对应的处理块中将构造出相应的数据库列的字符串片段。注意，由于注解没有继承机制，如果要获取近似多态的行为，使用 `getDeclaredAnnotations()` 似乎是唯一的方式。

嵌套的 **@Constraint** 注解被传递给 `getConstraints()`方法，并用它来构造一个包含 SQL 约束的 String 对象。

需要提醒的是，上面演示的技巧对于真实的对象/映射关系而言，是十分幼稚的。使用 **@DBTable** 的注解来获取表的名称，这使得如果要修改表的名字，则迫使你重新编译 Java 代码。这种效果并不理想。现在已经有了很多可用的框架，用于将对象映射到数据库中，并且越来越多的框架开始使用注解了。

24.3 使用 javac 处理注解

通过 **javac**，你可以通过创建编译时（compile-time）注解处理器在 Java 源文件上使用注解，而不是编译之后的 class 文件。但是这里有一个重大限制：你不能通过处理器来改变源代码。唯一影响输出的方式就是创建新的文件。

如果你的注解处理器创建了新的源文件，在新一轮处理中注解会检查源文件本身。工具在检测一轮之后持续循环，直到不再有新的源文件产生。然后它编译所有的源文件。

每一个你编写的注解都需要处理器，但是 **javac** 可以非常容易的将多个注解处理器合并在一起。你可以指定多个需要处理的类，并且你可以添加监听器用于监听注解处理完成后接到通知。

本节中的示例将帮助你开始学习，但如果你必须深入学习，请做好反复学习，大量访问 Google 和 StackOverflow 的准备。

软件在发布前要经过 beta 测试。beta 即拉丁语“仍然无法正常工作”的意思。

24.3.1 最简单的处理器

让我们开始定义我们能想到的最简单的处理器，只是为了编译和测试。如下是注解的定义：

```
// annotations/simplest/Simple.java
// A bare-bones annotation
package annotations.simplest;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;
@Retention(RetentionPolicy.SOURCE)
@Target({ElementType.TYPE, ElementType.METHOD,
        ElementType.CONSTRUCTOR,
        ElementType.ANNOTATION_TYPE,
        ElementType.PACKAGE, ElementType.FIELD,
        ElementType.LOCAL_VARIABLE})
public @interface Simple {
    String value() default "-default-";
}
```

@Retention 的参数现在为 **SOURCE**，这意味着注解不会再存留在编译后的代码。这在编译时处理注解是没有必要的，它只是指出，在这里，**javac** 是唯一有机会处理注解的代理。

@Target 声明了几乎所有的目标类型（除了 **PACKAGE**），同样是为了演示。下面是一个测试示例。

```
// annotations/simplest/SimpleTest.java
// Test the "Simple" annotation
// {java annotations.simplest.SimpleTest}
package annotations.simplest;
@Simple
public class SimpleTest {
    @Simple
    int i;
    @Simple
    public SimpleTest() {}
    @Simple
    public void foo() {
        System.out.println("SimpleTest.foo()");
    }
    @Simple
    public void bar(String s, int i, float f) {
        System.out.println("SimpleTest.bar()");
    }
    @Simple
    public static void main(String[] args) {
        @Simple
        SimpleTest st = new SimpleTest();
        st.foo();
```

```
    }
}
```

输出为:

```
SimpleTest.foo()
```

在这里我们使用 **@Simple** 注解了所有 **@Target** 声明允许的地方。

SimpleTest.java 只需要 **Simple.java** 就可以编译成功。当我们编译的时候什么都没有发生。

javac 允许 **@Simple** 注解（只要它存在）在我们创建处理器并将其 hook 到编译器之前，不做任何事情。

如下是一个十分简单的处理器，其所作的事情就是把注解相关的信息打印出来:

```
// annotations/simplest/SimpleProcessor.java
// A bare-bones annotation processor
package annotations.simplest;
import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import java.util.*;
@SupportedAnnotationTypes(
        "annotations.simplest.Simple")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SimpleProcessor
        extends AbstractProcessor {
    @Override
    public boolean process(
            Set<? extends TypeElement> annotations,
            RoundEnvironment env) {
        for(TypeElement t : annotations)
            System.out.println(t);
        for(Element el :
                env.getElementsAnnotatedWith(Simple.class))
            display(el);
        return false;
    }
    private void display(Element el) {
        System.out.println("==== " + el + " ====");
        System.out.println(el.getKind() +
                " : " + el.getModifiers() +
                " : " + el.getSimpleName() +
                " : " + el.asType());
        if(el.getKind().equals(ElementKind.CLASS)) {
            TypeElement te = (TypeElement)el;
            System.out.println(te.getQualifiedName());
            System.out.println(te.getSuperclass());
            System.out.println(te.getEnclosedElements());
        }
        if(el.getKind().equals(ElementKind.METHOD)) {
```

```
            ExecutableElement ex = (ExecutableElement)el;
            System.out.print(ex.getReturnType() + " ");
            System.out.print(ex.getSimpleName() + "(");
            System.out.println(ex.getParameters() + ")");
        }
    }
}
```

（旧的，失效的）**apt** 版本的处理器需要额外的方法来确定支持哪些注解以及支持的 Java 版本。不过，你现在可以简单的使用 **@SupportedAnnotationTypes** 和 **@SupportedSourceVersion** 注解（这是一个很好的示例关于注解如何简化你的代码）。

你唯一需要实现的方法就是 `process()`，这里是所有行为发生的地方。第一个参数告诉你哪个注解是存在的，第二个参数保留了剩余信息。我们所做的事情只是打印了注解（这里只存在一个），可以看 **TypeElement** 文档中的其他行为。通过使用 `process()` 的第二个操作，我们循环所有被 **@Simple** 注解的元素，并且针对每一个元素调用我们的 `display()` 方法。所有 **Element** 展示了本身的基本信息；例如，`getModifiers()` 告诉你它是否为 **public** 和 **static** 的。

Element 只能执行那些编译器解析的所有基本对象共有的操作，而类和方法之类的东西有额外的信息需要提取。所以（如果你阅读了正确的文档，但是我没有在任何文档中找到——我不得不通过 StackOverflow 寻找线索）你检查它是哪种 **ElementKind**，然后将其向下转换为更具体的元素类型，注入针对 CLASS 的 TypeElement 和针对 METHOD 的 ExecutableElement。此时，可以为这些元素调用其他方法。

动态向下转型（在编译期不进行检查）并不像是 Java 的做事方式，这非常不直观这也是为什么我从未想过要这样做事。相反，我花了好几天的时间，试图发现你应该如何访问这些信息，而这些信息至少在某种程度上是用不起作用的恰当方法简单明了的。我还没有遇到任何东西说上面是规范的形式，但在我看来是。

如果只是通过平常的方式来编译 **SimpleTest.java**，你不会得到任何结果。为了得到注解输出，你必须增加一个 **processor** 标志并且连接注解处理器类

```
javac -processor annotations.simplest.SimpleProcessor SimpleTest.java
```

现在编译器有了输出

```
annotations.simplest.Simple
==== annotations.simplest.SimpleTest ====
CLASS : [public] : SimpleTest : annotations.simplest.SimpleTest
annotations.simplest.SimpleTest
java.lang.Object
i,SimpleTest(),foo(),bar(java.lang.String,int,float),main(java.lang.
    String[])
==== i ====
FIELD : [] : i : int
==== SimpleTest() ====
CONSTRUCTOR : [public] : <init> : ()void
```

```
==== foo() ====
METHOD : [public] : foo : ()void
void foo()
==== bar(java.lang.String,int,float) ====
METHOD : [public] : bar : (java.lang.String,int,float)void
void bar(s,i,f)
==== main(java.lang.String[]) ====
METHOD : [public, static] : main : (java.lang.String[])void
void main(args)
```

这给了你一些可以发现的东西，包括参数名和类型、返回值等。

24.3.2 更复杂的处理器

当你创建用于 javac 注解处理器时，你不能使用 Java 的反射特性，因为你处理的是源代码，而并非是编译后的 class 文件。各种 mirror[^3] 解决这个问题的方法是，通过允许你在未编译的源代码中查看方法、字段和类型。

如下是一个用于提取类中方法的注解，所以它可以被抽取成为一个接口：

```java
// annotations/ifx/ExtractInterface.java
// javac-based annotation processing
package annotations.ifx;
import java.lang.annotation.*;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ExtractInterface {
    String interfaceName() default "-!!-";
}
```

RetentionPolicy 的值为 **SOURCE**，这是为了在提取类中的接口之后不再将注解信息保留在 class 文件中。接下来的测试类提供了一些公用方法，这些方法可以成为接口的一部分：

```java
// annotations/ifx/Multiplier.java
// javac-based annotation processing
// {java annotations.ifx.Multiplier}
package annotations.ifx;
@ExtractInterface(interfaceName="IMultiplier")
public class Multiplier {
    public boolean flag = false;
    private int n = 0;
    public int multiply(int x, int y) {
        int total = 0;
        for(int i = 0; i < x; i++)
            total = add(total, y);
        return total;
    }
    public int fortySeven() { return 47; }
    private int add(int x, int y) {
```

最便宜，最快和最可靠的组件是不存在的。

```
        return x + y;
    }
    public double timesTen(double arg) {
        return arg * 10;
    }
    public static void main(String[] args) {
        Multiplier m = new Multiplier();
        System.out.println(
                "11 * 16 = " + m.multiply(11, 16));
    }
}
```

输出为:

```
11 * 16 = 176
```

Multiplier 类（只能处理正整数）拥有一个 `multiply()` 方法，这个方法会多次调用私有方法 `add()` 来模拟乘法操作。`add()` 是私有方法，因此不能成为接口的一部分。其他的方法提供了语法多样性。注解被赋予 **IMultiplier** 的 **InterfaceName** 作为要创建的接口的名称。

这里有一个编译时处理器用于提取有趣的方法，并创建一个新的 interface 源代码文件（这个源文件将会在下一轮中被自动编译）:

```
// annotations/ifx/IfaceExtractorProcessor.java
// javac-based annotation processing
package annotations.ifx;
import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import javax.lang.model.util.*;
import java.util.*;
import java.util.stream.*;
import java.io.*;
@SupportedAnnotationTypes(
        "annotations.ifx.ExtractInterface")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class IfaceExtractorProcessor
        extends AbstractProcessor {
    private ArrayList<Element>
            interfaceMethods = new ArrayList<>();
    Elements elementUtils;
    private ProcessingEnvironment processingEnv;
    @Override
    public void init(
            ProcessingEnvironment processingEnv) {
        this.processingEnv = processingEnv;
        elementUtils = processingEnv.getElementUtils();
    }
    @Override
    public boolean process(
```

```
            Set<? extends TypeElement> annotations,
            RoundEnvironment env) {
        for(Element elem:env.getElementsAnnotatedWith(
                ExtractInterface.class)) {
            String interfaceName = elem.getAnnotation(
                    ExtractInterface.class).interfaceName();
            for(Element enclosed :
                    elem.getEnclosedElements()) {
                if(enclosed.getKind()
                        .equals(ElementKind.METHOD) &&
                        enclosed.getModifiers()
                                .contains(Modifier.PUBLIC) &&
                        !enclosed.getModifiers()
                                .contains(Modifier.STATIC)) {
                    interfaceMethods.add(enclosed);
                }
            }
            if(interfaceMethods.size() > 0)
                writeInterfaceFile(interfaceName);
        }
        return false;
    }
    private void
    writeInterfaceFile(String interfaceName) {
        try(
                Writer writer = processingEnv.getFiler()
                        .createSourceFile(interfaceName)
                        .openWriter()
        ) {
            String packageName = elementUtils
                    .getPackageOf(interfaceMethods
                            .get(0)).toString();
            writer.write(
                    "package " + packageName + ";\n");
            writer.write("public interface " +
                    interfaceName + " {\n");
            for(Element elem : interfaceMethods) {
                ExecutableElement method =
                        (ExecutableElement)elem;
                String signature = " public ";
                signature += method.getReturnType() + " ";
                signature += method.getSimpleName();
                signature += createArgList(
                        method.getParameters());
                System.out.println(signature);
                writer.write(signature + ";\n");
            }
            writer.write("}");
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
```

```
    }
    private String createArgList(
            List<? extends VariableElement> parameters) {
        String args = parameters.stream()
                .map(p -> p.asType() + " " + p.getSimpleName())
                .collect(Collectors.joining(", "));
        return "(" + args + ")";
    }
}
```

Elements 对象实例 **elementUtils** 是一组静态方法的工具；我们用它来寻找 **writeInterfaceFile()** 中含有的包名。

`getEnclosedElements()`方法会通过指定的元素生成所有的“闭包”元素。在这里，这个类闭包了它的所有元素。通过使用 `getKind()` 我们会找到所有的 **public** 和 **static** 方法，并将其添加到 **interfaceMethods** 列表中。接下来 `writeInterfaceFile()` 使用 **interfaceMethods** 列表里面的值生成新的接口定义。注意，在 `writeInterfaceFile()` 使用了向下转型到 **ExecutableElement**，这使得我们可以获取所有的方法信息。**createArgList()** 是一个帮助方法，用于生成参数列表。

Filer 是 `getFiler()` 生成的，并且是 **PrintWriter** 的一种实例，可以用于创建新文件。我们使用 **Filer** 对象，而不是原生的 **PrintWriter** 原因是，这个对象可以运行 **javac** 追踪你创建的新文件，这使得它可以在新一轮中检查新文件中的注解并编译文件。

如下是一个命令行，可以在编译的时候使用处理器：

```
javac -processor annotations.ifx.IfaceExtractorProcessor Multiplier.
    java
```

新生成的 **IMultiplier.java** 的文件，正如你通过查看上面处理器的 `println()` 语句所猜测的那样，如下所示：

```
package annotations.ifx;
public interface IMultiplier {
    public int multiply(int x, int y);
    public int fortySeven();
    public double timesTen(double arg);
}
```

这个类同样会被 **javac** 编译（在某一轮中），所以你会在同一个目录中看到 **IMultiplier.class** 文件。

24.4 基于注解的单元测试

单元测试是对类中每个方法提供一个或者多个测试的一种事件，其目的是为了有规律的测试一个类中每个部分是否具备正确的行为。在 Java 中，最著名的单元测试工具就是 **JUnit**。**JUnit** 4 版本已经包含了注解。在注解版本之前的 JUnit 一个最主要的问题是，为了启动和运行 **JUnit** 测试，有大量的“仪

式”需要标注。这种负担已经减轻了一些，**但是**注解使得测试更接近“可以工作的最简单的测试系统”。

在注解版本之前的 JUnit，你必须创建一个单独的文件来保存单元测试。通过注解，我们可以将单元测试集成在需要被测试的类中，从而将单元测试的时间和麻烦降到了最低。这种方式有额外的好处，就是使得测试私有方法和公有方法变的一样容易。

这个基于注解的测试框架叫做 **@Unit**。其最基本的测试形式，可能也是你使用的最多的一个注解是 **@Test**，我们使用 **@Test** 来标记测试方法。测试方法不带参数，并返回 **boolean** 结果来说明测试方法成功或者失败。你可以任意命名它的测试方法。同时 **@Unit** 测试方法可以是任意你喜欢的访问修饰方法，包括 **private**。

要使用 **@Unit**，你必须导入 **onjava.atunit** 包，并且使用 **@Unit** 的测试标记为合适的方法和字段打上标签（在接下来的例子中你会学到），然后让你的构建系统对编译后的类运行 **@Unit**，下面是一个简单的例子：

```
// annotations/AtUnitExample1.java
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/AtUnitExample1.class}
package annotations;
import onjava.atunit.*;
import onjava.*;
public class AtUnitExample1 {
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
    @Test
    boolean methodOneTest() {
        return methodOne().equals("This is methodOne");
    }
    @Test
    boolean m2() { return methodTwo() == 2; }
    @Test
    private boolean m3() { return true; }
    // Shows output for failure:
    @Test
    boolean failureTest() { return false; }
    @Test
    boolean anotherDisappointment() {
        return false;
    }
}
```

输出为：

```
annotations.AtUnitExample1
```

```
. m3
. methodOneTest
. m2 This is methodTwo
. failureTest (failed)
. anotherDisappointment (failed)
(5 tests)
>>> 2 FAILURES <<<
annotations.AtUnitExample1: failureTest
annotations.AtUnitExample1: anotherDisappointment
```

使用 **@Unit** 进行测试的类必须定义在某个包中（即必须包括 **package** 声明）。

@Test 注解被置于 `methodOneTest()`、`m2()`、`m3()`、`failureTest()` 以及 `anotherDisappointment()` 方法之前，它们告诉 **@Unit** 方法作为单元测试来运行。同时 **@Test** 确保这些方法没有任何参数并且返回值为 **boolean** 或者 **void**。当你填写单元测试时，唯一需要做的就是决定测试是成功还是失败，（对于返回值为 **boolean** 的方法）应该返回 **ture** 还是 **false**。

如果你熟悉 **JUnit**，你还将注意到 **@Unit** 输出的信息更多。你会看到现在正在运行的测试的输出更有用，最后它会告诉你导致失败的类和测试。

你并非必须将测试方法嵌入到原来的类中，有时候这种事情根本做不到。要生产一个非嵌入式的测试，最简单的方式就是继承：

```java
// annotations/AUExternalTest.java
// Creating non-embedded tests
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/AUExternalTest.class}
package annotations;
import onjava.atunit.*;
import onjava.*;
public class AUExternalTest extends AtUnitExample1 {
    @Test
    boolean _MethodOne() {
        return methodOne().equals("This is methodOne");
    }
    @Test
    boolean _MethodTwo() {
        return methodTwo() == 2;
    }
}
```

输出为：

```
annotations.AUExternalTest
. tMethodOne
. tMethodTwo This is methodTwo
OK (2 tests)
```

这个示例还表现出灵活命名的价值。在这里，**@Test** 方法被命名为下划线前缀加上要测试的方法名称（我并不认为这是一种理想的命名形式，这只是表现一种可能性罢了）。

你也可以使用组合来创建非嵌入式的测试:

```
// annotations/AUComposition.java
// Creating non-embedded tests
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/AUComposition.class}
package annotations;
import onjava.atunit.*;
import onjava.*;
public class AUComposition {
    AtUnitExample1 testObject = new AtUnitExample1();
    @Test
    boolean tMethodOne() {
        return testObject.methodOne()
                .equals("This is methodOne");
    }
    @Test
    boolean tMethodTwo() {
        return testObject.methodTwo() == 2;
    }
}
```

输出为:

```
annotations.AUComposition
. tMethodTwo This is methodTwo
. tMethodOne
OK (2 tests)
```

因为在每一个测试里面都会创建 **AUComposition** 对象，所以创建新的成员变量 **testObject** 用于以后的每一个测试方法。

因为 **@Unit** 中没有 **JUnit** 中特殊的 **assert** 方法，不过另一种形式的 **@Test** 方法仍然允许返回值为 **void**（如果你还想使用 **true** 或者 **false** 的话，也可以使用 **boolean** 作为方法返回值类型）。为了表示测试成功，可以使用 Java 的 **assert** 语句。Java 断言机制需要你在 java 命令行行加上 **-ea** 标志来开启，但是 **@Unit** 已经自动开启了该功能。要表示测试失败的话，你甚至可以使用异常。**@Unit** 的设计目标之一就是尽可能减少添加额外的语法，而 Java 的 **assert** 和异常对于报告错误而言，即已经足够了。一个失败的 **assert** 或者从方法从抛出的异常都被视为测试失败，但是 **@Unit** 不会在这个失败的测试上卡住，它会继续运行，直到所有测试完毕，下面是一个示例程序:

```
// annotations/AtUnitExample2.java
// Assertions and exceptions can be used in @Tests
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/AtUnitExample2.class}
package annotations;
import java.io.*;
import onjava.atunit.*;
import onjava.*;
```

```java
public class AtUnitExample2 {
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
    @Test
    void assertExample() {
        assert methodOne().equals("This is methodOne");
    }
    @Test
    void assertFailureExample() {
        assert 1 == 2: "What a surprise!";
    }
    @Test
    void exceptionExample() throws IOException {
        try(FileInputStream fis =
                    new FileInputStream("nofile.txt")) {} // Throws
    }
    @Test
    boolean assertAndReturn() {
        // Assertion with message:
        assert methodTwo() == 2: "methodTwo must equal 2";
        return methodOne().equals("This is methodOne");
    }
}
```

输出为:

```
annotations.AtUnitExample2
. exceptionExample java.io.FileNotFoundException:
nofile.txt (The system cannot find the file specified)
(failed)
. assertExample
. assertAndReturn This is methodTwo
. assertFailureExample java.lang.AssertionError: What
a surprise!
(failed)
(4 tests)
>>> 2 FAILURES <<<
annotations.AtUnitExample2: exceptionExample
annotations.AtUnitExample2: assertFailureExample
```

如下是一个使用非嵌入式测试的例子，并且使用了断言，它将会对 **java.util.HashSet** 进行一些简单的测试:

```java
// annotations/HashSetTest.java
// {java onjava.atunit.AtUnit
```

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves. — Alan Kay

```
// build/classes/main/annotations/HashSetTest.class}
package annotations;
import java.util.*;
import onjava.atunit.*;
import onjava.*;
public class HashSetTest {
    HashSet<String> testObject = new HashSet<>();
    @Test
    void initialization() {
        assert testObject.isEmpty();
    }
    @Test
    void _Contains() {
        testObject.add("one");
        assert testObject.contains("one");
    }
    @Test
    void _Remove() {
        testObject.add("one");
        testObject.remove("one");
        assert testObject.isEmpty();
    }
}
```

采用继承的方式可能会更简单，也没有一些其他的约束。

对每一个单元测试而言，**@Unit** 都会使用默认的无参构造器，为该测试类所属的类创建出一个新的实例。并在此新创建的对象上运行测试，然后丢弃该对象，以免对其他测试产生副作用。如此创建对象导致我们依赖于类的默认构造器。如果你的类没有默认构造器，或者对象需要复杂的构造过程，那么你可以创建一个 **static** 方法专门负责构造对象，然后使用 **@TestObjectCreate** 注解标记该方法，例子如下：

```
// annotations/AtUnitExample3.java
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/AtUnitExample3.class}
package annotations;
import onjava.atunit.*;
import onjava.*;
public class AtUnitExample3 {
    private int n;
    public AtUnitExample3(int n) { this.n = n; }
    public int getN() { return n; }
    public String methodOne() {
        return "This is methodOne";
    }
    public int methodTwo() {
        System.out.println("This is methodTwo");
        return 2;
    }
```

现在的大多数软件非常像埃及金字塔，在彼此之间堆建了成千上万的砖块，缺乏结构完整性，只是靠蛮力和成千上万的奴隶完成。

```
    @TestObjectCreate
    static AtUnitExample3 create() {
        return new AtUnitExample3(47);
    }
    @Test
    boolean initialization() { return n == 47; }
    @Test
    boolean methodOneTest() {
        return methodOne().equals("This is methodOne");
    }
    @Test
    boolean m2() { return methodTwo() == 2; }
}
```

输出为:

```
annotations.AtUnitExample3
. initialization
. m2 This is methodTwo
. methodOneTest
OK (3 tests)
```

@TestObjectCreate 修饰的方法必须声明为 **static**，且必须返回一个你正在测试的类型对象，这一切都由 **@Unit** 负责确保成立。

有的时候，你需要向单元测试中增加一些字段。这时候可以使用 **@TestProperty** 注解，由它注解的字段表示只在单元测试中使用（因此，在你将产品发布给客户之前，他们应该被删除）。在下面的例子中，一个 **String** 通过 `String.split()` 方法进行分割，从其中读取一个值，这个值将会被生成测试对象:

```
// annotations/AtUnitExample4.java
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/AtUnitExample4.class}
// {VisuallyInspectOutput}
package annotations;
import java.util.*;
import onjava.atunit.*;
import onjava.*;
public class AtUnitExample4 {
    static String theory = "All brontosauruses " +
            "are thin at one end, much MUCH thicker in the " +
            "middle, and then thin again at the far end.";
    private String word;
    private Random rand = new Random(); // Time-based seed
    public AtUnitExample4(String word) {
        this.word = word;
    }
    public String getWord() { return word; }
    public String scrambleWord() {
        List<Character> chars = Arrays.asList(
```

```
                ConvertTo.boxed(word.toCharArray()));
        Collections.shuffle(chars, rand);
        StringBuilder result = new StringBuilder();
        for(char ch : chars)
            result.append(ch);
        return result.toString();
    }
    @TestProperty
    static List<String> input =
            Arrays.asList(theory.split(" "));
    @TestProperty
    static Iterator<String> words = input.iterator();
    @TestObjectCreate
    static AtUnitExample4 create() {
        if(words.hasNext())
            return new AtUnitExample4(words.next());
        else
            return null;
    }
    @Test
    boolean words() {
        System.out.println("'" + getWord() + "'");
        return getWord().equals("are");
    }
    @Test
    boolean scramble1() {
// Use specific seed to get verifiable results:
        rand = new Random(47);
        System.out.println("'" + getWord() + "'");
        String scrambled = scrambleWord();
        System.out.println(scrambled);
        return scrambled.equals("lAl");
    }
    @Test
    boolean scramble2() {
        rand = new Random(74);
        System.out.println("'" + getWord() + "'");
        String scrambled = scrambleWord();
        System.out.println(scrambled);
        return scrambled.equals("tsaeborornussu");
    }
}
```

输出为:

```
annotations.AtUnitExample4
. words 'All'
(failed)
. scramble1 'brontosauruses'
ntsaueorosurbs
(failed)
```

```
. scramble2 'are'
are
(failed)
(3 tests)
>>> 3 FAILURES <<<
annotations.AtUnitExample4: words
annotations.AtUnitExample4: scramble1
annotations.AtUnitExample4: scramble2
```

@TestProperty 也可以用来标记那些只在测试中使用的方法，但是它们本身不是测试方法。

如果你的测试对象需要执行某些初始化工作，并且使用完成之后还需要执行清理工作，那么可以选择使用 **static** 的 **@TestObjectCleanup** 方法，当测试对象使用结束之后，该方法会为你执行清理工作。在下面的示例中，**@TestObjectCleanup** 为每一个测试对象都打开了一个文件，因此必须在丢弃测试的时候关闭该文件：

```java
// annotations/AtUnitExample5.java
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/AtUnitExample5.class}
package annotations;
import java.io.*;
import onjava.atunit.*;
import onjava.*;
public class AtUnitExample5 {
    private String text;
    public AtUnitExample5(String text) {
        this.text = text;
    }
    @Override
    public String toString() { return text; }
    @TestProperty
    static PrintWriter output;
    @TestProperty
    static int counter;
    @TestObjectCreate
    static AtUnitExample5 create() {
        String id = Integer.toString(counter++);
        try {
            output = new PrintWriter("Test" + id + ".txt");
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        return new AtUnitExample5(id);
    }
    @TestObjectCleanup
    static void cleanup(AtUnitExample5 tobj) {
        System.out.println("Running cleanup");
        output.close();
    }
    @Test
```

```
    boolean test1() {
        output.print("test1");
        return true;
    }
    @Test
    boolean test2() {
        output.print("test2");
        return true;
    }
    @Test
    boolean test3() {
        output.print("test3");
        return true;
    }
}
```

输出为:

```
annotations.AtUnitExample5
. test1
Running cleanup
. test3
Running cleanup
. test2
Running cleanup
OK (3 tests)
```

在输出中我们可以看到，清理方法会在每个测试方法结束之后自动运行。

24.4.1 在 @Unit 中使用泛型

泛型为 **@Unit** 出了一个难题，因为我们不可能“通用测试”。我们必须针对某个特定类型的参数或者参数集才能进行测试。解决方法十分简单，让测试类继承自泛型类的一个特定版本即可:

下面是一个 **stack** 的简单实现:

```
// annotations/StackL.java
// A stack built on a LinkedList
package annotations;
import java.util.*;
public class StackL<T> {
    private LinkedList<T> list = new LinkedList<>();
    public void push(T v) { list.addFirst(v); }
    public T top() { return list.getFirst(); }
    public T pop() { return list.removeFirst(); }
}
```

为了测试 String 版本，我们直接让测试类继承一个 Stack<String> :

```
// annotations/StackLStringTst.java
```

```
// Applying @Unit to generics
// {java onjava.atunit.AtUnit
// build/classes/main/annotations/StackLStringTst.class}
package annotations;
import onjava.atunit.*;
import onjava.*;
public class
StackLStringTst extends StackL<String> {
    @Test
    void tPush() {
        push("one");
        assert top().equals("one");
        push("two");
        assert top().equals("two");
    }
    @Test
    void tPop() {
        push("one");
        push("two");
        assert pop().equals("two");
        assert pop().equals("one");
    }
    @Test
    void tTop() {
        push("A");
        push("B");
        assert top().equals("B");
        assert top().equals("B");
    }
}
```

输出为:

```
annotations.StackLStringTst
. tTop
. tPush
. tPop
OK (3 tests)
```

这种方法存在的唯一缺点是，继承使我们失去了访问被测试的类中 **private** 方法的能力。这对你非常重要，那你要么把 private 方法变为 **protected**，要么添加一个非 **private** 的 **@TestProperty** 方法，由它来调用 **private** 方法（稍后我们会看到，**AtUnitRemover** 会删除产品中的 **@TestProperty** 方法）。

@Unit 搜索那些包含合适注解的类文件，然后运行 **@Test** 方法。我的主要目标就是让 **@Unit** 测试系统尽可能的透明，使得人们使用它的时候只需要添加 **@Test** 注解，而不需要特殊的编码和知识（现在版本的 **JUnit** 符合这个实践）。不过，如果说编写测试不会遇到任何困难，也不太可能，因此 **@Unit** 会尽量让这些困难变的微不足道，希望通过这种方式，你们会更乐意编写测试。

24.4.2 实现 @Unit

首先我们需要定义所有的注解类型。这些都是简单的标签，并且没有任何字段。@Test 标签在本章开头已经定义过了，这里是其他所需要的注解：

```
// onjava/atunit/TestObjectCreate.java
// The @Unit @TestObjectCreate tag
package onjava.atunit;
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {}
```

```
// onjava/atunit/TestObjectCleanup.java
// The @Unit @TestObjectCleanup tag
package onjava.atunit;
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {}
```

```
// onjava/atunit/TestProperty.java
// The @Unit @TestProperty tag
package onjava.atunit;
import java.lang.annotation.*;
// Both fields and methods can be tagged as properties:
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {}
```

所有测试的保留属性都为 **RUNTIME**，这是因为 **@Unit** 必须在编译后的代码中发现这些注解。

要实现系统并运行测试，我们还需要反射机制来提取注解。下面这个程序通过注解中的信息，决定如何构造测试对象，并在测试对象上运行测试。正是由于注解帮助，这个程序才会如此短小而直接：

```
// onjava/atunit/AtUnit.java
// An annotation-based unit-test framework
// {java onjava.atunit.AtUnit}
package onjava.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import java.nio.file.*;
import java.util.stream.*;
import onjava.*;
public class AtUnit implements ProcessFiles.Strategy {
    static Class<?> testClass;
    static List<String> failedTests= new ArrayList<>();
```

```
    static long testsRun = 0;
    static long failures = 0;
    public static void
    main(String[] args) throws Exception {
        ClassLoader.getSystemClassLoader()
                .setDefaultAssertionStatus(true); // Enable assert
        new ProcessFiles(new AtUnit(), "class").start(args);
        if(failures == 0)
            System.out.println("OK (" + testsRun + " tests)");
        else {
            System.out.println("(" + testsRun + " tests)");
            System.out.println(
                    "\n>>> " + failures + " FAILURE" +
                            (failures > 1 ? "S" : "") + " <<<");
            for(String failed : failedTests)
                System.out.println(" " + failed);
        }
    }
    @Override
    public void process(File cFile) {
        try {
            String cName = ClassNameFinder.thisClass(
                    Files.readAllBytes(cFile.toPath()));
            if(!cName.startsWith("public:"))
                return;
            cName = cName.split(":")[1];
            if(!cName.contains("."))
                return; // Ignore unpackaged classes
            testClass = Class.forName(cName);
        } catch(IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        TestMethods testMethods = new TestMethods();
        Method creator = null;
        Method cleanup = null;
        for(Method m : testClass.getDeclaredMethods()) {
            testMethods.addIfTestMethod(m);
            if(creator == null)
                creator = checkForCreatorMethod(m);
            if(cleanup == null)
                cleanup = checkForCleanupMethod(m);
        }
        if(testMethods.size() > 0) {
            if(creator == null)
                try {
                    if(!Modifier.isPublic(testClass
                            .getDeclaredConstructor()
                            .getModifiers())) {
                        System.out.println("Error: " + testClass +
                                " no-arg constructor must be public");
                        System.exit(1);
```

```
                    }
                } catch(NoSuchMethodException e) {
// Synthesized no-arg constructor; OK
                }
            System.out.println(testClass.getName());
        }
        for(Method m : testMethods) {
            System.out.print(" . " + m.getName() + " ");
            try {
                Object testObject = createTestObject(creator);
                boolean success = false;
                try {
                    if(m.getReturnType().equals(boolean.class))
                        success = (Boolean)m.invoke(testObject);
                    else {
                        m.invoke(testObject);
                        success = true; // If no assert fails
                    }
                } catch(InvocationTargetException e) {
// Actual exception is inside e:
                    System.out.println(e.getCause());
                }
                System.out.println(success ? "" : "(failed)");
                testsRun++;
                if(!success) {
                    failures++;
                    failedTests.add(testClass.getName() +
                            ": " + m.getName());
                }
                if(cleanup != null)
                    cleanup.invoke(testObject, testObject);
            } catch(IllegalAccessException |
                    IllegalArgumentException |
                    InvocationTargetException e) {
                throw new RuntimeException(e);
            }
        }
    }
    public static
    class TestMethods extends ArrayList<Method> {
        void addIfTestMethod(Method m) {
            if(m.getAnnotation(Test.class) == null)
                return;
            if(!(m.getReturnType().equals(boolean.class) ||
                    m.getReturnType().equals(void.class)))
                throw new RuntimeException("@Test method" +
                        " must return boolean or void");
            m.setAccessible(true); // If it's private, etc.
            add(m);
        }
    }
```

```
    private static
    Method checkForCreatorMethod(Method m) {
        if(m.getAnnotation(TestObjectCreate.class) == null)
            return null;
        if(!m.getReturnType().equals(testClass))
            throw new RuntimeException("@TestObjectCreate " +
                    "must return instance of Class to be tested");
        if((m.getModifiers() &
                java.lang.reflect.Modifier.STATIC) < 1)
            throw new RuntimeException("@TestObjectCreate " +
                    "must be static.");
        m.setAccessible(true);
        return m;
    }
    private static
    Method checkForCleanupMethod(Method m) {
        if(m.getAnnotation(TestObjectCleanup.class) == null)
            return null;
        if(!m.getReturnType().equals(void.class))
            throw new RuntimeException("@TestObjectCleanup " +
                    "must return void");
        if((m.getModifiers() &
                java.lang.reflect.Modifier.STATIC) < 1)
            throw new RuntimeException("@TestObjectCleanup " +
                    "must be static.");
        if(m.getParameterTypes().length == 0 ||
                m.getParameterTypes()[0] != testClass)
            throw new RuntimeException("@TestObjectCleanup " +
                    "must take an argument of the tested type.");
        m.setAccessible(true);
        return m;
    }
    private static Object
    createTestObject(Method creator) {
        if(creator != null) {
            try {
                return creator.invoke(testClass);
            } catch(IllegalAccessException |
                    IllegalArgumentException |
                    InvocationTargetException e) {
                throw new RuntimeException("Couldn't run " +
                        "@TestObject (creator) method.");
            }
        } else { // Use the no-arg constructor:
            try {
                return testClass.newInstance();
            } catch(InstantiationException |
                    IllegalAccessException e) {
                throw new RuntimeException(
                        "Couldn't create a test object. " +
                                "Try using a @TestObject method.");
```

```
            }
        }
    }
}
```

虽然它可能是“过早的重构”（因为它只在书中使用过一次），**AtUnit.java** 使用了 **ProcessFiles** 工具逐步判断命令行中的参数，决定它是一个目录还是文件，并采取相应的行为。这可以应用于不同的解决方法，是因为它包含了一个可用于自定义的 **Strategy** 接口：

```
// onjava/ProcessFiles.java
package onjava;
import java.io.*;
import java.nio.file.*;
public class ProcessFiles {
    public interface Strategy {
        void process(File file);
    }
    private Strategy strategy;
    private String ext;
    public ProcessFiles(Strategy strategy, String ext) {
        this.strategy = strategy;
        this.ext = ext;
    }
    public void start(String[] args) {
        try {
            if(args.length == 0)
                processDirectoryTree(new File("."));
            else
                for(String arg : args) {
                    File fileArg = new File(arg);
                    if(fileArg.isDirectory())
                        processDirectoryTree(fileArg);
                    else {
// Allow user to leave off extension:
                        if(!arg.endsWith("." + ext))
                            arg += "." + ext;
                        strategy.process(
                                new File(arg).getCanonicalFile());
                    }
                }
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }
    public void processDirectoryTree(File root) throws IOException {
        PathMatcher matcher = FileSystems.getDefault()
                .getPathMatcher("glob:**/*.{" + ext + "}");
        Files.walk(root.toPath())
                .filter(matcher::matches)
                .forEach(p -> strategy.process(p.toFile()));
```

```
    }
}
```

AtUnit 类实现了 **ProcessFiles.Strategy**，其包含了一个 `process()` 方法。在这种方式下，**AtUnit** 实例可以作为参数传递给 **ProcessFiles** 构造器。第二个构造器的参数告诉 **ProcessFiles** 如寻找所有包含 “class” 拓展名的文件。

如下是一个简单的使用示例：

```java
// annotations/DemoProcessFiles.java
import onjava.ProcessFiles;
public class DemoProcessFiles {
    public static void main(String[] args) {
        new ProcessFiles(file -> System.out.println(file),
                "java").start(args);
    }
}
```

输出为：

```
.\AtUnitExample1.java
.\AtUnitExample2.java
.\AtUnitExample3.java
.\AtUnitExample4.java
.\AtUnitExample5.java
.\AUComposition.java
.\AUExternalTest.java
.\database\Constraints.java
.\database\DBTable.java
.\database\Member.java
.\database\SQLInteger.java
.\database\SQLString.java
.\database\TableCreator.java
.\database\Uniqueness.java
.\DemoProcessFiles.java
.\HashSetTest.java
.\ifx\ExtractInterface.java
.\ifx\IfaceExtractorProcessor.java
.\ifx\Multiplier.java
.\PasswordUtils.java
.\simplest\Simple.java
.\simplest\SimpleProcessor.java
.\simplest\SimpleTest.java
.\SimulatingNull.java
.\StackL.java
.\StackLStringTst.java
.\Testable.java
.\UseCase.java
.\UseCaseTracker.java
```

如果没有命令行参数，这个程序会遍历当前的目录树。你还可以提供多个参数，这些参数可以是类

文件（带或不带.class 扩展名）或目录。

回到我们对 **AtUnit.java** 的讨论，因为 **@Unit** 会自动找到可测试的类和方法，所以不需要“套件”机制。

AtUnit.java 中存在的一个我们必须要解决的问题是，当它发现类文件时，类文件名中的限定类名（包括包）不明显。为了发现这个信息，必须解析类文件 - 这不是微不足道的，但也不是不可能的。找到.class 文件时，会打开它并读取其二进制数据并将其传递给 `ClassNameFinder.thisClass()`。在这里，我们正在进入“字节码工程”领域，因为我们实际上正在分析类文件的内容：

```
// onjava/atunit/ClassNameFinder.java
// {java onjava.atunit.ClassNameFinder}
package onjava.atunit;
import java.io.*;
import java.nio.file.*;
import java.util.*;
import onjava.*;
public class ClassNameFinder {
    public static String thisClass(byte[] classBytes) {
        Map<Integer,Integer> offsetTable = new HashMap<>();
        Map<Integer,String> classNameTable = new HashMap<>();
        try {
            DataInputStream data = new DataInputStream(
                    new ByteArrayInputStream(classBytes));
            int magic = data.readInt(); // 0xcafebabe
            int minorVersion = data.readShort();
            int majorVersion = data.readShort();
            int constantPoolCount = data.readShort();
            int[] constantPool = new int[constantPoolCount];
            for(int i = 1; i < constantPoolCount; i++) {
                int tag = data.read();
                // int tableSize;
                switch(tag) {
                    case 1: // UTF
                        int length = data.readShort();
                        char[] bytes = new char[length];
                        for(int k = 0; k < bytes.length; k++)
                            bytes[k] = (char)data.read();
                        String className = new String(bytes);
                        classNameTable.put(i, className);
                        break;
                    case 5: // LONG
                    case 6: // DOUBLE
                        data.readLong(); // discard 8 bytes
                        i++; // Special skip necessary
                        break;
                    case 7: // CLASS
                        int offset = data.readShort();
                        offsetTable.put(i, offset);
                        break;
```

```
                    case 8: // STRING
                        data.readShort(); // discard 2 bytes
                        break;
                    case 3: // INTEGER
                    case 4: // FLOAT
                    case 9: // FIELD_REF
                    case 10: // METHOD_REF
                    case 11: // INTERFACE_METHOD_REF
                    case 12: // NAME_AND_TYPE
                    case 18: // Invoke Dynamic
                        data.readInt(); // discard 4 bytes
                        break;
                    case 15: // Method Handle
                        data.readByte();
                        data.readShort();
                        break;
                    case 16: // Method Type
                        data.readShort();
                        break;
                    default:
                        throw
                                new RuntimeException("Bad tag " + tag);
                }
            }
            short accessFlags = data.readShort();
            String access = (accessFlags & 0x0001) == 0 ?
                    "nonpublic:" : "public:";
            int thisClass = data.readShort();
            int superClass = data.readShort();
            return access + classNameTable.get(
                    offsetTable.get(thisClass)).replace('/', '.');
        } catch(IOException | RuntimeException e) {
            throw new RuntimeException(e);
        }
    }
    // Demonstration:
    public static void main(String[] args) throws Exception {
        PathMatcher matcher = FileSystems.getDefault()
                .getPathMatcher("glob:**/*.class");
// Walk the entire tree:
        Files.walk(Paths.get(".."))
                .filter(matcher::matches)
                .map(p -> {
                    try {
                        return thisClass(Files.readAllBytes(p));
                    } catch(Exception e) {
                        throw new RuntimeException(e);
                    }
                })
                .filter(s -> s.startsWith("public:"))
// .filter(s -> s.indexOf('$') >= 0)
```

```
                    .map(s -> s.split(":")[1])
                    .filter(s -> !s.startsWith("enums."))
                    .filter(s -> s.contains("."))
                    .forEach(System.out::println);
    }
}
```

输出为:

```
onjava.ArrayShow
onjava.atunit.AtUnit$TestMethods
onjava.atunit.AtUnit
onjava.atunit.ClassNameFinder
onjava.atunit.Test
onjava.atunit.TestObjectCleanup
onjava.atunit.TestObjectCreate
onjava.atunit.TestProperty
onjava.BasicSupplier
onjava.CollectionMethodDifferences
onjava.ConvertTo
onjava.Count$Boolean
onjava.Count$Byte
onjava.Count$Character
onjava.Count$Double
onjava.Count$Float
onjava.Count$Integer
onjava.Count$Long
onjava.Count$Pboolean
onjava.Count$Pbyte
onjava.Count$Pchar
onjava.Count$Pdouble
onjava.Count$Pfloat
onjava.Count$Pint
onjava.Count$Plong
onjava.Count$Pshort
onjava.Count$Short
onjava.Count
onjava.CountingIntegerList
onjava.CountMap
onjava.Countries
onjava.Enums
onjava.FillMap
onjava.HTMLColors
onjava.MouseClick
onjava.Nap
onjava.Null
onjava.Operations
onjava.OSExecute
onjava.OSExecuteException
onjava.Pair
onjava.ProcessFiles$Strategy
```

```
onjava.ProcessFiles
onjava.Rand$Boolean
onjava.Rand$Byte
onjava.Rand$Character
onjava.Rand$Double
onjava.Rand$Float
onjava.Rand$Integer
onjava.Rand$Long
onjava.Rand$Pboolean
onjava.Rand$Pbyte
onjava.Rand$Pchar
onjava.Rand$Pdouble
onjava.Rand$Pfloat
onjava.Rand$Pint
onjava.Rand$Plong
onjava.Rand$Pshort
onjava.Rand$Short
onjava.Rand$String
onjava.Rand
onjava.Range
onjava.Repeat
onjava.RmDir
onjava.Sets
onjava.Stack
onjava.Suppliers
onjava.TimedAbort
onjava.Timer
onjava.Tuple
onjava.Tuple2
onjava.Tuple3
onjava.Tuple4
onjava.Tuple5
onjava.TypeCounter
```

虽然无法在这里介绍其中所有的细节，但是每个类文件都必须遵循一定的格式，而我已经尽力用有意义的字段来表示这些从 **ByteArrayInputStream** 中提取出来的数据片段。通过施加在输入流上的读操作，你能看出每个信息片的大小。例如每一个类的头 32 个 bit 总是一个 “神秘数字” **0xcafebabe**，而接下来的两个 **short** 值是版本信息。常量池包含了程序的常量，所以这是一个可变的值。接下来的 **short** 告诉我们这个常量池有多大，然后我们为其创建一个尺寸合适的数组。常量池中的每一个元素，其长度可能是固定式，也可能是可变的值，因此我们必须检查每一个常量的起始标记，然后才能知道该怎么做，这就是 switch 语句的工作。我们并不打算精确的分析类中所有的数据，仅仅是从文件的起始一步一步的走，直到取得我们所需的信息，因此你会发现，在这个过程中我们丢弃了大量的数据。关于类的信息都保存在 **classNameTable** 和 **offsetTable** 中。在读取常量池之后，就找到了 **this_class** 信息，这是 **offsetTable** 的一个坐标，通过它可以找到进入 **classNameTable** 的坐标，然后就可以得到我们所需的类的名字了。

现在让我们回到 **AtUtil.java** 中，process() 方法中拥有了类的名字，然后检查它是否包含“.”，如

果有就表示该类定义于一个包中。没有包的类会被忽略。如果一个类在包中，那么我们就可以使用标准的类加载器通过 `Class.forName()` 将其加载进来。现在我们可以对这个类进行 **@Unit** 注解的分析工作了。

我们只需要关注三件事：首先是 **@Test** 方法，它们被保存在 **TestMehtods** 列表中，然后检查其是否具有 @TestObjectCreate 和 **@TestObjectCleanup****** 方法。从代码中可以看到，我们通过调用相应的方法来查询注解从而找到这些方法。

每找到一个 @Test 方法，就打印出来当前类的名字，于是观察者立刻就可以知道发生了什么。接下来开始执行测试，也就是打印出方法名，然后调用 createTestObject() （如果存在一个加了 @TestObjectCreate 注解的方法），或者调用默认构造器。一旦创建出来测试对象，如果调用其上的测试方法。如果测试的返回值为 boolean，就捕获该结果。如果测试方法没有返回值，那么就没有异常发生，我们就假设测试成功，反之，如果当 assert 失败或者有任何异常抛出的时候，就说明测试失败，这时将异常信息打印出来以显示错误的原因。如果有失败的测试发生，那么还要统计失败的次数，并将失败所属的类和方法加入到 failedTests 中，以便最后报告给用户。

24.5 本章小结

注解是 Java 引入的一项非常受欢迎的补充，它提供了一种结构化，并且具有类型检查能力的新途径，从而使得你能够为代码中加入元数据，而且不会导致代码杂乱并难以阅读。使用注解能够帮助我们避免编写累赘的部署描述性文件，以及其他的生成文件。而 Javadoc 中的 @deprecated 被 @Deprecated 注解所替代的事实也说明，与注释性文字相比，注解绝对更适用于描述类相关的信息。

Java 提供了很少的内置注解。这意味着如果你在别处找不到可用的类库，那么就只能自己创建新的注解以及相应的处理器。通过将注解处理器链接到 javac，你可以一步完成编译新生成的文件，简化了构造过程。

API 的提供方和框架将会将注解作为他们工具的一部分。通过 @Unit 系统，我们可以想象，注解会极大的改变我们的 Java 编程体验。

[^3]: The Java designers coyly suggest that a mirror is where you find a reflection.

25 并发编程

> 爱丽丝："但是我不想进入疯狂的人群众"
>
> 猫咪："oh，你无能为力，我们都疯了，我疯了，你也疯了"
>
> 爱丽丝："你怎么知道我疯了"。
>
> 猫咪："你一定疯了，否则你不会来到这里"——爱丽丝梦游仙境第 6 章。

到目前为止，我们一直在编程，就像文学中的意识流叙事设备一样：首先发生一件事，然后是下一件事。我们完全控制所有步骤及其发生的顺序。如果我们将值设置为 5，那么稍后会回来并发现它是 47，这将是非常令人惊讶的。

我们现在进入了一个奇怪的并发世界，在此这个结果并不令人惊讶。你信赖的一切都不再可靠。它可能有效，也可能没有。很可能它会在某些条件下有效，而不是在其他条件下，你必须知道和了解这些情况以确定哪些有效。

作为类比，你的正常生活是在牛顿力学中发生的。物体具有质量：它们会下降并移动它们的动量。电线具有阻力，光线可以直线传播。但是，如果你进入非常小、热、冷、或者大的世界（我们不能生存），这些事情会发生变化。我们无法判断某物体是粒子还是波，光是否受到重力影响，一些物质变为超导体。

而不是单一的意识流叙事，我们在同时多条故事线进行的间谍小说里。一个间谍在一个特殊的岩石下李璐下微缩胶片，当第二个间谍来取回包裹时，它可能已经被第三个间谍带走了。但是这部特别的小说并没有把事情搞得一团糟; 你可以轻松地走到尽头，永远不会弄明白什么。

构建并发应用程序非常类似于游戏Jenga，每当你拉出一个块并将其放置在塔上时，一切都会崩溃。每个塔楼和每个应用程序都是独一无二的，有自己的作用。你从构建系统中学到的东西可能不适用于下一个系统。

本章是对并发性的一个非常基本的介绍。虽然我使用了最现代的 Java 8 工具来演示原理，但这一章远非对该主题的全面处理。我的目标是为你提供足够的基础知识，使你能够解决问题的复杂性和危险性，从而安全的通过这些鲨鱼肆虐的困难水域。

对于更多凌乱，低级别的细节，请参阅附录：并发底层原理。要进一步深入这个领域，你还必须阅读 Brian Goetz 等人的 Java Concurrency in Practice。虽然在写作时，这本书已有十多年的历史，但它仍然包含你必须了解和理解的必需品。理想情况下，本章和附录是该书的精心准备。另一个有价值的资源是 **Bill Venner** 的 Inside the Java Virtual Machine，它详细描述了 JVM 的最内部工作方式，包

括线程。

25.1 术语问题

在编程文献中并发、并行、多任务、多处理、多线程、分布式系统（以及可能的其他）使用了许多相互冲突的方式，并且经常被混淆。Brian Goetz 在 2016 年的演讲中指出了这一点From Concurrent to Parallel，他提出了一个合理的解释：

- 并发是关于正确有效地控制对共享资源的访问。
- 并行是使用额外的资源来更快地产生结果。

这些都是很好的定义，但有几十年的混乱产生了反对解决问题的历史。一般来说，当人们使用“并发”这个词时，他们的意思是“一切变得混乱”，事实上，我可能会在很多地方自己陷入这种想法，大多数书籍，包括 Brian Goetz 的 Java Concurrency in Practice，都在标题中使用这个词。

并发通常意味着“不止一个任务正在执行中”，而并行性几乎总是意味着“不止一个任务同时执行。”你可以立即看到这些定义的区别：并行也有不止一个任务“正在进行”。区别在于细节，究竟是如何“执行”发生的。此外，重叠：为并行编写的程序有时可以在单个处理器上运行，而一些并发编程系统可以利用多个处理器。

这是另一种方法，在减速 [原文：slowdown] 发生的地方写下定义：

并发

同时完成多个任务。在开始处理其他任务之前，当前任务不需要完成。并发解决了阻塞发生的问题。当任务无法进一步执行，直到外部环境发生变化时才会继续执行。最常见的例子是 I/O，其中任务必须等待一些 input（在这种情况下会被阻止）。这个问题产生在 I/O 密集型。

并行

同时在多个地方完成多个任务。这解决了所谓的计算密集型问题，如果将程序分成多个部分并在不同的处理器上编辑不同的部分，程序可以运行得更快。

术语混淆的原因在上面的定义中显示：其中核心是“在同一时间完成多个任务。”并行性通过多个处理器增加分布。更重要的是，两者解决了不同类型的问题：解决 I/O 密集型问题，并行化可能对你没有任何好处，因为问题不是整体速度，而是阻塞。并且考虑到计算力限制问题并试图在单个处理器上使用并发来解决它可能会浪费时间。两种方法都试图在更短的时间内完成更多，但它们实现加速的方式是不同的，并且取决于问题所带来的约束。

这两个概念混合在一起的一个主要原因是包括 Java 在内的许多编程语言使用相同的机制**线程**来实现并发和并行。

我们甚至可以尝试添加细致的粒度去定义（但是，这不是标准化的术语）：

- **纯并发**：任务仍然在单个 CPU 上运行。纯并发系统产生的结果比顺序系统更快，但如果有更多的处理器，则运行速度不会更快
- **并发-并行**：使用并发技术，结果程序利用更多处理器并更快地生成结果

你的一个程序有时正常有时不正常，而你已经完全遵循编程的规则，为什么？事实上我认为相信只要遵循别人所说就能得到想当然的结果的人其实是个傻瓜。

- **并行-并发**：使用并行编程技术编写，如果只有一个处理器，结果程序仍然可以运行（Java 8 **Streams** 就是一个很好的例子）。
- **纯并行**：除非有多个处理器，否则不会运行。

在某些情况下，这可能是一个有用的分类法。

对并发性的语言和库支持似乎Leaky Abstraction是完美候选者。抽象的目标是“抽象出”那些对于手头想法不重要的东西，从不必要的细节中汲取灵感。如果抽象是漏洞，那些碎片和细节会不断重新声明自己是重要的，无论你试图隐藏它们多少

我开始怀疑是否真的有高度抽象。当编写这些类型的程序时，你永远不会被底层系统和工具屏蔽，甚至关于 CPU 缓存如何工作的细节。最后，如果你非常小心，你创作的东西在特定的情况下起作用，但它在其他情况下不起作用。有时，区别在于两台机器的配置方式，或者程序的估计负载。这不是 Java 特有的-它是并发和并行编程的本质。

你可能会认为纯函数式语言没有这些限制。实际上，纯函数式语言解决了大量并发问题，所以如果你正在解决一个困难的并发问题，你可以考虑用纯函数语言编写这个部分。但最终，如果你编写一个使用队列的系统，例如，如果它没有正确调整并且输入速率要么没有被正确估计或被限制（并且限制意味着, 在不同情况下不同的东西具有不同的影响），该队列将填满并阻塞或溢出。最后，你必须了解所有细节，任何问题都可能会破坏你的系统。这是一种非常不同的编程方式

25.1.1 并发的新定义

几十年来，我一直在努力解决各种形式的并发问题，其中一个最大的挑战一直是简单地定义它。在撰写本章的过程中，我终于有了这样的洞察力，我认为可以定义它：> **并发性是一系列性能技术，专注于减少等待**

这实际上是一个相当多的声明，所以我将其分解：

- 这是一个集合：有许多不同的方法来解决这个问题。这是使定义并发性如此具有挑战性的问题之一，因为技术差别很大
- 这些是性能技术：就是这样。并发的关键点在于让你的程序运行得更快。在 Java 中，并发是非常棘手和困难的，所以绝对不要使用它，除非你有一个重大的性能问题 - 即使这样，使用最简单的方法产生你需要的性能，因为并发很快变得无法管理。
- “减少等待”部分很重要而且微妙。无论（例如）你运行多少个处理器，你只能在等待某个地方时产生结果。如果你发起 I/O 请求并立即获得结果，没有延迟，因此无需改进。如果你在多个处理器上运行多个任务，并且每个处理器都以满容量运行，并且任何其他任务都没有等待，那么尝试提高吞吐量是没有意义的。并发的唯一形式是如果程序的某些部分被迫等待。等待可以以多种形式出现 - 这解释了为什么存在如此不同的并发方法。

值得强调的是，这个定义的有效性取决于等待这个词。如果没有什么可以等待，那就没有机会了。如果有什么东西在等待，那么就会有很多方法可以加快速度，这取决于多种因素，包括系统运行的配置，

你要解决的问题类型以及其他许多问题。## 并发的超能力

想象一下，你是一部科幻电影。你必须在高层建筑中搜索一个精心巧妙地隐藏在建筑物的一千万个房间之一中的单个物品。你进入建筑物并移动走廊。走廊分开了。

你自己完成这项任务需要一百个生命周期。

现在假设你有一个奇怪的超级大国。你可以将自己分开，然后在继续前进的同时将另一半送到另一个走廊。每当你在走廊或楼梯上遇到分隔到下一层时，你都会重复这个分裂的技巧。最后，你会有一个人在整个建筑物的每个终点走廊。

每个走廊都有一千个房间。你的超级大国正在变得有点瘦，所以你只能让自己 50 个人同时搜索房间。

一旦克隆体进入房间，它必须搜索房间的所有裂缝和隐藏的口袋。它切换到第二个超级大国。它分成了一百万个纳米机器人，每个机器人都会飞到或爬到房间里一些看不见的地方。你不明白这种力量 - 一旦你启动它就会起作用。在他们自己的控制下，纳米机器人开始行动，搜索房间然后回来重新组装成你，突然，不知何故，你只知道物品是否在房间里

我很想能够说，“你在科幻小说中的超级大国？这就是并发性。“每当你有更多的任务要解决时，它就像分裂两个一样简单。问题是我们用来描述这种现象的任何模型最终都是抽象的

以下是其中一个漏洞：在理想的世界中，每次克隆自己时，你还会复制硬件处理器来运行该克隆。但当然不会发生这种情况 - 你的机器上可能有四个或八个处理器（通常在写入时）。你可能还有更多，并且仍有许多情况只有一个处理器。在抽象的讨论中，物理处理器的分配方式不仅可以泄漏，甚至可以支配你的决策

让我们在科幻电影中改变一些东西。现在当每个克隆搜索者最终到达一扇门时，他们必须敲门并等到有人回答。如果我们每个搜索者有一个处理器，这没有问题 - 处理器只是空闲，直到门被回答。但是如果我们只有 8 个处理器和数千个搜索者，那么只是因为搜索者恰好是因为处理器闲置了被锁，等待一扇门被接听。相反，我们希望将处理器应用于搜索，在那里它可以做一些真正的工作，因此需要将处理器从一个任务切换到另一个任务的机制。

许多型号能够有效地隐藏处理器的数量，并允许你假装你的数量非常大。但是有些情况会发生故障的时候，你必须知道处理器的数量，以便你可以解决这个问题。

其中一个最大的影响取决于你是单个处理器还是多个处理器。如果你只有一个处理器，那么任务切换的成本也由该处理器承担，将并发技术应用于你的系统会使它运行得更慢。

这可能会让你决定，在单个处理器的情况下，编写并发代码时没有意义。然而，有些情况下，并发模型会产生更简单的代码，实际上值得让它运行得更慢以实现。

在克隆体敲门等待的情况下，即使单处理器系统也能从并发中受益，因为它可以从等待（阻塞）的任务切换到准备好的任务。但是如果所有任务都可以一直运行那么切换的成本会降低一切，在这种情况下，如果你有多个进程，并发通常只会有意义。

在接听电话的客户服务部门，你只有一定数量的人，但是你可以拨打很多电话。那些人（处理器）必须一次拨打一个电话，直到完成电话和额外的电话必须排队。

在“鞋匠和精灵”的童话故事中，鞋匠做了很多工作，当他睡着时，一群精灵来为他制作鞋子。这里的工作是分布式的，但即使使用大量的物理处理器，在制造鞋子的某些部件时会产生限制 - 例如，如果鞋底需要制作鞋子，这会限制制鞋的速度并改变你设计解决方案的方式。

因此，你尝试解决的问题驱动解决方案的设计。打破一个“独立运行”问题的高级 [原文：lovely] 抽象，然后就是实际发生的现实。物理现实不断侵入和震撼，这种抽象。

这只是问题的一部分。考虑一个制作蛋糕的工厂。我们不知何故在工人中分发了蛋糕制作任务，但是现在是时候让工人把蛋糕放在盒子里了。那里有一个盒子，准备收到蛋糕。但是，在工人将蛋糕放入盒子之前，另一名工人投入并将蛋糕放入盒子中！我们的工人已经把蛋糕放进去了，然后就开始了！这两个蛋糕被砸碎并毁了。这是常见的“共享内存”问题，产生我们称之为竞争条件的问题，其结果取决于哪个工作人员可以首先在框中获取蛋糕（通常使用锁定机制来解决问题，因此一个工作人员可以先抓住框并防止蛋糕砸）。

当“同时”执行的任务相互干扰时，会出现问题。他可以以如此微妙和偶然的方式发生，可能公平地说，并发性“可以说是确定性的，但实际上是非确定性的。”也就是说，你可以假设编写通过维护和代码检查正常工作的并发程序。然而，在实践中，编写仅看起来可行的并发程序更为常见，但是在适当的条件下，将会失败。这些情况可能会发生，或者很少发生，你在测试期间从未看到它们。实际上，编写测试代码通常无法为并发程序生成故障条件。由此产生的失败只会偶尔发生，因此它们以客户投诉的形式出现。这是推动并发的最强有力的论据之一：如果你忽略它，你可能会被咬。

因此，并发似乎充满了危险，如果这让你有点害怕，这可能是一件好事。尽管 Java 8 在并发性方面做出了很大改进，但仍然没有像编译时验证或检查异常那样的安全网来告诉你何时出现错误。通过并发，你可以自己动手，只有知识渊博，可疑和积极，才能用 Java 编写可靠的并发代码。

25.2 并发为速度而生

在听说并发编程的问题之后，你可能会想知道它是否值得这么麻烦。答案是“不，除非你的程序运行速度不够快。”并且在决定它没有之前你会想要仔细思考。不要随便跳进并发编程的悲痛之中。如果有一种方法可以在更快的机器上运行你的程序，或者如果你可以对其进行分析并发现瓶颈并在该位置交换更快的算法，那么请执行此操作。只有在显然没有其他选择时才开始使用并发，然后仅在孤立的地方。

速度问题一开始听起来很简单：如果你想要一个程序运行得更快，将其分解成碎片并在一个单独的处理器上运行每个部分。由于我们能够提高时钟速度流（至少对于传统芯片），速度的提高是出现在多核处理器的形式而不是更快的芯片。为了使你的程序运行得更快，你必须学习利用那些超级处理器，这是并发性给你的一个建议。

使用多处理器机器，可以在这些处理器之间分配多个任务，这可以显着提高吞吐量。强大的多处理器 Web 服务器通常就是这种情况，它可以在程序中为 CPU 分配大量用户请求，每个请求分配一个线程。

但是，并发性通常可以提高在单个处理器上运行的程序的性能。这听起来像是一个双向的。如果考

我们应该重视团队的精神，一个人作用再大，也不过是一碗水中比较大的一粒水珠而已。

虑一下，由于上下文切换的成本增加（从一个任务更改为另一个任务），在单个处理器上运行的并发程序实际上应该比程序的所有部分顺序运行具有更多的开销。在表面上，将程序的所有部分作为单个任务运行并节省上下文切换的成本似乎更便宜。

可以产生影响的问题是阻塞。如果你的程序中的一个任务由于程序控制之外的某些条件（通常是 I/O）而无法继续，我们会说任务或线程阻塞（在我们的科幻故事中，克隆体已敲门而且是等待它打开）。如果没有并发性，整个程序就会停止，直到外部条件发生变化。但是，如果使用并发编写程序，则当一个任务被阻止时，程序中的其他任务可以继续执行，因此程序继续向前移动。实际上，从性能的角度来看，在单处理器机器上使用并发是没有意义的，除非其中一个任务可能阻塞。

单处理器系统中性能改进的一个常见例子是事件驱动编程，特别是用户界面编程。考虑一个程序执行一些长时间运行操作，从而最终忽略用户输入和无响应。如果你有一个“退出”按钮，你不想在你编写的每段代码中轮询它。这会产生笨拙的代码，无法保证程序员不会忘记执行检查。没有并发性，生成响应式用户界面的唯一方法是让所有任务定期检查用户输入。通过创建单独的执行线程来响应用户输入，该程序保证了一定程度的响应。

实现并发的直接方法是在操作系统级别，使用与线程不同的进程。进程是一个在自己的地址空间内运行的自包含程序。进程很有吸引力，因为操作系统通常将一个进程与另一个进程隔离，因此它们不会相互干扰，这使得进程编程相对容易。相比之下，线程共享内存和 I/O 等资源，因此编写多线程程序时遇到的困难是在不同的线程驱动的任务之间协调这些资源，一次不能通过多个任务访问它们。有些人甚至提倡将进程作为并发的唯一合理方法[1]，但不幸的是，通常存在数量和开销限制，以防止它们在并发频谱中的适用性（最终你习惯了标准的并发性克制，“这种方法适用于一些情况但不适用于其他情况”）

一些编程语言旨在将并发任务彼此隔离。这些通常被称为 _ 函数式语言 _，其中每个函数调用不产生其他影响（因此不能与其他函数干涉），因此可以作为独立的任务来驱动。Erlang 就是这样一种语言，它包括一个任务与另一个任务进行通信的安全机制。如果你发现程序的一部分必须大量使用并发性并且你在尝试构建该部分时遇到了过多的问题，那么你可能会考虑使用专用并发语言创建程序的那一部分。Java 采用了更传统的方法[2]，即在顺序语言之上添加对线程的支持而不是在多任务操作系统中分配外部进程，线程在执行程序所代表的单个进程中创建任务交换。

并发性会带来成本，包括复杂性成本，但可以通过程序设计，资源平衡和用户便利性的改进来抵消。通常，并发性使你能够创建更加松散耦合的设计; 否则，你的代码部分将被迫明确标注通常由并发处理的操作。

[1]

[2] 事实证明，质数实际上并不是散列桶的理想容量。近来，（经过广泛的测试）Java 的散列函数都使用 2 的整数次方。对现代的处理器来说，除法与求余数是最慢的操作。使用 2 的整数次方长度的散列表，可用掩码代替除法。

25.3 四句格言

在经历了多年的 Java 并发之后，我总结了以下四个格言：>1. 不要这样做 > >2. 没有什么是真的，一切可能都有问题 > >3. 它起作用，并不意味着它没有问题 > >4. 你仍然必须理解它

这些特别是关于 Java 设计中的问题，尽管它也可以应用于其他一些语言。但是，确实存在旨在防止这些问题的语言。

25.3.1 1. 不要这样做

（不要自己动手）

避免纠缠于并发产生的深层问题的最简单方法就是不要这样做。虽然它是诱人的，并且似乎足够安全，可以尝试做简单的事情，但它存在无数、微妙的陷阱。如果你可以避免它，你的生活会更容易。

证明并发性的唯一因素是速度。如果你的程序运行速度不够快 - 在这里要小心，因为只是希望它运行得更快是不合理的 - 首先应用一个分析器（参见代码校验章中分析和优化）来发现你是否可以执行其他一些优化。

如果你被迫进行并发，请采取最简单，最安全的方法来解决问题。使用众所周知的库并尽可能少地编写自己的代码。有了并发性，就没有“太简单了”。自负才是你的敌人。

25.3.2 2. 没有什么是真的，一切可能都有问题

没有并发性的编程，你会发现你的世界有一定的顺序和一致性。通过简单地将变量赋值给某个值，很明显它应该始终正常工作。

在并发领域，有些事情可能是真的而有些事情却不是，你必须认为没有什么是真的。你必须质疑一切。即使将变量设置为某个值也可能或者可能不会按预期的方式工作，并且从那里开始走下坡路。我已经很熟悉的东西，认为它显然有效但实际上并没有。

在非并发程序中你可以忽略的各种事情突然变得非常重要。例如，你必须知道处理器缓存以及保持本地缓存与主内存一致的问题。你必须了解对象构造的深度复杂性，以便你的构造对象不会意外地将数据暴露给其他线程进行更改。问题还有很多。

虽然这些主题太复杂，无法为你提供本章的专业知识（再次参见 Java Concurrency in Practice），但你必须了解它们。

25.3.3 3. 它起作用，并不意味着它没有问题

我们很容易编写出一个看似完美实则有问题的并发程序，并且往往问题直在极端情况下才暴露出来 - 在程序部署后不可避免地会出现用户问题。

- 你不能证明并发程序是正确的，你只能（有时）证明它是不正确的。

- 大多数情况下你甚至不能这样做：如果它有问题，你可能无法检测到它。
- 你通常不能编写有用的测试，因此你必须依靠代码检查结合深入的并发知识来发现错误。
- 即使是有效的程序也只能在其设计参数下工作。当超出这些设计参数时，大多数并发程序会以某种方式失败。

在其他 Java 主题中，我们培养了一种感觉-决定论。一切都按照语言的承诺（或隐含）进行，这是令人欣慰和期待的 - 毕竟，编程语言的目的是让机器做我们想要的。从确定性编程的世界进入并发编程领域，我们遇到了一种称为Dunning-Kruger效应的认知偏差，可以概括为“你知道的越多，你认为你知道得越多。”这意味着 “……相对不熟练的人拥有着虚幻的优越感，错误地评估他们的能力远高于实际。

我自己的经验是，无论你是多么确定你的代码是线程安全的，它可能已经无效了。你可以很容易地了解所有的问题，然后几个月或几年后你会发现一些概念让你意识到你编写的大多数内容实际上都容易受到并发错误的影响。当某些内容不正确时，编译器不会告诉你。为了使它正确，你必须在研究代码时掌握前脑的所有并发问题。

在 Java 的所有非并发领域，“没有明显的错误和没有明显的编译错误”似乎意味着一切都好。对于并发，它没有任何意义。你可以在这个情况下做的最糟糕的事情是“自信”。

25.3.4 4. 你必须仍然理解

在格言 1-3 之后，你可能会对并发性感到害怕，并且认为，“到目前为止，我已经避免了它，也许我可以继续保留它。

这是一种理性的反应。你可能知道其他编程语言更好地设计用于构建并发程序 - 甚至是在 JVM 上运行的程序（从而提供与 Java 的轻松通信），例如 Clojure 或 Scala。为什么不用这些语言编写并发部分并将 Java 用于其他所有部分呢?

唉，你不能轻易逃脱：

- 即使你从未明确地创建一个线程，你可能使用的框架 - 例如，Swing 图形用户界面（GUI）库，或者像 **Timer** clas 那样简单的东西。
- 这是最糟糕的事情：当你创建组件时，你必须假设这些组件可能在多线程环境中重用。即使你的解决方案是放弃并声明你的组件“不是线程安全的”，你仍然必须知道这样的声明是重要的，它是什么意思?

人们有时会认为并发性太难，不能包含在介绍该语言的书中。他们认为并发是一个可以独立对待的独立主题，并且它在日常编程中出现的少数情况（例如图形用户界面）可以用特殊的习语来处理。如果你可以避免它，为什么要介绍这样的复杂的主题。

唉，如果只是这样的话，那就太好了。但不幸的是，你无法选择何时在 Java 程序中出现线程。仅仅你从未写过自己的线程，并不意味着你可以避免编写线程代码。例如，Web 系统是最常见的 Java 应用程序之一，本质上是多线程的 Web 服务器通常包含多个处理器，而并行性是利用这些处理器的理想

调试完一个动态连接函数，固然值得兴奋，但真正的成功远还在无数个函数之后。

方式。就像这样的系统看起来那么简单，你必须理解并发才能正确地编写它。

Java 是一种多线程语言，如果你了解它们是否存在并发问题。因此，有许多 Java 程序正在使用中，或者只是偶然工作，或者大部分时间工作并且不时地发生问题，因为。有时这种问题是相对良性的，但有时它意味着丢失有价值的数据，如果你没有意识到并发问题，你最终可能会把问题放在其他地方而不是你的代码中。如果将程序移动到多处理器系统，则可以暴露或放大这些类型的问题。基本上，了解并发性使你意识到正确的程序可能会表现出错误的行为。

25.4 残酷的真相

当人类开始烹饪他们的食物时，他们大大减少了他们的身体分解和消化食物所需的能量。烹饪创造了一个“外化的胃”，从而释放出追去其他的的能力。火的使用促成了文明。

我们现在通过计算机和网络技术创造了一个“外化大脑”，开始了第二次基本转变。虽然我们只是触及表面，但已经引发了其他转变，例如设计生物机制的能力，并且已经看到文化演变的显着加速（过去，人们不得不前往混合文化，但现在他们开始混合互联网）。这些转变的影响和好处已经超出了科幻作家预测它们的能力（他们在预测文化和个人变化，甚至技术转变的次要影响方面都特别困难）。

有了这种根本性的人类变化，看到许多破坏和失败的实验并不令人惊讶。实际上，进化依赖于无数的实验，其中大多数都失败了。这些实验是向前发展的必要条件

Java 是在充满自信，热情和睿智的氛围中创建的。在发明一种编程语言时，很容易就像语言的初始可塑性会持续存在一样，你可以把某些东西拿出来，如果不能解决问题，那么就修复它。编程语言以这种方式是独一无二的 - 它们经历了类似水的改变：气态，液态和最终的固态。在气体相位期间，灵活性似乎是无限的，并且很容易认为它总是那样。一旦人们开始使用你的语言，变化就会变得更加严重，环境变得更加粘稠。语言设计的过程本身就是一门艺术。

紧迫感来自互联网的最初兴起。它似乎是一场比赛，第一个通过起跑线的人将“获胜”（事实上，Java，JavaScript 和 PHP 等语言的流行程度可以证明这一点）。唉，通过匆忙设计语言而产生的认知负荷和技术债务最终会赶上我们。

Turing completeness是不足够的; 语言需要更多的东西：它们必须能够创造性地表达，而不是用不必要的东西来衡量我们。解放我们的心理能力只是为了扭转并再次陷入困境，这是毫无意义的。我承认，尽管存在这些问题，我们已经完成了令人惊奇的事情，但我也知道如果没有这些问题我们能做得更多。

热情使原始 Java 设计师因为看起来有必要而投入功能。信心（以及原始语言的气味）让他们认为任何问题都可以解决。在时间轴的某个地方，有人认为任何加入 Java 的东西是固定的和永久性的 - 这是非常有信心，相信第一个决定永远是正确的，因此我们看到 Java 的体系中充斥着糟糕的决策。其中一些决定最终没有什么后果; 例如，你可以告诉人们不要使用 Vector，但保留了对之前版本的支持。

线程包含在 Java 1.0 中。当然，并发性是影响语言远角的基本语言设计决策，很难想象以后添加它。公平地说，当时并不清楚基本的并发性是多少。像 C 这样的其他语言能够将线程视为一个附加功能，

因此 Java 设计师也纷纷效仿，包括一个 Thread 类和必要的 JVM 支持（这比你想象的要复杂得多）。

C 语言是面向过程语言，这限制了它的野心。这些限制使附加线程库合理。当采用原始模型并将其粘贴到复杂语言中时，Java 的大规模扩展迅速暴露了基本问题。在 Thread 类中的许多方法的弃用以及后续的高级库浪潮中，这种情况变得明显，这些库试图提供更好的并发抽象。

不幸的是，为了在更高级别的语言中获得并发性，所有语言功能都会受到影响，包括最基本的功能，例如标识符代表可变值。在函数和方法中，所有不变和防止副作用的方法都会导致简化并发编程（这些是纯函数式编程语言的基础）的变化，但当时对于主流语言的创建者来说似乎是奇怪的想法。最初的 Java 设计师要么对这些选择有所了解，要么认为它们太不同了，并且会抛弃许多潜在的语言采用者。我们可以慷慨地说，语言设计社区当时根本没有足够的经验来理解调整在线程库中的影响。

Java 实验告诉我们，结果是悄然灾难性的。程序员很容易陷入认为 Java 线程并不那么困难的陷阱。似乎工作的程序充满了微妙的并发 bug。

为了获得正确的并发性，语言功能必须从头开始设计并考虑并发性。这艘船航行了;Java 将不再是为并发而设计的语言，而只是一种允许它的语言。

尽管有这些基本的不可修复的缺陷，但令人印象深刻的是它还有多远。Java 的后续版本添加了库，以便在使用并发时提升抽象级别。事实上，我根本不会想到有可能在 Java 8 中进行改进：并行流和 **CompletableFutures** - 这是惊人的史诗般的变化，我会惊奇地重复的查看它[3]。

这些改进非常有用，我们将在本章重点介绍并行流和 **CompletableFutures**。虽然它们可以大大简化你对并发和后续代码的思考方式，但基本问题仍然存在：由于 Java 的原始设计，代码的所有部分仍然容易受到攻击，你仍然必须理解这些复杂和微妙的问题。Java 中的线程绝不是简单或安全的; 那种经历必须降级为另一种更新的语言。

25.5 本章其余部分

这是我们将在本章的其余部分介绍的内容。请记住，本章的重点是使用最新的高级 Java 并发结构。使用这些使得你的生活比旧的替代品更加轻松。但是，你仍会在遗留代码中遇到一些低级工具。有时，你可能会被迫自己使用其中的一些。附录：并发底层原理包含一些更原始的 Java 并发元素的介绍。

- Parallel Streams（并行流）到目前为止，我已经强调了 Java 8 Streams 提供的改进语法。现在你对该语法（作为一个粉丝，我希望）感到满意，你可以获得额外的好处：你可以通过简单地将 parallel() 添加到表达式来并行化流。这是一种简单，强大，坦率地说是利用多处理器的惊人方式

添加 parallel() 来提高速度似乎是微不足道的，但是，唉，它就像你刚刚在残酷的真相中学到的那样简单。我将演示并解释一些盲目添加 parallel() 到 Stream 表达式的缺陷。

- 创建和运行任务任务是一段可以独立运行的代码。为了解释创建和运行任务的一些基础知识，本节介绍了一种比并行流或 CompletableFutures：Executor 更复杂的机制。执行者管理一些低级 Thread 对象（Java 中最原始的并发形式）。你创建一个任务，然后将其交给 Executorto 运行。

[3]请注意，在 64 位处理器上可能不会发生这种情况，从而消除了这个问题。

有多种类型的 Executor 用于不同的目的。在这里，我们将展示规范形式，代表创建和运行任务的最简单和最佳方法。

- 终止长时间运行的任务任务独立运行，因此需要一种机制来关闭它们。典型的方法使用了一个标志，这引入了共享内存的问题，我们将使用 Java 的“Atomic”库来回避它。
- Completable Futures 当你将衣服带到干洗店时，他们会给你一张收据。你继续完成其他任务，最终你的衣服很干净，你可以拿起它。收据是你与干洗店在后台执行的任务的连接。这是 Java 5 中引入的 Future 的方法。

Future 比以前的方法更方便，但你仍然必须出现并用收据取出干洗，等待任务没有完成。对于一系列操作，Futures 并没有真正帮助那么多。

Java 8 CompletableFuture 是一个更好的解决方案：它允许你将操作链接在一起，因此你不必将代码写入接口排序操作。有了 CompletableFuture 完美的结合，就可以更容易地做出“采购原料，组合成分，烹饪食物，提供食物，清理菜肴，储存菜肴”等一系列链式操作。

- 死锁某些任务必须去**等待 - 阻塞**来获得其他任务的结果。被阻止的任务有可能等待另一个被阻止的任务，等待另一个被阻止的任务，等等。如果被阻止的任务链循环到第一个，没有人可以取得任何进展，你就会陷入僵局。

如果在运行程序时没有立即出现死锁，则会出现最大的问题。你的系统可能容易出现死锁，并且只会在某些条件下死锁。程序可能在某个平台上运行正常，例如你的开发机器，但是当你将其部署到不同的硬件时会开始死锁。

死锁通常源于细微的编程错误; 一系列无辜的决定，最终意外地创建了一个依赖循环。本节包含一个经典示例，演示了死锁的特性。

我们将通过模拟创建披萨的过程完成本章，首先使用并行流实现它，然后是完成配置。这不仅仅是两种方法的比较，更重要的是探索你应该投入多少工作来加速计划。

- 努力，复杂，成本 ## 并行流

Java 8 流的一个显着优点是，在某些情况下，它们可以很容易地并行化。这来自仔细的库设计，特别是流使用内部迭代的方式 - 也就是说，它们控制着自己的迭代器。特别是，他们使用一种特殊的迭代器，称为 Spliterator，它被限制为易于自动分割。这产生了相当神奇的结果，即能够简单用 parallel() 然后流中的所有内容都作为一组并行任务运行。如果你的代码是使用 Streams 编写的，那么并行化以提高速度似乎是一种琐事

例如，考虑来自 Streams 的 Prime.java。查找质数可能是一个耗时的过程，我们可以看到该程序的计时：

```
// concurrent/ParallelPrime.java
import java.util.*;
import java.util.stream.*;
import static java.util.stream.LongStream.*;
import java.io.*;
import java.nio.file.*;
```

```
import onjava.Timer;

public class ParallelPrime {
    static final int COUNT = 100_000;
    public static boolean isPrime(long n){
        return rangeClosed(2, (long)Math.sqrt(n)).noneMatch(i -> n % i
            == 0);
        }
    public static void main(String[] args)
        throws IOException {
        Timer timer = new Timer();
        List<String> primes =
            iterate(2, i -> i + 1)
                .parallel()              // [1]
                .filter(ParallelPrime::isPrime)
                .limit(COUNT)
                .mapToObj(Long::toString)
                .collect(Collectors.toList());
        System.out.println(timer.duration());
        Files.write(Paths.get("primes.txt"), primes, StandardOpenOption
            .CREATE);
        }
    }
```

输出结果:

```
    Output:
    1224
```

请注意，这不是微基准测试，因为我们计时整个程序。我们将数据保存在磁盘上以防止过激的优化; 如果我们没有对结果做任何事情，那么一个高级的编译器可能会观察到程序没有意义并且消除了计算（这不太可能，但并非不可能）。请注意使用 nio2 库编写文件的简单性（在文件一章中有描述）。

当我注释掉1 parallel() 行时，我的结果大约是 parallel() 的三倍。

并行流似乎是一个甜蜜的交易。你所需要做的就是将编程问题转换为流，然后插入 parallel() 以加快速度。实际上，有时候这很容易。但遗憾的是，有许多陷阱。

- parallel() 不是灵丹妙药

作为对流和并行流的不确定性的探索，让我们看一个看似简单的问题：求和数字的增量序列。事实证明这是一个令人惊讶的数量，并且我将冒险将它们进行比较 - 试图小心，但承认我可能会在计时代码执行时遇到许多基本陷阱之一。结果可能有一些缺陷（例如 JVM 没有“升温”），但我认为它仍然提供了一些有用的指示。

我将从一个计时方法 rigorously 开始，它采用 **LongSupplier**，测量 **getAsLong()** 调用的长度，将结果与 **checkValue** 进行比较并显示结果。

请注意，一切都必须严格使用 **long**; 我花了一些时间发现隐蔽的溢出，然后才意识到在重要的地方错过了 **long**。

一个人静静坐在电脑面前写代码的感觉，那是什么感觉? 那是武林高手闭关修炼的感觉。

所有关于时间和内存的数字和讨论都是指“我的机器”。你的经历可能会有所不同。

```
// concurrent/Summing.java
import java.util.stream.*;
import java.util.function.*;
import onjava.Timer;
public class Summing {
    static void timeTest(String id, long checkValue,    LongSupplier operation){
        System.out.print(id + ": ");
        Timer timer = newTimer();
        long result = operation.getAsLong();
        if(result == checkValue)
            System.out.println(timer.duration() + "ms");
        else
            System.out.format("result: %d%ncheckValue: %d%n", result, checkValue);
        }
    public static final int SZ = 100_000_000;// This even works://
    public static final int SZ = 1_000_000_000;
    public static final long CHECK = (long)SZ * ((long)SZ + 1)/2; // Gauss's formula
    public static void main(String[] args){
        System.out.println(CHECK);
        timeTest("Sum Stream", CHECK, () ->
        LongStream.rangeClosed(0, SZ).sum());
        timeTest("Sum Stream Parallel", CHECK, () ->
        LongStream.rangeClosed(0, SZ).parallel().sum());
        timeTest("Sum Iterated", CHECK, () ->
        LongStream.iterate(0, i -> i + 1)
        .limit(SZ+1).sum());
        // Slower & runs out of memory above 1_000_000:
        // timeTest("Sum Iterated Parallel", CHECK, () ->
        //   LongStream.iterate(0, i -> i + 1)
        //     .parallel()
        //     .limit(SZ+1).sum());
    }
}
```

输出结果:

```
5000000050000000
Sum Stream: 167ms
Sum Stream Parallel: 46ms
Sum Iterated: 284ms
```

CHECK 值是使用 Carl Friedrich Gauss 在 1700 年代后期仍在小学时创建的公式计算出来的.

main() 的第一个版本使用直接生成 **Stream** 并调用 **sum()** 的方法。我们看到流的好处在于十亿分之一的 SZ 在没有溢出的情况下处理（我使用较小的数字，因此程序运行时间不长）。使用 **parallel()**

的基本范围操跟快。

如果使用 **iterate()** 来生成序列，则减速是戏剧性的，可能是因为每次生成数字时都必须调用 lambda。但是如果我们尝试并行化，那么结果通常比非并行版本花费的时间更长，但是当 **SZ** 超过一百万时，它也会耗尽内存（在某些机器上）。当然，当你可以使用 **range()** 时，你不会使用 **iterate()**，但如果你生成的东西不是简单的序列，你必须使用 **iterate()**。应用 **parallel()** 是一个合理的尝试，但会产生令人惊讶的结果。我们将在后面的部分中探讨内存限制的原因，但我们可以对流并行算法进行初步观察：

- 流并行性将输入数据分成多个部分，因此算法可以应用于那些单独的部分。
- 阵列分割成本低廉，均匀且具有完美的分裂知识。
- 链接列表没有这些属性;“拆分”一个链表仅仅意味着把它分成“第一元素”和“其余列表”，这相对无用。
- 无状态生成器的行为类似于数组; 使用上述范围是无可争议的。
- 迭代生成器的行为类似于链表; **iterate()** 是一个迭代生成器。

现在让我们尝试通过在数组中填充值来填充数组来解决问题。因为数组只分配了一次，所以我们不太可能遇到垃圾收集时序问题。

首先我们将尝试一个充满原始 **long** 的数组：

```
// concurrent/Summing2.java
// {ExcludeFromTravisCI}import java.util.*;
public class Summing2 {
    static long basicSum(long[] ia) {
        long sum = 0;
        int size = ia.length;
        for(int i = 0; i < size; i++)
            sum += ia[i];return sum;
    }
    // Approximate largest value of SZ before
    // running out of memory on mymachine:
    public static final int SZ = 20_000_000;
    public static final long CHECK = (long)SZ * ((long)SZ + 1)/2;
    public static void main(String[] args) {
        System.out.println(CHECK);
        long[] la = newlong[SZ+1];
        Arrays.parallelSetAll(la, i -> i);
        Summing.timeTest("Array Stream Sum", CHECK, () ->
        Arrays.stream(la).sum());
        Summing.timeTest("Parallel", CHECK, () ->
        Arrays.stream(la).parallel().sum());
        Summing.timeTest("Basic Sum", CHECK, () ->
        basicSum(la));// Destructive summation:
        Summing.timeTest("parallelPrefix", CHECK, () -> {
            Arrays.parallelPrefix(la, Long::sum);
        return la[la.length - 1];
        });
```

```
    }
}
```

输出结果:

```
200000010000000
Array Stream
Sum: 104ms
Parallel: 81ms
Basic Sum: 106ms
parallelPrefix: 265ms
```

第一个限制是内存大小; 因为数组是预先分配的，所以我们不能创建几乎与以前版本一样大的任何东西。并行化可以加快速度，甚至比使用 **basicSum()** 循环更快。有趣的是，**Arrays.parallelPrefix()** 似乎实际上减慢了速度。但是，这些技术中的任何一种在其他条件下都可能更有用 - 这就是为什么你不能做出任何确定性的声明，除了“你必须尝试一下”。”

最后，考虑使用盒装 **Long** 的效果:

```java
// concurrent/Summing3.java
// {ExcludeFromTravisCI}
import java.util.*;
public class Summing3 {
    static long basicSum(Long[] ia) {
        long sum = 0;
        int size = ia.length;
        for(int i = 0; i < size; i++)
            sum += ia[i];
            return sum;
    }
    // Approximate largest value of SZ before
    // running out of memory on my machine:
    public static final int SZ = 10_000_000;
    public static final long CHECK = (long)SZ * ((long)SZ + 1)/2;
    public static void main(String[] args) {
        System.out.println(CHECK);
        Long[] aL = newLong[SZ+1];
        Arrays.parallelSetAll(aL, i -> (long)i);
        Summing.timeTest("Long Array Stream Reduce", CHECK, () ->
        Arrays.stream(aL).reduce(0L, Long::sum));
        Summing.timeTest("Long Basic Sum", CHECK, () ->
        basicSum(aL));
        // Destructive summation:
        Summing.timeTest("Long parallelPrefix",CHECK, ()-> {
            Arrays.parallelPrefix(aL, Long::sum);
            return aL[aL.length - 1];
            });
    }
}
```

输出结果:

```
50000005000000
Long Array
Stream Reduce: 1038ms
Long Basic
Sum: 21ms
Long parallelPrefix: 3616ms
```

现在可用的内存量大约减半，并且所有情况下所需的时间都会很长，除了 **basicSum()**，它只是循环遍历数组。令人惊讶的是，**Arrays.parallelPrefix()** 比任何其他方法都要花费更长的时间。

我将 **parallel()** 版本分开了，因为在上面的程序中运行它导致了一个冗长的垃圾收集，扭曲了结果:

```java
// concurrent/Summing4.java
// {ExcludeFromTravisCI}
import java.util.*;
public class Summing4 {
    public static void main(String[] args) {
        System.out.println(Summing3.CHECK);
        Long[] aL = newLong[Summing3.SZ+1];
        Arrays.parallelSetAll(aL, i -> (long)i);
        Summing.timeTest("Long Parallel",
        Summing3.CHECK, () ->
        Arrays.stream(aL)
        .parallel()
        .reduce(0L,Long::sum));
    }
}
```

输出结果:

```
50000005000000
Long Parallel: 1014ms
```

它比非 parallel() 版本略快，但并不显着。

这种时间增加的一个重要原因是处理器内存缓存。使用 **Summing2.java** 中的原始 **long**，数组 **la** 是连续的内存。处理器可以更容易地预测该阵列的使用，并使缓存充满下一个需要的阵列元素。访问缓存比访问主内存快得多。似乎 **Long parallelPrefix** 计算受到影响，因为它为每个计算读取两个数组元素，并将结果写回到数组中，并且每个都为 **Long** 生成一个超出缓存的引用。

使用 **Summing3.java** 和 **Summing4.java**，**aL** 是一个 **Long** 数组，它不是一个连续的数据数组，而是一个连续的 **Long** 对象引用数组。尽管该数组可能会在缓存中出现，但指向的对象几乎总是超出缓存。

这些示例使用不同的 SZ 值来显示内存限制。

为了进行时间比较，以下是 SZ 设置为最小值 1000 万的结果:

Sum Stream: 69msSum Stream Parallel: 18msSum Iterated: 277ms Array Stream Sum: 57ms Parallel: 14ms Basic Sum: 16ms parallelPrefix: 28ms Long Array Stream Reduce: 1046ms Long Basic Sum: 21ms Long parallelPrefix: 3287ms Long Parallel: 1008ms

虽然 Java 8 的各种内置"并行"工具非常棒，但我认为它们被视为神奇的灵丹妙药："只需添加 parallel() 并且它会更快！"我希望我已经开始表明情况并非所有都是如此，并且盲目地应用内置的"并行"操作有时甚至会使运行速度明显变慢。

- parallel()/limit() 交点

使用 parallel() 时会有更复杂的问题。从其他语言中吸取的流是围绕无限流模型设计的。如果你拥有有限数量的元素，则可以使用集合以及为有限大小的集合设计的关联算法。如果你使用无限流，则使用针对流优化的算法。

Java 8 将两者合并起来。例如，**Collections** 没有内置的 **map()** 操作。在 Collection 和 Map 中唯一类似流的批处理操作是 **forEach()**。如果要执行 **map()** 和 **reduce()** 等操作，必须首先将 Collection 转换为存在这些操作的 Stream:

```java
// concurrent/CollectionIntoStream.java
import onjava.*;
import java.util.*;
import java.util.stream.*;
public class CollectionIntoStream {
    public static void main(String[] args) {
    List<String> strings = Stream.generate(new Rand.String(5))
    .limit(10)
    .collect(Collectors.toList());
    strings.forEach(System.out::println);
    // Convert to a Stream for many more options:
    String result = strings.stream()
    .map(String::toUpperCase)
    .map(s -> s.substring(2))
    .reduce(":", (s1, s2) -> s1 + s2);
    System.out.println(result);
    }
}
```

输出结果:

```
pccux
szgvg
meinn
eeloz
tdvew
cippc
ygpoa
lkljl
bynxt
```

```
:PENCUXGVGINNLOZVEWPPCPOALJLNXT
```

Collection 确实有一些批处理操作，如 **removeAll()**，**removeIf()** 和 **retainAll()**，但这些都是破坏性的操作.**ConcurrentHashMap** 对 **forEachand** 和 **reduce** 操作有特别广泛的支持。

在许多情况下，只在集合上调用 **stream()** 或者 **parallelStream()** 没有问题。但是，有时将 **Stream** 与 **Collection** 混合会产生意外。这是一个有趣的难题：

```java
// concurrent/ParallelStreamPuzzle.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
public class ParallelStreamPuzzle {
    static class IntGenerator
    implements Supplier<Integer> {
        private int current = 0;
        public Integer get() {
            return current++;
        }
    }
    public static void main(String[] args) {
        List<Integer> x = Stream.generate(newIntGenerator())
        .limit(10)
        .parallel()  // [1]
        .collect(Collectors.toList());
        System.out.println(x);
    }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
*/
```

如果1注释运行它，它会产生预期的：**[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]** 每次。但是包含了 parallel()，它看起来像一个随机数生成器，带有输出（从一次运行到下一次运行不同），如：**[0, 3, 6, 8, 11, 14, 17, 20, 23, 26]** 这样一个简单的程序怎么会这么破碎呢？让我们考虑一下我们在这里要实现的目标："并行生成。""那意味着什么？一堆线程都在拉动一个生成器，在某种程度上选择一组有限的结果？代码使它看起来很简单，但它转向是一个特别凌乱的问题。

为了看到它，我们将添加一些仪器。由于我们正在处理线程，因此我们必须将任何跟踪信息捕获到并发数据结构中。在这里我使用 **ConcurrentLinkedDeque**：

```java
// concurrent/ParallelStreamPuzzle2.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.nio.file.*;
public class ParallelStreamPuzzle2 {
```

```
    public static final Deque<String> trace =
    new ConcurrentLinkedDeque<>();
    static class
    IntGenerator implements Supplier<Integer> {
        private AtomicInteger current =
        new AtomicInteger();
        public Integerget() {
            trace.add(current.get() + ": " +Thread.currentThread().
                getName());
            return current.getAndIncrement();
        }
    }
    public static void main(String[] args) throws Exception {
    List<Integer> x = Stream.generate(newIntGenerator())
    .limit(10)
    .parallel()
    .collect(Collectors.toList());
    System.out.println(x);
    Files.write(Paths.get("PSP2.txt"), trace);
    }
}
```

输出结果:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

current 是使用线程安全的 **AtomicInteger** 类定义的，可以防止竞争条件；**parallel()** 允许多个线程调用 **get()**。

在查看 **PSP2.txt**.**IntGenerator.get()** 被调用 1024 次时，你可能会感到惊讶。

0: main 1: ForkJoinPool.commonPool-worker-1 2: ForkJoinPool.commonPool-worker-2 3: ForkJoinPool.commonPool-worker-2 4: ForkJoinPool.commonPool-worker-1 5: ForkJoinPool.commonPool-worker-1 6: ForkJoinPool.commonPool-worker-1 7: ForkJoinPool.commonPool-worker-1 8: ForkJoinPool.commonPool-worker-4 9: ForkJoinPool.commonPool-worker-4 10: ForkJoinPool.commonPool-worker-4 11: main 12: main 13: main 14: main 15: main…10 17: ForkJoinPool.commonPool-worker-110 18: ForkJoinPool.commonPool-worker-610 19: ForkJoinPool.commonPool-worker-610 20: ForkJoinPool.commonPool-worker-110 21: ForkJoinPool.commonPool-worker-110 22: ForkJoinPool.commonPool-worker-110 23: ForkJoinPool.commonPool-worker-1

这个块大小似乎是内部实现的一部分（尝试使用 **limit()** 的不同参数来查看不同的块大小）。将 **parallel()** 与 **limit()** 结合使用可以预取一串值，作为流输出。

试着想象一下这里发生了什么：一个流抽象出无限序列，按需生成。当你要求它并行产生流时，你要求所有这些线程尽可能地调用 get()。添加 limit()，你说“只需要这些。”基本上，当你将 parallel() 与 limit() 结合使用时，你要求随机输出 - 这可能对你正在解决的问题很好。但是当你这样做时，你必须明

白。这是一个仅限专家的功能，而不是要争辩说“Java 弄错了”。

什么是更合理的方法来解决问题? 好吧, 如果你想生成一个 int 流, 你可以使用 IntStream.range(), 如下所示:

```
// concurrent/ParallelStreamPuzzle3.java
// {VisuallyInspectOutput}
import java.util.*;
import java.util.stream.*;
public class ParallelStreamPuzzle3 {
    public static void main(String[] args) {
    List<Integer> x = IntStream.range(0, 30)
        .peek(e -> System.out.println(e + ": " +Thread.currentThread()
        .getName()))
        .limit(10)
        .parallel()
        .boxed()
        .collect(Collectors.toList());
        System.out.println(x);
    }
}
```

输出结果:

```
8: main
6: ForkJoinPool.commonPool-worker-5
3: ForkJoinPool.commonPool-worker-7
5: ForkJoinPool.commonPool-worker-5
1: ForkJoinPool.commonPool-worker-3
2: ForkJoinPool.commonPool-worker-6
4: ForkJoinPool.commonPool-worker-1
0: ForkJoinPool.commonPool-worker-4
7: ForkJoinPool.commonPool-worker-1
9: ForkJoinPool.commonPool-worker-2
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

为了表明 **parallel()** 确实有效，我添加了一个对 **peek()** 的调用，这是一个主要用于调试的流函数：它从流中提取一个值并执行某些操作但不影响从流向下传递的元素。注意这会干扰线程行为，但我只是尝试在这里做一些事情，而不是实际调试任何东西。

你还可以看到 boxed() 的添加，它接受 int 流并将其转换为 Integer 流。

现在我们得到多个线程产生不同的值，但它只产生 10 个请求的值，而不是 1024 个产生 10 个值。

它更快吗？一个更好的问题是：什么时候开始有意义？当然不是这么小的一套; 上下文切换的代价远远超过并行性的任何加速。当一个简单的数字序列并行生成时，有点难以想象。如果你使用昂贵的产品，它可能有意义 - 但这都是猜测。唯一知道的是通过测试。记住这句格言：“首先制作它，然后快速制作 - 但只有你必须这样做。”**parallel()** 和 **limit()** 仅供专家使用（并且要清楚，我不认为自己是这里的

专家)。

- 并行流只看起来很容易

实际上，在许多情况下，并行流确实可以毫不费力地更快地产生结果。但正如你所见，只需将 **parallel()** 打到你的 Stream 操作上并不一定是安全的事情。在使用 **parallel()** 之前，你必须了解并行性如何帮助或损害你的操作。有个错误认识是认为并行性总是一个好主意。事实上并不是。Stream 意味着你不需要重写所有代码以便并行运行它。流什么都不做的是取代理解并行性如何工作的需要，以及它是否有助于实现你的目标。

25.6 创建和运行任务

如果无法通过并行流实现并发，则必须创建并运行自己的任务。稍后你将看到运行任务的理想 Java 8 方法是 CompletableFuture，但我们将使用更基本的工具介绍概念。

Java 并发的历史始于非常原始和有问题的机制，并且充满了各种尝试的改进。这些主要归入附录：低级并发 (Appendix: Low-Level Concurrency)。在这里，我们将展示一个规范形式，表示创建和运行任务的最简单，最好的方法。与并发中的所有内容一样，存在各种变体，但这些变体要么降级到该附录，要么超出本书的范围。

- Tasks and Executors

在 Java 的早期版本中，你通过直接创建自己的 Thread 对象来使用线程，甚至将它们子类化以创建你自己的特定“任务线程”对象。你手动调用了构造函数并自己启动了线程。

创建所有这些线程的开销变得非常重要，现在不鼓励采用实际操作方法。在 Java 5 中，添加了类来为你处理线程池。你可以将任务创建为单独的类型，然后将其交给 ExecutorService 以运行该任务，而不是为每种不同类型的任务创建新的 Thread 子类型。ExecutorService 为你管理线程，并且在运行任务后重新循环线程而不是丢弃线程。

首先，我们将创建一个几乎不执行任务的任务。它“sleep”（暂停执行）100 毫秒，显示其标识符和正在执行任务的线程的名称，然后完成：

```
// concurrent/NapTask.java
import onjava.Nap;
public class NapTask implements Runnable {
    finalint id;
    public NapTask(int id) {
        this.id = id;
        }
    @Override
    public void run() {
        new Nap(0.1);// Seconds
        System.out.println(this + " "+
            Thread.currentThread().getName());
        }
```

```
    @Override
    public String toString() {
        return"NapTask[" + id + "]";
    }
}
```

这只是一个 **Runnable**：一个包含 **run()** 方法的类。它没有包含实际运行任务的机制。我们使用 **Nap** 类中的“sleep”：

```
// onjava/Nap.java
package onjava;
import java.util.concurrent.*;
public class Nap {
    public Nap(double t) { // Seconds
        try {
            TimeUnit.MILLISECONDS.sleep((int)(1000 * t));
        } catch(InterruptedException e){
            throw new RuntimeException(e);
        }
    }
    public Nap(double t, String msg) {
        this(t);
        System.out.println(msg);
    }
}
```

为了消除异常处理的视觉噪声，这被定义为实用程序。第二个构造函数在超时时显示一条消息

对 **TimeUnit.MILLISECONDS.sleep()** 的调用获取“当前线程”并在参数中将其置于休眠状态，这意味着该线程被挂起。这并不意味着底层处理器停止。操作系统将其切换到其他任务，例如在你的计算机上运行另一个窗口。OS 任务管理器定期检查 **sleep()** 是否超时。当它执行时，线程被“唤醒”并给予更多处理时间。

你可以看到 **sleep()** 抛出一个已检查的 **InterruptedException**; 这是原始 Java 设计中的一个工件，它通过突然断开它们来终止任务。因为它往往会产生不稳定的状态，所以后来不鼓励终止。但是，我们必须在需要或仍然发生终止的情况下捕获异常。

要执行任务，我们将从最简单的方法–SingleThreadExecutor 开始:

```
//concurrent/SingleThreadExecutor.java
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.*;
public class SingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService exec =
            Executors.newSingleThreadExecutor();
        IntStream.range(0, 10)
            .mapToObj(NapTask::new)
            .forEach(exec::execute);
```

```
        System.out.println("All tasks submitted");
        exec.shutdown();
        while(!exec.isTerminated()) {
            System.out.println(
            Thread.currentThread().getName()+
            " awaiting termination");
            new Nap(0.1);
        }
    }
}
```

输出结果:

```
All tasks submitted
main awaiting termination
main awaiting termination
NapTask[0] pool-1-thread-1
main awaiting termination
NapTask[1] pool-1-thread-1
main awaiting termination
NapTask[2] pool-1-thread-1
main awaiting termination
NapTask[3] pool-1-thread-1
main awaiting termination
NapTask[4] pool-1-thread-1
main awaiting termination
NapTask[5] pool-1-thread-1
main awaiting termination
NapTask[6] pool-1-thread-1
main awaiting termination
NapTask[7] pool-1-thread-1
main awaiting termination
NapTask[8] pool-1-thread-1
main awaiting termination
NapTask[9] pool-1-thread-1
```

首先请注意，没有 **SingleThreadExecutor** 类。**newSingleThreadExecutor()** 是 **Executors** 中的工厂，它创建特定类型的[4]

我创建了十个 NapTasks 并将它们提交给 ExecutorService，这意味着它们开始自己运行。然而，在此期间，main() 继续做事。当我运行 callexec.shutdown() 时，它告诉 ExecutorService 完成已经提交的任务，但不接受任何新任务。此时，这些任务仍然在运行，因此我们必须等到它们在退出 main() 之前完成。这是通过检查 exec.isTerminated() 来实现的，这在所有任务完成后变为 true。

请注意，main() 中线程的名称是 main，并且只有一个其他线程 pool-1-thread-1。此外，交错输出显示两个线程确实同时运行。

如果你只是调用 exec.shutdown()，程序将完成所有任务。也就是说，虽然不需要（!

[4]这个测试的推论是，“如果某人表示线程是容易并且简单的，请确保这个人没有对你的项目做出重要的决策。如果那个人已经做出，那么你就已经陷入麻烦之中了。”

exec.isTerminated())。

```
// concurrent/SingleThreadExecutor2.java
import java.util.concurrent.*;
import java.util.stream.*;
public class SingleThreadExecutor2 {
    public static void main(String[] args)throws InterruptedException {
        ExecutorService exec
        =Executors.newSingleThreadExecutor();
        IntStream.range(0, 10)
            .mapToObj(NapTask::new)
            .forEach(exec::execute);
        exec.shutdown();
    }
}
```

输出结果:

```
NapTask[0] pool-1-thread-1
NapTask[1] pool-1-thread-1
NapTask[2] pool-1-thread-1
NapTask[3] pool-1-thread-1
NapTask[4] pool-1-thread-1
NapTask[5] pool-1-thread-1
NapTask[6] pool-1-thread-1
NapTask[7] pool-1-thread-1
NapTask[8] pool-1-thread-1
NapTask[9] pool-1-thread-1
```

一旦你 callexec.shutdown()，尝试提交新任务将抛出 RejectedExecutionException。

```
// concurrent/MoreTasksAfterShutdown.java
import java.util.concurrent.*;
public class MoreTasksAfterShutdown {
    public static void main(String[] args) {
        ExecutorService exec
        =Executors.newSingleThreadExecutor();
        exec.execute(newNapTask(1));
        exec.shutdown();
        try {
            exec.execute(newNapTask(99));
        } catch(RejectedExecutionException e) {
            System.out.println(e);
        }
    }
}
```

输出结果:

```
java.util.concurrent.RejectedExecutionException: TaskNapTask[99]
    rejected from java.util.concurrent.ThreadPoolExecutor@4e25154f[
```

```
    Shutting down, pool size = 1,active threads = 1, queued tasks = 0,
    completed tasks =0]NapTask[1] pool-1-thread-1
```

exec.shutdown() 的替代方法是 **exec.shutdownNow()**，它除了不接受新任务外，还会尝试通过中断任务来停止任何当前正在运行的任务。同样，中断是错误的，容易出错并且不鼓励。

- 使用更多线程

使用线程的重点是（几乎总是）更快地完成任务，那么我们为什么要限制自己使用 SingleThreadExecutor 呢？查看执行 **Executors** 的 Javadoc，你将看到更多选项。例如 CachedThreadPool：

```java
// concurrent/CachedThreadPool.java
import java.util.concurrent.*;
import java.util.stream.*;
public class CachedThreadPool {
    public static void main(String[] args) {
        ExecutorService exec
        =Executors.newCachedThreadPool();
        IntStream.range(0, 10)
        .mapToObj(NapTask::new)
        .forEach(exec::execute);
        exec.shutdown();
    }
}
```

输出结果：

```
NapTask[7] pool-1-thread-8
NapTask[4] pool-1-thread-5
NapTask[1] pool-1-thread-2
NapTask[3] pool-1-thread-4
NapTask[0] pool-1-thread-1
NapTask[8] pool-1-thread-9
NapTask[2] pool-1-thread-3
NapTask[9] pool-1-thread-10
NapTask[6] pool-1-thread-7
NapTask[5] pool-1-thread-6
```

当你运行这个程序时，你会发现它完成得更快。这是有道理的，而不是使用相同的线程来顺序运行每个任务，每个任务都有自己的线程，所以它们都并行运行。似乎没有缺点，很难看出为什么有人会使用 SingleThreadExecutor。

要理解这个问题，我们需要一个更复杂的任务：

```java
// concurrent/InterferingTask.java
public class InterferingTask implements Runnable {
    final int id;
    private static Integer val = 0;
    public InterferingTask(int id) {
        this.id = id;
    }
```

```
    @Override
    public void run() {
        for(int i = 0; i < 100; i++)
        val++;
    System.out.println(id + " "+
        Thread.currentThread().getName() + " " + val);
    }
}
```

每个任务增加 val 一百次。这似乎很简单。让我们用 CachedThreadPool 尝试一下:

```
// concurrent/CachedThreadPool2.java
import java.util.concurrent.*;
import java.util.stream.*;
public class CachedThreadPool2 {
    public static void main(String[] args) {
    ExecutorService exec
    =Executors.newCachedThreadPool();
    IntStream.range(0, 10)
    .mapToObj(InterferingTask::new)
    .forEach(exec::execute);
    exec.shutdown();
    }
}
```

输出结果:

```
0 pool-1-thread-1 200
1 pool-1-thread-2 200
4 pool-1-thread-5 300
5 pool-1-thread-6 400
8 pool-1-thread-9 500
9 pool-1-thread-10 600
2 pool-1-thread-3 700
7 pool-1-thread-8 800
3 pool-1-thread-4 900
6 pool-1-thread-7 1000
```

输出不是我们所期望的，并且从一次运行到下一次运行会有所不同。问题是所有的任务都试图写入 val 的单个实例，并且他们正在踩着彼此的脚趾。我们说这样的类不是线程安全的。让我们看看 SingleThreadExecutor 会发生什么:

```
// concurrent/SingleThreadExecutor3.java
import java.util.concurrent.*;
import java.util.stream.*;
public class SingleThreadExecutor3 {
    public static void main(String[] args)throws InterruptedException {
        ExecutorService exec
        =Executors.newSingleThreadExecutor();
        IntStream.range(0, 10)
```

```
        .mapToObj(InterferingTask::new)
        .forEach(exec::execute);
        exec.shutdown();
    }
}
```

输出结果:

```
0 pool-1-thread-1 100
1 pool-1-thread-1 200
2 pool-1-thread-1 300
3 pool-1-thread-1 400
4 pool-1-thread-1 500
5 pool-1-thread-1 600
6 pool-1-thread-1 700
7 pool-1-thread-1 800
8 pool-1-thread-1 900
9 pool-1-thread-1 1000
```

现在我们每次都得到一致的结果，尽管 **InterferingTask** 缺乏线程安全性。这是 SingleThreadExecutor 的主要好处 - 因为它一次运行一个任务，这些任务不会相互干扰，因此强加了线程安全性。这种现象称为线程限制，因为在单线程上运行任务限制了它们的影响。线程限制限制了加速，但可以节省很多困难的调试和重写。

- 产生结果

因为 **InterferingTask** 是一个 **Runnable**，它没有返回值，因此只能使用副作用产生结果 - 操纵缓冲值而不是返回结果。副作用是并发编程中的主要问题之一，因为我们看到了 **CachedThreadPool2.java**。**InterferingTask** 中的 **val** 被称为可变共享状态，这就是问题所在：多个任务同时修改同一个变量会产生竞争。结果取决于首先在终点线上执行哪个任务，并修改变量（以及其他可能性的各种变化）。

避免竞争条件的最好方法是避免可变的共享状态。我们可以称之为自私的孩子原则：什么都不分享。

使用 **InterferingTask**，最好删除副作用并返回任务结果。为此，我们创建 **Callable** 而不是 **Runnable**:

```
// concurrent/CountingTask.java
import java.util.concurrent.*;
public class CountingTask implements Callable<Integer> {
    final int id;
    public CountingTask(int id) { this.id = id; }
    @Override
    public Integer call() {
    Integer val = 0;
    for(int i = 0; i < 100; i++)
        val++;
    System.out.println(id + " " +
```

```
        Thread.currentThread().getName() + " " + val);
    return val;
    }
}
```

call() 完全独立于所有其他 CountingTasks 生成其结果，这意味着没有可变的共享状态

ExecutorService 允许你使用 **invokeAll()** 启动集合中的每个 Callable:

```
// concurrent/CachedThreadPool3.java
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
public class CachedThreadPool3 {
    public static Integer extractResult(Future<Integer> f) {
        try {
            return f.get();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
    public static void main(String[] args)throws InterruptedException {
    ExecutorService exec =
    Executors.newCachedThreadPool();
    List<CountingTask> tasks =
        IntStream.range(0, 10)
            .mapToObj(CountingTask::new)
            .collect(Collectors.toList());
        List<Future<Integer>> futures =
            exec.invokeAll(tasks);
        Integer sum = futures.stream()
            .map(CachedThreadPool3::extractResult)
            .reduce(0, Integer::sum);
        System.out.println("sum = " + sum);
        exec.shutdown();
    }
}
```

输出结果:

```
1 pool-1-thread-2 100
0 pool-1-thread-1 100
4 pool-1-thread-5 100
5 pool-1-thread-6 100
8 pool-1-thread-9 100
9 pool-1-thread-10 100
2 pool-1-thread-3 100
3 pool-1-thread-4 100
6 pool-1-thread-7 100
7 pool-1-thread-8 100
sum = 1000
```

只有在所有任务完成后，**invokeAll()** 才会返回一个 **Future** 列表，每个任务一个 **Future**。**Future** 是 Java 5 中引入的机制，允许你提交任务而无需等待它完成。在这里，我们使用 **ExecutorService.submit()**:

```
// concurrent/Futures.java
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
public class Futures {
    public static void main(String[] args)throws InterruptedException,
        ExecutionException {
    ExecutorService exec
        =Executors.newSingleThreadExecutor();
    Future<Integer> f =
        exec.submit(newCountingTask(99));
    System.out.println(f.get()); // [1]
    exec.shutdown();
    }
}
```

输出结果:

```
99 pool-1-thread-1 100
100
```

- 1 当你的任务尚未完成的 **Future** 上调用 **get()** 时，调用会阻塞（等待）直到结果可用。

但这意味着，在 **CachedThreadPool3.java** 中，**Future** 似乎是多余的，因为 **invokeAll()** 甚至在所有任务完成之前都不会返回。但是，这里的 Future 并不用于延迟结果，而是用于捕获任何可能发生的异常。

还要注意在 **CachedThreadPool3.java.get()** 中抛出异常，因此 **extractResult()** 在 Stream 中执行此提取。

因为当你调用 **get()** 时，**Future** 会阻塞，所以它只能解决等待任务完成的问题。最终，**Futures** 被认为是一种无效的解决方案，现在不鼓励，支持 Java 8 的 **CompletableFuture**，我们将在本章后面探讨。当然，你仍会在遗留库中遇到 Futures

我们可以使用并行 Stream 以更简单，更优雅的方式解决这个问题:

```
// concurrent/CountingStream.java
// {VisuallyInspectOutput}
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
public class CountingStream {
    public static void main(String[] args) {
        System.out.println(
            IntStream.range(0, 10)
                .parallel()
```

```
                    .mapToObj(CountingTask::new)
                    .map(ct -> ct.call())
                    .reduce(0, Integer::sum));
    }
}
```

输出结果:

```
1 ForkJoinPool.commonPool-worker-3 100
8 ForkJoinPool.commonPool-worker-2 100
0 ForkJoinPool.commonPool-worker-6 100
2 ForkJoinPool.commonPool-worker-1 100
4 ForkJoinPool.commonPool-worker-5 100
9 ForkJoinPool.commonPool-worker-7 100
6 main 100
7 ForkJoinPool.commonPool-worker-4 100
5 ForkJoinPool.commonPool-worker-2 100
3 ForkJoinPool.commonPool-worker-3 100
1000
```

这不仅更容易理解，我们需要做的就是将 **parallel()** 插入到其他顺序操作中，然后一切都在同时运行。

- Lambda 和方法引用作为任务

使用 lambdas 和方法引用，你不仅限于使用 **Runnables** 和 **Callables**。因为 Java 8 通过匹配签名来支持 lambda 和方法引用（即，它支持结构一致性），所以我们可以将 notRunnables 或 Callables 的参数传递给 ExecutorService:

使用 lambdas 和方法引用，你不仅限于使用 **Runnables** 和 **Callables**。因为 Java 8 通过匹配签名来支持 lambda 和方法引用（即，它支持结构一致性），所以我们可以将不是 **Runnables** 或 **Callables** 的参数传递给 **ExecutorService**:

```
// concurrent/LambdasAndMethodReferences.java
import java.util.concurrent.*;
class NotRunnable {
    public void go() {
        System.out.println("NotRunnable");
    }
}
class NotCallable {
    public Integer get() {
        System.out.println("NotCallable");
        return 1;
    }
}
public class LambdasAndMethodReferences {
    public static void main(String[] args)throws InterruptedException {
    ExecutorService exec =
        Executors.newCachedThreadPool();
```

```
    exec.submit(() -> System.out.println("Lambda1"));
    exec.submit(newNotRunnable()::go);
    exec.submit(() -> {
        System.out.println("Lambda2");
        return 1;
    });
    exec.submit(newNotCallable()::get);
    exec.shutdown();
    }
}
```

输出结果:

```
Lambda1
NotCallable
NotRunnable
Lambda2
```

这里，前两个 **submit()** 调用可以改为调用 **execute()**。所有 **submit()** 调用都返回 **Futures**，你可以在后两次调用的情况下提取结果。

25.7 终止耗时任务

并发程序通常使用长时间运行的任务。可调用任务在完成时返回值; 虽然这给它一个有限的寿命，但仍然可能很长。可运行的任务有时被设置为永远运行的后台进程。你经常需要一种方法在正常完成之前停止 **Runnable** 和 **Callable** 任务，例如当你关闭程序时。

最初的 Java 设计提供了中断运行任务的机制（为了向后兼容，仍然存在）; 中断机制包括阻塞问题。中断任务既乱又复杂，因为你必须了解可能发生中断的所有可能状态，以及可能导致的数据丢失。使用中断被视为反对模式，但我们仍然被迫接受。

InterruptedException，因为设计的向后兼容性残留。

任务终止的最佳方法是设置任务周期性检查的标志。然后任务可以通过自己的 shutdown 进程并正常终止。不是在任务中随机关闭线程，而是要求任务在到达了一个较好时自行终止。这总是产生比中断更好的结果，以及更容易理解的更合理的代码。

以这种方式终止任务听起来很简单：设置任务可以看到的 **boolean** flag。编写任务，以便定期检查标志并执行正常终止。这实际上就是你所做的，但是有一个复杂的问题：我们的旧克星，共同的可变状态。如果该标志可以被另一个任务操纵，则存在碰撞可能性。

在研究 Java 文献时，你会发现很多解决这个问题的方法，经常使用 **volatile** 关键字。我们将使用更简单的技术并避免所有易变的参数，这些都在附录：低级并发中有所涉及。

Java 5 引入了 **Atomic** 类，它提供了一组可以使用的类型，而不必担心并发问题。我们将添加 **AtomicBoolean** 标志，告诉任务清理自己并退出。

```java
// concurrent/QuittableTask.java
import java.util.concurrent.atomic.AtomicBoolean;import onjava.Nap;
public class QuittableTask implements Runnable {
    final int id;
    public QuittableTask(int id) {
        this.id = id;
    }
    private AtomicBoolean running =
        new AtomicBoolean(true);
    public void quit() {
        running.set(false);
    }
    @Override
    public void run() {
        while(running.get())         // [1]
            new Nap(0.1);
        System.out.print(id + " ");  // [2]
    }
}
```

虽然多个任务可以在同一个实例上成功调用 **quit()**，但是 **AtomicBoolean** 可以防止多个任务同时实际修改 **running**，从而使 **quit()** 方法成为线程安全的。

-
-

需要 **running AtomicBoolean** 证明编写 Java program 并发时最基本的困难之一是，如果 **running** 是一个普通的布尔值，你可能无法在执行程序中看到问题。实际上，在这个例子中，你可能永远不会有任何问题 - 但是代码仍然是不安全的。编写表明该问题的测试可能很困难或不可能。因此，你没有任何反馈来告诉你已经做错了。通常，你编写线程安全代码的唯一方法就是通过了解事情可能出错的所有细微之处。

作为测试，我们将启动很多 QuittableTasks 然后关闭它们。尝试使用较大的 COUNT 值

```java
// concurrent/QuittingTasks.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.Nap;
public class QuittingTasks {
    public static final int COUNT = 150;
    public static void main(String[] args) {
    ExecutorService es =
    Executors.newCachedThreadPool();
    List<QuittableTask> tasks =
    IntStream.range(1, COUNT)
        .mapToObj(QuittableTask::new)
        .peek(qt -> es.execute(qt))
        .collect(Collectors.toList());
    new Nap(1);
```

```
    tasks.forEach(QuittableTask::quit);    es.shutdown();
    }
}
```

输出结果:

```
24 27 31 8 11 7 19 12 16 4 23 3 28 32 15 20 63 60 68 6764 39 47 52 51
    55 40 43 48 59 44 56 36 35 71 72 83 10396 92 88 99 100 87 91 79 75
    84 76 115 108 112 104 107111 95 80 147 120 127 119 123 144 143 116
    132 124 128
136 131 135 139 148 140 2 126 6 5 1 18 129 17 14 13 2122 9 10 30 33 58
    37 125 26 34 133 145 78 137 141 138 6274 142 86 65 73 146 70 42 149
    121 110 134 105 82 117106 113 122 45 114 118 38 50 29 90 101 89 57
    53 94 4161 66 130 69 77 81 85 93 25 102 54 109 98 49 46 97
```

我使用 **peek()** 将 **QuittableTasks** 传递给 **ExecutorService**，然后将这些任务收集到 **List.main()** 中，只要任何任务仍在运行，就会阻止程序退出。即使为每个任务按顺序调用 quit() 方法，任务也不会按照它们创建的顺序关闭。独立运行的任务不会确定性地响应信号。

25.8 CompletableFuture 类

作为介绍，这里是使用 CompletableFutures 在 QuittingTasks.java 中:

```java
// concurrent/QuittingCompletable.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.Nap;
public class QuittingCompletable {
    public static void main(String[] args) {
    List<QuittableTask> tasks =
        IntStream.range(1, QuittingTasks.COUNT)
            .mapToObj(QuittableTask::new)
            .collect(Collectors.toList());
        List<CompletableFuture<Void>> cfutures =
        tasks.stream()
            .map(CompletableFuture::runAsync)
            .collect(Collectors.toList());
        new Nap(1);
        tasks.forEach(QuittableTask::quit);
        cfutures.forEach(CompletableFuture::join);
    }
}
```

输出结果:

```
7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 2526 27 28 29 30 31
    32 33 34 6 35 4 38 39 40 41 42 43 4445 46 47 48 49 50 51 52 53 54 55
```

```
 56 57 58 59 60 61 6263 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78
 79 8081 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 9899 100
101 102 103 104 105 106 107 108 109 110 111 1121 113 114 116 117 118
 119 120 121 122 123 124 125 126127 128 129 130 131 132 133 134 135
136 137 138 139 140141 142 143 144 145 146 147 148 149 5 115 37 36 2
 3
```

任务是一个 **List** ，就像在 **QuittingTasks.java** 中一样，但是在这个例子中，没有 **peek()** 将每个 **QuittableTask** 提交给 **ExecutorService**。相反，在创建 cfutures 期间，每个任务都交给 **CompletableFuture::runAsync**。这执行 **VerifyTask.run()** 并返回 **CompletableFuture** 。因为 **run()** 不返回任何内容，所以在这种情况下我只使用 **CompletableFuture** 调用 **join()** 来等待它完成。

在此示例中需要注意的重要事项是，运行任务不需要 **ExecutorService**。这由 **CompletableFuture** 管理（尽管有提供自己的 **ExecutorService** 的选项）。你也不需要调用 **shutdown()**; 事实上，除非你像我这样明确地调用 **join()**，程序将尽快退出，而不必等待任务完成。

这个例子只是一个起点。你很快就会看到 ComplempleFutures 能够做得更多。

25.8.1 基本用法

这是一个带有静态方法 **work()** 的类，它对该类的对象执行某些工作：

```java
// concurrent/Machina.java
import onjava.Nap;
public class Machina {
    public enum State {
        START, ONE, TWO, THREE, END;
        State step() {
            if(equals(END))
            return END;
            return values()[ordinal() + 1];
        }
    }
    private State state = State.START;
    private final int id;
    public Machina(int id) {
        this.id = id;
    }
    public static Machina work(Machina m) {
        if(!m.state.equals(State.END)){
            new Nap(0.1);
            m.state = m.state.step();
        }
        System.out.println(m);return m;
    }
    @Override
    public StringtoString() {
```

```
        return"Machina" + id + ": " +      (state.equals(State.END)? "
            complete" : state);
    }
}
```

这是一个有限状态机，一个微不足道的机器，因为它没有分支……它只是从头到尾遍历一条路径。**work()** 方法将机器从一个状态移动到下一个状态，并且需要 100 毫秒才能完成“工作”。

我们可以用 **CompletableFuture** 做的一件事是使用 **completedFuture()** 将它包装在感兴趣的对象中

```
// concurrent/CompletedMachina.java
import java.util.concurrent.*;
public class CompletedMachina {
    public static void main(String[] args) {
        CompletableFuture<Machina> cf =
        CompletableFuture.completedFuture(
            new Machina(0));
        try {
            Machina m = cf.get();  // Doesn't block
        } catch(InterruptedException |
            ExecutionException e) {
        throw new RuntimeException(e);
        }
    }
}
```

completedFuture() 创建一个“已经完成”的 **CompletableFuture**。对这样一个未来做的唯一有用的事情是 **get()** 里面的对象，所以这看起来似乎没有用。注意 **CompletableFuture** 被输入到它包含的对象。这个很重要。

通常，**get()** 在等待结果时阻塞调用线程。此块可以通过 **InterruptedException** 或 **ExecutionException** 中断。在这种情况下，阻止永远不会发生，因为 CompletableFutureis 已经完成，所以答案立即可用。

当我们将 **handle()** 包装在 **CompletableFuture** 中时，我们发现我们可以在 **CompletableFuture** 上添加操作来处理所包含的对象，事情变得更加有趣：

```
// concurrent/CompletableApply.java
import java.util.concurrent.*;
public class CompletableApply {
    public static void main(String[] args) {
        CompletableFuture<Machina> cf =
        CompletableFuture.completedFuture(
            new Machina(0));
        CompletableFuture<Machina> cf2 =
            cf.thenApply(Machina::work);
        CompletableFuture<Machina> cf3 =
            cf2.thenApply(Machina::work);
```

```
        CompletableFuture<Machina> cf4 =
            cf3.thenApply(Machina::work);
        CompletableFuture<Machina> cf5 =
            cf4.thenApply(Machina::work);
    }
}
```

输出结果:

```
Machina0: ONE
Machina0: TWO
Machina0: THREE
Machina0: complete
```

thenApply() 应用一个接受输入并产生输出的函数。在这种情况下，**work()** 函数产生与它相同的类型，因此每个得到的 **CompletableFuture** 仍然被输入为 **Machina**，但是（类似于 **Streams** 中的 **map()**）**Function** 也可以返回不同的类型，这将反映在返回类型

你可以在此处看到有关 **CompletableFutures** 的重要信息：它们会在你执行操作时自动解包并重新包装它们所携带的对象。这样你就不会陷入麻烦的细节，这使得编写和理解代码变得更加简单。

我们可以消除中间变量并将操作链接在一起，就像我们使用 Streams 一样:

```
// concurrent/CompletableApplyChained.javaimport java.util.concurrent
    .*;
import onjava.Timer;
public class CompletableApplyChained {
    public static void main(String[] args) {
        Timer timer = new Timer();
        CompletableFuture<Machina> cf =
        CompletableFuture.completedFuture(
            new Machina(0))
                  .thenApply(Machina::work)
                  .thenApply(Machina::work)
                  .thenApply(Machina::work)
                  .thenApply(Machina::work);
        System.out.println(timer.duration());
    }
}
```

输出结果:

```
Machina0: ONE
Machina0: TWO
Machina0: THREE
Machina0: complete
514
```

在这里，我们还添加了一个 **Timer**，它向我们展示每一步增加 100 毫秒，还有一些额外的开销。**CompletableFutures** 的一个重要好处是它们鼓励使用私有子类原则（不分享任何东西）。默认情况

下，使用 **thenApply()** 来应用一个不与任何人通信的函数 - 它只需要一个参数并返回一个结果。这是函数式编程的基础，并且它在并发性方面非常有效[5]。并行流和 ComplempleFutures 旨在支持这些原则。只要你不决定共享数据（共享非常容易，甚至意外）你可以编写相对安全的并发程序。

回调 **thenApply()** 开始一个操作，在这种情况下，在完成所有任务之前，不会完成 **e CompletableFuture** 的创建。虽然这有时很有用，但是启动所有任务通常更有价值，这样就可以运行时继续前进并执行其他操作。我们通过在操作结束时添加 Async 来实现此目的：

```
// concurrent/CompletableApplyAsync.java
import java.util.concurrent.*;
import onjava.*;
public class CompletableApplyAsync {
    public static void main(String[] args) {
        Timer timer = new Timer();
        CompletableFuture<Machina> cf =
            CompletableFuture.completedFuture(
                new Machina(0))
                .thenApplyAsync(Machina::work)
                .thenApplyAsync(Machina::work)
                .thenApplyAsync(Machina::work)
                .thenApplyAsync(Machina::work);
            System.out.println(timer.duration());
            System.out.println(cf.join());
            System.out.println(timer.duration());
    }
}
```

输出结果：

```
116
Machina0: ONE
Machina0: TWO
Machina0:THREE
Machina0: complete
Machina0: complete
552
```

同步调用 (我们通常使用得那种) 意味着“当你完成工作时，返回”，而异步调用以意味着“立刻返回但是继续后台工作。”正如你所看到的，**cf** 的创建现在发生得跟快。每次调用 **thenApplyAsync()** 都会立刻返回，因此可以进行下一次调用，整个链接序列的完成速度比以前快得快。

事实上，如果没有回调 **cf.join() t** 方法，程序会在完成其工作之前退出（尝试取出该行）对 **join()** 阻止了 **main()** 进程的进行，直到 cf 操作完成，我们可以看到大部分时间的确在哪里度过。

这种“立即返回”的异步能力需要 **CompletableFuture** 库进行一些秘密工作。特别是，它必须将你需要的操作链存储为一组回调。当第一个后台操作完成并返回时，第二个后台操作必须获取生成的 **Machina** 并开始工作，当完成后，下一个操作将接管，等等。但是没有我们普通的函数调用序列，通过

[5]这在即将产生的 C++ 的标准中得到了补救。

程序调用栈控制，这个顺序会丢失，所以它使用回调 - 一个函数地址表来存储。

幸运的是，你需要了解有关回调的所有信息。程序员将你手工造成的混乱称为“回调地狱”。通过异步调用，**CompletableFuture** 为你管理所有回调。除非你知道关于你的系统有什么特定的改变，否则你可能想要使用异步调用。

- 其他操作当你查看 **CompletableFuture** 的 Javadoc 时,你会看到它有很多方法,但这个方法的大部分来自不同操作的变体。例如, 有 **thenApply()**, **thenApplyAsync()** 和 **thenApplyAsync()** 的第二种形式，它接受运行任务的 **Executor**（在本书中我们忽略了 **Executor** 选项）。

这是一个显示所有“基本”操作的示例，它们不涉及组合两个 CompletableFutures 或异常（我们将在稍后查看）。首先，我们将重复使用两个实用程序以提供简洁和方便:

```
// concurrent/CompletableUtilities.java
package onjava; import java.util.concurrent.*;
public class CompletableUtilities {
    // Get and show value stored in a CF:
    public static void showr(CompletableFuture<?> c) {
        try {
            System.out.println(c.get());
        } catch(InterruptedException
                | ExecutionException e) {
            throw new RuntimeException(e);
        }
    }
    // For CF operations that have no value:
    public static void voidr(CompletableFuture<Void> c) {
        try {
            c.get(); // Returns void
        } catch(InterruptedException
                | ExecutionException e) {
            throw new RuntimeException(e);
        }
    }
}
```

showr() 在 **CompletableFuture** 上调用 **get()** 并显示结果，捕获两个可能的异常。**voidr()** 是 **CompletableFuture** 的 **showr()** 版本，即 **CompletableFutures**，仅在任务完成或失败时显示。

为简单起见，以下 **CompletableFutures** 只包装整数。**cfi()** 是一个方便的方法，它在完成的 **CompletableFuture** 中包装一个 **int**:

```
// concurrent/CompletableOperations.java
import java.util.concurrent.*;
import static onjava.CompletableUtilities.*;
public class CompletableOperations {
    static CompletableFuture<Integer> cfi(int i) {
        return CompletableFuture.completedFuture( Integer.valueOf(i));
    }
    public static void main(String[] args) {
```

```
        showr(cfi(1)); // Basic test
        voidr(cfi(2).runAsync(() ->
            System.out.println("runAsync")));
        voidr(cfi(3).thenRunAsync(() ->
            System.out.println("thenRunAsync")));
        voidr(CompletableFuture.runAsync(() ->
            System.out.println("runAsync is static")));
        showr(CompletableFuture.supplyAsync(() -> 99));
        voidr(cfi(4).thenAcceptAsync(i ->
            System.out.println("thenAcceptAsync: " + i)));
        showr(cfi(5).thenApplyAsync(i -> i + 42));
        showr(cfi(6).thenComposeAsync(i -> cfi(i + 99)));
        CompletableFuture<Integer> c = cfi(7);
        c.obtrudeValue(111);
        showr(c);
        showr(cfi(8).toCompletableFuture());
        c = new CompletableFuture<>();
        c.complete(9);
        showr(c);
        c = new CompletableFuture<>();
        c.cancel(true);
        System.out.println("cancelled: " + c.isCancelled());
        System.out.println("completed exceptionally: " +
            c.isCompletedExceptionally());
        System.out.println("done: " + c.isDone());
        System.out.println(c);
        c = new CompletableFuture<>();
        System.out.println(c.getNow(777));
        c = new CompletableFuture<>();
        c.thenApplyAsync(i -> i + 42)
            .thenApplyAsync(i -> i * 12);
        System.out.println("dependents: " + c.getNumberOfDependents());
        c.thenApplyAsync(i -> i / 2);
        System.out.println("dependents: " + c.getNumberOfDependents());
    }
}
```

输出结果:

```
1
runAsync
thenRunAsync
runAsync is static
99
thenAcceptAsync: 4
47
105
111
8
9
cancelled: true
```

```
completed exceptionally: true
done: true
java.util.concurrent.CompletableFuture@6d311334[Complet ed
    exceptionally]
777
dependents: 1
dependents: 2
```

main() 包含一系列可由其 **int** 值引用的测试。**cfi(1)** 演示了 **showr()** 正常工作。**cfi(2)** 是调用 **runAsync()** 的示例。由于 **Runnable** 不产生返回值，因此结果是 **CompletableFuture** ，因此使用 **voidr()**。注意使用 **cfi(3),thenRunAsync()** 似乎与 **runAsync()** 一致，差异显示在后续的测试中：**runAsync()** 是一个静态方法，所以你不会像 **cfi(2)** 一样调用它。相反你可以在 **QuittingCompletable.java** 中使用它。后续测试中 **supplyAsync()** 也是静态方法，但是需要一个 **Supplier** 而不是 **Runnable** 并产生一个 **CompletableFuture** 来代替 **CompletableFuture**。含有"then"的方法将进一步的操作应用于现有的 **CompletableFuture** 。与 **thenRunAsync()** 不同的是，将 **cfi(4)**，**cfi(5)** 和 **cfi(6)** 的 " then" 方法作为未包装的 **Integer** 的参数。如你通过使用 **voidr()** 所见，然后 **AcceptAsync()** 接受了一个 **Consumer**，因此不会产生结果。**thenApplyAsync()** 接受一个 **Function** 并因此产生一个结果（该结果的类型可以不同于其参数）。**thenComposeAsync()** 与 **thenApplyAsync()** 非常相似，不同之处在于其 Function 必须产生已经包装在 **CompletableFuture** 中的结果。**cfi(7)** 示例演示了 **obtrudeValue()**，它强制将值作为结果。**cfi(8)** 使用 **toCompletableFuture()** 从 **CompletionStage** 生成 **CompletableFuture**。**c.complete(9)** 显示了如何通过给它一个结果来完成一个任务（**future**）（与 **obtrudeValue()** 相对，后者可能会迫使其结果替换该结果）。如果你调用 **CompletableFuture** 中的 **cancel()** 方法，它也会完成并且是非常好的完成。如果任务（**future**）未完成，则 **getNow()** 方法返回 **CompletableFuture** 的完成值，或者返回 **getNow()** 的替换参数。最后，我们看一下依赖 (dependents) 的概念。如果我们将两个 **thenApplyAsync()** 调用链接到 **CompletableFuture** 上，则依赖项的数量仍为 1。但是，如果我们将另一个 **thenApplyAsync()** 直接附加到 **c**，则现在有两个依赖项：两个链和另一个链。这表明你可以拥有一个 **CompletionStage**，当它完成时，可以根据其结果派生多个新任务。

25.8.2 结合 CompletableFutures

第二类 **CompletableFuture** 方法采用两个 **CompletableFuture** 并以各种方式将它们组合在一起。一个 **CompletableFuture** 通常会先于另一个完成，就好像两者都在比赛中一样。这些方法使你可以以不同的方式处理结果。为了对此进行测试，我们将创建一个任务，该任务将完成的时间作为其参数之一，因此我们可以控制。**CompletableFuture** 首先完成：

```
// concurrent/Workable.java
import java.util.concurrent.*;
import onjava.Nap;
public class Workable {
```

```
    String id;
    final double duration;
    public Workable(String id, double duration) {
        this.id = id;
        this.duration = duration;
    }
    @Override
    public String toString() {
        return "Workable[" + id + "]";
    }
    public static Workable work(Workable tt) {
        new Nap(tt.duration); // Seconds
        tt.id = tt.id + "W";
        System.out.println(tt);
        return tt;
    }
    public static CompletableFuture<Workable> make(String id, double
        duration) {
        return CompletableFuture.completedFuture( new Workable(id,
            duration)) .thenApplyAsync(Workable::work);
    }
}
```

在 **make()** 中，**work()** 方法应用于 **CompletableFuture.work()** 需要持续时间才能完成，然后将字母 W 附加到 id 上以指示工作已完成。现在，我们可以创建多个竞争的 **CompletableFuture**，并使用 **CompletableFuture** 库：

```
// concurrent/DualCompletableOperations.java
import java.util.concurrent.*;
import static onjava.CompletableUtilities.*;
public class DualCompletableOperations {
    static CompletableFuture<Workable> cfA, cfB;
    static void init() {
        cfA = Workable.make("A", 0.15);
        cfB = Workable.make("B", 0.10);// Always wins
    }
    static void join() {
        cfA.join();
        cfB.join();
        System.out.println("*****************");
    }
    public static void main(String[] args) {
        init();
        voidr(cfA.runAfterEitherAsync(cfB, () -> System.out.println("
            runAfterEither")));
        join();
        init();
        voidr(cfA.runAfterBothAsync(cfB, () -> System.out.println("
            runAfterBoth")));
        join();
```

```
        init();
        showr(cfA.applyToEitherAsync(cfB, w -> {
            System.out.println("applyToEither: " + w);
            return w;
        }));
        join();
        init();
        voidr(cfA.acceptEitherAsync(cfB, w -> {
            System.out.println("acceptEither: " + w);
        }));
        join();
        init();
        voidr(cfA.thenAcceptBothAsync(cfB, (w1, w2) -> { System.out.
            println("thenAcceptBoth: " + w1 + ", " + w2);
        }));
        join();
        init();
        showr(cfA.thenCombineAsync(cfB, (w1, w2) -> {
            System.out.println("thenCombine: " + w1 + ", " + w2);
            return w1;
        }));
        join();
        init();
        CompletableFuture<Workable>
            cfC = Workable.make("C", 0.08),
            cfD = Workable.make("D", 0.09);
        CompletableFuture.anyOf(cfA, cfB, cfC, cfD)
        .thenRunAsync(() -> System.out.println("anyOf"));
        join();
        init();
        cfC = Workable.make("C", 0.08);
        cfD = Workable.make("D", 0.09);
        CompletableFuture.allOf(cfA, cfB, cfC, cfD)
        .thenRunAsync(() -> System.out.println("allOf"));
        join();
    }
}
```

输出结果:

```
Workable[BW]
runAfterEither
Workable[AW]
*****************
Workable[BW]
Workable[AW]
runAfterBoth
*****************
Workable[BW]
applyToEither: Workable[BW]
Workable[BW]
```

```
Workable[AW]
*****************
Workable[BW]
acceptEither: Workable[BW]
Workable[AW]
*****************
Workable[BW]
Workable[AW]
thenAcceptBoth: Workable[AW], Workable[BW]
****************
 Workable[BW]
 Workable[AW]
 thenCombine: Workable[AW], Workable[BW]
 Workable[AW]
 *****************
 Workable[CW]
 anyOf
 Workable[DW]
 Workable[BW]
 Workable[AW]
 *****************
 Workable[CW]
 Workable[DW]
 Workable[BW]
 Workable[AW]
 *****************
 allOf
```

为了便于访问，**cfA** 和 **cfB** 是静态的。**init()** 总是使用较短的延迟（因此总是“获胜”）使用 “ B” 初始化两者。**join()** 是在这两种方法上调用 **join()** 并显示边框的另一种便捷方法。所有这些“双重”方法都以一个 **CompletableFuture** 作为调用该方法的对象，第二个 **CompletableFuture** 作为第一个参数，然后是要执行的操作。通过使用 **Shower()** 和 **void()**，你可以看到“运行”和“接受”是终端操作，而“应用”和“组合”产生了新的承载载荷的 **CompletableFutures**。

方法的名称是不言自明的，你可以通过查看输出来验证这一点。一个特别有趣的方法是 CombineAsync()，它等待两个 **CompletableFuture** 完成，然后将它们都交给 BiFunction，然后 BiFunction 可以将结果加入到所得 **CompletableFuture** 的有效负载中。

25.8.3 模拟

作为一个示例，说明如何使用 **CompletableFutures** 将一系列操作组合在一起，让我们模拟制作蛋糕的过程。在第一个阶段中，我们准备并将成分混合成面糊：

```java
// concurrent/Batter.java
import java.util.concurrent.*;
import onjava.Nap;
public class Batter {
```

```
    static class Eggs {}
    static class Milk {}
    static class Sugar {}
    static class Flour {}
    static <T> T prepare(T ingredient) {
        new Nap(0.1);
        return ingredient;
    }
    static <T> CompletableFuture<T> prep(T ingredient) {
        return CompletableFuture
                .completedFuture(ingredient)
                .thenApplyAsync(Batter::prepare);
    }
    public static CompletableFuture<Batter> mix() {
        CompletableFuture<Eggs> eggs = prep(new Eggs());
           CompletableFuture<Milk> milk = prep(new Milk());
           CompletableFuture<Sugar> sugar = prep(new Sugar());
           CompletableFuture<Flour> flour = prep(new Flour());
           CompletableFuture.allOf(eggs, milk, sugar, flour)
                        .join();
        new Nap(0.1); // Mixing time
        return CompletableFuture.completedFuture(new Batter());
    }
}
```

每种成分都需要一些时间来准备。**allOf()** 等待所有配料准备就绪，然后需要更多时间将其混合到面糊中。

接下来，我们将单批面糊放入四个锅中进行烘烤。产品作为 **CompletableFutures** 流返回：

```
// concurrent/Baked.java
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.Nap;
public class Baked {
    static class Pan {}
    static Pan pan(Batter b) {
        new Nap(0.1);
        return new Pan();
    }
    static Baked heat(Pan p) {
        new Nap(0.1);
        return new Baked();
    }
    static CompletableFuture<Baked> bake(CompletableFuture<Batter> cfb)
       {
        return cfb.thenApplyAsync(Baked::pan)
                    .thenApplyAsync(Baked::heat);
    }
    public static Stream<CompletableFuture<Baked>> batch() {
        CompletableFuture<Batter> batter = Batter.mix();
```

```
        return Stream.of(bake(batter), bake(batter), bake(batter), bake
            (batter));
    }
}
```

最后，我们创建了一批糖，并用它对蛋糕进行糖化：

```
// concurrent/FrostedCake.java
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.Nap;
final class Frosting {
    private Frosting() {}
    static CompletableFuture<Frosting> make() {
        new Nap(0.1);
        return CompletableFuture.completedFuture(new Frosting());
    }
}
public class FrostedCake {
    public FrostedCake(Baked baked, Frosting frosting) {
        new Nap(0.1);
    }
    @Override
    public String toString() {
        return "FrostedCake";
    }
    public static void main(String[] args) {
        Baked.batch()
            .forEach(baked -> baked.thenCombineAsync(Frosting.make(), (
                cake, frosting) -> new FrostedCake(cake, frosting)) .
                thenAcceptAsync(System.out::println)
            .join());
    }
}
```

一旦你对背后的想法感到满意。**CompletableFutures** 它们相对易于使用。

25.8.4 例外情况

与 **CompletableFuture** 在处理链中包装对象的方式相同，它还可以缓冲异常。这些不会在处理过程中显示给调用者，而只会在你尝试提取结果时显示。为了展示它们是如何工作的，我们将从创建一个在某些情况下引发异常的类开始：

```
// concurrent/Breakable.java
import java.util.concurrent.*;
public class Breakable {
    String id;
    private int failcount;
    public Breakable(String id, int failcount) {
```

```
        this.id = id;
        this.failcount = failcount;
    }
    @Override
    public String toString() {
        return "Breakable_" + id + " [" + failcount + "]";
    }
    public static Breakable work(Breakable b) {
        if(--b.failcount == 0) {
            System.out.println( "Throwing Exception for " + b.id + "");
            throw new RuntimeException( "Breakable_" + b.id + " failed"
                );
        }
        System.out.println(b);
        return b;
    }
}
```

failcount 为正时，每次将对象传递给 **work()** 方法可减少 **failcount**。当它为零时，**work()** 会引发异常。如果你给它的 **failcount** 为零，则它永远不会引发异常。请注意，它报告在抛出异常时抛出异常。在下面的 **test()** 方法中，**work()** 多次应用于 **Breakable**，因此，如果 **failcount** 在范围内，则会引发异常。但是，在测试 **A** 到 **E** 中，你可以从输出中看到抛出了异常，但是它们从未出现：

```
// concurrent/CompletableExceptions.java
import java.util.concurrent.*;
public class CompletableExceptions {
    static CompletableFuture<Breakable> test(String id, int failcount)
        {
        return
            CompletableFuture.completedFuture(
                new Breakable(id, failcount))
                .thenApply(Breakable::work)
                .thenApply(Breakable::work)
                .thenApply(Breakable::work)
                .thenApply(Breakable::work);
    }
    public static void main(String[] args) {
        // Exceptions don't appear ...
        test("A", 1);
        test("B", 2);
        test("C", 3);
        test("D", 4);
        test("E", 5);
        // ... until you try to fetch the value:
        try {
            test("F", 2).get(); // or join()
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
```

```
        // Test for exceptions:
        System.out.println(
            test("G", 2).isCompletedExceptionally());
        // Counts as "done":
        System.out.println(test("H", 2).isDone());
        // Force an exception:
        CompletableFuture<Integer> cfi =
            new CompletableFuture<>();
        System.out.println("done? " + cfi.isDone());
        cfi.completeExceptionally( new RuntimeException("forced"));
        try {
            cfi.get();
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

输出结果:

```
Throwing Exception for A
Breakable_B [1]
Throwing Exception for B
Breakable_C [2]
Breakable_C [1]
Throwing Exception for C
Breakable_D [3]
Breakable_D [2]
Breakable_D [1]
Throwing Exception for D
Breakable_E [4]
Breakable_E [3]
Breakable_E [2]
Breakable_E [1]
Breakable_F [1]
Throwing Exception for F
java.lang.RuntimeException: Breakable_F failed
Breakable_G [1]
Throwing Exception for G
true
Breakable_H [1]
Throwing Exception for H
true
done? false
java.lang.RuntimeException: forced
```

测试 **A** 到 **E** 运行到抛出异常的地步，然后……什么都没有。只有在测试 **F** 中调用 **get()** 时，我们才能看到抛出的异常。测试 **G** 显示，你可以首先检查在处理过程中是否引发了异常，而没有引发该异常。但是，测试 H 告诉我们，无论异常成功与否，异常仍然可以被视为“完成” 代码的最后一部分显示了如何在 **CompletableFuture** 中插入异常，而不管是否存在任何故障。加入或获取结果时，我们不

会使用粗略的 try-catch，而是使用 **CompletableFuture** 提供的更复杂的机制来自动响应异常。你可以使用与所有 **CompletableFuture** 相同的表格来执行此操作：在链中插入 **CompletableFuture** 调用。有三个选项：**exclusively()**，**handle()** 和 **whenComplete()**：

```
// concurrent/CatchCompletableExceptions.java
import java.util.concurrent.*;
public class CatchCompletableExceptions {
    static void handleException(int failcount) {
        // Call the Function only if there's an
        // exception, must produce same type as came in:
        CompletableExceptions
        .test("exceptionally", failcount)
        .exceptionally((ex) -> { // Function
            if(ex == null)
                System.out.println("I don't get it yet");
            return new Breakable(ex.getMessage(), 0);
        })
        .thenAccept(str ->
            System.out.println("result: " + str));
        // Create a new result (recover):
        CompletableExceptions
            .test("handle", failcount)
            .handle((result, fail) -> { // BiFunction
                if(fail != null)
                    return "Failure recovery object";
                else
                    return result + " is good"; })
            .thenAccept(str ->
                System.out.println("result: " + str));
        // Do something but pass the same result through:
        CompletableExceptions
            .test("whenComplete", failcount)
            .whenComplete((result, fail) -> {// BiConsumer
                if(fail != null)
                    System.out.println("It failed");
                else
                    System.out.println(result + " OK");
            })
            .thenAccept(r ->
            System.out.println("result: " + r));
    }
    public static void main(String[] args) {
        System.out.println("**** Failure Mode ****");
        handleException(2);
        System.out.println("**** Success Mode ****");
        handleException(0);
    }
}
```

输出结果：

```
**** Failure Mode ****
Breakable_exceptionally [1]
Throwing Exception for exceptionally
result: Breakable_java.lang.RuntimeException:
Breakable_exceptionally failed [0]
Breakable_handle [1]
Throwing Exception for handle
result: Failure recovery object
Breakable_whenComplete [1]
Throwing Exception for whenComplete
It failed
**** Success Mode ****
Breakable_exceptionally [-1]
Breakable_exceptionally [-2]
Breakable_exceptionally [-3]
Breakable_exceptionally [-4]
result: Breakable_exceptionally [-4]
Breakable_handle [-1]
Breakable_handle [-2]
Breakable_handle [-3]
Breakable_handle [-4]
result: Breakable_handle [-4] is good
Breakable_whenComplete [-1]
Breakable_whenComplete [-2]
Breakable_whenComplete [-3]
Breakable_whenComplete [-4]
Breakable_whenComplete [-4] OK
result: Breakable_whenComplete [-4]
```

只有在有异常的情况下，**exclusively()** 参数才会运行。**Exclusively()** 的局限性在于，该函数只能返回输入的相同类型的值。**exclusively()** 通过将一个好的对象重新插入流中而恢复到可行状态。**handle()** 始终被调用，你必须检查一下 **fail** 是否为 **true** 才能查看是否发生了异常。但是 **handle()** 可以产生任何新类型，因此它使你可以执行处理，而不仅可以像 **exception()** 那样进行恢复。**whenComplete()** 就像 **handle()** 一样，你必须测试是否失败，但是该参数是使用者，并且不会修改正在传递的结果对象。

25.8.5 流异常

通过修改 **CompletableExceptions.java**，看看 **CompletableFuture** 异常与 **Streams** 异常有何不同：

```java
// concurrent/StreamExceptions.java
import java.util.concurrent.*;
import java.util.stream.*;
public class StreamExceptions {
    static Stream<Breakable> test(String id, int failcount) {
        return Stream.of(new Breakable(id, failcount)).
```

```
        map(Breakable::work)
        .map(Breakable::work
        .map(Breakable::work)
        .map(Breakable::work);
    }
    public static void main(String[] args) {
        // No operations are even applied ...
        test("A", 1);
        test("B", 2);
        Stream<Breakable> c = test("C", 3);
        test("D", 4);
        test("E", 5);
        // ... until there's a terminal operation:
        System.out.println("Entering try");
        try {
            c.forEach(System.out::println);// [1]
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

输出结果:

```
Entering try
Breakable_C [2]
Breakable_C [1]
Throwing Exception for C
Breakable_C failed
```

使用 **CompletableFutures**，我们看到了测试 **A** 到 **E** 的进展，但是使用 **Streams**，直到你应用了终端操作 (如1的 **forEach()**)，一切都没有开始。**CompletableFuture** 执行工作并捕获任何异常以供以后检索。比较这两者并不是一件容易的事，因为 **Stream** 没有终端操作根本无法执行任何操作，但是 **Stream** 绝对不会存储其异常。

25.8.6 检查异常

CompletableFutures 和并行 Streams 都不支持包含已检查异常的操作。相反，你必须在调用操作时处理检查到的异常，这会产生不太优雅的代码:

```
// concurrent/ThrowsChecked.java
import java.util.stream.*;
import java.util.concurrent.*;
public class ThrowsChecked {
    class Checked extends Exception {}
    static ThrowsChecked nochecked(ThrowsChecked tc) {
        return tc;
    }
```

```
    static ThrowsChecked withchecked(ThrowsChecked tc) throws Checked {
        return tc;
    }
    static void testStream() {
        Stream.of(new ThrowsChecked())
            .map(ThrowsChecked::nochecked)
            // .map(ThrowsChecked::withchecked); // [1]
            .map(tc -> {
                try {
                    return withchecked(tc);
                } catch(Checked e) {
                    throw new RuntimeException(e);
                }
            });
    }
    static void testCompletableFuture() {
        CompletableFuture .completedFuture(new ThrowsChecked())
            .thenApply(ThrowsChecked::nochecked)
            // .thenApply(ThrowsChecked::withchecked); // [2]
            .thenApply(tc -> {
                try {
                    return withchecked(tc);
                } catch(Checked e) {
                    throw new RuntimeException(e);
                }
            });
    }
}
```

如果你尝试像对 **nochecked()** 一样对 **withchecked()** 使用方法引用，则编译器会抱怨1和2。相反，你必须写出 lambda 表达式（或编写一个不会引发异常的包装器方法）。## 死锁

由于任务可能会被阻塞，因此一个任务有可能卡在等待另一个任务上，而任务又在等待另一个任务，依此类推，直到链回到第一个任务上。你会遇到一个不断循环的任务，彼此等待，没有人能动。这称为死锁[6] 如果你尝试运行某个程序并立即陷入死锁，则可以立即查找该错误。真正的问题是，当你的程序看起来运行良好，但具有隐藏潜力死锁。在这里，你可能没有任何迹象表明可能发生死锁，因此该缺陷在你的程序中是潜在的，直到它意外发生为止（通常是对客户而言（几乎肯定很难复制））。因此，通过仔细的程序设计防止死锁是开发并发系统的关键部分。埃德斯·迪克斯特拉（Essger Dijkstra）发明的“哲学家进餐”问题是经典的死锁例证。基本描述指定了五位哲学家（此处显示的示例允许任何数字）。这些哲学家将一部分时间花在思考上，一部分时间在吃饭上。他们在思考的时候并不需要任何共享资源，但是他们使用的餐具数量有限。在最初的问题描述中，器物是叉子，需要两个叉子才能从桌子中间的碗里取出意大利面。常见的版本是使用筷子。显然，每个哲学家都需要两个筷子才能吃饭。引入了一个困难：作为哲学家，他们的钱很少，所以他们只能买五根筷子（更普遍地说，筷子的数量与哲学家相同）。它们之间围绕桌子隔开。当一个哲学家想要吃饭时，该哲学家必须拿起左边和右边的筷子。如果任一侧

[6] 当两个任务能够更改其状态以使它们不会被阻止但它们从未取得任何有用的进展时，你也可以使用活动锁。

的哲学家都在使用所需的筷子，则我们的哲学家必须等待，直到必要的筷子可用为止。**StickHolder** 类通过将单个筷子保持在大小为 1 的 **BlockingQueue** 中来管理它。**BlockingQueue** 是一个设计用于在并发程序中安全使用的集合，如果你调用 take() 并且队列为空，则它将阻塞（等待）。将新元素放入队列后，将释放该块并返回该值:

```
// concurrent/StickHolder.java
import java.util.concurrent.*;
public class StickHolder {
    private static class Chopstick {}
    private Chopstick stick = new Chopstick();
    private BlockingQueue<Chopstick> holder = new ArrayBlockingQueue
        <>(1);
    public StickHolder() {
        putDown();
    }
    public void pickUp() {
        try {
            holder.take();// Blocks if unavailable
        } catch(InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
    public void putDown() {
        try {
            holder.put(stick);
        } catch(InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

为简单起见，**StickHolder** 从未真正制作过 **Chopstick**，而是在类中将其保密。如果调用 **pickUp()** 而该筷子不可用，则 **pickUp()** 会阻塞，直到另一位调用 **putDown()** 的哲学家返回了该摇杆。请注意，此类中的所有线程安全性都是通过 **BlockingQueue** 实现的。

每个哲学家都是一个任务，尝试将左右两把筷子都拿起，使其可以进食，然后使用 **putDown()** 释放这些筷子:

```
// concurrent/Philosopher.java
public class Philosopher implements Runnable {
    private final int seat;
    private final StickHolder left, right;
    public Philosopher(int seat, StickHolder left, StickHolder right) {
        this.seat = seat;
        this.left = left;
        this.right = right;
    }
    @Override
    public String toString() {
```

```
        return "P" + seat;
    }
    @Override
    public void run() {
        while(true) {
            // System.out.println("Thinking");
            // [1] right.pickUp();
            left.pickUp();
            System.out.println(this + " eating");
            right.putDown();
            left.putDown();
        }
    }
}
```

没有两个哲学家可以同时成功调用 take() 同一只筷子。另外，如果一个哲学家已经拿过筷子，那么下一个试图拿起同一根筷子的哲学家将阻塞，等待其被释放。结果是一个看似无辜的程序陷入了死锁。我在这里使用数组而不是集合，只是因为结果语法更简洁：

```
// concurrent/DiningPhilosophers.java
// Hidden deadlock
// {ExcludeFromGradle} Gradle has trouble
import java.util.*;
import java.util.concurrent.*;
import onjava.Nap;
public class DiningPhilosophers {
    private StickHolder[] sticks;
    private Philosopher[] philosophers;
    public DiningPhilosophers(int n) {
        sticks = new StickHolder[n];
        Arrays.setAll(sticks, i -> new StickHolder());
        philosophers = new Philosopher[n];
        Arrays.setAll(philosophers,
            i -> new Philosopher(i, sticks[i], sticks[(i + 1) % n]));//
                [1]
        // Fix by reversing stick order for this one:
        // philosophers[1] = // [2]
        // new Philosopher(0, sticks[0], sticks[1]);
        Arrays.stream(philosophers)
            .forEach(CompletableFuture::runAsync);// [3]
    }
    public static void main(String[] args) {
        // Returns right away:
        new DiningPhilosophers(5);// [4]
        // Keeps main() from exiting:
        new Nap(3, "Shutdown");
    }
}
```

当你停止查看输出时，该程序将死锁。但是，根据你的计算机配置，你可能不会看到死锁。看来这

取决于计算机上的内核数[7]。两个核心似乎不会产生死锁，但似乎有两个以上的核心很容易产生死锁。此行为使该示例更好地说明了死锁，因为你可能正在具有两个内核的计算机上编写程序（如果确实是导致问题的原因），并且确信该程序可以正常工作，只能启动它将其安装在另一台计算机上时出现死锁。请注意，仅仅因为你不容易看到死锁，并不意味着该程序就不会在两核计算机上死锁。该程序仍然容易死锁，很少发生-可以说是最坏的情况，因为问题不容易解决。在 DiningPhilosophers 构造函数中，每个哲学家都获得一个左右 StickHolder 的引用。除最后一个哲学家外，每个哲学家都通过以下方式初始化：哲学家之间的下一双筷子。最后一位哲学家右手的筷子为零，因此圆桌会议完成了。那是因为最后一位哲学家正坐在第一个哲学家的旁边，而且他们俩都共用零筷子。1显示了以 n 为模数选择的右摇杆，将最后一个哲学家缠绕在第一个哲学家的旁边。现在，所有哲学家都可以尝试吃饭，每个哲学家都在旁边等待哲学家放下筷子。要开始在 [3] 上运行的每个 Philosopher，我调用 runAsync（），这意味着 DiningPhilosophers 构造函数立即在 [4] 处返回。没有任何东西可以阻止 main（）完成，该程序只是退出而无济于事。Nap 对象阻止 main（）退出，然后在三秒钟后强制退出（可能是）死锁的程序。在给定的配置中，哲学家几乎没有时间思考。因此，他们都在尝试吃饭时争夺筷子，而且僵局往往很快发生。你可以更改此：

1. 通过增加 [4] 的值来添加更多哲学家。
2. 在 Philosopher.java 中取消注释行1。

任一种方法都会减少死锁的可能性，这表明编写并发程序并认为它是安全的危险，因为它似乎“在我的机器上运行正常”。你可以轻松地说服自己该程序没有死锁，即使它不是。这个例子很有趣，因为它演示了程序似乎可以正确运行，同时仍然容易出现死锁。为了解决该问题，我们观察到当四个同时满足条件：

1. 互斥。任务使用的至少一种资源必须不可共享。在这里，筷子一次只能由一位哲学家使用。
2. 至少一个任务必须拥有资源，并等待获取当前由另一任务拥有的资源。也就是说，要使僵局发生，哲学家必须握住一根筷子，等待另一根筷子。
3. 不能抢先从任务中夺走资源。任务仅作为正常事件释放资源。我们的哲学家很有礼貌，他们不会抓住其他哲学家的筷子。
4. 可能发生循环等待，即一个任务等待另一个任务持有的资源，而该任务又等待另一个任务持有的资源，依此类推，直到一个任务正在等待另一个任务持有的资源。第一项任务，从而使一切陷入僵局。在 **DiningPhilosophers.java** 中，发生循环等待是因为每个哲学家都先尝试获取右筷子，然后再获取左筷子。

因为必须满足所有这些条件才能导致死锁，所以你只能阻止其中一个解除死锁。在此程序中，防止死锁的一种简单方法是打破第四个条件。之所以会发生这种情况，是因为每个哲学家都尝试按照特定的顺序拾起自己的筷子：先右后左。因此，每个哲学家都有可能在等待左手的同时握住右手的筷子，从而导致循环等待状态。但是，如果其中一位哲学家尝试首先拿起左筷子，则该哲学家决不会阻止紧邻右方

[7]而不是超线程；通常每个内核有两个超线程，并且在询问内核数量时，本书所使用的 Java 版本会报告超线程的数量。超线程产生了更快的上下文切换，但是只有实际的内核才真的工作，而不是超线程。

的哲学家拿起筷子，从而排除了循环等待。在 **DiningPhilosophers.java** 中，取消注释1和其后的一行。这将原来的哲学家1替换为筷子颠倒的哲学家。通过确保第二位哲学家拾起并在右手之前放下左筷子，我们消除了死锁的可能性。这只是解决问题的一种方法。你也可以通过防止其他情况之一来解决它。没有语言支持可以帮助防止死锁；你有责任通过精心设计来避免这种情况。对于试图调试死锁程序的人来说，这些都不是安慰。当然，避免并发问题的最简单，最好的方法是永远不要共享资源-不幸的是，这并不总是可能的。

25.9 构造函数非线程安全

当你在脑子里想象一个对象构造的过程，你会很容易认为这个过程是线程安全的。毕竟，在对象初始化完成前对外不可见，所以又怎会对此产生争议呢？确实，Java 语言规范 (JLS) 自信满满地陈述道："*没必要使构造器的线程同步，因为它会锁定正在构造的对象，直到构造器完成初始化后才对其他线程可见。*" 不幸的是，对象的构造过程如其他操作一样，也会受到共享内存并发问题的影响，只是作用机制可能更微妙罢了。

设想下使用一个**静态**字段为每个对象自动创建唯一标识符的过程。为了测试其不同的实现过程，我们从一个接口开始。代码示例：

```
//concurrent/HasID.java
public interface HasID {
    int getID();
}
```

然后 **StaticIDField** 类显式地实现该接口。代码示例：

```
// concurrent/StaticIDField.java
public class StaticIDField implements HasID {
    private static int counter = 0;
    private int id = counter++;
    public int getID() { return id; }
}
```

如你所想，该类是个简单无害的类，它甚至都没一个显式的构造器来引发问题。当我们运行多个用于创建此类对象的线程时，究竟会发生什么？为了搞清楚这点，我们做了以下测试。代码示例：

```
// concurrent/IDChecker.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import java.util.concurrent.*;
import com.google.common.collect.Sets;
public class IDChecker {
    public static final int SIZE = 100_000;

```

```
    static class MakeObjects implements
        Supplier<List<Integer>> {
        private Supplier<HasID> gen;

        MakeObjects(Supplier<HasID> gen) {
            this.gen = gen;
        }

        @Override public List<Integer> get() {
            return Stream.generate(gen)
            .limit(SIZE)
            .map(HasID::getID)
            .collect(Collectors.toList());
        }
    }

    public static void test(Supplier<HasID> gen) {
        CompletableFuture<List<Integer>>
        groupA = CompletableFuture.supplyAsync(new
            MakeObjects(gen)),
        groupB = CompletableFuture.supplyAsync(new
            MakeObjects(gen));

        groupA.thenAcceptBoth(groupB, (a, b) -> {
            System.out.println(
                Sets.intersection(
                Sets.newHashSet(a),
                Sets.newHashSet(b)).size());
            }).join();
    }
}
```

MakeObjects 类是一个生产者类，包含一个能够产生 List<Integer> 类型的列表对象的 `get()` 方法。通过从每个 `HasID` 对象提取 `ID` 并放入列表中来生成这个列表对象，而 `test()` 方法则创建了两个并行的 **CompletableFuture** 对象，用于运行 **MakeObjects** 生产者类，然后获取运行结果。使用 Guava 库中的 **Sets.**`intersection()` 方法，计算出这两个返回的 List<Integer> 对象中有多少相同的 `ID`（使用谷歌 Guava 库里的方法比使用官方的 `retainAll()` 方法速度快得多）。

现在我们可以测试上面的 **StaticIDField** 类了。代码示例：

```
// concurrent/TestStaticIDField.java
public class TestStaticIDField {

    public static void main(String[] args) {
        IDChecker.test(StaticIDField::new);
    }
}
```

输出结果：

```
    13287
```

结果中出现了很多重复项。很显然，纯静态 **int** 用于构造过程并不是线程安全的。让我们使用 **AtomicInteger** 来使其变为线程安全的。代码示例：

```java
// concurrent/GuardedIDField.java
import java.util.concurrent.atomic.*;
public class GuardedIDField implements HasID {
    private static AtomicInteger counter = new
        AtomicInteger();

    private int id = counter.getAndIncrement();

    public int getID() { return id; }

    public static void main(String[] args) {                IDChecker.
        test(GuardedIDField::new);
    }
}
```

输出结果：

```
    0
```

构造器有一种更微妙的状态共享方式：通过构造器参数：

```java
// concurrent/SharedConstructorArgument.java
import java.util.concurrent.atomic.*;
interface SharedArg{
    int get();
}

class Unsafe implements SharedArg{
    private int i = 0;

    public int get(){
        return i++;
    }
}

class Safe implements SharedArg{
    private static AtomicInteger counter = new AtomicInteger();

    public int get(){
        return counter.getAndIncrement();
    }
}

class SharedUser implements HasID{
    private final int id;
```

```java

    SharedUser(SharedArg sa){
        id = sa.get();
    }

    @Override
    public int getID(){
        return id;
    }
}

public class SharedConstructorArgument{
    public static void main(String[] args){
        Unsafe unsafe = new Unsafe();
        IDChecker.test(() -> new SharedUser(unsafe));

        Safe safe = new Safe();
        IDChecker.test(() -> new SharedUser(safe));
    }
}
```

输出结果:

```
    24838
    0
```

在这里，**SharedUser** 构造器实际上共享了相同的参数。即使 **SharedUser** 以完全无害且合理的方式使用其自己的参数，其构造器的调用方式也会引起冲突。**SharedUser** 甚至不知道它是以这种方式调用的，更不必说控制它了。同步构造器并不被 java 语言所支持，但是通过使用同步语块来创建你自己的同步构造器是可能的（请参阅附录：并发底层原理，来进一步了解同步关键字——`synchronized`）。尽管 JLS（java 语言规范）这样陈述道："……它会锁定正在构造的对象"，但这并不是真的——构造器实际上只是一个静态方法，因此同步构造器实际上会锁定该类的 Class 对象。我们可以通过创建自己的静态对象并锁定它，来达到与同步构造器相同的效果:

```java
// concurrent/SynchronizedConstructor.java

import java.util.concurrent.atomic.*;

class SyncConstructor implements HasID{
    private final int id;
    private static Object constructorLock =
        new Object();

    SyncConstructor(SharedArg sa){
        synchronized (constructorLock){
            id = sa.get();
        }
    }
```

```java

    @Override
    public int getID(){
        return id;
    }
}

public class SynchronizedConstructor{
    public static void main(String[] args){
        Unsafe unsafe = new Unsafe();
        IDChecker.test(() -> new SyncConstructor(unsafe));
    }
}
```

输出结果:

```
    0
```

Unsafe 类的共享使用现在就变得安全了。另一种方法是将构造器设为私有（因此可以防止继承），并提供一个静态 Factory 方法来生成新对象:

```java
// concurrent/SynchronizedFactory.java
import java.util.concurrent.atomic.*;

final class SyncFactory implements HasID{
    private final int id;

    private SyncFactory(SharedArg sa){
        id = sa.get();
    }

    @Override
    public int getID(){
        return id;
    }

    public static synchronized SyncFactory factory(SharedArg sa){
        return new SyncFactory(sa);
    }
}

public class SynchronizedFactory{
    public static void main(String[] args){
        Unsafe unsafe = new Unsafe();
        IDChecker.test(() -> SyncFactory.factory(unsafe));
    }
}
```

输出结果:

```
    0
```

通过同步静态工厂方法，可以在构造过程中锁定 **Class** 对象。这些示例充分表明了在并发 Java 程序中检测和管理共享状态有多困难。即使你采取“不共享任何内容”的策略，也很容易产生意外的共享事件。## 复杂性和代价

假设你正在做披萨，我们把从整个流程的当前步骤到下一个步骤所需的工作量，在这里一一表示为枚举变量的一部分：

```
// concurrent/Pizza.java import java.util.function.*;

import onjava.Nap;
public class Pizza{
    public enum Step{
        DOUGH(4), ROLLED(1), SAUCED(1), CHEESED(2),
        TOPPED(5), BAKED(2), SLICED(1), BOXED(0);
        int effort;// Needed to get to the next step

        Step(int effort){
            this.effort = effort;
        }

        Step forward(){
            if (equals(BOXED)) return BOXED;
            new Nap(effort * 0.1);
            return values()[ordinal() + 1];
        }
    }

    private Step step = Step.DOUGH;
    private final int id;

    public Pizza(int id){
        this.id = id;
    }

    public Pizza next(){
        step = step.forward();
        System.out.println("Pizza " + id + ": " + step);
        return this;
    }

    public Pizza next(Step previousStep){
        if (!step.equals(previousStep))
            throw new IllegalStateException("Expected " +
                      previousStep + " but found " + step);
        return next();
    }

    public Pizza roll(){
```

```
        return next(Step.DOUGH);
    }

    public Pizza sauce(){
        return next(Step.ROLLED);
    }

    public Pizza cheese(){
        return next(Step.SAUCED);
    }

    public Pizza toppings(){
        return next(Step.CHEESED);
    }

    public Pizza bake(){
        return next(Step.TOPPED);
    }

    public Pizza slice(){
        return next(Step.BAKED);
    }

    public Pizza box(){
        return next(Step.SLICED);
    }

    public boolean complete(){
        return step.equals(Step.BOXED);
    }

    @Override
    public String toString(){
        return "Pizza" + id + ": " + (step.equals(Step.BOXED) ? "
            complete" : step);
    }
}
```

这只算得上是一个平凡的状态机，就像 **Machina** 类一样。

制作一个披萨，当披萨饼最终被放在盒子中时，就算完成最终任务了。如果一个人在做一个披萨饼，那么所有步骤都是线性进行的，即一个接一个地进行：

```
// concurrent/OnePizza.java

import onjava.Timer;

public class OnePizza{
    public static void main(String[] args){
        Pizza za = new Pizza(0);
        System.out.println(Timer.duration(() -> {
```

```
            while (!za.complete()) za.next();
        }));
    }
}
```

输出结果:

```
Pizza 0: ROLLED
Pizza 0: SAUCED
Pizza 0: CHEESED
Pizza 0: TOPPED
Pizza 0: BAKED
Pizza 0: SLICED
Pizza 0: BOXED
    1622
```

时间以毫秒为单位，加总所有步骤的工作量，会得出与我们的期望值相符的数字。如果你以这种方式制作了五个披萨，那么你会认为它花费的时间是原来的五倍。但是，如果这还不够快怎么办？我们可以从尝试并行流方法开始:

```
// concurrent/PizzaStreams.java
// import java.util.*; import java.util.stream.*;

import onjava.Timer;

public class PizzaStreams{
    static final int QUANTITY = 5;

    public static void main(String[] args){
        Timer timer = new Timer();
        IntStream.range(0, QUANTITY)
            .mapToObj(Pizza::new)
            .parallel()//[1]
            .forEach(za -> { while(!za.complete()) za.next(); });
                        System.out.println(timer.duration());
    }
}
```

输出结果:

```
Pizza 2: ROLLED
Pizza 0: ROLLED
Pizza 1: ROLLED
Pizza 4: ROLLED
Pizza 3:ROLLED
Pizza 2:SAUCED
Pizza 1:SAUCED
Pizza 0:SAUCED
Pizza 4:SAUCED
Pizza 3:SAUCED
```

```
Pizza 2:CHEESED
Pizza 1:CHEESED
Pizza 0:CHEESED
Pizza 4:CHEESED
Pizza 3:CHEESED
Pizza 2:TOPPED
Pizza 1:TOPPED
Pizza 0:TOPPED
Pizza 4:TOPPED
Pizza 3:TOPPED
Pizza 2:BAKED
Pizza 1:BAKED
Pizza 0:BAKED
Pizza 4:BAKED
Pizza 3:BAKED
Pizza 2:SLICED
Pizza 1:SLICED
Pizza 0:SLICED
Pizza 4:SLICED
Pizza 3:SLICED
Pizza 2:BOXED
Pizza 1:BOXED
Pizza 0:BOXED
Pizza 4:BOXED
Pizza 3:BOXED
1739
```

现在，我们制作五个披萨的时间与制作单个披萨的时间就差不多了。尝试删除标记为1的行后，你会发现它花费的时间是原来的五倍。你还可以尝试将 **QUANTITY** 更改为 4、8、10、16 和 17，看看会有什么不同，并猜猜看为什么会这样。

PizzaStreams 类产生的每个并行流在它的`forEach()`内完成所有工作，如果我们将其各个步骤用映射的方式一步一步处理，情况会有所不同吗？

```java
// concurrent/PizzaParallelSteps.java

import java.util.*;
import java.util.stream.*;
import onjava.Timer;

public class PizzaParallelSteps{
    static final int QUANTITY = 5;

    public static void main(String[] args){
        Timer timer = new Timer();
        IntStream.range(0, QUANTITY)
            .mapToObj(Pizza::new)
            .parallel()
            .map(Pizza::roll)
            .map(Pizza::sauce)
```

```
            .map(Pizza::cheese)
            .map(Pizza::toppings)
            .map(Pizza::bake)
            .map(Pizza::slice)
            .map(Pizza::box)
            .forEach(za -> System.out.println(za));
        System.out.println(timer.duration());
    }
}
```

输出结果:

```
Pizza 2: ROLLED
Pizza 0: ROLLED
Pizza 1: ROLLED
Pizza 4: ROLLED
Pizza 3: ROLLED
Pizza 1: SAUCED
Pizza 0: SAUCED
Pizza 2: SAUCED
Pizza 3: SAUCED
Pizza 4: SAUCED
Pizza 1: CHEESED
Pizza 0: CHEESED
Pizza 2: CHEESED
Pizza 3: CHEESED
Pizza 4: CHEESED
Pizza 0: TOPPED
Pizza 2: TOPPED
Pizza 1: TOPPED
Pizza 3: TOPPED
Pizza 4: TOPPED
Pizza 1: BAKED
Pizza 2: BAKED
Pizza 0: BAKED
Pizza 4: BAKED
Pizza 3: BAKED
Pizza 0: SLICED
Pizza 2: SLICED
Pizza 1: SLICED
Pizza 3: SLICED
Pizza 4: SLICED
Pizza 1: BOXED
Pizza1: complete
Pizza 2: BOXED
Pizza 0: BOXED
Pizza2: complete
Pizza0: complete
Pizza 3: BOXED
Pizza 4: BOXED
Pizza4: complete
```

```
Pizza3: complete
1738
```

答案是“否”，事后看来这并不奇怪，因为每个披萨都需要按顺序执行步骤。因此，没法通过分步执行操作来进一步提高速度，就像上文的 PizzaParallelSteps.java 里面展示的一样。

我们可以使用 **CompletableFutures** 重写这个例子：

```java
// concurrent/CompletablePizza.java

import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.Timer;

public class CompletablePizza{
    static final int QUANTITY = 5;

    public static CompletableFuture<Pizza> makeCF(Pizza za){
        return CompletableFuture
                .completedFuture(za)
            .thenApplyAsync(Pizza::roll)
            .thenApplyAsync(Pizza::sauce)
            .thenApplyAsync(Pizza::cheese)
            .thenApplyAsync(Pizza::toppings)
            .thenApplyAsync(Pizza::bake)
            .thenApplyAsync(Pizza::slice)
            .thenApplyAsync(Pizza::box);
    }

    public static void show(CompletableFuture<Pizza> cf){
        try{
            System.out.println(cf.get());
        } catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args){
        Timer timer = new Timer();
        List<CompletableFuture<Pizza>> pizzas =
                IntStream.range(0, QUANTITY)
            .mapToObj(Pizza::new)
            .map(CompletablePizza::makeCF)
            .collect(Collectors.toList());
        System.out.println(timer.duration());
        pizzas.forEach(CompletablePizza::show);
        System.out.println(timer.duration());
    }
}
```

输出结果:

```
169
Pizza 0: ROLLED
Pizza 1: ROLLED
Pizza 2: ROLLED
Pizza 4: ROLLED
Pizza 3: ROLLED
Pizza 1: SAUCED
Pizza 0: SAUCED
Pizza 2: SAUCED
Pizza 4: SAUCED
Pizza 3: SAUCED
Pizza 0: CHEESED
Pizza 4: CHEESED
Pizza 1: CHEESED
Pizza 2: CHEESED
Pizza 3: CHEESED
Pizza 0: TOPPED
Pizza 4: TOPPED
Pizza 1: TOPPED
Pizza 2: TOPPED
Pizza 3: TOPPED
Pizza 0: BAKED
Pizza 4: BAKED
Pizza 1: BAKED
Pizza 3: BAKED
Pizza 2: BAKED
Pizza 0: SLICED
Pizza 4: SLICED
Pizza 1: SLICED
Pizza 3: SLICED
Pizza 2: SLICED
Pizza 4: BOXED
Pizza 0: BOXED
Pizza0: complete
Pizza 1: BOXED
Pizza1: complete
Pizza 3: BOXED
Pizza 2: BOXED
Pizza2: complete
Pizza3: complete
Pizza4: complete
1797
```

并行流和 **CompletableFutures** 是 Java 并发工具箱中最先进发达的技术。你应该始终首先选择其中之一。当一个问题很容易并行处理时，或者说，很容易把数据分解成相同的、易于处理的各个部分时，使用并行流方法处理最为合适（而如果你决定不借助它而由自己完成，你就必须撸起袖子，深入研究 **Spliterator** 的文档）。

而当工作的各个部分内容各不相同时，使用 **CompletableFutures** 是最好的选择。比起面向数据，**CompletableFutures** 更像是面向任务的。

对于披萨问题，结果似乎也没有什么不同。实际上，并行流方法看起来更简洁，仅出于这个原因，我认为并行流作为解决问题的首次尝试方法更具吸引力。

由于制作披萨总需要一定的时间，无论你使用哪种并发方法，你能做到的最好情况，是在制作一个披萨的相同时间内制作 n 个披萨。在这里当然很容易看出来，但是当你处理更复杂的问题时，你就可能忘记这一点。通常，在项目开始时进行粗略的计算，就能很快弄清楚最大可能的并行吞吐量，这可以防止你因为采取无用的加快运行速度的举措而忙得团团转。

使用 **CompletableFutures** 或许可以轻易地带来重大收益，但是在尝试更进一步时需要倍加小心，因为额外增加的成本和工作量会非常容易远远超出你之前拼命挤出的那一点点收益。

25.10 本章小结

需要并发的唯一理由是“等待太多”。这也可以包括用户界面的响应速度，但是由于 Java 用于构建用户界面时并不高效，因此[8]这仅仅意味着“您的程序运行速度还不够快”。

如果并发很容易，则没有理由拒绝并发。正因为并发实际上很难，所以您应该仔细考虑是否值得为此付出努力，并考虑您能否以其他方式提升速度。

例如，迁移到更快的硬件（这可能比消耗程序员的时间要便宜得多）或者将程序分解成多个部分，然后在不同的机器上运行这些部分。

奥卡姆剃刀是一个经常被误解的原则。我看过至少一部电影，他们将其定义为” 最简单的解决方案是正确的解决方案“，就好像这是某种毋庸置疑的法律。实际上，这是一个准则：面对多种方法时，请先尝试需要最少假设的方法。在编程世界中，这已演变为 “尝试可能可行的最简单的方法”。当您了解了特定工具的知识时——就像你现在了解了有关并发性的知识一样，你可能会很想使用它，或者提前规定你的解决方案必须能够“速度飞快”，从而来证明从一开始就进行并发设计是合理的。但是，我们的奥卡姆剃刀编程版本表示您应该首先尝试最简单的方法（这种方法开发起来也更便宜），然后看看它是否足够好。

由于我出身于底层学术背景（物理学和计算机工程），所以我很容易想到所有小轮子转动的成本。我确定使用最简单的方法不够快的场景出现的次数已经数不过来了，但是尝试后却发现它实际上绰绰有余。

25.10.1 缺点

并发编程的主要缺点是：

1. 在线程等待共享资源时会降低速度。

[8]库就在那里用于调用，而语言本身就被设计用于此目的，但实际上它很少发生，以至于可以说” 没有 “。

2. 线程管理产生额外 CPU 开销。
3. 糟糕的设计决策带来无法弥补的复杂性。
4. 诸如饥饿，竞速，死锁和活锁（多线程各自处理单个任务而整体却无法完成）之类的问题。
5. 跨平台的不一致。通过一些示例，我发现了某些计算机上很快出现的竞争状况，而在其他计算机上却没有。如果您在后者上开发程序，则在分发程序时可能会感到非常惊讶。

另外，并发的应用是一门艺术。Java 旨在允许您创建尽可能多的所需要的对象来解决问题——至少在理论上是这样。[9]但是，线程不是典型的对象：每个线程都有其自己的执行环境，包括堆栈和其他必要的元素，使其比普通对象大得多。在大多数环境中，只能在内存用光之前创建数千个 **Thread** 对象。通常，您只需要几个线程即可解决问题，因此一般来说创建线程没有什么限制，但是对于某些设计而言，它会成为一种约束，可能迫使您使用完全不同的方案。

25.10.2 共享内存陷阱

并发性的主要困难之一是因为可能有多个任务共享一个资源（例如对象中的内存），并且您必须确保多个任务不会同时读取和更改该资源。

我花了多年的时间研究并发并发。我了解到您永远无法相信使用共享内存并发的程序可以正常工作。您可以轻易发现它是错误的，但永远无法证明它是正确的。这是众所周知的并发原则之一。[10]

我遇到了许多人，他们对编写正确的线程程序的能力充满信心。我偶尔开始认为我也可以做好。对于一个特定的程序，我最初是在只有单个 CPU 的机器上编写的。那时我能够说服自己该程序是正确的，因为我以为我对 Java 工具很了解。而且在我的单 CPU 计算机上也没有失败。而到了具有多个 CPU 的计算机，程序出现问题不能运行后，我感到很惊讶，但这还只是众多问题中的一个而已。这不是 Java 的错；“写一次，到处运行”，在单核与多核计算机间无法扩展到并发编程领域。这是并发编程的基本问题。实际上您可以在单 CPU 机器上发现一些并发问题，但是在多线程实际上真的在并行运行的多 CPU 机器上，就会出现一些其他问题。

再举一个例子，哲学家就餐的问题可以很容易地进行调整，因此几乎不会产生死锁，这会给您一种一切都棒极了的印象。当涉及到共享内存并发编程时，您永远不应该对自己的编程能力变得过于自信。

25.10.3 This Albatross is Big

如果您对 Java 并发感到不知所措，那说明您身处在一家出色的公司里。您可以访问 **Thread** 类的Javadoc页面，看一下哪些方法现在是 **Deprecated**（废弃的）。这些是 Java 语言设计者犯过错的地方，因为他们在设计语言时对并发性了解不足。

事实证明，在 Java 的后续版本中添加的许多库解决方案都是无效的，甚至是无用的。幸运的是，

[9]举例来说，如果没有 Flyweight 设计模式，在工程中创建数百万个对象用于有限元分析可能在 Java 中不可行。

[10]在科学中，虽然从来没有一种理论被证实过，但是一种理论必须是可证伪的才有意义。而对于并发性，我们大部分时间甚至都无法得到这种可证伪性。

Java 8 中的并行 **Streams** 和 **CompletableFutures** 都非常有价值。但是当您使用旧代码时，仍然会遇到旧的解决方案。

在本书的其他地方，我谈到了 Java 的一个基本问题：每个失败的实验都永远嵌入在语言或库中。Java 并发强调了这个问题。尽管有不少错误，但错误并不是那么多，因为有很多不同的尝试方法来解决问题。好的方面是，这些尝试产生了更好，更简单的设计。不利之处在于，在找到好的方法之前，您很容易迷失于旧的设计中。

25.10.4 其他类库

本章重点介绍了相对安全易用的并行工具流和 **CompletableFutures**, 并且仅涉及 Java 标准库中一些更细粒度的工具。为避免您不知所措，我没有介绍您可能实际在实践中使用的某些库。我们使用了几个 **Atomic**（原子）类，**ConcurrentLinkedDeque**，**ExecutorService** 和 **ArrayBlockingQueue**。附录：并发底层原理涵盖了其他一些内容，但是您还想探索 **java.util.concurrent** 的 Javadocs。但是要小心，因为某些库组件已被新的更好的组件所取代。

25.10.5 考虑为并发设计的语言

通常，请谨慎地使用并发。如果需要使用它，请尝试使用最现代的方法：并行流或 **CompletableFutures**。这些功能旨在（假设您不尝试共享内存）使您摆脱麻烦（在 Java 的世界范围内）。

如果您的并发问题变得比高级 Java 构造所支持的问题更大且更复杂，请考虑使用专为并发设计的语言，仅在需要并发的程序部分中使用这种语言是有可能的。在撰写本文时，JVM 上最纯粹的功能语言是 Clojure（Lisp 的一种版本）和 Frege（Haskell 的一种实现）。这些使您可以在其中编写应用程序的并发部分语言，并通过 JVM 轻松地与您的主要 Java 代码进行交互。或者，您可以选择更复杂的方法，即通过外部功能接口（FFI）将 JVM 之外的语言与另一种为并发设计的语言进行通信。[11]

你很容易被一种语言绑定，迫使自己尝试使用该语言来做所有事情。一个常见的示例是构建 HTML / JavaScript 用户界面。这些工具确实很难使用，令人讨厌，并且有许多库允许您通过使用自己喜欢的语言编写代码来生成这些工具（例如，**Scala.js** 允许您在 Scala 中完成代码）。

心理上的便利是一个合理的考虑因素。但是，我希望我在本章（以及附录：并发底层原理）中已经表明 Java 并发是一个你可能无法逃离很深的洞。与 Java 语言的任何其他部分相比，在视觉上检查代码同时记住所有陷阱所需要的的知识要困难得多。

无论使用特定的语言、库使得并发看起来多么简单，都要将其视为一种妖术，因为总是有东西会在您最不期望出现的时候咬您。

[11] 尽管 **Go** 语言显示了 FFI 的前景，但在撰写本文时，它并未提供跨所有平台的解决方案。

25.10.6 拓展阅读

《Java Concurrency in Practice》，出自 Brian Goetz，Tim Peierls，Joshua Bloch，Joseph Bowbeer，David Holmes 和 Doug Lea (Addison Wesley，2006 年)——这些基本上就是 Java 并发世界中的名人名单了《Java Concurrency in Practice》第二版，出自 Doug Lea (Addison-Wesley，2000 年)。尽管这本书出版时间远远早于 Java 5 发布，但 Doug 的大部分工作都写入了 **java.util.concurrent** 库。因此，这本书对于全面理解并发问题至关重要。它超越了 Java，讨论了跨语言和技术的并发编程。尽管它在某些地方可能很钝，但值得多次重读（最好是在两个月之间进行消化）。道格（Doug）是世界上为数不多的真正了解并发编程的人之一，因此这是值得的。

26 设计模式

26.1 概念

最初，你可以将模式视为解决特定类问题的一种特别巧妙且有深刻见解的方法。这就像前辈已经从所有角度去解决问题，并提出了最通用，最灵活的解决方案。问题可能是你之前看到并解决过的问题，但你的解决方案可能没有你在模式中体现的那种完整性。

虽然它们被称为“设计模式”，但它们实际上并不与设计领域相关联。模式似乎与传统的分析、设计和实现的思维方式不同。相反，模式在程序中体现了一个完整的思想，因此它有时会出现在分析阶段或高级设计阶段。因为模式在代码中有一个直接的实现，所以你可能不会期望模式在低级设计或实现之前出现 (而且通常在到达这些阶段之前，你不会意识到需要一个特定的模式)。

模式的基本概念也可以看作是程序设计的基本概念: 添加抽象层。当你抽象一些东西的时候，就像在剥离特定的细节，而这背后最重要的动机之一是: > **将易变的事物与不变的事物分开**

另一种方法是，一旦你发现程序的某些部分可能因某种原因而发生变化，你要保持这些变化不会引起整个代码中其他变化。如果代码更容易理解，那么维护起来会更容易。

通常，开发一个优雅且易维护设计中最困难的部分是发现我称之为变化的载体（也就是最易改变的地方）。这意味着找到系统中最重要的变化，换而言之，找到变化会导致最严重后果的地方。一旦发现变化载体，就可以围绕构建设计的焦点。

因此，设计模式的目标是隔离代码中的更改。如果以这种方式去看，你已经在本书中看到了设计模式。例如，继承可以被认为是一种设计模式（虽然是由编译器实现的）。它允许你表达所有具有相同接口的对象（即保持相同的行为）中的行为差异（这就是变化的部分）。组合也可以被视为一种模式，因为它允许你动态或静态地更改实现类的对象，从而改变类的工作方式。

你还看到了设计模式中出现的另一种模式：迭代器（Java 1.0 和 1.1 随意地将其称为枚举; Java 2 集合才使用 Iterator）。当你逐个选择元素时并逐步处理，这会隐藏集合的特定实现。迭代器允许你编写通用代码，该代码对序列中的所有元素执行操作，而不考虑序列的构建方式。因此，你的通用代码可以与任何可以生成迭代器的集合一起使用。

即使模式是非常有用的，但有些人断言：> **设计模式代表语言的失败**。

这是一个非常重要的见解，因为一个模式在 C++ 有意义，可能在 JAVA 或者其他语言中就没有意义。出于这个原因，所以一个模式可能出现在设计模式书上，不意味着应用于你的编程语言是有用的。

我认为“语言失败”这个观点是有道理的，但是我也认为这个观点过于简单化。如果你试图解决一个

特定的问题，而你使用的语言没有直接提供支持你使用的技巧，你可以说这个是语言的失败。但是，你使用特定的技巧的频率的是多少呢？也许平衡是对的：当你使用特定的技巧的时候，你必须付出更多的努力，但是你又没有足够的理由去使得语言支持这个技术。另一方面，没有语言的支持，使用这种技术常常会很混乱，但是在语言支持下，你可能会改变编程方式（例如，Java 8 流实现此目的）。

26.1.1 单例模式

也许单例模式是最简单的设计模式，它是一种提供一个且只有一个对象实例的方法。这在 java 库中使用，但是这有个更直接的示例：

```java
// patterns/SingletonPattern.java
interface Resource {
    int getValue();
    void setValue(int x);
}

/*
* 由于这不是从Cloneable基类继承而且没有添加可克隆性，
* 因此将其设置为final可防止通过继承添加可克隆性。
* 这也实现了线程安全的延迟初始化：
*/
final class Singleton {
    private static final class ResourceImpl implements Resource {
        private int i;
        private ResourceImpl(int i) {
            this.i = i;
        }
        public synchronized int getValue() {
            return i;
        }
        public synchronized void setValue(int x) {
            i = x;
        }
    }

    private static class ResourceHolder {
        private static Resource resource = new ResourceImpl(47);
    }
    public static Resource getResource() {
        return ResourceHolder.resource;
    }
}

public class SingletonPattern {
    public static void main(String[] args) {
        Resource r = Singleton.getResource();
        System.out.println(r.getValue());
```

```
        Resource s2 = Singleton.getResource();
        s2.setValue(9);
        System.out.println(r.getValue());
        try {
            // 不能这么做，会发生: compile-time error (编译时错误) .
            // Singleton s3 = (Singleton)s2.clone();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
} /* Output: 47 9 */
```

创建单例的关键是防止客户端程序员直接创建对象。在这里，这是通过在 Singleton 类中将 Resource 的实现作为私有类来实现的。

此时，你将决定如何创建对象。在这里，它是按需创建的，在第一次访问的时候创建。该对象是私有的，只能通过 public getResource () 方法访问。

懒惰地创建对象的原因是它嵌套的私有类 resourceHolder 在首次引用之前不会加载（在 getResource () 中)。当 Resource 对象加载的时候，静态初始化块将被调用。由于 JVM 的工作方式，这种静态初始化是线程安全的。为保证线程安全，Resource 中的 getter 和 setter 是同步的。

26.1.2 模式分类

“设计模式”一书讨论了 23 种不同的模式，分为以下三种类别（所有这些模式都围绕着可能变化的特定方面)。

1. **创建型**：如何创建对象。这通常涉及隔离对象创建的细节，这样你的代码就不依赖于具体的对象的类型，因此在添加新类型的对象时不会更改。单例模式（Singleton）被归类为创作模式，本章稍后你将看到 Factory Method 的示例。
2. **构造型**：设计对象以满足特定的项目约束。它们处理对象与其他对象连接的方式，以确保系统中的更改不需要更改这些连接。
3. **行为型**：处理程序中特定类型的操作的对象。这些封装要执行的过程，例如解释语言、实现请求、遍历序列 (如在迭代器中) 或实现算法。本章包含观察者和访问者模式的例子。

《设计模式》一书中每个设计模式都有单独的一个章节，每个章节都有一个或者多个例子，通常使用 C++，但有时也使用 SmallTalk。本章不重复设计模式中显示的所有模式，因为该书独立存在，应单独研究。相反，你会看到一些示例，可以为你提供关于模式的理解以及它们如此重要的原因。

26.2 构建应用程序框架

应用程序框架允许您从一个类或一组类开始，创建一个新的应用程序，重用现有类中的大部分代码，并根据需要覆盖一个或多个方法来定制应用程序。

模板方法模式

应用程序框架中的一个基本概念是模板方法模式，它通常隐藏在底层，通过调用基类中的各种方法来驱动应用程序 (为了创建应用程序，您已经覆盖了其中的一些方法)。

模板方法模式的一个重要特性是它是在基类中定义的，并且不能更改。它有时是一个 **private** 方法，但实际上总是 **final**。它调用其他基类方法 (您覆盖的那些) 来完成它的工作, 但是它通常只作为初始化过程的一部分被调用 (因此框架使用者不一定能够直接调用它)。

```
// patterns/TemplateMethod.java
// Simple demonstration of Template Method

abstract class ApplicationFramework {
    ApplicationFramework() {
        templateMethod();
    }

    abstract void customize1();

    abstract void customize2(); // "private" means automatically "final
        ": private void templateMethod() { IntStream.range(0, 5).forEach
        ( n -> { customize1(); customize2(); }); }}// Create a new "
        application": class MyApp extends ApplicationFramework {
        @Override void customize1() { System.out.print("Hello "); }
        @Override

    void customize2() {
        System.out.println("World!");
    }
}

public class TemplateMethod {
    public static void main(String[] args) {
        new MyApp();
    }
}
/*
Output:
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
*/
```

基类构造函数负责执行必要的初始化，然后启动运行应用程序的“engine”(模板方法模式)(在 GUI 应用程序中，这个“engine”是主事件循环)。框架使用者只提供 **customize1()** 和 **customize2()** 的定义，然后“应用程序”已经就绪运行。

26.3 面向实现

代理模式和桥接模式都提供了在代码中使用的代理类; 完成工作的真正类隐藏在这个代理类的后面。当您在代理中调用一个方法时，它只是反过来调用实现类中的方法。这两种模式非常相似，所以代理模式只是桥接模式的一种特殊情况。人们倾向于将两者合并, 称为代理模式，但是术语“代理”有一个长期的和专门的含义，这可能解释了这两种模式不同的原因。基本思想很简单: 从基类派生代理，同时派生一个或多个提供实现的类: 创建代理对象时，给它一个可以调用实际工作类的方法的实现。

在结构上，代理模式和桥接模式的区别很简单: 代理模式只有一个实现，而桥接模式有多个实现。在设计模式中被认为是不同的: 代理模式用于控制对其实现的访问，而桥接模式允许您动态更改实现。但是，如果您扩展了“控制对实现的访问”的概念，那么这两者就可以完美地结合在一起

代理模式

如果我们按照上面的关系图实现，它看起来是这样的:

```
// patterns/ProxyDemo.java
// Simple demonstration of the Proxy pattern
interface ProxyBase {
    void f();

    void g();

    void h();
}

class Proxy implements ProxyBase {
    private ProxyBase implementation;

    Proxy() {
        implementation = new Implementation();
    }
    // Pass method calls to the implementation:
    @Override
```

```
    public void f() { implementation.f(); }
    @Override
    public void g() { implementation.g(); }
    @Override
    public void h() { implementation.h(); }
}

class Implementation implements ProxyBase {
    public void f() {
        System.out.println("Implementation.f()");
    }

    public void g() {
        System.out.println("Implementation.g()");
    }

    public void h() {
        System.out.println("Implementation.h()");
    }
}

public class ProxyDemo {
    public static void main(String[] args) {
        Proxy p = new Proxy();
        p.f();
        p.g();
        p.h();
    }
}
/*
Output:
Implementation.f()
Implementation.g()
Implementation.h()
*/
```

具体实现不需要与代理对象具有相同的接口; 只要代理对象以某种方式 “代表具体实现的方法调用，那么基本思想就算实现了。然而，拥有一个公共接口是很方便的，因此具体实现必须实现代理对象调用的所有方法。

状态模式

状态模式向代理对象添加了更多的实现，以及在代理对象的生命周期内从一个实现切换到另一种实现的方法:

```
// patterns/StateDemo.java // Simple demonstration of the State pattern
interface StateBase {
    void f();

    void g();
```

```java

    void h();

    void changeImp(StateBase newImp);
}

class State implements StateBase {
    private StateBase implementation;

    State(StateBase imp) {
        implementation = imp;
    }

    @Override
    public void changeImp(StateBase newImp) {
        implementation = newImp;
    }// Pass method calls to the implementation: @Override public void f() { implementation.f(); } @Override public void g() { implementation.g(); } @Override

    public void h() {
        implementation.h();
    }
}

class Implementation1 implements StateBase {
    @Override
    public void f() {
        System.out.println("Implementation1.f()");
    }

    @Override
    public void g() {
        System.out.println("Implementation1.g()");
    }

    @Override
    public void h() {
        System.out.println("Implementation1.h()");
    }

    @Override
    public void changeImp(StateBase newImp) {
    }
}

class Implementation2 implements StateBase {
    @Override
    public void f() {
        System.out.println("Implementation2.f()");
    }
```

```java

    @Override
    public void g() {
        System.out.println("Implementation2.g()");
    }

    @Override
    public void h() {
        System.out.println("Implementation2.h()");
    }

    @Override
    public void changeImp(StateBase newImp) {
    }
}

public class StateDemo {
    static void test(StateBase b) {
        b.f();
        b.g();
        b.h();
    }

    public static void main(String[] args) {
        StateBase b = new State(new Implementation1());
        test(b);
        b.changeImp(new Implementation2());
        test(b);
    }
}
/* Output:
Implementation1.f()
Implementation1.g()
Implementation1.h()
Implementation2.f()
Implementation2.g()
Implementation2.h()
*/
```

在 main() 中，首先使用第一个实现，然后改变成第二个实现。代理模式和状态模式的区别在于它们解决的问题。设计模式中描述的代理模式的常见用途如下:

1. 远程代理。它在不同的地址空间中代理对象。远程方法调用 (RMI) 编译器 rmic 会自动为您创建一个远程代理。
2. 虚拟代理。这提供了“懒加载”来根据需要创建“昂贵”的对象。
3. 保护代理。当您希望对代理对象有权限访问控制时使用。
4. 智能引用。要在被代理的对象被访问时添加其他操作。例如，跟踪特定对象的引用数量，来实现写时复制用法，和防止对象别名。一个更简单的例子是跟踪特定方法的调用数量。您可以将 Java

引用视为一种保护代理，因为它控制在堆上实例对象的访问 (例如，确保不使用空引用)。

在设计模式中，代理模式和桥接模式并不是相互关联的，因为它们被赋予 (我认为是任意的) 不同的结构。桥接模式, 特别是使用一个单独的实现，但这似乎对我来说是不必要的, 除非你确定该实现是你无法控制的 (当然有可能，但是如果您编写所有代码，那么没有理由不从单基类的优雅中受益)。此外，只要代理对象控制对其“前置”对象的访问，代模式理就不需要为其实现使用相同的基类。不管具体情况如何，在代理模式和桥接模式中，代理对象都将方法调用传递给具体实现对象。

状态机

桥接模式允许程序员更改实现，状态机利用一个结构来自动地将实现更改到下一个。当前实现表示系统所处的状态，系统在不同状态下的行为不同 (因为它使用桥接模式)。基本上，这是一个利用对象的“状态机”。将系统从一种状态移动到另一种状态的代码通常是模板方法模式，如下例所示:

```java
// patterns/state/StateMachineDemo.java
// The StateMachine pattern and Template method
// {java patterns.state.StateMachineDemo}
package patterns.state;

import onjava.Nap;

interface State {
    void run();
}

abstract class StateMachine {
    protected State currentState;

    Nap(0.5);
System.out.println("Washing"); new

    protected abstract boolean changeState();

    // Template method:
    protected final void runAll() {
        while (changeState()) // Customizable
            currentState.run();
    }
}

// A different subclass for each state:
class Wash implements State {
    @Override
    public void run() {
    }
}

class Spin implements State {
    @Override
```

```
    public void run() {
        System.out.println("Spinning");
        new Nap(0.5);
    }
}

class Rinse implements State {
    @Override
    public void run() {
        System.out.println("Rinsing");
        new Nap(0.5);
    }
}

class Washer extends StateMachine {
    private int i = 0;

    // The state table:
    private State[] states = {new Wash(), new Spin(), new Rinse(), new
        Spin(),};

    Washer() {
        runAll();
    }

    @Override
    public boolean changeState() {
        if (i < states.length) {
            // Change the state by setting the
            // surrogate reference to a new object:
            currentState = states[i++];
            return true;
        } else return false;
    }
}

public class StateMachineDemo {
    public static void main(String[] args) {
        new Washer();
    }
}
/*
Output:
Washing
Spinning
Rinsing
Spinning
*/
```

在这里，控制状态的类 (本例中是状态机) 负责决定下一个状态。然而，状态对象本身也可以决定

下一步移动到什么状态，通常基于系统的某种输入。这是更灵活的解决方案。

26.4 工厂模式

当你发现必须将新类型添加到系统中时，合理的第一步是使用多态性为这些新类型创建一个通用接口。这会将你系统中的其余代码与要添加的特定类型的信息分开，使得可以在不改变现有代码的情况下添加新类型……或者看起来如此。起初，在这种设计中，似乎你必须更改代码的唯一地方就是你继承新类型的地方，但这并不是完全正确的。你仍然必须创建新类型的对象，并且在创建时必须指定要使用的确切构造器。因此，如果创建对象的代码分布在整个应用程序中，那么在添加新类型时，你将遇到相同的问题——你仍然必须追查你代码中新类型碍事的所有地方。恰好是类型的创建碍事，而不是类型的使用（通过多态处理），但是效果是一样的：添加新类型可能会引起问题。

解决方案是强制对象的创建都通过通用工厂进行，而不是允许创建代码在整个系统中传播。如果你程序中的所有代码都必须执行通过该工厂创建你的一个对象，那么在添加新类时只需要修改工厂即可。

由于每个面向对象的程序都会创建对象，并且很可能会通过添加新类型来扩展程序，因此工厂是最通用的设计模式之一。

举例来说，让我们重新看一下 **Shape** 系统。首先，我们需要一个用于所有示例的基本框架。如果无法创建 **Shape** 对象，则需要抛出一个合适的异常：

```
// patterns/shapes/BadShapeCreation.java package patterns.shapes;
public class BadShapeCreation extends RuntimeException {
    public BadShapeCreation(String msg) {
        super(msg);
    }
}
```

接下来，是一个 **Shape** 基类：

```
// patterns/shapes/Shape.java
package patterns.shapes;
public class Shape {
    private static int counter = 0;
    private int id = counter++;
    @Override
    public String toString(){
        return getClass().getSimpleName() + "[" + id + "]";
    }
    public void draw() {
        System.out.println(this + " draw");
    }
    public void erase() {
        System.out.println(this + " erase");
    }
```

```
}
```

该类自动为每一个 **Shape** 对象创建一个唯一的`id`。

`toString()`使用运行期信息来发现特定的 **Shape** 子类的名字。

现在我们能很快创建一些 **Shape** 子类了:

```
// patterns/shapes/Circle.java
package patterns.shapes;
public class Circle extends Shape {}
```

```
// patterns/shapes/Square.java
package patterns.shapes;
public class Square extends Shape {}
```

```
// patterns/shapes/Triangle.java
package patterns.shapes;
public class Triangle extends Shape {}
```

工厂是具有能够创建对象的方法的类。我们有几个示例版本，因此我们将定义一个接口:

```
// patterns/shapes/FactoryMethod.java
package patterns.shapes;
public interface FactoryMethod {
    Shape create(String type);
}
```

`create()`接收一个参数，这个参数使其决定要创建哪一种 **Shape** 对象，这里是`String`，但是它其实可以是任何数据集合。对象的初始化数据（这里是字符串）可能来自系统外部。这个例子将测试工厂:

```
// patterns/shapes/FactoryTest.java
package patterns.shapes;
import java.util.stream.*;
public class FactoryTest {
    public static void test(FactoryMethod factory) {
        Stream.of("Circle", "Square", "Triangle",
                  "Square", "Circle", "Circle", "Triangle")
        .map(factory::create)
        .peek(Shape::draw)
        .peek(Shape::erase)
        .count(); // Terminal operation
    }
}
```

在主函数`main()`里，要记住除非你在最后使用了一个终结操作，否则 **Stream** 不会做任何事情。在这里，`count()`的值被丢弃了。

创建工厂的一种方法是显式创建每种类型:

```java
// patterns/ShapeFactory1.java
// A simple static factory method
import java.util.*;
import java.util.stream.*;
import patterns.shapes.*;
public class ShapeFactory1 implements FactoryMethod {
    public Shape create(String type) {
        switch(type) {
            case "Circle": return new Circle();
            case "Square": return new Square();
            case "Triangle": return new Triangle();
            default: throw new BadShapeCreation(type);
        }
    }
    public static void main(String[] args) {
        FactoryTest.test(new ShapeFactory1());
    }
}
```

输出结果:

```
Circle[0] draw
Circle[0] erase
Square[1] draw
Square[1] erase
Triangle[2] draw
Triangle[2] erase
Square[3] draw
Square[3] erase
Circle[4] draw
Circle[4] erase
Circle[5] draw
Circle[5] erase
Triangle[6] draw
Triangle[6] erase
```

`create()`现在是添加新类型的 Shape 时系统中唯一需要更改的其他代码。

26.4.1 动态工厂

前面例子中的**静态**`create()`方法强制所有创建操作都集中在一个位置，因此这是添加新类型的 **Shape** 时唯一必须更改代码的地方。这当然是一个合理的解决方案，因为它把创建对象的过程限制在一个框内。但是，如果你在添加新类时无需修改任何内容，那就太好了。以下版本使用反射在首次需要时将 **Shape** 的构造器动态加载到工厂列表中:

```java
// patterns/ShapeFactory2.java
import java.util.*;
import java.lang.reflect.*;
```

```java
import java.util.stream.*;
import patterns.shapes.*;
public class ShapeFactory2 implements FactoryMethod {
    Map<String, Constructor> factories = new HashMap<>();
    static Constructor load(String id) {
        System.out.println("loading " + id);
        try {
            return Class.forName("patterns.shapes." + id)
                .getConstructor();
        } catch(ClassNotFoundException |
                NoSuchMethodException e) {
            throw new BadShapeCreation(id);
        }
    }
    public Shape create(String id) {
        try {
            return (Shape)factories
                .computeIfAbsent(id, ShapeFactory2::load)
                .newInstance();
        } catch(InstantiationException |
                IllegalAccessException |
                InvocationTargetException e) {
            throw new BadShapeCreation(id);
        }
    }
    public static void main(String[] args) {
        FactoryTest.test(new ShapeFactory2());
    }
}
```

输出结果:

```
loading Circle
Circle[0] draw
Circle[0] erase
loading Square
Square[1] draw
Square[1] erase
loading Triangle
Triangle[2] draw
Triangle[2] erase
Square[3] draw
Square[3] erase
Circle[4] draw
Circle[4] erase
Circle[5] draw
Circle[5] erase
Triangle[6] draw
Triangle[6] erase
```

和之前一样，`create()`方法基于你传递给它的 **String** 参数生成新的 **Shape**s，但是在这里，它

是通过在 **HashMap** 中查找作为键的 **String** 来实现的。返回的值是一个构造器，该构造器用于通过调用`newInstance()`创建新的 **Shape** 对象。

然而，当你开始运行程序时，工厂的`map`为空。`create()`使用`map`的`computeIfAbsent()`方法来查找构造器（如果该构造器已存在于`map`中）。如果不存在则使用`load()`计算出该构造器，并将其插入到`map`中。从输出中可以看到，每种特定类型的 **Shape** 都是在第一次请求时才加载的，然后只需要从`map`中检索它。

26.4.2 多态工厂

《设计模式》这本书强调指出，采用“工厂方法”模式的原因是可以从基本工厂中继承出不同类型的工厂。再次修改示例，使工厂方法位于单独的类中：

```
// patterns/ShapeFactory3.java
// Polymorphic factory methods
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import patterns.shapes.*;
interface PolymorphicFactory {
    Shape create();
}
class RandomShapes implements Supplier<Shape> {
    private final PolymorphicFactory[] factories;
    private Random rand = new Random(42);
    RandomShapes(PolymorphicFactory... factories){
        this.factories = factories;
    }
    public Shape get() {
        return factories[ rand.nextInt(factories.length)].create();
    }
}
public class ShapeFactory3 {
    public static void main(String[] args) {
        RandomShapes rs = new RandomShapes(
            Circle::new,
            Square::new,
            Triangle::new);
        Stream.generate(rs)
            .limit(6)
            .peek(Shape::draw)
            .peek(Shape::erase)
            .count();
    }
}
```

输出结果：

```
Triangle[0] draw
Triangle[0] erase
Circle[1] draw
Circle[1] erase
Circle[2] draw
Circle[2] erase
Triangle[3] draw
Triangle[3] erase
Circle[4] draw
Circle[4] erase
Square[5] draw
Square[5] erase
```

RandomShapes 实现了 **Supplier <Shape>**，因此可用于通过`Stream.generate()`创建 **Stream**。它的构造器采用 **PolymorphicFactory** 对象的可变参数列表。变量参数列表以数组形式出现，因此列表是以数组形式在内部存储的。`get()`方法随机获取此数组中一个对象的索引，并在结果上调用`create()`以产生新的 **Shape** 对象。添加新类型的 **Shape** 时，**RandomShapes** 构造器是唯一需要更改的地方。请注意，此构造器需要 **Supplier <Shape>**。我们传递给其 **Shape** 构造器的方法引用，该引用可满足 **Supplier <Shape>** 约定，因为 Java 8 支持结构一致性。

鉴于 **ShapeFactory2.java** 可能会抛出异常，使用此方法则没有任何异常——它在编译时完全确定。

26.4.3 抽象工厂

抽象工厂模式看起来像我们之前所见的工厂对象，但拥有不是一个工厂方法而是几个工厂方法，每个工厂方法都会创建不同种类的对象。这个想法是在创建工厂对象时，你决定如何使用该工厂创建的所有对象。《设计模式》中提供的示例实现了跨各种图形用户界面（GUI）的可移植性：你创建一个适合你正在使用的 GUI 的工厂对象，然后从中请求菜单，按钮，滑块等等，它将自动为 GUI 创建适合该项目版本的组件。因此，你可以将从一个 GUI 更改为另一个所产生的影响隔离限制在一处。作为另一个示例，假设你正在创建一个通用游戏环境来支持不同类型的游戏。使用抽象工厂看起来就像下文那样：

```java
// patterns/abstractfactory/GameEnvironment.java
// An example of the Abstract Factory pattern
// {java patterns.abstractfactory.GameEnvironment}
package patterns.abstractfactory;
import java.util.function.*;
interface Obstacle {
    void action();
}

interface Player {
    void interactWith(Obstacle o);
}
```

```java

class Kitty implements Player {
    @Override
    public void interactWith(Obstacle ob) {
        System.out.print("Kitty has encountered a ");
        ob.action();
    }
}

class KungFuGuy implements Player {
    @Override
    public void interactWith(Obstacle ob) {
        System.out.print("KungFuGuy now battles a ");
        ob.action();
    }
}

class Puzzle implements Obstacle {
    @Override
    public void action() {
        System.out.println("Puzzle");
    }
}

class NastyWeapon implements Obstacle {
    @Override
    public void action() {
        System.out.println("NastyWeapon");
    }
}

// The Abstract Factory:
class GameElementFactory {
    Supplier<Player> player;
    Supplier<Obstacle> obstacle;
}

// Concrete factories:
class KittiesAndPuzzles extends GameElementFactory {
    KittiesAndPuzzles() {
        player = Kitty::new;
        obstacle = Puzzle::new;
    }
}

class KillAndDismember extends GameElementFactory {
    KillAndDismember() {
        player = KungFuGuy::new;
        obstacle = NastyWeapon::new;
    }
}
```

```java

public class GameEnvironment {
    private Player p;
    private Obstacle ob;

    public GameEnvironment(GameElementFactory factory) {
        p = factory.player.get();
        ob = factory.obstacle.get();
    }
    public void play() {
        p.interactWith(ob);
    }
    public static void main(String[] args) {
        GameElementFactory kp = new KittiesAndPuzzles(), kd = new
            KillAndDismember();
        GameEnvironment g1 = new GameEnvironment(kp), g2 = new
            GameEnvironment(kd);
        g1.play();
        g2.play();
    }
}
```

输出结果:

```
Kitty has encountered a Puzzle
KungFuGuy now battles a NastyWeapon
```

在这种环境中，**Player** 对象与 **Obstacle** 对象进行交互，但是根据你所玩游戏的类型，存在不同类型的玩家和障碍物。你可以通过选择特定的 **GameElementFactory** 来确定游戏的类型，然后 **GameEnvironment** 控制游戏的设置和玩法。在此示例中，设置和玩法非常简单，但是这些活动（初始条件和状态变化）可以决定游戏的大部分结果。这里，**GameEnvironment** 不是为继承而设计的，尽管这样做很有意义。它还包含“双重调度”和“工厂方法”的示例，稍后将对这两个示例进行说明。

26.5 函数对象

26.6 接口改变

26.7 解释器

26.8 回调

26.9 多次调度

26.10 模式重构

26.11 抽象用法

26.12 多次派遣

26.13 访问者模式

26.14 RTTI 的优劣

26.15 本章小结

附录 1

补充

> 本书有许多补充内容，包括 MindView 网站提供的项目和服务。

本附录介绍了这些补充内容，你可以自行决定它们是否对你有所帮助。

1.1 可下载的补充

可以从 https://github.com/BruceEckel/OnJava8-examples 免费下载本书的代码。这里包括 Gradle 构建文件和其它一些必要的支持文件，以便成功构建和执行本书中所有的示例代码。

1.2 通过 Thinking-in-C 来巩固 Java 基础

在 www.OnJava8.com 上，可以免费下载 *Thinking in C* 的演示文稿。此演示文稿由 Chuck Allison 创建，由 MindView 有限责任公司开发。这是一个电子演示文稿，介绍了 Java 语法所基于的 C 语法，运算符和函数。

1.3 Hand-On Java 电子演示文稿

Hand-On Java 电子演示文稿（Hands-On Java eSeminar）是基于 *Thinking in Java* 第 2 版。对应于该书中的每一章，它附带有一个音频讲解和相应的幻灯片。我创建了这个电子演示文稿，并讲述了这些材料。这个资料是 HTML5 格式的，所以它应该可以在大多数现代浏览器上运行。该演示文稿将在www.OnJava8.com上发售，你可以在该网站上找到该产品的试用版演示。

附录 2

编程指南

> 本附录包含了有助于指导你进行低级程序设计和编写代码的建议。

当然，这些只是指导方针，而不是规则。我们的想法是将它们用作灵感，并记住偶尔会违反这些指导方针的特殊情况。

2.1 设计

1. **优雅总是会有回报**。从短期来看，似乎需要更长的时间才能找到一个真正优雅的问题解决方案，但是当该解决方案第一次应用并能轻松适应新情况，而不需要数小时，数天或数月的挣扎时，你会看到奖励（即使没有人可以测量它们）。它不仅为你提供了一个更容易构建和调试的程序，而且它也更容易理解和维护，这也正是经济价值所在。这一点可以通过一些经验来理解，因为当你想要使一段代码变得优雅时，你可能看起来效率不是很高。抵制急于求成的冲动，它只会减慢你的速度。
2. **先让它工作，然后再让它变快**。即使你确定一段代码非常重要并且它是你系统中的主要瓶颈 **，也是如此。不要这样做。使用尽可能简单的设计使系统首先运行。然后如果速度不够快，请对其进行分析。你几乎总会发现“你的”瓶颈不是问题。节省时间，才是真正重要的东西。
3. **记住“分而治之”的原则**。如果所面临的问题太过混乱 **，就去想象一下程序的基本操作，因为存在一个处理困难部分的神奇“片段”（piece）。该“片段”是一个对象，编写使用该对象的代码，然后查看该对象并将其困难部分封装到其他对象中，等等。
4. **将类创建者与类用户（客户端程序员）分开**。类用户是“客户”，不需要也不想知道类幕后发生了什么。类创建者必须是设计类的专家，他们编写类，以便新手程序员都可以使用它，并仍然可以在应用程序中稳健地工作。将该类视为其他类的*服务提供者*（service provider）。只有对其它类透明，才能很容易地使用这个类。
5. **创建类时，给类起个清晰的名字，就算不需要注释也能理解这个类**。你的目标应该是使客户端程序员的接口在概念上变得简单。为此，在适当时使用方法重载来创建直观，易用的接口。

6. **你的分析和设计必须至少能够产生系统中的类、它们的公共接口以及它们与其他类的关系，尤其是基类**。如果你的设计方法产生的不止于此，就该问问自己，该方法生成的所有部分是否在程序的生命周期内都具有价值。如果不是，那么维护它们会很耗费精力。对于那些不会影响他们生产力的东西，开发团队的成员往往不会去维护，这是许多设计方法都没有考虑的生活现实。
7. **让一切自动化**。首先在编写类之前，编写测试代码，并将其与类保持一致。通过构建工具自动运行测试。你可能会使用事实上的标准 Java 构建工具 Gradle。这样，通过运行测试代码可以自动验证任何更改，将能够立即发现错误。因为你知道自己拥有测试框架的安全网，所以当发现需要时，可以更大胆地进行彻底的更改。请记住，语言的巨大改进来自内置的测试，包括类型检查，异常处理等，但这些内置功能很有限，你必须完成剩下的工作，针对具体的类或程序，去完善这些测试内容，从而创建一个强大的系统。
8. **在编写类之前，先编写测试代码，以验证类的设计是完善的**。如果不编写测试代码，那么就不知道类是什么样的。此外，通过编写测试代码，往往能够激发出类中所需的其他功能或约束。而这些功能或约束并不总是出现在分析和设计过程中。测试还会提供示例代码，显示了如何使用这个类。
9. **所有的软件设计问题，都可以通过引入一个额外的间接概念层次（extra level of conceptual indirection）来解决**。这个软件工程的基本规则[1]是抽象的基础，是面向对象编程的主要特征。在面向对象编程中，我们也可以这样说："如果你的代码太复杂，就要生成更多的对象。"
10. **间接（indirection）应具有意义（与准则 9 一致）**。这个含义可以简单到"将常用代码放在单个方法中。"如果添加没有意义的间接（抽象，封装等）级别，那么它就像没有足够的间接性那样糟糕。
11. **使类尽可能原子化**。为每个类提供一个明确的目的，它为其他类提供一致的服务。如果你的类或系统设计变得过于复杂，请将复杂类分解为更简单的类。最直观的指标是尺寸大小，如果一个类很大，那么它可能是做的事太多了，应该被拆分。建议重新设计类的线索是：
    - 一个复杂的 *switch* 语句：考虑使用多态。
    - 大量方法涵盖了很多不同类型的操作：考虑使用多个类。
    - 大量成员变量涉及很多不同的特征：考虑使用多个类。
    - 其他建议可以参见 Martin Fowler 的 *Refactoring: Improving the Design of Existing Code*（重构：改善既有代码的设计）（Addison-Wesley 1999）。
12. **注意长参数列表**。那样方法调用会变得难以编写，读取和维护。相反，尝试将方法移动到更合适的类，并且（或者）将对象作为参数传递。
13. **不要重复自己**。如果一段代码出现在派生类的许多方法中，则将该代码放入基类中的单个方法中，并从派生类方法中调用它。这样不仅可以节省代码空间，而且可以轻松地传播更改。有时，发现这个通用代码会为接口添加有价值的功能。此指南的更简单版本也可以在没有继承的情况下发生：如果类具有重复代码的方法，则将该重复代码放入一个公共方，法并在其他方法中调用它。
14. **注意 *switch* 语句或链式 *if-else* 子句**。一个*类型检查编码*（type-check coding）的指示器意味着需要根据某种类型信息选择要执行的代码（确切的类型最初可能不明显）。很多时候可以用继

[1]在某些平台上，特别是 Windows，默认值可能非常难以查明。你可以使用 -Xss 标志调整堆栈大小。

对于每一个不利条件，都会存在与之相对应的有利条件。

承和多态替换这种代码，多态方法调用将会执行类型检查，并提供了更可靠和更容易的可扩展性。

15. **从设计的角度，寻找和分离那些因不变的事物而改变的事物**。也就是说，在不强制重新设计的情况下搜索可能想要更改的系统中的元素，然后将这些元素封装在类中。
16. **不要通过子类扩展基本功能**。如果一个接口元素对于类来说是必不可少的，则它应该在基类中，而不是在派生期间添加。如果要在继承期间添加方法，请考虑重新设计。
17. **少即是多**。从一个类的最小接口开始，尽可能小而简单，以解决手头的问题，但不要试图预测类的所有使用方式。在使用该类时，就将会了解如何扩展接口。但是，一旦这个类已经在使用了，就无法在不破坏客户端代码的情况下缩小接口。如果必须添加更多方法，那很好，它不会破坏代码。但即使新方法取代旧方法的功能，也只能是保留现有接口（如果需要，可以结合底层实现中的功能）。如果必须通过添加更多参数来扩展现有方法的接口，请使用新参数创建重载方法，这样，就不会影响到对现有方法的任何调用。
18. **大声读出你的类以确保它们合乎逻辑**。将基类和派生类之间的关系称为“is-a”，将成员对象称为“has-a”。
19. **在需要在继承和组合之间作决定时，问一下自己是否必须向上转换为基类型**。如果不是，则使用组合（成员对象）更好。这可以消除对多种基类型的感知需求（perceived need）。如果使用继承，则用户会认为他们应该向上转型。
20. **注意重载**。方法不应该基于参数的值而有条件地执行代码。在这里，应该创建两个或多个重载方法。
21. **使用异常层次结构**，最好是从标准 Java 异常层次结构中的特定适当类派生。然后，捕获异常的人可以为特定类型的异常编写处理程序，然后为基类型编写处理程序。如果添加新的派生异常，现有客户端代码仍将通过基类型捕获异常。
22. **有时简单的聚合可以完成工作**。航空公司的“乘客舒适系统”由独立的元素组成：座位，空调，影视等，但必须在飞机上创建许多这样的元素。你创建私有成员并建立一个全新的接口了吗？如果不是，在这种情况下，组件也应该是公共接口的一部分，因此应该创建公共成员对象。这些对象有自己的私有实现，这些实现仍然是安全的。请注意，简单聚合不是经常使用的解决方案，但确实会有时候会用到。
23. **考虑客户程序员和维护代码的人的观点**。设计类以便尽可能直观地被使用。预测要进行的更改，并精心设计类，以便轻松地进行更改。
24. **注意“巨型对象综合症”**（giant object syndrome）。这通常是程序员的痛苦，他们是面向对象编程的新手，总是编写面向过程程序并将其粘贴在一个或两个巨型对象中。除应用程序框架外，对象代表应用程序中的概念，而不是应用程序本身。
25. **如果你必须做一些丑陋的事情，至少要把类内的丑陋本地化**。
26. **如果必须做一些不可移植的事情，那就对这个事情做一个抽象，并在一个类中进行本地化**。这种额外的间接级别可防止在整个程序中扩散这种不可移植性。（这个原则也体现在*桥接*模式中，等等）。

27. **对象不应该仅仅只是持有一些数据**。它们也应该有明确的行为。有时候，“数据传输对象”（data transfer objects）是合适的，但只有在泛型集合不合适时，才被明确用于打包和传输一组元素。
28. **在从现有类创建新类时首先选择组合**。仅在设计需要时才使用继承。如果在可以使用组合的地方使用继承，那么设计将会变得很复杂，这是没必要的。
29. **使用继承和覆盖方法来表达行为的差异，而不是使用字段来表示状态的变化**。如果发现一个类使用了状态变量，并且有一些方法是基于这些变量切换行为的，那么请重新设计它，以表示子类和覆盖方法中的行为差异。一个极端的反例是继承不同的类来表示颜色，而不是使用“颜色”字段。
30. **注意*协变*（variance）**。两个语义不同的对象可能具有相同的操作或职责。为了从继承中受益，会试图让其中一个成为另一个的子类，这是一种很自然的诱惑。这称为协变，但没有真正的理由去强制声明一个并不存在的父子类关系。更好的解决方案是创建一个通用基类，并为两者生成一个接口，使其成为这个通用基类的派生类。这仍然可以从继承中受益，并且这可能是关于设计的一个重要发现。
31. **在继承期间注意*限定*（limitation）**。最明确的设计为继承的类增加了新的功能。含糊的设计在继承期间删除旧功能而不添加新功能。但是规则是用来打破的，如果是通过调用一个旧的类库来工作，那么将一个现有类限制在其子类型中，可能比重构层次结构更有效，因此新类适合在旧类的上层。
32. **使用设计模式来消除“裸功能”（naked functionality）**。也就是说，如果类只需要创建一个对象，请不要推进应用程序并写下注释“只生成一个。”应该将其包装成一个单例（singleton）。如果主程序中有很多乱七八糟的代码去创建对象，那么找一个像工厂方法一样的创建模式，可以在其中封装创建过程。消除“裸功能”不仅会使代码更易于理解和维护，而且还会使其能够更加防范应对后面的善意维护者（well-intentioned maintainers）。
33. **注意“分析瘫痪”（analysis paralysis）**。记住，不得不经常在不了解整个项目的情况下推进项目，并且通常了解那些未知因素的最好、最快的方式是进入下一步而不是尝试在脑海中弄清楚。在获得解决方案之前，往往无法知道解决方案。Java 有内置的防火墙，让它们为你工作。你在一个类或一组类中的错误不会破坏整个系统的完整性。
34. **如果认为自己有很好的分析，设计或实施，请做一个演练**。从团队外部带来一些人，不一定是顾问，但可以是公司内其他团体的人。用一双新眼睛评审你的工作，可以在一个更容易修复它们的阶段发现问题，而不仅仅是把大量时间和金钱全扔到演练过程中。

2.2 实现

36. **遵循编码惯例**。有很多不同的约定，例如，谷歌使用的约定（本书中的代码尽可能地遵循这些约定）。如果坚持使用其他语言的编码风格，那么读者就会很难去阅读。无论决定采用何种编码约定，都要确保它们在整个项目中保持一致。集成开发环境通常包含内置的重新格式化（reformatter）和检查器（checker）。

37. **无论使用何种编码风格，如果你的团队（甚至更好是公司）对其进行标准化，它就确实会产生重大影响**。这意味着，如果不符合这个标准，那么每个人都认为修复别人的编码风格是公平的游戏。标准化的价值在于解析代码可以花费较少的脑力，因此可以更专注于代码的含义。
38. **遵循标准的大写规则**。类名的第一个字母大写。字段，方法和对象（引用）的第一个字母应为小写。所有标识符应该将各个单词组合在一起，并将所有中间单词的首字母大写。例如：
    - **ThisIsAClassName**
    - **thisIsAMethodOrFieldName**

    将 **static final** 类型的标识符的所有字母全部大写，并用下划线分隔各个单词，这些标识符在其定义中具有常量初始值。这表明它们是编译时常量。
    - **包是一个特例**，它们都是小写的字母，即使是中间词。域扩展（com，org，net，edu 等）也应该是小写的。这是 Java 1.1 和 Java 2 之间的变化。
39. **不要创建自己的"装饰"私有字段名称**。这通常以前置下划线和字符的形式出现。匈牙利命名法（译者注：一种命名规范，基本原则是：变量名 = 属性 + 类型 + 对象描述。Win32 程序风格采用这种命名法，如`WORD wParam1;LONG lParam2;HANDLE hInstance`）是最糟糕的例子，你可以在其中附加额外字符用于指示数据类型，用途，位置等，就好像你正在编写汇编语言一样，编译器根本没有提供额外的帮助。这些符号令人困惑，难以阅读，并且难以执行和维护。让类和包来指定名称范围。如果认为必须装饰名称以防止混淆，那么代码就可能过于混乱，这应该被简化。
40. 在创建一般用途的类时，**遵循"规范形式"**。包括 **equals()**，**hashCode()**，**toString()**，**clone()** 的定义（实现 **Cloneable**，或选择其他一些对象复制方法，如序列化），并实现 **Comparable** 和 **Serializable**。
41. **对读取和更改私有字段的方法使用"get"，"set"和"is"命名约定**。这种做法不仅使类易于使用，而且也是命名这些方法的标准方法，因此读者更容易理解。
42. **对于所创建的每个类，请包含该类的 JUnit 测试**（请参阅 *junit.org* 以及第十六章：代码校验中的示例）。无需删除测试代码即可在项目中使用该类，如果进行更改，则可以轻松地重新运行测试。测试代码也能成为如何使用这个类的示例。
43. **有时需要继承才能访问基类的 protected 成员**。这可能导致对多种基类型的感知需求（perceived need）。如果不需要向上转型，则可以首先派生一个新类来执行受保护的访问。然后把该新类作为使用它的任何类中的成员对象，以此来代替直接继承。
44. **为了提高效率，避免使用 *final* 方法**。只有在分析后发现方法调用是瓶颈时，才将 **final** 用于此目的。
45. **如果两个类以某种功能方式相互关联（例如集合和迭代器），则尝试使一个类成为另一个类的内部类**。这不仅强调了类之间的关联，而且通过将类嵌套在另一个类中，可以允许在单个包中重用类名。Java 集合库通过在每个集合类中定义内部 **Iterator** 类来实现此目的，从而为集合提供通用接口。使用内部类的另一个原因是作为**私有**实现的一部分。这里，内部类将有利于实现隐藏，而不是上面提到的类关联和防止命名空间污染。

46. **只要你注意到类似乎彼此之间具有高耦合，请考虑如果使用内部类可能获得的编码和维护改进**。内部类的使用不会解耦类，而是明确耦合关系，并且更方便。
47. **不要成为过早优化的牺牲品**。过早优化是很疯狂的行为。特别是，不要担心编写（或避免）本机方法（native methods），将某些方法设置为 **final**，或者在首次构建系统时调整代码以使其高效。你的主要目标应该是验证设计。即使设计需要一定的效率，也*先让它工作，然后再让它变快*。
48. **保持作用域尽可能小，以便能见度和对象的寿命尽可能小**。这减少了在错误的上下文中使用对象并隐藏了难以发现的 bug 的机会。例如，假设有一个集合和一段迭代它的代码。如果复制该代码以用于一个新集合，那么可能会意外地将旧集合的大小用作新集合的迭代上限。但是，如果旧集合比较大，则会在编译时捕获错误。
49. **使用标准 Java 库中的集合**。熟练使用它们，将会大大提高工作效率。首选 **ArrayList** 用于序列，**HashSet** 用于集合，**HashMap** 用于关联数组，**LinkedList** 用于堆栈（而不是 **Stack**，尽管也可以创建一个适配器来提供堆栈接口）和队列（也可以使用适配器，如本书所示）。当使用前三个时，将其分别向上转型为 **List**，**Set** 和 **Map**，那么就可以根据需要轻松更改为其他实现。
50. **为使整个程序健壮，每个组件必须健壮**。在所创建的每个类中，使用 Java 所提供的所有工具，如访问控制，异常，类型检查，同步等。这样，就可以在构建系统时安全地进入下一级抽象。
51. **编译时错误优于运行时错误**。尝试尽可能在错误发生点处理错误。在最近的处理程序中尽其所能地捕获它能处理的所有异常。在当前层面处理所能处理的所有异常，如果解决不了，就重新抛出异常。
52. **注意长方法定义**。方法应该是简短的功能单元，用于描述和实现类接口的离散部分。维护一个冗长而复杂的方法是很困难的，而且代价很大，并且这个方法可能是试图做了太多事情。如果看到这样的方法，这表明，至少应该将它分解为多种方法。也可能建议去创建一个新类。小的方法也可以促进类重用。（有时方法必须很大，但它们应该只做一件事。）
53. **保持“尽可能私有”**。一旦公开了你的类库中的一个方面（一个方法，一个类，一个字段），你就永远无法把它拿回来。如果这样做，就将破坏某些人的现有代码，迫使他们重写和重新设计。如果你只公开了必须公开的内容，就可以轻易地改变其他一切，而不会对其他人造成影响，而且由于设计趋于发展，这是一个重要的自由。通过这种方式，更改具体实现将对派生类造成的影响最小。在处理多线程时，私有尤其重要，只有**私有**字段可以防止不同步使用。具有包访问权限的类应该仍然具有**私有**字段，但通常有必要提供包访问权限的方法而不是将它们**公开**。
54. **大量使用注释，并使用 *Javadoc commentdocumentation* 语法生成程序文档**。但是，注释应该为代码增加真正的意义，如果注释只是重申了代码已经清楚表达的内容，这是令人讨厌的。请注意，Java 类和方法名称的典型详细信息减少了对某些注释的需求。
55. **避免使用“魔法数字”**。这些是指硬编码到代码中的数字。如果后续必须要更改它们，那将是一场噩梦，因为你永远不知道“100”是指“数组大小”还是“完全不同的东西”。相反，创建一个带有描述性名称的常量并在整个程序中使用常量标识符。这使程序更易于理解，更易于维护。
56. **在创建构造方法时，请考虑异常**。最好的情况是，构造方法不会做任何抛出异常的事情。次一级

的最佳方案是，该类仅由健壮的类组成或继承自健壮的类，因此如果抛出异常则不需要处理。否则，必须清除 **finally** 子句中的组合类。如果构造方法必然失败，则适当的操作是抛出异常，因此调用者不会认为该对象是正确创建的而盲目地继续下去。

57. **在构造方法内部，只需要将对象设置为正确的状态**。主动避免调用其他方法（**final** 方法除外），因为这些方法可以被其他人覆盖，从而在构造期间产生意外结果。（有关详细信息，请参阅第六章：初始化和清理章节。）较小，较简单的构造方法不太可能抛出异常或导致问题。
58. **如果类在客户端程序员用完对象时需要进行任何清理，请将清理代码放在一个明确定义的方法中**，并使用像 **dispose()** 这样的名称来清楚地表明其目的。另外，在类中放置一个 **boolean** 标志来指示是否调用了 **dispose()**，因此 **finalize()** 可以检查"终止条件"（参见第六章：初始化和清理章节）。
59. ***finalize()* 的职责只能是验证对象的"终止条件"以进行调试**。（参见第六章：初始化和清理一章）在特殊情况下，可能需要释放垃圾收集器无法释放的内存。因为可能无法为对象调用垃圾收集器，所以无法使用 **finalize()** 执行必要的清理。为此，必须创建自己的 **dispose()** 方法。在类的 **finalize()** 方法中，检查以确保对象已被清理，如果没有被清理，则抛出一个派生自 **RuntimeException** 的异常，以指示编程错误。在依赖这样的计划之前，请确保 **finalize()** 适用于你的系统。（可能需要调用 **System.gc()** 来确保此行为。）
60. **如果必须在特定范围内清理对象（除了通过垃圾收集器），请使用以下准则：**初始化对象，如果成功，立即进入一个带有 **finally** 子句的 **try** 块，并在 **finally** 中执行清理操作。
61. **在继承期间覆盖 *finalize()* 时，记得调用 *super.finalize()***。（如果是直接继承自 **Object** 则不需要这样做。）调用 **super.finalize()** 作为重写的 **finalize()** 的最终行为而不是在第一行调用它，这样可以确保基类组件在需要时仍然有效。
62. **创建固定大小的对象集合时，将它们转换为数组**，尤其是在从方法中返回此集合时。这样就可以获得数组编译时类型检查的好处，并且数组的接收者可能不需要在数组中强制转换对象来使用它们。请注意，集合库的基类 **java.util.Collection** 有两个 **toArray()** 方法来完成此任务。
63. **优先选择 *接口*而不是 *抽象类***。如果知道某些东西应该是基类，那么第一选择应该是使其成为一个接口，并且只有在需要方法定义或成员变量时才将其更改为抽象类。一个接口关心客户端想要做什么，而一个类倾向于关注（或允许）实现细节。
64. **为了避免非常令人沮丧的经历，请确保类路径中的每个名称只对应一个不在包中的类**。否则，编译器可以首先找到具有相同名称的其他类，并报告没有意义的错误消息。如果你怀疑自己有类路径问题，请尝试在类路径的每个起始点查找具有相同名称的 **.class** 文件。理想情况下，应该将所有类放在包中。
65. **注意意外重载**。如果尝试覆盖基类方法但是拼写错误，则最终会添加新方法而不是覆盖现有方法。但是，这是完全合法的，因此在编译时或运行时不会获得任何错误消息，但代码将无法正常工作。始终使用 **@Override** 注释来防止这种情况。
66. **注意过早优化**。先让它工作，然后再让它变快。除非发现代码的特定部分存在性能瓶颈。除非是

使用分析器发现瓶颈，否则过早优化会浪费时间。性能调整所隐藏的额外成本是代码将变得难以理解和维护。

67. **请注意，相比于编写代码，代码被阅读的机会更多**。清晰的设计可能产生易于理解的程序，但注释，详细解释，测试和示例是非常宝贵的，它们可以帮助你和你的所有后继者。如果不出意外，试图从 JDK 文档中找出有用信息的挫败感应该可以说服你。

附录 3

文档注释

编写代码文档的最大问题可能是维护该文档。如果文档和代码是分开的，那么每次更改代码时更改文档都会变得很繁琐。解决方案似乎很简单：将代码链接到文档。最简单的方法是将所有内容放在同一个文件中。然而，要完成这完整的画面，您需要一个特殊的注释语法来标记文档，以及一个工具来将这些注释提取为有用的表单中。这就是 Java 所做的。

提取注释的工具称为 Javadoc，它是 JDK 安装的一部分。它使用 Java 编译器中的一些技术来寻找特殊的注释标记。它不仅提取由这些标记所标记的信息，还提取与注释相邻的类名或方法名。通过这种方式，您就可以用最少的工作量来生成合适的程序文档。

Javadoc 输出为一个 html 文件，您可以使用 web 浏览器查看它。对于 Javadoc，您有一个简单的标准来创建文档，因此您可以期望所有 Java libraries 都有文档。

此外，您可以编写自己的 Javadoc 处理程序 doclet，对于 Javadoc（例如，以不同的格式生成输出）。

以下是对 Javadoc 基础知识的介绍和概述。在 JDK 文档中可以找到完整的描述。

3.1 句法规则

所有 Javadoc 指令都发生在以 /** 开头 (但仍然以 */ 结尾) 的注释中。

使用 Javadoc 有两种主要方法:

嵌入 HTML 或使用“doc 标签”。独立的 doc 标签是指令它以 **@** 开头，放在注释行的开头。(然而，前面的 *** 将被忽略。) 可能会出现内联 doc 标签

Javadoc 注释中的任何位置，也可以，以一个 **@** 开头，但是被花括号包围。

有三种类型的注释文档，它们对应于注释前面的元素: 类、字段或方法。也就是说，类注释出现在类定义之前，字段注释出现在字段定义之前，方法注释出现在方法定义之前。举个简单的例子:

```
// javadoc/Documentation1.java
/** 一个类注释 */
public class Documentation1 {
```

```
    /** 一个属性注释 */
    public int i;
    /** 一个方法注释 */
    public void f() {}
}
```

Javadoc 处理注释文档仅适用于 **公共**和 **受保护**的成员。

默认情况下，将忽略对 **私有成员** 和包访问成员的注释（请参阅“隐藏实现”一章），并且您将看不到任何输出。

这是有道理的，因为仅客户端程序员的观点是，在文件外部可以使用 **公共成员** 和 **受保护成员**。您可以使用 **-private** 标志和包含 **私人**成员。

要通过 Javadoc 处理前面的代码，命令是：

```
javadoc Documentation1.java
```

这将产生一组 HTML 文件。如果您在浏览器中打开 index.html，您将看到结果与所有其他 Java 文档具有相同的标准格式，因此用户对这种格式很熟悉，并可以轻松地浏览你的类。

3.2 内嵌 HTML

Javadoc 传递未修改的 HTML 代码，用以生成的 HTML 文档。这使你可以充分利用 HTML。但是，这样做的主要目的是让你格式化代码，例如：

```
// javadoc/Documentation2.java
/** <pre>
* System.out.println(new Date());
* </pre>
*/
public class Documentation2 {}
```

您你也可以像在其他任何 Web 文档中一样使用 HTML 来格式化说明中的文字：

```
// javadoc/Documentation3.java
/** You can <em>even</em> insert a list:
* <ol>
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
public class Documentation3 {}
```

请注意，在文档注释中，Javadoc 删除了行首的星号以及前导空格。Javadoc 重新格式化了所有内容，使其符合标准文档的外观。不要将诸如 <h1> 或 <hr> 之类的标题用作嵌入式 HTML，因为 Javadoc 会插入自己的标题，后插入的标题将对其生成的文档产生干扰。

所有类型的注释文档（类，字段和方法）都可以支持嵌入式 HTML。

3.3 示例标签

以下是一些可用于代码文档的 Javadoc 标记。在尝试使用 Javadoc 进行任何认真的操作之前，请查阅 JDK 文档中的 Javadoc 参考，以了解使用 Javadoc 的所有不同方法。

3.3.1 @see

这个标签可以将其他的类连接到文档中，Javadoc 将使用 @see 标记超链接到其他文档中，形式为：

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

每个都向生成的文档中添加超链接的“另请参阅”条目。Javadoc 不会检查超链接的有效性。

3.3.2 {@link package.class#member label}

和 @see 非常相似，不同之处在于它可以内联使用，并使用标签作为超链接文本，而不是“另请参阅”。

3.3.3 {@docRoot}

生成文档根目录的相对路径。对于显式超链接到文档树中的页面很有用。

3.3.4 {@inheritDoc}

将文档从此类的最近基类继承到当前文档注释中。

3.3.5 @version

其形式为：

```
@version version-information
```

其中 version-information 是你认为适合包含的任何重要信息。当在 Javadoc 命令行上放置 -version 标志时，特别在生成的 HTML 文档中用于生成 version 信息。

3.3.6 @author

其形式为:

```
@author author-information
```

author-information 大概率是你的名字，但是一样可以包含你的 email 地址或者其他合适的信息。当在 Javadoc 命令行上放置 -author 标志的时候，在生成的 HTML 文档中特别注明了作者信息。

你可以对作者列表使用多个作者标签，但是必须连续放置它们。所有作者信息都集中在生成的 HTML 中的单个段落中。

3.3.7 @since

此标记指示此代码的版本开始使用特定功能。例如，它出现在 HTML Java 文档中，以指示功能首次出现的 JDK 版本。

3.3.8 @param

这将生成有关方法参数的文档:

```
@param parameter-name description
```

其中 parameter-name 是方法参数列表中的标识符，description 是可以在后续行中继续的文本。当遇到新的文档标签时，说明被视为完成。@param 标签的可以任意使用，大概每个参数一个。

3.3.9 @return

这记录了返回值:

```
@return description
```

其中 description 给出了返回值的含义。它可延续到后面的行内。

3.3.10 @throws

一个方法可以产生许多不同类型的异常，所有这些异常都需要描述。异常标记的形式为:

```
@throws fully-qualified-class-name description
```

fully-qualified-class-name 给出明确的异常分类名称，并且 description（可延续到后面的行内）告诉你为什么这特定类型的异常会在方法调用后出现。

3.3.11 @deprecated

这表示已被改进的功能取代的功能。deprecated 标记表明你不再使用此特定功能，因为将来有可能将其删除。标记为 @ 不赞成使用的方法会导致编译器在使用时发出警告。在 Java 5 中，@deprecated Javadoc 标记已被 @Deprecated 注解取代（在注解一章中进行了描述）。

3.4 文档示例

objects/HelloDate.java 是带有文档注释的例子。

```
// javadoc/HelloDateDoc.java
import java.util.*;
/** The first On Java 8 example program.
 * Displays a String and today's date.
 * @author Bruce Eckel
 * @author www.MindviewInc.com
 * @version 5.0
 */
public class HelloDateDoc {
    /** Entry point to class & application.
     * @param args array of String arguments
     * @throws exceptions No exceptions thrown
     */
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
}
/* Output:
Hello, it's:
Tue May 09 06:07:27 MDT 2017
*/
```

你可以在 Java 标准库的源代码中找到许多 Javadoc 注释文档的示例。

附录 4

对象传递和返回

> 到现在为止，你已经对“传递”对象实际上是传递引用这一想法想法感到满意。

在许多编程语言中，你可以使用该语言的“常规”方式来传递对象，并且大多数情况下一切正常。但是通常会出现这种情况，你必须做一些不平常的事情，突然事情变得更加复杂。Java 也不例外，当您传递对象并对其进行操作时，准确了解正在发生的事情很重要。本附录提供了这种见解。

提出本附录问题的另一种方法是，如果你之前使用类似 C++ 的编程语言，则是 “ Java 是否有指针？ ” Java 中的每个对象标识符（除原语外）都是这些指针之一，但它们的用法是不仅受编译器的约束，而且受运行时系统的约束。换一种说法，Java 有指针，但没有指针算法。这些就是我一直所说的“引用”，您可以将它们视为“安全指针”，与小学的安全剪刀不同-它们不敏锐，因此您不费吹灰之力就无法伤害自己，但是它们有时可能很乏味。

4.1 传递引用

当你将引用传递给方法时，它仍指向同一对象。一个简单的实验演示了这一点：

```
// references/PassReferences.java
public class PassReferences {
public static void f(PassReferences h) {
        System.out.println("h inside f(): " + h);
    }
    public static void main(String[] args) {
        PassReferences p = new PassReferences();
        System.out.println("p inside main(): " + p);
        f(p);
    }
}
/* Output:
p inside main(): PassReferences@15db9742
h inside f(): PassReferences@15db9742
*/
```

方法 `toString()` 在打印语句中自动调用，并且 `PassReferences` 直接从 `Object` 继承而无需重新定义 `toString ()` 。因此，使用的是 `Object` 的 `toString ()` 版本，它打印出对象的类，然后打印出该对象所在的地址（不是引用，而是实际的对象存储）。

4.2 本地拷贝

4.3 控制克隆

4.4 不可变类

4.5 本章小结

附录 5

流式 IO

> Java 7 引入了一种简单明了的方式来读写文件和操作目录。大多情况下，文件这一章所介绍的那些库和技术就足够你用了。但是，如果你必须面对一些特殊的需求和比较底层的操作，或者处理一些老版本的代码，那么你就必须了解本附录中的内容。

对于编程语言的设计者来说，实现良好的输入/输出（I/O）系统是一项比较艰难的任务，不同实现方案的数量就可以证明这点。其中的挑战似乎在于要涵盖所有的可能性，你不仅要覆盖到不同的 I/O 源和 I/O 接收器（如文件、控制台、网络连接等），还要实现多种与它们进行通信的方式（如顺序、随机访问、缓冲、二进制、字符、按行和按字等）。

Java 类库的设计者通过创建大量的类来解决这一难题。一开始，你可能会对 Java I/O 系统提供了如此多的类而感到不知所措。Java 1.0 之后，Java 的 I/O 类库发生了明显的改变，在原来面向字节的类中添加了面向字符和基于 Unicode 的类。在 Java 1.4 中，为了改进性能和功能，又添加了 `nio` 类（全称是 “new I/O”，Java 1.4 引入，到现在已经很多年了）。这部分在附录：新 I/O 中介绍。

因此，要想充分理解 Java I/O 系统以便正确运用它，我们需要学习一定数量的类。另外，理解 I/O 类库的演化过程也很有必要，因为如果缺乏历史的眼光，很快我们就会对什么时候该使用哪些类，以及什么时候不该使用它们而感到困惑。

编程语言的 I/O 类库经常使用**流**这个抽象概念，它将所有数据源或者数据接收器表示为能够产生或者接收数据片的对象。

> **注意**：Java 8 函数式编程中的 `Stream` 类和这里的 I/O stream 没有任何关系。这又是另一个例子，如果再给设计者一次重来的机会，他们将使用不同的术语。

I/O 流屏蔽了实际的 I/O 设备中处理数据的细节：

1. 字节流对应原生的二进制数据；
2. 字符流对应字符数据，它会自动处理与本地字符集之间的转换；
3. 缓冲流可以提高性能，通过减少底层 API 的调用次数来优化 I/O。

从 JDK 文档的类层次结构中可以看到，Java 类库中的 I/O 类分成了输入和输出两部分。在设计 Java 1.0 时，类库的设计者们就决定让所有与输入有关系的类都继承自 `InputStream`，所有与输

出有关系的类都继承自 OutputStream。所有从 InputStream 或 Reader 派生而来的类都含有名为 read() 的基本方法，用于读取单个字节或者字节数组。同样，所有从 OutputStream 或 Writer 派生而来的类都含有名为 write() 的基本方法，用于写单个字节或者字节数组。但是，我们通常不会用到这些方法，它们之所以存在是因为别的类可以使用它们，以便提供更有用的接口。

我们很少使用单一的类来创建流对象，而是通过叠合多个对象来提供所期望的功能（这是**装饰器设计模式**）。为了创建一个流，你却要创建多个对象，这也是 Java I/O 类库让人困惑的主要原因。

这里我只会提供这些类的概述，并假定你会使用 JDK 文档来获取它们的详细信息（比如某个类的所以方法的详细列表）。

5.1 输入流类型

InputStream 表示那些从不同数据源产生输入的类，如表 I/O-1 所示，这些数据源包括：

1. 字节数组；
2. String 对象；
3. 文件；
4. “管道”，工作方式与实际生活中的管道类似：从一端输入，从另一端输出；
5. 一个由其它种类的流组成的序列，然后我们可以把它们汇聚成一个流；
6. 其它数据源，如 Internet 连接。

每种数据源都有相应的 InputStream 子类。另外，FilterInputStream 也属于一种 InputStream，它的作用是为“装饰器”类提供基类。其中，“装饰器”类可以把属性或有用的接口与输入流连接在一起，这个我们稍后再讨论。

表 I/O-1 InputStream 类型

类	功能	构造器参数	如何使用
ByteArrayInputStream	允许将内存的缓冲区当做 InputStream 使用	缓冲区，字节将从中取出	作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口
StringBufferInputStream	将 String 转换成 InputStream	字符串。底层实现实际使用 StringBuffer	作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口

类	功能	构造器参数	如何使用
FileInputStream	用于从文件中读取信息	字符串，表示文件名、文件或 FileDescriptor 对象	作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口
PipedInputStream	产生用于写入相关 PipedOutputStream 的数据。实现“管道化”概念	PipedOutputSteam	作为多线程中的数据源：将其与 FilterInputStream 对象相连以提供有用接口
SequenceInputStream	将两个或多个 InputStream 对象转换成一个 InputStream	两个 InputStream 对象或一个容纳 InputStream 对象的容器 Enumeration	作为一种数据源：将其与 FilterInputStream 对象相连以提供有用接口
FilterInputStream	抽象类，作为“装饰器”的接口。其中，“装饰器”为其它的 InputStream 类提供有用的功能。见表 I/O-3	见表 I/O-3	见表 I/O-3

5.2 输出流类型

如表 I/O-2 所示，该类别的类决定了输出所要去往的目标：字节数组（但不是 String，当然，你也可以用字节数组自己创建）、文件或管道。

另外，FilterOutputStream 为“装饰器”类提供了一个基类，“装饰器”类把属性或者有用的接口与输出流连接了起来，这些稍后会讨论。

表 I/O-2：OutputStream 类型

类	功能	构造器参数	如何使用
ByteArrayOutputStream	在内存中创建缓冲区。所有送往“流”的数据都要放置在此缓冲区	缓冲区初始大小（可选）	用于指定数据的目的地：将其与 FilterOutputStream 对象相连以提供有用接口
FileOutputStream	用于将信息写入文件	字符串，表示文件名、文件或 FileDescriptor 对象	用于指定数据的目的地：将其与 FilterOutputStream 对象相连以提供有用接口
PipedOutputStream	任何写入其中的信息都会自动作为相关 PipedInputStream 的输出。实现“管道化”概念	PipedInputStream	指定用于多线程的数据的目的地：将其与 FilterOutputStream 对象相连以提供有用接口
FilterOutputStream	抽象类，作为“装饰器”的接口。其中，“装饰器”为其它 OutputStream 提供有用功能。见表 I/O-4	见表 I/O-4	见表 I/O-4

5.3 添加属性和有用的接口

装饰器在泛型这一章引入。Java I/O 类库需要多种不同功能的组合，这正是使用装饰器模式的原因所在[1]。而之所以存在 **filter**（过滤器）类，是因为让抽象类 **filter** 作为所有装饰器类的基类。装饰器必须具有和它所装饰对象相同的接口，但它也可以扩展接口，不过这种情况只发生在个别 **filter** 类中。

但是，装饰器模式也有一个缺点：在编写程序的时候，它给我们带来了相当多的灵活性（因为我们可以很容易地对属性进行混搭），但它同时也增加了代码的复杂性。Java I/O 类库操作不便的原因在于：我们必须创建许多类（“核心” I/O 类型加上所有的装饰器）才能得到我们所希望的单个 I/O 对象。

FilterInputStream 和 FilterOutputStream 是用来提供装饰器类接口以控制特定输入流

[1]在某些平台上，特别是 Windows，默认值可能非常难以查明。你可以使用 -Xss 标志调整堆栈大小。

InputStream 和输出流 OutputStream 的两个类,但它们的名字并不是很直观。FilterInputStream 和 FilterOutputStream 分别从 I/O 类库中的基类 InputStream 和 OutputStream 派生而来，这两个类是创建装饰器的必要条件（这样它们才能为所有被装饰的对象提供统一接口）。

5.3.1 通过 FilterInputStream 从 InputStream 读取

FilterInputStream 类能够完成两件截然不同的事情。其中，DataInputStream 允许我们读取不同的基本数据类型和 String 类型的对象（所有方法都以 “read” 开头，例如 readByte()、readFloat()等等）。搭配其对应的 DataOutputStream，我们就可以通过数据“流”将基本数据类型的数据从一个地方迁移到另一个地方。具体是那些“地方”是由表 I/O-1 中的那些类决定的。

其它 FilterInputStream 类则在内部修改 InputStream 的行为方式：是否缓冲，是否保留它所读过的行（允许我们查询行数或设置行数），以及是否允许把单个字符推回输入流等等。最后两个类看起来就像是为了创建编译器提供的（它们被添加进来可能是为了对“用 Java 构建编译器”实现提供支持），因此我们在一般编程中不会用到它们。

在实际应用中，不管连接的是什么 I/O 设备，我们基本上都会对输入进行缓冲。所以当初 I/O 类库如果能默认都让输入进行缓冲，同时将无缓冲输入作为一种特殊情况（或者只是简单地提供一个方法调用），这样会更加合理，而不是像现在这样迫使我们基本上每次都得手动添加缓冲。

表 I/O-3：FilterInputStream 类型

类	功能	构造器参数	如何使用
DataInputStream	与 DataOutputStream 搭配使用，按照移植方式从流读取基本数据类型（int、char、long 等）	InputStream	包含用于读取基本数据类型的全部接口
BufferedInputStream	使用它可以防止每次读取时都得进行实际写操作。代表“使用缓冲区”	InputStream，可以指定缓冲区大小（可选）	本质上不提供接口，只是向进程添加缓冲功能。与接口对象搭配
LineNumberInputStream	跟踪输入流中的行号，可调用 getLineNumber() 和 setLineNumber(int)	InputStream	仅增加了行号，因此可能要与接口对象搭配使用

类	功能	构造器参数	如何使用
PushbackInputStream	具有能弹出一个字节的缓冲区，因此可以将读到的最后一个字符回退	InputStream	通常作为编译器的扫描器，我们可能永远也不会用到

5.3.2 通过 FilterOutputStream 向 OutputStream 写入

与 DataInputStream 对应的是 DataOutputStream，它可以将各种基本数据类型和 String 类型的对象格式化输出到“流”中,。这样一来，任何机器上的任何 DataInputStream 都可以读出它们。所有方法都以 “write” 开头，例如 writeByte()、writeFloat() 等等。

PrintStream 最初的目的就是为了以可视化格式打印所有基本数据类型和 String 类型的对象。这和 DataOutputStream 不同，后者的目的是将数据元素置入“流”中，使 DataInputStream 能够可移植地重构它们。

PrintStream 内有两个重要方法：print() 和 println()。它们都被重载了，可以打印各种各种数据类型。print() 和 println() 之间的差异是，后者在操作完毕后会添加一个换行符。

PrintStream 可能会造成一些问题，因为它捕获了所有 IOException（因此，我们必须使用 checkError() 自行测试错误状态，如果出现错误它会返回 **true**）。另外，PrintStream 没有处理好国际化问题。这些问题都在 PrintWriter 中得到了解决，这在后面会讲到。

BufferedOutputStream 是一个修饰符，表明这个“流”使用了缓冲技术，因此每次向流写入的时候，不是每次都会执行物理写操作。我们在进行输出操作的时候可能会经常用到它。

表 I/O-4：FilterOutputStream 类型

类	功能	构造器参数	如何使用
DataOutputStream	与 DataInputStream 搭配使用，因此可以按照移植方式向流中写入基本数据类型（**int**、**char**、**long** 等）	OutputStream	包含用于写入基本数据类型的全部接口

类	功能	构造器参数	如何使用
PrintStream	用于产生格式化输出。其中 DataOutputStream 处理数据的存储，PrintStream 处理显示	OutputStream，可以用 **boolean** 值指示是否每次换行时清空缓冲区（可选）	应该是对 OutputStream 对象的 **final** 封装。可能会经常用到它
BufferedOutputStream	使用它以避免每次发送数据时都进行实际的写操作。代表“使用缓冲区”。可以调用 flush() 清空缓冲区	OutputStream，可以指定缓冲区大小（可选）	本质上并不提供接口，只是向进程添加缓冲功能。与接口对象搭配

5.4 Reader 和 Writer

Java 1.1 对基本的 I/O 流类库做了重大的修改。你初次遇到 Reader 和 Writer 时，可能会以为这两个类是用来替代 InputStream 和 OutputStream 的，但实际上并不是这样。尽管一些原始的“流”类库已经过时了（如果使用它们，编译器会发出警告），但是 InputStream 和 OutputStream 在面向字节 I/O 这方面仍然发挥着极其重要的作用，而 Reader 和 Writer 则提供兼容 Unicode 和面向字符 I/O 的功能。另外：

1. Java 1.1 往 InputStream 和 OutputStream 的继承体系中又添加了一些新类，所以这两个类显然是不会被取代的；
2. 有时我们必须把来自“字节”层级结构中的类和来自“字符”层次结构中的类结合起来使用。为了达到这个目的，需要用到“适配器（adapter）类”：InputStreamReader 可以把 InputStream 转换为 Reader，而 OutputStreamWriter 可以把 OutputStream 转换为 Writer。

设计 Reader 和 Writer 继承体系主要是为了国际化。老的 I/O 流继承体系仅支持 8 比特的字节流，并且不能很好地处理 16 比特的 Unicode 字符。由于 Unicode 用于字符国际化（Java 本身的 **char** 也是 16 比特的 Unicode），所以添加 Reader 和 Writer 继承体系就是为了让所有的 I/O 操作都支持 Unicode。另外，新类库的设计使得它的操作比旧类库要快。

5.4.1 数据的来源和去处

几乎所有原始的 Java I/O 流类都有相应的 `Reader` 和 `Writer` 类来提供原生的 Unicode 操作。但是在某些场合，面向字节的 `InputStream` 和 `OutputStream` 才是正确的解决方案。特别是 `java.util.zip` 类库就是面向字节而不是面向字符的。因此，最明智的做法是尽量**尝试**使用 `Reader` 和 `Writer`，一旦代码没法成功编译，你就会发现此时应该使用面向字节的类库了。

下表展示了在两个继承体系中，信息的来源和去处（即数据物理上来自哪里又去向哪里）之间的对应关系：

来源与去处：Java 1.0 类	相应的 Java 1.1 类
`InputStream`	`Reader` 适配器：`InputStreamReader`
`OutputStream`	`Writer` 适配器：`OutputStreamWriter`
`FileInputStream`	`FileReader`
`FileOutputStream`	`FileWriter`
`StringBufferInputStream`（已弃用）	`StringReader`
（无相应的类）	`StringWriter`
`ByteArrayInputStream`	`CharArrayReader`
`ByteArrayOutputStream`	`CharArrayWriter`
`PipedInputStream`	`PipedReader`
`PipedOutputStream`	`PipedWriter`

总的来说，这两个不同的继承体系中的接口即便不能说完全相同，但也是非常相似的。

5.4.2 更改流的行为

对于 `InputStream` 和 `OutputStream` 来说，我们会使用 `FilterInputStream` 和 `FilterOutputStream` 的装饰器子类来修改“流”以满足特殊需要。`Reader` 和 `Writer` 的类继承体系沿用了相同的思想——但是并不完全相同。

在下表中，左右之间对应关系的近似程度现比上一个表格更加粗略一些。造成这种差别的原因是类的组织形式不同，`BufferedOutputStream` 是 `FilterOutputStream` 的子类，但 `BufferedWriter` 却不是 `FilterWriter` 的子类（尽管 `FilterWriter` 是抽象类，但却没有任何子类，把它放在表格里只是占个位置，不然你可能奇怪 `FilterWriter` 上哪去了）。然而，这些类的接口却又十分相似。

过滤器：Java 1.0 类	相应 Java 1.1 类
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (抽象类，没有子类)
BufferedInputStream	BufferedReader（也有 readLine())
BufferedOutputStream	BufferedWriter
DataInputStream	使用 DataInputStream（如果必须用到 readLine()，那你就得使用 BufferedReader。否则，一般情况下就用 DataInputStream
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer（使用具有 Reader 参数的构造器）
PushbackInputStream	PushbackReader

有一条限制需要明确：一旦要使用 readLine()，我们就不应该用 DataInputStream（否则，编译时会得到使用了过时方法的警告），而应该使用 BufferedReader。除了这种情况之外的情形中，DataInputStream 仍是 I/O 类库的首选成员。

为了使用时更容易过渡到 PrintWriter，它提供了一个既能接受 Writer 对象又能接受任何 OutputStream 对象的构造器。PrintWriter 的格式化接口实际上与 PrintStream 相同。

Java 5 添加了几种 PrintWriter 构造器，以便在将输出写入时简化文件的创建过程，你马上就会见到它们。

其中一种 PrintWriter 构造器还有一个执行**自动 flush**[2] 的选项。如果构造器设置了该选项，就会在每个 println() 调用之后，自动执行 flush。

5.4.3 未发生改变的类

有一些类在 Java 1.0 和 Java 1.1 之间未做改变。

[2]引自 Brian Goetz, Java Concurrency in Practice 一书的作者，该书由 Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea 联合著作 (Addison-Wesley 出版社, 2006)。

以下这些 Java 1.0 类在 Java 1.1 中没有相应类
DataOutputStream
File
RandomAccessFile
SequenceInputStream

特别是 DataOutputStream，在使用时没有任何变化；因此如果想以可传输的格式存储和检索数据，请用 InputStream 和 OutputStream 继承体系。

5.5 RandomAccessFile 类

RandomAccessFile 适用于由大小已知的记录组成的文件，所以我们可以使用 seek() 将文件指针从一条记录移动到另一条记录，然后对记录进行读取和修改。文件中记录的大小不一定都相同，只要我们能确定那些记录有多大以及它们在文件中的位置即可。

最初，我们可能难以相信 RandomAccessFile 不是 InputStream 或者 OutputStream 继承体系中的一部分。除了实现了 DataInput 和 DataOutput 接口（DataInputStream 和 DataOutputStream 也实现了这两个接口）之外，它和这两个继承体系没有任何关系。它甚至都不使用 InputStream 和 OutputStream 类中已有的任何功能。它是一个完全独立的类，其所有的方法（大多数都是 **native** 方法）都是从头开始编写的。这么做是因为 RandomAccessFile 拥有和别的 I/O 类型本质上不同的行为，因为我们可以在一个文件内向前和向后移动。在任何情况下，它都是自我独立的，直接继承自 Object。

从本质上来讲，RandomAccessFile 的工作方式类似于把 DataIunputStream 和 DataOutputStream 组合起来使用。另外它还有一些额外的方法，比如使用 getFilePointer() 可以得到当前文件指针在文件中的位置，使用 seek() 可以移动文件指针，使用 length() 可以得到文件的长度。另外，其构造器还需要传入第二个参数（和 C 语言中的 fopen() 相同）用来表示我们是准备对文件进行“随机读”（r）还是“读写”（rw）。它并不支持只写文件，从这点来看，如果当初 RandomAccessFile 能设计成继承自 DataInputStream，可能也是个不错的实现方式。

在 Java 1.4 中，RandomAccessFile 的大多数功能（但不是全部）都被 nio 中的**内存映射文件**（mmap）取代，详见附录：新 I/O。

5.6 IO 流典型用途

尽管我们可以用不同的方式来组合 I/O 流类，但常用的也就其中几种。你可以下面的例子可以作为 I/O 典型用法的基本参照（在你确定无法使用文件这一章所述的库之后）。

在这些示例中，异常处理都被简化为将异常传递给控制台，但是这样做只适用于小型的示例和工具。在你自己的代码中，你需要考虑更加复杂的错误处理方式。

5.6.1 缓冲输入文件

如果想要打开一个文件进行字符输入，我们可以使用一个 `FileInputReader` 对象，然后传入一个 `String` 或者 `File` 对象作为文件名。为了提高速度，我们希望对那个文件进行缓冲，那么我们可以将所产生的引用传递给一个 `BufferedReader` 构造器。`BufferedReader` 提供了 `line()` 方法，它会产生一个 `Stream<String>` 对象：

```
// iostreams/BufferedInputFile.java
// {VisuallyInspectOutput}
import java.io.*;
import java.util.stream.*;

public class BufferedInputFile {
    public static String read(String filename) {
        try (BufferedReader in = new BufferedReader(
                new FileReader(filename))) {
            return in.lines()
                    .collect(Collectors.joining("\n"));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        System.out.print(
                read("BufferedInputFile.java"));
    }
}
```

`Collectors.joining()` 在其内部使用了一个 `StringBuilder` 来累加其运行结果。该文件会通过 `try-with-resources` 子句自动关闭。

5.6.2 从内存输入

下面示例中，从 `BufferedInputFile.read()` 读入的 `String` 被用来创建一个 `StringReader` 对象。然后调用其 `read()` 方法，每次读取一个字符，并把它显示在控制台上：

```java
// iostreams/MemoryInput.java
// {VisuallyInspectOutput}
import java.io.*;

public class MemoryInput {
    public static void
    main(String[] args) throws IOException {
        StringReader in = new StringReader(
                BufferedInputFile.read("MemoryInput.java"));
        int c;
        while ((c = in.read()) != -1)
            System.out.print((char) c);
    }
}
```

注意 `read()` 是以 **`int`** 形式返回下一个字节，所以必须类型转换为 **`char`** 才能正确打印。

5.6.3 格式化内存输入

要读取格式化数据，我们可以使用 `DataInputStream`，它是一个面向字节的 I/O 类（不是面向字符的）。这样我们就必须使用 `InputStream` 类而不是 `Reader` 类。我们可以使用 `InputStream` 以字节形式读取任何数据（比如一个文件），但这里使用的是字符串。

```java
// iostreams/FormattedMemoryInput.java
// {VisuallyInspectOutput}
import java.io.*;

public class FormattedMemoryInput {
    public static void main(String[] args) {
        try (
            DataInputStream in = new DataInputStream(
                new ByteArrayInputStream(
                    BufferedInputFile.read(
                        "FormattedMemoryInput.java")
                            .getBytes()))
        ) {
            while (true)
                System.out.write((char) in.readByte());
        } catch (EOFException e) {
            System.out.println("\nEnd of stream");
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

`ByteArrayInputStream` 必须接收一个字节数组，所以这里我们调用了 `String.getBytes()` 方法。所产生的的 `ByteArrayInputStream` 是一个适合传递给 `DataInputStream` 的 `InputStream`

。

如果我们用 readByte() 从 DataInputStream 一次一个字节地读取字符，那么任何字节的值都是合法结果，因此返回值不能用来检测输入是否结束。取而代之的是，我们可以使用 available() 方法得到剩余可用字符的数量。下面例子演示了怎么一次一个字节地读取文件：

```
// iostreams/TestEOF.java
// Testing for end of file
// {VisuallyInspectOutput}
import java.io.*;

public class TestEOF {
    public static void main(String[] args) {
        try (
            DataInputStream in = new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("TestEOF.java")))
        ) {
            while (in.available() != 0)
                System.out.write(in.readByte());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

注意，available() 的工作方式会随着所读取媒介类型的不同而有所差异，它的字面意思就是“在没有阻塞的情况下所能读取的字节数”。对于文件，能够读取的是整个文件；但是对于其它类型的“流”，可能就不是这样，所以要谨慎使用。

我们也可以通过捕获异常来检测输入的末尾。但是，用异常作为控制流是对异常的一种错误使用方式。

5.6.4 基本文件的输出

FileWriter 对象用于向文件写入数据。实际使用时，我们通常会用 BufferedWriter 将其包装起来以增加缓冲的功能（可以试试移除此包装来感受一下它对性能的影响——缓冲往往能显著地增加 I/O 操作的性能）。在本例中，为了提供格式化功能，它又被装饰成了 PrintWriter。按照这种方式创建的数据文件可作为普通文本文件来读取。

```
// iostreams/BasicFileOutput.java
// {VisuallyInspectOutput}
import java.io.*;

public class BasicFileOutput {
    static String file = "BasicFileOutput.dat";

```

```
    public static void main(String[] args) {
        try (
            BufferedReader in = new BufferedReader(
                new StringReader(
                     BufferedInputFile.read(
                         "BasicFileOutput.java")));
                PrintWriter out = new PrintWriter(
                    new BufferedWriter(new FileWriter(file)))
        ) {
            in.lines().forEach(out::println);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        // Show the stored file:
        System.out.println(BufferedInputFile.read(file));
    }
}
```

`try-with-resources` 语句会自动 flush 并关闭文件。

5.6.5 文本文件输出快捷方式

Java 5 在 `PrintWriter` 中添加了一个辅助构造器，有了它，你在创建并写入文件时，就不必每次都手动执行一些装饰的工作。下面的代码使用这种快捷方式重写了 `BasicFileOutput.java`:

```
// iostreams/FileOutputShortcut.java
// {VisuallyInspectOutput}
import java.io.*;

public class FileOutputShortcut {
    static String file = "FileOutputShortcut.dat";

    public static void main(String[] args) {
        try (
            BufferedReader in = new BufferedReader(
                new StringReader(BufferedInputFile.read(
                    "FileOutputShortcut.java")));
            // Here's the shortcut:
            PrintWriter out = new PrintWriter(file)
        ) {
            in.lines().forEach(out::println);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        System.out.println(BufferedInputFile.read(file));
    }
}
```

使用这种方式仍具备了缓冲的功能，只是现在不必自己手动添加缓冲了。但遗憾的是，其它常见的

写入任务都没有快捷方式，因此典型的 I/O 流依旧涉及大量冗余的代码。本书文件一章中介绍的另一种方式，对此类任务进行了极大的简化。

5.6.6 存储和恢复数据

PrintWriter 是用来对可读的数据进行格式化。但如果要输出可供另一个“流”恢复的数据，我们可以用 DataOutputStream 写入数据，然后用 DataInputStream 恢复数据。当然，这些流可能是任何形式，在下面的示例中使用的是一个文件，并且对读写都进行了缓冲。注意 DataOutputStream 和 DataInputStream 是面向字节的，因此要使用 InputStream 和 OutputStream 体系的类。

```java
// iostreams/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
    public static void main(String[] args) {
        try (
            DataOutputStream out = new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("Data.txt")))
        ) {
            out.writeDouble(3.14159);
            out.writeUTF("That was pi");
            out.writeDouble(1.41413);
            out.writeUTF("Square root of 2");
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        try (
            DataInputStream in = new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("Data.txt")))
        ) {
            System.out.println(in.readDouble());
            // Only readUTF() will recover the
            // Java-UTF String properly:
            System.out.println(in.readUTF());
            System.out.println(in.readDouble());
            System.out.println(in.readUTF());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

输出结果:

```
3.14159
That was pi
```

```
1.41413
Square root of 2
```

如果我们使用 DataOutputStream 进行数据写入，那么 Java 就保证了即便读和写数据的平台多么不同，我们仍可以使用 DataInputStream 准确地读取数据。这一点很有价值，众所周知，人们曾把大量精力耗费在数据的平台相关性问题上。但现在，只要两个平台上都有 Java，就不会存在这样的问题[3]。

当我们使用 DastaOutputStream 时，写字符串并且让 DataInputStream 能够恢复它的唯一可靠方式就是使用 UTF-8 编码，在这个示例中是用 writeUTF() 和 readUTF() 来实现的。UTF-8 是一种多字节格式，其编码长度根据实际使用的字符集会有所变化。如果我们使用的只是 ASCII 或者几乎都是 ASCII 字符（只占 7 比特），那么就显得及其浪费空间和带宽，所以 UTF-8 将 ASCII 字符编码成一个字节的形式，而非 ASCII 字符则编码成两到三个字节的形式。另外，字符串的长度保存在 UTF-8 字符串的前两个字节中。但是，writeUTF() 和 readUTF() 使用的是一种适用于 Java 的 UTF-8 变体（JDK 文档中有这些方法的详尽描述），因此如果我们用一个非 Java 程序读取用 writeUTF() 所写的字符串时，必须编写一些特殊的代码才能正确读取。

有了 writeUTF() 和 readUTF()，我们就可以在 DataOutputStream 中把字符串和其它数据类型混合使用。因为字符串完全可以作为 Unicode 格式存储，并且可以很容易地使用 DataInputStream 来恢复它。

writeDouble() 将 **double** 类型的数字存储在流中，并用相应的 readDouble() 恢复它（对于其它的书类型，也有类似的方法用于读写）。但是为了保证所有的读方法都能够正常工作，我们必须知道流中数据项所在的确切位置，因为极有可能将保存的 **double** 数据作为一个简单的字节序列、**char** 或其它类型读入。因此，我们必须：要么为文件中的数据采用固定的格式；要么将额外的信息保存到文件中，通过解析额外信息来确定数据的存放位置。注意，对象序列化和 XML （二者都在附录：对象序列化中介绍）是存储和读取复杂数据结构的更简单的方式。

5.6.7 读写随机访问文件

使用 RandomAccessFile 就像是使用了一个 DataInputStream 和 DataOutputStream 的结合体（因为它实现了相同的接口：DataInput 和 DataOutput）。另外，我们还可以使用 seek() 方法移动文件指针并修改对应位置的值。

在使用 RandomAccessFile 时，你必须清楚文件的结构，否则没法正确使用它。RandomAccessFile 有一套专门的方法来读写基本数据类型的数据和 UTF-8 编码的字符串：

```
// iostreams/UsingRandomAccessFile.java
import java.io.*;
```

[3]请注意，在 64 位处理器上可能不会发生这种情况，从而消除了这个问题。

```java

public class UsingRandomAccessFile {
    static String file = "rtest.dat";

    public static void display() {
        try (
            RandomAccessFile rf =
                new RandomAccessFile(file, "r")
        ) {
            for (int i = 0; i < 7; i++)
                System.out.println(
                    "Value " + i + ": " + rf.readDouble());
            System.out.println(rf.readUTF());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        try (
            RandomAccessFile rf =
                new RandomAccessFile(file, "rw")
        ) {
            for (int i = 0; i < 7; i++)
                rf.writeDouble(i * 1.414);
            rf.writeUTF("The end of the file");
            rf.close();
            display();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        try (
            RandomAccessFile rf =
                new RandomAccessFile(file, "rw")
        ) {
            rf.seek(5 * 8);
            rf.writeDouble(47.0001);
            rf.close();
            display();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

输出结果:

```
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
```

```
Value 4: 5.656
Value 5: 7.069999999999999
Value 6: 8.484
The end of the file
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 47.0001
Value 6: 8.484
The end of the file
```

`display()` 方法打开了一个文件，并以 **`double`** 值的形式显示了其中的七个元素。在 `main()` 中，首先创建了文件，然后打开并修改了它。因为 **`double`** 总是 8 字节长，所以如果要用 `seek()` 定位到第 5 个（从 0 开始计数）**`double`** 值，则要传入的地址值应该为 `5*8`。

正如前面所诉，虽然 `RandomAccess` 实现了 `DataInput` 和 `DataOutput` 接口，但实际上它和 I/O 继承体系中的其它部分是分离的。它不支持装饰，故而不能将其与 `InputStream` 及 `OutputStream` 子类中的任何一个组合起来，所以我们也没法给它添加缓冲的功能。

该类的构造器还有第二个必选参数：我们可以指定让 `RandomAccessFile` 以“只读”（r）方式或“读写”（rw）方式打开文件。

除此之外，还可以使用 `nio` 中的“内存映射文件”代替 `RandomAccessFile`，这在附录：新 I/O中有介绍。

5.7 本章小结

Java 的 I/O 流类库的确能够满足我们的基本需求：我们可以通过控制台、文件、内存块，甚至因特网进行读写。通过继承，我们可以创建新类型的输入和输出对象。并且我们甚至可以通过重新定义“流”所接受对象类型的 `toString()` 方法，进行简单的扩展。当我们向一个期望收到字符串的方法传送一个非字符串对象时，会自动调用对象的 `toString()` 方法（这是 Java 中有限的“自动类型转换”功能之一）。

在 I/O 流类库的文档和设计中，仍留有一些没有解决的问题。例如，我们打开一个文件用于输出，如果在我们试图覆盖这个文件时能抛出一个异常，这样会比较好（有的编程系统只有当该文件不存在时，才允许你将其作为输出文件打开）。在 Java 中，我们应该使用一个 `File` 对象来判断文件是否存在，因为如果我们用 `FileOutputStream` 或者 `FileWriter` 打开，那么这个文件肯定会被覆盖。

I/O 流类库让我们喜忧参半。它确实挺有用的，而且还具有可移植性。但是如果我们没有理解“装饰器”模式，那么这种设计就会显得不是很直观。所以，它的学习成本相对较高。而且它并不完善，比如说在过去，我不得不编写相当数量的代码去实现一个读取文本文件的工具——所幸的是，Java 7 中的 nio 消除了此类需求。

一旦你理解了装饰器模式，并且开始在某些需要这种灵活性的场景中使用该类库，那么你就开始能从这种设计中受益了。到那时候，为此额外多写几行代码的开销应该不至于让人觉得太麻烦。但还是请务必检查一下，确保使用文件一章中的库和技术没法解决问题后，再考虑使用本章的 I/O 流库。

附录 6

标准 IO

> *标准 I/O* 这个术语参考 Unix 中的概念，指程序所使用的单一信息流（这种思想在大多数操作系统中，也有相似形式的实现）。

程序的所有输入都可以来自于*标准输入*，其所有输出都可以流向*标准输出*，并且其所有错误信息均可以发送到*标准错误*。*标准 I/O* 的意义在于程序之间可以很容易地连接起来，一个程序的标准输出可以作为另一个程序的标准输入。这是一个非常强大的工具。

6.1 从标准输入中读取

遵循标准 I/O 模型，Java 提供了标准输入流 `System.in`、标准输出流 `System.out` 和标准错误流 `System.err`。在本书中，你已经了解到如何使用 `System.out`将数据写到标准输出。`System.out` 已经预先包装[1]成了 `PrintStream` 对象。标准错误流 `System.err` 也预先包装为 `PrintStream` 对象，但是标准输入流 `System.in` 是原生的没有经过包装的 `InputStream`。这意味着尽管可以直接使用标准输出流 `System.in` 和标准错误流 `System.err`，但是在读取 `System.in` 之前必须先对其进行包装。

我们通常一次一行地读取输入。为了实现这个功能，将 `System.in` 包装成 `BufferedReader` 来使用，这要求我们用 `InputStreamReader` 把 `System.in` 转换[2]成 `Reader` 。下面这个例子将键入的每一行显示出来：

```
// standardio/Echo.java
// How to read from standard input
import java.io.*;
import onjava.TimedAbort;

public class Echo {
```

[1]在某些平台上，特别是 Windows ，默认值可能非常难以查明。你可以使用 -Xss 标志调整堆栈大小。

[2]引自 Brian Goetz, Java Concurrency in Practice 一书的作者，该书由 Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea 联合著作 (Addison-Wesley 出版社, 2006)。

```
    public static void main(String[] args) {
        TimedAbort abort = new TimedAbort(2);
        new BufferedReader(
                new InputStreamReader(System.in))
                .lines()
                .peek(ln -> abort.restart())
                .forEach(System.out::println);
        // Ctrl-Z or two seconds inactivity
        // terminates the program
    }
}
```

BufferedReader 提供了 lines() 方法，返回类型是 Stream<String> 。这显示出流模型的的灵活性：仅使用标准输入就能很好地工作。peek() 方法重启 TimeAbort，只要保证至少每隔两秒有输入就能够使程序保持开启状态。

6.2 将 System.out 转换成 PrintWriter

System.out 是一个 PrintStream，而 PrintStream 是一个OutputStream。PrintWriter 有一个把 OutputStream 作为参数的构造器。因此，如果你需要的话，可以使用这个构造器把 System.out 转换成 PrintWriter 。

```
// standardio/ChangeSystemOut.java
// Turn System.out into a PrintWriter

import java.io.*;

public class ChangeSystemOut {
    public static void main(String[] args) {
        PrintWriter out =
                new PrintWriter(System.out, true);
        out.println("Hello, world");
    }
}
```

输出结果：

```
Hello, world
```

要使用 PrintWriter 带有两个参数的构造器，并设置第二个参数为 **true**，从而使能自动刷新到输出缓冲区的功能；否则，可能无法看到打印输出。

6.3 重定向标准 I/O

Java 的 System 类提供了简单的 **static** 方法调用，从而能够重定向标准输入流、标准输出流和标准错误流：- setIn（InputStream）- setOut（PrintStream）- setErr(PrintStream)

如果我们突然需要在显示器上创建大量的输出，而这些输出滚动的速度太快以至于无法阅读时，重定向输出就显得格外有用，可把输出内容重定向到文件中供后续查看。对于我们想重复测试特定的用户输入序列的命令行程序来说，重定向输入就很有价值。下例简单演示了这些方法的使用：

```
// standardio/Redirecting.java
// Demonstrates standard I/O redirection
import java.io.*;

public class Redirecting {
    public static void main(String[] args) {
        PrintStream console = System.out;
        try (
                BufferedInputStream in = new BufferedInputStream(
                        new FileInputStream("Redirecting.java"));
                PrintStream out = new PrintStream(
                        new BufferedOutputStream(
                                new FileOutputStream("Redirecting.txt")
                                    ))
        ) {
            System.setIn(in);
            System.setOut(out);
            System.setErr(out);
            new BufferedReader(
                    new InputStreamReader(System.in))
                    .lines()
                    .forEach(System.out::println);
        } catch (IOException e) {
            throw new RuntimeException(e);
        } finally {
            System.setOut(console);
        }
    }
}
```

该程序将文件中内容载入到标准输入，并把标准输出和标准错误重定向到另一个文件。它在程序的开始保存了最初对 System.out 对象的引用，并且在程序结束时将系统输出恢复到了该对象上。

I/O 重定向操作的是字节流而不是字符流，因此使用 InputStream 和 OutputStream，而不是 Reader 和 Writer。

6.4 执行控制

你经常需要在 Java 内部直接执行操作系统的程序，并控制这些程序的输入输出，Java 类库提供了执行这些操作的类。

一项常见的任务是运行程序并将输出结果发送到控制台。本节包含了一个可以简化此任务的实用工具。

在使用这个工具时可能会产生两种类型的错误：导致异常的普通错误——对于这些错误我们只需要重新抛出一个 RuntimeException 即可，以及进程自身的执行过程中导致的错误——我们需要用单独的异常来报告这些错误：

```
// onjava/OSExecuteException.java
package onjava;

public class OSExecuteException extends RuntimeException {
    public OSExecuteException(String why) {
        super(why);
    }
}
```

为了运行程序，我们需要传递给 OSExecute.command() 一个 String command，我们可以在控制台键入同样的指令运行程序。该命令传递给 java.lang.ProcessBuilder 的构造器（需要将其作为 String 对象的序列），然后启动生成的 ProcessBuilder 对象。

```
// onjava/OSExecute.java
// Run an operating system command
// and send the output to the console
package onjava;
import java.io.*;

public class OSExecute {
    public static void command(String command) {
        boolean err = false;
        try {
            Process process = new ProcessBuilder(
                    command.split(" ")).start();
            try (
                    BufferedReader results = new BufferedReader(
                            new InputStreamReader(
                                    process.getInputStream()));
                    BufferedReader errors = new BufferedReader(
                            new InputStreamReader(
                                    process.getErrorStream()))
            ) {
                results.lines()
                        .forEach(System.out::println);
                err = errors.lines()
```

```
                        .peek(System.err::println)
                        .count() > 0;
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        if (err)
            throw new OSExecuteException(
                    "Errors executing " + command);
    }
}
```

为了捕获在程序执行时产生的标准输出流，我们可以调用 getInputStream()。这是因为 InputStream 是我们可以从中读取信息的流。

这里这些行只是被打印了出来，但是你也可以从 command() 捕获和返回它们。

该程序的错误被发送到了标准错误流，可以调用 getErrorStream() 捕获。如果存在任何错误，它们都会被打印并且抛出 OSExcuteException ，以便调用程序处理这个问题。

下面是展示如何使用 OSExecute 的示例：

```
// standardio/OSExecuteDemo.java
// Demonstrates standard I/O redirection
// {javap -cp build/classes/main OSExecuteDemo}
import onjava.*;

public class OSExecuteDemo {}
```

这里使用 javap 反编译器（随 JDK 发布）来反编译程序，编译结果：

```
Compiled from "OSExecuteDemo.java"
public class OSExecuteDemo {
  public OSExecuteDemo();
}
```

梯子的梯阶从来不是用来搁脚的，它只是让人们的脚放上一段时间，以便让别一只脚能够再往上登。

附录 7

新 IO

> Java 新 I/O 库是在 1.4 版本引入到 `Java .nio.*` **package** 中的，旨在更快速。

实际上，新 I/O 使用 **NIO**（同步非阻塞）的方式重写了老的 I/O 了，因此它获得了 **NIO** 的种种优点。即使我们不显式地使用 **NIO** 方式来编写代码，也能带来性能和速度的提高。这种提升不仅仅体现在文件读写（File I/O），同时也体现在网络读写（Network I/O）中。例如，网络编程。

速度的提升来自于使用了更接近操作系统 I/O 执行方式的结构：**Channel**（通道）和 **Buffer**（缓冲区）。我们可以想象一个煤矿：通道就是连接矿层（数据）的矿井，缓冲区是运送煤矿的小车。通过小车装煤，再从车里取矿。换句话说，我们不能直接和 **Channel** 交互; 我们需要与 **Buffer** 交互并将 **Buffer** 中的数据发送到 **Channel** 中；**Channel** 需要从 **Buffer** 中提取或放入数据。

本篇我们将深入探讨 `nio` 包。虽然像 I/O 流这样的高级库使用了 **NIO**，但多数时候，我们考虑这个层次的问题。使用 Java 7 和 8 版本，理想情况下我们甚至不必费心去处理 I/O 流。当然，一些特殊情况除外。在文件（**File**）一章中基本涵盖了我们日常使用的相关内容。只有在遇到性能瓶颈（例如内存映射文件）或创建自己的 I/O 库时，我们才需要去理解 **NIO**。

7.1 ByteBuffer

有且仅有 **ByteBuffer**（字节缓冲区，保存原始字节的缓冲区）这一类型可直接与通道交互。查看 `java.nio.`**ByteBuffer** 的 JDK 文档，你会发现它是相当基础的：通过初始化某个大小的存储空间，再使用一些方法以原始字节形式或原始数据类型来放置和获取数据。但是我们无法直接存放对象，即使是最基本的 **String** 类型数据。这是一个相当底层的操作，也正因如此，使得它与大多数操作系统的映射更加高效。

旧式 I/O 中的三个类分别被更新成 **FileChannel**（文件通道），分别是：**FileInputStream**、**FileOutputStream**，以及用于读写的 **RandomAccessFile** 类。

注意，这些都是符合底层 **NIO** 特性的字节操作流。另外，还有 **Reader** 和 **Writer** 字符模式的类是不产生通道的。但 `java.nio.channels.`**Channels** 类具有从通道中生成 **Reader** 和 **Writer** 的实

你在学习上这种尝试精神很可贵。

用方法。

下面来练习上述三种类型的流生成可读、可写、可读/写的通道：

```
// (c)2017 MindView LLC: see Copyright.txt
// 我们不保证这段代码用于其他用途时是否有效
// 访问 http://OnJava8.com 了解更多信息
// 从流中获取通道
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
  private static String name = "data.txt";
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    // 写入一个文件:
    try(
      FileChannel fc = new FileOutputStream(name)
        .getChannel()
    ) {
      fc.write(ByteBuffer
        .wrap("Some text ".getBytes()));
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // 在文件尾添加:
    try(
      FileChannel fc = new RandomAccessFile(
        name, "rw").getChannel()
    ) {
      fc.position(fc.size()); // 移动到结尾
      fc.write(ByteBuffer
        .wrap("Some more".getBytes()));
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // 读取文件e:
    try(
      FileChannel fc = new FileInputStream(name)
        .getChannel()
    ) {
      ByteBuffer buff = ByteBuffer.allocate(BSIZE);
      fc.read(buff);
      buff.flip();
      while(buff.hasRemaining())
        System.out.write(buff.get());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    System.out.flush();
```

```
  }
}
```

输出结果:

```
Some text Some more
```

我们这里所讲的任何流类，都可以通过调用 `getChannel( )` 方法生成一个 **FileChannel**（文件通道）。**FileChannel** 的操作相当基础：操作 **ByteBuffer** 来用于读写，并独占式访问和锁定文件区域 (稍后将对此进行描述)。

将字节放入 **ByteBuffer** 的一种方法是直接调用 `put()` 方法将一个或多个字节放入 **ByteBuffer**；当然也可以是其它基本类型的数据。此外，参考上例，我们还可以调用 `wrap()` 方法包装现有字节数组到 **ByteBuffer**。执行此操作时，不会复制底层数组，而是将其用作生成的 **ByteBuffer** 存储。这样产生的 **ByteBuffer** 是数组“支持”的。

data.txt 文件被 **RandomAccessFile** 重新打开。**注意**，你可以在文件中移动 **FileChanne**。在这里，它被移动到末尾，以便添加额外的写操作。

对于只读访问，必须使用静态 `allocate()` 方法显式地分配 **ByteBuffer**。**NIO** 的目标是快速移动大量数据，因此 **ByteBuffer** 的大小应该很重要 —— 实际上，这里设置的 1K 都可能偏小了 (我们在工作中应该反复测试以找到最佳大小)。

通过使用 `allocateDirect()` 而不是 `allocate()` 来生成与操作系统具备更高耦合度的“直接”缓冲区，也有可能获得更高的速度。然而，这种分配的开销更大，而且实际效果因操作系统的不同而有所不同，因此，在工作中你必须再次测试程序，以检验直接缓冲区是否能为你带来速度上的优势。

一旦调用 **FileChannel** 类的 `read()` 方法将字节数据存储到 **ByteBuffer** 中，你还必须调用缓冲区上的 `flip()` 方法来准备好提取字节（这听起来有点粗糙，实际上这已是非常低层的操作，且为了达到最高速度）。如果要进一步调用 `read()` 来使用 **ByteBuffer** ，还需要每次 `clear()` 来准备缓冲区。下面是个简单的代码示例:

```java
// newio/ChannelCopy.java

// 使用 channels and buffers 移动文件
// {java ChannelCopy ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    if(args.length != 2) {
      System.out.println(
        "arguments: sourcefile destfile");
      System.exit(1);
    }
```

```
    try(
      FileChannel in = new FileInputStream(
        args[0]).getChannel();
      FileChannel out = new FileOutputStream(
        args[1]).getChannel()
    ) {
      ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
      while(in.read(buffer) != -1) {
        buffer.flip(); // 准备写入
        out.write(buffer);
        buffer.clear();  // 准备读取
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

FileChannel 用于读取; **FileChannel** 用于写入。当 **ByteBuffer** 分配好存储, 调用 **FileChannel** 的 `read()` 方法返回 **-1**（毫无疑问，这是来源于 Unix 和 C 语言）时，说明输入流读取完了。在每次 `read()` 将数据放入缓冲区之后，`flip()` 都会准备好缓冲区，以便 `write()` 提取它的信息。在 `write()` 之后, 数据仍然在缓冲区中, 我们需要 `clear()` 来重置所有内部指针, 以便在下一次 `read()` 中接受数据。

但是，上例并不是处理这种操作的理想方法。方法 `transferTo()` 和 `transferFrom()` 允许你直接连接此通道到彼通道:

```
// newio/TransferTo.java

// 使用 transferTo() 在通道间传输
// {java TransferTo TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
  public static void main(String[] args) {
    if(args.length != 2) {
      System.out.println(
        "arguments: sourcefile destfile");
      System.exit(1);
    }
    try(
      FileChannel in = new FileInputStream(
        args[0]).getChannel();
      FileChannel out = new FileOutputStream(
        args[1]).getChannel()
    ) {
      in.transferTo(0, in.size(), out);
      // Or:
```

```
      // out.transferFrom(in, 0, in.size());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

可能不会经常用到，但知道这一点很好。

7.2 数据转换

为了将 **GetChannel.java** 文件中的信息打印出来。在 Java 中，我们每次提取一个字节的数据并将其转换为字符。看起来很简单 —— 如果你有看过 `ava.nio.`**CharBuffer** 类，你会发现一个 `toString()` 方法。该方法的作用是"返回一个包含此缓冲区字符的字符串"。

既然 **ByteBuffer** 可以通过 **CharBuffer** 类的 `asCharBuffer()` 方法查看，那我们就来尝试一样。从下面输出语句的第一行可以看出，这并不正确：

```
// newio/BufferToText.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
// text 和 ByteBuffers 互转
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BufferToText {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {

    try(
      FileChannel fc = new FileOutputStream(
        "data2.txt").getChannel()
    ) {
      fc.write(ByteBuffer.wrap("Some text".getBytes()));
    } catch(IOException e) {
      throw new RuntimeException(e);
    }

    ByteBuffer buff = ByteBuffer.allocate(BSIZE);

    try(
      FileChannel fc = new FileInputStream(
        "data2.txt").getChannel()
    ) {
      fc.read(buff);
```

```
    } catch(IOException e) {
      throw new RuntimeException(e);
    }

    buff.flip();
    // 无法运行
    System.out.println(buff.asCharBuffer());
    // 使用默认系统默认编码解码
    buff.rewind();
    String encoding =
      System.getProperty("file.encoding");
    System.out.println("Decoded using " +
      encoding + ": "
      + Charset.forName(encoding).decode(buff));

    // 编码和打印
    try(
      FileChannel fc = new FileOutputStream(
        "data2.txt").getChannel()
    ) {
      fc.write(ByteBuffer.wrap(
        "Some text".getBytes("UTF-16BE")));
    } catch(IOException e) {
      throw new RuntimeException(e);
    }

    // 尝试再次读取:
    buff.clear();
    try(
      FileChannel fc = new FileInputStream(
        "data2.txt").getChannel()
    ) {
      fc.read(buff);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }

    buff.flip();
    System.out.println(buff.asCharBuffer());
    // 通过 CharBuffer 写入:
    buff = ByteBuffer.allocate(24);
    buff.asCharBuffer().put("Some text");

    try(
      FileChannel fc = new FileOutputStream(
        "data2.txt").getChannel()
    ) {
      fc.write(buff);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
```

```
    // 读取和显示:
    buff.clear();

    try(
      FileChannel fc = new FileInputStream(
        "data2.txt").getChannel()
    ) {
      fc.read(buff);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }

    buff.flip();
    System.out.println(buff.asCharBuffer());
  }
}
```

输出结果:

```
????
Decoded using windows-1252: Some text
Some text
Some textNULNULNUL
```

缓冲区包含普通字节，为了将这些字节转换为字符，我们必须在输入时对它们进行编码 (这样它们输出时就有意义了)，或者在输出时对它们进行解码。我们可以使用 `java.nio.charset.`**Charset** 字符集工具类来完成。代码示例:

```
// newio/AvailableCharSets.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
// 展示 Charsets 和 aliases
import java.nio.charset.*;
import java.util.*;

public class AvailableCharSets {

  public static void main(String[] args) {
      SortedMap<String,Charset> charSets =
      Charset.availableCharsets();

    for(String csName : charSets.keySet()) {
      System.out.print(csName);
      Iterator aliases = charSets.get(csName)
        .aliases().iterator();
      if(aliases.hasNext())
        System.out.print(": ");

      while(aliases.hasNext()) {
```

```
        System.out.print(aliases.next());
        if(aliases.hasNext())
          System.out.print(", ");
      }
      System.out.println();
    }
  }
}
```

输出结果:

```
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, Big5_HKSCS, big5hkscs
CESU-8: CESU8, csCESU-8
EUC-JP: csEUCPkdFmtjapanese, x-euc-jp, eucjis,
Extended_UNIX_Code_Packed_Format_for_Japanese, euc_jp,
eucjp, x-eucjp
EUC-KR: ksc5601-1987, csEUCKR, ksc5601_1987, ksc5601,
5601,
euc_kr, ksc_5601, ks_c_5601-1987, euckr
GB18030: gb18030-2000
GB2312: gb2312, euc-cn, x-EUC-CN, euccn, EUC_CN,
gb2312-80,
gb2312-1980
                    ...
```

回到 **BufferToText.java** 中，如果你 `rewind()` 缓冲区（回到数据的开头），使用该平台的默认字符集 `decode()` 数据，那么生成的 **CharBuffer** 数据将在控制台上正常显示。可以通过 `System.getProperty("file.encoding")` 方法来查看平台默认字符集名称。传递该名称参数到 `Charset.forName()` 方法可以生成对应的 `Charset` 对象用于解码字符串。

另一种方法是使用字符集 `encode()` 方法，该字符集在读取文件时生成可打印的内容，如你在 **BufferToText.java** 的第三部分中所看到的。上例中，**UTF-16BE** 被用于将文本写入文件，当文本被读取时，你所要做的就是将其转换为 **CharBuffer**，并生成预期的文本。

最后，如果将 **CharBuffer** 写入 **ByteBuffer**，你会看到发生了什么 (更多详情，稍后了解)。**注意**，为 **ByteBuffer** 分配了 24 个字节，按照每个字符占用 2 个自字节，12 个字符的空间已经足够了。由于“some text”只有 9 个字符，受其 `toString()` 方法影响，剩下的 0 字节部分也出现在了 **CharBuffer** 的展示中，如输出所示。

7.3 基本类型获取

虽然 **ByteBuffer** 只包含字节，但它包含了一些方法，用于从其所包含的字节中生成各种不同的基本类型数据。代码示例:

```
// newio/GetData.java
```

```java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
// 从 ByteBuffer 中获取不同的数据展示
import java.nio.*;

public class GetData {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    // 自动分配 0 到 ByteBuffer:
    int i = 0;
    while(i++ < bb.limit())
      if(bb.get() != 0)
        System.out.println("nonzero");
    System.out.println("i = " + i);
    bb.rewind();
    // 保存和读取 char 数组:
    bb.asCharBuffer().put("Howdy!");
    char c;
    while((c = bb.getChar()) != 0)
      System.out.print(c + " ");
    System.out.println();
    bb.rewind();
    // 保存和读取 short:
    bb.asShortBuffer().put((short)471142);
    System.out.println(bb.getShort());
    bb.rewind();
    // 保存和读取 int:
    bb.asIntBuffer().put(99471142);
    System.out.println(bb.getInt());
    bb.rewind();
    // 保存和读取 long:
    bb.asLongBuffer().put(99471142);
    System.out.println(bb.getLong());
    bb.rewind();
    // 保存和读取 float:
    bb.asFloatBuffer().put(99471142);
    System.out.println(bb.getFloat());
    bb.rewind();
    // 保存和读取 double:
    bb.asDoubleBuffer().put(99471142);
    System.out.println(bb.getDouble());
    bb.rewind();
  }
}
```

输出结果:

```
i = 1025
H o w d y !
```

```
12390
99471142
99471142
9.9471144E7
9.9471142E7
```

在分配 **ByteBuffer** 之后，我们检查并确认它的 1,024 元素被初始化为 0。（截至到达 `limit()` 结果的位置）。

将基本类型数据插入 **ByteBuffer** 的最简单方法就是使用 `asCharBuffer()`、`asShortBuffer()` 等方法获取该缓冲区适当的“视图”（View），然后调用该“视图”的 `put()` 方法。

这是针对每种基本数据类型执行的。其中唯一有点奇怪的是 **ShortBuffer** 的 `put()`，它需要类型强制转换。其他视图缓冲区不需要在其 `put()` 方法中进行转换。

7.4 视图缓冲区

“视图缓冲区”（view buffer）是通过特定的基本类型的窗口来查看底层 **ByteBuffer**。**ByteBuffer** 仍然是“支持”视图的实际存储，因此对视图所做的任何更改都反映在对 **ByteBuffer** 中的数据的修改中。

如前面的示例所示，这方便地将基本类型插入 **ByteBuffer**。视图缓冲区还可以从 **ByteBuffer** 读取基本类型数据，每次单个（**ByteBuffer** 规定），或者批量读取到数组。下面是一个通过 **IntBuffer** 在 **ByteBuffer** 中操作 **int** 的例子：

```java
// newio/IntBufferDemo.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
// 利用 IntBuffer 保存 int 数据到 ByteBuffer
import java.nio.*;

public class IntBufferDemo {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    IntBuffer ib = bb.asIntBuffer();
    // 保存 int 数组:
    ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
    //绝对位置读写:
    System.out.println(ib.get(3));
    ib.put(3, 1811);
    // 在重置缓冲区前设置新的限制

    ib.flip();
    while(ib.hasRemaining()) {
      int i = ib.get();
```

```
      System.out.println(i);
    }
  }
}
```

输出结果:

```
99
11
42
47
1811
143
811
1016
```

put() 方法重载，首先用于存储 **int** 数组。下面的 get() 和 put() 方法调用直接访问底层 **ByteBuffer** 中的 **int** 位置。**注意**，通过直接操作 **ByteBuffer** ，这些绝对位置访问也可以用于基本类型。

一旦底层 **ByteBuffer** 通过视图缓冲区填充了 **int** 或其他基本类型，那么就可以直接将该 **ByteBuffer** 写入通道。你可以轻松地从通道读取数据，并使用视图缓冲区将所有内容转换为特定的基本类型。下面是一个例子，通过在同一个 **ByteBuffer** 上生成不同的视图缓冲区，将相同的字节序列解释为 **short**、**int**、**float**、**long** 和 **double**。代码示例:

```
// newio/ViewBuffers.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
import java.nio.*;

public class ViewBuffers {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(
      new byte[]{ 0, 0, 0, 0, 0, 0, 0, 'a' });
    bb.rewind();
    System.out.print("Byte Buffer ");
    while(bb.hasRemaining())
      System.out.print(
        bb.position()+ " -> " + bb.get() + ", ");
    System.out.println();
    CharBuffer cb =
      ((ByteBuffer)bb.rewind()).asCharBuffer();
    System.out.print("Char Buffer ");
    while(cb.hasRemaining())
      System.out.print(
        cb.position() + " -> " + cb.get() + ", ");
    System.out.println();
    FloatBuffer fb =
```

```
      ((ByteBuffer)bb.rewind()).asFloatBuffer();
    System.out.print("Float Buffer ");
    while(fb.hasRemaining())
      System.out.print(
        fb.position()+ " -> " + fb.get() + ", ");
    System.out.println();
    IntBuffer ib =
      ((ByteBuffer)bb.rewind()).asIntBuffer();
    System.out.print("Int Buffer ");
    while(ib.hasRemaining())
      System.out.print(
        ib.position()+ " -> " + ib.get() + ", ");
    System.out.println();
    LongBuffer lb =
      ((ByteBuffer)bb.rewind()).asLongBuffer();
    System.out.print("Long Buffer ");
    while(lb.hasRemaining())
      System.out.print(
        lb.position()+ " -> " + lb.get() + ", ");
    System.out.println();
    ShortBuffer sb =
      ((ByteBuffer)bb.rewind()).asShortBuffer();
    System.out.print("Short Buffer ");
    while(sb.hasRemaining())
      System.out.print(
        sb.position()+ " -> " + sb.get() + ", ");
    System.out.println();
    DoubleBuffer db =
      ((ByteBuffer)bb.rewind()).asDoubleBuffer();
    System.out.print("Double Buffer ");
    while(db.hasRemaining())
      System.out.print(
        db.position()+ " -> " + db.get() + ", ");
  }
}
```

输出结果:

```
Byte Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5
-> 0, 6 -> 0, 7 -> 97,
Char Buffer 0 -> NUL, 1 -> NUL, 2 -> NUL, 3 -> a,
Float Buffer 0 -> 0.0, 1 -> 1.36E-43,
Int Buffer 0 -> 0, 1 -> 97,
Long Buffer 0 -> 97,
Short Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Double Buffer 0 -> 4.8E-322,
```

ByteBuffer 通过“包装”一个 8 字节数组生成，然后通过所有不同基本类型的视图缓冲区显示该数组。下图显示了从不同类型的缓冲区读取数据时，数据显示的差异:

Figure 7.1: 1554546258113

7.4.1 字节存储次序

不同的机器可以使用不同的字节存储顺序（Endians）来存储数据。“高位优先”（Big Endian）：将最重要的字节放在最低内存地址中，而“低位优先”（Little Endian）：将最重要的字节放在最高内存地址中。

当存储大于单字节的数据时，如 **int**、**float** 等，我们可能需要考虑字节排序问题。**ByteBuffer** 以“高位优先”形式存储数据；通过网络发送的数据总是使用“高位优先”形式。我们可以使用 **ByteOrder** 的 `order()` 方法和参数 **ByteOrder.BIG_ENDIAN** 或 **ByteOrder.LITTLE_ENDIAN** 来改变它的字节存储次序。

下例是一个包含两个字节的 **ByteBuffer**：

Figure 7.2: 1554546378822

将数据作为 **short** 型来读取（`ByteBuffer.asshortbuffer()`），生成数字 97 （00000000 01100001）。更改为“低位优先”后将生成数字 24832 （01100001 00000000）。

这显示了字节顺序的变化取决于字节存储次序设置:

```
// newio/Endians.java
// (c)2017 MindView LLC: see Copyright.txt
```

```java
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
// 不同字节存储次序的存储
import java.nio.*;
import java.util.*;

public class Endians {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.BIG_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.LITTLE_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays.toString(bb.array()));
  }
}
```

输出结果:

```
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
```

ByteBuffer 分配空间将 **charArray** 中的所有字节作为外部缓冲区保存, 因此可以调用 `array()` 方法来显示底层字节。`array()` 方法是"可选的", 你只能在数组支持的缓冲区上调用它, 否则将抛出 **UnsupportedOperationException** 异常。

charArray 通过 **CharBuffer** 视图插入到 **ByteBuffer** 中。当显示底层字节时, 默认排序与后续"高位"相同, 而"地位"交换字节

7.5 缓冲区数据操作

下图说明了 **nio** 类之间的关系, 展示了如何移动和转换数据。例如, 要将字节数组写入文件, 使用 **ByteBuffer.**`wrap()` 方法包装字节数组, 使用 `getChannel()` 在 **FileOutputStream** 上打开通道, 然后从 **ByteBuffer** 将数据写入 **FileChannel**。

Figure 7.3: 1554546452861

ByteBuffer 是将数据移入和移出通道的唯一方法, 我们只能创建一个独立的基本类型缓冲区, 或者使用 `as` 方法从 **ByteBuffer** 获得一个新缓冲区。也就是说, 不能将基本类型缓冲区转换为 **ByteBuffer**。但我们能够通过视图缓冲区将基本类型数据移动到 **ByteBuffer** 中或移出 **ByteBuffer**。

7.5.1 缓冲区细节

缓冲区由数据和四个索引组成，以有效地访问和操作该数据：mark、position、limit 和 capacity（标记、位置、限制和容量）。伴随着的还有一组方法可以设置和重置这些索引，并可查询它们的值。

capacity()	返回缓冲区的 capacity
clear()	清除缓冲区，将 position 设置为零并设 limit 为 capacity; 可调用此方法来覆盖现有缓冲区
flip()	将 limit 设置为 position，并将 position 设置为 0; 此方法用于准备缓冲区，以便在数据写入缓冲
limit()	返回 limit 的值
limit(int limit)	重设 limit
mark()	设置 mark 为当前的 position
position()	返回 position
position(int pos)	设置 position
remaining()	返回 limit 到 position
hasRemaining()	如果在 position 与 limit 中间有元素，返回 `true`

从缓冲区插入和提取数据的方法通过更新索引来反映所做的更改。下例使用一种非常简单的算法（交换相邻字符）来对 **CharBuffer** 中的字符进行加扰和解扰。代码示例：

```
// newio/UsingBuffers.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
import java.nio.*;

public class UsingBuffers {
  private static
  void symmetricScramble(CharBuffer buffer) {
    while(buffer.hasRemaining()) {
      buffer.mark();
      char c1 = buffer.get();
      char c2 = buffer.get();
      buffer.reset();
      buffer.put(c2).put(c1);
    }
  }

  public static void main(String[] args) {
    char[] data = "UsingBuffers".toCharArray();
    ByteBuffer bb =
```

```
      ByteBuffer.allocate(data.length * 2);
    CharBuffer cb = bb.asCharBuffer();
    cb.put(data);
    System.out.println(cb.rewind());
    symmetricScramble(cb);
    System.out.println(cb.rewind());
    symmetricScramble(cb);
    System.out.println(cb.rewind());
  }
}
```

输出结果:

```
UsingBuffers
sUniBgfuefsr
UsingBuffers
```

虽然可以通过使用 **char** 数组调用 `wrap()` 直接生成 **CharBuffer**，但是底层的 **ByteBuffer** 将被分配，而 **CharBuffer** 将作为 **ByteBuffer** 上的视图生成。这强调了目标始终是操作 **ByteBuffer**，因为它与通道交互。

下面是程序在 `symmetricgrab()` 方法入口时缓冲区的样子:

Figure 7.4: 1554546627710

position 指向缓冲区中的第一个元素，capacity 和 limit 紧接在最后一个元素之后。在`symmetricgrab()` 中，**while** 循环迭代到 position 等于 limit。当在缓冲区上调用相对位置的 `get()` 或 `put()` 函数时，缓冲区的位置会发生变化。你可以调用绝对位置的 `get()` 和 `put()` 方法，它们包含索引参数：`get()` 或 `put()` 发生的位置。这些方法不修改缓冲区 position 的值。

当控件进入 **while** 循环时，使用 `mark()` 设置 mark 的值。缓冲区的状态为:

Figure 7.5: 1554546666685

两个相对 `get()` 调用将前两个字符的值保存在变量 `c1` 和 `c2` 中。在这两个调用之后，缓冲区看起来是这样的：

Figure 7.6: 1554546693664

为了执行交换，我们在位置 0 处编写 `c2`，在位置 1 处编写 `c1`。我们可以使用绝对 `put()` 方法来实现这一点，或者用 `reset()` 方法，将 position 的值设置为 mark：

Figure 7.7: 1554546847181

两个 `put()` 方法分别编写 `c2` 和 `c1`：

Figure 7.8: 1554546861836

在下一次循环中，将 mark 设置为 position 的当前值：

Figure 7.9: 1554546881189

该过程将继续，直到遍历整个缓冲区为止。在 **while** 循环的末尾，position 位于缓冲区的末尾。如果显示缓冲区，则只显示位置和限制之间的字符。因此，要显示缓冲区的全部内容，必须使用 `rewind()` 将 position 设置为缓冲区的开始位置。这是 `rewind()` 调用后缓冲区的状态（mark 的值变成未定义）：

Figure 7.10: 1554546890132

再次调用 `symmetricgrab()` 方法时，**CharBuffer** 将经历相同的过程并恢复到原始状态。

7.6 内存映射文件

内存映射文件能让你创建和修改那些因为太大而无法放入内存的文件。有了内存映射文件，你就可以认为文件已经全部读进了内存，然后把它当成一个非常大的数组来访问。这种解决办法能大大简化修改文件的代码：

```java
// newio/LargeMappedFiles.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
// 使用内存映射来创建一个大文件
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LargeMappedFiles {
  static int length = 0x8000000; // 128 MB
  public static void
  main(String[] args) throws Exception {
    try(
      RandomAccessFile tdat =
        new RandomAccessFile("test.dat", "rw")
    ) {
      MappedByteBuffer out = tdat.getChannel().map(
        FileChannel.MapMode.READ_WRITE, 0, length);
      for(int i = 0; i < length; i++)
        out.put((byte)'x');
      System.out.println("Finished writing");
      for(int i = length/2; i < length/2 + 6; i++)
        System.out.print((char)out.get(i));
    }
  }
}
```

输出结果：

```
Finished writing
xxxxxx
```

为了读写，我们从 **RandomAccessFile** 开始，获取该文件的通道，然后调用 `map()` 来生成 **MappedByteBuffer**，这是一种特殊的直接缓冲区。你必须指定要在文件中映射的区域的起始点和长度—这意味着你可以选择映射大文件的较小区域。

MappedByteBuffer 继承了 **ByteBuffer**，所以拥有 **ByteBuffer** 全部的方法。这里只展示了 `put()` 和 `get()` 的最简单用法，但是你也可以使用 `asCharBuffer()` 等方法。

使用前面的程序创建的文件长度为 128MB，可能比你的操作系统单次所允许的操作的内存要大。该文件似乎可以同时访问，因为它只有一部分被带进内存，而其他部分被交换出去。这样，一个非常大

的文件（最多 2GB）可以很容易地修改。**注意**，操作系统底层的文件映射工具用于性能的最大化。

7.6.1 性能

虽然旧的 I/O 流的性能通过使用 **NIO** 实现得到了改进，但是映射文件访问往往要快得多。下例带来一个简单的性能比较。代码示例：

```
// newio/MappedIO.java
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
import java.util.*;
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
  private static int numOfInts =      4_000_000;
  private static int numOfUbuffInts = 100_000;
  private abstract static class Tester {
    private String name;
    Tester(String name) {
      this.name = name;
    }

    public void runTest() {
      System.out.print(name + ": ");
      long start = System.nanoTime();
      test();
      double duration = System.nanoTime() - start;
      System.out.format("%.3f%n", duration/1.0e9);
    }

    public abstract void test();
  }

  private static Tester[] tests = {
    new Tester("Stream Write") {
      @Override
      public void test() {
        try(
          DataOutputStream dos =
            new DataOutputStream(
              new BufferedOutputStream(
                new FileOutputStream(
                  new File("temp.tmp"))))
        ) {
          for(int i = 0; i < numOfInts; i++)
            dos.writeInt(i);
        } catch(IOException e) {
```

```
43            throw new RuntimeException(e);
44          }
45        }
46      },
47      new Tester("Mapped Write") {
48        @Override
49        public void test() {
50          try(
51            FileChannel fc =
52              new RandomAccessFile("temp.tmp", "rw")
53                .getChannel()
54          ) {
55            IntBuffer ib =
56              fc.map(FileChannel.MapMode.READ_WRITE,
57                0, fc.size()).asIntBuffer();
58            for(int i = 0; i < numOfInts; i++)
59              ib.put(i);
60          } catch(IOException e) {
61            throw new RuntimeException(e);
62          }
63        }
64      },
65      new Tester("Stream Read") {
66        @Override
67        public void test() {
68          try(
69            DataInputStream dis =
70              new DataInputStream(
71                new BufferedInputStream(
72                  new FileInputStream("temp.tmp")))
73          ) {
74            for(int i = 0; i < numOfInts; i++)
75              dis.readInt();
76          } catch(IOException e) {
77            throw new RuntimeException(e);
78          }
79        }
80      },
81      new Tester("Mapped Read") {
82        @Override
83        public void test() {
84          try(
85            FileChannel fc = new FileInputStream(
86              new File("temp.tmp")).getChannel()
87          ) {
88            IntBuffer ib =
89              fc.map(FileChannel.MapMode.READ_ONLY,
90                0, fc.size()).asIntBuffer();
91            while(ib.hasRemaining())
92              ib.get();
93          } catch(IOException e) {
```

```
            throw new RuntimeException(e);
          }
        }
      },
      new Tester("Stream Read/Write") {
        @Override
        public void test() {
          try(
            RandomAccessFile raf =
              new RandomAccessFile(
                new File("temp.tmp"), "rw")
          ) {
            raf.writeInt(1);
            for(int i = 0; i < numOfUbuffInts; i++) {
              raf.seek(raf.length() - 4);
              raf.writeInt(raf.readInt());
            }
          } catch(IOException e) {
            throw new RuntimeException(e);
          }
        }
      },
      new Tester("Mapped Read/Write") {
        @Override
        public void test() {
          try(
            FileChannel fc = new RandomAccessFile(
              new File("temp.tmp"), "rw").getChannel()
          ) {
            IntBuffer ib =
              fc.map(FileChannel.MapMode.READ_WRITE,
                0, fc.size()).asIntBuffer();
            ib.put(0);
            for(int i = 1; i < numOfUbuffInts; i++)
              ib.put(ib.get(i - 1));
          } catch(IOException e) {
            throw new RuntimeException(e);
          }
        }
      }
    };
    public static void main(String[] args) {
      Arrays.stream(tests).forEach(Tester::runTest);
    }
  }
```

输出结果:

```
Stream Write: 0.615
Mapped Write: 0.050
Stream Read: 0.577
```

```
Mapped Read: 0.015
Stream Read/Write: 4.069
Mapped Read/Write: 0.013
```

Tester 使用了模板方法（Template Method）模式，它为匿名内部子类中定义的 `test()` 的各种实现创建一个测试框架。每个子类都执行一种测试，因此 `test()` 方法还提供了执行各种 I/O 活动的原型。

虽然映射的写似乎使用 **FileOutputStream**，但是文件映射中的所有输出必须使用 **RandomAccessFile**，就像前面代码中的读/写一样。

请注意，`test()` 方法包括初始化各种 I/O 对象的时间，因此，尽管映射文件的设置可能很昂贵，但是与流 I/O 相比，总体收益非常可观。

7.7 文件锁定

文件锁定可同步访问，因此文件可以共享资源。但是，争用同一文件的两个线程可能位于不同的 JVM 中，或者一个可能是 Java 线程，另一个可能是操作系统中的本机线程。文件锁对其他操作系统进程可见，因为 Java 文件锁定直接映射到本机操作系统锁定工具。

```java
// newio/FileLocking.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
  public static void main(String[] args) {
    try(
      FileOutputStream fos =
        new FileOutputStream("file.txt");
      FileLock fl = fos.getChannel().tryLock()
    ) {
      if(fl != null) {
        System.out.println("Locked File");
        TimeUnit.MILLISECONDS.sleep(100);
        fl.release();
        System.out.println("Released Lock");
      }
    } catch(IOException | InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
}
```

输出结果:

```
Locked File
Released Lock
```

通过调用 **FileChannel** 上的 `tryLock()` 或 `lock()`，可以获得整个文件的 **FileLock**。(**SocketChannel**、**DatagramChannel** 和 **ServerSocketChannel** 不需要锁定，因为它们本质上是单进程实体；通常不会在两个进程之间共享一个网络套接字)。

`tryLock()` 是非阻塞的。它试图获取锁，若不能获取（当其他进程已经持有相同的锁，并且它不是共享的)，它只是从方法调用返回。

`lock()` 会阻塞，直到获得锁，或者调用 `lock()` 的线程中断，或者调用 `lock()` 方法的通道关闭。使用 **FileLock.**`release()` 释放锁。

还可以使用

```
tryLock(long position, long size, boolean shared)
```

或

```
lock(long position, long size, boolean shared)
```

锁定文件的一部分，锁住 **size-position** 区域。第三个参数指定是否共享此锁。

虽然零参数锁定方法适应文件大小的变化，但是如果文件大小发生变化，具有固定大小的锁不会发生变化。如果从一个位置到另一个位置获得一个锁，并且文件的增长超过了 position + size ，那么超出 position + size 的部分没有被锁定。零参数锁定方法锁定整个文件，即使它在增长。

底层操作系统必须提供对独占锁或共享锁的支持。如果操作系统不支持共享锁并且对一个操作系统发出请求，则使用独占锁。可以使用 **FileLock.**`isShared()` 查询锁的类型（共享或独占)。

7.7.1 映射文件的部分锁定

文件映射通常用于非常大的文件。你可能需要锁定此类文件的某些部分，以便其他进程可以修改未锁定的部分。例如，数据库必须同时对许多用户可用。这里你可以看到两个线程，每个线程都锁定文件的不同部分:

```
// newio/LockingMappedFiles.java
// (c)2017 MindView LLC: see Copyright.txt
// 我们无法保证该代码是否适用于其他用途。
// 访问 http://OnJava8.com 了解更多本书信息。
// Locking portions of a mapped file
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
  static final int LENGTH = 0x8FFFFFF; // 128 MB
```

```
  static FileChannel fc;
  public static void
  main(String[] args) throws Exception {
    fc = new RandomAccessFile("test.dat", "rw")
      .getChannel();
    MappedByteBuffer out = fc.map(
      FileChannel.MapMode.READ_WRITE, 0, LENGTH);
    for(int i = 0; i < LENGTH; i++)
      out.put((byte)'x');
    new LockAndModify(out, 0, 0 + LENGTH/3);
    new LockAndModify(
      out, LENGTH/2, LENGTH/2 + LENGTH/4);
  }

  private static class LockAndModify extends Thread {
    private ByteBuffer buff;
    private int start, end;
    LockAndModify(ByteBuffer mbb, int start, int end) {
      this.start = start;
      this.end = end;
      mbb.limit(end);
      mbb.position(start);
      buff = mbb.slice();
      start();
    }

    @Override
    public void run() {
      try {
        // Exclusive lock with no overlap:
        FileLock fl = fc.lock(start, end, false);
        System.out.println(
          "Locked: "+ start +" to "+ end);
        // Perform modification:
        while(buff.position() < buff.limit() - 1)
          buff.put((byte)(buff.get() + 1));
        fl.release();
        System.out.println(
          "Released: " + start + " to " + end);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  }
}
```

输出结果:

```
Locked: 75497471 to 113246206
Locked: 0 to 50331647
Released: 75497471 to 113246206
```

```
Released: 0 to 50331647
```

LockAndModify 线程类设置缓冲区并创建要修改的 `slice()`，在 `run()` 中，锁在文件通道上获取（不能在缓冲区上获取锁—只能在通道上获取锁）。`lock()` 的调用非常类似于获取对象上的线程锁 —— 现在有了一个“临界区”，可以对文件的这部分进行独占访问。[1]

当 JVM 退出或关闭获取锁的通道时，锁会自动释放，但是你也可以显式地调用 **FileLock** 对象上的 `release()`，如上所示。

[1]在某些平台上，特别是 Windows，默认值可能非常难以查明。你可以使用 -Xss 标志调整堆栈大小。

一个有信念者所开发出的力量，大于个只有兴趣者。

附录 8

理解 equals 和 hashCode 方法

假设有一个容器使用 hash 函数，当你创建一个放到这个容器时，你必须定义 **hashCode()** 函数和 **equals()** 函数。这两个函数一起被用于 hash 容器中的查询操作。

8.1 equals 规范

当你创建一个类的时候，它自动继承自 **Objcet** 类。如果你不覆写 **equals()**，你将会获得 **Objcet** 对象的 **equals()** 函数。默认情况下，这个函数会比较对象的地址。所以只有你在比较同一个对象的时候，你才会获得 **true**。默认的情况是“区分度最高的”。

```java
// equalshashcode/DefaultComparison.java
class DefaultComparison {
    private int i, j, k;
    DefaultComparison(int i, int j, int k) {
        this.i = i;
        this.j = j;
        this.k = k;
    }

    public static void main(String[] args) {
        DefaultComparison
        a = new DefaultComparison(1, 2, 3),
        b = new DefaultComparison(1, 2, 3);
        System.out.println(a == a);
        System.out.println(a == b);
    }
}
/*
Output:
true
false
*/
```

通常你会希望放宽这个限制。一般来说如果两个对象有相同的类型和相同的字段，你会认为这两个

对象相等，但也会有一些你不想加入 **equals()** 函数中来比较的字段。这是类型设计的一部分。

一个合适的 **equals()** 函数必须满足以下五点条件：1. 反身性：对于任何 **x**，**x.equals(x)** 应该返回 **true**。2. 对称性：对于任何 **x** 和 **y**，**x.equals(y)** 应该返回 **true** 当且仅当 **y.equals(x)** 返回 **true**。3. 传递性：对于任何 **x**,**y**, 还有 **z**，如果 **x.equals(y)** 返回 **true** 并且 **y.equals(z)** 返回 **true**，那么 **x.equals(z)** 应该返回 **true**。4. 一致性：对于任何 **x** 和 **y**，在对象没有被改变的情况下，多次调用 **x.equals(y)** 应该总是返回 **true** 或者 **false**。5. 对于任何非 **null** 的 **x**，**x.equals(null)** 应该返回 **false**。

下面是满足这些条件的测试，并且判断对象是否和自己相等（我们这里称呼其为**右值**）：1. 如果**右值**是 **null**，那么不相等。2. 如果**右值**是 **this**，那么两个对象相等。3. 如果**右值**不是同一个类型或者子类，那么两个对象不相等。4. 如果所有上面的检查通过了，那么你必须决定 **右值**中的哪些字段是重要的，然后比较这些字段。Java 7 引入了 **Objects** 类型来帮助这个流程，这样我们能够写出更好的 **equals()** 函数。

下面的例子比较了不同类型的 **Equality** 类。为了避免重复的代码，我们使用*工厂函数设计模式*来实现样例。**EqualityFactory** 接口提供 **make()** 函数来生成一个 **Equaity** 对象，这样不同的 **EqualityFactory** 能够生成 **Equality** 不同的子类。

```
// equalshashcode/EqualityFactory.java
import java.util.*;
interface EqualityFactory {
    Equality make(int i, String s, double d);
}
```

现在我们来定义 **Equality**，它包含三个字段（所有的字段我们认为在比较中都很重要）和一个 **equals()** 函数用来满足上述的四种检查。构造函数展示了它的类名来保证我们在执行我们想要的测试：

```
// equalshashcode/Equality.java
import java.util.*;
public class Equality {
    protected int i;
    protected String s;
    protected double d;public Equality(int i, String s, double d) {
        this.i = i;
        this.s = s;
        this.d = d;
        System.out.println("made 'Equality'");
    }

    @Override
    public boolean equals(Object rval) {
        if(rval == null)
            return false;
        if(rval == this)
            return true;
```

```
        if(!(rval instanceof Equality))
            return false;
        Equality other = (Equality)rval;
        if(!Objects.equals(i, other.i))
            return false;
        if(!Objects.equals(s, other.s))
            return false;
        if(!Objects.equals(d, other.d))return false;
            return true;
    }

    public void test(String descr, String expected, Object rval) {
        System.out.format("-- Testing %s --%n" + "%s instanceof
            Equality: %s%n" +
        "Expected %s, got %s%n",
        descr, descr, rval instanceof Equality,
        expected, equals(rval));
    }

    public static void testAll(EqualityFactory eqf) {
        Equality
        e = eqf.make(1, "Monty", 3.14),
        eq = eqf.make(1, "Monty", 3.14),
        neq = eqf.make(99, "Bob", 1.618);
        e.test("null", "false", null);
        e.test("same object", "true", e);
        e.test("different type",
        "false", Integer.valueOf(99));e.test("same values", "true", eq)
            ;
        e.test("different values", "false", neq);
    }

    public static void main(String[] args) {
        testAll( (i, s, d) -> new Equality(i, s, d));
    }

}
/*
Output:
made 'Equality'
made 'Equality'
made 'Equality'
-- Testing null --
null instanceof Equality: false
Expected false, got false
-- Testing same object --
same object instanceof Equality: true
Expected true, got true
-- Testing different type --
different type instanceof Equality: false
Expected false, got false-- Testing same values --
```

```
same values instanceof Equality: true
Expected true, got true
-- Testing different values --
different values instanceof Equality: true
Expected false, got false
*/
```

testAll() 执行了我们期望的所有不同类型对象的比较。它使用工厂创建了 **Equality** 对象。

在 **main()** 里，请注意对 **testAll()** 的调用很简单。因为 **EqualityFactory** 有着单一的函数，它能够和 lambda 表达式一起使用来表示 **make()** 函数。

上述的 **equals()** 函数非常繁琐，并且我们能够将其简化成规范的形式，请注意：1. **instanceof** 检查减少了 **null** 检查的需要。2. 和 **this** 的比较是多余的。一个正确书写的 **equals()** 函数能正确地和自己比较。

因为 **&&** 是一个短路比较，它会在第一次遇到失败的时候退出并返回 **false**。所以，通过使用 **&&** 将检查链接起来，我们可以写出更精简的 **equals()** 函数：

```java
// equalshashcode/SuccinctEquality.java
import java.util.*;
public class SuccinctEquality extends Equality {
    public SuccinctEquality(int i, String s, double d) {
        super(i, s, d);
        System.out.println("made 'SuccinctEquality'");
    }

    @Override
    public boolean equals(Object rval) {
        return rval instanceof SuccinctEquality &&
        Objects.equals(i, ((SuccinctEquality)rval).i) &&
        Objects.equals(s, ((SuccinctEquality)rval).s) &&
        Objects.equals(d, ((SuccinctEquality)rval).d);
    }
    public static void main(String[] args) {
        Equality.testAll( (i, s, d) ->
        new SuccinctEquality(i, s, d));
    }

}
/* Output:
made 'Equality'
made 'SuccinctEquality'
made 'Equality'
made 'SuccinctEquality'
made 'Equality'
made 'SuccinctEquality'
-- Testing null --
null instanceof Equality: false
Expected false, got false
-- Testing same object --
```

```
same object instanceof Equality: true
Expected true, got true
-- Testing different type --
different type instanceof Equality: false
Expected false, got false
-- Testing same values --
same values instanceof Equality: true
Expected true, got true
-- Testing different values --different values instanceof Equality:
    true
Expected false, got false
*/
```

对于每个 **SuccinctEquality**，基类构造函数在派生类构造函数前被调用，输出显示我们依然获得了正确的结果，你可以发现短路返回已经发生了，不然的话，**null** 测试和“不同类型”的测试会在 **equals()** 函数下面的比较中强制转化的时候抛出异常。**Objects.equals()** 会在你组合其他类型的时候发挥很大的作用。

```java
// equalshashcode/ComposedEquality.java
import java.util.*;
class Part {
    String ss;
    double dd;

    Part(String ss, double dd) {
        this.ss = ss;
        this.dd = dd;
    }

    @Override
    public boolean equals(Object rval) {
        return rval instanceof Part &&
        Objects.equals(ss, ((Part)rval).ss) &&
        Objects.equals(dd, ((Part)rval).dd);
    }

}

public class ComposedEquality extends SuccinctEquality {
    Part part;
    public ComposedEquality(int i, String s, double d) {
        super(i, s, d);
        part = new Part(s, d);
        System.out.println("made 'ComposedEquality'");
    }
    @Override
    public boolean equals(Object rval) {
        return rval instanceof ComposedEquality &&
        super.equals(rval) &&
```

```
        Objects.equals(part,
        ((ComposedEquality)rval).part);

    }

    public static void main(String[] args) {
        Equality.testAll( (i, s, d) ->
        new ComposedEquality(i, s, d));
    }
}
/*
Output:
made 'Equality'
made 'SuccinctEquality'
made 'ComposedEquality'
made 'Equality'
made 'SuccinctEquality'
made 'ComposedEquality'
made 'Equality'
made 'SuccinctEquality'
made 'ComposedEquality'
-- Testing null --null instanceof Equality: false
Expected false, got false
-- Testing same object --
same object instanceof Equality: true
Expected true, got true
-- Testing different type --
different type instanceof Equality: false
Expected false, got false
-- Testing same values --
same values instanceof Equality: true
Expected true, got true
-- Testing different values --
different values instanceof Equality: true
Expected false, got false
*/
```

注意 super.equals() 这个调用，没有必要重新发明它（因为你不总是有权限访问基类所有的必要字段）

8.1.1 不同子类的相等性

继承意味着两个不同子类的对象当其向上转型的时候可以是相等的。假设你有一个 Animal 对象的集合。这个集合天然接受 **Animal** 的子类。在这个例子中是 **Dog** 和 **Pig**。每个 **Animal** 有一个 **name** 和 **size**，还有唯一的内部 **id** 数字。

我们通过 **Objects** 类，以规范的形式定义 **equals()** 函数和 **hashCode()**。但是我们只能在基类 **Animal** 中定义他们。并且我们在这两个函数中没有包含 **id** 字段。从 **equals()** 函数的角度看待，这意

味着我们只关心它是否是 **Animal**，而不关心是否是 **Animal** 的某个子类。

```
// equalshashcode/SubtypeEquality.java
import java.util.*;
enum Size { SMALL, MEDIUM, LARGE }
class Animal {
    private static int counter = 0;
    private final int id = counter++;
    private final String name;
    private final Size size;
    Animal(String name, Size size) {
        this.name = name;
        this.size = size;
    }
    @Override
    public boolean equals(Object rval) {
        return rval instanceof Animal &&
        // Objects.equals(id, ((Animal)rval).id) && // [1]
        Objects.equals(name, ((Animal)rval).name) &&
        Objects.equals(size, ((Animal)rval).size);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, size);
        // return Objects.hash(name, size, id); // [2]
    }

    @Override
    public String toString() {
        return String.format("%s[%d]: %s %s %x",
        getClass().getSimpleName(), id,
        name, size, hashCode());
    }
}

class Dog extends Animal {
    Dog(String name, Size size) {
    super(name, size);
    }
}

class Pig extends Animal {
    Pig(String name, Size size) {
    super(name, size);
    }
}

public class SubtypeEquality {
    public static void main(String[] args) {
        Set<Animal> pets = new HashSet<>();
```

```
        pets.add(new Dog("Ralph", Size.MEDIUM));
        pets.add(new Pig("Ralph", Size.MEDIUM));
        pets.forEach(System.out::println);
    }
}
/*
Output:
Dog[0]: Ralph MEDIUM a752aeee
*/
```

如果我们只考虑类型的话，某些情况下它的确说得通——只从基类的角度看待问题，这是李氏替换原则的基石。这个代码完美符合替换理论因为派生类没有添加任何额外不再基类中的额外函数。派生类只是在表现上不同，而不是在接口上。（当然这不是常态）

但是当我们提供了两个有着相同数据的不同的对象类型，然后将他们放置在 **HashSet** 中。只有他们中的一个能存活。这强调了 **equals()** 不是完美的数学理论，而只是机械般的理论。**hashCode()** 和 **equals()** 必须能够允许类型在 hash 数据结构中正常工作。例子中 **Dog** 和 **Pig** 会被映射到同 **HashSet** 的同一个桶中。这个时候，**HashSet** 回退到 **equals()** 来区分对象，但是 **equals()** 也认为两个对象是相同的。**HashSet** 因为已经有一个相同的对象了，所以没有添加 **Pig**。我们依然能够通过使得其他字段对象不同来让例子能够正常工作。在这里每个 **Animal** 已经有了一个独一无二的 **id** ，所以你能够取消 **equals()** 函数中的 **[1]** 行注释，或者取消 **hashCode()** 函数中的 **[2]** 行注释。按照规范，你应该同时完成这两个操作，如此能够将所有“不变的”字段包含在两个操作中（“不变”所以 **equals()** 和 **hashCode()** 在哈希数据结构中的排序和取值时，不会生成不同的值。我将“不变的”放在引号中因为你必须计算出是否已经发生变化）。

> 旁注：在 **hashCode()** 中，如果你只能够使用一个字段，使用 **Objcets.hashCode()**。如果你使用多个字段，那么使用 **Objects.hash()**。

我们也可以通过标准方式，将 **equals()** 定义在子类中（不包含 **id** ）解决这个问题：

```
// equalshashcode/SubtypeEquality2.java
import java.util.*;
class Dog2 extends Animal {
    Dog2(String name, Size size) {
        super(name, size);
    }

    @Override
    public boolean equals(Object rval) {
        return rval instanceof Dog2 &&super.equals(rval);
    }
}

class Pig2 extends Animal {
    Pig2(String name, Size size) {
    super(name, size);
```

```
    }

    @Override
    public boolean equals(Object rval) {
        return rval instanceof Pig2 &&
        super.equals(rval);
    }
}

public class SubtypeEquality2 {
    public static void main(String[] args) {
        Set<Animal> pets = new HashSet<>();
        pets.add(new Dog2("Ralph", Size.MEDIUM));
        pets.add(new Pig2("Ralph", Size.MEDIUM));
        pets.forEach(System.out::println);
    }
}
/*
Output:
Dog2[0]: Ralph MEDIUM a752aeee
Pig2[1]: Ralph MEDIUM a752aeee
*/
```

注意 **hashCode()** 是独一无二的，但是因为对象不再 **equals()**，所以两个函数都出现在 **HashSet** 中。另外，**super.equals()** 意味着我们不需要访问基类的 **private** 字段。

一种说法是 Java 从 **equals()** 和 **hashCode()** 的定义中分离了可替代性。我们仍然能够将 **Dog** 和 **Pig** 放置在 **Set<Animal>** 中，无论 **equals()** 和 **hashCode()** 是如何定义的，但是对象不会在哈希数据结构中正常工作，除非这些函数能够被合理定义。不幸的是，**equals()** 不总是和 **hashCode()** 一起使用，这在你尝试为了某个特殊类型避免定义它的时候会让问题复杂化。并且这也是为什么遵循规范是有价值的。然而这会变得更加复杂，因为你不总是需要定义其中一个函数。

8.2 哈希和哈希码

在 集合 章节中，我们使用预先定义的类作为 HashMap 的键。这些示例之所以有用，是因为预定义的类具有所有必需的连线，以使它们正确地充当键。

当创建自己的类作为 HashMap 的键时，会发生一个常见的陷阱，从而忘记进行必要的接线。例如，考虑一个将 Earthhog 对象与 Prediction 对象匹配的天气预报系统。这似乎很简单：使用 Groundhog 作为键，使用 Prediction 作为值：

```
// equalshashcode/Groundhog.java
// Looks plausible, but doesn't work as a HashMap key
public class Groundhog {
    protected int number;
    public Groundhog(int n) { number = n; }
```

```
    @Override
    public String toString() {
        return "Groundhog #" + number;
    }
}
```

```
// equalshashcode/Prediction.java
// Predicting the weather
import java.util.*;
public class Prediction {
    private static Random rand = new Random(47);
    @Override
    public String toString() {
        return rand.nextBoolean() ?
                "Six more weeks of Winter!" : "Early Spring!";
    }
}
```

```
// equalshashcode/SpringDetector.java
// What will the weather be?
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import java.lang.reflect.*;
public class SpringDetector {
    public static <T extends Groundhog>
    void detectSpring(Class<T> type) {
        try {
            Constructor<T> ghog =
                    type.getConstructor(int.class);
            Map<Groundhog, Prediction> map =
                    IntStream.range(0, 10)
                            .mapToObj(i -> {
                                try {
                                    return ghog.newInstance(i);
                                } catch(Exception e) {
                                    throw new RuntimeException(e);
                                }
                            })
                            .collect(Collectors.toMap(
                                    Function.identity(),
                                    gh -> new Prediction()));
            map.forEach((k, v) ->
                    System.out.println(k + ": " + v));
            Groundhog gh = ghog.newInstance(3);
            System.out.println(
                    "Looking up prediction for " + gh);
            if(map.containsKey(gh))
                System.out.println(map.get(gh));
            else
```

```
                System.out.println("Key not found: " + gh);
        } catch(NoSuchMethodException |
                IllegalAccessException |
                InvocationTargetException |
                InstantiationException e) {
            throw new RuntimeException(e);
        }
    }
    public static void main(String[] args) {
        detectSpring(Groundhog.class);
    }
}
/* Output:
Groundhog #3: Six more weeks of Winter!
Groundhog #0: Early Spring!
Groundhog #8: Six more weeks of Winter!
Groundhog #6: Early Spring!
Groundhog #4: Early Spring!
Groundhog #2: Six more weeks of Winter!
Groundhog #1: Early Spring!
Groundhog #9: Early Spring!
Groundhog #5: Six more weeks of Winter!
Groundhog #7: Six more weeks of Winter!
Looking up prediction for Groundhog #3
Key not found: Groundhog #3
*/
```

每个 Groundhog 都被赋予了一个常数，因此你可以通过如下的方式在 HashMap 中寻找对应的 Prediction。“给我一个和 Groundhog#3 相关联的 Prediction”。而 Prediction 通过一个随机生成的 boolean 来选择天气。`detectSpring()` 方法通过反射来实例化 Groundhog 类，或者它的子类。稍后，当我们继承一种新型的“Groundhog”以解决此处演示的问题时，这将派上用场。

这里的 HashMap 被 Groundhog 和其相关联的 Prediction 充满。并且上面展示了 HashMap 里面填充的内容。接下来我们使用填充了常数 3 的 Groundhog 作为 key 用于寻找对应的 Prediction。(这个键值对肯定在 Map 中)。

这看起来十分简单，但是这样做并没有奏效 —— 它无法找到数字 3 这个键。问题出在 Groundhog 自动地继承自基类 Object，所以这里使用 Object 的 hashCode0 方法生成散列码，而它默认是使用对象的地址计算散列码。因此，由 Groundhog(3) 生成的第一个实例的散列码与由 Groundhog(3) 生成的第二个实例的散列码是不同的，而我们正是使用后者进行查找的。

我们需要恰当的重写 hashCode() 方法。但是它仍然无法正常运行，除非你同时重写 equals() 方法，它也是 Object 的一部分。HashMap 使用 equals() 判断当前的键是否与表中存在的键相同。

这是因为默认的 Object.equals() 只是比较对象的地址，所以一个 Groundhog(3) 并不等于另一个 Groundhog(3)，因此，如果要使用自己的类作为 HashMap 的键，必须同时重载 hashCode() 和 equals()，如下所示：

```java
// equalshashcode/Groundhog2.java
// A class that's used as a key in a HashMap
// must override hashCode() and equals()
import java.util.*;
public class Groundhog2 extends Groundhog {
    public Groundhog2(int n) { super(n); }
    @Override
    public int hashCode() { return number; }
    @Override
    public boolean equals(Object o) {
        return o instanceof Groundhog2 &&
                Objects.equals(
                        number, ((Groundhog2)o).number);
    }
}
```

```java
// equalshashcode/SpringDetector2.java
// A working key
public class SpringDetector2 {
    public static void main(String[] args) {
        SpringDetector.detectSpring(Groundhog2.class);
    }
}
/* Output:
Groundhog #0: Six more weeks of Winter!
Groundhog #1: Early Spring!
Groundhog #2: Six more weeks of Winter!
Groundhog #3: Early Spring!
Groundhog #4: Early Spring!
Groundhog #5: Six more weeks of Winter!
Groundhog #6: Early Spring!
Groundhog #7: Early Spring!
Groundhog #8: Six more weeks of Winter!
Groundhog #9: Six more weeks of Winter!
Looking up prediction for Groundhog #3
Early Spring!
*/
```

Groundhog2.hashCode0 返回 Groundhog 的标识数字（编号）作为散列码。在此例中，程序员负责确保不同的 Groundhog 具有不同的编号。hashCode() 并不需要总是能够返回唯一的标识码（稍后你会理解其原因），但是 equals() 方法必须严格地判断两个对象是否相同。此处的 equals() 是判断 Groundhog 的号码, 所以作为 HashMap 中的键, 如果两个 Groundhog2 对象具有相同的 Groundhog 编号，程序就出错了。

如何定义 equals() 方法在上一节 equals 规范中提到了。输出表明我们现在的输出是正确的。

8.2.1 理解 hashCode

前面的例子只是正确解决问题的第一步。它只说明，如果不为你的键覆盖 hashCode() 和 equals()，那么使用散列的数据结构（HashSet，HashMap，LinkedHashst 或 LinkedHashMap）就无法正确处理你的键。然而，要很好地解决此问题，你必须了解这些数据结构的内部构造。

首先，使用散列的目的在于：想要使用一个对象来查找另一个对象。不过使用 TreeMap 或者你自己实现的 Map 也可以达到此目的。与散列实现相反，下面的示例用一对 ArrayLists 实现了一个 Map，与 AssociativeArray.java 不同，这其中包含了 Map 接口的完整实现，因此提供了 entrySet() 方法：

```
// equalshashcode/SlowMap.java
// A Map implemented with ArrayLists
import java.util.*;
import onjava.*;
public class SlowMap<K, V> extends AbstractMap<K, V> {
    private List<K> keys = new ArrayList<>();
    private List<V> values = new ArrayList<>();
    @Override
    public V put(K key, V value) {
        V oldValue = get(key); // The old value or null
        if(!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else
            values.set(keys.indexOf(key), value);
        return oldValue;
    }
    @Override
    public V get(Object key) { // key: type Object, not K
        if(!keys.contains(key))
            return null;
        return values.get(keys.indexOf(key));
    }
    @Override
    public Set<Map.Entry<K, V>> entrySet() {
        Set<Map.Entry<K, V>> set= new HashSet<>();
        Iterator<K> ki = keys.iterator();
        Iterator<V> vi = values.iterator();
        while(ki.hasNext())
            set.add(new MapEntry<>(ki.next(), vi.next()));
        return set;
    }
    public static void main(String[] args) {
        SlowMap<String,String> m= new SlowMap<>();
        m.putAll(Countries.capitals(8));
        m.forEach((k, v) ->
                System.out.println(k + "=" + v));
        System.out.println(m.get("BENIN"));
        m.entrySet().forEach(System.out::println);
```

```
    }
}
/* Output:
CAMEROON=Yaounde
ANGOLA=Luanda
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
Porto-Novo
CAMEROON=Yaounde
ANGOLA=Luanda
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
*/
```

put() 方法只是将键与值放入相应的 ArrayList。为了与 Map 接口保持一致，它必须返回旧的键，或者在没有任何旧键的情况下返回 null。

同样遵循了 Map 规范，get() 会在键不在 SlowMap 中的时候产生 null。如果键存在，它将被用来查找表示它在 keys 列表中的位置的数值型索引，并且这个数字被用作索引来产生与 values 列表相关联的值。注意，在 get() 中 key 的类型是 Object，而不是你所期望的参数化类型 K（并且是在 AssociativeArrayjava 中真正使用的类型），这是将泛型注入到 Java 语言中的时刻如此之晚所导致的结果-如果泛型是 Java 语言最初就具备的属性，那么 get() 就可以执行其参数的类型。

Map.entrySet() 方法必须产生一个 Map.Entry 对象集。但是，Map.Entry 是一个接口，用来描述依赖于实现的结构，因此如果你想要创建自己的 Map 类型，就必须同时定义 Map.Entry 的实现：

```
// equalshashcode/MapEntry.java
// A simple Map.Entry for sample Map implementations
import java.util.*;
public class MapEntry<K, V> implements Map.Entry<K, V> {
    private K key;
    private V value;
    public MapEntry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    @Override
    public K getKey() { return key; }
    @Override
    public V getValue() { return value; }
    @Override
```

```
    public V setValue(V v) {
        V result = value;
        value = v;
        return result;
    }
    @Override
    public int hashCode() {
        return Objects.hash(key, value);
    }
    @SuppressWarnings("unchecked")
    @Override
    public boolean equals(Object rval) {
        return rval instanceof MapEntry &&
                Objects.equals(key,
                        ((MapEntry<K, V>)rval).getKey()) &&
                Objects.equals(value,
                        ((MapEntry<K, V>)rval).getValue());
    }
    @Override
    public String toString() {
        return key + "=" + value;
    }
}
```

这里 equals 方法的实现遵循了 equals 规范。在 Objects 类中有一个非常熟悉的方法可以帮助创建 hashCode() 方法：Objects.hash()。当你定义含有超过一个属性的对象的 `hashCode()` 时，你可以使用这个方法。如果你的对象只有一个属性，可以直接使用 `Objects.hashCode()`。

尽管这个解决方案非常简单，并且看起来在 SlowMap.main() 的琐碎测试中可以正常工作，但是这并不是一个恰当的实现，因为它创建了键和值的副本。entrySet() 的恰当实现应该在 Map 中提供视图，而不是副本，并且这个视图允许对原始映射表进行修改（副本就不行）。

8.2.2 为了速度而散列

SlowMap.java 说明了创建一种新的 Map 并不困难。但是正如它的名称 SlowMap 所示，它不会很快，所以如果有更好的选择，就应该放弃它。它的问题在于对键的查询，键没有按照任何特定顺序保存，所以只能使用简单的线性查询，而线性查询是最慢的查询方式。

散列的价值在于速度：散列使得查询得以快速进行。由于瓶颈位于键的查询速度，因此解决方案之一就是保持键的排序状态，然后使用 Collections.binarySearch() 进行查询。

散列则更进一步，它将键保存在某处，以便能够很快找到。存储一组元素最快的数据结构是数组，所以使用它来表示键的信息（请小心留意，我是说键的信息，而不是键本身）。但是因为数组不能调整容量，因此就有一个问题：我们希望在 Map 中保存数量不确定的值，但是如果键的数量被数组的容量限制了，该怎么办呢？

答案就是：数组并不保存键本身。而是通过键对象生成一个数字，将其作为数组的下标。这个数字

就是散列码，由定义在 Object 中的、且可能由你的类覆盖的 hashCode() 方法（在计算机科学的术语中称为散列函数）生成。

于是查询一个值的过程首先就是计算散列码，然后使用散列码查询数组。如果能够保证没有冲突（如果值的数量是固定的，那么就有可能），那可就有了一个完美的散列函数，但是这种情况只是特例。。通常，冲突由外部链接处理：数组并不直接保存值，而是保存值的 list。然后对 list 中的值使用 equals() 方法进行线性的查询。这部分的查询自然会比较慢，但是，如果散列函数好的话，数组的每个位置就只有较少的值。因此，不是查询整个 list，而是快速地跳到数组的某个位置，只对很少的元素进行比较。这便是 HashMap 会如此快的原因。

理解了散列的原理，我们就能够实现一个简单的散列 Map 了：

```java
// equalshashcode/SimpleHashMap.java
// A demonstration hashed Map
import java.util.*;
import onjava.*;
public
class SimpleHashMap<K, V> extends AbstractMap<K, V> {
    // Choose a prime number for the hash table
// size, to achieve a uniform distribution:
    static final int SIZE = 997;
    // You can't have a physical array of generics,
// but you can upcast to one:
    @SuppressWarnings("unchecked")
    LinkedList<MapEntry<K, V>>[] buckets =
            new LinkedList[SIZE];
    @Override
    public V put(K key, V value) {
        V oldValue = null;
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null)
            buckets[index] = new LinkedList<>();
        LinkedList<MapEntry<K, V>> bucket = buckets[index];
        MapEntry<K, V> pair = new MapEntry<>(key, value);
        boolean found = false;
        ListIterator<MapEntry<K, V>> it =
                bucket.listIterator();
        while(it.hasNext()) {
            MapEntry<K, V> iPair = it.next();
            if(iPair.getKey().equals(key)) {
                oldValue = iPair.getValue();
                it.set(pair); // Replace old with new
                found = true;
                break;
            }
        }
        if(!found)
            buckets[index].add(pair);
        return oldValue;
```

```
    }
    @Override
    public V get(Object key) {
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null) return null;
        for(MapEntry<K, V> iPair : buckets[index])
            if(iPair.getKey().equals(key))
                return iPair.getValue();
        return null;
    }
    @Override
    public Set<Map.Entry<K, V>> entrySet() {
        Set<Map.Entry<K, V>> set= new HashSet<>();
        for(LinkedList<MapEntry<K, V>> bucket : buckets) {
            if(bucket == null) continue;
            for(MapEntry<K, V> mpair : bucket)
                set.add(mpair);
        }
        return set;
    }
    public static void main(String[] args) {
        SimpleHashMap<String,String> m =
                new SimpleHashMap<>();
        m.putAll(Countries.capitals(8));
        m.forEach((k, v) ->
                System.out.println(k + "=" + v));
        System.out.println(m.get("BENIN"));
        m.entrySet().forEach(System.out::println);
    }
}
/* Output:
CAMEROON=Yaounde
ANGOLA=Luanda
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
Porto-Novo
CAMEROON=Yaounde
ANGOLA=Luanda
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
*/
```

由于散列表中的“槽位”（slot）通常称为桶位（bucket），因此我们将表示实际散列表的数组命名

为 bucket，为使散列分布均匀，桶的数量通常使用质数[1]。注意，为了能够自动处理冲突，使用了一个 LinkedList 的数组；每一个新的元素只是直接添加到 list 尾的某个特定桶位中。即使 Java 不允许你创建泛型数组，那你也可以创建指向这种数组的引用。这里，向上转型为这种数组是很方便的，这样可以防止在后面的代码中进行额外的转型。

对于 put() 方法，hashCode() 将针对键而被调用，并且其结果被强制转换为正数。为了使产生的数字适合 bucket 数组的大小，取模操作符将按照该数组的尺寸取模。如果数组的某个位置是 null，这表示还没有元素被散列至此，所以，为了保存刚散列到该定位的对象，需要创建一个新的 LinkedList。一般的过程是，查看当前位置的 ist 中是否有相同的元素，如果有，则将旧的值赋给 oldValue，然后用新的值取代旧的值。标记 found 用来跟踪是否找到（相同的）旧的键值对，如果没有，则将新的对添加到 list 的末尾。

get() 方法按照与 put() 方法相同的方式计算在 buckets 数组中的索引（这很重要，因为这样可以保证两个方法可以计算出相同的位置）如果此位置有 LinkedList 存在，就对其进行查询。

注意，这个实现并不意味着对性能进行了调优，它只是想要展示散列映射表执行的各种操作。如果你浏览一下 java.util.HashMap 的源代码，你就会看到一个调过优的实现。同样，为了简单，SimpleHashMap 使用了与 SlowMap 相同的方式来实现 entrySet()，这个方法有些过于简单，不能用于通用的 Map。

8.2.3 重写 hashCode()

在明白了如何散列之后，编写自己的 hashCode() 就更有意义了。

首先，你无法控制 bucket 数组的下标值的产生。这个值依赖于具体的 HashMap 对象的容量，而容量的改变与容器的充满程度和负载因子（本章稍后会介绍这个术语）有关。hashCode() 生成的结果，经过处理后成为桶位的下标（在 SimpleHashMap 中，只是对其取模，模数为 bucket 数组的大小）。

设计 hashCode() 时最重要的因素就是：无论何时，对同一个对象调用 hashCode() 都应该生成同样的值。如果在将一个对象用 put() 添加进 HashMap 时产生一个 hashCode() 值，而用 get() 取出时却产生了另一个 hashCode() 值，那么就无法重新取得该对象了。所以，如果你的 hashCode() 方法依赖于对象中易变的数据，用户就要当心了，因为此数据发生变化时，hashCode() 就会生成一个不同的散列码，相当于产生了一个不同的键。

此外，也不应该使 hashCode() 依赖于具有唯一性的对象信息，尤其是使用 this 值，这只能产生很糟糕的 hashCode()，因为这样做无法生成一个新的键，使之与 put() 中原始的键值对中的键相同。这正是 SpringDetector.java 的问题所在，因为它默认的 hashCode0 使用的是对象的地址。所以，应该使用对象内有意义的识别信息。

下面以 String 类为例。String 有个特点：如果程序中有多个 String 对象，都包含相同的字符串序

[1]引自 Brian Goetz, Java Concurrency in Practice 一书的作者，该书由 Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea 联合著作 (Addison-Wesley 出版社, 2006)。

列，那么这些 String 对象都映射到同一块内存区域。所以 new String("hello") 生成的两个实例，虽然是相互独立的，但是对它们使用 hashCode() 应该生成同样的结果。通过下面的程序可以看到这种情况：

```
// equalshashcode/StringHashCode.java
public class StringHashCode {
    public static void main(String[] args) {
        String[] hellos = "Hello Hello".split(" ");
        System.out.println(hellos[0].hashCode());
        System.out.println(hellos[1].hashCode());
    }
}
/* Output:
69609650
69609650
*/
```

对于 String 而言，hashCode() 明显是基于 String 的内容的。

因此，要想使 hashCode() 实用，它必须速度快，并且必须有意义。也就是说，它必须基于对象的内容生成散列码。记得吗，散列码不必是独一无二的（应该更关注生成速度，而不是唯一性），但是通过 hashCode() 和 equals() ，必须能够完全确定对象的身份。

因为在生成桶的下标前，hashCode() 还需要做进一步的处理，所以散列码的生成范围并不重要，只要是 int 即可。

还有另一个影响因素：好的 hashCode() 应该产生分布均匀的散列码。如果散列码都集中在一块，那么 HashMap 或者 HashSet 在某些区域的负载会很重，这样就不如分布均匀的散列函数快。

在 Effective Java Programming Language Guide（Addison-Wesley 2001）这本书中，Joshua Bloch 为怎样写出一份像样的 hashCode() 给出了基本的指导：

1. 给 int 变量 result 赋予某个非零值常量，例如 17。
2. 为对象内每个有意义的字段（即每个可以做 equals）操作的字段计算出一个 int 散列码 c：

字段类型	计算公式
boolean	c = (f ? 0 : 1)
byte , char , short , or int	c = (int)f
long	c = (int)(f ^ (f>>>32))
float	c = Float.floatToIntBits(f);
double	long l =Double.doubleToLongBits(f); c = (int)(l ^ (l >>>
Object , where equals() calls equals() for this field	c = f.hashCode()

字段类型	计算公式
Array	应用以上规则到每一个元素中

3. 合并计算得到的散列码：**result = 37 * result + c;**
4. 返回 result。
5. 检查 hashCode() 最后生成的结果，确保相同的对象有相同的散列码。

下面便是遵循这些指导的一个例子。提示，你没有必要书写像如下的代码 —— 相反，使用 `Objects.hash()` 去用于散列多字段的对象（如同在本例中的那样），然后使用 `Objects.hashCode()` 如散列单字段的对象。

```
// equalshashcode/CountedString.java
// Creating a good hashCode()
import java.util.*;
public class CountedString {
    private static List<String> created =
            new ArrayList<>();
    private String s;
    private int id = 0;
    public CountedString(String str) {
        s = str;
        created.add(s);
// id is the total number of instances
// of this String used by CountedString:
        for(String s2 : created)
            if(s2.equals(s))
                id++;
    }
    @Override
    public String toString() {
        return "String: " + s + " id: " + id +
                " hashCode(): " + hashCode();
    }
    @Override
    public int hashCode() {
// The very simple approach:
// return s.hashCode() * id;
// Using Joshua Bloch's recipe:
        int result = 17;
        result = 37 * result + s.hashCode();
        result = 37 * result + id;
        return result;
    }
    @Override
    public boolean equals(Object o) {
        return o instanceof CountedString &&
```

```
                Objects.equals(s, ((CountedString)o).s) &&
                Objects.equals(id, ((CountedString)o).id);
    }
    public static void main(String[] args) {
        Map<CountedString,Integer> map = new HashMap<>();
        CountedString[] cs = new CountedString[5];
        for(int i = 0; i < cs.length; i++) {
            cs[i] = new CountedString("hi");
            map.put(cs[i], i); // Autobox int to Integer
        }
        System.out.println(map);
        for(CountedString cstring : cs) {
            System.out.println("Looking up " + cstring);
            System.out.println(map.get(cstring));
        }
    }
}
/* Output:
{String: hi id: 4 hashCode(): 146450=3, String: hi id:
5 hashCode(): 146451=4, String: hi id: 2 hashCode():
146448=1, String: hi id: 3 hashCode(): 146449=2,
String: hi id: 1 hashCode(): 146447=0}
Looking up String: hi id: 1 hashCode(): 146447
0
Looking up String: hi id: 2 hashCode(): 146448
1
Looking up String: hi id: 3 hashCode(): 146449
2
Looking up String: hi id: 4 hashCode(): 146450
3
Looking up String: hi id: 5 hashCode(): 146451
4
*/
```

CountedString 由一个 String 和一个 id 组成，此 id 代表包含相同 String 的 CountedString 对象的编号。所有的 String 都被存储在 static ArrayList 中，在构造器中通过迭代遍历此 ArrayList 完成对 id 的计算。

hashCode() 和 equals() 都基于 CountedString 的这两个字段来生成结果；如果它们只基于 String 或者只基于 id，不同的对象就可能产生相同的值。

在 main）中，使用相同的 String 创建了多个 CountedString 对象。这说明，虽然 String 相同，但是由于 id 不同，所以使得它们的散列码并不相同。在程序中，HashMap 被打印了出来，因此可以看到它内部是如何存储元素的（以无法辨别的次序），然后单独查询每一个键，以此证明查询机制工作正常。

作为第二个示例，请考虑 Individual 类，它被用作类型信息中所定义的 typeinfo.pet 类库的基类。Individual 类在那一章中就用到了，而它的定义则放到了本章，因此你可以正确地理解其实现。

在这里替换了手工去计算 `hashCode()`，我们使用了更合适的方式 `Objects.hash()`：

```java
// typeinfo/pets/Individual.java
package typeinfo.pets;
import java.util.*;
public class
Individual implements Comparable<Individual> {
    private static long counter = 0;
    private final long id = counter++;
    private String name;
    public Individual(String name) { this.name = name; }
    // 'name' is optional:
    public Individual() {}
    @Override
    public String toString() {
        return getClass().getSimpleName() +
                (name == null ? "" : " " + name);
    }
    public long id() { return id; }
    @Override
    public boolean equals(Object o) {
        return o instanceof Individual &&
                Objects.equals(id, ((Individual)o).id);
    }
    @Override
    public int hashCode() {
        return Objects.hash(name, id);
    }
    @Override
    public int compareTo(Individual arg) {
        // Compare by class name first:
        String first = getClass().getSimpleName();
        String argFirst = arg.getClass().getSimpleName();
        int firstCompare = first.compareTo(argFirst);
        if(firstCompare != 0)
            return firstCompare;
        if(name != null && arg.name != null) {
            int secondCompare = name.compareTo(arg.name);
            if(secondCompare != 0)
                return secondCompare;
        }
        return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
    }
}
```

compareTo() 方法有一个比较结构，因此它会产生一个排序序列，排序的规则首先按照实际类型排序，然后如果有名字的话，按照 name 排序，最后按照创建的顺序排序。下面的示例说明了它是如何工作的：

```java
// equalshashcode/IndividualTest.java
import collections.MapOfList;
```

```
import typeinfo.pets.*;
import java.util.*;
public class IndividualTest {
    public static void main(String[] args) {
        Set<Individual> pets = new TreeSet<>();
        for(List<? extends Pet> lp :
                MapOfList.petPeople.values())
            for(Pet p : lp)
                pets.add(p);
        pets.forEach(System.out::println);
    }
}
/* Output:
Cat Elsie May
Cat Pinkola
Cat Shackleton
Cat Stanford
Cymric Molly
Dog Margrett
Mutt Spot
Pug Louie aka Louis Snorkelstein Dupree
Rat Fizzy
Rat Freckly
Rat Fuzzy
*/
```

由于所有的宠物都有名字，因此它们首先按照类型排序，然后在同类型中按照名字排序。

8.3 调优 HashMap

我们有可能手动调优 HashMap 以提高其在特定应用程序中的性能。为了理解调整 HashMap 时的性能问题，一些术语是必要的：

- 容量（Capacity）：表中存储的桶数量。
- 初始容量（Initial Capacity）：当表被创建时，桶的初始个数。HashMap 和 HashSet 有可以让你指定初始容量的构造器。
- 个数（Size）：目前存储在表中的键值对的个数。
- 负载因子（Load factor）：通常表现为 $\frac{size}{capacity}$。当负载因子大小为 0 的时候表示为一个空表。当负载因子大小为 0.5 表示为一个半满表（half-full table），以此类推。轻负载的表几乎没有冲突，因此是插入和查找的最佳选择（但会减慢使用迭代器进行遍历的过程）。HashMap 和 HashSet 有可以让你指定负载因子的构造器。当表内容量达到了负载因子，集合就会自动扩充为原始容量（桶的数量）的两倍，并且会将原始的对象存储在新的桶集合中（也被称为 rehashing）

HashMap 中负载因子的大小为 0.75（当表内容量大小不足四分之三的时候，不会发生 rehashing 现象）。这看起来是一个非常好的同时考虑到时间和空间消耗的平衡策略。更高的负载因子会减少空间

的消耗，但是会增加查询的耗时。重要的是，查询操作是你使用的最频繁的一个操作（包括 `get()` 和 `put()` 方法）。

如果你知道存储在 HashMap 中确切的条目个数，直接创建一个足够容量大小的 HashMap，以避免自动发生的 rehashing 操作。

附录 9

集合主题

> 本附录是一些比集合中介绍的更高级的内容。

9.1 示例数据

这里创建一些样本数据用于集合示例。以下数据将颜色名称与 HTML 颜色的 RGB 值相关联。请注意，每个键和值都是唯一的：

```
// onjava/HTMLColors.java
// Sample data for collection examples
package onjava;
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;

public class HTMLColors {
  public static final Object[][] ARRAY = {
    { 0xF0F8FF, "AliceBlue" },
    { 0xFAEBD7, "AntiqueWhite" },
    { 0x7FFFD4, "Aquamarine" },
    { 0xF0FFFF, "Azure" },
    { 0xF5F5DC, "Beige" },
    { 0xFFE4C4, "Bisque" },
    { 0x000000, "Black" },
    { 0xFFEBCD, "BlanchedAlmond" },
    { 0x0000FF, "Blue" },
    { 0x8A2BE2, "BlueViolet" },
    { 0xA52A2A, "Brown" },
    { 0xDEB887, "BurlyWood" },
    { 0x5F9EA0, "CadetBlue" },
    { 0x7FFF00, "Chartreuse" },
    { 0xD2691E, "Chocolate" },
    { 0xFF7F50, "Coral" },
    { 0x6495ED, "CornflowerBlue" },
```

```
    { 0xFFF8DC, "Cornsilk" },
    { 0xDC143C, "Crimson" },
    { 0x00FFFF, "Cyan" },
    { 0x00008B, "DarkBlue" },
    { 0x008B8B, "DarkCyan" },
    { 0xB8860B, "DarkGoldenRod" },
    { 0xA9A9A9, "DarkGray" },
    { 0x006400, "DarkGreen" },
    { 0xBDB76B, "DarkKhaki" },
    { 0x8B008B, "DarkMagenta" },
    { 0x556B2F, "DarkOliveGreen" },
    { 0xFF8C00, "DarkOrange" },
    { 0x9932CC, "DarkOrchid" },
    { 0x8B0000, "DarkRed" },
    { 0xE9967A, "DarkSalmon" },
    { 0x8FBC8F, "DarkSeaGreen" },
    { 0x483D8B, "DarkSlateBlue" },
    { 0x2F4F4F, "DarkSlateGray" },
    { 0x00CED1, "DarkTurquoise" },
    { 0x9400D3, "DarkViolet" },
    { 0xFF1493, "DeepPink" },
    { 0x00BFFF, "DeepSkyBlue" },
    { 0x696969, "DimGray" },
    { 0x1E90FF, "DodgerBlue" },
    { 0xB22222, "FireBrick" },
    { 0xFFFAF0, "FloralWhite" },
    { 0x228B22, "ForestGreen" },
    { 0xDCDCDC, "Gainsboro" },
    { 0xF8F8FF, "GhostWhite" },
    { 0xFFD700, "Gold" },
    { 0xDAA520, "GoldenRod" },
    { 0x808080, "Gray" },
    { 0x008000, "Green" },
    { 0xADFF2F, "GreenYellow" },
    { 0xF0FFF0, "HoneyDew" },
    { 0xFF69B4, "HotPink" },
    { 0xCD5C5C, "IndianRed" },
    { 0x4B0082, "Indigo" },
    { 0xFFFFF0, "Ivory" },
    { 0xF0E68C, "Khaki" },
    { 0xE6E6FA, "Lavender" },
    { 0xFFF0F5, "LavenderBlush" },
    { 0x7CFC00, "LawnGreen" },
    { 0xFFFACD, "LemonChiffon" },
    { 0xADD8E6, "LightBlue" },
    { 0xF08080, "LightCoral" },
    { 0xE0FFFF, "LightCyan" },
    { 0xFAFAD2, "LightGoldenRodYellow" },
    { 0xD3D3D3, "LightGray" },
    { 0x90EE90, "LightGreen" },
    { 0xFFB6C1, "LightPink" },
```

```
    { 0xFFA07A, "LightSalmon" },
    { 0x20B2AA, "LightSeaGreen" },
    { 0x87CEFA, "LightSkyBlue" },
    { 0x778899, "LightSlateGray" },
    { 0xB0C4DE, "LightSteelBlue" },
    { 0xFFFFE0, "LightYellow" },
    { 0x00FF00, "Lime" },
    { 0x32CD32, "LimeGreen" },
    { 0xFAF0E6, "Linen" },
    { 0xFF00FF, "Magenta" },
    { 0x800000, "Maroon" },
    { 0x66CDAA, "MediumAquaMarine" },
    { 0x0000CD, "MediumBlue" },
    { 0xBA55D3, "MediumOrchid" },
    { 0x9370DB, "MediumPurple" },
    { 0x3CB371, "MediumSeaGreen" },
    { 0x7B68EE, "MediumSlateBlue" },
    { 0x00FA9A, "MediumSpringGreen" },
    { 0x48D1CC, "MediumTurquoise" },
    { 0xC71585, "MediumVioletRed" },
    { 0x191970, "MidnightBlue" },
    { 0xF5FFFA, "MintCream" },
    { 0xFFE4E1, "MistyRose" },
    { 0xFFE4B5, "Moccasin" },
    { 0xFFDEAD, "NavajoWhite" },
    { 0x000080, "Navy" },
    { 0xFDF5E6, "OldLace" },
    { 0x808000, "Olive" },
    { 0x6B8E23, "OliveDrab" },
    { 0xFFA500, "Orange" },
    { 0xFF4500, "OrangeRed" },
    { 0xDA70D6, "Orchid" },
    { 0xEEE8AA, "PaleGoldenRod" },
    { 0x98FB98, "PaleGreen" },
    { 0xAFEEEE, "PaleTurquoise" },
    { 0xDB7093, "PaleVioletRed" },
    { 0xFFEFD5, "PapayaWhip" },
    { 0xFFDAB9, "PeachPuff" },
    { 0xCD853F, "Peru" },
    { 0xFFC0CB, "Pink" },
    { 0xDDA0DD, "Plum" },
    { 0xB0E0E6, "PowderBlue" },
    { 0x800080, "Purple" },
    { 0xFF0000, "Red" },
    { 0xBC8F8F, "RosyBrown" },
    { 0x4169E1, "RoyalBlue" },
    { 0x8B4513, "SaddleBrown" },
    { 0xFA8072, "Salmon" },
    { 0xF4A460, "SandyBrown" },
    { 0x2E8B57, "SeaGreen" },
    { 0xFFF5EE, "SeaShell" },
```

```
    { 0xA0522D, "Sienna" },
    { 0xC0C0C0, "Silver" },
    { 0x87CEEB, "SkyBlue" },
    { 0x6A5ACD, "SlateBlue" },
    { 0x708090, "SlateGray" },
    { 0xFFFAFA, "Snow" },
    { 0x00FF7F, "SpringGreen" },
    { 0x4682B4, "SteelBlue" },
    { 0xD2B48C, "Tan" },
    { 0x008080, "Teal" },
    { 0xD8BFD8, "Thistle" },
    { 0xFF6347, "Tomato" },
    { 0x40E0D0, "Turquoise" },
    { 0xEE82EE, "Violet" },
    { 0xF5DEB3, "Wheat" },
    { 0xFFFFFF, "White" },
    { 0xF5F5F5, "WhiteSmoke" },
    { 0xFFFF00, "Yellow" },
    { 0x9ACD32, "YellowGreen" },
  };
  public static final Map<Integer,String> MAP =
    Arrays.stream(ARRAY)
      .collect(Collectors.toMap(
        element -> (Integer)element[0],
        element -> (String)element[1],
        (v1, v2) -> { // Merge function
          throw new IllegalStateException();
        },
        LinkedHashMap::new
      ));
  // Inversion only works if values are unique:
  public static <V, K> Map<V, K>
  invert(Map<K, V> map) {
    return map.entrySet().stream()
      .collect(Collectors.toMap(
        Map.Entry::getValue,
        Map.Entry::getKey,
        (v1, v2) -> {
          throw new IllegalStateException();
        },
        LinkedHashMap::new
      ));
  }
  public static final Map<String,Integer>
    INVMAP = invert(MAP);
  // Look up RGB value given a name:
  public static Integer rgb(String colorName) {
    return INVMAP.get(colorName);
  }
  public static final List<String> LIST =
    Arrays.stream(ARRAY)
```

```
      .map(item -> (String)item[1])
      .collect(Collectors.toList());
  public static final List<Integer> RGBLIST =
    Arrays.stream(ARRAY)
      .map(item -> (Integer)item[0])
      .collect(Collectors.toList());
  public static
  void show(Map.Entry<Integer,String> e) {
    System.out.format(
      "0x%06X: %s%n", e.getKey(), e.getValue());
  }
  public static void
  show(Map<Integer,String> m, int count) {
    m.entrySet().stream()
      .limit(count)
      .forEach(e -> show(e));
  }
  public static void show(Map<Integer,String> m) {
    show(m, m.size());
  }
  public static
  void show(Collection<String> lst, int count) {
    lst.stream()
      .limit(count)
      .forEach(System.out::println);
  }
  public static void show(Collection<String> lst) {
    show(lst, lst.size());
  }
  public static
  void showrgb(Collection<Integer> lst, int count) {
    lst.stream()
      .limit(count)
      .forEach(n -> System.out.format("0x%06X%n", n));
  }
  public static void showrgb(Collection<Integer> lst) {
    showrgb(lst, lst.size());
  }
  public static
  void showInv(Map<String,Integer> m, int count) {
    m.entrySet().stream()
      .limit(count)
      .forEach(e ->
        System.out.format(
          "%-20s  0x%06X%n", e.getKey(), e.getValue()));
  }
  public static void showInv(Map<String,Integer> m) {
    showInv(m, m.size());
  }
  public static void border() {
    System.out.println(
```

```
        "******************************");
    }
}
```

MAP 是使用 Streams（流式编程）创建的。二维数组 **ARRAY** 作为流传输到 **Map** 中，但请注意我们不仅仅是使用简单版本的 `Collectors.toMap()` 。那个版本生成一个 **HashMap** ，它使用散列函数来控制对键的排序。为了保留原来的顺序，我们必须将键值对直接放入 **TreeMap** 中，这意味着我们需要使用更复杂的 `Collectors.toMap()` 版本。这需要两个函数从每个流元素中提取键和值，就像简单版本的`Collectors.toMap()` 一样。然后它需要一个*合并函数*（merge function），它解决了与同一个键相关的两个值之间的冲突。这里的数据已经预先审查过，因此绝不会发生这种情况，如果有的话，这里会抛出异常。最后，传递生成所需类型的空 map 的函数，然后用流来填充它。

`rgb()` 方法是一个便捷函数（convenience function），它接受颜色名称 **String** 参数并生成其数字 RGB 值。为此，我们需要一个反转版本的 **COLORS** ，它接受一个 **String** 键并查找 RGB 的 **Integer** 值。这是通过 `invert()` 方法实现的，如果任何 **COLORS** 值不唯一，则抛出异常。

我们还创建包含所有名称的 **LIST** ，以及包含十六进制表示法的 RGB 值的 **RGBLIST** 。

第一个 `show()` 方法接受一个 **Map.Entry** 并显示以十六进制表示的键，以便轻松地对原始 **ARRAY** 进行双重检查。名称以 **show** 开头的每个方法都会重载两个版本，其中一个版本采用 **count** 参数来指示要显示的元素数量，第二个版本显示序列中的所有元素。

这里是一个基本的测试:

```java
// collectiontopics/HTMLColorTest.java
import static onjava.HTMLColors.*;

public class HTMLColorTest {
  static final int DISPLAY_SIZE = 20;
  public static void main(String[] args) {
    show(MAP, DISPLAY_SIZE);
    border();
    showInv(INVMAP, DISPLAY_SIZE);
    border();
    show(LIST, DISPLAY_SIZE);
    border();
    showrgb(RGBLIST, DISPLAY_SIZE);
  }
}
/* Output:
0xF0F8FF: AliceBlue
0xFAEBD7: AntiqueWhite
0x7FFFD4: Aquamarine
0xF0FFFF: Azure
0xF5F5DC: Beige
0xFFE4C4: Bisque
0x000000: Black
0xFFEBCD: BlanchedAlmond
```

```
0x0000FF: Blue
0x8A2BE2: BlueViolet
0xA52A2A: Brown
0xDEB887: BurlyWood
0x5F9EA0: CadetBlue
0x7FFF00: Chartreuse
0xD2691E: Chocolate
0xFF7F50: Coral
0x6495ED: CornflowerBlue
0xFFF8DC: Cornsilk
0xDC143C: Crimson
0x00FFFF: Cyan
******************************
AliceBlue             0xF0F8FF
AntiqueWhite          0xFAEBD7
Aquamarine            0x7FFFD4
Azure                 0xF0FFFF
Beige                 0xF5F5DC
Bisque                0xFFE4C4
Black                 0x000000
BlanchedAlmond        0xFFEBCD
Blue                  0x0000FF
BlueViolet            0x8A2BE2
Brown                 0xA52A2A
BurlyWood             0xDEB887
CadetBlue             0x5F9EA0
Chartreuse            0x7FFF00
Chocolate             0xD2691E
Coral                 0xFF7F50
CornflowerBlue        0x6495ED
Cornsilk              0xFFF8DC
Crimson               0xDC143C
Cyan                  0x00FFFF
******************************
AliceBlue
AntiqueWhite
Aquamarine
Azure
Beige
Bisque
Black
BlanchedAlmond
Blue
BlueViolet
Brown
BurlyWood
CadetBlue
Chartreuse
Chocolate
Coral
CornflowerBlue
```

```
Cornsilk
Crimson
Cyan
*****************************
0xF0F8FF
0xFAEBD7
0x7FFFD4
0xF0FFFF
0xF5F5DC
0xFFE4C4
0x000000
0xFFEBCD
0x0000FF
0x8A2BE2
0xA52A2A
0xDEB887
0x5F9EA0
0x7FFF00
0xD2691E
0xFF7F50
0x6495ED
0xFFF8DC
0xDC143C
0x00FFFF
*/
```

可以看到，使用 **LinkedHashMap** 确实能够保留 **HTMLColors.ARRAY** 的顺序。

9.2 List 行为

Lists 是存储和检索对象（次于数组）的最基本方法。基本列表操作包括：

- `add()` 用于插入元素
- `get()` 用于随机访问元素
- `iterator()` 获取序列上的一个 **Iterator**
- `stream()` 生成元素的一个 **Stream**

列表构造方法始终保留元素的添加顺序。

以下示例中的方法各自涵盖了一组不同的行为：每个 **List** 可以执行的操作（`basicTest()`），使用 **Iterator**（`iterMotion()`）遍历序列，使用 **Iterator**（`iterManipulation()`）更改内容，查看 **List** 操作（`testVisual()`）的效果，以及仅可用于 **LinkedLists** 的操作：

```
// collectiontopics/ListOps.java
// Things you can do with Lists
import java.util.*;
import onjava.HTMLColors;

```

```
public class ListOps {
  // Create a short list for testing:
  static final List<String> LIST =
    HTMLColors.LIST.subList(0, 10);
  private static boolean b;
  private static String s;
  private static int i;
  private static Iterator<String> it;
  private static ListIterator<String> lit;
  public static void basicTest(List<String> a) {
    a.add(1, "x"); // Add at location 1
    a.add("x"); // Add at end
    // Add a collection:
    a.addAll(LIST);
    // Add a collection starting at location 3:
    a.addAll(3, LIST);
    b = a.contains("1"); // Is it in there?
    // Is the entire collection in there?
    b = a.containsAll(LIST);
    // Lists allow random access, which is cheap
    // for ArrayList, expensive for LinkedList:
    s = a.get(1); // Get (typed) object at location 1
    i = a.indexOf("1"); // Tell index of object
    b = a.isEmpty(); // Any elements inside?
    it = a.iterator(); // Ordinary Iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Start at location 3
    i = a.lastIndexOf("1"); // Last match
    a.remove(1); // Remove location 1
    a.remove("3"); // Remove this object
    a.set(1, "y"); // Set location 1 to "y"
    // Keep everything that's in the argument
    // (the intersection of the two sets):
    a.retainAll(LIST);
    // Remove everything that's in the argument:
    a.removeAll(LIST);
    i = a.size(); // How big is it?
    a.clear(); // Remove all elements
  }
  public static void iterMotion(List<String> a) {
    ListIterator<String> it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    s = it.next();
    i = it.nextIndex();
    s = it.previous();
    i = it.previousIndex();
  }
  public static void iterManipulation(List<String> a) {
    ListIterator<String> it = a.listIterator();
    it.add("47");
```

```
    // Must move to an element after add():
    it.next();
    // Remove the element after the new one:
    it.remove();
    // Must move to an element after remove():
    it.next();
    // Change the element after the deleted one:
    it.set("47");
  }
  public static void testVisual(List<String> a) {
    System.out.println(a);
    List<String> b = LIST;
    System.out.println("b = " + b);
    a.addAll(b);
    a.addAll(b);
    System.out.println(a);
    // Insert, remove, and replace elements
    // using a ListIterator:
    ListIterator<String> x =
      a.listIterator(a.size()/2);
    x.add("one");
    System.out.println(a);
    System.out.println(x.next());
    x.remove();
    System.out.println(x.next());
    x.set("47");
    System.out.println(a);
    // Traverse the list backwards:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      System.out.print(x.previous() + " ");
    System.out.println();
    System.out.println("testVisual finished");
  }
  // There are some things that only LinkedLists can do:
  public static void testLinkedList() {
    LinkedList<String> ll = new LinkedList<>();
    ll.addAll(LIST);
    System.out.println(ll);
    // Treat it like a stack, pushing:
    ll.addFirst("one");
    ll.addFirst("two");
    System.out.println(ll);
    // Like "peeking" at the top of a stack:
    System.out.println(ll.getFirst());
    // Like popping a stack:
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Treat it like a queue, pulling elements
    // off the tail end:
    System.out.println(ll.removeLast());
```

```
    System.out.println(ll);
  }
  public static void main(String[] args) {
    // Make and fill a new list each time:
    basicTest(new LinkedList<>(LIST));
    basicTest(new ArrayList<>(LIST));
    iterMotion(new LinkedList<>(LIST));
    iterMotion(new ArrayList<>(LIST));
    iterManipulation(new LinkedList<>(LIST));
    iterManipulation(new ArrayList<>(LIST));
    testVisual(new LinkedList<>(LIST));
    testLinkedList();
  }
}
/* Output:
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
b = [AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige, one,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
Bisque
Black
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige, one,
47, BlanchedAlmond, Blue, BlueViolet, AliceBlue,
AntiqueWhite, Aquamarine, Azure, Beige, Bisque, Black,
BlanchedAlmond, Blue, BlueViolet]
BlueViolet Blue BlanchedAlmond Black Bisque Beige Azure
Aquamarine AntiqueWhite AliceBlue BlueViolet Blue
BlanchedAlmond 47 one Beige Azure Aquamarine
AntiqueWhite AliceBlue BlueViolet Blue BlanchedAlmond
Black Bisque Beige Azure Aquamarine AntiqueWhite
AliceBlue
testVisual finished
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
[two, one, AliceBlue, AntiqueWhite, Aquamarine, Azure,
Beige, Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
two
```

```
two
one
BlueViolet
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue]
*/
```

在 basicTest() 和 iterMotion() 中，方法调用是为了展示正确的语法，尽管获取了返回值，但不会使用它。在某些情况下，根本不会去获取返回值。在使用这些方法之前，请查看 JDK 文档中这些方法的完整用法。

9.3 Set 行为

Set 的主要用处是测试成员身份，不过也可以将其用作删除重复元素的工具。如果不关心元素顺序或并发性，**HashSet** 总是最好的选择，因为它是专门为了快速查找而设计的（这里使用了在附录：理解 equals 和 hashCode 方法章节中探讨的散列函数）。

其它的 **Set** 实现产生不同的排序行为：

```java
// collectiontopics/SetOrder.java
import java.util.*;
import onjava.HTMLColors;

public class SetOrder {
  static String[] sets = {
    "java.util.HashSet",
    "java.util.TreeSet",
    "java.util.concurrent.ConcurrentSkipListSet",
    "java.util.LinkedHashSet",
    "java.util.concurrent.CopyOnWriteArraySet",
  };
  static final List<String> RLIST =
    new ArrayList<>(HTMLColors.LIST);
  static {
    Collections.reverse(RLIST);
  }
  public static void
  main(String[] args) throws Exception {
    for(String type: sets) {
      System.out.format("[-> %s <-]%n",
        type.substring(type.lastIndexOf('.') + 1));
      @SuppressWarnings("unchecked")
      Set<String> set = (Set<String>)
        Class.forName(type).newInstance();
      set.addAll(RLIST);
      set.stream()
        .limit(10)
```

```
29          .forEach(System.out::println);
30      }
31    }
32  }
33  /* Output:
34  [-> HashSet <-]
35  MediumOrchid
36  PaleGoldenRod
37  Sienna
38  LightSlateGray
39  DarkSeaGreen
40  Black
41  Gainsboro
42  Orange
43  LightCoral
44  DodgerBlue
45  [-> TreeSet <-]
46  AliceBlue
47  AntiqueWhite
48  Aquamarine
49  Azure
50  Beige
51  Bisque
52  Black
53  BlanchedAlmond
54  Blue
55  BlueViolet
56  [-> ConcurrentSkipListSet <-]
57  AliceBlue
58  AntiqueWhite
59  Aquamarine
60  Azure
61  Beige
62  Bisque
63  Black
64  BlanchedAlmond
65  Blue
66  BlueViolet
67  [-> LinkedHashSet <-]
68  YellowGreen
69  Yellow
70  WhiteSmoke
71  White
72  Wheat
73  Violet
74  Turquoise
75  Tomato
76  Thistle
77  Teal
78  [-> CopyOnWriteArraySet <-]
79  YellowGreen
```

```
Yellow
WhiteSmoke
White
Wheat
Violet
Turquoise
Tomato
Thistle
Teal
*/
```

这里需要使用 **@SuppressWarnings("unchecked")**，因为这里将一个 **String**（可能是任何东西）传递给了 `Class.forName(type).newInstance()`。编译器并不能保证这是一次成功的操作。

RLIST 是 **HTMLColors.LIST** 的反转版本。因为 `Collections.reverse()` 是通过修改参数来执行反向操作，而不是返回包含反向元素的新 **List**，所以该调用在 **static** 块内执行。**RLIST** 可以防止我们意外地认为 **Set** 对其结果进行了排序。

HashSet 的输出结果似乎没有可辨别的顺序，因为它是基于散列函数的。**TreeSet** 和 **ConcurrentSkipListSet** 都对它们的元素进行了排序，它们都实现了 **SortedSet** 接口来标识这个特点。因为实现该接口的 **Set** 按顺序排列，所以该接口还有一些其他的可用操作。**LinkedHashSet** 和 **CopyOnWriteArraySet** 尽管没有用于标识的接口，但它们还是保留了元素的插入顺序。

ConcurrentSkipListSet 和 **CopyOnWriteArraySet** 是线程安全的。

在附录的最后，我们将了解在非 **HashSet** 实现的 **Set** 上添加额外排序的性能成本，以及不同实现中的任何其他功能的成本。

9.4 在 Map 中使用函数式操作

与 **Collection** 接口一样，`forEach()` 也内置在 **Map** 接口中。但是如果想要执行任何其他的基本功能操作，比如 `map()`，`flatMap()`，`reduce()` 或 `filter()` 时，该怎么办？查看 **Map** 接口发现并没有这些。

可以通过 `entrySet()` 连接到这些方法，该方法会生成一个由 **Map.Entry** 对象组成的 **Set**。这个 **Set** 包含 `stream()` 和 `parallelStream()` 方法。只需要记住一件事，这里正在使用的是 **Map.Entry** 对象：

```
// collectiontopics/FunctionalMap.java
// Functional operations on a Map
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import static onjava.HTMLColors.*;

```

```
public class FunctionalMap {
  public static void main(String[] args) {
    MAP.entrySet().stream()
      .map(Map.Entry::getValue)
      .filter(v -> v.startsWith("Dark"))
      .map(v -> v.replaceFirst("Dark", "Hot"))
      .forEach(System.out::println);
  }
}
/* Output:
HotBlue
HotCyan
HotGoldenRod
HotGray
HotGreen
HotKhaki
HotMagenta
HotOliveGreen
HotOrange
HotOrchid
HotRed
HotSalmon
HotSeaGreen
HotSlateBlue
HotSlateGray
HotTurquoise
HotViolet
*/
```

生成 **Stream** 后，所有的基本功能方法，甚至更多就都可以使用了。

9.5 选择 Map 片段

由 **TreeMap** 和 **ConcurrentSkipListMap** 实现的 **NavigableMap** 接口解决了需要选择 Map 片段的问题。下面是一个示例，使用了 **HTMLColors**：

```
// collectiontopics/NavMap.java
// NavigableMap produces pieces of a Map
import java.util.*;
import java.util.concurrent.*;
import static onjava.HTMLColors.*;

public class NavMap {
  public static final
  NavigableMap<Integer,String> COLORS =
    new ConcurrentSkipListMap<>(MAP);
  public static void main(String[] args) {
    show(COLORS.firstEntry());
```

```
    border();
    show(COLORS.lastEntry());
    border();
    NavigableMap<Integer, String> toLime =
      COLORS.headMap(rgb("Lime"), true);
    show(toLime);
    border();
    show(COLORS.ceilingEntry(rgb("DeepSkyBlue") - 1));
    border();
    show(COLORS.floorEntry(rgb("DeepSkyBlue") - 1));
    border();
    show(toLime.descendingMap());
    border();
    show(COLORS.tailMap(rgb("MistyRose"), true));
    border();
    show(COLORS.subMap(
      rgb("Orchid"), true,
      rgb("DarkSalmon"), false));
  }
}
/* Output:
0x000000: Black
******************************
0xFFFFFF: White
******************************
0x000000: Black
0x000080: Navy
0x00008B: DarkBlue
0x0000CD: MediumBlue
0x0000FF: Blue
0x006400: DarkGreen
0x008000: Green
0x008080: Teal
0x008B8B: DarkCyan
0x00BFFF: DeepSkyBlue
0x00CED1: DarkTurquoise
0x00FA9A: MediumSpringGreen
0x00FF00: Lime
******************************
0x00BFFF: DeepSkyBlue
******************************
0x008B8B: DarkCyan
******************************
0x00FF00: Lime
0x00FA9A: MediumSpringGreen
0x00CED1: DarkTurquoise
0x00BFFF: DeepSkyBlue
0x008B8B: DarkCyan
0x008080: Teal
0x008000: Green
0x006400: DarkGreen
```

```
0x0000FF: Blue
0x0000CD: MediumBlue
0x00008B: DarkBlue
0x000080: Navy
0x000000: Black
******************************
0xFFE4E1: MistyRose
0xFFEBCD: BlanchedAlmond
0xFFEFD5: PapayaWhip
0xFFF0F5: LavenderBlush
0xFFF5EE: SeaShell
0xFFF8DC: Cornsilk
0xFFFACD: LemonChiffon
0xFFFAF0: FloralWhite
0xFFFAFA: Snow
0xFFFF00: Yellow
0xFFFFE0: LightYellow
0xFFFFF0: Ivory
0xFFFFFF: White
******************************
0xDA70D6: Orchid
0xDAA520: GoldenRod
0xDB7093: PaleVioletRed
0xDC143C: Crimson
0xDCDCDC: Gainsboro
0xDDA0DD: Plum
0xDEB887: BurlyWood
0xE0FFFF: LightCyan
0xE6E6FA: Lavender
*/
```

在主方法中可以看到 **NavigableMap** 的各种功能。因为 **NavigableMap** 具有键顺序，所以它使用了 `firstEntry()` 和 `lastEntry()` 的概念。调用 `headMap()` 会生成一个 **NavigableMap**，其中包含了从 **Map** 的开头到 `headMap()` 参数中所指向的一组元素，其中 **boolean** 值指示结果中是否包含该参数。调用 `tailMap()` 执行了类似的操作，只不过是从参数开始到 **Map** 的末尾。`subMap()` 则允许生成 **Map** 中间的一部分。

`ceilingEntry()` 从当前键值对向上搜索下一个键值对，`floorEntry()` 则是向下搜索。`descendingMap()` 反转了 **NavigableMap** 的顺序。

如果需要通过分割 **Map** 来简化所正在解决的问题，则 **NavigableMap** 可以做到。具有类似的功能的其它集合实现也可以用来帮助解决问题。

9.6 填充集合

与 **Arrays** 一样，这里有一个名为 **Collections** 的伴随类（companion class），包含了一些 **static** 的实用方法，其中包括一个名为 `fill()` 的方法。`fill()` 只复制整个集合中的单个对象引用。此外，它仅适用于 **List** 对象，但结果列表可以传递给构造方法或 `addAll()` 方法：

```
// collectiontopics/FillingLists.java
// Collections.fill() & Collections.nCopies()
import java.util.*;

class StringAddress {
  private String s;
  StringAddress(String s) { this.s = s; }
  @Override
  public String toString() {
    return super.toString() + " " + s;
  }
}

public class FillingLists {
  public static void main(String[] args) {
    List<StringAddress> list = new ArrayList<>(
      Collections.nCopies(4,
        new StringAddress("Hello")));
    System.out.println(list);
    Collections.fill(list,
      new StringAddress("World!"));
    System.out.println(list);
  }
}
/* Output:
[StringAddress@15db9742 Hello, StringAddress@15db9742
Hello, StringAddress@15db9742 Hello,
StringAddress@15db9742 Hello]
[StringAddress@6d06d69c World!, StringAddress@6d06d69c
World!, StringAddress@6d06d69c World!,
StringAddress@6d06d69c World!]
*/
```

这个示例展示了两种使用对单个对象的引用来填充 **Collection** 的方法。第一个：`Collections.nCopies()`，创建一个 **List**，并传递给 **ArrayList** 的构造方法，进而填充了 **ArrayList**。

StringAddress 中的 `toString()` 方法调用了 `Object.toString()`，它先生成类名，后跟着对象的哈希码的无符号十六进制表示（哈希吗由 `hashCode()` 方法生成）。输出显示所有的引用都指向同一个对象。调用第二个方法 `Collections.fill()` 后也是如此。`fill()` 方法的用处非常有限，它只能替换 **List** 中已有的元素，而且不会添加新元素，

9.6.1 使用 Suppliers 填充集合

泛型章节中介绍的 **onjava.Suppliers** 类为填充集合提供了通用解决方案。这是一个使用 **Suppliers** 初始化几种不同类型的 **Collection** 的示例:

```
// collectiontopics/SuppliersCollectionTest.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import onjava.*;

class Government implements Supplier<String> {
  static String[] foundation = (
    "strange women lying in ponds " +
    "distributing swords is no basis " +
    "for a system of government").split(" ");
  private int index;
  @Override
  public String get() {
    return foundation[index++];
  }
}

public class SuppliersCollectionTest {
  public static void main(String[] args) {
    // Suppliers class from the Generics chapter:
    Set<String> set = Suppliers.create(
      LinkedHashSet::new, new Government(), 15);
    System.out.println(set);
    List<String> list = Suppliers.create(
      LinkedList::new, new Government(), 15);
    System.out.println(list);
    list = new ArrayList<>();
    Suppliers.fill(list, new Government(), 15);
    System.out.println(list);

    // Or we can use Streams:
    set = Arrays.stream(Government.foundation)
      .collect(Collectors.toSet());
    System.out.println(set);
    list = Arrays.stream(Government.foundation)
      .collect(Collectors.toList());
    System.out.println(list);
    list = Arrays.stream(Government.foundation)
      .collect(Collectors
        .toCollection(LinkedList::new));
    System.out.println(list);
    set = Arrays.stream(Government.foundation)
      .collect(Collectors
        .toCollection(LinkedHashSet::new));
```

```
    System.out.println(set);
  }
}
/* Output:
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[ponds, no, a, in, swords, for, is, basis, strange,
system, government, distributing, of, women, lying]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
*/
```

LinkedHashSet 中的的元素按插入顺序排列，因为它维护一个链表来保存该顺序。

但是请注意示例的第二部分：大多数情况下都可以使用 **Stream** 来创建和填充 **Collection** 。在本例中的 **Stream** 版本不需要声明 **Supplier** 所想要创建的元素数量;，它直接吸收了 **Stream** 中的所有元素。

尽可能优先选择 **Stream** 来解决问题。

9.6.2 Map Suppliers

使用 **Supplier** 来填充 **Map** 时需要一个 **Pair** 类，因为每次调用一个 **Supplier** 的 `get()` 方法时，都必须生成一对对象（一个键和一个值）：

```
// onjava/Pair.java
package onjava;

public class Pair<K, V> {
  public final K key;
  public final V value;
  public Pair(K k, V v) {
    key = k;
    value = v;
  }
  public K key() { return key; }
  public V value() { return value; }
  public static <K,V> Pair<K, V> make(K k, V v) {
    return new Pair<K,V>(k, v);
  }
}
```

Pair 是一个只读的 *数据传输对象*（Data Transfer Object）或 *信使*（Messenger）。这与泛型章节中的 **Tuple2** 基本相同，但名字更适合 **Map** 初始化。我还添加了静态的 `make()` 方法，以便为创建 **Pair** 对象提供一个更简洁的名字。

Java 8 的 **Stream** 提供了填充 **Map** 的便捷方法:

```
// collectiontopics/StreamFillMaps.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import onjava.*;

class Letters
implements Supplier<Pair<Integer,String>> {
  private int number = 1;
  private char letter = 'A';
  @Override
  public Pair<Integer,String> get() {
    return new Pair<>(number++, "" + letter++);
  }
}

public class StreamFillMaps {
  public static void main(String[] args) {
    Map<Integer,String> m =
      Stream.generate(new Letters())
      .limit(11)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
    System.out.println(m);

    // Two separate Suppliers:
    Rand.String rs = new Rand.String(3);
    Count.Character cc = new Count.Character();
    Map<Character,String> mcs = Stream.generate(
      () -> Pair.make(cc.get(), rs.get()))
      .limit(8)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
    System.out.println(mcs);

    // A key Supplier and a single value:
    Map<Character,String> mcs2 = Stream.generate(
      () -> Pair.make(cc.get(), "Val"))
      .limit(8)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
    System.out.println(mcs2);
  }
}
```

```
/* Output:
{1=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J,
11=K}
{b=btp, c=enp, d=ccu, e=xsz, f=gvg, g=mei, h=nne,
i=elo}
{p=Val, q=Val, j=Val, k=Val, l=Val, m=Val, n=Val,
o=Val}
*/
```

上面的示例中出现了一个模式，可以使用它来创建一个自动创建和填充 **Map** 的工具：

```
// onjava/FillMap.java
package onjava;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class FillMap {
  public static <K, V> Map<K,V>
  basic(Supplier<Pair<K,V>> pairGen, int size) {
    return Stream.generate(pairGen)
      .limit(size)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
  }
  public static <K, V> Map<K,V>
  basic(Supplier<K> keyGen,
        Supplier<V> valueGen, int size) {
    return Stream.generate(
      () -> Pair.make(keyGen.get(), valueGen.get()))
      .limit(size)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
  }
  public static <K, V, M extends Map<K,V>>
  M create(Supplier<K> keyGen,
           Supplier<V> valueGen,
           Supplier<M> mapSupplier, int size) {
    return Stream.generate( () ->
      Pair.make(keyGen.get(), valueGen.get()))
        .limit(size)
        .collect(Collectors
          .toMap(Pair::key, Pair::value,
                 (k, v) -> k, mapSupplier));
  }
}
```

basic() 方法生成一个默认的 **Map** ，而 `create()` 方法允许指定一个确切的 **Map** 类型，并返回那个确切的类型。

下面是一个测试：

```java
// collectiontopics/FillMapTest.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import onjava.*;

public class FillMapTest {
  public static void main(String[] args) {
    Map<String,Integer> mcs = FillMap.basic(
      new Rand.String(4), new Count.Integer(), 7);
    System.out.println(mcs);
    HashMap<String,Integer> hashm =
      FillMap.create(new Rand.String(4),
        new Count.Integer(), HashMap::new, 7);
    System.out.println(hashm);
    LinkedHashMap<String,Integer> linkm =
      FillMap.create(new Rand.String(4),
        new Count.Integer(), LinkedHashMap::new, 7);
    System.out.println(linkm);
  }
}
/* Output:
{npcc=1, ztdv=6, gvgm=3, btpe=0, einn=4, eelo=5,
uxsz=2}
{npcc=1, ztdv=6, gvgm=3, btpe=0, einn=4, eelo=5,
uxsz=2}
{btpe=0, npcc=1, uxsz=2, gvgm=3, einn=4, eelo=5,
ztdv=6}
*/
```

9.7 使用享元（Flyweight）自定义 Collection 和 Map

本节介绍如何创建自定义 **Collection** 和 **Map** 实现。每个 **java.util** 中的集合都有自己的 **Abstract** 类，它提供了该集合的部分实现，因此只需要实现必要的方法来生成所需的集合。你将看到通过继承 **java.util.Abstract** 类来创建自定义 **Map** 和 **Collection** 是多么简单。例如，要创建一个只读的 **Set**，则可以从 **AbstractSet** 继承并实现 `iterator()` 和 `size()`。最后一个示例是生成测试数据的另一种方法。生成的集合通常是只读的，并且所提供的方法最少。

该解决方案还演示了 *享元*（Flyweight）设计模式。当普通解决方案需要太多对象时，或者当生成普通对象占用太多空间时，可以使用享元。享元设计模式将对象的一部分外部化（externalizes）。相比于把对象的所有内容都包含在对象中，这样做使得对象的部分或者全部可以在更有效的外部表中查找，或通过一些节省空间的其他计算生成。

下面是一个可以是任何大小的 **List**，并且（有效地）使用 **Integer** 数据进行预初始化。要从 **AbstractList** 创建只读 **List**，必须实现 `get()` 和 `size()`：

```java
// onjava/CountingIntegerList.java
// List of any length, containing sample data
// {java onjava.CountingIntegerList}
package onjava;
import java.util.*;

public class CountingIntegerList
extends AbstractList<Integer> {
  private int size;
  public CountingIntegerList() { size = 0; }
  public CountingIntegerList(int size) {
    this.size = size < 0 ? 0 : size;
  }
  @Override
  public Integer get(int index) {
    return index;
  }
  @Override
  public int size() { return size; }
  public static void main(String[] args) {
    List<Integer> cil =
      new CountingIntegerList(30);
    System.out.println(cil);
    System.out.println(cil.get(500));
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
500
*/
```

只有当想要限制 **List** 的长度时，**size** 值才是重要的，就像在主方法中那样。即使在这种情况下，`get()` 也会产生任何值。

这个类是享元模式的一个简洁的例子。当需要的时候，`get()` “计算”所需的值，因此没必要存储和初始化实际的底层 **List** 结构。

在大多数程序中，这里所保存的存储结构永远都不会改变。但是，它允许用非常大的 **index** 来调用 `List.get()`，而 **List** 并不需要填充到这么大。此外，还可以在程序中大量使用 **CountingIntegerLists** 而无需担心存储问题。实际上，享元的一个好处是它允许使用更好的抽象而不用担心资源。

可以使用享元设计模式来实现具有任何大小数据集的其他“初始化”自定义集合。下面是一个 **Map**，它为每一个 **Integer** 键产生唯一的值：

```java
// onjava/CountMap.java
// Unlimited-length Map containing sample data
// {java onjava.CountMap}
package onjava;
```

```java
import java.util.*;
import java.util.stream.*;

public class CountMap
extends AbstractMap<Integer,String> {
  private int size;
  private static char[] chars =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
  private static String value(int key) {
    return
      chars[key % chars.length] +
      Integer.toString(key / chars.length);
  }
  public CountMap(int size) {
    this.size = size < 0 ? 0 : size;
  }
  @Override
  public String get(Object key) {
    return value((Integer)key);
  }
  private static class Entry
  implements Map.Entry<Integer,String> {
    int index;
    Entry(int index) { this.index = index; }
    @Override
    public boolean equals(Object o) {
      return o instanceof Entry &&
        Objects.equals(index, ((Entry)o).index);
    }
    @Override
    public Integer getKey() { return index; }
    @Override
    public String getValue() {
      return value(index);
    }
    @Override
    public String setValue(String value) {
      throw new UnsupportedOperationException();
    }
    @Override
    public int hashCode() {
      return Objects.hashCode(index);
    }
  }
  @Override
  public Set<Map.Entry<Integer,String>> entrySet() {
    // LinkedHashSet retains initialization order:
    return IntStream.range(0, size)
      .mapToObj(Entry::new)
      .collect(Collectors
        .toCollection(LinkedHashSet::new));
```

```
  }
  public static void main(String[] args) {
    final int size = 6;
    CountMap cm = new CountMap(60);
    System.out.println(cm);
    System.out.println(cm.get(500));
    cm.values().stream()
      .limit(size)
      .forEach(System.out::println);
    System.out.println();
    new Random(47).ints(size, 0, 1000)
      .mapToObj(cm::get)
      .forEach(System.out::println);
  }
}
/* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0, 10=K0, 11=L0, 12=M0, 13=N0, 14=O0, 15=P0, 16=Q0,
17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0,
25=Z0, 26=A1, 27=B1, 28=C1, 29=D1, 30=E1, 31=F1, 32=G1,
33=H1, 34=I1, 35=J1, 36=K1, 37=L1, 38=M1, 39=N1, 40=O1,
41=P1, 42=Q1, 43=R1, 44=S1, 45=T1, 46=U1, 47=V1, 48=W1,
49=X1, 50=Y1, 51=Z1, 52=A2, 53=B2, 54=C2, 55=D2, 56=E2,
57=F2, 58=G2, 59=H2}
G19
A0
B0
C0
D0
E0
F0

Y9
J21
R26
D33
Z36
N16
*/
```

要创建一个只读的 **Map** ，则从 **AbstractMap** 继承并实现 `entrySet()` 。私有的 `value()` 方法计算任何键的值，并在 `get()` 和 `Entry.getValue()` 中使用。可以忽略 **CountMap** 的大小。

这里是使用了 **LinkedHashSet** 而不是创建自定义 **Set** 类，因此并未完全实现享元。只有在调用 `entrySet()` 时才会生成此对象。

现在创建一个更复杂的享元。这个示例中的数据集是世界各国及其首都的 **Map** 。`capitals()` 方法生成一个国家和首都的 **Map** 。`names()` 方法生成一个由国家名字组成的 **List** 。当给定了表示所需大小的 **int** 参数时，两种方法都生成对应大小的列表片段：

```java
// onjava/Countries.java
// "Flyweight" Maps and Lists of sample data
// {java onjava.Countries}
package onjava;
import java.util.*;

public class Countries {
  public static final String[][] DATA = {
    // Africa
    {"ALGERIA","Algiers"},
    {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"},
    {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"},
    {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"},
    {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"},
    {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"},
    {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"},
    {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"},
    {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"},
    {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"},
    {"GUINEA","Conakry"},
    {"BISSAU","Bissau"},
    {"COTE D'IVOIR (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"},
    {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"},
    {"LIBYA","Tripoli"},
    {"MADAGASCAR","Antananarivo"},
    {"MALAWI","Lilongwe"},
    {"MALI","Bamako"},
    {"MAURITANIA","Nouakchott"},
    {"MAURITIUS","Port Louis"},
    {"MOROCCO","Rabat"},
    {"MOZAMBIQUE","Maputo"},
    {"NAMIBIA","Windhoek"},
    {"NIGER","Niamey"},
    {"NIGERIA","Abuja"},
    {"RWANDA","Kigali"},
    {"SAO TOME E PRINCIPE","Sao Tome"},
    {"SENEGAL","Dakar"},
    {"SEYCHELLES","Victoria"},
    {"SIERRA LEONE","Freetown"},
```

```
    {"SOMALIA","Mogadishu"},
    {"SOUTH AFRICA","Pretoria/Cape Town"},
    {"SUDAN","Khartoum"},
    {"SWAZILAND","Mbabane"},
    {"TANZANIA","Dodoma"},
    {"TOGO","Lome"},
    {"TUNISIA","Tunis"},
    {"UGANDA","Kampala"},
    {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",
     "Kinshasa"},
    {"ZAMBIA","Lusaka"},
    {"ZIMBABWE","Harare"},
    // Asia
    {"AFGHANISTAN","Kabul"},
    {"BAHRAIN","Manama"},
    {"BANGLADESH","Dhaka"},
    {"BHUTAN","Thimphu"},
    {"BRUNEI","Bandar Seri Begawan"},
    {"CAMBODIA","Phnom Penh"},
    {"CHINA","Beijing"},
    {"CYPRUS","Nicosia"},
    {"INDIA","New Delhi"},
    {"INDONESIA","Jakarta"},
    {"IRAN","Tehran"},
    {"IRAQ","Baghdad"},
    {"ISRAEL","Jerusalem"},
    {"JAPAN","Tokyo"},
    {"JORDAN","Amman"},
    {"KUWAIT","Kuwait City"},
    {"LAOS","Vientiane"},
    {"LEBANON","Beirut"},
    {"MALAYSIA","Kuala Lumpur"},
    {"THE MALDIVES","Male"},
    {"MONGOLIA","Ulan Bator"},
    {"MYANMAR (BURMA)","Rangoon"},
    {"NEPAL","Katmandu"},
    {"NORTH KOREA","P'yongyang"},
    {"OMAN","Muscat"},
    {"PAKISTAN","Islamabad"},
    {"PHILIPPINES","Manila"},
    {"QATAR","Doha"},
    {"SAUDI ARABIA","Riyadh"},
    {"SINGAPORE","Singapore"},
    {"SOUTH KOREA","Seoul"},
    {"SRI LANKA","Colombo"},
    {"SYRIA","Damascus"},
    {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
    {"THAILAND","Bangkok"},
    {"TURKEY","Ankara"},
    {"UNITED ARAB EMIRATES","Abu Dhabi"},
    {"VIETNAM","Hanoi"},
```

```
    {"YEMEN","Sana'a"},
    // Australia and Oceania
    {"AUSTRALIA","Canberra"},
    {"FIJI","Suva"},
    {"KIRIBATI","Bairiki"},
    {"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
    {"MICRONESIA","Palikir"},
    {"NAURU","Yaren"},
    {"NEW ZEALAND","Wellington"},
    {"PALAU","Koror"},
    {"PAPUA NEW GUINEA","Port Moresby"},
    {"SOLOMON ISLANDS","Honaira"},
    {"TONGA","Nuku'alofa"},
    {"TUVALU","Fongafale"},
    {"VANUATU","Port Vila"},
    {"WESTERN SAMOA","Apia"},
    // Eastern Europe and former USSR
    {"ARMENIA","Yerevan"},
    {"AZERBAIJAN","Baku"},
    {"BELARUS (BYELORUSSIA)","Minsk"},
    {"BULGARIA","Sofia"},
    {"GEORGIA","Tbilisi"},
    {"KAZAKSTAN","Almaty"},
    {"KYRGYZSTAN","Alma-Ata"},
    {"MOLDOVA","Chisinau"},
    {"RUSSIA","Moscow"},
    {"TAJIKISTAN","Dushanbe"},
    {"TURKMENISTAN","Ashkabad"},
    {"UKRAINE","Kyiv"},
    {"UZBEKISTAN","Tashkent"},
    // Europe
    {"ALBANIA","Tirana"},
    {"ANDORRA","Andorra la Vella"},
    {"AUSTRIA","Vienna"},
    {"BELGIUM","Brussels"},
    {"BOSNIA-HERZEGOVINA","Sarajevo"},
    {"CROATIA","Zagreb"},
    {"CZECH REPUBLIC","Prague"},
    {"DENMARK","Copenhagen"},
    {"ESTONIA","Tallinn"},
    {"FINLAND","Helsinki"},
    {"FRANCE","Paris"},
    {"GERMANY","Berlin"},
    {"GREECE","Athens"},
    {"HUNGARY","Budapest"},
    {"ICELAND","Reykjavik"},
    {"IRELAND","Dublin"},
    {"ITALY","Rome"},
    {"LATVIA","Riga"},
    {"LIECHTENSTEIN","Vaduz"},
    {"LITHUANIA","Vilnius"},
```

```
    {"LUXEMBOURG","Luxembourg"},
    {"MACEDONIA","Skopje"},
    {"MALTA","Valletta"},
    {"MONACO","Monaco"},
    {"MONTENEGRO","Podgorica"},
    {"THE NETHERLANDS","Amsterdam"},
    {"NORWAY","Oslo"},
    {"POLAND","Warsaw"},
    {"PORTUGAL","Lisbon"},
    {"ROMANIA","Bucharest"},
    {"SAN MARINO","San Marino"},
    {"SERBIA","Belgrade"},
    {"SLOVAKIA","Bratislava"},
    {"SLOVENIA","Ljuijana"},
    {"SPAIN","Madrid"},
    {"SWEDEN","Stockholm"},
    {"SWITZERLAND","Berne"},
    {"UNITED KINGDOM","London"},
    {"VATICAN CITY","Vatican City"},
    // North and Central America
    {"ANTIGUA AND BARBUDA","Saint John's"},
    {"BAHAMAS","Nassau"},
    {"BARBADOS","Bridgetown"},
    {"BELIZE","Belmopan"},
    {"CANADA","Ottawa"},
    {"COSTA RICA","San Jose"},
    {"CUBA","Havana"},
    {"DOMINICA","Roseau"},
    {"DOMINICAN REPUBLIC","Santo Domingo"},
    {"EL SALVADOR","San Salvador"},
    {"GRENADA","Saint George's"},
    {"GUATEMALA","Guatemala City"},
    {"HAITI","Port-au-Prince"},
    {"HONDURAS","Tegucigalpa"},
    {"JAMAICA","Kingston"},
    {"MEXICO","Mexico City"},
    {"NICARAGUA","Managua"},
    {"PANAMA","Panama City"},
    {"ST. KITTS AND NEVIS","Basseterre"},
    {"ST. LUCIA","Castries"},
    {"ST. VINCENT AND THE GRENADINES","Kingstown"},
    {"UNITED STATES OF AMERICA","Washington, D.C."},
    // South America
    {"ARGENTINA","Buenos Aires"},
    {"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
    {"BRAZIL","Brasilia"},
    {"CHILE","Santiago"},
    {"COLOMBIA","Bogota"},
    {"ECUADOR","Quito"},
    {"GUYANA","Georgetown"},
    {"PARAGUAY","Asuncion"},
```

```
    {"PERU","Lima"},
    {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"},
    {"VENEZUELA","Caracas"},
  };
  // Use AbstractMap by implementing entrySet()
  private static class FlyweightMap
  extends AbstractMap<String,String> {
    private static class Entry
    implements Map.Entry<String,String> {
      int index;
      Entry(int index) { this.index = index; }
      @Override
      public boolean equals(Object o) {
        return o instanceof FlyweightMap &&
          Objects.equals(DATA[index][0], o);
      }
      @Override
      public int hashCode() {
        return Objects.hashCode(DATA[index][0]);
      }
      @Override
      public String getKey() { return DATA[index][0]; }
      @Override
      public String getValue() {
        return DATA[index][1];
      }
      @Override
      public String setValue(String value) {
        throw new UnsupportedOperationException();
      }
    }
    // Implement size() & iterator() for AbstractSet:
    static class EntrySet
    extends AbstractSet<Map.Entry<String,String>> {
      private int size;
      EntrySet(int size) {
        if(size < 0)
          this.size = 0;
        // Can't be any bigger than the array:
        else if(size > DATA.length)
          this.size = DATA.length;
        else
          this.size = size;
      }
      @Override
      public int size() { return size; }
      private class Iter
      implements Iterator<Map.Entry<String,String>> {
        // Only one Entry object per Iterator:
```

```
        private Entry entry = new Entry(-1);
        @Override
        public boolean hasNext() {
          return entry.index < size - 1;
        }
        @Override
        public Map.Entry<String,String> next() {
          entry.index++;
          return entry;
        }
        @Override
        public void remove() {
          throw new UnsupportedOperationException();
        }
      }
      @Override
      public
      Iterator<Map.Entry<String,String>> iterator() {
        return new Iter();
      }
    }
    private static
    Set<Map.Entry<String,String>> entries =
      new EntrySet(DATA.length);
    @Override
    public Set<Map.Entry<String,String>> entrySet() {
      return entries;
    }
  }
  // Create a partial map of 'size' countries:
  static Map<String,String> select(final int size) {
    return new FlyweightMap() {
      @Override
      public Set<Map.Entry<String,String>> entrySet() {
        return new EntrySet(size);
      }
    };
  }
  static Map<String,String> map = new FlyweightMap();
  public static Map<String,String> capitals() {
    return map; // The entire map
  }
  public static Map<String,String> capitals(int size) {
    return select(size); // A partial map
  }
  static List<String> names =
    new ArrayList<>(map.keySet());
  // All the names:
  public static List<String> names() { return names; }
  // A partial list:
  public static List<String> names(int size) {
```

```
    return new ArrayList<>(select(size).keySet());
  }
  public static void main(String[] args) {
    System.out.println(capitals(10));
    System.out.println(names(10));
    System.out.println(new HashMap<>(capitals(3)));
    System.out.println(
      new LinkedHashMap<>(capitals(3)));
    System.out.println(new TreeMap<>(capitals(3)));
    System.out.println(new Hashtable<>(capitals(3)));
    System.out.println(new HashSet<>(names(6)));
    System.out.println(new LinkedHashSet<>(names(6)));
    System.out.println(new TreeSet<>(names(6)));
    System.out.println(new ArrayList<>(names(6)));
    System.out.println(new LinkedList<>(names(6)));
    System.out.println(capitals().get("BRAZIL"));
  }
}
/* Output:
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo,
BOTSWANA=Gaberone, BURKINA FASO=Ouagadougou,
BURUNDI=Bujumbura, CAMEROON=Yaounde, CAPE VERDE=Praia,
CENTRAL AFRICAN REPUBLIC=Bangui, CHAD=N'djamena}
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI, CAMEROON, CAPE VERDE, CENTRAL AFRICAN
REPUBLIC, CHAD]
{BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
[BENIN, BOTSWANA, ANGOLA, BURKINA FASO, ALGERIA,
BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
Brasilia
*/
```

二维数组 **String DATA** 是 **public** 的,因此可以在别处使用。**FlyweightMap** 必须实现 `entrySet()` 方法，该方法需要一个自定义 **Set** 实现和一个自定义 **Map.Entry** 类。这是实现享元的另一种方法：每个 **Map.Entry** 对象存储它自身的索引，而不是实际的键和值。当调用 `getKey()` 或 `getValue()` 时，它使用索引返回相应的 **DATA** 元素。**EntrySet** 确保它的 **size** 不大于 **DATA** 。

享元的另一部分在 **EntrySet.Iterator** 中实现。相比于为 **DATA** 中的每个数据对创建一个

Map.Entry 对象，这里每个迭代器只有一个 **Map.Entry** 对象。**Entry** 对象作为数据的窗口，它只包含 **String** 静态数组的索引。每次为迭代器调用 `next()` 时，**Entry** 中的索引都会递增，因此它会指向下一个数据对，然后从 `next()` 返回 **Iterators** 的单个 **Entry** 对象。[1]

`select()` 方法生成一个包含所需大小的 **EntrySet** 的 **FlyweightMap** ，这用于在主方法中演示的重载的 `capitals()` 和 `names()` 方法。

9.8 集合功能

下面这个表格展示了可以对 **Collection** 执行的所有操作（不包括自动继承自 **Object** 的方法），因此，可以用 **List** ，**Set** ，**Queue** 或 **Deque** 执行这里的所有操作（这些接口可能也提供了一些其他的功能）。**Map** 不是从 **Collection** 继承的，所以要单独处理它。

方法名	描述
boolean add(T)	确保集合包含该泛型类型 **T** 的参数。如果不添加参数，则返回 **false** 。（这是一种“可选”方法，将在下一节中介绍。）
boolean addAll(Collection<? extends T>)	添加参数集合中的所有元素。只要有元素被成功添加则返回 **true**。（“可选的”）
void clear()	删除集合中的所有元素。（“可选的”）
boolean contains(T)	如果目标集合包含该泛型类型 **T** 的参数，则返回 **true** 。
boolean containsAll(Collection<?>)	如果目标集合包含参数集合中的所有元素，则返回 **true**
boolean isEmpty()	如果集合为空，则返回 **true**
Iterator<T> iterator() Spliterator<T> spliterator()	返回一个迭代器来遍历集合中的元素。**Spliterators** 更复杂一些，它用在并发场景
boolean remove(Object)	如果目标集合包含该参数，则在集合中删除该参数，如果成功删除则返回 **true** 。（“可选的”）

[1]在某些平台上，特别是 Windows ，默认值可能非常难以查明。你可以使用 -Xss 标志调整堆栈大小。

方法名	描述
boolean removeAll(Collection<?>)	删除目标集合中，参数集合所包含的全部元素。如果有元素被成功删除则返回 **true** 。（“可选的”）
boolean removeIf(Predicate<? super E>)	删除此集合中，满足给定断言（predicate）的所有元素
Stream<E> stream() Stream<E> parallelStream()	返回由该 **Collection** 中元素所组成的一个 **Stream**
int size()	返回集合中所包含元素的个数
Object[] toArrat()	返回包含该集合所有元素的一个数组
<T> T[] toArray(T[] a)	返回包含该集合所有元素的一个数组。结果的运行时类型是参数数组而不是普通的 **Object** 数组。

这里没有提供用于随机访问元素的 **get()** 方法，因为 **Collection** 还包含 **Set** ，它维护自己的内部排序，所以随机访问查找就没有意义了。因此，要查找 **Collection** 中的元素必须使用迭代器。

下面这个示例演示了 **Collection** 的所有方法。这里以 **ArrayList** 为例：

```
// collectiontopics/CollectionMethods.java
// Things you can do with all Collections
import java.util.*;
import static onjava.HTMLColors.*;

public class CollectionMethods {
  public static void main(String[] args) {
    Collection<String> c =
        new ArrayList<>(LIST.subList(0, 4));
    c.add("ten");
    c.add("eleven");
    show(c);
    border();
    // Make an array from the List:
    Object[] array = c.toArray();
    // Make a String array from the List:
    String[] str = c.toArray(new String[0]);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println(
      "Collections.max(c) = " + Collections.max(c));
```

```
    System.out.println(
      "Collections.min(c) = " + Collections.min(c));
    border();
    // Add a Collection to another Collection
    Collection<String> c2 =
        new ArrayList<>(LIST.subList(10, 14));
    c.addAll(c2);
    show(c);
    border();
    c.remove(LIST.get(0));
    show(c);
    border();
    // Remove all components that are
    // in the argument collection:
    c.removeAll(c2);
    show(c);
    border();
    c.addAll(c2);
    show(c);
    border();
    // Is an element in this Collection?
    String val = LIST.get(3);
    System.out.println(
      "c.contains(" + val  + ") = " + c.contains(val));
    // Is a Collection in this Collection?
    System.out.println(
      "c.containsAll(c2) = " + c.containsAll(c2));
    Collection<String> c3 =
      ((List<String>)c).subList(3, 5);
    // Keep all the elements that are in both
    // c2 and c3 (an intersection of sets):
    c2.retainAll(c3);
    show(c2);
    // Throw away all the elements
    // in c2 that also appear in c3:
    c2.removeAll(c3);
    System.out.println(
      "c2.isEmpty() = " +  c2.isEmpty());
    border();
    // Functional operation:
    c = new ArrayList<>(LIST);
    c.removeIf(s -> !s.startsWith("P"));
    c.removeIf(s -> s.startsWith("Pale"));
    // Stream operation:
    c.stream().forEach(System.out::println);
    c.clear(); // Remove all elements
    System.out.println("after c.clear():" + c);
  }
}
/* Output:
AliceBlue
```

```
AntiqueWhite
Aquamarine
Azure
ten
eleven
******************************
Collections.max(c) = ten
Collections.min(c) = AliceBlue
******************************
AliceBlue
AntiqueWhite
Aquamarine
Azure
ten
eleven
Brown
BurlyWood
CadetBlue
Chartreuse
******************************
AntiqueWhite
Aquamarine
Azure
ten
eleven
Brown
BurlyWood
CadetBlue
Chartreuse
******************************
AntiqueWhite
Aquamarine
Azure
ten
eleven
******************************
AntiqueWhite
Aquamarine
Azure
ten
eleven
Brown
BurlyWood
CadetBlue
Chartreuse
******************************
c.contains(Azure) = true
c.containsAll(c2) = true
c2.isEmpty() = true
******************************
PapayaWhip
```

```
PeachPuff
Peru
Pink
Plum
PowderBlue
Purple
after c.clear():[]
*/
```

为了只演示 **Collection** 接口的方法，而没有其它额外的内容，所以这里创建包含不同数据集的 **ArrayList** ，并向上转型为 **Collection** 对象。

9.9 可选操作

在 **Collection** 接口中执行各种添加和删除操作的方法是 *可选操作*（optional operations）。这意味着实现类不需要为这些方法提供功能定义。

这是一种非常不寻常的定义接口的方式。正如我们所知，接口是一种合约（contract）。它表达的意思是，“无论你如何选择实现这个接口，我保证你可以将这些消息发送到这个对象”（我在这里使用术语“接口”来描述正式的 **interface** 关键字和“任何类或子类都支持的方法”的更一般含义）。但“可选”操作违反了这一基本原则，它表示调用某些方法不会执行有意义的行为。相反，它们会抛出异常！这看起来似乎丢失了编译时的类型安全性。

其实没那么糟糕。如果操作是可选的，编译器仍然能够限制你仅调用该接口中的方法。它不像动态语言那样，可以为任何对象调用任何方法，并在运行时查找特定的调用是否可行。[2]此外，大多数将 **Collection** 作为参数的方法仅从该 **Collection** 中读取，并且 **Collection** 的所有“读取”方法都不是可选的。

为什么要将方法定义为“可选”的？因为这样做可以防止设计中的接口爆炸。集合库的其他设计往往会产生令人困惑的过多接口来描述主题的每个变体。这甚至使得不可能捕获到接口中的所有特殊情况，因为总有人能发明一个新的接口。“不支持的操作（unsupported operation）”这种方式实现了 Java 集合库的一个重要目标：集合要易于学习和使用。不支持的操作是一种特殊情况，可以推迟到必要的时候。但是，要使用此方法：

1. **UnsupportedOperationException** 必须是一个罕见的事件。也就是说，对于大多数类，所有操作都应该起作用，并且只有在特殊情况下才应该不支持某项操作。这在 Java 集合库中是正确的，因为 99% 的时间使用到的类 —— **ArrayList** ，**LinkedList** ，**HashSet** 和 **HashMap** ，以及其他具体实现，都支持所有操作。该设计确实为创建一个新的 **Collection** 提供了一个“后门”，可以不为 **Collection** 接口中的所有方法都提供有意义的定义，这些定义仍然适合现有的类库。

[2]引自 Brian Goetz, Java Concurrency in Practice 一书的作者，该书由 Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea 联合著作 (Addison-Wesley 出版社, 2006)。

2. 当不支持某个操作时，**UnsupportedOperationException** 应该出现在实现阶段，而不是在将产品发送给客户之后。毕竟，这个异常表示编程错误：错误地使用了一个具体实现。

值得注意的是，不支持的操作只能在运行时检测到，因此这代表动态类型检查。如果你来自像 C++ 这样的静态类型语言，Java 可能看起来只是另一种静态类型语言。当然，Java 肯定有静态类型检查，但它也有大量的动态类型，因此很难说它只是静态语言或动态语言。一旦你开始注意到这一点，你就会开始看到 Java 中动态类型检查的其他示例。

9.9.1 不支持的操作

不支持的操作的常见来源是由固定大小的数据结构所支持的集合。使用 `Arrays.asList()` 方法将数组转换为 **List** 时，就会得到这样的集合。此外，还可以选择使用 **Collections** 类中的“不可修改（unmodifiable）”方法使任何集合（包括 **Map** ）抛出 **UnsupportedOperationException** 异常。此示例展示了这两种情况：

```
// collectiontopics/Unsupported.java
// Unsupported operations in Java collections
import java.util.*;

public class Unsupported {
  static void
  check(String description, Runnable tst) {
    try {
      tst.run();
    } catch(Exception e) {
      System.out.println(description + "(): " + e);
    }
  }
  static void test(String msg, List<String> list) {
    System.out.println("--- " + msg + " ---");
    Collection<String> c = list;
    Collection<String> subList = list.subList(1,8);
    // Copy of the sublist:
    Collection<String> c2 = new ArrayList<>(subList);
    check("retainAll", () -> c.retainAll(c2));
    check("removeAll", () -> c.removeAll(c2));
    check("clear", () -> c.clear());
    check("add", () -> c.add("X"));
    check("addAll", () -> c.addAll(c2));
    check("remove", () -> c.remove("C"));
    // The List.set() method modifies the value but
    // doesn't change the size of the data structure:
    check("List.set", () -> list.set(0, "X"));
  }
  public static void main(String[] args) {
    List<String> list = Arrays.asList(
      "A B C D E F G H I J K L".split(" "));
```

```
    test("Modifiable Copy", new ArrayList<>(list));
    test("Arrays.asList()", list);
    test("unmodifiableList()",
      Collections.unmodifiableList(
        new ArrayList<>(list)));
  }
}
/* Output:
--- Modifiable Copy ---
--- Arrays.asList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
--- unmodifiableList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
List.set(): java.lang.UnsupportedOperationException
*/
```

因为 `Arrays.asList()` 生成的 **List** 由一个固定大小的数组所支持，所以唯一支持的操作是那些不改变数组大小的操作。任何会导致更改基础数据结构大小的方法都会产生 **UnsupportedOperationException** 异常，来说明这是对不支持的方法的调用（编程错误）。

请注意，始终可以将 `Arrays.asList()` 的结果作为一个参数传递给任何 **Collection** 的构造方法（或使用 `addAll()` 方法或静态的 `Collections.addAll()` 方法）来创建一个允许使用所有方法的常规集合，在主方法中第一次调用 `test()` 时显示了这种情况。这种调用产生了一个新的可调整大小的底层数据结构。

Collections 类中的“unmodifiable”方法会将集合包装一个代理中，如果执行任何想要修改集合的操作，则该代理会生成 **UnsupportedOperationException** 异常。使用这些方法的目的是生成一个“常量”集合对象。稍后将描述 “unmodifiable“集合方法的完整列表。

`test()` 中的最后一个 `check()` 用于测试 **List** 的 `set()` 方法。这里，“不支持的操作”技术的粒度（granularity）就派上用场了，得到的“接口”可以通过一种方法在 `Arrays.asList()` 返回的对象和 `Collections.unmodifiableList()` 返回的对象之间变换。`Arrays.asList()` 返回固定大小的 **List** ，而 `Collections.unmodifiableList()` 生成无法更改的 **List** 。如输出中所示，`Arrays.asList()` 返回的 **List** 中的元素是可以修改的，因为这不会违反该 **List** 的“固定大小”特性。但很明显，`unmodifiableList()` 的结果不应该以任何方式修改。如果使用接口来描述，则需要两个额外的接口，一个具有可用的 `set()` 方法，而另一个没有。**Collection** 的各种不可修改的子类型都将

需要额外的接口。

如果一个方法将一个集合作为它的参数，那么它的文档应该说明必须实现哪些可选方法。

9.10 Set 和存储顺序

集合章节中的 **Set** 有关示例对 **Set** 的基本操作做了很好的介绍。但是，这些示例可以方便地使用预定义的 Java 类型，例如 **Integer** 和 **String** ，它们可以在集合中使用。在创建自己的类型时请注意，**Set** （以及稍后会看到的 **Map** ）需要一种维护存储顺序的方法，该顺序因 **Set** 的不同实现而异。因此，不同的 **Set** 实现不仅具有不同的行为，而且它们对可以放入特定 **Set** 中的对象类型也有不同的要求：

Set 类型	约束
Set(interface)	添加到 **Set** 中的每个元素必须是唯一的，否则，**Set** 不会添加重复元素。添加到 **Set** 的元素必须至少定义 `equals()` 方法以建立对象唯一性。**Set** 与 **Collection** 具有完全相同的接口。**Set** 接口不保证它将以任何特定顺序维护其元素。
HashSet*	注重快速查找元素的集合，其中元素必须定义 `hashCode()` 和 `equals()` 方法。
TreeSet	由树支持的有序 **Set**。这样，就可以从 **Set** 中获取有序序列，其中元素必须实现 **Comparable** 接口。
LinkedHashSet	具有 **HashSet** 的查找速度，但在内部使用链表维护元素的插入顺序。因此，当在遍历 **Set** 时，结果将按元素的插入顺序显示。元素必须定义 `hashCode()` 和 `equals()` 方法。

HashSet 上的星号表示，在没有其他约束的情况下，这应该是你的默认选择，因为它针对速度进行了优化。

定义 `hashCode()` 方法在理解 equals 和 hashCode 方法中进行了描述。必须为散列和树存储结构创建 `equals()` 方法，但只有当把类放在 **HashSet** 中时才需要 `hashCode()` （当然这很有可能，因为 **HashSet** 通常应该是作为 **Set** 实现的首选）或 **LinkedHashSet** 。但是，作为一种良好的编程风格，在覆盖 `equals()` 时应始终覆盖 `hashCode()` 。

下面的示例演示了成功使用具有特定 **Set** 实现的类型所需的方法：

```
// collectiontopics/TypesForSets.java
// Methods necessary to put your own type in a Set
import java.util.*;
import java.util.function.*;
```

```
import java.util.Objects;

class SetType {
  protected int i;
  SetType(int n) { i = n; }
  @Override
  public boolean equals(Object o) {
    return o instanceof SetType &&
      Objects.equals(i, ((SetType)o).i);
  }
  @Override
  public String toString() {
    return Integer.toString(i);
  }
}

class HashType extends SetType {
  HashType(int n) { super(n); }
  @Override
  public int hashCode() {
    return Objects.hashCode(i);
  }
}

class TreeType extends SetType
implements Comparable<TreeType> {
  TreeType(int n) { super(n); }
  @Override
  public int compareTo(TreeType arg) {
    return Integer.compare(arg.i, i);
    // Equivalent to:
    // return arg.i < i ? -1 : (arg.i == i ? 0 : 1);
  }
}

public class TypesForSets {
  static <T> void
  fill(Set<T> set, Function<Integer, T> type) {
    for(int i = 10; i >= 5; i--) // Descending
      set.add(type.apply(i));
    for(int i = 0; i < 5; i++) // Ascending
      set.add(type.apply(i));
  }
  static <T> void
  test(Set<T> set, Function<Integer, T> type) {
    fill(set, type);
    fill(set, type); // Try to add duplicates
    fill(set, type);
    System.out.println(set);
  }
  public static void main(String[] args) {
```

```
    test(new HashSet<>(), HashType::new);
    test(new LinkedHashSet<>(), HashType::new);
    test(new TreeSet<>(), TreeType::new);
    // Things that don't work:
    test(new HashSet<>(), SetType::new);
    test(new HashSet<>(), TreeType::new);
    test(new LinkedHashSet<>(), SetType::new);
    test(new LinkedHashSet<>(), TreeType::new);
    try {
      test(new TreeSet<>(), SetType::new);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
    try {
      test(new TreeSet<>(), HashType::new);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[1, 6, 8, 6, 2, 7, 8, 9, 4, 10, 7, 5, 1, 3, 4, 9, 9,
10, 5, 3, 2, 0, 4, 1, 2, 0, 8, 3, 0, 10, 6, 5, 7]
[3, 1, 4, 8, 7, 6, 9, 5, 3, 0, 10, 5, 5, 10, 7, 8, 8,
9, 1, 4, 10, 2, 6, 9, 1, 6, 0, 3, 2, 0, 7, 2, 4]
[10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5,
0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
[10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5,
0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
SetType cannot be cast to java.lang.Comparable
HashType cannot be cast to java.lang.Comparable
*/
```

为了证明特定 **Set** 需要哪些方法，同时避免代码重复，这里创建了三个类。基类 **SetType** 存储一个 **int** 值，并通过 `toString()` 方法打印它。由于存储在 **Set** 中的所有类都必须具有 `equals()`，因此该方法也放在基类中。基于 `int i` 来判断元素是否相等。

HashType 继承自 **SetType**，并添加了 `hashCode()` 方法，该方法对于 **Set** 的散列实现是必需的。

要在任何类型的有序集合中使用对象，由 **TreeType** 实现的 **Comparable** 接口都是必需的，例如 **SortedSet**（**TreeSet** 是其唯一实现）。在 `compareTo()` 中，请注意我没有使用“简单明了”的形式：`return i-i2`。虽然这是一个常见的编程错误，但只有当 **i** 和 **i2** 是“无符号（unsigned）”整型时才能正常工作（如果 Java 有一个“unsigned”关键字的话，不过它没有）。它破坏了 Java 的有符号 **int**，它不足以代表两个有符号整数的差异。如果 **i** 是一个大的正整数而 **j** 是一个大的负整数，`i-j` 将溢出并返

回一个负值，这不是我们所需要的。

通常希望 `compareTo()` 方法生成与 `equals()` 方法一致的自然顺序。如果 `equals()` 对于特定比较产生 **true**，则 `compareTo()` 应该为该比较返回结果零，并且如果 `equals()` 为比较产生 **false** ，则 `compareTo()` 应该为该比较产生非零结果。

在 **TypesForSets** 中，`fill()` 和 `test()` 都是使用泛型定义的，以防止代码重复。为了验证 **Set** 的行为，`test()` 在测试集上调用 `fill()` 三次，尝试引入重复的对象。`fill()` 方法的参数可以接收任意一个 **Set** 类型，以及生成该类型的 **Function** 对象。因为此示例中使用的所有对象都有一个带有单个 **int** 参数的构造方法，所以可以将构造方法作为此 **Function** 传递，它将提供用于填充 **Set** 的对象。

请注意，`fill()` 方法按降序添加前五个元素，按升序添加后五个元素，以此来指出生成的存储顺序。输出显示 **HashSet** 按升序保留元素，但是，在理解 equals 和 hashCode 方法中，你会发现这只是偶然的，因为散列会创建自己的存储顺序。这里只是因为元素是一个简单的 **int** ，在这种情况下它是升序的。**LinkedHashSet** 按照插入顺序保存元素，**TreeSet** 按排序顺序维护元素（在此示例中因为 `compareTo()` 的实现方式，所以元素按降序排列。）

特定的 **Set** 类型一般都有所必需的操作，如果尝试使用没能正确支持这些操作的类型，那么事情就会出错。将没有重新定义 `hashCode()` 方法的 **SetType** 或 **TreeType** 对象放入任何散列实现会导致重复值，因此违反了 **Set** 的主要契约。这是相当令人不安的，因为这甚至不产生运行时错误。但是，默认的 `hashCode()` 是合法的，所以即使它是不正确的，这也是合法的行为。确保此类程序正确性的唯一可靠方法是将单元测试合并到构建系统中。

如果尝试在 **TreeSet** 中使用没有实现 **Comparable** 接口的类型，则会得到更明确的结果：当 **TreeSet** 尝试将对象用作一个 **Comparable** 时，将会抛出异常。

9.10.1 SortedSet

SortedSet 中的元素保证按排序规则顺序，**SortedSet** 接口中的以下方法可以产生其他功能：

- `Comparator comparator()`：生成用于此 **Set** 的 **Comparator** 或 **null** 来用于自然排序。
- `Object first()`：返回第一个元素。
- `Object last()`：返回最后一个元素。
- `SortedSet subSet(fromElement, toElement)`：使用 **fromElement**（包含）和 **toElement**（不包括）中的元素生成此 **Set** 的一个视图。
- `SortedSet headSet(toElement)`：使用顺序在 **toElement** 之前的元素生成此 **Set** 的一个视图。
- `SortedSet tailSet(fromElement)`：使用顺序在 **fromElement** 之后(包含 **fromElement** ）的元素生成此 **Set** 的一个视图。

下面是一个简单的演示：

```
// collectiontopics/SortedSetDemo.java
```

```
import java.util.*;
import static java.util.stream.Collectors.*;

public class SortedSetDemo {
  public static void main(String[] args) {
    SortedSet<String> sortedSet =
      Arrays.stream(
        "one two three four five six seven eight"
        .split(" "))
        .collect(toCollection(TreeSet::new));
    System.out.println(sortedSet);
    String low = sortedSet.first();
    String high = sortedSet.last();
    System.out.println(low);
    System.out.println(high);
    Iterator<String> it = sortedSet.iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedSet.subSet(low, high));
    System.out.println(sortedSet.headSet(high));
    System.out.println(sortedSet.tailSet(low));
  }
}
/* Output:
[eight, five, four, one, seven, six, three, two]
eight
two
one
two
[one, seven, six, three]
[eight, five, four, one, seven, six, three]
[one, seven, six, three, two]
*/
```

注意，**SortedSet** 表示“根据对象的比较函数进行排序”，而不是“根据插入顺序”。可以使用 **LinkedHashSet** 保留元素的插入顺序。

9.11 队列

有许多 **Queue** 实现，其中大多数是为并发应用程序设计的。许多实现都是通过排序行为而不是性能来区分的。这是一个涉及大多数 **Queue** 实现的基本示例，包括基于并发的队列。队列将元素从一端放入并从另一端取出：

```java
// collectiontopics/QueueBehavior.java
// Compares basic behavior
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;

public class QueueBehavior {
  static Stream<String> strings() {
    return Arrays.stream(
      ("one two three four five six seven " +
       "eight nine ten").split(" "));
  }
  static void test(int id, Queue<String> queue) {
    System.out.print(id + ": ");
    strings().map(queue::offer).count();
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    int count = 10;
    test(1, new LinkedList<>());
    test(2, new PriorityQueue<>());
    test(3, new ArrayBlockingQueue<>(count));
    test(4, new ConcurrentLinkedQueue<>());
    test(5, new LinkedBlockingQueue<>());
    test(6, new PriorityBlockingQueue<>());
    test(7, new ArrayDeque<>());
    test(8, new ConcurrentLinkedDeque<>());
    test(9, new LinkedBlockingDeque<>());
    test(10, new LinkedTransferQueue<>());
    test(11, new SynchronousQueue<>());
  }
}
/* Output:
1: one two three four five six seven eight nine ten
2: eight five four nine one seven six ten three two
3: one two three four five six seven eight nine ten
4: one two three four five six seven eight nine ten
5: one two three four five six seven eight nine ten
6: eight five four nine one seven six ten three two
7: one two three four five six seven eight nine ten
8: one two three four five six seven eight nine ten
9: one two three four five six seven eight nine ten
10: one two three four five six seven eight nine ten
11:
*/
```

Deque 接口也继承自 **Queue** 。除优先级队列外，**Queue** 按照元素的插入顺序生成元素。在此示例中，**SynchronousQueue** 不会产生任何结果，因为它是一个阻塞队列，其中每个插入操作必须等待

另一个线程执行相应的删除操作，反之亦然。

9.11.1 优先级队列

考虑一个待办事项列表，其中每个对象包含一个 **String** 以及主要和次要优先级值。通过实现 **Comparable** 接口来控制此待办事项列表的顺序：

```
// collectiontopics/ToDoList.java
// A more complex use of PriorityQueue
import java.util.*;

class ToDoItem implements Comparable<ToDoItem> {
  private char primary;
  private int secondary;
  private String item;
  ToDoItem(String td, char pri, int sec) {
    primary = pri;
    secondary = sec;
    item = td;
  }
  @Override
  public int compareTo(ToDoItem arg) {
    if(primary > arg.primary)
      return +1;
    if(primary == arg.primary)
      if(secondary > arg.secondary)
        return +1;
      else if(secondary == arg.secondary)
        return 0;
    return -1;
  }
  @Override
  public String toString() {
    return Character.toString(primary) +
      secondary + ": " + item;
  }
}

class ToDoList {
  public static void main(String[] args) {
    PriorityQueue<ToDoItem> toDo =
      new PriorityQueue<>();
    toDo.add(new ToDoItem("Empty trash", 'C', 4));
    toDo.add(new ToDoItem("Feed dog", 'A', 2));
    toDo.add(new ToDoItem("Feed bird", 'B', 7));
    toDo.add(new ToDoItem("Mow lawn", 'C', 3));
    toDo.add(new ToDoItem("Water lawn", 'A', 1));
    toDo.add(new ToDoItem("Feed cat", 'B', 1));
    while(!toDo.isEmpty())
```

```
      System.out.println(toDo.remove());
  }
}
/* Output:
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
*/
```

这展示了通过优先级队列自动排序待办事项。

9.11.2 双端队列

Deque （双端队列）就像一个队列，但是可以从任一端添加和删除元素。Java 6 为 **Deque** 添加了一个显式接口。以下是对实现了 **Deque** 的类的最基本的 **Deque** 方法的测试：

```
// collectiontopics/SimpleDeques.java
// Very basic test of Deques
import java.util.*;
import java.util.concurrent.*;
import java.util.function.*;

class CountString implements Supplier<String> {
  private int n = 0;
  CountString() {}
  CountString(int start) { n = start; }
  @Override
  public String get() {
    return Integer.toString(n++);
  }
}

public class SimpleDeques {
  static void test(Deque<String> deque) {
    CountString s1 = new CountString(),
                s2 = new CountString(20);
    for(int n = 0; n < 8; n++) {
        deque.offerFirst(s1.get());
        deque.offerLast(s2.get()); // Same as offer()
    }
    System.out.println(deque);
    String result = "";
    while(deque.size() > 0) {
      System.out.print(deque.peekFirst() + " ");
      result += deque.pollFirst() + " ";
      System.out.print(deque.peekLast() + " ");
```

```
      result += deque.pollLast() + " ";
    }
    System.out.println("\n" + result);
  }
  public static void main(String[] args) {
    int count = 10;
    System.out.println("LinkedList");
    test(new LinkedList<>());
    System.out.println("ArrayDeque");
    test(new ArrayDeque<>());
    System.out.println("LinkedBlockingDeque");
    test(new LinkedBlockingDeque<>(count));
    System.out.println("ConcurrentLinkedDeque");
    test(new ConcurrentLinkedDeque<>());
  }
}
/* Output:
LinkedList
[7, 6, 5, 4, 3, 2, 1, 0, 20, 21, 22, 23, 24, 25, 26,
27]
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
ArrayDeque
[7, 6, 5, 4, 3, 2, 1, 0, 20, 21, 22, 23, 24, 25, 26,
27]
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
LinkedBlockingDeque
[4, 3, 2, 1, 0, 20, 21, 22, 23, 24]
4 24 3 23 2 22 1 21 0 20
4 24 3 23 2 22 1 21 0 20
ConcurrentLinkedDeque
[7, 6, 5, 4, 3, 2, 1, 0, 20, 21, 22, 23, 24, 25, 26,
27]
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
*/
```

我只使用了 **Deque** 方法的“offer”和“poll”版本，因为当 **LinkedBlockingDeque** 的大小有限时，这些方法不会抛出异常。请注意，**LinkedBlockingDeque** 仅填充到它的限制大小为止，然后忽略额外的添加。

9.12 理解 Map

正如在集合章节中所了解到的，**Map**（也称为 *关联数组*）维护键值关联（对），因此可以使用键来查找值。标准 Java 库包含不同的 **Map** 基本实现，例如 **HashMap** ，**TreeMap** ，**LinkedHashMap** ，**WeakHashMap** ，**ConcurrentHashMap** 和 **IdentityHashMap** 。它们都具有相同的基本 **Map**

接口，但它们的行为不同，包括效率，键值对的保存顺序和呈现顺序，保存对象的时间，如何在多线程程序中工作，以及如何确定键的相等性。**Map** 接口的实现数量应该告诉你一些关于此工具重要性的信息。

为了更深入地了解 **Map**，学习如何构造关联数组会很有帮助。下面是一个非常简单的实现：

```
// collectiontopics/AssociativeArray.java
// Associates keys with values

public class AssociativeArray<K, V> {
  private Object[][] pairs;
  private int index;
  public AssociativeArray(int length) {
    pairs = new Object[length][2];
  }
  public void put(K key, V value) {
    if(index >= pairs.length)
      throw new ArrayIndexOutOfBoundsException();
    pairs[index++] = new Object[]{ key, value };
  }
  @SuppressWarnings("unchecked")
  public V get(K key) {
    for(int i = 0; i < index; i++)
      if(key.equals(pairs[i][0]))
        return (V)pairs[i][1];
    return null; // Did not find key
  }
  @Override
  public String toString() {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < index; i++) {
      result.append(pairs[i][0].toString());
      result.append(" : ");
      result.append(pairs[i][1].toString());
      if(i < index - 1)
        result.append("\n");
    }
    return result.toString();
  }
  public static void main(String[] args) {
    AssociativeArray<String,String> map =
      new AssociativeArray<>(6);
    map.put("sky", "blue");
    map.put("grass", "green");
    map.put("ocean", "dancing");
    map.put("tree", "tall");
    map.put("earth", "brown");
    map.put("sun", "warm");
    try {
      map.put("extra", "object"); // Past the end
```

```
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Too many objects!");
    }
    System.out.println(map);
    System.out.println(map.get("ocean"));
  }
}
/* Output:
Too many objects!
sky : blue
grass : green
ocean : dancing
tree : tall
earth : brown
sun : warm
dancing
*/
```

关联数组中的基本方法是 `put()` 和 `get()` ，但为了便于显示，重写了 `toString()` 方法以打印键值对。为了显示它的工作原理，主方法加载一个带有字符串对的 **AssociativeArray** 并打印生成的映射，然后调用其中一个值的 `get()` 方法。

要使用 `get()` 方法，可以传入要查找的 **key** ，它将生成相关联的值作为结果，如果找不到则返回 **null** 。`get()` 方法使用可能是效率最低的方法来定位值：从数组的头部开始并使用 `equals()` 来比较键。但这里是侧重于简单，而不是效率。

这个版本很有启发性，但它不是很有效，而且它只有一个固定的大小，这是不灵活的。幸运的是，**java.util** 中的那些 **Map** 没有这些问题。

9.12.1 性能

性能是 **Map** 的基本问题，在 `get()` 中使用线性方法搜索一个键时会非常慢。这就是 **HashMap** 要加速的地方。它使用一个称为 *哈希码*的特殊值来替代慢速搜索一个键。哈希码是一种从相关对象中获取一些信息并将其转换为该对象的“相对唯一” **int** 的方法。`hashCode()` 是根类 **Object** 中的一个方法，因此所有 Java 对象都可以生成哈希码。**HashMap** 获取对象的 `hashCode()` 并使用它来快速搜索键。这就使得性能有了显著的提升。[3]

以下是基本的 **Map** 实现。**HashMap** 上的星号表示，在没有其他约束的情况下，这应该是你的默认选择，因为它针对速度进行了优化。其他实现强调其他特性，因此不如 **HashMap** 快。

[3]请注意，在 64 位处理器上可能不会发生这种情况，从而消除了这个问题。

Map 实现	描述
HashMap*	基于哈希表的实现。（使用此类来代替 **Hashtable** 。）为插入和定位键值对提供了常数时间性能。可以通过构造方法调整性能，这些构造方法允许你设置哈希表的容量和装填因子。
LinkedHashMap	与 **HashMap** 类似，但是当遍历时，可以按插入顺序或最近最少使用（LRU）顺序获取键值对。只比 **HashMap** 略慢，一个例外是在迭代时，由于其使用链表维护内部顺序，所以会更快些。
TreeMap	基于红黑树的实现。当查看键或键值对时，它们按排序顺序（由 **Comparable** 或 **Comparator** 确定）。**TreeMap** 的侧重点是按排序顺序获得结果。**TreeMap** 是唯一使用 `subMap()` 方法的 **Map** ，它返回红黑树的一部分。
WeakHashMap	一种具有 *弱键*（weak keys）的 **Map** ，为了解决某些类型的问题，它允许释放 **Map** 所引用的对象。如果在 **Map** 外没有对特定键的引用，则可以对该键进行垃圾回收。
ConcurrentHashMap	不使用同步锁定的线程安全 **Mao** 。这在第二十四章并发编程 一章中讨论。
IdentityHashMap	使用 == 而不是 `equals()` 来比较键。仅用于解决特殊问题，不适用于一般用途。

散列是在 **Map** 中存储元素的最常用方法。

Map 中使用的键的要求与 **Set** 中的元素的要求相同。可以在 **TypesForSets.java** 中看到这些。任何键必须具有 `equals()` 方法。如果键用于散列映射，则它还必须具有正确的 `hashCode()` 方法。如果键在 **TreeMap** 中使用，则必须实现 **Comparable** 接口。

以下示例使用先前定义的 **CountMap** 测试数据集显示通过 **Map** 接口可用的操作：

```
// collectiontopics/MapOps.java
// Things you can do with Maps
import java.util.concurrent.*;
import java.util.*;
import onjava.*;

public class MapOps {
  public static
  void printKeys(Map<Integer,String> map) {
    System.out.print("Size = " + map.size() + ", ");
    System.out.print("Keys: ");
    // Produce a Set of the keys:
```

```
    System.out.println(map.keySet());
  }
  public static
  void test(Map<Integer,String> map) {
    System.out.println(
      map.getClass().getSimpleName());
    map.putAll(new CountMap(25));
    // Map has 'Set' behavior for keys:
    map.putAll(new CountMap(25));
    printKeys(map);
    // Producing a Collection of the values:
    System.out.print("Values: ");
    System.out.println(map.values());
    System.out.println(map);
    System.out.println("map.containsKey(11): " +
      map.containsKey(11));
    System.out.println(
      "map.get(11): " + map.get(11));
    System.out.println("map.containsValue(\"F0\"): "
      + map.containsValue("F0"));
    Integer key = map.keySet().iterator().next();
    System.out.println("First key in map: " + key);
    map.remove(key);
    printKeys(map);
    map.clear();
    System.out.println(
      "map.isEmpty(): " + map.isEmpty());
    map.putAll(new CountMap(25));
    // Operations on the Set change the Map:
    map.keySet().removeAll(map.keySet());
    System.out.println(
      "map.isEmpty(): " + map.isEmpty());
  }
  public static void main(String[] args) {
    test(new HashMap<>());
    test(new TreeMap<>());
    test(new LinkedHashMap<>());
    test(new IdentityHashMap<>());
    test(new ConcurrentHashMap<>());
    test(new WeakHashMap<>());
  }
}
/* Output: (First 11 Lines)
HashMap
Size = 25, Keys: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
Values: [A0, B0, C0, D0, E0, F0, G0, H0, I0, J0, K0,
L0, M0, N0, O0, P0, Q0, R0, S0, T0, U0, V0, W0, X0, Y0]
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0, 10=K0, 11=L0, 12=M0, 13=N0, 14=O0, 15=P0, 16=Q0,
17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0}
```

```
map.containsKey(11): true
map.get(11): L0
map.containsValue("F0"): true
First key in map: 0
Size = 24, Keys: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
map.isEmpty(): true
map.isEmpty(): true
                    ...
*/
```

`printKeys()` 方法演示了如何生成 **Map** 的 **Collection** 视图。`keySet()` 方法生成一个由 **Map** 中的键组成的 **Set** 。打印 `values()` 方法的结果会生成一个包含 **Map** 中所有值的 **Collection** 。（请注意，键必须是唯一的，但值可以包含重复项。）由于这些 **Collection** 由 **Map** 支持，因此 **Collection** 中的任何更改都会反映在所关联的 **Map** 中。

程序的其余部分提供了每个 **Map** 操作的简单示例，并测试了每种基本类型的 **Map** 。

9.12.2 SortedMap

使用 **SortedMap** （由 **TreeMap** 或 **ConcurrentSkipListMap** 实现），键保证按排序顺序，这允许在 **SortedMap** 接口中使用这些方法来提供其他功能：

- `Comparator comparator()` ：生成用于此 **Map** 的比较器，**null** 表示自然排序。
- `T firstKey()` ：返回第一个键。
- `T lastKey()` ：返回最后一个键。
- `SortedMap subMap(fromKey, toKey)` ：生成此 **Map** 的视图，其中键从 **fromKey**（包括），到 **toKey** （不包括）。
- `SortedMap headMap(toKey)` ：使用小于 **toKey** 的键生成此 **Map** 的视图。
- `SortedMap tailMap(fromKey)` ：使用大于或等于 **fromKey** 的键生成此 **Map** 的视图。

这是一个类似于 **SortedSetDemo.java** 的示例，显示了 **TreeMap** 的这种额外行为：

```java
// collectiontopics/SortedMapDemo.java
// What you can do with a TreeMap
import java.util.*;
import onjava.*;

public class SortedMapDemo {
  public static void main(String[] args) {
    TreeMap<Integer,String> sortedMap =
      new TreeMap<>(new CountMap(10));
    System.out.println(sortedMap);
    Integer low = sortedMap.firstKey();
    Integer high = sortedMap.lastKey();
    System.out.println(low);
    System.out.println(high);
```

```
    Iterator<Integer> it =
      sortedMap.keySet().iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedMap.subMap(low, high));
    System.out.println(sortedMap.headMap(high));
    System.out.println(sortedMap.tailMap(low));
  }
}
/* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0}
0
9
3
7
{3=D0, 4=E0, 5=F0, 6=G0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
{3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
*/
```

这里，键值对按照键的排序顺序进行排序。因为 **TreeMap** 中存在顺序感，所以“位置”的概念很有意义，因此可以拥有第一个、最后一个元素或子图。

9.12.3 LinkedHashMap

LinkedHashMap 针对速度进行哈希处理，但在遍历期间也会按插入顺序生成键值对（`System.out.println()` 可以遍历它，因此可以看到遍历的结果）。此外，可以在构造方法中配置 **LinkedHashMap** 以使用基于访问的 *最近最少使用*（LRU）算法，因此未访问的元素（因此是删除的候选者）会出现在列表的前面。这样可以轻松创建一个能够定期清理以节省空间的程序。下面是一个显示这两个功能的简单示例：

```
// collectiontopics/LinkedHashMapDemo.java
// What you can do with a LinkedHashMap
import java.util.*;
import onjava.*;

public class LinkedHashMapDemo {
  public static void main(String[] args) {
    LinkedHashMap<Integer,String> linkedMap =
      new LinkedHashMap<>(new CountMap(9));
    System.out.println(linkedMap);
```

```
    // Least-recently-used order:
    linkedMap =
      new LinkedHashMap<>(16, 0.75f, true);
    linkedMap.putAll(new CountMap(9));
    System.out.println(linkedMap);
    for(int i = 0; i < 6; i++)
      linkedMap.get(i);
    System.out.println(linkedMap);
    linkedMap.get(0);
    System.out.println(linkedMap);
  }
}
/* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
{6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
*/
```

这些键值对确实是按照插入顺序进行遍历，即使对于 LRU 版本也是如此。但是，在 LRU 版本中访问前六项（仅限）后，最后三项将移至列表的前面。然后，当再次访问“ **0** ”后，它移动到了列表的后面。

9.13 集合工具类

集合有许多独立的实用工具程序，在 **java.util.Collections** 中表示为静态方法。之前已经见过其中一些，例如 `addAll()` ，`reverseOrder()` 和 `binarySearch()` 。以下是其他内容（同步和不可修改的实用工具程序将在后面的章节中介绍）。在此表中，在需要的时候使用了泛型：

方法	描述

方法	描述
checkedCollection(Collection<T> c, Class<T> type) checkedList(List<T> list, Class<T> type) checkedMap(Map<K, V> m, Class<K> keyType, Class<V> valueType) checkedSet(Set<T> s, Class<T> type) checkedSortedMap(SortedMap<K, V> m, Class<K> keyType, Class<V> valueType) checkedSortedSet(SortedSet<T> s, Class<T> type)	生成 **Collection** 的动态类型安全视图或 **Collection** 的特定子类型。当无法使用静态检查版本时使用这个版本。这些方法的使用在多态章节的“动态类型安全”标题下进行了展示。
max(Collection) min(Collection)	使用 **Collection** 中对象的自然比较方法生成参数集合中的最大或最小元素。
max(Collection, Comparator) min(Collection, Comparator)	使用 **Comparator** 指定的比较方法生成参数集合中的最大或最小元素。
indexOfSubList(List source, List target)	返回 **target** 在 **source** 内第一次出现的起始索引，如果不存在则返回 -1。
lastIndexOfSubList(List source, List target)	返回 **target** 在 **source** 内最后一次出现的起始索引，如果不存在则返回 -1。
replaceAll(List<T> list, T oldVal, T newVal)	用 **newVal** 替换列表中所有的 **oldVal** 。
reverse(List)	反转列表
reverseOrder() reverseOrder(Comparator<T>)	返回一个 **Comparator** ，它与集合中实现了 **comparable<T>** 接口的对象的自然顺序相反。第二个版本颠倒了所提供的 **Comparator** 的顺序。
rotate(List, int distance)	将所有元素向前移动 **distance** ，将尾部的元素移到开头。（译者注：即循环移动）

方法	描述
shuffle(List) shuffle(List, Random)	随机置换指定列表（即打乱顺序）。第一个版本使用了默认的随机化源，或者也可以使用第二个版本，提供自己的随机化源。
sort(List<T>) sort(List<T>, Comparator<? super T> c)	第一个版本使用元素的自然顺序排序该 **List<T>** 。第二个版本根据提供的 **Comparator** 排序。
copy(List<? super T> dest, List<? extends T> src)	将 **src** 中的元素复制到 **dest** 。
swap(List, int i, int j)	交换 **List** 中位置 **i** 和位置 **j** 的元素。可能比你手工编写的速度快。
fill(List<? super T>, T x)	用 **x** 替换 **List** 中的所有元素。
nCopies(int n, T x)	返回大小为 **n** 的不可变 **List<T>** ，其引用都指向 **x** 。
disjoint(Collection, Collection)	如果两个集合没有共同元素，则返回 **true** 。
frequency(Collection, Object x)	返回 **Collection** 中，等于 **x** 的元素个数。
emptyList() emptyMap() emptySet()	返回不可变的空 **List** ，**Map** 或 **Set** 。这些是泛型的，因此生成的 **Collection** 可以被参数化为所需的类型。
singleton(T x) singletonList(T x) singletonMap(K key, V value)	生成一个不可变的 **List** ，**Set** 或 **Map** ，其中只包含基于给定参数的单个元素。
list(Enumeration<T> e)	生成一个 **ArrayList<T>** ，其中元素为（旧式）**Enumeration** （**Iterator** 的前身）中的元素。用于从遗留代码向新式转换。
enumeration(Collection<T>)	为参数集合生成一个旧式的 **Enumeration<T>** 。

请注意，`min ()` 和 `max()` 使用 **Collection** 对象，而不使用 **List** ，因此不必担心是否应对 **Collection** 进行排序。（如前所述，在执行 `binarySearch()` 之前，将会对 **List** 或数组进行`sort()` 排序。）

下面是一个示例，展示了上表中大多数实用工具程序的基本用法：

```java
// collectiontopics/Utilities.java
// Simple demonstrations of the Collections utilities
import java.util.*;

public class Utilities {
  static List<String> list = Arrays.asList(
    "one Two three Four five six one".split(" "));
  public static void main(String[] args) {
    System.out.println(list);
    System.out.println("'list' disjoint (Four)?: " +
      Collections.disjoint(list,
        Collections.singletonList("Four")));
    System.out.println(
      "max: " + Collections.max(list));
    System.out.println(
      "min: " + Collections.min(list));
    System.out.println(
      "max w/ comparator: " + Collections.max(list,
      String.CASE_INSENSITIVE_ORDER));
    System.out.println(
      "min w/ comparator: " + Collections.min(list,
      String.CASE_INSENSITIVE_ORDER));
    List<String> sublist =
      Arrays.asList("Four five six".split(" "));
    System.out.println("indexOfSubList: " +
      Collections.indexOfSubList(list, sublist));
    System.out.println("lastIndexOfSubList: " +
      Collections.lastIndexOfSubList(list, sublist));
    Collections.replaceAll(list, "one", "Yo");
    System.out.println("replaceAll: " + list);
    Collections.reverse(list);
    System.out.println("reverse: " + list);
    Collections.rotate(list, 3);
    System.out.println("rotate: " + list);
    List<String> source =
      Arrays.asList("in the matrix".split(" "));
    Collections.copy(list, source);
    System.out.println("copy: " + list);
    Collections.swap(list, 0, list.size() - 1);
    System.out.println("swap: " + list);
    Collections.shuffle(list, new Random(47));
    System.out.println("shuffled: " + list);
    Collections.fill(list, "pop");
    System.out.println("fill: " + list);
    System.out.println("frequency of 'pop': " +
      Collections.frequency(list, "pop"));
    List<String> dups =
      Collections.nCopies(3, "snap");
    System.out.println("dups: " + dups);
    System.out.println("'list' disjoint 'dups'?: " +
```

```
      Collections.disjoint(list, dups));
    // Getting an old-style Enumeration:
    Enumeration<String> e =
      Collections.enumeration(dups);
    Vector<String> v = new Vector<>();
    while(e.hasMoreElements())
      v.addElement(e.nextElement());
    // Converting an old-style Vector
    // to a List via an Enumeration:
    ArrayList<String> arrayList =
      Collections.list(v.elements());
    System.out.println("arrayList: " + arrayList);
  }
}
/* Output:
[one, Two, three, Four, five, six, one]
'list' disjoint (Four)?: false
max: three
min: Four
max w/ comparator: Two
min w/ comparator: five
indexOfSubList: 3
lastIndexOfSubList: 3
replaceAll: [Yo, Two, three, Four, five, six, Yo]
reverse: [Yo, six, five, Four, three, Two, Yo]
rotate: [three, Two, Yo, Yo, six, five, Four]
copy: [in, the, matrix, Yo, six, five, Four]
swap: [Four, the, matrix, Yo, six, five, in]
shuffled: [six, matrix, the, Four, Yo, five, in]
fill: [pop, pop, pop, pop, pop, pop, pop]
frequency of 'pop': 7
dups: [snap, snap, snap]
'list' disjoint 'dups'?: true
arrayList: [snap, snap, snap]
*/
```

输出解释了每种实用方法的行为。请注意由于大小写的缘故，普通版本的 `min()` 和 `max()` 与带有 **String.CASE_INSENSITIVE_ORDER** 比较器参数的版本的区别。

9.13.1 排序和搜索列表

用于执行排序和搜索 **List** 的实用工具程序与用于排序对象数组的程序具有相同的名字和方法签名，只不过是 **Collections** 的静态方法而不是 **Arrays** 。这是一个使用 **Utilities.java** 中的 **list** 数据的示例：

```
// collectiontopics/ListSortSearch.java
// Sorting/searching Lists with Collections utilities
import java.util.*;
```

```java

public class ListSortSearch {
  public static void main(String[] args) {
    List<String> list =
      new ArrayList<>(Utilities.list);
    list.addAll(Utilities.list);
    System.out.println(list);
    Collections.shuffle(list, new Random(47));
    System.out.println("Shuffled: " + list);
    // Use ListIterator to trim off last elements:
    ListIterator<String> it = list.listIterator(10);
    while(it.hasNext()) {
      it.next();
      it.remove();
    }
    System.out.println("Trimmed: " + list);
    Collections.sort(list);
    System.out.println("Sorted: " + list);
    String key = list.get(7);
    int index = Collections.binarySearch(list, key);
    System.out.println(
      "Location of " + key + " is " + index +
      ", list.get(" + index + ") = " +
      list.get(index));
    Collections.sort(list,
      String.CASE_INSENSITIVE_ORDER);
    System.out.println(
      "Case-insensitive sorted: " + list);
    key = list.get(7);
    index = Collections.binarySearch(list, key,
      String.CASE_INSENSITIVE_ORDER);
    System.out.println(
      "Location of " + key + " is " + index +
      ", list.get(" + index + ") = " +
      list.get(index));
  }
}
/* Output:
[one, Two, three, Four, five, six, one, one, Two,
three, Four, five, six, one]
Shuffled: [Four, five, one, one, Two, six, six, three,
three, five, Four, Two, one, one]
Trimmed: [Four, five, one, one, Two, six, six, three,
three, five]
Sorted: [Four, Two, five, five, one, one, six, six,
three, three]
Location of six is 7, list.get(7) = six
Case-insensitive sorted: [five, five, Four, one, one,
six, six, three, three, Two]
Location of three is 7, list.get(7) = three
*/
```

就像使用数组进行搜索和排序一样，如果使用 **Comparator** 进行排序，则必须使用相同的 **Comparator** 执行 `binarySearch()` 。

该程序还演示了 **Collections** 中的 `shuffle()` 方法，该方法随机打乱了 **List** 的顺序。**ListIterator** 是在打乱后的列表中的特定位置创建的，用于从该位置删除元素，直到列表末尾。

9.13.2 创建不可修改的 Collection 或 Map

通常，创建 **Collection** 或 **Map** 的只读版本会很方便。**Collections** 类通过将原始集合传递给一个方法然后返回一个只读版本的集合。对于 **Collection** （如果不能将 **Collection** 视为更具体的类型），**List** ，**Set** 和 **Map** ，这类方法有许多变体。这个示例展示了针对每种类型，正确构建只读版本集合的方法：

```
// collectiontopics/ReadOnly.java
// Using the Collections.unmodifiable methods
import java.util.*;
import onjava.*;

public class ReadOnly {
  static Collection<String> data =
    new ArrayList<>(Countries.names(6));
  public static void main(String[] args) {
    Collection<String> c =
      Collections.unmodifiableCollection(
        new ArrayList<>(data));
    System.out.println(c); // Reading is OK
    //- c.add("one"); // Can't change it

    List<String> a = Collections.unmodifiableList(
        new ArrayList<>(data));
    ListIterator<String> lit = a.listIterator();
    System.out.println(lit.next()); // Reading is OK
    //- lit.add("one"); // Can't change it

    Set<String> s = Collections.unmodifiableSet(
      new HashSet<>(data));
    System.out.println(s); // Reading is OK
    //- s.add("one"); // Can't change it

    // For a SortedSet:
    Set<String> ss =
      Collections.unmodifiableSortedSet(
        new TreeSet<>(data));

    Map<String,String> m =
      Collections.unmodifiableMap(
```

```
        new HashMap<>(Countries.capitals(6)));
    System.out.println(m); // Reading is OK
    //- m.put("Ralph", "Howdy!");

    // For a SortedMap:
    Map<String,String> sm =
      Collections.unmodifiableSortedMap(
        new TreeMap<>(Countries.capitals(6)));
  }
}
/* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
ALGERIA
[BENIN, BOTSWANA, ANGOLA, BURKINA FASO, ALGERIA,
BURUNDI]
{BENIN=Porto-Novo, BOTSWANA=Gaberone, ANGOLA=Luanda,
BURKINA FASO=Ouagadougou, ALGERIA=Algiers,
BURUNDI=Bujumbura}
*/
```

为特定类型调用 “unmodifiable” 方法不会导致编译时检查，但是一旦发生转换，对修改特定集合内容的任何方法调用都将产生 **UnsupportedOperationException** 异常。

在每种情况下，在将集合设置为只读之前，必须使用有意义的数据填充集合。填充完成后，最好的方法是用 “unmodifiable” 方法调用生成的引用替换现有引用。这样，一旦使得内容无法修改，那么就不会冒有意外更改内容的风险。另一方面，此工具还允许将可修改的集合保留为类中的**私有**集合，并从方法调用处返回对该集合的只读引用。所以，你可以在类内修改它，但其他人只能读它。

9.13.3 同步 Collection 或 Map

synchronized 关键字是多线程主题的重要组成部分，更复杂的内容在并发编程中介绍。在这里，只需要注意到 **Collections** 类包含一种自动同步整个集合的方法。语法类似于 “unmodifiable” 方法：

```
// collectiontopics/Synchronization.java
// Using the Collections.synchronized methods
import java.util.*;

public class Synchronization {
  public static void main(String[] args) {
    Collection<String> c =
      Collections.synchronizedCollection(
        new ArrayList<>());
    List<String> list = Collections
      .synchronizedList(new ArrayList<>());
    Set<String> s = Collections
```

```
      .synchronizedSet(new HashSet<>());
    Set<String> ss = Collections
      .synchronizedSortedSet(new TreeSet<>());
    Map<String,String> m = Collections
      .synchronizedMap(new HashMap<>());
    Map<String,String> sm = Collections
      .synchronizedSortedMap(new TreeMap<>());
  }
}
```

最好立即通过适当的 “synchronized” 方法传递新集合，如上所示。这样，就不会意外地暴露出非同步版本。

Fail Fast Java 集合还具有防止多个进程修改集合内容的机制。如果当前正在迭代集合，然后有其他一些进程介入并插入，删除或更改该集合中的对象，则会出现此问题。也许在集合中已经遍历过了那个元素，也许还没有遍历到，也许在调用 `size()` 之后集合的大小会缩小...有许多灾难情景。Java 集合库使用一种 *fail-fast* 的机制，该机制可以检测到除了当前进程引起的更改之外，其它任何对集合的更改操作。如果它检测到其他人正在修改集合，则会立即生成 **ConcurrentModificationException** 异常。这就是“fail-fast”的含义——它不会在以后使用更复杂的算法尝试检测问题（快速失败）。

通过创建迭代器并向迭代器指向的集合中添加元素，可以很容易地看到操作中的 fail-fast 机制，如下所示：

```
// collectiontopics/FailFast.java
// Demonstrates the "fail-fast" behavior
import java.util.*;

public class FailFast {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<>();
    Iterator<String> it = c.iterator();
    c.add("An object");
    try {
      String s = it.next();
    } catch(ConcurrentModificationException e) {
      System.out.println(e);
    }
  }
}
/* Output:
java.util.ConcurrentModificationException
*/
```

异常来自于在从集合中获得迭代器之后，又尝试在集合中添加元素。程序的两个部分可能会修改同一个集合，这种可能性的存在会产生不确定状态，因此异常会通知你更改代码。在这种情况下，应先将所有元素添加到集合，然后再获取迭代器。

ConcurrentHashMap，**CopyOnWriteArrayList** 和 **CopyOnWriteArraySet** 使用了特定的技术来避免产生 **ConcurrentModificationException** 异常。

9.14 持有引用

java.lang.ref 中库包含一组类，这些类允许垃圾收集具有更大的灵活性。特别是当拥有可能导致内存耗尽的大对象时，这些类特别有用。这里有三个从抽象类 **Reference** 继承来的类：**SoftReference**（软引用），**WeakReference**（弱引用）和 **PhantomReference**（虚引用）继承了三个类。如果一个对象只能通过这其中的一个 **Reference** 对象访问，那么这三种类型每个都为垃圾收集器提供不同级别的间接引用（indirection）。

如果一个对象是 *可达的*（reachable），那么意味着在程序中的某个位置可以找到该对象。这可能意味着在栈上有一个直接引用该对象的普通引用，但也有可能是引用了一个对该对象有引用的对象，这可以有很多中间环节。如果某个对象是可达的，则垃圾收集器无法释放它，因为它仍然被程序所使用。如果某个对象是不可达的，则程序无法使用它，那么垃圾收集器回收该对象就是安全的。

使用 **Reference** 对象继续保持对该对象的引用，以到达该对象，但也允许垃圾收集器释放该对象。因此，程序可以使用该对象，但如果内存即将耗尽，则允许释放该对象。

可以通过使用 **Reference** 对象作为你和普通引用之间的中介（代理）来实现此目的。此外，必须没有对象的普通引用（未包含在 **Reference** 对象中的对象）。如果垃圾收集器发现对象可通过普通引用访问，则它不会释放该对象。

按照 **SoftReference**，**WeakReference** 和 **PhantomReference** 的顺序，每个都比前一个更“弱”，并且对应于不同的可达性级别。软引用用于实现对内存敏感的缓存。弱引用用于实现“规范化映射”（canonicalized mappings）——对象的实例可以在程序的多个位置同时使用，以节省存储，但不会阻止其键（或值）被回收。虚引用用于调度 pre-mortem 清理操作，这是一种比 Java 终结机制（Java finalization mechanism）更灵活的方式。

使用 **SoftReference** 和 **WeakReference**，可以选择是否将它们放在 **ReferenceQueue**（用于 pre-mortem 清理操作的设备）中，但 **PhantomReference** 只能在 **ReferenceQueue** 上构建。下面是一个简单的演示：

```
// collectiontopics/References.java
// Demonstrates Reference objects
import java.lang.ref.*;
import java.util.*;

class VeryBig {
  private static final int SIZE = 10000;
  private long[] la = new long[SIZE];
  private String ident;
  VeryBig(String id) { ident = id; }
```

```
  @Override
  public String toString() { return ident; }
  @Override
  protected void finalize() {
    System.out.println("Finalizing " + ident);
  }
}

public class References {
  private static ReferenceQueue<VeryBig> rq =
    new ReferenceQueue<>();
  public static void checkQueue() {
    Reference<? extends VeryBig> inq = rq.poll();
    if(inq != null)
      System.out.println("In queue: " + inq.get());
  }
  public static void main(String[] args) {
    int size = 10;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.valueOf(args[0]);
    LinkedList<SoftReference<VeryBig>> sa =
      new LinkedList<>();
    for(int i = 0; i < size; i++) {
      sa.add(new SoftReference<>(
        new VeryBig("Soft " + i), rq));
      System.out.println(
        "Just created: " + sa.getLast());
      checkQueue();
    }
    LinkedList<WeakReference<VeryBig>> wa =
      new LinkedList<>();
    for(int i = 0; i < size; i++) {
      wa.add(new WeakReference<>(
        new VeryBig("Weak " + i), rq));
      System.out.println(
        "Just created: " + wa.getLast());
      checkQueue();
    }
    SoftReference<VeryBig> s =
      new SoftReference<>(new VeryBig("Soft"));
    WeakReference<VeryBig> w =
      new WeakReference<>(new VeryBig("Weak"));
    System.gc();
    LinkedList<PhantomReference<VeryBig>> pa =
      new LinkedList<>();
    for(int i = 0; i < size; i++) {
      pa.add(new PhantomReference<>(
        new VeryBig("Phantom " + i), rq));
      System.out.println(
        "Just created: " + pa.getLast());
```

```
      checkQueue();
    }
  }
}
/* Output: (First and Last 10 Lines)
Just created: java.lang.ref.SoftReference@15db9742
Just created: java.lang.ref.SoftReference@6d06d69c
Just created: java.lang.ref.SoftReference@7852e922
Just created: java.lang.ref.SoftReference@4e25154f
Just created: java.lang.ref.SoftReference@70dea4e
Just created: java.lang.ref.SoftReference@5c647e05
Just created: java.lang.ref.SoftReference@33909752
Just created: java.lang.ref.SoftReference@55f96302
Just created: java.lang.ref.SoftReference@3d4eac69
Just created: java.lang.ref.SoftReference@42a57993
...________...________...________...________...
Just created: java.lang.ref.PhantomReference@45ee12a7
In queue: null
Just created: java.lang.ref.PhantomReference@330bedb4
In queue: null
Just created: java.lang.ref.PhantomReference@2503dbd3
In queue: null
Just created: java.lang.ref.PhantomReference@4b67cf4d
In queue: null
Just created: java.lang.ref.PhantomReference@7ea987ac
In queue: null
*/
```

当运行此程序（将输出重定向到文本文件以查看页面中的输出）时，将会看到对象是被垃圾收集了的，虽然仍然可以通过 **Reference** 对象访问它们（使用 `get()` 来获取实际的对象引用）。还可以看到 **ReferenceQueue** 始终生成包含 **null** 对象的 **Reference** 。要使用它，请从特定的 **Reference** 类继承，并为新类添加更多有用的方法。

9.14.1 WeakHashMap

集合类库中有一个特殊的 **Map** 来保存弱引用：**WeakHashMap** 。此类可以更轻松地创建规范化映射。在这种映射中，可以通过仅仅创建一个特定值的实例来节省存储空间。当程序需要该值时，它会查找映射中的现有对象并使用它（而不是从头开始创建一个）。该映射可以将值作为其初始化的一部分，但更有可能的是在需要时创建该值。

由于这是一种节省存储空间的技术，因此 **WeakHashMap** 允许垃圾收集器自动清理键和值，这是非常方便的。不能对放在 **WeakHashMap** 中的键和值做任何特殊操作，它们由 map 自动包装在 **WeakReference** 中。当键不再被使用的时候才允许清理，如下所示：

```
// collectiontopics/CanonicalMapping.java
// Demonstrates WeakHashMap
```

```java
import java.util.*;

class Element {
  private String ident;
  Element(String id) { ident = id; }
  @Override
  public String toString() { return ident; }
  @Override
  public int hashCode() {
    return Objects.hashCode(ident);
  }
  @Override
  public boolean equals(Object r) {
    return r instanceof Element &&
      Objects.equals(ident, ((Element)r).ident);
  }
  @Override
  protected void finalize() {
    System.out.println("Finalizing " +
      getClass().getSimpleName() + " " + ident);
  }
}

class Key extends Element {
  Key(String id) { super(id); }
}

class Value extends Element {
  Value(String id) { super(id); }
}

public class CanonicalMapping {
  public static void main(String[] args) {
    int size = 1000;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.valueOf(args[0]);
    Key[] keys = new Key[size];
    WeakHashMap<Key,Value> map =
      new WeakHashMap<>();
    for(int i = 0; i < size; i++) {
      Key k = new Key(Integer.toString(i));
      Value v = new Value(Integer.toString(i));
      if(i % 3 == 0)
        keys[i] = k; // Save as "real" references
      map.put(k, v);
    }
    System.gc();
  }
}
```

Key 类必须具有 `hashCode()` 和 `equals()`，因为它将被用作散列数据结构中的键。`hashCode()` 的内容在附录：理解 hashCode 和 equals 方法中进行了描述。

运行程序，你会看到垃圾收集器每三个键跳过一次。对该键的普通引用也被放置在 **keys** 数组中，因此这些对象不能被垃圾收集。

9.15 Java 1.0 / 1.1 的集合类

不幸的是，许多代码是使用 Java 1.0 / 1.1 中的集合编写的，甚至新代码有时也是使用这些类编写的。编写新代码时切勿使用旧集合。旧的集合类有限，所以关于它们的讨论不多。由于它们是不合时宜的，所以我会尽量避免过分强调一些可怕的设计决定。

9.15.1 Vector 和 Enumeration

Java 1.0 / 1.1 中唯一的自扩展序列是 **Vector**，因此它被用于很多地方。它的缺陷太多了，无法在这里描述（参见《Java 编程思想》第 1 版，可从www.OnJava8.com免费下载）。基本上，你可以将它看作是具有冗长且笨拙的方法名称的 **ArrayList**。在修订后的 Java 集合库中，**Vector** 已经被调整适配过，因此可以作为 **Collection** 和 **List** 来使用。事实证明这有点不正常，集合类库仍然包含它只是为了支持旧的 Java 代码，但这会让一些人误以为 **Vector** 已经变得更好了。

迭代器的 Java 1.0 / 1.1 版本选择创建一个新名称“enumeration”，而不是使用每个人都熟悉的术语（“iterator”）。**Enumeration** 接口小于 **Iterator**，只包含两个方法，并且它使用更长的方法名称：如果还有更多元素，则 `boolean hasMoreElements()` 返回 `true`，`Object nextElement()` 返回此 enumeration 的下一个元素（否则会抛出异常）。

Enumeration 只是一个接口，而不是一个实现，甚至新的类库有时仍然使用旧的 **Enumeration**，这是不幸的，但通常是无害的。应该总是在自己的代码中使用 **Iterator**，但要做好准备应对那些提供 **Enumeration** 的类库。

此外，可以使用 `Collections.enumeration()` 方法为任何 **Collection** 生成 **Enumeration**，如下例所示：

```
// collectiontopics/Enumerations.java
// Java 1.0/1.1 Vector and Enumeration
import java.util.*;
import onjava.*;

public class Enumerations {
  public static void main(String[] args) {
    Vector<String> v =
      new Vector<>(Countries.names(10));
    Enumeration<String> e = v.elements();
    while(e.hasMoreElements())
```

```
      System.out.print(e.nextElement() + ", ");
    // Produce an Enumeration from a Collection:
    e = Collections.enumeration(new ArrayList<>());
  }
}
/* Output:
ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI, CAMEROON, CAPE VERDE, CENTRAL AFRICAN
REPUBLIC, CHAD,
*/
```

要生成 **Enumeration** ，可以调用 `elements()` ，然后可以使用它来执行向前迭代。

最后一行创建一个 **ArrayList** ，并使用 `enumeration()` 来将 **ArrayList** 适配为一个 **Enumeration** 。因此，如果有旧代码需要使用 **Enumeration** ，你仍然可以使用新集合。

9.15.2 Hashtable

正如你在本附录中的性能比较中所看到的，基本的 **Hashtable** 与 **HashMap** 非常相似，甚至方法名称都相似。在新代码中没有理由使用 **Hashtable** 而不是 **HashMap** 。

9.15.3 Stack

之前使用 **LinkedList** 引入了栈的概念。Java 1.0 / 1.1 **Stack** 的奇怪之处在于，不是以组合方式使用 **Vector** ，而是继承自 **Vector** 。因此它具有 **Vector** 的所有特征和行为以及一些额外的 **Stack** 行为。很难去知道设计师是否有意识地认为这样做是有用的，或者它是否只是太天真了，无论如何，它在进入发行版之前显然没有经过审查，所以这个糟糕的设计仍然存在（但不要使用它）。

这是 **Stack** 的简单演示，向栈中放入枚举中每一个类型的 **String** 形式。它还展示了如何轻松地将 **LinkedList** 用作栈，或者使用在第十二章：集合章节中创建的 **Stack** 类：

```
// collectiontopics/Stacks.java
// Demonstration of Stack Class
import java.util.*;

enum Month { JANUARY, FEBRUARY, MARCH, APRIL,
  MAY, JUNE, JULY, AUGUST, SEPTEMBER,
  OCTOBER, NOVEMBER }

public class Stacks {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    for(Month m : Month.values())
      stack.push(m.toString());
    System.out.println("stack = " + stack);
    // Treating a stack as a Vector:
```

```
    stack.addElement("The last line");
    System.out.println(
      "element 5 = " + stack.elementAt(5));
    System.out.println("popping elements:");
    while(!stack.empty())
      System.out.print(stack.pop() + " ");

    // Using a LinkedList as a Stack:
    LinkedList<String> lstack = new LinkedList<>();
    for(Month m : Month.values())
      lstack.addFirst(m.toString());
    System.out.println("lstack = " + lstack);
    while(!lstack.isEmpty())
      System.out.print(lstack.removeFirst() + " ");

    // Using the Stack class from
    // the Collections Chapter:
    onjava.Stack<String> stack2 =
      new onjava.Stack<>();
    for(Month m : Month.values())
      stack2.push(m.toString());
    System.out.println("stack2 = " + stack2);
    while(!stack2.isEmpty())
      System.out.print(stack2.pop() + " ");

  }
}
/* Output:
stack = [JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER]
element 5 = JUNE
popping elements:
The last line NOVEMBER OCTOBER SEPTEMBER AUGUST JULY
JUNE MAY APRIL MARCH FEBRUARY JANUARY lstack =
[NOVEMBER, OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE, MAY,
APRIL, MARCH, FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL
MARCH FEBRUARY JANUARY stack2 = [NOVEMBER, OCTOBER,
SEPTEMBER, AUGUST, JULY, JUNE, MAY, APRIL, MARCH,
FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL
MARCH FEBRUARY JANUARY
*/
```

String 形式是由 **Month** 中的枚举常量生成的，使用 `push()` 压入到栈中，然后使用 `pop()` 从栈顶部取出。为了说明一点，将 **Vector** 的操作也在 **Stack** 对象上执行，这是可能的，因为凭借继承，**Stack** 是 **Vector** 。因此，可以在 **Vector** 上执行的所有操作也可以在 **Stack** 上执行，例如 `elementAt()` 。

如前所述，在需要栈行为时使用 **LinkedList** ，或者从 **LinkedList** 类创建的 **onjava.Stack** 类。

9.15.4 BitSet

BitSet 用于有效地存储大量的开关信息。仅从尺寸大小的角度来看它是有效的，如果你正在寻找有效的访问，它比使用本机数组（native array）稍慢。

此外，**BitSet** 的最小大小是 **long**：64 位。这意味着如果你要存储更小的东西，比如 8 位，**BitSet** 就是浪费，如果尺寸有问题，你最好创建自己的类，或者只是用一个数组来保存你的标志。（只有在你创建许多包含开关信息列表的对象时才会出现这种情况，并且只应根据分析和其他指标来决定。如果你做出此决定只是因为您认为 **BitSet** 太大，那么最终会产生不必要的复杂性并且浪费大量时间。）

当添加更多元素时，普通集合会扩展，**BitSet** 也会这样做。以下示例显示了 **BitSet** 的工作原理：

```
// collectiontopics/Bits.java
// Demonstration of BitSet
import java.util.*;

public class Bits {
  public static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    StringBuilder bbits = new StringBuilder();
    for(int j = 0; j < b.size() ; j++)
      bbits.append(b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    // Take the LSB of nextInt():
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >= 0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("byte value: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >= 0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    System.out.println("short value: " + st);
    printBitSet(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >= 0; i--)
```

```
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("int value: " + it);
    printBitSet(bi);

    // Test bitsets >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    System.out.println("set bit 1023: " + b1023);
  }
}
/* Output:
byte value: -107
bits: {0, 2, 4, 7}
bit pattern: 1010100100000000000000000000000000000000
00000000000000000000
short value: 1302
bits: {1, 2, 4, 8, 10}
bit pattern: 0110100010100000000000000000000000000000
00000000000000000000
int value: -2014573909
bits: {0, 1, 3, 5, 7, 9, 11, 18, 19, 21, 22, 23, 24,
25, 26, 31}
bit pattern: 1101010101010000001101111110000100000000
00000000000000000000
set bit 127: {127}
set bit 255: {255}
set bit 1023: {1023, 1024}
*/
```

随机数生成器用于创建随机 **byte**，**short** 和 **int**，并且每个都在 **BitSet** 中转换为相应的位模式。这样可以正常工作，因为 **BitSet** 是 64 位，所以这些都不会导致它的大小增加，然后创建更大的 **BitSet**。请注意，**BitSet** 会根据需要进行扩展。

对于可以命名的固定标志集，**EnumSet**（参见第二十二章：枚举章节）通常比 **BitSet** 更好，因为 **EnumSet** 允许操作名称而不是数字位位置，从而可以减少错误。**EnumSet** 还可以防止意外地添加新的标记位置，这可能会导致一些严重的，难以发现的错误。使用 **BitSet** 而不是 **EnumSet** 的唯一原因是，不知道在运行时需要多少标志，或者为标志分配名称是不合理的，或者需要 **BitSet** 中的一个特殊操作（请参阅 **BitSet** 和 **EnumSet** 的 JDK 文档）。

9.16 本章小结

集合可以说是编程语言中最常用的工具。有些语言（例如 Python）甚至将基本集合组件（列表，映射和集合）作为内置函数包含在其中。

正如在第十二章：集合章节中看到的那样，可以使用集合执行许多非常有用的操作，而不需要太多努力。但是，在某些时候，为了正确地使用它们而不得不更多地了解集合，特别是，必须充分了解散列操作以编写自己的 `hashCode()` 方法（并且必须知道何时需要），并且你必须充分了解各种集合实现，以根据你的需求选择合适的集合。本附录涵盖了这些概念，并讨论了有关集合库的其他有用详细信息。你现在应该已经准备好在日常编程任务中使用 Java 集合了。

集合库的设计很困难（大多数库设计问题都是如此）。在 C++ 中，集合类涵盖了许多不同类的基础。这比之前可用的 C++ 集合类更好，但它没有很好地转换为 Java 。在另一个极端，我看到了一个由单个类“collection”组成的集合库，它同时充当线性序列和关联数组。Java 集合库试图在功能和复杂性之间取得平衡。结果在某些地方看起来有点奇怪。与早期 Java 库中的一些决策不同，这些奇怪的不是事故，而是在基于复杂性的权衡下而仔细考虑的决策。

附录 10

并发底层原理

> 尽管不建议你自己编写底层 Java 并发代码，但是这样通常有助于了解它是如何工作的。

并发编程 章节中介绍了一些用于高级并发的概念，包括为 Java 并发编程而最新提出的，更安全的概念（parallel Streams 和 CompletableFutures ）。本附录则介绍在 Java 中底层并发概念，因此在阅读本篇时，你能有所了解掌握这些代码。你还会将进一步了解并发的普遍问题。

在 Java 的早期版本中, 底层并发概念是并发编程的重要组成部分。我们会着眼于围绕这些技巧的复杂性以及为何你应该避免它们而谈。“并发编程” 章节展示最新的 Java 版本 (尤其是 Java 8) 所提供的改进技巧，这些技巧使得并发的使用，如果本来不容易使用，也会变得更容易些。

10.1 什么是线程?

并发将程序划分成独立分离运行的任务。每个任务都由一个 *执行线程*来驱动，我们通常将其简称为 *线程*。而一个 *线程*就是操作系统进程中单一顺序的控制流。因此，单个进程可以有多个并发执行的任务，但是你的程序使得每个任务都好像有自己的处理器一样。此线程模型为编程带来了便利，它简化了在单一程序中处理变戏法般的多任务过程。操作系统则从处理器上分配时间片到你程序的所有线程中。

Java 并发的核心机制是 **Thread** 类，在该语言最初版本中，**Thread （线程）**是由程序员直接创建和管理的。随着语言的发展以及人们发现了更好的一些方法，中间层机制 - 特别是 **Executor** 框架 - 被添加进来，以消除自己管理线程时候的心理负担（及错误）。最终，甚至发展出比 **Executor** 更好的机制，如 并发编程 一章所示。

Thread（线程）是将任务关联到处理器的软件概念。虽然创建和使用 **Thread** 类看起来与任何其他类都很相似，但实际上它们是非常不同的。当你创建一个 **Thread** 时，JVM 将分配一大块内存到专为线程保留的特殊区域上，用于提供运行任务时所需的一切，包括:

- 程序计数器，指明要执行的下一个 JVM 字节码指令。
- 用于支持 Java 代码执行的栈，包含有关此线程已到达当时执行位置所调用方法的信息。它也包含每个正在执行的方法的所有局部变量 (包括原语和堆对象的引用)。每个线程的栈通常在 64K

到 1M 之间 [1] 。

- 第二个则用于 native code（本机方法代码）执行的栈
- *thread-local variables* （线程本地变量）的存储区域
- 用于控制线程的状态管理变量

包括 `main()` 在内的所有代码都会在某个线程内运行。每当调用一个方法时，当前程序计数器被推到该线程的栈上，然后栈指针向下移动以足够来创建一个栈帧，其栈帧里存储该方法的所有局部变量，参数和返回值。所有基本类型变量都直接在栈上，虽然方法中创建（或方法中使用）对象的任何引用都位于栈帧中，但对象本身存于堆中。这仅且只有一个堆，被程序中所有线程所共享。

除此以外，线程必须绑定到操作系统，这样它就可以在某个时候连接到处理器。这是作为线程构建过程的一部分为你管理的。Java 使用底层操作系统中的机制来管理线程的执行。

10.1.1 最佳线程数

如果你查看第 24 章 并发编程 中使用 *CachedThreadPool* 的用例，你会发现 **ExecutorService** 为每个我们提交的任务分配一个线程。然而，并行流（**parallel Stream**）在 **CountingStream.java** 中只分配了 8 个线程（id 中 1-7 为工作线程，8 为 `main()` 方法的主线程，它巧妙地将其用作额外的并行流）。如果你尝试提高 `range()` 方法中的上限值，你会看到没有创建额外的线程。这是为什么？

我们可以查出当前机器上处理器的数量：

```java
// lowlevel/NumberOfProcessors.java

public class NumberOfProcessors {
  public static void main(String[] args) {
    System.out.println(
    Runtime.getRuntime().availableProcessors());
  }
}
/* Output:
8
*/
```

在我的机器上（使用英特尔酷睿 i7），我有四个内核，每个内核呈现两个*超线程*（指一种硬件技巧，能在单个处理器上产生非常快速的上下文切换，在某些情况下可以使内核看起来像运行两个硬件线程）。虽然这是 “最近” 计算机上的常见配置 (在撰写本文时)，但你可能会看到不同的结果，包括 **CountingStream.java** 中同等数量的默认线程。

你的操作系统可能有办法来查出关于处理器的更多信息，例如，在 Windows 10 上，按下 “开始” 键，输入 “任务管理器” 和 Enter 键。点击 “详细信息” 。选择 “性能” 标签, 你将会看到各种各样的关于你的硬件信息, 包括“内核” 和 “逻辑处理器” 。

事实证明，“通用”线程的最佳数量就算是可用处理器的数量 (对于特定的问题可能不是这样)。这

[1] 在某些平台上，特别是 Windows ，默认值可能非常难以查明。你可以使用 -Xss 标志调整堆栈大小。

原因来自在 Java 线程之间切换上下文的代价：存储被挂起线程的当前状态，并检索另一个线程的当前状态，以便从它进入挂起的位置继续执行。对于 8 个处理器和 8 个（计算密集型）Java 线程，JVM 在运行这 8 个任务时从不需要切换上下文。对于比处理器数量少的任务，分配更多线程没有帮助。

定义了 "逻辑处理器" 数量的 Intel 超线程，但并没有增加计算能力 - 该特性在硬件级别维护额外的线程上下文，从而加快了上下文切换，这有助于提高用户界面的响应能力。对于计算密集型任务，请考虑将线程数量与物理内核 (而不是超线程) 的数量匹配。尽管 Java 认为每个超线程都是一个处理器，但这似乎是由于 Intel 对超线程的过度营销造成的错误。尽管如此，为了简化编程，我只允许 JVM 决定默认的线程数。你将需要试验你的产品应用。这并不意味着将线程数与处理器数相匹配就适用于所有问题; 相反，它主要用于计算密集型解决方案。

10.1.2 我可以创建多少个线程?

Thread（线程）对象的最大部分是用于执行方法的 Java 堆栈。查看 Thread （线程）对象的大小因操作系统而异。该程序通过创建 Thread 对象来测试它，直到 JVM 内存不足为止：

```java
// lowlevel/ThreadSize.java
// {ExcludeFromGradle} Takes a long time or hangs
import java.util.concurrent.*;
import onjava.Nap;

public class ThreadSize {
  static class Dummy extends Thread {
    @Override
    public void run() { new Nap(1); }
  }
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newCachedThreadPool();
    int count = 0;
    try {
      while(true) {
        exec.execute(new Dummy());
        count++;
      }
    } catch(Error e) {
      System.out.println(
      e.getClass().getSimpleName() + ": " + count);
      System.exit(0);
    } finally {
      exec.shutdown();
    }
  }
}
```

只要你不断递交任务,**CachedThreadPool** 就会继续创建线程。将 **Dummy** 对象递交到 `execute`

() 方法以开始任务，如果线程池无可用线程，则分配一个新线程。执行的暂停方法 `pause()` 运行时间必须足够长，使任务不会开始即完成 (从而为新任务释放现有线程)。只要任务不断进入而没有完成，**CachedThreadPool** 最终就会耗尽内存。

我并不总是能够在我尝试的每台机器上造成内存不足的错误。在一台机器上，我看到这样的结果:

```
> java ThreadSize
OutOfMemoryError: 2816
```

我们可以使用 **-Xss** 标记减少每个线程栈分配的内存大小。允许的最小线程栈大小是 64k:

```
>java -Xss64K ThreadSize
OutOfMemoryError: 4952
```

如果我们将线程栈大小增加到 2M ，我们就可以分配更少的线程。

```
>java -Xss2M ThreadSize
OutOfMemoryError: 722
```

Windows 操作系统默认栈大小是 320K，我们可以通过验证它给出的数字与我们完全不设置栈大小时的数字是大致相同:

```
>java -Xss320K ThreadSize
OutOfMemoryError: 2816
```

你还可以使用 **-Xmx** 标志增加 JVM 的最大内存分配:

```
>java -Xss64K -Xmx5M ThreadSize
OutOfMemoryError: 5703
```

请注意的是操作系统还可能对允许的线程数施加限制。

因此，“我可以拥有多少线程”这一问题的答案是“几千个”。但是，如果你发现自己分配了数千个线程，那么你可能需要重新考虑你的做法; 恰当的问题是“我需要多少线程？”

10.1.3 The WorkStealingPool (工作窃取线程池)

这是一个 **ExecutorService** ，它使用所有可用的 (由 JVM 报告) 处理器自动创建线程池。

```java
// lowlevel/WorkStealingPool.java
import java.util.stream.*;
import java.util.concurrent.*;

class ShowThread implements Runnable {
  @Override
  public void run() {
    System.out.println(
    Thread.currentThread().getName());
```

“不可能”只存在于蠢人的字典里。

```
  }
}

public class WorkStealingPool {
  public static void main(String[] args)
    throws InterruptedException {
    System.out.println(
      Runtime.getRuntime().availableProcessors());
    ExecutorService exec =
      Executors.newWorkStealingPool();
    IntStream.range(0, 10)
      .mapToObj(n -> new ShowThread())
      .forEach(exec::execute);
    exec.awaitTermination(1, TimeUnit.SECONDS);
  }
}
/* Output:
8
ForkJoinPool-1-worker-2
ForkJoinPool-1-worker-1
ForkJoinPool-1-worker-2
ForkJoinPool-1-worker-3
ForkJoinPool-1-worker-2
ForkJoinPool-1-worker-1
ForkJoinPool-1-worker-3
ForkJoinPool-1-worker-1
ForkJoinPool-1-worker-4
ForkJoinPool-1-worker-2
*/
```

工作窃取算法允许已经耗尽输入队列中的工作项的线程从其他队列“窃取”工作项。目标是在处理器之间分配工作项，从而最大限度地利用所有可用的处理器来完成计算密集型任务。这项算法也用于 Java 的 fork/join 框架。

10.2 异常捕获

这可能会让你感到惊讶：

```
// lowlevel/SwallowedException.java
import java.util.concurrent.*;

public class SwallowedException {
  public static void main(String[] args)
    throws InterruptedException {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    exec.submit(() -> {
      throw new RuntimeException();
```

```
    });
    exec.shutdown();
  }
}
```

这个程序什么也不输出（然而，如果你用 **execute** 方法替换 `submit()` 方法，你就将会看到异常抛出。这说明在线程中抛出异常是很棘手的，需要特别注意的事情。

你无法捕获到从线程逃逸的异常。一旦异常越过了任务的 `run()` 方法，它就会传递至控制台，除非你采取特殊步骤来捕获此类错误异常。

下面是一个抛出异常的代码，该异常会传递到它的 `run()` 方法之外，而 `main()` 方法会显示运行它时会发生什么：

```
// lowlevel/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
  @Override
  public void run() {
    throw new RuntimeException();
  }
  public static void main(String[] args) {
    ExecutorService es =
      Executors.newCachedThreadPool();
    es.execute(new ExceptionThread());
    es.shutdown();
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "pool-1-thread-1"
java.lang.RuntimeException
        at ExceptionThread.run(ExceptionThread.java:8)
        at java.util.concurrent.ThreadPoolExecutor.runW
orker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Work
er.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
*/
```

输出是 (经过调整一些限定符以适应阅读)：

```
Exception in thread "pool-1-thread-1" RuntimeException
  at ExceptionThread.run(ExceptionThread.java:9)
  at ThreadPoolExecutor.runWorker(...)
  at ThreadPoolExecutor$Worker.run(...)
  at java.lang.Thread.run(Thread.java:745)
```

即使在 `main()` 方法体内包裹 **try-catch** 代码块来捕获异常也不成功：

```java
// lowlevel/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;

public class NaiveExceptionHandling {
  public static void main(String[] args) {
    ExecutorService es =
      Executors.newCachedThreadPool();
    try {
      es.execute(new ExceptionThread());
    } catch(RuntimeException ue) {
      // This statement will NOT execute!
      System.out.println("Exception was handled!");
    } finally {
      es.shutdown();
    }
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "pool-1-thread-1"
java.lang.RuntimeException
        at ExceptionThread.run(ExceptionThread.java:8)
        at java.util.concurrent.ThreadPoolExecutor.runW
orker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Work
er.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
*/
```

这会产生与前一个示例相同的结果: 未捕获异常。

为解决这个问题,需要改变 **Executor**(执行器)生成线程的方式。**Thread.UncaughtExceptionHandler** 是一个添加给每个 **Thread** 对象，用于进行异常处理的接口。

当该线程即将死于未捕获的异常时，将自动调用 `Thread.UncaughtExceptionHandler.uncaughtException()` 方法。为了调用该方法，我们创建一个新的 **ThreadFactory** 类型来让 **Thread.UncaughtExceptionHandler** 对象附加到每个它所新创建的 **Thread**（线程）对象上。我们赋值该工厂对象给 **Executors** 对象的方法，让它的方法来生成新的 **ExecutorService** 对象:

```java
// lowlevel/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
  @Override
  public void run() {
    Thread t = Thread.currentThread();
    System.out.println("run() by " + t.getName());
    System.out.println(
```

```
      "eh = " + t.getUncaughtExceptionHandler());
    throw new RuntimeException();
  }
}

class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
  @Override
  public void uncaughtException(Thread t, Throwable e) {
    System.out.println("caught " + e);
  }
}

class HandlerThreadFactory implements ThreadFactory {
  @Override
  public Thread newThread(Runnable r) {
    System.out.println(this + " creating new Thread");
    Thread t = new Thread(r);
    System.out.println("created " + t);
    t.setUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    System.out.println(
      "eh = " + t.getUncaughtExceptionHandler());
    return t;
  }
}

public class CaptureUncaughtException {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newCachedThreadPool(
        new HandlerThreadFactory());
    exec.execute(new ExceptionThread2());
    exec.shutdown();
  }
}
/* Output:
HandlerThreadFactory@4e25154f creating new Thread
created Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@70dea4e
run() by Thread-0
eh = MyUncaughtExceptionHandler@70dea4e
caught java.lang.RuntimeException
*/
```

额外会在代码中添加跟踪机制，用来验证工厂对象创建的线程是否获得新 **UncaughtExceptionHandler** 。现在未捕获的异常由 **uncaughtException** 方法捕获。

上面的示例根据具体情况来设置处理器。如果你知道你将要在代码中处处使用相同的异常处理器，那么更简单的方式是在 **Thread** 类中设置一个 **static**（静态）字段，并将这个处理器设置为默认的未捕

获异常处理器:

```
// lowlevel/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
  public static void main(String[] args) {
    Thread.setDefaultUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    ExecutorService es =
      Executors.newCachedThreadPool();
    es.execute(new ExceptionThread());
    es.shutdown();
  }
}
/* Output:
caught java.lang.RuntimeException
*/
```

只有在每个线程没有设置异常处理器时候，默认处理器才会被调用。系统会检查线程专有的版本，如果没有，则检查是否线程组中有专有的 `uncaughtException()` 方法；如果都没有，就会调用 **defaultUncaughtExceptionHandler** 方法。

可以将此方法与 **CompletableFuture** 的改进方法进行比较。

10.3 资源共享

你可以将单线程程序看作一个孤独的实体，在你的问题空间中移动并同一时间只做一件事。因为只有一个实体，你永远不会想到两个实体试图同时使用相同资源的问题：问题犹如两个人试图同时停放在同一个空间，同时走过一扇门，甚至同时说话。

通过并发，事情不再孤单，但现在两个或更多任务可能会相互干扰。如果你不阻止这种冲突，你将有两个任务同时尝试访问同一个银行帐户，打印到同一个打印机，调整同一个阀门，等等。

10.3.1 资源竞争

当你启动一个任务来执行某些工作时，可以通过两种不同的方式捕获该工作的结果: 通过副作用或通过返回值。

从编程方式上看，副作用似乎更容易: 你只需使用结果来操作环境中的某些东西。例如，你的任务可能会执行一些计算，然后直接将其结果写入集合。

伴随这种方式的问题是集合通常是共享资源。当运行多个任务时，任何任务都可能同时读写 *共享资源*。这揭示了 *资源竞争*问题，这是处理任务时的主要陷阱之一。

在单线程系统中，你不需要考虑资源竞争，因为你永远不可能同时做多件事。当你有多个任务时，

你就必须始终防止资源竞争。

解决此问题的的一种方法是使用能够应对资源竞争的集合，如果多个任务同时尝试对此类集合进行写入，那么此类集合可以应付该问题。在 Java 并发库中，你将发现许多尝试解决资源竞争问题的类；在本附录中，你将看到其中的一些，但覆盖范围并不全面。

请思考以下的示例，其中一个任务负责生成偶数，其他任务则负责消费这些数字。在这里，消费者任务的唯一工作就是检查偶数的有效性。

我们将定义消费者任务 **EvenChecker** 类，以便在后续示例中可复用。为了将 **EvenChecker** 与我们的各种实验生成器类解耦，我们首先创建名为 **IntGenerator** 的抽象类，它包含 **EvenChecker** 必须知道的最低必要方法：它包含 `next()` 方法，以及可以取消它执行生成的方法。

```
// lowlevel/IntGenerator.java
import java.util.concurrent.atomic.AtomicBoolean;

public abstract class IntGenerator {
  private AtomicBoolean canceled =
    new AtomicBoolean();
  public abstract int next();
  public void cancel() { canceled.set(true); }
  public boolean isCanceled() {
    return canceled.get();
  }
}
```

`cancel()` 方法改变 **AtomicBoolean** 类型的 **canceled** 标志位的状态，而 `isCanceled()` 方法则告诉标志位是否设置。因为 **canceled** 标志位是 **AtomicBoolean** 类型，由于它是原子性的，这意味着分配和值返回等简单操作发生时没有中断的可能性，因此你无法在这些简单操作中看到该字段处于中间状态。你将在本附录的后面部分了解有关原子性和 **Atomic** 类的更多信息

任何 **IntGenerator** 都可以使用下面的 **EvenChecker** 类进行测试:

```
// lowlevel/EvenChecker.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.TimedAbort;

public class EvenChecker implements Runnable {
  private IntGenerator generator;
  private final int id;
  public EvenChecker(IntGenerator generator, int id) {
    this.generator = generator;
    this.id = id;
  }
  @Override
  public void run() {
    while(!generator.isCanceled()) {
```

```
      int val = generator.next();
      if(val % 2 != 0) {
        System.out.println(val + " not even!");
        generator.cancel(); // Cancels all EvenCheckers
      }
    }
  }
  // Test any IntGenerator:
  public static void test(IntGenerator gp, int count) {
    List<CompletableFuture<Void>> checkers =
      IntStream.range(0, count)
        .mapToObj(i -> new EvenChecker(gp, i))
        .map(CompletableFuture::runAsync)
        .collect(Collectors.toList());
    checkers.forEach(CompletableFuture::join);
  }
  // Default value for count:
  public static void test(IntGenerator gp) {
    new TimedAbort(4, "No odd numbers discovered");
    test(gp, 10);
  }
}
```

`test()` 方法开启了许多访问同一个 **IntGenerator** 的 **EvenChecker**。**EvenChecker** 任务们会不断读取和测试与其关联的 **IntGenerator** 对象中的生成值。如果 **IntGenerator** 导致失败，`test()` 方法会报告并返回。

依赖于 **IntGenerator** 对象的所有 **EvenChecker** 任务都会检查它是否已被取消。如果 `generator.isCanceled()` 返回值为 true ，则 `run()` 方法返回。任何 **EvenChecker** 任务都可以在 **IntGenerator** 上调用 `cancel()` ，这会导致使用该 **IntGenerator** 的其他所有 **EvenChecker** 正常关闭。

在本设计中，共享公共资源（**IntGenerator** ）的任务会监视该资源的终止信号。这消除所谓的竞争条件，其中两个或更多的任务竞争响应某个条件并因此冲突或不一致结果的情况。

你必须仔细考虑并防止并发系统失败的所有可能途径。例如，一个任务不能依赖于另一个任务，因为任务关闭的顺序无法得到保证。这里，通过使任务依赖于非任务对象，我们可以消除潜在的竞争条件。

一般来说，我们假设 `test()` 方法最终失败，因为各个 **EvenChecker** 的任务在 **IntGenerator** 处于“不恰当的”状态时，仍能够访问其中的信息。但是，直到 **IntGenerator** 完成许多循环之前，它可能无法检测到问题，具体取决于操作系统的详细信息和其他实现细节。为确保本书的自动构建不会卡住，我们使用 **TimedAbort** 类，在此处定义：

```
// onjava/TimedAbort.java
// Terminate a program after t seconds
package onjava;
import java.util.concurrent.*;
```

```

public class TimedAbort {
  private volatile boolean restart = true;
  public TimedAbort(double t, String msg) {
    CompletableFuture.runAsync(() -> {
      try {
        while(restart) {
          restart = false;
          TimeUnit.MILLISECONDS
            .sleep((int)(1000 * t));
        }
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
      System.out.println(msg);
      System.exit(0);
    });
  }
  public TimedAbort(double t) {
    this(t, "TimedAbort " + t);
  }
  public void restart() { restart = true; }
}
```

我们使用 lambda 表达式创建一个 **Runnable**，该表达式使用 **CompletableFuture** 的 `runAsync()` 静态方法执行。`runAsync()` 方法的值会立即返回。因此，**TimedAbort** 不会保持任何打开的任务，否则已完成任务，但如果它需要太长时间，它仍将终止该任务（**TimedAbort** 有时被称为守护进程）。

TimedAbort 还允许你 `restart()` 方法重启任务，在有某些有用的活动进行时保持程序打开。

我们可以看到正在运行的 **TimedAbort** 示例:

```
// lowlevel/TestAbort.java
import onjava.*;

public class TestAbort {
  public static void main(String[] args) {
    new TimedAbort(1);
    System.out.println("Napping for 4");
    new Nap(4);
  }
}
/* Output:
Napping for 4
TimedAbort 1.0
*/
```

如果你注释掉 **Nap** 创建实列那行，程序执行会立即退出，表明 **TimedAbort** 没有维持程序打开。

我们将看到第一个 **IntGenerator** 示例有一个生成一系列偶数值的 `next()` 方法:

```
// lowlevel/EvenProducer.java
// When threads collide
// {VisuallyInspectOutput}

public class EvenProducer extends IntGenerator {
  private int currentEvenValue = 0;
  @Override
  public int next() {
    ++currentEvenValue; // [1]
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new EvenProducer());
  }
}
/* Output:
419 not even!
425 not even!
423 not even!
421 not even!
417 not even!
*/
```

- [1] 一个任务有可能在另外一个任务执行第一个对 **currentEvenValue** 的自增操作之后，但是没有执行第二个操作之前，调用 `next()` 方法。这将使这个值处于 “不恰当” 的状态。

为了证明这是可能发生的，`EvenChecker.test()` 创建了一组 **EventChecker** 对象，以连续读取 **EvenProducer** 的输出并测试检查每个数值是否都是偶数。如果不是，就会报告错误，而程序也将关闭。

多线程程序的部分问题是，即使存在 bug ，如果失败的可能性很低，程序仍然可以正确显示。

重要的是要注意到自增操作自身需要多个步骤，并且在自增过程中任务可能会被线程机制挂起 - 也就是说，在 Java 中，自增不是原子性的操作。因此，如果不保护任务，即使单纯的自增也不是线程安全的。

该示例程序并不总是在第一次非偶数产生时终止。所有任务都不会立即关闭，这是并发程序的典型特征。

10.3.2 解决资源竞争

前面的示例揭示了当你使用线程时的基本问题：你永远不知道线程哪个时刻运行。想象一下坐在一张桌子上，用叉子，将最后一块食物放在盘子上，当叉子到达时，食物突然消失...仅因为你的线程被挂起而另一个用餐者进来吃了食物了。这就是在编写并发程序时要处理的问题。为了使并发工作有效，你需要某种方式来阻止两个任务访问同一个资源，至少在关键时期是这样。

防止这种冲突的方法就是当资源被一个任务使用时，在其上加锁。第一个访问某项资源的任务必须锁定这项资源，使其他任务在其被解锁之前，就无法访问它，而在其被解锁时候，另一个任务就可以锁定并使用它，以此类推。如果汽车前排座位是受限资源，那么大喊着“冲呀”的孩子就会（在这次旅途过程中）获得该资源的锁。

为了解决线程冲突的问题，基本的并发方案将序列化访问共享资源。这意味着一次只允许一个任务访问共享资源。这通常是通过在访问资源的代码片段周围加上一个子句来实现的，该子句一次只允许一个任务访问这段代码。因为这个子句产生 *互斥*效果，所以这种机制的通常称为是 *mutex* （互斥量）。

考虑一下屋子里的浴室：多个人（即多个由线程驱动的任务）都希望能独立使用浴室（即共享资源）。为了使用浴室，一个人先敲门来看看是否可用。如果没人的话，他就能进入浴室并锁上门。任何其他想使用浴室的任务就会被“阻挡”，因此这些任务就在门口等待，直到浴室是可用的。

当浴室使用完毕，就是时候给其他任务进入，这时比喻就有点不准确了。事实上没有人排队，我们也不知道下一个使用浴室是谁，因为线程调度机制并不是确定性的。相反，就好像在浴室前面有一组被阻止的任务一样，当锁定浴室的任务解锁并出现时，线程调度机制将会决定下一个要进入的任务。

Java 以提供关键字 **synchronized** 的形式，为防止资源冲突提供了内置支持。当任务希望执行被 **synchronized** 关键字保护的代码片段的时候，Java 编译器会生成代码以查看锁是否可用。如果可用，该任务获取锁，执行代码，然后释放锁。

共享资源一般是以对象形式存在的内存片段，但也可以是文件、I/O 端口，或者类似打印机的东西。要控制对共享资源的访问，得先把它包装进一个对象。然后把任何访问该资源的方法标记为 **synchronized** 。如果一个任务在调用其中一个 **synchronized** 方法之内，那么在这个任务从该方法返回之前，其他所有要调用该对象的 **synchronized** 方法的任务都会被阻塞。

通常你会将字段设为 **private**，并仅通过方法访问这些字段。你可用通过使用 **synchronized** 关键字声明方法来防止资源冲突。如下所示：

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

所有对象都自动包含独立的锁（也称为 *monitor*，即监视器）。当你调用对象上任何 **synchronized** 方法，此对象将被加锁，并且该对象上的的其他 **synchronized** 方法调用只有等到前一个方法执行完成并释放了锁之后才能被调用。如果一个任务对对象调用了 `f()` ，对于同一个对象而言，就只能等到 `f()` 调用结束并释放了锁之后，其他任务才能调用 `f()` 和 `g()`。所以，某个特定对象的所有 **synchronized** 方法共享同一个锁，这个锁可以防止多个任务同时写入对象内存。

在使用并发时，将字段设为 **private** 特别重要；否则，**synchronized** 关键字不能阻止其他任务直接访问字段，从而产生资源冲突。

一个线程可以获取对象的锁多次。如果一个方法调用在同一个对象上的第二个方法，而后者又在同一个对象上调用另一个方法，就会发生这种情况。JVM 会跟踪对象被锁定的次数。如果对象已解锁，则其计数为 0 。当一个线程首次获得锁时，计数变为 1 。每次同一线程在同一对象上获取另一个锁时，计数就会自增。显然，只有首先获得锁的线程才允许多次获取多个锁。每当线程离开 **synchronized** 方

法时，计数递减，直到计数变为 0 ，完全释放锁以给其他线程使用。每个类也有一个锁（作为该类的 **Class** 对象的一部分），因此 **synchronized** 静态方法可以在类范围的基础上彼此锁定，不让同时访问静态数据。

你应该什么时候使用同步呢？可以永远 *Brian* 的同步法则[2]。

> 如果你正在写一个变量，它可能接下来被另一个线程读取，或者正在读取一个上一次已经被另一个线程写过的变量，那么你必须使用同步，并且，读写线程都必须用相同的监视器锁同步。

如果在你的类中有超过一个方法在处理临界数据，那么你必须同步所有相关方法。如果只同步其中一个方法，那么其他方法可以忽略对象锁，并且可以不受惩罚地调用。这是很重要的一点：每个访问临界共享资源的方法都必须被同步，否则将不会正确地工作。

10.3.3 同步控制 EventProducer

通过在 **EvenProducer.java** 文件中添加 **synchronized** 关键字，可以防止不希望的线程访问：

```java
// lowlevel/SynchronizedEvenProducer.java
// Simplifying mutexes with the synchronized keyword
import onjava.Nap;

public class
SynchronizedEvenProducer extends IntGenerator {
  private int currentEvenValue = 0;
  @Override
  public synchronized int next() {
    ++currentEvenValue;
    new Nap(0.01); // Cause failure faster
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new SynchronizedEvenProducer());
  }
}
/* Output:
No odd numbers discovered
*/
```

在两个自增操作之间插入 `Nap()` 构造器方法，以提高在 **currentEvenValue** 是奇数的状态时上下文切换的可能性。因为互斥锁可以阻止多个任务同时进入临界区，所有这不会产生失败。第一个进入 `next()` 方法的任务将获得锁，任何试图获取锁的后续任务都将被阻塞，直到第一个任务释放锁。此时，调度机制选择另一个等待锁的任务。通过这种方式，任何时刻只能有一个任务通过互斥锁保护的代码。

[2]引自 Brian Goetz, Java Concurrency in Practice 一书的作者，该书由 Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea 联合著作 (Addison-Wesley 出版社, 2006)。

10.4 volatile 关键字

volatile 可能是 Java 中最微妙和最难用的关键字。幸运的是，在现代 Java 中，你几乎总能避免使用它，如果你确实看到它在代码中使用，你应该保持怀疑态度和怀疑 - 这很有可能代码是过时的，或者编写代码的人不清楚使用它在大体上（或两者都有）易变性（**volatile**）或并发性的后果。

使用 **volatile** 有三个理由。

10.4.1 字分裂

当你的 Java 数据类型足够大（在 Java 中 **long** 和 **double** 类型都是 64 位），写入变量的过程分两步进行，就会发生 *Word tearing* （字分裂）情况。JVM 被允许将 64 位数量的读写作为两个单独的 32 位操作执行[3]，这增加了在读写过程中发生上下文切换的可能性，因此其他任务会看到不正确的结果。这被称为 *Word tearing* （字分裂），因为你可能只看到其中一部分修改后的值。基本上，任务有时可以在第一步之后但在第二步之前读取变量，从而产生垃圾值（对于例如 **boolean** 或 **int** 类型的小变量是没有问题的；任何 **long** 或 **double** 类型则除外）。

在缺乏任何其他保护的情况下，用 **volatile** 修饰符定义一个 **long** 或 **double** 变量，可阻止字分裂情况。然而，如果使用 **synchronized** 或 **java.util.concurrent.atomic** 类之一保护这些变量，则 **volatile** 将被取代。此外，**volatile** 不会影响到增量操作并不是原子操作的事实。

10.4.2 可见性

第二个问题属于 Java 并发的四句格言里第二句格言 “一切都重要” 的部分。你必须假设每个任务拥有自己的处理器，并且每个处理器都有自己的本地内存缓存。该缓存准许处理器允许的更快，因为处理器并不总是需要从比起使用缓存显著花费更多时间的主内存中获取数据。

出现这个问题是因为 Java 尝试尽可能地提高执行效率。缓存的主要目的是避免从主内存中读取数据。当并发时，有时不清楚 Java 什么时候应该将值从主内存刷新到本地缓存 — 而这个问题称为 *缓存一致性*（*cache coherence* ）。

每个线程都可以在处理器缓存中存储变量的本地副本。将字段定义为 **volatile** 可以防止这些编译器优化，这样读写就可以直接进入内存，而不会被缓存。一旦该字段发生写操作，所有任务的读操作都将看到更改。如果一个 **volatile** 字段刚好存储在本地缓存，则会立即将其写入主内存，并且该字段的任何读取都始终发生在主内存中。

volatile 应该在何时适用于变量：

1. 该变量同时被多个任务访问。
2. 这些访问中至少有一个是写操作。
3. 你尝试避免同步（在现代 Java 中，你可以使用高级工具来避免进行同步）。

[3]请注意，在 64 位处理器上可能不会发生这种情况，从而消除了这个问题。

举个例字，如果你使用变量作为停止任务的标志值。那么该变量至少必须声明为 **volatile** （尽管这并不一定能保证这种标志的线程安全）。否则，当一个任务更改标志值时，这些更改可以存储在本地处理器缓存中，而不会刷新到主内存。当另一个任务查看标记值时，它不会看到更改。我更喜欢在 并发编程 中 终止耗时任务 章节中使用 **AtomicBoolean** 类型作为标志值的办法

任务对其自身变量所做的任何写操作都始终对该任务可见，因此，如果只在任务中使用变量，你不需要使其变量声明为 **volatile** 。

如果单个线程对变量写入而其他线程只读取它，你可以放弃该变量声明为 **volatile**。通常，如果你有多个线程对变量写入，**volatile** 无法解决你的问题，并且你必须使用 **synchronized** 来防止竞争条件。这有一个特殊的例外：可以让多个线程对该变量写入，*只要它们不需要先读取它并使用该值创建新值来写入变量*。如果这些多个线程在结果中使用旧值，则会出现竞争条件，因为其余一个线程之一可能会在你的线程进行计算时修改该变量。即使你开始做对了，想象一下在代码修改或维护过程中忘记和引入一个重大变化是多么容易，或者对于不理解问题的不同程序员来说是多么容易（这在 Java 中尤其成问题因为程序员倾向于严重依赖编译时检查来告诉他们，他们的代码是否正确）。

重要的是要理解原子性和可见性是两个不同的概念。在非 **volatile** 变量上的原子操作是不能保证是否将其刷新到主内存。

同步也会让主内存刷新，所以如果一个变量完全由 **synchronized** 的方法或代码段 (或者 **java.util.concurrent.atomic** 库里类型之一) 所保护，则不需要让变量用 **volatile**。

10.4.3 重排与 *Happen-Before* 原则

只要结果不会改变程序表现，Java 可以通过重排指令来优化性能。然而，重排可能会影响本地处理器缓存与主内存交互的方式，从而产生细微的程序 bug 。直到 Java 5 才理解并解决了这个无法阻止重排的问题。现在，**volatile** 关键字可以阻止重排 **volatile** 变量周围的读写指令。这种重排规则称为 *happens before* 担保原则。

这项原则保证在 **volatile** 变量读写之前发生的指令先于它们的读写之前发生。同样，任何跟随 **volatile** 变量之后读写的操作都保证发生在它们的读写之后。例如：

```
// lowlevel/ReOrdering.java

public class ReOrdering implements Runnable {
  int one, two, three, four, five, six;
  volatile int volaTile;
  @Override
  public void run() {
    one = 1;
    two = 2;
    three = 3;
    volaTile = 92;
    int x = four;
    int y = five;
```

```
    int z = six;
  }
}
```

例子中 **one**，**two**，**three** 变量赋值操作就可以被重排，只要它们都发生在 **volatile** 变量写操作之前。同样，只要 **volatile** 变量写操作发生在所有语句之前，**x**，**y**，**z** 语句可以被重排。这种 **volatile**（易变性）操作通常称为 *memory barrier* （内存屏障）。*happens before* 担保原则确保 **volatile** 变量的读写指令不能跨过内存屏障进行重排。

happens before 担保原则还有另一个作用：当线程向一个 **volatile** 变量写入时，在线程写入之前的其他所有变量（包括非 **volatile** 变量）也会刷新到主内存。当线程读取一个 **volatile** 变量时，它也会读取其他所有变量（包括非 **volatile** 变量）与 **volatile** 变量一起刷新到主内存。尽管这是一个重要的特性，它解决了 Java 5 版本之前出现的一些非常狡猾的 bug ，但是你不应该依赖这项特性来“自动”使周围的变量变得易变性（**volatile** ）的。如果你希望变量是易变性（**volatile** ）的，那么维护代码的任何人都应该清楚这一点。

10.4.4 什么时候使用 volatile

对于 Java 早期版本，编写一个证明需要 **volatile** 的示例并不难。如果你进行搜索，你可以找到这样的例子，但是如果你在 Java 8 中尝试这些例子，它们就不起作用了 (我没有找到任何一个)。我努力写这样一个例子，但没什么用。这可能原因是 JVM 或者硬件，或两者都得到了改进。这种效果对现有的应该 **volatile** （易变性）但不 **volatile** 的存储的程序是有益的；对于此类程序，失误发生的频率要低得多，而且问题更难追踪。

如果你尝试使用 **volatile** ，你可能更应该尝试让一个变量线程安全而不是引起同步的成本。因为 **volatile** 使用起来非常微妙和棘手，所以我建议根本不要使用它; 相反，请使用本附录后面介绍的 **java.util.concurrent.atomic** 里面类之一。它们以比同步低得多的成本提供了完全的线程安全性。

如果你正在尝试调试其他人的并发代码，请首先查找使用 **volatile** 的代码并将其替换为 **Atomic** 变量。除非你确定程序员对并发性有很高的理解，否则它们很可能会误用 **volatile** 。

10.5 原子性

在 Java 线程的讨论中，经常反复提交但不正确的知识是：“原子操作不需要同步”。一个 *原子操作*是不能被线程调度机制中断的操作；一旦操作开始，那么它一定可以在可能发生的“上下文切换”之前（切换到其他线程执行）执行完毕。依赖于原子性是很棘手且很危险的，如果你是一个并发编程专家，或者你得到了来自这样的专家的帮助，你才应该使用原子性来代替同步，如果你认为自己足够聪明可以应付这种玩火似的情况，那么请接受下面的测试：

> Goetz 测试：如果你可以编写用于现代微处理器的高性能 JVM，那么就有资格考虑是否可以避免同步[a]。
>
> [a]这个测试的推论是，“如果某人表示线程是容易并且简单的，请确保这个人没有对你的项目做出重要的决策。如果那个人已经做出，那么你就已经陷入麻烦之中了。”

了解原子性是很有用的，并且知道它与其他高级技术一起用于实现一些更加巧妙的 **java.util.concurrent** 库组件。但是要坚决抵制自己依赖它的冲动。

原子性可以应用于除 **long** 和 **double** 之外的所有基本类型之上的“简单操作”。对于读写和写入除 **long** 和 **double** 之外的基本类型变量这样的操作，可以保证它们作为不可分 (原子) 的操作执行。

因为原子操作不能被线程机制中断。专家程序员可以利用这个来编写无锁代码（*lock-free code*），这些代码不需要被同步。但即使这样也过于简单化了。有时候，甚至看起来应该是安全的原子操作，实际上也可能不安全。本书的读者通常不会通过前面提到的 Goetz 测试，因此也就不具备用原子操作来替换同步的能力。尝试着移除同步通常是一种表示不成熟优化的信号，并且会给你带来大量的麻烦，可能不会获得太多或任何的好处。

在多核处理器系统，相对于单核处理器而言，可见性问题远比原子性问题多得多。一个任务所做的修改，即使它们是原子性的，也可能对其他任务不可见（例如，修改只是暂时性存储在本地处理器缓存中），因此不同的任务对应用的状态有不同的视图。另一方面，同步机制强制多核处理器系统上的一个任务做出的修改必须在应用程序中是可见的。如果没有同步机制，那么修改时可见性将无法确认。

什么才属于原子操作时？对于属性中的值做赋值和返回操作通常都是原子性的，但是在 C++ 中，甚至下面的操作都可能是原子性的：

```
i++; // Might be atomic in C++
i += 2; // Might be atomic in C++
```

但是在 C++ 中，这取决于编译器和处理器。你无法编写出依赖于原子性的 C++ 跨平台代码，因为 C++ [4]没有像 Java 那样的一致 *内存模型*（memory model）。

在 Java 中，上面的操作肯定不是原子性的，正如下面的方法产生的 JVM 指令中可以看到的那样：

```
// lowlevel/NotAtomic.java
// {javap -c NotAtomic}
// {VisuallyInspectOutput}

public class NotAtomic {
  int i;
  void f1() { i++; }
  void f2() { i += 3; }
}
/* Output:
Compiled from "NotAtomic.java"
```

[4]这在即将产生的 C++ 的标准中得到了补救。

```
public class NotAtomic {
  int i;

  public NotAtomic();
    Code:
       0: aload_0
       1: invokespecial #1 // Method
java/lang/Object."<init>":()V
       4: return

  void f1();
    Code:
       0: aload_0
       1: dup
       2: getfield      #2 // Field
i:I
       5: iconst_1
       6: iadd
       7: putfield      #2 // Field
i:I
      10: return

  void f2();
    Code:
       0: aload_0
       1: dup
       2: getfield      #2 // Field
i:I
       5: iconst_3
       6: iadd
       7: putfield      #2 // Field
i:I
      10: return
}
*/
```

每条指令都会产生一个 “get” 和 “put”，它们之间还有一些其他指令。因此在获取指令和放置指令之间，另有一个任务可能会修改这个属性，所有，这些操作不是原子性的。

让我们通过定义一个抽象类来测试原子性的概念，这个抽象类的方法是将一个整数类型进行偶数自增，并且 `run()` 不断地调用这个方法:

```java
// lowlevel/IntTestable.java
import java.util.function.*;

public abstract class
IntTestable implements Runnable, IntSupplier {
  abstract void evenIncrement();
  @Override
  public void run() {
```

```
    while(true)
      evenIncrement();
  }
}
```

IntSupplier 是一个带 `getAsInt()` 方法的函数式接口。

现在我们可以创建一个测试，它作为一个独立的任务启动 `run()` 方法，然后获取值来检查它们是否为偶数:

```
// lowlevel/Atomicity.java
import java.util.concurrent.*;
import onjava.TimedAbort;

public class Atomicity {
  public static void test(IntTestable it) {
    new TimedAbort(4, "No failures found");
    CompletableFuture.runAsync(it);
    while(true) {
      int val = it.getAsInt();
      if(val % 2 != 0) {
        System.out.println("failed with: " + val);
        System.exit(0);
      }
    }
  }
}
```

很容易盲目地应用原子性的概念。在这里，`getAsInt()` 似乎是安全的原子性方法:

```
// lowlevel/UnsafeReturn.java
import java.util.function.*;
import java.util.concurrent.*;

public class UnsafeReturn extends IntTestable {
  private int i = 0;
  public int getAsInt() { return i; }
  public synchronized void evenIncrement() {
    i++; i++;
  }
  public static void main(String[] args) {
    Atomicity.test(new UnsafeReturn());
  }
}
/* Output:
failed with: 79
*/
```

但是，`Atomicity.test()` 方法还是出现有非偶数的失败。尽管，返回 **i** 变量确实是原子操作，但是同步缺失允许了在对象处于不稳定的中间状态时读取值。最重要的是，由于 **i** 也不是 **volatile** 变

量，所以存在可见性问题。包括 `getValue()` 和 `evenIncrement()` 都必须同步 (这也顾及到没有使用 **volatile** 修饰的 **i** 变量):

```
// lowlevel/SafeReturn.java
import java.util.function.*;
import java.util.concurrent.*;

public class SafeReturn extends IntTestable {
  private int i = 0;
  public synchronized int getAsInt() { return i; }
  public synchronized void evenIncrement() {
    i++; i++;
  }
  public static void main(String[] args) {
    Atomicity.test(new SafeReturn());
  }
}
/* Output:
No failures found
*/
```

只有并发编程专家有能力去尝试做像前面例子情况的优化；再次强调，请遵循 Brain 的同步法则。

10.5.1 Josh 的序列号

作为第二个示例，考虑某些更简单的东西：创建一个产生序列号的类，灵感启发于 Joshua Bloch 的 *Effective Java Programming Language Guide* (Addison-Wesley 出版社, 2001) 第 190 页。每次调用 `nextSerialNumber()` 都必须返回唯一值。

```
// lowlevel/SerialNumbers.java

public class SerialNumbers {
  private volatile int serialNumber = 0;
  public int nextSerialNumber() {
    return serialNumber++; // Not thread-safe
  }
}
```

SerialNumbers 是你可以想象到最简单的类，如果你具备 C++ 或者其他底层的知识背景，你可能会认为自增是一个原子操作，因为 C++ 的自增操作通常被单个微处理器指令所实现（尽管不是以任何一致，可靠，跨平台的方式）。但是，正如前面所提到的，Java 自增操作不是原子性的，并且操作同时涉及读取和写入，因此即使在这样一个简单的操作中，也存在有线程问题的空间。

我们在这里加入 volatile ，看看它是否有帮助。然而，真正的问题是 `nextSerialNumber()` 方法在不进行线程同步的情况下访问共享的可变变量值。

为了测试 **SerialNumbers**，我们将创建一个不会耗尽内存的集合，假如需要很长时间来检测问题。这里展示的 **CircularSet** 重用了存储 **int** 变量的内存，最终新值会覆盖旧值 (复制的速度通常发生足够快，你也可以使用 **java.util.Set** 来代替):

```java
// lowlevel/CircularSet.java
// Reuses storage so we don't run out of memory
import java.util.*;

public class CircularSet {
  private int[] array;
  private int size;
  private int index = 0;
  public CircularSet(int size) {
    this.size = size;
    array = new int[size];
    // Initialize to a value not produced
    // by SerialNumbers:
    Arrays.fill(array, -1);
  }
  public synchronized void add(int i) {
    array[index] = i;
    // Wrap index and write over old elements:
    index = ++index % size;
  }
  public synchronized boolean contains(int val) {
    for(int i = 0; i < size; i++)
      if(array[i] == val) return true;
    return false;
  }
}
```

`add()` 和 `contains()` 方法是线程同步的，以防止线程冲突。The add() and contains() methods are synchronized to prevent thread collisions.

SerialNumberChecker 类包含一个存储最近序列号的 **CircularSet** 变量，以及一个填充数值给 **CircularSet** 和确保它里面的序列号是唯一的 `run()` 方法。

```java
// lowlevel/SerialNumberChecker.java
// Test SerialNumbers implementations for thread-safety
import java.util.concurrent.*;
import onjava.Nap;

public class SerialNumberChecker implements Runnable {
  private CircularSet serials = new CircularSet(1000);
  private SerialNumbers producer;
  public SerialNumberChecker(SerialNumbers producer) {
    this.producer = producer;
  }
  @Override
```

```
  public void run() {
    while(true) {
      int serial = producer.nextSerialNumber();
      if(serials.contains(serial)) {
        System.out.println("Duplicate: " + serial);
        System.exit(0);
      }
      serials.add(serial);
    }
  }
  static void test(SerialNumbers producer) {
    for(int i = 0; i < 10; i++)
      CompletableFuture.runAsync(
        new SerialNumberChecker(producer));
    new Nap(4, "No duplicates detected");
  }
}
```

`test()` 方法创建多个任务来竞争单独的 **SerialNumbers** 对象。这时参于竞争的的 SerialNumberChecker 任务们就会试图生成重复的序列号（这情况在具有更多内核处理器的机器上发生得更快)。

当我们测试基本的 **SerialNumbers** 类，它会失败（产生重复序列号)：

```
// lowlevel/SerialNumberTest.java

public class SerialNumberTest {
  public static void main(String[] args) {
    SerialNumberChecker.test(new SerialNumbers());
  }
}
/* Output:
Duplicate: 148044
*/
```

volatile 在这里没有帮助。要解决这个问题，将 **synchronized** 关键字添加到 `nextSerialNumber()` 方法:

```
// lowlevel/SynchronizedSerialNumbers.java

public class
SynchronizedSerialNumbers extends SerialNumbers {
  private int serialNumber = 0;
  public synchronized int nextSerialNumber() {
    return serialNumber++;
  }
  public static void main(String[] args) {
    SerialNumberChecker.test(
      new SynchronizedSerialNumbers());
  }
```

```
}
/* Output:
No duplicates detected
*/
```

volatile 不再是必需的，因为 **synchronized** 关键字保证了 volatile （易变性）的特性。

读取和赋值原语应该是安全的原子操作。然后，正如在 **UnsafeReturn.java** 中所看到，使用原子操作访问处于不稳定中间状态的对象仍然很容易。对这个问题做出假设既棘手又危险。最明智的做法就是遵循 Brian 的同步规则 (如果可以，首先不要共享变量)。

10.5.2 原子类

Java 5 引入了专用的原子变量类，例如 **AtomicInteger**、**AtomicLong**、**AtomicReference** 等。这些提供了原子性升级。这些快速、无锁的操作，它们是利用了现代处理器上可用的机器级原子性。

下面，我们可以使用 **atomicinteger** 重写 **unsafereturn.java** 示例：

```
// lowlevel/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import onjava.*;

public class AtomicIntegerTest extends IntTestable {
  private AtomicInteger i = new AtomicInteger(0);
  public int getAsInt() { return i.get(); }
  public void evenIncrement() { i.addAndGet(2); }
  public static void main(String[] args) {
    Atomicity.test(new AtomicIntegerTest());
  }
}
/* Output:
No failures found
*/
```

现在，我们通过使用 **AtomicInteger** 来消除了 **synchronized** 关键字。

下面使用 **AtomicInteger** 来重写 **SynchronizedEvenProducer.java** 示例：

```
// lowlevel/AtomicEvenProducer.java
// Atomic classes: occasionally useful in regular code
import java.util.concurrent.atomic.*;

public class AtomicEvenProducer extends IntGenerator {
  private AtomicInteger currentEvenValue =
    new AtomicInteger(0);
  @Override
  public int next() {
    return currentEvenValue.addAndGet(2);
```

```
  }
  public static void main(String[] args) {
    EvenChecker.test(new AtomicEvenProducer());
  }
}
/* Output:
No odd numbers discovered
*/
```

再次，使用 **AtomicInteger** 消除了对所有其他同步方式的需要。

下面是一个使用 **AtomicInteger** 实现 **SerialNumbers** 的例子:

```
// lowlevel/AtomicSerialNumbers.java
import java.util.concurrent.atomic.*;

public class
AtomicSerialNumbers extends SerialNumbers {
  private AtomicInteger serialNumber =
    new AtomicInteger();
  public synchronized int nextSerialNumber() {
    return serialNumber.getAndIncrement();
  }
  public static void main(String[] args) {
    SerialNumberChecker.test(
      new AtomicSerialNumbers());
  }
}
/* Output:
No duplicates detected
*/
```

这些都是对单一字段的简单示例；当你创建更复杂的类时，你必须确定哪些字段需要保护，在某些情况下，你可能仍然最后在方法上使用 **synchronized** 关键字。

10.6 临界区

有时，你只是想防止多线程访问方法中的部分代码，而不是整个方法。要隔离的代码部分称为临界区，它使用我们用于保护整个方法相同的 **synchronized** 关键字创建，但使用不同的语法。语法如下，**synchronized** 指定某个对象作为锁用于同步控制花括号内的代码:

```
synchronized(syncObject) {
  // This code can be accessed
  // by only one task at a time
}
```

这也被称为 *同步控制块*（synchronized block）；在进入此段代码前，必须得到 **syncObject** 对象的锁。如果一些其他任务已经得到这个锁，那么就得等到锁被释放以后，才能进入临界区。当发生这

种情况时，尝试获取该锁的任务就会挂起。线程调度会定期回来并检查锁是否已经释放；如果释放了锁则唤醒任务。

使用同步控制块而不是同步控制整个方法的主要动机是性能（有时，算法确实聪明，但还是要特别警惕来自并发性问题上的聪明）。下面的示例演示了同步控制代码块而不是整个方法可以使方法更容易被其他任务访问。该示例会统计成功访问 `method()` 的计数并且发起一些任务来尝试竞争调用 `method()` 方法。

```
// lowlevel/SynchronizedComparison.java
// speeds up access.
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import onjava.Nap;

abstract class Guarded {
  AtomicLong callCount = new AtomicLong();
  public abstract void method();
  @Override
  public String toString() {
    return getClass().getSimpleName() +
      ": " + callCount.get();
  }
}

class SynchronizedMethod extends Guarded {
  public synchronized void method() {
    new Nap(0.01);
    callCount.incrementAndGet();
  }
}

class CriticalSection extends Guarded {
  public void method() {
    new Nap(0.01);
    synchronized(this) {
      callCount.incrementAndGet();
    }
  }
}

class Caller implements Runnable {
  private Guarded g;
  Caller(Guarded g) { this.g = g; }
  private AtomicLong successfulCalls =
    new AtomicLong();
  private AtomicBoolean stop =
    new AtomicBoolean(false);
```

```
  @Override
  public void run() {
    new Timer().schedule(new TimerTask() {
      public void run() { stop.set(true); }
    }, 2500);
    while(!stop.get()) {
      g.method();
      successfulCalls.getAndIncrement();
    }
    System.out.println(
      "-> " + successfulCalls.get());
  }
}

public class SynchronizedComparison {
  static void test(Guarded g) {
    List<CompletableFuture<Void>> callers =
      Stream.of(
        new Caller(g),
        new Caller(g),
        new Caller(g),
        new Caller(g))
        .map(CompletableFuture::runAsync)
        .collect(Collectors.toList());
    callers.forEach(CompletableFuture::join);
    System.out.println(g);
  }
  public static void main(String[] args) {
    test(new CriticalSection());
    test(new SynchronizedMethod());
  }
}
/* Output:
-> 243
-> 243
-> 243
-> 243
CriticalSection: 972
-> 69
-> 61
-> 83
-> 36
SynchronizedMethod: 249
*/
```

Guarded 类负责跟踪 **callCount** 中成功调用 `method()` 的次数。**SynchronizedMethod** 的方式是同步控制整个 `method` 方法，而 **CriticalSection** 的方式是使用同步控制块来仅同步 `method` 方法的一部分代码。这样，耗时的 **Nap** 对象可以被排除到同步控制块外。输出会显示 **CriticalSection** 中可用的 `method()` 有多少。

请记住，使用同步控制块是有风险；它要求你确切知道同步控制块外的非同步代码是实际上要线程安全的。

Caller 是尝试在给定的时间周期内尽可能多地调用 `method()` 方法（并报告调用次数）的任务。为了构建这个时间周期，我们会使用虽然有点过时但仍然可以很好地工作的 **java.util.Timer** 类。此类接收一个 **TimerTask** 参数，但该参数并不是函数式接口，所以我们不能使用 **lambda** 表达式，必须显式创建该类对象（在这种情况下，使用匿名内部类）。当超时的时候，定时对象将设置 **AtomicBoolean** 类型的 **stop** 字段为 true ，这样循环就会退出。

`test()` 方法接收一个 **Guarded** 类对象并创建四个 **Caller** 任务。所有这些任务都添加到同一个 **Guarded** 对象上，因此它们竞争来获取使用 `method()` 方法的锁。

你通常会看到从一次运行到下一次运行的输出变化。结果表明，**CriticalSection** 方式比起 **SynchronizedMethod** 方式允许更多地访问 `method()` 方法。这通常是使用 **synchronized** 块取代同步控制整个方法的原因：允许其他任务更多访问 (只要这样做是线程安全的)。

10.6.1 在其他对象上同步

synchronized 块必须给定一个在其上进行同步的对象。并且最合理的方式是，使用其方法正在被调用的当前对象：**synchronized(this)**，这正是前面示例中 **CriticalSection** 采取的方式。在这种方式中，当 **synchronized** 块获得锁的时候，那么该对象其他的 **synchronized** 方法和临界区就不能被调用了。因此，在进行同步时，临界区的作用是减小同步的范围。

有时必须在另一个对象上同步，但是如果你要这样做，就必须确保所有相关的任务都是在同一个任务上同步的。下面的示例演示了当对象中的方法在不同的锁上同步时，两个任务可以同时进入同一对象：

```
// lowlevel/SyncOnObject.java
// Synchronizing on another object
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.Nap;

class DualSynch {
  ConcurrentLinkedQueue<String> trace =
    new ConcurrentLinkedQueue<>();
  public synchronized void f(boolean nap) {
    for(int i = 0; i < 5; i++) {
      trace.add(String.format("f() " + i));
      if(nap) new Nap(0.01);
    }
  }
  private Object syncObject = new Object();
  public void g(boolean nap) {
    synchronized(syncObject) {
```

```
      for(int i = 0; i < 5; i++) {
        trace.add(String.format("g() " + i));
        if(nap) new Nap(0.01);
      }
    }
  }
}

public class SyncOnObject {
  static void test(boolean fNap, boolean gNap) {
    DualSynch ds = new DualSynch();
    List<CompletableFuture<Void>> cfs =
      Arrays.stream(new Runnable[] {
        () -> ds.f(fNap), () -> ds.g(gNap) })
        .map(CompletableFuture::runAsync)
        .collect(Collectors.toList());
    cfs.forEach(CompletableFuture::join);
    ds.trace.forEach(System.out::println);
  }
  public static void main(String[] args) {
    test(true, false);
    System.out.println("****");
    test(false, true);
  }
}
/* Output:
f() 0
g() 0
g() 1
g() 2
g() 3
g() 4
f() 1
f() 2
f() 3
f() 4
****
f() 0
g() 0
f() 1
f() 2
f() 3
f() 4
g() 1
g() 2
g() 3
g() 4
*/
```

`DualSync.f()` 方法（通过同步整个方法）在 **this** 上同步，而 `g()` 方法有一个在 **syncObject**

上同步的 **synchronized** 块。因此，这两个同步是互相独立的。在 `test()` 方法中运行的两个调用 `f()` 和 `g()` 方法的独立任务演示了这一点。**fNap** 和 **gNap** 标志变量分别指示 `f()` 和 `g()` 是否应该在其 **for** 循环中调用 `Nap()` 方法。例如，当 f() 线程休眠时，该线程继续持有它的锁，但是你可以看到这并不阻止调用 `g()`，反之亦然。

10.6.2 使用显式锁对象

java.util.concurrent 库包含在 **java.util.concurrent.locks** 中定义的显示互斥锁机制。必须显式地创建，锁定和解锁 **Lock** 对象，因此它产出的代码没有内置 **synchronized** 关键字那么优雅。然而，它在解决某些类型的问题时更加灵活。下面是使用显式 **Lock** 对象重写 **SynchronizedEvenProducer.java** 代码：

```
// lowlevel/MutexEvenProducer.java
// Preventing thread collisions with mutexes
import java.util.concurrent.locks.*;
import onjava.Nap;

public class MutexEvenProducer extends IntGenerator {
  private int currentEvenValue = 0;
  private Lock lock = new ReentrantLock();
  @Override
  public int next() {
    lock.lock();
    try {
      ++currentEvenValue;
      new Nap(0.01); // Cause failure faster
      ++currentEvenValue;
      return currentEvenValue;
    } finally {
      lock.unlock();
    }
  }
  public static void main(String[] args) {
    EvenChecker.test(new MutexEvenProducer());
  }
}
/*
No odd numbers discovered
*/
```

MutexEvenProducer 添加一个名为 **lock** 的互斥锁并在 `next()` 中使用 `lock()` 和 `unlock()` 方法创建一个临界区。当你使用 **Lock** 对象时，使用下面显示的习惯用法很重要：在调用 `Lock()` 之后，你必须放置 **try-finally** 语句，该语句在 **finally** 子句中带有 `unlock()` 方法 - 这是确保锁总是被释放的惟一方法。注意，**return** 语句必须出现在 **try** 子句中，以确保 **unlock()** 不会过早发生并将数据暴露给第二个任务。

尽管 **try-finally** 比起使用 **synchronized** 关键字需要用得更多代码，但它也代表了显式锁对象的优势之一。如果使用 **synchronized** 关键字失败，就会抛出异常，但是你没有机会进行任何清理以保持系统处于良好状态。而使用显式锁对象，可以使用 **finally** 子句在系统中维护适当的状态。

一般来说，当你使用 **synchronized** 的时候，需要编写的代码更少，并且用户出错的机会也大大减少，因此通常只在解决特殊问题时使用显式锁对象。例如，使用 **synchronized** 关键字，你不能尝试获得锁并让其失败，或者你在一段时间内尝试获得锁，然后放弃 - 为此，你必须使用这个并发库。

```
// lowlevel/AttemptLocking.java
// Locks in the concurrent library allow you
// to give up on trying to acquire a lock
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import onjava.Nap;

public class AttemptLocking {
  private ReentrantLock lock = new ReentrantLock();
  public void untimed() {
    boolean captured = lock.tryLock();
    try {
      System.out.println("tryLock(): " + captured);
    } finally {
      if(captured)
        lock.unlock();
    }
  }
  public void timed() {
    boolean captured = false;
    try {
      captured = lock.tryLock(2, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    try {
      System.out.println(
        "tryLock(2, TimeUnit.SECONDS): " + captured);
    } finally {
      if(captured)
        lock.unlock();
    }
  }
  public static void main(String[] args) {
    final AttemptLocking al = new AttemptLocking();
    al.untimed(); // True -- lock is available
    al.timed();   // True -- lock is available
    // Now create a second task to grab the lock:
    CompletableFuture.runAsync( () -> {
        al.lock.lock();
        System.out.println("acquired");
```

```
    });
    new Nap(0.1);  // Give the second task a chance
    al.untimed(); // False -- lock grabbed by task
    al.timed();   // False -- lock grabbed by task
  }
}
/* Output:
tryLock(): true
tryLock(2, TimeUnit.SECONDS): true
acquired
tryLock(): false
tryLock(2, TimeUnit.SECONDS): false
*/
```

ReentrantLock 可以尝试或者放弃获取锁，因此如果某些任务已经拥有锁，你可以决定放弃并执行其他操作，而不是一直等到锁释放，就像 `untimed()` 方法那样。而在 `timed()` 方法中，则尝试获取可能在 2 秒后没成功而放弃的锁。在 `main()` 方法中，一个单独的线程被匿名类所创建，并且它会获得锁，因此让 `untimed()` 和 `timed()` 方法有东西可以去竞争。

显式锁比起内置同步锁提供更细粒度的加锁和解锁控制。这对于实现专门的同步并发结构，比如用于遍历链表节点的 *交替锁* (*hand-over-hand locking*) ，也称为 *锁耦合*（*lock coupling* ）- 该遍历代码要求必须在当前节点的解锁之前捕获下一个节点的锁。

10.7 库组件

java.util.concurrent 库提供大量旨在解决并发问题的类，可以帮助你生成更简单，更鲁棒的并发程序。但请注意，这些工具是比起并行流和 **CompletableFuture** 更底层的机制。

在本节中，我们将看一些使用不同组件的示例，然后讨论一下 *lock-free*（无锁）库组件是如何工作的。

10.7.1 DelayQueue

这是一个无界阻塞队列（**BlockingQueue** ），用于放置实现了 **Delayed** 接口的对象，其中的对象只能在其到期时才能从队列中取走。这种队列是有序的，因此队首对象的延迟到期的时间最长。如果没有任何延迟到期，那么就不会有队首元素，并且 `poll()` 将返回 **null**（正因为这样，你不能将 **null** 放置到这种队列中）。

下面是一个示例，其中的 **Delayed** 对象自身就是任务，而 **DelayedTaskConsumer** 将最“紧急”的任务（到期时间最长的任务）从队列中取出，然后运行它。注意的是这样 **DelayQueue** 就成为了优先级队列的一种变体。

```
// lowlevel/DelayQueueDemo.java
```

```java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import static java.util.concurrent.TimeUnit.*;

class DelayedTask implements Runnable, Delayed {
  private static int counter = 0;
  private final int id = counter++;
  private final int delta;
  private final long trigger;
  protected static List<DelayedTask> sequence =
    new ArrayList<>();
  DelayedTask(int delayInMilliseconds) {
    delta = delayInMilliseconds;
    trigger = System.nanoTime() +
      NANOSECONDS.convert(delta, MILLISECONDS);
    sequence.add(this);
  }
  @Override
  public long getDelay(TimeUnit unit) {
    return unit.convert(
      trigger - System.nanoTime(), NANOSECONDS);
  }
  @Override
  public int compareTo(Delayed arg) {
    DelayedTask that = (DelayedTask)arg;
    if(trigger < that.trigger) return -1;
    if(trigger > that.trigger) return 1;
    return 0;
  }
  @Override
  public void run() {
    System.out.print(this + " ");
  }
  @Override
  public String toString() {
    return
      String.format("[%d] Task %d", delta, id);
  }
  public String summary() {
    return String.format("(%d:%d)", id, delta);
  }
  public static class EndTask extends DelayedTask {
    EndTask(int delay) { super(delay); }
    @Override
    public void run() {
      sequence.forEach(dt ->
        System.out.println(dt.summary()));
    }
  }
}
```

```

public class DelayQueueDemo {
  public static void
  main(String[] args) throws Exception {
    DelayQueue<DelayedTask> tasks =
      Stream.concat( // Random delays:
        new Random(47).ints(20, 0, 4000)
          .mapToObj(DelayedTask::new),
        // Add the summarizing task:
        Stream.of(new DelayedTask.EndTask(4000)))
      .collect(Collectors
        .toCollection(DelayQueue::new));
    while(tasks.size() > 0)
      tasks.take().run();
  }
}
/* Output:
[128] Task 12 [429] Task 6 [551] Task 13 [555] Task 2
[693] Task 3 [809] Task 15 [961] Task 5 [1258] Task 1
[1258] Task 20 [1520] Task 19 [1861] Task 4 [1998] Task
17 [2200] Task 8 [2207] Task 10 [2288] Task 11 [2522]
Task 9 [2589] Task 14 [2861] Task 18 [2868] Task 7
[3278] Task 16 (0:4000)
(1:1258)
(2:555)
(3:693)
(4:1861)
(5:961)
(6:429)
(7:2868)
(8:2200)
(9:2522)
(10:2207)
(11:2288)
(12:128)
(13:551)
(14:2589)
(15:809)
(16:3278)
(17:1998)
(18:2861)
(19:1520)
(20:1258)
*/
```

DelayedTask 包含一个称为 **sequence** 的 **List<DelayedTask>**，它保存了任务被创建的顺序，因此我们可以看到排序是按照实际发生的顺序执行的。

Delay 接口有一个方法，`getDelay()`，该方法用来告知延迟到期有多长时间，或者延迟在多长时间之前已经到期了。这个方法强制我们去使用 **TimeUnit** 类，因为这就是参数类型。这会产生一个非

常方便的类，因为你可以很容易地转换单位而无需作任何声明。例如，**delta** 的值是以毫秒为单位存储的，但是 `System.nanoTime()` 产生的时间则是以纳秒为单位的。你可以转换 **delta** 的值，方法是声明它的单位以及你希望以什么单位来表示，就像下面这样：

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

在 `getDelay()` 中，所希望的单位是作为 **unit** 参数传递进来的，你使用它将当前时间与触发时间之间的差转换为调用者要求的单位，而无需知道这些单位是什么（这是*策略*设计模式的一个简单示例，在这种模式中，算法的一部分是作为参数传递进来的）。

为了排序，**Delayed** 接口还继承了 **Comparable** 接口，因此必须实现 `compareTo()` ，使其可以产生合理的比较。

从输出中可以看到，任务创建的顺序对执行顺序没有任何影响 - 相反，任务是按照所期望的延迟顺序所执行的。

10.7.2 PriorityBlockingQueue

这是一个很基础的优先级队列，它具有可阻塞的读取操作。在下面的示例中，**Prioritized** 对象会被赋予优先级编号。几个 **Producer** 任务的实例会插入 **Prioritized** 对象到 **PriorityBlockingQueue** 中，但插入之间会有随机延时。然后，单个 **Consumer** 任务在执行 `take()` 时会显示多个选项，**PriorityBlockingQueue** 会将当前具有最高优先级的 **Prioritized** 对象提供给它。

在 **Prioritized** 中的静态变量 **counter** 是 **AtomicInteger** 类型。这是必要的，因为有多个 **Producer** 并行运行；如果不是 **AtomicInteger** 类型，你将会看到重复的 **id** 号。这个问题在 并发编程 的 构造函数非线程安全 一节中讨论过。

```
// lowlevel/PriorityBlockingQueueDemo.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import onjava.Nap;

class Prioritized implements Comparable<Prioritized>  {
  private static AtomicInteger counter =
    new AtomicInteger();
  private final int id = counter.getAndIncrement();
  private final int priority;
  private static List<Prioritized> sequence =
    new CopyOnWriteArrayList<>();
  Prioritized(int priority) {
    this.priority = priority;
    sequence.add(this);
  }
  @Override
```

```
  public int compareTo(Prioritized arg) {
    return priority < arg.priority ? 1 :
      (priority > arg.priority ? -1 : 0);
  }
  @Override
  public String toString() {
    return String.format(
      "[%d] Prioritized %d", priority, id);
  }
  public void displaySequence() {
    int count = 0;
    for(Prioritized pt : sequence) {
      System.out.printf("(%d:%d)", pt.id, pt.priority);
      if(++count % 5 == 0)
        System.out.println();
    }
  }
  public static class EndSentinel extends Prioritized {
    EndSentinel() { super(-1); }
  }
}

class Producer implements Runnable {
  private static AtomicInteger seed =
    new AtomicInteger(47);
  private SplittableRandom rand =
    new SplittableRandom(seed.getAndAdd(10));
  private Queue<Prioritized> queue;
  Producer(Queue<Prioritized> q) {
    queue = q;
  }
  @Override
  public void run() {
    rand.ints(10, 0, 20)
      .mapToObj(Prioritized::new)
      .peek(p -> new Nap(rand.nextDouble() / 10))
      .forEach(p -> queue.add(p));
    queue.add(new Prioritized.EndSentinel());
  }
}

class Consumer implements Runnable {
  private PriorityBlockingQueue<Prioritized> q;
  private SplittableRandom rand =
    new SplittableRandom(47);
  Consumer(PriorityBlockingQueue<Prioritized> q) {
    this.q = q;
  }
  @Override
  public void run() {
    while(true) {
```

```
      try {
        Prioritized pt = q.take();
        System.out.println(pt);
        if(pt instanceof Prioritized.EndSentinel) {
          pt.displaySequence();
          break;
        }
        new Nap(rand.nextDouble() / 10);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
}

public class PriorityBlockingQueueDemo {
  public static void main(String[] args) {
    PriorityBlockingQueue<Prioritized> queue =
      new PriorityBlockingQueue<>();
    CompletableFuture.runAsync(new Producer(queue));
    CompletableFuture.runAsync(new Producer(queue));
    CompletableFuture.runAsync(new Producer(queue));
    CompletableFuture.runAsync(new Consumer(queue))
      .join();
  }
}
/* Output:
[15] Prioritized 2
[17] Prioritized 1
[17] Prioritized 5
[16] Prioritized 6
[14] Prioritized 9
[12] Prioritized 0
[11] Prioritized 4
[11] Prioritized 12
[13] Prioritized 13
[12] Prioritized 16
[14] Prioritized 18
[15] Prioritized 23
[18] Prioritized 26
[16] Prioritized 29
[12] Prioritized 17
[11] Prioritized 30
[11] Prioritized 24
[10] Prioritized 15
[10] Prioritized 22
[8] Prioritized 25
[8] Prioritized 11
[8] Prioritized 10
[6] Prioritized 31
[3] Prioritized 7
```

```
[2] Prioritized 20
[1] Prioritized 3
[0] Prioritized 19
[0] Prioritized 8
[0] Prioritized 14
[0] Prioritized 21
[-1] Prioritized 28
(0:12)(2:15)(1:17)(3:1)(4:11)
(5:17)(6:16)(7:3)(8:0)(9:14)
(10:8)(11:8)(12:11)(13:13)(14:0)
(15:10)(16:12)(17:12)(18:14)(19:0)
(20:2)(21:0)(22:10)(23:15)(24:11)
(25:8)(26:18)(27:-1)(28:-1)(29:16)
(30:11)(31:6)(32:-1)
*/
```

与前面的示例一样，**Prioritized** 对象的创建顺序在 **sequence** 的 **list** 对象上所记入，以便与实际执行顺序进行比较。**EndSentinel** 是用于告知 **Consumer** 对象关闭的特殊类型。

Producer 使用 **AtomicInteger** 变量为 **SplittableRandom** 设置随机生成种子，以便不同的 **Producer** 生成不同的队列。这是必需的，因为多个生产者并行创建，如果不是这样，创建过程并不会是线程安全的。

Producer 和 **Consumer** 通过 **PriorityBlockingQueue** 相互连接。因为阻塞队列的性质提供了所有必要的同步，因为阻塞队列的性质提供了所有必要的同步，请注意，显式同步是并不需要的 —— 从队列中读取数据时，你不用考虑队列中是否有任何元素，因为队列在没有元素时将阻塞读取。

10.7.3 无锁集合

集合 章节强调集合是基本的编程工具，这也要求包含并发性。因此，早期的集合比如 **Vector** 和 **Hashtable** 有许多使用 **synchronized** 机制的方法。当这些集合不是在多线程应用中使用时，这就导致了不可接收的开销。在 Java 1.2 版本中，新的集合库是非同步的，而给 **Collection** 类赋予了各种 **static synchronized** 修饰的方法来同步不同的集合类型。虽然这是一个改进，因为它让你可以选择是否对集合使用同步，但是开销仍然基于同步锁定。Java 5 版本添加新的集合类型，专门用于增加线程安全性能，使用巧妙的技术来消除锁定。

无锁集合有一个有趣的特性：只要读取者仅能看到已完成修改的结果，对集合的修改就可以同时发生在读取发生时。这是通过一些策略实现的。为了让你了解它们是如何工作的，我们来看看其中的一些。

复制策略 使用“复制”策略，修改是在数据结构一部分的单独副本（或有时是整个数据的副本）上进行的，并且在整个修改过程期间这个副本是不可见的。仅当修改完成时，修改后的结构才与“主”数据结构安全地交换，然后读取者才会看到修改。

在 **CopyOnWriteArrayList**，写入操作会复制整个底层数组。保留原来的数组，以便在修改复制的数组时可以线程安全地进行读取。当修改完成后，原子操作会将其交换到新数组中，以便新的读取操作能够看到新数组内容。**CopyOnWriteArrayList** 的其中一个好处是，当多个迭代器遍历和修改列表时，它不会抛出 **ConcurrentModificationException** 异常，因此你不用就像过去必须做的那样，编写特殊的代码来防止此类异常。

CopyOnWriteArraySet 使用 **CopyOnWriteArrayList** 来实现其无锁行为。

ConcurrentHashMap 和 **ConcurrentLinkedQueue** 使用类似的技术来允许并发读写，但是只复制和修改集合的一部分，而不是整个集合。然而，读取者仍然不会看到任何不完整的修改。**ConcurrentHashMap 不会抛出 concurrentmodificationexception** 异常。

比较并交换 (CAS) 在比较并交换 (CAS) 中，你从内存中获取一个值，并在计算新值时保留原始值。然后使用 CAS 指令，它将原始值与当前内存中的值进行比较，如果这两个值是相等的，则将内存中的旧值替换为计算新值的结果，所有操作都在一个原子操作中完成。如果原始值比较失败，则不会进行交换，因为这意味着另一个线程同时修改了内存。在这种情况下，你的代码必须再次尝试，获取一个新的原始值并重复该操作。

如果内存仅轻量竞争，CAS 操作几乎总是在没有重复尝试的情况下完成，因此它非常快。相反，**synchronized** 操作需要考虑每次获取和释放锁的成本，这要昂贵得多，而且没有额外的好处。随着内存竞争的增加，使用 CAS 的操作会变慢，因为它必须更频繁地重复自己的操作，但这是对更多资源竞争的动态响应。这确实是一种优雅的方法。

最重要的是，许多现代处理器的汇编语言中都有一条 CAS 指令，并且也被 JVM 中的 CAS 操作 (例如 **Atomic** 类中的操作) 所使用。CAS 指令在硬件层面中是原子性的，并且与你所期望的操作一样快。

10.8 本章小结

本附录主要是为了让你在遇到底层并发代码时能对此有一定的了解，尽管本文还远没对这个主题进行全面的讨论。为此，你需要先从阅读由 Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea (Addison-Wesley 出版社, 2006) 所著作的 *Java Concurrency in Practice*（国内译名：Java 并发编程实战）开始了解。理想情况下，这本书会完全吓跑你在 Java 中尝试去编写底层并发代码。如果没有，那么你几乎肯定患上了达克效应 (DunningKruger Effect)，这是一种认知偏差，“你知道的越少，对自己的能力就越有信心”。请记住，当前的语言设计人员仍然在清理早期语言设计人员过于自信造成的混乱 (例如，查看 Thread 类中有多少方法被弃用，而 volatile 直到 Java 5 才正确工作)。

以下是并发编程的步骤:

1. 不要使用它。想一些其他方法来使你写的程序变的更快。

2. 如果你必须使用它，请使用在 并发编程 - parallel Streams and CompletableFutures 中展示的现代高级工具。
3. 不要在任务间共享变量，在任务之间必须传递的任何信息都应该使用 Java.util.concurrent 库中的并发数据结构。
4. 如果必须在任务之间共享变量，请使用 java.util.concurrent.atomic 里面其中一种类型，或在任何直接或间接访问这些变量的方法上应用 synchronized。当你不这样做时，很容易被愚弄，以为你已经把所有东西都包括在内。说真的，尝试使用步骤 3。
5. 如果步骤 4 产生的结果太慢，你可以尝试使用 volatile 或其他技术来调整代码，但是如果你正在阅读本书并认为你已经准备好尝试这些方法，那么你就超出了你的深度。返回步骤＃ 1。

通常可以只使用 java.util.concurrent 库组件来编写并发程序，完全避免来自应用 volatile 和 synchronized 的挑战。注意，我可以通过 并发编程 中的示例来做到这一点。

附录 11

数据压缩

Java I/O 类库提供了可以读写压缩格式流的类。你可以将其他 I/O 类包装起来用于提供压缩功能。

这些类不是从 **Reader** 和 **Writer** 类派生的，而是 **InputStream** 和 **OutputStream** 层级结构的一部分。这是由于压缩库处理的是字节，而不是字符。但是，你可能会被迫混合使用两种类型的流（请记住，你可以使用 **InputStreamReader** 和 **OutputStreamWriter**，这两个类可以在字节类型和字符类型之间轻松转换）。

压缩类	功能
CheckedInputStream	`getCheckSum()` 可以对任意 **InputStream** 计算校验和（而不只是解压）
CheckedOutputStream	`getCheckSum()` 可以对任意 **OutputStream** 计算校验和（而不只是压缩）
DeflaterOutputStream	压缩类的基类
ZipOutputStream	**DeflaterOutputStream** 类的一种，用于压缩数据到 Zip 文件结构
GZIPOutputStream	**DeflaterOutputStream** 类的一种，用于压缩数据到 GZIP 文件结构
InflaterInputStream	解压类的基类
ZipInputStream	**InflaterInputStream** 类的一种，用于解压 Zip 文件结构的数据
GZIPInputStream	**InflaterInputStream** 类的一种，用于解压 GZIP 文件结构的数据

尽管存在很多压缩算法，但是 Zip 和 GZIP 可能是最常见的。你可以使用许多用于读取和写入这些格式的工具，来轻松操作压缩数据。

11.1 使用 Gzip 简单压缩

GZIP 接口十分简单，因此当你有一个需要压缩的数据流（而不是一个包含不同数据分片的容器）时，使用 GZIP 更为合适。如下是一个压缩单个文件的示例：

```java
// compression/GZIPcompress.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// {java GZIPcompress GZIPcompress.java}
// {VisuallyInspectOutput}

public class GZIPcompress {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println(
                    "Usage: \nGZIPcompress file\n" +
                            "\tUses GZIP compression to compress " +
                            "the file to test.gz");
            System.exit(1);
        }
        try (
                InputStream in = new BufferedInputStream(
                        new FileInputStream(args[0]));
                BufferedOutputStream out =
                        new BufferedOutputStream(
                                new GZIPOutputStream(
                                        new FileOutputStream("test.gz")
                                        ))
        ) {
            System.out.println("Writing file");
            int c;
            while ((c = in.read()) != -1)
                out.write(c);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        System.out.println("Reading file");
        try (
                BufferedReader in2 = new BufferedReader(
                        new InputStreamReader(new GZIPInputStream(
                                new FileInputStream("test.gz"))))
        ) {
            in2.lines().forEach(System.out::println);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

使用压缩类非常简单，你只需要把你的输出流包装在 **GZIPOutputStream** 或 **ZipOutputStream** 中，将输入流包装在 **GZIPInputStream** 或 **ZipInputStream**。其他的一切就只是普通的 I/O 读写。这是面向字符流和面向字节流的混合示例；in 使用 Reader 类，而 **GZIPOutputStreams** 构造函数只能接受 **OutputStream** 对象，而不能接受 **Writer** 对象。当打开文件的时候，**GZIPInputStream** 会转换成为 **Reader**。

11.2 使用 zip 多文件存储

支持 Zip 格式的库比 GZIP 库更广泛。有了它，你可以轻松存储多个文件，甚至还有一个单独的类可以轻松地读取 Zip 文件。该库使用标准 Zip 格式，因此它可以与当前可在 Internet 上下载的所有 Zip 工具无缝协作。以下示例与前一个示例具有相同的形式，但它可以根据需要处理任意数量的命令行参数。此外，它还显示了 **Checksum** 类计算和验证文件的校验和。有两种校验和类型：Adler32（更快）和 CRC32（更慢但更准确）。

```
// compression/ZipCompress.java
// (c)2017 MindView LLC: see Copyright.txt
// We make no guarantees that this code is fit for any purpose.
// Visit http://OnJava8.com for more book information.
// Uses Zip compression to compress any
// number of files given on the command line
// {java ZipCompress ZipCompress.java}
// {VisuallyInspectOutput}
public class ZipCompress {
    public static void main(String[] args) {
        try (
                FileOutputStream f =
                        new FileOutputStream("test.zip");
                CheckedOutputStream csum =
                        new CheckedOutputStream(f, new Adler32());
                ZipOutputStream zos = new ZipOutputStream(csum);
                BufferedOutputStream out =
                        new BufferedOutputStream(zos)
        ) {
            zos.setComment("A test of Java Zipping");
            // No corresponding getComment(), though.
            for (String arg : args) {
                System.out.println("Writing file " + arg);
                try (
                        InputStream in = new BufferedInputStream(
                                new FileInputStream(arg))
                ) {
                    zos.putNextEntry(new ZipEntry(arg));
                    int c;
                    while ((c = in.read()) != -1)
                        out.write(c);
```

```
                }
                out.flush();
            }
            // Checksum valid only after the file is closed!
            System.out.println(
                    "Checksum: " + csum.getChecksum().getValue());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        // Now extract the files:
        System.out.println("Reading file");
        try (
                FileInputStream fi =
                        new FileInputStream("test.zip");
                CheckedInputStream csumi =
                        new CheckedInputStream(fi, new Adler32());
                ZipInputStream in2 = new ZipInputStream(csumi);
                BufferedInputStream bis =
                        new BufferedInputStream(in2)
        ) {
            ZipEntry ze;
            while ((ze = in2.getNextEntry()) != null) {
                System.out.println("Reading file " + ze);
                int x;
                while ((x = bis.read()) != -1)
                    System.out.write(x);
            }
            if (args.length == 1)
                System.out.println(
                        "Checksum: " + csumi.getChecksum().getValue());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        // Alternative way to open and read Zip files:
        try (
                ZipFile zf = new ZipFile("test.zip")
        ) {
            Enumeration e = zf.entries();
            while (e.hasMoreElements()) {
                ZipEntry ze2 = (ZipEntry) e.nextElement();
                System.out.println("File: " + ze2);
                // ... and extract the data as before
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

对于要添加到存档的每个文件，必须调用 `putNextEntry()` 并传递 **ZipEntry** 对象。**ZipEntry**

对象包含一个扩展接口，用于获取和设置 Zip 文件中该特定条目的所有可用数据：名称，压缩和未压缩大小，日期，CRC 校验和，额外字段数据，注释，压缩方法以及它是否是目录条目。但是，即使 Zip 格式有设置密码的方法，Java 的 Zip 库也不支持。虽然 **CheckedInputStream** 和 **CheckedOutputStream** 都支持 Adler32 和 CRC32 校验和，但 **ZipEntry** 类仅支持 CRC 接口。这是对基础 Zip 格式的限制，但它可能会限制你使用更快的 Adler32。

要提取文件，**ZipInputStream** 有一个 `getNextEntry()` 方法，这个方法在有文件存在的情况下调用，会返回下一个 **ZipEntry**。作为一个更简洁的替代方法，你可以使用 **ZipFile** 对象读取该文件，该对象具有方法 entries() 返回一个包裹 **ZipEntries** 的 **Enumeration**。

要读取校验和，你必须以某种方式访问关联的 **Checksum** 对象。这里保留了对 **CheckedOutputStream** 和 **CheckedInputStream** 对象的引用，但你也可以保持对 **Checksum** 对象的引用。Zip 流中的一个令人困惑的方法是 `setComment()`。如 **ZipCompress** 所示。在 Java 中，你可以在编写文件时设置注释，但是没有办法恢复 **ZipInputStream** 中的注释。注释似乎仅通过 **ZipEntry** 在逐个条目的基础上完全支持。

使用 GZIP 或 Zip 库时，你不仅被限制于文件——你可以压缩任何内容，包括通过网络连接发送的数据。

11.3 Java 的 jar

Zip 格式也用于 JAR（Java ARchive）文件格式，这是一种将一组文件收集到单个压缩文件中的方法，就像 Zip 一样。但是，与 Java 中的其他所有内容一样，JAR 文件是跨平台的，因此你不必担心平台问题。你还可以将音频和图像文件像类文件一样包含在其中。

JAR 文件由一个包含压缩文件集合的文件和一个描述它们的“清单（manifest）”组成。（你可以创建自己的清单文件；否则，jar 程序将为你执行此操作。）你可以在 JDK 文档中，找到更多关于 JAR 清单的信息。

JDK 附带的 jar 工具会自动压缩你选择的文件。你可以在命令行上调用它：

```
jar [options] destination [manifest] inputfile(s)
```

选项是一组字母（不需要连字符或任何其他指示符）。Unix / Linux 用户会注意到这些选项与 tar 命令选项的相似性。这些是：

选项	功能
c	创建一个新的或者空的归档文件
t	列出内容目录
x	提取所有文件

选项	功能
x file	提取指定的文件
f	这代表着，“传递文件的名称。”如果你不使用它，jar 假定它的输入将来自标准输入，或者，如果它正在创建一个文件，它的输出将转到标准输出。
m	代表第一个参数是用户创建的清单文件的名称。
v	生成详细的输出用于表述 jar 所作的事情
0	仅存储文件; 不压缩文件（用于创建放在类路径中的 JAR 文件）。
M	不要自动创建清单文件

如果放入 JAR 文件的文件中包含子目录，则会自动添加该子目录，包括其所有子目录等。还会保留路径信息。

以下是一些调用 jar 的典型方法。以下命令创建名为 myJarFile 的 JAR 文件。jar 包含当前目录中的所有类文件，以及自动生成的清单文件:

```
jar cf myJarFile.jar *.class
```

下一个命令与前面的示例类似，但它添加了一个名为 myManifestFile.mf 的用户创建的清单文件。:

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

这个命令输出了 myJarFile.jar 中的文件目录:

```
jar tf myJarFile.jar
```

如下添加了一个“verbose”的标志，用于生成更多关于 myJarFile.jar 中文件的详细信息:

```
jar tvf myJarFile.jar
```

假设 audio，classes 和 image 都是子目录，它将所有子目录组合到文件 myApp.jar 中。还包括“verbose”标志，以便在 jar 程序工作时提供额外的反馈:

```
jar cvf myApp.jar audio classes image
```

如果你在创建 JAR 文件时使用了 0（零）选项，该文件将会被替换在你的类路径（CLASSPATH）中:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

然后 Java 可以搜索到 lib1.jar 和 lib2.jar 的类文件。

jar 工具不像 Zip 实用程序那样通用。例如，你无法将文件添加或更新到现有 JAR 文件；只能从

头开始创建 JAR 文件。

此外，你无法将文件移动到 JAR 文件中，在移动文件时将其删除。

但是，在一个平台上创建的 JAR 文件可以通过任何其他平台上的 jar 工具透明地读取（这个问题有时会困扰 Zip 实用程序）。

附录 12

对象序列化

当你创建对象时，只要你需要，它就会一直存在，但是在程序终止时，无论如何它都不会继续存在。尽管这么做肯定是有意义的，但是仍旧存在某些情况，如果对象能够在程序不运行的情况下仍能存在并保存其信息，那将非常有用。这样，在下次运行程序时，该对象将被重建并且拥有的信息与在程序上次运行时它所拥有的信息相同。当然，你可以通过将信息写入文件或数据库来达到相同的效果，但是在使万物都成为对象的精神中，如果能够将一个对象声明为是“持久性”的，并为我们处理掉所有细节，那将会显得十分方便。

Java 的对象序列化将那些实现了 Serializable 接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。这一过程甚至可通过网络进行，这意味着序列化机制能自动弥补不同操作系统之间的差异。也就是说，可以在运行 Windows 系统的计算机上创建一个对象，将其序列化，通过网络将它发送给一台运行 Unix 系统的计算机，然后在那里准确地重新组装，而却不必担心数据在不同机器上的表示会不同，也不必关心字节的顺序或者其他任何细节。

就其本身来说，对象序列化可以实现轻量级持久性（lightweight persistence），“持久性”意味着一个对象的生存周期并不取决于程序是否正在执行它可以生存于程序的调用之间。通过将一个序列化对象写人磁盘，然后在重新调用程序时恢复该对象，就能够实现持久性的效果。之所以称其为“轻量级”，是因为不能用某种“persistent”（持久）关键字来简单地定义一个对象，并让系统自动维护其他细节问题（尽管将来有可能实现）。相反，对象必须在程序中显式地序列化（serialize）和反序列化还原（deserialize），如果需要个更严格的持久性机制，可以考虑像 Hibemate 之类的工具。

对象序列化的概念加入到语言中是为了支持两种主要特性。一是 Java 的远程方法调用（Remote Method Invocation，RMI），它使存活于其他计算机上的对象使用起来就像是存活于本机上一样。当向远程对象发送消息时，需要通过对象序列化来传输参数和返回值。

再者，对 Java Beans 来说，对象的序列化也是必需的（在撰写本文时被视为失败的技术），使用一个 Bean 时，一般情况下是在设计阶段对它的状态信息进行配置。这种状态信息必须保存下来，并在程序启动时进行后期恢复，这种具体工作就是由对象序列化完成的。

只要对象实现了 Serializable 接口（该接口仅是一个标记接口，不包括任何方法），对象的序列化处理就会非常简单。当序列化的概念被加入到语言中时，许多标准库类都发生了改变，以便具备序列化

特性-其中包括所有基本数据类型的封装器、所有容器类以及许多其他的东西。甚至 Class 对象也可以被序列化。

要序列化一个对象，首先要创建某些 OutputStream 对象，然后将其封装在一个 ObjectOutputStream 对象内。这时，只需调用 writeObject() 即可将对象序列化，并将其发送给 OutputStream（对象化序列是基于字节的，因要使用 InputStream 和 OutputStream 继承层次结构）。要反向进行该过程（即将一个序列还原为一个对象），需要将一个 InputStream 封装在 ObjectInputStream 内，然后调用 readObject()。和往常一样，我们最后获得的是一个引用，它指向一个向上转型的 Object，所以必须向下转型才能直接设置它们。

对象序列化特别“聪明”的一个地方是它不仅保存了对象的“全景图”，而且能追踪对象内所包含的所有引用，并保存那些对象；接着又能对对象内包含的每个这样的引用进行追踪，依此类推。这种情况有时被称为“对象网”，单个对象可与之建立连接，而且它还包含了对象的引用数组以及成员对象。如果必须保持一套自己的对象序列化机制，那么维护那些可追踪到所有链接的代码可能会显得非常麻烦。然而，由于 Java 的对象序列化似乎找不出什么缺点，所以请尽量不要自己动手，让它用优化的算法自动维护整个对象网。下面这个例子通过对链接的对象生成一个 worm（蠕虫）对序列化机制进行了测试。每个对象都与 worm 中的下一段链接，同时又与属于不同类（Data）的对象引用数组链接：

```
// serialization/Worm.java
// Demonstrates object serialization
import java.io.*;
import java.util.*;
class Data implements Serializable {
    private int n;
    Data(int n) { this.n = n; }
    @Override
    public String toString() {
        return Integer.toString(n);
    }
}
public class Worm implements Serializable {
    private static Random rand = new Random(47);
    private Data[] d = {
            new Data(rand.nextInt(10)),
            new Data(rand.nextInt(10)),
            new Data(rand.nextInt(10))
    };
    private Worm next;
    private char c;
    // Value of i == number of segments
    public Worm(int i, char x) {
        System.out.println("Worm constructor: " + i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
```

```
    public Worm() {
        System.out.println("No-arg constructor");
    }
    @Override
    public String toString() {
        StringBuilder result = new StringBuilder(":");
        result.append(c);
        result.append("(");
        for(Data dat : d)
            result.append(dat);
        result.append(")");
        if(next != null)
            result.append(next);
        return result.toString();
    }
    public static void
    main(String[] args) throws ClassNotFoundException,
            IOException {
        Worm w = new Worm(6, 'a');
        System.out.println("w = " + w);
        try(
                ObjectOutputStream out = new ObjectOutputStream(
                        new FileOutputStream("worm.dat"))
        ) {
            out.writeObject("Worm storage\n");
            out.writeObject(w);
        }
        try(
                ObjectInputStream in = new ObjectInputStream(
                        new FileInputStream("worm.dat"))
        ) {
            String s = (String)in.readObject();
            Worm w2 = (Worm)in.readObject();
            System.out.println(s + "w2 = " + w2);
        }
        try(
                ByteArrayOutputStream bout =
                        new ByteArrayOutputStream();
                ObjectOutputStream out2 =
                        new ObjectOutputStream(bout)
        ) {
            out2.writeObject("Worm storage\n");
            out2.writeObject(w);
            out2.flush();
            try(
                    ObjectInputStream in2 = new ObjectInputStream(
                            new ByteArrayInputStream(
                                    bout.toByteArray()))
            ) {
                String s = (String)in2.readObject();
                Worm w3 = (Worm)in2.readObject();
```

```
                    System.out.println(s + "w3 = " + w3);
                }
            }
        }
}
```

输出为:

```
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
```

更有趣的是，Worm 内的 Data 对象数组是用随机数初始化的（这样就不用怀疑编译器保留了某种原始信息），每个 Worm 段都用一个 char 加以标记。该 char 是在递归生成链接的 Worm 列表时自动产生的。要创建一个 Worm，必须告诉构造器你所希望的它的长度。在产生下一个引用时，要调用 Worm 构造器，并将长度减 1，以此类推。最后一个 next 引用则为 null（空），表示已到达 Worm 的尾部

以上这些操作都使得事情变得更加复杂，从而加大了对象序列化的难度。然而，真正的序列化过程却是非常简单的。一旦从另外某个流创建了 ObjectOutputstream，writeObject() 就会将对象序列化。注意也可以为一个 String 调用 writeObject() 也可以用与 DataOutputStream 相同的方法写入所有基本数据类型（它们具有同样的接口）。

有两段看起来相似的独立的代码。一个读写的是文件，而另一个读写的是字节数组（ByteArray），可利用序列化将对象读写到任何 DataInputStream 或者 DataOutputStream。

从输出中可以看出，被还原后的对象确实包含了原对象中的所有链接。

注意在对一个 Serializable 对象进行还原的过程中，没有调用任何构造器，包括默认的构造器。整个对象都是通过从 InputStream 中取得数据恢复而来的。

12.1 查找类

你或许会奇怪，将一个对象从它的序列化状态中恢复出来，有哪些工作是必须的呢？举个例子来说，假如我们将一个对象序列化，并通过网络将其作为文件传送给另一台计算机，那么，另一台计算机上的程序可以只利用该文件内容来还原这个对象吗？

回答这个问题的最好方法就是做一个实验。下面这个文件位于本章的子目录下:

```
// serialization/Alien.java
```

```
// A serializable class
import java.io.*;
public class Alien implements Serializable {}
```

而用于创建和序列化一个 Alien 对象的文件也位于相同的目录下:

```
// serialization/FreezeAlien.java
// Create a serialized output file
import java.io.*;
public class FreezeAlien {
    public static void main(String[] args) throws Exception {
        try(
                ObjectOutputStream out = new ObjectOutputStream(
                        new FileOutputStream("X.file"));
        ) {
            Alien quellek = new Alien();
            out.writeObject(quellek);
        }
    }
}
```

一旦该程序被编译和运行，它就会在 c12 目录下产生一个名为 X.file 的文件。以下代码位于一个名为 xiles 的子目录下:

```
// serialization/xfiles/ThawAlien.java
// Recover a serialized file
// {java serialization.xfiles.ThawAlien}
// {RunFirst: FreezeAlien}
package serialization.xfiles;
import java.io.*;
public class ThawAlien {
    public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
                new FileInputStream(new File("X.file")));
        Object mystery = in.readObject();
        System.out.println(mystery.getClass());
    }
}
```

输出为:

```
class Alien
```

为了正常运行，必须保证 Java 虚拟机能找到相关的.class 文件。

12.2 控制序列化

正如大家所看到的，默认的序列化机制并不难操纵。然而，如果有特殊的需要那又该怎么办呢？例如，也许要考虑特殊的安全问题，而且你不希望对象的某一部分被序列化；或者一个对象被还原以后，某子对象需要重新创建，从而不必将该子对象序列化。

在这些特殊情况下，可通过实现 Externalizable 接口——代替实现 Serializable 接口-来对序列化过程进行控制。这个 Externalizable 接口继承了 Serializable 接口，同时增添了两个方法：writeExternal0 和 readExternal0。这两个方法会在序列化和反序列化还原的过程中被自动调用，以便执行一些特殊操作。

下面这个例子展示了 Externalizable 接口方法的简单实现。注意 Blip1 和 Blip2 除了细微的差别之外，几乎完全一致（研究一下代码，看看你能否发现）：

```java
// serialization/Blips.java
// Simple use of Externalizable & a pitfall
import java.io.*;
class Blip1 implements Externalizable {
    public Blip1() {
        System.out.println("Blip1 Constructor");
    }
    @Override
    public void writeExternal(ObjectOutput out)
            throws IOException {
        System.out.println("Blip1.writeExternal");
    }
    @Override
    public void readExternal(ObjectInput in)
            throws IOException, ClassNotFoundException {
        System.out.println("Blip1.readExternal");
    }
}
class Blip2 implements Externalizable {
    Blip2() {
        System.out.println("Blip2 Constructor");
    }
    @Override
    public void writeExternal(ObjectOutput out)
            throws IOException {
        System.out.println("Blip2.writeExternal");
    }
    @Override
    public void readExternal(ObjectInput in)
            throws IOException, ClassNotFoundException {
        System.out.println("Blip2.readExternal");
    }
}
public class Blips {
```

```
    public static void main(String[] args) {
        System.out.println("Constructing objects:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        try(
                ObjectOutputStream o = new ObjectOutputStream(
                        new FileOutputStream("Blips.serialized"))
        ) {
            System.out.println("Saving objects:");
            o.writeObject(b1);
            o.writeObject(b2);
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        // Now get them back:
        System.out.println("Recovering b1:");
        try(
                ObjectInputStream in = new ObjectInputStream(
                        new FileInputStream("Blips.serialized"))
        ) {
            b1 = (Blip1)in.readObject();
        } catch(IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        // OOPS! Throws an exception:
        //- System.out.println("Recovering b2:");
        //- b2 = (Blip2)in.readObject();
    }
}
```

输出为:

```
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
```

没有恢复 Blip2 对象的原因是那样做会导致一个异常。你找出 Blip1 和 Blip2 之间的区别了吗? Blipl 的构造器是“公共的”（pablic），Blip2 的构造器却不是，这样就会在恢复时造成异常。试试将 Blip2 的构造器变成 public 的，然后删除//注释标记，看看是否能得到正确的结果。

恢复 b1 后，会调用 Blip1 默认构造器。这与恢复一个 Serializable 对象不同。对于 Serializable 对象，对象完全以它存储的二进制位为基础来构造，而不调用构造器。而对于一个 Externalizable 对象，所有普通的默认构造器都会被调用（包括在字段定义时的初始化），然后调用 readExternal() 必须

注意这一点–所有默认的构造器都会被调用，才能使 Externalizable 对象产生正确的行为。

下面这个例子示范了如何完整保存和恢复一个 Externalizable 对象：

```
// serialization/Blip3.java
// Reconstructing an externalizable object
import java.io.*;
public class Blip3 implements Externalizable {
    private int i;
    private String s; // No initialization
    public Blip3() {
        System.out.println("Blip3 Constructor");
// s, i not initialized
    }
    public Blip3(String x, int a) {
        System.out.println("Blip3(String x, int a)");
        s = x;
        i = a;
// s & i initialized only in non-no-arg constructor.
    }
    @Override
    public String toString() { return s + i; }
    @Override
    public void writeExternal(ObjectOutput out)
            throws IOException {
        System.out.println("Blip3.writeExternal");
// You must do this:
        out.writeObject(s);
        out.writeInt(i);
    }
    @Override
    public void readExternal(ObjectInput in)
            throws IOException, ClassNotFoundException {
        System.out.println("Blip3.readExternal");
// You must do this:
        s = (String)in.readObject();
        i = in.readInt();
    }
    public static void main(String[] args) {
        System.out.println("Constructing objects:");
        Blip3 b3 = new Blip3("A String ", 47);
        System.out.println(b3);
        try(
                ObjectOutputStream o = new ObjectOutputStream(
                        new FileOutputStream("Blip3.serialized"))
        ) {
            System.out.println("Saving object:");
            o.writeObject(b3);
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
```

```
// Now get it back:
        System.out.println("Recovering b3:");
        try(
                ObjectInputStream in = new ObjectInputStream(
                        new FileInputStream("Blip3.serialized"))
        ) {
            b3 = (Blip3)in.readObject();
        } catch(IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        System.out.println(b3);
    }
}
```

输出为:

```
Constructing objects:
Blip3(String x, int a)
A String 47
Saving object:
Blip3.writeExternal
Recovering b3:
Blip3 Constructor
Blip3.readExternal
A String 47
```

其中，字段 s 和只在第二个构造器中初始化，而不是在默认的构造器中初始化。这意味着假如不在 readExternal0 中初始化 s 和 i，s 就会为 null，而就会为零（因为在创建对象的第一步中将对象的存储空间清理为 0）。如果注释掉跟随于“You must do this”后面的两行代码，然后运行程序，就会发现当对象被还原后，s 是 null，而 i 是零。

我们如果从一个 Externalizable 对象继承，通常需要调用基类版本的 writeExternal() 和 readExternal() 来为基类组件提供恰当的存储和恢复功能。

因此,为了正常运行,我们不仅需要在 writeExternal() 方法(没有任何默认行为来为 Externalizable 对象写入任何成员对象）中将来自对象的重要信息写入，还必须在 readExternal() 方法中恢复数据。起先，可能会有一点迷惑，因为 Externalizable 对象的默认构造行为使其看起来似乎像某种自动发生的存储与恢复操作。但实际上并非如此。

12.2.1 transient 关键字

当我们对序列化进行控制时，可能某个特定子对象不想让 Java 的序列化机制自动保存与恢复。如果子对象表示的是我们不希望将其序列化的敏感信息（如密码），通常就会面临这种情况。即使对象中的这些信息是 private（私有）属性，一经序列化处理，人们就可以通过读取文件或者拦截网络传输的方式来访问到它。

有一种办法可防止对象的敏感部分被序列化，就是将类实现为 Externalizable，如前面所示。这样

一来，没有任何东西可以自动序列化，并且可以在 writeExternal() 内部只对所需部分进行显式的序列化。

然而，如果我们正在操作的是一个 Seralizable 对象，那么所有序列化操作都会自动进行。为了能够予以控制，可以用 transient（瞬时）关键字逐个字段地关闭序列化，它的意思是“不用麻烦你保存或恢复数据——我自己会处理的”。

例如，假设某个 Logon 对象保存某个特定的登录会话信息，登录的合法性通过校验之后，我们想把数据保存下来，但不包括密码。为做到这一点，最简单的办法是实现 Serializable，并将 password 字段标志为 transient，下面是具体的代码：

```
// serialization/Logon.java
// Demonstrates the "transient" keyword
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import onjava.Nap;
public class Logon implements Serializable {
    private Date date = new Date();
    private String username;
    private transient String password;
    public Logon(String name, String pwd) {
        username = name;
        password = pwd;
    }
    @Override
    public String toString() {
        return "logon info: \n username: " +
                username + "\n date: " + date +
                "\n password: " + password;
    }
    public static void main(String[] args) {
        Logon a = new Logon("Hulk", "myLittlePony");
        System.out.println("logon a = " + a);
        try(
                ObjectOutputStream o =
                        new ObjectOutputStream(
                                new FileOutputStream("Logon.dat"))
        ) {
            o.writeObject(a);
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        new Nap(1);
// Now get them back:
        try(
                ObjectInputStream in = new ObjectInputStream(
                        new FileInputStream("Logon.dat"))
        ) {
```

```
            System.out.println(
                    "Recovering object at " + new Date());
            a = (Logon)in.readObject();
        } catch(IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
        System.out.println("logon a = " + a);
    }
}
```

输出为:

```
logon a = logon info:
username: Hulk
date: Tue May 09 06:07:47 MDT 2017
password: myLittlePony
Recovering object at Tue May 09 06:07:49 MDT 2017
logon a = logon info:
username: Hulk
date: Tue May 09 06:07:47 MDT 2017
password: null
```

可以看到，其中的 date 和 username 是一般的（不是 transient 的），所以它们会被自动序列化。而 password 是 transient 的，所以不会被自动保存到磁盘；另外，自动序列化机制也不会尝试去恢复它。当对象被恢复时，password 就会变成 null。注意，虽然 toString() 是用重载后的 + 运算符来连接 String 对象，但是 null 引用会被自动转换成字符串 null。

我们还可以发现：date 字段被存储了到磁盘并从磁盘上被恢复了出来，而且没有再重新生成。由于 Externalizable 对象在默认情况下不保存它们的任何字段，所以 transient 关键字只能和 Serializable 对象一起使用。

12.2.2 Externalizable 的替代方法

如果不是特别坚持实现 Externalizable 接口，那么还有另一种方法。我们可以实现 Serializable 接口，并添加（注意我说的是“添加”，而非“覆盖”或者“实现”）名为 writeObject() 和 readObject() 的方法。这样一旦对象被序列化或者被反序列化还原，就会自动地分别调用这两个方法。也就是说，只要我们提供了这两个方法，就会使用它们而不是默认的序列化机制。

这些方法必须具有准确的方法特征签名:

```
private void writeObject(ObjectOutputStream stream) throws IOException

private void readObject(ObjectInputStream stream) throws IOException,
    ClassNotFoundException
```

从设计的观点来看，现在事情变得真是不可思议。首先，我们可能会认为由于这些方法不是基类或者 Serializable 接口的一部分，所以应该在它们自己的接口中进行定义。但是注意它们被定义成了

private，这意味着它们仅能被这个类的其他成员调用。然而，实际上我们并没有从这个类的其他方法中调用它们，而是 ObjectOutputStream 和 ObjectInputStream 对象的 writeObject() 和 readobject() 方法调用你的对象的 writeObject() 和 readObject() 方法（注意关于这里用到的相同方法名，我尽量抑制住不去谩骂。一句话：混乱）。读者可能想知道 ObjectOutputStream 和 ObjectInputStream 对象是怎样访问你的类中的 private 方法的。我们只能假设这正是序列化神奇的一部分。

在接口中定义的所有东西都自动是 public 的，因此如果 writeObject() 和 readObject() 必须是 private 的，那么它们不会是接口的一部分。因为我们必须要完全遵循其方法特征签名，所以其效果就和实现了接口一样。

在调用 ObjectOutputStream.writeObject() 时，会检查所传递的 Serializable 对象，看看是否实现了它自己的 writeObject()。如果是这样，就跳过正常的序列化过程并调用它的 writeObiect()。readObject() 的情形与此相同。

还有另外一个技巧。在你的 writeObject() 内部，可以调用 defaultWriteObject() 来选择执行默认的 writeObject()。类似地，在 readObject() 内部，我们可以调用 defaultReadObject()，下面这个简单的例子演示了如何对一个 Serializable 对象的存储与恢复进行控制：

```
// serialization/SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods
import java.io.*;
public class SerialCtl implements Serializable {
    private String a;
    private transient String b;
    public SerialCtl(String aa, String bb) {
        a = "Not Transient: " + aa;
        b = "Transient: " + bb;
    }
    @Override
    public String toString() { return a + "\n" + b; }
    private void writeObject(ObjectOutputStream stream)
            throws IOException {
        stream.defaultWriteObject();
        stream.writeObject(b);
    }
    private void readObject(ObjectInputStream stream)
            throws IOException, ClassNotFoundException {
        stream.defaultReadObject();
        b = (String)stream.readObject();
    }
    public static void main(String[] args) {
        SerialCtl sc = new SerialCtl("Test1", "Test2");
        System.out.println("Before:\n" + sc);
        try (
                ByteArrayOutputStream buf =
                        new ByteArrayOutputStream();
                ObjectOutputStream o =
```

```
                        new ObjectOutputStream(buf);
        ) {
            o.writeObject(sc);
// Now get it back:
            try (
                    ObjectInputStream in =
                            new ObjectInputStream(
                                    new ByteArrayInputStream(
                                            buf.toByteArray()));
            ) {
                SerialCtl sc2 = (SerialCtl)in.readObject();
                System.out.println("After:\n" + sc2);
            }
        } catch(IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

输出为:

```
Before:
Not Transient: Test1
Transient: Test2
After:
Not Transient: Test1
Transient: Test2
```

在这个例子中，有一个 String 字段是普通字段，而另一个是 transient 字段，用来证明非 transient 字段由 defaultWriteObject() 方法保存，而 transient 字段必须在程序中明确保存和恢复。字段是在构造器内部而不是在定义处进行初始化的，以此可以证实它们在反序列化还原期间没有被一些自动化机制初始化。

如果我们打算使用默认机制写入对象的非 transient 部分，那么必须调用 defaultwriteObject() 作为 writeObject() 中的第一个操作，并让 defaultReadObject() 作为 readObject() 中的第一个操作。这些都是奇怪的方法调用。例如，如果我们正在为 ObjectOutputStream 调用 defaultWriteObject() 且没有传递任何参数，然而不知何故它却可以运行，并且知道对象的引用以及如何写入非 transient 部分。真是奇怪之极。

对 transient 对象的存储和恢复使用了我们比较熟悉的代码。请再考虑一下在这里所发生的事情。在 main0）中，创建 SerialCtl 对象，然后将其序列化到 ObjectOutputStream（注意在这种情况下，使用的是缓冲区而不是文件-这对于 ObjectOutputStream 来说是完全一样的）。序列化发生在下面这行代码当中

```
o.writeObject(sc);
```

writeObject() 方法必须检查 sc，判断它是否拥有自己的 writeObject() 方法（不是检查接口—

—这里根本就没有接口，也不是检查类的类型，而是利用反射来真正地搜索方法)。如果有，那么就会使用它。对 readObject() 也采用了类似的方法。或许这是解决这个问题的唯一切实可行的方法，但它确实有点古怪。

12.2.3 版本控制

有时可能想要改变可序列化类的版本（比如源类的对象可能保存在数据库中）。虽然 Java 支持这种做法，但是你可能只在特殊的情况下才这样做，此外，还需要对它有相当深程度的了解（在这里我们就不再试图达到这一点）。从 http://java.oracle.com 下的 JDK 文档中对这一主题进行了非常彻底的论述。

12.3 使用持久化

个比较诱人的使用序列化技术的想法是：存储程序的一些状态，以便我们随后可以很容易地将程序恢复到当前状态。但是在我们能够这样做之前，必须回答几个问题。如果我们将两个对象-它们都具有指向第三个对象的引用-进行序列化，会发生什么情况？当我们从它们的序列化状态恢复这两个对象时，第三个对象会只出现一次吗？如果将这两个对象序列化成独立的文件，然后在代码的不同部分对它们进行反序列化还原，又会怎样呢？

下面这个例子说明了上述问题：

```
// serialization/MyWorld.java
import java.io.*;
import java.util.*;
class House implements Serializable {}
class Animal implements Serializable {
    private String name;
    private House preferredHouse;
    Animal(String nm, House h) {
        name = nm;
        preferredHouse = h;
    }
    @Override
    public String toString() {
        return name + "[" + super.toString() +
                "], " + preferredHouse + "\n";
    }
}
public class MyWorld {
    public static void main(String[] args) {
        House house = new House();
        List<Animal> animals = new ArrayList<>();
        animals.add(
                new Animal("Bosco the dog", house));
```

```
        animals.add(
                new Animal("Ralph the hamster", house));
        animals.add(
                new Animal("Molly the cat", house));
        System.out.println("animals: " + animals);
        try(
                ByteArrayOutputStream buf1 =
                        new ByteArrayOutputStream();
                ObjectOutputStream o1 =
                        new ObjectOutputStream(buf1)
        ) {
            o1.writeObject(animals);
            o1.writeObject(animals); // Write a 2nd set
// Write to a different stream:
            try(
                    ByteArrayOutputStream buf2 = new
                       ByteArrayOutputStream();
                    ObjectOutputStream o2 = new ObjectOutputStream(buf2
                       )
            ) {
                o2.writeObject(animals);
// Now get them back:
                try(
                        ObjectInputStream in1 =
                                new ObjectInputStream(
                                        new ByteArrayInputStream(
                                                buf1.toByteArray()));
                        ObjectInputStream in2 =
                                new ObjectInputStream(
                                        new ByteArrayInputStream(
                                                buf2.toByteArray()))
                ) {
                    List
                            animals1 = (List)in1.readObject(),
                            animals2 = (List)in1.readObject(),
                            animals3 = (List)in2.readObject();
                    System.out.println(
                            "animals1: " + animals1);
                    System.out.println(
                            "animals2: " + animals2);
                    System.out.println(
                            "animals3: " + animals3);
                }
            }
        } catch(IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

输出为:

```
animals: [Bosco the dog[Animal@15db9742],
House@6d06d69c
, Ralph the hamster[Animal@7852e922], House@6d06d69c
, Molly the cat[Animal@4e25154f], House@6d06d69c
]
animals1: [Bosco the dog[Animal@7ba4f24f],
House@3b9a45b3
, Ralph the hamster[Animal@7699a589], House@3b9a45b3
, Molly the cat[Animal@58372a00], House@3b9a45b3
]
animals2: [Bosco the dog[Animal@7ba4f24f],
House@3b9a45b3
, Ralph the hamster[Animal@7699a589], House@3b9a45b3
, Molly the cat[Animal@58372a00], House@3b9a45b3
]
animals3: [Bosco the dog[Animal@4dd8dc3],
House@6d03e736
, Ralph the hamster[Animal@568db2f2], House@6d03e736
, Molly the cat[Animal@378bf509], House@6d03e736
]
```

这里有一件有趣的事：我们可以通过一个字节数组来使用对象序列化，从而实现对任何可 Serializable 对象的“深度复制”（deep copy）—— 深度复制意味着我们复制的是整个对象网，而不仅仅是基本对象及其引用。复制对象将在本书的 附录：传递和返回对象 一章中进行深入地探讨。

在这个例子中，Animal 对象包含有 House 类型的字段。在 main() 方法中，创建了一个 Animal 列表并将其两次序列化，分别送至不同的流。当其被反序列化还原并被打印时，我们可以看到所示的执行某次运行后的结果（每次运行时，对象将会处在不同的内存地址）。

当然，我们期望这些反序列化还原后的对象地址与原来的地址不同。但请注意，在 animals1 和 animals2 中却出现了相同的地址，包括二者共享的那个指向 House 对象的引用。另一方面，当恢复 animals3 时，系统无法知道另一个流内的对象是第一个流内的对象的别名，因此它会产生出完全不同的对象网。

只要将任何对象序列化到单一流中，就可以恢复出与我们写出时一样的对象网，并且没有任何意外重复复制出的对象。当然，我们可以在写出第一个对象和写出最后一个对象期间改变这些对象的状态，但是这是我们自己的事，无论对象在被序列化时处于什么状态（无论它们和其他对象有什么样的连接关系），它们都可以被写出。

最安全的做法是将其作为“原子”操作进行序列化。如果我们序列化了某些东西，再去做其他一些工作，再来序列化更多的东西，如此等等，那么将无法安全地保存系统状态。取而代之的是，将构成系统状态的所有对象都置入单一容器内，并在一个操作中将该容器直接写出。然后同样只需一次方法调用，即可以将其恢复。

下面这个例子是一个想象的计算机辅助设计（CAD）系统，该例演示了这一方法。此外，它还引入了 static 字段的问题：如果我们查看 JDK 文档，就会发现 Class 是 Serializable 的，因此只需直接

对 Class 对象序列化，就可以很容易地保存 static 字段。在任何情况下，这都是一种明智的做法。

```
// serialization/AStoreCADState.java
// Saving the state of a fictitious CAD system
import java.io.*;
import java.util.*;
import java.util.stream.*;
enum Color { RED, BLUE, GREEN }
abstract class Shape implements Serializable {
    private int xPos, yPos, dimension;
    private static Random rand = new Random(47);
    private static int counter = 0;
    public abstract void setColor(Color newColor);
    public abstract Color getColor();
    Shape(int xVal, int yVal, int dim) {
        xPos = xVal;
        yPos = yVal;
        dimension = dim;
    }
    public String toString() {
        return getClass() + "color[" + getColor() +
                "] xPos[" + xPos + "] yPos[" + yPos +
                "] dim[" + dimension + "]\n";
    }
    public static Shape randomFactory() {
        int xVal = rand.nextInt(100);
        int yVal = rand.nextInt(100);
        int dim = rand.nextInt(100);
        switch(counter++ % 3) {
            default:
            case 0: return new Circle(xVal, yVal, dim);
            case 1: return new Square(xVal, yVal, dim);
            case 2: return new Line(xVal, yVal, dim);
        }
    }
}
class Circle extends Shape {
    private static Color color = Color.RED;
    Circle(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Color getColor() { return color; }
}
class Square extends Shape {
    private static Color color = Color.RED;
    Square(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
```

```
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Color getColor() { return color; }
}
class Line extends Shape {
    private static Color color = Color.RED;
    public static void
    serializeStaticState(ObjectOutputStream os)
            throws IOException { os.writeObject(color); }
    public static void
    deserializeStaticState(ObjectInputStream os)
            throws IOException, ClassNotFoundException {
        color = (Color)os.readObject();
    }
    Line(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Color getColor() { return color; }
}
public class AStoreCADState {
    public static void main(String[] args) {
        List<Class<? extends Shape>> shapeTypes =
                Arrays.asList(
                        Circle.class, Square.class, Line.class);
        List<Shape> shapes = IntStream.range(0, 10)
                .mapToObj(i -> Shape.randomFactory())
                .collect(Collectors.toList());
        // Set all the static colors to GREEN:
        shapes.forEach(s -> s.setColor(Color.GREEN));
        // Save the state vector:
        try(
                ObjectOutputStream out =
                        new ObjectOutputStream(
                                new FileOutputStream("CADState.dat"))
        ) {
            out.writeObject(shapeTypes);
            Line.serializeStaticState(out);
            out.writeObject(shapes);
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        // Display the shapes:
        System.out.println(shapes);
    }
}
```

输出为:

```
[class Circlecolor[GREEN] xPos[58] yPos[55] dim[93]
, class Squarecolor[GREEN] xPos[61] yPos[61] dim[29]
, class Linecolor[GREEN] xPos[68] yPos[0] dim[22]
, class Circlecolor[GREEN] xPos[7] yPos[88] dim[28]
, class Squarecolor[GREEN] xPos[51] yPos[89] dim[9]
, class Linecolor[GREEN] xPos[78] yPos[98] dim[61]
, class Circlecolor[GREEN] xPos[20] yPos[58] dim[16]
, class Squarecolor[GREEN] xPos[40] yPos[11] dim[22]
, class Linecolor[GREEN] xPos[4] yPos[83] dim[6]
, class Circlecolor[GREEN] xPos[75] yPos[10] dim[42]
]
```

Shape 类实现了 Serializable，所以任何自 Shape 继承的类也都会自动是 Serializable 的。每个 Shape 都含有数据，而且每个派生自 Shape 的类都包含一个 static 字段，用来确定各种 Shape 类型的颜色（如果将 static 字段置入基类，只会产生一个 static 字段，因为 static 字段不能在派生类中复制）。可对基类中的方法进行重载，以便为不同的类型设置颜色（static 方法不会动态绑定，所以这些都是普通的方法）。每次调用 randomFactory() 方法时，它都会使用不同的随机数作为 Shape 的数据，从而创建不同的 Shape。

在 main() 中，一个 ArrayList 用于保存 Class 对象，而另一个用于保存几何形状。

恢复对象相当直观：

```java
// serialization/RecoverCADState.java
// Restoring the state of the fictitious CAD system
// {RunFirst: AStoreCADState}
import java.io.*;
import java.util.*;
public class RecoverCADState {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        try(
                ObjectInputStream in =
                        new ObjectInputStream(
                                new FileInputStream("CADState.dat"))
        ) {
// Read in the same order they were written:
            List<Class<? extends Shape>> shapeTypes =
                    (List<Class<? extends Shape>>)in.readObject();
            Line.deserializeStaticState(in);
            List<Shape> shapes =
                    (List<Shape>)in.readObject();
            System.out.println(shapes);
        } catch(IOException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

输出为:

```
[class Circlecolor[RED] xPos[58] yPos[55] dim[93]
, class Squarecolor[RED] xPos[61] yPos[61] dim[29]
, class Linecolor[GREEN] xPos[68] yPos[0] dim[22]
, class Circlecolor[RED] xPos[7] yPos[88] dim[28]
, class Squarecolor[RED] xPos[51] yPos[89] dim[9]
, class Linecolor[GREEN] xPos[78] yPos[98] dim[61]
, class Circlecolor[RED] xPos[20] yPos[58] dim[16]
, class Squarecolor[RED] xPos[40] yPos[11] dim[22]
, class Linecolor[GREEN] xPos[4] yPos[83] dim[6]
, class Circlecolor[RED] xPos[75] yPos[10] dim[42]
]
```

可以看到，xPos，yPos 以及 dim 的值都被成功地保存和恢复了，但是对 static 信息的读取却出现了问题。所有读回的颜色应该都是“3”，但是真实情况却并非如此。Circle 的值为 1（定义为 RED），而 Square 的值为 0（记住，它们是在构造器中被初始化的）。看上去似乎 static 数据根本没有被序列化！确实如此——尽管 Class 类是 Serializable 的，但它却不能按我们所期望的方式运行。所以假如想序列化 static 值，必须自己动手去实现。

这正是 Line 中的 serializeStaticState() 和 deserializeStaticState() 两个 static 方法的用途。可以看到，它们是作为存储和读取过程的一部分被显式地调用的。（注意必须维护写入序列化文件和从该文件中读回的顺序。）因此，为了使 CADStatejava 正确运转起来，你必须:

1. 为几何形状添加 serializeStaticState() 和 deserializeStaticState()
2. 移除 ArrayList shapeTypes 以及与之有关的所有代码。
3. 在几何形状内添加对新的序列化和反序列化还原静态方法的调用。

另一个要注意的问题是安全，因为序列化也会将 private 数据保存下来。如果你关心安全问题，那么应将其标记成 transient，但是这之后，还必须设计一种安全的保存信息的方法，以便在执行恢复时可以复位那些 private 变量。

12.4 XML

对象序列化的一个重要限制是它只是 Java 的解决方案：只有 Java 程序才能反序列化这种对象。一种更具互操作性的解决方案是将数据转换为 XML 格式，这可以使其被各种各样的平台和语言使用。

因为 XML 十分流行，所以用它来编程时的各种选择不胜枚举，包括随 JDK 发布的 javax.xml.* 类库。我选择使用 Elliotte Rusty Harold 的开源 XOM 类库（可从 www.xom.nu 下载并获得文档），因为它看起来最简单，同时也是最直观的用 Java 产生和修改 XML 的方式。另外，XOM 还强调了 XML 的正确性。

作为一个示例，假设有一个 APerson 对象，它包含姓和名，你想将它们序列化到 XML 中。下面的 APerson 类有一个 getXML() 方法，它使用 XOM 来产生被转换为 XML 的 Element 对象的

APerson 数据；还有一个构造器，接受 Element 并从中抽取恰当的 APerson 数据（注意，XML 示例都在它们自己的子目录中）：

```
// serialization/APerson.java
// Use the XOM library to write and read XML
// nu.xom.Node comes from http://www.xom.nu
import nu.xom.*;
import java.io.*;
import java.util.*;
public class APerson {
    private String first, last;
    public APerson(String first, String last) {
        this.first = first;
        this.last = last;
    }
    // Produce an XML Element from this APerson object:
    public Element getXML() {
        Element person = new Element("person");
        Element firstName = new Element("first");
        firstName.appendChild(first);
        Element lastName = new Element("last");
        lastName.appendChild(last);
        person.appendChild(firstName);
        person.appendChild(lastName);
        return person;
    }
    // Constructor restores a APerson from XML:
    public APerson(Element person) {
        first = person
                .getFirstChildElement("first").getValue();
        last = person
                .getFirstChildElement("last").getValue();
    }
    @Override
    public String toString() {
        return first + " " + last;
    }
    // Make it human-readable:
    public static void
    format(OutputStream os, Document doc)
            throws Exception {
        Serializer serializer =
                new Serializer(os,"ISO-8859-1");
        serializer.setIndent(4);
        serializer.setMaxLength(60);
        serializer.write(doc);
        serializer.flush();
    }
    public static void main(String[] args) throws Exception {
        List<APerson> people = Arrays.asList(
```

```
                new APerson("Dr. Bunsen", "Honeydew"),
                new APerson("Gonzo", "The Great"),
                new APerson("Phillip J.", "Fry"));
        System.out.println(people);
        Element root = new Element("people");
        for(APerson p : people)
            root.appendChild(p.getXML());
        Document doc = new Document(root);
        format(System.out, doc);
        format(new BufferedOutputStream(
                new FileOutputStream("People.xml")), doc);
    }
}
```

输出为:

```
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
    <person>
        <first>Dr. Bunsen</first>
        <last>Honeydew</last>
    </person>
    <person>
        <first>Gonzo</first>
        <last>The Great</last>
    </person>
    <person>
        <first>Phillip J.</first>
        <last>Fry</last>
    </person>
</people>
```

XOM 的方法都具有相当的自解释性，可以在 XOM 文档中找到它们。XOM 还包含一个 Serializer 类，你可以在 format() 方法中看到它被用来将 XML 转换为更具可读性的格式。如果只调用 toXML()，那么所有东西都会混在一起，因此 Serializer 是一种便利工具。

从 XML 文件中反序列化 Person 对象也很简单:

```
// serialization/People.java
// nu.xom.Node comes from http://www.xom.nu
// {RunFirst: APerson}
import nu.xom.*;
import java.io.File;
import java.util.*;
public class People extends ArrayList<APerson> {
    public People(String fileName) throws Exception {
        Document doc =
                new Builder().build(new File(fileName));
        Elements elements =
                doc.getRootElement().getChildElements();
```

```
        for(int i = 0; i < elements.size(); i++)
            add(new APerson(elements.get(i)));
    }
    public static void main(String[] args) throws Exception {
        People p = new People("People.xml");
        System.out.println(p);
    }
}
/* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
*/
```

People 构造器使用 XOM 的 Builder.build() 方法打开并读取一个文件，而 getChildElements() 方法产生了一个 Elements 列表（不是标准的 Java List，只是一个拥有 size() 和 get() 方法的对象，因为 Harold 不想强制人们使用特定版本的 Java，但是仍旧希望使用类型安全的容器）。在这个列表中的每个 Element 都表示一个 Person 对象，因此它可以传递给第二个 Person 构造器。注意，这要求你提前知道 XML 文件的确切结构，但是这经常会有些问题。如果文件结构与你预期的结构不匹配，那么 XOM 将抛出异常。对你来说，如果你缺乏有关将来的 XML 结构的信息，那么就有可能会编写更复杂的代码去探测 XML 文档，而不是只对其做出假设。

为了获取这些示例去编译它们，你必须将 XOM 发布包中的 JAR 文件放置到你的类路径中。

这里只给出了用 Java 和 XOM 类库进行 XML 编程的简介，更详细的信息可以浏览 www.xom.nu 。

附录 13

静态语言类型检查

> 这是一本我多年来撰写的经过编辑过的论文集，论文集试图将静态检查语言和动态语言之间的争论放到一个正确的角度。还有一个前言部分，描述了我最近对这个话题的思考和见解。

13.1 前言

13.2 静态类型检查和测试

13.3 如何提升打字

13.4 生产力的成本

13.5 静态和动态

附录 14

C++ 和 Java 的优良传统

> 在各种讨论声中，有一些人认为 C++ 是一种设计糟糕的语言。我认为理解 C++ 和 Java 语言的选择有助于了解更大的视角。

也就是说，我几乎不再使用 C++ 了。当我使用它的时候，要么是用来检查遗留代码，要么是编写性能关键（performance-critical）部分，程序通常尽可能小，以便用其他语言编写的其他程序来调用。

因为我在最初的 8 年里一直在 C++ 标准委员会工作，所以我见证了那些被做出的决定。它们都经过了极其谨慎的考虑，远远超过了许多在 Java 中做出的决定。

然而，正如人们正确地指出的那样，由此产生的语言使用起来既复杂又痛苦，而且只要我一段时间不使用它，我就会忘记那些古怪的规则。在我写书的时候，我是从第一原理（first principles）处了解这些规则的，而不是记住了它们。

为了理解 C++ 语言为何既令人不愉快且复杂，同时又是精心设计的，必须要牢记 C++ 中所有内容的主要设计决策：与 C. Bjarne Stroustrup（该语言的创造者，即“C++ 之父”）的兼容性决定。这样的设计似乎是为了可以让大量的 C 程序员透明地转移到对象（代指 C++）上：允许他们在 C++ 下编译他们的 C 代码。这是一个巨大的限制，一直是 C++ 最大的优势……而且也是它的祸根。这就是使得 C++ 成功的原因，也是使它复杂的原因。

它也欺骗了那些不太了解 C++ 的 Java 设计师。例如，他们认为运算符重载对于程序员来说很难正确使用。这在 C++ 中基本上是正确的，因为 C++ 既有栈分配又有堆分配，你必须重载运算符来处理所有情况而且不要造成内存泄漏。这确实很困难。然而，Java 有单一的内存分配机制和一个垃圾收集器，这使得运算符重载变得微不足道，正如 C ＃中那样（但在早于 Java 的 Python 中已经可以看到）。但多年来，来自 Java 团队的一贯态度是“运算符重载太复杂了”。这里还有许多决策，所做的事明显不应该是他们做的。正是由于这些原因，让我有了蔑视 Gosling（即“Java 之父”）和 Java 团队决策的名声。（Java 7 和 8 由于某种原因包含了更好的决策。但是向后兼容性这个约束总是会阻碍真正的改进。语言永远不会是它本来的样子。）

还有很多其他的例子。“为了提高效率，必须包含基本类型”；坚持“万物皆对象”是正确的；当对性能有要求的时候，提供一个陷阱门（trap door）来做低级别的活动（lower-level activities）（这里也可以使用 hotspot 技术透明地提高性能，正如他们最终做的那样）；不能直接使用浮点处理器去计算超

越函数，它用软件来完成。我已经尽可能多地提出了这样的问题，但我得到的却一直是类似“这是 Java 方式”这样的回复。

当我提出关于泛型的设计有多糟糕的时候，我得到了相同的回复，以及“我们必须向后兼容那样以前用 Java 做出的决策”（即使它们是糟糕的决策）。最近越来越多的人已经获得了足够的泛型经验，可以发现泛型真的很难用。事实上，C++ 模板更强大、更一致（现在更容易使用，因为编译器的错误消息是可以容忍的）。人们一直在认真对待物化（reification），这可能是有用的东西，但是在那种被严格约束所削弱的设计中并没有多大影响。

这样的例子还有很多很多。这是否意味着 Java 失败了？绝对不。Java 将程序员的主流带入了垃圾收集、虚拟机和一致的错误处理模型的世界。由于它的所有缺陷，它将我们提升到了一个水平，现在我们已经准备好使用更高级别的语言了。

有一点，C++ 是领先的语言，人们认为它总是如此。许多人对 Java 有同样的看法，但由于 JVM，Java 使得取代自己变得更加容易。现在有可能会有人创建一种新语言，并使其在短时间内像 Java 一样高效运行。以前，为新语言开发一个正确有效的编译器需要花费大部分开发时间。

这种情况已经发生了，包括像 Scala 这样的高级静态语言，以及动态语言，新的且可移植的，如 Groovy，Clojure，JRuby 和 Jython。这是未来，并且过渡很顺畅，因为可以很轻易地将这些新语言与现有 Java 代码结合使用，并且必要时可以重写那些在 Java 中的瓶颈。

在撰写本文时，Java 是世界上的头号编程语言。然而，Java 最终将会减弱，就像 C++ 一样，沦只在特殊情况下使用（或者只是用来支持传统的代码，因为它不能像 C++ 那样和硬件连接）。但是无意中的好处，也是 Java 真正意外的光彩之处在于它为自己的替代品创造了一条非常畅通的道路，即使 Java 本身已经达到了无法再发展的程度。未来所有的语言都应该从中学习：要么创建一个可以重构的文化（像 Python 和 Ruby 做的那样），要么就让竞争者茁壮成长。

昨晚多几分钟的准备，今天少几小时的麻烦。

附录 15

成为一名程序员

> 我分别于 2003，2006，2007 和 2009 年撰写的博客文章混搭

15.1 如何开始

这是一条相当漫长和曲折的道路。我在高一学代数时 (1971 年)，有个非常古怪的老师有一台计算机，还弄到了一台配有一个 300 波特的音频电话耦合器的 ASR-33 电传打字机，我学会了如何执行命令并得到响应，以及一个可以在高中区使用的 HP-1000 计算机上的帐户。我们能够创建和运行 BASIC 程序并将它们保存在打孔磁带上。我对此非常着迷，所以尽可能地把它带回家后在晚上写程序。我写了一个赛马模拟游戏–HOSRAC.BAS，用星号来代表马的移动，由于是在纸上打印输出，所以需要一点想象力。

我的朋友丹尼尔（就是设计我的书封面的人）有一个兄弟，他有段时间通过向酒吧和餐馆提供弹球机来赚钱。他有一台投币式街机（老虎机），最早的《乓》游戏之一，我对此全然不知，到现在我还忍受不了这东西（现在我几乎不玩电脑游戏，这样看来我可能是个没有幽默的人，但似乎编程比玩电脑游戏更有趣、更具挑战性。）

后来我在高中参与了摄影和新闻工作，在大学的第一年就主修新闻学。我觉得自己已经从学校学到了足够多的东西，又转修了物理学。后来我在加州大学欧文分校完成了物理学位，如果我当时选择了一个特定的工程领域，修了足够的工程课就能拿到双专业，但我试图走得更远一些，所以最后我获得的本科学位是“应用物理”。作为一名本科生，我多多少少学习了一些可以自娱自乐，但又没有任何深度的计算机编程课程。我个人认为在这些课程细细熏陶下，帮我打下了一定的基础，但事实我理解的这些东西没有任何深度。我不知道计算机、编译器或解释器有什么区别（只是对编译器和解释器一点点的理解）。对我来说计算机是绝对可靠的，而且我从来没有想过在程序语言和操作系统中会有出现错误的可能。

后来我去了在加州州立理工大学攻读研究生，主要有三点原因

1. 我真的非常喜欢物理学这个领域
2. 他们接受了我，甚至给了我一份教学工作和奖学金
3. 出乎意料的是他们给我的工作时间不止一个夏天

而我完全没做好上班的准备。

作为一名物理专业的学生，我学习的是太阳能发电系统，当时太阳能发电系统很大 (如果你的房子上装了太阳能或生意上是关于太阳能系统，加州就会给予税收抵免，因此也兴起很多生意)，加州理工大学也承诺会在工程系开设相应的课程。然而因为学校没有提供必要的课程，要想获得在太阳能工程的学位得花好几年时间。所以我学习了研究生其他的工程课，包括介绍机械，太阳能，电气和电子工程。我上的课是非电气工程专业的电气工程导论。最常见的研究生工程课程是计算机工程专业，所以最后我拿了那个学位。我还上了艺术课，几门舞蹈课，还有一些计算机科学课程 (Pascal 和数据结构)，在计算机工程中，我终于弄清楚了处理器的工作流程，从那以后我一直带着一个处理器在身上。这些就是我学的计算机基础知识。

刚开始工作的时候，凭借着一堆硬件和相对简单低水平的编程，做了一名计算机工程师。因为 C 语言似乎是理想的嵌入式系统语言，于是我开始自学，并慢慢开始了解更多关于编程语言的东西。我们在这家公司从源代码构建编译器，这让我大开眼界。(想象一下一个编译器只是另一个软件的一部分!)

当我去华盛顿大学海洋学院为 Tom Keffer 后来创建了“疯狗浪”）工作时，我们决定使用 C++。我只有一本 Stroustrup 写的非初学者书可以参考，最终不得不通过检查 C++ 预处理器生成的中间 C 代码来了解语言的功能。这个过程非常痛苦，但学习的效果很好。从那以后我就用相同的方式学习，因为它让我学习了如何剖析一种语言，并看到它本质的能力，与此同时开始有了批判性思维。

我并没有理解清楚所有的概念。只是在之后的日子里不断反复，我所知道的一切需要时间才能消化吸收。如果我现在能很容易地理解一个新概念，那只是因为它是我已经知道的积累概念的一个变种。在加州理工大学招收非计算机本科学历的计算机科学研究生项目中，学生们曾经说他们花了一年的时间才弄清楚他们对计算机的困惑 (他们正在沉浸程序之中)。当人们学习计算机时，他们往往会对自己抱有不切实际的期望，通常是他们听说学计算机编程的好处，就希望在几周内找到一份高薪的工作。但是，最好的学习过程是先对计算机感兴趣，随着时间的推移，学习的越来越多，自然的就开始自学。

这些就是我主要做的事，尽管我通过学计算机工程有还算扎实的基础，但我没上过编程课，而是通过自学。在此期间我也在不断地学习新事物，在这个行业里，不断学习是非常重要的一部分。

15.2 码农生涯

我会定期收到有关职业建议的请求，所以我尝试在这里回答一下这个问题。

人们提出的问题通常是错误的问题：“我应该学习 C++ 还是 Java？”在本文中，我将尝试阐述我对选择计算机职业所涉及的真正问题的看法。

请注意，我在这里并不是和那些已经知道自己使命的人聊（译者注：指计划成为程序员或者已经从业的程序员，暗指这里是讲给外行的小白的）。因为无论别人怎么说，你都要去做，因为它已经渗入你的血液，并且你将无法摆脱它。你已经知道答案了：你当然会学到 C++ ，Java ，shell 脚本，Python 和许多其他语言和技术。即使你只有 14 岁，你也已经知道其中几种语言。

问我这个问题的人可能来自另一职业。也许他们来自 Web 开发等领域，他们已经发现 HTML 只

是一种类似编程，他们想尝试构建更实质的内容。但是，我特别希望，如果你提出这个问题，你就已经意识到，要在计算机领域取得成功，你必须教自己如何学习，并且永不停止学习。

随着我做的越来越多，在我看来，软件越发比其他任何东西都更像写作。而且我们还没有弄清怎样成为一个好的作家，我们只知道何时我们喜欢别人写的东西。这不是像一些工程那样，我们要做的只是将某些东西放到一端，然后转动曲柄。诱人的是将软件视为确定性的，这就是我们想要的，这就是我们不断推出工具来帮助我们实现所需行为的原因。但是我的经验不断表明事实是相反的：它更多地是关于人而不是过程，并且它在确定性机器上运行的事实变得越来越没有影响力（指运行环境受机器影响，与机器相关这个事实），就像海森堡原理（不确定性原理：不可能同时知道一个粒子的位置和它的速度）不会在人类规模上影响事物一样。

在我青年时期，父亲是建造民居的，我偶尔会为他工作，大部分时间都从事艰苦的工作，有时还得悬挂石膏板。他和他的木匠会告诉我说，他们是为了我才把这些工作交给了我 —— 为了不让我从事这项工作。这确实是有效的。

因此，我也可以用比喻说，建造软件就像盖房子一样。我们并不是指每个在房屋上工作的人都一样。有混凝土泥瓦匠，屋顶工，水管工，电工，石膏板工人，抹灰工，瓷砖铺砌工，普通劳工，粗木匠，精整木匠，当然还有总承包商。这些中的每一个都需要一套不同的技能，这需要花费不同的时间和精力房屋建造也受制于繁荣和萧条的周期，例如编程。为了快速起步，你可能需要当普通劳工或石膏板工人工作，在那里你可以在没有太多学习曲线的情况下开始获得报酬。只要需求旺盛，你就可以稳定工作，而且如果没有足够的人来工作，你的薪水甚至可能会上涨。但是一旦经济低迷，木匠甚至总承包商就可以自己将石膏板挂起来。

当 Internet 刚兴起时，你所要做的就是花一些时间学习 HTML ，就可以找到一份工作并赚到很多钱。但是，当情况恶化时，你很快就会发现需要的技能层次结构很深，HTML 程序员（例如劳工和石膏板工）排在第一位，而高技能的码农和木匠则被保留。

我想在这里说的是：除非你准备致力于终身学习，否则请不要从事这项业务。有时，编程似乎是一份报酬丰厚，值得信赖的工作，但确保这一点的唯一方法是，始终使自己变得更有价值。

当然，也可以找到例外。总会有一些人只学习一种语言，并且足够精通，甚至足够聪明，那么可以在不用多学很多其他知识的情况下继续工作。但是他们靠运气生存，最终很脆弱。为了减少自身的脆弱性，必须通过阅读，参加用户组，会议和研讨会来不断提高自己的能力。你在该领域的走得越深，你的价值就越大，这意味着你的工作前景更稳定，并且可以获得更高的薪水。

另一种方法是从总体上看待该领域，并找到一个你能成为专家的点。例如，我的兄弟对软件感兴趣，并且涉足软件，但是他的业务是安装计算机，维修计算机和升级计算机。他一直都很细致，因此，当他安装或修理计算机时，你会知道计算机状态良好。不仅是软件，而且一直到电缆，电缆都整齐地捆扎在一起，并且不成束。他的工作多到做不完，而且他从不关心网络泡沫破灭。毋庸置疑，他是不可能失业的。

我在大学待了很长时间，并以各种方式设法度过了难关。我甚至开始在加州大学洛杉矶分校攻读博士学位。这里的课程很短，我欣慰地说是因为我不再爱上大学了，而我在大学待了这么长时间的原因是

因为我非常喜欢。但是我喜欢的通常是跑偏的东西。例如艺术，舞蹈课程，在大学报社工作，以及我参加的少数计算机编程课程（由于我是物理本科生和计算机工程专业的研究生，所以也算跑偏）。尽管我在学业上还算是出色的（具有讽刺意味的是，当时许多不接受我作为学生的大学现在都在课程中使用我的书），但我确实很享受大学生的生活，并且完成了博士学位。我可能会走上简单的道路，最终成为一名教授。

但是事实证明，我从大学获得的最大价值一部分来自那些跑偏的课程，这些课程使我的思维超出了“我们已经知道的东西”。我认为在计算机领域尤其如此，因为你总是通过编程来实现其他目标，而你对该目标越了解，你的表现就会越好（我学习了一些欧洲研究生课程，这些课程要求结合其他一些专业研究计算，通过解决这个领域相关的问题，你就会形成一种新的理论体系并可以将它用在别处）。

我还认为，不仅编程，多了解一些其它的知识，还可以大大提高你的解决问题的能力（就像了解一种以上的编程语言可以极大地提高你的编程能力一样）。在很多情况下，我遇到过仅接受过计算机科学训练的人，他们的思维似乎比其他背景（例如数学或物理学）的人更受限制，但其实这些人（数学或物理学领域的人）才更需要严格的思维。

在我组织的一次会议上，主题之一是为理想的求职者提供一系列功能：

- 将学习作为一种生活方式。例如，学习一种以上的语言；没有什么比学习另一种语言更能吸引你的眼球。
- 知道在哪里以及如何获得新知识。
- 研究现有技术。
- 我们是工具使用者，即要善于利用工具。
- 学习做最简单的事情。
- 了解业务（阅读杂志。从 *fast company*（国外一家商业杂志）开始，该公司的文章非常简短有趣。然后你就会知道是否要阅读其他的）
- 应对错误负责。“我用着没事”是不可接受的策略。查找自己的错误。
- 成为领导者：那些沟通和鼓舞别人的人。
- 你在为谁服务？
- 没有正确的答案……但总是更好的方法。展示和讨论你的代码，不要有情感上的依恋。你不是你的代码。
- 这是通往完美的渐进旅程。

承担一切可能的风险，最好的风险是那些可怕的风险，但是在尝试时你会比想象中的更加活跃。最好不要刻意去预测某个特定的结果，因为如果你过于重视某个结果，就会经常错过真正的可能性。应该“让我们做一点实验，看看会把我们带到哪里”。这些实验是我最好的冒险。

有些人对这个答案感到失望，然后回答“是的，这都是非常有趣和有用的。但是实际上，我应该学习什么？C++ 还是 Java？”，以防这些问题，我将在这里重复一遍：我知道似乎所有的 1 和 0 都应该使一切具有确定性因此此类问题应该有一个简单的答案，但事实并非如此。这与做出选择并完成选择无关，这是有关持续学习和有时需要大胆的选择。相信我，这样你的生活会更加令人兴奋。

我依旧相信努力的意义，奋斗的价值，因为那是本质问题。

15.2.1 延伸阅读

- Teach Yourself Programming In Ten Years, by Peter Norvig.
- How To Be A Programmer, by Robert Read.
- A speech by Steve Jobs to inspire a group of graduating college students.
- Kathy Sierra: Does College Matter?
- Paul Graham on College.
- Joel Spolsky: Advice for Computer Science College Students.
- James Shore: Five Design Skills Every Programmer Should Have.
- Steve Yegge: The Truth About Interviewing.

15.3 百分之五的神话

15.4 重在动手

15.5 像打字般编程

15.6 做你喜欢的事

"1960 年，一位研究人员对 1500 名商学院学生进行了访谈，并将他们分为两类：那些为了钱财来这里上学的人，1245 人，以及那些打算利用学位做他们非常关心的事情的人，255 人。二十年后，研究人员再次访谈了这些毕业生，发现其中有 101 位百万富翁，除了其中一位，所有百万富翁都来自追求他们喜欢做的事的那 255 人！"

"现在你可能觉得你对巴洛克时期的冰岛诗歌，或者蝴蝶收集，或者高尔夫，抑或是对社会正义的热情，会因为要养家糊口而让你和你喜欢做的事分道扬镳，并非一定要如此。弗拉基米尔·纳博科夫（Vladimir Nabokov）是本世纪最伟大的小说家之一，他对蝴蝶收藏的热情远远超过写作。事实上，他的第一个大学教学工作是关于鳞翅类昆虫。在过去 40 年里，对 40 万美国群众的研究表明，即使是部分的、零散的追求培养你的激情，也可以帮助你充分利用你目前的能力，激励你培养新的能力。"—摘自《The Other 90%》Robert K.Cooper

当然你可以看 Po Bronson 写的《What Should I Do With My Life?》这本书，对这些想法进行更多的探索。

15.7 词汇表

词汇	解释
OOP (*Object-oriented programming*)	面向对象编程，一种编程思维模式和编程架构
UML (*Unified Modeling Language*)	统一建模语言，类图
Aggregation	聚合，关联关系的一种，是强的关联关系
Composition	组合，关联关系的一种，是比聚合关系强的关系
STL（*the Standard Template Library*）	C++ 标准模板库
Fibonacci Sequence	斐波那契数列，又称黄金分割数列