

Sun 公司核心技术 丛书

Effective Java
Programming Language Guide

Effective Java 中文版

(美) Joshua Bloch 著



2002年度
Jolt大奖

BOOKS

software
development,
2002
Jolt
distinction
awards



潘爱民 译



机械工业出版社
China Machine Press

Effective Java 中文版

《Effective Java中文版》介绍了57条极具实用价值的经验规则。这些经验规则涵盖了大多数开发人员每天所面临的问题的解决方案。通过对Java平台设计专家所使用的技术的全面描述，揭示了应做什么和不应做什么，才能产生清晰、健壮和高效的代码。

本书中每一条规则都以简短、独立的小文章形式出现，它们提供了深入的见解、代码示例，以及出自作者亲身经历的一些“实战经验”。这些小文章包含详细的建议和对语言中许多细微之处的深入分析，并通过代码示例加以说明。贯穿全书的是通用的语言用法和设计模式，以及一些具有启发意义的技巧和技术。

全书包括以下内容：

- 习惯和高效的语言用法，以简明、可读和易于使用的形式介绍专家的建议
- 有助于你最有效地使用Java平台的模式、反模式和习惯用法
- Java语言和它的库中通常被误解的细微之处：如何避免这些陷阱和缺陷
- 关注Java语言本身和最基本的库：java.lang、java.util和一个较小的扩展java.io
- 关于序列化的详细介绍，其中包括其他地方没有提及的一些实践建议

AWARDS



Jolt & Productivity
Award, 2002



JavaWorld Editors' Choice
Award, 2002



best selling title
at JavaOne 2001, 2002



Rated 5 stars on amazon.com.
with over 60 reviews

Recommended or required reading at many colleges and universities

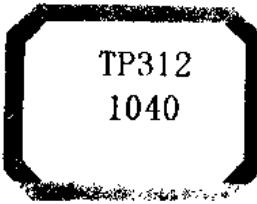


These institutions include Boston College, Clarkson University, DePaul University, Johns Hopkins University, George Mason University, Harvard, Harvey Mudd College, Imperial College, Kogakuen University, Marquette University, MIT, NYU, Princeton, Stanford, UC Berkeley, UC San Diego, UC Santa Barbara, University College London, University of Colorado, University of Hawaii, University of Maryland, University of Virginia, University of Victoria, Vanguard University, University of Texas, Western Carolina University

摘自 <http://java.sun.com/docs/books/effective/awards.html>

适用水平：中、高级





Sun公司核心技术丛书

Effective Java中文版

Effective Java Programming Language Guide

(美) Joshua Bloch 著

潘爱民 译



本书介绍了在Java编程中57条极具实用价值的经验规则，这些经验规则涵盖了大多数开发人员每天所面临的问题的解决方案。通过对Java平台设计专家所使用的技术的全面描述，揭示了应该做什么、不应该做什么才能产生清晰、健壮和高效的代码。

本书中的每条规则都以简短、独立的小文章形式出现，这些小文章包含了详细而精确的建议，以及对语言中许多细微之处的深入分析，并通过例子代码加以进一步说明。贯穿全书的是通用的语言用法和设计模式，以及一些具有启发意义的技巧和技术。

Simplified Chinese edition copyright © 2002 by PEARSON EDUCATION NORTH ASIA LIMITED and China Machine Press.

Original English language title: Effective Java Programming Language Guide, by Joshua Bloch, Copyright © 2002 Sun Microsystems, Inc. ISBN 0-201-31005-8

Published by arrangement with the original publisher, Pearson Education, Inc., publishing by Prentice-Hall, Inc..

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macau).

本书封面贴有Pearson Education培生教育出版集团激光防伪标签，无标签者不得销售。
版权所有，侵权必究。

本书版权登记号：图字：01-2001-4782

图书在版编目（CIP）数据

Effective Java中文版 / (美) 布洛克 (Bloch, J.) 著；潘爱民译. -北京：机械工业出版社，
2003.1

(Sun公司核心技术丛书)

书名原文：Effective Java Programming Language Guide

ISBN 7-111-11385-3

I. E… II. ①布… ②潘… III. JAVA语言—程序设计 IV. TP312

中国版本图书馆CIP数据核字（2002）第102866号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：贾 梅

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

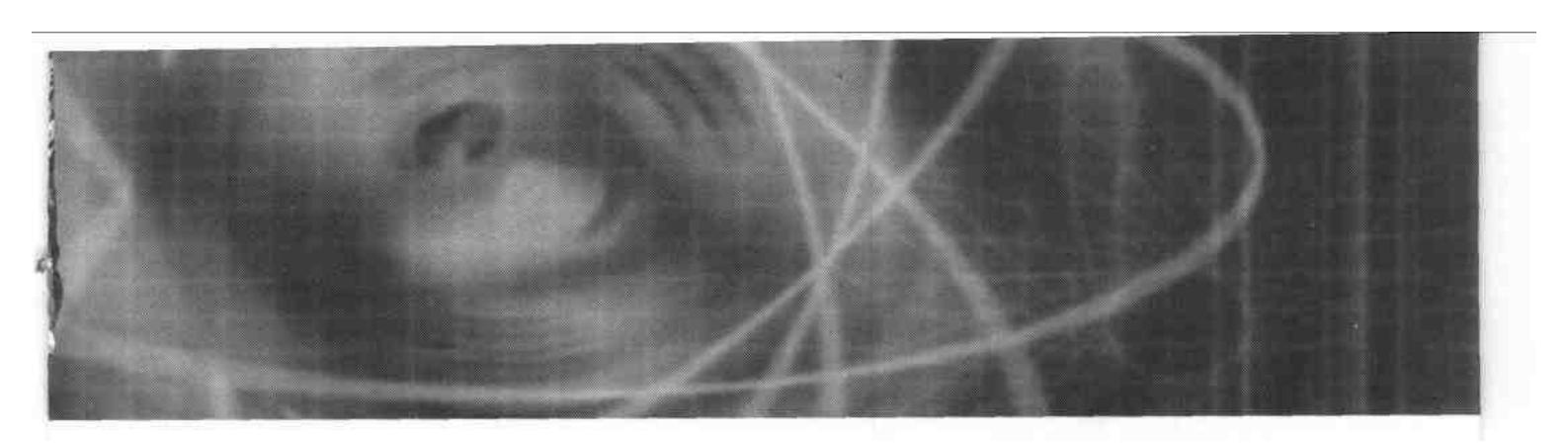
2003年1月第1版·2003年3月第2次印刷

787mm×1092mm 1/16·14.75印张

印数：5 001-8 000册

定价：39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换



译者序

这是一本不起眼的小书，但是它里边介绍了不平凡的内容。

世界上每一件事情之所以会发生都有一定的原因，这本薄薄的小书能够获得第12届软件开发Jolt图书大奖（<http://www.sdmagazine.com/>），起初让我非常惊讶，但是当我接手重新翻译这本书的任务，仔细阅读之后，我知道它获得这个奖项是当之无愧的。

我曾经译写过多部软件开发类的图书，但是从来没有进入到Java世界中来，也没有做过Java平台上的开发工作，所以，翻译这本书主要靠在其他领域的经验和感觉，也算是一种胆大的尝试。尤其让我倍感压力的是，这本书已经有了第一次翻译，我能翻译得更好吗？不管怎么样，我尽力去做就是了。

这本书之于Java程序设计语言的意义，决不亚于Scott Meyers的《Effective C++》之于C++程序设计语言的意义。我之所以这么说，不仅仅是因为这本书借用了Scott Meyers的著书风格，更重要的是，它所介绍的内容都来自于第一线实践经验，书中的每一条规则都抓住了Java程序设计语言和平台库的关键之处。这些经验对于普通程序员而言，可以让他们迅速回归到Java语言所规定的正确道路上来；对于高级程序员而言，可以让他们把每天的使用经验上升到理论和规范的高度，使之适用于更广泛的场合。本书作者Joshua Bloch是Java Collections Framework的设计师，他还设计、指导和实现了其他一些可重用的类库。本书融入了作者多年的可重用组件设计经验，所以，您在阅读本书的时候，处处都能感受到作者的设计经历，包括经验和教训，也能感受到Java平台如何克服其原始的设计缺点，逐步走向完美和成熟。

毫无疑问，这本书不适合Java语言的初学者。它并没有指导您如何编写实用的Java程序，而是指导您在Java程序设计中需要遵循什么样的规则才能编写出高效、灵活、健壮、可重用的程序。作者把焦点集中在API的设计上，所以，如果您正在设计一个可重用的系统或者子系统，那么本书中的内容对您再合适不过了。

本书共包括57条程序设计建议，有的建议是几乎每一个熟练的Java程序设计人员都亲身经历过的，比如第2章中的一些通用方法，以及第3章中有关类和接口的设计规则；有的建议涉及到Java程序设计语言的一些高级用法，比如第5章介绍了如何代替C语言中的一些类型设施，第8章讨论了异常的一些用法；有的建议涉及到Java平台的底层系统，比如第9章讨论了Java平

台上设计多线程程序应该注意的事项。这些建议涵盖了Java程序设计的方方面面，可以这样说，本书是Java程序设计浓缩的精华，每一条都值得您细细阅读和品味。

在内容叙述上，本书也颇具特色：

- 在讲述每条规则的时候，使用了一些很有说服力的短小例子，这些例子大多来自于作者的亲身经历。
- 客观地指出了Java语言和平台库中的一些设计缺陷，以及容易被忽略的实现细节。从这个意义上讲，本书称得上是一本“Java语言和平台库的技术内幕”。
- 大量运用了模式和反模式。设计模式是人们对于普遍适用的设计方案的经验总结和提炼，反模式则是应该避免的设计方案。本书把Java语言和大量现有的设计模式有机地结合起来，同时也展现了一些在Java平台库成长过程中提炼出来的新模式。
- 严谨的参考引用信息。作者不仅在叙述每个条目的时候，提供了前后相关的参考信息，同时也引用到其他一些经典资料。作者在许多细节上着墨并不多，但是他都提供了有关的参考资料，从而既保证了叙述的完整性，也保持了本书“精、巧”的特色。

本书包括这么多高质量的程序设计建议，我相信每一个有经验的Java程序设计人员都会喜欢和赞同这些建议。但是，如果把这些设计建议应用到日常的Java程序设计中，一定会编写出高质量的代码来吗？应该会的，但是这将使Java程序设计工作非常复杂，并且编写出来的代码不简洁，也不直观，这当然不符合Java语言的设计思想。所以，您需要用正确的态度来学习和使用这些设计规则。如果您正在设计可重用组件库，那么几乎每一个条目都有助于您设计出更合理的API来，您无论花多少时间来研究这些设计规则都是值得的。如果您正在设计普通的Java应用，那么，这些设计规则将有助于您更好地利用Java平台库来完成开发任务，而且一旦应用程序出现问题，您可以快速地诊断出问题所在，并找到合理的解决方案。

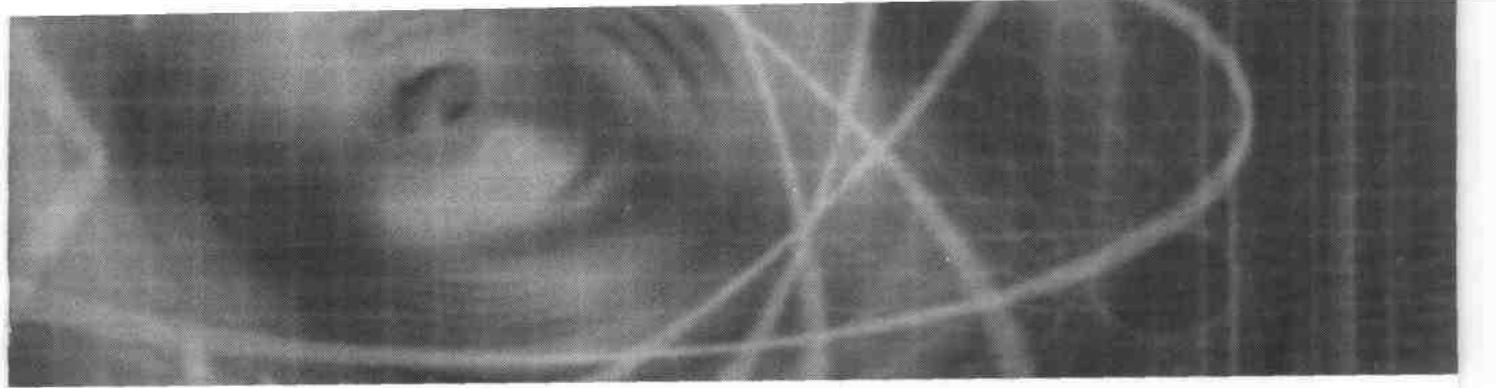
虽然这本书不适合Java语言的初学者，但是，如果您具有其他语言的程序设计经验，特别是C/C++语言的程序设计经验，那么本书对于您了解Java语言和平台库非常有帮助。而且，本书中的许多内容具有普遍适用性，并不局限于Java语言和平台，所以您一样可以从中学到有用的知识，甚至全面提升自己的程序设计能力。这是我翻译这本书过程中的切身体会。

由于本书内容深入，涉及面又比较广，加之我对于Java缺乏足够的实践经验，所以，翻译过程并不轻松，尽管对于一些疑难之处我查阅了有关的文档，特别是Java 2 平台的在线文档，但是，译文中错误在所难免，敬请读者谅解。为阅读方便，特在书后附上中英文术语对照。

这是一本好书，希望它不会辜负您的期待。

潘爱民

2002年10月于北京大学燕北园



序

如果有一个同事这样对你说，“我的配偶今天这个晚上在家里制造了一场不同寻常的晚餐，你愿意参加我们吗？”(Spouse of me this night today manufactures the unusual meal in a home. You will join?) 这时候你脑子里会想到三件事情：第一，同事是在邀请你参加家庭晚宴；第二，英语不是这位同事的母语；第三，有一顿可口的晚餐在等着你。

如果曾经学习过第二种语言，并尝试在课堂之外使用这种语言，那么你该知道有三件事情是必须要掌握的：这门语言的结构如何（语法）、如何命名你想谈论的事物（词汇），以及如何用习惯和高效的方式来表达事情（用法）。在课堂上通常只是涉及到前面两点，而当你努力使对方明白你的意思的时候，你常常会发现当地人对你的表述忍俊不禁。

对于程序设计语言，也是如此。你需要理解语言的核心：它是面向算法的，还是面向函数的，或者是面向对象的？你需要知道词汇表：标准库提供了哪些数据结构、操作和功能设施？你还需要熟悉如何用习惯和高效的方式来构建代码。关于程序设计语言的书籍通常只是涉及到前面两点，或者只是蜻蜓点水般地介绍一下用法。也许原因在于，前面两点更加容易编写。语法和词汇是语言本身固有的特性，但是用法则反映了使用这门语言的群体的特征。

例如，Java程序设计语言是一门只支持单继承的面向对象程序设计语言，在每一个方法内部，它也支持命令方式的（面向语句的，statement-oriented）编码风格。Java库包括对图形显示、网络、分布式计算和安全性的支持。但是，如何把这门语言以最佳的方式用到实践中呢？

还有一点，程序与口头的句子以及大多数书籍和杂志不同，它是会随着时间而变化的。仅仅编写出能够有效地工作并且能够被别人理解的代码往往是不够的，我们还必须要把代码组织成易于修改的形式。针对一个任务T可能会有10种不同的编码方法，而在这些10种方法中，有7种方法是笨拙的、低效的或者是难以理解的。而在剩下的3种编码方法中，哪一种会最接近该任务T的下一年度版本的代码呢？

目前有大量的书籍可以供你学习Java程序设计语言的语法，包括《The Java Programming Language》[Arnold00]（作者Arnold、Gosling和Holmes，2000），以及《The Java Language

Specification》[JLS]（作者Gosling、Joy和Bracha）。同样地，关于Java库和API的书籍也有很多。

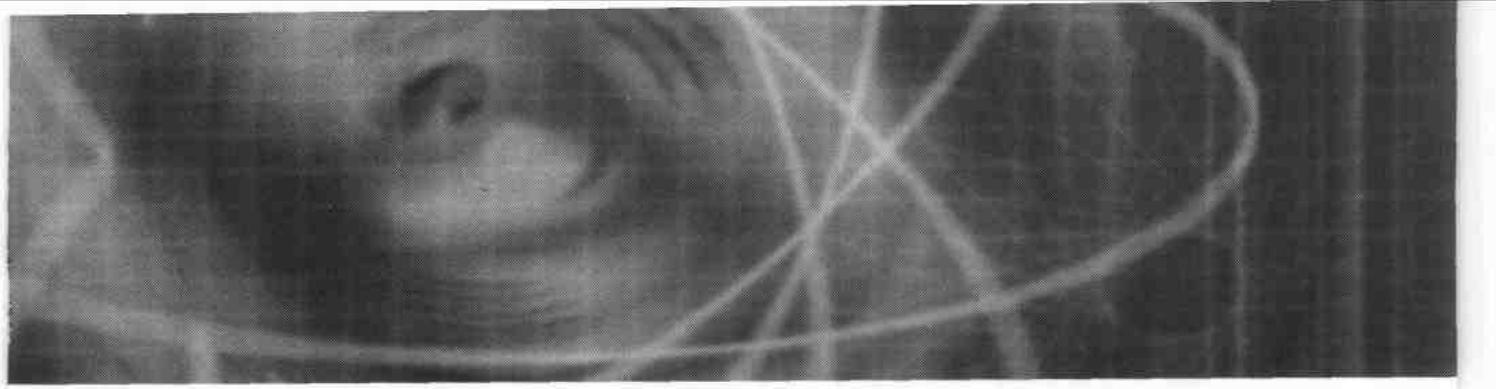
本书定位在你的第三个需要上：习惯和高效的用法。作者Joshua Bloch在Sun Microsystems公司多年来一直从事Java语言的扩展、实现和使用的工作；他还阅读了其他人的大量代码，包括我的代码。他在本书中提出了许多好的建议，他按照系统化的方式把这些建议组织起来，这些建议的宗旨在于如何更好地构造你的代码以便它们工作得更好，以便其他人也能够理解这些代码，以便将来对代码做修改和增强的时候不会头痛，甚至，你的程序因此而变得更加令人舒适、更加优美和雅致。

Guy L. Steele Jr.^Θ

Burlington, Massachusetts

2001年4月

^Θ Guy Steele是Sun研究院的杰出工程师（Distinguished Engineer）。他是程序设计语言领域的世界级专家，Scheme语言的设计者之一，1988年ACM Grace Murray Hopper奖得主。其著名著作《C: A Reference Manual》即将由机械工业出版社出版。——译注



前 言

1996年，我打点行囊，西行来到了当时的JavaSoft，因为我很清楚那里将会出现奇迹。在这5年间，我担任Java平台库的设计师。我曾经设计、实现和维护过许多库，同时也担任其他一些库的技术顾问。伴随着Java平台的成熟和壮大，主持这些库的设计工作是一个人一生中难得的机会。毫不夸张地说，我有幸与一些当代最杰出的软件工程师一起工作。在这个过程中，我学到了许多关于Java程序设计语言的知识——它能够做什么，不能够做什么，如何最有效地使用这门语言和它的库。

本书是我的一次尝试，我希望与你分享我的经验，你可以因此而吸取我的经验，避免重复我的失败。本书中我借用了Scott Meyers的《Effective C++》[Meyers98]一书的格式，该书中包含有50个条目，每个条目给出了一条用于改进程序性能和设计方案的规则。我觉得这种格式非常有效，希望你也有这样的感觉。

在许多例子中，我冒昧地使用了Java平台库中的真实例子来说明相应的条目。在介绍那些做得很不完美的工作时，我尽量使用我自己编写的代码，但是偶尔我也会使用其他同事的代码。尽管我尽力做得更好一点，但是如果我真的冒犯了他人，我在这里致以最诚挚的歉意。引用反面例子是出于协作的考虑，而不是要羞辱例子中的做法，我希望大家都能够从我们过去的错误经历中得到启发。

尽管本书并不只是针对可重用组件开发人员的，但是过去20多年来我编写此类组件的经历一定会影响到这本书。我很自然地会按照可导出API的方式来思考问题，而且我鼓励你也这样做。即使你并没有开发可重用的组件，但是这样的思考方法有助于你提高软件的质量。进一步来说，毫无意识地编写可重用组件的情形并不少见：你编写了一些很有用的代码，然后在同伴之间共享，不久之后你就有了很多用户。这时候，你就不能随心所欲地改变API了，并且如果你刚开始编写软件的时候在设计API上付出了较多的努力，那么这时你就会非常庆幸了。

我把焦点放在API的设计上，这对于那些热衷于新兴的轻量级软件开发方法学（比如Extreme Programming[Beck99]，中文译为“极限编程”，简称XP）的读者来说，也许会显得有点不太自然。这些方法学强调编写最简单的、能够工作的程序。如果你正在使用某种此类

的程序设计方法，那么你会发现，把焦点放在API设计上对于“重构（*refactoring*）”过程是多么有益。重构（*refactoring*）的基本目标是改进系统结构，以及避免代码重复。如果系统的组件没有设计良好的API，则要达到这样的目标是不可能的。

没有一门语言是完美的，但是有些语言非常优秀。我认为Java程序设计语言以及它的库非常有益于代码质量和效率的提高，并且使得编码工作成为一种乐趣。我希望本书能够抓住我的热情并传递给你，帮助你更有效地使用Java语言，工作更为愉快。

Joshua Bloch
Cupertino, California
2001年4月

请注意

本书索引所列页码，皆为英文原书页码。

“真希望10年前我就能拥有这本书。
可能有人会认为我不需要任何关于Java
的书籍，但是我确实需要这本书。”

James Gosling，Java之父，
Sun 公司副总裁

“一本非常优秀的书，充满了各种
关于使用Java程序设计语言和面向对象
程序设计的好的建议。”

Gilad Bracha，Sun公司计算机科学家，
《The Java Language Specification》
(Second Edition) 的作者之一

你正在寻找一本简明扼要地阐述Java
精髓的书吗？你希望深入地理解Java程序
设计语言吗？你希望编写出清晰、正确、
健壮和可重用的代码吗？不用再找了，你
手上这本书将会使你实现这些愿望，而且
还能提供其他许多你意想不到的好处。



作者简介

Joshua Bloch 是Sun公司的高级工程师，也是“Java平台核心组”的设计师。他设计并实现了获奖的 Java Collections Framework和java.math 软件包，并且对Java平台的其他部分也做出了贡献。Joshua是许多技术文章和论文的作者，他的关于抽象数据对象复制的博士论文获得过“ACM杰出博士论文奖”提名。他拥有哥伦比亚大学的学士学位和卡耐基-梅隆大学的博士学位。

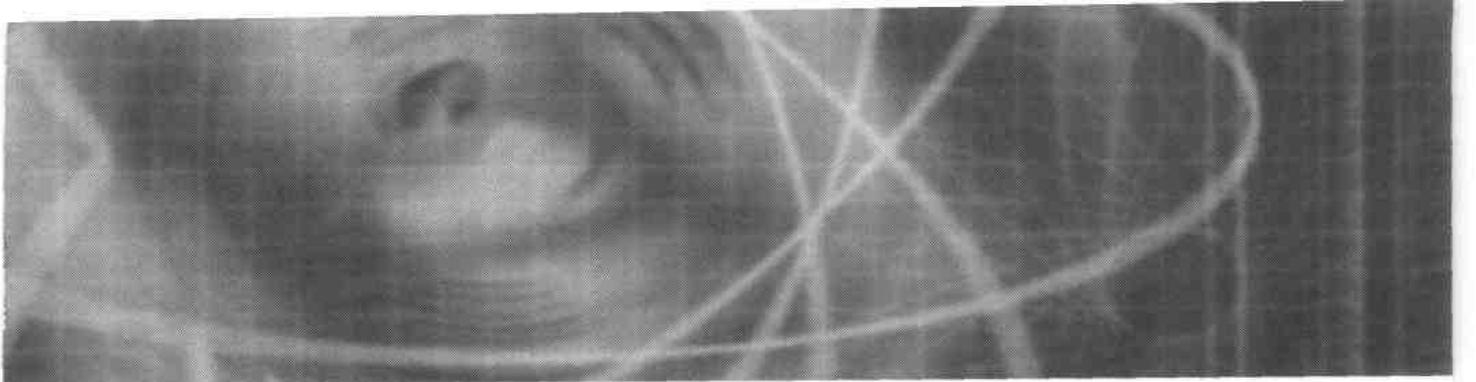
译者简介

潘爱民 浙江海宁人，现任职于北京大学计算机科学技术研究所，副研究员；研究方向为信息安全（包括网络安全和公钥技术）和软件开发（包括组件技术和模式）；主要著作有《COM原理与应用》等，译著有《Visual C++技术内幕》（第4版）、《COM本质论》和《C++ Primer中文版》等。

目 录

译者序	
序	
前言	
第1章 引言	1
第2章 创建和销毁对象	4
第1条：考虑用静态工厂方法代替构造函数	4
第2条：使用私有构造函数强化singleton属性	8
第3条：通过私有构造函数强化不可实例化的能力	10
第4条：避免创建重复的对象	11
第5条：消除过期的对象引用	14
第6条：避免使用终结函数	17
第3章 对于所有对象都通用的方法	21
第7条：在改写equals的时候请遵守通用约定	21
第8条：改写equals时总是要改写hashCode	31
第9条：总是要改写toString	36
第10条：谨慎地改写clone	39
第11条：考虑实现Comparable接口	46
第4章 类和接口	51
第12条：使类和成员的可访问能力最小化	51
第13条：支持非可变性	55
第14条：复合优先于继承	62
第15条：要么专门为继承而设计，并给出文档说明，要么禁止继承	67
第16条：接口优于抽象类	72
第17条：接口只是被用于定义类型	76
第18条：优先考虑静态成员类	78
第5章 C语言结构的替代	82
第19条：用类代替结构	82
第20条：用类层次来代替联合	84
第21条：用类来代替enum结构	88
第22条：用类和接口来代替函数指针	97
第6章 方法	100
第23条：检查参数的有效性	100
第24条：需要时使用保护性拷贝	103
第25条：谨慎设计方法的原型	107
第26条：谨慎地使用重载	109
第27条：返回零长度的数组而不是null	114
第28条：为所有导出的API元素编写文档注释	116
第7章 通用程序设计	120
第29条：将局部变量的作用域最小化	120
第30条：了解和使用库	123
第31条：如果要求精确的答案，请避免使用float和double	127
第32条：如果其他类型更适合，则尽量避免使用字符串	129
第33条：了解字符串连接的性能	131
第34条：通过接口引用对象	132
第35条：接口优先于映像机制	134
第36条：谨慎地使用本地方方法	137
第37条：谨慎地进行优化	138

第38条：遵守普遍接受的命名惯例	141
第8章 异常	144
第39条：只针对不正常的条件才使用异常	144
第40条：对于可恢复的条件使用被检查的 异常，对于程序错误使用运行时 异常	147
第41条：避免不必要的使用被检查的异常	149
第42条：尽量使用标准的异常	151
第43条：抛出的异常要适合于相应的抽象	153
第44条：每个方法抛出的异常都要有文档	155
第45条：在细节消息中包含失败 – 捕获信息	157
第46条：努力使失败保持原子性	159
第47条：不要忽略异常	161
第9章 线程	162
第48条：对共享可变数据的同步访问	162
第49条：避免过多的同步	168
第50条：永远不要在循环的外面调用wait	173
第51条：不要依赖于线程调度器	175
第52条：线程安全性的文档化	178
第53条：避免使用线程组	181
第10章 序列化	182
第54条：谨慎地实现Serializable	182
第55条：考虑使用自定义的序列化形式	187
第56条：保护性地编写readObject方法	193
第57条：必要时提供一个readResolve方法	199
中英文术语对照	202
参考文献	207
模式和习惯用法索引	212
索引	214



第1章

引言

本书的目标是帮助你最有效地使用Java™程序设计语言，以及它的基本库：`java.lang`、`java.util`，某种程度上还包括`java.io`。本书也会不时地讨论到其他的库，但是没有涉及图形用户界面编程以及企业级API。

本书共包含57个条目，每一条都涵盖一个规则。这些规则反映了最有经验的程序员在实践中的一些有益的做法。这些条目以一种比较松散的方式组织成9章，每一章涉及软件设计的一个主要方面。对本书并不一定要从前到后逐页阅读，每一个条目都有一定程度的独立性。这些条目相互之间有交叉索引，你可以按照自己的思路来通读全书。

大多数条目都通过程序例子来说明每一条的内容。本书的一个突出的特点是，它包含了許多代码例子，这些例子说明了许多设计模式（*design pattern*）和习惯用法（*idiom*）。其中一些模式是老的，例如Singleton（见第2条）；其他一些模式则是新的，例如Finalizer Guardian（终结函数守卫者）（见第6条）和Defensive `readResolve`（保护性的`readResolve`方法）（见第57条）。书末有一个单独的索引表可用来快速地查找到这些设计模式和习惯用法。在需要参考设计模式领域标准参考书[Gamma 95]的地方，还为这些模式和习惯用法提供了交叉索引。

许多条目包含有一个或多个在实践中应该避免的程序例子。像这样的例子，有时候也被称为“反模式（*antipattern*）”，在注释中被清楚地标注为“//Never do this!”。对于每一种情况，该条目都解释了为什么此例不好，并提出了可选的解决方法。

本书并不是针对初学者的：本书假设读者已经熟悉Java程序设计语言。如果还没有，请考虑先参考一本好的Java入门书籍[Arnold00, Campione00]。本书的目标是适用于任何一个具有实际Java工作经验的程序员，对于高级程序员，它也应该能够提供很有价值的参考材料。

本书中大多数规则源于少数几条基本的原则。清晰性和简洁性是最为重要的：一个模块的用户永远也不应该被模块的行为所迷惑（那样就不清晰了）；模块要尽可能的小，但又不能太小 [术语模块（*module*）在本书中的用法，是指任何可重用的软件组件，从单个方法，到

包含多个包的复杂系统都可以是一个模块]。代码应该被重用，而不是被拷贝。模块之间的相依性应该尽可能地降低到最小。错误应该尽早被检测出来，理想情况下是在编译时刻。

虽然本书中的规则不会百分之百地适用于任何时刻和任何场合，但是，它们确实刻画了绝大多数情况下最佳的程序设计行为。你不应该盲目地遵从这些规则，但是，你只应该在偶尔的情况下，有了充分的理由之后才打破这些规则。学习程序设计的艺术，如同大多数其他的学科一样，首先要学会基本的规则，然后才能知道什么时候可以打破这些规则。

本书中大部分内容都不是讨论性能的，而是关心如何编写出清晰、正确、可用、健壮、灵活和可维护的程序来。如果你能够做到这一点的话，那么要想获得你所需要的性能往往是相对比较简单的（见第37条）。有些条目确实讨论了性能问题，甚至有的还提供了性能指标。但是，在提及这些指标的时候，也会出现“在我的机器上”这样的话，所以，你最好把这些指标看做近似值。

有必要提及的是，我的机器是一台过时的家用电脑，主机为400MHz Pentium II，128M内存，在Microsoft Windows NT 4.0操作系统平台上运行Sun 1.3版本的Java 2 Standard Edition Software Development Kit (SDK)。该SDK包括Sun的Java HotSpotTM Client VM——这是一个专门供客户端使用的新一代Java虚拟机（JVM）。

在讨论到Java程序设计语言及其库的特性的时候，有时候必须要指明具体的发行版本。为简单起见，本书使用了工程版本号（engineering version number），而不是正式的发行号。表1-1列出了发行号与工程版本号之间的对应关系。

表1-1 Java平台的版本

正式发行号	工程版本号
JDK 1.1.x/JRE 1.1.x	1.1
Java 2 Platform, Standard Edition, v 1.2	1.2
Java 2 Platform, Standard Edition, v 1.3	1.3
Java 2 Platform, Standard Edition, v 1.4	1.4

虽然有些条目讨论到了1.4发行版本中引入的特性，但是程序例子中除了个别情况外，尽量避免使用这些特性。这些例子在发行版本1.3上已通过测试，大多数例子（不是所有的例子）无需修改就可以在发行版本1.2上运行。

尽管这些例子都很完整，但是它们更注重于可读性。它们直接使用了java.util和java.io包中的类。为了编译这些例子程序，你可能需要在程序中加上一行或者两行import语句：

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

其他的代码示例中也有类似被省略的情况。本书的Web站点，<http://java.sun.com/docs/books/effective>，包含了每个例子的完整版本，你可以直接编译和运行这些例子。

在很大程度上，本书使用的技术术语与《The Java Language Specification, Second Edition》[JLS]相同，而有一些术语值得特别提出来。Java语言支持四种类型：接口 (*interface*)、类 (*class*)、数组 (*array*) 和原语类型 (*primitive*)。前三种类型通常被称为引用类型 (*reference type*)，类的实例和数组是对象 (*object*)，而原语类型的值不是对象。一个类的成员 (*member*) 包括它的域 (*field*)、方法 (*method*)、成员类 (*member class*) 和成员接口 (*member interface*)。一个方法的原型 (*signature*) 包括它的名字和所有形参的类型，方法原型不包括它的返回类型。

本书也使用了一些与《The Java Language Specification》不同的术语。与《The Java Language Specification》不同的是，本书把术语“继承 (*inheritance*)”用做“子类化 (*subclassing*)”的同义词。本书不再使用“接口继承”这种说法，而是简单地说，一个类实现 (*implement*) 了一个接口，或者一个接口扩展 (*extend*) 了另一个接口。为了描述“在没有指定访问级别的的情况下所使用的访问级别”，本书使用了描述性的术语“包级私有 (*package-private*)”，而不是如[JLS, 6.6.1]中所使用的技术性术语“默认访问 (*default access*)”级别”。

本书也使用了一些在《The Java Language Specification》中没有定义的技术术语。术语“导出的API (*exported API*)”，或者简单地说API，是指类、接口、构造函数 (*constructors*)、成员和序列化形式 (*serialized form*)，程序员通过它们可以访问一个类、接口或包（术语API是*Application Programming Interface*的简写，这里之所以使用API而不用接口 (*interface*)，是为了不与Java语言中的interface类型相混淆）。使用API编写程序的程序员被称为该API的用户 (*user*)，在类的实现中使用了API的类被称为该API的客户 (*client*)。

3

类、接口、构造函数、成员以及序列化形式被统称为API元素 (*API element*)。导出的API包括所有可在定义该API的包之外访问的API元素。任何客户都可以使用这些API元素，而API的创建者负责支持这些API元素。无独有偶，Javadoc实用工具在它的默认操作模式下生成的文档中也正是这些元素。可以不严格地讲，一个包的导出API包括该包中每一个公有 (*public*) 类或者接口中所有公有的或者受保护的 (*protected*) 成员和构造函数。

4

第2章

创建和销毁对象

本章的主题是创建和销毁对象：什么时候、如何创建对象；什么时候、如何避免创建对象；如何保证对象能够适时地销毁；对象被销毁之前如何管理各种清理工作。

第1条：考虑用静态工厂方法代替构造函数

对于一个类，为了让客户获得它的一个实例，最通常的方法是提供一个公有的构造函数。实际上还有另外一种技术，尽管较少为人所知，但也应该成为每个程序员的工具箱中的一部分。类可以提供一个公有的静态工厂方法（*static factory method*），所谓静态工厂方法，实际上只是一个简单的静态方法，它返回的是类的一个实例。下面是一个来自 Boolean 类（原语类型 boolean 的包装类）的简单例子。其中静态工厂方法是 1.4 版新增的，它把一个 boolean 原语值转换为一个 Boolean 对象引用：

```
public static Boolean valueOf(boolean b) {  
    return (b ? Boolean.TRUE : Boolean.FALSE);  
}
```

类可以为它的客户提供一些静态工厂方法，来替代构造函数，或者同时也提供一些构造函数。用静态工厂方法来代替公有的构造函数，既有好处，也有不足之处。

静态工厂方法的一个好处是，与构造函数不同，静态工厂方法具有名字。如果一个构造函数的参数并没有确切地描述被返回的对象，那么选用适当名字的静态工厂可以使一个类更易于使用，

5 并且相应的客户代码更易于阅读。例如，构造函数 BigInteger (int, int, Random) 返回的 BigInteger 可能是素数，然而，如果使用一个名为 BigInteger.probablePrime 的静态工厂方法，表达显然更为清楚。（静态工厂方法 BigInteger.probablePrime 最终已加入 1.4 版了。）

一个类只能有一个原型相同的构造函数。程序员通常知道该如何绕开这种限制，他可以提

供两个构造函数，它们的参数列表只是在参数类型的顺序上有所不同。这并不是一个好主意，面对这样的API，用户永远也记不住该用哪个构造函数，结果常常会调用到错误的构造函数上。并且，读到使用这样的构造函数的代码时往往会不知所云，除非去查看该类的文档。

因为静态工厂方法自己有名字，所以它们没有构造函数那样的限制，对于给定的原型特征，可以有不止一个静态工厂方法。如果一个类看起来需要多个构造函数，并且它们的原型特征相同，那么你应该考虑用静态工厂方法来代替其中一个或者多个构造函数，并且慎重选择它们的名字以便明显地标示出它们的不同。

静态工厂方法的第二个好处是，与构造函数不同，它们每次被调用的时候，不要求非得创建一个新的对象。这使得一些非可变类（见第13条）可以使用一个预先构造好的实例，或者把已经构造好的实例缓存起来，以后再把这些实例分发给客户，从而避免创建不必要的重复对象。`Boolean.valueOf(boolean)`方法说明了这项技术：它从来不创建对象。如果一个程序要频繁地创建相同的对象，并且创建对象的代价很昂贵，则这项技术可以极大地提高性能。

静态工厂方法可以为重复的调用返回同一个对象，这也可以被用来控制“在某一时刻哪些实例应该存在”。这样做有两个理由。第一，它使得一个类可以保证是一个singleton（见第2条）。第二，它使非可变类可以保证“不会有两个相等的实例存在”，即当且仅当`a==b`的时候才有`a.equals(b)`为true。如果一个类保证了这一点，那么它的客户就可以用`==`操作符来代替`equals(Object)`方法，其结果是实质性的性能提高。在第21条中介绍的类型安全枚举(*typesafe enum*)模式实现了这项优化，而`String.intern`方法以一种有限的形式实现了这种优化。

静态工厂方法的第三个好处是，与构造函数不同，它们可以返回一个原返回类型的子类型的对象。这样我们在选择被返回对象的类型时就有了更大的灵活性。

6

这种灵活性的一个应用是，一个API可以返回一个对象，同时又不使该对象的类成为公有的。以这种方式把具体的实现类隐藏起来，可以得到一个非常简洁的API。这项技术非常适合于基于接口的框架结构，因为在这样的框架结构中，接口成为静态工厂方法的自然返回类型。

例如，Collections Framework有20个实用的集合接口实现，分别提供了不可修改的集合、同步集合等等。这些实现绝大多数都是通过一个不可实例化的类(`java.util.Collections`)中的静态工厂方法而被导出的，所有返回对象的类都不是公有的。

现在的Collections Framework API比导出20个独立的公有类的那种实现方式要小得多，这不仅是指API数量上的减少，而且也是概念意义上的减少。用户知道，被返回的对象是由相关的接口精确指定的，所以他们不需要阅读有关的类文档。更进一步，使用这样的静态工厂

方法，可以强迫客户通过接口来引用被返回的对象，而不是通过实现类来引用被返回的对象，这是一个很好的习惯（见第34条）。

公有的静态工厂方法所返回的对象的类不仅可以是非公有的，而且该类可以随着每次调用而发生变化，这取决于静态工厂方法的参数值。只要是已声明的返回类型的子类型，都是允许的。而且，为了增强软件的可维护性，返回对象的类也可以随着不同的发行版本而不同。

静态工厂方法返回的对象所属的类，在编写包含该静态工厂方法的类时可以并不存在。这种灵活的静态工厂方法构成了服务提供者框架（*service provider framework*）的基础，比如Java密码系统扩展（JCE，Java Cryptography Extension），服务提供者框架是指这样一个系统：提供者为框架的用户提供了多个API实现，框架必须提供一种机制来注册（register）这些实现，以便用户能够使用它们。框架的客户直接使用API，无需关心自己到底在使用哪个实现。

在JCE中，系统管理员通过编辑一个被人熟知的Properties文件，加入一个条目，把一个字符串键（string key）映射到对应的类名，以此来注册一个实现类。客户使用静态工厂方法的时候把字符串键作为参数传递进去。静态工厂方法在一个映射表中查找Class对象，该映射表根据Properties文件进行初始化，然后静态工厂方法使用Class.newInstance方法实例化这个类。下面的程序框架说明了这项技术：

```
// Provider framework sketch
public abstract class Foo {
    // Maps String key to corresponding Class object
    private static Map implementations = null;

    // Initializes implementations map the first time it's called
    private static synchronized void initMapIfNecessary() {
        if (implementations == null) {
            implementations = new HashMap();

            // Load implementation class names and keys from
            // Properties file, translate names into Class
            // objects using Class.forName and store mappings.
            ...
        }
    }

    public static Foo getInstance(String key) {
        initMapIfNecessary();
        Class c = (Class) implementations.get(key);
        if (c == null)
            return new DefaultFoo();

        try {
            return (Foo) c.newInstance();
        } catch (Exception e) {
            return new DefaultFoo();
        }
    }
}
```

静态工厂方法的主要缺点是，类如果不含公有的或者受保护的构造函数，就不能被子类化。对于公有的静态工厂所返回的非公有类，也同样如此。例如，要想子类化Collections Framework中的任何一个方便的实现类，是不可能的。然而这也许会因祸得福，因为它会鼓励程序员使用复合结构，而不是继承（见第14条）。

静态工厂方法的第二个缺点是，它们与其他的静态方法没有任何区别。在API文档中，它们不会像构造函数那样被明确标示出来。而且，静态工厂方法代表了一种对规范的背离，因此，对于提供了静态工厂方法而不是构造函数的类来说，要想在类文档中说明如何实例化一个类，这是非常困难的。如果遵守标准的命名习惯，就可以将这个缺点减小到最少。这些命名习惯仍在演化中，但是静态工厂方法的两个名字已经变得很流行了：

- `valueOf`——不太严格地讲，该方法返回的实例与它的参数具有同样的值。使用这个名字的静态工厂方法是一些非常有效的类型转换操作符。
- `getInstance`——返回的实例是由方法的参数来描述的，但是不能够说与参数具有同样的值。对于`singleton`的情形，该方法返回惟一的实例。这个名字在提供者框架中用得很普遍。

总的来说，静态工厂方法和公有的构造函数都有它们各自的用途，我们需要理解它们各自的长处。要避免一上来就提供构造函数，而不考虑静态工厂，因为静态工厂通常更加合适。如果你正在权衡这两种选择，又没有其他因素强烈地影响你的选择，那么你最好还是简单地使用构造函数，毕竟它是语言提供的规范。

8

9

第2条：使用私有构造函数强化singleton属性

`singleton`是指这样的类，它只能实例化一次[Gamma95, p. 127]。`singleton`通常被用来代表那些本质上具有惟一性的系统组件，比如视频显示或者文件系统。

实现`singleton`有两种方法。这两种方法都要把构造函数保持为私有的，并且提供一个静态成员，以便允许客户能够访问该类惟一的实例。在第一种方法中，公有静态成员是一个`final`域：

```
// Singleton with final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() {
        ...
    }
    ... // Remainder omitted
}
```

私有构造函数仅被调用一次，用来实例化公有的静态`final`域`Elvis.INSTANCE`。由于缺少公有的或者受保护的构造函数，所以保证了`Elvis`的全局惟一性：一旦`Elvis`类被实例化之后，只有一个`Elvis`实例存在——不多也不少。客户的任何行为都不会改变这一点。

第二种方法提供了一个公有的静态工厂方法，而不是公有的静态`final`域：

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() {
        ...
    }
    public static Elvis getInstance() {
        return INSTANCE;
    }
    ... // Remainder omitted
}
```

所有对于静态方法`Elvis.getInstance`的调用，都会返回同一个对象引用，所以，不会有别的`Elvis`实例被创建。

第一种方法的主要好处在于，组成类的成员的声明很清楚地表明了这个类是一个`singleton`：公有的静态域是`final`的，所以该域将总是包含相同的对象引用。第一种方法可能在性能上稍

微领先，但是在第二种方法中，一个优秀的JVM实现应该能够通过将静态工厂方法的调用内联化（inlining），来消除这种差别。

第二种方法的主要好处在于，它提供了灵活性：在不改变API的前提下，允许我们改变想法，把该类做成singleton，或者不做singleton。singleton的静态工厂方法返回该类的惟一实例，但是，它也很容易被修改，比如说，为每个调用该方法的线程返回一个惟一的实例。

总而言之，如果你确信该类将永远是一个singleton，那么使用第一种方法是有意义的。如果你希望保留一点余地，那么请使用第二种方法。

为了使一个singleton类变成可序列化的（serializable）（见第10章），仅仅在声明中加上“implements Serializable”是不够的。为了维护singleton性，你必须也要提供一个readResolve方法（见第57条）。否则的话，一个序列化的实例在每次反序列化的时候，都会导致创建一个新的实例，比如说，在我们的例子中，会导致“假冒的Elvis”。为了防止这种情况，在Elvis类中加入下面的readResolve方法：

```
// readResolve method to preserve singleton property
private Object readResolve() throws ObjectStreamException {
    /*
     * Return the one true Elvis and let the garbage collector
     * take care of the Elvis impersonator.
     */
    return INSTANCE;
}
```

这一条和第21条（讲述了类型安全枚举模式）实质上反映了同样的主题，在两种情况下，私有的构造函数与公有的静态成员联合起来，以确保当该类被初始化之后，不会再有新的实例被创建。在本条目的情形下，该类只有一个实例被创建；而在第21条中，枚举类型的每一个成员都有一个实例被创建。在下一条（第3条）中，这种方法会被进一步发挥：不存在公有的构造函数，以确保一个类永远也不会有实例被创建。

11

第3条：通过私有构造函数强化不可实例化的能力

偶尔情况下，你可能会编写出只包含静态方法和静态域的类。这样的类有一些很不好的名声，因为有些人在面向对象的语言中滥用这样的类来编写过程化的程序。尽管如此，它们也确实有它们特有的用处。我们可以利用这种类，把操作在原语类型的值或者数组类型上的相关方法组织起来，例如`java.lang.Math`或者`java.util.Arrays`；我们也可以把操作在实现特定接口的对象上的方法组织起来，例如`java.util.Collections`。我们还可以利用这种类把操作在`final`类上的方法组织起来，以取代扩展该类的做法。

这样的工具类（utility class）不希望被实例化，对它进行实例化没有任何意义。然而，在缺少显式构造函数的情况下，编译器会自动提供一个公有的、无参数的默认构造函数（default constructor）。对于用户而言，这个构造函数与其他的构造函数没有任何区别。在已发行的API中常常可以看到一些被无意识地实例化的类，这样的情形并不少见。

企图通过将一个类做成抽象类来强制该类不可被实例化，这是行不通的。该类可以被子类化，并且该子类也可以被实例化。更进一步，这样做会误导用户，以为这种类是专门为了继承而设计的（见第15条）。然而，有一些简单的习惯用法可以确保一个类不可被实例化。由于只有当一个类不包含显式的构造函数的时候，编译器才会生成默认构造函数，所以，我们只要让这个类包含单个显式的私有构造函数，则它就不可被实例化了：

```
// Noninstantiable utility class
public class UtilityClass {

    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        // This constructor will never be invoked
    }
    ... // Remainder omitted
}
```

因为显式构造函数是私有的，所以在该类的外部它是不可被访问的。假设该构造函数不会被类自身从内部调用，就能保证该类永远不会被实例化。这种习惯用法有点违反直觉，好像构造函数就专门设计成不能被调用一样。所以，明智的做法是在代码中对于构造函数的作用增加一些注释。

这种习惯用法也有副作用，它使得一个类不能被子类化。所有的构造函数都必须要调用一个可访问的超类（superclass）构造函数，无论显式地或隐式地调用，在这种情形下，子类就没有可访问的构造函数来调用了。

第4条：避免创建重复的对象

重复使用同一个对象，而不是每次需要的时候就创建一个功能上等价的新对象，通常前者更为合适。重用方式既快速，也更为流行。如果一个对象是非可变的 (*immutable*)（见第13条），那么它总是可以被重用。

作为一个极端的反面例子，考虑下面的语句：

```
String s = new String("silly"); // DON'T DO THIS!
```

该语句每次被执行的时候都创建一个新的String实例，但是这些创建对象的动作没有一个是真正必需的。传递给String构造函数的实参 (“silly”) 本身就是一个String实例，功能上等同于所有被构造函数创建的对象。如果这种用法是在一个循环中，或者是在一个被频繁调用的方法中，那么成千上万不必要的String实例会被创建出来。

一个改进版本如下所示：

```
String s = "No longer silly";
```

这个版本只使用一个String实例，而不是每次被执行的时候创建一个新的实例。而且，它可以保证，对于所有在同一个虚拟机中运行的代码，只要它们包含相同的字符串字面常量，则该对象就会被重用[JLS, 3.10.5]。

对于同时提供了静态工厂方法（见第1条）和构造函数的非可变类，你通常可以利用静态工厂方法而不是构造函数，以避免创建重复的对象。例如，静态工厂方法Boolean.valueOf(String)几乎总是优先于构造函数Boolean(String)。构造函数在每次被调用的时候都会创建一个新的对象，而静态工厂方法从来不要求这样做。

除了重用非可变的对象之外，对于那些已知不会被修改的可变对象，你也可以重用它们。下面是一个比较微妙、也比较常见的反例，其中涉及到可变对象，它们的值一旦被计算出来之后就不会再有变化。代码如下：

```
public class Person {
    private final Date birthDate;
    // Other fields omitted

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }
    // DON'T DO THIS!
    public boolean isBabyBoomer() {
```

```

        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomStart = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        Date boomEnd = gmtCal.getTime();
        return birthDate.compareTo(boomStart) >= 0 &&
            birthDate.compareTo(boomEnd) < 0;
    }
}

```

`isBabyBoomer`每次被调用的时候，都会创建一个新的`Calendar`、一个新的`TimeZone`和两个新的`Date`实例，这是不必要的。下面的版本用一个静态的初始化器(initializer)，避免了上面例子的低效率：

```

class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = birthDate;
    }

    /**
     * The starting and ending dates of the baby boom.
     */
    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
        Calendar gmtCal =
            Calendar.getInstance(TimeZone.getTimeZone("GMT"));
        gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_START = gmtCal.getTime();
        gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
        BOOM_END = gmtCal.getTime();
    }

    public boolean isBabyBoomer() {
        return birthDate.compareTo(BOOM_START) >= 0 &&
            birthDate.compareTo(BOOM_END) < 0;
    }
}

```

改进版本的`Person`类仅在初始化时刻创建`Calendar`、`TimeZone`和`Date`实例一次，而不是在每次`isBabyBoomer`被调用的时候创建它们。如果`isBabyBoomer`方法被频繁调用的话，则这将会带来显著的性能提高。在我的机器上，每调用一百万次，原来的版本需要36 000ms，而改进的版本只需370ms，大约快了100倍。除了性能提高之外，代码的含义也更加清晰了：把`boomStart`和`boomEnd`从局部变量改为final静态域，使这一点更加清晰：这些日期被作为常量对待，从而使得代码更易于理解。但是，这种优化带来的效果并不总是那么明显，这里是因为`Calendar`实例的创建代价特别昂贵。

如果 `isBabyBoomer` 方法永远也不会被调用，那么 `Person` 类的改进版本就没有必要去初始化 `BOOM_START` 和 `BOOM_END` 域。通过迟缓初始化 (*lazily initializing*) (见第48条) 将对这些域的初始化推迟到 `isBabyBoomer` 方法第一次被调用的时候，则有可能消除这些不必要的初始化工作，但不推荐这样做。如迟缓初始化 (*lazy initialization*) 中常见的情况一样，这样做会使方法的实现更加复杂，从而无法获得性能上的显著提高 (见第37条)。

在本条目前面所有的例子中，很显然，讨论到的对象都能够被重用，因为它们被初始化之后不会再改变。其他有些情形则并不总是这么显然了。考虑适配器 (*adapter*) 的情形 [Gamma95, p. 139]，有时也被称为视图 (*view*)。一个适配器是指这样一个对象：它把功能委托给后面的一个对象，从而为后面的对象提供一个可选的接口。由于适配器除了后面的对象之外，没有其他的状态信息，所以针对某个给定对象的特定适配器而言，它不需要创建多个适配器实例。

例如，`Map` 接口的 `keySet` 方法返回该 `Map` 对象的 `Set` 视图，其中包含该 `Map` 中所有的键 (`key`)。粗看起来，好像每次调用 `keySet` 都应该创建一个新的 `Set` 实例，但是，对于一个给定的 `Map` 对象，每次调用 `keySet` 都返回同样的 `Set` 实例。虽然被返回的 `Set` 实例一般是可改变的，但是所有返回的对象在功能上是等同的：当其中一个返回对象发生变化的时候，所有其他的返回对象也要发生变化，因为它们是由同一个 `Map` 实例支撑的。

不要错误地认为本条目所介绍的内容暗示着“创建对象的代价是非常昂贵的，我们应该要尽可能地避免创建对象”。相反，由于小对象的构造函数只做很少量的工作，所以，小对象的创建和回收动作是非常廉价的，特别是在现代的JVM实现上更是如此。通过创建附加的对象，以使得一个程序更加清晰、简洁、功能强大，这往往也是一件好事。

反之，通过维护自己的对象池 (*object pool*) 来避免对象的创建工作并不是一个好的做法，除非池中的对象是非常重量级的。一个正确使用对象池的典型例子就是数据库连接池。建立数据库连接的代价是非常昂贵的，因此重用这样的对象非常有意义。然而，一般而言，维护自己的对象池会把代码弄得很乱，增加内存占用 (*footprint*)，并且还会损害性能。现代的JVM实现有高度优化的垃圾回收器，其性能很容易就会超过轻量级对象池的性能。

与本条目对应的是第24条中有关“保护性拷贝 (*defensive copying*)”的内容。本条目提及“当你应该重用一个已有的对象的时候，请不要创建新的对象”，而第24条这样说“当你应该创建一个新的对象的时候，请不要重用一个已有的对象”。注意，在提倡使用保护性拷贝的场合，因重用一个对象而招致的代价要远远大于因创建重复对象而招致的代价。在要求保护性拷贝的情况下却没有实施保护性拷贝，将会导致潜在的错误和安全漏洞；而不必要地创建对象仅仅会影响程序的风格和性能。

15

16

第5条：消除过期的对象引用

当你从一种手工管理内存的语言（比如C或C++）转换到一种具有垃圾回收功能的语言的时候，你作为一个程序员，工作会变得更加容易，因为当你用完了对象之后，它们会被自动回收。当你第一次经历对象回收功能的时候，你会觉得这有点不可思议。这很容易会让你留下这样的印象，认为自己不再需要考虑内存管理的事情了。实际上，这是不正确的。

考虑下面简单的栈实现例子：

```
// Can you spot the "memory leak"?
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }

    /**
     * Ensure space for at least one more element, roughly
     * doubling the capacity each time the array needs to grow.
     */
    private void ensureCapacity() {
        if (elements.length == size) {
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

在这个程序中没有很显然的错误。无论你如何测试，它都会成功地通过你的每一项测试，但是，这个程序中潜伏着一个问题。不严格地讲，这个程序有一个“内存泄漏”，随着垃圾回收器活动的增加，或者由于不断增加的内存占用，程序性能的降低会逐渐表现出来。在极端情况下，这样的内存泄漏会导致磁盘分页，甚至导致程序失败（`OutOfMemoryError`错误）。但是，这样的失败情形相对比较少见。

那么，程序中哪里会发生内存泄漏呢？如果一个栈先是增长，然后再收缩，那么，从栈

中弹出来的对象将不会被当做垃圾回收，即使使用栈的客户程序不再引用这些对象，它们也不会被回收。这是因为，栈内部维护着对这些对象的过期引用 (*obsolete reference*)。所谓过期引用，是指永远也不会再被解除的引用。在本例中，凡是在elements数组的“活动区域 (active portion)”之外的引用都是过期的，elements的活动区域是指下标小于size的那一部分。

在支持垃圾回收的语言中，内存泄漏是很隐蔽的（这样的内存泄漏被称为“无意识的对象保持 (*unintentional object retention*)”更为恰当）。如果一个对象引用被无意识地保留起来了，那么，垃圾回收机制不仅不会处理这个对象，而且也不会处理被这个对象引用的所有其他对象。即使只有少量的几个对象引用被无意识地保留下来，也会有很多的对象被排除在垃圾回收机制之外，从而对性能造成潜在的重大影响。

要想修复这一类问题也很简单：一旦对象引用已经过期，只需清空这些引用即可。在上述例子的Stack类中，只要一个单元被弹出栈，指向它的引用就过期了。pop方法的修订版本如下所示：

```
public Object pop() {
    if (size==0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

清空过期引用的另一个好处是，如果它们在以后又被错误地解除引用，则程序会立即抛出NullPointerException异常，而不是悄悄地错误运行下去。尽可能早地检测出程序中的错误总是有益的。

当程序员第一次被类似这样的问题困扰的时候，他们往往会过分小心：对于每一个对象引用，一旦程序不再用到它，就把它清空。这样做既没必要，也不是我们所期望的，因为这样做会把程序代码弄得很乱，并且可以想像还会降低程序的性能。“清空对象引用”这样的操作应该是一种例外，而不是一种规范行为。消除过期引用最好的方法是重用一个本来已经包含对象引用的变量，或者让这个变量结束其生命周期。如果你是在最紧凑的作用域范围内定义每一个变量（见第29条），则这种情形就会自然而然地发生。应该注意到，在目前的JVM实现平台上，仅仅退出定义变量的代码块是不够的，要想使引用消失，必须退出包含该变量的方法。

18

那么，何时应该清空一个引用呢？Stack类的哪方面特性使得它遭受了内存泄漏的影响？简而言之，问题在于，Stack类自己管理内存 (*manage its own memory*)，存储池 (*storage pool*) 包含了elements数组（对象引用单元，而不是对象本身）的元素。数组的活动区域（同前面的定义）中的元素是已分配的 (*allocated*)，而数组其余部分的元素是自由的

(*free*)。但是垃圾回收器并不知道这一点；对于垃圾回收器而言，elements数组中的所有对象引用都同等有效。只有程序员知道数组的非活动区域是不重要的，程序员可以把这个情况告知垃圾回收器，做法很简单：一旦数组元素变成了非活动区域的一部分，程序员就手工清空这些元素。

一般而言，只要一个类自己管理它的内存，程序员就应该警惕内存泄漏问题。一旦一个元素被释放掉，则该元素中包含的任何对象引用应该要被清空。

内存泄漏的另一个常见来源是缓存。一旦你把一个对象引用放到一个缓存中，它就很容易被遗忘掉，从而使得它不再有用之后很长一段时间内仍然留在缓存中。对于这个问题，有两种可能的解决方案。如果你正巧要实现这样的缓存：只要在缓存之外存在对某个条目的键的引用，该条目就有意义，那么你可以使用WeakHashMap来代表缓存；当缓存中的条目过期之后，它们会自动被删除。而更为常见的情形是，“被缓存的条目是否有意义”的周期并不很容易确定，其中的条目会在运行的过程中变得越来越没有价值。在这样的情况下，缓存应该时不时地清除掉无用的条目，这项清除工作可以由一个后台线程（可能要通过java.util.Timer API）来完成，或者也可以在加入新条目的时候做清理工作。在1.4发行版中新加入的java.util.LinkedHashMap类，利用它的removeEldestEntry方法可以很容易地实现后一种方案。

因为内存泄漏通常不会像其他很明显的失败那样自己表现出来，所以，它们可以在一个系统中存在很多年。往往只有通过对代码的仔细检查，或者借助于调试工具（称为*heap profiler*）才能发现内存泄漏问题。因此，如果能够在内存泄漏发生之前就知道如何预测此类问题，并阻止它们，则是最好不过的了。

第6条：避免使用终结函数

终结函数（finalizer）通常是不可预测的，常常也是很危险的，一般情况下是不必要的。使用终结函数会导致不稳定的行为、更差的性能，以及带来移植性问题。当然，终结函数也有其可用之处，我们在本条目的最后再做介绍；但是根据经验，应该避免使用终结函数。

C++的程序员被告知“不要把终结函数当做C++中析构函数（destructors）的对应物”。在C++中，析构函数是回收一个对象所占用资源的常规方法，是构造函数所必需的对应物。在Java程序设计语言中，当一个对象变得不可到达的时候，垃圾回收器会回收与该对象相关联的存储空间，并不需要程序员做专门的工作。C++的析构函数也可以被用来回收其他的非内存资源，而在Java程序设计语言中，一般用try-finally块来完成类似的工作。

终结函数并不能保证会被及时地执行[JLS, 12.6]。从一个对象变得不可到达开始，到它的终结函数被执行，这段时间的长度是任意的、不确定的。这意味着，**时间关键（time-critical）的任务不应该由终结函数来完成**。例如，由终结函数来关闭一个已经被打开的文件，这是严重错误，因为已打开文件的描述符是一种很有限的资源。由于JVM会延迟执行终结函数，所以大量的文件会保留在打开状态，当一个程序不能再打开文件的时候，它可能会运行失败。

及时地执行终结函数正是垃圾回收算法的一个主要功能，这种算法在不同的JVM实现中会大相径庭。如果一个程序依赖于终结函数被执行的时间点，那么这个程序的行为在不同的JVM中运行的表现可能就会截然不同。一个程序在你测试用的JVM平台上运行得非常好，而在你最重要的顾客那里的JVM平台上根本无法运行，这是完全可能的。

延迟终结过程并不只是一个理论问题。在很少见的情况下，为一个类提供一个终结函数，可能会随意地延迟其实例的回收过程。一位同事最近在调试一个长时间运行的GUI应用程序的时候，该应用程序莫名其妙地由于OutOfMemoryError错误而死掉。分析表明，该应用程序死掉的时候，其终结函数队列中有数千个图形对象正在等待被终结和回收。不幸的是，执行终结函数的线程的优先级比该应用程序的其他线程要低得多，所以，图形对象被终结的速度达不到它们进入队列的速度。JLS并不保证哪个线程将会执行终结函数，所以，除了不使用终结函数外，并没有一个可移植的办法能够避免这样的问题。

[20]

JLS不仅不保证终结函数会被及时地执行，而且根本就不保证它们会被执行。当一个程序终止的时候，其中某些已经无法访问的对象上的终结函数根本就没有被执行，这是完全有可能的。结果是，**我们不应该依赖一个终结函数来更新关键性的永久状态**。例如，依赖于终结函数来释放一个共享资源（比如数据库）上的永久锁，是让你整个分布式系统垮掉的好办法。

不要被`System.gc`和`System.runFinalization`这两个方法所诱惑，它们确实增加了终结函数被执行的机会，但是它们并不保证终结函数一定会被执行。唯一声称保证终结函数被执行的方法是`System.runFinalizersOnExit`，以及它臭名昭著的孪生兄弟`Runtime.runFinalizersOnExit`。这两个方法都有致命的缺陷，已经被废弃了。

当你并不确定是否应该避免使用终结函数的时候，这里还有一个值得考虑的情形：如果一种未被捕获的异常会在终结过程中被抛出来，那么这种异常可以被忽略，并且该对象的终结过程也会终止[JLS, 12.6]。未被捕获的异常会使对象处于破坏的状态（a corrupt state），如果另一个线程企图使用这样一个被破坏的对象，则任何不确定的行为都有可能发生。正常情况下，一个未被捕获的异常将会使线程终止，并打印出栈轨迹（stack trace），但是，如果异常发生在一个终结函数之中，则情形不会这样——甚至连警告都不会打印出来。

那么，如果一个类封装的资源（例如文件或者线程）确实需要回收，我们应该怎么办才能不再需要编写终结函数呢？只需提供一个显式的终止方法，并要求该类的客户在每个实例不再有用的时候调用这个方法。一个值得提及的细节是，该实例必须记录下自己是否已经被终止了：显式的终止方法必须在一个私有域中记录下“该对象已经不再有效了”，其他的方法必须检查这个域，如果在对象已经被终止之后，这些方法被调用的话，那么它们应该抛出`IllegalStateException`异常。

显式终止方法的一个典型例子是`InputStream`和`OutputStream`上的`close`方法。另一个例子是`java.util.Timer`上的`cancel`方法，它执行必要的状态改变，使得与`Timer`实例相关联的该线程温和地终止自己。其他的例子还包括`java.awt`中的`Graphics.dispose`和`Window.dispose`。这些方法通常由于性能不好而不被人们关注。一个相关的方法是`Image.flush`，它会释放所有与`Image`实例相关联的资源，但是该实例仍然处于可用的状态，如果有必要的话，会重新分配资源。
21

显式的终止方法通常与try-finally结构结合起来使用，以确保及时终止。在`finally`子句内部调用显式的终止方法可以保证：即使在对象被使用的时候有异常被抛出来，该终止方法也会被执行：

```
// try-finally block guarantees execution of termination method
Foo foo = new Foo(...);
try {
    // Do what must be done with foo
    ...
} finally {
    foo.terminate(); // Explicit termination method
}
```

那么终结函数有什么好处呢？它们有两个合理的用途。第一种用途是，当一个对象的所有

者忘记了调用前面段落中建议的显式终止方法的情况下，终结函数可以充当“安全网（safety net）”。虽然这样做并不能保证终结函数会被及时地调用到，但是在客户无法通过调用显式的终止方法来正常结束操作的情况下（希望这种情形尽可能少发生），迟一点释放关键资源总比永远不释放要好。上文中，作为显式终止方法模式的例子而提到的三个类（InputStream、OutputStream和Timer）也都有终结函数，在终止方法未能被调用的情况下，这些终结函数被用来充当安全网。

终结函数的第二种合理的用途与对象的本地对等体（native peer）有关。本地对等体是一个本地对象（native object），普通对象通过本地方法（native method）委托给一个本地对象。因为本地对等体不是一个普通对象，所以垃圾回收器不会知道它，当它的普通对等体被回收的时候，它不会被回收。在本地对等体并不拥有关键资源的前提下，终结函数正是执行这项任务最合适工具。如果本地对等体拥有必须被及时终止的资源，那么该类应该具有一个显式的终止方法，如前所述。终止方法应该完成必要的工作以便释放关键的资源。终止方法可以是一个本地方法，或者它也可以调用本地方法。

值得注意的重要一点是，“终结函数链（finalizer chaining）”并不会被自动执行。如果一个类（不是Object）有一个终结函数，并且一个子类改写了终结函数，那么子类的终结函数必须要手工调用超类的终结函数。你应该在一个try块中终结子类，并且在对应的finally块中调用超类的终结函数。这样做可以保证，即使子类的终结过程抛出一个异常，超类的终结函数也会被执行，反之亦然。代码示例如下：

```
// Manual finalizer chaining
protected void finalize() throws Throwable {
    try {
        // Finalize subclass state
        ...
    } finally {
        super.finalize();
    }
}
```

22

如果一个子类实现者改写了超类的终结函数，但是忘了手工调用超类的终结函数（或者有意选择不调用超类的终结函数），那么超类的终结函数将永远也不会被调用到。要防范这样粗心大意的或者恶意的子类是有可能的，只要为每一个将被终结的对象创建一个附加的对象。不是把终结函数放在要求终结处理的类中，而是把终结函数放在一个匿名的类（见第18条）中，该匿名类的惟一用途是终结其外围实例（enclosing instance）。该匿名类的单个实例被称为终结函数守卫者（finalizer guardian），外围类的每一个实例都会创建这样一个守卫者。外围实例在它的私有实例域中保存着一个对其守卫者的惟一引用，所以，终结函数守卫者与外围实例可以同时启动终结过程。当守卫者被终结的时候，它执行外围实例所期望的终结行为，就好像它的终结函数是外围对象上的一个方法一样：

```
// Finalizer Guardian idiom
public class Foo {
    // Sole purpose of this object is to finalize outer Foo object
    private final Object finalizerGuardian = new Object() {
        protected void finalize() throws Throwable {
            // Finalize outer Foo object
            ...
        }
    };
    ... // Remainder omitted
}
```

注意，公有类Foo并没有终结函数（除了它从Object中继承了一个无关紧要的之外），所以子类的终结函数是否调用super.finalize并不重要。对于每一个带有终结函数的非final公有类，都应该考虑使用这项技术。
23

总之，除非是作为安全网，或者是为了终止非关键的本地资源，否则请不要使用终结函数。在这些很少见的情况下，既然你使用了终结函数，那么就要记住调用super.finalize。最后，如果你要把一个终结函数与一个公有的非final类关联起来，那么请考虑使用终结函数守卫者，以确保即使子类的终结函数未能调用super.finalize，该终结函数也会被执行。
24

第3章

对于所有对象都通用的方法

尽管Object是一个具体类，但是设计它主要是为了扩展。它的所有非final方法（`equals`、`hashCode`、`toString`、`clone`和`finalize`）都有明确的通用约定（*general contract*），因为它们都是为了要被改写（*override*）而设计的。任何一个类，它在改写这些方法的时候，有责任遵守这些通用约定；如果不能做到这一点，则其他一些依赖于这些约定的类就无法与这些类结合在一起正常运作。

本章将告诉你何时以及如何改写这些非final的Object方法。本章不再讨论`finalize`方法，因为第6条已经讨论过这个方法了。而`Comparable.compareTo`虽然不是Object方法，但是本章也对它进行讨论，因为它有类似的特征。

第7条：在改写`equals`的时候请遵守通用约定

改写`equals`方法看起来非常简单，但是有许多改写的方式会导致错误，并且后果非常严重。要避免问题最容易的办法是不改写`equals`方法，在这种情况下，每个实例只与它自己相等。如果下面的任何一个条件满足的话，这正是所期望的结果：

- 一个类的每个实例本质上都是惟一的。对于代表了活动实体而不是值（`value`）的类，确实是这样的，比如`Thread`。`Object`提供的`equals`实现对于这些类是正确的。
- 不关心一个类是否提供了“逻辑相等（logical equality）”的测试功能。例如，`java.util.Random`改写了`equals`，它检查两个`Random`实例是否产生相同的随机数序列，但是设计者并不认为客户会需要或者期望这样的功能。在这样的情况下，从`Object`继承得到的`equals`实现已经足够了。
- 超类已经改写了`equals`，从超类继承过来的行为对于子类也是合适的。例如，大多数的`Set`实现都从`AbstractSet`继承了`equals`实现，`List`实现从`AbstractList`继

承了equals实现，Map实现从AbstractMap继承了equals实现。

- 一个类是私有的，或者是包级私有的，并且可以确定它的equals方法永远也不会被调用。尽管如此，在这样的情形下，应该要改写equals方法，以免万一有一天它会被调用到。改写如下：

```
public boolean equals(Object o) {
    throw new UnsupportedOperationException();
}
```

那么，什么时候应该改写Object.equals呢？当一个类都有自己特有的“逻辑相等”概念（不同于对象身份的概念），而且超类也没有改写equals以实现期望的行为，这时我们需要改写equals方法。这通常适合于“值类（value class）”的情形，比如Integer或者Date。程序员在利用equals方法来比较两个指向值对象的引用的时候，希望知道它们逻辑上是否相等，而不是它们是否指向同一个对象。为了满足程序员的要求，改写equals方法是必需的，而且这样做也使得这个类的实例可以被用做映射表（map）的键（key）、或者集合（set）的元素，并使映射表或者集合表现出预期的行为。

有一种“值类”可以不要求改写equals方法，即类型安全枚举类型（typesafe enum）（见第21条），因为类型安全枚举类型保证每一个值至多只存在一个对象，所以对于这样的类而言，Object的equals方法等同于逻辑意义上的equals方法。

在改写equals方法的时候，你必须要遵守它的通用约定。下面是约定的内容，来自java.lang.Object的规范：

equals方法实现了等价关系（equivalence relation）：

- 自反性（reflexive）。对于任意的引用值x，x.equals(x)一定为true。
- 对称性（symmetric）。对于任意的引用值x和y，当且仅当y.equals(x)返回true时，x.equals(y)也一定返回true。
- 传递性（transitive）。对于任意的引用值x、y和z，如果x.equals(y)返回true，并且y.equals(z)也返回true，那么x.equals(z)也一定返回true。
- 一致性（consistent）。对于任意的引用值x和y，如果用于equals比较的对象信息没有被修改的话，那么，多次调用x.equals(y)要么一致地返回true，要么一致地返回false。
- 对于任意的非空引用值x，x.equals(null)一定返回false。

除非你对数学特别有兴趣，否则这些规定看起来有点让人恐惧，但是，绝对不要忽视这些规定！如果你违反了它们，你会发现，你的程序将会表现不正常，甚至崩溃，而且很难找到失败的根源。用John Donne的话说，没有哪个类是孤立的。一个类的实例通常会被频繁地传递给另一个类的实例。有许多类，包括所有的集合类（collection classe）在内，都依赖于传递给它们的对象是否遵守了equals约定。

现在你已经知道了违反equals约定有多么可怕，现在我们来更细致地讨论这些约定。值得欣慰的是，这些约定虽然看起来很吓人，实际上并不复杂。一旦你理解了这些约定，要遵守它们并不困难。现在我们按照顺序逐一查看：

自反性 (reflexivity) ——第一个要求仅仅说明一个对象必须等于其自身。很难想像无意识地违反这一条，情形会怎么样。如果你的类违背了这一条，然后你把该类的实例加入到一个集合（collection）中，则该集合的contains方法将果断地告诉你，该集合不包含刚刚你加入的实例。

对称性 (symmetry) ——第二个要求是说，任何两个对象对于“它们是否相等”这个问题必须要保持一致。与第一个要求不同，若无意中违反了这一条，其情形不难想像。例如，考虑下面的类：

```
/*
 * Case-insensitive string. Case of the original string is
 * preserved by toString, but ignored in comparisons.
 */
public final class CaseInsensitiveString {
    private String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }

    // Broken - violates symmetry!
    public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString)o).s);
        if (o instanceof String) // One-way interoperability!
            return s.equalsIgnoreCase((String)o);
        return false;
    }
    ... // Remainder omitted
}
```

27

在这个类中，equals方法的意图非常好，它企图与普通的字符串（String）对象可以互操作。假设我们有一个大小写不敏感的字符串（CaseInsensitiveString）对象和一个普通的字符串：

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

正如所期望的，`cis.equals(s)`返回`true`。问题在于，`CaseInsensitiveString`类中的`equals`方法知道普通的字符串(`String`)对象，但是，`String`类中的`equals`方法却并不知道大小写不敏感的字符串(即`CaseInsensitiveString`对象)。因此，`s.equals(cis)`返回`false`，很显然违反了对称性。假设你把大小写不敏感的字符串对象放到一个集合中：

```
List list = new ArrayList();
list.add(cis);
```

这时候，`list.contains(s)`会返回什么结果？没人知道。在Sun的当前实现中，它碰巧返回`false`，但这只是这个特定实现得出的结果而已。在其他的实现中，它有可能返回`true`，或者抛出一个运行时(`run-time`)异常。一旦你违反了`equals`约定，当其他的对象面对你的对象的时候，你无法知道这些对象的行为会怎么样。

为了解决这个问题，只需把企图与`String`互操作的这段代码从`equals`方法中去掉就可以了。这样做之后，你可以重构代码，使它变成一条返回语句：

```
public boolean equals(Object o) {
    return o instanceof CaseInsensitiveString &&
        ((CaseInsensitiveString)o).s.equalsIgnoreCase(s);
}
```

传递性(transitivity)——`equals`约定的第三个要求是，如果一个对象等于第二个对象，并且第二个对象又等于第三个对象，则第一个对象一定等于第三个对象。同样地，无意识地违反这一条规则的情形也不难想像。考虑这样的情形：一个程序员创建了一个子类，它为超类增加了一个新的特征(*aspect*)。换句话说，子类增加的信息会影响到`equals`的比较结果。我们首先以一个简单的非可变的二维点类作为开始：

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }
    ... // Remainder omitted
}
```

假设你想要扩展这个类，为一个点增加颜色信息：

```
public class ColorPoint extends Point {
    private Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    ... // Remainder omitted
}
```

`equals`方法会怎么样呢？如果你完全不提供`equals`方法，而是直接从`Point`继承过来，那么在`equals`做比较的时候颜色信息被忽略掉了。虽然这样做不会违反`equals`约定，但是很明显这是不可接受的。假设你编写了一个`equals`方法，只有当实参是另一个有色点，并且具有同样的位置和颜色的时候，它才返回`true`：

```
// Broken - violates symmetry!
public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
}
```

29

这个方法的问题在于，你在比较一个普通点和一个有色点，以及反过来的情形的时候，可能会得到不同的结果。前一种比较忽略了颜色信息，而后一种比较总是返回`false`，因为实参的类型不正确。为了直观地说明问题所在，我们创建一个普通点和一个有色点：

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

然后，`p.equals(cp)`返回`true`，而`cp.equals(p)`返回`false`。你可以做这样的尝试来修正这个问题：让`ColorPoint.equals`在进行“混合比较”的时候忽略颜色信息：

```
// Broken - violates transitivity.
public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;

    // If o is a normal Point, do a color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);

    // o is a ColorPoint; do a full comparison
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.color == color;
}
```

这种方法确实提供了对称性，但是却牺牲了传递性：

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

此时，`p1.equals(p2)`和`p2.equals(p3)`都返回`true`，但是`p1.equals(p3)`返回`false`，很显然违反了传递性。前两个比较不考虑颜色信息（“色盲”），而第三个比较考虑了颜色信息。

怎么解决呢？事实上，这是面向对象语言中关于等价关系的一个基本问题。要想在扩展一个可实例化的类的同时，既要增加新的特征，同时还要保留`equals`约定，没有一个简单的方法可以做到这一点。30然而，根据第14条的建议：复合优先于继承，这个问题还是有好的解决办法。我们不再让`ColorPoint`扩展`Point`，而是在`ColorPoint`中加入一个私有的`Point`域，以及一个公有的视图（`view`）方法（见第4条），此方法返回一个与该有色点在同一位置上的普通`Point`对象：

```
// Adds an aspect without violating the equals contract
public class ColorPoint {
    private Point point;
    private Color color;

    public ColorPoint(int x, int y, Color color) {
        point = new Point(x, y);
        this.color = color;
    }

    /**
     * Returns the point-view of this color point.
     */
    public Point asPoint() {
        return point;
    }

    public boolean equals(Object o) {
        if (!(o instanceof ColorPoint))
            return false;
        ColorPoint cp = (ColorPoint)o;
        return cp.point.equals(point) && cp.color.equals(color);
    }

    ... // Remainder omitted
}
```

在Java平台库中，有一些类是可实例化类的子类，并且加入了新的特征。例如，`java.sql.Timestamp`对`java.util.Date`进行子类化，并且增加了`nanoseconds`域，`Timestamp`的`equals`实现确实违反了对称性，如果`Timestamp`和`Date`对象被用于同一个集合中，或者以其他方式被混合在一起，则会出现不正确的行为。`Timestamp`类有一个否认声明，告诫程序员不要混合使用`Date`和`Timestamp`对象。只要你不把它们混合在一起，你就不会陷入麻烦，除此之外没有其他的措施可以防止你这么做，而且结果导致的错误很难调

试。Timestamp类是一个反常的类，不值得仿效。

注意，你可以在一个抽象(*abstract*)类的子类中增加新的特征，而不会违反*equals*约定。这一点对于根据第20条的建议“用类层次(class hierarchies)来代替联合(union)”而得到的一种类层次结构非常重要。例如，你可能有一个抽象的Shape类，它内部没有任何特征信息，Circle子类加入了一个radius域，Rectangle子类加入了length和width域。31只要不可能创建超类的实例，那么前面所述的种种问题都不会发生。

一致性(consistency)——*equals*约定的第四个要求是，如果两个对象相等的话，那么它们必须始终保持相等，除非有一个对象(或者两个都)被修改了。由于可变对象在不同的时候可以与不同的对象相等，而非可变对象不会这样，所以，这一条作为提醒实际上算不上是*equals*方法的要求。当你在写一个类的时候，应该仔细考虑它是否为非可变的(见第13条)，如果认为它应该是非可变的，那么你就必须要保证*equals*方法满足这样的限制条件：相等的对象永远相等，不相等的对象永远不相等。

非空性(Non-nullity)——最后一个要求没有名称，我就将它称为“非空性(Non-nullity)”，意思是指所有的对象都必须不等于null。尽管很难想像什么情况下o.equals(null)会返回true，但是抛出NullPointerException异常的情形并不难想像。通用约定不允许抛出NullPointerException异常。许多类的*equals*方法通过一个显式的null测试来防止这种情况：

```
public boolean equals(Object o) {
    if (o == null)
        return false;
    ...
}
```

这项测试不是必需的。为了测试实参与当前对象的相等情况，*equals*方法必须首先把实参转换为一种适当的类型，以便可以调用它的访问方法(accessor)或者访问它的域。在做转换之前，*equals*方法必须使用instanceof操作符，检查它的实参是否为正确的类型：

```
public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    ...
}
```

如果漏掉了这一步类型检查，并且传递给*equals*方法的实参又是错误的类型，那么*equals*方法将会抛出一个ClassCastException异常，这违反了*equals*的约定。但是，如果instanceof的第一个操作数为null的话，那么，不管第二个操作数是哪种类型，按instanceof操作符的规定，它应该返回false[JLS, 15.19.2]。因此，如果把null传给*equals*方法的话，则类型检查的结果为false，所以，你并不需要做单独的null检查。把

32 所有这些合在一起，下面是为实现高质量equals方法的一个“处方”：

1. 使用`==`操作符检查“实参是否为指向对象的一个引用”。如果是的话，则返回`true`，这只不过是一种性能优化，如果比较操作有可能非常耗时的话，这样做是值得的。
2. 使用`instanceof`操作符检查“实参是否为正确的类型”。如果不是的话，则返回`false`。通常，这里“正确的类型”是指`equals`方法所在的那个类。有些情况下，是指该类所实现的某个接口。如果一个类实现的一个接口改进了`equals`约定，允许在实现了该接口的类之间进行比较，那么使用这个接口作为正确的类型。集合接口（collection interface）`Set`、`List`、`Map`和`Map.Entry`具有这样的特点。
3. 把实参转换到正确的类型。因为前面已经有了`instanceof`测试，所以这个转换可以确保成功。
4. 对于该类中每一个“关键（significant）”域，检查实参中的域与当前对象中对应的域值是否匹配。如果所有的测试都成功，则返回`true`；否则返回`false`。如果第2步中的类型是一个接口，那么你必须通过接口的方法，访问实参中的关键域；如果该类型是一个类，那么你也许能够直接访问实参中的关键域，这要取决于它们的可访问性。对于既不是`float`也不是`double`类型的原语类型域，可以使用`==`操作符进行比较；对于对象引用域，可以递归地调用`equals`方法；对于`float`域，先使用`Float.floatToIntBits`转换成`int`类型的值，然后使用`==`操作符比较`int`类型的值；对于`double`域，先使用`Double.doubleToLongBits`转换成`long`类型的值，然后使用`==`操作符比较`long`类型的值（这里，对`float`和`double`域进行特殊的处理是有必要的，因为存在`Float.NaN`、`-0.0f`以及类似的`double`类型的常量；细节请参考`Float.equals`的文档）。对于数组域，把以上这些指导原则应用到每个元素上。有些对象引用域包含`null`是合法的，所以，为了避免可能导致`NullPointerException`异常，使用下面的习惯用法来比较这样的域：

```
(field == null ? o.field == null : field.equals(o.field))
```

如果`field`和`o.field`通常是相同的对象引用，那么下面的做法会更快一些：

```
(field == o.field || (field != null && field.equals(o.field)))
```

对于有些类，比如前面提到的`CaseInsensitiveString`类，针对每个域的比较操作比简单的相等测试要复杂得多。如果是这样的情形，应该在该类的规范上明确加以说明。如果确实是这样的话，你可能会期望在每一种对象内部保存一个“范式（canonical form）”，这样`equals`方法可以根据这些范式进行低开销的精确比较，而不是高开销的非精确比较。这项技术对于非可变类（见第13条）是最为合适的，因为如果对象发生变化的话，其范式也必须相应地更新。

域的比较顺序可能会影响到`equals`方法的性能。为了获得最佳的性能，最先进行比较的域应该是最有可能不一致的域，或者是比较开销最低的域，理想情况是两个条件同时满足的域。你不应该去比较那些不属于对象逻辑状态的域，比如`Object`中用于同步操作的域。你不需要比较冗余域 (*redundant field*)，因为这些冗余域可以由“关键域”计算获得，但是这样做也有可能提高`equals`方法的性能。如果一个冗余域代表了这整个对象的一个概括描述，那么，当最终比较结果为`false`时，通过比较这些冗余域，可以省下比较实际数据所需要的开销。

5. 当你编写完成了`equals`方法之后，应该问自己三个问题：它是否是对称的、传递的、一致的？（其他两个特性通常会自行满足。）如果答案是否定的，那么请找到这些特性未能满足的原因，再修改`equals`方法的代码。

根据上面的“处方”构建的`equals`方法的具体例子，请参看第8条的`PhoneNumber.equals`。下面是最后的一些告诫：

- 当你改写`equals`的时候，总是要改写`hashCode`（见第8条）。
- 不要企图让`equals`方法过于聪明。如果只是简单地测试域中的值是否相等，则不难做到遵守`equals`约定。如果想过度地去寻求各种等价关系，则很容易陷入麻烦之中。把任何一种别名形式考虑在等价的范围内，往往不会是一个好主意。例如，作为`File`类，它不应该试图把指向同一个文件的符号链接（symbolic link）当作相等的对象来看待。谢天谢地，`File`类没有这样做。
- 不要使`equals`方法依赖于不可靠的资源。如果你这样做的话，要想满足一致性要求是非常困难的。例如，`java.net.URL`的`equals`方法依赖于被比较的URL中主机的IP地址。把一个主机名转换为一个IP地址，这项工作需要访问网络，而且不保证每次都会产生同样的结果。这样会导致URL的`equals`方法违反`equals`约定，在实践中有可能会引发问题。（不幸的是，由于兼容性的要求，这种行为不可能被改变。）除了少数的例外，`equals`方法应该针对驻留在内存中的对象执行确定性的计算。
- 不要将`equals`声明中的`Object`对象替换为其他的类型。程序员编写出下面这样的`equals`方法并不鲜见，这会使程序员数个小时都搞不清为什么它不能正常工作：

```
public boolean equals(MyClass o) {
    ...
}
```

问题在于，这个方法并没有改写（*override*）`Object.equals`，因为`Object.equals`

的实参应该是Object类型，相反，它重载（*overload*）了Object.equals（见第26条）。在原有equals方法的基础上，再提供一个“强类型化（*strongly typed*）”的equals方法，只要这两个方法返回同样的结果（没有强制的理由必须这样做），那么这是可以接受的。在某些特定的情况下，它也许会带来一些性能上的提高，但是与增加的复杂性相比，这种做法是不值得的（见第37条）。

第8条：改写equals时总是要改写hashCode

一个很常见的错误根源在于没有改写hashCode方法。在每个改写了equals方法的类中，你必须也要改写hashCode方法。如果不这样做的话，就会违反Object.hashCode的通用约定，从而导致该类无法与所有基于散列值（hash）的集合类结合在一起正常运作，这样的集合类包括HashMap、HashSet和Hashtable。

下面是hashCode约定的内容，来自java.lang.Object的规范：

- 在一个应用程序执行期间，如果一个对象的equals方法做比较所用到的信息没有被修改的话，那么，对该对象调用hashCode方法多次，它必须始终如一地返回同一个整数。在同一个应用程序的多次执行过程中，这个整数可以不同，即这个应用程序这次执行返回的整数与下一次执行返回的整数可以不一致。
- 如果两个对象根据equals(Object)方法是相等的，那么调用这两个对象中任一个对象的hashCode方法必须产生同样的整数结果。
- 如果两个对象根据equals(Object)方法是不相等的，那么调用这两个对象中任一个对象的hashCode方法，不要求必须产生不同的整数结果。然而，程序员应该意识到这样的事实，对于不相等的对象产生截然不同的整数结果，有可能提高散列表（hash table）的性能。

因没有改写hashCode而违反的关键约定是第二条：相等的对象必须具有相等的散列码（hash code）。根据一个类的equals方法，两个截然不同的实例有可能在逻辑上是相等的，但是，根据Object类的hashCode方法，它们仅仅是两个对象，没有其他共同的地方。因此，对象的hashCode方法返回两个看起来是随机的整数，而不是根据第二个约定所要求的那样，返回两个相等的整数。

例如，考虑下面极为简单的PhoneNumber类，它的equals方法是根据第7条中给出的“处方”构造出来的：

```
public final class PhoneNumber {
    private final short areaCode;
    private final short exchange;
    private final short extension;

    public PhoneNumber(int areaCode, int exchange,
                      int extension) {
        rangeCheck(areaCode, 999, "area code");
        rangeCheck(exchange, 999, "exchange");
        rangeCheck(extension, 9999, "extension");
    }
```

```

        this.areaCode = (short) areaCode;
        this.exchange = (short) exchange;
        this.extension = (short) extension;
    }

    private static void rangeCheck(int arg, int max,
                                   String name) {
        if (arg < 0 || arg > max)
            throw new IllegalArgumentException(name +": " + arg);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.extension == extension &&
               pn.exchange == exchange &&
               pn.areaCode == areaCode;
    }

    // No hashCode method!
    ...
}

```

假设你企图将这个类与HashMap一起使用：

```

Map m = new HashMap();
m.put(new PhoneNumber(408, 867, 5309), "Jenny");

```

这时候，你可能会期望`m.get(new PhoneNumber(408, 867, 5309))`会返回“Jenny”，但是它实际上返回null。注意，这里涉及到两个PhoneNumber实例：第一个被用于插入到HashMap中，第二个实例与第一个相等，被用于（试图用于）检索。由于PhoneNumber类没有改写hashCode方法，从而导致两个相等的实例具有不相等的散列码，违反了hashCode的约定。因此，put方法把Jenny的电话号码对象存放在一个散列桶（hash bucket）中，而get方法会在另一个散列桶中查找她的电话号码对象。要想修正这个问题非常简单，只需为PhoneNumber类提供一个适当的hashCode方法即可。

37

那么，hashCode方法应该是什么样的呢？编写一个合法但并不好用的hashCode方法没有任何价值。例如，下面这个方法总是合法的，但是永远也不应该被正式使用：

```

// The worst possible legal hash function - never use!
public int hashCode() { return 42; }

```

上面这个hashCode方法是合法的，因为相等的对象总是具有同样的散列码。但它也极为恶劣，因为它使得每一个对象都具有同样的散列码。因此，每个对象都被映射到同一个散列桶中，从而散列表被退化为链表（linked list）。它使得本该线性时间运行的程序变成了平方

运行时间。对于规模很大的散列表而言，这关系到散列表能否正常工作。

一个好的散列函数通常倾向于“为不相等的对象产生不相等的散列码”。这正是 hashCode 约定中第三条的含义。理想情况下，一个散列函数应该把一个集合中不相等的实例均匀地分布到所有可能的散列值上。要想完全达到这种理想的情形是非常困难的，幸运的是，相对接近这种理想情形并不太困难。下面给出一种简单的“处方”：

1. 把某个非零常数值，比如说 17，保存在一个叫 result 的 int 类型的变量中。
2. 对于对象中每一个关键域 f（指 equals 方法中考虑的每一个域），完成以下步骤：
 - a. 为该域计算 int 类型的散列码 c：
 - i. 如果该域是 boolean 类型，则计算 ($f ? 0 : 1$)。
 - ii. 如果该域是 byte、char、short 或者 int 类型，则计算 ($\text{int})f$ 。
 - iii. 如果该域是 long 类型，则计算 ($\text{int})(f \wedge (f >>> 32))$ 。
 - iv. 如果该域是 float 类型，则计算 $\text{Float.floatToIntBits}(f)$ 。
 - v. 如果该域是 double 类型，则计算 $\text{Double.doubleToLongBits}(f)$ 得到一个 long 类型的值，然后按照步骤 2.a.iii，对该 long 型值计算散列值。
 - vi. 如果该域是一个对象引用，并且该类的 equals 方法通过递归调用 equals 的方式来比较这个域，则同样对这个域递归调用 hashCode。如果要求一个更为复杂的比较，则为这个域计算一个“规范表示（canonical representation）”，然后针对这个范式表示调用 hashCode。如果这个域的值为 null，则返回 0（或者其他某个常数，但习惯上使用 0）。
 - b. 按照下面的公式，把步骤 a 中计算得到的散列码 c 组合到 result 中：


```
result = 37 * result + c;
```
3. 返回 result。
4. 写完了 hashCode 方法之后，问自己“是否相等的实例具有相等的散列码”。如果不是的话，找出原因，并修正错误。

在散列码的计算过程中，把冗余域（*redundant field*）排除在外是可以接受的。换句话说，如果一个域的值可以根据参与计算的其他域值计算出来，则把这样的域排除在外是可以接受的。对于在相等比较计算中没有被用到的任何域，你要把它们排除在外，这是一个要求。如果不这样做的话，可能会导致违反hashCode约定的第二条。

上面步骤1中用到了一个非零的初始值，所以，对于在步骤2.a中计算的散列值为0的那些初始域，它们也会影响到散列值。如果步骤1中的初始值为0，则整个散列值将不受这些初始域的影响，从而会增加冲突的可能性。值17是任选的。

步骤2.b中的乘法部分使得散列值依赖于域的顺序，如果一个类包含多个相似的域，那么这样的乘法运算会产生一个更好的散列函数。例如，如果String类也根据上面的步骤来建立散列函数，并且把乘法部分省去，则那些仅仅是字母顺序不同的所有字符串，都会有同样的散列码。之所以选择37，是因为它是一个奇素数。如果乘数是偶数，并且乘法溢出的话，则信息会丢失，因为与2相乘等价于移位运算。使用素数的好处并不很明显，但是习惯上使用素数来计算散列结果。

现在我们把这种方法用到PhoneNumber类中。它有三个关键域，都是short类型。根据上面的步骤，很直接地会得到下面的散列函数：

```
public int hashCode() {
    int result = 17;
    result = 37*result + areaCode;
    result = 37*result + exchange;
    result = 37*result + extension;
    return result;
}
```

39

因为这个方法返回的结果是一个简单的、确定的计算结果，它的输入只是PhoneNumber实例中的三个关键域，所以，很清楚，相等的PhoneNumber会有相等的散列码。实际上，对于PhoneNumber的hashCode实现而言，上面这个方法是非常合理的，等同于Java平台库1.4发行版本中的实现。它的做法非常简单，速度也非常快，恰当地把不相等的电话号码分散到不同的散列桶中。

如果一个类是非可变的，并且计算散列码的代价也比较大，那么你应该考虑把散列码缓存在对象内部，而不是每次请求的时候都重新计算散列码。如果你觉得这种类型的大多数对象会被用做散列键（hash keys），那么你应该在实例被创建的时候就计算散列码。否则的话，你可以选择“迟缓初始化”散列码，一直到hashCode被第一次调用的时候才初始化（见第48条）。现在尚不清楚我们的PhoneNumber类是否值得这样处理，但可以通过它来说明这种方法该如何实现：

```

// Lazily initialized, cached hashCode
private volatile int hashCode = 0; // (See Item 48)

public int hashCode() {
    if (hashCode == 0) {
        int result = 17;
        result = 37*result + areaCode;
        result = 37*result + exchange;
        result = 37*result + extension;
        hashCode = result;
    }
    return hashCode;
}

```

虽然本条目中前面给出的`hashCode`实现方法能够获得相对比较好的散列函数，但是它并不能产生最新的散列函数，并且Java平台库1.4发行版本也没有提供这样最新的散列函数。“编写这种最佳散列函数”是一个目前非常活跃的研究课题，应该留给数学家和理论方面的计算机科学家来完成。也许Java平台的下一个发行版本将会为它的类提供这种最佳的散列函数，并提供一些实用方法来帮助普通程序员构造出这样的散列函数。与此同时，本条目中介绍的技术对于绝大多数应用程序而言已经足够了。

不要试图从散列码计算中排除掉一个对象的关键部分以提高性能。虽然这样得到的散列函数运行起来可能非常快，但是它的效果不见得会好，可能会导致散列表慢得根本不可用。特别是在实践中，散列函数可能会面临着大量的实例，而且，在你选择可以忽略的区域之中，这些实例仍然区别非常大。如果是这样的话，散列函数会把所有这些实例映射到非常少量的散列码上，基于散列的集合将会表现出平方级的性能指标。这不仅仅是一个理论问题，在Java 1.2发行版本之前，`String`类的散列函数至多只检查16个字符，从第一个字符开始，在整个字符串中均匀选取。对于像URL这样的层次状名字的大型集合，该散列函数正好表现了这里所提到的病态行为。[40]

Java平台库中的许多类，比如`String`、`Integer`和`Date`，可以把它们的`hashCode`方法返回的确切值规定为该实例值的一个函数。通常而言，这并不是一个好主意，因为这样做严格地限制了在将来的版本中改进散列函数的能力。如果你没有规定散列函数的细节，那么当你发现了它内部的缺陷的时候，就可以在下一个版本中修正它，而无需担心打破与客户之间的兼容性（这些客户依赖于散列函数返回的确切值）。[41]

第9条：总是要改写toString

虽然java.lang.Object提供了toString方法的一个实现，但是，它返回的字符串通常并不是类的用户所期望看到的，它包含类的名字，以及一个“@”符号，接着是散列码的无符号十六进制表示，例如“PhoneNumber@163b91”。toString的通用约定指出，被返回的字符串应该是一个“简洁的，但信息丰富，并且易于阅读的表达形式”。尽管“PhoneNumber@163b91”算得上是简洁的和易于阅读的，但是与“(408) 867-5309”比较起来，它不是信息丰富的。toString的约定进一步指出，“建议所有的子类都改写这个方法。”很好的建议，真的。

虽然遵守toString的约定并不像遵守equals和hashCode的约定（见第7和第8条）那么重要，但是，提供一个好的toString实现，可以使一个类用起来更加愉快。当一个对象被传递给println、字符串连接操作符（+）以及1.4发行版本中的assert的时候，toString方法会自动被调用。如果你提供了一个好的toString方法，那么，要产生一个有用的诊断消息会非常容易，如下：

```
System.out.println("Failed to connect: " + phoneNumber);
```

不管是否改写了toString方法，程序员都将会按这种方式来产生诊断消息，但是如果你没有改写toString方法的话，产生的消息将难以理解。提供一个好的toString方法，不仅有益于这个类的实例，同样也有益于那些“包含了指向这些实例的引用”的对象，特别是集合对象。下面这两条消息：“Jenny=PhoneNumber@163b91”和“Jenny=(408) 867-5309”，你更愿意看到哪一个？

在实际应用中，toString方法应该返回对象中包含的所有令人感兴趣的信息，譬如上面的电话号码例子那样。如果对象太大，或者对象中包含的状态信息难以用字符串来表达，这样做就有点不切实际。在这样的情况下，toString应该返回一个摘要信息，比如“Manhattan white pages (1487536 listings)”或者“Thread[main, 5, main]”。理想情况下，字符串应该是自描述的(self-explanatory)，(Thread例子不满足这样的要求。)

在实现toString的时候，你必须要做出的一个很重要的决定是，是否在文档中指定返回值的格式。对于值类(*value class*)，比如电话号码类、矩阵类，推荐这样做。指定格式的好处是，它可以被用做一种标准的、无二义性的、适合人阅读的对象表达形式。这种表达形式可以用于输入和输出以及用在永久性的适合于人阅读的数据对象中，如XML文档。如果你指定了格式，一个好的做法是同时也提供一个相匹配的String构造函数（或者静态工厂，见

第1条), 这样, 程序员可以很容易地在对象和它的字符串表示之间来回转换。Java平台库中的许多值类都采用了这种做法, 包括 BigInteger、BigDecimal 和绝大多数的原语类型包装类 (wrapper class)。

指定 `toString` 返回值的格式也有不足之处: 如果这个类已经被广泛使用了, 则一旦你指定了格式, 则必须始终如一地坚持这种格式。程序员将会编写出相应的代码来解析这种字符串表示、产生字符串表示, 以及把字符串表示嵌入到永久数据中。如果在将来的发行版本中改变了表达形式, 则会打破他们的代码和数据, 他们当然会抱怨。如果不指定格式的话, 你可以保留灵活性, 便于在将来的版本中增加信息, 或者改进格式。

不管你是否决定指定格式, 都应该在文档中明确地表明你的意图。如果你要指定格式, 则应该严格地这样去做。例如, 下面是第8条中 `PhoneNumber` 类的 `toString` 方法:

```
/*
 * Returns the string representation of this phone number.
 * The string consists of fourteen characters whose format
 * is "(XXX) YYY-ZZZZ", where XXX is the area code, YYY is
 * the exchange, and ZZZZ is the extension. (Each of the
 * capital letters represents a single decimal digit.)
 *
 * If any of the three parts of this phone number is too small
 * to fill up its field, the field is padded with leading zeros.
 * For example, if the value of the extension is 123, the last
 * four characters of the string representation will be "0123".
 *
 * Note that there is a single space separating the closing
 * parenthesis after the area code from the first digit of the
 * exchange.
 */
public String toString() {
    return "(" + toPaddedString(areaCode, 3) + ") " +
        toPaddedString(exchange, 3) + "-" +
        toPaddedString(extension, 4);
}

/*
 * Translates an int to a string of the specified length,
 * padded with leading zeros. Assumes i >= 0,
 * 1 <= length <= 10, and Integer.toString(i) <= length.
 */
private static String toPaddedString(int i, int length) {
    String s = Integer.toString(i);
    return ZEROS[length - s.length()] + s;
}

private static String[] ZEROS =
    {"", "0", "00", "000", "0000", "00000",
     "000000", "0000000", "00000000", "000000000"};
```

43

如果你决定不指定格式, 那么文档注释部分也应该有如下所示的指示信息:

```
/*
 * Returns a brief description of this portion. The exact details
```

```
* of the representation are unspecified and subject to change,  
* but the following may be regarded as typical:  
*  
* "[Potion #9: type=love, smell=turpentine, look=india ink]"  
*/  
public String toString() { ... }
```

对于那些依赖于格式的细节进行编程或者产生永久数据的程序员，在读到这段注释之后，一旦格式被改变，则只能自己承担后果。

无论是否指定格式，为 `toString` 返回值中包含的所有信息，提供一种编程访问途径，这总是一个好的做法。例如，`PhoneNumber`类应该包含针对 `areaCode`、`exchange` 和 `extension` 的访问方法。如果不这样做的话，就迫使那些需要这些信息的程序员不得不自己去解析这些字符串。除了降低了程序的性能，并使得程序员们去做这些不必要的工作之外，这个解析过程也很容易出错，会导致系统不稳定，如果格式变化了的话，还会导致系统崩溃。如果没有提供这些访问方法，即使你已经指明了字符串的格式是可以变化的，字符串格式也成了事实上的API。

第10条：谨慎地改写clone

Cloneable接口的目的是作为对象的一个mixin接口 (*mixin interface*) (见第16条)，表明这样的对象允许克隆 (*cloning*)。不幸的是，它并没有成功地达到这个目的。其主要的缺陷在于，它缺少一个clone方法，Object的clone方法是被保护的。如果不借助于映像机制 (*reflection*) (见第35条)，则不能仅仅因为一个对象实现了Cloneable，就可以调用clone方法。即使是映像调用也可能会失败，因为并不保证该对象一定具有可访问的clone方法。尽管存在这样那样的缺陷，这项设施仍然被广泛地使用着，值得我们进一步了解它。本条目将告诉你如何实现一个行为良好的clone方法，并讨论何时这样做是恰当的，同时也简单地讨论了其他的可替换做法。

那么，既然Cloneable并没有包含任何方法，那么它到底做了什么呢？它决定了Object中受保护的clone方法实现的行为：如果一个类实现了Cloneable，则Object的clone方法返回该对象的逐域拷贝，否则的话，它抛出一个CloneNotSupportedException异常。这是接口的一种极端非典型的用法，也不值得仿效。通常情况下，实现一个接口是为了表明一个类可以为客户做某些事情。然而，对于Cloneable接口，它改变了超类中一个受保护的方法的行为。

为了实现Cloneable接口，使它对一个类确实产生效果，它和所有的超类都必须遵守一个相当复杂的、不可实施的，并且基本上没有文档说明的协议。由此得到一种语言本身之外的 (*extralinguistic*) 机制：无须调用构造函数就可以创建一个对象。

Clone方法的通用约定是非常弱的，下面是来自java.lang.Object规范中的约定内容：

创建和返回该对象的一个拷贝，这里“拷贝”的精确含义取决于该对象的类。一般的含义是，对于任何对象x，表达式

`x.clone() != x`

将会是true，并且，表达式

`x.clone().getClass() == x.getClass()`

将会是true，但是，这些都不是绝对的要求。虽然，通常情况下，表达式

`x.clone().equals(x)`

将会是true，但是，这也不是一个绝对要求。拷贝一个对象往往会导致创建该类的一个新实例，但同时它也会要求拷贝内部的数据结构。这个过程中没有调用构造函数。

这个约定存在几个问题。规定“没有调用构造函数”太强了。一个行为良好的clone方法可以调用构造函数来创建对象，构造之后再复制内部数据。如果这个类是final的，则clone甚至可能会返回一个由构造函数创建的对象。

然而，规定“`x.clone().getClass()通常应该等同于x.getClass()`”则太弱了。在实践中，程序员会假设：如果他们扩展了一个类，并且从子类中调用了`super.clone`，则返回的对象将是该子类的实例。超类能够提供这种功能的唯一途径是，返回一个通过调用`super.clone`而得到的对象。如果一个clone方法返回一个由构造函数创建的对象，它将具有错误的类。因此，如果你改写了一个非final类的clone方法，则应该返回一个通过调用`super.clone`而得到的对象。如果一个类的所有超类都遵守这条规则，那么一直调用`super.clone`，最终会调用到Object的clone方法，从而创建出正确的类的实例。这种机制大致上类似于自动的构造函数链，只不过它不是强制要求的。

在1.3发行版本中，Cloneable接口并没有清楚地指明，一个类在实现这个接口时应该承担哪些责任。规范仅仅指出了：实现这个接口会以哪些方式影响到Object的clone实现的行为。实际上，对于实现了cloneable的类，我们总是期望它也提供了一个功能适当的公有clone方法。但通常情况下，除非该类的所有超类都提供了一个行为良好的clone实现，不管是公有的，还是受保护的，否则，这是不可能的。

假设你希望在一个类中实现Cloneable，并且它的超类都提供了行为良好的clone方法。你从`super.clone()`中得到的对象可能会接近于最终要返回的对象，也可能相差甚远，这要取决于这个类的本质。从每一个超类的角度来看，这个对象将是原始对象功能完整的克隆(clone)。在这个类中声明的域（如果有的话）将等同于被克隆对象中相应的域值。如果每个域包含一个原语类型的值，或者包含一个指向非可变对象的引用，那么被返回的对象可能正是你所需要的对象，在这种情况下不需要再做进一步处理。例如，第8条中的PhoneNumber类正是这样的情形。在这种情况下，你所需要做的是对Object中受保护的clone方法提供一条公有的访问途径：

```
public Object clone() {
    try {
        return super.clone();
    } catch(CloneNotSupportedException e) {
        throw new Error("Assertion failure"); // Can't happen
    }
}
```

46

然而，如果对象中包含的域引用了可变的对象，那么使用这样的clone实现可能会导致灾难性后果。例如，考虑第5条中的Stack类：

```
public class Stack {
```

```

private Object[] elements;
private int size = 0;

public Stack(int initialCapacity) {
    this.elements = new Object[initialCapacity];
}

public void push(Object e) {
    ensureCapacity();
    elements[size++] = e;
}

public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}

// Ensure space for at least one more element.
private void ensureCapacity() {
    if (elements.length == size) {
        Object oldElements[] = elements;
        elements = new Object[2 * elements.length + 1];
        System.arraycopy(oldElements, 0, elements, 0, size);
    }
}
}

```

假设你希望把这个类做成可克隆的（cloneable）。如果它的clone方法仅仅返回super.clone()，那么这样得到的Stack实例，在其size域中具有正确的值，但是它的elements域将引用到与原始Stack实例中相同的数组上。修改原始的实例会破坏被克隆对象中的数组，反之亦然。很快你就会发现，这个程序将产生毫无意义的结果，或者抛出NullPointerException异常。

47

如果调用Stack类惟一的构造函数，那么这种情况永远不会发生。实际上，clone方法是另一个构造函数；你必须确保它不会伤害到原始的对象，并且正确地建立起被克隆对象中的约束关系（invariant）。为了使Stack类中的clone方法正常地工作，它必须要拷贝栈的内部信息。最容易的做法是，对于elements数组递归地调用clone：

```

public Object clone() throws CloneNotSupportedException {
    Stack result = (Stack) super.clone();
    result.elements = (Object[]) elements.clone();
    return result;
}

```

注意，如果elements域是final的，则这种方案并不能正常工作，因为clone方法是被禁止给elements域赋一个新值的。这是一个基本问题：clone结构与指向可变对象的final域的正常用法是不兼容的。除非在原始对象和克隆对象之间可以安全地共享此可变对象。

为了使一个类成为可克隆的，可能有必要从某些域中去掉final修饰符。

递归地调用clone往往还不够。例如，假设你正在为一个散列表编写clone方法，它的内部数据是由一个散列桶数组组成，每一个散列桶都指向“键-值”对链表的第一个条目，如果桶是空的，则为null。出于性能的考虑，该类实现了它自己的单向链表，而没有使用Java内部的java.util.LinkedList。该类如下：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    ... // Remainder omitted
}
```

48

假设你仅仅递归地克隆这个散列桶数组，就像我们为Stack类所做的那样：

```
// Broken - results in shared internal state!
public Object clone() throws CloneNotSupportedException {
    HashTable result = (HashTable) super.clone();
    result.buckets = (Entry[]) buckets.clone();
    return result;
}
```

虽然被克隆的对象有它自己的散列桶数组，但是，这个数组引用的链表与原始对象是一样的，从而很容易引起克隆对象和原始对象之中的不确定行为。为了修正这个问题，你必须为每一个组成桶的链表单独地拷贝。下面是一种常见的做法：

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    // Recursively copy the linked list headed by this Entry
}
```

```

Entry deepCopy() {
    return new Entry(key, value,
        next == null ? null : next.deepCopy());
}
}

public Object clone() throws CloneNotSupportedException {
    HashTable result = (HashTable) super.clone();
    result.buckets = new Entry[buckets.length];
    for (int i = 0; i < buckets.length; i++)
        if (buckets[i] != null)
            result.buckets[i] = (Entry)
                buckets[i].deepCopy();

    return result;
}
... // Remainder omitted
}

```

49

私有类`HashTable.Entry`被加强了，它支持一个“深度拷贝（deep copy）”方法。`HashTable`上的`clone`方法分配了一个适当大小的、新的`buckets`数组，并且遍历原始的`buckets`数组，对每一个非空散列桶进行深度拷贝。`Entry`类中的深度拷贝方法递归地调用它自己，以便拷贝整个链表（它是链表的头节点）。虽然这项技术手法很漂亮，并且，如果散列桶不是很长的话，也会工作得很好，但是，这样克隆一个链表并不是一个好的方案，因为针对链表中的每个元素，它都要消耗一段栈空间。如果链表比较长的话，这将很容易导致栈溢出。为了避免发生这种情况，你可以在`deepCopy`中用迭代（iteration）代替递归（recursion）：

```

// Iteratively copy the linked list headed by this Entry
Entry deepCopy() {
    Entry result = new Entry(key, value, next);
    for (Entry p = result; p.next != null; p = p.next)
        p.next = new Entry(p.next.key, p.next.value, p.next.next);
    return result;
}

```

克隆复杂对象的最后一个办法是，先调用`super.clone`，然后把结果对象中的所有域都设置到它们的空白状态（virgin state），然后调用高层（higher-level）的方法来重新产生对象的状态。在我们的`HashTable`例子中，`buckets`域将被初始化为一个新的散列桶数组，然后，对于正在被克隆的散列表中的每一个键–值对，都调用`put(key, value)`方法（上面没有给出其代码）。这种做法往往会产生一个简单的、合理的、优美的`clone`方法，但是它运行起来没有“直接操作对象和其克隆的内部状态的`clone`方法”快。

如同构造函数一样，`clone`方法不应该在构造过程中，调用新对象中任何非final方法（见第15条）。如果`clone`调用了一个被改写过的方法，那么在该方法所在的子类有机会修正

新对象中的状态之前，该方法就会先被执行，这样很有可能会导致克隆对象和原始对象之间的不一致。因此，上一段落中讨论到的`put(key, value)`方法应该要么是final，要么是私有的（如果是私有的，那么它应该算是非final公有方法的“辅助方法[helper method]”）。

50 Object的`clone`方法被声明为可抛出`CloneNotSupportedException`异常，但是，改写版本的`clone`方法可能会忽略这个声明。`final`类的`clone`方法应该省略这个声明，因为不会抛出被检查异常（checked exception）的方法比会抛出异常的方法，用起来更舒服（见第41条）。如果一个可扩展的类（特别是专门为了继承而设计的类[见第15条]）改写了`clone`方法，那么改写版本的`clone`方法应该包含“抛出`CloneNotSupportedException`异常”这样的声明。这样做可以使子类通过提供下面的`clone`方法，选择温和地放弃克隆（复制）的能力：

```
// Clone method to guarantee that instances cannot be cloned
public final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

遵守前面的建议并不是必需的，因为如果一个子类不希望它的实例被克隆，并且它改写的`clone`方法没有被声明为可抛出`CloneNotSupportedException`异常，那么它的`clone`方法总是可以抛出一个未被检查的异常（unchecked exception），比如`UnsupportedOperationException`。然而，通常的实践表明，在这样的情况下，`CloneNotSupportedException`是正确的异常。

简而言之，所有实现了`Cloneable`接口的类都应该用一个公有的方法改写`clone`。此公有方法首先调用`super.clone`，然后修正任何需要修正的域。通常情况下，这意味着要拷贝任何包含内部“深层结构”的可变对象，并且用指向新对象的引用代替原来指向这些对象的引用。虽然，这些内部拷贝操作往往可以通过递归地调用`clone`来完成，但这通常并不是最佳方法。如果该类只包含原语类型的域，或者指向非可变对象的引用，那么多半的情况是没有域需要修正。这条规则也有例外，譬如，代表序列号或其他惟一ID值的域，或者代表对象的创建时间的域，尽管这些域是原语类型或者是非可变的，但它们也需要被修正。

所有这些复杂性真的是必需的吗？很少是。如果你扩展一个实现了`Cloneable`接口的类，那么你除了实现一个行为良好的`clone`方法外，没有别的选择。否则的话，最好的做法是，提供某些其他的途径来代替对象拷贝，或者干脆不提供这样的能力。例如，对于非可变类、支持对象拷贝并没有太大的意义，因为被拷贝的对象与原始对象并没有实质的不同。

另一个实现对象拷贝的好办法是提供一个拷贝构造函数（copy constructor）。所谓拷贝构造函数，它也是一个构造函数，其惟一的参数的类型是包含该构造函数的类，例如：

```
public Yum(Yum yum);
```

另一种方法是它的一个微小的变形：提供一个静态工厂来代替构造函数：

```
public static Yum newInstance(Yum yum);
```

[51]

拷贝构造函数的做法，以及它的静态工厂方法变形，比Cloneable/clone方法具有更多的优势：它们不依赖于某一种很有风险的、语言之外的对象创建机制；它们不要求遵守尚未良好文档化的规范；它们不会与final域的正常使用发生冲突；它们不会要求客户捕获不必要的被检查异常（checked exception）；它们为客户提供了一个静态类型化的对象。虽然你不可能把一个拷贝构造函数或者静态工厂放到一个接口中，但是由于Cloneable接口缺少一个公有的clone方法，所以它也没有提供一个接口该有的功能。因此，你使用拷贝构造函数来代替clone方法，并没有放弃接口的功能特性。

更进一步，一个拷贝构造函数（或者静态工厂）可以带一个参数，它的类型可以是该类实现的一个适当的接口。例如，按照惯例，所有通用集合实现都提供了一个拷贝构造函数，它的参数类型是Collection或者Map。基于接口的拷贝构造函数允许客户来选择拷贝动作的具体实现，而不是强迫客户接受原始的实现。例如，假设你有一个LinkedList l，并且希望把它拷贝成一个ArrayList。clone方法并没有提供这样的功能，但是用拷贝构造函数很容易实现：new ArrayList(l)。

既然Cloneable具有以上那么多问题，所以，可以安全地说，其他的接口不应该扩展（extend）这个接口，并且，为了继承而设计的类（见第15条）也不应该实现（implement）这个接口。由于它具有这么多缺点，有些专家级的程序员从来不去改写clone方法，也从来不去调用它，除非是为了低开销地拷贝一个数组。你必须清楚一点，对于一个专门为继承而设计的类，如果你未能提供至少一个行为良好的受保护的（protected）clone方法，那么它的子类要想实现Cloneable接口是不可能的。

[52]

第11条：考虑实现Comparable接口

与本章中讨论的其他方法不同，`compareTo`方法在`Object`中并没有被声明。相反，它是`java.lang.Comparable`接口中惟一的方法。`compareTo`方法允许进行简单的相等比较，也允许执行顺序比较，除此之外，它与`Object`的`equals`方法具有相似的特征。一个类实现了`Comparable`接口，就表明它的实例具有内在的排序关系（*natural ordering*）。若一个数组中的对象实现了`Comparable`接口，则对这个数组进行排序非常简单：

```
Arrays.sort(a);
```

对于存储在集合中的`Comparable`对象，搜索、计算极值以及自动维护工作都非常简单，例如，下面的程序依赖于`String`实现了`Comparable`接口，它去掉了命令行参数列表中的重复参数，并按字母表顺序打印出来：

```
public class WordList {
    public static void main(String[] args) {
        Set s = new TreeSet();
        s.addAll(Arrays.asList(args));
        System.out.println(s);
    }
}
```

一旦你的类实现了`Comparable`接口，它就可以跟许多泛型算法（generic algorithm），以及依赖于该接口的集合实现（collection implementation）进行协作。你可以以很小的努力，来获得非常强大的功能。事实上，Java平台库中的所有值类（value classes）都实现了`Comparable`。如果你正在编写一个值类，它具有非常明显的内在排序关系，比如按字母表顺序、按数值顺序或者按年代顺序，那么你几乎总是应该考虑实现这个接口。本条目告诉你应该如何去做。

`compareTo`方法的通用约定与`equals`方法的通用约定具有相似的特征，下面是它的内容，摘自`Comparable`的规范：

将当前这个对象与指定的对象进行顺序比较。当该对象小于、等于或大于指定对象的时候，分别返回一个负整数、零或者正整数。如果由于指定对象的类型而使得无法进行比较，则抛出`ClassCastException`异常。

在下面的说明中，记号`sgn (expression)`表示数学上的`signum`函数，它根据`expression`的值为负值、零和正值，分别返回`-1`、`0`或`1`。

实现者必须保证对于所有的`x`和`y`，满足`sgn (x.compareTo (y)) == -sgn`

($y.compareTo(x)$)。(这也暗示着，当且仅当 $y.compareTo(x)$ 抛出一个异常的时候， $x.compareTo(y)$ 也必须抛出一个异常。)

- 实现者也必须保证这个比较关系是可传递的： $(x.compareTo(y) > 0 \&& y.compareTo(z) > 0)$ 暗示着 $x.compareTo(z) > 0$ 。
- 最后，实现者保证 $x.compareTo(y) == 0$ 暗示着：对于所有的 z ， $\text{sgn}(x.compareTo(z)) == \text{sgn}(y.compareTo(z))$ 。
- 强烈建议 $(x.compareTo(y) == 0) == (x.equals(y))$ ，但这不是严格要求。一般而言，任何实现了Comparable接口的类，若违反了这个条件，应该明确予以说明。推荐使用这样的说法：“注意：该类具有内在的排序功能，但是与equals不一致。”

请不要对上述约定中的数学关系感到厌烦。如同equals约定（见第7条）一样，compareTo约定并没有它看起来那么复杂。在一个类的内部，任何一种合理的顺序关系都可以满足compareTo约定。然而，与equals不同的是，在跨越不同类的时候，compareTo可以不做比较：如果两个被比较的对象引用分别指向不同类的对象，那么compareTo可以抛出ClassCastException异常。通常，这正是compareTo在这种情况下应该做的事情。虽然以上的约定并没有把跨类之间的比较排除之外，但是，在Java 1.4发行版本中，Java平台库中没有哪个类支持这种特性。

就好像一个违反了hashCode约定的类会破坏其他的依赖于散列做法的类一样，一个违反了compareTo约定的类也会破坏其他依赖于比较关系的类。依赖于比较关系的类包括有序集合类TreeSet和TreeMap，以及工具类Collections和Arrays，它们内部包含有搜索和排序算法。

现在我们来回顾一下compareTo约定中的每一条款。第一条指出，如果你反转了两个对象引用之间的比较方向，则只会发生下面的情况：如果第一个对象小于第二个对象，则第二个对象一定大于第一个对象；如果第一个对象等于第二个对象，则第二个对象一定等于第一个对象；如果第一个对象大于第二个对象，则第二个对象一定小于第一个对象。第二条指出，如果一个对象大于第二个对象，并且第二个对象又大于第三个对象，那么第一个对象一定大于第三个对象。最后一条指出，在比较时被认为相等的所有对象，它们跟别的对象做比较时一定会产生同样的结果。

54

这三个条款的一个直接结果是，由compareTo方法施加的相等测试（equality test），也一定遵守与equals约定施加同样的限制条件：自反性、对称性、传递性和非空性。因此，下面的告诫也同样适用：没有一种简单的方法可以做到，在扩展一个新的可实例化的类的时候，既增加了新的特征，同时又保持了compareTo约定（见第7条）。针对equals的解决方案也同样适用于compareTo方法。如果你想为一个实现了Comparable接口的类增加一个关键特征，请不

要扩展这个类；而是编写一个不相关的类，其中包含一个域，其类型是第一个类。然后提供一个“视图（view）”方法返回这个域。这样做，你既可以自由地在第二个类上实现`compareTo`方法，同时也允许客户在必要的时候，把第二个类的实例当做第一个类的实例来看待。

`compareTo`约定的最后一段是一个强烈的建议，而不是真正的规则，只是说明了`compareTo`方法施加的相等测试，在通常情况下应该返回与`equals`方法同样的结果。如果遵守了这条，那么由`compareTo`方法施加的顺序关系被称为“与`equals`一致（consistent with `equals`）”。如果这条规则没有被遵守的话，则顺序关系被称为“与`equals`不一致（inconsistent with `equals`）”。如果一个类的`compareTo`方法施加了一个与`equals`方法不一致的顺序关系，那么，它仍然能正常工作，但是，如果一个有序集合（sorted collection）包含了该类的元素，则这个集合可能无法遵守某些集合接口（Collection、Set或Map）的通用约定。这是因为，对于这些接口的通用约定是按照`equals`方法来定义的，但是有序集合使用了由`compareTo`方法而不是`equals`方法施加的相等测试。尽管出现这种情况不会造成灾难性的后果，但是应该有所了解。

例如，考虑`BigDecimal`类，它的`compareTo`方法与`equals`不一致。如果你创建了一个`HashSet`，并且加入一个`new BigDecimal("1.0")`和一个`new BigDecimal("1.00")`，则这个集合将包含两个元素，因为两个新加入到集合中的`BigDecimal`实例，通过`equals`方法来比较是不相等的。然而，如果你使用`TreeSet`而不是`HashSet`来执行同样的过程，则集合中只包含一个元素，因为这两个`BigDecimal`实例通过`compareTo`方法进行比较是相等的。（细节请参阅`BigDecimal`的文档。）

编写`compareTo`方法与编写`equals`方法非常相似，但也存在几个关键的不同之处。在对实参进行类型转换之前，你并不需要检查实参的类型。如果实参的类型不合适，则`compareTo`方法应该抛出`ClassCastException`异常。如果实参为`null`，则`compareTo`方法应该抛出`NullPointerException`异常。这正好是“当你把实参转换到正确的类型，然后试图访问它的成员时”所期待的行为。

域的比较本身是顺序比较，而不是相等比较。比较对象引用域可以通过递归地调用`compareTo`方法来实现。如果一个域并没有实现`Comparable`接口，或者你需要使用一个非标准的排序关系，那么你可以使用一个显式的`Comparator`。或者编写专门的`Comparator`，或者使用已有的`Comparator`，譬如针对第7条中的`CaseInsensitiveString`类，其`compareTo`方法使用一个已有的`Comparator`：

```
public int compareTo(Object o) {
    CaseInsensitiveString cis = (CaseInsensitiveString)o;
    return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
}
```

比较原语类型的域，你可以使用关系操作符 < 和 >；比较数组域时，你可以把这些指导原则应用到每一个元素上。如果一个类有多个关键域，那么，按什么样的顺序来比较这些域是非常关键的。你必须从最关键的域开始，逐步进行到所有的重要域。如果有一个域的比较产生了非零的结果（零代表相等），则整个比较操作结束，并返回该结果。如果最关键的域是相等的，则进一步比较次最关键的域，以此类推。如果所有的域都是相等的，则对象是相等的，返回零。下面通过第8条中的PhoneNumber类的compareTo方法来说明这项技术：

```
public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    // Compare area codes
    if (areaCode < pn.areaCode)
        return -1;
    if (areaCode > pn.areaCode)
        return 1;

    // Area codes are equal, compare exchanges
    if (exchange < pn.exchange)
        return -1;
    if (exchange > pn.exchange)
        return 1;

    // Area codes and exchanges are equal, compare extensions
    if (extension < pn.extension)
        return -1;
    if (extension > pn.extension)
        return 1;

    return 0; // All fields are equal
}
```

56

虽然这个方法可以工作得很好，但它还有改进的余地。回想一下，compareTo方法的约定并没有指定返回值的大小（magnitude），而只是指定了返回值的符号。你可以利用这一点来简化代码，甚至还可以提高它的运行速度：

```
public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    // Compare area codes
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff != 0)
        return areaCodeDiff;

    // Area codes are equal, compare exchanges
    int exchangeDiff = exchange - pn.exchange;
    if (exchangeDiff != 0)
        return exchangeDiff;

    // Area codes and exchanges are equal, compare extensions
    return extension - pn.extension;
}
```

这项技巧在这里能够工作得很好，但是用起来要非常小心。除非你确信问题中的域不会是

负的，或者更一般的情况，最小和最大的可能域值之差小于或等于`INTEGER.MAX_VALUE`($2^{31}-1$)，否则就不要使用这种方法。这项技巧往往不能正常工作的原因在于，一个有符号的32位整数还没有大到足以表达任意两个32位整数的差。如果*i*是一个很大的正整数(`int`类型)，而*j*是一个绝对值很大的负整数(`int`类型)，那么(*i*-*j*)将会溢出，并返回一个负值。这样就使得`compareTo`方法不能工作。对于某些实参，它会返回无意义的结果，从而违反了`compareTo`约定的第一条和第二条。这不是一个纯粹的理论问题，它已经在实时系统中导致了失败。这些失败可能非常难以调试，因为这样的`compareTo`方法对于大多数的输入值都能正常工作。

第4章

类和接口

类和接口是Java程序设计语言的核心，它们也是Java语言的基本抽象单元。Java语言提供了许多强大的基本元素，供程序员用来设计类和接口。本章包含的一些指导原则，可以帮助你更好地利用这些语言元素，以便你设计出来的类和接口更加有用、健壮和灵活。

第12条：使类和成员的可访问能力最小化

要想区别一个设计良好的模块与一个设计不好的模块，最重要的因素是，这个模块对于外部的其他模块而言，是否隐藏了内部的数据和其他的实现细节。一个设计良好的模块会隐藏所有的实现细节，把它的API与实现清晰地隔离开来。然后，模块之间只通过它们的API进行通信，一个模块不需要知道其他模块的内部工作情况。这个概念被称为信息隐藏(*information hiding*)，或封装(*encapsulation*)，是软件设计的基本原则之一[Parnas72]。

信息隐藏之所以非常重要的许多理由，其中大多数理由都源于这样一个事实：它可以有效地解除一个系统中各模块之间的耦合关系，使得这些模块可以被独立地开发、测试、优化、使用、理解和修改。这样可以加速系统开发的速度，因为这些模块可以被并行地开发。它也减轻了维护的负担，因为程序员很快就可以理解这些模块，并且在调试它们的时候可以不伤害其他的模块。虽然信息隐藏本身无论是对内还是对外，都不会带来更好的性能，但是它使得有效的性能调节成为可能。一旦一个系统已经完成，通过分析就可以知道哪些模块影响了系统的性能(见第37条)，那么这些模块可以被进一步优化，而不会影响到其他模块的正确性。信息隐藏可以提高软件的可重用性，因为单独的模块并不依赖于其他的模块，除了开发这些模块所使用的环境之外，它们在其他的环境中也往往是有用的。最后，信息隐藏也降低了构建大型系统的风险；即使整个系统并不成功，这些独立的模块也有可能是成功的。

59

Java程序设计语言提供了许多设施(facility)来帮助做到信息隐藏。其中一个设施是访问控制(*access control*)机制[JLS, 6.6]，它决定了类、接口和成员的可访问性(*accessibility*)。

一个实体的可访问性是由该实体声明所在的位置，以及该实体声明中所出现的访问修饰符(`private`、`protected`和`public`)共同决定的。正确地使用这些修饰符对于实现信息隐藏是非常关键的。

经验表明，你应该尽可能地使每一个类或成员不被外界访问。换句话说，你应该使用最低可能的、并且与该软件的正确功能相一致的访问级别。

对于顶层的（非嵌套的）类和接口，它们只有两种可能的访问级别：包级私有的(*package-private*)和公有的(*public*)。如果你声明了一个具有`public`修饰符的顶层类或者接口，那么它是公有的；否则，它将是包级私有的。如果一个类或者接口能够被做成包级私有的，那么它就应该被做成包级私有的。通过把一个类或者接口做成包级私有的，它实际上成了这个包的实现的一部分，而不是该包导出的API的一部分；并且，在以后的发行版本中，你可以对它进行修改、替换，或者去除，而无需担心会伤害到现有的客户。如果你把它做成公有的，你就有义务永远支持它，以保持兼容性。

如果一个包级私有的顶层类或者接口只是在某一个类的内部被用到，那么你应该考虑使它成为后者的一个私有嵌套类（或者接口）（见第18条）。这样可以进一步降低它的可访问性。然而，这样做不像“使一个不必要的公有类成为包级私有的类”那样重要，因为一个包级私有的类已经是这个包的实现的一部分，而不是其API的一部分。

对于成员（域、方法、嵌套类和嵌套接口）有四种可能的访问级别，下面按照可访问性递增的顺序列出来：

- **私有的(`private`)**——只有在声明该成员的顶层类内部才可以访问这个成员。
- **包级私有的(*package-private*)**——声明该成员的包内部的任何类都可以访问这个成员。在技术上，它被称为“默认(*default*)访问级别”，如果没有为成员指定访问修饰符的话，那么它就具有这样的访问级别。
- **受保护的(`protected`)**——该成员声明所在类的子类可以访问这个成员（有一些限制[JLS, 6.6.2]），并且，该成员声明所在的包内部的任何类也可以访问这个成员。
- **公有的(`public`)**——任何地方都可以访问该成员。

当你仔细地设计了一个类的公有API之后，接下去应该把所有其他的成员都变成私有的。只有当同一个包内的另一个类真正需要访问一个成员的时候，你才应该去掉`private`修饰符，使该成员变成包级私有的。如果你发现自己经常要做这样的事情，那么你应该重新检查你的系统设计，看看是否另一种分解方案所得到的类具有更好的分离特性，彼此之间耦合度更小。可以说，私有成员和包级私有成员都是一个类的实现中的一部分，并不会影响到其导出

的API。然而，如果这些域所在的类实现了Serializable接口（见第54和55条），那么这些域可能会被“泄漏（leak）”到导出的API中。

对于公有类的成员，当访问级别从包级私有变成保护级别时，会出现可访问性的巨大增加。受保护的成员是一个类的导出的API的一部分，必须永远被支持。更进一步，一个导出的类的每一个受保护的成员代表了该类对于一个实现细节的公开承诺（见第15条）。受保护的成员应该尽量少用。

有一条规则使得你无法降低一个方法的可访问性。如果一个方法改写了超类中的一个方法，那么子类中该方法的访问级别低于超类中的访问级别是不允许的[JLS, 8.4.6.3]。这样可以确保子类的实例可以被用在任何可使用超类的实例的场合。如果你违反了这条规则，那么当你试图编译该子类的时候，编译器会产生一条错误信息。这条规则的一种特殊情形是，如果一个类实现了一个接口，那么接口中所有的方法在这个类中都必须被声明为公有的。这是因为接口中所有的方法都隐含着公有访问级别。

公有类应该尽可能少地包含公有的域（相对于公有的方法）。如果一个域是非final的，或者是一个指向可变对象的final引用，那么一旦你使它成为公有的，就放弃了对存储在这个域中的值进行限制的能力；当这个域被修改的时候，你也失去了采取任何行动的能力。一个简单的后果是，包含公有可变域的类不是线程安全的。即使一个域是final的，并且没有指向任何一个可变对象，那么，一旦你把这个域变成公有的，也就放弃了“切换到一个新的内部数据表示（其中这个域不存在）”的灵活性。

对于“公有类不应该包含公有域”这条规则也有一个例外，通过公有的静态final域来暴露类的常量是允许的。按照惯例，这样的域的名字由大写字母组成，单词之间用下划线隔开（见第38条）。很重要的一点是，这些域要么包含原语类型的值，要么包含指向非可变对象的引用（见第13条）。如果一个final域包含一个指向可变对象的引用，那么它具有非final域的所有缺点。虽然引用本身不能被修改，但是它引用的对象可以被修改——这会导致灾难性的后果。[61]

注意，非零长度的数组总是可变的，所以，具有公有的静态final数组域几乎总是错误的。如果一个类包含这样的一个域，客户将能够修改数组中的内容。这是安全漏洞的一个常见根源：

```
// Potential security hole!
public static final Type[] VALUES = { ... };
```

公有数组应该被替换为一个私有数组，以及一个公有的非可变列表：

```
private static final Type[] PRIVATE_VALUES = { ... };
public static final List VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

另一种办法是，如果你要求编译时（compile-time）的类型安全性，并且愿意损失一点性能的话，你可以把公有的数组域替换为一个公有的方法，它返回私有数组的一份拷贝：

```
private static final Type[] PRIVATE_VALUES = { ... };

public static final Type[] values() {
    return (Type[]) PRIVATE_VALUES.clone();
}
```

总而言之，你应该总是尽可能地降低可访问性。你在仔细地设计了一个最小的公有API之后，应该防止把任何杂散的类、接口和成员变成API的一部分。除了公有静态final域的特殊情形之外，公有类不应该包含公有域。并且确保公有静态final域所引用的对象是不可变的。

第13条：支持非可变性

一个非可变类是一个简单的类，它的实例不能被修改。每个实例中包含的所有信息都必须在该实例被创建的时候就提供出来，并且在对象的整个生存期（lifetime）内固定不变。Java平台库包含许多非可变类，其中有String、原语类型的包装类、BigInteger和BigDecimal。非可变类的存在有许多理由：非可变类比可变类更加易于设计、实现和使用。它们不容易出错，更加安全。

为了使一个类成为非可变类，要遵循下面五条规则：

1. 不要提供任何会修改对象的方法（也称为*mutator*）[⊖]。
2. 保证没有可被子类改写的方法。这可以防止粗心的或者恶意的子类破坏该类的不可变行为。为了禁止改写方法，一般做法是使这个类成为final的，但是后面我们还会讨论到其他的做法。
3. 使所有的域都是final的。通过系统的强制方式，这可以清楚地表明你的意图。而且，如果一个指向新创建的实例的引用在缺乏同步机制的情况下，被从一个线程传递到另一个线程，那么有可能还必须要保证正确的行为，这取决于正在重新设计中的内存模型（*memory model*）的情况[Pugh01a]。
4. 使所有的域都成为私有的。这样可以防止客户直接修改域中的信息。虽然非可变类可以只有公有的final域，只要这些域包含原语类型的值或者指向非可变对象的引用，从技术上讲这是允许的，但是，这样做不值得提倡，因为这使得在以后的版本中不可能再改变内部表示。
5. 保证对于任何可变组件的互斥访问。如果你的类具有指向可变对象的域，则必须确保该类的客户无法获得指向这些对象的引用。并且，永远不要用客户提供的对象引用来初始化这样的域，也不要在任何一个访问方法（accessor）中返回该对象引用。在构造函数、访问方法和readObject方法（见第56条）中请使用保护性拷贝（*defensive copy*）技术（见第24条）。

63

前面条目中的许多例子都是非可变的，其中一个例子是第8条中的PhoneNumber，针对每一个属性它都有一个访问方法（accessor），但是没有对应的改变函数（mutator）。下面是一个稍微复杂一点的例子：

[⊖] 即改变对象属性的成员函数。——译注

```

public final class Complex {
    private final float re;
    private final float im;

    public Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    // Accessors with no corresponding mutators
    public float realPart() { return re; }
    public float imaginaryPart() { return im; }

    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }

    public Complex subtract(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex multiply(Complex c) {
        return new Complex(re*c.re - im*c.im,
                           re*c.im + im*c.re);
    }

    public Complex divide(Complex c) {
        float tmp = c.re*c.re + c.im*c.im;
        return new Complex((re*c.re + im*c.im)/tmp,
                           (im*c.re - re*c.im)/tmp);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Complex))
            return false;
        Complex c = (Complex)o;
        return (Float.floatToIntBits(re) == // See page 28 to
                Float.floatToIntBits(c.re)) && // find out why
               (Float.floatToIntBits(im) == // floatToIntBits
                Float.floatToIntBits(c.im)); // is used.
    }

    public int hashCode() {
        int result = 17 + Float.floatToIntBits(re);
        result = 37*result + Float.floatToIntBits(im);
        return result;
    }

    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}

```

[64] 这个类表示一个复数 (*complex number*, 具有实部和虚部)。除了标准的Object方法之外, 它还提供了针对实部和虚部的访问方法, 以及4个基本的算术运算: 加法、减法、乘法和除法。注意这些算术运算是如何创建并返回一个新的Complex实例, 而不是修改当前这个实例。大多数重要的非可变类都使用了这种模式。它被称为函数的 (*functional*) 做法, 因为这

些方法返回了一个函数的结果，这些函数对操作数进行运算但并不修改它。与之相对应的更常见的是过程的（procedural）做法，在这种方式下，方法内部有一个过程作用在它们的操作数上使得它的状态发生了改变。

如果你对函数方式的做法还不太熟悉的话，你会觉得它显得不太自然，但是它带来的好处是非可变性，具有许多优点。非可变对象比较简单。一个非可变对象可以只有一个状态，即最初被创建时刻的状态。如果你能够确保所有的构造函数都建立了这个类的约束关系，那么可以保证这些约束关系在整个生命周期内永远不再发生变化，无需你和使用这个类的程序员再做额外的工作来维护这些约束关系。相反，可变对象可以有任意复杂的状态空间。如果文档中没有对mutator方法所执行的状态转换提供精确的描述，那么，要可靠地使用一个可变类是非常困难的，甚至是不可能的。

非可变对象本质上是线程安全的，它们不要求同步。当多个线程并发访问这样的对象时，它们不会被破坏。这无疑是获得线程安全最容易的办法。实际上，没有一个线程会注意到其他线程对于一个非可变对象所施加的影响。所以，**非可变对象可以被自由地共享**。非可变类应该充分利用这种优势，鼓励客户尽可能地重用已有的实例。要做到这一点，一个很简便的办法是，对于频繁被用到的值，为它们提供公有的静态final常量。例如，Complex类有可能会提供下面的常量：

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE = new Complex(1, 0);
public static final Complex I = new Complex(0, 1);
```

这种方法可以被进一步扩展。一个非可变类可以提供一些静态工厂，它们把频繁被用到的实例缓存起来，从而当请求一个预先存在的实例的时候，可以不再创建新的实例。BigInteger和Boolean类都有这样的静态工厂。使用这样的静态工厂也使得客户之间可以共享已有的实例，而不用创建新的实例，从而降低内存占用和垃圾回收的代价。

“非可变对象可以被自由地共享”导致的一个结果是，你永远也不需要进行保护性拷贝（见第24条）。实际上，你根本不需要做任何拷贝，因为这些拷贝始终等于原始的对象。因此，你不需要，也不应该为非可变类提供clone方法或者拷贝构造函数（见第10条）。这一点在Java平台早期出现的时候并不好理解，所以String类仍然提供了一个拷贝构造函数，但是它应该很少被使用（见第4条）。

你不仅可以共享非可变对象，甚至也可以共享它们的内部信息。例如，BigInteger类内部使用了符号数值表示法，其中符号部分用一个int类型的值来表示，而数值部分用一个int数组来表示。negate方法产生一个新的BigInteger，其中数值是一样的，而符号是相反的。它并不需要拷贝数组，新建的BigInteger也指向原来实例中的同一内部数组。

非可变对象为其他对象——无论是可变的还是非可变的——提供了大量的构件 (building blocks)。如果一个复杂对象内部的组件对象不会改变的话，那么要维护它的不变性约束是很容易的。这条原则的一种特殊情形是，非可变对象构成了大量的映射键 (map key) 和集合元素 (set element)；一旦非可变对象进入到一个映射表 (map) 或者集合 (set) 中，尽管这破坏了映射表或者集合的不变性约束，但是你不用担心它们的值会发生变化。

非可变类真正惟一的缺点是，对于每一个不同的值都要求一个单独的对象。创建这样的对象可能会代价很高，特别是对于大型对象的情形。例如，假设你有一个上百万位的 BigInteger，希望对它的低位求反：

66 `BigInteger moby = ...;`
`moby = moby.flipBit(0);`

flipBit方法创建一个新的BigInteger实例，也有上百万位长，它与原来的对象只有一位不同。这个操作所消耗的时间和空间与BigInteger的尺寸成正比。我们拿它与java.util.BitSet进行比较。与BigInteger类似，BitSet代表一个任意长度的位序列，但是，与BigInteger不同的是，BitSet是可变的。BitSet类提供了一个方法允许你在常数时间 (constant time) 内改变此“百万位”实例中单个位的状态。

如果你执行一个多步骤的操作，并且每个步骤都会产生一个新的对象，除了最后的结果之外其他的对象最终都会被丢弃，那么此时性能问题就会显露出来。处理这种问题有两种办法：第一种办法，先猜测一下哪些多步骤操作会经常被用到，然后把它们作为基本单元来提供。如果一个多步骤操作已经成为一个基本单元了，那么非可变类就可以不必在每个步骤上创建一个独立的对象了。非可变类可以在内部做得非常灵活。例如，BigInteger有一个包级私有的可变“配套类 (companion class)”，它的用途是加速诸如“模乘幂 (modular exponentiation)”这样的多步骤操作。由于前面提到的诸多原因，使用可变配套类是非常困难的，但是幸运的是，你并不需要这样做。BigInteger的实现者已经为你完成了所有的困难工作。

如果你能够精确地预测出哪些复杂的多阶段操作将会作用在你的非可变类上，那么这种方法可以工作得很好。如果不能预测的话，那么最好的办法是提供一个公有的可变配套类。在Java平台库中，这种方法的一个主要例子是String类，它的可变配套类是StringBuffer。可以这样认为，在特定的环境下，相对于BigInteger而言，BitSet同样扮演了可变配套类的角色。

现在你已经知道了如何构建一个非可变类，并且理解了非可变性的优点和缺点。现在我们来讨论其他一些设计方案。前面提到过，为了保证非可变性，一个类绝对不允许它的方法被子类改写。除了“使一个类成为final的”这种方法之外，还可以有其他两种办法做到这一点。

一种办法是，让这个类的每一个方法都成为final的，而不是让整个类成为final的。这种方法的惟一好处在于，它使得程序员可以扩展这个类，在原来的基础上增加新的方法。这种做法其效果与“在一个独立的、不可被实例化的工具类中增加新的静态方法”（见第3条）相同，所以这种方法并不提倡。

“使一个非可变类变为final”的第二种替代做法是，使其所有的构造函数成为私有的，或者包级私有的，并且增加公有的静态工厂，来代替公有的构造函数（见第1条）。为了具体说明这种做法，下面以Complex为例，看看如何使用这种方法：

```
// Immutable class with static factories instead of constructors
public class Complex {
    private final float re;
    private final float im;

    private Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(float re, float im) {
        return new Complex(re, im);
    }

    ... // Remainder unchanged
}
```

虽然这种方法并不常用，但是它往往是三种可选择的做法中最好的一种。它最灵活，因为它允许使用多个包级私有的实现类。对于包外面的客户而言，非可变类实际上是final的，因为要扩展这样一个来自其他包的类是不可能的，它缺少一个公有的或者受保护的构造函数。除了允许多个实现类的灵活性之外，这种做法也使得有可能在后续的发行版本中改进这个类的性能，具体做法是增强静态工厂的对象缓存能力。

静态工厂相比构造函数具有许多其他的优势，正如在第1条所讨论的。例如，假设你希望Complex类提供一种“基于极坐标创建复数对象”的方式。如果使用构造函数来实现这样的功能，可能会使得这个类很零乱，因为这样的构造函数与已有的构造函数Complex(float, float)具有同样的原型特征。但是通过静态工厂，这很容易做到：只需用第二个静态工厂，并且工厂的名字清楚地表明它的功能即可：

```
public static Complex valueOfPolar(float r, float theta) {
    return new Complex((float) (r * Math.cos(theta)),
                      (float) (r * Math.sin(theta)));
}
```

当BigInteger和BigDecimal最初被编写出来的时候，“为什么非可变类必须要有final的等同效果”还没有被普遍理解，所以它们的所有方法都有可能被改写。不幸的是，

[67]

[68]

为了保持向上兼容，这个问题一直无法得以改正。如果你当前正在编写一个类，它的安全性依赖于 BigInteger 或者 BigDecimal 实参（来自不可信的客户）的非可变性，那么，你必须检查，以确定这个实参是不是一个“真正的”的 BigInteger 或者 BigDecimal，而不是一个不可信子类的实例。如果是后者的话，你必须在“假设它可能是可变的”前提下对它进行保护性拷贝（见第24条）：

```
public void foo(BigInteger b) {
    if (b.getClass() != BigInteger.class)
        b = new BigInteger(b.toByteArray());
    ...
}
```

在本条目刚开始时给出的关于非可变类的规则指出，没有方法会修改对象，并且所有的域必须是 final 的。实际上，这些规则比真正的要求强了一点，为了提高性能可以有所放松。事实上应该是这样：没有一个方法能够对对象的状态产生外部可见（*externally visible*）的改变，然而，许多非可变的类拥有一个或者多个非 final 的冗余域，它们把一些开销昂贵的计算结果缓存在这些域中（当第一次请求执行这些计算的时候）。如果将来再次请求这些计算，则直接返回这些被缓存的值，从而节约了重新计算所需要的开销。这种技巧可以很好地工作，因为对象是非可变的，它的非可变性保证了这些计算如果被再次执行的话，会产生同样的结果。

例如，PhoneNumber 类的 hashCode 方法（见第8条）在第一次被调用的时候，计算出散列码，然后把它缓存起来，以备将来被再次使用。这项技术是迟缓初始化（*lazy initialization*）（见第48条）的一个典型例子，String 类也用到了这项技术。它不需要同步机制，因为即使散列值被重新计算一次或者两次，都不会成问题。下面的代码是一种习惯用法，它使用了迟缓初始化技术，并且把非可变对象的函数结果缓存起来：

```
// Cached, lazily initialized function of an immutable object
private volatile Foo cachedFooVal = UNLIKELY_FOO_VALUE;

public Foo foo() {
    Foo result = cachedFooVal;
    if (result == UNLIKELY_FOO_VALUE)
        result = cachedFooVal = fooVal();
    return result;
}

// Private helper function to calculate our foo value
private Foo fooVal() { ... }
```

69

一个有关序列化功能的告诫有必要在这里提出来。如果你选择让自己的非可变类实现 Serializable 接口，并且它包含一个或者多个指向可变对象的域，那么你必须提供一个显式的 readObject 或者 readResolve 方法，即使默认的序列化形式是可以接受的，也是如此。默认的 readObject 方法使得一个攻击者可以从非可变类创建可变的实例。这个话题的细节请参考第56条。

总而言之，坚决不要为每个get方法编写一个相应的set方法。除非有很好的理由要让一个类成为可变类，否则就应该是非可变的。非可变类有许多优点，惟一的缺点是在特定的情况下存在潜在的性能问题。你应该总是使一些小的值对象，比如PhoneNumber和Complex，成为非可变的（在Java平台库中，有几个类如java.util.Date和java.awt.Point，它们本应该是非可变的，但实际上却不是）。你也应该认真考虑把一些大的值对象做成非可变的，例如String和BigInteger。只有当你确实认为达到满意的性能非常有必要时（见第37条），你才应该为非可变类提供公有的可变配套类。

对于有些类而言，非可变性是不切实际的，这样的类包括“过程类（process class）”，例如Thread和TimerTask。如果一个类不能被做成非可变类，那么你仍然应该尽可能地限制它的可变性。降低一个对象中存在的状态的数目，可以更容易地分析该对象的行为，同时降低出错的可能性。因此，构造函数应该创建完全初始化的对象，所有的约束关系应该在这时候建立起来，构造函数不应该把“只构造了一部分的实例”传递给其他的方法。你不应该在构造函数之外再提供一个公有的初始化方法，除非有绝对很好的理由要这么做。类似地，你也不应该提供一个“重新初始化”方法（它使得一个对象可以被重用，就好像这个对象是由另一不同的初始状态构造出来的一样）。与所增加的复杂性相比，“重新初始化”方法通常并没有带来太多的性能优势。

可以通过TimerTask类来说明这些原则。它是可变的，但是它的状态空间被有意地设计得非常小。你可以创建一个实例，对它进行调度使它执行起来，也可以随意地取消它。一旦一个定时器任务（timer task）已经完成，或者已经被取消，你就不可能再对它重新调度。

最后值得注意的一点与本条目中的Complex类有关。这个例子只是被用来演示非可变性的，它不是一个工业强度（即产品级）的复数实现。它对复数乘法和除法使用了标准的计算公式，会导致不正确的舍入；对于复数NaN和无穷大没有提供很好的语义[Kahan91, Smith62, Thomas94]。

第14条：复合优先于继承

继承 (inheritance) 是实现代码重用的有力手段，但它并不总是完成这项工作的最佳工具。不适当使用继承会导致脆弱的软件，在一个包的内部使用继承是非常安全的，在那儿子类和超类的实现在同一个程序员的控制之下。对于专门为了继承而设计、并且具有很好文档说明的类（见第15条），使用继承也是非常安全的。然而，对普通的具体类（concrete class）进行跨越包边界的继承，则是非常危险的。再次提醒，本书使用单词“继承”，其含义为实现继承（*implementation inheritance*，当一个类扩展另一个类的时候）。本条目讨论的问题并不适用于接口继承（*interface inheritance*，当一个类实现一个接口的时候，或者一个接口扩展另一个接口的时候）。

与方法调用不同的是，继承打破了封装性[Snyder86]。换句话说，一个子类依赖于其超类中特定功能的实现细节。超类的实现有可能会随着发行版本的不同而有所变化，如果真的发生了变化，则子类可能会被打破，即使它的代码完全没有改变。因而，一个子类必须要跟着其超类的更新而发展，除非超类是专门为了扩展而被设计的，并且具有很好的文档说明。

为了说得更加具体一点，我们假设有一个程序使用了 HashSet。为了调节该程序的性能，我们需要查询 HashSet，看一看自从它被创建以来曾经有多少个元素被加了进来（不要与它当前的大小混淆起来，大小值会随着元素的删除而递减）。为了提供这样的功能，我们编写一个 HashSet 变种类，它记录下试图插入元素的数量，并针对该计数值导出一个访问方法。HashSet 类中有两个方法可以增加元素：add 和 addAll，所以，我们要改写这两个方法：

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(Collection c) {
        super(c);
    }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
    }
}
```

```

        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}

```

这个类看起来非常合理，但是它并不能正常工作。假设我们创建了一个实例，并使用`addAll`方法添加了三个元素：

```

InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[] {"Snap", "Crackle", "Pop"}));

```

这时候，我们期望`getAddCount`方法将会返回3，但是它实际上返回6。哪里错了呢？在`HashSet`的内部，`addAll`方法是基于它的`add`方法来实现的，但是，`HashSet`的文档中并没有说明这样的实现细节，这也是很合理的。`InstrumentedHashSet`中的`addAll`方法首先给`addCount`增加3，然后通过`super.addAll`来调用`HashSet`的`addAll`实现。然后又顺次调用到被`InstrumentedHashSet`改写了的`add`方法，每个元素调用一次。这三次调用又分别给`addCount`多加了1，所以，总共增加了6：通过`addAll`方法增加的每个元素都被计算了两次。

我们只要去掉被改写的`addAll`方法，就可以“修正”这个子类。虽然这样得到的类可以正常工作，但是，它的功能正确性需要依赖于这样的事实：`HashSet`的`addAll`方法是在`add`方法上实现的。这种“自用性（self-use）”是实现细节，不是承诺，不能保证在所有的Java平台上都保持不变、不会随着版本的不同而不发生变化。因此，这样得到的`InstrumentedHashSet`类将是非常脆弱的。

一种稍好一点的做法是，改写`addAll`方法来对指定集合中的每个元素进行迭代，为每个元素依次调用`add`方法一次。这样做可以保证得到正确的结果，不管`HashSet`的`addAll`方法是否是在`add`方法的基础上实现的，因为`HashSet`的`addAll`实现没有再被调用到。然而，这项技术并没有解决我们所有的问题，它相当于重新实现了超类的方法，超类的方法可能是自用的（self-use），也可能不是自用的，这种方法很困难，也非常耗时，并且容易出错。而且，这样做并不总是可行的，因为对于有些方法，它们无法访问对于子类来说是私有的域，这样的方法在子类中是无法重新实现的。

72

造成子类脆弱性的一个相关的原因是，它们的超类在后续版本中可以获得新的方法。假设一个程序的安全性依赖于这样的事实：所有被插入到某个集合中的元素都满足某个先决条件。下面的做法就可以保证这一点：对集合进行子类化，对所有能够添加元素的方法都进行改写，以便确保在加入每个元素之前它满足这个先决条件。如果在后续的版本中，超类中没有增加新的“能添加元素的方法”，那么这种做法可以正常工作。然而，一旦超类增加了这样的新方

法，则很可能由于调用了这个未被子类改写的新方法，因此“非法的”元素会被添加到子类的实例中。这不是一个纯粹的理论问题，在把Hashtable和Vector加入到Collections Framework中的时候，就修正了几个这类实质的安全漏洞。

上面的问题都来源于对方法的改写（overriding）动作。如果你在扩展一个类的时候，仅仅是增加新的方法，而不改写已有的方法，你可能会认为这样做是安全的。虽然这种扩展方式会更加安全一些，但是也并不是完全没有风险。如果超类在后续的版本中获得了一个新的方法，并且不幸的是，你在子类中的一个方法也有同样的原型特征，只是返回类型不同，那么这样的子类将无法通过编译[JLS, 8.4.6.3]。如果子类中的方法与超类中的新方法具有完全一致的原型特征，那么你实际上改写了超类中的方法，所以，你又回到上面讲述的两个问题上去了。而且，子类中的方法是否能够遵守超类中新方法的约定，这是很值得怀疑的，因为当你在编写子类的方法的时候，超类中新方法的约定还根本没有面世呢。

幸运的是，有一种办法可以避免前面提到的所有问题：你不再是扩展一个已有的类，而是在新的类中增加一个私有域，它引用了这个已有的类的一个实例。这种设计被称做“复合（composition）”，因为原来已有的类变成了新类的一个组成部分。新类中的每个实例方法都可以调用被包含的已有类实例中对应的方法，并返回它的结果。这被称为转发（forwarding），

[73] 新类中的方法被称为转发方法（forwarding method）。这样得到的类将会非常稳固，它不依赖于已有类的实现细节。即使已有的类增加了新的方法，也不会影响新的类。为了使这种设计方法更加直观，请看下面的InstrumentedSet类，它使用了复合/转发的手段来代替InstrumentedHashSet类：

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {
        this.s = s;
    }

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }

    // Forwarding methods
}
```

```

public void clear()           { s.clear(); }
public boolean contains(Object o) { return s.contains(o); }
public boolean isEmpty()        { return s.isEmpty(); }
public int size()              { return s.size(); }
public Iterator iterator()     { return s.iterator(); }
public boolean remove(Object o) { return s.remove(o); }
public boolean containsAll(Collection c)
    { return s.containsAll(c); }
public boolean removeAll(Collection c)
    { return s.removeAll(c); }
public boolean retainAll(Collection c)
    { return s.retainAll(c); }
public Object[] toArray()       { return s.toArray(); }
public Object[] toArray(Object[] a) { return s.toArray(a); }
public boolean equals(Object o) { return s.equals(o); }
public int hashCode()          { return s.hashCode(); }
public String toString()        { return s.toString(); }
}

```

74

InstrumentedSet类的设计是通过Set接口的存在来达到目的的，因为Set接口抓住了HashSet类的功能特性。除了获得了健壮性之外，这种设计也带来了格外的灵活性。InstrumentedSet类实现了Set接口，并且拥有惟一的一个构造函数，它的参数也是一个Set类型。从本质上讲，这个类把一个Set转变成另外一个Set，同时还增加了计数的功能。前面提到的基于继承的方法只能为一个具体的类而工作，并且对于超类所支持的每一个构造函数都要求有一个单独的构造函数，与此不同的是，这里的包装类（wrapper class）可以被用来包装任何一个Set实现，并且可以与任何先前已有的构造函数一起工作。例如：

```

Set s1 = new InstrumentedSet(new TreeSet(list));
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));

```

InstrumentedSet甚至也可以用来临时替换一个原本没有计数特性的Set实例：

```

static void f(Set s) {
    InstrumentedSet sInst = new InstrumentedSet(s);
    ... // Within this method use sInst instead of s
}

```

因为每一个InstrumentedSet实例都把另一个Set实例包装起来了，所以InstrumentedSet类被称做包装类（wrapper class）。这也正是Decorator模式[Gamma95, p.175]，因为InstrumentedSet类对一个集合进行了修饰，为它增加了计数特性。有时候，复合和转发这两项技术的结合被错误地引用为“委托（delegation）”。从技术的角度而言，这不是委托，除非包装对象把自己传递给一个被包装的对象[Gamma95, p. 20]。

包装类几乎没有什么缺点。需要注意的一点是，包装类不适合用在回调框架（callback framework）中；在回调框架中，对象把自己的引用传递给其他的对象，以便将来调用回来（“回调”）。因为被包装起来的对象并不知道它外面的包装对象，所以它传递一个指向自己的引用（this），回调时绕开了外面的包装对象。这被称为SELF问题[Lieberman86]。有些人会

75

担心转发方法调用所带来的性能影响，或者包装对象导致的内存占用。在实践中，这两者都不会有很大的影响，编写转发方法倒是有点琐碎，但是因为只需要编写一个构造函数，所以这也算是—部分补偿。

只有当子类真正是超类的“子类型（*subtype*）”的时候，继承才是合适的。换句话说，对于两个类A和B，只有当两者之间确实存在“is-a”关系的时候，类B才应该扩展类A。如果你打算让类B扩展类A，那么你应该问自己：“每一个B确实也是A吗？”如果你不能够确定这个问题的答案是yes，那么B就不应该扩展A。如果答案是no，那么通常情况下，B应该包含A的一个私有实例，并且暴露一个小的、简单的API：A本质上不是B的一部分，只是它的实现细节而已。

在Java平台库中，有许多明显违反这条原则的地方。例如，栈（stack）并不是一个向量（vector），所以stack并不应该扩展vector。类似地，属性列表也不是一个散列表，所以Properties不应该扩展Hashtable。在这两种情况下，复合模式才是恰当的。

如果你在本该使用复合的场合使用了继承，则会不必要地暴露实现细节。这样得到的API会把你限制在原始的实现上，永远限定了你的类的性能。更为严重的是，由于暴露了内部的细节，客户就有可能直接访问这些内部细节。至少而言，这样会导致混淆语义。例如，如果p指向一个Properties实例，那么p.getProperty(key)有可能会与p.get(key)产生不同的结果：前者考虑了默认的属性表，而后者是继承自Hashtable的，它没有考虑默认属性表。最严重的是，客户有可能会直接修改超类，从而打破子类的约束条件。在Properties的情形中，设计者的目是，只允许字符串才可以作为键（key）和值（value），但是直接访问底层的Hashtable就可以违反这种约束条件。一旦违反了约束条件，那么要使用Properties API的其他部分（load和store）就不再可能了。等到发现这个问题时，要改正它已经太晚了，因为客户依赖于使用非字符串的键和值了。

在决定使用继承而不是复合之前，还有最后一组问题应该问自己。对于你正在试图扩展的类，其API中有没有缺陷呢？如果有的话，你是否愿意把这些缺陷传播到自己类的API中？继承机制会把超类API中的所有缺陷传播到子类中，而复合技术则允许你设计一个新的API，从而隐藏这些缺陷。

76

77

总而言之，继承机制的功能非常强大，但是它存在诸多问题，因为它违背了封装原则。只有当子类和超类之间确实存在子类型关系时，使用继承才是恰当的。即便如此，如果子类和超类在不同的包中，并且超类并不是为了扩展而设计的，那么继承将会导致脆弱性（fragility）。为了避免这种脆弱性，可以用复合和转发机制来代替继承，尤其是当存在一个适当的接口来实现一个包装类的时候。包装类不仅比子类更加健壮，而且功能也更加强大。

第15条：要么专门为继承而设计，并给出文档说明，要么禁止继承

第14条警告我们，对一个不是为了继承而设计、并且没有文档说明的“外来”类进行子类化是多么危险。那么对于一个专门为继承而设计并且具有良好文档说明的类而言，这意味着什么呢？

首先，该类的文档必须精确地描述了改写每一个方法所带来的影响。换句话说，该类必须有文档说明其可改写（*overridable*）的方法的自用性（*self-use*）；对于每一个公有的或受保护的方法或者构造函数，它的文档必须指明它调用了哪些可改写的方法，是以什么顺序调用的，每个调用的结果又是如何影响后续的处理过程的（所谓可改写（*overridable*）的方法，是指非final的，公有的或受保护的）。更一般地，一个类必须在文档中说明，在哪些情况下它会调用一个可改写的方法。例如，后台的线程或者静态的初始化器（*initializer*）可能会调用这样的方法。

按照习惯，如果一个方法调用到了可改写的方法，那么在它的文档注释的末尾应该包含关于这些调用的描述信息。这段描述信息要以这样的句子开头：“This implementation.（该方法实现……）”。这样的句子不应该被认为是在表明该行为可能会随着版本的变迁而改变。它意味着这段描述关注该方法的内部工作情况。下面是一个例子，摘自java.util.AbstractCollection的规范：

```
public boolean remove(Object o)
```

Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element *e* such that (*o==null ? e==null : o.equals(e)*), if the collection contains one or more such elements. Returns true if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).

This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method. Note that this implementation throws an UnsupportedOperationException if the iterator returned by this collection's iterator method does not implement the remove method.

该文档清楚地说明了，改写iterator方法将会影响remove方法的行为。而且，它确切地描述了iterator方法返回的Iterator的行为将会怎样影响remove方法的行为。与此相反的是，在第14条的情形中，程序员在子类化HashSet的时候，并没有说明改写add方法是否会影响addAll方法的行为。

关于程序文档有一句格言：好的API文档应该描述一个方法做了什么工作，而并非描述它

是如何做到的。那么，上面这种做法是否违反了这句格言呢？是的，它确实违反了。这正是继承破坏了封装性带来的不幸后果。所以，为了设计一个类的文档，以便它能够被安全地子类化，你应该描述清楚那些可能会导致未定义行为的实现细节。

为了继承而进行的设计不仅涉及到自用的模式的文档设计。为了使程序员能够编写出更加有效的子类，而无需承受不必要的痛苦，一个类必须通过某种形式提供适当的钩子（hook），以便能够进入到它的内部工作流程中，这样的形式可以是精心选择的受保护（protected）方法，也可以是保护域，后者比较少见。例如，考虑`java.util.AbstractList`中的`removeRange`方法：

`protected void removeRange(int fromIndex, int toIndex)`

Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the `ArrayList` by `(toIndex - fromIndex)` elements. (If `toIndex==fromIndex`, this operation has no effect.)

This method is called by the `clear` operation on this list and its sublists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the `clear` operation on this list and its subLists.

This implementation gets a list iterator positioned before `fromIndex` and repeatedly calls `ListIterator.next` followed by `ListIterator.remove`, until the entire range has been removed. Note: If `ListIterator.remove` requires linear time, this implementation requires quadratic time.

Parameters:

`fromIndex` index of first element to be removed.

`toIndex` index after last element to be removed.

这个方法对于List实现的最终用户并没有意义。提供该方法的惟一目的在于，使“子类提供针对子表（sublist）的快速clear方法”更加容易。如果没有这样的`removeRange`方法，那么在子表（sublist）上调用`clear`方法，子类将不得不以平方级性能来完成它的工作。否则的话，就得从头重写整个`subList`机制——这可不是一件容易的事情。
79

那么，当你为了继承而设计一个类的时候，你如何决定应该暴露哪些受保护方法或者域呢？非常不幸，没有神奇的法则可供你使用。你能够做到的最佳途径是努力思考、发挥最好的想像，然后编写一些子类进行测试。你应该尽可能少量地提供受保护方法和域，因为每一个这样的方法或者域都代表了一项关于实现细节的承诺。另一方面，你也不能提供得太少，因为漏掉一个受保护方法可能会导致这个类无法被真正用于继承。

当你为了继承的目的而设计一个有可能会被广泛使用的类时，必须要意识到，对于文档中

所说明的自用模式 (self-use pattern)，以及对于其受保护方法和域所隐含的实现细节，你实际上已经做出了永久的承诺。这些承诺使得你在后续的版本中要提高这个类的性能，或者增加新的功能非常困难，甚至不可能。

而且，由于继承而需要的特殊文档内容也会打乱正常的文档信息，普通的文档被设计用来让程序员可以创建该类的实例，并调用类中的方法。在本书写作的时候，几乎还没有适当的工具或者注释规范，能够把“普通的API文档内容”与“专门针对实现子类的程序员的信息”分离开来。

为了允许继承，一个类还必须遵守其他一些约束。构造函数一定不能调用可被改写的方法，无论是直接进行还是间接进行。如果违反了这条规则，很有可能会导致程序失败。超类的构造函数在子类的构造函数之前运行，所以，子类中改写版本的方法将会在子类的构造函数运行之前就先被调用。如果该改写版本的方法依赖于子类构造函数所执行的初始化工作，那么该方法将不会如预期般地执行。为了更加直观地说明这一点，下面这个很小的类违反了这条规则：

```
public class Super {
    // Broken - constructor invokes overridable method
    public Super() {
        m();
    }

    public void m() {
    }
}
```

80

下面的子类改写了方法m，Super唯一的构造函数就错误地调用了这个方法m：

```
final class Sub extends Super {
    private final Date date; // Blank final, set by constructor

    Sub() {
        date = new Date();
    }

    // Overrides Super.m, invoked by the constructor Super()
    public void m() {
        System.out.println(date);
    }

    public static void main(String[] args) {
        Sub s = new Sub();
        s.m();
    }
}
```

你可能会期望这个程序会打印出日期两次，但是它第一次打印出null，因为方法m被构造函数Super()调用的时候，构造函数Sub()还没有机会初始化date域。注意，这个程序

观察到的final域处于两种不同的状态。

在为了继承而设计类的时候，Cloneable和Serializable接口表现出了特殊的困难。如果一个类是为了继承而被设计的，那么无论实现其中哪个接口都不是一个好主意，因为它们把一些实质性的负担转嫁到了扩展这个类的程序员的身上。然而，你还是可以有一些特殊的手段可以实施，使得子类实现这些接口而无需强迫子类的程序员处理这些负担。第10条和54条讲述了这些特殊的手段。

如果你决定在一个为了继承而设计的类中实现Cloneable或者Serializable接口，你应该意识到，因为clone和readObject方法在行为上非常类似于构造函数，所以一个类似的限制规则也是适用的：无论是clone还是readObject，都不能调用一个可改写的方法，不管是直接的方式，还是间接的方式。对于readObject方法，子类中改写版本的方法将在子类的状态被反序列化（deserialized）之前先被运行；而对于clone方法，改写版本的方法将在子类的clone方法有机会修正被克隆对象的状态之前先被运行。无论哪种情形，都不可避免地将导致程序失败。在clone方法的情形中，这种失败可能会同时损害到原始的对象以及被克隆的对象本身。
[81]

最后，如果你决定在一个为了继承而设计的类中实现Serializable，并且该类有一个readResolve或者writeReplace方法，那么，你必须使readResolve或者writeReplace成为受保护的方法，而不是私有的方法。如果这些方法是私有的，那么子类将会不声不响地忽略掉这两个方法。这正是“为了允许继承，而把实现细节变成一个类的API的一部分”的情形。

到现在为止，应该很明显：为了继承而设计一个类，要求对这个类有一些实质性的限制。这并不是很轻松就可以承诺的决定。在某些情形下，这样的决定很明显是正确的，比如抽象类，包括接口的骨架实现（skeletal implementation）（见第16条）。但是，在另外一些情形下，这样的决定却很明显是错误的，比如非可变类（见第13条）。

但是，对于普通的具体类应该怎么办呢？从传统意义上讲，它们既不是final的，也不是为了子类化而被设计和编写文档的，所以，这种情况下实现继承是很危险的。每次对这个类做了一些修改，那么，从这个类扩展得到的客户类就有可能被打破。这不仅是一个理论问题。对于一个并非为了继承而设计的非final具体类，在修改了它的内部实现之后，与子类化相关的错误报告往往并不少见。

对于这个问题的最好解决方案是，对于那些并非为了安全地进行子类化而设计和编写文档的类，禁止子类化。有两种办法可以禁止子类化。比较容易的办法是把这个类声明为final的。另一种办法是把所有的构造函数变成私有的，或者包级私有的，并且增加一些公有的静

态工厂来替代构造函数的位置。后一种办法在第13条中讨论过，它为内部使用子类提供了灵活性，这两种办法都是可以接受的。

上面的建议可能会引来争议，因为许多程序员已经习惯于对于普通的具体类进行子类化，以便增加新的功能设施，比如仪表功能（instrumentation，如计数显示等）、通知机制或者同步功能，也有可能为了限制原有类中的功能。如果一个类实现了某些能够反映其本质的接口，比如Set、List或者Map，那么，你就不应该为了禁止子类化而感到后悔。第14条中介绍的包装类模式提供了另一种更好的办法，来替代继承机制实现功能的定制。

如果一个具体类并没有实现标准的接口，那么禁止继承可能会给有些程序员带来不便。如果你认为必须允许从这样的类继承，那么一个合理的办法是确保这个类永远也不会调用它的可改写的方法，并且在文档中说明这一点。换句话说，完全消除这个类中可改写的方法的自用特性。这样做之后，你就可以创建“能够安全地进行子类化”的类。改写一个方法将永远也不会影响其他任何方法的行为。82

你可以机械地消除一个类中可改写的方法的自用特性，而不改变它的行为。做法如下：把每个可改写的方法的代码体移到一个私有的“辅助方法（helper method）”中，并且让每个可改写的方法调用它的私有辅助方法。然后，用“直接调用可改写方法的私有辅助方法”来代替“可改写方法的每个自用调用”。83

第16条：接口优于抽象类

Java程序设计语言提供了两种机制，可以用来定义一个允许多个实现的类型：接口和抽象类。两种机制之间最明显的区别是，抽象类允许包含某些方法的实现，但是接口是不允许的。一个更为重要的不同之处在于，为了实现一个由抽象类定义的类型，它必须成为抽象类的一个子类。任何一个类，只要它定义了所有要求的方法，并且遵守通用约定，那么它就允许实现一个接口，不管这个类位于类层次（class hierarchy）的哪个地方。因为Java只允许单继承，所以，抽象类作为类型定义受到了极大的限制。

已有的类可以很容易被更新，以实现新的接口。你所需要做的是：增加要求的方法，如果这些方法原先还不存在的话；然后在类的声明上增加一个`implements`子句。例如，当`Comparable`接口被引入到Java平台中时，许多已有的类被更新，以实现`Comparable`接口。一般地，要更新一个已有的类，使它扩展一个新的抽象类，这往往是不可能的。如果你希望让两个类扩展同一个抽象类，那么你必须把抽象类放到类型层次（type hierarchy）的上部，以便这两个类的一个祖先成为它的子类。不幸的是，这样做会间接地伤害到类层次，强迫这个公共祖先的所有后代类都扩展这个新的抽象类，而不管它对于这些后代类是否合适。

接口是定义mixin（混合类型）的理想选择。一个mixin是指这样的类型：一个类除了实现它的“基本类型（primary type）”之外，还可以实现这个mixin类型，以表明它提供了某些可供选择的行为。例如，`Comparable`是一个mixin接口，它允许一个类表明它的实例可以与其他的可相互比较的对象进行排序操作。这样的接口之所以被称为mixin，是因为它允许可选的功能可被混合到一个类型的基本功能中。抽象类不能被用于定义mixin，同样的理由是因为它们不能被更新到已有的类中：一个类不可能有一个以上的父类，并且在类层次结构中没有适当的地方来放置mixin。

接口使得我们可以构造出非层次结构的类型框架。类型层次对于组织某些事物是非常合适的，但是其他有些事物并不能被整齐地组织成一个严格的层次结构。例如，假设我们有一个接口代表一个singer（歌唱家），另一个接口代表一个songwriter（作曲家）：

```
public interface Singer {
    AudioClip Sing(Song s);
}

public interface Songwriter {
    Song compose(boolean hit);
}
```

在现实生活中，有些歌唱家本身也是作曲家。因为我们使用接口而不是抽象类来定义这些类型，所以对于一个类而言，它同时实现`Singer`和`Songwriter`是完全允许的。实际上，

我们可以定义第三个接口，它同时扩展了Singer和Songwriter，并且加入了一些适合于这种组合的新方法：

```
public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}
```

你并不总是需要这种灵活性，但是一旦你这样做了，则接口就成了一个救世主，帮助你解决了大问题。另外一种做法是编写一个膨胀（bloated）的类层次，对于每一种要被支持的属性组合，都包含一个单独的类。如果在整个类型系统中有n个属性，则总共有 2^n 种可能的组合有待于你提供支持。这种现象被称为“组合爆炸（combinatorial explosion）”。膨胀的类层次可以导致膨胀的类，这些类包含许多方法，并且这些方法只是在参数的类型上有所不同而已，因为类层次中没有一种类型抓住了公共的行为特征。

接口使得安全地增强一个类的功能成为可能，做法是通过第14条介绍的包装类（wrapper class）模式。如果你用抽象类来定义类型，那么这样就使得程序员除了使用继承的手段来增加功能之外，没有别的选择途径。这样得到的类与包装类相比，功能更差，也更加脆弱。

虽然接口不允许包含方法的实现，但是，使用接口来定义类型并不妨碍你为程序员提供实现上的帮助。你可以把接口和抽象类的优点结合起来，对于你期望导出的每一个重要接口，都提供一个抽象的骨架实现（skeletal implementation）类。接口的作用仍然是定义类型，但是骨架实现类负责所有与接口实现相关的工作。

按照惯例，骨架实现被称为AbstractInterface，这里Interface是所实现的接口的名字。例如，Collections Framework为每一个重要的集合接口都提供了一个骨架实现，包括AbstractCollection、AbstractSet、AbstractList和AbstractMap。

85

如果设计得合理的话，利用骨架实现，程序员可以很容易地提供他们自己的接口实现。例如，下面是一个静态工厂方法，它包含一个完整的、功能全面的List实现：

```
// List adapter for int array
static List intArrayList(final int[] a) {
    if (a == null)
        throw new NullPointerException();

    return new AbstractList() {
        public Object get(int i) {
            return new Integer(a[i]);
        }

        public int size() {
            return a.length;
        }
    };
}
```

```

        public Object set(int i, Object o) {
            int oldVal = a[i];
            a[i] = ((Integer)o).intValue();
            return new Integer(oldVal);
        }
    };
}

```

当你考虑一个List实现应该为你完成哪些工作的时候，可以看出，这个例子充分演示了骨架实现的强大能力。顺便提一下，这个例子是一个Adapter[Gamma95, p.139]，它使得一个int数组可以被看做一個Integer实例列表。由于在int值和Integer实例之间来回转换的开销，它的性能不会非常好。注意，这个例子只提供了一个静态工厂，并且这个类是一个不可被访问的匿名类（*anonymous class*）（见第18条），它被隐藏在静态工厂的内部。

骨架实现的优美之处在于，它们为抽象类提供了实现上的帮助，但又没有强加“抽象类被用作类型定义时候”所特有的严格限制。对于一个接口的大多数实现来讲，扩展骨架实现类是一个很显然的选择，但也确实只是一个选择而已。如果一个预先已经定义好的类无法扩展骨架实现类，那么，这个类总是可以手工实现这个接口。尽管如此，骨架实现类仍然能够有助于接口的实现。实现了这个接口的类可以把对于接口方法的调用，转发到一个内部私有类的实例上，而这个内部私有类扩展了骨架实现类。这项技术被称为模拟多重继承（*simulated multiple inheritance*），它与第14条讨论的包装类模式有密切的关系。这项技术具有多重继承的绝大多数优点，并且避免了相应的缺陷。
86

编写一个骨架实现类相对比较简单，只是有点单调乏味。首先你要认真研究接口，并且确定哪些方法是最基本的（primitive），其他的方法在实现的时候将以它们为基础。这些基本方法将是骨架实现类中的抽象方法。然后，你必须为接口中所有其他的方法提供具体的实现。例如，下面是Map.Entry接口的骨架实现类，在本书写作的时候，这个类还没有被包含到Java平台库中，但是它应该被包含进去：

```

// Skeletal Implementation
public abstract class AbstractMapEntry implements Map.Entry {
    // Primitives
    public abstract Object getKey();
    public abstract Object getValue();

    // Entries in modifiable maps must override this method
    public Object setValue(Object value) {
        throw new UnsupportedOperationException();
    }

    // Implements the general contract of Map.Entry.equals
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry arg = (Map.Entry)o;

```

```

        return eq(getKey(), arg.getKey()) &&
           eq(getValue(), arg.getValue());
    }

    private static boolean eq(Object o1, Object o2) {
        return (o1 == null ? o2 == null : o1.equals(o2));
    }

    // Implements the general contract of Map.Entry.hashCode
    public int hashCode() {
        return
            (getKey() == null ? 0 : getKey().hashCode()) ^
            (getValue() == null ? 0 : getValue().hashCode());
    }
}

```

[87]

因为骨架实现类不是为了继承的目的而设计的，所以你应该遵从第15条中介绍的所有关于设计和文档的指导原则。为了简短起见，上面例子中的文档注释部分被省略掉了，但是对于骨架实现类而言，好的文档绝对是非常必要的。

使用抽象类来定义允许多个实现的类型，比使用接口有一个明显的优势：**抽象类的演化比接口的演化要容易得多**。如果在后续的发行版本中，你希望在抽象类中增加一个新的方法，那么，你总是可以增加一个具体方法，它包含了一个合理的默认实现。然后，该抽象类的所有已有的实现都自动提供了这个新的方法。对于接口，这样做是行不通的。

从一般意义上讲，要想在一个公有接口中增加一个方法，而不打破现有的、已经在使用这个接口的所有程序，这是不可能的。在此之前实现该接口的类将会漏掉新增加的方法，所以根本就不能通过编译。你可以在为接口增加新方法的同时，也为骨架实现类增加同样的新方法，这样可以在一定程度上减小由此带来的破坏，但是，这样做并没有真正解决问题。所有不从骨架实现类继承的接口实现仍然会被打破。

因此，设计公有的接口要非常谨慎。一旦一个接口被公开发行，并且已被广泛实现，再想改变这个接口是不可能的。你必须在第一次设计的时候就保证接口是正确的。如果一个接口包含一个微小的瑕疵，它将会一直影响你以及接口的用户。如果一个接口具有严重的缺陷，它可以导致API彻底失败。在发行一个新的接口的时候，最好的做法是，在接口被“冻结”之前，尽可能让更多的程序员用尽可能多的方式来实现这个新接口。这使得你可以尽早地发现任何可能存在的缺陷，并且还有机会改正这些缺陷。

总而言之，接口通常是定义具有多个实现的类型的最佳途径。这条规则的一个例外情形是：当演化的容易性比灵活性和功能更为重要的时候。在这样的情况下，你应该使用抽象类来定义类型，但是你也必须理解抽象类的局限性，并且确保可以接受这些局限性。如果你导出了一个重要的接口，你也应该考虑同时提供一个骨架实现类。最后，你应该尽可能地谨慎设计所有的公有接口，并且通过编写多个实现来对它们进行全面的测试。

[88]

第17条：接口只是被用于定义类型

当一个类实现了一个接口的时候，这个接口被用做一个类型（*type*），通过此类型可以引用这个类的实例。因此，一个类实现了一个接口，就表明客户可以对这个类的实例实施某些动作。为了任何其他目的而定义接口是不合适的。

有一种接口被称为常量接口（*constant interface*），它不满足上面的条件。这样的接口没有包含任何方法，它只包含静态的final域，每个域都导出一个常量。如果一个类要使用这些常量，它只要实现这个接口，就可以避免用类名来修饰常量名。下面是一个例子：

```
// Constant interface pattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER = 6.02214199e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

常量接口模式是对接口的不良使用。一个类要在内部使用某些常量，这纯粹是实现细节。实现一个常量接口，会导致把这样的实现细节泄露到该类的导出API中。实现常量接口对于这个类的用户来讲也没有什么价值。实际上，这样做反而会使他们更加糊涂。更坏的是，它代表了一种承诺：如果在将来的发行版本中，这个类被修改了，它不再需要使用这些常量了，那么它仍必须要实现这个接口，以保证二进制兼容性。如果一个非final类实现了一个常量接口，它的所有子类的名字空间会被接口中的常量污染。

在Java平台库中有几个常量接口，例如java.io.ObjectStreamConstants。这些接口应该被认为是反常的事物，不值得效仿。

如果要导出常量，你可以有几种选择方案。如果这些常量与某个已有的类或者接口紧密地联系在一起，那么你应该把这些常量添加到这个类或者接口中。例如，Java平台库中所有的数值包装类，比如Integer和Float，都导出了MIN_VALUE和MAX_VALUE常量。如果这些常量最好被看做…个枚举类型的成员，那么你应该用一个类型安全枚举类（*typesafe enum class*）（见第21条）来导出这些常量。否则的话，你应该使用一个不可被实例化的工具类（*utility class*）（见第3条）来导出这些常量。下面的例子是前面的PhysicalConstants例子的工具类版本：

```
// Constant utility class
public class PhysicalConstants {
```

```
private PhysicalConstants() { } // Prevents instantiation

public static final double AVOGADROS_NUMBER = 6.02214199e23;
public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
public static final double ELECTRON_MASS = 9.10938188e-31;
}
```

虽然PhysicalConstants的工具类版本要求客户用一个类名来修饰这些常量名，但是，对于实际的API来说，这是很小的代价。并且，Java语言最终有可能允许直接导入静态域。同时，你只须把一些频繁使用的常量放到局部变量中，或者放到私有的静态域中，就可以使额外的键盘录入工作量降低到最小。下面是一个例子：

```
private static final double PI = Math.PI;
```

总之，接口应该只是被用来定义类型的，它们不应该被用来导出常量。

90

第18条：优先考虑静态成员类

嵌套类（*nested class*）是指被定义在另一个类的内部的类。嵌套类存在的目的应该只是为它的外围类提供服务。如果一个嵌套类将来可能会用于其他的某个环境中，那么它应该是顶层类（*top-level class*）。嵌套类有四种：静态成员类（*static member class*）、非静态成员类（*nonstatic member class*）、匿名类（*anonymous class*）和局部类（*local class*）。除了第一种之外，其他三种都被称为内部类（*inner class*）。本条目将告诉你什么时候应该使用哪种嵌套类，以及这样做的原因。

静态成员类是一种最简单的嵌套类。最好把它看做是一个普通的类，只是碰巧被声明在另一个类的内部而已。它可以访问外围类的所有成员，包括那些声明为私有的成员。静态成员类是外围类的一个静态成员，与其他的静态成员一样，也遵守同样的可访问性规则。如果它被声明为私有的，那么它只能在外围类的内部才可以被访问等等。

静态成员类的一种通常用法是作为公有的辅助类，仅当与它的外部类一起使用时才有意义。例如，考虑一个类型安全枚举类，它描述了一个计算器应该支持的各种操作（见第21条）。`Operation`类应该是`Calculator`类的一个公有的静态成员类，然后，`Calculator`类的客户可以用诸如`Calculator.Operation.PLUS`和`Calculator.Operation_MINUS`这样的形式来访问这些操作。本条目后面演示了这种用法。

从语法上讲，静态成员类和非静态成员类之间惟一的区别是，静态成员类的声明中包含修饰符`static`。尽管它们的语法非常相似，但是这两种嵌套类有很大的不同。非静态成员类的每一个实例都隐含着与外围类的一个外围实例（*enclosing instance*）紧密关联在一起。在非静态成员类的实例方法内部，调用外围实例上的方法是有可能的，或者使用经过修饰的`this`也可以得到一个指向外围实例的引用[JLS, 15.8.4]。如果一个嵌套类的实例可以在它的外围类的实例之外独立存在，那么这个嵌套类不可能是一个非静态成员类；在没有外围实例的情况下要想创建非静态成员类的实例是不可能的。

当非静态成员类的实例被创建的时候，它和外围实例之间的关联关系也随之被建立起来；而且，这种关联关系以后不能被修改。通常情况下，当外围类的某个实例方法的内部调用非静态成员类的构造函数时，这种关联关系被自动建立起来。使用表达式`enclosingInstance.new MemberClass(args)`来手工建立这种关联关系也是有可能的，但是很少使用。正如你所预料的那样，这种关联关系需要消耗非静态成员类实例的空间，并且在构造的过程中也增加了时间开销。

91 非静态成员类的一种通常用法是定义一个`Adapter`[Gamma95, p.139]，它允许外部类的--

个实例被看做另一个不相关的类的实例。例如，Map接口的实现往往使用非静态成员类来实现它们的集合视图（*collection view*），这些集合视图是由Map的keySet、entrySet和Value方法返回的。类似地，诸如Set和List这样的集合接口的实现往往也使用非静态成员类来实现它们的迭代器（iterator）：

```
// Typical use of a nonstatic member class
public class MySet extends AbstractSet {
    ... // Bulk of the class omitted

    public Iterator iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator {
        ...
    }
}
```

如果你声明的成员类不要求访问外围实例，那么请记住把static修饰符放到成员类的声明中，使它成为一个静态成员类，而不是一个非静态成员类。如果你省略了static修饰符，则每个实例都将包含一个额外的指向外围对象的引用。维护这份引用要消耗时间和空间，但又没有相应的好处。如果在没有外围实例的情况下，你也要分配实例的话，则不能使用非静态成员类，因为非静态成员类的实例必须要有一个外围实例。

私有静态成员类的一种通常用法是用来代表外围类对象的组件。例如，考虑一个Map实例，它把键（key）和值（value）关联起来。Map实例的内部通常有一个Entry对象对应于Map中的每一对键-值。虽然每一个Entry都与一个Map关联，但是Entry上的方法（getKey、getValue和setValue）并不需要访问该Map。因此，使用非静态成员来表示Entry是很浪费的，私有的静态成员类是最佳的选择。如果你偶然地省略掉了Entry声明中的static修饰符，该Map仍然可以工作，但是每个Entry中将会包含一个指向该Map的引用，这是多余的，只会浪费空间和时间。

[92]

如果问题中的嵌套类是一个导出类的公有的或受保护的成员，则毫无疑问，在静态和非静态成员类之间做出正确的选择是非常重要的。在这种情况下，该成员类是导出的API中的一个元素，在后续的发行版本中，要想不违反二进制兼容性，就不能从非静态成员类变为静态成员类。

匿名类不同于Java程序设计语言中的其他任何语法单元，正如你所想像的，匿名类没有名字。它不是外围类的一个成员。它并不与其他的成员一起被声明，而是在被使用的点上同时被声明和实例化。匿名类可以出现在代码中任何允许表达式出现的地方。匿名类的行为与静态的或者非静态的成员类非常类似，取决于它所在的环境：如果匿名类出现在一个非静态的环境中，则它有一个外围实例。

匿名类的适用性有一些限制。因为它们同时被声明和实例化，所以匿名类只能被用在代码中它将被实例化的那个点上。因为匿名类没有名字，所以在它们被实例化之后，就不能够再对它们进行引用，如果不是这种情况就不能使用匿名类。匿名类通常只实现了其接口中或者超类中的方法。它们不会声明任何新的方法，因为不存在可命名的类型可以访问新增加的方法。因为匿名类出现在表达式的中间，所以它们应该非常简短，可能是20行或者更少。太长的匿名类会影响程序的可读性。

匿名类的一个通常用法是创建一个函数对象 (*function object*)，比如Comparator实例。例如，下面的方法调用对一组字符串按照其长度进行排序：

```
// Typical use of an anonymous class
Arrays.sort(args, new Comparator() {
    public int compare(Object o1, Object o2) {
        return ((String)o1).length() - ((String)o2).length();
    }
});
```

匿名类的另一个常见用法是创建一个过程对象 (*process object*)，比如Thread、Runnable或者TimerTask实例。第三个常见的用法是在一个静态工厂方法的内部（参见第16条中的intArrayAsList方法）。第四个常见的用法是在复杂的类型安全枚举类型（它要求为每个实例提供单独的子类）中，用于公有的静态final域的初始化器中（见第21条中的Operation类）。正如前面所建议的那样，如果Operation类是Calculator的一个静态成员类，那么每个Operation常量是双重嵌套类：

```
// Typical use of a public static member class
public class Calculator {
    public static abstract class Operation {
        private final String name;

        Operation(String name) { this.name = name; }

        public String toString() { return this.name; }

        // Perform arithmetic op represented by this constant
        abstract double eval(double x, double y);

        // Doubly nested anonymous classes
        public static final Operation PLUS = new Operation("+") {
            double eval(double x, double y) { return x + y; }
        };
        public static final Operation MINUS = new Operation("-") {
            double eval(double x, double y) { return x - y; }
        };
        public static final Operation TIMES = new Operation("*") {
            double eval(double x, double y) { return x * y; }
        };
        public static final Operation DIVIDE = new Operation("/") {
            double eval(double x, double y) { return x / y; }
        };
    }
}
```

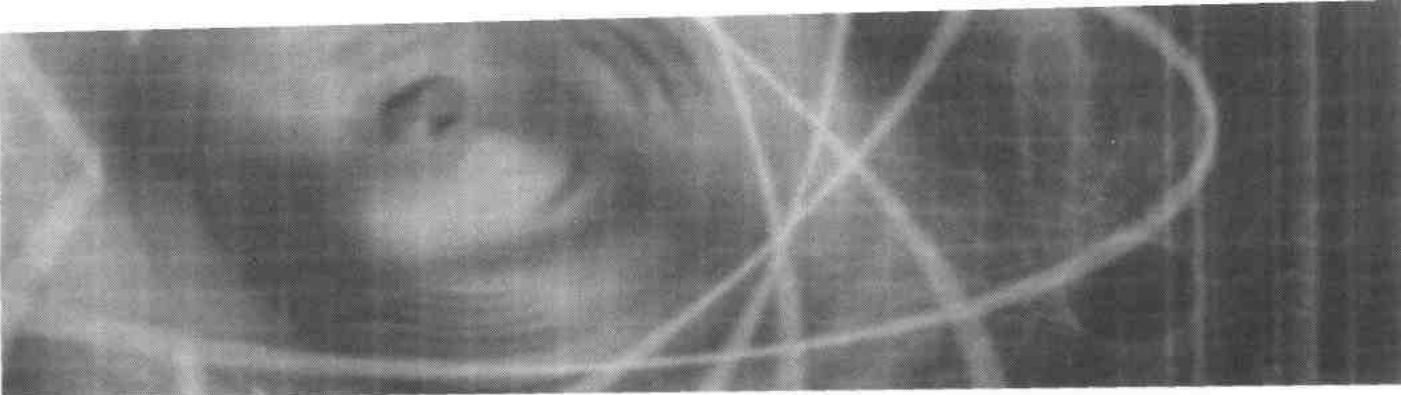
```
// Return the results of the specified calculation
public double calculate(double x, Operation op, double y) {
    return op.eval(x, y);
}
```

局部类是四种嵌套类中最少使用的类。在任何“可以声明局部变量”的地方，都可以声明局部类，并且局部类也遵守同样的作用域规则。局部类与其他三种嵌套类的每一种都有一些共同属性。与成员类一样，局部类有名字，可以被重复使用。与匿名类一样，当且仅当局部类被用于非静态环境下的时候，它们才有外围实例。与匿名类一样，它们必须非常简短，以便不会损害外围方法或者初始化器的可读性。

[94]

简而言之，共有四种不同的嵌套类，每一种都有自己的用途。如果一个嵌套类需要在单个方法之外仍然是可见的，或者它太长了，不适合于放在一个方法内部，那么应该使用成员类。如果成员类的每个实例都需要一个指向其外围实例的引用，则把成员类做成非静态的；否则就做成静态的。假设一个嵌套类属于一个方法的内部，如果你只需要在一个地方创建它的实例，并且已经有了一个预先存在的类型可以说明这个类的特征，则把它做成匿名类；否则就做成局部类。

[95]



第5章

C语言结构的替代

Java程序设计语言与C程序设计语言有许多相似之处，但是C语言中有些语法结构被省略掉了。在大多数情况下，省略这些C语言结构的原因很显然，在没有这些语言结构的情况下该如何处理问题也很清楚。本章针对几种被省略的C语言结构介绍了替代的方法，这些替代方法并不那么显而易见。

贯穿本章所有条目的共同线索是，所有这些被省略的语言结构都是面向数据的，而不是面向对象的。Java程序设计语言提供了一个功能强大的类型系统，本章介绍的替代方法充分利用了这个类型系统，获得了比它们替代的C语言结构更高质量的抽象。

你可以选择跳过本章的内容，即使这样，读一读第21条的内容仍然是很有价值的，它讨论了类型安全枚举（*typesafe enum*）模式，用来代替C语言中的enum类型。在本书写作的时候，这种模式尚未广为人知，但是比起当前常用的一些方法而言，它具有一些显著的优点。

第19条：用类代替结构

C语言的结构（`struct`）在Java语言中被省略了，因为结构（`struct`）能做的事情，类（`class`）都能做到，而且它可以做得更多。一个结构只是把多个数据域组织到单个对象中；一个类可以把各种操作与这样得到的数据对象关联起来，可以把这些数据域隐藏起来，不让对象的用户看到。换句话说，一个类可以把数据封装（*encapsulate*）到一个对象中，并且只有对象的方法才可以访问这些数据，从而使得实现者可以随着时间的不同，自由地选择数据的表示形式（见第12条）。

97 Java程序设计语言刚刚而世时，有些C程序员相信，在某些情况下，用类来代替结构显得分量太重了，但是实际上并不是这样。让一个类退化到只包含一些数据域，这样的类与C语言的结构大致上是等价的：

```
// Degenerate classes like this should not be public!
class Point {
    public float x;
    public float y;
}
```

因为客户可以通过数据域直接访问这样的类，所以它们并没有提供封装性的好处。如果不改变这种类的API，就不能改变它的数据表示形式，同样也不能强加任何约束条件；当一个域被修改的时候，你不可能采取任何辅助的动作。坚持面向对象程序设计的程序员会感到这样的类非常令人厌烦，而应该代之以包含私有域和公有访问方法（accessor method）的类：

```
// Encapsulated structure class
class Point {
    private float x;
    private float y;

    public Point(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public float getX() { return x; }
    public float getY() { return y; }

    public void setX(float x) { this.x = x; }
    public void setY(float y) { this.y = y; }
}
```

毫无疑问，当这样的类成为公有类的时候，这些面向对象程序设计思想坚持者们是正确的。如果一个类有可能会被它所在包的外部访问的话，那么，谨慎的程序员会提供访问方法（accessor method），以保留将来改变该类内部表示的灵活性。如果一个公有类暴露了它的数据域，那么要想在将来的版本中改变它的内部数据表示是不可能的，因为公有类的客户代码已经遍布各处了。

然而，如果一个类是包级私有的，或者是一个私有的嵌套类，则直接暴露其数据域并没有本质的错误——假设这些数据域确实描述了该类所提供的抽象。这种做法比访问方法的做法会产生更少的视觉混乱，无论是在类定义中，还是在使用该类的客户代码中。虽然客户代码与该类的内部表示紧紧绑在一起，但是这些代码被限定在包含该类的包的内部。在某些情况下确实有必要改变内部数据表示，这时，不改变包外部的任何代码而改变内部表示是有可能的。在私有嵌套类的情况下，改变的作用范围被进一步限制在外围类中。

98

Java平台库中有几个类违反了“公有类不应该直接暴露数据域”的告诫。显著的例子包括java.awt包中的Point和Dimension类。它们是不值得效仿的例子，相反，应该被当作反面的警告示例。正如第37条中所讲述的，“暴露Dimension类的内部数据”这项决定导致了严重的性能问题，而且，要想解决该问题必然会影响到客户。

99

第20条：用类层次来代替联合

C语言中的union最常见的用法，是用来定义可以容纳多种数据类型的数据结构。这样的数据结构往往至少包含两个域：一个联合（union）和一个标签（tag）。标签域只是一个普通的域，用来指示联合（union）中容纳的是哪一种可能的类型。标签通常是某一个枚举（enum）类型。像这样包含一个联合和一个标签的数据结构有时候也被称为可区分的联合（discriminated union）。

在下面的C例子中，shape_t类型是一个可区分的联合，被用来表示一个矩形或者一个圆。area函数用一个指向shape_t结构的指针作为参数，返回它的面积，如果结构无效的话，返回-1.0：

```
/* Discriminated union */
#include "math.h"
typedef enum {RECTANGLE, CIRCLE} shapeType_t;

typedef struct {
    double length;
    double width;
} rectangleDimensions_t;

typedef struct {
    double radius;
} circleDimensions_t;

typedef struct {
    shapeType_t tag;
    union {
        rectangleDimensions_t rectangle;
        circleDimensions_t   circle;
    } dimensions;
} shape_t;

double area(shape_t *shape) {
    switch(shape->tag) {
        case RECTANGLE: {
            double length = shape->dimensions.rectangle.length;
            double width  = shape->dimensions.rectangle.width;
            return length * width;
        }
        case CIRCLE: {
            double r = shape->dimensions.circle.radius;
            return M_PI * (r*r);
        }
        default: return -1.0; /* Invalid tag */
    }
}
```

100

Java程序设计语言的设计者决定省略union结构的原因是，有一种更好的机制可以定义单个数据类型，它能够表示不同类型的对象，这种机制就是子类型化（*subtyping*）。可区分的

联合实际上只是类层次结构的一种简单的效仿而已。

为了把一个可区分的联合变换为一个类层次，先定义一个抽象类，在抽象类中为每个操作定义一个抽象方法，其行为取决于标签的值。在上面的例子中，只有一个这样的操作：area。这个抽象类是类层次的根（root）。如果还有其他的操作其行为不依赖于标签的值，则把这样的操作变成根类（root class）中的具体方法。类似地，如果在可区分的联合中，除了标签和联合两个域之外，还有其他的数据域，那么，这些域代表了对于所有类型都共同的数据，所以应该被加到根类中。在上面的例子中，不存在这种类型独立的操作或者数据域。

接下去，对于可区分联合所能够表示的每一种类型，都在根类下面定义一个具体子类。在前面的例子中，这样的类型为矩形（rectangle）和圆（circle）。在每个子类中都包含特定于该类型的数据域。在例子中，radius是特定于圆的，length和width是特定于矩形的。同时在每个子类中还包括针对根类中每个抽象方法的适当的实现。以下是前面的可区分联合例子所对应的类层次：

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double area() { return length * width; }
}
```

101

类层次与可区分的联合相比有多方面的好处。其中最主要的好处在于，类层次提供了类型安全性。在例子中，每一个Shape实例或者是一个有效的Circle，或者是一个有效的Rectangle。产生一个完全没用的垃圾shape_t结构是非常简单的，因为C语言本身并没有强迫标签和联合之间的关联关系。如果标签域的值指示这个shape_t对象代表一个矩形，但是联合域却被设置为一个圆，则必将导致失败。即使一个可区分的联合已经被正确地初始化了，把它传递给一个与其标签值不一致的函数也是有可能的。

类层次的第二个好处在于，代码非常简洁明了。可区分的联合往往使用如下的样板，其代

码很零乱：声明enum类型，再声明标签域，针对标签域的switch语句，处理预料之外的标签值等等。可区分联合的代码缺乏可读性，针对各种类型的操作混在一起，而不是按照类型分开的。

类层次的第三个好处在于，它很容易扩展，即使是多方在独立地工作。为了扩展一个类层次，只需简单地添加新的子类。如果你忘了改写超类中某一个抽象方法，编译器将给出相应的提示。为了扩展一个可区分的联合，你需要访问源代码；你必须在enum类型中增加一个新的值，在针对该可区分联合的每个操作中的switch语句中增加一个新的case。最后，你还必须重新编译。如果你忘了为某个方法提供一个新的case，则只有到运行时刻才会发现问题。然后，只有仔细地检查所有不可识别的标签值的情形，产生适当的错误消息，你才能发现问题所在。

类层次的第四个好处在于，它可以反映出这些类型之间本质上的层次关系，从而允许更强的灵活性，以及更好的编译时类型检查。假设前面例子中的可区分联合也允许表达正方形。那么，类层次结构能够反映出这样的事实：正方形是一种特殊的矩形（假设两者都是不可变的）：

```
class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }

    double side() {
        return length; // or equivalently, width
    }
}
```

[102]

为了替代例子中的可区分联合，这样编写类层次的做法并不是唯一的。层次结构中有一些具体的设计抉择值得注意。上述类层次中的类，除了Square之外，其他的类都可以直接访问它们的域，而无需访问方法(accessor method)。这么做是为了简短的原因，如果这些类是公有的，则这样做是不可接受的(见第19条)。类层次中的类都是不可变的，这样做并不总是合适的，但通常是一件好事(见第13条)。

由于Java程序设计语言并没有提供union结构，所以你可能会认为在Java中不存在实现可区分联合的危险；但是，编写出具有同样这些缺点的代码则是有可能的。无论何时当你想编写一个类，其中包含一个显式的标签域的时候，应该考虑一下，这个标签是否可以被取消、这个类是否可以用一个类层次来代替。

C语言中union结构的另一个用途与可区分的联合完全没有关系，它涉及到查看一片数据的内部表示，目的在于突破类型系统的限制。下面的C语言代码片断演示了这种用法，它打印出一个float对象的机器相关的十六进制表示形式：

```
union {
    float f;
    int   bits;
} sleaze;

sleaze.f = 6.699e-41; /* Put data in one field of union... */
printf("%x\n", sleaze.bits); /* ...and read it out the other. */
```

虽然这种做法很有用，特别是对于系统程序设计而言，但是，这种不可移植的用法在Java程序设计语言中没有对应的做法。实际上，它与Java语言的精神背道而驰，Java语言保证类型安全性，并且尽最大可能将程序员与机器相关的内部表示隔离开。

java.lang包中包含有相应的方法可以把浮点数转换成“位表示形式”，但是，这些方法根据精确指定的位表示形式来进行转换，以保证可移植性。下面的代码片断与前面的C片断基本上是等价的，它可以保证不管在哪里运行都会打印出同样的结果：

```
System.out.println(
    Integer.toHexString(Float.floatToIntBits(6.699e-41f)));
```

103

第21条：用类来代替enum结构

Java语言省略了C语言中的enum结构。从名义上讲，enum结构定义了一个枚举类型(*enumerated type*)：它的合法值是由一组固定的常量组成的。不幸的是，enum结构在定义枚举类型方面并没有做得很好。它只是定义了一组被命名的整数常量，在类型安全性和使用方便性方面没有提供任何帮助。下面的C语言代码是合法的：

```
typedef enum {FUJI, PIPPIN, GRANNY_SMITH} apple_t;
typedef enum {NAVEL, TEMPLE, BLOOD} orange_t;
orange_t myFavorite = PIPPIN; /* Mixing apples and oranges */
```

但是下面的代码就非常可怕：

```
orange_t x = (FUJI - PIPPIN)/TEMPLE; /* Applesauce! */
```

enum结构并没有为它产生的常量建立起一个名字空间。因此，下面的声明试图重用已有的名字，会与orange_t的声明产生冲突：

```
typedef enum {BLOOD, SWEAT, TEARS} fluid_t;
```

用enum结构定义的类型是非常脆弱的。除非小心地维护好所有先前存在的常量值，否则，在enum类型中增加新的常量但又不重新编译客户代码，会引起不可预知的行为。多个开发方不能独立地为这样的类型增加常量，因为新的枚举常量可能会产生冲突。enum结构也没有提供很便利的办法把它的枚举常量转换成可打印的字符串，或者枚举出一个类型中所有的常量。

不幸的是，Java程序设计语言中最常用的针对枚举类型的模式，与C语言的enum结构具有同样的缺点。下面是一个例子：

```
// The int enum pattern - problematic!
public class PlayingCard {
    public static final int SUIT_CLUBS      = 0;
    public static final int SUIT_DIAMONDS   = 1;
    public static final int SUIT_HEARTS     = 2;
    public static final int SUIT_SPADES     = 3;
    ...
}
```

你也许碰到过这种模式的变形，只不过你碰到的是String常量，而不是int常量。这样的变形永远也不应该被使用。虽然它为这些常量提供了可打印的字符串，但是它会导致性能问题，因为它依赖于字符串的比较操作。而且，它会导致初级用户把字符串常量硬编码到客户的代码中，而不是使用适当的域(field)名。如果这样的硬编码字符串常量包含一个书写的错误，那么，这样的错误在编译的时候不会被检测到。但是在运行的时候会导致出错。

幸运的是，Java程序设计语言提出了另一种方案，可以避免常见的int和String模式的所有缺点，并且还提供了许多额外的好处。这种方案被称为类型安全枚举（*typesafe enum*）模式。不幸的是，它还尚未被广泛知晓。基本的想法非常简单：定义一个类来代表枚举类型的单个元素，并且不提供任何公有的构造函数。相反，提供公有的静态final域，使枚举类型中的每一个常量都对应一个域。下面演示了这种模式最简单的形式：

```
// The typesafe enum pattern
public class Suit {
    private final String name;

    private Suit(String name) { this.name = name; }

    public String toString() { return name; }

    public static final Suit CLUBS = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS = new Suit("hearts");
    public static final Suit SPADES = new Suit("spades");
}
```

因为客户没有办法创建这个类的对象，也无法扩展这个类，所以，除了通过这些公有的静态final域导出的Suit对象之外，永远不会再有其他的对象存在。即使这个类没有被声明为final，也没有办法对它进行扩展：子类的构造函数必须调用超类的构造函数，但是这个类中没有可访问的构造函数。

顾名思义，类型安全枚举模式提供了编译时的类型安全性。如果你声明了一个方法，它的一个参数为Suit类型，则可以保证，任何传入的非null的对象引用一定表示了这四种纸牌花色（suit）之一。企图传递一个类型不正确的对象将会在编译的时候被捕捉到，就如同企图“将一种枚举类型的表达式赋给另一种枚举类型的变量”一样。多个“类型安全枚举类”可以包含相同名字的枚举常量，它们可以在一个系统中和平共处，因为每一个类都有自己的名字空间。105

新的常量可以被加入到一个类型安全枚举类中，而无需重新编译客户代码，因为包含枚举常量的公有静态对象引用域为客户和枚举类之间提供了一层隔离层。常量本身并没有被编译到客户代码中，而在更常见的int模式和String变形之中，常量被编译到客户代码中。

因为类型安全的枚举类是完全意义上的类，所以你可以像以前介绍的那样改写toString方法，从而允许其中的值被转换为可打印的字符串。如果你愿意的话，还可以走得更远，用标准的形式对类型安全枚举类进行国际化。注意，只有toString方法用到了字符串名字[⊕]，equals比较操作并没有用到它，因为从Object继承得来的equals实现只执行引用比较。

[⊕] 即name域。——译注

更为一般地，你可以在类型安全枚举类中增加一些适当的方法，以此壮大这样的类。例如，例子中的Suit类可能希望增加一个能够返回纸牌颜色的方法，或者一个能够返回纸牌图案的方法。一个类可以从最简单的类型安全枚举类开始，随时间的推移逐步演化为一个具有全面特性的抽象。

因为任何方法都可以被添加到类型安全枚举类中，所以它们也可以实现任何接口。例如，假设你希望Suit实现Comparable接口，以便客户可以根据花色排列桥牌。以下代码是在原来模式的基础上做了细微的变化，它实现了这种功能。静态变量nextOrdinal被用做序数，在一个实例被创建的时候，为它分配一个序数。compareTo方法使用这些序数来对实例进行排序：

```
// Ordinal-based typesafe enum
public class Suit implements Comparable {
    private final String name;

    // Ordinal of next suit to be created
    private static int nextOrdinal = 0;

    // Assign an ordinal to this suit
    private final int ordinal = nextOrdinal++;

    private Suit(String name) { this.name = name; }

    public String toString() { return name; }

    public int compareTo(Object o) {
        return ordinal - ((Suit)o).ordinal;
    }

    public static final Suit CLUBS      = new Suit("clubs");
    public static final Suit DIAMONDS   = new Suit("diamonds");
    public static final Suit HEARTS    = new Suit("hearts");
    public static final Suit SPADES    = new Suit("spades");
}
```

106

因为类型安全枚举类型的常量是对象，所以你可以把这些常量放到集合中。例如，假设你希望Suit类导出一个按标准顺序的、非可变的纸牌列表。你只需在类中加入下面两个域声明：

```
private static final Suit[] PRIVATE_VALUES =
{ CLUBS, DIAMONDS, HEARTS, SPADES };
public static final List VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

与最简单形式的类型安全枚举模式不同，基于序数形式的类型安全枚举类只需少量的工作就可以支持序列化（参见第10章）。仅仅在类的声明中增加implements Serializable是不够的，你还必须提供一个readResolve方法（见第57条）：

```
private Object readResolve() throws ObjectStreamException {
    return PRIVATE_VALUES[ordinal]; // Canonicalize
}
```

这个方法被序列化系统（*serialization system*）自动调用，它可以避免在反序列化（*deserialization*）的时候产生重复的常量。这样可以保证每个枚举常量只有一个对象作为它的代表，所以就不再需要改写*Object.equals*。如果没有这种保证的话，则当*Object.equals*面对两个虽然相等但是不同的枚举常量时，它会报告不相等的结果，这显然是错误的。注意，*readResolve*方法引用了*PRIVATE_VALUES*数组，所以，即使不导出*VALUES*，你也必须要声明*PRIVATE_VALUES*数组。同时还要注意，*readResolve*方法并没有用到*name*域，所以该域可以被做成*transient*的，而且也应该这样。

这样得到的类有点脆弱，针对新值的构造函数调用必须要出现在所有已有的值之后，从而可以保证以前已经被序列化的实例在反序列化的时候，不会改变它们的值，因为一个枚举常量的序列化形式（见第55条）只包含它的序数。如果属于某个序数的枚举常量改变了，那么在反序列化的时候，具有该序数的序列化常量将会获得新的值。

若一个枚举常量只被用于包含该安全枚举类的包中，对于这样的常量，也许会有一个或者多个行为与它相关联。这样的行为最好作为该类中的包级私有方法来实现。于是，每个枚举常量都支持一组隐藏的行为，这使得包含该枚举类型的包在看到这样的常量的时候可以采取适当的动作。[107]

如果一个类型安全枚举类有一些方法，对于不同的常量其行为有很大的变化，那么你应该为每一个常量用一个单独的私有类或者匿名的内部类。这使得每个常量对于每个这样的方法都有自己的实现，而且也可以自动调用正确的方法实现。另一种解决办法是，用多路分支（*multiway branch*）的结构来实现每个这样的方法，如何分支取决于该方法是在哪个常量上被调用的。这种办法很不美观，也很容易出错，而且其性能可能不如虚拟机自动分发方法调用更好。

下面的类型安全枚举类说明了上面段落中讲述的两种技术。*Operation*类代表了一个基本的四功能计算器所执行的操作，在定义该类的包的外部，对于*Operation*常量所能够做的事情是调用*Object*方法（*toString*、*hashCode*、*equals*等等）。然而，在包的内部，你可以执行该常量所代表的算术运算。可以想像，这个包会导出某个高层的计算器对象，它会导出一个或者多个方法，它们以*Operation*常量作为参数。注意，*Operation*本身是一个抽象类，只包含一个包级私有的抽象方法*eval*，它执行适当的算术运算。针对每个常量都定义了一个匿名的内部类，所以每个常量都可以定义它自己的*eval*方法：

```
// Typesafe enum with behaviors attached to constants
public abstract class Operation {
    private final String name;

    Operation(String name) { this.name = name; }

    public String toString() { ... }
    public int hashCode() { ... }
    public boolean equals(Object o) { ... }

    protected abstract long eval(Operation o);
}
```

```

public String toString() { return this.name; }

// Perform arithmetic operation represented by this constant
abstract double eval(double x, double y);

public static final Operation PLUS = new Operation("+") {
    double eval(double x, double y) { return x + y; }
};

public static final Operation MINUS = new Operation("-") {
    double eval(double x, double y) { return x - y; }
};

public static final Operation TIMES = new Operation("*") {
    double eval(double x, double y) { return x * y; }
};

public static final Operation DIVIDED_BY =
    new Operation("/") {
        double eval(double x, double y) { return x / y; }
    };
}

```

一般来讲，类型安全枚举类在性能上可以与int枚举常量相媲美。类型安全枚举类的两个不同的实例永远也不会代表同样的值，所以可以使用“引用的同一性比较”来检查逻辑上的相等关系，它的速度非常快。类型安全枚举类的客户可以使用`==`操作符，来代替`equals`方法；而且可以保证两者的结果是相同的，并且`==`操作符可能会更快一些。

如果一个类型安全枚举类具有普遍适用性，那么它应该成为一个顶层类（top-level class）；如果它只是被用在一个特定的顶层类中，那么它应该成为该顶层类的一个静态成员类（见第18条）。例如，`java.math.BigDecimal`类包含了一组int枚举常量，它们分别代表了十进制小数的舍入模式（*rounding mode*）。这些舍入模式提供了一个非常有用的抽象，但是这种抽象本质上又不属于`BigDecimal`类；对于它们来说，如果作为一个独立的`java.math.RoundingMode`类来实现，可能会更好。这样就可以鼓励任何需要舍入模式的程序员重用这些舍入模式，从而增强了API之间的一致性。

如前面给出的两种Suit实现所示，基本的类型安全枚举模式是固定的（*fixed*）：用户要想在枚举类型中增加新的元素是不可能的，因为枚举类中没有用户可访问的构造函数。这使得这样的类实际上是final的，无论它的声明中是否用到了final访问修饰符。这通常正是你所期望的，但是偶尔，你可能希望一个类型安全枚举类是可扩展的（*extensible*）。例如，如果你用一个类型安全枚举类来表示图像编码格式，并且也希望第三方能够增加对于新格式的支持，那么这可能就是可扩展的类型安全枚举类的情形。

为了使得一个类型安全枚举类可以被扩展，只要提供一个受保护的构造函数就可以。然后，其他人就可以扩展这个类，并且在子类中增加新的常量。你不必像使用int枚举模式那样担心枚举常量会有冲突。类型安全枚举模式的这种扩展变形充分利用了包的名字空间，为可扩展的枚举类型创建了一个“被神奇地管理起来的”名字空间。多个组织可以在相互不知情的情

况下扩展枚举类型，而且他们的扩展永远不会发生冲突。

109

仅仅在可扩展的枚举类型中增加一个元素并不能确保新的元素会被全面支持：以该枚举类型的元素作为参数的方法，必须要处理“被传入一个连程序员都不认识的元素”的可能性。基于固定模式枚举类型的多路分支方法是很值得怀疑的；但对于可扩展枚举类型，多路分支方法是致命的，因为每当一个程序员扩展了这个枚举类型，它们不可能神奇地长出一个分支来。

对付这个问题的一种办法是，为这个类型安全枚举类提供所有“描述该类常量的行为”必要的方法。对该类的客户没有用的方法应该被声明为protected，以便对客户隐藏起来，而允许子类改写它们。如果这样的方法没有合理的默认实现，那么它应该被声明为abstract，同时也是protected的。

对于可扩展的类型安全枚举类来说，改写equals和hashCode方法，使它们成为final并且调用Object相应的方法，这是个好主意。这样可以确保子类不会意外地改写这些方法，从而可以保证该枚举类型所有相等的对象也一定是相同的，即当且仅当 $a==b$ 时 $a.equals(b)$ 为true：

```
// Override-prevention methods
public final boolean equals(Object that) {
    return super.equals(that);
}

public final int hashCode() {
    return super.hashCode();
}
```

注意，可扩展的变形模式与可比较的变形模式是不兼容的，如果把它们组合起来，子类的元素之间的排序关系将是“子类被初始化的顺序”的一个函数关系，而子类被初始化的顺序随程序的不同而不同，而且每次运行也可能不同。

类型安全枚举模式的可扩展变形与可序列化变形是兼容的，但是两者组合的时候要非常小心，每个子类必须分配自己的序数，并且提供自己的readResolve方法。本质上讲，每个类自己负责序列化和反序列化其实例。为了具体地说明这一点，下面的Operation类版本已经被修改成可扩展的、可序列化的类型安全枚举类：

```
// Serializable, extensible typesafe enum
public abstract class Operation implements Serializable {
    private final transient String name;
    protected Operation(String name) { this.name = name; }

    public static Operation PLUS = new Operation("+") {
        protected double eval(double x, double y) { return x+y; }
    };
    public static Operation MINUS = new Operation("-") {
        protected double eval(double x, double y) { return x-y; }
    };
}
```

110

```

};

public static Operation TIMES = new Operation("*") {
    protected double eval(double x, double y) { return x*y; }
};

public static Operation DIVIDE = new Operation("/") {
    protected double eval(double x, double y) { return x/y; }
};

// Perform arithmetic operation represented by this constant
protected abstract double eval(double x, double y);

public String toString() { return this.name; }

// Prevent subclasses from overriding Object.equals
public final boolean equals(Object that) {
    return super.equals(that);
}

public final int hashCode() {
    return super.hashCode();
}

// The 4 declarations below are necessary for serialization
private static int nextOrdinal = 0;
private final int ordinal = nextOrdinal++;
private static final Operation[] VALUES =
    { PLUS, MINUS, TIMES, DIVIDE };
Object readResolve() throws ObjectStreamException {
    return VALUES[ordinal]; // Canonicalize
}
}
}

```

下面是Operation的一个子类，它增加了对数(logarithm)和指数(exponential)操作。

[11] 这个子类可以位于以上修订版本Operation类所在的包之外。它可以是公有的，本身也可以是可扩展的。多个独立编写的子类可以和平共处：

```

// Subclass of extensible, serializable typesafe enum
abstract class ExtendedOperation extends Operation {
    ExtendedOperation(String name) { super(name); }

    public static Operation LOG = new ExtendedOperation("log") {
        protected double eval(double x, double y) {
            return Math.log(y) / Math.log(x);
        }
    };
    public static Operation EXP = new ExtendedOperation("exp") {
        protected double eval(double x, double y) {
            return Math.pow(x, y);
        }
    };

    // The 4 declarations below are necessary for serialization
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private static final Operation[] VALUES = { LOG, EXP };
    Object readResolve() throws ObjectStreamException {
        return VALUES[ordinal]; // Canonicalize
    }
}

```

注意，上面这些类中的`readResolve`方法都是包级私有的，而不是私有的。这样做是必要的，因为`Operation`和`ExtendedOperation`的实例实际上都是匿名子类的实例，所以私有的`readResolve`方法将不会起作用（见第57条）。

与`int`模式相比，类型安全枚举模式缺点很少。可能惟一一个严重的缺点是，要把类型安全枚举常量聚集到一起比较困难。对于基于`int`的枚举类型，传统的做法是，选择枚举常量的值，使得每一个常量都是2的不同幂次方，因而相关的常量通过按位或（OR）就可以表示一个常量集合：

```
// Bit-flag variant of int enum pattern
public static final int SUIT_CLUBS    = 1;
public static final int SUIT_DIAMONDS = 2;
public static final int SUIT_HEARTS   = 4;
public static final int SUIT_SPADES   = 8;

public static final int SUIT_BLACK = SUIT_CLUBS | SUIT_SPADES;
```

[112]

用这种方式来表示一组枚举类型常量的集合是非常简练的，也非常快速。对于类型安全枚举常量的组合，你可以从Collections Framework中选择一个通用的集合实现，但是，这种做法既不简练，也不快速：

```
Set blackSuits = new HashSet();
blackSuits.add(Suit.CLUBS);
blackSuits.add(Suit.SPADES);
```

虽然这种做法可能不如`int`枚举常量的集合那样简练和快速，但是通过提供一个专用的`Set`实现，使它只接受一种类型的元素，并且在集合内部用位向量来表示，从而可以缩小上述两种方案之间的差距。这样的集合最好与它的元素类型在同一个包内实现，从而使得通过一个包级私有的域或者方法，就可以访问到内部每一个与类型安全枚举常量相关联的位值。提供一些公有的构造函数（它们以一个短的元素序列作为参数）是非常有意义的，从而使下面这样的习惯用法成为可能：

```
hand.discard(new SuitSet(Suit.CLUBS, Suit.SPADES));
```

与`int`枚举常量相比，类型安全枚举常量还有一个小的缺点是，它们不能用在`switch`语句中，因为它们不是整值常量（integral constant）。相反，你可以使用`if`语句来代替，如下所示：

```
if (suit == Suit.CLUBS) {
    ...
} else if (suit == Suit.DIAMONDS) {
    ...
} else if (suit == Suit.HEARTS) {
    ...
} else if (suit == Suit.SPADES) {
    ...
```

```
    } else {
        throw new NullPointerException("Null Suit"); // suit == null
    }
}
```

if语句可能执行起来没有switch语句那么好，但是两者之间的差异不会非常显著。而且，对于类型安全枚举常量，用到多路分支的地方应该非常少，因为它们服从于JVM的自动方法分发机制，就像Operation例子中的情形一样。
[113]

类型安全枚举类型的另一个小小的性能缺点是，装载枚举类和构造常量对象时，需要一定的时间和空间开销。除非是在资源很受限制的设备比如蜂窝电话和烤面包机上，否则，在实际中这个问题不会引起注意。

总之，类型安全枚举类型明显地优于int枚举类型，除非是一个枚举类型主要被用作一个集合元素，或者主要用在一个资源非常受限的环境下，否则类型安全枚举类型的缺点都不成问题。因此，在要求使用一个枚举类型的环境下，我们首先应该考虑类型安全枚举模式。与int枚举类型相比，使用类型安全枚举类型的API对于程序员来说要友好得多。在Java平台API中，类型安全枚举类型没有被大力使用的唯一原因在于，在编写大多数这些API的时候，这种类型安全枚举模式尚未被人知晓。最终需要重申的是，对于任何枚举类型的需求都应该相对较少，因为随着子类化技术的推广，这些类型的主要用途已经过时了（见第20条）。
[114]

第22条：用类和接口来代替函数指针

C语言支持函数指针 (*function pointer*)，允许一个程序把“调用一个特定函数的能力”存储起来，或者传递这样的能力。函数指针的通常用法是，函数的调用者传入一个指向另一函数的指针，以此来指定自己的行为，有时候这种做法也被称为回调 (*callback*)。例如，C语言标准库中的`qsort`函数要求一个指向`comparator`（比较器）函数的指针作为参数，它用这个函数来比较待排序的元素。比较器函数有两个参数，都是指向元素的指针。如果第一个参数所指的元素小于第二个参数所指的元素，则返回一个负整数；如果两个元素相等则返回零；如果第一个参数所指的元素大于第二个参数所指的元素，则返回一个正整数。通过传递不同的比较器函数，就可以获得各种不同的排列顺序。这正是 *Strategy*（策略）模式 [Gamma95, p.315] 的一个例子。比较器函数代表一种排序元素的策略。

Java程序设计语言省略了函数指针，因为对象引用可以被用于提供同样的功能。调用一个对象上的一个方法通常是执行该对象 (*that object*) 上的某个操作。然而，我们也可能定义这样一个对象，它的方法执行其他对象 (*other objects*)（这些对象被显式传递进来）上的操作。如果一个类仅仅导出这样的一个方法，那么它的实例实际上就等同于一个指向该方法的指针。这样的实例被称为函数对象 (*function object*)。例如，考虑下面的类：

```
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

这个类导出一个带两个字符串参数的方法，如果第一个字符串的长度比第二个短，则返回一个负整数；如果两个字符串长度相等，则返回零；如果第一个字符串比第二个长，则返回一个正整数。这个方法是一个比较器，它根据长度来排序字符串，而不是根据常用的字典顺序。指向`StringLengthComparator`对象的引用可以被当做一个指向该比较器的“函数指针”，对于任意的字符串对，它都可以被调用。换句话说，`StringLengthComparator`实例是一个用于字符串比较操作的具体策略 (*concrete strategy*)。

作为一个典型的具体策略类，`StringLengthComparator`类是无状态的 (*stateless*)；它没有域，所以，这个类的所有实例在功能上都是相互等价的。因此，它作为一个 singleton 是非常合适的，从而可以节省不必要的对象创建开销（见第4条和第2条）：

```
class StringLengthComparator {
    private StringLengthComparator() {}

    public static final StringLengthComparator
        INSTANCE = new StringLengthComparator();
```

```

public int compare(String s1, String s2) {
    return s1.length() - s2.length();
}
}

```

为了把StringLengthComparator实例传递给一个方法，我们需要合适的参数类型。直接使用StringLengthComparator并不好，因为客户将无法传递任何其他的比较策略。相反，我们需要定义一个Comparator接口，并且对StringLengthComparator进行修改、让它实现这个接口。换句话说，我们在设计具体策略类的时候，还需要定义一个策略接口（*strategy interface*），如下所示：

```

// Strategy interface
public interface Comparator {
    public int compare(Object o1, Object o2);
}

```

Comparator接口的这个定义碰巧也出现在java.util包中，但是这并不神奇；你自己也完全可以定义它。所以，它也可以作为除字符串对象之外的其他对象的比较器，因为它的compare方法的参数类型为Object，而不是String。因此，前面给出的StringLengthComparator类必须稍做修改才能实现Comparator接口：在调用length方法之前，Object参数必须被转换为String。

具体的策略类往往使用匿名类声明（见第18条）。下面的语句根据长度对一个字符串数组进行排序：

```

Arrays.sort(stringArray, new Comparator() {
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.length() - s2.length();
    }
});

```

116

因为策略接口被用做所有具体策略实例的类型，所以我们并不需要为了导出一个具体策略，而把具体策略类做成公有的。相反，“宿主类（host class）”可以导出一个公有的静态域（或者静态工厂方法），其类型为策略接口，并且具体策略类可以是宿主类的一个私有嵌套类。下面的例子使用一个静态成员类，而不是匿名类，以便允许具体策略类实现第二个接口Serializable：

```

// Exporting a concrete strategy
class Host {
    ... // Bulk of class omitted

    private static class StrLenCmp
        implements Comparator, Serializable {
        public int compare(Object o1, Object o2) {
            String s1 = (String)o1;

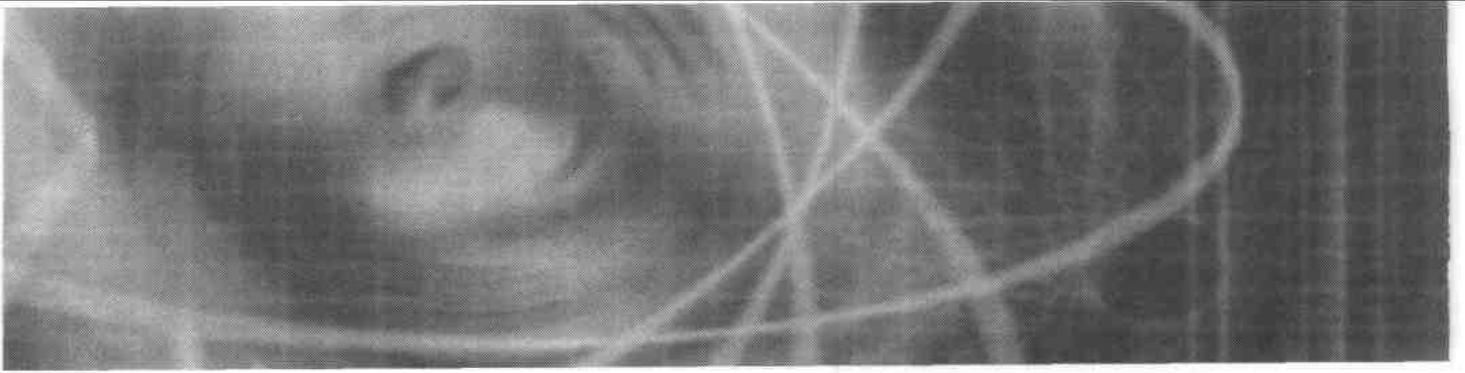
```

```
        String s2 = (String)o2;
        return s1.length() - s2.length();
    }
}

// Returned comparator is serializable
public static final Comparator
STRING_LENGTH_COMPARATOR = new StrLenCmp();
}
```

String类使用这种模式，通过它的CASE_INSENSITIVE_ORDER域，导出一个大小写不敏感的字符串比较器。

简而言之，C语言中函数指针的主要用途是实现Strategy（策略）模式。为了在Java程序设计语言中实现这种模式，声明一个接口来表示该策略，并且为每个具体策略声明一个实现了该接口的类。如果一个具体策略只被使用一次，那么通常使用匿名类来声明和实例化这个具体策略类。如果一个具体策略需要被导出去以便重复使用，那么它的类通常是一个私有的静态成员类，并且通过一个公有静态final域被导出，其类型为该策略接口。



第6章

方法

本章讨论方法设计的几个方面：如何处理参数和返回值，如何设计方法原型，如何为方法编写文档。本章中大多数材料既适用于构造函数，也适用于普通方法。与第5章一样，本章的焦点也集中在可用性、健壮性和灵活性上。

第23条：检查参数的有效性

绝大多数方法和构造函数都会对于传递给它们的参数值有某些限制。例如，索引值必须是非负数，对象引用不能为null，等等，这都是很常见的。你应该在文档中清楚地指明这些限制，并且在方法体的起始处对参数进行检查，以强迫施加这些限制。这是一般原则的具体情形，你应该在错误发生时尽可能快地检查到这些错误。如果不能做到这一点，则错误被检测到的可能性就很小，即使检测到错误了，也更加难以断定错误的根源。

如果一个无效的参数值被传递给一个方法，而这个方法在执行之前首先对参数进行了检查，则它很快就会失败，并且清楚地以一个适当的异常(exception)指明错误的原因。如果这个方法没有进行参数检查，那么有可能会发生几种情形。该方法会在处理过程中失败，并且产生一个令人迷惑的异常。更差的情况是，该方法会正常返回，但是计算出的结果是错误的。最糟糕的是，该方法可能会正常返回，但是它使得对象处于一种被破坏的状态，将来在某个不确定的时候，在某个不相关的点上引发出错误来。

对于公有的方法，使用Javadoc @throws标签(tag)可以使文档中记录下“一旦针对参数值的限制被违反之后将会被抛出的异常”(见第44条)。典型情况下，这样的异常为IllegalArgumentExceptio、IndexOutOfBoundsException或NullPointerException(见第42条)。如果你在文档中记录了方法参数上的限制，并且记录了一旦违反了这些限制将要抛出的异常，那么，施加这些限制是非常简单的事情。下面是一个典型的例子：

```

/**
 * Returns a BigInteger whose value is (this mod m). This method
 * differs from the remainder method in that it always returns a
 * nonnegative BigInteger.
 *
 * @param m the modulus, which must be positive.
 * @return this mod m.
 * @throws ArithmeticException if m <= 0.
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus not positive");
    ... // Do the computation
}

```

对于一个未被导出的方法，作为包的编写者，你可以控制这个方法将在哪些情形下被调用，所以你可以、而且也应该确保只有有效的参数值才会被传递进来。因此，非公有的方法通常应该使用*assertions*（断言）来检查它们的参数，而不使用正常的检查语句。如果你使用的开发平台版本（1.4或更高）支持断言，那么你应该使用*assert*结构；否则的话，你应该使用一种临时的断言机制。

对于有些参数，方法本身并没有用到它们，而是将它们存储起来供以后使用，检查这些参数的有效性尤为重要。例如，考虑第16条中的静态工厂方法*intArrayAsList*，它的参数为一个int数组，它返回该数组的List视图。如果这个方法的客户传递一个null，那么该方法将会抛出一个*NullPointerException*，因为该方法包含一个显式的条件检查。如果这个条件检查被省略的话，那么它就会返回一个指向新建的List实例的引用，只要客户企图使用这个引用，就会抛出一个*NullPointerException*。不幸的是，到了那个时候，要想找到List实例的来源可能就非常困难了，从而使得调试工作很是艰难和复杂。

前面提到，有些参数被方法保存起来供以后使用，构造函数正是代表了这种原则的一种具体情形。检查构造函数的参数的有效性是非常重要的，这样可以避免构造出来的对象违反这个类的约束条件。

120

“在一个方法执行它的计算任务之前，应该检查它的参数”，这条规则也有例外。一个很重要的例外是，在有些情况下，有效性检查工作非常昂贵，或者根本是不切实际的，并且在计算过程中有效性检查工作也被隐含着完成了。例如，考虑一个排序对象列表的方法，比如*Collections.sort(List)*。列表中的所有对象必须是可相互比较的。在排序列表的过程中，列表中的每个对象将与其他某个对象进行比较操作。如果这些对象不是可相互比较的，则必定有一个比较操作会抛出*ClassCastException*，这正是这个排序方法所应该做的事情。因此，提前检查列表中的元素是否是可相互比较的，这并没有多大意义，然而，请注意，不加选择地使用这项技术将会导致失去失败原子性（failure atomicity）（见第46条）。

偶尔情况下，某些计算会针对特定参数隐式地执行有效性检查，但是如果检查不成功的话，却抛出错误的异常。也就是说，由于无效参数而导致这个计算过程抛出的异常，与你在文档中标明这个方法将抛出的异常并不匹配。在这样的情况下，你应该使用第43条中讲述的异常转译（exception translation）技术，将计算过程中抛出的异常转换为正确的异常。

不要从本条目介绍的内容中得出结论“任何对参数的限制都是一件好事”。相反，你在设计方法时，应该使它们尽可能通用，并切合实际的需要。假设一个方法对于它能接受的所有参数值都能够完成合理的工作，那么，你对于参数的限制应该是越少越好。然而，通常情况下，有些限制对于被实现的抽象来说是很本质的。

总而言之，每当你编写一个方法或者构造函数的时候，应该考虑对于它的参数有哪些限制。你应该把这些限制写到文档中，并且在这个方法体的起始处，通过显式的检查来实施这些限制。养成这样的习惯是非常重要的；有效性检查所需要的适量工作从第一次合法性检查失败

121 中就可以连本带利得到补偿。

第24条：需要时使用保护性拷贝

Java程序设计语言用起来如此愉悦的一个原因是，它是一门安全的语言 (*safe language*)。这意味着，无需专门的手段，它对于缓冲区溢出、数组越界、非法指针以及其他内存破坏错误自动免疫，而这些错误却困扰着诸如C和C++这样的不安全语言。在一门安全语言中，我们在设计类的时候，可以确切地知道，无论系统的其他部分发生什么事情，这些类的约束都可以保持为真。但是，对于那些“把所有的内存当作一个巨大的数组来看待”的语言来说，这是不可能的。

即使在一门安全的语言中，你如果不采取一点措施，还是无法使自己与其他的类隔离开。假设类的客户会尽一切手段来破坏这个类的约束条件，在这样的前提下，你必须保护性地设计程序。实际上，只有当有人试图要破坏系统的安全性时，这种情形才会发生；更有可能的是，那些对你提供的API产生误解的程序员所编写的代码可能导致各种不可预期的行为，你的类必须能够妥善处理这些问题。无论是哪种情况，编写出一些“面对客户的不良行为时仍能保持健壮性的类”是非常值得投入时间去做的。

在对象本身没有提供帮助的情况下，另一个类要想修改这个对象的内部状态是不可能的；尽管如此，令人惊奇的是，无意识地提供这种帮助却又非常容易。例如，考虑下面的类，它声称可以表达一段不可改变的时间周期：

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start the beginning of the period.
     * @param end the end of the period; must not precede start.
     * @throws IllegalArgumentException if start is after end.
     * @throws NullPointerException if start or end is null.
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + " after "
                                           + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }

    public Date end() {
        return end;
    }
}
```

```

    ... // Remainder omitted
}

```

乍一看，这个类看上去是非可变的，并且强加了约束条件：周期的起始时间（start）不能在结束时间（end）之后。然而，只要意识到Date类本身是可变的，就可以知道这个约束条件很容易被违反：

```

// Attack the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78); // Modifies internals of p!

```

为了保护Period实例的内部信息避免受到这种攻击，对于构造函数的每个可变参数进行**保护性拷贝（defensive copy）**是必要的，并且使用拷贝之后的对象作为Period实例的组件，而不使用原始的对象。代码改写如下：

```

// Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(start + " after " + end);
}

```

用了新的构造函数之后，前面的攻击对于Period实例不再有效。注意，**保护性拷贝动作是在检查参数的有效性（见第23条）之前进行的，并且有效性检查是针对拷贝之后的对象，而不是原始的对象。**虽然这样做看起来有点不太自然，但这是必要的。这样做可以避免在“脆弱性窗口”中另外一个线程会改变原始的参数对象，这里的脆弱性窗口是指从参数检查开始，一直到参数对象被拷贝之间的一段时间窗。

同时也请注意，我们不要使用Date的clone方法来进行保护性拷贝。因为Date是非final的，clone方法并不保证返回的对象一定是java.util.Date类；它有可能返回另一个专门为了邪恶目的而设计的不可信子类的实例。例如，这样的子类可以在每个实例被创建的时候，把指向该实例的引用记录到一个私有的静态列表中，并且允许攻击者访问这个列表，这将使得攻击者可以自由地控制所有的实例。为了阻止这种攻击，对于“参数类型可以被不可信方子类化”的情形，请不要使用clone方法进行参数的保护性拷贝。
[123]

虽然替换了构造函数就可以成功地避免前述的攻击，但是要改变一个Period实例仍然是有可能的，因为它的两个访问方法（指start()和end()方法）提供了对其可变内部成员的访问能力：

```

// Second attack on the internals of a Period instance
Date start = new Date();

```

```
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78); // Modifies internals of p!
```

为了防御这第二种攻击，只需修改这两个访问方法，使它返回可变内部域的保护性拷贝即可：

```
// Repaired accessors - make defensive copies of internal fields
public Date start() {
    return (Date) start.clone();
}

public Date end() {
    return (Date) end.clone();
}
```

采用了新的构造函数和新的访问方法之后，`Period`成为真正的非可变类。不管一个程序员是多么不怀好意，或者多么不合格，他都没有简单的办法可以违反“一个周期的起始时间不允许落后于结束时间”这一基本约束条件。确实是这样的，因为除了`Period`类本身之外，其他任何一个类都无法访问到`Period`实例中的两个可变域。这些域被真正封装在对象的内部。

注意，新的访问方法与新的构造函数不同，它们在做保护性拷贝的时候使用了`clone`方法。这是可以接受的（虽然不要求这样做），因为我们可以肯定地知道，`Period`内部的`Date`对象的类是`java.util.Date`，而不可能是其他某个潜在的不可信子类。

对于参数的保护性拷贝并不仅仅在于非可变类。每当你编写一个方法或者构造函数时，如果它要接受客户提供的对象，允许它进入到内部数据结构中，则有必要考虑一下，客户提供的对象是否有可能是可变的。如果是的话，则考虑你的类是否能够容忍“这个对象在进入到内部数据结构中之后还有可能发生变化”。如果答案是否定的，那么你必须对对象进行保护性拷贝，并且让拷贝之后的对象而不是原始对象进入到内部数据结构中。例如，如果你正在考虑使用一个由客户提供的对象引用作为内部`Set`实例的一个元素，或者作为内部`Map`实例的一个键(`key`)，那么你应该意识到，如果这个对象在插入之后再被修改的话，那么`Set`和`Map`的约束规则就会被破坏。

124

在内部组件被返回给客户之前，对它们进行保护性拷贝也是同样的道理。不管这个类是否为非可变的，在把一个指向内部可变组件的引用返回给客户之前，你也应该认真考虑两次。解决的方案是，你应该返回一个保护性拷贝。同样地，记住非零长度的数组总是可变的，这是非常重要的。因此，你在把内部数组返回给客户之前，应该总是要进行保护性拷贝。另外一种解决方案是，给用户返回该数组的一个非可变视图。这两种技术在第12条中都已经演示过了。

可以肯定地说，所有这些讨论的真正启示是，只要有可能，你都应该使用非可变的对象作

为你的对象内部的组件，这样你就不必再关心保护性拷贝（见第13条）问题。在我们的 `Period` 例子的情形中，值得指出的是，有经验的程序员通常会使用 `Date.getTime()` 返回的 `long` 原语类型作为内部的时间表示，而不会使用 `Date` 对象引用。他们之所以这样做，主要因为 `Date` 是可变的。

在把可变参数融入到一个对象内部之前进行保护性拷贝并不总是合适的。对于有些方法和构造函数，它们在被调用的时候，其参数所引用的对象有一个显式的交接（*handoff*）过程。当客户调用这样的方法的时候，它承诺以后不再直接修改该对象。如果一个方法或者构造函数期望接管一个由客户提供的可变对象，那么它必须在文档中明确地指明这一点。

如果一个类包含的方法或者构造函数表明它需要接管一个对象的控制权，那么它就无法抵御恶意的客户。只有当一个类和它的客户之间有双向的信任关系，或者破坏一个类的约束条件不会伤害到除了客户之外的其他对象时，这样的类才是可以接受的。后一种情形的一个例子是包装类模式（*wrapper class pattern*）（见第14条）。根据包装类的本质特征，客户只需直接访问被包装的对象，就可以破坏包装类的约束，但是，这样做往往只会伤害到客户自己。
125

第25条：谨慎设计方法的原型

本条目是若干API设计技巧的总结，它们都达不到拥有独立条目的程度。所以把它们放在一起。这些设计提示将有助于使你的API更易于学习和使用，并且少犯错误。

谨慎选择方法的名字。方法的名字应该总是遵循标准的命名习惯（见第38条）。你的首要目标应该是选择易于理解的，并且与同一个包中其他名字风格一致的名字。你的第二个目标应该是选择与大众认可的名字（如果存在的话）相一致的名字。如果还有疑问的话，请看一看Java库的API作为参考。尽管Java库的API中也有大量不一致的地方，考虑到Java库的规模和范围，这是不可避免的，但它还是得到了广泛认可。*Patrick Chan的《The Java Developers Almanac》[Chan00]*^②是一个极为宝贵的资源，它包含了Java平台库中每一个方法的声明，有按字母顺序编排的索引。例如，如果你无法确定到底将一个方法命名为remove还是delete，则通过快速查看这本书的索引，你就会知道显然应该选择remove。那里有数百个以remove打头的方法名字，而只有少数的方法是以delete打头的。

不要过于追求提供便利的方法。每一个方法都应该提供其应具备的功能点。方法太多会使一个类难以学习、使用、文档化、测试和维护。对于接口而言，这无疑是正确的，接口的方法太多会使接口实现者和接口用户的生活变得复杂化。对于一个类型所支持的每一个动作，都提供一个功能完全的方法。只有当一个操作被用得非常频繁的时候，才考虑为它提供一个快捷方法（shorthand）。如果不能确定的话，还是不考虑为好。

避免长长的参数列表。通常，三个参数应该被看做实践中的最大值，而且参数越少越好。大多数程序员不能够记住更长的参数列表。如果你编写的方法大多数都超过了这个限制，那么你的API不会有用，除非用户不停地参考它的文档。类型相同的长参数序列尤其有害。API的用户不仅不能够记住参数的顺序，而且，当他们弄错了参数顺序的时候，他们的程序仍然可以编译和运行。只不过这些程序不会按照编写者的意图进行工作。

126

有两项技术可以缩短太长的参数列表。第一项技术是把一个方法分解成多个方法，每一个方法只要求这些参数的一个子集。如果做得不小心的话，这很容易会导致太多的方法。但是通过增加它们的正交性（orthogonality），可以减少（reduce）方法的数目。例如，考虑java.util.List接口。它并没有提供“在一个子列表（sublist）中查找一个元素的第一个索引和最后一个索引”这样的方法，这两个方法都要求三个参数。相反，它提供了subList方法，这个方法带两个参数，它返回子列表的一个视图（view）。这个方法与indexOf或者lastIndexOf方法结合起来，可以获得期望的功能，而这两个方法只有一个参数。而且，

^② 此书已由机械工业出版社出版。——译注

`subList`方法与其他任何“针对`List`实例进行操作”的方法结合起来，就可以在子列表上执行任意的计算。这样得到的API有一个很高的“功能 – 重量”(power-to-weight)比。

第二项缩短长参数列表的技术是创建辅助类(*helper class*)，用来保存参数的聚集(*aggregate*)。这些辅助类往往是静态成员类(见第18条)。如果一个频繁出现的参数序列可以被看做代表了某个独特的实体，则建议使用这项技术。例如，假设你正在编写一个代表纸牌游戏的类，你会发现，经常要传递一个两参数组合来代表一张纸牌的点数和花色。如果你增加一个辅助类来代表一张纸牌，并且把每个两参数组合都换成这个辅助类作为单个参数，那么这个纸牌游戏类的API以及它的内部表示都可能会得到改进。

对于参数类型，优先使用接口而不是类。无论什么时候，只要存在可用来定义参数的适当接口，就优先使用这个接口，而不是实现该接口的类。例如，没有理由在编写一个方法时使用`Hashtable`作为输入，相反，应该使用`Map`。这使得你可以传入一个`Hashtable`、`HashMap`、`TreeMap`、`TreeMap`的子映射表(`submap`)，或者任何有待于将来编写的`Map`实现。如果你使用一个类而不是一个接口，则限制了客户只能传入一个特定的实现，如果碰巧输入的数据是以其他形式存在的话，则会导致不必要的、可能非常昂贵的拷贝操作。

谨慎地使用函数对象(见第22条)。有一些语言，特别是Smalltalk和各种Lisp语言版本，鼓励这样一种程序设计风格：尽量用对象来表示“被应用到其他对象中的函数”。对这些语言有丰富经验的程序员可能会在Java程序设计语言中也采用类似的风格，但是这样做并不非常合适。创建函数对象最容易的办法莫过于使用匿名类(见第18条)，但是那样会带来语法上的混乱，并且与内联的控制结构相比，这样也会导致功能和性能上的局限性。而且，在这样的程序设计风格下，你要频繁地创建函数对象，并且将它们从一个方法传递到另一个方法，这种程序设计风格并非主流，所以，如果你使用这种风格编写代码，那么其他的程序员在理解你的代码时将会有一段非常困难的时期。这并不意味着函数对象没有合法的用途；相反，对于许多功能强大的设计模式，比如`Strategy`[Gamma95, p.315]和`Visitor`[Gamma95, p.331]模式，函数对象是必需的。然而，只有当存在充分的理由时，才应该使用函数对象。

第26条：谨慎地使用重载

下面是一个意图良好的集合分类器，根据一个集合（collection）是Set、List，或是其他的集合类型，对它进行分类：

```
// Broken - incorrect use of overloading!
public class CollectionClassifier {
    public static String classify(Set s) {
        return "Set";
    }

    public static String classify(List l) {
        return "List";
    }

    public static String classify(Collection c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection[] tests = new Collection[] {
            new HashSet(),           // A Set
            new ArrayList(),         // A List
            new HashMap().values()  // Neither Set nor List
        };

        for (int i = 0; i < tests.length; i++)
            System.out.println(classify(tests[i]));
    }
}
```

你可能期望这个程序会打印出“Set”，紧接着“List”，以及“Unknown Collection”，但是，它并没有这样做；相反，它打印“Unknown Collection”三次。为什么会这样呢？因为classify方法被重载（overloaded）了，而到底调用哪个重载（overloading）方法是在编译时刻作出决定的。对于for循环中的全部三次迭代，参数的编译时类型都是相同的：Collection。每次迭代的运行时类型是不同的，但这并不影响对重载方法的选择。因为该参数的编译时类型为Collection，所以，惟一合适的重载方法是第三个：classify(Collection)，在循环的每次迭代中，都会调用这个重载方法。

这个程序的行为是违反直觉的，因为对于重载方法（overloaded method）的选择是静态的，而对于被改写的方法（overridden method）的选择是动态的。对于被改写的方法，选择正确的版本是在运行时刻进行的，选择的依据是被调用方法所在对象的运行时类型。这里重新说明一下，当一个子类包含的方法声明与其祖先类中的一个方法声明具有同样的原型时，我们称一个方法被改写了。如果一个实例方法在一个子类中被改写了，并且这个方法是在该子类的实例上被调用的，那么子类中被改写的方法（overriding method）将会执行，而不管该子类实例的编译时类型到底是什么。为了更直观地说明这一点，考虑下面的小程序：

```

class A {
    String name() { return "A"; }
}

class B extends A {
    String name() { return "B"; }
}

class C extends A {
    String name() { return "C"; }
}

public class Overriding {
    public static void main(String[] args) {
        A[] tests = new A[] { new A(), new B(), new C() };

        for (int i = 0; i < tests.length; i++)
            System.out.print(tests[i].name());
    }
}

```

name方法是在类A中被声明的，但是在类B和C中被改写了。正如你所预期的那样，这个程序打印出“ABC”，尽管在循环的每次迭代中，实例的编译时类型是A。当调用一个被改写的方法时，对象的编译时类型不会影响到哪个方法将被执行；“最为特殊化的（most specific）”那个改写版本总是会得到执行。相反，在重载的情形中，对象的运行时类型对于“哪个重载版本将被执行”没有任何影响；选择工作是在编译时刻进行的，完全基于参数的编译时类型。

[129] 在CollectionClassifier例子中，该程序的意图是：期望编译器根据参数的运行时类型自动将调用分发给适当的重载方法，以此来识别出参数的类型，就好像“ABC”例子中的name方法所做的那样。方法重载机制并没有提供这样的功能。这个问题的修正方案是，用一个方法来替换这三个重载的classify方法，并且在这个方法中做一个显式的instanceof测试：

```

public static String classify(Collection c) {
    return (c instanceof Set ? "Set" :
           (c instanceof List ? "List" : "Unknown Collection"));
}

```

因为改写机制是规范，而重载机制是例外，所以，改写机制满足了人们对于方法调用行为的期望。正如CollectionClassifier例子所示，重载机制很容易混淆这些期望。如果编写出来的代码对于普通程序员来说，其行为并非显而易见，那么这就是很糟糕的实践工作。对于API来说尤其如此。如果一个API的典型用户根本不知道“对于一组给定的参数，哪个重载方法将会被调用”，那么，使用这样的API极有可能会导致错误。这样的错误要等到运行时刻发生了怪异的行为之后才会显现出来，许多程序员无法诊断出这样的错误。因此，你应该避免方法重载机制的混淆用法。

到底是什么造成了重载机制的混淆用法？这仍是争论的话题。一个安全而保守的策略是，永远不要导出两个具有相同参数数目的重载方法。如果你遵守这条规则，程序员永远也不会陷入到“对于任何一组参数，到底哪个重载方法是适用的”这样的疑问中。这项限制规则并不麻烦，因为你总是可以给方法起不同的名字，而不使用重载机制。

例如，考虑ObjectOutputStream类。对于每一个原语类型，以及几种引用类型，它的write方法都有一个变形。这些变形方法并不是重载write方法，而是具有诸如writeBoolean(boolean)、writeInt(int)和writeLong(long)这样的原型。与重载方案相比较，这种命名模式带来的一个好处是，我们有可能提供对应名字的读方法，比如readBoolean()、readInt()和readLong()。实际上，ObjectInputStream类正是提供了采用这些名字的读方法。

对于构造函数，你没有选择使用不同名字的机会；一个类的多个构造函数总是重载的。在某些情况下，你可以选择导出静态工厂，而不是构造函数（见第1条），但是这样做并不总是切合实际的。好的一面是，对于构造函数，你不用担心改写和重载的相互影响，因为构造函数不可能被改写。因为你有可能会导出多个具有相同参数数目的构造函数，所以有必要了解一下什么时候这样做是安全的。

130

如果对于任何一组给定的实际参数，“哪 一个重载方法将会应用于这一组参数”是非常清楚的，那么，导出多个具有相同参数数目的重载方法不可能混淆程序员。如果对于每一对重载方法，至少有一个对应的形参具有“根本不同（radically different）”的类型，那么这就属于这样的情形。所谓根本不同的两种类型，是指要把一种类型的实例转换为另一种类型显然是不可能的。在这种情况下，哪个重载方法应用于一组给定的实际参数完全是由参数的运行时类型来决定的，不可能受到其编译时类型的影响，所以主要的混淆源头就消失了。

例如，ArrayList有一个构造函数带一个int参数，另一个构造函数带一个Collection参数。难以想像在什么情况下，这两个构造函数被调用时会产生混淆，因为原语类型和引用类型是根本不同的。类似地，BigInteger有一个构造函数带一个byte数组，另一个构造函数带一个String参数，这也不会引起混淆。数组类型和除了Object之外的类是根本不同的，而且，数组类型与除了Serializable和Cloneable之外的接口也是根本不同的。最后，1.4发行版本中的Throwable有一个构造函数带一个String参数，另一个构造函数带一个Throwable参数。String类和Throwable类是不相关的（unrelated），所谓两个类是不相关的，是指它们中任何一个都不是另一个的后代。一个对象不可能是两个不相关的类的实例，所以不相关的类是根本不同的。

还有其他一些“类型对”的例子也是相互不能转换的[JLS, 5.1.7]，但是，一旦超出了这

些简单的情形，普通程序员要想搞清楚“哪个重载方法应用于一组实际的参数”就会非常困难。确定选择哪个重载方法的规范非常复杂，很少有程序员能够理解它的所有微妙之处[JLS, 15.12.1-3]。

偶尔情况下，当你更新一个已有的类以实现新的接口的时候，你可能会被迫违反上面的指导原则。例如，Java平台库中许多值类型（value type）在引入Comparable接口之前，都有自身类型的（self-typed）compareTo方法。下面是String原来的自身类型的compareTo方法的声明：

[13] `public int compareTo(String s);`

随着引入了Comparable接口之后，所有这些类都被更新了，以实现这个新的接口；这涉及到增加一个更为一般化的compareTo方法，声明如下：

`public int compareTo(Object o);`

尽管这样的重载很显然违反了上面的指导原则，但是只要当这两个重载方法在同样的参数上被调用时，它们执行相同的功能，则重载就不会带来危害。程序员可能并不知道哪个重载函数会被调用，但只要这两个方法返回相同的结果就行。确保这种行为的标准做法是，让更一般化的重载方法把调用转发给更特殊化的重载方法：

```
public int compareTo(Object o) {
    return compareTo((String) o);
}
```

一种类似的做法有时被用于equals方法：

```
public boolean equals(Object o) {
    return o instanceof String && equals((String) o);
}
```

这种做法是无害的，如果参数的编译时类型与更特殊的重载方法的参数类型匹配的话，可能还会获得性能上的稍微提高。但是，实际上这种改进并不值得去做（见第37条）。

虽然Java平台库很大程度上遵循了本条目中的建议，但是也有诸多违背之处。例如，String类导出两个重载的静态工厂方法：valueOf(char[])和valueOf(Object)，当这两个方法被传递了同样的对象引用的时候，它们做的事情完全不同。没有正当的理由可以解释这一点，它应该被看作一种反常行为，有可能会带来潜在的真正的混淆。

总而言之，“你能够重载方法”并不意味着“你应该重载方法”。一般地，对于多个具有相同参数数目的方法来说，你应该尽量避免重载方法。在某些情况下，特别是涉及到构造函

数的时候，遵循这条建议也许是不可能的。在这种情况下，你至少应该避免这样的情形：同一组参数只需经过类型转换就可以被传递给不同的重载方法。如果这样的情形不能避免的话，例如因为你正在更新一个已有的类以实现一个新的接口，那么你应该保证：当传递同样的参数时，所有的重载方法行为一致。如果你不能做到这一点，程序员就不能有效地使用被重载的方法或者构造函数，他们就不能理解为什么它不能正常地工作。[132]

第27条：返回零长度的数组而不是null

像下面这样的方法并不少见：

```
private List cheesesInStock = ...;

/**
 * @return an array containing all of the cheeses in the shop,
 *         or null if no cheeses are available for purchase.
 */
public Cheese[] getCheeses() {
    if (cheesesInStock.size() == 0)
        return null;
    ...
}
```

当没有奶酪（cheese）可买时，上述代码把它当做一种特殊情况来对待，这是没有理由的。这样做也要求客户方必须有额外的代码来处理null返回值，例如：

```
Cheese[] cheeses = shop.getCheeses();
if (cheeses != null &&
    Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

而不是下面简单的代码：

```
if (Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

对于一个返回null而不是零长度数组的方法，几乎每次用到该方法的时候都需要这种曲折的处理方式。这样做很容易出错，因为编写客户程序的程序员可能会忘记写这种专门的代码来处理null返回值。这样的错误也许几年都不会被注意到，因为这样的方法通常返回一个或者多个对象。返回null而不是零长度的数组也会使返回数组的方法本身更加复杂，这一点虽然不是特别重要，但是也值得注意。

有时候有人会争辩说，null返回值比零长度数组更好，因为它避免了分配数组所需要的开销。这种观点是站不住脚的，原因有两点。第一，在这个层次上担心性能问题是不明智的，除非分析表明这个方法正是造成性能问题的真正源头（见第37条）。第二，对于不返回任何元素的调用，每次都返回同一个零长度数组是有可能的，因为零长度数组是非可变的，而非可变对象有可能被自由地共享（见第13条）。实际上，当你使用标准做法（standard idiom）把一些元素从一个集合转储到一个类型化的数组（typed array）中时，它正是这样做的：

```
private List cheesesInStock = ...;

private final static Cheese[] NULL_CHEESE_ARRAY = new Cheese[0];
```

```
/**  
 * @return an array containing all of the cheeses in the shop.  
 */  
public Cheese[] getCheeses() {  
    return (Cheese[]) cheesesInStock.toArray(NULL_CHEESE_ARRAY);  
}
```

在这种习惯用法中，一个零长度数组常量被传递给 toArray方法，以指明所期望的返回类型。正常情况下， toArray方法分配了返回的数组，但是，如果集合是空的，它将使用输入数组。Collection.toArray(Object[]) 的规范保证：如果输入数组足够大到能够容纳这个集合的话，它将返回这个输入数组。因此，这种做法永远也不会分配一个零长度数组，而是重用这个“类型特有的常量（type-specifier constant）”。

简而言之，没有理由从一个取数组值（array-valued）的方法中返回null，而不是返回一个零长度数组。这种习惯做法（指返回null）很有可能是从C程序设计语言中沿袭过来的，在C语言中，数组长度在被返回的时候，是与实际的数组分离的。并且，在C语言中，如果返回的数组长度为零的话，再分配一个数组是没有任何好处的。

第28条：为所有导出的API元素编写文档注释

如果要想使一个API真正可用，你必须为其编写文档。传统意义上的API文档是手工生成的，所以保持文档与代码的同步是一件很繁琐的事情。Java语言环境提供了一个被称为Javadoc的实用工具，从而使这项任务变得容易。这个工具可以根据源代码自动生成API文档，它利用了源代码中特殊格式的文档注释（*documentation comment*, 通常被写作*doc comment*）。Javadoc工具为API文档的编写提供了一条简单而有效的途径，目前它已经被广泛使用了。

如果你对文档注释的规范还不太熟悉，那么你应该学习这些规范。虽然这些规范还不是Java程序设计语言的一部分，但是它们已经构成了一个事实上的API，每个程序员都应该知道的API。这些规范的定义位于Javadoc工具的主页[Javadoc-b]。

为了正确地编写API文档，你必须在每一个被导出的类、接口、构造函数、方法和域声明之前增加一个文档注释，只有本条目末尾讨论的一种情形[⊖]例外。如果没有文档注释，那么Javadoc所能够做到的也就是重新生成该声明，作为受影响的API元素的惟一文档。使用一个缺少文档注释的API是非常痛苦的，也很容易出错。为了编写出可维护的代码，你也应该为那些没有被导出的类、接口、构造函数、方法和域编写文档注释。

每一个方法的文档注释应该简洁地描述出它和客户之间的约定。除了专门为继承而设计的类中的方法（见第15条）是例外情况外，这个约定应该说明了这个方法做了什么，而不是说明它是如何完成这项工作的。文档注释应该列举出这个方法所有的前提条件（*precondition*）和后置条件（*postcondition*），所谓前提条件是指为了使客户能够调用这个方法，而必须要满足的条件；所谓后置条件是指在调用成功完成之后，哪些条件必须要满足。典型情况下，前提条件是由@throws标签所隐含描述的（针对那些未被检查的异常）；每一个未被检查的异常都对应着一个被违背的前提条件。同样地，你也可以在一些受影响的参数的@param标记中指定前提条件。

除了前提条件和后置条件之外，每一个方法也应该在文档中描述它的副作用（*side effect*）。所谓副作用是指系统状态中一个可观察到的变化，它不是为了获得后置条件而要求的变化。例如，如果一个方法启动了一个后台线程，那么文档中应该说明这一点。最后，文档注释也应该描述一个类的线程安全性（*thread safety*），正如第52条所讨论的。

为了完整地描述一个方法的约定，方法的文档注释应该对每一个参数都有一个@param标签，以及一个@return标签（除非这个方法的返回类型为void），以及对于该方法抛出的每

[⊖] 指可以从接口或者超类中继承文档注释。——译注

一个异常，无论是被检查的还是未被检查的，都有一个@throws标签（见第44条）。按照惯例，跟在@param标签或者@return标签后面的文字应该是一个名词短语，描述了这个参数或者返回值所表示的值。跟在@throws标签之后的文字应该包含单词“if”（如果），紧接着是一个名词短语，它描述了这个异常将在什么样的条件下会被抛出来。偶尔情况下，也会用算术表达式来代替名词短语。下面这段摘自List接口的文档注释演示了所有这些习惯做法：

```
/*
 * Returns the element at the specified position in this list.
 *
 * @param index index of element to return; must be
 *              nonnegative and less than the size of this list.
 * @return the element at the specified position in this list.
 * @throws IndexOutOfBoundsException if the index is out of range
 *          (<tt>index < 0 || index >= this.size()</tt>).
 */
Object get(int index);
```

注意，这份文档注释中使用了HTML元字符和HTML标签。Javadoc工具会把文档注释翻译成HTML，文档注释中包含的任意HTML元素都会出现在结果HTML文档中。偶尔情况下，程序员可以把HTML表格嵌入到文档注释中，但是这并不多见。最通常使用的标签有：`<p>`，用来分割段落；`<code>`和`<tt>`，用于将代码分段；`<pre>`，用于更长的代码分段。

`<code>`和`<tt>`标签在很大程度上是等价的。`<code>`标签更为常用，根据HTML 4.01的规范，一般情况下它是优先使用的，因为`<tt>`是一个字体样式元素(*font style element*)（我们应该使用样式表(*style sheet*)，而不是字体样式元素[HTML401]）。有些程序员喜欢使用`<tt>`，是因为它更短并且不容易带来干扰。

不要忘记，为了产生HTML元字符，比如小于号(<)、大于号(>)以及“与”号(&)，转义序列是必要的。为了产生一个小于号，请使用转义序列“<”；为了产生一个大于号，请使用转义序列“>”；为了产生一个“与”号，请使用转义序列“&”。上面文档注释中的@throws标签演示了转义序列的用法。

137

最后，请注意这份文档注释中单词“this”的用法，按照习惯，当单词“this”被用于一个实例方法的文档注释中时，它总是指该方法被调用时所在的对象。

每一个文档注释的第一句话是该注释所属元素的概要描述(*summary description*)。概要描述必须独立地描述目标实体的功能。为了避免混淆，同一个类或者接口中，不应该存在两个成员或者构造函数具有同样的概要描述。特别要注意重载的情形，在这种情况下，往往很自然地在描述中使用同样的第一句话。

小心，在文档注释的第一句话内部不要包括句号。如果你包括了句号，则它会终止整个概

要描述。例如，一个以“`A college degree, such as B.S., M.S., or Ph.D.`”开头的文档注释，它的概要描述为“`A college degree, such as B.`”避免这种问题最容易的做法是，在概要描述中不要使用缩写和十进制小数。然而，在概要描述中使用句号也是可能的，你只需用句号的数字编码形式(*numeric encoding*)“`.`”来代替它。虽然这样做可以工作，但不会生成漂亮的源代码：

```
/**  
 * A college degree, such as B&#46;S&#46;; M&#46;S&#46;; or  
 * Ph&#46;D.  
 */  
public class Degree { ... }
```

“概要描述是文档注释中的第一个句子(*sentence*)”这种说法有点误导人，规范指出，概要描述很少是一个完整的句子。对于方法和构造函数，概要描述应该是一个动词短语，它描述了该方法所执行的动作。例如：

- `ArrayList(int initialCapacity)`—Constructs an empty list with the specified initial capacity. (用指定的初始容量构造一个空的列表)
- `Collection.size()`—Returns the number of elements in this collection. (返回该集合中元素的数目)

对于类、接口和域，概要描述应该是一个名词短语，它描述了该类或者接口的实例，或者域本身所代表的事物。例如：

- `TimerTask`—A task that can be scheduled for on-time or repeated execution by a Timer. (可以被一次调度的一项任务，或者被一个Timer重复执行的任务)
- `Math.PI`—The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter. (非常接近于pi(圆周长度与直径的比值)的double值)

本条目中讨论的文档注释惯例已经够用了，但是还有很多没有介绍。文档注释的编写风格有一些指导规则[Javadoc-a, Vermeulen00]。而且，还有一些工具可用来检查是否遵循了这些规则[Doclint]。

自从1.2.2发行版本以来，Javadoc已经有“自动重用”，或者“继承”方法注释的能力了。如果一个方法并没有文档注释，那么Javadoc将会搜索最为适用的文档注释，接口的文档注释优先于超类的文档注释。搜索算法的细节可以在《The Javadoc Manual》中找到。

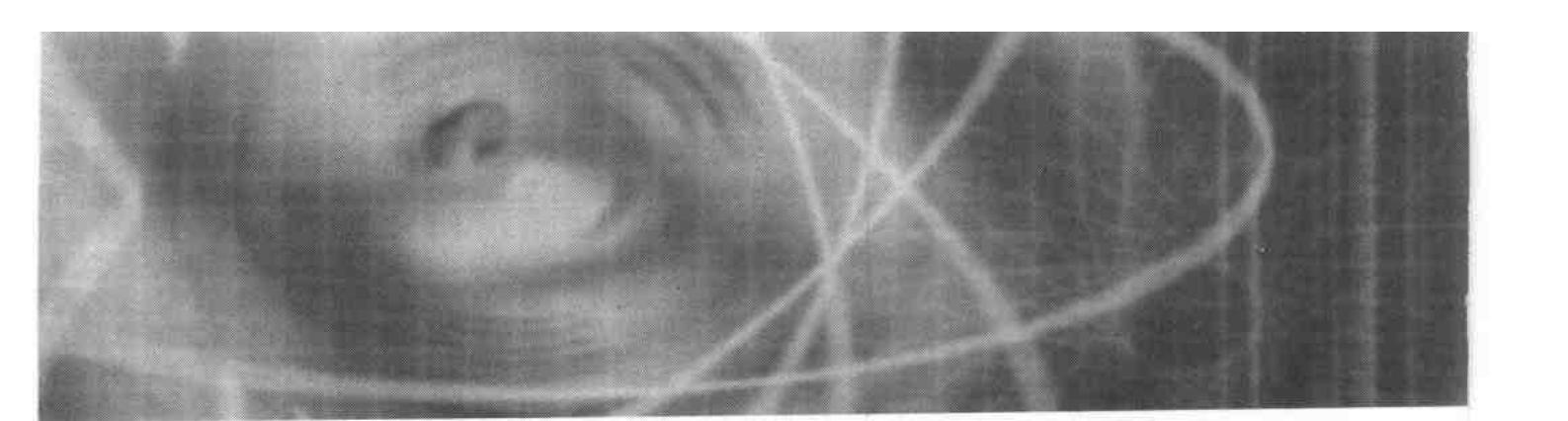
这意味着，一个类可以重用它所实现的接口的文档注释，而不需要拷贝这些注释。这项设

施有潜力可用来降低或者消除“维护多个几乎相同的文档注释”的负担，但是它也有一些局限性。文档注释的继承是一种“要么全有要么全无”的能力：继承的方法不可能以任何方式修改被继承的文档注释。一个方法要对“继承自接口的约定”进行针对具体需求的特定改造是很常见的，在这种情况下，这个方法确实需要自己专门的文档注释。

为了降低文档注释中出错的可能性，一种简单的办法是通过一个HTML有效性检查器(*HTML validity checker*)，运行由Javadoc产生的HTML文件。这可以检测出HTML标签的许多不正确用法，以及应该被转义的HTML元字符。Internet上有一些HTML有效性检查器可供下载并使用，例如`weblint`[[Weblint](#)]。

关于文档注释有一点需要特别注意。虽然为所有导出的API元素提供文档注释是必要的，但是这样做并不总是足够的。对于由多个相互关联的类组成的复杂API，通常有必要在提供文档注释的同时，也要提供一份额外的文档来描述该API的总体结构。如果这样的文档存在，那么相关的类或者包的文档注释应该包含一个对它的链接。

总而言之，要为API编写文档，文档注释是最好、最有效的途径。对于所有可导出的API元素来说，使用文档注释应该是强制性的。采用一种一致的风格来遵循标准的约定。记住，在文档注释内部出现任意HTML标签都是允许的，但是HTML元字符必须要经过转义。[139]



第7章

通用程序设计

本章主要讨论Java语言的具体细节，讨论了局部变量的处理、库的使用、各种数据类型的用法，以及两种不是由语言本身提供的设施（*reflection and native method*，映像机制和本地方法）的用法。最后讨论了优化和命名惯例。

第29条：将局部变量的作用域最小化

本条目与第12条（使类和成员的可访问能力最小化）本质上是类似的。将局部变量的作用域最小化，可以增加代码的可读性和可维护性，并降低出错的可能性。

C程序设计语言要求局部变量必须被声明在一个代码块的开始处，出于习惯，程序员们继续这样做。这是一个应该被打破的习惯。在此提醒，Java程序设计语言允许你在任何可以出现语句的地方声明变量。

使一个局部变量的作用域最小化，最有力的技术是在第一次使用它的地方声明。如果一个变量在使用之前就已经被声明了，那么这只会带来混乱——对于试图理解程序功能的读者来说，这种做法只会分散他们的注意力。到变量被使用的时候，读者可能已经记不起变量的类型或者初始值了。当随着程序的演化，这个变量不再有用时，如果变量的声明与变量的第一个使用点相距比较远的话，很容易忘记删除其声明。

过早地声明一个局部变量不仅会使它的作用域被扩展到太早的点上，同样也会被扩展到太晚的点上。局部变量的作用域从它被声明的点开始，一直到外围块（block）的结束处。如果一个变量是在“使用它的块”之外被声明的，那么当程序退出该块之后，该变量仍是可见的。

[141] 如果一个变量在它的目标使用区域之前或者之后被意外地使用的话，则后果将是灾难性的。

几乎每一个局部变量的声明都应该包含一个初始化表达式。如果你还没有足够的信息来对一个变量进行有意义的初始化，那么你应该推迟这个声明，直到可以初始化为止。这条规

则的一个例外情况与try-catch语句有关。如果一个变量被一个方法初始化，而这个方法可能会抛出一个被检查的异常，那么该变量必须在try块的内部被初始化。如果变量的值必须在try块的外部被使用到，那么它必须在try块之前被声明，但是在try块之前，它还不能被“有意义地初始化”。请参照第35条的例子。

在循环中经常要用到最小化变量作用域这一规则。for循环使你可以声明循环变量(*loop variable*)，它们的作用域被限定在正好需要的范围之内(这个范围包括循环体，以及循环体之前的初始化、测试、更新部分)。因此，如果在循环终止之后循环变量的内容不再被需要的话，则for循环优先于while循环。

例如，下面是一种对集合进行迭代的首选做法：

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
```

为了弄清楚为什么这个for循环比更为显然的while循环要好，请考虑下面的代码片断，它包含两个while循环，以及一个错误(bug)：

```
Iterator i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
...
Iterator i2 = c2.iterator();
while (i.hasNext()) {           // BUG!
    doSomethingElse(i2.next());
}
```

第二个循环包含一个“剪切 - 粘贴”错误：它本来是要初始化一个新的循环变量*i2*，但是却使用了老的循环变量*i*，不幸的是，这时*i*仍然还在有效范围之内。结果代码仍然可以通过编译，运行的时候也不会抛出异常，但是它所做的事情却是错误的。第二个循环并没有在*c2*上迭代，而是立即终止，造成*c2*是空集合的假象。因为这个程序的错误是悄然发生的，所以可能在很长时间内都不会被发现。

如果类似的“剪切 - 粘贴”错误发生在前面for循环的用法中，则结果代码根本不能通过编译。在第二个循环开始之前，第一个循环的循环变量已经不在它的作用域范围内了：

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
...
// Compile-time error - the symbol i cannot be resolved
for (Iterator i2 = c2.iterator(); i.hasNext(); ) {
```

```

    doSomething(i2.next());
}

```

而且，如果你使用for循环，那么犯这种“剪切~粘贴”错误的可能性会大大降低，因为没有明显的动机要在两个循环中使用不同的变量名。循环是完全独立的，所以重用循环变量名字不会有任何危害。实际上，这也是很流行的做法。

使用for循环比使用while循环还有另外一个优势，尽管这个优势没有前面的那么显著。使用for循环要少一行代码，有助于把方法装入到大小固定的编辑器窗口中，从而增强可读性。

下面是另外一种对列表进行迭代的循环做法，它也使局部变量的作用域最小化：

```

// High-performance idiom for iterating over random access lists
for (int i = 0, n = list.size(); i < n; i++) {
    doSomething(list.get(i));
}

```

对于随机访问的List实现，比如ArrayList和Vector，这种用法是非常有用的，因为它可能比前面我们推荐的方法运行得更快。关于这种用法，很重要并且值得注意的一点是，它有两个循环变量：i和n，两者具有完全相同的作用域。第二个变量的使用是提高性能的关键。如果没有这个变量，那么这个循环的每次迭代都必须调用size方法，从而削弱这种用法的性能优势。当你确定列表对象真的提供了随机访问能力时，这种用法是可以接受的；否则它将会导致平方级的性能开销。

对于其他的循环任务，也存在类似的做法，例如：

```

for (int i = 0, n = expensiveComputation(); i < n; i++) {
    doSomething(i);
}

```

同样地，这个用法也使用了两个循环变量，并用第二个变量n可以避免在每一次迭代时执行冗余计算的代价。通常，如果循环测试中涉及到一个方法调用，并且可以保证，每次迭代中这个方法调用都会返回同样的结果，那么你应该使用这种用法。

[143] 最后一项“最小化局部变量的作用域”的技术是使方法小而集中。如果你把两个操作(activity)组合到同一个方法中，那么，与一个操作相关的局部变量有可能会出现在执行另一个操作的代码范围之中。为了防止这种情况发生，只需简单地把这个方法分成两个：每个操作一个方法。

第30条：了解和使用库

假设你希望产生位于0和某个上界之间的随机整数。面对如此常见的任务，许多程序员会编写出如下所示的方法：

```
static Random rnd = new Random();

// Common but flawed!
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这种方法不错，但是并不完美——它有三个缺点。第一个缺点是，如果n是一个比较小的2的乘方，那么，经过一段相当短的周期之后，它产生的随机数序列将会重复。第二个缺点是，如果n不是2的乘方，那么平均起来，有些数比其他的数出现得更为频繁。如果n比较大，则这个缺点会更加显著。这可以通过下面的程序直观地体现出来，它会产生一万个指定范围内的随机数，然后打印出有多少个数落在随机数取值范围的前半部分：

```
public static void main(String[] args) {
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i = 0; i < 1000000; i++)
        if (random(n) < n/2)
            low++;
    System.out.println(low);
}
```

如果random方法工作正常的话，这个程序打印出来的数接近于一百万的一半，但是如果你运行这个程序的话，你就会发现它打印出来的数接近666 666。由random方法产生的数有2/3落在随机数取值范围的前半部分。

random方法的第三个缺点是，在极偶尔的情况下，它会灾难性地失败，返回一个落在指定范围之外的数。这是因为这个方法试图利用Math.abs，将rnd.nextInt()返回的值映射为一个非负整数。如果nextInt()返回Integer.MIN_VALUE，那么Math.abs也会返回Integer.MIN_VALUE；然后，假设n不是2的乘方，那么取模操作符（%）将返回一个负数。这几乎肯定会使你的程序失败，而且这种失败很难重现。145

为了编写一个能修正这三个缺点的random版本，你有必要了解关于线性同余伪随机数发生器、数论和2的求补算法的知识。很幸运，你并不需要自己来做这些工作——已经有现成的成果可以为你所用。它被称为Random.nextInt(int)，已经被添加到1.2发行版本的标准库java.util包中了。

你无需关心`nextInt(int)`的实现细节（如果你有强烈好奇心的话，可以研究它的文档或者源代码）。具有算法背景的高级工程师已经花了大量的时间来设计、实现和测试这个方法，然后经过这个领域中的专家的审查，以确保它的正确性。然后，标准库经过了beta测试，再被发行，几年之间已经有成千上万的程序员在使用它。在这个方法中还没有发现过缺陷，但是，如果将来发现有缺陷的话，在下一个发行版本中它就会被修正。通过使用标准库，你可以充分利用这些编写标准库的专家的知识，以及在你之前其他人的使用经验。

使用标准库的第二个好处是，你不必浪费时间为那些与你的工作关系不大的问题提供特别的解决方案。像大多数程序员一样，你应该把时间花在你的应用上，而不是底层的细节上。

使用标准库的第三个好处是，它们的性能会不断提高，而无需你做任何努力。因为许多人在使用它们，被当做工业标准在使用，所以，提供这些标准库的组织有强烈的动机要使它们运行得更快。例如，1.3发行版本中标准的多倍精度运算库`java.math`经过了重新设计和编写，从而导致性能上有了显著的提高。

标准库也会随着时间而增加新的功能。如果库中的一个类漏掉了某个重要的功能，开发者社团（developer community）会把这样的缺点告示出来。Java平台库也正是在这个社团的推动下不断发展的。以前这个程序是非正式的，现在有一个正式的程序，被称为Java社团程序（JCP, Java Community Process）。无论通过哪种方式，被遗漏的特征都会随着时间的推移而被加入进来。

使用标准库的最后一个好处是，你可以使自己的代码融入主流。这样的代码更易读、易维护、易被其他的开发人员重用。

既然有那么多的优点，那么，使用标准库设施而不选择专门的实现，这显然是符合逻辑的，然而还是有一部分程序员没有这样做。为什么？可能他们并不知道有这些库设施存在。在每一个主要的发行版本中，都会有许多新的特性被加入到库中，所以与这些库保持同步是值得的。你可以仔细阅读在线文档，或者阅读一些关于标准库的书籍[J2SE-APIs, Chan00, Flanagan99, Chan98]。标准库太庞大了，以至于不可能学习所有的文档，但是每一个程序员应该熟悉`java.lang`、`java.util`，某种程度还有`java.io`中的内容。关于其他库的知识可以根据需要随时学习。

总结库中所有的设施已经超出了本条目的范围，但是有一些设施值得一提。在1.2发行版本中，*Collections Framework*（集合框架）被加入到`java.util`包中。它应该成为每个程序员基本工具箱的一部分。*Collections Framework*是一个统一的体系结构，用来表示和管理集合，它使得这些集合的管理工作独立于它们的内部表示细节。它降低了编程的负担，同时还提高了性能。它允许不相关的API之间可以互操作，减少了为设计和学习新的API所要付出的

努力，并且鼓励软件重用。

这个框架是以六个集合接口为基础的，这六个集合接口是Collection、Set、List、Map、SortedSet和SortedMap。该框架包括这些接口以及相应的维护算法的实现。以前遗留下来的集合类Vector和Hashtable被更新之后，也加入到这个框架中，所以你不必为了使用这个框架而放弃这些类。

Collections Framework充分地降低了许多普通任务所需要的代码数量。例如，假设你有一个由一组字符串组成的向量（vector），并且希望对它按字母顺序进行排序。只需一行代码就可以完成这项任务：

```
Collections.sort(v);
```

如果你希望做同样的事情，但是忽略大小写区别，则使用下面的语句：

```
Collections.sort(v, String.CASE_INSENSITIVE_ORDER);
```

假设你要打印出一个数组中所有的元素，许多程序员会使用for循环，但是如果使用下面的做法，则根本不需要for循环：

```
System.out.println(Arrays.asList(a));
```

最后，假设你想知道在两个Hashtable实例h1和h2中包含同样映射值的所有键。[147]
Collections Framework被加进来之前，这项工作要求相当数量的代码，但是现在只需要三行代码：

```
Map tmp = new HashMap(h1);
tmp.entrySet().retainAll(h2.entrySet());
Set result = tmp.keySet();
```

前面的例子仅仅让你看到了Collections Framework功能的表面。如果要想知道得更多，请参考Sun的Web站点上的文档[Collections]，或者阅读有关的教程[Bloch99]。

一个值得注意的第三方库是Doug Lea的util.concurrent[Lea01]，它提供了一些高层的并发工具，可以简化多线程编程的工作。

Java 1.4发行版本中还有许多新增加的内容。其中值得注意的有下面一些：

- java.util.regex——一个非常成熟的类似Perl的正则表达式工具。
- java.util.prefs——一个用于永久存储“用户喜好信息和程序配置数据”的工具。
- java.nio——一个高性能的I/O工具，包括scalable I/O（类似于Unix的poll调用）

和*memory-mapped I/O*（类似于 Unix 的 mmap 调用），

- `java.util.LinkedHashSet`、`LinkedHashMap`、`IdentityHashMap`——新的集合实现。

在有些情况下，一个库设施并不能满足你的需要。你的需求越是特殊，这种情形越有可能发生。虽然你的第一个念头应该是选择标准库，但是，如果你在观察了它们在某些方面提供的功能之后，确定它不能满足你的需要，那么你就得使用其他的实现。任何一组库提供的功能总是难免会有缺漏。如果你所需要的功能不存在，那么，你只有自己实现这些功能，别无选择。

总而言之，不要从头发明轮子。如果你要做的事情看起来是很常见的，则可能库中已经有一个类完成了这样的工作。如果确实是这样，那么就使用它；如果还不清楚是否存在这样的类，那么就去查一查。一般而言，库的代码可能比你自己编写的代码更好，并且会随着时间的推移而不断改进。这并不是在影射你作为一个程序员的能力；从经济角度的分析表明：库代码受到的关注远远超过普通程序员在同样的功能上所能够给予的投入。
148

第31条：如果要求精确的答案，请避免使用float和double

`float`和`double`类型的主要设计目标是为了科学计算和工程计算。它们执行二进制浮点运算，这是为了在广域数值范围上提供较为精确的快速近似计算而精心设计的。然而，它们没有提供完全精确的结果，所以不应该被用于要求精确结果的场合。`float`和`double`类型对于货币计算尤为不合适，因为要让一个`float`或者`double`精确地表达0.1（或者10的任何其他负数次方值）是不可能的。

例如，假设你的口袋中有\$1.03，花掉了42¢之后还剩下多少钱呢？下面是一个很简单的程序片断，企图回答这个问题：

```
System.out.println(1.03 - .42);
```

不幸的是，它输出的结果是0.6100000000000001。这并不是偶然情况下才会出现的情形。假设你的口袋里有1美元，你买了9个垫圈，每个为10美分。那么你应该会找回多少钱呢？

```
System.out.println(1.00 - 9*.10);
```

根据这个程序片断，你得到\$0.0999999999999995。你可能会认为，只要在打印之前将结果做一下舍入就可以解决这个问题，但不幸的是，这并不总是可行的。例如，假设你的口袋里有1美元，你看到货架上有一排美味的糖果，标价分别为10¢、20¢、30¢等等，一直到1美元。你打算从标价为10¢的糖果开始，每一种买1个，一直到不能支付货架上下一种价格的糖果为止。那么你可以买多少个糖果？还会找回多少零钱？下面是一个简单的程序，用来解决这个问题：

```
// Broken - uses floating point for monetary calculation!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = .10; funds >= price; price += .10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Change: $" + funds);
}
```

149

如果你运行这个程序，你会发现你可以支付3个糖果，并且还剩下\$0.3999999999999999。这个答案是不正确的！解决这个问题的正确办法是使用`BigDecimal`、`int`或者`long`进行货币计算。下面的程序是上一个程序的直接翻译版本，它使用`BigDecimal`类型代替`double`：

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

```

final BigDecimal TEN_CENTS = new BigDecimal( ".10");

int itemsBought = 0;
BigDecimal funds = new BigDecimal("1.00");
for (BigDecimal price = TEN_CENTS;
     funds.compareTo(price) >= 0;
     price = price.add(TEN_CENTS)) {
    itemsBought++;
    funds = funds.subtract(price);
}
System.out.println(itemsBought + " items bought.");
System.out.println("Money left over: $" + funds);
}

```

如果你运行这个修改过的程序，你就会发现你可以支付4个糖果，还剩下\$0.00。这是正确的答案。然而，使用`BigDecimal`有两个缺点：与使用原语运算类型相比，这样做很不方便，而且更慢。对于解决这样一个简单的问题，后一个缺点并不要紧，但是前一个缺点可能会让你很不舒服。

除了使用`BigDecimal`之外，还有一种办法是使用`int`或者`long`（到底选用`int`或者`long`要取决于所涉及数值的大小），同时自己处理十进制小数点。在这个例子中，最明显的做法是以美分为单位进行计算，而不是以美元为单位。下面是这个例子的直接翻译版本，它说明了这种做法：

```

public static void main(String[] args) {
    int itemsBought = 0;
    int funds = 100;
    for (int price = 10; funds >= price; price += 10) {
        itemsBought++;
        funds -= price;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: +" + funds + " cents");
}

```

150

总而言之，对于有些要求精确答案的计算任务，请不要使用`float`或者`double`。如果你希望系统来处理十进制小数点，并且不介意因为不使用原语类型而带来的不便，那么请使用`BigDecimal`。使用`BigDecimal`还有一些额外的好处，它允许你完全控制舍入：当一个操作涉及到舍入的时候，它让你从8种舍入模式中选择其一。如果你正在进行商务计算，并且要求特别的舍入行为，那么使用`BigDecimal`是非常方便的。如果性能非常关键，并且你又不介意自己处理十进制小数点，而且所涉及的数值又不太大，那么可以使用`int`或者`long`。如果数值范围没有超过9位十进制数字，则你可以使用`int`；如果不超过18位数字，则可以使用`long`。如果数值范围超过了18位数字，你就必须使用`BigDecimal`。

151

第32条：如果其他类型更适合，则尽量避免使用字符串

字符串被用来表示文本，它在这方面也确实做得很好。因为字符串很常用，并且Java语言也支持得很好，所以自然地就会有这样一种倾向：即使在不适合于使用字符串的场合，人们往往也会使用字符串。本条目讨论这样一些不应该使用字符串的场合。

字符串不适合代替其他的值类型。当一段数据从文件、网络，或者键盘设备，进入到程序中之后，它通常以字符串的形式存在。一种自然的倾向是让它继续保留这种形式，但是，只有当这段数据本质上确实是文本信息时，这种倾向才是合理的。如果它是数值，那么它应该被转换为适当的数值类型，比如int、float或者BigInteger。如果它是一个“是-或-否”问题的答案，那么它应该被转换为boolean类型。更为一般地，如果存在一个适当的值类型，不管是原语类型，还是对象引用，你都应该使用这种类型；如果不存在这样的类型，那么你应该编写一个类型。虽然这条建议是显而易见的，但却经常被违反。

字符串不适合代替枚举类型。正如第21条中所讨论的，类型安全枚举类型（typesafe enum）和int值都比字符串更加适合用来表示枚举类型的常量。

字符串不适合代替聚集类型。如果一个实体有多个组件，那么，用一个字符串来表示这个实体通常是很不恰当的。例如，下面一行代码来自于一个真实的系统——标识符的名字已经被改变了，以免发生纠纷：

```
// Inappropriate use of string as aggregate type
String compoundKey = className + "#" + i.next();
```

这种方法有许多缺点。如果用做分隔域的字符也出现在某个域中，那么结果就会出现混乱。为了访问单独的域，你必须解析该字符串，这个过程非常慢，也很繁琐，还容易出错。你无法提供equals、toString或者compareTo方法，只好被迫接受String提供的行为。一个更好的做法是，简单地编写一个类来描述这个数据集，通常是一个私有的静态成员类（见第18条）。

字符串也不适合代替能力表（capabilities）。有时候，字符串被用于对某种功能进行授权访问。例如，考虑设计一个提供线程局部变量（thread-local variable）的基本设施。这个设施提供的变量在每个线程中都有自己的值。几年前面对这样的设计任务时，有些人独立地提出了同样的设计方案：利用客户提供的字符串键，对线程局部变量的内容进行访问授权：

```
// Broken - inappropriate use of String as capability!
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable
    // Sets the current thread's value for the named variable.
```

```

public static void set(String key, Object value);
// Returns the current thread's value for the named variable.
public static Object get(String key);
}

```

这种方法的问题在于，这些键代表了一个共享的全局名字空间。如果一个包的两个独立的客户决定为它们的线程局部变量使用同样的名字，那么，它们实际上无意识地共享了这个变量，这往往导致两个客户都会失败。而且，安全性就更差了，一个恶意的客户可以有意地使用与另一个客户相同的键，以便非法访问其他客户的数据。

修正这个API并不难，只要用一个不可伪造的键（unforgeable key，有时被称为能力（capability））来代替字符串就可以：

```

public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable

    public static class Key {
        Key() { }
    }

    // Generates a unique, unforgeable key
    public static Key getKey() {
        return new Key();
    }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}

```

虽然这解决了基于字符串的API的两个问题，但是你还可以做得更好。你实际上并不需要静态方法。它们可以被代之以键（Key）中的实例方法，这样这个键就不再是一个键了：它是一个线程局部变量。这时，这个不可被实例化的顶层类也不再做任何实质性的工作，所以你可以去掉这个顶层类，并且将嵌套类命名为ThreadLocal：

```

public class ThreadLocal {
    public ThreadLocal() { }
    public void set(Object value);
    public Object get();
}

```

粗略地讲，这正是java.util.ThreadLocal提供的API。除了解决了基于字符串的API的问题外，与前面两个基于键的API相比，它更快速、更加优美。

总而言之，如果可以使用更加合适的数据类型，或者可以编写更加适当的数据类型，那么应该避免使用字符串来表示对象。若使用不当，则字符串比其他类型更加笨拙、缺乏灵活性、速度缓慢，更加容易出错。通常被错误地用字符串来代替的类型包括原语类型、枚举类型和

154

聚集类型。

第33条：了解字符串连接的性能

字符串连接操作符（`+`, string concatenation operator）是把多个字符串合并为一个字符串的便利途径。要想产生一行输出，或者构造一个字符串来表示一个小的、大小固定的对象，使用连接操作符是非常合适的，但是它不适合规模比较大的情形。为连接n个字符串而重复地使用字符串连接操作符，要求n的平方级的时间。这是由于字符串是非可变的（见第13条）而导致的不幸结果。当两个字符串被连接的时候，它们的内容都要被拷贝。

例如，考虑下面的方法，它通过重复地连接账单的每一个项目，以便构造出一个代表该账单声明（billing statement）的字符串。代码如下：

```
// Inappropriate use of string concatenation - Performs horribly!
public String statement() {
    String s = "";
    for (int i = 0; i < numItems(); i++)
        s += lineForItem(i); // String concatenation
    return s;
}
```

如果项目数量巨大，那么这个方法的执行时间难以估算。为了获得可接受的性能，请使用**StringBuffer**替代**String**，用来存储构造过程中的账单声明：

```
public String statement() {
    StringBuffer s = new StringBuffer(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        s.append(lineForItem(i));
    return s.toString();
}
```

上述两种做法的性能差别是非常大的。如果**numItems**返回100，并且**lineForItem**返回一个固定长度为80字符的串，那么，在我的机器上，第二种做法比第一种做法快90倍。因为第一种做法的开销随项目数量呈平方级增加，而第二种做法是线性增加，所以，项目数越大，性能的差别会越显著。注意，第二种做法预先分配了一个**StringBuffer**，使它足够大以便能够容纳结果字符串。即使因为预先不知道字符串长度，而使用一个默认大小的**StringBuffer**，它仍然比第一种做法快45倍。

原则很简单：不要使用字符串连接操作符来合并多个字符串，除非性能无关紧要。相反，应该使用**StringBuffer**的**append**方法，或者采用其他的方案，比如使用字符数组，或者每次只处理一个字符串，而不是将它们组合起来。

第34条：通过接口引用对象

第25条中有一个建议：你应该使用接口，而不是类作为参数的类型。更一般地讲，你应该优先使用接口而不是类来引用对象。如果有合适的接口存在，那么对参数、返回值、变量和域的声明都应该使用接口类型。只有当你创建某个对象的时候，你才真正需要引用这个对象的类。为了使这一点更为具体，考虑Vector的情形，它是List接口的一个实现。在声明变量的时候应该养成这样的习惯：

```
// Good - uses interface as type
List subscribers = new Vector();
```

而不是这样的声明：

```
// Bad - uses class as type!
Vector subscribers = new Vector();
```

如果你养成了使用接口作为类型的习惯，那么你的程序将会更加灵活。如果你决定换一种实现，那么，你所需要做的全部工作是，改变构造代码中类的名字（或者使用一个不同的静态工厂）。例如，第一个声明可以被改变为：

```
List subscribers = new ArrayList();
```

周围的所有代码可以继续工作。周围的代码并不知道原来的实现类型，所以它们对于这种变化并不在意。

有一点值得注意：如果原来的实现提供了某种特殊的功能，而这种功能并不是这个接口的通用约定所要求的，并且周围的代码又依赖于这种功能，那么很关键的一点是，新的实现也要提供同样的功能。例如，如果第一个声明周围的代码依赖于Vector是同步的，那么在声明中用ArrayList代替Vector将是不正确的。

那么，为什么你要改变实现呢？因为新的实现提供了更好的性能，或者因为它提供了期望得到的额外功能。一个实际的例子与ThreadLocal类有关。在内部，这个类在Thread中使用了一个包级私有的Map域，将每线程的值（per-thread values）与ThreadLocal实例关联起来。156 在1.3发行版本中，这个域被初始化为一个HashMap实例。在1.4发行版本中，一个新的、被称为IdentityHashMap的专用Map实现被加入到平台中。只需将初始化域的那行代码改变为IdentityHashMap，代替原来的HashMap，则ThreadLocal设施就快了许多。

如果把这个域声明为HashMap而不是Map，则无法保证只改变一行代码就足够了。如果客户代码已经在Map接口之外使用了HashMap操作，或者把这个映射表（Map）传递给一个

要求HashMap的方法，那么，若将该域改变为一个IdentityHashMap，则代码就不再能通过编译。用接口类型声明域要“保持你的诚实”。

如果没有合适的接口存在的话，那么，用类而不是接口来引用一个对象，是完全合适的。例如，考虑值类（*value class*），比如String和BigInteger。记住，值类很少会有多个实现。它们通常是final的，并且很少有对应的接口。使用一个值类作为参数、变量、域，或者返回类型是完全合适的。更一般地，如果一个具体类没有相关联的接口，那么不管它是否表示一个值，你都没有别的选择，只有通过这个类来引用它的对象。Random类就属于这种情形。

不存在合适的接口类型的第二种情形是，对象属于一个框架，而框架的基本类型是类，不是接口。如果一个对象属于这样一个基于类的框架（*class-based framework*），那么应该用相关的基类（*base class*）（往往是抽象类）来引用这个对象，而不是用它的实现类。java.util.TimerTask类就属于这种情形。

不存在合适的接口类型的最后一种情形是，一个类实现了一个接口，但是它提供了接口中不存在的额外方法——例如LinkedList。如果程序依赖于这些额外的方法，那么这样的类应该只被用来引用它的实例：它永远也不应该被用作参数类型（见第25条）。

以上这些情形并不全面，而只是代表了一些“适合于用类来引用对象”的情形。实际上，一个给定的对象是否具有合适的接口应该是很显然的。如果是的话，那么，使用接口来引用对象会使程序更加灵活；如果不是，则使用类层次结构中提供了所需功能的最高层的类。

157

第35条：接口优先于映像机制

映像设施（reflection facility）`java.lang.reflect`，提供了“通过程序来访问关于已装载的类的信息”的能力。给定一个`Class`实例，你可以获得`Constructor`、`Method`和`Field`实例，分别代表了该`Class`实例所表示的类的构造函数、方法和域。这些对象提供了“通过程序来访问类的成员名字、域类型、方法原型等信息”的能力。

而且，`Constructor`、`Method`和`Field`实例使你能够维护它们的底层对等体（“反射到底层”）：通过调用`Constructor`、`Method`和`Field`实例上的方法，你可以构造底层类的实例、调用底层类的方法、访问底层类中的域。例如，`Method.invoke`使你可以调用任何类的任何对象上的任何方法（当然遵从常规的安全限制）。映像机制（reflection）允许一个类使用另一个类，即使当前者被编译的时候后者还根本不存在。然而，这种能力也需要付出代价：

- 你损失了编译时类型检查的好处，也包括异常检查。如果一个程序企图用映像方式调用一个不存在的方法，或者一个不可访问的方法，那么在运行时刻它将会失败，除非你采取了特别的预防措施。
- 要求执行映像访问的代码非常笨拙和冗长。编写这样的代码非常乏味，阅读这样的代码也很困难。
- 性能损失。对于1.3发行版本，在我的机器上，映像方法调用比普通方法调用慢了40倍。在1.4发行版本中，为了大幅提高性能，映像机制被重新构造了，但是仍然比普通的访问慢2倍，而且，这个差距不太可能再缩小了。

映像设施最初是为了基于组件的应用创建工具而设计的。这样的工具通常要根据需要装载类，并且用映像功能找出它们支持哪些方法和构造函数。这些工具允许用户交互式地构建出访问这些类的应用程序，但是这些被构建出来的应用程序可以以正常方式访问这些类，而不是映像方式。映像功能是在设计时刻（*design time*）被使用的：通常，普通应用在运行时刻不应该以映像方式访问对象。

有一些复杂的应用程序需要使用映像机制，其中包括类浏览器、对象监视器、代码分析工具、内嵌的解释器系统。在RPC系统中使用映像机制也是非常合适的，这样可以不再需要存根编译器（stub compiler）。如果你对自己的应用程序是否也属于这一类应用程序感到疑惑，那么它很有可能不属于。

如果只是在很有限的情况下使用映像机制，那么虽然也会付出少许代价，但你可以获得许多好处。对于有些程序，它们用到的类在编译时刻是不可用的，但是在编译时刻存在适当

的接口或者超类，通过它们可以引用到这些类（见第34条）。如果是这种情况，那么你可以以映像方式创建实例，然后通过它们的接口或者超类，以正常方式访问这些实例。如果存在适当的构造函数不带参数（这是通常的情形），那么你甚至根本不需要使用`java.lang.reflect包; Class.newInstance`方法就已经提供了所需的功能。

例如，下面的程序创建一个`Set`实例，它的类是由第一个命令行参数指定的。该程序把其余的命令行参数插入到这个集合中，然后打印该集合。不管第一个参数是什么，程序都会打印出余下的命令行参数，其中重复的参数被消除掉。这些参数被打印的顺序取决于第一个参数中指定的类。如果你指定“`java.util.HashSet`”，则显然这些参数会以随机的顺序被打印出来；如果指定“`java.util.TreeSet`”，则它们被按照字母顺序打印出来，因为`TreeSet`中的元素是已排序的。代码如下：

```
// Reflective instantiation with interface access
public static void main(String[] args) {
    // Translate the class name into a class object
    Class cl = null;
    try {
        cl = Class.forName(args[0]);
    } catch(ClassNotFoundException e) {
        System.err.println("Class not found.");
        System.exit(1);
    }

    // Instantiate the class
    Set s = null;
    try {
        s = (Set) cl.newInstance();
    } catch(IllegalAccessException e) {
        System.err.println("Class not accessible.");
        System.exit(1);
    } catchInstantiationException e) {
        System.err.println("Class not instantiable.");
        System.exit(1);
    }

    // Exercise the set
    s.addAll(Arrays.asList(args).subList(1, args.length-1));
    System.out.println(s);
}
```

159

尽管这个程序只是一个“玩具”，但是它所演示的这项技术是非常强大的。这个玩具程序可以很容易地变成一个集合类测试器，这个集合类测试器通过操作一个或者多个集合实例，并检查是否遵守`Set`接口的约定，以此来验证指定的`Set`实现。类似地，它也可以变成一个通用的集合类性能分析工具。实际上，它演示的这项技术足可以实现一个成熟的服务提供者框架（*service provider framework*）（见第1条）。绝大多数情况下，你对于映像机制的需要也就是这项技术。

在例子中，你也可以看到映像机制的两个缺点。第一，这个例子能够产生3个运行时错误，

如果不使用映像方式的实例化，那么这3个错误都会成为编译时错误。第二，根据类名生成它的实例需要20行冗长的代码，而调用一个构造函数可以非常简洁地只使用一行代码。然而，这些缺点仅仅局限于实例化对象的那部分代码。一旦一个对象被实例化，那么它与其他的set实例就难以区分。在实际程序中，通过限定使用映像技术，绝大部分代码可以不受影响。

一种合法（但较少）使用映像的做法是，打破一个类对于其他类、方法或者域（它们在运行时刻可能不存在）的依赖性。如果你要编写一个包，它运行的时候需要依赖其他某个包的多个版本，那么这种做法可能会非常有用。这项技术是，在你的包所需要的最小环境下对它进行编译，所谓最小环境通常是最老的版本，然后以映像方式访问任何新的类或者方法。为了使这项技术能够工作，如果你企图访问的新类或者新方法在运行时刻不存在，那么你还需要做一些适当的动作。所谓适当的动作，包括使用某种其他可替换的办法来达到同样的目的，或者使用简化的功能进行处理。

简而言之，映像机制是一项功能强大的设施，对于一些特定的复杂程序设计任务它是非常必要的，但它也有一些缺点。如果你编写的程序必须要与编译时刻未知的类一起工作，那么，有可能的话，仅仅使用映像机制实例化对象，而访问对象时使用编译时刻已知的某个接口或者超类。

第36条：谨慎地使用本地方法

Java Native Interface (JNI) 允许Java应用可以调用本地方法 (*native method*)，所谓本地方法是指用本地程序设计语言（比如C或者C++）来编写的特殊方法。本地方法在本地语言中可以执行任意的计算任务，然后返回到Java程序设计语言。

从历史上看，本地方法主要有三种用途。它们提供了“访问与平台相关的设施”的能力，比如访问注册表（registry）和文件锁。它们也提供了访问老式代码库的能力，通过这些老式代码库进一步可以访问老式数据（legacy data）。最后，应用程序还可以使用本地方法，通过本地语言，实现性能关键的部分，以提高系统的性能。

使用本地方法来访问与平台相关的设施是合法的，但是随着Java平台的不断成熟，它提供了越来越多以前只有在宿主平台上才拥有的特性。例如，1.4发行版本中新增加的 `java.util.prefs` 包，提供了注册表功能。使用本地方法来访问老式代码也是合法的，但是，访问某些老式代码还有更好的办法。例如，JDBC API提供了访问老式数据库的能力。

随着1.3发行版本的推出，使用本地方法来提高性能的做法已经不值得提倡。在早期的发行版本中，这样做往往是很必要的，但是JVM实现变得越来越快了。对于大多数任务，即使不使用本地方法也可以获得与之相当的性能。举例来说，当 `java.math` 被加入到1.1发行版本的平台中时，`BigInteger` 是在一个用C编写的快速多倍精度运算库的基础上实现的。在当时，为了获得足够的性能这样做是必要的。在1.3发行版本中，`BigInteger` 被完全用Java重写了，并且也被仔细调整了性能。在所有Sun的1.3 JVM实现中，对于绝大多数操作和操作数尺寸（size），新的版本都比原来的版本更快。

使用本地方法有一些严重的缺点。因为本地语言不是安全的（见第24条），所以，使用本地方法的应用程序也不再免受内存毁坏错误的影响。因为本地语言是平台相关的，所以使用本地方法的应用程序也不再是可自由移植的。对于每一个目标平台，本地代码都必须经过重新编译，也有可能要求做一些修改。同时，在进入和退出本地代码时，也需要较高的固定开销。所以，如果本地代码只是做少量工作的话，本地方法可能会降低（decrease）性能。最后，本地方法编写起来单调乏味，并且难以阅读。

简而言之，在使用本地方法之前请仔细考虑。很少情况下需要使用本地方法来提高性能。如果你必须要使用本地方法来访问底层的资源，或者遗留代码库，那么，尽可能少用本地代码，并且全面进行测试。本地代码中的一个错误（bug）可以破坏整个应用程序。

第37条：谨慎地进行优化

有三条与优化有关的格言是每个人都应该知道的。我们对它们可能已经听得太多了，但是，如果你对它们还不太熟悉，请看下面：

很多计算上的过失都被归咎于效率原因（没有获得必要的效率），而不是其他的原因——甚至包括盲目地做傻事。

——William A. Wulf[Wulf72]

不要去计较一些小的效率上的得失，在97%的情况下，不成熟的优化是一切问题的根源

——Donald E. Knuth[Knuth74]

在优化方面，我们应该遵守两条规则：

规则1：不要做优化。

规则2（仅针对专家）：还是不要做优化——也就是说，在你还没有绝对清晰的未优化方案之前，请不要做优化。

——M. A. Jackson[Jackson75]

所有这些格言都比Java程序设计语言的出现早了20年。它们讲述了一个关于优化的深刻真理：优化更容易带来伤害，而不是好处，特别是不成熟的优化。在优化过程中，你产生的软件可能既不快速，也不正确，而且还不容易被修正。

不要因为性能而牺牲合理的结构。努力编写好的程序而不是快的程序。如果一个好的程序不够快，它的结构将使得它可以被优化。好的程序体现了信息隐藏的原则：只要有可能，它们就会把设计决定限定在局部的单个模块中，所以，单个决定可以被改变，并且不会影响到系统的其他部分（见第12条）。

这并不意味着，在完成程序之前你就可以忽略性能问题。实现上的问题可以通过后期的优化而被改正，但是，遍布全局并且限制性能的结构缺陷几乎是不可能被改正的，除非重新编写系统。在系统完成之后再改变你的设计的某个基本方面，会导致你的系统结构病态，从而难以维护和改进。因此你应该在设计过程中考虑性能问题。

努力避免那些限制性能的设计决定。当一个系统设计完成之后，其中最难以更改的组件

[162] 是那些指定了模块之间交互关系以及模块与外界交互关系的组件。在这些设计组件之中，最

主要的是API、线路层（wire-level）协议以及永久数据格式。不仅这些设计组件在事后难以甚至不可能被改变，而且它们都有可能对系统本该达到的性能产生重要的限制。

考虑你的API设计决定的性能后果。使一个公有的类型成为可变的（*mutable*），这可能会导致大量不必要的保护性拷贝（见第24条）。类似地，在一个适合使用复合模式的公有类中使用继承，会把这个类与它的超类永久地束缚在一起，从而人为地限制了子类的性能（见第14条）。最后一个例子，在一个API中使用实现类型而不是接口，会把你束缚在一个具体的实现上，即使将来出现更快的实现你也无法使用（见第34条）。

API设计对于性能的影响是非常实际的。考虑java.awt.Component类中的getSize方法。第一个决定是，这个性能关键的方法返回一个Dimension实例；第二个决定是，Dimension实例是可变的。这两个决定使得getSize方法的实现必须为每一个调用分配一个新的Dimension实例。尽管从1.3发行版本开始，分配小的对象开销并不大，但是分配数百万个不必要的对象仍然会严重损害性能。

在这种情况下，有几种可供选择的替换方案。理想情况下，Dimension应该是非可变的（见第13条）；另一种方案是，用两个方法来替换getSize方法，它们分别返回Dimension对象的单个原语属性^Θ。实际上，在1.2发行版本中，出于性能的原因，两个这样的方法已经被加入到Component API中。然而，原先的客户代码仍然可以使用getSize方法，但是仍然要承受原始API设计决定所带来的性能后果。

幸运的是，一般而言，好的API设计也伴随着好的性能。**为获得好的性能而对API进行曲改，这是一个非常不好的想法。**导致你对API进行曲改的性能因素可能会在将来的平台版本中，或者在将来的底层软件中不复存在，但是被曲改的API以及由它引起的问题将永远困扰着你。

一旦你已经谨慎地设计了你的程序，并且产生了一个清晰、简明、结构良好的实现，那么是该考虑优化的时候了（假定此时你对于程序的性能并不满意）。回想一下本条目刚开始时提到的Jackson的两条规则：“不要做优化”以及“（仅针对专家）还是不要做优化”。他可以再增加一条：**在每次试图做优化之前和之后，请对性能进行测量。**

163

你可能会惊讶于你的发现。你试图做的优化通常对于性能没有明显的影响，有时候甚至会使性能更坏。主要的原因在于，要猜测出你的程序把时间花在哪些地方并不容易。你认为你的程序慢的地方可能并没有问题，这种情况下你实际上在浪费时间去做优化。一个普遍的认识是：程序把80%的时间花在20%的代码上。

性能分析工具有助于你做出决定：应该把优化的重心放在哪里。这样的工具可以为你提供

^Θ 即每一维的大小。——译注

第8章

异常

充分发挥异常的优点，可以提高一个程序的可读性、可靠性和可维护性。如果使用不当的话，它们也会带来负面影响。本章提供了一些关于有效使用异常的指导原则。

第39条：只针对不正常的条件才使用异常

某一天，如果你不走运的话，可能会碰到下面这样的代码：

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while(true)
        a[i++].f();
} catch(ArrayIndexOutOfBoundsException e) {
}
```

这段代码在做什么？看起来并不很明显，这也是它没有被真正使用的原因。事实证明，作为一个对数组元素进行遍历的实现方式，它的构想是非常拙劣的。当这个循环企图访问数组边界之外的第一个元素时，用抛出（`throw`）、捕获（`catch`）、忽略`ArrayIndexOutOfBoundsException`的手段来达到终止无限循环的目的。假定它与数组循环的标准模式是等价的，那么，对于任何一个Java程序员，下面的标准模式一看就明白：

169 `for (int i = 0; i < a.length; i++)
 a[i].f();`

那么，为什么有人会优先使用基于异常的模式，而不是经过实验的可靠模式呢？这是被误导了，他们企图利用Java的错误判断机制来提高性能，因为VM对每次数组访问都要检查越界情况，所以他们认为循环终止测试（`i < a.length`）是多余的，应该被避免。这种想法有三个错误：

- 因为异常机制的设计初衷是用于不正常的情形，所以很少会有JVM实现试图对它们的性

第38条：遵守普遍接受的命名惯例

Java平台有一整套建立得很好的命名惯例 (*naming convention*)，其中许多被包含在《The Java Language Specification》[JLS, 6.8]中。不严格地讲，这些命名惯例分为两大类：字面的 (typographical) 和语法的 (grammatical)。

字面的命名惯例比较少，但也涉及到包、类、接口、方法和域。你应该尽量少违反这些惯例，如果没有理由的话永远不要违反。如果一个API违反了这些惯例，它可能会难以使用。如果一个实现违反了它们，它可能会难以维护。在这两种情况下，违反惯例都会潜在地给使用这些代码的其他程序员带来困惑和苦恼，并且使他们做出错误的假设，导致程序的错误。本条目将对这些惯例做简要的介绍。

包的名字应该是层次状的，用句号分隔每一部分。每一部分包括小写字母和数字（很少使用数字）。如果你的包将在你的组织之外被使用，那么包的名字应该以你的组织的Internet域名作为开头，并且顶级域名放在前面，例如`edu.cmu`、`com.sun`、`gov.nsa`。标准库和一些可选的库，其名字以`java`和`javax`作为开头，它们是这条规则的例外。用户创建的包的名字不应该以`java`和`javax`作为开头。关于将Internet域名转换为包名字前缀的详细规则，请参见《The Java Language Specification》[JLS, 7.7]。

包名字的剩余部分应该包括一个或者多个描述该包的组成部分。这些组成部分应该比较简短，通常不超过8个字符。鼓励使用有意义的缩写形式，例如，使用`util`而不是`utilities`，只取首字母的缩写形式也是可以接受的，例如`awt`。每一个组成部分通常应该由一个单词，或者缩写词组成。

许多包的名字只有一个组成部分再加上Internet域名。对于比较大的设施使用附加的部分是合适的，它们的规模要求它们被分割成一个非正式的层次结构。例如，`javax.swing`包有一个非常丰富的包层次，比如`javax.swing.plaf.metal`。这样的包通常被称为子包，尽管它们只是习惯意义上的子包；Java语言并没有提供对包层次的支持。

类和接口的名字应该包括一个或者多个单词，每个单词的首字母大写，例如`Timer`和`TimerTask`。缩写应该尽量避免，除非是一些首字母缩写和一些通用的缩写，比如`max`和`min`。对于首字母缩写，到底应该全部大写还是只有首字母大写，没有统一的说法。虽然全部大写更为常见一些，但是强烈建议采用仅仅首字母大写的形式。即使多个“首字母缩写形式”连续出现，你仍然可以区分开一个单词的起始处和结束处。下面两个类名你更愿意看到哪一个，`HTTPURL`还是`HttpUrl`？

165

方法和域的名字与类和接口的名字遵守相同的字面惯例，只不过方法或者域的名字的第一

个字母应该为小写，例如remove、ensureCapacity。如果一个由首字母缩写组成的单词是一个方法或者域名字的第一个单词，那么它应该是小写形式。

上述规则的惟一例外是“常量域”，它的名字应该包含一个或者多个大写形式的单词，中间由下划线符号分隔开，例如VALUES或NEGATIVE_INFINITY。常量域是一个静态final域，它的值是不可变的，如果一个静态final域有一个原语类型，或者一个非可变的引用类型（见第13条），那么它是一个常量域。如果这个类型可能是可变的，那么，若被引用的对象是不可变的，则它仍然可以是一个常量域。例如，一个类型安全枚举类型可以把它的枚举常量导出到一个非可变的List常量中（见第21条关于readResolve的例子）。注意，常量域是惟一推荐使用下划线的情形。

局部变量名字的字面命名惯例与成员名字类似，只不过它也允许缩写，单个字符和短字符串的意义依赖于局部变量所在的上下文环境，例如i、xref和houseNumber。

为了快速查阅，表7-1显示了字面惯例的例子。

表7-1 字面惯例的例子

标识符类型	例 子
包	com.sun.media.lib, com.sun.jdi.event
类或者接口	Timer, TimerTask, KeyFactorySpi, HttpServlet
方法或域	remove, ensureCapacity, getCrc
常量域	VALUES, NEGATIVE_INFINITY
局部变量	i, xref, houseNumber

语法命名惯例比字面惯例更加灵活，也更有争议。对于包而言，没有语法命名惯例。
166 常用一个名词或者名词短语命名，例如Timer或者BufferedWriter。接口的命名与类相似，例如Collection或Comparator，或者用一个以“-able”或“-ible”结尾的形容词来命名，例如Runnable或Accessible。

执行某个动作的方法通常用一个动词或者动词短语来命名，例如append或drawImage。对于返回boolean值的方法，其名字往往以单词“is”开头，后面跟一个名词或名词短语，或者具有形容词功能的任何单词或短语，例如isDigit、isProbablePrime、isEmpty、isEnabled、isRunning。

如果一个方法返回被调用对象的一个非boolean的函数或者属性，则往往用一个名词、名词短语，或者一个以动词“get”开头的动词短语来对它命名，例如size、hashCode或者getTime。有一个组织声称只有第三种形式（以“get”开头）才可以接受，但是这种声称没有得到支持。前两种形式往往会产生可读性更好的代码，例如：

```
if (car.speed() > 2 * SPEED_LIMIT)
    generateAudibleAlert("Watch out for cops!");
```

如果一个方法所在的类是一个Bean[JavaBeans]，则强制要求使用以“get”开头的形式，而且，如果你在考虑将来把这个类转变成一个Bean，那么这样做也是明智的。另外，如果这个类包含一个方法用于设置同样的属性，则强烈建议采用这种形式。在这种情况下，这两个方法应该被命名为`getAttribute`和`setAttribute`。

有些方法的名字值得专门提及。转换对象类型的方法、返回一个不同类型的独立对象的方法，通常被称为`toType`，例如`toString`和`toArray`。返回一个视图（`view`，见第4条，视图的类型不同于接收对象的类型）的方法通常被称为`asType`，例如`asList`。返回一个与被调用对象同值的原语类型的方法，通常被称为`typeValue`，例如`intValue`。静态工厂的常用名字为`valueOf`和`getInstance`（见第1条）。

域名字的语法惯例没有很好地建立起来，也没有类、接口和方法名字的惯例那么重要，因为设计良好的API很少会包含被暴露出来的域。`boolean`类型的域与`boolean`类型的访问方法（accessor method）很类似，但是省去了初始的“is”，例如`initialized`和`composite`。其他类型的域通常用名词或者名词短语来命名，比如`height`、`digits`或`bodyStyle`。局部变量的语法惯例类似于域的语法惯例，但是更弱。

167

总而言之，把标准的命名惯例当做一种内在的机制来看待，并且学习如何使用它们。字面惯例是非常直接和明确的；而语法惯例更复杂，也更松散。下面这句话引自《The Java Language Specification》[JLS, 6.8]：“如果长期养成的习惯用法与此不同的话，请不要盲目遵从这些命名惯例。”请使用大家公认的做法。

168

第8章

异常

充分发挥异常的优点，可以提高一个程序的可读性、可靠性和可维护性。如果使用不当的话，它们也会带来负面影响。本章提供了一些关于有效使用异常的指导原则。

第39条：只针对不正常的条件才使用异常

某一天，如果你不走运的话，可能会碰到下面这样的代码：

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while(true)
        a[i++].f();
} catch(ArrayIndexOutOfBoundsException e) { }
```

这段代码在做什么？看起来并不很明显，这也是它没有被真正使用的原因。事实证明，作为一个对数组元素进行遍历的实现方式，它的构想是非常拙劣的。当这个循环企图访问数组边界之外的第一个元素时，用抛出（throw）、捕获（catch）、忽略`ArrayIndexOutOfBoundsException`的手段来达到终止无限循环的目的。假定它与数组循环的标准模式是等价的，那么，对于任何一个Java程序员，下面的标准模式一看就明白：

```
for (int i = 0; i < a.length; i++)
    a[i].f();
```

那么，为什么有人会优先使用基于异常的模式，而不是经过实验的可靠模式呢？这是被误导了，他们企图利用Java的错误判断机制来提高性能，因为VM对每次数组访问都要检查越界情况，所以他们认为循环终止测试（`i < a.length`）是多余的，应该被避免。这种想法有三个错误：

- 因为异常机制的设计初衷是用于不正常的情形，所以很少会有JVM实现试图对它们的性

能做优化。所以，创建、抛出和捕获异常的开销是很昂贵的。

- 把代码放在try-catch块中反而阻止了现代JVM实现本来可能要执行的某些特定的优化。
- 对数组进行遍历的标准模式并不会导致冗余的检查；有些现代的JVM实现会将它们优化掉。

实际上，在所有当前的JVM实现上，基于异常的模式比标准模式要慢得多。在我的机器上，当循环从0到99的时候，基于异常的模式比标准模式慢70倍。

基于异常的循环模式不仅模糊了代码的意图，降低了它的性能，而且它不能保证正常工作。如果出现了不相关的错误（bug），这个模式会悄悄地失效，从而掩盖了这个错误，极大地增加了调试过程的复杂性。假设循环体中的计算过程包含一个错误，该错误会导致对某个不相关的数组的越界访问。如果使用了一个合理的循环模式，那么这个错误会产生一个未被捕捉的异常，从而导致线程立即结束，产生相应的错误消息。如果使用了这个可恶的基于异常的循环模式，那么与这个错误相关的异常将会被捕捉到，并且被误译为一个正常的循环终止条件。

这个例子的教训很简单：顾名思义，异常只应该被用于不正常的条件，它们永远不应该被用于正常的控制流。更一般地，你应该优先使用标准的、容易理解的模式，而不是那些声称可以提供更好性能的、弄巧成拙的模式。即使真的能够改进性能，面对JVM实现的不断改进，这种模式的性能优势也许不复存在。然而，由这种过度聪明的模式带来的隐藏的错误，以及维护的痛苦却依然存在。

这条原则对API设计也有启示。一个设计良好的API不应该强迫它的客户为了正常的控制流而使用异常。如果一个类具有一个“状态相关（state-dependent）”的方法，即只有在特定的不可预知的条件下才可以被调用的方法，那么这个类往往也应该有一个单独的“状态测试（state-testing）”方法，即指示是否可以调用第一个方法。例如，Iterator类有一个“状态相关”的next方法，它返回迭代过程中的下一个元素，对应的“状态测试”方法为hasNext，这使得下面针对集合迭代的标准模式成为可能：

```
for (Iterator i = collection.iterator(); i.hasNext(); ) {
    Foo foo = (Foo) i.next();
    ...
}
```

如果Iterator缺少hasNext方法，那么客户将被迫改用下面的做法：

```
// Do not use this hideous idiom for iteration over a collection!
try {
    Iterator i = collection.iterator();
    while(true) {
        Foo foo = (Foo) i.next();
```

170

```
    ...
}
```

```
} catch (NoSuchElementException e) {
```

```
}
```

这个对集合进行迭代的例子非常类似于本条目刚开始时针对数组进行迭代的例子。除了代码啰嗦和令人误解之外，这个基于异常的模式可能执行起来比标准模式更差，而且也可能掩盖系统中其他不相关部分中的错误。

除了提供一个单独的状态测试方法之外，另一种做法是，如果“状态相关”方法被调用的时候，该对象处于不适当的状态之中，则它返回一个可被识别的值，比如`null`。这项技术对于`Iterator`而言并不合适，因为`null`是`next`方法的合法返回值。

对于“状态测试方法”和“可被识别的返回值”这两种做法，下面有一些指导原则可以帮助你在两者之中做出选择。如果一个对象将会在缺少外部同步的情况下被并发访问，或者可被外界改变状态，那么使用一个可被识别的返回值可能是很有必要的，因为在调用“状态测试”方法和调用对应的“状态相关”方法的时间间隔之中，对象的状态有可能会发生变化。如果一个单独的“状态测试”方法必须要重复“状态相关”方法的工作，那么从性能角度考虑，应该使用可被识别的返回值。然而，如果所有其他方面都是等同的，那么“状态测试”方法略优于可被识别的返回值。它提供了更好的可读性，对于使用不当的情形，可能更加易于检测和改正。

第40条：对于可恢复的条件使用被检查的异常，对于程序错误使用运行时异常

Java程序设计语言提供了三种可抛出结构 (*throwable*)：被检查的异常 (*checked exception*)、运行时异常 (*run-time exception*) 和错误 (*error*)。什么时候适合使用哪一种可抛出结构呢？对此，程序员中间存在一些困惑。虽然这项决定并不总是那么清晰，但还是有一些一般性的原则有助于做出合理的选择，也使得选择起来更为容易。

在决定使用一个被检查的异常或是一个未被检查的异常时，主要的原则是：如果期望调用者能够恢复，那么，对于这样的条件应该使用被检查的异常。通过抛出一个被检查的异常，你强迫调用者在一个 `catch` 子句中处理该异常，或者将它传播到外面。对于一个方法声明要抛出的每一个被检查的异常，它是对API用户的一种潜在指示：与异常相关联的条件是调用这个方法的一种可能结果。

API设计者让API用户面对一个被检查的异常，以此强制要求用户能从这个异常条件下恢复。用户可以忽视这样的强制要求，只需把异常捕获下来然后忽略即可，但是这往往不是一个好的做法（见第47条）。

有两种未被检查的可抛出结构：运行时异常和错误。在行为上两者是等同的：它们都是不需要，也不应该被捕获的抛出物。如果一个程序抛出一个未被检查的异常，或者一个错误，则往往是不可恢复的情形，继续执行下去有害无益。如果一个程序没有捕捉这样的可抛出结构，则将会导致当前线程停止 (`halt`)，并伴以一个适当的错误消息。

用运行时异常来指明程序错误。大多数的运行时异常都是表明前提违例 (*precondition violation*)。所谓前提违例是指API的客户没有遵守API规范建立的约定。例如，关于数组访问的约定指明了数组的下标值必须在零和数组长度减1之间。`ArrayIndexOutOfBoundsException` 表明这个前提被违反了。

虽然JLS（Java语言规范）并没有要求，但是按照惯例，错误往往被JVM保留用于指示资源不足、约束失败，或者其他使程序无法继续执行的条件[Chan98, Horstman00]。由于这已经是一个几乎被普遍接受的惯例，所以最好不要再实现任何新的Error子类。你所实现的所有未被检查的抛出结构都应该是`RuntimeException`的子类（直接的或者间接的）。 172

要想定义一个抛出物，它并非是`Exception`、`RuntimeException`或`Error`的子类，这也是可能的。JLS并没有直接规定这样的抛出物，但是隐式地指定了：从行为意义上讲它们等同于普通的被检查异常（即`Exception`的子类，但不是`RuntimeException`的子类）。那么，你什么时候会用到这样的抛出结构呢？总之，永远也不会。它与普通的被检查异常相

比没有任何益处，只会困扰API的用户。

总而言之，对于可恢复的条件，使用被检查的异常；对于程序错误，使用运行时异常。当然，情况并不总是那么黑白分明。例如，考虑资源枯竭的情形，这可能是由于程序错误而引起的，比如分配了一块不合理的过大的数组，也可能确实是由于资源不足而引起。如果资源枯竭是由于临时的短缺，或是临时需求太大所造成的，那么这个条件可能是可恢复的。这是API设计者需要做出判断的事情：这样的资源枯竭是否允许恢复。如果你相信一个条件可能允许恢复，那么使用一个被检查的异常；如果不是，则使用运行时异常。如果你并不清楚是否有可能恢复，那么最好使用一个未被检查的异常，原因请参见第41条的讨论。

API设计者往往会忘记，异常也是一个完全意义上的对象，在其上可以定义任意的方法。这些方法的主要用途是为捕获异常的代码提供额外的信息，特别是关于引发这个异常的条件的信息。如果缺少这样的方法，那么程序员必须要懂得如何解析“该异常的字符串表示”，以便获得这些额外信息。这是很不好的做法。一个类很少会规定它的字符串表示中的细节，因此，不同的实现，不同的版本，字符串表示会大相径庭。所以，“解析一个异常的字符串表示”的代码可能是不可移植的，也是非常脆弱的。

因为被检查的异常往往指示了可恢复的条件，所以，对于这样的异常，提供一些辅助方法是非常重要的，通过这些方法，调用者可以获得一些有助于恢复的信息。例如，假设因为一个用户没有储存足够数量的钱，所以他企图在一个收费电话上进行呼叫就会失败，于是一个被检查的异常被抛出来。这个异常应该提供一个访问方法，以便允许客户查询不足费用的数量，从而可将这个数目传递给电话用户。

第41条：避免不必要的使用被检查的异常

被检查的异常是Java程序设计语言的一个很好的特性。与返回代码不同，它们强迫程序员处理例外的条件，大大提高了可靠性。然而，过分使用被检查的异常会使API用起来非常不方便。如果一个方法会抛出一个或者多个被检查的异常，那么调用该方法的代码必须在一个或者多个catch块中处理这些异常，或者它必须声明这些异常，以便让它们传播出去。无论哪种方法，都给程序员增添了不可忽视的负担。

如果正确地使用API并不能阻止这种异常条件的产生，并且一旦产生了异常，使用API的程序员可以采取有用的动作，那么这种负担被认为是正当的。除非这两个条件都成立，否则更适合于使用未被检查的异常。作为一个“石蕊”测试（比喻简单而具有决定性的测试），你可以试着问自己，程序员将会如何处理该异常。下面的做法是最好的吗？

```
} catch(TheCheckedException e) {
    throw new Error("Assertion error"); // Should never happen!
}
```

下面这种做法如何？

```
} catch(TheCheckedException e) {
    e.printStackTrace();           // Oh well, we lose.
    System.exit(1);
}
```

如果使用API的程序员无法做得比这更好，那么未被检查的异常可能更为合适。这样的一个例子是CloneNotSupportedException。它是被Object.clone抛出来的，而Object.clone应该只是在实现了Cloneable的对象上才可以被调用（见第10条）。在实践中，catch块几乎总是具有断言（assertion）失败的特征。异常被检查的本质为程序员并没有提供任何好处，相反，它会要求付出专门的努力，并且使程序更为复杂。

如果一个方法抛出的被检查异常是唯一的（*sole*），那么它给程序员带来的额外负担会非常高。如果这个方法还有其他的被检查异常，那么，该方法被调用的时候，必须已经出现在一个try块中，所以这个异常仅仅要求另外一个catch块。如果一个方法只抛出一个被检查的异常，那么仅仅为了这个异常，该方法必须放置于try块中。在这样的情形下，你应该问自己，是否可以有别的途径来避免使用被检查的异常。

174

“把被检查的异常变成未被检查的异常”的一种技术是，把这个要抛出异常的方法分成两个方法，其中第一个方法返回一个boolean，表明是否应该抛出异常。这是一种API转换，它实际上把下面的调用序列：

```
// Invocation with checked exception
try {
    obj.action(args);
} catch(TheCheckedException e) {
    // Handle exceptional condition
    ...
}
```

转换为：

```
// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    // Handle exceptional condition
    ...
}
```

这种转换并不总是合适的，但是，凡是在合适的地方，它都会使API用起来更加舒服。虽然后者的调用序列没有前者漂亮，但是这样得到的API更加灵活。如果程序员知道调用将会成功，或者不介意由于调用失败而导致的线程终止，那么，这种转换还允许以下更为简单的调用形式：

```
obj.action(args);
```

如果你怀疑这个简单的调用序列是否会合乎要求，那么这个API转换可能是合适的。转换之后的API在本质上等同于第39条中的“状态测试方法”，并且，同样的告诫也是适用的：如果一个对象将会在缺少外部同步的情况下被并发访问，或者可被外界改变状态，那么这种转换是不合适的，因为在actionPermitted和action这两个调用的时间间隔之中，对象的状态有可能会发生变化。如果一个单独的actionPermitted方法必须要重复action方法的工作，那么出于性能的考虑，这种API转换不值得再做。

第42条：尽量使用标准的异常

专家级程序员与缺乏经验的程序员一个最主要的区分是，专家追求高度的代码重用，并且通常也能实现这样的代码重用。代码重用是值得提倡的，这是一条通用的规则，异常也不例外。Java平台库提供了一组基本的未被检查的异常，它们覆盖了绝大多数API抛出异常的需要。本条目中，我们将讨论这些通用的、可重用的异常。

重用现有的异常有多方面的好处。其中最主要的好处是，它使得你的API更加易于学习和使用，因为它与程序员原来已经熟悉的习惯用法是一致的。第二个好处是，对于用到这些API的程序而言，它们的可读性更好，因为它们不会充斥着程序员不熟悉的异常。最后一点是，异常类越少，意味着内存占用（footprint）越小，并且装载这些类的时间开销也越小。

最常被重用的异常是`IllegalArgumentException`。通常，当调用者传递的参数值不合适的时候，这个异常就会被抛出来。例如，假设一个参数代表了“某个动作的重复次数”，如果程序员传递了一个负数，则这个异常就会被抛出来。

另一个常被重用的异常是`IllegalStateException`。若给定了接收对象的状态，如果调用非法的话，则通常会抛出这个异常。例如，如果在一个对象被正确地初始化之前，调用者就企图使用这个对象，那么这个异常就会被抛出来。

可以这么说，所有错误的方法调用都可以被归结为非法参数或者非法状态，但是，其他还有一些标准异常也被用于某些特定情况下的非法参数和非法状态。如果一个调用者在某个不允许`null`值的参数中传递了一个`null`，则习惯的做法是抛出`NullPointerException`，而不是`IllegalArgumentException`。类似地，如果调用者在一个表示序列下标的参数中，传递了一个越界的值，则应该抛出的是`IndexOutOfBoundsException`，而不是`IllegalArgumentException`。

另一个值得知晓的通用异常是`ConcurrentModificationException`。如果一个专门被设计为用于单线程的对象，或者一个与外部同步机制配合使用的对象检测到它被并发地修改，则这个异常应该被抛出。

最后一个值得注意的通用异常是`UnsupportedOperationException`。如果一个对象并不支持所请求的方法，那么这个异常将会被抛出。与本条目中讨论的其他异常相比，它很少被使用，因为绝大多数对象都会支持它们实现的所有方法。如果一个接口的具体实现没有实现该接口所定义的一个或者多个可选操作，那么它就可以使用这个异常。例如，对于一个仅支持追加操作的`List`实现，如果有人试图删除一个元素，则它就会抛出这个异常。

表8-1概括了最常见的可重用异常。

表8-1 常用的异常

异常	使用场合
IllegalArgumentException	参数的值不合适
IllegalStateException	对于这个方法调用而言，对象状态不合适
NullPointerException	在null被禁止的情况下参数值为null
IndexOutOfBoundsException	下标越界
ConcurrentModificationException	在禁止并发修改的情况下，对象检测到并发修改
UnsupportedOperationException	对象不支持客户请求的方法

虽然它们是Java平台库中迄今为止最常被重用的异常，但是，在条件许可的情况下，其他的异常也可以被重用。例如，如果你要实现诸如复数或者矩阵之类的算术对象，那么重用ArithmaticException和NumberFormatException将是非常合适的。如果一个异常能够满足你的需要，则不要犹豫，使用就可以，不过，你一定要确保你抛出异常的条件与该异常的文档中描述的条件一致。这种重用必须建立在语义的基础上，而不是名字的基础上。而且，如果你希望增加更多的失败 - 捕获(failure-capture)信息(见第45条)，那么不用客气，你可以对已有的异常进行子类化。

最后，一定要清楚，选择重用哪一个异常并不总是一门精确的科学，因为表8-1中的“使用场合”不是相互排斥的。例如，考虑纸牌对象的情形。假设有一个用于发牌操作的方法，它的参数(handSize)是发一手牌的纸牌张数。假设调用者在这个参数中传递的值大于整副纸牌的剩余张数。那么，这种情形既可以被解释为IllegalArgumentException(handSize参数的值太大)，也可以被解释为IllegalStateException(相对于客户的请求而言，纸牌对象包含的纸牌太少)。在这个例子中，感觉上IllegalArgumentException要好一些，不过，这里并没有必须遵循的规则。

第43条：抛出的异常要适合于相应的抽象

如果一个方法抛出的异常与它所执行的任务没有明显的关联关系，这种情形会使人不知所措。当一个方法传递一个由低层抽象抛出的异常时，往往会发生这样的情况。除了使人感到困惑之外，这也“污染”了具有实现细节的高层API。如果高层的实现在后续的发行版本中发生了变化，那么它所抛出的异常也可能会跟着发生变化，从而会潜在地打破现有的客户程序。

为了避免这个问题，高层的实现应该捕获低层的异常，同时抛出一个可以按照高层抽象进行解释的异常。这种做法被称为异常转译（exception translation），如下所示：

```
// Exception Translation
try {
    // Use lower-level abstraction to do our bidding
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

下面的异常转译例子取自于AbstractSequentialList类，该类是List接口的一个骨架实现（skeletal implementation）（见第16条）。在这个例子中，由于List接口中get方法的规范的要求，异常转译是必需的：

```
/***
 * Returns the element at the specified position in this list.
 * @throws IndexOutOfBoundsException if the index is out of range
 *         (index < 0 || index >= size()).
 */
public Object get(int index) {
    ListIterator i = listIterator(index);
    try {
        return i.next();
    } catch(NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

一种特殊形式的异常转译被称为异常链接（exception chaining），如果低层的异常对于调试该异常被抛出的情形非常有帮助，那么使用异常链接是很合适的。在这种方法中，低层的异常被高层的异常保存起来，并且高层的异常提供一个公有的访问方法来获得低层的异常：

```
// Exception Chaining
try {
    // Use lower-level abstraction to do our bidding
    ...
} catch (LowerLevelException e) {
    throw new HigherLevelException(e);
}
```

从1.4发行版本开始，异常链接可通过**Throwable**来获得支持。如果你的目标平台是1.4版本或更高，那么，你可以利用这种支持，做法很简单，只要让你的高层异常的构造函数链接到**Throwable(Throwable t)**即可：

```
// Exception chaining in release 1.4
HigherLevelException(Throwable t) {
    super(t);
}
```

如果你的目标平台是一个早期的版本，那么你的异常必须把低层的异常保存下来，并且提供一个访问方法：

```
// Exception chaining prior to release 1.4
private Throwable cause;

HigherLevelException(Throwable t) {
    cause = t;
}

public Throwable getCause() {
    return cause;
}
```

通过将访问方法命名为**getCause**，并使用上面显示的声明，你可以确保你的异常将能够与Java平台上的链接设施相互操作，就像使用1.4发行版本中的异常一样。这种做法的好处是，可以以标准的方式把低层异常的栈轨迹集成到高层异常中，而且，这样做也允许标准的调试工具能够访问低层的异常。
179

尽管异常转译比不加选择地传递低层异常的做法有所改进，但是它也不能被滥用。如果可能的话，处理来自低层异常的最好做法是，在调用低层方法之前确保它们会成功执行，从而避免它们会抛出异常。有时候，你可以在给低层传递实参之前，显式地检查这些实参的有效性，从而避免低层方法会抛出异常。

如果无法阻止来自低层的异常，那么，其次的做法是，让高层来处理这些异常，从而将高层方法的调用者与低层的问题隔离开。在这种情况下，用某种适当的记录设施（比如1.4发行版本中引入的**java.util.logging**）将低层的异常记录下来可能是很合适的。这使得管理员可以调查问题，同时将客户代码和最终用户与问题隔离开。

如果既不能阻止来自低层的异常，也无法将它们与高层隔离开，那么一般的做法是使用异常转译。只有在低层方法的规范碰巧可以保证“它所抛出的异常对于高层也是合适的”情况下，才可以将异常从低层传播到高层。
180

第44条：每个方法抛出的异常都要有文档

描述一个方法所抛出的异常，是正确使用这个方法所需文档的重要组成部分。因此，花点时间仔细地为每个方法抛出的异常做文档是特别重要的。

总是要单独地声明被检查的异常，并且利用Javadoc的@throws标记，准确地记录下每个异常被抛出的条件。如果一个方法可能会抛出多个异常类，则不要使用“快捷方式”：即声明它会抛出这些异常类的某个超类。作为一个极端的例子，永远不要声明一个方法“throws Exception”，或者更差的做法，“throws Throwable”。这样的声明不仅没有为程序员提供关于“这个方法能够抛出哪些异常”的任何指导信息，而且大大地妨碍了该方法的使用，因为它实际上掩盖了在同样的执行环境中该方法可能会抛出的任何其他异常。

虽然Java语言本身并不要求程序员为一个方法声明它可能会抛出的未被检查的异常，但是，如同被检查的异常一样，仔细地为它们做文档是非常明智的。一般地，未被检查的异常代表了编程上的错误（见第40条），让程序员熟悉这些错误有助于让他们避免犯这样的错误。对于一个方法可能会抛出的未被检查的异常，如果将这些异常信息很好地组织成一个列表文档，则可以有效地描述出这个方法被成功执行的前提条件（*precondition*）。每个方法的文档应该描述它的前提条件，这是很重要的，在文档中记录下未被检查的异常是描述前提条件的最佳做法。

对于接口中的方法，在文档中记录下它可能会抛出的未被检查的异常显得尤为重要，这份文档是该接口的通用约定（*general contract*）的一部分，它指定了该接口的多个实现必须遵循的公共行为。

使用Javadoc的@throws标签记录下一个方法可能会抛出的每个未被检查的异常，但是不要使用throws关键字将未被检查的异常包含在方法的声明中。使用你的API的程序员必须知道哪些异常是需要检查的，哪些是不需要检查的，这是很重要的，因为他们有责任区别对待这两种情形。当缺少由throws声明产生的方法头时，由Javadoc@throws标签产生的文档会提供明显的提示信息，以帮助程序员区分被检查的异常和未被检查的异常。

[181]

应该注意到，为一个方法所抛出的所有未被检查的异常做文档是非常理想的，但是在实践中并不总是可以做到。当一个类被修订之后，如果有一个导出方法被修改了，它将会抛出额外的未被检查的异常，那么这不算违反源代码或者二进制兼容性。假设一个类调用了另一个独立类中的一个方法。第一个类的编写者可能会为每个方法抛出的未被检查的异常仔细地做文档，但是，如果第二个类被修订了，抛出了额外的未被检查的异常，那么，很有可能第一个类（它并没有被修订）将会把新的未被检查的异常传播出去，尽管它并没有声明这些异常。

如果一个类中的许多方法出于同样的原因而抛出同一个异常，那么在该类的文档注释中对这个异常做文档，而不是为每个方法单独做文档，这是可以接受的。一个常见的例子是NullPointerException。如果一个类的文档注释中有这样的描述“如果一个null对象引用被传递到任何一个参数中，那么这个类中的所有方法都会抛出NullPointerException”，或者[182]其他类似的语句，则这种做法是很合适的。

第45条 在细节消息中包含失败 - 捕获信息

当一个程序由于一个未被捕获的异常而失败的时候，系统会自动地打印出该异常的栈轨迹。在栈轨迹中包含该异常的字符串表示 (*string representation*)，即它的 `toString` 方法的结果。典型情况下它包含该异常的类名，以及紧随其后的细节消息 (*detail message*)。大多数情况下，这是程序员或者域服务人员 (field service personnel, 指检查软件失败的人) 必须要检查的信息。如果失败的情形不容易重现的话，要想获得更多的信息会非常困难，甚至是不可能的。因此，异常类型的 `toString` 方法应该尽可能多地返回有关失败原因的信息，这是特别重要的。换句话说，一个异常的字符串表示应该捕获失败，以便于以后的分析。

为了捕获失败，一个异常的字符串表示应该包含所有“对该异常有贡献”的参数和域的值。例如，`IndexOutOfBoundsException` 异常的细节消息应该包含下界、上界，以及没有落在其中的实际下标值。该细节消息提供了许多关于失败的信息。这三个值中任何一个都有可能是错的。实际的下标值可能小于下界或等于上界（一个“越界错误”），或者它可能是一个无效值，太小或太大。下界也有可能大于上界（一个严重违反内部约束条件的案例）。每一种情形都代表了不同的问题，如果程序员知道应该去寻找哪一种错误，则可以极大地有助于诊断过程。

虽然在一个异常的字符串表示中包含所有相关的“硬数据 (hard date)”是非常重要的，但是包含大量的描述信息往往是没有价值的。栈轨迹的用途是与源文件结合起来进行分析，它通常包含该异常被抛出点的确切文件和行数，以及栈中所有其他方法调用所在的文件和行数。关于失败的冗长描述信息通常是不必要的，这些信息可以通过阅读源代码而获得。

异常的字符串表示不应该与“针对用户层次的错误消息”混为一谈，后者对于最终用户而言必须是可理解的。与用户层次的错误消息不同，异常的字符串表示主要是针对程序员或者域服务人员的，用于分析失败的原因。因此，对它来讲，内容比可理解性要重要得多。

为了确保在异常的字符串表示中包含足够的失败 - 捕捉信息 (failure-capture information)，一种办法是在异常的构造函数中以参数形式引入这些信息。然后，有了这些信息，只要把它们放到消息描述中，就可以自动产生消息细节描述。例如，`IndexOutOfBoundsException` 并不是有一个 `String` 构造函数，而是有一个这样的构造函数：

```
/*
 * Construct an IndexOutOfBoundsException.
 *
 * @param lowerBound the lowest legal index value.
 * @param upperBound the highest legal index value plus one.
 * @param index      the actual index value.
 */
```

```
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                int index) {
    // Generate a detail message that captures the failure
    super("Lower bound: " + lowerBound +
          ", Upper bound: " + upperBound +
          ", Index: " + index);
}
```

不幸的是，Java平台库并没有广泛地使用这种做法，但是，这种做法仍然值得推荐。它使得程序员更加易于抛出异常以捕获失败。实际上，这种做法使程序员不想捕获失败也不容易！这种做法可以有效地把代码集中起来放在异常类中，由这些代码产生一个针对特定异常的高质量字符串表示，而不是产生雷同的描述信息。

正如第40条中所建议的，为一个异常的失败-捕获信息（在上述例子中为lowerBound、upperBound和index）提供一些访问方法是合适的。提供这样的访问方法对于被检查的异常，比对于未被检查的异常更为重要，因为失败-捕获信息对于从失败中恢复是非常有用的。程序员希望通过程序的手段来访问一个未被检查异常的细节是很少见的（尽管也是可以想像得到的）。然而，即使对于未被检查的异常，作为一般原则提供这些访问方法也是明智的（见

[184] 第9条最后一段）。

第46条：努力使失败保持原子性

当一个对象抛出一个异常之后，我们总是期望这个对象仍然保持在一种定义良好的可用状态之中，即使失败发生在执行某个操作的过程中间。对于被检查的异常而言，这尤为重要，因为调用者会期望能从这种异常中恢复过来。一般而言，一个失败的方法调用应该使对象保持“它在被调用之前的状态”。具有这种属性的方法被称为具有失败原子性 (*failure atomic*)。

有几种途径可以获得这种效果。最简单的办法莫过于设计一个非可变的对象（见第13条）。如果一个对象是非可变的，那么失败原子性是显然的。如果一个操作失败了，它可能会阻止创建新的对象，但是永远也不会使已有的对象保持在不一致的状态中，因为当每个对象被创建之后它就处于一致的状态中，以后不会再发生变化。

对于在可变对象上执行操作的方法，获得失败原子性最常见的办法是，在执行操作之前检查参数的有效性（见第23条）。这可以使得在对象的状态被修改之前，适当的异常首先被抛出来。例如，考虑第5条目中的`Stack.pop`方法：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

如果初始的大小 (`size`) 检查被去掉的话，当这个方法企图从一个空栈中弹出元素时，它仍然会抛出一个异常。然而，这将会导致 `size` 域保持在不一致的状态（负数）中，从而使得将来对该对象的任何方法调用都会失败。而且，那时候，`pop` 方法抛出的异常对于该 `Stack` 抽象来说也是不恰当的（见第43条）。

一种类似的获得失败原子性的办法是，对计算处理过程调整顺序，使得任何可能会失败的计算部分都发生在对象状态被修改之前。如果对实参的检查只有在执行了一部分计算之后才能进行的话，那么这种办法实际上是上一种办法的自然扩展。例如，考虑 `TreeMap` 的情形，它的元素被按照某种特定的顺序做了排序。为了向 `TreeMap` 中添加一个元素，该元素的类型必须可通过 `TreeMap` 的排序准则与其他的元素进行比较。企图增加一个类型不正确的元素将会导致 `ClassCastException` 异常。所以，本例子的情形是，在 `TreeMap` 被以任何方式修改之前，在 `TreeMap` 中搜索被添加的元素就可能会导致 `ClassCastException` 异常。

185

第三种获得失败原子性的办法没有那么常用，做法是编写一段恢复代码 (*recovery code*)，由它来解释操作过程中发生的失败，以及使对象回滚到操作开始之前的状态上。这种办法主

要用于永久性的数据结构。

最后一种获得失败原子性的办法是，在对象的一份临时拷贝上执行操作，当操作完成之后再把临时拷贝中的结果复制给原来的对象。如果数据被保存在临时的数据结构中，计算过程会更加快速，那么这种办法非常适用。例如，`Collections.sort`在执行排序之前，首先把它的输入列表转储到一个数组中，以便降低在排序的内循环中访问元素所需要的开销。这是出于性能原因的做法，但是，作为一个附加的好处，它保证了即使排序失败，输入列表将会保持原样。

虽然失败原子性总是期望的目标，但它并不总是可以做得到。例如，如果两个线程企图在没有适当的同步机制的情况下，并发地修改同一个对象，那么对象就有可能被留在不一致的状态中。因此，在捕获了一个`ConcurrentModificationException`异常之后再假设对象仍然是可用的，这是不正确的。错误（相对于异常）通常是不可恢复的，当一个方法抛出错误时，它不需要保持失败原子性。

即使在可以实现失败原子性的场合，它也并不总是所期望的。对于某些操作，它会显著地增加开销或者复杂性。然而，一旦你知道了问题所在，那么获得失败原子性往往是简单而容易的。总结一条规则：作为方法规范的一部分，任何一个异常都不应该改变对象调用该方法之前的状态。如果这条规则被违反，则API文档应该清楚地指明对象将会处于什么样的状态。不幸的是，大量现有的API文档都未能做到这一点。

第47条：不要忽略异常

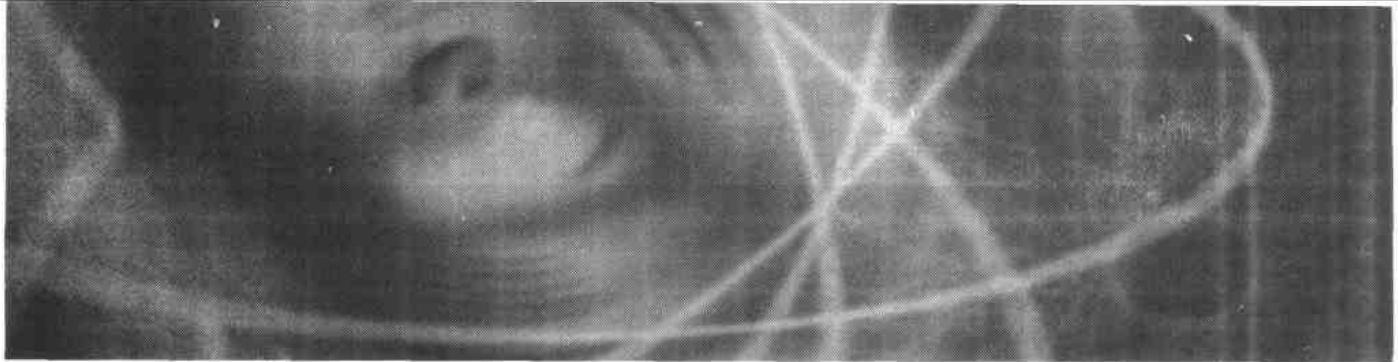
尽管这条建议看上去是显而易见的，但是它却常常被违反，因而值得再次提出来。当一个API的设计者声明一个方法将会抛出某个异常的时候，他们正在试图说明某些事情。所以，请不要忽略它！要忽略一个异常非常容易，只需将方法调用用一个try语句包围起来，并且包含一个空的catch块，如下所示：

```
// Empty catch block ignores exception - Highly suspect!
try {
    ...
} catch (SomeException e) {
```

空的catch块会使异常达不到应有的目的，异常的目的是强迫你处理不正常的条件。忽略一个异常就如同忽略一个火警信号一样——若把火警信号器关掉了，那么当真正的火灾发生时，就没有人能看到火警信号了。或许你会侥幸逃过劫难，或许结果将是灾难性的。至少catch块也应该包含一条说明，用来解释为什么忽略掉这个异常是合适的。

一个可以忽略异常的例子是动画中的多帧图像连续播放。如果屏幕会在一个规则的时间间隔内被持续地更新，那么处理一个临时错误的最好办法是忽略这个错误，并等待下一次更新时刻到来。

本条目中的建议同样适用于被检查的异常和未被检查的异常。不管一个异常代表了一个可预见的例外条件，还是一个程序错误，用一个空的catch块忽略它将会导致程序在遇到错误的情况下悄然地执行下去。然后，有可能在将来的某个点上，当程序不能再容忍错误源带来的问题时，它就会失败。正确地处理异常能够避免无可挽回的失败。简单地将一个未被检查的异常传播给外界至少会使程序迅速地失败，从而保留了有助于调试该失败条件的信息。



第9章

线 程

通过使用线程（`thread`），可以在同一个程序中同时进行多个活动。多线程程序设计比单线程程序设计要困难得多，所以，第30条中的建议在这里特别合适：如果一个库中的类能够帮助你从低层的多线程程序设计中解脱出来，那么一定要使用这个类。`java.util.Timer`就是一个例子，Doug Lea的util.concurrent包[Lea01]是一个高层的线程工具集合。即使你在适当的地方使用了这样的库，你有时候仍然要编写或者维护多线程代码。本章包含的建议可以帮助你编写出清晰、正确、文档组织良好的多线程程序。

第48条：对共享可变数据的同步访问

`synchronized`关键字可以保证在同一时刻，只有一个线程在执行一条语句，或者一段代码块。许多程序员把同步的概念仅仅理解为一种互斥的方式，即，当一个对象被一个线程修改的时候，可以阻止另一个线程观察到内部不一致的状态。按照这种观点，对象被创建的时候处于一致的状态（见第13条），当有方法访问它的时候它就被锁定了。这些方法观察到对象的状态，并且可能会引起一个状态转变（*state transition*），即把对象从一种一致的状态转换到另一种一致的状态。正确地使用同步可以保证其他任何方法都不会看到对象处于不一致的状态中。

这种观点是正确的，但是它并没有说明同步的全部意义。同步不仅可以阻止一个线程看到对象处于不一致的状态中，它还可以保证通过一系列看似顺序执行的状态转变序列，对象从一种一致的状态变迁到另一种一致的状态。每一个线程进入到一个被同步的方法或者代码块的时候，它会看到由同一个锁控制的以前所有状态转变的结果。当线程退出了这个被同步的区域之后，任何线程在进入到由这同一把锁同步的区域时，它就可以看到由前面那个线程带来的状态转变（如果有的话）。

Java语言保证读或者写一个变量是原子的（*atomic*），除非这个变量的类型为`long`或

`double` 换句话说，读入一个非`long`或`double`类型的变量，可以保证返回的值一定是某个线程保存在该变量中的，即使多个线程在没有同步的情况下并发地修改这个变量，也是如此。

你可能听说过，为了提高性能，在读或写原子数据的时候，你应该避免使用同步。这个建议是非常危险而错误的。虽然原子性保证了一个线程在读取原子数据的时候，不会看到一个随机的数值，但是它并不保证一个线程写入的值对于另一个线程将是可见的：为了在线程之间可靠地通信，以及为了互斥访问，同步是需要的。这是Java程序设计语言的一项技术的必然结果，即Java的内存模型（*memory model*）[JLS, 17]。虽然Java的内存模型在将来的发行版本中可能会有实质性的修订[Pugh01a]，但是几乎可以肯定，这个事实不会改变。

如果对一个共享变量的访问不能同步的话，其结果将是非常可怕的，即使这个变量是原子可读写的。考虑下面的序列号生成工具：

```
// Broken - requires synchronization!
private static int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

这个生成工具的意图是保证每次调用`generateSerialNumber`都会返回一个不同的序列号，只要不超过 2^{32} 次调用就行。它并不需要用同步来保护序列号生成器的约束条件，因为它没有约束条件；它的状态只包含一个原子可写的域（`nextSerialNumber`），这个域所有可能的值都是合法的。然而，如果没有同步，这个方法并不能正确地工作。递增操作符（`++`）既要读`nextSerialNumber`域，也要写`nextSerialNumber`域，所以它不是原子的。读和写是相互独立的操作，按顺序执行。因此，多个并发的线程可能会看到`nextSerialNumber`域中有同样的值，因而返回相同的序列号。

190

更加令人惊奇的是，一个线程重复地调用`generateSerialNumber`，获得从0到n的一系列序列号，之后，另一个线程调用`generateSerialNumber`并获得一个序列号0，这是有可能发生的。如果没有同步机制的话，第二个线程可能根本看不到第一个线程所作的改变。这正是前面提到的内存模型所导致的结果。

修正`generateSerialNumber`方法是非常简单的，只要在它的声明中增加`synchronized`修饰符即可。这就确保了多个调用不会相互插入，并且每个调用都可以看到所有以前的调用结果。为了保护这个方法，使用`long`来代替`int`，或者当`nextSerialNumber`将要回转到0时抛出一个异常，这两种做法都是明智的。

接下去，考虑如何终止一个线程。虽然Java平台提供了一些方法用来主动地终止一个线程，

但是这些方法不值得提倡使用，因为它们本质上都是不安全的（*unsafe*）——它们会导致对象被破坏。为了终止一个线程，一种推荐的做法非常简单，只要让线程轮询（poll）某个域，该域的值如果发生变化，就表明此线程应该终止自己。通常这个域是一个boolean，或者一个对象引用。因为读或者写这样的域是原子操作，所以有些程序员在访问该域的时候就不再使用同步。因此，像下面这样的代码并不少见：

```
// Broken - requires synchronization!
public class StoppableThread extends Thread {
    private boolean stopRequested = false;

    public void run() {
        boolean done = false;

        while (!stopRequested && !done) {
            ... // do what needs to be done.
        }
    }

    public void requestStop() {
        stopRequested = true;
    }
}
```

这段代码的问题在于，由于缺少同步，所以并不能保证这个可终止的线程将会“看到”其他线程对stopRequested的值所做的改变。其结果是，requestStop方法有可能完全无效。
[19] 除非你是在一台多处理器的机器上运行，否则你在实践中不可能观察到这样的错误行为，但这是没有保证的。修正这个问题最直接的办法是，对stopRequested域的所有访问都加上同步特性：

```
// Properly synchronized cooperative thread termination
public class StoppableThread extends Thread {
    private boolean stopRequested = false;

    public void run() {
        boolean done = false;

        while (!stopRequested() && !done) {
            ... // do what needs to be done.
        }
    }

    public synchronized void requestStop() {
        stopRequested = true;
    }

    private synchronized boolean stopRequested() {
        return stopRequested;
    }
}
```

注意，这里每一个被同步的方法中的动作都是原子的：使用同步的唯一目的是为了通信，

而不是为了互斥访问。很显然，修正之后的代码可以正常工作，while循环的每次迭代中的同步开销并不显著。也就是说，有一个正确的解决方案，其代码并不冗长、性能也比较好。如果stopRequested被声明为volatile的话，则同步可以被省略。volatile修饰符可以保证任何一个线程在读取一个域的时候都将会看到最近刚刚被写入的值。

192

在前一个例子中，如果对stopRequested的访问不做同步，则造成的后果相对而言还不算严重；requestStop方法的作用可能会被无限延迟。如果对可变共享数据的访问不作同步，则造成的后果就要严重得多。考虑下面的用于迟缓初始化（lazy initialization）的双重检查模式（double-check idiom）：

```
// The double-check idiom for lazy initialization - broken!
private static Foo foo = null;

public static Foo getFoo() {
    if (foo == null) {
        synchronized (Foo.class) {
            if (foo == null)
                foo = new Foo();
        }
    }
    return foo;
}
```

该模式背后的思想是，在域（foo）被初始化之后，再要访问该域时无需同步，从而避免多数情形下的同步开销。同步只是被用来避免多个线程对该域做初始化。这种模式保证了该域将至多被初始化一次，所有调用getFoo的线程都将会得到正确的对象引用值。不幸的是，该对象引用并不能保证可以正常地工作。如果一个线程在不使用同步的情况下读入该引用，并调用被引用的对象上的方法，那么这个方法可能会看到对象被部分初始化的状态，从而导致灾难性的失败。

一个线程看到一个被迟缓构建的对象处于部分初始化的状态，这种情形并不直观。在将引用“发布”到一个域（foo，其他的线程将从这个域读取它的值）中之前，对象应该被完整地构造出来。但是，在缺少同步的情况下，读入一个已“发布”的对象引用并不保证：一个线程将会看到在对象引用发布之前所有保存在内存中的数据。特别是，读入一个已发布的对象引用并不保证：读线程将会看到被引用的对象内部的最新数据值。**一般情况下，双重检查模式并不能正确地工作，但是如果被共享的变量包含一个原语值，而不是一个对象引用，则它可以正确地工作[Pugh01b]。**

193

下面有几种办法来修正这个问题。最容易的办法是完全省去迟缓初始化：

```
// Normal static initialization (not lazy)
private static final Foo foo = new Foo();
```

```
public static Foo getFoo() {
    return foo;
}
```

这种做法当然可以工作，而且getFoo方法会执行得最快。它不用同步，也不用计算。正如第37条中讨论的那样，你可以编写出简单、清晰、正确的程序，把优化的任务放到最后，只有当性能测试表明有必要做优化的时候才做优化。因此，省去迟缓初始化往往是最好的解决办法。如果你省去了迟缓初始化，测量程序的开销，发现省去迟缓初始化行不通，则其次最好的办法是使用正确的同步方法来执行迟缓初始化：

```
// Properly synchronized lazy initialization
private static Foo foo = null;

public static synchronized Foo getFoo() {
    if (foo == null)
        foo = new Foo();
    return foo;
}
```

这个方法可以保证正常工作，但是它会招致在每个调用上的同步开销。在现代的JVM实现上，这份开销相对比较小。然而，如果你测量了系统的性能之后，你觉得既不能承受正常初始化的开销，也不能承受同步每个访问而招致的开销，那么还有另外一个选择。如果一个静态域的初始化非常昂贵，并且它也不见得会被用到，但一旦需要则会被充分地使用，那么，在这样的情况下，按需初始化容器类（*initialize-on-demand holder class*）模式是非常合适的。下面的代码演示了这种模式：

```
// The initialize-on-demand holder class idiom
private static class FooHolder {
    static final Foo foo = new Foo();
}

public static Foo getFoo() { return FooHolder.foo; }
```

194

该模式充分利用了Java语言中“只有当一个类被用到的时候它才被初始化”[JLS, 12.4.1]。当getFoo方法第一次被调用的时候，它读入FooHolder.foo域，使得FooHolder类被初始化。该模式的优美之处在于，getFoo方法并没有被同步，它只执行一次域访问，所以迟缓初始化并没有引入实际的访问开销。这种模式的缺点在于，它不能用于实例域，只能用于静态域。

简而言之，无论何时当多个线程共享可变数据的时候，每个读或者写数据的线程必须获得一把锁。不要由于原子读和写而妨碍你执行正确的同步。如果没有同步，则一个线程所做的修改就无法保证被另一个线程观察到。未被同步的数据访问会造成程序的活性失败（liveness failure）和安全性失败（safety failure）。而且这样的失败将会难以重现。它们可能

是时间相关的，可能会高度依赖于JVM实现的细节，以及所运行的硬件平台。

在某些特定的条件下，使用`volatile`修饰符可以提供另一种不同于普通同步机制的选择。但这是一项高级的技术。而且，由于当前正在进行之中的内存模型尚未完成，所以，这项技术的适用范围还不得而知。

[195]

第49条：避免过多的同步

第48条警告我们缺少同步的危险性，本条目关注相反的问题。依据情况的不同，过多的同步可能会导致性能降低、死锁，甚至不确定的行为。

为了避免死锁的危险，在一个被同步的方法或者代码块中，永远不要放弃对客户的控制。换句话说，在一个被同步的区域内部，不要调用一个可被改写的公有或受保护的方法（这样的方法往往是抽象的，但偶尔它们也会有一个具体的默认实现）。从包含该同步区域的类的角度来看，这样的方法是一个外来者（*alien*）。这个类不知道该方法会做什么事情，也控制不了它。客户可以为这个外来方法提供一个实现，并且在该方法中创建另一个线程，再回调到这个类中。然后，新建的线程试图获取原线程所拥有的那把锁，这样会导致新建的线程被阻塞。如果创建该线程的方法正在等待这个线程完成任务，则死锁就形成了。

为了使这个过程更为具体化，考虑这样的类，它实现了一个工作队列（*work queue*）。该类允许客户将工作项目（*work item*）排入队列，以便进行异步方式的处理。`enqueue`方法可能会被经常调用到。构造函数启动了一个后台线程，该后台线程按照项目进入队列的顺序从中移除项目，并调用`processItem`方法对它们进行处理。当工作队列不再被需要时，客户调用`stop`方法，请求该线程在完成任何正在进行之中的项目之后，温和地终止。

```

public abstract class WorkQueue {
    private final List queue = new LinkedList();
    private boolean stopped = false;

    protected WorkQueue() { new WorkerThread().start(); }

    public final void enqueue(Object workItem) {
        synchronized (queue) {
            queue.add(workItem);
            queue.notify();
        }
    }

    public final void stop() {
        synchronized (queue) {
            stopped = true;
            queue.notify();
        }
    }

    protected abstract void processItem(Object workItem)
        throws InterruptedException;

    // Broken - invokes alien method from synchronized block!
    private class WorkerThread extends Thread {
        public void run() {
            while (true) { // Main loop
                synchronized (queue) {
                    try {

```

```
        while (queue.isEmpty() && !stopped)
            queue.wait();
    } catch (InterruptedException e) {
        return;
    }

    if (stopped)
        return;

    Object workItem = queue.remove(0);
    try {
        processItem(workItem); // Lock held!
    } catch (InterruptedException e) {
        return;
    }
}
}
```

为了使用这个类，你必须对它子类化，为抽象方法processItem提供一个具体的实现。例如，下面的子类打印出每一个工作项目，不管这些项目进入队列的频率多高，它每秒钟打印不超过一个项目：

```
class DisplayQueue extends WorkQueue {  
    protected void processItem(Object workItem)  
        throws InterruptedException {  
        System.out.println(workItem);  
        Thread.sleep(1000);  
    }  
}
```

[197]

因为WorkQueue类从一个被同步的块内调用了抽象方法processItem，所以这有可能会导致死锁。下面的子类利用前面提到的方法，而造成死锁：

```
class DeadlockQueue extends WorkQueue {  
    protected void processItem(final Object workItem)  
        throws InterruptedException {  
        // Create a new thread that returns workItem to queue  
        Thread child = new Thread() {  
            public void run() { enqueue(workItem); }  
        };  
        child.start();  
        child.join(); // Deadlock!  
    }  
}
```

这个例子是刻意编出来的，因为`processItem`方法没有理由创建一个后台线程，但是存在的问题却是真实的。从一个被同步的代码块中调用外部提供的方法从而引起死锁，这在真实的系统中也有发生，比如GUI工具箱。幸运的是，修正这个问题很容易。只需简单地把方法调用移到同步代码块的外面即可，如下所示：

```

// Alien method outside synchronized block - "Open call"
private class WorkerThread extends Thread {
    public void run() {
        while (true) { // Main loop
            Object workItem = null;
            synchronized (queue) {
                try {
                    while (queue.isEmpty() && !stopped)
                        queue.wait();
                } catch (InterruptedException e) {
                    return;
                }
                if (stopped)
                    return;
                workItem = queue.remove(0);
            }
            try {
                processItem(workItem); // No lock held
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}

```

[198] 同步区域之外被调用的外来方法被称为“开放调用 (*open call*)” [Lea00, 2.4.1.3]。除了可以避免死锁以外，开放调用还可以极大地增加并发性。外来方法的运行时间可能会任意长，如果在同步区域内调用外来方法的话，那么在外来方法执行期间，其他线程要想访问共享对象都将被不必要地拒绝。

通常，在同步区域内你应该做尽可能少的工作。获得锁，检查共享数据，根据需要转换数据，然后放掉锁。如果你必须要执行某个很耗时的动作，则应该设法把这个动作移到同步区域的外面。

如果在调用外来方法时，由同步区域保护的约束条件临时无效，则从同步区域内调用一个外来方法可能会引起比死锁更为严重的失败（这种情形在上面修正之前的工作队列例子中不会发生，因为当processItem被调用的时候，队列对象处于一致的状态）。这种失败与“在外来方法内新建线程”没有关系；当外来方法回调到这种有缺陷的类（即状态不一致）的时候，才会发生这种类型的失败。因为Java程序设计语言中的锁是递归的（*recursive*），如果这样的调用是由另一个线程发出的，则它们会死锁，但是现在它们不会死锁。调用线程已经拥有锁了，所以当它试图再次获得锁的时候，即使此时该锁保护的数据上有另一个完全不相关的操作正在进行，它也会成功。这样的失败其后果可能是灾难性的：本质上讲，该锁没有起到应有的作用。递归锁简化了多线程面向对象程序的设计和构造，但是它们可以把活性失败（*liveness failure*）变成安全性失败（*safety failure*）。

本条目的第一部分是关于并发性问题的。现在我们把注意力转移到性能上来。虽然同步的

开销自从Java平台早期开始就一直在下降，但是它永远也不会完全消失。如果一个很常用的操作被不必要的同步了，那么它对于性能会有严重的影响。例如，考虑`StringBuffer`类和`BufferedInputStream`类。这些类都是线程安全的（*thread-safe*，见第52条），但它们几乎总是被用于单个线程之中，所以它们所做的锁操作往往是不必要的。它们支持一些细粒度的（*fine-grained*）方法，可以在单个字符或者字节的层次上进行操作。所以，这些类不仅仅在做不必要的锁操作，而且还有做得太多的倾向。这会导致严重的性能损失。一篇论文指出，在实际应用中有接近20%的性能损失[Heydon99]。你可能没有看到过这种由不必要的同步造成的性能损失如此之大，但5%至10%的损失还是有可能的。

毫无疑问，这属于Knuth所说的我们不该计较的“小效率”（见第37条）。然而，如果你正在编写一个低层的抽象，它一般只被一个线程使用，或者作为大的同步对象的一个组件，那么，你应该考虑制止该类的内部同步。不管你是否决定要同步一个类，很重要的一点是，你应该在文档中说明它的线程安全属性（见第52条）。

[199]

一个给定的类是否应该执行内部同步并不总是很清楚。按照第52条的说法，一个类是否应该被做成线程安全的（*thread-safe*），或者线程兼容的（*thread-compatible*），并不总是很清楚。下面有一些指导原则可以帮助你做出选择。

如果你正在编写的类将主要被用于要求同步的环境中，同时也被用于不要求同步的环境中，那么，一个合理的方法是，同时提供同步的（线程安全的）版本，和未同步的（线程兼容的）版本。一种做法是提供一个包装类（*wrapper class*）（见第14条），它实现一个描述该类的接口，同时，在将方法调用转发给内部对象中对应的方法之前执行适当的同步操作。这正是Collections Framework采用的方法。无疑，`java.util.Random`也应该采用这种方法。第二种做法适合于那些不是被设计用来扩展或者重新实现的类，它提供一个未同步的类和一个子类，在子类中只包含一些被同步的方法，它们依次调用到超类中对应的方法上。

要在一个类的内部进行同步，一个很好的理由是因为它将被大量地并发使用，而且通过执行内部细粒度的同步操作你可以获得很高的并发性。例如，实现一个不可改变尺寸的散列表，并且它可以独立地对每个散列桶进行同步访问，这是有可能的。这样做比“锁住整个散列表以访问单个条目”的做法，可获得更高的并发性。

如果一个类或者一个静态方法依赖于一个可变的静态域，那么它必须要在内部进行同步，即使它往往只用于单个线程。与共享实例不同，这种情况下，对于客户要执行外部同步是不可能的，因为不可能保证其他的客户也会执行外部同步。静态方法`Math.random`就是这样一个例子。

简而言之，为了避免死锁和数据破坏，千万不要从同步区域内部调用外来方法。更为一般

地，请尽量限制同步区域内部的工作量。当你在设计一个可变类的时候，请考虑一下它们是否需要自己完成同步操作。因省去同步而节省下来的开销不会很大，但也是可测量的。当你决定了一个抽象（比如类）是否将被主要用于多线程环境之后，你应该将这个决定清楚地写到文档中。
[200]

第50条：永远不要在循环的外面调用wait

`Object.wait`方法的作用是使一个线程等待某个条件。它一定是在一个同步区域中被调用的，而且该同步区域锁住了被调用的对象。下面是使用`wait`方法的标准模式：

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    ...
    // Perform action appropriate to condition
}
```

总是使用`wait`循环模式来调用`wait`方法。永远不要在循环的外面调用`wait`。循环被用于在等待的前后测试条件。

在等待之前测试条件，如果条件已经成立的话则跳过等待，这对于确保活性（*liveness*）是必要的。如果条件已经成立，并且在线程等待之前`notify`（或者`notifyAll`）方法已经被调用过，则无法保证该线程将总会从等待中醒过来。

在等待之后测试条件，如果条件不成立的话继续等待，这对于确保安全性（*safety*）是必要的。当条件不成立的时候，如果线程继续执行，则可能会破坏被锁保护的约束关系。当条件不成立时，有下面一些理由可使一个线程醒过来：

- 另一个线程可能得到了锁，并且在一个线程调用`notify`的时刻，到等待线程醒过来的时刻之间，得到锁的线程已经改变了被保护的状态。
- 条件并没有成立，但是另一个线程可能意外地或恶意地调用了`notify`。在公有可访问的对象上等待，这些类实际上把自己暴露在危险的境地中。在一个公有可访问对象的同步方法中包含的`wait`都会出现这样的问题。
- 通知线程（*notifying thread*）在唤醒等待线程时可能会过度“大方”。例如，即使只有某些等待线程的条件已经被满足，但是通知线程仍必须调用`notifyAll`。
- 在没有被通知的情况下等待线程也可能会醒过来。这被称为“伪唤醒（*spurious wakeup*）”。虽然《The Java Language Specification》[JLS]并没有提到这种可能性，但是许多JVM实现都使用了具有伪唤醒功能的线程设施，尽管用得很少[Posix, 11.4.3.6.1]。 [201]

一个相关的话题是，为了唤醒正在等待的线程，你应该使用`notify`还是`notifyAll`（请回忆一下，`notify`唤醒一个正在等待的线程，如果这样的线程存在的话，而`notifyAll`唤醒所有正在等待的线程）。一种常见的说法是，你总是应该使用`notifyAll`。

假设所有的wait调用都在while循环的内部，那么这是合理而保守的建议。它总会产生正确的结果，因为它可以保证你将会唤醒所有需要被唤醒的线程。你可能也会唤醒其他一些线程，但是这不会影响程序的正确性。这些线程醒来之后会检查它们正在等待的条件，发现条件并不满足，就会继续等待。

作为一种优化，如果处于等待状态的所有线程都在等待同一个条件，而每次只有一个线程可以从这个条件中被唤醒，那么你可以选择调用notify而不是notifyAll。如果只有一个线程在一个特定的对象上等待，那么这两个条件很容易被满足（见第49条中的WorkQueue例子）。

即使这些条件都是真的，也许还是有理由使用notifyAll而不是notify。就好像把wait调用放在一个循环中，以避免在公有可访问的对象上的意外或恶意的通知，与此类似，使用notifyAll代替notify可以避免来自不相关线程的意外或恶意的等待。否则的话，这样的等待会“吞掉”一个关键的通知，使真正的接收线程无限地等待下去。在WorkQueue例子中没有使用notifyAll的原因是，因为辅助线程（worker thread）在一个私有的（private）对象（queue）上等待，所以这里不存在意外或者恶意地等待的危险。

关于使用notifyAll优先于notify的建议有一个告诫：虽然使用notifyAll不会影响正确性，但是会影响性能。实际上，从系统角度来看，它会使某些数据结构的性能，从等待线程数的线性级退化到平方级。这一类数据结构之所以受到如此影响，是因为在任何给定时刻，只有特定数量的线程才允许获得某种特定的状态，其他线程必须等待。这样的例子包括信号量（semaphore）、有界缓冲区（bounded buffer）和读写锁（read-write lock）。

如果你正在实现这种数据结构，并且当它达到“特定状态”时你会唤醒每一个线程，那么，你每次唤醒一个线程，总共n次唤醒。如果你唤醒所有这n个线程，但只有一个线程可以获得特定的状态，剩下的n-1个线程回去继续等待，那么，为了所有这些等待线程获得特定的状态，你最终需要 $n + (n-1) + (n-2) + \dots + 1$ 次唤醒。此序列的总和为 $O(n^2)$ 。如果你知道线程的数量总是比较小，那么在实践中这不会成为问题，但是如果你没有这样的保证，那么，使用一个精心选择的唤醒策略是非常重要的。

202 如果竞争某个特定状态的所有线程在逻辑上是等价的，那么，你必须要谨慎地使用notify，而不是notifyAll。然而，如果在给定的时刻，只有某些等待线程才允许获得特定的状态，那么你必须要使用一种被称为“特殊通知（Specific Notification）”的模式[Cargill96, Lea99]。对该模式的讨论超出了本书的范围。

简而言之，总是在一个while循环中调用wait，并且使用标准的模式。你没有理由不这样做。一般情况下，你应该使用notifyAll优先于notify。然而，在有些情况下这样做会导致实质性的性能负担。如果使用notify，请一定要小心，以确保程序的活性（liveness）。

第51条：不要依赖于线程调度器

当有多个线程可以运行时，线程调度器（thread scheduler）决定哪个线程将会运行，以及运行多长时间。任何一个合理的JVM实现在作出这样的决定的时候，都努力做到某种公正性，但是对于不同的JVM实现，其策略大相径庭。因此，一个实现良好的多线程程序不应该依赖于此策略的细节。任何依赖于线程调度器而达到正确性或性能要求的程序，很有可能是不可移植的。

编写健壮的、响应良好的、可移植的多线程应用程序的最好办法是，尽可能确保在任何给定时刻只有少量的可运行线程。这使得线程调度器没有更多的选择：它只需运行这些可运行的线程，直到它们不再可运行为止。这样做的结果是，即使在差异很大的线程调度算法下，这些程序的行为也不会有很大的变化。

保持可运行线程数量尽可能少的主要技术是，让每个线程做少量的工作，然后使用Object.wait等待某个条件发生，或者使用Thread.sleep睡眠一段时间。线程不应该忙—等（*busy-wait*），即反复地检查一个数据结构，以等待某些事情发生。除了使程序易受到调度器的变化的影响之外，忙—等这种做法也会增加处理器的负担，降低了同一机器上其他进程可以完成的有用工作量。

第49条中的工作队列例子符合这些建议：假设客户提供的processItem方法是行为良好的，则辅助线程（worker thread）花费大部分的时间在一个监视器（monitor）上等待，等待队列变成非空的。作为一个极端的反面例子，考虑下面重新实现的WorkQueue，它没有使用监视器，而是使用忙—等：

```
// HORRIBLE PROGRAM - uses busy-wait instead of Object.wait!
public abstract class WorkQueue {
    private final List queue = new LinkedList();
    private boolean stopped = false;

    protected WorkQueue() { new WorkerThread().start(); }

    public final void enqueue(Object workItem) {
        synchronized (queue) { queue.add(workItem); }
    }
    public final void stop() {
        synchronized (queue) { stopped = true; }
    }
    protected abstract void processItem(Object workItem)
        throws InterruptedException;
    private class WorkerThread extends Thread {
        public void run() {
            final Object QUEUE_IS_EMPTY = new Object();
            while (true) { // Main loop
                Object workItem = QUEUE_IS_EMPTY;
```

```

        synchronized (queue) {
            if (stopped)
                return;
            if (!queue.isEmpty())
                workItem = queue.remove(0);
        }

        if (workItem != QUEUE_IS_EMPTY) {
            try {
                processItem(workItem);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}

```

为了让你认识到这种实现所需要付出的代价，请考虑下面的小型基准测试程序，它创建了两个工作队列，并且在它们之间来回传递一个工作项目（从一个工作队列传递到另一个队列的工作项目是一个指向前一个队列的引用，被用做一种回程地址）。在开始测量之前，程序运行10秒钟，作为系统的“热身”运动，然后在下一个10秒钟内，对队列之间来回传递的过程进行计数。在我的机器上，第49条中最终版本的WorkQueue显示出每秒23 000次来回传递，而上面这种不恰当的实现只有每秒17次来回传递：

```

class PingPongQueue extends WorkQueue {
    volatile int count = 0;

    protected void processItem(final Object sender) {
        count++;
        WorkQueue recipient = (WorkQueue) sender;
        recipient.enqueue(this);
    }
}

public class WaitQueuePerf {
    public static void main(String[] args) {
        PingPongQueue q1 = new PingPongQueue();
        PingPongQueue q2 = new PingPongQueue();
        q1.enqueue(q2); // Kick-start the system

        // Give the system 10 seconds to warm up
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
        }

        // Measure the number of round trips in 10 seconds
        int count = q1.count;
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
        }
        System.out.println(q1.count - count);
    }
}

```

```

        q1.stop();
        q2.stop();
    }
}

```

虽然上面的`workQueue`实现可能看起来有点牵强，但是，在多线程系统中有一个或者多个不必要的可运行线程并不少见。其结果可能并不像上面例子中演示的那样极端，但是性能和可移植性可能会受到损害。

如果一个程序因为某些线程无法像其他的线程那样获得足够的CPU时间，而不能工作，那么，不要企图通过调用`Thread.yield`来“修正”该程序。你可能会成功地让程序工作，但是从性能的角度来看，这样得到的程序是不可移植的。同一个`yield`调用在一个JVM实现上能提高性能，而在另一个JVM实现上可能会更差，在第三个JVM实现上可能没有影响。`Thread.yield`没有可测试语义（testable semantic）。更好的解决办法是重新构造应用程序，以减少并发可运行线程的数量。

一项相关的技术是调整线程优先级（*thread priorities*），也可以算是一条建议。线程优先级是Java平台上最不可移植的特征了。通过调整某些线程的优先级来改善一个应用程序的响应能力，这样做并非不合理，但却是不必要的，其结果会因为JVM实现的不同而有所差别。[206]通过调整线程的优先级来解决一个严重的活性问题是不合理的；在你找到并修正底层的真正原因之前，这个问题可能会再次出现。

对于大多数程序员来说，`Thread.yield`的惟一用途是在测试期间人为地增加一个程序的并发性。通过探查一个程序更大部分的状态空间，可以发现一些隐蔽错误（bug），从而对系统的正确性增强信心。这项技术已经被证明对于找出“微妙的并发性错误”非常有效。

简而言之，不要让应用程序的正确性依赖于线程调度器。否则，结果得到的应用程序既不健壮，也不具有可移植性。作为一个推论，不要依赖`Thread.yield`或者线程优先级。这些设施都只是影响到调度器。它们可以被用来提高一个已经能够正常工作的系统的服务质量，但永远不应该被用来“修正”一个原本并不能工作的程序。[207]

第52条：线程安全性的文档化

当一个类的实例或者静态方法被并发使用的时候，这个类的行为怎么样，这是该类与其客户建立的约定的重要组成部分。如果你没有在一个类的文档中描述它的行为的并发性情况，则使用这个类的程序员将不得不做出某些假设。如果这些假设是错误的，则这样得到的程序可能缺少足够的同步（见第48条），或者同步过多（见第49条）。无论哪种情况，都可能会发生严重的错误。

有这样一种说法，用户通过检查Javadoc产生的文档中是否出现synchronized修饰符，可以确定一个方法的线程安全性。从几个方面来说这是错误的。在1.2发行版本之前，Javadoc工具确实在它的输出中包含有synchronized修饰符，但这是一个错误（bug），后来已经被修正了。**在一个方法的声明中出现synchronized修饰符，这是一个实现细节，并不是导出的API的一部分。**出现了synchronized修饰符并不一定表明这个方法是线程安全的，它有可能随着版本的不同而发生变化。

而且，“出现了synchronized关键字就足以将线程安全性文档化了”这种说法隐含了一个错误的观念，即认为线程安全性是一种“要么全有要么全无”的属性。实际上，一个类支持的线程安全性有很多级别。一个类为了可被多个线程安全地使用，必须在文档中清楚地说明它所支持的线程安全性级别。

下面的列表概括了一个类可能支持的线程安全性级别。这份列表并没有涵盖所有的可能，而只是常见的情形。列表中使用的名字不是标准的，因为在这个领域中还没有被广泛接受的惯例：

- **非可变的（immutable）**——这个类的实例对于其客户而言是不变的。所以，不需要外部的同步。这样的例子包括String、Integer和BigInteger（见第13条）。
- **线程安全的（thread-safe）**——这个类的实例是可变的，但是所有的方法都包含足够的同步手段，所以，这些实例可以被并发使用，无需外部同步。这些并发的调用将表现为按照某种全局一致的顺序，被依次执行。其例子包括Random和java.util.Timer。
- **有条件的线程安全（conditionally thread-safe）**——这个类（或者关联的类）包含有某些方法，它们必须被顺序调用，而不能受到其他线程的干扰，除此之外，这种线程安全级别与上一种情形（线程安全的）相同。为了消除被其他线程干扰的可能性，客户在执行此方法序列期间，必须获得一把适当的锁。这样的例子包括Hashtable和Vector，它们的迭代器（iterator）要求外部同步。

- 线程兼容的 (thread-compatible) —— 在每个方法调用 (有些情况下，在每个方法调用序列) 的外围使用外部同步，此时，这个类的实例可以被安全地并发使用。其例子包括通用的集合实现，比如ArrayList和HashMap。
- 线程对立的 (thread-hostile) —— 这个类不能安全地被多个线程并发使用，即使所有的方法调用都被外部同步包围。通常情况下，线程对立的根源在于，这个类的方法要修改静态数据，而这些静态数据可能会影响其他的线程。幸运的是，在Java平台库中，线程对立的类或者方法非常少。System.runFinalizersOnExit方法是线程对立的，但已经被废弃了。

在文档中描述一个有条件的线程安全类要特别小心。你必须指明哪个调用序列要求外部同步，还要指明为了避免并发访问，哪一把锁（极少的情况下是多把锁）必须要获得。通常情况下，这是指作用在实例自身的那把锁，但也有例外。如果一个对象代表了另一个对象的一个视图 (view)，那么客户必须要获得一把作用在后台对象上的锁，以防止其他线程直接修改后台对象。例如，Hashtable.keys的文档应该有这样的说明：

If there is any danger of another thread modifying this hash table, safely enumerating over its keys requires that you lock the Hashtable instance prior to calling this method, and retain the lock until you are finished using the returned Enumeration, as demonstrated in the following code fragment:

[如果可能会有其他线程修改此散列表，那么，为了安全地对它的键进行枚举，要求你在调用此方法之前先锁住Hashtable实例，然后一直保持该锁，直到枚举完了被返回的Enumeration为止。如下面的代码片断所示：]

```
Hashtable h = ...;

synchronized (h) {
    for (Enumeration e = h.keys(); e.hasMoreElements(); )
        f(e.nextElement());
}
```

从1.3发行版本开始，Hashtable的文档不再包含以上这段说明，但是，希望这种情况很快会被改正。更一般地，Java平台库应该对有关线程安全性的文档做得更好一些。

承诺了“使用一个公有可访问的锁对象”，这就允许客户以原子的方式执行一个方法调用序列，但是，这种灵活性是要付出代价的。一个恶意的客户可以发起拒绝服务 (denial-of service) 攻击，它只需简单地保持住该对象上的锁即可：

```
// Denial-of-service attack
synchronized (importantObject) {
```

```

    Thread.sleep(Integer.MAX_VALUE); // Disable importantObject
}

```

如果你很在意这种拒绝服务攻击，那么你应该使用一个私有锁对象（*private lock object*）对操作进行同步：

```

// Private lock object idiom - thwarts denial-of-service attack
private Object lock = new Object();

public void foo() {
    synchronized(lock) {
        ...
    }
}

```

因为该锁是在客户不可访问的对象上被获得的，所以外围对象可以免受上述拒绝服务攻击。注意，有条件的线程安全类总是容易受到这种攻击，因为它们必须在文档中指明、当以原子方式执行操作序列时，必须要使用哪一把锁。然而，利用上面的私有锁对象模式，线程安全类可以免受这种攻击。

使用内部对象作为锁对象特别适合于那些专门为继承而设计的类（见第15条），比如第49条中的`workQueue`类。如果超类使用它的实例作为锁对象，则子类可能会无意中妨碍它的操作。出于不同的目的而使用相同的锁，超类和子类可能会“相互绊住对方的脚”。

简而言之，每一个类都应该清楚地在文档中说明它的线程安全属性。要做到这一点，惟一的办法是提供字句确凿的描述。`synchronized`修饰符并不能成为一个类的线程安全性文档。然而，对于有条件的线程安全类，在文档中指明“为了允许方法调用序列以原子方式执行，哪一个对象应被锁住”，这是非常重要的。一个类的线程安全性描述通常属于这个类的文档注释，但是，对于具有特殊线程安全属性的方法来说，它们应该在自己的文档注释中描述这些线程安全属性。

第53条：避免使用线程组

除了线程、锁和监视器之外，线程系统还提供了一个基本的抽象，即线程组（*thread group*）。线程组的初衷是作为一种隔离applet（小程序）的机制，当然是出于安全的考虑。它们并没有真正实现这个承诺，它们的安全重要性已经差到在Java 2平台安全模型的核心工作中不被提及的地步[Gong99]。

既然线程组并没有提供所提及的任何安全功能，那么它们到底提供了什么功能？最为贴近的说法是，它们允许你把Thread的基本功能直接应用到一组线程上。其中有一些原语功能已经被废弃了，剩下的也很少使用。总而言之，线程组并没有提供太多有用的功能。

具有讽刺意味的是，从线程安全性的角度来看，`ThreadGroup` API非常弱。为了得到一个线程组中的活动线程列表，你必须调用`enumerate`方法，它有一个数组参数，并且数组的容量必须足够大，以便容纳所有的活动线程。`activeCount`方法返回一个线程组中活动线程的数量，但是，客户在得到这个数值之后，分配一个数组，然后把数组传递给`enumerate`方法，此时，原先得到的活动线程数已不再保证仍是正确的。如果数组太小的话，`enumerate`方法会悄然地忽略掉任何额外的线程。

获取线程组中子组列表的API也有类似的缺陷。虽然通过增加新的方法，这些问题都有可能被修正，但是，它们目前还没有被修正，因为没有实际的需要；线程组基本上已经过时了。

总之，线程组并没有提供太多有用的功能，而且它们提供的许多功能还都是有缺陷的。我们最好把线程组看做一个不成功的试验，你可以忽略掉它们，就当它们根本不存在一样。如果你正在设计的一个类需要处理线程的逻辑组，那么，你只要把每个逻辑组中的所有Thread引用保存到一个数组或者集合中就可以了。敏锐的读者可能会注意到，这条建议似乎与第30条的建议“了解和使用库”矛盾。对于这里的特殊情形，第30条的建议是错误的。

上面提到的建议“你应该忽略线程组”有一个小小的例外。有一种小功能只有`ThreadGroup` API才有。当线程组中的一个线程抛出一个未被捕获的异常时，`ThreadGroup.uncaughtException`方法会自动被调用，“执行环境”使用这个方法，以便用适当的方式来响应未被捕获的异常。该方法的默认实现将栈轨迹打印到标准错误流中。你有时候可能会希望改写这个实现，以便完成特定的功能，比如，将栈轨迹定向到一个由应用指定的日志中。

第10章

序 列 化

本章关注对象的序列化 (*object serialization*) API，它提供了一个框架，用来将对象编码成一个字节流，以及从字节流编码中重新构建对象。“将一个对象编码成一个字节流”这被称为序列化 (*serializing*) 该对象；相反的处理过程被称为反序列化 (*deserializing*)。一旦一个对象被序列化后，则它的编码可以从一个正在运行的虚拟机被传递到另一个虚拟机上，或者被存储到磁盘上，供以后反序列化用。序列化技术为远程通信提供了标准的线路层 (*wire-level*) 对象表示，也为JavaBeanTM组件结构提供了标准的永久数据格式。

第54条：谨慎地实现Serializable

要想使一个类的实例可被序列化，非常简单，只要在它的声明中加入“`implements Serializable`”即可。正因为太容易了，所以普遍存在这样一种误解：程序员只需要做极少的工作就可以支持序列化了。实际情形要复杂得多。虽然使一个类可被序列化的直接开销非常低，甚至可以被忽略，但是为了序列化而付出的长期开销往往是实实在在的。

因为实现`Serializable`而付出的最大代价是，一旦一个类被发布，则“改变这个类的实现”的灵活性将大大降低。如果一个类实现了`Serializable`，那么它的字节流编码（或者说序列化形式，*serialized form*）变成了它的导出的API的一部分。一旦这个类已经被广泛使用，那么你往往必须要永远支持这种序列化形式，就好像你必须要支持所有其他部分导出的API一样。如果你没有精力来设计一个自定义的序列化形式 (*custom serialized form*)，而仅仅接受了默认的序列化形式，那么，这个类的序列化形式将被永远束缚在该类最初的内部表示上。换句话说，如果你接受了默认的序列化形式，那么这个类中私有的和包级私有的实例域都变成了导出的API的一部分，这不符合“对域的访问进行最小化”的实践准则（见第12条），从而它就失去了作为信息隐藏工具的有效性。

如果你接受了默认的序列化形式，并且以后又要改变这个类的内部表示，那么结果会导致

序列化形式的不兼容。客户企图用这个类的老版本来序列化一个类，然后用新的版本来做反序列化，则结果将导致程序失败。在改变内部表示的同时仍然维持原来的序列化形式（使用`ObjectOutputStream.putFields`和`ObjectInputStream.readFields`），这也是可能的，但是做起来比较困难，并且会在源代码中留下一些看得见的“瘤”。因此，你应该仔细地设计一个高质量的序列化形式，并且在很长时间内你都会愿意使用这种形式（见第55条）。这样做将会增加开发成本，但这是值得的：一个设计良好的序列化形式也许会给类的演化带来限制；但设计不好的序列化形式可能会使一个类根本无法演化。

序列化会使类的演化受到限制，这种限制的一个例子与流的惟一标识符（*stream unique identifier*）有关，通常它也被称为序列版本`UID`（*serial version UID*）。每一个可序列化的类都有一个惟一标识号与它相关联。如果你没有在一个名为`serialVersionUID`的私有静态final的long域中显式地指定该标识号，那么系统会自动将一个确定性的复杂过程作用在这个类上，从而产生该标识号。这个自动产生的值会受到类名字、它实现的接口的名字、以及所有公有和受保护的成员的名字的影响。如果你用任何方式改变了这些信息，比如，增加了一个不是很重要的工具方法，则自动产生的序列版本`UID`也会有变化。因此，如果你没有声明一个显式的序列版本`UID`，则兼容性将会被打破。

实现`Serializable`的第二个代价是，它增加了错误（bug）和安全漏洞的可能性。通常情况下，对象是由构造函数来创建的；序列化机制是一种语言之外的对象创建机制（*extralinguistic mechanism*）。无论你是接受了默认的行为，还是改写了默认的行为，反序列化机制（`deserialization`）是一个“隐藏的构造函数”，具备与其他构造函数相同的特点。因为反序列化机制中没有显式的构造函数，所以你很容易忘记了要确保这一点：反序列化过程必须也要保证所有“由真正的构造函数建立起来的约束关系”，并且不允许攻击者能访问到正在构造过程中的对象的内部信息。依靠默认的反序列化机制会很容易使对象的约束关系受到破坏，以及遭受到非法访问（见第56条）。

214

实现`Serializable`的第三个代价是，随着一个类的新版本的发行，相关的测试负担增加了。当一个可序列化的类被修订的时候，很重要的一点是，要检查是否可以“在新版本中序列化一个实例，然后在老版本中反序列化”，或者相反的过程。因此，测试所需要的工作量与“可序列化的类的数量 × 版本数”的乘积成正比，这个乘积可能会非常大。这些测试不可能被自动构建，因为除了二进制兼容性（*binary compatibility*）以外，你还必须测试语义兼容性（*semantic compatibility*）。换句话说，你必须既要确保“序列化 - 反序列化”过程成功，也要确保结果产生的对象真正是原始对象的复制品。可序列化类的变化越大，它就越需要测试。如果在最初编写一个类的时候，就精心设计了自定义的序列化形式，那么测试的要求可以有所降低，但是也不能完全没有测试。

实现**Serializable**接口不是一个很轻松就可以做出的决定。它提供了一些实际的益处：如果一个类将要加入到某个框架中，并且该框架依赖于序列化来实现对象传输或者永久化，那么对于这个类来说，实现**Serializable**接口就非常有必要。更进一步，如果这个类要成为另一个类的一个组件，并且后者必须实现**Serializable**接口，那么若前者也实现了**Serializable**接口，则它会更易于被后者使用。然而，有许多实际的开销都与实现**Serializable**接口有关。每当你实现一个类的时候，你都需要权衡一下所付出的代价和带来的好处。根据经验，比如**Date**和**BigInteger**这样的值类应该实现**Serializable**，大多数的集合类也应该如此。代表活动实体的类，比如线程池（thread pool），一般不应该实现**Serializable**。从Java 1.4发行版本开始，有了一种基于XML的JavaBeans永久机制，所以，对于Beans来说，实现**Serializable**不再是必需的了。

为了继承而设计的类应该很少实现**Serializable**，接口也应该很少会扩展它。如果违反了这条规则，则扩展这个类或者实现这个接口的程序员会背上沉重的负担。然而在有些情况下违反这条规则是合适的。例如，如果一个类或者接口存在的主要目的是为了参与到某个框架中，而该框架要求所有的参与者必须实现**Serializable**，那么，对于这个类或者接口来说，实现或者扩展**Serializable**是非常有意义的。

有一条告诫与“不要实现**Serializable**接口”有关。如果一个专门为继承而设计的类不是可序列化的，那么要编写出可序列化的子类几乎是不可能的。特别地，如果超类没有提供一个可访问的、无参数的构造函数的话，那么子类要做到可序列化是不可能的。因此，对于为继承而设计的不可序列化的类，你应该考虑提供一个无参数的构造函数。通常这并不要求付出特别的努力，因为许多为继承而设计的类都没有状态，但是情况并不总是这样的。

最好在所有的约束关系都已经建立的情况下再创建对象（见第13条）。如果为了建立这些约束关系而要求客户提供一些信息，那么这实际上排除了使用无参数的构造函数的可能性。简单地为一个类增加一个无参数的构造函数和一个初始化方法，而它的约束关系仍由其他的构造函数来建立，则这样做会使该类的状态空间更加复杂，并且增加出错的可能性。
215

有一种办法可以给“不可序列化但可扩展的类”增加无参数的构造函数，同时避免以上的不足。假设该类有这样一个构造函数：

```
public AbstractFoo(int x, int y) { ... }
```

下面的转换增加了一个受保护的无参数构造函数，和一个初始化方法。初始化方法与正常的构造函数有相同的参数，并且它也建立起同样的约束关系：

```
// Nonserializable stateful class allowing serializable subclass
public abstract class AbstractFoo {
    private int x, y; // The state
```

```

private boolean initialized = false;

public AbstractFoo(int x, int y) { initialize(x, y); }

/**
 * This constructor and the following method allow subclass's
 * readObject method to initialize our internal state.
 */
protected AbstractFoo() { }

protected final void initialize(int x, int y) {
    if (initialized)
        throw new IllegalStateException(
            "Already initialized");
    this.x = x;
    this.y = y;
    ... // Do anything else the original constructor did
    initialized = true;
}

/**
 * These methods provide access to internal state so it can
 * be manually serialized by subclass's writeObject method.
 */
protected final int getX() { return x; }
protected final int getY() { return y; }

// Must be called by all public instance methods
private void checkInit() throws IllegalStateException {
    if (!initialized)
        throw new IllegalStateException("Uninitialized");
}
... // Remainder omitted
}

```

[216]

AbstractFoo中所有的实例方法在开始自己的工作之前必须调用checkInit。这样可以确保如果有一个编写得很差的子类没有初始化实例的话，该方法调用可以快速而干净地失败（而不会拖泥带水地做了很多无用功之后才失败）。如果有这样的机制做保证，则实现一个可序列化的子类就非常简单明了：

```

// Serializable subclass of nonserializable stateful class
public class Foo extends AbstractFoo implements Serializable {
    private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
        s.defaultReadObject();

        // Manually deserialize and initialize superclass state
        int x = s.readInt();
        int y = s.readInt();
        initialize(x, y);
    }

    private void writeObject(ObjectOutputStream s)
        throws IOException {
        s.defaultWriteObject();
    }
}

```

```
// Manually serialize superclass state
s.writeInt(getX());
s.writeInt(getY());
}

// Constructor does not use any of the fancy mechanism
public Foo(int x, int y) { super(x, y); }
}
```

内部类（*inner class*）（见第18条）应该很少实现Serializable。它们使用编译器产生的合成域（*synthetic field*）来保存指向外围实例（*enclosing instance*）的引用，以及保存来自外围作用域的局部变量的值。“这些域如何对应到类定义中”没有被指定，就好像匿名类和局部类的名字没有被指定一样。因此，内部类的默认序列化形式是定义不清楚的。然而，静态成员类可以实现Serializable。

简而言之，不要相信实现Serializable会很容易。除非一个类在用了一段时间之后就会被抛弃，否则，实现Serializable是一个很严重的承诺，必须认真对待。如果一个类是为了继承而设计的，则更加需要额外的小心。对于这样的类而言，在“允许子类实现Serializable”或“禁止子类实现Serializable”两者之间的一个折衷设计方案是，

提供一个可访问的无参数构造函数。该设计方案允许（但不要求）子类实现Serializable。

第55条：考虑使用自定义的序列化形式

当你在时间紧迫的情况下设计一个类时，一般合理的设计是把工作重心集中在API的设计上，努力设计出最好的API。有时候，这意味着要发行一个“用完后即丢弃”的实现，因为你知道很快你将会用一个新的版本来替换它。正常情况下，这不是一个问题，但是，如果这个类实现了Serializable，并且使用了默认的序列化形式，那么你永远也摆脱不了那个要被丢弃的实现。它将永远牵制住这个类的序列化形式。这不是一个纯理论的问题，在Java平台库中已经有几个类出现了这样的问题，比如BigInteger。

若没有认真考虑默认序列化形式是否合适，则不要接受这种形式。接受默认的序列化形式是一个非常重要的决定，你需要从灵活性、性能和正确性多个角度对这种编码形式进行考察。一般来讲，只有当你自行设计的自定义序列化形式与默认序列化形式基本相同时，你才应该接受默认的序列化形式。

考虑以一个对象为根的对象图，相对于它的物理表示而言，该对象的默认序列化形式是一种比较有效的编码形式。换句话说，默认序列化形式描述了该对象内部所包含的数据，以及每一个可以从这个对象到达的其他对象的内部数据。它也描述了所有这些对象被链接起来的拓扑结构。对于一个对象来说，理想的序列化形式应该只包含该对象所表示的逻辑数据，而逻辑数据与物理表示应该是独立的。

如果一个对象的物理表示等同于它的逻辑内容，则默认的序列化形式可能是合适的。例如，对于下面的表示人名字的类，默认序列化形式是合理的：

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
     * Last name. Must be non-null.
     * @serial
     */
    private String lastName;

    /**
     * First name. Must be non-null.
     * @serial
     */
    private String firstName;

    /**
     * Middle initial, or '\u0000' if name lacks middle initial.
     * @serial
     */
    private char middleInitial;

    ... // Remainder omitted
}
```

从逻辑的角度而言，一个名字包含两个字符串（分别代表姓和名）和一个字符（代表中间名的大写首字母）。Name中的实例域精确地反映了它的逻辑内容。

即使你确定了默认序列化形式是合适的，通常你仍然要提供一个`readObject`方法以保证约束关系和安全性。对于Name这个类的情形，`readObject`方法可以确保`lastName`和`firstName`是非null的。第56条将细致讨论这个问题。

注意，虽然`lastName`、`firstName`和`middleInitial`域是私有的，但是它们仍然有相应的注释文档。这是因为，这些私有域定义了一个公有的API，即这个类的序列化形式，并且该公有的API必须被文档化。`@serial`标签告诉Javadoc工具，把这些文档信息放在有关序列化形式的文档页中。

下面的类与Name不同，它是另一个极端，该类表示了一个字符串列表（此刻我们暂时忽略关于“最好使用标准库中`List`实现”的建议）：

```
// Awful candidate for default serialized form
public class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }

    ... // Remainder omitted
}
```

从逻辑意义上讲，这个类表示了一个字符串序列。但是从物理意义上讲，它把该序列表示成一个双向链表。如果你接受了默认的序列化形式，则该序列化形式将不遗余力地镜像出（mirror）链表中的所有项目，以及这些项目之间的所有双向链接。
[219]

当一个对象的物理表示与它的逻辑数据内容有实质性的区别时，使用默认序列化形式有4个缺点：

- 它使这个类的导出API永久地束缚在该类的内部表示上。在上面的例子中，私有的`StringList.Entry`类变成了公有API的一部分。如果在将来的版本中，内部表示发生了变化，则`StringList`类仍将要接受链表形式的输入，并产生链表形式的输出。这个类永远也摆脱不掉维护链表所要求的代码，即使它不再使用链表作为内部数据结构了，它仍然需要这些代码。
- 它要消耗过多的空间。在上面的例子中，序列化形式既表示了链表中的每个项目，也表示了所有的链接关系，这是不必要的。这些链表项目以及链接只不过是实现细节，不值

得记录在序列化形式中。因为这样的序列化形式过分庞大，所以，把它写到磁盘中，或者在网络上发送都将会非常慢。

- 它要消耗过多的时间。序列化逻辑并没有关于对象图拓扑关系的知识，所以它必须要经过一个昂贵的图遍历（traversal）过程。在上面的例子中，沿着next引用进行遍历是非常简单的。
- 它会引起栈溢出。默认的序列化过程要对对象图执行一次递归遍历，即使对于中等规模的对象图，这样的操作也可能会引起栈溢出。在我的机器上，如果StringList实例包含1200个元素，则对它进行序列化就会导致栈溢出。到底多少个元素会引发栈溢出，这要取决于JVM实现，有些实现可能根本不存在这样的问题。

对于StringList类，合理的序列化形式可以非常简单，只需先包含链表中字符串的数目，然后紧跟着这些字符串即可。这样正好构成了StringList所表示的逻辑数据，脱离了它的物理表示细节。下面是StringList的一个修订版本，它包含两个方法writeObject和readObject，用来实现这样的序列化形式。顺便提醒一下，transient修饰符表明这个实例域将从一个类的默认序列化形式中省略掉：

```
// StringList with a reasonable custom serialized form
public class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;
    // No longer Serializable!
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    // Appends the specified string to the list
    public void add(String s) { ... }

    /**
     * Serialize this <tt>StringList</tt> instance.
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted (<tt>int</tt>), followed by all of
     * its elements (each a <tt>String</tt>), in the proper
     * sequence.
     */
    private void writeObject(ObjectOutputStream s)
        throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }
}
```

[220]

```

private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();
    int numElements = s.readInt();

    // Read in all elements and insert them in list
    for (int i = 0; i < numElements; i++)
        add((String)s.readObject());
}

... // Remainder omitted
}

```

注意，尽管 `StringList` 的所有域都是 `transient` 的，但 `writeObject` 方法仍然调用了 `defaultWriteObject`，`readObject` 方法仍然调用了 `defaultReadObject`。如果所有的实例域都是 `transient` 的，那么，从技术角度而言，省去调用 `defaultWriteObject` 和 `defaultReadObject` 也是允许的，但是不推荐这样做。即使所有的实例域都是 `transient` 的，调用 `defaultWriteObject` 也会影响该类的序列化形式，从而极大地增强灵活性。这样得到的序列化形式允许在以后的发行版本中增加非 `transient` 的实例域，并且还能保持前向或者后向兼容性。如果一个实例是在后来的版本中被序列化的，然后在前一个版本中被反序列化，那么，后增加的域将被忽略掉。如果以前版本的 `readObject` 方法没有调用 `defaultReadObject` 的话，则反序列化过程将失败，引发 `StreamCorruptedException` 异常。
[22]

注意，尽管 `writeObject` 方法是私有的，它也有文档注释。这与 `Name` 类中私有域的文档注释是同样的道理。该私有方法定义了一个公有的 API，即序列化形式，并且这个公有的 API 应该被文档化。如同针对域的 `@serial` 标签一样，针对方法的 `@serialData` 标签也告知 Javadoc 工具，要把该文档信息放在有关序列化形式的文档页上。

套用以前关于性能的讨论形式，如果平均字符串长度为 10 个字符，则 `StringList` 修订版本的序列化形式只占用原来序列化形式一半的空间。在我的机器上，同样是 10 个字符长度的情况下，`StringList` 修订版本的序列化速度比原来版本快 2.5 倍。最后，修订版本中不存在栈溢出的问题，因此，对于可被序列化的 `StringList` 的大小也没有实际的上限。

默认的序列化形式对于 `StringList` 类只是不适合而已，而对于有些类，情况会变得更加糟糕。对于 `StringList`，默认的序列化形式不够灵活，并且执行效果不佳，但是序列化和反序列化一个 `StringList` 实例会产生原始对象的忠实拷贝，它的约束关系没有被破坏，从这个意义上讲，序列化形式是正确的。但是，其他有些对象的约束关系要依赖于特定的实现细节，对于它们来说，情况就不是这样了。

例如，考虑散列表的情形。它的物理表示是一系列包含“键 - 值”对元素的散列桶。到底一个“键 - 值”对元素将被放在哪个桶中，这是该键的散列码的一个函数，一般情况下，不同

的JVM实现不保证会有同样的结果。实际上，即使在同一个JVM实现中，也无法保证每次运行会一样。因此，对于散列表而言，接受默认的序列化形式将是一个严重的错误（bug）。对散列表对象进行序列化和反序列化操作所产生的对象，其约束关系会被严重打破。

无论你是否使用默认的序列化形式，当`defaultWriteObject`方法被调用的时候，每一个未被标记为`transient`的实例域都会被序列化。因此，每一个可以被标记为`transient`的实例域都应该带上这样的标记。这包括那些信息冗余的域，即，其值可以根据其他“基本数据域”计算而得的域，比如缓存起来的散列值。它也包括那些“其值依赖于JVM的某一次运行”的域，比如一个`long`域代表了一个指向本地数据结构的指针。**在决定将一个域做成非`transient`之前，请一定要确信它的值将是该对象逻辑状态的一部分。**如果你正在使用一种自定义的序列化形式，那么，大多数实例域，或者所有的实例域都应该被标记为`transient`，就像上面例子中的`StringList`那样。

222

如果你正在使用默认的序列化形式，并且你把一个或者多个域标记为`transient`，则记住，当一个实例被反序列化的时候，这些域将被初始化为它们的默认值（default value）：对于对象引用域，默认值为`null`；对于数值原语域，默认值为0；对于`boolean`域，默认值为`false`[JLS, 4.5.5]。如果这些值不能被任何`transient`域所接受，那么你必须提供一个`readObject`方法，它首先调用`defaultReadObject`，然后把这些`transient`域恢复为可接受的值（见第56条）。另一种方法是，这些域可以被迟缓初始化，直到第一次被使用的时候才真正初始化。

不管你选择了哪种序列化形式，你都要为自己编写的每个可序列化的类声明一个显式的序列版本UID（serial version UID）。这样可以消除序列版本UID成为潜在的不兼容根源（见第54条）。而且，这样做也会带来小小的性能好处。如果没有提供显式的序列版本UID，则需要在运行时刻通过一个高开销的计算过程产生一个序列版本UID。

要声明一个序列版本UID非常简单，只要在你的类中增加下面一行：

```
private static final long serialVersionUID = randomLongValue;
```

你为`randomLongValue`选择什么值并不重要。实践中常用的做法是，通过在该类上运行`serialver`工具，你就可以得到一个这样的值，但是，如果你随意地编造一个数值，那也是可以的。如果你想为一个类生成一个新的版本，并且希望它与现有的类不兼容（*incompatible*），那么你只需修改声明中的序列版本UID即可。结果是，以前版本的实例经序列化之后，再做反序列化时会引发`InvalidClassException`而失败。

总而言之，当你决定要将一个类做成可序列化的时候（见第54条），请仔细考虑应该采用什么样的序列化形式。只有当默认的序列化形式能够合理地描述对象的逻辑状态的时候，你

才使用默认的序列化形式；否则就设计一个自定义的序列化形式，通过它合理地描述对象的状态。你应该分配足够多的时间来设计一个类的序列化形式，就好像你分配足够多的时间来设计它的导出方法一样。正如你无法在将来的版本中去掉导出方法一样，你也不能去掉序列化形式中的域；它们必须被永久地保留下去，以确保序列化兼容性（*serialization compatibility*）。选择错误的序列化形式对于一个类的复杂性和性能都会有永久的负面影响。
[223]

第56条：保护性地编写readObject方法

第24条介绍了一个非可变的日期范围类Period，它包含两个可变的私有数据域。该类通过在其构造函数和访问方法（accessor）中保护性地拷贝Date对象，极力地维护其约束条件和非可变性。下面是这个类：

```
// Immutable class that uses defensive copying
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param start the beginning of the period.
     * @param end the end of the period; must not precede start.
     * @throws IllegalArgumentException if start is after end.
     * @throws NullPointerException if start or end is null.
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());

        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(start + " > " + end);
    }

    public Date start() { return (Date) start.clone(); }
    public Date end() { return (Date) end.clone(); }
    public String toString() { return start + " ~ " + end; }
    ... // Remainder omitted
}
```

假设你决定要把这个类做成可序列化的。因为Period对象的物理表示正好吻合它的逻辑数据内容，所以，使用默认的序列化形式没有什么不合理的（见第55条）。因此，为了使这个类成为可序列化的，似乎你所需要做的也就是在类的声明中增加“implements Serializable”字样。然而，如果你真的这样做了，那么这个类就不再保证它的关键约束了。

问题在于，readObject方法实际上相当于另一个公有的构造函数，如同其他的构造函数一样，它也要求所有同样的注意事项。一个构造函数必须检查它的实参的有效性（见第23条），并且必要的时候对参数进行保护性拷贝（见第24条）。同样地，readObject方法也需要这样做。如果readObject方法无法做到这两者之一，那么，对于攻击者来说，要违反这个类的约束条件相对比较简单。

224

不严格地说，readObject是一个“用字节流作为唯一参数”的构造函数。在正常使用

的情况下，字节流是这样产生的：对一个被正常构造的实例进行序列化可以产生一个字节流。但是，当面对一个人工伪造的字节流的时候，`readObject`产生的对象会违反它所属的类的约束条件，这时问题就产生了。假设我们仅仅在`Period`类的声明中加上了“`implements Serializable`”。那么，这个不完整的程序所产生的`Period`实例将违反它的基本约束条件（结束时间比起始时间还要早）：

```

public class BogusPeriod {
    // Byte stream could not have come from real Period instance
    private static final byte[] serializedForm = new byte[] {
        (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
        0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
        0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
        0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
        0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
        0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
        0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
        0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
        0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
        (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
        0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
        0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
        0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
        0x00, 0x78 };
}

public static void main(String[] args) {
    Period p = (Period) deserialize(serializedForm);
    System.out.println(p);
}

// Returns the object with the specified serialized form
public static Object deserialize(byte[] sf) {
    try {
        InputStream is = new ByteArrayInputStream(sf);
        ObjectInputStream ois = new ObjectInputStream(is);
        return ois.readObject();
    } catch (Exception e) {
        throw new IllegalArgumentException(e.toString());
    }
}
}

```

225

被用来初始化`serializedForm`的`byte`数组常量是这样产生的：首先对一个正常的`Period`实例进行序列化，然后对结果得到的字节流手工进行编辑。对于这个例子而言，字节流的细节并不重要，但如果你很好奇的话，你可以在《Java™ Object Serialization Specification》[Serialization, 6]中查到有关序列化字节流格式的描述信息。如果你运行这个程序，它会打印出“Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984”。这样简单地把`Period`做成一个可序列化的类，因而使得我们有可能创建出违反其约束条件的实例对象。为了修正这个问题，你可以为`Period`提供一个`readObject`方法，该方法首先调用`defaultReadObject`，然后检查被反序列化之后的对象的有效性。如果有效性检查失败，则`readObject`方法抛出一个`InvalidObjectException`异常，使

反序列化过程不能得以成功完成：

```
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}
```

尽管这样的修正使得攻击者无法创建出无效的Period实例来，但是，这里仍然隐藏着一个更为微妙的问题：通过伪造一个字节流，要想创建一个可变的Period实例还是有可能的，做法是：该字节流以一个有效的Period实例所产生的字节流作为开始，然后附加上两个额外的引用，指向Period实例中的两个内部私有Date域。攻击者从ObjectInputStream中读取Period实例，然后读取附加在其后面的“恶意编制的对象引用”。这些对象引用使得攻击者能够访问到Period对象内部的私有Date域所引用的对象。通过改变这些Date实例，攻击者可以改变Period实例，下面的类演示了这种攻击：

```
public class MutablePeriod {
    // A period instance
    public final Period period;

    // period's start field, to which we shouldn't have access
    public final Date start;

    // period's end field, to which we shouldn't have access
    public final Date end;
    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            ObjectOutputStream out =
                new ObjectOutputStream(bos);

            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * Append rogue "previous object refs" for internal
             * Date fields in Period. For details, see "Java
             * Object Serialization Specification," Section 6.4.
             */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // The start field
            ref[4] = 4; // Ref #4
            bos.write(ref); // The end field

            // Deserialize Period and "stolen" Date references
            ObjectInputStream in = new ObjectInputStream(
                new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start = (Date) in.readObject();
            end = (Date) in.readObject();
        }
    }
}
```

226

```

        } catch (Exception e) {
            throw new RuntimeException(e.toString());
        }
    }
}

```

运行下面的程序，可以看到攻击的效果：

```

public static void main(String[] args) {
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;

    // Let's turn back the clock
    pEnd.setYear(78);
    System.out.println(p);

    // Bring back the 60's!
    pEnd.setYear(69);
    System.out.println(p);
}

```

[227]

运行这个程序，产生如下的输出结果：

```

Wed Mar 07 23:30:01 PST 2001 - Tue Mar 07 23:30:01 PST 1978
Wed Mar 07 23:30:01 PST 2001 - Fri Mar 07 23:30:01 PST 1969

```

虽然Period实例被创建之后，它的约束条件没有被直接破坏，但是要随意地修改它的内部组件仍然是有可能的。一旦攻击者获得了一个可变的Period实例，他可以将这个实例传递给一个“安全性依赖于Period的非可变性”的类，从而造成更大的危害。这并不牵强：实际上，有许多类的安全性就依赖于String的非可变性。

问题的根源在于，Period的readObject方法并没有实施足够的保护性拷贝。当一个对象被反序列化的时候，对于客户不应该拥有的对象引用，如果哪个域包含了这样的对象引用，则必须要做保护性拷贝，这是非常重要的。因此，对于每一个可序列化的非可变类，如果它包含了私有的可变组件，那么在它的readObject方法中，必须要对这些组件进行保护性拷贝。下面的readObject方法可以确保Period的约束条件不会受到破坏，以保持它的非可变性：

```

private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Defensively copy our mutable components
    start = new Date(start.getTime());
    end   = new Date(end.getTime());

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}

```

注意，保护性拷贝是在有效性检查之前进行的，而且，我们没有使用Date的clone方法来执行保护性拷贝。这两点对于保护Period免受攻击是必要的（见第24条）。同时也要注意到，对于final域，保护性拷贝是不可能的。为了使用readObject方法，我们必须要将start和end域做成非final的。这是很不幸的，但是很明显，这是相对比较好的做法。有了这个新的readObject方法，并去掉了start和end域的final修饰符之后，MutablePeriod类将不再有效。此时，上面的攻击程序会产生这样的输出：

```
Thu Mar 08 00:03:45 PST 2001 - Thu Mar 08 00:03:45 PST 2001
Thu Mar 08 00:03:45 PST 2001 - Thu Mar 08 00:03:45 PST 2001
```

228

有一个简单的“石蕊”测试，可以用来确定默认的readObject方法是否可以被接受。对于这样的做法——“增加一个公有的构造函数，其参数对应于该对象中每一个非transient的域，并且无论参数的值是什么，都不做检查就保存到相应的域中”——你是否会感到舒适？如果你对这个问题不能回答“是”，那么你必须提供一个显式的readObject方法，并且它必须要执行构造函数所要求的所有有效性检查和保护性拷贝。

对于非final的可序列化的类，在readObject方法和构造函数之间还有其他类似的地方readObject方法不可以调用可被改写的方法，无论是直接调用还是间接调用都不可以（见第15条）。如果违反了这条规则，并且该方法被改写了的话，则被改写的方法将在子类的状态被反序列化之前先运行。程序很可能会失败。

总而言之，每当你编写readObject方法的时候，你都要这样想：你正在编写一个公有的构造函数，无论给它传递一个什么样的字节流，它都必须产生一个有效的实例。不要假设这个字节流一定代表一个真正被序列化之后的实例。虽然在本条目的例子中，类使用了默认的序列化形式，但是，所有讨论到的有可能发生的问题也同样适用于使用自定义序列化形式的类。下面以摘要的形式给出一些为编写出更加健壮的readObject方法而应该遵循的指导原则：

- 对于对象引用域必须保持为私有的类，对“将被保存到这些域中的对象”进行保护性拷贝。非可变类的可变组件就属于这一类别。
- 对于具有约束条件的类，一定要检查约束条件是否满足，如果不满足的话，则抛出一个InvalidObjectException异常。这些检查动作应该跟在所有的保护性拷贝之后。
- 如果在对象图被反序列化之后，整个对象图必须都是有效的，则应该使用ObjectInputValidation接口。关于这个接口的用法已经超出了本书的范围，读者可以从《The Java Class Libraries, Second Edition, Volume I》[Chan98, p.1256]中找到

一个示例用法。

- 无论是直接方式还是间接方式，都不要调用类中可被改写的方法。

`readResolve`方法有可能被用来替代保护性的`readObject`方法。第57条将讨论这种

[229] 替代方案。

第57条：必要时提供一个readResolve方法

第2条讲述了Singleton模式，并且给出了以下的singleton类的示例。这个类限制了对其构造函数的访问，以确保永远只有一个实例被创建：

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();

    private Elvis() {
        ...
    }

    ... // Remainder omitted
}
```

正如在第2条中提到的，如果这个类的声明中加上了“implements Serializable”字样，那么它就不再是一个singleton。无论该类使用了默认的序列化形式，还是自定义的序列化形式（见第55条），都没有关系；也跟它是否提供了显式的readObject方法（见第56条）无关。任何一个readObject方法，不管是显式的还是默认的，它都会返回一个新建的实例，这个新建的实例不同于该类初始化时刻创建的实例。在1.2发行版本之前，要想编写一个可序列化的singleton类是不可能的。

在1.2发行版本中，序列化设施中新增加了readResolve特性[Serialization, 3.6]。对于一个正在被反序列化的对象，如果它的类定义了一个readResolve方法，并且它的声明是正确的，那么在反序列化之后，新创建对象上的readResolve方法就会被调用。然后，该方法返回的对象引用将被返回，它取代了新创建的对象。在这个特性的绝大多数用法中，指向新创建对象的引用不需要再被保留；实际上这个对象将不再有用，立即成为垃圾回收器的回收对象。

如果Elvis类要实现Serializable接口，则下面的readResolve方法足以保证它的singleton属性：

```
private Object readResolve() throws ObjectStreamException {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;
}
```

230

该方法忽略掉被反序列化的对象，简单地返回该类初始化时刻创建的那个特殊的Elvis实例。因此，Elvis实例的序列化形式并不需要包含任何实际的数据；所有的实例域都应该被标记为transient的。这种做法不仅适用于Elvis，同样也适用于所有的singleton对象。

`readResolve`方法不仅仅对于`singleton`对象是必要的，而且对于所有其他的实例受控的(*instance-controlled*)类也是必需的。这种实例受控的类的另一个例子是类型安全枚举类型(*typesafe enum*) (见第21条)，它的`readResolve`方法必须返回代表特定枚举常量的规范的实例(*canonical instance*)。凭经验，如果你正在编写的可序列化的类没有包含公有的、或者受保护的构造函数，那么请考虑它是否需要一个`readResolve`方法。

`readResolve`方法的第二个用法是，就像在第56条中建议的那样，作为保护性的`readObject`方法的一种保守的替代选择。在这种方法中，所有的有效性检查和保护性拷贝都从`readObject`方法中去掉了，而采用普通构造函数提供的有效性检查和保护性拷贝。如果使用了默认的序列化形式，那么`readObject`方法完全可以被取消。正如第56条中介绍的那样，这使得恶意的客户可以创建出违反约束条件的实例。然而，这样被反序列化出来的潜在受损的实例永远也不会进入到活动的服务中；它的相关状态信息将被输入到一个公有的构造函数或者静态工厂中，然后它被简单地丢弃掉。

这种方法的优美之处在于，它真正消除了序列化机制中语言本身之外的部分，从而使得要违反“在使一个类成为可序列化之前存在的约束条件”是不可能的。为了使这项技术更为清楚直观，下面的`readResolve`方法可以被用来代替第56条的`Period`例子中的保护性`readObject`方法：

```
// The defensive readResolve idiom
private Object readResolve() throws ObjectStreamException {
    return new Period(start, end);
}
```

对于第56条中描述的两种攻击手段，该`readResolve`方法都可以有效地阻挡住。保护性的`readResolve`模式与保护性的`readObject`方案相比有几个好处。它是这样一项很机械(*mechanical*)的技术：既使一个类可序列化，同时又不危及该类的约束条件。它要求很少的代码，也不要求考虑太多的细节，而且保证可以工作。最后，它也消除了“序列化对于使用final域的人为限制”。[23]

尽管保护性`readResolve`模式并没有被广泛使用，但是它值得认真考虑。它最大的缺点在于，它不适合于那些允许包外继承的类。对于非可变类，这不是问题，因为它们往往是final的(见第13条)。这种模式的一个次要缺点是，它略微地降低了反序列化的性能，因为它要求创建一个额外的对象。在我的机器上，与保护性的`readObject`方法相比，`Period`实例的反序列化速度减慢了大约1%。

`readResolve`方法的可访问性(accessibility)是非常重要的。如果你把`readResolve`方法放在一个final类中，比如`singleton`中，那么它应该是私有的。如果你把`readResolve`方法放在一个非final类中，则你必须要仔细考虑它的可访问性。如果它是私有的，那么它将不适

用于任何子类。如果它是包级私有的，那么它将只适用于同一个包内的子类上；如果它是受保护的或者公有的，那么它将适用于所有未改写此方法的子类。如果readResolve方法是受保护的或者公有的，并且某个子类并没有改写它，那么，先将一个子类实例做序列化，再对结果得到的字节流进行反序列化，最终会产生一个超类实例，这可能并不是你想要的结果。

上一段实际上暗示了“对于那些允许继承的类，readResolve方法可能无法替代保护性的readObject方法”。如果超类的readResolve方法是final的，那么它将使得子类实例无法被正常地反序列化。如果超类的readResolve方法是可改写的，则恶意的子类可能会用一个方法改写它，该方法返回一个受损的实例。

总而言之，无论是singleton，或是其他实例受控（instance-controlled）的类，你必须使用readResolve方法来保护“实例-控制的约束（instance-control invariant）”。从本质来讲，readResolve方法把readObject方法从一个事实上的公有构造函数变成一个事实上的公有静态工厂。对于那些禁止包外继承的类而言，readResolve方法作为保护性的readObject方法的一种替代选择，也是非常有用的。

中英文术语对照

access control	访问控制
accessibility	可访问性
accessor method	访问方法
adapter pattern	适配器模式
anonymous class	匿名类
antipattern	反模式
API (Application Programming Interface)	应用编程接口
API element	API元素
array	数组
assertion	断言
binary compatibility	二进制兼容性
callback	回调
callback framework	回调框架
checked exception	被检查的异常
class	类
client	客户
clone	克隆
comparator	比较器
composition	复合
concrete strategy	具体策略
constant interface	常量接口
copy constructor	拷贝构造函数
custom serialized form	自定义的序列化形式
decorator pattern	decorator模式
default access	默认访问
default constructor	默认构造函数
defensive copy	保护性拷贝
delegation	委托
deserializing	反序列化

design pattern	设计模式
discriminated union	可区分的联合
doc comment	文档注释
documentation comment	文档注释
double-check idiom	双重检查模式
encapsulation	封装
enclosing instance	外围实例
enumerated type	枚举类型
exception	异常
exception chaining	异常链接
exception translation	异常转译
exported API	导出的API
extend	扩展
failure atomicity	失败原子性
field	域
finalizer guardian	终结函数守卫者
forwarding	转发
forwarding method	转发方法
function object	函数对象
function pointer	函数指针
general contract	通用约定
HTML validity checker	HTML有效性检查器
idiom	习惯用(做)法, 模式
immutable	非可变的
implement	实现(用做动词)
implementation	实现(用做名词)
implementation inheritance	实现继承
information hiding	信息隐藏
inheritance	继承
inner class	内部类
integral constant	整值常量
interface	接口
interface inheritance	接口继承
Java Cryptography Extension	Java密码系统扩展, 简称JCE

lazy initialization	迟缓初始化
liveness failure	活性失败
local class	局部类
member	成员
member class	成员类
member interface	成员接口
memory footprint	内存占用
memory model	内存模型
method	方法
mixin	混合类型
module	模块
mutator	改变对象属性的方法
naming convention	命名惯例
native method	本地方法
native object	本地对象
nested class	嵌套类
nonstatic member class	非静态成员类
object	对象
object pool	对象池
object serialization	对象的序列化
obsolete reference	过期引用
open call	开放调用
overload	重载
override	改写
package-private	包级私有
performance model	性能模型
postcondition	后置条件
precondition	前提条件
precondition violation	前提违例
primitive	基本的原语的，原语类型
private	私有的
public	公有的
redundant field	冗余域
reference type	引用类型

reflection	映像机制
register	注册
rounding mode	舍入模式
run-time exception	运行时异常
safe language	安全的语言
safety	安全性
semantic compatibility	语义兼容性
self-use	自用(性)
serial version UID	序列版本UID
serialized form	序列化形式
serializing	序列化
service provider framework	服务提供者框架
signature	原型
singleton pattern	singleton模式
skeletal implementation	骨架实现
Stack trace	栈轨迹
state transition	状态转变
static factory method	静态工厂方法
static member class	静态的成员类
storage pool	存储池
strategy interface	策略接口
strategy pattern	策略模式
stream unique identifier	流的唯一标识符
subclassing	子类化
summary description	概要描述
synthetic field	合成域
thread group	线程组
thread safety	线程安全性
thread-compatible	线程兼容的
thread-safe	线程安全的
top-level class	顶层类
typesafe enum class	类型安全枚举类
typesafe enum pattern	类型安全枚举模式
unchecked exception	未被检查的异常

unintentional object retention	无意识的对象保持
utility class	工具类
value class	值类
value type	值类型
view	视图
visitor pattern	visitor模式
wrapper class	包装类

参 考 文 献

- [Arnold00] Arnold, Ken, James Gosling, David Holmes. *The Java™ Programming Language, Third Edition*. Addison-Wesley, Boston, 2000. ISBN: 0201704331.
- [Beck99] Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201616416.
- [Bloch99] Bloch, Joshua. Collections. In *The Java™ Tutorial Continued: The Rest of the JDK™*. Mary Campione, Kathy Walrath, Alison Huml, and the Tutorial Team. Addison-Wesley, Reading, MA, 1999. ISBN: 0201485583. Pages 17–93. Also available as <<http://java.sun.com/docs/books/tutorial/collections/index.html>>.
- [Campione00] Campione, Mary, Kathy Walrath, Alison Huml. *The Java™ Tutorial Continued: A Short Course on the Basics*. Addison-Wesley, Boston, MA, 2000. ISBN: 0201703939. Also available as <<http://java.sun.com/docs/books/tutorial/index.html>>. \ominus
- [Cargill96] Cargill, Thomas. Specific Notification for Java Thread Synchronization. *Proceedings of the Pattern Languages of Programming Conference*, 1996.
- [Chan00] Chan, Patrick. *The Java™ Developers Almanac 2000*, Addison-Wesley, Boston, MA, 2000. ISBN: 0201432994. \ominus
- [Chan98] Chan, Patrick, Rosanna Lee, and Douglas Kramer. *The Java™ Class Libraries Second Edition, Volume 1*, Addison-Wesley, Reading, MA, 1998. ISBN: 0201310023. \ominus

\ominus 《Java语言导学》, 中文版, 机械工业出版社, 2002。(09585)

\ominus 《Java Developer's Almanac中文版》, (1.4版), 中文版, 机械工业出版社, 2003。(11113)

\ominus 《Java类库》(增补版), 中文版, 机械工业出版社, 1999。(08232)

- [Collections] *The Collections Framework*. Sun Microsystems. March 2001.
<<http://java.sun.com/j2se/1.3/docs/guide/collections/index.html>>.
- [Doclint] *Doclint*. Ernst de Haan. March, 2001.
<<http://www.znerd.demon.nl/doclint/>>.
- [Flanagan99] Flanagan, David. *Java™ in a Nutshell, Third Edition*, O'Reilly and Associates, Sebastopol, CA, 1999. ISBN: 1565924878.
- [Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995. ISBN: 0201633612. ⊖
- [Gong99] Gong, Li. *Inside Java™ 2 Platform Security*, Addison-Wesley, Reading, MA, 1999. ISBN: 0201310007. ⊖
- [Heydon99] Allan Heydon and Marc A. Najork. Performance Limitations of the Java Core Libraries. In *ACM 1999 Java Grande Conference*, pages 35–41. ACM Press, June 1999. Also available as
<<http://research.compaq.com/SRC/mercator/papers/Java99/final.pdf>>
- [Horstman00] Horstmann, Cay, and Gary Cornell. *Core Java™ 2: Volume II—Advanced Features*, Prentice Hall, Palo Alto, CA, 2000. ISBN: 0130819344. ⊖
- [HTML401] *HTML 4.01 Specification*. World Wide Web Consortium. December 1999.
<<http://www.w3.org/TR/1999/REC-html401-19991224/>>.
- [J2SE-APIs] *Java™ 2 Platform, Standard Edition, v 1.3 API Specification*. Sun Microsystems. March 2001.
<<http://java.sun.com/j2se/1.3/docs/api/overview-summary.html>>.

⊖ 《设计模式》，中文版，机械工业出版社，2002。（07575）

⊖ 《Java平台安全技术》，中文版，机械工业出版社，1999。（07807）

⊖ 《最新Java核心技术，卷2，高级特性》，中文版，机械工业出版社，2003。（08244）

- [Jackson75] Jackson, M.A. *Principles of Program Design*, Academic Press, London, 1975. ISBN: 0123790506.
- [JavaBeans] *JavaBeans™ Spec*. Sun Microsystems. March 2001.
<<http://java.sun.com/products/javabeans/docs/spec.html>>.
- [Javadoc-a] *How to Write Doc Comments for Javadoc*. Sun Microsystems. January 2001.
<<http://java.sun.com/j2se/javadoc/writingdoccomments/>>.
- [Javadoc-b] *Javadoc Tool Home Page*. Sun Microsystems. January, 2001.
<<http://java.sun.com/j2se/javadoc/index.html>>.
- [JLS] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310082.
- [Kahan91] Kahan, William, and J. W. Thomas. *Augmenting a Programming Language with Complex Arithmetic*, UCB/CSD-91-667, University of California, Berkeley, 1991.
- [Knuth74] Knuth, Donald. Structured Programming with go to Statements. *Computing Surveys* 6 (1974): 261–301.
- [Lea01] *Overview of Package util.concurrent Release 1.3.0*. State University of New York, Oswego. January 12, 2001.
<<http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html>>.
- [Lea00] Lea, Doug. *Concurrent Programming in Java™: Design Principles and Patterns, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310090.
- [Lieberman86] Lieberman, Henry. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *Proceedings of the First ACM Conference on Object-Oriented Programming Systems*,

- Languages, and Applications*, pages 214–223, Portland, September 1986. ACM Press.
- [Meyers98] Meyers, Scott. *Effective C++*, Second Edition: 50 Specific Ways to Improve Your Programs and Designs. Addison-Wesley, Reading, MA, 1998. ISBN: 0201924889.
- [Parnas72] Parnas, D.L. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15 (1972): 1053–1058.
- [Posix] 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std. 1003.1 1995 Edition] Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language] (ANSI), IEEE Standards Press, ISBN: 1559375736.
- [Pugh01a] *The Java Memory Model*. Ed. William Pugh. University of Maryland. March 2001.
<http://www.cs.umd.edu/~pugh/java/memoryModel/>.
- [Pugh01b] *The “Double-Checked Locking is Broken” Declaration*. Ed. William Pugh. University of Maryland. March 2001.
<http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>.
- [Serialization] *Java™ Object Serialization Specification*. Sun Microsystems. March 2001.
<http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serialTOC.doc.html>.
- [Smith62] Smith, Robert. Algorithm 116 Complex Division. In *Communications of the ACM*, 5.8 (August 1962): 435.
- [Snyder86] Synder, Alan. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 38–45, 1986. ACM Press.

-
- [Thomas94] Thomas, Jim, and Jerome T. Coonen. Issues Regarding Imaginary Types for C and C++. In *The Journal of C Language Translation*, 5.3 (March 1994): 134–138.
- [Vermeulen00] Vermeulen, Allan, Scott W. Ambler, Greg Bumgardener, Eldon Metz, Trevor Mesfeldt, Jim Shur, Patrick Thompson. *The Elements of Java™ Style*, Cambridge University Press, Cambridge, United Kingdom, 2001. ISBN: 0521777682.
- [Weblint] *The Weblint Home Page*. Weblint.org. March 2001.
<<http://www.weblint.org/>>.
- [Wulf72] Wulf, W. A Case Against the GOTO. *Proceedings of the 25th ACM National Conference* 2 (1972): 791–797.

模式和习惯用法索引

本书索引所列页码，皆为英文原书页码。

A

adapter, 86, 92
adding an aspect, 31
antipattern
busy-wait, 204
constant interface, 89
discriminated union, 100
double-check idiom for lazy initialization, 193
empty catch block, 187
exception for loop control, 169
excessive string concatenation, 155
floating point for monetary calculation, 149
int enum, 104
null as substitute for zero-length array, 134–135
serializable class designed for extension, 215
serializable inner class, 217
string overuse, 152
unsynchronized concurrent access, 189–195
array printing, 147

B

busy-wait (*antipattern*), 204

C

capability, 153
class designed for inheritance, 78–83
class hierarchy, 101–103
`clone`, 45–52
collection iteration, 142
`compareTo`, 55–57
constant interface (*antipattern*), 89
constant utility class, 90
cooperative thread termination, 192
copy constructor, 51

D

decorator, 75
defensive copy, 122–125
defensive `readObject`, 224–229
defensive `readResolve`, 231
discriminated union (*antipattern*), 100
doc comment, 136–139
double-check idiom for lazy initialization (*antipattern*), 193

E

eliminating self-use, 83
empty catch block (*antipattern*), 187
encapsulated structure class, 98
`equals`, 32
exception chaining, 179
exception for loop control (*antipattern*), 169
exception translation, 178
excessive string concatenation (*antipattern*), 155
explicit serial version UID, 223
explicit termination method, 21–22

F

failure-capture, 183–184
finalizer chaining, 23
finalizer guardian, 23
floating point for monetary calculation (*antipattern*), 149
function object, 115, 127

H

`hashCode`, 38

I

immutable class, 63

initialize-on-demand holder class, 194
int enum (*antipattern*), 104
int enum, bit-flag variant of, 112
 interface as type, 156
 iteration, collection, 142
 iteration, high-performance list, 143

L

list iteration, high-performance, 143

M

mixin interface, 84

N

nonhierarchical type framework, 84
 noninstantiability enforcement, 12
 nonserializable stateful class allowing
 serializable subclass, 216
 null as substitute for zero-length array
 (*antipattern*), 134–135

O

open call, 198
 override-prevention method, 110

P

parameter check, 119–121
 private lock object, 210
 provider framework, 8

R

readObject, defensive, 224–229
readResolve, defensive, 231
 reflective instantiation with interface access,
 159

S

safe array access, 62
 self-use elimination, 83
 serializable class designed for extension
 (*antipattern*), 215
 serializable inner class (*antipattern*), 217
 simulated multiple inheritance, 87
 singleton, 10–11
 skeletal implementation, 85
 state-testing method, 170, 175

static factory method, 5–9
 strategy, 115
 string concatenation, excessive (*antipattern*),
 155
 string overuse (*antipattern*), 152

T

toString, 42–44
typesafe enum, 105
 behaviors attached to constants of, 108
 ordinal-based, 106
 serializable, extensible, 111

U

unsynchronized concurrent access
 (*antipattern*), 189–195
 utility class, 12

W

wait loop, 201
 wrapper class, 74

索引

A

abstract classes

- adding an aspect to, 31
- as replacement for discriminated union, 101
- designing for inheritance, 82
- evolution of, vs. interfaces, 88
- examples
 - adding behaviors to typesafe enum, 108
 - replacement for discriminated union, 101
 - skeletal implementation, 87
 - static member class, 94
- for adding behaviors to typesafe enum, 108
- for service provider framework, 8
- for skeletal implementations, 82, 85
- noninstantiability and, 12
- vs. interfaces, 84–88

access control, 60

access levels, 3

- method overriding and, 61
- of classes and interfaces, 60
- of constants, 61
- of members, 60
- of static member classes, 91
- readResolve** and, 232

accessor methods, 98

- defensive copies and, 63, 124

examples

- defensive copies, 124
- immutability, 64
- for failure-capture information, 173, 184
- for information returned by `toString`, 44
- for lower-level exception, 179
- immutability and, 63
- naming conventions for, 167
- vs. public fields, 98–99, 103

alien methods, 196

- deadlock and, 196–199

example, 197

safety failures and, 199

anonymous classes, 91, 93–94

as concrete strategy classes, 116

as function objects, 127

examples

- finalizer guardian**, 23
- in adapters, 86
- in typesafe enums, 94, 108
- finalizer guardian** and, 23
- in adapters, 86
- in typesafe enums, 108
- limitations of, 93
- uses of, 93

API

See exported API

API design

exceptions and, 170, 172

performance consequences of, 163

API elements, 4

documenting, 136–139

arrays, 3

defensive copying of, 62, 125

in public fields as security holes, 62

nonzero-length and mutability, 62, 125

zero-length and immutability, 134

zero-length vs. null as return value, 134–135

assertions, and parameter checking, 120

atomic data, 190

synchronization and, 190–192

auxiliary class, 91

B

base classes, 157

BigDecimal, for monetary calculations, 149–151

binary compatibility, 215

- bounded buffers, 202
 busy-wait, 204
- C**
- callback frameworks, 75
 callbacks, 75, 115
 canonical form, 34
 Chapter, 1
checked exceptions, 172
 accessor methods in, 173, 184
 avoiding unnecessary use of, 174–175
 documenting, 181
 failure atomicity and, 185
 ignoring, 187
 purpose of, 172
 transforming into unchecked exception, 175
- class hierarchies**, 84
 as replacement for discriminated unions, 100–103
 combinatorial explosion of, 85
- class-based frameworks, 157
- classes**, 3, 59–95
 access levels of, 60
 anonymous
See anonymous classes
 as replacement for C enums, 104–114
 as replacement for C structs, 97–99
 as replacement for function pointers, 115–117
 base, 157
 designing for inheritance, 79–83
 documenting, 136, 138
 for inheritance, 78–79
 thread safety of, 208–210
 helpers, for shortening parameter lists, 127
 hierarchies as replacement for discriminated unions, 100–103
 immutable
See immutability
 instance, 3
 levels of thread safety for, 208–209
 member classes, 3
 members, 3
 minimizing accessibility of, 59–62
 naming conventions for, 165–166
 singletons
See singletons
- stateful, 216
 stateless, 115
 unrelated, 55, 92, 131
 utility
See utility classes
- clients, 3
- clone**, 25, 45–52
 as a constructor, 48
 as another constructor, 81
 copy constructor as alternative to, 51–52
 defensive copies and, 62, 123–124, 228
 examples
 defensive copies, 62, 124, 224
 implementing, 46, 48, 49
 general contract of, 45–46
 immutable objects and, 66
 incompatibility with final fields, 48
 nonfinal methods and, 50, 81
 references to mutable objects and, 47–50
- Cloneable interface**, 45
 alternatives to, 51–52
 behavior of, 45
 designing for inheritance and, 81
 examples, 48, 49
 instructions for implementing, 51
 purpose of, 45
 combinatorial explosion, 85
 companion classes, mutable, 67
- Comparable interface**, 53–57
- Comparator**
 anonymous classes and, 93
 class, 56, 115
 examples, 93, 115, 116, 117
 instance, 93
 interface, 116
- compareTo**, 25, 53
 consistent or inconsistent with equals, 55
 differences from equals, 54
 examples, 56, 57, 106, 226
 overloading in, 132
 use of, 14
 using, 122, 123, 150, 224, 228
 general contract for, 53–55
 instructions for writing, 55–57
- composition**, 8, 73
 favor over inheritance, 71–77
- concrete strategy, 115

- concurrency**
 documenting method behavior for, 208–210
 fine-grained synchronization and, 200
 increasing for testing, 207
 open calls and, 199
 utilities to simplify multithreaded programming, 148, 189
- conditionally thread-safe**, 208
 denial-of-service attack and, 210
 documenting, 209
 documenting lock object for, 210
- consistency requirement**
 in `equals` contract, 26, 32
 in `hashCode` contract, 36
- consistency, data**
See data consistency
 consistent with `equals`, 55
 constant interface, 89
 constant utility class, 90
- constants**
 accessibility of, 61
 data types of, 61
 in interfaces, 89–90
 in typesafe enums, 104–114
 naming convention for, 166
- constructor**, 3
 calling overridable methods in, 80
`clone` as a, 48
`copy`, 51, 66
 default, 12
 defensive copying and, 123
 deserialization as a, 214
 documenting self-use, 78
 enforcing noninstantiability with, 12
 enforcing singleton property with, 10–11
 establishing invariants, 65, 70
 examples
 enforcing noninstantiability, 12
 for typesafe enums, 105
 in immutable classes, 122, 123
 in singletons, 10
 use of overridable methods in, 80
 for typesafe enums, 105, 109
 overloading, 130
 parameterless, 12, 215
`readObject` as a, 224
 replacing with static factory, 5–9
 signature of, 6
- cooperative thread termination, 192
`copy` constructor, 51, 66
- custom serialized form**, 213, 218–223
 example, 220
 transient fields and, 223
- D**
- data consistency**
 maintaining in face of failure, 185–186
 synchronization, 189–195
 unreliable resources and, 34
- deadlock, avoiding, 196, 199
- default access, 3, 60
- default constructor, 12
- default serialized form**
 criteria for accepting, 218
 disadvantages of, 220
 initial values of transient fields and, 223
`transient` modifier and, 220
- defaultReadObject**
 examples, 217, 221, 226, 228
 transient fields and, 221, 223
- defaultWriteObject**
 examples, 217, 221
 transient fields and, 221, 222
- defensive copy**, 122–125
`clone` and, 123, 124, 228
 deserialization and, 228
 immutable objects and, 66
 of array, 125
 of mutable input parameters, 123–124
 of mutable internal fields, 124
`readObject` and, 228
`readResolve` as alternative to, 231
 vs. object reuse, 16
- degenerate class, 98
- delegation, 75
- denial-of-service attack, 210
- deserialization**, 213–232
 as a constructor, 214
 preventing completion of, 226
 singletons and, 11
 typesafe enums and, 107
- detail message, 183
- discriminated unions**, 100
 class hierarchies as replacement for, 100–103

- distinguished return value, vs. state-testing method, 171
- doc comments, 136
- documentation**, 136–139
- @**param** tag, 136, 137
 - @**return** tag, 137
 - @**serial** tag, 219
 - @**serialData** tag, 222
 - @**throws** tag, 119, 136, 137, 181
 - for inheritance, 78–79
 - for serialized fields, 219
 - HTML in Javadoc, 137
 - inheritance of doc comments, 139
 - Javadoc, 136–139
 - links to architecture documents from Javadoc output, 139
 - moving descriptions of common exceptions to class comment, 182
 - of an object's state after an exception, 186
 - of conditional thread safety, 209
 - of exceptions, 181–182
 - of methods, 136
 - of parameter restrictions, 119
 - of postconditions, 136
 - of preconditions, 136, 181
 - of required locks, 209, 210
 - of return value of `toString`, 42
 - of self-use of overridable methods, 78, 82
 - of side effects, 136
 - of thread safety, 136, 208–210
 - of transfer of control, 125
 - of `writeObject` for serialization, 222
 - summary description in, 138
- synchronized** modifier and, 208
- documentation comments, 136
- double**, when to avoid, 149–151
- double-check idiom, 193, 193–195
- E**
- encapsulated structure class, 98
- encapsulation**, 59, 97
- broken by inheritance, 71
 - data fields and, 98
- enclosing instance**, 91
- anonymous classes and, 93
 - finalizer guardian and, 23
 - local class and, 94
 - nonstatic member class and, 91
- serialization and, 217
- enum**, classes as replacement for, 104–114
- equals**, 25–35
- accidental overloading of, 35
 - canonical forms and, 34
 - examples
 - accidental overloading, 35
 - as unsupported operation, 26
 - for typesafe enums, 110
 - forwarding, 132
 - general contract and, 32
 - general contract of, 28, 31, 32, 87
 - preventing overriding, 111
 - violation of general contract, 27, 29, 30
 - extending an abstract class and, 31
 - extending an instantiable class and, 30
 - general contract for, 26–32
 - how to write, 32
 - method-forwarding and, 132
 - overriding `hashCode` and, 34, 36–41
 - typesafe enums and, 26, 110
 - unreliable resources and, 34
 - when to override, 25–26
- errors, 172
- examples**
- A, B, C, 129
 - AbstractFoo, 216
 - AbstractMapEntry, 87
 - anonymous inner class, 93
 - BogusPeriod, 225
 - Calculator, 94
 - CaseInsensitiveString, 27
 - Circle, 101
 - CollectionClassifier, 128
 - ColorPoint, 29, 31
 - Comparator, 116
 - Complex, 64, 68
 - DeadlockQueue, 198
 - Degree, 138
 - DisplayQueue, 197
 - Elvis, 10, 230
 - Entry, 48, 49, 219, 221
 - ExtendedOperation, 112
 - Foo**
 - finalizer guardian, 23
 - provider framework, 8
 - serializable subclass of nonserializable class, 217

- FooHolder**, 194
HashTable, 48, 49
Host, 117
InstrumentedHashSet, 71
InstrumentedSet, 74
Key, 153
MutablePeriod, 226
MyIterator, 92
MySet, 92
Name, 218
Operation, 94, 108, 111
Overriding, 129
Period, 122, 224
Person, 13, 14
PhoneNumber, 36
PhysicalConstants, 89, 90
PlayingCard, 104
Point, 29, 98
Rectangle, 101
Shape, 101
Singer, 84
SingerSongwriter, 85
SongWriter, 85
Square, 102
Stack, 17, 47
StoppableThread, 191, 192
StringLengthComparator, 115, 116
StringList, 219, 220
StrLenCmp, 117
Sub, 81
Suit, 105, 106
Super, 80
ThreadLocal, 153, 154
UtilityClass, 12
WaitQueuePerf, 206
WordList, 53
WorkerThread, 197, 198, 205
WorkQueue, 196, 204
exception chaining, 178
exception translation, 121, 178
exceptions, 169–187
 avoidance of, 174–175, 180
 chaining of, 178
 checked vs. run-time, 172–173
 commonly used, 119, 177
 control flow and, 170
 defining methods on, 173
 detail messages for, 183–184
 documentation of, 181–182
 as part of method documentation, 136, 137
 failure-capture information and, 183
 favor standard, 176–177
 for exceptional conditions only, 169–171
 handling with
 ThreadGroup.uncaughtException, 211
 ignoring, 187
 performance and, 170
 purpose of
 checked, 172
 run-time, 172
 transforming checked into unchecked, 175
 translation, 121
 translation of, 178
 uncaught, during finalization, 21
 explicit termination method, 21–22
exported API, 3
 access levels and, 60
 constant interface pattern and, 89
 documentation comments and, 139
 member classes and, 93
 serialization and, 213
 synchronized modifier and, 208
extension, 3
 Cloneable interface and, 81
 of class hierarchies, 102
 of classes, 71, 82
 appropriateness, 76
 clone and, 46
 compareTo and, 55
 equals and, 30
 immutability and, 67
 private constructors and, 68
 Serializable interface and, 81
 of interfaces, 85
 of skeletal implementations, 86
 of typesafe enums, 109
 Serializable interface and, 215
extralinguistic mechanisms
 cloning, 45
 native methods, 161
 reflection, 158
 serialization, 214

F

failure atomicity, 121, 185–186

- fields**, 3
 access levels of, 60
`clone` and, 47
`compareTo` and, 56
 constant, 61
 constant interface pattern and, 89
 default values of, 223
 defensive copies of, 124
 documenting, 136, 138, 219
 encapsulation and, 98
`equals` and, 33
 exposing, 61, 98
`final`
See final fields
`hashCode` method and, 38
 immutability and, 63
 interface types and, 156
 naming conventions for, 166, 167
 protected, 79
 public, 61
 redundant, 34, 39
 reflection and, 158
 serialization and, 223
 stateless classes and, 115
 synthetic, 217
 thread safety and, 61
 transient
See transient fields
final fields
 constant interface pattern and, 89
 constants and, 61, 166
 incompatibility with `clone`, 48
 incompatibility with defensive copy in `readObject`, 228
`readResolve` and, 231
 references to mutable objects and, 61
 to implement singletons, 10
 typesafe enums and, 105
 finalizer chaining, 22
 finalizer guardian, 23
finalizers, 20–24
 chaining of, 22
 critical persistent state and, 21
 execution time, 20
 uses for, 22
 vs. explicit termination method, 21–22
 fine-grained synchronization, 200
`Float.compareTo` inconsistent with `equals`, 55
`float`, when to avoid, 149–151
`for` loop, prefer to `while` loop, 142
forwarding, 73
 to private extension of skeletal implementation, 86
forwarding methods, 73
 ensuring consistent behavior in overloaded methods, 132
 example, 74
 in wrapper classes, 75, 200
 object composition and, 73
frameworks
 callback, 75
 class-based, 157
 Collections Framework, 147
 interface-based, 7
 nonhierarchical type, 84
 object serialization, 213
 service provider, 7
 function objects, 93, 115–117, 127
 function pointers, 115–117
 functional approach, 65
- G**
- general contract**, 25, 181
`clone`, 45
`compareTo`, 53
`equals`, 26
`hashCode`, 36
 implementing interfaces and, 84
`toString`, 42
- H**
- handoff, 125
hashCode, 25
 general contract for, 36
 how to write, 38
 immutable objects and, 40
 lazy initialization and, 40, 69
 overriding `equals` and, 34, 36–41
 typesafe enums and, 110
 heap profiler, 19
 helper classes, 127

I

immutability, 63–70
 advantages of, 65
 and `hashCode`, 40
 canonical forms and, 34
`Cloneable` and, 51
 constants and, 61, 166
 defensive copies and, 122, 124
 disadvantage of, 66
 examples, 64
 broken, 122
 serialization and, 224
 static factories and, 68
 failure atomicity and, 185
 functional approach and, 65
 object reuse and, 13
`readObject` and, 224–228
`readResolve` and, 232
 rules for, 63
 serialization and, 70, 224–229
 static factory methods and, 67
 thread safety and, 208
 zero length arrays and, 134
immutable, as a level of thread safety, 208
implementation inheritance, 71
implements, 3
inconsistent with equals, 55
information hiding, 59, 162
inheritance, 3
 and encapsulation, 71
 causes of fragility in, 73
 designing for, 79–83
 documenting for, 78–79
 eliminating self-use of overridable methods
 for, 83
 example, 71
 favor composition over, 71–77
 hooks to facilitate, 79
 implementation vs. interface, 71
 incompatibility with `readResolve`, 232
 of doc-comments, 139
 overridable methods and, 80
 prohibiting, 82
See also extension
 serialization and, 215
 uses of, 76
 using internal objects for locking and, 210

initialization

at object creation, 70
 defensive copying and, 63
 examples, 194
 of double-check idiom, 193
 of lazy, 40, 69
 of loop variables, 142
 of static, 14, 194
 to allow serializable subclasses, 216
initialize-on-demand holder class and, 194
 lazy, 7, 15, 40, 69, 193
 of fields on deserialization, 223
 of local variables, 142
 static, 14
 to allow serializable subclasses, 216
 viewing objects before complete, 81, 193
initialize-on-demand holder class, 194
inner classes, 91
 and serialization, 217
 anonymous
 See anonymous classes
 extending skeletal implementations with, 86
instance-controlled classes, 231
`readResolve` and, 231
 singleton, 10
 typesafe enum, 105
 utility classes, 12
int, use for monetary calculations, 149–151
interface inheritance, 71
interface-based frameworks, 7, 84–88
interfaces, 3, 59–95
 access levels of, 60
 as parameter types, 127
`Cloneable`, 45–52
`Comparable`, 53–57
 constant, 89–90
 documenting, 136, 138, 181
 evolving, 88
 extending `Serializable`, 215
 for defining mixins, 84
 for nonhierarchical type frameworks, 84
 for type definition, 89–90
 member, 3
 mixin, 45, 84
 naming conventions for, 165–166
 purpose of, 45, 89–90
`Serializable`, 213–217
 skeletal implementations and, 85–88

strategy, 116
 to enable functionality enhancements, 85
 to refer to objects, 156–157
 vs. abstract classes, 84–88
 vs. reflection, 158–160

J**JavaBeans**

serialization and, 213

XML and, 215

Javadoc, 136**L**

lazy initialization, 7, 15, 40, 69
 double-check idiom for, 193
 initialize-on-demand holder class for, 194

libraries, 145–148

for multithreaded programming, 189

liveness, ensuring, 196–200, 201

thread priorities and, 206

local classes, 91, 94**local variables**, 141

declaring, 141

initializing, 142

minimizing scope of, 141–144

naming conventions for, 166, 167

locks

for classes at various levels of thread safety, 208

in multithreaded programs, 189–195

recursive, 199

using private objects for, 210

logging, 180**long**, use for monetary calculations, 149–151**loop variables**, 142**loops**

for invoking `wait`, 201–203

minimizing scope of variables and, 142

prefer `for` to `while`, 142

M**member classes**, 3

nonstatic

See nonstatic member classes

static

See static member classes

static vs. nonstatic, 91–95

members, 3

accessibility of, 60

minimizing accessibility of, 59–62

memory leaks, 18

causes of, 18–19

example, 17

memory model, 63, 190**methods**, 3, 119–139

access levels of, 60, 61

accessor, vs. public fields, 98–99

adding to exception classes, 173

alien

See alien methods

checking parameters for validity, 119–121

common to all objects, 25–57

defensive copying before parameter

checking, 123

designing signatures of, 126–127

documenting, 136–137, 138

exceptions thrown by, 181–182

overridable, 78

thread safety of, 208–210

explicit termination, 21–22

failure atomicity and, 185–186

forwarding

See forwarding methods

immutability and overriding, 67

inheriting doc comments, 139

naming conventions for, 9, 166, 167

naming of, 126

native, 22, 161

ordering computation in, 185

overloadings, 128–133

same number of parameters in, 130–132

static selection among, 128

overridings, 128–129

dynamic selection among, 128

parameter lists for, 126

small and focused, 144

state-testing, vs. distinguished return value,

171

static factory

See static factory methods

mixin interfaces, 45, 84**module**, 2

Monty Python reference, subtle, 134

mutable companion classes, 67

mutual exclusion, 189

N

naming conventions, 9, 165, 165–168
 native methods, 22, 161
 native peers, 22
 nested classes, 60, 91–95
 access levels of, 60
 as concrete strategy classes, 117
 nonhierarchical type frameworks, 84
 noninstantiability, 12
 non-nullity
 in general contract of `compareTo`, 55
 in general contract of `equals`, 32
 nonstatic member classes, 91
 example, 92
 for defining adapters, 92
 prefer static member classes to, 91–95
notify vs. **notifyAll**, 202–203

O

object pool, 16
 objects, 3
 avoiding reflective access, 158–160
 creating and destroying, 5–24
 deserializing, 213–232
 eliminating obsolete references to, 17–19
 function, 93
 immutable
 See immutability
 methods common to all, 25–57
 process, 93
 reuse, 13–16
 serializing, 213–232
 using base classes to refer to, 157
 using interfaces to refer to, 156–157
 viewing in partially initialized state, 81, 193
 obsolete object references, 17–19, 47, 185
 optimizations, 162–164
 `==` instead of `equals`, 6, 33
 initialization and, 194
notify vs. **notifyAll**, 202
 object reuse, 13–16
 static initialization, 14–15
StringBuffer and, 155
try-catch blocks and, 170
 overloading
 See methods, overloading
 overriding

See methods, overriding

P

package-private
 access level, 3, 60
 constructors, 67, 82
 packages, naming conventions for, 165, 166
@param tag, 136, 137
 parameter lists, 126–127
 of constructors, 6
 parameterless constructor, 12, 215
 parameters, checking for validity, 119–121
 performance
 See optimizations
 performance model, 164
 postconditions, 136
 preconditions, 136, 181
 primitives, 3
 `compareTo` and, 56
 `equals` and, 33
 `hashCode` and, 38
 process objects, 93
 protected, 61

R

radically different types, 131
readObject, 224–229
 acceptability of default, 229
 as another constructor, 224
 default serialized form and, 219, 223
 defensive, 224–229
 defensive copying in, 228
 final fields and, 228
 for immutable objects, 70
 incompatibility with singletons, 230
 overridable methods and, 81, 229
 readResolve as alternative to, 231
 transient fields and, 223
readResolve
 access levels of, 82, 112, 232
 as alternative to **readObject**, 231
 examples, 107, 111, 112, 230, 231
 for immutable objects, 70
 for instance-controlled classes, 231
 for singletons, 11, 230
 for typesafe enums, 107, 110
 when to provide, 230–232

- read-write locks, 202
- recipes**
 - `clone`, 51
 - `compareTo`, 55–57
 - composition, 73
 - `equals`, 32
 - finalizer guardian, 23
 - `hashCode`, 38
 - invoking `wait`, 201
 - noninstantiable classes, 12
 - `readObject`, 229
 - singletons, 10
 - static factory method, 5
 - typesafe enums
 - extensible, 109
 - extensible serializable, 110
 - serializable, 107
 - simple, 105
- recovery code, 186
- recursive lock, 199
- redundant fields, 34, 39
- reference types**, 3
 - `compareTo` and, 56
 - `equals` and, 32
 - `hashCode` and, 38
- reflection**
 - `clone` method and, 45
 - drawbacks of, 158, 160
 - for object creation, 159
 - original purpose of, 158
 - to break run-time dependencies, 160
 - vs. interfaces, 158–160
- reflexivity**
 - `compareTo` and, 55
 - `equals` and, 27
- @return tag**, 137
- run-time exceptions**, 172
 - documenting, 136
 - vs. checked exceptions, 172–173
- S**
- safety**
 - ensuring, 189–195
 - failures, 195, 199
 - `wait` and, 201
- scope of variables**
 - local, 141–144
- loop, 142
- obsolete references and, 18
- self-use**
 - documenting, for inheritance, 78
 - eliminating, for inheritance, 82
- semaphores, 202
- @serial tag**, 219
- serial version UIDs**, 214
 - declaring in serializable classes, 223
- @serialData tag**, 222
- serialization**, 213–232
 - costs of, 213–215
 - documenting for, 219, 222
 - effect on exported APIs, 213
 - extralinguistic mechanism, 214
 - immutability and, 70
 - inheritance and, 215
 - inner classes and, 217
 - interface extension and, 215
 - JavaBeans and, 213
- serialized form**
 - as part of exported API, 213
 - custom
 - See* custom serialized form
 - default
 - See* default serialized form
- defaultWriteObject** and, 221
- documenting, 219
 - of inner class, 217
 - of singletons, 231
 - of typesafe enum, 107
- serialver utility**, 223
- service provider framework**, 7
 - reflection and, 160
- signature, 3, 126–127
- simulated multiple inheritance, 87
- singletons**, 10
 - deserialization and, 11
 - enforcing with a private constructor, 10–11
 - incompatibility with `readObject`, 230
 - `readResolve` and, 11, 230
 - serialized form of, 231
- skeletal implementation, 85, 178
- specific notification, 203
- state transition, 65, 189
- state-testing method, 170–171, 175

- static factory methods**, 5, 11, 13
 advantages over constructors, 5
 and immutable objects, 66
 anonymous classes within, 93
 as basis of service provider frameworks, 7
 as replacement for cloning, 51
 as replacement for constructors, 5–9
 disadvantages, compared to constructors, 8
 flexibility of, 6
 for immutable objects, 67
 for instance-controlled classes, 6
 naming conventions for, 9
 strategy pattern and, 117
- static fields**
 immutable objects and, 11
 initialize-on-demand holder class and, 194
 strategy pattern and, 117
 synchronization of mutable, 200
- static initializer, 14
- static member classes**, 91
 and serialization, 217
 common uses of, 91, 92, 94
 for shortening parameter lists, 127
 prefer to nonstatic member classes, 91–95
 prefer to strings for representing an aggregate, 152
 to implement strategy pattern, 117
 to implement typesafe enums, 109
- strategy interface, 116
- stream unique identifiers, 214, 223
- string concatenation, 155
- string representation, 42–44
- strings, as inappropriate substitutes for other types, 152–154
- structure, class as replacement for, 97–99
- subclassing**, 3
 access levels of methods and, 61
equals and, 28, 31
 of `RuntimeException` vs. `Error`, 172
 prohibiting, 8, 12, 82
See inheritance
- summary description, 138
- symmetry**
`compareTo` and, 55
`equals` and, 27
- synchronization**
 fine-grained, 200
 for mutual exclusion, 189
 for shared mutable data, 189–195
 for thread communication, 190–195
 internal, 200
 of atomic data, 190, 192
 performance and, 196, 199
 providing in a subclass of an unsynchronized class, 200
- synchronized modifier**
 as an implementation detail, 208
 documentation and, 208
 purpose of, 189
 synthetic fields, 217
- T**
- thread groups, 211
- thread priorities, 206
- thread safety**, 210
 and immutability, 65
 documenting, 208–210
 levels of, 208
`ThreadGroup` API and, 211
- thread scheduler, 204–207
- thread termination, cooperative, 192
- Thread.yield**, 206
 testing and, 207
- thread-compatible**, 209
 vs. thread-safe, 200
- thread-hostile, 209
- threads, 189–211
- thread-safe**, 208
 vs. thread-compatible, 200
- thread-safety**
 classes with public mutable fields and, 61
 immutable objects and, 65
- `@throws` tag, 119, 136, 137, 181
- toString**, 25
 documenting return value of, 42–44
 exceptions and, 183
 general contract for, 42
 overriding, 42–44, 106
 providing programmatic access to data
 returned by, 44
 return value as a de facto API, 44
- transient fields**, 220
 custom serialized form and, 223
- `defaultReadObject` and, 221, 223

- defaultWriteObject** and, 221, 222
deserialization and, 223
examples, 111, 220
 logical state of an object and, 223
readResolve and, 107
singletons and, 231
transient modifier, 220
- transitivity**
compareTo and, 55
equals and, 28
- translation**
 of exceptions, 121, 178
- types**
 radically different, 131
- typesafe enums**, 105
 adding behaviors to, 106–109
 anonymous classes and, 108
 as replacement for C enum, 104–114
 constructors for, 105
 disadvantages of, 112
equals and, 26, 110
hashCode and, 110
 implementing with top-level class vs. static member class, 109
readResolve and, 231
- variants**
comparable, 106
extensible, 109
extensible serializable, 110
serializable, 107
- U**
- unchecked exceptions**, 172, 174
 documenting, 136, 181
 idiom for highlighting, 181
 ignoring, 187
 standard, 176
 transforming checked exceptions into, 175
 vs. checked exceptions, 172
- unintentional object retentions**
See memory leaks
- unintentionally instantiable classes, 12
- unions, class hierarchies as replacement for,
- 100–103
user, 3
util.concurrent, 148, 189
- utility classes**, 12
 as alternative to constant interfaces, 90
 in Collections Framework, 54
- V**
- variables**
 atomic operations on, 190
 declaring with interface types, 156
local, 141
See also local variable
loop, 142
- views**
 as use of nonstatic member classes, 92
 locking for, 209
 naming conventions for, 167
object reuse and, 15
 to avoid subclassing, 31, 55
 to maintain immutability, 125
- volatile modifier**, 192
 examples, 40, 69, 205
 vs. subclassing, 82, 85
- W**
- wait loop**, 201–203
while loop, prefer **for** loop to, 142
- wrapper classes**, 74–75
 as alternative to constant interfaces, 89
 for synchronization, 200
 for synchronization of unsynchronized classes, 200
- X**
- XML**, for JavaBean persistence, 215
- Z**
- zero-length arrays**
 immutability of, 134
 vs. null as return value, 134–135