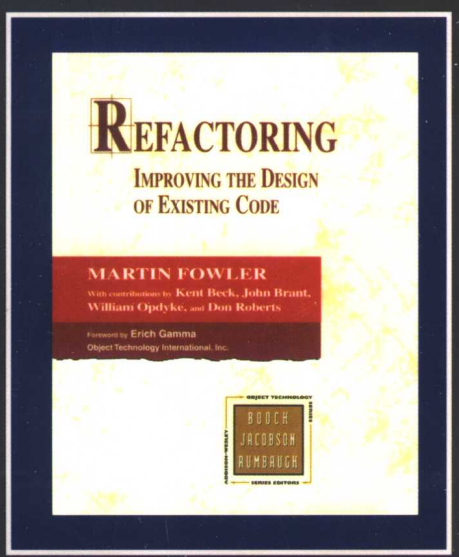
软 件 工 程 系 列

PEARSON
Addison Wesley

Refactoring: Improving the Design of Existing Code

重构——改善既有代码的设计（中文版）

[美]Martin Fowler 著
侯捷 熊节 译



- 与《设计模式》齐名的经典巨著
- 《设计模式》作者 Erich Gamma 为本书作序
- 超过 70 种行之有效的重构方法

Pearson Education

中国电力出版社
www.infopower.com.cn

重构

— 改善既有代码的设计 —

Refactoring

Improving the Design of Existing Code

Martin Fowler 著

（以及 Kent Beck, John Brant , William Opdyke,
Don Roberts 对最后三章的贡献）

侯捷 / 熊节 合译

Refactoring: Improving the Design of Existing Code (ISBN 0-201-48567-2)
Martin Fowler

图书在版编目（CIP）数据

重构：改善既有代码的设计 /（美）福勒（Fowler, M.）著；侯捷，熊节译.
—北京：中国电力出版社，2003（软件工程系列）
ISBN 7-5083-1554-5
Ⅰ.重... Ⅱ.①福...②侯...③熊... Ⅲ.代码－程序设计 Ⅳ.TP311.11
中国版本图书馆 CIP 数据核字（2003）第 057350 号

责任编辑：程璐 乔晶
书　　名：重构：改善既有代码的设计
原　　著：（美）Martin Fowler
翻　　译：侯捷 熊节
出版发行：中国电力出版社
地址：北京市三里河路6号 邮政编码：100044
电话：（010）88515918 传真：（010）88518169
印　　刷：汇鑫印务有限公司印刷
开　　本：787×1092 1/16　　**印　　张：**29　　**字　　数：**540 千字
书　　号：ISBN 7-5083-1554-5
版　　次：2003年8月北京第一版
印　　次：2003年8月第一次印刷
定　　价：68.00 元

Refactorings（重构）列表

译序
by 侯捷

看过铁路道班工人吗？提着手持式砸道机，机身带着钝钝扁扁的钻头，在铁道上、枕木间卖力地「砍劈钻凿」。他们在做什么？他们在使路基上的碎石块（道碴）因持续剧烈的震动而翻转方向、滑动位置，甚至震碎为更小石块填满缝隙，以求道碴更紧密契合，提供铁道更安全更强固的体质。

当「重构」（refactoring）映入眼帘，我的大脑牵动「道班工人+电动砸道机+枕木道碴」这样一幅联想画面。「重构」一词非常清楚地说明了它自身的意义和价值：在不破坏可察功能的前提下，借由搬移、提炼、打散、凝聚…，改善事物的体质。很多人认同这样一个信念：「非常的建设需要非常的破坏」，但是现役的应用软件、构筑过半的项目、运转中的系统，容不得推倒重来。这时候，在不破坏可察功能的前提下改善体质、强化当前的可读性、为将来的扩充性和维护性做准备、乃至于在过程中找出潜伏的「臭虫」，就成了大受欢迎的稳步前进的良方。

作为一个程序员，任谁都有看不顺眼手上代码的经验 — 代码来自你邻桌那个菜鸟，或三个月前的自己。面临此境，有人选择得过且过；然而根据我对「程序员」人格特质的了解，更多人盼望插手整顿。挽起袖子剑及履及，其勇可嘉，其虑未缜。过去或许不得不暴虎凭河，忍受风险。现在，有了严谨的重构准则和严密的重构手法，「稳定中求发展」终于有了保障。

是的，把重构的概念和想法逐一落实在严谨的准则和严密的手法之中，正是这本《*Refactoring*》的最大贡献。重构?! 呵呵，上进的程序员每天的进行式，从来不新鲜，但要强力保证「维持程序原有的可察功能，不带进新臭虫」，重构就不能是一项靠着天份挥洒的艺术，必须是一项工程。

我对本书的看法

初初阅读本书，屡屡感觉书中所列的许多重构目标过于平淡，重构步骤过于琐屑。这些我们平常也都做、习惯大气挥洒的动作，何必以近乎枯燥的过程小步前进？然后，渐渐我才体会，正是这样的小步与缓步前进，不过激，不躁进，再加上完整的测试配套（是的，测试之于重构极其重要），才是「不带来破坏，不引入臭虫」的最佳保障。我个人其实不敢置信有谁能够乖乖地按步遵循实现本书所列诸多被我（从人的角度）认为平淡而琐屑的重构步骤。我个人认为，本书的最大价值，除了呼吁对软件质量的追求态度，以及对重构「工程性」的认识，最终最重要的价值还在于：建立起吾人对于「目前和未来之自动化重构工具」的基本理论和实现技术上的认识与信赖。人类眼中平淡琐屑的步骤，正是自动化重构工具的基础。机器缺乏人类的「大局观」智慧，机器需要的正是切割为一个一个极小步骤的指令。一板一眼，一次一点点，这正是机器所需要的，也正是机器的专长。

本书第 14 章提到，Smalltalk 开发环境已含自动化重构工具。我并非 Smalltalk guy，我没有用过这些工具。基于技术的飞快滚动（或我个人的孤陋寡闻），或许如今你已经可以在 Java, C++ 等面向对象编程环境中找到这一类自动化重构工具。

软件技术圈内，重构（refactoring）常常被拿来与设计模式（design patterns）并论。书籍市场上，《*Refactoring*》也与《*Design Patterns*》齐名。GoF 曾经说『设计模式为重构提供了目标』，但本书作者 Martin 亦言『本书并没有提供助你完成所有知名模式的重构手法，甚至连 GoF 的 23 个知名模式都没有能够全部覆盖。』我们可以从这些话中理解技术的方向，以及书籍所反映的局限。我并不完全赞同 Martin 所言『哪怕你手上有一个糟糕的设计或甚至一团混乱，你也可以借由重构将它加工成设计良好的代码。』但我十分同意 Martin 说『你会发现所谓设计不再是一切动作的前提，而是在整个开发过程中逐渐浮现出来。』我比较担心，阅历不足的程序员在读过本书后可能发酵出「先动手再说，死活可重构」的心态，轻忽了事前优秀设计的重要性。任何技术上的说法都必须有基本假设；虽然重构（或更向上说 XP，eXtreme Programming）的精神的确是「不妨先动手」，但若草率行事，代价还是很高的。重型开发和轻型开发各有所长，各有应用，世间并无万应灵药，任何东西都不能极端。过犹不及，皆不可取！

当然，「重构工程」与「自动化重构工具」可为我们带来相当大幅度的软件质量提升，这一点我毫无异议，并且非常期待☺。

关于本书制作

此书在翻译与制作上保留了所有坏味道（bad smell）、重构（refactoring）、设计模式（design patterns）的英文名称，并表现以特殊字体；只在封面内页、目录、小节标题中相应地给出一个根据字面或技术意义而做的中文译名。各种「坏味道」名称尽量就其意义选用负面字眼，如泥团、夸夸、过长、过大、过多、情结、偏执、惊悚、狎昵、纯稚、冗赘…。这些其实都是助忆之用，与茶余饭后的谈资（以及读者批评的根据☺）。

原书各小节并无序号。为参考、检索或讨论时的方便，我为译本加上了序号。

本书保留相当份量的英文术语，时而英中并陈（英文为主，中文为辅）。这么做的考量是，本书读者不可能不知道 class, final, reference, public, package...这些简短的、与 Java 编程息息相关的用词。另一方面，我确实认为，中文书内保留经过挑选的某些英文术语，有利于整体阅读效果。

两个需要特别说明的用词是 Java 编程界惯用的 "field" 和 "method"。它们相当于 C++ 的 "data member" 和 "member function"。由于出现次数实在频繁，为降低中英夹杂程度，我把它们分别译为「值域」和「函数」— 如果将 "method" 译为「方法」，恐怕术语突出性不高。本书将 "type" 译为「型别」而非「类型」，亦是为了中文术语之突出性；"instance" 译为「实体」而非「实例」、"argument" 译为「引数」而非「实参」，有意义上的考量。「*static* 值域与 *reference* 值域」、「*reference* 对象与 *value* 对象」等等则保留部分英文，并选用如上的特殊字体。凡此种种，相信一进入书中您很快可以感受本书术语风格。

本书还有诸多地方采中英并陈（中文为主，英文为辅）方式，意在告诉读者，我们（译者）深知自己的不足与局限，惟恐造成您对中译名词的误解或不习惯，所以附上原文。

中文版（本书）已将英文版截至 2003/06/18 为止之勘误，修正于纸本。

一点点感想

Martin Fowler 表现于原书的写作风格是：简洁，爱用代名词和略称。这使得读者往往需要在字面上揣度推敲。我期盼（并相信）经过技术意义的反刍、中英术语的并陈、中文表述的努力，中文版（本书）在阅读时间、理解时间和记忆深度上，较之英文版，能够为以华文为母语的读者提高 10 倍以上的成效。

本书由我和熊节先生合译。熊节负责第一个 pass，我负责后继工作。中文版（本书）为读者带来的阅读和理解上的效益，熊节居于首功 — 虽说做的是第一个 pass，我从初稿质量便可看出他多次反复推敲和文字琢磨的刻痕。至于整体风格、中英术语的选定、版面的呈现、乃至于全盘技术内涵的表现，如果有任何差错，责任都是我的☺。

作为一个信息技术教育者，以及一个信息技术传播者，我在超过 10 年的写译历程中，观察了不同级别的技术书品在读书市场上的兴衰起伏。这些适可反映大环境下技术从业人员及学子们的某些面向和取向。我很高兴看到我们的中文技术书籍（著译皆含）从早期盈盈满满的初阶语言用书，逐渐进化到中高阶语言用书、操作系统、技术内核、程序库/框架、再至设计/分析、软件工程。我很高兴看到这样的变化，我很高兴看到《*Design Patterns*》、《*Refactoring*》、《*Agile...*》、《*UML...*》、《*XP...*》之类的书在中文书籍市场中现身，并期盼它们有丰富的读者。

中文版（本书）支援网站有一个「术语 英中繁简」对照表。如果您有需要，欢迎访问，网址如下，并欢迎给我任何意见。谢谢。

侯捷 2003/06/18 于台湾.新竹

jjhou@jjhou.com（电子邮箱）
http://www.jjhou.com（繁体）（术语对照表 http://www.jjhou.com/terms.htm）
http://jjhou.csdn.net（简体）（术语对照表 http:// jjhou.csdn.net/terms.htm）

译序

by 熊节

重构的生活方式

还记得那一天，当我把《重构》的全部译稿整理完毕，发送给侯老师时，心里竟然不经意地有了一丝惘然。我是一只习惯的动物，总是安于一种习惯的生活方式。在那之前的很长一段时间里，习惯了每天晚上翻译这本书，习惯了随手把问题写成mail发给Martin Fowler先生，习惯了阅读Martin及时而耐心的回信，习惯了在那本复印的、略显粗糙的书本上勾勾画画，习惯了躺在床上咀嚼回味那些带有一点点英国绅士矜持口吻的词句，习惯了背后嗡嗡作响的老空调…当深秋的风再次染红了香山的叶，这种生活方式也就告一段落了。

只有几位相熟的朋友知道我在翻译这本书，他们不太明白为什么常把经济学挂在嘴边的我会乐于帮侯老师翻译这本书 — 我自己也不明白，大概只能用爱好来解释吧。既然已经衣食无忧，既然还有一点属于自己的时间，能够亲手把这本《重构》翻译出来，也算是给自己的一个交代。

第一次听到「重构」这个词，是在2001年10月。在当时，它的思想足以令我感到震撼。软件自有其美感所在。软件工程希望建立完美的需求与设计，按照既有的规范编写标准划一的代码，这是结构的美；快速迭代和RAD颠覆「全知全能」的神话，用近乎刀劈斧砍（crack）的方式解决问题，在混沌的循环往复中实现需求，这是解构的美；而Kent Beck与Martin Fowler两人站在一起，XP那敏捷而又严谨的方法论演绎了重构的美 — 我不知道是谁最初把refactoring一词翻译为「重构」，或许无心插柳，却成了点睛之笔。

我一直是设计模式的爱好者。曾经在我的思想中，软件开发应该有一个「理想国」— 当然，在这个理想国维持着完美秩序的，不是哲学家，而是模式。设计模式给我们

的，不仅仅是一些问题的解决方案，更有追求完美「理型」的渴望。但是，Joshua Kerievsky 在那篇著名的《模式与 XP》（收录于《极限编程研究》一书）中明白地指出：在设计前期使用模式常常导致过度工程（over-engineering）。这是一个残酷的现实，单凭对完美的追求无法写出实用的代码，而「实用」是软件压倒一切的要素。从一篇《停止过度工程》开始，Joshua 撰写了"Refactoring to Patterns"系列文章。这位犹太人用他民族性的睿智头脑，敏锐地发现了软件的后结构主义道路。而让设计模式在飞速变化的 Internet 时代重新闪现光辉的，又是重构的力量。

在一篇流传甚广的帖子里，有人把《重构》与《设计模式》并列为「Java 行业的圣经」。在我看来这种并列其实并不准确。实际上，尽管我如此喜爱这本《重构》，但自从完成翻译之后，我再也没有读过它。不，不是因为我已经对它烂熟于心，而是因为重构已经变成了我的另一种生活方式，变成了我每天的「面包与黄油」，变成了我们整个团队的空气与水，以至于无须再到书中寻找任何「神谕」。而《设计模式》，我倒是放在手边时常翻阅，因为总是记得不那么真切。

所以，在你开始阅读本书之前，我有两个建议要给你：首先，把你的敬畏扔到大西洋里去，对于即将变得像空气与水一样普通的技术，你无须对它敬畏；其次，找到合适的开发工具（如果你和我一样是 Java 人，那么这个「合适的工具」就是 Eclipse），学会使用其中的自动测试和重构功能，然后再尝试使用本书介绍的任何技术。懒惰是程序员的美德之一，绝不要因为这本书让你变得勤快。

最后，即使你完全掌握了这本书中的所有东西，也千万不要跟别人吹嘘。在我们的团队里，程序员常常会说：『如果没有单元测试和重构，我没办法写代码。』

好了，感谢你耗费一点点的时间来倾听我现在对重构、对这本《重构》的想法。Martin Fowler 经常说，花一点时间来重构是值得的，希望你会觉得花一点时间看我的文字也是值得的。

熊节 2003 年 6 月 11 日
夜 杭州

P.S. 我想借这个难得的机会感谢一个人：我亲爱的女友马姗姗。在北京的日子里，是她陪伴着我度过每个日日夜夜，照顾我的生活，使我能够有精力做些喜欢的事（包括翻译这本书）。当我埋头在屏幕前敲打键盘时，当我抱着书本冥思苦想时，她无私地容忍了我的痴迷与冷淡。谢谢你，姗姗，我永远爱你。

序言
by Erich Gamma

重构（**refactoring**）这个概念来自 Smalltalk 圈子，没多久就进入了其他语言阵营之中。由于重构是 framework（框架）开发中不可缺少的一部分，所以当 framework 开发人员讨论自己的工作时，这个术语就诞生了。当他们精炼自己的 class hierarchies（类阶层体系）时，当他们叫喊自己可以拿掉多少多少行代码时，重构的概念慢慢浮出水面。framework 设计者知道，这东西不可能一开始就完全正确，它将随着设计者的经验成长而进化；他们也知道，代码被阅读和被修改的次数远远多于它被编写的次数。保持代码易读、易修改的关键，就是重构 — 对 framework 而言如此，对一般软件也如此。

好极了，还有什么问题吗？很显然：重构具有风险。它必须修改运作中的程序，这可能引入一些幽微的错误。如果重构方式不恰当，可能毁掉你数天甚至数星期的成果。如果重构时不做好准备，不遵守规则，风险就更大。你挖掘自己的代码，很快发现了一些值得修改的地方，于是你挖得更深。挖得愈深，找到的重构机会就越多…于是你的修改也愈多。最后你给自己挖了个大坑，却爬不出去了。为了避免自掘坟墓，重构必须系统化进行。我在《*Design Patterns*》书中和另外三位（协同）作者曾经提过：design patterns（设计模式）为 refactoring（重构）提供了目标。然而「确定目标」只是问题的一部分而已，改造程序以达目标，是另一个难题。

Martin Fowler 和本书另几位作者清楚揭示了重构过程，他们为面向对象软件开发所做的贡献，难以衡量。本书解释重构的原理（principles）和最佳实践方式（best practices），并指出何时何地你应该开始挖掘你的代码以求改善。本书的核心是一份完整的重构名录（catalog of refactoring），其中每一项都介绍一种经过实证的代码变换手法（code transformation）的动机和技术。某些项目如 *Extract Method* 和

*Move Field*看起来可能很浅显，但不要掉以轻心，因为理解这类技术正是有条不紊地进行重构的关键。本书所提的这些重构准则将帮助你一次一小步地修改你的代码，这就减少了过程中的风险。很快你就会把这些重构准则和其名称加入自己的开发词典中，并且朗朗上口。

我第一次体验有纪律的、一次一小步的重构，是在30000英呎高空和Kent Beck共同编写程序（译注：原文为pair-programming，应该指的是*eXtreme Programming*中的所谓「成对/搭档 编程」）。我们运用本书收录的重构准则，保证每次只走一步。最后，我对这种实践方式的效果感到十分惊讶。我不但对最后结果更有信心，而且开发压力也小了很多。所以，我高度推荐你试试这些重构准则，你和你的程序都将因此更美好。

— *Erich Gamma*

Object Technology International, Inc.

前言

by Martin Fowler

从前，有位咨询顾问参访一个开发项目。系统核心是个 class hierarchy（类阶层体系），顾问看了开发人员所写的一些代码。他发现整个体系相当凌乱，上层 classes 对于 classes 的运作做了一些假设，下层（继承）classes 实现这些假设。但是这些假设并不适合所有 subclasses，导致覆写（overridden）行为非常繁重。只要在 superclass 内做点修改，就可以减少许多覆写必要。在另一些地方，superclass 的某些意图并未被良好理解，因此其中某些行为在 subclasses 内重复出现。还有一些地方，好几个 subclasses 做相同的事情，其实可以把它们搬到 class hierarchy 的上层去做。

这位顾问于是建议项目经理看看这些代码，把它们整理一下，但是经理并不热衷于此，毕竟程序看上去还可以运行，而且项目面临很大的进度压力。于是经理说，晚些时候再抽时间做这些整理工作。

顾问也把他的想法告诉了在这个 class hierarchy 上工作的程序员，告诉他们可能发生的事情。程序员都很敏锐，马上就看出问题的严重性。他们知道这并不全是他们的错，有时候的确需要借助外力才能发现问题。程序员立刻用了一两天的时间整理好这个 class hierarchy，并删掉了其中一半代码，功能毫发无损。他们对此十分满意，而且发现系统速度变得更快，更容易加入新 classes 或使用其他 classes。

项目经理并不高兴。进度排得很紧，许多工作要做。系统必须在几个月之后发布，许多功能还等着加进去，这些程序员却白白耗费两天时间，什么活儿都没干。原先的代码运行起来还算正常，他们的新设计显然有点过于「理论」且过于「无瑕」。项目要出货给客户的，是可以有效运行的代码，不是用以取悦学究们的完美东西。顾问接下来又建议应该在系统的其他核心部分进行这样的整理工作，这会使整个项目停顿一至二个星期。所有这些工作只是为了让代码看起来更漂亮，并不能给系统

添加任何新功能。

你对这个故事有什么看法？你认为这个顾问的建议（更进一步整理程序）是对的吗？你会因循那句古老的工程谚语吗：「如果它还可以运行，就不要动它」。

我必须承认我自己有某些偏见，因为我就是那个顾问。六个月之后这个项目宣告失败，很大的原因是代码太复杂，无法除错，也无法获得可被接受的性能。

后来，项目重新启动，几乎从头开始编写整个系统，Kent Beck 被请去做了顾问。他做了几件迥异以往的事，其中最重要的一件就是坚持以持续不断的重构行为来整理代码。这个项目的成功，以及重构（refactoring）在这个成功项目中扮演的角色，促成了我写这本书的动机，如此一来我就能够把 Kent 和其他一些人已经学会的「以重构方式改进软件质量」的知识，传播给所有读者。

什么是重构（Refactoring）？

所谓重构是这样一个过程：「在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构」。重构是一种有纪律的、经过训练的、有条不紊的程序整理方法，可以将整理过程中不小心引入错误的机率降到最低。本质上说，重构就是「在代码写好之后改进它的设计」。

「在代码写好之后改进它的设计」？这种说法有点奇怪。按照目前对软件开发的理解，我们相信应该先设计而后编码（coding）。首先得有一个良好的设计，然后才能开始编码。但是，随着时间流逝，人们不断修改代码，于是根据原先设计所得的系统，整体结构逐渐衰弱。代码质量慢慢沉沦，编码工作从严谨的工程堕落为胡砍乱劈的随性行为。

「重构」正好与此相反。哪怕你手上有一个糟糕的设计，甚至是一堆混乱的代码，你也可以借由重构将它加工成设计良好的代码。重构的每个步骤都很简单，甚至简单过了头，你只需要把某个值域（field）从一个 class 移到另一个 class，把某些代码从一个函数（method）拉出来构成另一个函数，或是在 class hierarchy 中把某些代码推上推下就行了。但是，聚沙成塔，这些小小的修改累积起来就可以根本改善设计质量。这和一般常见的「软件会慢慢腐烂」的观点恰恰相反。

通过重构（refactoring），你可以找出改变的平衡点。你会发现所谓设计不再是一切动作的前提，而是在整个开发过程中逐渐浮现出来。在系统构筑过程中，你可以学习如何强化设计；其间带来的互动可以让一个程序在开发过程中持续保有良好的设计。

本书有些什么？

本书是一本重构指南（guide to refactoring），为专业程序员而写。我的目的是告诉你如何以一种可控制且高效率的方式进行重构。你将学会这样的重构方式：不引入「臭虫」（错误），并且有条不紊地改进程序结构。

按照传统，书籍应该以一个简介开头。尽管我也同意这个原则，但是我发现以概括性的讨论或定义来介绍重构，实在不是件容易的事。所以我决定拿一个实例做为开路先锋。第 1 章展示一个小程序，其中有些常见的设计缺陷，我把它重构为更合格的面向对象程序。其间我们可以看到重构的过程，以及数个很有用的重构准则。如果你想知道重构到底是怎么回事，这一章不可不读。

第 2 章涵盖重构的一般性原则、定义，以及进行原因，我也大致介绍了重构所存在的一些问题。第 3 章由 Kent Beck 介绍如何嗅出代码中的「坏味道」，以及如何运用重构清除这些坏味道。「测试」在重构中扮演非常重要的角色，第 4 章介绍如何运用一个简单的（源码开放的）Java 测试框架，在代码中构筑测试环境。

本书的核心部分，重构名录（catalog of refactorings），从第 5 章延伸至第 12 章。这不是一份全面性的名录，只是一个起步，其中包括迄今为止我在工作中整理下来的所有重构准则。每当我想做点什么 — 例如 *Replace Conditional with Polymorphism* — 的时候，这份名录就会提醒我如何一步一步安全前进。我希望这是值得你日后一再回顾的部分。

本书介绍了其他人的许多研究成果，最后数章就是由他们之中的几位所客串写就。Bill Opdyke 在第 13 章记述他将重构技术应用于商业开发过程中遇到的一些问题。Don Roberts 和 John Brant 在第 14 章展望重构技术的未来 — 自动化工具。我把最后一章（第 15 章）留给重构技术的顶尖大师，Kent Beck。

在 Java 中运用重构

本书全部以 Java 撰写实例。重构当然也可以在其他语言中实现，而且我也希望这本书能够给其他语言使用者带来帮助。但我觉得我最好在本书中只使用 Java，因为那是我最熟悉的语言。我会不时写下一些提示，告诉读者如何在其他语言中进行重构，不过我真心希望看到其他人在本书基础上针对其他语言写出更多重构方面的书籍。

为了最大程度地帮助读者理解我的想法，我不想使用 Java 语言中特别复杂的部分。所以我避免使用 inner class（内嵌类）、reflection（反射机制）、thread（线程）以及很多强大的 Java 特性。这是因为我希望尽可能清楚展现重构的核心。

我应该提醒你，这些重构准则并不针对并发（concurrent）或分布式（distributed）编程。那些主题会引出更多重要的事，超越了本书的关心范围。

谁该阅读本书？

本书瞄准专业程序员，也就是那些以编写软件为生的人。书中的示例和讨论，涉及大量需要详细阅读和理解的代码。这些例子都以 Java 完成。之所以选择 Java，因为它是一种应用范围愈来愈广的语言，而且任何具备 C 语言背景的人都可以轻易理解它。Java 是一种面向对象语言，而面向对象机制对于重构有很大帮助。

尽管关注对象是代码，重构（refactoring）对于系统设计也有巨大影响。资深设计师（senior designers）和架构规划师（architects）也很有必要了解重构原理，并在自己的项目中运用重构技术。最好是由老资格、经验丰富的开发人员来引入重构技术，因为这样的人最能够良好理解重构背后的原理，并加以调整，使之适用于特定工作领域。如果你使用 Java 以外的语言，这一点尤其必要，因为你必须把我给出的范例以其他语言改写。

下面我要告诉你：如何能够在不遍读全书的情况下得到最多知识。

- 如果你想知道重构是什么，请阅读第 1 章，其中示例会让你清楚重构过程。
- 如果你想知道为什么应该重构，请阅读前两章。它们告诉你「重构是什么」以及「为什么应该重构」。