PEARSON

Refactoring Improving the Design of Existing Code

重构 改善既有代码的设计

[美] Martin Fowler 著
熊节 译

- 软件开发的不朽经典
- 生动阐述重构原理和具体做法
- 普通程序员进阶到编程高手必须修炼的秘笈

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

PEARSON

Refactoring Improving the Design of Existing Code

重构 改善既有代码的设计

[美] Martin Fowler 著
熊节 译

人民邮电出版社
北京

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 ；熊节译. -- 2版. -- 北京 ： 人民邮电出版社, 2015.8
书名原文: Refactoring: Improving the Design of Existing Code
ISBN 978-7-115-36909-3

Ⅰ. ①重… Ⅱ. ①福… ②熊… Ⅲ. ①机器代码程序—程序设计 Ⅳ. ①TP311.11

中国版本图书馆CIP数据核字(2015)第137778号

内 容 提 要

本书清晰揭示了重构的过程，解释了重构的原理和最佳实践方式，并给出了何时以及何地应该开始挖掘代码以求改善。书中给出了 70 多个可行的重构，每个重构都介绍了一种经过验证的代码变换手法的动机和技术。本书提出的重构准则将帮助你一次一小步地修改你的代码，从而减少了开发过程中的风险。

本书适合软件开发人员、项目管理人员等阅读，也可作为高等院校计算机及相关专业师生的参考读物。

◆ 著　　[美] Martin Fowler
译　　熊　节
责任编辑　杨海玲
责任印制　张佳莹　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
三河市中晟雅豪印务有限公司印刷

◆ 开本：800×1000　1/16
印张：28.25
字数：490 千字　　2015 年 8 月第 2 版
印数：1-6 000 册　　2015 年 8 月河北第 1 次印刷

著作权合同登记号　图字：01-2009-5707 号

定价：69.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

重 构 列 表

版 权 声 明

重构的重新认识
（再版序）

光阴荏苒，从当年译完这本《重构》，到如今重新整理译稿，不知不觉已经过去6年了。6年来，在各种大型系统中进行重构和指导别人重构，一直是我的一项工作。对于这本早已烂熟于心的书，也有了一些新的认识。

不得不遗憾地说，尽管“重构”已经成了常用词汇，但重构技术并没有像我当初乐观认为的那样“变得像空气与水一样普通”。一方面，一种甚嚣尘上的观点认为只要掌握重构的思想就足够了，没必要记住那些详细琐碎的重构手法；另一方面，倒是有很多人高擎“重构”大旗，刀劈斧砍进行着令人触目惊心的大胆修改——有些干脆就是在重做整个系统。

这些人常常忘了一个最基本的定义：重构是**在不改变软件可观察行为的前提下改善其内部结构**。当你面对一个最需要重构的遗留系统时，其规模之大、历史之久、代码质量之差，常会使得添加单元测试或者理解其逻辑都成为不可能的任务。此时你唯一能依靠的就是那些已经被证明是行为保持的重构手法：用绝对安全的手法从“焦油坑”中整理出可测试的接口，给它添加测试，以此作为继续重构的立足点。

六年来，在各种语言、各种行业、各种软件形态，包括规模达到上百万行代码的项目中进行重构的经验让我明白，“不改变软件行为”只是重构的最基本要求。要想真正让重构技术发挥威力，就必须做到“不需了解软件行为”——听起来很荒谬，但事实如此。如果一段代码能让你容易了解其行为，说明它还不是那么迫切需要被重构。那些最需要重构的代码，你只能看到其中的“坏味道”，接着选择对应的重构手法来消除这些“坏味道”，然后才有可能理解它的行为。而这整个过程之所以可行，全赖你在脑子里记录着一份“坏味道”与重构手法的对应表。

而且，尽管Java和.NET的自动化重构工具已经相当成熟，但另一些重要的面向对象语言（C++、Ruby、Python……）还远未享受到这样的便利。在重构这些语言编写的程序时，我们仍然必须遵循这些看似琐碎的做法指导（加上语言特有的细节调整），按部就班地进行——如果你还想以安全的方式重构的话。

所以，仅仅掌握思想是没用的。如果把重构比作一门功夫的话，它的威力全都来自日积月累的勤学苦练。记住所有的“坏味道”，记住它们对应的重构手法，记住常见的重构步骤，然后你才可能有信心面对各种复杂情况——学会所有的招式，才可能“无招胜有招”。我知道这听起来很难，但我也知道这并不像你想象的那么难。你所需要的只是耐心、毅力和不断重读这本书。

熊　节

2009年10月21日

重构的生活方式
（译　序）

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，以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，不仅仅是一些具体问题的解决方案，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，Kerievsky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网络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，就再也没有读过它。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，以至于无需再到书中寻找任何“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，我要给你两个建议：首先，把你的敬畏扔到太平洋里去，对于即将变得像空气与水一样普通的技术，你无需对它敬畏；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《重构》这本书的想法。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日于杭州

序

“重构”这个概念来自Smalltalk圈子，没多久就进入了其他语言阵营之中。由于重构是框架开发中不可缺少的一部分，所以当框架开发人员讨论自己的工作时，这个术语就诞生了。当他们精炼自己的类继承体系时，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。框架设计者知道，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——对框架而言如此，对一般软件也如此。

好极了，还有什么问题吗？问题很显然：重构具有风险。它必须修改运作中的程序，这可能引入一些不易察觉的错误。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。挖得越深，找到的重构机会就越多，于是你的修改也越多……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《设计模式》书中和另外三位作者曾经提过：设计模式为重构提供了目标。然而“确定目标”只是问题的一部分而已，改造程序以达到目标，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。本书解释了重构的原理和最佳实践，并指出何时何地你应该开始挖掘你的代码以求改善。本书的核心是一系列完整的重构方法，其中每一项都介绍一种经过实践检验的代码变换手法的动机和技术。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手法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手法和其名称加入自己的开发词典中，并且朗朗上口。

我第一次体验有讲究的、一次一小步的重构，是某次与Kent Beck在30 000英尺高空的飞行旅途中结对编程。我们运用本书收录的重构手法，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，我极力推荐你试试这些重构手法，你和你的程序都将因此更美好。

Erich Gamma

《设计模式》第一作者，Eclipse平台主架构师

前　言

从前，有位咨询顾问造访客户调研其开发项目。系统核心是个类继承体系，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，上层超类对于系统的运作做了一些假设，下层子类实现这些假设。但是这些假设并不适合所有子类，导致覆写（override）工作非常繁重。只要在超类做点修改，就可以减少许多覆写工作。在另一些地方，超类的某些意图并未被良好理解，因此其中某些行为在子类内重复出现。还有一些地方，好几个子类做相同的事情，其实可以把它们搬到继承体系的上层去做。

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继承体系上工作的程序员，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继承体系，并删掉了其中一半代码，功能毫发无损。他们对此十分满意，而且发现在继承体系中加入新的类或使用系统中的其他类都更快、更容易了。

项目经理并不高兴。进度排得很紧，有许多工作要做。系统必须在几个月之后发布，而这些程序员却白白耗费了两天时间，干的工作与要交付的多数功能毫无关系。原先的代码运行起来还算正常，他们的新设计看来有点过于追求完美。项目要交付给客户的，是可以有效运行的代码，不是用以取悦学究的完美东西。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感想？你认为这个顾问的建议（更进一步整理程序）是对的吗？你会遵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，无法调试，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受邀做了顾问。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，以及重构在这个成功项目中扮演的角色，启发了我写这本书，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

所谓重构（refactoring）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。重构是一种经千锤百炼形成的有条不紊的程序整理方法，可以最大限度地减少整理过程中引入错误的几率。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：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，甚至显得有些过于简单：你只需要把某个字段从一个类移到另一个类，把某些代码从一个函数拉出来构成另一个函数，或是在继承体系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，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。

本书有什么

本书是一本为专业程序员而写的重构指南。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。你将学会如何有条不紊地改进程序结构，而且不会引入错误，这就是正确的重构方式。

按照传统，图书应该以引言开头。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。所以我决定用一个实例做为开路先锋。第1章展示了一个小程序，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，以及几个很有用的重构手法。如果你想知道重构到底是怎么回事，这一章不可不读。

第2章讨论重构的一般性原则、定义，以及进行重构的原因，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。测试在重构中扮演着非常重要的角色，第4章介绍如何运用一个简单而且开源的Java测试框架，在代码中构筑测试环境。

本书的核心部分——重构列表——从第5章延伸至第12章。它不能说是一份全面的列表，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手法。每当我想做点什么（例如*Replace Conditional with Polymorphism* (255) ）的时候，这份列表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，最后几章就是由他们之中的几位所客串写就的。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来压轴。

在 Java 中运用重构

本书范例全部使用Java撰写。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。

为了很好地与读者交流我的想法，我没有使用Java语言中特别复杂的部分。所以我避免使用内嵌类、反射机制、线程以及很多强大的Java特性。这是因为我希望尽可能清楚地展现重构的核心。

我应该提醒你，这些重构手法并不针对并发或分布式编程。那些主题会引出更多的考虑，本书并未涉及。

谁该阅读本书

本书的目标读者是专业程序员，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写成。之所以选择Java，因为它是一种应用范围越来越广的语言，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，但重构对于系统设计也有巨大影响。资深设计师和架构师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透彻理解重构背后的原理，并根据情况加以调整，使之适用于特定工作领域。如果你使用的不是Java，这一点尤其重要，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，如何能够在不通读全书的情况下充分用好它。

- 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的过程。
- 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。

- 如果你想知道该在什么地方重构，请阅读第3章。它会告诉你一些代码特征，这些特征指出“这里需要重构”。
- 如果你想着手进行重构，请完整阅读前四章，然后选择性地阅读重构列表。一开始只需概略浏览列表，看看其中有些什么，不必理解所有细节。一旦真正需要实施某个准则，再详细阅读它，从中获取帮助。列表部分是供查阅的参考性内容，你不必一次就把它全部读完。此外你还应该读一读列表之后其他作者的“客串章节”，特别是第15章。

站在前人的肩膀上

就在本书一开始的此时此刻，我必须说：这本书让我欠了一大笔人情债，欠那些在过去十年中做了大量研究工作并开创重构领域的人一大笔债。这本书原本应该由他们之中的某个人来写，但最后却是由我这个有时间有精力的人捡了便宜。

重构技术的两位最早倡导者是Ward Cunningham和Kent Beck。他们很早就把重构作为开发过程的一个核心成分，并且在自己的开发过程中运用它。尤其需要说明的是，正因为和Kent的合作，才让我真正看到了重构的重要性，并直接激励了我写出这本书。

Ralph Johnson在UIUC（伊利诺伊大学厄巴纳-尚佩恩分校）领导了一个小组，这个小组因其在对象技术方面的实际贡献而声名远扬。Ralph很早就是重构技术的拥护者，他的一些学生也一直在研究这个课题。Bill Opdyke的博士论文是重构研究的第一份详细的书面成果。John Brant和Don Roberts则早已不满足于写文章了，他们写了一个工具叫Refactoring Browser（重构浏览器），对Smalltalk程序实施重构工程。

致谢

尽管有这些研究成果可以借鉴，我还是需要很多协助才能写出这本书。首先，并且也是最重要的，Kent Beck给了我巨大的帮助。Kent在底特律的某个酒吧和我谈起他正在为*Smalltalk Report*撰写一篇论文[Beck，hanoi]，从此播下本书的第一颗种子。那次谈话不但让我开始注意到重构技术，而且我还从中“偷”了许多想法放到