TURING 图灵程序设计丛书 程序员修炼系列

PRENTICE HALL PTR

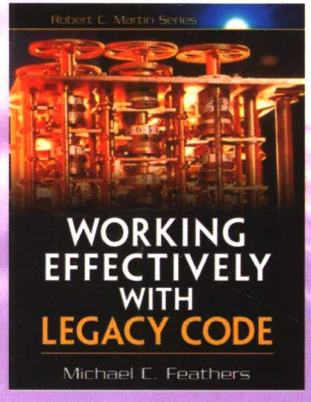
Working Effectively with Legacy Code

修改代码的艺术

《重构》之后又一里程碑

[美] Michael C. Feathers 著
Robert Martin 序
刘未鹏 译

- 修改代码的集大成之作
- Amazon 全五星图书
- 适用于各种语言或平台



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书 程序员修炼系列

修改代码的艺术

Working Effectively with Legacy Code

[美] Michael C. Feathers 著
Robert Martin 序
刘未鹏 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

修改代码的艺术 /（美）费瑟（Feathers, M.C.）著；刘未鹏译. —北京：人民邮电出版社，2007.11
（图灵程序设计丛书）
ISBN 978-7-115-16362-2

Ⅰ. 修… Ⅱ. ①费…②刘… Ⅲ. 软件开发 Ⅳ. TP311.52

中国版本图书馆 CIP 数据核字（2007）第 084231 号

内 容 提 要

修改代码是每一位软件开发人员的日常工作。开发人员常常面对的现实是，即便是最训练有素的开发团队也会写出混乱的代码，而且系统的腐化程度也会日积月累。本书是一部里程碑式的著作，针对大型的、无测试的遗留代码基，提供了从头到尾的方案，让你能够更有效地应付它们，将你的遗留代码基改善得具有更高性能、更多功能、更好的可靠性和可控性。本书还包括了一组共 24 项解依赖技术，它们能帮助你单独对付代码中的问题片段，并实现更安全的修改。

本书适合各层次软件开发人员、管理人员和测试人员阅读。

图灵程序设计丛书

修改代码的艺术

◆ 著 [美] Michael C. Feathers
序 Robert Martin
译 刘未鹏
责任编辑 陈兴璐

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷
新华书店总店北京发行所经销

◆ 开本：800×1000 1/16
印张：22.5
字数：538 千字 2007 年 11 月第 1 版
印数：1 – 5 000 册 2007 年 11 月河北第 1 次印刷

著作权合同登记号 图字：01-2006-3687 号

ISBN 978-7-115-16362-2/TP

定价：59.00 元

读者服务热线：(010)88593802 印装质量热线：(010)67129223

版 权 声 明

修改代码之三十六计

（译者序）

六六三十六，数中有术，术中有数。阴阳燮理，机在其中。机不可设，设则不中。

——《三十六计》

好的技术书籍一般有两种情况，一种是介绍一些新奇而有趣的技术，另一种是能将现有的技术阐述或概括得通透淋漓。然而，实际上还有第三种——既非介绍新奇的技术，也非阐述既有技术，而是将被长期实践所证明了的大量技术手法囊括至一起。看起来琳琅满目五花八门，但又各有各的用武之地。这样的书一般较少见，因为需要长期的积累和时间的洗礼。

本书正是这样一本书。

说实话，对于这样一本由“鲍勃大叔”亲自作序，且Amazon上篇篇书评都是五星加夸赞的书，我这个译者反倒有点遑于置评了。要想知道本书为什么填补了一项重要的空白（在Kent Beck的《测试驱动开发》、Martin Fowler的《重构：改善既有代码的设计》、Robert C. Martin的《敏捷软件开发：原则、模式与实践》等重磅炸弹投下之后），可以看Michael Feathers的前言。要想知道这本书为什么值得你放在书架上，可以看鲍勃大叔的序。要想知道读者怎么认为，可以看Amazon上的书评。

所以，与其画蛇添足，不如随手摘来Amazon上的一些书评片段：

“大多数软件开发的图书都是关于原生开发的：教你如何从无到有地创建出一个新的应用来。然而实际情况是，真正身处业界但往往大部分时候面对的却是既有代码：添加特性、寻找bug以及重构别人写的代码。因此，图书与实践这两个世界就产生了不平衡，而本书正是在平衡这两个世界上迈出了漂亮的一步。”

“Feathers用简洁清晰的代码示例漂亮地阐述了我们面对的各种问题场景……书中的代码示例跟我在实际工作中常常遇到的那些问题代码非常相近……”

“总的来说，这本书写得非常漂亮，将一个以前很少被涉及但很重要的主题作了极好的阐述。”

“我想在接下来的几年中我都会时常把这本书从书架上拿下来翻阅。”

那么，请带上这只妙计锦囊吧，enjoy！

最后，感谢刘江先生容忍我一而再的拖稿，让我得以在繁忙的一年仍能够认真译完这本好书。感谢父母一直以来的支持和鼓励。

刘未鹏

2007年2月

于南京

序

“……所有的一切就从那一刻开始……”

Michael Feathers在对本书的介绍中用这句话来描述他当初是怎样迷上软件开发的。

“……所有的一切就从那一刻开始……”

你能够体会那种感觉吗？你是否能够回忆起你生命中的某个时刻，说“……所有的一切就从那一刻开始……”？有没有某一刻某件事改变了你生命的进程，最终，使你拿起了这本书读到了这篇序言？

对我来说，所有的一切是从六年级的时候开始的。当时我对科学、太空以及一切与技术相关的东西都感兴趣。母亲在店里发现了一台塑料电脑玩具并买下来送给了我，我还记得它的名字叫“Digi-Comp I”。40年过去了，那台小小的塑料电脑玩具在我的书架上仍光荣地占有一席之地。它点燃并催化了我对软件开发的持续热情，它让我第一次隐约感受到了编写程序来解决人们的问题是多么有意思的一件事情。那只不过是一台由3个塑料的S-R触发器和6个与门组成的简单机器，但这已经足够了。于是……对我来说……一切就从那一刻开始……

后来我的热情逐渐冷却下来，因为我发现现实中的软件系统几乎总是会慢慢变为一个烂摊子。程序员脑子里原先那些漂亮的设计随着时间的推移会慢慢“发出腐化的臭味”。我们往往会发现，去年才构建的漂亮小巧的系统，到了今年却变成了由一堆纠缠不清的函数和变量搅和在一起的“代码浆糊”。

为什么会这样？为什么一个原先好好的系统会逐渐“发出腐化的臭味”？为什么它们不能保持原先那样的清晰简洁呢？

有时候，我们会把原因归咎于客户，责怪他们总是改变需求。我们自我安慰地认为，只要客户的需求仅限于他们最初所声明的，那么我们的设计就是没问题的，所以错就错在客户改变了他们的需求。

呃……然而问题在于：需求总是在改变。那些不能适应未来需求变更的设计是糟糕的设计。能够适应未来需求变更的设计是每一位合格的软件开发者的目标。

这听起来似乎是个极难解决的问题。难到什么程度呢？实际上，迄今为止人们构建出的几乎所有软件系统都遭遇了缓慢的、不可抗拒的腐化。这种现象是如此的普遍，以至于我们给那些腐化得散发着臭味的程序起了一个别致的名字：**遗留代码**。

遗留代码，一个令程序员感到头大的词。它往往令人联想到“在黑暗的、乱糟糟的灌木丛中艰难地跋涉，脚下有吸血的蚂蟥，旁边还有蜇人的昆虫飞来飞去，它散发着某种黑暗的、粘乎乎的、钝重的、腐烂的垃圾般的气味”。尽管我们初尝编程的滋味可能会是很美妙的，但在处理遗留代码时的痛苦往往会无情地将你的热情之火浇灭。

我们中的许多人都曾尝试过以某种方式避免让代码沦为遗留代码。我们已经编写了关于原则、模式和实践的书[1]，以帮助程序员保持其软件系统的清晰简洁。然而Michael Feathers具有我们许多人都没有的洞察力。他指出，光采用预防措施是不够的。即便是最训练有素的开发团队（通晓最佳原则，使用最佳模式，遵循最佳实践方式）也常常会写出混乱的代码，而且系统的腐化程度也会日积月累。所以，仅是努力防止腐化是不够的，你必须设法扭转它。

这便是本书所要讲述的内容。简言之，本书教你如何扭转腐化，教你在面对一个错综复杂的、不透明的、令人费解的系统时如何慢慢地、逐步地将其变成一个简单的、有良好组织和设计的系统。打个比方，就好比扭转一个热力学系统在自发状态下熵增的趋势。

在你摩拳擦掌、跃跃欲试之前，我得先警告你：扭转腐化趋势的过程并不轻松，而且也算不上迅速。Michael在本书中给出的技术、模式及工具是非常有效的，但仍需要你花费精力、时间、耐心以及细致。本书并不是什么神丹妙药。它不会告诉你如何在一夜之间就把系统中积累的腐化成分统统去除，而是告诉你一些在今后的开发中应当谨记的原则、概念和态度，这样才能帮助你将逐渐退化的系统转变为渐趋完善。

Robert C. Martin
面向对象技术大师
Object Mentor公司总裁
2004年6月29日

1. 指Robert Martin的《敏捷软件开发：原则、模式与实践》。人民邮电出版社即将推出此书Java版的英文注释版和C#版的英文注释版和中文版。——编者注

前 言

还记得你自己编写的第一个程序吗？我可记得。当时我编写的是早期PC上的一个小小的图形程序。虽然在孩提时代我就已经见过计算机了，但我开始编程的年龄比我的大部分朋友都要大。我记得很清楚，有一次我在一间办公室里见到了微机，当时留下了深刻印象。在后来好几年间，我都没有任何机会接触计算机。直到我十来岁的时候，我的几个朋友买了几台第一代的TRS-80型计算机。我跃跃欲试，但又有点儿担心，因为我知道一旦我开始接触了计算机，就会深陷其中不能自拔。我不知道当时为什么会有这种担心，但我的确退缩了。后来，我进了大学，一位室友买了台电脑，而我买了一套C编译器，这样就能够自学编程了。于是，一切就从那一刻开始了。我在不断地尝试，在尝试中度过了一个又一个不眠之夜，我一遍一遍地啃编译器附带的emacs编辑器的源代码。我上瘾了，这项工作充满了挑战性，我喜爱它。

我希望你也有过类似的体验——那种因程序终于在计算机上运行起来而产生的巨大成功带来的喜悦。几乎所有我问过的程序员都说曾有过类似的感觉。这种感觉正是让我们喜爱这个行业的原因之一。然而，在日复一日的编程中，这种感觉为何消失得无影无踪了呢？

几年前的某个晚上，我结束了手头的工作，给我的朋友Erik Meade打了一个电话。当时我知道他正在给一个新的开发团队做咨询，所以我就问他："他们做得怎么样？"Erik回答道："唉，他们在编写遗留代码。"他的话令我心头一震，但我打心底里觉得他说的是对的。Erik精确地描述出了我第一次接触开发团队时的感觉：他们工作非常努力，然而一天下来，由于进度压力、"历史"包袱，或者由于没有任何更好的代码能够与他们的成果相比之类的种种原因，结果许多人编写的代码都成了"遗留代码"。

什么是遗留代码？我未加定义就直接使用了这一术语。现在来看一看它的严格定义：遗留代码就是指从其他人那儿得来的代码。导致这一点的原因很多，例如，可能是我们的公司从其他公司那儿获取了代码；可能是原来的团队转而去做另一个项目了（从而遗留下了一堆代码）。总而言之，遗留代码就是指其他人编写的代码。不过，在程序员的口中，该术语所蕴涵的意义却远远不只这些。遗留代码这一说法随着时间的推移已经拥有了某些独特的含义。

那么，当你初次听到"遗留代码"这一名词的时候，心里是怎么想的呢？如果你也和我一样，那么大抵会联想到错综复杂的、难以理清的结构，需要改变然而实际上又根本不能理解的代码；你会联想到那些不眠之夜，试图添加一个本该很容易就添加上去的特性；你会联想到自己是如何的垂头丧气，以及你的团队中的每个人对一个似乎没人管的代码基是如何打心底里感到厌烦的，这种代码正是你希望彻底扔进垃圾堆的那种。你内心深处甚至对于想一想怎样才能改善这种代码都感到痛苦。这种事情似乎太不值得我们付出努力了。此外，遗留代码的定义中没有任何地方提到代码编写者。实际上，代码退化的方式是多种多样的，其中许多与是否来自另一个开发团队根

本没有任何关系。

在业内人士的口中，“遗留代码”一词常常是“无法理解的、难以修改的代码”的代名词。然而，在多年来与形形色色的开发团队共事，并帮助他们解决重大的编码问题的过程中，我总结出了一个不同的定义。

对我来说，遗留代码就是那些没有编写相应测试的代码。明白这一点是很痛苦的。人们会问，代码的好坏与是否编写了测试有什么关系呢？答案很明显，而这也正是我将要在本书中阐述的：

没有编写测试的代码是糟糕的代码。不管我们有多细心地去编写它们，不管它们有多漂亮、面向对象或封装良好，只要没有编写测试，我们实际上就不知道修改后的代码是变得更好了还是更糟了。反之，有了测试，我们就能够迅速、可验证地修改代码的行为。

你可能会觉得这有点危言耸听了。难道那些干净的代码也需要这样吗？只要一个代码基是非常干净且结构良好的不就得了？呃……别误会，我当然喜欢干净的代码，然而只是干净还不够。在没有相应的测试的情况下就进行大规模的修改是要冒很大风险的。这就好像在没有防护网的情况下进行高空体操表演。总之，需要极高的技巧，并要对每一步会发生什么有着清晰的认识。而在软件开发中，精确地预知在改变了几个变量后会发生什么，通常无异于在高空体操中预知另一位体操运动员是否会准确地在你翻完一个筋斗之后抓住你的胳膊。如果你所在团队的代码有那么清晰，那么你比大多数程序员都要幸运。根据我个人的工作经验，我发现团队拥有的代码极少是处处都那么清晰的，其概率微乎其微。而且，即便你很幸运，只要你们的代码没有编写相应的测试，其进行修改时的速度仍然比不上那些有测试的团队。

开发团队的水平在不断提高，他们编写的代码也变得越来越清晰，然而旧代码要想变得更清晰就要花更长的时间。许多情况下旧代码甚至永远都不可能变得完全清晰。正因如此，我将“遗留代码”定义为“没有编写测试的代码”，而且它本身就提出了问题的一条解决方案。

到目前为止，我已经就测试说了很多，然而本书并不是关于测试的，而是关于如何才能够放心地对任何代码基进行修改的。在后面的章节中，我描述了许多技术，有些是关于理解代码的，有些是用于将代码放入测试之下的，有些是关于重构代码的，还有些是关于添加特性的。

通过阅读本书，你将会注意到一点：这并非是一本关于漂亮代码的书。我在书中使用的例子都是虚构的，这是因为我与客户之间有保密协议，不能泄漏他们的源代码。不过，在许多例子中，我都尽量保留了在业界见到的实际代码的精神。我不敢说这些代码全都具有代表性。在实际中当然存在着大量不错的代码，不过坦白地说，我也遇到过一些远远不够资格用作本书例子的代码。对于这些代码，即使没有客户保密协议的约束，我也不会将它们用作书中的例子，我可不想把读者弄得一头雾水，更不想把重点埋进细节的沼泽中。因此，你会看到，书中的许多例子相对来说都是比较简洁的。如果你会有异议，“不，他没弄明白——我的函数可要比这大得多、糟糕得多”，那么，我建议你逐字逐句地读一读我给出的相应的建议，看看它是否适用（即使书中的例子看似更简单）。

书中的技术已经在充分大的代码段上得到了验证。只不过由于篇幅限制，例子的长度缩减了。特别地，当你在一个代码片段中看到省略号（……）时，可以将它想象成“在这里插入500行丑

陋的代码”：

```
m_pDispatcher->register(listener);
……   // 想象成“在这里插入500行丑陋的代码”
m_nMargins++;
```

本书不仅不是关于漂亮代码的，它甚至也不能算是关于漂亮设计的。良好的设计应当是所有开发者的追求，然而对于遗留代码来说，良好的设计只是我们不断逼近的目标。在某些章节中，我描述了用于向既有代码基中添加新代码的方法，并指出了如何在头脑中保持良好设计原则的前提下做这件事情。你可以在遗留代码基上“培养”出高质量的代码，不过倘若你在修改的某些步骤中发现某些代码变得比原来更丑陋了，千万别感到惊讶。因为这就像动手术一样，先开一个切口，进而在五脏六腑中动手术，先别管是否美观。这个病人的病可以医治好吗？是的。那么我们是否应把他的迫在眉睫的问题放在一旁，缝合伤口，然后告诉他注意饮食并立刻进行马拉松锻炼？我们当然可以这么做，但我们真正需要做的是医好他的病，让他更健康。他可能永远也不会成为一位奥运会运动员，但我们不能让追寻“最好”之心妨碍了我们去实现“更好”。代码基可以变得更好，更有利于我们在其上进行工作。同样地，一位病人身体恢复一点的时候常常就是你可以帮他实现更健康的生活方式的时候。这也正是我们对于遗留代码所要做的。我们设法到达一种时常感到轻松的状态，并积极设法让代码修改工作变得更轻松。当我们能够在一个团队中保持这种感觉时，就意味着设计变得更好了。

书中描述的技术是我与同事和客户在多年的工作（设法建立起对难以驾驭的代码基的控制）中发现并总结出来的。我的工作重点转向遗留代码完全出于偶然。当时我刚开始在Object Mentor工作，大部分工作是帮助有严重问题的团队提高他们的技术水平以及增进他们之间的交流，直到他们能够定期交付高质量的代码。我们常常使用极限编程实践来帮助团队控制他们的工作、实现通力合作以及代码的交付。我常常觉得极限编程（XP）与其说是一种软件开发的方式，倒不如说是一种有助于组建起一支良好合作的工作团队的思想理念，而这个团队能够每两星期交付漂亮的软件则只不过碰巧是这一理念的副产品之一而已。

话虽如此，在一开始的时候还是有点问题。最初的许多XP项目都是“新开”的项目。我的客户都是那些拥有相当庞大的代码基且遇到麻烦的客户。他们需要某种办法来控制其工作，并开始交付。随着时间的推移，我发现自己重复地做着同样的工作。这种感觉在一次与一个金融业的团队一起工作的时候强烈到了顶点。当时的情况是：在我加入他们之前，他们已经意识到单元测试非常有用，然而实际上进行的却是全景式的测试，他们写的测试很繁琐，需要多次调用数据库并执行大量的代码。这种测试难于编写，而且也并不常用，因为运行耗费的时间实在是太长了。后来，我帮助他们解开代码间的依赖。在将较小块的代码纳入到测试当中的时候，我有一种强烈的似曾相识的感觉：我在每个团队中做的都是同样的工作，一种没有人真正想去深入思考的工作。然而，当任何人想要控制并处理他们的代码时（如果他们知道怎么做的话），这一工作恰恰又是必不可少的。于是，当时我决定了一件事，即思考我们该如何处理这类问题，并将它们记下来，这样就能够帮助开发团队将他们的代码基变得更易“相处”。

另外，关于书中的例子还有一点要注意的，它们并非使用同一种语言编写，其中多数是Java、

C++和C代码。我选择了Java，因为它是一门非常常用的语言；我也选择了C++因为在处理C++遗留代码时有一些特有的挑战；我还选择了C，因为C遗留代码突出了在处理过程式遗留代码时会出现的许多问题。这些语言的代码覆盖了在处理遗留代码时需要考虑的大多数因素。然而，即便例子中没有使用你所使用的语言，通过这些例子你照样可以学到东西。书中描述的许多技术都可以在其他语言环境下使用，例如Delphi、Visual Basic、COBOL以及FORTRAN。

我希望你能认为本书中的技术对你有所帮助，并助你重拾编程的乐趣。编程可以是一项回报丰厚并让人感觉是一种享受的工作。如果你在日复一日的编程生涯中并没有感受到这一点，希望书中提供的技术能够帮你找到这种感觉，并把它带给你的整个团队。

致谢

首先我想真诚地感谢我的妻子Ann，还有我的两个孩子，Deborah和Ryan。没有他们的爱和支持我绝对无法完成本书以及之前的种种准备工作。同样也要感谢Object Mentor的总裁和创建者，人称“Bob大叔”的Martin；他在开发和设计中的严谨务实的态度使当年（大约十年前吧）被一大堆不切实际的鼓吹弄得晕头转向的我从迷惘中走了出来。另外还要感谢Bob，他让我在过去的五年中有机会接触更多的代码与更多的人一起工作，这样的机会我以前是不敢想象的。

此外我还要感谢Kent Beck、Martin Fowler、Ron Jeffries和Ward Cunningham，他们不仅给了我许多宝贵的建议，还在团队工作、设计以及编程方面令我获益良多。尤其要感谢所有审稿人，其中正式的有：Sven Gorts、Robert C. Martin、Erik Meade、Bill Wake；非正式的有：Dr. Robert Koss、James Grenning、Lowell Lindstrom、Micah Martin、Russ Rufer，还有硅谷模式小组和James Newkirk。

此外同样也要感谢另一组审稿人：Darren Hobbs、Martin Lippert、Keith Nicholas、Phlip Plumlee、C. Keith Ray、Robert Blum、Bill Burris、William Caputo、Brian Marick、Steve Freeman、David Putman、Emily Bache、Dave Astels、Russel Hill、Christian Sepulveda、Brian Christopher Robinson等人。在创作的早期，我将草稿放在了网上，他们的反馈对书（在我重新组织内容之后）的导向起了很大的影响。

感谢Joshua Kerievsky对本书做了关键的早期审稿，感谢Jeff Langr给本书提的宝贵建议和审校意见。

在我完善草稿的时候，所有审阅本书的人都对我提供了莫大的帮助。但如果你发现本书还是有错误的话，那肯定是我个人造成的。

感谢Martin Fowler、Ralph Johnson、Bill Opdyke、Don Roberts、John Brant在重构领域的工作，给了我许多灵感。

还要特别感谢Jay Packlick、Jacques Morel、Sabre Holdings 的Kelly Mower、Workshare Technology的Graham Wright，他们的支持和意见给了我不少帮助。

特别感谢Prentice-Hall出版社的Paul Petralia、Michelle Vincenti、Lori Lyons、Krista Hansing以及团队中的其余所有成员。感谢Paul对一个写作新手的帮助和鼓励。

特别感谢Gary和Joan Feathers、April Roberts、Dr. Raimund Ege、David Lopez de Quintana、

Carlos Perez、Carlos M. Rodriguez，还有Dr. John C. Comfort，在过去的几年中他们一直帮助和鼓励我。还要感谢Brian Button为本书第21章提供的例子，那次我们在一起做一个重构课程的时候他只用了大约1个小时就写出了这个例子，它现在已经是我在教学中最喜欢用的一个例子了。

感谢Janik Top的歌“De Futura”，在我写作本书的最后几个星期一直陪伴我。

最后，感谢我过去几年工作中的所有同事，他们的意见和质疑令本书的内容更经得起考验。

Michael Feathers
mfeathers@objectmentor.com
www.objectmentor.com
www.michaelfeathers.com

如何使用本书

本书的形式在最终确定之前曾几经易改。在修改遗留代码的过程中有许多不同的技术和实践如果独立开来是很难阐述好的。考虑到一旦人们能在代码中找到接缝（seam）、制造伪对象（fake object），并利用某些解依赖技术来解开代码中的依赖的话，简单的修改就会变得更容易。因此我想，要想让本书用起来更方便更顺手，最简单的办法莫过于将其主要内容（第二部分——修改代码的技术）以FAQ的形式来组织了。因为特定的技术往往要用到其他技术，所以FAQ章节之间经常有交叉链接。几乎在每章你都会发现一些对其他章节的引用及页码，后者描述了特定的解依赖或重构技术。如果这种组织形式使得你在寻找一个问题的解决方案的时候需要在书中翻来翻去的话，我感到很抱歉，但我仍然觉得你宁可这样也肯定不愿意去一页一页地读，并试图去理解那些技术都是怎样用的。

在修改软件的过程中我曾遇到过许多问题，我把其中比较常见的问题总结出来，本书每章都对应一个特定的问题。当然，这使得每章的标题比较长，但我觉得这样也好，你能够很快就找到对应你当前遇到的问题的章节。

在书的第二部分之前有一组介绍性章节（第一部分），之后则是一个重构技术的目录（第三部分），这些技术在修改遗留代码时是非常有用的。我建议你先阅读引入章节，尤其是第4章。这些章节中包含了后面要涉及的所有技术的上下文和术语。如果后面你还发现没有在上下文中涉及的术语，可以到术语表中去找。

解依赖技术中的重构工作是比较特殊的，因为它们本就应该是在没有测试的情况之下完成的，它们的作用就是给后面安放测试铺好道路。我建议你把所有的解依赖技术都浏览一下，这有助于你在修改代码的时候有更多的选择。

目　录

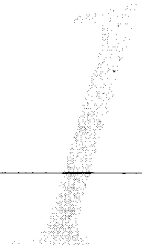
Part 1

第一部分

修 改 机 理

本部分内容

第1章 修改软件



修改（既有）代码本身并无什么问题，我们正是以此谋生的。然而，如果修改的方式不当则会招来麻烦，当然，只要方法正确，我们也可以令事情变得简单得多。在业界，对于修改代码的方法学讨论得不是很多，其中最接近的恐怕是重构方面的文献了。因此我觉得可以将讨论的范畴稍微扩大一点，即讨论如何在最为棘手的情况下处理代码。为此，我们首先要深入了解修改的深层机理。

1.1 修改软件的四个起因

为了简明起见，让我们来看看修改软件的四个主要起因：

(1) 添加新特性；

(2) 修正bug；

(3) 改善设计；

(4) 优化资源使用。

1.1.1 添加特性和修正 bug

添加特性看起来似乎是最直接的一种改动：软件原先是以某种方式运作的，现在用户提出需要这个系统能够做其他事情。

假设我们正在构建一个基于Web的应用，这时经理告诉我们她（指客户）想要把公司的logo从页面的左侧移到右侧。于是我们与她交谈，发现这件事情并不是想象中那么简单。她不但要移动logo，还想进行其他改动。她希望在系统的下一个版本中能够让它动起来。那么，这算是修正bug还是添加新特性呢？答案取决于你看待这个问题的角度。从客户的角度来看，她很明显是在要求我们修正一个问题。因为不久前她预览了网站，然后召集其部门的人员举行了一个会议，最后大家决定改动logo的位置，并要求更多一点的功能[1]。而站在开发者的立场上，这种改动则可以看成是添加一个全新的特性。开发者会说："如果客户不改变主意的话，我们的工作现在就已经算是完成了。"然而，对于某些公司，移动logo的位置只是看作bug修正，他们并不管开发团队为此而不得不从头开始做一些新工作的事实。

1. 即动态logo。——译者注