TURING 图灵程序设计丛书 程序员修炼系列

PRENTICE HALL PTR

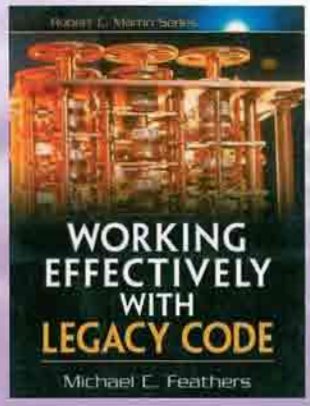
Working Effectively with Legacy Code

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

站在巨人的肩上

Standing on Shoulders of Giants

www.turingbook.com

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，旁边还有蜚人的昆虫飞来飞去，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

第一部分　修改机理

第二部分　修改代码的技术

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

我们可以认为以上这些纯属主观看法的差异：在你的眼里它是一次bug修正，而在我看来则是添加新特性，就这么简单。然而实际情况是，许多公司出于合同或质量方面的某些原因和目的，bug修正与特性添加是必须分开记录和解决的。从人的层面来看，我们可以在“我们是在添加特性还是在修正bug”这个问题上争论不休。然而从代码层面来说，这些终究不过是在修改代码以及其他遗留下来的东西罢了。不幸的是，这种关于究竟是bug修正还是特性添加的争执掩盖了某些从技术上来说对我们要重要得多的东西：行为改变。事实上，在添加新行为与改变旧行为之间存在着巨大的差异。

> 行为对于软件来说是最重要的一样东西。软件的用户要依赖于软件的行为。用户喜欢我们添加行为（前提是新的行为确实是他们所需要的），然而如果我们改变或移除了他们原本所依赖的行为（引入bug），那么他们就不会再相信我们。

回过头来，在前面提到的公司logo的案例中，我们是在添加新行为吗？是的。因为在改动之后，系统将会在页面的右侧显示logo。那么我们是否移除掉了某些行为呢？是的，因为页面的左侧将不会再有logo。

让我们再来看一个更为复杂的案例。假设客户想要在页面的右侧添加一个logo，同时在页面的左侧原本并没有任何logo。在这种情况下我们就是在添加新行为，但我们有没有移除任何已有行为呢？我们即将要放置logo的地方原本是否是由其他图案或文字占据的呢？

我们是在改变行为，添加行为，还是两者皆是？

事实上，我们可以抽出一个对于程序员来说更为有用的差别。即如果我们必须修改代码（HTML某种程度上也算代码），那么我们就是在改变行为。如果我们只是往其中添加代码并调用它，则通常是在添加行为。对此我们再来看一个例子。下面是一个Java类的方法：

4

```
public class CDPlayer
{
    public void addTrackListing(Track track) {
        ...
    }
    ...
}
```

该类拥有一个方法addTrackListing，我们通过该方法能够添加音轨列表。现在，让我们添加一个用于替换音轨列表的新方法：

```
public class CDPlayer
{
    public void addTrackListing(Track track) {
        ...
    }
    public void replaceTrackListing(String name, Track track) {
        ...
    }
    ...
}
```

当添加该方法时，我们是在往该应用程序中添加新行为呢，还是改变了现有行为？答案是：两者都不是。添加一个方法并不会改变代码的行为，除非我们以某种方式调用了该方法。

现在我们来进行另一处修改，往这个CD播放软件的用户界面上放置一个新的按钮。该按钮的功能是让用户能够替换音轨列表。这一举动不仅添加了replaceTrackListing方法所指定的行为，同时也细微地改变了该软件的行为。因为有了这个新按钮，用户界面的渲染（render）就与以前不一样了，用户界面的显示大概需要多耗一毫秒（用于渲染新的按钮）。所以，想要完全不改变现有行为地添加新行为几乎是不可能的。

1.1.2　改善设计

改善设计则是另一种软件修改。当我们想要改变既有软件的结构和组织，以令其更易于维护时，通常也会希望能够在此过程中不改变其行为。倘若在这个过程中丢掉了某个行为，我们通常会将其称作引入了一个bug。许多程序员通常并不试图改善既有设计，其主要原因之一就是这一举动相对容易导致行为丧失或坏行为的诞生。

在不改变软件行为的前提下改善其设计的举动称为*重构*（refactoring）。重构背后的理念是，如果我们编写测试以确保现有行为不变，并在重构过程中的每一小步都小心验证其行为的不变性的话，我们就可以在不改变软件行为的前提下通过重构使其更具可维护性。多年来人们一直都在做着清理[1]系统中既有代码的事情，而重构的出现则是近几年的事。重构与一般的代码清理不同，在重构时我们并不只是在做那些低危险性的工作（如重整源代码的格式）或侵入性的危险工作（如重写代码块），而是进行一系列的结构上的小改动，并通过测试的支持来使得代码的修改更容易着手。从改变的角度来说，重构的关键在于在进行重构的过程中不应当有任何功能上的改变。（不过行为可以稍有改变，因为你在代码结构上的改动可能会导致性能上的改变，其性能可能会变得差一点，也可能会变得好一点。）

1.1.3　优化

优化与重构类似，但目标不同。对于重构和优化，我们都可以说：“我们在进行修改的过程中将会保持功能不变，但我们可能会改变某些其他东西。”对于重构来说，这里的“某些其他东西”就是指程序的结构，我们想让代码更容易维护。而对于优化来说，“某些其他东西”则是指程序所使用的某些资源，通常指时间或内存。

1.1.4　综合起来

重构与优化的相似性看起来似乎有点奇怪。它们彼此间的相似性看上去比添加特性与修正bug之间的相似性还要高。然而，真的是这样吗？重构与优化之间的共同点就是在改变某些东西的过程中保持软件的功能不变。

---

1. 这里的“清理”并非“清扫掉”的意思。原文为clean up，含有“使整洁干净”的意思，是指在这个过程中可能会删掉代码，也可能会修改既有代码。之所以不译为“整理”是因为“整理”没有体现出潜在的无用代码消除这一行为，而“清理”则含有“清洁整理”的双重意思。——译者注

一般而言，当对一个系统进行修改的时候，其三个方面可能会发生改变：结构、功能以及资源使用。

让我们来看一看，当进行上述四种修改的时候，系统通常在哪些方面发生改变，以及哪些方面基本保持不变（通常其三个方面都会发生改变，但让我们来看看什么是典型的情况）：

	添加特性	修正bug	重　构	优　化
结构	改变	改变	改变	—
功能	改变	改变	—	—
资源使用	—	—	—	改变

从表面上看，重构和优化的确是蛮相似的。它们都保持功能不变。但倘若我们将新功能的出现分离出来考虑又会怎样呢？当我们添加一个新特性时，通常是在维持现有功能不变的前提下添加新的功能。

	添加特性	修正bug	重　构	优　化
结构	改变	改变	改变	—
新功能	改变	—	—	—
功能	—	改变	—	—
资源使用	—	—	—	改变

添加特性、重构以及优化这三种举动统统都维持既有功能不变。如果仔细观察bug修正的话，我们会发现它是会改变（既有）功能的，只不过这种改变比起那些没被改变的既有功能通常显得非常微小罢了。

特性添加和bug修正与重构和优化是非常相似的。在所有四种情况下，我们都想要改变某些功能、某些行为，但我们想要保持不变的地方则要多得多（见图1-1）。

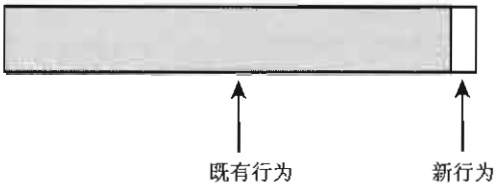


图1-1　行为保持

图1-1很好地表现了当对系统进行修改时所发生的情况，那么在实际工作中这幅图对我们来说又意味着什么呢？从积极的角度来说，这幅图似乎告诉我们应当将注意力集中在什么方面。我们要确保所修改的少数几处已经正确修改了。从消极的角度来说，我们不仅需要关注这些，还得知道如何保持其他行为不变。然而遗憾的是，保留既有行为不变并非意味着只要不碰那些代码就成。我们要知道这些行为并没有发生改变，而这可能是件棘手的事情。需要保持的行为的数量通常是非常巨大的，不过这倒不是什么大问题。问题在于我们通常并不知道在修改的过程中哪些行为存在被连带改变的风险。如果我们知道，就可以将精力集中在那些行为上而不用管其他的。所

以，要想安全地进行修改，关键就在于“理解”。

保留既有行为不变是软件开发中最具挑战性的任务之一。即便是在改变主要特性时，通常也有很多行为是必须保留不变的。

1.2　危险的修改

行为保持是一项巨大的挑战。在我们需要作出修改并保持行为时，往往伴随着相当大的风险。
7 为了减小风险，我们要考虑下面这三个问题：

(1) 我们要进行哪些修改？

(2) 我们如何得知已经正确地完成了修改？

(3) 我们如何得知没有破坏任何（既有的）东西？

如果所作改动是有风险的话，你能够承担得起多少改动？

我曾共事过的大多数团队都曾试图以一种非常传统的方式来控制风险。他们把对代码基的改动数量降至最低。有时候这是一种团队策略：“如果没有被破坏，就别去修正。”开发者在进行修改的时候是非常谨慎的。“什么？为此创建一个新方法？不不不，我还是把这几行代码直接放到这个现成的方法里面算了，这样我就可以看到新老代码在一起。况且这种做法费力少，也更为安全。”

人们可能会认为可以通过“避免”二字诀来将软件问题的数量降至最低，然而遗憾的是，问题总是不可避免。当我们避免创建新类和新方法时，既有的类和方法就会变得越来越庞大，越来越难以理解。当在任何大型系统中进行修改时，你可能需要一点时间来熟悉一下将要修改的区域。这时好的系统和差的系统之间的差别就体现出来了。对于前者，当你熟悉了待修改的区域之后，你会对将要进行的修改充满信心。而对于那些结构糟糕的代码，从理清存在的问题到着手进行修改的过程简直就像是为了躲避一只老虎而跳下悬崖一样痛苦。你一再犹疑：“我真的准备好这么做了吗？唔，好吧，我想我别无选择。”

避免修改还会导致其他不良后果。如果人们不做修改代码的工作的话，久而久之他们就会对此感到生疏。即便是将一个较大的类分解成几个小类的工作也可能会因为生疏而变得棘手起来。要想保持熟练，唯一的途径就是经常练习，如一个星期进行好几次。熟能生巧之后，这件事情对你来说就变得像例行公事一样自然而然了。你也会变得越来越精于判断，知道哪些代码可以分解哪些则不可以，而且做起来也容易得多。

避免改动的最后一个后果就是会带来恐惧心理。不幸的是，许多团队都在忍受着很重的恐惧心理，而且每况愈下。通常他们并不知道他们的恐惧到底有多重，直到他们学到更好的技术，而恐惧心理也随之减退[1]。

1. 这里表面上讲恐惧心理，实质上是说总惧怕进行修改从而导致系统结构日益糟糕的情况，而开发者们又采取鸵鸟战术，所以“并不知道恐惧到底有多重”（即系统到底变得有多糟糕），直到有一天他们学到了更好的技术，开始去对系统进行改善的时候，他们才会发现原来系统已经在长期的积累下变得难以理解了，这时他们才真正开始意识到问题有多严重了。——译者注

我们已经讨论了，避免修改是不可取的做法，然而既然不能避免，那我们又该怎么做呢？答案是我们应该更努力地去修改。或许我们可以雇佣更多的员工，这样每个人就会有足够的时间来进行分析，仔细审查所有的代码并“正确地”进行修改。更多的时间和详细审查应该会让修改变得更为安全。但果真如此吗？在所有的代码审查完毕之后，究竟有没有人确切知道他们是否把事情做对了呢？

第2章 带着反馈工作

对系统进行改动有两种主要方式。我喜欢将它们分别称为*编辑并祈祷*（edit and pray）和*覆盖并修改*（cover and modify）。遗憾的是，前一种方式几乎可算是业界的标准做法。使用这种方式来进行改动时，先是仔细地计划你所要进行的改动，并确保自己理解了将要修改的代码，然后再开始改动。结束之后，运行修改后的系统，看看所做的改动是否已经生效，再然后就是对系统整体的修复，以确保改动没有破坏什么东西。这最后一个步骤是必不可少的，且非常重要。在进行改动时，你希望并祈祷能把事情做对，在完成改动后，你要用额外的时间来验证是否真的把事情做对了。

从表面上来看，“编辑并祈祷”这种方式似乎意味着“小心下手”，这是一件需要专业水平的工作。在事情的一开始就需要小心，而当改动变得非常具有侵入性的时候，你还需分外小心，因为这时出错的可能性就更大了。然而可惜的是安全性并不取决于你的细心程度。我想谁都不会仅仅因为一位外科医生会非常细心地进行手术就允许他拿着切黄油的刀来切你吧。精湛的软件改动就像精湛的外科手术一样，除了细心之外还要有深厚的技术。如果没有辅以正确的工具和技术，即便“小心下手”也起不到多大作用。

而“覆盖并修改”则是另一种方式。它背后的理念在于，在我们改动软件的时候张开一张安全网。这里所谓的“安全网”并不是指放在桌子底下，当我们从椅子上跌下去时能够托住我们的那种，而是像张斗篷一样“盖”在我们进行修改的代码上面，以确保糟糕的改动不会泄漏出去并感染到软件的其他部分。覆盖软件即意味着用测试来覆盖它。当对一段代码有一组良好的测试时，我们就可以放心对它进行修改，并快速检验出修改是好是坏。我们仍然会辅以同样的细心，但有了测试的反馈结果，我们就得以进行更为细致的修改。

如果对这种使用测试的方式不熟悉的话，上面这些话听起来或许就有点儿令人不解了。传统上，测试总是在开发之后编写并执行的。一组程序员编写代码，另一组测试员在代码写好之后对其进行测试，看看它们是否满足特定的要求。一些非常传统的开发团队正是以这种方式来开发软件的。这类团队也能得到反馈，但整个反馈周期非常长，往往是在几个星期乃至几个月后，另一个小组的人才会告诉你是否已经把事情做对了。

以这种方式进行的测试实际上可以表述为“通过测试来检验正确性”。虽说这是一个很好的目标，但测试还可以用来做其他事情，我们可以“通过测试来检测变化”。

用传统的术语来说，这叫做*回归测试*（regression testing）。我们周期性地运行测试来检验已

知的良好行为，以便确诊软件是否还像它以前那样工作。

当要动手进行改动的区域由测试包围着时，这些测试的作用就好比一把“软件夹钳（vise）”。你可以用这把“软件夹钳”来固定住目标软件的大部分行为，只改动那些你真正想要改动的地方。

软件夹钳

夹钳：名词，由金属或木头做成的钳夹工具，通常由两个靠螺旋或杠杆进行开合的部件组成，用于木工业或五金业中使物件定位。[《美国英语传统词典》，第4版]

能够起到检测改动的作用的测试就好比是为我们的代码上了一把夹钳，使代码的行为被固定起来。于是当进行改动时，我们得以知道在一个特定的时间只改动某一处的行为。简而言之，我们得以掌控我们的工作。

回归测试是一个好主意。那么为什么人们不更频繁地进行回归测试呢？这是因为回归测试存在着一个小问题：通常进行回归测试时，都是在应用程序接口（application interface）层面进行测试。不管目标应用是Web应用、命令行程序，还是GUI程序，回归测试在传统上都是看作一种应用层面的测试。然而这只是一个不幸的传统观念。事实上，我们从回归测试中得到的反馈信息是非常有用的。所以若能把回归测试运用到一个更细粒度的层面则将是大有裨益的。

为证实这一点，让我们来做一个思想实验：假设我们正单步跟踪一个大函数，该函数包含大量复杂的逻辑。我们分析、思考，并与更熟悉这块代码的人交流，然后我们着手进行修改。我们想要确保所做的改动没有破坏任何东西，然而如何才能确保这一点呢？幸运的是我们有一个质量小组，他们有一组回归测试，可以在夜里运行这组测试。于是我们打电话给他们，让他们安排一次测试，他们答应了，可以替我们安排一次夜里的回归测试。幸运的是，这个电话打得早，因为其他小组通常会在星期三左右要求他们安排回归测试，要是再晚一些的话，他们可能就腾不出可用的时间段和机器了。听了这话我们就放心了，回头继续工作。我们还有五处像刚才那样的改动要做，而且都是位于像刚才那样复杂的地方。同时我们心里也清楚，还有其他几个人也在像我们一样忙活着修改代码并等着安排测试呢。

10

第二天早晨我们接到一个电话。测试部门的Daiva告诉我们，第AE1021和AE1029号测试昨晚失败了。她不能肯定是不是由于我们所做的改动导致的，但她给我们打了电话，因为她知道我们会帮她搞定这件事情。我们会进行调试，查出失败是否由于所做的某处改动还是因为其他原因。

以上场景听起来真实吗？不幸的是，它恰恰是非常真实的。

让我们来看另外一个场景。

我们需要对一个相当长且复杂的函数进行改动。幸运的是，我们发现有一组针对它的单元测试。最后一次接触这段代码的人编写了一组大约20个单元测试，彻底地检验了这段代码。我们运行这组测试，发现全部通过。接下来我们就浏览这些测试，以便对这段代码的实际行为有一些认识。

我们准备好做改动了，然而却发现很难想出具体进行改动的方法。代码模糊不清，我们很希望能够在着手之前先更好地认识代码。现有的测试并不能覆盖所有的情况，因此我们想让代码变得更清晰一些，这样我们才能够对进行的改动更有信心。除此之外，我们不想自己或任何其他人

再次经历这样一个痛苦的过程，那实在太浪费时间了！

我们开始对代码做一点点重构。我们将一些方法抽取出来，并移动一些条件逻辑。在进行的每一步细小的改动后，我们都会运行上面提到的那套单元测试。几乎我们每次运行它的时候都是通过的。就在几分钟前我们犯了一个错误，反转了一个条件逻辑，然而单元测试迅速给出了失败的结果，于是我们得以在大约一分钟内纠正了所犯的错误。当我们完成重构之后，代码变得清晰多了。我们完成了要做的改动，而且确信我们的改动是正确的。接下来，我们添加了几个测试来验证新加上去的特性。于是，面对这些代码的下一个程序员做起来就会轻松得多，而且他会有覆盖其所有功能的全套测试。

你是希望在一分钟内就获得反馈呢？还是希望等一整个晚上？以上哪个场景更有效率？

单元测试是用于对付遗留代码的极其重要的组件之一。系统层面的回归测试的确很棒，然而相比之下，小巧而局部性的测试才是无价之宝，它们能够在进行改动的过程中不断给你反馈，使重构工作的安全性大大增强。

11

2.1 什么是单元测试

术语单元测试（unit testing）在软件开发领域有着悠久的历史。在大多数有关单元测试的观念中都有这么一个共同的理念，即它们由一组独立的测试构成，其中每个测试针对一个单独的软件组件。那么组件又是什么呢？实际上组件有多种定义，不过在单元测试这一领域，我们通常关心的是一个系统的最为“原子”的行为单元。譬如在过程式程序设计的代码中，“单元”一般来说指的就是函数，而在面向对象的代码中则指的是类。

测试用具

本书中我使用了术语测试用具（test harness）。该术语是泛称，代表我们为了检验软件的某部分而编写的测试代码以及用来运行这些测试代码的代码。我们可以针对代码使用多种不同的测试用具。第5章中讨论了xUnit测试框架以及FIT框架。这两者都可以用来进行本书中描述的测试工作。

那么，我们究竟能否做到只测试系统中的某一个函数或类呢？在过程式系统中，通常是难以孤立地测试函数的，因为这种系统中的情况往往是顶层的函数调用其他函数，后者再调用另一些函数，最后直到机器层。而在面向对象的系统中，单独测试类则要简单一点，然而实际上类却往往并不是“离群索居”的生物。想想看，在你写过的所有类中有多少是没有使用到别的类的？非常少这些极个别分子往往是那些小型的数据类或数据结构类，如栈和队列（甚至就算是这样的类也可能会用到其他类）。

测试的隔离性是单元测试的一个重要方面，然而为什么说它是重要的呢？毕竟，当整合软件的各个部分时还可能出现许多错误。难道不应该是那些能够覆盖代码中的广泛功能区域的大型测试更为重要吗？诚然，它们是重要的，我并不否认这一点，然而大型测试存在着一些问题：

- **错误定位**：测试离被测试者越远，就越难确定测试失败究竟意味着什么。要想精确定位

测试失败的根源往往需要耗费大量的工作。你得检查测试输入、还要检查失败本身，然后还得确定这次失败发生在从输入到输出的执行路径上的哪一点。虽说对于单元测试来说这样的工作也是免不了的，然而通常其工作量微乎其微。

- **执行时间**：大型测试往往需要更长时间来运行。而这种长时耗性往往让人无法忍受。需要太长时间运行的测试，结果往往是无法运行。
- **覆盖**：在大型测试中，往往难以看出某段代码与用来测试它的值之间的联系。我们通常可以通过覆盖工具来查出某段代码是否被一个测试覆盖到了，但当添加新的代码时，可能就需要花费可观的工作量来创建检验这段新代码的高层测试了。

大型测试最令人不能接受的事情就是可以通过频繁地运行测试来实现错误定位，然而这偏偏又是难以实现的。假设我们运行测试并且测试通过，接着我们做了一点点改动于是测试失败了，这时我们就能够精确地知道问题是在哪儿被触发的，就是我们最后做的那点改动中的某处地方。于是我们可以将改动回滚，并重新尝试改动。这一场景看上去也许没什么问题，然而如果我们的测试相当大，其执行时间可能长得令人无法忍受，可想而知我们更可能做的事情便是尽量避免去运行它，于是就无法达到错误定位的目的了。

而单元测试则做到了大型测试所不能做到的那些事情。利用单元测试可以独立地对某一段代码进行测试。我们可以将测试分组以便在某些特定条件下运行某些特定的测试，并在其他条件下运行另一些测试。我们还可以迅速定位错误。如果认为在某段代码中存在着一个错误而且又可以在测试用具中使用这段代码的话，我们通常能够迅速地编写出一段测试，看看我们所推测的错误是不是真的在那里。

下面是好的单元测试所应具备的品质：

(1) 运行快；

(2) 能帮助我们定位问题所在。

在业界，人们在判断某个特定的测试是否是单元测试这个问题上常常摇摆不定。如果一个测试中涉及了另外一个产品类，那它还能算是单元测试吗？为了回答这个问题，我们回到刚才提到的两点品质上来，即该测试运行起来快不快？它能帮我们快速定位错误吗？比如有些测试较大，其中用到了好多类。那么实际上这种测试或许看上去像是小型的集成测试。就它们自身而言，可能运行起来比较快，然而要是你将它们一起运行呢？一个测试如果不仅测试了某个类还测试了与该类一起工作的几个类，那么它往往会“越长越大”。如果你当时不花时间来使得一个类能够在测试用具中单独实例化的话，难道你还能指望当更多的代码被添加进系统之后这件事会变得更容易吗？永远也不会。人们会不断推诿，并且随着时间的推移，原本短小的测试可能会变得需要十分之一秒才能执行完。

一个需要耗时十分之一秒才能执行完的单元测试就已算是一个慢的单元测试了。

我说这话是认真的。在我写作本书时，十分之一秒对于单元测试来说简直就像一个世纪一样。

不信的话让我们来做一点简单的算术吧：假设你有一个项目，其中包含3 000个类，每个类平均
13 大约有10个测试，一共算起来就是30 000个测试。倘若这些测试个个都耗时十分之一秒才能运行完的话，那整个项目测试一遍需要多少时间呢？将近一个小时！对于反馈来说这段等待时间可不短。什么？你的项目没有3 000个类？那一半总有吧，这样算下来也仍然要等半个小时呢。另一方面，假如我们的测试只需耗时百分之一秒呢？很显然，我们一下子从需要等一个小时变成了只需等5到10分钟！这样的话，虽说我还是比较谨慎的只取出其中的部分单元测试来用，但哪怕每隔几个小时就将它们全部运行一遍我也不再害怕。

此外，根据摩尔定律，在我有生之年我有望看到即便是在最大型的系统上测试反馈也能在近乎一瞬之间完成。我猜到那时候在这类系统中工作就会像是在一堆会“反咬一口”的代码中工作一样。一旦你改错了一步，系统就会立即“反咬一口”让你知道。

> 单元测试运行得快。运行得不快的不是单元测试。
>
> 有些测试容易跟单元测试混淆起来。譬如下面这些测试就不是单元测试：
>
> (1) 跟数据库有交互；
>
> (2) 进行了网络间通信；
>
> (3) 调用了文件系统；
>
> (4) 需要你对环境作特定的准备（如编辑配置文件）才能运行的。
>
> 当然，这并不是说这些测试就是坏的。编写它们常常也是有价值的，而且你通常也会在单元测试用具内来编写它们。然而，将它们跟真正的单元测试区分开来还是很有必要的，因为这样你就能够知道哪些测试是你可以（在你进行代码修改的时候）快速运行的。

2.2　高层测试

单元测试的确很棒，但高层测试也有其一席之地。所谓高层测试便是那些覆盖了某个应用中的场景和交互的测试。高层测试可以用来一下子就确定一组类的行为。能够这样做往往就意味着你可以更容易地为单个类编写测试。

2.3　测试覆盖

那么在一个遗留项目中我们究竟该如何着手进行修改呢？我们首先注意到，如果可以选择的话，在进行修改时有测试“罩着”总是要安全一些的。对代码的修改可能会引入bug，因为我们
14 毕竟是人而不是神。但假如在修改代码之前先用测试将代码“护住”，我们就能更容易地捕获到在改动过程中所犯的错误了。

图2-1展示了一小组类。我们想要改动`InvoiceUpdateResponder`的`getResponseText`方法以及`Invoice`的`getValue`方法。这两个方法是我们的改动点，我们可以通过为它们所在的类编写测试来覆盖它们。

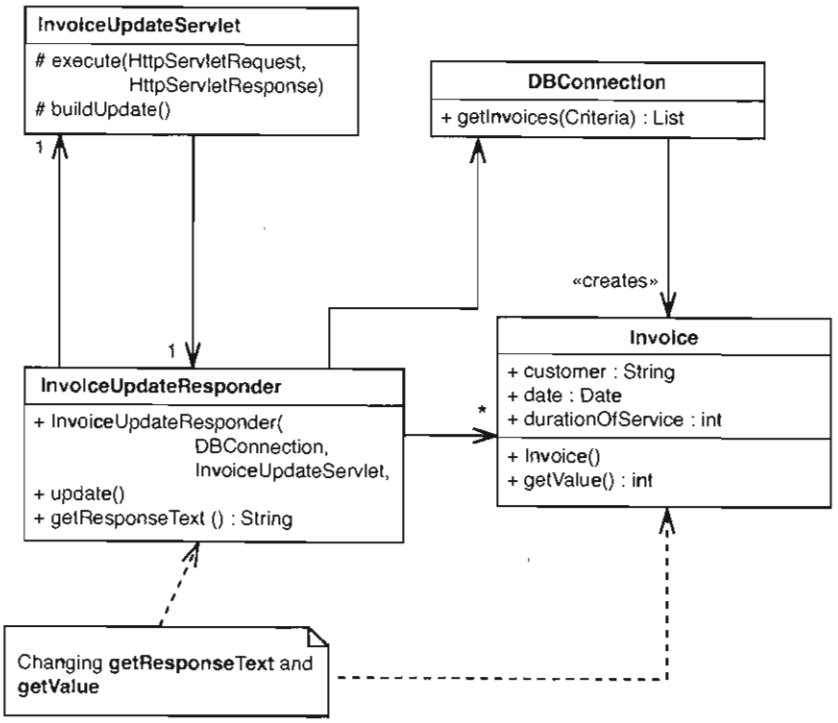


图2-1　发票更新类

要想编写并运行测试，我们先要能够在一个测试用具中创建InvoiceUpdateResponder和Invoice的实例。那么我们能否做到这一点呢？看起来创建Invoice的实例可以不费吹灰之力地完成，因为它有一个无参的构造函数。而InvoiceUpdateResponder的情况则可能要复杂一些，它的构造函数接受一个DBConnection，参数这是一个数据库连接，必须真正连接到一个实实在在的数据库。问题来了，在测试中我们该怎样处理这种情况呢？我们是否要为此建立一个数据库，并在其中存上测试所需的数据呢？这么做的工作量可不小。而且有没有考虑到涉及数据库的测试会比较慢呢？无论如何，我们在做这个测试的时候对数据库并不特别关心，只是想覆盖我们对InvoiceUpdateResponder和Invoice的改动。此外还有一个更大的问题，即InvoiceUpdateResponder的构造函数需要一个InvoiceUpdateServlet作为参数。创建一个这样的对象可想而知有多麻烦。当然我们可以改动一下InvoiceUpdateResponder的代码，让它不再接受InvoiceUpdateServlet为参数。如果InvoiceUpdateResponder只是需要从InvoiceUpdateServlet那里获取很少一点信息的话，我们就可以将这些信息传给它，而不是传给它整个servlet，然而，在做上述改动的时候我们是不是也需要做个测试来确保改动的正确性呢？

15

以上这些问题都属于依赖问题。当一个类直接依赖于某些难以在测试中使用的东西时，这个类就是难以修改和处理的。

依赖性是软件开发中最为关键的问题之一。在处理遗留代码的过程中很大一部分工作都是围绕着“解除依赖性以便使改动变得更容易”这个目标来进行的。

那么，我们具体又该怎么做呢？在不改变代码的前提下我们如何才能将测试安置到位呢？不幸的是，在许多情况下想要做到这一点是不大容易的，某些时候甚至根本不可能。就在我们刚刚看到的这个例子中，我们当然可以通过使用一个真实的数据库来解决DBConnection问题，然而servlet问题又该怎么办呢？我们是不是也要创建一个完整的servlet对象并将它传递给InvoiceUpdateResponder的构造函数呢？我们能否将这个servlet设置到正确的状态呢？可能吧。但假如我们将要测试的是一个图形用户界面的桌面应用程序又该怎么办呢？这样一个应用程序也许并没有任何可供我们测试时利用的可编程接口，其逻辑可能被捆绑在GUI类当中[1]。这时候我们该怎么办呢？

遗留代码的困境

我们在修改代码时，应当有测试在周围“护”着。而为了将这些测试安置妥当，我们往往又得先去修改代码。

在上面的Invoice例子当中，我们可以试着在一个更高的层别来进行测试。如果对于某个特定的类来说，不改变它就难以为它编写测试的话，那么转而去测试使用它的那些类往往会简单一些。然而不管怎么样，我们通常最终还是免不了要在某个点上解开类之间的依赖。在当前的这个例子中，我们可以解开InvoiceUpdateResponder对InvoiceUpdateServlet的依赖：只需将InvoiceUpdateResponder真正需要的东西传给它就行。InvoiceUpdateResponder需要的是InvoiceUpdateServlet所持有的一组发票ID。同样，我们也可以解开InvoiceUpdate-Responder对于DBConnection的依赖：只需引入一个接口（IDBConnection）并将Invoice-
16 UpdateResponder改为使用该接口即可。图2-2展示了这些类在上述改动之后的样子和关系。

那么，在没有测试保护的情况下进行上述的重构到底安不安全呢？实际上它们可以是安全的。上述的用于解开InvoiceUpdateResponder对InvoiceUpdateServlet和对DBConnection的依赖的两种重构手法分别称作朴素化参数（Primitivize Parameter，302页）[2]和接口提取（Extract Interface，285页）。在本书的最后，解依赖的技术一部分中对它们有详细描述。在解依赖时，我们通常可以采用编写测试的手段来让较具侵入性的修改更为安全。诀窍就在于要非常保守地进行上述最初的重构。

1. 譬如说那些被写到了消息响应函数（一般位于GUI类当中）中去的代码逻辑。——译者注
2. 这里用“朴素化”而非“基元化”、“简化”之类的词是因为“简化”一词含义过于模糊，而且不够突出。“基元化”容易产生将参数类型变成基元类型的错误联想，实际上Primitivize Parameter并不一定要将参数简化成基元类型。“朴素化”在这里则形象表达出了“as simple as possible, but no simpler”的意思，即将原本复杂华丽但无必要那般的参数变得“朴素”，该要多少信息就给多少信息。——译者注

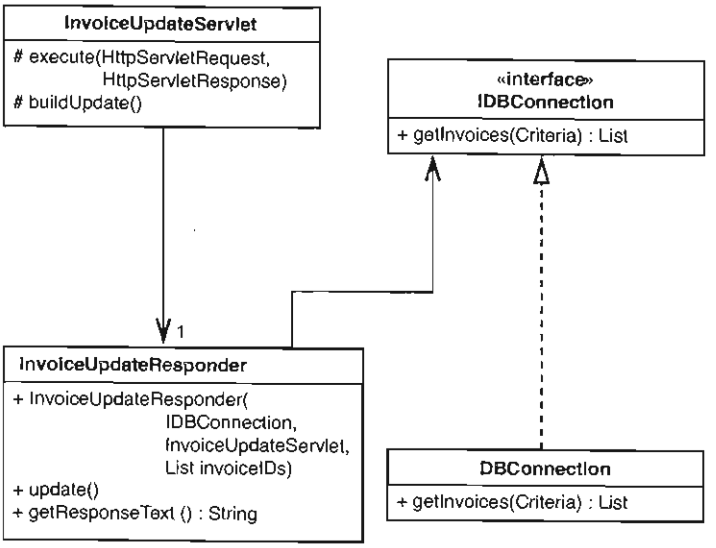


图2-2　解除依赖后的发票更新类

当我们的改动可能会引入错误的时候，保守地进行改动就成了不二之选，然而有时候（为了让测试覆盖代码而解依赖）结果代码却并不像前面的例子中那样光鲜漂亮，例如我们可能只是为了能够将测试安置到位而为某个方法引入了某个在产品代码中并不严格需要的形参，或者以古怪的方式将某个类分裂开了。当这么做的时候，我们可能最终会令代码看上去稍微糟糕一些。另一方面，如果不那么保守，则我们可以立即解决这个问题。但话虽如此，具体还要看这么做会带来多大的风险。如果错误是（它们的确通常是）个重要的考虑因素的话，保守改动常常是有好处的。 17

当在遗留代码中解依赖时，你常常不得不暂时将自己的审美感放在一旁。有些依赖能够干净利落地解除，而有些从设计的角度来看最终还是解决得不那么完满。这就好像做手术总要有一个刀口一样，刀口在缝合之后可能会变成一道疤痕，你的改动也可能会在代码中留下“疤痕”，然而“疤痕”之下的东西则已得到了治愈。

而且，如果以后能够用测试覆盖“疤痕”四周（即你当初解依赖的点）的话，你就可以将“疤痕”也抹掉了。

2.4　遗留代码修改算法

以下算法可以用于对遗留代码基进行修改：

(1) 确定改动点；

(2) 找出测试点；

(3) 解依赖；

(4) 编写测试；

(5) 修改、重构。

每日对付遗留代码的工作其实就是改改改，不过并不是随意地改。我们想要在做出能够带来价值的功能性改动的同时使系统中的更多部分进入测试的笼罩之下。在每次编完程之后，我们应当不仅能够指出那些提供了某些新特性的代码，还要能够指出其测试。随着时间的推移，代码基的受测试部分将会变得越来越大，就好像在海里不断生长的岛屿一样。在这些“岛屿”之上工作会容易得多。渐渐地，岛屿变成更大的陆地。最终你将能够在一块全面由测试覆盖的代码大陆板块上工作。

下面就让我们逐个看看以上描述的这些步骤，以及本书是如何帮助你实施它们的。

2.4.1 确定修改点

你所要进行修改的地点与代码的架构联系很紧密，前者敏感地依赖后者。如果你对代码设计
18 的理解程度不足以让你觉得是在正确的地点进行修改的话，请阅读第16章以及第17章。

2.4.2 找出测试点

有些情况下找出测试点殊为易事，然而对于遗留代码来说这事儿往往就不那么容易了。请阅读第11章以及第12章。这些章节中介绍的一些技术可以用来确定出对于给定的改动要在哪些地方编写测试。

2.4.3 解依赖

依赖性往往是进行测试的最为明显的障碍。这表现在两个方面：一是难以在测试用具中实例化目标对象；二是难以在测试用具中（调用）运行方法。通常在遗留代码中你得先解依赖而后才能将测试安置到位。理想情况下我们为安置测试而进行解依赖的动作本身也应当受到测试的保护，这样一旦我们在解依赖时做了不妥当的事就会及时得知，然而这往往只是个奢望。请阅读第23章，其中描述的一些实践技术可以使你在将一个系统放入测试之下时所下手的第一刀（即第一处改动）变得更安全。之后再看一下第9章和第10章，这两章中描述的场景展示了如何解决一般的依赖问题。这些章节都大量引用了书的后面部分（第25章）所列举出来的解依赖技术，但并非全部。所以我建议你花点时间浏览一下后面罗列的目录，里面有更多用于解依赖的技术。

依赖性还会在另一种场合下显现出来，即当我们有一个关于测试的想法然而却不能容易地将其编写实施时。如果你发现之所以不能编写测试是因为大型方法中的依赖性未解决的话，请阅读第22章。如果你发现虽能够解依赖，但用在创建测试上的时间太长了的话，可以参考一下第7章。该章中描述的一些额外的解依赖技术可以用来减少平均构建时间。

2.4.4 编写测试

我发现为遗留代码编写的测试和为新代码编写的测试有些许不同。关于测试在遗留代码工作
19 中的角色的进一步认识可参考第13章。

2.4.5 改动和重构

我提倡使用测试驱动的开发方式（TDD）来往遗留代码中添加特性。第8章中就描述了TDD以及其他用于添加新特性的技术。在对遗留代码做了一番改动之后，我们对其中存在的问题往往也就有了更多的了解，而所编写的用于辅助添加特性的测试则常常也能够“保护”我们去进行一些重构。第20章、第22章以及第21章这三章涵盖了可以用于开始将你的遗留代码挪上一条朝着更良好的结构前进的正轨的大部分技术。别忘了我在这些章节中介绍的东西只是“婴儿阶段”。它们并没有展示如何令你的设计变得理想、干净或者富含模式。这些问题在很多书中都有讨论，而当你有机会去使用这些技术时，我鼓励你去这么做。而本书中的上述章节则是告诉你如何令设计变得更好，这里更好的含义取决于具体问题的上下文，往往只是意味着比原先的设计可维护性更好一些。但你可别低估这一点。往往就是那些最为简单的事情（例如将一个大型的类分解成多个小类以便能够更容易下手）恰恰能够在应用程序中带来显著的不同，尽管这些事情看上去可能有点儿没啥创意。

2.4.6 其他内容

本书的其他内容展示了如何在遗留代码中进行改动。接下来的两章则包含了一些关于遗留代码工作的三个关键概念：感知、分离和接缝的背景知识。

20

第3章 感知和分离

理想情况下我们无需对一个类做任何特殊处理就可以开始对其进行测试了。在理想的系统中，我们应当能够在测试用具中创建任何类的对象并开始对其进行测试。我们应当能够创建它们的对象，为它们编写测试，然后接着做其他事情。如果事情真有这么简单的话我也就无需写这些了，遗憾的是事情往往没那么简单。类之间的依赖会导致我们难以将一簇对象放入测试当中。我们可能想要创建某个类的对象并向它“询问”一些问题，然而要想创建这么一个对象我们或许又得先创建另一个类的对象，而后者的创建又要用到其他类的对象，如此一环扣一环，直到几乎把整个系统都给扯进了测试用具之中。对于某些语言来说这或许并不算什么大问题。然而对于其他一些语言，尤其是对C++来说，若是不先解依赖，则光是连接所耗的时间就可能会令“快速周转”变成一个空想。

对于那些在开发时并没有同步编写单元测试的系统，我们常常得先解依赖而后才能够将某个类放入测试用具中去，然而这并非是进行解依赖的唯一原因。有时我们想要测试的类会对其他类产生一些影响，而我们的测试需要获知这些影响。面对这类情况，有时我们可以利用被影响的类的接口来感知到这些影响，而其他时候（如果没有适当解依赖的话）则不能。总而言之，唯一的办法就是通过伪装成被影响的类来直接感知到它所受到的影响。

通常，如果我们想要将测试安置到位，有两个理由去进行解依赖：感知和分离。

(1) 感知：当我们无法访问到代码计算出的值时，就需要通过解依赖来“感知”这些值。

(2) 分离：当我们无法将哪怕一小块代码放入到测试用具中去运行时，就需要通过解依赖将这块代码“分离”出来。

下面就是一个例子。NetworkBridge是某个网络管理应用程序中的一个类：

```
public class NetworkBridge
{
    public NetworkBridge(EndPoint [] endpoints) {
        ...
    }

    public void formRouting(String sourceID, String destID) {
        ...
    }
    ...
}
```

NetworkBridge的构造函数接收一个EndPoint的数组，并通过一些本地硬件来管理它们的配置。NetworkBridge的用户能够通过其功能跟踪数据流从一个端点到另一个端点，具体实现上则是通过改变相应EndPoint的设置，EndPoint的每个实例都打开一个套接字，并通过网络跟一个特定的设备进行通信。

以上只是对NetworkBridge的功能的一个简单描述。当然，要讲的细节还有很多，这里只从测试的角度来说，目前的状况已经出现了一些明显的问题。比方说，如果我们想要为NetworkBridge编写测试的话，该如何来编写呢？NetworkBridge类的对象在构造时完全可能对某个实际的硬件设施进行调用。那么这是否意味着我们在创建它们时应当准备好相应的硬件设施呢？更糟的是，我们究竟如何才能得知它们对该硬件设施或那些端点做了什么？从我们的角度来说，这个类就是一个黑盒（closed box，也称black box）。

话说回来，这或许并不算太坏。或许我们可以编写一些代码来嗅探经过网络的包；或许我们可以为NetworkBridge准备相应的硬件设施，这样至少其对象在创建的过程中就不会因找不着硬件而中止；又或许我们应该布一下线，这样我们就有了一簇本地的网络端点，并使用它们来辅助测试。这些解决方案都可行，只不过涉及的工作量太大了。我们想要在NetworkBridge中改动的那部分逻辑或许根本就不需要用到上面提到的这些东西，只是我们没法把这部分逻辑单独拿出来运行看看它究竟是怎么工作的。

这个例子不仅阐释了感知问题，同时也解释了什么是分离问题。对于上面提到的NetworkBridge类来说，我们无法感知其上的方法调用所产生的效果或影响，而且也无法将它跟它所在的应用程序的其他部分分离开来单独运行。

哪个问题更严重？感知还是分离？这个问题并没有明确的答案。一般来说两者都需要，而且它们都是我们进行解依赖的理由或动机。但有一件事是明确的：分离的途径是多种多样的。实际上，关于这个主题，在本书的后面罗列了一系列的解依赖技术。但是关于感知的技术却有最突出的一种。 22

3.1 伪装成合作者

在处理遗留代码的过程中要面对的一个重要问题就是依赖。如果想通过仅执行一段代码本身来看看它做了什么的话，通常就得先解开它跟其他代码之间的依赖才行。然而事情几乎从不会那么简单。往往这里的其他代码正是容易感知到我们的行为所产生影响的地方。如果可以用另一些代码来取代其位置并通过这些代码来进行测试的话，我们就可以放手编写测试了。用面向对象的术语来说，这里的另一些代码就是*伪对象*（fake object）。

3.1.1 伪对象

*伪对象*就是指那些在测试中用于伪装成被测试类的“合作者”的对象。下面就是一个例子：在一个POS系统中有一个Sale类（见图3-1），它有一个方法叫scan()，该方法的参数是客户欲购买商品的条形码。当scan()被调用时，Sale对象需要在收银机的显示器上显示出被扫描商品

的名称及价格。

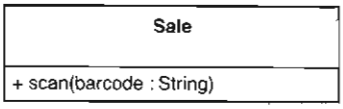


图3-1　Sale

现在，假设我们想测试一下Sale对象能否在显示器上正确显示商品的名称及价格，那么我们该怎么做？如果对收银机显示器API的调用被深深埋在了Sale类的层层逻辑之中，事情可就棘手了。要想感知Sale对收银机显示器所做的事情可能并没那么容易。然而倘若能够在Sale中找到对显示器进行刷新的那部分代码，我们就可以将原先的设计调整为图3-2所示的这样。

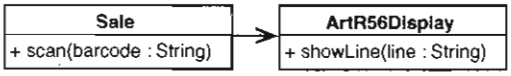


图3-2　Sale与一个显示类进行通信

这里引入了一个新的类：ArtR56Display。该类中包含了所有用于跟我们所使用的收银机的显示器进行交互的必要代码。在使用该类时，只需简单地给出一行想要显示的字符串即可。在把Sale中的所有与显示相关的代码转移到ArtR56Display中之后，就能够得到一个与原先系统从功能上完全一样的新系统。那么，这么做到底给我们带来了什么好处呢？在如此改动了一番之后，我们就可以进一步得到如图3-3所示的设计。

23

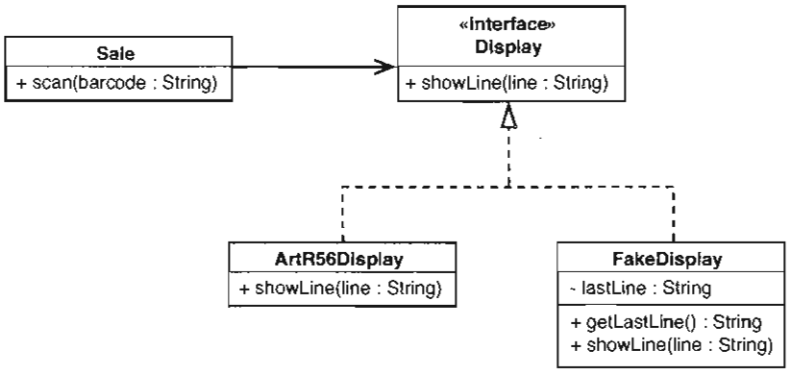


图3-3　Sale与显示器继承体系图

Sale类现在不仅可以持有ArtR56Display，还可以持有其他类的对象（只要它实现了Display接口），如FakeDisplay。拥有一个伪显示器的好处就在于我们可以基于这个伪显示器来编写测试[1]，从而知道Sale对实际的显示器会做些什么。

1. 使用伪显示器来感知Sale对显示器的所作所为。——译者注

那么，这一方案具体实施起来又该怎么做呢？首先，Sale接收一个显示器（display）为参数，这个显示器可以是任何实现了Display接口的类的对象。

```
public interface Display
{
    void showLine(String line);
}
```

在我们的这个例子中，ArtR56Display和FakeDisplay都实现了Display接口。

Sale对象可以通过构造函数的参数来获取显示器的引用，并将其存放在自己的内部成员中：

```
public class Sale
{
    private Display display;

    public Sale(Display display) {
        this.display = display;
    }

    public void scan(String barcode) {
        ...
        String itemLine = item.name()
                + " " + item.price().asDisplayText();
        display.showLine(itemLine);
        ...
    }
}
```

24

而scan方法内的代码则调用了刚才接收的显示器的Display接口上的showLine方法。然而这个调用背后实际发生了什么则取决于我们在创建该Sale对象时交给它的是哪种显示器。如果我们给的是ArtR56Display，则这个调用将会试图向实际的收银机屏幕上显示东西。而倘若我们给的是FakeDisplay，取而代之的是我们能够看到（或感知到）Sale想要显示哪些东西。下面就是一个相关测试：

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        FakeDisplay display = new FakeDisplay();
        Sale sale = new Sale(display);

        sale.scan("1");
        assertEquals("Milk $3.99", display.getLastLine());
    }
}
```

FakeDisplay类看上去有点古怪：

```
public class FakeDisplay implements Display
{
    private String lastLine = "";

    public void showLine(String line) {
```

```
        lastLine = line;
    }
    public String getLastLine() {
        return lastLine;
    }
}
```

showLine方法接收一行文本并将它赋给lastLine变量。而getLastLine方法则返回原先保存下来的文本[1]。这个特性本身看上去微不足道，但对我们的帮助很大。通过上面写的那个TestCase，就可以知道Sale是否会将正确的文本送到显示器上。

25

伪对象支持真正的测试

有时候人们看到伪对象的使用会说："那并不算真正的测试"。毕竟，它不能告诉我们到底屏幕上显示了什么[2]。假设收银机显示器上运行的软件[3]的某个地方出了问题，则上面的测试永远也不会反映出这个问题。话虽没错，但这并不代表该测试就不是真正的测试。就算可以构造出一个能够精确地告诉我们到底哪些像素会被显示在一个货真价实的收银机显示器上的测试来，也并不意味着Sale所在的程序对所有型号的收银机显示器都能正常工作，同样也并不表示它算不上一个真正的测试。在编写测试时，我们得采取分而治之（逐个击破）的方案。比如说，刚才提到的基于FakeDisplay的测试就告诉了我们Sale对象是如何影响显示器的，那么它就只负责这一方面的任务。但这并不代表它负责的任务毫无价值。例如假设某一天我们在系统中发现了一个bug[4]，那么通过运行该测试我们就能够得知问题是不是出在Sale身上。进一步说，我们若能利用这一信息来辅助进行错误定位，就可以节省下大量的时间。

遵循为独立单元编写测试的理念[5]，我们最终写出来的就会是短小而易于理解的一个个测试。这会使得我们的代码变得更易理解。

3.1.2　伪对象的两面性

第一次碰到伪对象这个概念的时候你可能会被搞糊涂。它们最怪异之处就在于，从某种意义上来说，它们有两张"脸"。让我们再来回顾一下FakeDisplay，见图3-4。

一方面，FakeDisplay必须具有一个showLine方法，因为它实现了Display接口，而showLine是该接口上的唯一一个函数，也是Sale能够"看到的"唯一一个函数。另一方面，getLastLine这个方法则是为测试准备的，这也正是我们将变量display声明为一个FakeDisplay而非Display的原因[6]。

1. 即最近一次被显示的文本。——译者注
2. 因为FakeDisplay并不是真正的Display。——译者注
3. 如驱动程序。——译者注
4. 如收银机上不能正确显示商品名称和价格了。——译者注
5. 即上段提到的分而治之理念。——译者注
6. 因为若是将它声明为Display，则测试者便无法通过display来调用getLastLine方法了，至少要经过一次向下转型（downcast）。——译者注

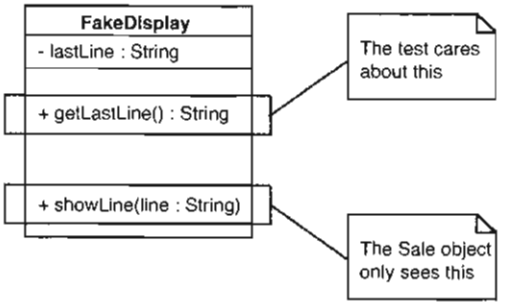


图3-4 伪对象的两面性

26

```
import junit.framework.*;

public class SaleTest extends TestCase
{
    public void testDisplayAnItem() {
        FakeDisplay display = new FakeDisplay();
        Sale sale = new Sale(display);

        sale.scan("1");
        assertEquals("Milk $3.99", display.getLastLine());
    }
}
```

Sale类会把FakeDisplay对象看作一个Display，而站在测试者的角度，则必须知道它的本来面目。否则就无法调用它上面的getLastLine()方法来感知Sale对象到底试图显示什么。

3.1.3 伪对象手法的核心理念

本节展示的伪对象手法十分简单，但已经说明了其背后的核心理念。伪对象的实现途径多种多样。比如，在面向对象的语言中，我们通常可以使用像上例中的FakeDisplay这样的简单类来实现。而在非面向对象语言中，则可以通过定义一个替代函数来达到伪装的目的，该替代函数将某些值记录在全局数据结构中，从而我们可以在测试中访问这些值。具体细节参见第19章。

3.1.4 仿对象

伪对象不仅写起来容易，还是有力的感知工具。但如果你要编写许许多多的伪对象的话，建议你考虑一种更高级的伪对象，即所谓的仿对象[1]（mock object）。简单讲仿对象就是在内部进行断言检查的伪对象。下面就是一个使用仿对象的测试：

```
import junit.framework.*;

public class SaleTest extends TestCase
```

1. 伪对象跟仿对象的语义有着极细微的差别。前者是“伪装”成目标对象，目的是为了欺骗被测试者，让它以为使用的是真正的目标对象。而后者虽说也是“假”货，然而其目的却在于尽量模仿真实的目标对象的行为，被测试者可以（从行为上）把它看作一个真正的目标对象来使用。——译者注

```
{
    public void testDisplayAnItem() {
        MockDisplay display = new MockDisplay();
        display.setExpectation("showLine", "Milk $3.99");
        Sale sale = new Sale(display);
        sale.scan("1");
        display.verify();
    }
}
```

27

在上面的测试中，我们创建了一个仿显示器对象。仿对象的妙处就在于我们可以告诉它们哪些调用是被期望发生的，并告诉它们去检查是否确实接收到了这些调用。上面的示例代码也正显示了这一点：我们告诉display所指的仿显示器对象说，“我们期望被测试者会调用你的showLine方法，并以"Milk $3.99"为参数”。在设置了这一期望之后，我们接着便使用了被测的目标对象，即sale。本例中我们调用的是sale的scan()方法。之后我们再调用display的verify()方法，该方法负责检查确认是否前面设置的期望都被满足了。如果没有的话它就会令测试失败[1]。

仿对象是一个强大的工具，现有的仿对象框架已经有很多很多了。然而，并非所有语言环境下都有可用的仿对象框架，而简单的伪对象在大部分情况下就已经够用了。

28

1. 通常是抛出异常。——译者注

第4章 接缝模型

几乎每个尝试过为既有代码编写测试的人都会发现既有代码对测试是多么的不友好。这并非一种局限于特定程序或语言的现象，而是普遍存在的。一般而言编程语言对测试的支持不太好。要想最终得到易于测试的程序似乎只有两条路：一条路是边开发边编写测试，另一条路则是在先期花点时间试着将易测试性纳入整体的设计考量。人们对第一种方案的期望甚高，不过若是以业界的大量代码为证的话，第二种方法在过去并没有取得多大的成功。

在一次次试图将代码纳入测试中去的过程中，我发觉自己逐渐开始用一种相当独特的方式来思考代码。我完全可以把这归为个人的怪癖而不去管它，但是后来发现这样一种看待代码的方式在我遇到新的不熟悉的编程语言时给了我帮助。因为在本书中我无法涵盖所有的编程语言，所以我决定把这种思考代码的方式简单介绍一下，希望也能对你有所帮助。

4.1 一大段文本

由于我开始接触编程比较晚，所以幸运地拥有了自己的电脑，以及运行于其上的编译器，而我的许多朋友则早在纸带打孔的年代就开始编程了。我在学校里学习编程后，得到了实验室里的一台终端作为工作平台，而代码的编译则是在一台DEC VAX机器上远程进行的。当时那里还配有一台账目记录机，每次请求编译都会在我们的账户上扣掉一点钱，而且每学期的上机时间也是有限的。

那时候，程序对我来说就是一张清单。每过一两个小时我都要从实验室走到打印室，将程序打印出来并仔细检查，试图找出哪些地方对了哪些地方错了。当时我知道的还不够多，不足以让我对模块产生多少关注。但我们又必须得编写模块化的代码以表明我们能够做到这一点，而实际上那时我更为关心的却是代码是否最终会给出正确的结果。当我接触到面向对象的概念之后，模块性就显得有些学院派了。在完成学校布置的作业时我并没有打算使用面向对象的概念。再后来我投身工业界，便开始大量关注起这些概念。但是在学校里面，程序对我来说就是一张张清单：我得写一长串的函数，并一个个地理解。

这种将程序看作清单的视角似乎是正确的，至少我们看一看人们在编写程序时的行为就知道了。假设我们对编程一无所知并看到一间屋子里面全是程序员，我们可能会认为他们是一群正在检查和修改庞大且重要文档的专家。程序看起来可能像是一段长文本，牵一发而动全身，哪怕改

动其中很小的一个地方都可能会导致整个文档发生改变，所以人们得小心改动以免带来错误。

看起来这些都没错，但模块性呢？我们经常说最好将程序编写成由一个个小的可复用组件构成的整体，但问题是这些组件被单独复用的情况到底有多少呢？答案是，并不算很多。复用可不是件好差事。就算一个软件的各个部分看起来是独立的，它们也会常常以一些微妙的、不易觉察的方式互相依赖着。

4.2　接缝

当你开始试图将单个的类从系统中剥离出来以便能够将它纳入单元测试中去时，你会发现你得常常先进行一系列的解依赖工作。有趣的是，不管系统原本的设计有多么良好，你还是要常常做上许多工作。将类从现有项目中剥离出来以便能够对其进行测试，凡做过这个工作的人对“什么是好的设计”这个问题都会有一番与原先不同的理解，并开始以一种全新的角度去审视软件。这时简单地将程序看作一长串文本的做法就不能再帮上什么忙了。那么，究竟该如何看待软件系统呢？让我们先来看如下一个C++函数的例子：

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();

    return true;
}
```

30

这看上去的确跟一段文本没什么区别，对不对？现在假设，我们想要运行除了以下这行之外的所有代码：

```
PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
```

如何才能做到这一点呢？

很简单对不对？我们只需将上面这行代码从这个函数中删掉即可。

接下来让我们给问题再加点限制。我们想要避免执行这行代码，因为PostReceiveError是一个会跟另一个子系统进行交互的全局函数，如果在测试中牵扯到那个子系统的话就太麻烦了。这样一来问题就变为：如何才能做到在测试中既执行了这个方法又避免了调用到PostReceiveError呢？这里的关键在于既要在测试中避免调用到PostReceiveError又不能让最终的产品代码调用不到它[1]。

对我来说这个问题有多个解，这就引入了接缝这一概念。

我们先看一下接缝的定义，然后再看一些例子。

接缝

接缝（seam），顾名思义，就是指程序中的一些特殊的点，在这些点上你无需作任何修改就可以达到改动程序行为的目的。

那么，回过头来考虑前面给出的示例代码，在PostReceiveError的调用点存在接缝吗？答案是肯定的。我们可以有多种方式来修改该处的行为。以下是最直接的一种：PostReceiveError是一个全局函数，并非CAsyncSslRec类的一部分。所以，如果我们往CasyncSslRec类中添加一个签名一样的PostReceiveError（不同之处在于后者是成员函数），会导致什么结果？

```
class CAsyncSslRec
{
    ...
    virtual void PostReceiveError(UINT type, UINT errorcode);
    ...
};
```

31

其实现如下：

```
void CAsyncSslRec::PostReceiveError(UINT type, UINT errorcode)
{
    ::PostReceiveError(type, errorcode);
}
```

以上这一更动不会改变程序行为。CAsyncSslRec::PostReceiveError只是简单地将任务转发给了全局的::PostReceiveError来完成。这里虽说引入了一个小小的间接层，但最终调用到的还是同样的函数，即全局的::PostReceiveError。

好的，现在如果我们把CAsyncSslRec给子类化（subclass）了，并覆盖PostReceiveError，将会怎么样：

```
class TestingAsyncSslRec : public CAsyncSslRec
```

1. 也就是说，简单地将这行代码从所在函数中删除掉的做法就不再可行了。为什么呢？因为产品代码还需要这行代码。总不能在测试时把它删掉，而在产品编译构建时再添加进去吧？一行两行代码这样做还可以，然而如果有大块、多处这种情况（通常如此），这种方法根本就是不切实际的了。——译者注

```
{
    virtual void PostReceiveError(UINT type, UINT errorcode)
    {
    }
};
```

如果像上面这样做了，并回到测试代码中创建CAsyncSslRec对象的地方，改为创建TestingAsyncSslRec，就能够有效地将下面代码中的调用PostReceiveError的行为屏蔽掉：

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();

    return true;
}
```

32

现在我们在为以上代码编写测试的时候就可以避开那些肮脏的副作用了[1]。

上面讨论的这类接缝我把它们称之为对象接缝（object seam）。遇到此类接缝时，我们可以在无需修改调用函数[2]的情况下改变被调用的函数[3]。对象接缝存在于面向对象语言中，它们只不过是许许多多不同种类的接缝中的一种。

那么，为什么要关心接缝这个概念？它有什么好处呢？

在为遗留代码编写测试时，一个最大的挑战就是解依赖。运气好的话这些依赖可能较小、较局部化，然而碰到极端情况时，你可能得对付大量的、在代码基中分布得到处都是的依赖。而接

1. 本例中特指PostReceiveError的副作用。——译者注
2. 本例中是CAsyncSslRec::Init。——译者注
3. 即选择哪个函数被调用。——译者注

缝的概念则帮助我们去发现代码基中既有的可利因素。倘若能够将接缝处的行为取代掉，我们就等于有选择性地排除了某些依赖。我们还可以将被依赖方替换为其他代码，以此来感知被测试代码对被依赖方的要求或影响，并针对这些要求或影响来编写测试。往往这么做了以后我们就能够获得足够的测试以便采取更进一步的举动。

4.3 接缝类型

对于不同的编程语言，可用的接缝类型也不同。考察它们的最佳途径是观察该语言的程序代码被转换至机器代码的过程的各个阶段，其中每个显著的阶段都蕴涵着不同种类的接缝。

4.3.1 预处理期接缝

对于大多数编程环境来说，都是由编译器读进代码然后生成目标代码或字节码。取决于编程语言的不同，有可能后面还会添上一个或多个处理步骤。然而在编译器编译代码之前，有没有什么更早期的步骤？

事实上，只有寥寥几门语言在编译前会有另一个处理阶段（预处理），C与C++就是这其中最常见的代表。

在C和C++中，程序代码被编译之前会先由宏预处理器进行预处理。这么多年来宏预处理饱受着人们的诅咒和讥讽。借助于它我们可以编写出外表看起来无可厚非的代码：

```
TEST(getBalance,Account)
{
    Account account;
    LONGS_EQUAL(0, account.getBalance());
}
```

而实际上到了编译器眼里却成了这样：

```
class AccountgetBalanceTest : public Test
    { public: AccountgetBalanceTest () : Test ("getBalance" "Test") {}
          void run (TestResult& result_); }
   AccountgetBalanceInstance;
     void AccountgetBalanceTest::run (TestResult& result_)
{
     Account account;
{ result_.countCheck();
  long actualTemp = (account.getBalance());
  long expectedTemp = (0);
  if ((expectedTemp) != (actualTemp))
{ result_.addFailure (Failure (name_, "c:\\seamexample.cpp", 24,
StringFrom(expectedTemp),
StringFrom(actualTemp))); return; } }

}
```

我们也可以像下面这样用条件编译语句将代码包围起来，从而达到支持调试模式以及不同平台的目的：

```
...
m_pRtg->Adj(2.0);

#ifdef DEBUG
#ifndef WINDOWS
    { FILE *fp = fopen(TGLOGNAME,"w");
    if (fp) { fprintf(fp,"%s", m_pRtg->pszState); fclose(fp); }}
#endif

m_pTSRTable->p_nFlush |= GF_PLOT;
#endif


...
```

在产品代码中过度使用预处理并不是个好主意，因为它会降低代码的清晰性。条件编译指令（如#ifdef、#ifndef和#if等）几乎等于是在强迫你在同一份源代码中维护多个不同的程序。宏（使用#define来定义）若是使用得当固然可以被用来做一些很好的事情，但是别忘了它们的运作机制只不过是文本替换，因此很容易就可以制造出隐藏了极度隐晦的bug的宏。

姑且先把这些考虑搁在一边，我对C和C++中具有预处理功能还是感到高兴的，因为它给程序中带来了更多的接缝。下面就是一个具体的例子。这是一个C程序，其中的问题在于对db_update这个库函数的依赖。db_update会直接跟数据库进行沟通。因此，除非能用另一个实现来将db_update原先的实现给替换掉，否则就无法感知该函数的行为：

```
#include <DFHLItem.h>
#include <DHLSRecord.h>
extern int db_update(int, struct DFHLItem *);

void account_update(
    int account_no, struct DHLSRecord *record, int activated)
{
    if (activated) {
        if (record->dateStamped && record->quantity > MAX_ITEMS) {
            db_update(account_no, record->item);
        } else {
            db_update(account_no, record->backup_item);
        }
    }
    db_update(MASTER_ACCOUNT, record->item);
}
```

34

我们可以使用预处理接缝来替换掉程序中对db_update的调用。为此，我们可以引入一个头文件，称作localdefs.h：

```
#include <DFHLItem.h>
#include <DHLSRecord.h>

extern int db_update(int, struct DFHLItem *);

#include "localdefs.h"
```

```
void account_update(
    int account_no, struct DHLSRecord *record, int activated)
{
    if (activated) {
        if (record->dateStamped && record->quantity > MAX_ITEMS) {
            db_update(account_no, record->item);
        } else {
            db_update(account_no, record->backup_item);
        }
    }
    db_update(MASTER_ACCOUNT, record->item);
}
```

在该头文件中我们可以提供另一个db_update的定义，以及一些有用的变量：

```
#ifdef TESTING
...
struct DFHLItem *last_item = NULL;
int last_account_no = -1;

#define db_update(account_no,item)\
    {last_item = (item); last_account_no = (account_no);}
...
#endif
```

35

有了这个实现来替换db_update原先的实现，我们就可以编写测试来验证db_update被调用的时候接收到的是否是正确的参数了。我们之所以能够这么做是全靠C预处理指令#include提供的一个接缝，利用这个接缝我们得以在（代码）文本被编译之前将它替换掉。

预处理期接缝是个非常强大的手段。我不认为我会真的希望像Java以及其他更现代的语言中有这个功能，然而对于C和C++来说则不同，因为它起到了弥补这两门语言中的一些其他测试障碍的作用，所以还是有好处的。

关于接缝，还有一个重要的地方，即，每个接缝都有一个所谓的*激活点*（enabling point）。为此我们先来回顾一下接缝的定义：

接缝

接缝（seam），顾名思义，就是指程序中的一些特殊的点，在这些点上你无需作任何修改就可以达到改动程序行为的目的。

当遇到一个接缝时，即意味着我们可以改变其所在处的行为。当然，我们不能仅仅为了测试就真的去修改其所在之处的代码，源代码在产品阶段和测试阶段应当是完全一样的。在前面的例子中我们曾尝试改变db_update调用点处的行为，为了利用起该处的接缝，我们得在其他某个地方进行改动。而对于db_update这个例子来说，其激活点就是TESTING这个预处理符号的定义。当定义了TESTING时，localdefs.h文件中就会包含一些宏[1]，而这些宏将会把源文件中对db_update的调用替换（展开）为localdefs.h中的db_update定义。

1. 其中最主要的就是db_update宏。——译者注

激活点

每个接缝都有一个激活点，在这些点上你可以决定使用哪种行为。

4.3.2　连接期接缝

在许多语言系统中，编译并非构建过程的最后一步。编译器产生的只不过是代码的中间表示，其中包含了对其他文件中的代码的调用。然后，连接器负责将这些中间表示[1]连接起来。连接器会对每个调用进行决议以便最终能得到一个可运行的完整程序。

对于C和C++这类语言来说的确存在着一个单独的连接器，其作用如上所述。而在Java以及类似的语言中则是编译器在幕后负责进行连接过程。当一个Java源文件包含了import语句时，编译器就会检查import的类是否已被编译，如果没有，就先对其编译，然后再检查它的所有调
36 用是否都能够在运行期正确决议。

不过，不管你的语言的编译系统是用哪种方式来进行符号引用决议的，你一般都可以利用它来替换一个程序的某些部分。以Java为例，下面是一个名为FitFilter的类：

```
package fitnesse;

import fit.Parse;
import fit.Fixture;

import java.i0.*;
import java.util.Date;


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

public class FitFilter {

    public String input;
    public Parse tables;
    public Fixture fixture = new Fixture();
    public PrintWriter output;

    public static void main (String argv[]) {
        new FitFilter().run(argv);
    }

    public void run (String argv[]) {
        args(argv);
        process();
        exit();
    }

    public void process() {
        try {
```

1. 在C和C++中即目标文件。——译者注

```
            tables = new Parse(input);
            fixture.doTables(tables);
        } catch (Exception e) {
            exception(e);
        }
        tables.print(output);
    }
    ...
}
```

该文件import了fit.Parse和fit.Fixture。那么，编译器和JVM究竟是如何找到这些类的呢？在Java中你可以使用一个名为classpath的环境变量来告诉Java系统到哪里去寻找这些类。因此，你可以创建同名类，将它们置于另一个目录中，并令classpath指向该目录，从而诱使编译器去发现从而连接到你写的另一个版本的fit.Parse和fit.Fixture来。虽说这种技巧用在产品代码中可能会带来混乱，但用在测试中却是个相当实用的解依赖手段。 37

在上例中，假设我们想要为Parse类提供另一个版本的实现以便用于测试。我们想要的接缝在哪里？

答案是process方法中的new Parse(input)处。

激活点又在哪里？

答案是classpath。

许多语言中都有这种动态连接功能。对于其中大部分来说，我们都有办法来利用其连接期接缝。但并非所有的连接都是动态连接。对于许多较为古老的语言来说，几乎所有的连接都是跟在编译之后静态连接。

C和C++的许多编译构建系统都使用静态连接来产生可执行文件。通常来说利用这种连接的接缝的最简单的途径是为你想要替换其实现的所有类和函数创建另一个单独的库文件，然后，当你进行测试时，就可以通过修改构建脚本文件[1]来引导你的构建系统去连接到供测试之用的库文件了，而另一方面，在产品阶段的构建中，则可以修改构建脚本使得程序连接到产品代码的库。这种做法可能要花点工夫，但倘若你的代码基中到处散布着对某个第三方库的调用的话，这样做就是值得的了。例如，设想这么一个CAD应用，其代码里面嵌有很多对某个图形库的调用。下面就是其中一种典型的代码：

```
void CrossPlaneFigure::rerender()
{
    // draw the label
    drawText(m_nX, m_nY, m_pchLabel, getClipLen());
    drawLine(m_nX, m_nY, m_nX + getClipLen(), m_nY);
    drawLine(m_nX, m_nY, m_nX, m_nY + getDropLen());
    if (!m_bShadowBox) {
        drawLine(m_nX + getClipLen(), m_nY,
                 m_nX + getClipLen(), m_nY + getDropLen());
```

1. 如makefile之类。——译者注

```
        drawLine(m_nX, m_nY + getDropLen(),
                    m_nX + getClipLen(), m_nY + getDropLen());
    }

    // draw the figure
    for (int n = 0; n < edges.size(); n++) {
        ...
    }

    ...
}
```

38 以上代码中对一个图形库进行了许多直接调用。遗憾的是，想要验证这些代码是否做了你想要它们做的事情，唯一的途径就是运行它们并观察计算机屏幕上画出来的图形。然而当这种做法遇到复杂的代码时便会非常容易出错。一个替代方案就是使用连接期接缝。具体来说，如果其中所有的画图函数都是来自某个特定的库，那么你就可以创建该库的一个stub[1]，用它来替代原先的库去连接到这个应用程序。如果你只想解开依赖的话，这个stub库里面就可以全放上相应的空函数，如下：

```
void drawText(int x, int y, char *text, int textLength)
{
}

void drawLine(int firstX, int firstY, int secondX, int secondY)
{
}
```

遇到带有返回值的函数，你则需要在相应的stub函数里面也返回某些东西。通常你可以选择返回一个代表成功的值或者返回某个类型的默认值，例如：

```
int getStatus()
{
    return FLAG_OKAY;
}
```

这里举图形（graphic）库的例子有点儿不那么典型。但为什么又说它是适合使用该技术的一个很好的例子呢？原因之一就是它几乎是个纯粹的命令型接口。换句话说，一般是你去调用其库函数来命令它们做某些事情，而并不请求它们反馈什么信息。遇到后一种情况会比较难对付，因为当试图测试你的代码时，为这种函数编写的stub版本一般来说不能简单地返回默认值。

使用连接期接缝的目的之一是分离。当然你也可以实现感知，只不过后者需要多花点工夫。例如，在刚才举的那个假想的图形库的例子中，可以引入某种额外的数据结构来记录对这些库函数的调用：

```
std::queue<GraphicsAction>  actions;
```

1. stub在程序员口中一般是指“一小段（可能是由某种工具自动生成的）代码（可能是二进制的），用来占取某个位置，以达到某个特定目的（如转发，或这里的行为消除等）。”也有地方译为“桩子”，本书中选择保留不译。——译者注

```
void drawLine(int firstX, int firstY, int secondX, int secondY)
{
    actions.push_back(GraphicsAction(LINE_DRAW,
            firstX, firstY, secondX, secondY);
}
```

借助于这些数据结构，就可以在测试中感知某个函数的作用了：

```
TEST(simpleRender,Figure)
{
    std::string text = "simple";
    Figure figure(text, 0, 0);

    figure.rerender();
    LONGS_EQUAL(5, actions.size());
    GraphicsAction action;
    action = actions.pop_front();
    LONGS_EQUAL(LABEL_DRAW, action.type);

    action = actions.pop_front();
    LONGS_EQUAL(0, action.firstX);
    LONGS_EQUAL(0, action.firstY);
    LONGS_EQUAL(text.size(), action.secondX);
}
```

39

虽说可用来进行感知的方案一般都会逐渐变得复杂，但一开始最好还是选一个比较简单的方案，并尽量控制它的复杂度只在绝对需要的情况下进行必要的增长。

连接期接缝的激活点从来都是位于程序代码之外的，比如有时是在构建或部署脚本中。这就使得连接期接缝的使用显得不那么醒目。

使用小提示

使用连接期接缝时，请确保测试和产品环境之间的差别是显而易见的。

4.3.3 对象接缝

对象接缝几乎可算是面向对象语言中最为有用的一种接缝了。对于这种接缝来说，值得注意的根本一点是，面向对象程序中的一处函数调用并没有指出到底哪个方法会被实际上调用起来。以Java为例：

```
cell.Recalculate();
```

上面的代码看上去会有一个名为Recalculate的方法被调用。当该程序运行起来时，得存在这么一个方法，然而实际上，这样的方法不止一个：

40

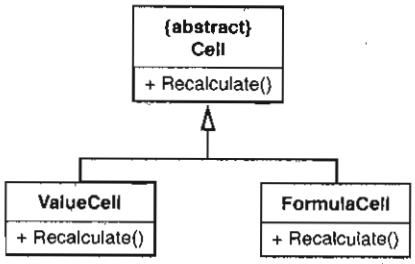


图4-1　Cell继承体系

那么，这里到底会调用哪个方法呢？

```
cell.Recalculate();
```

在不知道cell实际指向哪个对象的情况下，我们无法作出判断。可能是ValueCell类中的Recalculate方法，也可能是FormulaCell的，甚至还可能是某个其他并非继承自Cell的类的Recalculate方法（要真是这样的话，将这个变量起名cell就太失败了！）。如果可以在不用修改这行以及周围代码的情况下就让它调用到另一个Recalculate方法，这行调用就是一个接缝。

在面向对象语言中，并非所有的方法调用都是接缝。下面就是一个反例：

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet() {
        ...
        Cell cell = new FormulaCell(this, "A1", "=A2+A3");
        ...
        cell.Recalculate();
        ...
    }
    ...
}
```

以上代码创建了一个cell，并紧接着在同一个方法中使用了它。那么，其中对Recalculate的调用是一个对象接缝吗？不是。因为它并没有对应的激活点。我们无法控制哪个Recalculate方法被调用，因为这取决于cell所指向对象的实际类型，而这在那行new FormulaCell的地方就已经确定了，除非去修改buildMartSheet方法，否则我们无法改变cell的实际类型。

41　然而，如果代码像下面这样呢？

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        cell.Recalculate();
        ...
    }
    ...
}
```

现在，buildMartSheet中对cell.Recalculate的调用是不是一个接缝呢？是的。我们可以在测试用例中创建一个CustomSpreadsheet，并使用我们愿意的任何种类的Cell对象为参来调用这个buildMartSheet方法。换句话说，我们无需修改cell.Recalculate这行调用所在的方法就可以达到改变这行调用所实际干的事情的目的。

那么，既然这是一个接缝，它的激活点又在哪里呢？

答案是buildMartSheet的参数列表。通过给出不同类型的对象作为其参数，我们可以根据测试需要任意改变buildMartSheet里调用的Recalculate的行为。

到目前为止我们看到的大部分对象接缝都还算是比较简单的。而下面我们就来看一个微妙的对象接缝。在下面的代码中，在Recalculate的调用处存在一个对象接缝吗？

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
        ...
    }

    private static void Recalculate(Cell cell) {
        ...
    }

    ...
}
```

这里的Recalculate方法是个静态方法。那么，buildMartSheet中对Recalculate的调用处存在一个接缝吗？答案是肯定的。我们无需修改buildMartSheet就可以改变该调用处的行为。具体做法是删掉Recalculate方法定义前的static关键字，并将其访问权限从私有改为受保护的，这么一来我们就可以在测试中对CustomSpreadsheet进行子类化并重写其Recalculate了，如下：

```
public class CustomSpreadsheet extends Spreadsheet
{
    public Spreadsheet buildMartSheet(Cell cell) {
        ...
        Recalculate(cell);
        ...
    }

    protected void Recalculate(Cell cell) {
        ...
    }

    ...
}

public class TestingCustomSpreadsheet extends CustomSpreadsheet {
    protected void Recalculate(Cell cell) {
```

```
        ...
    }
}
```

以上做法看起来是不是显得有点过于迂回了？如果我们不喜欢某处依赖，干嘛不直接进到代码中去修改一通把它改掉呢？没错，有时候这的确可行，然而当你为那些特别脏乱的遗留代码安放测试时，往往最好的途径是尽量少去修改其代码。如果知道你的语言所支持的接缝类型，并知道怎样去使用它们，则通常可以更安全地将测试安置妥当。

到目前为止我们所展示的接缝类型都是一些主要的。你可以在许多编程语言中见到它们的身影。现在就让我们来回顾一下本章开头给出的例子，看看里面能看到哪些接缝：

```
bool CAsyncSslRec::Init()
{
    if (m_bSslInitialized) {
        return true;
    }
    m_smutex.Unlock();
    m_nSslRefCount++;

    m_bSslInitialized = true;

    FreeLibrary(m_hSslDll1);
    m_hSslDll1=0;
    FreeLibrary(m_hSslDll2);
    m_hSslDll2=0;

    if (!m_bFailureSent) {
        m_bFailureSent=TRUE;
        PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
    }

    CreateLibrary(m_hSslDll1,"syncesel1.dll");
    CreateLibrary(m_hSslDll2,"syncesel2.dll");

    m_hSslDll1->Init();
    m_hSslDll2->Init();
    return true;
}
```

43

在对PostReceiveError的调用处存在哪些接缝呢？如下所示：

(1) PostReceiveError是个全局函数，因此很容易就可以使用连接期接缝。具体做法可以是创建一个库，其中包含PostReceiveError的一个stub版本（一般是一个空函数），并将原先的程序连接到这个库，从而消除那个调用处的行为。这一接缝的激活点应该是项目的makefile文件，或者IDE里的某些设置（如果你用了IDE的话）。需要更改项目的构建设置，以便在测试的时候连接到测试用库，而在构建真正系统的时候连接到产品库。

(2) 可以在代码中添加一个#include语句，并定义一个名为PostReceiveError的宏，当我们进行测试时就激活这个宏。也就是说这里有一个预处理期接缝。那么，其激活点又在哪呢？答案是可以使用一个预处理宏定义来控制这个PostReceiveError宏是否被定义。

(3) 还可以定义一个名为PostReceiveError的虚函数，就像本章的开始所做的那样。也就是说这里还存在一个对象接缝。激活点则在对象的创建处。可以创建一个CAsyncSslRec对象，也可以创建CAsyncSslRec的某个为测试而写的并重写了PostReceiveError的子类化版本。

很令人惊讶，居然有这么多途径能达到这一目的，不用修改下面这个方法，就可以替换掉PostReceiveError调用处的行为：

```
bool CAsyncSslRec::Init()
{
   ...
   if (!m_bFailureSent) {
      m_bFailureSent=TRUE;
      PostReceiveError(SOCKETCALLBACK, SSL_FAILURE);
   }
   ...

   return true;
}
```

当想要将某段代码置入测试中时，选择正确类型的接缝很重要。一般来说，如果你用的是面向对象语言，则对象接缝是最佳选择。预处理期接缝以及连接期接缝某些时候是有用的，但它们没有对象接缝那么清楚明显。此外依赖于这两种接缝的测试可能会难以维护。我个人倾向于将预处理期接缝和连接期接缝这两种接缝保留到代码中到处浸透着依赖并且没有其他更好的方案可选的情况下。

当习惯了以“接缝之眼”来看待代码时，就能更容易看出如何测试某段代码以及如何组织新代码的结构以令其更具“测试友好性”。

44

第5章 工 具

在对付遗留代码时，你需要哪些工具呢？首先需要一个编辑器（或IDE）以及一个构建系统，同时还需要一个测试框架。要是再有一个对应于你使用的语言的重构工具那就更好了，它们也可能会带来很大的帮助。

本章会介绍一些现有的工具，以及它们在你对付遗留代码的过程中能够扮演的角色。

5.1 自动化重构工具

手动重构当然也没什么不可以，但若是有个工具能够帮你完成一些重构工作岂不更好？这样你就能够节省许多时间了。20世纪90年代，Bill Opdyke为写一篇以重构为主题的论文而开始做一个C++重构工具。尽管这一工具最终并没有商业化，但据我所知，他的工作提供了其他语言领域的许多成果。其中最为著名的就要数Smalltalk的重构浏览器了，这一重构浏览器由伊利诺斯大学的John Brant和Don Robert开发，支持众多重构手法，并且很长时间以来都是自动化重构技术的经典范例。自那时起，许多人开始尝试往各种被广泛使用的语言中加入重构支持。在本书写作时已经出现了许多Java重构工具，其中大多数都被集成到了IDE中。Delphi同样也有重构工具，而C++也有一些相对较新的重构工具。在写作本书时一些C#的重构工具则正处于积极开发过程中。

有了这些工具，重构似乎看上去简单许多了。就某些环境下来说，的确如此。只不过，这些工具对于重构的支持良莠不齐。让我们再来回顾一下重构的概念。下面是Martin Fowler对重构下的定义（《重构：改善既有代码的设计》[1]）：

> **重构**
>
> 名词。对软件内部结构的一种调整，目的是在不改变软件的外在行为的前提下，提高其可理解性，降低其修改成本。

只有当你的修改不会改变行为时，才能算是重构。重构工具应当检验某处修改是否改变了行为，而许多重构工具的确做到了这点。这在Smalltalk的重构浏览器、前面提到的Bill Opdyke的工作以及许多早期的Java重构工具中都是一个基本规则。不过实际上对于一些极少数情况，有些工具却并不进行检查，那么你在重构的过程中很可能将一些细微的bug带进去了。

1. 此书英文注释版即将由人民邮电出版社出版。——编者注

谨慎选择重构工具是有好处的。要了解工具开发商们对他们的工具的安全性怎么说，并要自己进行一些测试。我自己在遇到一个新的重构工具时，通常会对它做一点健全性检查。比如，当你试图提取一个方法并将其命名为与它所在类中的某个既有方法同名的方法时，该重构工具会不会将这个标示为一个错误？而倘若是跟该类的基类中的一个方法同名，工具又能否检测出来？如果不能，你很可能在无意间错误地重写了基类的某个方法，从而破坏了现有代码。

本书讨论了在有和没有自动化重构支持的情况下分别应该怎么做。我会在示例中说明是否假定你使用了自动化重构工具。

任何情况下我都一律假定你的工具所提供的重构会保留行为。如果你发现你的工具支持的某些重构不能保留行为，那就别用自动的重构。遵循在没有重构工具情况下的建议，这样更安全。

测试与自动化重构

如果有一个工具，它能够替你完成重构工作，那么我们会倾向于认为无需为待重构的代码编写测试。某些情况下的确如此。如果你的工具能够进行安全的重构，并且你是从一次自动化重构到另一次自动化重构，其间不进行任何其他（人为）编辑修改的话，你就可以认为这些改动是不会改变行为的。然而，事情并非总是如此，下面就是一个例子：

```
public class A {
    private int alpha = 0;
    private int getValue() {
        alpha++;
        return 12;
    }
    public void doSomething() {
        int v = getValue();
        int total = 0;
        for (int n = 0; n < 10; n++) {
            total += v;
        }
    }
}
```

46

在至少两款Java重构工具下，我们都可以使用一次重构来消除doSomething当中的变量v。而在重构之后，代码看上去则像这样：

```
public class A {
    private int alpha = 0;
    private int getValue() {
        alpha++;
        return 12;
    }
    public void doSomething() {
        int total = 0;
        for (int n = 0; n < 10; n++) {
```

```
                total += getValue();
            }
        }
    }
```

看到问题了吗？变量虽被移除了，然而现在alpha的值却被递增了十次，而原先则是一次！这一改动明显不能保留行为。

所以说在开始使用自动化重构之前先编写必要的测试还是有好处的。当然你也可以在没有测试保护的情况下进行一些自动化重构，不过这时候你得清楚你的工具会进行和不会进行哪些检查。我在开始使用一个新工具时所做的第一件事就是看看它对提取方法的支持怎么样，如果发现对它的这一能力足够信任，能够在没有测试的情况下使用的话，我就可以使用它的这一功能先将代码重构至一个测试起来容易得多的状态。

5.2　仿对象

在对付遗留代码的过程中需要面对的一个大问题就是依赖。想要单独执行某段代码来看看它干了些什么的话，常常必须得先解开这段代码与其他代码之间的依赖。而这项工作几乎从来都不那么简单。如果将它依赖的代码拿走的话，就得有什么东西来填补这些空档，以便在进行测试的时候提供出一些适当的值（如函数返回值），从而能够彻底地检测这段代码。在面向对象语言的代码中，这个填补物通常就被称为仿对象（mock object）。

47　有好几个免费的仿对象库。其中大部分都可以通过www.mockobjects.com找到。

5.3　单元测试用具

测试工具的历史是漫长而多彩多姿的。在不到一年的时间里我接触了四五个团队，他们花大价钱买了一些昂贵的测试工具，最后却发现根本不值。公平地说，测试的确是个棘手的问题，而且人们常常会被“只需通过程序的GUI或Web界面即可对其进行测试而无需进行任何特别动作”这样的念头所引诱。这固然是可行的，然而其工作量却往往超过了团队中的任何一个成员所能承受的量。此外，用户界面通常也并非编写测试的最佳地点。用户界面往往是不稳定的、易于变化的，而且离你想要测试的功能往往也太遥远了。而且当基于用户界面的测试失败时想要查明失败的原因也会较困难。话虽如此，人们常常还是花相当多的钱在这些工具上面，试图通过它们来完成所有的测试工作。

我所见过的最有效的测试工具是一些免费的工具。首先是xUnit测试框架，xUnit最初由Kent Beck用Smalltalk编写，接着由Kent Beck和Erich Gamma移植到Java上，这是一个小巧而强大的单元测试框架。下面是它的关键特性：

- 它允许程序员使用开发语言来编写测试。
- 所有测试互不干扰独立运行。
- 一组测试可以集合起来成为一个测试套件（suite），根据需要不断运行。

现在xUnit框架已经被移植到了大部分主流语言乃至相当一部分非主流的、冷僻的语言下。

xUnit的设计里面最具革命性的特点便是其简单性和集中性。它使得我们可以投入最少的精力和时间来完成测试的编写。尽管xUnit最初是为单元测试编写的，但你也可以用它来编写较大的测试，因为xUnit其实并不关心你的测试是大是小。只要你的测试可以用你开发用的语言来编写，xUnit就可以运行它。

本书中的大部分例子都是用Java和C++写的。Java中的首选xUnit用具是JUnit，它跟xUnit家族的大部分其他成员看上去几乎没什么不同。在C++中，我则常常使用一个我自己写的名为CppUnitLite的测试用具，它看上去有一些不同之处，我会在本章予以介绍。顺便一提，我用CppUnitLite完全没有看不起CppUnit原作者的意思，事实上我正是那个原作者。而在CppUnit发布之后我发现，如果当初使用一些C惯用法，并且去除对C++语言中的那些不必要特性使用的话，它还可以变得小巧易用得多，且可移植性也能得到大大增强。

48

5.3.1 JUnit

在JUnit中，你通过对一个名为TestCase的类进行子类化来编写测试，例如：

```
import junit.framework.*;

public class FormulaTest extends TestCase {
    public void testEmpty() {
        assertEquals(0, new Formula("").value());
    }

    public void testDigit() {
        assertEquals(1, new Formula("1").value());
    }
}
```

测试类中每个具有void testXXX()这种签名的方法都定义了一个测试，其中XXX是你想要给该测试起的名字。每个测试方法都可以包含代码和断言。例如，在上例中的testEmpty中，代码是我们new了一个Formula对象，并调用其上的value方法。而断言则是检查该value方法返回的值是否等于0。如果是则测试通过，否则测试失败。

简言之，当你运行一个JUnit测试时会发生下列事情：首先是JUnit的测试运行器负责加载像上述那样的测试类（FormulaTest），然后它使用反射机制来寻找其中所有的测试方法。而接下来的工作则有点“不足为外人道”：它为找到的每一个测试方法都创建一个单独的对象。还以刚才的测试类为例，其中有两个测试方法，于是JUnit就会为它们创建两个单独的对象：第一个对象唯一的任务就是运行testEmpty，而第二个对象唯一的任务就是运行testDigit。如果你想知道这两个对象的类型分别是什么，答案是它们都是同一个类型——FormulaTest（即测试类）。每个对象都被配置用来运行FormulaTest上的某一个测试方法。这里的关键就是，每个测试方法的运行都是由完全独立的对象来完成的，它们不可能互相影响。下面就是一个例子：

```
public class EmployeeTest extends TestCase {
    private Employee employee;

    protected void setUp() {
```

```
        employee = new Employee("Fred", 0, 10);
        TDate cardDate = new TDate(10, 10, 2000);
        employee.addTimeCard(new TimeCard(cardDate,40));
    }

    public void testOvertime() {
        TDate newCardDate = new TDate(11, 10, 2000);
        employee.addTimeCard(new TimeCard(newCardDate, 50));
49      assertTrue(employee.hasOvertimeFor(newCardDate));
    }

    public void testNormalPay() {
        assertEquals(400, employee.getPay());
    }
}
```

在上面的EmployeeTest类中有一个特殊的方法setUp。setUp方法是在TestCase中定义的，它会在每个测试对象的测试方法运行之前运行，还允许我们创建一组用于测试的对象。这组对象是在每个测试方法被执行之前以相同方式创建的。在负责运行testNormalPay的那个对象中，setUp中创建的一个Employee对象在testNormalPay方法中被检查是否能够正确计算一张考勤卡情况下的薪水，这张考勤卡是在setUp方法里面添加的。而在负责运行testOvertime的那个对象中，setUp中创建出来的那个Employee对象则在testOvertime中被添加了另一张考勤卡，然后检查第二张考勤卡是否触发了加班条件。每个EmployeeTest对象的setUp方法都会得到调用，使得它们各自拥有独立的一组测试用对象。此外，如果你想要在一个测试方法执行结束之后做一些其他事情，则可以重写另一个方法——tearDown，该方法也是定义在TestCase中，它会在每个对象对应的测试方法执行完毕之后被运行。

人们第一次见到xUnit时或许会感到有点奇怪：为什么测试类要有setUp和tearDown这两个方法呢？为什么不能在构造函数中创建测试所需的对象呢？答案是当然可以，但别忘了测试运行器是怎样对待测试类的。它会创建测试类的一组对象，其中每个对象用于运行测试类中的一个特定的测试方法。这可能是一组数目不菲的对象，但如果它们尚未分配所需测试用对象的话其资源消耗可能并不那么多。通过将某些代码置于setUp方法之中，我们就可以做到只在需要的时候才去创建对象，如此一来便可节省相当一部分资源。此外，通过延迟setUp的执行，我们还可以在检测并汇报setUp期间发生的问题时去运行它。

5.3.2　CppUnitLite

我在进行CppUnit最初的移植工作时曾试图尽量使其接近于JUnit。我想这么一来对于那些见过xUnit架构的人上手就会比较容易，因此这似乎是较好的做法。然而在我着手时立即就遇到了麻烦，由于C++和Java的语言特性的区别，导致有一些东西难于甚至根本无法在C++中干净地实现出来。主要问题在于C++缺少反射机制。在Java中，你可以持有一个对派生类中方法的引用，你可以在运行期查找方法等。而在C++中要实现这些则必须编写代码来预先注册那些你需要在运行期访问的方法。因此CppUnit要比JUnit难于使用和理解一些。你得在测试类上编写你自己的suite
50 函数，这样测试运行器才能够为每个单独的测试函数运行单独的对象，如下：

```
Test *EmployeeTest::suite()
{
    TestSuite *suite = new TestSuite;
    suite.addTest(new TestCaller<EmployeeTest>("testNormalPay",
            testNormalPay));
    suite.addTest(new TestCaller<EmployeeTest>("testOvertime",
            testOvertime));
    return suite;
}
```

毋庸置疑，这种做法可能会相当烦人。当弄妥一个测试方法必须得涉及三处地方（在头文件中声明，在源文件中定义，在suite方法中注册）时，就很难保证人们还能维持编写测试的动力了。当然，这个问题是可以通过一个宏手法来弥补的[1]。不过我决定推翻重来。于是就诞生了CppUnitLite，在CppUnitLite采用的方案中，人们要编写一个测试只需在相应源文件中编写一些代码即可，如下：

```
#include "testharness.h"
#include "employee.h"
#include <memory>

using namespace std;

TEST(testNormalPay,Employee)
{
    auto_ptr<Employee>  employee(new Employee("Fred", 0, 10));
    LONGS_EQUAL(400, employee->getPay());
}
```

上面这段测试使用了一个名为LONGS_EQUAL的宏，该宏会比较两个长整型是否相等。其行为跟JUnit中的assertEquals一样，只不过前者是为长整型量身定做的。

TEST宏在幕后替你完成了许多麻烦事。它首先定义测试类[2]的一个子类，该子类的名字是TEST宏接受到的两个参数（即该测试的名字以及被测试类的名字）的文本连接[3]。接着它创建该子类的一个实例，注意，该子类被配置为能够自动执行大括号内的代码[4]。此外，由于该实例是全局静态分配的，因而在程序加载时它的构造函数就会被调用起来，后者会将这个实例自身添加到一个包含测试对象的静态链表中。于是后面当测试运行器运行时便能通过遍历该链表来运行每个测试。

在编写完这个微型框架之后，我决定不发布它，因为这个宏背后的代码并不是十分清晰明了，而我曾花了许多时间来说服人们编写清晰的代码。我的一个朋友Mike Hill在我们俩认识之前也遇到过一些类似的问题，并写了一个特定于微软开发平台的测试框架，叫做TestKit，其中使用了与

1. 请参考CppUnit的官方文档。——译者注
2. 这里指辅助进行测试的类（名为Test），而非被测试类。——译者注
3. 即通过“##”这个预编译操作符进行的文本连接，将两个名字“粘”成一个。在实际实现中通常后面还要“粘”上一个“Test”。——译者注
4. 实际上这里用的技术很简单，译者建议你去阅读一下CppUnitLite的源代码。——译者注

CppUnitLite同样的方式来解决测试对象的注册问题。受到Mike的鼓舞，于是我开始消除CppUnitLite中使用后期C++特性的数量，然后我发布了这个框架（别小看这些问题，它曾是CppUnit中的一个大问题。几乎每天我都能收到关于这些问题的电子邮件，有些问题是不会使用
51 模板或标准库，有些问题则是手头的编译器在语言特性方面有这样或那样的不支持）。

CppUnit和CppUnitLite都有作为测试用具的资格。考虑到使用CppUnitLite编写的测试要简洁一些，因此书中出现的C++示例皆使用它。

5.3.3 NUnit

NUnit是.NET语言的一个测试框架。你可以为C#、VB.NET或其他运行于.NET平台之上的语言的代码编写测试。NUnit在操作上跟JUnit很接近。一个显著的区别就是它使用特性（attribute）来标识测试方法和测试类。特性的语法取决于编写测试所用的.NET语言。

下面就是一个使用VB.NET写的NUnit测试：

```
Imports NUnit.Framework

<TestFixture()> Public Class LogOnTest
    Inherits Assertion

    <Test()> Public Sub TestRunValid()
        Dim display As New MockDisplay()
        Dim reader As New MockATMReader()
        Dim logon As New LogOn(display, reader)
        logon.Run()
        AssertEquals("Please Enter Card", display.LastDisplayedText)
        AssertEquals("MainMenu",logon.GetNextTransaction().GetType.Name)
    End Sub

End Class
```

<Test()>和<TestFixture()>这两个特性分别标识出了LogonTest作为一个测试类以及TestRunValid作为一个测试方法。

5.3.4 其他 xUnit 框架

xUnit还被移植到了其他许多不同的语言和平台上。一般来说它们都支持单元测试的指定、分组及运行。如果你想看看你所用的平台或语言有没有相应的xUnit，请访问www.xprogramming.com，
52 在下载区查找。这个网站是Ron Jeffries的，它是所有xUnit家族成员的准仓库。

5.4 一般测试用具

前面描述的xUnit框架是为单元测试设计的。它们固然也可以被用来一趟测试多个类，但这种工作更应该属于FIT和Fitnesse的范畴。

5.4.1 集成测试框架

FIT是一个简练而优雅的集成测试框架，由Ward Cunningham开发。其背后的理念是简单而强大的。如果你可以为你的系统编写文档，并在其中嵌入描述了系统的输入和输出的表格，并且这些文档可以被保存为HTML的话，FIT就可以将它们作为测试来运行。

FIT接受HTML数据，运行其中的HTML表格所定义的测试，然后以HTML的形式输出结果。FIT的输出看上去跟输入一样，其中的所有文本和表格均被保留。然而，表格中的单元却被适当着色了，绿色代表使测试通过的数据，红色代表使测试失败的数据。你还可以通过一些选项来告诉它将测试摘要信息输出到结果HTML中。

要实现这些目的，你只需自定义一些表格处理代码，这样FIT就能够知道如何去运行你的代码块并取得返回结果。通常这项工作相当简单，因为框架本身就提供了支持一系列不同表格类型的代码。

FIT的一个非常强大的地方是，它能够鼓励软件编写者与需要指定软件应当做什么的人之间的沟通。指定软件应当做什么的人可以编写文档并将实际的测试嵌在里面。然后运行测试，无法通过，不过别担心，接下来开发者会给软件添加特性，测试便会通过。如此一来用户和开发者都能够对系统的能力有一个最近的共识。

这里所描述的远非FIT的全部，更多信息请访问http://fit.c2.com。

5.4.2 Fitnesse

Fitnesse本质上是一个以Wiki为宿主的FIT，大部分由Robert Martin和Micah Martin开发，我参与了其中的一点工作，后来因为要集中精力写作本书而退出了，希望不久能够再重新参加。 53

Fitnesse支持用于定义FIT测试的分级网页。测试表格的页面可以单独运行，或放在测试套件中运行，众多的选择使得团队间合作变得很容易。Fitnesse可以通过http://www.fitnesse.org获取。和本章中提到的所有测试工具一样，它是免费的，并由一个开发者社区支持。 54

Part 2 第二部分

修改代码的技术

本部分内容

第6章 时间紧迫，但必须修改

让我们面对现实吧：本书中描述了一些“份外”的工作，一些你可能目前并不在做的工作，一些可能会令你花更长时间来完成代码修改的工作。你或许会怀疑目前是否值得去做这些事情。

答案是：虽说不管是解依赖还是为所要进行的修改编写测试都要花上一些时间，但大部分情况下最终还是节省了时间，同时也避免了一次又一次的沮丧感。那么，究竟什么时候会发生这样的事呢？这取决于项目本身。有些情况下可能要为一些需要进行修改的代码编写测试，假如花了两个小时。而在这之后修改代码才花了不过15分钟。当回顾这番工作的时候，你可能会说：“唉，我刚才浪费了两个钟头——值得花这两个钟头的时间吗？”答案要视具体情况而定。因为你并不知道当初如果你没有编写测试的话后面的工作要花多少时间。你同样也并不知道这时如果在修改的过程中出了岔子的话需要花上多少时间去调试，而如果当初编写了测试的话这些时间是可以省下来的。所以，测试可以“捕获”你在修改过程中不慎引入的错误，从而节省为此所花的时间；另一方面，当试图寻找代码中的错误时测试也可以帮你节省时间。有测试在，我们常常就可以更容易地定位功能上的问题。

退一步说，假设遇到最坏的情况，即所要进行的改动很简单，但我们还是先编写了测试，然后正确地完成了所要进行的改动。那么，这种情况下，是否值得编写测试呢？问题是，我们并不知道什么时候会再回到这些代码上去进行其他的修改。最好的情况是，你在项目的下一个迭代期就回到那块代码上，于是当初的“投资”很快得到了回报。而最坏的情况则是，几年之后这块古老的代码才再次被人问津。不过最可能的情况则是我们会周期性地访问这块代码，哪怕只是为了看看是否需要在它上面还是其他什么地方做些改动。那么这时候如果需要修改的类比较小，或者说有现成的单元测试在那的话，是不是理解起来会容易一些呢？答案是肯定的。不过别忘了这只是最坏的情况，其发生的几率又能有多大呢？要知道，通常情况下系统中的修改是相对集中的。今天修改了某处，很可能很快又要去修改附近的某地方了。

在跟团队合作的过程中，我常常一开始就请他们参与一个实验。在一个迭代期，我们试着坚持不要在没有测试覆盖的情况下去改动代码。如果某个人觉得他们无法编写某个测试，就得召集一个临时会议，询问整个团队是否可能编写该测试。这样一个迭代期在开始的时候是糟糕的。人们觉得他们做了无用功。但是慢慢地，他们就开始发现当重访代码时看到的是更好的代码。并且代码的修改也变得越来越容易，这时他们就会打心底里觉得这么做是值得的了。当然，一个团队要想越过这个障碍期还是需要花上一些时间的，但如果说世界上还有一件事情是我可以立即为每

个团队做的话，那就是与他们分享这样的体验："好家伙，我们再也不用遭那些罪了！"

如果你还没有这种体验，那么现在就开始吧。

这种方式最终将会大大提升工作效率，而这几乎在每个开发团体中都很重要。不过坦白地说，作为一个程序员，我感到最庆幸的还是它令我们的工作变得远远不像原先那么令人沮丧了。

不过，并不是说在越过了这个障碍之后一切就是无限光明了。当你意识到测试的价值，并且感受到有和没有测试之间的差别之后，剩下来唯一要做的事情便是针对每种特定的情况决定该怎么做了。

这种情景每天都在重演

老板走了进来，说："客户们嚷着要这个特性。今天能完成吗？"

"我不知道。"

你检查了一下手头的项目，有现成的测试吗？没有。

你问："到底有多急迫？"

你知道自己可以在需要改动的一共10处地方逐一就地修改，这事在五点之前就能完成。事情非常紧急？明天我们会把代码修改一下的，对吧？

记住，代码就是你的家，你是得在其中生活的。

当处在期限压力之下时，决定是否编写测试就成了个难题，最困难的地方就在于你可能并不知道添加某个特性需要花多少时间。尤其当对象是遗留代码时，更是难以作出有效的估计。有一些技术可能帮得上一点忙，具体阅读第16章。当你并不知道添加某个特性会花费多少时间，并且 58
怀疑无法在期限前完工时，很多人可能会不管三七二十一，忍不住以最快的速度先把这个特性给弄出来再说，然后等有了充足时间之后再回过头去进行一些测试和重构。然而，这里的"回过头去进行一些测试和重构"正是困难所在。人们在越过前面提到的障碍期之前常常是采取回避的态度。这可能会成为一个"士气"上的问题。请阅读第24章，这部分对此有一些建设性的意见。

到目前为止我们所描述的似乎是个左右为难、进退维谷的境地：是现在就花时间呢还是等到以后付出更多的代价？要么在进行修改时编写测试，要么就得忍受系统变得越来越难对付的现实。虽然有时候的情况的确会像这样棘手，但有些时候并不是。

如果现在就需要对某个类进行修改，可以先试着在测试用具中实例化这个类。若不能，请参考第9章或者第10章。将要修改的代码放入测试用具可能比想象的要简单。如果在了解这两章所提到的方法之后还是觉得实在没法承受现在就去解依赖并安置好测试的代价的话，那就仔细分析一下你所要进行的修改。可以通过编写全新的代码来完成它吗？很多时候这是可行的。本章的其余部分就描述了这方面的几个技术。

阅读下面介绍的技术并考虑采用它们，但要记住，用这些技术的时候必须小心。当你用这些技术时，虽然是在系统中添加已被测试的代码，然而除非你用测试覆盖了所有调用这些代码的代码，否则还是没有对这些代码的使用进行测试。所以，小心使用。

6.1　新生方法

当需要往一个系统中添加特性且这个特性可以用全新的代码来编写时，建议你将这些代码放在一个新的方法中，并在需要用到这个新功能的地方调用这一方法。你可能没法很容易地将这些调用点置于测试之下，但至少可以为新编写的那部分代码进行测试。下面就是一个例子：

```
public class TransactionGate
{
    public void postEntries(List entries) {
        for (Iterator it = entries.iterator(); it.hasNext(); ) {
            Entry entry = (Entry)it.next();
            entry.postDate();
        }
        transactionBundle.getListManager().add(entries);
    }
    ...
}
```

59

对于上面的类，我们需要添加代码来检查entries中的对象在日期被发送并添加到transactionBundle中去之前是否已经存在了。从上面的代码来看，似乎这一检查需要放在postEntries方法的一开始进行，即for循环的前面。然而实际上它可以在循环当中进行。我们可以这样来进行修改：

```
public class TransactionGate
{
    public void postEntries(List entries) {
        List entriesToAdd = new LinkedList();
        for (Iterator it = entries.iterator(); it.hasNext(); ) {
            Entry entry = (Entry)it.next();
            if (!transactionBundle.getListManager().hasEntry(entry) {
                entry.postDate();
                entriesToAdd.add(entry);
            }
        }
        transactionBundle.getListManager().add(entriesToAdd);
    }
    ...
}
```

这看上去是个挺简单的修改，然而其侵入性相当强。比如说，我们怎么知道修改是正确的呢？在添加的新代码与原有的老代码之间并没有任何分界。更糟的是，代码的清晰性与原来相比稍稍下降了。问题在于我们将两个操作混在一起了：一个是日期发送，另一个是重复项检查。这个方法尚且是相当小的，就已经显出混淆不清的端倪了，而且还多出了一个新的临时变量，可想而知大的方法会怎样。临时变量并不一定是坏事，但有些时候它们会招来新的代码。试想，如果我们接下来要进行的一项修改与那些未重复的项相关（当然，是在将它们添加到transaction-Bundle中去之前）。这时你会发现代码中只有一个地方存在一个满足要求的变量：就是这个方法中的entriesToAdd临时变量。于是一个极具诱惑力的选择就产生了：将新的代码直接添加到这

个方法当中。但是，这件事情能否用另一种方式来完成呢？

答案是肯定的。我们可以将重复项移除看作一个完全独立的操作。可以使用测试驱动开发（74页）方式来创建一个新的方法uniqueEntries，如下所示：

```
public class TransactionGate
{
    ...
    List uniqueEntries(List entries) {
        List result = new ArrayList();
        for (Iterator it = entries.iterator(); it.hasNext(); ) {
            Entry entry = (Entry)it.next();
            if (!transactionBundle.getListManager().hasEntry(entry)) {
                result.add(entry);
            }
        }
        return result;
    }
    ...
}
```

60

按照测试驱动开发的思想，编写一个测试以便驱动我们写出上面这样的方法应当不算难事。完成之后，可以回到原先的代码，也就是待修改的postEntries那里，添加一个对该方法的调用，如下所示：

```
public class TransactionGate
{
    ...
    public void postEntries(List entries) {
        List entriesToAdd = uniqueEntries(entries);
        for (Iterator it = entriesToAdd.iterator(); it.hasNext(); ) {
            Entry entry = (Entry)it.next();
            entry.postDate();
        }
        transactionBundle.getListManager().add(entriesToAdd);
    }
    ...
}
```

你会发现上面的代码中依然出现了一个临时变量，但关键是现在的代码清晰多了。如果我们想要添加一些针对非重复项[1]的代码，就同样可以创建一个新方法来做这件事，然后在适当的地方调用它即可。如果最后我们发现还有其他一些代码也是针对这些非重复项的话，便可以引入一个新的类，将所有这些新生的方法移到该类中。这样做的效果就是保持了postEntries的简短，同时所有相关的方法，无论是postEntries还是新生方法都保持了简短和易于理解性。

以上就是新生方法（Sprout Method）技术的一个例子。实施这一技术时实际需要采取的步骤如下：

(1) 确定修改点。

(2) 如果你的修改可以在一个方法中的一处地方以单块连续的语句序列出现，那么在修改点

1. 也就是entriesToAdd。——译者注

插入一个方法调用，而被调用的就是我们下面要编写的、用于完成有关工作的新方法。然后我们将这一调用先注释掉（我个人喜欢把这一步放在编写新方法之前，因为这样我就能对新方法调用
61 在上下文中的样子有一个认识）。

(3) 确定你需要原方法中的哪些局部变量，并将它们作为实参传给新方法调用。

(4) 确定新方法是否需要返回什么值给原方法。如果需要的话就得相应修改对它的调用，使用一个变量来接收其返回值。

(5) 使用测试驱动的开发方式来开发新的方法。

(6) 使原方法中被注释掉的调用重新生效。

任何时候，只要你发现待添加的功能可以写成一块独立的代码，或者暂时还没法用测试来覆盖待修改方法时，我都建议你采用*新生方法*。这比直接往原方法中添加代码好多了。

有时候，当你想要使用*新生方法*技术时，会发现方法所在类的依赖关系实在太恶劣，以至于（为了测试这个*新生方法*）要想创建一个该类的实例，就得先“伪造”一大堆它的构造函数的实参才行。这时，一个替代方案便是使用*传Null*技术。而当这条路也行不通时，考虑将*新生方法*设为一个公用静态方法。这样一来，你可能不得不将原类的实例作为实参[1]传给这个*新生方法*，但至少你可以进行修改了。因此而将一个方法设为静态似乎有点奇怪，但对于遗留代码来说这种做法有时是有用的。我倾向于将类的静态方法看成是一个“临时场地”。通常当静态方法的数目累积起来了并且你发现它们共享了某些相通的变量时，就可以新建一个类，将这些静态方法移到这个新类中，并让它们成为新类的实例方法。而后面当是时候把它们变回原先那个类的实例方法时，还可以移回去，当然，前提是你终于将原先的类置于测试之下了。

优点和缺点

*新生方法*技术有优点也有缺点。让我们先来看看它的缺点。首先，当使用它时，效果上等于暂时放弃了原方法以及它所属的类，也就是说你暂时不打算将它们置于测试之下和改善它们了，而只打算写一个新的方法来实现某个新功能。有些时候放弃一个方法或类是迫于现实，但还是有点令人惋惜的，因为你的代码处于一个尴尬的境地。原方法可能包含了大量复杂的代码以及一个新生方法。有时候事情并不明朗，为什么偏偏就那点工作要放到其他地方去呢？况且它又令原方法“身陷囹圄”。不过至少它也提醒你，当终于将原类置于测试之下时，可以回头做一点补救。

62 尽管*新生方法*技术有一些缺点，但它也有一些突出的优点。比如新旧代码被清楚地隔离开。这样即使暂时没法将旧代码置于测试之下，至少还能单独去关注所要作的改动，并在新旧代码之间建立清晰的接口。你会看到所有被影响到的变量，更容易确定新的代码在上下文中是否是正确的。

6.2　新生类

虽然*新生方法*已算是一门强大的技术，但在一些依赖关系错综复杂的场合，这一技术还不够

1. 作用相当于this隐参。——译者注

强大。

设想，这样一种情形：你要修改一个类，但想尽一切办法也不可能在合理时间限期之内使这个类在测试用具中被实例化，这就表示你不可能在这个类上创建新生方法并为其编写测试。或许你的类在对象创建方面有大量的依赖（比如构造函数的参数的依赖），使得该类难以实例化。又或者有许多隐藏的依赖。要想解除这些依赖，得进行大量侵入式重构将它们分离出来，从而最终使该类能够在测试用具中编译。

遇到这类情况，就可以创建另一个类来容纳所要进行的改动，并在原类中使用这个新类。让我们来看一个简化的例子。

下面是一个名为QuarterlyReportGenerator的C++类上的一个古老方法：

```
std::string QuarterlyReportGenerator::generate()
{
    std::vector<Result> results = database.queryResults(
                                              beginDate, endDate);
    std::string pageText;

    pageText += "<html><head><title>"
            "Quarterly Report"
            "</title></head><body><table>";
    if (results.size() != 0) {
        for (std::vector<Result>::iterator it = results.begin();
                it != results.end();
                ++it) {
            pageText += "<tr>";
            pageText += "<td>" + it->department + "</td>";
            pageText += "<td>" + it->manager + "</td>";
            char buffer [128];
            sprintf(buffer, "<td>$%d</td>", it->netProfit / 100);
            pageText += std::string(buffer);
            sprintf(buffer, "<td>$%d</td>", it->operatingExpense / 100);
            pageText += std::string(buffer);
            pageText += "</tr>";
        }
    } else {
        pageText += "No results for this period";
    }
    pageText += "</table>";
    pageText += "</body>";
    pageText += "</html>";

    return pageText;
}
```

63

现在我们要给它生成的HTML表格添加一行标题栏，格式就像下面这样：

```
"<tr><td>Department</td><td>Manager</td><td>Profit</td><td>Expenses</td></tr>"
```

此外，我们假设QuarterlyReportGenerator是一个巨大的类，要将它成功放入测试用具中得花上大约一天工夫，而这么长的时间是我们无法忍受的。

于是，我们可以将改动做成一个小型的类，叫QuarterlyReportTableHeaderProducer，并使用测试驱动的开发方式来开发这个类。

```
using namespace std;

class QuarterlyReportTableHeaderProducer
{
public:
    string makeHeader();
};

string QuarterlyReportTableProducer::makeHeader()
{
    return "<tr><td>Department</td><td>Manager</td>"
        "<td>Profit</td><td>Expenses</td>";
}
```

完成这个类之后，可以创建它的实例，并在QuarterlyReportGenerator::generate()中使用它：

```
...
QuarterlyReportTableHeaderProducer producer;
pageText += producer.makeHeader();
...
```

相信读到这里你肯定会忍不住说："嗨！这家伙肯定在说笑。就为这点小事去创建一个新类也太荒唐了！这只是一个小得不能再小的类，在设计上也没有带来任何好处，而且它引入的一个全新概念使代码更混乱了。"诚然，这个时候你的确是对的。但请注意，之所以这么做，唯一的目的就是为了摆脱一个恶劣的依赖环境，让我们更仔细地考察这个问题。

64

如果我们将这个新类命名为QuarterlyReportTableHeaderGenerator，并把它的接口调整为下面这样，情况又会如何呢？

```
class QuarterlyReportTableHeaderGenerator
{
public:
    string generate();
};
```

这么一来这个类就成了我们所熟悉的概念的一部分了。QuarterlyReportTableHeaderGenerator与QuarterlyReportGenerator一样是个生成器，它们同样都具有返回字符串的generate()方法。因此我们用代码的形式来将这一共性文档化，即创建一个公共的接口类，并让这两个类都继承这个接口，如下：

```
class HTMLGenerator
{
public:
    virtual ~HTMLGenerator() = 0;
    virtual string generate() = 0;
};
```

```
class QuarterlyReportTableHeaderGenerator : public HTMLGenerator
{
public:
    ...
    virtual string generate();
    ...
};

class QuarterlyReportGenerator : public HTMLGenerator
{
public:
    ...
    virtual string generate();
    ...
};
```

随着工作的进一步开展，我们最终可能将QuarterlyReportGenerator放入测试并修改其实现，让它使用新的生成器类来完成其大部分工作。

在这个例子中，我们能够快速将新类归入到应用中既有的那些概念中去。而在许多其他场合下则做不到这一点，但这并不意味着应该退却。有些新生类可能永远都无法归入到当前应用中的那些主要概念中。取而代之的是它们自身变成了新的概念。你最初新建一个类时，可能会觉得它对于你的设计来说相当无关紧要，直到发现在某些其他地方也做了类似的事情并看到了它们之间的相似之处。有时能够将那些新生类中重复的代码分解出来，通常你还得给它们重命名，但别指望这个过程能够一蹴而就。 65

你初次创建一个新生类时看待它的方式和几个月后看待它的方式往往会有相当大的差别。系统中存在这么个怪异的新类，这一事实本身就带给了你大量的对它进行思考的机会。比如当你需要做一个与它联系紧密的修改时，可能就会开始想这个修改是否是这个新生类的概念的一部分，或者说这个概念本身需要做点调整。所有这些都是设计过程的一部分。

从根本上来说，两种情况下我们得使用新生类（Sprout Class）。第一种情况：所要进行的修改迫使你为某个类添加一个全新的职责。例如，在报税软件中，当前年份的某些特定时期某些课税减免可能是不可行的。为此你可能想到往TaxCalculator类中添加日期检查功能，但这个检查已经偏离了TaxCalculator的主要职责了。顾名思义，TaxCalculator的主要职责就是税的计算。所以日期检查功能应该被做成一个新的类。另一个例子就是本章开头的那个例子。我们想要添加的只是一点小小的功能，可以将它放入一个现有的类中，但问题是我们无法将这个类放入测试用具。哪怕至少能将它编译进测试用具，也还能试着用新生方法技术，只可惜有时候就连这点运气都没有。

弄清这两种情况的关键在于认识到虽说它们之间的动机不同[1]，但从结果来看其实并无显著区别。一个功能是否强大到足以成为一个新的职责，完全凭个人判断。此外，由于代码会随着时间的推移不断变化，所以决定催生出一个新类常常在事后被证明是较好的选择。

1. 一个是为了避免职责混淆，另一个则是因原类无法放入测试用具。——译者注

新生类技术的步骤如下：

(1) 确定修改点。

(2) 如果你的修改可以在一个方法中的一处地方以单块连续的语句序列出现，那么用一个类来完成这些工作，并为这个类起一个恰当的名字。然后，在修改点插入代码创建该类的对象，调用其上的方法（这个方法就是负责完成你需要完成的任务的方法），然后将刚插入的这几行代码注释掉。

(3) 确定你需要原方法的哪些局部变量，并将它们作为参数传递给新类的构造函数。

(4) 确定新生类是否需要返回什么值给原方法，如果需要，则在该类中提供一个相应的方法，并在原方法中插入对它的调用来获得其返回值。

66 (5) 使用测试驱动的开发方式来开发这个新类。

(6) 使原方法中（第一步）被注释掉的代码重新生效。

优点和缺点

新生类技术的主要优点就在于，它让你在进行侵入性较强的修改时有更大的自信去继续开展自己的工作。在C++中，新生类技术还有一个额外的好处，就是不必改动任何已有的头文件就可以完成修改。你可以在原类的实现文件中包含新生类的头文件。此外，往项目中添加一个新头文件也是件好事。随着时间的推移你会逐渐把声明都放到新头文件中去（否则最终这些声明就会落到原类的头文件中了）。这一做法降低了原类的编译负担。至少你知道并没有雪上加霜。一段时间以后，你或许能够重访原类，并将其置于测试之下。

新生类技术的主要缺点在于它可能会使系统中的概念复杂化。随着程序员对一个新的代码基的学习不断深入，他们会对其中的核心类如何协同工作建立起一种认识。而当使用新生类时，就开始破坏系统中原有的抽象，并将大批工作放在其他类中进行。不可否认，有些时候这么做的确是完全正确的。但也有一些时候则是因为别无选择，为了能够安全地进行修改，理想情况下应当呆在原有类中的代码最终却栖身在新生的类当中，实属无奈之举。

6.3　外覆方法

给现有方法添加行为是件简单的事情，但常常却并非正确。一个方法最初被建立起来时通常只为一个客户做单一的事情。此后往里面添加的任何代码某种程度上都是值得怀疑的。很可能你添加这些代码只是因为它必须跟其他那些代码同时执行而已。早些年这被称为时间耦合，耦合的过度出现是件相当可怕的事情。仅因为两段代码在同一时间发生就组织到一起，它们之间的关系并不是很强的。一段时间后你可能又会发现只需要执行其中的一段代码而不需要同时执行另一段了，可是到那时候也许这两段代码已经“生长”在一起了。在没有接缝的情况下，要想将它们剥
67 离开来是件困难的工作。

当需要添加行为时，可以考虑使用不那么“纠缠”的方式。可以使用的技术之一就是新生方法，但还有一项技术有些时候也是很有用的。我把它称为外覆方法（Wrap Method）。下面就是一

个简单的例子：

```
public class Employee
{
    ...
    public void pay() {
        Money amount = new Money();
        for (Iterator it = timecards.iterator(); it.hasNext(); ) {
            Timecard card = (Timecard)it.next();
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * payRate);
            }
        }
        payDispatcher.pay(this, date, amount);
    }
}
```

在上面的方法中，我们合计了一个雇员（Employee）每天的考勤卡（Timecard），然后将他的薪水支付信息发给一个PayDispatcher。现在假设出现了一个新的需求。每次我们给一个雇员支付薪水时都得做一下日志记录，以便将日志发给某个报表系统。不管怎样，这个新的功能必须跟原有功能在同一时间发生。如果我们像下面这么做：

```
public class Employee
{
    private void dispatchPayment() {
        Money amount = new Money();
        for (Iterator it = timecards.iterator(); it.hasNext(); ) {
            Timecard card = (Timecard)it.next();
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * payRate);
            }
        }
        payDispatcher.pay(this, date, amount);
    }

    public void pay() {
        logPayment();
        dispatchPayment();
    }


    private void logPayment() {
    ...
    }
}
```

在上面的代码中，我们将原先的pay()重命名为dispatchPayment()，并将它改为私有方法。接着我们创建一个新的pay()方法，它调用了重命名后的dispatchPayment()。不过，这个新的pay()方法会先将一次薪水支付记录下来，然后才去调用dispatchPayment()。如此一来，以前一直是调用pay()方法的客户则不必知道也不必关心这次改动。他们还像原来那样进行调用，一切都运行良好。

这是外覆方法的运用形式之一：创建一个与原方法同名的新方法，并在新方法中调用更名后的原方法。当想要为原方法的既有调用添加行为时，就可以采用这种做法。如果希望客户每次调用pay()时都会产生日志记录的话，该技术是十分有用的。

外覆方法还有另一种运用形式，如果只是想增加一个尚未有任何人调用的新方法，就可以采用这一形式。例如，在前面的例子中，如果想要将日志记录显式暴露出来，则可以为Employee类增加一个makeLoggedPayment方法，如下：

```
public class Employee
{
    public void makeLoggedPayment() {
        logPayment();
        pay();
    }

    public void pay() {
        ...
    }

    private void logPayment() {
        ...
    }

}
```

这样用户就可以在两种支付方式之间自由选择。Kent Beck在*Smalltalk Patterns: Best Practices*（Pearson，1996）一书中描述了这一技术。

要想在添加新特性的同时引入接缝，外覆方法是极好的选择。它只有少数几个缺点。第一，你添加的新特性无法跟旧特性的逻辑“交融”在一起。它们要么在旧特性之前要么在之后完成。事实上这并非坏事，建议你尽量这么做。第二个缺点，也是更为实际的一个缺点就是，你得为原方法中的旧代码起一个新名字。本例中我是将原pay()方法中的那些代码命名为dispatchPayment()。其实这并不十分恰当，而且说实话我也并不喜欢例子中代码最终的样子。dispatchPayment()其实并不仅仅是进行“dispatch（分发）”，它还负责计算薪水。如果它的测试已被安置到位的话，很可能我会将它的第一部分提取到一个独立的方法calculatePay()中去，并对pay()方法作相应改动：

69

```
public void pay() {
    logPayment();
    Money amount = calculatePay();
    dispatchPayment(amount);
}
```

这样所有职责就都被良好地分离开来了。

外覆方法的第一种形式的实施步骤如下：

(1) 确定待修改的方法。

(2) 如果你的修改可以在一处地方以单块连续的语句序列出现，那么将待修改方法重命名，

并使用其原先的名字和签名创建一个新方法。在这么做的时候记住要*签名保持*（249页）。

(3) 在新方法中调用重命名后的原方法。

(4) 为欲添加的新特性编写一个方法（当然，还是编写测试在先），并在第2步创建的新方法中调用这个方法。

在外覆方法的第二种运用形式中，并不一定要使用跟旧方法同样的名字来命名新方法，步骤如下：

(1) 确定待修改的方法。

(2) 如果你的修改可以在一处地方以单块连续的语句序列出现，那么用测试驱动的开发方式编写一个新方法来容纳新的特性。

(3) 创建另一个函数来调用新旧两个方法。

优点和缺点

当我们没法为调用代码编写测试时，外覆方法是将新的、经过测试的功能添加进应用中的好途径。新生方法和新生类都会将代码添加到现有方法中，至少增加一行，而外覆方法则不会增加现有方法的体积。

外覆方法另一个好处就是它显式地使新功能独立于既有功能。为某一目的而作的代码不会跟另一意图的代码互相纠缠在一起。

外覆方法的主要缺点在于它可能会导致糟糕的命名。在上面的例子中，我们将原方法`pay()`重命名为`dispatchPay()`只是因为要给它起一个新名字。如果代码并非十分脆弱或复杂，或者如果有一个能够安全实施*方法提取*（325页）的重构工具，就能够进行进一步的提取，并最终得到更好的命名。然而，许多时候我们之所以进行方法外覆正是因为缺少相应的测试，代码脆弱且没有上述工具。 70

6.4　外覆类

外覆方法的类版本便是*外覆类*（Wrap Class），两者概念几乎一模一样。如果需要往一个系统中添加行为，我们固然可以将该行为放到一个现有的方法中，但我们同样可以将它放到一个使用了该方法的类当中。

让我们来回顾一下`Employee`类：

```
class Employee
{
    public void pay() {
        Money amount = new Money();
        for (Iterator it = timecards.iterator(); it.hasNext(); ) {
            Timecard card = (Timecard)it.next();
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * payRate);
            }
        }
        payDispatcher.pay(this, date, amount);
```

```
    }
    ...
}
```

假设想要让日志记录我们正在向某个特定的雇员支付薪水的事实。可以新建一个具有pay()方法的类。该类的对象可以持有一个雇员对象，在这个pay()方法中完成日志记录，然后将剩下的任务委托给这个雇员对象上的相应方法[1]来完成。通常，如果无法在测试用具中实例化原类的话，完成上述工作的最简单途径就是对原类使用实现提取（281页）或接口提取（285页），并让外覆类实现该接口。

在下面的代码中，我们使用了实现提取手法来将Employee类变成一个接口。然后我们新建一个名为LoggingEmployee的新类，让它实现该接口。我们可以将任何Employee传递给LoggingEmployee以便后者用它来做日志并同时付酬。

```
class LoggingEmployee extends Employee
{
    public LoggingEmployee(Employee e) {
        employee = e;
    }

    public void pay() {
        logPayment();
        employee.pay();
    }

71  private void logPayment() {
        ...
    }
    ...
}
```

该技术在设计模式里面被称作装饰模式。我们用一个类来外覆另一个类，并创建/传递那个外覆类的对象。这里的外覆类须得具有与被外覆类相同的接口，这样一来使用者就不会知道他们使用的是一个外覆类了。上例中的LoggingEmployee是Employee的一个饰类，它得具有pay()方法，以及Employee上会被客户用到的所有其他方法。

装饰模式

装饰模式允许你通过在运行期组合对象来构建复杂的行为。例如，在一个工业过程控制系统中，我们可能会看到一个叫做ToolController的类，该类具有raise()、lower()、step()、on()以及off()等方法。如果想要在每次raise()或lower()时都做一些额外的事情（如发出蜂鸣来警告人们注意操作安全），一条显而易见的途径就是将这一功能直接放到这两个方法中。然而很可能接下来还想进行其他改进。例如最后我们可能需要记录下控制器开关的次数。可能还想在我们进行step()时通知附近的其他控制器不要同时step()。伴随着这五步简简单单的操作（raise、lower、step、on和off）可以进行的额外工作是无穷无尽的，

1. 上面的示例代码中的pay()。——译者注

为其中每种组合都创建一个子类的做法显然是不实际的，因为可能的组合是无穷无尽的。

装饰模式正适合用在这类场合下。使用该模式时，首先创建一个抽象类，该抽象类定义了你需要支持的一组操作。然后创建该抽象类的一个子类，该子类的构造函数接受任一从抽象类派生出的实体类的对象，并且该子类为抽象类中的每个方法提供一个实现。下面就是ToolController问题的解决方案：

```
abstract class ToolControllerDecorator extends ToolController
{
protected ToolController controller;
    public ToolControllerDecorator(ToolController controller) {
        this.controller = controller;
    }
    public void raise() { oontroller.raise(); }
    public void lower() { controller.lower(); }
    public void step() { controller.step(); }
    public void on() { controller.on(); }
    public void off() { controller.off(); }
}
```

72

上面这个类看起来可能并不是很有用，但实际上恰恰相反。你可以对它进行子类化并重写它的任一或所有方法来添加额外的行为。例如，如果我们需要在步进时通知其他控制器的话，就可以编写一个名为StepNotifyingController的类，如下：

```
publio class StepNotifyingController extends ToolControllerDecorator
{
    private List notifyees;
    publio StepNotifyingController(ToolController controller,
            List notifyees) {
        super(controller);
        this.notifyees = notifyees;
    }
    public void step() {
        // notify all notifyees here
            ...
        controller.step();
    }
}
```

这一手法的真正漂亮之处在于我们可以将ToolControllerDecorator的子类“层层嵌套”起来：

```
ToolController controller = new StepNotifyingController(
                        new AlarmingController
                        ( new ACMEController()), notifyees);
```

对于像上面这样创建起来的一个对象（以控制器持有），当我们调用控制器上的某个操作（如step()）时，它不仅会通知其他相应的控制器，还会发出蜂鸣，同时最后会执行真正的step()方法，这最后一步是在ACMEController内部发生的，ACMEController是ToolController的一个实体子类，而不是ToolControllerDecorator的子类。它并不将

任务交给其他ToolController去做，而是自己完成。使用装饰模式的时候你至少需要一个像这样的“基础”类，并从这种类开始一层层进行外覆。

装饰是个不错的模式，但还是保守使用比较好。在一个装饰类套一个装饰类的代码中“行走”就好像是在一层一层地剥洋葱皮一样，剥洋葱皮固然是必要的工作，但弄不好会让你呛出眼泪来。

当已经存在了许多对类似pay()方法的调用时，以上就是一种很好的往其中添加功能的途径。不过，还有一个不那么“装饰性”的方法也可以用来进行外覆。设想这样一种情况：只需在唯一一处地方记录对pay()的调用。这次我们不采取将这一功能作为饰类来对pay()进行外覆的做法，而是把它放到另一个类当中，该类接受一个Employee对象，调用它的薪水支付方法，然后用日志记录下关于这次支付的相关信息。

下面就是这样一个类：

```
class LoggingPayDispatcher
{
    private Employee e;

    public LoggingPayDispatcher(Employee e) {
        this.e  = e;
    }

    public void pay() {
        employee.pay();
        logPayment();
    }

    private void logPayment() {
        ...
    }
    ...
}
```

73

于是现在就可以在那处需要记录支付信息的地方改用LogPayDispatcher（而不是Employee）了。

外覆类的关键在于，当用它往系统中添加新行为时，无需将新行为塞到现有类当中去。倘若已经存在了一堆对你想要进行外覆的代码的调用，使用“装饰性”的外覆手法通常是不错的选择。借助于装饰模式，你可以透明地一次性将新的行为添加到一组像pay()这样的现存调用上。另一方面，如果你只需要将新的行为添加到少数几个地方的话，一个很有用的做法就是创建一个非“装饰性”的外覆类[1]。并且，随着时间的推移，应该对外覆类的职责加以关注，看看它能否成为系统中的另一个更高级的概念。

下面就是外覆类手法的步骤：

1. 见上面的例子。——译者注

(1) 确定修改点。

(2) 如果你的修改可以在一处地方以单块连续的语句序列出现，则新建一个类，该类的构造函数接受需要被外覆的类的对象为参数。如果你无法在测试用具中创建外覆类的实例的话，你可能需要先对被覆类使用实现提取或接口提取技术，以便能够实例化外覆类。

(3) 使用测试驱动的开发方式为你的外覆类编写一个方法，该方法负责完成你想要添加进系统中去的工作。编写另一个方法，这个方法负责调用刚才创建的那个方法以及被覆类中的旧方法。

(4) 在系统中需要使用新行为的地方创建并使用外覆类的对象。

新生方法与外覆方法的区别是相当细微的。在使用新生方法时，你创建一个新方法，并在现存方法中调用它。而当使用外覆方法时，则是先重命名一个现有方法，然后用一个新建的方法来 74
替代它（起名为它原先的名字），这个新建的方法会完成你想要往系统中添加的工作，并将剩下的任务委托给重命名后的旧方法。通常，如果现有方法中的代码将一个清晰的算法传达给了读者，我就会使用新生方法。而如果我觉得欲添加的新特性的重要性跟已经存在的特性不相上下，就会转而使用外覆方法。在后一种情况下，完成外覆之后，通常会得到一个新的高层算法，如下所示：

```
public void pay() {
    logPayment();
    Money amount = calculatePay();
    dispatchPayment(amount);
}
```

而是否选择外覆类则完全是另一个问题。外覆类的“使用阈值”要更高一些。一般来说，两种情况促使我们去使用外覆类：

(1) 欲添加的行为是完全独立的，并且我们不希望让低层或不相关的行为污染现有类。

(2) 原类已经够大了，我实在不能想像把它撑得更大会如何。遇到这类情况，使用外覆类只是相当于在地上插个木桩，为后面的修改设下一个标识。

以上第二种情况比较难办，也比较难以习惯。设想你有一个非常大的类，具有例如说10个甚至15个不同的职责，这时只为了添加某个微不足道的功能就去新建一个外覆类似乎有点说不过去。事实上，如果你这么做而又不能在同事面前给出有说服力的理由的话，可能会被他们“优待”一番，更糟的是你的同事可能从此对你嗤之以鼻。所以……让我告诉你怎么处理吧。

在对一个大的代码基进行改进时，最大的障碍就是现存代码。对此，你可能不以为然。但请注意，关键不在于那些难对付的代码要花多少工夫去对付，而是这样的代码给你带来的心理负担。如果你把一天中的大部分时间浪费在丑陋的代码之中，很快就会觉得这些代码永远也没有漂亮起来的一天，而且试图对它进行改进的任何努力都是不值得的。你可能会想：“我做的这点工作相对于整个系统的糟糕现状来说无异于杯水车薪，又能起到多大作用呢？没错，我可以把这一小块代码加以改进，但那对我今天下午的工作又有多大好处呢？明天又将如何呢？”唉，如果你这么想的话，我当然没法不同意你。但如果你持续不断进行这类小改进的话，几个月后你的系统会变 75
得大不一样了。某天早晨当你一如既往地开始工作时，会惊讶地发现原来那团丑陋不堪的代码消失了，你想：“唔……这代码看起来相当不错嘛。似乎有人最近一直在对它进行重构。”那一刻你会由衷发现好的代码跟糟糕的代码之间的区别，这就代表你的观念彻底转变了。你甚至发现自己

主动想要去进行更多的重构了，而这么做只是为了令自己以后的工作轻松些。当然，如果你从未经历过类似以上的场景，这一切对你来说可能有点滑稽，但可以告诉你的是，我一次又一次地见证了这样的场景发生在一个又一个团队中。要想做到这些，最困难的地方就在于最初的几步，因为有时候你可能会觉得没必要。“什么？就为了添加这么个小小的特性还要去大费周章地外覆一个类？这下弄得似乎比之前还要糟糕了、更复杂了。”是的，就当时来说这的确没错，但当你开始逐渐分解那个被覆类中的10或15个职责时就不会这么想了。

6.5　小结

本章罗列了一系列技术，借助于它们，你无需将现有类纳入测试便可以对其进行修改。从设计的角度来说，很难说清到底应该怎么来理解它。许多时候，这么做使得我们能够在新旧职责之间拉开一段距离。换句话说，我们正朝着更好的设计前进。然而在另一些情况下，之所以去创建一个新类唯一的原因就是我们想要编写受测试的新代码，而目前还没有时间去将那个现有类纳入测试。这是个很现实的情况。当你在实际项目中这样做时，将会看到新的类和方法在旧的、巨大的类旁不断冒出来。但接下来会发生一件有趣的事情：一段时间之后，你对于总是避开旧类庞大的身躯感到厌烦了，于是开始试图将它纳入测试中。而这项工作的一个成分就是去熟悉你要纳入测试的类，所幸的是，因为你常常要去查看这个巨大的未测试类来决定从什么地方抽生出新类来，也变得越来越了解它。因此把它纳入测试也就变得越来越不可怕了。此外，这项工作的另一个部分就是要避开厌烦情绪。你看着屋子里的垃圾会感到厌烦，想要将它们扫地出门，请参考第9章和第20章。

76

第7章 漫长的修改

实施一次修改要花多少时间？答案不一定，不同的情况下差异可能相当大。对于那些代码极度不清晰的项目，大部分修改可能都需要花上很长时间。首先我们得阅读并认识代码，明白修改会带来的所有影响，然后才能动手去修改。对于比较清晰的部分，修改起来可能很快，而对于那些“盘根错节”的部分，则可能要花上很长时间。此外，团队与团队之间的情况也有所不同，有些团队情况很糟糕，即便是最简单的修改也要花上大量时间。他们知道需要添加的是哪些特性，能够想像出应当在哪儿进行修改并在5分钟内完成，然而几小时后仍无法发布他们的修改。

让我们来看一看其背后的原因以及一些可能的解决方案。

7.1 理解代码

随着代码量的增加，项目就会变得越来越难理解。于是人们也就需要花费越来越多的时间才能弄清应当修改什么。

某种程度上这是不可避免的。当往一个系统中添加代码时，可以将代码添加到现有的类/方法/函数中，也可以创建新的类/方法/函数。不管采取哪种做法，只要我们尚不熟悉相关的上下文，就没法很快知道如何进行修改。

然而，一个维护良好的系统和一个遗留系统之间有一个显著的区别：对于前者，你可能要花上一点时间来想想该如何修改，一旦想清楚了，改起来往往很容易，改完后的系统也感觉舒服多了。而在一个遗留系统中，可能得花上很长一段时间来搞清楚应该怎么做，同时修改起来往往也不容易。此外你可能还会觉得，除了因为修改而必须去理解的那一小块代码之外，并没有了解到多少其他东西。最糟的情况是，需要预先理解的代码好像不管花上多少时间都理解不完似的，最后你不得不闭着眼睛冲进代码，心里默默祈祷自己能够搞定所有将要遇到的问题。

那些由小块的、命名良好的、可理解的部件组成的系统对付起来更为容易。如果在你的项目中，“代码理解”是一个大问题的话，请阅读第16章和第17章寻找应对之策。

7.2 时滞

还有一个非常普遍的因素会导致修改耗时的延长，这个因素就是时滞（lag time），是指从做出

修改到得到反馈所经历的时间。打个比方，在我写下这些时，火星漫步者Spirit正在火星表面缓慢爬行，拍摄照片。信号从地球传到火星大约要花上7分钟。幸运的是，Spirit车载的导航软件能够帮助它自行在地面上移动。不过你可以设想一下，若是在地球上手动驾驶这个远在火星的探测器会是什么样的情形。每一个操作都要等上14分钟才能看到结果。然后根据结果再决定下一步干什么，完了之后再等14分钟……效率如此之低简直荒唐。然而，如果仔细想想，你会发现这正是大多数人目前在软件开发中使用的方式。我们通常作一些改动，编译并构建，看看会发生什么。然而遗憾的是，并没有一个软件知道如何躲避构建过程中的暗礁，所谓暗礁就是指像测试失败这类东西。取而代之的是，我们试图一次性进行一系列的修改，以免构建活动太过频繁。如果修改没问题，则一切平安无事，我们继续，行动速度就如Spirit一样慢。而如果触礁了的话，则会慢上加慢。

这种工作方式的悲哀之处就在于，在大多数语言中完全没必要这样做，这完全是浪费时间。实际上，在大多数主流语言中，都可以通过解依赖在10秒钟内实现对代码的重编译并运行测试。在大多数情况下，如果一个团队真的鼓起劲来干的话，这个时间甚至可以缩短到5秒以内。最终的情况就像这样：对于系统中的每个类或模块，你应当能够独立地在它们各自的测试用具中编译
78 它们。只要做到了这一点，就能得到快速反馈，从而提升开发效率。

人类大脑有一些有趣的地方。比如说，假设我们要执行一个短任务（5~10秒），然而每分钟却只能执行该任务的一个步骤。那么我们会怎么做呢？通常我们会完成一步，然后停下来等着。如果这期间需要做一些工作来决定下一步该干什么的话，我们就开始计划。计划完之后，大脑便无事可做，等着下一步开始时刻的到来。然而，如果将步与步之间的间隔从一分钟缩短到几秒钟，情况就不同了。我们可以利用反馈来快速尝试不同的方案。于是我们的工作更像是在驾驶汽车，而不是在车站等公交车了。此外，我们也更能集中注意力，因为不再老是等着走下一步了。然而最重要的还是，花在发现并改正错误上的时间大大缩短了。

那么，究竟是什么使我们一直以来都无法以这种快捷的方式工作呢？事实上，有些人是可以的。那些用解释型语言编程的程序员在工作当中通常可以获得几乎实时的反馈。而对于我们这些使用编译型语言的程序员，阻碍我们获得快速反馈的主要拦路石则是依赖，具体地说就是为了编译我们想要编译的代码而不得不连带编译那些我们并不关心的代码。

7.3　解依赖

依赖可能会带来一些问题，不过所幸我们可以解开它们。对于面向对象的代码，通常第一步就是试图在测试用具中实例化我们所需的类。在最简单的情况下，我们只需导入或者包含我们所依赖的那些类即可。如果情况麻烦一些，可以试试第9章所描述的技术。此外，就算能够将某个类放入测试用具中，但为了测试该类的个别方法，或许还得去解开其他某些依赖。对于这些情况，请参考第10章。

如果需要在测试用具中修改某个类，通常可以利用耗时较短的“编辑－编译－连接－测试”过程。一般来说，大多数方法的执行开销跟它们所调用的方法的开销相比都是相对较低的，尤其是当被调用方是像数据库、硬件或通信设施之类的外部资源的情况下。而其他方法大都是计算密

集型的。对此第22章描述的技术会有所帮助。

许多时候我们所作的修改都是比较直观的，但面对遗留代码，人们往往在第一步就被卡死了：就是指将一个类放入测试用具中。对于某些系统来说这可能需要花很多工夫。比如，可能有些类 79
非常大，而有些类当中的依赖是如此之多，以至于贯穿了你想要修改的功能。这种情况下建议你考虑从中切出一大块代码放入测试。第12章中包含的一组技术可以用于寻找到汇点（149页），在这些地方编写测试比较容易。

本章余下的部分将介绍如何着手修改代码的组织方式，以使其编译构建更为容易。

构建依赖

在一个面向对象系统中，如果想要让一簇类的构建更快，所要弄清的第一件事就是哪些依赖会成为拦路石。通常这很简单：你只需试着在测试用具中使用该类，所遇到的几乎所有问题都会源于某个依赖。在成功在测试用具中运行该类之后，别忘了还有一些依赖也可能会影响编译时间。一个有益的做法就是看看有哪些代码是依赖于你已经能够在测试用具中实例化的那些类的。这些代码在你重新构建系统时也需要被重新编译。那么如何将它们减到最少呢？

解决这一问题的途径，是看看这簇类里面有哪些类是被簇外面的其他类使用到的，对这些类进行接口提取。许多IDE都提供了接口提取的功能，你只需选定一个类，然后选择某个菜单来列出该类的所有方法，然后就可以从中选择某些方法来构成一个新的接口。完了之后IDE会让你为新的接口起一个名字，同样，它还会让你选择是否将代码基中所有对该类的引用尽可能替换为对该接口的引用。这是个极其有用的特性。不过在C++中，实现提取（281页）要比接口提取（285页）容易一些，因为对于前者你无需到处修改引用的名字，不过还是得修改那些创建旧类的对象的地方，将它们改为创建新类的对象（具体细节请参考关于实现提取的描述）。

一旦将这簇类置于测试之下，我们便可以修改项目的物理结构从而使其构建更快了。具体做法是将这簇类移到一个新的包或者库中去。之后我们的构建过程的确变得更复杂了，但这并不要紧，关键在于随着我们解依赖并将这些类选出来放入新的包或库中，虽然重新构建整个系统的代价增加了，但每次构建的平均时间却会降低。 80

让我们来看一个例子。图7-1展示了一组互相合作的类，它们位于同一个包中。

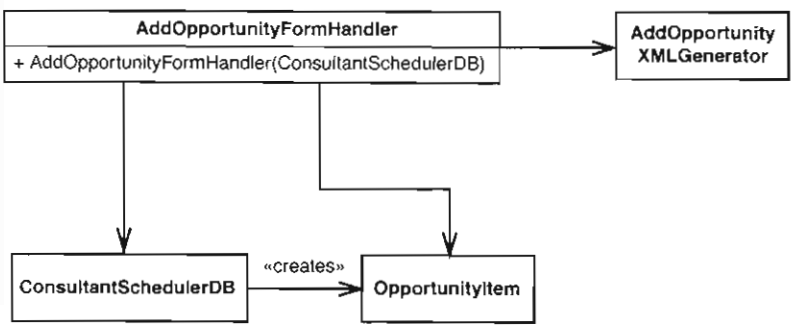


图7-1　一组互相合作的类

我们想要对AddOpportunityFormHandler作一些改动，但若是顺便能让我们的构建过程更快的话就更好了。第一步是尝试实例化AddOpportunityFormHandler。然而遗憾的是，它所依赖的类皆是具体类。AddOpportunityFormHandler需要一个ConsultantSchedulerDB以及一个AddOpportunityXMLGenerator。而很可能这两个类又依赖于其他不在这幅图中的类。

在目前的情况下，如果我们试图实例化AddOpportunityFormHandler，天知道最终会波及到哪些类。要想解决这个问题，可以开始进行解依赖。我们遇到的第一处依赖就是ConsultantSchedulerDB。需要创建一个ConsultantSchedulerDB对象并将它传递给AddOpportunityFormHandler的构造函数，而由于ConsultantSchedulerDB需要连接到数据库，所以创建起来会不大方便，我们可不想在测试时惹上这个麻烦。然而，我们可以运用实现提取技
81 术，像图7-2所示那样进行解依赖。

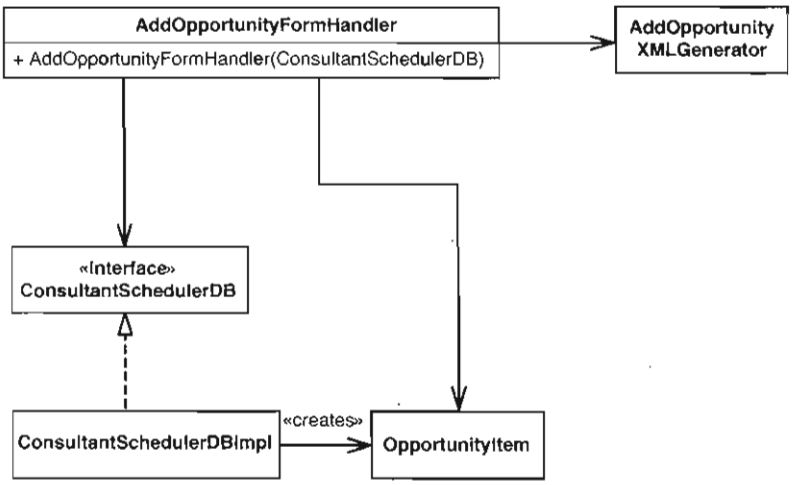


图7-2　对ConsultantSchedulerDB进行实现提取

在ConsultantSchedulerDB变成了一个接口之后，我们便可以用一个实现了ConsultantSchedulerDB接口的伪对象作为AddOpportunityFormHandler的构造函数的参数来创建AddOpportunityFormHandler对象了。有趣的是，这个依赖被解开之后，构建过程在某些情况下也变得更快了。比如说，下次我们对ConsultantSchedulerDBImpl作修改时，AddOpportunityFormHandler就无需重编译了。为什么呢？因为AddOpportunityFormHandler已不再直接依赖于ConsultantSchedulerDBImpl中的代码了。无论对ConsultantSchedulerDBImpl所在文件作多少修改，只要这些修改不至于迫使我们去改动ConsultantSchedulerDB接口，就无需重编译AddOpportunityFormHandler类。

如果我们愿意的话，甚至还可以进一步避免连带性重编译，如图7-3所示。其中显示了另一
82 个设计，这是通过对OpportunityItem类使用实现提取技术达到的。

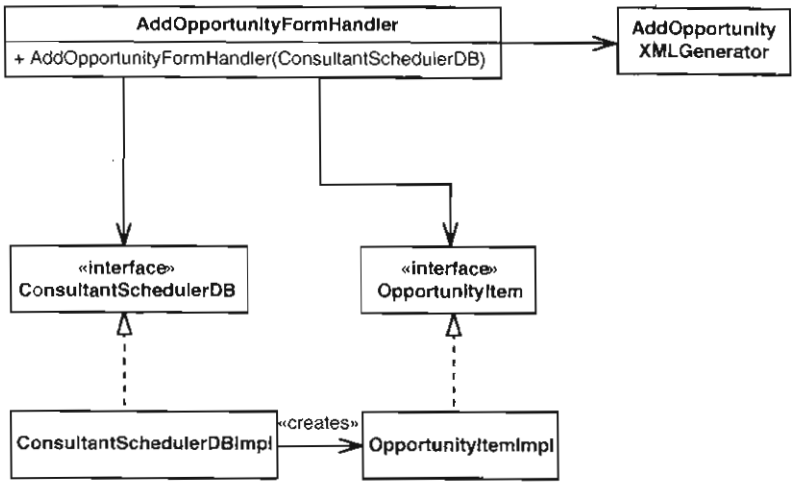


图7-3　对OpportunityItem进行实现提取

现在，AddOpportunityFormHandler不再依赖于原先OpportunityItem中的代码了。从某种意义上来说，我们在代码中插入了一面编译期的防火墙。这么一来无论对ConsultantSchedulerDBImpl和OpportunityItemImpl作多少改动都不会导致AddOpportunityFormHandler被重编译，也不会迫使任何使用了AddOpportunityFormHandler的代码进行重编译。如果我们想让应用程序的包结构也明确反映这些，可以将设计分解成下面这几个分离的包，如图7-4所示：

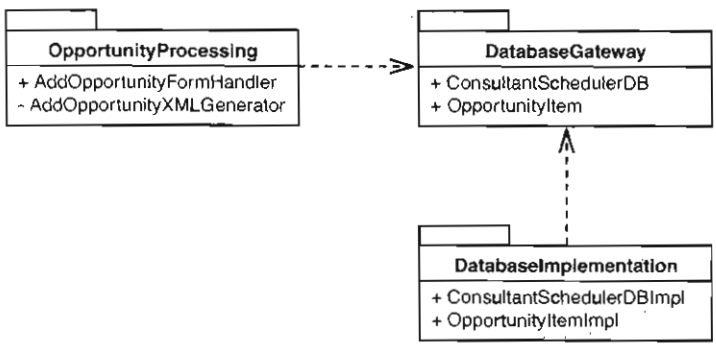


图7-4　重构后的包结构

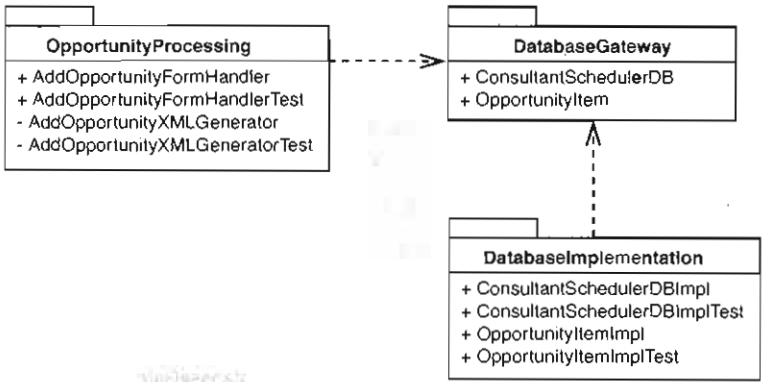
这么一来，我们的OpportunityProcessing包实质上便不依赖于数据库实现了。那么任何我们编写并放在该包中的测试都应该能够快速完成编译，而且当我们修改数据库实现类的代码时，OpportunityProcessing包本身则不必重新编译。

依赖倒置原则

如果你的代码依赖于一个接口，那么这个依赖一般来说是很次要的。除非这个接口发生改变，否则你的代码是无需改变的，而接口的改动频率通常情况下要远远低于接口背后的那些代码。在接口不变的前提下，不管是修改实现了该接口的类，还是添加实现了该接口的新类，接口的客户代码都不会受到影响。

因为这个原因，较好的做法是让代码依赖于接口或抽象类，而不是依赖于具体类。当代码依赖的是较为稳定的东西时，因特定改动而导致大规模重编译的可能性也就被降到了最低。

到目前为止我们已经做了一些工作来防止当AddOpportunityFormHandler依赖的类改变时引起AddOpportunityFormHandler被重编译。这的确会令编译变得更快，但这还只是问题的一半。剩下的一半是我们还可以令那些依赖于AddOpportunityFormHandler的代码的编译更快。让我们再来看一下包的设计，如图7-5所示：



84

图7-5　包结构

AddOpportunityFormHandler是OpportunityProcessing包中唯一公共的产品（非测试）类。当我们对其进行修改时，其他包中任何依赖于它的类都得重编译。我们同样可以通过接口提取或实现提取技术来解开其他代码对它的依赖。之后其他包中的代码便可以依赖于提取出的接口。如此一来大部分时候我们对该包的修改都不再会引起其客户代码的重编译了。

我们可以通过解依赖并将类分配给不同的包来加快编译过程，这么做是非常值得的。当你可以快速编译并运行测试时，在开发过程中便能够获得更佳的反馈。大多数时候，这就意味着更少的错误，另外情况也没那么令人恼火了。但这也并非免费的午餐，增加接口和包的数量也会增加一些概念的复杂性。值得这么做吗？答案是肯定的。的确，有时候，包和接口的数量多了以后，找起东西来要多花点工夫，但找到之后用它们工作会非常容易。

当为了解依赖而往设计中引入了额外的接口和包之后，重新构建整个系统的时间就会稍微变长一点。因为有更多的文件要去编译。但基于需要被重编译的文件而进行的局部重建的平均时耗反而大大缩短了。

当开始优化平均构建时间时，最终将会得到一块块非常易于工作的代码。是的，使一小组类能够单独编译和测试这件工作做起来可能有点痛苦，但重点是，这是一劳永逸的。一旦完成这件工作，后面就能永远享受它带来的好处。

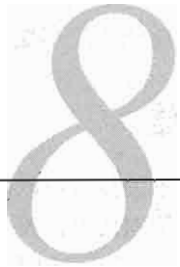
7.4　小结

本章展示的技术可以用于使小簇类的构建变得更快，但这还只是一小部分，事实上，借助于接口和包来管理依赖，我们还能做其他很多事情。Robert C. Martin的书《敏捷软件开发：原则、模式与实践》[1]展示了更多这方面的技术，每个开发者都应当知道这些技术。

85

1. 此书英文注释版和C#版的中文版即将由人民邮电出版社出版。——编者注

第8章 添加特性



“如何添加一个特性？”这或许是本书中最抽象、最与问题领域相关的问题了。但事实上，不管我们采取什么样的设计方案或面对什么样的特定约束，总还是有一些技术可以使我们的工作变得更轻松。

让我们来看一看问题的上下文。面对遗留代码，最重要的考虑之一便是其中许多代码都是没有测试的。更糟的是，我们难以将测试安置到位。因此许多团队都更倾向于退而采用第6章中描述的技术。没错，第6章描述的技术是可以用于在没有测试的情况下添加代码，但除此之外也存在着一些危险。一方面，在使用新生方法/类或外覆方法/类时，我们并没有对既有代码作多么明显的修改，因此也就不用指望这些代码会在短期内得到改善。代码重复则是另一个危险因素。如果我们添加的代码跟既有的未被测试代码中的某些部分构成了重复，那么很可能那些旧代码便只能躺在那儿等着烂掉了。更糟的是，很可能直到作了一大堆修改之后我们才发现进行了重复编码。最后一个危险因素就是恐惧乃至退缩：恐惧是指害怕无法修改某块特定的代码从而使其更易对付，而退缩则是因为整块代码一点也没改观。恐惧使你无法作出好的决定。留在代码中的新生类/方法和外覆类/方法就是明证。

一般来说，面对问题比逃避问题更好。如果我们可以将代码置于测试之下，便可以使用本章的技术来使系统朝好的方向发展。如果不知道怎么才能将测试安置到位，请参考第13章。如果觉得依赖碍手碍脚的话，请参考第9章以及第10章。

86
~
87

一旦测试到位，我们的系统便处于一个更利于添加新特性的位置上。我们便有了一个坚固的根基。

8.1 测试驱动开发

我所知道的最为强大的特性添加技术便是测试驱动开发（TDD）。简单地说，测试驱动的开发过程是这样的：设想有这么一个方法，能够帮我们解决问题的某个部分；接下来我们为这个设想中的方法编写一个失败测试用例。此时该方法尚不存在，但既然我们能够为它编写测试，我们就对接下来将要编写的代码要做什么事情有一个确定的认识。

测试驱动开发使用的算法如下：

(1) 编写一个失败测试用例。

(2) 让它通过编译。

(3) 让测试通过。

(4) 消除重复。

(5) 重复上述步骤。

下面就是一个具体的例子。假设现在有一个财务系统，我们需要一个能够利用某些高性能数学方法来验证某些商品是否应该被买卖的类。为此需要一个能够计算有关某个点的所谓“一阶统计矩”的Java类。目前这个方法尚不存在，但我们知道的是可以为它编写一个测试用例。我们了解相关的数学知识，所以知道对于在测试中给出的数据，结果应该是-0.5。

8.1.1 编写一个失败测试用例

下面就是用来测试我们所需的功能的用例：

```
public void testFirstMoment() {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);

    assertEquals(-0.5, calculator.firstMomentAbout(2.0), TOLERANCE);
}
```

8.1.2 让它通过编译

我们刚编写的这个测试没什么问题，但它无法通过编译。因为现在InstrumentCalculator上还不存在一个名为firstMomentAbout的方法。但我们可以添加一个空的firstMoment-About到InstrumentCalculator里面，这样测试用例便能够通过编译了。我们希望这个测试失败，所以让该方法返回NaN（当然不等于测试里面所期望的-0.5）。

88

```
public class InstrumentCalculator
{
    double firstMomentAbout(double point) {
        return Double.NaN;
    }
    ...
}
```

8.1.3 让测试通过

有了这个测试，便可以编写相应的代码来让测试通过了：

```
public double firstMomentAbout(double point) {
    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

在测试驱动开发当中，对于仅仅为了令测试通过这一目的来说，以上代码量已经算是异常多的了。通常情况下我们的步骤要细得多，不过，如果你对自己需要使用的算法有足够的信心，也可以采取像上面这种做法。

8.1.4　消除重复

上面的例子中有重复代码吗？没有。所以我们接着来看下一个用例。

8.1.5　编写一个失败测试用例

刚才我们编写的代码通过测试了，但显然它并未覆盖所有的情况。例如，在返回语句那行可能会出现除零的情况。对于这种情况，我们该怎么办呢？如果elements.size()为0的话，我们该返回什么？答案是抛出一个异常。因为当elements链表里面没有任何数据时，结果是无意义的。

下面这个测试比较特殊。倘若没有抛出InvalidBasisException异常，该测试就会失败。反之，如果抛出InvalidBasisException异常，那么测试便通过。当运行该测试时，由于抛出的是一个ArithmeticException（由firstMomentAbout中的除0操作引发），因而测试失败了。

```
public void testFirstMoment() {
    try {
        new InstrumentCalculator().firstMomentAbout(0.0);
        fail("expected InvalidBasisException");
    }
    catch (InvalidBasisException e) {
    }
}
```

89

8.1.6　让它通过编译

要想让上面的测试通过编译，我们得修改firstMomentAbout的声明，为它加上InvalidBasisException：

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

但这还不够。编译错误告诉我们，既然在异常规格列表里面声明了InvalidBasisException异常，就得真的在该函数体中抛出这个异常。于是再做修改：

```
public double firstMomentAbout(double point)
            throws InvalidBasisException  {
```

```
    if (element.size() == 0)
        throw new InvalidBasisException("no elements");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

8.1.7 让测试通过

现在我们的测试通过了。

8.1.8 消除重复代码

本例中没有任何重复代码。

90

8.1.9 编写一个失败测试用例

接下来我们要写的一段代码是一个计算一个点的二阶矩的方法。实际上该方法只是前面那个计算一阶矩的方法的变种。下面就是驱动我们编写相应代码的测试。注意，这里的期望值不再是原来的-0.5，而是0.5。下面便是为这样一个尚不存在的方法secondMomentAbout所编写的测试：

```
public void testSecondMoment() throws Exception {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);

    assertEquals(0.5, calculator.secondMomentAbout(2.0), TOLERANCE);
}
```

8.1.10 让它通过编译

要想让它通过编译，就得先定义secondMomentAbout。我们可以使用前面定义firstMomentAbout方法时所用的技巧，但看起来这个计算二阶矩的方法的代码实现跟前面计算一阶矩的方法只有一些细微差别。

我们只需将firstMoment中的这行代码：

```
numerator += element - point;
```

改为：

```
numerator += Math.pow(element - point, 2.0);
```

而且，这类函数具有一个一般模式。一般而言，*n*阶统计矩的计算对应于如下的表达式：

```
numerator += Math.pow(element - point, N);
```

根据上面的公式，我们知道，firstMomentAbout的代码之所以是正确的，是因为element-

point其实就等于Math.pow(element - point, 1.0)。

事情进行到这里，我们有几个选择。首先，有了上面的一般公式，便可以编写出一个一般性的、计算N阶矩的方法，该方法的参数为一个“中心”点以及N的值。这样我们便可以将每个对firstMomentAbout(double)的使用替换为对这个一般方法的调用了。以上这些当然是可以做到的，只不过那样的话调用方就得多提供一个N值，而我们又不希望允许客户随意给定N的值。不过，我们似乎想得太多了，暂且把这个问题搁在一边吧，先把手头的事情做完。目前唯一的任务就是让测试代码通过编译。以后如果我们仍觉得需要将这个方法一般化的话，到时再着手也不迟。

要让测试代码通过编译，我们可以将firstMomentAbout复制一份并更改函数名为
91 secondMomentAbout：

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("no elements");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

8.1.11　让测试通过

光是复制代码，测试仍无法通过。既然如此，可以回头将代码修改一下，如下，测试便可以通过了：

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("no elements");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += Math.pow(element - point, 2.0);
    }
    return numerator / elements.size();
}
```

你可能会被我们刚才进行的一番“剪切/复制/粘贴”吓一大跳，但别害怕，接下来就要开始消除重复代码了。虽说我们编写的是全新的代码，但在对付遗留代码时，“剪剪贴贴再改改”还是比较有效的办法。一般来说，当想要往特别糟糕的代码中添加特性时，如果能够将代码放到新的地方，使我们能够直观对照新旧代码的话，就会比较有利于理解。事后再去消除重复代码，从

而可以将新代码更好地安置在类当中；或者我们也可以干脆撤销掉所作的修改，并重新进行，你应该知道，即使撤销了原先所做的修改，仍然还有旧代码可以参照。 92

8.1.12 消除重复

现在，既然两个测试都通过了，那么接着进行下一步：消除重复。具体怎么做呢？

一种做法是将secondMomentAbout的函数体完全提取出来，重新命名为nthMomentAbout，并添加一个参数N，如下所示：

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {
    return nthMomentAbout(point, 2.0);
}

private double nthMomentAbout(double point, double n)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("no elements");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += Math.pow(element - point, n);
    }
    return numerator / elements.size();
}
```

如果现在再运行测试，仍然可以通过。然后我们可以回到firstMomentAbout的定义，将它修改为一个简单的对nthMomentAbout的调用，如下所示：

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {
    return nthMomentAbout(point, 1.0);
}
```

这最后一步，即消除重复，是非常重要的。我们可以通过诸如复制粘贴整块代码这样的方式来快速但粗暴地往既有代码中添加新特性，但如果事后不消除重复代码的话，无异会带来麻烦和维护负担。另一方面，有测试的帮助，我们便可以很容易地消除重复代码。前面的讨论显然已经展示了这点，只不过，我们之所以能够在消除重复代码时得到测试的辅助，全要归功于在一开始便选用了的测试驱动的开发方式。在修改遗留代码的过程中，当我们使用测试驱动开发时，为既有代码编写的那些测试是非常重要的。有了这些测试作后盾，便可以放手去编写新特性的实现代码了，而且最后我们可以妥善安全地把这些新代码安置到其余代码当中。 93

测试驱动开发与遗留代码

测试驱动开发的最有价值的一个方面是它使得我们可以在同一时间只关注于一件事情。要么是在编码，要么是在重构；永远也不会在同一时刻做两件事情。

这一好处对于对付遗留代码的人们来说显得尤其有价值，因为它使我们能够独立地编写新代码。

在编写完一些新代码之后，我们便可以通过重构来消除新旧代码之间的任何重复。

在遗留代码的工作场景中，我们可以将测试驱动开发的算法稍微扩展一点：

(1) 将想要修改的类置于测试之下。
(2) 编写一个失败测试用例。
(3) 让它通过编译。
(4) 让测试通过（在进行这一步的过程中尽量不要改动既有代码）。
(5) 消除重复。
(6) 重复上述步骤。

8.2 差异式编程

测试驱动开发并不仅仅是属于面向对象开发领域的东西。实际上，上一节中的例子其实只不过是一段包裹在类之下的过程式代码。只不过在面向对象系统中我们还有另一个选择：借助于类的继承，我们可以在不直接改动一个类的前提下引入新的特性。在添加完特性之后，我们便可以弄清楚到底想要如何添加新特性。

要做到以上这些，关键的技术就是所谓的“*差异式编程*（programming by difference）”。这是曾在20世纪80年代被讨论和使用得比较多的一项相当古老的技术，到了20世纪90年代，面向对象系统社群中的许多人注意到继承的滥用也会带来相当严重的问题，于是该技术也就逐渐淡出了人们的视线。但实际上，一开始使用继承并不意味着后面一直都得保持那个样子。有了测试的帮助，一旦我们发现继承不再合适，便可以很容易地改成其他设计。

下面这个例子展示了这一技术的工作方式。我们有一个已测的Java类，叫做MailForwarder，该类是一个负责管理邮件列表的Java程序的一部分。MailForwarder类拥有一个名叫getFromAddress的方法，如下所示：

94

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    Address [] from = message.getFrom ();
    if (from != null && from.length > 0)
        return new InternetAddress (from [0].toString ());
    return new InternetAddress (getDefaultFrom());
}
```

该方法的意图是从一则接收到的邮件消息中提取出发件人地址并返回它，以便后面被用来作为转发邮件的发送地址。

该方法只在一处地方被用到，即一个名为forwardMessage的方法中的以下两行代码：

```
MimeMessage forward = new MimeMessage (session);
forward.setFrom (getFromAddress (message));
```

现在，假设我们面临一个新的需求，需要支持匿名邮件列表，那么，该怎么做呢？这类列表中的成员当然还是可以发送邮件的，但它们所发送的邮件的发件人地址则应当基于domain（MessageFowarder类的一个实例变量）的值而被设置为一个特殊的电子邮件地址。在进行修改之前，先要编写一个相应的失败测试用例，如下所示：（expectedMessage变量会预先被设置为MessageFowarder转发的那则消息。）

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new MessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

为了添加这一功能，必须修改MessageForwarder吗？其实并非如此。我们可以从MessageForwarder派生出一个新类AnonymousMessageForwarder，并将测试中原先创建/使用MessageForwarder的地方改为创建/使用AnonymousMessageForwarder，如下所示：

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new AnonymousMessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

完成测试用例的修改之后，便可以着手进行MessageForwarder的子类化了（见图8-1）： 95

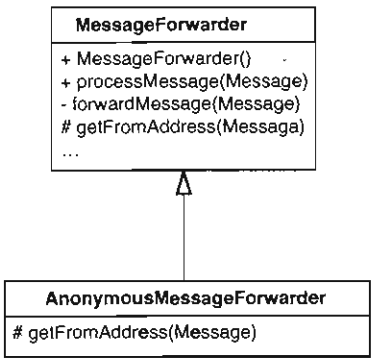


图8-1 子类化MessageForwarder

这里，我们将MessageForwarder中的getFromAddress方法设为受保护而非私有。然后在AnonymousMessageForwarder中重写该方法，如下所示：

```
protected InternetAddress getFromAddress(Message message)
        throws MessagingException {
    String anonymousAddress = "anon-" + listAddress;
    return new InternetAddress(anonymousAddress);
}
```

这给我们带来了什么呢？唔……答案是……问题解决了。但另一个事实是，为了添加一个非常简单的行为而往系统中添加了一个新类。仅仅为了改变一个消息转发类的发件人地址就将这个类整个子类化了，这样做是不是合理呢？从长远来看答案是否定的，但好就好在这样做能够让我们的测试立即通过。而且，一旦测试通过了，以后当我们决定再改动设计时，便可以利用该测试来确保我们的改动不会影响这一新行为。

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new AnonymousMessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

以上这些看上去简直太简单了。但也有问题：如果我们一再使用该技术，并且对我们的设计
96 中的某些重要方面不予关注的话，情况便会很快恶化。为了说明情况会糟到什么地步，考虑另一个修改：我们想要将邮件转发给邮件列表内的收件人，但同样想要通过“密送（bcc）”的方式发给另一些不能出现在正式邮件列表上的收件人。我们可以把这些人称为“编外”收件人。

这听起来简单得不能再简单了：我们只需再对MessageForwarder进行一次子类化，重写相关的邮件处理方法，让它不但转发给正式邮件列表内的收件人，而且发送给那些“编外”收件人即可。如图8-2所示：

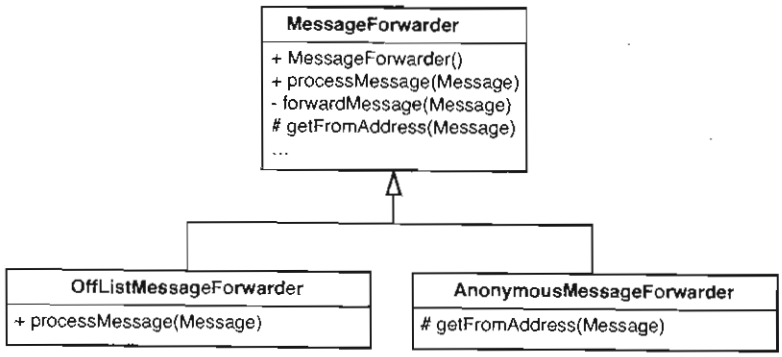


图8-2　为两种不同的差异实施子类化

这的确可行，但有一个问题：如果我们既想要鱼又想要熊掌怎么办呢？也就是说，如果我们需要一个既能发送邮件给不在列表中的收件人，又能进行匿名转发的MessageForwarder，该怎么办呢？

这便是处处使用继承所带来的棘手问题之一了。倘若将不同的特性放入不同的子类中，我们在同一时间便只能展示其中一样特性了。

那么，如何摆脱这个约束？办法之一就是在添加“编外收件人”特性之前做一点重构，以便让该特性能够“清爽”地进入系统。幸运的是，有前面编写的测试作后盾，当我们启用另一种修改策略时，可以利用该测试来检验之前的行为是否被保持下来了。

至于那个匿名转发的功能，其实我们原本可以不用子类化就实现它的。可以选择将匿名转发功能做成一个配置选项。办法之一便是通过修改MessageForwarder的构造函数，让它接受一组属性：

```
Properties configuration = new Properties();
configuration.setProperty("anonymous", "true");
MessageForwarder forwarder = new MessageForwarder(configuration);
```

之后，如何才能让原先的测试通过呢？为此，让我们再来回顾一下前面的测试：

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new AnonymousMessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

97

目前这个测试是通过的。AnonymousMessageForwarder重写了MessageForwarder中的getFromAddress方法。那么，假如我们像下面这样修改MessageForwarder中的getFromAddress方法：

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress = getDefaultFrom();
    if (configuration.getProperty("anonymous").equals("true")) {
        fromAddress = "anon-members@" + domain;
    }
    else {
        Address [] from = message.getFrom ();
        if (from != null && from.length > 0) {
            fromAddress = from [0].toString ();
        }
    }
    return new InternetAddress (fromAddress);
}
```

这样一来，MessageFowarder中就有了一个能够同时处理匿名情况和正常情况的getFromAddress方法了。要验证这一点，可以将AnonymousMessageForwarder中的那个重写版本注释掉（如下），看看测试是否仍然通过。

```
public class AnonymousMessageForwarder extends MessageForwarder
{
/*
    protected InternetAddress getFromAddress(Message message)
            throws MessagingException {
        String anonymousAddress = "anon-" + listAddress;
        return new InternetAddress(anonymousAddress);
    }
*/
}
```

毫无疑问，测试仍然通过。

所以，现在我们不再需要AnonymousMessageForwarder类了，可以将它删除掉。接着我们得找到所有创建AnonymousMessageForwarder对象的地方，将它们全部改为创建MessageForwarder的对象（其构造函数参数为一个属性集合（Properties对象））。

当然，我们还可以利用这一属性集合来添加其他新特性。例如，我们可以加入一个属性用于
98 控制是否启用“编外收件人”功能。

完事儿了吗？还没有。现在的问题是我们把MessageForwarder的getFromAddress方法弄得有点儿乱了，但好在还有测试，所以我们可以很快地进行一次方法提取来让它变得干净点儿。在这之前先来看一下该方法目前的样子：

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress  = getDefaultFrom();
    if (configuration.getProperty("anonymous").equals("true")) {
        fromAddress = "anon-members@" + domain;
    }
    else {
        Address [] from = message.getFrom ();
        if (from != null && from.length > 0)
        fromAddress = from [0].toString ();
    }
    return new InternetAddress (fromAddress);
}
```

做了一些重构之后，变成下面这样：

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress  = getDefaultFrom();
    if (configuration.getProperty("anonymous").equals("true")) {
        from  = getAnonymousFrom();
    }
    else {
        from = getFrom(Message);
    }
    return new InternetAddress (from);
}
```

这样看起来的确是干净了些，但是匿名发送以及“编外收件人”这两个特性现在全都被放到MessageForwarder中了。这岂不是不符合单一职责原则了？或许吧。答案取决于与该职责相关的代码部分到底有多大，以及它们与其他代码“纠缠”得有多厉害。本例中，检测邮件列表是否匿名算不上什么大动作。利用属性集的做法使得后续工作比较顺利。假设后面又多出了许多其他属性，从而使MessageForwarder的代码变得杂乱起来，到处都是条件判断语句，那时该怎么办？一个方案是放弃使用属性集而改用类。设想我们创建一个名叫MailingConfiguration的类，
99 让该类来持有以前的那个属性集，如图8-3所示：

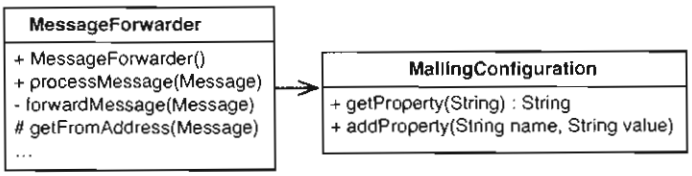


图8-3 委托给MailingConfiguration

看起来不错，但是不是有点小题大做了？MailingConfiguration跟原先那个属性集做的似乎是一模一样的事情。

然而，假如我们将getFromAddress从MessageForwarder中转移到MailingConfiguration中去呢？这样一来MailingConfiguration便可以接受一则邮件消息并自行判断应当返回什么样的发件人地址了。如果其中的配置被设置为匿名转发，那么它便返回匿名发件人地址。如果没有，它便从邮件消息中提取第一个地址并返回它。我们的设计应当像图8-4所示的那样。注意，现在我们再也不需要那些获取/设置属性的方法了。MailingConfiguration现在本身就已支持高阶的功能了。

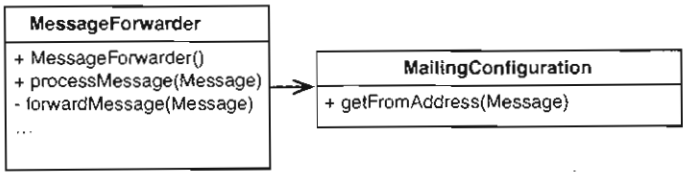


图8-4 将行为移至MailingConfiguration中

我们同样可以开始往MailingConfiguration中添加其他方法。比如说，如果想要实现前面提到的“编外收件人”特性，那么只需往MailingConfiguration中添加一个名为buildRecipientList的方法，并让MessageForwarder使用该方法即可。如图8-5所示：

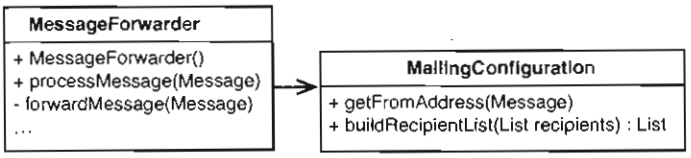


图8-5 往MailingConfiguration中迁移更多的行为

100

在经历了以上这些改动之后，MailingConfiguration这个类名字就不再适合它了。所谓“配置”通常是一件被动的工作，而这个类现在已经能够积极主动地根据MessageFowarder对象的要求为它建立和修改数据了。“MailingList”倒是个蛮合适的名字，如果系统中尚没有其他类起名为MailingList的话。MessageForwarder对象请求MailingList（邮件列表对象）为它计算发件人地址和建立收件人列表。我们可以把决定如何修饰邮件消息归为MailingList（邮件列表）的责任。图8-6展示了重命名之后的状况：

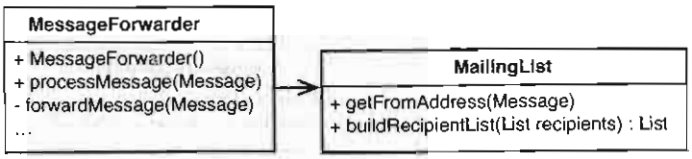


图8-6　MailingConfiguration被重命名为MailingList

有许多重构手法都是相当强大的，但重命名类（rename class）是其中最为强大的一项。它能够改变人们看待代码的方式，并使他们注意到一些以前可能从未考虑过的可能性。

差异式编程是一项有用的技术。它使你能够快速地作出改动，事后还可以再靠测试的帮助来换成更干净的设计。但是，要想正确运用该技术，得注意一些小陷阱：其中之一便是小心别违反Liskov置换原则（LSP）。

Liskov置换原则

在使用继承时我们可能会犯一些细微的错误。考虑如下代码：

```
public class Rectangle
{
    ...
    public Rectangle(int x, int y, int width, int height) { ... }
    public void setWidth(int width) { ... }
    public void setHeight(int height) { ... }
    public int getArea() { ... }
}
```

101

我们有一个Rectangle类。那么，可以由它派生一个名叫Square的子类吗？

```
public class Square extends Rectangle
{
    ...
    public Square(int x, int y, int width) { ... }
    ...
}
```

Square继承了Rectangle 的setWidth跟setHeight这两个方法。那么，考虑下面的代码，在这几行代码执行完毕之后，面积将是多少呢？

```
Rectangle r = new Square();
r.setWidth(3);
r.setHeight(4);
```

如果是12的话，r这个正方形便不能算是个真正的正方形了。于是，我们在Square类中重写Rectangle的setWidth/setHeight方法，以确保其正方形的身份不被改变。例如，我们可以让setWidth和setHeight都去修改Square的宽（Square的面积用宽的平方来计算），但这么一来又会造成违反直觉的结果了：如长宽分别被设为3和4之后人们当然期望面积为12了，然而他们得到的结果却是16。

这是违反Liskov置换原则（LSP）的经典案例之一。子类对象应当能够用于替换代码中出现的它们的父类的对象，不管后者被用在什么地方。如果不能的话，代码中就有可能悄无声息地出现一些错误。

Liskov置换原则意味着一个给定类的客户代码应当能够在毫不知情的情况下使用该类的任何子类对象。不存在任何“机械性”的方法来避免违反该原则。一个类是否符合Liskov置换原则取决于它的客户代码，以及这些客户代码对代码行为或结果的期望。不过，还是存在一些一般规则的：

(1) 尽可能避免重写具体方法[1]。

(2) 倘若真的重写了某个具体方法，那么看看能否在重写方法中调用被重写的那个方法。

然而，前面我们对MessageForwarder进行派生时并没有遵循这些规则。实际上我们所做的恰恰相反。我们在其子类AnonymousMessageForwarder中重写了一个具体方法。这有什么问题吗？

当我们像（在AnonymousMessageForwarder中）重写MessageForwarder的getFromAddress方法那样重写具体方法时，就可能会改变某些使用MessageFowarder对象的代码的意义了。例如，假设应用中到处散布着对MessageForwarder的引用，并且我们将其中一处引用改为引用一个AnonymousMessageForwarder对象，那么此时使用该引用的人可能根本不知道他引用的不再是一个简单的MessageFowarder对象了，他或许还以为该对象会从他处理的邮件消息中获取收件人地址，并在处理邮件消息的时候使用该地址呢。那么，该对象究竟是像刚才描述的这样还 102
是直接使用一个特殊的收件人地址，对于客户代码来说有什么区别吗？答案取决于应用。一般来说，当我们过于频繁地重写具体方法时，代码就容易变得混乱。比如说，某个人可能会注意到代码中使用的是个MessageForwarder引用，于是他翻开MessageFowarder类的定义，并认为被执行的会是MessageFowarder的getFromAddress方法。他可能根本不知道那个引用其实是指向一个AnonymousMessageForwarder对象，因而实际上被调用的是AnonymousMessageForwarder的getFromAddress方法。如果一定要保留继承的话，可以将MessageForwarder做成一个抽象类，其中包含一个抽象的getFromAddress方法，并让子类各自去提供具体的实现。图8-7展示了这样的设计：

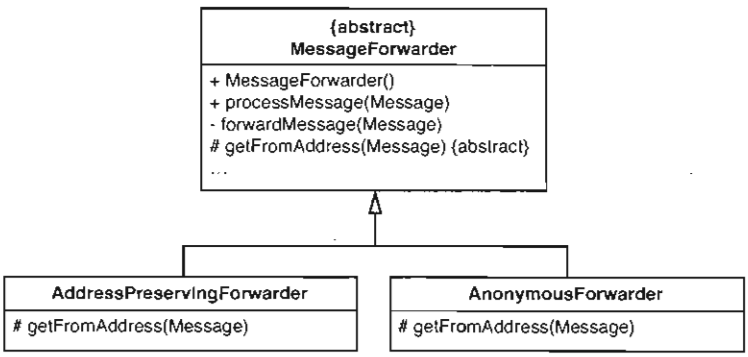


图8-7 规范化继承体系 103

1. 对应于抽象方法；如，接口上的方法便不是具体方法，C++中纯虚函数不是具体方法。——译者注

我把这种继承体系称作规范化的继承体系。在一个规范化的继承体系中，任何类都不会包含同一方法的多个实现。换句话说，任何类都不会重写其父类中的具体方法。当你问“该类是怎样完成工作x的？”时，只需翻开该类的定义看一看便可知道。要么赫然在目，要么就是一个抽象方法，由该类的某个子类来实现。在一个规范化继承体系中，无需担心子类会重写从它们的父类那儿继承来的行为。

那么，是不是任何时候都值得这么去做呢？答案是，偶尔重写一两次具体方法其实是无伤大雅的，只要不违反Liskov置换原则。然而，在我们准备分离出类里面的职责时，最好想一想我们的类从偶尔规范化的形式到整体朝规范化迈进还有多少距离。

差异式编程使我们能够快速往系统中引入变更。在这一过程中，我们可以利用测试来固定住新的行为，便于以后根据需要改用更妥当的设计。有了测试的存在，修改设计就变得非常迅速了。

8.3　小结

本章介绍的技术可以用来往任何能够置于测试之下的代码中添加新特性。近年来关于测试驱动开发的书图不断增多。特别地，我推荐Kent Beck的《测试驱动开发》（Addison-Wesley, 2002），
104 以及Dave Astel的《测试驱动开发——实用指南》（Prentice Hall, 2003）。

第9章 无法将类放入测试用具中

本章将要讨论的是一个困难的问题。如果在测试用具中实例化一个类总是那么容易的话，本书也就会简短得多了。遗憾的是情况并非如此。

下面就是我们会遇到的四种最为常见的问题（其中“该类”代表我们所针对的问题类）：

(1) 无法轻易创建该类的对象。

(2) 当该类位于测试用具中时，测试用具无法轻易通过编译构建。

(3) 我们需要用到的构造函数具有副作用。

(4) 构造函数中有一些要紧的工作，我们需要感知到它们。

本章我们将会看到一系列的例子，这些来自不同语言编程环境下的例子展示了上面提到的这些问题。实际上，每个问题都不止一种解决方案。不过，要熟悉一大堆解依赖技术，学习如何在它们之间进行权衡，以及如何在特定场合下运用的话，通过这些例子来学习是极好的途径。

9.1 令人恼火的参数

当需要在一个遗留系统中作修改时，我通常是以一种轻松乐观的心态开始工作的。我不知道为什么。虽然一直试图让自己尽量变得更现实一些，但总会存在那么一点儿乐观情绪。“嗨，”我对自己（或工作伙伴）说，“这听起来蛮简单的，我们只需把这个类这么处理一下就成了。”然而，虽然嘴上说得简单，当正儿八经打开那个类时却发现……“好吧，看来我们需要在这儿添加一个方法，并修改这儿的另一个方法。当然，我们还得把它放入测试用具。”直到这时我才开始有点怀疑。“唉……看来这个类就连最简单的构造函数也接受3个参数嘛，不过……”我乐观地安慰自己，“也许构造起来并没那么困难。”

让我们来看一个具体的例子，看看到底我的乐观是有道理的还是自我安慰。

在一个计费系统中，我们有一个未测试的Java类：CreditValidator。

```
public class CreditValidator
{
    public CreditValidator(RGHConnection connection,
                           CreditMaster master,
                           String validatorID) {
        ...
    }
```

```
    Certificate validateCustomer(Customer customer)
            throws InvalidCredit {
        ...
    }

    ...
}
```

该类的众多职责之一便是告诉我们某个客户是否有足够的余额。是则返回一个认证，告诉我们该客户的余额到底是多少。否则抛出一个异常。

而我们的任务（如果我们选择接受该任务的话）则是往该类中添加一个新方法。这个方法将被命名为getValidationPercent，其职责是告诉我们在一个CreditValidator对象的生命周期中对validateCustomer()的调用的成功率。

那么，我们该怎么做呢？

如果需要看看能否在测试用具中创建一个对象，最佳做法就是试一试。我们当然可以通过一
106 系列的分析来确定在测试用具中创建某个类的对象容易与否；不过还有一个方法同样轻而易举，即创建一个JUnit测试类，并将下面的代码填到里面然后编译：

```
public void testCreate() {
    CreditValidator validator = new CreditValidator();
}
```

要想知道能否在测试用具中实例化一个类，最佳途径就是试一试。编写一个测试用例并在里面创建该类的对象。编译器会告诉你要令代码工作起来还需要哪些东西。

以上测试是个构造测试。构造测试看起来的确有点怪异。编写这类测试时我通常并不在里面放置任何断言，而只是试着创建对象。最后，当终于能够在测试用具中构造某类的对象时，我通常会删掉该测试，或将它重命名以便能够用来做一些更为实质性的测试。

返回到我们的例子。

在上面那个简单的testCreate方法中，我们在创建CreditValidator对象时并未向它的构造函数提供任何实参，因此编译会报错。错误消息会说CreditValidator没有默认构造函数。于是我们翻开CreditValidator的代码一看，发现其构造函数需要3个参数：一个RGHConnection、一个CreditMaster以及一个密码。而这3个类全都只有一个构造函数。如下所示：

```
public class RGHConnection
{
    public RGHConnection(int port, String Name, String passwd)
            throws IOException {
        ...
    }
}

public class CreditMaster
```

```
{
    public CreditMaster(String filename, boolean isLocal) {
        ...
    }
}
```

RGHConnection在构造时会连接到一个服务器，这个连接被用来从服务器获取必要的信息以检查客户的余额。

另一个类CreditMaster，则提供了一些我们在检查余额的过程中会用到的策略信息。CreditMaster的构造函数会从一个文件中加载相关信息，并把这些信息保存在内存中以备后用。

这么看来，把这个类放入测试用具的确是举手之劳，对不对？别这么快下结论。虽说我们的确可以为它编写出一个测试来，但我们能否忍受这个测试的速度呢？如下所示： 107

```
public void testCreate() throws Exception {
    RGHConnection connection = new RGHConnection(DEFAULT_PORT,
                                                 "admin", "rii8ii9s");
    CreditMaster master = new CreditMaster("crm2.mas", true);
    CreditValidator validator = new CreditValidator(
                                        connection, master, "a");
}
```

看起来，在测试中建立到服务器的RGHConnection并不是个好主意。首先其耗时就比较长，况且服务器也并不总是处于服务状态。相比之下CreditMaster倒不算什么问题了。CreditMaster在创建时会加载一个文件，而这个过程比较快速。况且它所加载的文件还是只读的，这就确保了我们无需担心它被测试程序破坏掉。

真正妨碍我们顺利创建CreditValidator对象的其实是RGHConnection。这是个令人恼火的参数。我们的设想是：若能够创建某种伪造的RGHConnection对象并使CreditValidator相信它是一个真正的RGHConnection的话，就可以避开所有的连接问题了。所以，让我们来看一看RGHConnection所拥有的方法（如图9-1所示）：

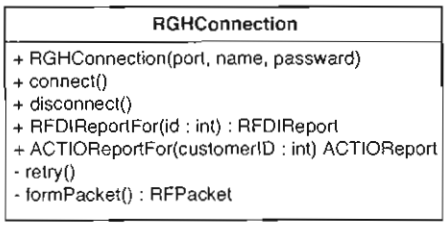


图9-1 RGHConnection

看上去RGHConnection中有一些方法是用来处理与连接相关的任务的：如connect、disconnect以及retry。另外还有一些与业务相关的方法，如RFDIReportFor和ACTIOReportFor。前面曾提过，我们的任务是往CreditValidator中添加一个新方法，用于获知在一个CreditValidator对象的生命周期中它上面的validateCustomer()调用的成功百分比，为此得调用RFDIReportFor来获取我们所需要的一切信息，通常所有这些信息都是来

自服务器的，所以，如果我们想如刚才所言伪造一个RGHConnection的话，就必须想办法让那个伪造的RGHConnection也能提供这些信息。

在这些条件之下，伪造一个RGHConnection的最佳方法是对RGHConnection类应用接口提取。如果你手头有一个支持重构的工具，那么它很可能也会支持接口提取手法。如果没有支持这
108 一手法的工具的话，别担心，自己动手也同样简单。

在对RGHConnection运用接口提取技术之后，我们最终得到了如图9-2所示的结构：

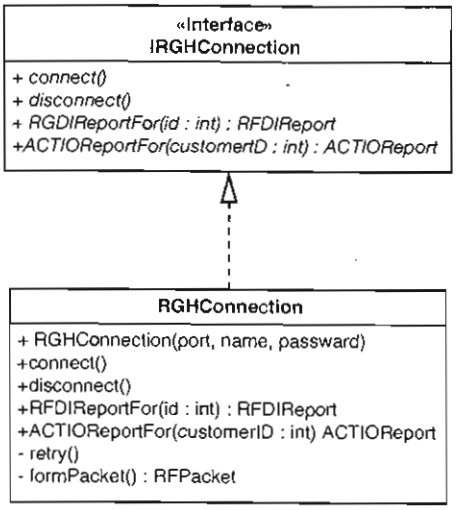


图9-2　接口提取后的RGHConnection

至此，我们便可以创建一个轻便的FakeConnection类，并使它能够提供我们所需的反馈信息，然后将这个伪造的“RGHConnection”用在测试中：

```
public class FakeConnection implements IRGHConnection
{
    public RFDIReport report;

    public void connect() {}
    public void disconnect() {}
    public RFDIReport RFDIReportFor(int id) { return report; }
    public ACTIOReport ACTIOReportFor(int customerID) { return null; }
}
```

有了上面的类，便可以开始编写测试了，如下所示：

```
void testNoSuccess() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection();
    CreditValidator validator = new CreditValidator(
                                        connection, master, "a");
    connection.report  = new RFDIReport(...);
```

```
    Certificate result = validator.validateCustomer(new Customer(...));

    assertEquals(Certificate.VALID, result.getStatus());
}
```

109

FakeConnection类看起来有点古怪：它的方法要么是空的要么就是简单地返回null。这种情形并不常见。更糟的是，它有一个任何人都可以看到并随意设置的公共变量。这样一个类似乎违反了所有的良好准则。但你要看到，实际上并非如此。对于一个用来使得测试可行的类，规则是有所不同的。FakeConnection中的代码并非产品代码。它永远也不属于最终投入运行的应用，而只是为了测试用具而诞生的。

现在，既然我们已经可以创建一个CreditValidator，那么便可以开始编写它的getValidationPercent方法了。在这之前我们先编写其测试，如下所示：

```
void testAllPassed100Percent() throws Exception {
    CreditMaster master = new CreditMaster("crm2.mas", true);
    IRGHConnection connection = new FakeConnection("admin", "rii8ii9s");
    CreditValidator validator = new CreditValidator(
                                        connection, master, "a");

    connection.report  = new RFDIReport(...);
    Certificate result = validator.validateCustomer(new Customer(...));
    assertEquals(100.0, validator.getValidationPercent(), THRESHOLD);
}
```

测试代码vs.产品代码

测试代码并不需要具有产品代码那么高的品质。一般来说，如果将变量设为公有能够使测试的编写更为容易的话，我是不会介意这么做的，尽管这会破坏封装。不过就算是测试代码也应当是干净、易于理解和修改的。

让我们来看一看上面的testNoSuccess和testAllPassed100Percent这两个测试。它们之间有重复代码吗？是的，它们的头三行代码完全一样，应当提取出来放到一个公共方法中，即所属类的setUp()方法。

以上测试检查CreditValidator对象在验证了一个具有足够余额的客户之后其验证成功率是否约为百分之百。

这个测试工作起来没问题，但随着我们真正开始编写getValidationPercent，却发现一些有趣的现象。似乎getValidationPercent并不需要用到CreditMaster，既然这样，当我们在测试用例中创建CreditValidator对象时为什么还要提供一个CreditMaster呢？或许并不一定要这么做。我们可以像这样来创建测试用的CreditValidator对象：

```
CreditValidator validator = new CreditValidator(connection, null, "a");
```

是不是很麻烦？

人们对于上述代码的反应往往很能体现出他们对付的是一个什么样的系统。比如一种反应是这样的："好吧，也就是说他将一个null传递给了这个构造函数——我们在系统中就经常这么做。"

110

这种人很可能对付的是一个相当糟糕的系统。系统中可能到处都是针对null的检查，另外还有许多条件代码用于判断某些引用到底引用的是什么以及能用来做什么。另一种反应则是这样的："这家伙到底怎么了？！竟然在一个系统中把null传来传去？到底有没有知识啊？"呃……如果你属于后一种人（或者至少你读到这里还没有把书狠狠合上并扔回书店的书架上），请听我一句善意的辩解：别忘了我们只是在测试中才这么做。可能发生的最糟糕的事情就是某些代码试图去使用我们传递的null，如果这真的发生的话，Java运行时便会抛出一个异常。又由于测试用具能够捕获测试当中抛出的任何异常，所以最终我们很快就会发现参数是否被使用了。

传Null

在编写测试时，如果你发现某个对象需要的某参数难以构造，便可以考虑传递一个null。这样一来如果这个参数在测试运行的过程中被用到了的话，代码就会抛出一个异常，而测试用具则会捕获这个异常。这时候，如果确实需要传递一个对象而不是null的话，再去构造这个对象并将它作为参数传递也不晚。

传Null手法在某些语言中是一项非常便利的技术。在Java和C#中使用起来没什么问题，其实只要是那些会对运行期使用空引用抛出异常的语言，这项技术都适用。但这也意味着在C和C++中使用这项技术并不是个好主意，除非你知道运行时系统会检测出空指针的使用。如果系统并不具备这样的能力，还是别用为妙，否则最后会发现你的测试神秘崩溃，而这还算是运气好的结果，运气不好的话测试则会悄无声息地出问题，而你拿它们一点办法也没有；它们在运行时会破坏内存，而你则被蒙在鼓里。

当我使用Java语言来工作时，通常一开始编写的测试是像下面这样的，然后根据实际情况需要决定是否把其中的null替换成有血有肉的对象。

```
public void testCreate() {
    CreditValidator validator = new CreditValidator(null, null, "a");
}
```

关键要记住一点：不到万不得已千万别在产品代码中传递null。我知道现在有些库会期望你在某些情况下传递null，这是无法控制的事情，但当编写自己的代码时，请务必记住还有更好的选择。如果忍不住想在产品代码中使用null，那么建议你找出所有返回null或传null的地方，并考虑采用另一种协议。比如，考虑使用空对象模式（Null Object Pattern）。

111

空对象模式

空对象模式用于避免在程序中传递null。例如，假设我们有一个方法，该方法根据调用者提供的ID返回相应雇员对象，那么，如果某个ID是无效雇员ID的话，应当返回什么呢？

```
for(Iterator it = idList.iterator(); it.hasNext(); ) {
        EmployeeID id = (EmployeeID)it.next();
        Employee e = finder.getEmployeeForID(id);
        e.pay();
}
```

有如下几个选择：一个选择是干脆抛出一个异常，这样就无需返回任何东西了，但同时也

就迫使调用方显式地去处理这个异常；另一个选择便是返回null，但这么一来调用方就必须显式检查其返回值是否null。

但其实我们还有第三个选择。回顾上面的示例代码，这段代码真的关心是否存在一个需要支酬的雇员吗？另外它是否必须得关心这个呢？如果我们新添一个叫做NullEmployee的类怎么样？NullEmployee的对象相当于一个既没有名字也没有地址的雇员，当你向他支酬时（调用其pay()方法），什么也不会发生。

空对象模式在这类情况下就可以派上用场了；它们可以免除让调用方进行显式错误检查的烦恼。不过，虽说这个模式不错，但用的时候也得小心。例如，下面这段代码试图计算已被支酬的雇员人数，而事实上这样做是非常糟糕的：

```
int employeesPaid = 0;
for(Iterator it = idList.iterator(); it.hasNext(); ) {
    EmployeeID id = (EmployeeID)it.next();
    Employee e = finder.getEmployeeForID(id);
    e.pay();
    mployeesPaid++;          // bug!
}
```

在上面的循环中，一旦获取到的是空雇员，最终的计算结果就会是错误的。

当调用方无须关心某个操作是否成功时，空对象模式尤其有用。而许多时候我们又都可以通过对设计进行一些巧妙的处理从而使其符合这个前提。

传Null和接口提取（285页）是两种可以用来解决恼人的参数的途径。但有时我们也可以使用另一个方案，如果一个参数类型中的问题依赖并不是被硬编码在其构造函数中的话，就可以使用子类化并重写方法（Subclass and Override Method，314页）来对付该依赖。比如在上面的例子中这一方案就是可行的。如果RGHConnection的构造函数使用了它的connect()方法来建立一个连接的话，我们就可以通过在RGHConnection的一个为测试而造的子类中重写其connect()方法来解开依赖。子类化并重写方法在有些场合下是非常有用的解依赖手段，但在使用的时候我们得注意别把想要测试的行为给篡改了。 112

9.2　隐藏依赖

有些类是具有欺骗性的。比如我们发现它上面有一个构造函数是我们想要使用的，然后试图去调用这个构造函数。结果，砰的一声！我们撞上了一块石头。而这块“石头”最常见的可能就是隐藏依赖；比如构造函数中使用了一些我们在测试用具中根本无法很好地访问到的资源。下面就是一个有关的例子，这是一个设计得很糟糕的C++类，它负责管理一个邮件列表：

```
class mailing_list_dispatcher
{
public:
                    mailing_list_dispatcher ();
    virtual         ~mailing_list_dispatcher;
```

```
    void            send_message(const std::string& message);
    void            add_recipient(const mail_txm_id id,
                                  const mail_address& address);
    ...

private:
    mail_service    *service;
    int             status;
};
```

下面是该类的构造函数的部分代码。它先是在构造函数的初始化列表中new了一个mail_service对象。这是个差劲的编码风格，而且还不止这些。这个构造函数后面又用这个mail_service对象做了很多细节的工作。此外它还使用了一个所谓的“魔数”——12，这里12究竟代表什么意思？

```
mailing_list_dispatcher::mailing_list_dispatcher()
: service(new mail_service), status(MAIL_OKAY)
{
    const int client_type = 12;
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

113

我们可以在测试中创建该类的对象，但或许这么做没什么好处。首先，需要连接到邮件库，并配置邮件系统以便进行注册。而且倘若我们在测试中使用send_message函数，就会真的给某个人发邮件了。所以很难自动地对该功能进行测试，除非我们配置一个特殊的邮箱，然后不断重复登录它，看看邮件消息是否已到达。如果我们要做的是一个整体的系统测试，那么这样做没问题，但如果我们只是想往该类中添加一些经过测试的新功能的话，这么做就未免太小题大做了。我们只是想创建一个用于测试的简单对象，以便添加一些新的功能而已，如何才能达到这一目的呢？

根本问题在于对mail_service的依赖被隐藏在了mailing_list_dispatcher这个构造函数中。如果有办法把那个mail_service对象替换成一个伪对象的话，就可以通过这个伪对象来进行感知，并在修改过程中获取一些反馈信息了。

这里可以采用的一个技术是*参数化构造函数*（Parameterize Constructor，297页）。运用该技术，可以将一个藏在构造函数中的依赖“外在化”，即让它以参数的形式从外面传进来。

下面就是对mailing_list_dispatcher的构造函数运用参数化构造函数的结果：

```
mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
: status(MAIL_OKAY)
{
    const int client_type = 12;
```

```
    service->connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}
```

实际上这里面唯一的区别就在于现在mail_service对象是在外面创建并以参数的形式传递进来的了。看上去这可能算不上多大的改进，但它的确给了我们不可低估的优势。现在便可以使用*接口提取*技术来给mail_service提取一个接口了。该接口的实现之一是一个真正会发送邮件的产品类，而另一个实现则可以是一个“伪造的”类，它的作用只是在测试时感知我们对它所做的动作，让我们确信这些事情都发生了。

利用*参数化构造函数*技术，可以很方便地将构造函数中的依赖外在化，但人们却常常想不到使用它。一个原因是人们往往会认为随着某构造函数中的依赖被外在化，它的所有调用方都得进行相应的改动，以便传递新的参数。但实际上这种想法并不正确。我们可以解决这个问题：首先将构造函数的函数体提取到一个新的方法initialize中。跟大多数的方法提取有所不同，这一提取在没有测试保护的情况下也是相当安全的，因为我们可以在提取时运用*签名保持*（249页）技术。 114

```
void mailing_list_dispatcher::initialize(mail_service *service)
{
    status = MAIL_OKAY;
    const int client_type = 12;
    service.connect();
    if (service->get_status() == MS_AVAILABLE) {
        service->register(this, client_type, MARK_MESSAGES_OFF);
        service->set_param(client_type, ML_NOBOUNCE | ML_REPEATOFF);
    }
    else
        status = MAIL_OFFLINE;
    ...
}

mailing_list_dispatcher::mailing_list_dispatcher(mail_service *service)
{
    initialize(service);
}
```

现在，我们可以额外再提供一个跟原先的签名一模一样的构造函数（如下所示）。于是，在测试时可以调用具有mail_service参数的那个构造函数，而其他客户代码则调用那个签名跟原先一样的构造函数，这么一来客户代码便可以维持原样，无需关心我们作了哪些改动。

```
mailing_list_dispatcher::mailing_list_dispatcher()
{
    initialize(new mail_service);
}
```

这种重构在像C#和Java这样的语言中甚至更为容易，因为这些语言支持在一个构造函数中调用另一个构造函数。

例如，假设我们要在C#中完成类似的工作，结果代码可能看起来像这样：

```
public class MailingListDispatcher
{
    public MailingListDispatcher()
    : this(new MailService())
    {}

    public MailingListDispatcher(MailService service) {
        ...
    }
}
```

有多种技术可以用来克服隐藏在构造函数中的依赖。通常我们可以使用*提取并重写获取方法*（Extract and Override Getter，278页）、*提取并重写工厂方法*（Extract and Override Factory Method，276页）以及*替换实例变量*（Supersede Instance Variable，317页），但我比较倾向于尽可能地使用*参数化构造函数*。当一个构造函数在它的函数体中创建了一个对象，并且该对象本身并没有任何构造性依赖时，运用参数化构造函数就比较轻松了。

115

9.3　构造块

*参数化构造函数*在解开构造函数中隐藏的依赖方面是一项易用的技术，而且它往往也是我第一个诉诸的技术。然而遗憾的是，它并非总是最佳选择。如果一个构造函数中创建了大量的对象，或者访问了大量的全局变量，采用这一技术可能最终会导致一个非常长的参数列表。最糟糕的是，某个构造函数可能会先创建一些对象，然后使用它们来创建另一些对象：

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel  = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }
    ...
}
```

如果想要通过cursor变量进行感知的话，我们就遇到麻烦了。cursor对象被嵌在一系列的对象创建中。我们可以尝试将所有用于创建cursor对象的代码都移至类外面，让客户代码去创建cursor并将它作为参数传递给该类。但这么做没有测试保护的话不是很安全，而且还可能会

给该类的客户代码带来不小的负担。

如果有一个能够安全地提取方法的重构工具，我们就可以对构造函数中的代码运用*提取并重写工厂方法*（276页）手法了，然而这一做法也并非在所有语言中都可行。在Java和C#中没问题，但C++中就不同了，因为在C++中，构造函数中对虚函数的调用是不会被决议到派生类的相应虚函数上去的。况且一般来讲这也并不是个好主意，因为派生类中的虚函数往往会认为它们可以使用基类的成员变量，所以倘若在基类的构造函数完全结束之前，其派生类中的虚函数被调用起来并且后者试图去访问基类中的某个成员变量的话，很可能它访问到的就会是一个未初始化的变量。 116

还有一个选择，便是采用*替换实例变量*（317页）手法。我们为这个类编写一个设置方法，该方法允许我们在对象构造完毕之后替换掉其中的某个成员变量。

```
class WatercolorPane
{
public:
    WatercolorPane(Form *border, WashBrush *brush, Pattern *backdrop)
    {
        ...
        anteriorPanel  = new Panel(border);
        anteriorPanel->setBorderColor(brush->getForeColor());
        backgroundPanel = new Panel(border, backdrop);

        cursor = new FocusWidget(brush, backgroundPanel);
        ...
    }

    void supersedeCursor(FocusWidget *newCursor)
    {
        delete cursor;
        cursor = newCursor;
    }
}
```

在C++中进行这类重构时得非常小心。在替换掉一个对象之前，得先将旧的对象处理掉。通常这就意味着使用删除操作符来调用其析构函数并释放其内存。这么做的时候我们得清楚该析构函数会做些什么事情以及它是否会销毁某些当初传递给该对象的构造函数的东西[1]。如果我们在清理内存的时候不小心的话，就可能会引入一些难以察觉的bug。

而在大多数其他语言中，*替换实例变量*手法则是相当直观的。下面就是上例在Java中的版本。我们无需操心cursor原先指向的对象，垃圾收集器会负责将其回收。但我们得非常注意的是别在产品代码中使用这一技术，因为如果被替换的对象管理了某些资源的话，替换它们就可能会导致一些严重的资源问题。

```
void supersedeCursor(FocusWidget newCursor) {
    cursor = newCursor;
}
```

1. 比如上例中的brush和backgroundPanel。——译者注

现在，既然我们已经有了一个替换方法（supersedeCursor），便可以试着在类的外部创建一个FocusWidget，然后在该对象构造完毕之后通过这个替换方法把它传递进去。又因为我们需要进行感知，所以可以对FocusWidget使用接口提取或实现提取，然后创建其伪对象传递进
117 去。显然，这个伪对象的创建比起前面的在构造函数中创建FocusWidget的过程要简单多了。

```
TEST(renderBorder, WatercolorPane)
{
    ...
    TestingFocusWidget *widget = new TestingFocusWidget;
    WatercolorPane pane(form, border, backdrop);

    pane.supersedeCursor(widget);

    LONGS_EQUAL(0, pane.getComponentCount());
}
```

除非万不得已，否则我是不愿使用替换实例变量手法的。这一做法很可能带来一些资源管理方面的问题。不过在C++中的有些场合下我还是会使用它的。由于我常常想要使用提取并重写工厂方法，而在C++中这一技术又是无法用在构造函数中的，所以有时我只能改用替换实例变量作为替代方案了。

9.4　恼人的全局依赖

多年来业界人们一直抱怨市面上没有更多的可复用组件。随着时间的推移，这种情况也在逐渐好转，市面上已经出现了大量商业的和开源的框架，但总体上，它们中的许多其实并不是在被我们使用，而是我们的代码在被它们“使用”着。框架通常会管理一个应用的生命周期，而我们则是往框架的空档之中填塞代码。这一点在各种框架中都能看到，从ASP.NET到Java Struts。甚至xUnit这样的框架也如此，我们编写测试类，而xUnit则负责调用这些类并显示结果。

框架解决了许多问题，而且它们也的确能在项目开始时助我们一臂之力，但这并非人们所真正期盼的那种复用。旧风格的复用是这样的：我们发现一些类或一组类是我们想要用在项目中的，于是便这么做了：将它们添加到项目中，并使用它们，就这么简单。如果这种情形能够成为日常程序的话倒是不错，但坦白的说，通常我们连把一个类从它所在的项目中挖出来并在测试用具中单独编译它都无法轻松做到，既然如此，也就根本不用奢望这种复用了。

有各种各样的依赖可能会令我们难以在测试框架中创建并使用一个类，其中最难对付的依赖之一便是全局变量的使用。简单的情况下，我们可以使用参数化构造函数（297页）、参数化方法
118 （301页）以及提取并重写调用（275页）这三种技术来对付这些依赖，但有些时候对于全局变量的依赖是如此的广泛，以致于从根本上解决问题倒成了较容易的途径。下面的例子就展示了这种情况，这是一个Java类，其作用是记录政府机构颁发的建筑许可。下面就是其中一个主要的类：

```
public class Facility
{
    private Permit basePermit;
```

```
    public Facility(int facilityCode, String owner, PermitNotice notice)
                throws PermitViolation {

        Permit associatedPermit =
            PermitRepository.getInstance().findAssociatedPermit(notice);

        if (associatedPermit.isValid() && !notice.isValid()) {
            basePermit = associatedPermit;
        }
        else if (!notice.isValid()) {
            Permit permit = new Permit(notice);
            permit.validate();
            basePermit = permit;
        }
        else
            throw new PermitViolation(permit);
    }
    ...
}
```

我们的目的是要在测试用具中创建一个Facility，因此作如下尝试：

```
public void testCreate() {
    PermitNotice notice = new PermitNotice(0, "a");
    Facility facility = new Facility(Facility.RESIDENCE, "b", notice);
}
```

以上测试代码能够通过编译，但我们开始编写其他测试时便会发现一个问题：Facility的构造函数使用了一个名为PermitRepository的类，为了将我们的测试条件设置妥当，必须使用一个或一组许可证（permit）来初始化这个全局的PermitRepository。下面就是Facility构造函数中的相关语句：

```
Permit associatedPermit =
            PermitRepository.getInstance().findAssociatedPermit(notice);
```

我们固然可以通过参数化构造函数来解决这儿的问题，但整个应用中并非只有此处有这个问题，另外还有10个类中也有类似的代码。构造函数中有，一般方法中有，静态方法中也有。可想而知，不花上一大把时间是没法处理代码基中的这些问题的。 119

如果你学过设计模式，可能会发现这里用的正是单件模式：PermitRepository的getInstance方法是一个静态方法，其职责是返回该应用中有且仅允许有的那唯一一个PermitRepository实例。持有该实例的变量也是静态的，是PermitRepository的一个静态成员变量。

在Java当中，单件模式是人们用于实现全局变量的机制之一。通常，全局变量不是个好做法，原因有好几个方面，其一就是不透明性。当我们查看一段代码时，能够知道这段代码会产生什么样的影响是件不错的事情。例如，在Java中，当我们想要理解某段代码会产生什么样的影响时，只需查看几个地方：

```
Account example = new Account();
example.deposit(1);
```

```
int balance = example.getBalance();
```

对于上面这段代码，我们知道一个Account对象可能会影响我们传递给它的构造函数的参数，但在上面的代码中并没有传递任何参数给它的构造函数。Account对象同样也可能会影响我们传递给它的方法的实参对象，不过在这里我们并没有传递任何可被改变的东西，只不过是一个整型数。最后，我们将getBalance的返回值赋给了一个变量，这其实也应当就是这几行语句所影响到的唯一的变量了。

然而，一旦其中涉及全局变量的使用，情况就完全不同了。对于像Account这样的类，光从它的使用代码上可能怎么也看不出来它在背后是否访问或修改了在程序的其他地方声明的变量。毫无疑问，这使得我们对程序的理解变得更困难了。

测试过程中一个麻烦的部分就是我们得找出哪些全局变量正在被我们的类使用，并根据测试需要将它们设置到适当的状态。而且，如果对于不同的测试，设置也不同的话，每个测试开始之前就都得做一番设置工作。这个过程相当烦人，虽说我在好多个系统上都做过这类事情，但还是觉得相当枯燥乏味。

回到我们原先讨论的线路上来。

PermitRepository是一个单件。而正因为它是个单件，所以“仿造”它变得尤其困难。单件模式的核心理念便是使人们无法在应用中创建一个以上单件类的实例。这在产品代码中或许是件好事，然而到了测试中可能就成了一场灾难：一套测试中的每个测试在某种程度上都应当被看成是一个小型的应用，互相之间应当完全隔离开来。因此，要在测试用具中运行一段涉及了单件
120 的代码，就得放松单件性约束。下面就来看看具体是怎么做的。

第一步就是为单件类添加一个新的静态方法。该方法允许我们替换掉该单件类中的那个静态实例。我们将这个方法命名为setTestingInstance，如下所示：

```
public class PermitRepository
{
    private static PermitRepository instance = null;

    private PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice) {
        ...
    }
```

```
    ...
}
```

有了这个设置方法，便可以创建一个PermitRepository测试用例并设置它。从而在测试的setUp()方法中，可以编写如下的代码：

```
public void setUp() {
    PermitRepository repository = new PermitRepository();
    ...
    // add permits to the repository here
    ...
    PermitRepository.setTestingInstance(repository);
}
```

121

引入静态设置方法并非解决这个问题的唯一途径。还有另一个方法：我们可以为单件类添加一个resetForTesting()方法，如下所示：

```
public class PermitRepository
{
    ...
    public void resetForTesting() {
        instance = null;
    }
    ...
}
```

如果我们在测试的setUp()中调用这个方法（另外在tearDown()中调用这个方法也是个好主意），就等于为每个测试都创建了一个全新的单件，因为单件类会为每个测试重新初始化它自身。如果单件类上的公用方法允许你在测试中根据需要任意设置单件的状态，那么这一策略就是行之有效的。而如果单件类并不具有这些公用方法，或使用了一些影响其状态的外部资源的话，引入静态设置方法（setter）就是个较好的选择了。你可以子类化单件类并重写其方法，从而解开依赖，然后往子类中添加一些用于恰当设置其状态的方法。

这样就可以了吗？还没有。人们在使用单件模式时往往会将单件类的构造函数设为私有，这样做是有充分理由的，因为这是令外部代码无法创建该类的其他实例的最明明白白的办法。

到这里，我们发现在两个设计目标之间出现了一个冲突。一方面，我们想要确保在系统中只有一个PermitRepository的实例，而另一方面我们又想要系统中的类能够被单独测试。两者可以兼得吗？

让我们作一点回顾：为什么我们会希望某类的实例在系统中是唯一的呢？答案因系统不同而有所不同，但仍还是有一些最为常见的原因的：

(1) **我们建模的是现实世界，而在现实世界中这种东西只有一个。**比如一些硬件控制系统就是这样的。人们为它们需要控制的每种电路板都创建一个类，如果每种电路板都只有一块的话，这些类就应该是单件类。对于数据库也是这样。前面提到的政府机构中只有唯一一组许可证，因此提供对它们的访问的对象应当是个单件。

(2) **创建某个类的两个（乃至多个）对象可能会导致严重的问题。**同样，这种情况在硬件控制领域也经常发生。假设我们一不小心创建了两个核控制棒的控制器对象，并且同一个程序中的两个不同的部分在互相不知道的情况下（通过这两个控制器对象）操纵同一个控制棒，想想看有多可怕。

122

(3) **创建某个类的两个（乃至多个）对象可能会使用过多的资源。**这种情况也经常出现。这里所说的资源既可以是物理资源，如磁盘空间或内存；也可以是一些抽象的资源，如软件许可证的数目。

以上便是人们要求实例唯一性的主要原因，但并不是他们使用单件模式的主要原因。实际上，人们常常会为了创建全局变量而使用单件模式，他们觉得每次都得将变量传递到需要的地方实在是太麻烦了。

如果是出于后一个原因而使用单件的，那么其实并没有什么理由一定要保持其单件性。我们完全可以将构造函数改设为受保护的、公用的或者包作用域的，同时系统仍是一个可测试的良好系统。而且，就算是出于其他原因而使用单件，仍然值得去尝试一下这一方案。如果需要的话我们可以引入其他的保护措施。例如可以在构建系统中加入一个检查，通过搜索所有的源文件来确保setTestingInstance没有被非测试代码调用。运行期检测也能实现同样的目的：如果setTestingInstance在运行期被调用，我们便可以发出一个警告或挂起系统，等待人为操作介入。事实上，在许多“前面向对象”语言中是无法实施单件性强制的，但人们仍然做出了许多安全的系统。说到底，关键还是要看你的设计和编码是否可靠。

如果打破单件性并不会带来严重问题的话，我们就可以依赖于一个团队守则。例如，团队中的每个人都应当清楚，某应用程序中的某数据库只能有一个实例，不能有第二个。

按照这一说法，要想放松PermitRepository的单件性约束，可以将其构造函数设为公用的。只要PermitRepository中的公用方法允许我们配置出一个测试用的PermitRepository，这么做就是行之有效的。例如，倘若PermitRepository上有一个名为addPermit的方法可以用来将这个许可证仓库用一堆测试用的许可证填满的话，我们就只需简单地用一组测试用许可证来把它填充，然后在测试中使用这个许可证仓库对象即可。不过，有时候我们可能没有这个便利条件，或者，更糟的是，单件可能做了一些我们决不想在测试用具中见到的事情，例如在幕后与一个数据库进行通信。这时可以采用*子类化并重写方法*（314页）技术，创建一个派生类来让测试更为容易。

下面就是来自我们的许可证系统中的一个例子。PermitRepository中除了一些使其成为单件类的方法和变量之外，还有一个名为findAssociatedPermit的方法：

```
public class PermitRepository
{
    ...
    public Permit findAssociatedPermit(PermitNotice notice) {
123      // open permit database
        ...

        // select using values in notice
        ...
```

```
        // verify we have only one matching permit, if not report error
        ...

        // return the matching permit
        ...
        }
}
```

要想避免跟数据库通信，可以如下子类化PermitRepository：

```
public class TestingPermitRepository extends PermitRepository
{
    private Map permits = new HashMap();

    public void addAssociatedPermit(PermitNotice notice, permit) {
        permits.put(notice, permit);
    }

    public Permit findAssociatedPermit(PermitNotice notice) {
        return (Permit)permits.get(notice);
    }
}
```

这么一来便可以保留住部分的单件性：由于我们使用的是PermitRepository的一个子类而不是PermitRepository本身，所以可以将PermitRepository的构造函数设为受保护的，而不用设成公用的。这便阻止了客户代码创建PermitRepository对象，同时又允许派生出一个PermitRepository的子类：

```
public class PermitRepository
{
    private static PermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(PermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static PermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice)
    {
        ...
    }
    ...
}
```

124

许多时候我们都可以像这样通过子类化并重写方法来设置一个“伪造的”单件。其他时候，依赖是如此的广泛以至于更简单的做法是对单件使用接口提取并将应用中所有相关的引用改为提取出的接口的引用。这么做可能工作量很大，不过我们可以利用依靠编译器（lean on the compiler，251页）手法来进行修改。以下便是对PermitRepository进行接口提取后的样子：

```
public class PermitRepository implements IPermitRepository
{
    private static IPermitRepository instance = null;

    protected PermitRepository() {}

    public static void setTestingInstance(IPermitRepository newInstance)
    {
        instance = newInstance;
    }

    public static IPermitRepository getInstance()
    {
        if (instance == null) {
            instance = new PermitRepository();
        }
        return instance;
    }

    public Permit findAssociatedPermit(PermitNotice notice)
    {
        ...
    }
    ...
}
```

125

IPermitRepository接口将包含PermitRepository的所有公用非静态方法：

```
public interface IPermitRepository
{
    Permit findAssociatedPermit(PermitNotice notice);
    ...
}
```

如果你使用的语言有重构工具支持，则或许可以利用该工具自动进行接口提取。而如果你的语言没有重构工具支持的话，则使用实现提取可能是较容易的办法。

以上整个重构过程被称为引入静态设置方法（292页）。即便是在面对广泛的全局依赖的情况下，我们也可以运用该技术来将测试安置妥当。然而遗憾的是，它对全局依赖的状况并无多大改善。若想要解决全局依赖问题，可以使用参数化方法（301页）和参数化构造函数（297页）。这两种技术相当于将一个全局变量变为方法中的局部变量或对象中的成员变量。参数化方法的缺点在于最后的类当中会出现很多额外的方法，影响人们的理解。而参数化构造函数的缺点则在于那些原先使用全局变量的对象都新添了一个成员变量。这个成员变量在该对象构造的时候通过其构造函数传递进来，因而创建这个对象的客户代码必须要能够访问到那个全局实例才行。如果太多

的对象需要这个额外的成员变量，就可能会极大影响整个应用程序的内存使用量，但如果这种情况真的出现的话通常就意味着存在其他的设计问题。

让我们来看看最糟糕的情况。考虑这样一个应用：它里面有几百个类，在运行期会创建数千个对象，其中每个对象都需要访问数据库。甚至无需看这个应用一眼，我们大脑里就会首先闪现出这样一个问题：为什么这么多类都要访问数据库。如果该系统所做的不仅是访问数据库，那么应当能够将做其他事情的那些代码分解出来，如此一来，一些类负责做其他事情，而另一些类则负责数据的存储和获取。当我们努力将应用程序中的各种职责互相分离开来，依赖变得局部化了；我们或许也就不需要让每个对象都自己去访问数据库了。有些对象内的数据来自数据库；另一些对象则对它们的构造函数接受的数据进行计算。

作为一个练习，你可以在一个大型应用中选取一个全局变量，并在整个项目中搜索对它的使用。大多数情况下全局变量都是全局可访问的，但它们其实并没有被全局使用，而是被用在相对较少的一些地方。设想，如果它们不是全局变量，该如何将它们传递给那些需要它们的对象呢？应当如何来重构我们的应用？我们可以从一组组类当中分离出某些职责从而缩小全局变量的作用域吗？ 126

如果你发现一个全局变量真的被到处使用，这便意味着你的代码没有进行任何层次化设计。请阅读第15章以及第17章。

9.5　可怕的包含依赖

C++是我使用的第一门面向对象语言，我得承认，对于自己能够掌握C++中的许多细节和复杂性而感到非常自豪。C++当初能够成为业界主流是因为它对当时许多麻烦的问题提供了完全实际的解决方案。机器太慢？没关系，在这门语言中所有东西都是可选的。如果你只使用其中的C特性，你就能够得到跟C语言一样的效率。你的团队还不会使用一门面向对象语言？没关系，这里是一个C++编译器，你可以先用C++的C子集编码，然后再慢慢学习面向对象。

尽管C++确实非常流行了一段时间，但最终还是落在了Java以及一些新兴的语言后面。C++在保持对C的向后兼容方面的确有些优势，但对于一门语言来说，易用性要重要得多。用C++开发的团队们一次又一次地认识到，这门语言在“默认”情况下可维护性并不理想，他们必须得走得更远一点才能使系统敏捷且易于修改。

C++从C那儿继承了一些特性，尤为严重的问题便是C语言使程序的一部分“知道”另一部分。在Java和C#中，如果某个文件中的类需要使用另一个文件中的类，我们可以使用import或using语句。编译器会去寻找相应的类，并检查它是否已被编译。如果没有则对它进行编译。如果已被编译，则编译器从该类编译后的文件中读出一小段摘要信息，只要足够确认原先那个类所试图使用的方法在这个类身上都存在即可。

然而，C++编译器通常并没有这项优化。在C++中，如果一个类需要知道另一个类，则后者的声明（位于另一个文件中）就会从文本上被一字不差地包含到欲使用它的文件中。这可能会是一个慢得多的过程。编译器每次见到该声明时都得重新解析它，并建立其内部表示。更糟的是，

这种包含机制一不小心就会被误用。一个文件包含另一个文件，而后者再包含另一个文件，如此一层层包含下去。如果你在项目中对此未加小心的话，搞不好就会发现一些小的文件最终因一层层的包含而实际上包含上万行的代码。人们想知道为什么项目的构建要花上那么长的时间，但由于系统中到处都是文件包含，所以，对任何一个特定的文件，要想弄清它为何需要花那么长时间
127 编译都不是件易事。

读到这里你可能会觉得我现在已经不再喜欢C++了，但实际上并非如此。C++是一门重要的语言，而且目前C++的现存代码量非常巨大，只不过要想驾驭好这门语言的确得花更多的心思。

在遗留代码中，要想将一个C++类放入测试用具可能会很难。我们面对的最直接的挑战之一便是头文件依赖。要在测试用具中创建一个类需要哪些头文件呢？

下面便是一个庞大的C++类Scheduler的部分声明。它含有200多个成员函数，不过我只在下面列出了其中的5个。除了体积庞大之外，这个类还跟其他许多类之间有着非常紧密的、千丝万缕的依赖。问题是：我们怎样才能在测试中创建Scheduler对象呢？

```
#ifndef SCHEDULER_H
#define SCHEDULER_H

#include "Meeting.h"
#include "MailDaemon.h"
...
#include "SchedulerDisplay.h"
#include "DayTime.h"

class Scheduler
{
public:
        Scheduler(const string& owner);
        ~Scheduler();

    void addEvent(Event *event);
    bool hasEvents(Date date);
    bool performConsistencyCheck(string& message);
    ...
};

#endif
```

Scheduler用到的类，包括Meetings、MailDaemons、Events、SchedulerDisplays、Dates等。如果我们想要为Schedulers创建一个测试，最简单的做法便是在同一目录下新建一个名为SchedulerTests的文件，然后在里面编写测试。那么，为什么要在同一目录下呢？因为在有预处理的情况下这样做通常更容易。如果你的项目不使用目录路径来统一文件包含的方式，
128 那么在其他目录下创建测试的话就可能会很麻烦。

```
#include "TestHarness.h"
#include "Scheduler.h"

TEST(create,Scheduler)
{
```

```
    Scheduler scheduler("fred");
}
```

如果我们创建一个文件，并将上面的那行对象声明放入测试的话，就可能会遇到包含问题了：为了编译Scheduler，我们得确保编译器和连接器知道Scheduler所需要用到的一切东西，以及后者所需要用到的一切东西，依此类推。幸运的是，构建系统给了我们一大堆出错信息，无比详尽地告诉了我们有关情况。

在简单的情况下，Scheduler.h文件包含了创建Scheduler对象所需的所有东西，但有时候头文件并不能包含得面面俱到，我们得另外再包含一些文件才能创建并使用该类的对象。

我们可以简单地将Scheduler类的源文件中的所有#include语句一股脑儿拷贝到我们的测试文件中，但实际上可能并不需要它们中的全部。最好的办法是一个一个的添加这些#include语句，看看我们是否真的需要那些依赖。

在理想情况下，最简单的做法就是不断包含我们所需的文件，直到不出现任何构建错误为止，不过这种做法可能会把我们推进一种混乱的情况当中。如果依赖传递的链条很长，则我们可能最终会发现包含了许多我们其实根本不需要的东西。而且，就算依赖链并不长，最后也可能依赖于一些在测试用具中很难对付的东西。如本例中的SchedulerDisplay类就是这样一个依赖。这在上面给出的代码中没有显示出来，但Scheduler的构造函数的确访问了它。我们可以像下面这样来对付这个依赖：

```
#include "TestHarness.h"
#include "Scheduler.h"

void SchedulerDisplay::displayEntry(const string& entyDescription)
{
}

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
}
```

这里我们为SchedulerDisplay::displayEntry编写了一个替代定义。不过遗憾的是，如果这么做，就需要对该文件中的测试用例单独进行编译/构建了，因为SchedulerDisplay中的每个方法只能有唯一定义。所以，我们的Scheduler测试必须构建为一个单独的程序。 129

幸运的是，像上面这样编写的“伪定义”是可以复用的，我们只需将它们从测试文件中剪切出来放到一个单独的头文件当中，这样一来我们便可以在多个源文件当中通过#include该头文件来使用这个“伪定义”了。如下所示：

```
#include "TestHarness.h"
#include "Scheduler.h"
#include "Fakes.h"

TEST(create,Scheduler)
{
    Scheduler scheduler("fred");
```

```
}
```

熟练了之后，像这样使一个C++类能够在测试用具中被实例化就会变得相当容易和机械化。不过，这种做法有几个非常严重的缺点。首先，我们得为这个测试单独构建程序，另外我们其实也并没有在语言层面将依赖解开，因此代码并没有变得更清晰。更糟的是，只要我们的测试还在，我们放在测试文件中的重复定义（本例中即SchedulerDisplay::displayEntry）就必须被保留。

这一技术适合留给类非常巨大且依赖非常严重的时候。这不是一门应该经常使用的技术，也不是一门能轻松使用的技术。如果那个类随着时间的推移会被逐渐分解为一大堆小类，那么为一个类创建一个单独的测试程序也许是有用的。它可以作为一系列重构的测试点。随着时间的推移，你会提取出更多的类，并将这些类纳入测试，从而这些单独的测试程序也就可以随之“退休”了。

9.6　“洋葱”参数

我喜欢简单的构造函数。想想看，当你想要创建一个类时，只要敲进一个构造函数调用，立即就得到了一个可用的对象，这是一件多么美好的事情。但很多时候，创建对象可能会较难。每个对象都得被设置到一个适当的状态，一个使它准备投入后续工作的状态。许多时候这便意味着我们得向它提供本身已被设置妥当的对象，而后者可能又需要另一些对象来设置，这样一来我们
130 为了创建一个对象可能先要进行一连串的“创建—设置—创建”工作，最后才能得到一组对象作为待测试类的构造函数的参数。“对象里面包着对象”，听起来就跟洋葱似的，一层一层的。下面就是这类问题的一个例子：

下面这个类的作用是显示一个SchedulingTask：

```
public class SchedulingTaskPane extends SchedulerPane
{
    public SchedulingTaskPane(SchedulingTask task) {
        ...
    }
}
```

为了创建这个类的对象，我们需要传一个SchedulingTask对象给它的构造函数，而为了创建SchedulingTask对象，我们又不得不使用它那唯一的一个构造函数，如下所示：

```
public class SchedulingTask extends SerialTask
{
    public SchedulingTask(Scheduler scheduler, MeetingResolver resolver)
    {
        ...
    }
}
```

然后，如果我们接着又发现还需要另一些对象来作为创建Schedulers和MeetingResolver对象的参数的话，可就真要抓狂了。唯一能够使我们免于完全绝望的一点就是这个过程总是有尽头的，最终总会有对象不再需要其他对象而创建。要是这个过程永无止境的话，系统岂不是永远也

编译不完了？！

应付这种情形的办法是仔细考察我们究竟想要做什么。是的，我们得编写测试，但对于那些传给构造函数的参数，我们真的需要它们吗？如果并不需要的话，就可以使用传Null手法了。而如果我们需要的只是一些基本的行为，则可以使用接口提取或实现提取。

遗憾的是，SchedulingTask是从一个名为SerialTask的类派生来的，SchedulingTask所做的仅仅是重写SerialTask中的一些受保护的方法。所有的公用方法都来自于SerialTask。我们可以对SchedulingTask使用接口提取吗？或者说，我们必须也对SerialTask使用接口提取吗？在Java中我们不必这么做。我们可以为SchedulingTask创建一个包含了来自SerialTask的方法的接口。

提取后的结果如图9-3所示：

131

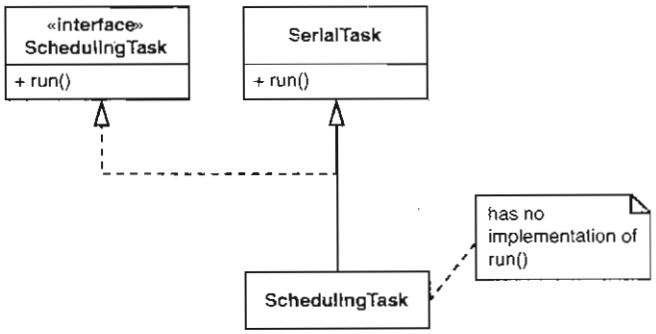


图9-3 SchedulingTask

就这个例子而言，应该说我们是比较幸运的，因为我们使用的是Java而不是C++。如果是C++就不能这样来处理了。因为C++并不支持像Java那样的接口。C++中的接口通常是由只包含纯虚函数的抽象类来实现的。如果把上面这个例子移植到C++下面，SchedulingTask就应该变成一个抽象类，因为它从ISchedulingTask那里继承了一个纯虚函数。所以要想创建SchedulingTask的对象，就必须为它的纯虚成员函数run()提供一个定义体，该定义体负责把任务转发给SerialTask的run()去做。幸运的是这很容易做到。下面就是代码：

```
class SerialTask
{
public:
    virtual void run();
    ...
};

class ISchedulingTask
{
public:
    virtual void run() = 0;
    ...
};
```

```
class SchedulingTask : public SerialTask, public ISchedulingTask
{
public:
    virtual void run() { SerialTask::run(); }
};
```

对于一门语言来说，只要能用它来创建接口，或者类似接口行为的类，我们就可以系统地使
132 用它们来进行解依赖。

9.7 化名参数

当遇到构造函数参数问题时，通常可以借助于接口提取或实现提取技术来克服。但有时候这两种做法却是不实际的。让我们来看一看上节提到的关于建筑许可系统中的另一个类：

```
public class IndustrialFacility extends Facility
{
    Permit basePermit;

    public IndustrialFacility(int facilityCode, String owner,
                    OriginationPermit permit) throws PermitViolation {

        Permit associatedPermit =
            PermitRepository.GetInstance()
                            .findAssociatedFromOrigination(permit);

        if (associatedPermit.isValid() && !permit.isValid()) {
            basePermit = associatedPermit;
        }
        else if (!permit.isValid()) {
            permit.validate();
            basePermit = permit;
        }
        else
            throw new PermitViolation(permit);
    }
    ...
}
```

我们想要在测试用具中实例化该类，但这里存在几个问题。其中之一便是，我们又一次遇到了单件PermitRepository。当然，可以借助于前面讨论全局依赖时提到的技术来解决这个问题。但这之前我们首先得解决另一个问题，就是我们很难创建出一个OriginationPermit对象来传给这个类的构造函数。OriginationPermit上的依赖情况很严重。我第一时间想到的是："噢，我可以对OriginationPermit使用接口提取技术来解决这些依赖问题啊。"但事情并没那么简
133 单。请看图9-4展示的这个Permit体系结构图：

IndustrialFacility的构造函数的参数之一便是OriginationPermit，该构造函数接着从PermitRepository获取一个与之关联的许可（Permit对象）：这是通过PermitRepository

上的方法findAssociatedFromOrigination来实现的，该方法接受一个OriginationPermit对象并返回与之关联的Permit对象。如果找到了这个关联的许可，IndustrialFacility的构造函数就会将它保存到成员basePermit中。如果没有，则将那个作为构造函数参数的OriginationPermit对象保存到basePermit中。我们可以为OriginationPermit创建一个接口，但这样做没什么好处，因为这么一来当我们将这个参数赋给basePermit时，它们的类型一个是IOriginationPermit，而另一个则是Permit，这是行不通的。在Java里面，接口不能从类派生得来。最显而易见的解决方案就是顺着继承体系一路往下创建接口，并将basePermit的类型改为IPermit。图9-5显示了这种情形：

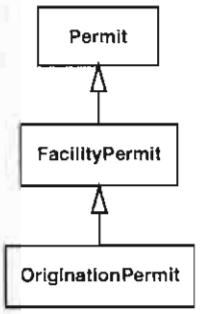


图9-4 Permit继承体系

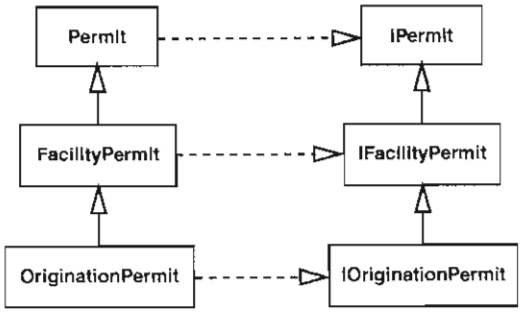


图9-5 接口提取后的Permit继承体系

这样做的工作量也太大了，而且我也不是特别喜欢代码最后的样子。接口的确是解依赖的利器，但如果出现了接口与类几乎一一对应的情形，设计就变得混乱起来了。别误会，如果我们别无退路，那么像上面这样的设计也是可以的，但如果还有其他选择的话，我们当然应该尝试一下。幸运的是，的确存在其他方案：

134

接口提取只是对参数进行解依赖的途径之一。有时候问一问“为什么这个依赖是糟糕的”往往是有好处的。例如，有时候对象创建是麻烦所在，而有时候参数则会具有糟糕的副作用，比如跟一个文件系统或数据库进行通信；而还有些时候呢，也许只不过是创建起来太费时间了。当我们使用接口提取时，可以克服所有这些问题，但代价是我们粗鲁地切断了与一个类之间的联系。如果某个类只是有某几个地方有问题，则我们可以采取另一个方案，只切断与这些地方之间的联系即可。

让我们更仔细地来考察一下OriginationPermit类。我们不想在测试中使用它，因为当我们希望它进行自身验证时，它便会悄无声息地在幕后访问一个数据库：

```
public class OriginationPermit extends FacilityPermit
{
    ...
    public void validate() {
        // form connection to database
        ...
        // query for validation information
        ...
```

```
            // set the validation flag
            ...
            // close database
            ...
        }
    }
```

我们可不希望在测试时发生这种事情：那样的话就不得不在数据库中放一些“伪造的”（测试用）条目，搞得数据库管理员不得安宁。一旦被他发现，我们就不得不请他吃顿饭还是什么的来赔礼道歉，而且就算那样他还未必领情。他的工作本就已经难做了。

可以采取的另一个策略就是*子类化并重写方法*。我们可以创建一个名为FakeOrigination-Permit的类，该类提供一些方法使得外界可以很容易地改变其验证标志。接着，便可以在FakeOriginationPermit的子类中重写validate()方法，按照测试所需来设置验证标志。下面就是我们的第一个有效的测试：

```
public void testHasPermits() {
    class AlwaysValidPermit extends FakeOriginationPermit
    {
        public void validate() {
            // set the validation flag
            becomeValid();
        }
    };

    Facility facility = new IndustrialFacility(Facility.HT_1, "b",
                                               new AlwaysValidPermit());
    assertTrue(facility.hasPermits());
}
```

135

在许多语言当中，我们都可以像这样在一个方法当中“即时”地创建一个新类。虽然我并不喜欢经常在产品代码中这么做，但在测试当中这是个非常便利的特性。借助于它我们很容易就能实现一些特殊的用例。

*子类化并重写方法*能够帮助我们解开参数上的依赖，但有时类中的方法的分解方式并不十分适合这个技术。就本例来说我们比较幸运，因为我们不希望看到的依赖被隔离在了那个validate()方法中。在情况最糟糕的时候，那些依赖可能会和我们所需要的逻辑混在一起，令我们不得不先进行方法提取。这时如果我们手头有一个重构工具就会很好办，但如果没有，请参考第22章中的一些技术，可能会有帮助。

136

第10章 无法在测试用具中运行方法

有时候将测试安置到位并不是件简单的事情。如果能够在测试用具中单独实例化你的类，那你算是比较幸运的。许多人无法做到这一点。不过，如果你也在这上面遇到麻烦了，不妨尝试一下第9章描述的技术。

（在测试用具中）实例化一个类通常只是第一步。接下去便是为需要修改的方法编写测试。有些时候我们无需实例化那个类便可以直接进入第二步。例如，假设待修改的方法并没有使用多少实例变量，便可以使用*暴露静态方法*（273页）手法来访问该方法的代码。倘若该方法相当长且难于对付，则可使用*分解出方法对象*（261页）手法来将其中的代码移到一个相对来讲更容易实例化的类当中去。

幸运的是，大多数情况下，为一个方法编写测试所需的必要工作量并不算太夸张。下面列出了我们可能会遇到的一些问题：

- 无法在测试中访问那个方法。比如说，它可能是私有的，或者有其他可访问性限制。
- 无法轻易地调用那个方法，因为很难构建调用它所需的参数。
- 那个方法可能会产生糟糕的副作用（如修改数据库、发射一枚巡航导弹，等等），因而无法在测试用具中运行它。
- 我们可能会需要通过该方法所使用的某些对象来进行感知。

本章接下来将会描述一系列的问题场景，展示了解决这些问题的不同方式，以及解决过程中的权衡与折中。

10.1 隐藏的方法

我们需要对一个类中的某方法作修改，但它是一个私有的方法，这时该怎么办呢？

第一个问题就是，能否通过一个公用的方法来进行我们的测试。如果能，则值得那么做，以免我们得想方设法去访问那个私有方法。而且，这么做还有另一个好处，即当我们通过公用方法来进行测试时，肯定是按照该方法被用在实际代码中的方式来测试它的[1]。这有助于稍微缩小我们的工作范围。在遗留代码中，常常会出现一些质量有问题的方法。要想使一个私有方法对其每

1. 因为公用的方法属于类的接口，而私有方法则一般属于内部实现范畴。——译者注

个调用方都可用，可能需要进行相当量的重构工作才行。有些一般性很强的方法能够被许多调用方调用，能这样固然不错，但实际上每个方法在功能上应当刚好足够满足它的调用者，在清晰程度上也应当足够清晰，以便于理解和修改。在测试一个私有方法时，倘若是通过一个使用了它的公用方法来间接测试的话，则把这个私有方法做得一般化倒也没多大危险。如果有朝一日该方法需要成为公用的，那么其外的第一个使用者应当编写一系列测试用例，准确说明该方法的用途以及调用者该如何使用它。

以上这些都没问题，但有时我们就是想要直接为一个私有方法编写测试用例（对该方法的调用被深深埋藏在类中）。这么做的原因可能是因为我们想要获得一些具体的反馈，以及能够解释该方法是怎样被使用的测试用例，或者，也有可能只是因为通过它的类上面的公用方法来测试它太困难了，等等。

那么，如何为一个私有方法编写测试呢？这肯定是在测试中被问得最多的问题之一了。幸运的是，这个问题有一个非常直接的答案：如果需要测试一个私有方法，那么就应该将它设为公用的。如果不大方便将其设为公用的，则大多数情况下便意味着我们的类做的事情太多了，应该进行适当调整。让我们来看看什么时候不大方便将一个私有方法置为公用的呢？有两个可能的原因：

(1) 该方法只是个工具方法；客户并不会去关心它。

(2) 如果客户代码使用了该方法，那么他们可能会反过来影响到该类上的其他方法调用的结果。

第一个原因并不算很严重。类的接口上多出一个公用方法并没什么大不了，不过我们还是应该试试看将它放到另一个类当中会不会更好一些。而第二个原因要严重一点，不过幸运的是还有补救措施：这个私有方法可以被转移到一个新类当中去。我们可以让它成为这个新类上的公用方法，而我们原先的那个类则可以在内部创建该新类的实例。这么一来，这个方法也就变得可测试了，而同时我们的设计也得到了改善。

138

是的，我知道这个建议有点不中听，但它带来的一些效果却是很积极的。无论如何这个事实都不会改变：好的设计应当是可测试的，不具可测试性的设计是糟糕的。遇到上面这类情况，应对之策是尝试采用第20章中所描述的技术。不过，当并没有多少现有测试可用时，就不得不小心行事，先做一些其他工作，然后再开始分解。

先来看一个真实的案例，看看我们是如何来解决上面提到的问题的。下面是一个C++类声明的一部分：

```
class CCAImage
{
private:
    void setSnapRegion(int x, int y, int dx, int dy);
    ...
public:
    void snap();
    ...
};
```

CCAImage是一个保安系统中的一个类，负责拍照功能。你可能想知道为什么一个图像类会负责拍照，但别忘了，这是遗留代码。该类有一个snap()方法，该方法使用一个低层的C API来控制一个摄像头进行“拍”照，但是，它“拍”下来的是一种非常特殊的图像。一次对snap()的调用会导致好几个不同的摄像头被调动起来，它们各自都会拍下一幅图片，这些图片被分别放到CCAImage内部的图像缓存中的不同部位。而决定每幅图片分别被安放在哪个部位的逻辑则是动态的，取决于被拍对象的移动。根据其移动方式的不同，snap()方法会重复调用setSnapRegion()来确定当前照片应当被放在缓存的哪个部位。然而遗憾的是，现在，摄像头的API改变了，于是我们需要对setSnapRegion作相应的修改。那么，具体该怎样进行呢？

一种可能的做法就是简单地将该方法设为公用的。但可惜的是这么做会带来非常消极的影响。CCAImage类中有一些变量负责记录拍照区域的当前位置。所以倘若产品代码不小心在snap()方法外部直接调用了setSnapRegion的话，便会给摄像头跟踪系统带来严重问题。

是的，问题就在这儿。不过，在开始寻找解决方案之前，让我们先来看一下当初是怎么踏进这团泥沼中的。之所以无法测试这个图像类，真正的原因在于它负担的职责太多了。理想情况下我们应当可以使用第20章所讲的技术来将它分解为几个小类，那样的确不错，但首先得仔细考虑一下眼前是否应该进行这么多的重构工作。诚然，这么做是有极大好处的，不过能否这么做却取决于当前我们在整个产品发布周期中所处的阶段，有无足够时间以及所有相关的风险。 139

如果目前我们负担不起这个风险去将职责分离开来，那至少能为待修改方法编写测试吧？幸运的是，答案是肯定的。下面便是具体做法：

第一步是将setSnapRegion的访问权限从私有改为受保护的。

```
class CCAImage
{
protected:
    void setSnapRegion(int x, int y, int dx, int dy);
    ...
public:
    void snap();
    ...
};
```

接下来，我们通过子类化CCAImage来获得对setSnapRegion的访问权：

```
class TestingCCAImage : public CCAImage
{
public:
    void setSnapRegion(int x, int y, int dx, int dy)
    {
        // call the setSnapRegion of the superclass
      CCAImage::setSnapRegion(x, y, dx, dy);
    }
};
```

在大多数现代C++编译器下，我们还可以使用using声明来达到同样的目的。在TestingCCAImage中通过using声明来自动完成任务委托（无需像上面那样经过一层无谓的转发），如

下所示：

```
class TestingCCAImage : public CCAImage
{
public:
    // Expose all CCAImage implementations of setSnapRegion
    // as part of my public interface. Delegate all calls to CCAImage.
    using CCAImage::setSnapRegion;
}
```

在完成这些之后，便可以在测试中调用CCAImage上的setSnapRegion方法了（虽然是间接调用）。然而，这样做是好主意吗？记得一开始的时候我们并不想将这个方法设为公用，但现在我们通过其他途径达到了类似的效果。我们将方法设为受保护的，从而削弱了它的访问控制。

坦白地说，我并不介意这么做。对我来说，这样做令我们能够编写测试，是一宗公平交易。没错，这样的改动的确破坏了封装，从而当我们在分析代码是如何工作的时候，就得把CCAImage的子类也能调用setSnapRegion这一事实也考虑进去，但是，这毕竟是一个相对来说比较次要
140 的问题。或许后面当我们再次接触该类时，当初那点小小的改动就足以引发我们对该类的彻底重构。届时我们便可以将CCAImage中的职责分解到几个不同的类当中，并让后者成为可测试的。

推翻访问保护

在许多比C++晚的面向对象语言中，都可以使用反射（reflection）和特殊许可去（在运行期）访问私有变量。虽说这样的功能有些时候很好用，但有点“欺骗”的味道。诚然，当我们想要解依赖时是非常有帮助的，但我可不喜欢将访问私有变量的测试代码留在项目中。因为这种以“欺骗”手段访问私有变量的方式会蒙蔽团队的眼睛，使他们看不到代码变得有多糟糕。这样说或许有点残酷，但也别忘了，在一个遗留代码基中工作的痛苦经历反过来也能够成为我们去进行修改的强大驱动力。当然，我们也可以用“旁门左道”来达到目的，但除非我们着眼于解决根本问题（即责任过多的类和错综复杂的依赖），否则再怎么努力也只是暂时逃避问题，最终，当每个人都发现代码变得糟糕时再去进行改善所花的代价就难以想象了。

10.2　“有益的”语言特性

语言设计者经常会试图加入一些方便的语言特性，但这件事并不容易。他们得在易编程性与安全性中进行折中。有些语言特性一开始看上去的确是“面面俱到”了，然而当我们想要测试使用了这些特性的代码时，残酷的现实就显露出来了。

下面这段C#代码负责从一个Web客户端接收一组上载的文件，然后遍历它们，从中挑出具有特定特征的文件，然后返回一组与这些文件关联的流。

```
public void IList getKSRStreams(HttpFileCollection files) {
    ArrayList list = new ArrayList();
    foreach(string name in files) {
        HttpPostedFile file = files[name];
        if (file.FileName.EndsWith(".ksr") ||
```

```
                (file.FileName.EndsWith(".txt")
                        && file.ContentLength > MIN_LEN)) {
            ...
            list.Add(file.InputStream);
        }
    }
    return list;
}
```

现在，我们想要对以上代码作一些修改，或许再来一点重构，但问题是给它编写测试就不那么容易了。我们想要创建一个HttpFileCollection容器对象并往它里面放入一组 141
HttpPostedFile对象，而这又是不可能的。首先，HttpPostedFile类并没有公用的构造函数。其次，它是一个封闭的类。在C#中，这两点便意味着我们没法创建HttpPostedFile的实例，而且我们也无法从它进行派生。HttpPostedFile是.NET库的一部分。里面的其他一些类会在运行期创建它的实例，但我们却没有对它的访问权。此外，打开HttpFileCollection类的定义稍微看一看就会发现，它也有同样的问题：无公用构造函数，无法创建其派生类。

.NET类库为什么要这么做呢？毕竟我们花钱买了它的许可证啊。说实话我倒并不认为这是故意的，如果说微软是故意这么做的话，那Sun也一样，因为这并不只是微软的语言的问题。Sun的语言也有阻止子类化的特性。在Java中，final关键字正是用来干这个的，当一个类在安全性方面很敏感时，便可对它使用final。若是任何人都能创建HttpPostedFile（或者甚至像String这样的类）的子类的话，他们岂不就能编写出一些恶意的代码并将其用于那些使用了这些类的代码中了？这是非常危险的，不过sealed和final关键字有时候对我们来说又显得过于激进了，就拿刚才讨论的情况来说吧，它给我们带来了不小的麻烦。

那么，要想为getKSRStreams编写测试的话有什么办法呢？我们不能使用*接口提取*（285页）或*实现提取*（281页），因为HttpPostedFile和HttpFileCollection并不由我们控制，它们是类库里面的类，是不能随便去修改的。所以我们只能使用*参数适配*（Adapt Parameter，258页）手法了。

不过，就这个例子来说我们还是挺幸运的，因为我们对那个HttpFileCollection所作的只不过是遍历而已。虽然HttpFileCollection是个封闭类，但它却有一个非封闭的基类，叫做NameObjectCollectionBase。我们可以对后者进行子类化，并将所得子类的对象传给getKSRStreams方法。借助于*依靠编译器*（251页）技术，我们的修改既安全又容易。

```
public void LList getKSRStreams(OurHttpFileCollection files) {
    ArrayList list = new ArrayList();
    foreach(string name in files) {
        HttpPostedFile file = files[name];
        if (file.FileName.EndsWith(".ksr") ||
                (file.FileName.EndsWith(".txt")
                    && file.ContentLength > MAX_LEN)) {
            ...
            list.Add(file.InputStream);
        }
    }
    return list;
}
```

OurHttpFileCollection是NameObjectCollectionBase的子类，而NameObject-
142 CollectionBase则是一个抽象类，其功能是将字符串关联到对象。

这样一来我们就解决了其中一个问题。接下来的问题要稍困难一些：要在测试中运行getKSRStreams，我们需要一组HttpPostedFiles，而不幸的是我们偏偏又无法创建HttpPostedFile的实例。既然如此，不妨换个角度来思考：我们实际上需要用到的是HttpPostedFile上的两个属性：FileName和ContentLength。故而我们可以利用剥离并外覆API（169页）技术，解开代码与HttpPostedFile之间的耦合。为此，首先提取一个接口IHttpPostedFile，然后编写一个外覆类HttpPostedFileWrapper：

```
public class HttpPostedFileWrapper : IHttpPostedFile
{
    public HttpPostedFileWrapper(HttpPostedFile file) {
        this.file = file;
    }

    public int ContentLength {
        get { return file.ContentLength; }
    }
    ...
}
```

既然有了这么一个接口，也就可以利用它来创建一个测试用类了：

```
public class FakeHttpPostedFile : IHttpPostedFile
{
    public FakeHttpPostedFile(int length, Stream stream, ...) { ... }

    public int ContentLength {
        get { return length; }
    }
}
```

现在，如果我们依靠编译器（251页），并对产品代码作一点修改的话，就可以通过IHttpPostedFile接口来使用HttpPostedFileWrapper或FakeHttpPostedFile对象，而无需知道到底背后被使用的是哪一个了。

```
public IList getKSRStreams(OurHttpFileCollection) {
    ArrayList list = new ArrayList();
    foreach(string name in files) {
        IHttpPostedFile file = files[name];
        if (file.FileName.EndsWith(".ksr") ||
                (file.FileName.EndsWith(".txt"))
                    && file.ContentLength > MAX_LEN)) {
            ...
            list.Add(file.InputStream);
        }
    }
    return list;
}
```

唯一麻烦的就是，我们必须得在产品代码中先遍历一遍原先的HttpFileCollection容器，将其中的每个HttpPostedFile对象都“打包”进一个相应的HttpPostedFileWrapper对象中，然后将这些外覆对象放进一个新的容器（OurHttpFileCollection）中，并将后者传给getKSRStreams方法。这就是安全性所要付出的代价。 143

说真的，我们很容易相信sealed跟final都是错误的特性，本就不应该加入到编程语言中。然而实际上，真正的错误却出在我们自己身上，是我们自己选择直接依赖于不由我们控制的库的，这一举动等于是在自寻烦恼。

将来的主流编程语言或许会给测试提供特殊访问权限，但现在，保守使用sealed和final还是有好处的。当我们需要使用标记为sealed/final的类时，最好将它们隔离在一层外覆类后面，这样以后对代码作修改时才能有一些回旋余地。关于如何解决该问题的更多讨论和技术，请参考第14章以及第15章。

10.3　无法探知的副作用

理论上，为一段功能编写测试不应该太难。实例化一个类，调用它的方法，然后检查结果，就这样简单。那么，哪个环节可能出问题呢？实际上，如果我们欲创建的对象不跟其他任何对象沟通的话，事情的确就像刚才描述的那样简单。甚至就算其他对象使用了该对象，但只要该对象并不使用其他对象，我们的测试就可以像程序的其他部分那样使用它。然而遗憾的是，不使用其他对象的对象少之又少。

程序是一个各部分互相协作的整体。常常会看到一些并不返回任何值的方法。我们调用这些方法，它们完成各自的工作，而我们（指调用方代码）则根本不知道它背后都干了些什么。某对象调用其他对象上的方法，而我们则根本无从知道结果。

下面这个类暴露了上面所说的问题：

```
public class AccountDetailFrame extends Frame
            implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    ...
    public AccountDetailFrame(...) { ... }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
        if (source.equals("project activity")) {
            detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                    getDetailText() + " " + getProjectionText());
            detailDisplay.show();                                    144
            String accountDescription
                    = detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
```

```
            ...
        }
    }
    ...
}
```

这个遗留的Java类什么都做。它创建GUI构件，并使用actionPerformed事件处理函数从它们那里接收通知消息，然后计算需要显示的内容并显示它们。不过，它做这些事情的方式非常奇怪：首先建造一段详细文本，然后创建并显示另一个窗口。当该窗口完成它的工作时，该类再直接从它获取信息，作一点处理，然后放入到一个文本框[1]中。

我们可以试着在一个测试用具中运行该方法，但这么做没有任何意义。它会创建一个窗口，显示给我们，让我们输入数据，然后接着在另一个窗口显示一些东西。没有合适的地方可以感知这段代码做了什么[2]。

那么，我们可以做些什么呢？首先，可以将依赖于GUI的代码与不依赖于GUI的代码分离开来。由于所用的编程语言是Java，所以我们可以选一个Java的重构工具来用。第一步工作就是执行一组*方法提取*（325页），将这个大方法中的工作分割成小块。

那么，具体从哪开始呢？

该方法本身主要是起到一个事件响应“钩子”的作用，负责响应窗口框架传递来的通知消息。它所做的第一件事情便是从接受到的动作事件[3]中获取命令的名字。所以，如果将该方法的整个方法体都提取出来的话，也就可以完全脱离对ActionEvent类的依赖了，如下所示：

```
public class AccountDetailFrame extends Frame
            implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    ...
    public AccountDetailFrame(...) { ... }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
        performCommand(source);
    }

    public void performCommand(String source) {
        if (source.equals("project activity")) {
            detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                    getDetailText() + " " + getProjectionText());
            detailDisplay.show();
            String accountDescription
                    = detailDisplay.getAccountSymbol();
            accountDescription += ": ";
```

145

1. 成员变量display。——译者注
2. 副作用。——译者注
3. 参数event。——译者注

```
            ...
            display.setText(accountDescription);
            ...
        }
    }
    ...
}
```

然而要想让这些代码真正变成可测试的，这点工作还不够。下一步便是将访问另一个窗体的代码提取成方法。为此，一个有益的做法是将detailDisplay设成该类的一个实例变量，如下所示：

```
public class AccountDetailFrame extends Frame
            implements ActionListener, WindowListener
{
    private TextField display = new TextField(10);
    private DetailFrame detailDisplay;
    ...
    public AccountDetailFrame(...) { .. }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
        performCommand(source);
    }

    public void performCommand(String source) {
        if (source.equals("project activity")) {
            detailDisplay = new DetailFrame();
            detailDisplay.setDescription(
                    getDetailText() + " " + getProjectionText());
            detailDisplay.show();
            String accountDescription
                    = detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
        }
    }
    ...
}
```

146

有了这一步铺垫，我们便可以将使用detailDisplay窗体的代码提取成一组方法了。那么该如何为这组方法命名呢？为此，我们从该类的角度来考虑每段代码都做了些什么，或者说它们都为该类计算了些什么。此外，我们不应该使用与显示组件有关的名字，可以在提取出的代码中使用显示组件，但其方法名却应当隐藏这一事实。有了这些考虑，我们便可以将提取出的每块代码做成一个命令式方法或查询式方法了。

命令/查询分离

命令/查询分离是最先由Bertrand Meyer提出的设计准则。简而言之就是：一个方法要么是

一个命令，要么是一个查询；但不能两者都是。命令式方法指那些会改变对象状态但并不返回值的方法。而查询式方法则是指那些有返回值但不改变对象状态的方法。

那么，为什么说这是个重要的原则呢？有几个原因，其中最重要的就是它向用户传达的信息。例如，如果一个方法是查询式的，那么无需查看其方法体就知道可以连续多次使用它而不用担心会带来什么副作用。

在经过了一系列的方法提取之后，performCommand方法看起来就像这样：

```
public class AccountDetailFrame extends Frame
            implements ActionListener, WindowListener
{
    public void performCommand(String source) {
        if (source.equals("project activity")) {
            setDescription(getDetailText() + " " + getProjectionText());
            ...
            String accountDescription = getAccountSymbol();
            accountDescription += ": ";
            ...
            display.setText(accountDescription);
            ...
        }
    }

    void setDescription(String description) {
        detailDisplay = new DetailFrame();
        detailDisplay.setDescription(description);
        detailDisplay.show();
    }

    String getAccountSymbol() {
        return detailDisplay.getAccountSymbol();
    }
    ...
}
```

147

既然我们已经将所有与detailDisplay窗体有关的代码都提取出来了，那么接下来就可以找出并提取那些访问AccountDetailFrame上的组件的代码了。

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener {
    public void performCommand(String source) {
        if (source.equals("project activity")) {
            setDescription(getDetailText() + " " + getProjectionText());
            ...
            String accountDescription
                    = detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            ...
            setDisplayText(accountDescription);
            ...
        }
```

```
    }

    void setDescription(String description) {
        detailDisplay = new DetailFrame();
        detailDisplay.setDescription(description);
        detailDisplay.show();
    }

    String getAccountSymbol() {
        return detailDisplay.getAccountSymbol();
    }

    void setDisplayText(String description) {
        display.setText(description);
    }
    ...
}
```

在作了这一番提取之后，我们便可以运用一下子类化并重写方法技术，并对performCommand中剩下来的代码进行测试了。例如，像下面这样子类化了AccountDetailFrame之后，我们便可以测试当给出"project activity"命令时display文本框能否得到正确的文本了：

```
public class TestingAccountDetailFrame extends AccountDetailFrame
{
    String displayText = "";
    String accountSymbol = "";

    void setDescription(String description) {
    }

    String getAccountSymbol() {
        return accountSymbol;
    }

    void setDisplayText(String text) {
        displayText = text;
    }

}
```

148

下面这段代码就是用来测试performCommand方法的：

```
public void testPerformCommand() {
    TestingAccountDetailFrame frame = new TestingAccountDetailFrame();
    frame.accountSymbol = "SYM";
    frame.performCommand("project activity");
    assertEquals("SYM: basic account", frame.displayText);
}
```

在像上面这样非常保守地通过自动方法提取进行解依赖之后，得到的代码可能会让我们心生怯意。例如，setDescription方法负责创建并显示一个窗体，这样的方法绝对是令人头大的。如果我们不小心两次调用了它会怎么样呢？得解决这个问题，以上一系列粗糙的方法提取是个不

错的起点。接下来我们可以看看能否将这些窗体创建代码重新安置到一个更好的地方去。

我们来理一下目前的状况：一开始我们有一个类，该类上面有一个重要的方法：performAction。而现在的状况则可以用下图来显示（图10-1）。

虽然从UML图中看不出来，但实际上getAccountSymbol和setDescription只使用了detailDisplay成员变量。而setDisplayText也只是使用了display成员变量而已。于是我
149 们可以将它们看作互相独立的职责，从而最终得到如图10-2所示的设计：

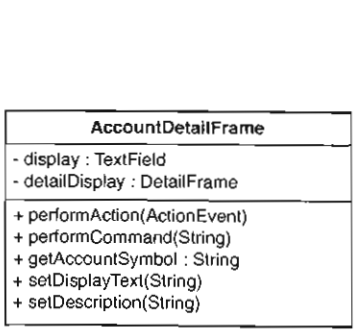


图10-1　AccountDetailFrame

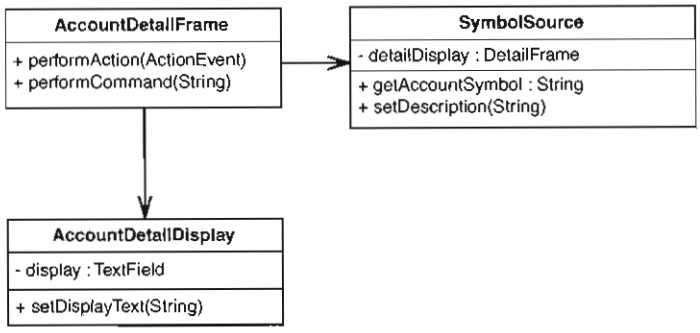


图10-2　AccountDetailFrame粗略地重构之后

虽说这是极其粗糙的重构，但至少在某种程度上将几个职责分离开来了。不过，AccountDetailFrame（Frame的子类）仍跟GUI联系紧密，并且它里面仍然包含着业务逻辑。进一步的重构可以解决这个问题，但至少现在我们可以在测试用例中运行那个原先包含了业务逻辑的方法了。这不能不说是一个积极的进展。

SymbolSource类是一个具体类，它代表着创建另一个窗体并从其获取信息的决策。然而，我们之所以将它起名为SymbolSource，是因为从AccountDetailFrame的角度来说该类的工作只是通过它所认为必要的方法获取到某些符号形式的信息。如果SymbolSource演化成了一个接口我也肯定不会感到惊讶，只要背后获取信息的途径[1]改变了便可能会出现这一情况。

本例中我们采取的步骤是很常见的。在有重构工具可用的情况下，很容易就可以对一个类进行方法提取，然后将方法分组，以便可以放到新类中去。一个好的重构工具能够判断你想要进行的自动方法提取是不是安全的，如果不安全便不会予以进行。然而，这只会令我们进行的其他修改成为工作中最危险的部分。所以说，记住，如果目的是为了让测试能够安置到位的话，提取出具有糟糕名字或糟糕结构的方法是可以接受的。毕竟，安全才是第一位。在测试到位之后，就可
150 以放心着手让代码变得更清爽了。

1. 如不再通过窗体+用户输入的方式，而是通过网络或数据库来获取，等等。——译者注

第11章 修改时应当测试哪些方法

假设我们现在需要作一些代码修改，并且需要编写特征测试（153页）来“固定”住已有的行为，那么应当为哪些地方编写测试呢？最简单的答案就是为我们所要修改的每个方法都编写测试。但这就够了吗？如果代码较为简单且易于理解的话的确是够了，但对于遗留代码来说，往往并非如此。一个地方的改动可能会影响到其他地方的行为；除非有测试“坐镇”，否则我们可能永远也不知道自己的修改造成了什么影响。

当需要在特别错综复杂的遗留代码中作改动时，我通常会先花点时间考虑一下应当在哪儿编写测试。这一过程包括考察将要进行的改动，看看它会带来哪些影响，看看被影响的东西又进而会对哪些东西造成影响，后者又会造成哪些影响……。这种推理方式并不新鲜，人们在计算机的启蒙时代就这么做了。

程序员们可能会因为各种各样的原因而坐下来对他们的程序进行推理。有意思的是，对此我们谈论得并不多。我们只是假设每个人都知道怎么做，以及假设这是程序员的“份内之事”。然而当我们面对的是错综复杂的、不易理清的代码时，光是嘴上说说是无济于事的。我们知道应该对代码作一点重构来让它更易于理解，但前面遇到过的测试困境又出现了：如果没有测试在手，我们又如何能知道正在进行的重构是正确的呢？

本章描述的技术填补了这个空白。看来，对于遗留代码，通常我们的确得花点功夫来推测一下代码修改会产生哪些影响，以便找到编写测试的最佳地点。

11.1 推测代码修改所产生的影响

虽说在业界我们就这个问题谈论得并不多，然而实际情况是，每对一个软件作一次功能上的改动，都会带来一连串互相关联的影响。例如，假设我们将下面这段C#代码中的3改为4，就会影响到该函数的返回值，并进而影响到调用该函数的函数的返回值……一路影响下去，直到遇到某种系统边界为止。话虽如此，仍有许多代码的行为还是跟以前一样。由于并没有调用`getBalancePoint()`，所以它们给出的仍是原来的结果。


```
int getBalancePoint() {
    const int SCALE_FACTOR = 3;
    int result = startingLoad + (LOAD_FACTOR * residual * SCALE_FACTOR);
    foreach(Load load in loads) {
```

```
            result += load.getPointWeight() * SCALE_FACTOR;
        }
        return result;
    }
```

IDE对代码影响分析的支持

有时候真希望有个IDE能帮我在遗留代码中“看到”代码修改所产生的影响。想象这样一种情景：选中某块代码，敲下一个快捷键，于是IDE便给出了对该块代码作改动所可能影响到的所有变量和方法的列表。

或许有一天人们会开发出这样的工具。但在那一天到来之前我们还是得学习如何在没有工具的情况下仅凭大脑去推测代码修改的影响。这个技能学起来不难，但我们很难知道何时才算正确掌握了它。

要想了解影响推测是个什么概念，最佳途径就是从实例入手。下面就是一个Java类，该类所属应用程序的功能是操纵C++代码。这听起来似乎太专业，但其实在推测代码修改的影响时，有没有相关的领域知识并不重要。

让我们来做一个小练习。下面是一个名为CppClass的类，列出其中所有能够在CppClass对象创建之后被改变，从而对其方法的返回值产生影响的东西。

```
public class CppClass {
    private String name;
    private List declarations;

    public CppClass(String name, List declarations) {
        this.name = name;
        this.declarations = declarations;
    }

    public int getDeclarationCount() {
        return declarations.size();
    }

152 public String getName() {
        return name;
    }

    public Declaration getDeclaration(int index) {
        return ((Declaration)declarations.get(index));
    }

    public String getInterface(String interfaceName, int [] indices) {
        String result = "class " + interfaceName + " {\npublic:\n";
        for (int n = 0; n < indices.length; n++) {
            Declaration virtualFunction
                    = (Declaration)(declarations.get(indices[n]));
            result += "\t" + virtualFunction.asAbstract() + "\n";
        }
        result += "};\n";
```

```
            return result;
        }
    }
```

你的答案看起来应该像下面这样：

(1) 可以在declarations列表被传递给CppClass的构造函数之后[1]再往它里面添加额外的元素。由于该列表是按引用传递给CppClass的构造函数并由CppClass的declarations成员变量按引用持有的，因此对它的改动会影响到getInterface、getDeclaration以及getDeclarationCount的结果。

(2) 可以改动或替换declarations列表内的元素，同样还是影响到那几个方法。

有些人看到getName()可能会想，如果有人改动了成员变量name的话，它的返回值便也被改变了，然而实际上，在Java中，String对象是常性的。也就是说它们一旦被创建，值就无法改变了。所以，在CppClass对象被创建出来之后，其getName()便总会返回同样的String值。

下面的一幅图展示了对declarations的改动是怎样影响到getdeclarationCount()的(图11-1)。

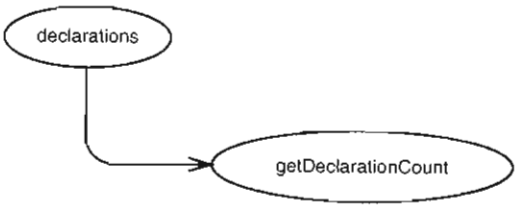


图11-1　declarations影响getDeclarationCount

153

从该图中我们可以看出，如果declarations发生了某些改变，例如其中的元素数目改变了，那么getDeclarationCount()的返回值也会随之改变。

同样，对于getDeclaration()，我们也可以画出一张图来（图11-2）。

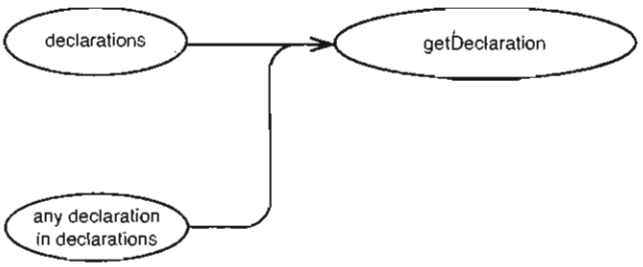


图11-2　declarations以及它持有的元素对getDeclaration的影响

1. 即CppClass对象已经被构造起来之后。——译者注

如果有人改动了declarations或者其中的元素，则getDeclaration(int index)的返回值也会改变。

图11-3展示了getInterface被影响的情况。

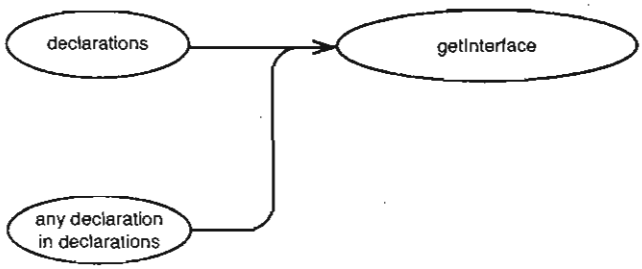


图11-3　影响getInterface的因素

154　现在，我们可以将上面这几幅图结合起来，成为一张大图（图11-4）。

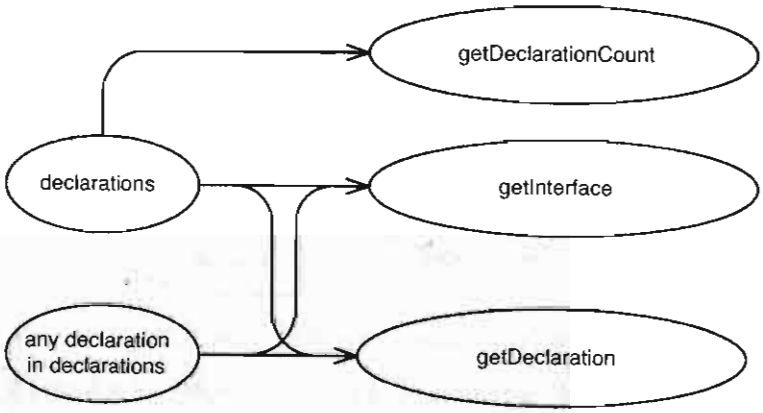


图11-4　合并起来的影响图

这种图的规则并不复杂。我把它们称为影响草图[1]。作图的关键是：为每个可能会被影响到的变量以及每个返回值可能改变的方法画一个单独的椭圆。这些变量可能来自同一个对象，也可能来自不同的对象。究竟属于何者并不重要，我们只需为每个会改变的东西画上一个椭圆，并从它们出发画一个箭头指向那些因它们的改变而在运行期改变的东西。

倘若你的代码结构良好，则其中的大多数方法的影响结构也会比较简单。实际上，衡量软件好坏的标准之一便是，看看该软件对外部世界的相当复杂的影响能否由代码内的一组相对简单得多的影响所构成。任何改动，只要能够使代码的影响结构图简单化，就能够使其更易理解和维护。

1. 后文也有称“影响结构图”、“影响结构示意图”的，意思一样。——译者注

让我们把视野放远一点，看一下CppClass所处系统的影响结构图。CppClass对象是在一个名为ClassReader的类中被创建出来的。实际上，我们已经能够确定，它们仅在ClassReader中被创建。

```
public class ClassReader {
    private boolean inPublicSection = false;
    private CppClass parsedClass;
    private List declarations = new ArrayList();
    private Reader reader;

    public ClassReader(Reader reader) {
        this.reader = reader;
    }
    public void parse() throws Exception {
        TokenReader source = new TokenReader(reader);
        Token classToken = source.readToken();
        Token className = source.readToken();

        Token lbrace = source.readToken();
        matchBody(source);
        Token rbrace = source.readToken();

        Token semicolon = source.readToken();

        if (classToken.getType() == Token.CLASS
                && className.getType() == Token.IDENT
                && lbrace.getType() == Token.LBRACE
                && rbrace.getType() == Token.RBRACE
                && semicolon.getType() == Token.SEMIC) {
            parsedClass = new CppClass(className.getText(),
                                       declarations);
        }
    }
    ...
}
```

记得我们之前对CppClass有哪些了解吗？当时我们能否知道一个CppClass对象在被创建出来之后，它所持有的declarations列表会不会再改变呢？这个问题是没法在CppClass那儿找到答案的，我们需要弄清楚declarations列表是怎么被填充的。如果进一步考察上面这个类，便能看出，ClassReader中只有一处地方往declarations列表中添加了Declaration对象，那就是在matchVirtualDeclaration方法当中。具体过程为：parse()中调用了matchBody()，后者进而调用了下面这个matchVirtualDeclaration方法：

```
private void matchVirtualDeclaration(TokenReader source)
                    throws IOException {
    if (!source.peekToken().getType() == Token.VIRTUAL)
        return;
    List declarationTokens = new ArrayList();
    declarationTokens.add(source.readToken());
    while(source.peekToken().getType() != Token.SEMIC) {
        declarationTokens.add(source.readToken());
```

```
        }
        declarationTokens.add(source.readToken());
        if (inPublicSection)
            declarations.add(new Declaration(declarationTokens));
    }
```

看上去，所有在这个列表上发生的事情都在CppClass对象被创建出来之前发生了。由于我
156 们将一个元素存入declarations列表之后便不再保留其任何引用[1]，所以该列表便没法再被更改了。

再来考察一下declarations列表里面的元素。TokenReader的readToken方法返回的是一个Token对象，后者仅持有一个字符串和一个永不改变的整型数。毫不夸张地说，只需扫一眼Declaration类的定义便可发现，一旦其对象被创建起来，就没有任何其他东西可以再改变它的状态了，于是我们可以很放心地断言，一个CppClass对象被创建后，其中的declarations列表以及列表里的元素便不再变动了。

以上这些分析对我们是有帮助的，如果我们从CppClass对象那儿得到了一些意外的结果，那么我们知道只需从几个地方下手来找原因就可以了。一般来说，可以查看CppClass的子对象都是在哪些地方创建的，从而去弄清楚发生了什么。另外我们还可以给CppClass持有的某些引用加上final修饰，从而使它们成为常量，这样代码便更加清晰了。

对于写得糟糕的程序，我们往往会发现很难弄明白眼下发生的事情因何而起。当面对一个意料之外的结果时，需要对付的其实是一个调试问题，我们得从问题的现象一路推到它的源头。然而倘若面对的是遗留代码，需要考虑的便是另外一个问题了，即一次特定的修改可能会对程序的其余结果产生何种影响。

这就要求我们从修改点一路向前推测影响。掌握了这种推理方式，也就初步掌握了寻找编写测试的合适地点的技术。

11.2　前向推测

在前面的例子中，我们试着从代码中某个特定地点的值的异常情况出发，推断出是哪些对象对它产生影响。然而，在编写特征测试（153页）时，这个过程是反过来的。具体来说就是，面对一组对象，试图搞清如果它们停止工作的话“下游”会发生什么情况。例如，下面这个类来自一个内存中的文件系统。针对它目前还没有任何测试存在，不过我们想要对它作一些修改。

```
public class InMemoryDirectory {
    private List elements = new ArrayList();

    public void addElement(Element newElement) {
        elements.add(newElement);
157 }
```

1. declarations.add(new Declaration(declarationTokens))这行代码显示出，我们并没有保留被加入到declarations列表中的元素的引用，一个Declaration一旦new出来便当即被加到declarations中去了。——译者注

```
    public void generateIndex() {
        Element index = new Element("index");
        for (Iterator it = elements.iterator(); it.hasNext(); ) {
            Element current = (Element)it.next();
          index.addText(current.getName() + "\n");
        }
        addElement(index);
    }

    public int getElementCount() {
        return elements.size();
    }

    public Element getElement(String name) {
        for (Iterator it = elements.iterator(); it.hasNext(); ) {
            Element current = (Element)it.next();
            if (current.getName().equals(name)) {
                return current;
            }
        }
        return null;
    }
}
```

InMemoryDirectory是一个不大的Java类。我们可以创建一个InMemoryDirectory对象，往其中添加元素（addElement()），生成索引（generateIndex()），并访问其中的元素。这里的元素（Element）即内部存有文本的对象，就像文件一样。生成索引的过程是这样的：创建一个名为“index”的元素，然后将其他所有元素的名字添加到该元素内的文本区中（每个名字一行）。

InMemoryDirectory有一个旧特性，就是generateIndex不能被调用两次，否则就会导致存在两个不同的索引元素(第二个索引创建时实际上是把第一个索引元素当作自己的一般元素了)。

幸运的是，我们的程序使用InMemoryDirectory的方式是非常规矩的。创建目录对象，用元素填充它，调用它的generateIndex方法，然后传递给需要访问它的元素的地方。目前为止一切都好，然而问题是现在我们要进行一次修改，从而允许人们在目录对象的生命周期中的任何时候都可以往里添加元素。

理想情况下，我们希望随着元素被添加进目录，索引的创建和维护工作会自动完成。当第一个元素添加进目录时，索引就应该自动建立起来，而且应当包含被添加元素的名字。下一次，当又有新元素添加进这个目录时，刚才建立的那个索引应该自动被更新以包含这个新元素的名字。 158
看起来给这个新功能编写测试以及满足测试的代码是件再简单不过的事情，但问题是针对目前的行为还没有任何测试存在。那么，我们如何知道将测试安置在哪呢？

本例中答案很清楚：我们需要一系列的测试，它们以各种方式调用addElement，然后生成一个索引，最后获取这些元素看看它们是否正确。问题是，我们怎么知道应该使用这些而不是其他方法呢？本例中问题比较简单，测试只不过是对我们期望的使用目录的方式的描述。我们甚至

用不着去看目录类的代码就可以写出这些测试来，因为我们对该类应该做什么早就有很好的把握。不过遗憾的是，有些时候找出合适的测试地点并不是件简单的事。我本可以在这个例子中采用一个大而复杂的类，就像那种常常潜藏在遗留系统中的那些类。但那样一来你可能就会不耐烦地把书给合上了。所以还是让我们假设这是一个棘手的类吧，看看通过考察代码怎样才能弄明白应该测试哪些东西。而对于比这更棘手的问题，这里的推理方式同样适用。

本例中我们首先需要做的事情便是弄清楚该在哪些地方进行修改。答案是我们需要从generateIndex()中移除一些功能，并往addElement()中添加一些功能。确定了这些修改点之后，便可以开始勾勒影响结构示意图了。

首先来看generateIndex()。这个类当中没有任何其他方法调用它，唯一调用它的地方就是客户代码了。那么，另一方面，generateIndex()当中创建了一个新的元素（索引元素）并将该新元素添加进了目录中，它会对该目录类中的元素集合产生影响（见图11-5）。

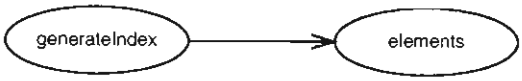


图11-5　generateIndex 影响了元素集合

现在我们便可以转而考虑元素集合会影响哪些东西了。那么，还有哪些地方使用了元素集合呢？getElementCount似乎算一个，getElement也是。另外，addElement也会用到这个元素集合，不过我们可以不考虑它，因为不管元素集合怎么改变，addElement的行为始终是不变的：
159 也就是说，无论我们对元素集合做什么，addElement的用户都不会受到影响（见图11-6）。

这幅图到这儿算是完成了吗？还没有，我们的修改点是在generateIndex和addElement这两个方法中，因此还需要考察addElement是如何影响周围的系统的。首先，看起来addElement同样会对元素集合造成影响（见图11-7）。

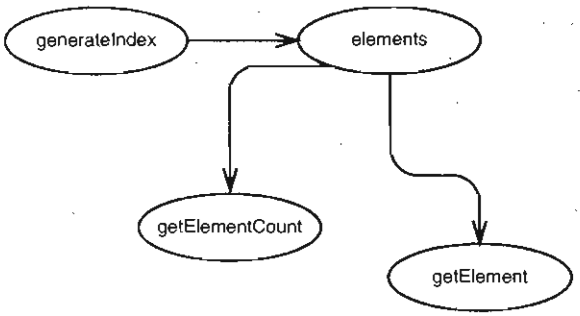


图11-6　generateIndex的改变进一步带来的影响

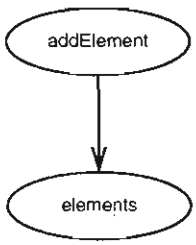


图11-7　addElement影响元素集合

我们可以进一步看看元素集合的改变会造成哪些影响，不过由于前面我们在分析generateIndex的影响结构时已经这么做过了，所以不再重复。

现在可以画出整个的影响结构示意图了（见图11-8）。

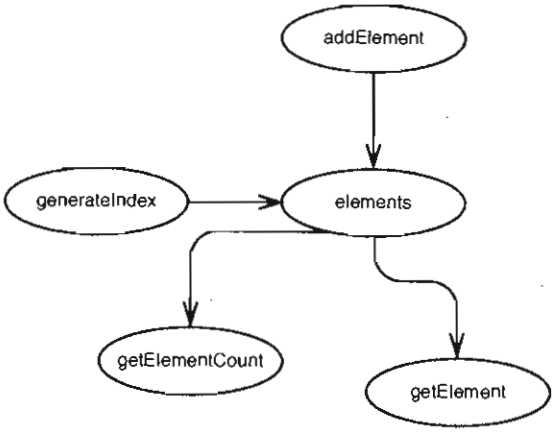


图11-8　InMemoryDirectory类的影响结构示意图

InMemoryDirectory类的用户能够感知到影响的唯一途径便是通过getElementCount和getElement这两个方法。只要我们能够对这两个方法编写测试，似乎也就可以涵盖可能造成的一切影响了。 160

不过，有没有可能漏掉了什么东西呢？我们考虑过它的基类和派生类吗？如果InMemoryDirectory中的某些数据是公有的、受保护的或者包作用域的，那么其派生类中的方法便能以我们所不知道的方式来修改它们。不过，由于本例中InMemoryDirectory中的实例变量是私有的，因此我们无需担心这种情况。

在画影响结构图的时候，你得确保找到了所考察的类的所有客户端。如果你的类有一个基类或派生类，那么得注意一下它们里面是不是还有没有被注意到的客户代码。

到目前为止方方面面都考虑到了吗？还没有，事实上还有一件事情我们一直没有提及。我们的目录类中的元素的类型为Element，但影响示意图中并未出现这个类。下面我们来仔细考察一下它。

generateIndex方法的工作方式是先创建一个Element，然后不断往它里面添加文本。现在让我们来看一看Element类的实现代码：

```
public class Element {
    private String name;
    private String text = "";

    public Element(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
```

161

```
    public void addText(String newText) {
        text += newText;
    }

    public String getText() {
        return text;
    }
}
```

所幸的是Element类的定义还算简单。在下面的影响结构图中给generateIndex创建的新元素画一个椭圆（见图11-9）。

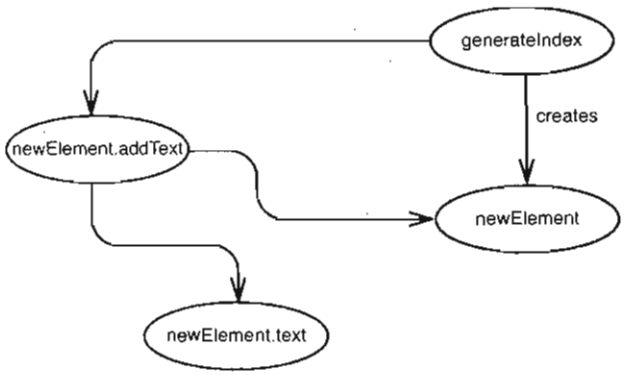


图11-9　通过Element类的影响

当新建的元素内部被填充了文本之后，generateIndex便会将它添加进元素集合中，所以，
162 这个新元素影响了元素集合（见图11-10）。

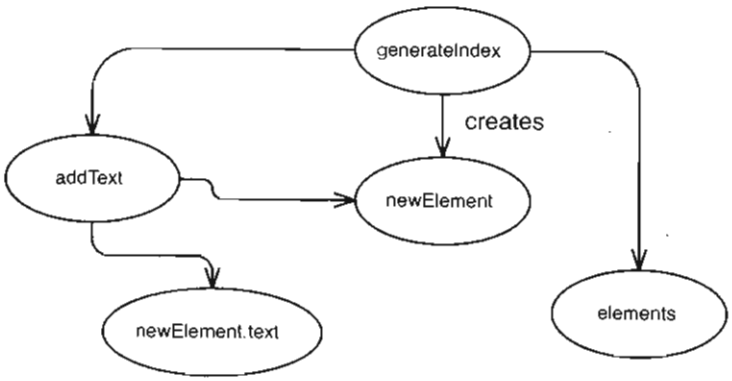


图11-10　generateIndex影响了elements集合

从我们前面的分析中得知，addText方法会影响元素集合，后者又会进一步影响getElement和getElementCount方法的返回值。如果我们想要看看索引元素当中的文本是否

正确，可以先用getElement获取它，然后调用其上的getText方法。以上便是所有需要编写测试来侦测修改能带来影响的地方了。

正如前面提到的，这虽是一个相当小的例子，然而却很能代表我们在估计对遗留代码的修改所带来的影响时需要进行的那种推断。要找到安放测试的地点，第一步便是推断出哪儿可以探测到我们的修改所带来的影响，即修改会带来哪些影响。知道了在哪儿能够探测到影响之后，在编写测试的时候便可以在这些地方进行选择了。

11.3　影响的传播

代码修改所产生的影响的传播方式有的难以察觉，有的则明显一些。在上一节中的InMemoryDirectory例子中，我们最后的任务是寻找返回值给调用方的方法。尽管一开始是从修改点开始跟踪修改所产生的影响的，我通常还是会先注意到那些带有返回值的方法。除非它们的返回值没有被使用，否则便会将影响传播到它们的调用者那儿。 163

影响也可能会悄无声息地以不易觉察的方式传播。如果我们有一个对象，而该对象又以另一个对象为参数的话，前者便可能修改后者的状态，而这一修改则会在应用当中的其他地方体现出来。

> 关于方法的参数是如何被对待（传递）的，每门语言都有不同的规则。许多时候默认的做法便是按值传递对象的引用。这也正是Java和C#的默认做法。这种做法的关键在于我们并不是将对象本身传递给一个方法，而是传递它的一个“句柄（Handle）”。这一事实带来的结果便是，任何方法都可以通过它们接受到的句柄来修改相应对象的状态。此外，有些语言也会提供一些关键字来指出一个句柄只能用于读取而不能用于修改它所指向的对象的状态。例如C++中的const关键字，当将它用于方法形参声明中时，就能够起到上述作用。

代码影响代码的最难以觉察的方式便是通过全局或静态数据了，如下所示：

```
public class Element {
    private String name;
    private String text = "";

    public Element(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void addText(String newText) {
        text += newText;
        View.getCurrentDisplay().addText(newText);
    }

    public String getText() {
```

```
            return text;
        }
    }
```

上面这个类跟我们在InMemoryDirectory中用到的那个Element类几乎一模一样，只不过有一行代码不同：addText中的第二行代码。光看Element的成员方法的签名根本无助于我们发现元素的改变对视图产生的影响。信息隐藏是件好事，只不过，若是被隐藏的是我们需要知道的
164 信息就不妙了。

影响在代码中的传递有三种基本途径：

(1) 调用方使用被调用函数的返回值。

(2) 修改传参[1]传进来的对象，且后者接下来会被使用到。

(3) 修改后面会被用到的静态或全局数据。

不过，有些语言中也有其他途径。例如，在面向方面（aspect-oriented）的语言中，程序员可以编写所谓的“方面”代码，后者能够影响系统中其他地方的代码行为。

我在寻找修改造成的影响时会使用如下的启发式方法：

(1) 确定一个将要修改的方法。

(2) 如果该方法有返回值，查看它的调用方。

(3) 看看该方法是否修改了什么值。是则查看其他使用了这些值的方法，以及使用了这些方法的方法。

(4) 别忘了查看父类和子类，它们也可能使用了这些实例变量和方法。

(5) 查看这些方法的参数，看看你要修改的代码是否使用了某参数对象或它的方法所返回的对象。

(6) 找出到目前为止被你所找出的任何方法修改的全局变量和静态数据。

11.4　进行影响推测的工具

我们最重要的筹码便是对编程语言的认识。每门语言当中都存在所谓的“防火墙”，即能够阻止影响继续传播的语言结构。若能知道这些“防火墙”分别是什么，我们便清楚什么时候不必穿越它们去追溯影响的足迹了。

假设我们想要修改下面这个Coordinate类的表现形式，想要将它泛化成一个能够表示三维和四维坐标的坐标类，为此我们打算改用向量来存储各坐标分量。然而，如果这个类是像下面这样来实现的话，在追踪修改所带来的影响时就无需考虑这个类的代码之外了。

```
public class Coordinate {
    private double x = 0;
    private double y = 0;
```

1. 通常是按引用或传地址。——译者注

```
    public Coordinate() {}
    public Coordinate(double x, double y) {
        this.x = x; this.y = x;
    }
    public double distance(Coordinate other) {
        return Math.sqrt(
            Math.pow(other.x - x, 2.0) + Math.pow(other.y - y, 2.0));
    }
}
```

165

而对于下面这个实现来说就不是这样了：

```
public class Coordinate {
    double x = 0;
    double y = 0;

    public Coordinate() {}
    public Coordinate(double x, double y) {
    this.x = x; this.y = x;
    }
    public double distance(Coordinate other) {
        return Math.sqrt(
            Math.pow(other.x - x, 2.0) + Math.pow(other.y - y, 2.0));
    }
}
```

上面这两个实现的区别很细微。在第一个实现当中，x和y变量是私有的。而在第二个实现里则是包作用域的。所以，对于第一个实现来说，我们对于x和y变量的任何改动只能通过distance()成员方法来对类的客户代码造成影响，不管其客户代码使用的是Coordinate还是它的某个子类。而在第二个实现中就不同了，与Coordinate类位于同一个包内的代码可以直接访问它的x和y成员变量。于是我们就得关心一下这些代码了，可以将这两个变量也设为私有，从而确保没有客户代码能够直接访问到它们。此外，Coordinate的子类也可能会使用到这两个成员变量，因此我们同样不得不照顾到它的所有子类，看看其中有没有使用了x或y的。

对所用语言的了解和把握是十分关键的，一些微妙的语言规则经常会把我们弄得晕头转向。比如下面这个C++类：

```
class PolarCoordinate : public Coordinate {
public:
            PolarCoordinate();
    double getRho() const;
    double getTheta() const;
};
```

在C++中，当const修饰符跟在方法声明的后面时，被声明的方法便不能修改其所属对象的实例变量。但真的是这样吗？假设PolarCoordinate的父类看起来如下：

166

```
class Coordinate {
protected:
    mutable double first, second;
};
```

当C++中的mutable关键字用于修饰一个（变量）声明时，便意味着该变量可以被const方法所修改。不可否认，mutable的这种用法非常古怪，然而当我们面对的是一个不甚了解的程序，需要弄清哪些会改变而哪些不会改变的时候，不管用法多古怪，也得硬起头皮进行影响分析。在C++中不作认真检查而简单地把const当作真正的常性是危险的。像这类能够被“绕过去”的语言特性都要加以注意。

了解你所用的语言。

11.5　从影响分析当中学习

建议你只要一有机会就去分析分析代码中的影响。有时候，在对一个代码基很熟悉了之后，就会明白在做影响分析的时候无需操心某些角落[1]，有这种感觉就意味着你已经发现了代码基所具有的一些“基本品质”。最好的代码中是没有多少“陷阱（gotcha）”的，它们里面所包含的一些“规则”（不管这些“规则”有没有被显式表达出来）使你在寻找可能的影响时不至于钻牛角尖。找出这些“规则”的最佳方式便是，首先设想一个在软件的不同部分之间传递影响的途径，这一影响传递途径必须是你从未在代码基中见到过的，然后对自己说：“不，那样的话就太愚蠢了。”如果代码基中有许多那样的规则的话，你就会发现在其中工作起来要容易得多。在糟糕的代码中人们不知道这些“规则”是什么，或者说即便有所谓“规则”也到处都是“例外”。

上面所谓“规则”并不一定是指编码规范或编程风格（如“决不使用受保护的成员变量”）之类的东西，而通常是一些与实际场景相关的方面。比如在本章开头的CppClass例子当中，我们做了一个小小的练习，试图搞清当一个CppClass对象被创建出来之后哪些动作会对其使用者
167 产生影响。下面就是相关代码的摘录：

```
public class CppClass {
    private String name;
    private List declarations;

    public CppClass(String name, List declarations) {
        this.name = name;
        this.declarations = declarations;
    }
    ...
}
```

我们知道这样一个事实，就是当一个declarations列表被传递给CppClass的构造函数之后，人们可能还会对该列表进行修改。“但那样就太愚蠢了”，这正是刚才提到的规则的理想候选。如果在最初查看CppClass的时候就知道接受到的declarations列表是不会再改变的话，接下去的影响推测便会容易得多了。

总的来说，我们对影响的扩大范围的限制越是严厉，编起程序来就越容易，从而减少理解一

1. 如mutable之于const。——译者注

段代码所需的前提知识。最极端的情况便是使用像Scheme和Haskell这样的函数式编程语言来编程。用这些语言编写的程序有时候真的很容易就能够理解[1]，不过这些语言被使用得并不广泛。不管怎么说，在面向对象语言中，限制影响的作用范围可以令测试变得容易得多，而且并没什么东西阻碍你这么做。

11.6　简化影响结构示意图

这本书讲的是如何令遗留代码更易对付，所以列举的很多例子都有一种“泼出去的水”的感觉，泼出去的水当然是收不回来的，同样，对于早已写成的遗留代码你也别指望能够从头来过了。不过，我希望能够借此机会展示一些能够从影响示意图中看出来的非常有用的东西。它能影响你以后编写代码的方式。

还记得CppClass的影响示意图吗（见图11-11）？ 168

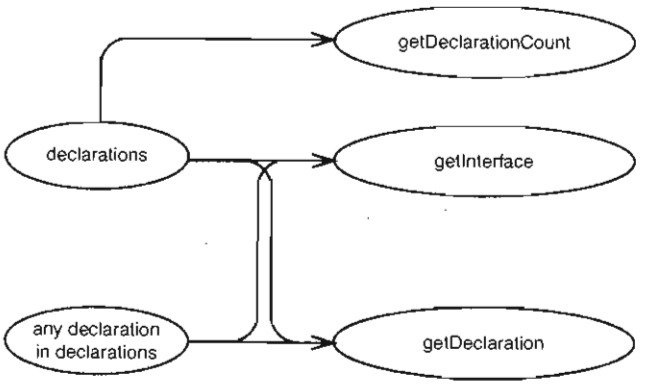


图11-11　CppClass的影响结构示意图

看起来这幅影响结构图有点“散”。两块数据（一个是declarations集合，一个是单个declaration）对好几个不同的方法都有影响。我们可以在这几个方法中选出一个或几个来进行测试。最佳选择就是getInterface，因为它对declarations对象的使用比较全面一点。有些东西我们能够通过getInterface方法轻易感知到的，却并不能同样容易地通过getDeclaration或getDeclarationCount感知到。如果要描述CppClass的话，我并不介意只对getInterface编写测试，但这么一来就很遗憾，不能覆盖到getDeclaration和getDeclarationCount了。然而，如果getInterface的实现像下面这样呢？

```
public String getInterface(String interfaceName, int [] indices) {
    String result = "class " + interfaceName + " {\npublic:\n";
    for (int n = 0; n < indices.length; n++) {
        Declaration virtualFunction = getDeclaration(indices[n]);
```

1. 因为在这类语言中，所有操作都是没有副作用的，所有东西都像Java的String那样是immutable的。——译者注

```
            result += "\t" + virtualFunction.asAbstract() + "\n";
        }
        result += "};\n";
        return result;
    }
```

这里的改动很小：getInterface在内部使用了getDeclaration。因而我们的影响结构图
169 便从图11-12变成了图11-13。

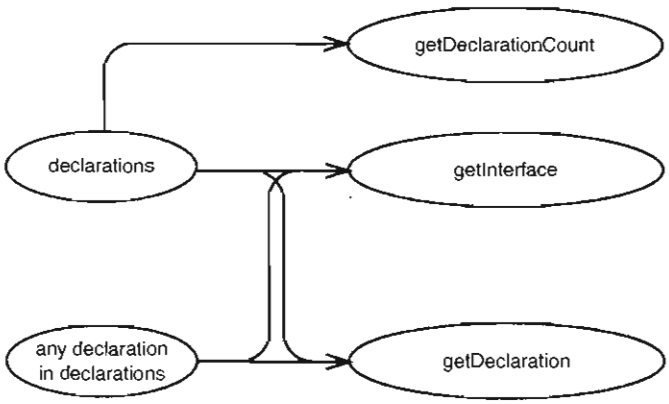


图11-12　CppClass的影响结构示意图

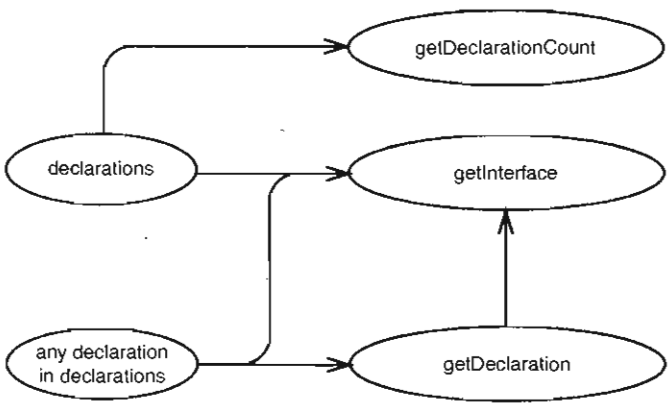


图11-13　修改后的CppClass的影响结构示意图

这只是一个小小的改动，然而带来的影响却是显著的。getInterface方法现在在内部使用
170 了getDeclaration方法，所以在测试getInterface的同时也就连带测试了getDeclaration。

在消除了一点点的代码重复之后，我们往往能够得到一张“终点”更少的影响结构图。而后者则往往能够令你的测试决策更容易。

影响和封装

关于面向对象，一个常常被人们挂在嘴边的好处便是封装。我在向人们展示本书中的一些解依赖技术时，他们常常会指出许多解依赖技术都破坏了封装性。没错，的确如此。

封装的重要性毋庸置疑，然而更重要的是它的重要性背后的原因。封装有助于我们对代码进行推测。跟封装不佳的代码相比，理解封装良好的代码所需要跟踪的路径更少。例如，假设我们给一个构造函数新添一个参数来达到解依赖的目的（参数化构造函数），则在推测影响的时候就可能需要多考虑一条路径了。没错，打破封装会令代码中的影响推测变得更难，然而若是最终我们能够给出具有很好的说明和阐释作用的测试，情况就恰恰相反了。因为一旦一个类有了测试用例，就可以使用这些测试用例来更为直接地进行影响推测了。如果对代码的行为有任何问题，都可以去编写新的测试。

实际上封装与测试覆盖也并不总是冲突的，只不过，当它们真的发生冲突时，我会倾向于选择测试覆盖。通常它能够帮助我实现更多的封装。

而且封装本身也并不是最终目的，而是帮助理解代码的工具。

当需要确定在何处编写测试时，很重要的一点就是要弄清修改会带来哪些影响。为此我们得进行影响推测。这种推测可以是非正式的，也可以稍微严密一些，借助一点草图来进行。但最重要的就是要知道这些工夫花得都是值得的。在特别错综复杂的代码当中，要想将测试安置到位，这是我们所能依赖的为数不多的几项技能之一。 171

第 12 章

在同一地进行多处修改，是否应该将相关的所有类都解依赖

有些情况下，要开始给一个类编写测试还是比较容易的。不过要是遗留代码的话往往就不那么简单了。可能会遇到一些难以解开的依赖。在痛下决心将一批类弄进测试用具从而让以后的日子好过些之后，最令人窝火的事情莫过于却又发现要进行一堆挤在一块儿的修改。你要把一个新特性加进系统，同时发现为此得修改三四个紧密相关的类，其中每一个类都是不花上几个钟头就没指望能放入测试之下的。你心里头当然清楚要是捏着鼻子干完这事儿的话代码肯定会变得容易对付得多，然而，真的必须得一个一个的来解开所有这些依赖吗？那可不一定。

通常有一个办法值得一试，那就是所谓“退一层测试”，“退后一层”，从而找到一个地点能够同时给多处修改编写测试。比如要对一系列私有方法进行修改，只需为某一个公有方法编写测试就行了。或者，要对某个对象所持有的一组互相协作的对象进行测试，我们只需测试前者的接口即可。采用这种方法不仅能够达到覆盖所作修改的目的，还能在代码重构方面提供更大的自由度；在不违反测试所限定的行为的前提下，测试覆盖之下的代码无论怎么改都不要紧。

172
~
173

高层测试对代码重构也是比较有用的。与精细到类的测试相比，人们一般更喜欢较为高层的测试，因为他们觉得接口上如果有一大堆零零碎碎的测试的话改起来就要难一些了。然而实际往往比人们想象得要简单，因为你可以先改测试再改代码，以安全的小步骤来一点一点地改进代码的结构。

不过，高层测试虽说是个重要的工具，但并不能替代单元测试。而是为最终将单元测试安置到位而进行的铺垫。

那么我们到底该如何才能将这些“覆盖测试”安置妥当呢？首先便是确定测试的地点。如果还没有的话，建议你看一看第11章。该章描述了所谓的“影响结构图”，影响结构图是一个强大的工具，可以用它来推断出测试地点。而本章则描述了拦截点（interception point）的概念，并展示了如何寻找到拦截点。此外还描述了在代码中所能找到的最佳拦截点，即所谓的汇点（pinch point）。我会告诉你如何寻找到这些点，以及当你想要编写测试来覆盖将要修改的代码时，寻找

到的这些点将会给你带来什么样的帮助。

12.1　拦截点

给定一处修改，在程序中存在某些点能够探测到该修改的影响，我们把这些点称为拦截点。寻找拦截点的难易程度跟具体的应用程序是有关系的。比如，假设一个应用程序中的各个部件都搅和在一块，没多少自然接缝的话，在其中要想寻找到一个合适的拦截点就相当费工夫，不进行一点影响分析以及解开大量的依赖是别想达到目的的。

最佳切入点莫过于先确定需要进行修改的点，并从这些修改点开始一路向外追踪影响。每个可以探测影响的点都是一个拦截点，但并非每个都是最佳拦截点，在整个过程中你都需要自己进行判断。

12.1.1　简单的情形

现在，假想我们需要修改一个名为Invoice的Java类，目的是要更改该类计算费用的方式。Invoice类上面的那个计算总计费用的方法叫做getValue，如下所示：

174

```
public class Invoice
{
    ...
    public Money getValue() {
        Money total = itemsSum();
        if (billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                    originator.getState().equals("NY"))
                total.add(getLocalShipping());
            else
                total.add(getDefaultShipping());
        }
        else
            total.add(getSpanningShipping());
        total.add(getTax());
        return total;
    }
    ...
}
```

我们需要修改发货到纽约的运输费用的计算方式，因为立法机关刚刚增加了一项税，从而对纽约那边的运输业务造成了影响，而且不幸的是，我们只得把这项费用算到客户头上。首先我们把负责运输费用计算的代码提取到一个新类ShippingPricer当中。如下所示：

```
public class Invoice
{
    public Money getValue() {
        Money total = itemsSum();
        total.add(shippingPricer.getPrice());
        total.add(getTax());
        return total;
```

```
    }
}
```

这么一来，原先由getValue完成的工作现在就全部改由ShippingPricer来完成了。之后我们还得对Invoice的构造函数作一点改动，在里面创建一个知道开票日期的ShippingPricer对象。

要找出拦截点，就得从修改点起一路追踪影响：getValue方法将会返回一个与原先不同的值。我们发现Invoice中并没有其他方法使用getValue，但另一个类却使用了它，那就是
175 BillingStatement类的makeStatement方法。图12-1展示了这一点：

此外我们还要对Invoice的构造函数作改动，所以还得看一看哪些代码是依赖于它的。本例中要进行的改动是在Invoice的构造函数中创建一个ShippingPricer对象。该对象除了影响使用了它的方法之外别无其他影响，而唯一一个使用了它的方法便是getValue。图12-2展示了这一影响：

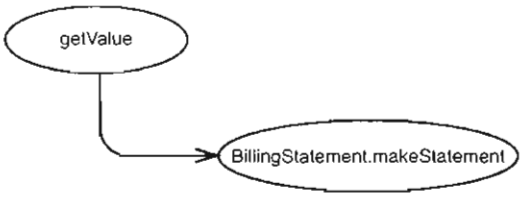


图12-1　getValue影响了BillingStatement.makeStatement

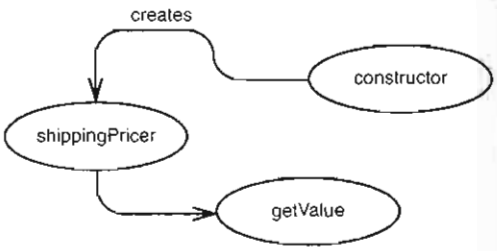


图12-2　对getValue的影响

176 我们可以将上面两幅图合并在一起，结果见图12-3：

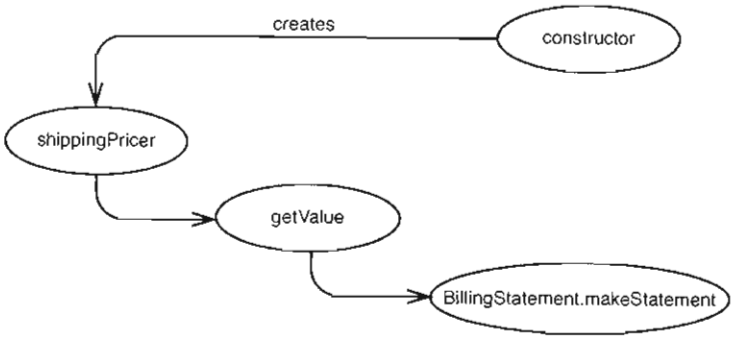


图12-3　影响链

现在的问题是，我们的拦截点在哪里？其实我们可以将上图中的任意一个椭圆节点当作拦截点，当然，前提是我们得对它们所对应的实体（方法/类/变量等）有访问权才行。我们可以尝试通过shippingPricer变量来进行测试，然而它是Invoice类的一个私有变量，所以无法访问。

实际上，就算在测试中可以访问shippingPricer变量，它也只能算是一个相当“窄”的拦截点，通过它可以感知到我们对构造函数所作的改动（创建shippingPricer），并可以确保shippingPricer的行为如我们所预期的那样，但是却无法通过它来确保getValue的改动也是良好的。

我们也可以为BillingStatement的makeStatement方法编写测试，通过检查其返回值来确保修改的正确性，但实际上还有更好的办法，那就是针对Invoice的getValue编写测试；这一方案甚至还更加省事。没错，能够将BillingStatement纳入测试之下固然是件好事，但在目前的情况下其实并没这个必要。等到后面真要修改BillingStatement类时再将它纳入测试也不迟。 177

一般来说拦截点离修改点越近越好。之所以这样说，有如下几方面的原因。首先是安全性。从修改点至拦截点，途中的每一步都好比是逻辑论证过程中的一步。从根本上我们要表达的其实就是这样的话：“我们之所以可以在这儿测试，是因为这儿影响了这儿，后者进而影响了那儿，那儿最终影响了我们所测试的这个东西。”论证过程中的步骤越多，我们就越难判断论证的正确性。有时候唯一能有信心的做法就是在拦截点编写测试，然后回到修改点对代码作一点小小的改动，再观察测试失败了没有。的确，有些情况下你不得不退而采用该技术，但应该不会总是需要这么做。拦截点的选择离修改点越近越好的另一个原因就是在离得较近的地方安置测试通常比较容易一些。但这也并不是绝对的；具体还是要看实际的代码。从修改点至拦截点的步骤越多，安置测试就越难。通常你得在大脑里面模拟代码运行来确定一个测试是否覆盖了某块遥远的功能。

拿本例来说，我们想要对Invoice进行修改，最佳的测试点或许就是getValue。可以在测试用具中创建一个Invoice对象，以各种方式来设置它，然后通过调用getValue（并检查其返回值）来固定住行为，以防被我们的修改所破坏。

12.1.2　高层拦截点

大多数情况下，对于一次修改来说，我们所能找到的最佳拦截点就是被修改类上的一个公共方法。这类拦截点容易寻找，也容易使用，但有时候并非最佳的选择。关于这一点，只要我们把Invoice例子稍微扩展一点就不难看出来。

假设除了修改Invoice的运输费用计算方式之外，还得修改一个名为Item的类，给它添加一个成员变量来记录其运输方式。此外在BillingStatement中还需要给每个托运人配置一个单独的细目分类。图12-4展示了目前的设计的UML图。 178

如果这几个类都没有测试的话，我们可以通过给每个类分别编写测试，并进行所需的修改开始。这种做法当然是可行的，但并非最有效率的。更高效的做法是试着找出一个能够用来刻画这块代码的特征的高层拦截点。这么做的好处有两点：首先我们需要进行的解依赖可能减少了，另外我们的“软件夹钳”所夹住的代码块也更大。有了能够刻画这组类的特征的测试，重构工作也就得到了更多的守护。比如在更改Invoice和Item类的结构时，我们就可以使用

BillingStatement的测试来作为不变式。下面就是为同时刻画BillingStatement、Invoice和Item类而编写的一个启动测试：

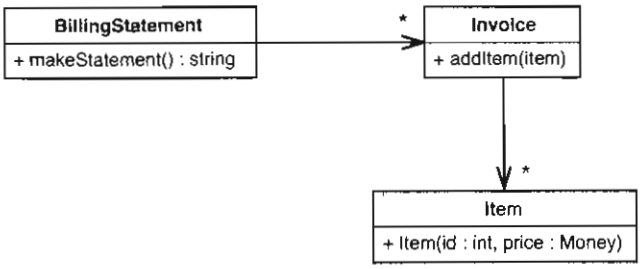
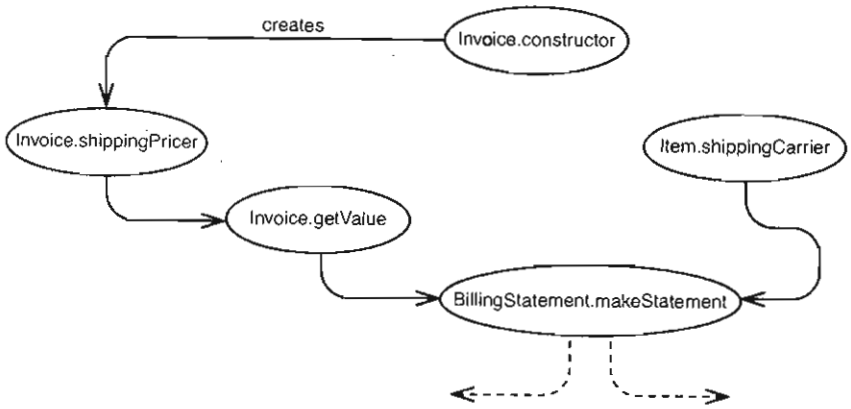


图12-4　扩展的开票系统

```
void testSimpleStatement() {
    Invoice invoice = new Invoice();
    invoice.addItem(new Item(0,new Money(10));
    BillingStatement statement = new BillingStatement();
    statement.addInvoice(invoice);
    assertEquals("", statement.makeStatement());
}
```

如上面的代码，我们可以搞清BillingStatement为只含一项货物的发货单生成的是什么说明文本，并在测试中使用它。接下来，我们可以添加更多的测试，看看对于发货单和货物的不同组合，说明文本的格式如何变化。对于那些将引入接缝的代码块，编写测试用例的时候要格外小心。

那么，使得BillingStatement成了一个理想的拦截点的原因在于，在它这一个点上我们就能够探测到一簇类的修改所造成的影响。图12-5展示了即将进行的修改的影响结构图。



179

图12-5　账单系统的影响结构图

从上图中我们可以看到，一切影响都可以通过makeStatement探测到。或许并不容易，但

至少是可行的，而且别忘了这可是“在单一地点探测所有影响”。在设计中，我把这类地点称作汇点（pinch point）。汇点是影响结构图中的交通要冲，在这类地点编写的测试能够覆盖大量的修改。若能在设计中找到汇点的话，你的工作就会轻松许多。

不过，关于汇点，有一个关键的地方是要记住的，那就是它们是由具体的修改点来决定的。有时候就算一个类有多个客户，对它的一组修改也仍然存在着一个很好的汇点。为了说明这一点，再次回顾一下我们的开票系统，这次我们把视野稍微拉远一点（见图12-6）：

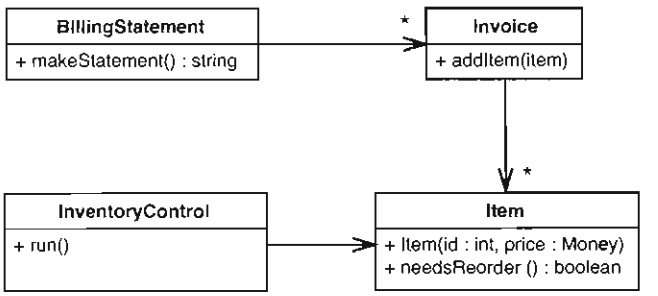


图12-6　加入了详细目录的开票系统

有一点并没有注意到，那就是Item还有一个叫做needsReorder的方法。每当需要了解是否需要进行重新排序时，InventoryControl类便会调用这个方法。那么，就我们刚才所讨论的修改而言，现在有了needsReorder方法的加入，影响结构图会有什么样的变动呢？答案是没有丝毫变动。往Item类中添加shippingCarrier成员变量根本不会影响到needsReorder方法，所以汇点还是原来那个汇点，即BillingStatement。

让我们稍微改变一下场景。假设我们还需要作另一处改动。需要往Item中添加一个方法，以便能够获取/设置货物（Item）的供应商。此外InventoryControl和BillingStatement这两个类会使用供应商的名字。图12-7显示了以上场景对影响结构图造成的影响。

现在事情看来似乎没刚才那么顺利了。我们的修改所产生的影响可以通过BillingStatement的makeStatement方法测得，也可以通过InventoryControl的run方法所影响到的变量测得，但问题是不再存在一个单一的拦截点了。不过，合起来看的话，run和makeStatement这两个方法是可以被看作汇点的：它们加起来也只是两个方法而已，比起我们要进行的修改（需触及8个方法/变量），这还算是比较经济的。如果我们在这两个方法处编写测试，就能覆盖大量的修改工作。 180

汇点

汇点是影响结构图中的瓶口和交通要冲，在汇点处编写测试的好处就是，只需针对少数几个方法编写测试，就能够达到探测大量其他方法的改动的目的。

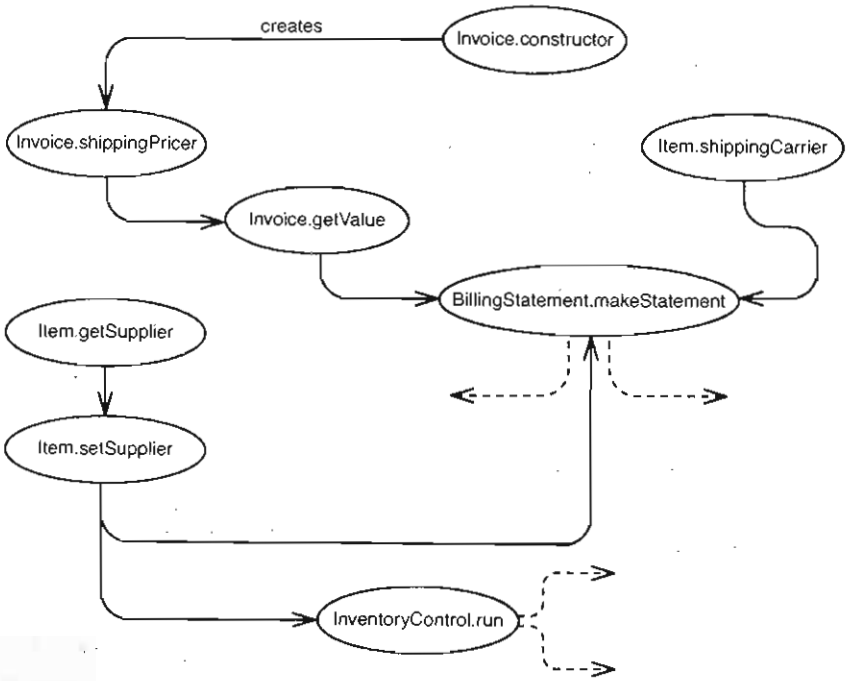


图12-7　账单系统的全景图

对于某些代码基来说，在其中寻找一组修改的汇点是件相当容易的事，但也有很多时候几乎是不可能找到的。某个类或方法可能会直接影响一大堆东西，于是以它为中心延展出来的影响结构图看起来可能就会相当复杂。这时候该怎么办呢？办法之一便是重新考量我们的修改点。问问自己是不是太“贪”了，考虑能否一次仅为其中一两个修改点寻找汇点。最后，要是实在没法找出汇点的话，那就按就近原则直接给每个修改编写测试吧。

181

寻找汇点的另一个办法就是找出方法或类的被使用方式之间的共同之处（第11章中介绍了影响结构图）。例如，某个方法或变量可能会有三个用户，但这并不就意味着这三个用户使用它的方式各不相同。假设我们想对上例中的`Item`类的`needsReorder`方法作一点重构。我没有展示代码，但只要画出影响结构图我们就能看出，`InventoryControl`的`run`和`BillingStatement`的`makeStatement`这两个方法构成一个汇点，但这个汇点已经无法再缩小了。然而一般而言，能否只为这些类当中的一个编写测试呢？对此关键的一个问题就是：“如果破坏该方法，在那个地方能否感知到？”答案取决于该方法是怎样被使用的。如果它在一组对象上的使用方式都是一样的，则只需在其中一处测试即可。建议你跟你的同事一起试试这一分析过程吧。

12.2　通过汇点来判断设计的好坏

上一节我们讨论了汇点在测试中扮演的重要角色，但除了用在测试中之外汇点还有其他的用

处。其在影响结构图中的位置其实暗示了你如何才能让代码变得更好。

那么，到底什么是汇点呢？一个汇点其实就相当于一个自然封装边界。发现一个汇点就相当于发现了一个“漏斗口”，一大块代码的影响都得从这个口经过。就拿我们这个例子来说，如果BillingStatement.makeStatement方法是一堆发货单和货物的汇点的话，我们就知道当账单上列出的内容跟预期不符的时候该到哪去找原因了：问题只可能出在BillingStatement类本身或发货单和货物身上。同样，我们无需知道发货单和货物就可以调用makeStatement。以上这两点就基本体现了封装的精神：无需关心内部，而真的需要关心内部时，无需通过查看外部信息来理解它。我在寻找汇点的时候常常注意到，可以通过在类之间转移职责来达到更佳的封装性。 182

借助影响结构图来发现潜在的类

假设你手头有一个庞大的类，那么就可以借助于影响结构图来发现如何将这个类分解成较小的类。下面是一个Java中的例子。这个名为Parser的Java类有一个叫做parseExpression的公共方法。

```
public class Parser
{
      private Node root;
      private int currentPosition;
      private String stringToParse;
      public void parseExpression(String expression) { .. }
      private Token getToken() { .. }
      private boolean hasMoreTokens() { .. }
}
```

倘若我们为这个类描绘一幅影响结构图的话，就会发现parseExpression依赖于getToken和hasMoreTokens，但并不直接依赖于stringToParse或currentPosition（而getToken和hasMoreTokens则是直接依赖于它俩的）。这儿我们看到了一个自然封装边界，尽管这个边界并不十分狭窄（两个方法隐藏了两块信息）。我们可以把上面提到的这些方法和成员变量提取到一个名为Tokenizer的新类中，从而简化Parser类。

当然，要分离类里面的职责并非只有这一个办法。有时候我们也可以从名字当中获得一些线索，比如刚才这个例子中我们就看到有两个方法的名字中都具有“Token”这个单词。这可以帮助你用另一种眼光来审视一个庞大的类，后者可能进而会启发你完成一些漂亮的提取。

作一个练习，请为一个庞大的类之中的修改勾勒一幅影响结构图，并故意不去管那些椭圆节点的名字，而只是关注它们是怎样联系和聚集在一起的。在这样一种审视方式之下，看看图中的自然封装边界，把目光定位到这个边界内部的椭圆节点上，考虑给这组方法/变量起个什么样的名字，而这个名字就会成为你即将分离出来的新类的名字了。此外考虑是否需要适当修改某个方法/变量的名字。

上面这个练习最好跟你的队友们一起完成。你们就命名问题进行探讨带来的好处不仅止于目前正在做的工作。它们能够帮助你和你的团队形成一个关于“该系统是什么”以及“它能够成为什么”的共识。

要在程序的某部分完成一些侵入性的改动，理想的切入点便是在汇点编写测试。花点工夫将一组类从系统中划分出来，对它们做一点修改，以便可以在测试用具中实例化它们。之后，在你完成了特征测试的编写之后，便可以肆无忌惮地进行修改了。这时你已经在应用程序中制造出了一个“小绿洲”，在这块弹丸之地上你的工作变得相对容易不少。但是，请当心，这片“绿洲”
183 有可能只是一个虚幻的海市蜃楼！

12.3　汇点的陷阱

在编写单元测试的时候我们可能会遇到各式各样的麻烦。其一就是我们的单元测试可能会缓慢但逐步地演化为“迷你型”的集成测试。一般来说情况是这样的，开始的时候我们需要测试一个类，所以实例化了好几个被它用到的类，并将实例化出来的对象传递给该类。我们检查某些值，而且相信这组类互相“合作愉快”。但这种做法的缺点在于，一旦它被过于频繁地使用，最终就有可能演化成庞大而笨重的、不知何年何月才能运行完成的“单元测试”来。要想避免这一点，比如我们在给新代码编写单元测试的时候，就要尽可能单独而孤立地去测试它们。一旦意识到手头的测试已经过于笨重了，就应该去分解被测试类，分解出较容易测试的独立小类来。另外我们还不时需要去伪造被测试类所需要用到的某些对象，因为单元测试的任务并非检查一簇类是否能够合作良好，而是检查单个的对象行为是否正确。通过伪造合作类对象我们就可以更容易地达到这一目的。

然而，以上只是对新编写的代码而言；在给既有代码编写测试时，情况就反过来了。往往比较好的做法是把程序中的某一块切割下来，利用测试来坚固它。当这些测试都安置到位之后，我们就可以更容易地为这块区域内的每个类编写更狭窄的单元测试了。然后，最终当这些单元测试全都完成，原先在汇点编写的测试也就可以功成身退了。

在汇点编写测试就有点像走了几步进入一片森林，然后划一条线，说：“这块地方现在是我的了。”然后，在拥有了这块（代码）区域之后，你就可以通过重构和编写更多的测试来改善它。而随着时间的推移，你将可以扔掉原先在汇点编写的那些测试，而让每个类的单元测试来支持自
184 己的开发工作。

第 13 章 修改时应该怎样写测试

人们在谈论测试的时候所说的通常都是那些被用来寻找bug的测试，而这些测试通常又都是手动测试。对于遗留代码来说，编写自动化的测试来寻找bug常常让人感觉还没有直接运行代码来得高效。如果你有办法直接手动测试遗留代码的话，往往能很快找到bug。缺点是伴随着一次次的代码修改，每次都得从头手动测试一遍。而且坦白地说，事实上人们并不采用这种办法。在我所接触过的团队中，几乎每一个依赖于手动测试的团队最终都远远落在了后面，结果团队的信心大受打击。

别误会，在遗留代码中寻找bug通常并不是问题。从策略上来说，把工夫花在这上面很可能是将力气用错了地方。通常还不如把精力放在如何让你的团队始终能够编写出正确的代码上面。一句话，正确的策略是关心如何才能从一开始就避免让bug进入代码。

自动化测试是一个非常有用的工具，但这并非对寻找bug而言，至少并没有直接的关系。一般而言，自动化测试的任务是明确说明一个我们想要实现的目标，或者试图保持代码中某些既有的行为。在自然的开发流程中，属于前者的测试逐渐就会变成后者[1]。当然，你会遇到bug，但通常并非在某个测试第一次运行的时候，而是在不小心改变了不想改变的行为时，这时运行测试就会指出问题。

那么对于遗留代码而言，这意味着什么呢？对于要在遗留代码中进行的修改，我们可能尚没有任何针对性的测试，于是也就没有办法去验证在修改之后是否有什么行为被破坏了。所以，最好就是把我们想要修改的那一块区域先用测试给罩起来，像安全网那样。然后，在修改的过程中我们会发现bug，并解决它们，但是对于大多数遗留代码来说，若是我们将寻找和修正所有bug当成目标的话，则永远也不会有做完的一天。

13.1 特征测试

好吧，我们需要测试，但问题是如何编写它们呢？办法之一便是先搞清你的软件应当能做什么，然后基于你所获得的认识去编写测试。我们可以把那些落了灰的需求文档和项目备忘录翻出来，努力从中挖掘出我们想要的信息，之后便坐下来开始编写测试。这的确是个办法，但并不算

1. 一旦某个测试所定义的开发目标实现了，该测试也就成为保持既有行为的测试了。——译者注

很好。因为对于几乎所有的遗留系统而言，更重要的不是“系统应该能够做些什么”，而是“系统当前能够做些什么”。所以如果我们基于从文档当中发掘出来的关于“系统应该能够做些什么”的假设来编写测试的话，就又回到了寻找bug的老路上了。寻找bug的确很重要，但我们当前的目标是把测试安置到位，从而减少代码修改过程当中的不确定性。

我把用于行为保持的测试称为特征测试（Characterization test）。特征测试刻画了一块代码的实际行为。而不是“嗯……这块代码应该具有这一行为”或者“我想它会那样的吧”。特征测试描述了系统当前的实际行为。

以下是编写特征测试的几个步骤：

(1) 在测试用具中使用目标代码块。

(2) 编写一个你知道会失败的断言。

(3) 从断言的失败中得知代码的行为。

(4) 修改你的测试，让它预期目标代码的实际行为。

(5) 重复上述步骤。

在下面这个例子中，我相当确信一个PageGenerator对象不会生成字符串"fred"，所以我写下一个断言：

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    assertEquals("fred", generator.generate());
}
```

运行测试，看它是否失败。一旦果真失败了，你也就明确地知道了代码当前在那种情况下的实际行为。例如在上面的代码中，一个新建的PageGenerator对象在它的generate方法被调用
186 时返回了一个空串：

```
 .F
Time: 0.01
There was 1 failure:
1) testGenerator(PageGeneratorTest)
junit.framework.ComparisonFailure: expected:<fred> but was:<>
     at PageGeneratorTest.testGenerator
         (PageGeneratorTest.java:9)
     at sun.reflect.NativeMethodAccessorImpl.invoke0
         (Native Method)
     at sun.reflect.NativeMethodAccessorImpl.invoke
         (NativeMethodAccessorImpl.java:39)
     at sun.reflect.DelegatingMethodAccessorImpl.invoke
         (DelegatingMethodAccessorImpl.java:25)

FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

我们可以修改测试从而让它能够通过：

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    assertEquals("", generator.generate());
```

```
}
```

现在测试通过了。而且，不仅是通过，它还起到了描述PageGenerator的一个最基本行为的作用，这个最基本的行为就是：如果我们创建一个PageGenerator并立即调用它的generate方法的话，就会得到一个空串。

可以使用同样的技巧来查明当我们给PageGenerator提供其他数据的时候会生成什么：

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    generator.assoc(RowMappings.getRow(Page.BASE_ROW));
    assertEquals("fred", generator.generate());
}
```

对于上面的测试，测试用具给出的出错信息告诉我们结果串是"<node><carry>1.1 vectrai</carry></node>"，于是我们可以把这个串填入测试当中所期望的结果串那儿。

```
void testGenerator() {
    PageGenerator generator = new PageGenerator();
    assertEquals("<node><carry>1.1 vectrai</carry></node>",
            generator.generate());
}
```

不过，如果你已经习惯了把这些测试也当作测试的话，就会发现这种做法的一些很奇怪的地方。比如，既然我们只是把代码产生的实际结果填入测试，那这些所谓的测试就没任何意义了。 187
要是代码本身就有bug的话，那我们填入测试的那些期望值岂不很可能都是错的？

然而，如果我们换个角度的话，这个问题就消失了。我们不把它们看成软件必须遵循的黄金准则，因为我们并不是为了寻找bug，我们是想设置一个机制以便于以后寻找bug。注意，这里所说的bug是指后面可能出现的、与当前系统行为不一致的行为。采用了这一视角之后，我们看待这类测试的方式也就相应发生了变化：它们不再是一个个的准则，而是描述了系统各部分的实际行为。一旦我们知道系统某部分的实际行为，结合之前对于系统“应该具有的行为”的认识，就可以明智地作出如何修改的决策。事实上，了解系统中某部分的实际行为是非常重要的。我们通常可以通过与其他人交流或者通过计算知道哪些行为是需要添加的，但若是少了（特征）测试的话，便没法知道系统当前的实际行为是什么了，当然，除非你在每次需要判断系统实际行为的时候都能够边读代码边在脑子里“运行”出结果来。对于后一种方式，有些人比较拿手，而有些人则不是，关键是不管我们“运行”得有多快，一遍遍地做这件事仍是相当乏味和浪费精力的。

> 特征测试描述了一块代码的实际行为。在编写特征测试的时候如果发现某些结果与我们所期望的不一致，最好弄清它。因为我们遇到的可能是个bug。但这并不是说我们就不能把该测试放进测试套装中，而是说我们应该将它标记为可疑的，并搞清修正它会带来哪些影响。

关于特征测试，目前我们所讲到的还只是很少一部分。比如在前面的PageGenerator例子当中，看起来当时我们就好像只是很随便地扔一个值给测试对象，然后通过断言来查看得到什么结果。当然，如果我们对代码所应当具有的行为有一个良好的把握的话，是可以这么做的。有些

情况下，像一开始我们对PageGenerator所做的，只是创建一个对象，然后立即调用它的方法查看结果，这一工作概念简单，而且也的确值得编写特征测试，但问题是，接下来我们该做什么呢？还是拿PageGenerator例子来说，像这样一个类，我们到底可以为它编写多少（特征）测试呢？答案是，无穷多。花上十年八年也写不完。那么，什么时候应该停止呢？有什么办法可以告诉我们其中哪些测试更重要呢？

要想解答这个问题，关键的一点就是要意识到我们并不是在编写黑盒测试。换句话说，我们在编写特征测试的时候是可以去查看所要刻画的代码的。代码本身能够告诉我们它们的行为，而如果看了代码还不能肯定的话，理想的办法就是编写测试去“询问”它们了。编写特征测试的第一步就是让自己对目标代码的行为感到好奇，在这个阶段我们不断编写测试直到感到已经理解了
188 代码。但这样我们的测试就能保证覆盖了代码的所有方面吗？还有第二步，就是设法弄清我们的修改如果引入了bug的话测试能否“感应”得到。如果存在可能的漏网之鱼，就要添加更多的测试，直到无遗漏为止。而如果我们没有这么大的信心，安全一点的办法就是考虑换一种方式来修改代码。或许我们可以再返回到第一步看看。

使用方法的规则

当准备在遗留系统中使用一个方法之前，请查看一下是否已有针对它的测试。没有的话就自己写一个。始终保持这一习惯，你的测试就能起到信息传递媒介的作用。别人只要一看到你的测试就能够知道对于某方法他们该期望什么而不该期望什么。试图使一个类变得可测试这一行为本身往往能够改善代码的质量。于是人们能够发现什么是可行的，以及是如何可行的，他们可以进行修改，更正bug，然后继续前进。

13.2　刻画类

假设现在有一个类，我们想要知道应该测试哪些东西。第一件事就是要从较高的层面上来领会该类会做些什么。比如我们可以先针对能想到的最简单的行为来编写几个测试，然后把任务交给我们的好奇心，让好奇心领着我们往前走。下面是几个有益的启发式方法：

(1) 寻找代码中逻辑复杂的部分。如果你不理解某块代码，可以考虑引入感知变量（239页）来刻画它。利用感知变量来确保代码中的某些特定的区域被执行到了。

(2) 随着你不断发现类或方法的一个个职责，不时停下来把你认为可能出错的地方列一个单子。看看能不能编写出能够触发这些问题的测试。

(3) 考虑你在测试中提供的输入。如果故意把输入的值变得极端化会出现什么情况呢？

(4) 对于某个类的对象，有没有某些条件在它的整个生命周期当中都是成立的？通常这被人们称为不变式（invariant）。尝试编写测试去验证它们。通常你可能需要重构才能够发现这些不变式。但重构往往能够给你带来关于“代码应该怎样”的新认识。

我们编写出来的用于刻画代码的测试是非常重要的。它们描述了系统实际的行为。和编写任何文档一样，你在编写特征测试的时候也得考虑哪些东西对于阅读它们的人来说是重要的。换位

思考，设想自己就是阅读它们的人，在面对一个以前从未见过的类的情况下，你想要知道关于它的什么信息呢？你又想以什么样的顺序来获得这些信息？如果使用的是xUnit框架，那么测试在文件中的存在形式就是一个个的方法。可以通过调整它们的顺序来让别人更容易了解他们所面对 189
的代码。一开始可以放置一些简单的、用于说明目标类的主要意图的测试用例，然后再放置一些突出该类与众不同的地方的用例。确保你所发现的那些重要的地方都以测试的形式呈现出来了。后面当你开始修改代码时，你往往就会发现前面编写的测试恰恰非常有利于将要进行的工作。不管是有意还是无意，事实就是我们将要进行的修改通常支配了我们的好奇心。

发现bug时……

在刻画遗留代码的整个过程中你都会不时发现一些bug。只要是遗留代码就免不了有bug，通常bug的量是跟它（遗留代码）被理解的程度成直接的比例关系的。那么，当发现bug时，又该做些什么呢？

答案要视具体情况而定。倘若系统尚未被部署，则答案很简单：修正bug。否则就需要调查调查，看看是否已经有用户依赖于系统的当前行为（甚至在你看来是bug的行为）了。对于后一种情况，通常得花上一点时间来分析如何才能在不导致连锁反应的前提下修正一个bug。

我比较倾向于一旦发现bug就尽早修正它们。如果一个行为很明显就是错的，那就该进行修正。如果你觉得某个行为有问题，则可以把相应的测试代码标成可疑的，并逐渐提升其优先级。尽早查明它是否为bug以及最好如何处置它。

13.3　目标测试

在编写了测试来理解一块代码之后，接下来的工作就是看看我们的测试是否真的覆盖了想要修改的地方。下面就是一个例子，FuelShare是一个Java类，它上面有一个用于计算出租燃料罐中燃料总价值的方法：

```
public class FuelShare
{
    private long cost = 0;
    private double corpBase = 12.0;
    private ZonedHawthorneLease lease;
    ...
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < Lease.CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```

190

我们想要对FuelShare类作一处非常直接的改动，甚至已经为它写好了相应的测试。我们想要进行的修改是这样的：将顶层if语句提取到一个新的方法中，然后将这个新方法移到一个名叫ZonedHawthorneLease的类当中。FuelShare类里面的实例变量lease便是该类的一个对象。

我们可以先设想一下代码在重构之后的样子，如下：

```
public class FuelShare
{
    public void addReading(int gallons, Date readingDate){
        cost += lease.computeValue(gallons,
                                   priceForGallons(gallons));
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}

public class ZonedHawthorneLease extends Lease
{
    public long computeValue(int gallons, long totalPrice) {
        long cost = 0;
        if (lease.isMonthly()) {
            if (gallons < Lease.CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * totalPrice;
        }
        return cost;
    }
    ...
}
```

那么，要想确保正确进行这些重构，需要哪些测试呢？有一点毫无疑问，那就是我们肯定不会去修改以下的逻辑：

```
if (gallons < Lease.CORP_MIN)
    cost += corpBase;
```

因此我们可以用一个测试来检查当加仑数小于Lease.CORP_MIN时计算出的值，不过严格来
191 说这个测试也不是必须的。另一方面，下面的else分支将会遭到修改：

```
else
    valueInCents += 1.2 * priceForGallons(gallons);
```

以上代码到了新方法中就会变成这样：

```
else
    valueInCents += 1.2 * totalPrice;
```

这个改动虽说不大，但仍然还是个改动。所以最好能够确保我们的测试能覆盖到这个else分支。让我们再次回顾一下原来的那个方法：

```
public class FuelShare
{
```

```
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
        ...
        lease.postReading(readingDate, gallons);
    }
...
}
```

从原先的代码中可以看出，只要能够建立一个按月度的FuelShare，并以大于Lease.CORP_MIN的加仑数来调用addReading的话，就能触及那个else分支了，以下就是测试代码：

```
public void testValueForGallonsMoreThanCorpMin() {
    StandardLease lease = new  StandardLease(Lease.MONTHLY);
    FuelShare share = new FuelShare(lease);

    share.addReading(FuelShare.CORP_MIN +1, new Date());
    assertEquals(12, share.getCost());
}
```

在为代码分支编写测试时，应该考虑除了那个分支被执行之外是否还存在其他能令测试通过的条件。如果不确定的话，可以使用一个感知变量或调试器来确定你的测试是否恰好命中目标。

像上面这样来刻画代码分支的时候有一件重要的事情必须弄清楚，那就是你所提供的输入会不会反而导致本该失败的测试成功了。下面就是一个例子。假设代码中改用double型变量来表示金额（原来是用整型）：

192

```
public class FuelShare
{
    private double cost = 0.0;
    ...
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```

这下我们就可能会遇到严重的麻烦了。哦，别误会，这里所说的严重的麻烦并不是指程序中

可能会到处发生的小数部分精度丧失（浮点数舍入误差所致）。而是指除非我们小心选择测试输入数据，否则在提取方法的时候就可能会犯错，而且永远也不知道自己错在哪。比如，一个可能的错误就是我们提取出一个方法并按整型而不是双精度型来接受参数。在Java以及许多其他语言当中都允许从双精度到整型的隐式转换；运行时会把值截断[1]。所以除非我们想办法设计出能够显式导致这些错误的测试数据来，否则错误就被隐藏起来了。

让我们来看一个具体的例子。假设Lease.COPR_MIN的值是10而corpBase是12.0，那么运行下面的测试代码会出现什么结果呢？

```
public void testValue () {
    StandardLease lease = new  StandardLease(Lease.MONTHLY);
    FuelShare share = new FuelShare(lease);

    share.addReading(1, new Date());
    assertEquals(12, share.getCost());
}
```

由于1小于10，所以12.0被加到cost的初始值0上面，结果是12.0。这一切都没有问题，但如果我们像下面这样来提取方法，并将cost变量的类型改为long：

```
public class ZonedHawthorneLease
{
    public long computeValue(int gallons, long totalPrice) {
        long cost = 0;
193     if (lease.isMonthly()) {
            if (gallons < CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * totalPrice;
        }
        return cost;
    }
}
```

这样改了之后，尽管返回的cost值被截断了，但测试依然能够通过。代码中存在从double到long的隐式转换，但这一点并没有被测试覆盖到。因为如果我们仅仅是把整型赋给整型，那么隐式转换发生与否对结果是没有影响的。

> 重构的时候我们通常需要关心两件事情：一是目标行为在重构之后是否仍然存在，二是它是否正确“连接”在系统当中。
>
> 许多特征测试就像定心丸一样。它们不去测试大量的特殊情况，而只是检验某些特定的行为是否存在。在一番移动或提取代码的重构之后，只要看到这些行为仍然存在，我们就可以放心地告诉自己：我们的重构保持了行为。

这个问题的解决有一些一般性的策略。其中之一便是手动计算你期望从某段代码那儿得到的

1. 这当然会首先以编译警告的形式表现出来。——译者注

值[1],在每一个存在类型转换的地方留意是否可能存在截断问题。另一种做法则是使用调试器，跟踪赋值运算，从而知道某组特定的输入会导致什么转换。第三种做法是使用感知变量来确认某条特定的代码路径被覆盖到了且目标转换也被测试到了。

最有价值的特征测试覆盖某条特定的代码路径并检查这条路径上的每个转换。

实际上还有第四种做法。那就是刻画一块更小的代码。如果手头有一个重构工具能够帮我们安全地提取方法的话，就可以将computeValue方法切分开来，并分别为切分出来的小块编写测试。然而遗憾的是并非所有的语言都有重构工具，而且有时候就算有也不一定会按照你设想的方式来提取方法。 194

重构工具的怪癖

一个好的重构工具可以带来极大的帮助，然而即便有了这些工具人们还是不时求助于手动重构。下面就是一种常见的情形。我们想要将A类的b()方法内的某些代码提取出来：

```
public class A
{
    int x = 1;
    public void b() {
        int y = 0;
        int c = x + y;
    }
};
```

如果想要从b()方法当中将x+y表达式提取出来形成一个新的方法add，我们会发现市面上至少有一种重构工具提取出的add是单参的add(y)而不是双参的add(x,y)。因为x是个实例变量，它对于我们提取出的每一个方法都是可访问的。

13.4　编写特征测试的启发式方法

(1) 为准备修改的代码区域编写测试，尽量编写用例，直到觉得你已经理解了那块代码的行为。

(2) 之后再开始考虑你所要进行的修改，并针对修改编写测试。

(3) 如果想要提取或转移某些功能，那就编写测试来验证这些行为的存在性和一致性，一种情况一种情况地编写。确认你的测试覆盖到了将被转移的代码，确认这些代码被正确连接在系统中。最后别忘了测试类型转换。 195

1. 即不是让程序运行给出，而是自己看着代码去算。——译者注

第 14 章

棘手的库依赖问题

14

给开发带来实际帮助的技术之一就是代码复用。如果购买到一个能够替我们解决某些问题的库（并了解如何使用它），项目的耗时往往会大大缩短。这种做法唯一的问题就是，很容易就会变得对某个库过分依赖。如果在代码中不分青红皂白到处乱用一气的话，结果差不多铁定就是陷进泥潭了。一些我接触过的团队的确曾被库依赖问题弄得焦头烂额。比如有这么一种情况，库供应商把版权税提得太高，以致于使用它的软件都无法盈利了。而另一方面，软件的团队又无法使用其他供应商的库，因为要把代码中的所有那些对当前库的调用都分离出来还不如整个儿重写。

> 尽量避免在你的代码中到处出现对库的直接调用。你可能会觉得永远也不会需要去修改这些调用，但最终可能只是自欺欺人。

在我写这本书的时候，开发者们的阵营呈Java/.NET两极分化之势。微软和Sun都试图把它们的平台尽量做大做宽，他们创建了数不清的库来吸引人们继续留在他们的平台上。从某种程度上来说对于许多项目这都不是件坏事，但你仍旧可能会过分依赖于特定的库。每一处以硬编码方式来直接使用类库的地方其实都可以以接缝的形式来实现。有些库在给其中的具体类定义接口方面做得较好，而有些库则不仅做不到这点，还把具体类做成final或sealed的，又或者是具有一些非虚的关键函数，让你没法在测试中“伪造”它们。遇到这类情况，你也只能给这种类写一个对应的外覆类了。而且别忘了发邮件给你的库供应商，抱怨他们的库给你们的开发带来了多大的麻烦。

> 借助于语言特性来施加设计约束的库设计者们往往是犯了一个错误。他们忘记了根本的一条，那就是好的代码除了要能在产品环境中运行之外，还要能在测试环境中运行。然而针对产品环境而施加在代码上的约束则常常会导致代码在测试环境中寸步难行。

196
~
197

实际上，在意图实现良好设计的语言特性与代码的易测试性之间有一条鸿沟。其中最普遍的一个问题就是所谓的“一次性困境”：如果一个库假定某个类在系统中只会出现一个实例，则后面就难对这个类使用伪对象手法了。像引入静态设置方法（292页）或其他许多原本可用来对付单件的解依赖技术或许也派不上用场了。于是把那个单件用一个外覆类包装起来可能就成了你唯一的选择。

另一个有关的问题就是“重写限制困境”。在有些面向对象语言当中，所有的类方法都是虚方法。还有一些语言则让类方法在默认情况下是虚的，但同时也提供途径让你可以把一个方法设置为非虚的。其余那些语言，则是要求你必须显式指定某方法是虚的（否则它们就会默认为非虚）。

从设计的角度来说，有时将一个方法做成非虚方法是有意义的。比如我们就不时会听到来自业界的一些声音，建议最好尽可能把方法设成非虚的。有时候他们给出的理由也的确不错，但我们同样也得承认，这么做使得往代码中引入感知和分离变得困难了，另外不可否认的是，使用Smalltalk的程序员也写出了很多很好的代码，而我们知道Smalltalk中是没有非虚方法的。同样，Java虽然提供了阻止重写的语言手段，但Java程序员通常不用它，可Java程序员也写出了很多很好的代码。就连在C++里面也是，大量代码并没有遵循这个原则，但也都很好。事实上，你完全可以在产品代码中把那些公有方法都“当成”是非虚的[1]，这么一来，你就可以有选择地在测试中重写它们，实现测试和产品两不误。

有时候使用编码惯例并不比使用某种限制性的语言特性差。你得为自己的测试考虑考虑。 198

1. 而实际上它们是虚的。——译者注

第 15 章 到处都是API调用

15

要么买，要么借，要么就自己开发一个。这是每个软件开发者都需要面对的选择。很多时候我们在做一个应用时会认为从其他地方买来一些库，或者利用开源库，或者甚至只使用平台（J2EE，.NET等）自带库里面的代码，会节省一些时间和精力。而事实上，如果想去整合那些你无权去更改的代码的话，有很多东西是需要加以考虑的。比如我们得知道它有多稳定，它是否能满足我们的要求以及易用性如何，等等。最终当我们确定了要使用其他人编写的代码时，还将面临一个问题，那就是我们的应用看起来就好像除了到处调用其他人写的库之外自己就没干什么事情。这样的代码我们该怎么修改？

你可能立即会想到，我们不需要什么测试。毕竟我们自己写的代码几乎没干什么重活，所有的重要工作都丢给库去做了，我们的代码非常简单。这么简单的代码里面能有什么错误呢？

然而事实上，很多遗留项目正是从这样一种微不足道的阶段开始的。随着项目的修改，代码也会不断增长，然后事情就会逐渐变得不那么简单了。一段时间之后，我们也许还能看到某些没有调用库API的代码块，但它们就算存在，也是被包夹在重重根本不可测试的代码之间。于是每次我们改点东西都不得不重新运行程序来确认它仍然还能运行，这也就意味着我们又回到了遗留系统程序员所面对的主要困境：在不能确定是否会破坏什么行为的情况下修改；我们并没有编写所有的代码，但却得维护它。

从许多方面来讲，到处都是库调用的系统比完全自己编写的系统还难对付。其首要的原因就是，对于这种系统你很难看出如何才能让代码的结构变得好起来，因为一眼望过去，到处都是API调用。看不到任何可以从中引出设计的东西。API密集的系统之所以难以对付的另一个原因就是我们并不拥有那些API。如果那些API归我们管的话，就可以通过重命名接口、类以及方法来让系统更清楚一些，还可以给类里面增加方法从而让它们在代码中的不同部分都可用。

下面就是一个例子。这是一段写得非常糟糕的代码，我们甚至根本就不清楚它是否能运行：

```
import java.io.IOException;
import java.util.Properties;

import javax.mail.*;
import javax.mail.internet.*;


public class MailingListServer
```

```
{
  public static final String SUBJECT_MARKER = "[list]";
  public static final String LOOP_HEADER = "X-Loop";

  public static void main (String [] args) {
    if (args.length != 8) {
      System.err.println ("Usage: java MailingList <popHost> " +
          "<smtpHost> <pop3user> <pop3password> " +
          "<smtpuser> <smtppassword> <listname> " +
          "<relayinterval>");
      return;
    }

    HostInformation host = new HostInformation (
          args [0], args [1], args [2], args [3],
          args [4], args [5]);
    String listAddress = args[6];
    int interval = new Integer (args [7]).intValue ();
    Roster roster = null;
    try {
      roster = new FileRoster("roster.txt");
    } catch (Exception e) {
      System.err.println ("unable to open roster.txt");
      return;
    }
    try {
      do {
        try {
          Properties properties = System.getProperties ();
          Session session = Session.getDefaultInstance (
              properties, null);
          Store store = session.getStore ("pop3");
          store.connect (host.pop3Host, -1,
              host.pop3User, host.pop3Password);
          Folder defaultFolder = store.getDefaultFolder();
          if (defaultFolder == null) {
            System.err.println("Unable to open default folder");
            return;
          }
          Folder folder = defaultFolder.getFolder ("INBOX");
          if (folder == null) {
            System.err.println("Unable to get: "
                  + defaultFolder);
            return;
          }
          folder.open (Folder.READ_WRITE);
          process(host, listAddress, roster, session,
              store, folder);
        } catch (Exception e) {
          System.err.println(e);
          System.err.println ("(retrying mail check)");
        }
        System.err.print (".");
        try { Thread.sleep (interval * 1000); }
```

200

```
        catch (InterruptedException e) {}
      } while (true);
    }
    catch (Exception e) {
      e.printStackTrace ();
    }
  }

  private static void process(
      HostInformation host, String listAddress, Roster roster,
      Session session,Store store, Folder folder)
          throws MessagingException {
    try {
      if (folder.getMessageCount() != 0) {
        Message[] messages = folder.getMessages ();
        doMessage(host, listAddress, roster, session,
            folder, messages);
      }
    } catch (Exception e) {
      System.err.println ("message handling error");
      e.printStackTrace (System.err);
    }
    finally {
      folder.close (true);
      store.close ();
    }
  }

  private static void doMessage(
          HostInformation host,
          String listAddress,
          Roster roster,
          Session session,
          Folder folder,
          Message[] messages) throws
              MessagingException, AddressException, IOException,
              NoSuchProviderException {
    FetchProfile fp = new FetchProfile ();
201 fp.add (FetchProfile.Item.ENVELOPE);
    fp.add (FetchProfile.Item.FLAGS);
    fp.add ("X-Mailer");
    folder.fetch (messages, fp);
    for (int i = 0; i < messages.length; i++) {
      Message message = messages [i];
      if (message.getFlags ().contains (Flags.Flag.DELETED))
        continue;
      System.out.println("message received: "
          + message.getSubject ());
      if (!roster.containsOneOf (message.getFrom ()))
        continue;
      MimeMessage forward = new MimeMessage (session);
      InternetAddress result = null;
      Address [] fromAddress = message.getFrom ();
      if (fromAddress != null && fromAddress.length > 0)
```

```
            result =
                new InternetAddress (fromAddress [0].toString ());
          InternetAddress from = result;
          forward.setFrom (from);
          forward.setReplyTo (new Address [] {
            new InternetAddress (listAddress) });
          forward.addRecipients (Message.RecipientType.TO,
              listAddress);
          forward.addRecipients (Message.RecipientType.BCC,
            roster.getAddresses ());
          String subject = message.getSubject();
          if (-1 == message.getSubject().indexOf (SUBJECT_MARKER))
            subject = SUBJECT_MARKER + " " + message.getSubject();
          forward.setSubject (subject);
          forward.setSentDate (message.getSentDate ());
          forward.addHeader (LOOP_HEADER, listAddress);
          Object content = message.getContent ();
          if (content instanceof Multipart)
            forward.setContent ((Multipart)content);
          else
            forward.setText ((String)content);

          Properties props = new Properties ();
          props.put ("mail.smtp.host", host.smtpHost);

          Session smtpSession =
              Session.getDefaultInstance (props, null);
          Transport transport = smtpSession.getTransport ("smtp");
          transport.connect (host.smtpHost,
              host.smtpUser, host.smtpPassword);
          transport.sendMessage (forward, roster.getAddresses ());
          message.setFlag (Flags.Flag.DELETED, true);
        }
      }
    }
```

202

这段代码算是相当短的了，然而非常不清晰。几乎没有一行代码没有调用API。这段代码的结构可以改善吗？其结构可以改善到能够令修改变得更容易吗？

是的，我可以做到。

第一步就是找出代码的计算核心：这段代码到底做了什么？

对于以上这个问题，最好给出一个简洁的答案：

这段代码从命令行读进配置信息，并从一个文件中读进一组电子邮件地址。然后它会周期性地检查邮件。如果收到新邮件就会把该邮件转发给刚才的那组电子邮件地址（即转发给其中的每一个邮件地址）。

这么看来这个程序似乎主要就是在输入和输出，但实际上除此之外还有一点遗漏掉了：代码中启动了一个线程，这个线程会进入休眠并周期性地醒来检查邮件。此外这段代码并不仅仅是把邮件收进发出就算了，它会基于收到的邮件生成新的邮件，它得设置新邮件的所有字段；检查并

修改邮件的标题，从而让它能够反映这封邮件是来自邮件列表这一事实。所以说这段代码并不是个空壳子。

现在，如果我们想要把代码内的职责进行分离，就可能会得到这样的结果：

(1) 一个模块负责收邮件，并转给我们的系统。

(2) 一个模块负责发送邮件。

(3) 一个模块负责基于收到的邮件来生成发给邮件列表中的每个成员的邮件。

(4) 一个模块负责周期性地醒来看有没有新邮件。

好，我们来看看上面这几个职责。里面有哪些职责跟Java Mail API结合得比较紧呢？职责1和2很显然跟mail API是分不开的了。职责3有点微妙：我们所需的邮件消息类是mail API的一部分，但我们应该可以通过简单的邮件消息来单独测试该职责。职责4跟mail一点关系也没有：只需要一个被设置好的周期性醒来的线程就可以了。

203 图15-1展示了根据上面的职责分离进行的一点设计。

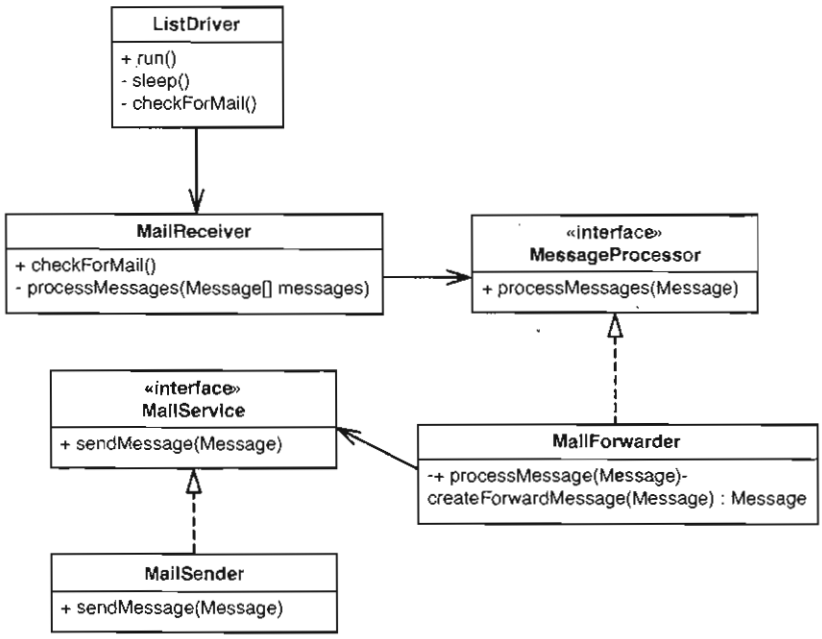


图15-1　一个更好的邮件列表系统

其中ListDriver负责驱动系统。它有一个会周期性醒来检查新邮件的线程，它把实际接收邮件的任务交给MailReceiver来完成。MailReceiver收到邮件后会把邮件一封封的交给MessageForwarder。而MessageForwarder则会为邮件列表中的每一个成员创建邮件消息并逐一交给MailSender发送。

这个设计已经相当不错了。MessageProcessor和MailService接口很方便，因为可以借助于它们来单独测试我们的类。尤其漂亮的就是我们可以在测试用具里面使用MessageForwarder

同时又不用实际发出任何邮件。这可以通过一个实现了MailService接口的FakeMailSender类来轻易完成。

几乎每个系统当中都有一些核心逻辑是可以与API调用分离开来的。尽管本章给出的这个程序规模不大，但它的糟糕程度其实要比大多数程序更甚。MessageForwarder就是一个与接收和发送邮件机制几乎完全没关系的部件，但它还是使用了Java Mail API中的邮件消息类。这几似乎并没有太多的空间留给旧式的Java类。无论如何，把系统分解成上图所示的四个类和两个接口的确给了我们一些层次分离。邮件列表的主要逻辑位于MessageForwarder类当中，而且我们可以 204
把这个类置于测试之下。而原先它却是被埋在一堆乱糟糟的API调用间够都够不着的。所以说，不弄出一些抽象层次来，系统是基本无法被分解成小块的。

如果我们面对的是一个看上去除了API调用什么也没有的系统，一个好的做法就是将它看成一个大大的对象，然后采用第20章所讲的启发式方法来对它进行职责分离。或许我们没法一步就得到更好的设计，但光是找出系统内的职责就已经能够帮助我们在后面的过程中做出更好的决策了。

刚才我们看到了一个更好的设计，但别忘了回到现实：后面该怎么办？从根本上来说有两个办法：

(1) *剥离并外覆API*（Skin and Wrap the API）；

(2) *基于职责的提取*。

所谓的*剥离并外覆API*，其实就是先编写能够尽量准确对应API的接口，然后再为库类创建外覆类。为了把犯错的可能性降到最低，建议你在这个过程中运用签名保持（249页）手法。剥离并外覆API的另一个好处就是我们最终可能会完全不依赖于底层的API代码。在产品类当中，我们的外覆类会负责把调用转发至实际的API，而在测试中，则可以使用伪对象手法。

那么，对于前面讲的那个邮件列表的例子，可以利用这个技术吗？

我们先来回顾一下那个邮件列表服务器的代码中负责实际发送邮件的部分：

```
...
Session smtpSession = Session.getDefaultInstance (props, null);
Transport transport = smtpSession.getTransport ("smtp");
transport.connect (host.smtpHost, host.smtpUser,
    host.smtpPassword);
transport.sendMessage (forward, roster.getAddresses ());
...
```

如果想要解开代码对Transport类的依赖，可以给Transport编写一个外覆类，但在这段代码中，我们并没有创建transport对象，而是从Session类那儿获取它。那么我们可以为Session类创建一个外覆类吗？不能，因为Session是一个final的类，在Java中，final类是没法被子类化（继承）的（抓狂……抓狂……）。

虽说这个邮件列表系统的代码实在是不好“剥”，因为API相对比较复杂。但如果我们手头没有任何重构工具的话，这可能已经算是最安全的做法了。 205

不过幸运的是Java有重构工具，因此我们可以采用所谓的*基于职责的提取*。这种提取手法就

是说我们先找出代码中蕴含的职责，然后针对这些职责来提取方法。

上面给出的那一小段代码有哪些职责呢？总体上来说它的职责就是发送邮件消息，它需要建立一个SMTP会话以及一个已连接的传输对象。于是，在下面的代码中我们将发送邮件消息这一职责提取到它自己的方法中，并将该方法放入一个新类MailSender当中：

```
import javax.mail.*;
import javax.mail.internet.InternetAddress;
import java.util.Properties;

public class MailSender
{
  private HostInformation host;
  private Roster roster;

  public MailSender (HostInformation host, Roster roster) {
    this.host = host;
    this.roster = roster;
  }

  public void sendMessage (Message message) throws Exception {
    Transport transport
        = getSMTPSession ().getTransport ("smtp");
    transport.connect (host.smtpHost,
                       host.smtpUser, host.smtpPassword);
    transport.sendMessage (message, roster.getAddresses ());
  }

  private Session getSMTPSession () {
    Properties props = new Properties ();
    props.put ("mail.smtp.host", host.smtpHost);
    return Session.getDefaultInstance (props, null);
  }
}
```

问题是我们如何在*剥离并外覆API*与*基于职责的提取*这两种手法之间进行抉择呢？下面就是你在选择之前需要进行的权衡：

剥离并外覆API在以下场合表现良好：

- API规模相对较小。
- 你想要完全分离出对第三方库的依赖。
- 没有现有测试，而且你也没法去编写，因为你没法通过API来进行测试。

206

使用剥离并外覆API，我们便有机会将所有的代码一举置于测试之下（除了负责把调用转发给实际的API的那层薄薄的外覆/分离层）。

基于职责的提取则在以下场合比较适合：

- API较为复杂。
- 你手头有支持安全的方法提取的重构工具，或者你觉得不用工具也能安全地完成提取。

在这些技术之间权衡优缺点是件很微妙的事情。剥离并外覆API工作量较大，但如果我们想

要把代码跟第三方库完全隔离开来的话这是个非常有用的手段，而且说实话我们常常会遇到这样的需求，详见第14章。另一方面，使用基于职责的提取时，我们可能会将自己的代码连同使用API的代码一齐提取出来以便得到一个更高层抽象的名字。之后我们的代码或许便能够依赖于高层接口而不是低层API调用了，但这同时也就意味着我们可能无法把提取的代码置于测试之下了。

事实上，许多团队两种技术都用：他们为测试编写一层薄薄的外覆，同时也编写高层的外覆（从而给他们的应用提供一个更好的接口）。 207

第16章 对代码的理解不足

一般来说踏入陌生代码的领地（尤其是遗留代码）时心里难免会有点发毛。不过一段时间以后有些人也就对此相对免疫了。他们在代码中一次次披荆斩棘，建立起了对付这类情况的信心，但要想临阵不惧还是挺难的。每个人难免都会遇到没法越过的鸿沟。如果在查看代码前就思前想后，那么情况只会变得更糟。你永远都不知道一个修改是简单的，还是会让你一个星期都焦头烂额直到咀咒系统咀咒自己的遭遇咀咒周围一切事物。如果我们具备进行修改所需的一切知识的话，事情应该会顺利许多。但怎么实现这一点呢？

以下就是一种典型的情况：你首先发现一个想要加到系统中去的特性，然后开始浏览代码。有时候的确能够在代码中得到所需的一切信息，但对于遗留代码来说，这可能会耗些时间。你在头脑中列出需要做的各件事情，在不同方案之间来回权衡。然后某个瞬间，你可能觉得有了进展，觉得有足够的信心可以开始了。又或者你也可能会被那一大堆需要实现的东西搞得头昏脑涨。阅读代码似乎并没有给你带来多大的帮助，于是你开始就你所知道的去做，并在心里祈祷一切顺利。

然而事实上情况还可以变得更好。并非只有这么一种理解代码的途径，只不过许多人就是不想去用其他办法，因为他们正忙着试图尽快把代码理解清楚。毕竟，把时间花在理解某某东西上面感觉就好像不是在用心工作一样。要是可以迅速完成理解的过程，就可以接着投入“真正的”工作了。这么想不对，但往往就是有人会这么干，这很让人感到惋惜，因为他们本可以只需花上一点点低技术含量的工夫就可以让自己后面的工作处于一个更坚固的基石上了。

208
~
209

16.1 注记/草图

如果阅读代码还是搞不清的话，你可以画草图并作一些注记。把最近看到的重要的地方写下来。如果你看到它们之间存在着某个联系，就在它们之间画一条线。这些草图不一定要像UML图或函数调用图那样复杂，以致于到处都是一些专用符号；不过要是事情本身真的变复杂了，则你可以适当考虑把图画得正式、干净一点，这样有助于你理清思路。草图往往能够帮助我们从另一个角度来看待问题。此外在我们试图理解一些特别复杂的东西的时候，草图对于记录我们的思考状态也是极有帮助的。

图16-1是我前几天跟另一个程序员在浏览代码的时候画的，我把它重新画在下面，当时我们是画在一个备忘录的反面的（出于某些考虑，图里面涉及的名字被我改掉了）。

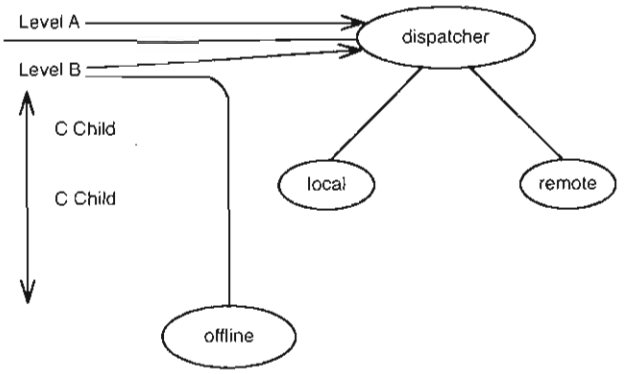


图16-1　草图

目前这幅草图还不是那么容易理解，但对于我们当时进行的讨论和交流来说已经够了。我们得到了一些认识，并最终建立了一个方案。

你可能会问，难道我们每个人不都正是这么做的吗？其实，很少有人会常常这么做。而且我怀疑原因在于没人告诉你这类事情到底该怎么做，人们很容易就会认为每次当他们把笔放到纸上的时候都应该是写代码片段或是画UML图。UML本身是没错，但圆圈和线条，还有那些不是在场人就无法理解的各种形状的图形也是有用的。图只是工具，其目的是让交流变得容易，并帮助我们记住所讨论的概念以及认识到的东西。

210

通过画草图来帮助理解一个设计的某部分的做法还有一个非常好的地方就是，它不拘一格且富有感染力。如果该技术有用，则根本无需强迫你的团队去使用它，你所需做的就是，等到你去协助某个希望理解某块代码的人的时候，在解释的时候通过画草图来帮助你解释代码的意思。如果你的队友真的在认真搞清系统的那部分的话，他/她便会看着草图并跟着你的思路去理解代码。

当开始要给一个系统做局部草图的时候，通常你可能会想先花点时间来从整体上理解系统。对此可参考第17章，里面有一组技术可以帮助你更容易地理解和照料大型代码基。

16.2　清单标注

要帮助理解代码，草图也并不是唯一的办法。比如一个我常用的方法就是“清单标注”。这对非常长的方法尤其有用。其实它的思想也很简单，而且几乎每个人或多或少都这么做过，只不过坦白地说，我认为人们没有能够把这个手法充分利用起来。

为一个代码清单作标注的方式取决于你到底想要理解什么。第一步是打印出你想要对付的代码。之后，就可以使用代码清单标注的方式来帮助你完成下面的任何步骤了。

16.2.1　职责分离

如果你想要分离职责，则可以使用一个记号来把有关的代码分组。比如你看到某几段/行代码是属于一起的，就可以在它们旁边都标上同一个特殊记号，这样你一眼就能认出来。另外，可

能的话，使用多种颜色。

16.2.2　理解方法结构

如果想要理解一个很长的方法，考虑把代码块对齐。长方法中的缩进常常会使我们无法阅读代码。这时你可以从代码块的开头到结尾画一条线，或者在每个代码块的结尾用注释标明它对应着哪个条件语句或是循环，从而把代码块首尾呼应起来。

对齐代码块的最简单办法就是从内到外。例如，如果你的代码是用C语言家族的一员写的话，你只要从代码开始一直往下读，越过每个左括号，直到遇到头一个右括号，把这个右括号标记一
211 下，然后回溯到与它匹配的那个左括号，标记它。然后接着刚才的右括号继续往下读，看到下一个右括号之后，做同样的事情；往回走直到遇到跟它匹配的那个左括号。

16.2.3　方法提取

如果想要分解一个长方法，可以考虑把你想要提取出来的代码圈起来，并注上它的耦合数（见第22章）。

16.2.4　理解你的修改产生的影响

如果想要弄清你准备进行的修改会产生哪些影响，除了使用影响结构图之外，还可以在你准备修改的那行代码旁做上记号。然后在每一个可能被此行代码的改动所影响的变量和方法调用旁做上记号。接下来再给每一个可能被你已经标记的东西的改动所影响的变量和方法做记号……尽量多做几次直到你看清影响是如何从修改点传播开来的。通过这一过程，你就会对自己所需要测试的东西有一个更好的认识。

16.3　草稿式重构

认识代码的最佳技术就是重构。你走进代码，一番捣鼓之后，代码变得更清晰了。唯一的问题就是，如果没有测试，这个过程可能会令人相当抓狂。在你为了理解代码而进行重构的整个过程中，如何知道你的修改不会破坏已有的东西？其实你可以根本无需考虑这些，这很容易做到：从代码控制系统中输出代码，别去管测试编写的事情，只管去提取方法、移动变量，以你喜欢的方式去进行重构吧，但这么做了之后你可得记住别再把这些代码输入到服务器上去了。这种做法叫做“草稿式重构”。

我第一次跟一个同事提起这事情的时候，他觉得那是浪费时间，但半个小时之后他的看法完完全全改变了，因为我们在半个小时捣鼓代码的过程中极大地深化了对代码的认识。

草稿式重构可以极大地帮助你了解代码的本质以及认识代码是如何工作的，但使用的时候也
212 并非没有一点风险。首先我们在重构的时候可能会犯一些恶劣的错误，从而使得我们误解代码的行为。这种对系统的错误认识在后面当我们开始进行真正的重构的时候会给我们带来麻烦。第二个风险也是与此相关的，那就是在第一次重构完代码之后，我们的思维可能就会固定在重构出来的系统的样子上了。这听上去似乎不应该是件多么糟糕的事情，但也未必。在我们后面进行真正

重构的时候，可能会出现各种各样的原因，导致系统最终呈现出来的结构并非我们草稿式重构出来的那样。也就是说后面我们还可能会看到更好的调整代码结构的方式。我们的代码可能时不时会被改动，自己也可能会不断产生新的想法。然而倘若我们的思维局限于草稿式重构所生成的代码结构的话，后面可能就会错过许多很好的想法了。

想要相信自己已经理解了代码中最重要的方面，草稿式重构是个很好的途径，而且前者本身也会令你的工作变得更容易。你会觉得相当自信，觉得那些边边角角的地方也并没有什么可怕的东西，或者说，即便有，你也会提前注意到。

16.4　删除不用的代码

如果你正在看的代码让你感到迷惑，而且你敢断定这段代码中有些部分并没有被使用到，那么就删掉它们吧。它们除了挡路之外没带来任何正面效应。

有些时候人们会觉得删除代码挺浪费的，毕竟编写它们的人曾在上面花了时间和精力。或许能在将来用得上呢？版本控制系统正是替你做这事情的，这段代码会被保留在先前的版本中。如果后面你发现又要用到它了，总是可以到版本控制系统那儿获取到。 213

第 17 章 应用毫无结构可言

一个开发周期很长的应用会越来越复杂臃肿。一开始的时候它或许还拥有设计良好的架构，然而几年之后，在进度的压力之下，其结构或许就复杂到没人能够真正理解的地步了。人们在一个项目上投入好几年的时间，却仍可能根本不知道新的特性应该放在哪里；他们只知道最近系统中哪儿又进行了毫无章法的修改（hack）。在添加新特性的时候，他们就会去找那些进行修改的地方（hack point），因为那些才是他们最了解的地方。

对于这类情况，并没有什么轻而易举的解决办法，况且事情也分轻重缓急；比如有时候项目组里的程序员撞了南墙，发现没法往系统里添加新特性了，然后整个项目组上上下下就立即如临大敌一般。于是有人便被临危授命，要去分析出当前到底是该对系统重新架构还是推翻重写。另一方面，对于其他一些团队来说，可能在几年的时间里系统都是磕磕碰碰地前进的。这种情况下添加一个新特性往往需要更长的时间，但人们只当这是必要的成本。没人知道项目因这种糟糕的代码结构损失了多少钱，没人知道代码的结构能够改善到什么程度。

如果一个团队不了解他们的代码的架构，后者就会变得越来越糟糕。那么，到底是什么东西阻碍了一个团队去了解他们的代码架构呢？

- 系统可能过于复杂，要想对它有一个整体认识需要花很长时间。
- 系统可能过于复杂，乃至于根本没有所谓的整体认识。
- 整个团队就像绷紧的弹簧一样，光顾着埋头解决一个又一个的紧急情况，根本无暇顾及什么整体认识了。

214 ~ 215

传统上，许多团队会使用架构师这一角色来解决这些问题。通常架构师的任务就是掌控全局，为团队作出能够维持系统整体架构不变的决策。其实这种利用架构师的做法也不是不行，只是有一个需要特别注意的地方，那就是架构师必须得跟团队的其他成员打成一片，否则代码就会逐渐偏离主航向。有两种情况可能会导致这一结果的发生：其一就是有人可能会对代码做一些不当的改动，另一种可能就是“主航向”本身需要调整。在我所遇到过的最糟糕的情形当中，就曾经出现过这样的事情：系统在架构师跟小组中的其他程序员眼里完全是不同的样子。发生这种事情通常是因为架构师有其他职责，从而无法参与编码或跟团队的其他成员交流来弄清楚系统最近的改变。结果团队的交流整个就无法进行了。

一个残酷的事实是，架构师不是少数人所专有的，而必须是大家的，因为这个角色太重要了。有架构师固然是件好事，但要想发挥架构师的最大作用，关键还是要看团队的成员是否能够清楚

架构师到底意味着什么，并能够感到架构师是跟他们休戚相关的一个角色。每一个接触代码的人都应该了解架构，而其他每一个接触代码的人都应该能够从刚才那个人所学到的东西那儿获益。如果团队里的每个人都有共同的想法，那么整体的力量就会大大增强。如果你有一个20个人的团队，其中只有3个人了解架构细节，则要么这3人需要多做许多额外的工作来让其余17人都能够跟上，要么就等着其余17人因对大局不熟而犯错误吧。

那么，我们怎样才能获得对一个大型系统的整体认识呢？方法很多。Serge Demeyer、Stephane Ducasse和Oscar M. Nierstrasz合著的*Object-Oriented Reengineering Patterns*（Morgan Kaufmann出版，2002）一书[1]中就包含了一组用来解决这些问题的技术。这里我再介绍几个相当强大的技术。如果你常在团队里实践这些技术，就会发现它们能够使你的团队始终保持对架构的关注，对于架构的维持来说，或许没有什么比这个还要重要了。因为通常我们很难对不常想到的事情保持关注。

17.1　讲述系统的故事

在跟团队共事的过程中，我常常会使用一种手法，我把它叫做“讲述系统的故事”。要成功实施这一手法至少需要两个人。像唱双簧那样，一个人开始问，“该系统的架构是怎样的？”然后另一个人就回答，他应该尽量只使用两到三个概念就把系统的架构解释清楚。如果你就是那个负责解释的人，那么就假设另一方对该系统一无所知。你用寥寥数句就解释清楚系统的设计由哪些部分构成以及它们之间是如何进行交互的。通过这寥寥数语，你就清楚地解释了这个系统最为 216
本质的东西。接下来再选第二重要的方面来讲述。就这样，一直到你们把有关这个系统的核心设计的所有重要的方面都说明白了为止。

但一开始的时候你可能会感觉有点别扭。因为要想真正简洁地将系统的架构表述出来，就得简化你的语言。你可能会这样说，“网关从活动数据库那儿获取规则集”，然而这时候你可能会听到内心的一个声音在大喊，“错！它不是还会从当前工作集那儿获取规则集嘛。”所以说当你用简化了的方式来表达的时候，你可能会感觉自己在撒谎似的，因为你觉得自己并没有把事情说全面。但是别忘了，你讲的这个简化版本能够描述一个易于理解的架构。比如说为什么网关要从不止一个地方获取规则集呢？统一对待不是更简单吗？

一些实际考虑常常会把事情弄得复杂化，但把复杂的东西看得简单起来仍然还是有价值的。至少它能够帮助每个人理解理想的系统会是什么样子的，以及哪些东西是作为权宜之计加进去的。该做法的另一个重要的作用就是，它能强迫你去思考系统当中重要的东西是哪些，要传达和交流的最重要的东西又是哪些？

如果一个团队连他们面对的系统都搞不清楚的话，可想而知是走不了多远的。从某种意义上来说，了解关于系统如何工作的简单“故事”就像是有了一个路标一样，当你想要弄清哪儿才是添加一个特性的正确地点时它能够帮你指明方向。而且，它还能够消除你对系统的畏惧感。

所以说，尽量经常讲讲你们系统的故事吧，让你的团队成员之间能够时时保持对系统的共识。

1. 此书中文版《软件再造：面向对象的软件再工程模式》由机械工业出版社2004年引进出版。——编者注

并且试着以多种不同的方式来讲述。权衡两个概念哪个更为重要。在考虑对系统的修改时，你会注意到其中有些修改与你们讲述的故事更为一致。也就是说他们令你讲述的那个简化版本的故事更接近真实。如果达到一个目标有两种方式，而你需要在这两者之间选择其一，那么系统的故事就能够帮助你判断出哪种方式能够导向一个更易理解的系统。

下面就是一个有关的例子，它描述了这种“讲述系统的故事”的做法是如何运用的。这是一个关于JUnit的会议讨论，我假定你了解一点关于JUnit架构的知识，否则建议你先花一点时间阅读JUnit的源代码（从www.junit.org下载）。

JUnit的架构

JUnit包含两个主要的类。一个叫Test，另一个叫TestResult。用户创建测试并运行它们，同时传递一个TestResult给它们。如果一个测试失败了，它就会把情况告诉TestResult。
217 于是人们就可以通过询问TestResult对象来了解发生的所有失败情况。

我们把上面这段描述所省掉的东西一一列出来：

(1) JUnit中还有许许多多其他的类。之所以说Test和TestResult是重要的只是因为我认为它们是重要的。对我来说它们之间的交互是系统当中最重要的交互。其他人可能会对架构有一个不同但同样有效的认识。

(2) 用户并不负责创建测试对象。测试对象是通过反射从测试用例类创建出来的。

(3) Test并不是一个类，而是一个接口。JUnit中运行的测试通常是在一个名叫TestCase的类的子类中写的，TestCase实现了Test接口。

(4) 人们通常并非主动向TestResult对象询问失败情况。TestResult可以接收监听对象的注册，当TestResult对象从测试那儿收到信息的时候，这些注册的监听对象都会收到通知。

(5) 测试除了报告失败之外还报告其他东西：比如运行的测试个数以及错误个数[错误(error)是指那些没有显式检查的问题，而失败（failure）则是指你检查了但失败了]。

那么，这种简化式描述是否给了我们一些关于JUnit可以怎样简化的启发呢？答案是肯定的。比如一些较简单的xUnit测试框架会让Test成为一个类，并完全扔掉TestCase。而另一些框架则将错误跟失败一起考虑，从而以统一的方式来报告它们。

回到我们的故事。

故事还有下文吗？

有。测试还可以被分组，组成一个个的所谓测试套件。我们可以像运行单个测试那样运行一个测试套件并得到测试结果。测试套件内的所有测试都会被运行起来并报告测试结果（如果失败的话）。

上面这段话我们又作了多少省略呢？

(1) 测试套件并不仅仅是持有并运行一组测试而已。它们还会通过反射来创建派生自TestCase的类的实例。

(2) 还有一个省略掉的地方，不过这个其实应该算是前面一段关于Test&TestResult的简化描述省掉的东西。即测试其实并不运行它们自身，而是把自身传递给TestResult类，后者再负责调用前者身上负责执行测试的方法。这个回调的过程发生在一个相当低的层次上。所以还不如把它看得简单一点。虽然这似乎是在隐瞒事实，但其实JUnit早年简单一点的版本里的确是这么做的。 218

还有吗？

实际上还有。Test是个接口。TestCase类则实现了这个接口。用户子类化TestCase，并将他们的测试写在返回void的公共方法里，该方法的名字要以“test”开头。TestSuite类负责使用反射来建立起一组测试，我们只需简单调用一下TestSuite的run方法，就可以运行这组测试。

其实还可以继续下去。但目前所展示的已经能够让你对JUnit有一定的认识了。我们一开始先给出一个简洁的描述。当我们通过简化以及剥掉细节的方式来描述一个系统时，其实是在进行抽象。我们往往在逼迫自己对系统作出一个非常简单的描述的时候，会发现新的抽象。

如果系统实际上并不像我们把它简化成的那样，是不是就意味着什么糟糕的事情呢？不是的。一个系统在不断成长的过程中总会变得越来越复杂。我们的简化描述只是为了给自己引路。

现在假设我们想给JUnit添加一个新特性。想要生成一个报告，汇报有哪些测试在被运行时没有调用任何断言。那么，基于我们对JUnit的描述，有哪些选择呢？

一个选择是往TestCase类中添加一个叫做buildUsageReport的方法，该方法逐一调用每个（测试）方法，把其中那些不调用断言的记录下来，建立一个报表。那么，要添加这一特性，这算是一个好方法吗？它会给我们的系统的故事带来什么样的影响呢？嗯……它会令我们原先关于系统的简洁描述里面又多出一个“隐瞒的事实”，原先的描述如下：

JUnit包含两个主要的类。一个叫Test，另一个叫TestResult。用户创建测试并运行它们，同时传递一个TestResult给它们。如果一个测试失败了，它就会把情况告诉TestResult。于是人们就可以通过询问TestResult对象来了解发生的所有失败情况。

现在，看上去Test类有了另一个完全不同的职责：生成报告。我们可从未提及。

那么倘若我们换种方式来添加该特性呢？我们可以修改TestCase与TestResult的交互方式，让TestResult在任一测试被运行时获知被调用的断言的数目。然后我们可以创建一个负责生成报告的类，并把它作为一个监听者挂到TestResult上面去。以上做法对我们的系统描述又会有什么样的影响？我们可以把它当成一个不错的动机，把前面对系统的描述表述得更一般化一点。考虑到Test对象并不仅仅向TestResult汇报失败数目，还汇报错误数目、运行的测试数目以及调用的断言数目，因此我们可以将系统的描述修改为： 219

JUnit包含两个主要的类。一个叫Test，另一个叫TestResult。用户创建测试并运行它们，同时传递一个TestResult给它们。测试运行时会把与它该次运行有关的信息传递给

TestResult对象。人们于是可以通过询问TestResult对象来获得所有有关测试运行的信息。

这样的表述是否好一些呢？坦白地说我倒还是倾向原来的，即那个用“失败”（而不是“信息”）字眼的版本。对我而言，这是JUnit的核心行为之一。如果我们修改代码让TestResult来记录运行的断言的个数，则我们的描述虽说还是不太符合事实，但我们本就已经省略了对从Test发送到TestResult的其他信息的具体描述。然而刚才提到的另一个策略，即让TestCase类去负责运行一堆用例并生成关于调用断言的报告，与事实就更为不符：我们在描述里面压根就没有提到TestCase的这个新职责。所以说，还不如就让测试在运行的时候报告它们所执行的断言数。我们的第一版描述虽说一般化得有点过，但至少是充分正确的。这意味着我们的修改跟系统的架构更一致。

17.2　Naked CRC

在面向对象理论发展的早期，许多人都曾在设计问题上有过痛苦的经历。对于大多数编程经验都是来自过程式语言的程序员来说，要想习惯面向对象不是件容易的事。简单地说就是你思考代码的方式被改变了。我还记得自己头一次“面向对象”的时候，有人把一张画了面向对象设计的纸给我看，我看着满纸的圆圈和线条，听着那人的描述，脑子里只想着问一句话“main()到哪儿去了？这些所谓的对象总得有一个入口点吧？”有那么一会儿我是一头雾水，但没多久就开始上手了。不仅仅是我有这个问题，那一段时间业界的大部分人基本都遭遇到了这个问题。而且每天都有新手加入这个行列，他们在第一次遇到面向对象代码的时候都得面对这些问题。

Ward Cunningham和Kent Beck在20世纪80年代对这个问题下了工夫。他们想帮助人们学会用对象来思考设计。那段时间Ward正使用着一个名叫Hypercard的工具，这个工具让你在计算机屏幕上创建卡片，并在卡片之间建立连接。于是，Ward脑子里灵光一闪：干嘛不干脆使用真正的索
220 引卡片来表示类呢？这将使得我们能够并容易地去谈论这些类。“我们来谈谈那个事务类吧？”“好啊，给，这是它的卡片，上面写着它的职责和与它交互的类。”

“CRC”这一缩写表示“类（Class）”、“职责（Responsibility）”和“协作（Collaboration）”。每一个卡片都被标上对应类的类名、职责以及与它交互的类（协作类）。如果你觉得某个职责并不属于某个类，可以把它从相应卡片上划掉，并写到另一个（它应属的那个）类的卡片上，或者新增一个类卡片。

CRC方案很是流行了一阵子，但最终人们还是纷纷呼唤以图方式来表达；因为几乎每一个教面向对象的人都有自己的一套表示类和关系的记号。于是最终人们不得不花好几年的时间来统一记号表达。UML就是这样诞生的，许多人都觉得UML的出现终结了人们关于如何设计系统的争论。人们开始觉得记号是手段，而UML是开发系统之道：画大量的图，然后编写代码。过了好一段时间人们才明白过来，发现尽管UML用在文档化系统方面不错，但在表达和讨论我们用于构建系统的概念方面，它并不是唯一的手段。现在我们知道，在团队中传达有关设计的概念还有一个更好的办法，我的一些做测试的朋友把它称为“Naked CRC”，因为除了不是把东西写在卡

片上之外，它几乎跟CRC一样。只不过遗憾的是，要用文字来描述它不是那么容易的。下面我尽量试图把它讲清楚。

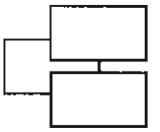
我是在几年前的一个会议上遇到Ron Jeffries的。当时他答应给我展示如何使用卡片来解释系统架构，并称他的方法可以使整个交流的过程生动且令人印象深刻。当然，后来他的确做给我看了。现在我再把这个方法传达给你，它是这样的：负责描述系统的那个人使用一组空白的索引卡片，把它们一个个摊在桌上。他可以作移动或指向卡片等动作，反正目的是通过这些卡片来传达系统中的典型对象以及它们之间是如何交互的。

下面是一个例子，是对一个在线投票系统的描述。

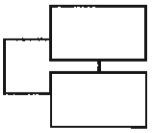
“下面我来描述一下这个实时投票系统是如何工作的。这是一个客户端会话。”（指着相应的卡片说）

221

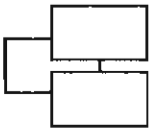
“每个会话上都有两个连接，一个接入，一个接出。”（动作：把两个卡片分别放到刚才那个卡片上，指一下它们）

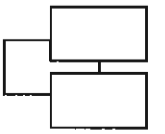


“开始的时候，服务器创建一个会话，这儿。”（动作：把一个卡片放在右边）

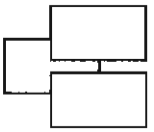


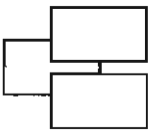
“每个服务器会话也都有两个连接。”（把表示连接的两张卡片分别放好）





“当一个服务器端会话发起时，它会向选票管理器注册。”（把代表选票管理器的那张卡片摊在服务器会话卡片的上方）





222

“服务器端可以有许多会话。”（再用三张卡片来代表一个新的服务器端会话和它的连接）

“客户端在进行投票时，选票被发送到服务器端的会话那里。”（用手指向客户端会话上的一个连接，然后在空中划一道线，到服务器端会话的一个连接那里）

“服务器端会话回复一个确认信息，然后用选票管理器记录下选票。”（用手从服务器端会话指向客户端会话，再从服务器端会话指向选票管理器）

“然后，选票管理器再叫每个服务器端会话去告知它们各自的客户端会话最新的选票数目。”（依次从选票管理器指向每一个服务器端会话）

当然，以上的假想场景肯定不全面，因为如果我们是真的一块儿坐在桌前的话，我就可以在桌上移动卡片，或者按我喜欢的方式来用手指着或操纵它们。但无论如何，该技术是相当强大的。它使得一个系统的各部分成为真实可触摸的东西。关键在于，借助卡片，你可以使用肢体动作来表达系统的各部分之间是如何交互的。这往往能令复杂的场景变得更易掌握。由于某些原因，这些借助于卡片的交流也更令人印象深刻。

“Naked CRC”在使用的时候有两条原则：

(1) 卡片代表实例，而非类；

223 (2) 用叠在一起的卡片来表示“一组实例”。

17.3　反省你们的交流或讨论

在对付遗留代码时，我们一般都不愿看到系统里面再出现新的抽象了。在面对四五个类，其中每个都具有大约一千行代码的时候，除了竭力试图搞清哪些地方得修改之外，我想谁也不会愿意再去往里面塞两个新类。

正因为我们在考察系统的时候瞻前顾后束手束脚，所以往往会错过一些能够给我们额外启发的地方。例如，我以前曾有一次跟一个团队的几个成员共事，他们那时正试图使一块规模不小的代码能够被多个线程调用执行。代码相当复杂，而且有好几处可能会导致死锁。我们意识到如果可以保证资源按照特定顺序被加锁和解锁的话，代码中的死锁就可以避免。于是便开始思考怎样修改代码才能使之成为可能。接下来的时间我们就一直在讨论这一新的资源加锁/解锁策略，并

想办法用数组来保存（锁）计数。当其他程序员中有一个人开始直接往既有代码中插入为该策略编写的代码时我说："等等，不就是锁策略吗？干嘛不创建一个LockingPolicy类，然后在它里面维护计数器呢？我们可以用恰当的方法名来描述我们想要做的事情，这可比在数组里保存计数清楚得多。"

这件事情的可怕之处在于，该团队并不是毫无经验的团队。代码基中一些其他地方的代码也有很漂亮的，但大块大块的过程式代码仿佛对人有种催眠的魔力：吸引人再往里面加代码。

听听那些关于你们的设计的讨论。你们在交流过程中使用的概念与实际出现在代码中的概念是一样的吗？我觉得肯定不尽然。实际编码可不像纸上谈兵那样只要意思到了就行，而是需要满足更强的约束，但若是在交流与实际编码之间没有一个牢固的共性，就得问问为什么了。实际上答案往往由两部分构成：代码不允许被修改成跟团队的理解一致的样子，或者团队需要换个角度去理解它。不管哪种情况，与人们自然地用来描述设计的概念保持一致是一个强有力的做法。人们在谈论设计时会努力试图让其他人理解被谈论的设计。放一些理解在代码里面吧。

本章描述了一系列的技术，这些技术可以用来揭示并传达大型系统的架构。其中许多技术也完全可以用于交流或讨论新系统的设计。记住，设计就是设计，不管它发生在开发周期中的哪一个环节。人们会犯的最糟糕的错误之一便是认为到了开发周期的某个阶段设计也就宣告死亡了。如果设计"已死"而人们却仍在修改的话，很可能就会把新的代码放在糟糕的地方，类会膨胀，因为没人会对引入新的抽象感到舒服。唉，如果你想把一个已是"遗留"的系统搞得更糟的话，以上方式真是再合适不过了。

224
~
225

第 18 章 测试代码碍手碍脚

第一次编写单元测试时你可能会觉得有点别扭。人们通常会有的一个感觉就是觉得自己编写的测试老是碍手碍脚的。他们在浏览项目的过程中有时会忘记到底是在看测试代码还是产品代码。随着他们的项目开始含有大量的测试代码，情况也未见好转。所以除非你着手建立一些约定惯例，否则可能会陷入麻烦当中。

18.1 类命名约定

第一个要建立的就是类命名约定。一般来说你至少要为某类编写一个相应的单元测试类，因此，一个有意义的做法就是从目标类的类名衍生出其单元测试类的类名。具体的办法一般有好几种，其中最常用的就是往类名前加一个“*Test*”前缀或后缀，比如我们有一个名叫DBEngine的类，就可以把它的测试类叫做TestDBEngine或DBEngineTest。至于是前缀还是后缀倒没多大关系。不过我个人倒是比较喜欢后缀的写法。因为如果你的IDE能够按字母顺序列出所有类的话，使用后缀写法就能够让每个类后面紧跟测试类，从而方便浏览。

那么，还有哪些其他类会在测试中出现呢？伪类[1]。在测试中有时需要去伪造一个类的协作类（collaborator，即与它交互的那些类），这时一个有用的做法就是把这些伪类放在一个包或目录中。对于这类情况我的惯例是使用前缀“*Fake*”。这样浏览的时候就会发现它们按照字母顺序排在了一起，同时离包中的主要类也有一定距离。这是个方便的法子，因为通常伪类是其他目录中的类的子类。

另外还有一种类也经常被用在测试中，那就是测试子类（testing subclass）。如果你想要测试某个类，然而该类上却有一些依赖是需要分离出来的，那么可能就需要为它写一个所谓的“测试子类”。这便是测试子类的由来。在使用*子类化并重写方法*（317页）技术时所编写的子类就是测试子类。我给测试子类的命名习惯是在原类名前加上“*Testing*”前缀。这样如果一个包或目录中的类按照字母表顺序列出来的话，所有的测试子类就都会挨在一起了。

226 ~ 227

比如我们有一个账目管理系统，把它里面的一个包内的类列出来就像这样：

- CheckingAccount;

1. 参见伪对象（19页）手法。——译者注

- CheckingAccountTest;
- FakeAccountOwner;
- FakeTransaction;
- SavingsAccount;
- SavingsAccountTest;
- TestingCheckingAccount;
- TestingSavingsAccount。

我们看到，在上面的列表中，每个产品类后面都紧跟着它的测试类。所有伪类和测试子类也都分别聚在一起。

我并不是说这种安排名字的方式就是唯一正确的。它在许多情况下的确有用，但同样也有许多原因和情况变更要求我们对它作些改动。总之，符合人的习惯，让人感到舒适，这一点非常重要。如何才能更容易地在测试类和产品类之间来回跳转，是一个重要的考虑。

18.2　测试代码放在哪儿

到目前为止我都是假设你会将测试代码和产品代码放在同一目录中，本章前面所讲的内容就是建立在这一假设基础上的。一般来说这种做法的确是组织一个项目的最简易办法，然而在决定是否这么干的时候，肯定还是有另外一些东西是你需要考虑的。

比如其中一个主要的考虑就是你的应用在部署的时候有没有大小限制。当然，一个运行在由你管理的服务器上的应用或许并没有多少约束。如果你可以在部署的过程中说要两倍空间就要两倍空间的话（产品代码及其测试的二进制文件），简单的做法就是把代码和测试放在同一目录下，并部署所有的二进制文件。

否则，如果你的软件是一个商业软件，并且运行在其他人的机器上，那么部署时的大小就需要加以考虑了。你可以试着把测试代码跟产品代码分开，但别忘了考虑一下这样做会给自己浏览代码带来什么影响。 228

有时候不会有什么影响，就像这个例子那样。在Java中，一个包可以横跨两个不同的目录。

```
source
     com
          orderprocessing
               dailyorders
test
     com
          orderprocessing
              dailyorders
```

即使我们把产品类放在source下的dailyorders目录下，把测试类放在test下的dailyorders目录下，它们仍可以位于同一个包中。实际上有些IDE就会把来自这两个不同目录内的类放在同一视图中，好让你无需去关心这些类的物理位置。

而在许多其他的语言和环境中，物理地点则是一个需要考虑的因素。如果你需要在目录结构

里面上下移动才能够往来于代码与测试之间的话，就好像工作的时候还得拖着个沙包。结果你会放弃编写测试，于是工作又会变得更缓慢。

一个办法是将产品代码和测试代码放在同一地点，通过构建（build）设定或脚本来阻止测试代码参加部署。如果你的类采用了良好的命名约定的话，这个法子就是可行的。

如果你选择将测试和产品代码分离开来，那么请确保你有充分的理由这么做，因为有些团队常常会为了“漂亮”而把它们分离开，他们觉得让测试代码跟产品代码待在一起是没法接受的。结果后来在浏览项目的时候就会发现麻烦了。实际上把测试代码跟产品代码放在一起也没那么
229 “丑陋”，一段时间之后你就习以为常了。

第19章 对非面向对象的项目，如何安全地对它进行修改

说实话这个标题可能存在一点争议性——当然并不是只有在面向对象语言中才能安全地修改代码，所有语言中都可以，只不过有些语言要比其他语言容易些。尽管如今面向对象思想已经在业界遍地开花，但这并不代表就不存在其他的语言，其他的编程方式了。比如我所知道的就有基于规则的编程语言、函数式编程语言、基于约束的编程语言……等等。然而，所有这些语言里面，没有哪一个能像古老的过程式语言（如C、COBOL、FORTRAN、Pascal以及BASIC）那样分布广泛。

遗留系统中的过程式代码尤其棘手。我们知道，修改代码前一个很重要的事情就是先要把代码置于测试之下，然而对于过程式代码来说，要想引入单元测试，可用的手段实在不多。通常最简单的做法就是冥思苦想，给系统打上补丁，然后祈祷你的修改是正确的。

只要是过程式的遗留代码，普遍都面临这个测试困境。过程式代码往往不具备面向对象（以及许多函数式）代码中的那些接缝。聪明的开发者可以通过小心掌控它们的依赖来解决这个问题（比如就有许多优秀的代码是用C写的），但同样只要一不小心，就会写出糟糕透顶，难以一步步验证并修改的代码来。

由于解开过程式代码中的依赖是如此之难，因此最佳策略往往是在做任何事之前先将一大块代码置于测试之下，然后在修改的过程中利用这些测试来获得一些反馈。第12章所讲的技术可以给你带来一些帮助。它们除了能用于面向对象代码之外，同样也能用于过程式代码。简而言之，找到汇点（149页）然后使用连接期接缝（32页）来适当对代码进行解依赖从而使其能被放进测试用具中，这是个有价值的尝试。如果你使用的语言支持宏预处理，则还可以使用预处理期接缝（29页）。

230 ~ 231

以上只是标准流程，并非除此之外别无他法。本章接下来将要介绍一些用于在过程式代码中进行局部解依赖的技术，以及怎样才能更轻松地进行可验证的修改，此外还有一些技术是用来对付代码可以往面向对象方式迁移时的情况。

19.1 一个简单的案例

过程式代码也并非总是那么难以入手。这里就有一个例子，`set_writetime`是Linux操作系

统中的一个C函数。如果我们得对它进行修改，给它编写测试是否会很难呢？

```
void set_writetime(struct buffer_head * buf, int flag)
{
    int newtime;

    if (buffer_dirty(buf)) {
        /* Move buffer to dirty list if jiffies is clear */
        newtime = jiffies + (flag ? bdf_prm.b_un.age_super :
            bdf_prm.b_un.age_buffer);
        if(!buf->b_flushtime || buf->b_flushtime > newtime)
            buf->b_flushtime = newtime;
    } else {
        buf->b_flushtime = 0;
    }
}
```

要测试这个函数，只需设置jiffies变量的值；创建一个buffer_head并将其传给该函数，然后在调用之后检查它的值。然而对于许多函数来说事情不是这么容易的。有时候一个函数会调用另一个函数，后者又会调用第三个函数……最后调用了一个麻烦的函数：比如一个I/O函数，或者一个来自某供应商的库函数。我们的目的是想测试一下这个代码干了什么，然而答案往往是："它的确干了些很酷的事情，但这些事只有在程序之外的某处地方才能知道，你没法知道。"

19.2　一个棘手的案例

232　假设我们想要修改下面这个C函数。首先想把它置于测试之下，再进行修改：

```
#include "ksrlib.h"

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}
```

这段代码调用了一个叫做ksr_notify的函数，后者具有一个很糟糕的副作用，它会把一个通知写到一个第三方系统里，然而我们在测试的时候宁愿它没有这个副作用。

一个办法就是使用*连接期接缝*（32页）。如果我们想要使测试期间某个库里面的所有函数的副作用都消失，可以自己做一个"伪库"来替代它，"伪库"里面的函数名字跟"真库"里的对应，不同

的是伪函数什么也不做。比如在我们的这个例子中，可以写一个赝品的ksr_notify，如下所示：

```
void ksr_notify(int scan_code, struct rnode_packet *packet)
{
}
```

我们可以把上面这个ksr_notify放在一个库中，构建该库，然后把它连接到刚才那个程序。这么一来，scan_packets这个函数的行为基本完全没有改变，只是它不会发出通知了。但后者其实并不要紧，因为我们本就想要在修改scan_packets函数之前将它的其他行为“钉牢”。

那么，该使用上面的这个策略吗？不一定，要看具体情况。如果ksr库中有许多函数，并且我们把这些函数看成系统主逻辑的“外围设施”，建立一个包含伪函数的库并连接到它是有意义的做法。而另一方面，如果我们想要通过这些函数来进行感知，或者如果我们想要改变它们返回的某些值的话，则使用连接期接缝并不是个好选择，甚至可以说是相当麻烦。由于函数的替换发生在连接期，因此对于构建的每一个可执行文件，我们只能提供唯一一份函数定义。如果想要一个伪造的ksr_notify在测试中具有一种行为而在另一个测试中具有另一种行为，就得在测试中设置一些条件，并在其函数体里面添加一些逻辑来判断该条件，从而调整函数的行为。总而言之 233
挺麻烦。然而无奈的是，在许多过程式语言面前我们别无选择。

在C里面还有一个可行的方案。C支持宏，因此我们可以利用它来简化我们给scan_packets函数编写的测试。比如在我们添加了测试代码之后，包含scan_packets的文件如下：

```
#include "ksrlib.h"

#ifdef TESTING
#define ksr_notify(code,packet)
#endif

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}

#ifdef TESTING
#include <assert.h>
int main () {
    struct rnode_packet packet;
    packet.body = ...
    ...
```

```
    int err = scan_packets(&packet, DUP_SCAN);
    assert(err & INVALID_PORT);
    ...
    return 0;
}
#endif
```

在上面的代码中，我们引入了条件编译，它的作用是当TESTING被定义的时候将ksr_notify替换为空。此外里面还提供了一小段测试代码。

将测试和源代码像这样混在同一个文件中显得不是十分清晰，往往会令代码难以浏览。一个
234 替代方案就是使用文件包含，让测试和产品代码分别位于不同的文件中：

```
#include "ksrlib.h"

#include "scannertestdefs.h"

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}

#include "testscanner.tst"
```

这样修改了之后，代码看起来就跟没有测试时的样子差不多了。唯一的区别就是文件底部多了一个#include语句。如果我们前导声明了被测试函数，就可以进一步把这个testscanner.tst文件中的所有内容都放到顶上那个#include的文件（scannertestdefs.h）中去。

要运行该测试，只需定义TESTING宏，然后单独构建这个文件。当TESTING宏被定义时，testscanner.tst中的main()函数就会被编译并连接到最终的可执行文件中，从而负责运行起测试来。testscanner.tst中的main()函数只对scan_packets的扫描例程进行测试。实际上我们还可以定义多个单独的测试函数，每个对应一个测试，然后一起调用它们，如下所示：

```
#ifdef TESTING
#include <assert.h>
void test_port_invalid() {
    struct rnode_packet packet;
    packet.body = ...
    ...
    int err = scan_packets(&packet, DUP_SCAN);
    assert(err & INVALID_PORT);
}
```

```
void test_body_not_corrupt() {
    ...
}

void test_header() {
    ...
}
#endif
```

235

在另一个文件中，我们可以在main里面把它们全部调用起来：

```
int main() {
    test_port_invalid();
    test_body_not_corrupt();
    test_header();

    return 0;
}
```

我们还可以进一步进行扩展，比如添加注册函数，使得测试的群组变得更容易。具体细节可参考www.xprogramming.com上的C单元测试框架。

尽管宏预处理很容易被误用，但在这儿的情况下它们倒是真的很有用的。文件包含和宏替换可以帮助我们战胜哪怕最棘手的代码里面存在的依赖。实际上只要把这些丑陋的宏使用局限在测试代码下，就无需过分担心我们对宏的误用会影响到产品代码。

C是主流编程语言里面为数不多的几个还支持宏预处理的。一般来说，对于其他过程式语言，要想进行解依赖，只能使用连接期接缝，并努力将更大块的代码纳入测试之下。

19.3　添加新行为

对于过程式遗留代码而言，最好遵循一条原则，即宁可引入新的函数也不要把代码直接添加到旧代码中。因为至少我们可以给我们引入的新函数编写测试。

那么，如何才能避免往过程式代码中引入依赖陷阱呢？一个办法（见第8章）就是使用测试驱动开发（TDD，74页）。TDD方法不仅适用于面向对象代码，也同样适用于过程式代码。在考虑编写一段代码前先试图写出其测试来，这样的做法往往会引发我们对代码的设计进行改良。我们只管先编写能够完成某些计算任务的函数，然后再把写好的函数集成到系统中去。

但为此我们通常得换个角度来审视将要编写的代码。比如，我们需要编写一个叫做send_command的函数。该函数会通过一个名叫mart_key_send的函数发送一个ID、一个名字、以及一个命令字符串给另一个系统。可想而知，这样一个函数的代码应该是比较简单的。我们可以想象得出，它的代码应该像这样：

236

```
void send_command(int id, char *name, char *command_string) {
    char *message, *header;
    if (id == KEY_TRUM) {
        message = rallòc(sizeof(int) + HEADER_LEN + ...
        ...
```

```
    } else {
        ...
    }
    sprintf(message, "%s%s%s", header, command_string, footer);
    mart_key_send(message);

    free(message);
}
```

然而，对这个函数，我们如何给它编写测试呢？尤其是，要想弄清这个函数做了什么，唯一的办法就是从mart_key_send入手。稍微换一换思路如何？

我们可以把这个函数里面位于mart_key_send调用之前的逻辑取出来，作为另一个单独的函数。为此我们编写测试如下：

```
char *command = form_command(1,
                             "Mike Ratledge",
                             "56:78:cusp-:78");
assert(!strcmp("<-rsp-Mike Ratledge><56:78:cusp-:78><-rspr>",
               command));
```

有了这一测试，我们便着手编写这个form_command函数，该函数返回一个命令字符串：

```
char *form_command(int id, char *name, char *command_string)
{
    char *message, *header;
    if (id == KEY_TRUM) {
        message = ralloc(sizeof(int) + HEADER_LEN + ...
        ...
    } else {
        ...
    }
    sprintf(message, "%s%s%s", header, command_string, footer);

    return message;
}
```

之后，我们再编写简单的send_command函数：

```
void send_command(int id, char *name, char *command_string) {
    char *command = form_command(id, name, command_string);
    mart_key_send(command);

    free(message);
}
```

237

很多时候这种简单的“变形”恰恰是我们所需要的。我们将所有非依赖逻辑放到一组函数中，从而使它们远离问题依赖。这番工作之后，我们往往会得到一些像send_command这样的薄薄的外覆函数，它们的作用就是将刚才取出来的逻辑与依赖混合到一起。当然，以上做法并非十全十美，但在依赖并不到处泛滥的情况下还是挺可行的。

而另一些时候，我们需要编写的函数内到处都是外部调用。这类函数基本不做什么计算工作，但它们对外部函数的调用顺序则非常关键。例如我们要编写一个计算贷款利息的函数，直观的方

法可能看上去像这样：

```
void calculate_loan_interest(struct temper_loan *loan, int calc_type)
{
    ...
    db_retrieve(loan->id);
    ...
    db_retrieve(loan->lender_id);
    ...
    db_update(loan->id, loan->record);
    ...
    loan->interest = ...
}
```

遇到这类情况该怎么办呢？对于许多过程式语言而言，最佳选择其实就是干脆别写测试，尽量直接把函数写对。也许我们可以在更高的层面测试该函数是否正确。但在C里面我们还有一个选择。C支持函数指针，我们可以利用该语言特性来设立另一个接缝，如下所示：

先创建一个包含一系列函数指针的结构体：

```
struct database
{
    void (*retrieve)(struct record_id id);
    void (*update)(struct record_id id, struct record_set *record);
    ...
};
```

然后我们将它里面的函数指针初始化为指向相应的数据库访问函数。于是，以后如果要编写需要访问数据库的函数，只需把这个结构体传给它即可。在产品代码中，我们的结构体里面的函数指针可以指向真正的数据库访问函数，而在测试的时候则可以将它们指向伪函数。

注意，对于比较老的编译器，我们可能需要采用旧的函数指针语法：

```
extern struct database db;
(*db.update)(load->id, loan->record);
```

不过，如果你的编译器比较新的话，你就可以用一种非常自然的“面向对象”的风格来调用这些函数了： 238

```
extern struct database db;
db.update(load->id, loan->record);
```

该技术并非仅适用于C。任何语言，只要支持函数指针，都支持该技术。

19.4　利用面向对象的优势

对于面向对象语言，我们可以利用对象接缝（object seam，35页）。对象接缝有一些很好的特质：

- 容易从代码中把它们认出来。
- 可以用于将代码分解为更小、更易理解的块。
- 提供了更好的灵活性。为了测试而引入的接缝在你需要扩展系统时或许还能起到帮助。

然而遗憾的是，并非所有的系统都可以容易地迁移到面向对象，但有些系统迁移起来的确要比其他系统容易得多。许多过程式语言已经进化成了面向对象语言。如微软的VB变成完全面向对象语言只是最近的事，COBOL和Fortran也有面向对象扩展，此外大多数C编译器也能够编译C++代码。

如果你的语言允许你把代码向面向对象迁移的话，你手头就有了更多的选择。第一步通常是使用封装全局引用（Encapsulate Global Reference，268页）来将你打算修改的代码块置于测试之下。还记得本章一开始的scan_packets所处的糟糕的依赖情况吗？利用该技术，我们便可以从中解脱出来。回想一下，当时问题的源头是ksr_notify函数：我们希望在测试期间，该函数不会发送任何的通知消息。

```
int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
239     current = current->next;
    }
    return err;
}
```

首先，我们要将代码在C++模式下编译。这个改动可大可小，具体取决于我们怎么做。我们可以主动出击，试着将整个项目在C++下重新编译；或者也可以一块一块的来，只是这样做要花点时间。

一旦代码在C++下编译成功，我们便可以搜索ksr_notify函数的声明并将它包装到一个类当中：

```
class ResultNotifier
{
public:
    virtual void ksr_notify(int scan_result,
                            struct rnode_packet *packet);
};
```

我们还可以为该类引入一个新的源文件，并将其默认实现放在该文件里：

```
extern "C" void ksr_notify(int scan_result,
                           struct rnode_packet *packet);

void ResultNotifier::ksr_notify(int scan_result,
                                struct rnode_packet *packet)
{
    ::ksr_notify(scan_result, packet);
}
```

注意，我们并没有修改函数的名字或签名。我们使用了*签名保持*（249页）技术来将出错的几率降到最低。

接下来声明一个全局的ResultNotifier对象，并将其放入一个源文件中：

```
ResultNotifier globalResultNotifier;
```

现在我们便可以重新编译代码，让编译错误告诉我们哪儿需要改动。由于已经把ksr_nofity的声明放入了一个类当中，所以编译器在全局作用域内就找不到它的声明了。

以下是原来的函数：

```
#include "ksrlib.h"

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}
```

240

要想让上面的代码再次通过编译，可以使用一个外部声明来让globalResultNotifier对象对编译器可见，并给ksr_notify调用加上globalResultNotifier前缀：

```
#include "ksrlib.h"

extern ResultNotifier globalResultNotifier;

int scan_packets(struct rnode_packet *packet, int flag)
{
    struct rnode_packet *current = packet;
    int scan_result, err = 0;

    while(current) {
        scan_result = loc_scan(current->body, flag);
        if(scan_result & INVALID_PORT) {
            globalResultNotifier.ksr_notify(scan_result, current);
        }
        ...
        current = current->next;
    }
    return err;
}
```

这样修改了之后，代码的行为还像原来那样。ResultNotifier上的ksr_notify方法会将任务转发给（全局的）::ksr_notify来完成。那么，以上这番工作究竟能给我们带来什么好处呢？嗯……目前还没有。下一步工作就是试图让我们能够在产品代码中使用这个Result-Notifier对象，而同时能在测试代码中使用另一个对象。实现这个目的有多种方法，但从长远考虑的话，最好还是再一次运用*封装全局引用*（268页）来把scan_packets也包进一个类之中，
241 我们姑且把这个类叫做Scanner吧，如下所示：

```
class Scanner
{
public:
    int scan_packets(struct rnode_packet *packet, int flag);
};
```

这么一来我们便可以运用*参数化构造函数*（297页）手法来让Scanner类使用我们提供的Result- Notifier对象了：

```
class Scanner
{
private:
    ResultNotifier& notifier;
public:
        Scanner();
        Scanner(ResultNotifier&  notifier);

    int scan_packets(struct rnode_packet *packet, int flag);
};


// in the source file

Scanner::Scanner()
: notifier(globalResultNotifier)
{}

Scanner::Scanner(ResultNotifier&  notifier)
: notifier(notifier)
{}
```

这番修改之后，我们便可以找到那些调用scan_packets的地方，创建一个Scanner实例，然后调用它上面的scan_packets方法。

以上这些修改都相当安全，也相当机械化。虽然它们并不是面向对象设计的良好案例，但作为解依赖技术它们已经表现得足够好了，使我们在后续的工作中得以进行测试。

19.5　一切都是面向对象

有些过程式程序员喜欢抨击面向对象，他们认为面向对象是没必要的，或者认为它所带来的复杂性并没换来什么实质性的好处。但如果你认真想一想的话，就会意识到，所有的过程式程序

其实都是面向对象的，只不过可惜的是它们中很多都只包含一个对象。为什么这么说呢？我们考虑一个包含了大约100个函数的程序，以下是这些函数的声明：

242

```
...

int db_find(char *id, unsigned int mnemonic_id,
            struct db_rec **rec);
...
...
void process_run(struct gfh_task **tasks, int task_count);
...
```

假设我们可以将以上这些声明全都放进同一个文件当中，并用一个类来将它们包起来：

```
class program
{
public:
    ...
    int db_find(char *id, unsigned int mnemonic_id,
                struct db_rec **rec);
    ...
    ...
    void process_run(struct gfh_task **tasks, int task_count);
    ...
};
```

然后，我们找到每个函数的定义，比如：

```
int db_find(char                *id,
            unsigned int        mnemonic_id,
            struct db_rec       **rec);
{
    ...
}
```

给它们的函数名加上刚才那个类名作为前缀：

```
int program::db_find(char               *id,
                     unsigned int       mnemonic_id,
                     struct db_rec      **rec)
{
    ...
}
```

经过以上这番改动之后，我们就得为程序编写一个新的main()函数了：

```
int main(int ac, char **av)
{
    program the_program;

    return the_program.main(ac, av);
}
```

243

那么，这些改变系统的行为了吗？没有。这些只不过是一些机械的改动，程序的意义和行为根本没有受到任何影响。所以说，“老式”的C程序其实就是一个大对象。而我们在使用封装全局

引用手法时，则是在创建新对象，将系统分成一个个的子部件，从而使其更容易对付。

如果一门过程式语言支持面向对象扩展，那么我们就可以像上面这样对系统进行迁移。这项技术并不是什么高深的面向对象技术，我们只不过是利用一点点面向对象的知识来将程序分解从而使其易于测试罢了。

如果我们的语言支持面向对象，那么除了提取依赖性之外，还能用它来做些什么呢？其一就是我们可以逐步将系统朝向更好的面向对象设计改进。一般而言这就意味着你得将相关的函数群组到一个个的类当中，提取出许许多多的函数以便将复杂的职责分解开来。更多这方面的建议可以参考第20章。

对于过程式代码，我们手头的选择要比面向对象代码少，但还是可以在过程式遗留代码中取得进展的。不可否认的是过程式代码所特有的接缝形式严重地加大了我们工作的难度。因此如果你的过程式语言支持面向对象扩展的话，我建议你对系统进行迁移。对象接缝（35页）除了将测试安置到位之外还有其他很多好处。连接期接缝和预处理期接缝虽说也是很好的选择，但除了能
244 将代码置于测试之下之外，它们在改善设计方面并无多大贡献。

第20章 处理大类

人们往系统中加入的特性其实有许多都是一些小调整。在添加它们的时候需要添加一点儿代码，或许再加上几个方法。这时候你就会发现，把这些东西添加到一个既有的类身上是一个较具诱惑力的选择。很可能你需要添加的代码必须用到某个既有类中的数据，因此最简单的做法便是直接把代码塞到那个类当中。遗憾的是，这种省力的修改代码的方式可能会带来严重的麻烦。随着我们一再地往既有类中添加代码，既有类的方法和类本身就会变得越来越庞大，我们的软件会变成一个沼泽，然后你就得花上更长的时间才能搞清如何往里添加新特性，甚至连理解旧特性都会花更多时间。

有一次我去协助一个团队，该团队做了一个从纸上看来相当不错的架构设计。他们告诉我系统中哪些类是主要类，以及通常情况下它们之间如何交互。接着他们给我看了一些展示架构的UML图。然而，当我看到代码时却愣住了。系统中的每个类几乎都可以被分解为大约10个类，而这么做正是帮助他们摆脱目前面临的最紧迫问题的钥匙。

那么，庞大的类有哪些问题呢？首先就是容易混淆。如果一个类有五六十个方法，那么要想搞清哪些东西需要修改以及你的修改是否会影响到其他什么东西往往就困难了。最糟糕的情况就是，一个庞大的类具有多得不可思议的成员变量，对于这样一个类，我们很难知道修改一个变量会带来什么样的影响。另一个问题就是任务调度。如果一个类具有大约20个职责，那么很可能有许多原因要修改它，多得你忙不过来。于是你可能会遇到在同一个迭代期中好几个程序员要对该类进行不同的修改的情况。如果他们同时进行修改，就可能会导致一些严重的冲突，尤其是当考虑到第三个问题的存在的时候，即庞大的类测试起来非常痛苦。封装是好事情，但可别对测试人员这么说，他们可不这样想。过分庞大的类往往隐藏了过多的东西。当封装能够帮助我们推断代码的行为，当我们知道某些特定的东西只有在特定的情况下才能被修改时，封装是好事情。然而，封装一旦过了头，被封在里头的东西便会腐烂发臭了。比如你没法容易地感知到修改带来的影响，于是只能退而采用*编辑并祈祷*的法子。而一旦事情到了这个地步，要么你的修改耗时漫长，要么bug数目增长。反正你得为代码缺乏清晰性付出代价。

如果有一个庞大的类，那么第一个需要面对的问题就是：怎样修改才能不至于令已经糟糕的状况雪上加霜？可以使用的两个关键技术是*新生类*（54页）和*新生方法*（52页）。当需要修改代码时，我们应该考虑将新添的代码放进一个新类或新方法中。新生类方法能够防止系统的状况变得更糟。没错，当你将新加的代码放进新类中时，可能会需要从原类中调用这个新类，但至少没

有令原类变得更大。新生方法也有帮助，不过比起新生类来，它起到的帮助就比较微妙了。把添加的代码放到新方法中的确会导致系统中多出一个方法来，但至少你给它所属类所做的另一件事情起了一个名字，并在系统中把它标识出来；而且往往这些新生方法的名字能够暗示你如何将一个类分解成更小的类。

对于庞大的类，一个关键的补救手段就是重构。将大类分解为一组小类是有帮助的。然而最大的问题在于要搞清楚这些小类应该是什么样子的。幸运的是，对此我们有一些指导原则。

单一职责原则（SRP）

每个类应该仅承担一个职责：它在系统中的意图应当是单一的，且修改它的原因应该只有一个。

单一职责原则描述起来有点困难，因为“职责”这个概念有点含糊。如果我们以一种非常单一的视角来看待这个概念的话，我们也许会说：“哦，那是不是意味着每个类应该只有一个方法？”呃……方法的确可以被看成职责。比如Task类，它的run方法的职责是运行该Task，它的taskCount方法的职责是告诉我们它有多少个子任务，等等。但当我们关心的是主要意图时，“职责”这个词到底意味着什么就不是无关紧要的了。图20-1给出了一个例子。

246

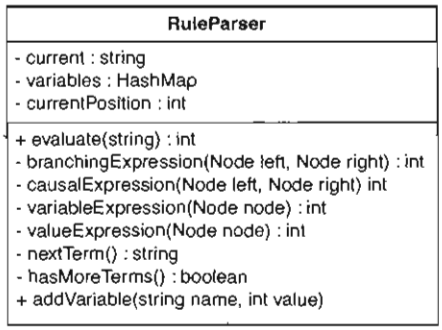


图20-1　规则（Rule）解析器

RuleParser是个不大的类，它会对一个包含了一组规则表达式的字符串进行求值（该字符串内的规则表达式是用什么语言写的我们不用关心）。顾名思义，该类的一个职责是解析。但这还不是它的主要意图，除了解析规则之外它还进行求值。

那么，这个类还做哪些事情呢？它持有一个当前字符串，即它正在解析的串。它还有一个成员变量，该变量保存当前解析到的位置。这两个小小的职责看起来还是属于“解析”这个职责范畴的。

RuleParser类还有另一个成员变量variables。它持有一组变量，这组变量被该类用来对规则内的如“a+3”这样的算数表达式进行求值。比如，你用a和1作为参数来调用addVariable方法，则“a+3”就会被求值为4。因此，这似乎意味着该类还有另一个职责——变量管理。

还有其他职责吗？另一个寻找职责的方法就是观察方法名。这些方法可以这样自然分组：

evaluate	branchingExpression causalExpression variableExpression valueExpression	nextTerm hasMoreTerms	addVariable

evaluate方法是这个类的进入点。一共只有两个公有方法，它就是其一，它标明了该类的一个关键职责：求值。然后，所有以Expression为后缀的方法基本都一样。它们不仅名字相似，并且 247
全都接受Nodes为参数，并返回一个int来表明一个子表达式的值。再然后就是nextTerm和hasMoreTerms这两个方法，它们是用来辅助对串里面的项进行某种特殊的字元化（tokenization）操作的。最后，前面已经说过，addVariable方法的作用是变量管理。

所以，总的来说，Parser类具有如下的职责：

- 解析；
- 表达式求值；
- 项的字元化；
- 变量管理。

假设我们现在必须得从头对这个类体系进行设计，并且在我们的设计中，上面列出的职责必须是分离开的。那么我们可以做出什么样的设计呢？如图20-2所示：

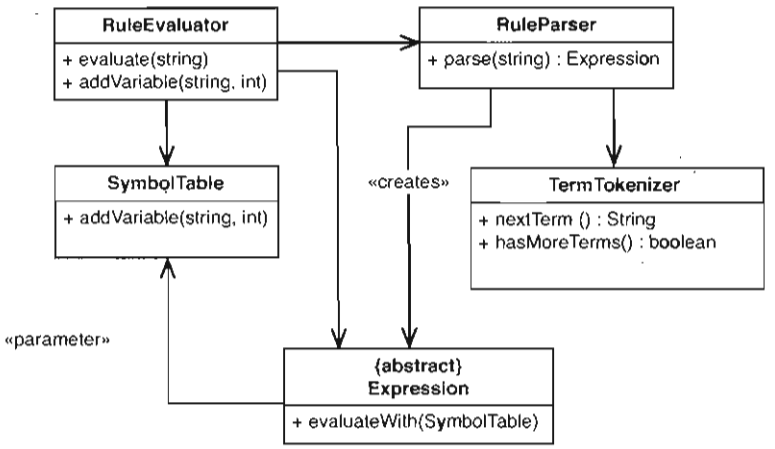


图20-2 Rule类体系，职责被分离开来

这是不是有点设计过度了？可以这么说。编写小型语言解释器的程序员们通常会把解析模块和表达式求值模块并在一起，一边解析一边求值。虽然这种做法比较方便，但随着语言的演化，其在可伸缩性上面的不足就会暴露出来了。另外一个有点微不足道的职责就是SymbolTable的职责。如果SymbolTable的唯一职责就是将变量映射到值的话，用一个类来实现并不比用一个散列表或链表来实现具有更多的优势。所以，这虽说是个不错的设计，但相当不现实。除非我们选择重写系统的这一部分，否则这堆漂亮的多类设计只是空中楼阁。 248

在现实世界中，对于大型类，关键在于找出其中不同的职责，然后想出一个方案能够逐步向更集中的职责发展。

20.1　职责识别

在上一节的RuleParser例子里，我展示了一种将这个类分解为一组小类的做法。实际上在做那个分解的时候基本是依葫芦画瓢的。把所有方法都列出来，然后就思考它们的意图是什么。我问自己的两个关键的问题就是，“这个方法为什么在这儿？”“它为这个类做了什么？”接着把这些方法分组，将具有相近目的的方法放在一起。

我把这种职责识别的方式叫做方法分组。当然，方法远不止这一种。

识别职责是一个关键的设计技能，而且需要锻炼才能掌握。你可能会觉得在一本关于如何对付遗留代码的书中谈论设计技能有点不伦不类，但其实从现有代码中发现职责跟描述出尚未编写的代码的职责并没有太大的区别。关键是要能够看清职责并学习如何将它们很好地分离开来。如果硬要说它们之间有什么区别的话，遗留代码比起新加特性来说，它提供了多得多的运用设计技能的机会。当会被你的设计所影响的代码真实地存在眼皮底下时，谈论起设计权衡来会更容易一些，同样，也更容易知道某个结构在特定上下文中是否合适，因为上下文就真实地存在于我们面前。

本节描述了一组可以用于在既有代码中识别出职责的启发式方法。注意，我们并没有创造出新的职责，而仅仅是识别出本就存在的职责。不管你的遗留代码具有什么样的结构，组成它的部件所干的事情肯定都是可识别的。但有时候比较难以识别，这时便可以借助于这里介绍的技术了。你甚至可以试着把它用到并不需要立即修改的代码上去。对代码内在的职责越是关注，你就越了解代码。

探索式方法#1：方法分组

寻找相似的方法名。将一个类上的所有方法列出来（别忘了它们的访问权限），找出那些看起来是一伙的。

249

这种方法分组的技术是个相当好的开始，尤其对于大型类而言。关键是要认识到，并不需要将所有方法都分组放到新类中去。你只需看看能否找到一些看上去属于同一职责的方法就行了。然后，如果能够找到一些有点偏离类的主要职责的职责，你心里对代码如何改进也就有了一个大概的方向了。然后你就等待，直到必须得去修改那些你已经分好类的方法时再决定是否要提取出一个类来。

方法分组还是一个极佳的团队练习。放上一块黑板，上面列出你们的每个主要类里面的所有方法名。团队成员可以在上面做标记，指出不同的方法分组方式。而整个团队则可以共同筛选出比较好的那些分组方式，并决定代码今后的走向。

探索式方法#2：观察隐藏方法

注意那些私有或受保护的方法。大量私有或受保护的方法往往意味着一个类内部有另一个

类急迫地想要独立出来。

庞大的类可能会隐藏很多东西。对于刚刚接触单元测试的人来说，这个问题总是会困扰他们。比如“我怎么才能测试私有方法呢？”许多人花了大量时间试图避开这个问题，然而正如我在前面的章节中所说，真正的答案是，如果你迫切需要测试一个私有方法，那么该方法就不应该是私有的；如果将它改为公有会带来麻烦，那么可能是因为它本就应属于另一个独立的职责。它应该在另一个类上。

前面提到的RuleParser类就是个很好的例子。它有两个公有方法：evaluate和addVariable。除此以外其他所有东西都是私有的。现在，假设我们将nextTerm和hasMoreTerm也设为公有，会发生什么事情呢？呃……会令RuleParser类看起来相当的怪异。使用RuleParser的人可能会认为他们需要使用这两个方法结合evaluate方法来解析并求值表达式。的确，让nextTerm和hasMoreTerm成为RuleParser类的公有方法是比较怪异，但若是把它们放到一个新类TermTokenizer上面作为公有方法的话就好得多了，甚至可以说是完全合适。这么做并不会损害RuleParser的封装性。因为虽说nextTerm和hasMoreTerm在TermTo- kenizer上面是公有的，但在RuleParser里面却是被“私下地”访问的。如图20-3所示：

250

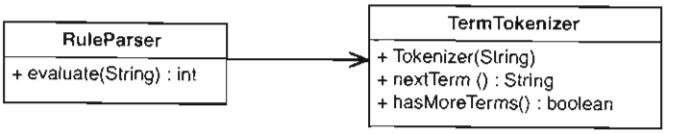


图20-3 RuleParser和TermTokenizer

探索式方法#3：寻找可以更改的决定

寻找代码中的决定——这里所说的“决定”并非指你正在做的决定，而是指已经作出的决定。比如代码中有什么地方（与数据库交互、与另一组对象交互，等等）采用了硬编码吗？你可以设想它们发生变化后的情况吗？

人们在试图去分解一个庞大的类时往往容易把大量精力花在关心那些方法名上。因为毕竟方法名是一个类里面最引人注意的东西之一。然而，方法名并不代表一切。通常，对于一个大型类而言，它里面的方法会在许多不同的抽象层面做许多事情。例如，一个叫做updateScreen的方法可能会给显示屏生成文本，格式化然后发送到好几个不同的GUI对象那里。这时如果光看方法名，你是没法知道背后真正发生了多少事情以及它里面到底隐藏了多少职责的。

因此，在真正开始拿类开刀之前，最好先对它的方法做一点方法提取。应该提取什么样的方法呢？我的做法是寻找代码中的决定。代码中对多少东西作了假设？代码调用了某个特定API中的方法吗？代码是否假定它将总是访问同一个数据库？如果你的代码作了以上这些事情，那么我建议你最好提取出一些能在较高层面反映你的意图的方法。如果你的代码从某个数据库中获取特定的信息，那么你可以提取出一个方法，给它起一个能够反映你所获取的信息的名字。做完这些方法提取之后会多出许多新的方法，但你同样可能会发现方法分组变得更容易了。更好的是，你

可能会发现你已经把某些资源完全封装在一组方法之后了。从而当你为这些方法提取出一个类时，一些底层细节上的依赖也就被你解开了。

探索式方法#4：寻找内部关系

251 寻找成员变量和方法之间的关系。“这个变量只被这些方法使用吗？”

随便找一个类来几乎总能发现其中并非所有的方法都使用了所有的成员变量。形象地说，一个类里面的方法总是堆成一团一团的。比如一组共三个成员变量，也许用到了它们的方法总共也就两三个。这种“抱团”的现象通常可以借助于方法名看出来。例如在RuleParser类中就有一个叫做variable的Collection，以及一个叫addVariable的方法。这就显示出该方法和该变量之间有明显联系。虽然它并没有告诉我们其他还有哪些方法使用了这个变量，但至少我们有了一个寻找的起点。

还有另一项技术也可以用于寻找这种方法抱团的现象，那就是给一个类内部的关系画一张草图。这种草图叫做特征草图（feature sketch[1]）。特征草图展示了一个类里面的每个方法使用了哪些成员变量以及其他方法。下面就是一个例子：

```
class Reservation
{
    private int duration;
    private int dailyRate;
    private Date date;
    private Customer customer;
    private List fees = new ArrayList();

    public Reservation(Customer customer, int duration,
            int dailyRate, Date date) {
        this.customer = customer;
        this.duration = duration;
        this.dailyRate = dailyRate;
        this.date = date;
    }

    public void extend(int additionalDays) {
        duration += additionalDays;
    }

    public void extendForWeek() {
        int weekRemainder = RentalCalendar.weekRemainderFor(date);
        final int DAYS_PER_WEEK = 7;
        extend(weekRemainder);
        dailyRate = RateCalculator.computeWeekly(
                                customer.getRateCode())
                        / DAYS_PER_WEEK;
    }

    public void addFee(FeeRider rider) {
```

1. 这里特征是指类的方法或成员变量。——译者注

```
        fees.add(rider);
    }

    int getAdditionalFees() {
        int total = 0;
        for(Iterator it = fees.iterator(); it.hasNext(); ) {
            total += ((FeeRider)(it.next())).getAmount();
        }
        return total;
    }

    int getPrincipalFee() {
        return dailyRate
               * RateCalculator.rateBase(customer)
               * duration;
    }

    public int getTotalFee() {
        return getPrincipalFee() + getAdditionalFees();
    }
}
```

252

第一步是为每一个成员变量画一个圈，如图20-4所示：

接着我们观察每个方法，也给它们各自都画一个圈。然后在任一方法与该方法用到的任何成员变量或方法之间画一个带箭头的线。通常我们不用考虑构造函数，因为一般来说构造函数会修改每个成员变量。

253

图 20-5 展示了初步的结果，其中，我们从extend()出发画了一个箭头到它所修改的变量duration上。

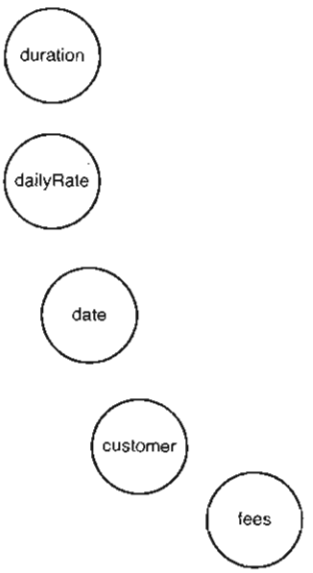


图20-4 Reservation类里面的成员变量

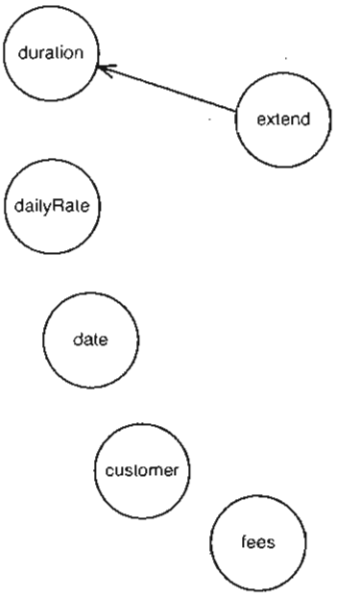


图20-5 extend使用了duration

如果你已经阅读过了介绍影响结构图的章节，你可能会注意到，这些特征草图跟影响结构图十分相像。实际上的确是这样，它们本来就相当接近。主要区别在于图中箭头的方向恰恰相反。在特征草图中箭头是指向被使用的变量或方法，而在影响结构图中箭头则是指向被影响的变量或方法。

特征草图和影响结构图是两种不同但同样完全合法的、描绘系统内交互的方式。要想反映类的内部结构，特征草图再合适不过了。而若是想从一个修改点开始推测影响传播的线路，则影响结构图是得力工具。

那么，它们长得这么相似会不会混在一块搞不清谁是谁呢？其实不会。这些草图是属于那种用完就丢的工具。也就是说在你进行修改之前，和你的同伴坐下来，花上十分钟描描画画，之后就可以把它们扔了。留着也没什么意思，所以基本不可能会混在一起分不清。

254

图20-6展示了完整的特征草图，每个相应的使用都对应着一条线。

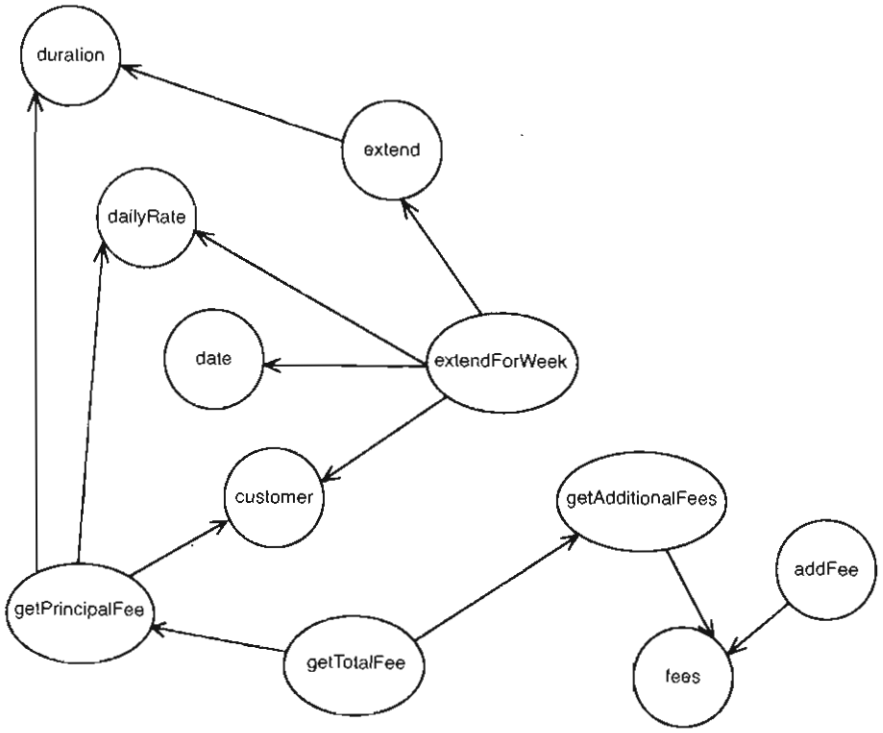


图20-6　Reservation的特征草图

那么，从这张草图中我们可以看出什么来呢？首先很明显的一点就是其中存在聚集的现象。duration、dailyRate、data以及customer变量主要是被getPrincipalFee、extend以及extendForWeek使用到的。这三个方法里面有公有的吗？有，extend和extendForWeek就是，但getPrincipalFee不是。如果我们将这一块切出来，做成一个它们自己的类，系统会变成什

么样子呢（见图20-7）？ 255

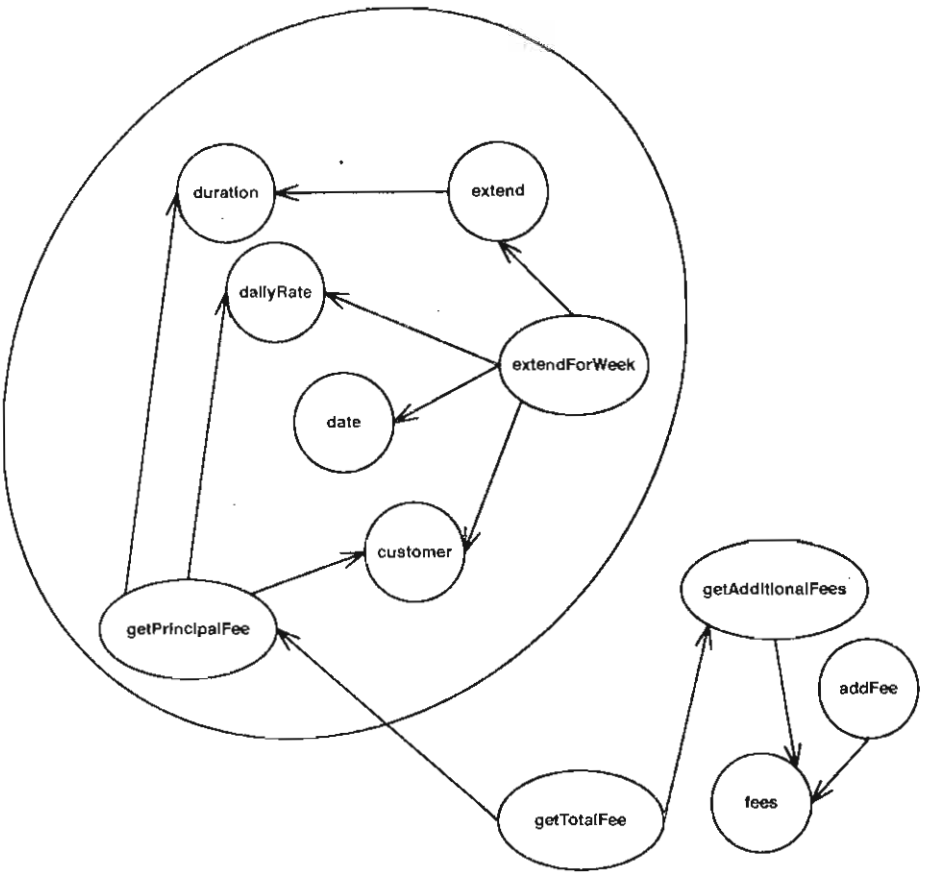


图20-7 Reservation中的一个聚集

上图中的那个大圆圈可以做成一个新类。extend、extendForWeek以及getPrincipalFee需要被设为公有方法，但其他所有方法都可以是私有的。我们可以将fees、addFee、getAdditionalFees以及getTotalFee保留在原类Reservation中，并将有关任务委托刚才那个新类来完成，如图20-8所示： 256

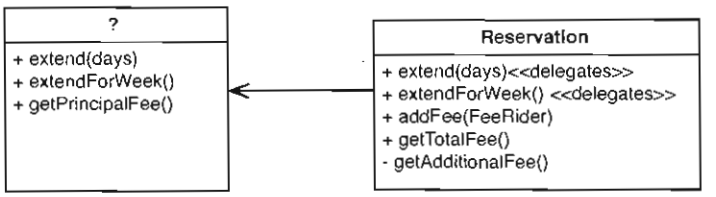


图20-8 Reservation使用一个新类

在试图这么做之前，有一个关键的问题要先弄清楚，那就是这个新类是否具有一个良好的、清晰的职责。我们能否替它想出一个名字来？答案是这个新类似乎会做两件事情：一是将一个预定（reservation）延期（extend、extendForWeek），另一是计算所谓的主要费用（getPrincipalFee）。所以似乎Reservation是个不错的名字，但可惜这名字已经被原来那个类用掉了。

其实我们还有另一种选择。刚才我们是把大圈圈里面的所有元素提取出来，现在我们反过来，把小圈里面的提取出来，如图20-9所示：

257

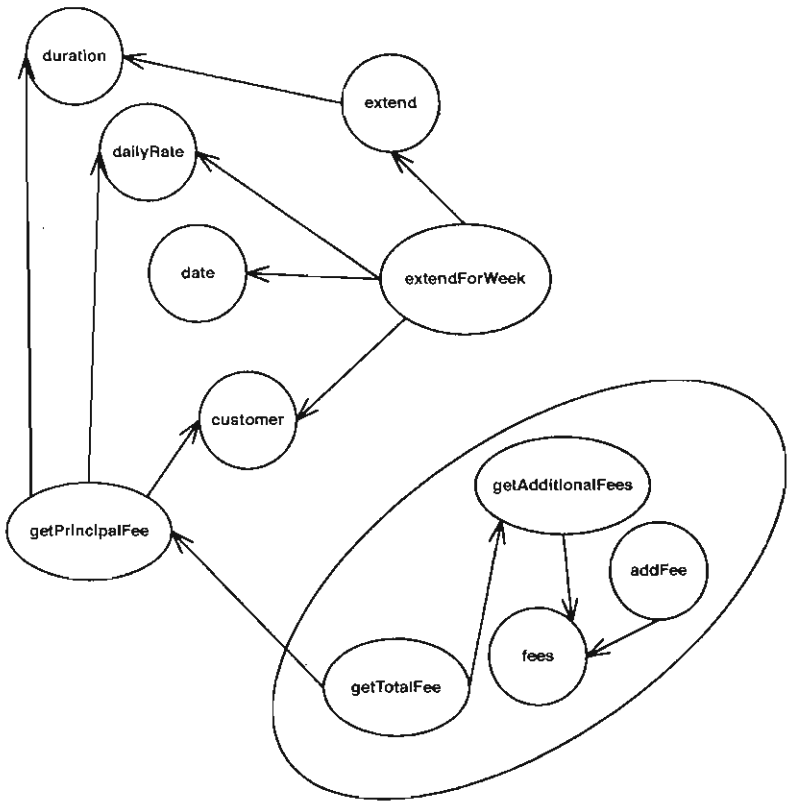


图20-9　换个角度看待Reservation

我们可以把提取出来的类叫做FeeCalculator。这种做法是可行的，但getTotalFee方法就需要调用Reservation上的getPrincipalFee了——对吧？

如果我们先调用getPrincipalFee，然后将得到的值传递给FeeCalculator，可不可以呢？代码大致如下：

258

```
public class Reservation
{
    ...
```

```
    private FeeCalculator calculator = new FeeCalculator();

    private int getPrincipalFee() {
        ...
    }

    public Reservation(Customer customer, int duration,
            int dailyRate, Date date) {
        this.customer = customer;
        this.duration = duration;
        this.dailyRate = dailyRate;
        this.date = date;
    }

    ...

    public void addFee(FeeRider fee) {
        calculator.addFee(fee);
    }

    public getTotalFee() {
        int baseFee = getPrincipalFee();
        return calculator.getTotalFee(baseFee);
    }

}
```

代码的最终结构如下（图20-10）：

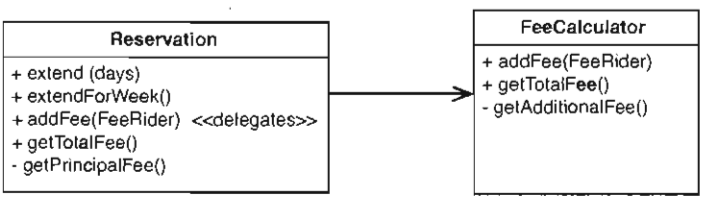


图20-10 Reservation使用FeeCalculator

我们甚至可以考虑将getPrincipalFee移至FeeCalculator类中，从而令该类的职责跟它的类名更一致，然而考虑到getPrincipalFee依赖于Reservation类中的好几个成员变量，所以最好还是把它留在Reservation里。

259

特征草图在寻找类中的独立职责方面是个极好的工具。我们可以试着将一个类的特征（方法跟成员变量）进行分组，然后看看能基于这些名字提取出什么类来。但除了能够帮助我们找出职责之外，特征草图还能帮助我们看清类中的依赖结构，在我们决定应该把哪些东西提取出来的时候，它往往显得和职责一样重要。比如，本例中有两块比较“密集”的特征聚集。它们之间的唯一联系就是getTotalFee在内部对getPrincipalFee的调用。在特征草图中，这类联系往往会以大块特征聚集团之间的寥寥几根连线表现出来。我把这些点叫做汇点，第12章对此有详细讨论。

有时候你会发现一张特征草图上找不到任何汇点。是的，并非每张草图都有汇点。但至少草

图仍然能够帮助你看清方法名以及类中各元素之间的依赖关系。

有了特征草图之后，就可以试用各种不同的方式来分解类了。首先可以将一组组的特征用圆圈圈起来。在画圈的时候，被圈所“切断”的线即意味着新类的接口。你可以一边画圈一边为每组特征想出一个类名来。坦白的说，撇开你在提取类时进行的那些抉择，这是个很好的提高命名技术的方法，此外还是个不错的探索设计方案的途径。

探索式方法#5：寻找主要职责

尝试仅用一句话来描述该类的职责。

单一职责原则告诉我们，一个类应该具有单一的职责。如果情况的确如此，那么应该很容易就用一句话把它概括出来。在你的系统内找出一个大类，试试看能否仅用一句话来描述其职责。随着你不断发现该类的客户期望以及需要的东西，句子会变得越来越臃肿，“这个类会做‘这个’、‘这个’，还有‘这个’，‘那个’……。”你得问自己：所有这些事情里面，存在一个比其余事情都要重要的吗？如果有，那么恭喜你，你或许已经发现了该类的关键职责。其余职责或许应该被分解到其他类当中去。

单一职责原则的违反有两种形式。一是在接口层面违反，二是在实现层面违反。当一个类的接口呈现出负责多样事务的形态时，它就在接口层面违反了单一职责原则。例如，该类的接口（图
260 20-11）看上去就似乎能够被分解成三到四个类。

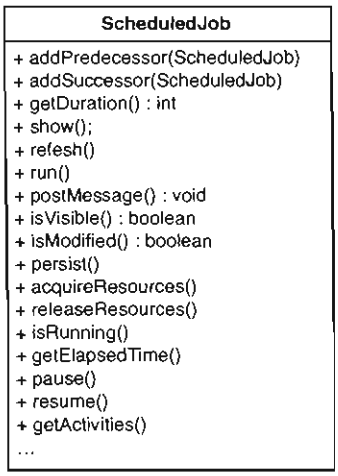


图20-11　ScheduledJob类的接口

然而我们最为关心的SRP违反还是实现层面的。简单地说就是，我们关心的是该类是否真的做了这些事情，还是仅仅将其委托给其他的类来完成。如果属于后者，那么该类并不能算是一个巨大的单片类；而只不过是一大帮小类的“前端”，一个更容易掌控的facade（门面类）。

261 图20-12就展示了这样一个类，ScheduledJob类将一系列的职责委托给另外几个类来完成：

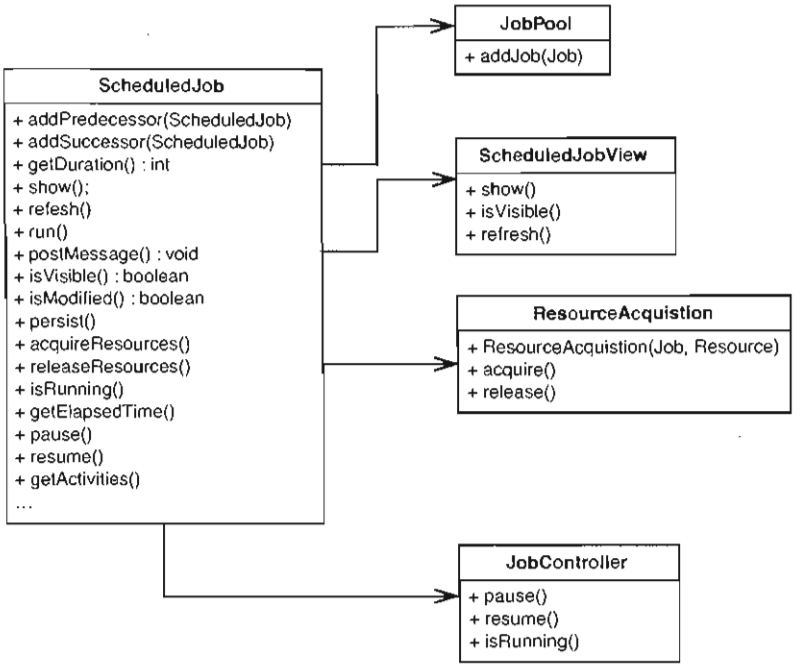


图20-12 ScheduledJob和提取出的类

这样一来虽说在接口层面仍然违反单一职责原则，但在实现层面情况就好一些了。

那么，如何在接口层面也解决这个问题呢？那就要困难一点了。一般步骤是首先看看被委托任务的那些类是否可以直接被客户使用。例如，假设只有某些客户对运行ScheduledJobs感兴趣，则可以像这样进行重构（图20-13）：

262

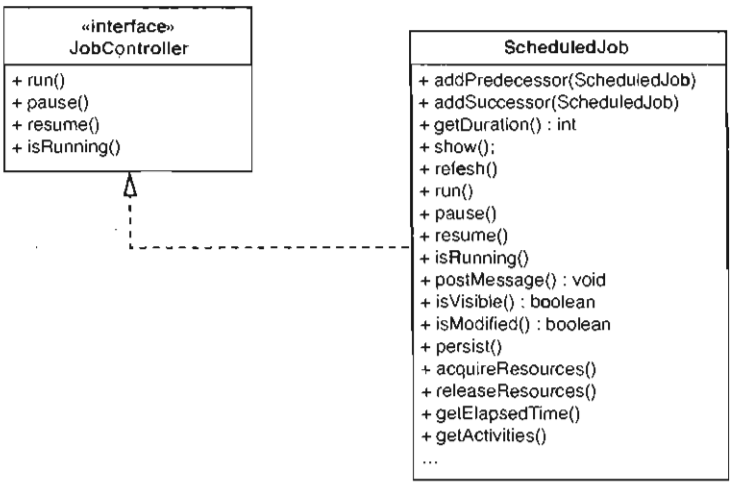


图20-13 给ScheduledJob做一个客户相关的接口

现在，那些只想对任务进行控制的客户就可以通过JobController接口来使用ScheduledJob对象了。这种为一组特定用户量身定做一个接口的设计手法能够保持你的设计符合接口隔离原则（ISP）。

接口隔离原则（ISP）

如果一个类体积较大，那么很可能它的客户并不会使用其所有方法。通常我们会看到特定用户使用特定的一组方法。如果我们给特定用户使用的那组方法创建一个接口，并让这个大类实现该接口，那么用户便可以使用“属于它的”那个接口来访问我们的类了。这种做法有利于信息隐藏，此外也减少了系统中存在的依赖。即当我们的大类发生改变的时候，其客户代码便不再需要重新编译了。

一旦为客户量身定做了接口之后，往往就可以开始将代码朝更好的方向改进了：原本是一个
263 大类，现在我们可以新建一个类，让这个类使用原来的大类，如图20-14所示：

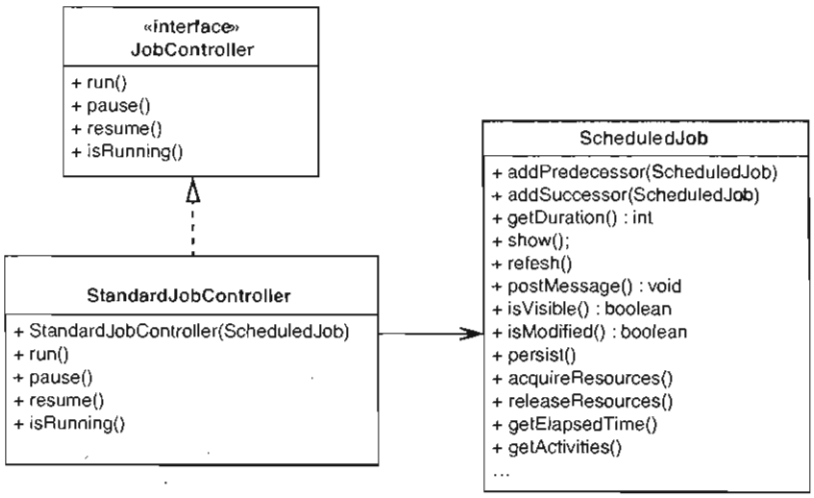


图20-14　隔离ScheduledJob的接口

这下我们不再是让ScheduledJob把任务委托给JobController，而是反过来，让后者委托前者。这样一来，不管什么时候，只要客户想要运行一个ScheduledJob，它就可以创建一个JobController，并传递一个ScheduledJob对象给它，然后便可以利用这个JobController来控制任务的执行。

不过真正做起来时，这种重构几乎总是比它听起来要更困难。通常，为了完成这一重构，你得在原类（ScheduledJob）上暴露更多的公有方法，以便让新的“前端”类（StandardJobController）能够访问到足够多的功能来完成它的工作。这类修改往往需要相当大的工作量。客户代码需要更改，这样它们才能从使用旧类转变为使用新类；而要想安全地完成这一更改，你又得先把有关的客户代码用测试护住。不过这一重构最大的好处就是它使得你能够一点点地对一

个大类的接口进行削减。在我们的例子中可以看到，经过重构之后的ScheduledJob 类上不再具有JobController里的那些方法了。

探索式方法#6：当所有方法都行不通时，作一点草稿式重构

如果实在很难看清一个类内部的职责，那么可以对它作一点草稿式重构。

264

草稿式重构（174页）是个强大的手段。但是，你得记住，草稿式重构只能看作是一次“模拟演习”。你在草稿式重构时看到的代码并不一定是后面真正重构时得到的。

探索式方法#7：关注当前工作

注意你目前手头正在做的事情。如果发现你自己正在为某件事情提供另一条解决方案，那么可能便意味着这里面存在一个应该被提取并允许替代的职责。

人们很容易就会被类里面数量庞大的职责吓住。别忘了，你正在进行的修改就等于是在告诉你该软件可以按某种特定的方式改变。通常只要认识到这种改变的方式就足以将你编写的新代码看作一个独立的职责了。

20.2 其他技术

上文介绍的探索式方法可以很好地帮助你深入了解并寻找出旧类里面的新抽象，但说到底它们只是小技巧。要想真正很好地认识你的代码内的职责，还是只有多读，包括读关于设计模式的书，更重要的还是读其他人编写的代码。花点时间来阅读开源项目的源代码，看看其他人是怎么做的。读的时候注意别人是如何命名系统里面的类的，此外还有类名与方法名之间的对应关系。渐渐地，你就会在识别出隐藏职责方面更加得心应手，并且在浏览不熟悉的代码的时候也能够发现它们。

20.3 继续前进

当你在一个大类中识别出了一系列不同的职责之后，就只剩下两个问题要解决了：战略和战术。先来看看战略。

20.3.1 战略

在识别出了所有独立的职责之后，我们该干些什么呢？应该花上一周的时间把这些大类都重整一遍吗？应该把它们都化整为零吗？如果你有这个时间，那再好不过。但实际上这种可能性很小，而且这一过程也有一定风险。就我所见过的有关案例中，几乎毫无例外的，当一个团队进行一番大规模重构时，系统稳定性就会有一段时间的下跌，即便他们干得再仔细，一边编写测试一边重构，还是如此。当然，如果你们仍处在发布周期的早期，愿意承担这样的风险，且有时间，那么一次大规模重构也并非不可。只要记得别让那些bug搞得你不再想去做其他重构就是了。

265

要分解大型的类，最佳步骤就是先确认出职责，并确保团队里的每个人都理解了这些职责，然后在需要的时候再去对该类进行分解。这种做法其实就是把修改的风险给减小了，从而让你能够在这个过程中同时完成其他事情。

20.3.2　战术

对于大多数遗留系统而言，一开始你所能奢望的也就是能够在实现层面运用单一职责原则了：本质上这意味着从大类中提取出新类，然后把一些任务委托给这些新类。另一方面，在接口层面引入SRP就要麻烦一些了。类的客户代码需要改动，所以你得先给它们编写测试。不过还算好的是，实现层面SRP的引入有助于后面在接口层面的引入。让我们先来看一看实现层面的内容。

具体用什么技术来进行类提取取决于众多因素。其一就是要考虑给那些被影响到的方法编写测试的难易程度。最好先看一看你的类，列出所有需要被移动的成员变量和方法。这么一来，对于应该给哪些方法编写测试，你心里就应该有数了。就拿我们前面提到的那个RuleParser类来说，如果考虑分解出一个TermTokenizer类，那么可能需要将string类型的成员变量current、int型的成员变量currentPosition、以及方法hasMoreTerms和nextTerm移动到新类中。然而，由于hasMoreTerms和nextTerm都是私有方法，因此我们无法直接为它们编写测试。当然，我们可以先将它们改为公有的（反正它们要“搬家”了嘛）再进行测试的编写，但在测试用具里头创建一个RuleParser并通过它对一组字符串求值可能也同样容易。完成这些之后我们便有了覆盖hasMoreTerms和nextTerm的测试，从而能够安全地将它们转移到新类当中去。

遗憾的是，许多大型的类都很难在测试用具中实例化。关于这个问题，参考第9章，里面有一组可以用于解决这类问题的技术。如果你已经能够在测试用具中实例化你的类，则仍可能需要利用第10章所讲的技术来帮助你将测试安置到位。

一旦将测试安置到位，你便可以用一种非常直观的方式来提取类了，即利用Martin Fowler在《重构：改善既有代码的设计》一书中提到的类提取重构。然而，如果无法将测试安置到位的话，仍然还是有办法的，尽管可能就要危险一些了。下面就是在测试无法安置到位的情况下可以采取的步骤（注意，这是一个非常保守的策略，并且无论你手头是否有重构工具，该方案都可行）。

266

(1) 确定出一个你想要分离到另一个类当中的职责。

(2) 弄清是否有成员变量需要被转移到新类当中。有的话就将它们放到类体内的一个单独的声明区段，跟其他成员变量区分开来。

(3) 如果一个方法需要整个儿被移至新类中，则将其函数体提取出来，放入新方法，别忘了新方法的名字要跟旧方法一致，除了一点不同，即新方法的名字前加了个前缀，比如MOVING（全大写）。如果你没有用重构工具，那么要记住在做方法提取的时候利用签名保持（249页）手法。最后，每提取一个方法就把它放入前面所说的那个单独的声明区段，即跟待转移的那些成员变量放在一起。

(4) 倘若一个方法只有一部分需要被转移，就将它们从“原住地”提取出来，同样利用MOVING前缀来标识它们栖身的新方法，并将后者放至那个单独的区段。

(5) 到此，类里面应该就有一个区段放的全是待转移的成员变量和方法了。在当前类和当前类的所有派生类中作一次文本搜索，确保你要转移的成员变量里面不存在某个被要转移的方法集

之外的方法使用到的变量。这一步不应依靠编译器（251页），恪守这一点很重要，因为在许多面向对象语言中，一个派生类里面可以声明与基类里某个成员变量同名的成员变量。这通常被称为名字隐藏。因此如果你的类里面有变量隐藏了其基类中的同名变量，同时（除了你要转移的方法之外）还有一些方法是使用了该变量的，那么贸然将该变量转移走可能就会改变代码的行为[1]。同样道理，如果你依靠编译器来试图发现那些使用了你即将移走的变量的方法，那么一旦你移走的某个变量名字隐藏了基类中的某个变量，则你的编译器将不会帮你发现所有的这类方法——将一个名字隐藏变量注释掉然后重编译只会让那些使用它的方法转而使用基类中同名的变量而已。

(6) 到此，你已经可以将前面分离好的成员变量和方法一并转移到新类当中去了。然后在原类中创建新类的对象，接着就依靠编译器来发现哪些地方应该调用新类上的那些方法吧。 267

(7) 在完成“搬家”工作，并且代码也顺利编译了之后，便可以将新类上那些方法的MOVING前缀去掉了。最后，还是依靠编译器帮你找到那些需要相应修改名字的地方[2]。

前面这番重构的步骤是相当复杂的，但如果你面对的本来就是一堆非常复杂的代码，则要想不用测试就安全地将它们提取出来，这样复杂的步骤还是必要的。

在没有测试的情况下进行类提取可能会遇到一些不测。其中最难觉察的就是可能会引入与继承有关的bug。将一个方法从一个类移至另一个类是相当安全的，你可以依靠编译器的帮助，但对于大多数语言来说，倘若你试图移走一个重写了另一方法的方法，情况就不妙了，原类中对这个被移走方法的调用将会变成对基类中同名方法的调用。对于成员变量也存在类似的现象。派生类中的成员变量会隐藏基类中的同名变量。移走前者便会让后者暴露出来。

因此，为了解决这个问题，我们根本就不移走那个方法。而是通过提取原方法的函数体来创建新的方法。其中给新方法加的前缀只是为了避免在移走它们之前与原方法产生名字冲突。另一方面，成员变量的情况则要微妙一些：我们得在移走它们之前手动搜寻哪些方法使用了它们。这一过程是有可能出错的，所以须得非常小心，建议跟你的伙伴一起完成这一工作。

20.4 类提取之后

对一个庞大的类进行类提取通常是个好的开始。在实践当中我们发现，团队在做这项工作时面临的最大危险便是野心过大。你可能已经对这个类做了一些草稿式重构（174页），或是已经对该系统应该成什么样子建立起了一些其他的看法。但是请记住，你目前的系统结构在应用当中能够工作，它提供了软件的功能；它可能还没有准备好向前迈进。有时候，你所能做到的也就是建立起对一个大型类在重构之后的样子的“愿景”，然后再把这个愿景给放一边去。这么做只是为了去发现什么是可能的。要想真正让代码向前推进，你得敏锐地关注代码中的现状，并小心地向前改进（不一定要朝着你心目中理想的设计改进，但至少可以向更好的方向改进）。 268

1. 因为那些原先使用该变量的方法将会转而使用基类中的同名变量了（如果可访问的话）。——译者注
2. 调用点等。——译者注

第21章 需要修改大量相同的代码

在对付遗留系统时，也许没有比遇到下面的事情更令人沮丧的了。比如，你需要做一处修改，看到了需要修改的地方，但接下来你却发现另一处一模一样的地方也需要作相应的修改，然后又是另一处，另一处……，因为你的系统里面其他很多地方的代码都与这个地方的几乎一模一样。你可能会觉得，如果对系统做一些重构的话这样的问题可能就不会存在了，但谁又有这个时间呢？于是也只能让这个问题在系统里面搁着，看着心烦却又无计可施。

如果你会重构的话，情况就会好一些。你会知道消除重复代码并不一定要像重新设计或重新架构那样大张旗鼓，消耗大量精力；而是可以在工作的过程中小块小块地进行修改。这样系统的状况就会逐步得到改善；当然，前提是别人不要跟在你后面继续往里猛塞重复代码才行——不过就算这一情况不幸发生了，也仍然还是有办法解决的（别玩真人PK就成☺），但这就不是我们这里应该讨论的问题了。关键是，值不值得这么做。把一块代码里面的重复成分挤干对我们来说到底有什么好处？我想答案会让你吃惊。先让我们来看一个例子。

现有一个不大的用Java写的网络系统，我们需要用它来发送命令到一个服务器端。有两个命令类，分别叫AddEmployeeCmd和LogonCommand。当需要发起一个命令时，我们就实例化相应的命令类并传一个输出流给它的写方法。

下面就是这两个命令类的定义，从中你能看出有哪些重复代码吗？

```
import java.io.OutputStream;

public class AddEmployeeCmd {
    String name;
    String address;
    String city;
    String state;
    String yearlySalary;

    private static final byte[] header = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x02};
    private static final byte[] footer = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;

    private int getSize() {
        return header.length +
```

```
                SIZE_LENGTH +
                CMD_BYTE_LENGTH +
                footer.length +
                name.getBytes().length + 1 +
                address.getBytes().length + 1 +
                city.getBytes().length + 1 +
                state.getBytes().length + 1 +
                yearlySalary.getBytes().length + 1;
    }

    public AddEmployeeCmd(String name, String address,
                          String city, String state,
                          int yearlySalary) {
        this.name = name;
        this.address = address;
        this.city = city;
        this.state = state;
        this.yearlySalary = Integer.toString(yearlySalary);
    }

    public void write(OutputStream outputStream)
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        outputStream.write(name.getBytes());
        outputStream.write(0x00);
        outputStream.write(address.getBytes());
        outputStream.write(0x00);
        outputStream.write(city.getBytes());
        outputStream.write(0x00);
        outputStream.write(state.getBytes());
        outputStream.write(0x00);
        outputStream.write(yearlySalary.getBytes());
        outputStream.write(0x00);
        outputStream.write(footer);
    }
}
import java.io.OutputStream;

public class LoginCommand {
    private String userName;
    private String passwd;
    private static final byte[] header
                = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x01};
    private static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;

    public LoginCommand(String userName, String passwd) {
        this.userName = userName;
        this.passwd = passwd;
```

270

```
    }

    private int getSize() {
        return header.length +  SIZE_LENGTH +  CMD_BYTE_LENGTH +
                footer.length + userName.getBytes().length + 1 +
                passwd.getBytes().length + 1;
    }

    public void write(OutputStream outputStream)
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        outputStream.write(userName.getBytes());
        outputStream.write(0x00);
        outputStream.write(passwd.getBytes());
        outputStream.write(0x00);
        outputStream.write(footer);
    }
}
```

271 图21-1是相应的UML图。

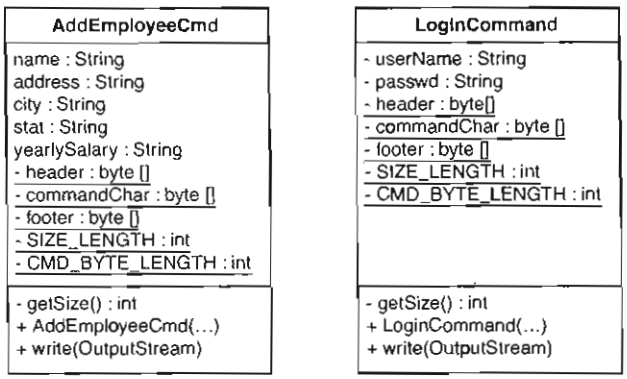


图21-1　AddEmployeeCmd和LoginCommand

看起来重复成分还不少，但那又怎样呢？反正代码量又不多。我们可以重构，切出重复成分，缩减代码量。但是，这么做是不是就让我们的工作变得容易些了呢？可能是，也可能不是；光是就这样看上两眼脑袋里想一想是很难给出答案的。

那么就让我们试试看吧。首先找出那些重复的地方，删除它们，然后看看得到的代码是什么样子的。这样我们便可以判断我们的重复消除能否带来帮助了。

开始之前，首先需要一组测试，以便在每次重构之后都能够运行它们来确保一切正常。不过为了叙述的简洁，这里就不描述这组测试了，你只需记住它们在哪儿就可以了。

开始步骤

面对代码重复，我的第一反应便是“后退一步”好对它有一个更整体的认知。这么做时，我便开始思考最终得到的类会是什么样子的，以及抽取出来的重复代码成分看起来会是怎样的。然后我意识到自己想得太多了。其实只需从删除小块重复代码开始，而且它使得我们以后更容易看到更大块的重复成分。例如，在LoginCommand的写方法中，有如下的代码：

```
outputStream.write(userName.getBytes());
outputStream.write(0x00);
outputStream.write(passwd.getBytes());
outputStream.write(0x00);
```

从以上代码可以看出，每写出一个字符串，结尾都会跟上一个结束符(0x00)。于是我们可以这样来提取这里的重复成分：创建一个名叫writeField的新方法，该方法接受一个字符串和一个输出流，然后负责将字符串写入流中，并写入一个结束符。 272

```
void writeField(OutputStream outputStream, String field) {
   outputStream.write(field.getBytes());

   outputStream.write(0x00);
}
```

决定从哪开始

经过一系列的重构，终于消除了重复之后，最终得到的代码结构是什么样子的，还是得取决于我们最初是从哪开始的。例如，假设我们有如下的方法：

```
void c() { a(); a(); b(); a(); b(); b(); }
```

我们可以这样来分解它：

```
void c() { aa(); b(); a(); bb(); }
```

或者也可以这样分解：

```
void c() { a(); ab(); ab(); b(); }
```

问题是，该选哪一个呢？事实上，两者结构并没有多大区别。两种分组法都比原来的要好，而且如果需要的话我们大可从其中的一种分组重构至另一种分组。以上并非最终决定。具体如何决定要看名字。如果我能给负责重复两次调用a()方法的新方法(上面的aa())找个好名字，而且这个名字在其上下文中比我们能给ab()找出的名字要更好的话，我们就会采用前者。

我使用的另一个启发式策略就是迈小步。如果有些很小的重复是可以消除的，那么我就先把它们搞定，往往这能够使整个大图景变得明朗起来。

有了这个新方法之后，我们便可以将每个成对的“串/结束符”写入都替换成单个调用，同时别忘了每次都要运行测试来确保一切正常。下面就是LoginCommand的write方法修改之后的样子：

```
public void write(OutputStream outputStream)
```

```
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeField(outputstream, username);
    writeField(outputStream, passwd);
    outputStream.write(footer);
}
```

273 这便解决了LoginCommand类的问题，但还剩下AddEmployeeCmd类呢？AddEmployeeCmd里面同样也存在重复的串/结束符写入序列。由于这两个类都是命令(Command)类，因此我们可以考虑引入一个公共基类，比如就叫Command。有了这个公共基类之后，便可以将writeField放在其中供这两个派生类使用。

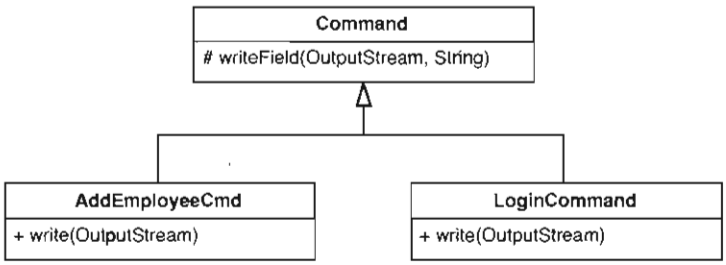


图21-2　命令类继承体系

现在我们可以回到AddEmployeeCmd，并将它里面的串/结束符写入序列也都替换为对writeField的调用了。修改之后AddEmployeeCmd的写方法看起来像这样：

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeField(outputStream, name);
    writeField(outputStream, address);
    writeField(outputStream, city);
    writeField(outputStream, state);
    writeField(outputStream, yearlySalary);
    outputStream.write(footer);
}
```

LoginCommand的写方法如下：

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeField(outputstream, userName);
    writeField(outputStream, passwd);
    outputStream.write(footer);
```

```
}
```

274

现在代码干净一些了，但还没有结束呢。AddEmployeeCmd和LoginCommand的写方法形式上其实是一样的：依次是写入命令的首部（header）、大小（size）以及命令符（command Char）；然后是写入一组字段；最后是写上尾部（footer）。如果我们可以将它们之间的差异成分抽取出来，就能够得到如下写方法（以LoginCommand的为例）。

```
public void write(OutputStream outputStream)
            throws Exception {
   outputStream.write(header);
   outputStream.write(getSize());
   outputStream.write(commandChar);
   writeBody(outputstream);
   outputStream.write(footer);
}
```

其中writeBody如下：

```
private void writeBody(OutputStream outputStream)
            throws Exception {
   writeField(outputstream, userName);
   writeField(outputStream, passwd);
}
```

AddEmployeeCmd的写方法从形式上跟上面的那个完全一样，只不过writeBody的定义有所不同。

```
private void writeBody(OutputStream outputStream) throws Exception {
   writeField(outputStream, name);
   writeField(outputStream, address);
   writeField(outputStream, city);
   writeField(outputStream, state);
   writeField(outputStream, yearlySalary);
}
```

如果两个方法看上去大致相同，则可以抽取出它们之间的差异成分。通过这种做法，我们往往能够令它们变得完全一样，从而消除掉其中一个。

现在，这两个类的写方法看上去完全相同了。那么我们是否可以将这个写方法提升到Command基类中了呢？等等。就算两个方法“看上去”完全一样了，但它们所使用到的成员数据仍然还是属于它们各自的类的：header、footer以及commandChar。如果想要实现单一的写方法，则这个写方法必须能够调用其派生类的特定方法来获取相应的数据成员才行。因此就让我们来考察一下AddEmployeeCmd和LoginCommand中的数据成员吧。

```
public class AddEmployeeCmd extends Command {
    String name;
    String address;
    String city;
    String state;
    String yearlySalary;
    private static final byte[] header
```

275

```
                = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x02};
    private static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;
    ...
}

public class LoginCommand extends Command {
    private String userName;
    private String passwd;

    private static final byte[] header
                = {(byte)0xde, (byte)0xad};
    private static final byte[] commandChar = {0x01};
    private static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;
    ...
}
```

看得出来，这两个类里面的公共数据还是挺多的，比如header、footer、SIZE_LENGTH以及CMD_BYTE_LENGTH，由于它们的值对应相同，因此我们可以将它们全都提升到Command类中来。为了重新编译和测试的目的，我们先将它们设为受保护的。

```
public class Command {
    protected static final byte[] header
                = {(byte)0xde, (byte)0xad};
    protected static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    protected static final int SIZE_LENGTH = 1;
    protected static final int CMD_BYTE_LENGTH = 1;
    ...
}
```

现在，我们所关心的就只剩下两个派生类中的commandChar变量了。该变量在两个派生类中是不同的。解决这一问题的办法之一是在Command类上引入一个抽象的获取方法。

```
public class Command {
    protected static final byte[] header
                = {(byte)0xde, (byte)0xad};
    protected static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    protected static final int SIZE_LENGTH = 1;
    protected static final int CMD_BYTE_LENGTH = 1;
    protected abstract char [] getCommandChar();
    ...
}
```

276

这样一来，我们只需在每个相应的派生类上使用重写的getCommandChar方法来表示

commandChar变量即可：

```
public class AddEmployeeCmd extends Command {
    protected char [] getCommandChar() {
        return new char [] { 0x02};
    }
    ...
}

public class LoginCommand extends Command {
    protected char [] getCommandChar() {
        return new char [] { 0x01};
    }
    ...
}
```

没问题，好。是时候把写方法提升到Command基类中去了。完成之后我们的Command类看上去像这样。

```
public class Command {
    protected static final byte[] header
                = {(byte)0xde, (byte)0xad};
    protected static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    protected static final int SIZE_LENGTH = 1;
    protected static final int CMD_BYTE_LENGTH = 1;

    protected abstract char [] getCommandChar();

    protected abstract void writeBody(OutputStream outputStream);

    protected void writeField(OutputStream outputStream,
                              String field) {
        outputStream.write(field.getBytes());
        outputStream.write(0x00);
    }

    public void write(OutputStream outputStream)
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        writeBody(outputstream);
        outputStream.write(footer);
    }
}
```

277

注意，我们还引入了一个抽象的writeBody方法（见图21-3）。

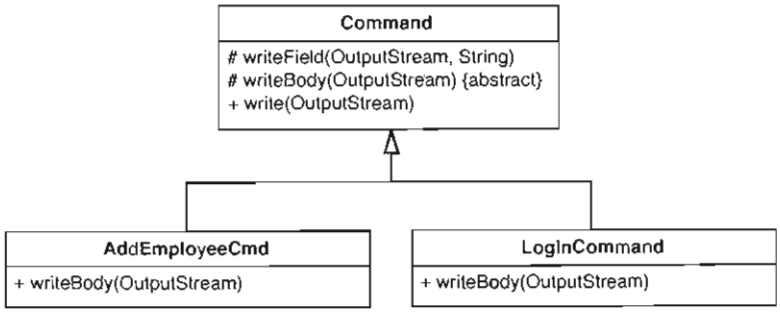


图21-3　提升writeField

将写方法提升至Command基类之后，剩下唯一的任务就是每个派生类中的getSize方法、getCommandChar方法，以及构造函数了。下面是我们的LoginCommand类：

```
public class LoginCommand extends Command {
    private String userName;
    private String passwd;

    public LoginCommand(String userName, String passwd) {
        this.userName = userName;
        this.passwd = passwd;
    }

    protected char [] getCommandChar() {
        return new char [] { 0x01};
    }

    protected int getSize() {
        return header.length +  SIZE_LENGTH + CMD_BYTE_LENGTH +
               footer.length + userName.getBytes().length + 1 +
                passwd.getBytes().length + 1;
    }
}
```

278

这是个相当简洁清爽的类。AddEmployeeCmd也与此相似：一个getSize方法、一个getCommandChar方法，几乎就这些。现在让我们稍微仔细地看一下getSize方法。

这是LoginCommand的：

```
protected int getSize() {
   return header.length +  SIZE_LENGTH +
           CMD_BYTE_LENGTH + footer.length +
           userName.getBytes().length + 1 +
           passwd.getBytes().length + 1;
}
```

这是AddEmployeeCmd的：

```
private int getSize() {
   return header.length +  SIZE_LENGTH +
           CMD_BYTE_LENGTH + footer.length +
```

```
            name.getBytes().length + 1 +
            address.getBytes().length + 1 +
            city.getBytes().length + 1 +
            state.getBytes().length + 1 +
            yearlySalary.getBytes().length + 1;
}
```

这两个getSize的相同点有哪些，不同点又有哪些呢？看起来它们都将首部长度、SIZE_LENGTH、命令字节长度以及尾部长度计算进去了。然后剩下的就是每个字段的长度，在这一点上两者不同。那么，若是我们将这个不同点提取出来呢？把getBodySize()作为计算各字段长度总和的函数，于是：

```
private int getSize() {
   return header.length +  SIZE_LENGTH
      +  CMD_BYTE_LENGTH + footer.length + getBodySize();
}
```

这么一来，两者的getSize()就又一样了。getSize的逻辑是所有的簿记（bookkeeping）数据的长度，加上命令消息体的长度（即所有字段的长度总和）。这么一来我们便又可以将getSize提升到Command基类当中去了，而不同的派生类只需实现不同的getBodySize即可（见图21-4）。

279

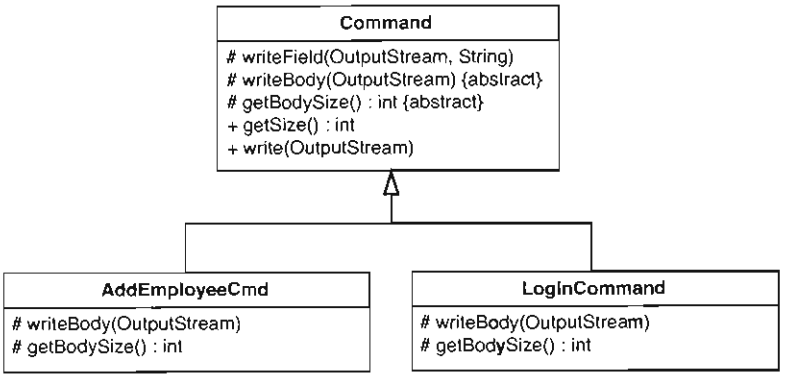


图21-4　提升getSize

现在，停下来看看我们走到哪了。AddEmployeeCmd的getBodySize实现如下：

```
protected int getBodySize() {
   return name.getBytes().length + 1 +
          address.getBytes().length + 1 +
          city.getBytes().length + 1 +
          state.getBytes().length + 1 +
          yearlySalary.getBytes().length + 1;
}
```

看起来，我们还是漏掉了一些相当讨厌的重复代码。虽然数量不多，但就让我们再热心一回，把它们完全消除掉看看。

```
protected int getFieldSize(String field) {
   return field.getBytes().length + 1;
}

protected int getBodySize() {
   return getFieldSize(name) +
          getFieldSize(address) +
          getFieldSize(city) +
          getFieldSize(state) +
          getFieldSize(yearlySalary);
}
```

只需将上面的getFieldSize方法提升到Command基类中，便可以在LoginCommand的getBodySize方法中利用它了。

```
protected int getBodySize() {
   return getFieldSize(name) + getFieldSize(password);
}
```

280

还有什么重复的成分吗？实际上还有，但已经不多了。我们注意到LoginCommand和AddEmployeeCmd都是接受一组参数，获取它们的大小，然后将它们写出。除了commandChar变量之外，这两个类也就只剩这点差别了：我们能否通过一点泛化来解决掉这一重复呢？事实上，通过在基类中引入一个链表，我们便可以在每个派生类中的构造函数里面这样做。

```
class LoginCommand extends Command
{
    ...
    public AddEmployeeCmd(String name, String password) {
        fields.add(name);
        fields.add(password);
    }
    ...
}
```

在派生类的构造函数填充完这个字段列表之后，我们便可以用同样的代码来获取消息体的大小了。

```
int getBodySize() {
     int result = 0;
     for(Iterator it = fields.iterator(); it.hasNext(); ) {
        String field = (String)it.next();
        result += getFieldSize(field);
     }
     return result;
}
```

类似的，writeBody方法看起来可能像这样。

```
void writeBody(Outputstream outputstream) {
   for(Iterator it = fields.iterator(); it.hasNext(); ) {
      String field = (String)it.next();
      writeField(outputStream, field);
   }
```

```
}
```

现在我们可以将这些方法统统都提升到Command基类中了。这样便真正消除了所有的重复。以下便是Command类修改后的样子。注意Command里面的那些不再被派生类访问的方法都已经改成私有的了。

```
public class Command {
    private static final byte[] header
                = {(byte)0xde, (byte)0xad};
    private static final byte[] footer
                = {(byte)0xbe, (byte)0xef};
    private static final int SIZE_LENGTH = 1;
    private static final int CMD_BYTE_LENGTH = 1;

    protected List fields = new ArrayList();
    protected abstract char [] getCommandChar();

    private void writeBody(Outputstream outputstream) {
        for(Iterator it = fields.iterator(); it.hasNext(); ) {
            String field = (String)it.next();
            writeField(outputStream, field);
        }
    }

    private int getFieldSize(String field) {
        return field.getBytes().length + 1;
    }

    private int getBodySize() {
        int result = 0;
        for(Iterator it = fields.iterator(); it.hasNext(); ) {
            String field = (String)it.next();
            result += getFieldSize(field);
        }
        return result;
    }

    private int getSize() {
        return header.length +  SIZE_LENGTH
            +  CMD_BYTE_LENGTH + footer.length
            + getBodySize();
    }

    private void writeField(OutputStream outputStream,
                            String field) {
        outputStream.write(field.getBytes());
        outputStream.write(0x00);
    }

    public void write(OutputStream outputStream)
                throws Exception {
        outputStream.write(header);
        outputStream.write(getSize());
        outputStream.write(commandChar);
        writeBody(outputstream);
```

281

```
        outputStream.write(footer);
    }
}
```

再看看我们的LoginCommand和AddEmployeeCmd，简直太简洁了：

```
public class LoginCommand extends Command {
    public LoginCommand(String userName, String passwd) {
        fields.add(username);
        fields.add(passwd);
    }

    protected char [] getCommandChar() {
        return new char [] { 0x01};
    }
}

public class AddEmployeeCmd extends Command {
    public AddEmployeeCmd(String name, String address,
                          String city, String state,
                          int yearlySalary) {
        fields.add(name);
        fields.add(address);
        fields.add(city);
        fields.add(state);
        fields.add(Integer.toString(yearlySalary));
    }

    protected char [] getCommandChar() {
        return new char [] { 0x02 };
    }
}
```

图21-5是我们最终成果的UML图展示。

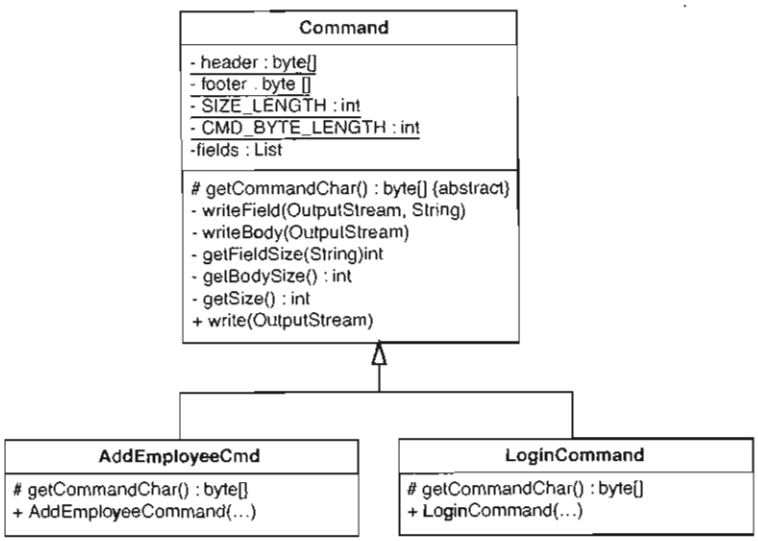


图21-5　消除了重复代码之后的Command继承体系图

现在我们到哪了呢？消除掉了那么多的重复代码，以致于我们的两个命令类简直都成了一层空壳了。所有的功能性代码都转移到了公共基类Command中。实际上，如果你开始怀疑究竟是否要给这两种命令分别引入各自的类，你的怀疑是不无道理的。那么替代方案又是什么呢？

我们可以拿掉这两个派生类，并往Command类中加入一个静态方法，该方法允许我们发送一个命令。

```
List arguments = new ArrayList();
arguments.add("Mike");
arguments.add("asdsad");
Command.send(stream, 0x01, arguments);
```

但这样一来对用户来说又显得太麻烦了。有一件事是肯定的：我们肯定得发送两个不同的命令字符，而且我们不想让用户来管理这些字符。

另一种方案就是为每种命令添加一个单独的静态方法。

```
Command.SendAddEmployee(stream,
         "Mike", "122 Elm St", "Miami", "FL", 10000);

Command.SendLogin(stream, "Mike", "asdsad");
```

但这又使得所有的客户代码都得作出相应改动。就目前来说，代码中创建AddEmployeeCmd和LoginCommand对象的地方还真不少。

或许最好的办法还是就让这两个类留在那儿。的确，它们小到不够资格，但那又如何呢？反正它们又没有带来什么坏的影响，是吧。

完了吗？还没有。有一件事情我们还没做，实际上早该做了。那就是我们可以将AddEmployeeCmd重命名为AddEmployeeCommand。这样两个命令类的名字就一致了。命名的一致性可以减少出错的机会。

缩写

类名和方法名缩写是问题的来源之一。缩写风格一致的话倒还好，但总的来说我不喜欢这种做法。

我曾遇到过这样的一个团队，该团队试图在它们系统中的几乎每一个类的类名里面使用manager和management这两个字眼。这种命名习惯没带来好处，更糟的是它们居然还用了好多种不同的缩写法。例如，有些类被命名为XXXXMgr，而同时另一些类则被命名为XXXXMngr。这就使得想要使用这些类的人几乎每次都要把类翻出来看看以确信自己使用了正确的类。而对于我个人来说，要猜测某个特定的类用的应该是哪个后缀，有一半以上的时候猜错。

284

好了，现在我们已经将所有重复成分消除。剩下的问题就是检验它是否真的带来了好处。为此让我们来设想几个场景。场景一：当我们需要增加一个新命令时，我们该怎么办？我们可以从Command派生出一个新类，创建该类的对象即可。而如果是在消除重复之前的系统里呢？我们便需要创建一个新的命令类，然后从另一个命令类那儿剪切/粘贴一些代码，修改变量等等。而这又会引入更多的重复，并让事情变得更糟；除此之外还更容易出错。比如可能会弄错了变量什么

的。而且在消除重复之前，这一过程绝对要花上更长一点的时间。

那么，重复成分的消除是否导致我们丢掉了一部分灵活性呢？如果我们必须得发送非字符串类的命令该怎么办呢？其实，从某种程度上说我们已经解决了这个问题。AddEmployeeCommand本身就能接受一个整型参数，并将其转换为字符串然后作为命令发送。对于其他任何类型，我们也可以用同样的做法。因为要想发送一个命令，反正总是要转换为字符串形式的。我们可以在具体派生类的构造函数中完成这一步骤。

另一个问题：如果我们需要另一种形式的命令怎么办？比如说一个可以将其他命令嵌套进命令体的命令。这个问题其实也可以轻易解决。只需从Command派生一个新类，重写其writeBody方法：

```
public class AggregateCommand extends Command
{
    private List commands = new ArrayList();
    protected char [] getCommandChar() {
        return new char [] { 0x03 };
    }

    public void appendCommand(Command newCommand) {
        commands.add(newCommand);
    }

    protected void writeBody(OutputStream out) {
        out.write(commands.getSize());
        for(Iterator it = commands.iterator(); it.hasNext(); ) {
            Command innerCommand = (Command)it.next();
            innerCommand.write(out);
        }
    }
}
```

剩下的就不用你操心了。

现在，想象一下倘若不是消除了重复，情况会怎样吧。

这最后一个例子揭示出了一些非常重要的东西。即当消除类之间的重复成分时，会得到一些很小，功能很集中的方法。每一个方法所做的事情都跟其他方法不同，这其实便意味着一个极大
285 的好处：正交性。

正交性说白了其实就是独立性（无关性）。比如说你想要修改代码的既有行为，然后发现只需到代码中的一处地方改一下就成了。这就是正交性。如果把我们的应用比作外面嵌着一些旋钮的盒子，那么具有正交性的应用就好比是一个行为只对应一个旋钮，调节起来很容易。反之如果重复情况很严重的话，就可能会出现一个行为涉及多个旋钮的情况。就拿前面例子中的字段写出来说，在一开始的设计中，如果我们想改用0x01作为结束符（原来是0x00），我们就得“遍历”代码，将每一处使用0x00的地方都改成0x01。此外设想有人要求我们用两个0x00作为结束符，那么情况也会变得相当糟糕。一句话，没有单一切入点。另一方面，在经过我们重构之后的代码中，要想改变字段的写出行为，只需编辑或重写writeField方法即可，对于一些像命令嵌套之类的特殊任务则可以重写writeBody方法。总之，一旦将行为局部化到了单个方法中，想要替换

或是修改它就很容易了。

本例中我们做了很多事情：把一些方法和变量从一个类移到另一个类、将方法分解，等等。但其中大部分事情都是机械的。我们只不过是注意到了代码中的重复情况然后将它们消除而已。真正具有创造性的事情其实只有一件，那就是给新方法想出合适的名字。原来的代码并没有字段或命令体的概念，但其实某种程度上这些概念是隐藏在代码中的。例如，有些变量是被特殊对待的，我们将它们称为字段。最后我们得到了一个优雅得多的正交设计，但我们并不觉得是在进行设计；而更多只是将一段代码改得更符合它的本意而已。

当你热情地去消除代码中的重复时，就会惊讶地发现，设计会自己浮现出来。是的，系统中的大部分正交性都无需你刻意去实现，它们自然而然就会出现。但这还不够完美，例如，考虑Command类上的如下方法：

```
public void write(OutputStream outputStream)
            throws Exception {
    outputStream.write(header);
    outputStream.write(getSize());
    outputStream.write(commandChar);
    writeBody(outputstream);
    outputStream.write(footer);
}
```

286

假设该方法像下面这样：

```
public void write(OutputStream outputStream)
            throws Exception {
    writeHeader(outputStream);
    writeBody(outputstream);
    writeFooter(outputStream);
}
```

现在我们的代码中就多出了两个新的切入点。一个是命令头的写出，另一个是命令尾的写出。我们当然可以根据需要往代码中添加这样的切入点，然而毕竟看到它们自然而然地出现还是令人欣喜的。

重复消除是锤炼设计的强大手段。它不仅能让设计变得更灵活，还能令代码修改更快更容易。

开放/封闭原则

开放/封闭原则是由Bertrand Meyer首先提出的。其背后的理念是，代码对于扩展应该是开放的而对于修改则应是封闭的。这就是说，对于一个好的设计，我们无需对代码作太多的修改就可以添加新的特性。

那么，具体到本章的例子，我们最后得到的代码具备这一要素吗？答案是肯定的。我们前面已经考察了一系列的修改场景，许多情况下只需要对极少的方法稍加修改即可。还有一些情况下则只需派生一个新类即可解决问题。当然，派生之后很重要的一点是要记得消除重复（见关于差异式编程的描述，其中介绍了如何通过派生来添加新特性以及通过重构来整合它们）。

此外，消除了重复之后，我们的代码往往便会自然而然地往开放/封闭原则靠拢了。

287

第22章 要修改一个巨型方法，却没法为它编写测试

在对付遗留代码时最麻烦的事情莫过于遇到庞大的方法了。许多时候可以通过新生方法（52页）和新生类（54页）手法来避免对长方法进行重构。但就算你逃掉了，也应该为不得不如此感到羞愧。长方法就像是代码基中的沼泽。不管什么时候，只要你试图去改变它，就得退一步先努力把情况弄清楚，然后再去进行修改。通常，如果代码比较干净的话修改起来要省时省事得多。

代码基里有长方法是件痛苦的事，但巨型方法就更恐怖了。所谓“巨型”方法就是指那些庞大复杂到你碰都不想去碰一下的方法。巨型方法可能包含成百上千行代码，其中还到处都是缩进，搞得你几乎无法浏览。所以，碰到这样的方法，你很可能会把它打印在一张长长的纸上，把纸摊在走廊里以便和你的同事们一起把它读懂。

有一次，我去参加一个会议，在跟朋友走回旅馆的时候，他们其中有一个说：“嗨，你一定得来看看这个。”他跑进房间拿出笔记本，给我看一个方法，这个方法长达千行以上。朋友知道我当时在研究重构，于是就说：“遇到这种方法你怎么办？”于是我们就开始想办法。我们知道测试是关键，但遇到这么巨型的方法，从哪里下手却成了问题。

本章所讲的就是从那以后我在这个问题上积累的经验。

22.1 巨型方法的种类

巨型方法不止一种。而且巨型方法的种类也并不都是能截然区分开来的。一般来说，我们看到的巨型方法都是几种不同特征的混合体（就像鸭嘴兽那样）。

22.1.1 项目列表式方法

项目列表式方法就是指那种几乎毫无缩进的方法，你只能看到罗列下来的一串仿佛项目列表似的代码块。其中也许有个别代码块中含有一定的缩进，但就整个函数范围来说是没什么缩进的。一般来说这种方法一眼扫过去是这样的（如图22-1所示）。

图22-1是最常见的一种项目列表式方法。如果运气好，可能会碰到写方法的人在各个代码区段之间加上几个回车或者加点注释说明一下各部分是做什么的。理想情况下，你应该可以将每个

代码区段提取为一个单独的方法，但通常代码并没有这么容易重构。看起来代码段之间是被一些空行隔开了，然而这个表象实际上是有欺骗性的，往往在一个区段声明的局部变量会在另一个区段被用到。这就导致方法分解往往并不像你想象的那样只需把代码拷贝粘贴出来就可以了。话虽如此，项目列表式方法跟巨型方法的其他“品种”比起来还算是稍微好一点的，这主要是因为代码中没有那种疯狂的缩进，后者往往会使阅读者找不着北。 290

```
void Reservation::extend(int additionalDays)
{
    int status = RIXInterface::checkAvailable(type, location, startingDate);

    int identCookie = -1;
    switch(status) {
        case NOT_AVAILABLE_UPGRADE_LUXURY:
            identCookie = RIXInterface::holdReservation(Luxury,location,startingDate,
                additionalDays +ddditionalDays);
            break;
        case NOT_AVAILABLE_UPGRADE_SUV:
        {
            int theDays = additionalDays + additionalDays;
            if (RIXInterface::getOpCode(customerID) != 0)
                theDays++;
            identCookie = RIXInterface::holdReservation(SUV,location,startingDate, theDays);
        }
        break;
        case NOT_AVAILABLE_UPGRADE_VAN:
            identCookie = RIXInterface::holdReservation(Van,
                location,startingDate, additionalDays + additionalDays);
           break;
        case AVAILABLE:
        default:
            RIXInterface::holdReservation(type,location,startingDate);
            break;
    }

    if (identCookie != -1 && state == Initial) {
        RIXInterface::waitlistReservation(type,location,startingDate);
    }

    Customer c = res_db.getCustomer(customerID);

    if (c.vipProgramStatus == VIP_DIAMOND) {
        upgradeQuery = true;
    }

    if (!upgradeQuery)
        RIXInterface::extend(lastCookie, days + additionalDays);
    else {
        RIXInterface::waitlistReservation(type,location,startingDate);
        RIXInterface::extend(lastCookie, days + additionalDays +1);
    }
        ...
}
```

图22-1　项目列表式方法 291

22.1.2　锯齿状方法

锯齿状方法就是指那些具有单个庞大的缩进块的方法。最简单的例子就是下面这样的，一个具有一个庞大的if块的方法（如图22-2所示）。

```
Reservation::Reservation(VehicleType type, int customerID, long startingDate, int days,
 XLocation l)
: type(type), customerID(customerID), startingDate(startingDate), days(days), lastCookie(-1),
state(Initial), tempTotal(0)
{
        location = l;
        upgradeQuery = false;

        if (!RIXInterface::available()) {
                RIXInterface::doEvents(100);
                PostLogMessage(0, 0, "delay on reservation creation");
                int holdCookie = -1;
                switch(status) {
                        case NOT_AVAILABLE_UPGRADE_LUXURY:
                                holdCookie = RIXInterface::holdReservation(Luxury,l,startingDate);
                                if (holdCookie != -1) {
                                   holdCookie |= 9;
                                }
                                break;
                        case NOT_AVAILABLE_UPGRADE_SUV:
                                holdCookie = RIXInterface::holdReservation(SUV,l,startingDate);
                                break;
                        case NOT_AVAILABLE_UPGRADE_VAN:
                                holdCookie = RIXInterface::holdReservation(Van,l,startingDate);
                                break;
                        case AVAILABLE:
                        default:
                                RIXInterface::holdReservation;
                                state = Held;
                                break;
                }
        }
        ...

}
```

图22-2　锯齿状方法——一个简单的例子

但以上这个还算好的，至少跟上面的项目列表式方法差不多。下面展示的这个（见图22-3）才是真正令人头大的。

要想判断你手头的方法是否属于这一类，最好的办法就是试图把方法内的代码块都按照缩进
292 格式格式化好。如果结果代码让你感到晕头转向，那么就是了。

大多数方法其实并不能严格归为上面所讲的两类，而更多的是属于两者的混合体。许多锯齿状方法在它们的深层的缩进区块中也会含有大段的项目列表式代码，但由于这些代码被层层嵌套
293 在了里面，因此很难给它们编写测试。对付锯齿状方法很具有挑战性。

在对长方法进行重构时，有没有重构工具会带来很大的区别。几乎每个重构工具都支持方法提取，因为有很多地方都可以利用这一支持。如果一个工具能够为你安全地提取方法，你就不需要自己编写测试去验证提取是否正确。工具代替了你的手工劳动，于是你可以专心致志地考虑如何使用方法提取来将一个方法重构成体面的样子，从而让后续工作更容易。

如果不幸没有自动提取方法的重构工具支持，则整理巨型方法可就更具挑战性了。通常这时候你的改动就得更保守一点，因为只有测试设置妥当了，相应的改动才能进行。

```
Reservation::Reservation(VehicleType type, int customerID, long startingDate, int days,
 XLocation l)
: type(type), customerID(customerID), startingDate(startingDate), days(days), lastCookie(-1),
state(Initial), tempTotal(0)
{
    location = l;
    upgradeQuery = false;

    while(!RIXInterface::available()) {
        RIXInterface::doEvents(100);
        PostLogMessage(0, 0, "delay on reservation creation");
        int holdCookie = -1;
        switch(status) {
            case NOT_AVAILABLE_UPGRADE_LUXURY:
                holdCookie =
                RIXInterface::holdReservation(Luxury,l,startingDate);
                if (holdCookie != -1) {
                    if (l == GIG && customerID == 45) {
                        // Special #1222
                        while (RIXInterface::notBusy()) {
                            int code =
                            RIXInterface::getOpCode(customerID);
                            if (code == 1 || customerID > 0)) {
                                PostLogMessage(1, 0, "QEX PID");
                                for (int n = 0; n < 12; n++) {
                                    int total = 2000;
                                    if (state == Initial || state == Held)
                                    {
                                        total += getTotalByLocation(location);
                                        tempTotal = total;
                                        if (location == GIG && days > 2)
                                        {
                                            if (state == Held)
                                                total += 30;
                                        }
                                    }
                                    RIXInterface::serveIDCode(n, total);
                                }
                            } else {
                                RIXInterface::serveCode(customerID);
                            }
                        }
                    }
                }
                break;
            case NOT_AVAILABLE_UPGRADE_SUV:
                holdCookie =
                RIXInterface::holdReservation(SUV,l,startingDate);
                break;
            case NOT_AVAILABLE_UPGRADE_VAN:
                holdCookie =
                RIXInterface::holdReservation(Van,l,startingDate);
                break;
            case AVAILABLE:
            default:
                RIXInterface::holdReservation(type,l,startingDate);
                state = Held;
                break;
        }
    }
    ...
}
```

图22-3　恶劣的情形

22.2　利用自动重构支持来对付巨型方法

如果你的工具能进行方法提取，那么得首先弄清楚什么是它能替你做的，什么是它不能替你做的。如今的大多数重构工具都能做简单的方法提取以及一些其他的重构，但人们在分解大型方法的时候往往需要更多的辅助重构，这时候就不是随便某个重构工具能胜任的了。举个例子，我们常常喜欢对语句进行重排，分组，以便将它们提取出来。对于这一需求，目前就没有任何工具能够进行必要的分析来判断给定的重排是否安全。这是件很可惜的事情，因为它可能成为bug的来源。

要想针对大型方法有效地运用重构工具，最好仅用工具进行一系列的修改，并避免对代码进行其他任何改动。这听起来似乎是用一根小指头就能完成的事情，但它的确带来了好处，就是能把那些已知为安全的修改和可能不安全的修改完全分离开来。在做这一阶段的重构时，你应当避免像语句重排和表达式分解这类修改，就算它们很简单也不行。此外，如果你的工具支持变量重命名，那固然很好，但如果它不支持，则你应该把这一工作推迟到后面再做。

> 在没有测试的情况下进行自动重构时，一定要只用工具进行（其中不要参杂手工修改）。而在一系列的自动重构完成之后，往往就可以将测试安置到位，并用这些测试来验证你所进行的任何手动修改了。

在做提取的时候，以下是你的主要目标：

(1) 将代码中的逻辑部分从尴尬的依赖中分离出来。

294 (2) 引入接缝，以后在重构时才能更容易地将测试安置到位。

下面是一个例子：

```
class CommoditySelectionPanel
{
    ...
    public void update() {
        if (commodities.size() > 0
                && commodities.GetSource().equals("local")) {
            listbox.clear();
            for (Iterator it = commodities.iterator();
                        it.hasNext(); ) {
                Commodity current = (Commodity)it.next();
                if (commodity.isTwilight()
                            && !commodity.match(broker))
                    listbox.add(commodity.getView());
            }
        }
        ...
    }
    ...
}
```

以上方法中有很多地方都是可以清理一下的。其中最怪异的事情之一就是该方法所做的“过

滤”工作是发生在一个面板（Panel）类上，按理说一个叫做面板的类应该只管显示才对。要想解开这段代码肯定是困难的。照现在的情况，要想编写测试的话，可以针对listbox的状态来写，但那样的话比原来的设计也好不了很多。

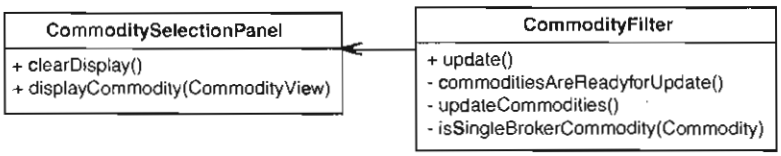
然而，如果有重构支持，情况就不一样了，我们可以在提升抽象层次的同时达到解开依赖的目的。比如，下面就是代码经过了一系列提取之后的样子：

```
class CommoditySelectionPanel
{
    ...
    public void update() {
        if (commoditiesAreReadyForUpdate()) {
            clearDisplay();
            updateCommodities();
        }
        ...
    }

    private boolean commoditiesAreReadyForUpdate() {
        return commodities.size() > 0
                && commodities.GetSource().equals("local");
    }

    private void clearDisplay() {
        listbox.clear();
    }

    private void updateCommodities() {
        for (Iterator it = commodities.iterator(); it.hasNext(); ) {
             Commodity current = (Commodity)it.next();)
            if (singleBrokerCommodity(commodity)) {
                displayCommodity(current.getView());
            }
        }
    }

    private boolean singleBrokerCommodity(Commodity commodity) {
        return commodity.isTwilight() && !commodity.match(broker);
    }

    private void displayCommodity(CommodityView view) {
        listbox.add(view);
    }

    ...
}
```

295

坦白的说，重构之后的代码从结构上看并没多大区别；仍然只不过是一个内含一些语句的if块。但不同的是，现在if块内的工作被委托给其他方法来完成了。现在我们的update方法就像是原来的update方法的骨架。但是，你会说，工具给自动起的那些方法名怎么办呢？它们看起来总有点古怪。别介意，你可以把它们当成一个好的开始，至少它们让你的代码能够从一个更高

的层面来传达语意了不是吗？而且它们还引入了接缝，这样的话后面我们就可以利用这些接缝来进行解依赖了。例如，我们可以运用子类化并重写方法手法来通过displayCommodity和clearDisplay进行感知。完成这些之后，我们可以考虑利用这些测试作后盾，创建一个新的显示类并将这些方法移到该类中。不过在本例中更妥当的办法是看看能否将update和updateCommodities转移到另一个类中；而将clearDisplay和displayCommodity留下，因为这样该类才能算是一个面板类，显示类。至于方法的重命名，则可以等到各个类都安放到位之后再进行。再加上一些重构，最终我们的设计看起来可能就像图22-4所示的这样。



296　图22-4　从CommoditySelectionPanel中提取出来的逻辑类

在使用自动工具进行方法提取时，有一点认识很重要，那就是藉此你可以完成许多粗糙的工作，之后，等其他测试安置到位了再去做那些细节的工作。在这个过程中，别太在意那些看上去跟当前类格格不入的方法，因为这样通常意味着之后还要进行类提取。具体怎么做可参考第20章。

22.3　手动重构的挑战

自动重构工具的支持可以让你无需做任何特殊准备就可以开始对大方法进行分解。好的重构工具能够帮你检查试图进行的重构是否安全，如果不安全的话就不予执行。但如果没有重构工具，就得靠自己来确保重构的正确性了，这时测试便成了最强的工具。

巨型方法使得测试、重构以及特性添加变得非常困难。但如果你能够在测试用具中实例化该方法所在的类，那就可以试着写出一组测试用例来保证你在分解该方法的过程中的安全。当然，如果该方法中的逻辑特别复杂，编写测试用例也可能会变成噩梦。但幸运的是，这种情况下我们可以求助于一系列的技术。在学习这组技术之前，让我们先来看看在方法提取时可能会犯的一些错误。

下面是一个简单的列表，虽然并不全面，但已经包含了最常见的错误。

(1) 我们可能会忘记向提取出来的方法传递变量。通常编译器会提醒我们这一错误（除非该变量跟某个成员变量重名），但我们还是可能会错误地认为该变量应当是一个局部变量，于是将其声明在了新方法的方法体中。

(2) 我们可能会给提取出来的方法起了一个会覆盖或重写基类中某个同名方法的名字。

(3) 我们可能会在传参或接受返回值的时候犯错。愚蠢的错误如返回了错误的值。更小一点的错误如返回或接受了错误的类型。

总之有很多地方可能会出错。本节的技术可以帮助你在没有测试的情况下更安全地进行方法
297　提取。

22.3.1　引入感知变量

我们可能不希望在对产品代码进行重构的时候往里面加入特性，但这并不意味着不能往里面添加任何代码。比如，可能会想要往一个类里面添加一个变量并使用它来感知待重构方法内的条件，而在完成了重构之后则可以将该变量删除，于是我们的代码就又回到了干净的状态。这一手法叫做*引入感知变量*。下面就是一个例子。我们想要重构Java类DOMBuilder上的一个方法。我们想要将这个方法清理一下，但不幸的是手头没有重构工具。

```
public class DOMBuilder
{
    ...
    void processNode(XDOMNSnippet root, List childNodes)
    {
        if (root != null) {
            if (childNodes != null)
                root.addNode(new XDOMNSnippet(childNodes));
            root.addChild(XDOMNSnippet.NullSnippet);
        }
        List paraList = new ArrayList();
        XDOMNSnippet snippet = new XDOMNReSnippet();
        snippet.setSource(m_state);
        for (Iterator it = childNodes.iterator();
                it.hasNext();) {
            XDOMNNode node = (XDOMNNode)it.next();
            if (node.type() == TF_G || node.type() == TF_H ||
                    (node.type() == TF_GLOT && node.isChild())) {
                paraList.addNode(node);
            }
            ...
        }
        ...
    }
    ...
}
```

本例中，似乎该方法中的许多工作都是围绕一个XDOMNSnippet对象来进行的。这意味着我们可以通过传递不同的值给该方法来编写想要的测试。而实际上在背后发生了许多无关的事情，这些事情只可以通过非常间接的方式来感知。比如说本例，我们可以引入感知变量来辅助我们的工作；可以引入一个实例变量来发现当一个变量具有恰当的结点类型时即会被添加进paraList。

```
public class DOMBuilder                                          298
{
    public boolean nodeAdded = false;
    ...
    void processNode(XDOMNSnippet root, List childNodes)
    {
        if (root != null) {
            if (childNodes != null)
                root.addNode(new XDOMNSnippet(childNodes));
            root.addChild(XDOMNSnippet.NullSnippet);
        }
```

```
            List paraList = new ArrayList();
            XDOMNSnippet snippet = new XDOMNReSnippet();
            snippet.setSource(m_state);
            for (Iterator it = childNodes.iterator();
                    it.hasNext(); ) {
                XDOMNNode node = (XDOMNNode)it.next();
                if (node.type() == TF_G || node.type() == TF_H ||
                        (node.type() == TF_GLOT && node.isChild())) {
                    paraList.add(node);
                    nodeAdded = true;
                }
                ...
            }
            ...
        }
        ...
    }
```

有了这个感知变量，我们仍需要设计一个能够产生这一条件的输入。然后便可以将这块逻辑提取出来，同时我们的测试仍能通过。

在下面这个测试中，我们添加一个类型为TF_G的节点。

```
void testAddNodeOnBasicChild()
{
    DOMBuilder builder = new DomBuilder();
    List children = new ArrayList();
    children.add(new XDOMNNode(XDOMNNode.TF_G));
    Builder.processNode(new XDOMNSnippet(), children);

    assertTrue(builder.nodeAdded);
}
```

下面这个测试旨在验证当一个节点的类型不符时是不会被添加的：

```
void testNoAddNodeOnNonBasicChild()
{
    DOMBuilder builder = new DomBuilder();
    List children = new ArrayList();
    children.add(new XDOMNNode(XDOMNNode.TF_A));
299 Builder.processNode(new XDOMNSnippet(), children);

    assertTrue(!builder.nodeAdded);
}
```

有了这些测试的保护，我们在提取processNode()中决定一个节点是否被添加的条件的时候就放心一些了。我们将整个条件表达式复制出来，测试表明当条件满足的时候节点被添加了。

```
public class DOMBuilder
{
    void processNode(XDOMNSnippet root, List childNodes)
    {
        if (root != null) {
            if (childNodes != null)
                root.addNode(new XDOMNSnippet(childNodes));
```

```
            root.addChild(XDOMNSnippet.NullSnippet);
        }
        List paraList = new ArrayList();
        XDOMNSnippet snippet = new XDOMNReSnippet();
        snippet.setSource(m_state);
        for (Iterator it = childNodes.iterator();
                it.hasNext();) {
            XDOMNNode node = (XDOMNNode)it.next();
            if (isBasicChild(node)) {
                paraList.addNode(node);
                nodeAdded = true;
            }
            ...
        }
        ...
    }
    private boolean isBasicChild(XDOMNNode node) {
        return node.type() == TF_G
            || node.type() == TF_H
            || node.type() == TF_GLOT && node.isChild());
    }
    ...
}
```

以后不需要这个感知变量时，我们便可以将它删除。

本例中我们使用的是一个布尔变量。使用该变量的目的是为了确定在提取条件表达式之后节点是否仍被添加了。我相当确信自己能够毫不出错地将整个条件表达式提取出来，因此没有对其内部的各个分式进行测试。以上测试简单地确保了在代码提取之后我们所关心的条件表达式仍然是代码路径的一部分。关于在方法提取时到底要做多少测试，可以参见第13章提到的目标测试（157页）。 300

使用感知变量时最好将变量放在被重构的类中，直到一系列的重构完成之后再将它们删除。我常常是这么做的，因为这样做有一个好处，就是能够看到自己为了一系列提取所写的所有测试，万一想要更换提取方式的话就可以轻松地撤销这些测试。完成提取之后，我往往会删除这些测试，或者将它们重构一下用来测试提取出来的新方法。

感知变量是分解巨型方法的利器。你可以用它们来对锯齿状方法中内嵌层次很深的代码进行重构，但你同样也可以用它们来逐步将一个锯齿状方法“反锯齿”。例如，如果一个方法内的大部分代码都被深深地嵌在了一组条件语句内的话，便可以利用感知变量来提取其中的条件语句或块。而对于提取出来的新方法，同样也可以使用感知变量。直到整个代码被成功“反锯齿”。

22.3.2　只提取你所了解的

另一个可以用来对付巨型方法的策略就是一开始迈小步，寻找那些我们可以不用测试也能放心提取出来的小块代码，然后添加测试来覆盖它们。但是，等等，我得换个方式来表达，因为每个人对“小”的定义是不一样的。当我说“小块代码”时，我指的是两到三行，最多五行的代码，一块你能够容易地给它想出名字的代码。在进行这些小步提取时需要关心的关键的一个因素是耦合数。耦合数就是指传进传出你所提取的方法的值的总数。例如，如果我们从下面这段代码中提

取出一个max方法的话，它的耦合数就会是3。

```
void process(int a, int b, int c) {
    int maximum;
    if (a > b)
        maximum = a;
    else
        maximum = b;
    ...
}
```

提取后的代码像这样：

```
void process(int a, int b, int c) {
    int maximum = max(a,b);
    ...
}
```

该方法的耦合数就是3：两个传进的值（参数），一个传出的值（返回值）。一般来说最好提
301 取那些耦合数小的方法，因为这样犯错的概率较小。因此，当你在选择提取哪些代码时，可以寻找一段行数较小的代码，然后数一数进入这块代码以及从这块代码出去的变量一共有多少。对成员变量的访问不算，因为我们只是简单地将这块代码剪切复制出来而已；因此成员变量并不“穿过”我们所提取出来的方法的接口。

方法提取过程中的一个主要危险是类型转换错误。因而如果我们只提取那些低耦合数的方法，就能够更好地避免这类情况。在确定了一个可能的提取之后，应该回头看看那些传进这块代码的变量是在哪儿定义的，这么做是为了避免弄错方法的签名。

如果说低耦合数的提取是安全的，那么是不是就意味着0耦合数的提取是最安全的呢？是的。实际上，通过将一个巨型方法中的那些不接受任何参数也不返回任何值的代码块提取成方法，我们就可以获得许多活动空间。这类方法实际上就是所谓的“命令式”方法——比如你命令一个对象对它的状态做某些事情，或者更恶劣地对全局状态做某些事情。不管怎样，对于这类代码，当你试图给它们起名字时，通常能够对该块代码获得更深刻的认识，比如关于它是做什么的，它会怎样影响特定的对象等。而这种认识又进而能导致更多的认识，最终你将能够换一种更有效率的视角来看待你的设计。

当使用只提取你所了解的（Extract What You Know）手法时，记住别去选太大的代码块，如果耦合数大于0，那么通常使用一个感知变量是有好处的。提取完之后，别忘了给你的新方法写几个测试。

然而，当把这一技术用在小块小块的代码上时，你可能会觉得这对整个庞然大物的方法来说有点杯水车薪的感觉。然而，实际上正如俗语所云：积跬步以至千里。每次当你回过头去提取出又一小块代码时，就不知不觉间又迈进了一步，同时你的方法也更清晰了一分。渐渐地，你会发现对该方法有了更好的认识，同时也更清楚如何修改它了。

当手头没有重构工具时，我通常一开始会提取0耦合数的方法，这一步只是为了能够对代码的整体结构有一个认识。对后面进行测试以及其他工作是一个很好的准备。

如果你有一个项目列表式方法，那么可能会觉得你将能够提取出许多0耦合数的方法，而且每个都不错。是的，有些代码块的确如你所想，但通常许多代码块都会用到前面声明了的局部变量。所以，有时候你必须得抛开所看到的代码块结构，而是从块内或者块间去寻找低耦合数的方法。 302

22.3.3　依赖收集

有时候一个巨型方法里面会出现一些看起来跟该方法的主要意图不怎么有联系的代码。这些方法也许是必要的，但并不十分复杂，而且如果你不小心破坏了它，很明显就能看出来。但尽管这些都是实话，你仍然还是没法冒这个险。那么在这类情况下该怎么办呢？你可以使用一种叫做依赖收集（gleaning dependencies）的手法。首先你编写测试来保护你要保护的逻辑。然后，你提取出你的测试所没有覆盖到的部分。这么一来，你至少可以确信你保护住了重要的行为。下面就是一个简单的例子：

```
void addEntry(Entry entry) {
    if (view != null && DISPLAY == true) {
        view.show(entry);
    }
    ...
    if (entry.category().equals("single")
                || entry.category("dual")) {
        entries.add(entry);
        view.showUpdate(entry, view.GREEN);
    }
    else {
        ...
    }
}
```

如果我们把负责显示的代码搞糟了，则很快就能看到后果。然而，在添加entry的代码部分，如果引入了一个错误的话，就不是那么容易能找出来的了。在像这样的例子中，我们可以编写测试来确保entry的添加总是在正确前提下发生的。然而，一旦确信这些重要行为已经被保护起来了，我们便可以将负责显示部分的代码提取出来，并同时确信我们的提取不会影响到entry的添加。

某种意义上，这一手法就像鸵鸟战术。你保护了一组行为，而同时在无保护的情况下修改其他代码。但在一个应用当中，并非所有的行为都是平等的。有些行为更重要，我们在修改代码时能够将它们识别出来。

依赖收集是一种强大的策略，尤其是当重要行为与其他行为纠缠在一起的时候。一旦对重要行为建立起了坚固的测试，便可以做许多编辑修改，虽然从技术上讲这些修改并没有全被测试覆盖，但测试至少护住了那些关键的行为。 303

22.3.4　分解出方法对象

感知变量是非常强大的工具，但有时候我们会注意到，方法里面本就已经有了可以直接被用作感知变量的变量了，只不过它们也许是局部变量。要是成员变量的话，我们便可以在一个方法被调用之后通过它们来进行感知了。而实际上，我们的确可以将一个局部变量变成成员变量，只不过许多情况下这么做可能会带来一些混乱——所提取出来的成员变量只对你的巨型方

法以及从该方法中提取出来的方法有意义。尽管每次你的巨型方法被调用起来的时候该变量都会被重新初始化，但如果想要单独调用所提取出来的那些方法的话，就难以弄清这些变量到底持有什么值了。

一个替代方案是使用*分解出方法对象*（Break Out Method Object）手法。该手法是由Ward Cunningham首先引入的，它是一种典型的人为抽象。当你分解出一个方法对象时，实际上就是创建了一个类，其唯一职责是做原来的巨型方法所做的工作。原巨型方法的参数变成了该类的构造函数的参数，原巨型方法中的代码则可以放到该类中的一个名叫`run()`或`execute()`的方法中。一旦代码被移到了新类中，重构起来就方便多了。我们可以将方法中的局部变量做成该类的成员变量，并让它们充当我们的感知变量。

分解出方法对象是相当激烈的改动，但与*引入感知变量*（239页）手法不同的是，前者的“感知变量”同样也是产品代码要用到的变量。这就意味着你写出来的测试会一直可用。具体例子可参见后面专门介绍这一手法的章节。

22.4　策略

本章介绍的技术能够帮助你将巨型方法分解，从而利于后面的重构或特性添加。本节介绍的一些原则会帮助你在做这项工作的时候在代码结构上作出权衡。

22.4.1　主干提取

如果摆在你面前的是一个条件语句，而你的任务是找出哪儿可以提取出一个方法来，那么你有两个选择：一是将条件和分支一同提取出来，二是分别提取。例如：

```
if (marginalRate() > 2 && order.hasLimit()) {
    order.readjust(rateCalculator.rateForToday());
    order.recalculate();
}
```

304

如果你将代码中的条件和分支体分别提取到两个不同的方法中，那么后面想重组代码的逻辑就会容易一些。

```
if (orderNeedsRecalculation(order)) {
    recalculateOrder(order, rateCalculator);
}
```

我将这一手法叫做*主干提取*（Skeletonize），因为你实质上是将代码的主干提取出来：即控制结构以及对其他方法的委托（调用）。

22.4.2　序列发现

假设你手里有一个条件语句，而你想要找到提取方法的地点。那么你有两个选择：一，将条件和分支体一同提取出来；二，分开来提取。下面是另一个例子：

```
...
if (marginalRate() > 2 && order.hasLimit()) {
```

```
    order.readjust(rateCalculator.rateForToday());
    order.recalculate();
}
...
```

如果你将条件和分支一同提取出来，好处就是容易从代码中识别出一个操作序列：

```
    ...
    recalculateOrder(order, rateCalculator);
    ...


void recalculateOrder(Order order,
                      RateCalculator rateCalculator) {
    if (marginalRate() > 2 && order.hasLimit()) {
        order.readjust(rateCalculator.rateForToday());
        order.recalculate();
    }
}
```

之所以这么说，是因为在这个长方法内的其他（除我们提取出来的这部分之外）代码可能只是一系列的操作，一个接一个；于是只要将这块条件代码也提取为单一的操作（方法调用），就能够对整个方法有一个更清晰的认识。

等一下，我是不是自相矛盾了。没错。实际上，我常常在主干提取跟序列发现这两者之间来来回回。而且我打赌你也会。当我觉得某个控制结构在被澄清之后还需要被重构的话，就会采用主干提取。而另一方面，当我觉得呈现出代码中的序列结构能让代码变得更清晰的话，就会使用序列发现。 305

面对项目列表式方法，我往往会使用序列发现，而锯齿状方法则是主干提取。然而到底选用哪种策略其实还是取决于你在提取的时候对设计的洞察。

22.4.3　优先提取到当前类中

在从一个巨型方法中提取代码时，你可能会注意到其中有些代码块其实是应该属于其他类的。对此一个很强的暗示就是你想给新方法起的名字。比如说你看到一块将要提取出来的代码，然后很想用它用到的某个变量的名字来给这块代码命名，那么很可能这就意味着你提取出来的代码应属于那个变量的类。比如下面这段代码：

```
if (marginalRate() > 2 && order.hasLimit()) {
    order.readjust(rateCalculator.rateForToday());
    order.recalculate();
}
```

看起来我们可以将这块代码叫做`recalculateOrder`。这是个不错的名字，但如果我们在名字中用到了“order”这个单词，那么或许这块代码应当转移到`Order`类当中去，并起名叫做`recalculate`。`Order`已经有了一个名叫`recalculate`的方法，所以我们或许应该想想现在的这个`recalculate`跟`Order`上原有的那个到底有何区别，并将这一信息用在方法名上；或者我们也可以重命名原来的那个`recalculate`。但不管怎样，看起来这块代码确实应该属于`Order`类。

尽管将代码直接提取到另一个类中听起来很诱人，但实际上，别这么做。笨拙的名字可以先用着。比如recalculateOrder这个名字，笨重是笨重，但它让我们得以进行一些能够轻易撤销的提取，从而试探我们的提取是否正确，是否能够继续往下提取。之后，当好的修改自然而然浮现出来的时候，我们再将这些方法转移到别的类中去也不迟。而就目前来说，提取到当前类中能够确保我们继续手头的工作，而且更不容易出错。

22.4.4　小块提取

我曾在上文提到这一手法，但这里我想再强调一遍：优先提取小块代码。虽说对于庞然大物般的方法来说，提取一小块代码看起来是蝼蚁撼树，但假以时间，随着小块小块的代码不断被提
306 取出来，你便会发现自己对该方法有了新的认识。比如，你可能会发现一个操作序列从代码结构中清晰地浮现出来（原本被埋在一堆分支结构中），比如，你可能会发现一个更好的组织该方法的方式。于是你可以朝着你所看到的方向前进。这样一种渐进式的方法比起一开始就想将巨型方法大卸八块的做法要好得多。后者常常并不像它看上去那么容易；而且也不安全，一不小心就会忘记一些细节，而细节却是代码中必不可少的成分。

22.4.5　时刻准备重新提取

切一块蛋糕有很多种切法，同样，分解一个巨型方法也有很多办法。往往在做了一些提取之后，你会发现还有能更容易地适应新特性的提取方式。对于这种情况，最佳做法有时便是撤销一两个提取，并重新提取。这么做并不是说前面的提取都白做了，事实上它们带给了你一些非常重
307 要的认识：对原有设计以及更佳的改进方式的认识。

第23章 降低修改的风险

代码是一种奇怪的建筑材料。大多数能够用来做出东西的材料都会磨损或疲劳，如金属、木材、塑料等，用久了便会坏掉。然而代码不同。你把一块代码丢在那儿，它怎么也不会坏，除非有人去修改它（或者你的硬盘让宇宙射线给破坏了）。如果你使用一台机器，那么用久了总会坏的。但一次一次地运行一段代码却完全不会破坏它。

代码的这种性质给我们开发者带来了很大的负担。因为我们不仅是往软件（代码）里面引入错误的人，而且实际上一不小心就会这么做。那么，修改代码的难易程度又如何呢？如果是指机械地修改的话，答案是相当容易。每个人都可以打开编辑器，然后敲上一堆奇形怪状的代码。敲首诗进去或许都能编译（见www.ioccc.org上的模糊C代码大赛作品）。话说回来，在写代码时捅出娄子实在是太容易了。你有没有过这样的经历：在费尽周折抓住了一个bug之后，却发现只是因为当时不小心敲错了一个键而导致的。比如在把书递给同事的时候书皮掉下来砸到了键盘。代码真是相当脆弱的东西。

本章将会讨论一些帮助我们降低编码过程中的危险的方法。它们有些只是一些机械的方法，有些则是心理方法，但关注它们是很重要的，尤其是当我们在遗留代码中解依赖时。

308 ~ 309

23.1 超感编辑

你在编辑代码的时候都做些什么？我们的目标是什么？通常会有一个大的目标。我们想要添加一个特性或修正一个bug。知道目标是什么当然是件好事，但如何付诸行动呢？

这么想，我们现在坐在键盘前，每敲一次键盘会有两种可能，一种可能是我们的动作改变了软件（代码）的行为，另一种可能则是不改变。比如，往一段注释里面添加文字？不会改变行为。往一段字符串里面添加文字？大多数时候会。除非该字符串位于一段不被调用到的代码中。但如果之后我们又添加代码完成对包含该字符串的方法的调用呢，那是会改变行为的。所以严格来说，就算只是敲敲空格键对代码做点格式化也算是某种意义上的重构。有时敲代码也是重构。不过，修改一个表达式里面的数值不是重构，而是功能改变，分清这一点很重要。

这便是编程的有趣之处了，精确地了解我们的每一次敲击会带来什么影响。当然，这并非意味着我们必须是万能的，而是说任何能够帮助我们了解（真正了解）我们敲入的字符会如何影响系统行为的方法都能够帮助我们减少bug。从这个意义上说，测试驱动开发（74页）是一门强大

的技术。只要你能将代码塞进测试用具并在一秒内运行完其测试，你就可以在任何必要的时候只花上极短时间就了解到你的修改给系统带来了什么影响。

> 我相信不久（就算本书出版时还没有，我相信也不会太远）便会有人开发出这样一个IDE：它能够允许你指定一组测试在每次按键时都运行。这样一来反馈周期就几近于瞬时了。
>
> 我相信这样的IDE迟早会有人开发出来的。因为这一需求看上去是如此的不可避免。目前已经有IDE能够在每次敲击时进行语法检查，并在发现代码中的错误时用加亮或下划线之类的手段来提醒程序员。所以，很自然的，“编辑时测试”就是下一步了。

和结对编程一样，测试也能够带来所谓的“超感”编程。但“超感”编程听起来是不是挺费神的？[1]没错，但任何事情过度了都是不行的。关键的一点是，这种“超感”编程并不令人沮丧，它是一种流动状态，在这种状态下你能够隔绝外界一切影响，进入代码的世界，时刻感知它。实际上，“超感”编程是非常“提神”的。我个人的感觉是，如果我在编程的时候得不到任何反馈，我就会感到非常疲劳；总是害怕自己是不是不小心犯下了什么破坏代码的错误。我需要在脑子里记录和维护所有的状态，记住修改了什么和没有修改什么，并想着待会怎么才能说服自己所作的
310 改动的确是当初计划的那些。

23.2　单一目标的编辑

我不清楚每个人对计算机行业的第一印象是否都一样，但就我个人来说，我第一次想要当一个程序员时，实在是被那些超级聪明的程序员的故事所迷住了。那些家伙能够将整个系统的状态放在脑子里，在谈笑间就能写出漂亮的代码，并立即知晓某些修改是正确的还是错误的。我承认，并非每个人都能像他们那样在脑子里记住那么多古怪的细节。我个人也只是在一定程度上能做到。以前我曾经掌握了C++语言里的许多晦涩的部分，而且有一阵子我脑子里还记了许多关于UML元模型的细节。直到有一天我发觉，作为一个程序员，记得关于UML的那么多细节其实根本没用，而且简直有点可悲。

事实上，聪明也分很多种。在脑子里记住很多东西的这种聪明有时候是很有用的，但它并不能帮助我们作出更好的决策。比如我自己吧，虽然现在的我对于所用语言的细节的掌握比以前要少，但我觉得作为程序员我比以前要强了。判断力是一项关键的编程技能，如果我们非要试图表现得像那些超级聪明的程序员那样的话，结果只会给自己带来麻烦。

以下场景曾经在你身上出现过吗：你在写代码，写着写着突然意识到，“嗯……或许应该把这块代码清理一下。”于是停下来开始重构，然而，你不由开始设想这块代码实际应该是什么样子的，然后你就停住了。无论如何你正在做的这个特性还是要完成啊，所以你就回到刚才你编辑代码的地方。你认为需要调用一个方法，于是跳转到那个方法的所在地，却发现你需要该方法做一些其他事情；于是你又开始修改这个方法，把刚才的修改晾在那儿，这时你旁边的编程伙伴开

1. 因为要时刻保持感知嘛。——译者注

始冲你叫了“嘿！老兄！先把刚才的工作做完再来改这个吧。”于是你感觉自己像个拉磨的骡子一样，而旁边的家伙偏偏还来添乱。

以上的确就是某些团队的现状。比如两个人结对编程，有了一段有趣的编程体验，但其中有四分之三的时间花在了修正前四分之一时间内破坏的代码上了。听起来很可怕是吗？没错，但有时候这也是挺有趣的，你和你的伙伴得以从容不迫地枪下逃生。你们遇到了代码中的魔鬼并将其杀死。你们是胜利者。

问题是，是否值得？让我们来看一看另一种方式。

你需要修改一个方法，并且已经将你的类弄进测试用具了，于是开始修改。但你不由开始想了：“我还需要修改一下那边一个方法”。于是你停下手头的修改，跳转到那个方法去了，后者看 311
起来一团糟，所以你开始对它进行一点格式化，以便弄清楚它到底干了些什么。这时你的结对编程伙伴发话了：“你在干什么？”你回答道：“哦，只是看看是否需要修改方法X。”于是他说：“别，还是同一时间做一件事吧。”他拿出一张纸写下方法X的名字，放在电脑旁边，于是你回到原来的代码继续未完的修改。完成之后你运行了一遍测试，发现全部通过。于是你再次跳到那个方法，毫无疑问，你得对它作一些修改。首先你开始编写另一个测试。编了一会程序之后，你运行编好的测试，然后开始做集成。这时你和你的伙伴注意到桌子对面的另外两个程序员。其中一个正在对着另一个喊：“嘿！老兄！先把刚才的工作做完再来改这个吧。”他们已经在那个任务上耗了好几个小时了，看起来筋疲力尽。如果时间可以倒流的话，他们会选择集成，并节省好几个小时。

我在工作的时候时常用一句话来提醒自己：“编程是关于同一时间只做一件事的艺术。”如果我是在和另一个人结对编程，那么我会让我的伙伴监督我，在适当的时候提醒我：“你在干嘛？”如果我的答案包含了两件事情，那么我们就会在其中选出一件。同样，对我的伙伴我也会这么做。坦白的说，这种编程方式快多了。编程的时候一不小心就会掉入“贪心不足蛇吞象”的局面，结果是不仅受到打击，而且落到只能通过尝试来让代码工作的境地，而不是胸有成竹。

23.3　签名保持

在编辑代码的时候有众多原因可能造成错误。比如打错字、用错数据类型、用错变量……可能性太多了。尤其是重构，重构通常意味着极具侵入性的编辑。我们将代码复制来复制去，并建立新类和新方法；从尺度上说这可比单单添加一两行代码大多了。

一般来说对付这种情形的手段是测试。一旦测试在手，我们便能够捕获在修改代码的过程中引入的许多错误。然而可惜的是，对于许多系统而言，要想让它足够可测试，以便能够对其进一步重构，就必须首先对它作一点重构。这种初始的重构（第25章列出的解依赖技术）注定要在没有测试的情况下完成，而且它们必须得是很保守的重构。

在一开始使用这些技术的时候，我总是忍不住太贪心了。当需要提取某个方法的整个方法体时，除了复制粘贴之外我还做了其他清理工作。例如，假设我要提取一个方法的方法体，并让该 312
方法成为静态的（*暴露静态方法*，273页），如下所示：

```
public void process(List orders,
```

```
                    int dailyTarget,
                    double interestRate,
                    int compensationPercent) {
    ...
    // complicated code here
    ...
}
```

如下进行提取，在过程中还顺带创建了一些辅助类：

```
public void process(List orders,
                    int dailyTarget,
                    double interestRate,
                    int compensationPercent) {
    processOrders(new OrderBatch(orders),
                  new CompensationTarget(dailyTarget,
                    interestRate * 100,
                    compensationPercent));
}
```

动机是好的。我是想在解依赖的过程中顺带改善设计，但事实并不像我想象的那样美好。我在修改的过程中犯了一些愚蠢的错误，而同时又没有任何测试能够帮我捕获它们，于是这些错误常常过了很久才被发现。

在为了代码的可测试性而进行解依赖的过程中，你得格外小心。我的做法之一是尽可能地采用签名保持手法。如果完全避免了签名的改动，就能够将方法的整个签名从一处剪切复制到另一处，并最小化引入错误的风险。

在上面的例子中，我本该这么做：

```
public void process(List orders,
                    int dailyTarget,
                    double interestRate,
                    int compensationPercent) {
        processOrders(orders, dailyTarget, interestRate,
                      compensationPercent);
}

private static void processOrders(List orders,
                      int dailyTarget,
                      double interestRate,
                      int compensationPercent) {
    ...
}
```

313

这样一来，只需花很少的精力就可以搞定新方法的参数。从根本上，只需如下几个步骤：

(1) 将整个参数列表复制到剪切板中：

```
List orders,
int dailyTarget,
double interestRate,
int compensationPercent
```

(2) 声明新方法：

```
private void processOrders() {
}
```

(3) 将刚才复制的参数列表粘贴到新方法这里：

```
private void processOrders(List orders,
                           int dailyTarget,
                           double interestRate,
                           int compensationPercent) {
}
```

(4) 调用新方法：

```
processOrders();
```

(5) 仍然将复制的参数列表粘贴过来：

```
processOrders(List orders,
              int dailyTarget,
              double interestRate,
              int compensationPercent);
```

(6) 删除变量的类型：

```
processOrders(orders,
              dailyTarget,
              interestRate,
              compensationPercent);
```

一旦熟练了之后，这个过程就变得机械化，你也就对自己的修改越来越有信心了，从而在解依赖时能够把精力集中在那些可能导致错误的顽固问题上。例如你的新方法是否隐藏住了基类中某个同名方法，等等。

关于*签名保持*，还有另外一些场景。如，你可以利用该技术来声明新方法，也可以用它来建立一组成员变量（具体细节见*分解出方法对象*手法）。 314

23.4　依靠编译器

编译器的主要目的是将源代码转换成另外一种形式，不过在静态类型的语言中，编译器还能担当更多的职责。你可以利用它的类型检查机制来找出需要修改的地方。我把这种做法叫做*依靠编译器*（Lean on the Compiler）。以下是一个例子：

假设在一个C++程序中，我们有一些全局变量：

```
double domestic_exchange_rate;
double foreign_exchange_rate;
```

与它们位于同一文件中的还有一些函数，后者使用了这些全局变量。我想将这些函数纳入测试，于是使用了目录中列出的*封装全局引用*（268页）技术。

为此我编写了一个类来包住这两个变量，并声明了该类的一个对象：

```
class Exchange
{
```

```
public:
    double domestic_exchange_rate;
    double foreign_exchange_rate;
};

Exchange exchange;
```

然后编译代码，让编译器帮我找出所有引用了domestic_exchange_rate和foreign_exchange_rate的地方，然后将这些地方统统改为通过exchange对象来访问。下面是更改前后的代码对比：

```
total = domestic_exchange_rate * instrument_shares;
```

变成：

```
total = exchange.domestic_exchange_rate * instrument_shares;
```

该技术的最关键的一点就是让编译器帮助你找到需要修改的地点。注意，这并不代表你就不需要思考该修改什么；而只是说在某些情况下可以让编译器帮你做搜集信息的工作。非常重要的
315 一点是要弄清什么是编译器能够帮你找到的，什么是它所不能的，这样才不至于陷入盲目的自信。

依靠编译器手法包含两步：

(1) 修改一处声明从而引发编译错误；

(2) 转到编译出错的地点，修改。

在对代码结构进行修改时，可以采用依靠编译器手法，就像我们在*封装全局引用*例子中做的那样。此外你还可以用它来发起类型修改。一个常见情形是将某个变量的类型从一个类改为一个接口，并利用编译错误来发现哪些方法需要放在该接口上。

不过，这一手法也并非总是可行的。如果你的项目构建耗时很长，那么更实际的做法往往是自己搜索那些需要修改的地方。第7章介绍了对付这一问题的方法。不过话说回来，能依靠编译器还是得依靠编译器，这是个有用的手法。只是要注意，盲目采用这一手法可能会引入一些微小的错误。

比如当涉及到继承时，就得小心依靠编译器了，继承在这种场合下最容易带来问题，下面就是一个例子。

假设我们有一个叫做getX()的实例方法，它位于一个Java类中。

```
public int getX() {
    return x;
}
```

我们想要找到所有调用它的地方。于是先将该方法注释掉。

```
/*
public int getX() {
    return x;
} */
```

然后重新编译。

猜怎么着？什么编译错误也没有。那这是不是就意味着getX()根本没有被任何地方调用

呢？不一定。如果同样有一个getX()被声明在了基类中，那么将现在这个（派生）类中的getX()注释掉只会让基类的那个暴露出来而已。成员变量也存在这个问题。

依靠编译器是一门强大的技术，但你得了解它的局限性在哪里；不了解的话就可能会遇到一些严重的问题。

结对编程

很可能你已经听说过*结对编程*（Pair Programming）这一概念了。如果在开发过程中运用了极限编程（XP）的话，你很可能已经这么做了。很好，因为结对编程对于提高质量以及在团队中传递知识都是很有好处的。 316

如果你现在还未使用结对编程，我建议你试试。尤其建议当你在使用本书中描述的解依赖技术时进行结对编程。

我们在编辑代码时很容易犯错误，并且自己还根本不知道已经破坏了代码。而多一双眼睛看着当然是有好处的。现实是，对付遗留代码就好比是动手术，而医生是从来不会一个人做手术的。

更多关于结对编程的介绍可参考Laurie Williams和Robert Kessler的*Pair Programming Illuminated*（Addison-Wesley 2002），并访问www.pairprogramming.com。 317

第24章 当你感到绝望时

对付遗留代码是件苦差事，这一点没什么好否认的。尽管各自情况不同，但有衡量工作价值的方式是相同的（不管你是不是程序员）：算算你能从中得到什么。对某些人来说也许是薪水——这没什么不好意思的，毕竟人人都得生活。但我认为肯定还有其他什么原因让你成为一个程序员的。

有些幸运的读者也许是因为兴趣才加入这行业的。你怀着对计算机的迷恋开始编程，前面的路充满无尽的可能，你可以通过编程实现各种各样很酷的东西。你学习并掌握很多知识，于是你开始想："这似乎挺有意思的。如果我能够在这上面做得很好，或许我能把它当成一职业呢！"

当然，并非每个人都是走的这条路，但就算不是这样的，也多多少少跟编程的乐趣有点关系。如果你能够体会，并且你的同事们也能体会这种乐趣，那么所面对的是什么系统有什么类系呢？即使是遗留系统也能在其中做出漂亮的东西出来。否则难道你想整天垂头丧气不成？那可没啥意思。我们都不应该那样。

对付遗留系统的人们常常希望他们能去做全新的系统。从头开始构建一个系统固然有意思，但坦白地说，全新的系统也有它们自己的问题。比如，我就不止一次看到如下场景：一个既有系统，随着时间的推移，逐渐变得混乱，难以改动。每次修改都得花上很大的精力和时间，于是人们感到沮丧不堪。于是他们将队伍里面最好的程序员（有时候这些家伙才是问题的罪魁祸首！）放到一个新团队中，并交给他们任务：创建一个替代系统，要有更好的架构。一开始事情都还算顺利。他们知道旧的架构问题在哪里，于是花了一些时间做出一个新的设计。在他们做这些事情的同时，其他开发者仍然奋战在老系统上。由于老系统仍在运行，因此他们会收到一些修正bug的请求，间或还会需要添加新特性。业务人员冷静地审视这些新特性的请求，并决定是否要将它们加入老系统中，或者客户能否等到新系统出来（提供这一特性）。由于许多时候客户都是等不及的，所以不仅新系统要添加这一特性，老系统同样也要。这就意味着新系统开发团队面对的是双重职责：他们要替换一个仍处于不断变化中的旧系统。几个月过去了，事实越来越明显：他们没法替换你正在维护的这个旧系统。压力还在增大。他们夜以继日地工作，甚至牺牲周末。然而许多时候结果却是，公司发觉你在旧系统上做的工作才是最重要的，关于公司命脉和未来存亡。

318
~
319

看来新系统开发也不是那么好干的差事，不是么？

要想在对付遗留代码时保持积极向上的心态，关键是要找到动力。尽管有很多程序员的生活圈子都较小，但工作时有一个良好的环境，有你所尊敬的并且知道如何快乐工作的人做同事仍然

是一个非常大的诱惑。比如，我的几个最好的朋友就是在工作中认识的。至今，当我在编程中遇到一些有趣的问题或学到一些东西时还会跟他们交流。

另一个有用的做法就是跟社区中的广大程序员接触。在如今的网络时代，要想跟其他程序员接触并从他们那儿学习或分享知识真是前所未有的容易。你可以订阅邮件列表、参加讨论会、利用网络上可获取的一切资源、分享经验和技术，等等，并让自己时刻处在软件开发的最前沿。

但就算项目组里面的所有成员都关心项目且希望能把事情做好，仍还是会有沮丧的情况发生。比如有时候人们会觉得面对的代码基太大了，以致于他们觉得就算在上面做上十年也改进不了百分之十。但这样就有理由沮丧并放弃了吗？不。我就曾见过有的团队面对百万行遗留代码仍然面不改色，每天迎接挑战并将其作为改进系统的契机乃至从中获得乐趣。我同样也见过一些团队，有着相对好得多的代码基却仍然士气低迷。所以说态度很重要。

工作之余，自己练一练测试驱动开发，让自己仅仅为了兴趣而编一点程序。感受一下你自己做的小项目跟实际工作中的大项目之间的差异。于是很可能，当你在工作中将项目置入快速测试用具之下时，就会有与自己做小项目时同样的感觉了。

如果你的团队士气低迷，而且是由于代码质量太低而低迷的，那么有个办法可以一试——从项目中找出一组最糟糕的类，将它们置入测试之下。一旦你们作为一个团队共同克服了一个最困 320
难的问题，那么就会感觉一切都不过如此了。这种情况我经历得太多了。

一旦掌控住了代码基，你们便会开始在里面不断制造出一块块良好的代码绿洲。在其中工作 321
便成了快乐的体验。

Part 3

第三部分

解依赖技术

本部分内容

第25章 解依赖技术

本章介绍了一系列的解依赖技术。当然这个列表并不全面，它包含了我与不同团队共事的过程中，用于将类解依赖以使它们能被置于测试之下的技术。从技术上讲，这些是重构技术，因为它们全都保持了代码的行为。但它们与业界目前所给出的那些重构技术不同，这些技术是要在没有测试的情况下使用的，其目的是为了将测试安置到位。在大多数情况下，如果你小心按步骤行事，那么出错的可能性是很小的。但这并不就意味着这些技术是完全安全的；错误仍可能发生，因此你应该小心行事。在运用本章描述的重构技术之前，请先阅读第23章。该章所介绍的一些技巧有助于你安全地运用这儿所讲的技术，从而将测试安置到位；而一旦测试到位了，你也就可以更放心地去做一些更为侵入性的改动了。

当然，这些技术并不能立竿见影地让设计变得更好。实际上，如果你有良好的设计感，这里的某些技术甚至可能会令你退缩。但它们可以助你将方法、类以及类簇纳入测试，从而让你的系统更具可维护性。一旦代码被置入测试的保护之下，便可以使用测试支持的重构来改善你的设计了。

322～325

本章提到的一些重构技术在Martin Fowler的著作《重构：改善既有代码的设计》（Addison-Wesley, 1999）中也有描述。只是本章对它们的描述在步骤上有所不同，我将之适当剪裁以使得它们能够被安全地用在没有测试的情况下。

25.1 参数适配

在对方法作改动时常常会遇到一些令人头疼的依赖，这些依赖由方法的参数导致。比如有时会发现难以创建所需的参数，又比如需要测试某方法对其某个参数的影响。许多时候我发现参数的类型正是带来麻烦的根源。如果该类是我可以修改的，则可以使用接口提取来解开参数对该类的依赖。在参数依赖问题上，接口提取（285页）往往是不二之选。

一般来说，我们都想通过一些简单的方法来解开那些妨碍我们安置测试的依赖，方法最好简单到用的时候不会出错。而从这个意义上说接口提取有时并不是那么好的选择。比如参数类型的抽象层次较低，或特定于某些实现技术的话，提取其接口就有可能达不到预期效果甚至根本就不可行。

当无法对一个参数的类型使用接口提取，或者当该参数难以“伪装”的时候，可采用参数适配手法。

下面是一个具体的例子：

```
public class ARMDispatcher
{
    public void populate(HttpServletRequest request) {
        String [] values
            = request.getParameterValues(pageStateName);
        if (values != null && values.length  > 0)
        {
            marketBindings.put(pageStateName + getDateStamp(),
                               values[0]);
        }
        ...
    }
    ...
}
```

在该类中，populate方法接受一个HttpServletRequest参数。HttpServletRequest是J2EE中的接口。就populate现在的这个样子，要想测试它的话就必须创建一个实现了HttpServletRequest的类，而且该类须提供某种方法让HttpServletRequest能够在测试时返回所需的参数值。目前的Java SDK文档显示HttpServletRequest上声明了大约23个方法，而且还不算派生自更高级接口内的方法，后者我们也需实现。要是能用接口提取手法提取出一个窄一些的、只含我们所需方法的接口就好了，只可惜我们无法从一个接口提取出另一个接口。那样便需要HttpServletRequest扩展我们提取出的那个接口，然而我们又无法像这样修改一个 326
标准接口。但所幸的是我们还有另一个选择。

有一些针对J2EE的仿对象（mock object）库。若下载一个，我们便可以为HttpServletRequest创建一个仿对象并实现我们的测试。这么做可以节省不少时间；利用这种做法，我们无需花时间手动“伪造”一个servlet request类。那么，这是不是说我们终于有了解决方案了呢？未必。

在解依赖时我总是会尽量往前看，推测结果会如何。然后确定自己能否忍受该结果。本例中我们的产品代码基本也是一样，我们将要花上许多时间才能将HttpServletRequest这个API接口保留在原位。但有没有办法既能令代码看起来更好，又能令解依赖更简单呢？实际上是有的。我们可以将参数类型外覆起来，从而完全解除对API接口的依赖。这么做之后的代码如下：

```
public class ARMDispatcher
    public void populate(ParameterSource source) {
        String values = source.getParameterForName(pageStateName);
        if (value != null) {
            marketBindings.put(pageStateName + getDateStamp(),
                               value);
        }
        ...
```

```
    }
}
```

我们对代码做了什么？嗯……我们引入了一个新的接口——ParameterSource。目前它上面只有一个getParameterForName方法。和HttpServletRequest的getParmeterValue不一样，getParameterForName只返回一个串。之所以这样编写该方法是因为在当前的上下文中我们只关心第一个参数。

> 接口应传达职责而非实现细节。这样的接口令代码易于阅读和维护。

以下是一个实现了ParameterSource的“伪”类。可以将它用在我们的测试中：

```
class FakeParameterSource implements ParameterSource
{
    public String value;

    public String getParameterForName(String name) {
        return value;
    }
}
```

327

而产品代码用的ParameterSource实现则看起来像这样：

```
class ServletParameterSource implements ParameterSource
{
    private HttpServletRequest request;

    public ServletParameterSource(HttpServletRequest request) {
        this.request = request;
    }

    String getParameterValue(String name) {
        String [] values = request.getParameterValues(name);
        if (values == null || values.length < 1)
            return null;
        return values[0];
    }
}
```

从表面上看，这么做似乎仅仅是为了漂亮而漂亮，但遗留代码基中的一个普遍问题就是抽象层次不够；系统中最重要的代码往往跟底层API调用耦合在一起。我们当然已经看到了这会令测试变得多难，但问题还不仅仅是测试。当代码中到处都是动辄包含大量未被使用的方法的宽接口时，代码便会变得难以理解。反之，若你能够创建窄一些并针对特定需求的接口，代码便会更好地传达语义，而且其中也有了更佳的接缝。

如果我们改用ParameterSource，则populate方法的逻辑就跟特定的参数源分离开来了。从而代码便不再依赖于特定的J2EE接口。

> 参数适配是一个违反签名保持（249页）的例子。因此，用的时候请格外小心。

有些时候参数适配手法也可能会带来危险，比如你创建的简化接口与原来参数的接口类型相差太远。如果我们在修改的时候不小心，就可能会引入微小的bug。正如往常一样，记住我们是为了将测试安置到位而解开依赖。所以应该去做那些你更有信心的修改，而不是能导致最佳代码结构的修改。一旦测试到位，一切都好办了，那时你自然会得到最佳代码结构。例如，本例中我们或许想要对ParameterSource作一点改动，从而使它的客户代码不用检查返回null（详见空对象模式，94页）。

安全第一。一旦测试到位，你便可以更有信心地进行侵入性的改动了。 328

步骤

参数适配手法的步骤如下：

(1) 创建将被用于该方法的新接口，该接口越简单且能表达意图越好。但也要注意，该接口不应导致需要对该方法的代码作大规模修改。

(2) 为新接口创建一个用于产品代码的实现。

(3) 为新接口创建一个用于测试的“伪造”实现。

(4) 编写一个简单的测试用例，将伪对象传给该方法。

(5) 对该方法作必要的修改以使其能使用新的参数。

(6) 运行测试来确保你能使用伪对象来测试该方法。 329

25.2　分解出方法对象

在许多应用中，长方法都是非常难对付的角色。但如果你能够实例化包含这种方法的类并将其放入测试用具的话，往往就意味着下一步便可以开始编写测试了。有时候，为了让一个类能够被独立地实例化，需要付出相当大的努力；甚至可能对于你要进行的修改来说显得有点得不偿失了。如果你想要对付的方法规模较小，并且没有使用实例数据，那么可以使用暴露静态方法（273页）手法来将代码置入测试之下。另一方面，倘若你的方法规模较大，或者使用了实例数据或其他方法的话，则可以考虑使用分解出方法对象（Break Out Method Object）手法。简单说来，该手法的核心理念就是将一个长方法移至一个新类中。后者的对象便被称为方法对象，因为它们只含单个方法的代码。通常在运用了该手法之后你也就可以给新类编写测试了，这会比为旧方法编写测试来得更容易一些。旧方法中的局部变量可以做成新类中的成员变量，这通常能令解依赖并改善代码状况变得更容易。

下面是一个C++的例子（为简洁起见，大量的类和方法并没有列出来）：

```
class GDIBrush
{
public:
    void draw(vector<point>& renderingRoots,
              ColorMatrix& colors,
              vector<point>& selection);
```

```
    ...

private:
    void drawPoint(int x, int y, COLOR color);
    ...

};

void GDIBrush::draw(vector<point>& renderingRoots,
                    ColorMatrix& colors,
                    vector<point>& selection)
{
    for(vector<points>::iterator it = renderingRoots.begin();
            it != renderingRoots.end();
            ++it) {
        point p = *it;
        ...

        drawPoint(p.x, p.y, colors[n]);
    }
    ...
}
```

330

GDIBrush的draw方法是一个长方法。我们没法轻易地为其编写测试，而且要想在测试用具中创建GDIBrush的实例也是很难的。因此，让我们来试试使用分解出方法对象技术将其移到一个新类中看看。

第一步就是创建一个负责画图工作的新类。我们可以把它叫做Renderer。创建该类之后，再给它编写一个公有的构造函数。该构造函数的参数包括对原类的引用，以及原（旧）方法的所有参数。对于后者我们可以采用签名保持（249页）手法：

```
class Renderer
{
public:
    Renderer(GBIBrush *brush,
             vector<point>& renderingRoots,
             ColorMatrix &colors,
             vector<point>& selection);
    ...
};
```

创建了构造函数之后，我们便可以为它的每个参数设立一个成员变量，并初始化它们。为了保持签名，这也是通过剪切/复制/粘贴来完成的。

```
class Renderer
{
private:
    GDIBrush *brush;
    vector<point>& renderingRoots;
    ColorMatrix& colors;
    vector<point>& selection;

public:
    Renderer(GDIBrush *brush,
```

```
            vector<point>& renderingRoots,
            ColorMatrix& colors,
            vector<point>& selection)
        : brush(brush), renderingRoots(renderingRoots),
          colors(colors), selection(selection)
        {}
};
```

看到这里你可能会说："怎么看上去我们要面对的状况跟原来一样呢？这个构造函数接受一个GDIBrush引用，而我们还是没法在测试用具中创建后者的对象！所以这番修改到底带给了我们什么好处呢？"稍安毋躁，马上你就会发现情况完全不同了。

331

完成了构造函数之后，我们便可以接着往该类中添加另一个方法了，该方法负责原draw()方法的工作。同样可以将它起名为draw()。

```
class Renderer
{
private:
    GDIBrush *brush;
    vector<point>& renderingRoots;
    ColorMatrix& colors;
    vector<point>& selection;

public:
    Renderer(GDIBrush *brush,
             vector<point>& renderingRoots,
             ColorMatrix& colors,
             vector<point>& selection)
        : brush(brush), renderingRoots(renderingRoots),
          colors(colors), selection(selection)
        {}

    void draw();
};
```

现在我们将原来的draw()方法的方法体复制到Renderer的draw()当中，然后利用依靠编译器技术告诉我们哪儿需要改动：

```
void Renderer::draw()
{
    for(vector<points>::iterator it = renderingRoots.begin();
            it != renderingRoots.end();
            ++it) {
        point p = *it;
        ...
        drawPoint(p.x, p.y, colors[n]);
    }
    ...
}
```

如果Renderer上的draw()用到了GDIBrush上的任何成员变量或方法的话，编译器就会替我们找出。要让编译通过，我们可以简单地为draw()所依赖的那些成员变量分别引入一个获取方法，并将它依赖的那些方法设成公有的即可。本例中，实际上draw()只依赖于一个叫做

drawPoint的私有方法。因而我们只需把它改成公有的，便可以在Renderer类中直接访问它了。

332 OK，现在我们可以在GDIBrush::draw()中将任务委托给Renderer的draw()了：

```
void GDIBrush::draw(vector<point>& renderingRoots,
                    ColorMatrix &colors,
                    vector<point>& selection)
{
    Renderer renderer(this, renderingRoots,
                      colors, selection);
    renderer.draw();
}
```

好，现在回到Renderer对GDIBrush的依赖问题上来。若我们无法在测试用具中实例化GDIBrush，则仍可以使用接口提取（285页）来完全解除Renderer对GDIBrush的依赖。关于接口提取技术书中有详细介绍，简而言之，我们需建立一个空接口，然后让GDIBrush实现它。本例中我们可以将该接口叫做PointRenderer，因为我们想要通过该接口访问的GDIBrush方法其实只是drawPoint。接下来，我们将Renderer中持有的对GDIBrush的引用改为对PointRenderer接口的引用，再编译，然后让编译器告诉我们应该往接口上添加哪些方法。最后的代码像这样：

```
class PointRenderer
{
    public:
        virtual void drawPoint(int x, int y, COLOR color) = 0;
};


class GDIBrush : public PointRenderer
{
public:
    void drawPoint(int x, int y, COLOR color);
    ...
};


class Renderer
{
private:
    PointRender *pointRenderer;
    vector<point>& renderingRoots;
    ColorMatrix& colors;
    vector<point>& selection;

public:
    Renderer(PointRenderer *renderer,
             vector<point>& renderingRoots,
             ColorMatrix& colors,
             vector<point>& selection)
        : pointRenderer(pointRenderer),
333       renderingRoots(renderingRoots),
          colors(colors), selection(selection)
        {}
```

```
    void draw();
};

void Renderer::draw()
{
    for(vector<points>::iterator it = renderingRoots.begin();
            it != renderingRoots.end();
            ++it) {
        point p = *it;
        ...
        pointRenderer->drawPoint(p.x,p.y,colors[n]);
    }
    ...
}
```

图25-1是UML图。

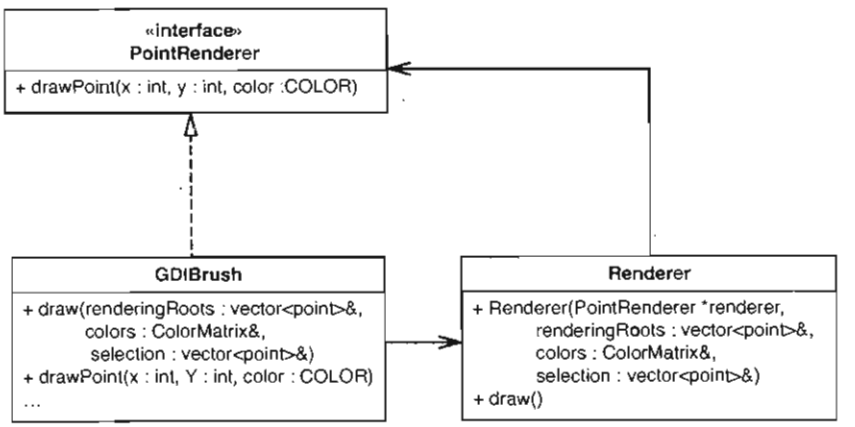


图25-1　分解出方法对象之后的GDIBrush

这个结果看起来有点怪异。我们让一个类(GDIBrush)实现了一个新的接口(PointRenderer)，然而该接口的唯一一个用户却是一个由该类创建出来的对象。你可能感觉不爽，因为我们为了实现该技术，将原类(GDIBrush)中的某些原本是私有的东西给暴露(公有)出来了。比如GDIBrush上的原本私有的drawPoint方法现在就成了公有方法，完全暴露给了外界。但是别慌，因为这还不是结局。

334

随着时间的推移，你会对没法将GDIBrush放入测试用具感到越来越不满，于是就会开始尝试对其进行解依赖。等你成功将其放入测试用具之后，就会开始思考其他设计方案。例如，PointRenderer必须是接口吗？难道不能将它做成一个类并让它持有一个GDIBrush对象？如果可以，那么或许你就可以基于这一新概念开始改进你的设计了。

以上还只是一个简单的例子，一旦GDIBrush被测试覆盖，我们便可以做许许多多其他的事情，所以最后的代码结构可能会大相径庭。

分解出方法对象技术有几个变种。最简单的如原方法不使用原类中任何实例成员的情况。这种情况下我们无需传原类的引用给它。

另一些时候，目标方法只使用原类中的数据成员而不使用其方法。这时我们往往可以建立一个新类，将被用到的数据成员放到该类中，然后传递该类的对象给分解出的方法对象。

本节中展示的情况其实是最糟的一种：被分解出来的方法用到了原类上的方法。因此我们用了接口提取，并在提取出的方法对象与原类之间建立起一定程度的抽象。

步骤

以下步骤用于在没有测试的情况下安全地分解出方法对象：

(1) 创建一个将包含目标方法的类。

(2) 为该类创建一个构造函数，并利用签名保持（249页）手法来让它具有跟目标方法完全一样的参数列表。如果目标方法用到了原类中的成员变量或方法的话，再往该构造函数的参数列表里面加上一个对原类的引用（添加为第一个参数）。

(3) 对于构造函数参数列表里面的每个参数，创建一个相应的成员变量，类型分别与对应的参数类型完全相同。这一步仍可以利用签名保持手法：将构造函数参数列表内的所有参数直接复制到成员变量声明区段，并对格式作适当调整。在构造函数里面对刚才建立的所有成员变量赋值或初始化。

(4) 在新类中建立一个空的执行方法。通常该方法可以叫做`run`。前面的例子中使用的是`draw()`。

(5) 将目标方法的方法体复制到刚才创建的执行方法中，然后编译，依靠编译器发现下一步所要作的修改。

(6) 编译出错信息应当会告诉你该方法在哪儿使用了原类的方法或成员变量。作相应改动以便令代码通过编译。通常可以通过改用原类的引用（指针）来调用其成员方法来达到目的。或者也有可能你需要将原类中的相应方法置为公有，如果是成员变量，或许还得为其引入获取方法函数，以免将其直接暴露出来。

335

(7) 新类通过编译之后，回到原先的目标方法，对其进行修改，让它将工作全权委托给上面创建出来的方法对象（只需创建新类的实例，然后调用其执行方法即可）。

(8) 如果需要，使用接口提取（285页）来解开对原类的依赖。

336

25.3　定义补全

有些语言允许你在一个地方声明类型然后在另一个地方定义它。这一点体现得最明显的就是在C/C++中。在C/C++中我们可以在一处地方声明一个函数/方法，然后在另一处地方（通常是实现文件中）定义它。这一能力可以用来帮助我们解依赖。

下面就是一个例子：

```
class CLateBindingDispatchDriver : public CDispatchDriver
```

```
{
public:
                CLateBindingDispatchDriver ();
    virtual    ~CLateBindingDispatchDriver ();

    ROOTID      GetROOTID (int id) const;

    void        BindName (int id,
                          OLECHAR FAR *name);
    ...

private:
    CArray<ROOTID, ROOTID& > rootids;
};
```

以上是一个C++类。用户创建CLateBindingDispatchDrivers的对象，然后调用其BindName方法来将名字绑定到ID。然而，我们希望在测试该类时能够以另一种方式来进行名字绑定（而不是采用原来的BindName方法）。在C++中这可以通过定义补全（Definition Completion）技术来实现。BindName方法是被声明在该类的头文件中的，我们怎样才能在测试的时候替换掉它的定义呢？这样，我们在测试文件中包含其头文件，然后给目标方法提供另一个定义，如下：

```
#include "LateBindingDispatchDriver.h"

CLateBindingDispatchDriver::CLateBindingDispatchDriver() {}

CLateBindingDispatchDriver::~CLateBindingDispatchDriver() {}

ROOTID GetROOTID (int id) const { return ROOTID(-1); }

void BindName(int id, OLECHAR FAR *name) {}


TEST(AddOrder,BOMTreeCtrl)
{
    CLateBindingDispatchDriver driver;
    CBOMTreeCtrl  ctrl(&driver);

    ctrl.AddOrder(COrderFactory::makeDefault());
    LONGS_EQUAL(1, ctrl.OrderCount());
}
```

337

只需在测试中直接定义有关方法，我们便可以提供只用于测试的方法定义。我们可以给那些我们在测试时并不关心的方法定义一个空的方法体，也可以定义可用于所有测试的感知方法。

在C/C++中使用定义补全技术，这就意味着我们得为使用了定义补全的测试创建单独的可执行文件了。因为如果不这么做的话，替换的定义就会跟原有的定义在连接期产生冲突。另一个缺点就是目标类中的方法现在有了两组定义，一组位于测试源文件中，另一组位于产品代码源文件中。这可能会给代码维护带来很大负担。而且如果你的调试环境没有设置妥当的话，甚至可能会使调试器加载了错误的调试数据。因此，并不推荐使用该技术，除非你遇上了最糟糕的依赖情况。而且即便如此，我也建议你只把该技术用在解开初始依赖上。一旦解除了初始依赖，你应该就能

快速地将类置入测试之下，这时重复的定义便可以删除了。

步骤

在C++中使用定义补全技术的步骤如下：

(1) 找出你想要对其成员函数实施定义替换的类。

(2) 确认该类的成员函数定义是在源文件而非头文件中。

(3) 将该头文件包含到待测试类的测试源文件中。

(4) 确保该类的源文件并不参与构建。

(5) 构建，找出没替换定义的成员函数。

338 (6) 往测试源文件中添加相应的成员函数定义，直到构建成功。

25.4　封装全局引用

在测试依赖于全局变量的代码时本质上有三个选择：想办法让它依赖的全局变量在测试期间具有另一种行为；利用连接器，连接到另一个全局变量定义；或将它封装起来，从而可以进一步进行解耦。最后一个选择称作*封装全局引用*（Encapsulate Global References）。下面是一个C++的例子：

```
bool AGG230_activeframe[AGG230_SIZE];
bool AGG230_suspendedframe[AGG230_SIZE];

void AGGController::suspend_frame()
{
    frame_copy(AGG230_suspendedframe,
               AGG230_activeframe);
    clear(AGG230_activeframe);
    flush_frame_buffers();
}

void AGGController::flush_frame_buffers()
{
    for (int n = 0; n < AGG230_SIZE; ++n) {
        AGG230_activeframe[n] = false;
        AGG230_suspendedframe[n] = false;
    }
}
```

以上代码用到了几个全局数组。suspend_frame函数需要访问AGG230_activeframe和AGG230_suspendedframe这两个全局数组。初看起来似乎可以将这两个数组做成AGGController类的成员变量，然而其实行不通，因为还有另外一些类（没有展示在代码中）也要用到这两个数组。那怎么办呢？

你可能会立即想到：干嘛不使用*参数化方法*（301页），将它们作为参数传给suspend_frame函数呢。实际上这么做有一个问题：如果suspend_frame调用了某个函数，而后者也使用了该全局变量的话，我们就必须同样将该变量作为参数传递给它。如本例中的flush_frame_buffer。

另一个选择是将两个全局数组传递给AGGController的构造函数。这么做是可行的，但实

际上还可以进一步检查一下这两个全局数组还在哪些地方被用到，如果发现它们每次都是被一起用到的，则可以考虑把它们绑在一起。

> 如果若干全局变量总是被一起使用一起修改，则它们应属同一个类。

应付这种状况的最佳办法就是建立一个“智能”的类（并给它想一个好名字）来持有这两个全局变量。有时候这并不像听起来那么简单。我们需要考虑这些全局变量在设计中的意义以及它们为什么在那里。如果为它们建立了一个新类，那么我们迟早会往里面添加/转移方法的，而且很可能这些方法的代码早已存在于某些使用这些变量的代码段中了。 339

> 命名一个类的时候，考虑最终会位于它里面的方法。当然我们应当给它起一个好名字，但并不一定是完美的。别忘了，你总是可以重命名它的。

在上例中，我期望随着时间的推移，frame_copy和clear能被移至我们将要建立的新类中。那么，有哪些工作对于这两个全局变量来说是公共的呢？AGGController 的suspend_frame函数或许可以被移到新类中，只要后者包含那两个数组（suspended_frame和active_frame）就行。我们可以管这个新类叫什么呢？可以叫Frame，并说明每个Frame包含一个活动（active）缓冲区和悬置（suspended）缓冲区。这一做法要求我们对系统中的概念作一些修改，并重命名几个变量，但我们所得到的是一个智能的、隐藏了更多细节的类。

> 你想到的类名可能已经被用掉了。这时候可以考虑重命名那些使用了该名字的实体，从而将该名字腾出来。

下面就是具体的步骤。

首先创建如下的类：

```
class Frame
{
public:
    // declare AGG230_SIZE as a constant
    enum { AGG230_SIZE = 256 };

    bool AGG230_activeframe[AGG230_SIZE];
    bool AGG230_suspendedframe[AGG230_SIZE];
};
```

我们故意保留了这两个数组原来的名字，这是为了简化后面的步骤。接下来声明Frame类的一个全局对象：

```
Frame  frameForAGG230;
```

接着，将两个全局数组原本的声明注释掉，并构建：

```
// bool AGG230_activeframe[AGG230_SIZE];
// bool AGG230_suspendedframe[AGG230_SIZE];
```

340

这时，编译器会告诉我们AGG_activeframe和AGG230_suspendedframe不存在。如果你的构建系统足够烂的话，它可能会徒劳地试图通过连接来寻找符号，结果给出长达10页的连接错误。但是别慌张，这些都是意料之中的。

要解决所有这些错误。我们可以顺着编译错误找到每一行出错代码，将里面对这两个全局数组的引用加上“frameForAGG230”前缀，如下：

```
void AGGController::suspend_frame()
{
    frame_copy(frameForAGG230.AGG230_suspendedframe,
               frameForAGG230.AGG230_activeframe);
    clear(frameForAGG20.AGG230_activeframe);
    flush_frame_buffer();
}
```

完成这些之后，你会发现代码更加丑陋了，但重要的是它能正确编译运行，所以说前面的步骤是保持行为的。完成了这些工作之后，我们便可以通过AGGController类的构造函数来传递Frame对象，并得到所需的分离了。

从引用一个简单的全局变量到引用一个类成员只是第一步。之后你还需要考虑是否应当使用引入静态设置方法或参数化构造函数，又或者参数化方法。

以上我们创建了一个新类，将两个全局变量添加到其中并设成公有的。为什么要这么做呢？毕竟我们已经花了一些时间思考新类应该叫什么名字以及什么样的方法可以放到里面。我们本可以创建一个伪Frame对象，并在AGG_Controller中将任务委托给它；然后我们可以把所有用到了这两个变量的代码都移到一个真正的Frame类中。没错，我们是可以这么做。但凡事不宜操之过急。更何况手头还没有测试，而且我们要花尽量少的工作先将测试安置到位再说，这时候最好能不碰就不去碰代码中的逻辑。我们应当尽量避免触动这些逻辑，并尝试往代码中引入接缝，以便通过这些接缝来植入测试用的代码或数据。后面，当测试逐渐丰富的时候，便可以开始放心地将行为从一个类转移到另一个类了。

一旦实现了将Frame对象传递进AGGController之后，我们便可以做一点小小的重命名，让代码变得稍微清晰一些。以上重构之后的代码可能像这样：

```
class Frame
{
public:
    enum { BUFFER_SIZE = 256 };
    bool activebuffer[BUFFER_SIZE];
    bool suspendedbuffer[BUFFER_SIZE];
};

Frame  frameForAGG230;

void AGGController::suspend_frame()
{
    frame_copy(frame.suspendedbuffer,
```

341

```
                frame.activebuffer);
    clear(frame.activeframe);
    flush_frame_buffer();
}
```

看起来改进不大，是不是？但实际上这是非常有价值的第一步。在将数据转移到一个类中之后，我们便拥有了分离，并能够在接下来的工作中不紧不慢地将代码朝着良性的方向改进。我们甚至看到了创建一个FrameBuffer类的潜在可能性。

在使用封装全局引用手法时，从数据或小型方法开始着手。稍大一点的方法可以等测试到位之后再移至新类中。

前面的例子展示了如何对全局数据作封装全局引用。实际上，不仅是全局数据，对于C++中的非成员函数（也称自由函数），也可以做同样的事情。常常，当你面对的是C API时，会发现你想要改进的代码里面随处可见对全局函数的调用，这时你唯一的接缝就是被调用函数的连接期接缝。你可以使用*连接替换*（296页）手法来实现分离，但实际上还有更好的选择，如果你使用*封装全局引用*来建立另一种接缝的话，就能得到更佳的代码结构。下面就是一个例子：

在一块我们想要测试的代码中，有两个对全局函数的调用：GetOption(const string optionName)和SetOption(string name, Option option)。这两个函数是自由函数，没有附着在任何类上，但代码中到处都用到了它们，比如下面这段：

```
void ColumnModel::update()
{
    alignRows();
    Option resizeWidth = ::GetOption("ResizeWidth");
    if (resizeWidth.isTrue()) {
        resize();
    }    else {
        resizeToDefault();
    }
}
```

遇到这类情况，我们可以诉诸一些老技术，如*参数化方法*（301页）和*提取并重写获取方法*（278页），但如果这些调用跨越多个方法多个类，则使用*封装全局引用*就要干净一些了。做法如下，首先创建一个新类：

342

```
class OptionSource
{
public:
    virtual          ~OptionSource() = 0;
    virtual Option   GetOption(const string& optionName) = 0;
    virtual void     SetOption(const string& optionName,
                               const Option& newOption) = 0;
};
```

该类包含了我们所需的每个自由函数的抽象成员版本。下一步，从OptionSource派生出一个伪类。比如说这儿我们可以让该伪类持有一个vector或map，内含测试期间用到的Option对象。

我们可以给该伪类提供一个add成员函数，或者也可以直接让它的构造函数接受一个map，哪种方便就选哪种。完成伪类之后，再创建真正的OptionSource：

```
class  ProductionOptionSource : public OptionSource
{
public:
    Option GetOption(const string& optionName);
    void SetOption(const string& optionName,
                   const Option& newOption) ;
};

Option ProductionOptionSource::GetOption(
        const string& optionName)
{
    ::GetOption(optionName);
}

void ProductionOptionSource::SetOption(
        const string& optionName,
        const Option& newOption)
{
    ::SetOption(optionName, newOption);
}
```

要封装对全局自由函数的引用，只需创建一个接口类，然后从它派生出伪类及产品类。产品类中的代码什么都不用做，只需直接委托/调用相应的全局函数即可。

这一重构表现不错。我们引入了接缝，并最终将任务简单地委托给相应的全局API函数完成。之后我们便可以对目标类进行参数化，让它接受一个OptionSource对象（通过指针或引用），
343 然后我们便可以在测试时向它传递伪OptionSource对象，并在产品代码中传入真正的对象。

在上例中，我们将函数放入类，并把它们做成虚的。但可不可以不这么做呢？可以。我们可以建立一些自由函数，让它们委托给其他自由函数，或者将它们做成一个新类的静态函数，但这两种方案都不能提供良好的接缝。根据它们的做法，我们就不得不使用连接期接缝（32页）或预编译期接缝（29页）来替换函数实现了。然而，若是使用类/虚函数方案并辅以参数化类的话，引入的接缝就既明显又易于掌控了。

步骤

封装全局引用手法的步骤如下：

(1) 找出有待封装的全局变量/函数。

(2) 为它们创建一个类（你将通过该类来引用它们）。

(3) 将全局变量/函数复制到该类中。如果其中有些是变量，别忘了在类的构造函数中进行适当的初始化。

(4) 将全局变量/函数的原始声明注释掉。

(5) 声明新类的一个全局对象。

(6) 依靠编译器（251页）帮你找出所有用到了这些全局变量/函数的地方。

(7) 将所有对它们的引用加上刚建立的那个新类的全局对象为前缀。

(8) 在想要使用伪对象的地方，利用引入静态设置方法（292页）、参数化构造函数（297页）、参数化方法（301页）或以获取方法替换全局引用（313页）。

344

25.5　暴露静态方法

对付那些没法在测试用具中实例化的类是件麻烦事。下面我就为你介绍一项我有时候会使用的技术。假设你有一个方法，该方法不使用实例变量或其他方法，就可以将它设成静态的。而一旦它成了静态的，你便无需实例化其类就可以将它置于测试之下了。下面是一个Java的例子。

RSCWorkflow有一个叫做validate的方法，现在我们需要添加一个新的验证条件。然而遗憾的是，该方法所在的类很难实例化。这里就不把整个类列出来了，免得你头疼。仅列出需要修改的方法：

```
class RSCWorkflow
{
    ...
    public void validate(Packet packet)
            throws InvalidFlowException {
        if (packet.getOriginator().equals( "MIA")
                || packet.getLength() > MAX_LENGTH
                || !packet.hasValidCheckSum()) {
            throw new InvalidFlowException();
        }
        ...
    }
    ...
}
```

怎么才能将这个方法置于测试之下呢？仔细一看我们就会发现，validate方法用到了Packet上的许多方法。实际上，把validate整个移到Packet类上面倒的确是个不错的主意，但就目前来说这么做的风险还是大了点，比如首先我们就没法实施签名保持（249页）。所以如果你没有方法转移的自动化支持的话，通常最好还是先把测试安置到位再说。*暴露静态方法*（Expose Static Method）手法可以帮你做到这一点。一旦测试到位，就可以放心做所需的改动，并大胆改进代码了。

> *在没有测试的情况下解依赖时，尽可能对方法进行签名保持。对整个方法进行剪切/复制可以降低引入错误的可能性。*

validate的代码并没有依赖于任何的实例变量或方法。所以，如果把validate设成公有静态的会怎样呢？那样就可以在任何地方这样调用它：

```
RSCWorkflow.validate(packet);
```

345

很可能当初RSCWorkflow的创建者根本没有想到会有这么一天，它的validate方法被做成静态的，更不用说公有了。但这是不是说这么做就不对了呢？非也。封装对于类来说固然是件好

事，但类的静态部分其实并不属于该类。实际上，在某些语言中，它隶属于另一个类，有时候也叫做元类。

静态方法不会访问类的任何私有数据，它只是一个实用方法。如果把它设成公有的，就可以编写测试了。之后如果你想要将该方法转移到另一个类中去，这些测试就会是你的强大后盾。

实际上，静态方法和数据表现得就好像它们是属于另一个类的一样。比如静态数据的生命周期是整个程序，而不是随着特定的实例生灭。此外静态成员无需实例便可以访问。

一个类的静态区段可以看作是“临时场地”，用于存放不是十分隶属于该类的东西。如果你看到某个方法没有使用任何实例数据，那么把它设成静态的是个好主意，这样可以让它变得醒目，直到你弄清它应该属于哪个类。

下面就是对RSCWorkflow类提取静态方法之后的样子：

```
public class RSCWorkflow {
    public void validate(Packet packet)
            throws InvalidFlowException {
        validatePacket(packet);
    }

    public static void validatePacket(Packet packet)
            throws InvalidFlowException {
        if (packet.getOriginator() == "MIA"
                || packet.getLength() <= MAX_LENGTH
                || packet.hasValidCheckSum()) {
            throw new InvalidFlowException();
        }
        ...
    }
    ...
}
```

在某些语言中其实还可以更简单——只需直接把原来的方法设为静态的即可。如果该方法被其他类用到了，则那些用它的地方仍然可以工作，也就是说可以通过实例来调用静态方法，如下：

```
RSCWorkflow workflow = new RCSWorkflow();
...
// static call that looks like a non-static call
workflow.validatePacket(packet);
```

346

不过在有些语言中这么做会招来编译警告。而如果没有编译警告当然是最好不过的了。

如果你担心之后又会有人来使用这个静态方法从而带来依赖问题，可以考虑使用非公有的访问限制。比如在Java和C#中有包内可见性或内部可见性，你可以用它们来限制别人对你的静态方法的访问，或把它做成受保护的并通过一个测试基类来访问它。在C++中也可以做类似的事情：可以把你的静态方法设为受保护的，或引入一个名字空间。

步骤

*暴露静态方法*手法的步骤如下：

(1) 编写一个测试，访问你打算设为公有静态的那个方法。

(2) 将目标方法的方法体提取到一个静态方法中。记住实施*签名保持*（249页）。给这个方法起一个新的名字，看一看它的参数名，或许会有所启发。例如一个名叫validate的方法接受一个Packet参数，那么就可以提取出一个叫做validatePacket的静态方法。

(3) 编译。

(4) 如果收到关于访问实例变量或方法的编译错误，看一下那些被访问到的变量或方法，看它们能否也能被设为静态的。如果可以，就将它们也一并设为静态的并通过编译。 347

25.6　提取并重写调用

许多时候，在测试时遇到的依赖问题都是相当局部的。比如我们可能会遇到一个想要替换掉的方法调用。于是若能解开对那个方法的依赖的话，就能够防止测试带来古怪的副作用，或可以感知被传给该调用的值。

例如：

```
public class PageLayout
{
    private int id = 0;
    private List styles;
    private StyleTemplate template;
    ...
    protected void rebindStyles() {
        styles = StyleMaster.formStyles(template, id);
        ...
    }
    ...
}
```

PageLayout调用了一个名叫formStyles的函数，后者位于一个叫做StyleMaster的类上。该调用的返回值被赋给一个实例变量：styles。那么，倘若我们想通过formStyles来进行感知，或者想解开对StyleMaster的依赖，该怎么办呢？有一个选择是，将该调用提取到一个新方法中，并用一个测试子类来覆盖它。这一手法也被称为*提取并重写调用*（Extract and Override Call）。

提取之后的代码像这样：

```
public class PageLayout
{
    private int id = 0;
    private List styles;
    private StyleTemplate template;
    ...
    protected void rebindStyles() {
        styles = formStyles(template, id);
        ...
```

```
    }

    protected List formStyles(StyleTemplate template,
                              int id) {
        return StyleMaster.formStyles(template, id);
    }
    ...
}
```

348

一旦有了我们自己的formStyles方法，便可以通过重写它来解开依赖了。由于styles的值和我们正在测试的东西没有任何关系，所以可以直接让重写的formStyles返回一个空列表。

```
public class TestingPageLayout extends PageLayout {
    protected List formStyles(StyleTemplate template,
                              int id) {
        return new ArrayList();
    }
    ...
}
```

如果我们的测试需要各种各样的styles，则我们可以通过修改这个方法来配制所需的styles。

*提取并重写调用*是个非常有用的重构手法，我经常会用到它。如果你的目的是解开对全局变量和静态方法的依赖，它是个理想的选择。一般来说，如果对于同一个全局对象没有太多位于不同地点的调用的话，我倾向于采用该手法；否则，我往往会采用*以获取方法替换全局引用*的办法。

如果手头有自动重构工具，则该手法实施起来简直太容易了。你只需使用*方法提取*来对目标方法进行提取即可。然而如果没有重构工具，则可以考虑如下步骤，遵循它们可以让你在即使没有测试的情况下也能安全地完成提取。

步骤

*提取并重写调用*手法的步骤如下：

(1) 确定你想要提取的调用。找出它所调用的方法声明。复制其方法签名以便实施*签名保持*(249页)。

(2) 在当前类上创建一个新方法，用刚刚复制的方法签名来武装它。

349

(3) 把对目标方法的调用复制到新方法中，然后在原来的地方改调用这个新方法。

25.7　提取并重写工厂方法

在你试图将一个类纳入测试时，可能会发现构造函数中的对象创建令你头疼不已。有时候你并不想在测试的时候创建这些对象。还有些时候你或许只是想放置一个感知对象。然而现实是你没法做到这一点，因为这些对象的创建被固定在构造函数体中了。

> 构造函数中固定了的初始化工作可能会给测试带来很大的麻烦。

让我们来看一个具体的例子：

```
public class WorkflowEngine
{
    public WorkflowEngine () {
        Reader reader
            = new ModelReader(
                AppConfig.getDryConfiguration());

        Persister persister
            = new XMLStore(
                AppConfiguration.getDryConfiguration());

        this.tm = new TransactionManager(reader, persister);
        ...
    }
    ...
}
```

WorkflowEngine的构造函数中创建了一个TransactionManager。如果这个对象是在其他地方被创建的话，我们便能更容易地引入分离。要实现这个条件，选择之一便是使用提取并重写工厂方法（Extract and Override Factory Method）。

提取并重写工厂方法是个相当强大的手法，但它存在一些语言相关的问题。例如它在C++里面是行不通的。C++并不允许在构造函数中的虚函数调用被决议到派生类中去。而Java及许多其他语言则允许这一点。那么在C++中怎么办呢？可以考虑用替换实例变量（317页）和提取并重写获取方法（278页）来代替。关于这个问题替换实例变量一节有示例及讨论。

```
public class WorkflowEngine
{
    public WorkflowEngine () {
        this.tm = makeTransactionManager();
        ...
    }

    protected TransactionManager makeTransactionManager() {
        Reader reader
            = new ModelReader(
                AppConfiguration.getDryConfiguration());

        Persister persister
            = new XMLStore(
                AppConfiguration.getDryConfiguration());

        return new TransactionManager(reader, persister);
    }
    ...
}
```

350

有了工厂方法之后，便可以将它子类化并重写了，我们可以在重写的工厂方法中返回我们想要返回的东西：

```
public class TestWorkflowEngine extends WorkflowEngine
{
    protected TransactionManager makeTransactionManager() {
        return new FakeTransactionManager();
    }
}
```

步骤

提取并重写工厂方法的步骤如下：

(1) 找出构造函数中的新建对象处。

(2) 将所有涉及新建该对象的代码通通转移到一个工厂方法中。

351 (3) 创建一个测试子类，重写刚才的那个工厂方法以避免测试期间对问题类型的依赖。

25.8　提取并重写获取方法

提取并重写工厂方法（276页）在分离对于类型的依赖方面是个强大的手法，但它并不是万能的。其最大的问题就是不适用于C++。在C++里面你无法在基类构造函数中调用派生类的虚函数。但所幸还是有解决办法的，如果你只在构造函数中创建新对象，并且并不用这个新对象做其他任何事情的话，本节的手法是适用的。

该手法的关键在于为你想要替换的成员变量引入一个获取方法，以便可以通过该获取方法来换入伪对象。引入了获取方法之后，将该类中所有使用该对象的地方改为通过获取方法来获取。这么一来，你就可以在派生类中通过子类化并重写该获取方法来换入测试用的对象了。

本例中，WorkflowEngine的构造函数中创建的是一个TransactionManager。我们修改的最终目标是要让该类能在产品环境下使用真正的TransactionManager，而在测试环境下使用测试用的TransactionManager（比如一个只作感知用途的伪TransactionManager）。

代码一开始如下所示：

```
// WorkflowEngine.h
class WorkflowEngine
{
private:
    TransactionManager    *tm;
public:
    WorkflowEngine ();
    ....
}

// WorkflowEngine.cpp
WorkflowEngine::WorkflowEngine()
{
    Reader *reader
        = new ModelReader(
            AppConfig.getDryConfiguration());
```

```
    Persister *persister
        = new XMLStore(
            AppConfiguration.getDryConfiguration());

    tm = new TransactionManager(reader, persister);
    ...
}
```

最终的代码如下所示：

```
// WorkflowEngine.h
class WorkflowEngine
{
private:
    TransactionManager    *tm;

protected:
    TransactionManager    *getTransaction() const;

public:
                           WorkflowEngine ();
    ...
}


// WorkflowEngine.cpp
WorkflowEngine::WorkflowEngine()
:tm (0)
{
    ...
}

TransactionManager *getTransactionManager() const
{
    if (tm == 0) {
        Reader *reader
            = new ModelReader(
                AppConfig.getDryConfiguration());

        Persister *persister
            = new XMLStore(
                AppConfiguration.getDryConfiguration());

        tm = new TransactionManager(reader,persister);
    }
    return tm;
}
...
```

352

353

我们所做的第一件事情就是引入一个*迟求值*的获取方法，该方法会在第一次被调用的时候创建TransactionManager对象。然后我们将该类里面所有用到该对象的地方都改为通过调用这个获取方法来获得它。

一个迟求值的获取方法在调用方看来跟其他获取方法并没什么两样。但有一点关键的区别，就是迟求值的获取方法在第一次被调用的时候才去创建被返回的对象。为此它们经常会包含如下的逻辑（注意thing是怎样被初始化的）：

```
Thing getThing() {
    if (thing == null) {
        thing = new Thing();
    }
    return thing;
}
```

迟求值的获取方法也用在单件模式中。

一旦有了这个获取方法，我们便可以对WorkflowEngine进行子类化，并重写该获取方法，以换入我们自己的测试用对象：

```
class TestWorkflowEngine : public WorkflowEngine
{
public:
    TransactionManager  *getTransactionManager()
                         { return &transactionManager; }

    FakeTransactionManager  transactionManager;
};
```

使用提取并重写获取方法（Extract and Override Getter）手法时，你须得对对象的生命周期格外小心，尤其是对于像C++这样的没有内建垃圾收集的语言。确保你释放测试用对象的方式跟产品代码释放产品用对象的方式是一致的。

在测试中，只要我们需要，就可以很容易地访问伪的TransactionManager对象：

```
TEST(transactionCount, WorkflowEngine)
{
    auto_ptr<TestWorkflowEngine>  engine(new TestWorkflowEngine);
    engine.run();
    LONGS_EQUAL(0,
        engine.transactionManager.getTransactionCount());
}
```

提取并重写获取方法手法的缺点之一便是问题成员变量可能在未被初始化之前就被不小心访问到。所以最好确保类里面的所有用到该成员变量的地方都是通过我们的获取方法来访问的。

354

其实我并不常用这一手法。如果我发现问题对象上只有唯一一个方法的话，使用提取并重写调用（275页）会容易得多。但如果同一对象上有多个问题方法的话，提取并重写获取方法就是更好的选择了。想想看，只需提取出一个获取方法然后重写它，问题就都解决了。这样的结局当然是最好的。

步骤

提取并重写获取方法手法的步骤如下：

(1) 找出需要为其引入获取方法的对象。

(2) 将创建该对象所需的所有逻辑都提取到一个获取方法中。

(3) 将所有对该对象的使用都替换为通过该获取方法来获取，并在所有构造函数中将该对象的引用初始化为null（C++中则是指针初始化为0）。

(4) 在获取方法里面加入“首次调用时创建”功能，这样当成员引用为null时该获取方法就会负责创建新对象。

(5) 子类化该类，重写这个获取方法并在其中提供你自己的测试用对象。 355

25.9　实现提取

接口提取（285页）是个有用且用起来既方便又顺手的技术，但它也有其困难之处：命名。我常常遇到这样的情况：我想要提取出一个接口，然而却发现我想给它取的名字已经被当前类给占用了。这时如果我的IDE支持重命名类或接口提取的话，情况就会很简单。但如果不幸不支持，那么就只能退而求其次了，一般来说有如下几个选择：

- 起一个愚蠢的名字。
- 看看你想要放到接口中的方法是不是全都是当前类上的公有方法，是的话或许能从它们的名字中得到启发从而给你的接口想出个名字来。

通常我不赞成往当前类的名字前草草加上一个“I”前缀就了事的做法，除非这种做法已经是你的代码基里面的命名惯例。想想看：一个代码基里面差不多一半的名字是带“I”前缀的而另一半则不带，而且你对这块代码又不熟——真是没比这更糟糕的事情了。当你使用一个类型的名字时，有一半的几率会用错。要么没加“I”，要么加了。

> 命名是设计的关键部分。好的名字有助于人们理解系统，并令系统更易对付。反之，糟糕的名字则会影响理解，并给你身后的程序员带来无尽烦恼。

如果一个类的名字恰恰适合用来作你的接口名，而且你手头又没有自动重构工具的话，可以使用实现提取（Extract Implemeter）手法来获得所需的分离。提取一个类的实现时，只需从它派生出一个新类，并将其中的所有具体方法都塞到这个派生类中，也就是说把它架空。

下面是一个C++的例子：

```
// ModelNode.h
class ModelNode
{
private:
    list<ModelNode *>    m_interiorNodes;
    list<ModelNode *>    m_exteriorNodes;
    double               m_weight;
    void                 createSpanningLinks();

public:
    void addExteriorNode(ModelNode *newNode);
```

```
    void addInternalNode(ModelNode *newNode);
    void colorize();
    ...
356 };
```

第一步是将ModelNode类的声明复制到另一个头文件中并将该副本的名字改为ProductionModelNode。下面便是复制出来的类的一部分：

```
// ProductionModelNode.h
class ProductionModeNode
{
private:
    list<ModelNode *>    m_interiorNodes;
    list<ModelNode *>    m_exteriorNodes;
    double               m_weight;
    void                 createSpanningLinks();
public:
    void addExteriorNode(ModelNode *newNode);
    void addInternalNode(ModelNode *newNode);
    void colorize();
    ...
};
```

接着，回到ModelNode的头文件中，删掉ModelNode里面的所有非公有成员变量/函数的声明。然后将所有剩下来的成员函数设为纯虚（抽象）的：

```
// ModelNode.h
class ModelNode
{
public:
virtual void addExteriorNode(ModelNode *newNode) = 0;
virtual void addInternalNode(ModelNode *newNode) = 0;
virtual void colorize() = 0;
    ...
};
```

这时候ModelNode就成了一个纯粹的接口类。它只包含抽象方法。由于是在C++里面，因此别忘了加上纯虚析构函数，并在一个实现文件中定义它[1]。

```
// ModelNode.h
class ModelNode
{
public:
    virtual          ~ModelNode () = 0;
    virtual void      addExteriorNode(ModelNode *newNode) = 0;
    virtual void      addInternalNode(ModelNode *newNode) = 0;
    virtual void      colorize() = 0;
    ...
};
```

1. 这是因为派生类的析构函数总是要调用基类的析构函数的，不管基类是不是抽象类。——译者注

```
// ModelNode.cpp
ModelNode::~ModelNode()
{}
```

357

现在，再回到ProductionModelNode类的头文件中，让它继承ModelNode这个抽象类：

```
#include "ModelNode.h"
class ProductionModelNode : public ModelNode
{
private:
    list<ModelNode *>    m_interiorNodes;
    list<ModelNode *>    m_exteriorNodes;

    double               m_weight;
    void                 createSpanningLinks();

public:
    void addExteriorNode(ModelNode *newNode);
    void addInternalNode(ModelNode *newNode);
    void colorize();
    ...

};
```

如此一来，ProductionModelNode应当就能清清爽爽地通过编译了。但现在如果编译系统的其余部分，你就会发现那些试图创建ModelNodes对象的代码现在不能通过编译了。于是你可以将它们修改为创建ProductionModelNodes对象。好，到目前为止我们的重构只是将代码中创建某具体类的对象的地方改为创建另一个具体类的对象，这对系统中的依赖情形并没有任何改善。然而，由于有了接口类ModelNodes的存在，现在我们便可以考察所有那些创建ProductionModelNodes对象的地方，看看是否可以适当运用工厂方法来进一步减少依赖了。

25.9.1　步骤

实现提取的步骤如下：

(1) 将目标类的声明复制一份，给复制的类起一个新名字。这里最好建立一个命名习惯。比如我通常使用的就是添加“Production”前缀以表明这是一个产品实现。

(2) 将目标类变成一个接口，这一步通过删除所有非公有方法和数据成员来实现。

(3) 将所有剩下来的公有方法设为抽象方法。如果你的语言是C++，则还需确保你设成抽象的那些方法都是被虚方法重写的。

358

(4) 删除该接口类文件当中的不必要的import或include，往往有许多都可以删掉。可以依靠编译器（251页）来做这一步：挨个删除import/include，看编译出不出错，出错了就添回去，否则就删掉。

(5) 让你的产品类实现该接口。

(6) 编译你的产品类以确保新接口中的所有方法都被实现了。

(7) 编译系统的其余部分，找出那些创建原类的对象的地方，将它们修改为创建新的产品类的对象。

(8) 重编译并测试

25.9.2 一个更复杂的例子

如果目标类没有任何基类或派生类，则实现提取用起来还是相对简单的；否则就需要聪明一点了。图25-2再次展示了ModelNode，不过这次是用Java写的，而且有一个基类和一个派生类：

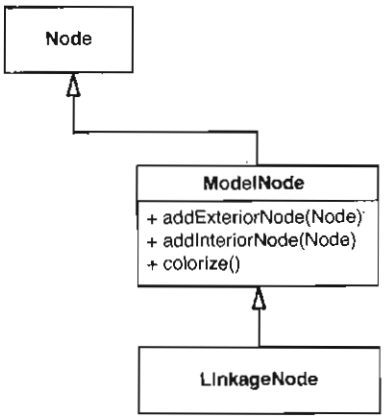
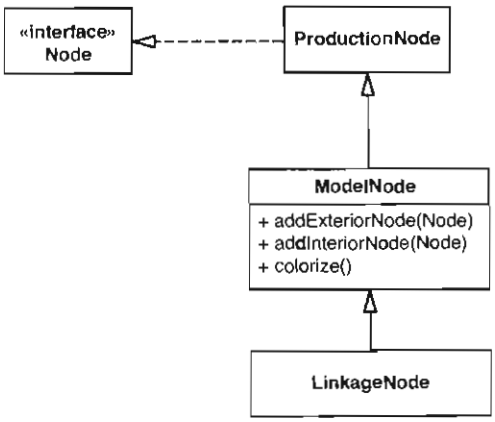


图25-2　具有基类和派生类的ModelNode

在这个设计中，Node、ModelNode以及LinkageNode都是具体类。ModelNode使用了Node
359 中的受保护方法。此外它自己也提供了供它的派生类LinkageNode使用的方法。实现提取需要一个能被转换为接口的具体类。并且在完成提取之后你会得到一个接口和一个具体类。

那么，遇到这种情况我们该怎么办呢？我们可以对Node类作实现提取，并使ProductionNode派生自Node。此外还要修改继承关系，让ModelNode继承ProductionNode而不是Node。图25-3展示了修改后的设计：



360 图25-3　在对Node作实现提取之后

下一步，对ModelNode作实现提取。由于ModelNode已经有了一个派生类，因此我们可以往ModelNode和LinkageNode之间引入一个ProductionModelNode。之后我们便可以让ModelNode接口扩展Node接口了，如图25-4所示：

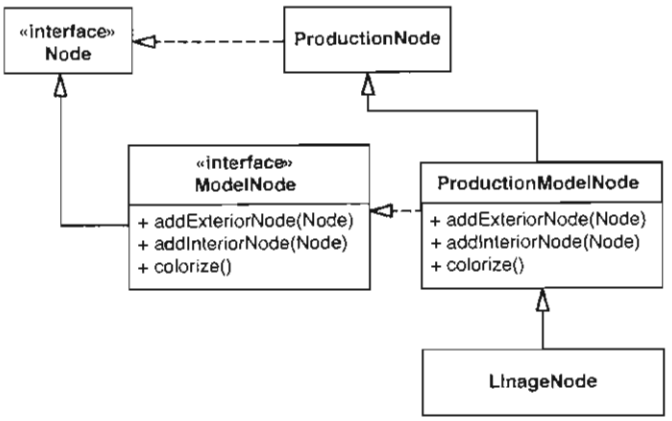


图25-4　对ModelNode实现提取

如果你发现自己将一个类像上面这样嵌入了继承体系，那么我建议你真的需要考虑一下是否应当改用接口提取，并给你的接口选择其他名字了。接口提取比实现提取要直接得多。 361

25.10　接口提取

在许多语言中接口提取（Extract Interface）都是最安全的解依赖技术之一。如果某步出错了，编译器就会立即告诉你，所以说用接口提取的时候，极少可能引入bug。接口提取的关键在于为你的类创建一个接口，该接口包含你想要在某些上下文中使用的所有方法。完成接口之后，就可以让你的类实现它，从而通过该接口来感知或分离（比如传递一个伪对象给你想要测试的类）。

接口提取有三种方式，以及一些注意点。第一种方式是利用现成的自动重构工具（如果你的IDE支持的话）。支持接口提取的工具往往会允许你选中一个类上的某些方法然后键入新接口的名字。更好一些的则会询问你是否需要它帮你将代码中的某些地方自动改为使用新接口。这些工具会帮你节省可观的工作量。

如果没有相应的工具，则可以采用第二种方式：即采用本节所讲的方法和步骤，一步一步地提取。

接口提取的第三种方式就是一次性从类里面剪切/复制出多个方法，然后将它们放到一个新的接口中。这种做法虽然没有前两种做法安全，但仍然还算是相当不错的；而且如果没有重构工具支持并且构建耗时很长的话，该方式往往是唯一的选择。

下面我们重点描述第二种方式。在讨论的过程中会提及一些需要注意的地方。

我们需要提取一个接口来将PaydayTransaction类置于测试之下。图25-5展示了

362 PaydayTransaction类的UML图，以及它所依赖的一个叫做TransactionLog的类。

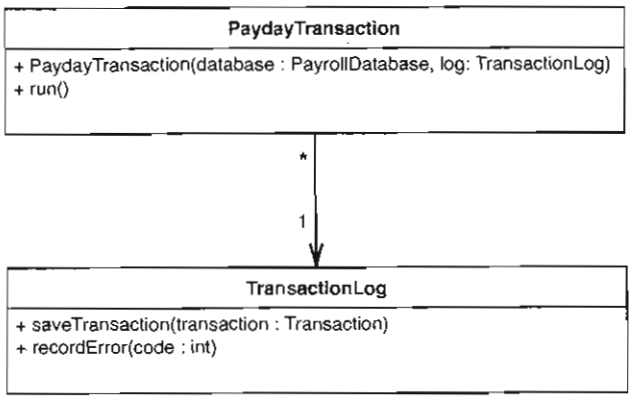


图25-5 PaydayTransaction 依赖于TRansactionLog

我们的测试用例如下：

```
void testPayday()
{
    Transaction t = new PaydayTransaction(getTestingDatabase());
    t.run();

    assertEquals(getSampleCheck(12),
                 getTestingDatabase().findCheck(12));
}
```

要想让上面的测试用例通过编译，还需一个transactionLog才行。那么就让我们假设一个FakeTransactionLog已经写好了，看看怎么用它：

```
void testPayday()
{
    FakeTransactionLog aLog = new FakeTransactionLog();
    Transaction t = new PaydayTransaction(
                            getTestingDatabase(),
                            aLog);
    t.run();

    assertEquals(getSampleCheck(12),
                 getTestingDatabase().findCheck(12));
}
```

要想让上面的代码通过编译，必须为TransactionLog类提取一个接口，然后从该接口派生出一个FakeTransactionLog，最后修改一下PaydayTransaction，让它能够接受Fake-
363 TransactionLog为参数。

但首要的是接口提取。为此我们创建一个新的空类，叫做TransactionRecorder。如果你想知道这个名字是哪来的，看一看下面的注记。

接口命名

接口，作为一种语言结构，算是比较新的事物。Java以及许多.NET语言都具备这一语言特性。在C++中你得通过创建一个只含纯虚函数的类来模拟它。

接口这个概念最初被引入语言的时候，有些人发明了用它们所来自的类的类名加上“I”前缀来命名的办法。例如，假设你有一个叫做Account的类，并且想要一个接口，于是就可以给这个接口起名叫IAccount。这种命名方式的好处就是不用动脑筋，只要加个前缀就行了。但也有它的坏处，那就是你最终会发现许多代码都不知道对付的究竟是一个接口还是一个类。当然，理想情况下你的代码应该无需关心这些。同样，最后你的代码基中将既有带“I”前缀的名字又有不带“I”前缀的名字。如果你想将一个带“I”前缀的接口做成一个普通类的话就会引发大规模的改动。而如果不改的话，那个名字又会作为一个微小的谎言躺在代码中。

在编写新类时，最简单的事情莫过于起个简单的类名了，即便是对于大型的抽象也是如此。例如，如果我们正在编写一个account包，则我们可以从一个就叫Account的类开始。然后开始编写测试来添加新特性。随着系统的增长，迟早你会想要把Account做成一个接口。这时可以在Account下创建一个派生类，并将Account中的所有数据和方法都转移到它里面，将Account架空成一个接口。这么做的好处是你无需将代码中所有引用Account的地方修改成引用新的接口（因为Account本身就是那个接口）。

在像PaydayTransaction这样的例子中，我们已经有了一个不错的候选名字（TransactionLog），这时同样可以采用上面的办法。但缺点是将所有数据成员和方法统统转移到另一个类中需要不少步骤。不过只要风险足够小，我还是会时不时用这招的。这其实就是所谓的实现提取。

如果我觉得缺少一些测试，于是想通过提取出一个接口好让更多的测试能够安置到位的话，常常会给这个接口起一个新名字。有时候想这么个名字也不是件易事。如果你手头缺少能够自动重命名类的工具的话，最好在使用该接口的代码还没有大量出现之前给它起个稳定的名字。

```
interface TransactionRecorder
{
}
```

现在，我们回过头去，让TransactionLog实现以上接口。

```
public class TransactionLog implements TransactionRecorder
{
   ...
}
```

364

下一步，创建一个空的FakeTransactionLog类：

```
public class FakeTransactionLog implements TransactionRecorder
{
}
```

这几步之后，所有代码应当仍然完全能够通过编译，因为我们所做的只不过是引入了几个新

类，并让一个既有类实现了一个空的接口。但后面就要进行真正的重构了。首先我们将想要使用接口的地方改为对接口的引用。比如PaydayTransaction原本使用的是一个TransactionLog；现在我们需将其改为引用TransactionRecorder。然后我们编译代码，并从编译错误中发现许多地方调用了TransactionRecorder上的方法，我们将相应方法加到TransactionRecorder接口中来解决这些编译错误，同时也将该方法的一个空的实现加到FakeTransactionLog中。

下面是示例代码：

```
public class PaydayTransaction extends Transaction
{
    public PaydayTransaction(PayrollDatabase db,
                             TransactionRecorder log) {
        super(db, log);
    }

    public void run() {
        for(Iterator it = db.getEmployees(); it.hasNext(); ) {
            Employee e = (Employee)it.next();
            if (e.isPayday(date)) {
                e.pay();
            }
        }
        log.saveTransaction(this);
    }
    ...
}
```

本例中，TransactionRecorder上唯一被调用的就是saveTransaction方法。而由于TransactionRecorder接口现在还是空的，所以会遇到编译错误。但我们只需将这个方法添加到TransactionRecorder上便可以了，同时也别忘了往FakeTransactionLog上添加一个空的实现：

```
interface TransactionRecorder
{
    void saveTransaction(Transaction transaction);
}

public class FakeTransactionLog implements TransactionRecorder
{
    void saveTransaction(Transaction transaction) {
    }
}
```

365

任务完成！现在我们无需在测试的时候创建真正的TransactionLog对象了。

你可能会说：“不是吧，我们还没有往接口以及伪类上添加recordError方法呢。”没错，TransactionLog上的确有这个方法。如果需要提取出TransactionLog的整个接口，我们或许的确会把recordError放到它上面，但实际情况是，我们的测试并不需要这个方法。尽管，把一个类的所有公有方法都放到接口上是个不错的做法，但如果顺着这条路走下去，就有可能要做许多不必要的工作才能将代码最终纳入测试了。如果你觉得代码的设计走向的确要求接口拥有相

应类上的所有公有方法的话，不妨考虑递增式地扩充该接口。许多时候，在得到足够的测试覆盖之前，最好避免大规模的改动。

> 提取接口时并不一定要提取类上的所有公有方法。你可以依靠编译器来帮你发现哪些方法需要加到接口上去。

该手法唯一的困难之处在于当面对非虚方法的时候。这里所谓的非虚方法在Java里面可能是静态方法，而在C#或C++里面则可能是非虚成员函数。对于这类情况请参考下文的附注。

步骤

接口提取的步骤如下：

(1) 创建一个新接口，给它起一个好名字。暂时不要往里面添加任何方法。

(2) 令你提取接口的目标类实现该接口。这一步不会破坏任何东西，因为接口上还没有任何方法。但你也可以编译确认一下。

(3) 将你想要使用伪对象的地方从引用原类改为引用你新建的接口。

(4) 编译系统。如果编译器汇报接口上缺少某某方法，则添加相应的方法（同时也往伪类上面添加一个空的实现），直到编译通过。 366

接口提取与非虚函数

如果你的代码当中有像“bondRegistry.newFixedYield(client)”这样的调用，则对于许多语言而言，光看这行调用是没法分辨出newFixedYield到底是静态方法、非虚实例方法，还是虚实例方法的。在允许非虚实例方法的语言中，如果提取接口并将目标类里面的某个非虚方法添加到其上的话，你就遇到麻烦了。一般而言，如果你的类没有派生类，则可以将目标方法设为虚的，并照常进行接口提取。一切都没问题。但如果你的类有派生类，那么把它的非虚函数设成虚的并放到接口中就会破坏既有代码的行为了。下面就是一个C++的例子。BondRegistry有一个非虚方法：

```
class BondRegistry
{
public:
    Bond *newFixedYield(Client *client) { ... }
};
```

而且BondRegistry有一个派生类，该类里面有一个同名同签名的方法：

```
class PremiumRegistry : public BondRegistry
{
public:
   Bond *newFixedYield(Client *client) { ... }
};
```

现在，假设我们从BondRegistry提取出一个接口：

```
class BondProvider
{
```

```
public:
    virtual Bond *newFixedYield(Client *client) = 0;
};
```

并令BondRegistry实现它：

367

```
class BondRegistry : public BondProvider { ... };
```

现在考虑如下的代码：

```
void disperse(BondRegistry *registry) {
    ...
    Bond *bond = registry->newFixedYield(existingClient);
    ...
}
```

如果我们传递给disperse一个PremiumRegistry对象，那么，根据原先的BondRegistry类定义，被调用的应该是非虚的那个BondRegistry::newFixedYield，因为对于非虚方法调用采用的是静态决议。然而经我们提取接口之后，被调用的就成了PremiumRegistry的newFixedYield。在C++中，只要基类中的方法是虚的，那么派生类中的相应方法，不管有没有加virtual，都会自动变成虚的。值得注意的是，在Java或C#中并没有这个问题。Java中所有的实例方法都是虚的。而C#中的情况则要安全一点，因为添加一个接口并不会影响到目前对非虚方法的调用。

一般而言，C++中，在派生类中创建一个与基类中某个非虚方法同名且同签名的方法是不好的做法，因为这样做可能会带来误解。如果你真的需要通过一个接口来访问某个类上的非虚方法，并且这个类有派生类的话，最好的做法就是添加一个新的虚方法。后者可以委托调用一个非虚甚至静态的方法。而你只要确保该方法对于你提取接口的那个类的所有派生类都做了正确的事情即可。

368

25.11　引入实例委托

人们会因为各种各样的原因使用静态方法。其中最常见的原因便是为了实现单件模式（293页）。而另一个常见的原因是使用静态方法来创建实用类。

实用类在许多设计中都是一眼就能看出来的。它们一般没有任何实例变量和实例方法。而是全部由静态方法和静态常量组成。

同样，人们也会出于各种各样的原因而创建实用类。大多数时候是因为无法为一组方法寻找出合适的公共抽象。比如JDK中的Math类。Math内有计算三角函数的静态方法（cos，sin，tan），但除此之外还有许多其他方法。当语言设计者从对象开始一路往下构建他们的语言时，应确保基本的数值对象知道如何进行这些操作。如，你应当能够对“1”这个对象调用sin()方法，并获得正确的正弦值。在本书写作的时候，Java尚不支持在基本类型上调用数学方法，因此实用类是个正当的解决方案，但它同时也是个特例。在几乎所有的情况下，你也可以使用传统的带有实例数据和方法的类来完成工作。

如果你的项目中有静态方法，则很可能它们并不会给你带来麻烦，除非其中包含了一些你没法或不想在测试的时候依赖的东西。（这个的技术术语叫做“静态粘着”。）遇到这类情况，你可能会想：“唉，要是有对象接缝（35页）就好了，那样的话就可以利用它来嵌入测试用的行为了。”那么，到底怎么办呢？

方案之一便是往目标类上引入委托实例方法。这时候，你需要想办法将对那些静态方法的调用替换为对目标类的实例方法的调用。例如：

```
public class BankingServices
{
    public static void updateAccountBalance(int userID,
                                            Money amount) {
        ...
    }
    ...
}
```

BankingServices仅包含静态方法。这里为了简化起见只写出了一个。我们可以往它上面添加一个实例方法（如下），并让后者委托那个静态方法：

```
public class BankingServices
{
    public static void updateAccountBalance(int userID,
                                            Money amount) {
        ...
    }

    public void updateBalance(int userID, Money amount) {
        updateAccountBalance(userID, amount);
    }
    ...
}
```

369

我们添加的那个实例方法叫updateBalance，它只是简单地将调用转发至静态方法updateAccountBalance。

这么一来，我们便可以将对updateAccountBalance的调用：

```
public class SomeClass
{
    public void someMethod() {
        ...
        BankingServices.updateAccountBalance(id, sum);
    }
}
```

替换为：

```
public class SomeClass
{
    public void someMethod(BankingServices services) {
        ...
        services.updateBalance(id,sum);
```

```
        }
        ...
    }
```

注意，只有当我们有办法在外部创建一个BankingServices对象从而能通过该对象来调用someMethod时，该做法才能成功。这就意味着我们需要做一些额外的重构，不过，在静态类型的语言里，可以依靠编译器（251页）来帮我们将对象放置到位。

对于许多静态方法来说，该技术还算是相当直观的。然而当遇到实用类的时候，你可能就会感觉不自然了。一个具有5到10个静态方法，却只有一两个实例方法的类看起来的确挺怪异的。而如果这仅有的两个实例方法还只是一层空壳，其功能只是将任务简单地转发给相应的静态方法，那就更加怪异了。但好处是，该技术引入了对象接缝，后者使你能够在测试时替换进另一种行为。随着时间的推移，你可能就会发现每个对该类的方法的调用都通过实例方法进行转发了，这时候你便可以将那些静态方法的方法体转移到相应的实例方法中，并删除所有静态
370 方法了。

步骤

引入实例委托（Introduce Instance Delegator）手法的步骤如下：

(1) 找出会在测试中带来问题的那个静态方法。

(2) 在它所属类上新建一个实例方法（记得用签名保持手法）。让该实例方法委托那个静态方法。

(3) 找出你想要纳入测试的类中有哪些地方使用了那个静态方法。使用参数化方法（301页）或其他解依赖技术来提供一个实例给代码中想要调用那个静态方法的地方（即改为从该实例间接
371 调用那个静态方法）。

25.12　引入静态设置方法

或许我是个纯粹主义者，但我确实不喜欢全局变量。在协助团队的过程中我常常发现，阻挠我们将一块代码放进测试用具的最大敌人就是它们。比如你想把一组类放入测试用具，却发现其中有些需要被设置成某些特定状态才能使用。于是在你将测试用具架起来了之后，还得查看所有的全局变量，确保它们的状态/值满足你进行测试所需的条件。看来“超距作用”并非量子物理学家率先发现的，在软件界这一效应由来已久了。

对全局变量的抱怨暂且放一边，事实是，许多系统里面都免不了有全局变量。有些系统中全局变量的存在形式是直接的——只是由于某个程序员在全局范围内定义了一个变量。而另一些系统中它们则可能以严格遵循单件模式的单件形式存在。不管哪种情况，引入一个伪对象并用它来进行感知是个非常直接的做法。如果你的全局变量就是个赤裸裸的全局变量，那么可以干脆替换它本身。如果对它的引用是const或final的，你可能就需要将这些修饰符去掉了（同时在代码里留下注释说明你这么做只是为了方便测试，在产品代码中不应利用这个漏洞）。

单件设计模式

单件模式被许多人用来确保某个特定的类在整个程序中只可能有唯一一个实例。大多数单件实现都有以下三个共性：

(1) 单件类的构造函数通常被设为私有。

(2) 单件类具有一个静态成员，该成员持有该类的唯一一个实例。

(3) 单件类具有一个静态方法，用来提供对单件实例的访问。通常该方法名叫instance。

虽说单件模式能够防止人们在产品代码中创建不只一个目标类的实例，但它同样阻止了人们在测试用具中创建第二个实例。这是一把双刃剑。

替换单件需要多花点工夫才行。首先是往单件类上添加一个静态的设置方法以便用它来替换单件实例，然后将构造函数设为受保护的。之后就可以对单件类进行子类化，创建一个全新的对象并将它传递给那个静态的设置方法了。

这种做法可能会令你心里感到不安，因为你觉得单件类的保护给打破了，但是，别忘了，访问限制的目的在于防止错误，而我们编写测试的目的同样也是防止错误。而为了在这种情况下引 372
入测试，我们不得已才用了强硬一点的手段。

下面的例子展示了如何在C++中引入静态设置方法（Introduce Static Setter）：

```
void MessageRouter::route(Message *message) {
    ...
    Dispatcher *dispatcher
            = ExternalRouter::instance()->getDispatcher();
    if (dispatcher != NULL)
        dispatcher->sendMessage(message);
}
```

在MessageRouter类中，许多地方都使用了单件来获取Dispatcher对象（getDispatcher()）。ExternalRouter类就是其中的一个单件类，它有一个静态方法instance()来提供对全局唯一的ExternalRouter对象的访问。此外ExternalRouter上有一个getDispatcher()方法用于获取Dispatcher。要想换入我们自己的测试用Dispatcher，我们可以把提供该Dispatcher的ExternalRouter对象替换掉。

在引入静态设置方法之前，ExternalRouter类看起来像这样：

```
class ExternalRouter
{
private:
    static ExternalRouter *_instance;
public:
    static ExternalRouter *instance();
    ...
};

ExternalRouter *ExternalRouter::_instance = 0;

ExternalRouter *ExternalRouter::instance()
```

```
{
    if (_instance == 0) {
        _instance = new ExternalRouter;
    }
    return _instance;
}
```

注意，ExternalRouter单件对象是在instance()方法被第一次调用的时候创建出来的。要换入我们自己的路由对象，就必须想办法修改instance()的返回值。为此，我们首先引入一个用于替换该实例的方法：

```
void ExternalRouter::setTestingInstance(ExternalRouter *newInstance)
{
    delete _instance;
    _instance = newInstance;
}
```

373

当然，这一做法有一个假定的前提，那就是我们能够创建一个新的实例。人们在使用单件模式的时候一般都是通过将构造函数设为私有来防止外界创建多个实例的。如果我们将构造函数的访问权限改设为受保护的，就可以通过子类化该单件类来实现感知和分离，并将新的实例传给setTestingInstance方法。在上例中，我们可以创建ExternalRouter的子类，比如叫TestingExternalRouter，然后重写其getDispatcher方法，让它返回我们想让它返回的东西，即一个伪对象：

```
class TestingExternalRouter : public ExternalRouter
{
public:
    virtual void Dispatcher *getDispatcher() const {
        return new FakeDispatcher;
    }
};
```

仅仅为了换入一个新对象就大费周章地如此折腾一气，看起来似乎太夸张了点。最终创建的一个新的ExternalRouter派生类仅仅只是为了换入我们的测试用对象。当然，捷径还是有的，但每条捷径都有它们自己的缺点。比如我们可以往ExternalRouter里面添加一个布尔变量，然后根据该变量的值来决定返回产品还是测试用对象。在C++或C#中我们也可以使用条件编译来切换对象。以上这两个替代手法都可行，但它们的侵入性太强，而且如果在整个代码基中普遍采用的话就会变得比较笨拙。一般而言我喜欢把产品代码和测试代码分得清清楚楚，互相井水不犯河水。

在单件上使用设置方法和受保护的构造函数侵入性不强，却能帮助你将测试安置到位。但你或许会发出疑问："人们会不会错用我们为测试而留的后门，在产品代码中创建出多于一个的'单件'出来呢？"答案是可能的。但我觉得，如果某个实例在系统中的唯一性是如此重要的话，最好的办法就是确保团队的每个成员都意识到这一重要性。

除了降低构造函数访问限制级别并利用子类化之外，还有一个替代手法就是利用接口提取，在单件类上提取出一个接口并在该接口上提供一个能接受实现了该接口的类对象的设置方

法。但该做法也有它的缺点，那就是你必须修改用来引用单件的引用类型以及instance()方法的返回类型。这些修改可能会很棘手，而且这些改动的方向并不好。我们认为“更好的方向”是减少对单件的全局引用，最终使单件类可以成为一个普通类。

在上面的例子中，我们利用了一个静态设置方法来替换单件对象。而我们的单件对象的任务其实只是负责提供一个Dispatcher对象。偶尔我们也会在有些系统中发现另一种全局变量——一个全局工厂。它们并非持有唯一一个对象，而是在每次静态方法被调用的时候提供全新的对象。 374
对于这类情况，要想换入我们自己的对象就有点难度了，但通常你都可以通过让这个工厂委托另一个工厂来达到你的目的。比如，让我们来看一个Java的例子：

```
public class RouterFactory
{
    static Router makeRouter() {
        return new EWNRouter();
    }
}
```

RouterFactory是一个很直观的全局工厂。就它现在的这个样子，我们是没法换入测试用路由对象的，但可以对它作如下修改：

```
interface RouterServer
{
    Router makeRouter();
}

public class RouterFactory implements RouterServer
{
    static Router makeRouter() {
        return server.makeRouter();
    }

    static setServer(RouterServer server) {
        this.server = server;
    }

    static RouterServer server = new RouterServer() {
        public RouterServer makeRouter() {
            return new EWNRouter();
        }
    };
}
```

这样，在测试中我们便可以这么做：

```
protected void setUp() {
    RouterServer.setServer(new RouterServer() {
        public RouterServer makeRouter() {
            return new FakeRouter();
        }
    });
```

```
}
```

但有一点需要注意，在所有这些关于引入静态设置方法的手法里，你对程序状态的修改对于所有测试来说都是可见的。如果你使用的是xUnit测试框架，则可以使用里面的tearDown方法来将状态复位，以便后续的测试在一个已知的状态环境下执行。一般来说，仅当错误的状态用于后续的测试会引起误解时我才会使用这种方法。假设我在每个测试中都是替换进一个伪的
375 MailSender对象，那么再弄一个伪MailSender对象出来似乎没多大意义。但另一方面，如果是用全局变量来保存某些状态并且这些状态会影响到系统结果的话，通常我就会在setUp和tearDown方法里面做同样的事情——保证系统处于一个干净的状态，如下：

```
protected void setUp() {
    Node.count = 0;
    ...
}

protected void tearDown() {
    Node.count = 0;
}
```

我猜你看到这儿肯定在想："不就为了把这个测试安置到位吗，用得着这么大动干戈的嘛？"你说得没错，这些模式的确会明显丑化系统。但别忘了，手术也从来都不是漂亮的，尤其是开始的时候。那么我们怎么才能让系统重新回到体面的状态呢？

需要考虑的问题之一是参数传递。考察一下需要访问你的全局变量的类，看看能否给它们一个公共基类。如果可以，就在创建它们的时候将全局对象传递给它们，并逐渐往消除全局变量的方向靠拢。人们常常会害怕系统中每个类都会需要全局对象，但结果往往会令你大吃一惊。比如我曾经遇到过一个嵌入式系统，该系统将内存管理和错误汇报机制都封装在了类中，将一个内存对象或错误汇报对象传给任何想要它的代码。随着时间的推移，在需要这些服务的类与不需要它们的类之间就形成了一道清晰的隔离。需要它们的类都具有一个共同的基类。在系统中被传来传去的对象在程序一开始就被创建出来，你几乎觉察不到。

步骤

引入静态设置方法的步骤如下：

(1) 降低构造函数的保护权限，这样你才能够通过子类化单件类来创建伪类及伪对象。

(2) 往单件类上添加一个静态设置方法。后者的参数类型是对该单件类的引用。确保该设置方法在设置新的单件对象之前将旧的对象销毁。

(3) 如果你需要访问单件类里面的受保护或私有方法才能将其设置妥当的话，可以考虑对单
376 件类子类化，也可以对其提取接口并改用该接口的引用来持有单件。

25.13　连接替换

面向对象方法学给了我们很多替换对象的契机。比如只要两个类实现了共同的接口，或具有

共同的基类，那么我们便可以轻松地将其中一个的对象替换为另一个的对象。但遗憾的是，像C这样的过程式语言的程序员没这个福气。比如下面这个函数，如果不用预处理手段，就完全没法在编译期将其替换为另一个函数：

```
void account_deposit(int amount);
```

还有其他办法吗？有。可以使用连接替换（Link Substitution）手法来将它替换为另一个函数。实施该手法时，你创建一个哑元库，该库里面包含一个跟以上函数签名完全一样的函数。如果你的目的是感知，那么需要在该函数里面设置一些机制来保存通知消息并对它们进行查找。可以使用文件，全局变量，或其他办法，只要你觉得方便就行。

下面就是我们的测试用account_deposit：

```
void account_deposit(int amount)
{
    struct Call *call =
        (struct Call *)calloc(1, sizeof (struct Call));
    call->type = ACC_DEPOSIT;
    call->arg0 = amount;
    append(g_calls, call);
}
```

从以上代码中可以看出，我们感兴趣的是感知，所以创建了一个全局列表，该列表里面包含了每次该函数被调用时的有关信息。在测试时，我们可以在测试完一组对象之后检查该列表来确认该函数是否按照正确的顺序被调用了。

我个人从未对C++类用过这一手法，但我想道理是一样的。当然，我相信C++编译器的名字粉碎机制肯定会带来一些困难；但如果调用的是C函数，则该手法是非常可行的。其最大的用处就是用来伪造外部库内的函数。而最好伪造的又属那些纯数据汇[1]的库：对于这类库，你只是调用其中的函数，而通常并不关心其返回值。图形库就是这样的一个例子。

连接替换手法也可以用在Java中。做法是创建一个同名同方法集的类，然后修改类路径，让调用被决议到你的新类上，从而避开原类上的糟糕依赖。 377

步骤

连接替换的步骤如下：

(1) 找出你想要伪造的函数或类。

(2) 为它们编写另一份定义。

(3) 修改你的构建参数，让你的伪造品能够代替真品被连接到项目中。 378

25.14　参数化构造函数

如果你用构造函数创建了一个对象，那么通常解除对该对象的依赖的最佳办法就是将它的创

1. sink，也称“宿”、“池”。——译者注

建过程外部化，即在该类外创建该对象，然后让该类的客户代码将这个对象传给该类的构造函数。下面就是一个例子。

一开始的代码如下所示：

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

然后我们引入一个新的参数，如下：

```
public class MailChecker
{
    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

人们之所以并不常常想到该技术，是因为他们觉得这种做法等于强迫客户代码传递额外参数给该类的构造函数。然而别忘了，你还可以另外再写一个方便的构造函数，它具有跟原来的构造函数相同的签名，如下：

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this(new MailReceiver(), checkPeriodSeconds);
    }

    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

这么一来你就既保证了测试代码能够换入必要的测试用对象，又丝毫不干扰产品代码。

379

让我们一步一步来，下面是原始代码：

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
```

```
}
```

首先我们将其构造函数复制一份：

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }

    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

然后给其中一个构造函数添加一个MailReceiver型的参数：

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }

    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

接着，我们将这个参数赋给相应的实例变量，删掉原来的new表达式：

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }

    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }

    ...
}
```

380

现在，转到另一个构造函数，删除其函数体，代以对刚才那个构造函数的调用。别忘了，调用的时候new一个MailReceiver对象出来传给它。

```
public class MailChecker
{
    public MailChecker (int checkPeriodSeconds) {
        this(new MailReceiver(), checkPeriodSeconds);
    }

    public MailChecker (MailReceiver receiver,
                        int checkPeriodSeconds) {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
    ...
}
```

该技术有什么缺点吗？有的。当我们往一个构造函数上添加一个新的参数时，可能会导致进一步对该参数的类型的依赖。该类的用户可能会在产品代码中使用这个新的构造函数，从而增加系统内的依赖。不过一般来说这个问题不要紧。参数化构造函数（Parameterize Constructor）是个非常简单的重构手法，我喜欢用它。

在支持默认参数的语言中实施参数化构造函数还有更简单的办法——只需简单地给现有构造函数的目标参数加一个默认实参即可：

381

下面是一个C++类的示例：

```
class AssemblyPoint
{
public:
    AssemblyPoint(EquipmentDispatcher *dispatcher
                  = new EquipmentDispatcher);
    ...
};
```

在C++中这么做只有一个缺点——该类所在的头文件必需包含EquipmentDispatcher的头文件。如果不是因为这个默认参数，我们只需前向声明EquipmentDispatcher就够了。而也正是因为这个原因，一般我并不使用默认实参。

步骤

参数化构造函数的步骤如下：

(1) 找出你想要参数化的构造函数，并将它复制一份。

(2) 给其中的一份复制增加一个参数，该参数用来传入你想要替换的对象。将该构造函数体中的相应的对象创建语句删掉，改为使用新增的那个参数（如果需要赋值的话就将那个参数赋给相应的实例变量）。

(3) 如果你的语言支持委托构造函数[1]，那么删掉另一份构造函数的函数体，代以对刚才那个构造函数的调用，别忘了调用的时候要new一个相应对象出来。如果你的语言不支持委托构造函

1. 即从一个构造函数中调用另一个构造函数的能力。——译者注

数，则可能需要将构造函数中的共同成分提取到一个比如叫common_init的方法中。 382

25.15　参数化方法

假设你有一个方法，该方法在内部创建了某个对象，而你想要通过替换该对象来实现感知或分离。往往最简单的办法就是从外面将你的对象传进来。下面是一个C++的例子：

```
void TestCase::run() {
    delete m_result;
    m_result = new TestResult;
    try {
        setUp();
        runTest(m_result);
    }
    catch (exception& e) {
        result->addFailure(e, this);
    }
    tearDown();
}
```

run()中创建了一个TestResult对象。如果我们想要通过该对象来进行感知或分离的话，可以将它作为一个参数传进去，如下：

```
void TestCase::run(TestResult *result) {
    delete m_result;
    m_result = result;
    try {
        setUp();
        runTest(m_result);
    }
    catch (exception& e) {
        result->addFailure(e, this);
    }
    tearDown();
}
```

借助于一点函数转发技巧，我们就可以完全保留原来的那个函数签名：

```
void TestCase::run() {
    run(new TestResult);
}
```

383

C++、Java、C#以及许多其他语言都允许同一个类上有多个同名方法，前提是只要它们的签名各不相同。在上例中，我们利用了该便利，使原方法和参数化之后的方法具有同样的名字。尽管这省了点事，但有时也会带来混乱。一个替代方案是将新参数的类型名嵌入方法名中。例如，在上面的例子中，我们可以保留run()作为原方法名，同时将加了参数的那个run叫做runWithTestResult(TestResult)。

与参数化构造函数手法一样，参数化方法（Parameterize Method）也可能会导致客户代码依赖于新出现的那个参数的类型。如果你觉得这的确会带来问题的话，可以考虑改用提取并重写工厂方法。

步骤

参数化方法的步骤如下：

(1) 找出目标方法，将它复制一份。

(2) 给其中一份增加一个参数，并将方法体中相应的对象创建语句去掉，改为使用刚增加的这个参数。

(3) 将另一份复制的方法体删掉，代以对被参数化了的那个版本的调用，记得创建相应的对象作参数。

384

25.16　朴素化参数

一般来说，修改一个类的最佳途径就是在测试用具中创建它的实例，为你想要进行的修改编写相应的测试，然后作出修改来满足该测试。然而有时候为了将一个类纳入测试需要花的工夫太大了。我就曾经遇到这样的一个团队，他们接手的一个遗留系统里面的那些领域相关的类几乎直接依赖了系统内的其他所有类。然后，就像情况还嫌不够糟似的，这些类居然还统统被绑进了一个持久化框架。当然，把里面的一个类纳入测试框架仍然还是可行的，但如果时间都花在跟那些领域类纠缠上面的话，做正事的时间就被占用掉了。在那种情况下，为了获得一些必要的分离，我们使用了本节所讲的技术。为了保护应有的权利，下面给出的例子作了必要的修改。

在一个音乐合成工具中，一条音轨包含了多个音乐事件序列。而我们则需要找出每个序列中的“死时间[1]”，这样才能往这些地方加进一些小的重复音乐模式。我们需要一个叫做`bool Sequence::hasGapFor(Sequence& pattern) const`的方法。该方法的返回值表示一段音乐模式能否放进一个序列。

理想情况下，该方法应该放在一个叫做`Sequence`的类上，但不幸的是`Sequence`类正是那种想要把整个世界都吸进测试用具中去的“黑洞”类。在开始编写这个方法之前，我们得先想想怎么给它编写测试。幸运的是，序列(`Sequence`)对象的内部表示可以简化，这就使得我们编写测试成为了可能。每个序列对象都包含一组事件。但仍然不幸的是，事件类的依赖情况并不比序列类好：它们都有相当严重的依赖，都会给构建带来麻烦。然而，再一次，幸运的是，要计算一段模式能否放进一个序列，我们其实只需要每个事件的持续时间。于是我们可以编写另一个基于整型数来进行计算的方法。有了该方法之后，便可以编写`hasGapFor`并让它将实质工作委托给基于整型计算的那个方法来完成。

让我们从编写第一个方法开始，下面是对它的测试：

```
TEST(hasGapFor, Sequence)
{
```

1. dead time，原为色谱分析术语。——译者注

```
    vector<unsigned int> baseSequence;
    baseSequence.push_back(1);
    baseSequence.push_back(0);
    baseSequence.push_back(0);

    vector<unsigned int> pattern;
    pattern.push_back(1);
    pattern.push_back(2);

    CHECK(SequenceHasGapFor(baseSequence, pattern));
}
```

385

SequenceHasGapFor是一个自由函数；它并不属于任何类，但关键的一点是，它所操作的对象是一个基于基本类型的序列表示（这里是unsigned int）。如果我们能编写出SequenceHasGapFor，就能进而在Sequence类上添加一个非常简单的hasGapFor函数，它只要将实质性的工作全部委托给SequenceHasGapFor来完成就行了：

```
bool Sequence::hasGapFor(Sequence& pattern) const
{
    vector<unsigned int> baseRepresentation
            = getDurationsCopy();

    vector<unsigned int> patternRepresentation
            = pattern.getDurationsCopy();

    return SequenceHasGapFor(baseRepresentation,
                             patternRepresentation);
}
```

该函数需要借助另一个函数来获取持续时间的数组，我们来编写这个函数：

```
vector<unsigned int> Sequence::getDurationsCopy() const
{
    vector<unsigned int> result;
    for (vector<Event>::iterator it = events.begin();
            it != events.end(); ++it) {
        result.push_back(it->duration);
    }
    return result;
}
```

到目前为止，已经可以实现我们想要的特性了，但做法是非常丑陋的。下面就列出其中的问题：

(1) 暴露了Sequence类的内部表示。

(2) 令Sequence类的实现更难理解，因为我们将其实现的一部分推到了一个自由函数中。

(3) 写了一些没有测试覆盖的代码（实际上我们是没法给getDurationsCopy()编写测试）。

(4) 重复数据。

(5) 拖延了问题。我们并未解开领域类与基础架构之间的依赖（这一点会给后面的工作带来很大的影响）。

386

尽管有这许多缺点，我们终究还是得以把一个有测试覆盖的特性加进去了。我并不喜欢这类重构，但如果已经没有其他选择的话，也只能这么办了。通常它是新生类（54页）手法的一个不错的准备。你可以设想将SequenceHasGapFor包覆在一个GapFinder类中的情形。

> 朴素化参数（Primitivize Parameter）手法对代码的状况并无多大改善。总的来说更好的办法是将新代码加到原类上，或使用新生类手法来建立新抽象，充当后续工作的基础。我使用朴素化参数手法的唯一一次是当我觉得有信心在后面能腾出时间来把我的类纳入测试时；到那时候，便可以把我的方法放到这个类上了。

步骤

朴素化参数手法的步骤如下：

(1) 编写一个自由函数来实现你想要对目标类做的事情。同时建立一个中间表示，以便你的自由函数进行处理。

(2) 往目标类上添加一个函数来构造这一中间表示，并将实际任务转发给上一步创建的那个自由函数。

387

25.17　特性提升

有时候，比如说，你要对付一个类上面的一簇方法，而阻止你在测试用具中实例化这个类的依赖却又是跟这簇方法毫无瓜葛的。这里所谓“毫无瓜葛”是指这些方法既没有直接也没有间接引用或触碰到那些问题依赖。当然，这时你可以通过重复采用暴露静态方法（275页）或分解出方法对象（261页）来“解决”问题，但那样做未必就是解决这个问题的最直接办法。

面对这种情况，你可以将这簇方法（即所说的“特性”）提取出来，提升到一个抽象基类中。然后再对这个抽象基类进行子类化，并在测试中创建这个子类的实例。例如：

```
public class Scheduler
{
    private List items;

    public void updateScheduleItem(ScheduleItem item)
            throws SchedulingException {
        try {
            validate(item);
        }
        catch (ConflictException e) {
            throw new SchedulingException(e);
        }
        ...
    }

    private void validate(ScheduleItem item)
            throws ConflictException {
        // make calls to a database
```

```
        ...
    }

    public int getDeadtime() {
        int result = 0;
        for (Iterator it = items.iterator(); it.hasNext(); ) {
            ScheduleItem item = (ScheduleItem)it.next();
            if (item.getType() != ScheduleItem.TRANSIENT
                    && notShared(item)) {
                result += item.getSetupTime() + clockTime();
            }
            if (item.getType() != ScheduleItem.TRANSIENT) {
                result += item.finishingTime();
            }
            else {
                result += getStandardFinish(item);
            }
        }
        return result;
    }
}
```

388

假设我们要修改getDeadTime，但并不关心updateScheduleItem。如果不用应付对数据库的依赖的话，情况会好很多。为此我们可以尝试使用*暴露静态方法*，但getDeadTime里面用到了Scheduler上的许多实例变量。也可以试试*分解出方法对象*，但这又是个很小很小的方法，似乎不值得这么做，况且它对于其他实例变量和方法的依赖会使得我们被许多根本不想看到的麻烦缠住——毕竟我们只是想把这么个小小的方法放入测试而已。

另一个做法就是将问题方法提升到一个基类中。我们可以将问题依赖留在原来的类中，免得它们阻挠测试。特性提升之后的Scheduler类大致像这样：

```
public class Scheduler extends SchedulingServices
{
    public void updateScheduleItem(ScheduleItem item)
            throws SchedulingException {
        ...
    }

    private void validate(ScheduleItem item)
            throws ConflictException {
        // make calls to the database
        ...
    }
    ...
}
```

我们已经将getDeadtime（我们想要测试的特性）以及它所用到的所有特性都提升到了一个抽象类SchedulingServices中了：

```
public abstract class SchedulingServices
{
    protected List items;
```

```
    protected boolean notShared(ScheduleItem item) {
        ...
    }

    protected  int getClockTime() {
        ...
    }

    protected int getStandardFinish(ScheduleItem item) {
        ...
    }

    public int getDeadtime() {
        int result = 0;
        for (Iterator it = items.iterator(); it.hasNext(); ) {
            ScheduleItem item = (ScheduleItem)it.next();
            if (item.getType() != ScheduleItem.TRANSIENT
                    && notShared(item)) {
                result += item.getSetupTime() + clockTime();
            }
            if (item.getType() != ScheduleItem.TRANSIENT) {
                result += item.finishingTime();
            }
            else {
                result += getStandardFinish(item);
            }
        }
        return result;
    }
    ...
}
```

389

于是现在便可以从SchedulingServices派生出一个测试子类来，这样我们便可以从容地在测试用具中访问这些方法了：

```
public class TestingSchedulingServices extends SchedulingServices
{
    public TestingSchedulingServices() {
    }

    public void addItem(ScheduleItem item) {
        items.add(item);
    }
}

import junit.framework.*;

class SchedulingServicesTest extends TestCase
{
    public void testGetDeadTime() {
        TestingSchedulingServices services
            = new TestingSchedulingServices();
        services.addItem(new ScheduleItem("a",
```

```
                        10, 20, ScheduleItem.BASIC));
            assertEquals(2, services.getDeadtime());
        }
        ...
    }
```

回顾一下，我们做了什么呢？首先将想要测试的方法提升到一个抽象基类中，然后创建该抽象基类的一个具体派生类，后者便是我们可以用在测试中的类。那么，这么做是不是件好事呢？从 390
设计的角度来说，这种做法还不算理想。我们令一组特性跨越了两个类，这么做的原因只不过是为了容易测试。如果这两个类的特性之间的关系并不密切的话，这种特性跨越就可能会带来混乱。而这儿的情况正是如此：我们有一个类叫Scheduler，其职责是更新计划项目，还有一个类叫SchedulingServices，但这个类的职责就广了，包括获取计划项目的默认时间、计算死时间等。另一种好一点的重构方式是让Scheduler委托某个validator对象，将访问数据库的任务放在后者身上，但如果这一步目前看上去还太危险，或者说还有其他糟糕的依赖存在的话，那么特性提升是个不错的开始。使用特性提升手法时，你如果实施签名保持（249页），并且依靠编译器（251页）的话，风险就会小得多。而当测试安置到位之后，我们可以再去考虑是否转向委托的模式。

步骤

特性提升（Pull up Feature）手法的步骤如下：

(1) 找出你想要提升到抽象基类中去的方法。

(2) 为它们创建一个抽象基类。

(3) 将这些方法转移到该抽象基类中，再编译。

(4) 编译错误会告诉你这些方法引用到的其他实例成员，将它们也转移到基类中。记住这么做的时候要保持签名，以尽量减少出错的机会。

(5) 当两个类都成功编译之后，为那个抽象基类创建一个测试子类，并往其中添加你觉得需要用于设置测试环境的方法。

> 你可能会想："干嘛非要让那个基类成为抽象的呢？"实际上，是为了让代码更易理解。设想你看到一块代码基，那么肯定会期望看到每个具体类都被用到了，如果有一个具体类没有被任何代码直接用到（实例化）的话，你肯定会感到困惑，因为在你看来它们无异于"死代码"。 391

25.18　依赖下推

有些类里面的问题依赖并不多。如果这些依赖被包含在少数几个方法中的话，你可以采用子类化并重写（314页）手法来将它们解决掉。但如果依赖猖獗，则这条路可能就行不通了，这时你可能就需要动用接口提取技术，重复运用接口提取来解除对某些特定类型的依赖。依赖下推（Push Down Dependency）则是另一个选择。该手法能够将目标类其他部分的问题依赖分离出来，使你能够更容易地在测试用具中将它实例化。

在使用依赖下推技术时，首先把目标类设为抽象类，然后创建一个它的子类，后者便是你的新的产品类了。接着将所有的问题依赖都“下推”到这个子类中。到这一步，你便可以通过子类化原类来让它的有关方法接受测试了。下面是一个C++的例子：

```
class OffMarketTradeValidator : public TradeValidator
{
private:
    Trade& trade;.
    bool flag;

    void showMessage() {
        int status  = AfxMessageBox(makeMessage(),
                                    MB_ABORTRETRYIGNORE);
        if (status == IDRETRY) {
            SubmitDialog dlg(this,
                    "Press okay if this is a valid trade");
            dlg.DoModal();
            if (dlg.wasSubmitted()) {
                g_dispatcher.undoLastSubmission();
                flag = true;
            }
        }
        else
        if (status == IDABORT) {
            flag = false;
        }
    }

public:
    OffMarketTradeValidator(Trade& trade)
    : trade(trade), flag(false)
    {}

392 bool isValid() const {
        if (inRange(trade.getDate())
                && validDestination(trade.destination)
                && inHours(trade) {
            flag = true;
        }
        showMessage();
        return flag;
    }
    ...
};
```

对于上面这个类，如果想要修改它里面的isValid()的验证逻辑的话就会遇到麻烦了，我们可不希望将UI相关的函数和类牵扯到测试用具中来。这时依赖下推手法便有用武之地了。

对上面的类作依赖下推之后的情形如下：

```
class OffMarketTradeValidator : public TradeValidator
{
protected:
```

```
    Trade& trade;
    bool flag;
    virtual void showMessage() = 0;

public:
    OffMarketTradeValidator(Trade& trade)
    : trade(trade), flag(false) {}

    bool isValid() const {
        if (inRange(trade.getDate())
                && validDestination(trade.destination)
                && inHours(trade) {
            flag = true;
        }
        showMessage();
        return flag;
    }
    ...
};

class WindowsOffMarketTradeValidator
        : public OffMarketTradeValidator
{
protected:
    virtual void showMessage() {
        int status  = AfxMessageBox(makeMessage(),
                        MB_ABORTRETRYIGNORE);
        if (status == IDRETRY) {
            SubmitDialog dlg(this,
                "Press okay if this is a valid trade");
            dlg.DoModal();
            if (dlg.wasSubmitted()) {
                g_dispatcher.undoLastSubmission();
                flag = true;
            }
        }
        else
        if (status == IDABORT) {
            flag = false;
        }
    }
    ...
};
```

393

一旦UI相关的工作被下推到了一个新的子类（WindowsOffMarketValidator）中，我们便可以创建另一个测试用的子类了，后者只需实现一个空的showMessage()方法即可：

```
class TestingOffMarketTradeValidator
        : public OffMarketTradeValidator
{
protected:
    virtual void  showMessage() {}
};
```

如此一来我们便有了一个可测试但同时又不依赖于任何UI相关事物的类。那么，在这个例子中使用继承是否为理想的方案呢？不是的，但它最大的好处就是能帮助我们将目标类的一部分逻辑纳入测试。而一旦有了对OffMarketTradeValidator的测试，便可以开始清理showMessage()里面的重试逻辑，并将其从WindowsOffMarketTradeValidator提升到基类中来。然后，当WindowsOffMarketTradeValidator最终被掏空得只剩UI相关的调用时，我们便可以转向委托式的设计了，即将UI相关调用委托给一个UI依赖的新类。

步骤

依赖下推的步骤如下：

(1) 在测试用具中构建目标类。

(2) 找出哪些依赖是导致构建问题的依赖。

(3) 创建目标类的子类，子类的名字须反映上一步找出的依赖的特征[1]。

(4) 将目标类中的依赖变量和方法全部复制到新建的子类中，注意保持签名；将目标类中的
394 相应方法设为受保护及抽象的；将目标类设为抽象的。

(5) 创建目标类的一个测试子类，修改你的测试，实例化该测试子类。
395

(6) 创建测试来验证你的确能实例化这个新的测试子类。

25.19　换函数为函数指针

在过程式语言中解依赖可没有在面向对象语言中那么多选择。比如你没法使用*封装全局引用*（268页）或是*子类化并重写方法*（314页）。所有这些面向对象的解依赖手法都行不通。当然，你还是可以使用*连接替换*（296页）或*定义补全*（266页），但它们对于解开一些小的依赖又显得有点杀鸡用牛刀了。对于支持函数指针的语言，*换函数为函数指针*（Replace Function with Function Pointer）手法是一个可选的方案。众所周知的支持函数指针的语言就属C了。

不同的团队对函数指针有着不同的看法。有些团队认为函数指针极度不安全，因为指针的内容可能会被破坏，从而导致被调用的是一块随机内存。而有些团队则把它当成有用的工具，当然，小心使用是前提。实际上，如果你站在后者的阵营，便可以借助于这一工具来解开用其他办法很难或者无法解开的依赖。

首先让我们来看一看函数指针的自然用法。下面这个例子是C写的，其中我们声明了一些函数指针并通过它们来发起调用：

```
struct base_operations
{
    double (*project)(double,double);
    double (*maximize)(double,double);
};

double default_projection(double first, double second) {
```

1. 比如是对windows UI的依赖，那么便可以加上“Windows”前缀。——译者注

```
    return second;
}

double maximize(double first, double second) {
    return first + second;
}

void init_ops(struct base_operations *operations) {
    operations->project = default_projection;
    operations->maximize = default_maximize;
}

void run_tesselation(struct node *base,
                     struct base_operations *operations) {
    double value = operations->project(base.first, base.second);
    ...
}
```

396

函数指针可以实现一些非常初级的基于对象的编程，然而在解依赖方面它们又有多大用处呢？考虑下面这个场景。

有一个网络应用，包信息被保存在一个在线数据库中。你通过如下的调用来与该数据库交互：

```
void db_store(
    struct receive_record *record,
    struct time_stamp receive_time);
struct receive_record * db_retrieve(time_stamp search_time);
```

我们可以使用连接替换（296页）来将这些函数连接到新的函数体，但连接替换有时会给构建过程带来较大麻烦。比如我们可能会需要将问题函数从它所在的库中分离出来以便替换它们。更重要的是，在产品代码中根本没法使用连接替换所获得的接缝来调节行为。如果你想要将代码纳入测试同时又能提供产品实现的灵活性（比如切换你的代码所访问的数据库的类型）的话，*换函数为函数指针*是个好主意。下面就是具体的步骤。

首先我们找出想要替换为函数指针的函数：

```
// db.h
void db_store(struct receive_record *record,
              struct time_stamp receive_time);
```

声明一个跟它同名的函数指针：

```
// db.h
void db_store(struct receive_record *record,
              struct time_stamp receive_time);

void (*db_store)(struct receive_record *record,
                 struct time_stamp receive_time);
```

重命名原函数：

```
// db.h
void db_store_production(struct receive_record *record,
                         struct time_stamp receive_time);
```

```
void (*db_store)(struct receive_record *record,
                 struct time_stamp receive_time);
```

在一个C源文件中初始化这个函数指针：

```
// main.c
extern void db_store_production(
                struct receive_record *record,
                struct time_stamp receive_time);

void initializeEnvironment() {
    db_store = db_store_production;
    ...
}

int main(int ac, char **av) {
    initializeEnvironment();
    ...
}
```

397

找到db_store函数的定义，将其重命名为db_store_production：

```
// db.c
void db_store_production(
        struct receive_record *record,
        struct time_stamp receive_time) {
    ...
}
```

现在便可以编译和测试了。

有了这个函数指针，测试文件便可以替换进，从而达到感知或分离的目的。

换函数为函数指针手法是解依赖的好办法。它的好处之一便是发生在编译期（而非连接期），因此对你的构建系统几乎没有影响。然而，如果你是在C里面运用这项技术的，那么请考虑转移到C++，以便利用C++提供的各种各样的接缝。在本书写作时，许多C编译器都提供了编译开关来允许你进行C/C++混合编译。利用这个性能你便可以将C项目逐步往C++迁移，每次只需对你关心的那些文件进行解依赖。

步骤

换函数为函数指针手法的步骤如下：

(1) 找到你想要替换的函数的声明。

(2) 在每个找到的函数之前创建一个同名的函数指针。

(3) 重命名原函数的声明，以避免跟刚才声明的函数指针重名。

(4) 在一个C文件中初始化这些函数指针，将它们指向相应的函数。

398

(5) 构建，通过构建错误来找出原函数的函数体。将它们改为新的函数名。

25.20　以获取方法替换全局引用

当你想单独对付某块代码时，可能常常会发现全局变量挡在你的路上。关于全局变量的害处这里就不再次说了，因为我曾经在*引入静态设置方法*（292页）一节作了相当完整的叙述。

要想解开一个类里面对全局变量的依赖，方法之一是在该类中为每个相应的全局变量引入一个获取方法。有了这些获取方法，便可以通过*子类化并重写方法*（314页）来令它们返回测试用对象了。不过也有些情况下你可能会需要动用*接口提取*（285页）。

下面是一个Java的例子：

```
public class RegisterSale
{
    public void addItem(Barcode code) {
        Item newItem =
            Inventory.getInventory().itemForBarcode(code);
        items.add(newItem);
    }
    ...
}
```

在上面的代码中，Inventory类是被作为一个全局变量来访问的。“什么？”你可能会嚷起来，“这不明明是个静态方法调用吗？”没错。但从我们的意图（测试）来说，它其实就相当于一个全局变量。在Java中，该类本身就是一个全局对象，而且似乎它还需要引用一些状态才能完成它的工作（基于给定的barcode（条形码）返回相应的item（商品））。那么，我们能否利用*以获取方法替换全局引用*（Replace Global Reference with Getter）手法来对付这个问题呢？试试看吧。

首先编写获取方法。注意，我们将该方法设为受保护的，这样才能在测试子类中重写它：

```
public class RegisterSale
{
    public void addItem(Barcode code) {
        Item newItem = Inventory.getInventory().itemForBarcode(code);
        items.add(newItem);
    }

    protected Inventory getInventory() {
        return Inventory.getInventory();
    }
    ...
}
```

然后将每一处对全局对象的引用替换为对该获取方法的调用：

399

```
public class  RegisterSale
{
    public void addItem(Barcode code) {
        Item newItem = getInventory().itemForBarcode(code);
        items.add(newItem);
    }

    protected Inventory getInventory() {
```

```
            return Inventory.getInventory();
        }
        ...
    }
```

接着我们创建Inventory的测试用伪类。由于Inventory是个单件类，因此需要先降低其构造函数的访问权限为受保护的，然后再像下面这样从它派生出一个FakeInventory，并在里面放置我们需要的逻辑：

```
public class FakeInventory extends Inventory
{
    public Item itemForBarcode(Barcode code) {
        ...
    }
    ...
}
```

最后编写RegisterSale的测试用子类：

```
class TestingRegisterSale extends RegisterSale
{
    Inventory inventory = new FakeInventory();

    protected Inventory getInventory() {
        return inventory;
    }
}
```

步骤

以获取方法替换全局引用的步骤如下：

(1) 找出你想要替换的全局引用。

(2) 给它编写一个相应的获取方法。确保该获取方法的访问权限允许你在派生类中重写它。

(3) 将对全局对象的引用替换为对该获取方法的调用。

400 (4) 创建测试子类并重写获取方法。

25.21　子类化并重写方法

子类化并重写方法（Subclass and Override Method）是面向对象程序中解依赖的核心技术。实际上本章所讲的许多其他的解依赖手法都是该手法的变种。

该手法的核心理念就是你可以在测试环境下利用继承来将并不关心的行为架空或访问到你所关心的行为。

让我们来看一看一个小型应用里面的一个方法：

```
class MessageForwarder
{
    private Message createForwardMessage(Session session,
                                          Message message)
```

```
                throws MessagingException, IOException {
        MimeMessage forward = new MimeMessage (session);
        forward.setFrom (getFromAddress (message));
        forward.setReplyTo (
                new Address [] {
                    new InternetAddress (listAddress) });
        forward.addRecipients (Message.RecipientType.TO,
                               listAddress);
        forward.addRecipients (Message.RecipientType.BCC,
                               getMailListAddresses ());
        forward.setSubject (
                transformedSubject (message.getSubject ()));
        forward.setSentDate (message.getSentDate ());
        forward.addHeader (LOOP_HEADER, listAddress);
        buildForwardContent(message, forward);

        return forward;
    }
    ...
}
```

MessageForwarder类上还有其他一些方法没有显示出来。其中某个公有的方法调用了上面给出的这个私有方法createForwardMessage来创建一条新的消息。现在，假设我们不想在测试的时候依赖于MimeMessage类。因为MimeMessage用到了一个叫做session（会话）的变量，而在测试的时候是没法构造出一个真正的session出来的。如果我们想要将对MimeMessage的依赖分离出来，便可以将createForwardMessage设为受保护的，并在一个测试用子类中重写它，如下：

```
class TestingMessageForwarder extends MessageForwarder
{
    protected Message createForwardMessage(Session session,
                                           Message message) {
        Message forward = new FakeMessage(message);
        return forward;
    }
    ...
}
```

401

在这个新建的测试子类中，我们可以做想做的事情，得到想要的分离和感知。在这个特定的例子中，我们可以完全架空createForwardMessage的大部分行为，但由于在测试的时候并不需要用到createForwardMessage原来的那些行为，所以一切都没问题。

在产品代码中，我们实例化的是MessageForwarders；而在测试代码中，实例化的则是TestingMessageForwarders。你看，我们以最少的修改换来了所需要的分离。实际上只不过是将createForwardMessage的访问权限从私有换成了受保护的。

通常，一个类当中的功能分解的好坏决定了你使用该手法来分离依赖时的难易程度。好的情况下，你想要分离出的依赖会被隔离在一个小小的方法当中。而在糟糕的情况下，你可能就需要重写一个较大的方法才能分离出依赖了。

子类化并重写方法是个强大的手段，但用的时候也需小心。比如在前面的例子中我可以返回一个没有主题、发件人地址等信息的空消息对象，但这么做是有特定前提的，比如我正在测试的是能否将一个消息从系统中的一个地方传到另一个地方，而并不关心消息的具体内容和地址，这时候这么做才是安全的。

对我来说编程活动大多数时候是可视化的。我在工作的时候脑袋里会设想各种各样的情景，这有助于我在不同的方案之间进行取舍。可惜的是我设想的这些图景没有一个是UML图，不过它们还是帮了我不少忙。

比如我经常设想的一种场景就是我所谓的“叠纸视图”。我看着一个方法，然后在脑海里设想所有可用于将其语句和表达式分组的方式。对于一个方法中我可以确定出来的几乎任何细小的代码片段，我猜想能否将其提取到一个方法中，进而在测试中将其替换为另一个方法。这就好像我将一层半透明的纸放在代码之上，在这层半透明的纸上我可以放置用于替换目标代码片段的代码。这叠纸就是我的测试对象，而从上往下看到的方法便是会在测试中被执行到的方法。图25-6
402 展示了我所设想的场景。

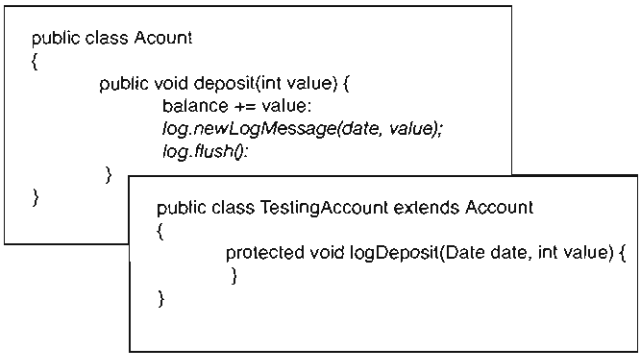


图25-6　TestingAccount 浮于Account之上

叠纸视图法能够帮助我看到什么是可能的，但当我真正开始使用子类化并重写方法手法时，仍然还是尽量去重写既有的方法。毕竟我们的目的是将测试安置到位，而在没有测试的情况下提取方法常常是危险的。

步骤

子类化并重写方法的步骤如下：

(1) 找出你想要分离出来的依赖，或者想要进行感知的地点。找出尽量少的一组方法来完成你的目标。

(2) 确定了重写哪些方法之后，还得确保它们都是可重写的。这一步根据语言的不同有所不同。C++中首先要将它们设为虚函数。Java中它们必须是非final方法。而在许多.NET语言中，你还得明确地将这些方法设为可重写的。

(3) 在某些语言中，你需要调整这些方法的访问权限才能在子类中重写它们。比如在Java和

C#中，必须至少是受保护的方法才能被子类中的方法重写。而在C++中私有虚函数仍然是可以在子类中重写的。

(4) 创建一个子类并在其中重写这些方法。确保你的确能够在测试用例中构建该类。 403

25.22　替换实例变量

构造函数中的对象创建可能会带来依赖问题，尤其是当测试很难依赖这些对象的时候。大多数情况下我们可以使用*提取并重写工厂方法*（276页）手法来对付这个问题。但对于那些不支持在构造函数中调用虚函数的语言，则必须另觅他法。而办法之一便是本节所要讲的替换实例变量（Supersede Instance Variable）。

下面这个例子演示了C++中的虚函数调用问题：

```
class Pager
{
public:
    Pager() {
        reset();
        formConnection();
    }

    virtual void formConnection() {
        assert(state == READY);
        // nasty code that talks to hardware here
        ...
    }

    void sendMessage(const std::string& address,
                     const std::string& message) {
        formConnection();
        ...
    }
    ...
};
```

我们发现Pager的构造函数中调用到了formConnection方法。本来，把工作委托给其他函数来完成是无可非议的，但这儿的代码有点令人误解。由于formConnection是个虚函数，所以人们可能会想当然的以为只要对它进行*子类化并重写方法*（314页）就够了。但是别着急，没有调查就没有发言权，我们来简单地验证一下：

```
class TestingPager : public Pager
{
public:
    virtual void formConnection() {
    }
};

TEST(messaging,Pager)
{
    TestingPager pager;
    pager.sendMessage("5551212",
```

404

```
                    "Hey, wanna go to a party? XXXOOO");
    LONGS_EQUAL(OKAY, pager.getStatus());
}
```

在C++中重写虚函数时，基类相应的行为会被派生类中的行为替换，这一点跟我们预期的一样，然而有一点例外，就是当你在构造函数中调用一个虚函数时，调用并不会被分发到派生类中的相应虚函数上去。在本例中，当sendMessage被调用时，TestingPager::formConnection固然会被调用起来，这很好，因为我们并不想发送这条搞怪的消息给信息操作员，然而遗憾的是结果并不如你所想。这个TestingPager对象在被构造起来的时候，基类Pager的构造函数被调用，后者调用了formConnection，而又由于在构造函数中虚函数机制是被禁止的，因此那一次的formConnection调用被决议到了Pager::formConnection上！

C++之所以有这个规则是因为在构造函数中允许虚函数调用可能会导致危险。设想下面这种情况：

```
class A
{
public:
    A() {
        someMethod();
    }

    virtual void someMethod() {
    }
};

class B : public A
{
    C *c;
public:

    B() {
        c = new C;
    }

    virtual void someMethod() {
        c.doSomething();
    }
};
```

B::someMethod重写了A::someMethod。当一个B对象被构造的时候，先是A的构造函数被调用起来（这时B的构造函数还没有进入），而A的构造函数调用了someMethod，如果这时允许虚函数转发机制，也就是说让这个someMethod调用转发到B中去的话，“c.doSomething();”语句就会被试图执行，然而问题是，既然还没轮到B的构造函数，那么就是说c根本就还没被初始化，结果可想而知。

405

这便是C++在构造函数中禁止虚函数转发机制的原因。一些其他语言在这个问题上则要放松一些，比如Java中允许这么做，但我不建议你在产品代码中这么干。

然而，C++的这个保护机制却阻碍了我们替换构造函数中的行为。所幸的是还有一些替代方

案。如果你想要替换的对象并没有在构造函数中被使用（而只是创建的话），就可以采用提取并重写获取方法来解开依赖。而如果构造函数中使用了该对象并且你需要确保在另一个方法被调用之前将该对象替换掉的话，就可以采用替换实例变量手法了。例如：

```
BlendingPen::BlendingPen()
{
    setName("BlendingPen");
    m_param = ParameterFactory::createParameter(
                    "cm", "Fade", "Aspect Alter");
    m_param->addChoice("blend");
    m_param->addChoice("add");
    m_param->addChoice("filter");

    setParamByName("cm", "blend");
}
```

BlendingPen的构造函数通过一个工厂来创建Parameter对象。我们可以使用引入静态设置方法（202页）手法来控制该工厂产出的对象，但这么改动的话就太具侵入性了。如果不介意往BlendingPen上添加一个方法的话，我们便可以替换掉构造函数中创建出来的那个Parameter对象，为此引入一个supersedeParameter方法：

```
void BlendingPen::supersedeParameter(Parameter *newParameter)
{
    delete m_param;

    m_param = newParameter;
}
```

在测试时，我们可以根据需要创建BlendingPen对象，并在需要放入感知对象的时候调用它的supersedeParameter方法。

从表面上来说，替换实例变量看起来是个挺糟糕的放置感知变量的手法，但在C++中，当参数化构造函数（297页）手法由于构造函数中纠缠的逻辑而变得难以使用时，替换实例变量就成了最好的选择。不过，在允许构造函数中调用虚函数的语言中，提取并重写工厂方法（276页）通常是更好的选择。

406

一般而言，提供设置方法来允许外界修改被用到的子对象属于不良实践。这些设置方法允许客户代码彻底改变一个对象在其生命周期当中的行为。在旁人可以调用这些设置方法时，你就必须得了解目标对象的历史状态方能知道对它的方法调用会带来什么后果。而当没有设置方法时，代码便更易于理解。

使用“*supersede*”作为这类方法的方法名前缀的一个好处就是这个单词比较奇异且不常见，于是如果你担心别人会在产品代码中使用这个方法的话，只需搜索一下“*supersede*”就能知道结果了。

步骤

替换实例变量的步骤如下：

(1) 找出你想要替换的实例变量。

(2) 创建一个名为supersedeXXX的方法，其中XXX是你想要替换的变量的名字。

(3) 在该方法中销毁原先被创建出来的那个对象，换入你新建出来的对象。如果持有该对象的实例成员是一个引用，则需要确保该类中没有其他成员引用了原先创建出来的那个对象。如果有的话，你可能就需要在supersedeXXX方法里面多做一点工作，来确保能够安全地换入你的新对象并且确保达到正确的效果。

407

25.23　模板重定义

本章提到的许多解依赖技术都依赖于面向对象的核心机制，如接口以及实现继承。而有些新的语言特性则提供了另外的选择。例如，如果你的语言支持泛型以及类型别名，则可以使用叫做模板重定义（Template Redefinition）的手法来解依赖。下面是一个以C++给出的例子：

```
// AsyncReceptionPort.h

class AsyncReceptionPort
{
private:
    CSocket m_socket;
    Packet m_packet;
    int m_segmentSize;
    ...

public:
        AsyncReceptionPort();
    void Run();
    ...
};


// AsynchReceptionPort.cpp

void AsyncReceptionPort::Run() {
    for(int n = 0; n < m_segmentSize; ++n) {
        int bufferSize = m_bufferMax;
        if (n = m_segmentSize - 1)
            bufferSize = m_remainingSize;
        m_socket.receive(m_receiveBuffer, bufferSize);
        m_packet.mark();
        m_packet.append(m_receiveBuffer,bufferSize);
        m_packet.pack();
    }
    m_packet.finalize();
}
```

对于以上代码，如果我们想修改run()函数中的逻辑，就会发现要想在测试用具中执行该方法，就必需通过一个套接字发送东西。在C++中我们可以通过把AsyncReceptionPort做成一个类模板来完全避免这个问题。下面是代码修改之后的样子：

408

```
// AsynchReceptionPort.h

template<typename SOCKET> class AsyncReceptionPortImpl
{
private:
    SOCKET m_socket;
    Packet m_packet;
    int m_segmentSize;
    ...

public:
    AsyncReceptionPortImpl();
    void Run();
    ...
};

template<typename SOCKET>
void AsyncReceptionPortImpl<SOCKET>::Run() {
    for(int n = 0; n < m_segmentSize; ++n) {
        int bufferSize = m_bufferMax;
        if (n = m_segmentSize - 1)
            bufferSize = m_remainingSize;
        m_socket.receive(m_receiveBuffer, bufferSize);
        m_packet.mark();
        m_packet.append(m_receiveBuffer,bufferSize);
        m_packet.pack();
    }
    m_packet.finalize();
}
typedef AsyncReceptionPortImpl<CSocket> AsyncReceptionPort;
```

有了这一步作铺垫，便可以在测试文件中用一个FakeSocket来实例化该类模板了：

```
// TestAsynchReceptionPort.cpp

#include "AsyncReceptionPort.h"

class FakeSocket
{
public:
    void receive(char *, int size) { ... }
};

TEST(Run,AsyncReceptionPort)
{
    AsyncReceptionPortImpl<FakeSocket> port;
    ...
}
```

409

该技术最漂亮之处就在于我们可以通过一个typedef来避免在代码基中到处修改对该类的使用。如果没有typedef的话，就需要将每处对AsyncReceptionPort的使用换成AsyncReceptionPort<CSocket>。这就意味着大量无聊的工作，但难倒是不难，我们可以依靠编译器（251页）来确保修改了每一处地方。在支持泛型但不支持类型别名机制（如typedef）的语言中，你只能依靠编译器。

在C++中你甚至可以利用该技术来替换方法的定义，只不过这么做就有点不够优雅了。C++的语言规则要求你必须提供一个模板参数，所以可以选择一个成员变量并将其类型泛化为模板参数或引入一个新的成员变量以便能够基于某个类型来参数化你的类[1]——但我非到万不得已是不会采取这种做法的。我会先非常谨慎地考察是否能使用基于继承的技术。

C++中的模板重定义手法有一个主要的缺点，即当你参数化一个类之后，它的实现代码就必须转移到头文件中来。这会增加系统中的依赖。每次修改类模板的代码之后，使用该类的代码都必须重编译。

一般来说我仍然倾向于采用基于继承的手法在C++中解依赖。然而如果想要解开的依赖本就处于模板代码中的话，该手法就可以用了，例如：

```
template<typename ArcContact> class CollaborationManager
{
    ...
    ContactManager<ArcContact>  m_contactManager;
    ...
};
```

这儿，如果我们想要解开对m_contactManager的类型的依赖，考虑到在这里模板的使用方式使得接口提取比较困难。我们可以换种方式来参数化CollaborationManager，问题就迎刃而解了：

```
template<typename ArcContactManager> class CollaborationManager
{
    ...
    ArcContactManager  m_contactManager;
    ...
};
```

410

步骤

以下是在C++中运用模板重定义手法的步骤。在其他支持泛型的语言中步骤或许有所不同，但原则一样：

(1) 在待测试类中找出你想要替换的特性。

(2) 将该类做成一个类模板，根据你想要替换的变量对它进行参数化，将方法体转移到头文

1. 这里作者的叙述似乎有问题，实际上在C++中一个类模板的模板参数并不一定要在该类模板里面使用到。——译者注

件中[1]。

(3) 给该类模板另起一个名字。可以将原类名后面加上“Impl”。

(4) 在类模板定义之后加上一行typedef，如：typedef XXXImpl<T> XXX;（其中XXX是原类名，T为参数化类型的具体类型，如CSocket）。

(5) 在测试文件中包含该类模板的定义，用新的测试用类型来实例化，如：XXXImpl<TestT>。 411

25.24　文本重定义

一些新的解释型语言提供了非常好的解依赖途径。在被解释的时候，方法可以被及时重定义。比如下面是个Ruby的例子：

```
# Account.rb
class Account
    def report_deposit(value)
        ...
    end

    def deposit(value)
        @balance += value
        report_deposit(value)
    end

    def withdraw(value)
        @balance -= value
    end
end
```

如果我们不想看到report_deposit在测试中被执行，则只需在测试文件中重定义，并将测试代码放在新的定义之后，如下：

```
# AccountTest.rb
require "runit/testcase"
require "Account"

class Account
    def report_deposit(value)
    end
end

# tests start here
class AccountTest < RUNIT::TestCase
    ...
end
```

有一点非常值得注意，即我们并没有重定义整个的Account类，而只是重定义了它的report_deposit方法。Ruby解释器将ruby代码文件的每行都看作可执行语句。而“class

1. C++编译器普遍不支持模板的分离编译。——译者注

Account”语句就相当于打开了Account类的定义，从而新的方法定义可以被添加进去。而“def report_deposit(value)”语句则开始往打开的类中添加新方法。Ruby解释器并不关心该方法
412 是否已有定义体存在，如果有的话就替换掉。

Ruby中的文本重定义（Text Redefinition）也有它的缺点：新方法对旧方法的替换会一直有效，直到程序结束。所以如果你后面忘记了某方法已被重定义了的话就可能会遇到麻烦。

其实我们在C/C++中也可以进行文本重定义——使用预处理[1]。关于这个的一个例子可以参考第4章描述预处理期接缝（29页）一节中的例子。

步骤

在Ruby中使用文本重定义的步骤如下：

(1) 找出你想要替换定义的方法所在的类。

(2) 在测试源文件开头添加一行require引入包含目标类的模块。
413

(3) 在测试源文件的开头给每一个你想要替换的方法提供新的定义。

1. 一般叫文本替换。——译者注

附 录

重 构

重构是改善代码的核心技术。关于重构的标准参考读物是Martin Fowler的著作《重构：改善既有代码的使用》[1]。更多关于在有测试情况下的重构，我建议你去阅读Martin的大作。

本附录的意图是描述一个关键的重构手法：方法提取（Extract Method）。从而给你一些关于在有测试情况下的重构机制的认识。

方法提取

在所有的重构手法中，方法提取大约是最有用的一种了。方法提取手法背后的理念是我们可以系统地将一个大的方法分解为一些小的。这会令代码更易理解。此外我们常常还能在系统的其他部分复用分解出来的代码块并避免逻辑重复。

> 在维护得很糟糕的代码基中，方法会倾向于越“长”越大。人们会不断往既有代码中添加逻辑，于是这些方法就会不断膨胀。并且，随着这一过程的发生，一个方法便会最终承担起两到三个不同的职责。极端情况下一个方法甚至会做十件乃至上百件事情。而方法提取正是这些病症的解药。

当想要提取一个方法时，首要的事情便是要有一组测试。如果你有一组测试能够完全覆盖目标方法，则可以采用下述步骤对它进行方法提取：

(1) 找出你想要提取出来的代码，将其注释掉。

(2) 创建一个新的空方法，给它起一个名字。

(3) 在原方法中调用这个新方法。

(4) 将你想要提取的代码放到新方法中。

(5) 依靠编译器（251页）帮你发现新方法要接受哪些参数以及返回什么值。

(6) 相应调整新方法的声明，声明参数列表和返回类型。

(7) 运行你的测试。

414
~
415

1. 此书英文注释版即将由人民邮电出版社出版。另外，Joshua Kerievsky的《重构与模式》（人民邮电出版社）是重构方面的另一部重要著作，体现了近年来重构领域的成就和发展，并充分阐述了重构与模式的深刻关系。

——编者注

(8) 删掉第一步注释掉的代码。

下面是一个简单的Java例子：

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
        }
        return result;
    }
    ...
}
```

以上代码中的else语句负责计算premium预定的手续费。考虑到系统中还有其他地方也需要用到这块逻辑，所以我们可以将它提取到一个新方法中并在其他地方复用它。

下面是第一步：

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
        }
        return result;
    }
    ...
}
```

416

我们想要调用新方法getPremiumFee，于是创建该方法并将对它的调用加到那个else子句中：

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
            result += getPremiumFee();
        }
```

```
        return result;
    }

    int getPremiumFee() {
    }
    ...
}
```

下一步，将else子句中原来的代码复制到getPremiumFee()中，编译：

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
            result += getPremiumFee();
        }
        return result;
    }

    int getPremiumFee() {
        result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
    }
    ...
}
```

正如预料的那样，编译通不过。因为被复制到getPremiumFee()中的代码使用了名为result和amount的两个变量，而这两个变量在getPremiumFee()中并未声明。由于我们计算的只是结果的一部分，所以可以直接返回计算的结果。此外，amount变量可以通过为getPremiumFee()增加一个相应的参数来获得：

417

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            // result += (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
            result += getPremiumFee(amount);
        }
        return result;
    }

    int getPremiumFee(int amount) {
        return (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
```

```
    }
    ...
}
```

现在便可以运行测试来检验我们的成果了。如果代码通过测试，则就可以将原来注释掉的代码完全删除了：

```
public class Reservation
{
    public int calculateHandlingFee(int amount) {
        int result = 0;

        if (amount < 100) {
            result += getBaseFee(amount);
        }
        else {
            result += getPremiumFee(amount);
        }
        return result;
    }

    int getPremiumFee(int amount) {
        return (amount * PREMIUM_RATE_ADJ) + SURCHARGE;
    }
    ...
}
```

我喜欢先将待提取的代码注释掉而不是直接删掉（你不一定要这么做），这样一来，如果后面的步骤出了错的话就可以容易地还原原来的代码，并重新尝试。

上面的例子只是展示了方法提取的一种方式。在有测试的情况下，方法提取是一个相对简单
418 和安全的操作。而如果你有重构工具的话那就更简单了——只需选择某函数中的一块代码，然后选择一个菜单即可。工具会帮你检查那块代码能否被提取成一个新方法并让你输入新方法的名字。

方法提取是对付遗留代码的核心手段。你可以用它来提取重复代码、分离职责，以及分解长
419 方法。

术 语 表

修改点（change point） 需要往代码中引入修改的点。

特征测试（characterization test） 描述软件某部分的当前行为的测试，当你修改代码时能够用来保持行为。

耦合数（coupling count） 当一个方法被调用时传给它以及从它传出来的值的数目。如果该方法没有返回值，则耦合数就是它的参数数目。否则就是参数数目加1。如果你想要在没有测试的情况下提取出一个小方法的话，计算一下它的耦合数是很有意义的。

影响草图（effect sketch） 一张不大的手画草图，其作用是展示出哪些变量和方法返回值会被某个特定的修改所影响。在你试图寻找合适的测试地点时影响草图可以帮上忙。

伪对象（fake object） 在测试中伪装成一个类的合作者的对象。

特性草图（feature sketch） 一张不大的手画草图，展示了一个类中的方法是如何使用其他实例方法和变量的。在你试图决定如何分解一个大类时特性草图可以帮上忙。

自由函数（free function） 一个不属于任何类的函数。在C和其他过程式语言中，自由函数被简单地称为函数。而在C++中它们被称为非成员函数。Java和C#中没有自由函数。

拦截点（interception point） 可以编写测试来感知某些条件的地点。

连接期接缝（link seam） 在连接期接缝处，你可以通过连接到另一个库来替换行为。在编译型语言中，可替换的东西包括产品库、DLL、程序集或JAR文件。其目的是为了解除依赖，或感知某些在测试期间可能会发生的条件。

仿对象（mock object） 在内部对条件进行断言的伪对象。

对象接缝（object seam） 在对象接缝处你可以通过替换一个对象为另一个对象来“更换”行为。在面向对象语言中，我们通常通过子类化产品代码中的类并重写其方法来实现这一点。

汇点（pinch point） 汇点是影响结构图中的隘口和交通要冲，在汇点编写测试的好处就是，只需针对少数几个方法编写测试，就能达到探测大量其他方法的改动的目的。

差异式编程（programming by difference） 一种使用继承来往面向对象系统中添加特性的编程方式，常常可以用于将一个新特性快速添加进系统中。而编写用来驱动新特性的测试则可以在之后被用于将系统重构至更好的状态。

接缝（seam） 接缝，顾名思义，就是指程序中的一些特殊的点，在这些点上你无需作任何修改就可以达到改动程序行为的目的。例如，对一个对象上的多态函数的调用就是一个接缝，因为你可以通过子类化该对象的类来让该调用具有另一种行为。

测试驱动开发（test-driven development，TDD）　测试驱动开发是一种开发过程，它包括编写失败测试用例，并一次一个地满足它们。在这么做的过程中，你通过重构来使代码尽量保持简单。使用TDD方法编写出来的代码默认就是测试覆盖的。

测试用具（test harness）　支持单元测试的软件。

测试子类（testing subclass）　为了访问被测试类而派生出来的子类。

单元测试（unit test）　单元测试的两个特点：一、运行时间小于十分之一秒；二、足够小，从而当失败的时候能够帮你锁定问题。

索　引

索引中页码为英文原书页码，与本书中页边标注的页码一致。

D

G

H

I

S

U

V

W

X

Working Effectively with Legacy Code
修改代码的艺术

"Michael Feathers 具有我们许多人都没有的洞察力。……他在本书中讲述的技术、模式和工具非常有效……能帮助你力挽狂澜，将逐渐退化的系统转变为渐趋完善。"

——Robert Martin，面向对象技术大师，《敏捷软件开发》作者

"据我所知，这是第一本成功地阐述了如何挽救问题代码的书，实战性极强……我认为它是近十年来最具影响力的计算机图书之一，它的深远影响现在才刚刚开始……"

——Brian Mavrick，著名技术专家，*Everyday Scripting with Ruby* 一书作者

我们都知道，即使是最训练有素的开发团队，也不能保证始终编写出清晰高效的代码。如果不积极地修改、挽救，随着时间流逝，所有软件都会不可避免地渐渐变得复杂、难以理解，最终腐化、变质。因此，理解并修改已经编写好的代码，是每一位程序员每天都要面对的工作，也是开发程序新特性的基础。然而，与开发新代码相比，修改代码更加令人生畏，而且长期以来缺乏文献和资料可供参考。

本书是继《重构》和《重构与模式》之后探讨修改代码技术的又一里程碑式的著作，而且从广度和深度上都超过了前两部经典。书中不仅讲述了面向对象语言（Java、C# 和 C++）代码，也有专章讨论 C 这样的过程式语言。作者将理解、测试和修改代码的原理、技术和最新工具（自动化重构工具、单元测试框架、仿对象、集成测试框架等）与解依赖技术及大量开发和设计优秀代码的原则、最佳实践相结合，许多内容非常深入，而且常常发前人所未发。书中处处体现出作者独到的洞察力，以及多年开发和指导软件项目所积累的丰富经验和深厚功力。通过这部集大成之作，你不仅能掌握最顶尖的修改代码技术，还可以大大提高对代码和软件开发的领悟力。

Michael Feathers 世界级面向对象技术专家，以丰富的软件项目开发经验著称。目前在世界顶尖的软件咨询公司 Object Mentor 从事敏捷方法 / 极限编程、测试驱动开发、重构、面向对象设计、Java、C# 和 C++ 等方面的培训和项目指导。他是著名测试框架 CppUnit 和 FitCpp 的开发者，已经主持了三次面向对象界盛会 OOPSLA 上的 CodeFest 比赛。

刘未鹏 热爱编程技术，长期关注 C++，现在南京大学计算机系攻读硕士学位，译有《Imperfect C++ 中文版》、《Exceptional C++ Style 中文版》（人民邮电出版社出版）。个人 blog：http://blog.csdn.net/pongba。

PRENTICE HALL PTR

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)88593802
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机 / 软件开发 / 程序设计

人民邮电出版社网址 www.ptpress.com.cn

ISBN 978-7-115-16362-2

9 787115 163622 >

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

定价：59.00 元